Projekt 2 GPU

Laboratorium z przetwarzania równoległego

Mateusz Kreczmer 151736

mateusz.kreczmer@student.put.poznan.pl

Piotr Krzyszowski 151909

piotr.krzyszowski@student.put.poznan.pl

Grupa L4, czwartek 16:50, tygodnie nieparzyste

Politechnika Poznańska

Informatyka, WIiT, semestr VI

Wymagany termin oddania: 31.05.2024

Rzeczywisty termin oddania: 31.05.2024

Wersja sprawozdania: 1.0

Opis zadania:

Celem projektu jest praktyczne zapoznanie się z zasadami programowania równoległego dla procesorów kart graficznych (PKG). Projekt obejmuje także naukę optymalizacji kodu dla PKG oraz ocenę prędkości przetwarzania przy użyciu tych procesorów. Dodatkowo, projekt ma na celu zrozumienie czynników wpływających na efektywność przetwarzania na GPU.

Opis użytej karty graficznej

Nazwa modelu karty: NVIDIA GeForce GTX 1650 SUPER

Układ scalony: TU116

Technologia: Architektura Turing

Parametr CC: CC 7.5

Liczba jednostek wykonawczych (SM): 22

Liczba rdzeni SP (jednostek zmiennoprzecinkowych pojedynczej precyzji): 1408

Liczba rdzeni DP (jednostek zmiennoprzecinkowych podwójnej precyzji): 0

Liczba jednostek całkowitoliczbowych: 88

Liczba jednostek SFU: 22

Pamięć karty graficznej: 4 GB pamięci GDDR6 z interfejsem 128-bitowym

Ograniczenia parametru CC: 7.5

Testy

1. Generowanie wartości testowych (tablica wejściowa):

Wartości testowe są generowane losowo przy użyciu funkcji rand() oraz są normalizowane do przedziału [0, 1] poprzez podzielenie przez RAND_MAX.

```
// Inicjalizacja danych wejściowych na CPU
for (int i = 0; i < N * N; ++i) {
    inputHost[i] = static_cast<float>(rand()) / RAND_MAX;
}
```

Kod 1. Generowanie wartości testowych

2. Testowanie poprawności obliczeń:

Do tego celu stworzona została funkcja computeSquential, która odpowiada za sekwencyjne wykonanie obliczeń na CPU dla każdego elementu tablicy wejściowej. Wyniki obliczeń CPU są później porównywane z wynikami obliczeń GPU w celu weryfikacji poprawności obliczeń.

```
void computeSequential(float* input, float* output, int N, int R) {
    for (int i = R; i < N - R; ++i) {
        for (int j = R; j < N - R; ++j) {
            float sum = 0.0f;
            for (int di = -R; di <= R; ++di) {
                for (int dj = -R; dj <= R; ++dj) {
                    sum += input[(i + di) * N + (j + dj)];
                }
            }
            output[(i - R) * (N - 2 * R) + (j - R)] = sum;
        }
    }
}
```

Kod 2. Testowanie poprawności obliczeń

3. Omówienie wyników testu poprawności:

Wynik testu poprawności jest omawiany na podstawie maksymalnego błędu pomiędzy wynikami obliczeń CPU i GPU. Jeśli maksymalny błąd jest mniejszy niż ustalona wartość graniczna (np. 1e-5), to obliczenia GPU są uznawane za poprawne. W przeciwnym przypadku, jeśli maksymalny błąd przekracza wartość graniczną, zostaje wyświetlony komunikat o błędzie.

```cpp
// Sprawdzenie poprawności obliczeń GPU poprzez porównanie z wynikami CPU
float maxError = 0.0f;
for (int i = 0; i < (N - 2 * R) * (N - 2 * R); ++i) {
    maxError = fmax(maxError, fabs(outputHostCPU[i] - outputHostGPU[i]));
}

if (maxError < 1e-5) {
    std::cout << "Poprawnosc obliczen GPU zostala zweryfikowana." << std::endl;
}
else {
    std::cout << "Blad obliczeń GPU! Maksymalny błąd: " << maxError << std::endl;
}
```

Kod 3. Wynik testu poprawności

Przeprowadzone analizy

W ramach zadania przeprowadzono kompleksową analizę wydajności przetwarzania danych na procesorze graficznym (GPU) w porównaniu z obliczeniami sekwencyjnymi wykonywanymi na procesorze centralnym (CPU). Głównym celem było zrozumienie zasad programowania równoległego na GPU oraz optymalizacja kodu pod kątem efektywności obliczeń. Przygotowano kod w dwóch wersjach: sekwencyjnej na CPU oraz równoległej na GPU przy użyciu CUDA. Po weryfikacji poprawności działania kodu poprzez porównanie wyników uzyskanych z obu wersji, przeprowadzono serię eksperymentów mających na celu ocenę prędkości przetwarzania oraz identyfikację czynników wpływających na wydajność. Analizy objęły zmienne takie jak rozmiar tablicy N, promień R, liczba wyników na wątek k oraz rozmiar bloku wątków BS.

