SZAKDOLGOZAT

Grafikus kártyán futtatott Monte Carlo gamma szimuláció szcintillációs detektorhoz

Incze Attila

Témavezető: *Dr. Légrády Dávid*
adjunktus
BME NTI

BME, 2009

Témakiírás:

Azonosító: Sz-2009-65	
Szakdolgozat címe:	**Grafikus kártyán futtatott Monte Carlo gamma szimuláció szcintillációs detektorhoz**
Melyik szakiránynak ajánlott?	
A jelentkezővel szemben támasztott elvárások:	igényes matematikai felkészültség, alapos Monte Carlo ismeretek, alapos programozási készség
Leírása:	A grafikus kártyák processzorainak (GPU) párhuzamos számítási kapacitási fejlődése átütő jelentőségű. A nagy számításigényű, de a részecsketranszport szempontjából pontosságában felülmúlhatatlan Monte Carlo (MC) módszer alkalmazási területe rendkívül széles: az orvosi berendezésektől az atomreaktorokig rengeteg felhasználási lehetőséget nyújt. A hallgató feladata a MC részecsketranszport GPU-n való számítási lehetőségnek feltérképezése meglévő szcintillációs detektor szimulációjára alkalmas MC program GPU-ra való átültetésével.

Témavezető:

Neve: Dr. Légrády Dávid
Tanszéke: NTI
E-mail címe: legrady@reak.bme.hu
Telefonszáma: 1254

Önállósági nyilatkozat:

Nyilatkozom, hogy a szakdolgozat keretében végzett munkám, és jelen dokumentum önálló szellemi termékem.

Incze Attila
Budapest, 2009. május 18.

Tartalomjegyzék

Külön mellékletek jegyzéke

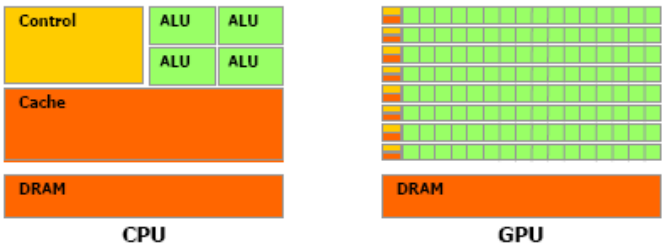
1. Bevezetés

A grafikus kártyák mára túllépték azt az eszközt, mely csak a kijelző vezérlésére szolgál, manapság dedikált processzorral és memóriával rendelkeznek, hogy kielégítsék a 3D szoftveripar igényeit. Az nVidia által közzétett CUDA fejlesztőkörnyezettel a grafikus processzorra írhatunk tetszőleges célú párhuzamos adatfeldolgozást nyújtó programot, és a felkínált számítási kapacitást nagyon sokrétűen tudjuk alkalmazni. Szakdolgozatom témája ennek segítségével gamma fotonok szcintillációs kristályban történő nyomon követése. A szakdolgozat készítése során nem volt lehetőségem más szerzők műveire hivatkozni, vagy összehasonlítani az elért eredményeket, ugyanis az általános célú grafikus kártyán való programozás csak az elmúlt években fejlődött valós alternatívává, és még senki sem publikált a témával kapcsolatos cikket. Éppen ezért e szakdolgozat némiképp úttörőnek tekinthető a témában.

2. Az nVidia GPU programozása

2.1 Az architektúra

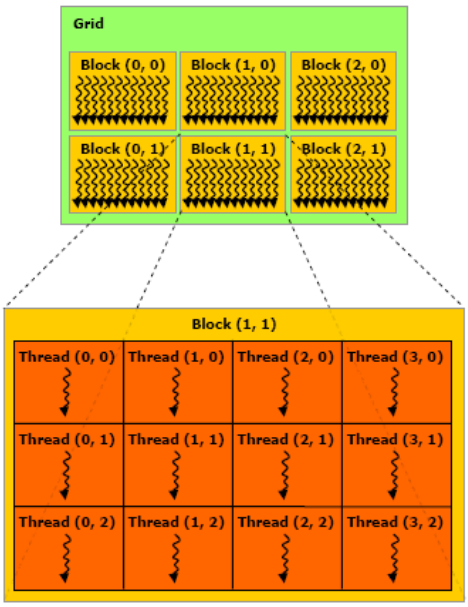
A GPU betűszó a Graphics Processing Unit (grafikus feldolgozóegység) rövidítése, a grafikus kártya központi egysége, mely a grafikus memóriával, az alaplappal (a PCI interfészen keresztül) kommunikál, és a megjelenítőket vezérli (RGB, DVI kimeneteken). Manapság számítógépekben, PDA-kban, és egyéb eszközökben nagyon széleskörűen alkalmazott segédprocesszor, mely a CPU (központi egység) terhelését csökkenti. Az általános célú CPU-val szemben a GPU speciális párhuzamos feldolgozásra optimalizált. Ma ezt leginkább a 3D alkalmazások követelik meg, mely számítások jellegzetessége, hogy ugyanazokat a műveleteket kell különböző adatokra elvégezni, ezért a GPU architektúrája SIMD (Single Instruction, Multiple Data) jellegű. A CPU és GPU sematikus architektúráját a 2.1 ábra szimbolizálja.



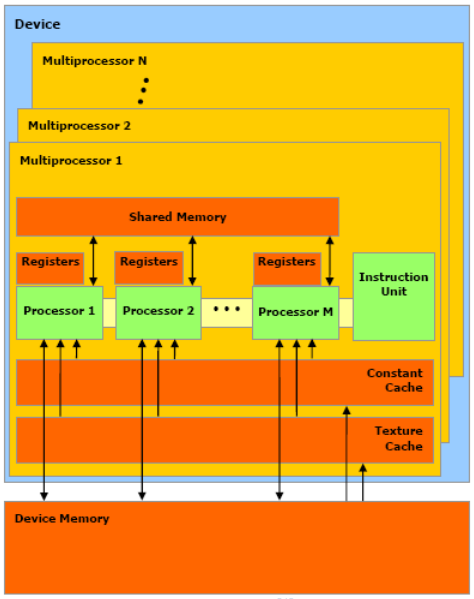
2.1. ábra[1]
A központi, és grafikus processzor eltérő struktúrája

A központi processzor tranzisztorainak csak kis hányadát használja műveletvégzésre, nagy része adatáramlás, valamint gyorsítótárért felelős egységekbe tartozik. A grafikus processzornál ezzel szemben fordított a helyzet. (A gyorsítótár – avagy cache – a számítástechnikában az átmeneti információtároló elemeket jelenti, melyek célja az információ-hozzáférés gyorsítása.)
Fontos még megjegyezni, hogy míg a CPU esetén hagyományosan kevés a magok száma (manapság 2-4), és azokhoz is közös irányító egység, valamint gyorsítótár tartozik, addig a GPU sok magos (manapság 128-240), és ezek a magok ún. multiprocesszorokba szerveződve külön irányító egységgel, és (viszonylag kevés) lokális memóriával rendelkeznek.

A megírt kódunk utasítás sorozattá fordítódik a GPU nyelvére, mely, ha feldolgozás alatt áll, szálnak (thread) hívjuk. A GPU egy órajel ciklusa alatt minden egyes magján egy utasítás hajtódik végre, ami azt jelenti, hogy annyi szálat képes párhuzamosan futtatni, ahány mag áll rendelkezésre. A GPU-ban futó ezen szálak struktúrákba rendeződnek, hogy könnyebben kezelhetőek legyenek. A legfelső szintű elem a rács (grid), ez egy két dimenziós tároló, melynek minden eleme egy blokk (block). A blokkok maximum három dimenziós tömbökként foghatóak fel, melynek minden eleme egy elemi szálat (thread) képvisel. Ezt láthatjuk a 2.2 ábrán.
Ezen szálakat azonban nem mi közvetlenül indítjuk, ahogy a CPU-n megszokhattuk. Amikor a GPU-n szeretnénk futtatni, akkor meg kell mondanunk, mely kódrészletet (függvényt) futtassa, és milyen grid/block struktúrában a GPU. Ezután nem vagyunk képesek kommunikálni a futó szálakkal, az összes szál lefutását teljes mértékben a GPU feladatütemező menedzseli, és akkor szól „nekünk" (a CPU kódnak), ha befejezte a munkát. Miután az összes szál ugyanazt a megadott függvényt fogja futtatni, felmerül a kérdés: mi értelme ugyanazt a függvényt ugyanazokkal a paraméterekkel többezerszer lefuttatni?
A megoldás, hogy a GPU-n futó szál le tudja kérdezni a grid/block indexét, mely segítségével tudja, hogy ő „melyik a sok közül", tehát milyen adattal kell dolgoznia (pl. a memóriából), és hova kell a futása eredményét a memóriában tárolni.



