



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

DIPLOMA

*Készítette*

Bakró Nagy István

*Konzulens*

Hartmann Péter

Reichardt András

2014. december 17.

# Tartalomjegyzék

<b>Feladatkiírás</b>	<b>v</b>
<b>Kivonat</b>	<b>vii</b>
<b>Abstract</b>	<b>ix</b>
<b>1. A poros plazma kísérlet</b>	<b>1</b>
1.1. A kísérlet . . . . .	3
1.2. Használt kamera tulajdonságai . . . . .	4
1.3. A mérendő mennyiségek és a származtatott értékek . . . . .	4
<b>2. A részecskék detektálása</b>	<b>7</b>
2.1. Detektálási módszerek . . . . .	7
2.1.1. Küszöb módszer . . . . .	7
2.1.2. Küszöb módszer szűréssel . . . . .	7
2.1.3. Adaptív küszöb módszer szűréssel . . . . .	8
2.2. A részecskék pozíciójának számítása . . . . .	10
<b>3. A multiprocesszoros OpenCL környezet</b>	<b>13</b>
3.1. OpenCL architektúrája . . . . .	13
3.2. OpenCL programozási modell . . . . .	14
3.3. Futási környezet bemutatása . . . . .	17
<b>4. A host program bemutatása</b>	<b>19</b>
4.1. A host program párhuzamos felépítése . . . . .	20
4.2. Main (producer) szál . . . . .	21
4.3. CProducer szál . . . . .	24
4.3.1. OpenCL inicializálás . . . . .	24
4.3.2. Kép kernelekkel történő feldolgozása . . . . .	25
4.4. Consumer szál . . . . .	25
<b>5. A kernel programok lépéseinek bemutatása</b>	<b>27</b>

5.1. Medián szűrés . . . . .	27
5.2. Átlagolás . . . . .	28
5.3. Detektálás . . . . .	29
<b>6. Haználati útmutató és vizsgálat</b>	<b>31</b>
<b>7. Összehasonlítás</b>	<b>33</b>
<b>8. Összegzés</b>	<b>35</b>
<b>A. Fejlesztőkörnyezet összeállítása</b>	<b>I</b>
<b>Ábrák jegyzéke</b>	<b>VII</b>
<b>Táblázatok jegyzéke</b>	<b>IX</b>
<b>Irodalomjegyzék</b>	<b>XII</b>

## HALLGATÓI NYILATKOZAT

Alulírott *Bakró Nagy István*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2014. december 17.

---

*Bakró Nagy István*  
hallgató



# 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 adatgyűjtés 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 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, SZFKI)

Budapest, 2014.03.10.

# Kivonat

hablaty





# Abstract

what? what? in the butt

# 1. fejezet

## A poros plazma kísérlet

A poros plazma kísérletek során elektromos gázkisülési plazmába szilárd szemcséket (port) szórunk, amelyek elektromos töltésre tesznek szert, és az így előállított erősen kölcsönható sokrészecske rendszert figyeljük meg. Adott alacsony nyomású gáz térben elhelyezett elektródákra kapcsolt feszültséggel lehetséges plazmát létrehozni. Az elektromos táplálás lehet egyenáramú, vagy váltakozó áramú a rádiófrekvenciás vagy akár a mikrohullámú tartományban. A rádiófrekvenciával váltakozó villamos tér a töltéssel rendelkező szabad elektronokra olyan erővel hat, hogy azok felgyorsulva, a háttérgáz semleges atomjaival ütközve azokat gerjeszteni és ionizálni tudják, így hozzájárulva a szabad töltéshordozók szaporításához. A leszakadó elektronok a térben szabadon a ráható erőknek megfelelően mozognak és további ütközésekben vehetnek részt egészen addig, míg valamely elektródán elnyelődnek. Ez makroszkópikus skálán az eredetileg szigetelő gáz vezető plazmává válását jelenti.

A kísérlet során az előbb említett elektromos gázkisülésbe plazma terébe szórt porszemcsék alatt  $100\text{nm} - 20\mu\text{m}$  nagyságú részecskéket értünk, melyek anyaga lehet például  $\text{SiO}_2$ ,  $\text{Al}_2\text{O}_3$  vagy melamin-formaldehid (MF). A porrészecskék a plazmával interakcióba lépve negatívan feltöltődnek az azokat érő ion-, és elektronáramok eredményeként. A szemcsék töltés per tömege aránya sok nagyságrenddel kisebb a háttérplazma atomos összetevőinél, aminek hatására a mozgásuk karakterisztikus ideje lényegesen hosszabb az ionokénál is. A szemcsék mozgását a környezetük időben kiátlagolt hatásai dominálják, vagyis dinamikájuk jórészt lecsatolódik az atomos háttérplazma gyors változásairól. Ennek következtében, sok esetben a töltött porszemcsékből álló rendszert önmagában tekinthetjük egy egykomponensű rendszernek, annak tudatában, hogy a szemcsék töltését és az összetartáshoz szükséges peremfeltételeket a gázkisülés biztosítja.

A porrészecskék transzportjának megértéséhez szükséges a ráható erők azonosítása. A különféle erők nagysága a porrészecskék nagyságával különféleképpen skálázódik. Elhanyagolásokat ennek megfelelően tehetünk.

**Gravitációs  $F_g$  erő:** Mikrogravitációban végzett kísérletek és nanométer nagyságú részecskék esetén elhanyagolhatóak, de a jelen esetben használt mikrométer nagyságú részecskék esetén dominánsak,

**Villamos tér keltette  $F_e$  erő:** A porrészecske töltésével és a villamos tér nagyságával arányos. A megfelelően irányított villamos térrel lehetséges a részecskék levitációja,

**Háttératomon való szóródás  $F_n$ :** A porrészecske driftje során a háttératomokkal való **ütközéseinek** makroszkopikus erőként való számításba vétele,

**Hőmérséklet gradiensi  $F_{th}$  erő:** A gáz hőmérsékletének gradiense okozta gázatomok diffúzív jellegű mozgása által okozott indirekt erőhatás,

**Ion sodrási  $F_i$  erő:** Az ionokra ható villamos tér okozta **ionáramok** hatása a porrészecskékre.

A poros plazma analízise során fontos szerepet játszik a porrészecskék csatolása. A csatolást gyengének és erősnek kategorizáljuk aszerint, hogy a részecskék szomszédja általi átlagos potenciális energia az átlagos termikus (kinetikus) energiájához képest kisebb vagy nagyobb. A csatolást a Coulomb csatolási paraméterrel ( $\Gamma$ ) lehet számosítani, ami a szomszédos részecskék Coulomb potenciáljának és termikus energiájának **hányadosa**:

$$\Gamma = \frac{1}{4\pi\epsilon_0} \frac{Q^2}{ak_B T_d} \quad (1.1)$$

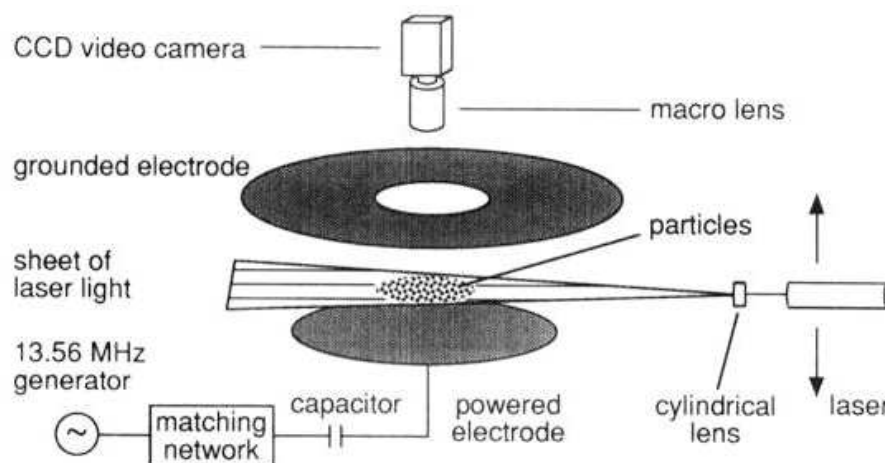
ahol  $Q$  a részecskék töltése,  $a$  az átlagos részecske távolságával **arányos Wigner-Seitz sugár** (amely a szemcsék sűrűségéből számolható  $a_{2D}^2 = 1/\pi n$ , illetve  $a_{3D}^3 = 3/4\pi n$  formulákkal, a rendszer dimenzionalitásától függően) és a  $k_B T_d$  a por **komponens egy részecskére jutó átlagos kinetikus energiája**. Ha a  $\Gamma$  értéke egy kritikus érték fölé **kb.**  $\Gamma > 150$  növekszik, akkor Ikawa jóslata szerint a porrészecskék **sokasága** kristályszerkezetbe rendeződik [?].

Poros plazmák tudatos alkalmazása jelenleg leginkább tudományos természetű, bár létezésükre szennyeződésként az iparban figyeltek fel először. Ez pl. VLSI áramkörök gyártásakor (plazma alapú maratás során) jelentkezik, ahol a kezelt felületre visszahulló szemcsék rövidzárlatot és egyéb hibákat okozhatnak, ami a kihozatal csökkenéséhez vezet. Laboratóriumi környezetben hasznos modellrendszerei lehetnek az atomos (hagyományos) anyagokban lejátszódó makroszkopikus folyamatok mikroszkopikus részleteinek feltárásához [?].

## 1.1. A kísérlet

A kísérlet lebonyolítására egy speciális kamrára van szükség, amiben lehetséges a plazma létrehozása, a porrészecskék szórása illetve a megfelelő villamos tér létrehozása. A kamrának hermetikusan jól zártnak kell lennie, hogy csak a kívánt gázt tartalmazza. A középvákuumú működéshez kétlépéses vákuumszivattyút kell alkalmazni.

A kamra sematikus ábrája a 1.1. ábrán látható. A porrészecskék levitációjáért a villamos tér felelős. A síkba való zárás parabolikus potenciállal lehetséges. Az ilyen tér létrehozása a következő elektróda elrendezéssel lehetséges: alul elhelyezett korong alakú elektróda, ami felett egy gyűrű alakú elektróda helyezkedik el. Az ilyen elektródarendszerre kapcsolt váltakozó feszültség a beszórt porrészecskéket lebegtetni tudja. Továbbá a részecskék követéséhez azokat lézerrel megvilágítjuk és nagysebességű kamerával felvételt készítünk róla.