Wyniki eksperymentów

1. Badanie nasycenia obliczeniami:

Eksperymenty obliczeniowe były przeprowadzane dla różnych wartości rozmiaru tablicy N przy stałej wartości parametrów R=1 i k=1 oraz przy zmieniającej się wartości BS, aby zidentyfikować punkt nasycenia obliczeniami, tj. wartość N, przy której dalsze zwiększanie rozmiaru danych nie prowadziło do wzrostu prędkości przetwarzania. Testy rozpoczęto od wartości N = 128, stopniowo zwiększając je o 128 do maksymalnej wartości równej 1536, przy której GPU mogło jeszcze efektywnie przetwarzać dane. Wyniki wykazały, że dla wartości N powyżej pewnego progu (określonego jako N_nas), prędkość obliczeń osiąga plateau, a dalsze zwiększanie N nie przynosiło zauważalnych korzyści wydajnościowych.

Dla BS = (8,8):

N	Czas obliczeń CPU [s]	Czas obliczeń GPU [s]	FLOP/s (GPU)
128	0.0004468	0.000135008	1.06e+09
256	0.0017602	0.000182688	3.18e+09
384	0.0040412	0.000400256	3.28e+09
512	0.0067372	0.000701952	3.33e+09
640	0.0110828	0.0010905	3.36e+09
768	0.015258	0.00156634	3.37e+09
896	0.0202126	0.00212653	3.38e+09
1024	0.02835	0.00277709	3.38e+09
1152	0.0334774	0.00351062	3.39e+09
1280	0.0428278	0.00433018	3.39e+09
1408	0.0504266	0.00523878	3.40e+09
1536	0.0594658	0.00623565	3.40e+09

Tabela 1. Nasycenie obliczeniami dla BS = (8,8)

Punkt nasycenia obliczeniami występuje przy wartości N ≈ 896, ponieważ prędkość obliczeń (FLOP/s) stabilizuje się w okolicach 3.38e+09.

Dla BS = (16,16):

N	Czas obliczeń CPU [s]	Czas obliczeń GPU [s]	FLOP/s (GPU)
128	0.0004799	0.000132736	1.08e+09
256	0.0017279	0.000182272	3.19e+09
384	0.0039731	0.000398016	3.30e+09
512	0.0071376	0.000700032	3.34e+09
640	0.0109655	0.00108838	3.37e+09
768	0.0161721	0.00156544	3.37e+09
896	0.0222954	0.0021263	3.38e+09
1024	0.0286779	0.0027768	3.39e+09
1152	0.0384732	0.00350899	3.39e+09
1280	0.0460187	0.00432742	3.40e+09
1408	0.0523105	0.00523824	3.40e+09
1536	0.0650619	0.00623203	3.40e+09

Tabela 2. Nasycenie obliczeniami dla BS = (16,16)

Punkt nasycenia obliczeniami występuje przy wartości N ≈ 768, ponieważ prędkość obliczeń (FLOP/s) stabilizuje się w okolicach 3.37e+09.

Dla BS = (32, 32):

N	Czas obliczeń CPU [s]	Czas obliczeń GPU [s]	FLOP/s (GPU)
128	0.0004761	0.000144384	9.90e+08
256	0.0018509	0.000192512	3.02e+09
384	0.0038386	0.00041984	3.13e+09
512	0.0071903	0.0007424	3.15e+09
640	0.0112004	0.00113971	3.21e+09
768	0.0157489	0.00164698	3.21e+09
896	0.0220123	0.00223334	3.22e+09
1024	0.0267485	0.00292282	3.22e+09
1152	0.0345386	0.00369107	3.22e+09
1280	0.041243	0.00455168	3.23e+09
1408	0.0501754	0.00551085	3.23e+09
1536	0.0600896	0.00655648	3.23e+09

Tabela 3. Nasycenie obliczeniami dla BS = (32,32)

Punkt nasycenia obliczeniami występuje przy wartości N ≈ 640, ponieważ prędkość obliczeń (FLOP/s) stabilizuje się w okolicach 3.21e+09.

