Laboratorium z
przetwarzania
równoległego - CUDA

Autorzy:
Ahmed Osman, nr indeksu 155986, ahmed.osman@student.put.poznan.pl
Dawid Wnukiewicz, nr indeksu 155858, dawid.wnukiewicz@student.put.poznan.pl
Dla obu autorów: grupa dziekańska L5, termin zajęć laboratoryjnych środy nieparz. godz. 9:45 – 11:15

Wymagany termin oddania sprawozdania: 07.06.2025

1 wersja sprawozdania

1. Opis zakresu realizowanego zadania

W ramach projektu przeprowadziliśmy analizę procesu mnożenia macierzy na procesorach kart graficznych z wykorzystaniem technologii CUDA. Projekt obejmował implementację i optymalizację algorytmów mnożenia macierzy kwadratowych w celu zwiększenia współczynnika CGMA i wydajności, a także otrzymanie optymalnego wykorzystania pamięci. Staraliśmy się wykorzystać pamięć współdzieloną dla redukcji dostępów do pamięci globalnej z wykorzystaniem techniki kafelkowania (tiling).
W sprawozdaniu analizujemy w jaki sposób zwiększenie liczby wyników obliczanych przez jeden wątek wpływa na parametry opisane wyżej.

2. Wykorzystane pojęcia

- Kafelek wyników - kafelek (tile) wyników zwracanych przez 1 wątek o rozmiarze $Rx \times Ry$, gdzie:
  - Rx – poziomy wymiar kafelka wyników
  - Ry – pionowy wymiar kafelka wyników
- Multiprocesor – struktura wykonawcza składająca się z wielu jednostek obliczeniowych t.j,rdzeni i jednostek przetwarzających specjalnego przeznaczenia (SFU)
- Kernel – funkcja – kod realizowany przez bloki wątków na multiprocesorach
- Blok wątków - jest to struktura maksymalnie trójwymiarowa, która jest zbudowana przez grupę wątków. Podczas swojej pracy blok realizuje funkcję kernel w ramach tylko jednego multiprocesora.
- Grid – jest to struktura maksymalnie dwuwymiarowa, składająca się z bloków wątków.
- Pamięć współdzielona - Jest to szybka pamięć, która jest współdzielona między wszystkimi wątkami w jednym bloku wątków.
- Pamięć globalna – Jest to wolniejsza pamięć, która jest współdzielona między wszystkimi wątkami w całym gridzie.
- Tiling (technika tilingu) - rodzaj podejścia w mnożeniu macierzy z wykorzystaniem GPU. To technika dzielenia macierzy na mniejsze bloki (tile), które są ładowane do współdzielonej pamięci, by zminimalizować dostęp do wolniejszej pamięci globalnej i zwiększyć wydajność.

3. Opis wykorzystanego środowiska do wykonywania obliczeń

Obliczenia i kody zostały uruchomione na maszynie z GPU za pośrednictwem komputera polluks i konta studenckiego w systemie Instytutu Informatyki PP.

Maszyna posiada dwie karty graficzne NVIDIA GeForce GTX 1080.
Do wyboru karty wykorzystujemy metodę helper findCudaDevice, która wybiera kartę najmniej obciążoną w danym momencie.

Parametry NVIDIA GeForce GTX 1080:

Architektura	Pascal
Compute Capability(C.C.)	6.1
Rozmiar pamięci globalnej (global memory)	8112 MB (8506114048 bajtów)
Rozmiar pamięci stałej (Constant Memory)	65536 bajtów
Liczba multiprocesorów (SM)	20
Liczba rdzeni CUDA	2560 (128 na multiprocesor)
Rozmiar pamięci podręcznej L2	2MB (2097152 bajtów)
Pamięć współdzielona dla całego procesora	98304 bajtów
Maksymalny rozmiar pamięci współdzielonej (dla jednego bloku)	49152 bajtów
Dostępna liczba rejestrów na blok wątków	65536
Rozmiar wiązki (warp)	32 wątki
Maksymalna liczba wątków na multiprocesor	2048
Maksymalna liczba wątków na blok	1024
Maksymalny rozmiar bloku wątków (x,y,z)	(1024, 1024, 64)
Maksymalny rozmiar gridu (x,y,z)	(2147483647, 65535, 65535)

Tabela nr 1 : Parametry użytej do eksperymentów karty graficznej NVIDIA GeForce GTX1080

4. Implementacja algorytmów

a. Ogólna koncepcja implementacji

Prezentowana implementacja to modyfikacja kodu od NVIDIA go do mnożenia macierzy, który opiera się na technice kafelkowania (tiling). Ta technika pozwala znacząco zredukować liczbę dostępów do pamięci globalnej oraz zwiększyć współczynnik CGMA (Compute to Global Memory Access). Kluczowym elementem zoptymalizowanym przez nas jest wykorzystanie pamięci współdzielonej w połączeniu ze strategią, gdzie pojedynczy wątek oblicza wiele sąsiadujących elementów macierzy wynikowej.

b. Struktura kernela i technika kafelkowania

Szablon kernela MatrixMulCUDA przyjmuje parametry RESULTS_PER_THREAD_X i RESULTS_PER_THREAD_Y określające wymiary kafla wyników przetwarzanego przez każdy wątek.

```
template <int RESULTS_PER_THREAD_X = 1, int RESULTS_PER_THREAD_Y = 1>
__global__ void MatrixMulCUDA(float *A, float *B, float *C, int size)
```

Listing nr 1 : Szablon kernela mnożącego macierze

Współrzędne bazowe w macierzy wynikowej

Każdy wątek najpierw oblicza swoje współrzędne bazowe w macierzy wynikowej:

```
  int baseRow = (blockIdx.y * blockDim.y + threadIdx.y) * RESULTS_PER_THREAD_Y;
  int baseCol = (blockIdx.x * blockDim.x + threadIdx.x) * RESULTS_PER_THREAD_X;
```

Listing nr 2 : Obliczanie współrzędnych bazowych macierzy wynikowej

Te wartości definiują lewy górny róg kafla wynikowego, za który odpowiedzialny jest dany wątek. Przykładowo, dla konfiguracji 4×4, każdy wątek będzie odpowiedzialny za obliczenie 16 sąsiadujących elementów macierzy C.

Wykorzystanie pamięci współdzielonej

Następnie kernel definiuje dwa obszary pamięci współdzielonej:

```
  __shared__ float As[BLOCK_SIZE * RESULTS_PER_THREAD_Y][BLOCK_SIZE];
  __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE * RESULTS_PER_THREAD_X];
```

Listing nr 3 : Definicja macierzy tymczasowych w pamięci współdzielonej

Te tablice przechowują fragmenty macierzy wejściowych, które są wielokrotnie wykorzystywane przez wątki w bloku. Macierz As ma zwiększoną wysokość, a macierz Bs zwiększoną szerokość, proporcjonalnie do liczby wyników przetwarzanych przez wątek.

Pętla kafelkowania i ładowanie danych

Mnożenie macierzy realizowane jest przez iterowanie po kafelkach o rozmiarze BLOCK_SIZE:

```
  // size – rozmiar macierzy
  int numTiles = (size + BLOCK_SIZE - 1) / BLOCK_SIZE;
  for (int m = 0; m < numTiles; ++m) {
    // (omówione poniżej) ładowanie kafelków macierzy A i B do pamięci
współdzielonej
    __syncthreads();
    // (omówione poniżej)  Obliczenia na danych z pamięci współdzielonej
    __syncthreads();
  }
```

Listing nr 4 : Pętla kafelkowania oraz synchronizacja wątków

Dyrektywa __syncthreads() jest barierą synchronizacyjną, która gwarantuje, że wszystkie wątki w bloku osiągną ten punkt wykonania kodu zanim którykolwiek z nich będzie mógł kontynuować dalsze wykonanie.

Pierwsza synchronizacja __syncthreads() w powyższym kodzie gwarantuje, że wszystkie wątki w bloku zakończyły ładowanie swoich części danych do pamięci współdzielonej, zanim którykolwiek z nich rozpocznie fazę obliczeń. Bez tej bariery synchronizacyjnej niektóre wątki mogłyby rozpocząć

obliczenia, próbując odczytać dane z pamięci współdzielonej, które nie zostały jeszcze zapisane przez inne wątki, co prowadziłoby do nieprawidłowych wyników.
Druga synchronizacja __syncthreads() zapewnia, że wszystkie wątki zakończyły korzystanie z bieżących danych w pamięci współdzielonej, zanim którykolwiek z nich rozpocznie wczytywanie nowych danych, co nadpisałoby istniejące wartości. Bez tej synchronizacji mogłoby dojść do wyścigu danych, gdzie jeden wątek próbuje czytać dane, a inny jednocześnie je modyfikuje.

Ładowanie kafelków macierzy A i B

```
// size – rozmiar macierzy
// tx - threadIdx.x
// ty - threadIdx.y
for (int y = 0; y < RESULTS_PER_THREAD_Y; ++y) {
   int row = baseRow + y;
   if (row < size && (m * BLOCK_SIZE + tx) < size) {
     As[ty * RESULTS_PER_THREAD_Y + y][tx] = A[row * size + m * BLOCK_SIZE + tx];
   } else {
     As[ty * RESULTS_PER_THREAD_Y + y][tx] = 0.0f;
   }
 }

for (int x = 0; x < RESULTS_PER_THREAD_X; ++x) {
   int col = baseCol + x;
   if ((m * BLOCK_SIZE + ty) < size && col < size) {
     Bs[ty][tx * RESULTS_PER_THREAD_X + x] = B[(m * BLOCK_SIZE + ty) * size + col];
   } else {
     Bs[ty][tx * RESULTS_PER_THREAD_X + x] = 0.0f;
   }
 }
```

Listing nr 5 : Uzupełnianie macierzy tymczasowych (kafelków) w pamięci współdzielonej

Ten fragment pokazuje, jak każdy wątek ładuje wiele elementów macierzy A i B, które będą potrzebne do obliczenia jego kafelka wyników. W danej iteracji pętli kafelkowania wątek pobiera do pamięci współdzielonej RESULTS_PER_THREAD_Y liczb z macierzy A oraz RESULTS_PER_THREAD_X liczb z macierzy B.

