Wydział Informatyki i Telekomunikacji

Przetwarzanie równoległe

Laboratorium
Termin:*X*

Mnożenie macierzy

Projekt 2 - *wersja 1*

Wymagany termin oddania: X

Rzeczywisty termin oddania: X

Autorzy:

Prowadzący zajęcia:
dr inż. Rafał Walkowiak

23 czerwca 2023

Spis treści

1 Opis użytej karty graficznej

Użyty sprzęt: **NVIDIA RTX 3070**, chip **GA104**

- Technologia **Ampere**
- CUDA Driver Version / Runtime Version: **11.6 / 11.6**
- CUDA Capability Major/Minor version number: **8.6**
- Total amount of global memoery: **8192 MBytes**
- (046) Multiprocessors, (128) CUDA Cores/MP: **5888 CUDA Cores**
- Total amount of shared memory per block: **49152 bytes**
- Total shared memory per multiprocessor: **102400 bytes**
- Total number of registers available per block: **65536**
- Warp size: **32**
- Maximum number of threads per multiprocessor: **1536**
- Maximum number of threads per block: **1024**

2 Opis użytego procesora

Uzyty sprzęt: **AMD Ryzen 5 5600X**

- Liczba wątków: **12**
- TDP: **65W**
- Proces litograficzny: **7nm**
- Taktowanie rdzenia: **3.7GHz**

3 Opis zakresu zrealizowanego zadania

Celem zadania byo porównanie efektywności przetwarzania dla programów CPU i GPU, optymalizacja i porównanie przebiegu przetwarzania na procesorach kart graficznych. Programy polegały na mnożeniu macierzy kwadratowych o takich samych rozmiarach. Dwa testowane rozmiary wynosiły 1024x1024 oraz 3072x3072. Dla programu CPU należało zrobić równoległe przetwarzanie metodą zagnieżdżonych 3 pętli, aby następnie wyniki tego przetwarzania były punktem odniesienia przy porównywaniu z efektywnością przetwarzania na karcie graficznej. Dla programu GPU należało zmodyfikować kod udostępniony przez NVIDIA matrixMul. Jedną z modyfikacji było zastosowanie techniki zero-copy, następnie zostały utworzone dwa warianty, jeden który korzystał z pamięci globalnej i podręcznej karty oraz drugi, który wykorzystywał dodatkowo pamięć współdzieloną bloku wątków. Dla programów GPU dodatkową wartością, którą zmienialiśmy podczas testów był rozmiar bloków wątków, które były równe 8x8, 16x16 oraz 32x32.

4 Prezentacja przygotowanych wariantów kodów

4.1 Program CPU

```
1  void multiply(int matrix_size, float *a, float *b, float *c)
2  {
3    int i,j,k;
4
5    #pragma omp parallel for
6      for(i=0; i<matrix_size; i++) {
7          for( int k=0; k<matrix_size; k++) {
8            for(int j=0; j<matrix_size; j++) {
9          c[i * matrix_size + j] += a[i * matrix_size + k] * b[k * matrix_size + j];
10       }
11     }
12   }
13 }
14
15 void fill_array(int matrix_size, float *array, float value)
16 {
17     for(int i=0; i<matrix_size; i++){
18         for(int j=0; j<matrix_size; j++){
19             array[i * matrix_size + j] = value;
20         }
21     }
22 }
23
24 int main(int argc, char **argv){
25     int matrix_size = MSIZE;
26     float *a = (float *)malloc((matrix_size * matrix_size) * sizeof(float));
27     float *b = (float *)malloc((matrix_size * matrix_size) * sizeof(float));
28     float *c = (float *)malloc((matrix_size * matrix_size) * sizeof(float));
29     fill_array(matrix_size, a, 1.0);
30     fill_array(matrix_size, b, 0.01);
31     fill_array(matrix_size, c, 0.0);
32     printf("matrix filled\n");
33     auto start = std::chrono::high_resolution_clock::now();
34     multiply(matrix_size, a, b, c);
35     auto end = std::chrono::high_resolution_clock::now();
36     auto int_s = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
37     std::cout << "time: " << int_s.count() << " miliseconds" << std::endl;
38     return 0;
39 }
```

Listing 1: Kod programu CPU

Kod programu CPU to proste mnożenie macierzy z wykorzystaniem trzech pętli **for**. Do zrównoleglenia została użyta dyrektywa **omp parallel for**. Wyniki programu posłużą do wyliczenia przyspieszenia programów GPU.

