Porazdeljeni sistemi

7. Računanje na grafičnih procesnih enotah II

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

Arhitektura CUDA – ponovitev

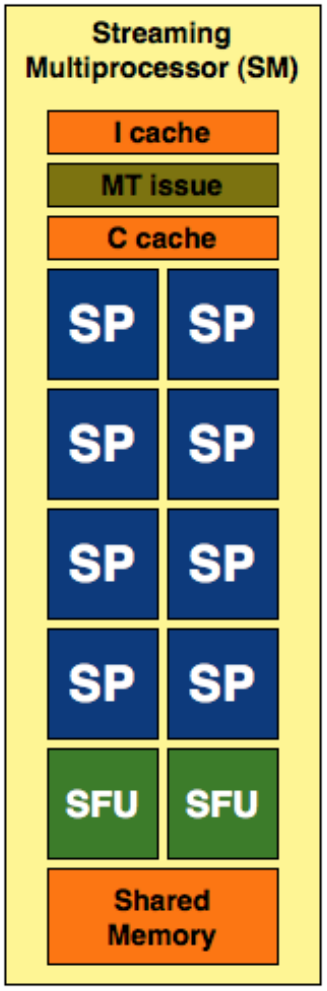
- Trg grafičnih procesorjev zahteva vedno hitrejše in močnejše GPE
- GPE so visoko paralelne naprave, danes namenjene tudi splošnemu računanju
- Uporabljajo model več-nitnosti z deljenim pomnilnikom, ki je precej prilagojen zahtevam grafičnih aplikacij
- Za razliko od CPE, kjer se hkrati lahko izvaja od 2 do 8 niti, se na GPE hkrati izvaja na tisoče niti

Arhitektura CUDA – ponovitev

- nVidia CUDA
  - Razvojno orodje za programiranje naprav CUDA
- Program CUDA je sestavljen iz
  - Kode, ki teče na gostitelju (CPE)
  - Ščepca (jedra), ki teče na napravi (GPE)
  - Niti so organizirane v bloke, ki sestavljajo mrežo.
  - Vsak ščepec naenkrat izvaja le eno mrežo niti

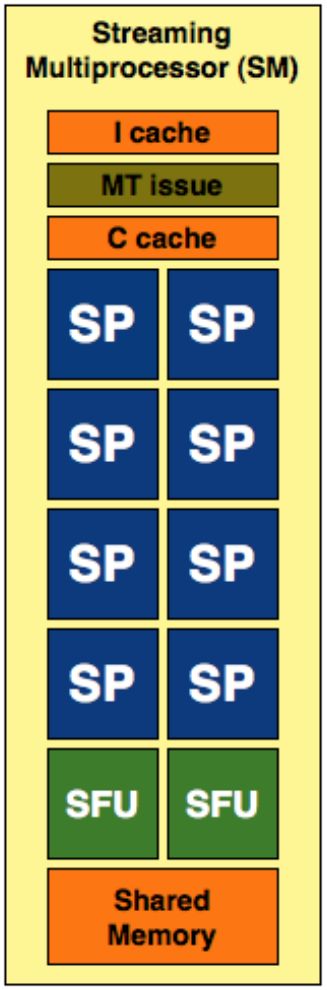
Arhitektura CUDA – ponovitev

- Moč naprav CUDA lahko izkoristimo le, če se dobro zavedamo njihove arhitekture
  - Procesorji SM (Streaming Multiprocessors)
    - Sestavljeni so iz izvajalnih enot SP (Streaming Processors)
    - Skupnega (deljenega) pomnilnika
    - Predpomnilnikov
  - Posamezen blok niti se v celoti izvaja na enem procesorju SM
  - En procesor SM lahko na enkrat izvaja več blokov niti



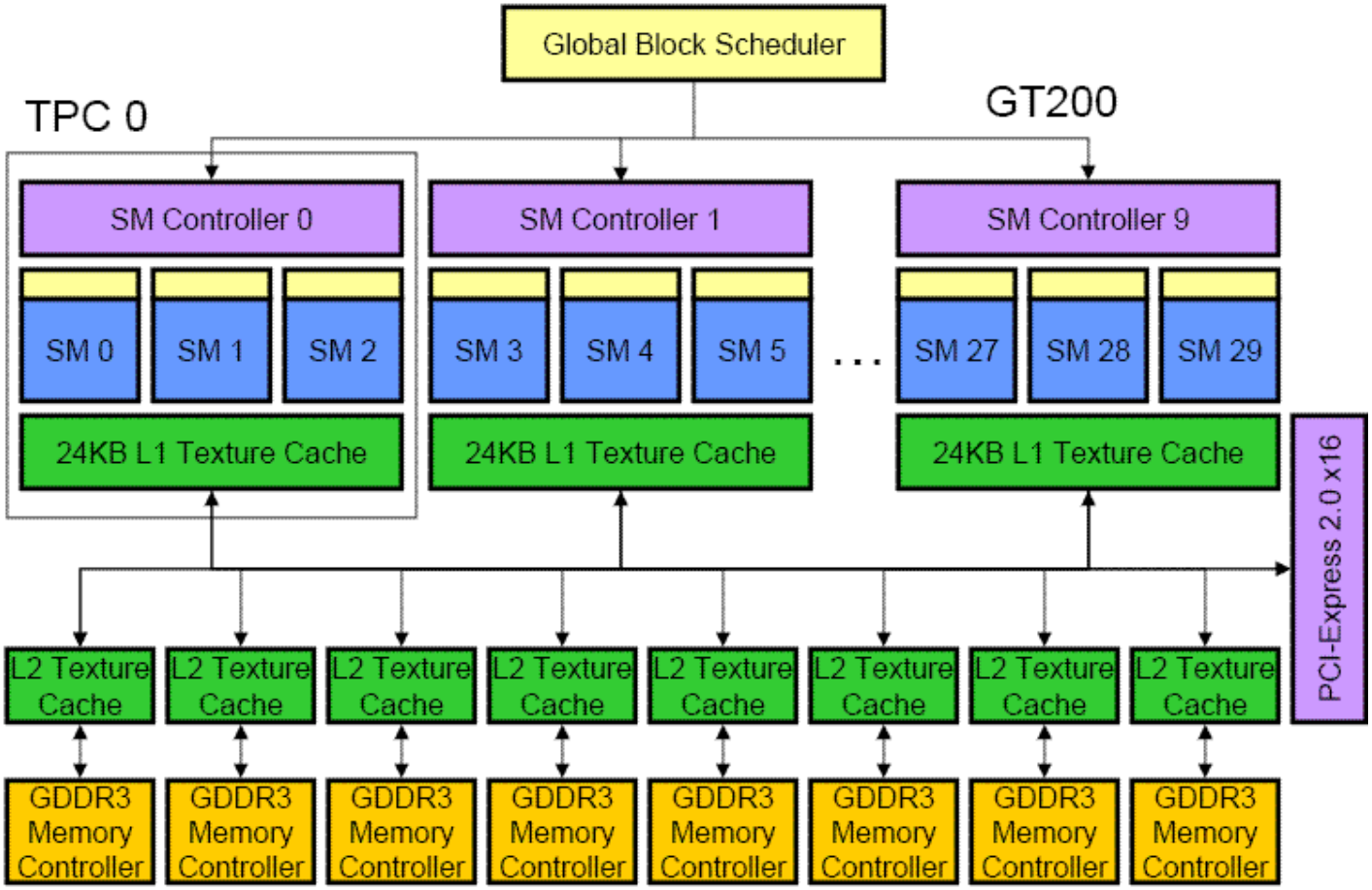
Arhitektura CUDA – ponovitev

- Moč naprav CUDA lahko izkoristimo le, če se dobro zavedamo njihove arhitekture
  - Sinhronizacija niti
    - Niti iz istega bloka se lahko sinhronizirajo in si učinkovito izmenjujejo podatke
    - Niti, ki se izvajajo na različnih procesorjih SM, ne moremo sinhronizirati!!!
    - Ker niti v različnih SM niso medsebojno odvisne, je potrebne veliko manj arbitraže, zaradi česar je izvajanje lahko hitrejše



