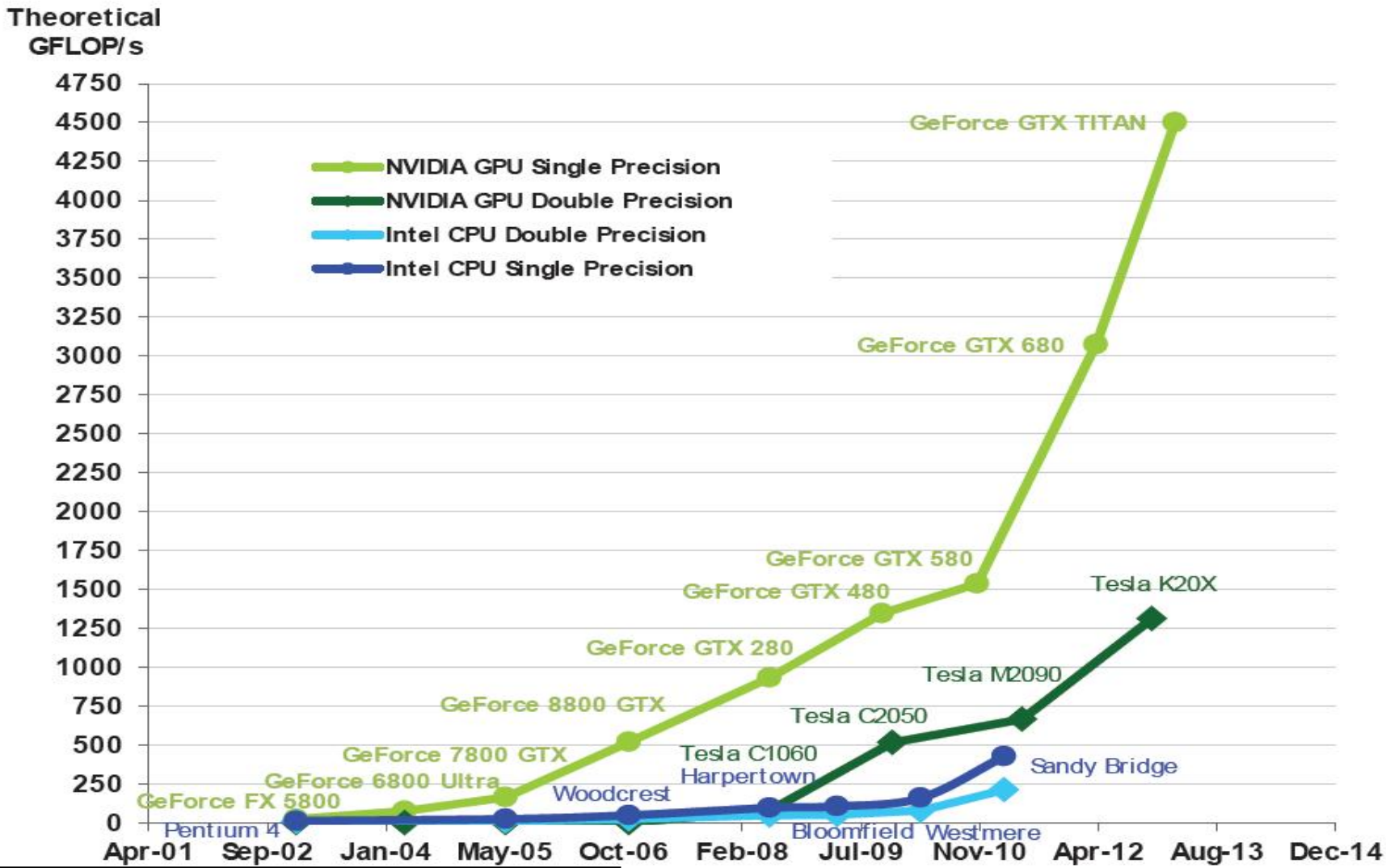
Programavimas CUDA

NVIDIA

NVIDIA (1993) – Amerikos IT kompanija - grafinių procesorių (*graphics processing unit - GPU*) gamintoja:

