Programowanie Równoległe - CUDA

Wersja pierwsza

Autorzy

Grupa dziekańska: 4
Grupa labolatoryjna: 7
Termin zajęć: czwartek, 16:50

Tymoteusz Jagła 151811 - tymoteusz.jagla@student.put.poznan.pl
Kaper Magnuszewski 151746 - kacper.magnuszewski@student.put.poznan.pl

Sprawozdanie

Wymagany termin oddania sprawozdania - 31.05.2024
Rzeczywisty termin oddania sprawozdania - 31.05.2024

Cel zadania

Celem ćwiczenia jest praktyczne zapoznanie z zasadami programowania równoległego procesorów kart graficznych (PKG), zapoznanie z zasadami optymalizacji kodu dla PKG oraz ocena prędkości przetwarzania przy użyciu PKG i poznanie czynników warunkujących realizację efektywnego przetwarzania.

Opis zadania

Zadanie polega na sumowaniu wartości tablicy o wielkości $N \times N$, które znajdują się w określonym przez „promień” R obszarze. Tablica wynikowa ma rozmiar $(N-2R) \times (N-2R)$, sumy obliczane są dla wszystkich pozycji w odległości większej lub równej R od krawędzi tablicy wejściowej.

Tablica Dwuwymiarowa $TAB[N][N]$ (o wierszu długości N, słowo tablicy $TAB[i][j]$ jest dostępne jako $TAB[i \cdot N+j]$). Dla tablicy wejściowej TAB należy wyliczyć tablicę wyjściową $OUT[N-2R][N-2R]$ (gdzie $N > 2R$) zawierającą sumy elementów w „promieniu” R. Każdy element tablicy wyjściowej to suma $(2 \cdot R+1) \cdot (2 \cdot R+1)$ wartości.

Przykładowo dla $R = 1$ $OUT[i][j] = TAB[i][j] + TAB[i][j-1] + TAB[i][j+1] + TAB[i-1][j-1] + TAB[i-1][j] + TAB[i-1][j+1] + TAB[i+1][j] + TAB[i+1][j-1] + TAB[i+1][j+1]$.

Wykorzystany system obliczeniowy

Procesor (CPU)

- Model: 13th Gen Intel® Core(TM) i5-13600KF
- Liczba procesorów fizycznych: 14
  - 6 Performance-cores
  - 8 Efficient-cores
- Liczba procesorów logicznych: 20
  - 2 wątki na pojedyńczy Performance-core
  - 1 wątek na pojedyńczy Effitient-core
- Oznaczenie typu procesora: KF
- Taktowanie procesora:
  - Minimalne: 800MHz
  - Maksymalne: 51000MHz
- Wielkości pamięci podręcznej procesora:
  - L1d cache: 544 KiB (14 instancji)
  - L1i cache: 704 KiB (14 instancji)
  - L2 cache: 20 MiB (8 instancji)
  - L3 cache: 24 MiB (1 instancja)
- Organizacja pamięci podręcznej: Intel® Smart Cache

Jednostka przetwarzania graficznego (GPU)

- Model: NVIDIA GeForce RTX 4070 SUPER 12G VENTUS 2X OC
- Nazwa technologii: Ada Lovelace
- Producent: Micro-Star International
- Układ graficzny: AD104-350
- Parametr CUDA compute capability: 8.9
- Liczba tranzystorów: 35.800 milionów
- Proces technologiczny: 5nm
- Rdzenie CUDA: 7168
- Jednostki TMU: 224
- Jednostki ROP: 80
- Jednostki RT: 56
- Jednostki Tensor: 224
- Pamięć VRAM: 12 GB GDDR6X
- Magistrala pamięci: 192-bitowa
- Taktowanie pamięci: 1313 MHz
- Taktowanie pamięci efektywne: 21000 MHz
- Przepustowość pamięci: 504 GB/s
- Taktowanie rdzenia (bazowe): 1980 MHz
- Taktowanie rdzenia (boost): 2505 MHz
- Pamięć cache L2: 48 MB
- Pobór mocy (TGP): 220 W
- Wersja sterownika: NVIDIA 551.61

System Operacyjny

- Nazwa systemu operacyjnego: Microsoft Windows 11 N
- Oprogramowanie wykorzystane do przygotowania kodu wynikowego: Visual Studio 2022

- Oprogramowanie wykorzystane do przeprowadzenia testów: NVIDIA Nsight Compute 2024.02

Wersje programów

Wykorzystane zmienne:

- N - wielkość wymiaru tablicy wejściowej
- R - promień, w jakim realizowane jest sumowanie
- K - liczba wyników obliczanych przez jeden wątek
- BS - wielkość wymiaru bloku wątków
- $tab[N \cdot N]$ - tablica wejściowa o wielkości $N * N$
- $out[(N-2R) \cdot (N-2R)]$ - tablica wyjściowa o wielkości $(N-2R) \cdot (N-2R)$

