

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

ALGORITMUSOK ÉS ALKALMAZÁSAIK

TANSZÉK

Titkosított Jelszavak Feltörésének Gyorsítása GPU Parallelizációval

Témavezető:

Eichhardt Iván
adjunktus, PhD

Szerző:

Nagy Richárd Antal
programtervező informatikus BSc

Budapest, 2021

EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:

Név: Nagy Richárd Antal

Neptun kód: V7BFDU

Képzési adatak:

Szak: programtervező informatikus, alapképzés (BA/BSc)

Tagozat: Nappali

Belső témavezetővel rendelkezem

Témavezető neve: Eichhardt Iván

munkahelyének neve, tanszéke: Eötvös Loránt Tudományegyetem, Informatikai Kar, Algoritmusok és Aalkalmazásai Tanszék

munkahelyének címe: 1117 Budapest, Pázmány Péter sétány 1/C.

beosztás és iskolai végzettsége: oktató, PhD

A szakdolgozat címe: Titkosított Jelszavak Feltörésének Gyorsítása GPU Parallelizációval

A szakdolgozat témaja:

(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását)

Napjainkban szinte minden program és platform amit használunk egy távoli szerverrel kommunikál, amelyhez a rendszereknek egy nem biztonságos csatornán keresztül kell beazonosítania a felhasználót. Általában valamelyen azonosító és jelszó megadásával.

A megadott jelszót a szervernek valamely formában tárolnia kell, hogy sikeresen el tudja végezni a felhasználó azonosítását. Ezek tárolására egy általános és a kisebb rendszerekben is használt megoldás a jelszó [1] hash-algoritmussal való egy-irányú titkosítása.

Ezeket a hash-eket "visszafelé" nem lehet lefuttatni, ezért kizártan próbálkozással törhető fel. Ezen próbálkozás azonban egy parallelizálható folyamat videókártya segítségével, ugyanis az egyik hash elkészítése nem befolyásolja a többit. A hash folyamat egyik fontos lépése a sózás (salt) ami még egy komplexitási réteget ad a feltörésükhez.

A feltöréshez fontos részt fog képzeni egy olyan [2] jelszó táblázat használata, amely a gyakran használt jelszavakat tartalmazza, ezáltal jelentősen leszűkítve a lehetséges próbálkozások számát.

A szakdolgozat célja a releváns technikák bemutatása és egy olyan program elkészítése, amely egy megkapott hash kódot megpróbál feltörni videókártya segítségével, illetve ezen projekt optimalizálása a lehető legmagasabb sebesség elérése érdekében. A projekthez az [3] ELTE IK GPGPU tantárgy anyagát fogom alapul venni.

[1] Hatzivasilis, George. "Password-hashing status." Cryptography 1.2 (2017): 10.

[2] <https://haveibeenpwned.com/Passwords> Jelszó Lista (2020.11.25)

[3] ELTE IK Algoritmusok és Alkalmazásai Tanszék, Computer Graphics GPGPU tantárgy anyaga: <http://cg.elte.hu/index.php/gpgpu/> (2020.09.14)

Tartalomjegyzék

1. Bevezetés	4
1.1. Motiváció	4
1.2. A Kriptográfia Alapjai	5
1.2.1. Történet	5
1.2.2. Modern Kriptográfia	5
1.2.3. Titkosítási Módszerek	6
1.2.4. Konklúzió	6
1.3. A Hash Alapjai	7
1.3.1. Történet	7
1.3.2. Kritériumok	7
1.4. Az SHA-256 algoritmus	8
1.4.1. Egyediség Bizonyítása	9
1.4.2. Alkalmazások	9
1.5. Hash a Biztonságtechnikában	10
1.5.1. A Jelszó Hash	10
1.5.2. A Jelszó Salt	11
1.5.3. Memóriaigény	11
2. Felhasználói dokumentáció	13
2.1. A Szoftver Célja	13
2.2. A Szoftver Használata	13
2.2.1. Rendszerkövetelmények	13
2.2.2. Telepítés	14
2.2.3. Vezérlés	14
2.2.4. Bemenet	16
2.2.5. Kimenet	17

2.2.6. Példák	17
2.2.7. Hiba Esetén	20
3. Fejlesztői dokumentáció	21
3.1. A Program Felépítése	21
3.2. Az OpenCL	23
3.2.1. Alapok	23
3.2.2. Optimalizálás	24
3.3. Az SHA-256 Aritmetikája	25
3.3.1. Definíciók	26
3.3.2. Funkciók és Konstansok	27
3.3.3. Blokkok Számítása	28
3.3.4. Hash Tömörítés	29
3.3.5. Kimenet	30
3.3.6. Salt	32
3.4. Párhuzamosítás	33
3.4.1. Alapok	33
3.4.2. Adatmozgatás	34
3.4.3. Hashelés	36
3.4.4. Feltörés	37
3.5. C++ Optimalizálás	39
3.5.1. Első Szimuláció	40
3.5.2. Double Buffer	43
3.5.3. Preprocesszor Konstansok	43
3.5.4. C Bemenet	44
3.5.5. Fordító Beállítások	47
3.5.6. Szálméret	49
3.6. OpenCL Optimalizálás	49
3.6.1. Lokális Memória	50
3.6.2. Kulcs Konvertálása	51
3.6.3. Kiterjesztés és Tömörítés	55
3.6.4. Összehasonlítás	56

4. Tesztelés	60
4.1. Memóriaszivárgás	60
4.2. Egységesztelés	61
4.3. Nagy Adattömeg	64
5. Konklúzió	66
5.1. Összegzés	66
5.2. Fejlesztési Lehetőségek	66
6. Köszönetnyilvánítás	68
Ábrajegyzék	69
Irodalomjegyzék	70

1. fejezet

Bevezetés

1.1. Motiváció

A szakdolgozatom célja egy program elkészítése, amely egy jelszó hash-et egy lehetséges jelszó lista felhasználásával megpróbál visszafejteni videókártyán többszálú megoldással, illetve ezen program optimalizálása jelentős teljesítmény javulás érdekében az általános processzorhoz viszonyítva. A programnak nem célja egy potenciális biztonsági rés kihasználására eszközök biztosítani, minden össze az általam kedvelt biztonságtechnika és alacsony szintű optimalizálás témakörök vegyítésére ad lehetőséget.

A program elkészítéséhez a C++ programozási nyelvet, illetve a nyílt forráskódú OpenCL környezetet fogom használni, ugyanis kellő mélységű hozzáférést biztosít a grafikus kártya általános célú számítási egységeinek programozásához, illetve a processzorral való kommunikációjához. Emellett elég általános ahhoz, hogy használható legyen több videokártya platformon is.

A program írása során az általam elkészített rendszer potenciális gyorsítási lehetőségeket fogom tesztelni és elemezni, ezeket a saját számítógépem teljesítménye alapján rangsorolni egy véletlen jelszó feltörésének komplexitása alapján.

1.2. A Kriptográfia Alapjai

1.2.1. Történet

A fontos adatok titkosítása történelmünk során mindig jelentős szerepet játszott akár háborúk közben a stratégia terv biztonságos szállításához, vagy találományok pontos specifikációjának a biztonságos tárolásához. Ennek köszönhetően a technológia fejlődésével a kriptográfiának mindig szorosan tartania kellett a lépést, hogy biztosítsa, hogy az adatokat nem csak most, hanem évek múlva sem lehet egyszerűen visszafejteni illetékteleneknek [1].

A titkosítást és a visszafejtést legtöbbször papíron végezték, amely során egy előre meghatározott üzenetet a kriptográfus titkosított valamely eljárással, majd az eredményét leírta egy másik papírra. Ezek után a titkosított üzenet elküldésre került, amelyet optimális esetben kizárolag a célszemély tudott visszafejteni. Ez azonban sok hibalehetőséget rejtett magában, hiszen amennyiben túl egyszerű a titkosítási módszer a feltörés is jelentősen könnyebb, azonban ha túl bonyolult akkor a kriptográfus hibáinak száma és egy üzenettel eltöltött ideje is megnőtt.

1.2.2. Modern Kriptográfia

Az első modern kriptográfus gépet két dán üzletember fejlesztette ki 1915-ben, mely az Enigma nevet kapta [2]. A gép bemeneteként szolgált egy alapbeállítás és az üzenetet, majd ebből visszaadott egy véletlenszerűnek tűnő karaktersorozatot. Megegyező beállításokkal és üzenettel a gép kimentene pontosan ugyan azt az eredményt adta minden alkalommal. Bárki a gép, az alap beállítás és a titkosított üzenet birtokában pontosan vissza tudta fejteni az eredeti üzenetet. Azonban az alapbeállítás hiányában ez manuálisan közel lehetetlennek bizonyult.

A gép segítségével az összetett műveletek automatikusan megtörténtek és ezzel jelentősen csökkentették az emberhibából adódó problémákat, illetve sokszorosára gyorsították a folyamatot egy manuális számoláshoz képest.

Napjainkban ezen műveleteket már számítógépek végzik, amelyek az Enigma gép teljesítményének milliárdszorosát képesek elvégezni. Ennek köszönhetően a titkosítási módszerek teljesítményének is növekednie kellett.

1.2.3. Titkosítási Módszerek

Az általános titkosítási módszer úgynevezett szimmetrikus, vagy privát kulcsos titkosítás, amelyben egy kulcs és egy üzenet ismeretében egy titkosított üzenetet állítunk elő, amelyet a kulcs ismeretében lehet kizárolag visszafejteni. Fontos feltétel hogy a kulcs ismeretében gyorsan visszafejthető legyen, míg a kulcs ismerete nélkül szinte lehetetlen.

Ezzel szemben az asszimmetrikus módszer esetén két kulcsra van szükség. Egyre a titkosításhoz és egyre a feloldáshoz. Amennyiben az üzenetet az első kulccsal titkosítjuk, kizárolag a másodikkal lesz lehetőségünk visszafejteni azt. Ugyan ez működik az ellenkező irányban is. Ennek köszönhetően például az RSA algoritmus használatával egy publikus kulcsot bárhol tárolhatunk az interneten és amennyiben valaki üzenetet szeretne küldeni nekünk, titkosítja az üzenetét a publikus kulcsunkkal, majd elküldi nekünk és kizárolag mi tudjuk feloldani az üzenetet a privát kulccsal.

A mai modern titkosítás az internetes korban az előző két fő módszer vegyítéssével működik. Amikor két számítógép először kommunikálni kezd, akkor valamely asszimmetrikus kulcs eljárással megosztanak egymással egy szimmetrikus kulcsot, melyet alkalmaznak majd a további kommunikációban a gyorsabb sebesség miatt.

1.2.4. Konklúzió

A kriptográfia mindig is komoly jelentőségű ágazat volt a történelmünk során: esetenként akár országok jövője múlhatott azon, hogy a bizalmas információkat biztonságosan tudják szállítani. Az eddigi bevezetőben olyan titkosítási eljárásokról esett szó, amelyben az egyik fél által küldött titkosított adatot a fogadó által a megfelelő információk ismeretében továbbra is visszafejthető. Erre azonban nem minden esetben van szükség, bizonyos helyzetekben pedig hátrányos is lehet. Ennek a megoldására alkalmazunk hash függvényeket.

1.3. A Hash Alapjai

1.3.1. Történet

A hash algoritmusokat egy egyszerűnek tűnő, de valójában összetett probléma megoldására alkották meg 1953-ban [3]. Tegyük fel, hogy adott két nagy méretű dokumentum, amelyek egyezését ellenőriznünk kell, akár úgy, hogy az egyikkel nem rendelkezünk. Mindezt egy olyan módszerrel, amelynek nem szükséges minden egyes karaktert összevetnie, hiszen az egy lassú folyamat és minden két fájlnak folyamatosan elérhetőnek kell lennie. Természetesen a dokumentumok pontos egyezését kizárolag a karakterenkénti vizsgálattal érhetjük el, azonban olyan módszerek kialakítása célszerű lehet, ahol a dokumentumok különbözősége esetén rendkívül gyorsan juthatunk erre a (negatív) megállapításra, illetve fordított esetben a fals pozitív kimenet is statisztikailag elenyésző. Számos tudományág hagyatkozik nem teljesen bizonyos kimenetű, de nagy valószínűségből adódóan konkluzívnak vett eljárásokra.

A DNS teszt esetében például előfordulhat, hogy két embernek pontosan meggyezik a teljes DNS lánca, viszont ennek az esélye annyira elenyésző, hogy nem vesszük figyelembe mint lehetőséget. Analógiaként, az informatikában hasonló problémák megoldására születettek meg a hasítófüggvények, és kapcsolódó eljárások.

1.3.2. Kritériumok

Egy hash algoritmus bemenetnek kap valamilyen adatot. Ez az algoritmustól függően tetszőleges bináris adatfolyam lehet, majd kimenetként visszaad egy fix hosszúságú kulcsot amelynek meg kell felelnie az alábbi alapsabályoknak [4]:

1. **Univerzális**, azaz bármekkora és akármilyen típusú adatfolyamra működik feltéve, hogy reprezentálható bináris formában,
2. **Adatvesztő**, azaz minden bemenetre azonos hosszúságú kimenetet ad, ezáltal az eredeti dokumentum nem visszaállítható,
3. **Determinisztikus**, azaz két azonos bemenet ugyan azt a kimenetet adja,

4. **Egyirányú**, azaz gyorsan kiszámolható a hash minden bemenet alapján, viszont kizárolag a kimenet ismeretében nem állítható vissza belátható időn belül a kiindulási dokumentum,
5. **Egyedi**, azaz elenyésző az esélye, hogy két különböző bemenet azonos kimenetet generál,
6. **Instabil**, azaz egy kisebb módosításnak a bemeneten nagy változást kell hogy eredményezzen a kimeneten.

Egy ilyen algoritmus használatával két dokumentum összehasonlítható úgy, hogy minden kettőn lefuttatjuk ugyan azt az algoritmust és amennyiben a kimenet különbözik, a bemeneti dokumentumok biztosan nem egyeznek. Azonban amennyiben egyeznek a hash-ek, szinte biztosan a dokumentumok is.

