STUDIA INFORMATICA 2010
Volume 31 Number 4A (92)

Jacek WIDUCH
Politechnika Śląska, Instytut Informatyki

ZARZĄDZANIE PAMIĘCIĄ I BLOKAMI WĄTKÓW W OBLICZENIACH RÓWNOLEGŁYCH Z UŻYCIEM ARCHITEKTURY CUDA

Streszczenie. Dzięki upowszechnieniu się procesorów wielordzeniowych przetwarzanie danych za pomocą obliczeń równoległych staje się coraz bardziej dostępne dla szerokiego grona użytkowników. Przykładem jest opracowana przez firmę NVIDIA architektura CUDA, będąca architekturą wielordzeniowych procesorów graficznych. Procesor graficzny może być traktowany jako procesor SIMD z pamięcią wspólną. Na przykładzie operacji mnożenia macierzy zbadano wpływ zarządzania pamięcią i blokami wątków na czas obliczeń z użyciem architektury CUDA.

Słowa kluczowe: procesor graficzny, procesor wielordzeniowy, pamięć wspólna, wątek, przetwarzanie wielowątkowe, synchronizacja wątków, obliczenia równoległe, mnożenie macierzy

A MEMORY AND BLOCKS OF THREADS MANAGEMENT IN PARALLEL COMPUTATION USING CUDA ARCHITECTURE

Summary. With the propagation of a multi-core processors a parallel data processing becomes more accessible to a wide range of users. An example is CUDA architecture developed by NVIDIA, which is a multi-core GPU architecture. The GPU can be treated as a SIMD processor with shared memory. The influence of memory management and blocks of threads management on time of computation using CUDA architecture was researched on the basis of matrix multiplication.

Keywords: graphics processor, multi-core processor, shared memory, thread, multithreaded processing, threads synchronization, parallel computation, matrix multiplication

1. Wstęp

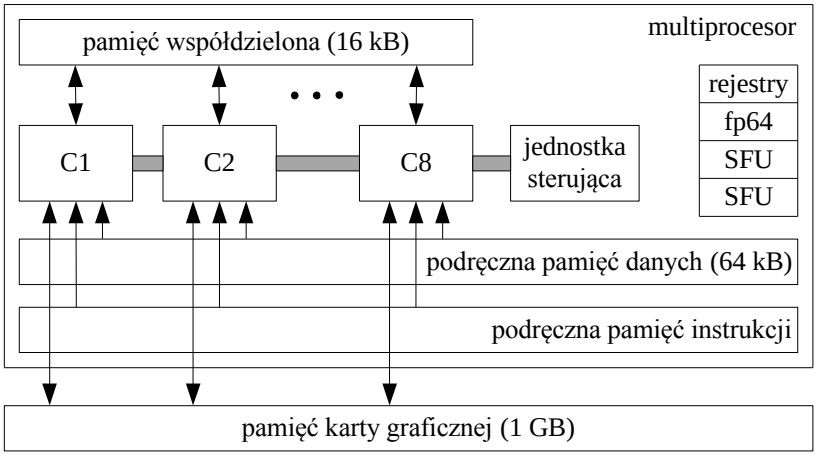
Wykonanie obliczeń inżynierskich w wielu przypadkach wymaga znacznej mocy obliczeniowej komputerów. Przeprowadzenie tego typu obliczeń w sposób efektywny jest możliwe dzięki zastosowaniu systemów wspomagających obliczenia równoległe. W 1966 roku została zaproponowana przez Michaela J. Flynna klasyfikacja komputerów, której podstawą jest liczba strumieni rozkazów realizowanych przez komputer i liczba strumieni przetwarzanych danych [4, 6, 7, 8, 9]. Zgodnie z klasyfikacją komputery dzielą się na 4 grupy: SISD (*single instruction stream single data stream*), SIMD (*single instruction stream multiple data stream*), MISD (*multiple instruction stream single data stream*) oraz MIMD (*multiple instruction stream multiple data stream*). Ze względu na dostęp do pamięci systemy można podzielić na systemy z pamięcią wspólną lub z pamięcią rozproszoną. W pierwszym przypadku obliczenia prowadzone są przez wszystkie jednostki we wspólnej przestrzeni adresowej, natomiast w drugim jednostki prowadzą obliczenia we własnej przestrzeni adresowej, przy czym rozdzielenie przestrzeni może być logiczne lub fizyczne.

Jednym z systemów wspomagających obliczenia równoległe są komputery wyposażone w procesor wielordzeniowy. W listopadzie 2006 roku firma NVIDIA wprowadziła na rynek karty graficzne wspierane przez architekturę CUDA (*Computed Unified Device Architecture*). CUDA jest architekturą wielordzeniowych procesorów graficznych, oznaczanych w dalszej części jako GPU. Wspiera ona wszystkie karty graficzne z serii G8x i nowsze. GPU dostępny na karcie może być traktowany jako procesor SIMD z pamięcią wspólną i może zostać użyty w celu przeprowadzenia obliczeń numerycznych. Moc obliczeniowa GPU umożliwia przyspieszenie obliczeń w porównaniu z obliczeniami wykonywanymi z użyciem procesora ogólnego przeznaczenia, tj. CPU. W obliczeniach z użyciem architektury CUDA stosowane jest przetwarzanie wielowątkowe. Projekt CUDA zakłada brak wrażliwości na zmianę sterowników, a także pełną przenośność i skalowalność programów, tj. możliwość wykonywania bez jakichkolwiek zmian dotychczas napisanych programów z użyciem GPU, które pojawią się w przyszłości, a będą miały większą moc obliczeniową, większą liczbę rdzeni i rejestrów oraz innych zasobów. Specyfikacja możliwości sprzętowych jest określana przez tzw. *compute capability*. Aktualnie dostępnymi wersjami *compute capability* są wersje 1.0, 1.1, 1.2, 1.3 i 2.0. Cechą wszystkich wersji jest kompatybilność wsteczna.

2. Architektura CUDA[1]

2.1. Architektura sprzętowa

Architektura sprzętowa CUDA zostanie opisana na przykładzie karty graficznej GeForce GTX 280 z procesorem graficznym GT 200 składającym się z 30 multiprocesorów, zgodnej z *compute capability* 1.3. Każdy multiprocesor (rys. 1) składa się z ośmiu rdzeni (ang. *cores*) C1 – C8 sterowanych przez jednostkę sterującą, dysponuje podręczną pamięcią instrukcji (ang. *instruction cache*), podręczną pamięcią danych, będącą pamięcią tylko do odczytu (ang. *constant cache*), a także współdzielonym obszarem pamięci (ang. *shared memory*). Wymienione rodzaje pamięci dostępne są dla wszystkich rdzeni multiprocesora. W skład multiprocesora wchodzi także 16 384 rejestry, jednostka arytmetyczna umożliwiająca obliczenia zmiennoprzecinkowe podwójnej precyzji (fp64), a także dwie jednostki arytmetyczne przeznaczone do obliczania funkcji specjalnych (ang. *special function unit*), np. funkcji trygonometrycznych, pierwiastkowania, odwrotności, eksponens.



Rys. 1. Architektura multiprocesora
Fig. 1. An architecture of multiprocessor

Pojedynczy rdzeń multiprocesora wykonuje jeden wątek programu, a w każdym cyklu wszystkie rdzenie multiprocesora wykonują jednocześnie instrukcje wątków. Wątki przetwarzane w tym samym multiprocesorze są automatycznie grupowane w bloki, przy czym miejsce przetwarzania bloków i kolejność ich wykonania są nieokreślone. Maksymalny rozmiar bloku jest równy 512 wątków. W pojedynczym cyklu multiprocesor przetwarza 32 wątki, zwane wiązką (ang. *warp*), a każdy multiprocesor może przetwarzać maksymalnie 32 wiązki. Oznacza to, że procesor GT 200 może przetwarzać 960 wiązek, czyli 30 720 wątków.

