Programowanie kart graficznych

Architektura i API
część 2

CUDA
hierarchia pamięci c.d.

Globalna pamięć urządzenia:

- funkcje CUDA API takie jak `cudaMalloc()` i `cudaFree()` z założenia służą do manipulowania pamięcią adresowaną **liniowo** (czyli predystynowaną do przechowywania i obróbki wektorów)
- nie wyklucza to użycia tych funkcji do obróbki tablic dwu- i trójwymiarowych, ale dla tych zastosowań CUDA API przewiduje użycie funkcji specjalizowanych
- są to:
  - `cudaMallocPitch()` - dla macierzy 2D
  - `cudaMalloc3D()` - dla macierzy 3D

Globalna pamięć urządzenia:

- użycie tych funkcji gwarantuje, że tablica zostanie rozmieszczona w pamięci w optymalny sposób, gwarantujący najkrótszy czas dostępu do wierszy oraz przy kopiowaniu jej elementów do innych regionów pamięci urządzenia
- najpierw rozpatrzymy aspekt 2D takiego przydziału

CUDA API – zarządzanie pamięcią:

przydzielenie pamięci dla tablicy 2D

```
cudaError_t cudaMallocPitch(
    void     **devPtr,
    size_t   *pitch,
    size_t   width,
    size_t   height
);
```

Funkcja przydziela co najmniej `width*height` bajtów i zwraca wskaźnik do przydzielonej pamięci w zmiennej `*devPtr`; funkcja może dopełnić każdy z wierszy dodatkowymi bajtami celem optymalnego rozmieszczenia danych; rzeczywisty rozmiar wiersza jest zwracany w `pitch`.

cudaMallocPitch:

parametry:

- `devPtr` : wskaźnik na wskaźnik reprezentujący przydzielony blok pamięci
- `pitch` : wskaźnik na daną reprezentującą przydzielony faktycznie rozmiar wiersza
- `width` : rozmiar wiersza przydzielanej tablicy (liczba bajtów)
- `height` : rozmiar kolumny przydzielanej tablicy (liczba wierszy)

wynik:

- `cudaSuccess` – poprawne zakończenie działania funkcji
- `cudaErrorMemoryAllocation` – błąd przydziału, awaryjny powrót z funkcji

cudaMallocPitch – przykład

(iterowanie po elementach tablicy 2D)

```
float  *devPtr;
int     pitch;

cudaMallocPitch((void**)&devPtr, &pitch,width * sizeof(float), height);

MyKernel<<<100, 512>>>(devPtr, pitch);

__global__ void MyKernel(float* devPtr, int pitch)
{
    for (int r = 0; r < height; ++r) {
        // wyliczenie adresu początkowego wiersza
        float *row = (float *)((char *)devPtr + r * pitch);

        // iterowanie po elementach wiersza
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

Pamięć urządzenia:

- a teraz pora na aspekt 3D

CUDA API – zarządzanie pamięcią:

reprezentowanie rozmiarów tablicy 3D

```
struct cudaExtent {
   size_t depth; // głębokość (liczba elementów)
   size_t height;// wysokość  (liczba elementów)
   size_t  width; // szerokość (liczba bajtów(!))
};
```

CUDA API – zarządzanie pamięcią:

wytworzenie struktury typu cudaExtent:

```
struct cudaExtent
make_cudaExtent(size_t w, size_t h, size_t d);
```

parametry:

- w : szerokość definiowanej tablicy (liczba bajtów)
- h : wysokość definiowanej tablicy (liczba elementów)
- d : głębokość definiowanej tablicy (liczba elementów)

wynik:

- nowo utworzona struktura typu `cudaExtent` wypełniona podanymi danymi

CUDA API – zarządzanie pamięcią:

reprezentowanie przydziału tablicy 3D

```
struct cudaPitchedPtr {
   size_t pitch; // rozmiar wiersza (liczba bajtów)
   void   ptr;   // wskaźnik na przydzieloną pamięć
   size_t xsize; // logiczna szerokość przydziału
                 // (liczba elementów)
   size_t ysize; // logiczna wysokość przydziału
                 // (liczba elementów)
};
```

CUDA API – zarządzanie pamięcią:

wytworzenie struktury typu cudaPitchedPtr:

```
struct cudaPitchedPtr
make_cudaPitchedPtr
(void *d, size_t p, size_t xsz, size_t ysz);
```

parametry:

- d : wskaźnik na przydzieloną tablicę
- p : rozmiar wiersza fizycznego (liczba bajtów)
- h : logiczna szerokość tablicy (liczba elementów)
- d : logiczna głębokość tablicy (liczba elementów)

wynik:

- nowo utworzona struktura typu `cudaPitchedPtr` wypełniona podanymi danymi

CUDA API – zarządzanie pamięcią:

przydzielenie pamięci dla tablicy 3D

```
cudaError_t cudaMalloc3D(
    struct cudaPitchedPtr *pitchedDevPtr,
    struct cudaExtent         extent
);
```

Funkcja przydziela co najmniej `width*height*depth` bajtów i wypełnia danymi strukturę wskazywaną przez `*pitchedDevPtr`; funkcja może dopełnić każdy z wierszy dodatkowymi bajtami celem optymalnego rozmieszczenia danych; rzeczywisty rozmiar wiersza jest zwracany w polu `pitch`.

cudaMallocPitch:

parametry:

- `pitchedDevPtr` : wskaźnik na strukturę przeznaczoną do reprezentowania przydzielonego bloku pamięci (zostanie wypełniona danymi przez funkcję)
- `extent`: struktura opisująca rozmiary przydzielanej tablicy

wynik:

- `cudaSuccess` – poprawne zakończenie działania funkcji
- `cudaErrorMemoryAllocation` – błąd przydziału, awaryjny powrót z funkcji

cudaMalloc3D – przykład

(iterowanie po elementach tablicy 3D)

```
CudaPitchedPtr    devPitchedPtr;
CudaExtent        extent = make_cudaExtent(64, 64, 64);

cudaMalloc3D(&devPitchedPtr, extent);

MyKernel<<<100, 512>>>(devPitchedPtr, extent);

__global__ void MyKernel(cudaPitchedPtr devPitchedPtr, cudaExtent extent)
{
    char* devPtr      = devPitchedPtr.ptr;
    size_t pitch      = devPitchedPtr.pitch;
    size_t slicePitch = pitch * extent.height;

