# Programowanie kart graficznych

Architektura i API
część 1

# Literatura:

„NVIDIA CUDA Programming Guide version 4.2”
http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

„NVIDIA CUDA Library Documentation”
http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/docs/online/index.html

# CUDA
# model architektury
# język CUDA C

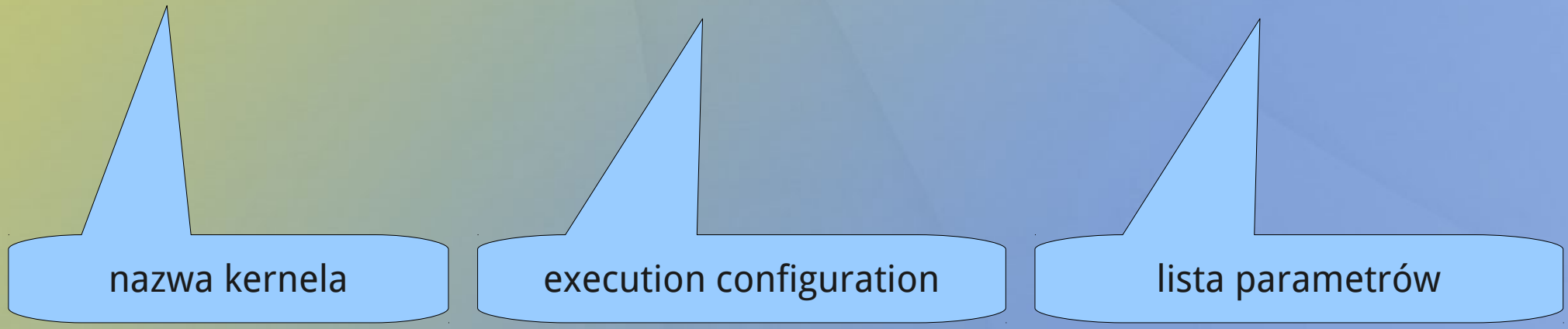
**Kernel:**

- element języka CUDA C - rozszerzenie standardowego języka C/C++
- stanowi specyficzny (składniowo i semantycznie) rodzaj **funkcji**
- wywołanie (aktywacja) kernela powoduje równoległe wykonanie jego treści (tekstu) w *N* równocześnie pracujących wątkach (gdzie *N* jest mniejsze równe liczbie dostępnych rdzeni)
- kernel definiowany jest poprzez użycie słowa kluczowego `__global__`

## Kernel:

- język CUDA C wprowadza specjalny element składniowy nazywany w oryginale mianem *„execution configuration”*
- element ten umieszcza się pomiędzy nazwą wywoływanego kernela i jego listą parametrów i ujmuje w parę „nawiasów” **<<<** i **>>>**
- np.

```
Add<<<numBlocks, threadsPerBlock>>>(A, B, C);
```



**Kernel:**

- każdy z wątków uruchomionych w ramach aktywacji kernela posiada własny, unikalny identyfikator (w oryginale: threadID), który w treści kernela jest dostępny poprzez predefiniowaną zmienną `threadIdx`
- zmienna `threadIdx` jest strukturą

## Przykład prostego kernela i jego aktywacji:

- poniższy kod dodaje w sposób równoległy elementy dwóch wektorów A i B umieszczając wynik w wektorze C
- każdy z N wątków wykonuje dodawanie jednej pary elementów

```
// definicja kernela
__global__ void VecAdd(float* A, float* B, float* C)
{
   int i = threadIdx.x;
   C[i] = A[i] + B[i];
}
int main()
{
   ...
   // wywołanie kernela z aktywacją N wątków
   VecAdd<<<1, N>>>(A, B, C);
}
```

*przykład nr 1*

**Hierarchia wątków:**

- zmienna `threadIdx` jest strukturą (chociaż oryginalny podręcznik twierdzi, że jest wektorem) o trzech polach `x`, `y` i `z`
- takie podejście umożliwia identyfikowanie wątków pracujących (umownie!) w przestrzeni jedno, dwu lub trójwymiarowej
- „przestrzeń” taką można utożsamiać z liczbą wymiarów przetwarzanych agregatów danych (wektory, macierze, prostopadłościany)
- taką umowną „przestrzeń” obliczeń nazywa się w terminologii dokumentacji CUDA blokiem wątków *(thread block)*

