1) Heterogeno računarstvo (GPU, CPU)

index = redniBroj_niti_u_bloku + redniBroj_bloka * dimenzija_bloka

GPU nema virtuelnu memoriju, prekide, i saradnju sa uredjajima poput tastature i misa. Veoma je neefikasna u slucaju da posao koji joj je dodeljen nije tipa SIMD
Heterogeno racunarstvo : Multicore CPU, Manycore GPU i rekonigurabilne jedinice (FPGA)

GPU - CPU

- CPU -- Optimizovan za malo kasnjenje zbog cache memorija, poseduje kontrolnu logiku za izvrsenje preko reda i predikcije, desetine niti, dobar za sve vrste obrada
    - Cilj CPU - minimizovati kasnjenj (cache i kompleksna logika)
- GPU -- Optimizovan za paralelnu obradu i propusnost, tolerise mem. kasnjenja, Hiljade niti(vise tranzistora), dobar za paralelnu obradu
    - Cilj GPU - maksimizirati propusnost(multithreading moze da skrije kasnjenje, kontorlna logika je ista za vise niti

GPU se razlikuje od CPU-a po samom dizajnu, njihova kompleksnost je manja. Optimizovana je za compute-intensive i highly paralle proracune.

2) Konkurentno vs Paralelno

Konkurentno(kvazi paralelno):
- 1 CPU - 1 Core

Cilj:
- -bolje iskoristiti procesor
- -poboljsati odzivnost sistema
- -podrzati sto vise korisnika

Problem:
- Prioriteti
- -Rasporedjivanje

Paralelno(vise niti se izvrsava na vise jezgara):
- vise CUP - vise Core-ova

Cilj:
- Paralelno izvrsavanje
- -Speedup

Problemi:
- -Paralelni algoritmi i strukture podataka
- -Mapiranje i balansiranje opterecenja

Zajednicki problemi: sinhronizacija, komunikacija, race-condition, deadlock, konzistentnost podataka

3) Nivoi paralelizma

- Na nivou instrukcije
    - -- izvrsenje preko reda, predikcije grananja
- Na nivou podataka
    - -- vektorski procesori, SIMD izvrsavanje - GPU
- Na nivou niti
    - povecacanje broja jezgara, niti, procesa

4) GPU - memorijska hierarhija (tipovi, vidljivost)

- Registri - Najbrzi prisup - dostupni tokom zivota niti
- Lokalna - sporiji od registara i deljene memorije, dostupna tokom zivota niti - nalzai se u globalnoj memoriji
- Deljena - Moze biti brza kao i registarska ako nema konflikta. Dostupna svim nitima iz blok(tokom zivota bloka)
- Globalna - sporija od registara i deljenen memorije, dostupna I device-u I host-u (zivotni vek app)
- -L1 - kesira pristup globalnoj memoriji - brzina ista kao i kod deljene memorije
- -L2 - uprzava atomske operacije - sporiji od L1, ali i dalje brzi pristup nego kod globalne memorije.

5) GPU - hierarhija niti

- Nit se izvrsava na jednom jezgru (1 Warp = 32 niti) - jedinstven ID
- Blok se izvrsava ne jednom SM-u (blokovi ne migriraju) - jedinstven ID
- Grid se izvrsava na jednom Device-u - ~~do 16 karnela(gridova) po uredjaju - sekvencjalno se izvrsavaju.~~

Blok se dodeljuje SM. U SM-u blok se deli na Warp-ove (32/64niti). Svih 32niti u Warpu moraju izvrsavati isti set instrukcija. Warp-ovi se izvrsavaju konkurentno na SM-u. - Nit se izvrsava na jednom jezgru, grpupa od 32 niti se zove warp vise warp-ova cine blok koji se izvrsava na SM-u. Blokovi se ne prkljucuju izmedju SM-a. Vise blokova se moze izvrsavati na istom SM-u - konkurentno. Grupa blokova se zove grid I I izvrsava se na device-u(sekvencijalni kerneli).

6) Paralelni programi - odnos GPU-CPU

- Imamo master proces koji je izvrsava na CPU I izvrsava sledece korake:
  - Inicializacija device-a
  - Alokacije memorije na hostu i device-u
  - Kopiranje podataka sa hosta na device
  - Pokretanje vise instaci izvrsavanja na device-u
  - Kopiranje podataka sa device-a na host
  - 3-5 ponavljanja prethodnog postupka
  - Dealokacija memorije i kraj programa!
- Na GPU:
  - Svaki blok se izvrsava na StreamingMultiprocessor-u(SM)
  - Ako je broj blokova veci od broja SM-a, onda ce se samo jedan blok izvrsavati u jednom trenutku(queue)
  - Sve niti jednog bloka mogu da pristupe deljenoj memoriji, ali ne mogu videti sta rade druge niti
  - Ne mozemo znati redosled izvesavanja niti i blokova.

