Sprawozdanie PROJEKT 2 CUDA

Nazwa zaliczanego przedmiotu: laboratorium z przetwarzania równoległego

Autorzy: Michał Olszewski 144482, grupa L11, środa 9:45, tygodnie nieparzyste,
Kacper Nawrot 145246, grupa L11, środa 9:45, tygodnie nieparzyste

Terminy: wymagany: 22.06.2022,
rzeczywisty: 15.06.2022

Wersja sprawozdania: pierwsza

Opis zadania:

- przygotować i wyjaśnić przebieg przetwarzania wymaganych wersji programów:
  - 0copy (wykorzystywanie pamięci przygotowanej przez procesor),
  - shared memory (wykorzystanie pamięci współdzielonej),

- wykonać eksperyment obliczeniowy dla przygotowanych kodów z pomiarem czasu przetwarzania dla zadanego zakresu instancji i parametrów uruchomienia,

- wykonać eksperyment pomiaru efektywności przetwarzania przy użyciu programu oceny efektywności przetwarzania Night Compute,

- wykorzystać uzyskane wyniki obliczeń i miary efektywności do porównania jakości przetwarzania i udowodnienia, że wymagany zakres projektu został wykonany.

Adresy email: michal.m.olszewski@student.put.poznan.pl,
kacper.nawrot@student.put.poznan.pl

1. **Opis użytej karty graficznej**

Procesor graficzny	
Model	ASUS ROG STRIX RTX 3080 GAMING OC
Architektura	Ampere
Litografia	8 nm
Tranzystory	28.3 miliona
Powierzchnia krzemu	628 mm^2
Magistrala	PCIe 4.0 x16
Pamięć	
Rozmiar pamięci	10 GB
Typ pamięci	GDDR6X
Magistrala pamięci	320 bit
Przepustowość	760.3 GB/s
Render Config (SFU)	
Shading Units	8704
TMUs (texture mapping unit)	272
ROPs (render output unit)	96
Liczba SM (streaming multiprocessor)	68
Liczba rdzeni Tensor	272
Liczba rdzeni RT	68
L1 Cache	128 KB (per SM)
L2 Cache	5 MB
Prędkości zegarów	
Base Clock	1440 Mhz
Boost Clock	1905 Mhz
Memory Clock	1188 Mhz 19 Gbps effective
Compute Capability 8.6	
Liczba specjalnych jednostek funkcyjnych dla zmiennoprzecinkowych funkcji transcendentalnych o pojedynczej precyzji	16
Liczba warp schedulerów	4
Maksymalna liczba instrukcji wydanych jednocześnie przez jednego schedulera	1
Liczba jednostek Tensor	4
Rozmiar w KB zunifikowanej pamięci dla pamięci podręcznej danych i pamięci współdzielonej na wiele procesorów	128
Brak ogarniczeń CUDA	

Wyniki device Query

CUDA Driver Version / Runtime Version	11.7 / 11.7
CUDA Capability Major/Minor version number:	8.6
Total amount of global memory:	10240 MBytes (10736893952 bytes)
(068) Multiprocessors, (128) CUDA Cores/MP:	8704 CUDA Cores
GPU Max Clock rate:	1905 MHz (1.90 GHz)
Memory Clock rate:	9501 Mhz
Memory Bus Width:	320-bit
L2 Cache Size:	5242880 bytes
Maximum Texture Dimension Size (x,y,z)	1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
Maximum Layered 1D Texture Size, (num) layers	1D=(32768), 2048 layers
Maximum Layered 2D Texture Size, (num) layers	2D=(32768, 32768), 2048 layers
Total amount of constant memory:	65536 bytes
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
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and kernel execution:	Yes with 5 copy engine(s)
Run time limit on kernels:	Yes
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Alignment requirement for Surfaces:	Yes
Device has ECC support:	Disabled
CUDA Device Driver Mode (TCC or WDDM):	WDDM (Windows Display Driver Model)
Device supports Unified Addressing (UVA):	Yes
Device supports Managed Memory:	Yes
Device supports Compute Preemption:	Yes
Supports Cooperative Kernel Launch:	Yes
Supports MultiDevice Co-op Kernel Launch:	No
Device PCI Domain ID / Bus ID / location ID:	0 / 10 / 0

