Jakub Ciulęba 148103 jakub.ciuleba@student.put.poznan.pl Rzeczywisty: 16.07.2023

Kamil Malinowski 148214 kamil.malinowski@student.put.poznan.pl Wymagany: 16.07.2023

Wtorek nieparzyste 11:45 grupa L12 Trzecia wersja sprawozdania

Przetwarzanie Równoległe

Programowanie w CUDA na NVIDIA GPU (PKG)

Wariant 1

3. Poniżej przedstawiono parametry karty biorącej udział w testach:

NVIDIA GeForce GTX 1660 Ti Max-Q

Układ scalony	TU116
Technologia	Turing
Computing Capability	7,5
SM	24
Rdzenie CUDA	1536
Jednostki zmiennoprzecinkowe	1536
Jednostki całkowitoliczbowe	1536
SFU	96
Pamięć	6 GB GDDR6
Szyna pamięci	192-bit
Przepustowość	288GB/s
TDP	120W
Powierzchnia krzemu	284mm^2
Proces technologiczny	12nm FFN
L1 Cache	64kB/SM
L2 Cache	1536 kB

Tabela 3.1

4. Opis zadania:

Poniżej znajduje się bazowy kod Nvidia:

Kod 4.1

```
template <int BLOCK_SIZE> __global__ void MatrixMulCUDA(float *C, float *A,
    float *B, int wA,
    int wB) {

  int bx = blockIdx.x;
  int by = blockIdx.y;

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int aBegin = wA * BLOCK_SIZE * by;

  int aEnd   = aBegin + wA - 1;

  int aStep  = BLOCK_SIZE;

  int bBegin = BLOCK_SIZE * bx;

  int bStep  = BLOCK_SIZE * wB;

  float Csub = 0;

  for (int a = aBegin, b = bBegin;
       a <= aEnd;
       a += aStep, b += bStep) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];

    __syncthreads();
#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k) {
      Csub += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
  }
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;}
```

Poniżej znajduje się nasza modyfikacja powyższego kodu:

Kod 4.2

```
template <int BLOCK_SIZE> __global__ void
matrixMulCUDA(float* C, float* A, float* B, int wA_B, int size_C)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int aBegin = wA_B * BLOCK_SIZE * 4 * by;
    int aEnd = aBegin + wA_B - 1;
    int aStep = BLOCK_SIZE;

    int bBegin = BLOCK_SIZE * 2 * bx;
    int bStep = BLOCK_SIZE * 2 * wA_B;

    float Csub[8] = { 0 };

    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
    {
        __shared__ float As[4 * BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][2 * BLOCK_SIZE];

        if (a + wA_B * ty + tx < wA_B * wA_B)
        {
            As[ty][tx] = A[a + wA_B * ty + tx];
            if (ty + BLOCK_SIZE < 2 * wA_B && a + wA_B * (ty + BLOCK_SIZE) + tx < wA_B * wA_B)
            {
                As[ty + BLOCK_SIZE][tx] = A[a + wA_B * (ty + BLOCK_SIZE) + tx];
                if (ty + 2 * BLOCK_SIZE < 3 * wA_B && a + wA_B * (ty + 2 * BLOCK_SIZE) + tx < wA_B * wA_B)
                {
                    As[ty + 2 * BLOCK_SIZE][tx] = A[a + wA_B * (ty + 2 * BLOCK_SIZE) + tx];
                    if (ty + 3 * BLOCK_SIZE < 4 * wA_B && a + wA_B * (ty + 3 * BLOCK_SIZE) + tx < wA_B * wA_B)
                    {
                        As[ty + 3 * BLOCK_SIZE][tx] = A[a + wA_B * (ty + 3 * BLOCK_SIZE) + tx];
                    }
                }
            }
        }
    if (b + wA_B * ty + tx < wA_B * wA_B)
    {
        Bs[ty][tx] = B[b + wA_B * ty + tx];
        if (tx + BLOCK_SIZE < wA_B && b + wA_B * ty + tx + BLOCK_SIZE < wA_B * wA_B)
        {
            Bs[ty][tx + BLOCK_SIZE] = B[b + wA_B * ty + tx + BLOCK_SIZE];
        }
    }

    __syncthreads();

    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        if (k + ty < wA_B && k + tx < wA_B)
        {
            Csub[0] += As[ty][k] * Bs[k][tx];
            Csub[4] += As[ty + 2 * BLOCK_SIZE][k] * Bs[k][tx];
            if (k + tx + BLOCK_SIZE < wA_B)
            {
                Csub[2] += As[ty][k] * Bs[k][tx + BLOCK_SIZE];
                Csub[6] += As[ty + 2 * BLOCK_SIZE][k] * Bs[k][tx + BLOCK_SIZE];
            }
        }
        if (k + ty + BLOCK_SIZE < 2 * wA_B && k + tx < wA_B)
        {
            Csub[1] += As[ty + BLOCK_SIZE][k] * Bs[k][tx];
            Csub[5] += As[ty + 3 * BLOCK_SIZE][k] * Bs[k][tx];
            if (k + tx + BLOCK_SIZE < wA_B)
            {
                Csub[3] += As[ty + BLOCK_SIZE][k] * Bs[k][tx + BLOCK_SIZE];
                Csub[7] += As[ty + 3 * BLOCK_SIZE][k] * Bs[k][tx + BLOCK_SIZE];
            }
        }
    }
```