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

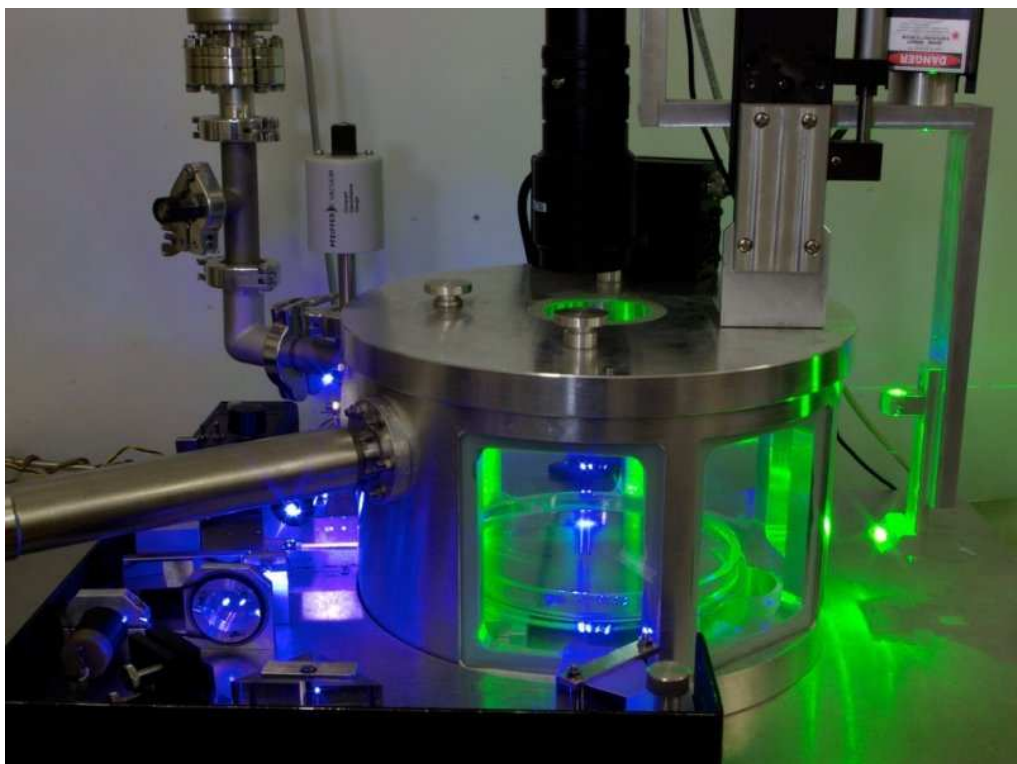
A kamrát (1.2. ábra) Hartmann Péter külső konzulensem építette és a MTA Wigner FK SZFI Gázkisülési Laboratóriumában található.

A paramétereit:

- Kamra belső átmérője: 25 cm
- Kamra belső magassága: 18 cm
- Alsó elektróda átmérője: 18 cm
- Felső gyűrű elektróda belső átmérője: 15 cm
- Felső elektróda távolsága az alsótól: 13 cm
- Argon gáz nyomása:  $1.2 \pm 0.05$  Pa
- Gáz átfolyása:  $\sim 0.01$  sccm
- RF gerjesztés: 7W @ 13.56 MHz

- Porrészecske: melamin-formaldehid
- Porrészecske átmérője:  $4.38 \pm 0.06 \mu\text{m}$
- Porrészecske tömege:  $6.64 \cdot 10^{-14} \text{ kg}$
- Látható porrészecskék száma:  $\sim 2500$
- Megvilágító lézer:  $200 \text{ mW @ } 532 \text{ nm}$
- Kamera: 1.4 MPixel @ 100 FPS

**Opcionálisan** kamra alsó elektródáját lehetséges egy motorral forgásba hozni. Az elektróda felületének érdessége által a kamrában lévő gáz is forgásba jön. A gáz forgása az  $F_n$  sodrási erővel hat a porrészecskékre. **A forgó rendszerrel együttforgó viszonyítási rendszerben fellépő Coriolis tehetetlenségi erő extrém nagy mágneses tér Lorentz erejével ekvivalens hatást gyakorol a porszemcsékre, megkerülve ezzel a valódi mágnesek alkalmazásával együttjáró nehézségeket.**

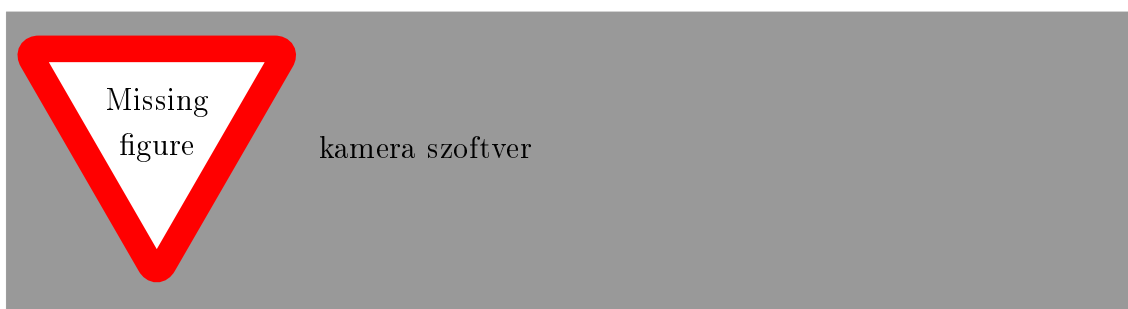


1.2. ábra. A konkrét kamra működése közben

## 1.2. Használt kamera tulajdonságai

## 1.3. A mérendő mennyiségek és a származtatott értékek

A kísérlet előkészítése a következő lépésekből áll:



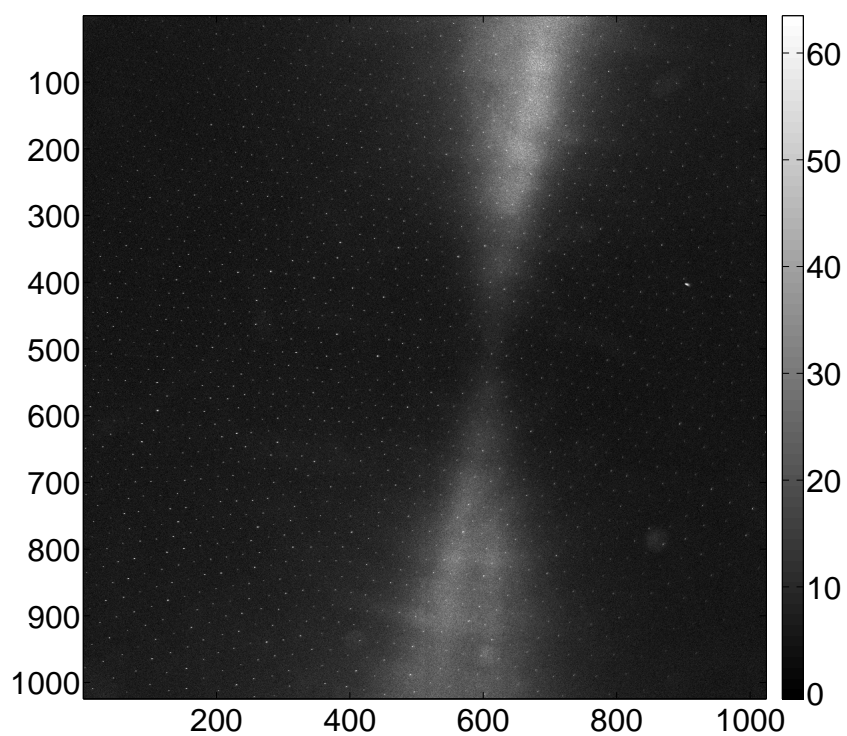
**1.3. ábra.** Kamera vezérlő szoftvere (forrás citexxx)

1. Elővákuum (rotációs) szivattyú bekapcsolása,
2. Középvákuum (turbómolekuláris) szivattyú bekapcsolása,
3. Argon palack megnyitása a megfelelő áramlás szintjére,
4. RF gerjesztés bekapcsolása,
5. Megvilágító lézer bekapcsolása,
6. Porrészecskék beszórása,
7. Kamera bekapcsolása és a megjelenítő szoftver futtatása,
8. Ha sok összetapadt porrészecske látható, illetve ha túl sok porrészecske látható, akkor az RF gerjesztés gyors ki-be kapcsolása után a 6.-tól való folytatása a folyamatnak.

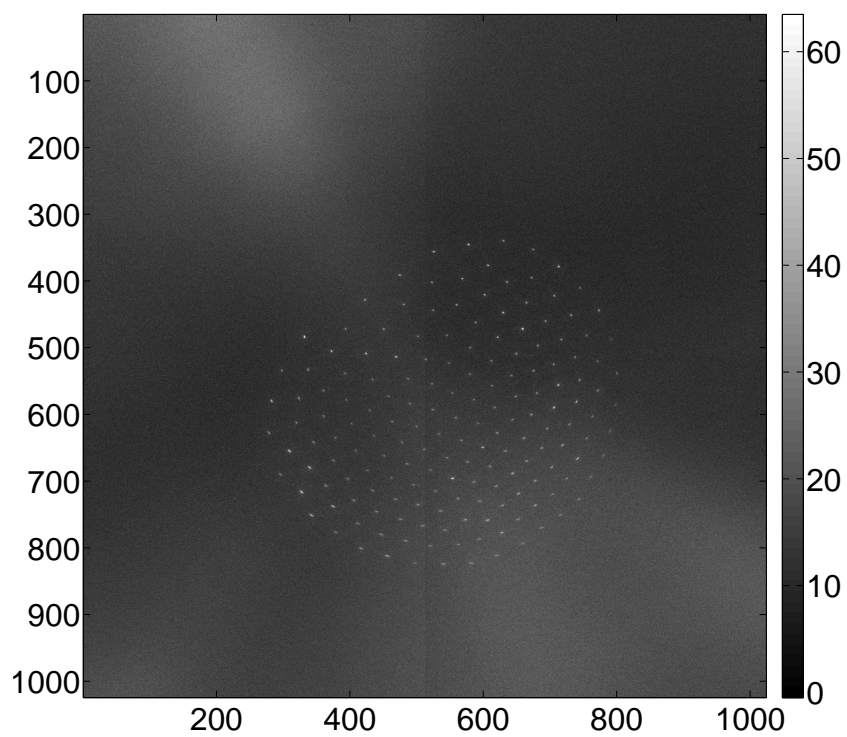
A 1.4. ábrán látható álló és forgó alsó elektródájú kísérlet során a porrészecskékről készült kép.

