

CUDA programozás

2. gyakorlat: memória, adathozzáférés és csempézés

Balázs Endre Szigeti

Algoritmusok és Alkalmazásaik Tanszék
Informatikai Kar, Eötvös Loránd Tudományegyetem

2026. október 8.

Célkitűzés és tanulási út

- Az alaprészt végére önállóan megírható és ellenőrizhető egy határhelyes CUDA-kernel.
- RGB-feldolgozás: kétdimenziós indexelés, csatornasorrend, sorhossz.
- Memória: regiszter, local, shared, global, cache és összevont hozzáférés.
- Mátrixszorzás: adatújrahasználat, csempézés, két blokkszintű barrier.
- Külön választható haladó rész: bankütközés, aszinkronitás, Tensor Core és profilozás.

Programozási javaslat

Az alapgyakorlat kódjai: `grayscale.cu`, `matrix.cu`. A diák kernelrészletei ugyanabból az `examples/kernels.cuh` fájlból származnak.

Fejlesztőkörnyezet és hordozhatóság

- A példák CUDA C++17-et használnak. A toolkithez támogatott host fordító és a GPU-hoz megfelelő NVIDIA-driver szükséges.
- A toolkit verziója és a GPU compute capability értéke két külön adat.
- A fordítás célarchitektúráját a tényleges géphez kell választani. Az új toolkit nem feltétlenül támogat minden régi GPU-t.
- A `nvidia-smi` CUDA-verziómezője a driver által támogatott verziót jelzi, nem bizonyítja egy toolkit telepítését.

```
nvcc --version
nvidia-smi
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -DCMAKE_CUDA_ARCHITECTURES=native
cmake --build build --config Release
```

A `native` CMake 3.24-től használható, helyi GPU-val. Más gépre explicit architektúralista kell.

Forrás: [6], [7]. A kattintható hivatkozások az előadás végén.

Szálak, blokkok és indexelés

- A kernel egy grid blokkjait indítja. A blokk szálai közösen használhatnak shared memóriát.
- A warp 32 szálból áll. A blokkon belüli lineáris sorrendben az x index változik leggyorsabban.
- A `block(32,8)` egy warpot egy sorba helyez. A `block(16,16)` warpja két, 16 elemű sort fed le.
- A blokkok futási sorrendjére és egyidejű rezidenciájára általános kernelben nem építhetünk.

```
int x = blockIdx.x * blockDim.x + threadIdx.x;  
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

Forrás: [1a], [1b]. A kattintható hivatkozások az előadás végén.

RGB-adatok és szinterek

- Az RGB8 tárolás egy pixelhez három, 0–255 közötti kódértéket rendel: R, G és B.
- A memóriaformátum lehet RGB, BGR vagy RGBA. Ezt a képbetöltő API szerződése dönti el.
- Az sRGB és az Adobe RGB eltérő alapszíneket és átviteli függvényt használ. A konverzió bemenetét előbb azonosítani kell.
- A CIE xy színességi ábrán $x = X/(X + Y + Z)$, $y = Y/(X + Y + Z)$. Ezek nem közvetlen RGB-csatornasúlyok.
- A színességi háromszög a gamutot szemlélteti. A világosságot az xy koordináták önmagukban nem adják meg.

Forrás: [5]. A kattintható hivatkozások az előadás végén.

Szürkeárnyaltos kép



A példakép az eredeti tananyag szemléltetése. A kimenet megjelenése függ a választott színtértől és konverziótól.

Gyors szürkeárnyalat és fizikai luminancia

Gyors, kódértékeken végzett közelítés

$$Y_8^{\text{approx}} = \left\lfloor \frac{77R_8 + 150G_8^{\text{in}} + 29B_8 + 128}{256} \right\rfloor.$$

A súlyok a 0,299, 0,587, 0,114 luma-súlyok egész közelítései. A fekete 0, a fehér 255 marad. Ez nem lineáris fényintenzitás.

Kolorimetriai sRGB-feldolgozás

$$Y = 0,2126R_{\text{lin}} + 0,7152G_{\text{lin}} + 0,0722B_{\text{lin}}.$$

Előbb az sRGB átviteli függvényének inverze kell. sRGB-kódolt szürke képhez Y -t vissza is kell kódolni. A részletek a haladó blokkban szerepelnek.

Forrás: [5]. A kattintható hivatkozások az előadás végén.

RGB-kernel helyes indexeléssel

