Računarski sistemi visokih performansi

Nikola Vukić, Petar Trifunović, Veljko Petrović

Računarske vežbe
Zimski semestar 2024/25.

CUDA

Sadržaj

- *CUDA* programski model.
- Arhitektura uređaja.
- Uobičajeni redosled događaja u toku izvršavanja *CUDA* programa.
- Kernel funkcija:
  - osnovno o kernelu
  - struktura mreže i blokova
- Primeri obrade 1D i 2D nizova.
- Tipovi memorije.
- Optimizacije:
  - zamena globalne deljenom memorijom
  - poravnat pristup globalnoj memoriji
- Literatura.
- Zadaci.

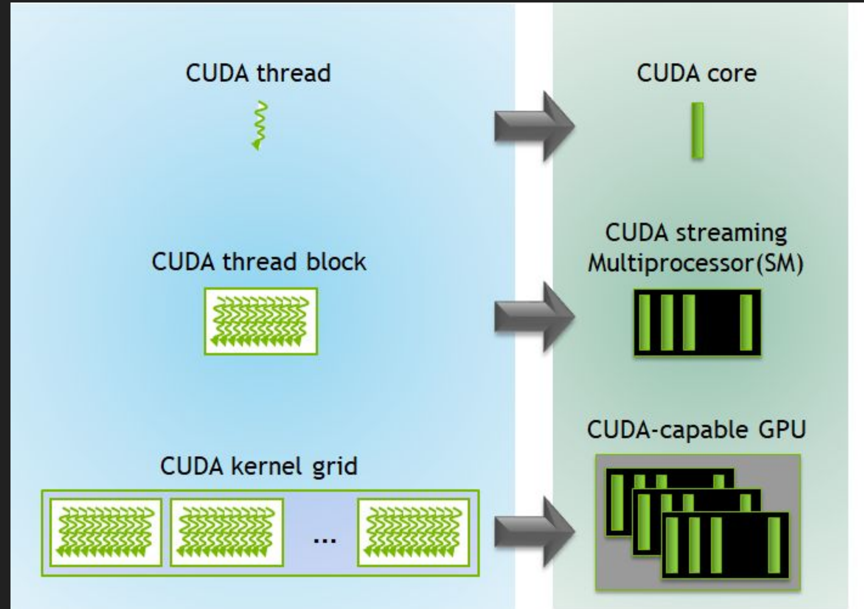
CUDA programski model

Programski model

- Može se napraviti argument da je množenje matrica na GPU-u jedan od najbitnijih algoritama koji postoji, stoga se mora makar pomenuti na ovom predmetu.
- Što se tiče programskog modela CUDA, sastoji se iz tri dela:
  - Domaćin (*host*) — CPU sa svojom memorijom.
  - Uređaj (*device*) — GPU sa svojom memorijom.
  - Kernel — funkcija koja se izvršava **na uređaju**.

Arhitektura uređaja

Arhitektura uređaja



Objašnjenje na narednom slajdu

Arhitektura uređaja

- Izvršavanje CUDA programa podrazumeva pokretanje **blokova niti** (blok sadrži najviše 1024 niti).
- Logički gledano, analogija je sledeća:
    - Nit == skalarni procesor (*CUDA core*).
    - Blok niti ~= *Streaming Multiprocessor,* SM (preslikavanje nije 1 na 1, jedan SM može da sadrži više blokova).
    - Mreža (grid) blokova == *CUDA* uređaj.
    - Na najnižem logičkom nivou, blokovi su podeljeni u *warp*-ove od po 32 niti.

