Przetwarzanie równoległe
projekt 2

Andrzej Gretkowski 127341
(andrzej.gretkwski@student.put.poznan.pl)
Tomasz Supłat 127214
(tomasz.suplat@student.put.poznan.pl)

Maj 2018

Spis treści

1 Wstęp

Celem niniejszego sprawozdania jest porównanie wydajności mnożenia macierzy za pomocą różnych algorytmów, na karcie graficznej.

Treść polecenia:
Badanie prędkości obliczeń w funkcji ilości pracy wątków:

1. *Jeden blok wątków przetwarzania, obliczenia przy wykorzystaniu pamięci globalnej, mnożenie dowolnych tablic o rozmiarach będących wielokrotnością rozmiaru bloku wątków.*
2. *Grid wieloblokowy, jeden wątek oblicza jeden element macierzy wynikowej, obliczenia przy wykorzystaniu pamięci współdzielonej bloku wątków*
3. *Grid wieloblokowy, jeden wątek oblicza 2 lub 4 (podział pracy dwuwymiarowy) sąsiednich elementów macierzy wynikowej, obliczenia przy wykorzystaniu pamięci współdzielonej bloku wątków*

Nazwa	GTX 970
Liczba rdzeni CUDA	1664
Liczba SM	13
Pamięć	(3,5 + 0,5) GB DDR5
Compute Capability	5.2
Zegar	1050 MHz
Zegar pamięci	7000 MHz

Tab. 1: Opis wykorzystanej karty graficznej

2 Opis algorytmów

2.1 Algorytm 1

```
1  template<int BLOCK_SIZE>
2  __global__ void mat_mul_1(const float* A, const float* B, float* C, int N)
3  {
4          for (int i = 0; i < (N / BLOCK_SIZE); ++i)
5          {
6                  for (int j = 0; j < (N / BLOCK_SIZE); ++j)
7                  {
8                          const int row = i * BLOCK_SIZE + threadIdx.y;
9                          const int col = j * BLOCK_SIZE + threadIdx.x;
10
11                         float result = 0;
12
13                         for (int k = 0; k < N; ++k)
14                                 result += A[row * N + k] * B[N * k + col];
15
16                         C[row * N + col] = result;
17                 }
18         }
19 }
```

Na początek obliczany jest wiersz i rząd macierzy, w której wątek będzie zapisywał otrzymany wynik (linijki 8,9). Następnie obliczany jest tenże element macierzy (linijki 11 - 14). Wynik częściowy przechowywany jest w zmiennej pomocniczej *result*, która będzie przechowywana w rejestrze. Po zakończeniu pętli wewnętrznej, wątek zapisuje wynik we właściwym miejscu macierzy (linia 16). Ponieważ cały kernel pracuje na 1 bloku, każdy z wątków musi być przygotowany na obliczanie wielu elementów macierzy. Gwarantują to 2 zewnętrzne pętle (linie 4-6). Dzielą one macierz na kwadraty o boku `BLOCK_SIZE`. Kiedy wątek skończy liczyć element, zaczyna obliczenia w następnym kwadracie.

2.2 Algorytm 2

```
1  template<int BLOCK_SIZE>
2  __global__ void mat_mul_2(const float* A, const float* B, float* C, int N)
3  {
4          const int bx = blockIdx.x;
5          const int by = blockIdx.y;
6          const int tx = threadIdx.x;
7          const int ty = threadIdx.y;
8
9          const int a_start = N * BLOCK_SIZE * by;
10         const int b_start = BLOCK_SIZE * bx;
11
12         const int a_end = a_start + N;
13
14         const int a_step = BLOCK_SIZE;
15         const int b_step = BLOCK_SIZE * N;
16
17         float result = 0;
18
19         __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
20         __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
21
22         for (int a = a_start, b = b_start; a < a_end; a += a_step, b += b_step)
23         {
24                 As[ty][tx] = A[a + N * ty + tx];
25                 Bs[ty][tx] = B[b + N * ty + tx];
26
27                 __syncthreads();
28
29                 #pragma unroll
30                 for (int k = 0; k < BLOCK_SIZE; ++k)
31                         result += As[ty][k] * Bs[k][tx];
32
33                 __syncthreads();
34         }
35
36         const int c = N * BLOCK_SIZE * by + BLOCK_SIZE * bx;
37         C[c + N * ty + tx] = result;
38 }
```

Na początku obliczane są potrzebne adresy macierzy A i B do obliczeń (linie 4-15), deklarowana jest pamięć współdzielona (19-20). Następnie w pętli, każdy wątek kopiuje z pamięci głównej do współdzielonej 1 element macierzy A i B (24-25), po czym następuje synchronizacja, w celu zapewnienia spójności obrazu pamięci współdzielonej dla każdego wątku. Następnie wątek oblicza wynik częściowy, korzystając z danych w pamięci współdzielonej (29-31). Następnie następuje synchronizacja wątków w bloku, w celu zapobieżenia nadpisania pamięci współdzielonej. Po obliczeniu wyniku, zapisywany on jest z powrotem w pamięci głównej (36-37).