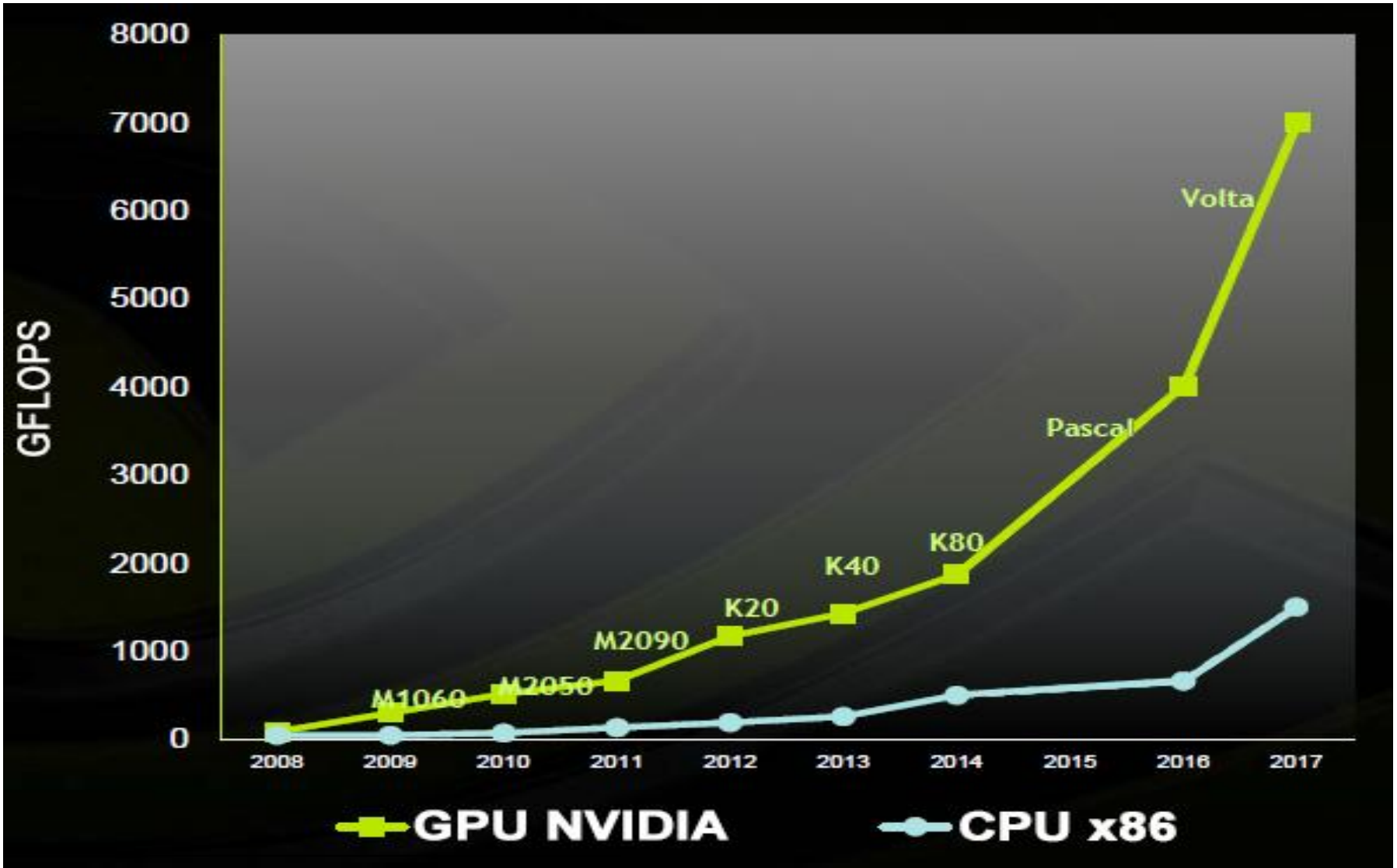
- GeForce - žaidimams ir namų kompiuteriams (GeForce 256 – pirmasis GPU);
- Quadro - profesionalams, 2D ir 3D grafikos kūrimui;
- Tesla - moksliniams dvigubo tikslumo skaičiavimams;
- Tegra - procesorius mobiliesiems įtaisams;
- NVIDIA GRID - sudėtingos grafikos žaidimai iš bet kurios pasaulio vietos.