Algorytm rozwiązujący problem sekwencyjnie dla głównego procesora komputera

Poniższa funkcja to część programu, która służy obliczeniom sekwencyjnym (wykorzystuje jeden wątek) przy użyciu głównego procesora komputera. Zewnętrzne pętle programu z zmiennymi iteracyjnymi *i* oraz *j* wskazują na kolejne pola tablicy wejściowej, dla których będziemy przeprowadzali sumowanie. Pola w odległości mniejszej od *R* są pomijane ze względu na niemożliwe przeprowadzenie sumy, gdy pola w zasięgu promienia wychodzą poza obszar tablicy. Pętle wewnętrzne wyznaczające *x* i *y* wskazują na kolejne pola występujące w obrębie promienia sumowania, kolejno odczytywane są wartości tablicy wejściowej na wskazanych indeksach, a następnie zwiększana jest wartość sumy dla komórki tablicy wyjściowej. Gdy zsumowane zostaną wszystkie pola w obrębie promienia *R* wartość sumy *sum* zapisywana jest do tablicy wyjściowej.

Kod 1. Obliczenia sekwencyjne

```
1
2  void sequential(float tab[N*N], float
    out[(N-2*R)*(N-2*R)])
3  {
4      for (int i = R; i < N - R; i++) {
5          for (int j = R; j < N - R; j++) {
6              float sum = 0;
7              for (int x = i - R; x <= i + R; x++) {
8                  for (int y = j - R; y <= j + R; y++)
    {
9                      sum += tab[x * N + y];
10                 }
11             }
12             out[(i - R) * (N - 2 * R) + j - R] =
    sum;
13         }
14     }
15 }
```

Algorytm rozwiązujący problem równolegle wykorzystujący pamięć globalną

Poniższy kernel służy obliczeniom równoległym przy użyciu technologii CUDA procesora graficznego. Algorytm efektywnie wykorzystuje dane w pamięci globalnej karty - wątki realizują jednocześnie dostępy do sąsiednich elementów w pamięci globalnej. Wartości *i* i *j* są indeksami określającymi pozycję w tablicy wynikowej, wyliczane są na podstawie indeksu wątku, indeksu bloku, a także wymiaru bloku. Wartość *i* jest dodatkowo przemnażana przez zmienną *kkk*, która odpowiada parametrowi *K* - liczbie komórek tablicy wyjściowej przetwarzanej przez pojedynczy wątek. Pierwsza pętla przechodzi po wartościach *k*, które oznaczają kolejne komórki tablicy wynikowej przetwarzane przez wątek. Następnie dwie wewnętrzne pętle ustalają wartości *y* i *x*, które służą do odczytu wartości w promieniu *R* w tablicy wejściowej. Po odczycie wartości sumowanej komórki zwiększana jest lokalna zmienna *sum*, która następnie jest wpisywana w odpowiadające miejsce w tablicy wyjściowej. Ze względu wykorzystanie przesunięcia *k* (kolejnych obliczanych komórek tablicy wyjściowej) w powiązaniu z indeksem w kolumnie, następne dane w tablicy są w bezpośrednim sąsiedztwie nawet, gdy obliczana jest więcej niż jedna komórka tablicy wyjściowej.

Kod 2. Obliczenia przy użyciu CUDA

```
__global__ void localKernel(float* tab, float* out,
 int* kkk)
{
    int i = (threadIdx.x + blockIdx.x * blockDim.x)
 * *kkk;
    int j = (threadIdx.y + blockIdx.y * blockDim.y);

    for (int k = 0; k < *kkk; k++) {
        int ik = i + k;
        if (ik < OUTSIZE) {
          float sum = 0;
          for (int y = 0; y <= 2*R; y++) {
            int jy = (j + y)*N;
            for (int x = 0; x <= 2*R; x++) {
              sum += tab[jy + (ik + x)];
            }
          }
          out[(j) * (OUTSIZE) + ik] = sum;
        }
    }
}
```

Wywołanie procedury kernela

Poniższy kod odpowiada za wywołanie procedury kernela wykorzystywanego to przeprowadzenia obliczeń. Kolejno ustawiane jest urządzenie GPU, alokowana jest pamięć karty graficznej, do której skopiowane zostaną tablice wejściowa i wyjściowa, a następnie tablica wejściowa kopiowana jest z urządzenia host'a do pamięci karty graficznej. Kernel wywoływany jest za pomocą polecenia `localKernel <<< blocksMatrix, threadsMatrix >>>(dev_tab, dev_out);`, podawana jest macierz bloków o wymiarach *ceil*(*OUTSIZE*/

BLOCKSIZE/*K*)/*ceil*(*OUTSIZE*/*BLOCKSIZE*), gdzie *OUTSIZE* jest równe wartości $N - 2R$ - wielkości krawędzi tablicy wynikowej. Macierz bloków uwzględnia wykonywanie obliczeń dla kilku komórek tablicy *out* przez jeden wątek - jeden z wymiarów jest dzielony przez wartość *K*.

Po wywołaniu kernela sprawdzane jest czy wystąpiły błędy, następuje synchronizacja - oczekiwanie na zakończenie wywoływania kernela na GPU, po synchronizacji kopiowane są wartości tablicy wynikowej `dev_out` znajdującej się w pamięci karty graficznej do tablicy *out* na urządzeniu hosta.

