

# Programiranje na grafičkoj kartici

Marko Stanojević

Matematička gimnazija

23. 04. 2021.

- + Napredak hardvera je postao sve neprimetniji
  - + nekada se umesto poboljšanja softvera jednostavno kupovao bolji hardver
- + Takođe, napredak u kompajlerskim tehnologijama ne ubrzava značajno softver
  - + u prošlosti je poboljšanje kompajlera donosilo mnogo kvalitetniji kod
- + Softver je postao dosta komplikovan
  - + i biće sve složeniji!
- + Kako onda danas ubrzati softver?
  - + kroz paralelizaciju
  - + pametnijim iskorišćenjem resursa
  - + izvršavanjem dela koda na namenskom hardveru

- + Pojava GPGPU
  - + grafičke kartice su ranije korišćene samo za crtanje primitiva na ekran
  - + postale programabilne 2000-ih godina
  - + danas GPGPU ima jako raznovrsne primene

# Gde se koristi GPU programiranje?



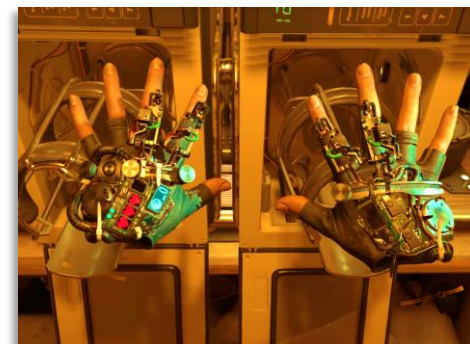
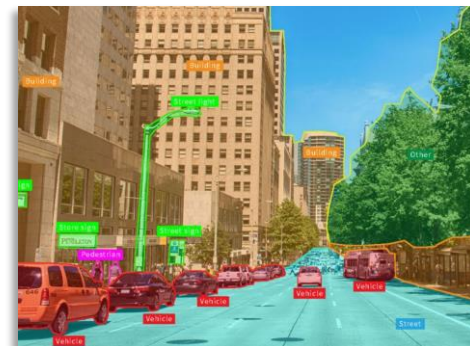
- Za video igre
  - + omogućava da igre budu real-time i da u isto vreme pruže uverljiv osećaj stvarnosti
- U filmskoj industriji
  - + pravljenje fotorealističnih animacija
- + U fizici, medicini, hemiji, astrofizici, ...
  - + za simulacije klime, molekula, proteina, kvantnih pojava
- U 3D modelovanju
  - + modelovanje sistema sa puno delova i interakcije



# Gde se koristi GPU programiranje? ii



- + U obradi slike, zvuka i videa
  - + primena filtera radi poboljšanja kvaliteta
  - + uklanjanje drugih izvora zvuka osim govornika
- Za computer vision
  - + razumevanje prostora i objekata u njemu
- + Za data mining
  - + otkrivanje korisnih zavisnosti u velikoj količini podataka
- U virtual i artificial reality-ju
  - + pravljenje uverljivog sveta / dodavanje objekata u stvarni svet
  - + generisanje zvuka tako da deluje stvarno



# Gde se koristi GPU programiranje? iii



- + Za rudarenje nekih kriptovaluta
- ...
- + Uopšteno, kada je potrebno obraditi veliku količinu podataka na vrlo regularan način
- + U daljem tekstu se koristi CUDA standard (NVIDIA)
  - + postoji i open-source OpenCL standard (AMD, NVIDIA), ali je trenutno malo slabiji od CUDA-e prema mogućnostima



# Arhitektura GPU-a i programski model: Niti

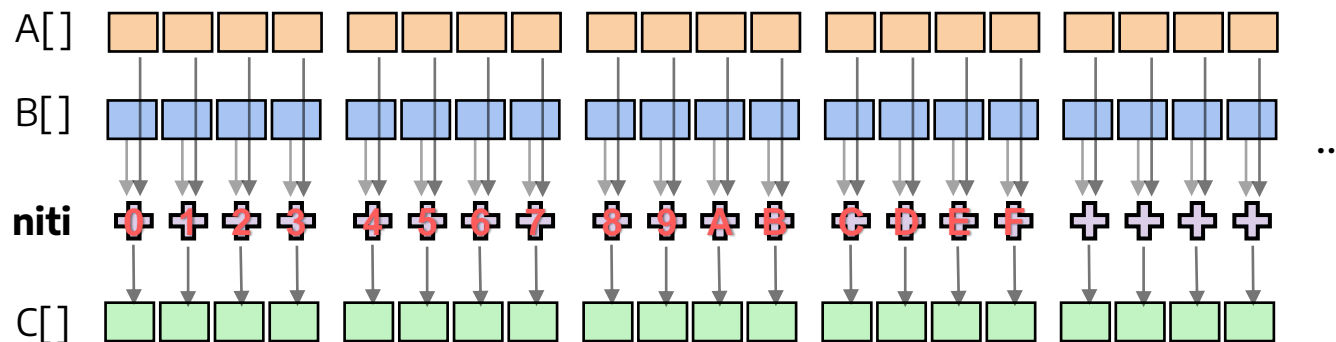
# [Niti] Jednostavan primer



- + Primer - sabrati nizove A[] i B[] dužine len u niz C[]
- + prilično jednostavno na CPU-u

```
1  for( int i = 0; i < len; i++ )  
2    C[i] = A[i] + B[i];
```

- + Primetiti da su iteracije petlje nezavisne!
- + može mnogo više zbrova istovremeno
- + u idealnom slučaju bi nizovi mogli da se saberu u jednoj instrukciji





# (Niti) Ultra-paralelno izvršavanje



- + Ideja - hajde da zamislimo da imamo mnogo paralelnih niti ("puno ljudi")
  - + sve niti vide iste globalne podatke – memoriju gde su nizovi A[], B[] i C[]
  - + svaka nit zna svoj jedinstveni indeks, neka se niti numerišu redom (0, 1, 2, ...)

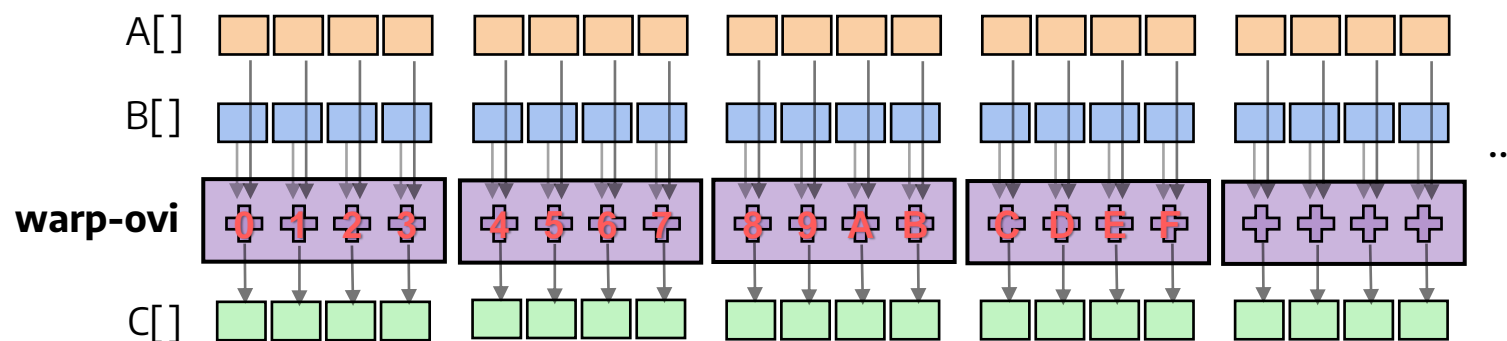
```
1 C[index] = A[index] + B[index];
```

- + Bitno - sve niti koriste isti programski kod!
  - + sam programski kod koji izvršava nit se naziva kernel
- + Primiti da nema nigde for ciklusa
  - + spoljašnji for ciklus se zameni velikim brojem niti
  - + jedna nit se mapira na jednu iteraciju petlje

# [Niti] Grupisanje niti u warp



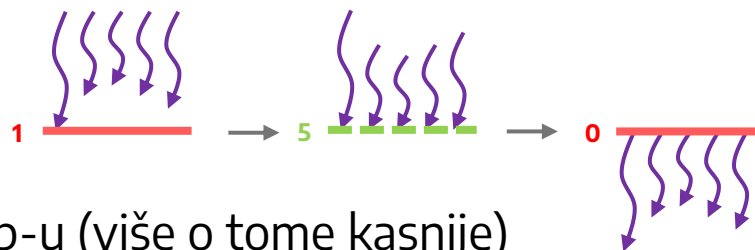
- + Mana - hardver za toliko paralelnih niti bi bio preskup i slabo iskorišćen
  - + ako bi sve niti izvršavale istu instrukciju istovremeno!
- + Rešenje - grupišimo niti u warp-ove
  - + niti u jednom warp-u se uvek izvršavaju istovremeno
  - + warp-ovi se slobodno mešaju tokom izvršavanja
  - + 32 niti u warp-u kod NVIDIA-je, 64 niti kod AMD-a



# [Niti] Sinhronizacija niti



- + Digresija - sinhronizacija je dogovor između više niti bez razmene podataka
  - + analogija "dogovor da se vidimo sa prijateljima u restoranu"
    - + da bi stvarna zabava počela, potrebno je da svi dođu na isto mesto
    - + posle provoda se svi opet razilaze