Kod 4.2

c.d

```
    int c = wA_B * BLOCK_SIZE * 4 * by + BLOCK_SIZE * 2 * bx;

    if (c + wA_B * ty + tx < wA_B * wA_B)
    {
        C[c + wA_B * ty + tx] = Csub[0];
        if (c + wA_B * (ty + BLOCK_SIZE) + tx < wA_B * wA_B)
        {
            C[c + wA_B * (ty + BLOCK_SIZE) + tx] = Csub[1];
            if (c + wA_B * (ty + 2 * BLOCK_SIZE) + tx < wA_B * wA_B)
            {
                C[c + wA_B * (ty + 2 * BLOCK_SIZE) + tx] = Csub[4];
                if (c + wA_B * (ty + 3 * BLOCK_SIZE) + tx < wA_B * wA_B)
                {
                    C[c + wA_B * (ty + 3 * BLOCK_SIZE) + tx] = Csub[5];
                }
            }
        }
        if (c + wA_B * ty + tx + BLOCK_SIZE < wA_B * wA_B)
        {
            C[c + wA_B * ty + tx + BLOCK_SIZE] = Csub[2];
            if (c + wA_B * (ty + BLOCK_SIZE) + tx + BLOCK_SIZE < wA_B * wA_B)
            {
                C[c + wA_B * (ty + BLOCK_SIZE) + tx + BLOCK_SIZE] = Csub[3];
                if (c + wA_B * (ty + 2 * BLOCK_SIZE) + tx + BLOCK_SIZE < wA_B * wA_B)
                {
                    C[c + wA_B * (ty + 2 * BLOCK_SIZE) + tx + BLOCK_SIZE] = Csub[6];
                    if (c + wA_B * (ty + 3 * BLOCK_SIZE) + tx + BLOCK_SIZE < wA_B * wA_B)
                    {
                        C[c + wA_B * (ty + 3 * BLOCK_SIZE) + tx + BLOCK_SIZE] = Csub[7];
                    }
                }
            }
        }
    }
}
```

Pierwsza funkcja to typowa implementacja mnożenia macierzy przy użyciu pamięci współdzielonej na GPU. W pamięci tej są przechowywane części (podmacierze) macierzy wejściowych, które są ładowane z pamięci globalnej GPU. Po załadowaniu tych podmacierzy, każdy wątek oblicza element wynikowej podmacierzy, mnożąc odpowiednie elementy macierzy wejściowych. Proces ten jest powtarzany dla wszystkich podmacierzy, aż do wyliczenia całej macierzy wynikowej.

Druga funkcja jest modyfikacją pierwszej, zmierzającą do optymalizacji wykorzystania pamięci współdzielonej na GPU. Prowadzi to do zwiększenia przepustowości obliczeń. Do najważniejszych zmian należą:

1. Zmiana rozmiaru pamięci współdzielonej macierzy `As` z `BLOCK_SIZE x BLOCK_SIZE` na `4 * BLOCK_SIZE x BLOCK_SIZE` i macierzy `Bs` na `BLOCK_SIZE x 2 * BLOCK_SIZE`. Pozwala to na przechowywanie ośmiu podmacierzy macierzy jednocześnie.

2. Wprowadzenie dodatkowych zmiennych `Csub[8]`, która przechowuje wynik mnożenia dla ośmiu podmacierzy.

3. Zmieniono operacje ładowania danych do pamięci współdzielonej tak, aby każdy wątek ładuje teraz cztery elementy macierzy A i dwa elementy macierzy B.

4. Modyfikacja pętli mnożenia tak, aby obliczała teraz osiem elementów wynikowej macierzy.

W efekcie tych zmian, nowa funkcja jest w stanie przetwarzać osiem podmacierzy jednocześnie, zamiast tylko jednej. Może to prowadzić do zwiększenia przepustowości obliczeń, szczególnie na GPU o wysokiej przepustowości pamięci, które są w stanie obsłużyć dodatkowe obciążenie związane z jednoczesnym przetwarzaniem ośmiu podmacierzy.

Podsumowując, poprawki wprowadzone w drugiej funkcji mają na celu lepsze wykorzystanie pamięci współdzielonej na GPU, co może prowadzić do zwiększenia wydajności obliczeń mnożenia macierzy.

5. Kluczowe fragmenty kodów

a) Zmienne i instrukcje:

- `blockIdx`, `blockDim`, `threadIdx` to predefiniowane zmienne CUDA, które reprezentują indeksy bloków i wątków oraz rozmiar bloków, odpowiednio.
- `BLOCK_SIZE` to rozmiar bloku wątków (2D), którzy współpracują przy przetwarzaniu podmacierzy o tym samym rozmiarze.
- `aBegin`, `aEnd`, `aStep`, `bBegin`, `bStep` służą do indeksowania podmacierzy A i B, które są przetwarzane przez blok wątków.
- `__shared__` oznacza, że zmienna jest przechowywana w pamięci współdzielonej na GPU, która jest szybsza niż pamięć globalna, ale ma ograniczoną pojemność i jest dostępna tylko dla wątków w tym samym bloku.
- `__syncthreads()` to funkcja CUDA, która synchronizuje wszystkie wątki w bloku, zapewniając, że wszystkie operacje pamięci wykonane przez wątki przed tą instrukcją są widoczne dla wszystkich wątków, które ją wywołują.

