Računarske vežbe
Letnji semestar 2023/24.

UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
KATEDRA ZA PRIMENJENE RAČUNARSKE NAUKE

# *CUDA* programiranje

# Sadržaj

- CUDA alati
- Programski model
- Arhitektura uređaja
- Uobičajeni redosled događaja u CUDA programima
- Kernel funkcija
- Primer 1 — sabiranje vektora
- Obrada matrice (slike) 2D mrežom
- Primer 2 — množenje matrice (slike) skalarom
- Primer 3 — pisanje kernela za zamućenje slike

# Programsko okruženje

- Lokalna mašina (neophodno imati grafičku karticu koja podržava *CUDA*-u):
  - direktno na mašini
  - doker Container
- *Google Colab Notebooks*:
  - alternativa lokalnoj mašini
  - ispratiti uputstva sa *ACS*-a za podešavanje

# CUDA alati

# CUDA alati

- NVIDIA nudi razne alate za rad sa CUDA programima:
    - nvcc – drajver kompajlera
    - CUDA-GDB – debager
    - *Compute Sanitizer* – alat za proveru ispravnosti CUDA aplikacija (pristup memoriji, trke do podataka, problemi sa sinhronizacijom i inicijalizacijom…)
    - *Nsight* serija alata – alati za prikupljanje informacija o izvršenju koda, pozivu funkcija, utrošenim resursima; uključuju i grafički interfejs
- Spisak alata: https://docs.nvidia.com/cuda/#tools

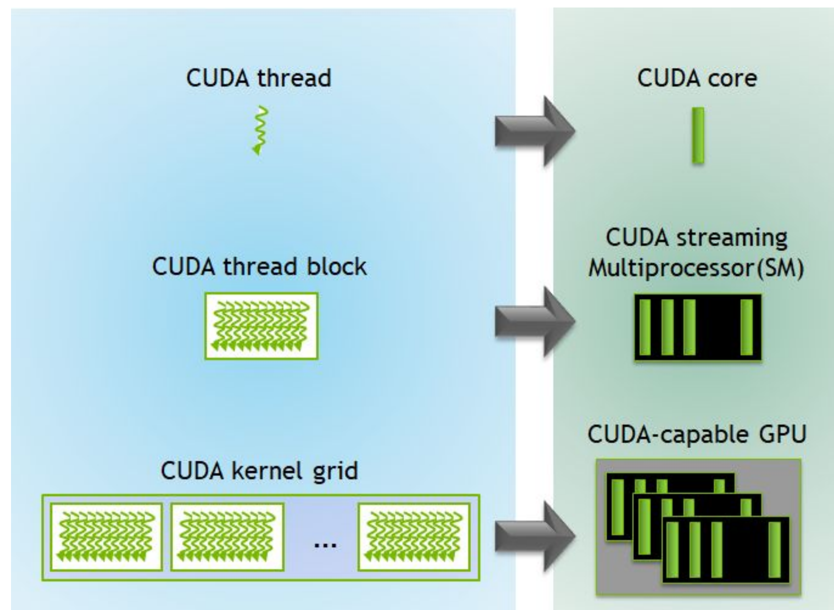
# Programski model

# Programski model

- Domaćin (*host*) — CPU sa svojom memorijom
- Uređaj (*device*) — GPU sa svojom memorijom
- Kernel — funkcija koja se izvršava **na uređaju**

# Arhitektura uređaja

# Arhitektura uređaja



# Arhitektura uređaja

- Izvršavanje CUDA programa podrazumeva pokretanje **blokova niti**
- Logički gledano, analogija je sledeća:
    - Nit == skalarni procesor (*CUDA core*)
    - Blok niti ~= SM (preslikavanje nije 1 na 1, jedan SM može da sadrži više blokova)
    - Mreža (grid) blokova == uređaj
    - Na najnižem logičkom nivou, blokovi su podeljeni u *warp*-ove od po 32 niti (16 na starijim arhitekturama)

# Uobičajeni redosled događaja u *CUDA* programima

# Uobičajeni redosled događaja u *CUDA* programima

1. Alocirati memoriju na domaćinu i uređaju.
2. Prebaciti podatke sa domaćina na uređaj.
3. Pozvati izvršenje jednog ili više kernela.
4. Prebaciti rezultate sa uređaja na domaćina.
5. Dealocirati memoriju na domaćinu i uređaju.