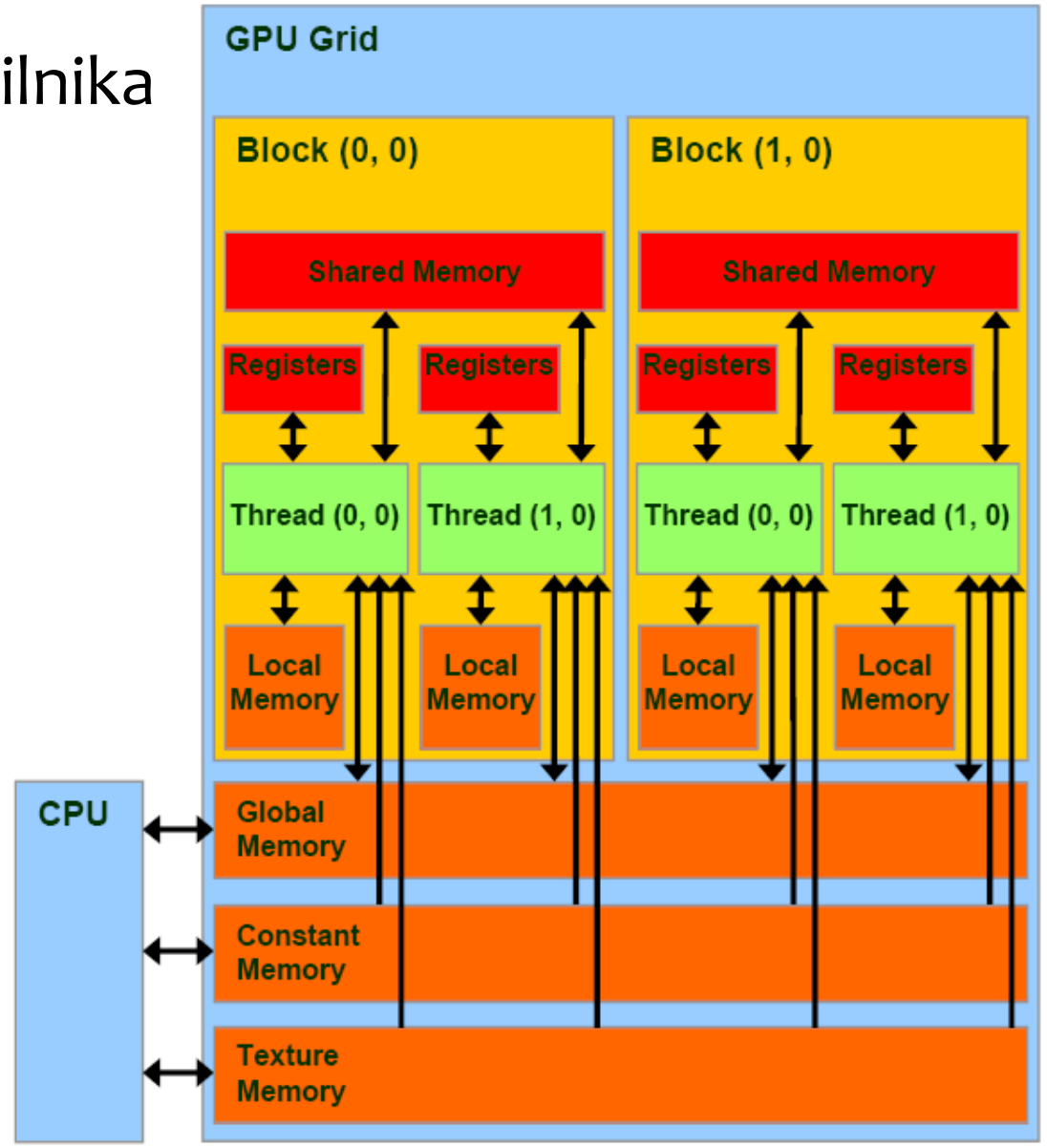
Arhitektura CUDA – ponovitev

- Moč naprav CUDA lahko izkoristimo le, če se dobro zavedamo njihove arhitekture



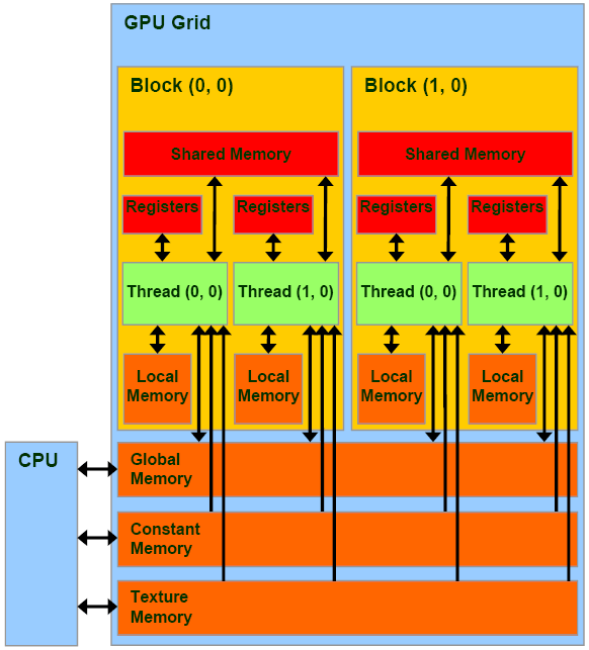
Arhitektura CUDA – ponovitev

- Organizacija pomnilnika



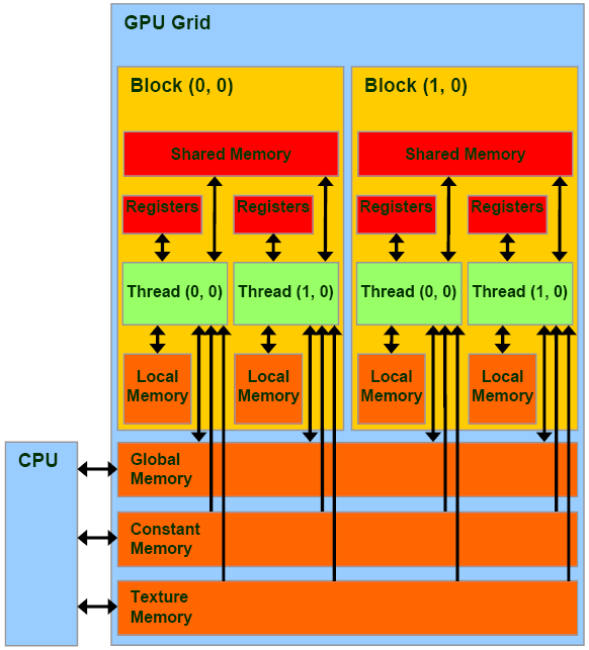
Arhitektura CUDA – ponovitev

- Organizacija pomnilnika
  - Skupni (deljeni) pomnilnik
    - Delijo si ga vse niti v procesorju SM
    - Je hiter, vendar ga je malo
    - Vsebina se ob naslednjem klicu ščepca izgubi
  - Globalni pomnilnik
    - Delijo si ga vse niti v mreži (tudi, če so v različnih blokih)
    - Je počasen, je mnogo večji kot deljeni pomnilnik
    - Vsebina se ob naslednjem klicu ščepca ne izgubi
    - Dosegljiv je tudi iz gostiteljeve CPE
  - Nobeden od omenjenih pomnilnikov ni predpomnjen!!!



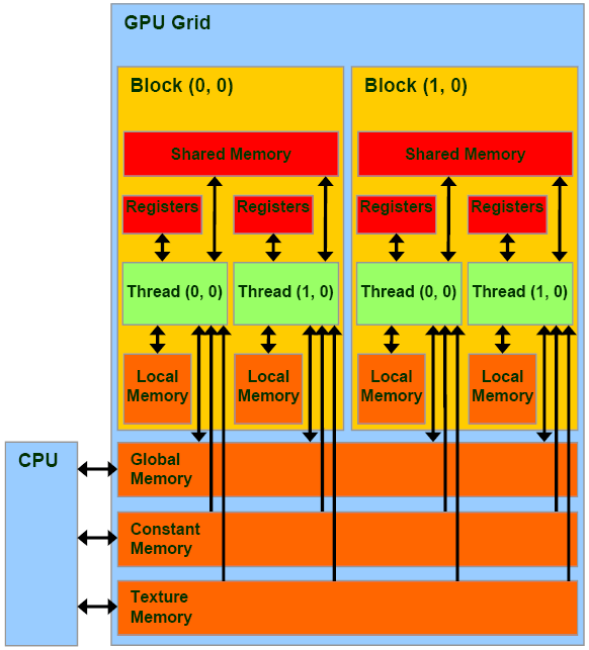
Arhitektura CUDA – ponovitev

- Organizacija pomnilnika
  - Registri
    - Vsak procesor SM ima svoj nabor registrov
    - Dostop do njih je hitrejši kot do skupnega (deljenega) pomnilnika
    - Omejeno število na izvajalno enoto SP
  - Lokalni pomnilnik
    - Lokalni pomnilnik za vsako nit
    - Fizično je del globalnega pomnilnika, zato so dostopni časi zelo dolgi
    - Prevajalnik ga uporablja samo v primerih, ko zmanjka registrov!!



Arhitektura CUDA – ponovitev

- Organizacija pomnilnika
  - Pomnilnik konstant
    - Niti lahko iz njega samo berejo
    - Iz njega lahko bere in vanj vpisuje gostitelj
    - Je del globalnega pomnilnika
  - Pomnilnik tekstur
    - Niti lahko iz njega samo berejo
    - Iz njega lahko bere in vanj vpisuje gostitelj
    - Je del globalnega pomnilnika
    - Je predpomnjen na SM



Programiranje CUDA – ponovitev

- Jezik CUDA C je precej podoben jeziku C.