1.4. Az SHA-256 algoritmus

Az SHA-256 egy kriptografikus hash algoritmus amely az SHA-2 egyik alváltozata, melyet az Egyesült Államokbeli National Security Agency fejlesztett ki 2001-ben [5]. Az SHA-2 két fő változattal rendelkezik: SHA-256 és SHA-512, amelyek között a fő különbség a használt szavak vagy szegmensek mérete. Míg az SHA-256 32 bites, addig az SHA-512 64 bites szavakat használ. Egyik algoritmus sem törhető fel egyelőre, ennek ellenére manapság a nagyobb cégek jobban preferálják ezen algoritmusok utódait az új rendszerekben. Integritás vizsgálathoz és kereséshez egyiket sem alkalmazzák viszonylag magas számítási igényük miatt, jelszavak titkosítására biztonságos tároláshoz azonban az SHA-256 egy elterjedt módszer.

Az SHA-2 elődjére, az SHA-1-re épül, amely pedig a MerkleDamgård struktúrára. Abból adódóan, hogy ezt a struktúrát már sikerült feltörni, ezáltal az SHA-1 algoritmust is, bizonyítottá vált hogy csak idő kérdése, mielőtt az SHA-2 is hasonló sorsra jut [6]. A megfelelő algoritmus kiválasztásánál vizsgált három fő tulajdonság:

1. **Kritériumok:** Az algoritmusnak meg kell felelnie a 1.3.2 bekezdésben foglalt hash algoritmusokra vonatkozó feltételeknek,

- 2. Relevancia:** Az algoritmusnak jelszótitkosítás tekintetében napjainkban gyakran használnak, vagy az algoritmus korábbi használatából adódóan jelentős mennyiségű feltörhető adattal kell rendelkezni,
 - 3. Gyorsíthatóság:** Az algoritmusnak látványosan gyorsíthatónak kell lennie VGA parallelizáció segítségével

1.4.1. Egyediség Bizonyítása

Az általam választott SHA-256 algoritmus [5] minden bemenetre egy 64 hexadecimális karakteres (256 bites) kimenetet képez, amely tartalmazza az angol ábécé betűit a-tól z-ig, illetve számjegyeket 0-tól 9-ig, így karakterenként 16 különböző lehetőséggel rendelkezik. Kis és nagybetű között nem tesz különbséget, azonban a sorrend számít. Ez

$$(6+10)^{64} > 1.15 \times 10^{77}$$

különböző kulcslehetőséget képez. Ez olyan hatalmas mennyiség, hogy ha a következő 1000 évben a földön jelenleg élő minden ember minden nap elkészítene 100 egyedi kulcsú dokumentumot, akkor az összes lehetséges hash nagyjából

$$(1000 * 7800000000 * 365 * 100) = 2.847 \times 10^{17} \quad (1.1)$$

$$\frac{2.847 \times 10^{17}}{(6+10)^{64}} \approx 2.46 \times 10^{-60} \quad (1.2)$$

-a készülne csak el. Emiatt kijelenthetjük, hogy hash egyezés esetén feltételezhető, hogy a bemeneti dokumentumok is pontosan megegyeznek.

1.4.2. Alkalmazások

A hash bevezetése sok problémára ad gyors és kényelmes megoldást, pl.:

- **Számítógéphálózat** esetén adatátvitel során az átvitt adatok megsérülhetnek, illetve támadók módosíthatják azokat. Ezért az üzenetek mellé társítunk egy hash-et is, amelyet a küldő létrehoz, a fogadó pedig ellenőriz. [7].
 - **Adatbázis**-ból történő lekérdezés során az adatok hash algoritmusok felhasználásával csoportosíthatóak a gyorsabb kezelés érdekében [8].

- **Adattárolás** esetében a merevlemezen a fájlok elérési útja, illetve a neve hash-ként van tárolva a gyorsabb beazonosítás érdekében [9].
- **C++ modellek** esetén a nyelv alapkönyvtárában számos tárolási eszköz használ hash funkciókat az adatok kategorizálására és egyediségének vizsgálatára (pl: std::unordered_set, std::unordered_map) [10].
- **Tranzakciókezelés** esetén egy limitként szolgálhat egy blokk kiszámítása, amelynek meg kell felelnie valamilyen sémának. Bitcoin esetén például 8 darab hexadecimális nulla karakterrel kell kezdődni, akkor adható hozzá a tranzakció a blokk lánchoz (blockchain) [11].
- **Tartalom optimalizáció** esetén ha több felhasználó feltölti ugyan azt a médiát (kép, videó, hang), akkor a hash alapján be tudjuk azonosítani hogy az adott fájl már más által is felkerült-e, ebben az esetben az új feltöltő az új fájl helyett a régire fog mutatni és a másolat törlésre kerül.
- **Titkosítás** és autentikáció, azaz biztonságos azonosítás során. Erről részletesebben a 1.5 részben.

1.5. Hash a Biztonságtechnikában

1.5.1. A Jelszó Hash

A számítógépes biztonságtechnikában a hash elsősorban jelszavak kapcsán jelenik meg, amelyeket szerverekre történő távoli bejelentkezéshez használunk. A megadott jelszót a szervernek valamely formában tárolnia kell, hogy sikeresen el tudja végezni a felhasználó azonosítását, azonban amennyiben ez az eredeti kulcs szöveges formájában történne meg, esetleges behatolás esetén minden regisztrált felhasználó jelszava megszerezhetővé válna. Ez veszélyes következményekkel járhat, ugyanis sokan egy jelszót több szolgáltatásnál is újra felhasználnak. Ezek tárolására egy általános megoldás a jelszó hash-algoritmussal történő egyirányú titkosítása.

1.5.2. A Jelszó Salt

Ez a megoldás azonban egy problémát még nem old meg. Abból adódóan, hogy legtöbben egyszerű jelszavakat használnak, sokaknak ezek egyezni fognak és ezáltal az adatbázisban tárolt hozzárendelt hash-ek is. Emiatt annak ellenére hogy nem tudjuk mi a pontos jelszó, elég egy felhasználó jelszavát visszafejteni és a kapott kulcs biztosan működni fog minden másik egyező hash esetén is.

Erre megoldásként használjuk az úgynevezett "salt"-ot. Amely egy rövid karakterszorozat amit a jelszó titkosításakor véletlenszerűen generálunk, majd a jelszó végéhez fűzünk. Ez a salt végül a kiszámolt hash mellé kerül. A jelszó ellenőrzésekor a kapott jelszó után fűzzük ismét a hash mellett talált salt-ot és így futtatjuk le rajta az algoritmust. Ezáltal az azonos jelszavak is különbözően jelennek meg a végeredményben.

A visszafejtést tovább komplikálhatjuk azzal, hogy többször is lefuttatjuk az algoritmust. Először a kulcs és salt kombináción, majd az ebből kapott hash eredményen. Ezt akármeddig ismételhetjük, azonban lineárisan növekszik a titkosítás és az ellenőrzés művelet ideje.

1.5.3. Memóriaigény

Az új generációs hash algoritmusok egyik fő szempontja a VGA gyorsítás megakadályozása. Ezt általában nagy mennyiségű memóriaigénnyel érik el, amely egy általános processzor számára, amely egyenletesen éri el a teljes memóriatartományt (dual-channel esetén a felét) nem okoz jelentős lassulást, azonban egy VGA esetén azt jelenti, hogy a lényegesen gyorsabb regiszter memória helyett a globális memóriát szükséges használni. Ennek köszönhetően esetenként a VGA-val történő feltörés lassabbnak is bizonyulhat, mint egy hasonló árú processzorral [12]. Ezen oknál fogva ezek az algoritmusok kizárásra kerültek.

Algoritmus	Memóriaigény képlete	Memóriaigény
SHA-256	$2 * 256 = 512 \text{ bit} =$	64 B
Scrypt	$N * 2r * 64 \text{ B} \rightarrow 16\ 384 * 2 * 8 * 64 \text{ B} =$	16 MB

1.1. táblázat. SHA-256 És Scrypt algoritmusok memóriaigényének összehasonlítása.

Scrypt esetén a példaként felhozott N és r értékek egy tipikus konfiguráció.

2. fejezet

Felhasználói dokumentáció

2.1. A Szoftver Célja

A szoftver egy SHA-256 algoritmussal hash-elt jelszavak feltörésére alkalmas C++ és OpenCL-ben írt program. A célja a modern videókártya használat hatásának vizsgálata a jelszófeltörés sebességére nézve. Jelenleg a program képes videokártya használatával hash-elni egy jelszót, egy fájl megadásával hashelni több jelszót, vagy feltörni egy általános, vagy salt-al ellátott jelszót, amennyiben az szerepel a megadott ismert jelszavak listában. Ezt a feladatát speciálisan a használt rendszerre optimalizálva teszi.

A programot biztonságtechnikai kutatóknak, vagy diákoknak javaslok, akik érdekeltek abban, hogy milyen sebességgel törhető fel egy biztonságosnak számító jelszó a számukra elérhető hardverekkel.

2.2. A Szoftver Használata

2.2.1. Rendszerkövetelmények

A program Microsoft Windows operációs rendszeren futtatható exe formájában érhető el.

	Minimális	Optimális (vagy újabb/jobb)
Operációs Rendszer	Microsoft Windows 7	Microsoft Windows 10
Videókártya	OpenCL-t támogató	NVidia GeForce GTX 600-as széria AMD Radeon R9 / HD 7000 széria
Háttértár	7200 RPM HDD	M.2 PCIe SSD

2.1. táblázat. A program rendszerkövetelményei.

2.2.2. Telepítés

A program letöltése után egy zip állomány áll a rendelkezésre. Ezt az állomány kell kicsomagolni egy erre alkalmas programmal (például 7-Zip) és a benne található mappát egy megfelelő helyre helyezni. Fontos, hogy a mappában található fájlok mindegyike továbbra is azonos mappában helyezkedjen el, ezen kívül bárhová helyezhető a számítógépen, ahol ön rendelkezik olvasási és futtatási joggal.

A program futtatása a konzolon keresztül a fájlokat tartalmazó mappából történik, ezért érdemes kényelmes elérést biztosítani a fájlhoz. Erre egy jó módszer lehet például az alapértelmezett C meghajtó gyökérkönyvtárába helyezni a mappát, amely a következő helyen elérhető lesz: C:\gpucrack\

2.2.3. Vezérlés

A szoftver konzolos felületen indítási paraméterek megadásával konfigurálható. A szoftvert tartalmazó mappában megnyitott konzol ablakban a következő parancssal tudjuk futtatni a programot (.sha256gpu.exe). Ekkor angol nyelven a program fel-sorolja a használható parancsokat, amelyek közül választhatunk:

- platform - Felsorolja az elérhető platformokat és eszközöket.
- hash single <password> - Hash-el egy jelszót, majd kiírja az eredményt.
- hash single <password> <salt> - Hash-el egy jelszót egy salt-al együtt, majd kiírja az eredményt.

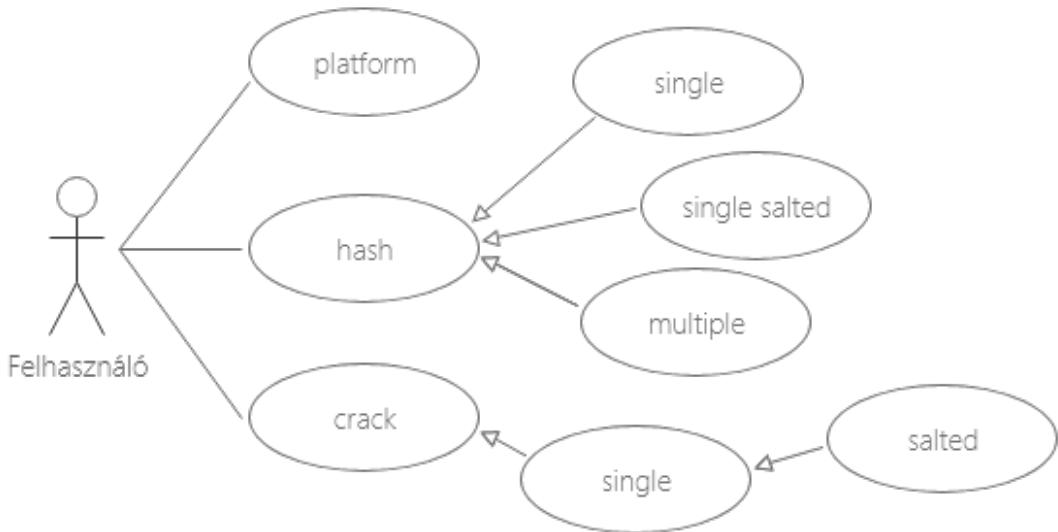
- hash multiple <source> <target> - Első paraméterként egy enterrel elválasztott kulcsokkal teli fájlt adhatunk meg, amely mindegyikét hashelni és enterrel elválasztva a target-nek meghatározott fájlba írja.
- crack single <source> <hash> - A megadott hash kódöt megpróbálja a passwords fájlban található jelszavak segítségével feltörni. Amennyiben sikerül, kiírja az eredeti kulcsot, ellenkező esetben jelzi, hogy nem talált megfelelő kulcsot.

Ezek mellett minden parancsnak megadhatunk egy kapcsolót.

Kapcsoló	Alapértelmezett	Hatás
-p <id>	0	A használt platform kiválasztása
-d <id>	0	A használt eszköz kiválasztása
-t <count>	1024	Egyszerre feltörhető jelszavak száma
-k <size>	24	Egy bemeneti kulcs maximális mérete

2.2. táblázat. A programban használható kapcsolók leírása

Minden kapcsoló használata opcionális, azonban a -t kifejezetten fontos az optimális sebesség elérésének érdekében, amelyet egy néhány teszt futtatással finomhangolni lehet. A -p és -d kapcsolók kizárolag akkor relevánsak, ha nem az alapértelmezett eszközt használjuk feltörésre. A kulcs maximális méretének csökkenése egy minimális teljesítmény növekedést eredményezhet, amennyiben ismert hogy, a maximum használt jelszó mérete például 16 volt és a mintadokumentum kizárolag ekkora vagy, ennél rövidebb jelszavakat tartalmaz.



2.1. ábra. Use-case diagram.

