



M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki 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	7
1.2. A mérendő mennyiségek és a származtatott értékek	7
2. A referenciaként szolgáló program	8
2.1. A program bemutatása	8
2.1.1. Detektálási algoritmus	8
2.1.2. Mennyiségek származtatása	8
2.2. Implementáció során hozott döntések	8
3. A multiprocesszoros OpenCL környezet	9
3.1. OpenCL architektúrája	9
3.2. OpenCL programozási modell	10
3.3. Futási környezet bemutatása	13
3.4. Implementációhoz szükséges megfontolások	13
3.5. Memória szervezés	13
3.5.1. Csak globális memória használata	13
3.5.2. Globális memória és adott esetben lokális memória használata	14
3.5.3. Globális memória és minden adódó alkalomkor a lokális memória használata	14
4. Multiprocesszoros program	15
4.1. Fejlesztőkörnyezet összeállítása	15
4.1.1. Software Development Kit-ek (SDK) telepítése	15
4.1.2. Eclipse – Integrated Development Environment	17
4.2. Új (Hello World) projekt létrehozása	17
4.2.1. Empty C project létrehozása	17
4.2.2. Compiler beállítása	18
4.2.3. Linker beállítása	19
4.3. Kód	19

5. Összehasonlítás	20
6. Összegzés	21
7. Függelék	22
7.1. Referencia program	22
Ábrák jegyzéke	23
Táblázatok jegyzéke	24
Irodalomjegyzék	26

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 üzemeltetésében, Terahertz technológiában egyre inkább előtérbe kerül.

A kísérleti adathgyűjtés 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 Intézetben található, kiegészítésével a mérés közbeni feldolgozással a mérést segítő analízist hajtson végre. Ennek eredményével a mérés előkészítése és elvégzése lényegesen gyorsulhatna.

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.

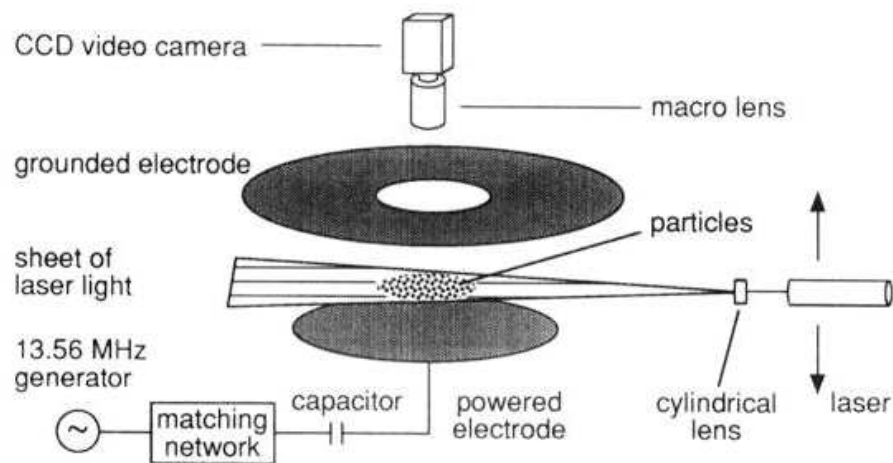
Kivonat

hablaty

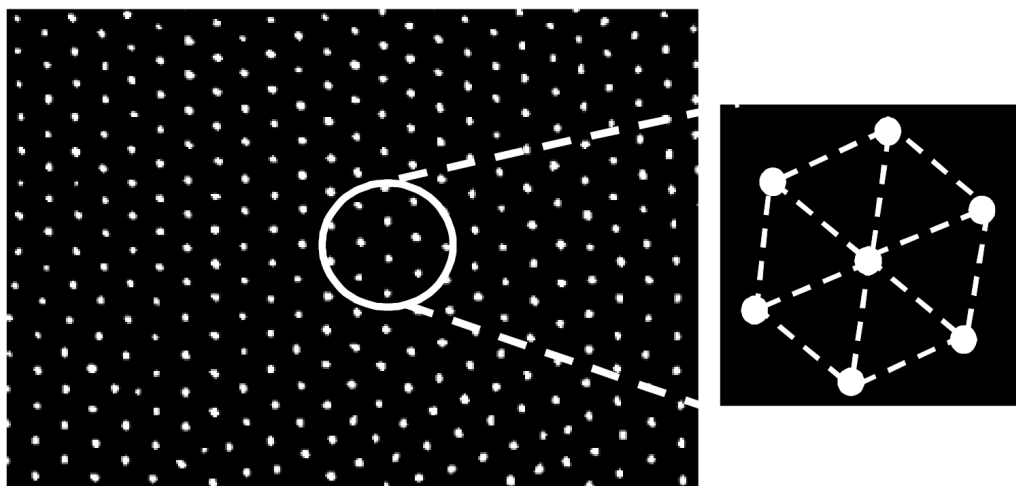
1. fejezet

A mérés

A poros plazma kísérleteket jól áttekintő cikk [4].