- + Ova sinhronizacija se naziva **barijera**
  - + može biti za niti u bloku ili\* za niti u warp-u (više o tome kasnije)
- + Problem je što ima previše niti da bi se lako sinhronizovale
  - + hardver bi bio previše složen i **neskalabilan**

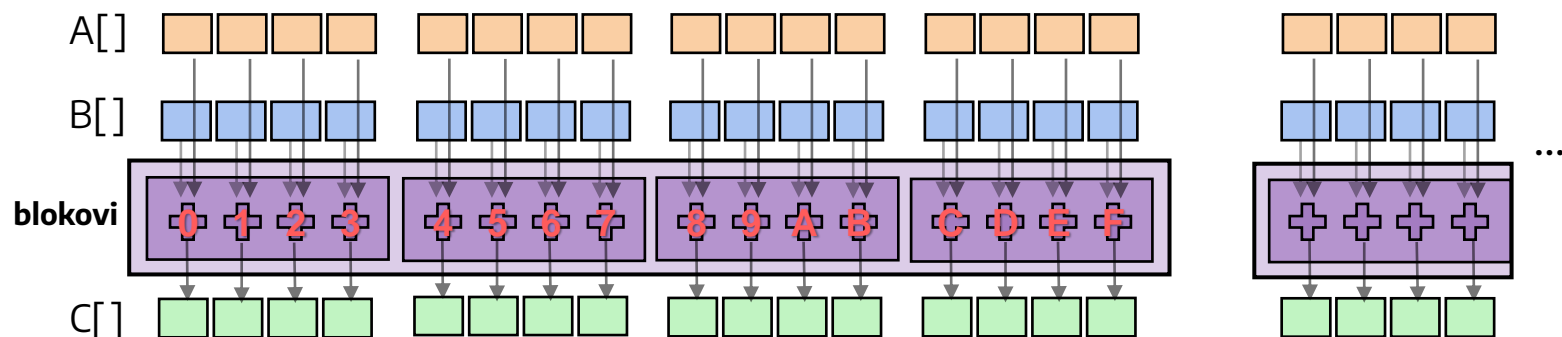
\* postoji i na nivou grid-a, ali onda ukupan broj blokova ne može da premaši raspoložive resurse

# [Niti] Grupisanje warp-ova u blok



- + Rešenje je: napraviti grupe warp-ova, takozvani **blok** niti
- + niti u istom bloku mogu da se sinhronizuju
- + međutim, niti u različitim blokovima se ne vide! – nezavisni su
- + blokovi niti se takođe numerišu redom (0, 1, 2, ...), kao i same niti u bloku
- + veličina bloka se zada unapred i fiksna je za jedno izvršavanje

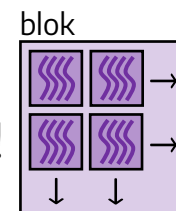
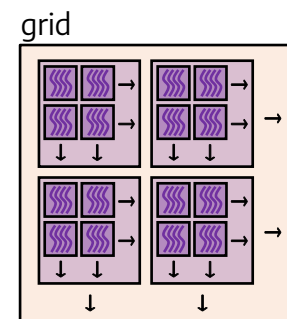
```
1 int index = BlockIdx.x * BlockDim.x + ThreadIdx.x;  
2 C[index] = A[index] + B[index];
```



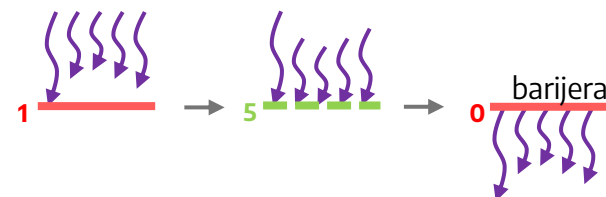
# [Niti] Pregled niti i blokova



- + **Grid** je 1D/2D/3D grupa blokova
- + **Blok** je 1D/2D/3D grupa niti koje mogu da se dogovaraju
  - + blok je tačno deljiv na warp-ove
- + **Warp** je fiksna podgrupa niti koje izvršavaju istu instrukciju istovremeno!
  - + warp je hardverski detalj, ali može da bude koristan
- + **Nit** izvršava kernel (običnu C/C++ funkciju)
  - + brzo se pravi i uklanja kada završi kernel
- + **Barijera** je mesto u programskom kodu gde se sve! niti u bloku/warp-u sastanu



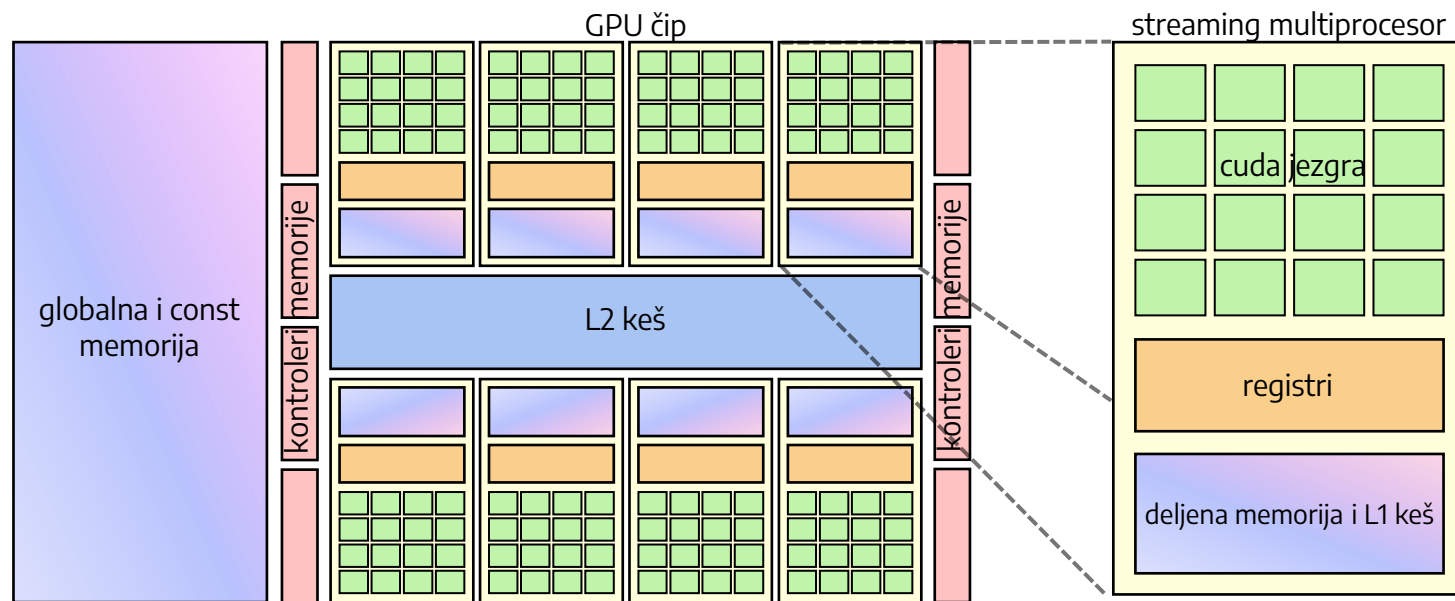
nit  
~



# (Mapiranje niti) Streaming multiprocesor



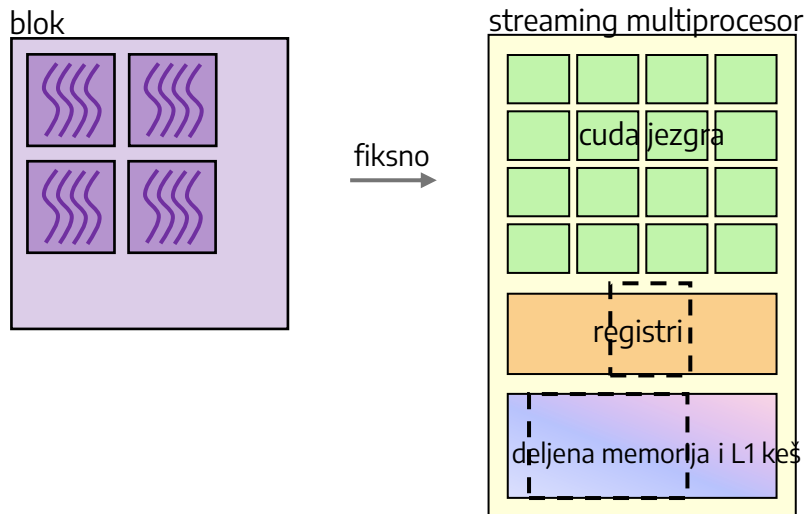
- + GPU čip se sastoji od puno **streaming multiprocesor**-a (SM)
  - + takođe sadrži globalnu memoriju i L2 keš (više o tome kasnije)
- + Streaming multiprocesor je organizaciona jedinica GPU čipa
  - + sadrži puno **cuda jezgara**, koji su analogni pojednostavljenim CPU core-ovima
  - + takođe sadrži registre i deljenu memoriju (ali više o tome kasnije)



# (Mapiranje niti) na hardver



- + Mapiranje je rezervisanje resursa (scheduling)
- + neki blok se mapira na neki streaming multiprocesor
  - ⇔ taj SM rezerviše za taj blok deo registara, deljene memorije, ...

