

# SZAKDOLGOZAT



MISKOLCI EGYETEM

## A MySQL adatbázismotor teljesítményének optimalizálása OpenCL segítségével

Készítette:

Várady Dániel

Programtervező informatikus

Témavezető:

Piller Imre

MISKOLC, 2021

**MISKOLCI EGYETEM**

Gépészszmérnöki és Informatikai Kar

Alkalmazott Matematikai Intézet Tanszék

**Szám:**

## **SZAKDOLGOZAT FELADAT**

Várady Dániel (CGW85R) programtervező informatikus jelölt részére.

**A szakdolgozat tárgyköre:** relációs adatbázis, MySQL, OpenCL

**A szakdolgozat címe:** A MySQL adatbázismotor teljesítményének optimalizálása OpenCL segítségével

### **A feladat részletezése:**

*A MySQL adatbázismotor egy széles körben használt, relációs adatmodellre épülő alkalmazás. A dolgozat célja, hogy megvizsgálja azokat az eseteket, amelyekben a számítási teljesítményt az OpenCL nyelv segítségével a több számítási maggal rendelkező konfigurációk esetén javítani lehet. A dolgozatban ehhez részletesen be kell mutatni a MySQL adatbázismotor felépítését, működésének a folyamatát, az SQL lekérdezések feldolgozási lépései. Példákat kell adni olyan lekérdezésekre, amelyek párhuzamosan végrehajthatók. Meg kell vizsgálni azt, hogy a feldolgozandó adatokat milyen struktúrákba célszerű rendezni, és azokat hogyan érdemes mozgatni a hatékonyabb számítások érdekében. Az eredmények szemléltetéséhez méréseket kell végezni adott hardveres konfiguráció mellett, amellyel objektív módon láthatóvá válnak az optimalizáció előnyei.*

**Témavezető:** Piller Imre (egyetemi tanársegéd)

**A feladat kiadásának ideje:** 2019. szeptember 27.

.....  
szakfelelős

## EREDETISÉGI NYILATKOZAT

Alulírott **Várdy Dániel**; Neptun-kód: CGW85R a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *A MySQL adatbázismotor teljesítményének optimalizálása OpenCL segítségével* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ..... .év ..... .hó ..... .nap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

.....

dátum

témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás):

.....  
.....  
.....

konzulens (dátum, aláírás):

.....  
.....  
.....

3. A szakdolgozat beadható:

.....

dátum

témavezető(k)

4. A szakdolgozat ..... szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....  
..... egyéb mellékletet (részletezve)

.....

tartalmaz.

.....

dátum

témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíró neve: .....

.....

dátum

szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....

a bíró javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, .....

a Záróvizsga Bizottság Elnöke

# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>1</b>
<b>2. A MySQL telepítése forráskódból</b>	<b>2</b>
<b>3. SQL lekérdezések végrehajtása</b>	<b>4</b>
3.1. Execution Plan . . . . .	4
3.1.1. Az EXPLAIN parancs . . . . .	4
3.2. Teszt adatbázis megtervezése . . . . .	4
3.2.1. A táblák feltöltése . . . . .	6
3.3. EXPLAIN a gyakorlatban . . . . .	7
3.3.1. Egyszerű lekérdezés . . . . .	7
3.3.2. Csoportosítás . . . . .	8
3.3.3. Kapcsolt táblák . . . . .	9
<b>4. A MySQL Connector</b>	<b>11</b>
4.1. A Connector/C++ használata . . . . .	11
4.2. Első program . . . . .	12
<b>5. Párhuzamosítási lehetőségek</b>	<b>13</b>
5.1. Az OpenCL nyelv . . . . .	13
5.2. Az OpenCL elemei . . . . .	14
5.2.1. Eszköz modell . . . . .	14
5.2.2. Feldolgozási modell . . . . .	15
5.2.3. Munkacsoportok . . . . .	16
5.2.4. OpenCL program általános felépítése . . . . .	16
5.2.5. Az OpenCL telepítése . . . . .	17
<b>6. Megvalósítás</b>	<b>18</b>
6.1. Adatok előkészítése . . . . .	18
6.1.1. Táblák tömbbé alakítása . . . . .	18
6.1.2. A táblákat beolvasó függvény . . . . .	19
6.2. Globális és lokális méret meghatározása . . . . .	20
6.2.1. Működési elv . . . . .	20
6.2.2. Globális méret meghatározása . . . . .	21
6.2.3. A kernelkód működése . . . . .	21
6.3. Eredmény kiolvasása . . . . .	22
6.3.1. Az elemek kiolvasása egyben . . . . .	22
6.3.2. Az elemek kiolvasása szakaszosan . . . . .	22
6.4. Összetett lekérdezés . . . . .	24

<b>7. Mérések, összehasonlítások</b>	<b>25</b>
7.1. Adatok mozgatásának sebességei . . . . .	25
7.1.1. A MySQL lekérdezés sebessége . . . . .	25
7.1.2. A query válaszának átmásolása a saját tömbbe . . . . .	26
7.1.3. Adatok bemásolása a pufferekbe . . . . .	27
7.1.4. Adatok kimásolása a pufferekből . . . . .	28
7.1.5. A globális méretből adódó sebességek . . . . .	30
7.2. Gyakorlati példák és következtetések . . . . .	32
7.2.1. Egy táblás lekérdezés . . . . .	33
7.2.2. Két táblás lekérdezés . . . . .	35
<b>8. Összefoglalás</b>	<b>37</b>
<b>Irodalomjegyzék</b>	<b>38</b>

# 1. fejezet

## Bevezetés

Napjainkban viszonylag kevés módszer áll rendelkezésünkre, ha egy adatbázis működését optimalizálni szeretnénk.

Az adatbázis rendszerek képesek megmutatni, miként fognak végrehajtani egy-egy lekérdezést, így ezt felhasználva lehetőségünk van olyan módosításokat végrehajtani amelyekkel javíthatjuk a teljesítményt. De sajnos bizonyos esetekben ez olyan költségekkel járhat, mint az adatbázis teljes újratervezése.

A hatékonyúság növelésének egy megoldása lehet egy saját kliens alkalmazás elkészítése. Ezzel olyan módon szeretnénk csökkenti a lekérdezések sebességét, hogy a számítás igényes műveleteket, nem az adatbázis szerver, hanem a kliens gép erőforrásával végeztetjük el. Ekkor kerül elő az a felvetés, hogy ki lehetne -e használni a grafikuskártya köztudottan magas számítási teljesítményét.

Ennek a kiemelkedő feldolgozó képessége a sok processzormagból és az ebből adódó párhuzamosságból adódik. Jelenleg is használják ezeket az eszközököt különféle számítások elvégzésére, például kripto pénzek bányászatakor, amikor úgy nevezett HESH -eket állítanak elő. Egyes esetekben a grafikus kártyák akár 800x is gyorsabbak lehetnek mint egy CPU.

Tehát jogosan merül fel a kérdés, hogyan lehet SQL lekérdezéseket megvalósítani GPU -n, illetve milyen előnyökkel és hátrányokkal járhat ez a megoldás.

<https://bitemycoin.com/cryptocurrency-mining/whats-the-difference-between-cpu-and-gpu-mining/>

## **2. fejezet**

### **Adatbázisok teljesítményének optimalizálása**

## 3. fejezet

# SQL lekérdezések végrehajtása

### 3.1. Execution Plan

A lekérdezés végrehajtási terv elkészítésével információkat nyerhetünk a lekérdezések hatékonyságáról. Ennek segítségével optimalizálhatjuk például egy olyan weboldal működését, mely sok és/vagy bonyolult lekérdezést végez adatbázisból. A hatékonyságot általában indexek hozzáadásával vagy elvételével illetve a táblák kapcsolási sorrendjének módosításával vagy kapcsolási módjának megváltoztatásával segíthetjük elő.

#### 3.1.1. Az EXPLAIN parancs

MySQL adatbázis kezelő használatakor az Executing Plan -hez szükséges információkat az EXPLAIN parancs segítségével kaphatjuk meg. Az utasítás megfelelő kifejezéssel együtt használva (SELECT, DELETE, INSERT, REPLACE, UPDATE) információkat jelenít meg az optimalizáló által előállított feldolgozási tervről. Láthatóvá válik például a táblák összekapcsolási sorrendje és a használt index neve.

További információk érhetők el a SHOW WARNINGS használatával. A FORMAT parancs használható kimeneti formátum választásra. TRADITIONAL az alapértelmezett táblázatos alak illetve JSON ami elnevezésének megfelelő formátumban jelenti meg a kimenetet.

A parancs használata rávilágít arra, hogy hol van szükség indexek használatára a lekérdezés gyorsításához, illetve ellenőrizhetjük, hogy a táblák megfelelő sorrendben vannak -e összekapcsolva. JOIN helyett STRAIGHT\_JOIN használatával tippek adhatók az optimalizálónak a táblák kapcsolási sorrendjéről. De a STRAIGHT\_JOIN letiltja a semijoin transzformációkat, így indexelés ebben az esetben nem használható.

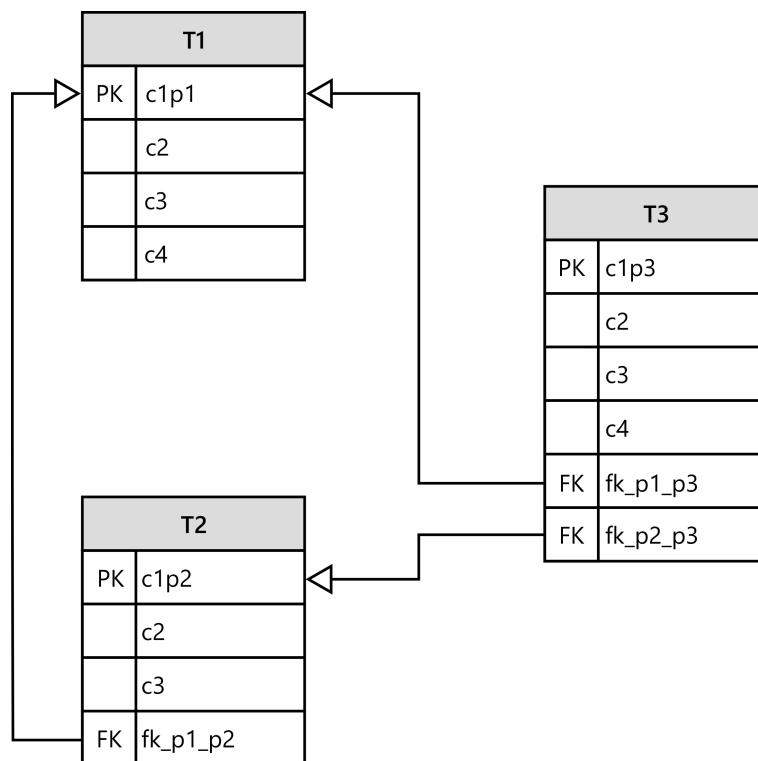
ANALYZE TABLE utasítással frissíthetők a statisztikák, mint a kulcsok számossága. Ez befolyásolhatja az optimalizáló döntéseit.

### 3.2. Teszt adatbázis megtervezése

Az execution plan optimalizálási módnál a kapott információk felhasználásával tudjuk optimalizálni a lekérdezéseket. Olyan adatbázist kell választanunk ennek kipróbáláshoz amelyben használhatók indexek de nincs feltétlen szükség rájuk, össze kapcsolhatunk több táblát és rendezhetjük az adatokat például növekvő sorrendben. A felhasználási mód miatt kevés dologra kell ügyelnünk a tábláknál, ilyen például a NotNull. A táblák minden eleme INT típusú, és csak a feltöltött adatok értékének intervalluma tér el. Az elsődleges kulcs minden esetben 0 -tól N-1 ig tart.

3.1. táblázat. EXPLAIN kimeneti oszlopai

oszlop	JSON elnevezés	Jelentése
id	select_id	A SELECT azonosító
select_type	-	A SELECT típusa
table	table_name	a kimeneti sor táblázatának neve
partitions	partitions	Az egyező partíciók
type	access_type	JOIN típusa
possible_keys	possible_keys	választható indexek
key	key	a választott index
key_len	key_length	a kiválasztott kulcs hossza
ref	ref	oszlopok az indexhez képest
rows	rows	a vizsgálatnó sorok becslése
filtered	filtered	a szűrt sorok százaléka a tábla állapota szerint



3.1. ábra. Adatbázis séma

T1 -es tábla:

- 100 elemet tartalmaz.
- c2 random érték [1:2] intervallumon.
- c3 [1:100]

- c4 [1:10000]

T2 -es tábla:

- 1000 elemet tartalmaz.
- c2 [1:100]
- c3 [1:10000]

Létrehozás során ügyeltem arra, hogy minden T1.c1p1 kulcsnak tartozzon legalább egy idegen kulcs ebben a táblában.

T3 -as tábla:

- 50.000 elemet tartalmaz
- c2 [1:100]
- c3 [1:10000]
- c4 [1:10]

Az idegen kulcsok véletlenszerűen kapcsolódnak a T1 -es és T2 -es táblához.

### 3.2.1. A táblák feltöltése

Az adatokat egy C++ kóddal generáltam .csv formátumba, ügyelve a kapcsolásokra. Majd ezeket a táblákat MySQLWorkbench segítségével importáltam. .csv állományok mintája:

```
c1p1, c2, c3, c4
0, 1, 25, 9707
1, 1, 10, 6539
```

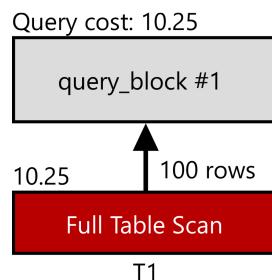
### 3.3. EXPLAIN a gyakorlatban

A MySQLWorkbench grafikus képeit fogom használni melyeket újraalkottam minőségi miatt. Ez a módszer sokkal jobban szemlélteti miképpen dolgozza fel az utasításokat az adatbázis kezelő motorja.

#### 3.3.1. Egyszerű lekérdezés

Vizsgáljuk a következő lekérdezést:

```
SELECT * from thesis.T1 WHERE c3=25;
```

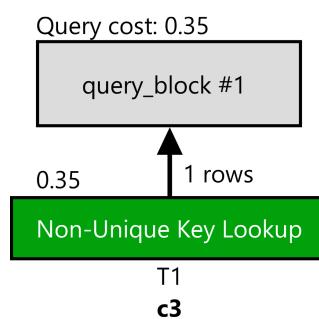


3.2. ábra. Indexelés nélküli lekérdezés

A program által előállított végrehajtási terven látható, hogy a T1 -es tábla összes sorát át kell vizsgálni a megfelelő értékek megtalálásához. Erre nyújthat megoldást a vizsgált oszlop indexelése, melynek segítségével optimális sebességgel lehet meghatározni az értékek lehetséges helyeit.

Indexek hozzáadása a c3 -as oszlophoz:

```
ALTER TABLE thesis.T1 ADD INDEX (c3);
```



3.3. ábra. Indexelés nélküli lekérdezés

Újra lefuttatva az előző lekérdezést láthatjuk, hogy a költsége a töredékére csökkent, melynek az az oka, hogy nem kell minden cella értékét megvizsgálni.