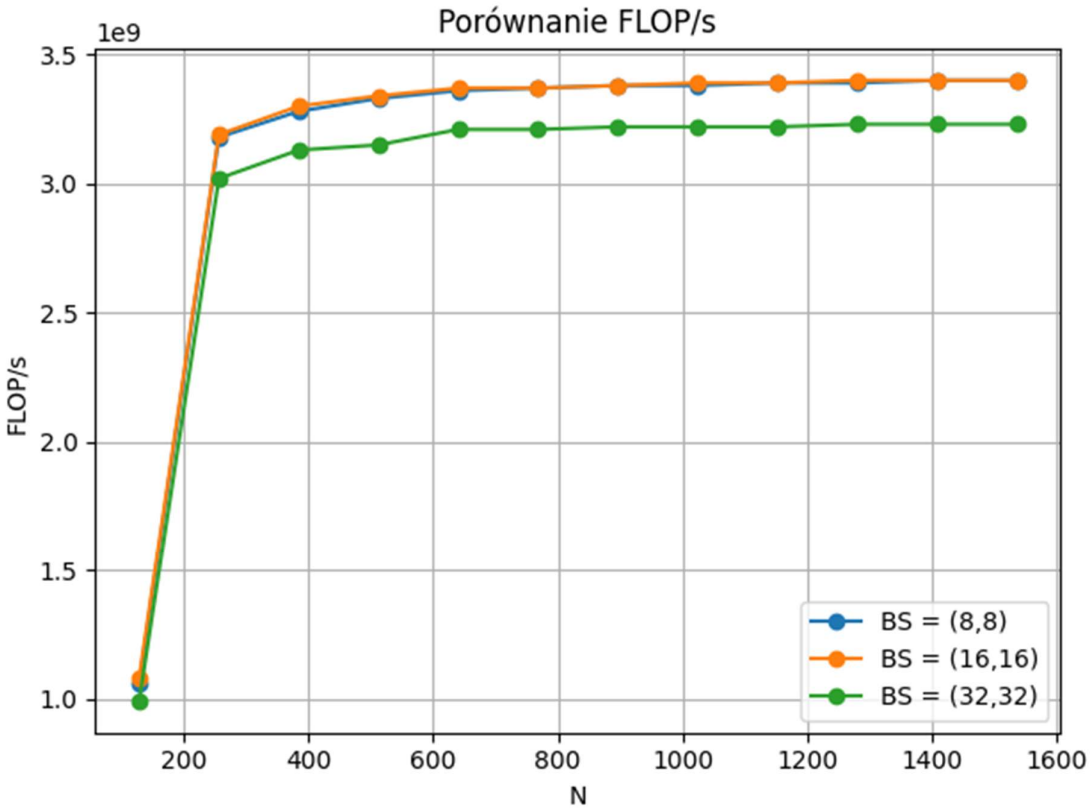


Figure 1. Porównanie FLOP/s

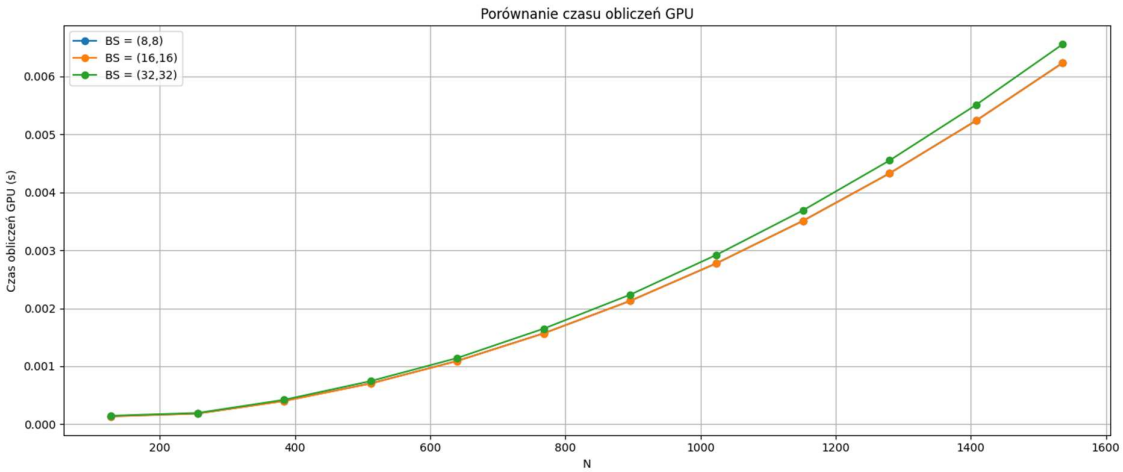


Figure 2. Porównanie czasu obliczeń GPU

Wyniki pokazują, że punkt nasycenia obliczeniami zależy od rozmiaru bloku wątków (BS). Mniejsze bloki (8,8) prowadzą do nasycenia przy większych wartościach N, podczas gdy większe bloki (32,32) osiągają nasycenie przy mniejszych wartościach N. Oznacza to, że rozmiar bloku wątków ma istotny wpływ na wydajność obliczeniową algorytmu na GPU.