2.3 Algorytm 3

```
1  template<int BLOCK_SIZE>
2  __global__ void mat_mul_3(const float* A, const float* B, float* C, int N)
3  {
4          const int bx = blockIdx.x;
5          const int by = blockIdx.y;
6          const int tx = threadIdx.x;
7          const int ty = threadIdx.y;
8
9          const int a_start = 2 * N * BLOCK_SIZE * by;
10         const int b_start = 2 * BLOCK_SIZE * bx;
11
12         const int a_end = a_start + N;
13
14         const int a_step = 2 * BLOCK_SIZE;
15         const int b_step = 2 * BLOCK_SIZE * N;
16
17         float result[4] = { 0, 0, 0, 0 };
18
19         __shared__ float As[BLOCK_SIZE][BLOCK_SIZE][4];
20         __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE][4];
21
22         for (int a = a_start, b = b_start; a < a_end; a += a_step, b += b_step)
23         {
24                 As[ty][tx][0] = A[a + N * (2 * ty) + (2 * tx)];
25                 As[ty][tx][1] = A[a + N * (2 * ty) + (2 * tx) + 1];
26                 As[ty][tx][2] = A[a + N * (2 * ty + 1) + (2 * tx)];
27                 As[ty][tx][3] = A[a + N * (2 * ty + 1) + (2 * tx) + 1];
28
29                 Bs[ty][tx][0] = B[b + N * (2 * ty) + (2 * tx)];
30                 Bs[ty][tx][1] = B[b + N * (2 * ty) + (2 * tx) + 1];
31                 Bs[ty][tx][2] = B[b + N * (2 * ty + 1) + (2 * tx)];
32                 Bs[ty][tx][3] = B[b + N * (2 * ty + 1) + (2 * tx) + 1];
33
34                 __syncthreads();
35
36                 #pragma unroll
37                 for (int k = 0; k < BLOCK_SIZE; ++k)
38                 {
39                         result[0] += As[ty][k][0] * Bs[k][tx][0];
40                         result[0] += As[ty][k][1] * Bs[k][tx][2];
41
42                         result[1] += As[ty][k][1] * Bs[k][tx][1];
43                         result[1] += As[ty][k][0] * Bs[k][tx][3];
44
45                         result[2] += As[ty][k][2] * Bs[k][tx][2];
46                         result[2] += As[ty][k][0] * Bs[k][tx][3];
47
48                         result[3] += As[ty][k][3] * Bs[k][tx][3];
49                         result[3] += As[ty][k][2] * Bs[k][tx][1];
50                 }
51
52                 __syncthreads();
53         }
54
55         const int c = 2 * N * BLOCK_SIZE * by + 2 * BLOCK_SIZE * bx;
56
57         C[c + N * (2 * ty) + (2 * tx)] = result[0];
58         C[c + N * (2 * ty) + (2 * tx) + 1] = result[1];
59         C[c + N * (2 * ty + 1) + (2 * tx)] = result[2];
60         C[c + N * (2 * ty + 1) + (2 * tx) + 1] = result[3];
61 }
```

Algorytm ten jest rozwinięciem algorytmu 2. Każdy wątek, zamiast obliczać 1 element macierzy, oblicza 4 elementy macierzy (kwadrat 2x2). Na początek obliczane są

niezbędne adresy macierzy (linijki 4-15). Następnie alokowana jest pamięć współdzielona (linie 19-20; zauważmy że potrzeba jej zdecydowanie więcej niż w poprzednim algorytmie!). W pętli głównej algorytmu (22-53) następuje obliczenie 4 elementów macierzy wynikowej. Jak poprzednio, wątki najpierw kopiują do pamięci współdzielonej elementy macierzy A i B, aczkolwiek tym razem po 4 dla każdej macierzy (24-32). Zaraz po operacji kopiowania, umieszczono barierę synchronizacyjną, w celu zapewnienia spójności obrazu pamięci współdzielonej. Po barierze, wątek oblicza na podstawie danych z pamięci współdzielonej wynik częściowy. Zaraz po obliczeniach, umieszczono kolejną barierę, w celu zapewnienia, iż wszystkie wątki będą mogły wykonać obliczenia na tych samych danych. Po obliczeniu wyników końcowych, wątek wysyła je do pamięci głównej (55-60).

