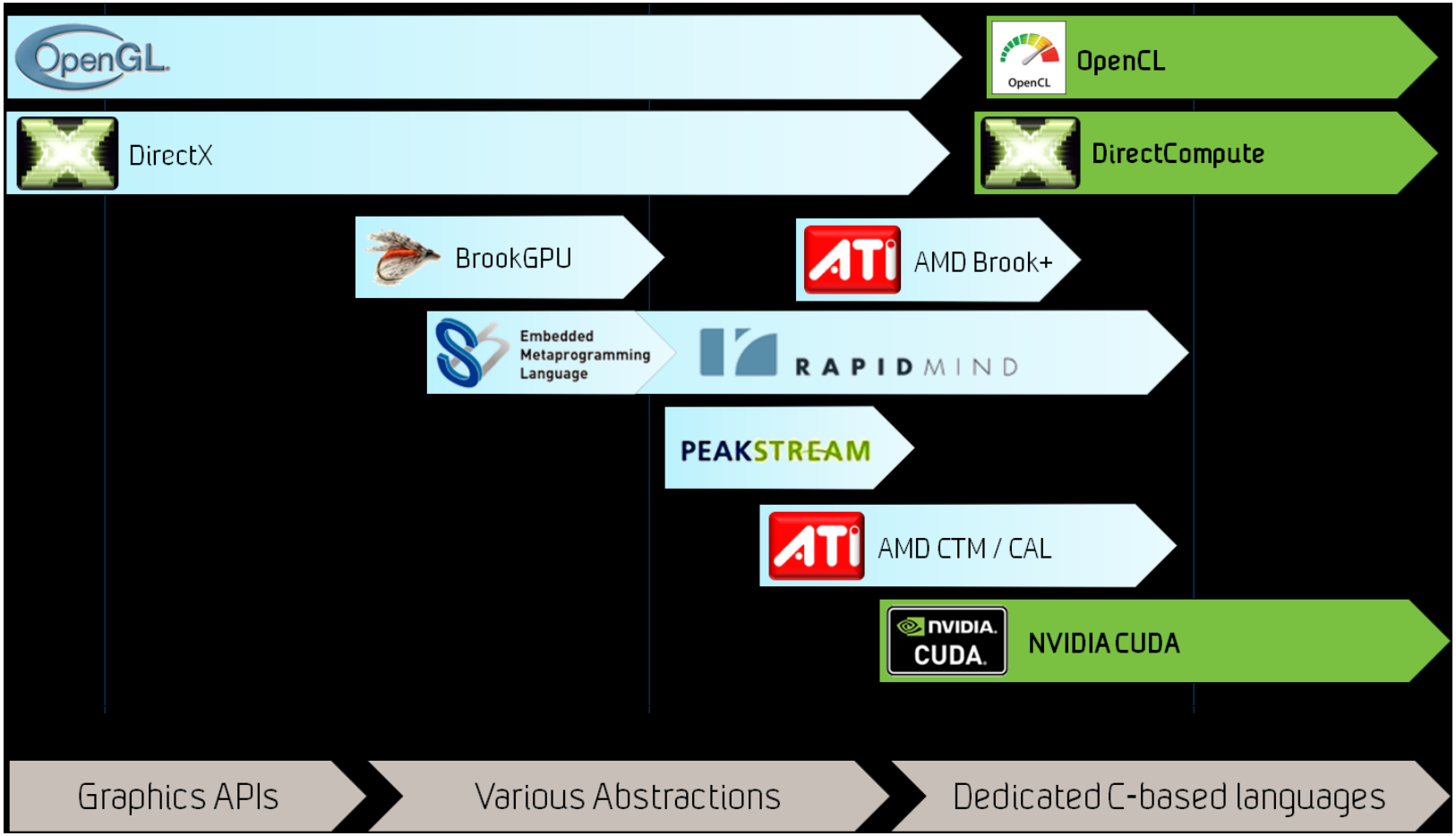
CUDA paralelno programiranje – osnove

Programski jezici za GPU izračunavanja



Izvor: Nvidia

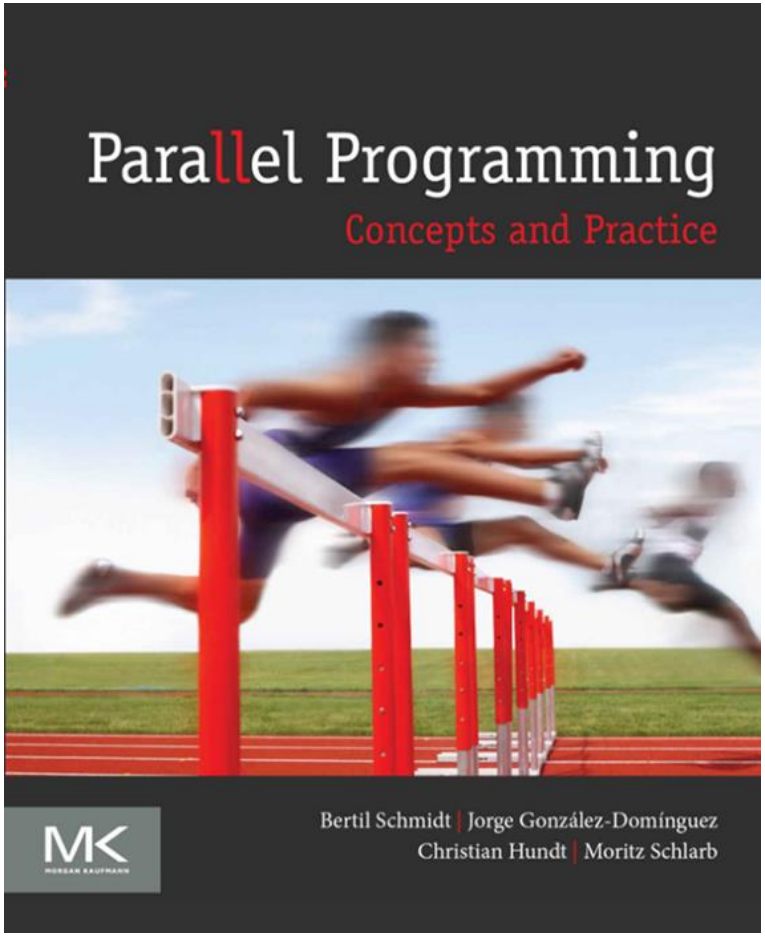
Online resursi – GPGPU programiranje

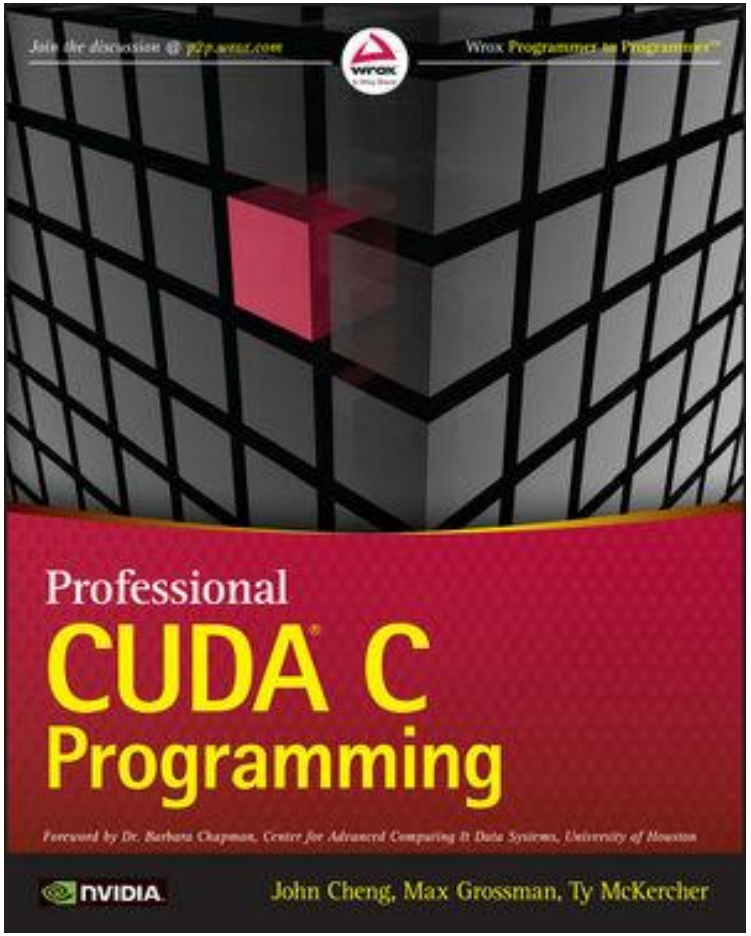
https://developer.nvidia.com/cuda-zone

https://www.khronos.org/opencl/

https://www.openacc.org/

CUDA literatura





https://parallelprogrammingbook.org/

Nvidia CUDA

- **Paralelna platforma** i **API**, prva verzija CUDA predstavljena je 2007.
- Nudi visoke performanse – **GPU** i **CUDA** su **osnova za implementaciju i izvršavanje algoritama dubokih neuronskih mreža** (engl. *deep learning*), kao i **konsenzus algoritama u javnim blokčejn** (engl. *blockchain*) **mrežama** (*proof-of-work* algoritmi, tj. *mining*)
- Podrška za **C**, **C++**, **Fortran** i **Python**, proširenje za C zasnovano na ISO C99 standardu
- **Dobro razvijeno programsko okruženje** sa većim brojem alata za profilisanje i debagiranje (*CUDA Toolkit*, *Nsight IDE*, *Visual Profiler*), **bogatim skupom biblioteka** (*cuFFT*, *cuBLAS*, *Thrust*, ...)
- **CUDA** je vezana **isključivo za Nvidia hardver**