**Hierarchia wątków:**

- identyfikator wątku jest powiązany z jego indeksem przechowywanym w zmiennej `threadIdx` w sposób następujący:
  - dla bloków jednowymiarowych identyfikator wątku jest wprost równy wartości

    `threadIdx.x`

## Hierarchia wątków:

- dla bloków dwuwymiarowych o rozmiarach (Dx, Dy) identyfikator wątku jest równy

```
threadIdx.x + threadIdx.y*Dx
```

- dla bloków trójwymiarowych o rozmiarach (Dx, Dy, Dz) identyfikator wątku jest równy

```
threadIdx.x + threadIdx.y*Dx + threadIdx.z*Dx*Dy
```

# Przykład dwuwymiarowej przestrzeni wątków:

- poniższy kod dodaje w sposób równoległy elementy dwóch macierzy A i B, umieszczając wynik w macierzy C
- każdy z N wątków wykonuje dodawanie jednej pary elementów

```
// definicja kernela
__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()
{
...
// aktywacja kernela w bloku N*N*1 wątków
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

*przykład nr 2*

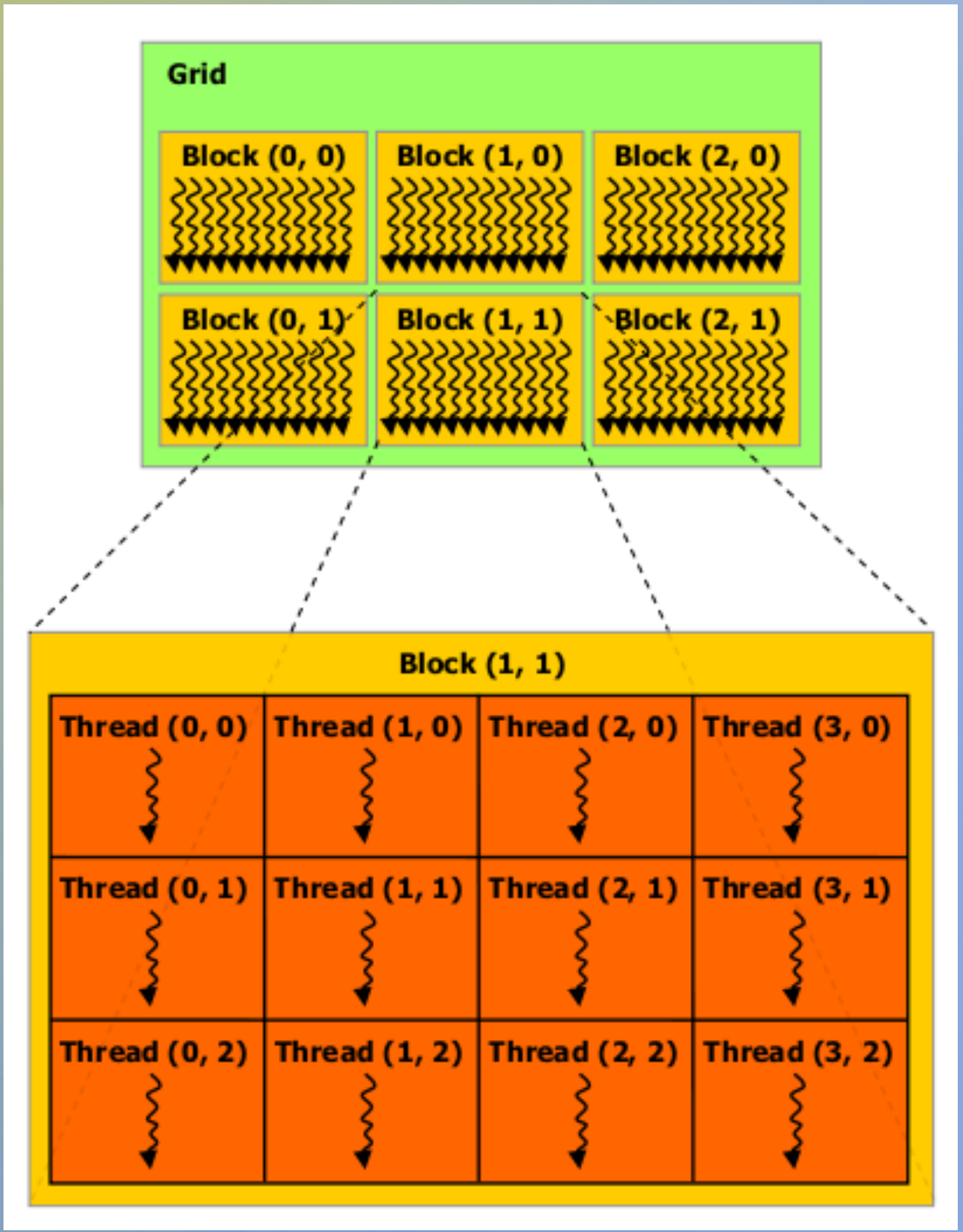
## Hierarchia wątków:

- liczba równolegle (jednocześnie) pracujących wątków jest ograniczona z góry liczbą rdzeni udostępnianych przez sprzęt konkretnej karty CUDA
- łączny rozmiar wszystkich danych przetwarzanych przez wątek jest ograniczony z góry rozmiarem pamięci karty CUDA dostępnej w konkretnym przypadku
- nie znaczy to jednak, że oba powyższe parametry ograniczają z góry rozmiar problemu, jaki może być przetwarzany przez kartę
- treść kernela może być wielokrotnie (sekwencyjnie) wykonana w jednakowo uformowanych blokach
- w takim przypadku łączna liczba wątków jest równa iloczynowi liczby wątków w bloku i liczby bloków

## Hierarchia wątków:

- bloki wątków organizuje się w tzw. siatki bloków (w oryginale *thread grids* )
- siatki bloków są jedno bądź dwuwymiarowe
- liczba bloków w siatce wynika zwykle z rozmiaru przetwarzanych danych lub liczby dostępnych rdzeni