```
__global__ void grayApprox(const unsigned char* rgb,
                           unsigned char* gray,
                           int width, int height,
                           std::size_t rgbPitch,
                           std::size_t grayPitch) {
    const int x = blockIdx.x * blockDim.x + threadIdx.x;
    const int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= width || y >= height) return;
    const auto* p = rgb + std::size_t(y)*rgbPitch + 3*x;
    // Integer luma approximation: weights sum to 256.
    const unsigned value = 77u*p[0] + 150u*p[1] + 29u*p[2];
    gray[std::size_t(y)*grayPitch + x] = (value + 128u) >> 8;
}
```

rgbPitch és grayPitch: bájt/sor. Folytonos RGB8 képhez $3 \cdot \text{width}$, illetve width . A bemenet és a kimenet nem fedheti egymást.

Képfeldolgozási programozási tanácsok

- RGB esetén a csatornaeltolások: 0, 1, 2. Az eredeti 0, 2, 3 indexelés a következő pixelbe is beleolvasott.
- A pitch bájttban értendő. Az igazított sorhossz eltérhet a hasznos pixelszélességtől.
- RGB8 esetén az egyes csatornaolvasások 3 bájtos lépésűek. A tényleges tranzakciókat a warp teljes címmintázata határozza meg.
- RGBA8/uchar4 vagy síkonkénti tárolás javíthat bizonyos mintákon, de az átalakítás és a plusz bájt költségét mérni kell.
- Egy egyszerű pontonkénti konverzióban nincs szálak közötti újrahasználat. Shared memória beiktatása önmagában nem előny.

Programozási javaslat

Tesztelj 1 pixelt, keskeny képet, 32-vel nem osztható szélességet és kitöltött sorvégeket. A mellékelt példa a padding sértetlenségét is ellenőrzi.

CUDA-memóriák: láthatóság és élettartam

Típus	Példa	Láthatóság	Élettartam / megjegyzés
Regiszter	lokális skalár	szál	fordító rendeli hozzá, szál végrehajtása
Local memory	lokális tömb, spill	szál	eszközmemória, cache-elhető
Shared memory	<code>__shared__ float s;</code>	blokk	blokk végrehajtása
Global memory	<code>cudaMalloc</code> , device változó	eszköztől elérhető	allokációig / CUDA-contextig
Constant memory	<code>__constant__ float c;</code>	eszköztől olvasható	context, host API-val feltöltve

- Egy automatikus változó *nem garantáltan* regiszter. A fordító optimalizálhat vagy local memóriába helyezhet adatot.
- A szálankénti tömb sem szükségképpen local: statikusan ismert indexek mellett regiszterekre bontható.

Forrás: [1a], [1c]. A kattintható hivatkozások az előadás végén.

Cache, shared és local memória

- A regiszter a szál privát állapotának tipikus tárolója. A túl nagy regiszterigény csökkentheti a rezidens warpok számát vagy spillt okozhat.
- A local memória név a szálhoz tartozó címtartományt jelzi. Fizikailag nem egy gyors, külön szálankénti SRAM.
- Az L1/L2 cache automatikusan tart adatokat. Az L2 az eszköz SM-jei között megosztott gyorsítótár.
- A shared memória tartalmát és újrahhasználatát a program vezérli. Az L1/shared kapacitás szervezése architektúrafüggő.
- A global címzés nem jelent minden olvasásnál külön DRAM-hozzáférést: cache-találat és tranzakció-összevonás is lehetséges.

Forrás: [1a], [1b], [2]. A kattintható hivatkozások az előadás végén.

Konstansok és csak olvasható bemenetek

- A `const float*` az adott mutatón keresztüli módosítást tiltja. Nem helyezi az adatot automatikusan constant memóriába.
- A host névtérben deklarált `const` objektum általában nem GPU-globális objektum. A fordítási idejű skalárkonstans és a címezhető tároló külön eset.
- A `__constant__` szimbólumot például `cudaMemcpyToSymbol` tölti fel.
- A constant cache különösen előnyös, amikor egy warp számai ugyanazt a címet olvassák. Sok különböző cím sorosíthatja a kiszolgálást.