Uobičajeni redosled događaja u toku izvršavanja *CUDA programa*

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

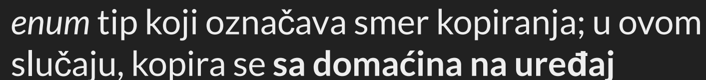
```
cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
```

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);
```



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

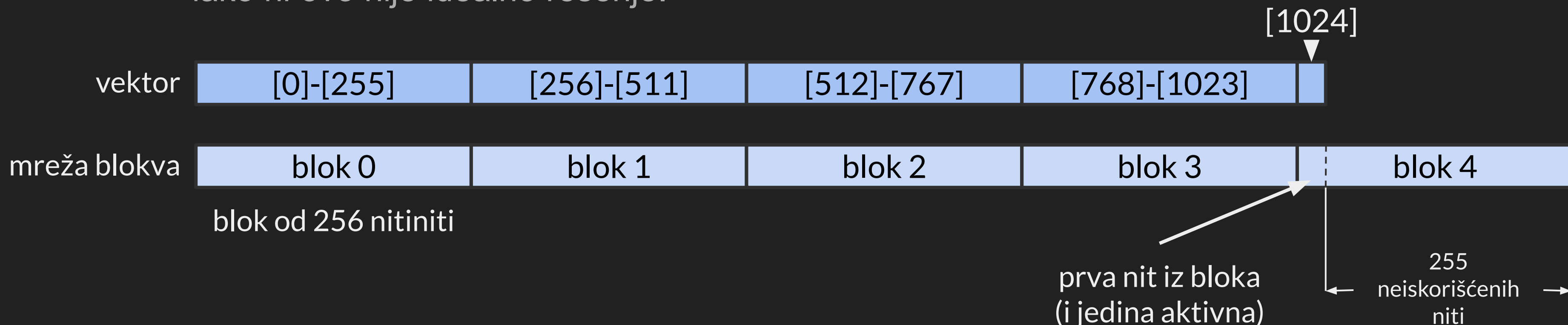
- Recimo da obrađujemo 1D vektor od 1025 elemenata.
- Treba odrediti:
  - veličinu bloka, i
  - veličinu mreže blokova.

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);
```

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);
```

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);
```

5. Dealocirati memoriju na domaćinu i uređaju

- 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 (odgovaraju osama koordinatnog sistema, dakle X ide horizontalno), 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>>>(...);
```

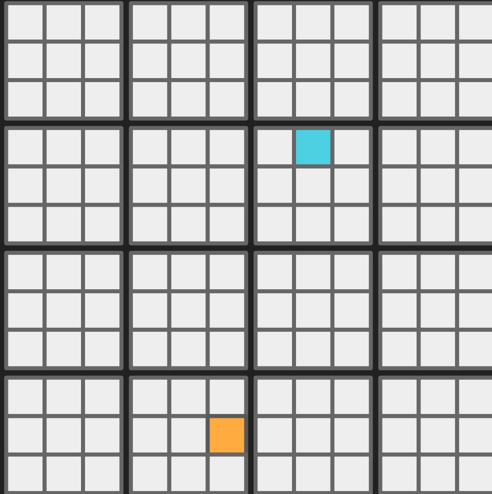
- Broj dimenzija mreže može biti različit od broja diemnzija bloka.

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:



- Vrednosti *CUDA* promenljivih:
  - *gridDim.y: 4*
  - *gridDim.x: 4*
  - *blockDim.y: 3*
  - *blockDim.x: 3*
  - *threadIdx.y: 0*
  - *threadIdx.x: 1*
  - *threadIdx.y: 1*
  - *threadIdx.x: 2*

Primer – sabiranje vektora

- Datoteka *primeri/p02_cuda_vector_add.cu*.
- 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.

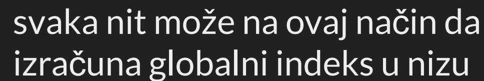
Primer – sabiranje vektora

- 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 – 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 – sabiranje vektora

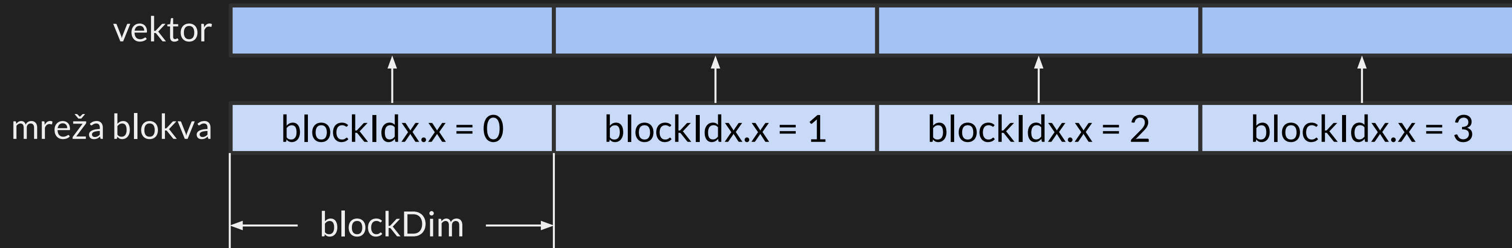
- Izgled kernela:

```
__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, često se nailazi termin “isključivanje viška niti”

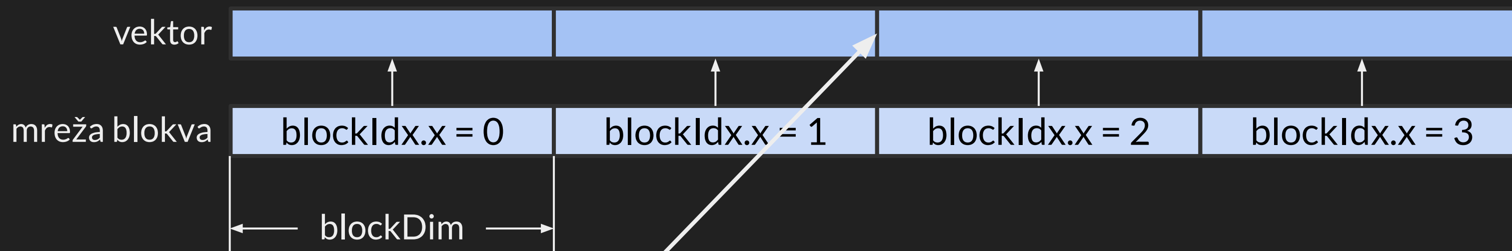
Primer – sabiranje vektora

- 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 – sabiranje vektora

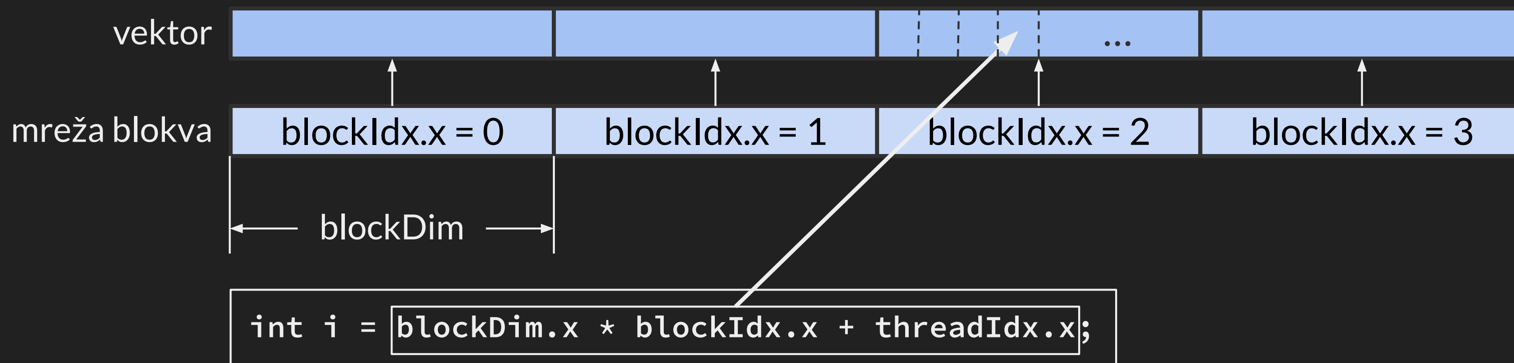
- Prvi deo izraza pronalazi početak dela vektora za koji je zadužen blok niti (prikazan slučaj za nit iz bloka 2).




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

Primer – 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).