3 Analiza teoretyczna wydajności

3.1 Algorytm 1

Jak możemy zauważyć, algorytm ten korzysta bezpośrednio z pamięci globalnej. Każdy wątek liczy na raz jeden element macierzy wynikowej. Aby obliczyć jeden wynik pośredni, musimy 2 razy odwołać się do pamięci globalnej, aby znać wartość elementów macierzy A i B. Następnie wykonujemy 2 operacje: mnożenie i dodawanie. Nasze teoretyczne CGMA wynosi zatem 1. Do obliczenia 1 elementu wynikowego, wątek wykona 2N ładowań z pamięci globalnej i 1 przesłań wyniku do pamięci. Ponieważ nie korzystamy z pamięci współdzielonej, zapotrzebowanie na nią wynosi 0.

Dostęp do pamięci globalnej jest bardzo wolny, ponadto liczba wątków przetwarzających macierz jest bardzo ograniczona, przez co ukrycie opóźnień jest wręcz niemożliwe. Algorytm ten nie osiągnie dużej wydajności i większość czasu będzie spędzał czekając na dane z pamięci.

3.2 Algorytm 2

W tym algorytmie zastosowano pamięć współdzieloną. Na początku każdy wątek pobiera do pamięci współdzielonej jeden element z pamięci głównej. Następnie, przeprowadzane są częściowe obliczenia na fragmentach macierzy A i B. Każdy wątek korzysta z zawartości pamięci współdzielonej. W bloku o rozmiarze `BLOCK_SIZE`, każdy wątek wykona w wewnętrznej iteracji pętli dwa pobrania z pamięci głównej: 1 element macierzy A i B, oraz 2 * `BLOCK_SIZE` obliczeń (1 mnożenie + 1 dodawanie). Współczynnik CGMA wynosi zatem `BLOCK_SIZE`.

$$CGMA = \frac{2 * BLOCK\_SIZE}{2} = BLOCK\_SIZE \tag{1}$$

Wz. 1: CGMA Algorytmu 2

W wyniku optymalizacji, przy obliczaniu jednego elementu wynikowego, wątek wykona $2\frac{N}{\texttt{BLOCK\_SIZE}}$ ładowań z pamięci globalnej oraz 1 przesłanie wyniku do pamięci

globalnej. Zapotrzebowanie na pamięć współdzieloną wynosi $8 * \texttt{BLOCK\_SIZE}^2$ bajtów na 1 blok wątków (2 tablice o wymiarach BLOCK_SIZE x BLOCK_SIZE, każdy element ma rozmiar 4 bajtów).

3.3 Algorytm 3

Ten algorytm jest drobną modyfikacją poprzedniego algorytmu. Każdy wątek przetwarza tutaj 4 elementy macierzy, a nie 1. Zasada działania jest taka sama jak w poprzedniku. W pętli wewnętrznej bloku o rozmiarze BLOCK_SIZE, każdy wątek wykonuje 8 pobrań z pamięci głównej (4 elementy macierzy A oraz B). Następnie wykonywane jest 16*BLOCK_SIZE obliczeń. Otrzymujemy współczynnik CGMA równy 2*BLOCK_SIZE.

$$CGMA = \frac{16 * BLOCK\_SIZE}{8} = 2 * BLOCK\_SIZE \tag{2}$$

Wz. 2: CGMA Algorytmu 3

Wątek obliczając na raz 4 elementy macierzy, wykonuje:

$$G_{load} = \frac{8N}{2 * BLOCK\_SIZE} = 4\frac{N}{BLOCK\_SIZE} \tag{3}$$

Wz. 3: Pobrania z pamięci głównej dla 1 wątku w celu obliczenia 4 elementów na raz

pobrań z pamięci globalnej. Czyli dla 1 elementu wątek wykonuje dokładnie $\frac{N}{BLOCK\_SIZE}$ pobrań z pamięci głównej i 1 przesłanie wyniku z powrotem do pamięci. W tym algorytmie zapotrzebowanie na pamięć współdzieloną wynosi $32 * \texttt{BLOCK\_SIZE}^2$ bajtów na 1 blok wątków (2 tablice o wymiarach BLOCK_SIZE x BLOCK_SIZE x 4, każdy element ma rozmiar 4 bajtów).

