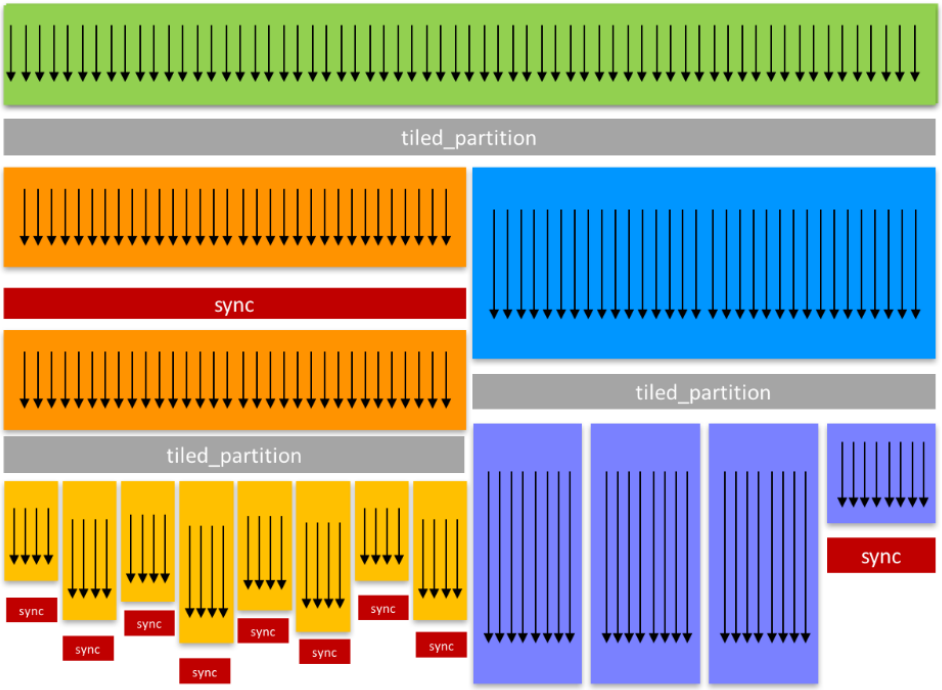
Szinkronizációs szintek NVIDIA Videókártyán

Jost Márk Benedek

2022. November 19.



Tartalomjegyzék

1. Kooperatív csoportok

Ebben a dokumentumban bemutatom, hogy CUDA API-ban hogyan lehet NVIDIA videókártyákon futó szálakat különböző szintű szinkronizációkba hozni.
Hatékony párhuzamosított programnak három alappillére van:

1. *Szálak csoportosításának hierarchiája*: MultiGrid - Grid - Blokk - Szál
2. *A szálak közti megosztott memória: shared memory*
3. *A szálak közti szinkronizáció*

Foglalkozzunk jelen esetben a 3. ponttal, a szinkronizáció kérdésével. Szinkronizáció szükséges ahhoz, hogy elágazó programunk biztonságos, fenntartható és moduláris legyen. A CUDA 9 bevezette az ún. Kooperatív csoport nevű absztrakciót (angolul Cooperative Groups), amely erős támogatást nyújt a kerneleknek ahhoz, hogy dinamikusan alakíthassanak ki csoportokat a szálakon belül.

Motiváció Korábban a CUDA API egy nagyon egyszerű, ámbár csak speciális esetekben működő megoldást biztosított a szinkronizáció kérdésére, a blokkon belüli barrier szinkronizációt: A "__syncthreads()" függvény addig nem engedte tovább futni a szálakat, amíg **a blokkon belül** minden még futó szál el nem jutott az adott pontig. Belátható, hogy nagy szál szám mellett ez nem elég, ugyanis egy blokkon belül (jelenlegi GPU-kon) legfeljebb 1024 szál futhat. Ha mi több, mint 1024 szálas programot írnánk, azaz több Streaming multiprocessor (SM) futna egymással párhuzamosan, akkor ezek összehangolására nem volt megfelelő szoftveres támogatás. Másik probléma az, hogyha a szálainknak csak egy kis, adott számú (például 4 vagy 32, de tipikusan 2-hatvány) részhalmazát akarjuk összehangolni.

Az API Leegyszerűsítve, a Cooperative Groups egy API Support Package, ami szálak csoportosítására és szinkronizálását segíti CUDA programokban. Az Package nagy része az összes CUDA 9-el kompatibilis GPU-ra működik, azaz Kepler és későbbi architektúrákra (Compute Capability 3.0+).