A fényképek alapján a részecskék pozíciójára és időbeli mozgására vagyunk kíváncsiak. A kamera objektíve által leképezett kép a függőleges irányú pozíciót nem tartalmazza. Mivel jelen kísérletek során az ezirányú mozgása a részecskéknek nem számottevő, így az  $x - y$  pozícióját jól lehet számítani a kép alapján. Nagy pontosság esetén szükséges a képek előzetes feldolgozása a perspektivikus torzítás kiküszöbölése végett. Természetesen ez lehetséges a pozíciók számítása után is, aminek a számításigénye kisebb is.

mi kerül ki-  
mentésre



(a) Álló elektróda esetén sok ( $\sim 1500$ ) részecske



(b) Forgó elektróda esetén kevés ( $\sim 100$ ) részecske

**1.4. ábra.** Két különböző kísérlet során készült fénykép.



## 2. fejezet

# A részecskék detektálása

### 2.1. Detektálási módszerek

A korábban látható 1.4. ábrán látható ábrákhoz hasonló képekről kell a részecskéket felismerni és azoknak a koordinátáit exportálni. Erre több lehetőség adódik, amit a [5] részletez. A detektáló módszereket a számítás igényeiknek növekvő sorrendjében mutatom be.

#### 2.1.1. Küszöb módszer

Legkézenfekvőbb módszer, hogy a kép pixeleinek világosságát összehasonlítjuk egy küszöb értékkel és ha az nagyobb ennél, akkor ezeket megjelöljük, mivel ott részecskét feltételezünk. A módszer egyenletes háttér-világosság esetén jól működik és extra gyorsan számítható.

#### 2.1.2. Küszöb módszer szűréssel

A háttér világossága a 1.4a. ábrán jól láthatóan nem egyenletes. A korábban említett egyszerű küszöb módszer itt nem alkalmazható, mivel a világos és sötét területek más és más küszöbértéket kívánnának meg. A megoldása erre, mint megannyi villamosmérnöki mérési feladatra, hogy a jel helyett a differenciális jelet mérjük/számítjuk. Jelen esetben ez azt jelenti, hogy először előállítjuk a részecske nélküli háttérképet, majd a méréssel kapott képből kivonva ezt a differenciális képet megkapjuk. A differenciális képen a részecskéket a korábban említett küszöb módszerrel lehet detektálni a részecskéket.

Ehhez csupán a mérési képből kell szűréssel származtatni a háttérképet azaz eliminálni a részecskéket. A [5, 6] cikkekben és általában is erre véges Gauss szűrőt használnak, ami egy lineáris véges impulzusválaszú (FIR) aluláteresztő szűrő. A szűrés egy adott pixel környezetének súlyozott átlagolását jelenti. A súlyozás során

szoroznunk kell, ami a bináris megvalósítás végett jóval lassabban történik, mint az összeadás avagy az összehasonlítás.

Mivel a részecskék mérete a képen véges és kis szórású, így a korábban említett FIR Gauss szűrő aránylag jól tudja a részecskéket eliminálni. Viszont a mérési képek 100 FPS sebességgel és  $1024 \times 1024$  mérettel érkeznek be. Ez 100 MByte/s adatfolyamot jelent. Ahhoz, hogy ezt közel real-time feltudjuk dolgozni a Gauss szűrés nem járható út. Hatékonyabb szűrőre van szükségünk! A megoldást a medián szűrőben látom, ami egy nemlineáris, de véges „impulzusválaszú” szűrő. A szűrő az adott pixel környezetének mediánjának számítását jelenti. A nemlinearitás jelen esetben nem okoz gondot, mivel csak detektálásra használjuk (a pozíció kinyerése az eredeti kép alapján készül, de ez később részletezve lesz.)

A Gauss szűrő  $N = n \times n$  környezet (ablak) esetén  $N^2$  szorzást és  $N^2 - 1$  összeadást jelent. Míg a medián szűrő  $N = n \times n$  ablak esetén: bubborék rendezés során  $O(N^2)$ , javított bubborék rendezés során  $O(N^2/2)$ , quicksort esetén  $O(N \log N)$  illetve kiválasztásos rendezés esetén  $O(N)$  összehasonlítást és cserét. Az összehasonlítás és a csere nagyságrendekkel gyorsabban végrehajtható, mint a szorzás és összeadás. A kedvező lépésszám és helybeli rendezés lehetősége végett a kiválasztásos rendezést választottam.

### 2.1.3. Adaptív küszöb módszer szűréssel

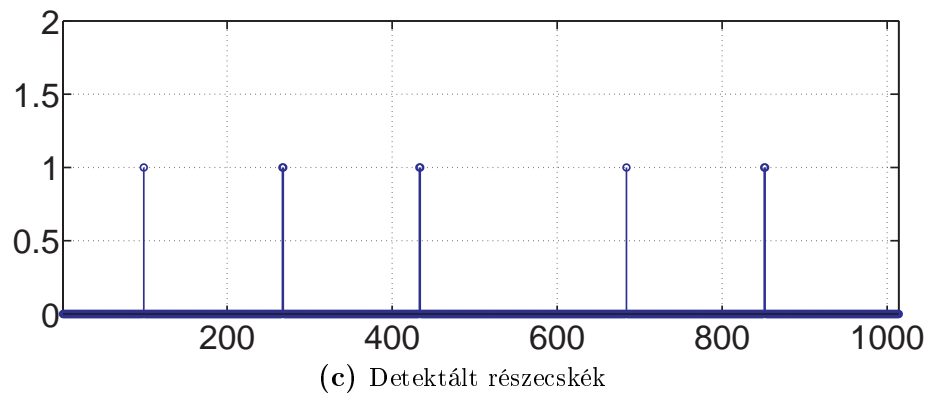
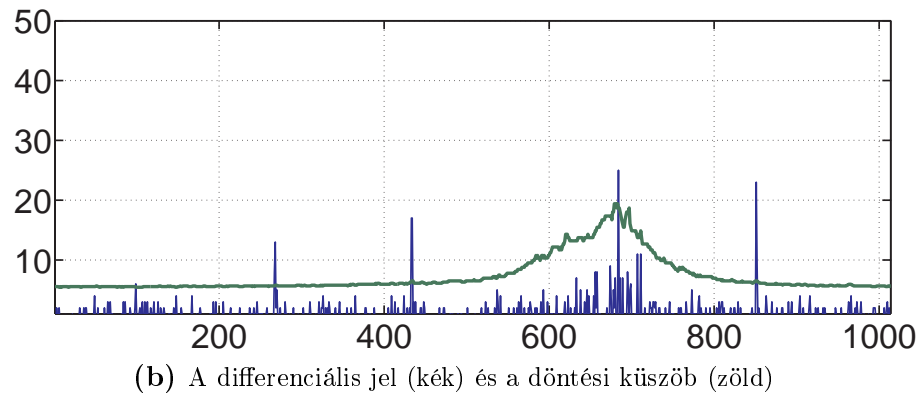
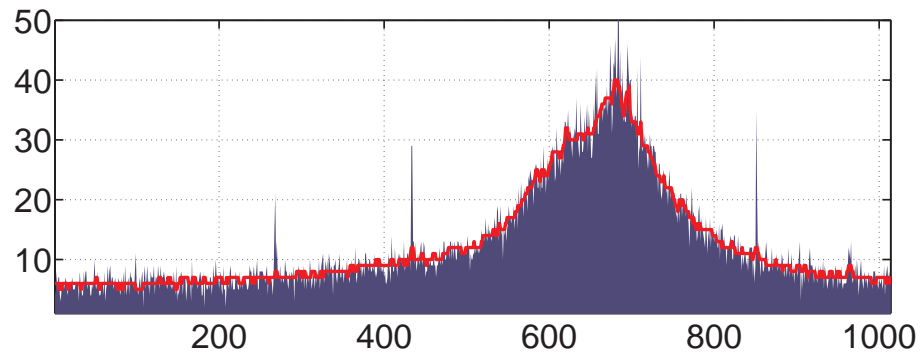
Látható a 1.4a. ábrán, hogy a sötétebb háttér kiterjedése nagyobb, ennek megfelelően az kamera expozíció szabályozója ezekre a területekre állítja be az expozíciót. Így a sötét területek részletesebbek, míg a világos területek részletszegények (túl-exponáltak) lesznek. Számunkra ez azt jelenti, hogy a sötétebb területeken jobban megbízhatunk a részecskék kiugró értékében, míg a világosabb területeknél nem. Ezt a döntési küszöb értékének adaptációját jelenti a háttér világosságához. Az adaptív küszöböt a következő kifejezéssel határoztam meg:

$$K = \mathbf{E} \{ P - \hat{M}P \} + \delta \cdot \mathbf{STD} \{ P - \hat{M}P \} \cdot \left[ 1 + a \left( \frac{\hat{M}P}{\max \{ \hat{M}P \} - \min \{ \hat{M}P \}} \right)^b \right] \quad (2.1)$$

ahol  $P$  az eredeti kép,  $\hat{M}P$  a medián szűrt kép,  $\mathbf{E} \{ \}$ ,  $\mathbf{STD} \{ \}$  az átlag és a szórás számításának függvénye és az  $a, b, \delta$  három alakparaméter.<sup>1</sup>

<sup>1</sup>Az adaptív döntést a mérési kép előzetes feldolgozásával is elérhettük volna, ha a Photoshop-ból ismert Curves tool-hoz hasonlóval módosítottunk volna rajta. (A tool a képre egy nemlineáris függvénnyel hattat.)

A 2.1. ábrán látható a detektálási algoritmus működése közben. A 2.1a. ábrán a mérési kép ( $P$ ) látható kitöltött görbével, ami az 1.4a. ábrán látható mérési kép  $x = 40$  sora. Piros görbével látható a medián szűrt jelet ( $\hat{MP}$ ), amin jól érzékelhető a hirtelen változások eliminációja. A következő 2.1b. ábrán az eredeti és a szűrt különbsége azaz a differenciális kép ( $P - \hat{MP}$ ) látható a kék görbével. A zöld görbe a (2.1) szerinti döntési küszöb. Az utolsó 2.2b. ábrán a detektált részecskék figyelhetőek meg.



