Laboratorium z przetwarzania równoległego		
Sprawozdanie z projektu CUDA (wariant 2)		
Wersja sprawozdania	1.0 - podejście pierwsze	
Wymagany termin oddania	**15.06.2022**	
Rzeczywisty termin oddania	15.06.2022	
Wykonawcy		
Imię i nazwisko	nr indeksu	Grupa dziekańska
Adrian Madajewski	145406	L14
Kajetan Wencierski	145282	L14
e-mail kontaktowy:	adrian.madajewski@student.put.poznan.pl	

Opis karty graficznej:

Procesor graficzny	
Model	GeForce GTX 1060
Architektura	Pascal
Litografia	16 nm
Tranzystory	4.400 milion
Powierzchnia krzemu	200 mm^2
Magistrala	PCIe 3.0 x16
Pamięć	
Rozmiar pamięci	6 GB
Typ pamięci	GDDR5
Magistrala pamięci	192 bit
Bandwidth	192.2 GB/s
Render Config (SFU)	
Shading Units	1280
TMUs (texture mapping unit)	80
ROPs (render output unit)	48

Liczba SM (streaming multiprocessor)	10
L1 Cache	48 KB (per SM)
L2 Cache	1536 KB
Prędkości zegarów	
Base Clock	1506 MHz
Boost Clock	1709 MHz
Memory Clock	2002 MHz 8Gbps effective

Compute Capability 6.1	
Liczba ALU dla liczb stało liczbowych i pojedynczej precyzji	128
Liczba SFU dla funkcji pojedynczej precyzji	32
Liczba jednostek ROP	8
Liczba warp schedulerów	4

Informacje szczegółowe	
16-bit floating-point add, multiply, multiply-add	2
(single precision) 32-bit floating-point add, multiply, multiply-add	128
(double precision) 64-bit floating-point add, multiply, multiply-add	4

Ograniczenia posiadanego Compute Capability (6.1) przez kartę:

- brak rdzeni tensorowych
- brak zmiennej precyzji dla funkcji warp-matrix
- brak sprzętowego wsparcia dla kopiowania asynchronicznego
- brak sprzętowego-przyśpieszenia podziału bariery arrive/wait
- brak zarządzania rezydencją dla pamięci L2

Wyniki wywołania deviceQuery:

CUDA Driver Version / Runtime Version	11.7 / 11.7
CUDA Capability Major/Minor version number:	6.1

Total amount of global memory:	6144 MBytes (6442188800 bytes)
(010) Multiprocessors, (128) CUDA Cores/MP:	1280 CUDA Cores
GPU Max Clock rate:	1785 MHz (1.78 GHz)
Memory Clock rate:	4004 Mhz
Memory Bus Width:	192-bit
L2 Cache Size:	1572864 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:	98304 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor:	2048
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 / 1 / 0

Bank-conflicts:

Pamięć współdzielona, do której można uzyskać dostęp równolegle, jest podzielona na moduły (nazywane również bankami).Jeżeli w tym samym banku występują dwie lokacje pamięci (adresy), to dochodzi do konfliktu banków, podczas którego dostęp odbywa się szeregowo, tracąc zalety dostępu równoległego.

False sharing:

False sharing - zjawisko to polega na wielokrotnym, cyklicznym (wywołanym przez wielokrotne zapisy w różnych procesorach wartości do tej samej linii pamięci podręcznej procesora):

- unieważnianiu powielonych linii pamięci w pamięci podręcznej procesorów, które przestają być aktualne w wyniku zapisu danych do jednej z wielu kopii tej linii
- konieczności sprowadzenia do pamięci podręcznej procesora realizującego kod wątku (korzystającego z danych sąsiednich) aktualnej kopii danych z unieważnionej linii.

Deklarując tablice każdy z wątków będzie odwoływał się do innej komórki - widoczne spowolnienie. Ponieważ tablica zostanie zapisana w cache jednego z rdzeni - reszta wątków będzie musiała się z nim komunikować - przez co jest wolniej niż gdyby każdy wątek pracował na swoim cache.

Wyścig:

Zjawisko wyścigu występuje gdy wątki współbieżnie korzystają z zasobu dzielonego (np. zmiennej), przy czym przynajmniej jeden z nich zmienia stan tego zasobu.
Wyścig często jest przyczyną niedeterministycznego zachowania się programu i może prowadzić do trudnych do wykrycia błędów. Kod jest bezpieczny dla wątków wykonanych równolegle, gdy jego współbieżne wykonanie nie powoduje sytuacji wyścigu.

Wariant 1 - mnożenie macierzy openmp

```
void multiplyMatrixIKJ(
      const float* A,
      const float* B,
      float* C,
      const int threads)
{
#pragma omp parallel for num_threads(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];
            }
      }
   }
}
```

Wykonanie programu odbywa się najbardziej optymalną metodą z wykorzystaniem 3 pętli for w dostępach I, K, J w celach minimalizacji false-sharingu. Pamięc zaalokowana jest dynamicznie na stosie. Program wykonany został na 8 wątkach - maksymalna ilość dostępna dla mojego CPU (Intel Core I7-7700K). Wyniki prezentują się następująco:

Zazwyczaj pamięć komputera jest podzielona na różne typy, które mają różne charakterystyki wydajności (często nazywa się to hierarchią pamięci). Najszybsza pamięć znajduje się w rejestrach procesora, do których (zwykle) można uzyskać dostęp i odczytać je w jednym cyklu zegara. Jednak zazwyczaj jest tylko kilka takich rejestrów (zwykle nie więcej niż 1 KB). Z drugiej strony główna pamięć komputera jest ogromna (powiedzmy 8 GB), ale dostęp do niej jest znacznie wolniejszy. Aby poprawić wydajność, komputer jest zwykle fizycznie skonstruowany tak, aby miał kilka poziomów pamięci podręcznej między procesorem a pamięcią główną. Te pamięci podręczne są wolniejsze niż rejestry, ale znacznie szybsze niż pamięć główna, więc jeśli uzyskamy dostęp do pamięci, który wyszukuje coś w pamięci podręcznej, jest to zwykle znacznie szybsze niż w przypadku korzystania z pamięci głównej (zwykle między 5 do 25 razy szybciej). Podczas uzyskiwania dostępu do pamięci procesor najpierw sprawdza pamięć podręczną pod kątem tej wartości, zanim wróci do pamięci głównej, aby odczytać wartość. Jeśli konsekwentnie korzystamy z dostępu do wartości w pamięci podręcznej to uzyskamy znacznie lepszą wydajność niż w przypadku przeskakiwania pamięci w losowy sposób.