4.2 GPU

Konfiguracja uruchomienia kernela:

```
unsigned int size_A = dimsA.x * dimsA.y;
unsigned int mem_size_A = sizeof(float) * size_A;
float *h_A;
checkCudaErrors(cudaMallocHost(&h_A, mem_size_A));
checkCudaErrors(cudaHostGetDevicePointer((void **)&d_A_ptr, (void *)h_A, 0));
unsigned int size_B = dimsB.x * dimsB.y;
unsigned int mem_size_B = sizeof(float) * size_B;
float *h_B;
checkCudaErrors(cudaMallocHost(&h_B, mem_size_B));
checkCudaErrors(cudaHostGetDevicePointer((void **)&d_B_ptr, (void *)h_B, 0));
...
dim3 threads(block_size, block_size);
dim3 grid(dimsB.x / threads.x, dimsA.y / threads.y);
unsigned int mem_size_A_info = sizeof(bool) * grid.x * grid.y;
unsigned int mem_size_B_info = sizeof(bool) * grid.x * grid.y;
checkCudaErrors(cudaMalloc(reinterpret_cast<void **>(&d_A_info), mem_size_A_info));
checkCudaErrors(cudaMalloc(reinterpret_cast<void **>(&d_B_info), mem_size_B_info));

if (block_size == 8) {
    MatrixMulCUDA<8>
    <<<grid, threads, 0, stream>>>(d_C, d_A_ptr, d_B_ptr, d_A, d_B, d_A_info,
    d_B_info, dimsA.x, dimsB.x);
}
else if (block_size == 16) {
    MatrixMulCUDA<16>
    <<<grid, threads, 0, stream>>>(d_C, d_A_ptr, d_B_ptr, d_A, d_B, d_A_info,
    d_B_info, dimsA.x, dimsB.x);
} else {
    MatrixMulCUDA<32>
    <<<grid, threads, 0, stream>>>(d_C, d_A_ptr, d_B_ptr, d_A, d_B, d_A_info,
    d_B_info, dimsA.x, dimsB.x);
}
```

Listing 2: Konfiguracja uruchomienia kernela

Funkcja **cudaHostGetDevicePointer** pozwala uzyskać dane adresów danych CPU macierzy, aby były widoczne z poziomu kernela.

```
template <int BLOCK_SIZE> __global__ void MatrixMulCUDA(float *C, float *A_ptr,
    float *B_ptr, float *A_cache, float *B_cache, bool *A_info, bool *B_info, int wA
    , int wB) {
  // Block index
  int bx = blockIdx.x;
  int by = blockIdx.y;

  // Thread index
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int row = by * blockDim.y + ty;
  int col = bx * blockDim.x + tx;
  float C_local = 0;

  for (int m = 0; m < wA / BLOCK_SIZE; ++m) {
    if (!A_info[by * gridDim.x + m]){
      A_cache[row * wA + m*BLOCK_SIZE + tx] = A_ptr[row * wA + m*BLOCK_SIZE + tx];
    }
    if (!B_info[m * gridDim.x + bx]){
      B_cache[(m*BLOCK_SIZE + ty)*wA + col] = B_ptr[(m*BLOCK_SIZE + ty)*wA + col];
    }
    __syncthreads();
    A_info[by * gridDim.x + m] = true;
    B_info[m * gridDim.x + bx] = true;
    #pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k) {
      C_local += A_cache[row * wA + m*BLOCK_SIZE + k]
      *
      B_cache[(m*BLOCK_SIZE + k)*wA + col];
    }
  }

  C[row*wA + col] = C_local;
}
```

Listing 3: Kod programu GPU wariant 1

Argumenty:

- C - wskaźnik na macierz wynikową w pamięci globalnej karty graficznej
- A_ptr - wskaźnik na macierz A w pamięci CPU(o stałej lokacji), dane udostępniane dla programów realizowanych na CPU i GPU
- B_ptr - wskaźnik na macierz B w pamięci CPU(o stałej lokacji), dane udostępniane dla programów realizowanych na CPU i GPU
- A_cache - wskaźnik na macierz służącą do zapamiętywania danych pobieranych z A_ptr, zaalokowana w pamięci globalnej karty graficznej
- B_cache - wskaźnik na macierz służącą do zapamiętywania danych pobieranych z B_ptr, zaalokowana w pamięci globalnej karty graficznej

