OpenCL, II deo: Model konkurentnog programiranja

- Paralelno i konkurentno izvršenje
- Ulančavanje i globalna sinhronizacija

Model paralelnog izvršenja

- Komanda za start jezgra enqueueNDRange:
  - Stvara N-dimenzion opseg (NDO) radnih stavki
  - NDO definiše 1, 2, ili 3-dim rešetku radnih-stavki (RS)
- Preslikavanja NDO na model fizičke arhitekture:
  - Svaka radna-stavka se dodeljuje jednom PE
  - PE može izvršavati više RS, po jednu u ciklusu
- Radna-stavka je nezavisna od drugih
  - Labav model izvršenja
  - Skalabilnost na veliki broj jezgara
  - Hijerarhija uređaja, tj. hijerahijska mem. struktura

Radna grupa

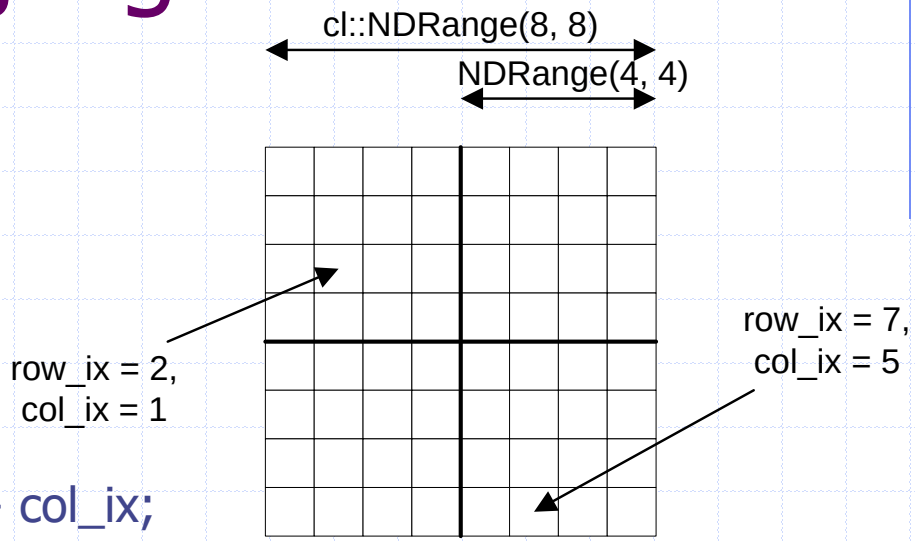
- Izvršni prostor se deli na 1, 2, ili 3-dim skupove RS iste veličine, tzv. radne-grupe (RG):
  - RG se može konkurentno izvršavati na jednoj RJ
  - U RG je dozvoljena komunikacija, ali je ograničena da bi se poboljšala skalabilnost
  - Unutar RG postoji mogućnost sinhronizacije
- Više RS se može izvršavati u jednoj niti:
  - Npr. na GPU se 64 RS izvršavaju u zaključanom koraku, kao jedna nit na SIMD jedinici
  - Rezultat je SIMD izvršenje po trakama (eng. lane)
  - Veličina grupe = umnožak širine SIMD u uređaju

Pozicija radne-stavke (RS) u NDO

- Intrinističke funkcije:
  - get_work_dim: broj dimenzija u NDO
  - get_global_size(dim): br. RS u zadatoj dimenziji
  - get_global_id(dim): indeks tekuće RS u dimenziji
  - get_local_size(dim): veličina RG u zadatoj dimenziji
  - get_local_id(dim): indeks tekuće RS
  - get_num_groups(dim): Br. RG u zadatoj dim
  - get_group_id(dim): Indeks tekuće RG