Skaičiavimų galimybės (1)[1]



[1] - paveikslas iš NVIDIA

Skaičiavimų galimybės (2)[2]



[2]- paveikslas iš NVIDIA

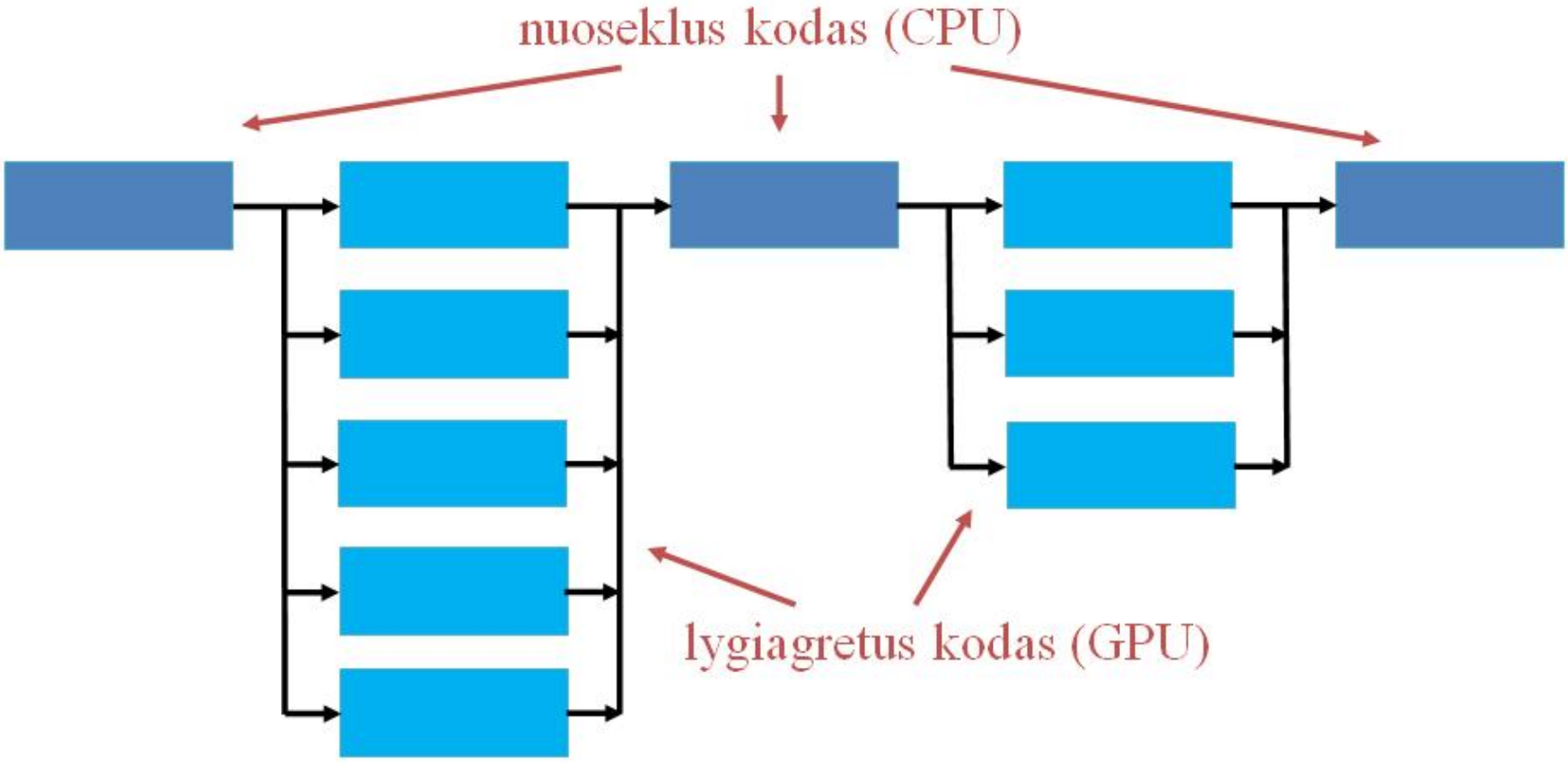
CUDA – kas tai?

- 2006 – paskelbta apie CUDA architektūros sukūrimą;
- CUDA -– tai lygiagrečiųjų skaičiavimų architektūra, kurioje didelis skaičiavimų greitis pasiekiamas naudojant GPU;
- CUDA skirtas kurti lygiagrečias programas C/C++, Fortran kalbomis;
- CUDA realizuoja duomenų lygiagretumo modelį;
- CUDA C/C++ sudaro: kompiliatorius, matematikos bibliotekos, derinimo ir optimizavimo įrankiai;
- naujausia versija: 8.0 (2016 m. rugsėjis.)