b) **Jakość dostępu do użytej pamięci:**

- Kod ładuje dane z pamięci globalnej do pamięci współdzielonej (`As`, `Bs`), minimalizując tym samym dostęp do pamięci globalnej, która jest stosunkowo wolna.
- Zastosowanie pamięci współdzielonej zamiast bezpośredniego dostępu do pamięci globalnej poprawia przepustowość pamięci dzięki łączeniu dostępu do pamięci. Wątki w bloku, które odczytują sąsiednie dane, mogą łączyć swoje odczyty w jeden, co jest efektywniejsze.
- Jest też problem konfliktów banków pamięci, które mogą wystąpić, gdy różne wątki próbują uzyskać dostęp do tego samego banku pamięci.

c) **Znaczenie poszczególnych synchronizacji występujących w kodzie**

- `__syncthreads()` jest używane po wczytaniu danych do `As` i `Bs`, aby upewnić się, że wszystkie wątki w bloku załadowały swoje dane przed rozpoczęciem mnożenia.

- Drugie `__syncthreads()` jest używane po mnożeniu, aby zapewnić, że wszystkie obliczenia są zakończone przed kolejnym cyklem ładującym nowe dane do pamięci współdzielonej. Bez tej synchronizacji, wątki, które zakończyły obliczenia wcześnie, mogłyby zacząć wczytywać nowe dane, zanim wszystkie inne wątki zakończyły swoje obliczenia, co prowadziłoby do niepoprawnych wyników.

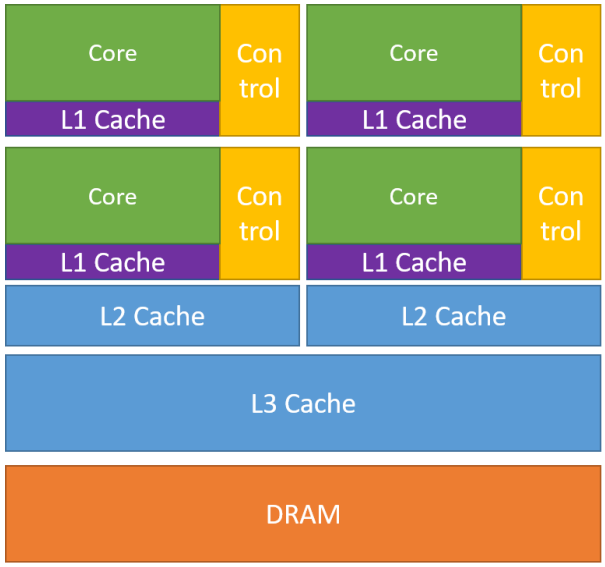
d) **Ilość przesyłanych danych pomiędzy pamięciami karty**

- Wszystkie elementy macierzy A i B są wczytywane z pamięci globalnej do pamięci współdzielonej, a wynik mnożenia macierzy jest zapisywany z powrotem do pamięci globalnej. W ten sposób minimalizujemy ilość przesyłanych danych, ograniczając dostęp do pamięci globalnej, co jest kosztowne pod względem wydajności.

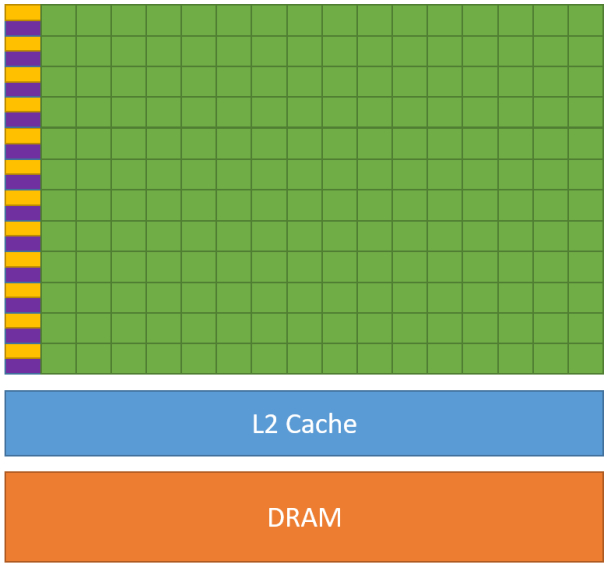
e) **Opis sposobu określania konfiguracji uruchomienia kernela**

- Konfiguracja uruchomienia kernela jest określana przez trzy parametry: ilość bloków, rozmiar bloku oraz rozmiar pamięci współdzielonej. Ilość bloków i rozmiar bloku zwykle są dobierane tak, aby maksymalnie wykorzystać dostępne zasoby sprzętowe - np. procesory strumieniowe na GPU. Rozmiar pamięci współdzielonej jest zwykle dobierany w zależności od rozmiaru danych, które muszą być załadowane. W przypadku tego kodu, bloki są dwuwymiarowe, a ich rozmiar jest określany przez parametr `BLOCK_SIZE`, a ilość bloków jest obliczana na podstawie rozmiaru macierzy wejściowych A i B. Pamięć współdzielona jest używana do przechowywania podmacierzy A i B, które są przetwarzane przez każdy blok.