**2.1. ábra.** A medián szűrést alkalmazó adaptív küszöbvel részecskét detektáló algoritmus bemutatása az álló mérési kép (1.4a. ábra)  $x = 40$  során.

## 2.2. A részecskék pozíciójának számítása

Kis felbontású kamera illetve nagyon kis porrészecskék esetén előállhat, hogy a részecskék csupán egy pixelnyi területet foglalnak el a képen. Detektálás szempontjából ez kedvező viszont a pozíciómérés szempontjából nem, mivel ilyenkor a felbontásunk 1 pixelnyi. Ezen javítani a dithereléssel a következőképpen lehet: picit elállítjuk az élességet úgy, hogy egy részecske több pixel nagyságú „maszat” legyen, majd a korábban részletezett detektálást elvégezzük.

Ennek hatására egy részecske több pixelnyi felületet fog elfoglalni és a detektáló algoritmus is több pixelt fog megjelölni. A pozíció megtalálásához csoportosítani kell a megjelölt pixeleket. Az egy részecskéhez tartozó pixel-csoportot egy téglalap fogja határolni, amit region of interest-el (ROI) szokás illetni. Azonban a részecske detektálása után amorf formájú megjelölt pixeleink lesznek. Ezeket a könnyebb csoportosítás végett kiterjesztem pár pixellel az így kapott pixeleket flood-fill algoritmussal egybefüggővé teszem és ennek eredményeképp megkapom a ROI határoló koordinátáit, amit a következő két módszer felhasznál a részecske pozíciójának számítása során.

### Maximum keresés

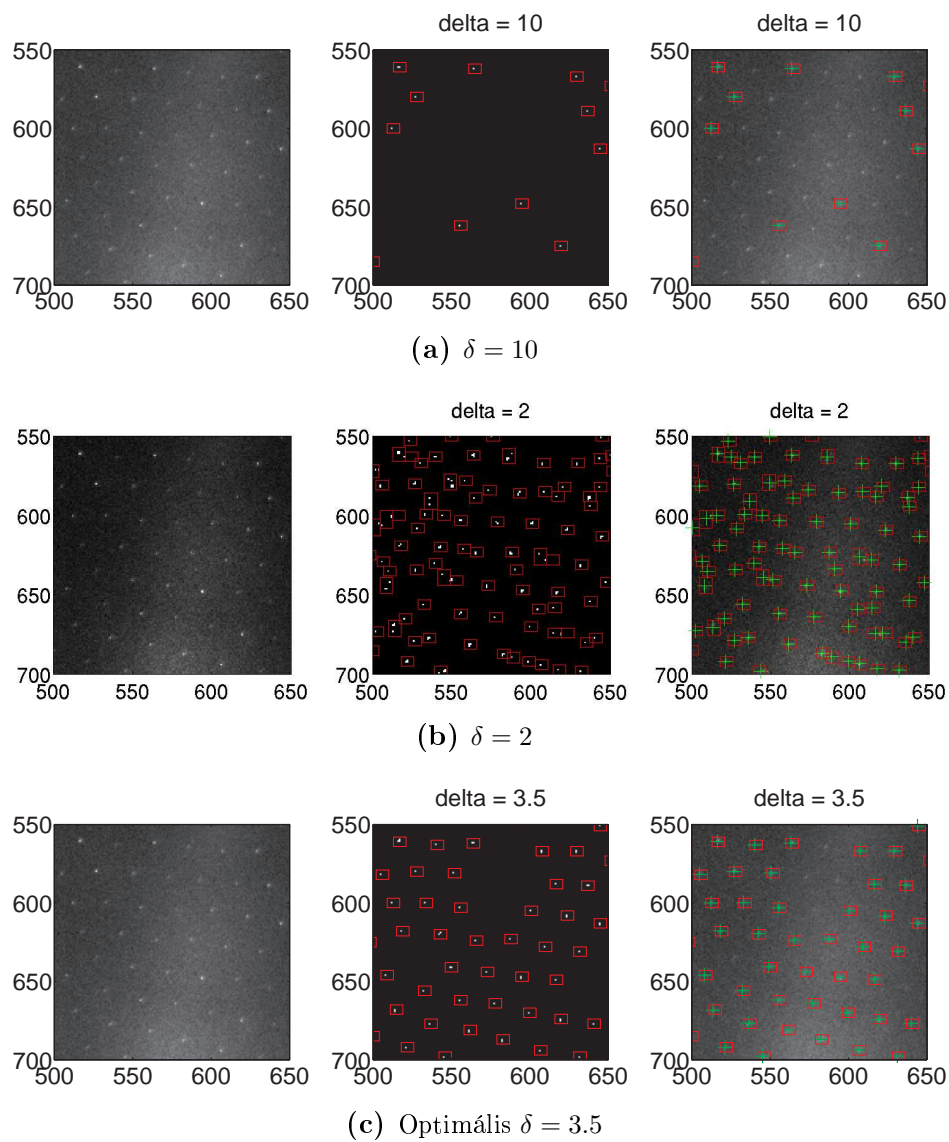
Legegyszerűbb eset, ha a ROI-n belül az eredeti pixelek közül a legvilágosabbat veszem a részecske pozíciójaként.

### Szubpixel felbontás momentum módszerrel

Szofisztikáltabb, ha a ROI-n belül az eredeti pixelek világosságát, mint tömegpont tömegének veszem és a ROI-által határolt test súlypontját megkeresem. A módszerre kritikusan hat az előbb említett kiterjesztés mértéke. Ha túl nagy a kiterjesztés, akkor azzal hibát viszek be pozíció mérésébe, míg ha túl kicsi akkor meg egy részecskének akár két képe/pozíciója keletkezhet. Továbbá nem érünk el nagyobb pontosságot a maximumkereséshez képest.

Az algoritmus hatékonyságát különböző  $\delta$  érték mellett a következő 2.2. ábrán látható.

Az algoritmus jól párhuzamosítható, ami a nagyteljesítményű multiprocesszoros környezetben kedvező futási időt eredményezhet. A párhuzamos program létrehozásának segítségére az OpenCL keretrendszert választottam, aminek a bemutatása következik.



**2.2. ábra.** Az adaptív küszöb módszerrel detektált részecskék momentum módszerrel számított pozíciója. **Bal oszlopban** az eredeti mérési kép egy részlete, **középső oszlopban** a detektálás eredménye és a ROI, az **utolsó oszlopban** az eredeti mérési képen a ROI és a detektált részecske pozícióját jelző kereszt.

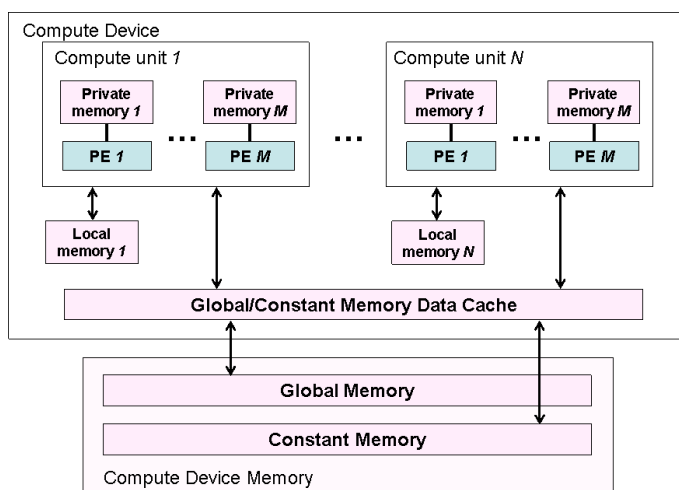


## 3. fejezet

# A multiprocesszoros OpenCL környezet

### 3.1. OpenCL architektúrája

Az Open Computing Language (OpenCL) keretrendszer [7] általános modellt, magas szintű programozási interfészt és hardware absztrakciót nyújt a fejlesztőknek adat- vagy feladat párhuzamos számítások gyorsítására különböző számítógépsímen (CPU, GPU, FPGA, DSP, ...). A hárdevgyártók 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 szátra 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 eszközöböl a kívánt eredményt.



3.1. ábra. OpenCL device architektúra (forrás: [7])

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 mondható, 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

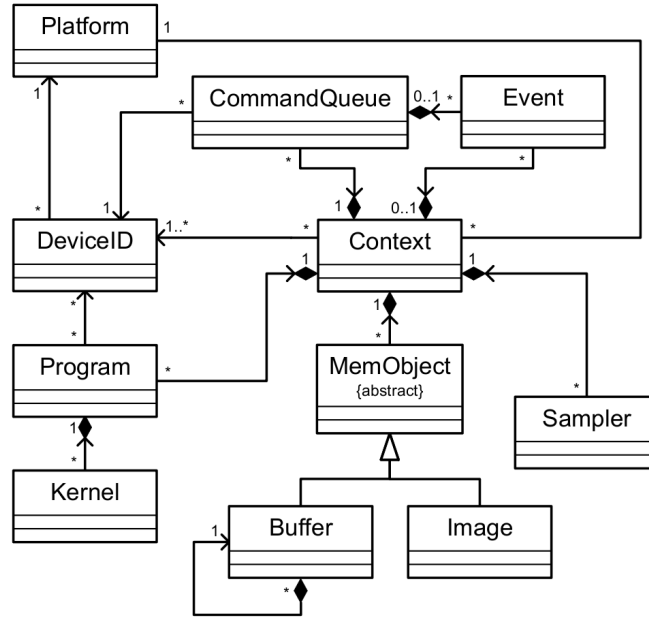
Ahhoz, hogy a rendszerben rejlő teljesítményt kihozzuk három fontos kérdést kell a szimulátor magjának implementálásakor megválaszoznunk:

- *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. ábra) 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





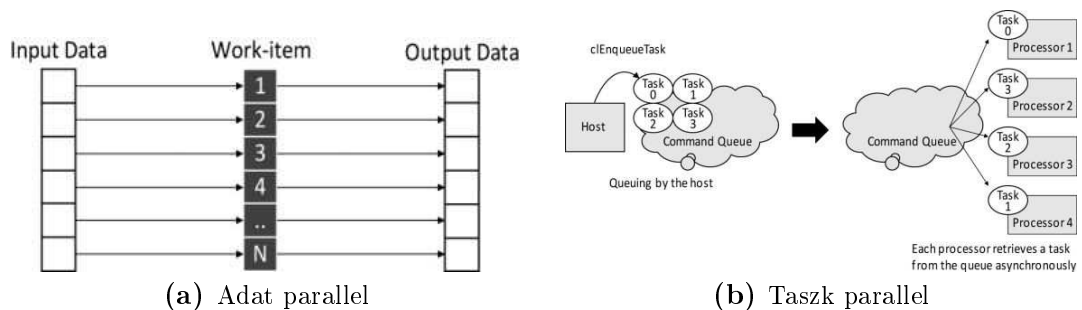
3.2. ábra. OpenCL context osztálydiagrammja (forrás: [7])

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 részéhez 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 lazán függenek össze.