2. Kluczowe fragmenty kodów kernel

```
constexpr int N = 3072;          // liczba kolumn i wierszy

float* A = new float[N * N];     // lewy operand
float* B = new float[N * N];     // pracy operand
float* C = new float[N * N];     // wynik

void initialize_matrices() {
    #pragma omp parallel for num_threads(omp_get_max_threads())
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i * N + j] = (float)rand() / RAND_MAX;
            B[i * N + j] = (float)rand() / RAND_MAX;
            C[i * N + j] = 0.0;
        }
    }
}

void multiply_matrices_IKJ() {
    #pragma omp parallel for num_threads(omp_get_max_threads())
    for (int i = 0; i < N; i++) {
        for (int k = 0; k < N; k++) {
            for (int j = 0; j < N; j++) {
                C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
        }
    }
}
```

Powyższy kod implementuje najbardziej optymalną wersję mnożenia macierzy z wykorzystaniem trzech pętli (***IKJ***), która minimalizuje zjawisko **false sharingu**[1]. Dla macierzy o rozmiarze Czas wykonywania wyniósł **0.398 sekundy** (AMD Ryzen 7 5800X 8 cores / 16 threads)

[1] **False sharing** jest zjawiskiem polegającym na unieważnieniu linii pamięci podręcznych procesorów. Sprowadza się to do konieczności aktualizowania danych na unieważnionej linii przez wszystkie wątki.

Program 0copy

```
checkCudaErrors(cudaSetDeviceFlags(cudaDeviceMapHost));
checkCudaErrors(cudaHostAlloc(&h_a, BYTES, cudaHostAllocMapped));
checkCudaErrors(cudaHostAlloc(&h_b, BYTES, cudaHostAllocMapped));
checkCudaErrors(cudaHostAlloc(&h_c, BYTES, cudaHostAllocMapped));
```

```
int WARP = THREADS; // 8 || 16 || 32
int BLOCKS = (int)ceil(SIZE / WARP); // sufit z 3072 / WARP

dim3 threads(WARP, WARP);
dim3 blocks(BLOCKS, BLOCKS);

std::cout << "Rozpoczynam przetwarzanie\n";
multiplyKernel <<<blocks, threads>>> (d_a, d_b, d_c, SIZE);
checkCudaErrors(cudaDeviceSynchronize());
std::cout << "Koncze...\n";
```

```
__global__ void multiplyKernel(const float* __restrict__ A,
const float* __restrict__ B,
float* __restrict__ C, const int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    C[row * N + col] = 0;
    for (int k = 0; k < N; k++) {
        C[row * N + col] += A[row * N + k] * B[k * N + col];
    }
}
```

cudaHostAlloc() przydziela rozmiar bajtów pamięci hosta, która jest dostępna dla urządzenia. Sterownik śledzi zakresy pamięci wirtualnej przydzielone za pomocą tej funkcji i automatycznie przyspiesza wywołania funkcji takich jak cudaMemcpy(). Ponieważ pamięć może być dostępna bezpośrednio przez urządzenie, może być odczytywana lub zapisywana ze znacznie większą przepustowością niż pamięć stronicowana uzyskana za pomocą funkcji takich jak malloc().

