1. Gyakorlat

CUDA Programozás

Balázs Endre Szigeti

Algoritmusok és Alkalmazásaik Tanszék
Informatikai Kar, Informatikatudományi Intézet
Eötvös Loránd Tudományegyetem

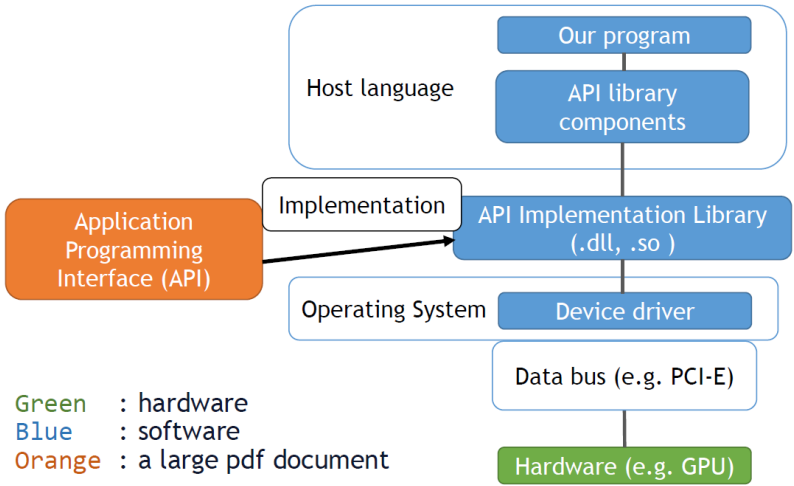
September 17, 2025

Bevezető

Ha kérdés, vagy probléma merül fel

- `https://google.com`
- Ami ide fog mutatni: `https://stackoverflow.com`
- Amennyiben nincs segítség próbálkozzatok itt: `https://docs.nvidia.com/cuda/`
- Kérdezzetek minket:
  - Teams
  - E-mail: szigetibalazs@inf.elte.hu

Hogyan lehet GPU-t programozni?



Különböző API-k

A GPU programozáshoz használt API-kat a különböző módon osztályozhatjuk

	Graphics APIs	Compute APIs
High level	OpenGL, Direct3D (9-10-11)	CUDA, OpenCL, HIP, SYCL
Low level	Vulkan, Direct3D 12, Metal, webGPU	

CUDA

- **Első megjelenés:** 2007
- **Két hozzáférési szint:**
  - *Driver API* (C driver API) — alacsony szintű, részletes vezérlés
  - *Runtime API* — C++98 nyelvkiterjesztés; speciális operátorok és dekorátorok különválasztják a host és device kódot
- **Fordító:** NVCC
- **Licensz / jelleg:** zárt, proprietáris technológia — képességei az aktuális NVIDIA hardverekre vannak hangolva

Fejlesztő: NVIDIA Corporation

Rövid kiegészítés

CUDA (Compute Unified Device Architecture) a GPGPU számítástechnika egyik vezető platformja; széles körben használt gépi tanulásban, numerikus számításokban és valós idejű nagy teljesítményű számításoknál.

CUDA fejlesztői eszközök: Nsight Systems és Nsight Compute

Nsight Systems

- Rendszerszintű teljesítményprofilozás
- Láthatóvá teszi a CPU–GPU interakciókat
- Idővonal-alapú vizualizáció
- Segít megtalálni a szűk keresztmetszeteket
- Hasznos több szálú és több GPU-s programoknál

→ **Nsight Systems: Hol van a gond?"**

Nsight Compute

- Részletes kernel-szintű elemzés
- Teljesítmény- és erőforrás-metrikák (pl. memóriahasználat, warp-kihasználtság)
- Forráskód és assembly összevetése
- Optimalizációs javaslatok
- Elsősorban egyes kernelek finomhangolására való

Nsight Compute: Miért lassú a kernel?"