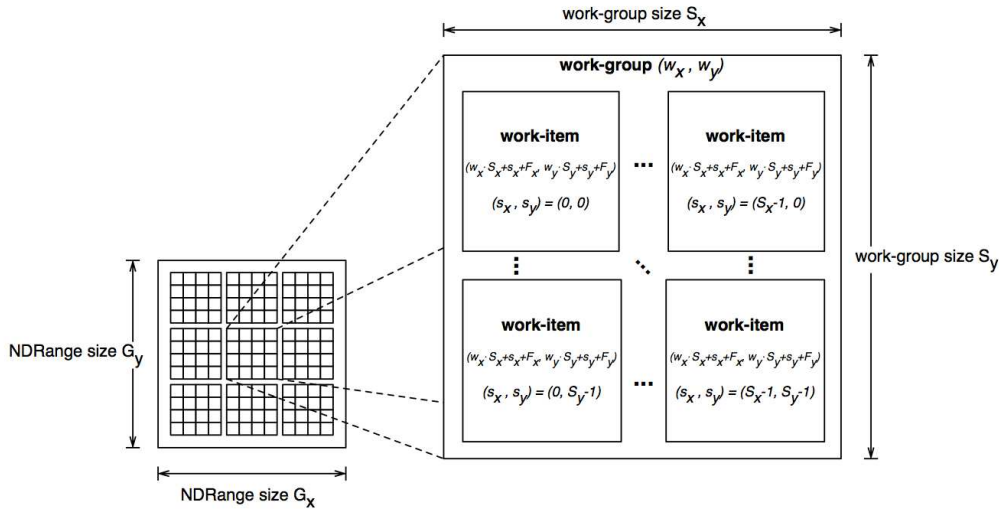


3.3. ábra. Feladat hozzárendelése work-item-hez (processzorhoz)

A processzor-magok megfelelő kihasználtságának elérése végett több ezer work-item virtuálisan osztozik rajta. Továbbá ezen work-item-eket work-group-okba ren-

dezzük.

A work-itemeket jelen pillanatban az OpenCL specifikációja [7] szerint 3 dimenziós work-group-ba tudjuk rendezni. A következő 3.4. ábrán egy 2D-s példát láthatunk egy work-item indexének a globális és lokális megfelelőjére.



**3.4. ábra.** 2D-s work-item-ek work-group-ba rendezése és indexelése  
(forrás: [7])

A work-group-okba rendezés a lokális memória jogosultsága miatt érdekes. Konkrétan az 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 processzormagok megfelelően legyenek az adatokkal táplálva.

### 3.3. Futási környezet bemutatása

A következő eszközök teljesítményét vizsgálom:

- A laptopomban található **Intel Core i5 M520** processzor,
- Asztali PC-ben található **Intel Xeon** processzor,
- Asztali PC-ben található **Intel Xeon Phi** co-processzor [8, 9],
- Asztali PC-ben található **nVidia GTX 590** videokártya.

Ezen eszközök legjelentősebb paraméterei a 3.2 táblázat tartalmazza.

**3.2. táblázat.** *Használandó eszközök összehasonlítása*

	Intel Core i5	Intel Xeon	Xeon PHI	nVidia GTX 590
MAX COMPUTE UNITS	4	8	224	16
MAX CLOCK FREQUENCY	2400	3000	1100	1225
MAX WORK GROUP SIZE	8192	8192	8192	1024
GLOBAL MEM SIZE	$\sim 4Gbyte$	$8Gbyte$	$\sim 4.5Gbyte$	$\sim 1.5Gbyte$
MAX MEM ALLOC SIZE	$\sim 2Gbyte$	$\sim 8Gbyte$	$\sim 1.5Gbyte$	$\sim 0.4Gbyte$
LOCAL MEM SIZE	$32Kbyte$	$32Kbyte$	$32Kbyte$	$48Kbyte$

Az összehasonlíthatóság végett a legkisebb memóriájú eszközre fogom a problémát skálázni. Tehát maximálisan  $32Kbyte$  lokális memóriát fogok használni. A többi eszköz memóriája nagyobb, így a kód mindegyiken tud futni.



## 4. fejezet

### A host program bemutatása

A 100 FPS-el érkező képeket az eszköz globális memóriájának méretét figyelembe véve dolgozom fel. Ha az eszköz memóriájába 100 képnél kevesebb fér be, akkor a másodperc fennmaradó képei eldobásra kerül. A megjelenítésnek nem fontos szigorúan real-time működésűnek lennie (soft real-time), adott fokú késleltetés megengedhető, a határidő elmulasztása nem jár súlyos következménnyel. A program ciklikusan a következő felsorolásban olvasható lépéseket hajtja végre. A lépések a későbbi 4.1 részben ismertetettek végett párhuzamosan időben átlapolódva történnek.

1. Eszközön futtatandó kernelek inicializálása, argumentumainak beállítása,
2. Kép fogadása (gyűjtése) a kamerától GigE porton keresztül (beolvasása a host-memóriába),
3. Képek leküldése a host-memóriájából az eszköz globális memóriájába,
4. Kernelek futtatása az eszközön:
  - (a) A másodperc első képének (medián) szűrése,
  - (b) Átlag és szórás számítása az eredeti és a szűrt kép különbségén (differenciális kép),
  - (c) Adaptív detektálási szint előállítása,
  - (d) Az első és a fennmaradó képeken detektálás.
5. Kernelek futása után az eredmény az eszköz globális memóriájából a host-memóriájába való visszatöltése,
6. Posztprocesszálas és OpenGL megjelenítés.

A kernel megírása során a korábbi 3. fejezetben említetteket figyelembe kell venni. Főként a véges lokális és globális memóriát és a work-ítemek számát. A kernelek adat-parallel módon lett megírva.

## 4.1. A host program párhuzamos felépítése

A korbábban megengedett késleltetésre és a real-time viselkedésre több tényező rossz hatással van, ezek a következők:

- A kamera GigE interfészének jittere,
- Operációs rendszer által futtatott további folyamatai,
- Feldolgozó (szűrő és detektáló) algoritmus futási idejének (ET = execution time) változatkozása.

A kernelek közül a medián szűrő, ami elrontja a fix futási időt (Fix E. T.) és azt véletlenné teszi. A változó futási időről elmondható, hogy a bemeneti kép értékeitől függ. Pontosabban a medián szűrő ablakain belül található pixelek rendezettségétől. Hiszen minél rendezettebb, annál gyorsabban található meg a mediánját. A medián számításának worst case execution time-ja (WCET), pontosabban wc. lépésszáma ismert algoritmuselméletből, ami korlátos és megegyezik - a bemenet  $N$  számossága esetén - a  $O(N \log N)$  értékkel.

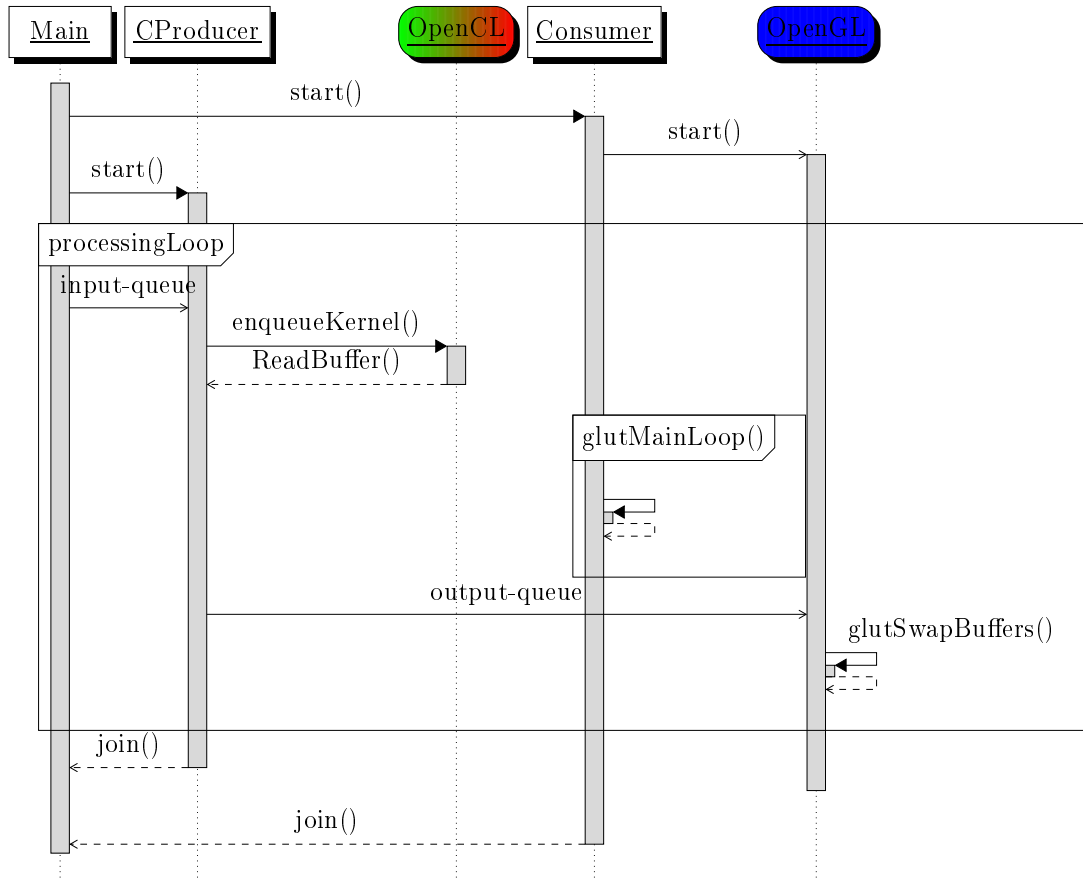
Adódik, hogy a host programot több konkurens szálra bontva kerüljön implementálásra az ismert Producer-Consumer [10] mintát alkalmazva. Ezzel elérhető, hogy a program feldolgozási sebessége ne a kép fogadásának és a kernel futási idejének összege legyen, hanem ezek közül a időben rövidebbig. Ez viszont szükségessé teszi, hogy az adatkapocsot a két szál között várakozási listával szükséges megoldani. Így a szálak egymásra várásának csökkentését lehet elérni.