  Nekaj najpomembnejših razlik:

  - Ne more uporabljati standardnih knjižnic in funkcij, na primer printf
  - Ni rekurzije
  - Ni sklada
  - Ni kazalcev na funkcije

Organizacija niti

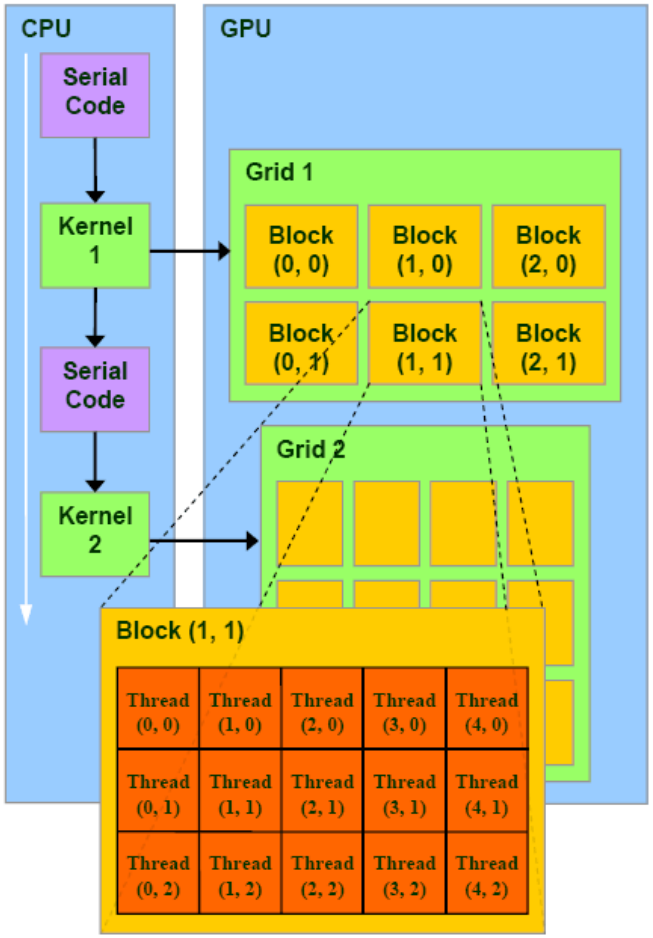
- Organizacija niti na napravi CUDA – ponovitev:
  - Niti so organizirane v bloke
    - Niti znotraj istega bloka se identificirajo z
      - eno-dimenzionalnim indeksom,
      - dvo-dimenzionalnim indeksom ali
      - tri-dimenzionalnim indeksom.
    - Niti v istem bloku se izvajajo na istem SM in lahko med seboj komunicirajo preko skupnega pomnilnika

Organizacija niti

- Organizacija niti na napravi CUDA – ponovitev:
  - Bloki so organizirani v mrežo
    - Bloke lahko označujemo z
      - eno-dimenzionalnim indeksom ali
      - dvo-dimenzionalnim indeksom.
    - Bloki se izvajajo v poljubnem vrstnem redu
    - Niti v različnih blokih lahko komunicirajo preko glavnega pomnilnika
    - Programer **NE VE** kje in kdaj se izvaja posamezen blok!!!
  - Vsakemu ščepcu ustreza natanko ena mreža

Organizacija niti

- Niti je vedno smiselno organizirati v podobno strukturo kot podatke
  - Če obdelujemo podatke v dvodimenzionalnih tabelah, potem je smiselno niti organizirati v dvodimenzionalno mrežo
  - Niti v mreži združujemo v bloke. Način združevanja je odvisen od želenega načina medsebojne komunikacije



Organizacija niti

- Programer se sam odloči, koliko niti bo uporabil za reševanje določenega problema – programer sam definira izvajalno okolje
- Dobro je, da imamo niti čim več, saj tako
  - olajšamo delo razvrščevalniku, ki ima za izvajanje vedno na voljo več snopov niti
  - pohitrimo izvajanje programa
- Omejitve:
  - število niti v bloku
  - število blokov v mreži
  - Število niti na multiprocesorju

Organizacija niti

- Izvajalno okolje podamo kot

```
<<< dimenzija mreže, dimenzija blokov >>>
```

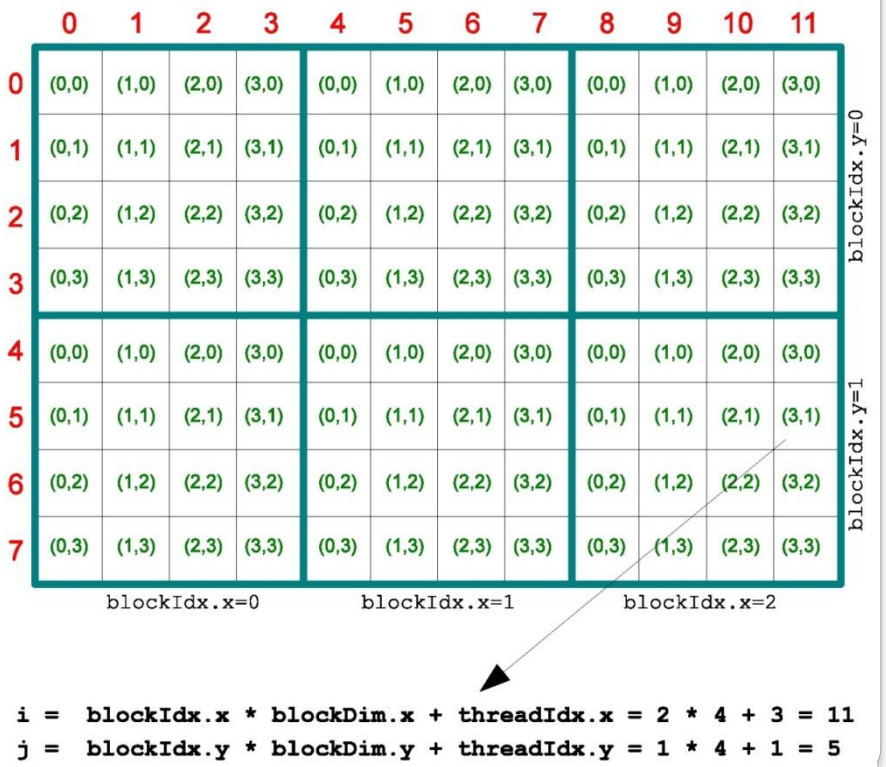
- Določanje izvajalnega okolja:
  - ```
    dim3 dimGrid(x,y)   // nove kartice tudi (x,y,z)
    dim3 dimBlock(x,y,z)
    ```
- Zgled: določitev izvajalnega okolja in klic ščepca
  - ```
    dim3 dimGrid(512,1)
    dim3 dimBlock(128,1,1)
    ```
  - ```
    scepec<<<dimGrid,dimBlock>>>(...);
    ```
    ```

Organizacija niti

- Označevanje niti
  - Število blokov v mreži nam pove
    - `gridDim.x, gridDim.y`
  - Število niti v vsakem bloku izvemo iz:
    - `blockDim.x, blockDim.y, blockDim.z`
  - Vsak blok v mreži ima svojo oznako:
    - `blockIdx.x, blockIdx.y`
  - Vsaka nit v bloku ima svojo oznako:
    - `threadIdx.x, threadIdx.y, threadIdx.z`
  - `gridDim, blockDim, blockIdx` in `threadIdx` so spremenljivke, vgrajene v izvajalno okolje

Organizacija niti

- Primer: dvo-dimenzionalna tabela
  - Izvajalno okolje in klic ščepca
    - `dim3 dimGrid(3,2)`
      `dim3 dimBlock(4,4)`
      `racunaj<<<dimGrid,dimBlock>>>(...);`
  - Položaj niti v mreži
    - `i = blockIdx.x*blockDim.x+ threadIdx.x`
    - `j = blockIdx.y*blockDim.y+ threadIdx.y`



Seštevanje vektorjev na napravi CUDA

- Najprej rezervirajmo prostor na gostitelju in napravi, ter vektorja a in b prenesimo na napravo. To že znamo:

```
int main(void) {

   float *h_a, *h_b, *h_c;
   float *d_a, *d_b, *d_c;

   // rezerviraj prostor v pomnilniku gostitelja za vse tri vektorje:
   h_a = (float*) malloc( N*sizeof(float) );
   h_b = (float*) malloc( N*sizeof(float) );
   h_c = (float*) malloc( N*sizeof(float) );

   // rezerviraj prostor v pomnilniku naprave za vse tri vektorje:
   cudaMalloc( (void**)&d_a, N * sizeof(float) );
   cudaMalloc( (void**)&d_b, N * sizeof(float) );
   cudaMalloc( (void**)&d_c, N * sizeof(float) );