2.2. ábra[1]
A szálak rács és blokk struktúrákba szerveződnek a GPU-ban



2.3. ábra[1]
A GPU-ban futó szálak különböző memóriaterületeket használhatnak céljaiknak megfelelően

Memória nélkül mit sem érnénk, ugyanis valószínűleg a bemeneti értékeket is ott adjuk át a GPU-n futó szálaknak, de a kimenetet pedig csak oda tudjuk tárolni, ugyanis a szálak által futtatott függvénynek nincs visszatérési értéke. (Hogy is lehetne, ha egyszer hívjuk meg, de az igazából többezerszer fut le.)
A szálak memóriakezelését a 2.3 ábra mutatja be. Minden multiprocesszoron futó szálhoz allokálódnak regiszterek az adott multiprocesszorhoz tartozó regiszter-blokkból, mely mérete 8192, ill. a legújabb GPU-kban 16384 regiszter.
A multiprocesszorok lokális, megosztott memóriáját használhatják az abban futó szálak. A szálaknak ezen kívül hozzáférésük van az ún. device memóriához, ami a

grafikus kártya processzoron kívüli DDR2 ill DDR3 típusú, nagy kapacitású (256MB-2GB) memóriája. Ez a memória azonban közel két nagyságrenddel lassabb elérésű, mint a GPU-n belül található lokális memóriák, ill. gyorsítótárak.
Két speciális memóriatípust használhatunk még a programjainkban. Mindkettő csak olvasható, valamint fizikailag a device memóriában tárolódik, viszont az elérések gyorsítása érdekében a GPU gyorsítótárazza ezeket a memóriaterületeket.
Az ún. konstans memóriába tetszőleges adatstruktúrát tárolhatunk, de (mint a neve is mutatja) csak olvasható elérésre. Ide olyan kisebb méretű adatokat érdemes helyezni, melyek a program futása közben sokszor kell (csak olvasásra) megnyitni, viszont a megosztott memóriában nem férne el. Amennyiben ilyen memóriából olvasunk, a GPU automatikusan gyorsítótárazza az olvasáshoz közeli bájtcsoportokat, így ha egy közeli területről fogunk újra olvasni, akkor azt a gyorsítótárból tudja nekünk jóval kisebb késleltetéssel visszaadni.
A textúra egy speciális típusú memória. Fizikailag szintén a device memóriában tárolódik, de ezt a környezet elrejti előlünk, mi a textúra elemeit, az ún. texeleket kérhetjük le. A textúra a grafikai alkalmazások egyik alapköve, eredetileg kétdimenziós tömb, melynek elemei egy-egy színt írnak le (r, g, b értékek), gyakorlatilag egy kép. Azóta elvontabb fogalom lett, és a CUDA implementáció alapján akár 3D textúrákat is létrehozhatunk, nagyon sokféle elemtípussal (byte, float alapú 3 és 4 elemszámú, stb.). A textúra egy helyen felvett értékét egyszerű függvényhívással kérhetjük le a texel indexei alapján. Amennyiben nem egész számot adtunk meg indexnek, automatikusan (a közeli texelek értékeiből) lineárisan interpolált értéket kapunk vissza, mely nem kerül többlet processzor időbe, ugyanis ezért a GPU-ban külön egység felelős. Ezen kívül a textúrához való hozzáférés is gyorsítótárazott, így növelve a textúra olvasások sebességét.

2.2 Hatékony kód készítése

Hogy hatékony kódot írhassunk, több feltételnek kell eleget tennünk. A lehető legkevesebb számításból kell megoldanunk a problémát, tehát jó algoritmust kell találnunk, de ugyanilyen fontos a futtató hardver ismerete is. A GPU programozásnál ez kiemelten nagy jelentőségű. A legújabb CUDA környezetben már alig találni limitálásokat a kódra nézve, de ha nem az architektúrát megfelelően kihasználó kódot írunk, semmit nem nyerünk azzal, hogy a programunkat GPU-n futtatjuk le CPU helyett.

A GPU egy multiprocesszorán futó szálak mindig csak ugyanazt a műveletet végezhetik (természetesen más-más adaton), ebből adódóan, amikor feltételhez érkezik a kód, az elágazás mindkét útját végigfuttatja a multiprocesszor, közben viszont azon szálak, amelyek nem az adott elágazásban haladtak, kimaszkolódnak. Ez annyit tesz, hogy az eltelt órajelek alatt üresen állnak, nem végeznek műveletet. Most már érthetjük, hogy a teljesítmény maximalizálásához ezt szükséges minél jobban elkerülni. Ha tehát az üresen állást (kimaszkolódást) minimalizálni akarjuk, minél kevesebb feltételt kell használnunk, a szükséges feltételeket pedig úgy kell megalkotnunk, hogy az elágazások rövidek legyenek. További megfontolással, még optimálisabb a helyzet, ha a feltétel rövidebb elágazásában fut a szálak kisebbik része, ugyanis ekkor a hosszabb elágazásban a szálak nagyobbik része fut (azaz csak kis része maszkolódik ki), így összességében kevesebb az üresjáratú szálak által összesen állással telt idő.
A terminológia szerint a szálak konvergálnak, amennyiben feltétel és elágazás nélkül futnak, és divergálnak, ha feltételhez érnek (különös módon, ha egymásba ágyazott feltételek sorozatán keresztül).

Az elágazások mellett a másik legfontosabb tényező a szálak méretével kapcsolatos. Mint már láthattuk, a multiprocesszorok megadott mennyiségű

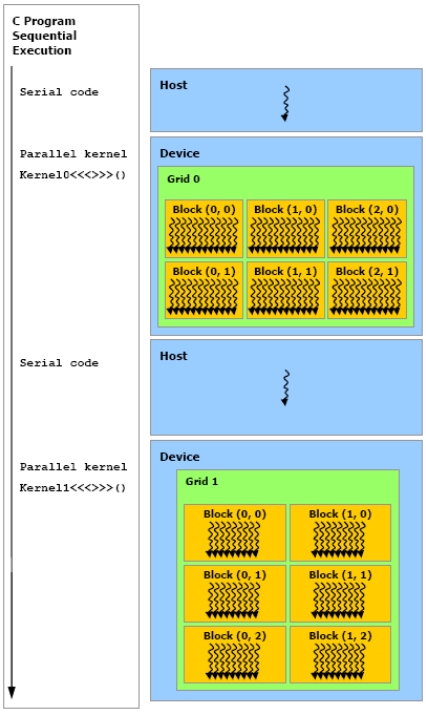
regiszterrel, valamint lokális memóriával rendelkeznek. Ahhoz tehát, hogy a GPU kód legalább annyi szálon elindítható legyen, amennyi a multiprocesszorban lévő magok (skalárprocesszorok) száma, az szükségeltetik, hogy annyi szálhoz szükséges regiszterek és lokális memória „beférjen" a multiprocesszorba. Ezek azok a tényezők, amelyek limitálják az egyszerre futtható szálak számát a GPU-ban. (Természetesen azon fizikai határon belül, hogy hány magot tartalmaz a GPU.).
Mindebből az következik, hogy GPU-ra egyszerű struktúrájú, kevés lokális memóriát használó programot kell írnunk a hatékony működés eredménye érdekében.

2.3 A fejlesztőkörnyezet

Az nVidia cég 2007-ben tette közzé a CUDA nevezetű fejlesztőkörnyezetét, mellyel az azt támogatott GPU-kra tudunk kódot fordítani. Amennyiben kompatibilis videokártyánk, és ehhez telepített meghajtó programunk van, abban az esetben a CUDA fordító által elkészített futtatható állományt minden további nélkül elindíthatjuk.

A CUDA fejlesztőkörnyezetben a nagy népszerűségnek örvendő C-ben programozhatunk, természetesen a forrásfájlokat (általában .cu kiterjesztésűek) nem standard C, hanem CUDA fordítóval kell fordítanunk, ami a standard C kiegészítéseként „megérti" a GPU specifikus hívásokat, műveleteket.

A CUDA programunk két fő részből áll. A host kód, és a device kód. A host kódot a CPU futtatja, a device-t pedig a GPU. A host kódból ún. kernelek hívásával tudunk számítást elindítani a GPU-n, mely párhuzamosan is képes futni a host kóddal (hála a két fizikailag is különböző futtató eszköznek). Ezt a futtatási sémát szimbolizálja a 2.4 ábra.



2.4. ábra[1]
Heterogén programozás: a host és device kód elkülönülése

A C függvények neve elé helyezett minősítő prefixum függvényében a fordító a GPU-ra fordítja a kódot (__global__, __device__), vagy a CPU-ra (__host__, vagy minősítő hiánya). A __global__ minősítésű függvény az ún. kernel, amit csak a host kódból hívhatunk, de a GPU-n fut. A __device__ függvényeket csak device kódból hívhatjuk, és maguk is a GPU-n futnak. A __host__ függvények standard módon a host kódból kerülnek hívásra, és a CPU-n futnak le.
Az elkészült CUDA programunk tehát - mely természetesen a CPU-n kezd el futni - amikor számítást kíván végezni a GPU-n, egy __global__ (kernel) függvény hívásával teheti azt meg. A GPU-n való számítás elindítására egy egyszerű példa:

```
// Kernel definition
__global__ void vecAdd(float* A, float* B, float* C)
{
}

int main()
{
        // Kernel invocation
        vecAdd<<<1, N>>>(A, B, C);
}
```

A <<<…>>> szintaxis segítségével indíthatjuk a GPU-n a számolást, mégpedig előre meghatározott számú szálon és struktúrában. A <<<…>>> közötti két érték közül az első a grid struktúrát adja meg, a második a block struktúrát. Ez a példa azt jelentette, hogy 1 gridet indítunk, és abban N threadet. Egy bonyolultabb példa, melyben kihasználjuk a többdimenziós futtatási struktúrákat:

```
dim3 dimGrid(16, 16);
dim3 dimBlock(4, 4, 4);
vecAdd<<<dimGrid, dimBlock>>>(A, B, C);
```

A kernel futása aszinkron módon történik, az indítása után a host kód folytatja a futást, majd egy CUDA API hívással blokkolhatja a végrehajtást addig, amíg a kernel futása véget nem ért:

```
cudaThreadSynchronize();
```

A CUDA API függvényeket nyújt ezen kívül a grafikus memóriában (device memory, azon belül konstans memória, textúra) való memóriakezeléshez is (allokálás, másolás, deallokálás), mely segítségével a kernel futása előtt felállíthatjuk a futásához szükséges környezetet, betölthetjük a bemeneti adatokat, a futás végén pedig kimásolhatjuk a futás során generált eredményeket.

A grafikus kártya kezelése CUDA-ban tehát olyan, mint egy külső erőforrás használata, a host kód tetszőleges pontjában „feltöltjük” rá a bemenetet, rábízunk egy párhuzamosított számolást, a futás végeztéről értesülünk, majd „letöltjük” az eredményeket, amit a későbbiekben tetszőlegesen használunk.