```
constexpr int Tile = 16;           // compile-time tile size
__constant__ float weights[3];     // device constant symbol
```

Forrás: [1c]. A kattintható hivatkozások az előadás végén.

Összevont globális hozzáférés: coalescing

- A hardver egy warp egy memóriautasításának címeit tranzakciókba vonja össze.
- Szomszédos szálak szomszédos, megfelelően igazított adatokat érjenek el.
- Tipikus modern esetben 32 szál egymást követő float olvasása 128 hasznos bájt. 32 bájtos szektorokkal és megfelelő igazítással ez négy szektor.
- Eltolt kezdőcím vagy nagy stride több szektort igényelhet. Az elemszám önmagában nem méri a DRAM-forgalmat.

jó kiindulás: $A[\text{base} + \text{lane}]$, vizsgálandó: $A[\text{base} + s \cdot \text{lane}]$.

Programozási javaslat

Sormajor mátrixnál általában az oszlopindexhez rendeljük a `threadIdx.x` értékét.

Forrás: [2]. A kattintható hivatkozások az előadás végén.

Aritmetikai intenzitás és roofline

$$I = \frac{\text{végrehajtott FLOP}}{\text{átvitt bájt}}, \quad P \leq \min(P_{\text{peak}}, B I).$$

- A sávszélesség B és a bájtszám ugyanarra a memóriaszintre vonatkozzon.
- Szemléltető példa: $P_{\text{peak}} = 1600 \text{ GFLOP/s}$, $B = 600 \text{ GB/s}$, $I = 0,25 \text{ FLOP/B}$.
- A memóriaoldali felső korlát $600 \cdot 0,25 = 150 \text{ GFLOP/s}$, a csúcérték 9,375%-a.
- Ez feltételes modell, nem a szürkeárnyaltos kernel mért eredménye.

Programozási javaslat

A sávszélességet helyes hozzáféréssel, az intenzitást adatújrahasználattal javíthatjuk. A szűk keresztmetszetet profilozással ellenőrizzük.

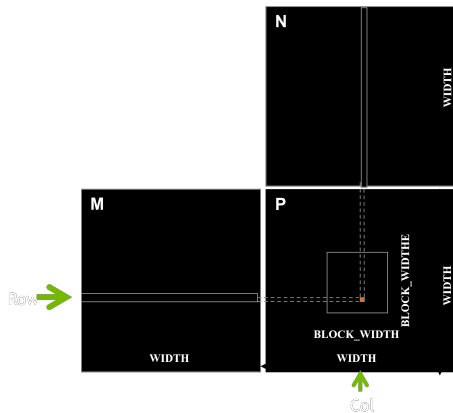
Forrás: [8]. A kattintható hivatkozások az előadás végén.

Mátrixszorzás: jelölések és tárolás

$$A \in \mathbb{R}^{M \times K}, \quad B \in \mathbb{R}^{K \times N}, \quad C = AB \in \mathbb{R}^{M \times N}, \quad C_{ij} = \sum_{k=0}^{K-1} A_{ik} B_{kj}.$$

- Folytonos sormajor tárolás: $A_{ik} \leftrightarrow A[iK + k]$, $B_{kj} \leftrightarrow B[kN + j]$.
- Egy szál egy C_{ij} elemet számít. Az összeg a szál privát változója.
- Az eredeti ábrák M, N, P mátrixnevei itt A, B, C -nek felelnek meg. Itt M, N, K méreteket jelöl.

Mátrixszorzás geometriai képe



Az eredeti ábra M , N , P jelölései rendre a bemeneti A , B és a kimeneti C mátrixot jelentik.

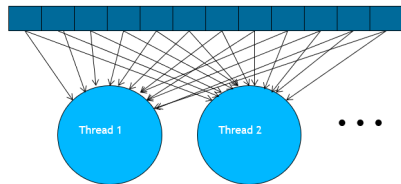
Naiv mátrixszorzó kernel

