Budapesti Mûszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Szélessávú Hírközlés és Villamosságtan Tanszék

Poros plazma kísérletek támogatása multiprocesszoros környezetben

Diplomatervezés 1. - beszámoló

Készítette
Bakró Nagy István

Konzulens
Hartmann Péter & Reichardt András

2014. április 28.

Tartalomjegyzék

Kivonat **5**

1. A mérés **6**
1.1. A mérési elrendezés 6
1.2. A mérendő mennyiségek és a származtatott értékek 6

2. A referenciaként szolgáló program **7**
2.1. A program bemutatása 7
2.1.1. Detektálási algoritmus 7
2.1.2. Mennyiségek származtatása 7
2.2. Implementáció során hozott döntések 7

3. A multiprocesszoros OpenCL környezet **8**
3.1. OpenCL architektúrája 8
3.2. OpenCL programozási modell 9
3.3. Futási környezet bemutatása 11
3.4. Implementációhoz szükséges megfontolások 11
3.5. Memória szervezés 12
3.5.1. Csak globális memória használata 12
3.5.2. Globális memória és adott esetben lokális memória használata 12
3.5.3. Globális memória és minden adódó alkalomkor a lokális memória használata 12

4. Multiprocesszoros program **13**
4.1. Fejlesztőkörnyezet összeállítása 13
4.1.1. Software Developement Kit-ek (SDK) telepítése 13
4.1.2. Eclipse – Integrated Developement Environment 15
4.2. Új (Hello World) projekt létrehozása 15
4.2.1. Empty C project létrehozása 15
4.2.2. Compiler beállítása 16
4.2.3. Linker beállítása 17
4.3. Kód 17

5. Összehasonlítás **18**

6. Összegzés **19**

7. Függelék **20**

7.1. Matlab szimulátor 20

Ábrák jegyzéke **28**

Táblázatok jegyzéke **29**

Irodalomjegyzék **30**

FELADATKIÍRÁS

A modern alacsony hőmérsékletű plazmafizikai kísérletek egy új, érdekes és izgalmas területe a poros plazmák kutatása. Egy elektromos gázkisülésbe helyezett apró (mikrométer méretű) szilárd szemcse a kisülési plazma atomi részecskéivel kölcsönhatva elektromosan feltöltődik. A sok töltött szemcséből kialakuló elrendezésben a szilárdtestfizikai jelenségek széles spektruma figyelhető meg, pl. kristályrács kialakulása, fázisátalakulás, diszlokációk dinamikája, transzport folyamatok, stb. Poros plazmákat jelenleg leginkább alapkutatásokban alkalmaznak, de jelentőségük az elektronikai gyártásban, fúziós reaktorok üzemeltelésében, Teraherz technológiában egyre inkább előtérbe kerül.

A kísérleti adadtgyűjtés és feldolgozás nagyrésze részecske-követő velocimetrián (particle tracking velocimetry) alapul, vagyis első lépésben egy nagysebességű kamera segítségével nagyfelbontású képek készülnek, amely képek segítségével a porszemcsék pontos (a kamera felbontásánál pontosabb) koordinátáit kell meghatározni. A képek elemzése ezidáig csak a mérést követően, hosszú idő alatt volt megvalósítható a vizsgálandó nagy adatmennyiség miatt. A multiprocesszoros környezetek segítségével a feldolgozás gyorsítása lehetséges akár több nagyságrenddel is.

A jelölt feladata, hogy a meglévő kísérleti elrendezés, amely az MTA Wigner Fizikai Kutatóközpont Szilárdtestfizikai és Optikai Intézezetben található, kiegészítésével a mérés közbeni feldolgozással a mérést segítő analízist hajtson végre. Ennek eredményével a mérés előkészítése és elvégzése lényegesen gyorsulhatnak.

A jelölt feladata

- Mutassa be a mérési elrendezést és elemezze a kapott adatokat! (Mutassa be a mérést!)
- Elemezze a lehetséges multiprocesszoros környezeteket, a feladat szempontjából lényeges paraméterek és feladatvégrehajtási elvárások szempontjából!
- Készítsen programot, amely az azonnali (valós idejű) analízisben résztvevő paramétereket számítja ki, a multiprocesszoros környezet kihasználása nélkül!
- Készítsen programot, amely a mérési környezetbe illeszkedve a mérésnél valós időben képes a vizsgált paraméterek megjelenítésére! Mutassa be és elemezze az elkészített programot!
- Hasonlítsa össze a multiprocesszoros és a nem-multiprocesszoros környezetre elkészített programokat erőforrás igény illetve egyéb paraméterek szempontjából!

Irodalom:

- [1] Hartmann P, et. al. ; "Crystallization Dynamics of a Single Layer Complex Plasma"; Phys. Rev. Lett., 105 (2010) 115004
- [2] Hartmann P, et. al. ; "Magnetoplasmons in Rotating Dusty Plasmas", Phys. Rev. Lett. 111, 155002 (2013)
- [3] Hartmann P, Donkó I, Donkó Z; "Single exposure three-dimensional imaging of dusty plasma clusters"; Rev. Sci. Instrum., 84 (2013) 023501/1-5;

Tanszéki konzulens: Reichardt András, egy. tanársegéd
Külső konzulens: Hartmann Péter, PhD., tud. főmunkatárs (MTA Wigner FK, SZFI)

Budapest, 2014.03.10.