# Przykład dwuwymiarowej siatki wątków:




Źródło: *NVIDIA CUDA Programming Guide Version 3.0*

## Hierarchia wątków:

- liczbę wątków w bloku oraz liczbę bloków w siatce specyfikuje się za pomocą „*execution configuration*" (składnia `<<< ... >>>` )
- wielkość rozmiaru bloku bądź siatki może być podana jako:
  - dana typu `int` (jak w pierwszym przykładzie)
  - dana typu `dim3` (jak w drugim przykładzie)

## Hierarchia wątków:

- każdy blok w siatce bloków jest identyfikowany przez jedno lub dwuwymiarowy indeks
- rozwiązanie to oparte jest na tej samym pomyśle, co identyfikator wątku i korzysta z tych samych mechanizmów
- identyfikator bloku jest przechowywany w predefiniowanej zmiennej (strukturze) o nazwie **`blockIdx`**
- rozmiar bloku dostępny jest w predefiniowanej zmiennej (strukturze) o nazwie **`blockDim`**

# Przykład użycia siatki bloków:

- rozmiar bloku wątków (w celach demonstracyjnych) ustalono na 16x16 (razem 256 wątków)
- dla uproszczenia założono, że rozmiar macierzy (N) dzieli się bez reszty przez rozmiar bloku (16)

```
// definicja kernela
__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()
{
    ...
    // wywołanie kernela
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

*przykład nr 3*

## Wymagania stawiane blokom wątków:

- bloki muszą zostać skonstruowane w sposób biorący pod uwagę fakt, że pracują całkowicie niezależnie
- algorytm podziału problemu na bloki musi uwzględniać każdy z poniższych wariantów:
  - bloki wykonywane są sekwencyjnie
  - bloki wykonywane są równolegle
  - bloki wykonywane są seriami (równolegle w serii, serie wykonywane sekwencyjnie)
- algorytm podziału nie może zakładać, że bloki wykonywane są w jakiejkolwiek znanej z góry kolejności
- tylko spełnienie tych warunków gwarantuje, że uzyskany kod będzie w pełni skalowalny