- A_info - wskaźnik na macierz służącą do zapamiętywania informacji, które dane zostały już pobrane z A_ptr, zaalokowana w pamięci globalnej karty graficznej
- B_info - wskaźnik na macierz służącą do zapamiętywania informacji, które dane zostały już pobrane z B_ptr, zaalokowana w pamięci globalnej karty graficznej
- wA - rozmiar macierzy A w osi OX
- wB - rozmiar macierzy B w osi OX

Zmienne:

- tx, ty - indeksy wątku w ramach bloku
- bx, by - indeksy bloku w ramach gridu
- row - numer wiersza w całej macierzy przypisany dla danego wątku, obliczony dla aktualnie przetwarzanego bloku
- col - numer kolumny w całej macierzy przypisany dla danego wątku, obliczony dla aktualnie przetwarzanego bloku
- C_local - zmienna lokalna do wyliczania komórek macierzy

Synchronizacja wątków następuje po skopiowaniu danych do pamięci globalnej karty graficznej, aby zapewnić skopiowanie danych przez wszystkie wątki.

```
template <int BLOCK_SIZE> __global__ void MatrixMulCUDA(float *C, float *A_ptr,
    float *B_ptr, float *A_cache, float *B_cache, bool *A_info, bool *B_info, int wA
    , int wB) {
  // Block index
  int bx = blockIdx.x;
  int by = blockIdx.y;

  // Thread index
  int tx = threadIdx.x;
  int ty = threadIdx.y;

 int row = by * blockDim.y + ty;
  int col = bx * blockDim.x + tx;
  float C_local = 0;

  for (int m = 0; m < wA / BLOCK_SIZE; ++m) {
    __shared__ float Ads[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bds[BLOCK_SIZE][BLOCK_SIZE];
    if (!A_info[by * gridDim.x + m]){
      A_cache[row * wA + m*BLOCK_SIZE + tx] = A_ptr[row * wA + m*BLOCK_SIZE + tx];
    }
    if (!B_info[m * gridDim.x + bx]){
      B_cache[(m*BLOCK_SIZE + ty)*wA + col] = B_ptr[(m*BLOCK_SIZE + ty)*wA + col];
    }
    Ads[ty][tx] = A_cache[row * wA + m*BLOCK_SIZE + tx];
    Bds[ty][tx] = B_cache[(m*BLOCK_SIZE + ty)*wA + col];
    __syncthreads();
    A_info[by * gridDim.x + m] = true;
    B_info[m * gridDim.x + bx] = true;
    #pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
      C_local += Ads[ty][k] * Bds[k][tx];
    __syncthreads();
  }

  C[row*wA + col] = C_local;
}
```