Proširenja C/C++ i API

- **Specifikacija deklaracija**

  global, device, shared, local, constant

- **Rezervisane reči**

  threadIdx, blockIdx

- **Intrinsičke funkcije**

  __syncthreads()

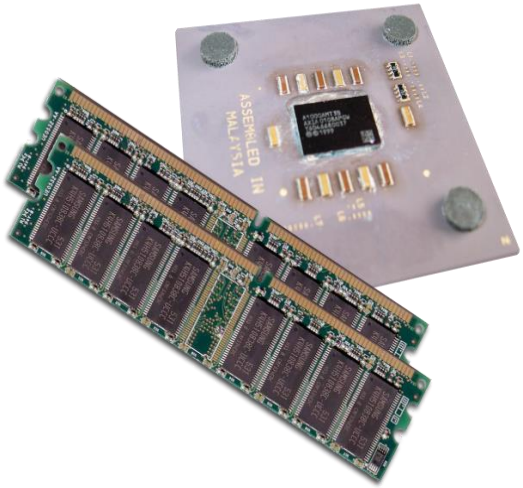
- **Driver API** i **Runtime API**
  (niskog i visokog nivoa, upravljanje memorijom i radom)

- **Pokretanje funkcija**

```
__device__ float vector[N];
__global__ void compute (float *data)  {

 __shared__ float region[M];
 ...

 region[threadIdx] = data[i];

 __syncthreads()
 ...

 data[j] = result;
}

// Alokacija GPU memorije
void *myVector = cudaMalloc(bytes)


// 64 bloka, 256 niti po bloku
compute<<<64, 256>>> (myVector);
```

Terminologija

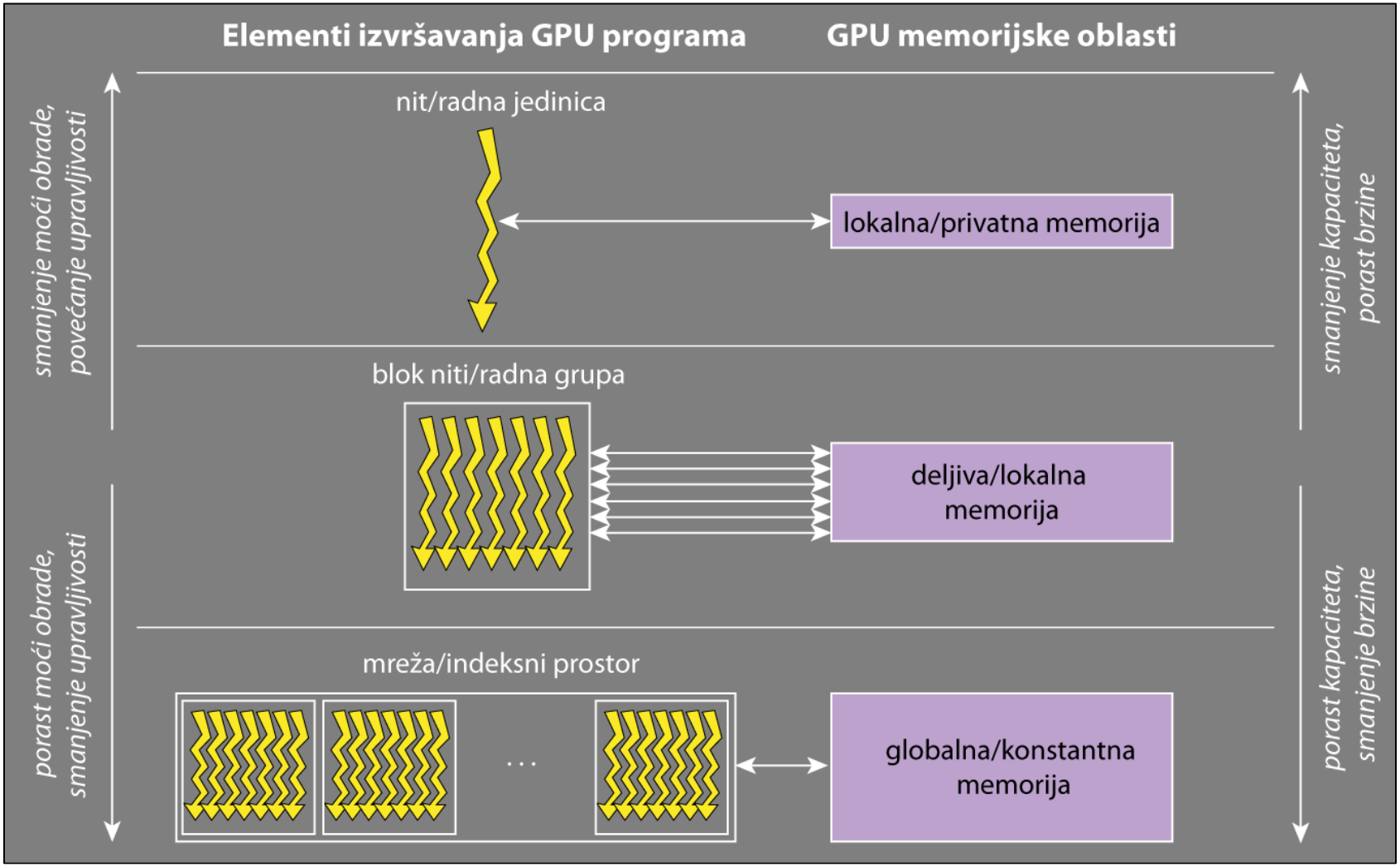
- Osnovni koncepti u CUDA paralelnom modelu:
  - **Domaćin** (engl. *host*) je CPU i njegova memorija (memorija domaćina – *host memory*)
  - **Uređaj** (engl. *device*) je GPU i njegova memorija (memorija uređaja – *device memory*)