Primer izvršenja jezgra

```
__kernel void simpleKernel(
__global float  *a,
__global float  *b )
{
    int col_ix = get_global_id(0);
    int row_ix = get_global_id(1);
    int row_size = get_global_size(0);
    int address = row_ix * row_size + col_ix;
    b[address] = a[address] * 2;
}
...
cl::Event event;
err = queue.enqueueNDRangeKernel(kernel,
cl::NullRange, cl::NDRange(8, 8),
cl::NDRange(4, 4), NULL, &event);
...
```



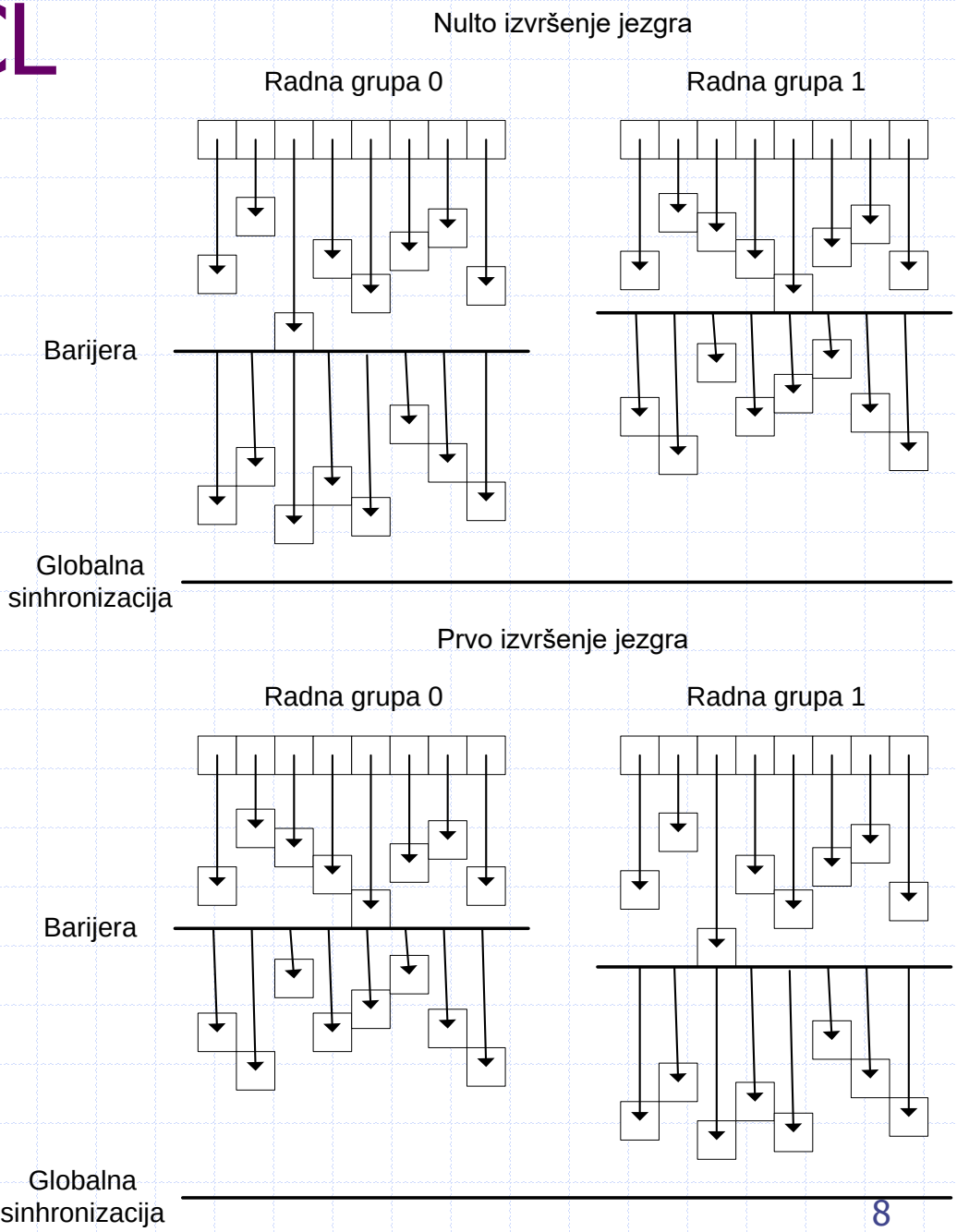
Model konkurentnog programiranja

- RS se izvršavaju nezavisno jedna od druge
  - Nema garancija u pogledu redosleda operacije upisa u jednoj RS i operacije čitanja u drugoj RS
  - Relaksiran model sinhronizacije i relaksiran model konzistentnosti memorije
  - OpenCL eksplicitno definiše sinhornizacione tačke
- U OpenCL omogućena potpuna konkurencija
  - Korišćenje semafora na GPU je problematično
  - Nit na GPU = FRONT-TALASA (eng. wavefront)
  - Zauzeti resursi su fiksni – opasnost od međusobnog blokiranja (eng. deadlock)

Operacija barijera

- Globalne sinhroniz. tačke na granicama jezgara
  - Obezbeđuju redosled između RS koje pripadaju različitim RG
- Deljenje podataka (u lok. mem.) između RS u istoj RG
  - Operacija BARIJERA, koja važi unutar jedne RG
  - RS ne može proći barijeru sve dok sve ostale RS u toj istoj RG ne dođu do barijere
  - Ponašanje barijere kroz koju ne prolaze sve RS u RG nije definisano

Primer OpenCL sinhronizacije



Ulančavanje i globalna sinhronizacija