Ahhoz, hogy használhassuk, be kell illeszteni az alábbi headert .cu vagy .cuh kiterjesztésű fájlunk fejlécébe.

```
#include <cooperative_groups.h>
```

A típusok és interfészek a "cooperative_groups" C++ névtérben vannak definiálva, így vagy mindig prefixként kiírjuk, hogy "cooperative_groups::", vagy betöltjük a névteret a "using" direktívával.

```
using namespace cooperative_groups; // Névtér betöltése
using cooperative_groups::thread_group; // stb.
namespace cg = cooperative_groups; // Használhatunk rövid aliast is
```

2. Thread csoportok

Az egyik legfontosabb típus a csomagon belül a "thread_group" típus, ami threadek, azaz szálak csoportját tudja kezelni. Ezt terjeszti ki az összes, később tárgyalandó csoport objektum. Alapvető függvények rájuk:
- Megkaphatjuk a csoport méretét, azaz a benne futó szálak számát a size() metódussal

```
unsigned size();
```

- Megkaphatjuk a hívó thread indexét (0 és size()-1 közötti) a thread_rank() metódussal

```
unsigned thread_rank();
```

- Megvizsgálhatjuk a csoport érvényességét az is_valid() függvénnyel

```
bool is_valid();
```

2.1. Thread csoportokon végrahajtható kollektív (egyenhatású) műveletek

A thread csoportok megadják a lehetőségeket, hogy együttesen hajtsunk rajtuk végre műveleteket. Legegyszerűbb operációink egyike a szinkronizálás, ami annyit tesz, hogy a csoport tagjait nem engedi túl egy műveletsoron addig, míg minden tagja el nem jut odáig. Az összes thread csoport fajta támogatja a szinkronizálást, azonban mindegyik kicsit másképp.
A g csoporthoz tartozó szálakat a kollektív sync() metódussal, vagy a g-re a cooperative_groups::synchronize() függvényt meghívva szinkronizálhatjuk. Ezek ún. barrier szinkronizációt hajtanak végre.

```
g.sync();            // synchronize group g
cg::synchronize(g); // an equivalent way to synchronize g
```

A következő egyszerű példában egy device (GPU-n futtatott) függvény szerepel kooperatív csoportok felhasználásával. Amikor egy thread csoport meghívja, a szálak kiszámolják a szálanként kapott értékek összegét a csoporton belül.

```
using namespace cooperative_groups;
__device__ int reduce_sum(thread_group g, int *temp, int val)
{
    int lane = g.thread_rank();

    // Each iteration halves the number of active threads
    // Each thread adds its partial sum[i] to sum[lane+i]
    for (int i = g.size() / 2; i > 0; i /= 2)
    {
        temp[lane] = val;
        g.sync(); // wait for all threads to store
        if(lane<i) val += temp[lane + i];
        g.sync(); // wait for all threads to load
    }
    return val; // note: only thread 0 will return full sum
}
```

2.2. Thread Blokkok

Az első thread csoport fajta a Thread blokk. A Cooperative Groups azért vezette be ezt az adattípust, hogy explicit reprezentálja a CUDA programozás azonos nevű, egyik legfontosabb koncepcióját. A szálak egy-, kettő-, vagy háromdimenziós logikai egységbe szervezhetők, amit blokknak nevezünk. Ez a megoldás egy természetes módot nyújt arra, hogy vektorok, mátrixok vagy testek elemein hajtsunk végre számításokat. Az egy blokkba tartozó szálak száma korlátozott, mivel ezeknek közös processzormagon kell futniuk és a mag korlátos memória-erőforrásain kell osztozniuk. A jelenlegi GPU-kon egy blokk legfeljebb 1024 thread futtatására alkalmas, viszont a kernel több egyenlő méretű blokkban futtatható, ezért a szálak száma egyenlő a blokkonkénti szálak száma és a blokkokszám szorzatával. Egy thread blokk példánya az alábbi módon inicializálható:

```
thread_block block = this_thread_block();
```

Azon threadek, amelyek ugyanazon beépített CUDA blockIdx értékkel rendelkeznek, ugyanazon thread blokkba tartoznak. A thread blokkok szinkronizálása nagyon hasonló a korábban említett __syncthreads() metódushoz. A következő kódok mind ugyanolyan hatást érnek el: (feltéve, ha a blokk összes szála elér oda)

```
__syncthreads();
block.sync();
cg::synchronize(block);
this_thread_block().sync();
cg::synchronize(this_thread_block());
```