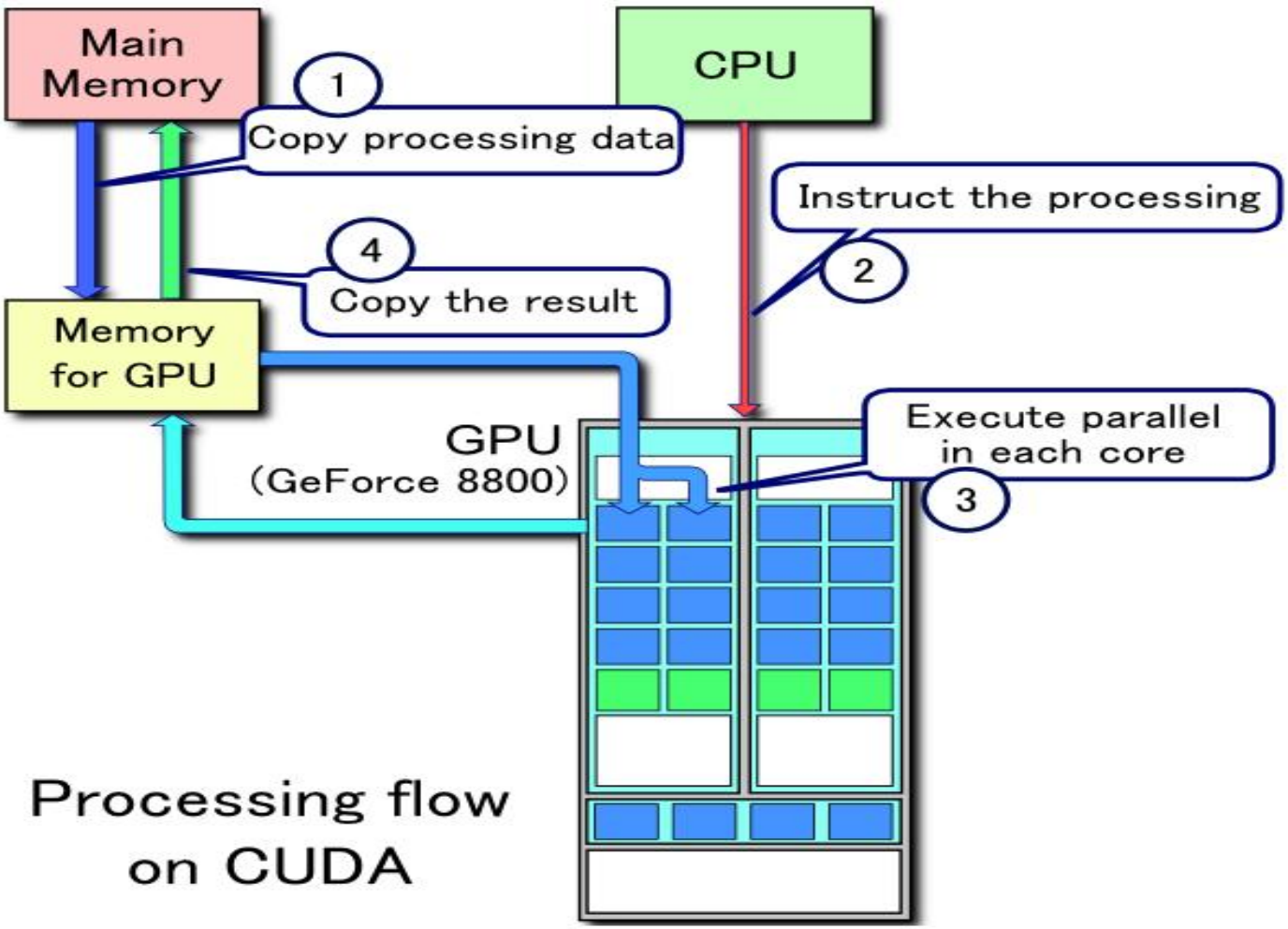
CUDA literatūra

- Jason Sanders, Edward Kandrot. CUDA by Example. Addison-Wesley, 2011.
- Shane Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs.Elsevier Inc., 2013.
- Pagrindinis **CUDA** puslapis. https://developer.nvidia.com/category/zone/cuda-zone

CUDA programos struktūra



Skaičiavimų eiga GPU[3]



[3]- paveikslas iš NVIDIA

Tipiniai programos veiksmai:

1. GPU atminties skyrimas (`cudaMalloc`);
2. duomenų kopijavimas iš CPU į GPU (`cudaMemcpy`);
3. lygiagretus GPU gijų vykdymas (`<<< >>>`);
4. duomenų kopijavimas iš GPU į CPU (`cudaMemcpy`);
5. GPU atminties atlaisvinimas (`cudaFree`).