- OpenCL je zasnovan na:
  - Paralelizmu zadataka, kojim upravlja domaćin +
  - Svaki zadatak koristi paralelizam obrade podataka
- To se postiže korišćenjem redova komandi
- Postoje sledeće vrste komandi:
  - Komande za izvršenje jezgra
  - Memorijske komande
  - Sinhronizacione komande

Sinhronizacija

- Komande za izvršenje jezgra i sinhronizacione komande se ulančavaju asinhrono
  - Završetak komande garantovan u sinhro. tačkama
- Primarne sinhronizacione tačke su:
  - Komanda clFinish, koja blokira program domaćina sve dok se sve komande u redu komandi ne završe
  - Čekanje na završetak zadatog događaja
  - Izvršenje blokirajuće memorijske komande
  - Videti primere programa u knjizi

Konzistentnost memorije (1/2)

- Mem. objekt koji dele ulančane komande je sigurno konzistentan:
  - U sinhronizacionim tačkama
  - Između dve komande u redu sa očuvanjem redosleda (eng. in-order queue)
  - Kod komunikacionih događaja gde jedna komanda generiše događaj koji druga komada čeka
- Za korektnost na nivou API sprege domaćina:
  - mora se koristiti neka od prethodno opisanih blokirajućih operacija

Konzistentnost memorije (2/2)

- Konzistentnosti između različitih uređaja?
  - Mem. objekti se pridružuju kontekstima, a ne uređajima
  - Biblioteka je dužna da obezbedi da su takvi objekti konzistenti preko više uređaja
  - U tom cilju, podaci se po potrebi prebacuju sa uređaja na uređaj
  - Ovaj prenos podataka je transparentan za korisničku aplikaciju