# 1. Alocirati memoriju na domaćinu i uređaju

- Na domaćinu — korišćenjem dinamičke alokacije memorije:

```
float *A = (float *) calloc(n, sizeof(float)); // niz A ce imati sve elemente 0
float *B = (float *) malloc(n * sizeof(float));
float *C = (float *) malloc(n * sizeof(float));
```

- Na uređaju — korišćenjem funkcije cudaMalloc

```
float *A_d, *B_d, *C_d;
cudaMalloc((void **) &A_d, size);
cudaMalloc((void **) &B_d, size);
cudaMalloc((void **) &C_d, size);
```

# *cudaMalloc*

- *cudaError_t cudaMalloc(void** buffer, size_t size)*
- Povratna vrednost je kôd greške (kao i kod većine CUDA funkcija)
- *void** buffer* — pokazivač na pokazivač na bilo šta:
  - Povratna vrednost je rezervisana za kôd greške, pa se kroz nju ne može vratiti pokazivač, kao što je to slučaj za *malloc* i *calloc* funkcije.
  - Zato se prosledi adresa pokazivača, kako bi se nova vrednost pokazivača vratila preko parametra funkcije.
  - Za bolje razumevanje ovog koncepta pogledati fajl *primeri/p01_cuda_pointer_to_pointer.c*.
- *size_t size* — količina alocirane memorije

# 2. Prebaciti podatke sa domaćina na uređaj

- Korišćenjem *cudaMemcpy* funkcije

```
cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
```

*destination* parametar, memorija **u** koju se kopira
sadržaj; u ovom slučaju, memorija zauzeta na uređaju

# 2. Prebaciti podatke sa domaćina na uređaj

- Korišćenjem *cudaMemcpy* funkcije

```
cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
```

*source* parametar, memorija **iz** koje se kopira sadržaj; u ovom slučaju, memorija zauzeta na domaćinu

# 2. Prebaciti podatke sa domaćina na uređaj

- Korišćenjem *cudaMemcpy* funkcije

```
cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
```

veličina kopirane memorije u bajtovima

# 2. Prebaciti podatke sa domaćina na uređaj

- Korišćenjem *cudaMemcpy* funkcije

```
cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
```

*enum* tip koji označava smer kopiranja; u ovom slučaju, kopira se **sa domaćina na uređaj**

# 3. Pozvati izvršenje jednog ili više kernela

- Koristi se sledeća sintaksa:

```
myKernel<<<gridDim, blockDim>>>(parameters…)
```

- *gridDim* – veličina mreže, odnosno broj blokova po dimenzijama mreže
- *blockDim* – veličina bloka, odnosno broj niti po dimenzijama bloka
- Mreža i blokovi mogu imati od jedne do tri dimenzije (više detalja na kasnijim slajdovima).
- Parametri koji se prosleđuju po vrednosti ne zahtevaju nikakve prethodne korake.
- Pokazivači zahtevaju prethodnu alokaciju prostora na uređaju (*cudaMalloc*).
- **Svaka nit pokrenuta u pozivu kernela izvršiće isti kôd napisan u kernelu.**

# Pokretanje dovoljnog broja niti

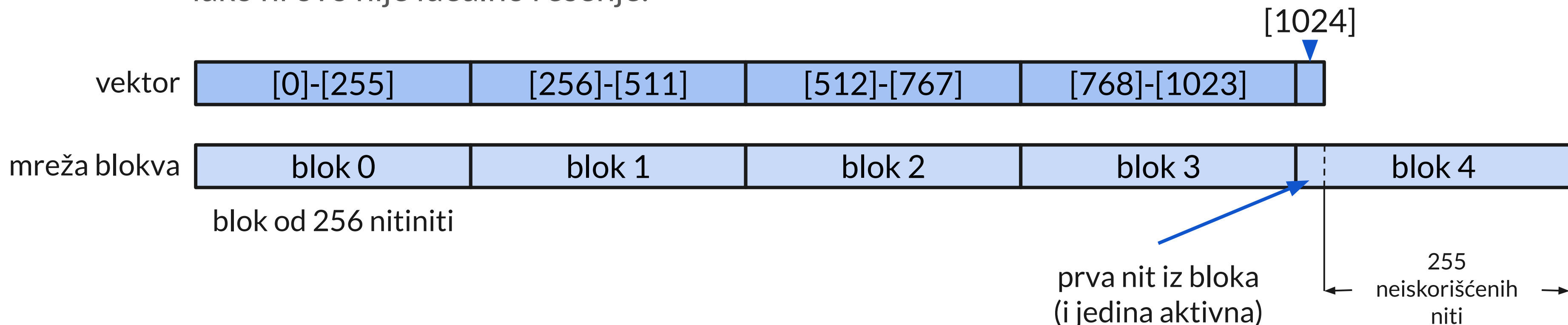
- Recimo da obrađujemo vektor od **1025** elemenata
- Treba odrediti:
  - veličinu bloka, i
  - veličinu grida (mreže blokova)

