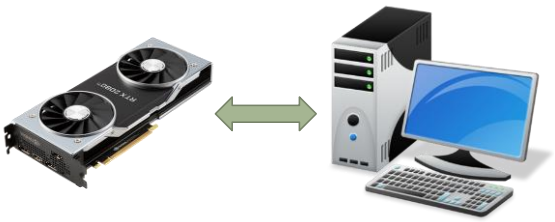
PROGRAMOWANIE WEKTOROWE I RÓWNOLEGŁE

Wykład dla kierunku:
Matematyka stosowana
i technologie
informatyczne

WSPÓŁDZIAŁANIE GOSPODARZA Z URZĄDZENIAMI

DEVICE & HOST



JAK TO DZIAŁA?

Gospodarz (Host / komputer) przesyła do pamięci urządzenia (device) program i dane przy użyciu funkcji CUDA API.

Urządzenie wykonuje zadanie i ewentualnie zwraca informację o błędach.

Host pobiera wyniki i zwalnia urządzenie.

PROGRAM W CUDA C

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size);

__global__ void addKernel(int *c, const int *a, const int *b)
{
int main()
{
// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size)
{
```


cudaError_t

Typ określający błąd w wywołaniu funkcji API CUDA.

Definicja:

typedef enumcudaError cudaError_t

cudaError_t

Wartości:

cudaSuccess = 0

The API call returned with no errors. In the case of query calls, this also means that the operation being queried is complete.

cudaErrorInvalidValue = 1

This indicates that one or more of the parameters passed to the API call is not within an acceptable range of values.

cudaErrorMemoryAllocation = 2

The API call failed because it was unable to allocate enough memory to perform the requested operation.

cudaErrorInitializationError = 3

The API call failed because the CUDA driver and runtime could not be initialized.

cudaErrorCudartUnloading = 4

This indicates that a CUDA Runtime API call cannot be executed because it is being called during process shutdown, at a point in time after CUDA driver has been unloaded.

.....

Jest jeszcze kilkadziesiąt innych błędów!

Specyfikatory przestrzeni wykonywania funkcji

Specyfikatory przestrzeni wykonywania funkcji (kwalifikatory) wskazują:

- czy funkcja jest wykonywana na hoście, czy na urządzeniu
- czy można ją wywoływać z hosta, czy z urządzenia.

__global__

Specyfikator przestrzeni wykonawczej __global__ deklaruje funkcję jako jądro (kernel). Taka funkcja:

- jest wykonywana na urządzeniu,
- ma możliwość wywołania z hosta,
- jest wywoływalna z urządzenia w przypadku urządzeń o możliwościach obliczeniowych 5.0 lub nowszych.

Funkcja __global__ zwraca typ void i nie może należeć do żadnej klasy.

Każde wywołanie funkcji __global__ musi określać jej konfigurację wykonywania zgodnie z opisem w konfiguracji wykonywania.

Wywołanie funkcji __global__ jest asynchroniczne, co oznacza, że następuje powrót przed zakończeniem wykonywania przez urządzenie.

__device__

Specyfikator przestrzeni wykonawczej __device__ deklaruje funkcję, która jest:

- wykonywana na urządzeniu,
- wywoływalna tylko z urządzenia.

Specyfikatorów przestrzeni wykonawczej __global__ i __device__ nie można używać razem.

__host__

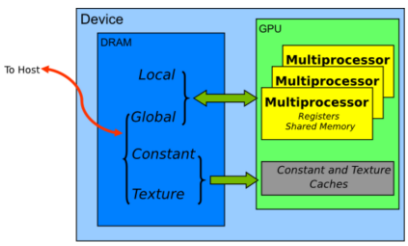
Specyfikator przestrzeni wykonawczej __host__ deklaruje funkcję, która jest:

- wykonywana na hoście,
- wywoływalna tylko z hosta.

Równoważne jest zadeklarowanie funkcji tylko ze specyfikatorem przestrzeni wykonawczej __host__ lub zadeklarowanie jej bez specyfikatora przestrzeni wykonawczej __host__, __device__ lub __global__; w obu przypadkach funkcja jest kompilowana tylko dla hosta.

Specyfikatorów przestrzeni wykonawczej __global__ i __host__ nie można używać razem.

ODWOŁANIA DO PAMIĘCI URZĄDZENIA



FUNKCJA JĄDRA - KERNEL

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size);

__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

int main()
{

// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size)
{
```


uint3 i dim3

uint3 jest typem wektorowym liczb całkowitych zawierającym 3 pola x, y i z.

dim3 jest typem wektorowym opartym na uint3, jest używany do określania wymiarów. Podczas definiowania zmiennej typu dim3 każdy komponent, który nie został określony, jest inicjowany na 1.

ZMIENNE WBUDOWANE

Zmienne wbudowane określają wymiary siatki i bloków oraz indeksy bloków i wątków.

Obowiązują one tylko w ramach funkcji wykonywanych na urządzeniu!

gridDim, blockIdx, blockDim, threadIdx i warpSize

gridDim jest typu dim3 i zawiera wymiary siatki.

blockIdx jest typu uint3 i zawiera indeks bloku w siatce.