```
__global__ void matmulNaive(const float* A, const float* B,
                           float* C, int M, int N, int K) {
    const int row = blockIdx.y * blockDim.y + threadIdx.y;
    const int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row >= M || col >= N) return;
    float sum = 0.0f;
    for (int k = 0; k < K; ++k)
        sum = fmaf(A[std::size_t(row)*K + k],
                   B[std::size_t(k)*N + col], sum);
    C[std::size_t(row)*N + col] = sum;
}
```

A `size_t` konverzió a szorzás előtt történik. A host pozitív, érvényes méreteket és elegendő tárterületet biztosít. A `fmaf` egy szorzást és összeadást végez egy kerekítéssel.

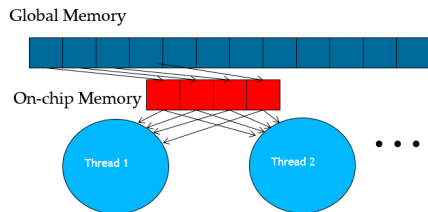
Adatújrahasználat a mátrixszorzásban

- Egy kimeneti sor több eleme ugyanazokat az A -értékeket használja.
- Egy kimeneti oszlop több eleme ugyanazokat a B -értékeket használja.
- A naiv kód forrásszinten ismételt global betöltéseket kér. A cache részben már így is hasznosíthatja az ismétlést.
- A csempézés blokk szinten teszi tudatossá az újrahasználatot.



A csempézés alapötlete

- Egy blokk egy $T \times T$ méretű kimeneti csempét számít.
- A K dimenziót T széles fázisokra bontjuk.
- Fázisonként egy A - és egy B -csempét töltünk global memóriából shared memóriába.
- A szálak közösen használják fel a betöltött elemeket.



Csempézés: a kooperatív lépések

- ➊ Minden szál kijelöli a saját kimeneti sorát és oszlopát.
- ➋ Egy A - és egy B -elemet tölt a blokk shared tömbjeibe.
- ➌ A blokk megvárja a betöltések végét.
- ➍ A szálak a két csempéből saját részösszeget képeznek.
- ➎ A blokk megvárja az olvasások végét, majd új csempét tölthet.
- ➏ Az utolsó fázis után az érvényes szálak kiírják a kimenetet.

Programozási javaslat

A global memória már device memória. Itt a lényeg a *globalból sharedbe* történő betöltés.

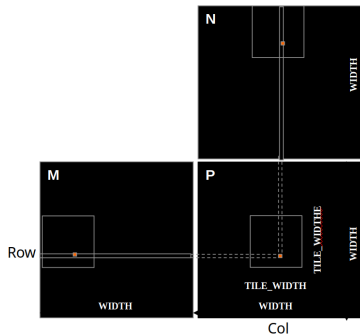
Betöltés az első fázisban

A $t = 0$ fázisban a blokk a K tengely első csempéjével dolgozik.

$$\text{row} = b_y T + t_y,$$

$$\text{col} = b_x T + t_x.$$

A szál betölti $A[\text{row}, t_x]$ és $B[t_y, \text{col}]$ értékét.



Betöltés a t -edik fázisban

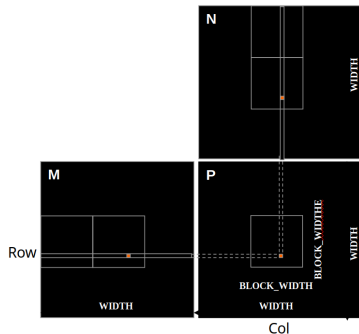
A forráskoordináták:

$$A[\text{row}, tT + t_x], \quad B[tT + t_y, \text{col}].$$

A sormajor lineáris indexek:

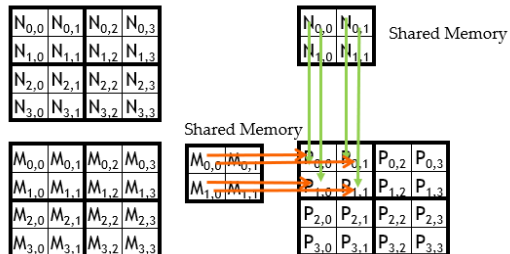
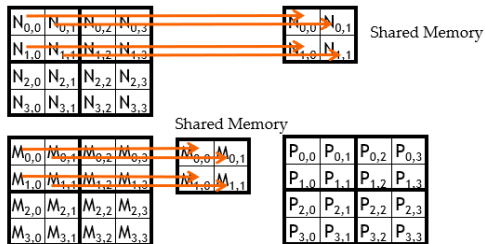
$$\begin{aligned} a &= \text{row } K + tT + t_x, \\ b &= (tT + t_y)N + \text{col}. \end{aligned}$$

Az A sorhossza K , a B sorhossza N .



Két fázis szemléltetése

Az eredeti ábrák színeezése a blokk által betöltött bemeneti csempéket és a számított kimeneti tartományt szemlélteti.



A két barrier külön feladata

- **Az első `__syncthreads()`:** minden shared elem elkészül, mielőtt bármely szál felhasználja.
- **A második `__syncthreads()`:** minden szál befejezi az olvasást, mielőtt a következő fázis felülírja a cempét.
- A barrier blokkon belüli együttműködést és a hozzá tartozó memória-láthatóságot biztosít.
- A helyesség nem függhet a warpok pillanatnyi ütemezési sebességétől.

Programozási javaslat

Ebben a kernelben minden indított szál végigmegy ugyanazon csempecikluson és mindkét barrieren. Nincs szélső szálakra alkalmazott korai visszatérés.

Forrás: [1a], [1c]. A kattintható hivatkozások az előadás végén.

Részleges csempék és határellenőrzés

$$n_{\text{tiles}} = \left\lceil \frac{K}{T} \right\rceil, \quad \text{grid} = \left(\left\lceil \frac{N}{T} \right\rceil, \left\lceil \frac{M}{T} \right\rceil \right).$$

- A betöltése: $\text{row} < M$ és $tT + t_x < K$.
- B betöltése: $tT + t_y < K$ és $\text{col} < N$.
- Tartományon kívüli bemenet helyére minden fázisban nullát írunk a shared tömbbe.
- A kimenetet csak $\text{row} < M$ és $\text{col} < N$ mellett írjuk ki.
- Egy érvénytelen kimeneti elemhez tartozó szál is tölthet a többiek számára szükséges bemenetet.

Programozási javaslat

A pozitív egész K esetén $(K-1)/T+1$ elkerüli a $K+T-1$ összeadás túlcsoordulását.

Teljes csempézett kernel: deklarációk

```
template<int T>
__global__ void matmulTiled(const float* __restrict__ A,
                           const float* __restrict__ B,
                           float* __restrict__ C,
                           int M, int N, int K) {
    static_assert(T > 0 && T <= 32);
    __shared__ float As[T][T], Bs[T][T];
    const int tx = threadIdx.x, ty = threadIdx.y;
    const int row = blockIdx.y * T + ty;
    const int col = blockIdx.x * T + tx;
    float sum = 0.0f;
```

Folytatás a következő dián. Indítási szerződés: $\text{block}=(T,T,1)$, pozitív méretek, nem átfedő pufferek. A `__restrict__` ezt az aliasolási ígéretet közli a fordítóval.

Teljes csempézett kernel: fázisok és kiírás