2. Wpływ wartości parametrów k i R na prędkość obliczeń:

Dla macierzy większych od zapewniających nasycenie obliczeniami, przeprowadzono eksperymenty mające na celu zbadanie wpływu parametrów k i R na prędkość obliczeń. Wartości k oraz R były kolejno zwiększane według wzoru 2, 4, 8, 16... do momentu uzyskania maksymalnej wartości prędkości obliczeń. Wzrost wartości k i R powodował adekwatny wzrost wielkości instancji problemu zgodnie ze wzorem: N(k, R) = (N_nas - 2) * k + 2 * R. Podczas tych eksperymentów wielkość gridu przy stałej wielkości bloku pozostawała niezmieniona, co pozwoliło na analizę samego wpływu parametrów k i R bez dodatkowych zmiennych.

Dla BS = (8,8):

N	k	R	Czas obliczeń GPU [s]	FLOP/s (GPU)
896	2	2	0.00805069	2.47e+09
896	4	4	0.0412693	1.55e+09
896	8	8	0.256488	8.73e+08
896	16	16	1.63234	4.98e+08
1024	2	2	0.0103311	2.52e+09
1024	4	4	0.0540145	1.55e+09
1024	8	8	0.339953	8.64e+08
1024	16	16	2.28945	4.68e+08
1152	2	2	0.147413	2.24e+08
1152	4	4	0.147635	7.18e+08
1152	8	8	0.60352	6.18e+08
1152	16	16	2.89537	4.72e+08
1280	2	2	0.181941	2.24e+08
1280	4	4	0.235267	5.57e+08
1280	8	8	0.734007	6.29e+08
1280	16	16	3.5226	4.81e+08
1408	2	2	0.195972	2.51e+08
1408	4	4	0.297304	5.34e+08
1408	8	8	0.741647	7.55e+08
1408	16	16	4.15117	4.97e+08
1536	2	2	0.106219	5.52e+08
1536	4	4	0.196376	9.63e+08
1536	8	8	0.789927	8.45e+08

1536	16	16	4.88968	3.75e+08

Tabela 4. Wartości eksperymentu dla zmienianych R i k dla BS = (8,8)

Dla BS = (16,16):

N	k	R	Czas obliczeń GPU [s]	FLOP/s (GPU)
768	2	2	0.00592054	2.46e+09
768	4	4	0.0305978	1.53e+09
768	8	8	0.192457	8.49e+08
768	16	16	1.19556	4.93e+08
896	2	2	0.00790515	2.52e+09
896	4	4	0.0415876	1.54e+09
896	8	8	0.2602	8.60e+08
896	16	16	1.6191	5.02e+08
1024	2	2	0.11639	2.23e+08
1024	4	4	0.235499	3.55e+08
1024	8	8	0.527342	5.57e+08
1024	16	16	2.33416	4.59e+08
1152	2	2	0.147188	2.24e+08
1152	4	4	0.158589	6.68e+08
1152	8	8	0.509798	7.32e+08
1152	16	16	2.81625	4.85e+08
1280	2	2	0.181653	2.24e+08
1280	4	4	0.23919	5.48e+08
1280	8	8	0.594478	7.77e+08
1280	16	16	3.4835	4.87e+08
1408	2	2	0.185991	2.65e+08
1408	4	4	0.30366	5.23e+08
1408	8	8	0.751922	7.45e+08
1408	16	16	4.25328	4.85e+08
1536	2	2	0.100662	5.83e+08
1536	4	4	0.194528	9.72e+08
1536	8	8	0.819178	8.15e+08
1536	16	16	5.01944	3.65e+08

Tabela 5. Wartości eksperymentu dla zmienianych R i k dla BS = (16,16)

Dla BS = (32, 32):

N	k	R	Czas obliczeń GPU [s]	FLOP/s (GPU)
640	2	2	0.00424141	2.38e+09
640	4	4	0.0212988	1.52e+09
640	8	8	0.140229	8.02e+08
640	16	16	0.843622	4.77e+08
768	2	2	0.00474726	3.07e+09
768	4	4	0.0307907	1.52e+09
768	8	8	0.199976	8.17e+08
768	16	16	1.18949	4.96e+08
896	2	2	0.0901385	2.21e+08
896	4	4	0.214884	2.97e+08
896	8	8	0.347217	6.45e+08
896	16	16	1.74212	4.67e+08
1024	2	2	0.118034	2.20e+08
1024	4	4	0.138103	6.05e+08
1024	8	8	0.52614	5.58e+08
1024	16	16	2.3297	4.60e+08
1152	2	2	0.149359	2.21e+08
1152	4	4	0.15233	6.96e+08
1152	8	8	0.512232	7.28e+08
1152	16	16	2.79412	4.89e+08
1280	2	2	0.184152	2.21e+08
1280	4	4	0.240978	5.44e+08
1280	8	8	0.643875	7.17e+08
1280	16	16	3.4925	4.86e+08
1408	2	2	0.185509	2.66e+08
1408	4	4	0.303023	5.24e+08
1408	8	8	0.745984	7.51e+08
1408	16	16	4.22777	4.88e+08
1536	2	2	0.0961758	6.10e+08
1536	4	4	0.200856	9.42e+08
1536	8	8	0.842537	7.92e+08
1536	16	16	4.90629	3.73e+08