7) CUDA - osnovni koncepti

-Proizvedena od strane Nvidia-e
- Programski model: Prenos podataka na GPU, Izvrsavanje , Pruzimanje razultata
- Racunarska arhitektura opste namene.
- Izvorni kod host-a i device-a se nalazi u istom fajlu(u OpelCL-u nije tako)
- CUDA je racunarska arhitektura opste namene - ukljucuje CUDA set instrukcija i paralelni engine na GPU.
- Koristi C/C++ uz minimalne izmene - minimalni nivo apstrakcije
- Relativno low-level pristup: Apstrakcija CUDA-e je usko vezana za mogucnosti/performanse GPU-a
- CUDA na Nvidia karticama, OpenCL na bilo kojim GPU-ovima I CPU-ovima.
- Apstrakcija : Hierarhija niti, “Jednostavne” metode sinhronizacije, Deljena memorija za saradju izmedju niti
- CUDA virtualizje HW tako sto niti virtualizuje skalarni procesor a blok virtualizuje multiprocesor. Sve se rasporedjuje na fizicki HW GPU-a. Niti I Blokovi se pokrecu I izvrsavaju do kraja, blokovi su nezavisni.

8) GPU - virtuelizacija

GPU virtualizje HW tako sto niti virtualizuje skalarni procesor a blok virtualizuje multiprocesor. Sve se rasporedjuje na fizicki HW GPU-a. Niti I Blokovi se pokrecu I izvrsavaju do kraja, blokovi su nezavisni.

9) Tipovi promenljivih u CUDA-i i njihova vidljivost

Variable declaration	Memory
`int var;`	register
`int array_var[10];`	local
`__shared__ int shared_var;`	shared
`__device__ int global_var;`	global
`__constant__ int constant_var;`	constant

10) Kreiranje kernela

- definisanje __global__ kernel funkcija(programa) koje ce se izvrsavati na GPU.
- alokacija memorije na GPU i oslobadjanje memorije na GPU nakon zavrsenog posla.
- kreiranje(startovanje) kernela sa definisanim brojem niti, blokova, i argumentima
  myKernel<<<nBlocks, nThreads>>>(par1,par2)
- sinhronizacija CPU i GPU (cekamo da se izvrsi kod na GPU

11) GPU - atomic i sinhronizacija

Na GPU ne mozemo koristiti klasicne load i store operacije zbog Race-Condition-a, umesto toga koriste se atomske instrukcije. Atomske opracije nam garantuju da samo jedna nit ima pristup memoriji sve dok se opracija ne izvrsi od kraja(neprekidno). Definicija atomskih instrukcija se postize kljucnom reci atomic. CUDA obezbedjuje atomske operacije nad globalnom memorijom.

Sinhronizacija:
- Sve niti u warp-u se moraju eksplicitno sinhronizovati.
-___syncthreads() se koristi unutar kernela kako bi se zaustavile sve niti i sacekale da sve budu na istom mestu. Veoma je bitno kada se pise kod, da se obezbedi da se sve niti mogu izvrsiti do sinhronizacije u protivnom nastaje deadlock. Izmedju blokova nema sinhronizacije.
Ako hocemo da sinhtonizujemo rad hosta i device-a tako da se nalaze na istoj tacki izvrsenja koristimo: cudaDeviceSynhronize();
- osigurava da se svi asinhroni zadaci zarvse pre nastavka rada!
cudaStramSynhronize(); - za sinhronizaciju stream-ova kernela(vise kernela).

Blokovi moraju biti nezavsni. Sva moguca preplitanja blokova bi trebala da budu validna:
- izvrsenje do kraja bez prekida, bilo koji redosled izvrsavanja, mogu se izvrsavati konkurentno ili sekvencijalno.

Blokovi nisu sinhronizovani ali mogu biti koordinisani: deljeni pokazivac u redu cekanja.

12) Strategija obrade podataka na GPU

- Podela podataka na delove koji se mogu smestiti u deljenu memoriju
- Svakim delom podatak rukuje jedna blok
- Iz globalne memorije se ucitavaju podaci u deljene memorije blokova
- Izvode se proracuni na blokovima, sa datim delovima podataka
- Rezultati se kopiraju nazad - iz deljene u globalnu memoriju.