```
for (int t = 0; t < (K - 1) / T + 1; ++t) {
    const int ak = t*T + tx, bk = t*T + ty;
    As[ty][tx] = (row < M && ak < K)
        ? A[std::size_t(row)*K + ak] : 0.0f;
    Bs[ty][tx] = (bk < K && col < N)
        ? B[std::size_t(bk)*N + col] : 0.0f;
    __syncthreads(); // all tile data ready
    for (int k = 0; k < T; ++k)
        sum = fmaf(As[ty][k], Bs[k][tx], sum);
    __syncthreads(); // all tile readers finished
}
if (row < M && col < N)
    C[std::size_t(row)*N + col] = sum;
}
```

A kernel indítása és a hibák helye

```
constexpr int T = 16;
dim3 block(T, T);
dim3 grid((N-1)/T+1, (M-1)/T+1);
matmulTiled<T><<<grid, block>>>(dA, dB, dC, M, N, K);
CUDA_CHECK(cudaGetLastError());
CUDA_CHECK(cudaDeviceSynchronize());
```

- A CUDA_CHECK minden CUDA-hibakódot ellenőriz. Definíció: `examples/cuda_utils.cuh`.
- A közvetlen lekérdezés az indítás hibáit, a szinkronizálás a végrehajtás aszinkron hibáit is felszínre hozhatja.
- A teljes eszköz szinkronizálása tanuláshoz egyszerű. Pipeline-ban célzott stream- vagy eseményvárakozás használható.

Forrás: [2], [9]. A kattintható hivatkozások az előadás végén.

Csempeméret: számszerű költségek

Egy teljes fázis / blokk	$T = 16$	$T = 32$
Szálak száma: T^2	256	1024
Betöltött float: $2T^2$	512	2048
Shared tárigény: $8T^2$ bájt	2 KiB	8 KiB
Hasznos művelet: $2T^3$ FLOP	8192	65 536
Bemeneti intenzitás: $T/4$ FLOP/B	4	8

- A modell teljes csempéket és FP32-t feltételez. Egy FMA két FLOP.
- Az intenzitás itt a betöltött bemenetekre vonatkozik, a kimeneti írás és a cache hatása nélkül.
- A 32×32 blokk 1024 szála miatt a hardver- és kernelkorlátok különösen fontosak.

Occupancy és erőforráskorlátok

- Az occupancy a rezidens aktív warpok számának aránya az SM hardveres maximumához.
- A rezidens blokkok számát együtt korlátozza a shared memória, a regiszterigény, a szál- és warplimit, illetve a blokklimit.
- A shared kapacitás elosztása a blokk tárigényével csak egy felső korlát. A korábbi 16 KiB-os példa nem általános modern GPU-modell.
- A nagyobb csempe csökkentheti a párhuzamosan rezidens blokkok számát. A magasabb occupancy sem garantál nagyobb sebességet.

```
cudaOccupancyMaxActiveBlocksPerMultiprocessor(  
    &bblocksPerSM, matmulTiled<16>, 16*16, 0);
```

A záró 0 a dinamikus shared tárigény. A statikus shared memóriát az API figyelembe veszi.

Forrás: [1b], [2]. A kattintható hivatkozások az előadás végén.

Numerikus helyesség ellenőrzése

$$|C_{ij}^{\text{GPU}} - C_{ij}^{\text{ref}}| \leq \text{atol} + \text{rtol} |C_{ij}^{\text{ref}}|.$$

- A CPU-referencia a tényleges FP32 bemeneteket FP64 összegzéssel dolgozza fel.
- A lebegőpontos összeadás nem asszociatív. FMA, átrendezés és eltérő pontosság megváltoztathatja a kerekítést.
- Külön ellenőrizzük a NaN/Inf értékeket. A toleranciát a méret, skála és pontossági igény alapján választjuk.
- A mellékelt, korlátos bemeneteken $\text{atol} = \text{rtol} = 10^{-4}$ oktatási kiindulás, nem általános hibakorlát.

Programozási javaslat

Méretepéldák (M, N, K) sorrendben: $(1, 1, 1)$, $(7, 13, 5)$, $(17, 31, 19)$, $(129, 127, 131)$. A négyzetes teszt önmagában kevés.

CUDA-eseményekkel végzett időmérés

```
CUDA_CHECK(cudaEventRecord(start));  
for (int i = 0; i < repeats; ++i) {  
    matmultiled<T><<<grid, block>>>(dA, dB, dC, M, N, K);  
    CUDA_CHECK(cudaGetLastError());  
}  
CUDA_CHECK(cudaEventRecord(stop));  
CUDA_CHECK(cudaEventSynchronize(stop));  
CUDA_CHECK(cudaEventElapsedTime(&elapsedMs, start, stop));
```

- A létrehozott események ugyanabba a streambe kerüljenek, mint a mért munka. Előtte bemelegítés szükséges.
- Az átlagos idő $t_{\text{ms}} = \text{elapsedMs} / \text{repeats}$, a sebesség $2MNK / (10^6 t_{\text{ms}})$ GFLOP/s.
- Külön mérjük a kernel idejét és a teljes folyamatot, az adatmásolásokkal együtt. Kis kerneleknél az indítások közti rések is számíthatnak.

Forrás: [9]. A kattintható hivatkozások az előadás végén.

Memória- és szinkronizációs hibakeresés

