Uniwersytet Wrocławski

Programowanie Równoległe
Wykład, 07.01.2014

CUDA praktycznie 1

Maciej Matyka
Instytut Fizyki Teoretycznej

Motywacja

- CPU vs GPU (*anims*)

Plan CUDA w praktyce

- Wykład 1: CUDA w praktyce
- Wykład 2: Cuda + Opengl
- Wykład 3: Thrust

Kompilacja LINUX

- Korzystamy z pracowni 426
- Komputery mają zainstalowane karty nVidia GeForce

- Środowisko CUDA jest zainstalowane

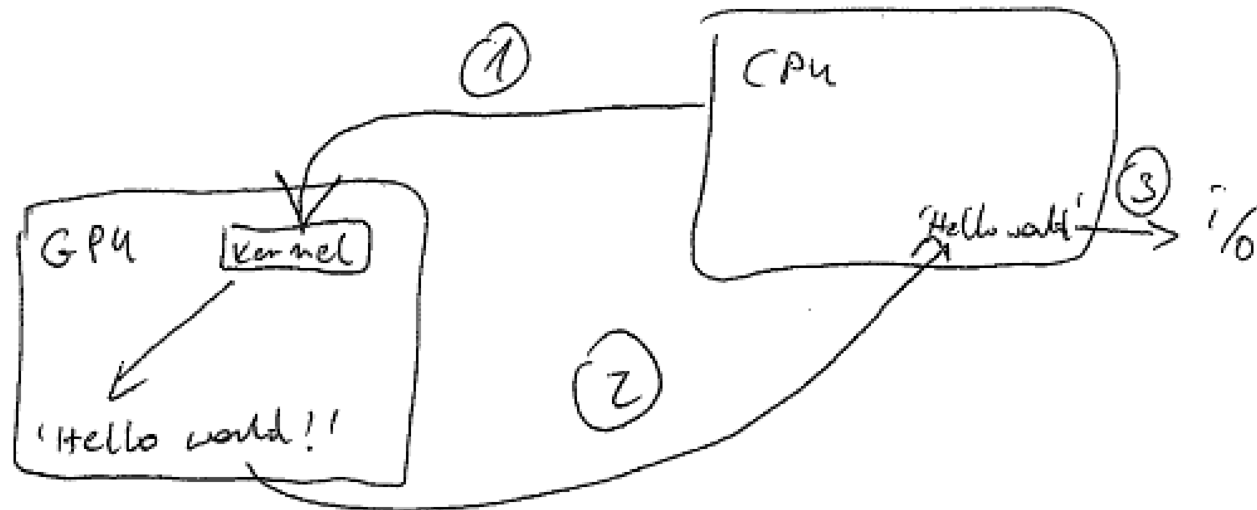
- Kompilacja (plik program.cu)

```
> nvcc program.cu
> ./a.out
```

- Kompilacja CUDA + GLUT (OpenGL)

```
> nvcc main.cpp kernels.cu -lglut –lGLU
> ./a.out
```