Obrada 2D struktura

- *CUDA* se vrlo često koristi za ubrzavanje obrade 2D nizova, odnosno za proračune sa matricama.
- Kako se računanje globalnog indeksa opisano za 1D niz prenosi na 2D nizove?
- Princip računanja je isti, samo treba izračunati dva indeksa:

```
int row = blockDim.y * blockIdx.y + threadIdx.y;
int col = blockDim.x * blockIdx.x + threadIdx.x;
int ind = row * width + col; // računanje indeksa u linearizovanoj matrici
```

- Prikazana računica podrazumeva da se dimenzija Y koristi za vrste, a X za kolone matrice.
- Ovo zavisi od interpretacije programera, ali u ovom primeru bi se prvi parametar u dim3 struktur računao na osnovu broja kolona matrice, a drugi na osnovu broja vrsta.

Primer – množenje matrice skalarom

- Datoteka *primeri/p03_cuda_mat_x_scalar.cu*.
- Postavljanje dimenzija mreže i blokova:

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

Primer – množenje matrice (slike) skalarom

- Datoteka *primeri/p03_cuda_mat_x_scalar.cu*.
- Postavljanje dimenzija mreže i blokova:

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

prva dimenzija mreže zavisi od širine slike, a druga od visine, što znači da će se X dimenzija koristiti za kolone, a Y za vrste

