Porazdeljeni sistemi

6. Računanje na grafičnih procesnih enotah

Predavatelj: izr. prof. Uroš Lotrič
Asistent: Davor Sluga

Začnimo ...

z lahkim uvodom ...

Razvoj računalniških iger

- Nekateri smo začeli spoznavati računalnik takole …

Razvoj računalniških iger

- … potem so stvari za nekaj časa postale slabše…

Razvoj računalniških iger

- … trend se je sredi 90. let počasi začel obračati na bolje …

Razvoj računalniških iger

- … boljše, še boljše, do današnjih že zelo realističnih iger.

Strojna oprema

- Zakaj so pokrajine v igrah lahko postale toliko bolj realistične, dogajanje v igri tako hitro?

Grafični pospeševalniki

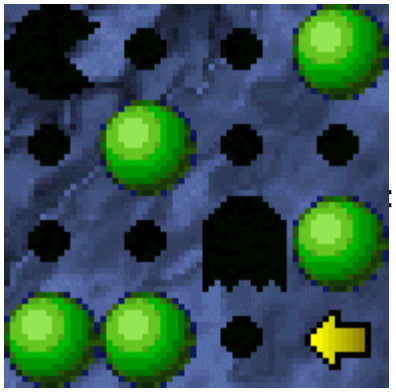
- Kaj so?
  - računalniška vezja z lastnim procesorjem in spominom, specializirana za prikazovanje računalniške grafike
    - grafične naloge opravijo hitreje od centralnih procesnih enot
    - hkrati razbremenijo centralne procesne enote
  - GPE = Grafična Procesna Enota
  - GPU = Graphics Processing Unit
- Pomembni izdelovalci
  - Nvidia,
  - ATI Technologies

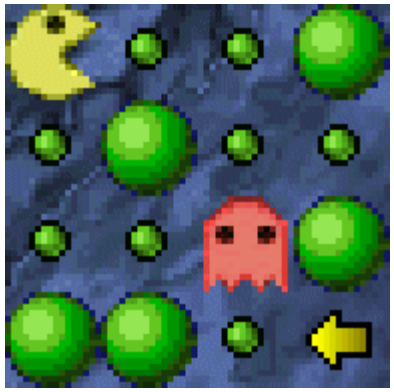
Grafični pospeševalniki

- Kaj počnejo?
  - prikazujejo sliko
  - predvsem pa računajo

Grafični pospeševalniki

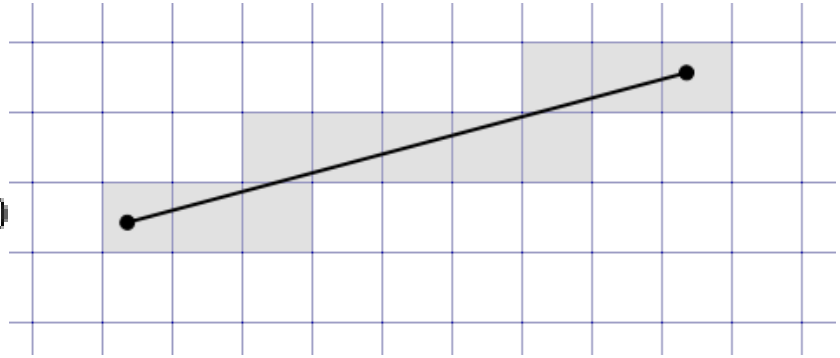
- Grafične operacije v dveh dimenzijah
  - bit-blit
    (BLock Image
    Transfer)





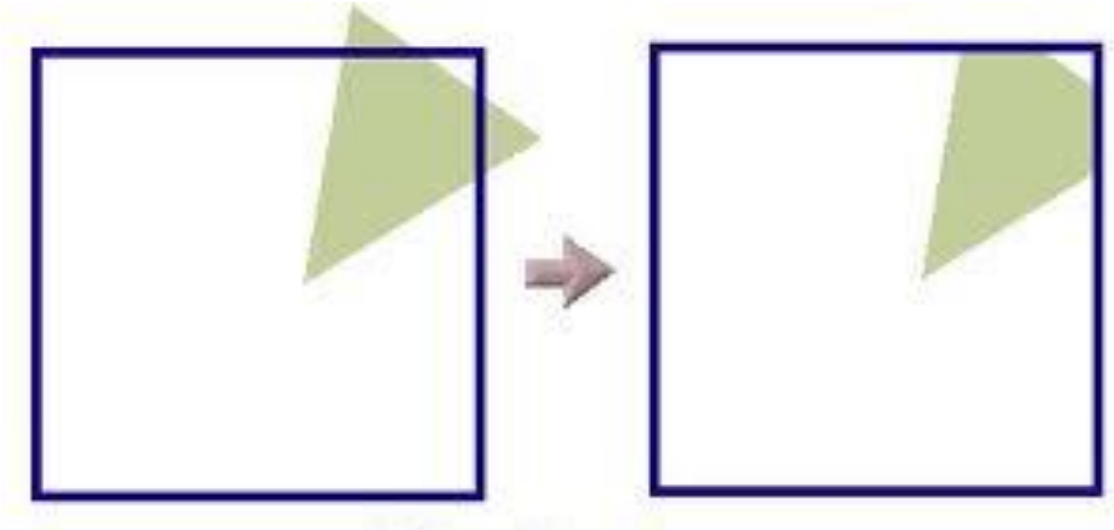
  - risanje črt

$$y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0$$



Grafični pospeševalniki

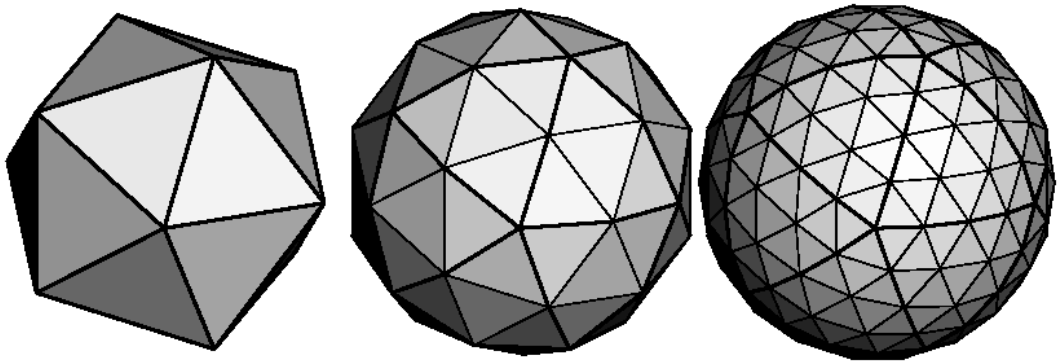
- Grafične operacije v dveh dimenzijah
  - barvanje in vzorčenje površin (zaprtih območij)
  - clipping



  - shranjevanje pogosto uporabljanih stvari v predpomnilnik

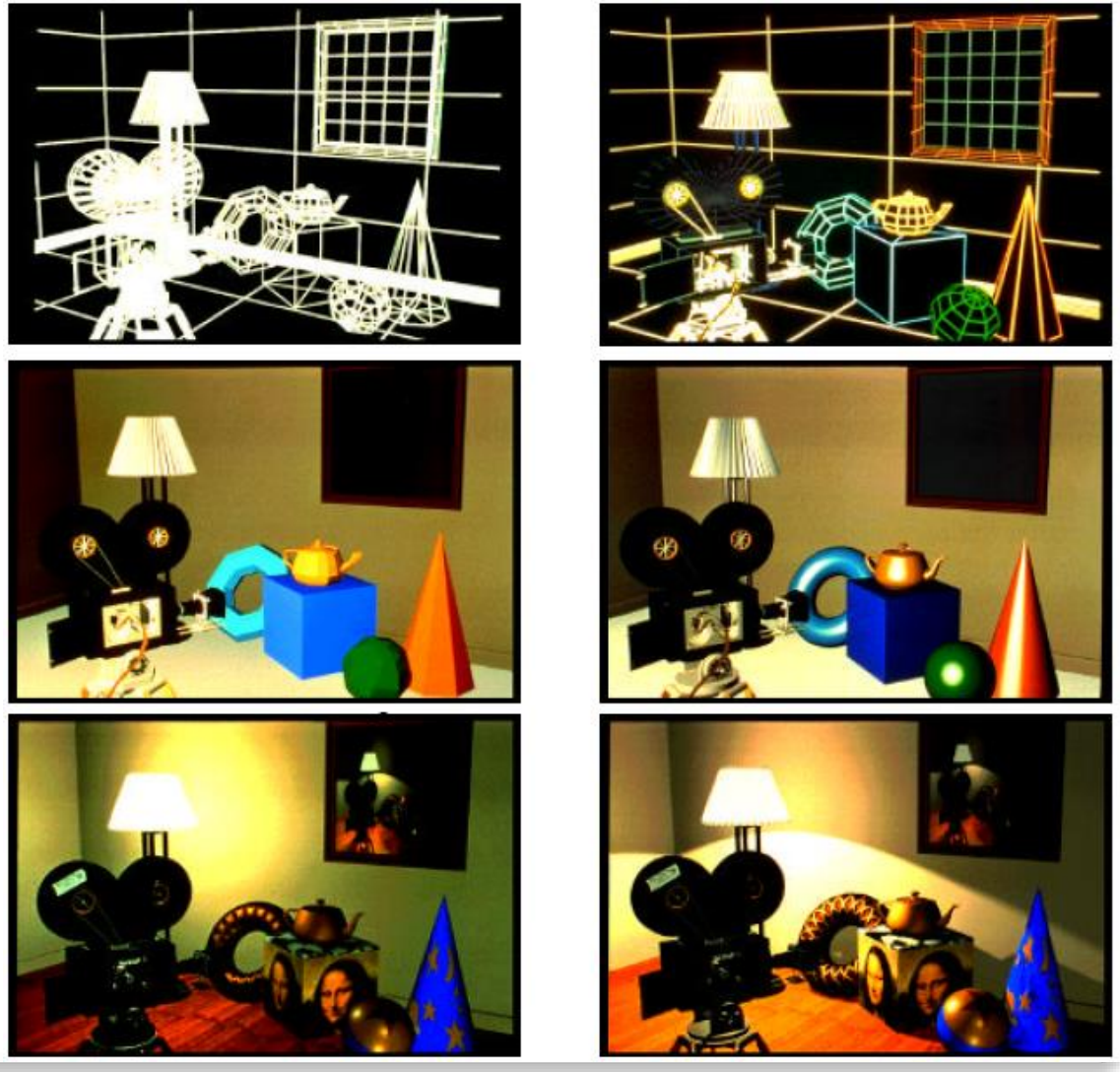
Grafični pospeševalniki

- Grafične operacije v treh dimenzijah
  - objekti 3D se običajno modelirajo s poligoni (trikotniki)
  - trikotnike se projicira na računalniški zaslon
  - večje kot je število trikotnikov, bolj realistična je slika



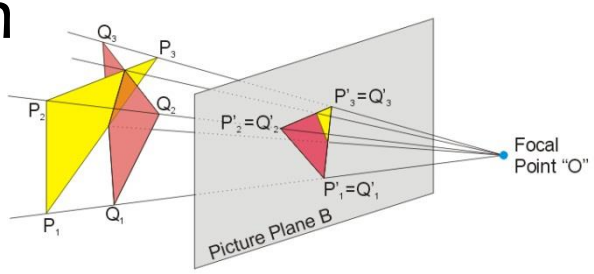
Grafični pospeševalniki

- Grafične operacije v treh dimenzijah
  - Postopek
    - žični okvir
    - nevidni robovi
    - barvanje
    - osvetlitev
    - teksture
    - senčenje



Grafični pospeševalniki

- Grafične operacije v treh dimenzijah
  - Za opisanimi postopki stojijo enačbe
    - Projekcije trikotnikov na želeno ravnino

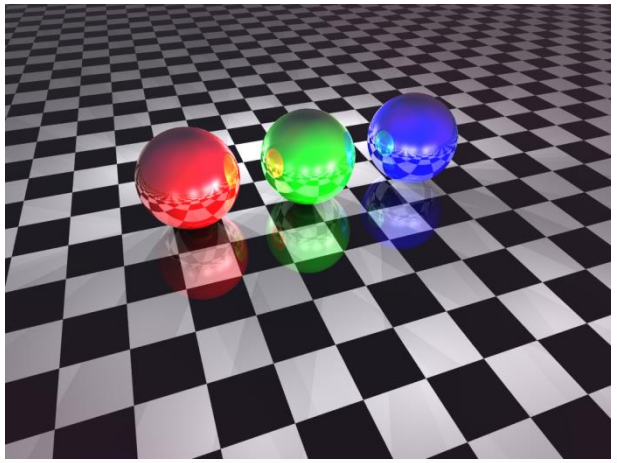


$$\begin{bmatrix}\mathbf{d}_x\\ \mathbf{d}_y\\ \mathbf{d}_z\end{bmatrix} = \begin{bmatrix}1 & 0 & 0\\ 0 & \cos\theta_x & -\sin\theta_x\\ 0 & \sin\theta_x & \cos\theta_x\end{bmatrix}\begin{bmatrix}\cos\theta_y & 0 & \sin\theta_y\\ 0 & 1 & 0\\ -\sin\theta_y & 0 & \cos\theta_y\end{bmatrix}\begin{bmatrix}\cos\theta_z & -\sin\theta_z & 0\\ \sin\theta_z & \cos\theta_z & 0\\ 0 & 0 & 1\end{bmatrix}\left(\begin{bmatrix}\mathbf{a}_x\\ \mathbf{a}_y\\ \mathbf{a}_z\end{bmatrix} - \begin{bmatrix}\mathbf{c}_x\\ \mathbf{c}_y\\ \mathbf{c}_z\end{bmatrix}\right)$$