13) Tipovi adresnih prostora na GPU

Ugradjeni primitiv **memcpy(cudaMemCpy)** nam omogucuje kopiranje podataka izmedju adresnih prostora na host-u i device-u.
Device ima 3 tipa adresnih prostora: **private** - za svaku nit, **shared -** za svaki blok, **global -** za svaki program.

14) Masivni paralelizam (fina i gruba podela posla)

Masivni paralelizam:
- Upotreba par stotina/hiljada blokova
Blok se izvrsava na jednom SM-u. Potreban nam je veliki broj blokova kako bi upotrebili 10-ak SM-a jer jedan SM izvrsava efikasno 2-8 blokova konkurentno. Ogroman broj blokova nam treba kako bi posao skalirali na vise GPU-ova.
Ovu logiku koristimo kada zelimo ugrubo da podelimo posao(task-ove)

-Upotreba stotine niti po bloku
Svaka nit se izvrsava na jednom jezgru. Treba nam do 512 niti po SM-u kako bi koristili sva jezgra istovremeno. Koristi se umnozak od 32 niti(warp) po bloku. Finija podela posla - paralelizam na nivou podatak, niti, instrukcija.

Zakljucak: Vise blokova - grublja podela posla, Vise niti po bloku - finija podela posla.

15) SIMT - koncept

SIMT - niti se izvrsavaju u grupi od 32 niti - warp. Sve niti u warp-u izvrsavaju iste instrukcije. HW multithreading - obuhvata alokaciju resursa na HW-u i rasporedjivanje niti. HW se oslanja na niti kako bi sakrio kasnjenja. Context-switch izmedju wapr-ova je bez overheda.
SIMT:
- Wapr - set od 32 paralelne niti koje izvrsavaju jednu instrukciju. SIMT - dodeljuje instrikciju warp-u

SM sva jezgra deli na dva warp-a , zato ima i dva rasporedjivaca. SIMT warp izvrsava svaku instrukciju na 32 niti. Predikati omogucuji ili onemoguciju individualno izvrsavanje niti. Stek upravnja grananjima za svaku nit. Redudantna izracunavanja su brza od neispravnog grananja(prilikom grananja izracunace se obe strane je je to brze nego da izaberemo pogresno grananje pa da se ponovo vracamo i racunamo).

16) Occupancy

Occupancy: odnos broja aktivnih warp-ova po SM-u u odnosu na maximalan broj warp-ova po SM-u.
- Treba da je sto veca kako bi se sakrilo kasnjenje
- Limitirana je upotrebom resrsa
- Zavisi od athitekture ali i od same aplikacije.

Occupancy optimization:
- Trenutnu zauzetost saznajemo preko profajlera.
- Prilagodjavanje upotrebe resursa kako bi se zauzetost povecala(promena velicine bloka, limitiranje upotrebe registara, dinamicka alokacija deljene moemorije)

17) Deljena memorija i optimizacija

- Za komunikaciju izmedju niti u bloku
- Koristi sekao kes kako bi se zaobisao pristup globalnoj memoriji (kod vise pristupa istom podatku)
- Izbegavanje pristupa koji nije po redosledu
- Kasnjenje je od 4-8 puta manje nego u slucaju L2 I DRAM memorija.
- Koristi sinhornizaciju kako bi se izvrsila komunikacija - sinhronizacija je jako brza!
- Niti u warp-u pristupaju deljenoj memoriji preko crossbar-mreze. Konflikti mogu da postoje ali oni neznatno uticu na performanse. Favorizuje se upotreba deljene memorije u odnosu na globalnu.

18) Restrikcije CUDA-e u odnosu na C/C++

- moze se pristupiti samo memoriji GPU-a, nema promenljivog broja argumenata, nema statickih promenljivih, nema rekurzije, nema polimorfizama.
- sve funkcije moraju imati kvalifikator:
  - __global__ - pokrenute od CPU-a, ne mogu biti pozvane sa GPU-a, moraju da vracaju void
  - __device__ - mogu da se pozovu samo sa GPU-a
  - __host__ - samo sa CPU-a
  - __host__ i __ device__ se mogu kombinovati

19) Preporučen način za pristup deljenoj memoriji od strane GPU-a

Niti bi trebale deljenoj memoriji da pristupaju redosledno, po indexu bez preplitanja
Preporuke:
- Koristiti sukcesivne mem. adrese za niti u warp-u
- Koristit pristup bez presecanja kada god je to moguce (kao kod niza - sekvencijalan pristup po indeksima)
- Load i Sotre operacije mogu narusiti konzistentnost - atomic nam serializuje pristup mem.

20) Memorijski prostori i pointeri

Najcesce se koriste pointeri, na osnovu vrednosti pointera ne mozemo znati da li je to na GPU-u ili na CPU-u. Dereferenciranje pointera koji pokazuje na memoriju na CPU na GPU-u moze dovesti do kraha celog procesa I obrnuto.

