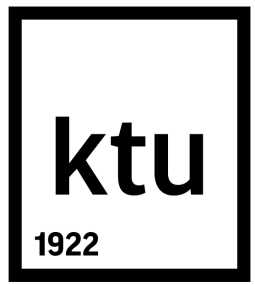
Lygiagretusis programavimas CUDA

Karolis Ryselis

Kauno Technologijos Universitetas



Two threads walk into a bar. The barkeeper looks up and yells, "hey, I want don't any conditions race like time last!"

Paskaitos turinys

1. Apie CUDA
2. CUDA atminties valdymas
3. GPU funkcijos

NVIDIA

- NVIDIA — JAV įmonė, gaminanti grafikos procesorius (*graphics processing unit — GPU*).
- GPU serijos:
  - GeForce - žaidimams ir namų kompiuteriams (GeForce 256 – pirmasis GPU);
  - Quadro - profesionalams, 2D ir 3D grafikos kūrimui;
  - Tesla - moksliniams dvigubo tikslumo skaičiavimams;
  - Tegra - procesorius mobiliesiems įtaisams;
  - NVIDIA GRID - sudėtingos grafikos žaidimai iš bet kurios pasaulio vietos.

CPU ir GPU gijos

- CPU gijos vykdomos pagrindiniame procesoriuje, GPU gijos — grafiniame procesoriuje.
- CPU ir GPU turi atskirą atmintį: CPU naudoja RAM atmintinę, GPU — VRAM atmintinę.
- CPU efektyviai gali būti vykdoma tik nedidelis kiekis gijų
- Intel Core i7-13700K turi 16 branduolių ir palaiko **24** lygiagrečias gijas.
- GPU gali būti vykdoma žymiai didesnis kiekis gijų.
- Nvidia GeForce RTX 4070 Ti turi **7680** CUDA branduolius.

CUDA sąvokos

host — pagrindinis procesorius (CPU)

device — grafinis procesorius (GPU)

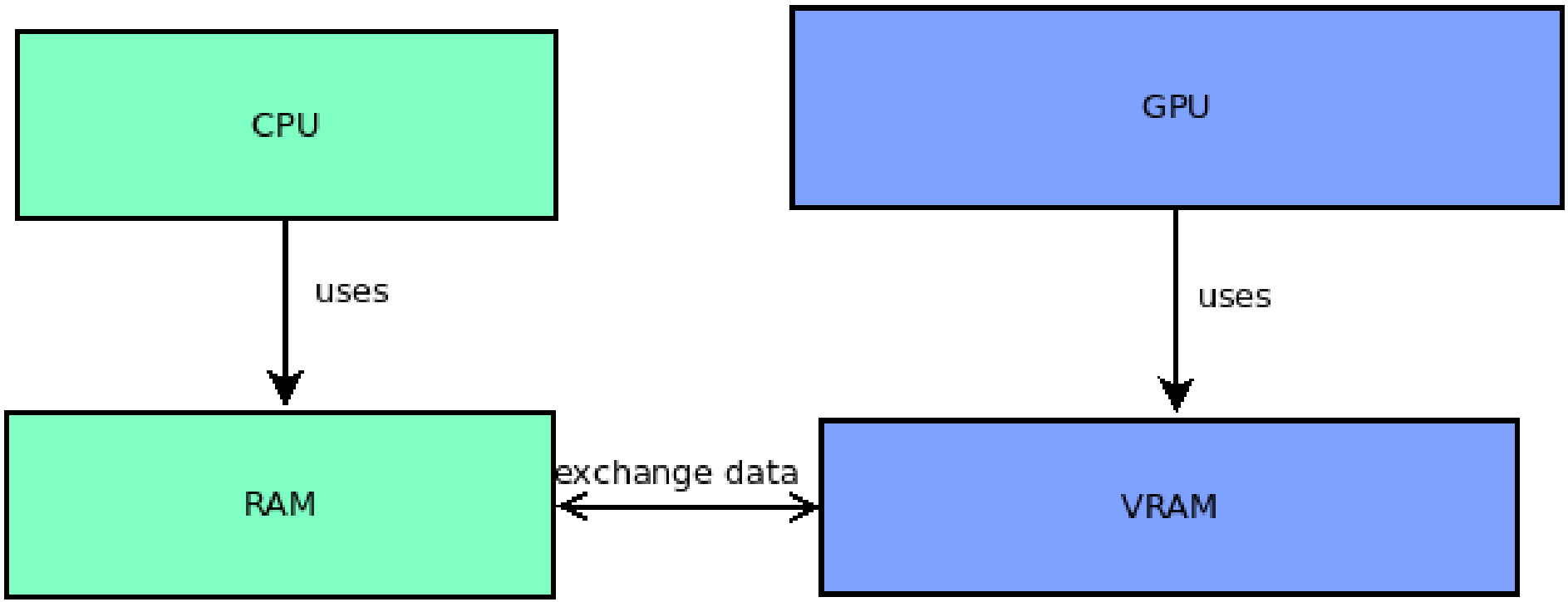
kernel — funkcija, vykdoma grafiniame procesoriuje

CUDA skirta tą pačią *kernel* funkciją vykdyti daugelyje gijų.

