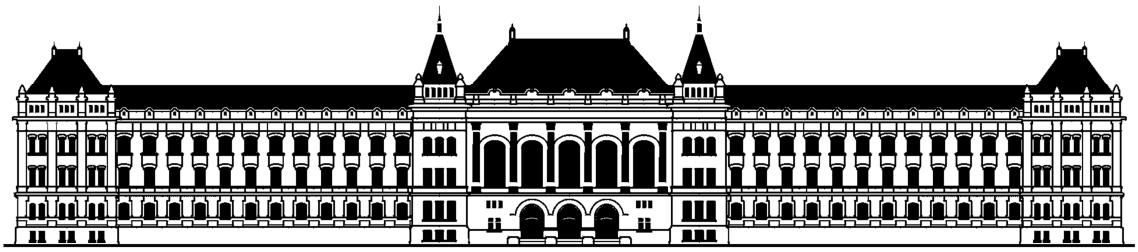


M Ű E G Y E T E M 1 7 8 2

Hallgatói segédlet:

Nvidia CUDA C programok debugolása Nvidia Optimus technológiás laptopokon

Készítette: *Kovács Andor*

2011/2012 első félév

A CUDA programok debugolásához kettő grafikus kártyára van szükség, ez annak köszönhető, hogy a CUDA képes kártyák a programok futtatása közben a debugoláshoz szükséges többletteendőit végzi, így nem képes megjelenítésére. Ezért szükségünk van egy második kártyára is.

Ekkor biztosan felmerül fel a hallgatóban az az egyszerű ötlet, hogy nem lehetne-e az **Optimus**-os laptopok integrált videokártyájára rábízni ezt a szerepet? A válasz röviden **igen**.

Ezen segédletben bemutatom a fejlesztői eszközök telepítéstől kezdve azok tesztelésig az összes fázist fűszerezve néhány alapvető CUDA tulajdonsággal.

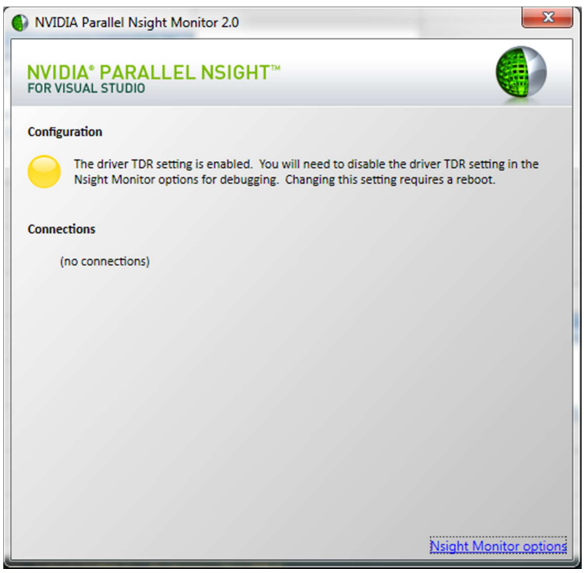
Ha nem vagy tisztában a CUDA alapjaival, akkor ajánlom figyelmedbe a segédlet végén található függeléket!

A telepítés fő lépései:

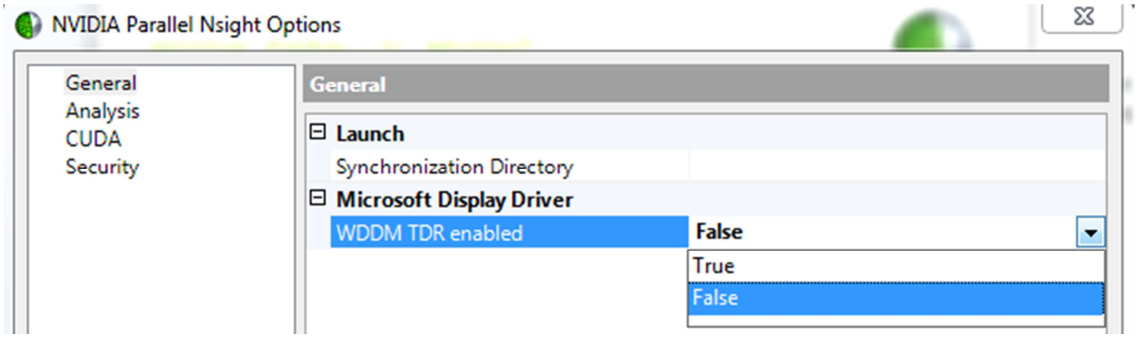
1. A telepítést ajánlott a megfelelő ***CUDA*** fejlesztői *driver*rel kezdeni, ezt az alábbi linkről tudod letölteni: http://developer.nvidia.com/cuda-toolkit-40 Az általam használt notebook verziószám: *275.33*
2. ***CUDA Toolkit*** legfrissebb ***4.0***-ás verzió telepítése.
3. ***Visual Studio 2008*** vagy ***2010*** telepítése (teljes verzió kell, Express Edition sajnos nem elég), a két verzió között nincs nagy különbség CUDA szempontból.
4. A tényleges debug szoftver a ***NVIDIA Parallel Nsight*** telepítése, ennek részeként telepítésre kerül az ***Nsight Monitor*** is, a telepítés végeztével ennek indításával kezdhetjük a konfigurálást.