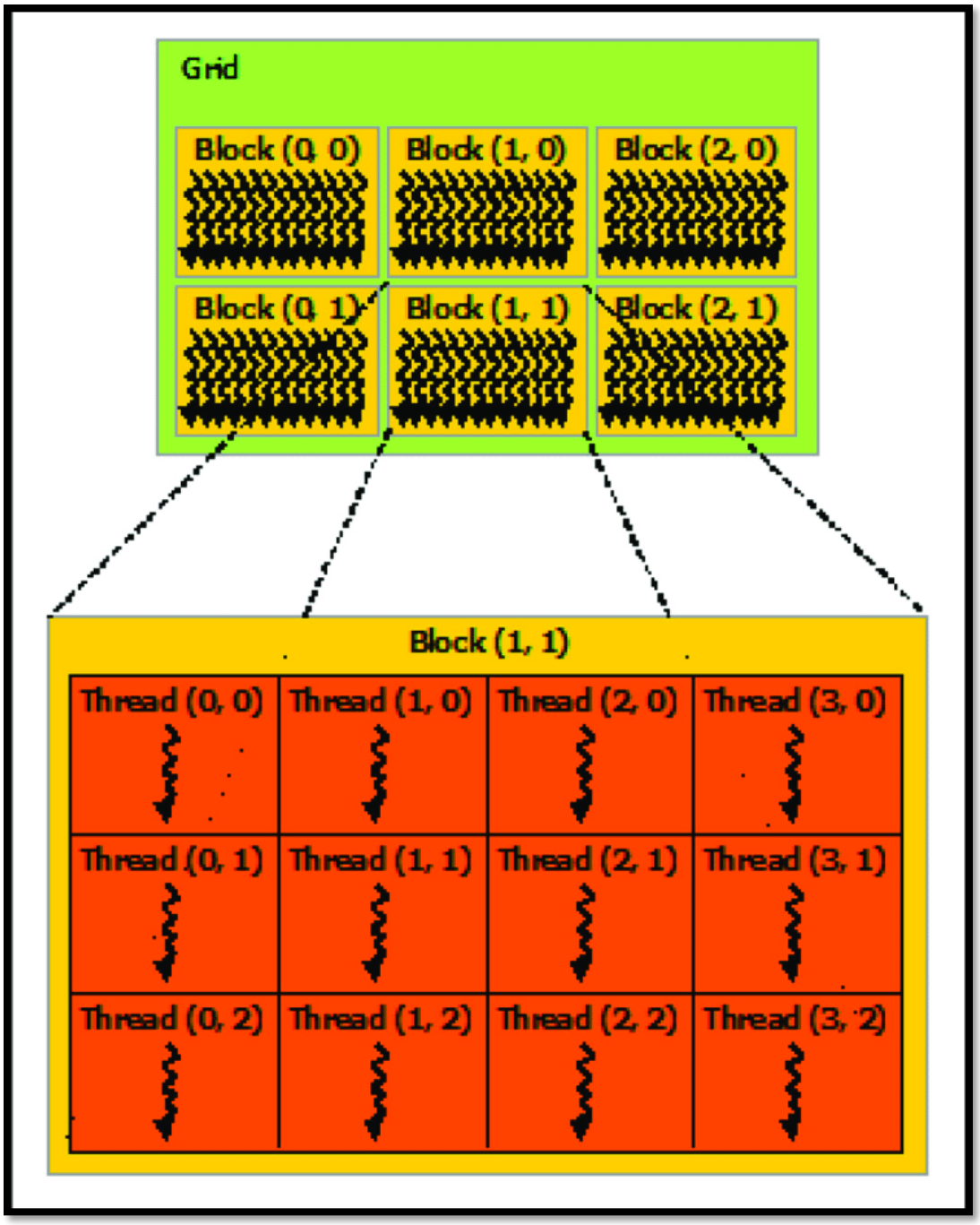
W funkcji **multiplyKernel** obliczany jest indeks wiersza oraz kolumny dla każdego wątku, a następnie wyznaczamy wartość macierzy C w oparciu o wartości macierzy A oraz B. W kodzie wykorzystujemy synchronizację po zakończeniu kernela, która sprawia, że dalszy kod wykona się dopiero w momencie, kiedy wszystkie wątki zakończą swoje działanie. Klauzula restrict została wprowadzona w celu optymalizacji kodu – mówimy kompilatorowi, że dane te są jedynie do odczytu.