Kod 3. Wywołanie kernela

```
1  cudaError_t sumLocalWithCuda(float* tab, float* out)
2  {
3      float* dev_tab = 0;
4      float* dev_out = 0;
5      cudaError_t cudaStatus;
6
7      cudaStatus = cudaSetDevice(0);
8      if (cudaStatus != cudaSuccess) {
9          fprintf(stderr, "cudaSetDevice failed!  Do
       you have a CUDA-capable GPU installed?");
10         goto Error;
11     }
12
13     cudaStatus = cudaMalloc((void**)&dev_tab, N * N
       * sizeof(float));
14     if (cudaStatus != cudaSuccess) {
15         fprintf(stderr, "cudaMalloc failed!");
16         goto Error;
17     }
18
19     cudaStatus = cudaMalloc((void**)&dev_out,
       (OUTSIZE) * (OUTSIZE) * sizeof(float));
20     if (cudaStatus != cudaSuccess) {
21         fprintf(stderr, "cudaMalloc failed!");
22         goto Error;
23     }
24
25     cudaStatus = cudaMemcpy(dev_tab, tab, N*N *
       sizeof(float), cudaMemcpyHostToDevice);
26     if (cudaStatus != cudaSuccess) {
27         fprintf(stderr, "cudaMemcpy failed!");
28         goto Error;
29     }
30
31     dim3 threadsMatrix(BLOCK_SIZE, BLOCK_SIZE);
32     dim3 blocksMatrix(ceil((OUTSIZE) /
       (float)BLOCK_SIZE / K), ceil((OUTSIZE) /
       (float)BLOCK_SIZE));
33
34     localKernel<<< blocksMatrix, threadsMatrix
       >>>(dev_tab, dev_out);
35
```

```
36      // Check for any errors launching the kernel
37      cudaStatus = cudaGetLastError();
38      if (cudaStatus != cudaSuccess) {
39          fprintf(stderr, "local launch failed: %s\n",
    cudaGetErrorString(cudaStatus));
40          goto Error;
41      }
42
43      // cudaDeviceSynchronize waits for the kernel to
    finish, and returns
44      // any errors encountered during the launch.
45      cudaStatus = cudaDeviceSynchronize();
46      if (cudaStatus != cudaSuccess) {
47          fprintf(stderr, "cudaDeviceSynchronize
    returned error code %d after launching addKernel!
    \n", cudaStatus);
48          goto Error;
49      }
50
51      cudaStatus = cudaMemcpy(out, dev_out, (OUTSIZE)
    * (OUTSIZE) * sizeof(float),
    cudaMemcpyDeviceToHost);
52      if (cudaStatus != cudaSuccess) {
53          fprintf(stderr, "cudaMemcpy failed!");
54          goto Error;
55      }
56
57      Error:
58      cudaFree(dev_tab);
59      cudaFree(dev_out);
60
61      return cudaStatus;
62  }
```

Opis wykonania zadania

Generowanie wartości testowych i sprawdzanie poprawności obliczeń

Do generowania wartości testowych użyliśmy liczb rzeczywistych pseudolosowych generowanych za pomocą funkcji `rand()` z biblioteki standardowej. Aby przetestować poprawność obliczeń przy użyciu algorytmu rozwiązującego problem równolegle porównywaliśmy za każdym razem jego wyjście do wyjścia funkcji obliczającej wartości tablicy sekwencyjnie. Ostateczna wersja algorytmu równoległego była w stanie rozwiązać zadany problem obliczeniowy poprawnie przy każdej próbie.

Użyte miary wydajności

Użytymi przez nas miarami wydajności był czas jaki zajęły obliczenia, ilość GigaFLOP’ów / s oraz ilość FLOP’ów / Bajt.

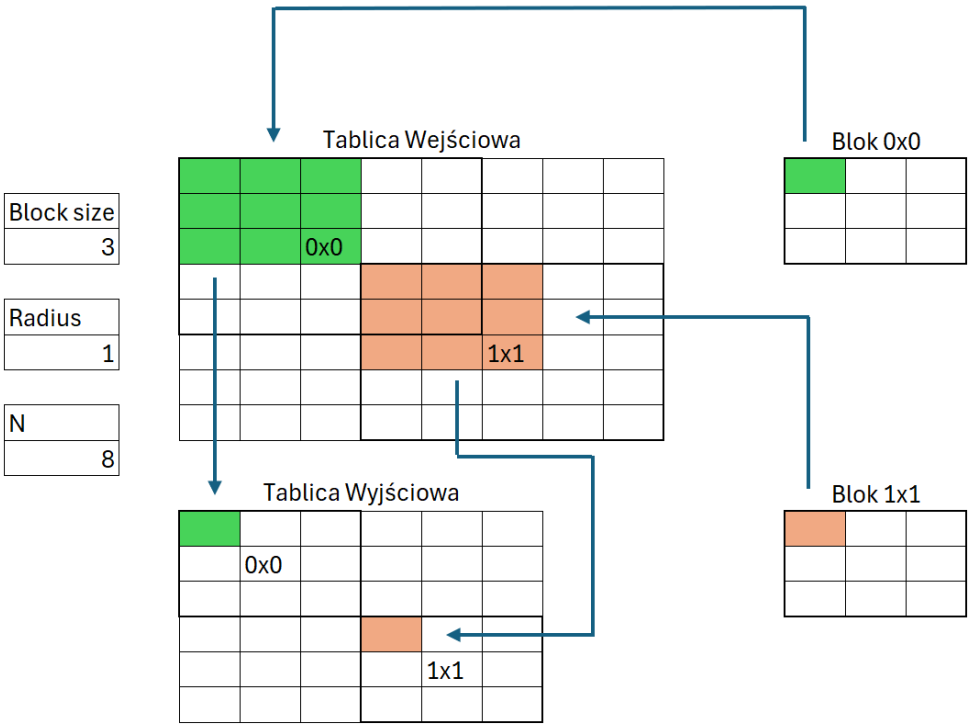
Do obliczenia ilości flopów w naszym algorytmie użyliśmy wzoru: $(N-2R)^2 \cdot (2R+1)^2$, gdzie $(N-2R)^2$ to rozmiar tablicy wynikowej, a $(2R+1)^2$ to ilość operacji wykonywanych dla jednej komórki tablicy wynikowej.