    - Senčenje
      - Računanje presečišč s telesi, …

$$t = \frac{A \cdot o.x + B \cdot o.y + C \cdot o.z + D}{A \cdot dir.x + B \cdot dir.y + C \cdot dir.z}$$

$$\vec{T} = \left(n_r \cdot (\vec{N} \cdot \vec{I}) - \sqrt{1 - n_r^2 \cdot (1 - (\vec{N} \cdot \vec{I})^2}\right) \cdot \vec{N} - n_r \cdot \vec{I}$$



Grafični pospeševalniki

- Velikost zaslona 1440 x 900
  - Nekdo mora za vsako od 1,3 mio. točk izračunati vse enačbe in to vsaj 25 krat na sekundo
- Nalogo odlično opravijo grafični pospeševalniki

Zasnova

- Senčilnik slikovnih točk (pixel shaders 2D) :
  - njegova naloga je, da za vsako točko določiti njeno barvo - vhodni podatki so numerični: barve svetlobnih teles, lastnosti materialov, teksture, …
- Senčilnik vozlišč (vertex shaders 3D):
  - Njegova naloga je, da za vsa vozlišča v navideznem 3D prostoru opravi vse geometrijske transformacije, potrebne, da se posamezno vozlišče prenese v zaslonske koordinate
    - transformacije modelov,
    - transformacija 3D pogledov,
    - perspektivna transformacija,
    - transformacija 2D pogledov,
    - rasterizacija

Zasnova

- Senčilniki slikovnih točk ter oglišč so bili implementirani kot ločene procesne enote
- Težave
  - Če imamo v neki aplikaciji veliko geometričnega računanja in malo računanja barv, potem so senčilniki vozlišč preobremenjeni, medtem ko senčilniki slikovnih točk ne delajo

  - Če imamo v neki aplikaciji veliko barvnega računanja in malo ali skoraj nič geometričnega, potem so senčilniki slikovnih točk preobremenjeni, medtem ko senčilniki vozlišč ne delajo

Odgovor – CUDA in OpenCL

- Arhitektura CUDA

  (Compute Unified Device Architecture)
  - 2006: NVIDIA GeForce 8800 GTX – prva naprava z arhitekturo CUDA
  - Vsaka ALE na čipu sedaj po potrebi računa, ali izvaja operacije senčenja slikovnih elementov ali oglišč
  - Enote ALE so razširili tako, da podpirajo IEEE standard za plavajočo vejico
  - Razširili so nabor ukazov: več splošno namenskih operacij
  - Procesorji lahko naključno dostopajo do pomnilnika (pišejo v in berejo iz poljubne pomnilniške besede)

Odgovor – CUDA in OpenCL

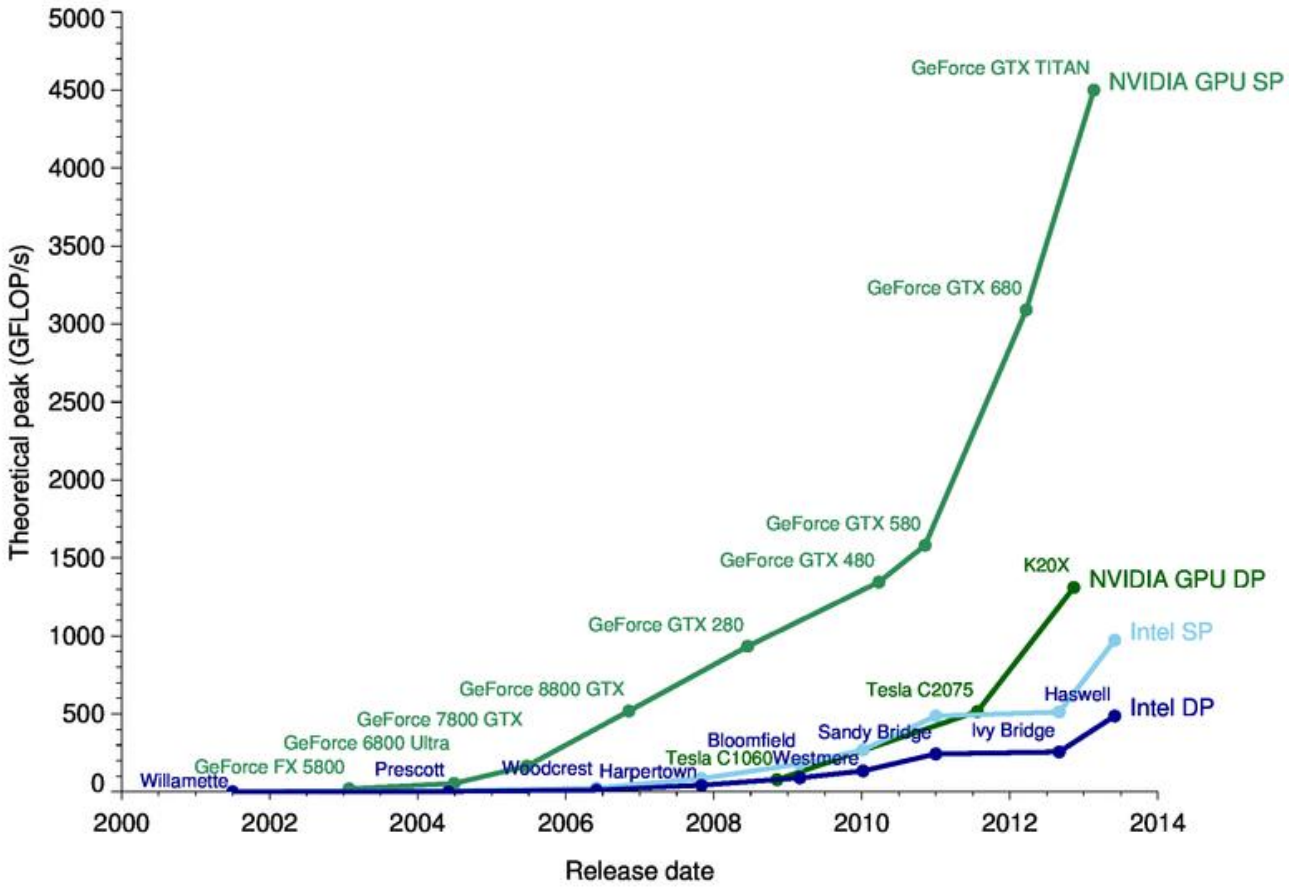
- V Nvidia omogočijo programerjem bolj neposreden dostop do grafične strojne opreme:
  - CUDA C – programski vmesnik z razširitvami programskega jezika C
  - Kmalu po izidu GeForce 8800 GTX, brezplačen prevajalnik za CUDA C
  - CUDA C postane prvi programski jezik, ki omogoča bolj splošno namensko računanje na grafični strojni opremi

Superračunalniki

- Titan - Cray XK7
  - 18.688 vozlišč
  - vozlišče
    - procesor AMD Opteron 6274, 16 jeder
    - 32 GB pomnilnika
    - Nvidia TESLA K20X s 6GB pomnilnika
  - 40 PB diskovja

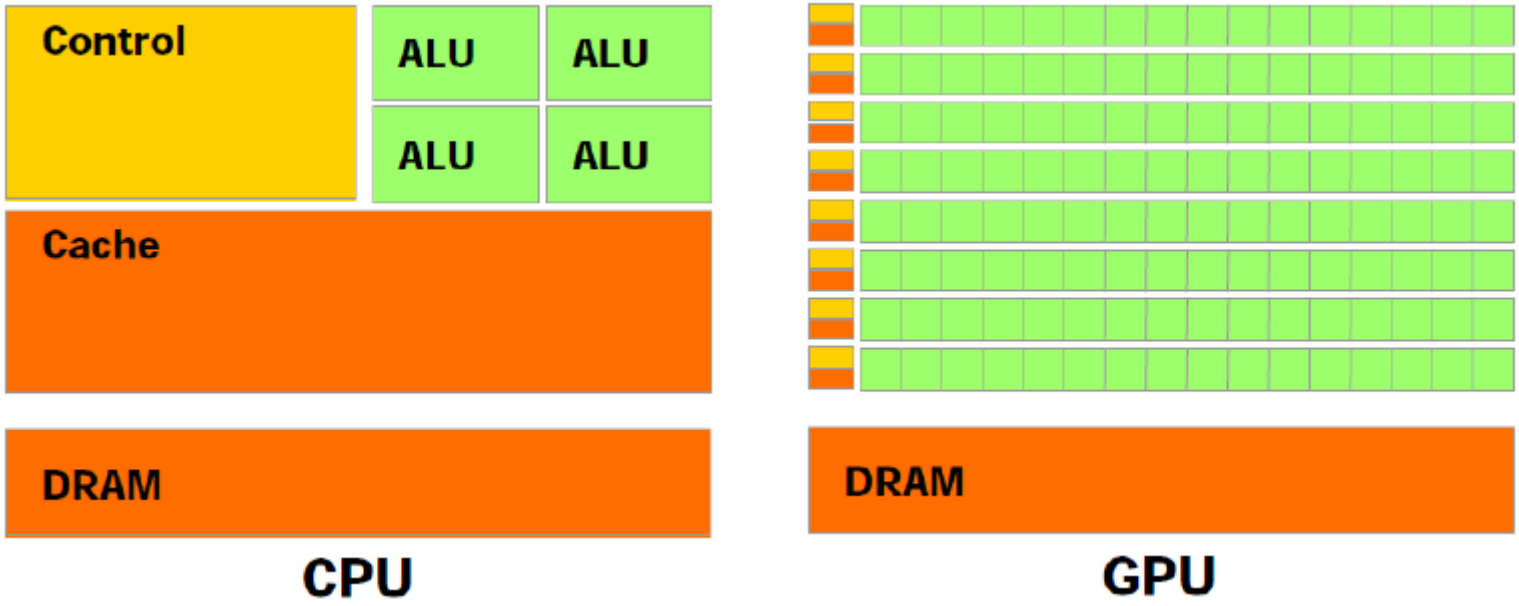
Računanje na grafičnih procesorjih

- Naraščanje zmogljivosti: grafični procesorji (zelena) proti centralnim procesorjem (modra) v enojni (SP) in dvojni (DP) natančnosti



Računanje na grafičnih procesorjih

- Grafični procesorji so računsko zelo zmogljive naprave
  - Delež tranzistorjev: CPU vs GPU

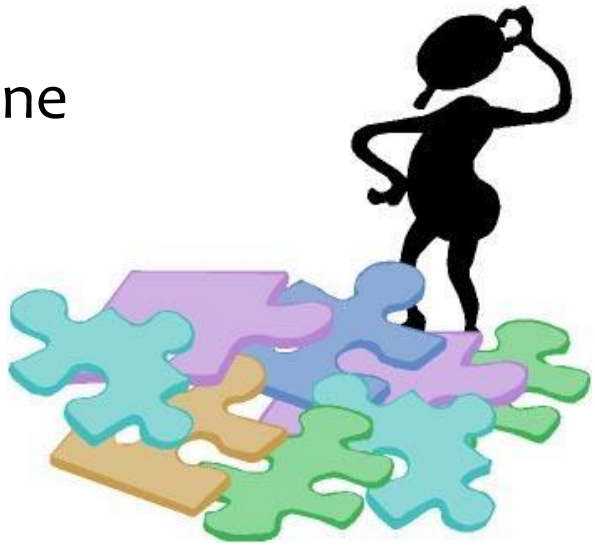


Problemi

- Grafične procesorje je težko uporabljati
- Zasnovani so bili za hitro prikazovanje grafike
  - neobičajen programski model
  - ukazi so vezani na računalniško grafiko
  - precej omejitev pri programiranju
- Strojna oprema je
  - močno paralelna
  - se močno razvija in spreminja
  - zasnova procesorjev precej tajna
- Programov iz CPE ne moremo enostavno prenesti