# Pokretanje dovoljnog broja niti

- Biramo da veličina bloka bude 256 niti:
  - dobro je da bude umnožak od 32, s obzirom na to da je veličina *warp*-a 32
- Za mrežu biramo broj blokova tako da ukupan broj niti pokrije sve elemente:
  - delimo veličinu vektora (1025) sa veličinom bloka (256) i dobijamo razlomljenu vrednost nešto veću od 4
  - biramo prvu veću vrednost za veličinu mreže, kako bismo pokrili ceo vektor

# Pokretanje dovoljnog broja niti

- Nedostatak ovakog rešenja je veliki broj nezaposlenih niti.
- Alternativa bi bila da se pokrene 1024 niti, a da jedna od njih obradi dva elementa, iako ni ovo nije idealno rešenje.



# 3. Pozvati izvršenje jednog ili više kernela

- Sažetak ove priče je sledeći deo koda:

```
vecAddKernel<<<ceil(n / 256.0), 256>>>(A_d, B_d, C_d, n);
```

- Funkcija *ceil* pretvara razlomljenu vrednost u prvu veću celobrojnu. To je jedan od načina da se postaramo da veličina mreže bude dovoljna za ceo niz.
- Prvi parametar unutar <<<>>> je broj blokova (veličina mreže), a drugi je broj niti po bloku (veličina bloka).

# 4. Prebaciti rezultate sa uređaja na domaćina

- Koristi se funkcija *cudaMemcpy*, sa drugačijim parametrom za smer kopiranja:

```
cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
```

# 4. Prebaciti rezultate sa uređaja na domaćina

- Koristi se funkcija *cudaMemcpy*, sa drugačijim parametrom za smer kopiranja:

```
cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
```

memorija zauzeta na domaćinu

# 4. Prebaciti rezultate sa uređaja na domaćina

- Koristi se funkcija *cudaMemcpy*, sa drugačijim parametrom za smer kopiranja:

```
cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
```

memorija zauzeta na uređaju

# 4. Prebaciti rezultate sa uređaja na domaćina

- Koristi se funkcija *cudaMemcpy*, sa drugačijim parametrom za smer kopiranja:

```
cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
```

veličina memorije za kopiranje

# 4. Prebaciti rezultate sa uređaja na domaćina

- Koristi se funkcija *cudaMemcpy*, sa drugačijim parametrom za smer kopiranja:

```
cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
```

kopiranje sa uređaja na domaćina

- Za dealokaciju na domaćinu koristi se standardna C funkcija *free*:

```
free(A);
free(B);
free(C);
```

- Za dealokaciju na uređaju koristi se ne tako različita funkcija, *cudaFree*:

```
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
```

# Kernel funkcija

# Kernel funkcija

- Piše se kao bilo koja druga *C/C++* funkcija, ali mora biti *void* tipa, i potpisu mora prethoditi *CUDA* ključna reč `__global__`, koja naznačava da se funkcija izvršava na uređaju, ali se može pozvati iz koda domaćina:

```
__global__ void myKernel(...) {...}
```

- Kernel je funkcija koju izvršavaju sve pokrenute niti.
- Iako sve izvršavaju istu funkciju, konkretan rezultat izvršavanja može zavisiti od pozicije niti u svom bloku, kao i od pozicije njenog bloka u mreži.
- Za uvođenje ovakve zavisnosti koriste se specifične *CUDA* promenljive koje sadrže informacije o strukturi mreže i o poziciji niti i bloka.