4 Pomiary

4.1 Opis miar

Algorytmy 2 i 3 zostały przebadane 24 razy - dla 3 rodzajów bloków oraz 8 rozmiarów macierzy. Algorytm 1 został przebadany tylko 18 razy - dla 3 rodzajów bloków oraz 6 rozmiarów macierzy - było to spowodowane dużym narzutem czasowym Visual Profilera, przez co czasy oczekiwania na wynik przekraczały jedną godzinę dla pojedyńczej instancji. Dodatkowo, każdy pojedynczy wariant został zbadany 300 razy, aby uśrednić wyniki czasowe.

4.1.1 Średni czas przetwarzania

Średni czas przetwarzania jednego pełnego przejścia algorytmu, uśredniany dla 300 instancji.

4.1.2 Przyspieszenie

Przyspieszenie obliczane jest wzorem:

$$P = \frac{T_{cpu}}{T_{gpu}} \tag{4}$$

Wz. 4: Przyspieszenie względem obliczeń na CPU

Gdzie T_{cpu} oznacza czas wykonania algorytmu mnożenia macierzy ikj sekwencyjnego, T_{gpu} oznacza czas wykonania mnożenia macierzy tego samego rozmiaru na karcie graficznej.

4.1.3 GFLOPS

Ilość operacji zmiennoprzecinkowych na sekundę, przedrostek G - Giga oznacza miliardy.

4.1.4 CGMA

Współczynnik CGMA (compute to global memory access ratio) – stosunek liczby operacji do liczby dostępów do pamięci. Poza wartościami teoretycznymi, wyznaczono wartości na podstawie danych z doświadczeń:

$$CGMA = \frac{F}{GT_{load} + GT_{store}} \tag{5}$$

Wz. 5: Obliczanie CGMA z danych

Gdzie F oznacza liczbę operacji zmiennoprzecinkowych, GT_{load} oznacza liczbę transakcji ładowania danych z pamięci głównej, GT_{store} oznacza liczbę transakcji przesyłania danych na pamięć główną.

4.1.5 Zajętość multiprocesora

Oznacza procent wykorzystania multiprocesora podczas wykonywania kodu. Większe wartości oznaczają więcej czasu spędzonego na wykonywaniu kodu przez kartę graficzną, niż na np. czekaniu na dane z pamięci.

4.1.6 Global load/store transactions

Ilość transakcji z pamięcią główną dla odczytów lub zapisów.

4.1.7 Memory efficiency

Miara obliczana przez profiler, zgodnie ze wzorem:

$$GL_{eff} = \frac{T_{requested}}{T_{required}} \tag{6}$$

Wz. 6: Obliczanie memory efficiency

Gdzie $T_{requested}$ oznacza żądaną przepustowość, a $T_{required}$ oznacza wymaganą przepustowość. Miarę tą możemy interpretować jako stopień łączenia dostępów do pamięci - im większa, tym więcej dostępów zostało scalonych.

5 Wyniki

	Algorytm 1			Algorytm 2			Algorytm 3		
N	8x8	16x16	32x32	8x8	16x16	32x32	8x8	16x16	32x32
64	277.67	47.87	23.34	4.73	4.05	5.24	4.29	4.07	9.60
128	1837.31	364.39	176.70	12.03	11.57	15.08	10.46	10.75	17.07
192	5829.40	1214.77	581.82	37.78	34.76	34.07	24.01	18.94	27.27
256	13642.08	2842.66	1331.65	82.34	74.34	70.55	52.18	36.58	66.03
320	26384.69	5588.07	2394.28	156.01	136.73	137.63	114.94	78.57	90.95
384	45569.94	9787.03	4000.79	263.61	236.18	239.30	212.24	120.66	151.26
448				407.33	367.52	372.13	279.53	198.30	233.33
512				635.86	528.04	524.76	380.26	278.30	324.82

Tab. 2: Czas przetwarzania (µs)

	Algorytm 1			Algorytm 2			Algorytm 3		
N	8x8	16x16	32x32	8x8	16x16	32x32	8x8	16x16	32x32
64	1.79	11	20.89	61.07	60.3	38.35	55.31	58.63	41.99
128	2.28	11.2	23.46	219.03	266.68	230.72	299.55	296.89	206.35
192	2.41	11.76	24.21	330.92	365.27	367.81	520.33	640.06	445.86
256	2.45	11.69	25.07	373.53	426.27	434.8	606.41	791.21	459.77
320	1.32	11.47	27.29	396.33	455.89	456.49	522.47	768.6	678.35
384	1.64	14.38	28.23	421.17	461.41	458.58	516.96	904.04	707.16
448				431.49	469.63	458.02	619.1	864.31	742.64
512				411.33	491.21	497.17	685.54	931.1	796