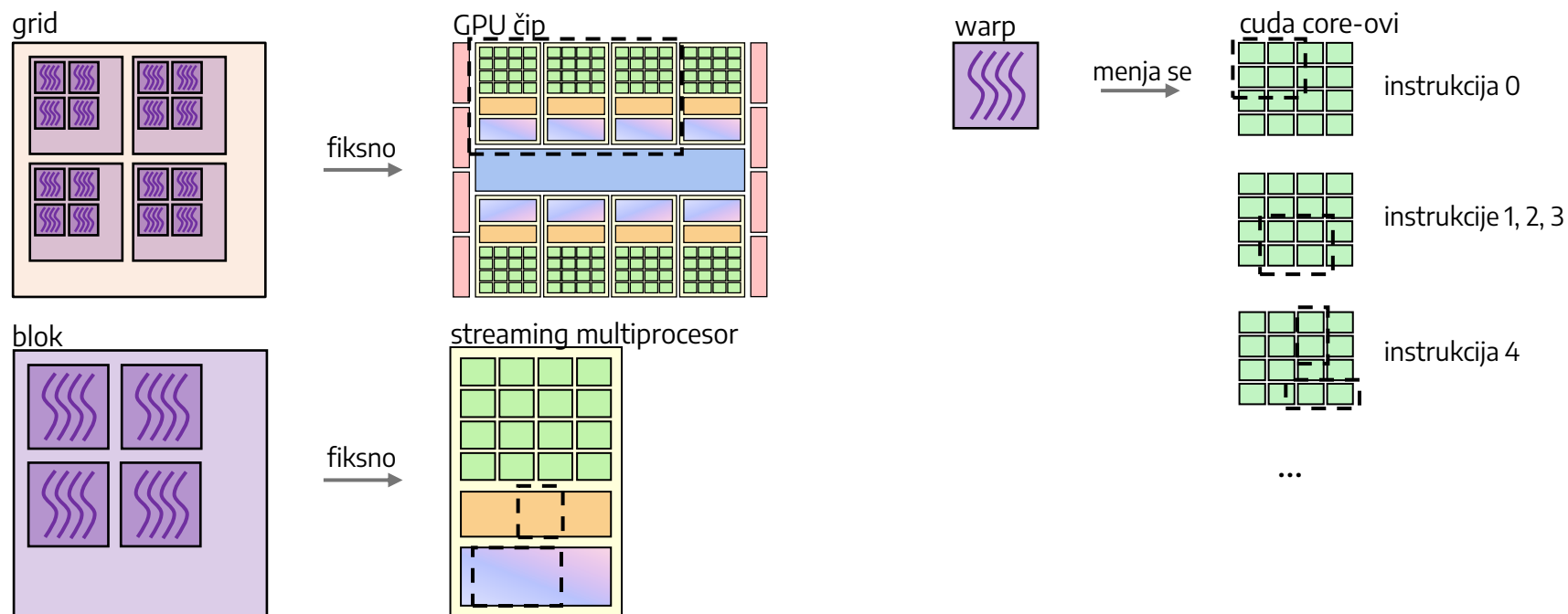


- + GPU čip radi mapiranje blokova
- + SM bira warp-ove koji će se sledeći izvršavati

# (Mapiranje niti) na hardver ii



- + Grid se mapira na jedan ili više GPU-a
- + Blok se mapira na tačno jedan streaming multiprocesor
- + Warp se mapira na neka 32\* cuda core-a
- + Pojedinačna nit se ne mapira!



\* Ako core ne izvršava više niti istovremeno, to zavisi od arhitekture core-a



# (Mapiranje niti) na hardver iii



- + Na jednom streaming multiprocesoru može biti nekoliko blokova
  - + bitno - kada se blok mapira na multiprocesor i učitava, ostaje tu dok se sve! niti u njemu ne završe
  - + ako trenutno nema dovoljno resursa, novi blok će sačekati da bude mapiran
- + Warp-ovi se slobodno mešaju u toku izvršavanja
  - + warp čeka na izvršenje sledeće instrukcije samo ako je spreman
  - + čim aktivni warp nema sve spremno za sledeću instrukciju, SM ga brzo zameni drugim spremnim warp-om
- + Drugi naziv za tu je *zero-overhead scheduling policy*
  - + tako se sakrivaju razna kašnjenja, jer obično ima mnogo spremnih warp-ova u bilo kom trenutku

- + GPU core je "pojednostavljen" CPU core
  - + nije namenjen za rešavanje generalnih problema kao CPU, već je specijalizovan
  - + ima mnogo manje raznovrsan skup instrukcija i dosta slabiju kontrolu toka
- + GPU je mnogo paralelan u odnosu na CPU-a
  - + primer - oko 4500 core-ova umesto 16 core-ova (za personalne računare)
  - + "nekoliko konja ili hiljade skakavaca protiv njive"
- + CPU smanjuje kašnjenje dobrim predviđanjem
  - + ima puno keševa, branch prediktor, spekulativno izvršavanje, ...
- + GPU sakriva kašnjenje sa velikim paralelizmom
  - + ima mnogo spremnih warp-ova u bilo kom trenutku

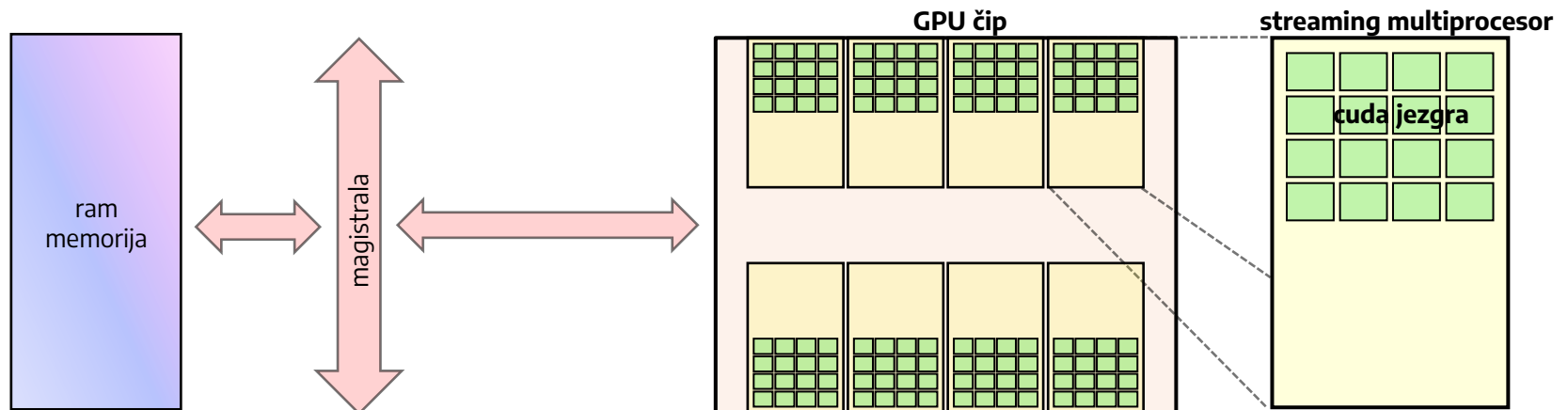


## Arhitektura GPU-a i programski model: Memorija

# [Memorija] Svrha memorije



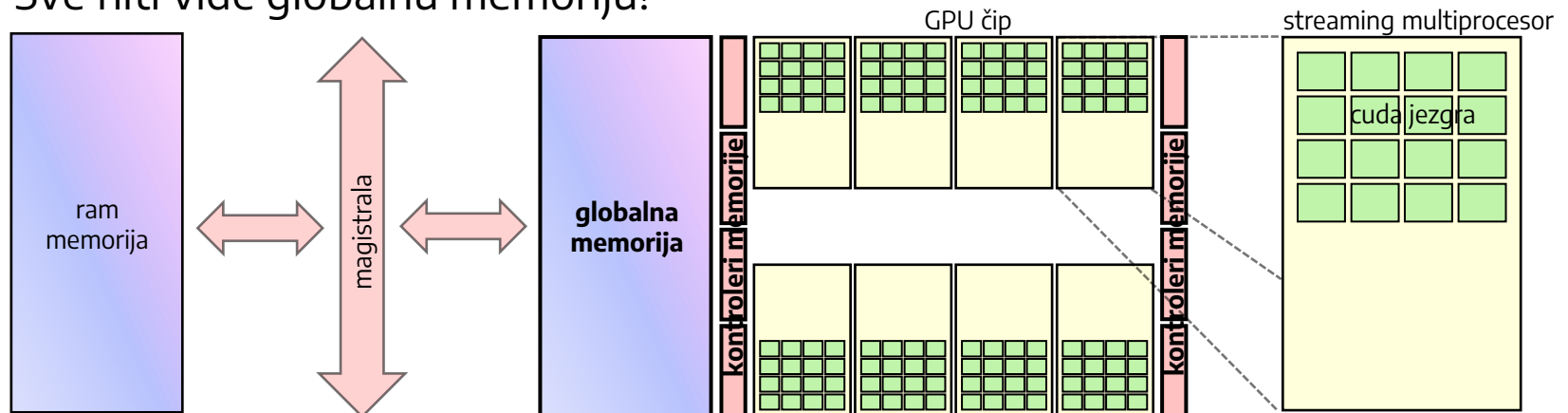
- + Cilj memorije je da bude velika i dosta jeftinija od registara
  - + čuva mnogo više podataka, ali je sporija!
- + GPU bi mogao da koristi RAM memoriju za rad, ali:
  - + RAM memorija nije praviljena tako da može da podrži mnogo istovremenih pristupa
  - + propusni opseg RAM-a je mnogo mali (za GPU)
  - + pristup RAM memoriji je previše spor (za GPU)



# [Memorija] Globalna memorija



- + Ideja - GPU treba da ima svoju "RAM" memoriju – globalnu memoriju
  - + tu memoriju vidi i CPU i GPU, međutim mnogo je bliža GPU-u
  - + globalna memorija ima mnogo veći propusni opseg od RAM-a
- + CPU i GPU dele posao prenošenja i obrade podataka
  - + CPU prenosi podatke iz RAM memorije u globalnu memoriju i obrnuto
  - + GPU radi samu obradu podataka iz globalnoj memoriji
- + Sve niti vide globalnu memoriju!