Analiza jakości dostępów do pamięci globalnej podczas ładowania danych:
Analizując ładowanie danych z macierzy A do pamięci współdzielonej możemy zauważyć, że wątki w tej samej wiązce (o tym samym *ty*, ale różnych *tx*) odczytują dane z sąsiadujących lokalizacji pamięci (*m * BLOCK_SIZE + tx)*. To zapewnia, że dostęp jest “coalesced”, czyli dostęp do pamięci globalnej wątków osnowy jest realizowany jako jedna transakcja z pamięcią.
W przypadku ładowania danych z macierzy B do pamięci współdzielonej wątki w tej samej wiązce odczytują dane z lokalizacji oddalonych o *size,* czyli z różnych wierszy macierzy. To sprawia, że każdy wątek generuje oddzielną transakcję, co w najgorszym przypadku daje 32 transakcje pamięci na wiązkę zamiast jednej transakcji. Jest to konieczny koszt, ponieważ nie można jednocześnie zapewnić efektywnych dostępów do obu macierzy A i B - zawsze jedna z macierzy będzie miała dostępy z

odstępem. Algorytm mnożenia macierzy przez kafelkowanie minimalizuje ten koszt dzięki wykorzystaniu pamięci współdzielonej.

Obliczanie iloczynu macierzy na danych z pamięci współdzielonej

Po załadowaniu danych do pamięci współdzielonej każdy wątek wykonuje obliczenia dla swojego kafelka wyników.

```
    for (int k = 0; k < BLOCK_SIZE; ++k) {
      for (int y = 0; y < RESULTS_PER_THREAD_Y; ++y) {
        float a_val = As[ty * RESULTS_PER_THREAD_Y + y][k];

        for (int x = 0; x < RESULTS_PER_THREAD_X; ++x) {
          float b_val = Bs[k][tx * RESULTS_PER_THREAD_X + x];
          sum[y][x] += a_val * b_val;
        }
      }
    }
```

Listing nr 6 : Obliczanie kafelka wynikowego w pamięci współdzielonej

Ta pętla oblicza wyniki częściowe operacji mnożenia przy danych załadowanych do pamięci współdzielonej. To znacznie redukuje czas dostępu w porównaniu do pamięci globalnej.

Analiza jakości dostępów do pamięci współdzielonej podczas obliczeń:

Dla macierzy As wątki w tej samej wiązce (mają różne *tx*) odczytują tę samą wartość, co prowadzi do rozgłaszania tej wartości do wielu wątków. Z kolei dla macierzy Bs wątki w tej samej wiązce odwołują się do sąsiednich danych w pamięci (*tx* * *RESULTS_PER_THREAD_X* + *x*), co zapewnia brak konfliktów banków pamięci współdzielonej.

Zapis wyników mnożenia

```
  for (int y = 0; y < RESULTS_PER_THREAD_Y; ++y) {
    int row = baseRow + y;
    if (row < size) {
      for (int x = 0; x < RESULTS_PER_THREAD_X; ++x) {
        int col = baseCol + x;
        if (col < size) {
          C[row * size + col] = sum[y][x];
        }
      }
    }
  }
```

Listing nr 7 : Zapisywanie obliczonych wartości do macierzy wynikowej

Po obliczeniu wszystkich wyników częściowych przez wszystkie wątki i po zakończeniu pętli kafelkowania każdy wątek zapisuje swój kafelek wyników do pamięci współdzielonej.

Analiza jakości dostępów do pamięci globalnej przy zapisywaniu wyników

Wątki w tej samej wiązce (takie które mają różne tx) zapisują do sąsiadujących lokalizacji pamięci. Zapisy są zwarte i efektywne - dostępy są zazwyczaj łączone, gdyż zapisujemy sąsiednie lokacje pamięci.

c. Korzyści z obliczania sąsiednich elementów

Podejście zastosowane w naszym kodzie, gdzie jeden wątek oblicza sąsiednie elementy macierzy wynikowej sprawia, że elementy wynikowe wykorzystują w dużej mierze te same dane wejściowe, co zwiększa ponowne wykorzystanie danych w pamięci podręcznej. Pamięć współdzielona jest efektywnie wykorzystana poprzez używanie raz załadowanych danych do obliczenia wielu sąsiednich wyników. Sąsiadujące wątki czytają sąsiadujące dane, co sprawia, że dostęp do pamięci globalnej jest “coalesced”.

d. Obliczanie rozmiaru gridu

Grid reprezentuje dwuwymiarową strukturę bloków wątków. W przypadku naszej implementacji, rozmiar gridu jest obliczany proporcjonalnie do rozmiaru macierzy wejściowych i liczby wyników przetwarzanych przez wątek.

```
  dim3 grid(
      (dimsB.x + BLOCK_SIZE * RESULTS_PER_THREAD_X - 1) / (BLOCK_SIZE *
RESULTS_PER_THREAD_X),
      (dimsA.y + BLOCK_SIZE * RESULTS_PER_THREAD_Y - 1) / (BLOCK_SIZE *
RESULTS_PER_THREAD_Y));
```

Listing nr 8 : Obliczanie rozmiaru gridu

Powyższy kod przekłada się na:

$$grid\left(\frac{N}{BS \cdot Rx}, \frac{N}{BS \cdot Ry}\right)$$

Gdzie:

N – rozmiar macierzy A i B

BS = BLOCK_SIZE – rozmiar bloku wątków

Rx = RESULTS_PER_THREAD_X – poziomy wymiar kafelka wyników

Ry = RESULTS_PER_THREAD_Y - pionowy wymiar kafelka wyników

Ten wzór zapewnia, że grid jest wystarczająco duży, aby pokryć całą macierz wynikową, uwzględniając fakt, że każdy wątek przetwarza $Rx \times Ry$ elementów. Każdy wymiar gridu jest zaokrąglony w górę.

Powyższy wzór na rozmiar gridu wpływa bezpośrednio na jakość dostępu do danych przez zapewnienie równomiernego rozłożenia obciążenia - każdy blok przetwarza równy fragment danych wyjściowych, a sąsiadujące bloki przetwarzają sąsiadujące obszary macierzy wynikowej.

e. Generowanie losowych macierzy i weryfikacja wyników

Implementacja zawiera mechanizmy do generowania losowych danych wejściowych oraz weryfikacji poprawności wyników.

```
void randomInit(float *data, int size)
{
  for (int i = 0; i < size; ++i)
    data[i] = static_cast<float>(rand() % 50000) / 100.0f;
}
```

Listing nr 9 : Wypełnianie macierzy losowymi, niejednorodnymi wartościami

Dla optymalizacji czasu testowania, program zapisuje wygenerowane macierze do plików binarnych.

```
bool saveMatrixToFile(const char *filename, float *data, size_t size)
{
  std::ofstream file(filename, std::ios::binary);
  if (!file.is_open())
  {
    printf("Failed to open file %s for writing\n", filename);
    return false;
  }
  file.write(reinterpret_cast<char *>(data), size * sizeof(float));
  file.close();
  printf("Matrix saved to %s\n", filename);
  return true;
}
```

Listing nr 10 : Zapis wygenerowanych macierzy do plików binarnych

Weryfikacja poprawności jest realizowana przez porównanie wyników GPU z referencyjnym wynikiem obliczonym na CPU. Do obliczenia mnożenia macierzy wykorzystuje strategię IKJ.

Po wykonaniu kernela mnożącego macierze na GPU wykonywane jest porównanie wyników mnożenia z wynikami CPU poprzez analizę błędów względnych z tolerancją ε^{-4}:

```
double eps = 1.e-4;
for (int i = 0; i < static_cast<int>(dimsC.x * dimsC.y); i++) {
  double abs_err = fabs(h_C[i] - h_C_cpu[i]);
  double abs_val = fabs(h_C_cpu[i]);
  double rel_err = abs_err / (abs_val > 1e-10 ? abs_val : 1.0);

  if (rel_err > eps) {
    // Oznaczanie błędu
  }
}
```

Listing nr 11 : Porównanie wyników mnożenia z wynikami CPU

5. Zilustrowanie pobierania danych do pamięci współdzielonej i wykonywanie obliczeń przez poszczególne wątki bloku

Na rysunkach przedstawimy w jaki sposób wątki pobierają dane z pamięci globalnej i wykonują obliczenia w poszczególnych iteracjach algorytmu wykorzystującego technikę kafelkowania zapewniając efektywność transferu danych.

Zaprezentujemy to dla dwóch konfiguracji kafelka wyników:

- 1 x 1 – 1 wynik obliczany przez 1 wątek
- 2 x 2 – 4 wyniki obliczane przez 1 wątek

Przedstawimy działania 4 różnych wątków oznaczonych kolorami:

Wątek 1 **Wątek 2** **Wątek 3** **Wątek 4**

Oznaczenia:

– Zamalowane komórki pokazują które liczby z macierzy A i B będą pobierane w danej iteracji pętli kafelkowania

– Kolorowa kreska pokazuje które liczby wątek będzie mnożył i dodawał do sumy częściowej w danej iteracji pętli kafelkowania

Rysunek nr 1 : Legenda wizualizacji mnożenia kafelkowego w kontekście dostępu do pamięci

W obu wizualizacjach rozmiar macierzy A i B (N) i rozmiar bloku wątków (BS):

$$N = 8 \qquad BS = 2$$

Zatem będziemy wykonywać **4 iteracje pętli kafelkowania** $\left(\frac{N}{BS}\right)$

Działanie pętli kafelkowania zostało omówione wyżej przy opisie kodu. Krótko podsumowując w każdej iteracji pętli:

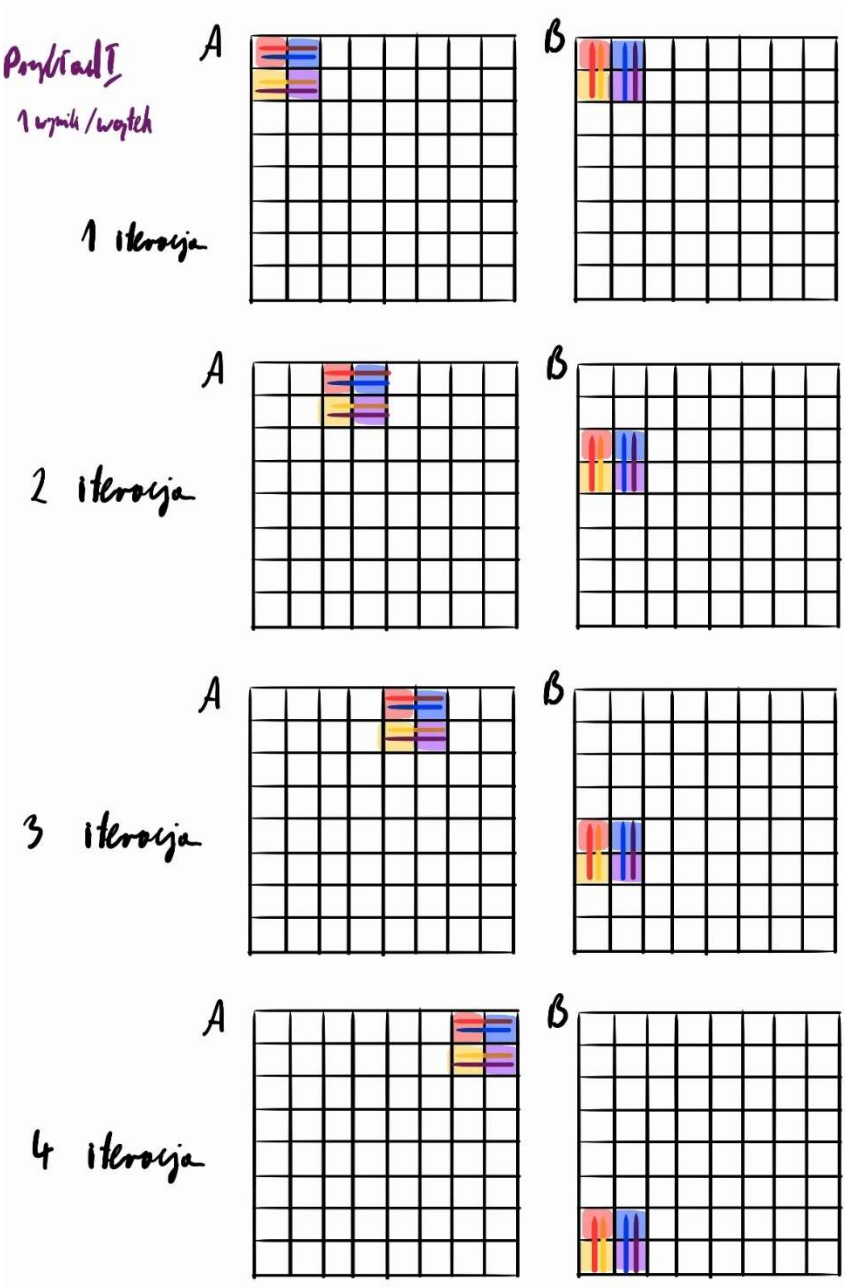
- Wątki wczytują fragmenty macierzy A i B do pamięci współdzielonej, które są oznaczone przez zamalowane komórki
- Po synchronizacji obliczają częściową sumę dla wczytanych danych

Przykład 1

Konfiguracja kafelka wyników 1 x 1, czyli 1 wynik obliczany przez 1 wątek

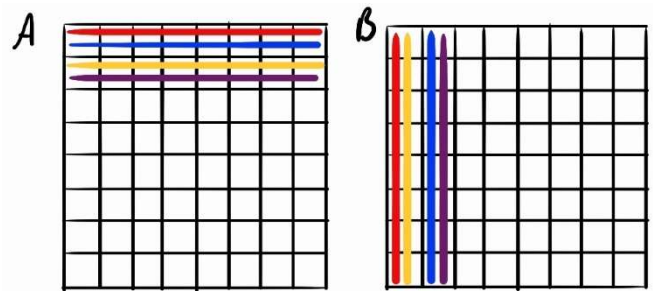
Ten przykład pokazuje obliczanie wyników mnożenia o następujących indeksach w macierzy C:

- **(0,0)** - obliczane przez wątek 1
- **(1,0)** - obliczane przez wątek 2
- **(0,1)** - obliczane przez wątek 3
- **(1,1)** - obliczane przez wątek 4



Rysunek nr 2 : Wizualizacja itearcji dla kafelka wyników 1x1

Zobrazowanie, które elementy macierzy A i B zostały wykorzystane do obliczenia wyniku mnożenia we wszystkich iteracjach:



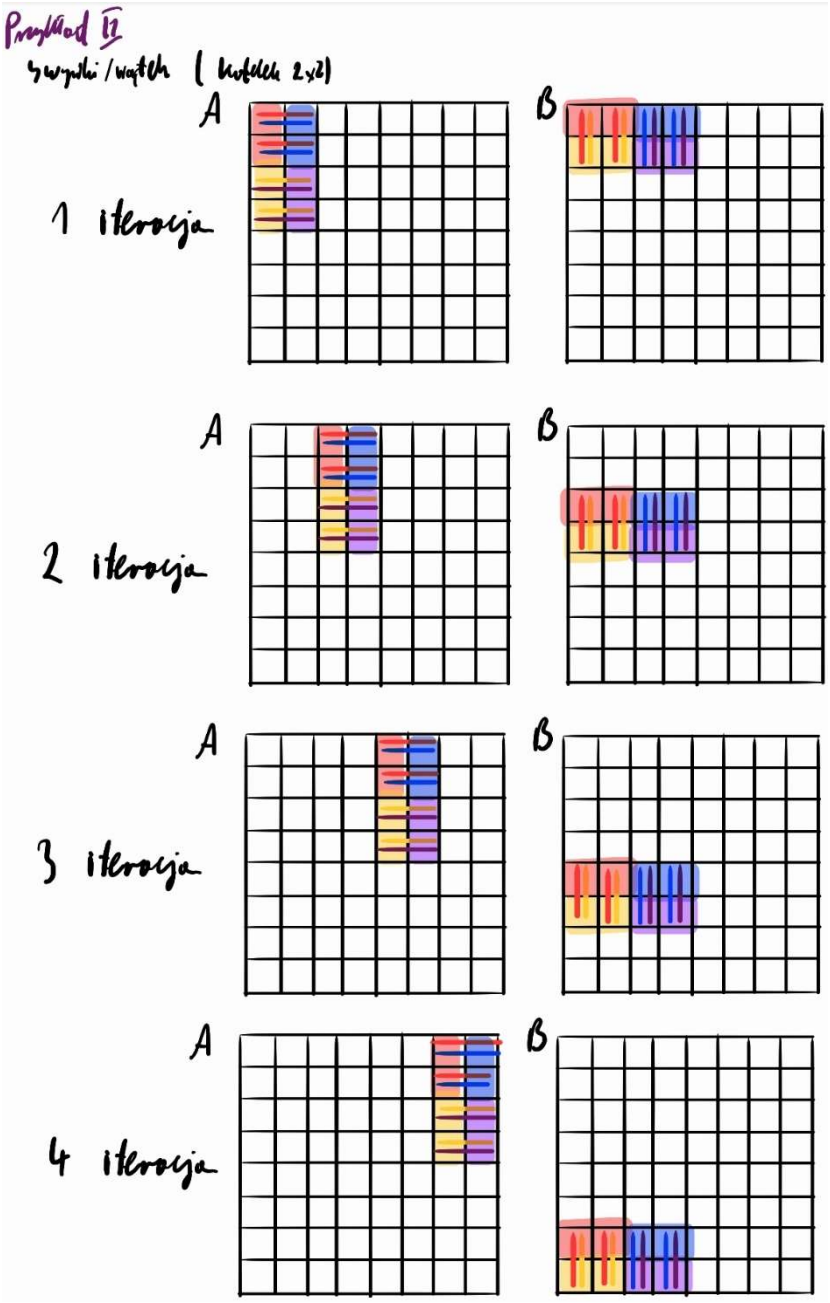
Rysunek nr 3 : Finalna wizualizacja dla kafelka 1x1

Przykład 2

Konfiguracja kafelka wyników 2 x 2, czyli 4 wyniki obliczany przez 1 wątek

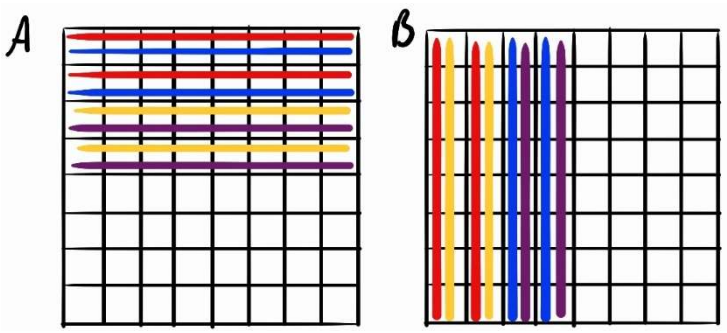
Ten przykład pokazuje obliczanie wyników mnożenia o następujących indeksach w macierzy C:

- **(0,0) (0,1) (1,0) (1,1)** - obliczane przez wątek 1
- **(2,0) (2,1) (3,0) (3,1)** - obliczane przez wątek 2
- **(0,2) (0,3) (1,2) (1,3)** - obliczane przez wątek 3
- **(2,2) (2,3) (3,2) (3,3)** - obliczane przez wątek 4



Rysunek nr 4 : Wizualizacja itearcji dla kafelka wyników 2x2

Zobrazowanie, które elementy macierzy A i B zostały wykorzystane do obliczenia wyniku mnożenia we wszystkich iteracjach:



Rysunek nr 5 : Finalna wizualizacja dla kafelka wyników 2x2

6. Obliczenia teoretyczne

a. Wyznaczenie teoretycznej liczby wyników jaką może obliczyć wątek

N - rozmiar macierzy testowych i wynikowej
BS = BLOCK_SIZE – rozmiar bloku wątków
Rx = RESULTS_PER_THREAD_X – poziomy wymiar kafelka wyników
Ry = RESULTS_PER_THREAD_Y - pionowy wymiar kafelka wyników

Wielkość słowa: 4 B
Wielkość pamięci współdzielonej na blok wątków: 49152 B

Rozmiar pamięci współdzielonej per blok: 49152 B
Pamięć współdzielona jest alokowana w postaci tablic:

```
__shared__ float As[BLOCK_SIZE * RESULTS_PER_THREAD_Y][BLOCK_SIZE];
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE * RESULTS_PER_THREAD_X];
```

Całkowite zużycie pamięci współdzielonej:

Dla As: $BS \cdot Ry \cdot BS \cdot 4$
Dla Bs: $BS \cdot BS \cdot Rx \cdot 4$
Razem: $BS^2 \cdot (Rx + Ry) \cdot 4$

Zatem dla bloków wątków o rozmiarze 16x16:

$$16^2 \cdot 4 \cdot (Rx + Ry) \leq 49152$$

$$(Rx + Ry) \leq 48$$

Dla optymalnego wykorzystania danych kafelek wyników powinien być możliwie kwadratowy, zatem

$$Rx = 24\ , Ry = 24$$

Kafelek wyników ma rozmiar 24 x 24, co daje 576 wyniki na jeden wątek.

Dla bloków wątków o rozmiarze 32x32:

$$32^2 \cdot 4 \cdot (Rx + Ry)\ \leq\ 49152$$

$$(Rx + Ry)\ \leq\ 12$$

Dla optymalnego wykorzystania danych blok kafelek powinien być możliwie kwadratowy, zatem
$Rx = 6\ , Ry\ =\ 6$
Kafelek wyników ma rozmiar 6 x 6, co daje 36 wyników na jeden wątek.

b. Liczba dostępów do pamięci globalnej z techniką kafelkowania

Prezentowane są obliczenia dla macierzy A i B, każda o rozmiarze $N \times N$.