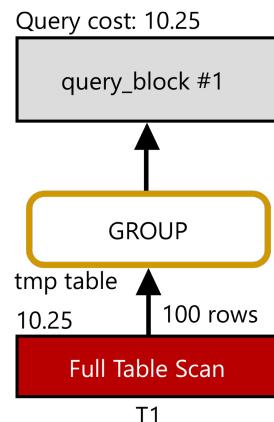
Parancs az indexek eltávolításához:

```
ALTER TABLE thesis.T1 DROP INDEX c3;
```

### 3.3.2. Csoportosítás

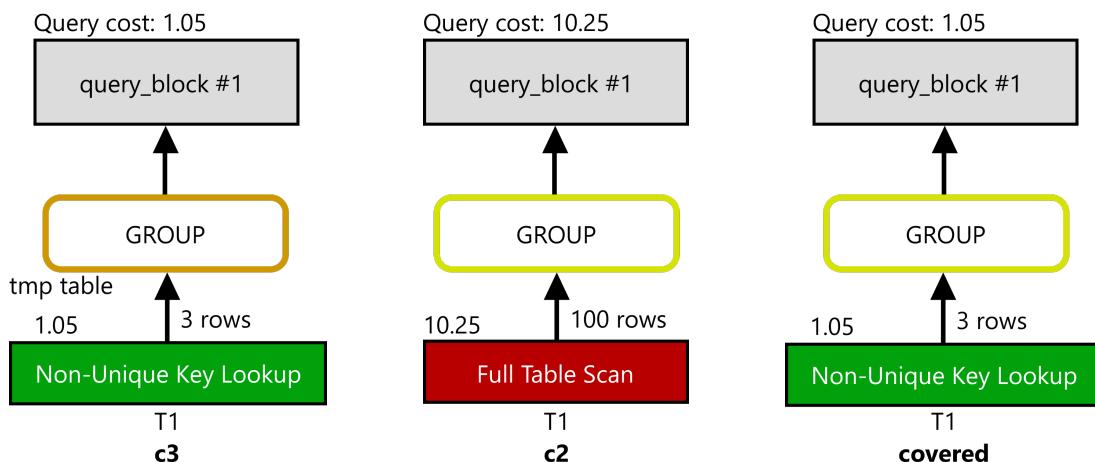
Nézzük meg egy kicsit összetettebb lekérdezést.

```
SELECT MAX(c4), c3, c2 FROM thesis.T1 WHERE T1.c3=18 GROUP BY c2;
```



3.4. ábra. Csoportosítás indexelés nélkül

A lekérdezés elemzésénél két problémát fedezhetünk fel. Az egyik, hogy a teljes tábla átvizsgálásra kerül, a másik pedig egy ideiglenes tábla létrejötte ami a lekérdezés idejére memóriát foglal.



3.5. ábra. Indexelések változatai.

Nézzük a következő négy indexelést:

- A **c3** -as oszlop indexelésével az első példához hasonlóan lecsökkent a szűrés költsége.
- A **c2** -es oszlop indexelésével az ideiglenes tábla nem jön létre lekérdezés közben.
- **c2** és **c3** -as külön indexelése esetén ugyan az történik mint **c3** indexelésekor.
- **c2** és **c3** közös indexelésével egyesíthetjük és két előnyt és így a lekérdezés optimális sebességű és memória igényű lesz.

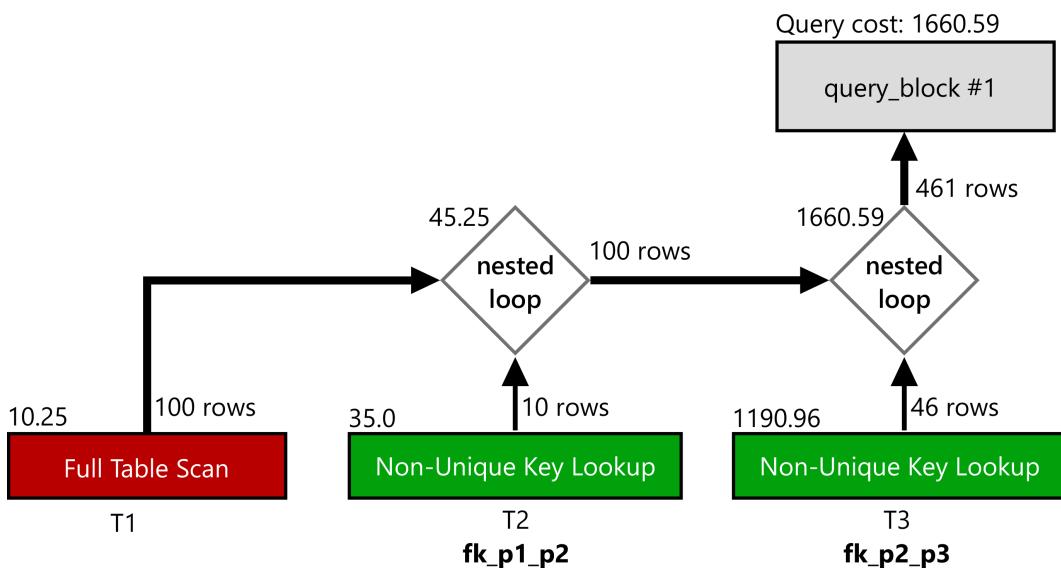
Kevert index létrehozása és törlése:

```
ALTER TABLE T1 ADD INDEX covered (c3,c2);
ALTER TABLE thesis.T1 DROP INDEX covered;
```

### 3.3.3. Kapcsolt táblák

Ez a lekérdezés már összetettebb, annak érdekében, hogy látszódjon a lekérdezés optimalizáló mely esetekben milyen sorrendben kapcsolja össze a táblákat.

```
SELECT T1.c3, T2.c2, T3.c3 FROM T3 JOIN T2 JOIN T1
    ON (T1.c1p1 = T2.fk_p1_p2 AND T2.c1p2 = T3.fk_p2_p3)
    WHERE T3.c4=5 AND T1.c2 = 1 ;
```

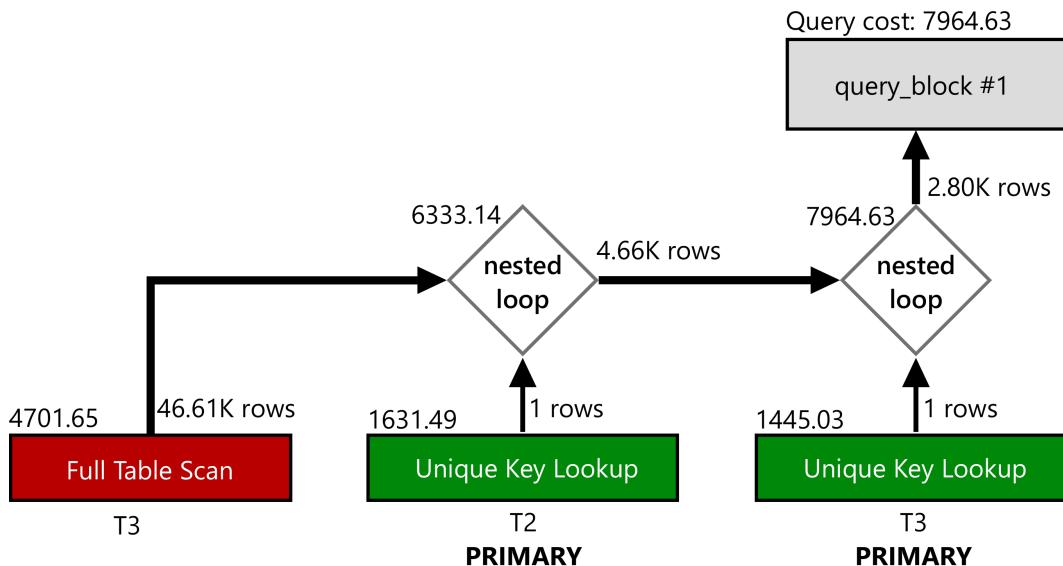


3.6. ábra. JOIN indexelés nélkül

Az első észrevétel, hogy a táblák kapcsolási sorrendje nem egyezik a lekérdezésben szereplő sorrenddel, tehát egyszerű JOIN esetében ezzel nem kell törődni. Láthatjuk, hogy a teljes tábla átvizsgálása itt is problémát jelent.

Miután T3 -as tábla vizsgált oszlopát indexeléssel láttam el, szinte ugyan ezt az eredményt kaptam, leszámítva két értéket. T3 -nál a költség 1190.96 -ról 1198.86 -ra nőtt illetve a 2. nested loop -nál a sorok száma 450 re csökkent.

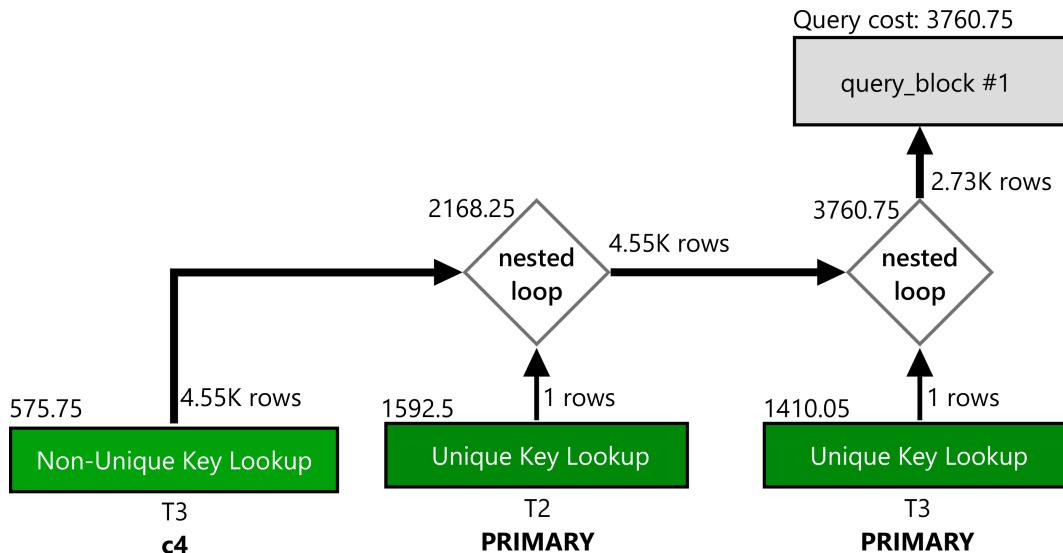
A következő vizsgálatban csak T1.c2 -es oszlopa indexelt.



3.7. ábra. JOIN T1.c2 és T3.c4 indexelésével

Az eredményből az látszik, hogy megváltozott a táblák kapcsolási sorrendje. A terv szerint a motor első lépésben szűri a T3 -as táblát és csak utána kapcsolja hozzá a többet. A másik dolog amit meg lehet figyelni, hogy nem az idegen kulcs szerepel a használt indexnél, hanem az elsődleges, ez a sorrend változás következménye.

A negyedik vizsgálatban minden a két tábla használt oszlopát indexekkel láttam el. Ezen az ábrán azt vehetjük észre, hogy a T3 -as táblánál is használatban vannak az



3.8. ábra. JOIN T1.c2 indexelésével

indexek, így már sebesség optimális a lekérdezés.

Az ábrákon látható query cost megtévesztő lehet, hisz arra enged következtetni, hogy nem érdemes indexelést használni ebben az esetben, viszont a lekérdezés sebességenek vizsgálata egyértelműen mutatja, hogy az indexek használata jelentős sebesség növekedéssel járt.

## 4. fejezet

# Saját kód elhelyezése

### 4.1. A MySQL adatbázis fordítása és telepítése

A MySQL forráskódból való fordítását egy frissen telepített és naprakészre frissített Manjaro 20.2.1 es linux disztribúción végeztem el. Telepítés során non-free grafikus driver választottam. A forrásállomány bárki számára elérhető, és a következő parancssal klónozható:

```
git clone https://github.com/mysql/mysql-server.git
```

A fordítás sikereségéhez rendszerenként eltérő csomagok telepítésére lehet szükség. Jelen esetben a következők telepítéseket igényelte a folyamat.

```
$ sudo snap install cmake --classic  
$ sudo pacman -S rpcsvc-proto  
$ sudo pacman -S pkgconfig  
$ sudo pacman -S make  
$ sudo pacman -S gcc  
$ sudo pacman -S bison
```

A következő lépés a CMake futtatása. Ehhez célszerű létrehozni egy mappát aholávaz új állományok létrejöhének. Ezt nem kötelező megtenni, a forrás könyvtárban is ki lehet adni a parancsot, ehhez a CMake megfogja adni a szükséges kapcsolót illetve figyelmeztet arra, hogy nem ajánlatos.

```
mkdir build
```

Amint ez kész a forrás mappájában futtathatjuk a következő parancsot:

```
cmake ./ ../build -DDOWNLOAD_BOOST=1 -DWITH_BOOST=../boost/
```

A megadott kapcsolóval a boost könyvtár letöltődik a megadott helyre és a CMake megjegyez ezt. Ha a generátor hibába ütközik a szükséges csomagokat telepíteni kell és a *CMakeCache.txt* fájlt törlölni. Ezek után a CMake újra futtatható, de már a Boost -hoz tartozó kapcsolók nélkül.

A sikeres futás jelzi, hogy minden készen áll a fordításra, nincs hiányzó csomag. A `make` parancs futtatása következik. Ez egy hosszú folyamat, jelen konfigurációján 120 percet vett igénybe.

A sikeres fordítást követően létre kell hoznunk egy data mappát a szerver számára.

```
mkdir data
```

Első futtatást rendszergazdaként a megadott kapcsolóval kell végrehajtani, csak ekkor tudja a szerver létrehozni a fájljait!

```
sudo ./bin/mysqld --initialize
```

Ekkor kapunk egy jelszót az a szerverre való csatlakozáshoz. Következő futtatás előtt szükség lehet a data mappa jogosultságainak állítására, különben a szerver a normál indítás során nem tudja inicializálni az InnoDB-t olvasási hiba miatt.

```
sudo chmod -R 777 ./data
```

Ezután futthatjuk a szervert `./bin/mysqld`. Vagy telepíthetjük is azt a `make install` parancssal, de ez jelen esetben szükségtelen.

Csatlakozás a szerverre:

```
./bin/mysql -u root -p
```

A generált jelszó bemásolásával belépünk. Majd megváltoztatjuk a jelszót.

```
ALTER USER 'root'@'localhost' IDENTIFIED BY 'uj_jelszo';
```

Vagy használhatjuk a MySQL Workbench-et ami első belépéskor megkér minket a módosításra.

## 4.2. A MySQL Connector

A Connector, MySQL alapszabványú drivereket biztosít különböző nyelveken, amelyek lehetővé teszik a fejlesztők számára, hogy adatbázis alkalmazásokat írjanak a támogatott nyelveken. Ezen kívül egy natív C könyvtár teszi lehetővé azt, hogy a MySQL -t közvetlenül beágyazhassák alkalmazásaikba.