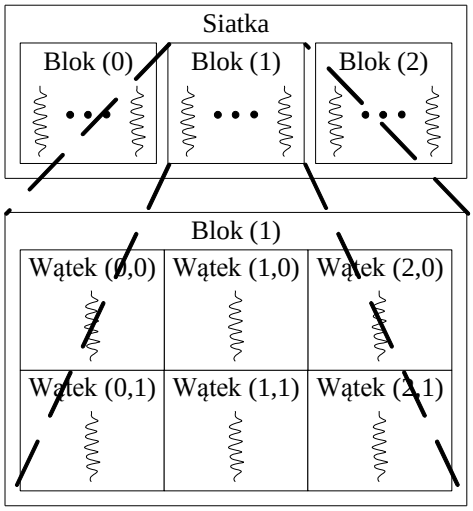
Komunikacja wątków należących do tego samego bloku może się odbywać za pomocą pamięci współdzielonej lub podręcznej pamięci danych. Komunikacja wątków należących do

[1] Materiałami źródłowymi, na podstawie których opracowano rozdział, były prace [1, 3, 10, 11, 12].

różnych bloków, a co za tym idzie wykonywanych przez różne multiprocesory, odbywa się przy użyciu pamięci globalnej karty graficznej, zwanej w dalszej części w skrócie pamięcią globalną, która jest dostępna dla wszystkich multiprocesorów. Zastosowany w obliczeniach sposób komunikacji ma znaczący wpływ na efektywność przeprowadzanych obliczeń. Otóż latencja (ang. *latency*), tj. czas między wysłaniem żądania a otrzymaniem odpowiedzi, w przypadku pamięci współdzielonej wynosi 8 cykli zegara, natomiast w przypadku pamięci globalnej 400 – 600 cykli. Zatem, w miarę możliwości obliczenia należy przeprowadzać tak, aby zminimalizować liczbę odwołań do pamięci globalnej.

2.2. Model programistyczny

Wraz z kartami graficznymi dostępne jest API dostarczające interfejs programistyczny programowania wielowątkowego z użyciem architektury CUDA, kompilator *nvcc* dla języka C, a także profiler umożliwiający dokonanie analizy wykonania programu korzystającego z bibliotek CUDA. Aby w pełni wykorzystać moc obliczeniową multiprocesorów, należy tak zaprojektować algorytm, by wykonywać jak najwięcej obliczeń, minimalizując odwołania do pamięci globalnej.



Rys. 2. Grupowanie wątków w bloki, a bloków w siatkę
Fig. 2. A grouping threads into blocks and grouping blocks into a grid

Wątki grupowane są w bloki, które mogą mieć geometrię jednowymiarową, dwuwymiarową lub trójwymiarową. Bloki wątków z kolei grupowane są w siatkę (ang. *grid*), która także może mieć geometrię jedno- dwu- lub trójwymiarową. Sposób grupowania bloków w siatkę nie musi być taki sam jak sposób grupowania wątków w blok. Na rysunku 2 przedstawiono siatkę o geometrii jednowymiarowej, składającą się z trzech bloków wątków. Każdy blok składa się z sześciu wątków i ma on geometrię dwuwymiarową o wymiarach 2×3.

Biblioteka CUDA dla języka C umożliwia zdefiniowanie funkcji, nazywanej jądrem (ang. *kernel*), która po wywołaniu wykonywana jest asynchronicznie przez wątki przy użyciu GPU. Jądro definiowane jest przez użycie kwalifikatora *__global__* w definicji funkcji. Nie może ono zwracać żadnej wartości, ponadto nie może być funkcją rekurencyjną i nie może zawierać zmiennych statycznych. W liście argumentów przekazywane są do jądra dane, na których mają być wykonywane obliczenia przez wątki. Dane te umieszczone są w pamięci globalnej i są one wspólne dla wszystkich wątków zgrupowanych w siatkę. Projektując jądro, należy unikać w nim instrukcji warunkowych, gdyż zmniejsza to efektywność obliczeń. Niech, przykładowo, będzie dany następujący kod jądra:

```
__global__ void f(lista argumentów)
{
   ...
   if (w)
      i_1;
   else
      i_2;
   ...
}
```

Niezależnie od wartości warunku *w* każdy wątek wykona obydwie instrukcje *i_1* oraz *i_2*. W wątkach, w których wartością warunku *w* jest logiczna prawda, instrukcja *i_2* zostanie zamaskowana, tj. wynik jej wykonania będzie unieważniony, co w rzeczywistości będzie miało skutek, jakby nie została wykonana. Podobna sytuacja jest z instrukcją *i_1*, która zostanie zamaskowana w wątkach, w których wartością warunku *w* jest logiczny fałsz.

Wykonanie obliczeń z użyciem architektury CUDA składa się z pięciu faz:

1. Przydzielenie w pamięci globalnej obszaru pamięci dla danych, na których będą wykonywane obliczenia przez jądro.
2. Przekopiowanie danych do przydzielonego obszaru pamięci.
3. Zainicjowanie przez CPU obliczeń wykonywanych przez GPU, tj. wywołanie kodu jądra.
4. Wykonanie przez wątki (z użyciem GPU) obliczeń zdefiniowanych w jądrze.
5. Przekopiowanie danych z pamięci globalnej do pamięci operacyjnej.

Wywołanie kodu jądra jest asynchroniczne, tj. po jego wywołaniu zwracane jest sterowanie do programu i mogą być kontynuowane obliczenia przez CPU. Podczas wykonywania obliczeń w systemie Windows z użyciem architektury CUDA należy odpowiednio dobrać rozmiar siatki i bloków wątków tak, aby czas obliczeń wykonywanych przez jądro nie przekroczył maksymalnego czasu zdefiniowanego w systemie [13]. Po jego przekroczeniu następuje przerwanie obliczeń.

3. Operacja mnożenia macierzy

Wpływ sposobu grupowania wątków oraz użycia pamięci współdzielonej i pamięci globalnej na wydajność obliczeń z użyciem GPU został zbadany na przykładzie operacji wyznaczania iloczynu macierzy: $A \cdot B = C$, gdzie macierze A i B są macierzami kwadratowymi o wymiarach $N \times N$. Dodatkowo przeprowadzono badania z użyciem CPU. W obliczeniach został zastosowany standardowy algorytm mnożenia macierzy [1], zgodnie z którym elementy macierzy $C = \{c_{i,j}\}$ $(1 \leq i, j \leq N)$ wyznaczane są według zależności:

$$c_{i,j} = \sum_{k=1}^{N} a_{i,k} \cdot b_{k,j} \tag{1}$$

W celu ułatwienia kopiowania macierzy do pamięci karty graficznej macierze są reprezentowane przez tablice jednowymiarowe o rozmiarach $N \cdot N$.

3.1. Mnożenie macierzy przy użyciu CPU

Operacja mnożenia macierzy przy użyciu CPU jest wykonywana przez funkcję *CPU* o następującej treści:

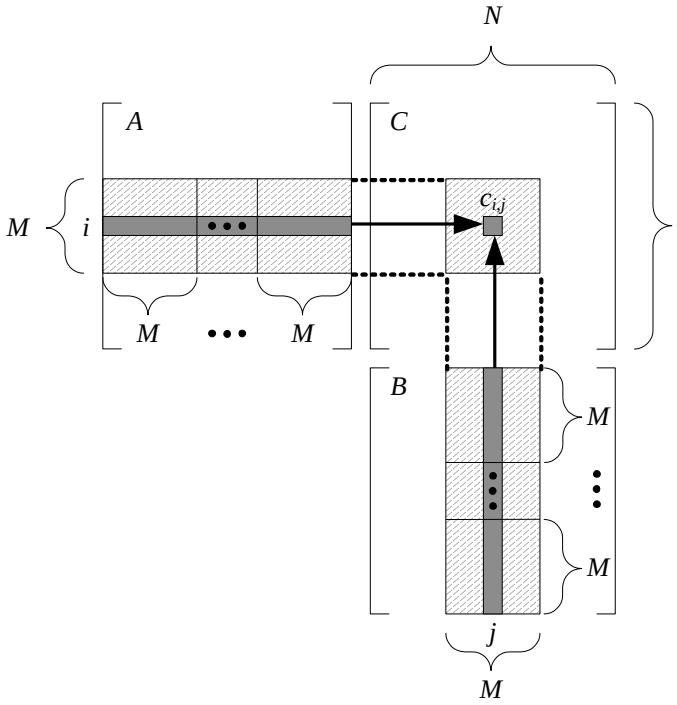
```
void CPU(int* A, int* B, int* C, int N)
{
   for (int i = 0; i < N; i++)
      for (int k = 0; k < N; k++)
      {
         C[i * N + k] = 0;
         for (int j = 0; j < N; j++)
            C[i * N + k] += A[i * N + j] * B[j * N + k];
      }
}
```

3.2. Mnożenie macierzy przy użyciu GPU

Mnożenie macierzy z użyciem GPU jest wykonywane przez blok wątków o geometrii dwuwymiarowej. W celu dokonania analizy wpływu rodzaju użytej pamięci na czas obliczeń rozpatrzono dwa warianty algorytmu. W pierwszym wariancie podczas obliczeń korzysta się wyłącznie z pamięci globalnej, a w drugim z pamięci współdzielonej. Ponadto, w obydwu wariantach zbadano dwa rodzaje bloków, różniące się między sobą liczbą wątków i wymiarem. W pierwszym przypadku użyty został blok o wymiarach $M \times M$, zawierający M^2 wątków, a w drugim przypadku blok o wymiarach $M \times 2M$, składający się z $2M^2$ wątków. W obydwu przypadkach, w celu równomiernego obciążenia wątków, wartości M ($M < N$) i N zostały dobrane tak, aby spełniona była zależność: $N \bmod 2M = 0$.