2.2.4. Bemenet

A bemeneti értékek minden parancsra különböznek, azonban ezek kategóriákra oszthatóak.

- <password> jelszó: egy karaktersorozat, amely kizárálag az UTF-8 táblázat elemeit tartalmazhatja
- <salt> salt: a követelmény megegyezik a jelszóéval
- <source> fájl: egy fájlt egyértelműen meghatározó elérési útvonal. Lehet abszolút és relatív. A fájl tartalmának ASCII, vagy utf-8 karaktereknek kell lennie, amelyeket sortörés karakterek választanak el egymástól.
- <target> fájl: egy fájlt egyértelműen meghatározó elérési útvonal. Lehet abszolút és relatív. A fájl vagy nem létezik (a mappában kell rendelkezni írási joggal), vagy írható (a régi fájl felülírásra kerül). Ha a mappa nem írható az ön felhasználói profiljából, indítsa a programot rendszergazdaként!
- <hash> hash: 64 hexadecimális (0-9, a-f) karaktert tartalmazó szöveg. Amennyiben salt-al rendelkezik, azok a karakterek a hash előtt helyezkednek el hozzáfűzve ahhoz. Ilyen esetben lehet hosszabb a szöveg mint 64 karakter.

Amennyiben a program hibát észlel a bemenetekkel, azonnal megszakítja a futást és a felhasználó tudtára adja, hogy mi okozta azt.

2.2.5. Kimenet

A program a kimenetet többnyire az azt elindító konzolos ablakra írja. Ezzel együtt minden esetleges hibaüzenet, vagy sikertelen futás eredménye is oda kerül. A kimenetek ezen felül tartalmaznak részletes információt például a megkapott hash felbontásáról és a futásidőről.

- platform: a számítógépen található platformok és eszközök listája
- hash single: a megadott jelszó, vagy jelszó és salt sha256 hash alakja
- hash multiple: a megadott jelszófájl jelszavainak a sha256 hash-jei a target fájlban.
- crack single: amennyiben sikeres volt a feltörés a hash-hez tartozó jelszó és egyéb információk, amennyiben nem volt sikeres, akkor ezt kiírja a felhasználónak.

2.2.6. Példák

Paraméter nélküli indítás:

```
PS A:\sha> .\sha256gpu.exe
SHA256GPU Cracker (v1.0) Commands:
platform                                : list available platforms and devices
hash single <password>                  : hash a single password
hash single <password> <salt>            : hash a single password with salt
hash multiple <source> <target>          : hash multiple passwords from source to target file
crack single <passwords> <hash>          : crack hash using a passwords source with options
Properties: -p <id>      (default: 0) platform identifier
           -d <id>      (default: 0) device identifier
           -t <count>    (default: 1024) keys cracked at once
           -k <size>     (default: 24) max key size
```

Számítási hardware-platformok kiíratása:

```
PS A:\sha> .\sha256gpu.exe platform
Use the given device ID of the graphics controller to specify the cracking device.
Platforms: 1

* NVIDIA CUDA
  OpenCL 1.2 CUDA 11.2.66
  Devices: 1

  * GeForce GTX 1070
    Device ID:      (0:0)
    Device available: yes
    Clock frequency: 1771 MHz
    Memory size: 8192 MB
    Memory alloc: 2048 MB
    Cache type: 2
    Cache size: 720 KB
    Device Version: OpenCL 1.2 CUDA
    Driver Version: 460.89
```

Hash kiszámítása:

```
PS A:\sha> .\sha256gpu.exe hash single banana
Device Attached: (0:0) GeForce GTX 1070
Compiling kernel...
Kernel compiled.
b493d48364afe44d11c0165cf470a4164d1e2609911ef998be868d46ade3de4e
```

Jelszó sikeres feltörése:

```
PS A:\sha> .\sha256gpu.exe crack single ./pw.txt b493d48364afe44d11c0165cf470a4164d1e2609911ef998be868d46ade3de4e
Device Attached: (0:0) GeForce GTX 1070
Hash: b493d48364afe44d11c0165cf470a4164d1e2609911ef998be868d46ade3de4e
Hash hexform: [ 'b493d483', '64afe44d', '11c0165c', 'f470a416', '4d1e2609', '911ef998', 'be868d46', 'ade3de4e' ]
Hash decform: [ 3029587075, 1689248845, 297801308, 4101022742, 1293821449, 2434726296, 3196489030, 2917391950 ]
Compiling kernel...
Kernel compiled.
Initializing kernel...
Cracking...
Crack kernel finished.
=====
Match found.
Key: 'banana'
Line: 436
=====
Runtime: 1172 microseconds.
  0.001172 seconds.
```

Jelszó sikertelen feltörése:

```
PS A:\sha> .\sha256gpu.exe crack single ./pw.txt 2519876965ed4521e57444da1573ae1elfb8944cf98c879339e6f5b68a8911b5
Device Attached: (0:0) GeForce GTX 1070
Hash: 2519876965ed4521e57444da1573ae1elfb8944cf98c879339e6f5b68a8911b5
Hash hexform: [ '25198769', '65ed4521', 'e57444da', '1573ae1e', '1fb8944c', 'f98c8793', '39e6f5b6', '8a8911b5' ]
Hash decform: [ 622430057, 1710048545, 3849602266, 359902750, 532190284, 4186736531, 971437494, 2324238773 ]
Compiling kernel...
Kernel compiled.
Initializing kernel...
Cracking...
Crack kernel finished.
=====
No match found.
Lines verified: 3735367
=====
Runtime: 582434 microseconds.
  0.582434 seconds.
```

Jelszó paraméterezett feltörése:

```
PS A:\sha> .\sha256gpu.exe crack single .\pw.txt 59557cf1890bf0b7458c1e66119ab01c3a796fd09df296ef7e70745d29934777
-p 0 -d 0 -t 22400 -k 16
Device Attached: (0:0) GeForce GTX 1070
Hash: 59557cf1890bf0b7458c1e66119ab01c3a796fd09df296ef7e70745d29934777
Hash hexform: [ '59557cf1', '890bf0b7', '458c1e66', '119ab01c', '3a796fd0', '9df296ef', '7e70745d', '29934777' ]
Hash decform: [ 1498774769, 2299261111, 1166810726, 295350380, 981037008, 2649921263, 2121299037, 697517943 ]
Compiling kernel...
Kernel compiled.
Initializing kernel...
Cracking...
Crack kernel finished.
=====
Match found.
Key: 'ex-wethouder'
Line: 3735367
=====
Runtime: 302462 microseconds.
0.302462 seconds.
```

Jelszavak hashelése kimeneti fájlba:

```
PS A:\sha> .\sha256gpu.exe hash multiple .\pw.txt .\hashes.txt -t 25000
Device Attached: (0:0) GeForce GTX 1070
Input file: .\pw.txt
Output file: .\hashes.txt
Compiling kernel...
Kernel compiled.
Initializing kernel...
Hashing...
Hash kernel finished.
Runtime: 1067441 microseconds.
1.067441 seconds.
```

Jelszó és hash-fájlok tartalma (bal: bemenet, jobb: kimenet):

pw.txt	hashes.txt
A: > sha > pw.txt	A: > sha > hashes.txt
1 12345	1 5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5
2 abc123	2 6ca13d52ca70c883e0f0bb101e425a89e8624de51db2d2392593af6a84118090
3 password	3 5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
4 passwd	4 0d6be69b264717f2dd33652e212b173104b4a647b7c11ae72e9885f11cd312fb
5 123456	5 8d96eeef6ecad3c29a3a629280e686fc0c3f5d5a86aff3ca12020c923adc6c92
6 newpass	6 253c2e786c2414dcaec8dbf11df51b5075371454b93a5687d24d96ddbfb3b939
7 notused	7 088931cd307077a62125d055221e3fe559c50f4690eb6c61ea6bbea6c1bf4be
8 Hockey	8 9b32950cea51fb5b7dea0c5a5597ff08a3e327647466382e61bcba14f84744ee
9 internet	9 3b0fe0d342e9fa16a5c68dba33f2e63c024f72a9d4c1ce1028570101d5229ff
10 asshole	10 f50c51ed2315dcf3fa88181cf033f8029cac64f7dea0408327ca032ec102ea74
11 Maddock	11 1420b3f93135f789a83d2cd8c44a3230dd6b4c04cdeafcfcdd672c93b3746c25
12 12345678	12 ef797c8118f02dfb649607dd5d3f8c7623048c9c063d532cc95c5ed7a898a64f
13 newuser	13 9c9064c59f1ffa2e174ee754d2979be80dd30db552ec03e7e327e9b1a4bd594e
14 computer	14 aa97302150fce811425cd84537028a5afbe37e3f1362ad45a51d467e17afdc9c
15 Internet	15 57e8a431deec0d70da026ea3392e59688b11b79edfd04e9da823b16bcd1d4d7
16 Mickey	16 18556c17689e414361542867d3038c235eaf63cde6fd69fdb3624ecb570151
17 qwerty	17 65e84be33532f784c48129675f9eff3a682b27168c0ea744b2cf58ee02337c5
18 fiction	18 d778b673710a7c96d31cae27a4cb6df7ce7b8470747b922091ee18ac2fa1dd7
19 Cowboys	19 0a2099876d72b6cab074c3b27a62d8155f0c2acef75de1b433f54e180da0ee8
20 Jordan	20 e8bfe1ed693510570ced8b5ee70049cc4b985a77ec066ee345892f685d72cca6
21 Hatton	21 ecf6772956e182294c10ebf17cff454d57d7e7d845ec2bebb937863f76742a0d
22 test	22 9f86d081884c7d659a2fea0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08
23 Michael	23 f089eaeef57aba315bc0e1455985c0c8e40c247f073ce1f4c5a1f8ffd8773176
24 ou812	24 f1e5a6b9186111bbd7e173b8aafe57ca3884eecb08f7fe561aca12ed3fd875a
25 orange	25 1b4c9133da73a71122404314402765ab0d23fd362a167d6f0c65bb215113d94
26 1234	26 03ac674216f3e15c761ee1a5e255f067953623c8b388b4459e13f978d7c846f4
27 Beavis	27 e685b416fb3fe8e9e74bd920b506ebd7c143cba34e0775631a0d11fa53ef20e7
28 123	28 a665a45920422f9d417e4867efdc4fb8a04a1f3fff1fa07e998e86f7f7a27ae3
29 tigger	29 abc529a4b673cbbb532e584706cb8137be876ad53269df3b97fdb40fc76fe57
30 Soccer	30 863e3defee2faf540de3428f08ab5e568486481987db7fadd12363f3282d7d9

2.2.7. Hiba Esetén

A program hiba esetén leáll és leírással próbálja segíteni a probléma diagnosztizálását. Amennyiben ismeretlen hiba történik, próbálja meg a következő lépéseket elvégezni:

1. Ellenőrizze le, hogy a parancs megfelelő formátumú-e.
2. Próbálja meg kapcsolók nélkül futtatni a programot.
3. Indítsa el a programot a (platform) parancs segítségével és bizonyosodjon meg róla, hogy található legalább egy eszköz.
4. Frissítse a használni kívánt eszközt a legújabb driver verzióra.
5. Frissítse a C++ Runtime szolgáltatást a legújabb verzióra.
6. Amennyiben fájlal dolgozik, bizonyosodjon meg róla, hogy az elérési út megfelelő, a fájlhoz van hozzáférése és az olvasható.
7. Bizonyosodjom meg róla hogy az eredeti zip-ben tárolt fájlok mindegyike megtalálható a futtatható fájlal egy mappában.
8. Futtassa a programot rendszergazdai jogosultsággal.

3. fejezet

Fejlesztői dokumentáció

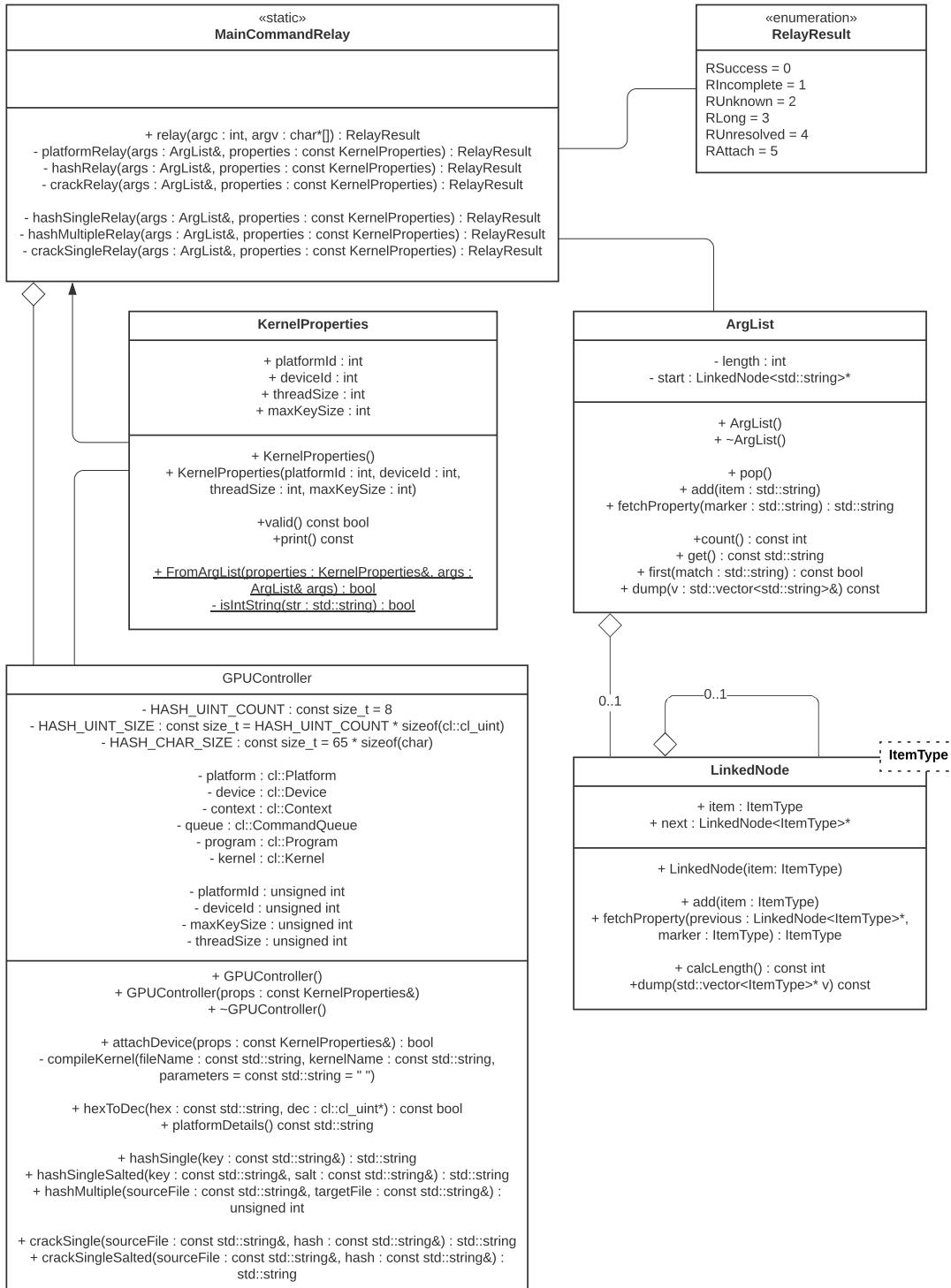
3.1. A Program Felépítése

A program futása a MainCommandRelay statikus osztályban kezdődik. Az itt található statikus relay metódus meghívása után az megpróbálja a felhasználótól futtatási paraméterként kapott parancsot értelmezni az ArgList segítségével. Ez alapján elkészít egy KernelProperties objektumot, amely egy pontos leírást ad a feladat tulajdonságairól a GPUController számára. Ennek használatával elkészíti a GPUController-t, majd futtatja a parancsnak megfelelő metódust.