## Bloki wątków:

- wątki pracujące w ramach jednego bloku mogą organizować wzajemną współpracę poprzez:
  - wymianę danych z użyciem pamięci współdzielonej (dostępnej dla wszystkich wątków)
  - synchronizując wzajemnie swoje wykonanie
- synchronizacja wątków jest możliwa w tzw. punktach synchronizacji (w oryginale: synchronization points) tworzonych poprzez wywołania funkcji `__syncthreads()`
- wg zapewnień zawartych w dokumentacji:
  - szybkość pracy pamięci współdzielonej jest porównywalna z szybkością cache'a L1
  - funkcja `__syncthreads()` jest implementowana sprzętowo i koszt jej wywołania jest minimalny

## Hierarchia pamięci:

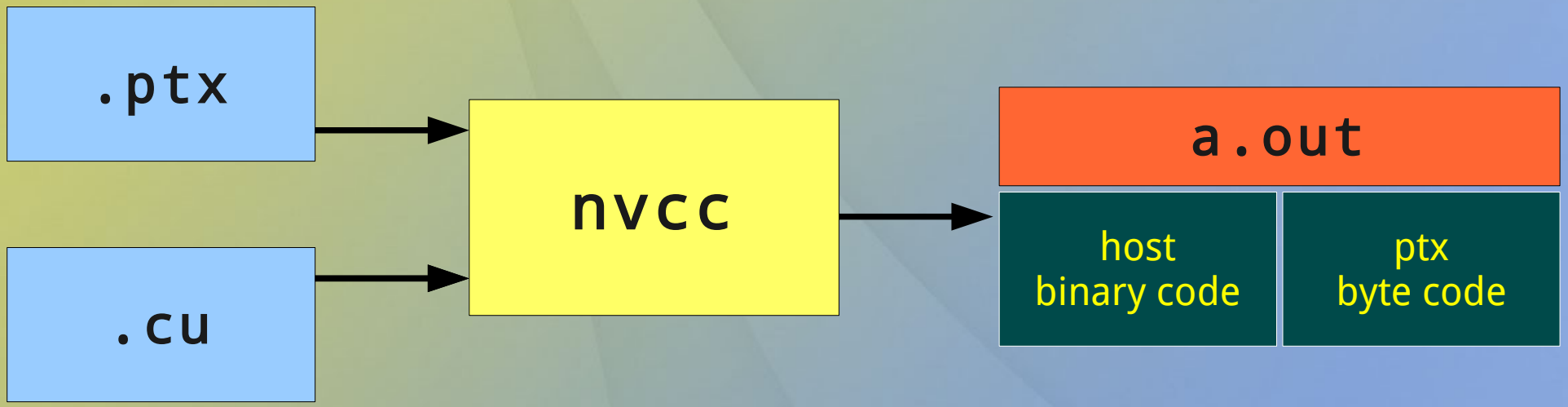
- pracujące wątki mają dostęp do danych przechowywanych w różnych przestrzeniach pamięci urządzenia CUDA:
  - każdy wątek może używać niewielkiej własnej (prywatnej) pamięci lokalnej używanej do przechowywania zmiennych zadeklarowanych lokalnie w ciele kernela; pamięc ta jest w pełni izolowana
  - każdy blok wątków może używać bloku pamięci współdzielonej widocznej dla wszystkich wątków w bloku; czas życia tej pamięci jest tożsamy z czasem życia bloku
  - wszystkie bloki wątków mają dostęp do pamięci globalnej