CUDA darbo principai

- CPU ir GPU naudoja skirtingą atmintį, todėl duomenis reikia kopijuoti iš vienos atminties į kitą.
- Darbas su GPU atmintimi CUDA vykdomas naudojant CUDA funkcijas atminties valdymui.

CPU ir GPU atmintys



CUDA atminties valdymo funkcijos

`cudaError_t cudaMalloc(void **devPtr, size_t size)`

GPU atmintyje išskiria nurodytą kiekį atminties.

`devPtr` rodyklė į rodyklę, į kurią bus įrašytas išskirtos atminties adresas.

`size` kiek baitų atminties išskirti.

`cudaError_t cudaFree(void **devPtr)`

Atlaisvina GPU išskirtą atmintį.

`devPtr` rodyklė į rodyklę, kur buvo išskirta atmintis.

CUDA atminties valdymo funkcijos

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

Kopijuoja duomenis tarp CPU ir GPU.

`dst` atminties, į kurią kopijuojami duomenys, pradžios adresas.

`src` atminties, iš kurios kopijuojami duomenys, pradžios adresas.

`count` kopijuojamų duomenų dydis.

`cudaMemcpyKind` kryptis, iš kur ir į kur kopijuojami duomenys.

CUDA atminties valdymo funkcijos

`cudaMemcpyKind` galimos reikšmės

`cudaMemcpyHostToHost` iš CPU į CPU

`cudaMemcpyHostToDevice` iš CPU į GPU

`cudaMemcpyDeviceToHost` iš GPU į CPU

`cudaMemcpyDeviceToDevice` iš GPU į GPU

GPU vykdomos funkcijos

- Funkcijos, vykdomos GPU, bet kviečiamos iš CPU, pažymimos raktiniu žodžiu `__global__`.
- `__global__ void run_on_gpu();`
- Funkcijos, vykdomos GPU ir kviečiamos iš GPU, pažymimos raktiniu žodžiu `__device__`.
- `__device__ void run_on_gpu();`

GPU funkcijų kvietimas

- GPU funkcija iš CPU kviečiama nurodant gijų bloko, kuriame bus vykdoma programa, dydį, bei gijų kiekį bloke.
- Jei funkcija kviečiama nurodant blokų ir gijų kiekius `2, 5`, bus paleidžiama 10 gijų (2 blokai po 5 gijas).
- Gijos koordinatės bloke pasiekiamas naudojantis `threadIdx.x`, bloko koordinatės — `blockIdx.x`.
- GPU funkcijos iškvietimo sintaksė:

```
run_on_gpu<<<2, 5>>>(parameter1);
```

GPU funkcijų iškvietimas

- Blokai ir gijos nebūtinai turi būti vienmačiai.
- Bloko dydis gali būti vienmatis arba dvimatis, gijų kiekis bloke — vienmatis, dvimatis arba trimatis.
- Norint nurodyti daugiamatį bloko dydį reikia naudotis CUDA struktūra `dim3`, pvz., `dim3 block(32, 32);` nurodoma, kad reikia sukurti dvimatį bloką 32×32.
- Paleidus daugiamatį bloką ar gijas jų indeksus pasiekti galima naudojant `blockIdx.x`, `blockIdx.y`, `threadIdx.x`, `threadIdx.y`, `threadIdx.z`.
- Paleidus bloką, kurio dydis 32×32 ir paleidus bloką, kurio dydis 1024, abiem atvejais bus paleista toks pat gijų kiekis.
- Paleistų gijų kiekį bloke galima patikrinti su `blockDim`, paleistų blokų kiekį — su `gridDim`.

Blokai ir gijos

- Blokų kiekis negali viršyti 65535 dėl aparatūrinės įrangos ribojimų.
- Maksimalus gijų kiekis bloke skiriasi tarp skirtingų GPU, jį galima pasitikrinti su funkcija `cudaGetDeviceProperties`, iš grąžintos struktūros paėmus atributą `maxThreadsPerBlock`.
- NVidia GeForce RTX 3080 ši reikšmė lygi 1024.