N – liczba elementów w kolumnie i wierszu macierzy A i B
BS – liczba wątków w kolumnie i wierszu bloku wątków

Każdy wątek oblicza Rx i Ry wyników, więc rozmiar kafelka wyników to $Rx \cdot Ry$

Liczba kafelków (tiling): $ceil\left(\frac{N}{BS}\right)$
Liczba wątków wykonujących obliczenia: $\frac{N^2}{Rx \cdot Ry}$

Odczyty:
Dla każdego kafelka (w każdej iteracji) wątek odczytuje Ry elementów z macierzy A i Rx elementów z macierzy B.
Łączna liczba odczytów z pamięci globalnej:

$$\frac{N^2}{Rx \cdot Ry} \cdot \frac{N}{BS} \cdot (Ry\ +\ Rx)\ =\ \frac{N^3}{BS} \cdot \left(\frac{1}{Rx} + \frac{1}{Ry}\right)$$

Zapisy:

Każdy wątek zapisuje $Rx\ \cdot Ry$ elementów macierzy C
Łączna liczba wątków: $\frac{N^2}{Rx \cdot Ry}$
Łączna liczba zapisów:

$$\frac{N^2}{Rx \cdot Ry} \cdot Rx \cdot Ry = N^2$$

Łączna liczba dostępów do pamięci globalnej (odczyty i zapisy):

$$\frac{N^3}{BS} \cdot \left(\frac{1}{Rx} + \frac{1}{Ry}\right) + N^2$$

Przykłady obliczania liczby globalnych dostępów z rozróżnieniem liczby wyników zwracanych przez 1 wątek ($Rx \times Ry$):

Liczba wyników na wątek	Liczba globalnych dostępów	Procent zmniejszenia liczby globalnych dostępów względem 1 wyniku na wątek
1 wynik (1 x 1)	$2 \cdot \frac{N^3}{BS} + N^2$	-
2 wyniki (2 x 1)	$\frac{3}{2} \cdot \frac{N^3}{BS} + N^2$	25
4 wyniki (2 x 2)	$1 \cdot \frac{N^3}{BS} + N^2$	50
6 wyników (3 x 2)	$\frac{5}{6} \cdot \frac{N^3}{BS} + N^2$	58,3
16 wyników (2 x 8)	$\frac{5}{8} \cdot \frac{N^3}{BS} + N^2$	68,75
16 wyników (4 x 4)	$\frac{1}{2} \cdot \frac{N^3}{BS} + N^2$	75
64 wyniki (8 x 8)	$\frac{1}{4} \cdot \frac{N^3}{BS} + N^2$	87,5

Tabela 2 : Przykładowe obliczenia liczby globalnych dostępów dla różnej liczby wyników obliczanych przez jeden wątek

Na postawie zaprezentowanych obliczeń wyraźnie możemy zaobserwować, że wraz ze zwiększaniem liczby wyników na wątek liczba globalnych dostępów maleje. Wynika to z lepszego wykorzystania lokalności danych oraz efektywniejszego współdzielenia elementów wejściowych między obliczeniami wykonywanymi przez ten sam wątek.
Gdy wątek oblicza wiele sąsiadujących elementów macierzy wynikowej, elementy w tym samym wierszu macierzy wynikowej C wykorzystują ten sam wiersz macierzy A. Przy konfiguracji $Rx > 1$, wątek wykorzystuje te same elementy A dla wielu elementów wynikowych. Wygląda to podobnie dla kolumn, gdzie elementy z kolumny macierzy wynikowej C wykorzystują tę samą kolumnę macierzy B (przy $Ry > 1$).
Z powyższej tabeli jasno wynika, że nie tylko liczba wyników obliczanych przez 1 wątek ma znaczenie, ale również odpowiednia konfiguracja wymiarów bloku wyniku ($Rx \times Ry$). Możemy zaobserwować, że wariant (2 x 8) wykonuje o 68,75% mniej dostępów do pamięci globalnej względem 1 wyniku na wątek, natomiast wariant (4 x 4) o 75%, pomimo że oba te warianty obliczają 16 wyników na wątek.

c. Wyprowadzenie wzorów na CGMA dla poszczególnych konfiguracji kafelków wyników

CGMA to wskaźnik efektywności algorytmu na GPU definiowany jako:

CGMA = Liczba operacji arytmetycznych / Liczba dostępów do pamięci globalnej

Obliczenie liczby operacji arytmetycznych dla mnożenia macierzy o rozmiarze $N \times N$:

Każdy z N^2 elementów wynikowych wymaga N mnożeń i $N-1$ dodawań, co zaokrąglamy do $2 \cdot N$ operacji na element macierzy wynikowej.

Łączna liczba operacji:

$$2 \cdot N^3$$

Liczba dostępów do pamięci globalnej zgodnie z obliczeniami wyżej:

$$\frac{N^3}{BS}\left(\frac{1}{Rx} + \frac{1}{Ry}\right) + N^2$$

Zatem aby obliczyć CGMA:

$$\text{CGMA} = \frac{2 \cdot N^3}{\frac{N^3}{BS}\left(\frac{1}{Rx} + \frac{1}{Ry}\right) + N^2}$$

W praktyce zastosujemy przybliżenie CGMA. W przybliżeniu pomijamy liczbę zapisów do pamięci globalnej, ponieważ dla macierzy obliczanych w naszym projekcie zapisy stanowią mniej niż 1% wszystkich dostępów do pamięci globalnej. Dodatkowo liczba zapisów nie zależy od wartości Rx i Ry - każdy element macierzy wynikowej C jest zapisywany dokładnie 1 raz.

Stosowane w praktyce przybliżenie CGMA:

$$\text{CGMA} \approx \frac{2 \cdot N^3}{\frac{N^3}{BS}\left(\frac{1}{Rx} + \frac{1}{Ry}\right)} = \frac{2 \cdot BS}{\left(\frac{1}{Rx} + \frac{1}{Ry}\right)}$$

Przykłady obliczania liczby CGMA z rozróżnieniem liczby wyników zwracanych przez 1 wątek $(Rx \times Ry)$:

Liczba wyników na wątek	CGMA	Krotność zwiększenia CGMA względem 1 wyniku na wątek
1 wynik (1 x 1)	BS	-
2 wyniki (2 x 1)	$\frac{4}{3} \cdot BS$	1,33
4 wyniki (2 x 2)	$2 \cdot BS$	2
6 wyników (3 x 2)	$\frac{12}{5} \cdot BS$	2,4
16 wyników (2 x 8)	$\frac{16}{5} \cdot BS$	3,2

16 wyników (4 x 4)	$4 \cdot BS$	4
64 wyniki (8 x 8)	$8 \cdot BS$	8

Tabela 3 : Przykładowe obliczenia CGMA dla różnej liczby wyników obliczanych przez jeden wątek

Z przedstawionych obliczeń obserwujemy, że wraz ze wzrostem liczby wyników na wątek rośnie CGMA. Podobnie jak przy obliczaniu liczby dostępów do pamięci globalnej obserwujemy, że wynika to z lepszego wielokrotnego wykorzystania danych już załadowanych do pamięci współdzielonej. Każdy wątek, obliczając większą liczbę elementów wynikowych, może współdzielić odczytane dane między sąsiadującymi obliczeniami, co prowadzi do znacznej redukcji liczby dostępów do pamięci globalnej w przeliczeniu na jeden element wynikowy. lepszego wykorzystania lokalności danych oraz efektywniejszego współdzielenia elementów wejściowych między obliczeniami wykonywanymi przez ten sam wątek.

d. Wydajność mierzona w liczby operacji zmiennoprzecinkowych na sekundę [GFLOPs]

Wydajność (GigaFLOP na sekundę) to miara wydajności obliczeniowej, która określa liczbę miliardów operacji zmiennoprzecinkowych wykonywanych w ciągu jednej sekundy.

Wydajność [GFLOPs]= (Liczba operacji zmiennoprzecinkowych) / (Czas wykonania w sekundach) · 10^{-9}
Każdy z N^2 elementów wynikowych wymaga N mnożeń i $N - 1$ dodawań, co zaokrąglamy do $2\ N$ operacji na element macierzy wynikowej.
Łączna liczna operacji zmiennoprzecinkowych:

$$2 \cdot N^3$$

Zatem nasz wzór na Wydajność to:

$$\text{Wydajność [GFLOPs]} = 2 \cdot N^3 \text{ / (Czas wykonania w milisekundach)} \cdot 10^{-6}$$

7. Ustalenie optymalnej wielkości macierzy testowej

W celu ustalenia optymalnego rozmiaru macierzy zapewniającego nasycenie karty graficznej obliczeniami, przeanalizowaliśmy parametry wykorzystywanego GPU:

- 20 multiprocesorów (SM)
- Każdy SM obsługuje do 2048 wątków
- Łącznie karta może obsługiwać 40 x 1024 = 40 960 wątków równolegle
- Pamięć współdzielona na blok: 49152 B
- Pamięć współdzielona na SM: 98304 B
- Rozmiar pamięci L2: 2 MB
- Rozmiar słowa: 4 B (float)

W obliczeniach wykorzystujemy:

N - rozmiar macierzy testowych i wynikowej

BS - rozmiar bloku wątków

Rx - poziomy wymiar kafelka wyników

Ry - pionowy wymiar kafelka wyników

Teraz przeanalizujemy potencjalne wielkości macierzy w zależności od kafelka wyników obliczanych przez 1 wątek ($Rx \times Ry$). Dla każdego z analizowanych rozmiarów bloku wątków zbadamy najmniejszy i największy możliwy kafelek wyników.

Analiza dla rozmiaru bloku wątków 32 x 32

- Każdy SM obsługuje do 2 bloków wątków po 32 x 32 wątki

$$\frac{2048}{32 \cdot 32} = \frac{2048}{1024} = 2$$

- Maksymalnie 40 bloków wątków (20 SM x 2 bloki/SM) może być wykonywanych jednocześnie

Aby nasycić obliczenia dla takiego bloku wątków potrzebujemy minimum 40 bloków wątków.