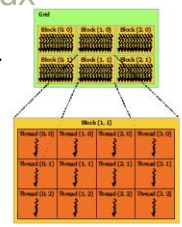
blockDim jest typu dim3 i zawiera wymiary bloku.

threadIdx jest typu uint3 i zawiera indeks wątku w bloku.

warpSize jest typu int i zawiera rozmiar osnowy podany w wątkach

gridDim, blockIdx, blockDim, threadIdx

Dwuwymiarowa siatka i bloki.




```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```


Funkcja main

```
int main()
{
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };

    // Add vectors in parallel.
    cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addWithCuda failed!");
        return 1;
    }

    printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",
        c[0], c[1], c[2], c[3], c[4]);

    // cudaDeviceReset must be called before exiting in order for profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
        return 1;
    }
```

Funkcja cudaDeviceReset

cudaError_t cudaDeviceReset (void)

Niszczy wszystkie alokacje i resetuje wszystkie stany na bieżącym urządzeniu w bieżącym procesie.

Jawnie niszczy i czyści wszystkie zasoby powiązane z bieżącym urządzeniem w bieżącym procesie. Obowiązkiem obiektu wywołującego jest upewnienie się, że zasoby nie są dostępne ani przekazywane w kolejnych wywołaniach interfejsu API, co spowoduje niezdefiniowane zachowanie. Zasoby te obejmują typy CUDA, takie jak:

cudaStream_t, cudaEvent_t, cudaArray_t, cudaMipmappedArray_t, cudaTextureObject_t, cudaSurfaceObject_t, textureReference, surfaceReference, cudaExternalMemory_t, cudaExternalSemaphore_t i cudaGraphicsResource_t.

Każde kolejne wywołanie interfejsu API do tego urządzenia spowoduje ponowną inicjalizację urządzenia. Należy pamiętać, że ta funkcja spowoduje natychmiastowe zresetowanie urządzenia. Obowiązkiem wywołującego jest upewnienie się, że żadne inne wątki hosta z procesu nie mają dostępu do urządzenia podczas wywoływania tej funkcji.

Funkcja addWithCuda - 1

```
// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size)
{
    int *dev_a = 0;
    int *dev_b = 0;
    int *dev_c = 0;
    cudaError_t cudaStatus;

    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU installed?");
        goto Error;
    }

    // Allocate GPU buffers for three vectors (two input, one output)    .
    cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }
```


Funkcja cudaSetDevice

cudaError_t cudaSetDevice (int device)

Ustawia urządzenie, które ma być używane do wykonywania GPU.

Parametry: device — urządzenie, na którym aktywny wątek hosta powinien wykonać kod urządzenia.

Zwraca: cudaSuccess, cudaErrorInvalidDevice, cudaErrorDeviceUnavailable,

Ustawia urządzenie jako bieżące urządzenie dla wywołującego wątku hosta. Prawidłowe identyfikatory urządzeń to 0 do (cudaGetDeviceCount() - 1). Każda pamięć urządzenia przydzielona następnie z tego wątku hosta za pomocą cudaMalloc(), cudaMallocPitch() lub cudaMallocArray() będzie fizycznie rezydować na urządzeniu. Każda pamięć hosta przydzielona z tego wątku hosta za pomocą cudaMallocHost() lub cudaHostAlloc() lub cudaHostRegister() będzie miała swój czas życia powiązany z urządzeniem. Wszelkie strumienie lub zdarzenia utworzone z tego wątku hosta zostaną powiązane z urządzeniem. Wszelkie jądra uruchomione z tego wątku hosta przy użyciu operatora <<<>>> lubcudaLaunchKernel() zostaną wykonane na urządzeniu. To wywołanie może być wykonane z dowolnego wątku hosta, do dowolnego urządzenia i w dowolnym momencie. Ta funkcja nie wykona synchronizacji z poprzednim lub nowym urządzeniem i powinna zająć dużo czasu tylko podczas inicjowania stanu kontekstu środowiska uruchomieniowego. To wywołanie powiąże kontekst podstawowy określonego urządzenia z wątkiem wywołującym, a wszystkie kolejne przydziały pamięci, tworzenie strumieni i zdarzeń oraz uruchomienia jądra zostaną powiązane z kontekstem podstawowym. Ta funkcja natychmiast zainicjuje również stan środowiska uruchomieniowego w kontekście podstawowym, a kontekst natychmiast stanie się aktualny na urządzeniu. Ta funkcja zwróci błąd, jeśli urządzenie jest w trybie cudaComputeModeExclusiveProcess i jest zajęte przez inny proces lub jeśli urządzenie jest w trybie cudaComputeModeProhibited. Nie jest wymagane wywołanie cudaInitDevice przed użyciem tej funkcji

Funkcja cudaMalloc

cudaError_t cudaMalloc (void **devPtr, size_t size)

Przydziela pamięć na urządzeniu.

Parametry:

devPtr - Wskaźnik do przydzielonej pamięci urządzenia

size - Żądany rozmiar alokacji w bajtach.

Zwraca: cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation.

Przydziela size bajtów pamięci liniowej na urządzeniu i zwraca w *devPtr wskaźnik do przydzielonej pamięci.

Przydzielona pamięć jest odpowiednio dopasowana do każdego rodzaju zmiennej. Pamięć nie jest wyczyszczona. cudaMalloc() zwraca cudaErrorMemoryAllocation w przypadku niepowodzenia.

Wersja dla urządzenia funkcji cudaFree nie może być używana z *devPtr przydzielony m za pomocą interfejsu API hosta i odwrotnie.

Funkcja addWithCuda - 2