Większość programów jest napisana w taki sposób, że jeśli pojedynczy bajt z pamięci zostanie wczytany do pamięci, program później odczytuje również wiele różnych wartości z tego obszaru pamięci. W związku z tym te pamięci podręczne są zwykle zaprojektowane w taki sposób, że podczas odczytywania pojedynczej wartości z pamięci, blok pamięci (zwykle gdzieś między 1 KB a 1 MB) wartości wokół tej pojedynczej wartości jest również

pobieranych do pamięci podręcznej. W ten sposób, jeśli program odczytuje pobliskie wartości, są one już w pamięci podręcznej i nie musisz przechodzić do pamięci głównej.

W C/C++ tablice są przechowywane w kolejności wiersz-główny, co oznacza, że wszystkie wartości w jednym wierszu macierzy są przechowywane obok siebie. Tak więc w pamięci tablica wygląda jak pierwszy wiersz, potem drugi, potem trzeci itd.

Biorąc to pod uwagę, poniższy kod wykona się stosunkowo wolno.

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
                for (int k = 0; k < n; k++)
                      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Na powyższym listingu w każdej iteracji wartość k zmienia się wzrastając. Oznacza to, że podczas uruchamiania najbardziej wewnętrznej pętli, każda iteracja pętli może nie trafić w pamięć podręczną podczas ładowania wartości b[k][j]. Powodem tego jest to, że macierz jest przechowywana według wierszy (row-ordered), za każdym razem, gdy zwiększamy k, pomijamy cały wiersz macierzy i przeskakujemy znacznie dalej do pamięci, prawdopodobnie daleko poza wartości, które zostały zachowane w pamięci podręcznej . Jednak nie przeoczymy trafienia c[i][j] (ponieważ i oraz j są takie same), ani prawdopodobnie nie przegapimy a[i][k], ponieważ wartości są w wierszu głównym kolejności i jeśli wartość a[i][k] jest buforowana z poprzedniej iteracji, wartość a[i][k] odczytana w tej iteracji pochodzi z sąsiedniej lokalizacji pamięci. W związku z tym w każdej iteracji najbardziej wewnętrznej pętli prawdopodobnie wystąpi jeden brak trafienia do pamięci podręcznej.

Rozważając drugą wersję - użyta w programie:

```
for (int i = 0; i < n; i++)
    for (int k = 0; k < n; k++)
                for (int j = 0; j < n; j++)
                      c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Teraz, ponieważ zwiększamy j w każdej iteracji, wartość c[i][j] prawdopodobnie znajduje się w pamięci podręcznej, ponieważ wartość c[i][j] z poprzedniej iteracji prawdopodobnie jest również buforowana i gotowa do odczytania z pamięci. Podobnie, b[k][j] jest prawdopodobnie buforowane, a ponieważ i i k się nie zmieniają, są szanse, że a[i][k] również jest buforowane. Oznacza to, że w każdej iteracji wewnętrznej pętli prawdopodobnie nie wystąpią żadne chybienia do pamięci podręcznej.

Podsumowując, oznacza to, że jest mało prawdopodobne, aby druga wersja kodu miała błędy w pamięci podręcznej przy każdej iteracji pętli, podczas gdy pierwsza wersja prawie na pewno to zrobi. W konsekwencji druga pętla będzie prawdopodobnie szybsza niż pierwsza - i faktycznie tak jest.

Dodatkowo można jeszcze bardziej zoptymalizować wydajność tej pętli, używając bardziej złożonych pętli. Na przykład technika zwana blokowaniem może być wykorzystana do znacznego zwiększenia wydajności poprzez podzielenie tablicy na podregiony, które mogą być dłużej przechowywane w pamięci podręcznej, a następnie użycie wielu operacji na tych blokach w celu obliczenia ogólnego wyniku.

W wariancie pierwszym zastosowano podział pracy między 8 wątków korzystając z OpenMP. Rodzaj schedulowania - statyczny (domyślnie). Pętla zewnętrzna wykona się z wykorzystaniem współdzielenia. Brak wyścigu. False sharing nie występuję - jak opisano powyżej.

Wielkość macierzy (N)	Czas wykonania (wall clock) [ms]
1024	984.309
2048	6829.84
3072	21519.90

Wariant 2 - CUDA zero-copy (bez współdzielenia)

Alokacja pamięci za pomocą Zero-Copy przebiega analogicznie w przypadkach 2 wariantów programów realizowanych na karcie graficznej.

Alokacja pamięci z użyciem Zero-Copy:

```
checkCudaErrors(cudaSetDeviceFlags(cudaDeviceMapHost));
checkCudaErrors(cudaHostAlloc(&A_host, BYTES, cudaHostAllocMapped));
checkCudaErrors(cudaHostGetDevicePointer(&A_device, A_host, 0));
...
cudaFreeHost(A_host);
```

Opis Zero Copy:

Zunifikowane adresowanie wirtualne (UVA) w CUDA zapewnia pojedynczą przestrzeń adresową pamięci wirtualnej dla procesora i pamięci GPU oraz umożliwia dostęp do wskaźników z kodu GPU. UVA włącza pamięć Zero-Copy, która jest przypiętą pamięcią procesora dostępną przez kod GPU bezpośrednio przez PCIe, bez memcpy. Zero-Copy zapewnia pewną wygodę pamięci ujednoliconej - unified memory (opisanej w dalszej części, która wykonuje niejawne kopiowanie pamięci). Oba mechanizmy zwalniają programistów z obowiązku jawnego zarządzania transferem pamięci. Zero-copy jest dobre dla kodu jądra uzyskującego dostęp do danych hosta rzadko lub nie wielokrotnie. W przeciwieństwie do pamięci Unified Memory, procesor graficzny ma zawsze dostęp do danych o niskiej przepustowości i dużym opóźnieniu PCIe. Zunifikowana pamięć zapewnia szybszy dostęp do pamięci, jeśli te same dane są odczytywane wielokrotnie. Zamiast używać malloc do przydzielania pamięci hosta, wywołujemy cudaHostAlloc, aby utworzyć przypiętą pamięć hosta z blokadą stron. Pamięć hosta z blokadą stron ma kilka zalet:

- Asynchroniczne współbieżne wykonywanie jąder i transfer pamięci za pomocą przesyłania strumieniowego.
- Dzięki pamięci mapowanej eliminujemy konieczność przydzielania bloku w pamięci urządzenia.
- Większa przepustowość między pamięcią hosta a pamięcią urządzenia, w szczególności w przypadku pamięci z funkcją łączenia zapisu.

Pamięć hosta z blokadą strony jest zarządzana przez: cudaHostAlloc i cudaFreeHost, które przydzielają i zwalniają pamięć hosta z blokadą stronicowania.

Jeśli urządzenie obsługuje pamięć mapowaną (mapped memory) to możemy jej użyć aktywując flagę cudaDeviceMapHost w cudaSetDeviceFlags(), a następnie przydzielając pamięć z użyciem funkcji cudaHostAlloc z flaga cudaHostAllocMaped. Blok pamięci hosta z blokada strony można zmapować do przestrzeni adresowej urządzenia. Blok pamięci po zmapowaniu ma wtedy 2 adresy - jeden dostępny z pamięci hosta, drugi dostępny z pamięci urządzenia - w celach synchronizacji tych 2 adresów ze sobą wywołujemy funkcję cudaHostGetDevicePointer(). Pamięć mapowana umożliwia jądru bezpośredni dostęp do pamięci hosta. Transfery danych są domyślnie wykonywane zgodnie z potrzebami jądra. W związku z tym transfer danych może być również równoczesny z wykonywaniem jądra. Ponieważ mapowana pamięć jest współdzielona między hostem a urządzeniem, aplikacja musi zsynchronizować dostęp do pamięci za pomocą strumieni lub zdarzeń między hostem a urządzeniem - funkcja cudaDeviceSynchronize().

Wariant 2 - CUDA zero-copy (bez współdzielenia)

Opis programu:

Wywołanie kernela:

```
multiplyKernel << <blocks, threads >> > (A_device, B_device, C_device, N);
```

Kod kernela:

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

    for (int i = 0; i < size; i++)
    {
       C[index(row, col, size)] += A[index(row, i, size)]
```

```
                                * B[index(i, col, size)];
    }
}
```

Opis __restrict__:

Klauzula __restrict__ ma takie samo znaczenie jak słowo kluczowe restrict w C99. Jest to podpowiedź dla kompilatora o zapewnieniu jedynie dostępu do danych w celach ich odczytu - brak modyfikacji tych danych. Podpowiedź ta została użyta w celach optymalizacyjnych. Dla compute capability 3.5 albo wyższych karty graficzne mają specjalną linie pamięci podręcznej - read only cache - pamięć tylko do odczytu, która jest niezależna od innych. Jeśli używamy w deklaracji parametru funkcji jednocześnie const __restrict__ float *A, to jest to dość silna podpowiedź dla kompilatora aby korzystał on właśnie z pamięci read-only. Użycie klauzuli __restrict__ nie gwarantuje takiego zachowania. Kompilator sam może zdecydować o użyciu pamięci read-only, nawet bez wprowadzania klauzul.

Opis kernela:

W pierwszym kroku następuje uniwersalne obliczenie bieżących współrzędnych macierzy - dla każdego wątku w danym bloku wątków. Deklarowana jest zmienna przechowująca wartość sumy częściowej (sum). Następnie pętla for po całej macierzy będzie mnożyć ze sobą odpowiednie elementy z wiersza macierzy A oraz kolumny macierzy B aktualizujące wartość zmiennej sum. Po wykonaniu pętli wartość zostanie przypisana do globalnej macierzy wynikowej C.

Funkcja index została zaimplementowania w celach czytelniejszego odnoszenia się do tablicy jednowymiarowej przy użyciu 2 wymiarowego indeksu (row, col):

```
__device__ inline int index(
    const int row,
    const int column,
    const int width)
{
    return row * width + column;
}
```

Ogólne wywołanie kernela z parametrami:

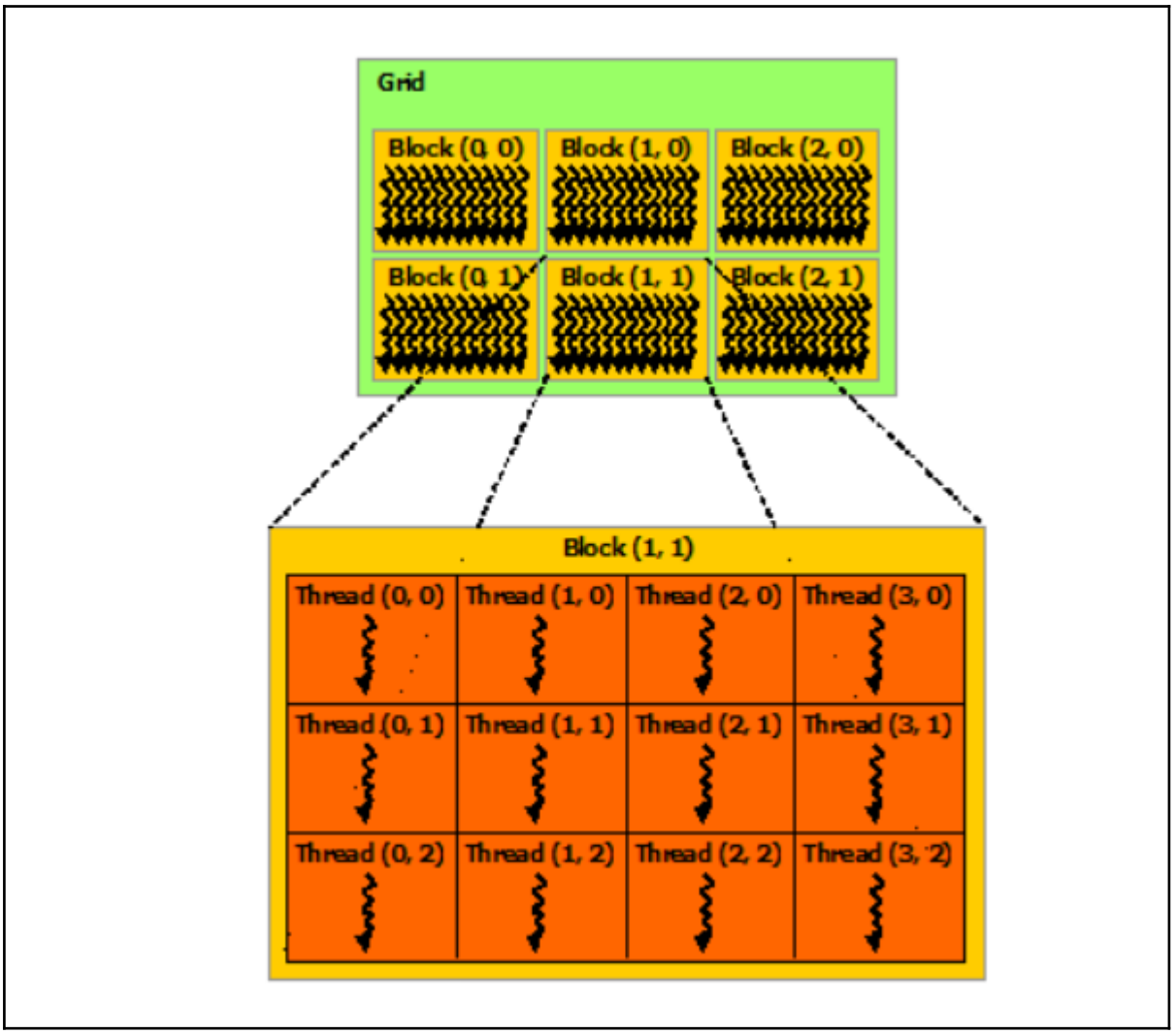
```
constexpr int THREADS = 32
...
int BLOCKS = (int)ceil(N / THREADS); // ceil is unnecessary as N divides
threads evenly
dim3 threads(THREADS, THREADS);
dim3 blocks(BLOCKS, BLOCKS);
```

Przykładowo dla N = 1024 (wielkość macierzy) i parametru THREADS = 32 otrzymamy:

Wymiary gridu: 32x32, 1024 bloki wątków
Wymiary bloku 32x32. 1024 wątków per blok

Istnieje limit dla karty graficznej na ilość wątków per blok, ponieważ wszystkie bloki powinny znaleźć się na tym samym rdzeniu (core) i muszą dzielić limitowany dostęp do wspólnej pamięci na rdzeniu. Dla mojej karty graficznej limit ten to 1024 wątki na jeden blok wątków.

Natomiast sam kernel może zostać wykonany przez wiele takim samych bloków wątków, więc łącznie liczba wątków jest równa liczbie wątków per blok razy liczba bloków.