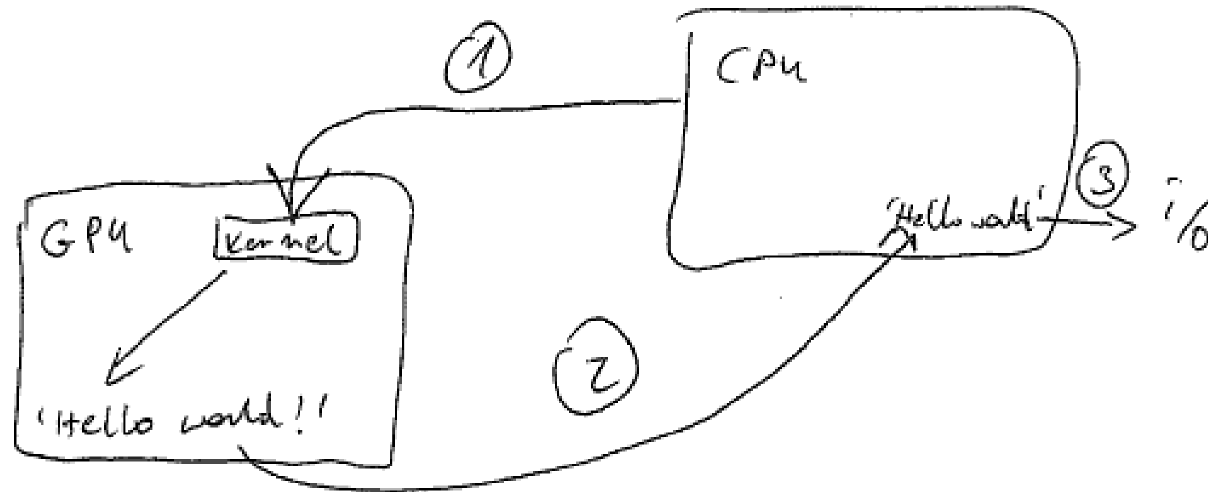
- Pierwszy prosty program w CUDA



1. CPU wywołuje funkcję na GPU
   - Funkcja wstawia do pamięci GPU ciąg „Hello world!”
2. Kopiujemy dane z GPU do pamięci CPU
3. Dane wypisujemy przy pomocy funkcji i/o (cout etc.)

Hello World

- Pierwszy prosty program w CUDA



1. CPU wywołuje funkcję na GPU
   – Funkcja wstawia do pamięci GPU ciąg „Hello world!”
2. Kopiujemy dane z GPU do pamięci CPU
3. Dane wypisujemy przy pomocy funkcji i/o (cout etc.)

```
cudaError_t cudaMemcpyFromSymbol ( void *                    dst,
                                   const char *              symbol,
                                   size_t                    count,
                                   size_t                    offset = 0,
                                   enum cudaMemcpyKind       kind = cudaMemcpyDeviceToHost
                                 )
```

Copies `count` bytes from the memory area pointed to by `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either **cudaMemcpyDeviceToHost** or **cudaMemcpyDeviceToDevice**.

Parameters:

dst	- Destination memory address
symbol	- Symbol source from device
count	- Size in bytes to copy
offset	- Offset from start of symbol in bytes
kind	- Type of transfer

Returns:

cudaSuccess, **cudaErrorInvalidValue**, **cudaErrorInvalidSymbol**, **cudaErrorInvalidDevicePointer**, **cudaErrorInvalidMemcpyDirection**

Note:

The `symbol` paramater may also be a character string, naming a variable that resides in global or constant memory space, however this usage is deprecated as of CUDA 4.1.

Note that this function may also return error codes from previous, asynchronous launches.

This function exhibits **synchronous** behavior for most use cases.

See also:

cudaMemcpy, **cudaMemcpy2D**, **cudaMemcpyToArray**, **cudaMemcpy2DToArray**, **cudaMemcpyFromArray**, **cudaMemcpy2DFromArray**, **cudaMemcpyArrayToArray**, **cudaMemcpy2DArrayToArray**, **cudaMemcpyToSymbol**, **cudaMemcpyAsync**, **cudaMemcpy2DAsync**, **cudaMemcpyToArrayAsync**, **cudaMemcpy2DToArrayAsync**, **cudaMemcpyFromArrayAsync**, **cudaMemcpy2DFromArrayAsync**, **cudaMemcpyToSymbolAsync**, **cudaMemcpyFromSymbolAsync**

Hello World

```
#include <stdio.h>
#include <cuda.h>

__device__ char napis_device[14];
char napis_host[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    …
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();




    cudaMemcpyFromSymbol (napis_host,
     napis_device, sizeof(char)*14, 0,
    cudaMemcpyDeviceToHost);

    printf("%s",napis_host);
}
```