Kivonat

hablaty

1. fejezet

A mérés

1.1. A mérési elrendezés

$$W(s) = \frac{A}{1 + 2T\xi s + s^2T^2}. \tag{1.1}$$

1.2. A mérendő mennyiségek és a származtatott értékek

2. fejezet

A referenciaként szolgáló program

2.1. A program bemutatása

2.1.1. Detektálási algoritmus

2.1.2. Mennyiségek származtatása

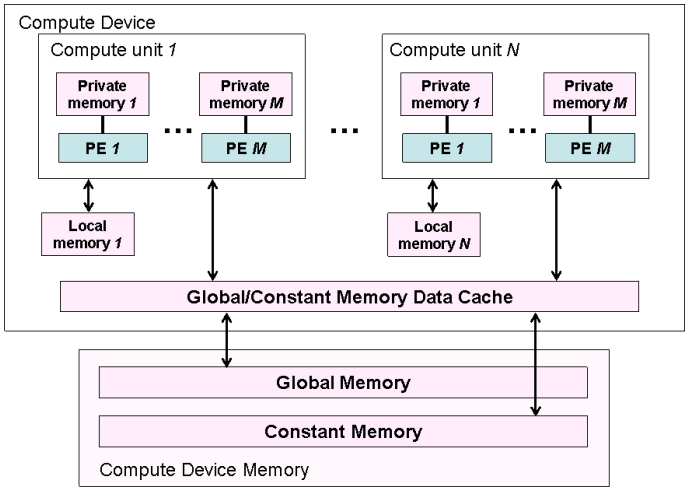
2.2. Implementáció során hozott döntések

3. fejezet

A multiprocesszoros OpenCL környezet

3.1. OpenCL architektúrája

Az Open Computing Language (OpenCL) keretrendszer [4] általános modellt, magas szintű programozási interfészt és hardware absztrakciót nyújt a fejlesztőknek adat- vagy feladat párhuzamos számítások gyorsítására különböző számítóegységen (CPU, GPU, FPGA, DSP, ...). A hárdvergyártók implementálják az OpenCL szabványt, ami által saját platformot hoznak létre. Egy ilyen platformon belüli eszközök alatt főként GPU-kat, de CPU-kat és FPGA-t ... is értünk. OpenCL keretrendszerben történő programozás során két programot kell írnunk. Az egyik a kernel, ami az eszközön futatott szálra fog leképeződni. A másik a gazda processzoron (host-on) futó host-program, ami elvégzi a probléma összeállítását, memória allokálást, argumentumok beállítását illetve a kernel meghívását az eszközön. A kernel futása végeztével a host-program kiolvassa az eszközből a kívánt eredményt.



3.1. ábra. *OpenCL device architektúra [4]*

Az eszközök multiprocesszoros architektúrával és ezek kiszolgálására képes memória architektúrával rendelkeznek, amit a 3.1 ábra vázol. Egy eszköz több compute unit-ot (processzor-magot) tartalmaz. Az OpenCL négy memória szintet különböztet meg, amikre a következőképpen hivatkozik:

- *Regiszterek:* Private memory,
- *Chipen belüli memória (cache):* Local memory,
- *Chipen kívüli memória:* Global memory és Constant Memory.

A regiszterek és lokális memória kis méretűnek és gyors elérésűnek modható, míg a globális memória nagynak de lassú elérésűnek. A memóriákra megkötésként szolgál, hogy ki allokálhat, írhat és olvashat belőle. A 3.1 táblázatban látható ezen jogosultságok.

3.1. táblázat. *OpenCL memória szintek*

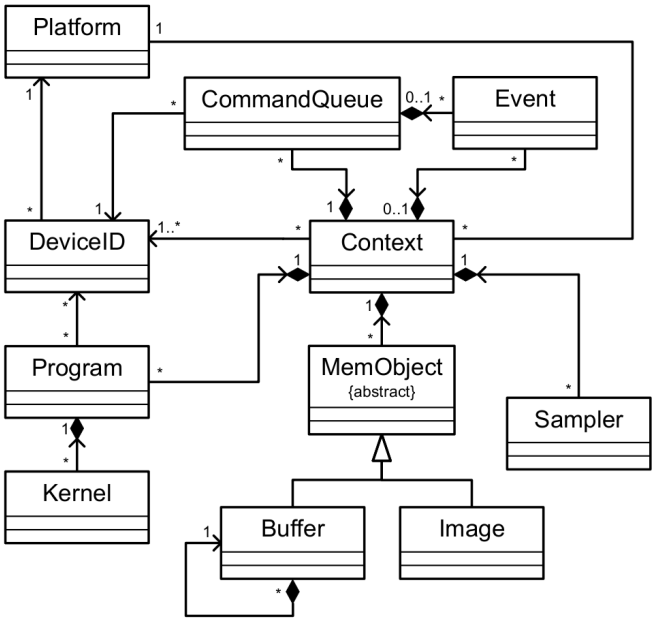
	Global memory	Constant mem.	Local mem.	Private mem.
Host	Dinamikusan R/W	Din. R/W	Din. R/W	
Kernel	R/W	Statikusan R	Satik. R/W	Statik. R/W
Sebesség	Lassú	Gyors	Gyors	Regiszter
Méret	1 Gbyte $<$	$\sim$ 64 Kbyte	$\sim$ 16 Kbyte	$<$ 1 Kbyte

Ahhoz, hogy a rendszerben rejlő teljesítményt kihozzuk három fontos kérdést kell a szimulátor magjának implementálásakor megválaszolnunk:

- *Mennyit?*: Tisztában kell lennünk az aktuális memória fogyasztással és a szükséges memóriamérettel.
- *Honnan-hova?*: Fontos, hogy a lehető legközelebb legyen az adat a processzormaghoz.
- *Mikor?*: Mivel a memória művelet alatt a futtatott kernel nem dolgozik, így átadja a helyét egy másiknak. (Ez Direct Memory Access (DMA) blokk létezése allatt igaz). Ennek a megfelelő szinkronizációjával nagyobb kihasználtság érhető el (load balance).