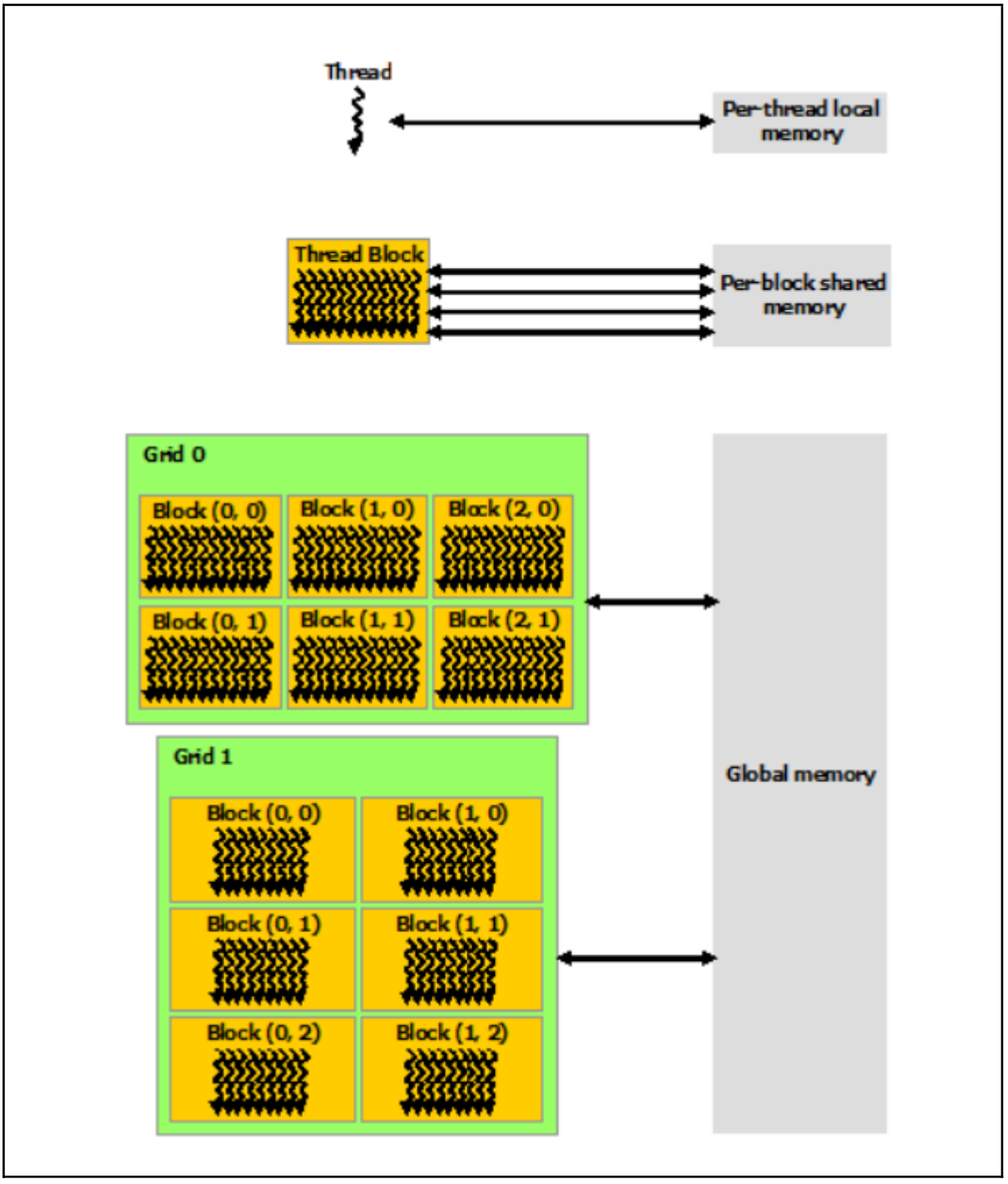


Rysunek 1.1 - rozmieszczenie wątków w karcie graficznej

Limiti posiadanej karty graficznej	
Maximum number of threads per multiprocessor	2048

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)

Odwołania do pamięci przy pomocy CUDA:



Rysunek 1.2 - odwołania do pamięci karty graficznej

Pamięć lokalna jest dostępna z poziomu pojedynczego wątku. Pamięć współdzielona jest dostępna z poziomu bloku wątków. Natomiast pamięć globalna (device memory) jest dostępna z poziomu każdego gridu.

Wariant 3 - CUDA zero-copy (z użyciem pamięci współdzielonej)

Optymalizacja z użyciem pamięci współdzielonej __shared__.

Opis pamięci współdzielonej:

Zastosowanie pamięci współdzielonej to jedna z metod optymalizacji przez zmniejszenie ilości pobierania danych prosto z DRAM, które wykonują się wolno - w tym celu zapisujemy pobrane dane w pamięci współdzielonej - dostępnej dla wielu wątków jednocześnie. Pamięć współdzielona karty graficznej jest user-managed - programista sprowadza dane do pamięci podręcznej, która tam zostaje. W naszym problemie mnożenia macierzy NxN każdy wątek oblicza jeden element macierzy wynikowej C. Optymalizacja z użyciem pamięci współdzielonej polega na dostępie do macierzy A oraz B - w wersji bez użycia pamięci współdzielonej potrzebowaliśmy całej macierzy aby wyliczyć wynik, natomiast w tej wersji będziemy odwoływać się do pod-macierzy - sprowadzonej wcześniej do pamięci podręcznej. W każdej iteracji pętli sprowadzamy do pamięci shared_A - element z wiersza macierzy A dostępny pod adresem access tile. Natomiast do pamięci shared_B będziemy sprowadzać wartości kolumny z macierzy B.

Wywołanie Kernela:

Analogicznie jak w wariancie bez współdzielenia pamięci. Dodatkowo w tym podejściu pamięć podręczna jest deklarowana przy pomocy zmiennej BLOCK_SIZE = 32.

```
multiplyKernel <<<blocks, threads >>> (A_device, B_device, C_device, N);
```