W przypadku bloku o wymiarach $M \times M$ operacja mnożenia macierzy wykonywana jest przez K^2 bloków wątków, gdzie $K = N / M$. Każdy wątek bloku wyznacza według zależności

(1) pojedynczą wartość $c_{i,j}$ macierzy C, a zatem pojedynczy blok wątków wyznacza M^2 elementów macierzy wynikowej, tj. podmacierz C o wymiarach $M \times M$ (rys. 3). W wersji algorytmu z pamięcią współdzieloną wewnątrz jądra przydzielane są dwa obszary pamięci współdzielonej o rozmiarach $M \times M$. Do obszarów tych wątki kopiują z pamięci globalnej odpowiednie elementy macierzy A i B, tj. podmacierze A i B o rozmiarach $M \times M$, przy czym każdy wątek kopiuje po jednym elemencie z każdej macierzy. Ponieważ do obszaru pamięci współdzielonej kopiowanych jest tylko M elementów, na podstawie których wyznaczana jest wartość c_{ij}, więc w celu jej wyznaczenia operacja kopiowania z pamięci globalnej do pamięci współdzielonej przeprowadzana jest K razy.



Rys. 3. Mnożenie macierzy za pomocą bloku wątków o wymiarach $M \times M$, zawierającego M^2 wątków
Fig. 3. A matrix multiplication using a block of threads of dimension $M \times M$ containing M^2 threads

Na podstawie elementów macierzy A i B znajdujących się w pamięci współdzielonej wątki w bloku wyznaczają wartości macierzy C. W wersji algorytmu bez pamięci współdzielonej podczas obliczeń elementy macierzy A i B odczytywane są bezpośrednio z pamięci globalnej. Dzięki kopiowaniu elementów macierzy A i B do pamięci współdzielonej minimalizowana jest liczba odwołań do pamięci globalnej podczas obliczeń. Pojedynczy blok wątków wyznacza M elementów wiersza macierzy C. Tak więc wyznaczając i-ty wiersz, zachodzi konieczność M-krotnego odwołania do każdego elementu i-tego wiersza macierzy A, natomiast podczas wyznaczania j-tej kolumny następuje M-krotne odwołanie do każdego elementu j-tej kolumny macierzy B. W wersji algorytmu bez użycia pamięci współdzielonej każdy z wymienionych elementów macierzy A i B jest więc M razy odczytywany z pamięci globalnej, natomiast dzięki użyciu pamięci współdzielonej jest on odczytywany tylko jeden

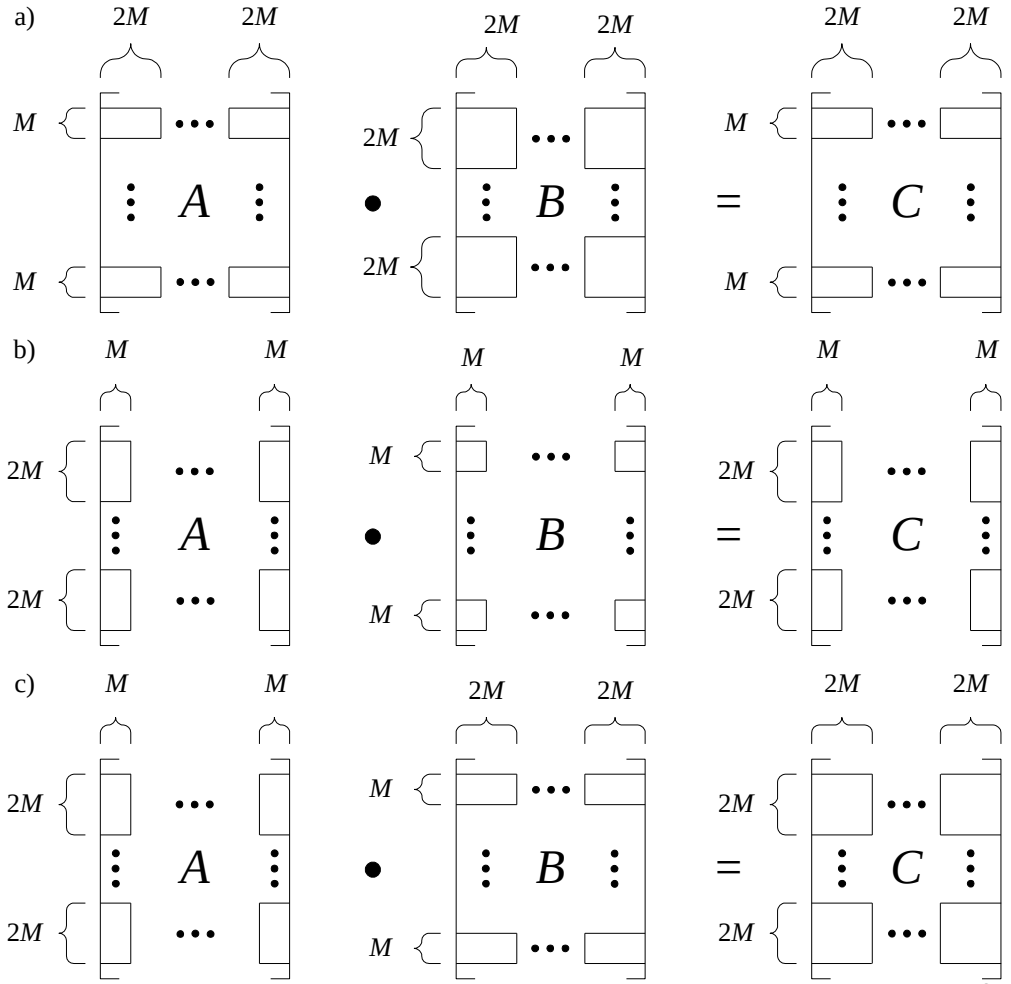
raz. Zatem, w wersji algorytmu bez użycia pamięci współdzielonej wyznaczenie wszystkich elementów *i*-tego wiersza macierzy *C* wymaga *N*-krotnego odczytania każdego elementu *i*-tego wiersza macierzy *A*, a w celu wyznaczenia wszystkich elementów *j*-tej kolumny konieczne jest *N*-krotne odczytanie każdego elementu *j*-tej kolumny macierzy *B*. W algorytmie z pamięcią współdzieloną elementy te są odczytywane tylko *K* razy. Wyznaczenie wszystkich elementów macierzy *C* bez użycia pamięci współdzielonej wymaga więc łącznie $2N^3$ odwołań do pamięci globalnej i $2N^3/M$ odwołań w przypadku użycia pamięci współdzielonej[2].

Drugim rodzajem użytego bloku wątków jest blok o wymiarach $M\times 2M$, zawierający $2M^2$ wątków. W tym przypadku przeprowadzenie obliczeń, tak aby każdy wątek wyznaczał pojedynczą wartość $c_{i,j}$ macierzy *C*, nie jest optymalnym rozwiązaniem. Postępując w ten sposób, pojedynczy blok wątków wyznacza podmacierz *C* o wymiarach $M\times 2M$ lub $2M\times M$. W przypadku wyznaczania podmacierzy *C* o wymiarach $M\times 2M$ do pamięci współdzielonej należy przekopiować podmacierz *A* o wymiarach $M\times 2M$, a także podmacierz *B* o wymiarach $2M\times 2M$ (rys. 4a). Każdy element macierzy *A* jest odczytywany z pamięci globalnej *K*/2 razy i *K* razy jest odczytywany każdy element macierzy *B*, co sprowadza się do $3N^3/2M$ odwołań do pamięci globalnej podczas wyznaczania macierzy *C*. W drugim przypadku, tj. wyznaczając podmacierz *C* o wymiarach $2M\times M$, do pamięci współdzielonej zachodzi konieczność przekopiowania podmacierzy *A* o wymiarach $2M\times M$ i podmacierzy *B* o wymiarach $M\times M$ (rys. 4b). W tym przypadku każdy element macierzy *A* jest odczytywany z pamięci globalnej *K* razy, a każdy element macierzy *B* jest odczytywany *K*/2 razy. Sprowadza się to, podobnie jak w poprzednim przypadku, do $3N^3/2M$ odwołań do pamięci globalnej podczas wyznaczania macierzy *C*. Warto jednak zwrócić uwagę na to, że tylko połowa wątków dokonuje kopiowania elementów macierzy *B* z pamięci globalnej do pamięci wspólnej. Wymaga to wprowadzenia w kodzie jądra instrukcji warunkowej, co jest zjawiskiem niekorzystnym. Zatem, z powodu opisanych niedogodności nie użyto rozwiązania, w którym każdy wątek wyznacza pojedynczy element macierzy *C*.