3. Az implementáció

3.1 Modellkörnyezet

Bár a szakdolgozat témakiírásában egy meglévő kód grafikus kártyára történő implementációja szerepel, a CPU-val történő sebesség összehasonlítás céljából a grafikus kártyára megírt kódot alakítottam át a GPU-specifikus részek kiiktatásával CPU kóddá, így azt mondhatjuk, hogy nagyon jó közelítéssel ugyanazt a számítási kapacitást igényli a GPU-n, és CPU-n futó szimuláció is.
Éppen emiatt tehettem néhány önkényes választást, az egyik a detektor alakjának megválasztása. Létezik egy nagyon optimális számolás egy téglatest és egy egyenes metszéspontjainak meghatározására, ami egy CUDA példaprogramban is szerepel, ezért téglatest alakú detektorra írtam meg a szimulációt. Természetesen a metszéspontot meghatározó algoritmus egy külön függvényben van elhelyezve, tehát ennek megváltoztatásával más formájú detektorokat is modellezhetünk.
A modellkörnyezet tehát egy fent említett téglatest alakú szcintillációs kristályból és egy monoenergetikus, izotróp módon sugárzó pontforrásból áll. A teret tökéletes vákuum tölti ki, így egyszerűsítve a modellt.
A modellezett kölcsönhatások a Compton-szórás, fotoeffektus, valamint a párkeltés. A fotonokat a forrásból egyesével indítjuk, és a kristályban egy foton által leadott összenergiát detektáljuk. A program konstans 1024 csatornás analizátort szimulál. Ez elég részletességet nyújt a spektrumok analizálásához, ettől függetlenül természetesen a későbbiekben ez az érték könnyen módosítható.
Mint később láthatjuk, számos fizikai paramétert futtatási időben, azaz bemenetként tudunk megadni, mint például a detektor tömb, a forrás térbeli elhelyezkedése, a fotonok induló energiája.

3.2 Program szerkezete

A szimulációs programot Microsoft Visual Studio 9.0 környezetben készítettem el. Ettől függetlenül ún. makefile-t is készítettem hozzá, mely segítségével Linux alatt is lefordíthatjuk a forráskódot. A készített program ennek megfelelően cross-platform, a (kevés) szükséges helyen fordítási makrókkal lekezelve, így a két legelterjedtebb platform: Windows és Linux alatt is használható.
A program három forrásfájlból áll. Az egyik a scintillator.cu nevű, mely a main() függvényt is tartalmazza. Ebben a fájlban host kód található, a szimuláció előkészítése, bemeneti értékek feldolgozása, a szimuláció meghívása (mind GPU és CPU részről is), majd az eredmények kiíratása és mentése történik. A scintillator_kernel.cu forrásfájl a GPU-n futó Monte Carlo device kódot tartalmazza, míg a scintillator_gold.cpp a CPU-n futó szimulációt tartalmazza, szintén host kód. Az utóbbi kettő között csak apró eltéréseket találhatunk, ezekről még lesz szó a későbbiekben.

3.3 Be- és kimeneti lehetőségek

A GPU és CPU futtatású szimuláció kimenete az 1024 csatorna mindegyikén mért beütési gyakoriság (a detektált beütésszám az indított fotonok számára normálva). A program elsődlegesen ezt a két 1024 elemű tömböt írja a GPU és CPU-nak megfelelő egy-egy fájlba ascii-formátumban.
Másodlagosan a program kimenete az egyes szimulációk futási ideje, mely azt a nagyon fontos információt tudatja velünk, hogy mennyivel gyorsabban tudunk a grafikus kártyán lefuttatni egy megadott paraméterezésű szimulációt, mint a

központi processzoron. Végső soron ez fogja eldönteni „megéri-e” grafikus kártyán szimulálni részecsketranszportot.
A program bemenete a szimuláció fizikai, és futtatási paraméterei. Ezeket a paramétereket egy konfigurációs fájlból olvassa be a host kód, de lehetőség van parancssori argumentumokként is megadni őket.
Ez a technika könnyű lehetőséget biztosít automatizált szimuláció futtatásra (ahogyan a „Paraméterek hatása a teljesítményre” alfejezetben a grafikonok is készültek), mivel a program paramétereit egyszerűen indításkor adhatjuk meg az argumentumokkal.
Szintén bemenetnek tekinthető a hatáskeresztmetszet fájl (`cross-section.txt`), mely szintén ascii formátumban 4 oszlopot tartalmaz tabulátorral elválasztva, az fájl 128 soros. Az oszlopokban található számok rendre a fotoeffektus, Compton-szórás, párkeltés kölcsönhatásához, valamint a totális hatáskeresztmetszethez tartozó értékek. Az értékek betöltéséről, valamint mintavételezéséről a későbbiekben lesz szó.

A konfigurációs fájl (`input-data.txt`) és az állítható paraméterek:

```
# Physical parameters
#------------------------------

# Position of the photon source, (x y z) coordinates
[source]
0.0 0.0 -5.0

# Energy of the gamma source, MeV
[energy]
3.0

# Density of the scintillator crystal, g/cm^3
[density]
3.67

# Crystal position in space, (x y z) coordinates
[boxmin]
-3.0 -3.0 -3.0
[boxmax]
3.0 3.0 3.0

# Gauss scatter FWHM to apply to detected energies
[scatter]
0.0


# Simulation details
#------------------------------

# GPU kernel invocation parameter
[threads]
192

# GPU kernel invocation parameter
[grids]
1024

# How many photons to track in one thread
[photonsperthread]
100

# How many kernel to invoke in a row
[repeat]
10
```

A fizikai paraméterek a forrás energiája (`energy`), a forrás térbeli helye (`source`), a szcintillációs kristály sűrűsége (`density`), valamint a szórás nagysága, melyet a detektálandó értékekhez szeretnénk adni, ha nem nulla félértékszélességet kívánunk a spektrumban. A szcintillációs kristály egy téglatest, melynek élei

párhuzamosak a koordinátarendszer tengelyeivel, és két átellenes csúcsának koordinátáival adhatjuk meg helyzetét (`boxmin, boxmax`).
A kernel indításához szükséges paraméterek az egy blokkban lévő szálak száma (`threads`), és a grid mérete (`grids`). Egy szálon annyi foton kerül leszimulálásra, amennyit a `photonsperthread` változóval beállítunk. A `repeat` változó segítségével mondhatjuk meg, hányszor futtassuk le egymás után a kernelt A `threads*grids*photonsperthread` kifejezés adja meg, hány fotont szimulál le egy kernel hívás, azonban ez viszonylag limitált, így lehetőségünk van még egy szorzóval állítani az indítani kívánt fotonok számát.

Amennyiben a programot a --help kapcsolóval indítjuk, megkapjuk a lehetséges parancssori argumentumok listáját, rövid leírásukkal:

```
Welcome to Scintillator v1.0
GPU based Monte Carlo simulation. (C) 2009 Attila Incze.
------------------------------------

Usage: scintillator [parameters]

Set configuration with input-data.txt file in the working directory.

You can also use the following command line parameters:
        --threads=X    Launch the kernel with X threads
        --grids=X      Launch the kernel with X grids
        --photonsperthread=X    Simulate X photon in each thread
        --repeat=X     How many kernels to launch
        --energy=X     Photon source energy (MeV)
        --density=X    Density of the scintillator crystal (g/cm^3)
        --scatter=X    Gauss scatter FWHM to apply to measured data (MeV)
        --sourcex=X, --sourcey=Y, --sourcez=Z
                       The position of the photon source (X, Y, Z)
        --boxminx=X, --boxminy=Y, --boxminz=Z
        --boxmaxx=X, --boxmaxy=Y, --boxmaxz=Z
                       The position of the crystal (bottom left and top right
corner)
        --help         Print this help
        --printheader   Prints to error output a header for results (for test
running)
        --printresult  Prints to error output the result of simulation (for test
running)

Example: ./scintillator --repeat=10 --sourcez=-5.0 --scatter=0.01
```

A szimuláció futása során az alábbihoz hasonló kimenetet produkál a program:

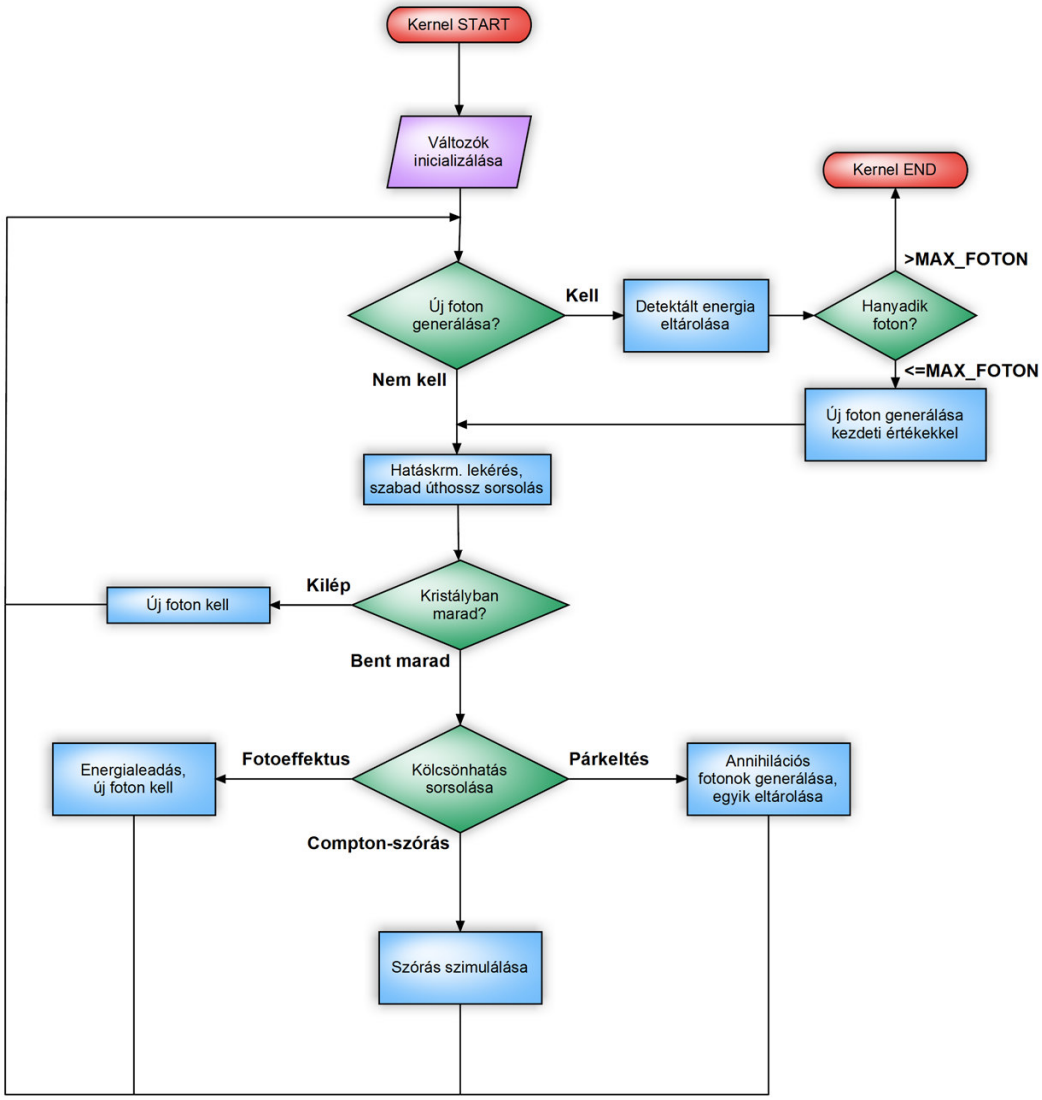
```
Welcome to Scintillator v1.0
GPU based Monte Carlo simulation. (C) 2009 Attila Incze.
------------------------------------
Simulation parameters:
Source: (0.00 0.00 -2.10), 3.00 MeV
Crystal: (-2.00 -2.00 -2.00) - (2.00 2.00 2.00), 3.67 g/cm^3
Threads: 192, Grids: 100, Photons per thread: 100, Repeat: 100
============================================================
Simulation on GPU in progress...
  -> GPU processing time: 4488.73 ms
  -> Photons/msec: 42774
Simulation on CPU in progress...
  -> CPU processing time (estimated): 141731.23 ms (real: 14173.12 ms)
  -> Photons/msec: 1355
------------------------------------
GPU/CPU ratio: 31.57
```