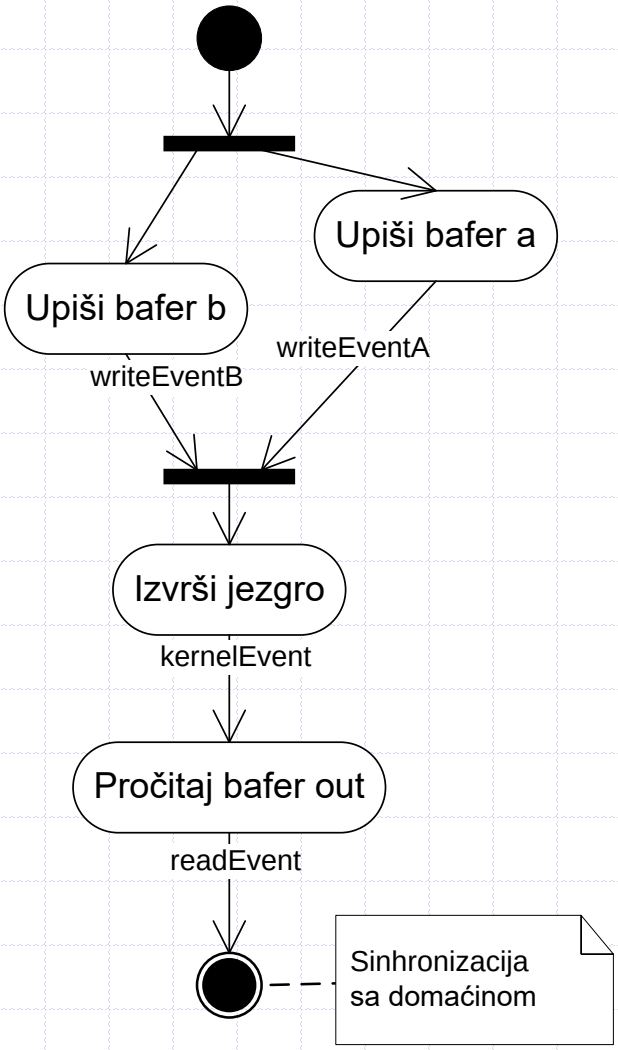
Događaji

- Redovi komandi:
  - Sa očuvanjem redosleda
  - Bez očuvanja redosleda (eng. out-of-order queue)
- U slučaju reda bez očuvanja redosleda:
  - Biblioteka može da rasporedi operacije u paraleli
- Kada biblioteka barata sa više redova komandi:
  - Ne važi nikakva predpostavka o redosledu izvršenja elemenata tih redova

Graf zadataka

- Graf zadataka se konstruiše od događaja
- Graf povezuje zadatke ulančane u bilo kom redu komandi, koji je prudružen zadatom kontekstu
- F-iji za ulančavanje komandi može se proslediti:
  - Pojedinačan događaj
  - Lista događaja
    - Tada izvršenje komande ne započinje sve dok se ne završe svi ulazni događaji