A GPUController valamely feladatának futása után, vagy hiba esetén, a meghívott relay metódus visszatér egy RelayResult-al, amely a futás konklúzióját tartalmazza. Ezt az értéket egész számkként visszaadja az operációs rendszernek a main függvény visszatérési értékeként, így az nyomon tudja követni, hogy fellépett-e bármiilyen hiba a futás során.

A GPUController egy feladat futtatását az OpenCL kernel fájlok betöltésével, konfigurálásával, majd fordításával kezdi. Ezeket hozzáköti a felhasználó által meghatározható eszközökhez és amennyiben minden megfelel, futtatni kezdi. Ez a futás lehet egyszálú gyors, mint például a hashSingle esetében, vagy sokszálú és időigényes például a crackSingle esetében.

Amennyiben az adott művelet futása sikeresen zárult, az az eredményét visszaadja szöveg vagy egész szám formájában.



3.1. ábra. A program UML osztálydiagramja.

3.2. Az OpenCL

3.2.1. Alapok

1. Definíció. *Heterogén rendszer: több processzor együttműködésén alapuló számítógépes rendszer.*

2. Definíció. *uint: egy 32 biten tárolt előjel nélküli (csak pozitív) egész szám, amelyen minden műveletet mod 2^{32} értelmezünk.*

3. Definíció. *Hoszt (host): Több processzoros rendszerek esetén azon eszköz, amely a többi irányításáért felel.*

4. Definíció. *Eszköz (device): Több processzoros rendszerek esetén azon eszköz, amely a hosztól kapott feladatokat elvégzi.*

5. Definíció. *H/s: Hash per second. Egy rendszeren átlagosan egy másodperc alatt elkészített hash-ek számát mutatja egy adott algoritmussal. A H/s a feltörésre is használható, ugyanis futási időben minimális a kettő között a különbség.*

Az OpenCL egy nyílt forráskódú, cross-platform keretrendszer olyan homogén számítógépes környezetekhez, ahol a CPU és GPU vagy egyéb processzorok és ezek szálai párhuzamos együttműködésére van szükség. [13].

A keretrendszer egy saját programozási nyelvvel rendelkezik, mely a C nyelvből származtatott. Úgynevezett kerneleket tudunk írni, amelyeket az eszközökön tudunk futtatni. A projekt során a C++ API-t fogom alkalmazni, azonban sok nyelvhez elérhető hasonló könyvtár.

Az OpenCL keretrendszer inicializálásakor megadható, hogy melyik eszközöket használjuk, illetve milyen szerepet fognak ezek betölteni. Szerepe alapján a hardvereket hoszt és eszköz kategóriákra bonthatjuk. Emellett fontos különbség az adott hardverek platform-ja. minden platform különböző implementációját futtatja a az OpenCL keretrendszernek (pl.: Intel, AMD, NVidia).

A program fordításának idejében nem ismerjük a jövőbeli felhasználó számítógépének pontos paramétereit, holott ez elengedhetetlen információ a fordító számára. Erre két megoldás közül tudunk választani:

1. A kódot minden napjainkban használt eszközre optimalizálva lefordítjuk, illetve a későbbiekben amennyiben ez változik, új verziókat készítünk.

2. Nem fordítjuk azonnal az eszközre szánt kernel kódot, ehelyett azt eltároljuk a program számára elérhető helyen, majd amikor a felhasználó először futtatja a programot, lefordítjuk a kernelt.

Látható, hogy az első megoldás szinte kivitelezhetetlen, hiszen naprakészen tartani több ezer vagy akár tízezer konfigurációt lényegesen nagyobb költséggel jár, mint a felhasználó számára az első alkarrömmel várni akár kevesebb mint egy másodpercet hogy leforduljon a kód. A második megoldás hátránya azonban, hogy nem tudjuk elrejteni a kernel kódunkat és az könnyen másolható lesz¹. Ezzel szemben a kód futásidőben szerkeszthető marad, amely tulajdonságát fontos lesz a program optimalizálása során.

```

1 kernel void add(global const int* A,
2                  global const int* B,
3                  global int* C)
4 {
5     int i = get_global_id(0); //Global ID in dimension 1 (0)
6     C[i] = A[i] + B[i];
7 }
```

3.1. forráskód. Tetszőleges méretű vektor összeadása OpenCL kernellel (forrás: Elte IK Computer Graphics GPGPU)

3.2.2. Optimalizálás

A szakdolgozat jelentős részét képzi az eszköz oldali parallel kód futásának optimalizálása. Ezen optimalizálás a lefordított kód elemzésével történhet legeffektívebben. Ehhez az AMD APP Kernel Analyzer-t használtam. Ez az eszköz a lefordított programkód utasításait visszafejt assembly nyelvre. Az assembly kód betekintést adhat az esetleges optimálatlan kódrészekbe.

Videókártyák esetén egy potenciális 50 % lassítást jelenthet az elágazások használata. A futó szálak egyszerre egy parancsot képesek elvégezni, így aztán amíg bizonyos szálak egy elágazás hosszabb igaz oldalával dolgoznak, addig a többinek várni kell azokra még akkor is, ha gyorsabban végeztek a második felével. Ez alapján

¹Erre egy részleges megoldást nyújthat a Spir-V

mindig elkerülöm a balanszolatlan hosszúságú elágazásokat, de igyekszem őket ahol csak lehetséges nélkülözni.

Szintén nagyobb teljesítményromálssal járhat a ciklusok használata. Egy ciklus minden lépése egy memóriacím növelésével, illetve egy elágazással jár. Emellett a cikluson belül felhasznált iterátor változót minden lépésnél fel kell használni akár műveletre (mely emiatt nem optimalizálható a fordító által), vagy egy memóriacím lekérdezésére, amely az iterációs változó volatilitása miatt nem kerülhet a előre cache-be. Erre nyilvánvaló megoldást jelenthet az elkerülésük, vagy a ciklusok fixált lépésszámúvá tétele, majd a kiterítése. Pl:

```

1 #pragma unroll
2 for (int i = 0; i < 4; i++)
3 {
4     keys[i] = (i + 1) * 10;
5 }
```

3.2. forráskód. A ciklus unroll előtt.

A **#pragma unroll** kulcsszó jelzi a fordítónak, hogy a következő ciklus kiteríthető. Ez természetesen csak fordítás időben konstans hosszúságú ciklusok esetén alkalmazható.

```

1 keys[0] = 10;
2 keys[1] = 20;
3 keys[2] = 30;
4 keys[3] = 40;
```

3.3. forráskód. A ciklus unroll után.

A kiterített kód-ban előre látszik hogy a jelenlegi lépés során melyik mező lesz a következő, amelybe írni fogunk, ezért ez megoldható cachelés segítségével. Emellett az értékeket is műveletek helyett konstansokra tudja váltani a fordító.

3.3. Az SHA-256 Aritmetikája

A hash kiszámolása közben több olyan műveletet alkalmazunk, amelyek együttes működése elősegíti a ténylegesen véletlenszerűnek tűnő eredmény előidézését.

3.3.1. Definíciók

6. Definíció. Logikai operátorok: *AND*, *OR*, *XOR*, *NOT* sorrendben a következő jelekkel feltüntetve: \wedge , \vee , \oplus , \neg .

7. Definíció. Integer összeadás: $A + B = A + B \bmod 2^{32}$. A modulusz hatására a memóriaszektoron túlcsorduló bitek eltűnnek és minimum értékről kezdődik újra a számolás.

8. Definíció. Integer Bitshift: $A << N = A * 2^N \bmod 2^{32}$, vagy $A >> N = A/2^N \bmod 2^{32}$. Az adott memóriaszektorban található bitek N darabszor balra vagy jobbra tolódnak. A szektoron kívülre eső biteket töröljük.

9. Definíció. Balanszolt bitművelet: Olyan bitműveletek, melyek interpretációjában azonos számú igaz és hamis szerepel. Példa balanszolt műveletre: *XOR*, *NOT*.

\neg	\wedge	\oplus
1 0	1 1 0	1 0 1
0 1	0 0 0	0 1 0

3.1. táblázat. Három bináris függvény eredménye.

A példákból látszik hogy nem minden bináris függvény esetén kapunk arányos mennyisésgű igaz és hamis értéket. Ez természetesen azt okozza hogy n lépés esetén az eredmény konvergálni fog a magasabb esélyű értékhez.

Művelet		Kimenet		Balanszolt
Neve	Jele	1	0	
NOT	\neg	50 %	50 %	igen
AND	\wedge	25 %	75 %	nem
OR	\vee	75 %	25 %	nem
XOR	\oplus	50 %	50 %	igen

3.2. táblázat. Bináris műveletek balanszoltságának összehasonlítása.

Példaként válasszunk véletlenszerű 8 bites egész számokat: [147, 71, 11, 156]

Decimális	Bináris
147	10010011
71	01000111
11	00001011
156	10011100
\wedge	00000000
\oplus	01000011

Látható, hogy már 4 művelet elvégzése után az \wedge művelet balanszolatlansága következményeként a kimenet csupa hamisból áll. Ezzel szemben a \oplus művelet esetén maradt 3 igaz bit, amely pontosan egy-el kevesebb mint a fele. A bemenet 32 bitje közül 15 igaz volt, ami szintén alulról közelíti a felét.

3.3.2. Funkciók és Konstansok

Az algoritmus a következő funkciókat fogja használni, melyek együttes működése balanszolt kimenetet ad:

$$\begin{aligned}
 Shr(A, N) &= (A >> N) \\
 Rotr(A, N) &= (A >> N) \vee (A << (32 - N)) \\
 Ch(X, Y, Z) &= (X \wedge Y) \oplus (\neg X \wedge Z) \\
 Maj(X, Y, Z) &= (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z) \\
 \Sigma_0(X) &= Rotr(X, 2) \oplus Rotr(X, 14) \oplus Rotr(X, 22) \\
 \Sigma_1(X) &= Rotr(X, 6) \oplus Rotr(X, 11) \oplus Rotr(X, 25) \\
 \sigma_0(X) &= Rotr(X, 7) \oplus Rotr(X, 18) \oplus Rotr(X, 3) \\
 \sigma_1(X) &= Rotr(X, 17) \oplus Rotr(X, 19) \oplus Rotr(X, 10)
 \end{aligned}$$

Az algoritmus kiindulópontjaként véletlenszerű számokat kellett választani. A tényleges véletlenszerűség elengedhetetlen részét képezi szinte minden titkosító eljárásnak. Ezeket számítógép nem tudja előállítani valamilyen külső bemenet megadása nélkül. Például a random.org weboldala atmoszférikus zaj alapján generálja a véletlenszerű számokat. Ennél egy egyszerűbb módszer volt az első 64 prímszám köbgyökének a tört részének az első 32 bitjét venni.

$$\{ \lfloor (\sqrt[3]{i} \bmod 1) * 2^{32} \rfloor \mid i \in P(64) \}, \text{ ahol } P(n) \text{ az első } n \text{ prímszám}$$

K_1	$\sqrt[3]{2} \approx$	1.25992104989	\rightarrow	0x428a2f98
K_2	$\sqrt[3]{3} \approx$	1.44224957031	\rightarrow	0x71374491
K_3	$\sqrt[3]{5} \approx$	1.70997594668	\rightarrow	0xb5c0fbcf
K_4	$\sqrt[3]{7} \approx$	1.91293118277	\rightarrow	0xe9b5dba5
K_5	$\sqrt[3]{11} \approx$	2.22398009057	\rightarrow	0x3956c25b
K_n	$\sqrt[3]{\dots} \approx$...	\rightarrow	...
K_{64}	$\sqrt[3]{311} \approx$	6.77516895227	\rightarrow	0xc67178f2

Az eljárás végén a tömörítéshez szükség volt még további 8 darab 32 bites számra.

Ezek értékét az első 8 prímszám négyzetgyökének a tört részének az első 32 bitjéből kapjuk. Ezek kiszámolására és ellenőrzésére használható a következő JavaScript kód:

```

1   () =>
2   {
3       [2,3,5,7,11,13,17,19].forEach((i) =>
4           console.log(parseInt((Math.sqrt(i) % 1).toString(2) .
5               slice(2, 34), 2).toString(16)))
6   }()

```

forráskód 3.4 JavaScript kód a kiinduló hexadecimális számok kiszámolására.