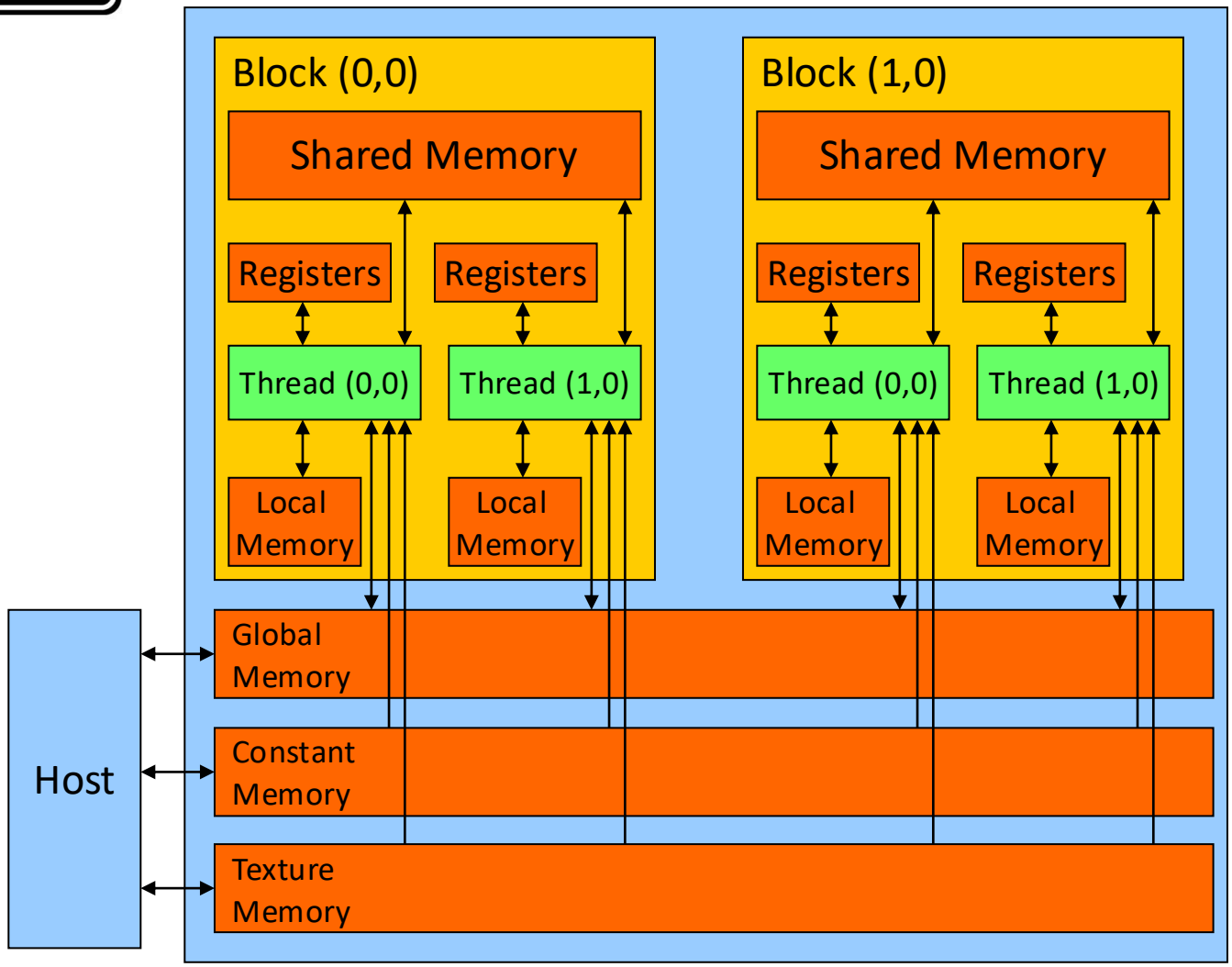
Domaćin (Host)