Tab. 3: GFLOPS

	Algorytm 1			Algorytm 2			Algorytm 3		
N	8x8	16x16	32x32	8x8	16x16	32x32	8x8	16x16	32x32
64	0.40	2.33	4.77	23.54	27.48	21.28	25.98	27.38	11.61
128	0.22	1.10	2.27	33.31	34.63	26.57	38.31	37.27	23.47
192	0.34	1.63	3.41	52.57	57.13	58.29	82.73	104.87	72.83
256	0.43	2.07	4.42	71.45	79.14	83.39	112.77	160.82	89.10
320	0.38	1.79	4.17	64.03	73.06	72.58	86.91	127.14	109.83
384	0.36	1.66	4.05	61.51	68.66	67.76	76.40	134.39	107.20
448				63.97	70.90	70.02	93.22	131.41	111.68
512				57.55	69.30	69.74	96.24	131.50	112.66

Tab. 4: Przyspieszenie w stosunku do obliczeń na CPU (ikj)

	Algorytm 1			Algorytm 2			Algorytm 3		
N	8x8	16x16	32x32	8x8	16x16	32x32	8x8	16x16	32x32
64	1.992	1.996	1.996	15.514	31.027	60.221	30.114	56.877	102.360
128	1.996	1.998	1.998	15.754	31.507	62.059	31.030	60.234	113.772
192	1.997	1.999	1.999	15.835	31.670	62.693	31.347	61.439	118.152
256	1.998	1.999	1.999	15.876	31.752	63.015	31.508	62.060	120.470
320	1.998	1.999	1.999	15.901	31.801	63.210	31.605	62.439	121.904
384	1.999	1.999	1.999	15.917	31.834	63.340	31.670	62.694	122.880
448				15.929	31.858	63.434	31.717	62.877	123.586
512				15.938	31.875	63.504	31.752	63.015	124.121

Tab. 5: CGMA

	Algorytm 1			Algorytm 2			Algorytm 3		
N	8x8	16x16	32x32	8x8	16x16	32x32	8x8	16x16	32x32
64	1024	512	512	1024	512	512	1024	1024	1024
128	4096	2048	2048	4096	2048	2048	4096	4096	4096
192	9216	4608	4608	9216	4608	4608	9216	9216	9216
256	16384	8192	8192	16384	8192	8192	16384	16384	16384
320	25600	12800	12800	25600	12800	12800	25600	25600	25600
384	36864	18432	18432	36864	18432	18432	36864	36864	36864
448				50176	25088	25088	50176	50176	50176
512				65536	32768	32768	65536	65536	65536

Tab. 6: Global store transactions

	Algorytm 1			Algorytm 2			Algorytm 3		
N	8x8	16x16	32x32	8x8	16x16	32x32	8x8	16x16	32x32
64	2.62E+05	2.62E+05	2.62E+05	3.28E+04	1.64E+04	8.19E+03	1.64E+04	8.19E+03	4.10E+03
128	2.10E+06	2.10E+06	2.10E+06	2.62E+05	1.31E+05	6.55E+04	1.31E+05	6.55E+04	3.28E+04
192	7.08E+06	7.08E+06	7.08E+06	8.85E+05	4.42E+05	2.21E+05	4.42E+05	2.21E+05	1.11E+05
256	1.68E+07	1.68E+07	1.68E+07	2.10E+06	1.05E+06	5.24E+05	1.05E+06	5.24E+05	2.62E+05
320	3.28E+07	3.28E+07	3.28E+07	4.10E+06	2.05E+06	1.02E+06	2.05E+06	1.02E+06	5.12E+05
384	5.66E+07	5.66E+07	5.66E+07	7.08E+06	3.54E+06	1.77E+06	3.54E+06	1.77E+06	8.85E+05
448				1.12E+07	5.62E+06	2.81E+06	5.62E+06	2.81E+06	1.40E+06
512				1.68E+07	8.39E+06	4.19E+06	8.39E+06	4.19E+06	2.10E+06