Dla bloku wątków o wymiarach $M\times 2M$ zaproponowano rozwiązanie, w którym każdy wątek wyznacza dwa elementy macierzy *C*, tj. wartości $c_{i,j}$ i $c_{i+M,j}$. W rozwiązaniu tym wewnątrz jądra przydzielane są dwa obszary pamięci współdzielonej, odpowiednio o rozmiarach $2M\times M$ i $M\times 2M$. Do pierwszego z nich kopiowana jest podmacierz *A*, natomiast do drugiego podmacierz *B*, gdzie każdy wątek kopiuje po jednym elemencie z macierzy *A* i *B* (rys. 4c). Pojedynczy blok wątków wyznacza $4M^2$ elementów macierzy wynikowej, tj. podmacierz *C* o wymiarach $2M\times 2M$. Każdy z elementów macierzy *A* i *B* jest odczytywany z pamięci

[2] Odwołania do pamięci globalnej dotyczą wyłącznie operacji odczytu elementów macierzy *A* i *B*, nie uwzględniają operacji zapisu elementów macierzy *C*, gdyż w obydwu przypadkach jest ona identyczna i wynosi N^2.

globalnej $K/2$ razy, co daje łącznie $2N^3/M$ odwołań do pamięci globalnej podczas wyznaczania macierzy C.



Rys. 4. Mnożenie macierzy za pomocą bloku wątków o wymiarach $M\times 2M$ złożonego z $2M^2$ wątków
Fig. 4. A matrix multiplication using a block of threads of dimension $M\times 2M$ containing $2M^2$ threads

Dla wersji algorytmu z użyciem pamięci współdzielonej i bez jej użycia wykonano obliczenia z użyciem dwóch rodzajów bloków wątków, które miały odpowiednio następujące wymiary $M\times M$ i $M\times 2M$. W każdym z przypadków zastosowano 3 rodzaje siatek grupujących bloki wątków: siatkę o geometrii dwuwymiarowej, siatkę o geometrii jednowymiarowej oraz siatkę zawierającą pojedynczy blok. Sumarycznie jest więc 12 wariantów przeprowadzenia obliczeń. W kolejnych punktach przedstawione zostaną sposoby przeprowadzenia obliczeń z użyciem bloku wątków o wymiarach $M\times 2M$ i pamięci współdzielonej dla wymienionych siatek grupujących. Obliczenia z użyciem bloków wątków o wymiarach $M\times M$ odbywają się w podobny sposób, dlatego nie będą szczegółowo omawiane.

3.2.1. Siatka bloków wątków o geometrii dwuwymiarowej

Bloki wątków o wymiarach $M \times 2M$ grupowane są w siatkę o geometrii dwuwymiarowej, mającą wymiary $K/2 \times K/2$. W rozwiązaniu tym w jednym wywołaniu kodu jądra wyznaczana jest wynikowa macierz *C*. Obliczenia przeprowadzane są w następujący sposób:

```
// przydzielenie w obszarze pamięci karty graficznej pamięci dla macierzy:
int *gpu_a, *gpu_b, *gpu_c;
assert(cudaMalloc((void**)&gpu_a, N * N * sizeof(int)) == cudaSuccess);
assert(cudaMalloc((void**)&gpu_b, N * N * sizeof(int)) == cudaSuccess);
assert(cudaMalloc((void**)&gpu_c, N * N * sizeof(int)) == cudaSuccess);

// przekopiowanie macierzy A i B do pamięci karty graficznej:
assert(cudaMemcpy(gpu_a, A, N * N * sizeof(int), cudaMemcpyHostToDevice) ==
       cudaSuccess);
assert(cudaMemcpy(gpu_b, B, N * N * sizeof(int), cudaMemcpyHostToDevice) ==
       cudaSuccess);

// ustalenie rozmiaru bloku wątków i rozmiaru siatki bloków:
dim3 blok(2 * M, M);
dim3 siatka(N / blok.x, N / blok.x);

// wywołanie kodu jądra:
GPU_SM_2D_M_2M<<< siatka, blok >>>(gpu_a, gpu_b, gpu_c);

// przekopiowanie wynikowej macierzy C z pamięci karty do pamięci operacyjnej:
assert(cudaMemcpy(C, gpu_c, N * N * sizeof(int), cudaMemcpyDeviceToHost) ==
       cudaSuccess);

// zwolnienie przydzielonej pamięci w obszarze karty graficznej:
cudaFree(gpu_a);
cudaFree(gpu_b);
cudaFree(gpu_c);
```

W pierwszym etapie, z użyciem bibliotecznej funkcji *cudaMalloc*, przydzielana jest w globalnym obszarze pamięci karty graficznej pamięć dla macierzy *A*, *B* i *C*. Następnie do przydzielonego obszaru pamięci kopiowane są macierze *A* i *B*. Do kopiowania danych użyta jest funkcja *cudaMemcpy*, umożliwiająca kopiowanie danych zarówno z pamięci operacyjnej do pamięci karty, jak i z pamięci karty do pamięci operacyjnej. Kierunek kopiowania danych określa jedna z dwóch flag: *cudaMemcpyHostToDevice* lub *cudaMemcpyDeviceToHost*. Jeżeli operacja przydziału pamięci i kopiowania danych zakończy się powodzeniem, to funkcje zwracają wartość *cudaSuccess*, w przeciwnym razie zwracają kod błędu, który jest zdefiniowany w pliku nagłówkowym *driver_types.h*.

W kolejnym kroku następuje zgrupowanie wątków w bloki, a bloków wątków w siatkę. Dokonując grupowania, określana jest geometria oraz rozmiar poszczególnych wymiarów. Grupowanie w geometrię dwu- lub trójwymiarową odbywa się z użyciem wektorowego typu *dim3*[3], którego składowymi są *x, y* i *z*. Następnie wywoływany jest kod jądra reprezentowany przez funkcję *GPU_SM_2D_M_2M*.

[3] Typ *dim3* jest typem danych zdefiniowanym w CUDA API.

W wywołaniu kodu jądra stosowana jest specjalna składnia: <<< ... >>>, w której podawane są kolejno: rozmiar i geometria siatki, rozmiar i geometria bloków wątków oraz opcjonalnie rozmiar pamięci współdzielonej, przydzielanej grupie wątków. Po wywołaniu kodu jądra kopiowana jest wynikowa macierz *C* z pamięci karty graficznej do pamięci operacyjnej. Ponieważ wywołanie kodu jądra jest asynchroniczne, więc operacja kopiowania danych z pamięci karty graficznej wprowadza w programie barierę synchronizacyjną, tj. wstrzymuje wykonanie programu do czasu zakończenia obliczeń przez jądro. Po wykonaniu obliczeń następuje zwolnienie pamięci przydzielonej w globalnym obszarze pamięci karty graficznej.

W jądrze reprezentowanym przez funkcję *GPU_SM_2D_M_2M* podczas obliczeń użyta jest pamięć współdzielona, przy czym jest ona przydzielana w jego wnętrzu, a nie podczas jego wywołania. Treść kodu jądra przedstawia się następująco:

```
__global__ void GPU_SM_2D_M_2M(int* a, int* b, int* c)
{
   // indeks początkowy i końcowy podmacierzy A:
      int a_pocz = N * blockDim.x * blockIdx.y;
      int a_kon = a_pocz + N - 1;

   // indeks początkowy podmacierzy B:
   int b_pocz = blockDim.x * blockIdx.x;

   // obliczane wartości wynikowe macierzy C:
      int c_1 = 0, c_2 = 0;

   // podmacierze A i B znajdujące się w pamięci współdzielonej:
      __shared__ int sm_a[2 * M][M], sm_b[M][2 * M];

      for (int i = a_pocz, j = b_pocz; i <= a_kon;
        i += blockDim.y, j += N * blockDim.y)
      {
         // przekopiowanie podmacierzy A i B do pamięci współdzielonej:
         sm_a[threadIdx.x][threadIdx.y] = a[i + N * threadIdx.x + threadIdx.y];
         sm_b[threadIdx.y][threadIdx.x] = b[j + N * threadIdx.y + threadIdx.x];

      // synchronizacja wątków:
         __syncthreads();

      // obliczenie wartości wynikowych macierzy C:
         for (int k = 0; k < blockDim.y; k++)
         {
            c_1 += sm_a[threadIdx.y][k] * sm_b[k][threadIdx.x];
            c_2 += sm_a[threadIdx.y + blockDim.y][k] * sm_b[k][threadIdx.x];
         }

      // synchronizacja wątków:
         __syncthreads();
      }

      // zapisanie obliczonych wartości w macierzy C:
      c[a_pocz + b_pocz + N * threadIdx.y + threadIdx.x] = c_1;
      c[a_pocz + b_pocz + N * threadIdx.y + threadIdx.x + N * blockDim.y] =
c_2;
}
```