Tabela 6. Wartości eksperymentu dla zmienianych R i k dla BS = (32,32)

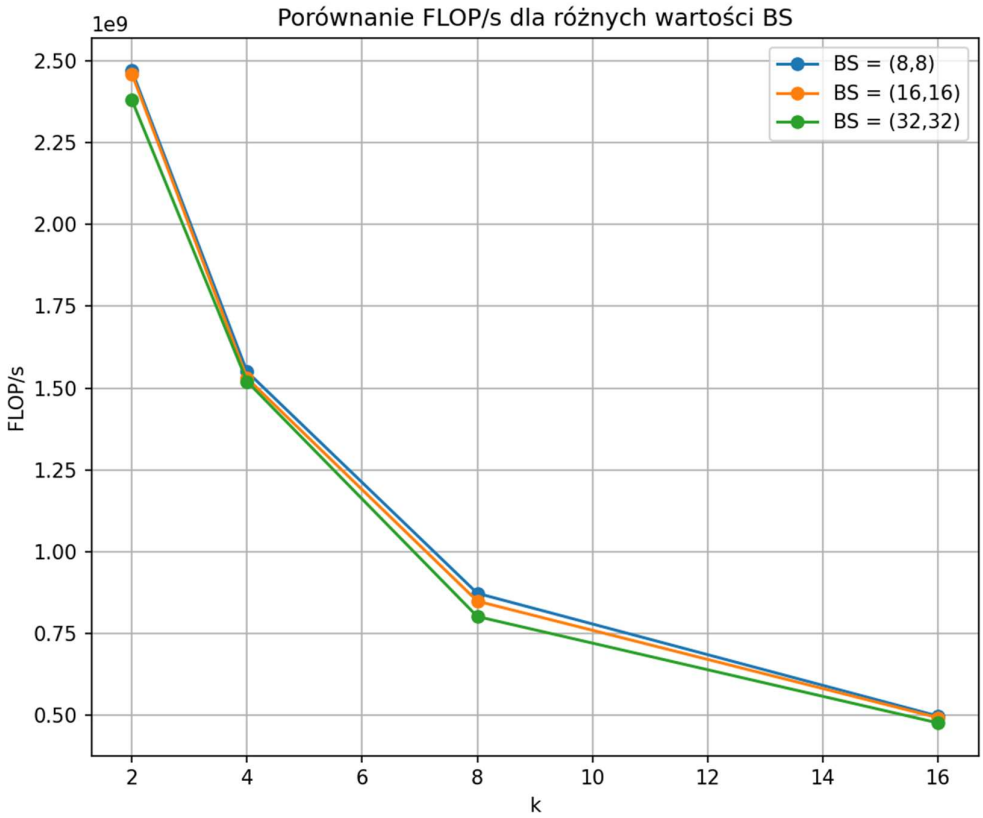


Figure 3. Porównanie FLOP/s dla różnych wartości BS i parametru k

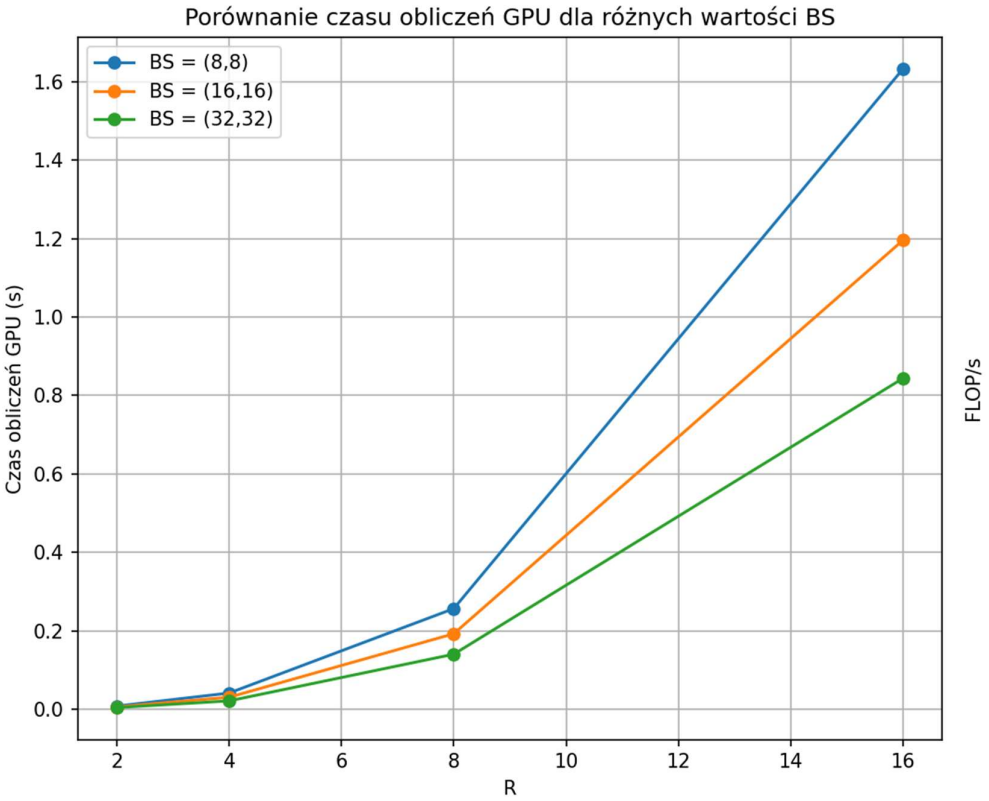


Figure 4. Porównanie czasu obliczeń dla różnych wartości BS i parametru R

Analiza wpływu wartości parametrów R i k na czas obliczeń GPU oraz wydajność (FLOP/s) dla różnych rozmiarów bloków wątków (BS) wykazała, że mniejsze wartości tych parametrów prowadzą do niższego czasu obliczeń i wyższej wydajności. Najlepszą wydajność osiągnięto dla najmniejszych wartości R i k, zwłaszcza przy większych rozmiarach bloków (BS = 32,32). Wraz ze wzrostem R i k, zarówno czas obliczeń, jak i wydajność znacząco się pogarszały. Wyniki te sugerują, że dla optymalnej wydajności obliczeń na GPU warto utrzymywać niskie wartości parametrów R i k, szczególnie w kontekście większych bloków wątków, aby maksymalizować efektywność przetwarzania.

Kluczowe fragmenty kodów kernela

1. Funkcja ‘computeOutput’:

Znaczenie użytych instrukcji i zmiennych:

- blockIdx, blockDim, threadIdx: CUDA zmienne predefiniowane używane do identyfikacji indeksów wątków w gridzie.
- i, j: Współrzędne globalne obliczane na podstawie indeksów bloku i wątku.
- sum: Akumulator do sumowania wartości z sąsiedztwa promienia R.

Jakość dostępów do pamięci:

- Kod używa łączenia dostępów do pamięci globalnej, odwołując się do sąsiadujących elementów tablicy input, co zwiększa efektywność dostępu.

```
__global__ void computeOutput(float* input, float* output, int N, int R, int k) {
    int i = blockIdx.y * blockDim.y + threadIdx.y + R;
    int j = blockIdx.x * blockDim.x + threadIdx.x + R;

    if (i < N - R && j < N - R) {
        for (int idx = 0; idx < k; ++idx) {
            float sum = 0.0f;
            for (int di = -R; di <= R; ++di) {
                for (int dj = -R; dj <= R; ++dj) {
                    sum += input[(i + di) * N + (j + dj)];