Hello World

```
#include <stdio.h>
#include <cuda.h>

__device__ char napis_device[14];
char napis_host[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    …
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();




    cudaMemcpyFromSymbol (napis_host,
     napis_device, sizeof(char)*14, 0,
    cudaMemcpyDeviceToHost);

    printf("%s",napis_host);
}
```

Hello World

```
#include <stdio.h>
#include <cuda.h>

__device__ char napis_device[14];
char napis_host[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    …
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();




    cudaMemcpyFromSymbol (napis_host,
     napis_device, sizeof(char)*14, 0,
    cudaMemcpyDeviceToHost);

    printf("%s",napis_host);
}
```

Hello World

```
#include <stdio.h>
#include <cuda.h>

__device__ char napis_device[14];
char napis_host[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    …
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();




    cudaMemcpyFromSymbol (napis_host,
     napis_device, sizeof(char)*14, 0,
    cudaMemcpyDeviceToHost);

    printf("%s",napis_host);
}
```

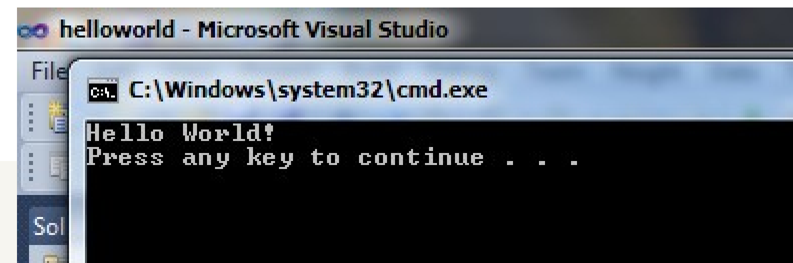
Hello World

```
#include <stdio.h>
#include <cuda.h>

__device__ char napis_device[14];
char napis_host[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    …
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();




    cudaMemcpyFromSymbol (napis_host,
     napis_device, sizeof(char)*14, 0,
    cudaMemcpyDeviceToHost);

    printf("%s",napis_host);
}
```

Hello World

```
#include <stdio.h>
#include <cuda.h>

__device__ char napis_device[14];
char napis_host[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    …
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();




    cudaMemcpyFromSymbol (napis_host,
     napis_device, sizeof(char)*14, 0,
    cudaMemcpyDeviceToHost);

    printf("%s",napis_host);
}
```

Hello World




```
#include <stdio.h>
#include <cuda.h>

__device__ char napis_device[14];
char napis_host[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    …
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();




    cudaMemcpyFromSymbol (napis_host,
     napis_device, sizeof(char)*14, 0,
    cudaMemcpyDeviceToHost);

    printf("%s",napis_host);
}
```

Deklaracja funkcji na GPU

- Funkcja uruchamiana na GPU to kernel (jądro)
- Zmienne na GPU – przedrostek **__device__**
- Preambuła jądra - przedrostek **__global__**

```
char napis_device[14];

void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    …
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}
```

```
__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    …
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}
```

Wywołanie funkcji na GPU

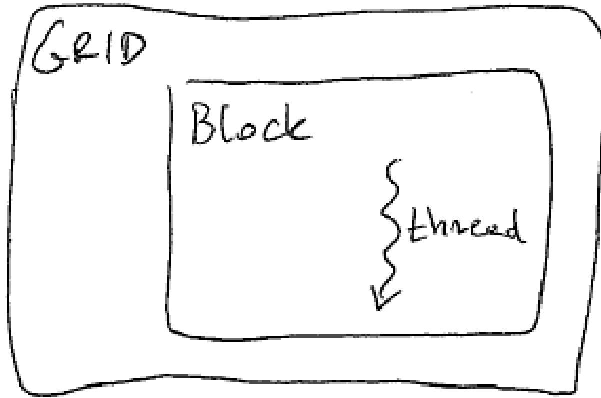
- Wywołanie funkcji GPU:

```
funkcja <<< numBlocks, threadsPerBlock >>> ( parametry );
```