Do wyznaczenia miary GFLOP / s użyliśmy wzoru: $FLOP/(1e9 \cdot t)$, gdzie FLOP, to ilość flop'ów obliczona za pomocą poprzedniego wzoru, a t to czas jaki zajęły obliczenia.

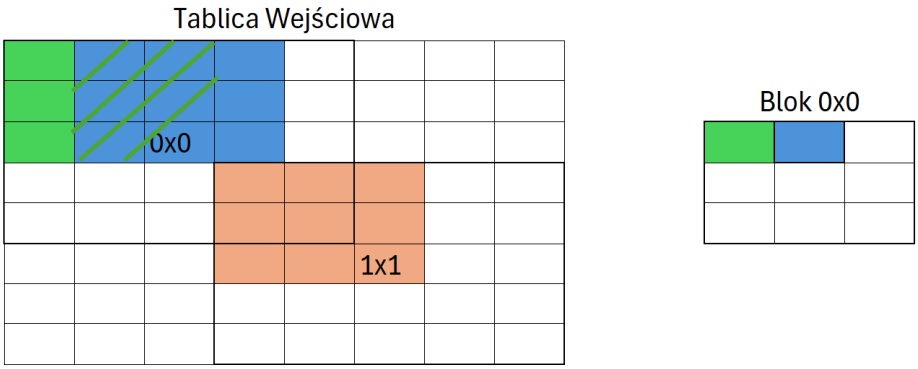
Ostatnią użytą przez nas miarą wydajności była ilość FLOP'ów / B. Wyniki zapisane w poniższych pomiarach zostały wygenerowane przez używane przez nas oprogramowanie - NVIDIA Nsight Compute. Nie wykorzystaliśmy wyników własnych obliczeń, ponieważ miara ta jest zależna od sposobu korzystania z pamięci globalnej karty. Kernel w naszym projekcie napisany został w taki sposób, aby wykorzystywać tę pamięć bardzo efektywnie. Oznacza to, że wątki realizują jednocześnie dostępy do sąsiadujących ze sobą w pamięci globalnej elementów tablicy (koalescencja dostępu do pamięci globalnej). Dzięki temu dostępy do pamięci karty są łączone. Oznacza to, że obliczane przez nas FLOP'y / B nie są wartością prawdziwą, ponieważ rozpatrzamy jedynie najgorszy możliwy przypadek. Wzór jakiego użyliśmy to: $FLOP((N-2R)^2 \cdot ((2R+1)^2+1) \cdot sizeof(float)$, gdzie FLOP oznacza ilość flop'ów, $(N-2R)^2$ to rozmiar tablicy wynikowej, $\{(2R+1)^2 + 1\}$ to liczba operacji wykonywanych dla jednej komórki tablicy wynikowej wraz z zapisem do niej obliczonej sumy.

Zobrazowanie problemu

Schemat 1. Miejsce dostępu i kolejność dostępu do danych realizowane przez poszczególne wątki i bloki



Schemat 2. Wartości wyników wyznaczane przez bloki i wątki



Pomiary

Tabele

Tabela 1. N = 1000, R = 2

Wariant	BlockSize	K	Czas [s]	GFLOP/s	FLOP/B
0 - Sekwencyjny	-	-	0.0260	0.9539	-
1	8	1	0.0590	123.4883	3,09
2	16	1	0.0610	119.0678	4,56
3	32	1	0.0600	95.8462	3,68
4	8	4	0.0610	78.1893	3,98
5	16	4	0.0610	92.1756	5,01
6	32	4	0.0570	53.2618	4,92
7	8	16	0.0600	81.9599	2,91
8	16	16	0.0550	103.8752	4,92
9	32	16	0.0570	58.3726	3,30

Tabela 2. N = 1000, R = 8

Wariant	BlockSize	K	Czas [s]	GFLOP/s	FLOP/B
0 - Sekwencyjny	-	-	0.4360	0.6418	-
1	8	1	0.0670	231.9205	10,75
2	16	1	0.0930	251.5046	54,23
3	32	1	0.0590	142.7381	52,66
4	8	4	0.0570	225.3405	54,79
5	16	4	0.0550	224.7324	52,92
6	32	4	0.0550	180.0812	28,24