H_1	$\sqrt{2} =$	1.41421356237	\rightarrow	0x6a09e667
H_2	$\sqrt{3} =$	1.73205080757	\rightarrow	0xbb67ae85
H_3	$\sqrt{5} =$	2.23606797750	\rightarrow	0x3c6ef372
H_4	$\sqrt{7} =$	2.64575131106	\rightarrow	0xa54ff53a
H_5	$\sqrt{11} =$	3.31662479036	\rightarrow	0x510e527f
H_6	$\sqrt{13} =$	3.60555127546	\rightarrow	0x9b05688c
H_7	$\sqrt{17} =$	4.12310562562	\rightarrow	0x1f83d9ab
H_8	$\sqrt{19} =$	4.35889894354	\rightarrow	0x5be0cd19

3.3.3. Blokkok Számítása

A blokká alakítás során a teljes üzenetet $n * 512$ bites méretűvé alakítjuk, ahol minden blokk 512 bites lesz. Egy optimalizálásként ezt a lépést részben kihagyhatjuk, hiszen feltételezzük, hogy a megkapott jelszó nem fogja felvenni az egy blokkban rendelkezésre álló 448 bitet (a maradék 64 az eredeti üzenet hossza), amely 56 byte,

tehát 64 ASCII, vagy 56 UTF-8 karakter tárolására képes. Így aztán egy blokkot készítünk a következőképpen:

1. Az első bitek az megkapott kulcs karaktereinek a bitjei,
2. a következő bit az üzenetet záró 1 bit,
3. ezt követi k darab 0 bit, ahol $k = 448 - 1 - h$ (h : szöveg bitjeinek száma),
4. az utolsó 64 bitre h azaz az üzenet hossza kerül 64 bites integerként.

Az elkészített 512 bites blokkot egyből fel is bontjuk 16 darab 32 bites számra, amelyeket elhelyezünk egy W tömb első 16 elemeként. A maradékot a következő formulával számoljuk:

$$W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} \quad (16 < i \leq 64)$$

Fontos hozzátenni, hogy az összeadások alatt az integer összeadást értjük.

3.3.4. Hash Tömörítés

Jelenleg a kód felbontásra került a W tömb elején, majd a további mezőit feltölöttük módosított elemekkel a tömb elején kiindulópontként véve a σ_0 és σ_1 segítségével. Ezeket az értékeket vissza kell tömöríteni 256 bitre. A tömörítés közben a Σ_0 , Σ_1 , Maj és Ch műveleteket fogjuk használni, illetve az értékeket forgatni.

Beállítás:

$$(a, b, c, d, e, f, g, h) = (H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8)$$

64 kör iterálás, mely a következőből áll: $(1 \leq i \leq 64)$

$$\begin{aligned} T_1 &= h + \Sigma_1(e) + Ch(e, f, g) + K_i + W_i \\ T_2 &= \Sigma_0(a) + Maj(a, b, c) \end{aligned}$$

$$\begin{aligned} h &= g \\ g &= f \\ f &= e \\ e &= d + T_1 \\ d &= c \\ c &= b \\ b &= a \\ a &= T_1 + T_2 \end{aligned}$$

Összefűzés:

$$H = (H_1 + a) \cdot (H_2 + b) \cdot (H_3 + c) \cdot (H_4 + d) \cdot (H_5 + e) \cdot (H_6 + f) \cdot (H_7 + g) \cdot (H_8 + h)$$

A (\cdot) művelet jelen esetben bináris sorozatok összefűzését jelenti. Az összefűzés során a memóriaterület minden bitjét felhasználjuk, nem hagyjuk figyelmen kívül az első 1-től balra található 0-kat. A H változó ebben az esetben egy 256 bites memóriaterületet takar.

3.3.5. Kimenet

A H változó értéke konvertálható hexadecimális formába, majd visszamásolható a host memóriába hash-elés esetén, vagy összehasonlítható egy előre megkapott 256 bites kulccsal ellenőrzés vagy feltörés esetén. Amennyiben legalább egy bit eltér a megkapott és a vizsgált hash-ek között, a bemeneti adatok biztosan különböznek. Amennyiben biteknénti az egyezés, ez elégsges bizonyítékként szolgál, hogy a bemeneti értékek megegyeztek, ugyanis két nem megegyező bemenetre ugyan azt az eredményt kapni elhanyagolható valószínűségű.

A hexadecimális felírási formához először 64 blokkra kell osztani a memóriaterületet, majd a blokkban található 4 bit méretű területekhez hozzárendelni a megfelelő hexadecimális karaktert a következő táblázat alapján

Bin	Dec	Hex	ASCII
0000	0	0	48
0001	1	1	49
0010	2	2	50
0011	3	3	51
0100	4	4	52
0101	5	5	53
0110	6	6	53
0111	7	7	53
1000	8	8	54
1001	9	9	55
1010	10	A	65
1011	11	B	66
1100	12	C	67
1101	13	D	68
1110	14	E	69
1111	15	F	70

Ezt egyszerűen meg tudjuk tenni úgy, hogy ha 8 bites számokként tekintjük a 256 bites memóriaterületet, hiszen ez a legkisebb egyedileg címezgető terület, majd minden 8 bites számot logikai \wedge művelettes éselünk az első vagy második felén csupa 1-ből álló 8 bites számmal. Pl:

Tegyük fel hogy, az első 8 bites memóriaterület: 10110011:

$$\begin{aligned} 10110011 \quad \wedge \quad 00001111 &= 0011 \rightarrow 4 \\ 10110011 \quad \wedge \quad 11110000 &= 1011 \rightarrow B \end{aligned}$$

Tehát a hexadecimálisan felírható alak: **4B**. Ezt a folyamatot ismételhetjük 32-szer. minden iteráció 2 karaktert eredményez, amely 64 karaktert jelent. A 64 hexadecimális karakter pedig megfelel az eredeti specifikációnak. Ezt a feladatot ellátja a következő kódrész ezt a feladatot oldja meg azzal a különbséggel, hogy 8 bit méretű blokkok helyett a már rendelkezésre álló 32 bites blokkokon fog iterálni. Ez azonban az unroll miatt teljesen lineáris időben fut majd.

```

1 char hex_charset [] = "0123456789abcdef";
2 #pragma unroll
3 for (int i = 0; i < 8; i++)
4 {
5     #pragma unroll
6     for (int j = 8-1; j >= 0; hashInts[i] >>= 4, --j)
7     {
8         result[(i * 8) + j] = hex_charset[ hashInts[i] & 0xf ];
9     }
10 }
11 result[64] = 0;

```

3.5. forráskód. 32 bites egész szám tömb hexadecimális karaktersorozattá konvertálása.

Az algoritmus bemeneteként megkapja a **hashInts** 8 darab 32 bites egész számot tartalmazó tömböt, és az ASCII Hexadecimális karaktereivé konvertált kimenetet elhelyezi a 65 byte méterű **result** karaktertömbben. A kimeneti tömb mérete a szöveg hosszánál egy byte-al nagyobb, hogy egy null karakter elférjen a string lezáráshoz. A karakterek kiválasztását legegyeszerűbben egy lookup táblázattal érjük el, ugyanis nem szekvenciálisak a kimenet karakterei. Több jelszó hashelése és fájlba írása esetén a null érték lezárást helyett egy enter (0x0D) karaktert használtam, így a kimenet azonnal írható lesz fájlba.

3.3.6. Salt

A salt hozzáadásával a jelszótörés tovább komplikálódik, hiszen azonos jelszavak különböző salt használatával más eredményt generálnak. A salt egy publikus kulcsnak tekinthető, hiszen a véleges hash-hoz hozzáfűzve tároljuk és minden hashelésnél véletlenszerűen újat generálunk.

Példa:

Kulcs	Salt	Hash
banana		b493d48364afe44d11c0165cf470a416 4d1e2609911ef998be868d46ade3de4e
banana	Q9wvI9A	50622ccfa4c8f58bd952b62f7fabe475 11fec498985921d6b13ac178cb413aee
banana	Joz1BL1T	7da2b105a959cff3b2c03c0c15fa11fa 124636a21451eeeadd00cb7654c664f7e

3.3. táblázat. Azonos jelszó más hashet generál különböző kulcsok használatával.

Ezeket a salt-okat minden crack esetén a kulcs után fűzzük és így végezzük el a hashelést. Ezek után a kimenet elé helyezzük. A kulcs hossza minden esetben $L = 64$, ahol L az összefűzött hash és salt hossza.

3.4. Párhuzamosítás

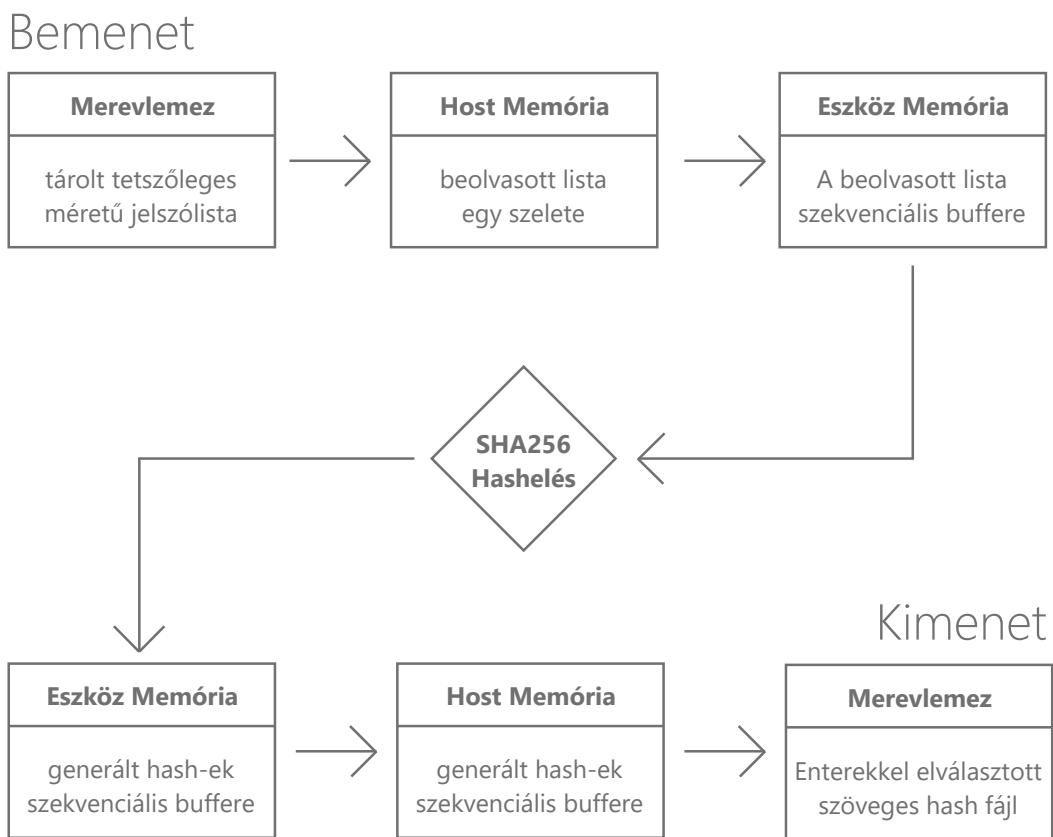
3.4.1. Alapok

A párhuzamosítás (parallelizáció) napjainkban az egyik fő módszere a számítógépek teljesítményének növelésére. Egyre nagyobb mennyiségű párhuzamosítható szálat tartalmaznak a mai processzorok, azonban ezek továbbra sem versenyképesek a videókártyák parallel teljesítményével, melyek akár százszor vagy ezerszer annyi műveletet képesek elvégezni, bár valamivel lassabban.

A videókártyák SIMD (Single Instruction, Multiple Data) programozási paradigmát használnak. Ez annyit jelent hogy az eszköz ugyan azt az utasítást hajtja végre sokszor több különböző adatra. Ez azt okozza, hogy például elágazások esetén minden lehetőség lefut minden szálon, de csak a választott irány értéke marad meg. Ezen kívül használhatnak SPMD (Single Program, Multiple Data) paradigmát is, amely esetén az egyforma programokat csoportokra bontjuk és ezek párhuzamosan futnak majd, de nem feltétlenül pontosan instrukciónként egyszerre.