Optimus Konfiguráció:

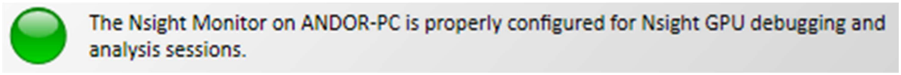
A monitor elindítása után a következő nem túl barátságos, de nagyon hasznos kép fogad bennünket.



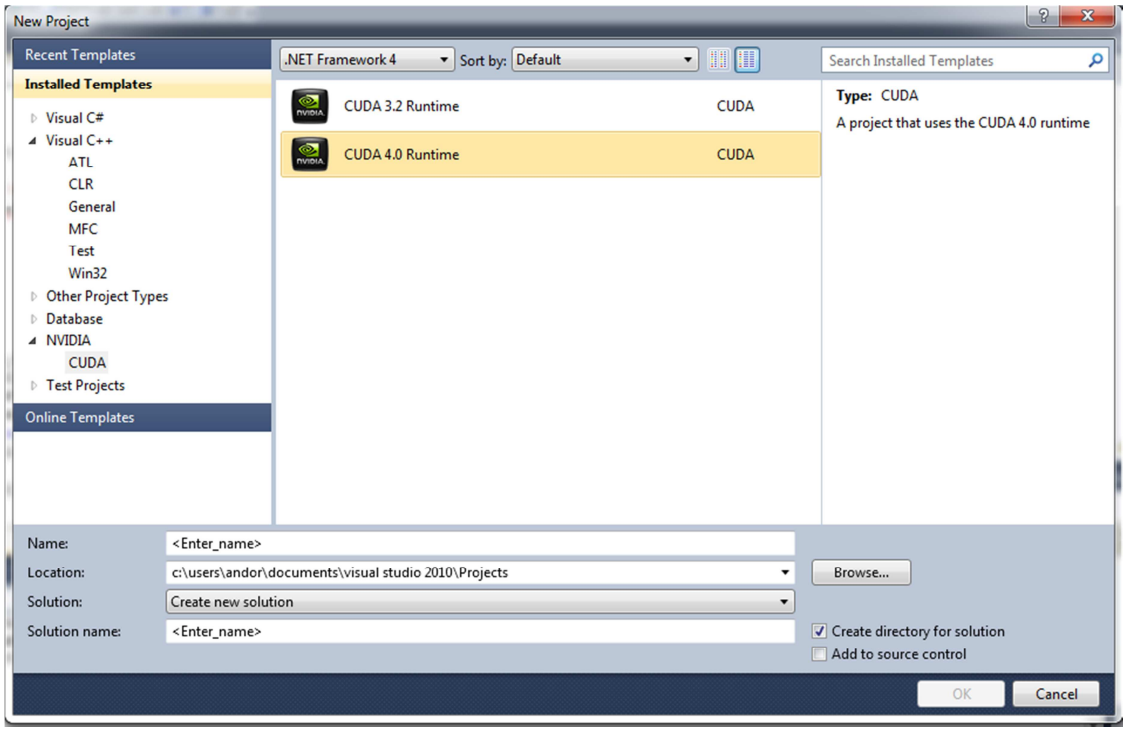
Tegyünk is eleget gyorsan a kérésnek, és az „*Nsight Monitor Options*” menüben a következő opciót állítsuk „*False*”-ra.



Ezután a szöveg a következőre változik, ekkor *indítsuk újra a gépet*, és jöhet a következő lépés.



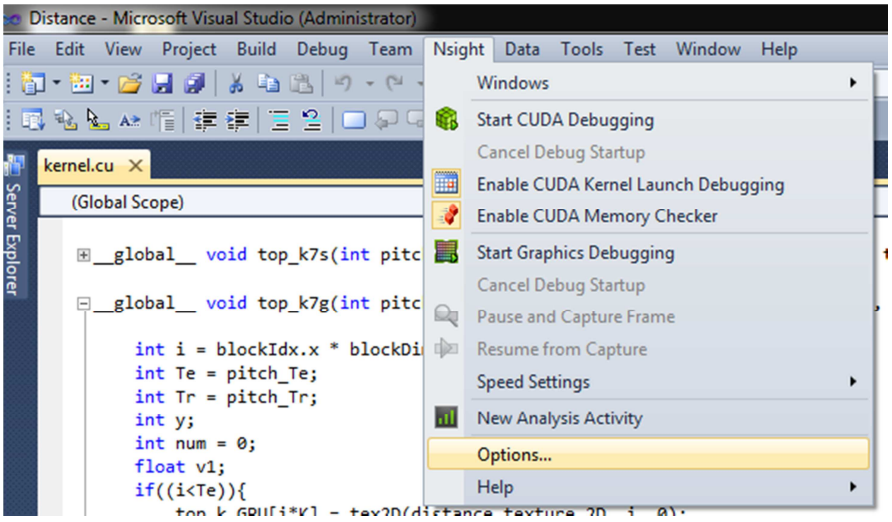
Újraindítás után indíthatjuk a *Visual Studio*-t, majd hozzunk létre egy új CUDA 4.0 projektet tetszőleges néven:



A projekt tartalmazni fog egy ***kernel.cu*** fájlt, amiben található egy mintaalkalmazás, ezt a későbbi tesztelés érdekében írjuk felül az alábbi egyszerű „*Hello CUDA*”-val:

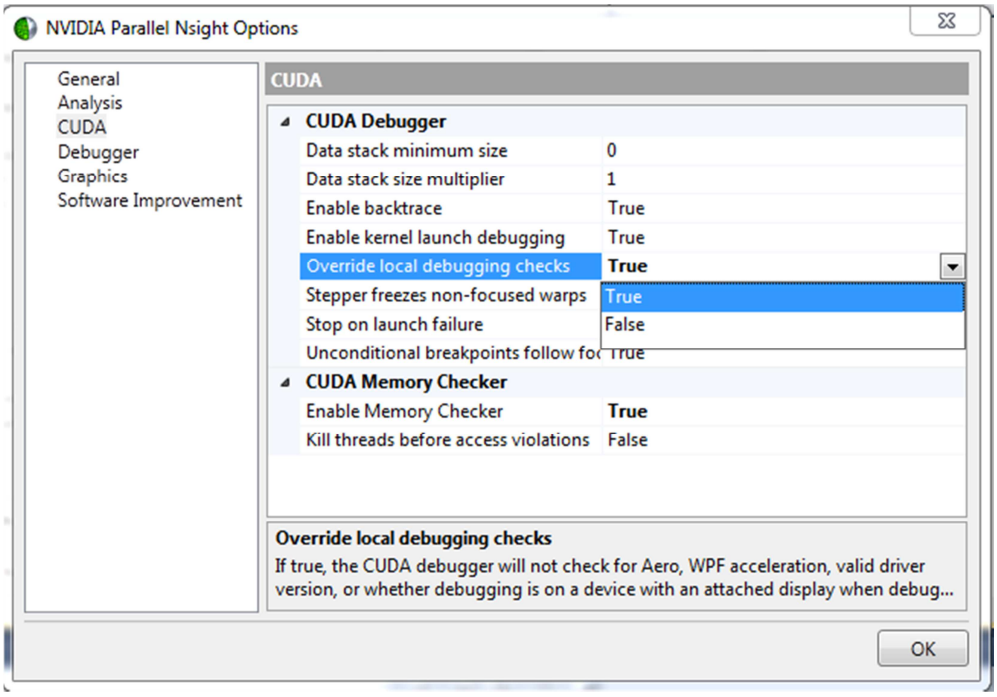
```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
__global__ void print(char *a,int N)//Kernel
{
        char p[11]="Hello CUDA";
        int idx=blockIdx.x*blockDim.x + threadIdx.x;
        if(idx<N)//Feltétel, hogy a „felesleges” szálak ne dolgozzanak
                {
                        a[idx]=p[idx];
                }
}
int main(void)
{
        char *a_h,*a_d;//Host és Device pointerek
        const int N=11;
        size_t size=N*sizeof(char);
        a_h=(char *)malloc(size);
        cudaMalloc((void **)&a_d,size); //Memóriafoglalás a GPU-n
        for(int i=0;i<N;i++)
        {
                a_h[i]=0;//CPU tömb kinullázása másolás előtt
        }
        cudaMemcpy(a_d,a_h,size,cudaMemcpyHostToDevice);//CPU->GPU másolás
        int blocksize=4;//Blokkméret=Blokkokon belüli szálak száma
        int nblock=N/blocksize+(N%blocksize==0?0:1);//Gridméret=Blokkok száma
        print<<<nblock,blocksize>>>(a_d,N);//Kernel futtatása
        cudaMemcpy(a_h,a_d,sizeof(char)*N,cudaMemcpyDeviceToHost);
        for(int i=0;i<N;i++)
        {
                printf("%c",a_h[i]);
        }
        free(a_h);
        cudaFree(a_d);//Memória felszabadítás

}
```

Megfigyelhetjük, hogy a menüsorban megjelent egy *Nsight* fül, ezen menü Options-ében kell a legfontosabb beállítást elvégeznünk:

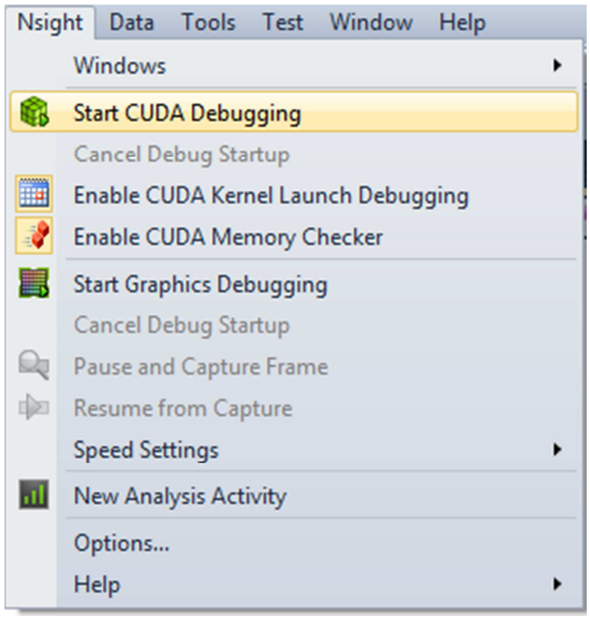


A következő ábrán látható a kulcsfontosságú opció, ennek beállítása után működni fog a debugolás, aki debugolt már korábban CUDA programot, a segédlet hátralévő részét ***kihagyhatja***.

A többiek számára bemutatom, mire érdemes **odafigyelni**.