A következő MySQL által fejlesztett driverek érhetők el:

ADO.NET, ODBC, JDBC, Node.js, Python, C++, C és C API a klienshez.

Az alábbiakat pedig a MySQL közössége fejleszti:

PHP, Perl, Ruby, C+ Wrapper.

\*<https://www.mysql.com/products/connector/>

### 4.2.1. A Connector/C++ használata

Az állományok letölthetők a készítő hivatalos weboldaláról:

<https://dev.mysql.com/downloads/connector/cpp/>

- Linux - Generic
- All
- Linux - Generic (glibc 2.12) (x86, 64-bit), Compressed TAR Archive

Letöltés után kicsomagoljuk. A tartalma egy `include` és egy `lib64` mappa. Az `include` mappa tartalmát a `usr/include` könyvtárba másoljuk, a `lib64`-ét pedig az `usr/lib64` mappába.

Ezután szükség lesz egy külön felhasználóra amellyel a program csatlakozni fog a szerverünkre. Ezt létrehozhatjuk a Workbench alkalmazásban is.

```
CREATE USER 'newuser'@'localhost' IDENTIFIED BY 'password';
```

## 4.2.2. Első program

Hozunk létre egy `connectortest.cpp` nevű fájlt. Szükséges osztályok létrehozása:

```
sql::Driver *driver;
sql::Connection *con;
sql::Statement *stmt;
sql::ResultSet *res;
sql::PreparedStatement *pstmt;
```

Csatlakozás a szerverhez és adatbázis kiválasztása:

```
driver = get_driver_instance();
con = driver->connect("tcp://192.168.0.43:3306", "program", "a");
con->setSchema("thesis");
```

A következő két sor maga a lekérdezés. Az első utasítás elkészíti a szövegből a szerver számára is értelmezhető utasítást, a második sor pedig elküldi azt a szervernek és megvárja a választ:

```
pstmt = con->prepareStatement("SELECT * FROM T3");
res = pstmt->executeQuery();
```

Az eredményt a `res` objektumon végig lépkedve tudjuk kiolvasni, a lekért oszlop nevének megadásával és a típusnak megfelelő `get` függvényénél.

```
while (res->next())
    cout << res->getInt("c1p3") << "\t" << res->getInt("c2") << "\t"
    << res->getInt("c3") << "\t" << res->getInt("c4") << "\t"
    << res->getInt("fk_p1_p3") << "\t" << res->getInt("fk_p2_p3")
    << endl;
```

Lekérdezés után fontos törölni az eredményhalmazt és az elkészített utasítást, ugyanis ezek memóriát foglalnak, és több lekérdezés esetén megtelhet akár a teljes memória is.

```
delete res;
delete pstmt;
```

Futtatáshoz és fordításhoz a következő parancsokat használhatjuk:

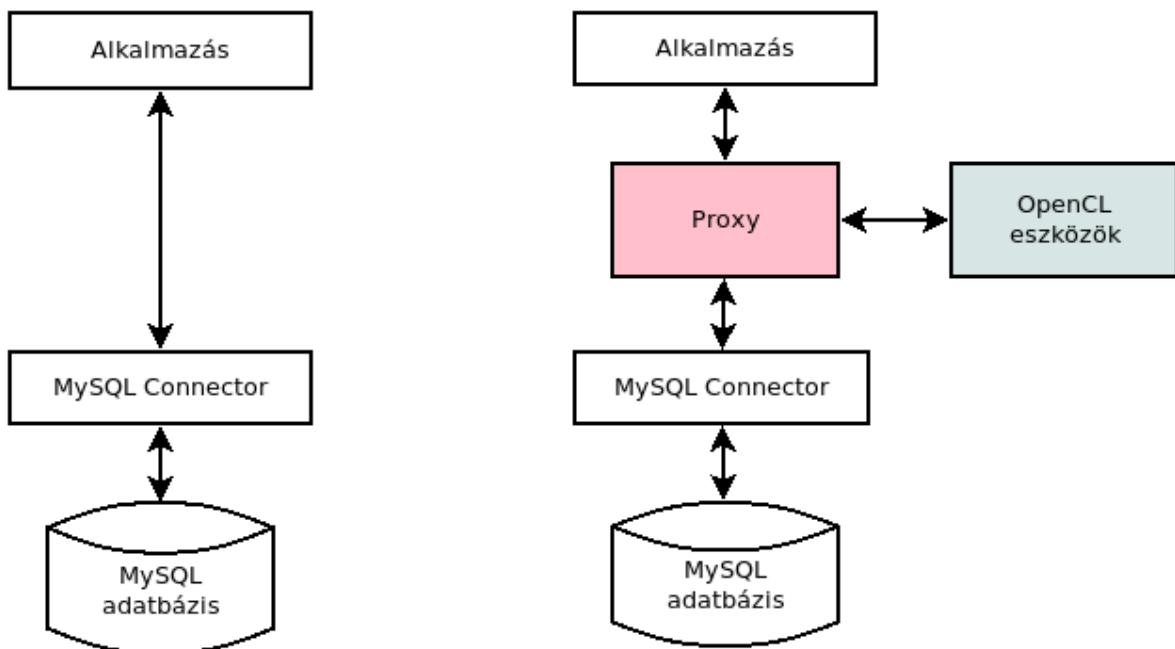
```
g++ -D_GLIBCXX_USE_CXX11_ABI=0 connectortest.cpp -o connectortest.out
-lmysqlcppconn
./connectortest.out
```

## 4.3. Connector proxy létrehozása

Azt feltételezhetjük, hogy a rendelkezésre álló MySQL elérési felületet érdemes meg-tartani valamilyen formában a lekérdezések optimalizálása során. Ez többek között az alábbiakkal indokolható.

- A felület jól átgondolt, a területen járatos szakemberek készítették és a használat szempontjából kiáltja az idő próbáját.
- Elterjedt. Amennyiben az a cél, hogy az elkészített szoftver előnyeit minél többen használni tudják, és akarják is, arra kell törekedni, hogy az új felhasználóknak minél kevesebb plusz munkát jelentsen az elsajátítása.
- Lehetőséget ad, hogy egy saját implementációval transzparens módon megoldható legyen az optimalizálás. Ez olyan szempontból kifejezetten praktikus, hogy így opcionálissá tehető az optimalizálási módszer használata. A szokványos és a dolgozatban bemutatásra kerülő OpenCL-el optimalizált elérés között a váltás kényelmesen megtehető.

A *proxy* nevű tervezési minta kiválóan alkalmas lehet az említett feladat megoldására. A ???. ábrán bal oldal láthatjuk a hagyományos módszert, ahogy az adatokhoz hozzá lehet félni egy alkalmazásból. A jobb oldalt a proxy segítségével optimalizált változat látható.

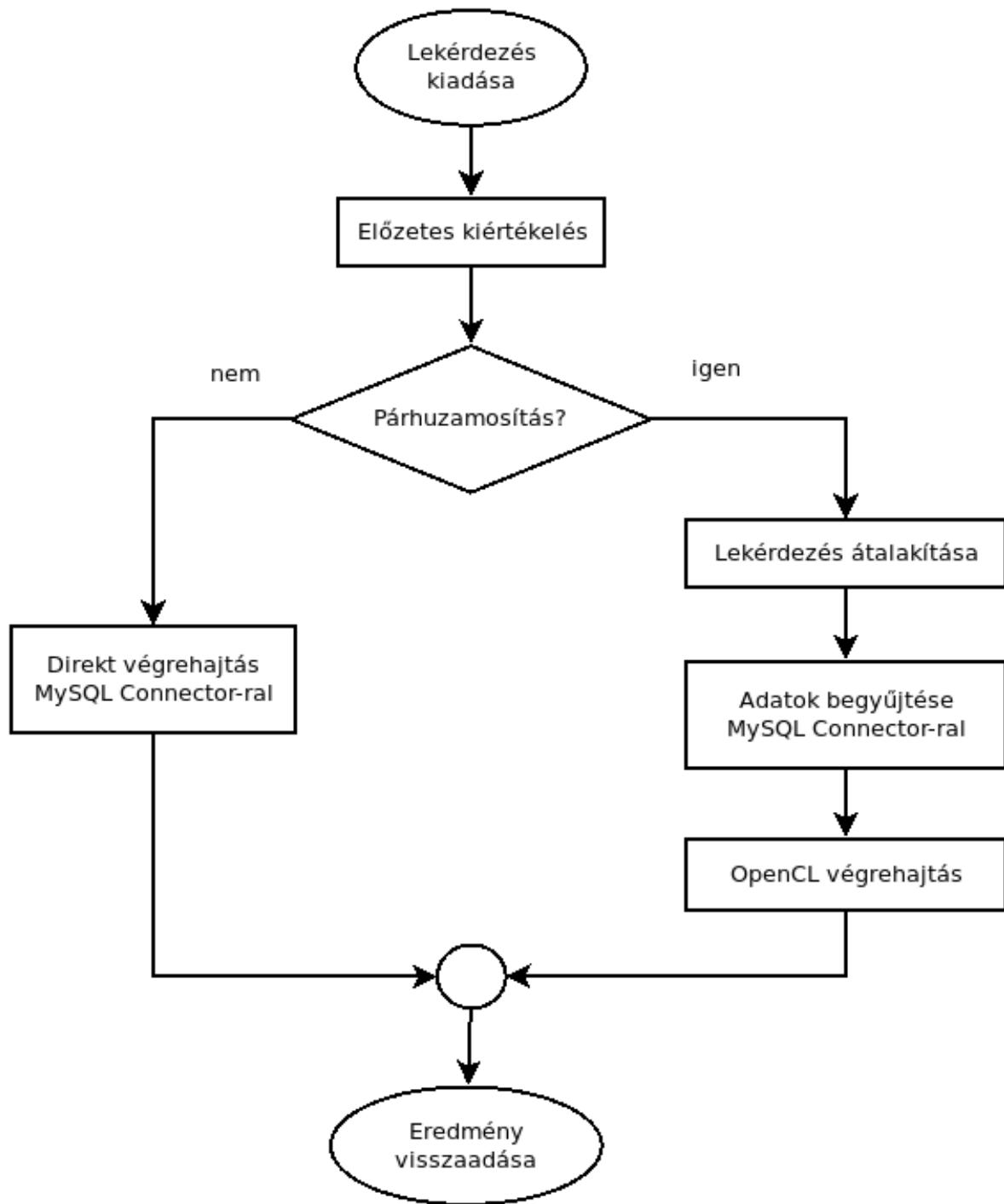


4.1. ábra. A hagyományos és az optimalizáláshoz használt proxy-val kiegészített felépítés

A feldolgozási folyamatot a ???. ábrán látható formában képzelhetjük el.

Ennek az egyik igen fontos eleme az előzetes kiértékelő, amely a kapott lekérdezésről meg tudja állapítani, hogy azt érdemes-e párhuzamosítani.

- Mivel ennek egy gyors döntést kell hoznia, a lekérdezést nem fogja végrehajtani, ezért csak egy becslést ad arra, hogy melyik végrehajtási mód lehet megfelelő.



4.2. ábra. A feldolgozási lépések a proxy beiktatásával

- A kimenete egyszerűen csak egy logikai érték.
- Figyelembe kell vennie a lekérdezés összetettségét, az aktuálisan rendelkezésre álló számítási kapacitásokat, és adatbázisban lévő adatok mennyiségét.

A következő fontos elem a lekérdezés átalakítása, amely a lekérdezést szétválasztja adatgyűjtési és kiértékelési részekre.

- A párhuzamos végrehajtásnál úgy általában azt feltételezzük, hogy szelekciós műveletről van szó.

- Az adatgyűjtést követően már a számítási rész végrehajtásához nem szükséges az adatbázishoz fordulni.

Érdekes lehet egy olyan változatot is felvettetni, amelyben a lekérdezés egyidejűleg elindul a direkt végrehajtás és a párhuzamos végrehajtás irányába is, majd azt az eredményt szolgáltatja, amelyik hamarabb visszatér. Ez elvi szinten ugyan megoldható, viszont feltételezhetően a konkurrens végrehajtás miatt mindenkor minden idő sérülni fog az erőforrásokat miatt. A későbbiekben tehát az előzetes kiértékelés eredményére hagyatkozhatunk.

#### 4.3.1. Az API illesztése

A dolgozat keretein belül nem cél a teljes Connector interfész megvalósítása, csak annak egy részhalmaza. Mivel proxy-ról van szó, ezért az összes olyan művelet átkötése mennyiségi problémát jelent csak, amely nem változtatja meg a végrehajtás módját.

A driver beállítások és az adatbázishoz kapcsolódás változatlan kell legyen, mivel azok szükségesek ahhoz, hogy hozzáférjünk az adatbázishoz.

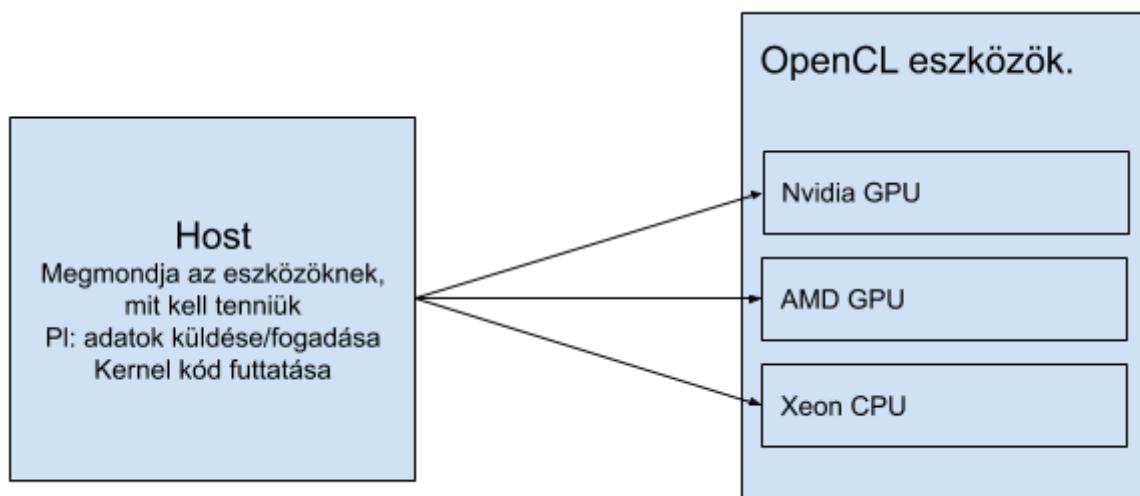
A különbség a `prepareStatement` kapcsán jelenik meg. Ehhez kell tehát egy saját implementációt adni, amelyik kompatibilitási okokból származhat közvetlenül az `sql::PreparedStatement` osztályból. Hogy ha azt szeretnénk, hogy a kódban csak az `include`-okat kelljen megváltoztatni, akkor érdemes a `Driver` és `Connection` osztályokat szintén származtatni. Mivel az interfész megtartható, ezért statikus linkeléssel is lehetőség adódhat a funkció cseréjére, vagyis hogy a fejlesztő (mint célfelhasználó) a saját alkalmazásához nem a `Connector`-t fogja használni, hanem a `proxy`-t (amely aztán használja a `Connector`-t).