                }
            }
            output[(i - R) * (N - 2 * R) + (j - R) + idx * (N - 2 * R) * (N - 2 * R)] = sum;
        }
    }
}
```

Kod 4. Funkcja 'computeOutput'

2. Funkcja ‘computeSequential’:

Znaczenie użytych instrukcji i zmiennych:

- i, j: Indeksy wierszy i kolumn w tablicy wejściowej input.
- sum: Akumulator do sumowania wartości z sąsiedztwa promienia R.

Jakość dostępów do pamięci:

- Funkcja sekwencyjna dokonuje sumowania sąsiednich wartości dla każdego elementu wyjściowego, co jest odpowiednikiem działania kernela CUDA, ale bez równoległego przetwarzania.

```
void computeSequential(float* input, float* output, int N, int R) {
    for (int i = R; i < N - R; ++i) {
        for (int j = R; j < N - R; ++j) {
            float sum = 0.0f;
            for (int di = -R; di <= R; ++di) {
                for (int dj = -R; dj <= R; ++dj) {
                    sum += input[(i + di) * N + (j + dj)];
                }
            }
            output[(i - R) * (N - 2 * R) + (j - R)] = sum;
        }
    }
}
```

Kod 5. 2. Funkcja 'computeSequential'

3. Konfiguracja uruchomienia kernela:

Blok wątków o rozmiarze 16x16 zapewnia optymalny balans pomiędzy liczbą wątków a zasobami sprzętowymi, takimi jak pamięć współdzielona. Grid jest skonfigurowany tak, aby pokryć całą tablicę wyjściową, z uwzględnieniem promienia R.

```
dim3 blockSize(16, 16);
dim3 gridSize((N - 2 * R + blockSize.x - 1) / blockSize.x, (N - 2 * R + blockSize.y - 1) / blockSize.y);
```

Kod 6. Konfiguracja uruchomienia kernela

4. Pomiar czasu GPU i obliczenie FLOP/s:

Znaczenie użytych instrukcji i zmiennych:

- cudaEventCreate, cudaEventRecord, cudaEventElapsedTime: CUDA API do tworzenia i mierzenia czasu wykonania kernela.
- elapsedTime: Zmienna przechowująca czas wykonania kernela w milisekundach.
- numOps: Liczba operacji zmiennoprzecinkowych.
- flops: Liczba operacji zmiennoprzecinkowych na sekundę (FLOP/s).