Listing 4: Kod programu GPU wariant 2

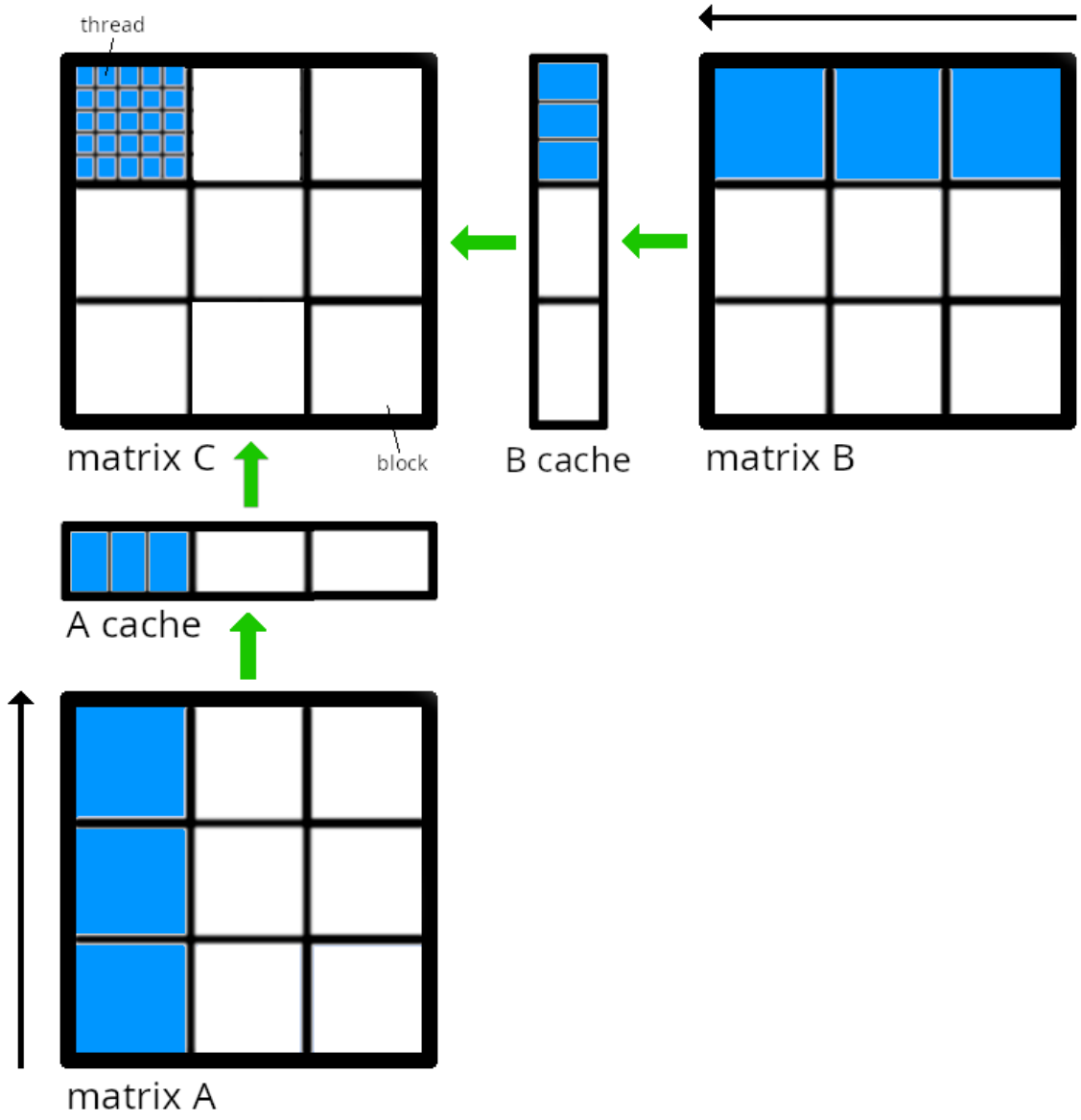
Zmienne:

- Ads - pamięć współdzielona na macierz A
- Bds - pamięć współdzielona na macierz B