## 5. fejezet

### Az OpenCL nyelv

Open Computing Language egy keretrendszer amely lehetőséget ad olyan programok írására amelyek különböző platformokon is futtathatóak. Az OpenCL meghatároz egy programozási nyelvet az eszközök és API-k számára a platformok vezérléséhez és a számítások végrehajtásához az eszközökön. Szabványos interfész biztosít a párhuzamos számításokhoz, melyekhez adatalapú és feladatalapú párhuzamosítást használ.

Fontos észrevenni, hogy az OpenCL natív módon képes beszélni az eszközök nagy részével, de ez nem azt jelenti, hogy a kód optimálisan fog futni. Ugyanis különböző CL eszközök különböző funkciókkal vannak ellátva. Gyártó specifikus kiterjesztések elkerülésével a kód hordozható lesz, de nem sebesség optimális.



5.1. ábra. OpenCL

A következő esetekben a GPU-t érdemes használni:

- Gyors permutáció: Az eszközök gyorsabban mozgatják a memóriát mint a Host.
- Adat átváltás: Egyik formátumról másikra.
- Numerikus gyorsítás: Az eszközök gyorsabban számolnak nagyobb adatdarabokkal mint a Host.

Jelen esetben a Host egy asztali számítógép. Számítási eszközei: CPU, GPU, FPGA, DSP. A számítási egységek: a magok száma. Elemek feldolgozása: ALU magonként.

OpenCL használata mellett szükségtelen gondolkozni azon, hogy pontosan mi is végzi el a számításokat. Ugyanis az OpenCL modelljének illeszkedése egy adott hardverhez, a gyártók feladata.

5.1. táblázat. CPU-k és GPU-k összehasonlítása

Alacsony számítási sűrűség	Magas számítási sűrűség
Komplex logikai vezérlő	Magas számítási és memória-hozzáférés
Nagy caches	Nincs nagyméretű gyorsítótár
Optimalizált a soros műveletekre.	Beépített párhuzamos műveletek. <ul style="list-style-type: none"> <li>• Kevesebb végrehajtási egység (ALU).</li> <li>• Magas órajel sebesség.</li> </ul>
Keskeny adatvezeték <30 szakasz	Széles adatvezeték több száz szakasz
Alacsony késleltetési tolerancia	Magas áteresztőképesség, magas késleltetési tolerancia
Az újabb CPU-k több párhuzamosításra képesek	Újabb GPU-k <ul style="list-style-type: none"> <li>• Jobb áramlási vezérlő logika</li> <li>• Scatter/Gather Memory Access</li> <li>• Már nem egyirányúak a pipeline-ok</li> </ul>

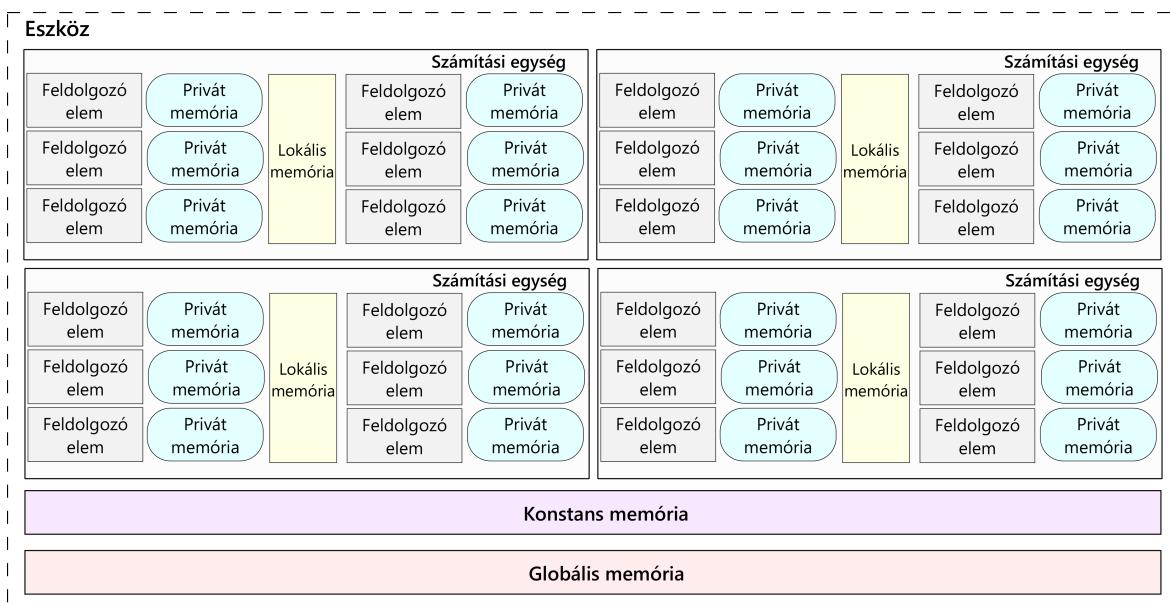
## 5.1. Az OpenCL elemei

### 5.1.1. Eszköz modell

Négy féle memóriát különböztetünk meg:

- Globális memória: minden eszközzel meg van osztva, de lassú. A kernel hívások között perzisztens.
- Konstans memória: Gyorsabb a globálisnál, szűrő paraméterek megadására használják.
- Lokális memória: minden számási egység számára privát, de megosztott a feldolgozó elemek között.
- Privát memória: Gyorsabb a lokálisnál, minden feldolgozó elemnek van.

A konstans, lokális és privát memóriába nem lehet adatokat menteni úgy, hogy más kernel használhassa majd azt.



5.2. ábra. OpenCL eszköz modell

### 5.1.2. Feldolgozási modell

A Host -on OpenCL alkalmazások futnak amelyek a számítási eszközökhöz küldik a munkát.

- Munkaelem: A számítási eszköz alapvető egysége.
- Kernel: A kód, ami fut a munkaegységen (Alap C függvények)
- Program: Kernelek és egyéb funkciók gyűjteménye
- Kontextus: A környezet a munkaelemek végrehajtásához. (Eszközök, annak memoriái és parancssorai)
- Parancssor: Sor melyet a Host arra használ, hogy a munkát (Kernelek, memória másolatok) az eszközbe küldje.

Ez egy keretrendszer, amely meghatározza, hogy a kernel hogyan hajtsa végre a probléma egyes pontjait. Vagy hogyan bontsa a feladatot munka elemekre.

Amire ehhez szükség van:

- Globális munkaméret. Ez általában egy bemeneti vektor teljes hossza.
- Globális eltolás.
- Munkacsoporthosszúság.

Megfeleltetés:

- Munkaelem - számítási elem: minden munkaelement egy számítási elem hajt végre.
- Munkacsoporthosszúság - számítási egység: minden munkacsoporthosszúság egy számítási egység hajt végre. A munkacsoporthosszúság minden munkaelemek, a számítási egység pedig számítási elemek csoporthosszúság.
- Kernel végrehajtási példány - Számítási eszköz. Munkacsoporthosszúság összessége illetve számítási elemek összessége.

### 5.1.3. Munkacsoporthoz

Ideális esetben végtelen számú feldolgozási elemmel rendelkezik az eszköz, minden ilyen elem az adatok egy részét kezeli anélkül, hogy szükségük lenne kommunikációra. De ez a gyakorlatban nem szokott ilyen egyszerű lenni, ezért a munkát fel kell osztani.

- A teljes munkát fel kell osztani kisebb darabokra.
- minden darab egy munkacsoporthoz tartozik.
- A munkacsoporthoz ütemezéssel kell végrehajtani a számítási egységeken.
- minden csoport rendelkezik egy megosztott memóriával, ami olyan mint a lokális memória a számítási-egységeknél

A folytatáshoz a munkacsoporthoz a munkacsoporthoz ütemezni kell a végrehajtási egységeken, és a munkaelemeket számítási egységen belül végrehajtani. minden munkaelem társulni fog egy számítási egységhoz, ha van elegendő. Ezt a folyamatot az OpenCL végzi, csak meg kell adni a globális munkaméretet és munkacsoporthoz méretet. A kernelt indító függvény a munkacsoporthoz számítási egységeknél kiszámolja. Ha nincs elég számítási egység, akkor a munkacsoporthoz sorban hajtódnak végre a meglévő egységeken.

### 5.1.4. OpenCL program általános felépítése

- Létre kell hozni egy szöveg típusú változót, ami tartalmazza a kernel kódját.
- Definiálni kell az eszközt, kontextust és parancssort.
- Definiálni kell a szükséges memória puffereket.
- Az adatokat be kell másolni a pufferekbe.
- Létre kell hozni a programot a szövegből.
- Fel kell építeni a programot.
- Létre kell hozni a kernelt és be kell állítani a paramétereit.
- Futtatni kell a kernel kódját.
- Ki kell olvasni a memóriaobjektumokban található választ a Host-on.

### 5.1.5. Az OpenCL telepítése

A megfelelő működéshez az első lépést már a rendszer telepítésénél megtettem amikor a non-free vagy az nvidia eredeti drivernének a telepítését választottam. ezen kívül a következő csomagokra volt szükség:

```
$ sudo pacman -S opencl-headers
$ sudo pacman -S opencl-nvidia
$ sudo pacman -S cuda
$ sudo pacman -S ocl-icd
```

# 6. fejezet

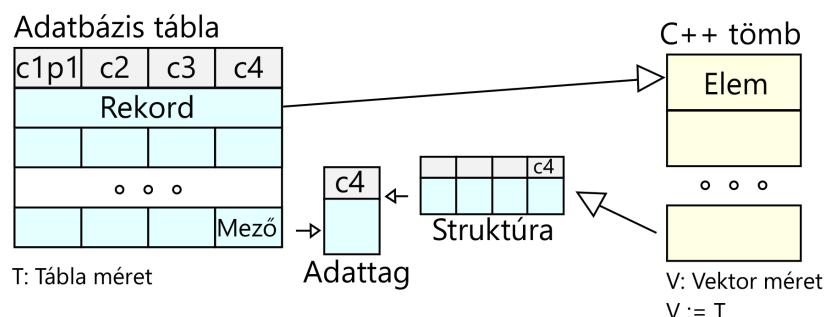
## Lekérdezések optimalizálása

Az alap ötlet egyszerű. A MySQL szerverről lekérjük a tábla tartalmát, amit be másolunk az OpenCL pufferébe, majd a kernelkód futtatása után az eredményt tartalmazó részeket kiolvassuk. De felvetődik pár problémás rész. Hogyan tároljuk a tábla tartalmát? Mekkora legyen a Globális munkaméret és a munkacsoporthméret? Milyen formában állítsuk elő az eredményt, és hogyan olvassuk azt ki?

### 6.1. Adatok előkészítése

#### 6.1.1. Táblák tömbbé alakítása

Ahhoz, hogy OpenCL el kezelni tudjuk az adatokat először szükség lesz egy struktúrára ami reprezentálja az adatbázis tábla felépítését. Az az a struktúra adattagjainak típusa egyezzen meg a táblázat oszlopainak típusával. Ezek után ebből létre kell hoznunk egy tömböt melynek hossza legalább annyi, mint a tábla sorainak száma.



6.1. ábra. Adatbázistábla átalakítása tömbbé

- A struktúra megfelel a tábla felépítésének.
- A tömb egy eleme, egy ilyen struktúra.
- A tömb egy eleme megfelel a tábla egy rekordjának azaz sorának.
- Egy elem adattagjai megfelelnek az adatbázis egyazon rekordjához tartozó mezőknek.

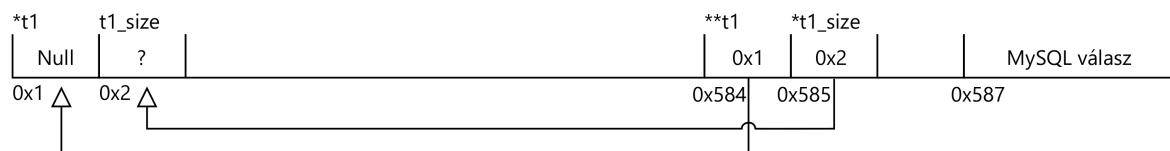
### 6.1.2. A táblákat beolvasó függvény.

A fix sorszámmal rendelkező adatbázis táblák kezelése egy nagyon speciális eset lenne, ezért a programot úgy kell megírni, hogy igazodni tudjon a különböző táblaméretekhez.

Mivel a tábla méret előre nem ismert ezért a tömböt a `malloc` függénnyel lehet lefoglalni. De mivel a lekérdezést külön függvény végzi, és a méretet sem tudjuk előre, ezért a következő megoldást alkalmazom: Első lépéskor létrehozok egy pointert ami késsőbb a táblára fog mutatni, most még NULL értékű, és ehhez egy változót ami a méretet tárolja. Az adatokat lekérő függvénynek átadom ezeket, és miután a `Connector` elvégezte a lekérdezést beállítom a méret változó értékét illetve a tábla memória területét csak ekkor foglalom le, és állítom rá a pointert. Ez úgy lehetséges, hogy a függvénynek egy pointerre mutató pointert adok át, ezáltal kettős indirektség jön létre.

```
Table1Type *t1 = NULL;
int t1_size;
load_database(&t1, &t1_size);

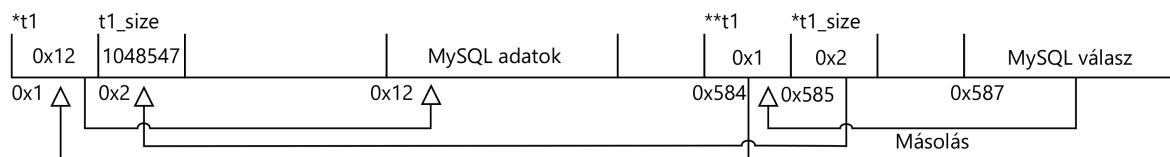
res = pstmt->executeQuery();
```



6.2. ábra. Pointerek a létrehozáskor

```
int i = 0;
*T1_size = res->rowsCount();
*T1 = (Table1Type*) malloc(sizeof(Table1Type) * *T1_size);

while (res->next()) {
    T1[0][i].c1p1 = res->getInt("c1p1"); T1[0][i].c2 = res->getInt("c2");
    T1[0][i].c3 = res->getInt("c3"); T1[0][i].c4 = res->getInt("c4");
    i++;
}
```



6.3. ábra. Pointerek teljes működése

## 6.2. Globális és lokális méret meghatározása

Ennél a pontnál oda kell figyelni pár alapvető dologra:

- A lokális méret legfeljebb 1024 lehet.
- A globális méretnek a lokális többszörösének kell lennie.
- A túl sok vagy túl kevés munkacsoport használata esetén a párhuzamosság sérül.
- A munkacsoportok és csoporton belül a munkaelemek feldolgozása párhuzamos.
- Munkacsoportok között nem lehetséges szinkronizáció, csak munka elemek között, amennyiben azok egy munkacsoportba tartoznak.
- Nem használható kölcsönös kizáráás, a szemafor végtelen ciklushoz vezet.