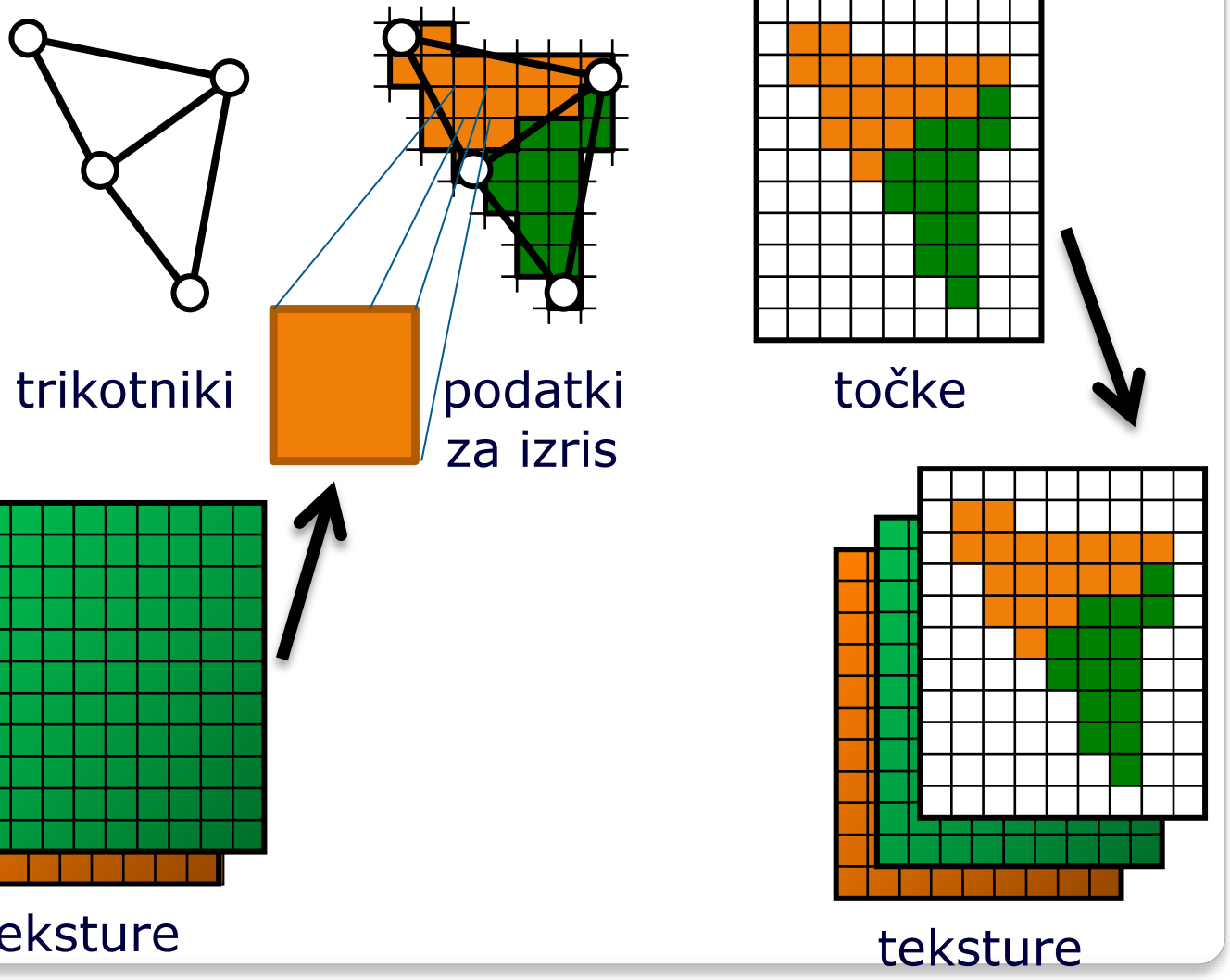
Problemi

- Grafični procesorji niso zdravilo za vse izzive
  - so hitri, ker so ozko specializirani
  - ne znajdejo se v zaporednem svetu
- Zgodovina
  - brez podpore za pomembne matematične operacije: cela števila, bitne operacije
  - paralelizem tipa SPMD (Single Process Multiple Data)
- Danes je mnogo pomanjkljivosti odpravljenih



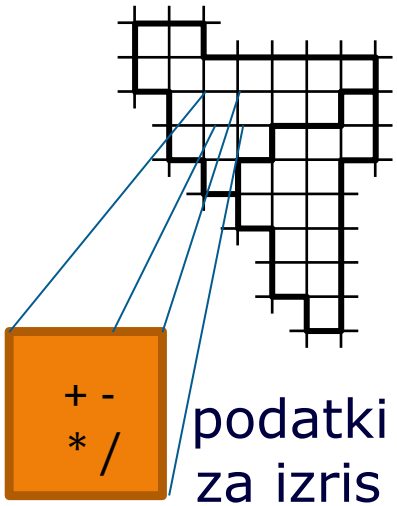
Programiranje

- Grafika
  - slike so sestavljene iz množice objektov
  - objekte moramo obdelati na enak način
  - natančno predpisani postopki

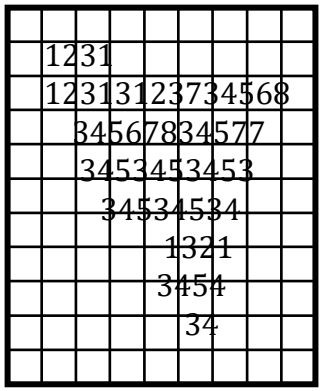


Programiranje

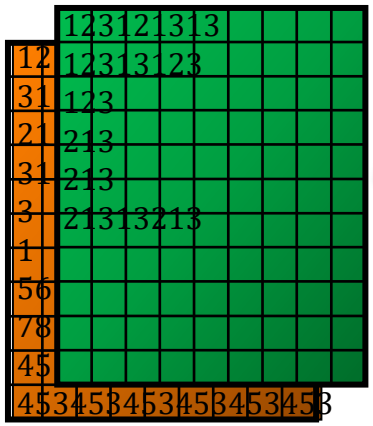
- Splošno
  - Abstrakcija
  - Namesto grafičnih elementov si predstavljamo številke



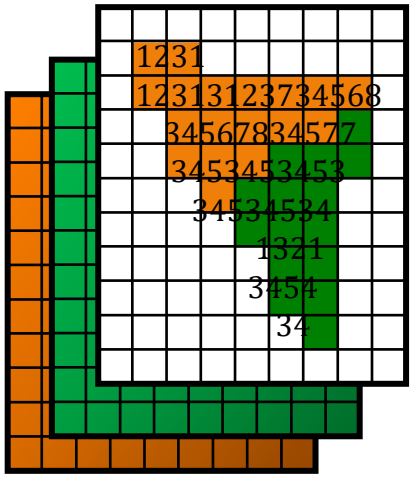
podatki za izris



točke



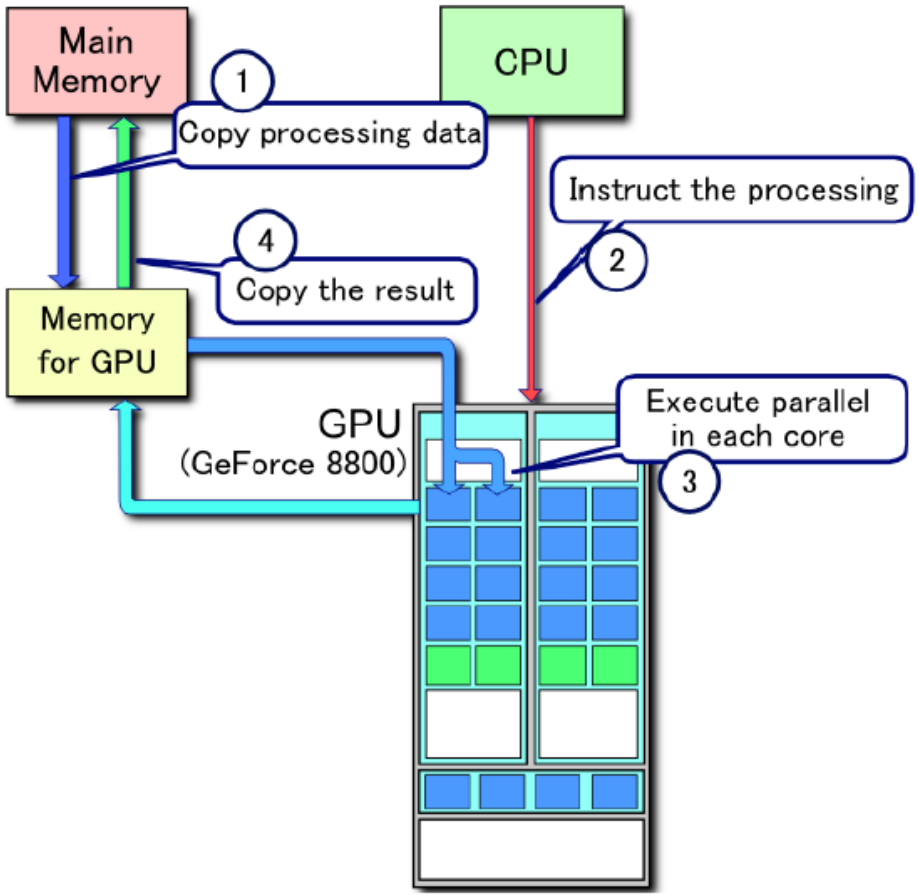
teksturni pomnilnik



teksturni pomnilnik

Programiranje

- Postopek
  1. prenos podatkov na grafični procesor
  2. zahteva za izračun
  3. računanje
  4. prenos podatkov v glavni pomnilnik

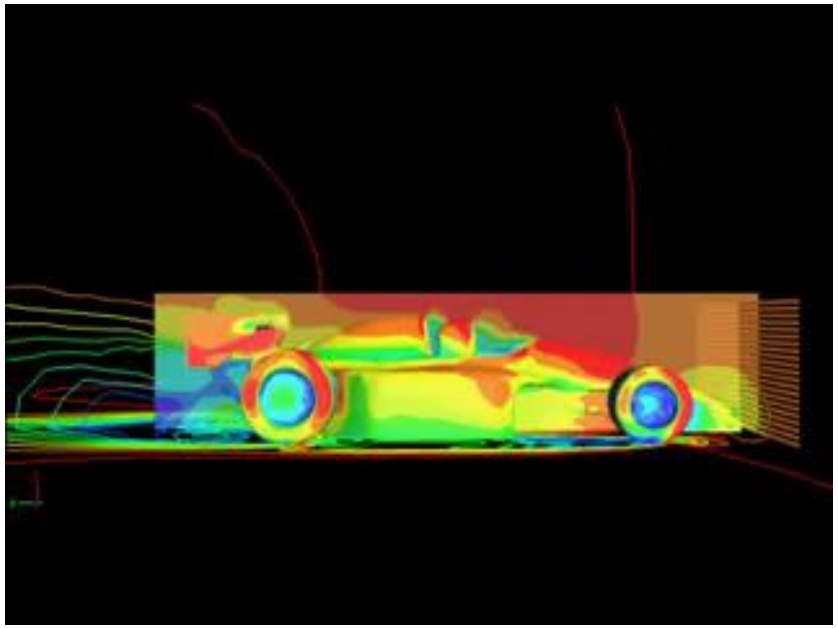


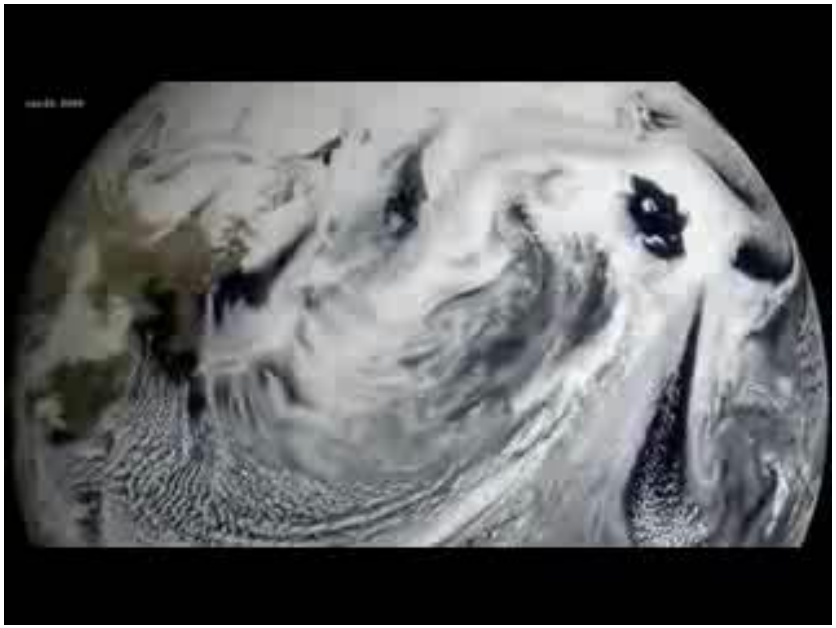
Kje to rabimo?

- Veliko problemov je računsko zelo zahtevnih
- Mnogi so pisani na kožo arhitekturi grafičnih procesnih enot (GPE)

Kje to rabimo?

- Raziskovanje dinamike tekočin in turbulenc
- Modeliranje vremena in obnašanja okolja





- kemija, fizika, statistična mehanika, razvoj materialov, raziskovanje vesolja, astrofizika, medicina, …
- Biologija, genetika, genski inženiring, določanje zgradbe proteinov, delovanje encimov, modeliranje celice, razvoj zdravil

Kje to rabimo?

- Problemi z visoko stopnjo podatkovnega paralelizma:
  - procesiranje slik,
  - simulacija fizikalnih modelov,
  - problemi N teles,
  - iskanje in urejanje,
  - dinamika fluidov,
  - vektorska polja, ...

Kdaj pa na GPE lahko pozabimo?

- Problemi, ki niso pisani na kožo arhitekturi grafičnih procesnih enot (GPE):
  - operacije nad drevesi,
  - operacije nad podatki v povezanih seznamih, ...