Program shared memory

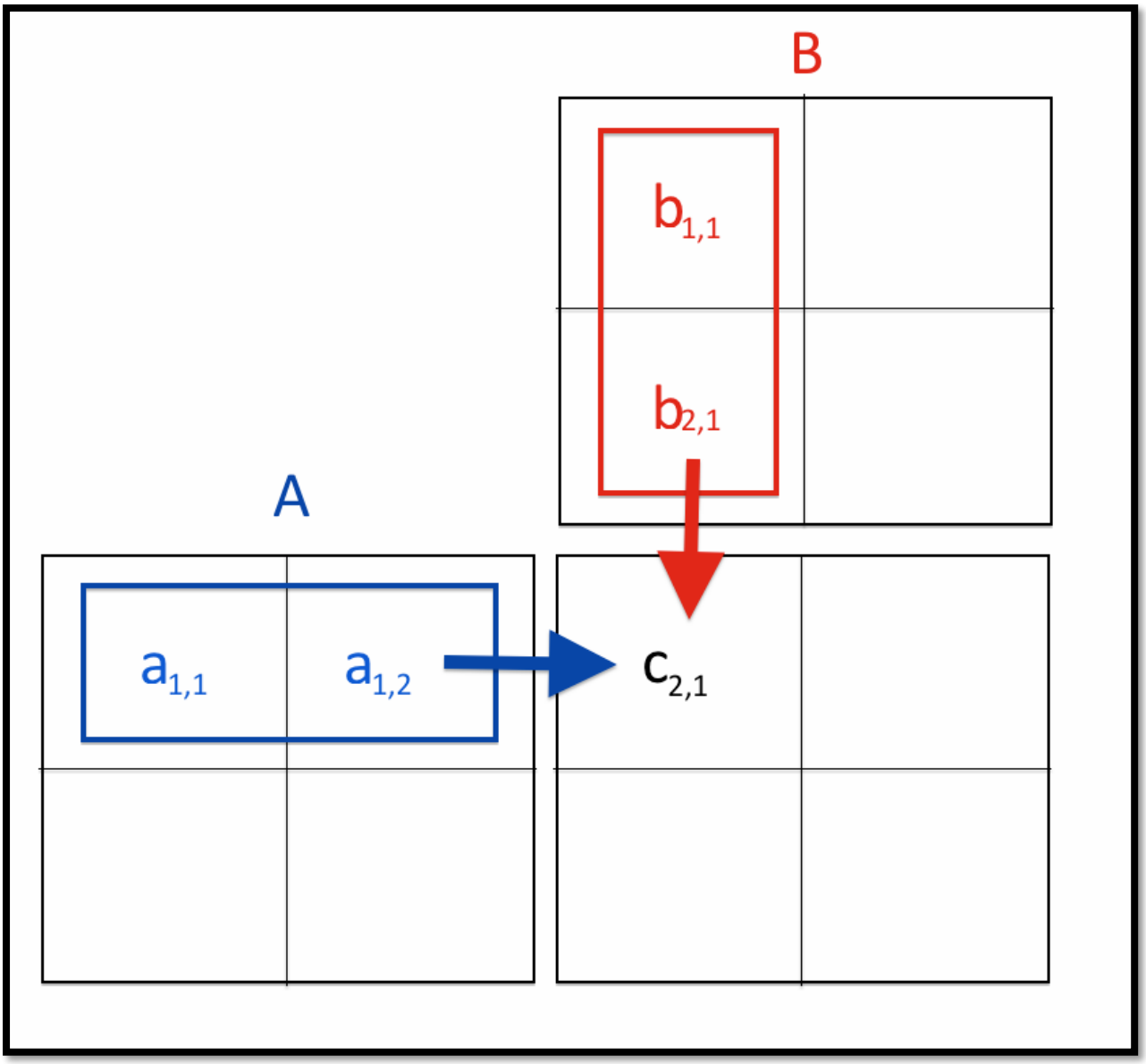
```
#define SHARED_MEMORY THREADS * THREADS * sizeof(float)
__global__ void multiplyKernel(const float* __restrict__ A,
const float* __restrict__ B,
float* __restrict__ C,
const int tile_size) {
    int row = blockIdx.y * tile_size + threadIdx.y;
    int col = blockIdx.x * tile_size + threadIdx.x;

    __shared__ float shared_A[SHARED_MEMORY];
    __shared__ float shared_B[SHARED_MEMORY];

    float sum = 0;
    for (int i = 0; i < (SIZE / tile_size); i++) {
        int tile = threadIdx.y * tile_size + threadIdx.x;
        shared_A[tile] = A[row * SIZE + (i * tile_size + threadIdx.x)];
        shared_B[tile] = B[(i * tile_size * SIZE + threadIdx.y * SIZE) + col];

        __syncthreads();

        for (int j = 0; j < tile_size; j++) {
            sum += shared_A[(threadIdx.y * tile_size) + j] * shared_B[(j * tile_size) + threadIdx.x];
        }
        __syncthreads();
    }
    C[row * SIZE + col] = sum;
}
```