A bemeneti paraméterek kiírása után a GPU-n és a CPU-n is lefuttatja a szimulációt, és kiírja a futásidőket, ebből pedig számítási kapacitást számol. A CPU időtakarékossági okokból a fotonok (a forráskódban konstanssal állítható) töredékét szimulálja, mint a GPU, a teljes futásidőt (estimated CPU processing time) csupán extrapolálja.

4. Kód adaptáció GPU-ra

4.1 A foton követő ciklus

A GPU-n futó kódot a lehetőségekhez mérten maximális mértékben konvergensre kellett fejleszteni, azaz, hogy az a lehető legkevesebb elágazást, feltételt tartalmazza, valamint a feltételes, divergáló kódrészletek a lehető legrövidebbek legyenek. Ilyen módon a szálak közül a lehető legtöbb tud kimaszkolódás nélkül párhuzamosan futni.
A 4.1 ábrán az általam implementált foton követő kód logikai felépítése látható.



4.1. ábra
A fotonkövető device kód (kernel) logikai felépítése

A kód egy ciklus köré szerveződik, az összes szál ebben a ciklusban konvergál. A ciklus egy lépésében egy foton előrehaladásának, valamint kölcsönhatásának szimulációja történik. Ez tekinthetjük tehát egy atomi műveletnek: szabad

úthosszt sorsolunk, és ha a foton a kristályban belül marad kölcsönhatást sorsolunk, és a kölcsönhatást leszimuláljuk.
Ezt az atomi műveletet ismételjük iteratív módon, amíg a foton el nem nyelődik, vagy ki nem lép a kristályból. Azonban, egy-egy foton esetén nagyon különböző ideig tartózkodhatnak ebben a ciklusban, ezért amint az egyik szál végzett a ciklussal, annak addig kell futás nélkül várnia, amíg a multiprocesszorban lévő összes többi szál is befejezi a futást. Ez értelemszerűen nagyon csökkenti a hatékonyságot, ezért egy trükkhöz folyamodunk: a ciklus nem csak egy foton követését végzi el, hanem sorban többét. Amint az egyik fotonnal végeztünk, áttérünk a következőre, ilyen módon sokkal kiegyenlítettebb lesz az egyes szálakban a ciklusok lefutásának száma. Ez a központi határeloszlás tétel következménye, és később láthatjuk, a gyakorlatban is jól működik a megfontolás.

A kernel, mely futtatásra kerül a GPU-n, a lokálisan definiált változók inicializálását végzi el, majd közvetlen ezután a

```
while(true) { ... }
```

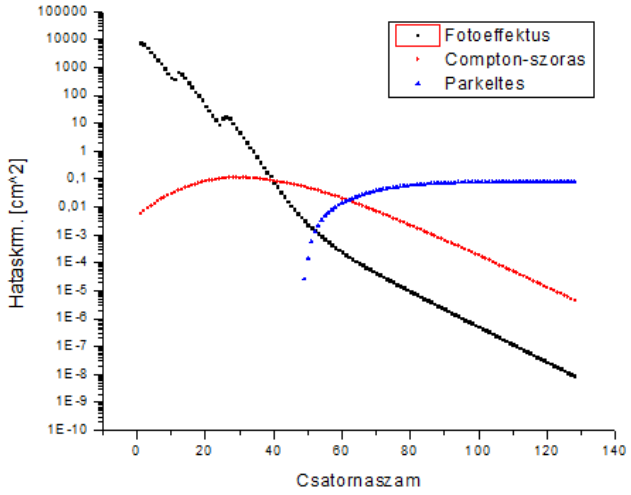
végtelen ciklus következik, melyből a kilépést csak az előre megadott számú foton leszimulálása biztosítja.

Ezek a bizonyos lokális változók közül a legfontosabbak:
gammaRay, energy: Az éppen követett foton pozíciója, iránya, valamint energiája.
gammaPair: A letárolt annihilációs foton pozíciója és iránya (lásd későbbiekben).
detectedE: Az aktuális követett foton által eddig leadott energia.
all: Hanyadik foton szimulálásánál tartunk. Amennyiben ez eléri a beállított értéket (`photonsperthread`), kilépünk a fotonkövető ciklusból.
new_photon_need: Egy flag (`boolean`, azaz igaz/hamis változó), mellyel azt tároljuk, hogy szükséges-e új fotont generálni. Incializáláskor igaz értékű, egyébként pedig akkor válik igazzá, ha az aktuális követett foton fotoeffektust szenved, vagy kilép a kristályból (lásd 4.1 ábra). Hamissá válik az új foton „legyártásakor".
pair_exists: Egy flag, mellyel azt tároljuk, hogy van-e letárolt annihilációs fotonunk (a `gammaPair`-ben). Igazzá válik párkeltés esetén. Hamissá válik az annihilációs foton leszimulálásakor.

4.2 Hatáskeresztmetszetek tárolása

A hatáskeresztmetszet adatokra elengedhetetlen szükség van a szimulációhoz. Mivel minden ciklusban egyszer le kell kérni a hatáskeresztmetszet értékeket az adott energiára, ezért jól megfontoltan kell kitalálnunk a tárolás helyét és módját, mert ez nagyban meghatározza a kód sebességét.
A helyet illetően két lehetőségünk van: konstans memória, vagy textúra a globális memóriában.
A textúra a mi alkalmazásunkra pontosan megfelelő, ugyanis tetszőleges valós indexnek az értékét lekérhetjük, és ezt automatikusan interpolálva kapjuk meg a legközelebbi adatpontokból.
A textúra ezen kívül gyorsítótárazódik a GPU-n belül, ezért gyorsabb eléréseket kapunk, ha közeli indexeket kérdezünk le.
A foton hatáskeresztmetszet adatok nagy energiaintervallumot fednek le, ezért fontos jól megválasztanunk a tárolás módját. A leghatékonyabbnak az adatsorok logaritmikus leskálázását találtam, ugyanis ilyen léptékkel megfelelően simának, egyenletesnek találjuk a görbéket, ami azt jelenti, hogy nem vesztünk adatot az esetleges alul-mintavételezéssel, viszont a tároláshoz szükséges hely töredékére esik vissza.

A hatáskeresztmetszet adatok a már említett cross-section.txt fájlból kerülnek beolvasásra. A szimulációk során NaI adatokkal dolgoztam, melyek előállítását röviden bemutatom. A nyers adatokat XCom (XCom: Photon Cross Section Database, http://physics.nist.gov) programmal generáltattam, ebből kiválasztottam a számunkra érdekes abszorpció, Compton-effektus, valamint párkeltés oszlopokat, és ezeket ábrázoltam. Következő lépésként az adatsorokat interpoláltattam, majd mintavételeztem (a logaritmikus skálán) egyenletes sűrűséggel, ezek a mintavételezett adatpontok kerültek a cross-section.txt-be. 128 érték szerepel a 10^{-3}MeV-től a 10^{5} MeV tartományban a három lehetséges kölcsönhatás mindegyikére. Ez a 3 oszlop az (plusz az összegükként előállított negyedik), mely a `cross-section.txt` fájlt alkotja. A szimulációimban is használt (NaI közegre vonatkozó) értékeket a 4.2 ábrán láthatjuk ábrázolva. A programmal akármilyen összetételű szcintillátor kristályt modellezhetünk, amennyiben a határkeresztmetszet fájlt hasonló módszerrel feltöltjük adatokkal.



4.2. ábra
A kis energiákon domináló fotoeffektus, a küszöbenergiás párkeltés és a Compton-szórás hatáskeresztmetszete NaI-ra

A 128 méretű 1D textúrát a host kód hozza létre, és tölti fel a hatáskeresztmetszet adatokkal (melyeket előzőleg fájlból beolvastunk) a grafikus kártya globális memóriájában a megfelelő CUDA hívásokkal:

```
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float4>();
cudaMallocArray( &d_crossSection, &channelDesc, 128, 1);
cudaMemcpyToArray( d_crossSection, 0, 0, h_csData, sizeof(float4)*128,
cudaMemcpyHostToDevice);
```

A textúra float4 (azaz 4 darab float-ot tartalmazó struktúra) típusú, ez azért előnyös, mert így a három különálló hatáskeresztmetszet mellett a totális (az előbbi három összege) értéket is külön tárolhatjuk, és a grafikus architektúra miatt ez nem jelent többletszámolást, vagy időveszteséget a textúrából való lekérdezésnél. (A grafikában használatos textúrák ugyanis döntően float4 típusúak a megszokott red, green, blue összetevőkön kívül az alpha csatorna tárolására, utóbbi az átlátszóság kezelésére.)
A textúra paramétereit az alábbi kódrészlet állítja be:

```
crossSection.filterMode = cudaFilterModeLinear;    // linear interpolation
crossSection.normalized = true;     // access with normalized texture coordinates
crossSection.addressMode[0] = cudaAddressModeClamp;    // clamp texture
coordinates
```

Lineáris interpolációt alkalmazunk, a textúra elemeit normalizált koordinátákkal érjük el (azaz nem 0 és 127 közötti indexeket kérdezünk le, hanem 0 és 1 közöttieket), ha pedig ennél kisebb, ill. nagyobb értéket kérdeznénk le, a 0 ill. 1-nek megfelelő értéket kapjuk vissza (levágó címzési mód).