… in nadaljujmo …

mnogo bolj zares …

Izvajalni model CUDA

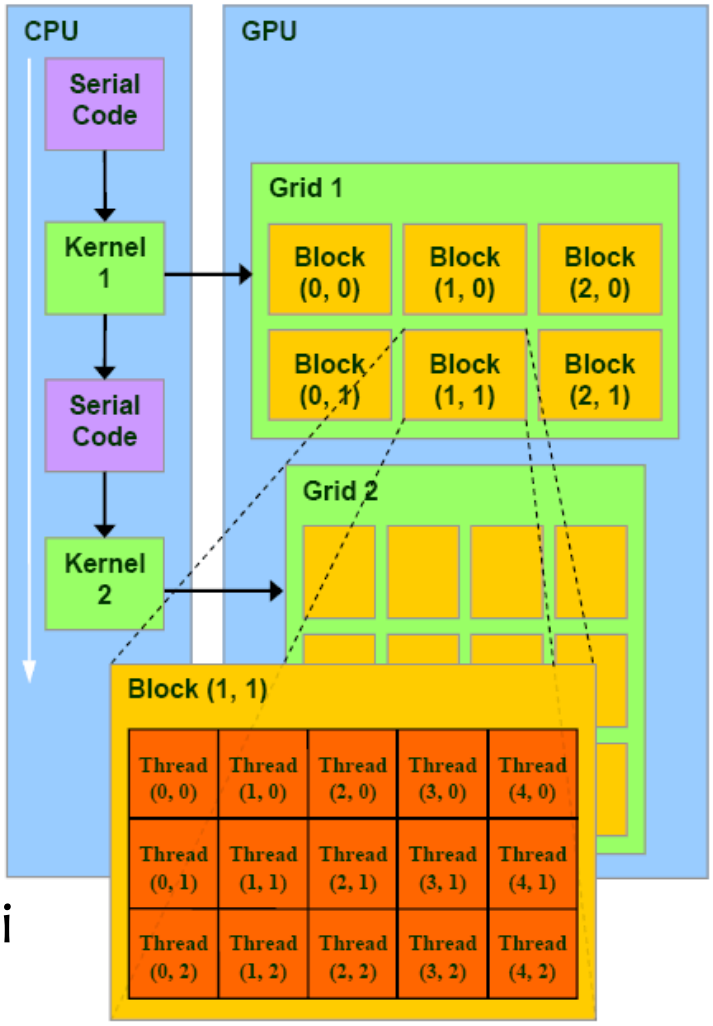
- Filozofija:
  - ustvarimo virtualno neomejeno število vzporednih niti, ki se bodo dinamično razvrščale in izvajale na stroji opremi
- Programski model: na GPE gledamo
  - kot na soprocesor, ki podpira neomejeno število niti,
  - s programsko vidnim hierarhičnim pomnilnikom
- GPE je praviloma uporabna le pri problemih, kjer imamo visoko stopnjo podatkovnega paralelizma
  - operacije se izvajajo na veliki količini podatkov,
  - ti so običajno porazdeljeni v podatkovni strukturi 2D/3D polje (mreža)

Izvajalni model CUDA

- Program sestavljajo:
  - ščepci:
    - ščepci so samostojni kosi programske kode,
      ki se izvajajo na napravi GPE,
    - v nekem trenutku izvaja naprava GPE en sam ščepec
      (na novejših sistemih je ščepcev lahko več)
  - serijska koda:
    - izvaja se na gostitelju
      - gostitelj je običajna CPE
    - serijska koda je potrebna za
      - prenos kode posameznega ščepca na napravo GPE
      - za prenos podatkov od gostitelja na napravo GPE in nazaj

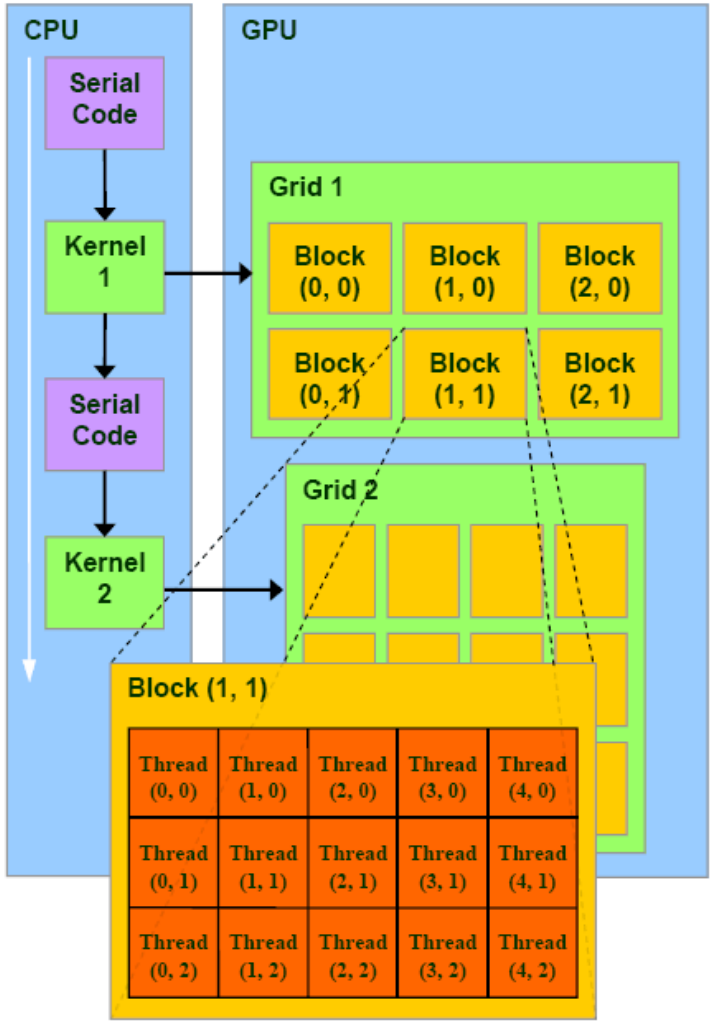
Izvajalni model CUDA

- Ščepec (ang. kernel)
  - sestavlja veliko število niti
  - niti so logično porazdeljene v polje
- Organizacija niti v ščepcu
  - blok niti (ang. block)
    - je skupek do največ 1024 niti
    - posamezne niti v bloku izvajajo enako programsko kodo
    - vse se začnejo izvajati z istim ukazom
    - med seboj lahko komunicirajo preko skupnega pomnilnika
  - mreža niti (ang. grid)
    - je sestavljena iz neodvisnih blokov niti
    - v mreži je največ 65535 blokov



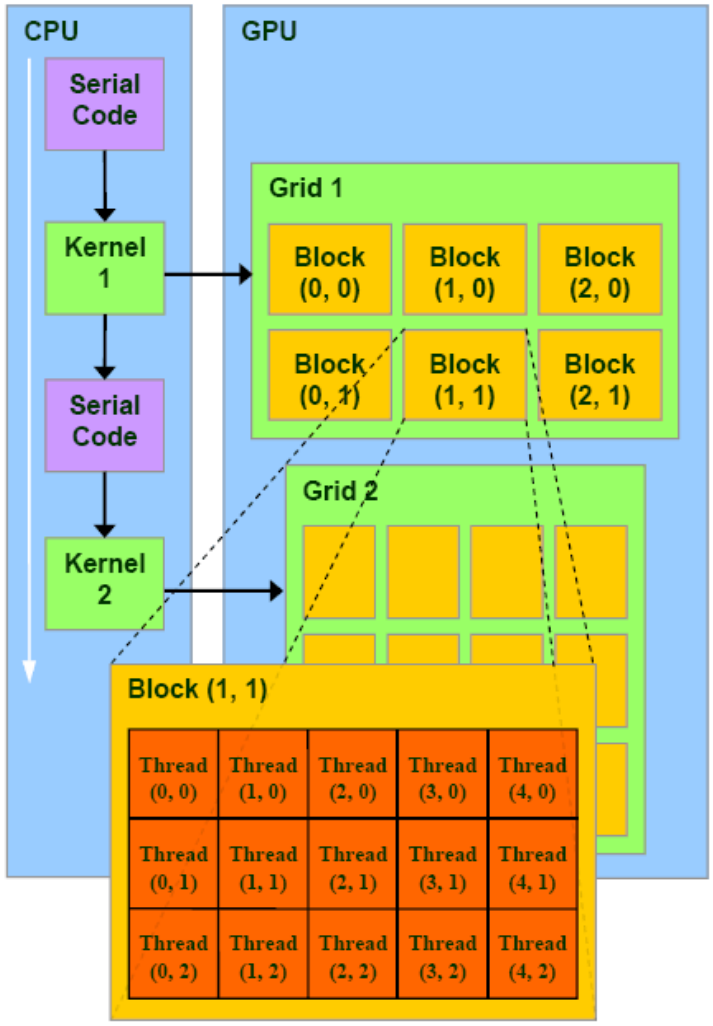
CUDA izvajalni model

- Izvajanje
  - Bloki se izvajajo
    - lahko vzporedno
    - v poljubnem vrstnem redu
  - Ščepec
    - če je pravilno organiziran v bloke
      - se bo pravilno izvajal na napravi GPE, ki lahko istočasno izvaja samo en blok, ali vzporedno več blokov
      - na ta način je zagotovljena visoka raztegljivost (ang. skalabilnost) programske opreme



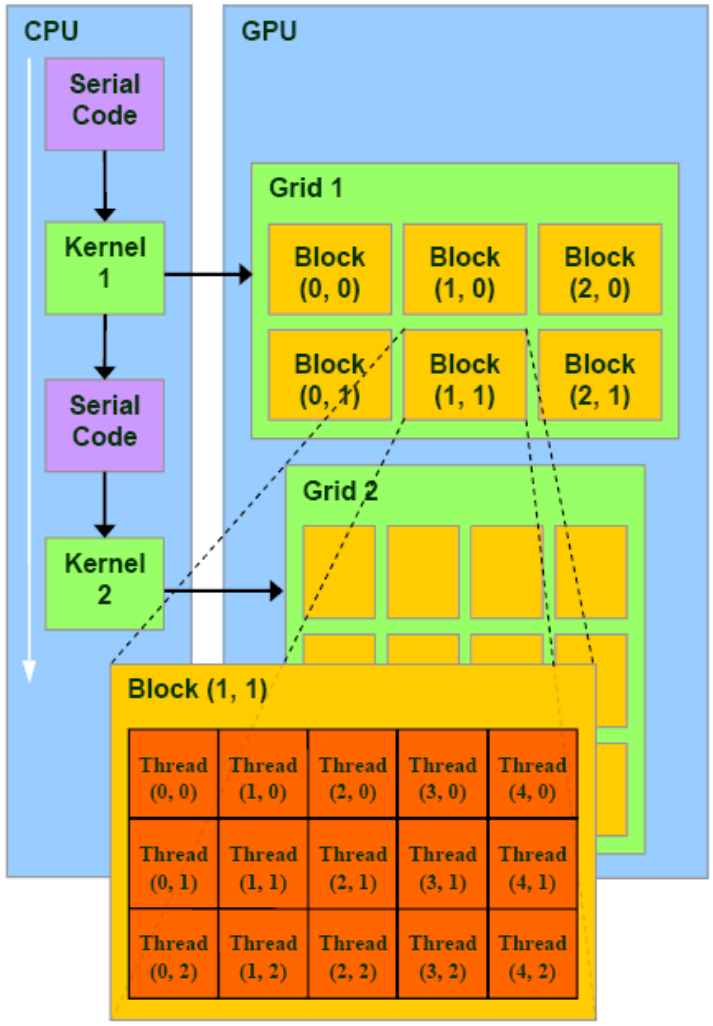
CUDA izvajalni model

- Programski model
  - Bloki
    - v mreži so logično razporejeni v 2D polje
    - vsak blok v mreži je programsko viden preko 2D indeksov
  - Niti
    - v blokih so logično razporejene v 2D ali 3D polje
    - vsaka nit v bloku je programerju vidna preko 2D ali 3D indeksov
  - Na ta način lahko niti prilagajamo dejanski organizaciji podatkov, ki jih obdelujemo