   // inicializiraj vektorja
   for (int i=0; i<N; i++) {
      h_a[i] = (float)(-i);
      h_b[i] = (float)(i * i);
   }

   // kopiraj vektorja a in b iz gostitelja na napravo:
   cudaMemcpy( d_a, h_a, N * sizeof(float), cudaMemcpyHostToDevice );
   cudaMemcpy( d_b, h_b, N * sizeof(float), cudaMemcpyHostToDevice );
```

Seštevanje vektorjev na CUDA napravi

- Definirajmo izvajalno okolje
  - Prilagodimo ga strukturi podatkov.
  - Seštevanje naj opravi N niti, pri čemer je N dolžina vektorjev.
  - Niti naj bodo organizirane v 1 samem bloku:

```
// klici scepec za sestevanje:
vectorAddGPU<<<1,N>>>( d_a, d_b, d_c );
```

- Pripravimo ščepec
  - Vsaka nit opravi samo eno seštevanje.
  - S stavkom if onemogočimo delo nitim, katerih ID >= N – razlog bomo spoznali v nadaljevanju

```
__global__ void vectorAddGPU( float *a, float *b, float *c ) {
   int tid = threadIdx.x; // vsaka nit sesteje svoj element v vektorjih

   if (tid < N) {
      c[tid] = a[tid] + b[tid];
   }
}
```

Seštevanje vektorjev na CUDA napravi

- Na koncu prenesimo vsoto vektorjev nazaj na gostitelja ter sprostimo vse pomnilnike:

```
    // kopiraj rezultatni vektor iz naprave v gostitelja:
    cudaMemcpy( h_c, d_c, N * sizeof(float), cudaMemcpyDeviceToHost );

    // izpisi rezultate
    for (int i=0; i<N; i++) {
       printf( "%f + %f = %f\n", h_a[i], h_b[i], h_c[i] );
    }

    //sprosti pomnilnik na napravi:
    cudaFree( d_a );
    cudaFree( d_b );
    cudaFree( d_c );

    // sprosti pomnilnik v gostitelju:
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;

}
```

Seštevanje vektorjev na napravi CUDA - 2

- Kaj je narobe s prejšnjim programom?
  - Izvaja ga *N* niti, pri čemer so vse niti v istem bloku
  - Ker je niti v bloku največ 1024, s takim programom ne moremo seštevati vektorjev daljših od 1024!

Seštevanje vektorjev na napravi CUDA - 2

- Definirajmo drugačno izvajalno okolje :
  - Vzemimo, da ima vsak blok 256 niti
  - Če želimo seštevati vektorje dolžine *N*, potrebujemo najmanj $\lceil N/256 \rceil$ blokov
    - Rezultat deljenja N/256 ni vedno celoštevilčen, zato zaokrožimo navzgor
    - V jeziku C zaokrožitev napišemo kot (N+255)/256

```
// nastavi okolje za runtime:
int threadsPerBlock = 256; // imamo 128 niti v bloku
int blocksPerGrid = (N+255)/256;    // ce N ni deljiv z N, potem bomo v zadnjem bloku imeli nekaj
                                    //  vec niti. Zato v scepcu preverjam, ce je tid < N
// klici scepec za sestevanje:
vectorAddGPU<<<blocksPerGrid,threadsPerBlock>>>( d_a, d_b, d_c );
```

  - Kadar rezultat deljenja ni celoštevilčen, je v zadnjem bloku manj niti. Ker nimamo posebnega klica za zadnji ščepec, moramo paziti, da kodo ščepca napišemo dovolj splošno!

Seštevanje vektorjev na CUDA napravi - 2

- Sedaj pokličemo ščepec
  - Upoštevamo, da imamo podatke v več blokih
  - Za vsako nit izračunamo njen absolutni položaj v mreži

```
__global__ void vectorAddGPU( float *a, float *b, float *c ) {
   int tid = blockIdx.x * blockDim.x + threadIdx.x; // vsaka nit sesteje svoj element v vektorjih

   if (tid < N) {
      c[tid] = a[tid] + b[tid];
   }
}
```

  - Pri zaokroževanju števila blokov navzgor imamo v zadnjem bloku manj niti kot v vseh ostalih
    - Pri računanju zato ne smemo upoštevati niti, ki seštevajo polja zunaj vektorja
    - Za te niti velja tid >= N
    - To v ščepcu spet preprečimo s stavkom if(tid < N)

Seštevanje vektorjev na CUDA napravi - 3

- Spomnimo se, da je največje število niti v bloku 512 in je blokov v mreži največ 65535
- S tako napisanim programom torej lahko seštevamo največ 33,553,920 elementov vektorjev

- Mogoče se nam to zdi veliko, vendar v resnih znanstvenih aplikacijah ni!

Seštevanje vektorjev na CUDA napravi - 3

- Definirajmo drugačno izvajalno okolje ter malo drugače zapišimo ščepec
- Omejimo izvajalno okolje na 256 niti v bloku in na 256 blokov v mreži – dovolj, da dobro zaposlimo vsa jedra

```
// nastavi okolje za runtime:
int threadsPerBlock = 256; // imamo 256 niti v bloku
int blocksPerGrid = 256;   // imamo 256 blokov v mrezi
// klici scepec za sestevanje:
vectorAddGPU<<<blocksPerGrid,threadsPerBlock>>>( d_a, d_b, d_c );
```

- Naš program sedaj izvaja 256 x 256 = 65536 niti
- V primeru vektorjev z več kot 65536 elementi bo morala posamezna nit sešteti več elementov polja

Seštevanje vektorjev na CUDA napravi - 3

- Napišemo nov ščepec, uporabimo idejo iz sekvenčne rešitve

```
__global__ void vectorAddGPU( float *a, float *b, float *c ) {
   int tid = blockIdx.x * blockDim.x + threadIdx.x; // vsaka nit sesteje svoj element v vektorjih
                                                    // vsaka nit starta na drugi poziciji

   while (tid < N) {
      c[tid] = a[tid] + b[tid];

      tid = tid + blockDim.x * gridDim.x; // sedaj se premaknemo na naslednji kos vektorja
   }
}
```

  - ob klicu ščepca se nit postavi na elemente vektorjev, katerih indeks je enak *tid*
  - nato v zanki ponavlja
    - seštevanje ustreznih elementov na lokaciji *tid* in
    - premik na lokacijo, ki je za število niti (grdiDim.x = 65536) oddaljena
  - Postopek je zaključen, ko indeks *tid* preseže število elementov *N*

Komunikacija med nitmi v bloku

- Vsakemu procesorju SM pripada 16 kB (nove 32 in 48 kB) hitrega skupnega pomnilnika (shared memory)
- En blok niti se izvaja na enem procesorju SM
  - Niti v istem bloku torej lahko komunicirajo preko spremenljivk v skupnem pomnilniku
  - Za to imajo na voljo 16 kB skupnega pomnilnika
  - Vse niti lahko pišejo v spremenljivke in berejo iz spremenljivk v skupnem pomnilniku

- Spremenljivke, ki jih želimo hraniti v skupnem pomnilniku morajo imeti oznako `__shared__`
  - Zgled:

```
__shared__ skupniblokpodatkov[N];
```

Skalarni produkt

- Pri skalarnem produktu moramo najprej zmnožiti vse istoležne elemente v vektorju, nato pa zmnožke sešteti
  - Množenje istoležnih elementov lahko izvedemo na enak način, kot smo to spoznali pri seštevanju vektorjev:
    - Vsaka nit zmnoži po en element vektorja
    - Če je elementov veliko, lahko tudi več
  - Kako izvesti seštevanje?
    - Lahko na CPE, vendar jo je škoda obremenjevati za take stvari.
    - Lahko pa na GPE. Kako?

Skalarni produkt - ščepec

```
__global__ void dotProductGPU( float *a, float *b, float *c ) {
   int tid = blockIdx.x * blockDim.x + threadIdx.x; // vsaka nit sesteje svoj element v vektorjih
                                                    // vsaka nit starta na drugi poziciji
   float temp;

   __shared__ float delniProduktiVBloku[threadsPerBlock]; //vsak blok niti bo hranil delne produkte
                                                          // ki jih izracunajo posamezne niti
                                                          // v bloku hranimo v hitrem skupnem
                                                          // pomnilniku
   //Vsaka nit najprej izracuna svoj delni produkt:
   temp = 0.0;
   while (tid < N) {
      temp += a[tid] * b[tid];
      tid = tid + blockDim.x * gridDim.x; // sedaj se premaknemo na naslednji kos vektorja
   }

   // nato vsaka nit vpise svoj delni produkt v skupni pomnilnik
   // v mesto, ki samo njej pripada:
   delniProduktiVBloku[threadIdx.x] = temp;