Powyższy kod różni się jedynie deklaracją rozmiaru pamięci współdzielonej i zmienioną funkcją kernel, co jednak będzie miało znaczący wpływ jeśli chodzi o czas przetwarzania. Wersja ta korzysta z pamięci współdzielonej, która gwarantuje mniejszą liczbę pobrań z wolnej pamięci globalnej (deklarujemy pamięć współdzieloną za pomocą klauzuli shared). Pierwsza pętla kernelu ma za zadanie podział pojedynczego mnożenia komórki macierzy na mniejsze fragmenty w celu lepszego podziału pracy i przyspieszenia obliczeń. Synchronizujemy wątki i przechodzimy do następnej pętli, która mówi ile razy dany wątek ma pobrać dane i obliczać sumy. Ostatecznie przypisujemy wartość do komórki macierzy C.

3. Rysunki



Rysunek 1 Charakterystyczna budowa architektury karty graficznej. Grid składa się z bloków, a bloki zawierają w sobie wiele wątków.



Rysunek 2 Sposób pobierania danych do pamięci współdzielonej (shared memory)

4. Wzory

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

gdzie:
P – prędkość,
N – rozmiar,
T – czas uruchomienia jednokrotnego kernela.

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

gdzie:
a – przyspieszenie,
t_{GPU} – czas przetwarzania kodu na karcie graficznej,
t_{CPU} – czas przetwarzania kodu na procesorze.

5. Wyniki

Rozmiar tablicy dla wszystkich pomiarów wynosił 3072x3072

POMIAR	**ID**
GLOBAL [8X8]	1
GLOBAL [16X16]	2
GLOBAL [32X32]	3
SHARED [8X8]	4
SHARED [16X16]	5
SHARED [32X32]	6
CPU	7

id	**Duration [s]**	**Performance [flop/s]**	**Arithmetic intensity [flop/byte]**	**Compute (SM) throughput [%]**	**Memory throughput [%]**	**L1 hit rate [%]**	**L2 hit rate [%]**
1	12.08	4.8E+9	0.65	2.41	22.61	91.31	46.86
2	5.01	1.2E+10	1.57	5.82	23.91	90.60	67.91
3	4.49	1.3E+10	1.75	6.49	6.33	98.02	74.22
4	1.07	5.4E+10	7.00	31.20	8.21	44.92	0.37
5	0.522	1.11E+11	15.09	58.98	15.96	30.28	0.44
6	0.421	1.38E+11	18.76	67.87	18.68	0 (?)	0.55
7	**0.398**	-	-	-	-	-	-

id	Wielkość transmisji z PG – odczyt	Wielkość transmisji z PG – zapis	Wielkość transmisji z PW – odczyt	Wielkość transmisji z PW – zapis	Liczba konfliktów w dostępie do PW – bank conflicts
1	?	?	-	-	-
2	?	?	-	-	-
3	35.63 [GB]	41.93 [MB]	-	-	-
4	8.13 [GB]	444 [KB]	1.81 [GB]	226 [MB]	0 (?)
5	3.71 [GB]	827 [KB]	1.81 [GB]	113 [MB]	0 (?)
6	2.96 [GB]	630 [KB]	1.81 [GB]	56.62 [MB]	0 (?)
7	-	-	-	-	-