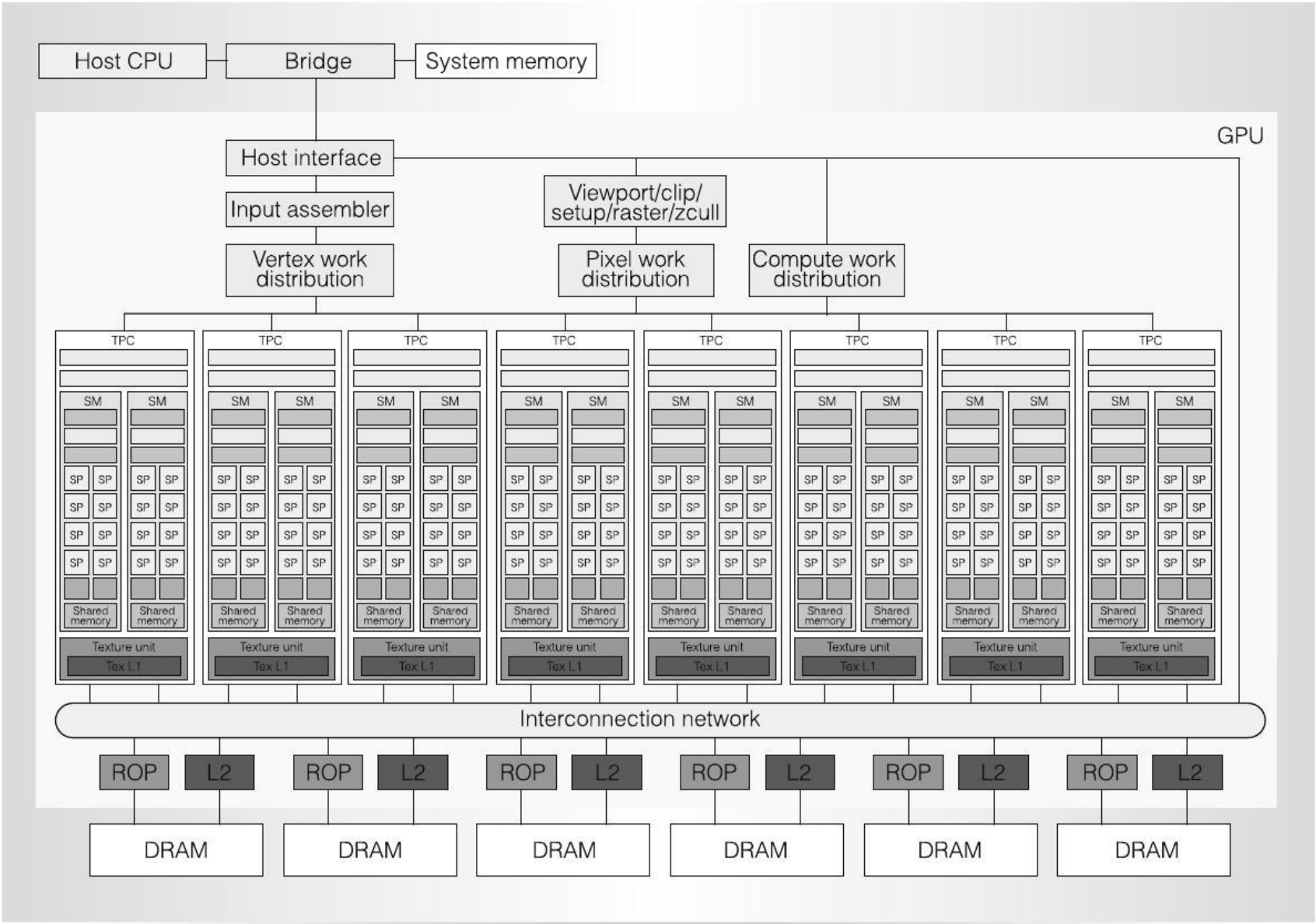


CUDA izvajalni model

- En ščepec je sestavljen iz samo ene mreže
  - to je trenutna omejitev, ki se bo mogoče v prihodnje sprostila
- Velikost mreže in izvajanje programov
  - nov ščepec ne more začeti z izvajanjem, dokler se ne izvedejo vse niti v prejšnji mreži
  - izvajanje blokov lahko po potrebi eksplicitno sinhroniziramo v programski kodi

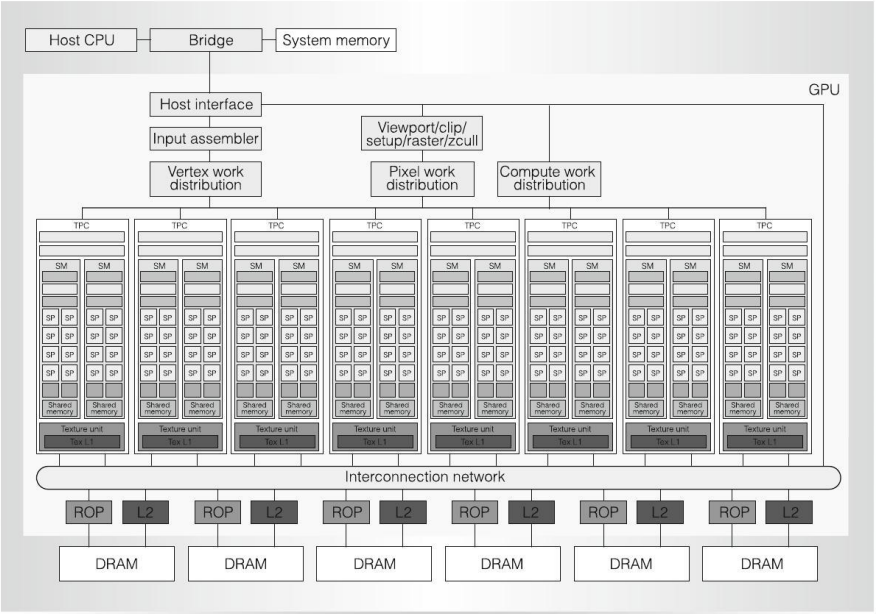


Nvidia GeForce 8800



Nvidia GeForce 8800

- Procesor SM (Streaming Multiprocesor)
  - Enemu SM se v izvajanje dodeli en blok niti
- Izvajalne enote SP (Streaming Processor)
  - Vsaka nit se izvaja v eni izvajalni enoti SP

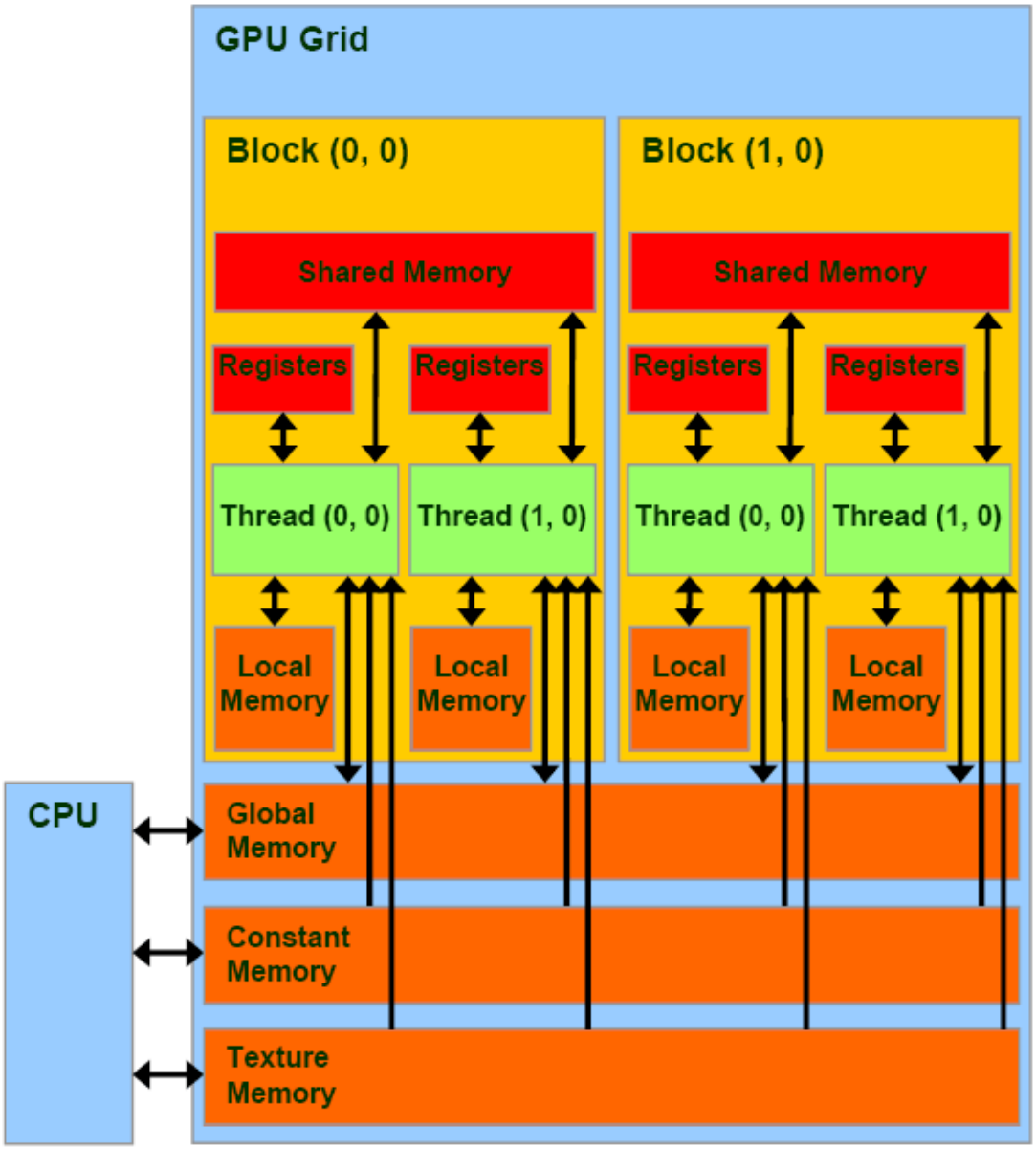


- Posameznemu procesorju SM pripada skupni ali deljeni pomnilnik (Shared Memory)
  - hiter dostop
  - preko njega si niti v bloku lahko izmenjujejo podatke
- Globalni pomnilnik DRAM
  - Preko njega si lahko podatke izmenjujejo niti iz različnih blokov

Pomnilniška hierarhija

- Na napravi GPE je celoten programsko viden pomnilnik sestavljen iz naslednjih ločenih pomnilniških prostorov:
  - registri (register file), 1 urina perioda
  - deljeni pomnilnik (shared memory), 5 urinih period
  - lokalni pomnilnik (local memory), 500 urinih period
  - globalni pomnilnik (global memory), 500 urinih period
  - pomnilnik konstant (constant memory), 5 urnih period
  - pomnilnik tekstur (texture memory), 5 urinih period

Pomnilniška hierarhija



Registri

- Registrski niz je privaten za vsako nit
  - Tesla (=GT200): 16K x 32 bitov v enem procesorju SM
  - Fermi (=GF100): 32K x 32 bitov v enem SM
  - Kepler (=GK110): 65K x 32 bitov v enem SM
  - Vsaki izvajalni enoti SP pripada 2K registrov (jih ni dovolj za vse!!!)
  - Izvajalna enota hkrati izvaja 32 niti
  - Nit lahko uporablja do 127 (Tesla), 63 (Fermi), 255 (Kepler) registrov
- Organizacija
  - registri so 32-bitni
  - implementirani so na isti rezini kot procesorji SM
- Dodeljevanje registrov
  - statično: za vsak blok
  - dinamično: za vsako nit v bloku od 4 – 128 registrov

Skupni pomnilnik (Shared Memory)

- Vsakemu procesorju SM pripada:
  - 16 kB skupnega pomnilnika (Tesla)
  - 16/48 kB skupnega pomnilnika (Fermi)
  - 16/32/48 kB skupnega pomnilnika (Kepler)
  - implementiran na isti rezini, kot SM (dostop 1 urina perioda)
- Organizacija pri GT200:
  - 16 modulov (bank) x 256 x 32 bit (16x256x4=16 kB)
  - skupaj 4096 x 32 bitov
- Dodeljevanje
  - dinamično posameznim blokom, ki se izvajajo na nekem procesorju SM
  - do pomnilnika, dodeljenega posameznemu bloku, lahko dostopa do 512 niti

Skupni pomnilnik (Shared Memory)

- Značilnosti
  - omogoča hitro komunikacijo med nitmi v istem bloku
  - podatki iz globalnega pomnilnika se s pomočjo ukaza LOAD prenesejo le v registre
  - če želimo podatke prenesti iz globalnega pomnilnika v skupni pomnilnik, jih moramo najprej prenesti v registre in nato iz registrov shraniti v skupni pomnilnik
  - uporabljamo ga
    - za shranjevanje skupnih podatkov
      - primer: skupni števec za vse niti v bloku,
    - za shranjevanje podatkov, do katerih niti pogosto dostopajo, ...