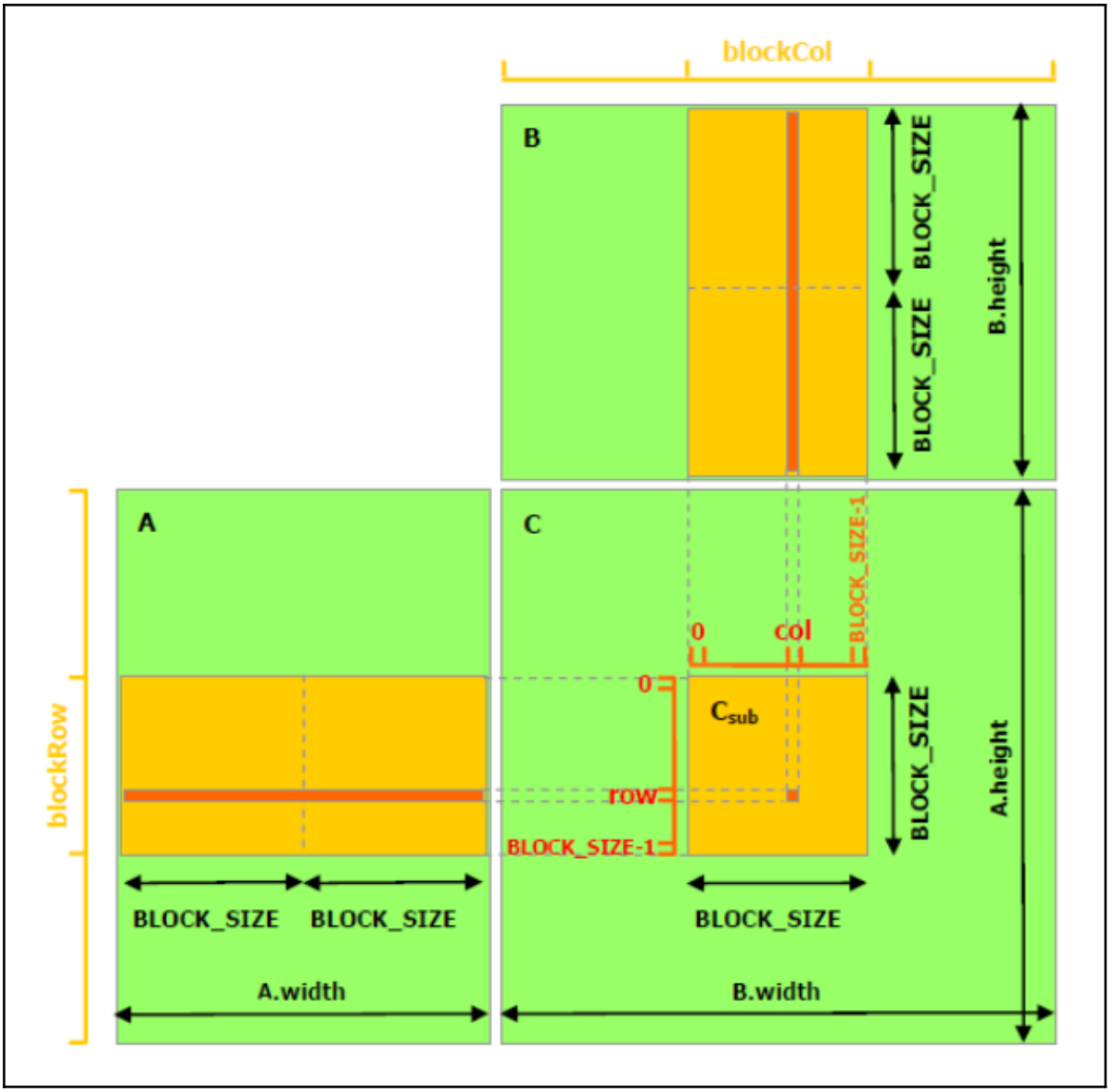
Kod Kernela:

```
__global__ void multiplyKernel(
    const float* __restrict__ A,
    const float* __restrict__ B,
    float* __restrict__ C,
    const int size)
{
    const int tx = threadIdx.x;
    const int ty = threadIdx.y;
    const int bx = blockIdx.x;
    const int by = blockIdx.y;

    const int row = by * BLOCK_SIZE + ty;
```

```
    const int col = bx * BLOCK_SIZE + tx;

    __shared__ float shared_A[BLOCK_SIZE * BLOCK_SIZE];
    __shared__ float shared_B[BLOCK_SIZE * BLOCK_SIZE];

    // No alliasing
    float sum = 0;
    for (int i = 0; i < (size / BLOCK_SIZE); i++)
    {
       // Load elements from tile to shared memory
       shared_A[index(ty, tx, BLOCK_SIZE)] =
             A[index(row, i * BLOCK_SIZE + tx, size)];
       shared_B[index(ty, tx, BLOCK_SIZE)] =
             B[index(i * BLOCK_SIZE + ty, col, size)];

       __syncthreads();

       // Matrix multiply
       #pragma unroll
       for (int j = 0; j < BLOCK_SIZE; j++)
       {
             sum += shared_A[index(ty, j, BLOCK_SIZE)]
                   * shared_B[index(j, tx, BLOCK_SIZE)];
       }

       __syncthreads();
    }

    // Store result in global matrix
    C[row * size + col] = sum;
}
```

Pamięć współdzielona jest dostępna dla wątków wewnątrz konkretnego bloku wątków. Każdy wątek z bloku wątków wczytuje jeden element do pamięci współdzielonej. Lokalizacja elementu w pamięci współdzielonej jest zależna od pozycji wątku w bloku. Np. dla wątku [0, 0] ładujemy do pamięci współdzielonej wartości z A[0 * BLOCK_SIZE + 0] oraz B[0 * BLOCK_SIZE + 0].

Podejście to polega na podzieleniu macierzy na mniejsze fragmenty, podmacierze kwadratowe o wielkości BLOCK_SIZE. Podział ten dokonujemy według rysunku poniżej.



Rysunek 1.3 - schemat podziału problemu mnożenia macierzy na podmacierze kwadratowe o szerokości BLOCK_SIZE.

Indeksowanie macierzy przebiega w sposób następujący - przy pomocy funkcji index() jak wyżej.

Dla macierzy A:

- row - odwołujemy się do globalnego wiersza (niezależnie od pętli, zależne od wątków).
- i * BLOCK_SIZE - indeksujemy nowy zestaw kolumn dla każdej iteracji
- tx - indeksowanie elementu kolumny w tym zestawie (obecny wątek przetwarza jedną komórkę)

Dla macierzy B:

- i * BLOCK_SIZE - indeksujemy nowy zestaw wierszy dla każdej iteracji

- ty - która wartość z analizowanego zestawu nas interesuje obecnie - dla tego konkretnego wątku
- col - indeksowanie globalnej kolumny macierzy (niezależne od pętli, zależne od wątków).

Zapis do tablicy współdzielonej odbywa się przez indeksowanie tablicy wartościami tx oraz ty - zatem odwoływać się będziemy do niej przy pomocy obecnego wątku - w tym który obecnie przetwarza macierz.

