

1. Gyakorlat

CUDA Programozás

Balázs Endre Szigeti

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

2026. szeptember 10.

Bevezető

Segítség a gyakorlaton

- Elsődleges referencia: NVIDIA CUDA dokumentáció.
- Kérdések: Teams vagy szigetibalazs@inf.elte.hu.
- Hibajelentéshez: minimális kód, teljes hibaüzenet, GPU típusa, driver- és Toolkit-verzió.
- Fórumválaszoknál ellenőrizzük a dátumot és a verziót.

Előismeret: C/C++ tömbök, pointerek, függvények és ciklusok.

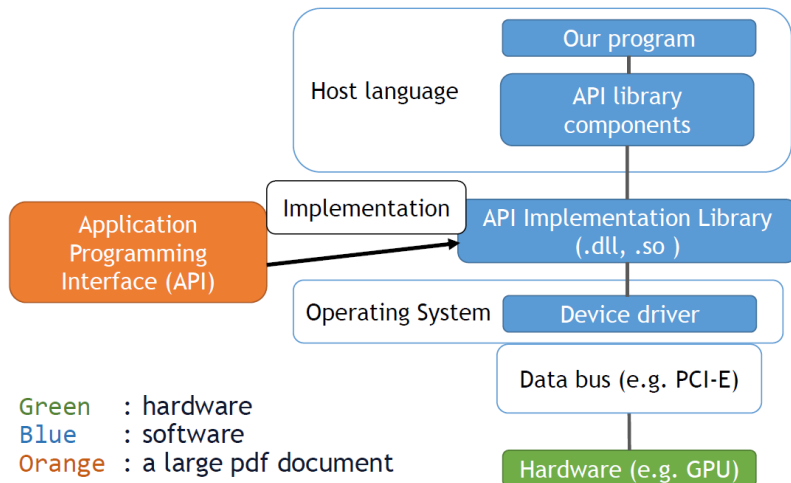
A gyakorlat céljai

- ➊ Megkülönböztetni a host és device szerepét.
- ➋ Tömbindexet képezni a blokk és a szál azonosítójából.
- ➌ Lefordítani és ellenőrizni egy több blokkos DAXPY programot.
- ➍ Külön mérni a kernel idejét és az adatmozgatást is tartalmazó időt.

Központi példa

$y_i \leftarrow ax_i + y_i$. Az egyes kimeneti elemek függetlenek, ezért könnyen kioszthatók külön szálaknak.

Hogyan lehet GPU-t programozni?



Különböző API-k

A GPU-programozáshoz használt API-kat többféleképpen osztályozhatjuk.

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

Hardveres áttekintés

A gyakorlatban diszkrét GPU-t feltételezünk: egy *PCIe* grafikus kártya, amely sokmagos GPU-val és nagy sebességű grafikus *device* memóriával rendelkezik, egy szabványos PC/szerver rendszerben helyezkedik el, amely többmagos CPU-t tartalmaz.

- A CPU tipikusan a vezérlési logikát, az elő- és utófeldolgozást, valamint az I/O műveleteket kezeli.
- A CPU és a GPU közötti kommunikáció a *PCIe* buszon keresztül történik, amelynek sávszélessége és késleltetése kritikus teljesítménytényező.
- A GPU saját, nagy sávszélességű *device* memóriával rendelkezik, amely elkülönül a hoszt (CPU) memóriájától.

Gyakorlati következmény

A hatékony GPU-gyorsítás kulcsa a párhuzamosítható feladatok GPU-ra helyezése, valamint a CPU–GPU adatmozgatás minimalizálása.

HPC és professzionális GPU-k fejlődése

Architektúra	Példa / CC	Megjegyzés
Ampere	A100: 8.0	HPC és gépi tanulás
Ampere	RTX A2000: 8.6	Munkaállomás, korlátozott FP64
Ada Lovelace	L4, L40: 8.9	Grafika és inferencia
Hopper	H100, H200: 9.0	Adatközponti számítás
Blackwell	B200: 10.0	A Blackwell név több CC-változatot is lefed

Compute capability (CC)

A hardver által támogatott CUDA-képességeket jelzi. Nem a telepített CUDA Toolkit verziója, és nem teljesítménypontszám.