21) Definisati blokove i objasniti njihove osobine

Blok je sacinjen od niti. Niti se grupisu u warpove od po 32 niti, vise warpova cini blok. Blok se izvrsava na SM-u, blokovi ne migriraju izmedju SM-a. Rasporedjivanje blokova na SM-u: HW rasporedjuje blokove na dostupne SM-e, nema garancije redosleda, blok ce biti rasporedjen cim neki drugi blok zavrsi! Svaki blok se mapira na jedan ili vise warp-ova. SM moze efikasno da izvrsava 2-8 blokova konkurentno(nalaze se u queue-u). Blokovi moraju biti nezavsni. Blokovi nisu sinhronizovani, ali mogu biti koordinisani. Svaki blok ima deljenu memoriju kojoj mogu da pristupe sve niti iz tog bloka (moze biti brza kao I registarska, brza od globalne)

22) Preklapanje CPU i GPU computinga i CPU-GPU (GPU-CPU) transfera podataka

- cudaMemCpy() izaziva prenos podataka na relaciji GPU - CPU I obrnuto. Ovo se moze preklopiti sa periodom procesuiranja GPU ili CPU -> Resenje: u razlicitim kernel-ima kada se na jednom vrsi procesiranje na drugom se vrsi kopiranje podataka (u razlicitim taktovima) - da nije ovako procesuiranje ne bi bilo moguce sve dok se podaci ne kopiraju (Single vs Dual Stream, Overlap)

23) Kako se može optimizovati GPU algoritam množenja matrica

A x B = C

1. Transponovacemo matricu B, kako bi dobili redosledni pristup po indeksima (umesto a11,a21,a31, bice a11,a12,a13…)
2. Podelom matrica A i B na blokove, svakom bloku cemo dati npr 32-vrste A i 32-kolone B

24) Ako funkciju definisanu u C,C++ programskom jeziku želimo da koristimo kao kernel funkciju u CUDA programiranju šta teba da uradimo?

Da joj prilikom definicije stavimo prefix __global__, alociramo memoriju na GPU, iskopiramo potrebne podatka na GPU, pozovemo je preko def. broja blokova i niti myKernel<<<nBlocks,nThreads>>>(par1,par2,par3)

25) Opisati programsku strategiju rada globalne i deljene memorije grafičkog procesora

- Podela podataka na delove koji se mogu smestiti u globalnu memoriju
- Svakim delom podatak rukuje jedna blok
- Iz globalne memorije se ucitavaju podaci u deljene memorije blokova
- Izvode se proracuni na blokovima, sa datim delovima podataka
- Rezultati se kopiraju nazad - iz deljene u globalnu memoriju.

26) Objasniti u rutini cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction) poslednji argument: enum cudaMemcpyKind direction

Oznacava smer prenosa podataka:
-cudaMemcpyHostToDevice - sa hosta na device
-cudaMemcpyDeviceToHost - sa device na host
-cudaMemcpyDeviceToDevice - sa device-a na device

27) Definisati nit, blok i rešetku (thread, block, grid) i gde se izvršavaju

- Nit se izvrsava na jednom jezgru (1 Warp = 32 niti) - jedinstven ID
- Blok se izvrsava ne jednom SM-u (blokovi ne migriraju, sastoje se iz vise warp-ova) - jedinstven ID
- Grid se izvrsava na jednom Device-u - do 16 karnela(gridova,vise blokova) po uredjaju - sekvencjalno se izvrsavaju.

28) Objasniti cache data locality

Koristi lokalnost podataka
- vremenska - podatak kome smo sada pristupili verovatno ce se ponovo koristiti
- prostorna - podaci u okolini trenutnog podataka ce se verovatno koristiti zato ih treba prebaciti u cache.

29) Uporediti brzine rada registra, lokalne memorije, deljene memorije i globalne memorije

-Registrarska - najbrza memorija
-Lokalna - nalazi se u globalnoj 150x sporija nego registarska
-Deljena - moze biti iste brzine kao i registarska
-Globalna - 150x sporija od registarske i deljene

30) Kako se Parallel Thread Execution (PTX) pseudo-assembly jezik (kod) koristi u Nvidia CUDA programskom okruženju (kodu)

PTX - asemblerski jezik za CUDA-u. CUDA se prevodi u PTX a drajver GPU-a prevodi PTX u binarni kod. Inline PTX izrazi se mogu koristiti u CUDA-I. PTX definise viruelnu masinu i set instrukcija opste namene za paralelno izvrsavanje niti.