    for(int z = 0; z < extent.depth; ++z) {
        //wyliczenie adresu początku „plasterka”
        char* slice = devPtr + z * slicePitch;
        for(int y = 0; y < extent.height; ++y) {
            // wyliczenie adresu początku wiersza
            float *row = (float *)(slice + y * pitch);
            // iterowanie po wierszu
            for(int x = 0; x < extent.width; ++x) {
                float element = row[x];
            }
        }
    }
}
```

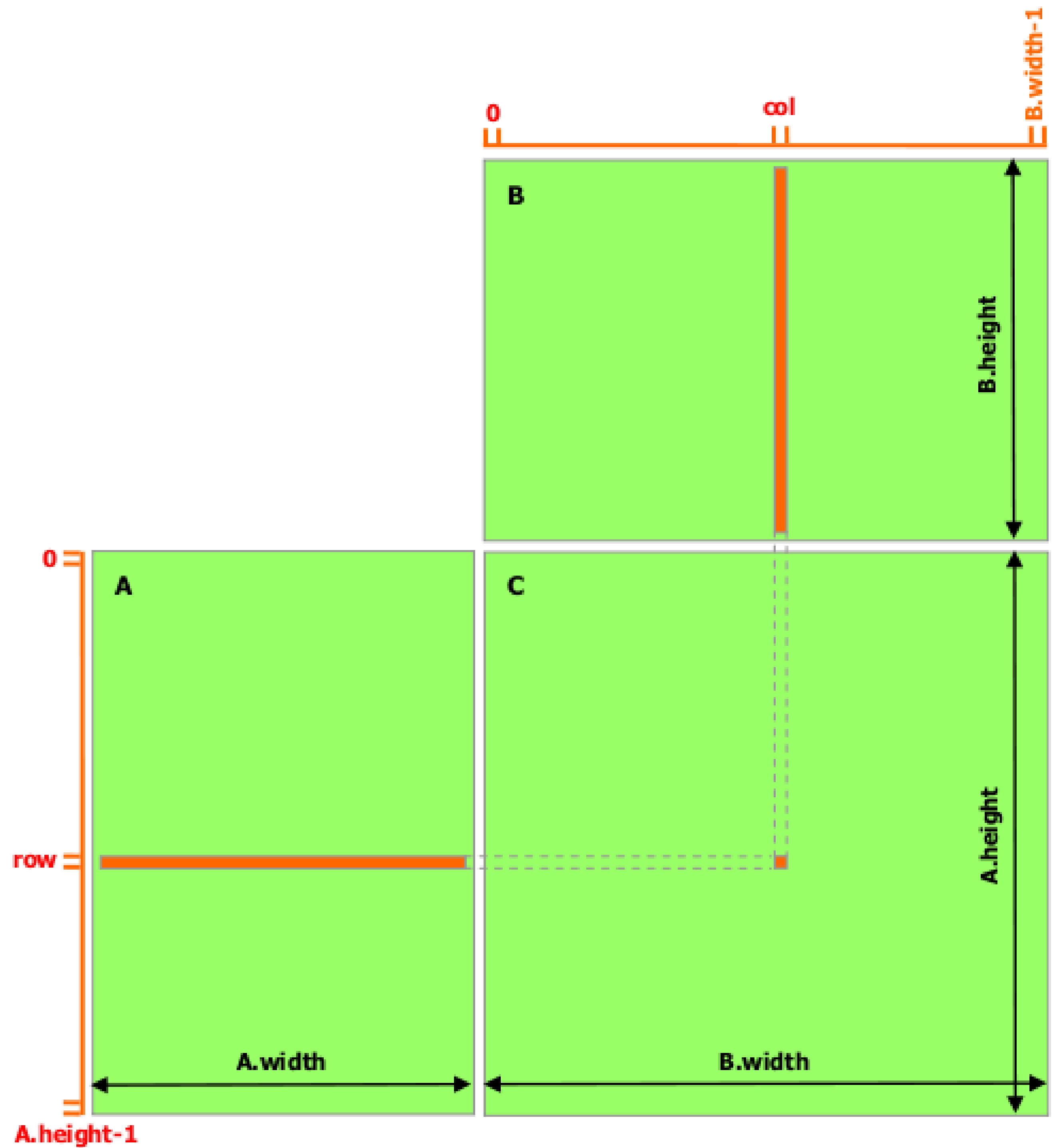
Dzielona pamięć urządzenia:

- użycie pamięci dzielonej urządzenia jest w kodzie źródłowym manifestowane użyciem specyfikatora **`__shared__`**
- zmienna zadeklarowana jako `__shared__` jest dostępna we wszystkich wątkach i w każdym z nich oznacza **ten sam byt**
- pamięć dzielona jest (według deklaracji producenta) znacząco szybsza (i w odczycie i w zapisie) niż pamięć globalna
- daje to szansę przyspieszania obliczeń wymagających intensywnego dostępu do pamięci

Dzielona pamięć urządzenia:

- zilustrujemy to przykładem opartym o rutynowe mnożenie dwóch macierzy
- każdy wątek ma za zadanie odczytać jeden wiersz macierzy A i jedną kolumnę macierzy B w celu wyliczenia odpowiedniego elementu macierzy wynikowej C

Schemat pracy algorytmu:



Źródło: *NVIDIA CUDA Programming Guide Version 3.0*

Mnożenie macierzy – implementacja klasyczna

część 1/4

```
// Macierze są pamiętane wierszami, a więc:
// M(row, col) = *(M.elements + row * M.width + col)

typedef struct {
    int    width;
    int    height;
    float *elements;
} Matrix;

// definiujemy rozmiar bloku wątków:
#define    BLOCK_SIZE    16

// prototyp funkcji mnożącej (kernela)
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);


// Zakładamy (dla uproszczenia rozważań), że wymiary macierzy są
// całkowitymi wielokrotnościami wartości BLOCK_SIZE
```

Mnożenie macierzy – implementacja klasyczna
część 2/4