Jakość dostępów do pamięci:

- Użycie wydarzeń CUDA pozwala na precyzyjne mierzenie czasu wykonania kernela, bez uwzględniania czasu komunikacji między CPU a GPU.
- Obliczenie FLOP/s pozwala na ocenę wydajności obliczeniowej kernela.

```
// Wywołanie kernela
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

computeOutput<<<gridSize, blockSize>>>(inputDevice, outputDevice, N, R, k);

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
std::cout << "GPU Time for N=" << N << ": " << elapsedTime / 1000.0f << "s\n";

// Kopiowanie wyników z GPU do CPU
cudaMemcpy(outputHostGPU, outputDevice, (N - 2 * R) * (N - 2 * R) * k * sizeof(float), cudaMemcpyDeviceToHost);

// Obliczanie FLOPS
int numOps = (N - 2 * R) * (N - 2 * R) * (2 * R + 1) * (2 * R + 1);
float flops = numOps / (elapsedTime / 1000.0f);
std::cout << "FLOP/s for N=" << N << ": " << flops << "\n";
```

Kod 7. Pomiar czasu GPU i obliczenie FLOP/s

5. Sprawdzenie poprawności wyników:

Znaczenie użytych instrukcji i zmiennych:

- maxError: Zmienna przechowująca największy błąd między wynikami obliczeń na CPU i GPU.
- fmax, fabs: Funkcje do obliczania wartości maksymalnej i wartości bezwzględnej różnicy.

Jakość dostępów do pamięci:

- Kod porównuje wyniki obliczeń sekwencyjnych i równoległych, sprawdzając poprawność implementacji GPU.

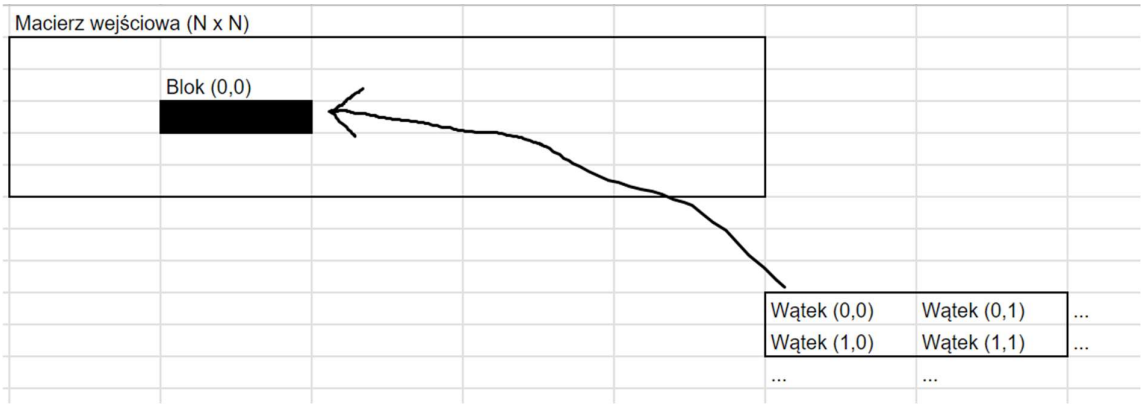
```
// Sprawdzenie poprawności obliczeń GPU poprzez porównanie z wynikami CPU
float maxError = 0.0f;
for (int i = 0; i < (N - 2 * R) * (N - 2 * R); ++i) {
    maxError = fmax(maxError, fabs(outputHostCPU[i] - outputHostGPU[i]));
}

if (maxError < 1e-5) {
    std::cout << "Poprawnosc obliczen GPU zostala zweryfikowana." << std::endl;
}
else {
    std::cout << "Blad obliczeń GPU! Maksymalny błąd: " << maxError << std::endl;
}
```