3.2. OpenCL programozási modell

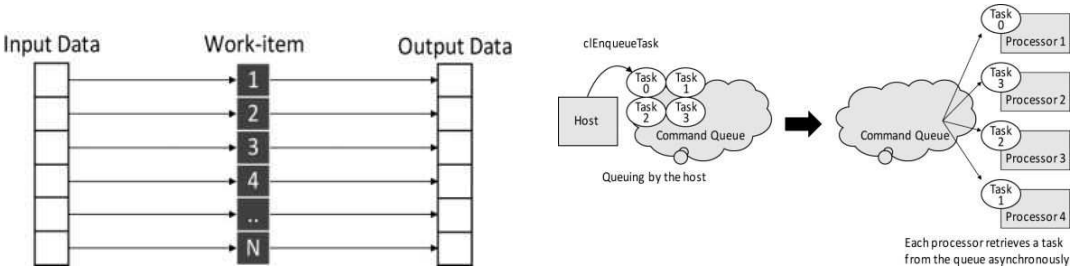
A programozási modell középpontjában a kontextus áll, ami az OpenCL osztálydiagrammján 3.2 figyelhető meg. A futtatáshoz szükséges, hogy a kontextushoz platformot, majd azon belül eszközt, az eszközhöz programot (kernelt) és memóriát rendeljünk. Figyelembe kell vennünk azt a megkötést, hogy csak az egy platformon



3.2. ábra. *OpenCL context osztálydiagrammja [4]*

belüli eszközök programozhatóak heterogén módon. Például: Intel platform esetén lehetséges CPU-t, processzorkártyát és Intel-es GPU-t programozni.

A programozással megoldandó problémát kétféleképpen lehetséges a feldolgozó egységekhez (work-item) avagy processzorokhoz rendelni: adat parallel módon vagy taszk parallel módon. Adat parallel módon (3.3a ábra) a feldolgozandó adat egy egységéhez rendelünk egy feldolgozó egységet. Fontos figyelembe venni az eszköz korlátos számú feldolgozó egységének számát. Ha nem elég a feldolgozó egysége akkor a feladat megfelelő partícionálásával lehetséges kordában tartani a szükséges erőforrás számát. Taszk parallel módot (3.3b ábra) olyan esetben célszerű használna, ha a bemenet dinamikus mérete a futási időben rendkívül változik illetve a végrehajtandó feladat lezán függenek össze.



(a) Adat parallel **(b)** Taszk parallel

3.3. ábra. *Feladat hozzárendelése work-item-hez (processzorhoz)*

A processzor-magok megfelelő kihasználtságának elérése végett több ezer work-item virtuális osztozik rajta. Továbbá ezen work-item-eket work-group-okba rendezzük, ami a lokális memória jogusultsága miatt érdekes. Konkrétan egy work-group-ba

tartozó összes work-item azonos lokális memórián osztozik. Ennek a következménye az, hogy adat parallel módú feldolgozás esetén az egymásra ható adatokhoz tartozó work-item-eket egy work groupba kell rendelnünk. Ha ez nem lehetséges, akkor a globális memóriához kell fordulnunk. A globális memória avagy a bank szervezésű külső (off-chip) memóriák hozzáférési ideje relatíve nagy így ezek használatát lehetőleg el kell kerülni és a programozónak kell „cahchelni" a lokális memóriába.

Összefoglalva nagy hangsúlyt kell a memóriaszervezésre fordítani, hogy a processzormag megfelelően legyenek az adatokkal táplálva.

3.3. Futási környezet bemutatása

3.2. táblázat. *Futási környezetek*

	nVidia GT330M	Intel Xeon PHI
MAX_COMPUTE_UNITS	6	224
MAX_WORK_GROUP_SIZES	$512 \times 512 \times 64$	$8192 \times 8192 \times 8192$
GLOBAL_MEM_SIZE	$1073020928 = 1Gbyte$	$4514811904 = \sim 4Gbyte$
MAX_CONSTANT_BUFFER_SIZE	65536	131072
LOCAL_MEM_SIZE	$16384 = 16Kbyte$	$32768 = 32Kbyte$

3.4. Implementációhoz szükséges megfontolások

A következőkben egy kissebb teljesítményű notebook videókártyát veszek alapul a megfontolások demonstrálására. Ez az nVidia GeForce 330M, 575 MHz-en futó 48 CUDA core-al, 1024GB memóriával és OpenCL 1.0 kompatibilitással. A videókártya továbbiakban fontos paraméterei a **??**. táblázatban látható.

Ha a tér ahol a laplace egyenletet meg kell oldanunk nagyon nagy, akkor érdemes szétbontani kissebb alterekre és azokhoz rendelni egy-egy „work-item"-eket. Mivel a diszkrét laplace egyenlet egy pontja a szomszédos pontokkal szoros kapcsolatban van, így az összefüggő „work-item"-eket egy „work-group"-ba érdemes szervezni, mivel így az átlapolódó pontok értékét a szomszédos „work-item" is tudják írni és olvasni. Az ilyen típusú problémának méretét a MAX_WORK_GROUP_SIZES tulajdonság korlátozza.

Jelen esetben a mérési eredmény egy pontjához tartozó tér átlagosan $11 \times 11 \times 30$ pontból áll. Tehát a korábbi nem áll fenn és egyszerű megfeleltetéssel szétoszthatjuk a feladatot. A teljes tér $512 \times 512 \times 11 \times 11 \times 30$ méretű, ami $951k$ pont. A tárolásához single-precision mellett ennek a számnak a 4-szerese szükségeltetik byte-okban mérva. Mivel ez a videókártyán nem áll rendelkezésre, így szétbontjuk kissebb feladatrészekre.

Ezen feladatrészek méretét egy paraméter állításával lehet változtati és az implementált algoritmus ettől generikusan függ. Emellett az interpoláció mértéke is generikusan paraméterrel állítható. Az algoritmus generikusságát csupán a futási időben történő dinamikus memória allokációval lehetséges megvalósítani. A korábban említettek végett (3.1 táblázat) az allokáció csak a „host" programban történhet.