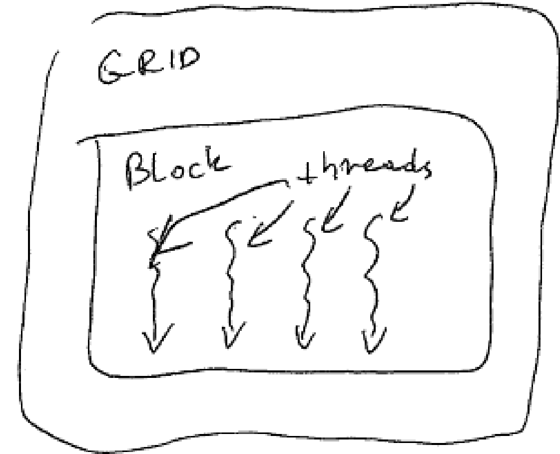
- numBlocks – liczba bloków w macierzy wątków
- threadsPerBlock – liczba wątków na blok

W naszym przykładzie <<<1,1>>> oznaczało uruchomienie jądra na jednym wątku który zawierał się w jednym jedynym bloku macierzy wątków.

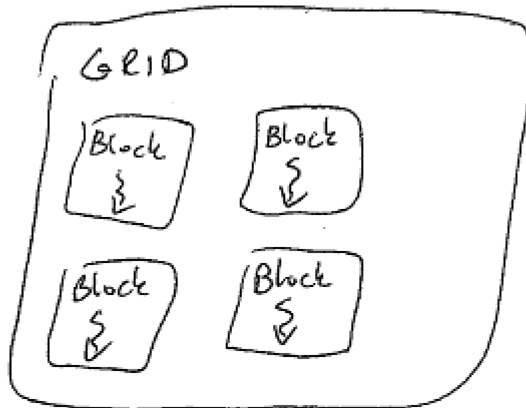
numBlocks, threadsPerBlock



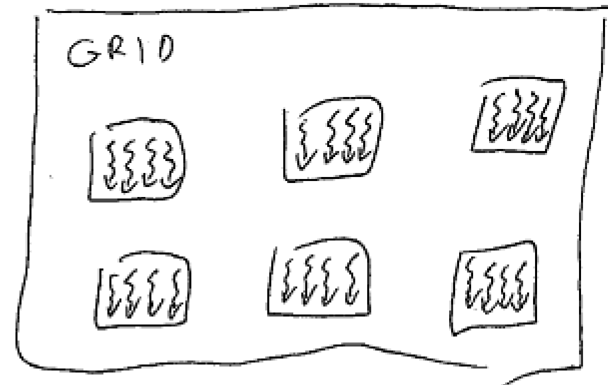
<<< 1 , 1 >>>



<<< 1 , 4 >>>



<<< 4 , 1 >>>



<<< dim3(3,2,1) , 4 >>>

A tak to widzi nVidia

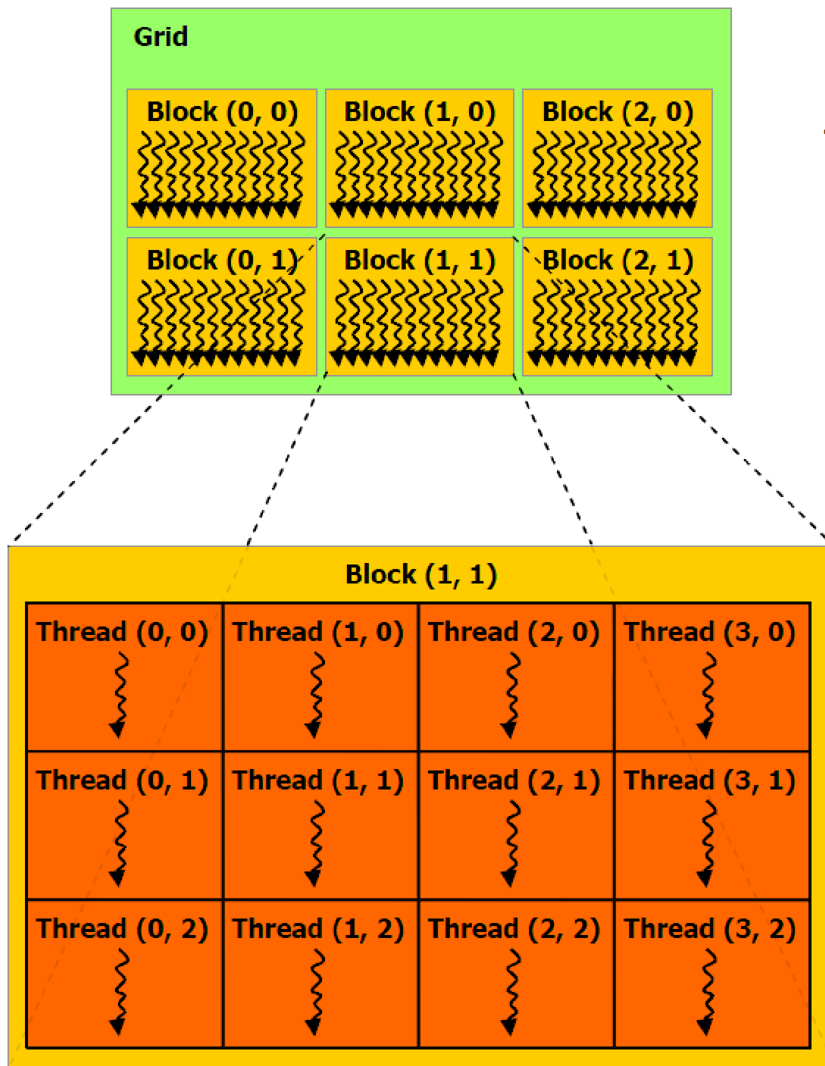
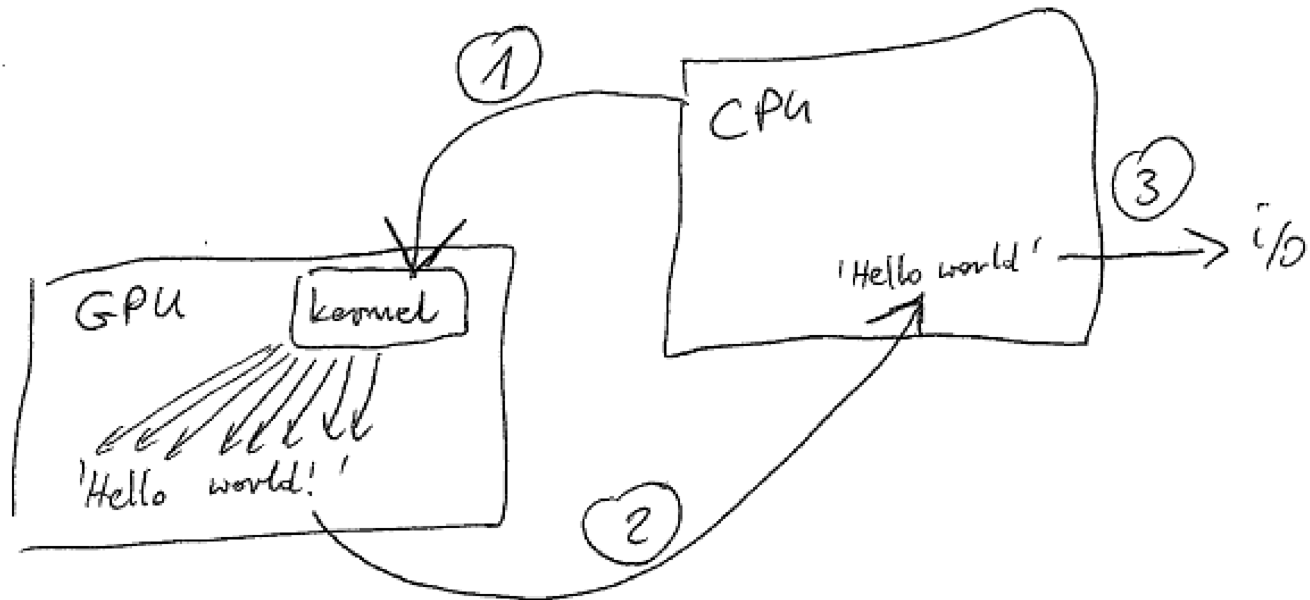