GPU atminties skyrimas ir atlaisvinimas

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

Allocates `size` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. **cudaMalloc()** returns **cudaErrorMemoryAllocation** in case of failure.

Parameters:

devPtr - Pointer to allocated device memory
size - Requested allocation size in bytes

cudaError_t cudaFree (void * *devPtr*)

Frees the memory space pointed to by `devPtr`, which must have been returned by a previous call to **cudaMalloc()** or **cudaMallocPitch()**. Otherwise, or if **cudaFree**(`devPtr`) has already been called before, an error is returned. If `devPtr` is 0, no operation is performed. **cudaFree()** returns **cudaErrorInvalidDevicePointer** in case of failure.

Parameters:

devPtr - Device pointer to memory to free

GPU atminties skyrimo pvz.

```
int n = ...;
size_t dydis = n * sizeof(double);
double* cpu_A = (double*)malloc(dydis);
double* cpu_B = (double*)malloc(dydis);
...
double* gpu_A;
cudaMalloc(&gpu_A, dydis);
double* gpu_B;
cudaMalloc(&gpu_B, dydis);
double* gpu_C;
cudaMalloc(&gpu_C, dydis);
```

GPU atminties atlaisvinimo pvz.

```
cudaFree(gpu_A);
cudaFree(gpu_B);
cudaFree(gpu_C);
```

Duomenų kopijavimas iš CPU į GPU ir iš GPU į CPU

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

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` is one of **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**, or **cudaMemcpyDeviceToDevice**, and specifies the direction of the copy. The memory areas may not overlap. Calling **cudaMemcpy()** with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior.

Parameters:

dst	- Destination memory address
src	- Source memory address
count	- Size in bytes to copy
kind	- Type of transfer

enum cudaMemcpyKind

CUDA memory copy types

Enumerator:

cudaMemcpyHostToHost	Host -> Host
cudaMemcpyHostToDevice	Host -> Device
cudaMemcpyDeviceToHost	Device -> Host
cudaMemcpyDeviceToDevice	Device -> Device

Duomenų kopijavimo pvz.

```
// cpu_A -> gpu_A, cpu_B -> gpu_B
cudaMemcpy(gpu_A, cpu_A, dydis,
           cudaMemcpyHostToDevice);
cudaMemcpy(gpu_B, cpu_B, dydis,
           cudaMemcpyHostToDevice);
...
// gpu_C -> cpu_C
cudaMemcpy(cpu_C, gpu_C, dydis,
           cudaMemcpyDeviceToHost);
```