3.4.2. Adatmozgatás

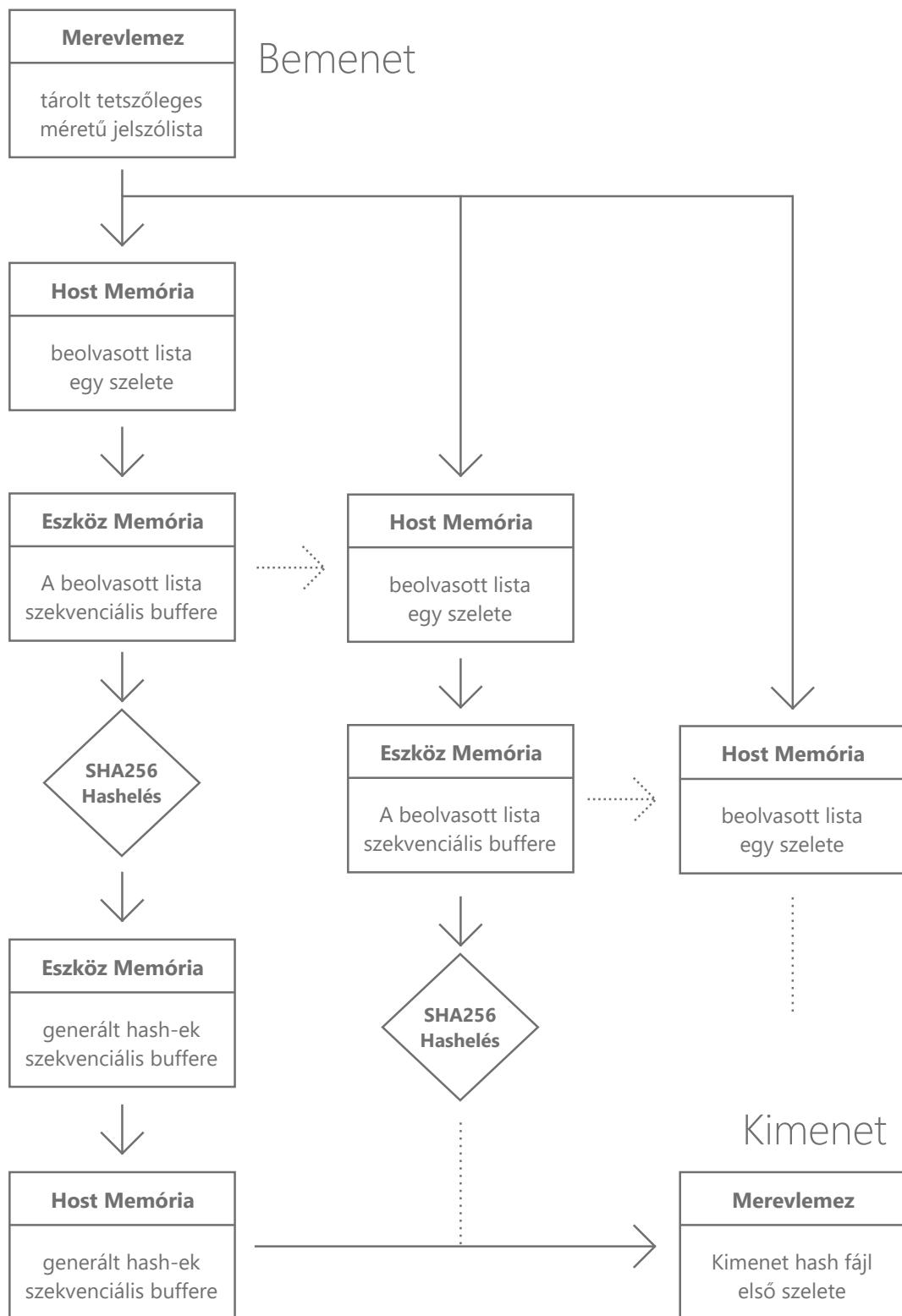
Az egyik fő limitáció az adatok mozgatása, melyet használat előtt be kell töltenünk az eszköz memóriájába, majd az eredményt visszamásolni a host memóriába. Ennek a sebességét sok tényező befolyásolhatja. Emellett esetünkben a jelszavak feltöréséhez használt jelszó táblázatot is be kell másolnunk először a host memóriába lemezről, amely további jelentős limitációt jelent. Emiatt a projekt egyik fő kihívását ezen adatmozgások parallel és aszinkron elvégzése fogja jelenteni.



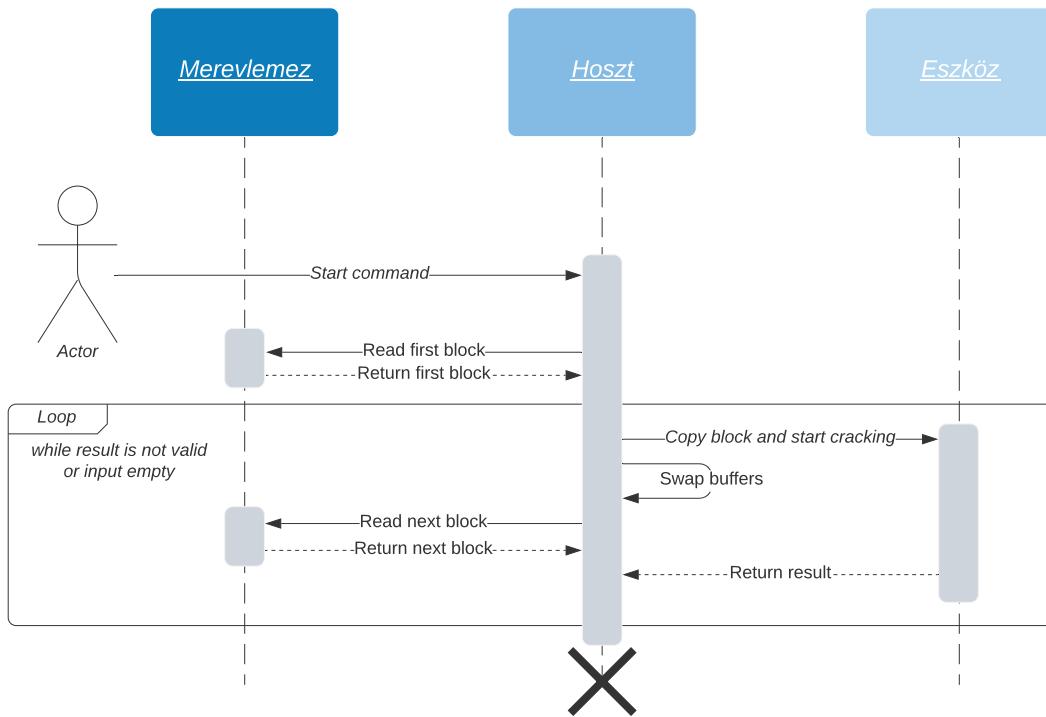
3.2. ábra. Adatmozgazás egy fájl tartalmának hashelése során.

Az ábrából látható, hogy amennyiben a hashelés közben történő adatmozgatást nem vesszük figyelembe, összesen 5 különböző alkalommal kell majd másolnunk. Ezen adatmásolások közül a legtöbbet az adatok beolvasása, illetve kiírása fogja igényelni. A második leglassabb a host és az eszköz memória közötti mozgatás, így aztán ezeket próbálom a lehető legjobban parallelizálni. Ahogy befejeződött a merevlemezről történő beolvasás és az továbbításra került az eszköz felé, egyből kezdhetjük

a következő adatbeolvasást a merevlemezről.



3.3. ábra. Adatmozgazás parallel módon egy fájl tartalmának hashelése során.



3.4. ábra. Az adatmozgatás és futtatás szekvenciadiagramja feltörés során.

3.4.3. Hashelés

Az SHA256 algoritmus lépései egy hashen belül nem párhuzamosíthatóak. Az algoritmus így lett elkészítve, hiszen ha parallelizálható lenne akkor könnyebben feltörhető lenne, hiszen:

- Egy hashelés részei gyorsíthatóak lennének, hiszen egy hashen sok szál tudna dolgozni egyszerre, ezzel exponenciálisan gyorsítva a feltörést,
- egy hashelés részekre bontható lenne, azaz bizonyos jelszavak részeinek kiszámolásához nem lenne szükség minden részét számolni újra, hiszen azt egyszer már kiszámoltuk.

Több jelszó hashelése azonban nem függ egymástól, így az egyszerre futtatható feltörések számának kizárával az eszköz és a beolvasás sebessége szab határt. A jelszavak parallel feltöréséhez szükség van egy fix méretű bemeneti és kimeneti bufferre, amelyen belül a szálak ki tudják számolni a pontos bemeneti és kimeneti pontjukat a következőképp:

Legyen:

- N : kulcsok száma
- M : egy kulcs maximális mérete
- P_0 : bemeneti buffer kezdőpontja, mérete: $[N * M]$
- P_1 : kimeneti buffer kezdőpontja, mérete: $[N * 65]$
- I : $[0..N - 1]$ jelenlegi szál indexe

ekkor a jelenlegi szál:

- Bemenetének kezdete = $P_0 + (I * M)$
- Bemenetének vége = $P_0 + (I * M) + (M - 1)$
- Kimenetének kezdete = $P_1 + (I * 65)$
- Kimenetének vége = $P_1 + (I * 65) + 64$

A kimeneti hash mérete 64 karakter, melyhez hozzájön egy null vagy enter.

3.4.4. Feltörés

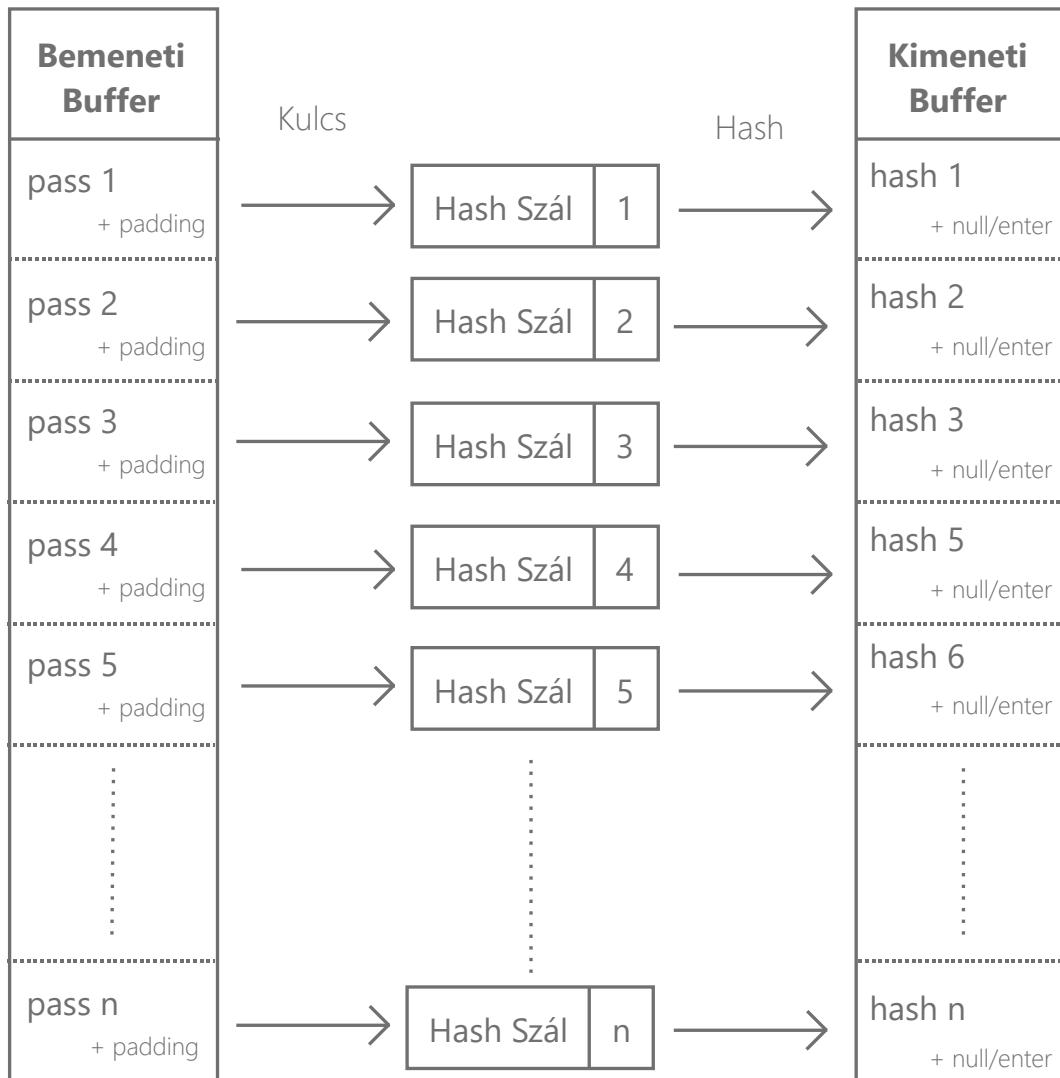
Feltörés esetén azonban hashelni minden kulcsot, majd a kimenetüket kimásolni és a host-on végigiterálni egyezést keresve egy lassú és felesleges művelet lenne. Ezért ebben az esetben az első beolvasás előtt már bemásoljuk az eszköz kerneljébe a keresendő hash kódöt és salt-ot és kizárolag egyezés esetén választ az output bufferbe.

Legyen:

- N : kulcsok száma
- M : egy kulcs maximális mérete
- S : a salt mérete
- P_0 : bemeneti buffer kezdőpontja, mérete: $[N * (M + S)]$
- P_1 : kimeneti buffer kezdőpontja, mérete egy 4 byte-os egész számnak felel meg
- I : $[0..N - 1]$ jelenlegi szál indexe

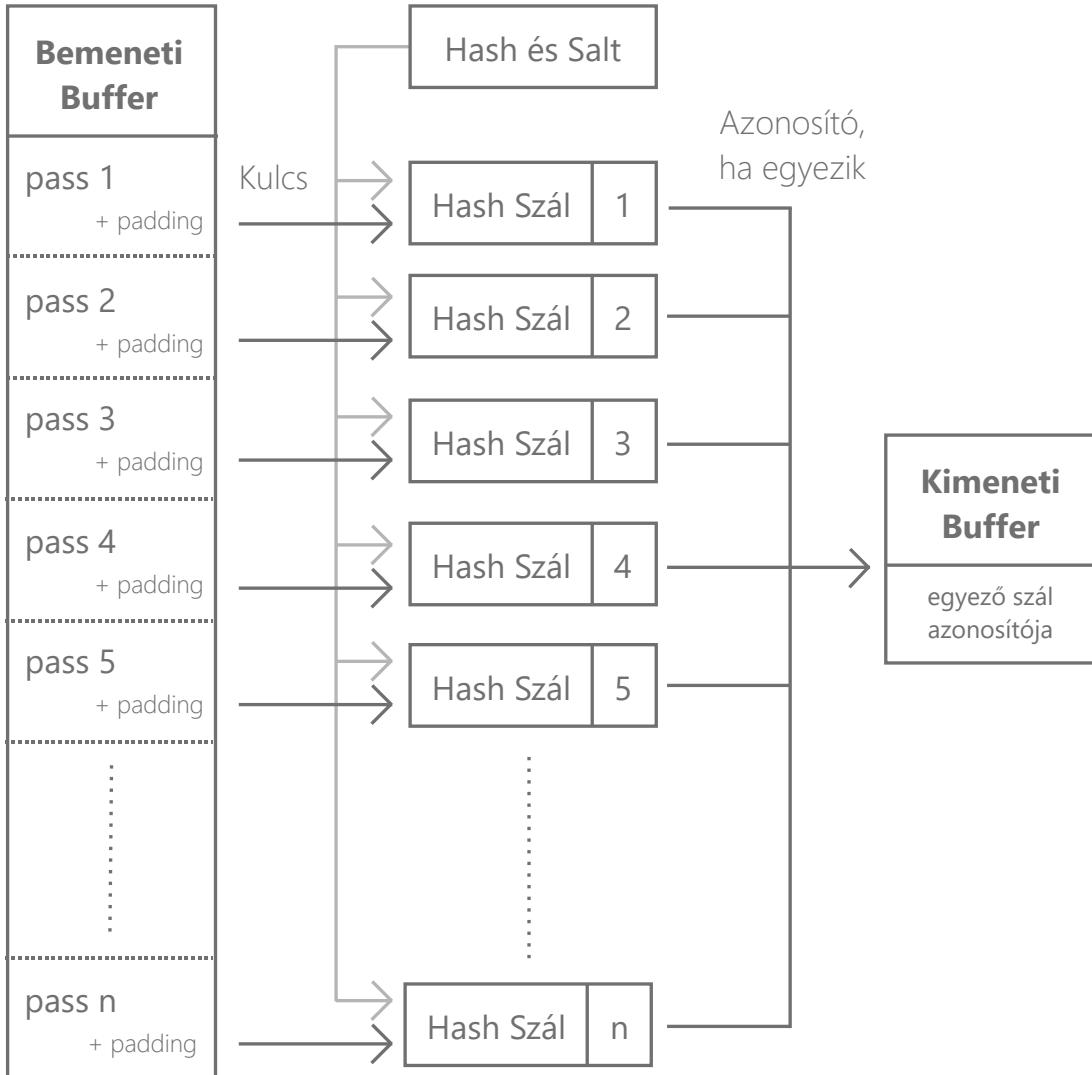
ekkor a jelenlegi szál:

- Bemenetének kezdete = $P_0 + (I * M) + (I * S)$
- Bemenetének vége = $P_0 + (I * M) + (I * S) + (M + S - 1)$
- Kimenete = P_1 , vagy nincs kimenet



3.5. ábra. Szálak párhuzamosan dolgoznak a bemeneti kulcs bufferen és a kimeneti hash bufferen.