GPU API-k tulajdonságai

Magas szinten, majdnem az összes API ugyanazt tudja:

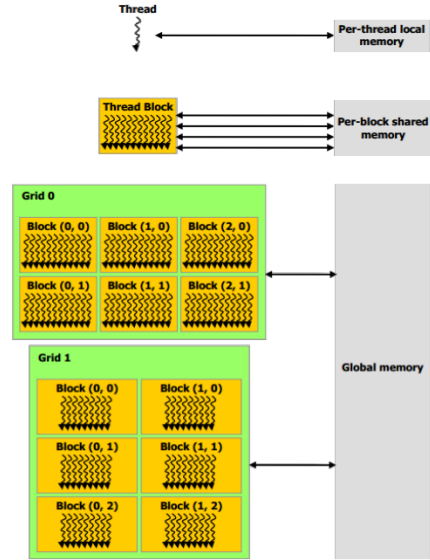
- Listázás: támogatott eszközök lekérdezése, tulajdonságok vizsgálata, eszköz(ök) kiválasztása futtatáshoz
- Memóriakezelés: memória lefoglalása az eszközön, adatmozgatás
- Eszközprogram futtatása: fordítás és/vagy bináris betöltés, majd végrehajtás a megadott adatokon
- Szinkronizáció és diagnosztika

Memóriakezelés:

- Általában explicit módon kell kezelni az adatok lefoglalását és mozgatását az eszközre.
- Két alapvetően eltérő adattárolási lehetőség van:
  - **Data buffers** — C-szerű 1D tömbök tetszőleges adattal
  - **Textures** — 1, 2 vagy 3 dimenziós adatok; korlátozott formátum, de extra szolgáltatások. Nagy előnyük, hogy optimálisabb elrendezést használnak, jobb adatlokalitással, így a szomszédos értékek gyorsabban elérhetők.

Memória Hierarchia

- **Registers / Shared memory:**
  - Nagyon gyors
  - Csak az adott szál / blokk fér hozzá
  - Élettartam: a szál vagy blokk futásáig
- **Global memory:**
  - Akár 150× lassabb a regiszternél vagy a megosztott memóriánál
  - Hozzáférhető mind a host, mind a device oldalról
  - Élettartam: az alkalmazás teljes ideje alatt



Eszközadatok

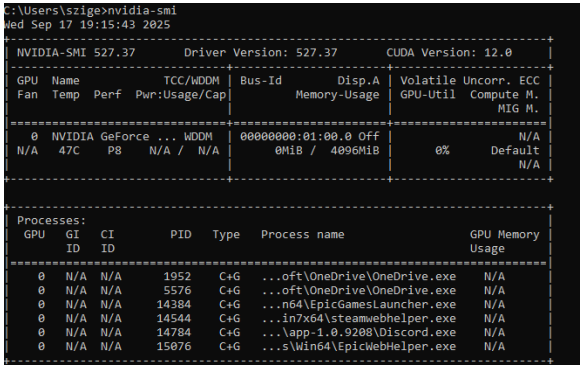


Figure: System management interface

Kezdjünk programozni!

Cave! Hic sunt dracones!

Hello World!

```
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C, ami a Host-on futtatódik
- NVIDIA compiler (`nvcc`) lehet olyan kód futtatására használni, nem tartalmazz device kódot.

Output:

```
$ nvcc hello_world.cu
$ ./a.out
Hello World!
$
```

CUDA kernel példa

```
// CUDA kernel
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Részletek:

- A CUDA C/C++ `__global__` kulcsszó olyan függvényt jelöl, amely:
  - a device-on (GPU-n) fut
  - de a Host (CPU) kódjából hívható meg
- Az `nvcc` szétválasztja a forráskódot Host és Device komponensekre:
  - Az eszközfüggvényeket (pl. `mykernel()`) az NVIDIA fordító dolgozza fel
  - A hagyományos függvényeket (pl. `main()`) a szokásos fordító (pl. `gcc`)