3.5. Memória szervezés

3.5.1. Csak globális memória használata

Az algoritmus pszeudó kódjának direkt leképezése esetén a „host"-on allokálunk memóriát a „device" globális memóriájában. Majd a megfelelő adatokat ide másoljuk és a kernel is itt ír és olvas. A problémát a globális memória nagy hozzáférési ideje jelenti, ami miatt sok „work-item" tétlenül a memóriára fog várakozni. Ilyenkor az egy mérési pontra vonatkoztatott szimulációs idő a referenciánál is lassabb.

3.5.2. Globális memória és adott esetben lokális memória használata

Kis erőfeszítéssel nagy javulást lehet elérni, ha a mérési ponthoz tartozó szimulációs tér éppen belefér a lokális memóriába. Tehát, mielött az **??** szerinti iteratív megoldót futtatnánk először a globális memóriából a lokális memóriába töltjük át a kérdéses pontokat, majd számolunk rajt és a végén visszatöltjük a globális memóriába. E javítással a referenciával azonos sebességet tudunk elérni.

3.5.3. Globális memória és minden adódó alkalomkor a lokális memória használata

Nagyobb erőfeszítést igényel, hogy a globális memórival való kommunikációt a lokális memória közbeékelésével tegyünk minden alkalomkor. Ezt úgy lehet felfogni, mintha a globális memóriát lokális memória méretű kvantumokban tudnám csak elérni. Ekkor nagy odafigyelést kíván a memóriacímzés megfelelő prgramozása, de eredményképp gyorsulás elérhető.

Összegezve elmondható, hogy az aktuálisan használt adat tárolását a lehető legközelebb kell tartani a „compute-unit"-hoz.

4. fejezet

Multiprocesszoros program

4.1. Fejlesztőkörnyezet összeállítása

OpenCL kód fejlesztése történhet Windows alatt NVIDIA Nsight Visual Studio Edition [5] és Linux alatt GCC-vel [6]. Az Open Source fejlesztőrendszer ingyenessége és az általa generált program hordozhatósága végett a Linux alatti fejlesztés mellett döntöttem. Az OpenCL-t támogató hardverek legtöbbször CPU-k, GPU-k és az Intel MIC [7] kártyái. Ezekre való OpenCL kód fejlesztéséhez a gyártók biztosítanak Software Developement Kit-et (SDK). Ezek telepítése szinte bármelyik Linux disztribúción sikerülhet a megfelelő követelmények előzetes telepítése után. A Linux diszrók közül a CentOS-re [8] esett a választás, ami csupán a fejlesztőkörnyezet egyszerűbb telepítése végett történt így.

4.1.1. Software Developement Kit-ek (SDK) telepítése

nVidia támogatás telepítése

A legtöbb mai Linux disztrók tartalmaznak drivert az nVidia videó kártyákhoz. Ez az open source Nouveau, ami még nem támogatja az OpenCL-t. Így a hivatalos nVidia drivert fel kell telepítenünk. Ehhez először le kell tiltanunk a Nouveau betöltését. Ezt két helyen is meg kell tennünk: egyrészt a `/etc/modprobe.d/blacklist.conf` fájlhoz hozzá kell adnunk a következő sort:

```
blacklist nouveau
```

majd újragenerálni az INITial RAM File System-et (initramfs), ami a rendszer inicializásáért felelős:

```
$ mv /boot/initramfs-$(uname -r).img /boot/initramfs-$(uname -r).img.bak
$ dracut -v /boot/initramfs-$(uname -r).img $(uname -r)
```

másrészt a rendszer indító GRand Unified Bootloader-ben (GRUB) is le kell tiltani a betöltését a kernel opció alábbi paranccsal való kiegészítésével:

```
nouveau.modeset=0
```

Továbbá a telepítéshez szükséges követelményeket a következő parancsokkal telepíthetjük:

```
$ yum groupinstall "Development Tools"
$ yum install kernel-devel kernel-headers dkms
```

Ekkor a rendszer újraindítása után készen állunk a hivatalos nVidia driver telepítésére. A drivert a következő linken lehet letölteni [9]. A grafikus felületet a telepítés idejére le kell állítani az X grafikus kiszolgálót

```
$ init 3
```

paranccsal, majd a konzolban telepíthető a driver, ami a legtöbb munkát elvégzi helyettünk. Ezután az

```
$ init 5
```

paranccsal áttérhetünk a grafikus felületre, ahol a megfelelő környezeti változókat kiegészíthetjük. Legcélratörőbb, ha a `~/.bashrc` fájlt módosítjuk és hozzáadjuk a következő sorokat:

```
PATH=$PATH:$HOME/bin:/usr/local/cuda/bin
export PATH

CUDA_INSTALL_PATH=/usr/local/cuda
export CUDA_INSTALL_PATH

LD_LIBRARY_PATH=/usr/local/cuda/lib64:/opt/intel/opencl/bin
export LD_LIBRARY_PATH

NVSDKCOMPUTE_ROOT=/usr/local/cuda/lib64
export NVSDKCOMPUTE_ROOT

INTELOCLSDKROOT=/opt/intel/opencl
export INTELOCLSDKROOT
```

Mivel az nVidia limitálja a kernel futási időt 5 másodpercben limitálja, hosszabb kernel futási idő esetén a rendszer lefagy. Ezt a korlátozást a `/etc/X11/xorg.conf` fájl `Device` részének a következővel való kiegészítésével érhetjük el:

```
Option "Interactive" "boolean"
```

Érvényre juttatásához az X újraindítása szükséges (CTRL+ALT+Backspace). Ezután nagyobb problémák esetén már nem fogja lefagyasztani a rendszert a watchdog.