# Struktura mreže i blokova

- Na prethodnim slajdovima pokazano je kako se može odrediti struktura mreže i blokova u slučaju kada se radi sa vektorima, odnosno kada su i mreža i blok jednodimenzionalni.
- *CUDA* pruža mogućnost kreiranja mreže i blokova sa jednom, dve, ili tri dimenzije.
- Dimenzije se redom označavaju sa X, Y i Z.
- Kada se, pri pozivu kernela, unutar oznaka <<< >>> navedu dve celobrojne vrednosti odvojene zarezom, te vrednosti definišu X dimenziju mreže i bloka.
- Za kreiranje višedimenzionalnih mreža i blokova, koristi se posebna *dim3* struktura.

# Struktura mreže i blokova – *dim3*

- Konstruktor *dim3* strukture prima jedan, dva, ili tri parametra.
- Parametri su redom X, Y i Z dimenzije, a izostavljanjem Z i/ili Y parametra, smanjuju se dimenzije mreže ili bloka.
- Za kreiranje mreže koja pokriva matricu veličine NxM elemenata blokovima veličine 16x16 niti, tako da svaka nit obradi po jedan element, može se koristiti sledeći izraz:

```
dim3 dimBlock(16, 16); // može i (16, 16, 1)
dim3 dimGrid(ceil((float)M / 16), ceil((float)N / 16));
kernelFunc<<<dimGrid, dimBlock>>>(...);
```

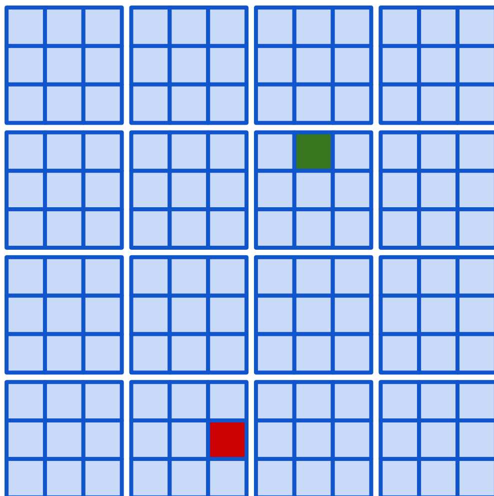
- Broj dimenzija mreže može biti različit od broja diemnzija bloka.

# Struktura mreže i blokova – unutar kernela

- Unutar kernela mogu se koristiti specifične promenljive koje obezbeđuju informacije o strukturi mreže i blokova, kao i o poziciji niti u bloku i bloka u mreži:
  - *gridDim* – informacije o veličini mreže (broju blokova)
  - *blockDim* – informacije o veličini bloka (broju niti)
  - *blockIdx* – informacije o poziciji konkretnog bloka u mreži
  - *threadIdx* – informacije o poziciji konkretne niti u bloku
- Sve navedene promenljive su zapravo strukture koje imaju tri polja koja se odnose na konkretne dimenzije – *x*, *y*, i *z*.
- Tako, na primer, *gridDim.x* je broj blokova po *x* dimenziji mreže, a *threadIdx.y* je pozicija niti po *y* dimenziji bloka.

# Struktura mreže i blokova – unutar kernela

- Recimo da se radi o mreži veličine 4x4 i o blokovima veličine 3x3:

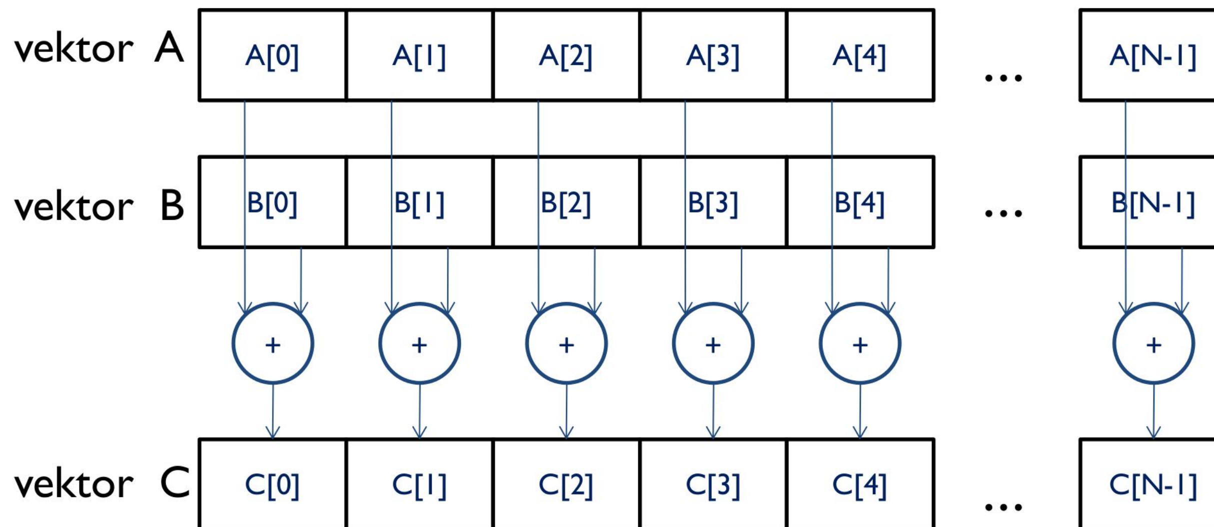


- *CUDA Dim* promenljive :
  - *gridDim.y: 4*
  - *gridDim.x: 4*
  - *blockDim.y: 3*
  - *blockDim.x: 3*

- *CUDA Idx* promenljive:
  - *blockIdx.y: 1*
  - *blockIdx.x: 2*
  - *threadIdx.y: 0*
  - *threadIdx.x: 1*
  - *blockIdx.y: 3*
  - *blockIdx.x: 1*
  - *threadIdx.y: 1*
  - *threadIdx.x: 2*

# Primer 1 — sabiranje vektora

# Primer 1 — sabiranje vektora



# Primer 1 — sabiranje vektora

- Primer ilustruje sabiranje dva vektora, *A* i *B*, od kojih prvi sadrži sve nule, a drugi sve jedinice.
- U primeru su mreža i blokovi jednodimenzionalni, s obzirom na to da se obrađuje 1D vektor.

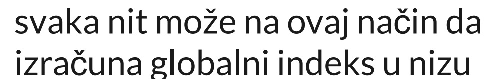
# Primer 1 — sabiranje vektora

- Izgled kernela:

```
__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];
}
```

# Primer 1 — sabiranje vektora

- Izgled kernela:




```
__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];
}
```

# Primer 1 — sabiranje vektora

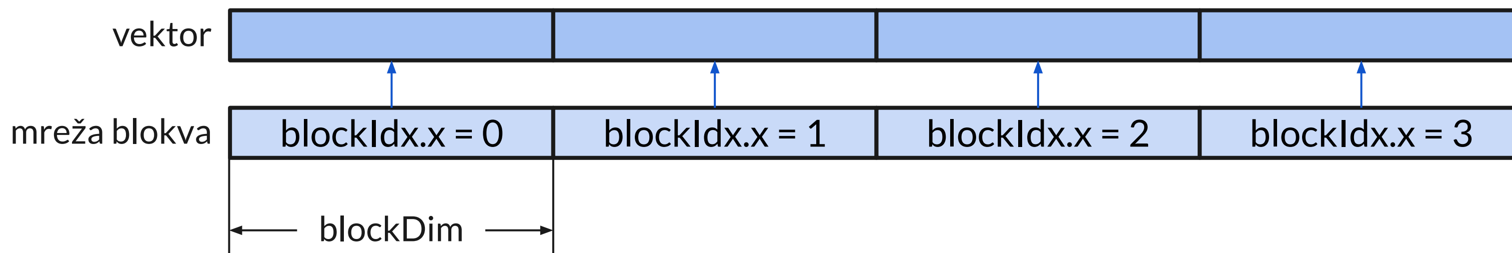
- Izgled kernela:

```
__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];
}
```

zbog potencijalno većeg broja niti od broja elemenata u vektoru, potrebno je ispitati da li je indeks u granicama vektora