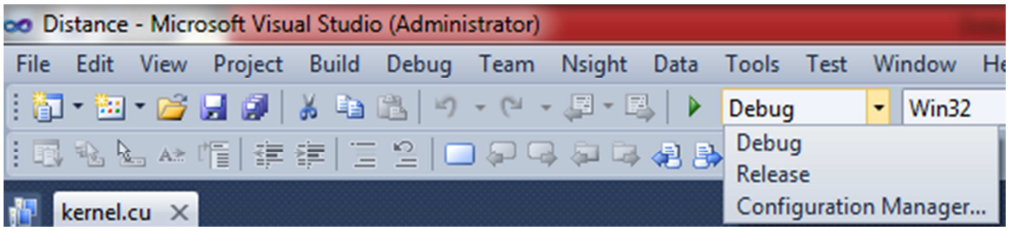
Fontos tudni, hogy a GPU kód debugolását nem a VS-ben megszokott módon kell indítani, mert úgy **átugorja** a kerneleket, ehelyett az Nsight menü alábbi opciójával:



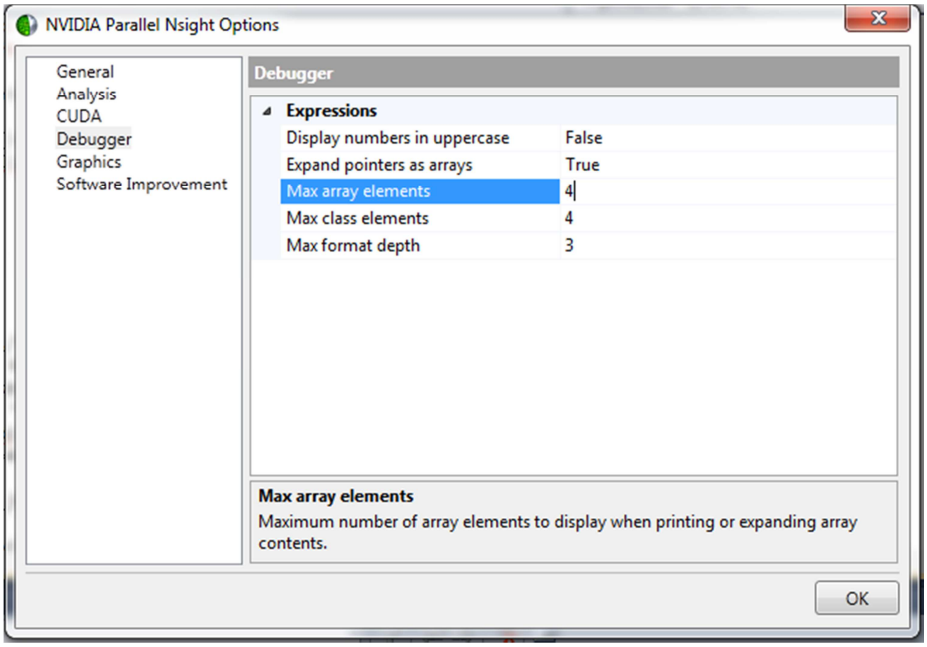
Úgy a legegyszerűbb tesztelni az eddigiek sikerességét, hogy a fenti minta alkalmazás alábbi pontjára beszúrunk egy *breakpoint*-ot, majd ha a futtatás megáll a kernelben (a program GPU-n futó kódrészlete) elhelyezett pontnál, akkor biztosak lehetünk benne, hogy a debugolás jól működik.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
__global__ void print(char *a,int N)
{
        char p[11]="Hello CUDA";
        int idx=blockIdx.x*blockDim.x + threadIdx.x;
        if(idx<N)
                {
                        a[idx]=p[idx];
At kernel.cu, line 10 ('print(char *a,int N)', line 6)
}
```

Még mielőtt elindítanánk, ellenőrizzünk le egy fontos dolgot, hogy az alábbi menüben a „*Debug*" opció van kiválasztva. „*Release"* módban nem tudunk debugolni, „*Debug"* módban pedig gyengébb a teljesítmény, mert nem optimalizálja a fordító „maximálisan" a kódot, így az jóval erőforrás-igényesebb lesz, ezt érdemes fejben tartani, kellemetlenségtől óvhat meg!



A *Parallel Nsight* másik fontos opciója az alábbi *képen* látható, ezzel azt állíthatjuk be GPU-n tárolt tömböknek hány elemét másolja vissza, és jelenítse meg debugolás közben, az alapbeállítás 4, ezt érdemes magasabb értékre állítani.

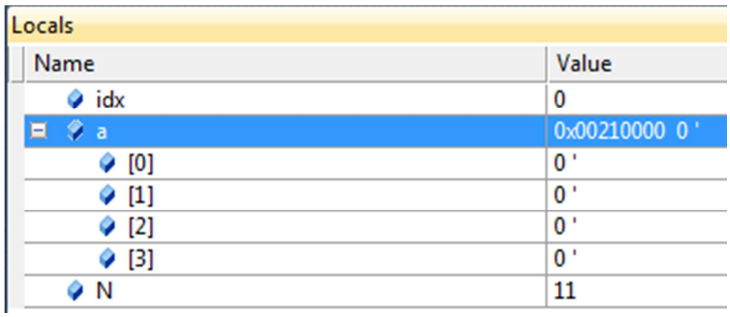


Ezen beállításokkal indíthatjuk a kernel debugolását. A programnak meg kell állnia a megadott ponton, és láthatóvá válnak az aktuális változó értékek.

A mintaalkalmazás egy kinullázott karakter tömböt másol a GPU-ra:

```
for(int i=0;i<N;i++)
        {
                a_h[i]=0;
        }

cudaMemcpy(a_d,a_h,size,cudaMemcpyHostToDevice);
```

Ez a tömb az „*a*” nevű az alábbi változók közül, habár 11 elemű a *fent említett* opció miatt csak az első 4 eleme látható.



A kernel mindösszesen annyit csinál, hogy minden aktív szállal, a szálak lokális memóriájában létrehozott „*p*” tömb egy elemét átmásolja az „*a*” tömbbe.

Kiegészítő tudás CUDA kezdőknek: A lokális memória egy „*kényelmi funkció*” a CUDA nyelvben, mert mint később látható lesz, a kernelekben nem lehet GPU memóriát foglalni. Tehát ha nem lenne lokális memória, akkor ezt az „*p*” tömböt is előre a CPU-s kódból kellett volna lefoglalni, majd egy CPU tömbbe beírni, ezeket a karaktereket, onnan átmásolni a GPU-ra majd a kernelnek átadni az erre mutató pointert, hasonlóan, mint most az „a”-ra mutatót.

Hogy ezt a kis memóriaterületet esetén ne kelljen megtenni, így lehetőségünk van architektúrától függően szálanként (!) 16-512 KB memóriát használni a következő egyszerű módon:

```
        char p[11]="Hello CUDA";
```

Erről azt érdemes tudni, hogy a lokális memória a globális memóriára (=DRAM) képződik le, tehát relatív lassú lesz, használatát, ha a sebesség is elsődleges szempont érdemes kerülni. **További részletek a függelékben!**

Ezen kezdőállapot után a kernel egyetlen lépesben lefut, ez annak köszönhető, mert a kernelt a következő párhuzamos szál és blokkszerkezettel indítottuk:

```
        print<<<nblock,blocksize>>>(a_d,N);//Kernel futtatása
```

Ahol *nblock* a „*Grid*” (grid=blokkok rácsa) mérete, ennek az értéke három. A *blocksize* pedig a blokkokon belüli szálak száma, ennek az értéke pedig négy. Tehát a GPU kernelünk futtatását úgy konfiguráltuk, hogy létrehoztunk 3 blokkot és 4 szálat blokkonként, tehát összesen 12 szálat hoztunk létre, és tettük ezt 1 dimenzióban az x tengely mentén. Ez nem látszik, mert a példa egyszerűsége miatt nem használtak *dim3* típusú objektumot, általában ezzel szokás a szálak és a blokkok 3d méretét megadni például következő módon:

```
dim3 nblock (Num_Block_X,Num_Block_Y,Num_Block_Z);
```

Fontos megjegyzés, hogy egy teljesítményorientált kernelnél a blokkonkénti szálak számát legalább 128-ra, de inkább 256-ra érdemes állítani, és 32-esével illik őket változtatni, mert futtatásuk úgynevezett *warp*-okban történik, amik 32-es csoportok.

Tehát van 12 szálunk, de csak 11 elemet szeretnénk „feldolgozni”, így a kernelbe be kell szúrnunk egy elágazást, hogy a jelen esetben egyetlen felesleges szálunk ne legyen aktív és ne írjon olyan memóriaterületet, amit nem lenne szabad. Ez így oldható meg:

```
int idx=blockIdx.x*blockDim.x + threadIdx.x;
        if(idx<N)//Feltétel, hogy a „felesleges” szálak ne dolgozzanak
                {}
```

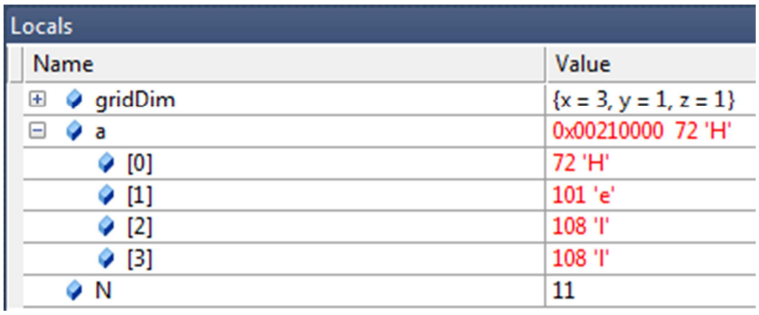
A *blockIdx*, *blockDim* és a *threadIdx* beépített dim3 típusú változók, ezek mindegyikének van x,y és z komponense, amik azt tárolják, hogy az adott szál éppen melyik blokkban milyen pozícióban található. A *blockDim.x* adja meg a blokkonkénti x tengelyen létrehozott szálak számát. Ezen változók segítségével nagyon könnyen lekérdezhetjük az éppen aktuális szálunk pozícióját, ami jelen esetben megegyezik az éppen feldolgozandó adat pozíciójával. **Szemléletesebb példa a függelékben.**

Tehát egy általános szabály: Minél közelebb áll a kernelünk szál és blokkszerkezete a feldolgozandó adat szerkezetéhez, annál könnyebb dolgunk van a kernel megírásakor.

Mivel csak az x tengelyen hoztunk létre szálakat így, elég az x értékeket néznünk, így az *idx* változó értéket 0 és 11 között fog változni, attól függően, hogy épp melyik blokk melyik szálról van szó. Azonban a tömbünk, csak 11 elemű, így elengedhetetlen a `idx<N` feltétel.

Ez egy meglehetősen egyszerű dolognak tűnik, de ezt jó, ha már az egyszerű eseteknél is figyelembe vesszük, mert később kettő vagy három dimenzióban nagyobb szálszám mellett komoly kellemetlenséget jelenthetnek ezek a „letiltatlan” felesleges szálak.

A párhuzamos feldolgozás tehát egy lépésben lezajlik, így az „a” tömb a következőre változik:



Reményeim szerint ezen segédlet hasznodra vált, kitartást és **sok szerencsét kívánok a fejlesztéshez!**

Kovács Andor

Függelék: A CUDA alapjai:

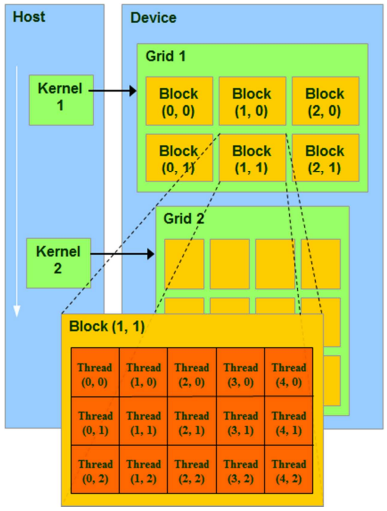
A CUDA[*2*] egy általános párhuzamos programozási architektúra, ami magas szintű nyelvi támogatást biztosít a grafikus kártyák számítási kapacitásának kihasználására. Az architektúra nagyban megkönnyíti a programozó feladatát, hiszen nincs szükség komoly számítógépes grafikai előismeretekre, emellett komoly skálázhatóságra ad lehetőséget.

Ki kell emelnem, hogy ezen összefoglalómban nem részletezem azt, hogy a nyelv hogyan képződik le ténylegesen a hardverre, mert ez jóval meghaladja ezen függelék terjedelmét, ezért a nyelv logikai felépítésének ismertetésével folytatom.

A párhuzamosítás alapkövei az úgynevezett kernel-ek, melyek a grafikus kártyán futó függvényeknek felelnek meg, egyszerre ilyenből egy kártya egyszerre csak egy félét tud futtatni. (kivétel, ha ***Fermi*** architektúráról van szó) Viszont az az egy kernel tetszőlegesen példányosítható, ezekhez külön szál jön létre, tehát az egyes szálakban ugyanaz a kód fut. Minden szál rendelkezik egy egyedi azonosítóval, ennek a szerepe azért jelentős, mert ez alapján tudjuk őket megkülönböztetni, ezen azonosító arra használható, hogy eldöntsük, az egyes szálaknak mely adatokon kell dolgoznia.

Annak érdekében, hogy a szálakat könnyebben tudjuk kezelni, illetve könnyebben tudjuk skálázni alkalmazásunk, a szálakat blokkokba rendezzük. Amikor a kernel-t meghívjuk, akkor meg kell adnunk a blokkok méretét, tehát, hogy hány szálat tartalmazzanak, így minden blokk mérete ugyanakkora lesz. Blokkokon belül a szálakat lehetőségünk van logikailag 1, 2 vagy 3 dimenzióba rendezni, illetve magukat a blokkokat is rendezhetjük 1 vagy 2 dimenzióba. Ennek jelentőségét és az ebben rejlő lehetőségeket később magyarázom.

A legfontosabb követelmény a szálakkal kapcsolatban, hogy futásuk egymástól teljesen független kell, hogy legyen, tehát ugyanazt az eredményt kell kapnunk, a szálak szekvenciális illetve párhuzamos futtatása esetén.

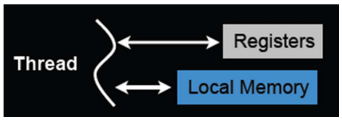


1. ábra: CUDA C szál hierarchiája [*3*]

A **1. ábrán** látható, hogy a blokkok felett is van egy hierarchiai réteg, az úgynevezett grid, ebből kernelenként csak egy lehet. Látható, hogy a kernel futtatásához, egy 3x2 blokkból álló gridet definiáltak, az egyes blokkok dimenziója pedig 5x3. A dimenziókat egy dim3 típusú objektummal szokás megadni, aminek x,y és z attribútumai vannak, ezeknek az alapértelmezett értéke 1.

A szálak és blokkok struktúrájának megértése utána a memóriatipusok és azok hatáskörét kellett megismernem. Az alábbi 3 ábra és az információk a [*4*] forrásból származnak.