Klauzula __syncthreads():

Wywołanie __syncthreads() jest potrzebne aby najpierw tablica współdzielona została załadowana wartościami - ponieważ pamięć współdzielona jest współdzielona w bloku wątków. Obecny blok musi wczytać wszystkie swoje wartości zanim przejdziemy do dalszego przetwarzania. Wywołanie __syncthreads() działa jako bariera przetwarzania dla wątków w danym bloku. __syncthreads() zabezpiecza nas program przed wyścigiem - czyli korzystaniem z nieaktualnej zmiennej (ponieważ bez tego inny wątek mógłby już nadpisać naszą pamięć współdzieloną). Przykładowo:

Thread 0: ----------> XXXXXXXXXXXXXXXXX
Thread 1: ---------->XXXXXXXXXXXXXXX
Thread 2: ---------->XXXXXX
Thread 3: ----------> syncthreads() dalsze przetwarzanie
….
Thread n-1: ---------->XXXXXXXXXXXX

X oznacza, że grupa wątków będzie czekać aż każdy wykona synchronizację (bariera). Ostatnie wywołanie __syncthreads() przez wątek 3 wznowi dalsze przetwarzanie dla każdego wątku w grupie.

Opis #pragma unroll:

Dyrektywa dla kompilatora który zamienia kod wynikowy pętli for na pojedyncze odwołania do ich komórek np. dla for(int i = 0; i < 5; i++) a[i] = i; otrzymamy:

a[0] = 0;
a[1] = 1;
a[2] = 2;
a[3] = 3;
a[4] = 4

Jako jawnie wygenerowany kod wynikowy - niezawierający pętli ale pojedyncze odwołania do komórek. Niekiedy takie podejście jest szybsze ze względu na brak porównania przy ewaluacji warunku i < 5 oraz inkrementacji zmiennej i. Zapewnia lepsze wykorzystanie ILP. (Instruction Level Parallelism).

Wykonanie operacji mnożenia:

Gdy dane są przygotowane do przetwarzania następuję ich wykorzystanie w kolejnej pętli for - w tej pętli przetwarzamy każdy element obecnie badanego bloku pod macierzy kwadratowej (wycinek macierzy), w której to odwołania do pamięci współdzielonej są szybsze niż bez jej użycia (odwołania bezpośrednio do DRAM). Wykonują się odwołania do pamięci współdzielonej w postaci:

shared_A[index(ty, j, BLOCK_SIZE)], gdzie:
- ty - id.y obecnego wątku
- j - obecna kolumna

Oznacza to przejście po każdym elemencie z wiersza (ty jest stałe, j zmienne)

shared_B[index(j, tx, BLOCK_SIZE)], gdzie:
- j - obecny wiersz
- tx - obecna kolumna (niezależna od pętli)

Oznacza przetwarzanie po każdym elemencie z kolumny macierzy B.

Przed ostatnim krokiem jest ponowna synchronizacja przy pomocy __syncthreads(). Dane dostępne w zmiennej sum muszą być gotowe do zapisu stąd też dla wywołania kernela w jednym wątku musimy poczekać aż wartość będzie gotowana do zapisana w globalnej tablicy C.

Zapis przebiega przez odwołanie się za pomocą obecnie realizowanego wątku (row, col) ponieważ wartości row i col są unikalne dla każdego wątku - każdy wątek bada inną komórkę macierzy.

Podejście z wykorzystaniem pamięci współdzielonej gwarantuje:
- zysk poprzez mniejszą liczbę pobrań z pamięci globalnej (DRAM), wielokrotnie wykorzystanie danych dostępnych w pamięci podręcznej.
- stratę - ponieważ ze względu na obliczenia konkretnego bloku wątków, wątki te nie mogą być realizowane jednocześnie z pobieraniem danych dla tego bloku - stąd synchronizacja.
- Im większy jest blok tym większa jest krotność wykorzystania raz pobranych danych do pamięci współdzielonej danych - CGMA (Compute to Global Memory Access) wzrasta.
- Im większy blok tym niższy stopień zrównoleglenia pobrań danych i obliczeń w ramach jednego SM (Streaming Multiprocessor), ale różne bloki mogą realizować różne etapy obliczeń.

Wyniki:

Porównanie parametrów GPU i CPU	CPU	GPU
	Intel Core I7 7700k	NVIDIA GTX 1060
Parametr energetyczny TDP	91 W	120 W
Litografia	14 nm	16 nm

Powierzchnia krzemu	149 mm^2	200 mm^2

Prędkość obliczeń uwzględniającą liczbę operacji zmiennoprzecinkowych wg złożoności algorytmu mnożenia macierzy o wielkości NxN i czasu T przetwarzania dla danej konfiguracji, wariantu, kodu i instancji problemu liczymy ze wzoru:

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

Przyspieszenie w stosunku do najlepszego przetwarzania na procesorze przy użyciu środowiska OpenMP, dla parametrów w kodzie:

THREADS = 32, (liczba wątków na blok)
BLOCK_SIZE = 32, (podział macierzy na pod macierze)

N	CPU (OpenMP) [ms]	GPU no-shared [ms]	Przyspieszenie (GPU / CPU)	P
1024	984.309	669.598	1.47	3207.12
2048	6829.84	4525.34	1.51	3796.37
3072	21519.90	15122.5	1.18	3834.15

N	CPU (OpenMP) [ms]	GPU shared [ms]	Przyspieszenie (GPU / CPU)	P
1024	984.309	226.37	4.33	9486.61
2048	6829.84	1561.3	4.37	11003.57
3072	21519.90	4876.03	4.41	11891.24

Analiza wyników przy pomocy narzędzia Nsight Compute dla programów GPU.

UWAGA! Ze względu na architekturę karty graficznej musiałem skorzystać ze starszej wersji Nsight Compute 2019.5.1. Wersja 2022 nie wspiera architektury Pascal mojej karty graficznej.