Wewnątrz jądra jest dostęp do stałych *blockDim*, *blockIdx*, *threadIdx*, które umożliwiają odczytanie odpowiednio: rozmiaru bloku wątków, współrzędnych bloku w siatce oraz

indeksu bloku w siatce. Wymienione stałe są typu wektorowego, którego składowymi są współrzędne *x*, *y* i *z*. Stałe *blockDim*, *blockIdx* zostały użyte do ustalenia indeksów podmacierzy *A* i *B*, na podstawie których będą obliczane wynikowe wartości macierzy *C*. Można zauważyć, że zamiast stałej *blockDim* można użyć wartości *M*. Kolejnym krokiem jest przydzielenie w obszarze pamięci współdzielonej pamięci, do której zostaną przekopiowane elementy podmacierzy *A* i *B*. Przydział pamięci jest realizowany za pomocą kwalifikatora *__shared__* w deklaracji zmiennych. Następnie wątki obliczają $4M^2$ elementów macierzy *C*, przy czym każdy wątek oblicza dwa elementy, oznaczone jako *c_1* i *c_2*. Elementy te są obliczane w iteracji *for*, która jest wykonywana *K* razy. W każdej iteracji do pamięci współdzielonej kopiowanych jest z pamięci globalnej $2M^2$ elementów macierzy *A* i *B*, przy czym każdy wątek kopiuje po jednym z każdej macierzy. Na podstawie elementów macierzy *A* i *B* znajdujących się w pamięci współdzielonej obliczane są elementy *c_1* i *c_2*. Po przekopiowaniu danych następuje synchronizacja wątków, co ma zapewnić obecność w pamięci współdzielonej wszystkich danych, na których wykonywane są obliczenia. Synchronizacja następuje także po wykonaniu obliczeń. Po obliczeniu elementów *c_1* i *c_2* zapisywane są one w macierzy *C*, znajdującej się w pamięci globalnej.

W wersji jądra bez użycia pamięci współdzielonej, podobnie jak w przypadku kodów jądra opisanych w punktach 3.2.2 i 3.2.3, podczas wyznaczania elementów *c_1* i *c_2* elementy macierzy *A* i *B* odczytywane są bezpośrednio z pamięci globalnej karty graficznej.

3.2.2. Siatka bloków wątków o geometrii jednowymiarowej

W rozwiązaniu tym bloki wątków o wymiarach $M \times 2M$ grupowane są w siatkę o geometrii jednowymiarowej, zawierającą $K/2$ bloków. W pojedynczym wywołaniu kodu jądra nie jest wyznaczana w całości macierz *C*, jak w przypadku opisanym w punkcie 3.2.1, tylko jest wyznaczanych $2M$ wierszy macierzy *C*, tj. podmacierz *C* o wymiarach $2M \times N$. Wobec tego kod jądra, reprezentowany przez funkcję *GPU_SM_1D_M_2M*, wywoływany jest $K/2$ razy, gdzie w poszczególnych iteracjach wyznaczane są kolejne podmacierze C[4]:

```
// ustalenie rozmiaru bloku wątków i siatki bloków:
dim3 blok(2 * M, M);
size_t siatka = N / blok.x;

// wywołanie kodu jądra:
for (int y = 0; y < siatka; y++)
   GPU_SM_1D_M_2M<<< siatka, blok >>>(gpu_a, gpu_b, gpu_c, y);
```

Użyta siatka jest siatką jednowymiarową, dlatego do zapisu jej wymiaru nie jest konieczne stosowanie typu wektorowego *dim3*. W porównaniu z obliczeniami wykonywanymi z użyciem siatki dwuwymiarowej dodatkowo do jądra przekazywana jest wartość *y*, będąca

[4] Przydział pamięci współdzielonej, jej zwalnianie oraz kopiowanie danych odbywają się w taki sam sposób jak w przypadku siatki dwuwymiarowej, dlatego instrukcje te zostały pominięte.

indeksem obliczanej podmacierzy wynikowej, odpowiada ona współrzędnej *blockIdx.y* dla siatki dwuwymiarowej. Indeks *y* jest użyty wewnątrz jądra *GPU_SM_1D_M_2M* podczas wyznaczania wartości *p_pocz* zamiast stałej *blockIdx.y*, która jest niedostępna z racji użycia siatki jednowymiarowej. Jest to jedyna różnica w porównaniu z jądrem *GPU_SM_2D_M-_2M*, dlatego jego treść zostanie pominięta.

3.2.3. Siatka zawierająca pojedynczy blok wątków

Ostatnim rodzajem siatki grupującej bloki wątków jest siatka zawierająca pojedynczy blok o wymiarach $M \times 2M$. W pojedynczym wywołaniu kodu jądra jest więc obliczanych $4M^2$ elementów macierzy *C*, tj. podmacierz *C* o wymiarach $2M \times 2M$. Zatem, kod jądra, reprezentowany przez funkcję `GPU_SM_1_M_2M`, jest wywoływany $K^2/4$ razy w dwóch zagnieżdżonych iteracjach[5]:

```
// ustalenie rozmiaru bloku wątków:
dim3 blok(2 * M, M);

// wywołanie kodu jądra:
for (int y = 0; y < N / blok.x; y++)
   for (int x = 0; x < N / blok.x; x++)
      GPU_SM_1_M_2M<<< 1, blok >>>(gpu_a, gpu_b, gpu_c, x, y);
```

Do jądra *GPU_SM_1_M_2M*, w porównaniu z jądrem *GPU_SM_1D_M_2M*, dodatkowo przekazywany jest indeks *x*. Odpowiada on współrzędnej *blockIdx.x* dla siatki dwuwymiarowej i jest on użyty w celu wyznaczenia wartości *b_pocz*. Jest to jedyna różnica w porównaniu z jądrem *GPU_SM_1D_M_2M*, dlatego jego treść nie zostanie zamieszczona.

4. Wyniki badań eksperymentalnych

Podczas badań eksperymentalnych obliczenia przy użyciu CPU wykonano stosując algorytm opisany w podrozdziale 3.1, a przy użyciu GPU z zastosowaniem algorytmów oraz siatek grupujących bloki wątków, które opisano w podrozdziale 3.2. W obliczeniach przy użyciu GPU przyjęto wymiar bloku $M = 16$, tj. wątki były grupowane w dwuwymiarowe bloki o wymiarach 16×16 (256 wątków) i 16×32 (512 wątków). Drugi z użytych bloków jest blokiem o maksymalnym rozmiarze. Wykonana została operacja mnożenia macierzy kwadratowych o rozmiarach *N* z zakresu 160, …, 6240. Badania zostały przeprowadzone na komputerze PC z procesorem Intel Core 2 Duo P8400 2.2 GHz, 4 GB RAM i kartą graficzną NVIDIA GeForce 9600M GT wyposażoną w 4 multiprocesory, co daje łącznie 32 rdzenie.

[5] Przydział i zwalnianie pamięci współdzielonej oraz kopiowanie danych odbywają się w taki sam sposób jak w poprzednich siatkach grupujących bloki wątków.

Celem przeprowadzonych badań eksperymentalnych było:

- ocenienie wydajności przeprowadzonych obliczeń przy użyciu CPU i GPU,
- zbadanie wpływu użycia pamięci współdzielonej i pamięci globalnej na czas obliczeń,
- zbadanie wpływu geometrii siatki grupującej bloki wątków na czas obliczeń,
- zbadanie wpływu rozmiaru bloku wątków na czas obliczeń.