   // pocakaj da vse niti vpisejo svoje delne produkte:
   __syncthreads();

   // sedaj morajo niti sesteti delne produkte. To bi lahko pocela ena sama,
   // vendar pri 256 delnih produktih bi potrebovala 128 iteracij, ker bi v eni
   // iteraciji sestela 2 delna produkta.
   // Zato naj vsaka nit sesetje 2 sosednja delna produkta delna produkta - redukcijsko:
   int i = blockDim.x / 2;
   while ( i !=0 ) {
      if (threadIdx.x < i) {
         delniProduktiVBloku[threadIdx.x] += delniProduktiVBloku[threadIdx.x + i];
      }
      // pocakaj da vse koncajo, predno spremenis i:
      __syncthreads();
      i = i/2;
   }

   // sedaj imamo v prvem elementu polja delniProduktiVBloku delni skalarni produkt vsakega bloka.
   // naj ga nit 0 prepise v globalni pomnilnik:
   if (threadIdx.x == 0)
      c[blockIdx.x] = delniProduktiVBloku[0];
}
```

Skalarni produkt – razlaga ščepca

- Če želimo zmnožke sešteti na napravi GPE, mora posamezna nit znati prebrati zmnožke, ki so jih izračunale druge niti → niti morajo med seboj komunicirati
- Za medsebojno komunikacijo uporabimo skupni pomnilnik
  - Niti v istem bloku lahko dostopajo do podatkov v skupnem pomnilniku
- IDEJA:
  - Naj vsaka nit računa in sešteva svoje delne produkte. Ko zaključi, naj svoj produkt vpiše v en element polja `delniProduktiVBloku`:

```
__shared__ float delniProduktiVBloku[threadsPerBlock]; //vsak blok niti bo hranil delne produkte
                                                       // ki jih izracunajo posamezne niti
                                                       // v bloku hranimo v hitrem skupnem
                                                       // pomnilniku
```

  - To polje mora imeti toliko elementov, kolikor je niti v bloku
  - To polje se mora nahajati v skupnem pomnilniku na procesorju SM, da bodo lahko vse niti v bloku dostopale do njega

Skalarni produkt – razlaga ščepca

- Vsaka nit v spremenljivki temp računa svoj delni skalarni produkt
  - Če je elementov v vektorju manj kot (blockDim.x * gridDim.x), potem vsaka nit opravi le eno množenje.
  - Sicer pa se premakne na naslednji kos vektorja (tako kot smo to spoznali pri seštevanju vektorjev) in računa novi produkt, ki ga prišteje k svojemu delnemu skalarnemu

```
//Vsaka nit najprej izracuna svoj delni produkt:
temp = 0.0;
while (tid < N) {
   temp += a[tid] * b[tid];
   tid = tid + blockDim.x * gridDim.x; // sedaj se premaknemo na naslednji kos vektorja
}
```

Skalarni produkt – razlaga ščepca

- Ko vse niti zaključijo z računanjem svojih delnih produktov, jih shranijo v skupni pomnilnik
  - Ko bomo imeli vse delne produkte zapisane v skupnem pomnilniku, jih bomo lahko začeli seštevati
  - Niti *i* v nekem bloku pripada v skupnem pomnilniku *i*-ti element polja delniProduktiVBloku - tu shrani svoj delni skalarni produkt, ki ga je računala v spremenljivki *temp*

```
// nato vsaka nit vpise svoj delni produkt v skupni pomnilnik
// v mesto, ki samo njej pripada:
delniProduktiVBloku[threadIdx.x] = temp;
```

Skalarni produkt – razlaga ščepca

- Sinhronizacija
  - Niti se izvajajo neodvisno ena od druge, zato ne moremo vedeti v kakšnem vrstnem redu se bodo delni skalarni produkti vpisovali v skupni pomnilnik
  - Delne skalarne produkte lahko seštejemo šele potem, ko vse niti zaključijo računanje
  - To dosežemo s pregrado

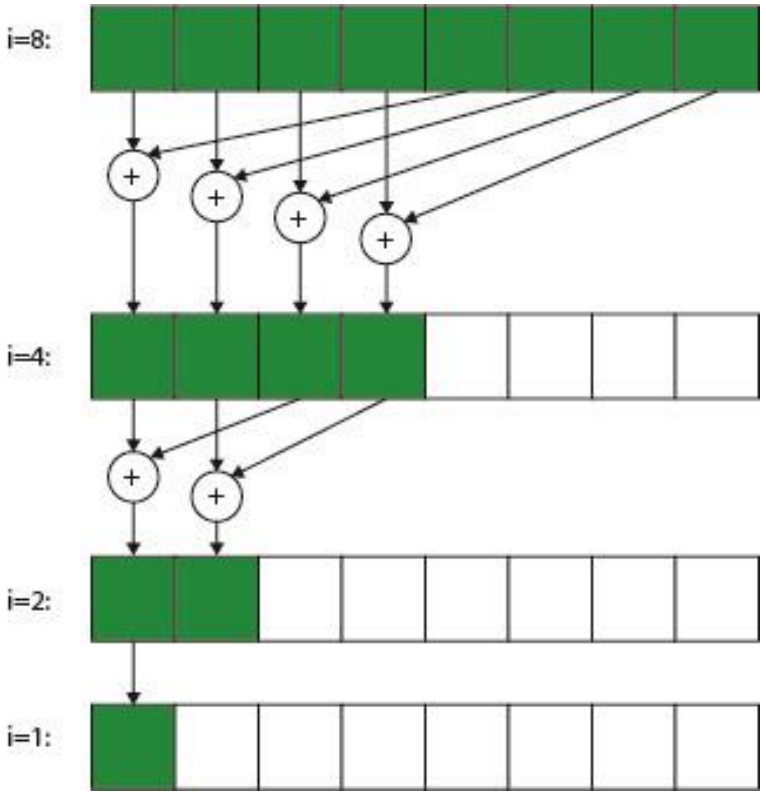
    `__synchthreads()`
    - Sinhronizira vse niti v bloku
    - Ustavi izvajanje vseh niti dokler vse ne pridejo do pregrade

```
// pocakaj da vse niti vpisejo svoje delne produkte:
__syncthreads();
```

Skalarni produkt – razlaga ščepca

- Seštevanje
  - To bi lahko počela ena sama nit, ostale niti bi čakale
  - Se da bolje?

- Redukcija
  - Kaj, če vsaka nit sešteje svoj in še en delni rezultat?
  - V tem primeru za seštevanje *N* delnih produktov potrebujemo $\log_2(N)$ korakov



Skalarni produkt - redukcija

- Redukcija
  - Po vsaki iteraciji se nam število produktov, ki jih moramo sešteti, razpolovi
  - V vsaki iteraciji se razpolovi število niti, ki sešteva delne produkte
  - Po vsaki iteraciji potrebujemo pregrado!

```
// sedaj morajo niti sesteti delne produkte. To bi lahko pocela ena sama,
// vendar pri 256 delnih produktih bi potrebovala 128 iteracij, ker bi v eni
// iteraciji sestela 2 delna produkta.
// Zato naj vsaka nit sesetje 2 sosednja delna produkta delna produkta - redukcijsko:
int i = blockDim.x / 2;
while ( i !=0 ) {
   if (threadIdx.x < i) {
      delniProduktiVBloku[threadIdx.x] += delniProduktiVBloku[threadIdx.x + i];
   }
   // pocakaj da vse koncajo, predno spremenis i:
   __syncthreads();
   i = i/2;
}
```

Skalarni produkt – nevarnosti pri redukciji

- Redukcija
  - Ali bi lahko pred pregrado počakale samo tiste niti, ki v posamezni iteraciji seštevajo delne produkte?
  - Ne bo šlo, saj `__syncthreads` vstavi pregrado za vse niti!
    - Niti, ki ne seštevajo, nikoli ne bi prišle do pregrade, zato bi se izvajanje na napravi CUDA ustavilo brez sporočila o napaki
    - Niti, ki seštevajo bi prišle do pregrade in tam čakale ostale niti!!!