Wariant	BlockSize	K	Czas [s]	GFLOP/s	FLOP/B
7	8	16	0.0580	178.5042	10,21
8	16	16	0.0530	183.3472	69,86
9	32	16	0.0560	107.4548	31,5

Tabela 3. N = 1000, R = 16

Wariant	BlockSize	K	Czas [s]	GFLOP/s	FLOP/B
0 - Sekwencyjny	-	-	1.6070	0.6350	-
1	8	1	0.0610	255.7616	30.11
2	16	1	0.0600	280.6606	190.77
3	32	1	0.0590	190.0318	22.49
4	8	4	0.0620	249.2368	202.73
5	16	4	0.0560	217.7881	111.71
6	32	4	0.0630	178.8893	11.73
7	8	16	0.0620	185.7848	21.79
8	16	16	0.0630	185.1505	10.65
9	32	16	0.0670	90.6231	19.88

Tabela 4. N = 1500, R = 2

Wariant	BlockSize	K	Czas [s]	GFLOP/s	FLOP/B
0 - Sekwencyjny	-	-	0.0570	0.9816	-
1	8	1	0.0530	164.2817	6,21
2	16	1	0.0570	127.9510	6,21
3	32	1	0.0590	117.8042	4,8
4	8	4	0.0620	157.6885	3,58
5	16	4	0.0620	164.5910	4,63
6	32	4	0.0550	125.2561	5,11
7	8	16	0.0530	123.8104	5,08
8	16	16	0.0580	134.6204	4,5
9	32	16	0.0540	108.3369	4,72

Tabela 5. N = 1500, R = 8

Wariant	BlockSize	K	Czas [s]	GFLOP/s	FLOP/B
0 - Sekwencyjny	-	-	0.9840	0.6468	-
1	8	1	0.0570	278.2980	54.49
2	16	1	0.0560	279.3534	13.75
3	32	1	0.0640	175.1682	13.38
4	8	4	0.0620	218.6267	56.68
5	16	4	0.0590	238.9543	56.07

Wariant	BlockSize	K	Czas [s]	GFLOP/s	FLOP/B
6	32	4	0.0630	171.7289	50.91
7	8	16	0.0630	175.7471	52.28
8	16	16	0.0620	190.9478	15.37
9	32	16	0.0610	144.9877	37.07

Tabela 6. N = 1500, R = 16

Wariant	BlockSize	K	Czas [s]	GFLOP/s	FLOP/B
0 - Sekwencyjny	-	-	3.7200	0.6309	-
1	8	1	0.0610	280.2225	36.79
2	16	1	0.0630	278.4902	41.09
3	32	1	0.0660	205.2083	18.04
4	8	4	0.0610	277.4199	50.42
5	16	4	0.0640	283.6924	50.68
6	32	4	0.0650	209.5466	41.49
7	8	16	0.0660	195.4095	33.50
8	16	16	0.0680	212.5732	30.62
9	32	16	0.0680	162.9650	34.75

Tabela 7. N = 2000, R = 2

Wariant	BlockSize	K	Czas [s]	GFLOP/s	FLOP/B
0 - Sekwencyjny	-	-	0.1040	0.9577	-
1	8	1	0.0590	173.8542	3,6
2	16	1	0.0690	168.9746	4,7
3	32	1	0.0610	86.5260	3,27
4	8	4	0.0610	180.2370	2,21
5	16	4	0.0580	177.5332	4,67
6	32	4	0.0600	121.2746	4,9
7	8	16	0.0620	138.9515	1,85
8	16	16	0.0590	121.1236	3,39
9	32	16	0.0670	95.9497	2,83

Tabela 8. N = 2000, R = 8

Wariant	BlockSize	K	Czas [s]	GFLOP/s	FLOP/B
0 - Sekwencyjny	-	-	1.7440	0.6523	-
1	8	1	0.0650	277.5946	35.16
2	16	1	0.0640	253.6790	54.13
3	32	1	0.0620	181.7208	49.61
4	8	4	0.0610	277.1098	53.98