Tabela 1

Zależność czasu obliczeń za pomocą GPU (bez użycia pamięci współdzielonej) od rozmiaru macierzy *N*, dla siatki o topologii 2D, 1D i zawierającej pojedynczy blok wątków oraz bloków wątków o wymiarach 16×16 i 16×32. Przyjętą jednostką czasu jest milisekunda

N	2D, 16×16	2D, 16×32	1D, 16×16	1D, 16×32	1, 16×16	1, 16×32
160	13,18	13,66	14,72	13,10	35,10	17,50
320	104,31	97,77	106,11	99,52	191,01	113,36
480	347,88	326,22	350,89	330,05	558,79	361,86
640	1109,60	1437,85	1408,17	1315,02	1738,43	1496,72
800	1607,73	1505,19	1600,29	1514,58	2270,51	1605,25
960	2765,62	2596,20	2749,46	2586,25	3676,66	2711,73
1120			4395,70	4095,93	5873,12	4254,29
1280			10634,09	10255,08	11681,44	11096,76
1440			9376,95	8743,08	12126,07	9021,87
1600			12728,59	11956,71	15847,18	12308,33
1760			17161,65	16102,39	21622,08	16483,04
2080			28362,01	26481,42	38022,41	26557,72
2400			43705,91	41515,09	53823,06	41454,95
2720			63821,65	59316,21	77742,55	58943,82
3040			89681,05	87199,73	109441,97	88152,65
3360			121366,55	112822,98	146578,51	110497,61
3680			159127,62	147986,42	187588,84	147116,58
4000			204410,45	190481,66	242134,42	189140,84
4320			257561,33	239324,47	303221,47	239458,64
4640			319456,31	297706,97	374951,18	297049,91
4960			390510,09		456537,44	357861,78
5280			471545,12		567421,75	436777,00
5600			562506,69		662444,44	514510,00
5920					815949,88	614306,63
6240					997832,00	746653,88

Czasy obliczeń za pomocą GPU bez użycia pamięci współdzielonej zostały przedstawione w tabeli 1, a z użyciem pamięci współdzielonej w tabeli 3. W kolumnach obydwu tabel zamieszczono informację o geometrii siatki grupującej bloki wątków i wymiarze bloku wątków użytych podczas obliczeń. Przez „2D” oznaczona została siatka dwuwymiarowa, przez „1D” siatka jednowymiarowa, a „1” oznacza siatkę, zawierającą pojedynczy blok wątków. W tabeli 2 przedstawiono czasy obliczeń za pomocą CPU.

Tabela 2

Zależność czasu obliczeń za pomocą CPU od rozmiaru macierzy. Przyjętą jednostką czasu jest milisekunda

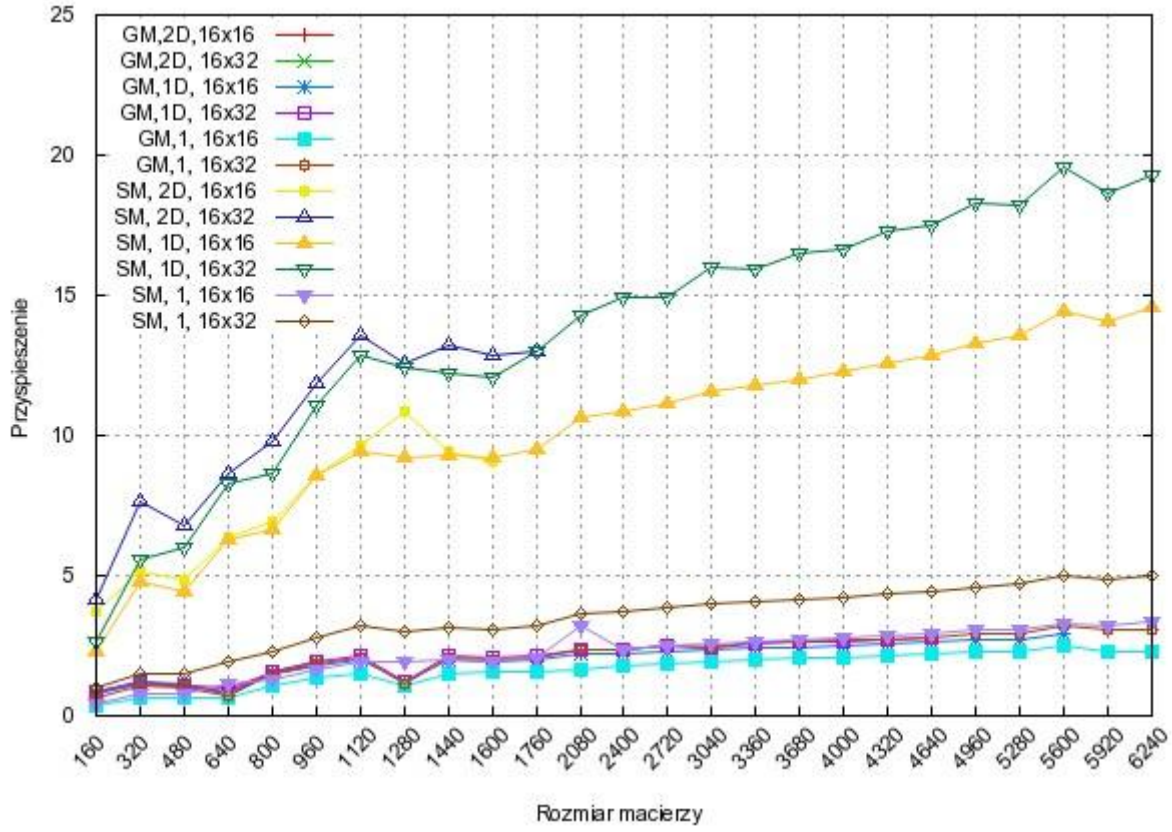
N	CPU		*N*	CPU		*N*	CPU
160	11,51		1600	24370,50		4320	656318,75
320	121,42		1760	34048,06		4640	832212,12
480	363,97		2080	62823,68		4960	1049322,50
640	1111,24		2400	98075,22		5280	1289284,50
800	2360,47		2720	146258,17		5600	1636224,25
960	4982,39		3040	211622,28		5920	1880821,63
1120	8874,86		3360	291360,28		6240	2278871,50
1280	12727,70		3680	387794,06			
1440	18436,38		4000	508773,56			

Tabela 3

Zależność czasu obliczeń za pomocą GPU (z użyciem pamięci współdzielonej) od rozmiaru macierzy N, dla siatki o topologii 2D, 1D i zawierającej pojedynczy blok wątków oraz bloków wątków o wymiarach 16×16 i 16×32. Przyjętą jednostką czasu jest milisekunda

N	2D, 16×16	2D, 16×32	1D, 16×16	1D, 16×32	1, 16×16	1, 16×32
160	3,12	2,76	5,12	4,31	26,76	11,86
320	23,49	15,96	25,39	21,77	156,34	79,48
480	74,95	53,93	81,90	60,82	444,98	242,35
640	175,16	128,39	175,78	133,68	970,04	565,63
800	339,85	240,31	353,79	273,51	1783,98	1045,48
960	582,41	420,95	581,16	451,46	2988,70	1771,41
1120	922,22	654,84	941,07	690,49	4599,30	2779,73
1280	1175,41	1009,96	1382,66	1021,33	6690,17	4253,33
1440	1953,36	1394,15	1978,33	1508,92	9338,58	5822,69
1600	2679,20	1899,50	2654,42	2023,40	12673,85	7936,62
1760		2616,55	3579,85	2623,24	16666,66	10509,59
2080			5890,73	4408,86	19713,19	17218,39
2400			9012,26	6569,83	41112,14	26323,62
2720			13121,59	9805,21	59150,77	38172,14
3040			18248,24	13217,94	82081,28	53190,79
3360			24642,69	18291,64	110119,51	71568,74
3680			32286,07	23542,33	142969,70	93866,52
4000			41474,11	30608,04	183437,08	120336,65
4320			52141,91	37911,79	229534,47	151358,88
4640			64607,38	47637,02	283394,75	187248,38
4960			78787,78	57476,45	344960,44	228519,19
5280			95086,47	70768,63	415371,81	275382,84
5600			113300,26	83655,02	494321,78	328482,81
5920			133974,20	100809,84	584558,69	388256,56
6240			156771,28	118265,61	683212,75	455517,25

Najmniejszy czas obliczeń został uzyskany dla siatki o topologii 2D, jednak maksymalnymi rozmiarami macierzy, dla których udało się przeprowadzić badania, były macierze o rozmiarach 1760 (dla pamięci współdzielonej) i 960 (dla pamięci globalnej). Czas obliczeń w obydwu przypadkach wynosi ok. 2 sekund. Przeprowadzenie obliczeń dla macierzy o większych rozmiarach było niemożliwe z powodu opisanego w podrozdziale 2.2. Każdorazowe wywołanie kodu jądra jest związane z pewnym narzutem czasowym. Najwięcej wywołań kodu jądra jest w przypadku siatki zawierającej pojedynczy blok, dlatego czas obliczeń dla tej siatki jest największy. Dla dwuwymiarowej siatki bloków następuje tylko jednokrotne wywołanie kodu jądra, a czas obliczeń jest zbliżony do czasu obliczeń dla siatki jednowymiarowej.