6. Rysunki



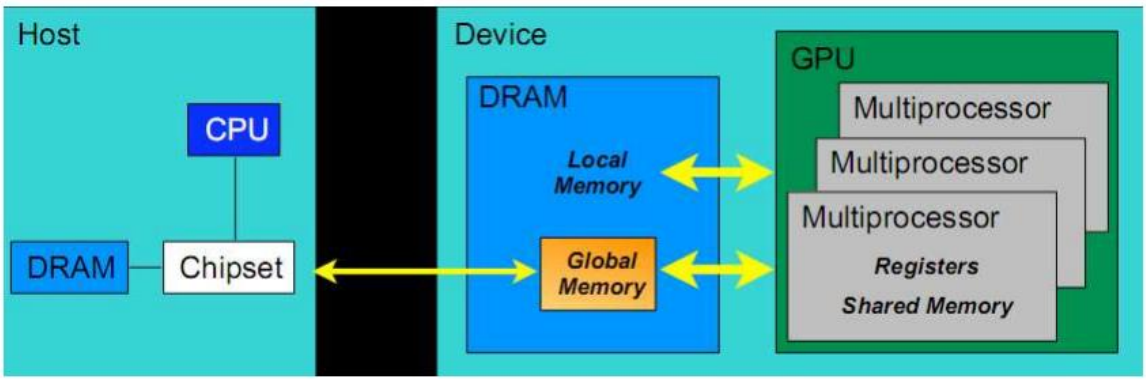
CPU



GPU

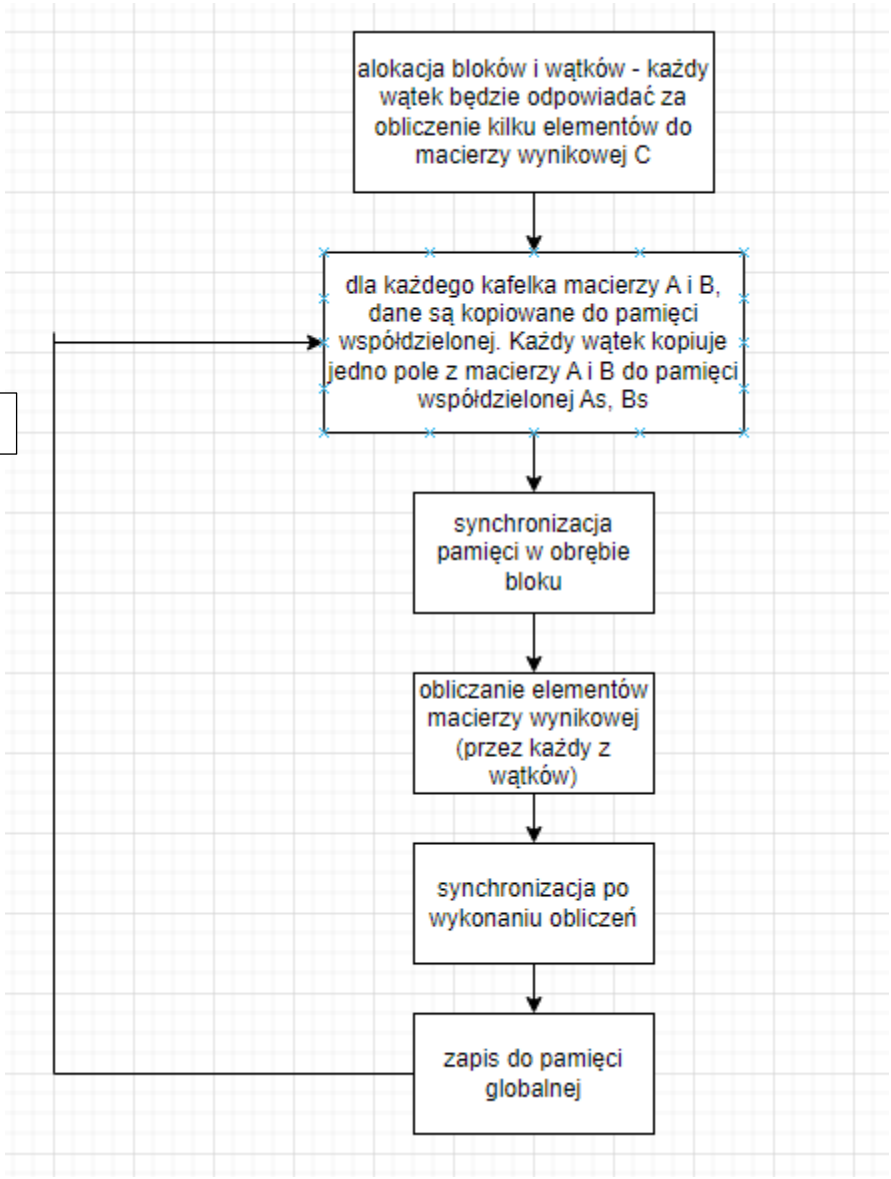
Różnica architektury pomiędzy CPU i GPU

Rysunek 6.1



Rysunek 6.2

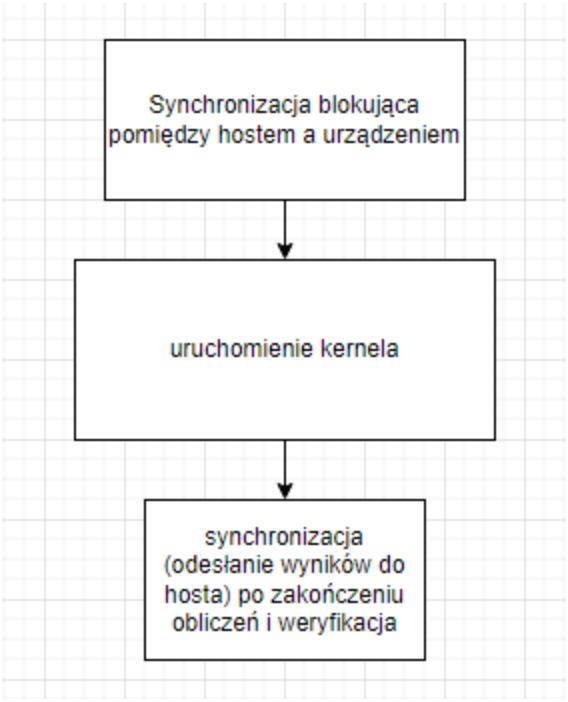
Kod procesora zarządza pamięcią karty graficznej, tzn. alokacja, zwolnienie, kopiowanie danych pomiędzy pamięciami pomiędzy urządzeniem, a hostem.



Rysunek 6.3

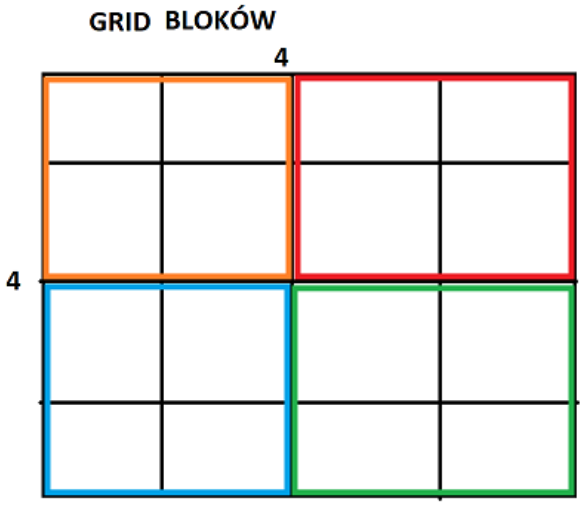
miejsce dostępu i kolejność dostępu do danych

Rysunek 6.4



Schemat ilustrujący komunikacje synchroniczną, pomiędzy hostem a urządzeniem