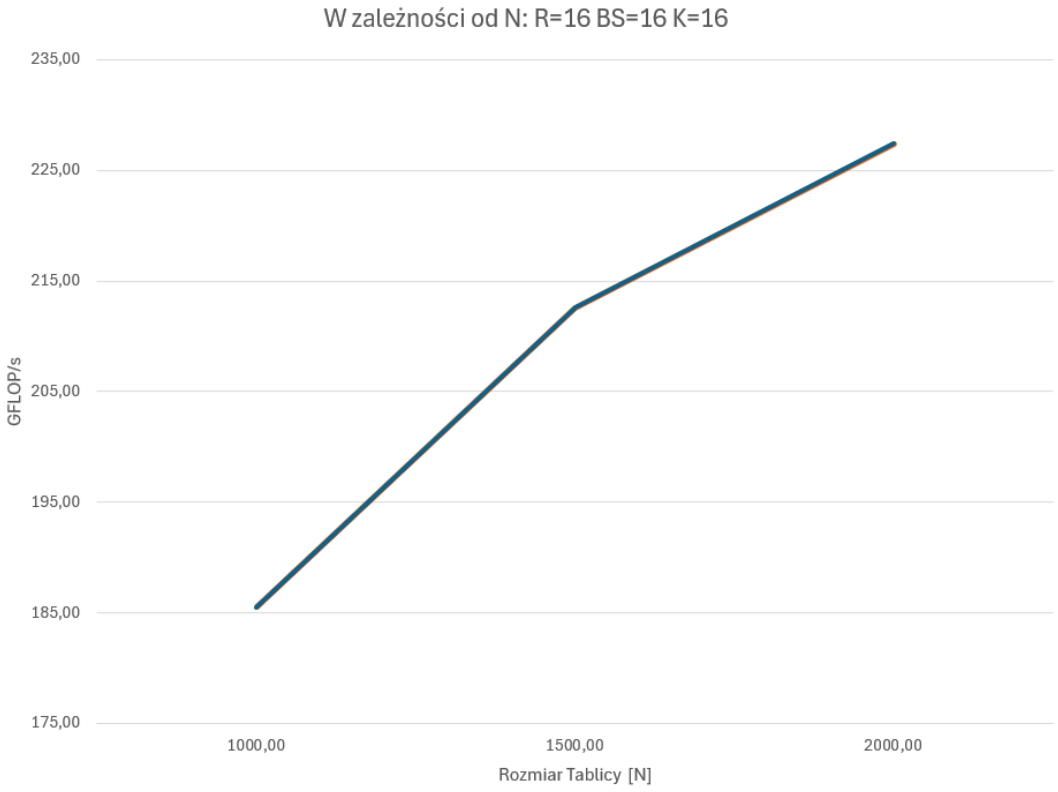
Wariant	BlockSize	K	Czas [s]	GFLOP/s	FLOP/B
5	16	4	0.0600	282.2315	28.39
6	32	4	0.0630	217.4602	14.49
7	8	16	0.0630	198.5152	53.92
8	16	16	0.0630	232.4457	54.37
9	32	16	0.0630	173.1191	25.03

Tabela 9. N = 2000, R = 16

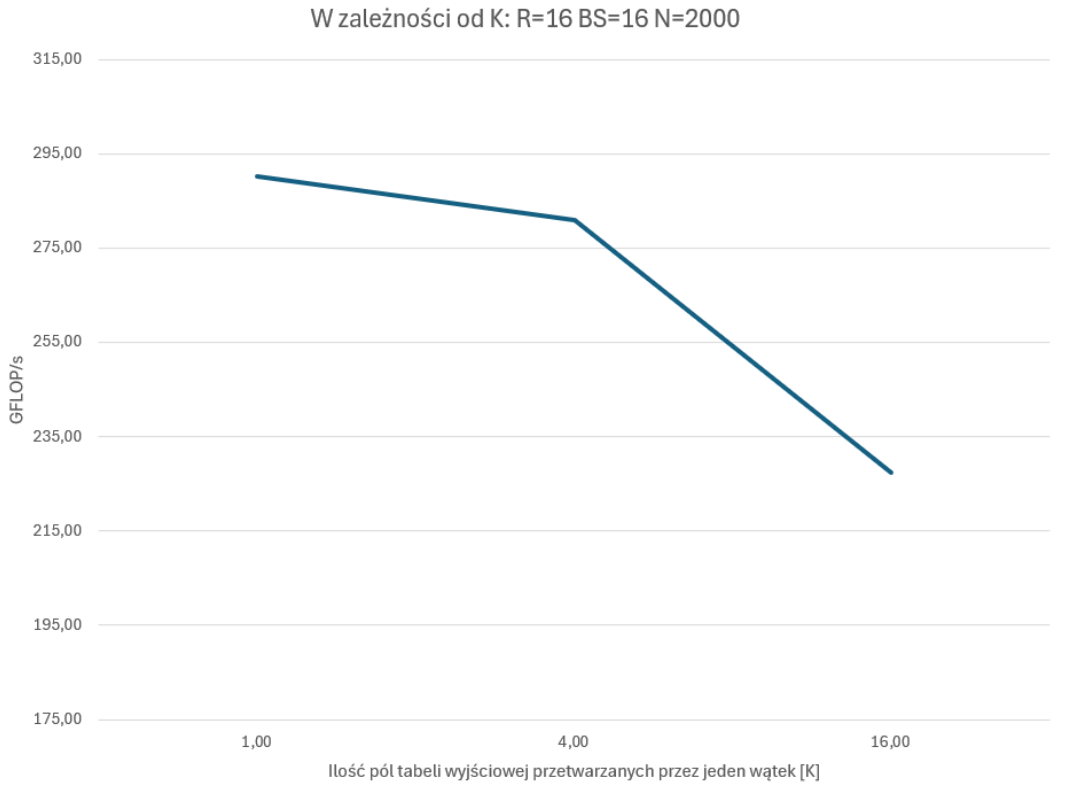
Wariant	BlockSize	K	Czas [s]	GFLOP/s	FLOP/B
0 - Sekwencyjny	-	-	6.6810	0.6313	-
1	8	1	0.0710	289.6552	42.98
2	16	1	0.0740	290.2676	32.79
3	32	1	0.0750	212.3841	41.42
4	8	4	0.0700	279.4615	54.73
5	16	4	0.0730	280.8640	25.74
6	32	4	0.0750	213.1179	48.22
7	8	16	0.0810	215.9682	38.96
8	16	16	0.0780	227.3868	42.44
9	32	16	0.0850	181.3594	36.27