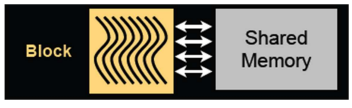
A **2. ábrán** láthatóak, azon memóriatípusok, amik kifejezetten egy szálhoz vannak rendelve, más szál nem férhet hozzájuk. Ide tartoznak a szál regiszterei, amik 1 órajel alatt elérhetőek, méretük 32 bit, mennyiségüket a fordító és a videókártya típusa határozza meg.



2. ábra: szálakhoz rendelt memóriatípusok

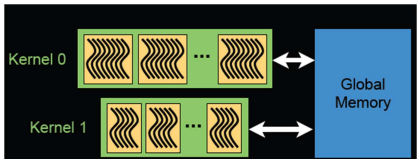
A lokális memóriában azok az adatok vagy adatszerkezetek tárolódnak, amik a regiszterekbe vagy nem férnek bele, vagy nem képezhetők le. Például a tömbök tárolásához egységes memóriaterületre van szükség, ezek 32 bites regiszterekre nem bonthatók szét. Fontos megjegyezni, hogy a lokális memória használata lehetőség szerint kerülendő, hiszen ezen terület a globális memóriának egy kifejezetten a szál számára elkülönített területe, így elérési ideje körülbelül 100-szor lassabb, mint a regisztereké.

A **3. ábrán** látható a CUDA architektúra, teljesítmény szempontjából egyik legkritikusabb eleme a megosztott memória. Ez, mint látható blokk szinten elérhető, tehát az adott blokk összes szála látja, és tudja írni. Úgynevezett „bankokból" áll, ha egyszerre több szál szeretnek ugyanahhoz a bankhoz hozzáférni, akkor bankkonfliktus lép fel, ilyenkor sorosítani kell a kéréseket, ami a teljesítmény rovására megy. Ha nem lép fel konfliktus, akkor a megosztott memória elérési ideje közel megegyezik a regiszterekével, de ez egy egységes memóriaterület, ahova tömböket is le tudunk képezni.



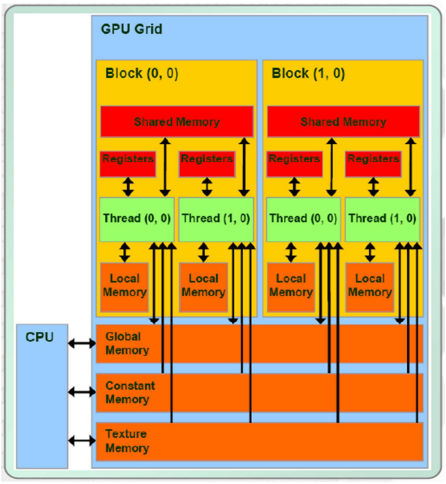
3. ábra: A blokkokhoz tartozó memóriatípus

Az **4. ábrán** látható a legnagyobb és egyben leglassabb elérhető memóriaterület a globális memória, ehhez az adott kernel összes szála hozzá tud férni, sőt, egy a kernel lefutása után indított újabb kernel is látja a változásokat. Körülbelül 100x lassabb, mint a regiszterek, vagy a bankkonfliktus nélküli megosztott memória.



4. ábra: globális memória

A **5. ábrán** megfigyelhetőek az eddig leírtak összefoglalva. Illetve látható, hogy van konstans és textúra memória is. Ezek a lokális memóriához hasonlóan a globális memóriába képződnek le, de cache-elve vannak, emiatt ha a ha olyan adatra hivatkozunk, ami benne van a cache-ben akkor azt a globális memória sebességénél jóval gyorsabban elérhetjük. A konstans memória mérete az egész kártyára nézve 64 Kbyte, ami meglehetősen korlátozza széleskörű alkalmazását. A textúra memória pedig teljesen más szemléletet és kezelés igényel, az ebben rejlő lehetőségek elemzése, későbbi munkám célját jelenti majd.



5. ábra: CUDA architektúra [*5*]

Két fontos CUDA fogalmat kell megemlítenem, ami a **5. ábrával** kapcsolatban felmerülhet, az egyik az úgynevezett hoszt, ez a CPU, ebből csak egy lehet, ez adja a vezérlést az eszközöknek (Device), amik a GPU-k, kell a többes szám, hiszen több kártya kezelésére is alkalmas a nyelv. Ezen tudás birtokában egy CUDA program futása általános esetben a következő módon néz ki [*6*]:

1. Adatok beolvasása a hoszton
2. Adatok másolása a hosztról az eszköz globális memóriájába
3. Kernel függvény hívása
    I. adatok másolása a globális memóriából a közös memóriába
    II. számítások
    III. eredmények visszaírása a közös memóriából a globális memóriába
4. Eredmény másolása az eszközről a hoszt memóriájába
5. Eredmény kiírása

Ezen pontok közül a 3. illetve annak alpontjai párhuzamosan kerülnek futtatásra. A függvény hívásakor meg kell adnunk azt, hogy a blokkokon belül mekkora legyen a szálak mennyisége és elrendezése. Fontos technikai részlet az I. alpont, tehát a megosztott memóriába való betöltés a globális memóriából, ezzel tulajdonképpen cache-nek használjuk a megosztott memóriát, jelentős sebességnövekedést jelent, mivel onnan optimális esetben legalább 100x gyorsabban tudunk adatokat kiolvasni.

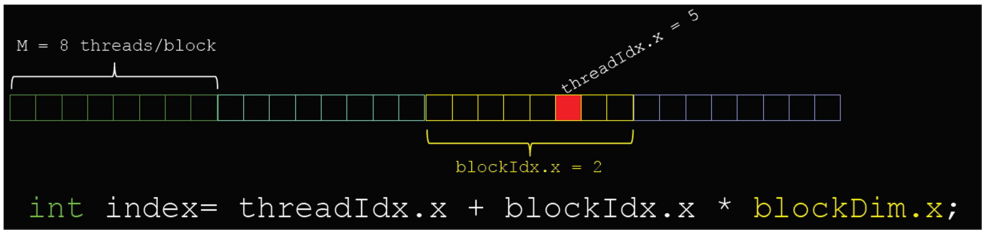
Fontos még, hogy ezen lépések közül a 3. alpontjait kivéve, mindegyiknek a futtatási parancsát a hoszt adja ki, azonban ezen parancsok között vannak szinkron és asszinkron tipusúak. Szinkron tipusúak például a CPU<->GPU másolások, tehát a CPU nem kapja vissza a vezérlést egészen addig, amíg az adott parancs le nem futott az eszközön. Asszinkron pedig a kernel hívás, ilyenkor a hívás után a CPU visszakapja a vezérlést, így végezhet műveleteket, vagy akár egy másik GPU-nak is adhat parancsot.[*7*]