Uređaj (Device)

Apstrakcije u GPGPU programiranju



CUDA memorijski model



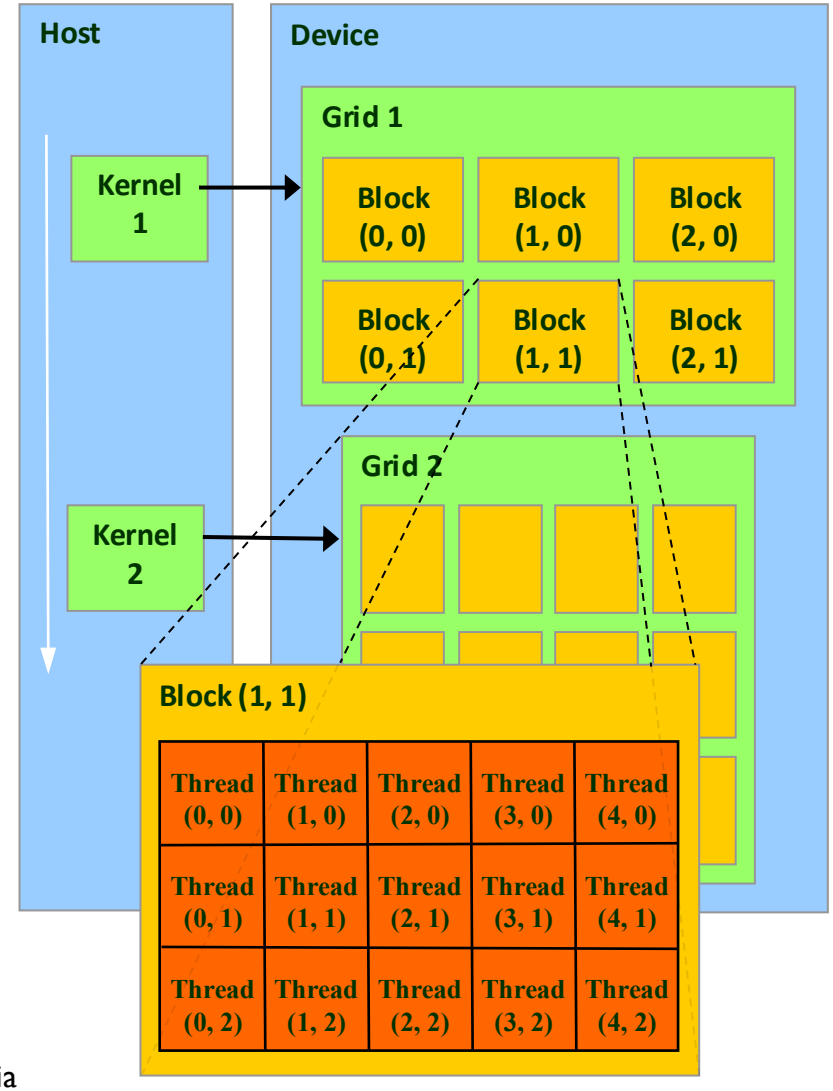
Izvor: Nvidia (https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

CUDA programski model

- GPU se posmatra kao uređaj za izračunavanje koji:
  - predstavlja **visoko-višenitni** (engl. *highly multithreaded*) **koprocesor** za **CPU**
  - ima **svoju globalnu memoriju**
  - izvršava **veliki broj niti paralelno**
- **Svaki kernel opisuje izračunavanja koja izvršava jedna nit**
- Razlike između GPU i CPU niti:
  - **GPU niti** su ekstremno **lagane** (veoma malo dodatnog napora prilikom kreiranja niti) – nitima upravlja hardver
  - GPU su **neophodne desetine hiljada niti za punu efikasnost**