- Kafelek wyników 1 x 1 (przypadek minimalny)
  - Konfiguracja siatki (grid): grid$(\frac{N}{Bs} \times \frac{N}{Bs})$
  - Całkowita liczba bloków wątków: $\left(\frac{N}{BS}\right)^2$

  Aby nasycić GPU obliczeniami potrzebujemy min. 40 bloków

$$\left(\frac{N}{BS}\right)^2 \geq 40$$
$$\frac{N^2}{1024} \geq 40$$
$$N \geq 203$$

- Kafelek wyników 6 x 6 (przypadek maksymalny)
  - Konfiguracja siatki (grid): grid$(\frac{N}{Bs \cdot 6} \times \frac{N}{Bs \cdot 6})$
  - Całkowita liczba bloków wątków: $\left(\frac{N}{BS \cdot 6}\right)^2$

$$\left(\frac{N}{BS \cdot 6}\right)^2 \geq 40$$

$$\frac{N^2}{36864} \geq 40$$

$$N \geq 1215$$

Analiza dla rozmiaru bloku wątków 16 x 16

- Każdy SM obsługuje do 8 bloków wątków po 16 x 16 wątki

$$\frac{2048}{16 \cdot 16} = \frac{2048}{256} = 8$$

- Maksymalnie 160 bloków wątków (20 SM x 8 bloki/SM) może być wykonywanych jednocześnie

  Aby nasycić obliczenia dla takiego bloku wątków potrzebujemy minimum 160 bloków wątków.

Dla kafelka wyników 1x1 i 24 x 24 obliczenia są analogiczne jak dla bloku wątków o rozmiarze 32 x 32, zatem przestawimy jedynie wyniki.

- Kafelek wyników 1 x 1 (przypadek minimalny)

$$N \geq 203$$

- Kafelek wyników 24 x 24 (przypadek maksymalny)
  - Konfiguracja siatki (grid): grid($\frac{N}{Bs \cdot 24} \times \frac{N}{Bs \cdot 24}$)
  - Całkowita liczba bloków wątków: $\left(\frac{N}{Bs \cdot 24}\right)^2$

$$\left(\frac{N}{Bs \cdot 24}\right)^2 \geq 160$$

$$\frac{N^2}{382^2} \geq 160$$

$$N \geq 4857$$

Wybór rozmiaru macierzy testowej - 4096 x 4096

Uwzględniając powyższe obliczenia zdecydowaliśmy się na wybranie macierzy testowej o wymiarach **4096 x 4096**. Ten rozmiar zapewnia podzielność przez testowane rozmiary bloków wątków, dzięki czemu obliczenia mnożenia macierzy mogą być po równo rozłożone na bloki. Mamy też podzielność przez wymiary kafelków wyników $(Rx \times Ry)$, co minimalizuje różnice wydajności wynikające z nierównomiernego podziału pracy na wątki.
Wybrany rozmiar macierzy gwarantuje nadmiar bloków wątków (min. 40 bloków wątków 32 x 32 i min. 160 bloków wątków 16 x 16) dla praktycznie wszystkich konfiguracji bloku wyników.
Co prawda proponowany przez nas rozmiar nie spełnia warunku najbardziej wymagającej konfiguracji kafelka wyników (24 x 24 przy bloku wątku 16 x 16):

$$\left(\frac{4096}{16 \cdot 24}\right)^2 \approx 113{,}77, \text{ co daje 114 bloków wątków}$$

Jednak, pomimo że 114 bloków jest mniejsze niż teoretyczne 160 bloków, to wciąż zapewnia ok. 5,7 bloku na SM, co daje wystarczające nasycenie dla dokładnego porównania wydajności między konfiguracjami liczby zwracanych wyników przez 1 wątek.
Dodatkowym argumentem za tym rozmiarem macierzy, jest fakt, że złożoność $O(N^3)$ sprawia, że zwiększenie rozmiaru o ~19% (z 4096 do 4857) wydłużyłoby czas obliczeń o ~67%.

8. Zastosowane konfiguracje uruchomienia kodu

Dobór konfiguracji kafelków wyników ($Rx \times Ry$) został przemyślany z uwzględnieniem rozmiarów bloków wątków $BS = 32$ oraz $BS = 16$.
Podstawowe konfiguracje testowane dla obu rozmiarów bloku wątków:

- **1x1**: stanowi bazowy punkt odniesienia - każdy wątek oblicza dokładnie jeden element macierzy wynikowej
- **2x1, 2x3, 2x8:** prostokątne konfiguracje, przez które chcemy zbadać jak stosunek wymiarów kafelka wyników wpływa na wzorce dostępu do pamięci
- **2x2, 4x4, 6x6:** przez kwadratowe konfiguracje chcemy zbadać jak zwiększenie liczby wyników na wątek wpływa na wydajność

Dla bloku wątków o rozmiarze 32:

- **5x6, 6x6:** konfiguracje graniczne testujące maksymalne wykorzystanie pamięci współdzielonej

Dla bloku wątków o rozmiarze 16:

- **8x8, 16x16:** kolejne kwadratowe konfiguracje, niemożliwe do zbadania, gdy $BS = 32$
- **23x24, 24x24:** konfiguracje graniczne testujące maksymalne wykorzystanie pamięci współdzielone

9. Wyniki NVIDIA Profiler

Testy zostały wykonane na niejednorodnych danych. Dane w macierzach A i B, są tworzone losowo wraz z uruchomieniem programu.

Wyniki zmierzone przez NVIDIA Profiler bazują na średniej wartości z dwóch wywołań danego kernela. (Kolumna z nvprof, Invocations = 2).

Do uruchomienia pojedynczego fragmentu kodu dla jednego typu kafelka wyników np. (1x1), wykorzystujemy funkcję:

```
template <int RESULTS_PER_THREAD_X = 1, int RESULTS_PER_THREAD_Y = 1>
bool RunMatrixMultiplyTest(float *h_A, float *h_B, float *h_C_cpu,
                           const dim3 &dimsA, const dim3 &dimsB)
```

Listing 12 : Szablon funkcji uruchamiącej kernel mnożenia macierzy

W środku tej funkcji dwa razy wywołujemy ten sam kernel dla tych samych parametrów, aby otrzymać rzetelne i prawidłowe wyniki (pierwsze wywołanie jest rozgrzewkowe - warmup):

```
// Performs warmup operation using MatrixMul CUDA kernel
  MatrixMulCUDA<RESULTS_PER_THREAD_X, RESULTS_PER_THREAD_Y><<<grid, threads, 0,
stream>>>(d_A, d_B, d_C, dimsA.x);

  printf("Warmup completed\n");
  checkCudaErrors(cudaStreamSynchronize(stream));

  // Record the start event
  checkCudaErrors(cudaEventRecord(start, stream));

  MatrixMulCUDA<RESULTS_PER_THREAD_X, RESULTS_PER_THREAD_Y><<<grid, threads, 0,
stream>>>(d_A, d_B, d_C, dimsA.x);
```

Listing 13 : Wywołanie kerneli wewnątrz funkcji RunMatrixMultiplyTest

W ramach testów przeprowadziliśmy pomiary na omówionych wcześniej konfiguracjach kafelków wyników najpierw dla rozmiaru macierzy proponowanego przez NVIDIA, a następnie dla obliczonej przez nas teoretycznej wielkości macierzy, która zapewnia nasycenie obliczeniami – 4096 x 4096.

Rozmiar macierzy, domyślny dla programu NVIDIA:
50 * 2 * BLOCK_SIZE x 50 * 2 * BLOCK_SIZE

Opis parametrów zmierzonych profilerem:

- **achieved_occupancy** - Osiągnięte zajęcie - procent teoretycznego maksymalnego zajęcia SM (Streaming Multiprocessor), wskazujący jak efektywnie wykorzystywane są zasoby procesora
- **sm_efficiency** - Efektywność SM - procent czasu, w którym jedna lub więcej wiązek są aktywne na multiprocesorze strumieniowym
- **ipc** - Instructions Per Clock - średnia liczba instrukcji wykonywanych na jeden cykl zegara, miara efektywności wykonywania kodu
- **flop_count_sp** - Liczba operacji zmiennoprzecinkowych pojedynczej precyzji - całkowita liczba wykonanych operacji matematycznych
- **gld_transactions** - Transakcje ładowania globalnego - liczba transakcji odczytu z pamięci globalnej GPU
- **gst_transactions** - Transakcje zapisywania globalnego - liczba transakcji zapisu do pamięci globalnej GPU
- **shared_load_transactions** - Transakcje ładowania pamięci współdzielonej - liczba operacji odczytu z szybkiej pamięci współdzielonej między wątkami w bloku
- **shared_store_transactions** - Transakcje zapisywania pamięci współdzielonej - liczba operacji zapisu do pamięci współdzielonej między wątkami

- **l2_read_transactions** - Transakcje odczytu L2 - liczba operacji odczytu z pamięci cache drugiego poziomu
- **l2_write_transactions** - Transakcje zapisu L2 - liczba operacji zapisu do pamięci cache drugiego poziomu
- **dram_read_transactions** - Transakcje odczytu DRAM - liczba dostępów odczytu do głównej pamięci urządzenia
- **dram_write_transactions** - Transakcje zapisu DRAM - liczba dostępów zapisu do głównej pamięci urządzenia
- **dram_read_bytes** - całkowita liczba bajtów pobrana z RAM karty do pamięci podręcznej
- **dram_write_bytes** - całkowita liczba bajtów zapisana do RAM karty z pamięci podręcznej
- **Czas przetwarzania (ms)** – Duration - całkowity czas wykonania kernela lub operacji na GPU
- **Wydajność / efficency (Gigaflops)** - Performance in Gigaflops- liczba miliardów operacji zmiennoprzecinkowych wykonywanych w ciągu sekundy.

  Wartość obliczona według wzoru: $\frac{\left(\frac{flopcountsp}{10^9}\right)}{\left(\frac{duration[ms]}{1000}\right)}$
- **CGMA (flop/liczba transakcji operacji pamięciowych)** - Compute to Global Memory Access ratio - stosunek liczby operacji obliczeniowych do transferu danych z pamięcią globalną, wskazujący intensywność obliczeniową algorytmu.

  Wartość obliczona według wzoru: $\frac{flopcountsp}{(gldtransactions + gsttransactions)}$

- **Intensywność arytmetyczna / intensity** – stosunek liczby operacji obliczeniowych do liczby do bajtów pobranych z RAM. Rosnąca wartość oznacza, że więcej obliczeń wykonuje się na każdym bajcie danych pobranym z pamięci.