Tab. 7: Global load transactions

	Algorytm 1			Algorytm 2			Algorytm 3		
N	8x8	16x16	32x32	8x8	16x16	32x32	8x8	16x16	32x32
64	0.03	0.12	0.49	0.15	0.16	0.49	0.04	0.12	0.48
128	0.03	0.12	0.50	0.59	0.59	0.64	0.15	0.16	0.49
192	0.03	0.13	0.50	0.80	0.81	0.88	0.33	0.34	0.49
256	0.03	0.13	0.50	0.89	0.89	0.91	0.59	0.60	0.50
320	0.03	0.13	0.50	0.95	0.95	0.96	0.57	0.63	0.50
384	0.03	0.13	0.50	0.95	0.95	0.97	0.58	0.70	0.50
448				0.96	0.97	0.98	0.61	0.69	0.50
512				0.97	0.98	0.98	0.61	0.71	0.50

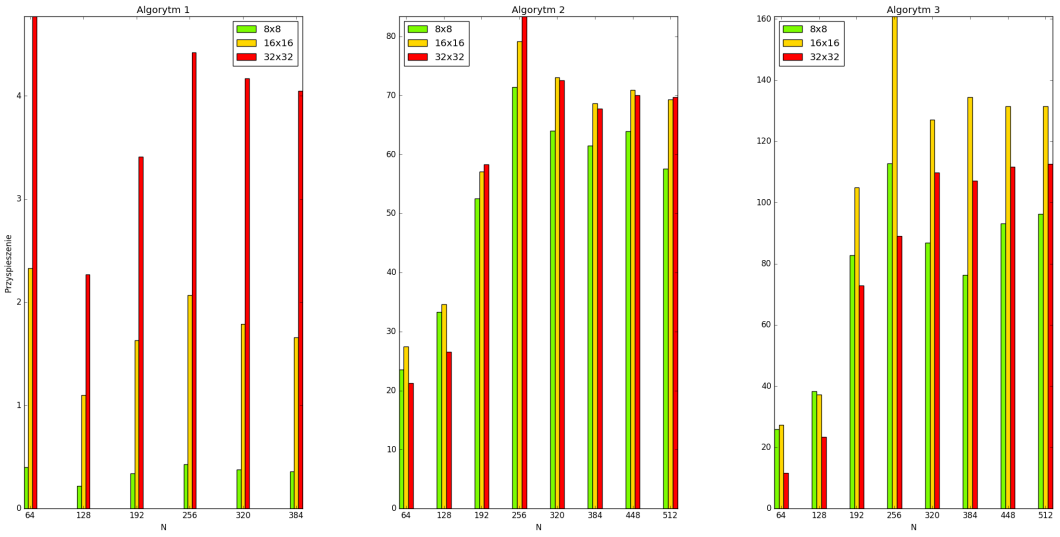
Tab. 8: Zajętość multiprocesora (1.0 - 100%, 0.0 - 0%)

		Algorytm 1			Algorytm 2			Algorytm 3		
	N	8x8	16x16	32x32	8x8	16x16	32x32	8x8	16x16	32x32
LOAD	64∼512	30	56.25	82.5	100	50		50		
STORE	64∼512	50	100	100	50	100	100	50		

Tab. 9: Memory efficiency (%)

		Algorytm 1			Algorytm 2			Algorytm 3		
	N	8x8	16x16	32x32	8x8	16x16	32x32	8x8	16x16	32x32
LOAD	64∼512	8	4	4	8	4	4	8	8	8
STORE	64∼512	16								

Tab. 10: Średnia ilość transakcji na 1 odwołanie do pamięci



Rys. 1: Przyspieszenie w stosunku do obliczeń na CPU

6 Wnioski

6.1 Czasy wykonania i przyspieszenie

Jest to najprostsza, ale też i najlepsza metoda oceny algorytmów na kartach graficznych, gdyż przeważnie używane są one do operacji wyświetlania obrazów (klatek), gdzie ważna jest płynność obrazu. Algorytmem, który działał dla nas najlepiej jest Algorytm 3 (Tab. 2), podczas gdy Algorytm 1 był momentami gorszy od obliczeń na CPU. Można więc śmiało stwierdzić, iż Algorytm 1 będzie tutaj złym przykładem kodu, podczas gdy Algorytmy 2 i 3 będą głównymi tematami do wniosków i porównań.

6.2 Co mówi nam CGMA?

Nasze pomiary wskazały niespójność wyników z naszymi przewidywaniami - otóż, wyliczone przez nas teoretyczne wartości CGMA (Wz. 1, Wz. 1, podpunkt 3 sprawozdania) znacząco różnią się od otrzymanych (Tab. 5). Łatwo można zauważyć pewną korelację - rzeczywiste wyniki są prawie dwa razy lepsze od wyliczonych ręcznie wartości.