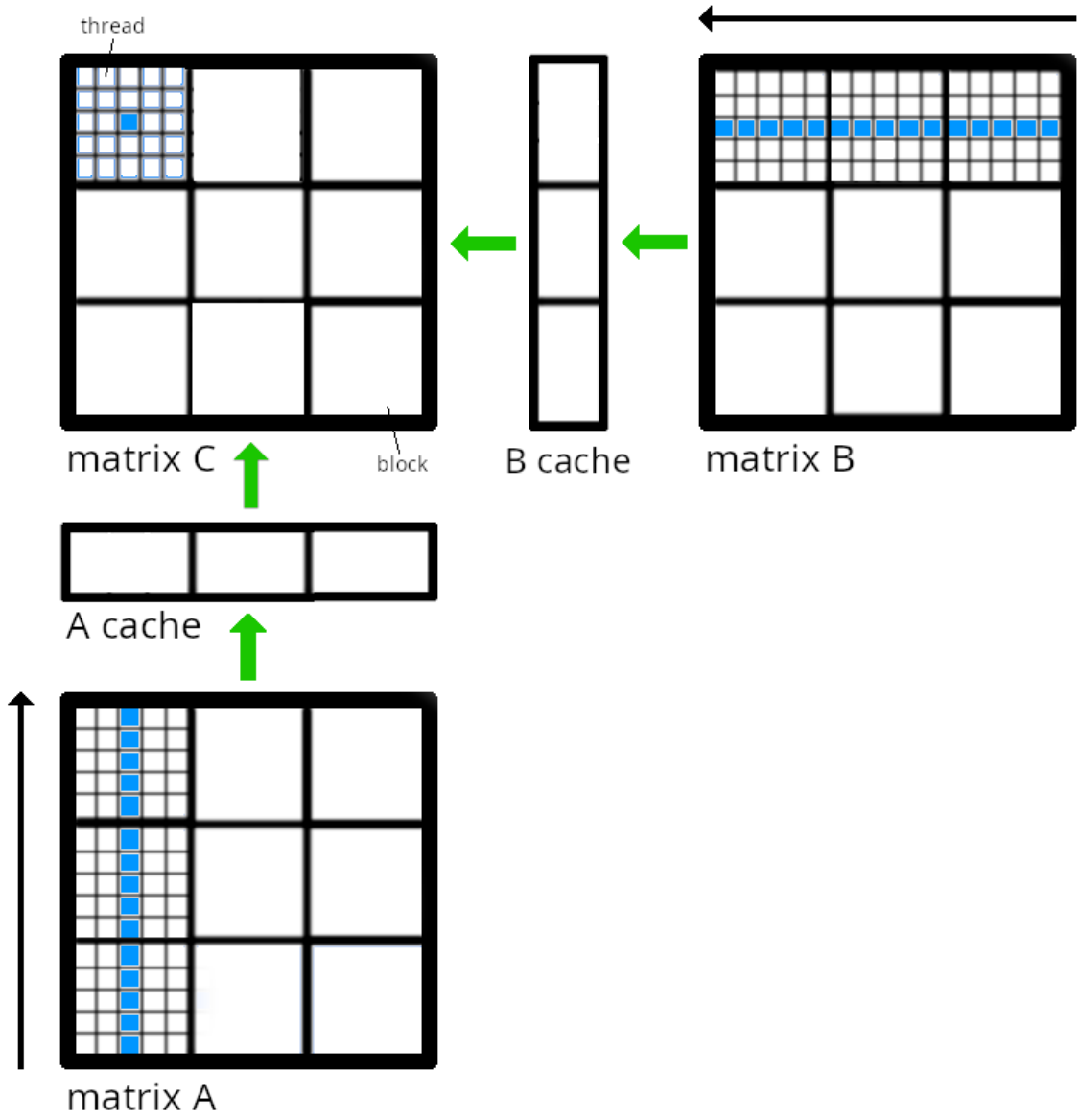
Synchronizacja wątków następuje po skopiowaniu danych do pamięci karty graficznej oraz po obliczeniach na dostępnych danych.

5 Rysunki

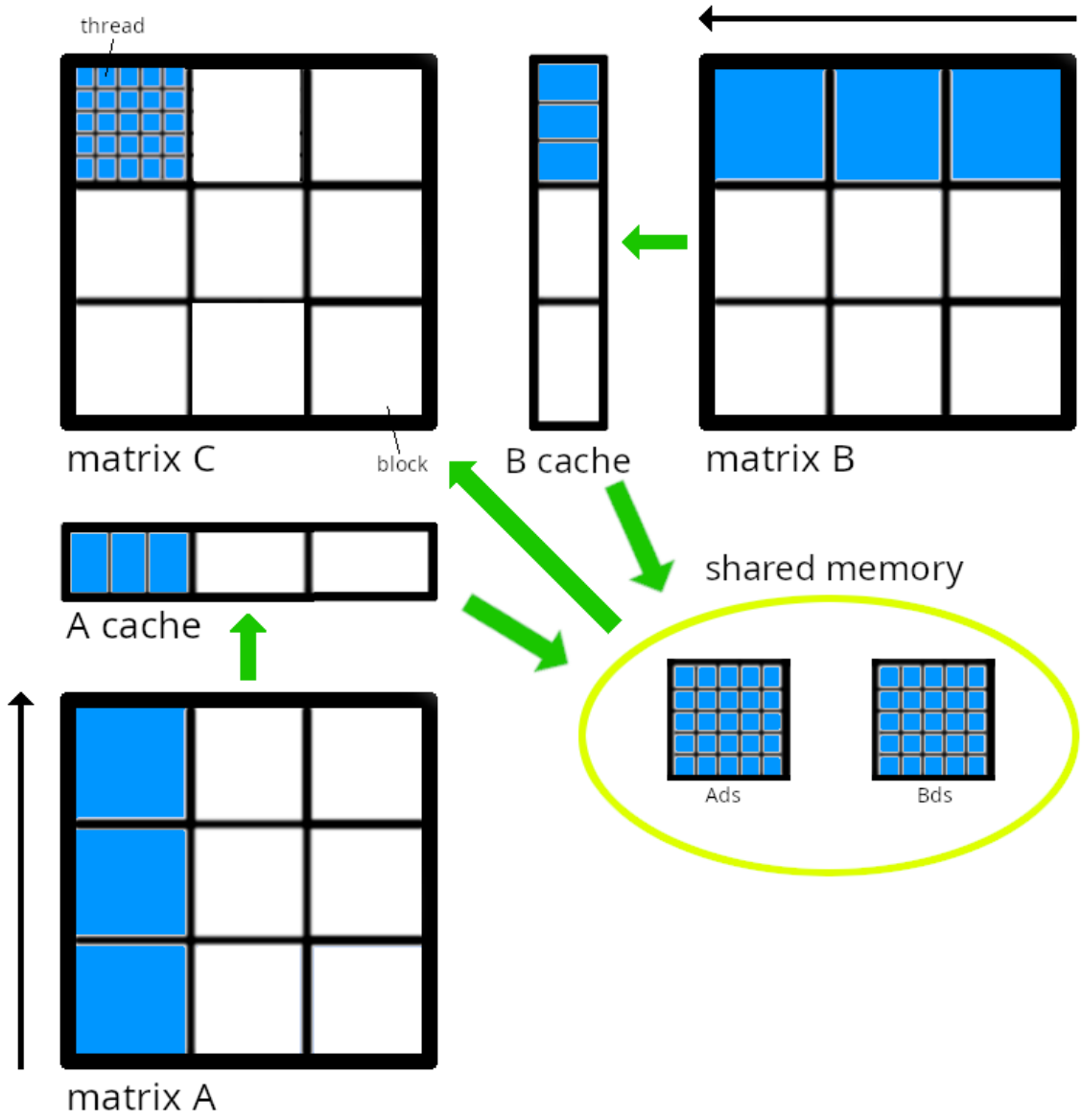
Zielone strzałki oznaczają przepływ danych, czarne strzałki przepływ przetwarzania.