Primer – množenje matrice (slike) skalarom

- Datoteka *primeri/p03_cuda_mat_x_scalar.cu*.
- 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];
   }
}
```

Tipovi memorije

Tipovi memorije – globalna i lokalna

- Globalna:
  - vidljiva svim nitima svih blokova.
  - dostupna domaćinu — za kopiranje memorije (*cudaMemcpy*), za slanje parametara kernelu.
  - spora u odnosu na ostale vidove memorije.
  - planiranim pristupom može se optimizovati njeno korišćenje.
- Lokalna
  - promenljive deklarisane unutar kernela, svaka nit ima sopstvenu kopiju
  - zapravo, izolovan deo globalne memorije; podjednako sporo.

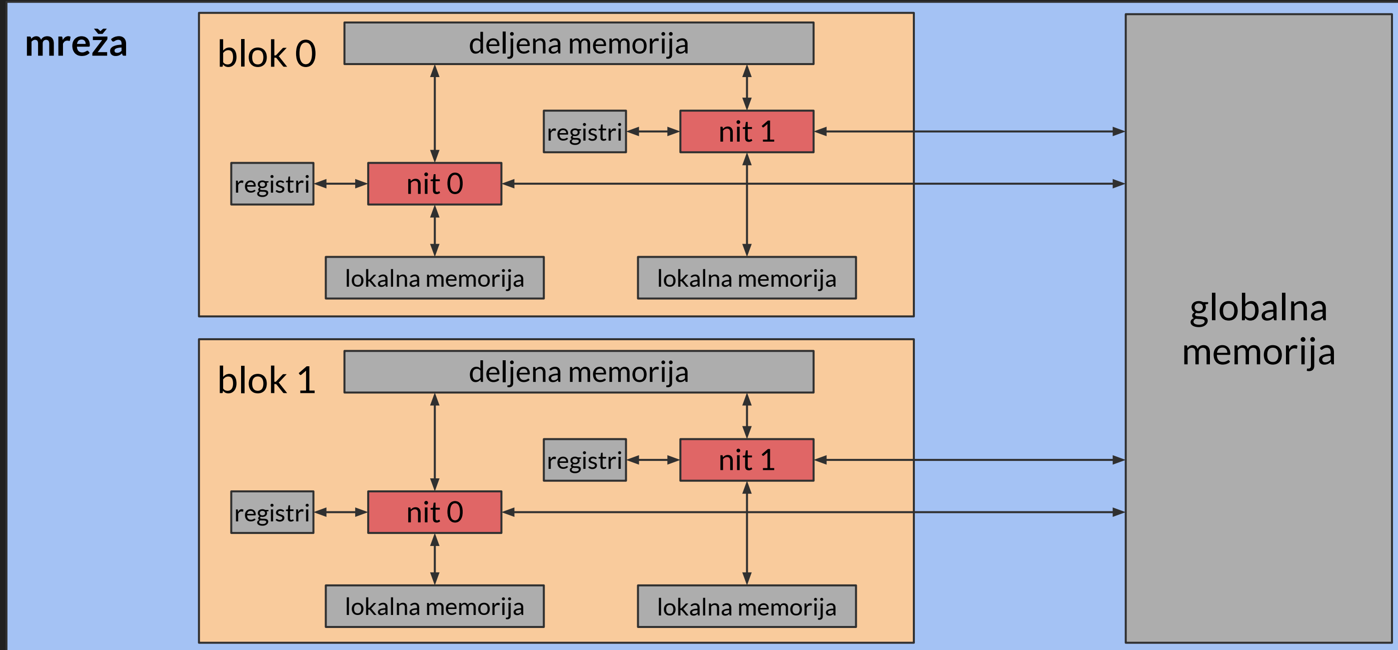
Tipovi memorije – registarska

- Ima efekat lokalne — svaka nit ima svoju privatnu.
- Najbrža, ali je ima najmanje.
- Skalarne promenljive i nizovi konstantne veličine **uglavnom** se smeštaju u registarsku memoriju ukoliko su deklarisani unutar kernela.
- **Uglavnom** — ne uvek; ako ih bude previše, preliće se u lokalnu memoriju.
- Ukoliko ih nema previše, svakako ih je dobro koristiti.

Tipovi memorije – deljena

- Postoji na nivou **bloka niti**.
- Sve niti istog bloka vide istu deljenu memoriju, ali nemaju pristup deljenoj memoriji drugih blokova.
- Brža od globalne.
- Deklarisana unutar kernela pomoću oznake `__shared__`.

Tipovi memorije



Optimizacije

Optimizacije – zamena globalne deljenom memorijom

- Zarad optimizacije rada *CUDA* programa, potrebno je minimizovati pristupe globalnoj memoriji, što se može izvesti pametnom upotrebom deljene memorije.
- U opštem slučaju, može se smatrati da ovo zahteva nekoliko uobičajenih koraka:
  1. Pristupiti glavnoj memoriji na početku kernela zarad učitavanja odgovarajućeg skupa podataka iz globalne u deljenu memoriju.
  2. Pri glavnoj obradi, pristupati podacima u deljenoj memoriji, umesto u globalnoj.
  3. Upisati rezultate obrade u globalnu memoriju, kako bi bili dostupni domaćinu.
- Ovi koraci se modifikuju kako bi se prilagodili konkretnim potrebama programa.