Wykresy

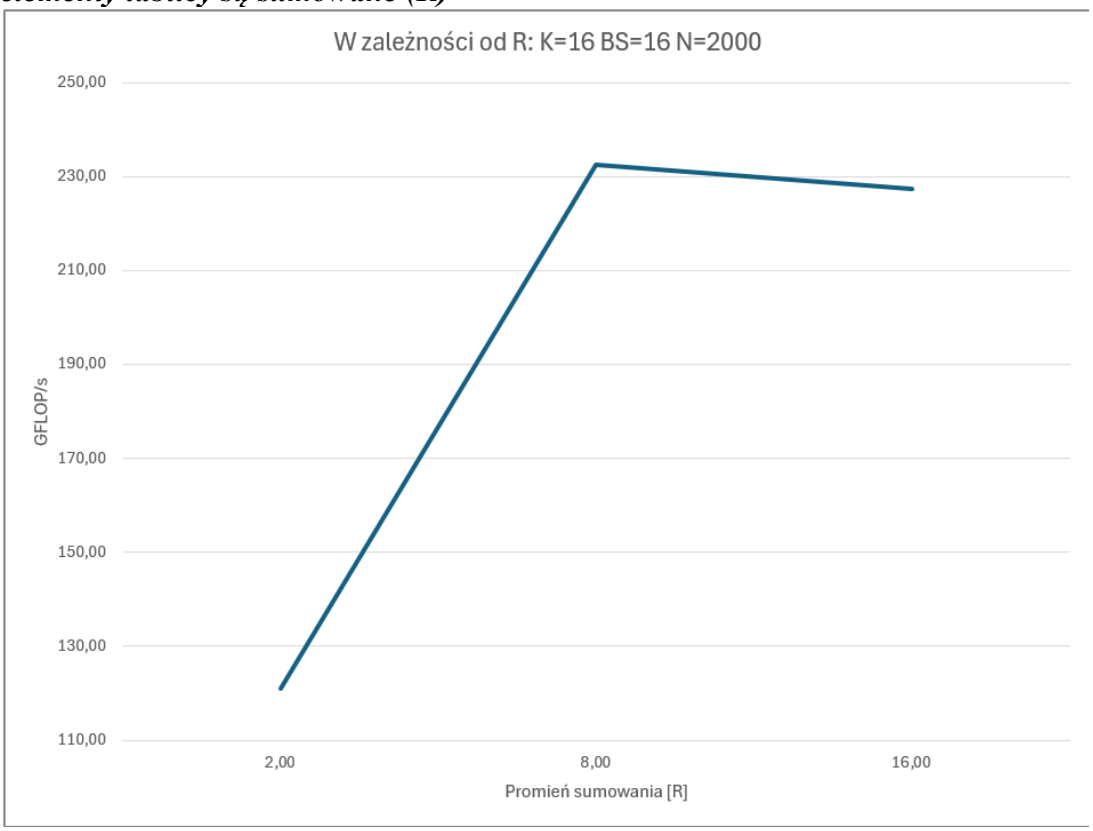
Wykres 1. Miara wydajności obliczeniowej w zależności od wymiaru tablicy wejściowej (N)