Miután beállításra került a textúra, a GPU kódban tudunk adatot lekérdezni belőle:

```
// Get cross section at photon energy
float norm = (LOG10(energy)+3.0f)/8.0f;
float4 cs = tex1D(crossSection, norm);
```

A foton energiáját először transzformáljuk: logaritmusát vesszük, majd shifteljük úgy, hogy 10^{-3}MeV esetén 0-t adjon, 10^{5}MeV esetén pedig 1-et. Ezután a `tex1D()` CUDA hívással a kapott helyen kérdezzük le a textúra értékét, mely automatikusan interpolált érték lesz.

A CPU-n futtatott MC kódban egyszerűen a fájlból beolvasott tömböt használjuk, manuálisan interpoláljuk ki a hatáskeresztmetszet értéket, mivel ott nem áll módunkban CUDA függvényeket használni.

4.3 Kölcsönhatások kezelése

A gamma sugárzás az anyagban való terjedés során háromféle kölcsönhatáson keresztül adhat át energiát a közeg elektronjainak, melyek kinetikus energia formájában veszik fel azt.

A kis energiákon domináló fotoeffektus során a foton egy atom elektronfelhőjében elnyelődik, teljes energiáját egy ún. fotoelektronnak leadja. A fotoelektron kötési energiáját jóval meghaladó energiarészt kinetikus energiaként viszi magával. Ezt egyetlen sorral modellezhetjük:

```
detectedE += energy;
```

Mivel ezzel a fotonnal ilyen formán végeztünk, ezért „igényeljük” a következőt:

```
if (!handlePair(pair_exists, gammaRay, gammaPair, energy)) {
        new_photon_need = true;
}
```

A *`handlePair()`* függvényről a párkeltésnél lesz szó.

A Compton-szórás során a foton egy elektronnal rugalmasan ütközik, ezt a kölcsönhatást a következő sorral kezeljük le:

```
detectedE += doCompton(gammaRay, energy, seed);
```

A *`doCompton()`* függvény valósítja meg a Compton-szórás modellezését, bemenő paraméterei a foton hely- és irányvektora, valamint az energiája, visszatérési értéke pedig a közegnek átadott energia. A függvény referenciaként fogadja a paramétereket, hiszen a szóródás után megváltoznak az értékek. A szimulációm során a Kahn-metódust használtam a Klein-Nishina eloszlás mintavételezésére. Az ismert rejekciós eljárás során három véletlen számot generál az algoritmus, és csak akkor lép tovább, ha ezek eleget tesznek az energiafüggő feltételeknek.

A legösszetettebb anyaggal való kölcsönhatás a párkeltés, ekkor a foton egy atommag közelében való elhaladás során elektron-pozitron párt kelthet, mely

közben a foton eltűnik. Az elektron és antirészecskéje, a pozitron nyugalmi energiája 511keV, tehát a fotonnak legalább kétszer ekkora energiával kell rendelkeznie, hogy a párkeltés megvalósulhasson. Ennél nagyobb energia esetén a keletkezett részecskepár kinetikus energia formájában viszi el a többletenergiát. A pozitron az anyagban rövid távon nulla sebességre lassul, és elektronnal találkozik, mikor is annihiláció során megsemmisül, és egymással ellentétes irányban két 511keV-es gamma kvantumot sugároznak ki.
A modellezés során tehát az eredetileg egy, nagyenergiás fotonból kettő, 511keV-es foton marad hátra, mely mindkettőt végig kell követnünk a teljes detektált energia meghatározásához.

```
detectedE += energy - 2 * REST_E;
gammaRay.d = isotropDirection(seed);
pair_exists = true;
gammaPair.o = gammaRay.o;
gammaPair.d = -gammaRay.d;
energy = REST_E;
```

Mivel az algoritmus egyszerre egy foton követését teszi lehetővé, ezért a keletkező két annihilációs fotont egymás után, sorban fogjuk leszimulálni. A fenti kódsorokkal beállítjuk a követendő foton paramétereit az egyik annihilációs fotonra, és beállítjuk a `pair_exists` flaget, valamint letároljuk `gammaPair` változóba a másik annihilációs foton induló helyét, és irányát.
Immár értelmet nyer az új foton „igénylésére” vonatkozó kódrészlet:

```
if (!handlePair(pair_exists, gammaRay, gammaPair, energy)) {
        new_photon_need = true;
}
```

Ahol a `handlePair()` függvény implementációja pedig:

```
if (!pair_exists)
  return false;
energy = REST_E;
gammaRay = gammaPair;
pair_exists = false;
return true;
```

Magyarán amennyiben be van állítva a `pair_exists` flag, új foton generálása helyett a még leszimulálásra váró annihilációs foton paramétereit töltjük be, és ezzel folytatódik a szimulációs ciklus. A `pair_exists` flag természetesen resetelésre kerül, hogy amikor a második annihilációs fotonnal is végeztünk, már ténylegesen a `new_photon_need` flag állítódjon be, mely új foton generálását triggereli.

Amikor új fotont szükséges generálnunk (lásd 4.1 ábra „Új foton generálása kezdeti értékekkel”), a fotonnak kezdeti irányt kell sorsolnunk. Mivel térben izotróp a forrásunk, ezért a kenyérszeletelő tétel alapján sorsolom ezt a bizonyos véletlen irányt.

Az MC szimuláció során elengedhetetlen véletlen számok generálása. A kódom LCG-t, azaz lineáris kongruenciális generátort használ. Az alábbi `getRandom()` függvény valósítja meg egy 0 és 1 közötti véletlenszám lekérését, és láthatóak a futtatásaimhoz használt konstansok is[5]:

```
// LCG generator parameters
#define G                   3249286849523012805l
#define C                   11
#define M                   9223372036854775808l

// LCG random generator
float getRandom(unsigned long long int &seed) {
  seed = (seed * G + C ) & (M-1);
  return (float)seed / M;
}
```

Láthatóan `unsigned long long int`, azaz 10^{19} nagyságrendű számokkal dolgozik az algoritmus, ami $>10^{14}$ periódusú LCG-t eredményez[5]. További fejtörésre ad okot, hogyan használjon minden szál a GPU-ban különböző véletlenszám sorozatokat. Egy egyszerű, de a tapasztalat alapján hatékony megoldást választottam. Az induló értéke a `seed` változónak minden szálban legyen különböző, a konkrét megvalósításomban eggyel növekvő egészek. Az első szálban tehát 0 a kezdő érték, a másodikban 1, és így tovább. Ilyen módon egy futó kernel minden szálának különböző a `seed`, az LCG generátor kezdőértéke. Természetesen elképzelhető (tegyük hozzá, igen kis valószínűséggel), hogy a `seed` valamennyi lépés után egy másik szál által kapott kezdőértéket vesz fel, és innentől ugyanazokat a véletlenszámokat kapja, mint az a bizonyos másik szál az elején, viszont elhanyagolhatóan kicsi az esélye, hogy pont ugyanolyan állapotban lesz a két, azonos véletlenszámokat generáló szál. (A szimuláció ugyanazon lépésénél, ugyanolyan kezdőértékek mellett tartózkodjon, stb. Hiába kapják ugyanazokat a véletlenszámokat, ha teljesen más lépésekhez használják fel azokat, tehát szinte teljesen korrelálatlanok maradnak.)
A tapasztalati hatékonyság a generált spektrumok folytonos volta. Minél több fotont indítunk, annál simább (kisebb szórású) spektrumot kell kapnunk. A szimulációim szerint ennek tökéletesen eleget tesz a kód. Amennyiben nagyfokú korreláltság lépne fel a szálak között, egyes csatornákon több beütést észlelnénk, mint a szomszédosakon, ami éppen ezt a folytonosságot borítaná fel.

