2. Gyakorlat

CUDA Programozás

Balázs Endre Szigeti

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

October 17, 2025

Célkitűzés

- A CUDA memória-típusok hatékony használatának elsajátítása párhuzamos programozásban
- A memória-hozzáférés hatékonyságának fontossága
- Regiszterek, megosztott memória, globális memória
- Láthatóság és élettartam
- Csempézett algoritmus megértése

RGB színkép ábrázolása

- A kép minden pixele egy RGB érték
- Egy kép sora a következő formátumú: (r g b) (r g b) ... (r g b)
- Az RGB tartományok nem oszlanak el egyenletesen
- Számos különböző színtér létezik - itt az AdobeRGB színtérhez való konverzió konstansait mutatjuk be
- A függőleges tengely (y érték) és a vízszintes tengely (x érték) a pixel intenzitásának azt a hányadát mutatja, amelyet a G és B komponensekhez kell rendelni
- A maradék hányad $(1 - y - x)$ a piros (R) komponenshez tartozik
- A háromszög az adott színtérben megjeleníthető összes színt tartalmazza

Grayscaling

Grayscaling Algoritmus

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                  unsigned char * rgbImage,
                               int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;
}
```

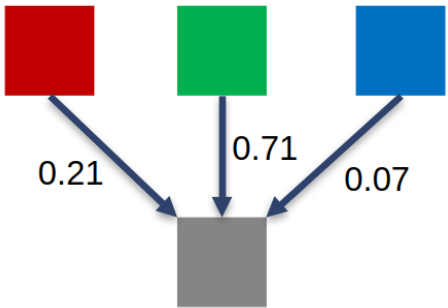
Grayscaling Agoritmus folytatás

```
if (x < width && y < height) {
    // get 1D coordinate for the grayscale image
    int grayOffset = y*width + x;
    // one can think of the RGB image having
    // CHANNEL times columns than the gray scale image
    int rgbOffset = grayOffset*CHANNELS;
    unsigned char r =  rgbImage[rgbOffset      ]; // red value for
       pixel
    unsigned char g = rgbImage[rgbOffset + 2]; // green value for
       pixel
    unsigned char b = rgbImage[rgbOffset + 3]; // blue value for
       pixel
    // perform the rescaling and store it
    // We multiply by floating point constants
    grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
```

Színkalkulációs képlet

- Minden pixelre (r, g, b) a (I, J) pozíción: grayPixel[I, J] $= 0.21 \cdot r + 0.71 \cdot g + 0.07 \cdot b$
- Ez egyszerűen a skaláris szorzat:
  $\langle [r, g, b], [0.21, 0.71, 0.07] \rangle$

- A konstansok az adott bemeneti RGB színtérhez tartoznak



Teljesítmény megfontolások GPU-n

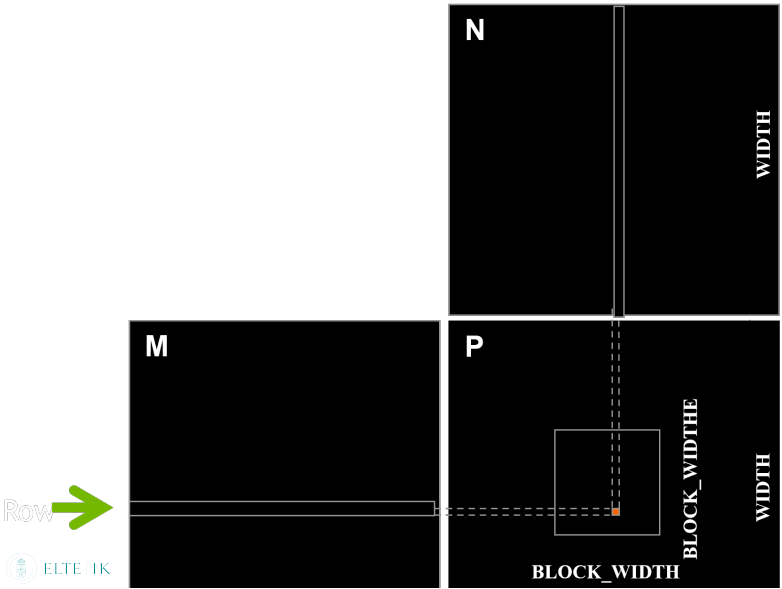
- Minden szál a globális memóriából olvassa az adatokat.
- Egy memória-hozzáférés (4 bájt) minden lebegőpontos összeadáshoz.
- Példa:
  - GPU: 1,600 GFLOPS csúcsteljesítmény, 600 GB/s DRAM sávszélesség
  - 4B/FLOP $\rightarrow$ 6,400 GB/s szükséges a csúcsteljesítményhez
  - 600 GB/s memória-sávszélesség $\rightarrow$ 150 GFLOPS effektív teljesítmény
  - Ez csupán 9,3
- Következtetés: drasztikusan csökkenteni kell a memória-hozzáférések számát.

Mátrixszorzás példa