Forrás: NVIDIA GPU / CC táblázat. A példák nem alkotnak teljes termékkatalógust.

NVIDIA GPU felépítése: SM – RTX A2000

Asztali RTX A2000: Ampere, CC 8.6, 3328 CUDA mag, 26 SM.

- SM-enként legfeljebb 48 rezidens warp, azaz 1536 szál.
- 64 K darab 32 bites regiszter SM-enként.
- 128 KiB közös L1/shared kapacitás, ebből legfeljebb 100 KiB shared.
- 6 vagy 12 GB GDDR6 memória.
- Névleges sávszélesség: 288 GB/s.
- Az elérhető párhuzamosságot a kernel regiszter- és shared memóriaigénye is korlátozza.

A CUDA-szál szoftveres végrehajtási egység. A magok száma nem azonos az egyszerre rezidens szálak számával.

Forrás: Ampere Tuning Guide; RTX A2000 adatlap.

Szál, blokk, grid és SM

- **Szál:** a kernel egy példánya, saját indexekkel és változókkal.
- **Blokk:** szálak csoportja, blokkhoz tartozó shared memóriával.
- **Grid:** az adott kernelindítás összes blokkja.
- **SM:** a hardver, amelyen egy vagy több blokk rezidens lehet.

$$i = \underbrace{\text{blockIdx.x}}_{\text{blokkindex}} \underbrace{\text{blockDim.x}}_{\text{blokkméret}} + \underbrace{\text{threadIdx.x}}_{\text{blokkon belüli index}}$$

Egy hagyományos blokk egy SM-en fut. A blokkok sorrendjére nem támaszkodhatunk. A grid nagyobb lehet a rezidens kapacitásnál.

Warp és SIMT

- Egy warp 32 szálból áll. A SIMT jelentése: *Single Instruction, Multiple Threads*.
- Az egy warpba tartozó szálak eltérő adatokat kezelhetnek.
- Eltérő elágazásoknál egyes szálak inaktívak lehetnek az adott ág végrehajtása közben: ez a warp divergence.
- A tömbhatárt védő feltétel szükséges. A teljesítmény miatt nem hagyhatjuk el a helyességet biztosító ellenőrzést.

Példa

256 szál/blokk esetén 8 warp jut egy blokkra. A 32 többszöröse jó kiindulás, de nem helyességi követelmény.

CUDA

- NVIDIA párhuzamos számítási platform, első megjelenés: 2007.
- **CUDA C++:** nyelvi kiterjesztések a GPU-kódhoz, például `__global__` és a kernelindítás szintaxisa.
- **Runtime API:** például `cudaMalloc`, `cudaMemcpy`. A bevezető gyakorlat ezt használja.
- **Driver API:** közvetlenebb kontextus- és modulkezelés.
- **nvcc:** együttműködik a host C++ fordítóval. A támogatott C++ szabvány a Toolkit és a host fordító verziójától függ.

A gyakorlat nyelve

CUDA C++, `.cu` forrásfájlokkal. Az API és a nyelvi kiterjesztés külön fogalom. A CUDA környezet NVIDIA GPU-kra épül.

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

A GPU API-k közös alpműveletei:

- 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

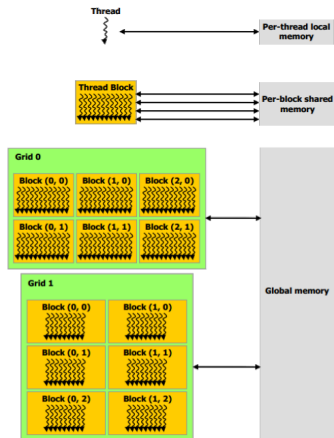
Memóriakezelés:

- 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. Előnyük a hozzáférési mintától függ, általános tömbművelethez nem automatikusan gyorsabbak.