```
// Funkcja mnożąca

void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // kopiujemy macierze A i B to globalnej pamięci urządzenia
    // najpierw A

    Matrix d_A;
    d_A.width  = A.width;
    d_A.height = A.height;

    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc((void **)&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

    // potem B
    Matrix d_B;
    d_B.width  = B.width;
    d_B.height = B.height;

    size = B.width * B.height * sizeof(float);
    cudaMalloc((void **)&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,cudaMemcpyHostToDevice);
    :
    :
```

Mnożenie macierzy – implementacja klasyczna

część 3/4

```
    :
    :
    // przydzielamy macierz C w globalnej pamięci urządzenia
    Matrix d_C;
    d_C.width  = C.width;
    d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc((void **)&d_C.elements, size);

    // preparujemy środowisko i wywołujemy kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // odbieramy obliczoną macierz C z pamięci globalnej urządzenia
    cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);


    // zwalniamy pamięć
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

Mnożenie macierzy – implementacja klasyczna

część 4/4

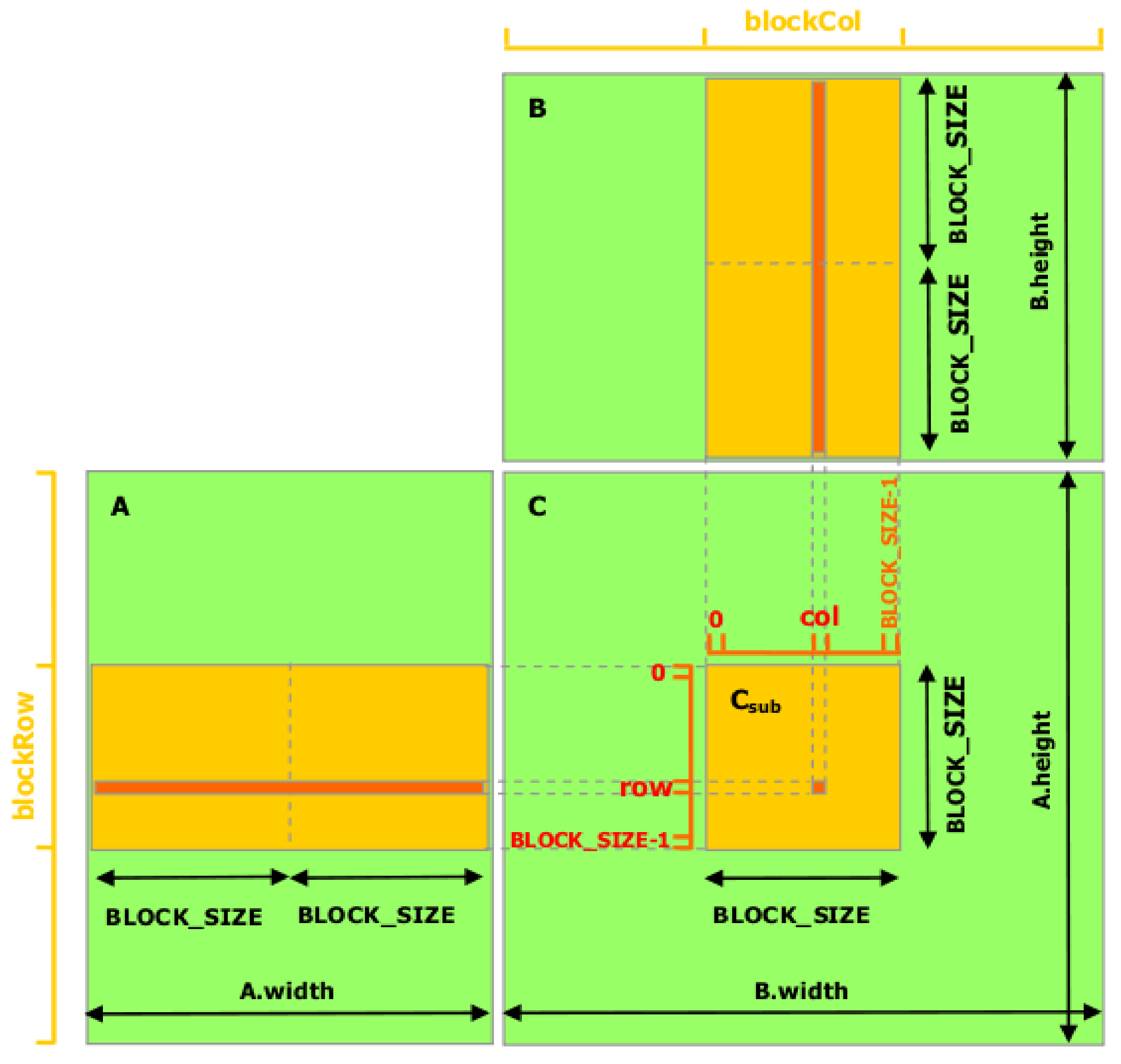
```
// kernel odpowiedzialny za wymnożenie macierzy
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{

    // każdy wątek oblicza jeden element macierzy C
    // akumulując wynik w zmiennej Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue +=    A.elements[row * A.width + e]
                   * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