A "thread_block" adattípus kiterjeszti a "thread_group" interfészt két, blokk-specifikus tagfüggvénnyel. Ezek megfeleltethetőek a CUDA API blockIdx és threadIdx tagváltozóinak.

```
dim3 group_index();  // 3-dimenziós blokk index griden belül
dim3 thread_index(); // 3-dimenziós thread index blokkon belül
```

Az alábbi egy egyszerű kernel, ami a korábban látott "reduce_sum()" device függvényt használja egy tömb elemeinek összegzésére. Gyorsítás érdekében több, párhuzamosan elvégzett rész összeadással kezd (thread_sum()). A kernel thread blokkokat használ az összegzésre, és az atomikus "atomicAdd()" metódussal adja össze a részösszegeket.

```
__device__ int thread_sum(int *input, int n)
{
    int sum = 0;

    for(int i = blockIdx.x * blockDim.x + threadIdx.x;
        i < n / 4;
        i += blockDim.x * gridDim.x)
    {
        int4 in = ((int4*)input)[i];
        sum += in.x + in.y + in.z + in.w;
    }
    return sum;
}

__global__ void sum_kernel_block(int *sum, int *input, int n)
{
    int my_sum = thread_sum(input, n);

    extern __shared__ int temp[];
```

```
    auto g = this_thread_block();
    int block_sum = reduce_sum(g, temp, my_sum);

    if (g.thread_rank() == 0) atomicAdd(sum, block_sum);
}
```

Az előbbi függvényt futtathatjuk például egy 16 millió elemszámú tömb kiértékelésére.

```
int n = 1<<24;
int blockSize = 256;
int nBlocks = (n + blockSize - 1) / blockSize;
int sharedBytes = blockSize * sizeof(int);

int *sum, *data;
cudaMallocManaged(&sum, sizeof(int));
cudaMallocManaged(&data, n * sizeof(int));
std::fill_n(data, n, 1); // Adatok feltöltése
cudaMemset(sum, 0, sizeof(int));

// Kernelhívás
sum_kernel_block<<<nBlocks, blockSize, sharedBytes>>>(sum, data, n);
```

2.3. Csoport Partíciók

A Cooperative Groups kódcsomag megengedi, hogy már meglévő csoportjainkat rugalmasan particionáljuk új, kisebb egységekre, ezáltal finomítva párhuzamos algoritmusainkat. A cooperative_groups::tiled_partition() függvény felosztja az adott thread blokkot több részegységre. Itt egy példa, amely a teljes thread blokkot 32 szálú részegységekre bontja. A 32 gyakori felosztás, ugyanis a warpok (a CUDA szoftvermodell egyik alapfogalma) méretével megegyező.

```
thread_group tile32 = cg::partition(this_thread_block(), 32);
```

Egy másik példa partíciók létrehozására, amit az NVIDIA CUDA Toolkit Documentation (C.4.2.1. Thread Block Tile) említ.

```
/// The following code will create two sets of tiled groups, of size 32 and 4 respectively:
/// The latter has the provenance encoded in the type, while the first stores it in the handle
thread_block block = this_thread_block();
thread_block_tile<32> tile32 = tiled_partition<32>(block);
thread_block_tile<4, thread_block> tile4 = tiled_partition<4>(block);
```

2.4. Modularitás

A kooperatív csoportok használata nem csak gyors, de hasznos is tud lenni. A kódcsomag ereje a modularitás, amikor a csoportot explicit átadjuk függvények között, és konzisztensen hivatkozunk annak méretére. Ez segít kritikus versenyhelyzetek, illetve holtpontok elkerülésében, ugyanis így nem teszünk hibás következtetéseket elágazó függvényhívások miatt. Az alábbi egy elkerülendő példa szinkronizáció használatakor.

```
__device__ int sum(int *x, int n)
{
    ...
    __syncthreads();
    ...
    return total;
}

__global__ void parallel_kernel(float *x, int n)
{
    if (threadIdx.x < blockDim.x / 2)
        sum(x, count);  // hiba: threadek fele nem hívja meg a függvényt
                        // __syncthreads() => holtpont
}
```