Intel támogatás telepítése

A következő oldalról letölthetjük az SDK-t [10]. A kicsomagolás után az `./install-cpu.sh` program futtatásával telepíthető. Ezután még szükséges a `LD_LIBRARY_PATH` beállítása.

4.1.2. Eclipse – Integrated Developement Environment

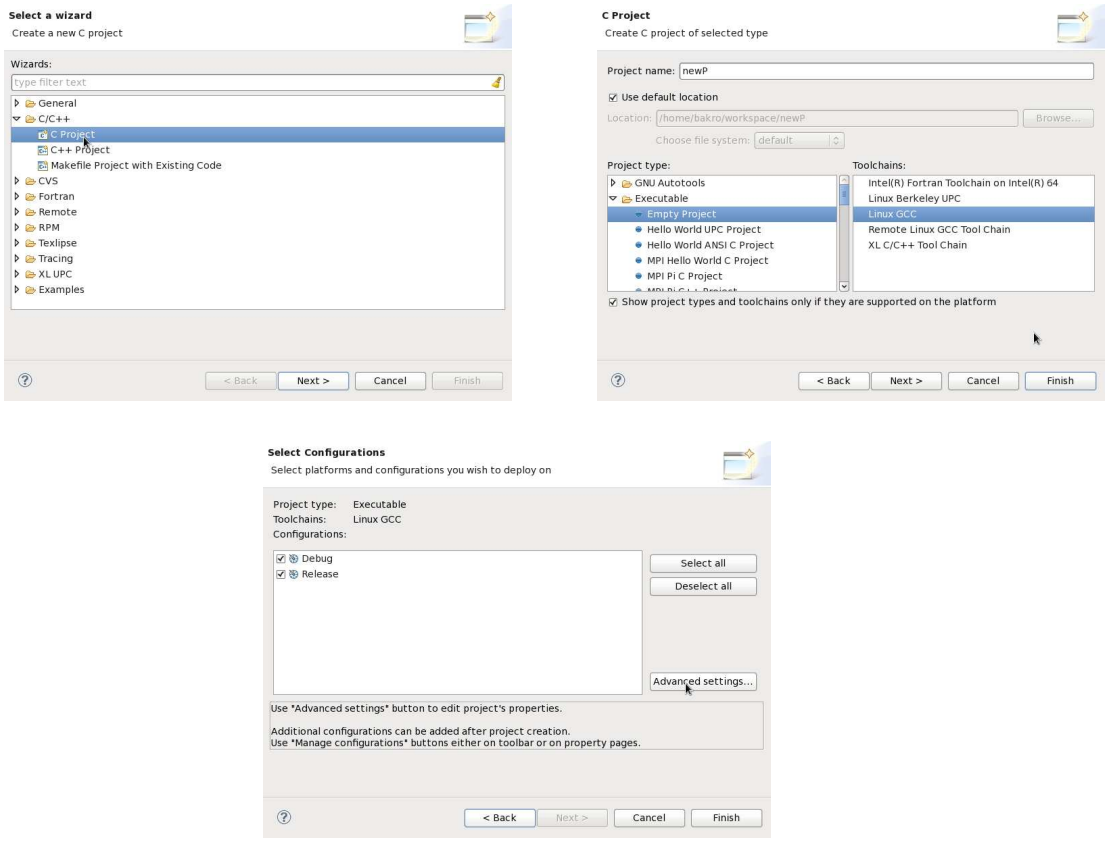
A fejlesztés és hibakeresés egy Integrated Developement Enviroment (IDE) segítségével könnyebb. Az open source Eclipse [11] fejlesztőkörnyezet a különböző pluginjaival épp megfelelő erre a célra. Például a C-nyelv fejlesztését segítő C/C++ Developement Tooling (CDT), a verziókövetést menedzselő EGit és a hibakeresést támogató GDT. A sok Eclipse változat közül az OpenCL fejlesztéshez legjobban az Eclipse for Parallel Application Developers verzió illik, mivel a korábban említett pluginokat már eleve tartalmazza.

4.2. Új (Hello World) projekt létrehozása

Az OpenCL fejlesztését konyhanyelven bemutató OpenCL Programming Guide [12] könyvben szereplő Hello World programot a következő linken lehet letölteni [13]. A kód fordítása elött egy Eclipse projektet létrehozunk és a fordításhoz szükséges beállításokat elvégezzük.