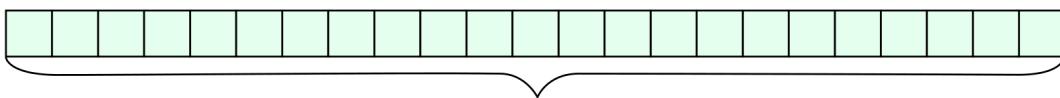
### 6.2.1. Működési elv

Egyszerű esetben a bementi elemszámot megadhatjuk mint globális méret, ez a gyakorlatban meghatározza, hogy hányszor fog lefutni a kernel függvény. A `get_global_id(0)` meghívásával megkapjuk az aktuális azonosítót, ez 0-tól globális méret -1 ig terjed. Innentől tekinthetünk a teljes kódra úgy, mintha egy párhuzamos futó `for` ciklusban lenne.

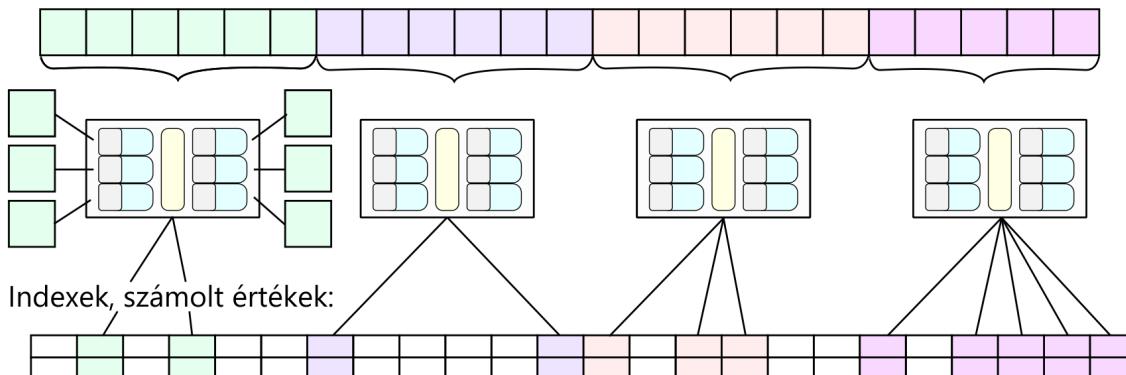
Ezen kívül meg kell adnunk a lokális méretet, ami az elemek száma egy munkacsoporthoz belül. A kernel futása közben ezt is megkaphatjuk a `get_local_id(0)` használatával, csak úgy mint a munkacsoporthoz azonosítót vagy az említett paraméterek méretét.

A végeredményt tartalmazó kimeneti tömb hossza egyezik a globális mérettel, így nem ütközhetünk bele abba a hibába, hogy két számítási elem ugyan oda próbáljon írni. Amennyiben például az 545. elem megfelel, akkor az indexe és egyéb hozzá tartozó értékek az 545. helyen lesznek a kimeneti tömbben.

Tömb elemek:



Munka csoportok:



Indexek, számolt értékek:

6.4. ábra. Munkacsoportokra bontás

### 6.2.2. Globális méret meghatározása

Tegyük fel, hogy kaptunk egy 1048547 sorból álló táblázatot. Ekkor több probléma is előkerül. Egyszerűtlen lenne választani ami ennek a számnak az osztója. Másfelől nem szeretnénk végignézni az ugyan ilyen hosszú kimeneti tömböt az eredmények helyét keresve.

A következő módon oldottam meg a problémát: Állítsuk a globális méretet például 1024-re, ehhez határozzunk meg egy intervallum méretet a sorok száma és globális méret hányadosának felső egész része ként. Ez jelen esetben szintén 1024 lesz. Viszont így át kell adnunk a kernelnek a tábla méretét, ami felső határként ügyel arra, hogy ne lépjük túl a tömböt méretét.

A módosításnak az lesz a következménye, hogy egy munkaelem nem csak a tömb egy elemén fog dolgozni hanem egy intervallumán. Ezzel létrehoztunk olyan részeket a tömbben amiknek a feldolgozása biztosan soros lesz. Ennek köszönhetően már tudunk használni számlálókat, nincs szükség kölcsönös kizárára. A számlálókat tartalmazó tömb mérete megegyezik a globális mérettel.

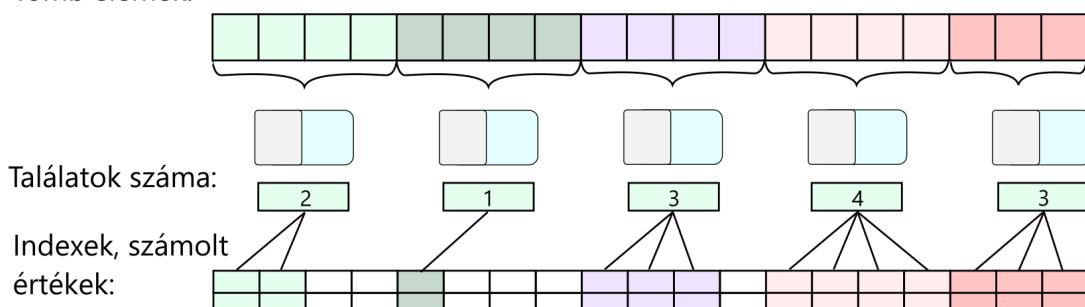
Ezekhez a paraméterekhez még tetszőlegesen meg kell határozni a lokális méretet, aminek csak a teljesítményre van hatása.

$$\begin{aligned} \text{global\_size} &= 1024 \\ \text{interval\_size} &= \left\lfloor \frac{\text{table\_size}}{\text{globalsize\_size}} \right\rfloor + 1 \end{aligned}$$

### 6.2.3. A kernelkód működése

A kernel meghatározza saját globális azonosítóját és ebből illetve a kapott intervallum méretből kiszámítja, honnan kezdőik az ő része a be-, és kimeneti tömbön. Ezután elkezdi az intervallumnyi elem vizsgálatát ügyelve arra, hogy a tábla méretet ne lépje túl. Amennyiben egy elem megfelel a szűrési paramétereknek annak indexét és egyéb lekérdezés függő kiszámolt értékeit a kimeneti tömbbe másol, majd növeli saját számlálóját. Ez a számláló határozza meg, hogy a kimeneti szakaszon a következő elem hányadik helyre kerüljön. Itt már nem munkacsoportonként, hanem feldolgozó

Tömb elemek:



6.5. ábra. Intervallumok feldolgozása

elemenként látjuk a folyamatot.

## 6.3. Eredmény kiolvasása

A kernelkód működéséből látjuk, hogy több módszert is tudunk alkalmazni az elemek kiolvasására. A találatok számát tartalmazó tömböt biztosan ki kell másolnunk, de kimenetre vonatkozóan nézzük meg az alábbi lehetőségeket.

### 6.3.1. Az elemek kiolvasása egyben

Dönthetünk úgy, hogy kimásoljuk a teljes kimenetet:

```
clStatus = clEnqueueReadBuffer(command_queue, TableResult_clmem,
    CL_TRUE, 0, table_size * sizeof(TableResultType), result,
    0, NULL, NULL);
```

Ekkor ennek olvasása a következőképpen zajlik:

```
for(i=0; i< global_size; i++)
{
    offset = i*interval_size;

    for(j=0; j<result_counter[i]; j++)
        cout << t1[ result[ offset + j ].index ].c1p1 << endl;
}
```

Végig kell menni a számlálón aminek hossza `global_size`, majd az `i * interval_size` eltolást használva annyi elemet kell sorban kiolvasni az adott részről, amekkora érték a hozzá tartozó számlálóban található. Ezzel végig lépkedve és ugrálva a kimeneten ki tudjuk olvasni az indexeket illetve kiszámított értékeket.

### 6.3.2. Az elemek kiolvasása szakaszosan

A másik módszer, hogy csak az információkat tartalmazó részeket olvassuk a pufferekből. Ez hasonló módon zajlik, mint előző esetben az eredmények kiírása.

```
int count = 0;
for(int i=0; i< global_size; i++)
{
    if(result_counter[i] < 1 ) continue;
    clStatus = clEnqueueReadBuffer(command_queue, TableResult_clmem, CL_TRUE,
        i*interval_size * sizeof(TableResultType),
        result_counter[i]* sizeof(TableResultType),
        &result[count], 0, NULL, NULL);
    count+=result_counter[i];
}
```

Kiolvasásnál megadjuk az eltolást amit az előző módon számolunk annyi különbözőggel, hogy megadjuk egy elem méretét:

$$i * \text{interval\_size} * \text{sizeof}(\text{TableResultType})$$

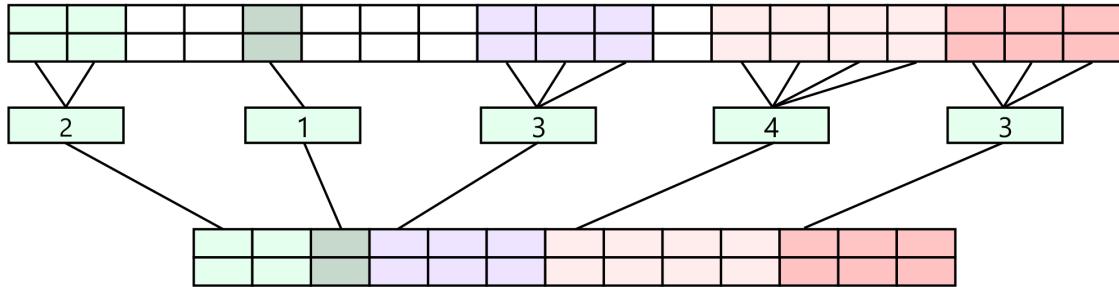
ezután megadjuk, hogy az eltolástól kezdve mekkora memória méretet szeretnénk kiolvasni:

`result_counter[i] * sizeof(TableResultType)`

Ezen kívül a másolás cél memóriacímét minden eltoljuk, az eddigi találatok számával:

`&result[count].`

Ha memóriát is szeretnénk spórolni, megtehetjük azt is, hogy először összeadjuk a találatokat és az alapján foglaljuk le a memóriaterületet a kiolvasáshoz.



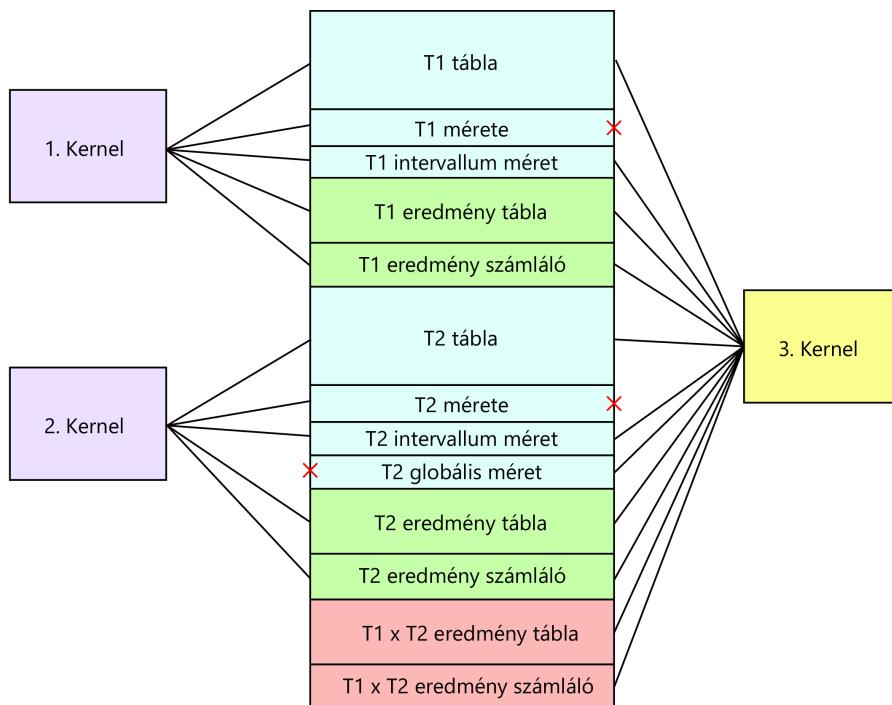
6.6. ábra. Szelektív eredménykiolvasás

## 6.4. Összetett lekérdezés

Az előzőekben taglalt elméleti részek egy táblás lekérdezésekre vonatkozóan próbálták szemléltetni, milyen logika alapján lehet egy SQL lekérdezést GPU -n párhuzamosítva végrehajtani. Most gondoljuk át, miképpen lehetne megoldani kettő vagy több tábla összekapcsolását. Első lépéskor a lekérdezést egy-egy táblára vonatkozó részekre. Ezzel két tábla kapcsolása esetén létrejöhet két különálló kernel amelyek például a szűréseket végeznek a táblákon. Ennél a lépéknél két dolgot is szükséges kiemelni. Egyrészt ezeknek a kerneleknek az előállított eredményét nem kell kiolvasni a pufferekből, ugyanis ez a globális memóriában van, így átadható másik kernel számára is. Másrészt pedig ezek a kernelek ha van kapacitás, akkor futhatnak párhuzamosan is, ezt úgy lehet elérni, hogy külön parancssort hozunk létre számukra.

Miután minden nem összekapcsolást végző kernel befejezte a munkát, adjuk át a szükséges paramétereket a befejező kernelnek. A paraméter lista elég hosszú lesz, ennek az az oka, hogy az eredmények kiolvasása elő észében taglalt módon kell végig menni a szűrt táblákon.

Ennek a kernelnek a globális mérete egyezzen meg a lekérdezésben szereplő legnagyobb tábla globális méretével és ez a tábla szerepeljen a külső ciklusban. Az eredményeket változatlan módokon lehet majd kiolvasni a főprogramba.



6.7. ábra. Kernelek munkaterületei

Az 1. és 2. kernel az előzőekben taglalt módon előállítja saját eredményeit (zöld rész). A 3. kernel megkapja a két táblát, a 2. tábla méretét és az előző két kernel által előállított eredményeket. Az első eredmény kiolvasási módszer alkalmazásával és a kulcsok figyelésével a két táblát összekapcsoljuk és előállítjuk az eredményt. A táblák méretére nincsen szükség, ugyanis az eredmény biztosan nem tartalmaz olyan indexet ami kívül esne az eredeti táblákon. A második táblánál viszont szükség van a globális méretre és az intervallum méretre a kiolvasáshoz.

## 7. fejezet

# Mérések, összehasonlítások

Ebben a fejezetben megvizsgáljuk a program különböző részeit futási idő szempontjából. Első sorban nézzük azokat a részeket amelyek nagyban függnek a táblamérettől vagy az előállítani kívánt kimenettől.