A példában a threadeknek csak a fele hívja meg a sum() függvényt, ami tartalmaz "__syncthreads()" utasítást. A thread blokk nem minden threadje éri el a "__syncthreads()"-et, így holtpont alakul ki, mivel a "__syncthreads()" gátat képez addig, míg minden blokkon belüli thread el nem éri. Amennyiben alkalmazzuk a kooperatív csoportok adta lehetőségeket, ez a hiba nehezebben elkövethető. Fontos átadni a csoport típust, mint paramétert a hívandó függvénynek, és ekkor azon a csoporton végzünk szinkronizációt.

```
// Nyilvánvaló, hogy a teljes blokk meg kell hivja
// Van benne sync utasítás, ami különben holtpontot okozna
__device__ int sum(thread_block block, int *x, int n)
{
```

```
    ...
    block.sync();
    ...
    return total;
}

__global__ void parallel_kernel(float *x, int n)
{
    sum(this_thread_block(), x, count); // nincs elágazó függvényhívás
}
```

2.5. Grid csoport

Ez a csoport objektum reprezentálja az összes szálat, melyek közös grid alatt futnak. A sync() operációt kivéve minden API elérhető mindig, azonban ahhoz, hogy griden belül szinkronizálhassunk, a speciális "cooperative launch API" használatára van szükség. Egy grid csoport példánya az alábbi módon inicializálható:

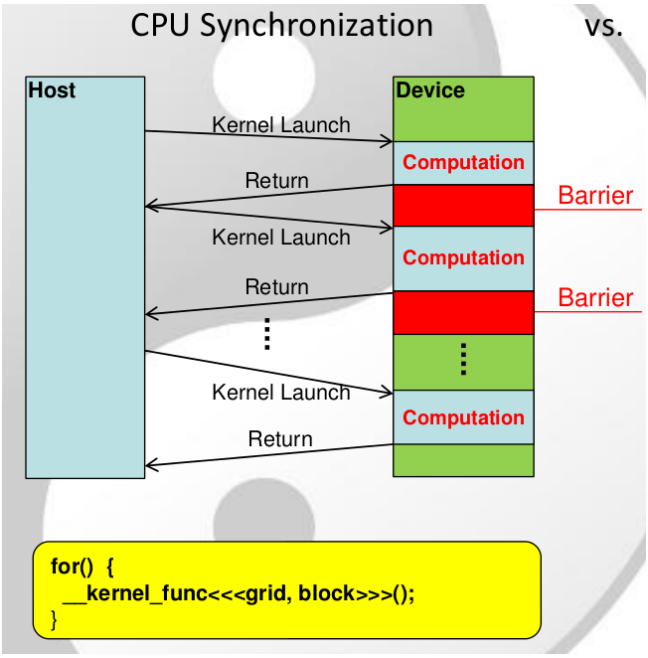
```
grid_group grid = this_grid();
```

A "grid_group" adattípus kiterjeszti a "thread_group" interfészt két, blokk-specifikus tagfüggvénnyel. Ezek megfeleltethetőek a CUDA API blockIdx és threadIdx tagváltozóinak.

```
dim3 block_index();  // 3-dimenziós blokk index griden belül
dim3 thread_index(); // 3-dimenziós thread index blokkon belül
```

2.5.1. Teljes Grid csoporton belüli szinkronizáció

A kooperatív csoportok bevezetése előtt a CUDA programozási modell natív támogatást csupán thread bokkokon belüli összehangolásra nyújtott. Régi gyakorlat az volt, hogy amennyiben grid szinten akartunk szinkronizációt alkalmazni, a kernelt felbontottuk több kisebb alkernelre, majd azon pontokon, ahol grid szintű szinkronizációra vágytunk, ott befejeztük az adott alkernelt, és hívtuk az újat. Ezen módszerre sokan "CPU Szinkronizáció" vagy "Implicit Szinkronizáció" néven hivatkoznak. Az 1. ábra szemlélteti a módszer lényegét. (Forrás: Prof. Wu Feng: To GPU Synchronize or Not GPU Synchronize?)



1. ábra. A kooperatív csoportok megjelenése előtt az egyetlen járható út teljes GPU szinkronizálásra. Látható, hogy gyakorlatilag minden programozási alapelvnek ellentmond.

A módszer több szempontból problémás. Egyrészt egy kernel hívása sok erőforrást igényel, ezért lehet, hogy sok kernelhívás mellett lassabb lesz a program, mintha egy szálon CPU-n futtattuk volna. Másrészt, ha az eredeti függvényben ciklus iterációnként akarnánk szinkronizálni, akkor Implicit szinkronizációs módszer mellett a kernelhívásokat kéne CPU cikluson belülre helyezni. Ez jobb esetben csak átláthatalan és fenntarthatatlan kódot eredményez, de akár a grafikus kártya meghibásodását is okozhatja a sűrű kernelhívás miatt. A probléma tehát adott, de nézzük, hogy a kooperatív csoportok hogyan jelentenek erre megoldást.