1.1. ábra. A mérési elrendezés sematikus ábrája (forrás: [4])



1.2. ábra. 2D-s Coulomb kristály hatszögletű struktúrájával (forrás: [4])

1.1. A mérési elrendezés

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

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

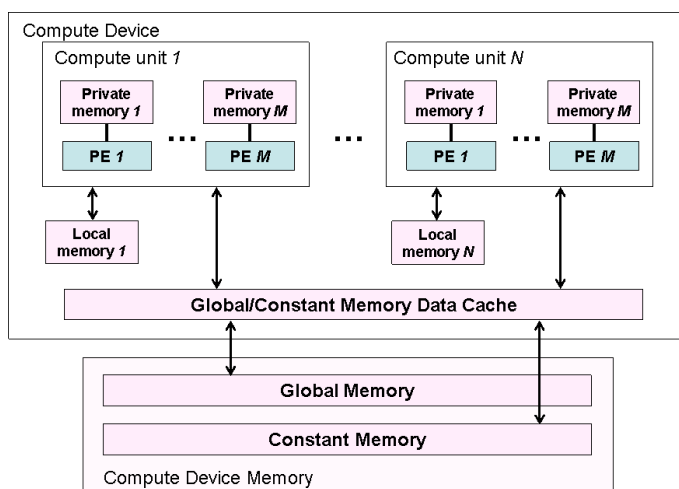
2.2. Implementáció során hozott döntések

3. fejezet

A multiprocesszoros OpenCL környezet

3.1. OpenCL architektúrája

Az Open Computing Language (OpenCL) keretrendszer [5] 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 hardvergyártók 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 írunk. Az egyik a kernel, ami az eszközön futatott szálra fog leképeződni. A másik a gazda processzoron (host-on) futó host-program, ami elvégzi az I/O műveleteket, a probléma összeállítását, a memória allokálást, az argumentumok beállítását illetve a kernel meghívását az eszközön. A kernel futása végeztével a host-program kiolvassa az eszközről a kívánt eredményt.



3.1. ábra. OpenCL device architektúra (forrás: [5])

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 allokalhat, írhat és olvashat belőle. A 3.1 táblázatban látható ezen jogosultságok.

3.1. táblázat. *OpenCL memória szintek*

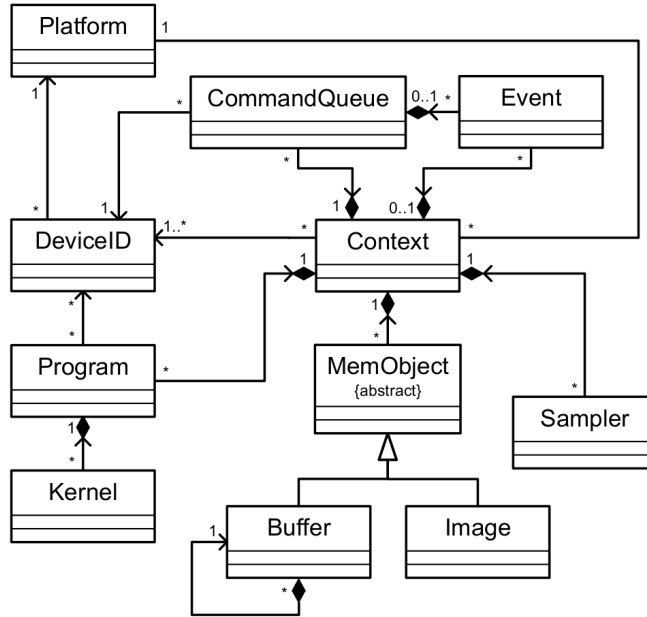
	Global memory	Constant mem.	Local mem.	Private mem.
Host	Dinamikusan R/W	Din. R/W	Din. R/W	
Kernel	R/W	Statikusan R	Satik. R/W	Statik. R/W
Sebesség	Lassú	Gyors	Gyors	Regiszter
Méret	1 Gbyte <	~ 64 Kbyte	~ 16 Kbyte	< 1 Kbyte

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 processzor-maghoz.
- *Mikor?*: Mivel a memória művelet alatt a futtatott kernel nem dolgozik, így átadja a helyét egy másiknak. (Ez Direct Memory Access (DMA) blokk létezése alatt igaz). Ennek a megfelelő szinkronizációjával nagyobb kihasználtság érhető el (load balance).