  Wartość obliczona według wzoru: $\frac{flopcountsp}{dramreadbytes}$

Wyniki z profilera dla bloku (BLOCK_SIZE) 16x16:

	1x1	2x1	2x2	2x3	4x4
achieved_occupancy	99,56%	99,21%	98,67%	74,58%	48,98%
sm_efficiency	99,90%	99,42%	98,08%	95,44%	98,18%
ipc	1,45E+00	1,83E+00	2,18E+00	2,52E+00	2,63E+00
flop_count_sp [GFlop]	8,19E+00	8,19E+00	8,19E+00	8,36E+00	8,19E+00
gld_transactions	2,56E+08	1,92E+08	1,28E+08	1,08E+08	6,40E+07
gst_transactions	3,20E+05	6,40E+05	6,40E+05	6,40E+05	1,28E+06
shared_load_transactions	1,92E+08	1,60E+08	9,60E+07	7,62E+07	4,80E+07
shared_store_transactions	1,60E+07	2,00E+07	1,60E+07	9,52E+06	1,20E+07
l2_read_transactions	6,40E+07	8,00E+07	4,80E+07	3,78E+07	4,00E+07
l2_write_transactions	3,20E+05	6,40E+05	6,40E+05	6,40E+05	1,28E+06
dram_read_transactions	3,13E+07	1,86E+07	5,64E+06	6,56E+06	3,10E+06
dram_write_transactions	3,33E+05	3,38E+05	3,30E+05	3,47E+05	3,25E+05

dram_read_bytes	1.01E+09	7.44E+08	1.87E+08	2.01E+08	9.86E+07
dram_write_bytes	1.02E+07	1.02E+07	1.02E+07	1.02E+07	1.03E+07
duration [ms]	1,12E+01	6,56E+00	3,98E+00	3,19E+00	2,24E+00
effiency [Gflops]	7,29E+02	1,25E+03	2,06E+03	2,62E+03	3,66E+03
CGMA [flop / access]	3,20E+01	4,25E+01	6,37E+01	7,69E+01	1,25E+02
intensity [flop / bytes]	8.11E+00	1.10E+01	4.37E+01	4.16E+01	8.31E+01

Tabela 4 : Wyniki NVIDIA Profiler dla domyślnego rozmiaru macierzy wynikowej i dla bloku 16x16 część 1

	5x6	6x6	8x8	16x16	23x24	24x24
achieved_occupancy	36,10%	24,37%	24,69%	12,49%	15,81%	12,50%
sm_efficiency	95,65%	94,70%	85,57%	81,81%	94,10%	85,07%
ipc	2,48E+00	2,90E+00	2,12E+00	1,09E+00	9,55E-02	5,13E-02
flop_count_sp [GFlop]	8,36E+00	8,52E+00	8,86E+00	1,03E+01	1,13E+01	1,18E+01
gld_transactions	4,74E+07	4,40E+07	3,39E+07	1,83E+07	1,38E+07	1,33E+07
gst_transactions	1,79E+06	1,92E+06	2,56E+06	2,56E+06	2,56E+06	2,56E+06
shared_load_transactions	7,40E+07	3,33E+07	6,92E+07	6,02E+07	2,27E+07	3,07E+07
shared_store_transactions	4,62E+06	5,55E+06	1,08E+07	1,13E+07	1,42E+06	4,80E+06
l2_read_transactions	3,36E+07	3,81E+07	3,75E+07	9,09E+07	7,22E+08	7,53E+08
l2_write_transactions	1,79E+06	1,92E+06	2,56E+06	7,30E+07	7,10E+08	7,40E+08
dram_read_transactions	2,44E+06	2,86E+06	1,93E+06	1,44E+06	7,10E+08	7,34E+08
dram_write_transactions	3,29E+05	3,31E+05	3,42E+05	1,15E+06	7,07E+08	7,38E+08
dram_read_bytes	7.86E+07	9.18E+07	6.27E+07	4.63E+07	2.27E+10	2.36E+10
dram_write_bytes	1.04E+07	1.02E+07	1.02E+07	3.42E+07	2.26E+10	2.36E+10
duration [ms]	2,75E+00	2,13E+00	3,20E+00	8,00E+00	1,84E+02	1,86E+02
efficiency [Gflops]	3,04E+03	4,00E+03	2,77E+03	1,28E+03	6,15E+01	6,33E+01
CGMA [flop / access]	1,70E+02	1,86E+02	2,43E+02	4,93E+02	6,93E+02	7,43E+02
intensity [flop / bytes]	1.06E+02	9.29E+01	1.41E+02	2.22E+02	4.97E-01	4.99E-01

Tabela 5 : Wyniki NVIDIA Profiler dla domyślnego rozmiaru macierzy wynikowej i dla bloku 16x16 część 2

Dla bloku 32x32:

	1x1	2x1	2x2	2x3	2x4	4x4
achieved_occupancy	99,87%	99,82%	99,89%	49,97%	49,97%	49,93%
sm_efficiency	99,86%	99,78%	99,27%	99,95%	99,17%	97,68%
ipc	1,65E+00	1,51E+00	1,83E+00	1,99E+00	2,19E+00	2,06E+00
flop_count_sp [GFlop]	6,55E+01	6,55E+01	6,55E+01	6,68E+01	6,55E+01	6,55E+01
gld_transactions	1,02E+09	7,68E+08	5,12E+08	4,30E+08	3,84E+08	2,56E+08
gst_transactions	1,28E+06	2,56E+06	2,56E+06	2,56E+06	2,56E+06	5,12E+06
shared_load_transactions	1,54E+09	1,28E+09	7,68E+08	6,09E+08	5,12E+08	3,84E+08
shared_store_transactions	6,40E+07	8,00E+07	4,80E+07	3,81E+07	3,20E+07	4,00E+07
l2_read_transactions	2,56E+08	3,20E+08	1,92E+08	1,51E+08	1,28E+08	1,60E+08
l2_write_transactions	1,28E+06	2,56E+06	2,56E+06	2,56E+06	2,56E+06	5,12E+06
dram_read_transactions	1,34E+08	1,33E+08	6,71E+07	4,75E+07	3,59E+07	3,46E+07
dram_write_transactions	1,29E+06	1,29E+06	1,29E+06	1,29E+06	1,29E+06	1,29E+06
dram_read_bytes	4.31E+09	4.29E+09	2.14E+09	1.52E+09	1.31E+09	1.11E+09
dram_write_bytes	4.09E+07	4.10E+07	4.10E+07	4.10E+07	4.10E+07	4.10E+07
duration [ms]	6,05E+01	4,33E+01	2,62E+01	2,48E+01	2,07E+01	1,73E+01
efficiency [Gflops]	1,08E+03	1,51E+03	2,50E+03	2,70E+03	3,16E+03	3,80E+03
CGMA [flop / access]	6,39E+01	8,50E+01	1,27E+02	1,55E+02	1,70E+02	2,51E+02
intensity [flop / bytes]	1.52E+01	1.53E+01	3.06E+01	4.40E+01	4.99E+01	5.91E+01

Tabela 6 : Wyniki NVIDIA Profiler dla domyślnego rozmiaru macierzy wynikowej i dla bloku 32x32

Obserwacje na temat dokonanych pomiarów za pomocą narzędzia NVIDIA Profiler:

Na podstawie wyników z profilera możemy zaobserwować, że powiększenie liczby wyników obliczanych przez jeden wątek skutkuje znacznym wzrost efektywności obliczeń. Potwierdzają to teoretyczne obliczenia.

Obserwacje dla bloku 16x16:

Dla BLOCK_SIZE = 16, wariant z rozmiarem kafelka 6x6, czyli 36 wyników na wątek, posiada największą wydajność: ~4000 GFLOPS. Taki wynik stanowi 548,7% wydajności wariantu 1x1. CGMA na poziomie 1,86E+02 jest dobrym wynikiem. Taka wartość w zestawieniu z wysoką wydajnością wskazuje na wysoką efektywność tego wariantu. Ponadto, czas przetwarzania wariantu dla kafelka 6x6 jest najmniejszy z wszystkich i wynosi 2,13ms. Słabym punktem tego wariantu jest jedynie procent teoretycznego maksymalnego zajęcia multiprocesora. Wynosi on ~24%, gdzie dla przykładu warianty z małymi rozmiarami kafelków (do 2x2) mają około 90%.

Dla BLOCK_SIZE = 16, wariant z rozmiarem kafelka 23x24 z kolei charakteryzuje się najgorszą wydajnością na poziomie ~61,5 GFLOPS. Jest to drastyczny spadek do zaledwie 8,4% względem

wydajności wariantu 1x1. Zajętość multiprocesora w tym przypadku jest najgorszą spośród wszystkich wariantów, około 13%. Z drugiej strony, ten wariant posiada bardzo dobry wynik w kontekście CGMA. Dla takiego dużego rozmiaru kafelka, liczba odczytów z pamięci globalnej jest bardzo niska w porównaniu do reszty wariantów. Bez względu na to, czas przetwarzania jest wysoki, bo ~180ms, a wydajność jest bardzo niska. Warto też zwrócić uwagę na intensywność arytmetyczą dla tych wariantów. Z reguły widoczny jest trend liniowy dla intensity, gdzie ta wartość wzrasta wraz z liczbą wyników na wątek, ale w przypadku granicznych rozmiarów kafelków, ta intensywność arytmetyczna spada wręcz do 0,5 operacji / bajty pamięci pobranej z RAM. Taki wynik wyraźnie wskazuje na przeforsowanie GPU i wystąpienie overflow pamięci, czyli program wykonuje zbyt mało obliczeń w stosunku do ilości pobieranych danych z pamięci RAM.

Co prawda udało się wykonać pomiar na wariantach kafelków 23x24 oraz 24x24, czyli maksymalnie granicznej liczbie wyników obliczonej przez jeden wątek, ale wyniki są zdecydowanie gorsze w porównaniu do innych wariantów.

Obserwacje dla bloku 32x32:

Dla BLOCK_SIZE=32, konfiguracja dla kafelka 4x4 posiada największą wydajność ~3800 GFLOPS i najmniejszy czas przetwarzania ~17ms. Podobnie jak w przypadku rozmiaru bloku 16x16, tutaj wariant z najlepszymi wynikami posiada stosunkowo małą zajętość multiprocesora, bo ~50%, gdzie warianty do 2x2 posiadają około 90%. Parametr CGMA w porównaniu do 1x1 jest wyższy, 251 operacji na dostęp, czyli wyższy o prawie 300%. Ten wariant okazuje się idealnie znajdować balans między wydajnością a wykorzystaniem zasobów.

Dla BLOCK_SIZE=32, wariant z rozmiarem kafelka 1x1 się okazuje być najgorszym w kontekście wydajności, ~1080 GFLOPS. Posiada dobre CGMA w wysokości ~64 operacji na dostęp, natomiast jego czas przetwarzania jest wysoki, ~60ms. Każdy inny wariant wprowadzający więcej wyników niż 1 na wątek gwarantuje większą wydajność i lepsze wyniki.

Tutaj intensywność arytmetyczna wzrasta systematycznie wraz ze zwiększaniem liczby wyników obliczanych przez wątek. Dla bloku 32x32 obserwujemy wzrost od 15,2 flop / bytes do 59,1 flop / bytes.

Próba wykonania pomiarów dla kafelków 5x6 oraz 6x6 skutkowała wynikiem FAIL. Profiler nie był w stanie zmierzyć żadnych metryk, ponieważ takie konfiguracje naruszały limity pamięci współdzielonej. Według teoretycznych obliczeń, dla BLOCK_SIZE 32x32 taka liczba wyników powinna być możliwa, ale w praktyce obliczenie takiej liczby wyników okazało się za bardzo wykraczać poza granice wykorzystania zasobów GPU.