Primer grafa zadataka

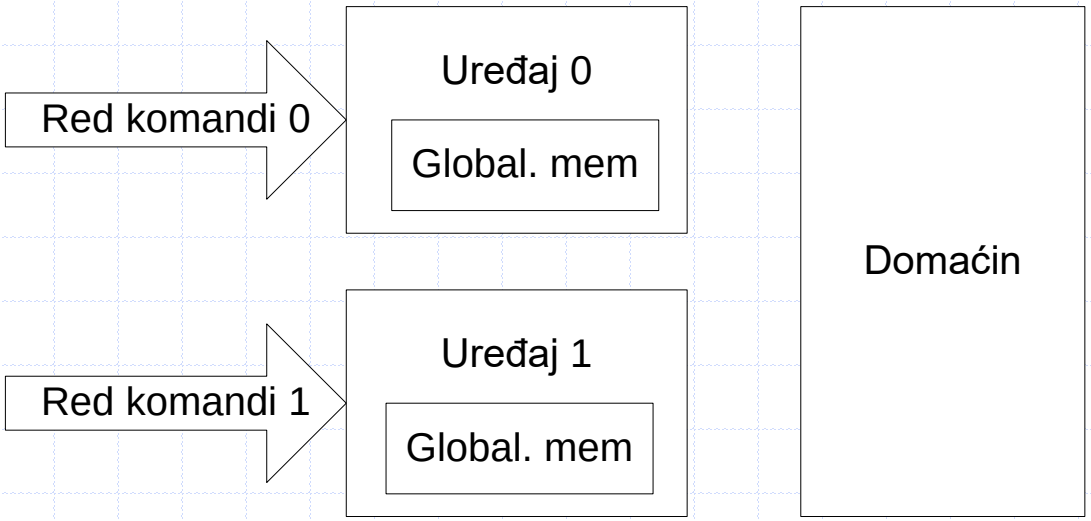


- Odgovarajući programski kod je dat u knjizi

Redovi komandi prema više uređaja

- Namena više redova komandi za isti uređaj:
  - Preklapanje izvršenja različitih komandi
  - Preklapanje izvršenja komandi i komunikacije domaćin-uređaj
- S druge strane, svaki uređaj mora da ima svoj sopstveni red komandi
- Sledi primer konteksta sa dva uređaja, i po jednim redom komandi za svaki uređaj

Primer konteksta sa dva uređaja

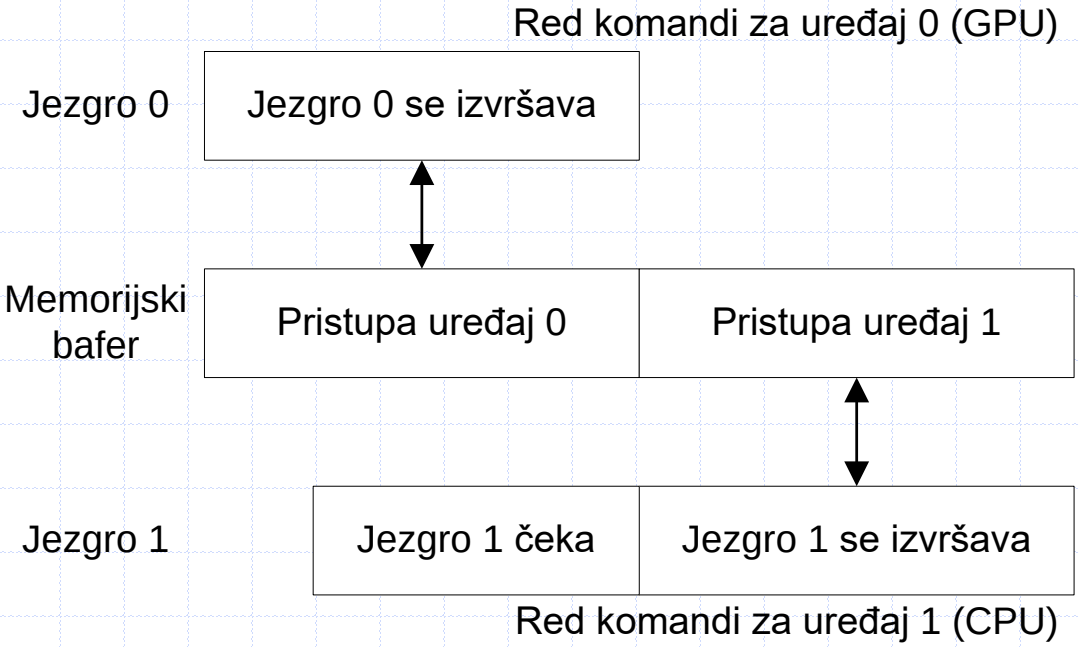


- Odgovarajući programski kod je dat u knjizi
- Napomena: sinhronizacija pomoću događaja se odnosi isključivo na komande u istom kontekstu
- Da su za ta dva uređaja bila napravljena dva konteksta, koristili bi clFinish i eksplicitno kopiranje bafera