CUDA kernel példa

```
// CUDA kernel
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Részletek:

- A CUDA C/C++ `__global__` kulcsszó olyan függvényt jelöl, amely:
  - a device-on (GPU-n) fut
  - de a Host (CPU) kódjából hívható meg
- Az `nvcc` szétválasztja a forráskódot Host és Device komponensekre:
  - Az eszközfüggvényeket (pl. `mykernel()`) az NVIDIA fordító dolgozza fel
  - A hagyományos függvényeket (pl. `main()`) a szokásos fordító (pl. `gcc`)

Két új szintaktikai elem:

- `__global__` – megjelöli a függvényt mint *kernelt*, amely a host-ról lehet meghívni, de az eszközön fut.
- `<<< >>>` – a *végrehajtási konfiguráció szintaxisa*, rács és blokk méreteit adja meg.

CUDA szintaxis

A CUDA szintaxis törekszik az egyszerűségre, C-analógiára támaszkodva

saxpy(...){ ... } $\longrightarrow$ Definiáljuk a kernelt, amely minden x_i elemre lefut
cudaMalloc(...)
cudaMemcpy(...)
saxpy<<< ... >>> (...)
cudaMemcpy(...)
cudaFree(...)

CUDA szintaxis

A CUDA szintaxis törekszik az egyszerűségre, C-analógiára támaszkodva

saxpy(...){ ... }	$\longrightarrow$ Definiáljuk a kernelt, amely minden x_i elemre lefut
cudaMalloc(...)	$\longrightarrow$ Lefoglaljuk az adataink számára a memóriát az eszközön
cudaMemcpy(...)	
saxpy<<< ... >>> (...)	
cudaMemcpy(...)	
cudaFree(...)	

CUDA szintaxis

A CUDA szintaxis törekszik az egyszerűségre, C-analógiára támaszkodva

saxpy(...){ ... }	$\longrightarrow$ Definiáljuk a kernelt, amely minden x_i elemre lefut
cudaMalloc(...)	$\longrightarrow$ Lefoglaljuk az adataink számára a memóriát az eszközön
cudaMemcpy(...)	$\longrightarrow$ Feltöltjük az adatainkat az eszközre
saxpy<<< ... >>> (...)	
cudaMemcpy(...)	
cudaFree(...)	

CUDA szintaxis

A CUDA szintaxis törekszik az egyszerűségre, C-analógiára támaszkodva

saxpy(...){ ... }	$\longrightarrow$ Definiáljuk a kernelt, amely minden x_i elemre lefut
cudaMalloc(...)	$\longrightarrow$ Lefoglaljuk az adataink számára a memóriát az eszközön
cudaMemcpy(...)	$\longrightarrow$ Feltöltjük az adatainkat az eszközre
saxpy<<< ... >>> (...)	$\longrightarrow$ Lefuttatjuk a számítást
cudaMemcpy(...)	
cudaFree(...)	

CUDA szintaxis

A CUDA szintaxis törekszik az egyszerűségre, C-analógiára támaszkodva

saxpy(...){ ... }	$\longrightarrow$ Definiáljuk a kernelt, amely minden x_i elemre lefut
cudaMalloc(...)	$\longrightarrow$ Lefoglaljuk az adataink számára a memóriát az eszközön
cudaMemcpy(...)	$\longrightarrow$ Feltöltjük az adatainkat az eszközre
saxpy<<< ... >>> (...)	$\longrightarrow$ Lefuttatjuk a számítást
cudaMemcpy(...)	$\longrightarrow$ Letöltjük az eredményt
cudaFree(...)	

CUDA szintaxis

A CUDA szintaxis törekszik az egyszerűségre, C-analógiára támaszkodva

saxpy(...){ ... }	$\longrightarrow$ Definiáljuk a kernelt, amely minden x_i elemre lefut
cudaMalloc(...)	$\longrightarrow$ Lefoglaljuk az adataink számára a memóriát az eszközön
cudaMemcpy(...)	$\longrightarrow$ Feltöltjük az adatainkat az eszközre
saxpy<<< ... >>> (...)	$\longrightarrow$ Lefuttatjuk a számítást
cudaMemcpy(...)	$\longrightarrow$ Letöltjük az eredményt
cudaFree(...)	$\longrightarrow$ Felszabadítjuk a lefoglalt memóriát

CUDA memória kezelés (Fontos!)

- **Host és device memória elkülönül**
  - **Device pointerek** a GPU memóriára mutatnak
    - Átadhatók host kódnak
    - Nem dereferálhatók a host kódból
  - **Host pointerek** a CPU memóriára mutatnak
    - Átadhatók device kódnak
    - Nem dereferálhatók a device kódból
- **Egyszerű CUDA API az eszközmemória kezelésére**
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Hasonlóak a C megfelelőikhez: `malloc()`, `free()`, `memcpy()`

Vektorösszeadás

Feladat: $C_i = A_i + B_i \quad \forall i$