Model synchroniczny nie przewiduje komunikacji pomiędzy wykonywaniem obliczeń między hostem a urządzeniem.



BS=2, K=4, L=2

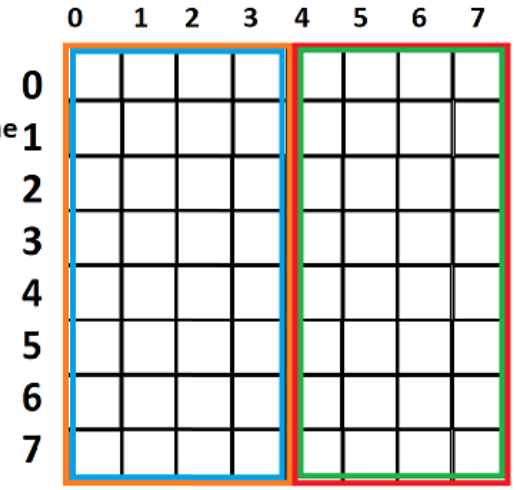
Dla każdego różnego koloru, jest osobne wywołanie kernela

Rysunek 6.5

Rysunek 6.6

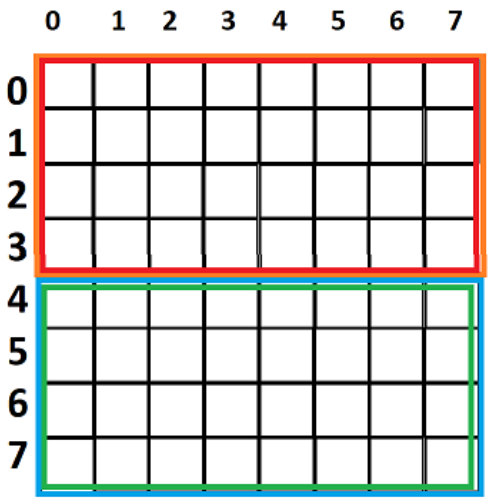
Wymagane do pobrania dane do PW nie są ciągłe w PG co powoduje niskej jakości dostęp do danych.

B

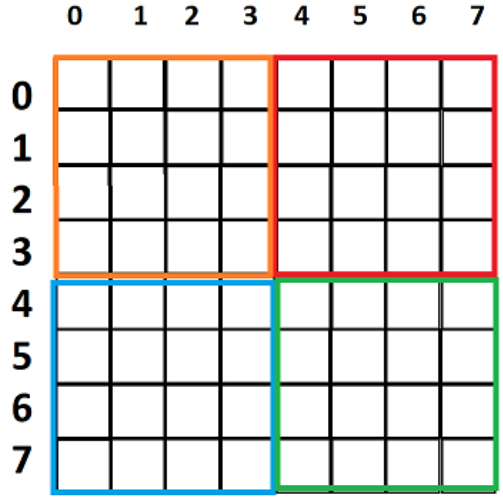


W macierzy A dostęp do danych jest w pełni łączony ze względu na ciągłość danych w pamięci głównej.