A teraz to samo, ale z użyciem pamięci dzielonej...

- zakładamy, że każdy blok wątków jest odpowiedzialny za wyliczenie kwadratowej podmacierzy macierzy C, którą nazwiemy tu C_{sub}
- w ten sposób każde wydzielone C_{sub} jest iloczynem dwóch prostokątnych podmacierzy:
  - macierzy A o wymiarach (A.width, n*block_size), która ma te same numery wierszy co C_{sub}
  - macierzy B o wymiarach (n*block_size, B.heigth), która ma te same numery kolumn co C_{sub}
- każdą z powyższych prostokątnych macierzy dzieli się na odpowiednią liczbę kwadratowych podpodmacierzy o boku BLOCK_SIZE

Schemat pracy algorytmu:



Źródło: *NVIDIA CUDA Programming Guide Version 3.0*

- w ten sposób każdy z elementów podmacierzy C_{sub} wyliczany jako suma iloczynów wyróżnionych podpodmacierzy
- każdy z iloczynów jest obliczany drogą wstępnego załadowania odpowiednich fragmentów macierzy A i B z pamięci globalnej do pamięci dzielonej
- zakładamy, że w pierwszej kolejności każdy wątek ładuje swoje dane, a następnie wylicza cząstkową wartość iloczynu
- każdy wątek akumuluje wartość cząstkową, aby na końcu zapisać ją w pamięci globalnej
- grupując obliczenia w powyższy sposób zmniejszamy transfer z pamięci globalnej, ponieważ macierz A jest czytana jedynie (B.width / block_size) razy, a macierz B (A.height / block_size) razy

Przy okazji zawieramy znajomość z nowym specyfikatorem:

`__device__`

- poprzedza deklaracje funkcji
- funkcja zadeklarowana w taki sposób przeznaczona jest do **wykonania na urządzeniu** (tylko i wyłącznie)
- funkcję taką można wywołać z kodu **wykonywanego na urządzeniu** (tylko i wyłącznie)

Mnożenie macierzy – implementacja z pamięcią dzieloną
część 1/6