- Ovaj vid optimizacije će dati najveće benefite ukoliko je algoritam *memory-bound,* a globalnoj memoriji se često pristupa, na primer, naivni pristup množenja matrica.

Optimizacije – zamena globalne deljenom memorijom

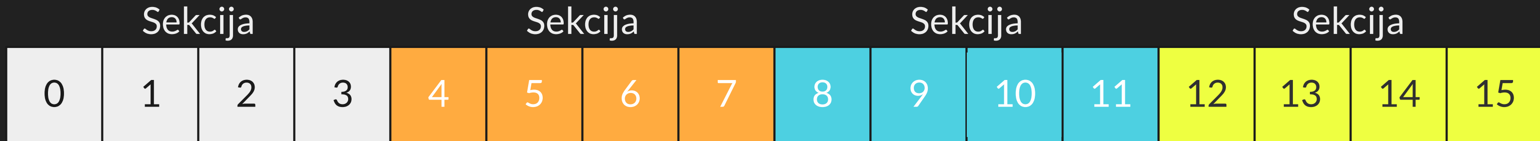
- Kada su 2D nizovi (matrice) u pitanju, princip popločavanja (engl. *tiling*) je poznat u *CUDA*-i kao tehnika optimizacije koja minimizuje pristup globalnoj na račun deljene memorije.
- Ovo se u osnovi radi tako što se učitavaju *delovi* (koji su dovoljno mali da mogu stati u deljenu memoriju) iz *globalne* pre nego što se vrše proračuni, zatim sabiranjem delova dolazi se do konačnog proizvoda.
- Više o ovoj temi možete naći na sajtu *acs*-a u repozitorijumu za predmet *Paralelno računarstvo* (*Vežbe -> CUDA -> CUDA2*).
- Tehnika je opšte poznata u *CUDA* zajednici, pa se materijali o ovoj temi mogu naći i *online*.

Optimizacije – poravnat pristup globalnoj memoriji

- Pristup globalnoj memoriji je svakako neizbežan, makar pri čitanju iz ulaznih parametara i upisu u izlazne parametre.
- Taj pristup koji se ne može izbeći, može se optimizovati poravnanjem pristupa (engl. *coalesced access*).

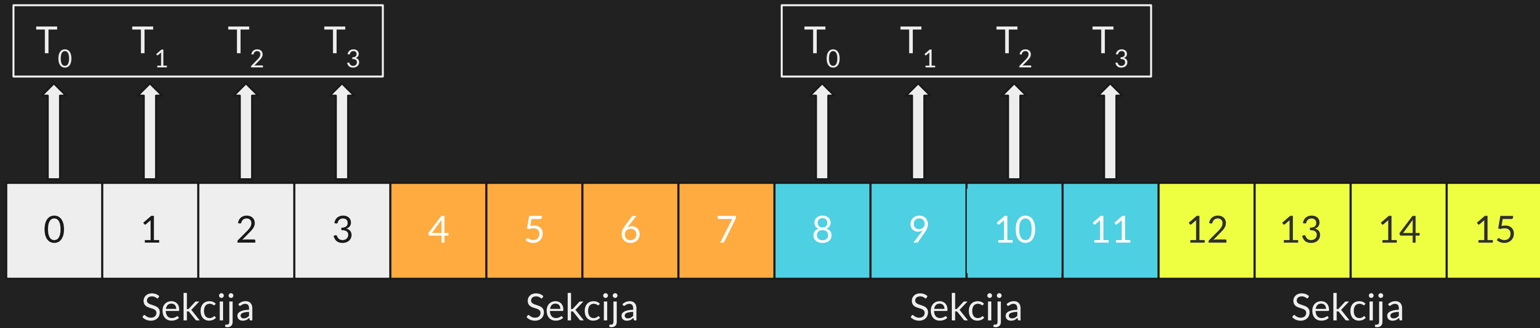
Optimizacije – poravnat pristup globalnoj memoriji

- Pristup DRAM memoriji:



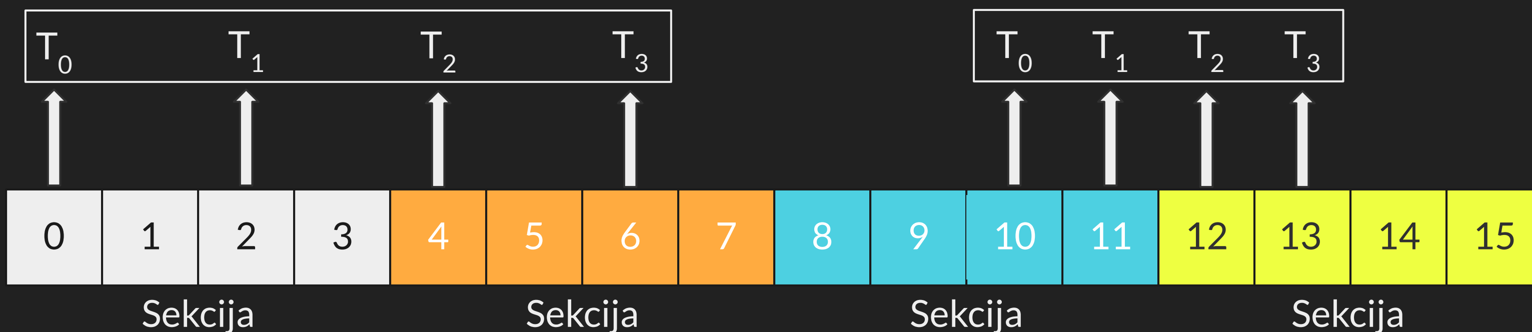
- Adresni prostor memorije je podeljen u sekcije.
- Kada se pristupa jednoj lokaciji, prenose se vrednosti svih memorijskih lokacija iz sekcije (slično kao čitanje iz keša, setimo se lažnog deljenja).
- Primer: 16-bajtni adresni prostor, 4-bajtne memorijske lokacije.

Optimizacije – poravnat pristup globalnoj memoriji



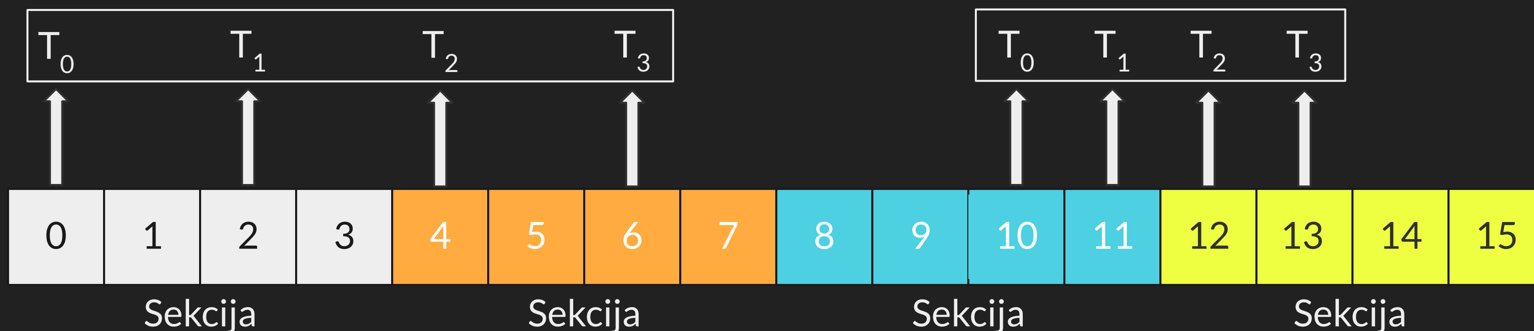
- Kada sve niti iz iste osnove (engl. *warp*) izvršavaju instrukciju učitavanja podataka, ako sve lokacije kojima se pristupa pripadaju istoj sekciji, pristup je **poravnat**.

Optimizacije – poravnat pristup globalnoj memoriji