Rys. 5. Zależność przyspieszenia od rodzaju użytej pamięci, topologii siatki i rozmiaru bloku wątków
Fig. 5. A speed-up depending on used type of memory, grid topology and size of block of threads

Czas obliczeń przy zastosowaniu pamięci współdzielonej jest mniejszy od czasu obliczeń, podczas których używana jest wyłącznie pamięć globalna. Fakt ten wynika z różnicy w czasie dostępu do pamięci współdzielonej i globalnej. Dla największych przebadanych macierzy, dla siatki zawierającej jeden blok wątków, jest on ok. 1.5 razy mniejszy, a dla siatek o topologiach jedno- i dwuwymiarowej jest ok. 5 razy mniejszy. Dokonując porównania czasów obliczeń za pomocą CPU i GPU dla macierzy o maksymalnym rozmiarze, można zauważyć, że wykonując obliczenia z użyciem GPU, pamięci współdzielonej i jednowymiarowej siatki

bloków, uzyskano odpowiednio ok. 20-krotne przyspieszenie dla bloku wątków o wymiarach 16×32 i prawie 15-krotne przyspieszenie dla bloku o wymiarach 16×16 (rys. 5[6]). W przypadku siatki zawierającej tylko jeden blok uzyskano dla tych bloków odpowiednio tylko ok. 5-krotne i ok. 3-krotne przyspieszenie. Jest ono niewiele większe od maksymalnego przyspieszenia dla obliczeń z użyciem GPU i wyłącznie pamięci globalnej, które wynosi prawie 2.5, a zostało uzyskane dla siatki zawierającej pojedynczy blok wątków o wymiarach 16×32. Dla macierzy o rozmiarach nie większych niż 1760 największe przyspieszenie zostało uzyskane w przypadku obliczeń przeprowadzonych z użyciem GPU, dwuwymiarowej siatki bloków i bloku wątków o wymiarach 16×32.

Tabela 4

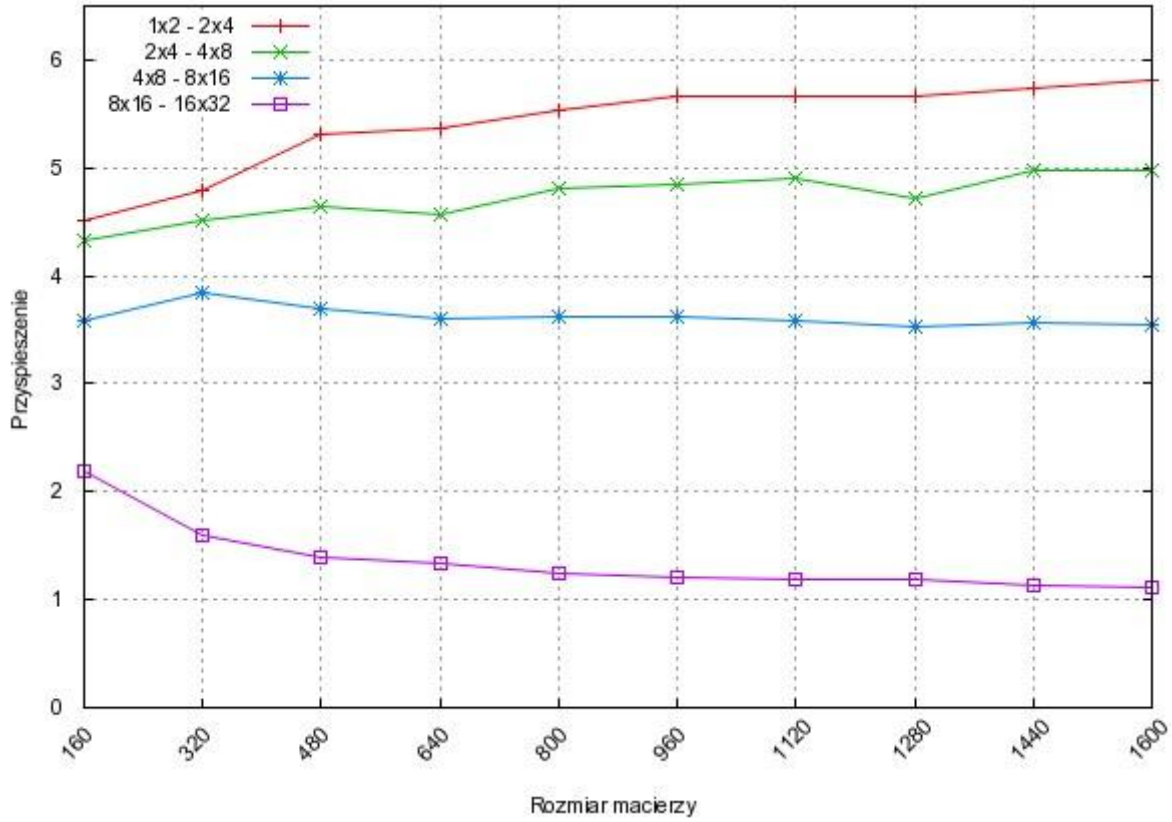
Zależność czasu obliczeń za pomocą GPU z użyciem pamięci współdzielonej od rozmiaru macierzy *N* i rozmiaru bloku wątków. Przyjętą jednostką czasu jest milisekunda

N	1×2	2×4	4×8	8×16	16×32
160	1821,47	403,55	93,17	25,98	11,86
320	10537,38	2196,87	487,14	126,57	79,48
480	30969,29	5831,38	1255,77	340,49	242,35
640	67318,05	12545,98	2745,68	760,66	565,63
800	125995,62	22789,77	4742,83	1306,24	1045,48
960	211780,00	37427,31	7730,57	2137,69	1771,41
1120	326619,31	57604,91	11752,63	3278,55	2779,73
1280	475871,09	84108,06	17810,19	5046,52	4253,33
1440	671058,94	116888,32	23528,43	6588,69	5822,69
1600	913281,75	157177,38	31586,68	8888,55	7936,62

Ostatnim celem badań było określenie wpływu rozmiaru bloku wątków na czas obliczeń. W tym celu wykonano dla macierzy o rozmiarach z zakresu 160, …, 1600 obliczenia z użyciem GPU, pamięci współdzielonej i siatki zawierającej pojedynczy blok. Badania przeprowadzono dla pięciu różnych bloków wątków, odpowiednio o wymiarach: 1×2, 2×4, 4×8, 8×16 i 16×32, a uzyskane czasy obliczeń przedstawiono w tabeli 4. Tylko dla dwóch bloków wątków, tj. o wymiarach 8×16 i 16×32, czasy obliczeń są mniejsze od czasów obliczeń z użyciem CPU. Wprawdzie dla bloku 8×16 i macierzy o rozmiarach 160 i 320 czas ten jest większy, jednak dla pozostałych macierzy jest mniejszy, a dla największej z przebadanych macierzy jest on ok. 3 razy mniejszy. Wynika to z tego, że wraz ze wzrostem rozmiaru przetwarzanych danych rośnie liczba wykonywanych operacji, co umożliwia w pełni wykorzystanie mocy obliczeniowej GPU, a co za tym idzie kompensuje latencję pamięci globalnej. Z podobną sytuacją mamy do czynienia w przypadku rozmiaru bloku wątków. W bloku o małych rozmiarach, jak blok 1×2, liczba wykonywanych operacji jest sto-

[6] Na rysunku przez SM oznaczono obliczenia z użyciem pamięci współdzielonej, a przez GM z użyciem wyłącznie pamięci globalnej. Pozostałe wartości oznaczają topologię siatki bloków (2D, 1D, 1) i wymiar bloku wątków (16 × 32, 16 × 16).

sunkowo mała w porównaniu z liczbą odwołań do pamięci globalnej, co nie pozwala skompensować latencji pamięci globalnej. Zatem, czas obliczeń z użyciem GPU dla macierzy o rozmiarach 1600 jest ok. 37 razy większy od czasu obliczeń z użyciem CPU.