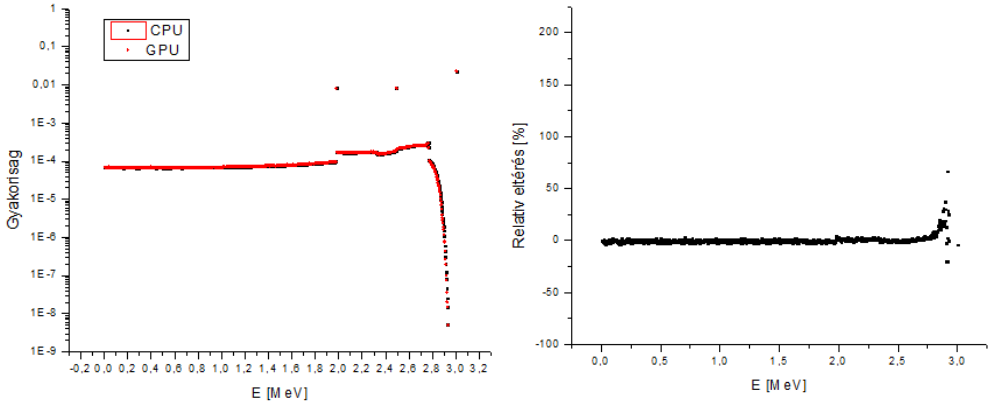
Amikor a 4.1 ábrán feltüntetett „detektált energia eltárolása” lépéshez érkezünk, a `detectedE` lokális változó tartalmát kell elmentenünk, ugyanis a `detectedE` közvetlen ezután nullázásra kerül, hogy egy újonnan induló foton által leadott energiát kezdje számlálni.
Ha azonban csak a detektált energiát mentjük el, azután azt a CPU kódnak kell a sokcsatornás analizátor megfelelő csatornaszámára konvertálnia, így logikusabb lépés ezt a számolást is a GPU-ra hagyni, és rögtön a csatornaszámot elmenteni.
A mentésre két kézenfekvő megoldás kínálkozik. Mivel a GPU kód csak a videokártya device memóriáját használhatja, ezért mindkét megoldás erre épít. Az egyik szerint egy 1024 nagyságú tömböt foglalunk le (és nullázunk ki) a device memóriában, az 1024 csatorna mindegyikének, majd amikor a szál elmenti az eredményt, akkor az a megfelelő „csatornán” megnöveli eggyel az értéket. Azonban amikor egyszerre két szál szeretné kiírni az eredményt pontosan ugyanabba a csatornába, akkor vagy (attól függően, hogy atomi műveletként kezeljük-e le az írást) a szálaknak várniuk kell egymásra, vagy az egyik írás nem hajtódik végre. Mindkét jelenség jelentős hatással van a kimeneti spektrumra, valamint a futási sebességre, így ez a megoldás nem optimális.
Egy jobb megoldás, ha minden indítandó foton számára lefoglalunk egy egésznyi (integer) helyet a memóriában, és az adott foton „eredményét” csatornaszámra transzformálva ide írjuk be. A módszer hátránya, hogy akkora memóriaterületet kell lefoglalnunk, amennyi fotont az egész kernel során indítunk (ez 10-100MB nagyságrendű device memóriát jelenthet). Ráadásul a CPU-ra hárul a feladat, hogy a kernel futása után végigmenjen ezen a „becsapódáslistán”, és összesummázza a becsapódásokat immár egy 1024 elemű spektrumba.
A kódban ez utóbbi megoldást implementáltam.

5. A kód teljesítménye

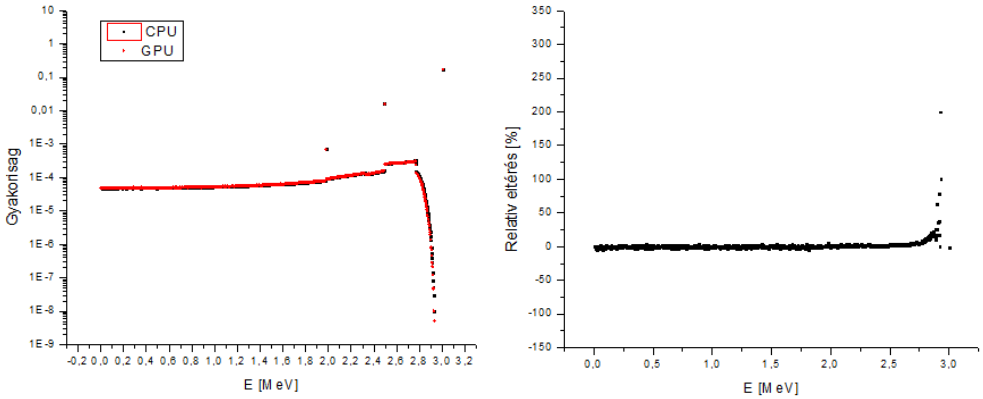
5.1 Spektrumok validálása

A szoftverfejlesztésnek mindig is komoly részét képezte a megírt kód tesztelése, helyességének ellenőrzése. Ezt az ellenőrzést esetünkben sem hagyhatjuk el, hiszen alapvető fontosságú, hogy a szimuláció a keretein belül pontos legyen, valamint megfelelő eredményeket szolgáltasson.
A validálás esetemben ún. működési teszt (functional test) volt, mely az elkészült program, mint egész számára támaszt követelményeket. Magyarán a program megadott paraméterezése esetén generált spektrumot vetjük össze egy elvárt spektrummal.

Az első összehasonlítást érdemes a GPU és CPU által generált spektrumon végezni. A CPU tekinthető hiteles számítást végző eszköznek, és arra vagyunk kíváncsiak, hogy a GPU az elvárásainknak megfelelően hajtja-e végre a kernelt.
A CPU és GPU kód között lényegi eltérés a hatáskeresztmetszet lekérésben van, valamint a használt matematikai függvények eltérő implementációjában. A két spektrum közti eltéréseket tehát ezekre az okokra kell tudnunk visszavezetnünk. Amennyiben nem tudjuk, de a kódunk helyes, abban az esetben valamilyen hibát találtunk a CUDA környezetben. Szerencsére egy több éves, bejáratott rendszerről van szó, így ez utóbbira kicsi az esélyünk.
Két eltérő esetben futtattam le a szimulációt, egyszer egy 4x4x4cm-es kristály mellett, majd egy 30x30x30cm-es kristály mellett. A forrás első esetben 0.1cm, a másodikban 5cm-re helyezkedett el a kristály egyik lapközepétől. Mindkét esetben azonos volt a forrás energiája: 3MeV, valamint a kristály sűrűsége: 3.67g/cm^3. Az 5.1 és 5.2 ábrán látható a CPU és GPU kód által generált spektrum, valamint a csatornákon mért beütés-gyakoriság relatív eltérése. Láthatóan nagyon jól illeszkedik a két spektrum, az eltérés ott válik jelentőssé, ahol csökken a beütésszám, ami a statisztikai ingadozást mutatja. Az eltérésből azonban egyértelműen kiolvashatunk egy pozitív irányba való ívelést. A tartomány, ahol konzisztensen eltér a CPU és GPU spektruma, a Compton-él utáni tartomány, aminek megjelenése a kisenergiás fluxussal van kapcsolatban. Vizsgálatot végeztem konstans, „bekódolt” értéket használva hatáskeresztmetszetnek, ahol nem jelentkezett ez az eltérés. Ez azt jelenti, hogy az eltérés a hatáskeresztmetszet eltérő lekérdezése miatt jelenik meg.

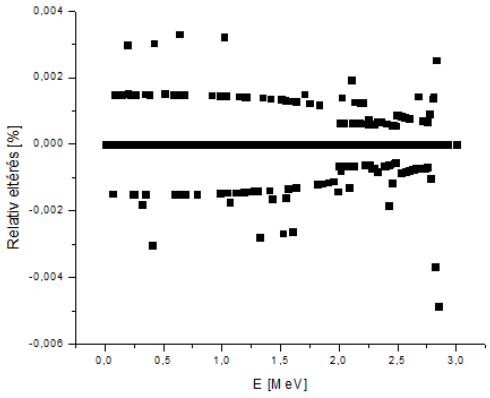


5.1. ábra
A CPU és GPU kód segítségével szimulált spektrum 4x4x4cm-es kristályban, valamint a spektrumok eltérése



5.2. ábra
A CPU és GPU kód segítségével szimulált spektrum 30x30x30cm-es kristályban, valamint a spektrumok eltérése

A GPU kódban lehetőség van használni a standard pontosságú matematikai függvényeket (sin, cos, pow, log), ugyanakkor implementáltak sokkal gyorsabb működésű, cserébe azonban pontatlanabb változatait is ezen függvényeknek. Az 5.3 ábrán a standard, valamint a gyorsított függvényeket használó GPU kódok által generált spektrumok közti relatív eltérést láthatjuk (százalékban!). Ez a százezrelékekben mérhető eltérés teljes mértékben elhanyagolható, tehát jogosan használhatjuk a gyorsított változatú függvényeket.



5.3. ábra
A GPU kódban használt gyorsított matematikai függvények hatása a spektrumra elhanyagolható

Miután meggyőződtünk, hogy a GPU és CPU kód jó egyezéssel ugyanazt az eredményt szolgáltatja, következhet az összevetés egy hitelesnek tekintett szimuláció eredményével. Az MCNPX széles körben használt és elismert program Monte Carlo szimulációkhoz, így ezt használjuk „etalonnak”.

Ugyanazt a modellkörnyezetet használtam, mint az előbbi összehasonlításban, ez alapján készítettem el az alábbi MCNPX bemeneti fájlt (ez a 4x4x4cm-es kristályhoz tartozik, analóg módon néz ki a másik esetben is):

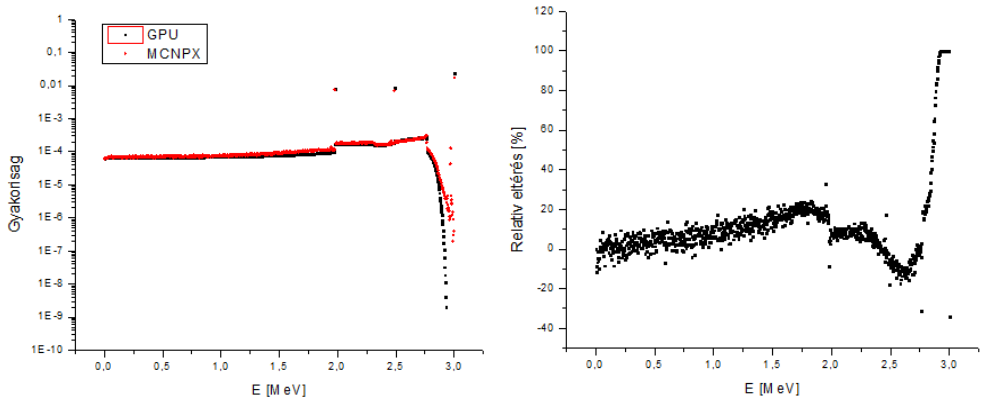
```
c cellák
1   1 -3.67     1 -2 3 -4 5 -6   imp:p=1
2   0          #1 -7            imp:p=1
3   0          7                imp:p=0

c felületek
1 px -2
2 px 2
3 py -2
4 py 2
5 pz -2
6 pz 2
7 so 1000

c minden egyéb
mode p
M1 11000 5 53000 5
SDEF POS -2.1 0 0 ERG=3
NPS 1E7
F8:p 1
E8 0 1023i 3.096
```

Az MCNPX kimeneti spektrumnak az összevetése a GPU által generált spektrummal az 5.5 és 5.6 számú ábrákon látható. A relatív eltérésben észlelhető véletlen szórás sokkal erősebb, mint az előző esetben. Ennek oka az MCNPX által szimulált relatív kis számú részecske (ami így is jelentős ideig tartott), ami a statisztikai szórást nagyban növeli. Észrevehetünk azonban szisztematikus eltérést a két spektrum között, egyrészt a Compton-háton a felfelé ívelő jelleget, másrészt a fotocsúcs előtti energiákon igen nagy pozitív irányú eltérést. A relatív eltérés 100%-nál „telítődik”, ennek oka a GPU spektrum nulla beütésszámú csatornái. A két kiszökési, és a teljesenergiás csúcsoknál fellépő relatív eltérést az 5.4 táblázatban foglaltam össze.