Wielkość macierzy N = 2048	CPU	[GPU] Global 8x8	[GPU] Global 16x16	[GPU] Global 32x32	[GPU] Shared 8x8	[GPU] Shared 16x16	[GPU] Shared 32x32
Duration [s]	6.82	8.41	4.38	4.02	1.74	1.70	1.75
Performance [flops/s]	X	4.6E+9	1.3E+10	1.4E+10	5.4E+10	11.12E +10	11.45E +10

CGMA [flop/byte]	X	0.65	1.35	1.75	7.00	14.28	16.54
Compute throughput SM [%]	X	2.43	6.32	7.02	72.65	74.65	72.28
Memory throughput [%]	X	0.41	0.85	0.78	5.15	1.71	4.22
L1 hit rate [%]	X	83.10	84.21	90.32	38.21	24.2	0
L2 hit rate [%]	X	56.47	45.21	45.30	4.49	0.49	0.71
PG (odczyt) [GB]	X	13.45	6.54 GB	7.05 GB	2.81 GB	2.70 GB	2.83 GB
PG (zapis) [GB]	X	19.23 MB	40.25 MB	52.66 MB	25.11 MB	650.87 KB	655.88 KB
PW (odczyt) [GB]	X	X	X	X	1.34 G REQ	805.31 M REQ	536.87 M REQ
PW (zapis) [GB]	X	X	X	X	67.11 M REQ	33.55 M REQ	16.78 M REQ
PW (bank conflicts)	X	X	X	X	855638 016	285212 672	0
Instructions	X	263201 75104	2632017 5104	2632017 5104	339586 58048	339586 58048	339586 58048
Grid_size	X	65536	16384	4092	65536	16384	4092
Block_size	X	64	256	1024	64	256	1024
Registers per thread	X	28	28	28	32	32	32
PW size (static shared memory per block) [Kbyte/block]	X	X	X	X	8.19	8.19	8.19
Occupancy theoretical/achieved [%]	X	100 / 99.82	100 / 99.68	100 / 98.41	37.50 / 37.48	100 / 99.81	100 / 99.81
Block limit register	X	32	8	2	32	8	2

Block Limit Shared Mem	X	32	32	32	12	12	12
Block limit Warps	X	32	8	2	32	8	2
Block Limit SM	X	32	32	32	32	32	32

Zajętość SM - SM throughput oznacza, że wątki tego samego bloku realizowane są na tym samym SM, czyli współpracują ze sobą przez pamięć współdzieloną oraz mogą się efektywnie synchronizować.

Wnioski:

Oprócz tego, że wersja programu Nsight Compute 2019 nie podaje wszystkich informacji jak wersja najnowsza i występują liczne błędy w aplikacji (częste blue screeny, błędy uruchomienia, o których naprawie mało znaleźliśmy w internecie) to udało się nam się wysnuć następujące wnioski:

Dla wersji GPU 8x8 no-shared:

Każdy scheduler jest w stanie wydać dwie instrukcje na cykl, ale w przypadku tego jądra każdy program planujący wydaje instrukcję tylko co 20,1 cykli. Może to pozostawić niewykorzystane zasoby sprzętowe i może prowadzić do mniej optymalnej wydajności. Spośród maksymalnie 16 warpów per scheduler jądro alokuje średnio 15,98 aktywnych warpów, (średnio 0,05 warp per cykl). Dostępne warpy to podzbiór aktywnych warpów, które są gotowe do wydania następnej instrukcji.Każdy cykl bez kwalifikującego się warpa nie powoduje żadnej wydanej instrukcji, a slot wydania pozostaje niewykorzystany.

Jądro wykazuje niską przepustowość obliczeniową i wykorzystanie przepustowości pamięci w stosunku do szczytowej wydajności mojej karty graficznej (według spreedsheetu Occupancy Calculator oraz analizy Nsight Compute). Osiągnięta przepustowość obliczeniowa i/lub przepustowość pamięci poniżej 60,0%.

Wersja 8x8 jest najgorszą możliwą implementacją przetwarzania mnożenia macierzy na mojej karcie graficznej - co potwierdzają wyniki Nsight Compute oraz globalny czas wykonywania programu.

Wersje 16x16 oraz 32x32 bez wykorzystania pamięci współdzielonej powodują lepsze przetwarzanie niż wersja 8x8 - więcej odwoływania do pamięci podręcznej L1/L2 karty graficznej. Wykonywanie programu jest szybsze.

Dla wariantów z pamięcią współdzieloną:

- Obliczenia są intensywniej wykorzystywane niż pamięć

- Programy wykonują się szybciej poprzez mniejszą liczbę odwołań do pamięci globalnej (device memory - DRAM)
- cechują się najwyższym współczynnikiem CGMA - Arithmetic Intensity - jest to liczba niezbędnych operacji na danych podzielona przez liczbę dostępów do danych w pamięci - z tego powodu ponieważ dane są dostępne *blisko* w pamięci. CGMA dla wariantów współdzielonych jest najwyższe (lepsze niż cache L1 oraz L2 w wariantach z pamięcią DRAM).
- większa liczba wykonywanych instrukcji jako konsekwencja rozszerzenia kodu kernela o dodatkową pętle for wykonującą dodawania elementów do sumy częściowej jednego elementu badanej macierzy.
- Najlepszym doborem liczby bloków okazał się dobór o wielkości bloku = 32x32.
- Pomimo narzucenia synchronizacji przez polecenie __syncthreads() kody współdzieloną wykonują zadaną pracę efektywniej - ze względu na mniejsze odwołania do pamięci globalnej i lepsze trafienia w pamięć współdzieloną w obrębie bloku wątków. Synchronizacja bloku nie jest tak czasochłonna w realizacji przetwarzania, poniewaz inne bloki dalej mogą pracować na pamięci współdzielonej i wykonywać obliczenia niezależnie.

W porównaniu do wersji CPU warianty na karcie graficznej wykonują się szybciej dla wersji bez współdzielenia pamięci średnio o 1.5 raza, natomiast dla wersji współdzielonej średnio aż 4 razy szybciej.