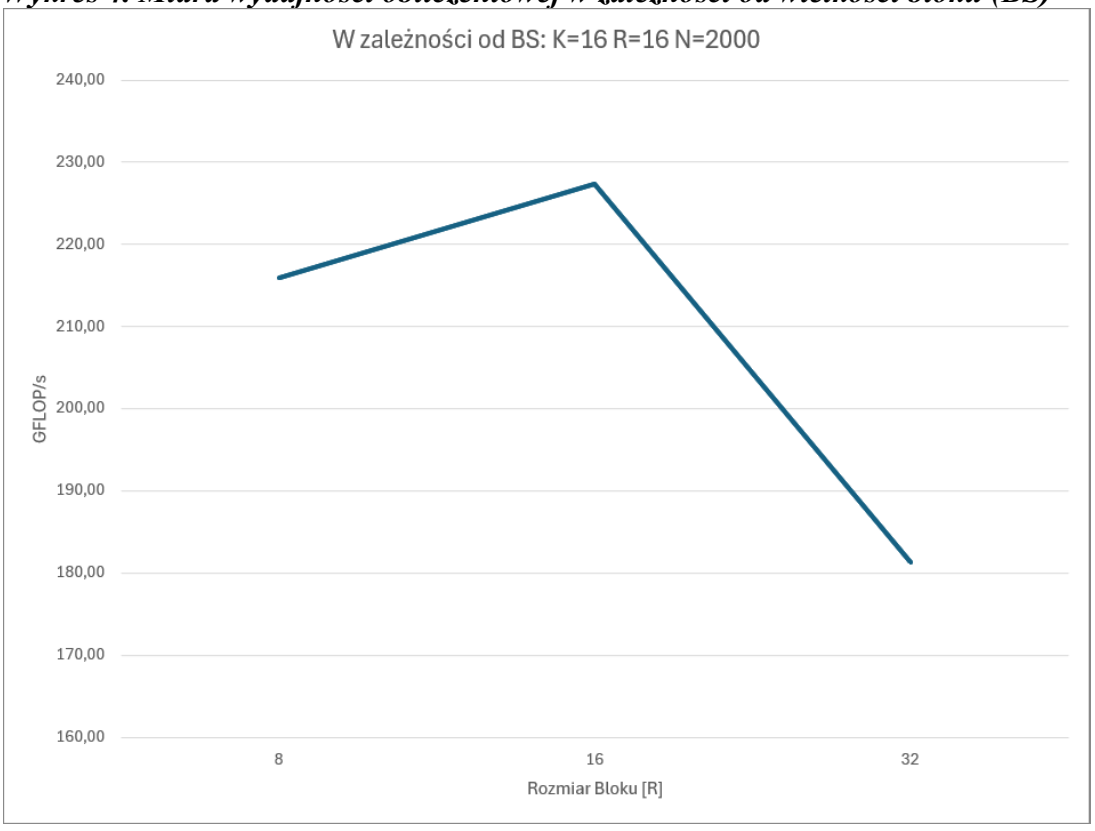
Wykres 2. Miara wydajności obliczeniowej w zależności od liczby wyników obliczanych przez jeden wątek (K)



Wykres 3. Miara wydajności obliczeniowej w zależności od promienia w jakim elementy tablicy są sumowane (R)



Wykres 4. Miara wydajności obliczeniowej w zależności od wielkości bloku (BS)



Wnioski

Wpływ parametrów na czas wykonywania obliczeń

Czas wykonywania obliczeń był zależny od użytych w eksperymencie parametrów. Największy wpływ miał parametr R, czyli promień w jakim wykonywane były obliczenia. Zauważyliśmy, że dla małej wartości R wyniki eksperymentu były nieprzewidywalne. Spowodowane jest to najprawdopodobniej małym wykorzystaniem mocy obliczeniowej używanej karty graficznej. Przy zwiększeniu promienia, czas wykonywania obliczeń zdecydowanie się wydłużał. Różnica jest najlepiej widoczna dla algorytmu sekwencyjnego, który przy dużej ilości operacji do wykonania zdecydowanie spowalniał. Algorytm wykonujący obliczenia równolegle również znacząco spowolnił. Jest to spowodowane dużo większą liczbą operacji sumowania do wykonania nawet przy niewielkim zwiększeniu promienia.

Kolejnym parametrem, który wpływał na wydłużenie wykonywania zadania jest liczba wyników obliczanych przez jeden wątek (parametr K). Jego zwiększenie również powoduje wydłużenie czasu wykonywania obliczeń. Powodowane jest to wzrostem liczby operacji, jakie musi wykonać każdy wątek, co prowadzi do zwiększenia czasu wykonania z powodu niekorzystnych wzorców dostępu do pamięci oraz zwiększonych opóźnień synchronizacji.

Podobną tendencję zauważyć można przy zmianie wielkości bloku. Im wyższa wartość BS, tym dłuższy czas wykonywania obliczeń. Jedynie w niektórych przypadkach blok o wielkości 16×16 pozwalał na wykonanie obliczeń w najkrótszym czasie (Tabela 1. Wariant 8; Tabela 2. Wariant 8; Tabela 5. Wariant 5; Tabela 8 Wariant 5; Tabela 9. Wariant 8).

Podsumowanie wyników

Badanie analizowało wydajność kernela CUDA przy różnych rozmiarach bloków (BS), liczbie wyników obliczanych przez wątki (K) dla macierzy wejściowej o rozmiarze (N) o wartościach $1000, 1500, 2000$ oraz różnych rozmiarach promienia w jakim wykonywane były obliczenia (R). Najlepsze wyniki pod względem wydajności ($GFLOP/s$) uzyskano dla małych i średnich bloków ($BS = 8$ lub $BS = 16$) z małą liczbą wyników na wątek ($K = 1$). Dla większych wartości R czas wykonywania obliczeń znacznie się wydłużał. Było to spowodowane zdecydowanie większą ilością operacji, które wątek musiał wykonać.

W ***Tabeli 1.*** możemy zauważyć, że algorytm sekwencyjny jest 2.27 razy szybszy od algorytmu wykonującego obliczenia równolegle. W ***Tabeli 4.*** następuje podobna sytuacja. Podejście sekwencyjne jest tam wykonywane w podobnym czasie, co najszybsze podejście równoległe. W pozostałych przypadkach możemy zauważyć, że podejście równoległe z wykorzystaniem CUDA znacznie przewyższa podejście sekwencyjne pod względem wydajności i czasu wykonania. Wydajność obliczeń w GFLOP/s również była znacznie wyższa. Kluczowym czynnikiem sukcesu był podział pracy na wątki oraz efektywna koalescencja dostępu do pamięci globalnej.