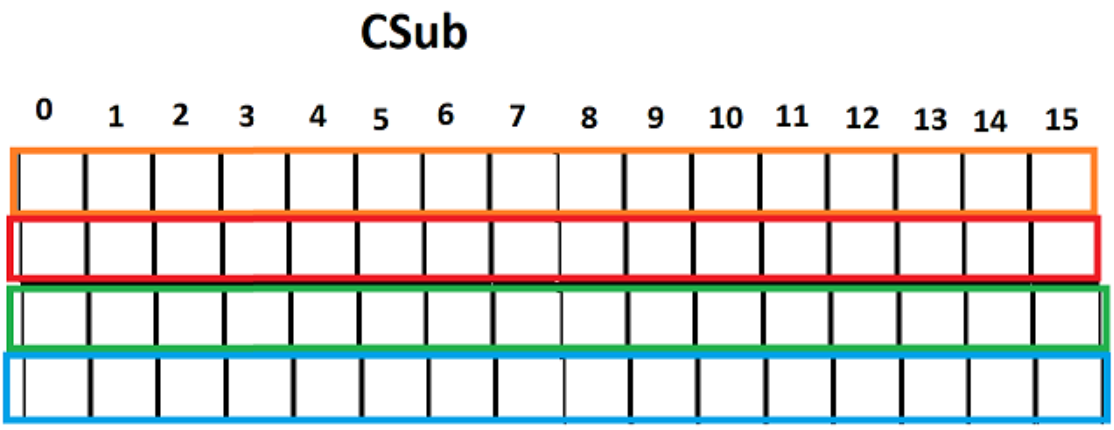
A



C

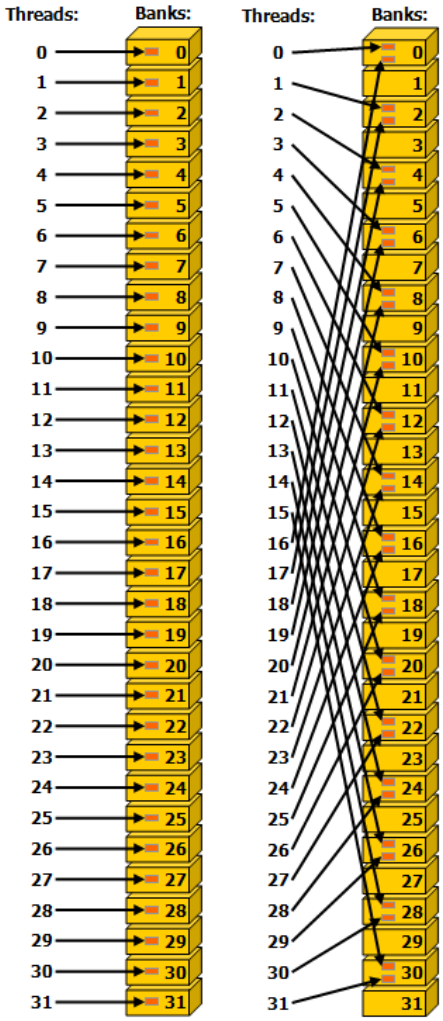


Rysunek 6.7



Do tablicy CSub będą dodawane wartości obliczone przez jeden wątek. Tablica CSub reprezentuje lokalną część macierzy wynikowej C. Wątek 0,0 będzie mnożył pozycje A[0,0] B[0,0], A[0,2], B[2,0] itd. I będzie dodawał wyniki do CSub[0] lub innych odpowiadających elementów tablicy CSub

Rysunek 6.8



Po lewej – dostęp prawidłowy

Po prawej – konflikt bankowy, dostęp (więcej niż jednego wątku z wiązki) do tego samego banku pamięci

7. Wzory

$$P = \frac{2 * N^3}{T}$$

Wzór 7.1

P – prędkość

N – rozmiar

T – czas jednokrotnego uruchomienia kernela

$$a = \frac{t_{CPU}}{t_{GPU}}$$

Wzór 7.2

gdzie:

a – przyspieszenie,

$tGPU$ – czas przetwarzania na karcie graficznej,

$tCPU$ – czas przetwarzania procesora

8. Znaczenie miar w programie Nvidia Nsight Compute

1. **Czas obliczeń**: To czas, w jakim wykonane są zadane obliczenia na procesorze graficznym.

2. **Wydajność obliczeń**: Jest to liczba operacji zmiennoprzecinkowych (flop) na sekundę, które GPU może wykonać. Jest to standardowa miara wydajności GPU.

3. **Arithmetic Intensity (CGMA)**: Jest to stosunek operacji zmiennoprzecinkowych do ilości danych przesyłanych do i z pamięci. Ta miara jest ważna, ponieważ operacje zmiennoprzecinkowe i operacje na pamięci często są ograniczeniem wydajności.

4. **Przepustowość obliczeń SM (Compute (SM) throughput)**: Przepustowość odnosi się do ilości danych, które mogą być przetwarzane przez jednostki obliczeniowe GPU (SM - streaming multiprocessors) w jednostce czasu.