Organizacija niti

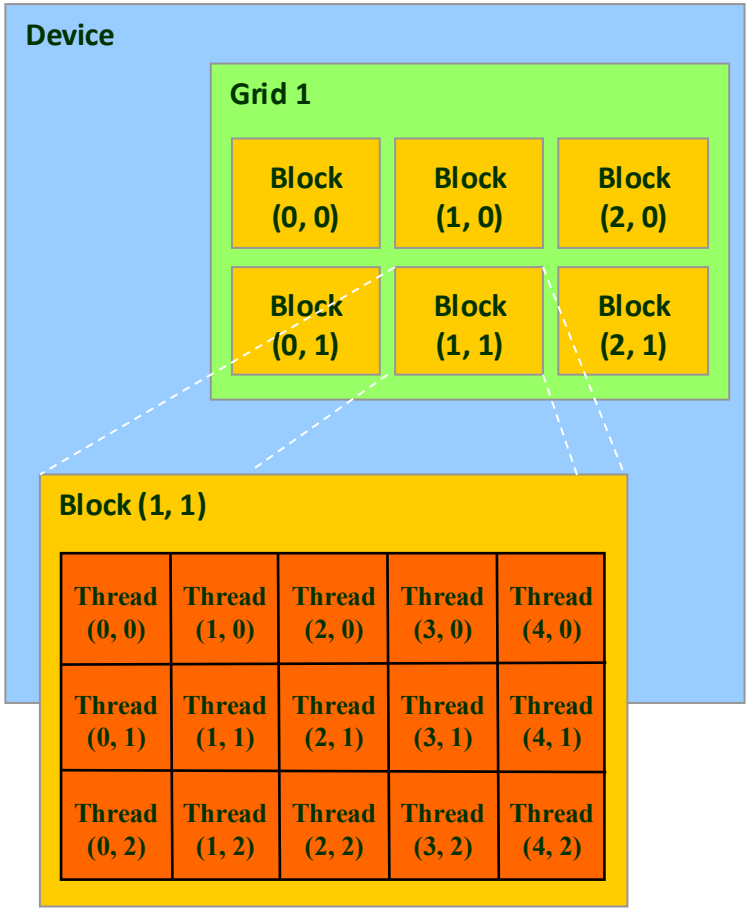
- **Na logičkom nivou, kernel** se **izvršava** kao **mreža** (engl. *grid*) **blokova** (engl. *blocks*) **sa nitima** (engl. *threads*)
  - Sve niti mogu pristupati globalnoj memoriji
- **Blok niti je skup niti koje mogu međusobno da sarađuju** putem:
  - Sinhronizacije svog izvršavanja
  - Deljenja podataka primenom deljene memorije sa niskom latencijom
- **Niti iz različitih blokova ne mogu da sarađuju**
- **Na nivou hardvera, niti se izvršavaju pomoću osnova** (engl. *warp*)



Izvor: Nvidia

Identifikatori blokova i niti

- Niti i blokovi imaju **identifikatore** (ID)
  - Svaka nit zna sa kojim podacima treba da radi na osnovu svog ID
  - Identifikatori blokova su do 2D (blockIdx.x, blockIdx.y)
  - Identifikatori niti su do 3D (threadIdx.x,threadIdx.y, threadIdx.z)



Izvor: Nvidia (https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

Primeri CUDA paralelnih programa

Primer 1 – "Zdravo CUDA svete!"

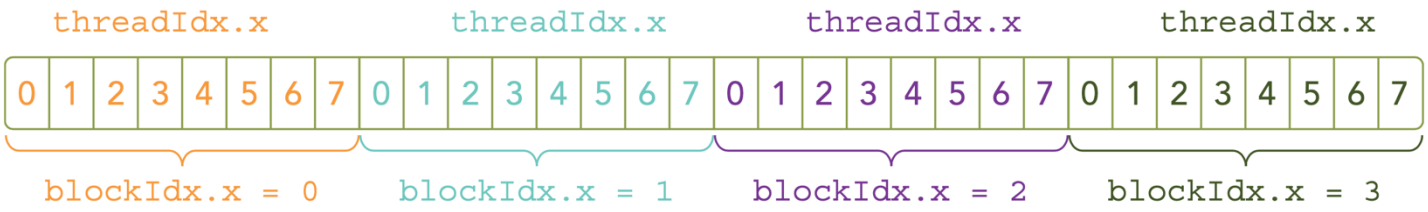
```
#include <iostream>
using namespace std;

// Kernel tj. funkcija koja se izvrsava na GPU tj. uredjaju
__global__ void helloWorldKernel()
{
	const int tid = blockDim.x * blockIdx.x + threadIdx.x;
	cout << "Zdravo CUDA svete od niti " << tid << "!\n";
}

// Funkcija koju izvrsava host tj. domacin
int main ()
{
	// Pokreni kernel pomocu 4 niti koje se izvrsavaju u jednom bloku
	helloWorldKernel<<<1, 4>>>();
	cudaDeviceSynchronize();
	return 0;
}
```

Poziv kompajlera: nvcc helloWorld.cu -02 –o helloWorld

CUDA konfiguracija izvršavanja kernela

- Konfiguracija izvršavanja (engl. *execution configuration*):

ime_kernela<<<4, 8>>>(lista argumenata);