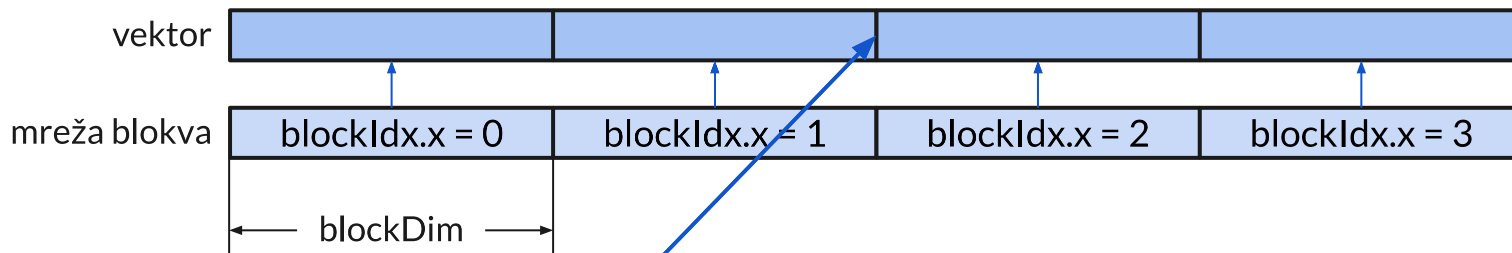
# Primer 1 — sabiranje vektora

- Svaki blok niti zadužen je za neki deo vektora (moguće je da je poslednji blok niti veći od svog dela vektora, ako veličina vektora nije umnožak veličine bloka).



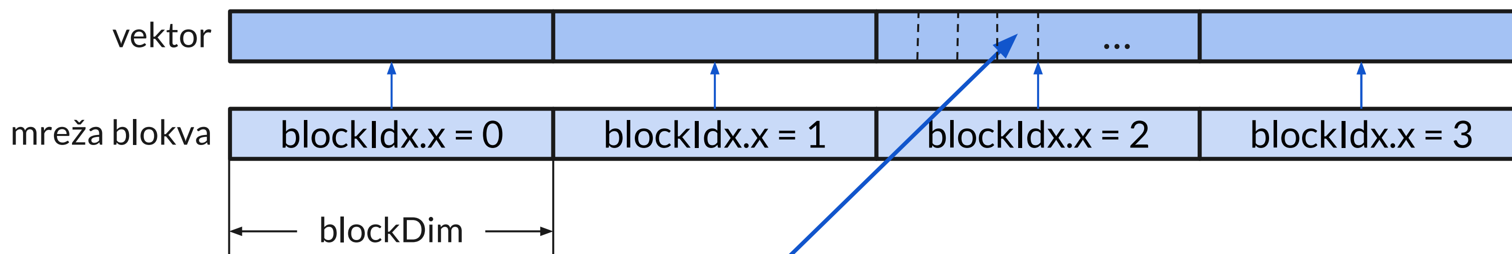
# Primer 1 — sabiranje vektora

- Prvi deo izraza pronalazi početak dela vektora za koji je zadužen blok niti (prikazan slučaj za nit iz bloka 2).



# Primer 1 — sabiranje vektora

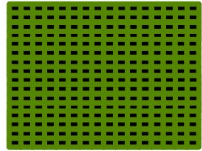
- Korišćenjem svog indeksa unutar bloka, nit pronalazi svoju udaljenost od početka dela vektora (svoj *offset*), odnosno pronalazi element za koji je zadužena (prikazan slučaj za nit sa indeksom 3).



