Programski vmesnik CUDA [IPP:6.4-6.7]

- ščepec ali jedro predstavlja kodo, ki se izvaja na GPE
- ščepec napišemo kot zaporedni program
- programski vmesnik poskrbi za prevajanje in prenos ščepca na napravo
- ščepec izvaja vsaka nit posebej na svojih podatkih

Hierarhična organizacija niti

- mreža niti (*angl.* grid)
  - vse niti v mreži izvajajo isti ščepec
  - niti v mreži si delijo globalni pomnilnik na GPE
  - mreža je sestavljena iz blokov niti
- blok niti
  - vse niti v bloku se izvajajo na isti računski enoti
  - preko skupnega pomnilnika lahko izmenjujejo in se sinhronizirajo
- nit
  - zaporedno izvaja ščepec na svojih podatkih
  - uporablja zasebni pomnilnik
  - z nitmi v bloku si deli skupni pomnilnik
  - lahko dostopa do globalne pomnilnika in pomnilnika konstant

Označevanje niti

- 1D, 2D ali 3D oštevilčenje
- število dimenzij izberemo glede na naravo problema
- CUDA C podpira množico spremenljivk, ki odražajo organizacijo niti
  - `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
  - `blockIdx.x`, `blockIdx.y`, `blockIdx.z`
  - `blockDim.x`, `blockDim.y`, `blockDim.z`
  - `gridDim.x`, `gridDim.y`, `gridDim.z`
- niti se v snope združujejo najprej po dimenziji `x`, potem `y` in nazadnje `z`

Ščepec ali jedro

- ščepec ali jedro je koda, ki jo zaženemo na gostitelju, izvaja pa se na napravi
- ščepec napišemo kot funkcijo, ki ji pred deklaracijo dodamo ključno besedo `__global__`
- ščepec ne vrača vrednosti
- primer ščepca

```
__global__ void pozdrav(void) {
    printf("Pozdrav z naprave, nit %d.%d!\n", blockIdx.x, threadIdx.x);
}
```

- ščepec lahko kliče tudi druge funkcije na napravi, ki jih označimo s ključno besedo `__device__`

Programski vmesnik ovojnice CudaGo

Namestitev ovojnice CudaGo

- v okviru diplomskega dela je Timotej Kroflič pripravil ovojnico CudaGo za jezik go
- za namestitev ovojnice sledite navodilom na spletni strani
- za vzpostavitev okolja na gruči Arnes v lupini izvedite spodnje ukaze

```
module load Go
module load CUDA
export CGO_CFLAGS=$(pkg-config --cflags cudart-12.6)
export CGO_LDFLAGS=$(pkg-config --libs cudart-12.6)
export PATH="~/go/bin/:$PATH"

go install github.com/InternatBlackhole/cudago/CudaGo@latest # samo prvič
```

- za vzpostavitev okolja lahko uporabite skripto cudago-init.sh, ki jo zaženete z ukazom `source cudago-init.sh`

Inicializacija naprave

- pri delu z ovojnico CudaGo moramo najprej inicializirati napravo
- z inicializacijo povežemo nit, ki izvaja gorutino na gostitelju, z gonilniki naprave
- inicializacijo izvedemo z ukazom

```
dev, err := cuda.Init(device int)
```

- povezavo ob zaključku dela z napravo sprostimo z ukazom `dev.Close()`

Prevajanje metod in klic ščepca

- pripravimo ščepec v jeziku C in ga prevedemo z ukazom

  ```
  CudaGo -package cudago pozdrav-gpe.cu
  ```

- da ščepec lahko kličemo iz jezika go, ga označimo z `extern "C"`
- z zgornjim ukazom prevajalnik CudaGo pripravi paket cudago v istoimenski mapi z datotekami, ki vključujejo metode, preko katerih v jeziku go kličemo ščepec
- za ščepec `Kernel` nam prevajalnik CudaGo pripravi metodi `cudago.Kernel` in `cudago.KernelEx`
  - prvi argument podaja organizacijo blokov niti
  - drugi argument podaja organizacijo niti v bloku
  - tretji argument pri metodi `cudago.KernelEx` je velikost lokalnega pomnilnika v bajtih
  - četrti argument pri metodi `cudago.KernelEx` je povezava na tok podatkov (`nil` , ker tega ne rabimo)
  - sledijo argumenti, navedeni v ščepcu
- večdimenzionalno organizacijo blokov in niti opišemo s strukturo `cuda.Dim3`

Zaženemo prvi program na grafičnem pospeševalniku

- koda na napravi - ščepec pozdrav-gpe.cu
- koda na gostitelju pozdrav-gpe.go
- pripravimo okolje: `source ../cudago-init.sh`
- prevedemo ščepec: `CudaGo -package cudago pozdrav-gpe.cu`
- zaženemo program: `srun --partition=gpu --gpus=1 go run pozdrav-gpe.go -b 2 -t 4`