Alapértelmezetten a kimenet bufferének értéke 0-ról indul. Ha a kimenet a futás után is nulla marad, akkor nem találtunk egyező kulcsot. Ezzel szemben amennyiben az érték megváltozott, akkor tudjuk, hogy az adott azonosítójú szál oldotta meg sikeresen a visszafejtést és az értéket ki tudjuk olvasni az előbb betöltött memória-területről.



3.6. ábra. Szálak párhuzamosan dolgoznak a bemeneti kulcs bufferen és a megadott hash és salt értékeken.

3.5. C++ Optimalizálás

Az optimalizáció tesztek futtatásához (amennyiben nincs egyéb meghatározva) egy asztali számítógépet használtam a következő paraméterekkel:

A fejlesztési idő jelentős részét az optimalizálás töltötte ki. Miután volt egy programom, amely képes volt jelszavakat beolvasni és feltörni processzoron és videókártyán is egyből tudtam tesztelni a sebességet. A teszteléshez minden alkalommal egy 4 millió (3 735 367) elemű listát használt a program, melynek az utolsó elemeként szerepelt a helyes kulcs (ex-wethouder). A futási idő méréséhez a C++ nyelv stan-

Megnevezés	Modell	Megjegyzés
Alaplap	ASUS Prime X470-PRO	
CPU	Ryzen 7 2700X 8c/16t 4.00Ghz	alap órajel
RAM	Corsair Vengeance 2x8GB 2400Mhz DDR4 dual channel	
GPU	Nvidia Geforce GTX 1070	alap órajel
SSD	Samsung 970 EVO 250GB	NVMe slot

3.4. táblázat. A tesztekhez használt számítógép paraméterei.

dard környezetében található chrono könyvtárat használtam, amely képes microsec pontossággal jelezni két utasítás között eltelt időt. A végleges időkben kizárolag a tényleges feltöréssel töltött idő szerepel, nem tartalmazza az elején az inicializálást és a fájl megnyitását, illetve a végén az eredmény kiíratását. Ezek ugyanis egyszer történnek csak meg, és tetszőlegesen nagy adattömeg esetén elenyésző a hatásuk. A program debug mód kikapcsolásával és x64 architektúrára van építve, illetve a /O2 fordító parancsal, amely többek között a kódoptimalizálást a futtatható fájl mérete helyett a sebességre fókuszálja.

10. Definíció. *Futási Stabilitás: Egy program futás időtartamának relatív eltérése több teszten keresztül azonos paraméterekkel, azonos hardveren és azonos alap kihasználtsággal.*

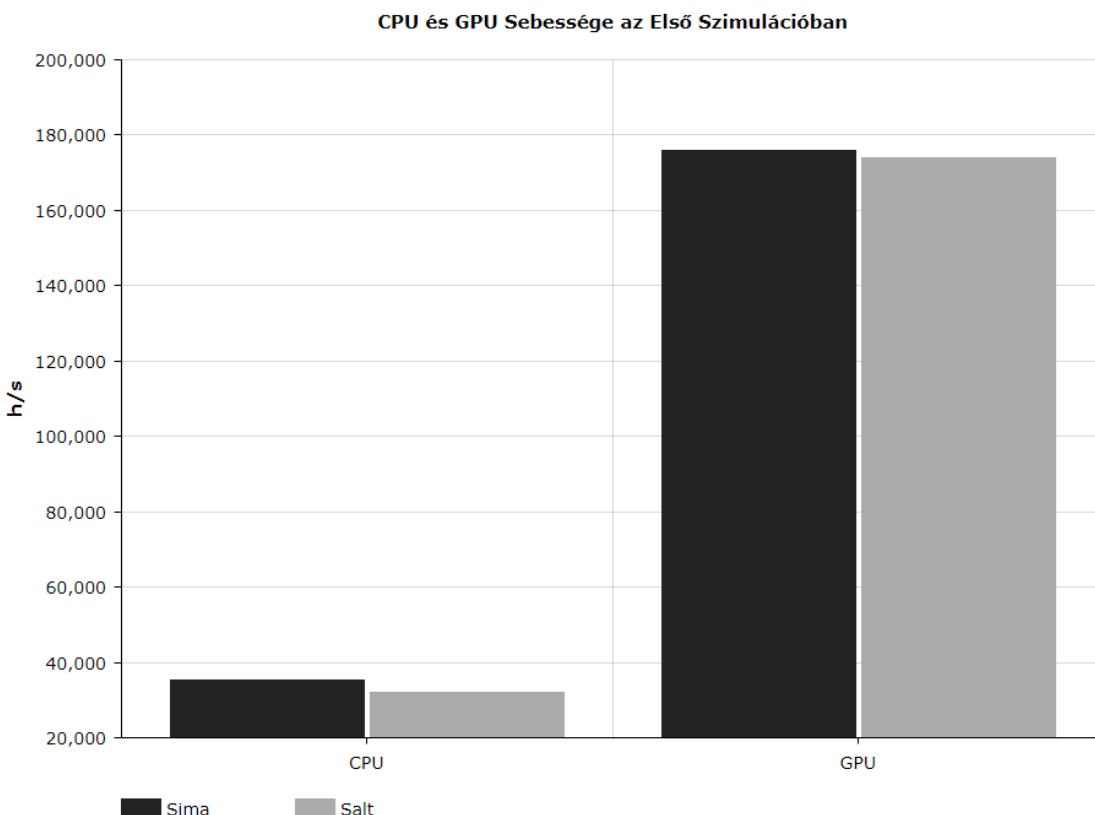
11. Definíció. *Hash per Second (H/s): Egy mértékegység, amely egy algoritmus egy másodperc alatt elkészíthető hash kódjainak számára, egy adott hash algoritmus használatával, azonos hardveren és azonos alap kihasználtsággal.*

3.5.1. Első Szimuláció

Az alap tesztet 20 alkalommal futtattam CPU és GPU használatával is. Ezáltal egyrészt tisztán láthatjuk az egymáshoz képest számolt sebessékgülönbséget, másról amennyiben a függvény futási ideje instabil, korrigálhatunk arra.

Eszköz	Salt	Futásidő	Teljesítmény
CPU	Nem	$105\,658\,927 \mu s \approx 106.7s$	$\approx 35\,353 H/s$
	Igen	$116\,019\,598 \mu s \approx 116.0s$	$\approx 32\,196 H/s$
GPU	Nem	$21\,250\,652 \mu s \approx 21.3s$	$\approx 175\,776 H/s$
	Igen	$21\,490\,658 \mu s \approx 21.5s$	$\approx 173\,814 H/s$

3.5. táblázat. Első szimuláció eredményei



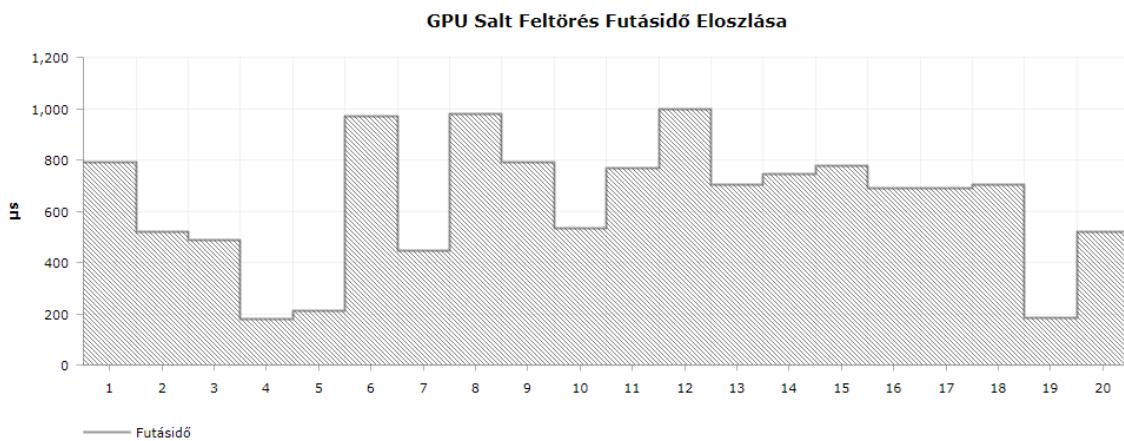
3.7. ábra. CPU és GPU alapesetben és salt-al történő összehasonlítása az első szimuláció során.

Látható a szimulációs eredményekből, hogy a GPU feltörés sebessége jelenleg a CPU megfelelőjének majdnem 550%-a. Továbbá megfigyelhető az is, hogy a hash alkalmazása nagyobb teljesítmény veszteséget okoz CPU-nál esetén (10%), mint GPU esetén 1%. Ez annak tudható be, hogy az utóbbinál a videókártyán történik a salt behelyezése a szöveg végére, így ez egy időben akár ezerszer is lefuthat.

A GPU-n salt használatával történő feltörés a projekt célja, ezért erre fókuszáltam futási stabilitás tesztelésénél. A teszt során 20 alkalommal futtattam a programot azonos körülmények között.

Iteráció	Futásidő	Iteráció	Futásidő
1.	$21\ 490\ 792\mu s$	11.	$21\ 490\ 768\mu s$
2.	$21\ 490\ 518\mu s$	12.	$21\ 490\ 999\mu s$
3.	$21\ 490\ 488\mu s$	13.	$21\ 490\ 703\mu s$
4.	$21\ 490\ 178\mu s$	14.	$21\ 490\ 744\mu s$
5.	$21\ 490\ 213\mu s$	15.	$21\ 490\ 776\mu s$
6.	$21\ 490\ 968\mu s$	16.	$21\ 490\ 688\mu s$
7.	$21\ 490\ 446\mu s$	17.	$21\ 490\ 688\mu s$
8.	$21\ 490\ 978\mu s$	18.	$21\ 490\ 702\mu s$
9.	$21\ 490\ 792\mu s$	19.	$21\ 490\ 182\mu s$
10.	$21\ 490\ 534\mu s$	20.	$21\ 490\ 518\mu s$

3.6. táblázat. Futási Stabilitás vizsgálatánál a tesztek futási eredményei



3.8. ábra. GPU Salt Feltörés Futásidő Eloszlása milliomod másodpercben számolva.

Ebből kiszámolható, hogy az adatok szórása (standard deviation) $\sigma = 240\mu s$. Ilyen kis számoknál a programnyelv belső órájának pontossága is közrejátszhat az inkonziszenciában, ezért a szórás a hibakorlátunk alatt helyezkedik el, tehát kijelenthetjük hogy a jelenlegi algoritmus futásideje stabil.

3.5.2. Double Buffer

Jelenleg az adatok beolvasása az 3.2 ábrának megfelelően zajlik, tehát megtörtenik egy adatszegmens beolvasása, amely továbbításra kerül a feltörésre használt eszköz számára majd az eredmény megérkezését követően elkezdődik a következő beolvasás. Ezen természetesen tudunk javítani a 3.3 ábrának megfelelően. Erre egy dupla bufferezéses megoldást alkalmaztam, amely esetén a beolvasott adatoknak két egyforma méretű buffer lett létrehozva. Ezek legyenek B_1, B_2, B_c, B_o (*current, other*)

Algoritmus 1 Double Buffer lépések

```

1:  $B_c := B_1, B_o := B_2$ 
2:  $read(B_c)$ 
3: while not eof() and not found() do
4:    $crack(B_c) \& read(B_o)$ 
5:    $swap(B_c, B_o)$ 
6: end while

```

A bufferek méretének kiválasztását nem lehet fordításnál eldönteni, hiszen attól függnek, hogy a feltörésre használt rendszer adatok olvasása vagy az eszköz sebessége a kisebb keresztmetszet. Ezért a bufferek méretének megválasztását a felhasználóra bízzuk, azonban a program mindenkorral választ magának egy alapértelmezett értéket, amennyiben egyéb utasítást nem kap.

Lépés	Futásidő	Teljesítmény	Különbség	Javítás
CPU	$116\ 019\ 598\ \mu s \approx 116.0s$	$\approx 32\ 196\ H/s$		
GPU 1	$21\ 490\ 658\ \mu s \approx 21.5s$	$\approx 173\ 814\ H/s$	+550.2%	
GPU 2	$20\ 201\ 236\ \mu s \approx 20.2s$	$\approx 190\ 322\ H/s$	+9.5%	Double Buffer

3.7. táblázat. Teljesítmény táblázat a double buffer hozzáadásával.