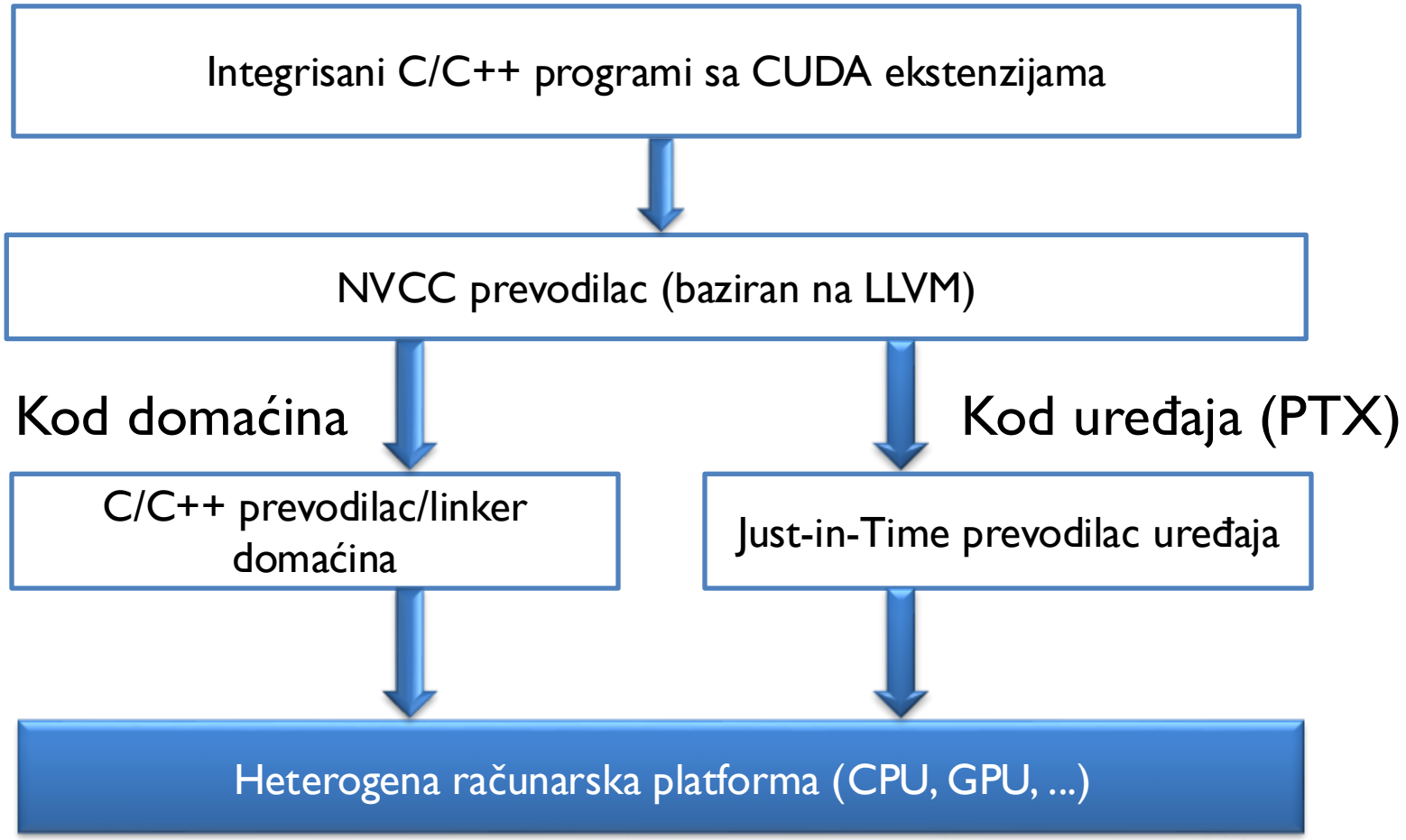
Izvor: Professional CUDA C Programming

CUDA C/C++ deklaracije funkcija

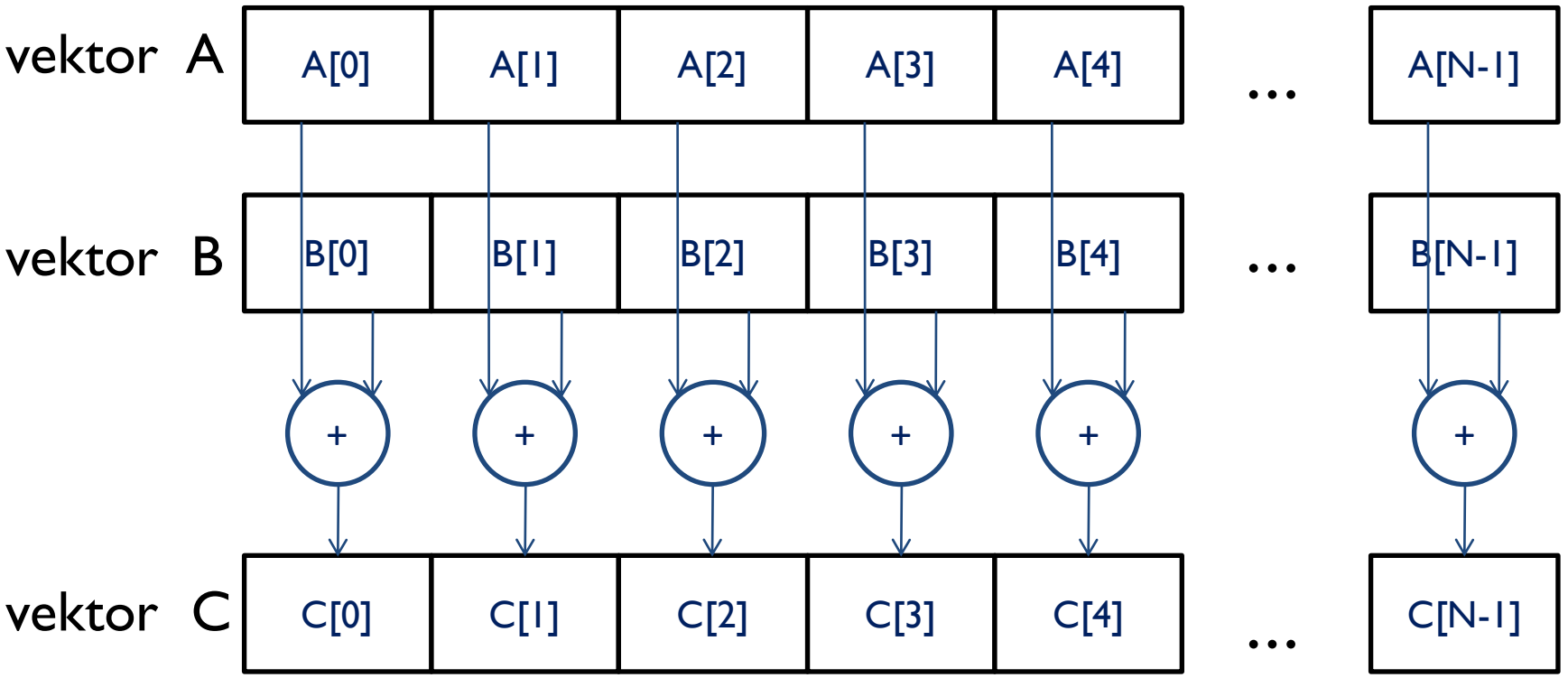
	Izvršava se na:	Može je pozvati samo:
__device__ float DeviceFunc()	uređaju	uređaj
__global__ void KernelFunc()	uređaju	domaćin
__host__ float HostFunc()	domaćinu	domaćin

- Kernel funkcija mora imati povratnu vrednost tipa **void**
- Kernel može pristupiti samo memoriji uređaja
- Kernel se izvršava asinhrono
- Kod kernela nema podrške za statičke promenljive

Prevođenje CUDA C/C++ programa



Primer 2 – Sabiranje vektora



Izvor: Nvidia (https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

CPU sabiranje vektora – "klasični" kod

```
// Izracunava sumu vektora C = A+B
void vectorAdd(float* A, float* B, float* C, int n)
{
  for (i = 0, i < n, i++)
    C[i] = A[i] + B[i];
}

int main()
{
   // Alokacija memorije za A_h, B_h i C_h
      // I/O za citanje A_h i B_h sa po N elemenata
      ...
   vectorAdd(A_h, B_h, C_h, N);
      ...
}
```

GPU sabiranje vektora – kod domaćina

```
#include <cuda.h>
…
void vectorAdd(float* A, float* B, float* C, int n)
{
        int size = n* sizeof(float);
        float* A_d, B_d, C_d;
        …
        // 1.
        // Alokacija memorije uredjaja za A, B i C
        // Kopiranje A i B u memoriju uredjaja
        // 2.
        // Kod za pokretanje kernela – uredjaj izvrsava
        // masivno paralelno sabiranje vektora