Delo s pomnilnikom in podatki

- gostitelj ima dostop samo do globalnega pomnilnika naprave

Eksplicitno prenašanje podatkov z ovojnico CudaGo

- na gostitelju pomnilnik rezerviramo z ukazom `hm := make(...)` in vanj vpišemo podatke
- globalni pomnilnik na napravi rezerviramo s klicem metode

  ```
  dm, err := cuda.DeviceMemAlloc(count uint64)
  ```

  - metoda rezervira `count` bajtov in vrne naslov v globalnem pomnilniku naprave v kazalcu `dm.Ptr`
- pomnilnik na napravi sprostimo s klicem metode `dm.Free()`
- za prenašanje podatkov iz pomnilnikom gostitelja v globalni pomnilnik naprave in uporabimo metodo

```
err = dm.MemcpyToDevice(hmPtr *unsafe.Pointer, count uint64)
```

  - metoda kopira `count` bajtov iz naslova `hmPtr = unsafe.Pointer(&hm[0])` na gostitelju na naslov `dm.Ptr` na napravi
  - funkcija je blokirajoča - izvajanje programa se nadaljuje šele po končanem prenosu podatkov
  - funkcijo `unsafe.Pointer()` uporabimo, da kazalec pretvorimo v pravi format za prenos v kodo jezika C
- za prenašanje podatkov med globalnim pomnilnikom naprave in pomnilnikom gostitelja uporabimo metodo

```
err = dm.MemcpyFromDevice(hmPtr unsafe.Pointer, count uint64)
```

  - metoda kopira `count` bajtov iz naslova `dm.Ptr` na napravi na naslov `hmPtr = unsafe.Pointer(&hm[0])` na gostitelju
  - funkcija je blokirajoča - izvajanje programa se nadaljuje šele po končanem prenosu podatkov

Delo z enotnim pomnilnikom

- novejše različice CUDA podpirajo enotni pomnilnik
- prenos podatkov izvaja CUDA po potrebi
- programer nima nadzora, večkrat manj učinkovito od eksplicitnega prenašanja
- enotni pomnilnik rezerviramo s klicem funkcije

```
m, err := cuda.ManagedMemAlloc[type](count uint64, typeSize)
```

  - `type` je podatkovni tip rezine, `count` število elementov, `typeSize` pa velikost podatkovnega tipa v bajtih
  - `m.Ptr` je kazalec na rezino na napravi
  - do elementov rezine dostopamo kot `m.Arr[index]`
- enotni pomnilnik sprostimo s klicem metode `m.Free()`

(Programski vmesnik jezika C)

Klic ščepca

- ščepec zaženemo na gostitelju, kjer med ime in argumente vrinemo trojne trikotne oklepaje
- med trojne trikotne oklepaje vpišemo organizacijo niti v mreži - število blokov in število niti v vsaki dimenziji
- za opis večdimenzionalne organizacije niti jezik CUDA C ponuja strukturo `dim3`

```
dim3 gridSize(numBlocks, 1, 1);
dim3 blockSize(numThreads, 1, 1);
pozdrav<<<gridSize, blockSize>>>();
```

Prvi program na grafičnem pospeševalniku

- pozdrav-gpeC.cu
- naložimo modul: `module load CUDA`
- kodo prevedemo s prevajalnikom za CUDA C: `srun --partition=gpu --gpus=1 nvcc -o pozdrav-gpe pozdrav-gpe.cu`
- zaženemo program: `srun --partition=gpu --gpus=1 ./pozdrav-gpe 2 4`

Rezervacija pomnilnika in prenašanje podatkov

- gostitelj ima dostop samo do globalnega pomnilnika naprave

Eksplicitno prenašanje podatkov

- na gostitelju pomnilnik rezerviramo s funkcijo `malloc` in vanj vpišemo podatke
- globalni pomnilnik na napravi rezerviramo s klicem funkcije

```
cudaError_t cudaMalloc(void** dPtr, size_t count)
```

  - funkcija rezervira `count` bajtov in vrne naslov v globalnem pomnilniku naprave v kazalcu `dPtr`
- za prenašanje podatkov med globalnim pomnilnikom naprave in pomnilnikom gostitelja uporabimo funkcijo

```
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)
```

  - funkcija kopira `count` bajtov iz naslova `src` na naslov `dst` v smeri določeni s `kind`, ki je
    - za prenos podatkov iz gostitelja na napravo `cudaMemcpyHostToDevice` in
    - za prenos podatkov iz naprave na gostitelja `cudaMemcpyDeviceToHost`
  - funkcija je blokirajoča - izvajanje programa se nadaljuje šele po končanem prenosu podatkov
- pomnilnik na napravi sprostimo s klicem funkcije

```
cudaError_t cudaFree(void *devPtr)
```

- pomnilnik na gostitelju sprostimo s klicem funkcije `free`

Enotni pomnilnik

- novejše različice CUDA podpirajo enotni pomnilnik
- prenos podatkov izvaja CUDA po potrebi
- programer nima nadzora, večkrat manj učinkovito od eksplicitnega prenašanja
- enotni pomnilnik rezerviramo s klicem funkcije

```
cudaError_t = cudaMallocManaged(void **hdPtr, count);
```

- enotni pomnilnik sprostimo s klicem funkcije

```
cudaError_t cudaFree(void *hdPtr)
```