```
int main() {
    int n = 5;
    int A[5] = {1,2,3,4,5};
    int B[5] = {10,20,30,40,50};
    int C[5];

    for (int i = 0; i < n; i++)
        C[i] = A[i] + B[i];

    for (int i = 0; i < n; i++)
        printf("%d ", C[i]);
}
```

Megjegyzés: nagyobb vektorokra dinamikus memória és párhuzamosítás → CUDA

CUDA példa: SAXPY kernel

Feladat: minden elemre számítsuk ki $x_i = a \cdot x_i + y_i$

```
__global__ void saxpy(double a,
                      double *x,
                      double *y,
                      int n) {
    int gidx = blockIdx.x * blockDim.x + threadIdx.x;
    if (gidx < n) {
        x[gidx] = a * x[gidx] + y[gidx];
    }
}
```

CUDA példa: SAXPY kernel

Feladat: minden elemre számítsuk ki $x_i = a \cdot x_i + y_i$

```
__global__ void saxpy(double a,
                      double *x,
                      double *y,
                      int n) {
    int gidx = blockIdx.x * blockDim.x + threadIdx.x;
    if (gidx < n) {
        x[gidx] = a * x[gidx] + y[gidx];
    }
}
```

$\Longrightarrow$ CUDA mindig `void`-dal tér vissza és kötelező annotálni `__global__` szintaxissal

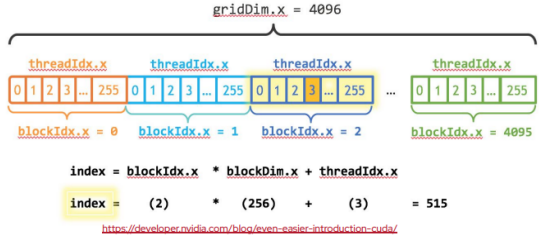
CUDA példa: SAXPY kernel

Feladat: minden elemre számítsuk ki $x_i = a \cdot x_i + y_i$

```
__global__ void saxpy(double a,
                      double *x,
                      double *y,
                      int n) {
    int gidx = blockIdx.x * blockDim.x + threadIdx.x;
    if (gidx < n) {
        x[gidx] = a * x[gidx] + y[gidx];
    }
}
```

$\Longrightarrow$ CUDA mindig `void`-dal tér vissza és kötelező annotálni `__global__` szintaxissal
$\Longrightarrow$ A beépített változók tartalmazzák a szál indexeit a többdimenziós számítási rácsban. Itt a lineáris indexet számítjuk ki.

CUDA példa: SAXPY kernel



Feladat: minden elemre számítsuk ki $x_i = a \cdot x_i$

```
__global__ void saxpy(double a,
                      double *x,
                      double *y,
                      int n) {
    int gidx = blockIdx.x * blockDim.x + threadIdx.x;
    if (gidx < n) {
        x[gidx] = a * x[gidx] + y[gidx];
    }
}
```

$\Longrightarrow$ CUDA mindig `void`-dal tér vissza és kötelező annotálni `__global__` szintaxissal

$\Longrightarrow$ A beépített változók tartalmazzák a szál indexeit a többdimenziós számítási rácsban. Itt a lineáris indexet számítjuk ki.

SAXPY: CUDA kernel vs. soros ciklus

Matematikai feladat: minden elemre számítsuk ki

$$x_i = a \cdot x_i + y_i$$

CUDA kernel

```
__global__ void saxpy(double a,
                      double *x,
                      double *y,
                      int n) {
    int gidx = blockIdx.x*blockDim.x + threadIdx.x;
    if (gidx < n) {
        x[gidx] = a*x[gidx] + y[gidx];
    }
}
```

Soros ciklus

```
for(int gidx=0; gidx<n; ++gidx) {
    x[gidx] = a*x[gidx] + y[gidx];
}
```

SAXPY: CUDA kernel vs. soros ciklus

Matematikai feladat: minden elemre számítsuk ki

$$x_i = a \cdot x_i + y_i$$

CUDA kernel

```
__global__ void saxpy(double a,
                      double *x,
                      double *y,
                      int n) {
    int gidx = blockIdx.x*blockDim.x + threadIdx.x;
    if (gidx < n) {
        x[gidx] = a*x[gidx] + y[gidx];
    }
}
```

Soros ciklus

```
for(int gidx=0; gidx<n; ++gidx) {
    x[gidx] = a*x[gidx] + y[gidx];
}
```

Megjegyzés: A CUDA kernel minden szál egy elemre számol, míg a soros ciklus egymás után végzi az összeadást.

Memóriafoglalás a Host-on

Feladat: foglaljuk le a GPU memóriát a vektorok számára.