Najbardziej prawdopodobną przyczyną jest optymalizacja dostępu do pamięci globalnej - na przykład pobranie danych z jednej macierzy jest liczone jako jeden dostęp do pamięci. Zmniejszyłoby to pobrania o (N - 1) i dało mniej więcej otrzymany rezultat.

Należy zwrócić uwagę na zwiększanie się CGMA wraz ze zwiększaniem bloku wątków w Algorytmach 2 i 3, oraz zastój w Algorytmie 1 (Tab. 5). Jest to oznaką nieefektywnego algorytmu, oraz mówi nam, iż Algorytm 1 nie powinien być stosowany w praktyce. Wartym wzmianki jest fakt, iż Algorytm 3 ma dwa razy większe CGMA niż Algorytm 2, co znacząco wpłynie na czas wykonania, który bezpośrednio zależy od ilości dostępów do globalnej pamięci.

6.3 Wykorzystanie GPU a efektywność algorytmu

Jak można łatwo wywnioskować, większe wykorzystanie karty graficznej powoduje szybsze rezultaty, przy odpowiednich algorytmach. Z pomiarów (Tab. 8) możemy bardzo szybko stwierdzić, iż Algorytm 1 jest złym pomysłem - maksymalne wykorzystanie karty to 50% oraz nie rośnie on przy zwiększeniu wielkości pamięci. Ciekawszymi przypadkami do analizy są dwa pozostałe - Algorytm 3, pomimo bycia szybszym (Tab. 2, Tab. 3) nie wykorzystuje w pełni możliwości karty. Co prawda wykorzystanie dla bloków 8x8 i 16x16 wzrasta, ale tylko nieznacznie. Widać także, iż blok 16x16 jest prawdopodobnie lepszy od 32x32, co jest potwierdzone przyspieszeniem (Rys. 1). Na duża ilość uwagi zasługuje tutaj Algorytm 2, który prawie w pełni wykorzystuje możliwości karty, gdzie kilka brakujących procent może być winą synchronizacji wątków. Dla większych rozmiarów macierzy (rozmiar N) istnieje możliwość, iż Algorytm 2 będzie zmniejszał różnice czasowe między Algorytmem 3, który nie korzysta ze wszystkich możliwych zasobów.

6.4 Wpływ pamięci na wydajność przetwarzania

Przeanalizujmy jak rodzaj, wielkość pamięci oraz ilość dostępów do niej przekłada się na wydajność algorytmów. Wskaźnikami wydajności będą następujące parametry: czas przetwarzania (Tab. 2), ilość operacji zmiennoprzecinkowych na sekundę (Tab. 3) i przyspieszenie w stosunku do obliczeń na CPU (Tab. 4, Rys. 1). Algorytm 1 jest najwolniejszy z całej trójki, niezależnie od rozmiaru bloku. Czasy wykonania są co najmniej o rząd wielkości większe, ilość operacji na sekundę jest niewielka w porównaniu z pozostałymi algorytmami. Ponadto w przypadku bloku 8x8, algorytm ten jest wolniejszy od obliczeń sekwencyjnych na CPU (Tab 4, przyspieszenie nawet nie przekracza 0.5)! Operowanie wyłącznie na pamięci globalnej jest przyczyną takiego spowolnienia. Zobaczmy iż algorytm ten wykonuje co najmniej 10 razy więcej transakcji ładowania danych z pamięci (Tab 7), w porównaniu z algorytmami 2 i 3. Możemy się domyślać, iż nie ma tutaj możliwości łączenia dostępów do pamięci, przez co ilość transakcji jest tak duża. Algorytm większość czasu spędza czekając na pamięć globalną, co objawia się bardzo niską zajętością multiprocesora (Tab. 8).

Algorytmy 2 i 3 są dużo lepsze od poprzednika. Ilość dostępów do pamięci głównej drastycznie zmniejsza się (Tab. 7). Zastosowanie pamięci współdzielonej zdecydowanie pomaga w zwiększeniu wydajności algorytmu (Tab. 2,3,4). Dzięki wyraźnie wydzielonym fazom pobierania danych z pamięci głównej do współdzielonej, możliwe jest lepsze łączenie dostępów do pamięci. Im więcej danych na raz pobiera każdy wątek, tym większe są możliwości łączenia (w porównaniu z algorytmem 2, algorytm 3 ma 2 razy mniej ładowań z pamięci głównej; Tab. 7). Pamiętając, iż zapotrzebowanie na pamięć współdzieloną jest zależne od rozmiaru bloku, możemy zauważyć kolejne ciekawe zjawisko: dla Algorytmu 3, zdecydowanie lepszy jest rozmiar bloku 16x16, a nie 32x32! Pomimo tego, iż CGMA jest większe dla 32x32, czasy wykonania, GFLOPS i przyspieszenie (Tab. 2,3,4) są dużo większe dla wersji 16x16. Możliwe iż ilość pamięci współdzielonej odgrywa tutaj istotną rolę, dla wersji 32x32 zaczyna jej brakować, co objawia się spadkiem wydajności (Rys. 1, Tab. 2,3,4) i zajętości multiprocesora (Tab. 8).