3.2. OpenCL programozási modell

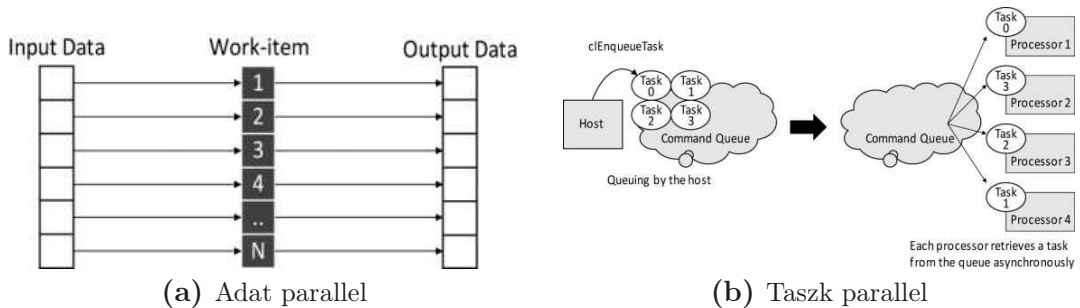
A programozási modell középpontjában a kontextus áll, ami az OpenCL osztálydiagramján 3.2 figyelhető meg. A futtatáshoz szükséges, hogy a kontextushoz platformot, majd azon belül eszközt, az eszközhöz programot (kernelt) és memóriát rendeljünk. Figyelembe kell vennünk azt a megkötést, hogy csak az egy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



3.2. ábra. OpenCL context osztálydiagrammja (forrás: [5])

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

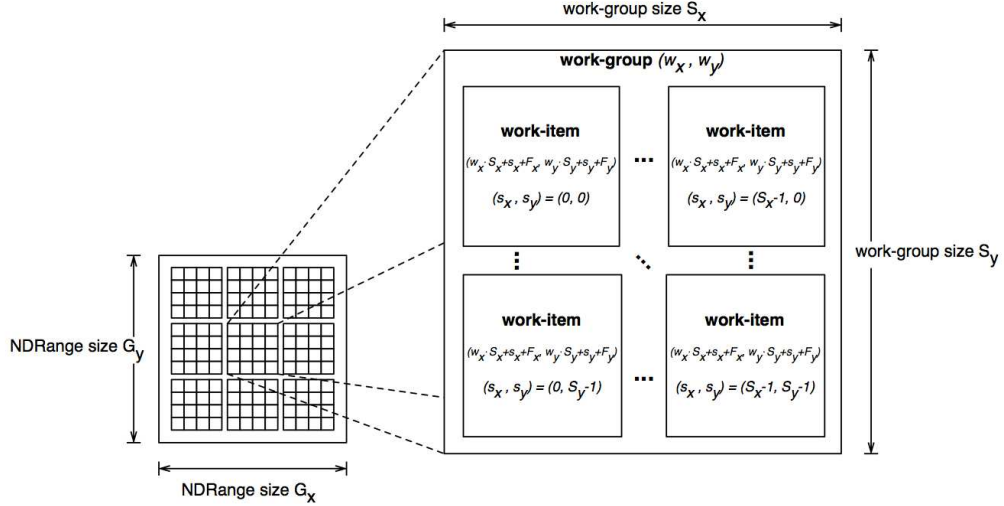
A programozással megoldandó problémát kétféleképpen lehetséges a feldolgozó egységekhez (work-item) avagy processzorokhoz rendelni: adat parallel módon vagy taszk parallel módon. Adat parallel módon (3.3a ábra) a feldolgozandó adat egy egységéhez rendelünk egy feldolgozó egységet. Fontos figyelembe venni 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 használni, ha a bemenet dinamikus mérete a futási időben rendkívül változik illetve a végrehajtandó feladat lezán függenek össze.



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.

A work-itemeket jelen pillanatban az OpenCL specifikációja szerint 3 dimenziós work-group-ba tudjuk rendezni. Egy példát láthatunk egy work-item indexének a globális és lokális megfelelőjére a következő 3.4 ábrán.