Gijų vykdymas GPU

- GPU gijas vykdo srautiniai multiprocesoriai, kurie priima gijų blokus ir juos vykdo.
- Bloko gijos vykdomos multiprocesoriuje lygiagrečiai, o kiekvienas multiprocesorius gali vykdyti vieną bloką vienu metu.
- Gijos yra grupuojamos į metmenis (angl. *warp*) po 32 ir vykdomos kartu, dėl to dažnai verta programą organizuoti taip, kad gijų kiekis bloke dalintųsi iš 32.

CPU ir GPU sinchronizacija

`cudaError_t cudaDeviceSynchronize()`

Blokuoja CPU kodą, kol GPU pabaigs visą jam priskirtą darbą.

Elementari programa

```
__global__ void run_on_gpu() {
    const char* name;
    if (threadIdx.x == 0) {
        name = "Thread 1";
    } else {
        name = "Thread 2";
    }
    execute(name);
}

__device__ void execute(const char* name) {
    printf("%s: first\n", name);
    printf("%s: second\n", name);
    printf("%s: third\n", name);
}
```

Elementari programa

```
int main() {
    run_on_gpu<<<1, 2>>>();
    cudaDeviceSynchronize();
    cout << "Finished" << endl;
}
```

Programos rezultatai:

```
Thread 1:  first
Thread 2:  first
Thread 1:  second
Thread 2:  second
Thread 1:  third
Thread 2:  third
Finished
```

Masyvų elementų sudėtis

```
__global__ void add(int* a, int* b, int* c) {
    int thread_id = threadIdx.x;
    if (thread_id < ARRAY_SIZE) {
        c[thread_id] = a[thread_id] + b[thread_id];
    }
}
```

Masyvų elementų sudėtis

```
int main() {
    int first[ARRAY_SIZE], second[ARRAY_SIZE], sum[ARRAY_SIZE];
    int *device_first, *device_second, *device_sum;
    int size = ARRAY_SIZE * sizeof(int);
    cudaMalloc((void**)&device_first, size);
    cudaMalloc((void**)&device_second, size);
    cudaMalloc((void**)&device_sum, size);
    cudaMemcpy(device_first, first, size,
        cudaMemcpyHostToDevice);
    cudaMemcpy(device_second, second, size,
        cudaMemcpyHostToDevice);
    add<<<1, ARRAY_SIZE>>>(device_first, device_second,
        device_sum);
    cudaDeviceSynchronize();
```

Masyvų elementų sudėtis

```
    cudaMemcpy(sum, device_sum, size, cudaMemcpyDeviceToHost);
    cudaFree(device_first);
    cudaFree(device_second);
    cudaFree(device_sum);
    for_each(sum, &sum[ARRAY_SIZE], [](int &n)
        { cout << n << endl;});
    return 0;
}
```

CUDA atminties tipai

- Vietinė (*local*) atmintis — matoma tik vienai gijai.
- Bendra (*shared*) atmintis — matoma visam gijų blokui. Galima sukurti prieš kintamojo paskelbimą pridedant raktinį žodį `__shared__`.
- Globali (*global*) atmintis — matoma visiems programos vykdymo metu sukurtiems blokams. `cudaMalloc` išskiria globalią atmintį.

Rodyklės CPU ir GPU atmintyje

- CPU ir GPU naudojasi bendra adresų erdve, padalinta į dvi dalis.
- Jei CPU turi 16GB RAM atmintinės, GPU — 6GB atmintinės, tai adresai iki `0x400000000` dažniausiai yra CPU atmintinės adresai, nuo `0x400000000` iki `0x580000000` yra GPU adresai.
- Tai reiškia, kad programai visos rodyklės, tiek CPU, tiek GPU, atrodo vienodai.
- Jei bandysime iš rodyklės, rodančios į GPU atmintį, pasiimti duomenis CPU kode, arba atvirkščiai, gausime klaidą.
- Nors tiek CPU, tiek GPU rodykles perdavinėti galima laisvai, pasiimti duomenis reikia tik iš savo atminties.