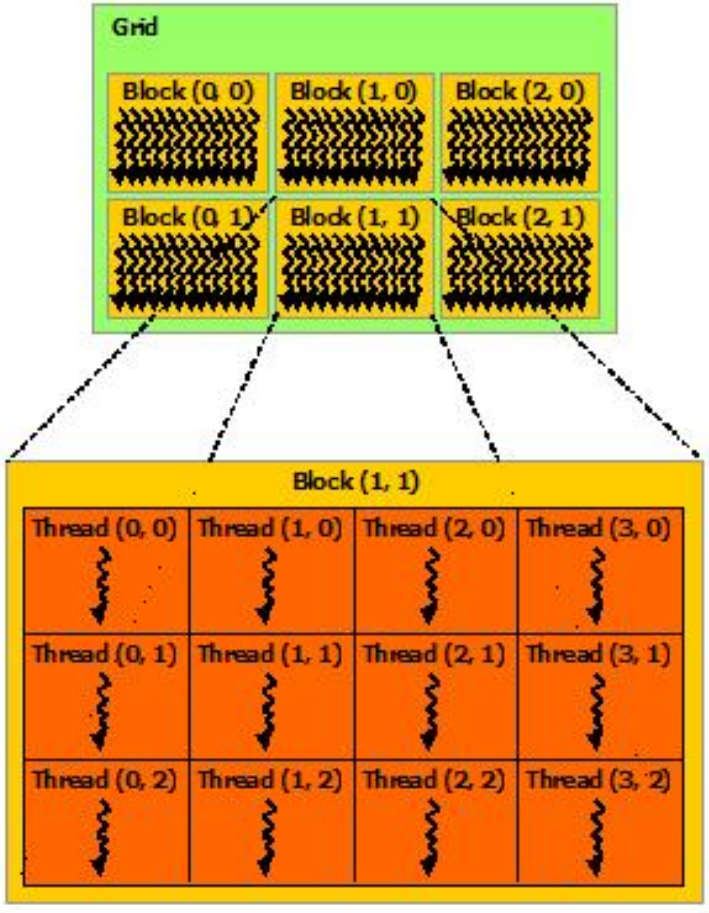
Gijų kūrimas ir vykdymas

```
// GPU (kernel'io) funkcija
__global__ void MasSud(int* A, int* B, int* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
int main()
{
  ...
  // Sudeda masyvus A(n) ir B(n); rezultatas - C(n)
  // vykdo n gijų
  MasSud<<<1, n>>>(A, B, C);
  ...
}
```

CUDA gijos

- visos gijos vykdo tą patį programinį kodą;
- gijos grupuojamos į blokus;
- gijos bloko viduje gali naudoti bendrą atmintį, vykdyti atominius veiksmus, gali būti sinchronizuojamos;
- gijos skirtinguose blokuose nesąveikauja.
- kiekviena gija turi savo unikalų ID (numerį):1D, 2D arba 3D (threadIdx.{x,y,z});

Gijų grupavimas



Gijų grupavimo pvz.

```
// GPU (kernel'io) funkcija
__global__ void MatSud(int A[n][n], int B[n][n],
                       int C[n][n])
{
  int i = threadIdx.x;  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}

  ...
  // Viename bloke n * n * 1 gijų
  int blokuSk = 1;
  dim3 gijuBloke(n, n);
  MatSud<<<blokuSk, gijuBloke>>>(A, B, C);
  ...
```

Thrust biblioteka

Thrust paskirtis

- `Thrust` – tai galinga greitų lygiagrečiųjų algoritmų ir duomenų struktūrų biblioteka.
- Bibliotekos organizavimo ir naudojimo principai panašūs C++ STL.
- `Thrust` biblioteka
  http://thrust.github.io/
- `Thrust` dokumentacija CUDA svetainėje
  http://docs.nvidia.com/cuda/thrust/

Thrust duomenų struktūros ir algoritmai

Data Structures

- `thrust::device_vector`
- `thrust::host_vector`
- `thrust::device_ptr`
- Etc.

Algorithms

- `thrust::sort`
- `thrust::reduce`
- `thrust::exclusive_scan`
- Etc.

Programos su Thrust pvz. (1)

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
...
// thrust host vektoriai
// (saugomi operatyvioje atmintyje)
thrust::host_vector<Duom> H1(n1);
skaityti(fd, H1);
rodyti(H1, "host 1-as vektorius:");