id	Wykonane instrukcje – executed instructions	Grid size	Block size	Liczba rejestrów na wątek	Rozmiar PW użytej przez blok wątków[b]	Occupancy theoretical/ achieved [%]	Ograniczenie na liczbę bloków [blok]
1	7.6E+10	147456	64	20	-	66.67/66.61	42/16/24/16
2	7.6E+10	36864	256	20	-	100/99.98	10/8/6/16
3	7.6E+10	9216	1024	20	-	66.67/66.66	2/8/1/16
4	8.8E+10	147456	64	26	2048	66.67/66.64	32/21/24/16
5	7.7E+10	36864	256	26	8192	100/99.78	8/7/6/16
6	7.1E+10	9216	1024	26	32768	66.67/66.67	2/1/1/16
7	-	-	-	-	-	-	-

id	N	Czas [s]	a	P
1	**3072**	**12.08**	**0.03**	**4.8E+9**
2	**3072**	**5.01**	**0.08**	**1.16E+10**
3	**3072**	**4.49**	**0.09**	**1.29E+10**
4	**3072**	**1.07**	**0.37**	**5.42E+10**
5	**3072**	**0.522**	**0.76**	**1.11E+11**
6	**3072**	**0.421**	**0.95**	**1.38E+11**

CPU	GPU
TDP	
105 W	320 W
LITOGRAFIA	
7 nm	8 nm
POWIERZCHNIA KRZEMU	
124 mm^2	628 mm^2

6. Wnioski

Dla wariantu o id: 1 (GLOBAL [8X8])

Jest to najgorszy wariant mnożenia macierzy, co potwierdzają wszystkie statystyki.

Dla wariantu o id: 4 (SHARED [8X8])

Wariant ten wykonał największą liczbę operacji i pod tym względem znacznie wyróżnia się na tle innych. Wynika to z faktu, iż rozmiar tablicy jest mniejszy i częściej musi się odwoływać do pamięci.

Dla wariantu o id: 6 (SHARED [32X32])

Najszybszym wariantem GPU jest wariant SHARED [32X32], który jest bardzo zbliżony do czasu przetwarzania CPU. Posiada on najwyższy wskaźnik CGMA (stosunek liczby operacji do dostępów do pamięci globalnej) oraz najwyższy wskaźnik Compute (SM) throughput (67.87), co może świadczyć o wysokiej synchronizacji wątków.

Scheduler może wydać jedną instrukcję na jeden cykl, ale w przypadku naszego kernela wydajemy je co 2,3 cyklu. W konsekwencji zasoby sprzętowe mogą zostać wykorzystane w sposób nieoptymalny. Maksymalnie nasz kernel wspiera 12 warpów na scheduler, lecz w praktyce przydzielanych jest średnio 8. Dodatkowo tylko 47% warpów kwalifikowało się na cykl. Te zakwalifikowane to podzbiór aktywnych warpów, które są gotowe do wykonywania nowych operacji. W związku z tym każdy warp bez kwalifikacji powoduje, że żadna instrukcja nie zostanie wydana przez co miejsce wydania pozostaje niewykorzystane. W celu zwiększenia kwalifikacji, powinniśmy unikać możliwej nierównowagi obciążenia wynikającej z różnych czasów trwania wykonania na każdego warpa.

Ogólnie rzecz biorąc kody **SHARED** wykonują się szybciej przez mniejszą liczbę odwołań do pamięci globalnej (DRAM) oraz mają lepszy współczynnik CGMA. Większa ilość wykonanych instrukcji wynika z dodatkowej pętli *for* w kodzie, która jest odpowiedzialna za dodawanie elementów do sumy częściowej jednego elementu badanej macierzy.

Znaki zapytania w tabelach spowodowane są błędami w przetwarzaniu Nsight Compute. Po wielokrotnych próbach naprawy niestety nie udało mi się rozwiązać tego problemu. Dodatkowo w trakcie korzystania z programu wielokrotnie pojawiał się „blue screen”.