```
compute-sanitizer --tool memcheck ./build/matrix  
compute-sanitizer --tool racecheck ./build/matrix  
compute-sanitizer --tool synccheck ./build/matrix  
compute-sanitizer --tool initcheck ./build/matrix
```

- **memcheck**: többek között hibás címzés és igazítás.
- **racecheck**: shared memória hozzáférési veszélyei. Nem általános bizonyítás minden global adattrace hiányára.
- **synccheck**: hibás szinkronizációs használat.
- **initcheck**: inicializálatlan global memória olvasása. Érdemes memcheck után futtatni.

A `-lineinfo` forrásorokat ad a diagnosztikához. A sanitizer alatti idő nem benchmark.

Forrás: [3]. A kattintható hivatkozások az előadás végén.

Profilozási munkafolyamat

- **Nsight Systems:** a teljes futás idővonala, host-költségek, másolások, streamenkénti munka és átfedés.
- **Nsight Compute:** kiválasztott kernel, memóriaforgalom, occupancy, regiszterigény és roofline.
- Egyszerre egy változtatást hasonlítsunk össze ugyanazon bemeneten és pontossági beállításokkal.
- A cache-állapot, GPU-órajel, energiaellátás és más GPU-folyamatok is befolyásolják az eredményt.

```
nsys profile --trace=cuda,nvtx -o timeline ./build/matrix  
ncu --set full --launch-count 1 ./build/matrix
```

Ezek induló parancsok. Az ncu replayt és mérési többletköltséget okozhat. A kívánt kernelre/indításra szűrjünk.

Forrás: [8], [10]. A kattintható hivatkozások az előadás végén.

Modern C++ a CUDA körül

- Host oldalon `std::vector`, eszközoldali erőforrásokhoz RAI: felszabadítás kivételes kilépéskor is.
- Az erőforrást birtokló osztály másolása legyen tiltott vagy világosan definiált. A destruktor ne dobjon kivételt.
- A méret- és bájt számokhoz `std::size_t`, a szorzás előtt megfelelő konverzió és allokációs túlcsordulás-ellenőrzés kell.
- Olvasható bemenetekhez `const`. A `__restrict__` csak valóban teljesülő aliasolási feltételekkel használható.
- Az alap példa nem használ `-use_fast_math` kapcsolót. Ezt külön numerikus és teljesítményvizsgálat után érdemes kipróbálni.

Programozási javaslat

A mellékelt `DeviceBuffer` és `CudaEvent` egyszerű oktatási RAI-osztályok, nem teljes általános GPU-könyvtár.

Alapfeladatok a gyakorlatra

- ➊ A grayscale példán magyarázd meg az RGB-indexeket és a két pitch szerepét. Adj hozzá BGR-bemenetet.
- ➋ A `matrix` programban hasonlítsd össze a naiv és csempézett kernelt részleges csempéknél is.
- ➌ Próbáld ki a $T = 8, 16, 32$ értékeket, ahol a GPU és a kernel engedi. A block és a sablonparaméter maradjon összhangban.
- ➍ Rögzítsd a GPU-t, toolkit/driver verziót, méreteket, toleranciát, futási időt és a mérés ismétlésszámát.
- ➎ Indokold, hogy melyik változtatás javította az adatelérést, és melyik az adatújrahasználatot.

Programozási javaslat

Előbb helyesség és hibakeresés, utána összehasonlítható mérés. A feladat beadásának része az eredmények magyarázata.

Haladó: sRGB linearizálása

Normalizált $c \in [0, 1]$ sRGB-kódértékre:

$$L(c) = \begin{cases} c/12,92, & c \leq 0,04045, \\ ((c + 0,055)/1,055)^{2,4}, & c > 0,04045. \end{cases}$$

A lineáris luminancia $Y = 0,2126L(R) + 0,7152L(G) + 0,0722L(B)$. A kimeneti sRGB-szürke kódérték:

$$S(Y) = \begin{cases} 12,92Y, & Y \leq 0,0031308, \\ 1,055Y^{1/2,4} - 0,055, & Y > 0,0031308. \end{cases}$$

Programozási javaslat

Feladat: a háromféle módszert (egész luma, lineáris luminancia, visszakódolt szürke) hasonlítsd össze telített alapszíneken. A 256 elemű linearizáló LUT helyét mérésel válaszd ki.

Forrás: [5]. A kattintható hivatkozások az előadás végén.

Haladó: shared memóriabankok

- A tipikus modern, 32 bankos modellen egymást követő 32 bites szavak egymást követő bankokhoz tartoznak.
- Különböző címek ugyanabban a bankban, egy warp azonos utasításán belül, ütközést és sorosítást okozhatnak.
- Ugyanannak a szónak az olvasása broadcast lehet. Ez nem azonos a külön címek ütközésével.

$$\text{bank}(\text{tile}[r][c]) = (r \cdot \text{stride} + c) \bmod 32.$$

- `float tile[32][32]` oszlopolvasásánál a stride 32 bankütközést okozhat.
- `float tile[32][33]` esetén a padding megváltoztatja a bankleképezést.

Programozási javaslat