Lokalni pomnilnik

- Lastnosti
  - implementiran je v pomnilniku DRAM
  - privaten za vsako nit (podobno kot registri)
  - relativno dolga latenca pri dostopu
  - uporablja se ga, kadar zmanjka registrov

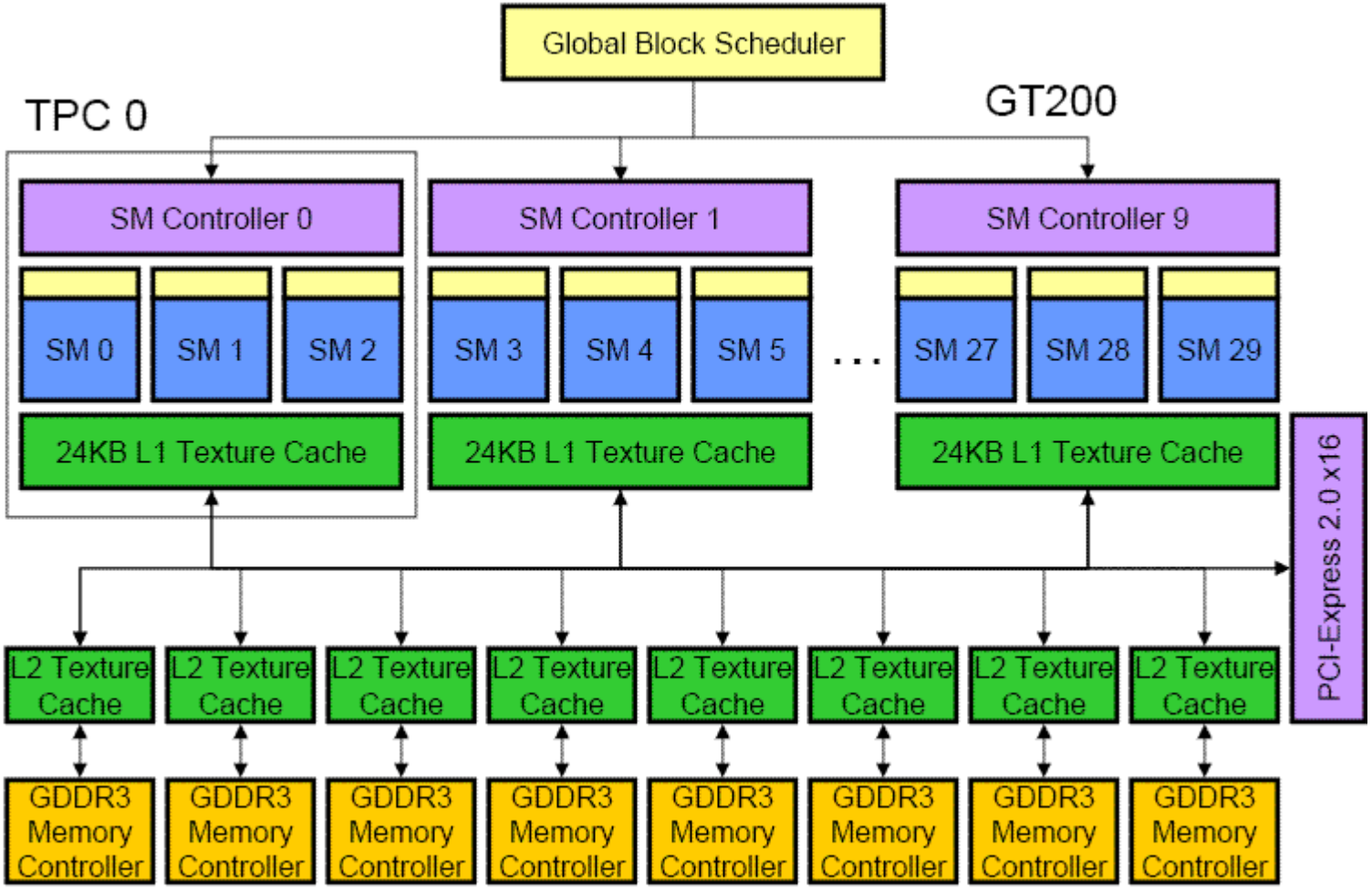
Globalni pomnilnik

- Značilnosti
  - viden je vsem nitim v mreži (vsem nitim v ščepcu)
  - za branje in pisanje je dostopen
    - iz gostiteljske CPE in
    - iz naprave GPE
  - (na GT200 ni predpomnjen)
  - implementiran v GDDRx (double data rate)
- GT200:
  - 8 x 64-bitnih GDDR3 krmilnikov – 512 bitno vodilo med procesorji SM in globalnim pomnilnikom
  - krmilniki delajo pri 1107 MHz:
    - teoretično:
      - 1107 MHz x 2 = 2,214 G prenosov/s
      - 2,214 G/s x 64 B = 141.696 GB/s

Pomnilnika konstant in tekstur

- Na vsaki napravi GPE sta še dva bralna pomnilniška prostora:
  - pomnilnik konstant
  - pomnilnik tekstur
- Ostanka grafične narave naprav GPE
- Oba sta bralno predpomnjena na napravi
  - v primeru, da CPE piše, se predpomnilnik razveljavi
- Pomnilnik konstant
  - realiziran v DRAM-u
  - velikost: 64 KB
  - uporablja se ga za ukaze
- Pomnilnik tekstur
  - realiziran v DRAM-u
  - ima dvodimenzionalno lokalnost
  - uporablja se ga za shranjevanje konstant in tekstur v grafičnih aplikacijah

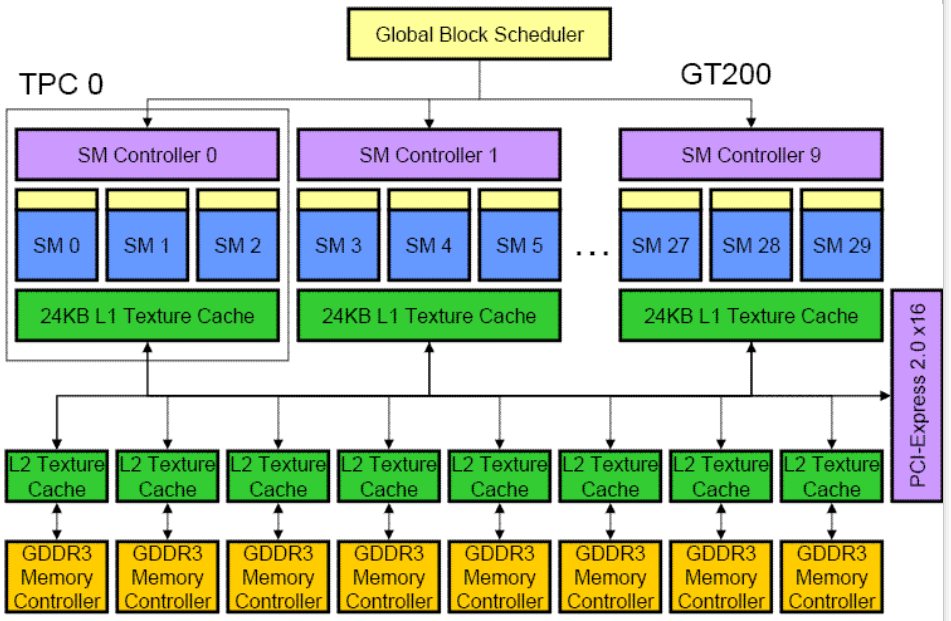
Zgradba GT200



TPC – Texture/Processor Cluster

Zgradba GT200

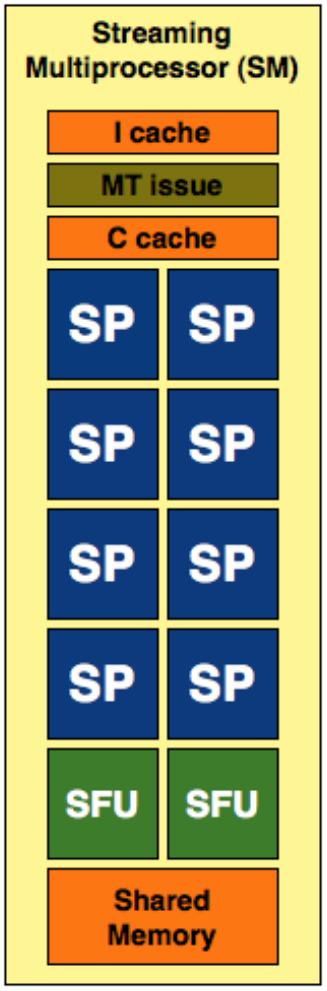
- GT200 je večjedrni procesor z dvema nivojema hierarhije
  - 1. nivo: 10 večjedrnih procesorjev TPC (Texture Processor Cluster)
  - 2. nivo: vsak TPC ima
    - 3 jedra s procesorji SM (Streaming Multiprocessors)
    - cevovod za dostop do globalnega pomnilnika (texture pipeline)



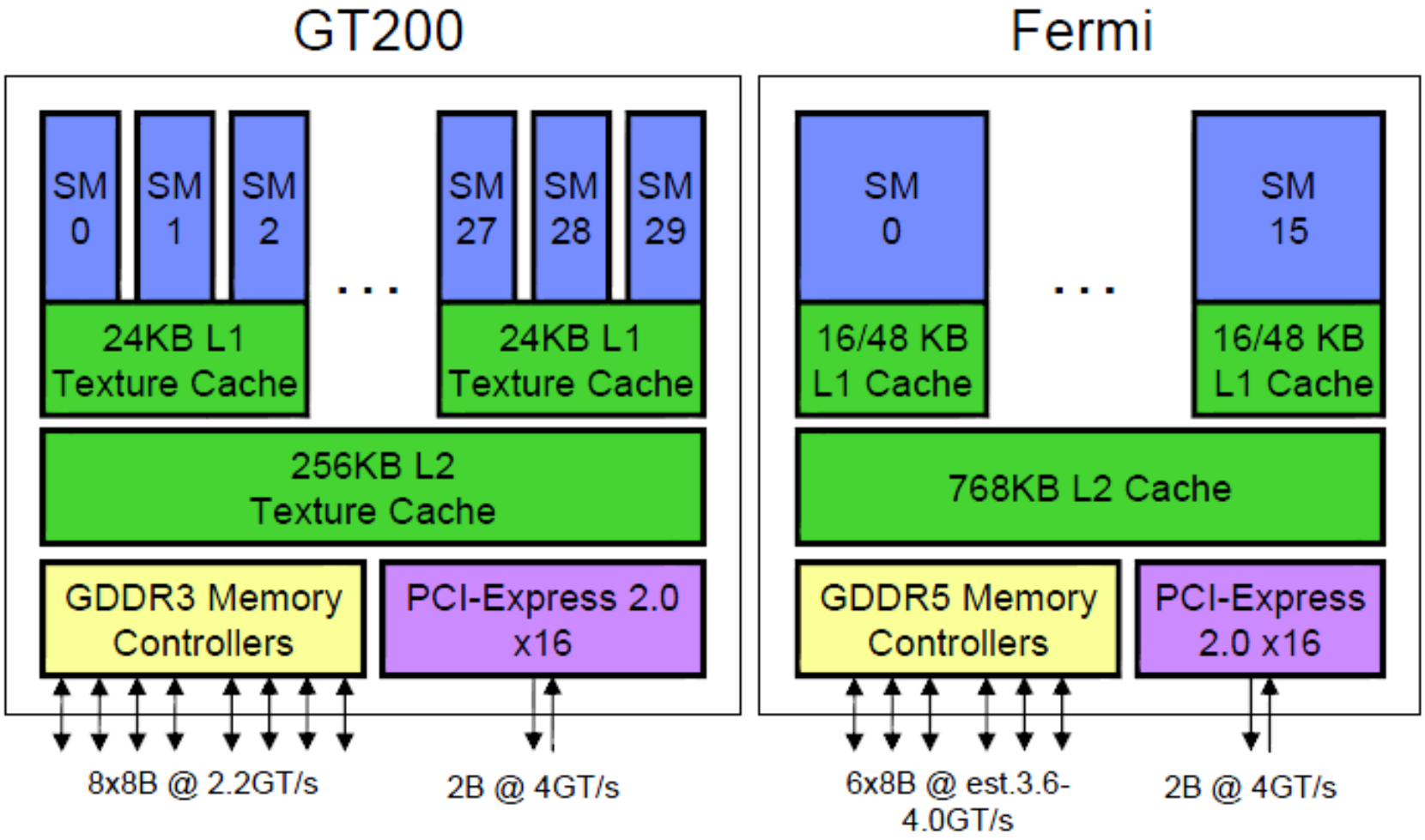
- Vsak procesor SM v TPC je v bistvu samostojen procesor
  - ima celotno logiko za zajem dekodiranje in izstavljanje ukazov, izvajalne (funkcijske) enote,
  - vendar si procesorji SM v istem TPC delijo logiko za dostop do globalnega pomnilnika

Zgradba GT200, SM

- Vsak SM ima 8 blokov izvajalnih enot SP
- Izvajalna enota SP ima
  - lasten programski števec
  - lastne registre
    (dodeli se mu del registrskega niza)
  - manjka mu pa vsa logika za zajem,
    dekodiranje in izstavljanje ukazov
- Izvajalne enote SP
  - niso pravi procesorji
  - so zelo podobne izvajalnemu delu cevovoda
    v modernih procesorjih



Zgradba Nvidia Tesla in Fermi



Zgradba Nvidia Fermi

- Fermi, SM
  - 32 SP
  - Dva razvrščevalnika snopov niti (pri Tesla en sam)
  - Oznake:
    - Core – SP
    - LD/ST – računanje naslovov vira in ponora
    - SFU – Special Function Unit

Zgradba Nvidia Kepler

Zgradba Nvidia Kepler

- Fermi, SM
  - 192 SP enot
  - 32 SFU enot
  - 32 LD/ST enot
  - 64 DP enot
  - 4 razvrščevalniki snopov niti
  - Oznaka
    - DP – Double Precision Unit

Zgradba Nvidia Kepler

- K20 – lastnosti
  - Zmogljivost (dvojna natančnost): 1,17 TFLOPS
  - Zmogljivost (enojna natančnost): 3,52 TFLOPS
  - Prenos podatkov: 208 GB/s
  - Pomnilnik: 5 GB
  - Število jeder: 13 (SMX) x 192 = 2496
  - Poraba: 225 W