Memóriahierarchia

- **Regiszter:** szához tartozó, chipen belüli tároló.
- **Shared memory:** a blokk számai között megosztható, a blokk élettartamára. Használatát a program szervezi.
- **Global memory:** a GPU számai elérhetik. A dinamikus foglalás felszabadításig él, több kernelen át is.
- **Local memory:** szálprivát címterület, de jellemzően az eszközmemóriában, cache-elve. Regiszterkifutás is ide vezethet.

A késleltetés függ a cache-találattól és a hozzáférési mintától.
Nincs általános, állandó “150-szeres” arány.



```
C:\Users\szige>nvidia-smi
Wed Sep 17 19:15:43 2025
```

NVIDIA-SMI 527.37				Driver Version: 527.37				CUDA Version: 12.0			
GPU	Name	TCC/WDDM		Bus-Id	Disp.A	Volatile	Uncorr.	ECC			
Fan	Temp	Perf	Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute	M.			
							MIG	M.			
0	NVIDIA GeForce ...	WDDM		00000000:01:00.0	Off			N/A			
N/A	47C	P8	N/A / N/A		0MiB / 4096MiB	0%	Default	N/A			

Processes:											
GPU	GI	CI	PID	Type	Process name	GPU Memory					
	ID	ID				Usage					
0	N/A	N/A	1952	C+G	...oft\OneDrive\OneDrive.exe	N/A					
0	N/A	N/A	5576	C+G	...oft\OneDrive\OneDrive.exe	N/A					
0	N/A	N/A	14384	C+G	...n64\EpicGamesLauncher.exe	N/A					
0	N/A	N/A	14544	C+G	...in7x64\steamwebhelper.exe	N/A					
0	N/A	N/A	14784	C+G	...app-1.0.9208\Discord.exe	N/A					
0	N/A	N/A	15076	C+G	...s\Win64\EpicWebHelper.exe	N/A					

Figure: System management interface

A fejlesztői környezet ellenőrzése

```
nvidia-smi  
nvcc --version  
nvcc -std=c++17 -O2 -lineinfo daxpy.cu -o daxpy  
./daxpy 1000  
compute-sanitizer --tool memcheck ./daxpy 1000
```

- Az `nvidia-smi` CUDA-verziója a driver által támogatott verziót jelzi, nem a telepített Toolkit kiadását.
- A `nvcc -version` a fordító/Toolkit kiadását mutatja.
- A GPU számát és képességeit a teljes kód `cudaGetDeviceCount` és `cudaGetDeviceProperties` hívással is ellenőrzi.

GPU programozási folyamat (CPU nézőpont)

Felső szintű működés

Legfelső szinten egy fő folyamat fut a CPU-n, amely az alábbi lépéseket hajtja végre:

- 1 a grafikus kártya inicializálása
- 2 memória lefoglalása a host-on és a device-on
- 3 adatok másolása a host-ról a device memoriába
- 4 egy vagy több *kernel* indítása sok párhuzamos szállal
- 5 adatok visszamásolása a device memóriából a host-ra
- 6 a 3–5. lépések ismétlése szükség szerint
- 7 minden memória felszabadítása és a program lezárása

Megjegyzés

A teljesítmény szempontjából kritikus az adatmozgatások számának minimalizálása és a kernelhívások hatékony szervezése.

Kezdjünk programozni!

Cave! Hic sunt dracones!

Hello World!

```
#include <stdio>
int main(void) {
    std::printf("Hello World!\n");
    return 0;
}
```

- Egyszerű C++ program, amely a hoston fut
- Az NVIDIA fordítója (nvcc) GPU-kód nélküli forrást is le tud fordítani.