A transzponálás klasszikus példa a padding hasznára. Ne adj minden shared tömbhöz automatikusan egy oszlopot: a konkrét hozzáférést vizsgáld.

Haladó: warp-szintű együttműködés

- Az önálló szálütemezés miatt ne feltételezz implicit warp-szinkronitást.
- A `__syncwarp(mask)` a megadott résztvevők együttműködését rendezi. Más warpokat nem vár meg.
- A `__shfl_down_sync(mask, value, delta)` regiszterértéket cserélhet a warp szálai között.
- A maszkot a valóban részt vevő szálakból kell képezni. A maszkban szereplő, még futó szálaknak megfelelően egyező kollektív hívást kell végrehajtaniuk.
- Egyszerűsítéshez Cooperative Groups, redukcióhoz gyakran CUB használható.

Programozási javaslat

Feladat: warp-redukciót készíts teljes és részleges warpra. A csempézett kernel blokkbarrierjét nem lehet általánosan warpbarrierre cserélni.

Forrás: [1c], [11]. A kattintható hivatkozások az előadás végén.

Haladó: memory model és atomikus műveletek

- Adatrace keletkezhet, ha szálak ugyanazt a helyet rendezetlenül érik el, és legalább az egyik ír.
- Az `atomicAdd` egy adott frissítést tesz atomikussá. Önmagában nem általános barrier más adatok számára.
- A `volatile` nem helyettesíti az atomikus műveletet vagy a szinkronizációt.
- A `fence` memóriaműveletek sorrendjét szabályozza. Nem várja be automatikusan a többi blokkot.
- Haladó kommunikációhoz a `cuda::atomic_ref`, a megfelelő thread scope és az `acquire/release` sorrend együtt értelmezendő.

Programozási javaslat

Grid-szintű fázishatárhoz gyakran két, azonos streambe indított kernel a legegyszerűbb megoldás. Az általános spin-barrier elakadhat, ha a szükséges blokkok még nem futnak.

Forrás: [12]. A kattintható hivatkozások az előadás végén.

Haladó: stream és másolás–számítás átfedés

- Egy stream műveletei sorban követik egymást. Külön streamek között explicit eseményfüggőségre lehet szükség.
- A host–device aszinkron másolások megbízható átfedéséhez jellemzően page-locked (pinned) host memória, megfelelő streamek és hardvertámogatás kell.
- A `cudaMemcpyAsync` név önmagában nem garantál tényleges átfedést. Pageable memória staginget és blokkolást okozhat.
- A pinned puffert csak a másolás befejezése után szabad módosítani vagy felszabadítani.
- Chunkonkénti feldolgozás és dupla pufferek segíthet. Túl kicsi chunkoknál az indítási többletköltség dominálhat.

Programozási javaslat

Feladat: két streammel készíts képfeldolgozó pipeline-t, majd Nsight Systems idővonalon igazold az átfedést.

Forrás: [9]. A kattintható hivatkozások az előadás végén.

Haladó: aszinkron global-shared másolás

- A `cuda::memcpy_async` és `pipeline/barrier` absztrakciók a következő csempe betöltését a jelenlegi számításával fedhetik át.
- A global-shared másolás hardveres gyorsítása `compute capability` 8.0-tól érhető el a megfelelő feltételekkel.
- Az igazítás, méret és memóriatípus a kiválasztott API útvonalát és teljesítményét is befolyásolja.
- Fogasztás előtt meg kell várni a másolást, felülírás előtt pedig az összes olvasót.
- A dupla pufferelés több shared memóriát használ. A nyereség és az occupancy változása együtt mérendő.

Programozási javaslat

Hopper és újabb architektúrákon TMA is elérhető. Ez külön architektúraspecifikus út, nem az alapkernel általános előfeltétele.

Forrás: [13]. A kattintható hivatkozások az előadás végén.

Haladó: több kimeneti elem szálánként

- A regisztercsempézésben egy szál több C elemet halmoz fel.
- A shared memóriából olvasott érték több szorzatban is felhasználható.
- A több független akkumulátor javíthatja az utasításszintű párhuzamosságot.
- A regiszterigény növekedhet, ami spillt vagy kevesebb rezidens blokkot okozhat.

Programozási javaslat

Feladat: szálánként két szomszédos kimeneti oszlopot számító változat. Mérd a regiszterszámot, shared forgalmat, futási időt és numerikus hibát. A blokk kimeneti csempéje ekkor már nem $T \times T$.

Forrás: [2], [8]. A kattintható hivatkozások az előadás végén.

Haladó: cuBLAS, cuBLASLt és Tensor Core

- Éles GEMM-feladathoz a cuBLAS/cuBLASLt erős kiindulási és összehasonlítási alap.