- Kada se lokacije kojima pristupaju niti iz iste osnove nalaze u različitim sekcijama, pristup **nije poravnat**, slično kao i pri učitavanju keš linija kod CPU-a.
- Potrebno je poslati više zahteva za učitavanje podataka, a niti neke od učitanih vrednosti neće ni koristiti.

Optimizacije – poravnat pristup globalnoj memoriji



- Na gornjoj slici, kod prvog skupa niti problem je postojanje razmaka između uzastopnih niti (engl. *stride*).
- Kod drugog skupa niti problem je udaljenost od početka sekcije (engl. *offset*).

Optimizacije – poravnat pristup globalnoj memoriji

- Na novijim arhitekturama, *stride* je znatno veći problem od *offset*-a, s obzirom na to da se problem *offset*-a rešava uvođenjem keša sa velikom keš linijom, pa je *offset* problem u malom broju graničnih slučajeva.
- Veliki *stride* je problem bez obzira na postojanje keša, s obzirom na to da razmak između uzastopnih elemenata može da bude dovoljno veliki da pobegne iz keš linije.
- Dobro objašenjenje ovog problema dato je na sledećem linku: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/.
- Deljena memorija može do određene mere da pomogne i kod ovog problema.
- Više o ovoj temi možete naći na sajtu *acs*-a u repozitorijumu za predmet *Paralelno računarstvo* (*Vežbe -> CUDA -> CUDA3*).

Literatura

Literatura

- Glavni izvor za ovu prezentaciju su slajdovi sa vežbi sa predmeta *Paralelno računarstvo* i oni se mogu iskoristiti za dalje istraživanje o ovoj temi (sajt *acs*-a, repozitorijum).
- U repozitorijumu predmeta (RSVP) nalazi se pdf sa linkovima ka *Google Colab* sveskama koje sadrže dodatne primere, zadatke i rešenja. Preporuka je pre njihovog otvaranja pogledati materijale sa predmeta *Paralelno računarstvo*.
- Serija blog postova dostupna na linku: https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/ je dobra za učenje *CUDA*-e, jer kreće od početnog nivoa i zalazi u osrednji nivo rada sa *CUDA*-om.
- Napredno: https://siboehm.com/articles/22/CUDA-MMM ukoliko neko želi stvarno da se udubi u optimizacije, može da prođe kroz taj blog.