A CUDA architektúra fontos jellemzője, hogy a szálak futását, csak blokk szinten lehet szinkronizálni, persze, ha az összes blokknak kiadjuk a szinkronizációs parancsot, akkor ha elérik a szinkronizációs pontot, akkor az összes blokk összes szála lefutott, tehát a kernel végzett. Erre szolgál a hoszton hívható ***cudaThreadSynchronize();*** függvény, ezáltal értesülünk arról, hogy a kernel lefutott, és elkezdhetjük az eredményeket visszamásolni.

A kernelhívás asszinkron mivoltára még a mérések során kellett figyelmet fordítanom.

A fentieket kiegészítve részletezem, hogy miért elengedhetetlen a CUDA működéséhez, a szálak megfelelő azonosítása, és miért jelent hatalmas előnyt, hogy a szálakat akár 2 vagy 3 dimenzióba is rendezhetjük.

Ehhez a **6. ábrán** látható példára van szükség, van egy 4x8 méretű float tömbünk a globális memóriában, a feladat például az, hogy floatok értékét olvassuk ki, majd írjuk vissza a helyére a négyzetüket. Először önkényesen úgy döntünk, hogy legyen 8 szál 1 blokkban, ezek 1 dimenzióban helyezkednek el. Így 4 blokkra lesz szükségünk, mint korábban leírtam a CUDA legfontosabb elve, az hogy a szálak egymástól függetlenül tudjanak dolgozni. Ezt itt most úgy garantáljuk, hogy minden szálnak annyi lesz a feladata, hogy kiolvassa a hozzá rendelt float értékét, majd visszaírja a négyzetét. Ekkor merül fel a kérdés, hogy melyik float tartozik, hozzá? Ezt könnyedén megadhatjuk a **6. ábrán** látható *dim3* típusú objektumok segítségével. A ***blockIdx*** azt adja meg, hogy az adott szál, ami a futtatást végzi, melyik blokkban található, a ***blockDim*** pedig azt, hogy egy blokkban hány szál található. Ha ezt a kettőt összeszorozzuk és hozzáadjuk ***threadIdx*** értékét, ami a blokkon belüli szálat azonosítja, akkor mivel a szálak száma megegyezik a tömb elemeinek a számával, egyértelműen kijelöltük, hogy az adott szálnak a tömb mely elemét kell négyzetre emelnie.



6. ábra: szál azonosítók [*8*]

A ***6. ábrán*** látható ***.x***-ek a *dim3* objektumok x attribútumára utalna. Korábban leírtam van y és z attribútum is, ehhez a példához hasonlóan kell eljárni akkor, ha 2 vagy 3 dimenziós adatszerkezetet akarunk feldolgozni. Tehát általánosságban kimondható, hogy a szálak elrendezését úgy kell kialakítanunk, hogy minél inkább igazodjuk azon adathalmaz szerkezetéhez, amin dolgozni szeretnénk. Minél inkább hasonlít annál könnyebb kijelölni a feldolgozandó adatok szálhoz tartozó részhalmazát.

A függelék végén meg kell említenem, hogy a CUDA kompatibilis kártyákat 5 féle csoportra lehet osztani, ezek nem térnek ki a globális memória (~grafikus kártya memóriája) méretére,a kártyán található Streaming Multiprocessor-ok és memória buszok sebességére. Ezen csoportok a Compure Capability-k.

A SM-ek mennyisége sem definiált a szabványokban, az viszont igen, hogy 8 db magot tartalmaznak, tehát a kártyák egyik fontos jellemzője a CUDA magok száma, ami egyenlő SM*8-al.

Irodalomjegyzék a függelékhez:

[1] University of Oxford Many Core Group, CUDA bevezető diasor - http://www.many-core.group.cam.ac.uk/archive/CUDAcourse09/gp/CUDA_May_09_GP_L1.pdf - 2011.05.02.

[2] *Edward Kandrot és Jason Sanders*: CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010, ISBN: 9780131387683

[3] University of Toronto, Programming Massively Parallel Multiprocessors Using CUDA tárgy diasora, 16. dia - http://www.eecg.toronto.edu/~moshovos/CUDA08/slides/003%20-%20Hardware.ppt - 2011.05.02.

[4] Harvard University, CS 264 Massively Paralell Computing tárgy, 125. dia - http://www.cs264.org/lectures/files/CS264_2011_03-CUDABasics_share.pdf - 2011.05.02.

[5] *Szénási Sándor*: Óbudai Egyetem, GPGPU alapjai tárgy, 25. dia - http://nik.uni-obuda.hu/app/APP02.pdf - 2011.05.02.

[6] CUDA C Programming Guide 3.2 elolvasása után összegezve - http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf - 2011.05.02.

[7] A [4] hivatkozás diasorának 162. diája

[8] Harvard University, CS 264 Massively Paralell Computing tárgy, 35. dia - http://www.cs264.org/lectures/files/CS264_2011_04-CUDAIntermediate_share_tmp.pdf - 2011.05.02.