# (Memorija) Lokalnost podataka

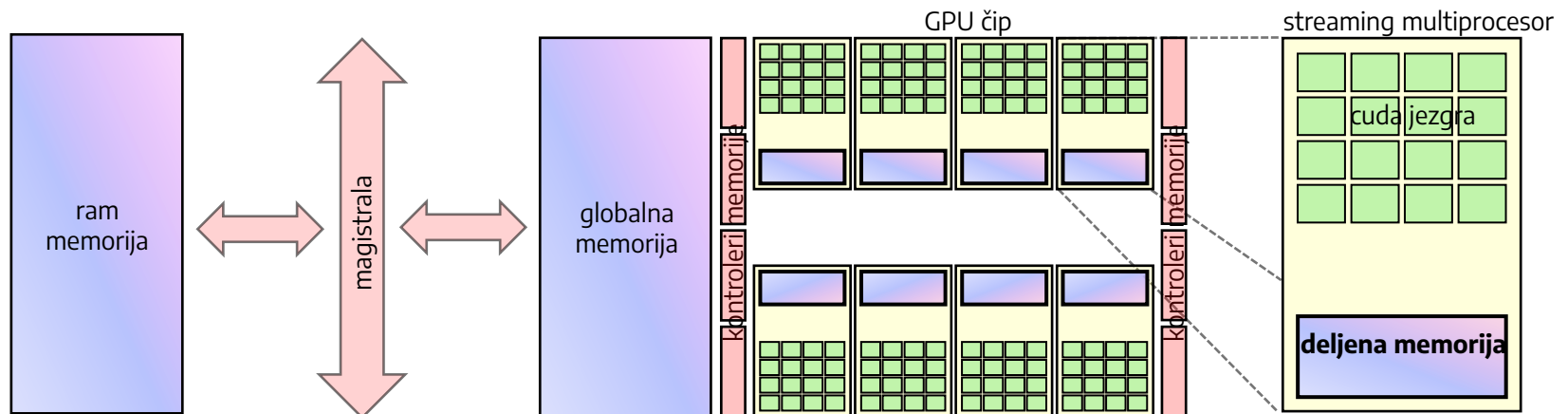


- + Mana - ne koristi se **lokalnost podataka**
  - + često je potreban samo mali deo svih podataka za izračunavanje u okviru jednog bloka niti
  - + bilo bi mudro kada bi te podatke približili samom bloku, tj. SM-u
  - + inače bi SM morao da istom podatku pristupa više puta
- + Takođe, globalna memorija je blago "spora"
  - + 200-tinak instrukcija bi moglo biti izdato umesto jednog pristupa toj memoriji

# [Memorija] Deljena memorija



- + Rešenje je **deljena (shared) memorija**
  - + streaming multiprocesor ima sopstvenu deljenu memoriju
  - + ona je 100x brža od globalne memorije
- + Deljena memorija se rezerviše na nivou bloka niti
  - + sve niti u jednom bloku vide njihov zajednički rezervisani deo
  - + sam blok niti treba da prenese podatke koje će koristiti iz globalne u svoju deljenu memoriju



# [Memorija] Deljena memorija ii



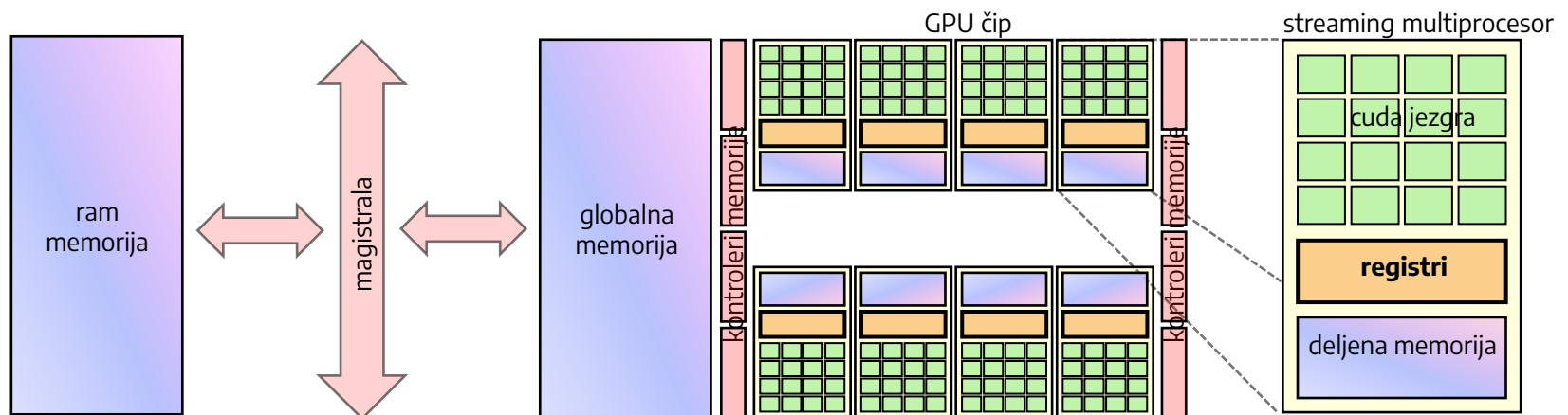
- + Deljena memorija  $\equiv$  keš koji kontroliše programer!
  - + neobično jer je cilj keša da bude nevidljiv programeru
  - + (i da ubrza pristup kodu i podacima koji pokazuju lokalnost)
- + Postoje i keševi koje ne vidi programer!



# [Memorija] Registri



- + Registarski fajl - niz **registara** u streaming multiprocesoru
- + tu se drže podaci koji se trenutno koriste u izračunavanju
- + to je resurs koji je najvredniji i koga ima najmanje (samo 64KB)
- + registre dodeljuje kompajler prilikom prevođenja koda



# [Memorija] Manjak registara za nit

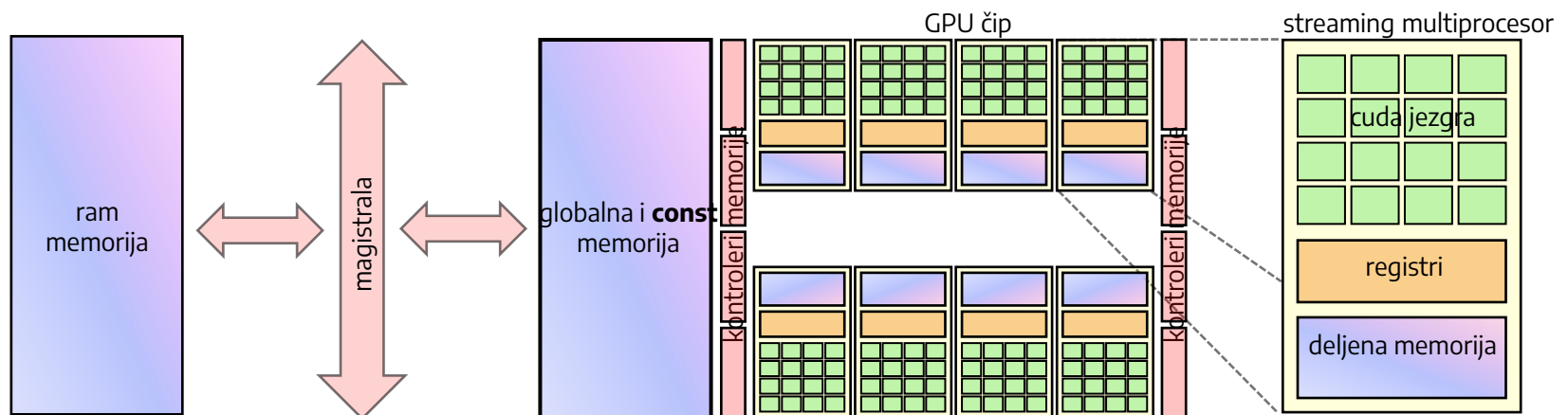


- + Može da se dogodi da nema dovoljno registara za nit
- + To zna kompajler u toku prevođenja!
  - + on neke registre oslobodi, a te podatke sačuva u deljenoj memoriji
  - + drugim rečima, uradi *register spilling*
  - + to ume da jako smanji performanse kernela
- + Programer treba da se čuva toga!
  - + može da se vidi koliko registara je kompajler morao da "izmisli" za dati kernel