```
global void MatrixMulKernel(float* M, float* N, float* P, int Width)
    {
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        for (int k = 0; k < Width; ++k) {
            Pvalue += M[Row * Width + k] * N[k * Width + Col];
        }
        P[Row * Width + Col] = Pvalue;
    }
}
```

Mátrixszorzás példa



CUDA memóriák

Változó deklaráció	Memória	Scope	Lifetime
int Localvar	regiszter	szál	szál
`__device__ __shared__ int SharedVar`	megosztott	blokk	blokk
`__device__ int GlobalVar`	globális	grid	app
`__device__ __constant__ int ConstantVar`	konstant	grid	app

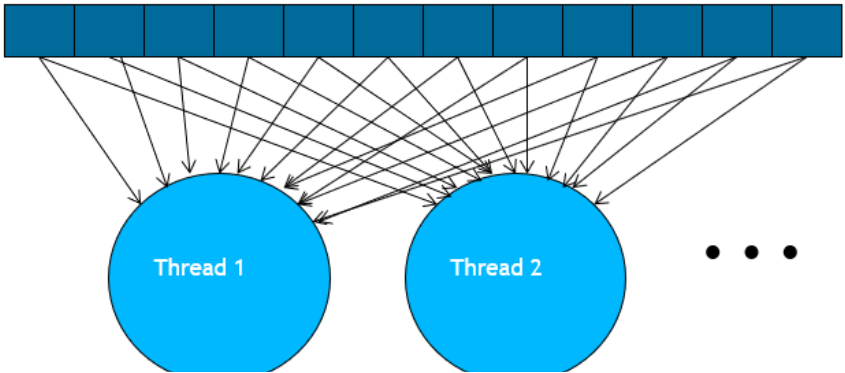
- `__device__` opcionális, ha `__shared__` vagy `__constant__` kulcsszóval együtt használjuk.
- Automatikus (lokális) változók regiszterben tárolódnak.
- Szálankénti tömbök a globális memóriában találhatók.

Megosztott memória a CUDA-ban

- Egy speciális, gyors memória-típus, amely minden SM-ben megtalálható.
- Jelentősen gyorsabb hozzáférés, mint a globális memóriához.
- Hozzáférési tartomány: szálblokkon belül
- Élettartam: a szálblokk végéig tart
- Tipikus használat: ideiglenes (scratchpad) memória

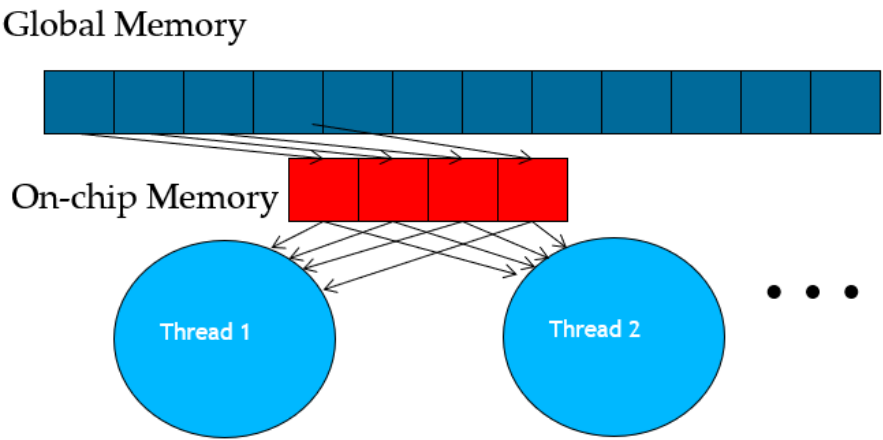
Globális memória-hozzáférés mintázata

- A memória-sávszélesség korlátozó hatásának csökkentése
- A csempézett (tiled) algoritmusok és a szinkronizáció megértése
- A szálak közvetlenül a globális memóriából olvassák az adatokat.
- Ez nagy memóriaforgalmat és alacsony hatékonyságot okoz.



A csempézés alapötlete

- A globális memóriát kisebb **csempékre (tile-okra)** osztjuk.
- A szálak egy időben csak egy (vagy néhány) csempével dolgoznak.
- Az adatok a gyors on-chip memóriába töltődnek.

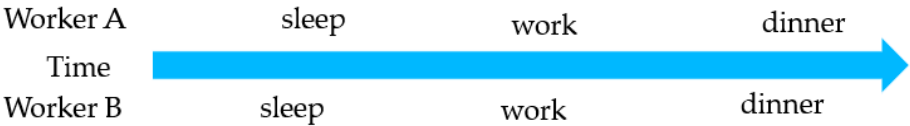