### 7.1. Adatok mozgatásának sebességei

Ebben a részben azt vizsgálom, mennyi idő eljuttatni az adatokat a MySQL szerverről a memóriába, azokat pufferekbe másolni majd onnan az eredményeket vissza ki. Annak érdekében, hogy az eredmények pontosabbak legyenek, minden eredmény 5 futás átlaga.

#### 7.1.1. A MySQL lekérdezés sebessége

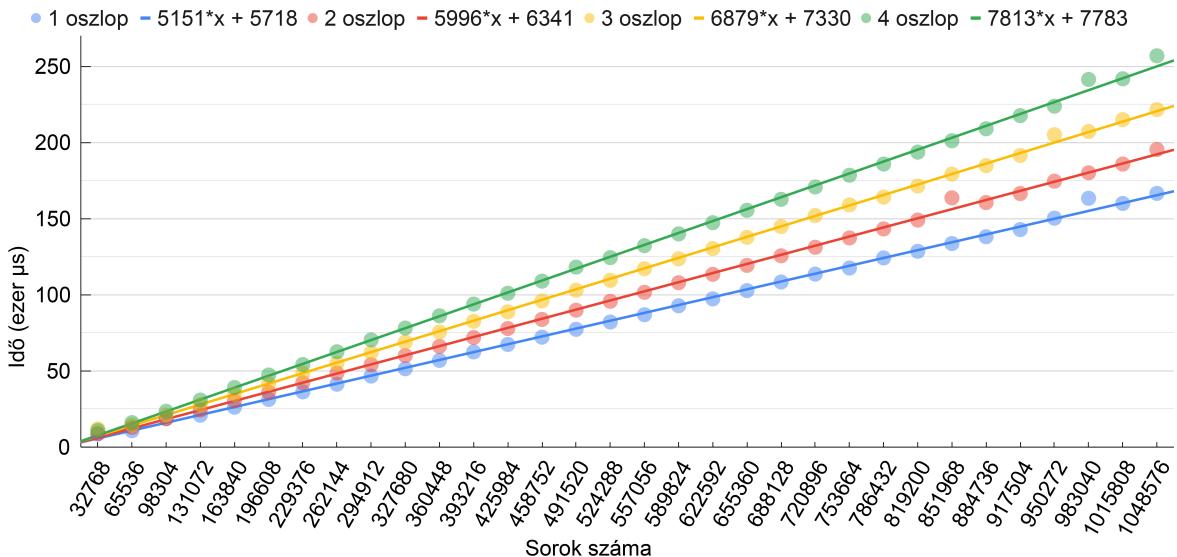
Ezt a sebességet a következő két sor vizsgálatával kaphatjuk meg a leg pontosabban:

```
stmt = con->prepareStatement(command);
res = stmt->executeQuery();
```

A `res` és `stmt` törlése időméréskor nagyon fontos, ugyanis ennek kihagyása miatt a memória megtelhet! `Command` egy `String` az SQL parancs és a következő módon áll össze:

```
range=32768;
while(i <= 1048576{
    command = "SELECT * FROM speedtest_1048576 Limit " + to_string(i);
    ...; i+=range; }
```

- A LIMIT hozzáadása a lekérdezéshez nem növeli annak idejét.
- A tábla tartalmaz elsődleges kulcsot, enélkül a lekérdezések ideje nő.
- A lekérdendő oszlopokat manuálisan módosítottam a programban:
  - `SELECT c1p1 FROM ...`
  - `SELECT c1p1, c2 FROM ...`
  - `SELECT c1p1, c2, c3 FROM ...`
  - `SELECT * FROM ...`



7.1. ábra. Az SQL lekérdezési sebessége

Az ábrán, egyértelműen látható, hogy a sorok számának növekedésével az idő is lineárisan nő. A jelmagyarázatnál látható egyenes egyenletekből becsléseket lehet készíteni adott paraméterekkel rendelkező szűrés nélküli lekérdezéshez.

Tegyük fel, hogy a négy oszlop felett az időnövekedés lineáris, így több oszlopra egyszerű szorzással megkaphatjuk az időket.

4 és 6 oszlop esetén:

$$x = \text{sorok száma}/32768 - 1$$

$$7813 * \text{sorok száma} + 7783 = \text{lekérdezés ideje}$$

$$\text{lekérdezés ideje} * 1.5 = \text{lekérdezés ideje 6 oszlopra}$$

### 7.1.2. A query válaszának átmásolása a saját tömbbe

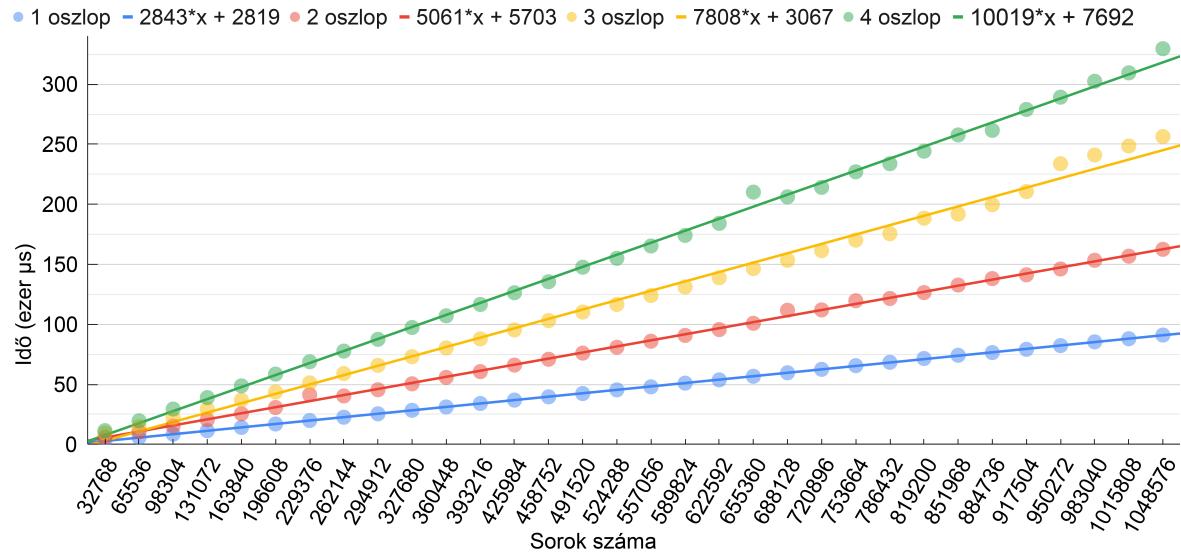
A lekérdezésből kapott választ át kell másolni az általunk létrehozott tömbbe. Ez az egyik legköltségesebb művelet, ezért optimalizálásként célszerű csak a felhasznált oszlopokat lekérni, ez vonatkozik az előző alfejezetre is. A mért program szakasz:

```

int i = 0;
*T1_size = res->rowsCount();
*T1 = (Table1Type*) malloc(sizeof(Table1Type) * *T1_size);

while (res->next())
{
    T1[0][i].c1p1 = res->getInt("c1p1");
    //Többi oszlop ...
    T1[0][i].c4 = res->getInt("c4");
    i++;
}
delete res;
delete pstmt;
    
```

Itt a válaszból lekérdezzük a méretet, majd ez alapján létrehozzuk a saját objektumunkat a memóriában, ezután egyesével belemásoljuk a kapott adatokat soronként. Az mérési módszer megegyezik az előzőekben használtakkal. A grafikonon két dolgot vehe-



7.2. ábra. Adatok másolása saját tömbbe

tünk észre azonnal. Az egyik az, hogy ez a folyamat lassabb mint maga a lekérdezés a másik pedig, hogy drasztikusabb a növekedés az oszlopok számának sokasodásával. Tegyük fel, hogy 4 oszlop felett már nincs szignifikáns eltérés az időnövekedésben. Ekkor használhatjuk az előbb is használt módszert a becslésekhez.

Másolás becslése 4 és 6 oszlop esetén:

$$x = \text{sorok száma}/32768 - 1$$

$$10019 * x + 7692 = \text{másolás ideje}$$

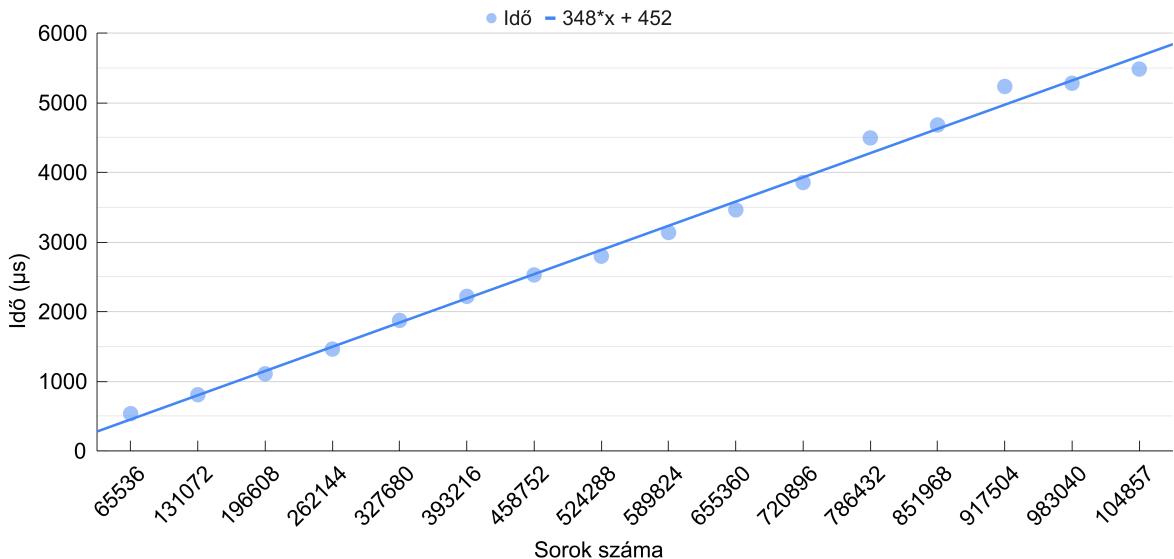
$$\text{másolás ideje} * 1.5 = \text{másolás ideje 6 oszlopra}$$

### 7.1.3. Adatok bemásolása a pufferekbe

Az adatok átmásolása az OpenCL puffereibe szintén egy olyan része a programnak, ami sok időt emészthet fel. Vizsgált rész:

```

int range=65536;
while( range <= 1048576){
    clStatus = clEnqueueWriteBuffer(command_queue, Table1_clmem,
        CL_TRUE, 0, range * sizeof(Table1Type), t1, 0, NULL, NULL);
    clStatus = clEnqueueWriteBuffer(command_queue, Table_size_clmem,
        CL_TRUE, 0, sizeof(int), &t1_size, 0, NULL, NULL);
    clStatus = clEnqueueWriteBuffer(command_queue, Interval_size_clmem,
        CL_TRUE, 0, sizeof(int), &interval_size, 0, NULL, NULL);
    clStatus = clFinish(command_queue);
    range += 65536;
}
    
```



7.3. ábra. Adatok bemásolása 2.

Mivel ebben az esetben a másolás egybefüggő memória területre vonatkozik, nem pedig elemenkénti hivatkozás, ezért elegendő egyetlen mérés. A grafikonon egy 4 oszlopos tábla átmásolásának a sebessége látható. Ebből Több vagy kevesebb oszlopra, illetve sorra a becslés egyszerű szorzásokkal megkapható.

4 és 3 oszlop esetén:

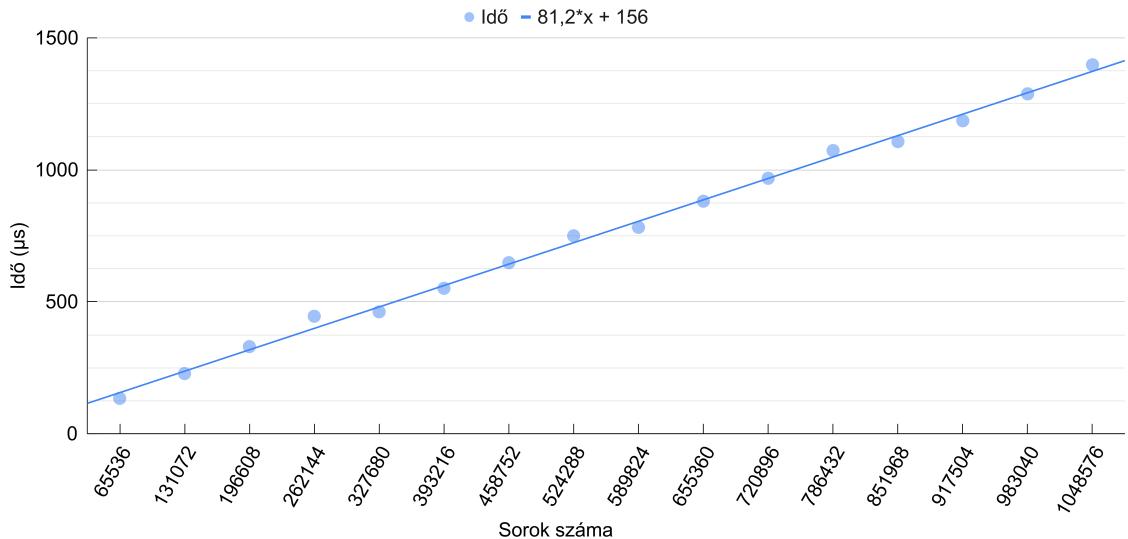
$$x = \text{sorok száma}/65536$$

$$348 * x + 452 = \text{pufferbe másolás ideje}$$

$$\text{pufferbe másolás ideje} * 0.75 = 3 \text{ oszlopos pufferbe másolás ideje}$$

#### 7.1.4. Adatok kimásolása a pufferekből

Nézzük az első kiolvasási módszert, amikor a teljes eredmény tömböt illetve a hozzá tartozó számlálót másoljuk ki. Jelen mérésnél a lista csak egyetlen indexet tartalmaz.