Matricos daugyba iš skaliaro

```
__global__ void get_doubled_matrix(int* original,
  int* result) {
    auto index = blockIdx.x * blockDim.x + threadIdx.x;
    result[index] = original[index] * 2;
}
```

Matricos daugyba iš skaliaro

```
int* flat_matrix = new int[FULL_ARRAY_SIZE];
int* flat_matrix_device, * doubled_matrix;
int* doubled_matrix_host = new int[FULL_ARRAY_SIZE];
generate_random_array(flat_matrix, FULL_ARRAY_SIZE);
cudaMalloc((void**)&flat_matrix_device,
    FULL_ARRAY_SIZE * sizeof(int));
cudaMemcpy(flat_matrix_device, flat_matrix,
    FULL_ARRAY_SIZE * sizeof(int), cudaMemcpyHostToDevice);
cudaMalloc((void**)&doubled_matrix,
    FULL_ARRAY_SIZE * sizeof(int));
get_doubled_matrix<<<ARRAY_SIZE, INNER_ARRAY_SIZE>>>(
    flat_matrix_device, doubled_matrix);
cudaDeviceSynchronize();
cudaMemcpy(doubled_matrix_host, doubled_matrix,
    FULL_ARRAY_SIZE * sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(flat_matrix_device);
cudaFree(doubled_matrix);
delete[] flat_matrix;
delete[] doubled_matrix_host;
```

CUDA atominės operacijos

- CUDA remiasi bendros atminties modeliu.
- Rašymas iš kelių CUDA gijų į tą pačią atmintį yra neapibrėžtas (*undefined behaviour*).
- CUDA turi atominių operacijų rinkinį, skirtą operacijų atomiškumui garantuoti.
- Visos CUDA atominės operacijos vykdomos viena tranzakcija ir iš kitų gijų tarpinė operacijos būsena nematoma.
- Visos CUDA atominės operacijos dirba tiek su globalia, tiek su bendra atmintimi.
- Dauguma atominių operacijų priima rodyklę į atmintį, ten saugomą reikšmę modifikuoja ir grąžina **seną** reikšmę.

CUDA atominės operacijos

`atomicAdd`	Sudeda dvi reikšmes
`atomicSub`	Atima vieną reikšmę iš kitos
`atomicExch`	Į nurodytą atmintį įrašo nurodytą reikšmę ir grąžina reikšmę, kuri buvo toje atmintyje
`atomicMin`	Į nurodytą atminties vietą įrašo nurodytą reikšmę, jei ji mažesnė už toje atminties vietoje esančią reikšmę
`atomicMax`	Į nurodytą atminties vietą įrašo nurodytą reikšmę, jei ji didesnė už toje atminties vietoje esančią reikšmę
`atomicInc`	Padidina nurodytą reikšmę 1
`atomicDec`	Sumažina nurodytą reikšmę 1
...	...

CUDA atominės operacijos

```
int initial_sum = 0;
int *device_numbers, *device_sum;
size_t *device_count;
cudaMalloc(&device_numbers, ARRAY_SIZE * sizeof(int));
cudaMalloc(&device_count, sizeof(size_t));
cudaMalloc(&device_sum, sizeof(int));
cudaMemcpy(device_numbers, numbers,
    ARRAY_SIZE * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(device_count, &ARRAY_SIZE, sizeof(size_t),
    cudaMemcpyHostToDevice);
cudaMemcpy(device_sum, &initial_sum, sizeof(int),
    cudaMemcpyHostToDevice);
```

CUDA atominės operacijos

```
cudaDeviceProp prop{};
cudaGetDeviceProperties(&prop, 0);
get_sum<<<1, prop.maxThreadsPerBlock>>>(device_numbers,
    device_count, device_sum);
int sum = 0;
cudaMemcpy(&sum, device_sum, sizeof(int),
    cudaMemcpyDeviceToHost);
cout << sum << endl;
delete [] numbers;
cudaFree(device_numbers);
cudaFree(device_sum);
cudaFree(device_count);
```

CUDA atominės operacijos

```
__global__ void get_sum(const int *data,
        const size_t* count, int* sum) {
    const auto slice_size = *count / blockDim.x;
    unsigned long start_index = slice_size * threadIdx.x;
    unsigned long end_index;
    if (threadIdx.x == blockDim.x - 1) {
        end_index = *count;
    } else {
        end_index = slice_size * (threadIdx.x + 1);
    }
    auto local_sum = 0;
    for (auto i = start_index; i < end_index; i++) {
        local_sum += data[i];
    }
    atomicAdd(sum, local_sum);
}
```

CUDA gijų sinchronizacija

- CUDA palaiko barjero tipo sinchronizaciją.
- Tokiai sinchronizacijai naudojama CUDA funkcija `__syncthreads()`.
- Iškvietus šią funkciją visos **bloko** gijos laukia, iki likusios gijos taip pat iškvies šią funkciją ir tik tada tęsia darbą.