Obserwacje ogólne:

Dla bloku 32x32 zauważalny jest linowy wzrost wydajności wraz z zwiększaniem liczby wyników na wątek. W przypadku bloku 16x16 występuje tutaj bardziej krzywa wydajności typu odwróconego U. Dla mniejszych liczby wyników mamy słabą wydajność, tak samo jak dla największej liczby wyników. Kafelki wyników o rozmiarach 6x6, 4x4 mają tą wydajność najlepszą.

W zestawieniu dwóch rozmiarów bloków widoczny jest również wzrost CGMA wprost proporcjonalny do liczby obliczonych wyników przez wątek. Taka obserwacja wynika z tego, że wraz z liczbą wyników spada liczba odczytów z pamięci globalnej.

Podobnie jak w przypadku CGMA widoczny jest wzrost intensywności arytmetycznej wprost proporcjonalny do liczby wyników na wątek.

Ważnym aspektem jest również fakt, że dla mniejszej liczby wyników na wątek zajętość procesora jest największa, zarówno dla bloku 16x16 jak i 32x32. Wraz z zwiększeniem liczby wyników, achieved occupancy maleje i to drastycznie. Dzieje się tak przynajmniej dla macierzy A i B w rozmiarze 50 * 2 * BLOCK_SIZE, 50 * 2 * BLOCK_SIZE. Dodatkowo w niektórych przypadkach sm_efficiency jest mniejsze (zależy od doboru parametrów, ponieważ nie widoczny jest żaden trend). Możliwe, że modyfikacja rozmiaru macierzy poprawi nasycenie, a wyniki będą jeszcze lepsze (zostało to omówione niżej).

Wnioski z obserwacji:

Analiza wyników tiled matrix multiplication na GPU wykazała, że optymalne rozmiary kafelków zależą od rozmiaru bloku wątków - dla bloków 16x16 najlepszy jest kafelek 6x6, a dla bloków 32x32 kafelek 4x4. Krzywa wydajności ma kształt odwróconego U, gdzie małe kafelki są nieefektywne z powodu słabszej lokalności danych, średnie kafelki (4x4-6x6) osiągają optimum wykorzystania zasobów GPU, a duże kafelki (>8x8) powodują drastyczny spadek wydajności, prawdopodobnie przez spilling rejestrów i niskie achieved occupancy. Spilling rejestrów oznacza, że gdy wątek potrzebuje więcej rejestrów niż ma dostępnych, część danych musi być zapisana w dużo wolniejszej pamięci zamiast w szybkich rejestrach, co spowalnia działanie programu. Kluczowym wskaźnikiem efektywności jest CGMA (stosunek obliczeń do dostępów pamięci globalnej), który dla optymalnych wariantów osiąga wartości 186-251 operacji / dostęp. Ten współczynnik rośnie liniowo wraz z liczbą wyników na wątek. Wysoka okupacja nie gwarantuje wysokiej wydajności - optymalne warianty osiągają okupację 24-50% przy maksymalnej wydajności.

Modyfikacja programu mnożenia macierzy od NVIDII w postaci zwiększenia liczby wyników obliczanych przez wątek w bloku jest sensowna i przynosi znaczące wzrosty w wydajności i wykorzystaniu GPU.

Rozmiar macierzy 4096 x 4096

W celu przeprowadzenia większego zakresu eksperymentów nasyciliśmy GPU dobierając odpowiedni rozmiar macierzy i sprawdzić wyniki dla innego rozmiaru. Według obliczeń teoretycznych optymalnym rozmiarem dla karty NVIDIA GTX 1080, w celu jej maksymalnego nasycenia, powinna być macierz kwadratowa o rozmiarze 4096 x 4096.
Sposób w jaki dobraliśmy optymalny rozmiar testowej macierzy wynikowej został opisany wcześniej w sprawozdaniu w sekcji “Ustalenie optymalnej wielkości macierzy testowej”.

Wyniki z profilera dla bloku 16x16:

	1x1	2x1	2x2	2x3	2x8	4x4
achieved_occupancy	99,95%	99,87%	99,94%	74,93%	37,41%	49,86%
sm_efficiency	99,93%	99,90%	99,48%	99,69%	99,17%	99,21%
ipc	1,84E+00	1,84E+00	2,18E+00	2,54E+00	2,85E+00	2,68E+00
flop_count_sp[Gflop]	1,37E+02	1,37E+02	1,37E+02	1,39E+02	1,37E+02	1,37E+02
gld_transactions	4,29E+09	3,22E+09	2,15E+09	1,80E+09	1,34E+09	1,07E+09

gst_transactions	2,10E+06	4,19E+06	4,19E+06	4,19E+06	4,19E+06	8,39E+06
shared_load_transactions	3,22E+09	2,68E+09	1,61E+09	1,26E+09	8,05E+08	8,05E+08
shared_store_transactions	2,68E+08	3,36E+08	2,68E+08	1,58E+08	1,68E+08	2,01E+08
l2_read_transactions	1,07E+09	1,34E+09	8,06E+08	6,29E+08	4,03E+08	6,71E+08
l2_write_transactions	2,10E+06	4,19E+06	4,20E+06	4,19E+06	4,19E+06	8,39E+06
dram_read_transactions	5,51E+08	5,46E+08	2,75E+08	1,94E+08	7,65E+07	1,42E+08
dram_write_transactions	2,10E+06	2,13E+06	2,10E+06	2,10E+06	2,09E+06	2,09E+06
dram_read_bytes	1.77E+10	1.75E+10	8.76E+09	6.17E+09	2.53E+09	4.39E+09
dram_write_bytes	6.71E+07	6.71E+07	6.71E+07	6.71E+07	6.71E+07	6.72E+07
duration [ms]	1,38E+02	9,80E+01	6,09E+01	4,61E+01	3,46E+01	3,44E+01
efficiency [Gflops]	9,94E+02	1,40E+03	2,26E+03	3,01E+03	3,98E+03	4,00E+03
CGMA [flop / access]	3,20E+01	4,26E+01	6,39E+01	7,70E+01	1,02E+02	1,27E+02
intensity [flop / bytes]	7,75E+00	7,84E+00	1,56E+01	2,25E+01	5,43E+01	3,12E+01

Tabela 7 : Wyniki NVIDIA Profiler dla optymalnego rozmiaru macierzy wynikowej i dla bloku 16x16 część 1

	5x6	6x6	8x8	16x16	23x24	24x24
achieved_occupancy	37,31%	24,91%	24,99%	12,50%	24,88%	12,50%
sm_efficiency	99,03%	98,87%	98,26%	97,76%	<OVERFLOW>	<OVERFLOW>
ipc	2,55E+00	3,00E+00	2,17E+00	1,08E+00	1,00E-01	4,58E-02
flop_count_sp[Gflop]	1,41E+02	1,40E+02	1,37E+02	1,37E+02	1,53E+02	1,46E+02
gld_transactions	7,99E+08	7,24E+08	5,37E+08	2,68E+08	1,96E+08	1,87E+08
gst_transactions	1,17E+07	1,26E+07	1,68E+07	1,68E+07	1,68E+07	1,68E+07
shared_load_transactions	1,25E+09	5,45E+08	1,07E+09	8,05E+08	3,07E+08	3,81E+08
shared_store_transactions	7,78E+07	9,09E+07	1,68E+08	1,51E+08	1,92E+07	5,95E+07
l2_read_transactions	5,60E+08	6,31E+08	6,04E+08	1,25E+09	9,77E+09	9,36E+09
l2_write_transactions	1,17E+07	1,26E+07	1,68E+07	9,59E+08	9,57E+09	9,15E+09
dram_read_transactions	9,46E+07	9,75E+07	7,35E+07	4,34E+07	9,61E+09	9,13E+09
dram_write_transactions	2,10E+06	2,12E+06	2,10E+06	1,45E+07	9,55E+09	9,10E+09
dram_read_transactions	3.10E+09	3.07E+09	2.26E+09	1.37E+09	3.07E+11	2.92E+11
dram_write_transactions	6.71E+07	6.72E+07	6.71E+07	4.05E+08	3.06E+11	2.91E+11
duration [ms]	4,08E+01	2,94E+01	4,14E+01	7,97E+01	2,67E+03	2,51E+03
efficiency [Gflops]	3,45E+03	4,75E+03	3,32E+03	1,73E+03	5,73E+01	5,83E+01
CGMA [flop / access]	1,74E+02	1,89E+02	2,48E+02	4,82E+02	7,18E+02	7,16E+02
intensity [flop / bytes]	4.54E+01	4.56E+01	6.07E+01	9.98E+01	4.98E-01	5.00E-01

Tabela 8 : Wyniki NVIDIA Profiler dla optymalnego rozmiaru macierzy wynikowej i dla bloku 16x16 część 2

Dla bloku 32x32:

	1x1	2x1	2x2	2x3	4x4
achieved_occupancy	99,91%	99,90%	99,90%	49,97%	49,94%
sm_efficiency	99,92%	99,76%	99,49%	99,69%	98,45%
ipc	1,66E+00	1,52E+00	1,84E+00	2,02E+00	2,13E+00
flop_count_sp [Gflop]	1,37E+02	1,37E+02	1,37E+02	1,39E+02	1,37E+02
gld_transactions	2,15E+09	1,61E+09	1,07E+09	8,98E+08	5,37E+08
gst_transactions	2,10E+06	4,19E+06	4,19E+06	4,19E+06	8,39E+06
shared_load_transactions	3,22E+09	2,68E+09	1,61E+09	1,26E+09	8,05E+08
shared_store_transactions	1,34E+08	1,68E+08	1,01E+08	7,89E+07	8,39E+07
l2_read_transactions	5,37E+08	6,71E+08	4,03E+08	3,15E+08	3,36E+08
l2_write_transactions	2,10E+06	4,20E+06	4,19E+06	4,20E+06	8,39E+06
dram_read_transactions	2,87E+08	2,88E+08	1,40E+08	9,88E+07	7,23E+07
dram_write_transactions	2,12E+06	2,11E+06	2,10E+06	2,10E+06	2,11E+06
dram_read_bytes	9.17E+09	9.22E+09	4.51E+09	3.16E+09	2.31E+09
dram_write_bytes	8.01E+07	6.80E+07	6.78E+07	6.77E+07	6.77E+07
Duration [ms]	1,19E+02	8,60E+01	5,52E+01	5,11E+01	3,57E+01
effiency [Gflops]	1,16E+03	1,60E+03	2,49E+03	2,71E+03	3,85E+03
CGMA [flop / access]	6,39E+01	8,51E+01	1,28E+02	1,54E+02	2,52E+02
intensity [flop / bytes]	1.49E+01	1.49E+01	3.04E+01	4.40E+01	5.92E+01

Tabela 9 : Wyniki NVIDIA Profiler dla optymalnego rozmiaru macierzy wynikowej i dla bloku 32x32

Obserwacje na temat dokonanych pomiarów za pomocą narzędzia NVIDIA Profiler ze zwiększonym rozmiarem macierzy wynikowej:

Na podstawie wyników pochodzących z nowego eksperymentu, widać poprawę względem wyników pochodzących z macierzy domyślnych.