```
    // Copy input vectors from host memory to GPU buffers.
    cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
```


Funkcja cudaMemcpy

cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, cudaMemcpyKind kind)

Kopiuje dane między hostem a urządzeniem.

Parametry: dst - Adres pamięci
Src - Adres pamięci źródłowej
Count - Rozmiar w bajtach do skopiowania
Kind – Kierunek przesyłu

Zwraca cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection

Kopiuje count bajtów z obszaru pamięci wskazywanego przez src do obszaru pamięci wskazywanego przez dst, gdzie kind określa kierunek kopiowania i musi być jednym z cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice lub cudaMemcpyDefault. Zalecane jest przekazanie cudaMemcpyDefault, w którym to przypadku typ transferu jest wywnioskowany z wartości wskaźnika. Jednak cudaMemcpyDefault jest dozwolone tylko w systemach obsługujących ujednolicone adresowanie wirtualne.

Wywołanie cudaMemcpy() ze wskaźnikami dst i src, które nie pasują do kierunku kopiowania, skutkuje niezdefiniowanym zachowaniem.

Funkcja addWithCuda - 3

```
// Launch a kernel on the GPU with one thread for each element.
addKernel<<<1, size>>>(dev_c, dev_a, dev_b);

// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
    goto Error;
}

// cudaDeviceSynchronize waits for the kernel to finish, and returns
// any errors encountered during the launch.
cudaStatus = cudaDeviceSynchronize();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaStatus);
    goto Error;
}
```


Operator <<< >>>

Jądro jest definiowane przy użyciu specyfikatora deklaracji __global__, a liczba wątków CUDA, które wykonują to jądro dla danego wywołania jądra, jest określana przy użyciu nowej składni konfiguracji wykonania <<<...>>> . Każdy wątek wykonujący jądro otrzymuje unikalny identyfikator wątku, który jest dostępny w jądrze za pośrednictwem wbudowanych zmiennych.

W <<<...>>> podaje się dwa parametry (dim3):

- Rozmiar siatki w blokach
- Liczbę wątków w bloku.

Operator <<< >>>

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```


Operator <<< >>>

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```


Operator <<< >>>

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```


Funkcja cudaGetLastError

cudaError_t cudaGetLastError(void)

Zwraca ostatni błąd wywołania w czasie wykonywania.

Zwraca ostatni błąd, który został wygenerowany przez dowolne wywołanie środowiska wykonawczego w tej samej instancji biblioteki CUDA Runtime w wątku hosta i resetuje go do cudaSuccess.

Uwaga: W przypadku korzystania z biblioteki, która statycznie łączy środowisko wykonawcze CUDA, w aplikacji może znajdować się wiele wystąpień biblioteki środowiska wykonawczego CUDA.

Funkcja cudaDeviceSynchronize

cudaError_t cudaDeviceSynchronize (void)

Czeka na zakończenie działania urządzenia obliczeniowego.

Zwraca: cudaSukces

Blokuje, dopóki urządzenie nie wykona wszystkich żądanych zadań. cudaDeviceSynchronize() zwraca błąd, jeśli jedno z poprzednich zadań nie powiodło się. Jeśli dla tego urządzenia ustawiono flagę cudaDeviceScheduleBlockingSync, wątek hosta będzie blokowany, dopóki urządzenie nie zakończy swojej pracy.

Funkcja addWithCuda - 4

```
108     // Copy output vector from GPU buffer to host memory.
109     cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
110     if (cudaStatus != cudaSuccess) {
111         fprintf(stderr, "cudaMemcpy failed!");
112         goto Error;
113     }
114
115 Error:
116     cudaFree(dev_c);
117     cudaFree(dev_a);
118     cudaFree(dev_b);
119
120     return cudaStatus;
121 }
```


Funkcja cudaFree

cudaError_t cudaFree (void*devPtr)

Zwalnia pamięć na urządzeniu.

Parametry: devPtr - Wskaźnik do zwalnianej pamięci urządzenia.

Zwraca: cudaSuccess, cudaErrorInvalidValue.

Zwalnia miejsce w pamięci wskazywane przezdevPtr, które musiało zostać zwrócone przez poprzednie wywołanie— cudaMalloc().

Uwaga — Wywołujący muszą upewnić się, że wszystkie dostępy do wskaźnika zostały zakończone przed wywołaniemcudaFree. Jeśli cudaFree(devPtr) była już wcześniej wywoływana, zwracany jest błąd. Jeśli devPtr wynosi 0, żadna operacja nie jest wykonywana. cudaFree() zwraca wartość cudaErrorValue w przypadku niepowodzenia.