GT200, SM

- Najmanjša samostojna procesna enota v GPU napravi
- Vsaka enota SM skrbi za
  - ‘istočasno’ izvajanje do 8 blokov nitk
  - vendar največ 1024 nitk
- Globalni razvrščevalnik (Global Block Scheduler)
  - dinamično razporeja bloke na tiste SM, ki imajo proste vire

GT200, SM

- Bloki in nitke so arhitekturno vidni programerju
  - strojna oprema z njimi ne zna učinkovito delati
  - na mikro arhitekturnem nivoju se razvršča in izvaja snope nitk (warps)
  - snopi niso programsko vidni (programerji jih ne vidijo in nič ne vedo o njih)
- Snop
  - je sestavljen iz 32 zaporednih nitk, ki izvajajo isto kodo
  - vsak SM lahko skrbi za izvajanje 32 snopov (vsega skupaj je to 32 x 32 = 1024 nitk)

GT200, SM

- Lastnosti
  - Enota SM za razliko od modernih super-skalarnih procesorjev nima špekulativnega izvajanja in napovedovanja vejitev
  - SM predvideva, da bodo vse nitke v snopu izvajale popolnoma isto kodo, ki ne vsebuje vejitev
  - SM so zaradi tega enostavnejši in porabijo manj energije
- Pri Nvidia pravijo takšnemu načinu izvajanja

  SIMT – Single Instruction Multiple Threads

GT200, SM

- Hitrost izvajanja niti
  - najbolj učinkovito izvajanje imamo takrat, ko niti ne vejijo – takrat vse niti v snopu potrebujejo iste izvajalne (funkcijske) enote
  - če imamo v snopu N različnih vejitev, računske zmogljivosti padejo približno za faktor N
    - pri N različnih vejitvah v snopu, ima v nekem trenutku N niti različne vrednosti programskega števca
    - takrat se izvaja samo ena nit, ostalih N-1 niti pa čaka
- Niti v snopu med seboj lahko komunicirajo le preko skupnega pomnilnika, saj so njihovi registri privatni
  - to je bistvena razlika v primerjavi z načinom SIMD (Single Instruction Multiple Data)

SM – funkcijske enote

- Vsako urino periodo se izstavi en ukaz, ki hkrati krmili do 8 funkcijskih enot
  - 8 32-bitnih FP ALE/MAD enot
    - floating point (FP), aritmetično logična enota (ALE), multiply and add (MAD)
  - 8 celoštevilskih enot za izvajanje skočnih ukazov
  - 1 64-bitna enota za množenje v plavajoči vejici
  - 2 enoti za računanje posebnih funkcij (Special Function Unit, SFU)
    - transcedenčne operacije, recipročne vrednosti, koren, ...
- Te funkcijske enote delajo z dvakrat višjo frekvenco (hitra ura) kot poteka zajemanje in izstavljanje ukazov!
- Večinoma se ukazi izvajajo v 32-bitnih FP ALE/MAD enotah
  - te izvajajo večino aritmetičnih operacij
  - običajno se en ukaz izstavi vsem ALE/MAD enotam, pri čemer vsaka uporablja drugačne podatke

SM – funkcijske enote

- 64 bitno funkcijska enota MAD (Multiply and Add)
  - omogoča izvajanje 64-bitnih ALE operacij
  - ker obstaja le ena taka enota v SM, se moramo zavedati da so 64-bitne operacije od 8 do 12 krat počasnejše od 32-bitnih operacij

SM – izvajanje ukazov

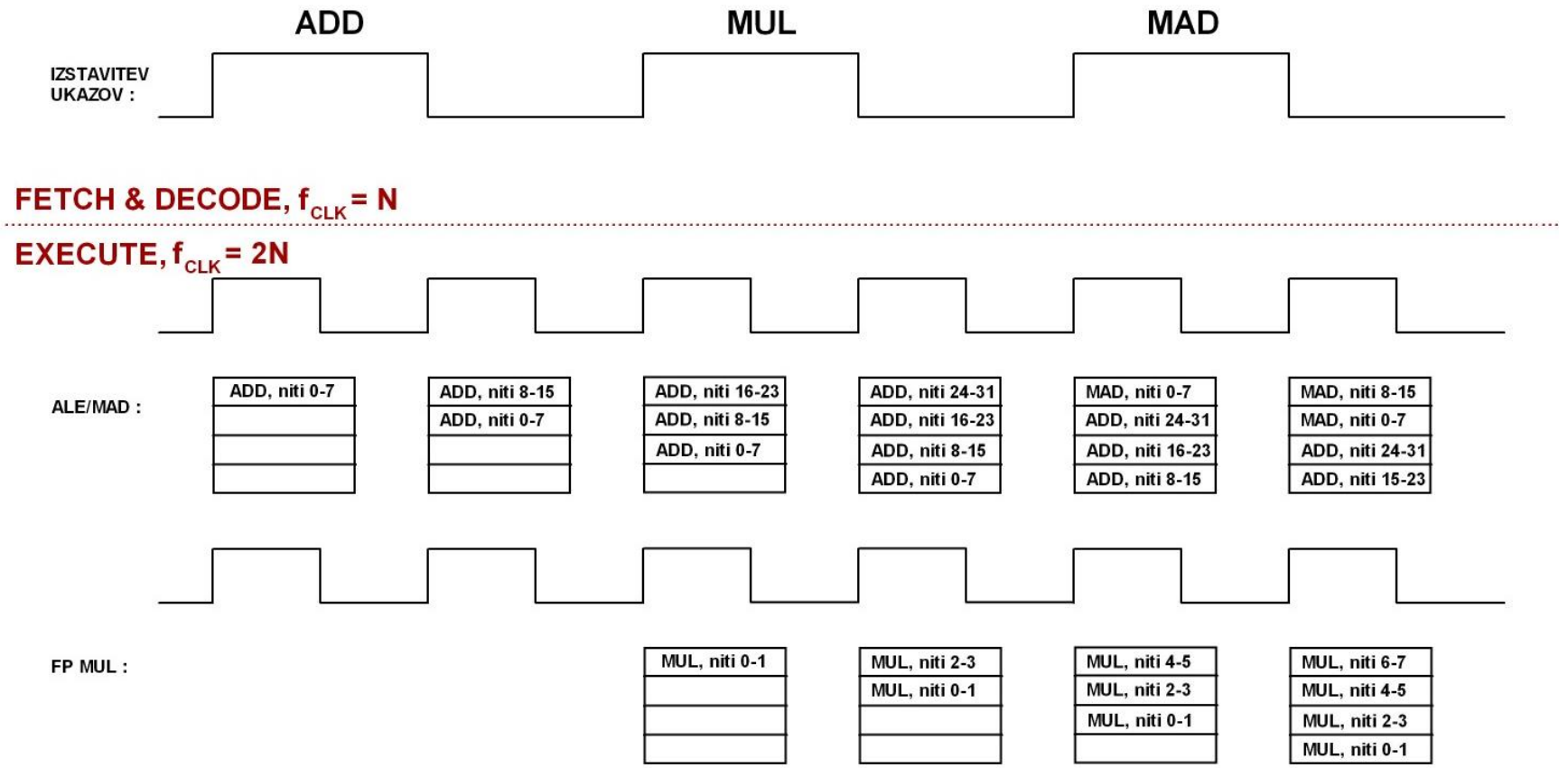
- Vsak SM ima 8 SP, snop ima 32 niti
  - Ena nit se izvaja na enem SP → nit se izvaja v štirih korakih (cevovod)
  - Niti v istem snopu ne potrebujejo sinhronizacije, saj se izvajajo hkrati
- Zakaj potrebujemo toliko snopov v vsakem SM?
  - Zakrivanje latence (dostop do glavnega pomnilnika 500 urinih period)
  - Med čakanjem se lahko izvajajo drugi, pripravljeni snopi
- Z razvrščanje niti ni stroškov
  - Na voljo je mnogo snopov, vedno se najde kakšen pripravljen, SM je tako ves čas izvajanja zaseden
  - Množica snopov je tudi razlog, da predpomnilniki niso potrebni

SM – izvajanje ukazov

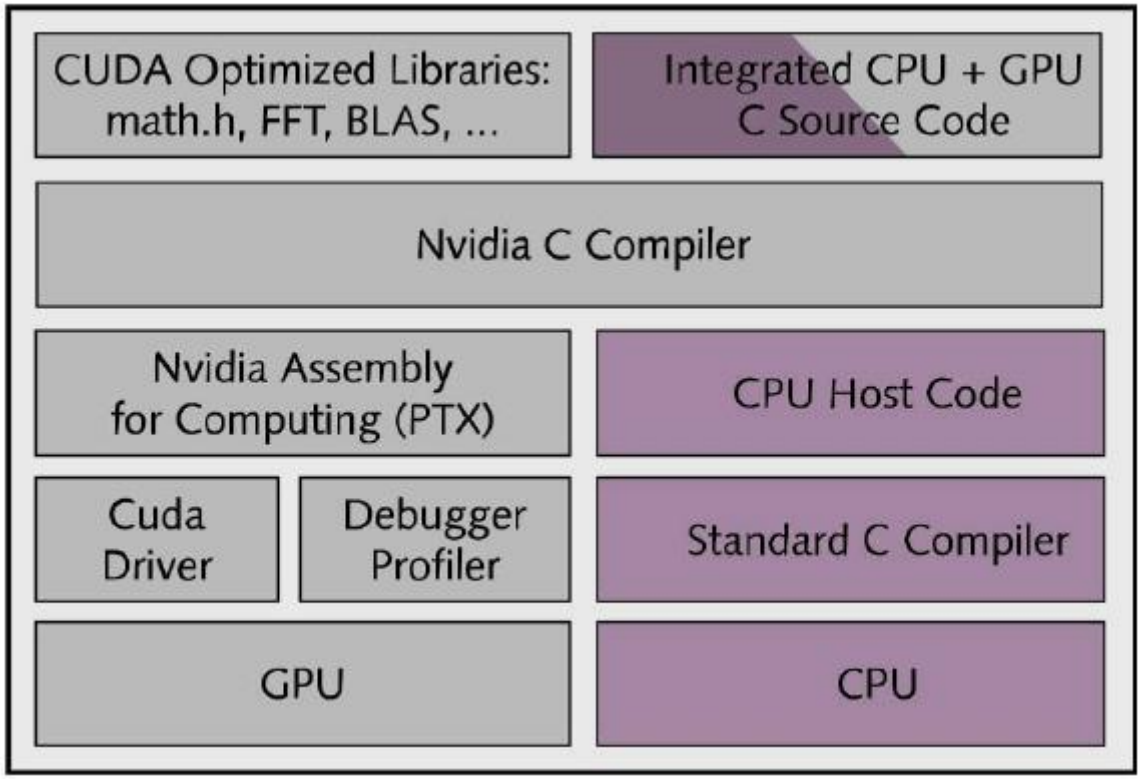
- Kako lahko 8 izvajalnih enot SP hkrati izvaja 32 nitk?
  - latenca ALE/MAD enot je 4 – to pomeni, da se en ukaz izvaja 4 urine periode hitre ure
  - vsako urino periodo vstopi v cevovode 8 nitk iz istega snopa
  - po 4 urinih periodah se cevovodi napolnijo z 32 nitkami iz enega snopa, nato po začetni latenci vsako urino periodo dobimo rezultat za 8 nitk
  - tudi enote za izvajanje skočnih ukazov imajo latenco 4
  - latenca enot SFU je 16-32 ciklov, ukaz za množenje izvedejo v 4 urinih periodah

SM - izvajanje ukazov

- V vsaki urini periodi se lahko v SM izvaja 8 ukazov v enotah ALE/MAD ter še ukazi v enotah SFU
  - Izstavitev ukazov dela s polovično frekvenco