# [Memorija] Još "globalnih" memorija



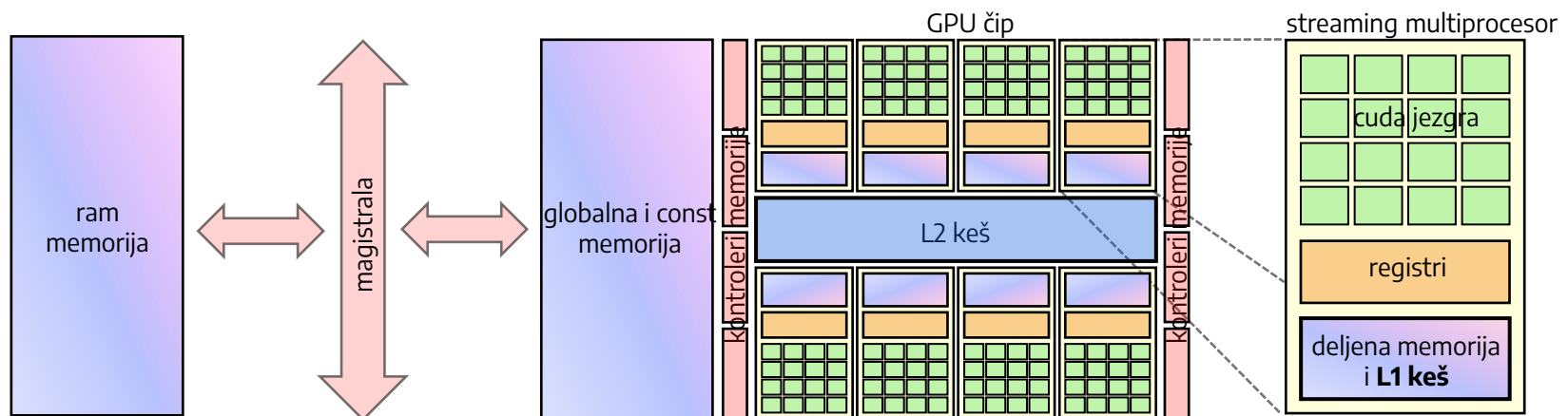
- + Konstantna memorija
  - + slična globalnoj memoriji, ali može samo da se čita
  - + korisna je ako mnogo niti treba da koristi istu konstantu
    - + štedi registre i deljenu memoriju
- + Texture memorija
  - + slična const memoriji, specijalizovana za pristup teksturama



# [Memorija] Keševi



- + **L2 keš** – svim nitima ubrzava pristup globalnoj memoriji
  - + tako što čuva podatke koji su SM-ovi nedavno koristili
  - + brži je od globalne memorije 100x
- + **L1 keš** – nitima u SM-u ubrzava pristup konstantnim! podacima
  - + tako što čuva konstantne podatke iz globalne memorije u samom SM-u
  - + brži je od konstantne memorije 100x

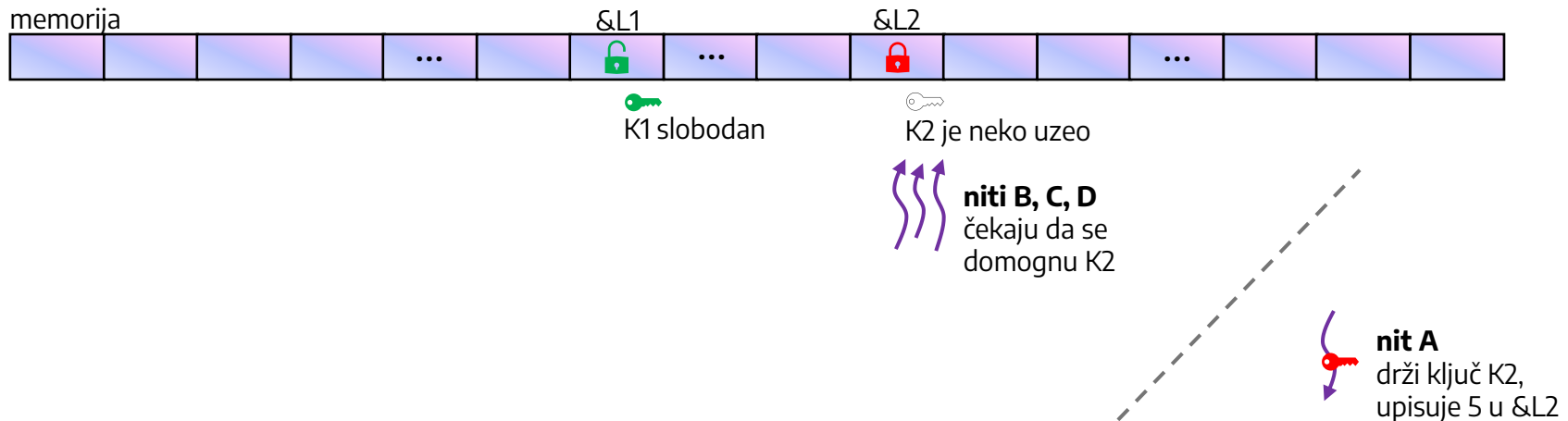


# [Memorija] Atomični pristup



- + Ponekad je korisno da ne čekaju sve niti na barijeri
  - + ukoliko niti pristupaju dovoljno slučajno **globalnoj/deljenoj** memoriji
  - + za to se koriste atomične operacije

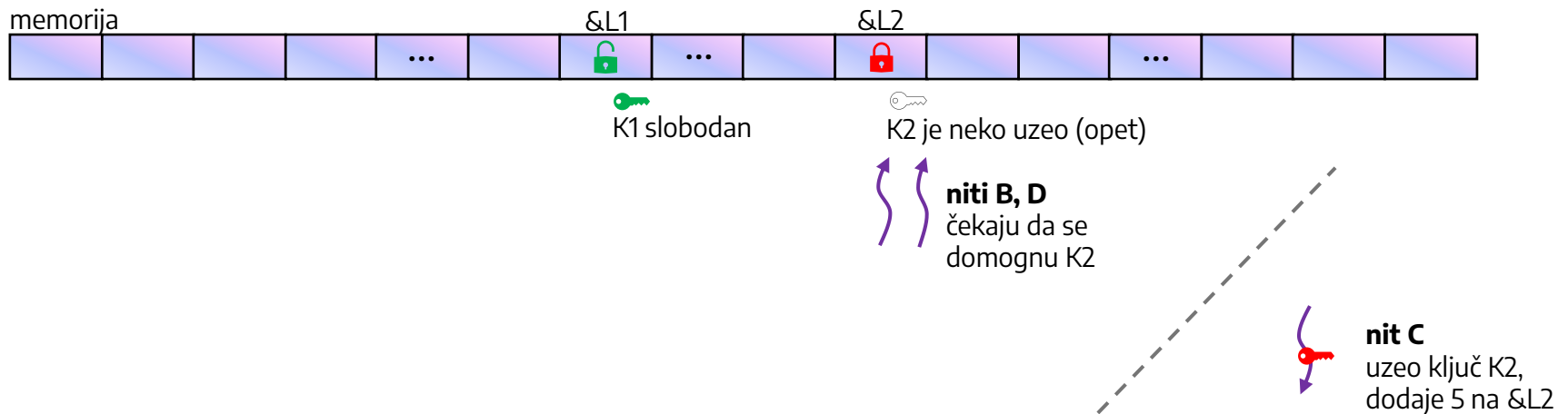
- + Kada nit radi neku **atomičnu** operaciju nad nekom lokacijom, sve druge niti koje probaju da atomično! pristupe toj lokaciji čekaju



# [Memorija] Atomični pristup ii



- + Kada nit "oslobodi" tu lokaciju
- + jedna od čekajućih niti će prva! uspeti da pristupi atomično, ostale će opet čekati
- + analogija "jedan ključ po ormariću u školi"



- + Atomične operacije su skupe! – ne treba ih koristiti ako postoje bolje opcije

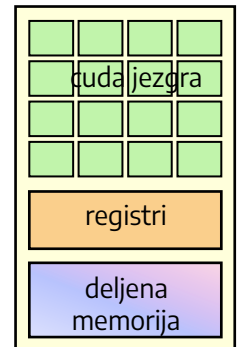
# [Memorija] Pregled memorija



- + U **globalnu** memoriju CPU dovodi podatke i čita rezultate
  - + GPU vrši sama izračunavanja koristeći te podatke
  - + nju kešira L1 keš
- + **Deljenu** memoriju koristi programer da bi bloku približio podatke iz globalne memorije
- + **Konstantna** memorija je slična globalnoj ali je read-only
  - + L2 keš ubrzava pristup bloka niti toj memoriji
- + Nit koja **atomično** pristupi lokaciji "pauzira" druge niti koje bi da to isto urade