4.2.1. Empty C project létrehozása

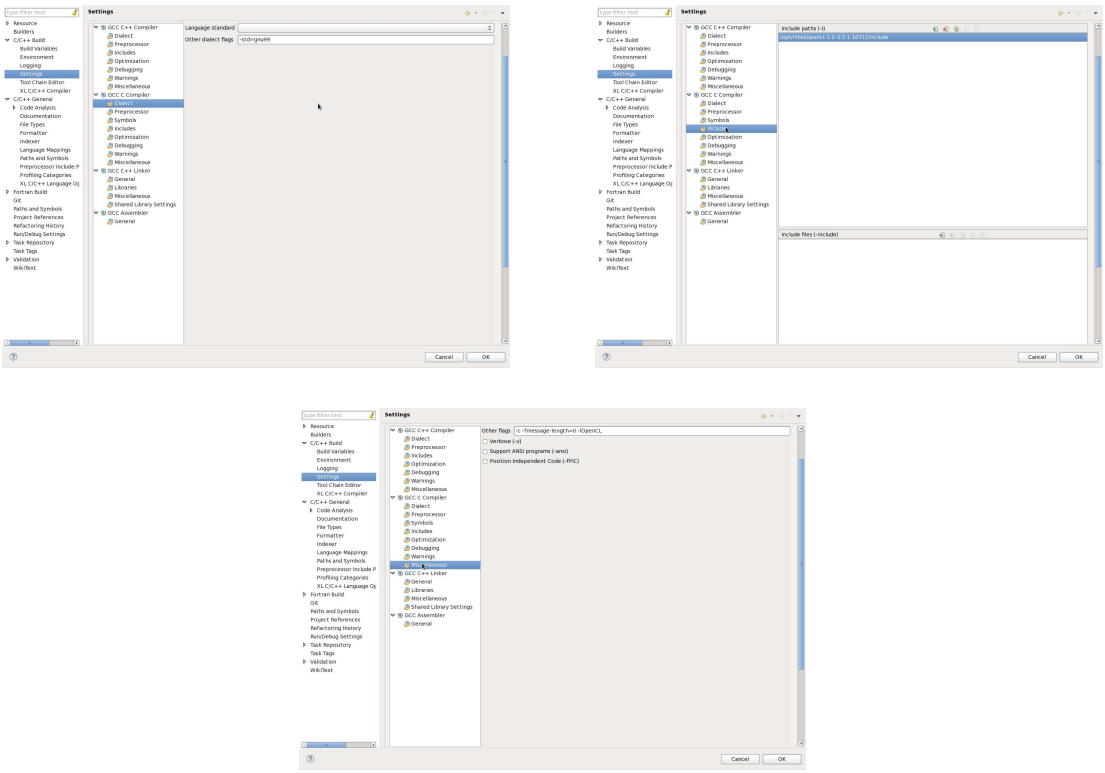
Először egy üres C projektet hozunk létre, ami folyamatát a 4.1 ábrán látjuk. A korábban említettek szerint fordítónak a Linux GCC-t állítjuk be.



4.1. ábra. *Új Eclipse projekt létrehozása*

4.2.2. Compiler beállítása

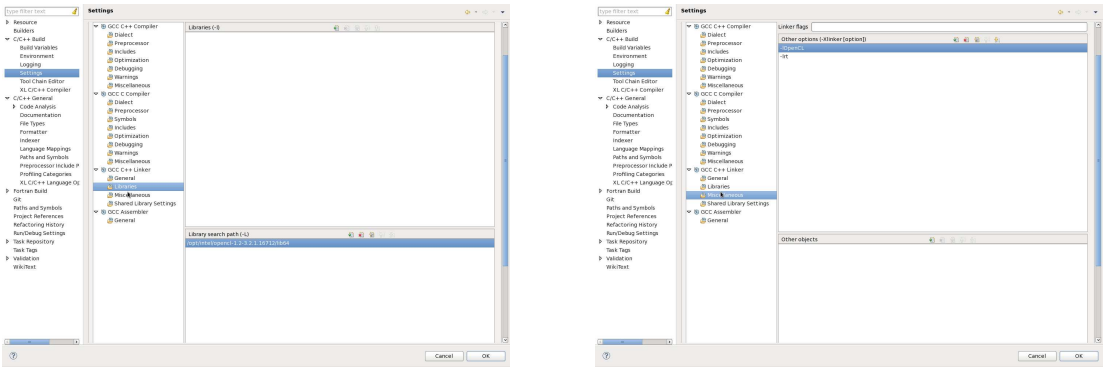
A létrehozott projektre jobb gombbal kattíntva a tulajdonságára kattintva állíthatjuk be a fordítót a képnek 4.2 megfelelően. A beállítások kiterjednek a GNU-C99 nyelv szerinti fordításra és a korábbi 4.1.1 részben telepített SDK-ban található include mappa beállítására.



4.2. ábra. *Compiler beállításai*

4.2.3. Linker beállítása

A linkert a 4.3 ábra szerint állítjuk be.



4.3. ábra. *Linker beállításai*

4.3. Kód

4.1. lista. *A fenti számozott felsorolás LaTeX- forráskódja*

```
sdfas
```

5. fejezet

Összehasonlítás

5.1. táblázat. *Az órajel-generátor chip órajel-kimenetei.*

Órajel	Frekvencia	Cél pin
CLKA	100 MHz	FPGA CLK0
CLKB	48 MHz	FPGA CLK1
CLKC	20 MHz	Processzor
CLKD	25 MHz	Ethernet chip
CLKE	72 MHz	FPGA CLK2
XBUF	20 MHz	FPGA CLK3

6. fejezet

Összegzés

Dolgozatomban ...

7. fejezet

Függelék

7.1. Matlab szimulátor

feszko.m

```
clear all; close all; clc;

L = 1;

N = 500;
dx = L / N;

f0 = 0;
f1 = 1;

IT = 10000;

%%

fi = zeros(IT, N);
fi(:, 1) = f0 * ones(IT, 1);
fi(:, end) = f1 * ones(IT, 1);

T = zeros(N, N);
for n = 2:N-1
    T(n-1, n) =  0.5;
    T(n+1, n) =  0.5;
end
T(1,1) = 1;
T(end,end) = 1;

tic

for i = 2:IT
   fi(i, :) =  fi(i-1, :) * T;
end

toc

figure;
ax1 = subplot(211);
plot(linspace(0, L, N), fi(end, :)); grid;
```

```
xlabel('x');
ylabel('phi(x)');
title('Potencial eloszlas az iteracio vegen');

ax2 = subplot(212);
plot(linspace(0, L, N-1), diff(fi(end, :)) ./ dx);  grid;
xlabel('x');
ylabel('E(x)');
title('Elektromos ter az iteracio vegen');

linkaxes([ax1 ax2], 'x');

mfig(fi(1:1000:IT,1:4:N), 1:1000:IT,1:4:N)
```