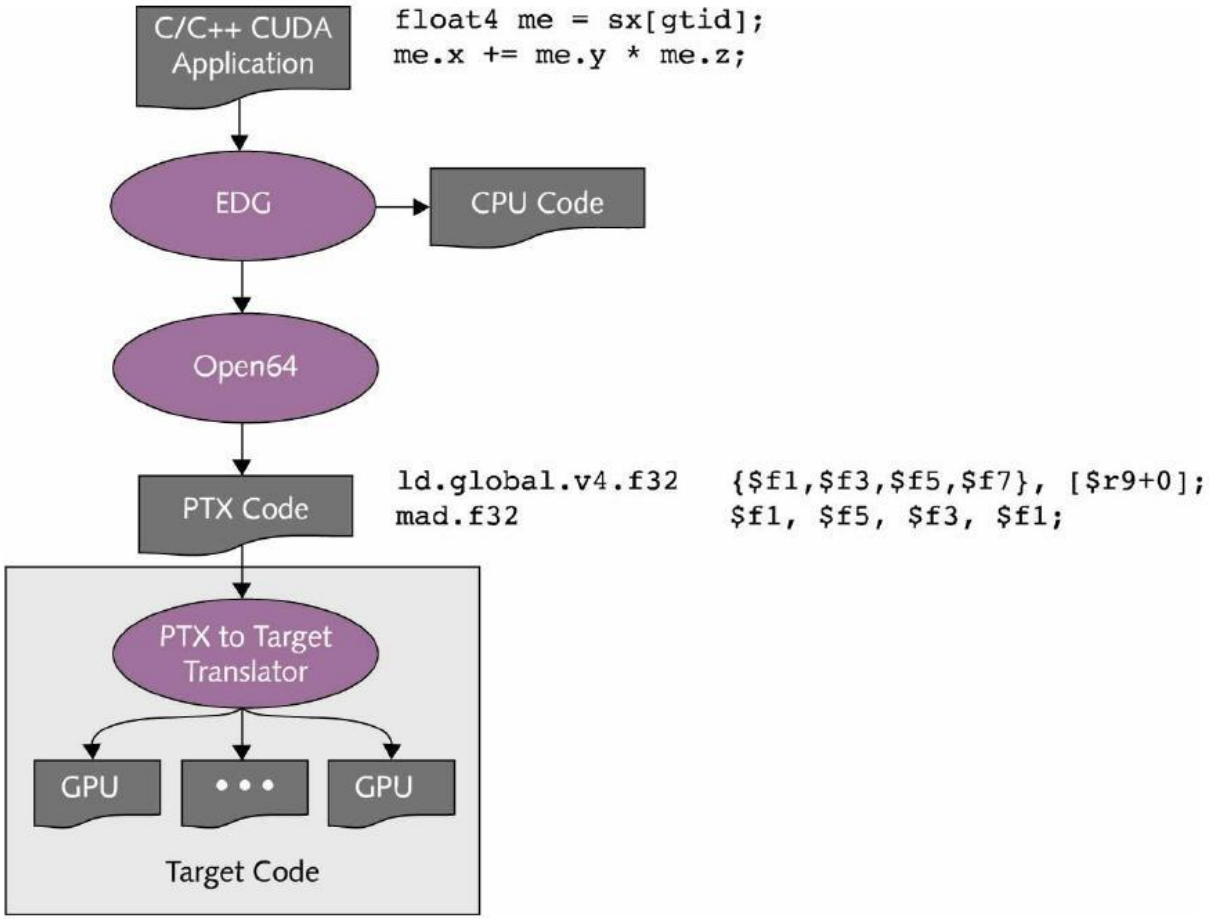


Programski vmesnik CUDA



Proces prevajanja

- programiramo v razširjenem C/C++
  - končnica .cu
  - prevajamo s prevajalnikom nvcc
  - koda za GPU napravo se prevede v vmesno kodo PTX



Pozdravljen svet na CPE

```
#include "common/book.h"

int main(void) {

   printf("Pozdravljen svet! \n");

   return 0;

}
```

- Kje je tu CUDA C?!?!
  - To je vsebina datoteke s končnico .cu – prevedemo jo z NVCC prevajalnikom
  - CUDA C je le razširitev jezika C
- Če kode ne izvajamo na napravi CUDA, potem je to le 'navadni' C
- Zgornja koda se izvede na gostitelju – na glavni CPE in v njenem naslovnem prostoru

Programski vidiki arhitekture CUDA

- Programska koda je sestavljena iz:
  - kode, ki se izvaja na gostitelju (CPE)
  - kode, ki se izvaja na napravi GPE

- Prevajalniku moramo s pomočjo dodatnih oznak sporočiti, kateri del kode (funkcija) se izvaja na gostitelji in kateri na napravi:
  - `__global__` : funkcija se izvaja na napravi in jo kliče gostitelj
  - `__device__` : funkcija se izvaja na napravi in jo lahko kličemo le iz naprave
  - `__host__` : funkcija se izvaja na gostitelju (privzeto)

Pozdravljen svet 2

```
#include "common/book.h"

// deklariramo scepec - scepec je funkcija, ki se izvede na CUDA napravi.
// nas prvi scepec ne pocne nicesar; takoj se vrne
// scepec ne more nicesar vracati - saj se izvaja na drugi napravi in v drugem naslovnem prostoru
// zato vraca argument tipa void.
__global__ void scepec(void) {}

int main(void) {

   // gostitelj poklice scepec na napravi.
   // dejansko za to poskrbita CUDA prevajalnik in runtume
   // v <<< >>> podamo argumente za CUDA runtume - dolocajo v kaksnem okolju naj se izvede scepec
   scepec <<<1,1>>>(); // scepec naj se izvede na polju niti 1x1 (ena sam nit)

   // Ta del kode, pa se izvaja na gostitelju:
   printf("Pozdravljen svet! Pravkar sem izvedel klic scepca na CUDA napravi!\n");

   return 0;

}
```

Ščepec - napoved

```
// deklariramo scepec - scepec je funkcija, ki se izvede na CUDA napravi.
// nas prvi scepec ne pocne nicesar; takoj se vrne
// scepec ne more nicesar vracati - saj se izvaja na drugi napravi in v drugem naslovnem prostoru
// zato vraca argument tipa void.
__global__ void scepec(void) {}
```

- Ščepec je funkcija, ki jo pokliče gostiteljeva CPE in se izvede na napravi CUDA
- Ščepec ima oznako `__global__`, ki pove,
  - da NVCC prevaja kodo za napravo CUDA in
  - da se koda izvede na napravi CUDA
- Ščepec ne more vračati vrednosti, saj se izvaja v popolnoma drugem naslovnem prostoru!!!

Ščepec - klic

```
// gostitelj poklice scepec na napravi.
// dejansko za to poskrbita CUDA prevajalnik in runtume
// v <<< >>> podamo argumente za CUDA runtume - dolocajo v kaksnem okolju naj se izvede scepec
scepec <<<1,1>>>(); // scepec naj se izvede na polju niti 1x1 (ena sam nit)
```

- Ščepec pokliče gostitelj z navedbo izvajalnega okolja v oklepajih <<< >>>
  - v primeru zgoraj zahtevamo, da ta ščepec izvede le ena nit
- Za razvrščanje niti po procesorjih MP in SP skrbi izvajalno okolje
- Takoj po klicu ščepca gostiteljeva CPE nadaljuje z izvajanjem kode na gostitelju! Ni čakanja, da se ščepec konča!

Pomnilnik naprave GPE

- Pomnilnik na napravi GPE lahko rezerviramo dinamično:
  - kot linearni pomnilnik
  - kot CUDA polje (CUDA array ; optimiziran za branje tekstur)
- Linearni pomnilnik:
  - 32-bitni naslovni prostor za naprave 1.0
  - 40-bitni naslovni prostor za naprave 2.0
  - rezerviranje: `cudaMalloc()`
  - sproščanje: `cudaFree()`
  - prenos podatkov: `cudaMemcpy()`

Naj CUDA sešteje dve števili

```
// Nas scepec sedaj sesteje dve celi stevili.
__global__ void sestej(int a, int b, int *c) {
   *c = a + b;
}

int main(void) {

   int h_c;  // rezultat na gostitelju
   int *d_c; // naslov rezultata na napravi - nikoli ne poskusaj dereferencirati ta kazalec!
             // gostitelj ga lahko poljubno uporablja, le brati in pisati na ta naslov ne more!

   // Ker scepec ne more vracati parametrov, moramo na CUDA napravi
   // rezervirati prostor za rezultat (c):
   cudaMalloc( (void **)&d_c,   // kam naj nam funkcija zapise naslov
               sizeof(int)      // kolicina rezerviranega prostora v B
               );

   sestej <<<1,1>>>(7, 11, d_c); // scepec naj se izvede na polju niti 1x1 (ena sam nit)


   // prenesimo rezultat iz naprave na gostitelja:
   cudaMemcpy (&h_c,                          // ponor
               d_c,                           // izvor
               sizeof(int),                   // kolicina podatkov v B
               cudaMemcpyDeviceToHost         // smer kopiranja
               );


   printf("Rezultat sestevanja n CUDA napravi je %d \n", h_c);

   return 0;

}
```

Ščepec z argumenti

```
// Nas scepec sedaj sesteje dve celi stevili.
__global__ void sestej(int a, int b, int *c) {
   *c = a + b;
}
```

- V ščepec lahko prenašamo tudi argumente
  - Praviloma prenašamo naslove

Vračanje vrednosti iz ščepca

```
int h_c;  // rezultat na gostitelju
int *d_c; // naslov rezultata na napravi - nikoli ne poskusaj dereferencirati ta kazalec!
          // gostitelj ga lahko poljubno uporablja, le brati in pisati na ta naslov ne more!
// Ker scepec ne more vracati parametrov, moramo na CUDA napravi
// rezervirati prostor za rezultat (c):
cudaMalloc( (void **)&d_c,   // kam naj nam funkcija zapise naslov
            sizeof(int)      // kolicina rezerviranega prostora v B
            );
```

- Če želimo iz ščepca vračati vrednost, to storimo posredno
  - Ščepec lahko vrne vrednost tako, da jo zapiše v pomnilnik na napravi
  - Ustrezen pomnilniški naslov moramo v ščepec prenesti kot argument
  - Nato iz naprave prenesemo vrnjeno vrednost na gostitelja

Vračanje vrednosti iz ščepca

```
int h_c;  // rezultat na gostitelju
int *d_c; // naslov rezultata na napravi - nikoli ne poskusaj dereferencirati ta kazalec!
          // gostitelj ga lahko poljubno uporablja, le brati in pisati na ta naslov ne more!
// Ker scepec ne more vracati parametrov, moramo na CUDA napravi
// rezervirati prostor za rezultat (c):
cudaMalloc( (void **)&d_c,   // kam naj nam funkcija zapise naslov
            sizeof(int)      // kolicina rezerviranega prostora v B
            );
```

- Na gostitelju in napravi rezerviramo pomnilniški prostor
  - Na gostitelju rezerviramo prostor za rezultat
    - V zgornjem primeru je to `h_c`
  - Na napravi rezerviramo prostor, kamor bomo rezultat najprej shranili in ga potem prenesli na gostitelja
    - V zgornjem bo rezultat na naslovu `*d_c`
    - Uporabimo funkcijo `cudaMalloc()`, ki vzame dva argumenta:
      - Naslov na gostitelju, kamor vpiše začetno lokacijo rezerviranega prostora
        - Naslov moramo prenesti po referenci, da ga lahko funkcija prepiše. Zato "kazalec na kazalec"
        - Naslov mora biti tipa `void`
      - Količino potrebnega prostora, izraženo v bajtih

Klic ščepca z argumenti

```
sestej <<<1,1>>>(7, 11, d_c); // scepec naj se izvede na polju niti 1x1 (ena sam nit)
```

- Ščepec sedaj lahko kličemo z argumenti
  - V ščepec prenesemo števili, ki ju želimo sešteti ter naslov na napravi, kamor naj se zapiše rezultat
  - Spomnimo se, da nam je ta naslov vrnila funkcija `cudaMalloc()`
  - Če ne bi bilo oklepajev <<< >>>, bi imeli klic, kot v navadnem C-ju

Kako dobimo razultat iz naprave GPE?

```
// prenesimo rezultat iz naprave na gostitelja:
cudaMemcpy (&h_c,                          // ponor
            d_c,                           // izvor
            sizeof(int),                   // kolicina podatkov v B
            cudaMemcpyDeviceToHost         // smer kopiranja
            );
```

- Ko se ščepec izvede, bo rezultat na napravi v pomnilniški besedi z naslovom, ki ga hrani `d_c` na gostitelju
  - **To je naslovni prostor na napravi in CPE ne more neposredno dostopati do vsebine!!!**
  - **Podatke na/iz naprave prenašamo s funkcijo `cudaMemcpy()` !**
  - **Nikoli ne poskušaj iz CPE dereferencirati `d_c`!!!**

Kako izvemo kaj o napravi GPE?

```
int main(void) {
   cudaDeviceProp devprop;
   int count; // koliko je posameznih CUDA naprav

   cudaGetDeviceCount(&count);  // preberi stevilo CUDA naprav v sistemu

   // za vsako napravo prebiraj parametre v strukturo devprop:
   for(int i=0; i<count; i++){
      cudaGetDeviceProperties(&devprop, i);

      // Delaj nekaj s pravkar prebranimi parametri (lastnostmi) CUDA naprave...
   }

   return 0;

}
```

- `cudaDeviceCount(cudaDeviceProp* devprop, int n)`:
  - Zapiše lastnosti n-te vgrajene CUDA naprave v strukturo `devprop`
  - V sistemu imamo lahko več CUDA naprav – številčijo se kot 0, 1, 2, ...

Lastnosti naprave GPE

```
int main(void) {
   cudaDeviceProp devprop;
   int count; // stevilo CUDA naprav v sistemu

   cudaGetDeviceCount(&count);  // preberi stevilo CUDA naprav

   // za vsako napravo prebiraj posamezne parametre v strukturo devprop:
   for(int i=0; i<count; i++){
      cudaGetDeviceProperties(&devprop, i);

      printf("    ------- Info o CUDA napravi %d ------\n", i);
      printf("Ime naprave: %s \n", devprop.name);
      printf("Hitrost ure: %d \n", devprop.clockRate);
      printf("Kolicina globalnega pomnilnika: %d \n", devprop.totalGlobalMem);
      printf("Kolicina pomnilnika konstant: %d \n", devprop.totalConstMem);
      printf("Stevilo multiprocesorjev SM: %d \n", devprop.multiProcessorCount);
      printf("Kolicina skupnega pomnilnika v SM: %d \n", devprop.sharedMemPerBlock);
      printf("Stevilo registrov v SM: %d \n", devprop.regsPerBlock);
   }

   return 0;

}
```