*A szálak létrehozását és kezelését a Boost C++ Libraries [11] keretrendszer megfelelő függvényhívásai oldják meg. A keretrendszer parancssori argumentumkezelést, szálkezelést, szemafort, várakozási listát és atomi működésű operátorokkal rendelkező változókat nyújt a programozó számára. A szálak közötti adatkapcsolást FIFO típusú (single producer - single consumer) várakozási listával oldom meg.*

A main szálon kerül implementálásra a kamera képeinek fogadása és a várakozási sorba állítása <sup>1</sup>. További consumer és egyben producer szálon a várakozási sorban található kép elővétele majd feldolgozására kerül sor az OpenCL kernel által, aminek eredménye a megjelenítő OpenGL bufferébe kerül letárolásra. Végül egy consumer szál a bufferben található eredmény OpenGL-es megjelenítést végzi.

Az implementálandó szálak „szekvencia diagrammja” a következő 4.1 ábrán látható. Az ábra alapja UML szekvencia diagramm, amit kiegészítettem az OpenCL kernel és az OpenGL callback függvényének futásával. Az ábrán három szál látható ezek a Main, CProducer, Consumer. A szálak részletes ismertetése a következő részben következik.

<sup>1</sup>Természetesen a main szál a program felhasználó általtörténő futtatása által meghívott függvény, így az említettek előtt még inicializáció is történik. Ezt később részletezem. A szálra producer szálként kell tekintenünk a kamera képeinek fogadása és várakozási listába tétele végett.



4.1. ábra. Host program „szekvencia diagrammja”

## 4.2. Main (producer) szál

A program felhasználó által való futtatása által jön létre és a program `main()` függvényét hajtja végre. A szál a parancssori argumentumok feldolgozását, az inicializálásért és a kamera adatfolyamának fogadásáért és letárolásáért felel. A paraméterek tárolására és konzisztenciájának megőrzésére definiáltam a következő osztályt:

```

class Params {
public:
    ...
    bool    only_global;    // csak globalis memoria hasznalata
    ...
    uint    file_N;         // file merete def.: 1024
    uint    nh_N;           // median ablakanak merete !!! paratlan !!!
    uint    tail;           // a szures altal létrejovo pixelek a kep szelen
    uint    Bfile_N;        // az ily kapott kep meret

    uint    pplN;           // kivant local meret

    uint    localN;         // work-group nagysage
    uint    globalN;        // osszes work-item szamossaga

    ulong   aSize;          // eszkoz global memoria max alloc. merete byte-ban
    ulong   lSize;          // eszkoz local memoria meret byte-ban
}
  
```

```

    ulong      mCuint;          // eszkoz max compute unit szama
};

```

A paraméterek értelmezése a következő:

**only\_global** Csak globális memória használata vagy lokálisat is használjon. Adott eszközök esetén a lokális memória a globális memóriába van mappelve, így használata csupán felesleges adatmozgatást jelentene `only_global = 0`,

**file\_N** A kép 2D-s mérete, `file_N = 1024`,

**nh\_N** A medián szűrő mozgó ablakának 2D-s mérete (páratlan szám) pl.: `nh_N = 3, 5, 7, 9` (magasabb fokú szűrőt nem érdemes használni, mivel nagy nemlinearitással rendelkezik),

**tail** a szűrés által létrejövő pixelek a kép szélén, azaz `tail = (nh_N - 1) / 2`,

**Bfile\_N** az így kapott kép 2D-s mérete, azaz `Bfile_N = file_N + 2*tail`,

**pplN** javasolt lokális méret, pl.: a compute unit-ok száma,

**localN** a tényleges lokális méret egy work-group-on belül,

**globalN** az összes work-item számossága,

**aSize** az eszköz globális memóriájában allokalható maximális memória méret  
(`CL_DEVICE_MAX_MEM_ALLOC_SIZE`),

**lSize** az eszköz lokális memóriájának mérete  
(`CL_DEVICE_LOCAL_MEM_SIZE`),

**mCuint** az eszköz egyszerre futtatható szálának (compute unit) száma  
(`CL_DEVICE_MAX_COMPUTE_UNITS`).

## Inicializálás

A korábban említett várakozási lista fix méretű a 100 FPS-el érkező képek 1 másodpercnyi feldolgozásához szükséges méretű. Ennek megfelelően mivel két várakozási listára van szükség és a kamera egy képe minkettől „végigmegy”, így `BUFF_N=50` hosszúsággal kerül inicializálásra és debug esetén `Mimage` osztályt illetve release esetén `uint8_t*` pointereket tartalmaz, amik közvetlenül/közvetve a képre mutatnak. A képek  $\text{BUFF\_N} \times 1024 \times 1024$  `uint8_t` típusú tömbben kerül tárolásra. Az `Mimage` osztályt a következőképpen definiáltam:

```

class Mimage {
public:
    unsigned int    i;
    unsigned int    N;
    uint8_t         *ptr;

    Mimage();
    Mimage(unsigned int _i, unsigned int _N, unsigned char *_ptr);
};

```

Definiálásra és inicializálásra kerül két single-producer single-consumer várakozási lista, amibe az előbbi osztály példányai kerülnek. Továbbá minden lista mellett



a hozzá tartozó buffer tömb megcímzésére alkalmas atomi művelet végrehajtással rendelkező változó és a kölcsönös kizárást biztosító szemafor is definiálásra kerül:

```
uint8_t *input = new uint8_t[BUFF_N * pms.Bfile_N*pms.Bfile_N];
uint8_t *output = new uint8_t[BUFF_N * pms.Bfile_N*pms.Bfile_N];

boost::lockfree::spsc_queue<Mimage,boost::lockfree::capacity<BUFF_N>> input_queue;
boost::lockfree::spsc_queue<Mimage,boost::lockfree::capacity<BUFF_N>> output_queue;

boost::mutex input_mtx;
boost::mutex output_mtx;

boost::atomic<int> in_N(-1);
boost::atomic<int> out_N(-1);
```

Az `input_queue`-ba a kamera képéhez tartozó, az `output_queue`-ba az OpenCL-es feldolgozás utáni `Mimage` osztály példánya kerül. A szemaforok és az atomi változók a producer és a consumer szálak szoftveres/hardveres párhuzamos futása végett van szükség, elkerülve a szálak között fennálló versenyhelyzetet.

Ezután a Consumer és a Producer szál létrehozása (forkja) következik.

Végül a kamera inicializálására kerül sor, ami megfelelően konfigurált hálózati kapcsolat esetén megtalálja a kamera IP és MAC címét és ezzel inicializálja a hozzá tartozó, globális változóként deklarált osztályt. A kamera számunka fontos paraméterei ezután beállításra kerülnek, mint például a felbontás, FPS, expozíció és a küldött IP csomag mérete<sup>2</sup>. Ezen beállítások után a kamera stream-hez `StreamCBFunc` callback függvény kerül hozzárendelésre.

## Kamera adatfolyamának fogadása

Az `AcquisitionStart` parancs kiadása után megkezdődik a felvétel készítés. Adott frame megérkezésekor a korábbi callback függvény kerül meghívásra. Az `input` bufferbe történő mentése előtt a hozzá tartozó `input_mtx` szemafor által az erőforrást lefoglalja. A bufferbe történő mentés a hozzá tartozó `in_N` atomi változó által meghatározott indexű területre történik.

```
input_mtx.lock(); // input buffer (eroforras)
    lefoglalasa

in_N++;
in_N = in_N % BUFF_N; // cirkularis korbeforogas vegett

Mimage iim(in_N, pms.Bfile_N, &input[in_N*pms.Bfile_N*pms.Bfile_N]);

while(!input_queue.write_available()) {} // a buffer/varakozasi lista
    kiurulesere varas

if((*pAqImageInfo).iImageSize != pms.file_N*pms.file_N) {
    std::cout << std::endl << "wrong camera image size" << std::endl;
```

---

<sup>2</sup>Növelésével az IP csomag fejléce okozta overhead és CPU kihasználtság csökkenthető (Jumbo packet).

```

        exit(EXIT_FAILURE);
    }

    /*
    * frame elmentese in_N-edik helyre
    */
    for(uint a = 0; a < pms.file_N; a++) {
        for(uint b = 0; b < pms.Bfile_N; b++) {
            input[in_N*pms.Bfile_N*pms.Bfile_N + (a+pms.tail)*pms.Bfile_N + (b+pms.
            tail)] = (*pAqImageInfo).pImageBuffer[a*pms.file_N + b];
        }
    }

    while(!input_queue.push(iim)) { ; } // a varakozasi lista berakas

    input_mtx.unlock(); // eroforras felszabaditasa (consumer mostmar dolgozhat rajta)

```

## 4.3. CProducer szál

Ezen szál feldolgozza az `input_queue` várakozási listában található képeket az OpenCL kernelek meghívásával és annak eredményét az `output_queue`-ba sorakoztatja fel a későbbi megjelenítés végett.

### 4.3.1. OpenCL inicializálás

Az inicializáláshoz első körben szükség van az eszköz fontosabb tulajdonságaira. Az eszköz globális, lokális memóriájának mérete, a globális memóriában maximálisan allokalható memória mérete és a compute-unite-ok száma.

#### Globalis memória mérete

A globális memóriában a következőknek kell elférnie:

- a képek (`[file_N][file_N]`),
- a szűrt képek (`[Bfile_N][Bfile_N]`),
- a detektálás után megjelölt pixelek (`[file_N][file_N]`).

#### Lokális memória mérete

A lokális memória nagy sebessége és gyors elérése végett alapvető a preferáltsága a párhuzamos applikációkban. A program futása során kerül megállításra a használt értéke.