feszko2D.m

```
clear all; close all; clc;

X = 500;
Y = 500;

% mindig oszthato legyen 2vel
%global NX NY;
NX = 12;
NY = 12;

dx = X / NX;
dy = Y / NY;

f0 = 0;
f1 =1;


IT = 20000;

%%
ij2n = ((ones(NY,1) * (1:NX)) + ((0:NX:NX*(NY-1))' * ones(1,NX)))';


index = zeros(NX, NY);

index( 1: NX/4, NY/4+1:end) = -5;
index((3*NX/4)+1:end, 1: 3*NY/4) = -5;

index(1: 3*NX/4, 1) = -2;
index(3*NX/4, 1: 3*NY/4) = -6;
index(3*NX/4:end, (3*NY/4)+1) = -2;
index(end, (3*NY/4)+1 : end) = -6;
index((NX/4)+1:end, end) = -8;
index((NX/4)+1, NY/4:end) = -4;
index(1:NX/4, NY/4) = -8;
index(1, 1: NY/4) = -4;

index(3*NX/4,1) = -3;
index(3*NX/4,(3*NY/4)+1) = -3;

index((NX/4)+1,NY/4) = -7;
index((NX/4)+1,end) = -7;
```

```
index(1, 1:NY/4) = -10;
index(end, (3*NY/4)+1:end) = -11;


flip(index', 1)
flip(ij2n', 1)

%%
T = zeros(NX*NY, NX*NY);
[ii, jj] = find(index == 0);

for n = 1: length(ii)
    nn = ij2n(ii(n), jj(n));
    T(ij2n(ii(n)+1, jj(n)  ) , nn ) = 0.25;
    T(ij2n(ii(n)  , jj(n)+1) , nn ) = 0.25;
    T(ij2n(ii(n)-1, jj(n)  ) , nn ) = 0.25;
    T(ij2n(ii(n)  , jj(n)-1) , nn ) = 0.25;
end

valind = [ 1  1;
         0  1;
        -1  1;
         1  0;
         NaN NaN;
        -1  0;
         1 -1;
         0 -1;
        -1 -1];


% mi mivel
m = 1;
for i = 1:NX
    for j=1:NY
        if(-10 < index(i,j) & index(i,j) < 0 & index(i,j) ~= -5)
            mm(m, 1) = ij2n(i,j);
            mm(m, 2) = ij2n(i+valind(-index(i,j),1), j+valind(-index(i,j),2));
            m = m + 1;
        end
    end
end

f00 = find(reshape(index, 1, []) == -10);
f11 = find(reshape(index, 1, []) == -11);

%%
fi = zeros(IT, NX*NY);
% dirich
   for m= 1:length(f00)
       fi(1, f00(m)) = f0;
   end
   for m= 1:length(f11)
       fi(1, f11(m)) = f1;
   end

tic

for i = 2:IT
```

```
   fi(i, :) =  fi(i-1, :) * T;
   for m=1:length(mm)
       fi(i, mm(m, 1)) = fi(i, mm(m, 2));
   end
   % dirich
   for m= 1:length(f00)
       fi(i, f00(m)) = f0;
   end
   for m= 1:length(f11)
       fi(i, f11(m)) = f1;
   end
end

toc

fii = zeros(NX,NY, IT);
for i= 1:IT
    fii(:,:, i) = reshape(fi(i, :), NX, NY);
end



%%
fii( 1: NX/4, NY/4+1:end, end) = NaN;
fii((3*NX/4)+1:end, 1: 3*NY/4, end) = NaN;

figure;
ax1 = subplot(211);
contourf(fii(:,:,end)');
xlabel('x');
ylabel('y');
title('Potencial eloszlas az iteracio vegen');

ax2 = subplot(212);
contourf(diff(fii(:,:,end))');
xlabel('x');
ylabel('y');
title('Elektromos ter az iteracio vegen');

linkaxes([ax1 ax2], 'x');

%%


figm = figure('Renderer','zbuffer');
contourf(fii(:,:,2)'); title('0');
axis tight
set(gca,'NextPlot','replaceChildren');
% Preallocate the struct array for the struct returned by getframe
F(200) = struct('cdata',[],'colormap',[]);
% Record the movie
for j = 1:200
    fii( 1: NX/4, NY/4+1:end, j*100) = NaN;
    fii((3*NX/4)+1:end, 1: 3*NY/4, j*100) = NaN;
    surf(fii(:,:,j*100)');
    title(['it = ' num2str(j*100)]);
    xlabel('N'); ylabel('IT');
```

```
    %contourf(fii(:,:,j*100)');
    F(j) = getframe;
end


figure;
movie(F, 2)
```

main.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#include <helper_functions.h>
#include <helper_cuda.h>


extern "C"
void CPU(
    float *h_C,
    float *h_A,
    float *h_B,
    int vectorN,
    int elementN
);



#include "kernel.cuh"


float RandFloat(float low, float high)
{
    float t = (float)rand() / (float)RAND_MAX;
    return (1.0f - t) * low + t * high;
}



////////////////////////////////////////////////////////////////////////////////
// Data configuration
////////////////////////////////////////////////////////////////////////////////

const int VECTOR_N = 456;
const int ELEMENT_N = 4096;

const int    DATA_N = VECTOR_N * ELEMENT_N;

const int   DATA_SZ = DATA_N * sizeof(float);
const int RESULT_SZ = VECTOR_N  * sizeof(float);



////////////////////////////////////////////////////////////////////////////////
// Main program
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv)
```

```
{
    float *h_A, *h_B, *h_C_CPU, *h_C_GPU;
    float *d_A, *d_B, *d_C;
    //double delta, ref, sum_delta, sum_ref, L1norm;
    StopWatchInterface *hTimer = NULL;
    int i;


        getchar();

    printf("%s Starting...\n\n", argv[0]);