Figure 2-1. Grid of Thread Blocks

CUDA Programming Guide 3.2

<<< dim3(3,2,1) , dim3(4,3,1) >>>

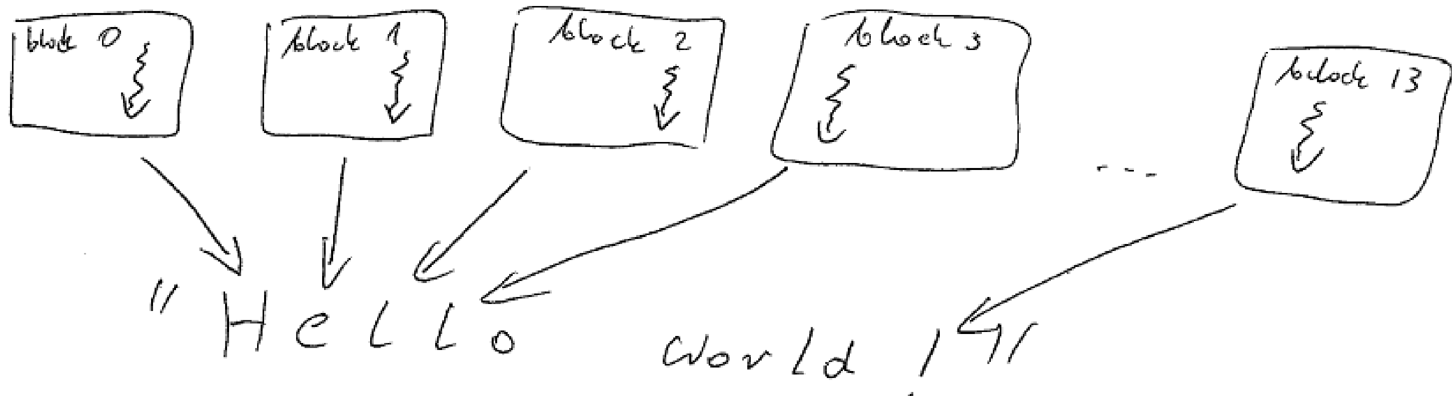
Hello World wielowątkowo



1. CPU wywołuje funkcję na GPU
   - Funkcja wstawia do pamięci GPU ciąg „Hello world!”
   - jedna litera na jeden wątek!

Hello World wielowątkowo

- jedna litera na jeden wątek



- stwórzmy 14 bloków po 1 wątku
- Każdy wątek **zna** swój numer i numer bloku
- Numer bloku -> miejsce w tablicy do skopiowania

Hello World wielowątkowo

- Jądro dla hello world w wersji wielowątkowej:

```
__constant__ __device__ char hw[] = "Hello World!\n";
__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    int idx = blockIdx.x;
    napis_device[idx] = hw[idx];
}

…

helloWorldOnDevice <<< 14, 1 >>> ();
```

- idx
  - zawiera numer bloku w którym znajduje się wątek
  - mapowanie blok/wątek -> fragment problemu

  - wątek z idx-ego bloku kopiuje idx-ą literę napisu
  - Kopiowanie GPU-GPU
  - (bez sensu, ale chodzi nam o najprostszy problem)