Modeli programiranja više uređaja

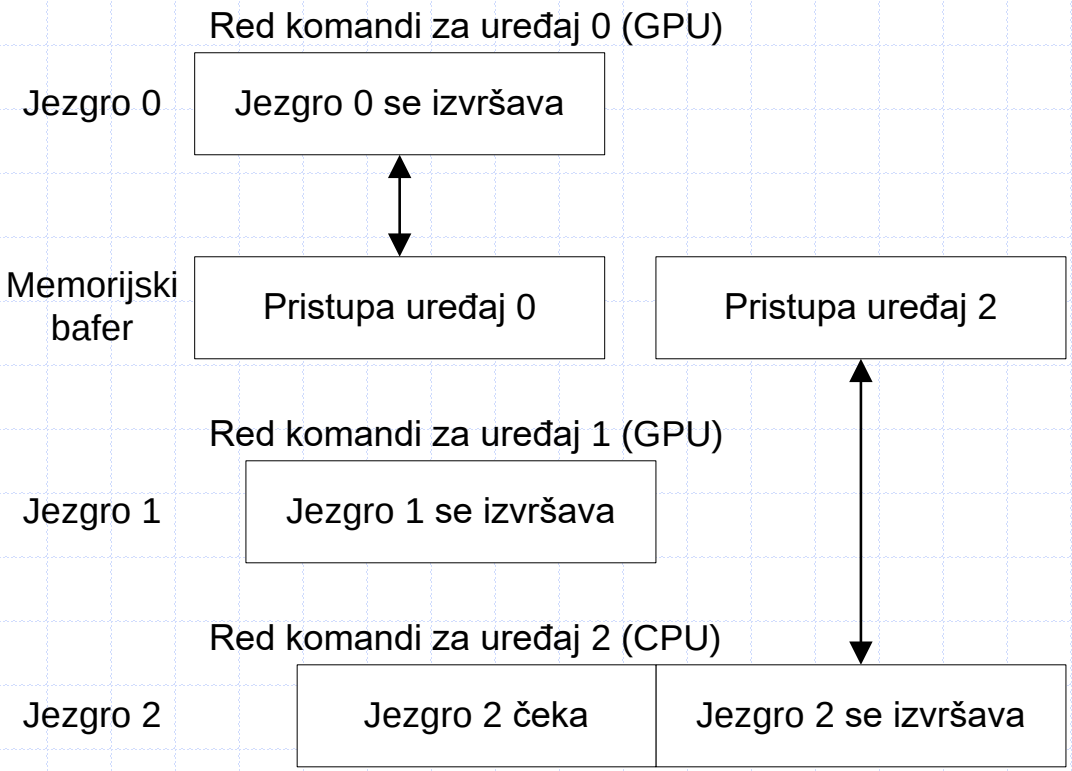
- Postoje dva modela:
  - Režim protočne obrade, gde jedan uređaj čeka rezultate drugog
  - Režim nezavisne obrade, gde više uređaja radi nezavisno jedan od drugog
- Slede primeri ova dva modela

Primer režima protočne obrade



- Lista čekanja uređuje izvršenje tako da se jezgro u redu za uređaj 0 završava pre nego započne izvršenje jezgra iz reda za uređaj 1
- Odgovarajući programski kod je dat u knjizi

Primer režima nezavisne obrade



- Dva GPU uređaja obrađuju svoja jezgra potpuno nezavisno
- Neophodno je da postoje odvojeni baferi za ta dva uređaja, da bi se omogućila paralelna obrada
- Odgovarajući programski kod je dat u knjizi

Šira upotreba događaja (1/2)

- Stanje komande se može proveriti putem f-ije getInfo nad njoj pridruženom događaju
- Moguće je pribaviti sledeću informaciju:
  - Red komandi događaja
  - Kontekst događaja
  - Tip komande pridružene događaju
  - Stanje komande:
    - ulančana, podneta, u izvršenju i završena