Ahhoz, hogy grid csoporton belül szinkronizáljunk, első ránézésre elég a grid.sync() függvényt használnunk, mint ahogy azt a thread blokkon belül is tettük.

```
grid_group grid = this_grid();
grid.sync();
```

A különbséget ott tapasztaljuk, amikor a kernel hívására kerül sor. A szokásos <<<...>>> konfigurációs szintaktika helyett a cudaLaunchCooperativeKernel CUDA runtime API hívást kell végrehajtani (vagy annak driver megfelelőjét).

Többféle módszer van a megfelelő több blokkos API hívásra, ezek közül nézzünk párat. Ahhoz, hogy biztosítsuk a thread blokkok megfelelő együttműködését a GPU-n, a blokkok száma előre megfontolandó. Ahhoz, hogy annyi blokkot futtassunk, ahány Streaming Multiprocessor (SM) van beépítve a rendszerbe, az alábbi kód alkalmas:

```
int device = 0;
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, device);
// initialize, then launch
cudaLaunchCooperativeKernel((void*)my_kernel, deviceProp.multiProcessorCount, numThreads, args
    );
```

Másik (javasolt) megoldás az, hogy megvizsgáljuk, hogy a kernel hívásához mekkora maximális blokkPerSM aránnyal dolgozhatunk (vagyis hány hány blokk fér el egy SM-en). Tegyük, fel, hogy szükségünk van 128 szálra, akkor így járhatunk el. A CUDA documentation az alábbi példakódhoz hasonló eljárást javasol:

```
/// This will launch a grid that can maximally fill the GPU, on the default stream with kernel
     arguments
int numBlocksPerSm = 0;
 // Number of threads my_kernel will be launched with
int numThreads = 128;
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSm, my_kernel, numThreads, 0);
// launch
void *kernelArgs[] = { /* add kernel args */ };
dim3 dimBlock(numThreads, 1, 1);
dim3 dimGrid(deviceProp.multiProcessorCount*numBlocksPerSm, 1, 1);
cudaLaunchCooperativeKernel((void*)my_kernel, dimGrid, dimBlock, kernelArgs);
```

Jó gyakorlatnak számít, ha hibatesztelés gyanánt odafigyelünk arra, hogy grafikus kártyánk egyáltalán támogatja-e a grid szinkronizációt, ugyanis ennek komoly feltételei vannak:

- Csak 6.0 compute capability eszközök támogatottak
- Támogatott Windows operációs rendszer (Jelenleg Windows 8.1/10/11)

! MPS ellátott Linux rendszer esetén csak 7.0 compute capability eszközök támogatottak

Az alábbi kód a cudaDevAttrCooperativeLaunch eszköz attribútumot vizsgálja meg.

```
int dev = 0;
int supportsCoopLaunch = 0;
cudaDeviceGetAttribute(&supportsCoopLaunch, cudaDevAttrCooperativeLaunch, dev);
```

Ez 1-be állítja a supportsCoopLaunch flaget ha a művelet támogatott a 0-s eszközön (egy grafikus kártyás rendszeren ez az alapértelmezett eszköz).

2.5.2. Program felkészítése több blokkos futtatásra

Mint azt már tárgyaltuk, bizonyos thread szám felett (jelenlegi GPU-kon 1024) a kerneleket már több blokkon szükséges futtani (hardveres okokból). Amennyiben elég egy blokk használata, márpedig számos alkalommal ez a helyzet, ki tudunk használni olyan trükköket gyorsításra, mint például a gyors megosztott memória használata a globális device memória helyett vagy a warpok hatékonyabb kihasználása. Éppen ezért előnyös lehet, ha algoritmusunkat egy és több blokkos használatra is megírjuk. Így képesek leszünk kezelni nagy thread számokat is, míg kis thread szám esetén kihasználjuk a lehetséges optimalizálási tulajdonságokat.

2.5.3. Egy gyakorlati példa

Esettanulmányként tekintsünk egy olyan algoritmust, ami N*N-es (N tetszőleges ismert konstans) mátrix determinánsát számolja Gauss elimináció segítségével. Eelméleti háttér, hogyha ekvivalens lépésekkel felsőháromszög mátrix-á alakítjuk kezdeti mátrixunk, azzal a determináns értékét nem befolyásoljuk. Egyszerű feladat abból a szempontból, hogy az algoritmus könnyen követhető, viszont az a része komplexebb, hogy ciklusszervezésen belüli szinkronizációt igényel.