    // use command-line specified CUDA device, otherwise use device with highest Gflops/s
    findCudaDevice(argc, (const char **)argv);

    sdkCreateTimer(&hTimer);

    printf("Initializing data...\n");
    printf("...allocating CPU memory.\n");
    h_A     = (float *)malloc(DATA_SZ);
    h_B     = (float *)malloc(DATA_SZ);
    h_C_CPU = (float *)malloc(RESULT_SZ);
    h_C_GPU = (float *)malloc(RESULT_SZ);

    printf("...allocating GPU memory.\n");
    checkCudaErrors(cudaMalloc((void **)&d_A, DATA_SZ));
    checkCudaErrors(cudaMalloc((void **)&d_B, DATA_SZ));
    checkCudaErrors(cudaMalloc((void **)&d_C, RESULT_SZ));

    printf("...generating input data in CPU mem.\n");
    srand(123);

    //Generating input data on CPU
    for (i = 0; i < DATA_N; i++)
    {
        h_A[i] = RandFloat(0.0f, 1.0f);
        h_B[i] = RandFloat(0.0f, 1.0f);
    }

    printf("...copying input data to GPU mem.\n");
    //Copy options data to GPU memory for further processing
    checkCudaErrors(cudaMemcpy(d_A, h_A, DATA_SZ, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_B, h_B, DATA_SZ, cudaMemcpyHostToDevice));
    printf("Data init done.\n");


    printf("Executing GPU kernel...\n");
    checkCudaErrors(cudaDeviceSynchronize());
    sdkResetTimer(&hTimer);
    sdkStartTimer(&hTimer);
    GPU<<<128, 256>>>(d_C, d_A, d_B, VECTOR_N, ELEMENT_N);
    getLastCudaError("scalarProdGPU() execution failed\n");
    checkCudaErrors(cudaDeviceSynchronize());
    sdkStopTimer(&hTimer);
    printf("GPU time: %f msecs.\n", sdkGetTimerValue(&hTimer));

    printf("Reading back GPU result...\n");
    //Read back GPU results to compare them to CPU results
```

```
    checkCudaErrors(cudaMemcpy(h_C_GPU, d_C, RESULT_SZ, cudaMemcpyDeviceToHost));


    printf("Checking GPU results...\n");
    printf("..running CPU scalar product calculation\n");

        sdkResetTimer(&hTimer);
    sdkStartTimer(&hTimer);
    CPU(h_C_CPU, h_A, h_B, VECTOR_N, ELEMENT_N);
         sdkStopTimer(&hTimer);
    printf("cPU time: %f msecs.\n", sdkGetTimerValue(&hTimer));



    exit(EXIT_SUCCESS);
}
```

kernel.cuh

```
#define IMUL(a, b) __mul24(a, b)



#define ACCUM_N 1024
__global__ void GPU(
    float *d_C,
    float *d_A,
    float *d_B,
    int vectorN,
    int elementN
)
{
    //Accumulators cache
    __shared__ float accumResult[ACCUM_N];

    for (int vec = blockIdx.x; vec < vectorN; vec += gridDim.x)
    {
        int vectorBase = IMUL(elementN, vec);
        int vectorEnd  = vectorBase + elementN;

        for (int iAccum = threadIdx.x; iAccum < ACCUM_N; iAccum += blockDim.x)
        {
            float sum = 0;

            for (int pos = vectorBase + iAccum; pos < vectorEnd; pos += ACCUM_N)
                sum += d_A[pos] * d_B[pos];

            accumResult[iAccum] = sum;
        }

        for (int stride = ACCUM_N / 2; stride > 0; stride >>= 1)
        {
            __syncthreads();

            for (int iAccum = threadIdx.x; iAccum < stride; iAccum += blockDim.x)
                accumResult[iAccum] += accumResult[stride + iAccum];
        }

        if (threadIdx.x == 0) d_C[vec] = accumResult[0];
```

```
    }
}
```

Ábrák jegyzéke

3.1. OpenCL device architektúra [4] 8
3.2. OpenCL context osztálydiagrammja [4] 10
3.3. Feladat hozzárendelése work-item-hez (processzorhoz) 10

4.1. Új Eclipse projekt létrehozása 16
4.2. Compiler beállításai 17
4.3. Linker beállításai 17

Táblázatok jegyzéke

3.1. OpenCL memória szintek 9
3.2. Futási környezetek 11

5.1. Az órajel-generátor chip órajel-kimenetei. 18

Irodalomjegyzék

[1] P. Hartmann, A. Douglass, J. C. Reyes, L. S. Matthews, T. W. Hyde, A. Kovács, and Z. Donkó, „Crystallization dynamics of a single layer complex plasma," *Phys. Rev. Lett.*, vol. 105, p. 115004, Sep 2010.

[2] P. Hartmann, Z. Donkó, T. Ott, H. Kählert, and M. Bonitz, „Magnetoplasmons in rotating dusty plasmas," *Phys. Rev. Lett.*, vol. 111, p. 155002, Oct 2013.

[3] D. Z. Hartmann P, Donkó I, „Single exposure three-dimensional imaging of dusty plasma clusters," *Rev. Sci. Instrum.*, vol. 84, p. 023501, 2013.

[4] T. K. O. W. Group, „Opencl - the open standard for parallel programming of heterogeneous systems.." `http://www.khronos.org/opencl`, 2014.

[5] „Nvidia nsight visual studio edition." `https://developer.nvidia.com/nvidia-nsight-visu`

[6] „Gcc, the gnu compiler collection - gnu project - free software foundation (fsf)." `http://gcc.gnu.org/`.

[7] „Intel® many integrated core architecture (intel® mic architecture)." `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integ`

[8] „Centos project." http://www.centos.org/.

[9] „Nvidia drivers." `http://www.nvidia.com/Download/index.aspx`.

[10] „Intel opencl sdk." `https://software.intel.com/en-us/vcsource/tools/opencl-sdk-xe`.

[11] „Eclipse - the eclipse foundation open source community website.." `https://www.eclipse.org`.

[12] A. Munshi, *OpenCL Programming Guide*. Addison-Wesley, 2011.

[13] „Opencl programming guide - examples." `https://code.google.com/p/opencl-book-sampl`