6.5 Ocena FLOPS

Jak można łatwo zauważyć, ilość FLOPS'ów zwiększa się w Algorytmach 2 i 3 oraz praktycznie nie zmienia się w Algorytmie 1 (Tab. 3). Brak zmian jest spowodowany dużymi przerwami w obliczeniach, które są skutkiem dużej ilości odwołań do pamięci globalnej. Natomiast w Algorytmach 2 i 3 stabilnie zwiększa się aż do rozmiaru N = 256, a następnie wahając się dalej ma tendencję wzrostową. Gwałtowny wzrost w małych wielkościach tablic wywołany jest przez zmniejszenie opóźnień występujących na początku obliczeń, takich jak inicjalizacja wartości i przygotowanie karty graficznej. Powolny wzrost jest normalnym działaniem karty na odpowiednio dobranym algorytmie, gdyż zacierają się tam koszty rozpoczęcia kodu.

6.6 Wpływ wielkości bloku wątków

Testowaliśmy algorytmy dla 3 różnych wielkości bloku - 8x8, 16x16 oraz 32x32.

W przypadku Algorytmu 1 wzrost szybkości obliczeń jest prosty do wytłumaczenia. Jest on wprost proporcjonalny do ilości wątków, im więcej, tym lepiej. Dzieje się tak przez grupowanie transakcji z pamięci globalnej, gdzie 8x8 pobiera je 4 razy wolniej niż 16x16 oraz 16 razy wolniej niż 32x32. Możemy to potwierdzić za pomocą CGMA (Tab. 5) oraz czasem przetwarzania i zajętością multiprocesora (Tab. 1, Tab. 9). Widać tam, iż CGMA praktycznie nie zmienia się dla różnych bloków, natomiast czas znacząco spada. Bardzo ładnie widać zależność większej ilości wątków na zajętości GPU - proporcje są praktycznie dokładnie równe 4 oraz 16 odpowiednio pomiędzy 16x16 a 8x8 oraz 32x32 a 8x8.

Przypadki algorytmów 2 i 3 są bardziej złożone. Dla algorytmu 2 możemy również zaobserwować iż kiedy blok ma rozmiar 16x16 lub 32x32, dostępy do pamięci są zdecydowanie lepiej grupowane. Objawia się to spadkiem ilości transakcji na 1 odwołanie do pamięci (Tab 10) oraz spadkiem ilości transakcji (Tab. 6, 7).

W Algorytmie 3 sytuacja komplikuje się. Ponieważ praca podzielona jest dwuwymiarowo, wątek potrzebuje do obliczeń 2 rzędów/kolumn odpowiednich macierzy do

obliczeń, a nie pojedynczych jak w pozostałych dwóch algorytmach. Z tego powodu ilość transakcji przypadających na żądanie pomimo zwiększenia rozmiaru bloku nie będzie spadać, co potwierdzają dane eksperymentalne (Tab. 10). Natomiast ogólna ilość transakcji, zgodnie z przewidywaniami, jest dwukrotnie mniejsza od Algorytmu 2 (Tab. 7), kosztem zwiększenia ilości zapisów, z powodu zapisu do 2 wierszy na raz (Tab 6).

6.7 Podsumowanie

Jak można łatwo wywnioskować z powyższych akapitów, Algorytm 1 jest przykładem niepoprawnego programu dla karty graficznej. Jego powolność i całkowity brak wsparcia pamięci współdzielonej czynią go wzorem do czego nie należy dążyć.

Z drugiej strony, algorytmy 2 i 3 są bardzo dobrym przyspieszeniem obliczeń w porównaniu do CPU. Jednakże patrząc na nie razem, możemy stwierdzić, iż prawdopodobnie istnieje lepszy sposób mnożenia macierzy - taki, który używa GPU i pamięć (Tab. 8, Tab. 9, Tab. 10) jak Algorytm 2, a zarazem osiąga wysoką ilość FLOPS oraz CGMA (Tab. 3, Tab. 5) jak Algorytm 3. Połączenie tych cech skutkowałoby szybszym oraz wydajniejszym algorytmem.