Output:

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

CUDA kernel példa

```
#include <cuda_runtime.h>
#include <cstdio>
__global__ void mykernel() { }
int main() {
    mykernel<<<1, 1>>>();
    cudaError_t err = cudaGetLastError();
    if (err == cudaSuccess) err = cudaDeviceSynchronize();
    if (err != cudaSuccess) {
        std::fprintf(stderr, "%s\n", cudaGetErrorString(err));
        return 1;
    }
    std::printf("A kernel befejezodott.\n");
}
```

A kernelindítás a host számára aszinkron. A `__global__` kernel visszatérési típusa `void`. A `__device__` segédfüggvény visszaadhat értéket.

CUDA szintaxis

Kódelem	Feladat
<code>__global__ void daxpy(...)</code>	A párhuzamos számítás definíciója
<code>cudaMalloc(...)</code>	Eszközmemória foglalása
<code>cudaMemcpy(...)</code>	Bemenet másolása az eszközre
<code>daxpy<<blocks, threads>>(...)</code>	A kernel indítása
<code>cudaMemcpy(...)</code>	Eredmény visszamásolása
<code>cudaFree(...)</code>	Eszközmemória felszabadítása

Host és device memória

A most használt modell: explicit másolás

`std::vector` a host memóriában, `cudaMalloc` az eszközön. A host tárolhat device pointert, de ebben a modellben nem dereferálja. A közönséges host tömb sem használható közvetlen GPU-bemenetként.

- Host–device másolás: `cudaMemcpy` és a megfelelő irány.
- A pointer átadása önmagában nem másolja át a megcímzett adatot.
- **Más modellek:** Unified Memory (`cudaMallocManaged`), illetve megfelelően leképezett host memória.
- Ezekhez is helyes szinkronizáció kell. A közös címezés nem jelent automatikusan azonos elérési sebességet.

Vektorösszeadás

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

```
#include <stdio>
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++)
        std::printf("%d ", C[i]);
}
```

Megjegyzés: nagyobb vektorokra dinamikus memória és párhuzamosítás → CUDA

CUDA példa: DAXPY kernel

Feladat: $y_i \leftarrow ax_i + y_i$, az x bemenet változatlan.

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

- DAXPY: double precision. A float változat neve SAXPY.
- Minden érvényes szál egy saját y_i elemet ír.
- Az utolsó blokk túlérhet a tömbön, ezért kell az `if (i < n)`.

A teljes példában $1 \leq N \leq 10^7$, ezért a használt 32 bites indexelés nem csordul túl. Külön x és y tömböt használunk.

DAXPY: CUDA kernel vs. soros ciklus

Matematikai feladat: minden elemre számítsuk ki

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

CUDA kernel

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

Soros ciklus

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

Megjegyzés: A kernelben egy szál egy elemet kezel. A soros ciklus egymás után számol.

Eszközmemória foglalása host kódból

Feladat: foglaljuk le a GPU memóriát a vektorok számára.

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

Megjegyzések:

- `buffer_x` és `buffer_y` pointerok a GPU memóriájára mutatnak.
- `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 = 1000;
std::vector<double> X(N); // initialize before copying
std::vector<double> Y0(N); // original second input

// Copy to GPU
CUDA_CHECK(cudaMemcpy(buffer_x, X.data(), N * sizeof(double),
    cudaMemcpyHostToDevice));
CUDA_CHECK(cudaMemcpy(buffer_y, Y0.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: DAXPY

```
const int threads = 256;
const int blocks = (N + threads - 1) / threads;
daxpy<<<blocks, threads>>>(scalar, buffer_x, buffer_y, N);
CUDA_CHECK(cudaGetLastError());
CUDA_CHECK(cudaDeviceSynchronize()); // oktatasi ellenorzes
```

- $B = \lceil N/T \rceil$ blokk kell T szállal, ha $N > 0$.
- $N = 1000$, $T = 256$: 4 blokk, 1024 szál, 24 szál nem dolgozik.
- A 256 kiinduló érték. A blokk- és erőforráskorlátokat ellenőrizzük.
- A CUDA_CHECK definíciója a következő dián és a teljes kódban.