5. **Przepustowość systemu pamięci (Memory throughput)**: Jest to miara tego, jak szybko GPU może czytać/zapisywać dane do swojej pamięci.

6. **L1/L2 hit rate**: Jest to procent operacji odczytu/zapisu, które mogą być obsłużone bezpośrednio z pamięci podręcznej L1/L2, zamiast musieć odczytywać dane z wolniejszej pamięci.

7. **Wielkość transmisji z PG/PW**: Ilość danych odczytanych/zapisanych do pamięci globalnej (PG) lub pamięci współdzielonej (PW) w GPU.

8. **Liczba konfliktów w dostępie do PW (bank conflicts)**: Konflikty bankowe występują, gdy różne wątki próbują jednocześnie uzyskać dostęp do tego samego banku pamięci. Mogą one znacznie spowolnić wydajność.

9. **Wykonane instrukcje**: Jest to liczba instrukcji, które zostały wykonane podczas wykonania zadania.

10. **Grid_size, Block_size**: Te parametry odnoszą się do organizacji wątków w GPU. Grid to zbiór bloków wątków, a blok to zbiór wątków, które mogą być schedulowane i wykonane wspólnie.

11. **Liczba rejestrów na wątek**: Jest to liczba rejestrów dostępnych dla każdego wątku. Rejestry są bardzo szybką pamięcią dostępną bezpośrednio w jednostkach obliczeniowych GPU.

12. **Rozmiar PW użytej przez blok wątków**: Jest to ilość pamięci współdzielonej, która jest używana przez każdy blok wątków.

13. **Occupancy theoretical/achieved [%]**: Zajętość to miara tego, ile z dostępnych zasobów obliczeniowych GPU jest faktycznie wykorzystywane. Teoretyczna zajętość to maksymalne możliwe wykorzystanie, podczas gdy osiągnięta zajętość to rzeczywiste wykorzystanie.

14. **Ograniczenie na liczbę bloków**: Jest to maksymalna liczba bloków, które mogą być jednocześnie obsługiwane przez GPU. Może to zależeć od wielu czynników, takich jak ilość dostępnych rejestrów, dostępna pamięć współdzielona, liczba jednostek obliczeniowych (SM) itp.

9. Wyniki (przepraszamy za podzieloną tabelę, była za duża)

Tabela 9.1

	Czas obliczeń [ms]	Wydajność obliczeń[GFlop/s]	CGMA [flop/byte]	Przepustowość obliczeń SM [%]	Przepustowość systemu pamięci
Kod 4.1 8x8	169.038	343.01	12,3	55.24%	55.24%
Kod 4.1 16x16	106.572	544.07	12,3	74.37%	74.37%
Kod 4.1 32x32	92.802	624.79	12,31	74.5%	74.5%
Kod 4.2 8x8	275.78	210.25	12,33	82.68%	82.68%
Kod 4.2 16x16	235.865	245.83	12,33	94.47%	94.47%
Kod 4.2 32x32	231.915	250.01	12,33	92.30%	92.30%
CPU	9292	-	-	-	-

Tabela 9.2

	L1 hit rate	L2 hit rate	Liczba konfliktów w dostępie do PW	Wykonane instrukcje	Grid size	Block size	Liczba rejestrów na wątek
Kod 4.1 8x8	98.8%	28.32%	0	3,24E+09	147456	64	42
Kod 4.1 16x16	0.04%	48.91%	0	2,84E+09	36864	256	39
Kod 4.1 32x32	0%	47.84%	0	2,44E+09	9216	1024	44
Kod 4.2 8x8	17.20%	68.01%	56 623 104	1,70E+10	73728	64	38
Kod 4.2 16x16	0.38%	63.43%	28 311 552	1,75E+10	18432	256	41
Kod 4.2 32x32	0%	62.98%	0	1,69E+10	4608	1024	41
CPU	-	-	-	-	-	-	-

Tabela 9.3

	Rozmiar PW użytej na blok wątków [B]	Occupancy theoretical/achieve	Ograniczenie na liczbę bloków	Ograniczenie na liczbę bloków PW	Ograniczenie na liczbę bloków	Ograniczenie na liczbę bloków SM
Kod 4.1 8x8	512	100%/99.87%	20	64	16	16
Kod 4.1 16x16	2099	100%/99.87%	6	16	4	16
Kod 4.1 32x32	8386	100%/99.98%	1	4	1	16
Kod 4.2 8x8	1577	100%/99.83%	24	21	16	16
Kod 4.2 16x16	6287	100%/99.84%	5	5	4	16
Kod 4.2 32x32	25169	100%/99.99%	1	1	1	16
CPU	-	-	-	-	-	-

	Przyspieszenie względem CPU
Kod 4.1 8x8	54,96988843
Kod 4.1 16x16	87,18988102
Kod 4.1 32x32	100,1271524
Kod 4.2 8x8	33,69352382
Kod 4.2 16x16	39,39541687
Kod 4.2 32x32	40,06640364
CPU	1

Tabela 9.4

10. Wnioski

1. Zdecydowanie najwyższą wydajność obliczeń obserwujemy dla kodu 4.1 o rozmiarze bloku 32x32. Może to wynikać z lepszego wykorzystania multiprocesorów przez większą liczbę wątków w bloku. W przypadku mniejszych bloków (8x8) zarówno dla kodu 4.1 Nvidia, jak i kodu 4.2, widzimy niższe wyniki wydajności, co może wynikać z nieefektywnego wykorzystania dostępnych zasobów. [tabela 9.1]