## Hierarchia pamięci:

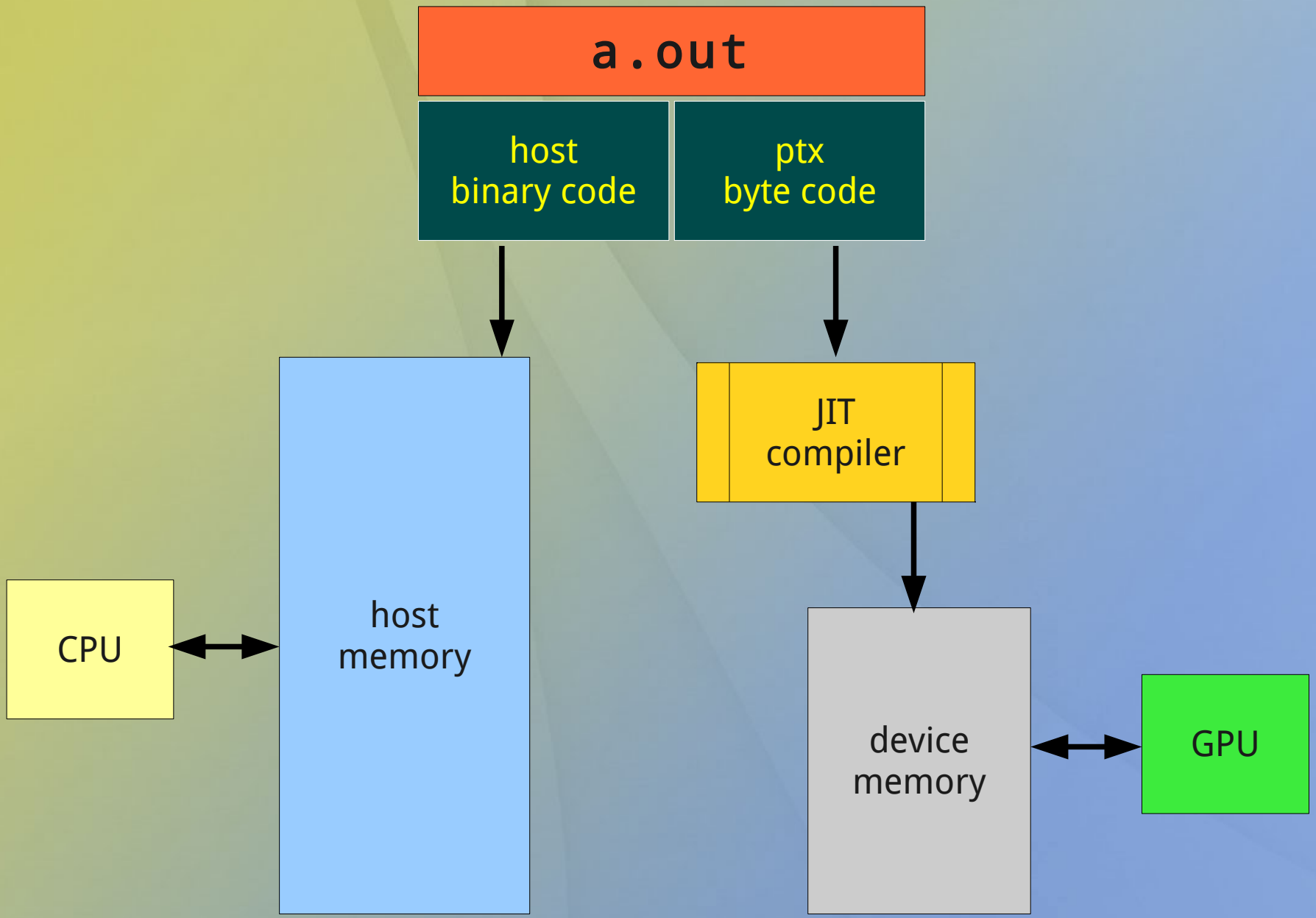
- dodatkowo karta CUDA zawiera dwa dodatkowe, specjalizowane bloki pamięci (z punktu widzenia wątków jest to pamięć tylko do odczytu):
  - pamięć stałych
  - pamięć tekstur
- Zawartość pamięci globalnej, stałych i tekstur może być traktowana jako nieulotna (także pomiędzy odpaleniami kerneli, jeśli tylko pochodzą z jednej i tej samej aplikacji)

# Kompilator nvcc

# Środowisko pracy kompilatora:



# Środowisko uruchomienia kodu :



# CUDA API

## CUDA API – rozpoznawanie cech środowiska:

### pobranie informacji o liczbie dostępnych kart CUDA

```
cudaError_t
cudaGetDeviceCount (int *count)
```

**parametr:**

- `count`: wskaźnik na daną typu int, w której funkcja umieści daną informującą ile urządzeń CUDA jest dostępnych w systemie