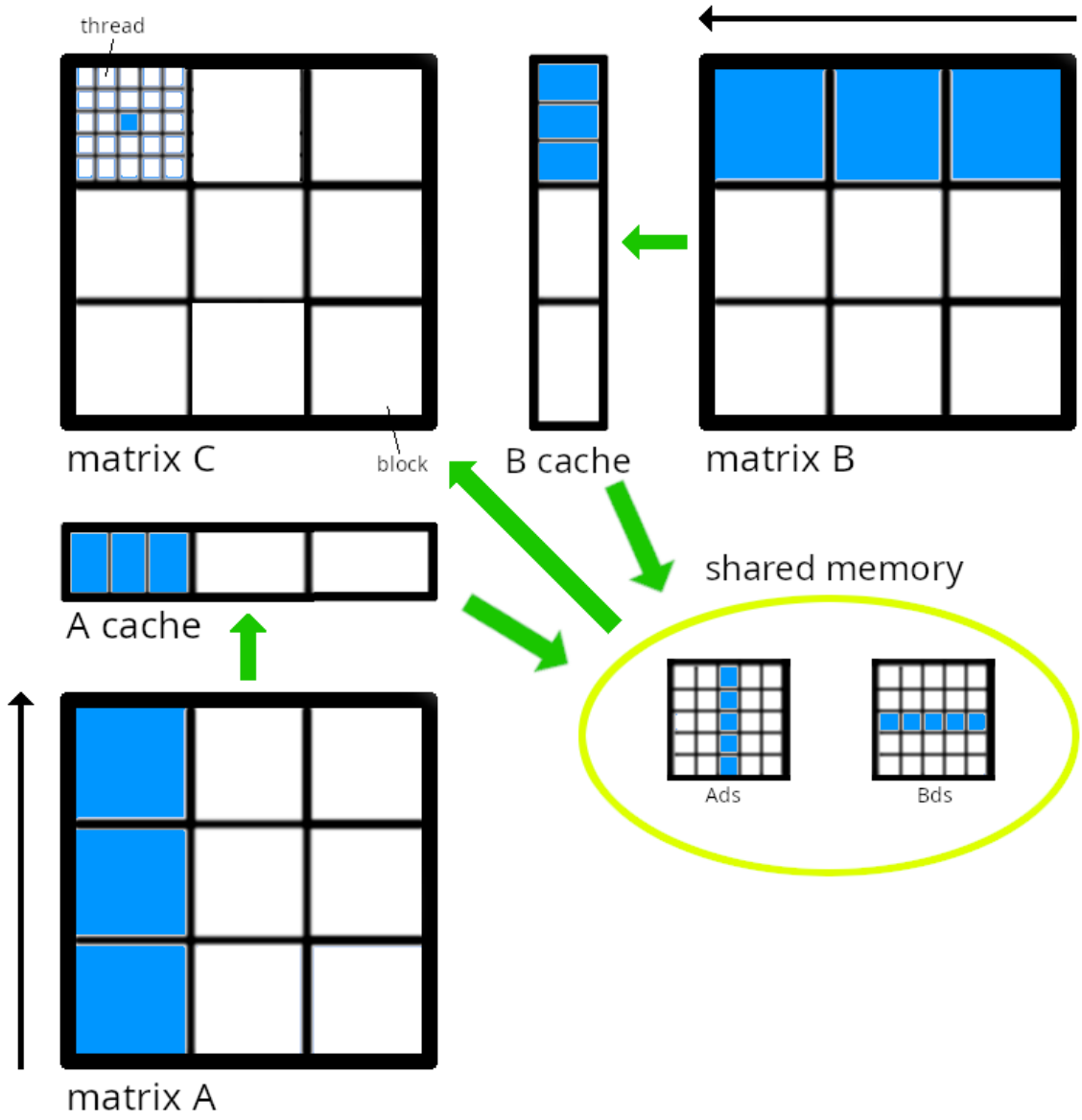
Rysunek 1: Kolejność i miejsce dostępu do danych, wartości wyników ze względu na bloki, m=3