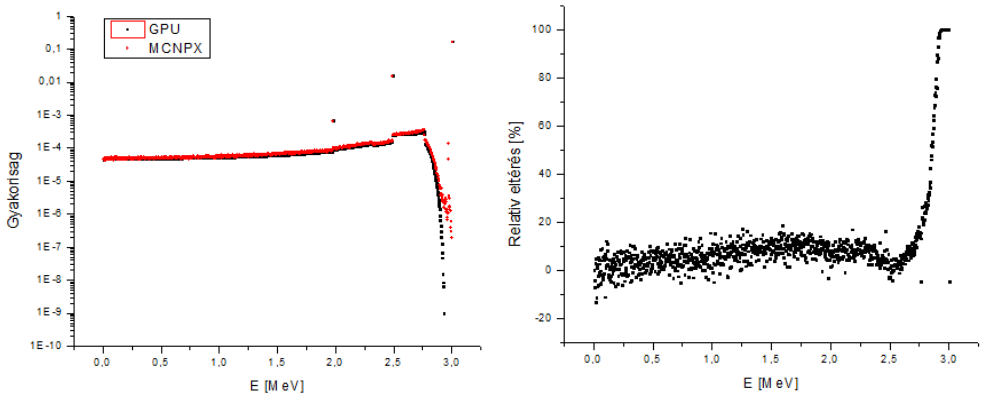
	1. kiszökési csúcs [%]	**2. kiszökési csúcs [%]**	**Teljesenergiás csúcs [%]**
4x4x4cm kristály	*-8.7*	*-18*	*-34.37*
30x30x30cm kristály	*-28.18*	*-20.6*	*-0.13*

5.4. táblázat
MCNPX és a GPU kód által generált spektrumok csúcsainak eltérései

A fotocsúcs előtti csatornákon a nagy relatív eltérés valójában igen kicsi abszolút eltérés, mert az itteni beütésszámok nagyságrendekkel kisebbek a Compton-hát beütésszámainál. Ennek az eltérésnek az oka valószínűleg az, hogy az MCNPX ún. detailed physic módban számol, azaz tartalmazza a fluoreszcens fotonokat, és a fékezési sugárzást is. A fluoreszcens fotonok okozzák a diszkrét csúcsokat, a fékezési sugárzás pedig folytonos spektrumú, ami rászuperponálódik az alap spektrumra.
A több zavaró tényező, és a hatáskeresztmetszet adatok nem egységes volta miatt ilyen formán elfogadhatónak tartom az eltérést, azonban további vizsgálatot érdemel az ügy. Az további vizsgálatához pedig szükséges lenne megbizonyosodni a használt hatáskeresztmetszet adatok egyezéséről, mely után tudnánk csak további következtetéseket levonni.



5.5. ábra
A GPU kód és MCNPX segítségével szimulált spektrum 4x4x4cm-es kristályban, valamint a spektrumok eltérése



5.6. ábra
A GPU kód és MCNPX segítségével szimulált spektrum 30x30x30cm-es kristályban, valamint a spektrumok eltérése

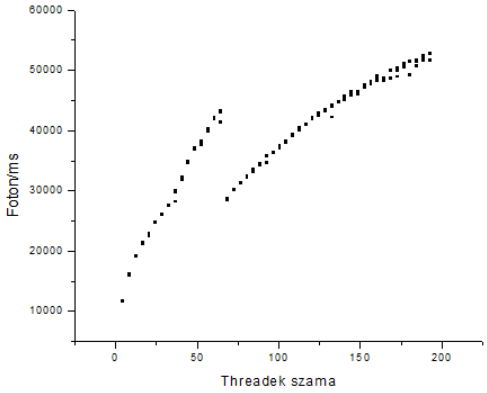
Az MCNPX egy nagymértékben általános szimulációs kód, így nem „szép dolog” a teljesítményét összehasonlítani egy speciális esetre írt programmal, az érdekesség kedvéért mégis megteszem. Az MCNPX-nek ~5 percbe telt leszimulálni a 10^7 fotont egy átlagos asztali gépen, míg egy átlagos GPU ~0.5s alatt végezne ezzel a számolással. Ez érzékelteti, hogy egy-egy speciális modellkörnyezet esetén mennyire lényeges lehet speciális program készítése (speciális hardverre).

5.2 Paraméterek hatása a teljesítményre

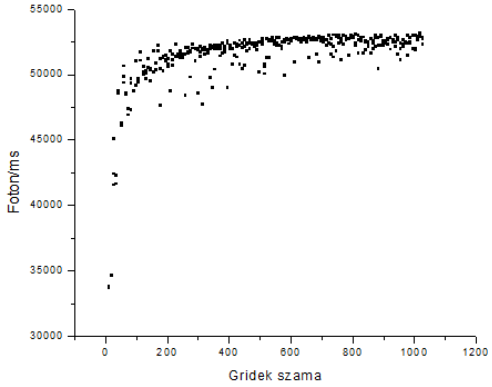
A szimuláció grafikus kártyán való futtatásának paraméterei nagy hatással vannak a számítási teljesítményre, ezért érdemes részletesen megvizsgálni ezeket. Ebben a fejezetben a teljesítményt, mint kapacitást használom, melynek mértékegysége: szimulált foton/időegység.
Ezek a paraméterek egyrészt a grafikus kártyán indított kernel paraméterei: thread és grid szám, valamint, hogy hány fotont követünk egy szálon belül (foton/thread), és a kernel hívások ismétlésének száma (repeat).

A fotonkövetés sebességét a fenti paraméterek függvényében az 5.7-5.10 ábrák mutatják. Mindegyik adatsoron telítési görbe jelleget tapasztalhatunk.

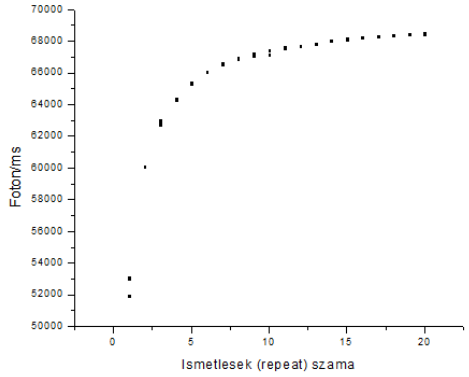
A legérdekesebb görbe az 5.7 ábrán figyelhető meg. Láthatóan a GPU architektúrája igen érzékeny a kernelben indított szálak számára, és esetünkben egy igen jelentős letörést produkált 192-ről 196 szálra való ugrás során.
A telítési görbe jelleget sok mindennek köszönhetjük: az inicializálásokhoz szükséges konstans időtartam relatív egyre kisebb a teljes futás (inicializálás+futtatás) idejéhez képest, ami egyre nagyobb teljesítményt jelent, nagyobb ismétlésszám esetén nagyobb valószínűséggel gyorsítótárazódnak be használt memóriaterületek, stb.



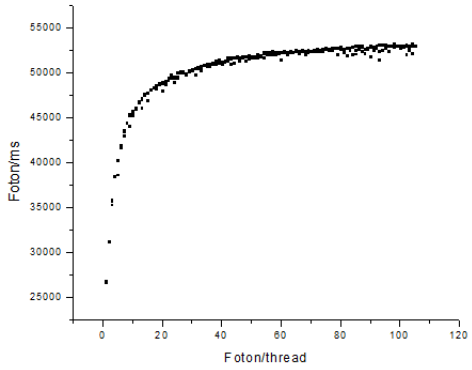
5.7. ábra
A fotonkövetés sebessége
letöréseket mutat az indított szálak függvényében



5.8. ábra
A fotonkövetés sebessége
növekszik az indított gridek számával



5.9. ábra
A fotonkövetés sebessége
nő az ismétlések számával



5.10. ábra
A fotonkövetés sebessége
nő az egy szálon szimulált fotonok számával

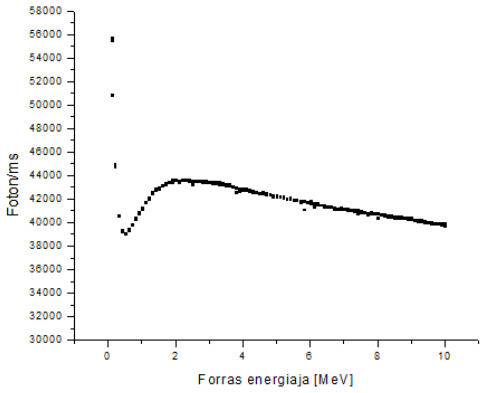
Az itt megtapasztalt összefüggéseket arra tudjuk felhasználni, hogy szimulációink során optimális paramétereket használhassunk a futtatáshoz. A telítési görbék felfutása alapján kijelenthetjük, hogy a szimulációkat legalább 200 grid, és 64 vagy 192 thread mellett, valamint egy szálon legalább 50 fotont követve, legalább 10 ismétléssel kell futtatnunk, így használhatjuk ki maximálisan a hardver nyújtotta lehetőségeket.

A szimuláció fizikai paramétereinek változtatása is természetesen hatással van teljesítményre. Fizikai paraméter a szcintillációs tömb pozíciója és mérete, annak sűrűsége, valamint a pontforrás térbeli elhelyezkedése, és a kibocsátott fotonok energiája. Ugyan ezeket nincs értelme „optimalizálni”, hiszen eleve egy megadott

modellkörnyezetet akarunk szimulálni, mégis érdekes következtetéseket tudunk levonni, ha megvizsgálunk egy-egy ilyen esetet.