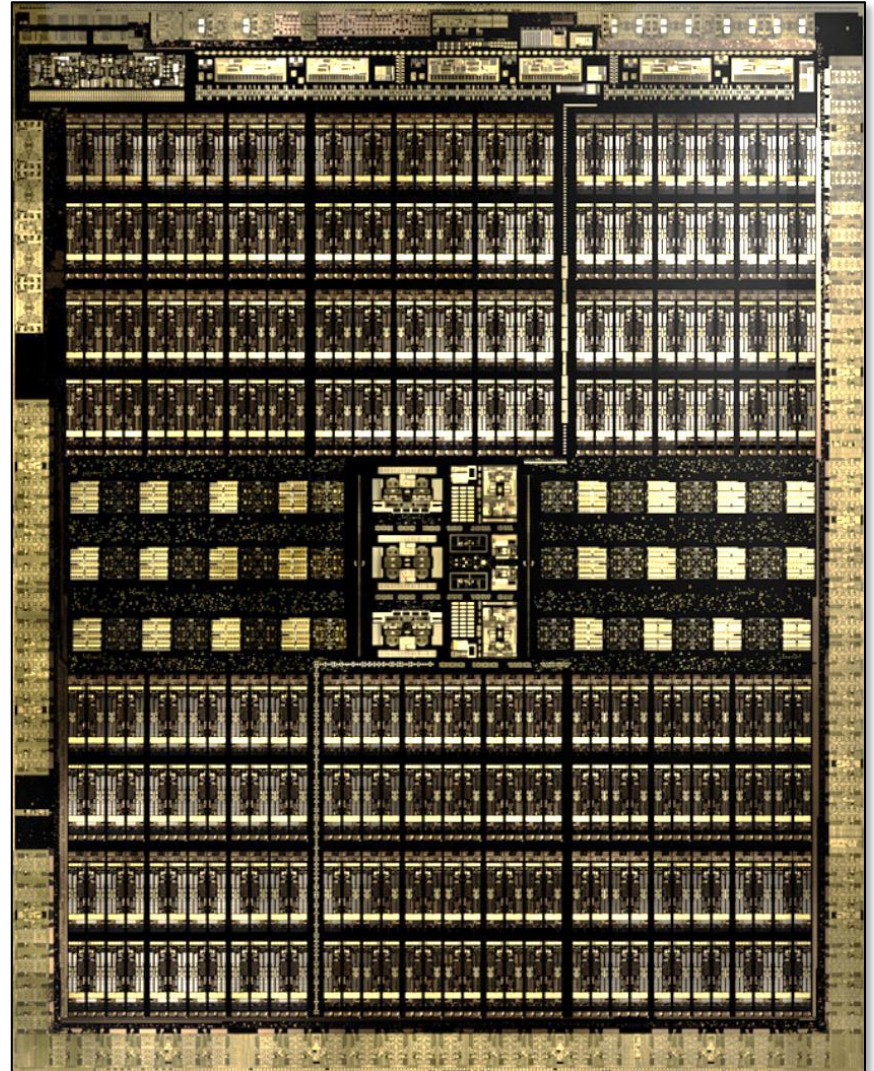


streaming multiprocessor



# Primer arhitekture – Turing

- + Na slici je Turing arhitektura
  - + najavljena je u februaru 2019.
- + Sve varijante GPU čipova sa Turing arhitekturom:
  - + TU102, TU104, TU106, TU116, TU117
- + Neke grafičke kartice sa Turing arhitekturom:
  - + GeForce GTX 1660
  - + GeForce RTX 2080 Ti
  - + Quadro RTX 8000
  - + Tesla T4
  - + ...



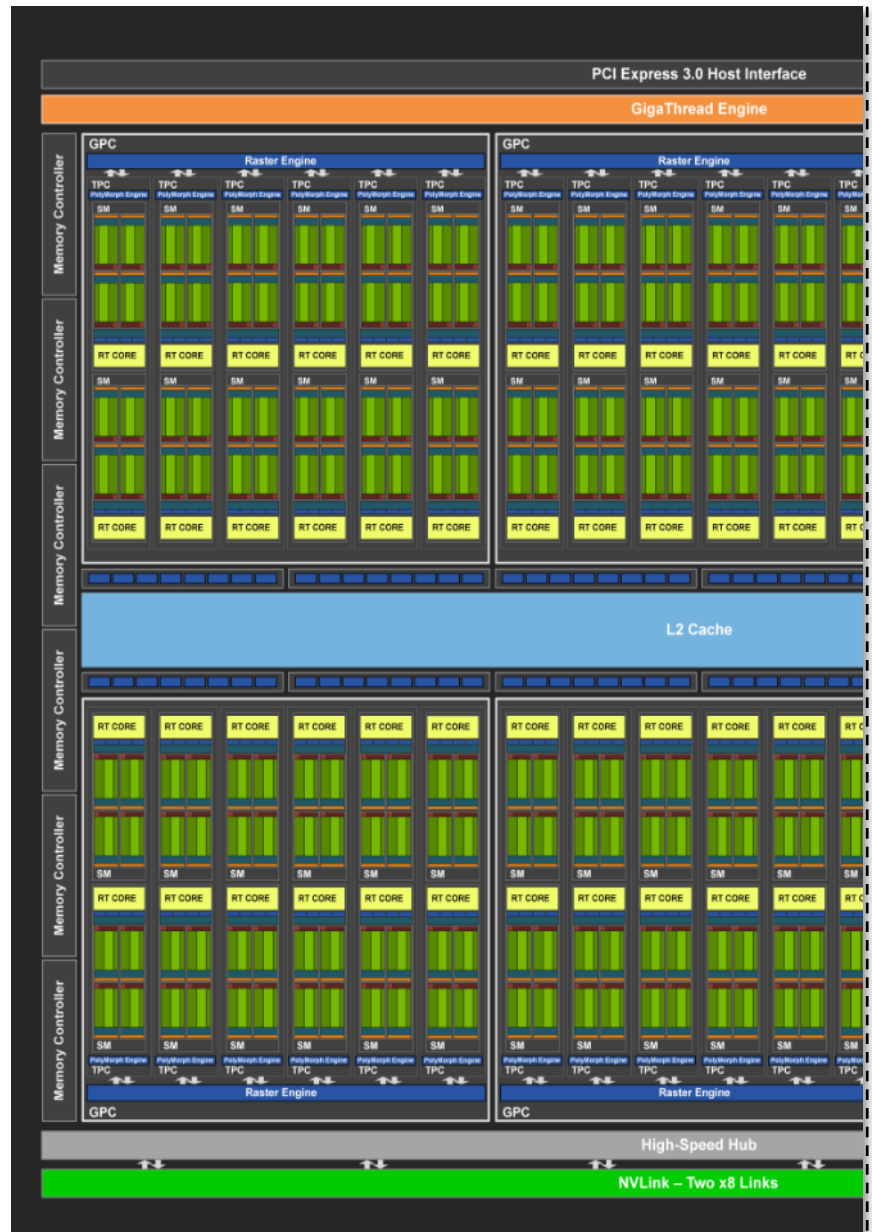


# Primer arhitekture – GPU čip

- + Raspoređivač blokova 1
- + Graphics proc. cluster\*: 6
- + streaming multiproc. 6\* (12)
- + Memorija:
- + L2 keš 6 MB
- + globalna\*\* 8192 MB

\* grupa streaming multiprocesora

\*\* globalna memorija nije deo samog čipa, zato je nema na slici



# Primer arhitekture – streaming multiprocesor

+ Raspoređivač warp-ova  $4^* (1)$

+ Jezgra:

+ float32, int32  $4^* (16, 16)$

+ tensor\*  $4^* (2)$

+ ray-tracing (1)

+ Jedinice:

+ load/store  $4^* (4)$

+ specijalne\*\*  $4^* (1)$

+ texture 4

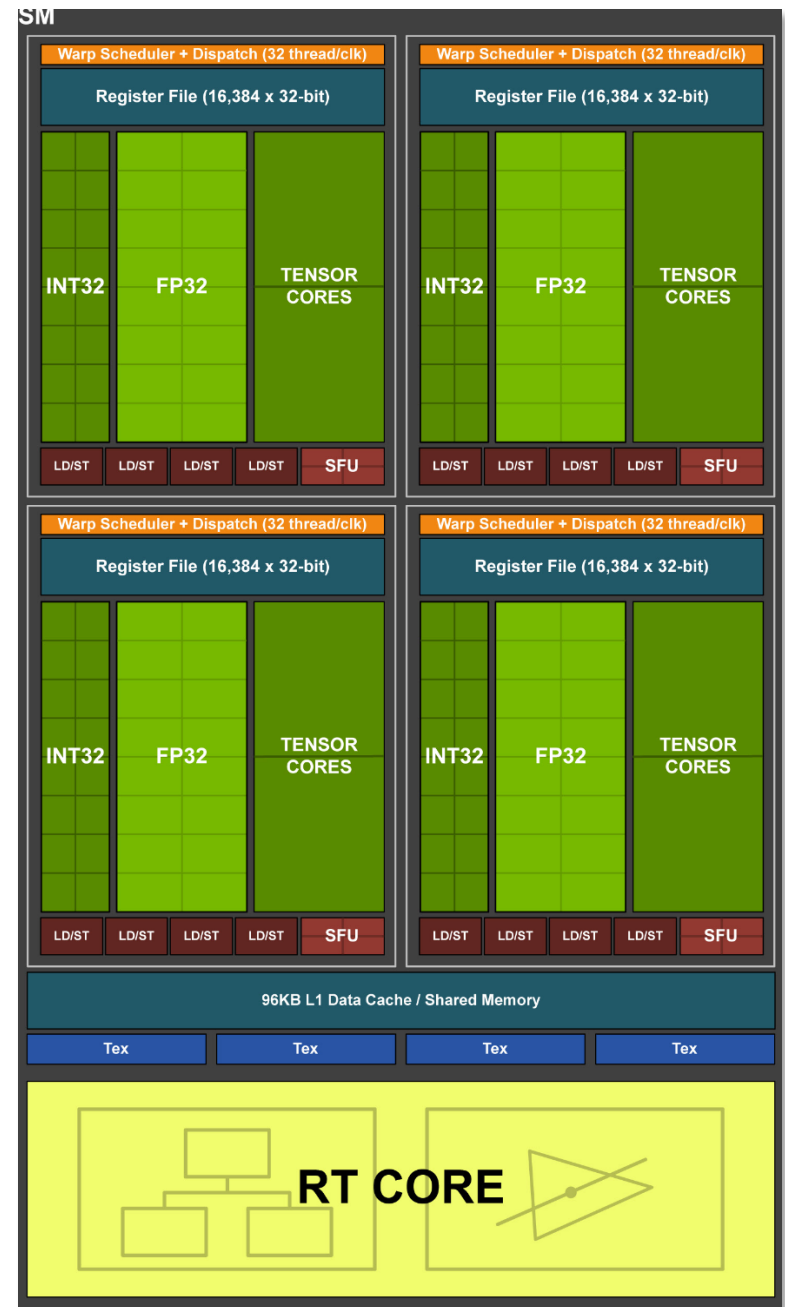
+ Memorija:

+ registri  $4^* (\sim 16K \text{ 32-bit})$

+ deljena + L1 keš 96 KB

\* tensor jedinice služe za mašinsko učenje

\*\* specijalne jedinice računaju "fensi" funkcije: sin, cos, tanh, ...



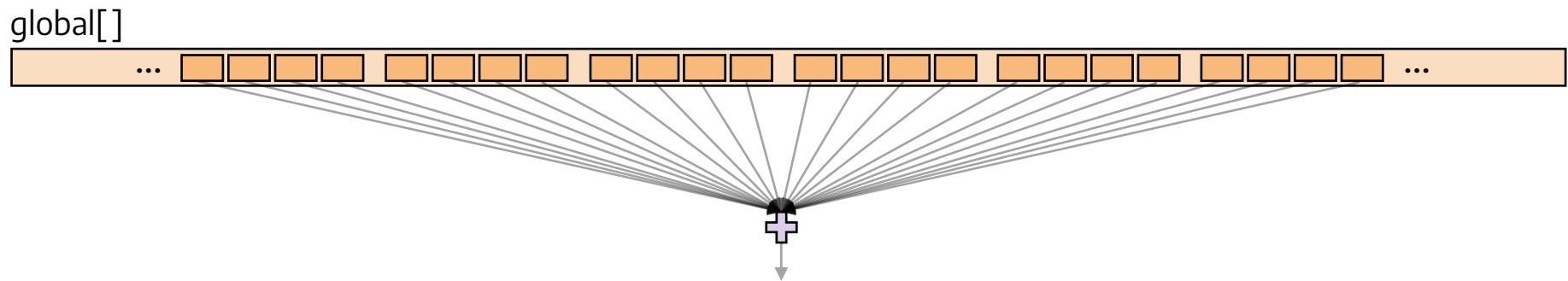


## Korisne ideje

# [Redukcija] vektora



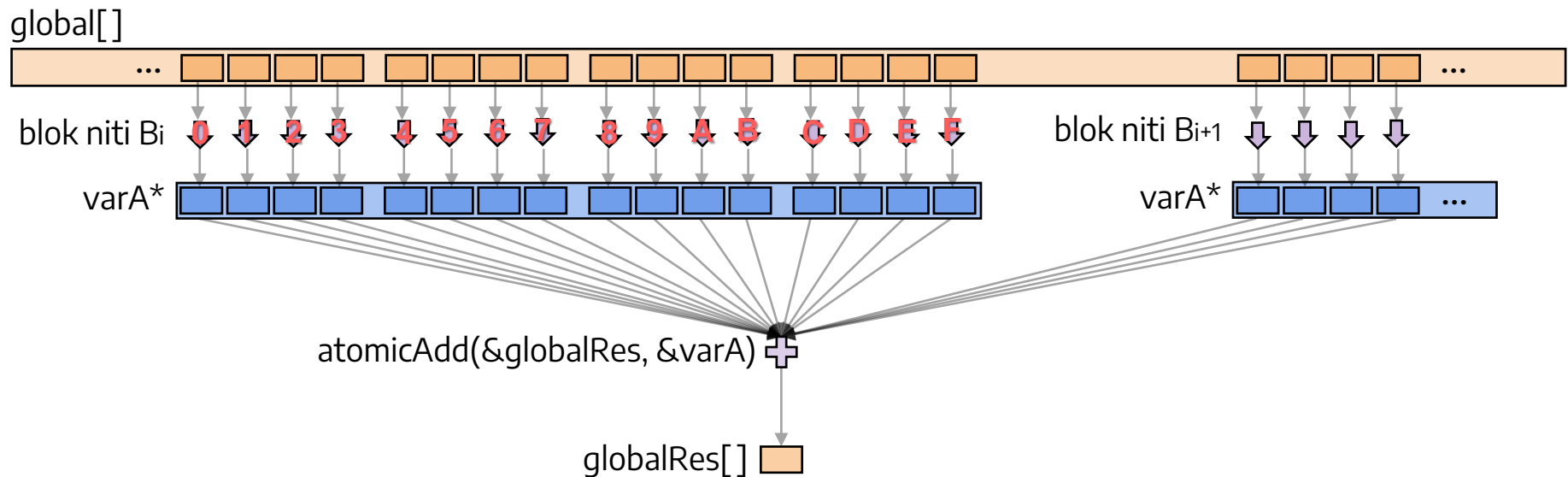
- + Čest problem – potrebno je primeniti asocijativnu operaciju na ceo niz i redukovati ga u broj
- + asocijativnost:  $(a + b) + c \equiv a + (b + c)$ , za neku operaciju +
- (+ pretpostavka – CPU je od ranije preneo niz u globalnu memoriju)



# [Redukcija] [1] "Atomično" rešenje



- + Ideja – svaka nit doda svoj broj u globalni brojač
- + koriste se atomične operacije
- + Sabiranje na CPU-u u jednom thread-u se radi redom, ali tu nema paralelizma!
- + ovo je imitacija sabiranja na CPU-u, i jako je spora!
- + na GPU ima više smisla sabirati kao binarno stablo (ne kao u ovom rešenju)



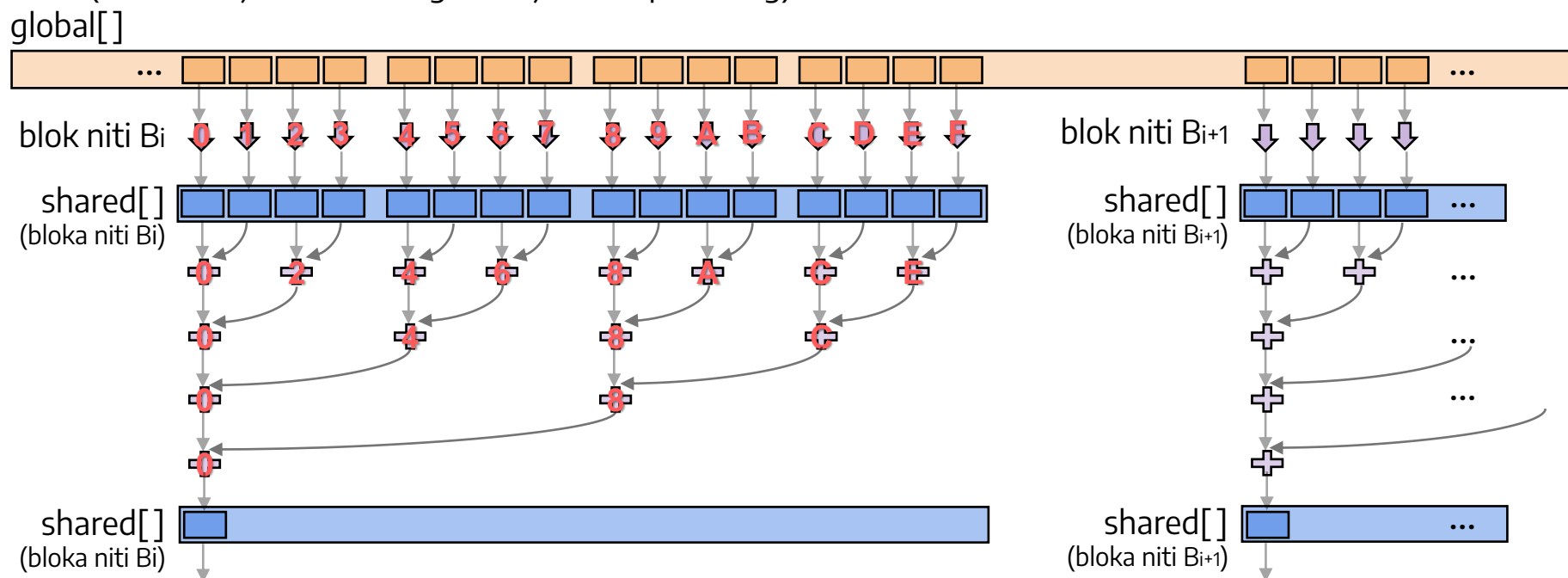
\* svaka nit u bloku ima svoj sopstveni registar koji sadrži promenljivu varA