Kod 8. Sprawdzenie poprawności wyników

Rysunkowe zobrazowanie działania kernela

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

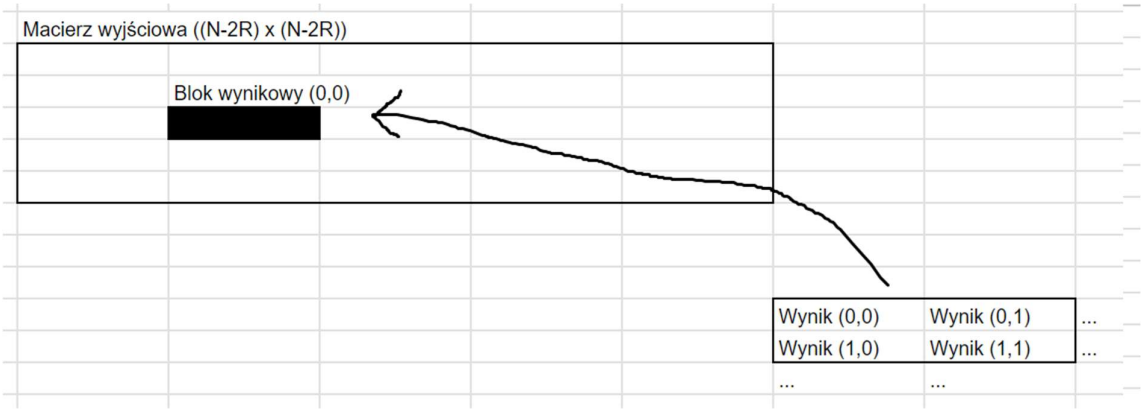
Rysunek przedstawia dwuwymiarową macierz danych input o rozmiarze N x N. Każdy blok wątków przetwarza podmacierz o rozmiarze BS x BS, a każdy wątek w bloku odpowiada za przetwarzanie k-elementów wynikowych. Wątki wewnątrz bloku są zorganizowane w siatkę, w której każdy wątek sumuje wartości z promienia R wokół swojej pozycji (i, j) i zapisuje wynik w odpowiadającej mu pozycji w tablicy output. Kolejność dostępu jest sekwencyjna, wątki najpierw pobierają wartości z wierszy, a następnie z kolumn w promieniu R, co zapewnia efektywne łączenie dostępów do pamięci globalnej.



Rysunek 1. Miejsce dostępu i kolejność dostępu do danych wejściowych

2. Wyznaczane przez wątki i bloki wartości wyników:

Rysunek przedstawia wynikową macierz output o rozmiarze (N - 2R) x (N - 2R). Każdy element wynikowy jest sumą wartości z promienia R wokół odpowiadającej mu pozycji w macierzy input. Wątki w bloku pracują równolegle, każdy zapisując swoje wyniki niezależnie, co pozwala na równoległe obliczenia i szybkie przetwarzanie danych. Każdy blok wątków przetwarza podmacierz o rozmiarze BS x BS, a każdy wątek w bloku odpowiada za przetwarzanie k-elementów wynikowych, co jest analogiczne do działania w przypadku k=1, ale zwiększa efektywność przez równoczesne przetwarzanie wielu elementów przez pojedynczy wątek.



Rysunek 2. Miejsce dostępu i kolejność dostępu do danych wyjściowych

Rysunki te ilustrują, jak wątki w blokach przetwarzają dane wejściowe i zapisują wyniki, pokazując sekwencję dostępu do danych oraz strukturę wynikowej macierzy. W pierwszym rysunku pokazano dostęp do danych w macierzy input, a w drugim - wyznaczanie wartości wynikowych w macierzy output.

Zastosowane wzory

1. Czas obliczeń (Duration):

Czas obliczeń jest mierzony w sekundach i reprezentuje czas, jaki kernel GPU spędza na wykonaniu obliczeń, nie uwzględniając czasu komunikacji między CPU a GPU.

Wzór:

$$Duration = elapsedTime$$

gdzie:

elapsedTime - czas wykonania kernela mierzony przy użyciu funkcji CUDA cudaEventElapsedTime, wyrażony w sekundach.

2. Prędkość obliczeń (FLOP/s):

Prędkość obliczeń, czyli liczba operacji zmiennoprzecinkowych na sekundę (FLOP/s), określa, ile operacji arytmetycznych jest wykonywanych na jednostkę czasu. Jest to kluczowy wskaźnik wydajności obliczeń na GPU.

Wzór:

$$FLOP/s = \frac{numOps}{Duration}$$

gdzie:

numOps - liczba operacji arytmetycznych, obliczona jako $numOps = (N - 2R) \times (N - 2) \times (2R + 1) \times (2R + 1)$.

Duration - czas wykonania kernela w sekundach.

Podsumowanie

Projekt ten umożliwił praktyczne zapoznanie się z zasadami programowania równoległego na GPU, z naciskiem na optymalizację kodu i ocenę wydajności przetwarzania. Eksperymenty wykazały, że mniejsze wartości parametrów R i k prowadzą do lepszej wydajności obliczeniowej, szczególnie przy większych rozmiarach bloków wątków. Punkty nasycenia obliczeniami różniły się w zależności od rozmiaru bloku, wskazując na konieczność dostosowania tych parametrów dla optymalizacji. Wzrost wartości R i k powodował spadek wydajności, co sugeruje, że dla maksymalnej efektywności obliczeń na GPU warto utrzymywać te wartości na niskim poziomie. Wyniki te potwierdzają znaczenie dostosowywania parametrów algorytmu do specyfikacji sprzętu dla osiągnięcia najwyższej wydajności.