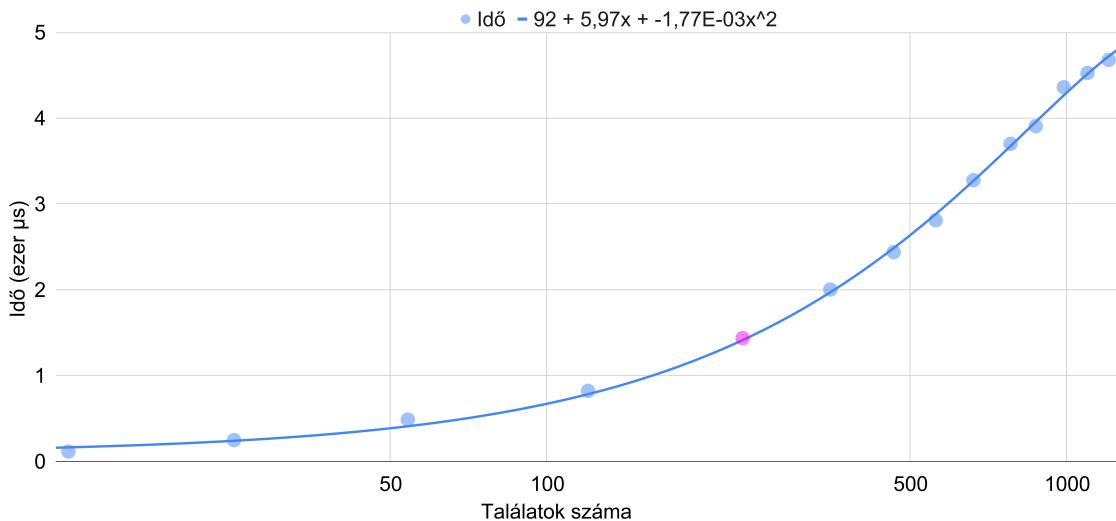
7.4. ábra. Adatok kimásolása 2.

$$x = \text{sorok száma}/65536$$

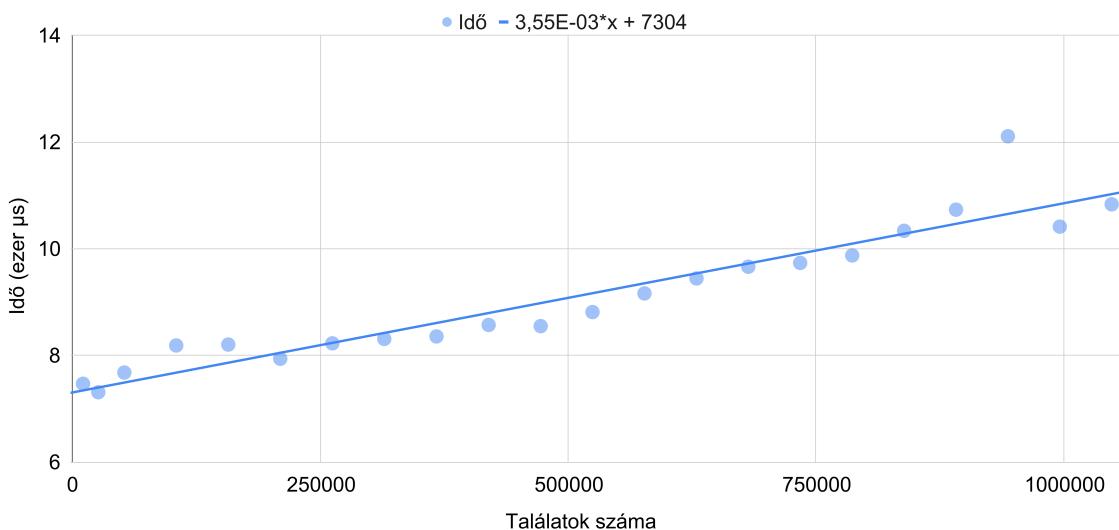
$$81,2 * x + 156 = \text{puffer kiolvasási ideje}$$

Az előzőhöz hasonlóan itt is egy memória darabot másolunk, ennek szorzásával megkapjuk mennyi időbe telne a kiolvasás más méretek esetén. Fontos megjegyezni, hogy a mérések egy, program futtatáson belül történetek. Ez azért olyan lényeges, mert operációs rendszer és egyéb memória kezelési tényezők miatt, az első kimásolás akár 25% al is lassabb lehet mint az azt követők ugyan azon pufferból, lásd későbbiekben!

A másik módszer miszerint nem olvassuk ki azonnal a teljes választ, hanem alkalmazzuk az előző fejezetben taglalt optimalizálási eljárást. Ennek hatékonysága a sok kisebb olvasás miatt függ az eredmények számától. Ennek méréséhez módosítani kell a kernel kódot, ezért manuálisan végeztem a méréseket, vagyis a szűrési feltételeket úgy módosítottam, hogy a találatok száma megfelelően változzon.



7.5. ábra. Adatok kimásolása 2.



7.6. ábra. Adatok kimásolása 2.

Végeredményként látjuk, hogy ez a fajta kiolvasás rendkívül hatékony, de csak abban az esetben, ha a találatok száma minimális. 238 találatnál a módszer elérte azt az időt, ami alatt el lehet végezni a teljes másolást egy darabban.

### 7.1.5. A globális méretből adódó sebességek

A megfelelő globális méret meghatározása futási idő szempontjából nagyon fontos.

Túl nagy globális méret hatásai:

- A lokális méret korlátai miatt túl sok csoport jön létre, így azok feldolgozása soros lesz.
- A kimeneti számláló mérete megnő, ennek kiolvasása több idő, és alacsony találati szám mellett a fölösleges ellenőrzések száma szignifikáns lehet.

Túl alacsony globális méret hatásai:

- A lokális méretet is szükséges lehet csökkenteni. Ha túl alacsony, akkor egyetlen munkaelem fogja végrehajtani, a párhuzamosság teljesen megszűnik.
- A kimeneti számláló mérete alacsony, kevés visszatérő értéknél a kiolvasás gyors.

A méréseket ezen pontok alapján a következők szerint végeztem. A globális méret változzon  $1 - 2^{20}$  értékig, a lokális pedig  $1 - 1024$  ig négyzetes lépcsőkkel, az oszthatóságra figyelve.

A vizsgált kódrészlet:

```
timer.start();

clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
    &global_size, &local_size, 0, NULL, NULL);

clStatus =
    clEnqueueReadBuffer(command_queue, Result_indexes_list_clmem,
    CL_TRUE, 0, global_size* sizeof(int),
    result_counter, 0, NULL, NULL);

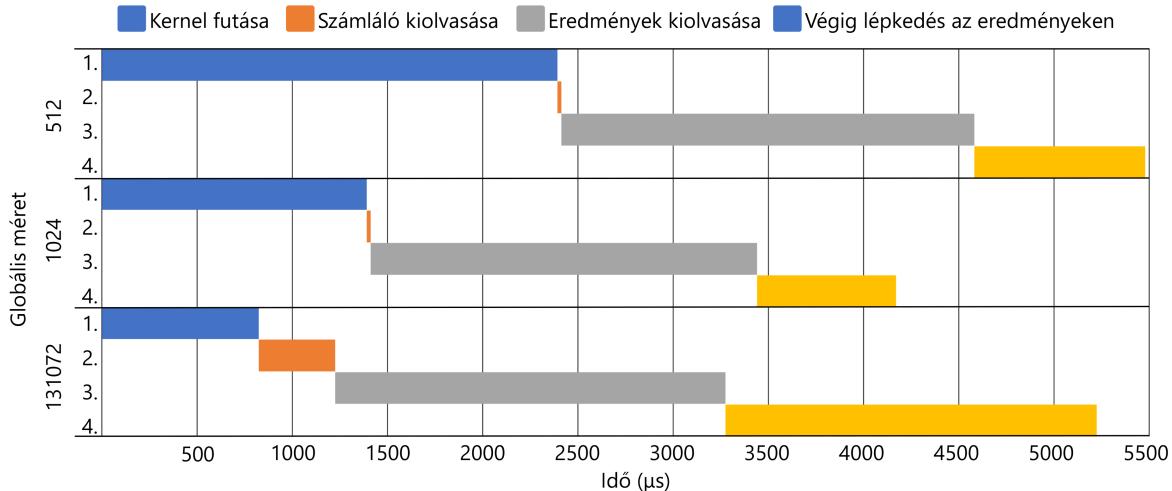
clStatus = clEnqueueReadBuffer(command_queue, TableResult_clmem,
    CL_TRUE, 0, t1_size * sizeof(TableResultType),
    result , 0,NULL, NULL);

clStatus = clFinish(command_queue);
clStatus = clFlush(command_queue);

for(int i=0; i<global_size; i++)
{
    for(int j = 0; j < result_counter[i]; j++ ){}
}
myfile << timer.elapsedMicroseconds() << ",";
```

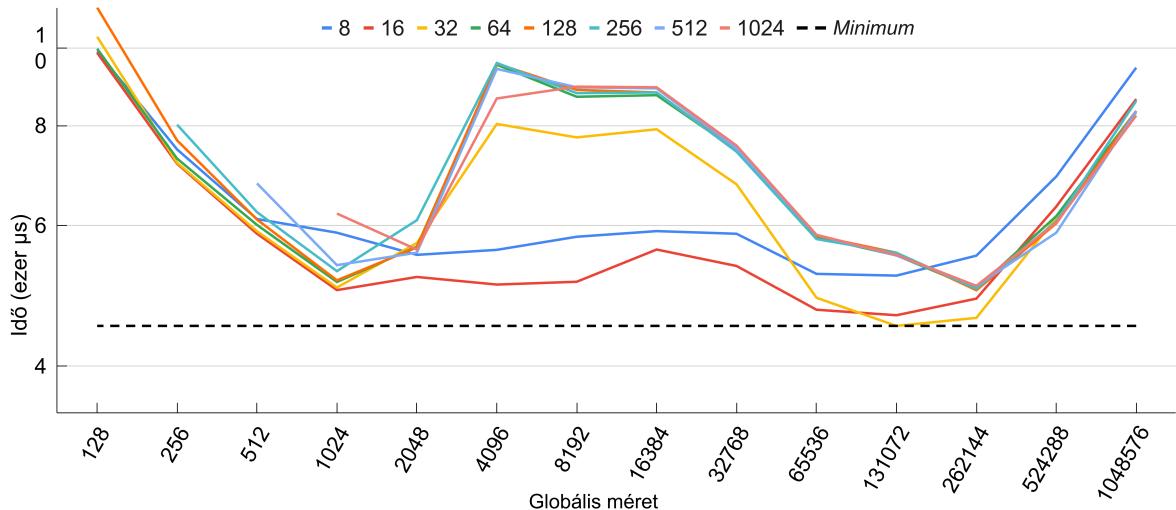
Szembetűnő lehet az egymásba ágyazott két ciklus. Azért szerepel ez kód részlet a mérésekbe, hogy az eredményeken való végiglépegetés idejének nagyságrendje valamilyen módon meghatározható legyen. Fontos megjegyezni, hogy a fordítás során nem kerül figyelmen kívül hagyásra ez a két ciklus.

A következő Gantt-diagram látható részletesen mely program szakasz mennyi időbe telik. minden esetben a Lokális méret 32 és a visszatérő értékek száma körülbelül 50%, az eredményben pedig csak az index szerepel.



7.7. ábra. Programszakaszok ideje.

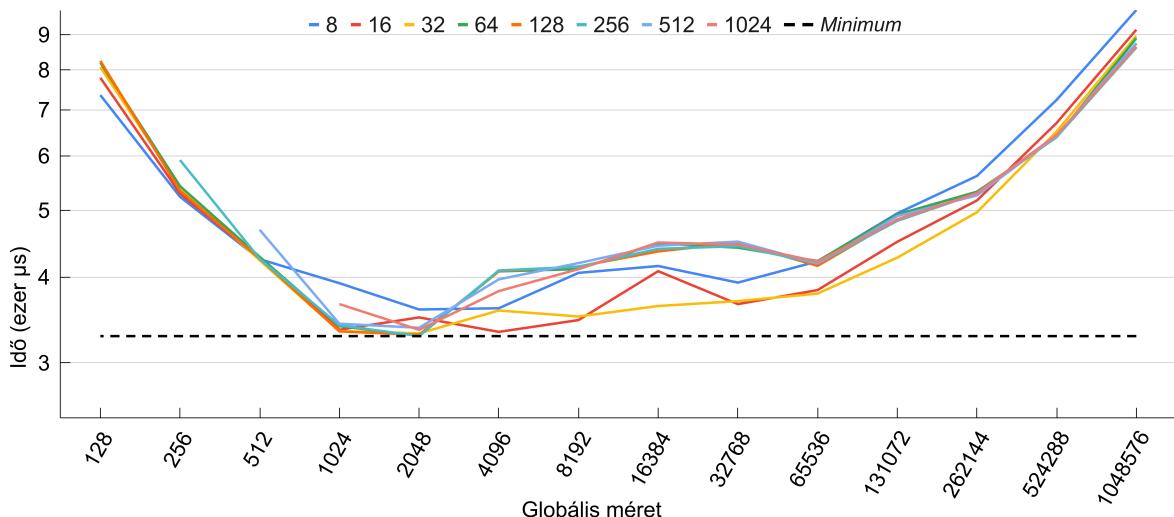
Látható, hogy a túl alacsony globális méret túl magas kernel futási időt eredményez. Túl nagy méret esetében pedig az eredmények kezelésének ideje válik számításigényessé.



7.8. ábra. Maximális találat, az eredmény csak index.

Az eredményt megvizsgálva látható, hogy van egy optimális globális méret tartomány 262144 és 1024 között. Azt is észrevehetjük, hogy bizonyos lokális méretek ebben a tartományban drasztikusan rosszabb eredményt mutatnak. A 16 és 32 -es méretek állnak legközelebb az ideális futási időhöz.

A második grafikonon egy olyan mérés látható, ahol a kernelben található feltételt úgy állítottam be, hogy megközelítőleg csak a tábla méretének 10% -ával térjen vissza a



7.9. ábra. kb. 10% találat, az eredmény csak index.

lekérdezés. Azt láthatjuk, hogy az előző középen lévő hullám lesimul, ez részben annak köszönhető hogy a kernel futási ideje csökken azzal, hogy nem kell annyi indexet az eredmény tömbbe írnia.

Azoknál a lekérdezéseknél, a kernelnek nem csak egy indexel kell visszatérnie hanem több kalkulált értékkel is, ott is hasonló grafikonokat kapunk. De a kernel növekvő munkája és a megnövekedett adatmennyiség kiolvasása miatt a középen lévő hullám nagyobb és nehezebben lapul le.

Ezek alapján azt mondhatjuk, hogy közel ideális választás volt az 1024 mint globális méret.

## 7.2. Gyakorlati példák és következtetések

A hatékonyság megállapításához a `A MySQL lekérdezés sebessége` részben használt módszert fogom alkalmazni. A fejezetben előfordulnak rövidítések: `CONN` - a MySQL Connector programra a `CL` pedig az OpenCL -t használó programra utal. A `k` betű ezres szorzót, az `i` betű pedig indexelés használatot jelöl.

Bontsuk az OpenCL programot négy részre:

- Előkészítés - minden ami a kernel futása előtt történik.
- Kernel(ek) futása
- Eredmények kiolvasása
- Eredmények kezelése

A `Connector` programot pedig 3 részre

- Előkészítés - minden ami a kernel futása előtt történik.
- Lekérdezés - A korábban már említett két sor
- Eredmények kezelése

Az eredmény kezelése résznél azt vizsgáljuk melyik esetben gyorsabb az eredményeket fájlba írni.

### 7.2.1. Egy táblás lekérdezés

```
SELECT * FROM speedtest_1048576 WHERE c3 = 1 AND c4 > 5001;
      5307 row(s) returned
SELECT * FROM speedtest_1048576 WHERE c3 > 1 AND c4 > 5001;
      514025 row(s) returned
```

Először vizsgáljuk meg azt, mennyi ideig tart a lekérdezés, ha a Connecotor -t használjuk.