Rysunek 2: Kolejność i miejsce dostępu do danych, wartości wyników ze względu na wątki, m=3



Rysunek 3: Kolejność i miejsce dostępu do danych, wartości wyników ze względu na bloki, m=3



Rysunek 4: Kolejność i miejsce dostępu do danych, wartości wyników ze względu na wątki, m=3

6 Zastosowane wzory

- t - czas przetwarzania - czas obliczeń
- SM - miara stopnia wykorzystania modułów SM (streaming multiprocessors), wyrażona w %
- Memory - miara stopnia wykorzystania pamięci, wyrażona w %
- Theoretical occupancy - teoretyczna zajętość multiprocesora, wyrażona w %
- Achieved occupancy - osiągnieta zajętość multiprocesora, wyrażona w %
- prędkość obliczeń - $\frac{2*N*N*N}{t}$
- a - przyspieszenie - iloraz czasu przetwarzania danego programu GPU do czasu przetwarzania programu na CPU
- CGMA - compute to global memory access ratio - miara uzyskana w programie Nsight Compute, jest to iloraz operacji arytmetycznych do dostępów do pamięci globalnej
- liczba bloków - $(\frac{rozmiar\_macierzy}{rozmiar\_bloku})^2$
- liczba rejestrów na wątek - miara uzyskana w programie Nsight Compute
- wymagana liczba rejestrów na blok - iloczyn liczby rejestrów na wątek oraz liczby wątków w bloku
- wymagana pamięć współdzielona na blok - miara uzyskana w programie Nvidia Occupancy Calculator, wyrażona w bajtach