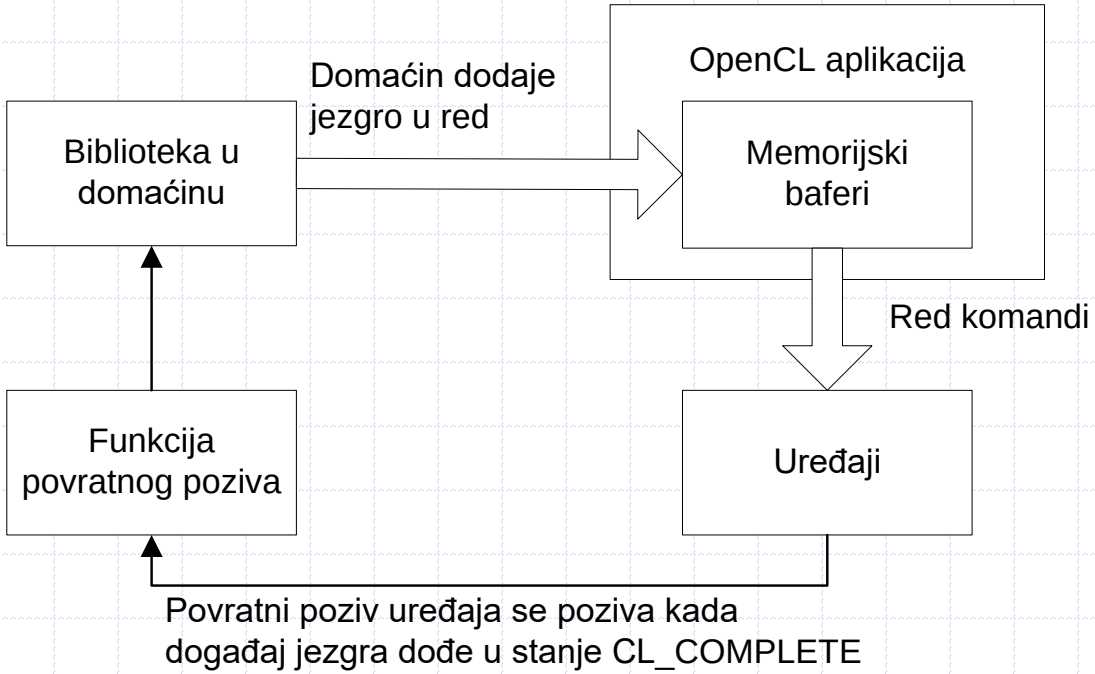
Šira upotreba događaja (2/2)

- Ako je profilisanje u kontekstu dozvoljeno
  - F-ija getProfilingInfo nad događajem vraća vremensku informaciju o izvršenju komande
- Korisnički definisan događaj
  - F-ija clCreateUserEvent pravi korisnički događaj
  - Njega je onda moguće definisati kao ulazni događaj za neku sledeću komandu
  - F-ija clSetUserEventStatus omogućava eksplicitno postavljanje stanja, npr. na CL_COMPLETE.
  - Vidi primer programskog koda u knjizi

Povratni pozivi (eng. callbacks)

- Funkcija povratnog poziva se poziva za zadato stanje izvršenja komande u redu komandi
- Može se koristiti za:
  - Ulančavanje novih komadni
  - Pozive funkcija domaćina, npr. unutar biblioteka
- Definiše se pomoću funkcije clSetEventCallback
  - Vidi primer u knjizi

Primer povratnog poziva



- Aplikacija u kojoj CPU domaćina tesno sarađuje sa uređajem kao što je GPU
- Domaćin može da dalje radi neki koristan posao, umesto da se vrti u prazno dok čeka GPU
- Odgovarajući prog. kod je dat u knjizi