Tegyük fel, hogy kernel függvényünk fejléce az alábbi lenne:

```
// Együtemű függvény fejléce
__global__ void detKernel(float *Matrix, float* det)
```

Bontsuk fel két függvényre, és nevében jelezzük, hogy azok egy vagy több thread blokkos futtatásra lettek tervezve.

```
// Grid: >1x1
__global__ void detKernel_multiBlock(float *Matrix, float* det)

// Grid: 1x1
__global__ void detKernel_1Block(float *Matrix, float* det)
```

Röviden ismertetném, hogy miért érdemes különvenni az alábbi két esetet.

```
__syncthreads();
grid.sync();
```

Bár nagyon hasonló működése van a két sornak, van egy fontos különbség. Amennyiben mi szeretnénk néhány szálat hamarabb leállítani (return-olni velük), a "__syncthreads()" függvény intelligens és nem fog rájuk várni. Sajnos ugyenezen lépés "grid.sync()" esetén azonnali holtpontot okoz, tehát több blokkos kódíráskor "halálfejes hiba"-ként kezelendő.

Különbség:

```
__global__ void detKernel_1Block(float *Matrix, float* det) { // 1 blokk
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // oszlopváltozó
    int j = blockIdx.y * blockDim.y + threadIdx.y;  // sorváltozó
    if (i >= N || j >= N)
        return;
__syncthreads();    // Működöképes
...
}

__global__ void detKernel_multiBlock(float *Matrix, float* det) { // >1 blokk
    // Szinkronizációs változó a teljes griden belül
    grid_group grid = this_grid();
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // oszlopvaltozo
    int j = blockIdx.y * blockDim.y + threadIdx.y;  // sorvaltozo
    if (i >= N || j >= N)
        return;
    grid.sync();    // Azonnali holtpont, ha valamely threadek visszaternek
```

Korábban már említett fontos előnye az egy blokkon futó parallel algoritmusnak, hogy ekkor minden thread hozzáfér ugyanazon megosztott memóriához. A megosztott memória egy nagyon fontos memóriatípus. Mérete tipikusan 164kB, mely flagek és kisebb bufferek tárolására tökéletesen alkalmas, ráadásul sávszélessége a globális memóriához képest akár húszszoros lehet.

Példánkban a Gauss elimináció során szeretnénk számon tartani a sorcserék számát (vagy legalább a paritását), ugyanis egy sorcsere a determináns értékét a (-1)-szeresére változtatja. Ezen kívül jó volna, hogy ha egy szál észreveszi, hogy van egy csupa 0 sor vagy oszlop (a determináns értéke ilyenkor 0), akkor ezt gyorsan tudná közvetíteni a többi szál felé.

Egy blokkos kivitelben: bátran fordulhatunk a megosztott meóriához, kihasználva annak kiváló sávszélességét.

```
__shared__ int sign;
    __shared__ bool fullZeroColoumn;

    // Kezdeti ertek adas a 0. thread altal
    if (i == 0 && j == 0) {
        sign = 1;
        fullZeroColoumn = false;
    }
```

Több blokkos esetben kénytelenek vagyunk a globális device memóriához fordulni, hiába van annak jelentősen kisebb sávszélessége.

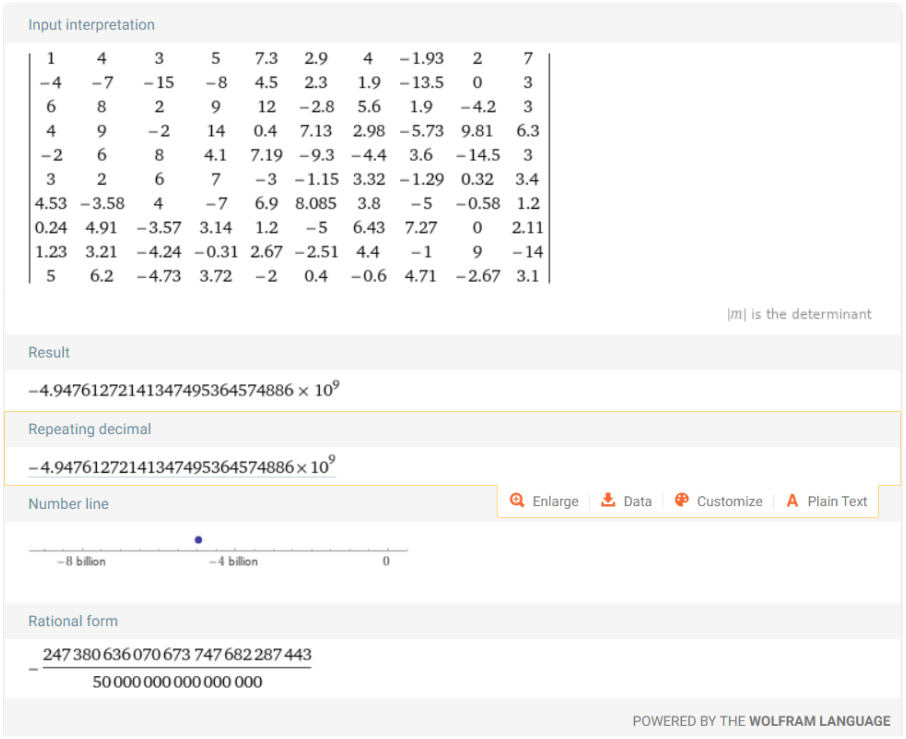
```
    // Kezdeti ertek adas a 0. thread altal
    if (i == 0 && j == 0) {
        global_sign = 1;
        global_fullZeroColoumn = false;
    }_
```