```
// sedaj morajo niti sesteti delne produkte. To bi lahko pocela ena sama,
// vendar pri 256 delnih produktih bi potrebovala 128 iteracij, ker bi v eni
// iteraciji sestela 2 delna produkta.
// Zato naj vsaka nit sesetje 2 sosednja delna produkta delna produkta - redukcijsko:
int i = blockDim.x / 2;
while ( i !=0 ) {
   if (threadIdx.x < i) {
      delniProduktiVBloku[threadIdx.x] += delniProduktiVBloku[threadIdx.x + i];
      // pocakaj da vse koncajo, predno spremenis i:
      __syncthreads();
   }

   i = i/2;
}
```

Skalarni produkt – redukcija II

- Sedaj imamo v vsakem bloku (na vsakem procesorju SM) izračunan delni skalarni produkt za vse niti v bloku
  - Delni skalarni produkt za vsak blok je izračunan v `delniProduktiVBloku[0]`
- Kako jih sedaj med seboj sešteti, da dobimo končni skalarni produkt?

Skalarni produkt – redukcija II

- Kako niti med seboj sešteti, da dobimo končni skalarni produkt?
  - Niti iz različnih blokov med seboj ne morejo komunicirati preko skupnega pomnilnika
  - Komunikacija med nitmi iz različnih blokov je možna le preko globalnega pomnilnika na napravi
  - Zato moramo delne skalarne produkte iz vsakega bloka prepisati v globalni pomnilnik v polje c:

```
// sedaj imamo v prvem elementu polja delniProduktiVBloku delni skalarni produkt vsakega bloka.
// naj ga nit 0 prepise v globalni pomnilnik:
if (threadIdx.x == 0)
   c[blockIdx.x] = delniProduktiVBloku[0];
```

Skalarni produkt – redukcija II

- Kako nastavimo ustrezno izvajalno okolje?
  - Izberemo fiksno število niti v posameznem bloku, na primer 128
  - Nato določimo koliko je maksimalno možnih blokov, glede na število elementov *N* v vektorjih:

```
blocksPerGridMax = (N - 1) / threadsPerBlock + 1
```

  - Če je *N* zelo velik, tvegamo da bomo imeli preveliko število blokov, zato število blokov omejimo navzgor, na primer na 512:

```
blocksPerGrid = min(512, blocksPerGridMax)
```

```
// nastavi okolje za runtime:
// 1. izberemo fiksno stevilo niti v bloku:
const int threadsPerBlock = 128; // imamo 128 niti v bloku
// Sedaj moramo dolociti koliko je maximalno moznih blokov glede na
// stevilo elementov v vektorjih (N) in stevilo niti v bloku:
const int blocksPerGridMax = (N + threadsPerBlock-1)/threadsPerBlock;
// Blokov pa spet ne sme biti prevec. Zato jih navzgor omejimo na 128.
// Ce je elementov N manj kot 128*128 = 2^14 = 16K, potem imamo le
// blocksPErGridMax blokov, sicer  pa 512.
const int blocksPerGrid = IMIN(512, blocksPerGridMax);
```

Skalarni produkt – redukcija II

- Preprosto pokličemo ščepec za skalarni produkt s pravkar nastavljenim izvajalnim okoljem:

```
// klici scepec za sestevanje:
dotProductGPU<<<blocksPerGrid,threadsPerBlock>>>( d_a, d_b, d_partial_products );
```

- Ko se ščepec izvede, imamo na napravi v globalnem pomnilniku toliko delnih skalarnih produktov kolikor je bilo blokov
  - Prenesimo jih nazaj na gostitelja
  - Naj gostitelj sešteje delne skalarne produkte iz blokov

```
// Sedaj gostitelj sesteje vse delne produkte:
dotProduct=0.0;
for (int i=0; i<blocksPerGrid; i++) {
   dotProduct += h_partial_products[i];
}
```

Alokacija pomnilnika za strukture 2D

- Vzemimo, da je podatkovno polje 2D sestavljeno iz Dj vrstic in Di stolpcev (Dj x Di)?
  - Jezik C uporablja t. i. row-major order – strukturo shranjuje po vrsticah – najprej prva vrstica, nato druga vrstica,....
  - Element (j, i) iz podatkovnega polja 2D najdemo v linearnem pomnilniku na mestu j*Di + i
  - Kako tu zagotovimo poravnanost?!?
- Za alokacijo linearnega pomnilnika za strukture 2D lahko uporabimo funkcijo `cudaMallocPitch()`
  - Zagotavlja nam poravnanost operandov ter omogoča hitrejše prenose med gostiteljem in napravo
  - Podatke v tem primeru prenašamo s funkcijo `cudaMemcpy2D()`

Zgled: seštevanje matrik

- Program na CPE in ščepec

```
void addMatrix
    (float *a, float *b, float *c, int N)
{
    int i, j, idx;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            idx = i + j*N;
            c[idx] = a[idx] + b[idx];
        }
    }
}
void  main()
{
    . . .
    addMatrix(a, b, c, N);
}
```

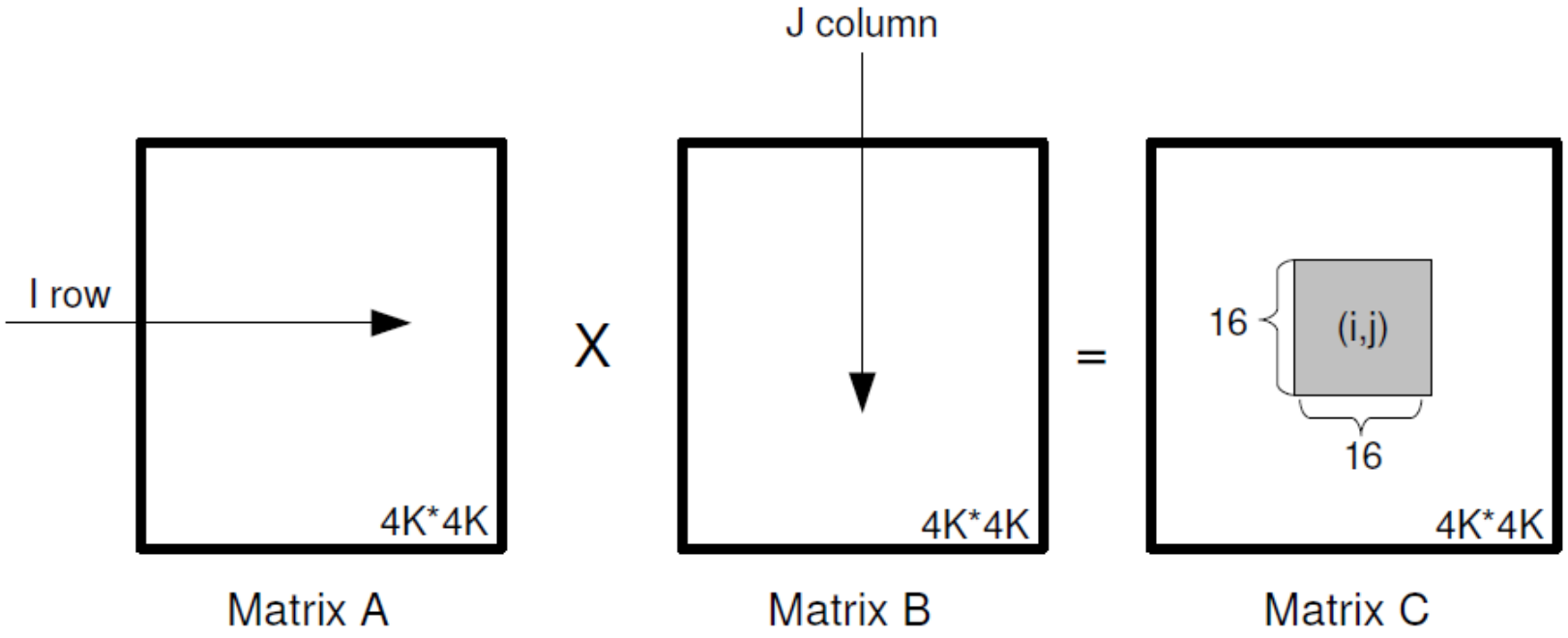
```
__global__ void addMatrixG
    (float *a, float *b, float *c, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = i + j*N;
    if (i < N && j < N)
        c[idx] = a[idx] + b[idx];
}

void  main()
{
    dim3 dimBlock (blocksize, blocksize);
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    addMatrixG<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

  - Običajno se za velikost bloka izbere 16x16, da so vsi SM dobro zaposleni

Zgled: množenje matrik

- Enostavna rešitev



Zgled: množenje matrik

- deklaracije

```
int main(int argc, char ** argv)
{
    int i, j, k;
    double sum;
    double runtime;
    float *hostA;       // The A matrix
    float *hostB;       // The B matrix
    float *hostC;       // The output C matrix
    float *hostP;       // The output C matrix
    float *deviceA;
    float *deviceB;
    float *deviceC;
    int numARows;       // number of rows in the matrix A
    int numAColumns;    // number of columns in the matrix A
    int numBRows;       // number of rows in the matrix B
    int numBColumns;    // number of columns in the matrix B
    int numCRows;       // number of rows in the matrix C (you have to set this)
    int numCColumns;    // number of columns in the matrix C (you have to set this)

    // Matrix sizes
    numARows = numAColumns = numBRows = numBColumns = MATRIX_SIZE;

    // Set numCRows and numCColumns of result matrix
    numCRows = numARows;
    numCColumns = numBColumns;
```

```
#include "cuda.h"
#include <stdio.h>

#define MATRIX_SIZE     1024
#define TILE SIZE       16
```