Hello World wielowątkowo

- Jądro dla hello world w wersji wielowątkowej:

```
__constant__ __device__ char hw[] = "Hello World!\n";
__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    int idx = blockIdx.x;
    napis_device[idx] = hw[idx];
}

…

helloWorldOnDevice <<< 14, 1 >>> ();
```

- idx
  - zawiera numer bloku w którym znajduje się wątek
  - mapowanie blok/wątek -> fragment problemu

  - wątek z idx-ego bloku kopiuje idx-ą literę napisu
  - Kopiowanie GPU-GPU
  - (bez sensu, ale chodzi nam o najprostszy problem)

Hello World wielowątkowo

- Jądro dla hello world w wersji wielowątkowej:

```
__constant__ __device__ char hw[] = "Hello World!\n";
__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    int idx = blockIdx.x;
    napis_device[idx] = hw[idx];
}

…

helloWorldOnDevice <<< 14, 1 >>> ();
```

- idx
  - zawiera numer bloku w którym znajduje się wątek
  - mapowanie blok/wątek -> fragment problemu

  - wątek z idx-ego bloku kopiuje idx-ą literę napisu
  - Kopiowanie GPU-GPU
  - (bez sensu, ale chodzi nam o najprostszy problem)

Hello World wielowątkowo

- Jądro dla hello world w wersji wielowątkowej:

```
__constant__ __device__ char hw[] = "Hello World!\n";
__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    int idx = blockIdx.x;
    napis_device[idx] = hw[idx];
}

…

helloWorldOnDevice <<< 14, 1 >>> ();
```

- idx
  - zawiera numer bloku w którym znajduje się wątek
  - mapowanie blok/wątek -> fragment problemu

  - wątek z idx-ego bloku kopiuje idx-ą literę napisu
  - Kopiowanie GPU-GPU

Hello World wielowątkowo

Jedno z zadań na ćwiczenia to napisanie programu
helloworld_parallel.cu tak, by wykonywany
był przez 14 wątków w jednym bloku.

Projekt wieloplikowy w CUDA

kernels.cu
kernels.h
main.cpp   (#include <kernels.h>)

- Kernele GPU trzymam w *.cu
- Interfejsy C++ do kerneli GPU w *.cu
- Deklaracje interfejsów trzymam w *.h
- W plikach C++ ładuję ww deklaracje interfejsów
- A w praktyce?

Projekt wieloplikowy CUDA

main.cpp

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <stdio.h>

#include "kernels.h"

int main(void)
{
    call_helloworld();

    ...

    printf("%s\n",napis_host);
}
```

Projekt wieloplikowy CUDA

main.cpp

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <stdio.h>

#include "kernels.h"

int main(void)
{
    call_helloworld();

    ...

    printf("%s\n",napis_host);
}
```

kernels.h

```
void call_helloworld(void);
```

Projekt wieloplikowy CUDA

kernels.cu

```
#include <cuda.h>

__device__ char napis_device[14];
__constant__ __device__ char hw[] = "Hello World!\n\0";

__global__ void helloWorldOnDevice(void)
{
    int idx = threadIdx.x;
    napis_device[idx] = hw[idx];
}

void call_helloworld(void)
{
    helloWorldOnDevice <<< 1, 15 >>> ();
}
```

+rozkazy cuda api

main.cpp

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <stdio.h>

#include "kernels.h"

int main(void)
{
    call_helloworld();

    ...

    printf("%s\n",napis_host);
}
```

kernels.h

```
void call_helloworld(void);
```

Ćwiczenia 2013-01-07:

1. Napisz prosty program Hello world jak na wykładzie.

2. Przepisz program tak, aby kolejne litery wypisywane były przez osobne wątki w ramach jednego bloku.

3. Przerób program na projekt wieloplikowy (c++, cu, h).

Uwaga: zadanie jest na ocenę, czas: 2h.