# [Redukcija] [2] Preostale parne niti u bloku čekaju



- + Nit u bloku učitava svoj broj iz globalne memorije u svoj broj u deljenoj memoriji
- + sabere svoja dva susedna broja u prvi broj i sačeka da sve ostale niti to urade
- + od **preostalih niti** u bloku, parne niti ne rade ništa nadalje
- + ponoviti dok ne ostane jedan broj po bloku u deljenoj memoriji


- + Konačne rezultate prebacuje jedna nit po bloku u globalnu memoriju  
(CPU kasnije sabira mnogo manji niz od početnog)



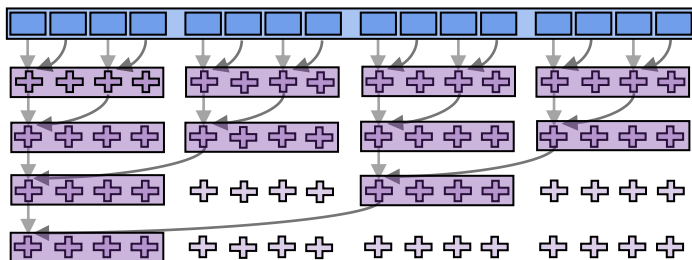
# [Redukcija] Previd - warp divergence



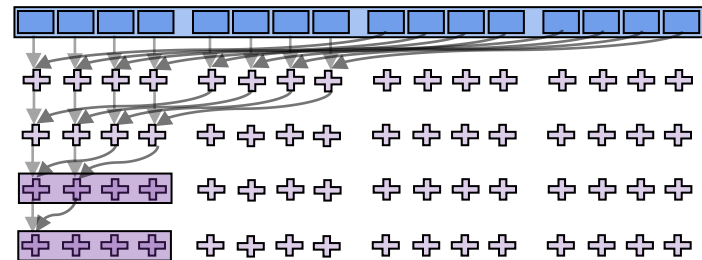
- + Niti u warp-u moraju da izvrše istu instrukciju *istovremeno*
  - + to je suština **SIMT modela** (single instruction – multiple *threads*)
- + kako se onda rade if, else, while, for?
- + Warp niti **izvrši obe grane**, ali nit A čuva samo rezultat svoje grane IF, a nit B čuva samo rezultat svoje grane ELSE
  - + drugim rečima, **warp divergence** treba izbeći ako je moguće
  - + na novijim arhitekturama nije više toliko hitno dovesti warp da konvergira

 problematični warp  
(sa warp divergencijom)

[1] *svaka druga* nit od preostalih niti čeka



[2] *gornja polovina* od preostalih niti čeka



# [Redukcija] Previd ii - memorijske banke



- + Deljena (i globalna) memorija su podeljene na 32 banke
  - + uzastopne 4B lokacije pripadaju susednim bankama
  - + zašto? – da bi bio paralelniji pristup ( $32 \times 4\text{B}/\text{cycle}$ )
- + 32 niti (warp) može istovremeno da pristupa 32 banke!
  - + ali, ako dve ili više niti pristupe istoj banci, ti pristupi se serijalizuju (loše, pristupi istoj banci ne mogu istovremeno)



# [Redukcija] Previd iii - memory coalescing



- + Memory [access] coalescing – spajanje osam susednih 4B-zahteva u jedan 32B!
  - + to kompajler uradi, ali samo ako prvi od tih zahteva cilja lokaciju deljivu sa  $8 \times 4B = 32B$  (32B-aligned lokaciju)
- + U našem slučaju, 32 zahteva bi se objedinilo u samo četiri!
  - + ako bi redom pristupali memorijskim bankama

# [Redukcija] Sve prethodne mane

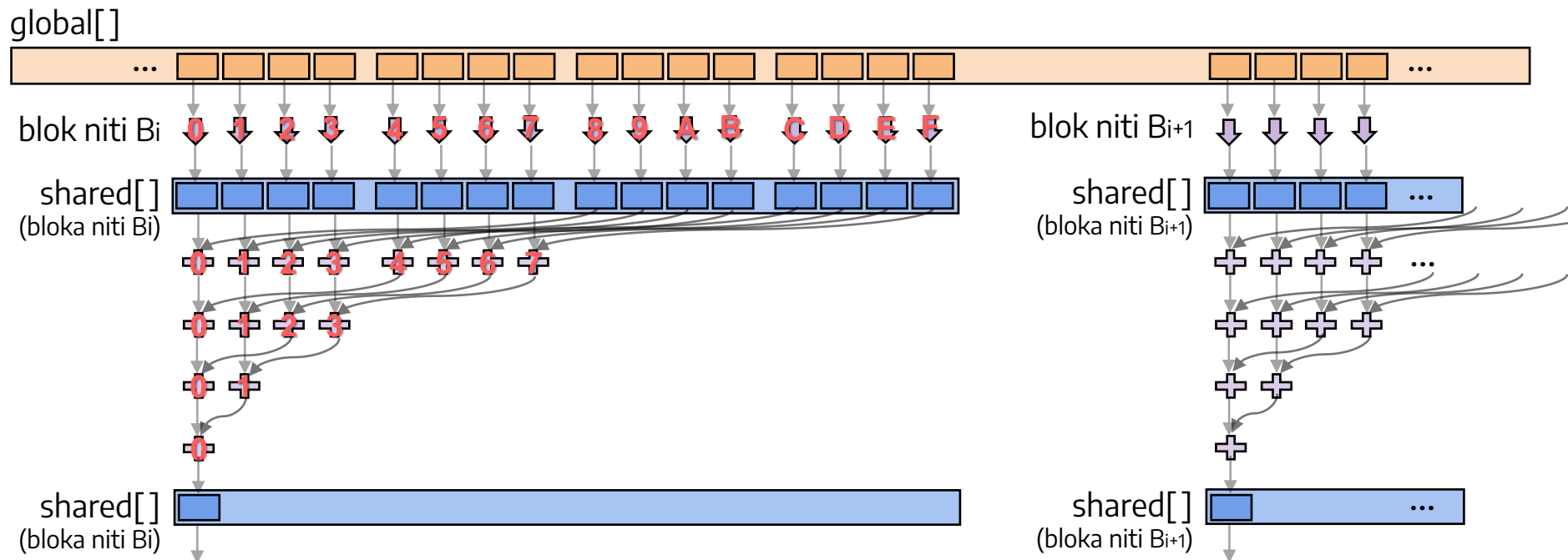


- + Kod nas su svi warp-ovi imali warp divergence
  - + bilo je sve manje aktivnih niti u warp-u  $\Rightarrow$  sve lošije iskorišćenje core-ova
- + Kod nas niti u warp-u nisu pristupale redom susednim lokacijama
  - + ne bi bilo kolizija u memorijskim bankama
  - + tada bi se dogodio memory coalescing  $\Rightarrow$  bio bi dosta povećan propusni opseg
- + Naš paralelni kod nije dobro optimizovan! 🤔

# [Redukcija] [3] Gornja polovina bloka niti čeka



- + Nit u bloku na  $A[i]$  doda  $A[i + \text{\#PolaAktivnihNiti}]$
- + gornja polovina aktivnih niti posle toga ne radi ništa
- + Ovo rešenje je moguće samo ako je operacija komutativna (često jeste)
- +  $a + b \equiv b + a$ , za neku operaciju +



# [Redukcija] Nedostatak GPU redukcije



- + Realan broj može da se predstavi sa određenom tačnošću
- + Kada se radi operacija nad float brojevima, često se gubi tačnost
  - + redosled sabiranja je bitan (float operacije nisu asocijativne)
  - +  $10^{100} + 1 - 10^{100} + 2 = 2$  !?
- + GPU rezultat redukcije će se verovatno razlikovati od CPU rezultata
  - + ali ni jedan rezultat verovatno neće biti tačan!
- + Opcije za CPU i GPU:
  - + ako tačnost nije bitna, sabirati nekim redosledom
  - + za često bolju tačnost, prvo sortirati brojeve rastuće a zatim ih redom sabirati
  - + za apsolutna tačnost, koristiti vektor float-ova za predstavu konačne sume

Preuzeti: + [cuda toolkit](#)

Primeri: + [cuda samples](#)\* + [predavanje](#)\*\*

Uputstva: + [cuda](#) + [programiranje](#) (pdf) + [best practices](#) + [profiler](#)

Arhitekture GPU-a: + [istorijski razvoj](#) + [GTX 1080](#) + [RTX 2080](#) + [lista gpu-ova](#)

Optimizacija: + [register spilling](#) + [kooperativne grupe](#) + [veličina bloka](#) (saveti)  
+ [stream-ovi](#) + [warp shuffle funkcije](#) + [warp primitive](#) + [merenje performansi](#)

Detaljnije: + [memorijske banke](#) + [memorijske funkcije](#) + [unified memorija](#)  
+ [adresni prostori](#) + [virtuelne funkcije](#) + [dogadjaji](#) + [cuda tipovi](#)  
+ [math api](#) + [int funkcije](#) + [float funkcije](#) + [accuracy and precision](#)  
+ [cuda grafovi](#) + [ptx instrukcije](#) + [ieee 754](#)

\* ovo je samo spisak, sami primeri dolaze uz cuda toolkit

\*\* redukcija i složeniji primeri: redukcija reda broja  $\pi$ , N-body problem i Needleman-Wunsch

Hvala na pažnji!