Sorok száma	5307	514025
Előkészítés	21237 $\mu s$	19125 $\mu s$
Lekérdezés	183674 $\mu s$	273130 $\mu s$
Eredmény kiírás	12591 $\mu s$	1110877 $\mu s$

Ugyanez indexeléssel:

Sorok száma	5307	514025
Előkészítés	21483 $\mu s$	16573 $\mu s$
Lekérdezés	23979 $\mu s$	268184 $\mu s$
Eredmény kiírás	17009 $\mu s$	1087573 $\mu s$

Illetve az OpenCL program használatakor:

Sorok száma	5307	514025
Előkészítés	692705 $\mu s$	682227 $\mu s$
Kernel futás	1129 $\mu s$	1602 $\mu s$
Eredmény kiolvasás	2197 $\mu s$	2013 $\mu s$
Eredmény kiírás	11897 $\mu s$	916984 $\mu s$

Láthatjuk, hogy összességében az OpenCL megvalósítás sokkal lassabb. De vegyük észre, hogy a program legtöbb részének csak egyszer kell lefutnia. Több lekérdezés végezhető úgy, hogy csak a kernel és eredménykiolvasás idejét kell többszörözni. Ezen kívül szembetűnő az is, hogy a tömbben lévő adatokat gyorsabban lehet kezelni.

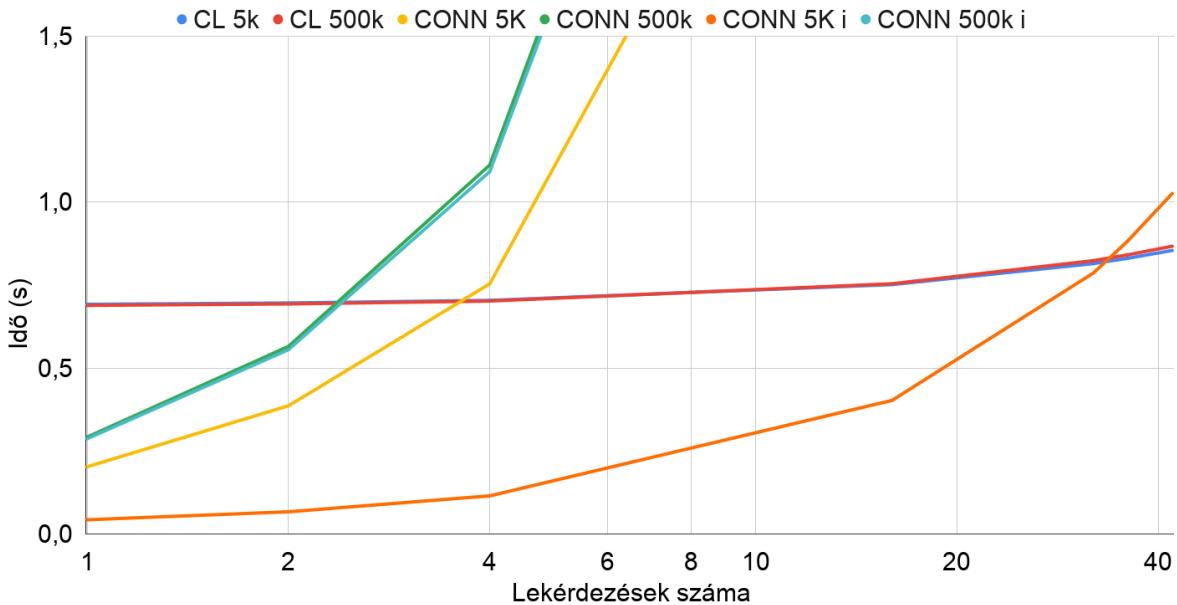
Úgy gondolom meg kell jegyezni azt is, hogy a táblázat utólagos indexelése megközelítőleg 5,5 másodpercent vett igénybe.

A hatékonysságot a következő módon becsülhetjük:

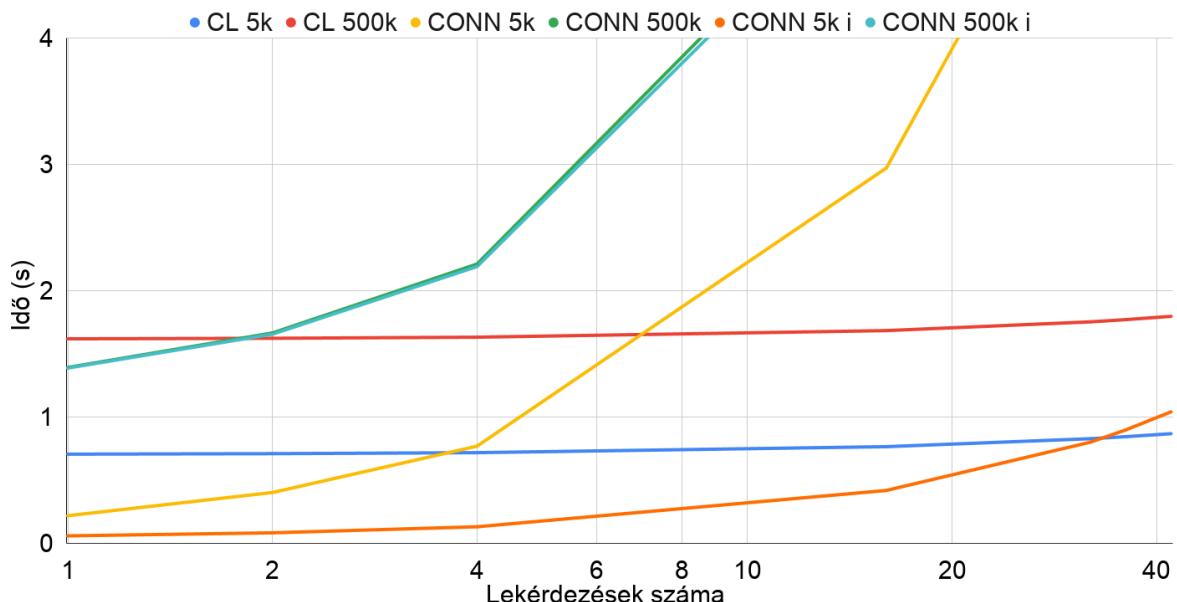
- Az előkészítést egyszer vesszük figyelembe.
- Az előkészítési időhöz hozzá adjuk az ismétlődő részeket. Lekérdezés, kernel(ek) futása, eredmény kiolvasás, eredmény kiírás.

A mérésekből a következő következtetéseket vonhatjuk le:

0.5% os visszatérési arány esetében ha nincs indexelés, akkor 7 lekérdezés után megtérülhetnek az OpenCL plusz költségei. Ha van indexelés abban az esetben ez a szám eléri



7.10. ábra. Becslés csak a lekérdezésre.



7.11. ábra. Becslés az eredmények kiírásával.

a 37 -et.

50% -os visszatérési aránynál már a 2. lekérdezsnél is nyereséges az OpenCL -es megvalósítás.

A grafikonon a CONN 500k és CONN500k i szinte egymást fedi, amiből látjuk, hogy ha a válaszban visszatérő sorok száma magas, akkor az indexelés hatékonysága alacsony.

### 7.2.2. Két táblás lekérdezés

```
SELECT TM.c1p1, TS.c1p1, TM.c3*TS.c3 FROM speed2.speed_262144 AS TM
JOIN speed2.speed_131072 AS TS ON (TM.fk_s = TS.c1p1)
WHERE TM.c3>9600 AND TS.c3>9200;
```

```
839 row(s) returned
WHERE TM.c3>9900 AND TS.c3>9900;
16 row(s) returned
```

Az Connector mérései:

Sorok száma	16	839
Előkészítés	21425 $\mu$ s	21893 $\mu$ s
Lekérdezés	39363 $\mu$ s	85889 $\mu$ s
Eredmény kiírás	90 $\mu$ s	2120 $\mu$ s

Indexeléssel:

Sorok száma	16	839
Előkészítés	20984 $\mu$ s	26650 $\mu$ s
Lekérdezés	10119 $\mu$ s	31663 $\mu$ s
Eredmény kiírás	97 $\mu$ s	2802 $\mu$ s

OpenCL megvalósítás

Sorok száma	16	839
Előkészítés	314073 $\mu$ s	315964 $\mu$ s
Kernel 1. 2. futása	559 $\mu$ s	593 $\mu$ s
Kernel 3. futása	7662 $\mu$ s	103988 $\mu$ s
Eredmény kiolvasás	1974 $\mu$ s	1745 $\mu$ s
Eredmény kiírás	161 $\mu$ s	1932 $\mu$ s

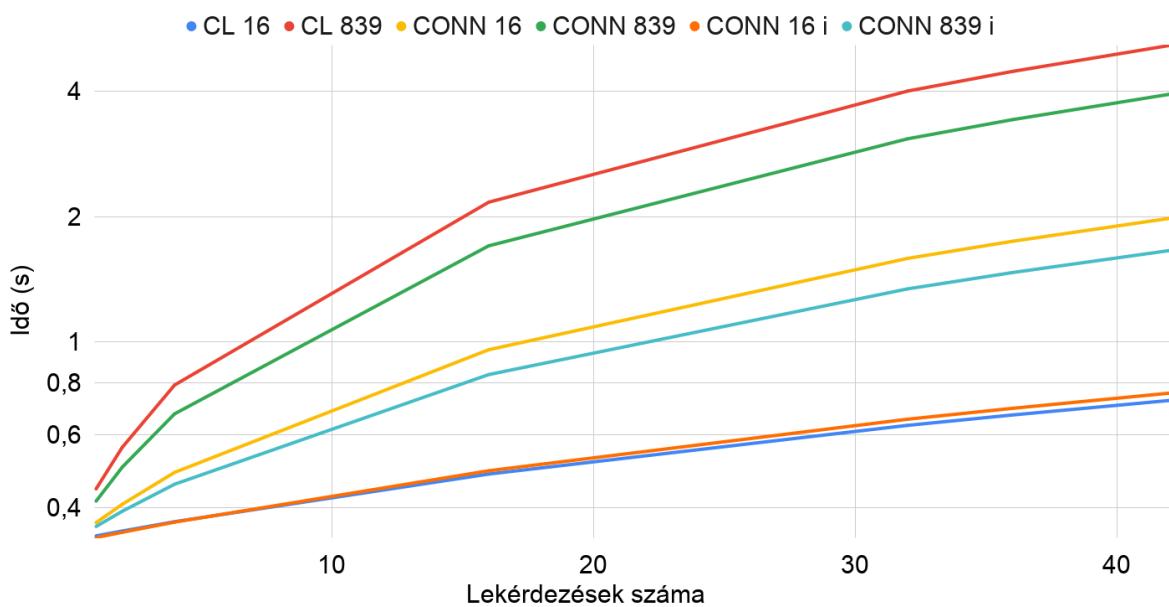
Az OpenCL -es futási időket igyekeztem optimalizálni azzal, hogy a szerverről csak a szükséges oszlopokat kértem le, így elkerülve a fölösleges adatok mozgatását.

A méréseket látva, arra következtetésre juthatunk, hogy az OpenCL program nem bizonyul hatékonynak, még ilyen alacsony találati számnál sem. A kiolvasási időt tovább lehetne optimalizálni az előzőekben említett másik módszerrel, ezzel akár tizedére is csökkenhet a kiolvasási idő, illetve minimálisan javulhat a kiírási idő is. Ez a kis számú lekérdezés annyira speciális esete lenne a lekérdezéseknek, hogy nem vizsgálom tovább.

A táblakapcsolás nélküli lekérdezéshez képesti drámai teljesítmény romlás oka az hogy a MySQL motorja a táblák kapcsolásánál felhasználja a másodlagos kulcsokat mint index így nem kell végignéznie minden sort a párokat keresve.

Valamilyen indexelési módszer használatával az OpenCL megvalósítás is optimalizálható lenne, ám ez bonyolult és költséges lehet.

Az ábrán jól látható, hogy már 839 visszatérő sornál is a CL megvalósítás a leglassabb. A 16 soros lekérdezés pedig alig előzi meg az indexelt lekérdezést. Az eredményekből azt is látjuk, hogy nagyobb mértékben nőtt a futási ideje a CL programnak, azaz még több visszatérő sor eseten a módszer még kevésbé hatékony.



7.12. ábra. Becslés csak a lekérdezésre.

## 8. fejezet

# Összefoglalás

A dolgozatban sok érdekes információ kiderült a mysql lekérdezések és az opencl -es megvalósítások hatékonyságával kapcsolatban.

A MySQL Workbench által előállított grafikus végrehajtási terv hatékonyan nyújt segítséget a lekérdezéseink optimalizálásához. Használatával egyszerű módon találhatjuk adatbázisunk hiányosságait, legyen az a felépítésből vagy akár csak egy indexelés hiányából adódó hátrány. Ez az eszköz bárki számára könnyedén elérhető és sok esetben nem igényel különleges szaktudást sem. Ideális esetben akár egy indexelés hozzáadásával is töredékére csökkenthetjük lekérdezéseink idejét.

Áttérve a lekérdezések OpenCL -es megvalósítására, azt a következtetést vontam le, hogy vannak olyan esetek, amikor sebesség szempontjából jelentős előnyre lehet szertetenni, ha kihasználjuk a grafikus kártyák teljesítményét. De ahoz, hogy ez a gyakorlatban is megvalósítható legyen nagyon sok előzetes vizsgálatra van szükség. A hatékonyság függ:

- Az adatbázis szerver sebességétől.
- A kliensgép erőforrásaitól.
- A lekérdezés bonyolultságától.

Hogy valóban használható legyen a módszer sok dolognak kellene teljesülnie. Például, hogy a lekérdezések idejére az adatbázis tartalma perzisztens legyen, ugyan is a kártya puffereiben lévő adat elavulttá válhat, melynek frissítése erősen rombolja a hatékonyságot.

Egy ilyen kliensalkalmazás megírása rendkívül bonyolult és időigényes ezért nagyon kevés olyan eset lehet, ahol ez a befektetett idő és energia valóban megtérül.

# CD Használati útmutató

Ennek a címe lehet például *A mellékelt CD tartalma* vagy *Adathordozó használati útmutató* is.

Ez jellemzően csak egy fél-egy oldalas leírás. Arra szolgál, hogy ha valaki kézhez kapja a szakdolgozathoz tartozó CD-t, akkor tudja, hogy mi hol van rajta. Jellemzően elég csak felsorolni, hogy milyen jegyzékek vannak, és azokban mi található. Az elkészített programok telepítéséhez, futtatásához tartozó instrukciók kerülhetnek ide.

A CD lemezre mindenképpen rá kell tenni

- a dolgozatot egy `dolgozat.pdf` fájl formájában,
- a LaTeX forráskódját a dolgozatnak,
- az elkészített programot, fontosabb futási eredményeket (például ha kép a kimenet),
- egy útmutatót a CD használatához (ami lehet ez a fejezet külön PDF-be vagy MarkDown fájlként kimentve).