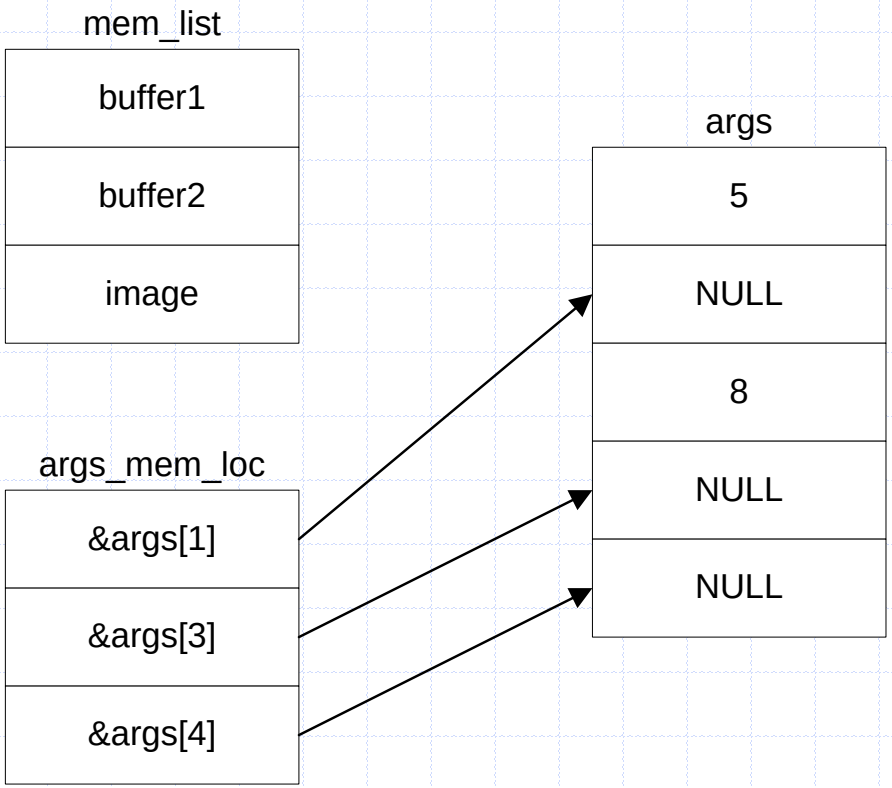
Detalji povratnih poziva

- Redosled:
  - Prvo API funkcija clEnqueueNDRangeKernel
  - Zatim API funkcija clSetEventCallback
- Napomene:
  - Nije garantovan redosled poziva f-ije povratnog poziva (registrovane za stanja višestrukog izvršenja)
  - Poziva se asinhrono i može je pozivati više niti
  - Nedefinisano ponašanje kod poziva sis. rutina i blokirajućih API funkcija (kao što je clFinish)

Urođena jezgra

- Urođena jezgra (eng. native kernels):
  - Alternativa povratnim pozivima
  - Čistije integrisana u OpenCL model izvršenja
  - Mogu se izvršavati u grafu zadataka, mogu biti pokrenuta događajima, i mogu pokretati događaje
- Za razliku od običnih jezgara, urođena se pokreću f-ijom clEnqueueNativeKernel
  - Prosleđuje se pokazivač na standardnu C funkciju
  - Lista stvarnih parametara C funkcije i njena veličina se prosleđuju zasebno

Raspakivanje (eng. unboxing) parametara urođenog jezgra



- Vrednosti se prenose kroz standardnu listu parametara args
- Memorijski objekti (baferi i slike) se prenose kroz niz objekata i niz adresa njihovih mesta u listi parametara args
- Primer: prenos dve vrednosti (5 i 8), dva bafera i jedne slike
- Vidi programski kod u knjizi

Barijere, markeri i čekanje dog.

- Barijera garantuje redosled pre i posle nje
  - Slična asinhronoj komandi clFinish
- Markeri:
  - Ulančavaju se u red funkcijom enqueueMarker
  - Slični su barijerama, ali ne blokiraju izvršenje
  - Završavaju se kada se svi preth. elem. u redu završe
  - Izlazni događaj je eksplicitan – pobuda sl. događaja
- Sinhronizaciona primitiva waitForEvents:
  - Suprotna od markera
  - Blokira se dok se zadati skup ul. događaja ne završi