Ezen két példával akartam szemléltetni, hogy nem mindig a legnagyobb szintű párhuzamosság az egyértelmű leghatékonyabb. Van, hogy kevesebb szálon, de ügyes hardveres megfontolásokkal gyorsabbak lehetünk.

Most pedig nézzük, hogy működnek-e a megírt függvények. A tesztelést N=10-el a következő (egydimenziós, NxN-es vektorral reprezenált) mátrixon végeztem:

```
float A[N * N] = {  // Teszt 10x10-es mátrix
        1,4,3,5,7.3,2.9,4,-1.93,2,7,
        -4,-7,-15,-8,4.5,2.3,1.9,-13.5,0,3,
        6,8,2,9,12,-2.8,5.6,1.9,-4.2,3,
        4,9,-2,14,0.4,7.13,2.98,-5.73,9.81,6.3,
        -2,6,8,4.1,7.19,-9.3,-4.4,3.6,-14.5,3,
        3,2,6,7,-3,-1.15,3.32,-1.29,0.32,3.4,
        4.53,-3.58,4,-7,6.9,8.085,3.8,-5,-0.58,1.2,
        0.24,4.91,-3.57,3.14,1.2,-5,6.43,7.27,0,2.11,
        1.23,3.21,-4.24,-0.31,2.67,-2.51,4.4,-1,9,-14,
        5,6.2,-4.73,3.72,-2,0.4,-0.6,4.71,-2.67,3.1
    };
```

Az eredmény például a Wolfram alpha nevű internetes matematikai segédprogram segítségével ellenőrizhető, végeredményét a(z) 2. ábrán ábrázoltam. Azt tapasztaltam, hogy sokkal lassabban számol eredményt a programomnál, viszont pontosabban, ugyanis én floatokkal dolgoztam.**Tervben van összehasonlítani, hogy okoz-e gyorsulást float adattípussal dolgozni double helyett.** Amennyiben a Blokkméret 10x10 vagy nagyobb, akkor az 1 blokkos determinánsszámító függvény alkalmazható, és helyes eredményt ad, lásd 3. ábra. Ha a Blokkméretet lecsökkentjük például 5x5-ösre, akkor az 1 blokkos determinánsszámító függvény már nem alkalmazható, lásd 4. ábrán. Ilyenkor ugyanis 2x2-es a Grid. Ekkor a multBlokkra tervezett függvényt kell használni, a 5. ábrán láthatóan az 2x2-es Grid esetén is helyes eredményre jut. **A függvények futási idejének vizsgáltatára sajnos még nem került sor, de tervben van (saját magamnak emlékeztető gyanánt félkövér)**



2. ábra. A Wolfram Alpha nevű internetes matematikai segédprogrammal kapott eredmény

3. ábra. Az 1 blokkra tervezett függvény eredénye 1x1-es Grid használatakor láthatóan helyes

4. ábra. Az 1 blokkra tervezett függvény eredénye 2x2-es Grid használatakor. Nem meglepően helytelen

5. ábra. A több blokkra tervezett függvény eredénye 2x2-es Grid használatakor is láthatóan helyes

2.6. Multi Grid Csoportok

Ezen csoport objektum reprezentálja az összes szálat, ami fut az eszközökön multi-device rendszer esetében. Itt már az összes API használatához előírt a megfelelő lauch API használata. Egy multi grid csoport a következőképpen példányosítható:

```
// Kernel must be launched with the cooperative multi-device API
multi_grid_group g = this_multi_grid();
```

Elavultsági figyelmeztetés Fontos kiemelni, hogy bár a multigrid csoportoknak elvi jelentősége vitathatatlan, jelenleg szoftveres restaurációra szorulnak, és a CUDA 11.3-as frissítése óta **hivatalosan is elavultnak minősíttetett** minden eszközön.