**wynik:**

- `cudaSuccess` – poprawne zakończenie działania funkcji

# cudaGetDeviceCount() - przykład użycia

```
int devCnt;

cudaGetDeviceCount(&devCnt);
if(devCnt == 0)
   puts("No CUDA devices available – exiting.");
```

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   canMapHostMemory;
   :
}
```

Pole podaje informację, czy dane urządzenie realizuje funkcjonalności udostępniane przez funkcje `cudaHostAlloc()` i `cudaHostGetDevicePointer()`

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   clockRate;
   :
}
```

Pole podaje informację o częstotliwości zegara danej karty CUDA (częstotliwość wyrażona jest w kilohercach)

# CUDA API – rozpoznawanie cech środowiska:

## reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   computeMode;
   :
}
```

Pole podaje informację o akceptowanym przez daną kartę trybie obliczeń:

- `cudaComputeModeDefault` - tryb domyślny (dla tego urządzenia więcej niż jeden wątek może użyć funkcji cudaSetDevice()).
- `cudaComputeModeExclusive` - tryb wyłączny (dla tego urządzenia tylko jeden wątek może użyć funkcji cudaSetDevice()).
- `cudaComputeModeProhibited` tryb chroniony (żaden wątek nie może użyć funkcji cudaSetDevice() dla tego urządzenia).

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   deviceOverlap;
   :
}
```

Pole podaje informację o tym, czy dany sprzęt urządzenia CUDA potrafi w tym samym czasie przemieszczać (kopiować) bloki pamięci i wykonywać kod kerneli.

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   integrated;
   :
}
```

Pole podaje informację o tym, czy dane urządzenie CUDA jest wbudowane czy dołączone.

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   kernelExecTimeoutEnabled;
   :
}
```

Pole podaje informację o tym, czy dane urządzenie CUDA narzuca limit czasu nieprzerwanej pracy kodu kernela.

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   major;
   :
}
```

Numer wersji CC (Computer Capability) – część „ważniejsza” (przed kropką).

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   maxGridSize[3];
   :
}
```

Maksymalny, akceptowany przed dane urządzenie, rozmiar każdego z trzech wymiarów siatki .

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   maxThreadsDim[3];
   :
}
```

Maksymalny, akceptowany przed dane urządzenie, rozmiar każdego z trzech wymiarów bloku .

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   maxThreadsPerBlock;
   :
}
```

Maksymalna, akceptowana przed dane urządzenie, liczba wątków w bloku.

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   memPitch;
   :
}
```

Maksymalna, akceptowana przed dane urządzenie, wyrażona w bajtach wielkość „boiska” (pitch), które może podlegać jednorazowemu kopiowaniu.

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   minor;
   :
}
```

Numer wersji CC (Computer Capability) – część „pośledniejsza” (za kropką).

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   multiProcessorCount;
   :
}
```

Liczba rdzeni zabudowanych w architekturze danego urządzenia CUDA.

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   char  name[256];
   :
}
```

Nazwa handlowa danego urządzenia CUDA (ASCIIZ).

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   regsPerBlock;
   :
}
```

Liczba 32 bitowych rejestrów dostępnych w danym urządzeniu CUDA dla każdego pracującego bloku wątków.

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   size_t   sharedMemPerBlock;
   :
}
```

Rozmiar (w bajtach) pamięci dzielonej dostępnej w danym urządzeniu CUDA dla każdego pracującego bloku wątków.

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   size_t   textureAlignment;
   :
}
```

Rozmiar (w bajtach) dopasowania tekstur w danym urządzeniu CUDA.

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   size_t   totalConstMem;
   :
}
```

Rozmiar (w bajtach) pamięci stałych w danym urządzeniu CUDA.