- A hagyományos cuBLAS column-major konvenciót használ. Row-major adatoknál a paraméterezést át kell gondolni, vagy cuBLASLt layoutot kell megadni.
- A Tensor Core mátrixműveletekre specializált. A támogatott bemeneti és akkumulációs típusok architektúrafüggők.
- A TF32, FP16 vagy BF16 megváltoztathatja az eredményt. Az FP32 kimenet önmagában nem bizonyít teljes FP32 számítási pontosságot.
- Az összevetéshez azonos probléma, adatmozgatás és megfelelően dokumentált számítási mód szükséges.

Programozási javaslat

Saját kernel akkor indokolt, ha speciális szerkezet, műveletfúzió vagy mért szűk keresztmetszet ezt alátámasztja.

Forrás: [4]. A kattintható hivatkozások az előadás végén.

Haladó: Unified Memory és allokáció

- A `cudaMallocManaged` egyszerűbb közös címezhetőséget adhat, de az adat elhelyezése és mozgatása továbbra is teljesítménykérdés.
- A laphibák és migráció mérési tuskéket okozhatnak. A támogatott prefetch és advice segíthet a várható elérés jelzésében.
- A támogatás és a CPU/GPU egyidejű hozzáférés szabályai platformfüggők. A hardver attribútumait is ellenőrizni kell.
- A `cudaMallocAsync/cudaFreeAsync` stream-sorrendű pool-allokációt tesz lehetővé támogatott rendszereken.
- A puffer élettartama fedje le minden streambeli felhasználását. Más stream esetén explicit függőségek szükségesek.

Programozási javaslat

A fő példa explicit másolást használ, hogy az adatmozgás látható legyen. Haladó feladatként ezt hasonlítsd össze managed memóriával.

Haladó: CUDA Graphs és műveletfúzió

- Sok rövid, ismétlődő műveletnél a host oldali indítási többletköltség jelentős lehet.
- A CUDA Graph egy függőségi struktúra ismételt indítását teszi lehetővé. Létrehozási költsége is van.
- A fúzió köztes global kiírásokat és indításokat takaríthat meg, de növelheti a regiszterigényt.
- Képfeldolgozási példa: szürkeárnyalat és normalizálás egy kernelben, ha a numerikus és tárolási szerződés ezt engedi.

Programozási javaslat

Kis batchnél mérd külön az egyszeri előkészítést és a sokszor ismételt végrehajtást. Graph és fúzió hasznát a teljes futás idejével igazold.

Forrás: [16]. A kattintható hivatkozások az előadás végén.

Haladó: kutatási jellegű mini-projektek

- ❶ **Shared transzponálás:** padding nélkül és +1 oszloppal. Előre számold ki a bankindexeket, majd mérd.
- ❷ **GEMM-változatok:** naiv, shared csempézett, regisztercsempézett és könyvtári megoldás azonos bemeneteken.
- ❸ **Pipeline:** két stream, pinned pufferek és több chunkméret. A helyességet eseményfüggőségekkel biztosítsd.
- ❹ **Pontosság:** FP32 és támogatott vegyes pontosságú GEMM. Vizsgáld a relatív hibát és az erős kioltású bemeneteket.

Minden projekthez: hipotézis, eszköz- és verzióadatok, reprodukálható parancsok, helyességi kritérium, mérési szórás és a kapott eredmény értelmezése.

Források: programozási modell és ellenőrzés

- [1a] NVIDIA: Writing SIMT Kernels
- [1b] NVIDIA: Compute Capabilities
- [1c] NVIDIA: C/C++ Language Extensions
- [2] CUDA C++ Best Practices Guide
- [3] Compute Sanitizer documentation
- [4] cuBLAS / cuBLASLt documentation
- [5] W3C: CSS Color 4, sRGB és színkonverziók

A dokumentáció ellenőrzésének dátuma: 2026. október 8. Az alapelvek mellett mindig az alkalmazott toolkit és GPU dokumentációja az irányadó.

Források: eszközök és haladó témák

- [6] CUDA Installation Guide for Linux
- [7] CMake: CUDA_ARCHITECTURES
- [8] Nsight Compute Profiling Guide
- [9] NVIDIA: Asynchronous Execution
- [10] Nsight Systems User Guide
- [11] CUDA Core Compute Libraries: CUB, Thrust, libcu++
- [12] CUDA C++ Memory Model
- [13] NVIDIA: Asynchronous Data Copies

Források és a csomag használata

[14] NVIDIA: Unified Memory

[15] NVIDIA: Stream-Ordered Memory Allocator

[16] NVIDIA: CUDA Graphs

- `main.tex`: az előadás szerkeszthető forrása. Fordítás: `latexmk -pdf main.tex`.
- `matrix.cu`, `grayscale.cu`, `examples/`: példák és közös kód.
- `CMakeLists.txt`, `README.md`: fordítás és futtatás.
- `VALTOZASOK.md`: a javítások részletes jegyzéke.
- Az eredeti képek megmaradtak. Az `original/` mappa összehasonlításhoz őrzi az eredeti két forrásfájlt.