A szűrt kép minden pixelének kiszámításához egy work-item-et rendelünk, így egy work-item-hez a medián szűrő ablakának megfelelő méretű,  $nh\_N \times nh\_N$  darab lokális memóriát rendelünk. Ezáltal a szűrés teljes mértékben a lokális memóriában történik, ezzel lehet optimális OpenCL kódot írni.

### 4.3.2. Kép kernelekkel történő feldolgozása

A kernelek fordítása és a megfelelő argumetnumok beállítása után először a medián szűrés, majd a detektálási szint számítása végül a detektálás történik. Hibakeresés során a közbülső eredmények is visszaolvasásra kerül.

## 4.4. Consumer szál

Először az OpenGL inicializálás és a megjelenítő ablak létrehozása történik. Ezután a rajzolásra és az időzítésre alkalmas callback függvények kerülnek regisztrálásra. Az időzítő függvény az aktuális megjelenítés i frekvenciát átlagolással számítja továbbá ennek megfelelően beállítja a következő rajzolás határidejét. A rajzoló callback függvény a kimeneti `output_queue`-ből kivesz egy képet (elemet), majd azt az OpenGL bufferébe másolja. Továbbá a várakozási lista annyi elemét törli ki (dobja el), hogy a korábban mért FPS érték szerint mind megjelíthető legyen.



## 5. fejezet

# A kernel programok lépéseinek bemutatása

Az összes work-item azonos kernel programot futtat párhuzamosan, ami következtében a program első lépéseként le kell kérdeznie a work-item indexeit. Ezt a következő parancsokkal lehetséges megtenni:

```
//----- item-indexes -----  
// group-ids  
size_t ggi = get_group_id(0);  
size_t ggj = get_group_id(1);  
// global-ids  
size_t Gi = get_global_id(0);  
size_t Gj = get_global_id(1);  
// local-ids  
size_t li = get_local_id(0);  
size_t lj = get_local_id(1);  
  
//----- dimensions -----  
size_t lN = get_local_size(0);  
size_t gN = get_num_groups(0);
```

Ezután a kernelek különböző feladatokat látnak el, amik bemutatása következik.

### 5.1. Medián szűrés

A szűrő feladata a részecskék eliminálása illetve a háttér lehető legalakhűbb átengedése, amjd a differenciális kép létrehozása.

A kernel függvényének fejléce a következő:

```
__kernel void median(    uint    N,        // 0  
                        uint    file_N,    // 1  
                        uint    nh_N,      // 2  
                        __global uchar *im, // 3 :: [N][Bfile_N][Bfile_N]  
                        __global uchar *Oim, // 4 :: [N][file_N][file_N]  
                        __local  uchar *Lwork // 5 :: [lN][lN][nh_N][nh_N]  
)
```

Az argumentumok értelmezése a következő

**N** az **im** bufferben tárolt, szűrendő képek száma,  
**file\_N** a képek oldalhossza pixelben,  
**nh\_N** a szűrő mozgó ablakának mérete,  
**im** a szűrendő (bemeneti) képet tartalmazó globális bufferre mutató pointer,  
**Oim** az eredeti és a szűrt kép különbségét tartalmazó globális bufferre (kimenet) mutató pointer,  
**Lwork** a szűrés során használt lokális memóriaterületre mutató pointer.

A kernel program lépései a következők:

1. A work-item globális és lokális indexének meghatározása,
2. A kép a szűrő ablakjának megfelelő ( $\text{nh\_N} \times \text{nh\_N}$ ) részének a globális (**im**) bufferből a lokális **Lwork** bufferbe való másolása,
3. Lokális bufferben kiválasztásos részleges sorbarendezés. Részleges alatt a kép-részlet feléig történő rendezést értem,
4. Medián megállapítása és a kimeneti **oim** bufferbe írása.

Továbbá a kimenet állítása során megvizsgálom, hogy a szűrő kimeneti értéke az eredetinél alacsonyabb vagy magasabb. Ha magasabb akkor esélyes, hogy az adott indexű pont környezetében nem volt részecske. Ennek megfelelően a szűrt értéket eldobom és az eredeti értéket változatlanul írom a kimeneti bufferbe. Ez a medián szűrő nemlineárisát csökkenti, hiszen a részecskéktől (pozitív kiugrásoktól) mentes területet alakhűbben (változás mentesen) engedi át.

## 5.2. Átlagolás

Az adaptív döntési szint számításához szükség van a differenciális kép átlagára és a szórására. A számítás során a képet az eszköz **CL\_DEVICE\_MAX\_COMPUTE\_UNITS** paraméterének megfelelően osztom fel részekre. A részösszegeket, így párhuzamosan tudom számítani. A kernel függvényének fejléce a következő:

```

__kernel void average(    uint    N,          // 0
                          uint    file_N,      // 1
                          __global uchar *Oim,  // 2 :: [N][file_N][file_N]
                          __global uint *Oavg,   // 3 :: [N][file_N][1N]
                          __local  uchar *Lwork // 4 :: [xN][yN]
)
  
```

A kernel program lépései a következők:

1. A work-item globális és lokális indexének meghatározása,
2. A képet a work-item-ek számának megfelelő részre bontom,
3. A képrészletek a globális **Oim** bufferből a lokális **Lwork** bufferbe való másolása,
4. Az átlag számításához ezek összegét számítom és a kimeneti **Oavg** bufferbe írom.

A részösszegekből a host oldalon történik az átlag számítása.

### 5.3. Detektálás

1. A work-item globális és lokális indexének meghatározása,
2. Az adaptív külszöb számítása és ennek megfelelő detektálás,
3. Kiterjesztés és a flood-fill algoritmussal a ROI meghatározása:
  - (a) Megjelölt pixel keresése,
  - (b) A megjelölés adott számú környezetére való kiterjesztése,
  - (c) A kiterjesztés során a két legtávolabbi pont lesz a ROI (region of interest) határpontjai.
4. ROI-n belüli pontokból a részecske pozíciójának számítása momentum módszerrel,
5. Eredmény mentése a globális memóriába.





## 6. fejezet

# Haználati útmutató és vizsgálat

```
$ ./pp-track -h
pp-track: Real-time particle detect and distribution display

Allowed options::
-h [ --help ]           produce this help message
--vendor arg            arg --> {amd, intel, nvidia}
--device arg           arg --> {cpu, gpu, acc}
-v [ --verbose ]       verbose mode
-c [ --convert ]       convert output raw files to tiff
```

Medián szűrés összehasonlítása csak global illetve global és local memória használata esetén.



## 7. fejezet

# Összehasonlítás


A programot a különböző eszközökön futtatva a 7.1. táblázatban látható futási eredményeket produkálta. A táblázatban látható futási idők 100 futási idő átlaga. A táblázathoz felvettem egy fiktív mérőszámot (teljesítmény tényező) a különböző architektúra összehasonlítására. A mérőszám értéke minél kisebb, annál gyorsabban hajtódik végre egy utasítás.

**7.1. táblázat.** *Az eszközök erőforrásainak és a rajta futtatott programok futási idejének összehasonlítása.*

	Intel Core i5 M520	nVidia GT330M
MAX COMPUTE UNITS [1]	4	6
MAX CLOCK FREQUENCY [MHz]	2400	1265
MAX WORK GROUP_SIZE	8192	512
GLOBAL MEM SIZE	~ 4 GByte	~ 1 GByte
LOCAL MEM SIZE	32 KByte	16 KByte
Futási idő ( $T$ )	478.71 ms	191.94 ms
Teljesítmény tényező $\left(P = \frac{1}{\text{UNITS} \times \text{FREQUENCY}}\right)$	$104.16 \cdot 10^{-6}$	$131.75 \cdot 10^{-6}$
Fajlagos utasításszám ( $T/P$ )	$4.59 \cdot 10^3$	$1.45 \cdot 10^3$

Látható, hogy a GPU-n való futtatás közel  $3\times$  gyorsulást jelent. Ezt két dolognak tudom be:

- *Memória:* A CPU memóriája DDR3 @ 1066 MHz 64 bites busz szélességgel, míg a GPU memóriája GDDR3 @ 1066 MHz 128 bites busz szélességgel,
- *Processzor mag:* A CPU 4 compute unit-al rendelkezik, ami 4 szála, ami 2 processzormagra az Intel HyperThread technológiájával képeződik le, míg a



GPU 6 compute unit-al rendelkezik, ami 48 CUDA core-ra képeződik le.

Továbbá figyelembe kell venni, hogy a program futása a többi programmal konkurrensten történik, CPU esetén az operációs rendszerrel, GPU esetén a megjelenítéssel.

## 8. fejezet

# Összegzés

Dolgozatomban bemutattam a poros plazma kísérletek apparátusát. A kísérlet során a kristályrácsba rendeződő részecskékről egy nagysebességű kamerával fényképek készülnek. A dolgozatomban ezen képeket, kellett feldolgoznom és a részecskék pozícióját detektálnom. A pozíciók a fizikai modell/szimuláció validálására szolgálnak.

Ismertettem a részecske detektálásának módszerét szűrés és adaptív döntési küszöb használatával. Az elterjedt FIR Gauss szűrő helyett a hatékonyabb medián szűrőt javasoltam és alkalmaztam. A pozíció számítására a momentum módszert implementáltam, ami nagyobb számítási energiát igényel, de szubpixeles felbontást tudtam vele elérni. Konstatáltam, hogy az így kialakult program masszívan párhuzamosítható.

Ezután áttekintettem az OpenCL keretrendszert, amit a párhuzamos program megírásának segítségére használtam. Az itt ismertetett megállapításokat figyelembe véve állítottam össze a párhuzamos program lépéseit, amit részleteztem is.

Végül az elkészült programot CPU-n és GPU-n is futtatva a futási idejüket összevetettem és azonosítottam a gyorsulás forrását kitérve a processzormagra és a memóriájára.

### További feladatok:

- A host-program real-time mérésbe helyezése egy producer-consumer sémájú szál megoldás alkalmazásával,
- Az eredmény grafikus megjelenítése pl.: OpenGL használatával,
- Az OpenCL szabvány által specifikált vektor műveletek támogatásának kiaknázása, ami az Intel Xeon PHI processzorkártyában rejlő teljesítményt ki tudná aknázni.