Üres tömbnél kihagyjuk az indítást. A végrehajtás utáni szinkronizáció itt a hibák lokalizálását segíti.

CUDA hibakezelés

```
inline void check(cudaError_t err, const char* file, int line) {  
    if (err != cudaSuccess) {  
        std::fprintf(stderr, "%s:%d: %s\n", file, line,  
                        cudaGetErrorString(err));  
        std::exit(EXIT_FAILURE);  
    }  
}  
  
#define CUDA_CHECK(call) check((call), __FILE__, __LINE__)
```

- API-hívások: ellenőrizzük a visszatérési kódot.
- `cudaGetLastError()`: az indítás után észlelhető hibák.
- `cudaDeviceSynchronize()`: várakozás és a futás közben jelentkező hibák észlelése a kezdő példában.

A makró hiba esetén lezárja a programot. Nagyobb alkalmazásban kivételkezelés és RAII erőforráskezelés célszerű.

Eredmények visszamásolása és memória felszabadítása

```
std::vector<double> Result(N);  
  
// Eredmenyek visszamasolasa a host memoriaba  
CUDA_CHECK(cudaMemcpy(Result.data(), buffer_y, N*sizeof(double),  
    cudaMemcpyDeviceToHost));  
  
// GPU memoria felszabaditasa  
CUDA_CHECK(cudaFree(buffer_x));  
CUDA_CHECK(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!

A GPU-eredmény ellenőrzése

Referencia: az eredeti X és Y_0 host tömbökből számolunk.

```
for (int i = 0; i < N; ++i) {  
    double ref = scalar * X[i] + Y0[i];  
    double err = std::abs(Result[i] - ref);  
    double tol = atol + rtol * std::abs(ref);  
    if (!std::isfinite(Result[i]) || err > tol) {  
        // Hibás eredmény: index es elteres kiirasa  
    }  
}
```

- A tolerancia a típushoz, skálához és algoritmushoz igazodik.
- FMA és eltérő műveleti sorrend kerekítési eltérést okozhat.
- A teljes kód minden elemet ellenőriz, a NaN/Inf értékeket is kiszűri.

A `daxpy.cu` önálló, teljes példa. A diák részletei ugyanennek a folyamatnak az egyes lépéseit szemléltetik.

Kernelidő mérése CUDA eseményekkel