Rys. 6. Zależność przyspieszenia od rozmiaru bloku wątków dla obliczeń z użyciem GPU i pamięci współdzielonej

Fig. 6. A computational speed-up depending on size of block of threads used GPU and shared memory

Na rys. 6 przedstawiono wykres uzyskanego przyspieszenia w zależności od rozmiaru bloku wątków. Przyspieszenie jest rozumiane jako stosunek czasu obliczeń z użyciem danego bloku wątków do czasu obliczeń z użyciem bloku wątków o większym rozmiarze. Przykładowo, przez „1×2 – 2×4” oznaczono przyspieszenie, jakie uzyskano zastępując w obliczeniach blok wątków o wymiarach 1×2 blokiem o wymiarach 2×4. Największe przyspieszenie (ok. 6-krotne) uzyskano zastępując blok 1×2 blokiem 2×4, natomiast najmniejsze (ok. 1.1) przez zamianę bloków 8×16 i 16×32. Niewielkie przyspieszenie dla bloku wątków o największym rozmiarze wynika z liczby rdzeni procesora graficznego karty użytej podczas badań eksperymentalnych.

5. Podsumowanie

W niniejszej pracy omówiono architekturę CUDA, będącą architekturą wielordzeniowych procesorów graficznych. Omówiona została architektura sprzętowa, a także model programistyczny umożliwiający użycie GPU dostępnego na karcie graficznej w celu przeprowadzenia obliczeń. GPU może być traktowany jako procesor SIMD z pamięcią wspólną, w którym obliczenia wykonywane są wielowątkowo. Obliczenia wykonywane w ramach wątku umieszczane są wewnątrz funkcji, zwanej jądrem. W trakcie obliczeń wątki grupowane są w bloki, natomiast bloki wątków grupowane są w siatkę. Wątki wykonujące obliczenia mają dostęp do szybkiej pamięci współdzielonej oraz pamięci globalnej karty graficznej.

W pracy skupiono się na zbadaniu wpływu użycia rodzaju pamięci oraz grupowania wątków na czas obliczeń. Badania przeprowadzono na przykładzie operacji mnożenia macierzy kwadratowych o wymiarach $N \times N$. Obliczenia przy użyciu GPU przeprowadzone zostały przez bloki wątków o wymiarach $M \times M$ i $M \times 2M$, dla których rozpatrzone zostały 3 rodzaje siatek różniących się geometrią, tj. siatki jedno- i dwuwymiarowa oraz siatka zawierająca pojedynczy blok wątków. W każdym przypadku przeprowadzono obliczenia z użyciem pamięci współdzielonej oraz wyłącznie pamięci globalnej. W badaniach przyjęto rozmiar macierzy N z zakresu 160, …, 6240 oraz rozmiar bloku wątków $M = 16$. Każdorazowe wywołanie kodu jądra związane jest z pewnym narzutem czasowym. Zatem, najmniejszy czas obliczeń został uzyskany dla dwuwymiarowej siatki bloków, dla której kod jądra jest wywoływany tylko jednokrotnie, a największy dla siatki zawierającej pojedynczy blok wątków, gdzie następuje najwięcej wywołań kodu jądra. W systemie Windows czas wykonania pojedynczego wywołania kodu jądra nie powinien przekroczyć pewnej granicznej wartości, zdefiniowanej w systemie [13]. Z tego powodu dla siatki dwuwymiarowej nie było możliwe przeprowadzenie obliczeń dla wszystkich macierzy z podanego zakresu. Zgodnie z przewidywaniami, czas obliczeń z użyciem wyłącznie pamięci globalnej jest większy od czasu obliczeń z użyciem pamięci współdzielonej.

Czas obliczeń z użyciem architektury CUDA zależy od liczby dostępnych rdzeni procesora graficznego. Zwiększenie rozmiaru bloku wątków nie zawsze prowadzi do znaczącego zmniejszenia czasu obliczeń. Zastępując blok 2 wątków (o wymiarach 1×2) blokiem 8 wątków (o wymiarach 2×4), uzyskano prawie 6-krotne przyspieszenie, a ok. 5-krotne przyspieszenie uzyskano przez zastąpienie bloku 8 wątków (o wymiarach 2×4) blokiem 32 wątków (o wymiarach 4×8). Najmniejsze przyspieszenie, ok. 1.1, uzyskane zostało przez zamianę bloku 256 wątków (o wymiarach 8×16) blokiem 512 wątków (o wymiarach 16×32), co wynika z liczby rdzeni procesora graficznego karty, która została użyta podczas badań eksperymentalnych.

Przeprowadzone badania potwierdziły, że podczas obliczeń należy zminimalizować liczbę odwołań do pamięci globalnej oraz wykonywać jak najwięcej obliczeń, aby wykorzystać moc obliczeniową procesora graficznego, kompensując w ten sposób czas dostępu do pamięci. Ponadto, korzystne jest przekopiowanie danych z pamięci globalnej do pamięci współdzielonej i w trakcie obliczeń odczytywanie danych z pamięci współdzielonej zamiast z pamięci globalnej. Na podstawie przeprowadzonych badań można też wyciągnąć wniosek odnośnie do sposobu zarządzania blokami – wątkami. Otóż, należy tworzyć siatki zawierające wiele bloków, a użycie siatek zawierających pojedyncze bloki wątków nie jest efektywne.

BIBLIOGRAFIA

1. Cormen T. H., Leiserson C. E., Rivest R. L.: Wprowadzenie do algorytmów. WNT, Warszawa 2000.
2. Kirk D. B., Hwu W. W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 1st edition (February 5, 2010).
3. Mattson T. G., Sanders B. A., Massingill B. L.: Patterns for Parallel Programming. Addison-Wesley Professional, 1st edition (September 25, 2004).
4. Pacheco P.: Parallel Programming with MPI. Morgan Laufmann, 1st edition (October 15, 1996).
5. Parberry I.: Parallel Complexity Theory. John Wiley & Sons Inc (September 1987).
6. Parhami B.: Introduction to Parallel Processing: Algorithms and Architectures. Springer, 1st Edition (January 31, 1999).
7. Rauber T., Rünger G.: Parallel Programming for Multicore and Cluster Systems. Springer, 1st edition (March 10, 2010).
8. Sanders J., Kandrot E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, 1st edition (July 30, 2010).
9. Wittwer T.: An Introduction to Parallel Programming. VSSD, 1st edition, 2006.
10. NVIDIA CUDA Best Practices Guide Version 3.0. NVIDIA Corporation (February 4, 2010), http://www.nvidia.pl/object/cuda_get_pl.html.
11. NVIDIA CUDA Programming Guide Version 3.0. NVIDIA Corporation (February 20, 2010), http://www.nvidia.pl/object/cuda_get_pl.html.
12. NVIDIA CUDA Reference Manual Version 3.0. NVIDIA Corporation (February, 2010), http://www.nvidia.pl/object/cuda_get_pl.html.
13. Timeout Detection and Recovery of GPUs through WDDM. Windows Hardware Developer Central, Microsoft Corporation, (Updated: April 27, 2009), http://www-.microsoft.com/whdc/device/display/wddm_timeout.mspx.

Recenzent: Dr inż. Arkadiusz Sochan

Wpłynęło do Redakcji 14 września 2010 r.

Abstract

In the paper we described CUDA architecture which is a architecture of multi-core graphics processors (GPUs). A hardware architecture (fig. 1) and a programming model that allows the use of the GPU for general purpose computing, are presented. The GPU can be treated as a SIMD processor with shared memory and the CUDA programming model is a set of threads running in parallel. Each thread executes a single instruction set represented by a function called the kernel. Threads are grouped into a blocks while a set of blocks of threads are grouped into a grid (fig. 2). Each thread and block has given unique ID that can be accessed within the kernel. All threads have access to a fast shared memory and a global memory.

The influence of used kind of memory and blocks of threads management on time of computation using the CUDA architecture were researched. The research was conducted on an example of matrix multiplication of matrices of $N \times N$ dimensions. There are used grouping threads into blocks of dimensions $M \times M$ (fig. 3) and $M \times 2M$ (fig. 4) and three kinds of grids, i.e. one-dimensional, two-dimensional and a grid containing single block, are examined. We tested it on a Windows PC with Intel Core 2 Duo P8400 2.2 GHz, 4 GB of RAM and NVIDIA GeForce 9600M GT graphics card. The GPU of the card includes 32 cores. The shortest running time has been obtained for the two-dimensional grid and the longest for the grid containing single block of threads. As anticipated, the running time of computation with a global memory is greater than the running time of computation with a shared memory.

The running time using the CUDA depends on the number of available cores of GPU. Increasing number of threads does not always lead to a significant reduction of the running time. The biggest speed-up (about 6 times) was obtained by replacing block of threads of dimension 1×2 to the block of dimension 2×4 and the smallest (about 1.1 times) was obtained by replacing block of threads of dimension 8×16 to the block of dimension 16×32 (fig. 6). Obtained speed-up results from a number of cores of GPU used during the test.

Adres

Jacek WIDUCH: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, jacek.widuch@polsl.pl .