2. Wszystkie kody GPU mają podobne wartości CGMA, co sugeruje, że stosunek liczby operacji do liczby danych jest zbliżony dla wszystkich tych kodów. [tabela 9.1]

3. Kod 4.2 we wszystkich wariantach był wolniejszy od kodu 4.1 dla odpowiadających rozmiarów bloków, jednakże jego przepustowość obliczeń, a także przepustowość systemu pamięci była większa dla kodu 4.2. Może to oznaczać, że kod 4.2 jest bandwidth-bound, tzn. kod wykonuje tak wiele operacji odczytu/zapisu do pamięci, że w większości czeka na zakończenie tych operacji, a nie na właściwym wykonywaniu obliczeń.

4. Większy rozmiar PW użytej na blok wątków wynika z większej ilości ładowanych danych do pamięci.

5. Różnica w grid size (dwa razy mniejszy rozmiar dla kodu 4.2 w porównaniu do 4.1 dla każdego wariantu block size) kod 4.2 przetwarza więcej danych za jednym razem, dzięki czemu może wykorzystać mniejszą siatkę, a kod 4.1 przetwarza mniej danych za jednym razem, więc potrzebuje większej siatki do przetworzenia tych samych danych.

Kod 4.1 lepiej wykorzystuje pamięć cache i unika konfliktów w dostępie do pamięci współdzielonej, co prowadzi do wyższej wydajności obliczeń. Z drugiej strony, kod 4.2 lepiej wykorzystuje zasoby procesora, ale jest hamowany przez konflikty w dostępie do pamięci współdzielonej. [tabela 9.2] Obie implementacje korzystają z pamięci współdzielonej dla przechowywania fragmentów macierzy, które są mnożone przez blok wątków. Kod 4.1 używa bloków o stałej wielkości, podczas gdy kod 4.2 próbuje naładować więcej danych do pamięci współdzielonej. To może tłumaczyć różnicę w hit rate dla pamięci podręcznej L1 dla kodu 4.1 w wersji 8x8: kod 4.1 w wersji 8x8 ma stały rozmiar bloku i prawdopodobnie lepiej korzysta z pamięci podręcznej, podczas gdy kod 4.2 we wszystkich wariantach musi się zmagać z nieregularnością i możliwością konfliktów w dostępie do pamięci współdzielonej.

Kod 4.2 dla wariantów 8x8 i 16x16 cechował się sporą liczbą konfliktów do pamięci współdzielonej, co nie miało miejsca dla kodu 4.1 w żadnym z wariantów, oraz dla kodu 4.2 w wersji 32x32.

Jeśli więcej niż jeden wątek próbuje odwołać się do tego samego banku pamięci w tym samym cyklu zegara, odwołania te muszą być obsługiwane sekwencyjnie, co prowadzi do opóźnień. [tabela 9.2]

Rozważmy teraz bloki o wymiarach 8x8. Kiedy wątki w bloku próbują odczytać dane z pamięci współdzielonej. W przypadku tablicy o wymiarach 8x8, wątki odwołują się do pamięci współdzielonej w sposób, który prowadzi do konfliktów bankowych, ponieważ wiele wątków próbuje odczytać dane z tego samego banku pamięci w tym samym cyklu zegara. [tabela 9.2]

Dla bloków o wymiarach 16x16, liczba konfliktów bankowych jest mniejsza. W przypadku tablicy o wymiarach 16x16, więcej wątków może odczytywać dane z różnych banków pamięci w tym samym cyklu zegara, co zmniejsza liczbę konfliktów bankowych. [tabela 9.2]

Dla bloków o wymiarach 32x32, liczba konfliktów bankowych wynosi 0, ponieważ każdy wątek odwołuje się do unikalnego banku pamięci. W przypadku tablicy o wymiarach 32x32, każdy wątek może odczytać dane z unikalnego banku pamięci w tym samym cyklu zegara, co eliminuje konflikty bankowe. [tabela 9.2]

Kod 4.2 jest bardziej złożony pod względem przepływu sterowania, z wieloma warunkami sprawdzającymi, czy dany wątek powinien wykonywać obliczenia i zapisywać wynik. Może to tłumaczyć różnicę w wykorzystaniu SM (streaming multiprocessors): zwiększona złożoność kodu może skutkować lepszym wykorzystaniem SM, ale może również prowadzić do niewystarczającego równoległego wykorzystania wątków (divergence). [tabela 9.2]

Kod 4.1 używa jednego wątku do obliczenia jednego elementu wynikowej macierzy, podczas gdy kod 4.2 używa jednego wątku do obliczenia kilku elementów. Ta strategia może przynieść korzyści pod względem wydajności, jeśli dodatkowe obliczenia można wykonać bez wprowadzania dodatkowego narzutu na synchronizację wątków lub zwiększania zużycia pamięci. Jednak może to również wpływać na hit rate pamięci podręcznej i przepustowość pamięci systemu, ponieważ jeden wątek musi ładować więcej danych.

Ostateczne wyniki przyspieszenia przedstawiono w tabeli 9.4. Dla kodów 4.1 i 4.2 najszybszym wariantem jest wariant 32x32.