Analógia – sávmegosztás

- A zsúfolt közlekedés hasonló a túlterhelt memóriához.
- A **carpooling** (autómegosztás) csökkenti a járművek számát → gyorsabb haladás.
- A **csempézéshez** hasonlóan csökkenti a memóriaműveletek számát.

Szinkronizáció a csempézés során

- A szálaknak össze kell hangolniuk a munkájukat (barrier szinkronizáció).
- Jó szinkronizáció: hasonló ütemezésű szálak esetén.
- Rossz szinkronizáció: eltérő időzítésű szálak esetén.

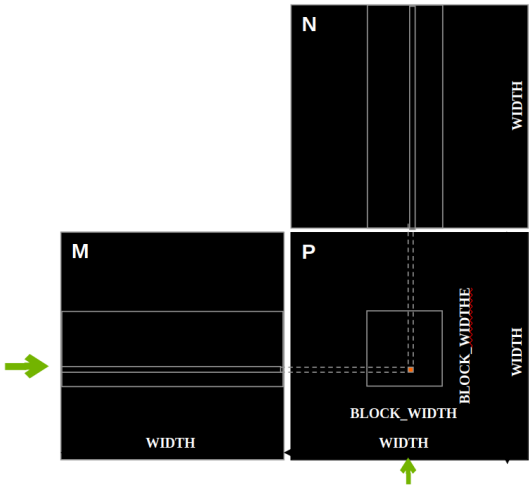


A tiling technika lépései

1. Azonosítsuk azokat a globális memóriarészeket, amelyeket több szál is használ.
2. Töltsük be ezeket a csempéket az on-chip memóriába.
3. Végezzük el a műveleteket szinkronizáltan.
4. Ismételjük a következő csempékre.

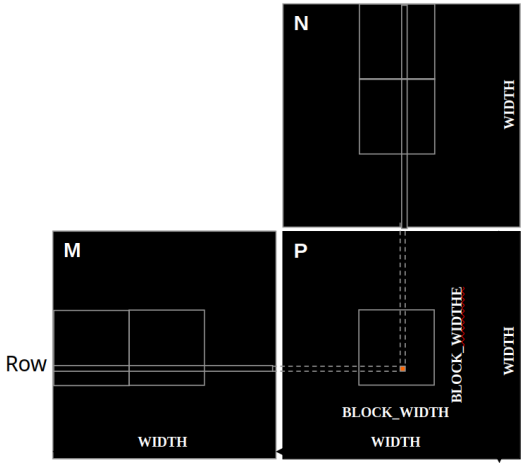
Mátrixszorzás – Adathozzáférési minta

- Minden szál egy M-mátrix sorát és egy N-mátrix oszlopát dolgozza fel.
- Minden szálblokk M és N egy-egy szalagját számolja.



Csempézett mátrixszorzás alapelve

- Az egyes szálak végrehajtása **fázisokra** oszlik.
- Minden fázisban a szálblokk az M és N egy-egy csempéjére összpontosít.
- A csempe mérete: `BLOCK_SIZE` × `BLOCK_SIZE`.



Csempe betöltése

- Minden szál részt vesz a csempe betöltésében.
- Minden szál egy M és egy N elemet tölt be a saját pozíciójának megfelelően.
- A betöltött adatok a **megosztott memóriában** tárolódnak.

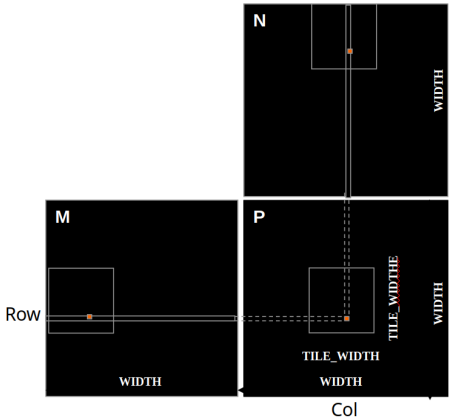
Fázis 0 – Betöltés és használat

Minden szál betölt egy M- és egy N-elemet

- Minden szál a saját pozíciójának megfelelő elemet olvassa be.
- Az elemek ugyanabban a relatív helyzetben vannak, mint a kiszámított P elem.
- A 2D indexelés meghatározza, melyik csempe-részhez fér hozzá a szál.