```
cudaEvent_t start, stop;
CUDA_CHECK(cudaEventCreate(&start));
CUDA_CHECK(cudaEventCreate(&stop));
CUDA_CHECK(cudaEventRecord(start));
daxpy<<<blocks, threads>>>(scalar, buffer_x, buffer_y, N);
CUDA_CHECK(cudaGetLastError());
CUDA_CHECK(cudaEventRecord(stop));
CUDA_CHECK(cudaEventSynchronize(stop));
float ms = 0.0f;
CUDA_CHECK(cudaEventElapsedTime(&ms, start, stop));
CUDA_CHECK(cudaEventDestroy(start));
CUDA_CHECK(cudaEventDestroy(stop));
```

Mindkét esemény és a kernel ugyanabban a streamben van. A mérés előtt bemelegítünk és visszaállítjuk a módosított y bemenetet.

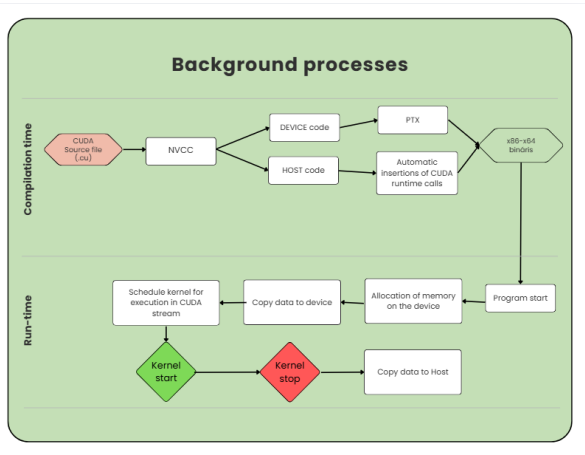
Kernelidő és teljes futási idő

$$t_{\text{teljes}} = t_{\text{H2D}} + t_{\text{kernel}} + t_{\text{D2H}} + t_{\text{egyéb}}.$$

- Host oldali faliórát a másolások elé és a befejezett visszamásolás után teszünk. Pontosan leírjuk, mit foglal magába.
- Az első futás inicializálást és esetleg JIT fordítást is tartalmazhat.
- Több mérés mediánját közöljük, a bemenetet minden ismétlés előtt helyreállítjuk.
- A `daxpy.cu` egy bemelegítés utáni mintát mér. A foglalás, bemelegítés és CPU-s ellenőrzés nincs a teljes időben.

Gyorsulás: $S = t_{\text{CPU}}/t_{\text{GPU}}$. Azonos feladatot és pontosságot hasonlítsunk össze, és nevezzük meg, hogy a GPU-idő tartalmazza-e az adatmozgatást.

Mi zajlik a háttérben



CUDA vs. OpenCL összehasonlítás

Szempont	CUDA	OpenCL
Ökoszisztéma	NVIDIA platform	Khronos nyílt szabvány
Eszközök	NVIDIA GPU	Az implementáció által támogatott CPU, GPU vagy más gyorsító
Kernelnyelv	Itt CUDA C++	Tipikusan OpenCL C
Forrásszervezés	Host/device lehet közös vagy külön fájlban	A kernelforrás lehet külön fájl vagy beágyazott szöveg
Fordítás	nvcc, host fordító, szükség szerint PTX JIT	Forrás, támogatott köztes forma vagy eszközbináris betöltése
A hardveres és nyelvi képességeket mindig le kell kérdezni. Forrás: OpenCL specifikáció.		

CUDA vs. OpenCL: fontos szempontok

Szempont	CUDA Runtime	OpenCL
Inicializálás	A runtime több részletet kezel	Platform, eszköz, kontextus kiválasztása
Memória	Explicit foglalás/másolás, más modellek is elérhetők	Bufferek és képek, megosztott memória képességfüggően
Végrehajtási sor	Default vagy explicit stream	Explicit command queue
Sorrend	Egy stream műveletei sorrendben	In-order vagy támogatott out-of-order sor
Függőségek	Stream, esemény, szinkronizáció	Események és várakozási listák

A külön sorok közötti függőségeket explicit módon kell kifejezni. A párhuzamos végrehajtás lehetőség, nem garancia.

CUDA / OpenCL szótár

CUDA fogalom	OpenCL megfelelő
Grid	NDRange
Thread block	Work-group
Thread	Work-item
blockIdx.x	get_group_id(0)
threadIdx.x	get_local_id(0)
blockDim.x	get_local_size(0)
Számított globális index	get_global_id(0)
Shared memory	Local address space
Szálprivát változó	Private address space

Az indexmegfeleltetés itt 1D, egyenletes csoportméretű, nulla OpenCL globális eltolású indításra értendő. A private memória nem garantáltan fizikai regiszter.

CUDA programozási jó tanácsok

Teljesítmény tippek

- Minimalizáld a host–device másolásokat
- Shared memóriát akkor használj, ha adat-újrahasználatot vagy jobb hozzáférést tesz lehetővé
- Egymáshoz közeli szálak egymáshoz közeli címeket olvassanak
- A blokk méretét mérés alapján válaszd meg