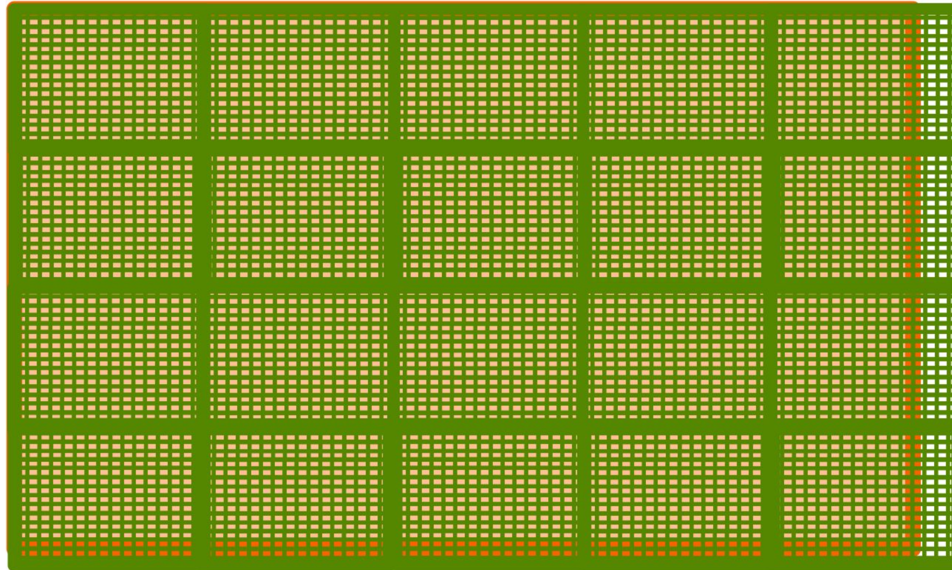

```
int i = blockDim.x * blockIdx.x + threadIdx.x;
```

# Obrada matrice (slike) 2D mrežom

# Obrada matrice 2D mrežom



16×16 blocks



62×76 picture

*CUDA*

# Obrada matrice 2D mrežom

- Sliku koju obrađujemo posmatramo kao matricu.
- Možemo da smatramo da dimenzije matrice odgovaraju dvema dimenzijama mreže, X za širinu i Y za visinu
- Ako koristimo blok od 16 niti, dimenzije mreže i blokova možemo postaviti na sledeći način:

```
dim3 dimBlock(16, 16, 1);
dim3 dimGrid(ceil((float) imageWidth / 16), ceil((float) imageHeight / 16), 1);
```

# Obrada matrice 2D mrežom

- Računanje u kernelu vrste i kolone koju obrađuje konkretna nit...

# Obrada matrice 2D mrežom

- Računanje vrste:
  - Koliko redova ima od „vrha“ slike do bloka u  kome se nalazi naša nit?

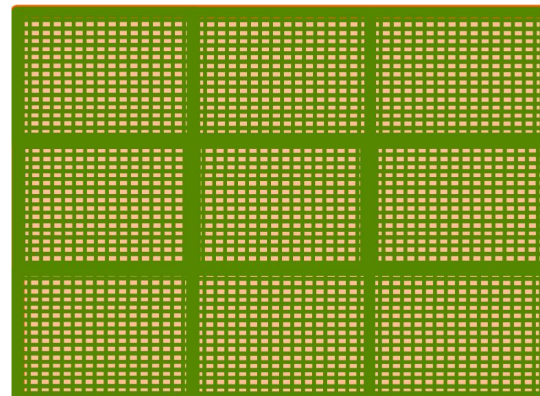
# Obrada matrice 2D mrežom

- Računanje vrste:
  - Koliko redova ima od „vrha“ slike do bloka u kome se nalazi naša nit?
  - Odgovor: *blockDim.y * blockIdx.y*

blockIdx.y = 0

blockIdx.y = 1

blockIdx.y = 2
Iznad ovog bloka ima 2*16, odnosno 2*blokcDim.y niti



# Obrada matrice 2D mrežom

- Računanje vrste:
  - Računicom sa prethodnog slajda smo praktično stigli do *vrha* bloka u kom je naša nit
  - Koliko se treba odaljiti od vrha?