```
// Macierze są pamiętane wierszami, a więc:
// M(row, col) = *(M.elements + row * M.width + col)

typedef struct {
    int     width;
    int     height;
    int     stride;
    float   *elements;
} Matrix;

// funkcja do odczytywania wartości elementu wskazanej macierzy
__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}

// funkcja do zapisywania wartości elementu wskazanej macierzy
__device__ void SetElement(Matrix A, int row, int col, float value)
{
    A.elements[row * A.stride + col] = value;
}
```

Mnożenie macierzy – implementacja z pamięcią dzieloną
część 2/6

```
// wykreowanie opisu podmacierzy o rozmiarze BLOCK_SIZExBLOCK_SIZE, która
// ulokowana jest col podmacierzy w prawo i row podmacierzy w dół
// licząc od lewego wierzchołka danej macierzy

__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;

    Asub.width     = BLOCK_SIZE;
    Asub.height    = BLOCK_SIZE;
    Asub.stride    = A.stride;
    Asub.elements  = &A.elements[A.stride * BLOCK_SIZE * row+ BLOCK_SIZE * col];

    return Asub;
}
```

Mnożenie macierzy – implementacja z pamięcią dzieloną

część 3/6

```
#define BLOCK_SIZE 16


__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

void MatMul(const Matrix A, const Matrix B, Matrix C)
{

    Matrix d_A;
    d_A.width = d_A.stride = A.width;
    d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc((void **)&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,cudaMemcpyHostToDevice);

    Matrix d_B;
    d_B.width = d_B.stride = B.width;
    d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc((void **)&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,cudaMemcpyHostToDevice);
    :
    :
```

Mnożenie macierzy – implementacja z pamięcią dzieloną
część 4/6

```
    :
    :
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc((void **)&d_C.elements, size);


    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

Mnożenie macierzy – implementacja z pamięcią dzieloną
część 5/6

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // ustalenie numeru wiersza i kolumny wewnątrz bloku
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // każdy blok oblicza jedną podmacierz Csub macierzy C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    // każdy wątek oblicza jeden element Csub akumulując wynik w Cvalue
    float Cvalue = 0;

    // ustalenie numeru wiersza i kolumny wewnątrz wątku
    int row = threadIdx.y;
    int col = threadIdx.x;
    :
    :
```

Mnożenie macierzy – implementacja z pamięcią dzieloną
część 6/6

```
    // iterujemy wszystkie podmacierze A i B, które
    // są potrzebne do obliczenia Csub – mnożymy ze sobą każdą parę
    // podmacierzy i akumulujemy wynik
    for(int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        // kreujemy podmacierz Asub macierzy A
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        // kreujemy podmacierz Bsub macierzy B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);
        // deklarujemy obszar pamięci dzielonej dla podmacierzy Asub i Bsub
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        // załaduj Asub i Bsub z pamięci globalnej do dzielonej
        // (każdy wątek ładuje jeden element z każdej podmacierzy)
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);
        // poczekajmy, aż wszystkie dane zostaną skopiowane
        __syncthreads();
        // mnożymy Asub i Bsub
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];
        // poczekajmy, aż obliczenia zostaną zakończone zanim zabierzemy
        // się za przetwarzanie następnej podmacierzy
        __syncthreads();
    }
    // odsyłamy obliczone Cvalue do pamięci urządzenia
    SetElement(Csub, row, col, Cvalue);
}
```

Wiele urządzeń CUDA w jednym systemie hosta:

- jeden host może zarządzać więcej niż jednym urządzeniem CUDA
- urządzenia te mogą być wyliczane i odpytywane o właściwości
- jedno z nich może być wybierane do wykonywania kodu kernela
- więcej niż jeden wątek hosta może wykonywać kod kernela na jednym urządzeniu CUDA
- jeden wątek hosta może wykonywać kod kernela tylko na jednym urządzeniu
- zasoby wykreowane i używane w jednym wątku hosta nie mogą być używane w innych wątkach

Wiele urządzeń CUDA w jednym systemie hosta:

- domyślnie jako urządzenie aktywne wybierane jest urządzenie o numerze 0 (pierwsze rozpoznane w systemie)
- ustalenie innego aktywnego urządzenia jest możliwy poprzez wywołanie funkcji `cudaSetDevice()`
- zmiana aktywnego urządzenia jest możliwa pod warunkiem wykonania funkcji `cudaThreadExit()`, które sygnalizuje zakończenie „współpracy” z dotychczasowym aktywnym urządzeniem

CUDA API – zarządzanie urządzeniami:

wybór aktywnego urządzenia

```
cudaError_t cudaSetDevice   (int device)
```

parametry:

- `device`: numer urządzenia CUDA wybieranego do wykonywania kodu kerneli (uwaga – urządzenia numeruje się od zera!)

wynik:

- `cudaSuccess` – wykonanie poprawne
- `cudaErrorInvalidDevice` – niepoprawne urządzenie
  `cudaErrorSetOnActiveProcess` - próba użycia funkcji w sytuacji, gdy urządzenie jest już wybrane i wykonuje kod

CUDA API – zarządzanie urządzeniami:

zwolnienie wszystkich zasobów na aktywnym urządzeniu

```
cudaError_t cudaThreadExit (void)
```

wynik:

- `cudaSuccess` – wykonanie poprawne