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   size_t   totalGlobalMem;
   :
}
```

Rozmiar (w bajtach) pamięci globalnej w danym urządzeniu CUDA.

## CUDA API – rozpoznawanie cech środowiska:

### reprezentowanie informacji o środowisku

```
struct cudaDeviceProp {
   :
   int   warpSize;
   :
}
```

Optymalny rozmiar osnowy wątków (temat ten omówimy przy najbliższej nadarzającej się okazji).

**CudaDeviceProp**
faktyczna
deklaracja:

```
struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    size_t totalConstMem;
    int major;
    int minor;
    int clockRate;
    size_t textureAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
    int integrated;
    int canMapHostMemory;
    int computeMode;
}
```

# CUDA API – rozpoznawanie cech środowiska:

## pobranie informacji o środowisku

```
cudaError_t
cudaGetDeviceProperties(
   struct cudaDeviceProp *prop,
   int   device)
```

**parametry**:

- `prop`: wskaźnik na strukturę, w której funkcja umieści efekt swojego działania
- `device`: numer urządzenia, którego cechy mają zostać rozpoznane (pierwsza karta ma numer 0)

**wynik:**

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

## CUDA API – elementarne zarządzanie pamięcią:

### przydzielenie bloku pamięci urządzenia

```
cudaError_t
cudaMalloc(void **devPtr,size_t size)
```

**parametry**:

- `devPtr`: wskaźnik na wskaźnik reprezentujący przydzielony blok pamięci
- `size`: rozmiar żądanego bloku pamięci (w bajtach)

**wynik:**

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

## CUDA API – elementarne zarządzanie pamięcią:

### zwolnienie bloku pamięci urządzenia

```
cudaError_t
cudaFree(void *devPtr)
```

**parametry**:

- `devPtr`: wskaźnik na blok pamięci przydzielony wcześniej funkcją `cudaMalloc()` lub `cudaMallocPitch()`

**wynik:**

- `cudaSuccess` – poprawne zakończenie działania funkcji
- `cudaInvalidDevicePointer` – niepoprawna wartość wskaźnika devPtr

## CUDA API – elementarne zarządzanie pamięcią:

### kopiowanie bloku pamięci z/do urządzenia

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

**Parametry:**

- `dst`: wskaźnik na docelowy blok pamięci (hosta lub urządzenia)
- `src`: wskaźnik na źródłowy blok pamięci (urządzenia lub hosta)
- `count`: rozmiar kopiowanego bloku pamięci (w bajtach)
- `kind`: kierunek kopiowania - jeden z:
  - `cudaMemcpyHostToHost` (host → host)
  - `cudaMemcpyHostToDevice` (host → karta)
  - `cudaMemcpyDeviceToHost` (karta → host)
  - `cudaMemcpyDeviceToDevice` (karta → karta)

**Wynik:**

- `cudaSuccess` – poprawne zakończenie działania funkcji
- możliwe sytuacje błędów:
  `cudaErrorInvalidValue`,
  `cudaInvalidDevicePointer`,
  `cudaErrorInvalidDevicePointer`,
  `cudaErrorInvalidMemcpyDirection`

# Kompletny program doświadczalny - część 1/2

```
#include      <stdio.h>
#define       N             1024
#define       BLOCK_SIZE    16

float         hArray[N];
float       * dArray;
int           blocks;

void prologue(void) {
    for(int i = 0; i < N; i++) {
        hArray[i] =  i + 1;
    }
    cudaMalloc((void**)&dArray, sizeof(hArray));
    cudaMemcpy(dArray, hArray, sizeof(hArray),cudaMemcpyHostToDevice);
}

void epilogue(void) {
    cudaMemcpy(hArray, dArray, sizeof(hArray), cudaMemcpyDeviceToHost);
    cudaFree(dArray);
}
```

# Kompletny program doświadczalny - część 2/2

```
__global__ void sqr(float *A) {
    int x = blockDim.x * blockIdx.x + threadIdx.x;

    if(x < N)
       A[x] = A[x] * A[x];
}

int main(int argc, char** argv)
{
    int    devCnt;
    cudaGetDeviceCount(&devCnt);
    if(devCnt == 0) {
       perror("No CUDA devices available -- exiting.");
       return 1;
    }
    prologue();
    blocks = N / BLOCK_SIZE;
    if(N % BLOCK_SIZE) blocks++;
    sqr<<<blocks, BLOCK_SIZE>>>(dArray);
    cudaThreadSynchronize();
    epilogue();
    return 0;
}
```