# Obrada matrice 2D mrežom

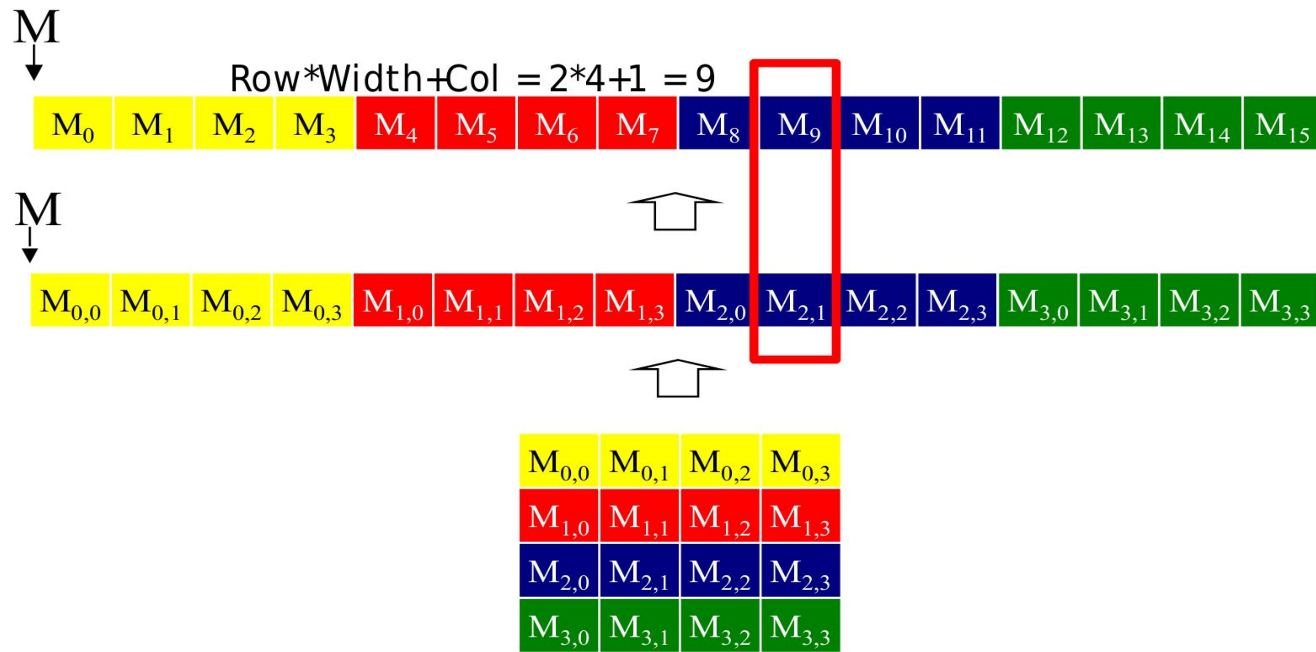
- Računanje vrste:
  - Računicom sa prethodnog slajda smo praktično stigli do *vrha* bloka u kom je naša nit
  - Koliko se treba odaljiti od vrha?
  - Odgovor: *threadIdx.y*
  - Konačna računica:
  - ***blockDim.y * blockIdx.y + threadIdx.y***

# Obrada matrice 2D mrežom

- Računanje kolone:
  - Logika je ista, samo umesto Y posmatramo X dimenziju
  - Konačna računica za kolonu:
  - ***blockDim.x * blockIdx.x + threadIdx.x***

- Izračunali smo vrstu i kolonu naše 2D strukture,  odnosno matrice, za svaku nit.
- Ipak, u kernelu posmatramo matricu kroz niz.
- Praktično, matrica je linearizovana — smestili smo njene vrednosti u niz, ređane vrstu po vrstu.

# Obrada matrice 2D mrežom — linearizacija matrice



# Obrada matrice 2D mrežom

- Treba obratiti pažnju na to da možda imamo više niti po jednoj dimenziji nego piksela, u slučaju da širina ili visina slike nisu deljive sa širinom i visinom bloka.
- Zato treba, pre obrade svog piksela, da nit ispita da li uopšte njen piksel postoji, odnosno da li je:
    - izračunata vrsta manja od visine slike, i
    - izračunata kolona manja od širine slike.

# Primer 2 — množenje matrice (slike) skalarom

# Primer 2 — množenje matrice skalarom

- *pictureKernel*
- Napisati *CUDA* kernel koji vrednost svakog piksela slike množi sa 2.

*CUDA*

# Primer 2 — množenje matrice skalarom

- Kernel:

```
__global__ void PictureKernel(float *d_Pin, float *d_Pout, int height, int width) {

   const int Row = blockDim.y * blockIdx.y + threadIdx.y;
   const int Col = blockDim.x * blockIdx.x + threadIdx.x;

   if (Row < height && Col < width) {
       d_Pout[Row * width + Col] = 2.0 * d_Pin[Row * width + Col];
   }
}
```

# Primer 3 — pisanje kernela za zamućenje slike