Obserwacje dla bloku 16x16:

Dla wyników otrzymanych po wykorzystaniu większego rozmiaru macierzy wynikowej, dla bloku 16x16 wariantem z największą wydajnością okazuje się rozmiar kafelka 6x6. W przypadku rozmiaru macierzy 50 * 2 * BLOCK_SIZE, 50 * 2 * BLOCK_SIZE było identycznie. Dzięki zwiększeniu rozmiaru macierzy, udało się bardziej nasycić GPU. Sm_efficiency wynosi 98,87%, co wskazuje na wzrost o 4%.

Podobnie jak dla wcześniejszego rozmiaru, tutaj dla bloku 16x16, najgorszym wariantem jest kafelek o rozmiarze 23x24 oraz 24x24. Wyniki pochodzące z eksperymentów na macierzy wynikowej 4096x4096 wyraźnie pokazują nieefektywne wykorzystanie zasobów GPU dla maksymalnie granicznej liczby wyników obliczonych przez jeden wątek. Widać to przy parametrze sm_efficiency, gdzie dla kafelka 23x24 i 24x24, nastąpiło OVERFLOW w kontekście efektywności multiprocesora. Overflow wskazuje albo na przekroczenie limitów rejestrów albo wyczerpanie pamięci współdzielonej lub po prostu naruszenie architektury i parametrów GPU. Wydajność multiprocesora wynika nie tylko z rozmiaru

bloku, ale przede wszystkim całkowitego obciążenia GPU. W przypadku zwiększenia rozmiaru macierzy wynikowej, czyli zwiększenia liczby bloków wątków multiprocesory są bardziej (w tym przypadku maksymalnie) obciążone. Tysiące bloków konkuruje o zasoby GPU przez co profiler nie jest w stanie przeprowadzić rzetelnych metryk i wypisuje overflow.

W przypadku intensywności arytmetycznej, jej wzrost i spadek objawiają się identycznie jak przy macierzy wynikowej 50 * 2 * BLOCK_SIZE, 50 * 2 * BLOCK_SIZE, czyli zauważalny jest liniowy wzrost a dla granicznych rozmiarów kafelka 23x24 oraz 24x24, następuje drastyczny spadek w postaci wartości 0,5 flop / bytes.

Dodatkowo, potwierdza się taki sam trend - wraz ze wzrostem rozmiaru bloku maleje liczba transakcji pamięci globalnej i wzrasta efektywność dostępu. Szczególnie niepokojące są wyniki dla największych bloków, gdzie transakcje L2 i DRAM osiągają wartości rzędu miliardów, co wskazuje na problemy z zarządzaniem pamięcią. Wariant 23x24 wykazuje 9,77E+09 transakcji odczytu L2 oraz 9,61E+09 transakcji odczytu DRAM, co znacznie przekracza wartości dla mniejszych bloków.

Obserwacje dla bloku 32x32:

Najlepszym wariantem pod względem ogólnej wydajności, podobnie jak dla mniejszego rozmiaru macierzy wynikowej, okazał się układ 4x4. Osiągnął on najwyższą efektywność wynoszącą 3850 Gflops przy najkrótszym czasie wykonania 35,7 ms. Ten wariant charakteryzuje się również najwyższą wartością CGMA równą 252 flop na dostęp, co wskazuje na bardzo efektywne wykorzystanie przepustowości pamięci. Konfiguracja 4x4 osiąga wykorzystanie zasobów na poziomie 49,94% przy najwyższej wartości IPC wynoszącej 2.13, co świadczy o optymalnym wykorzystaniu jednostek obliczeniowych.

Widać też minimalną poprawę w wartościach sm_efficiency. Wyniki też dla większych bloków 32x32 pokazują, że odpowiednio dobrany rozmiar macierzy wynikowej jest w stanie bardziej nasycić GPU i powodować wzrost wydajności multiprocesora. Dla każdej liczby wyników który zmierzyliśmy widoczny jest wzrost sm_efficiency o maksymalnie 1%, czyli znacznie mniejszy niż w przypadku bloku 16x16.

Konfiguracja kafelka wyników 1x1 (1 wynik na wątek) wykazuje najwyższe teoretyczne wykorzystanie zasobów wynoszące 99,91%, jednak jego praktyczna wydajność jest znacznie gorsza niż większych konfiguracji. Osiąga efektywność jedynie 1160 Gflops przy czasie wykonania 119 ms, co czyni go najmniej efektywnym pomimo wysokiego wykorzystania GPU. Podobnie jak w przypadku mniejszego rozmiaru macierzy wynikowej.

Podobnie jak w przypadku domyślnego rozmiaru macierzy wynikowej 50 * 2 * BLOCK_SIZE, 50 * 2 * BLOCK_SIZE, wykonania pomiarów dla kafelków 5x6 oraz 6x6 skutkowała wynikiem FAIL. Dzieje się to z tych samych powodów. Są to na tyle graniczne wartości, które w praktyce nie mieszczą się w realnych limitach karty graficznej.

10. Wnioski wynikające z przeprowadzenia wszystkich eksperymentów i przeanalizowaniu wyników

Wnioski opierają się na przeprowadzonych eksperymentach uwzględniające różne rozmiary bloków wątków, różne liczby wyników na wątek i różny rozmiar macierzy wynikowej.

Użycie profilera i wykonanie programu z wykorzystaniem GPU w praktyce pokazało, że teoretyczne maksymalne liczby wyników na wątek nie są wykonalne w praktyce prawidłowo. Dla bloku 32x32, 6 wyników na wątek (teoretyczna maksymalna wartość) jest nieosiągalne dla GPU. W teorii, karta graficzna umożliwia uruchomienie takich obliczeń, ale w praktyce limity karty graficznej zostają przekroczone. Z tego powodu też nie mamy widocznych wyników dla kafelków 5x6 oraz 6x6 w bloku o rozmiarze 32x32. Takie eksperymenty kończyły się wynikiem “FAIL”. Dla bloku 16x16, możemy obliczyć więcej wyników na wątek, ale dla kafelków 23x24, 24x24 (teoretycznie maksymalnej wartości) występuje zjawisko OVERFLOW lub niezadowalające wyniki. Niemożliwość wykonania kernela z kafelkami 5x6 oraz 6x6 na GPU może wynikać z braku dostępnej pamięci współdzielonej w bloku, której jest w przypadku naszej karty 49152 bajtów. Sam kafelek 6x6 w praktyce potrzebuje (32 * 32 * 4B * (6 + 6)) 49 152 bajty pamięci, czyli jest to teorytyczne maksimum. W praktyce kompilator dodatkowo potrzebuje pamięci współdzielonej na lokalne zmienne, metadane czy padding (dodatkowe miejsce w pamięci współdzielonej w celu zredukowania liczby konfliktów banków pamięci).

W porównaniu rozmiaru bloku 16x16 i 32x32 widać różnice, co pokazuje jaki wpływ ma dobór bloku. Dla przykładu, bloki 32x32 wykazują bardziej liniowy wzrost wydajności, podczas gdy bloki 16x16 mają bardziej wyraźną krzywą optimum.

Większe macierze (4096x4096) lepiej nasycają GPU, prowadząc między innymi do lepszego wykorzystania zasobów sprzętowych i wzrostu sm_efficiency o 1-4% w zależności od konfiguracji.

CGMA, czyli najważniejszy wskaźnik efektywności, wzrastający systematycznie z rozmiarem kafelka wyników. CGMA zmierzone za pomocą NVIDIA Profilera okazało się inne od tego, które policzyliśmy wzorami w sekcji teoretycznej. W praktyce CGMA okazuje się być około 2 razy większe w każdym przypadku. Taka zależność jest wynikiem oczekiwanym. Różnica w CGMA w praktyce a wyliczonym według wzoru polega na procesach optymalizacyjnych liczby transakcji odczytu do pamięci globalnej. W praktyce okazuje się, że otrzymujemy około 2 razy mniejszą liczbę transakcji odczytu dzięki Coalesced access, czyli redukcji dostępów do pamięci przez wątki w tej samej wiązce. Zasadą tutaj jest to, aby wątki odwoływały się do sąsiednich adresów w pamięci globalnej. Według NVIDII efektywność coalescing rzadko przekracza 50%, co by tłumaczyło, dlaczego liczba odczytów jest o połowę mniejsza niż ta wyliczona na kartce papieru. Mniejsza liczba odczytów z pamięci globalnej skutkuje większym CGMA względem teoretycznych obliczeń.

Intensywność arytymetyczna, czyli intensity, mierzone jako liczba operacji arytmetycznych / liczba bajtów pobranych z RAM wzrasta wraz z liczbą wyników obliczanych przez wątek. Wyniki pokazują, że intensywność arytmetyczna potrafi się zwiększyć aż o 550% względem rozmiaru kafelka 1x1.

Wysokie wykorzystanie zasobów, czyli achieved_occupancy (>90%) nie gwarantuje wysokiej wydajności. Dla bloku 16x16, optymalne warianty osiągają okupację 24-50% przy maksymalnej wydajności.

Wraz ze wzrostem rozmiaru kafelka liczba transakcji pamięci globalnej maleje nawet czterokrotnie.

Zwiększenie rozmiaru macierzy wynikowej prowadzi do pewnego ryzyka jakim jest overflow. Do policzenia większych macierzy potrzebna jest większa liczba bloków wątków które konkurują o zasoby GPU. Te ryzyko jest widoczne najczęściej tylko przy większych liczbach wyników na wątek np. przy rozmiarze kafelka 24x24.

Podsumowanie na podstawie przeprowadzonych eksperymentów:

Modyfikacja programu NVIDIA poprzez zwiększenie liczby wyników obliczanych przez wątek jest wysoce efektywna, przynosząc wzrost wydajności nawet o 548% względem wariantu 1x1. Optymalne rozmiary kafelków zależą od rozmiaru bloku i wymagają balansu między wykorzystaniem zasobów a lokalnością danych. Większe macierze lepiej wykorzystują możliwości GPU, ale wymagają ostrożności przy doborze parametrów ze względu na ograniczenia sprzętowe i narzędzi pomiarowych.