Zgled: množenje matrik

- Inicializacija in kopiranje podatkov

```
printf("Init\n");

// Allocate matrices on host
hostA = (float *)malloc(numARows*numAColumns*sizeof(float));
hostB = (float *)malloc(numARows*numAColumns*sizeof(float));
hostC = (float *)malloc(numCRows*numCColumns*sizeof(float));

for(i=0; i<numARows*numAColumns; i++)
    hostA[i] = rand()/(double)RAND_MAX;
for(i=0; i<numBRows*numBColumns; i++)
    hostB[i] = rand()/(double)RAND_MAX;

printf("GPU - zacetek\n");
runtime = clock()/(double)CLOCKS_PER_SEC;

// Allocate matrices on device
cudaMalloc((void **)&deviceA, numARows*numAColumns*sizeof(float));
cudaMalloc((void **)&deviceB, numBRows*numBColumns*sizeof(float));
cudaMalloc((void **)&deviceC, numCRows*numCColumns*sizeof(float));

//Copy matrices from host to device
cudaMemcpy(deviceA, hostA, numARows*numAColumns*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(deviceB, hostB, numBRows*numBColumns*sizeof(float), cudaMemcpyHostToDevice);
```

Zgled: množenje matrik

- Zaganjanje ščepca

```
//Copy matrices from host to device
cudaMemcpy(deviceA, hostA, numARows*numAColumns*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(deviceB, hostB, numBRows*numBColumns*sizeof(float), cudaMemcpyHostToDevice);

//Initialize the grid and block dimensions
dim3 dimGrid((numCColumns-1)/TILE_SIZE+1, (numCRows-1)/TILE_SIZE+1, 1);
dim3 dimBlock(TILE_SIZE, TILE_SIZE, 1);

//Launch the GPU Kernel
matrixMultiply<<<dimGrid,dimBlock>>>(deviceA, deviceB, deviceC,
                                     numARows, numAColumns,
                                     numBRows, numBColumns,
                                     numCRows, numCColumns);

cudaThreadSynchronize();
```

Zgled: množenje matrik

- Prenos podatkov na CPU in zaključek

```
    //Copy the GPU memory back to the CPU
    cudaMemcpy(hostC, deviceC, numCRows*numCColumns*sizeof(float), cudaMemcpyDeviceToHost);

    //Free the GPU memory
    cudaFree(deviceA);
    cudaFree(deviceB);
    cudaFree(deviceC);

    runtime = clock()/(double)CLOCKS_PER_SEC - runtime;
    printf("GPU - konec %lf\n", runtime);

    // Free the CPU memory
    free(hostA);
    free(hostB);
    free(hostC);

    return 0;
}
```

Zgled: množenje matrik

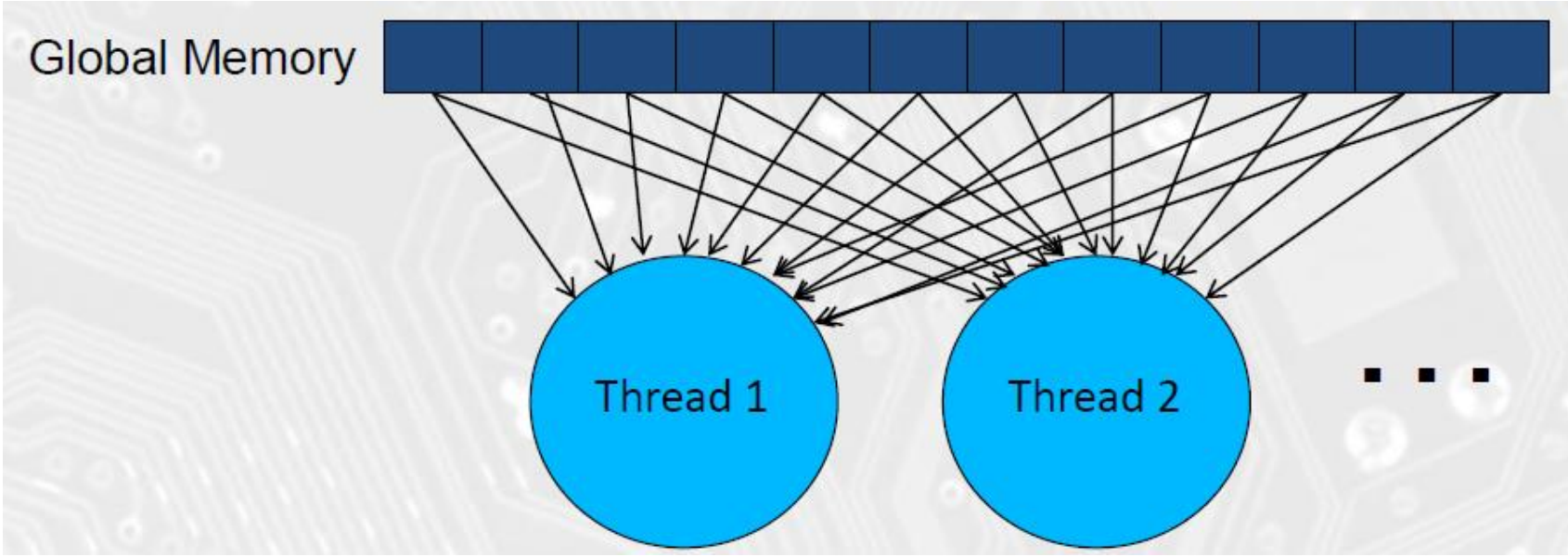
- Ščepec

```
// Compute C = A * B
__global__ void matrixMultiply(float *A, float *B, float *C,
                               int numARows, int numAColumns,
                               int numBRows, int numBColumns,
                               int numCRows, int numCColumns)
{
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((row < numCRows) && (col < numCColumns))
    {
        float sum = 0;
        for (int k = 0; k < numAColumns; k++)
            sum += A[row*numAColumns+k] * B[k*numBColumns+col];
        C[row*numCColumns+col] = sum;
    }
}
```

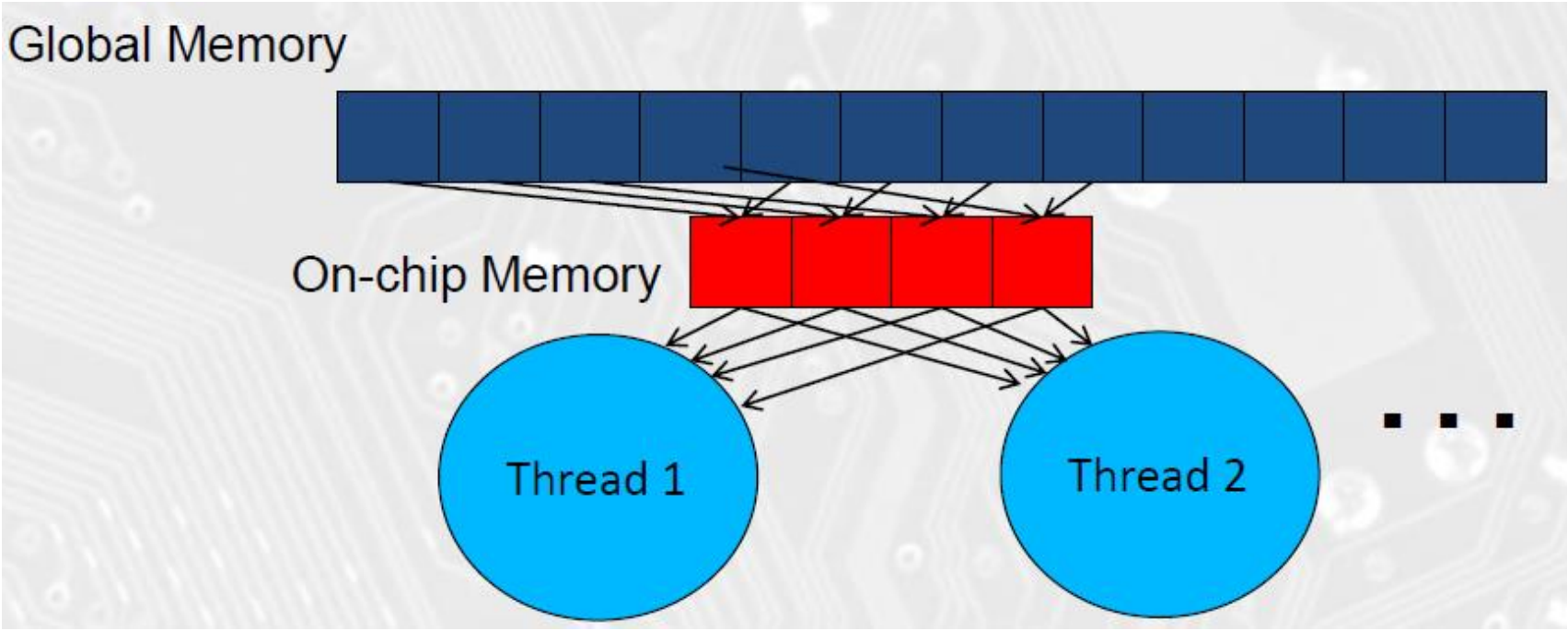
Zgled: množenje matrik

- Enostavna rešitev
  - Problem:
    - neporavnan dostop do glavnega pomnilnika
    - Dostopni časi niti do glavnega pomnilnika so različni