2.6.1. Teljes multigrid szinkronizáció

Hasonló megfontolásokkal, mint ahogyan azt a 2.5.1 Teljes Griden belüli szinkronizáció esetében láttuk, érthető, hogy milyen hasznos lehet multi Grid rendszerben futó algoritmusok teljes szinkronja.

Elavultsági figyelmeztetés Fontos kiemelni, hogy bár a multigrid szinkronizáció elvi jelentősége vitathatatlan, jelenleg szoftveres restaurációra szorulnak, és a CUDA 11.3-as frissítése óta **hivatalosan is elavultnak minősíttetett** minden eszközön.

Ahhoz, hogy több eszközön keresztül elérhető legyen az kooperáció, kötelező a cudaLaunchCooperativeKernelMultiDevice CUDA API használata. A kernel hívási paramétereket egy struct tömbbe kell rendezni (egy-egy eszközönként).

Példa:

```
cudaDeviceProp deviceProp;
cudaGetDeviceCount(&numGpus);

// Per device launch parameters
cudaLaunchParams *launchParams = (cudaLaunchParams*)malloc(sizeof(cudaLaunchParams) * numGpus)
    ;
cudaStream_t *streams = (cudaStream_t*)malloc(sizeof(cudaStream_t) * numGpus);

// The kernel arguments are copied over during launch
// Its also possible to have individual copies of kernel arguments per device, but
// the signature and name of the function/kernel must be the same.
void *kernelArgs[] = { /* Add kernel arguments */ };

for (int i = 0; i < numGpus; i++) {
    cudaSetDevice(i);
    // Per device stream, but its also possible to use the default NULL stream of each device
    cudaStreamCreate(&streams[i]);
    // Loop over other devices and cudaDeviceEnablePeerAccess to get a faster barrier
    implementation
}
// Since all devices must be of the same compute capability and have the same launch
    configuration
// it is sufficient to query device 0 here
cudaGetDeviceProperties(&deviceProp[i], 0);
dim3 dimBlock(numThreads, 1, 1);
dim3 dimGrid(deviceProp.multiProcessorCount, 1, 1);
for (int i = 0; i < numGpus; i++) {
    launchParamsList[i].func = (void*)my_kernel;
    launchParamsList[i].gridDim = dimGrid;
    launchParamsList[i].blockDim = dimBlock;
    launchParamsList[i].sharedMem = 0;
    launchParamsList[i].stream = streams[i];
    launchParamsList[i].args = kernelArgs;
}
cudaLaunchCooperativeKernelMultiDevice(launchParams, numGpus);
```

Ezek után, a kernelen belül a multigrid szintű szinkronizáció a sync() utasítással érhető el.

```
multi_grid_group multi_grid = this_multi_grid();
multi_grid.sync();
```

A kód futtatásához az indítási paramétereket megfelelő módon kell beállítani. A device kódnak külön fordítási ciklusban kell fordulnia, amit az **-rdc=true** beállítással tehetünk meg. Microsoft Visual Studio fejlesztői környezetben ez a következőképpen állítható be: Jobb klikk a projektre, majd Properties / CUDA C/C++ / Generate Relocatable Device Code : Yes (-rdc=true)

Jó gyakorlatnak számít, ha hibatesztelés gyanánt odafigyelünk arra, hogy grafikus kártyáink egyáltalán támogatják-e a multi grid szinkronizációt, ugyanis ennek komoly feltételei vannak:

- Csak 6.0 compute capability eszközök támogatottak
- Támogatott Windows operációs rendszer (Jelenleg Windows 8.1/10/11) és device TCC módban

! MPS mentes Linux rendszer

Az alábbi kód a cudaDevAttrCooperativeMultiDeviceLaunch eszköz attribútumot vizsgálja meg.

```
int dev = 0;
int supportsMdCoopLaunch = 0;
cudaDeviceGetAttribute(&supportsMdCoopLaunch, cudaDevAttrCooperativeMultiDeviceLaunch, dev);
```

Ez 1-be állítja a supportsMdCoopLaunch flaget ha a művelet támogatott a 0-s eszközön (egy grafikus kártyás rendszeren ez az alapértelmezett eszköz).