7 Uzyskane wyniki

rozmiar macierzy	1024			3072		
liczba bloków	8	16	32	8	16	32
Metoda	CPU					
Czas przetwarzania [s]	0,042			0,836		
Metoda	GPU shared					
Czas przetwarzania [s]	0,00867	0,00372	0,00206	0,251443	0,12149	0,05812
SM	31,00%	52,00%	77,00%	24,00%	43,00%	74,00%
M	31,00%	52,00%	77,00%	24,00%	43,00%	74,00%
TO	66,67%	100,00%	66,67%	66,67%	100,00%	66,67%
AO	65,77%	98,49%	67,38%	66,76%	99,91%	66,64%
Prędkość obliczeń [flop/s]	2,47691E+11	5,77281E+11	1,04247E+12	2,31E+11	4,77E+11	9,97627E+11
Przyspieszenie	4,84	11,29	20,39	3,32	6,88	14,38
CGMA [flop/byte]	23,5	26,23	20,21	3,25	7,9	15,31
liczba bloków	16384	4096	1024	147456	36864	9216
liczba rejestrów na wątek	40	40	40	40	40	40
rejestry na blok	2560	10240	40960	2560	10240	40960
pamięć współdzielona na blok [B]	512	2176	8448	512	2176	8448
Metoda	GPU global					
Czas przetwarzania [s]	0,011526	0,005473	0,003336	0,29749	0,17014	0,10153
SM	53,00%	75,00%	86,00%	45,00%	57,00%	88,00%
M	55,00%	75,00%	86,00%	47,00%	57,00%	88,00%
TO	66,67%	100,00%	66,67%	66,67%	100,00%	66,67%
AO	67,11%	97,72%	66,95%	66,77%	100,00%	66,65%
Prędkość obliczeń [flop/s]	1,86316E+11	3,92378E+11	6,4373E+11	1,95E+11	3,41E+11	5,71083E+11
Przyspieszenie	3,64	7,67	12,59	2,81	4,91	8,23
CGMA [flop/byte]	21,82	29,94	20,6	3,31	7,9	15,24
liczba bloków	16384	4096	1024	147456	36864	9216
liczba rejestrów na wątek	40	40	40	40	40	40
rejestry na blok	2560	10240	40960	2560	10240	40960
pamięć współdzielona na blok [B]	0	0	0	0	0	0

Rysunek 5: Tabela z wynikami wariantu CPU i wariantami GPU

8 Wnioski

Z powyższych wyników możemy wywnioskować, że zastosowane warianty mnożenia macierzy na karcie graficznej są dużo bardziej wydajne od wariantów wykonywanych na procesorze.

Miary stopnia wykorzystania modułów SM oraz stopnia wykorzystania pamięci M wzrastały wraz z zwiększaniem rozmiaru bloków. Wynikało to między innymi z tego, że liczba bloków wtedy malała, dzięki czemu potrzebna była mniejsza ilość synchornizacji między wątkami. Również z powodu synchronizacji miary SM oraz M są wyższe dla wariantów GPU global, ponieważ były one pozbawione jednej synchronizacji, która była niezbędna dla wariantu shared.

Zajętości teoretyczna i praktyczna modułów (AO i TO) we w szystkich wariantach były do siebie bardzo zbliżone, jednak warto zauważyć, że tylko dla rozmiaru bloku 16 osiągały wartości bliskie 100%, gdzie dla pozostałych wariantów, były to wartości zbliżone do 66%. Wynikało to z różnych ograniczeń, które ujawniały się podczas raportów z Nsight Compute. Jednym z ograniczeń, które wyświetlały się w raporcie były ograniczenie liczby bloków na SM oraz ograniczenie liczbą wymaganych rejestrów. Warto również dodać, że na innych kartach graficznych zaobserwowaliśmy równe zajętości AO i TO na wszystkich wariantach, które były równe 100%, jednak osiągały gorsze miary dla SM oraz M.

Zmierzone wskaźniki CGMA przez program dla wariantów z rozmiarem macierzy 1024x1024 są zdecydowanie większe, niż dla wariantów z roamirem macierzy 3072x3072. Wynik to z tego, że w wariancie z większą macierzą było więcej danych do przekopiowania na pamięć globalną, dlatego dostępów do tej pamięci było zdecydowanie więcej. Przekłada się to również na to, że dla tego wariantu rozmiar bloku, zwiększał miarę CGMA. Dla wariantu global CGMA jest w niektórych przypadkach wyższe, wpływ na to może mieć kopiowanie przez nas danych najpierw do pamięci globalnej karty, a następnie z niej do pamięci współdzielonej w wariancie shared. Jedynym zaskoczeniem był nagły spadek miary CGMA dla wariantów z rozmiarem macierzy 1024x1024 oraz rozmiarem bloku 32.

Największe przyspieszenie zaobserwowaliśmy dla wariantu shared mnożenia macierzy o rozmiarze 1024x1024, przy ustawionym rozmiarze bloku 32. Wariant ten był 20 razy szybszy od wariantu CPU. Składa się na to ponownie rozmiar bloku oraz wykorzystanie pamięci współdzielonych, które cechują się szybkim dostępem. Dlatego też możemy stwierdzić, że wykorzystanie pamięci współdzielonych znacznie przekłada się na czasy przetwarzania, co widać na lepszych czasach wariantów shared nad global.

Spis rysunków

Listings