```
int Row = by * blockDim.y + ty;
int Col = bx * blockDim.x + tx;

// Tile 0:
M[Row][tx];
N[ty][Col];
```



Fázis 1 – Betöltés és használat

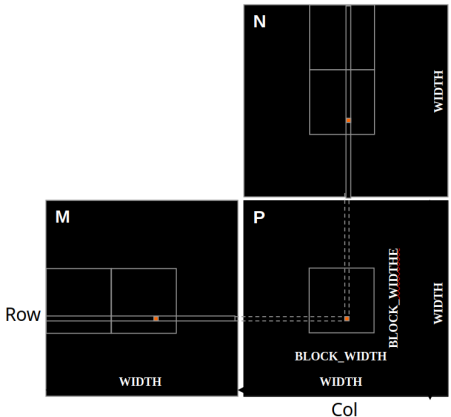
Minden szál betölt egy M- és egy N-elemet

- M és N dinamikusan allokálódik - 1D indexeléssel
- p szekvencia szám

```
2D indexing for accessing Tile 1:
    M[Row][1*TILE_WIDTH + tx]
    N[1*TILE*WIDTH + ty][Col]

1D indexin for:
    M[Row][p*TILE_WIDTH+tx]
    M[Row*Width + p*TILE_WIDTH + tx]

    N[p*TILE_WIDTH+ty][Col]
    N[(p*TILE_WIDTH+ty)*Width + Col]
```



Barrier szinkronizáció

- `__syncthreads()` — az összes szálnak el kell érnie ezt a pontot, mielőtt bármelyik továbblép.
- A fázisok koordinálására használjuk.
- Gondoskodik róla, hogy minden elem betöltődjön az adott fázis elején és feldolgozódjon a végén.
- Tiled mátrixszorzó kernel megírása.
- Csempék betöltése, használata és szinkronizáció.
- Erőforrás-figyelembe vétel (pl. `TILE_WIDTH`).

Csempe (blokkméret) megfontolások

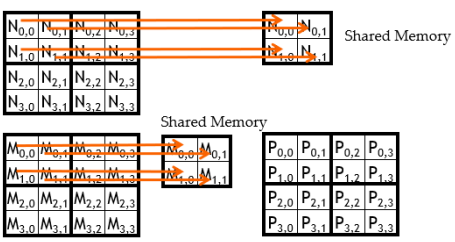
- **TILE_WIDTH = 16**: 256 szál/blokk, 512 float betöltés, 8192 művelet.
- **TILE_WIDTH = 32**: 1024 szál/blokk, 2048 float betöltés, 65 536 művelet.
- Több művelet kevesebb memória-hozzáféréssel: nagyobb hatékonyság.

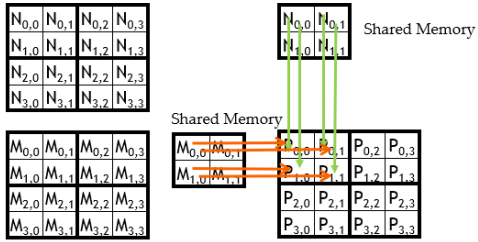
Megosztott memória és szálkezelés

- 16 KB megosztott memória esetén: **TILE_WIDTH=16** → 2 KB/blokk.
- Egyszerre akár 8 blokk is futhat egy SM-en (4 096 párhuzamos betöltés).
- **TILE_WIDTH=32** → 8 KB/blokk → kevesebb blokk futhat.
- A túl sok `__syncthreads()` hívás csökkentheti az aktív szálak számát.

Fázis 1 és 2 a blokk (0,0) esetre

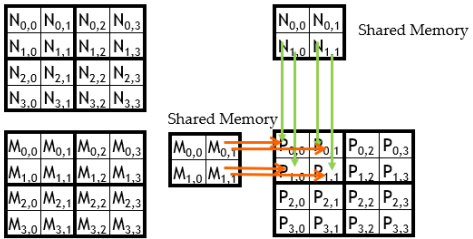
- Példa: Blokk (0,0) első fázisának betöltése.
- A szálak betöltik az első M és N csempét, majd közösen feldolgozzák.

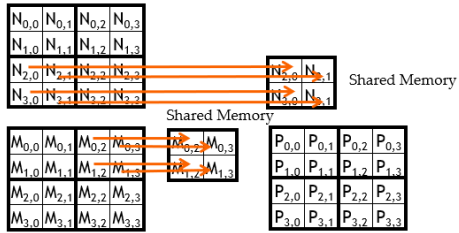




Fázis 2 (0,0) esetre és Fázis (1,1) esetre

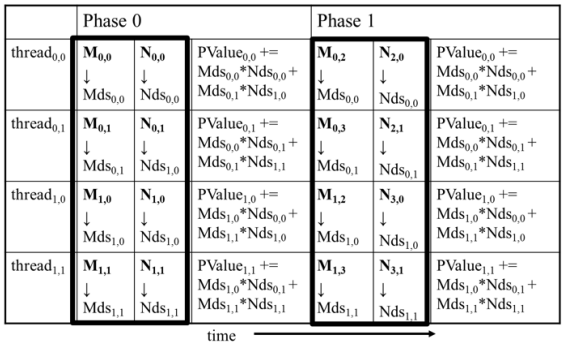
- Következő iterációban a szálak új csempét töltenek be.
- A korábban feldolgozott adatok után a következő M, N csempe következik.





Végrehajtási fázisok – Példa

- A megosztott memória lehetővé teszi, hogy több szál hozzáférjen ugyanahhoz az adatponthoz.
- Minden fázis elején és végén **szinkronizáció** szükséges.



Bemeneti csempék betöltése

2D indexelés példa