- Csökkentsd a warpon belüli hosszú, eltérő végrehajtási ágakat
- Kerüld a felesleges kernelindításokat

Hibakeresés és best practice

- Használd a compute-sanitizer-et memóriahibákhoz
- Profilozz Nsight Systems és Nsight Compute segítségével
- Legyen világos a host és device kód szerepe
- Kerüld a túl sok szinkronizációt
- Olvasható, moduláris kód → könnyebb optimalizálni

A DAXPY teljesítményének becslése

Elemenként két `double` olvasás és egy írás: nominálisan 24 bájt. A szorzás és összeadás 2 FLOP, FMA esetén is.

$$I = \frac{2}{24} = \frac{1}{12} \text{ FLOP/byte}, \quad B_{\text{eff}} = \frac{24N}{t_{\text{kernel}}}.$$

- Nagy, a cache-en túlnyúló vektoroknál gyakran a memória sávszélessége korlátoz.
- A szomszédos szálak egymás melletti elemeket érjenek el (coalesced hozzáférés).
- Az effektív bájtszám modell, nem a tényleges DRAM-forgalom mérése. Kis adatoknál cache és indítási költség is meghatározó lehet.
- A DAXPY nem használja újra a bemenetet blokkon belül: shared memóriába másolása önmagában nem ígér gyorsulást.

Szinkronizáció és adatverseny

- A DAXPY-ben minden szál külön y_i -t ír, így nincs írási verseny.
- Közös összegre végzett `sum += value` több szálból adatversenyt okozhat. Redukció vagy megfelelő atomi művelet kell.
- A `__syncthreads()` blokkhatókörű korlát. Blokkon belüli feltételes használata csak blokkegységes feltétellel helyes.
- A `__syncthreads()` nem szinkronizálja az egész gridet.
- A kezdő példában az egymást követő kerneleket ugyanabba a streambe tesszük, ez biztosítja a műveleti sorrendet.

A streamen belüli sorrend nem jelenti azt, hogy a host minden indítás után megvárja a GPU-t.

Önálló feladatok és ellenőrző kérdések

- 1 Próbáld ki: $N = 1, 255, 256, 257, 1000$. Hány szál marad munka nélkül?
- 2 Állítsd a blokkméretet 128-ra, 256-ra és 512-re. Ellenőrizd az eredményt, majd ismételt mérésekkel hasonlítsd össze az időket.
- 3 Készíts float változatot. Hogyan változik a tolerancia, a nominális adatforgalom és az idő?
- 4 Magyarázd meg, miért lehet egy 5 elemű feladat CPU-n gyorsabb!

Beszámoló

GPU, CC, Toolkit, N , blokkméret, pontosságellenőrzés, kernelidő és teljes idő. Mérési eredmény nélkül ne állíts gyorsulást.

Javasolt következő témák

- **2D indexelés:** mátrixösszeadás, sorfolytonos tárolás, külön sor- és oszlophatár-ellenőrzés.
- **Redukció:** párhuzamos összegzés, shared memória, szinkronizáció és lebegőpontos összeadási sorrend.
- **Adatmozgatás átlapolása:** streamek, események, pinned host memória. Az átlapolás hardver- és függőségfüggő.
- **Könyvtári megoldás:** az oktatási DAXPY összevetése a cuBLAS cublasDaxpy műveletével.

Az első gyakorlaton a helyes DAXPY és az időmérés legyen a cél. A redukció és a több stream külön gyakorlatnak is megfelel.

Dokumentáció és további olvasás

- CUDA Programming Guide: végrehajtás, memória, szinkronizáció.
- CUDA Best Practices Guide: mérés, adatmozgatás, hozzáférési minták.
- Ampere Tuning Guide: CC 8.0 és 8.6 erőforrásai.
- Compute Sanitizer: memória- és szinkronizációs hibák.
- cuBLAS: SAXPY/DAXPY definíció és könyvtári implementáció.
- OpenCL specifikáció: végrehajtási és memóriamodell.

Ellenőrzés: 2026. szeptember 10. A dokumentáció verziója és a gépen telepített Toolkit verziója eltérhet.