thrust::host_vector<Duom> H2(n2);
skaityti(fd, H2);
rodyti(H2, "host 2-as vektorius:");
```

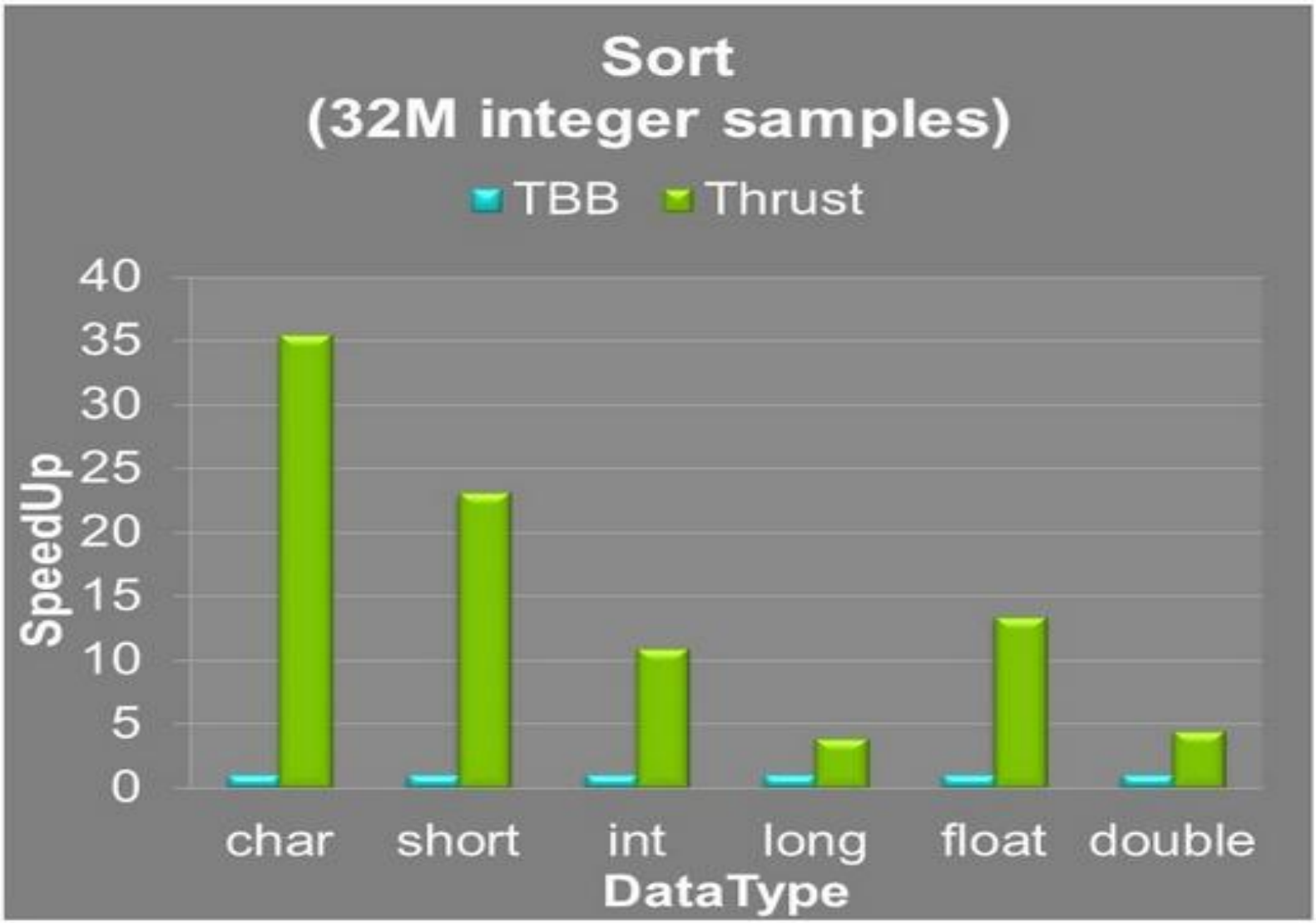
Programos su Thrust pvz. (2)

```
// thrust device vektoriai
// (saugomi GPU atmintyje)
// Kopijuoja host_vector H į device_vector D
thrust::device_vector<Duom> D1 = H1;
thrust::device_vector<Duom> D2 = H2;

thrust::device_vector<Duom> D(n1);
for(int i = 0; i < D.size(); i++) {
  D[i] = Plus(D1[i], D2[i]);
}
thrust::host_vector<Duom> H = D;
rodyti(H, "VEKTORIŲ SUMA:");
```

Thrust algoritmų panaudojimo pvz.

```
// thrust algoritmai
// ...
int x[6] = { -5, 0, 2, 3, 20, 40 };
int y[6] = { 3, 6, -2, 1, 2, 3 };
int z[6];
thrust::plus<int> operacija;
thrust::transform(x, x + 6, y, z, operacija);

// z= { -2, 6, 0, 4, 22, 43 }
```

Thrust palyginimas



Klausimai pakartojimui

1. Kokiose srityse gali būti naudojami NVIDIA procesoriai?
2. Kam skirta CUDA?
3. Koks lygiagretumo modelis realizuotas CUDA'oje?
4. Kaip vykdoma CUDA programa?
5. Kokia yra Thrust bibliotekos paskirtis?
6. Iš ko sudaryta Thrust biblioteka?
7. Kokiu būdu perduodami duomenys iš CPU į GPU ir atgal Thrust programoje?