        // 3.
        // Kopiranje C iz memorije uredjaja
        // Oslobadjanje memorije
}
```

```
void vectorAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float* A_d, B_d, C_d;

    // 1. Prenos A i B u memoriju uredjaja
    cudaMalloc((void **) &A_d, size);
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
    // Alokacija memorije uredjaja za C
    cudaMalloc((void **) &C_d, size);

    // 2. Kod za poziv kernela
     ...
    // 3. Transfer C sa uredjaja na domacina
    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
    // Oslobadjanje memorije uredjaja od A, B i C
    cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```

GPU sabiranje vektora – kod domaćina (kernel)

```
// Izracunava sumu vektora C = A+B
// Svaka nit izvrsava pokomponentno sabiranje
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) C_d[i] = A_d[i] + B_d[i];
}

void vectorAdd(float* A, float* B, float* C, int n)
{
    ...
    // Alokacija i kopiranje A_d, B_d, C_d
    // Izvrsava se ceil(n/256) blokova sa po 256 niti svaki
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
    ...
}
```

GPU sabiranje vektora – pokretanje kernela

```
void vectorAdd(float* A, float* B, float* C, int n)
{
    …
  // Alokacija i kopiranje A_d, B_d, C_d
  // Izvrsava se ceil(n/256) blokova sa po 256 niti svaki
  dim3 DimGrid(n/256, 1, 1);
  if (n%256) DimGrid.x++;
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
    …
}
```