```
double *buffer_x = nullptr;
double *buffer_y = nullptr;

// Memoriafoglalas a GPU-n
cudaMalloc((void**)&buffer_x, N * sizeof(double));
cudaMalloc((void**)&buffer_y, N * sizeof(double));
```

Memóriafoglalás a Host-on

Feladat: foglaljuk le a GPU memóriát a vektorok számára.

```
double *buffer_x = nullptr;
double *buffer_y = nullptr;

// Memoriafoglalas a GPU-n
cudaMalloc((void**)&buffer_x, N * sizeof(double));
cudaMalloc((void**)&buffer_y, N * sizeof(double));
```

Megjegyzések:

- `buffer_x` és `buffer_y` pointerek a GPU memóriát fogják mutatni.
- `cudaMalloc` lefoglal N darab `double` méretű elemet.
- A foglalt memóriát később `cudaFree()`-val kell felszabadítani.
- Hasonló a C `malloc()`-hoz, de a GPU memóriára vonatkozik.

Adatok feltöltése a GPU-ra

Feladat: másoljuk az adatokat a host memóriából a GPU memóriába.

```
const int N = ...;
std::vector<double> X(N); // Input data here
std::vector<double> Y(N); // and here

// Copy to GPU
cudaMemcpy(buffer_x, X.data(), N * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(buffer_y, Y.data(), N * sizeof(double), cudaMemcpyHostToDevice);
```

Adatok feltöltése a GPU-ra

Feladat: másoljuk az adatokat a host memóriából a GPU memóriába.

```
const int N = ...;
std::vector<double> X(N); // Input data here
std::vector<double> Y(N); // and here

// Copy to GPU
cudaMemcpy(buffer_x, X.data(), N * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(buffer_y, Y.data(), N * sizeof(double), cudaMemcpyHostToDevice);
```

Megjegyzések:

- `buffer_x` és `buffer_y` most már tartalmazzák a host adatait a GPU memóriában.
- `cudaMemcpy` hasonló a C/C++ `memcpy()` függvényéhez, de host → device irányban.
- A másolás irányát az utolsó paraméter (‘cudaMemcpyHostToDevice‘) határozza meg.
- Hasonlóan lehet visszamásolni a GPU eredményt host-ra

CUDA kernel indítása: SAXPY

Matematikai feladat: minden elemre számítsuk ki $x_i = a \cdot x_i + y_i$

```
// Kernel inditas
dim3 grid_size(1);
dim3 block_size(N);
saxpy<<<grid_size, block_size>>>(scalar, buffer_x, buffer_y, N);
```

CUDA kernel indítása: SAXPY

Matematikai feladat: minden elemre számítsuk ki $x_i = a \cdot x_i + y_i$

```
// Kernel inditas
dim3 grid_size(1);
dim3 block_size(N);
saxpy<<<grid_size, block_size>>>(scalar, buffer_x, buffer_y, N);
```

Megjegyzések:

- Itt 1 blokk indítódik N szállal.
- Egy blokkban maximum 1024 szál lehet.
- Ha több szálra van szükség, növelni kell a `grid_size`-ot.
- Minden szál egy-egy elemet számol ki a vektorban.

Eredmények visszamásolása és memória felszabadítása

```
std::vector<double> Result(N);

// Eredmenyek visszamasolasa a host memriba
cudaMemcpy(Result.data(), buffer_x, N*sizeof(double), cudaMemcpyDeviceToHost);