3.4. ábra. 2D-s work-item-ek work-group-ba rendezése és indexelése
(forrás: [5])

A work-group-okba rendezés a lokális memória jogosultsága 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 „cachelni” a lokális memóriába.

Mivel a work-item-ek konkurrensen hajtódnak végre, így az általuk közösen elérhető memóriákra (globális, lokális) nézve versenyhelyzetben vannak. Az OpenCL ezt a problémát a laza memóriamodell használatával oldja meg. Az alkalmazott szinkronizáció egy korlátot tesz a programban, amit csak akkor léphet át, ha az összes többi work-item az azonos work-group-ban ezta a korlátot már elérte. Erre a **barrier(FLAG)** függvényhívás szolgál. Fontos megjegyezni, hogy ez a szinkronizáció csak egy adott work-group-on belül történik, a work-group-ok közötti szinkronizációra nincs lehetőség.

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

3.3. Futási környezet bemutatása

A következő eszközök teljesítményét vizsgálom:

- A laptopomban található **Intel Core i5 M520** processzor,
- A laptopomban található kis teljesítményű **nVidia GT330M** videokártya,
- Asztali PC-ben található **Intel Xeon CPU**,
- Asztali PC-ben található **Intel Xeon Phi** co-processzor [6, ?].

Ezen eszközök legjelentősebb paraméterei a 3.2 táblázat tartalmazza.

3.2. táblázat. *Használandó eszközök összehasonlítása*

	Intel Core i5	nVidia GT330M	Intel Xeon	Xeon PHI
MAX COMPUTE UNITS	4	6	8	224
MAX CLOCK FREQUENCY	2400	1265	3000	1100
MAX WORK GROUP_SIZE	8192	512	8192	8192
GLOBAL MEM SIZE	$\sim 4Gbyte$	$1Gbyte$	$8Gbyte$	$\sim 4.5Gbyte$
LOCAL MEM SIZE	$32Kbyte$	$16Kbyte$	$32Kbyte$	$32Kbyte$

Az összehasonlíthatóság végett a legkisebb memóriájú eszközre fogom a problémát skálázni, ami a GT330M videokártya. A többi eszköz memóriája jóval nagyobb, így a kód könnyen portolható rájuk.

3.4. Implementációhoz szükséges megfontolások

Ezen feladatrészek méretét egy paraméter állításával lehet változtatni 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 allokalunk 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, mielőtt 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 memórral 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 programozása, de eredményképp gyorsulás elérhető.

Összegezve elmondható, hogy az aktuálisan használt adat tárolását a lehető legközelebb kell tartani a processzor-maghoz.

4. fejezet

Multiprocesszoros program

4.1. Fejlesztőkörnyezet összeállítása

OpenCL kód fejlesztése történhet Windows alatt NVIDIA Nsight Visual Studio Edition [7] és Linux alatt GCC-vel [8]. 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 [9] kártyái. Ezekre való OpenCL kód fejlesztéséhez a gyártók biztosítanak Software Development Kit-et (SDK). Ezek telepítése szinte bármelyik Linux disztribúción sikerülhet a megfelelő követelmények előzetes telepítése után. A Linux disztrók közül a CentOS-re [10] esett a választás, ami csupán a fejlesztőkörnyezet egyszerűbb telepítése végett történt így.

4.1.1. Software Development 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 [11]. A grafikus felületet a telepítés idejére le kell állítani az X grafikus kiszolgálót

```
$ init 3
```

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

```
$ init 5
```

parancssal áttérhetünk a grafikus felületre, ahol a megfelelő környezeti változókat kiegészíthetjük. Legcélratosabb, 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/oneapi/bin
export LD_LIBRARY_PATH

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

INTELOCLSDKROOT=/opt/intel/oneapi
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 [12]. 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