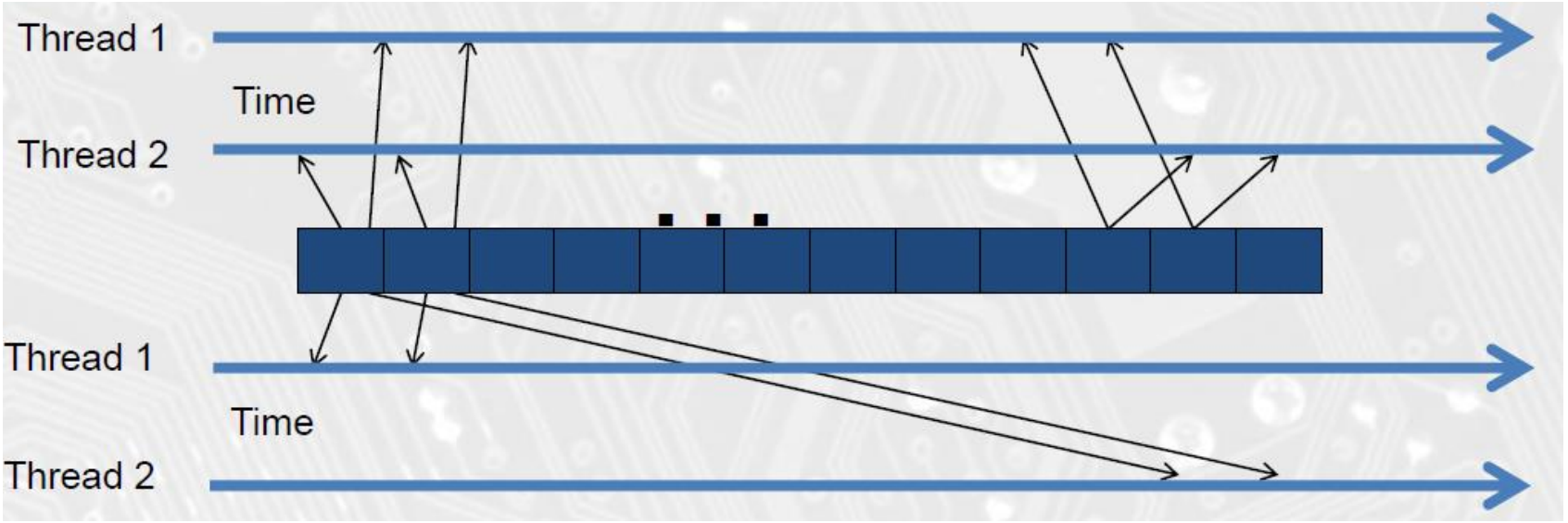
Zgled: množenje matrik

- Enostavna rešitev
  - Ideja: tehnika ploščic (ang. tiling):
    - uporaba deljenega pomnilnika
    - Poravnan prenos podatkov iz glavnega pomnilnika (dolg dostopni čas)



Zgled: množenje matrik

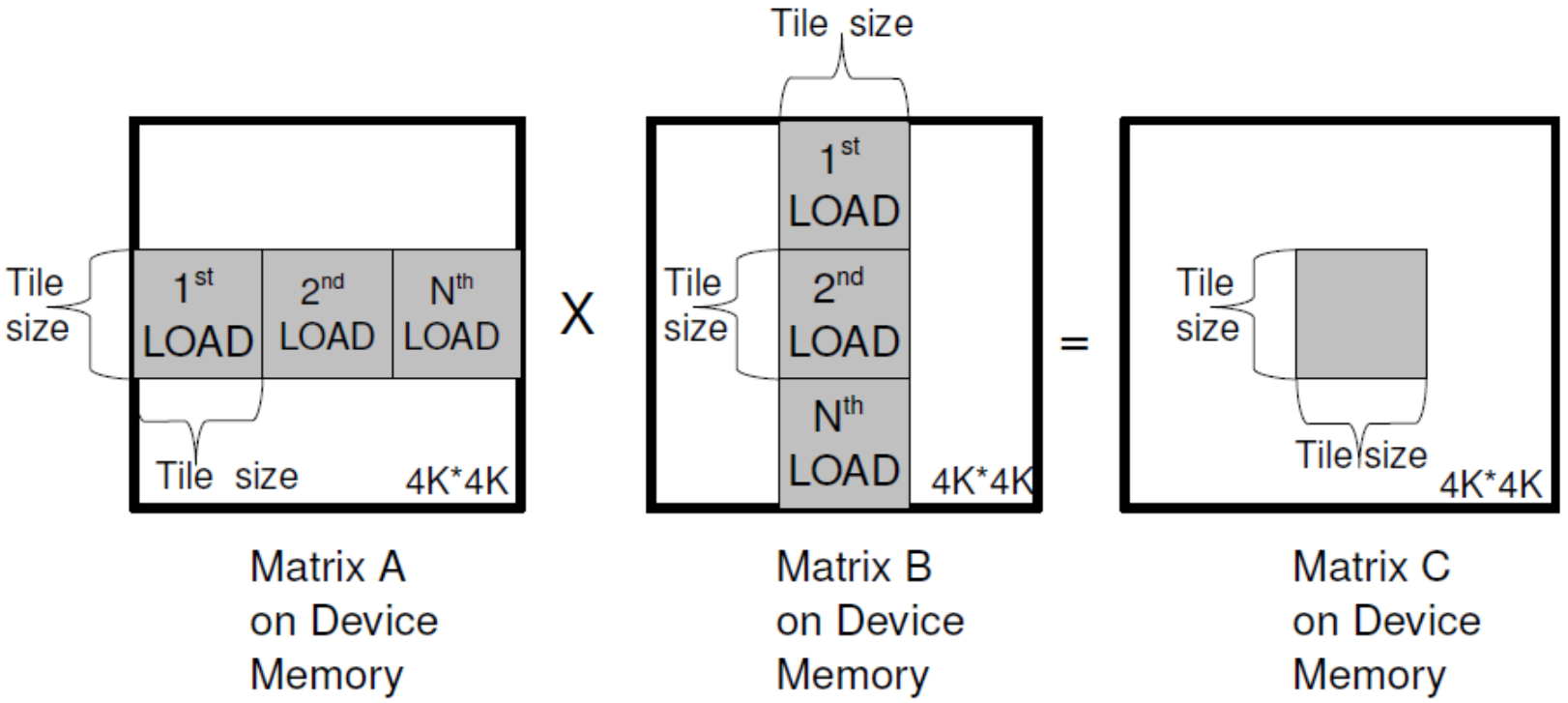
- Enostavna rešitev
  - Brez ploščic in s ploščicami
    - Slabo



    - Dobro

Zgled: množenje matrik

- Rešitev z deljenim pomnilnikom



Zgled: množenje matrik

- Ščepec – deljeni pomnilnik

```
// Compute C = A * B
__global__ void matrixMultiply(float *A, float *B, float *C, int numARows, int numAColumns,
                               int numBRows, int numBColumns, int numCRows, int numCColumns)
{
    __shared__ float ds_A[TILE_SIZE][TILE_SIZE];
    __shared__ float ds_B[TILE_SIZE][TILE_SIZE];

    float sum = 0.0;
    int row = blockIdx.y*TILE_SIZE+threadIdx.y;
    int col = blockIdx.x*TILE_SIZE+threadIdx.x;

    for (int i = 0; i < (numBRows-1)/TILE_SIZE+1; i++)
    {
        if( row < numARows && i*TILE_SIZE+threadIdx.x < numAColumns )
            ds_A[threadIdx.y][threadIdx.x] = A[row*numAColumns + i*TILE_SIZE+threadIdx.x];
        else
            ds_A[threadIdx.y][threadIdx.x] = 0;
        if( i*TILE_SIZE+threadIdx.y < numBRows && col < numBColumns )
            ds_B[threadIdx.y][threadIdx.x] = B[(i*TILE_SIZE+threadIdx.y)*numBColumns+col];
        else
            ds_B[threadIdx.y][threadIdx.x] = 0;
        __syncthreads();

        if( row < numCRows && col < numCColumns )
            for (int k = 0; k < TILE_SIZE; k++)
                sum += ds_A[threadIdx.y][k] * ds_B[k][threadIdx.x];
        __syncthreads();

        if( row < numCRows && col < numCColumns )
            C[row*numCColumns+col] = sum;
    }
}
```