// GPU memoria felszabaditasa
cudaFree(buffer_x);
cudaFree(buffer_y);
```

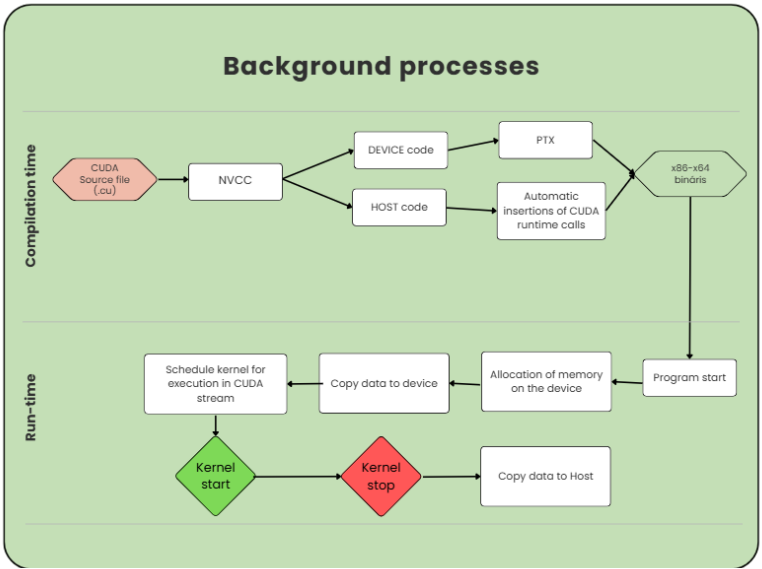
Eredmények visszamásolása és memória felszabadítása

```
std::vector<double> Result(N);

// Eredmenyek visszamasolasa a host memriba
cudaMemcpy(Result.data(), buffer_x, N*sizeof(double), cudaMemcpyDeviceToHost);

// GPU memoria felszabaditasa
cudaFree(buffer_x);
cudaFree(buffer_y);
```

Megjegyzések:

- Figyeld a `cudaMemcpyDeviceToHost` irányt, ami a GPU → host másolást jelzi.
- A `cudaFree()`-val felszabadítjuk a korábban lefoglalt GPU memóriát.
- Mindig ügyelj a memória felszabadítására a program végén!

Mi zajlik a háttérben



CUDA vs. OpenCL összehasonlítás

Kérdés	CUDA	OpenCL
Típusa	Zárt technológia	Szabadon implementálható szabvány
Ki fejleszti?	NVIDIA	Khronos, sok gyártó
Támogatott hardver	NVIDIA GPU-k	Sokféle eszköz (CPU, GPU, FPGA, DSP, ...)
Forráskód	Egyetlen source	Külön source
Nyelv	C / C++ nyújtás	A host oldal bármilyen nyelven kommunikálhat az API-val,
Fordító	nvcc	A host kód a nyelvi fordítóval, a device kód a gyártó által biztosított driverrel fordul

CUDA vs. OpenCL: fontos szempontok

Szempont	CUDA	OpenCL
Inicializáció	Implicit	Platform, eszköz, kontextus és parancssor létrehozása
Adatkezelés	Explicit memóriafoglalás, másolás a GPU-ra, visszamásolás a host-ra	Explicit memóriafoglalás, bizonyos adatmozgások implicit módon történnek, a host-ra visszamásolás explicit
Kernel	Olyan, mint egy függvényhívás, de speciális szintaxissal	Forráskód betöltése, programobjektum létrehozása, fordítás, argumentumok beállítása, parancssorba helyezés végrehajtásra
CQM	Implicit	Explicit

CUDA / OpenCL szótár

CUDA	OpenCL
Grid	NDRange
Thread block	Work group
Thread	Work item
Thread ID	Global ID
Block index	Block ID
Thread index	Local ID
Shared memory	Local memory
Registers	Private memory

CUDA programozási jó tanácsok

Teljesítmény tippek

- Minimalizáld a host–device másolásokat
- Használd a gyors memóriát: registers → shared → global
- Egymáshoz közeli szálak egymáshoz közeli címeket olvassanak
- Sok szál indítása a magas kihasználtság érdekében
- Azonos munkát végezzenek a szálak)
- Kerüld a felesleges kernelindításokat

Hibakeresés és best practice

- Használd a *cuda-memcheck*-et memóriahibákhoz
- Profilozz Nsight Systems és Nsight Compute segítségével
- Tartsd külön a host és device kódot
- Kerüld a túl sok szinkronizációt
- Olvasható, moduláris kód → könnyebb optimalizálni

Először helyesen kell működnie, csak utána érdemes optimalizálni!