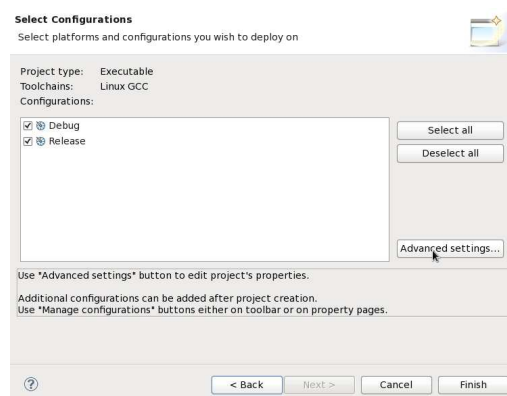
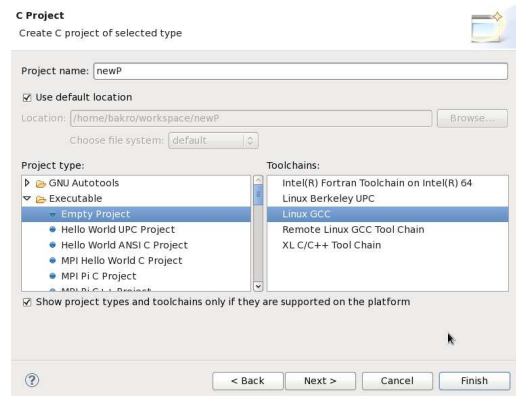
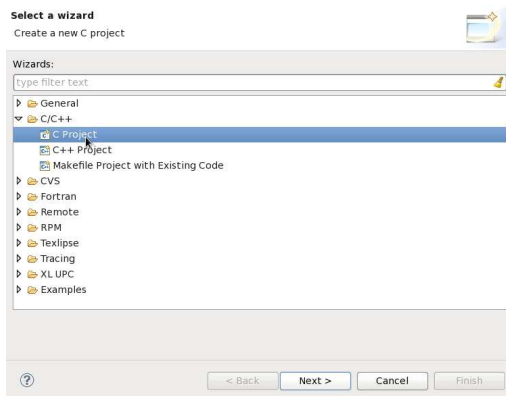
A fejlesztés és hibakeresés egy Integrated Developement Enviroment (IDE) segítségével könnyebb. Az open source Eclipse [13] fejlesztőkörnyezet a különböző pluginjaival épp megfelelő erre a célra. Például a C-nyelv fejlesztését segítő C/C++ Developement Tooling (CDT), a verziókövetést menedzselő EGit 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 [14] könyvben szereplő Hello World programot a következő linken lehet letölteni [15]. A kód fordítása előtt 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 kattintva a tulajdonságára kattintva állíthatjuk be a fordítót a képek 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.

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. Referencia program

refcode.c

7.1. lista. *A referencia programkódja*

```
main() { return; }
```

Ábrák jegyzéke

1.1. A mérési elrendezés sematikus ábrája (forrás: [4])	6
1.2. 2D-s Coulomb kristály hatszögletű struktúrájával (forrás: [4])	6
3.1. OpenCL device architektúra (forrás: [5])	9
3.2. OpenCL context osztálydiagrammja (forrás: [5])	11
3.3. Feladat hozzárendelése work-item-hez (processzorhoz)	11
3.4. 2D-s work-item-ek work-group-ba rendezése és indexelése (forrás: [5])	12
4.1. Új Eclipse projekt létrehozása	18
4.2. Compiler beállításai	19
4.3. Linker beállításai	19

Táblázatok jegyzéke

3.1. OpenCL memória szintek	10
3.2. Használandó eszközök összehasonlítása	13
5.1. Az órajel-generátor chip órajel-kimenetei.	20

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] R. L. Merlino, „Dusty plasmas and applications in space and industry,” *Plasma Physics Applied*, vol. 81, pp. 73–110, 2006.
- [5] T. K. O. W. Group, „Opencl - the open standard for parallel programming of heterogeneous systems..” <http://www.khronos.org/opencl>, 2014.
- [6] Intel, „Intel® xeon phi™ product family.” <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html>.
- [7] „Nvidia nsight visual studio edition.” <https://developer.nvidia.com/nvidia-nsight-visual-studio-edition>.
- [8] „Gcc, the gnu compiler collection - gnu project - free software foundation (fsf).” <http://gcc.gnu.org/>.
- [9] „Intel® many integrated core architecture (intel® mic architecture).” <http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core-architecture.html>.
- [10] „Centos project.” <http://www.centos.org/>.
- [11] „Nvidia drivers.” <http://www.nvidia.com/Download/index.aspx>.
- [12] „Intel opencl sdk.” <https://software.intel.com/en-us/vcsource/tools/opencl-sdk-xeon>.
- [13] „Eclipse - the eclipse foundation open source community website..” <https://www.eclipse.org>.
- [14] A. Munshi, *OpenCL Programming Guide*. Addison-Wesley, 2011.

[15] „Opencl programming guide - examples.” <https://code.google.com/p/opencl-book-samples/>