```
int Row = by * blockDim.y + ty;
int Col = bx * blockDim.x + tx;

// Tile 0:
M[Row][tx];
N[ty][Col];

// Tile 1:
M[Row][1TILE_WIDTH + tx];
N[1TILE_WIDTH + ty][Col];
```

Tiled mátrixszorzás kernel

CUDA kód

```
global void MatrixMulKernel(float* M, float* N, float* P, int Width)
    {
    float Pvalue = 0;
    for (int p = 0; p < Width / TILE_WIDTH; ++p) {
        ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
        ds_N[ty][tx] = N[(p * TILE_WIDTH + ty) * Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += ds_M[ty][i] * ds_N[i][tx];

        __syncthreads();
        }
    P[Row * Width + Col] = Pvalue;
}
```

Nem négyzetes mátrixok kezelése

A korábban bemutatott csempézett mátrixszorzó kernel csak olyan négyzetes mátrixokra alkalmazható, amelyek dimenziói (Width) a csempe méretének (`TILE_WIDTH`) többszörösei.
A valós alkalmazásokban azonban gyakran előfordulnak tetszőleges méretű mátrixok, amelyek nem illeszkednek pontosan a csempeméret határain belül.

- Az egyik lehetséges megoldás a **kitöltés (padding)**: a sorokat és oszlopokat további elemekkel egészítjük ki, hogy a dimenziók a `TILE_WIDTH` többszörösei legyenek.
- Ez azonban **jelentős memóriatöbbletet és adatátviteli időt** eredményezhet.

Másik megközelítés: olyan algoritmust alkalmazunk, amely közvetlenül képes kezelni a tetszőleges méretü mátrixokat, anélkül, hogy szükség lenne kitöltésre.

Egyszerű” megoldás

Alapelv: Amikor egy szálnak be kell töltenie egy bemeneti elemet, előbb ellenőrizni kell, hogy az index a **érvényes tartományba** esik-e.

- Ha az index **érvényes**, a szál végrehajtja a betöltést.
- Ha az index **érvénytelen**, nem tölt be adatot, hanem **0 értéket** ír a megfelelő helyre.

Indoklás: A nullával való szorzás biztosítja, hogy a *multiply-add* lépés nem befolyásolja az eredmény mátrix megfelelő elemét.
Megjegyzés: A bemeneti elemek betöltéséhez használt feltétel különbözik a P kimeneti elem számítására használt feltételtől. Egy olyan szál is részt vehet a bemeneti csempék betöltésében, amely maga nem számol érvényes P elemet.