## A. függelék

# Fejlesztőkörnyezet összeállítása

OpenCL kód fejlesztése történhet Windows alatt NVIDIA Nsight Visual Studio Edition [12] és Linux alatt GCC-vel [13]. 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 [14] esett a választás, ami csupán a fejlesztőkörnyezet egyszerűbb telepítése végett történt így.

## Software Development Kit-ek (SDK) telepítése

### nVidia támogatás telepítése

A legtöbb mai Linux disztrók tartalmazznak 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 inicializálásáért 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 [15]. 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. Legcélszerűbb, 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/opencv/bin
export LD_LIBRARY_PATH

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

INTELOCLSDKROOT=/opt/intel/opencv
export INTELOCLSDKROOT
```

Mivel az nVidia limitálja a kernel futási időt 5 másodpercen 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 [16]. A kicsomagolás után az ./install-cpu.sh program futtatásával telepíthető. Ezután még szükséges a LD\_LIBRARY\_PATH beállí-

tása.

## **Eclipse – Integrated Developement Environment**

A fejlesztés és hibakeresés egy Integrated Developement Enviroment (IDE) segítségével könnyebb. Az open source Eclipse [17] 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.

## **Új (Hello World) projekt létrehozása**

Az OpenCL fejlesztését konyhanyelven bemutató OpenCL Programming Guide [18] könyvben szereplő Hello World programot a következő linken lehet letölteni [19]. A kód fordítása előtt egy Eclipse projektet létrehozunk és a fordításhoz szükséges beállításokat elvégezzük.

## **Empty C project létrehozása**

Először egy üres C projektet hozunk létre, ami folyamatát a A.1 ábrán látjuk. A korábban említettek szerint fordítónak a Linux GCC-t állítjuk be.

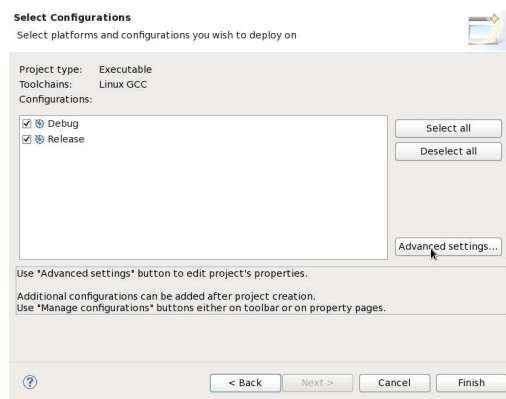
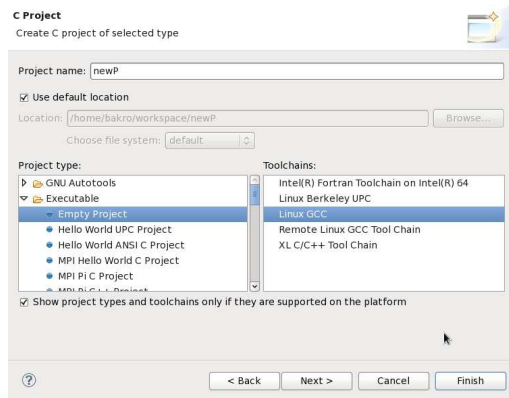
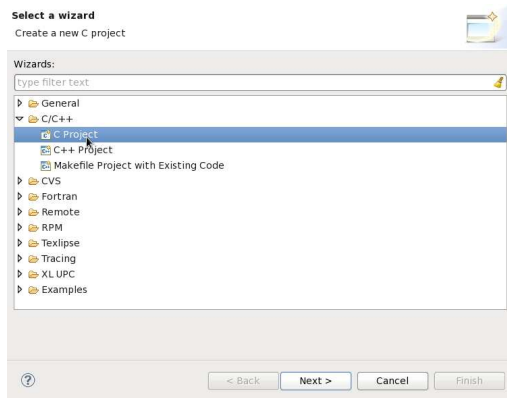
## **Compiler beállítása**

A létrehozott projektre jobb gombbal kattintva a tulajdonságára kattintva állíthatjuk be a fordítót a A.2 ábrának megfelelően. A beállítások kiterjednek a GNU-C99 nyelv szerinti fordításra és a korábbi részben telepített SDK-ban található include mappa beállítására.

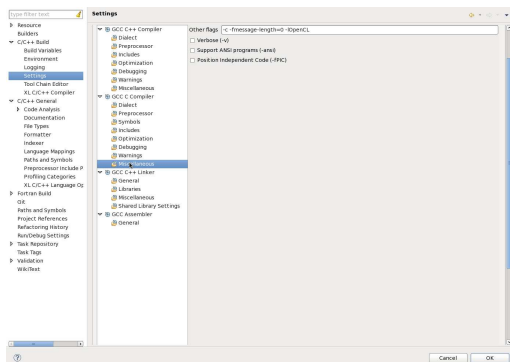
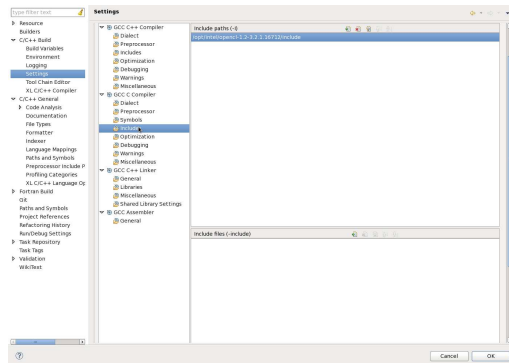
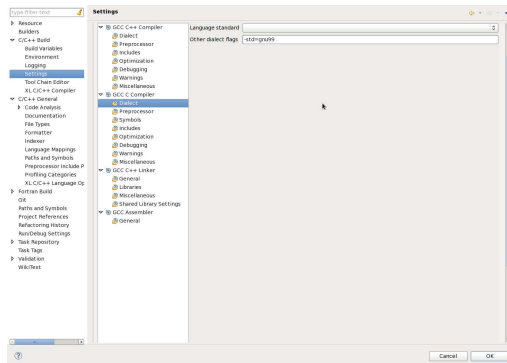
## **Linker beállítása**

A linkert a A.3 ábra szerint állítjuk be.

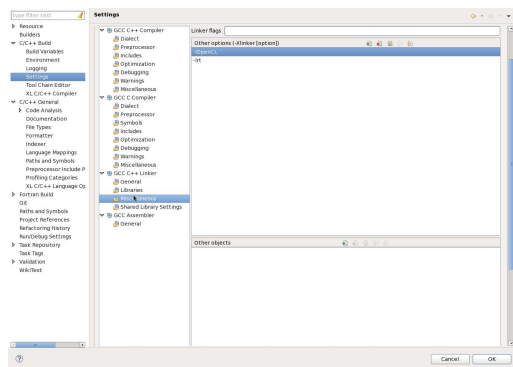
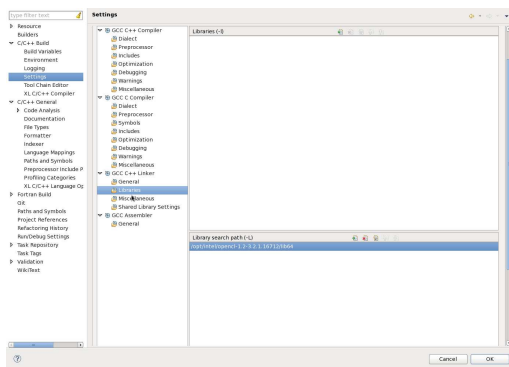




A.1. ábra. Új Eclipse projekt létrehozása



A.2. ábra. Compiler beállításai



A.3. ábra. *Linker beállításai*



# Ábrák jegyzéke

1.1. A mérési elrendezés sematikus ábrája . . . . .	3
1.2. A konkrét kamra működése közben . . . . .	4
1.3. Kamera vezérlő szoftvere . . . . .	5
1.4. Két különböző kísérlet során készült fénykép. . . . .	6
2.1. Adaptív küszöb bemutatása . . . . .	9
2.2. Pozíciómérés momentum módszerrel . . . . .	11
3.1. OpenCL device architektúra . . . . .	13
3.2. OpenCL context osztálydiagrammja . . . . .	15
3.3. Feladat hozzárendelése work-item-hez (processzorhoz) . . . . .	15
3.4. 2D-s work-item-ek work-group-ba rendezése és indexelése . . . . .	16
4.1. Host program . . . . .	
A.1. Új Eclipse projekt létrehozása . . . . .	IV
A.2. Compiler beállításai . . . . .	IV
A.3. Linker beállításai . . . . .	V



# Táblázatok jegyzéke

3.1. OpenCL memória szintek . . . . .	14
3.2. Használandó eszközök összehasonlítása . . . . .	17
7.1. Eszközök futási idejének összehasonlítása . . . . .	33



# 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] Y. Feng, J. Goree, and B. Liu, „Accurate particle position measurement from images,” *Review of Scientific Instruments*, vol. 78, no. 5, pp. –, 2007.
- [6] *Tracking interacting dust: comparison of tracking and state estimation techniques for dusty plasmas*, vol. 7698, 2010.
- [7] The Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0*, 6 August 2010.
- [8] Intel, „Intel® xeon phi™ product family.” <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html>.
- [9] „Intel® many integrated core architecture (intel® mic architecture).” <http://www.intel.com/content/www/us/en/architecture-and-technology/many-integ>
- [10] E. W. Dijkstra, „Information streams sharing a finite buffer,” *Inf. Proc. Letters*, vol. 1, pp. 179–180, 1972.
- [11] „Boost c++ libraries.” <http://www.boost.org/>.
- [12] „Nvidia nsight visual studio edition.” <https://developer.nvidia.com/nvidia-nsight-visu>



- [13] „Gcc, the gnu compiler collection - gnu project - free software foundation (fsf).”  
<http://gcc.gnu.org/>.
- [14] „Centos project.” <http://www.centos.org/>.
- [15] „Nvidia drivers.” <http://www.nvidia.com/Download/index.aspx>.
- [16] „Intel opencl sdk.” <https://software.intel.com/en-us/vcsource/tools/opencl-sdk-xe>.
- [17] „Eclipse - the eclipse foundation open source community website.”  
<https://www.eclipse.org>.
- [18] A. Munshi, *OpenCL Programming Guide*. Addison-Wesley, 2011.
- [19] „Opencl programming guide - examples.” <https://code.google.com/p/opencl-book-samples/sc>