A legtöbb paraméter esetén könnyen érthető eredményt kapunk. Például minél távolabb helyezkedik el a forrás a kristálytól, annál nagyobb a fotonkövetés sebessége, hiszen egyre kisebb az esélye, hogy a foton egyáltalán eltalálja a kristályt, így sokkal kevesebb az egy kilőtt fotonra eső kölcsönhatások száma.
Ugyanezt a következtetést vonhatjuk le a kristály méretéből is, egy kisebb kristályból könnyebben megszökik kölcsönhatás nélkül a foton, így szintén nagyobb a teljesítmény.

Érdekes azonban megvizsgálni a gamma forrás energiájának függvényében a számítási kapacitást, ahogy azt az 5.11 ábra is mutatja. A legtöbb fotont időegység alatt 200keV alatt tudjuk szimulálni, ennek oka az alacsony energiákon nagy abszorpciós hatáskeresztmetszet, ami miatt nagy valószínűséggel már az első lépések során elnyelődik a foton.
További érdekesség a lokális minimum 0.5MeV-nél, valamint a lokális maximum 2.3MeV energiánál.
Mivel a számítási kapacitás közelítőleg fordítottan arányos az egyes fotonok átlag-élettartamával (kristályban bekövetkezett kölcsönhatások számával), ezért ebből a görbéből messzebb menő következtetéseket is le lehet vonni.

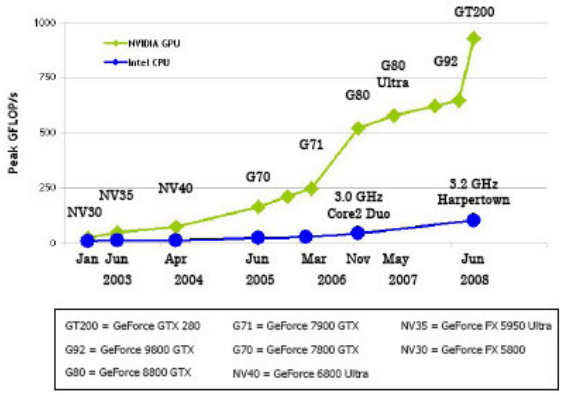


5.11. ábra
A fotonkövetés sebessége
a fotonok energiájának függvényében
lokális minimumot és maximumot produkál

5.3 GPU/CPU teljesítmény

Az eddigiekben beláthattuk, hogy a grafikus kártyán történő szimuláció megoldható, működőképes, és megfelelően működik, valamint a számítási kapacitását megismerhettük különböző paraméterek mellett. Hátra van még viszont az egyik legfontosabb tényező, mégpedig a CPU-val való sebesség összehasonlítása.
Ha ugyanis nem tudunk lényeges sebességnövekedést elérni, akkor felesleges munkát fektetni a GPU kód megírására. (Természetesen még ebben az esetben is felmerülhet az a megoldás, hogy a GPU és CPU is osztozik a számoláson, így is csökkentheti a csak CPU megoldás futásidejét.)
A 5.12 ábrán az nVidia GPU-k, valamint az Intel CPU-k számítási teljesítményei vannak feltüntetve Gflop/s egységekben (10^9 lebegőpontos művelet másodpercenként). Ez a számítási kapacitás a maximum elérhető, ennél a programunk gyakorlatban csak rosszabb eredményt érhet el, ha például nem tud

minden szál párhuzamosan futni, vagy ha memóriaműveletre kell várnia a processzornak.



5.12. ábra[1]
Az asztali GPU-k nagyságrendekkel
nagyobb teljesítményűek, mint a CPU-k

A GPU/CPU arányt természetesen egy adott megvalósításra/kódra kell vizsgálni, amennyiben optimálisan valósítottuk meg a GPU és CPU kódot is, közel kell kerülnünk az elméletben elérhető teljesítmény arányhoz.
Ennek érdekében készült el a CPU kód is, mely szinte teljesen megegyezik a GPU kóddal, a különbségek csak a GPU specifikus megoldásokban vannak.
Hatáskeresztmetszet számolása az ismertetett textúra megoldás helyett egyszerű tömbben történik a két legközelebbi elem lineáris interpolációjával. A CPU kód egyetlen fotont szimulál, az egy másodperc alatt lefutó hívások száma megadja a foton/s arányt a CPU-ra.
Ez az eredmény már közvetlenül összehasonlítható a GPU foton/s számértékével. Annyi korrigációt vehetünk figyelembe, hogy a CPU kód egyetlen szálon, mindenféle többszálúsítást nélkülözve fut, így a CPU-ban lévő magok számával felszorozva a kapott értéket egy felső határt kapunk egy több szálon futó (és ezzel a CPU teljesítményét teljesen kihasználó) megoldás által elérhető értékre.

Két konfiguráción volt lehetőségem próbára tenni a programot, az egyik gép GeForce 8800GT típusú videokártyával, valamint Intel Core2 Duo E8200 processzorral, a másik pedig egy élvonalbelinek számító GeForce GTX260 videokártyával, valamint Intel Pentium 4 processzorral rendelkezett.
A validálás során is megismert két modellkörnyezetre futtatva a szimulációt, a kapott fotonkövetési sebességeket az 5.13 táblázatban foglaltam össze (az eredmények 3 jól egyező futtatás átlagából adódtak).

	Intel Core2 Duo E8200 [foton/ms]	**GeForce 8800GT [foton/ms]**	**Intel Pentium 4 [foton/ms]**	**GeForce GTX260 [foton/ms]**
4x4x4cm kristály	*2608*	*39419*	*1302*	*42228*
30x30x30cm kristály	*1598*	*22886*	*819*	*41742*

5.13.táblázat
Különböző CPU-k és GPU-k esetében
a számítási kapacitás

Az összehasonlítás alapjául az egyes eszközök kereskedelmi árát célszerű választani, hiszen ez adja meg, hogy egységnyi összegből mekkora számítási kapacitást tudunk összerakni.

A szakdolgozat készítésének időpontjában az E8200 és GTX260 ára közel 40e Ft, míg a Pentium 4 és 8800GT ára szintén hasonló, kb. 20e Ft.
Ez alapján az 5.14 táblázatban látható számítási kapacitás/ár értékeket kapjuk.

	Intel Core2 Duo E8200 [foton/(ms*eFt)]	**GeForce 8800GT [foton/(ms*eFt)]**	**Intel Pentium 4 [foton/(ms*eFt)]**	**GeForce GTX260 [foton/(ms*eFt)]**
4x4x4cm kristály	*65x2*	*1971*	*65*	*1055*
30x30x30cm kristály	*40x2*	*1114*	*41*	*1043*

5.14. táblázat
Különböző CPU-k és GPU-k esetében
a számítási kapacitás egységnyi árra vonatkoztatva

A fent említett korrigáció miatt a Core2 Duo-nak megelőlegezhetünk közel kétszer ekkora értékeket, mivel ez egy kétmagos processzor, de mi csak egy szálon futtattuk a szimulációt. Azonban ezzel együtt is láthatóan közel egy nagyságrenddel magasabb a grafikus kártyák kapacitás/ár aránya a szimulációmra, ami azt jelenti, hogy hatékony eszközként tudjuk használni a GPU-t Monte Carlo részecsketranszport szimulálására. Amennyiben nagy kapacitás igényű szimulációt szeretnénk futtatni, a grafikus kártyák használatával sokkal kisebb összegből valósíthatjuk meg ugyanazt a számítási kapacitást nyújtó eszközparkot, vagy ha ugyanazt az összeget költjük el a GPU-s eszközparkra, akkor nagyságrenddel rövidül a szimulációk futási ideje, mint a csak CPU-s rendszer esetében. Természetesen plusz munkát igényel a GPU-ra is leimplementálni a szimulációt, de ez a munka a számítási kapacitás igen nagy növekedése miatt jól kifizetődő.

6. Összegzés

A szakdolgozat keretében megismerkedtem a grafikus processzor architektúrájával, a CUDA környezet segítségével pedig működő grafikus kártyán futtatott foton követő Monte Carlo szimulációt valósítottam meg. Teszteket futtattam a kód megfelelőségének, és a számítási kapacitásának ellenőrzésére. A validálás során a támasztott határokon belül teljesített a kód, a tapasztalt teljesítmény szerint pedig egy mai átlagos videokártyán egy nagyságrenddel több fotont tudunk követni időegység alatt, mint egy átlagos CPU-n.
A GPU-k rohamos fejlődésének, és a CUDA környezet folyamatos finomításának hála komoly jövőt jósolhatunk a GPU alapú Monte Carlo részecskekövető szimulációknak. Említést érdemel azonban a szimuláció speciális modellkörnyezetre implementálásának volta. Éppen ez a speciális kód, és az extra-könnyűsúlyú GPU szálak egymást kiegészítve teszik lehetővé ezt a számítási kapacitást. Egy bonyolult és általános célú szimuláció esetében valószínűleg nem mutatkozna ez az előny a GPU részére.
A témával kapcsolatos kutatást a meglévő hiányosságok, valamint fejlesztési lehetőségek további vizsgálatával lehetne folytatni. Ezek közül kiemelkedő a CPU/GPU és GPU/MCNPX spektrumokban felfedezhető eltérések elmélyültebb vizsgálata, valamint további módszerek keresése a GPU kód gyorsítására, elsősorban a Klein-Nishina eloszlás mintavételezése kapcsán.

7. Irodalomjegyzék

[1] nVidia CUDA Programming Guide Version 2.1
[2] nVidia CUDA Getting Started Version 2.1
[3] nVidia CUDA Reference Manual Version 2.1
[4] I. Lux, L. Koblinger, "Monte Carlo Particle Transport Methods: Neutron and Photon Calculations", CRC Press (1991)
[5] Forrest B. Brown, Yasunobu Nagaya, "The MCNP5 Random Number Generator", Trans. Am. Nucl. Soc. (Nov., 2002) (http://mcnp-green.lanl.gov/publication/pdf/LA-UR-02-3782.pdf)

(Mindhárom CUDA dokumentum letölthető a http://www.nvidia.com/object/cuda_develop.html címről.)