3.5.3. Preprocesszor Konstansok

A kulcs maximális mérete, a hash numerikus felbontása, a salt és annak a hossza már ismertek a kernel fordítása során és konstans értékek maradnak a teljes feltörés folyamata alatt. A just-in-time fordításnak köszönhetően ezeket az érékeket

beleépíthetjük a kódba preprocesszor direktívák segítségével, ezáltal segítve a fordítót az optimalizálásban és kevesebb adatmozgatás és buffer létrehozással. Ezeket az értékeket a fordítónak adjuk át plusz paraméterként.

```
1 -D DEFINED_STRING=somestring
```

3.6. forráskód. 32 bites egész szám tömb hexadecimális karaktersorozattá konvertálása.

Egy kisebb kihívást a szöveg beszúrása jelentette a salt esetén, ugyanis a preprocesszor nem támogatja ezt. Ezt egy makro használatával tudtam megoldani.

```
1 //String convert macro
2 #define STR(s) #s
3 #define XSTR(s) STR(s)
4
5 //...
6
7 const char* str = XSTR(DEFINED_STRING);
```

3.7. forráskód. Macro a fordítási paraméterként megkapott szöveg konvertálására.

Ez a konvertálás természetesen nem fog lassítani a futáson, hiszen megtörténik fordításnál.

Lépés	Futásidő	Teljesítmény	Különbség	Javítás
CPU	116 019 598 $\mu s \approx 116.0s$	$\approx 32\,196 H/s$		
GPU 1	21 490 658 $\mu s \approx 21.5s$	$\approx 173\,814 H/s$	+550.2%	
GPU 2	20 201 236 $\mu s \approx 20.2s$	$\approx 190\,322 H/s$	+9.5%	Double Buffer
GPU 3	20 112 915 $\mu s \approx 20.1s$	$\approx 191\,091 H/s$	+0.4%	Preprocessor

3.8. táblázat. Teljesítmény táblázat a preprocesszor definíciók alkalmazásával.

3.5.4. C Bemenet

Jelenleg a C++ eszközeit használom a fájl beolvasására. Ez a módszer std::string struktúrába másolja a fájl sorait, ahonnan később ki kell csomagolni és beleírni a bufferbe C alapú (null karakterrel lezárt) char* -ként.

```
1 std::ifstream infile(fileName);
2
```

```

3 // ...
4
5 std::string line;
6 for (int i = 0; i < chunkSize && std::getline(infile, line); i++)
7 {
8     strcpy(&currentBuffer[MAX_KEY_SIZE * i], line.c_str());
9 }
10
11 // ...
12
13 infile.close();

```

3.8. forráskód. Fájl sorainak beolvasása C++ fstream eszközökkel.

A példán az adatfolyam egy szegmensének beolvasása található. Ez a kód (5-9. sor) ismétlődik egészen addig, amíg elfogy a fájl, vagy az előző beolvasás feltörése sikeresen zárul. Látható hogy a beolvasás során a nyelv az adatokat egy std::fstream objektumon keresztül olvassa be, amelyet egy std::string-ben helyez el. Ezt a stringet végül egy null karakterrel terminált C string-re konvertáljuk és belemásoljuk a buffer megfelelő szegmensébe. Érezhető, hogy ezek felesleges extra műveletek, amelyek a futási idő egy jelentős részét jelenthetik.

Az optimalizáláshoz leváltottam a C++ nyelv által használt iostream és fstream eszközöket a standard C könyvtárra (stdio.h).

```

1 FILE* infile = fopen(fileName, "r");
2
3 // ...
4
5 for (int i = 0;
6     i < chunkSize && fgets(&currentBuffer[MAX_KEY_SIZE * i],
7         MAX_KEY_SIZE, infile) != NULL;
8     i++)
9 {
10 // ...
11
12 fclose(infile);

```

3.9. forráskód. Fájl sorainak beolvasása C++ fstream eszközökkel.

Ebben az esetben látszik, hogy a fájlból történő beolvasás azonnal a bufferbe helyezi a szöveget. Egy hátrány, hogy a szövegek nincsenek lezártva null karakterekkel, hanem a sor beolvasója behelyezi a sorok végén található sortörés karaktert. Ezt minden szövegnél ki kell javítani hogy ne tekintse a hash algoritmus az entert is a jelszó részének (hiszen azok nem tartalmazhatnak sortörés karaktert). Ezt azonban a crack-ot végző eszköz végzi, így módosítottam, hogy a hashelés előtt az eszköz ellenőrzi a szöveg pontos hosszát úgy, hogy null vagy enter karakterig iterál, majd a végére fűzi a salt-ot.

```

1 //Get key
2 uint length;
3 globalID = get_global_id(0);
4 globalKey = keys + globalID * KEY_LENGTH; //Get pointer to key
5 for (length = 0; length < KEY_LENGTH && (globalKey[length] != '\0',
6     && globalKey[length] != '\n'); length++)
7 {
8     key[length] = globalKey[length];
9 }
10 //Append salt
11 #pragma unroll
12 for (uint i = 0; i < SALT_LENGTH; i++)
13 {
14     key[length + i] = XSTR(SALT_STRING)[i];
15 }
16 length += SALT_LENGTH;
17 key[length] = 0;

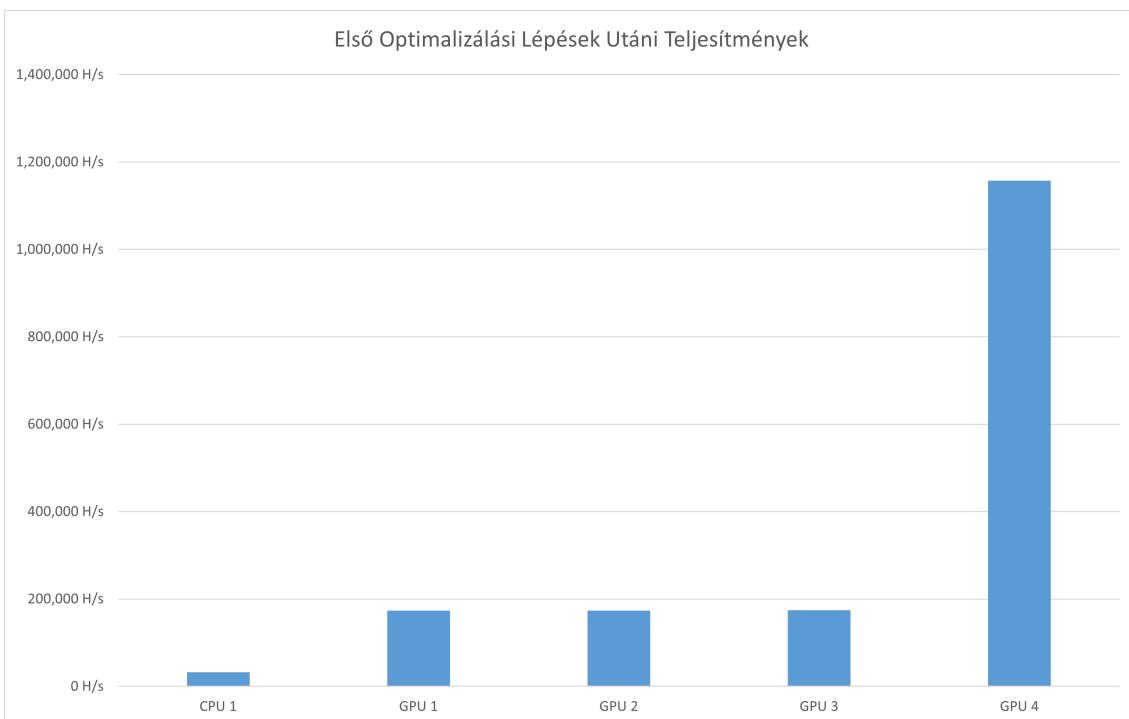
```

3.10. forráskód. Fájl sorainak beolvasása C++ fstream eszközökkel.

Látható, hogy a kulcs lokális memóriába történő másolása során a jelenlegi karakter hozzáadása előtt megnézzük, hogy a karakter null (\0), vagy sortörés-e. (\n). Amennyiben igen, a kulcs végére értünk és a length értéket nem növeljük tovább.

Lépés	Futásidő	Teljesítmény	Különbség	Javítás
CPU	$116\ 019\ 598\ \mu s \approx 116.0s$	$\approx 32\ 196\ H/s$		
GPU 1	$21\ 490\ 658\ \mu s \approx 21.5s$	$\approx 173\ 814\ H/s$	+550.2%	
GPU 2	$20\ 201\ 236\ \mu s \approx 20.2s$	$\approx 190\ 322\ H/s$	+9.5%	Double Buffer
GPU 3	$20\ 112\ 915\ \mu s \approx 20.1s$	$\approx 191\ 091\ H/s$	+0.4%	Preprocessor
GPU 4	$3\ 034\ 589\ \mu s \approx 3.0s$	$\approx 1\ 157\ 085\ H/s$	+665.7%	Cstdio

3.9. táblázat. Teljesítmény táblázat a C-s beolvasási módszerek alkalmazásával.



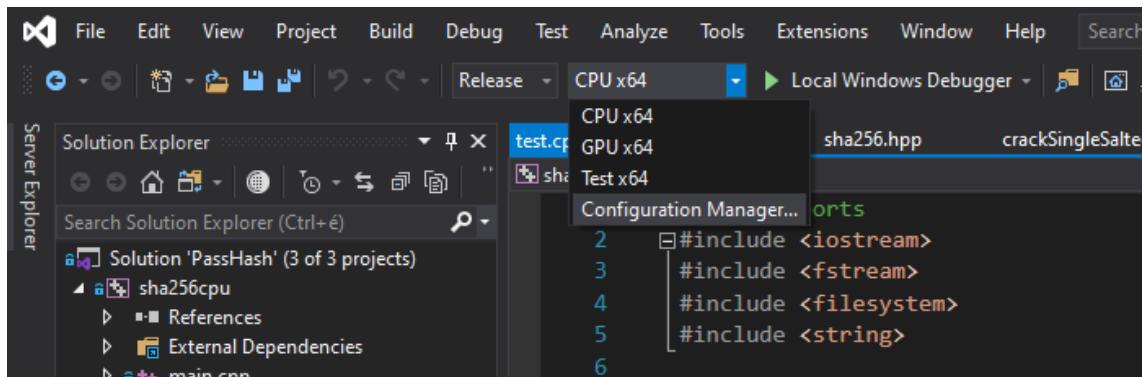
3.9. ábra. Az első optimalizálási lépések utáni teljesítmények összehasonlítása.

3.5.5. Fordító Beállítások

Eddig minden teszt x86 rendszerre lett fordítva Debug módban alap optimalizálási beállításokkal MSVC-ben. Ezen lehet javítani első sorban Release módra való átállással és egyéb fordító beállításokkal, amelyek a futási sebességet szem előtt tartva végeznek módosításokat az assembly-n. Ehhez átalakítottam a függőségi fát a GPGPU órán tanultakról és integráltam az OpenCL fájlokat a projektbe. Ez önmagában nem működőképes, mert az OpenCL headerek bizonyos deklarációi kizárólag

Debug fordítás esetén voltak elérhetőek. Ezeket a limitációkat eltávolítottam ezen műveletek inline metódusokká alakításával.

Ezek mellett le kellett töltenem az OpenCL előre fordított könyvtárának x64-es verzióját, amelyre a fordító és a linker megfelelő konfigurálásával sikerült átállítani a rendszert. A módok között a Visual Studio program-ban a fordítási beállításoknál lehet váltani.



3.10. ábra. A fordítási mód kiválasztása a Visual Studio-ban.

A megfelelő build módot kiválasztva a főmenüben található "Build" → "Build Solution" kombinációval indíthatjuk, amely kizárolag a meghatározott programot építi majd fel a Solution-ból.

Az architektúra választó mezőtől közvetlenül balra találjuk a fordítási mód beállítóját. Debug módban továbbra is bekerülnek az MSVC által definiált debug környezeti segítő metódusok és hook-ok. Ezzel szemben a release verziót a fordító a maximális optimalizálás érdekében a következő parancssal fordul:

```
1 MSVC_COMPILER_PATH\bin\HostX86\x64\CL.exe /c /I..\oclpack\include\
    /Zi /nologo /W3 /WX- /diagnostics:column /sdl /O2 /Ob2 /Oi /Ot /
    GT /GL /D WIN64 /D NDEBUG /D _CONSOLE /D _UNICODE /D UNICODE /Gm-
    - /EHsc /MD /GS /Gy /fp:precise /permissive- /Zc:wchar_t /Zc:
    forScope /Zc:inline /Fo".\build\obj\gpu-release\" /Fd".\build\
    obj\gpu-release\vc142.pdb" /Gd /TP /FC /errorReport:prompt
    crackSingle.cpp crackSingleSalted.cpp GPUController.cpp
    hashMultiple.cpp platformDetails.cpp hashSingle.cpp
    hashSingleSalted.cpp main.cpp
```

3.11. forráskód. A Release mód fordításának parancsa MSVC használatával.

Lépés	Futásidő	Teljesítmény	Különbség	Javítás
CPU	$116\ 019\ 598\ \mu s \approx 116.0s$	$\approx 32\ 196\ H/s$		
GPU 1	$21\ 490\ 658\ \mu s \approx 21.5s$	$\approx 173\ 814\ H/s$	+550.2%	
GPU 2	$20\ 201\ 236\ \mu s \approx 20.2s$	$\approx 190\ 322\ H/s$	+9.5%	Double Buffer
GPU 3	$20\ 112\ 915\ \mu s \approx 20.1s$	$\approx 191\ 091\ H/s$	+0.4%	Preprocessor
GPU 4	$3\ 034\ 589\ \mu s \approx 3.0s$	$\approx 1\ 157\ 085\ H/s$	+665.7%	Cstdio
GPU 5	$1\ 083\ 921\ \mu s \approx 1.0s$	$\approx 3\ 239\ 422\ H/s$	+279.9%	Release/x64

3.10. táblázat. Teljesítmény táblázat a release, az x64-os mód és a fordító optimalizálások bekapcsolásával.

3.5.6. Szálméret

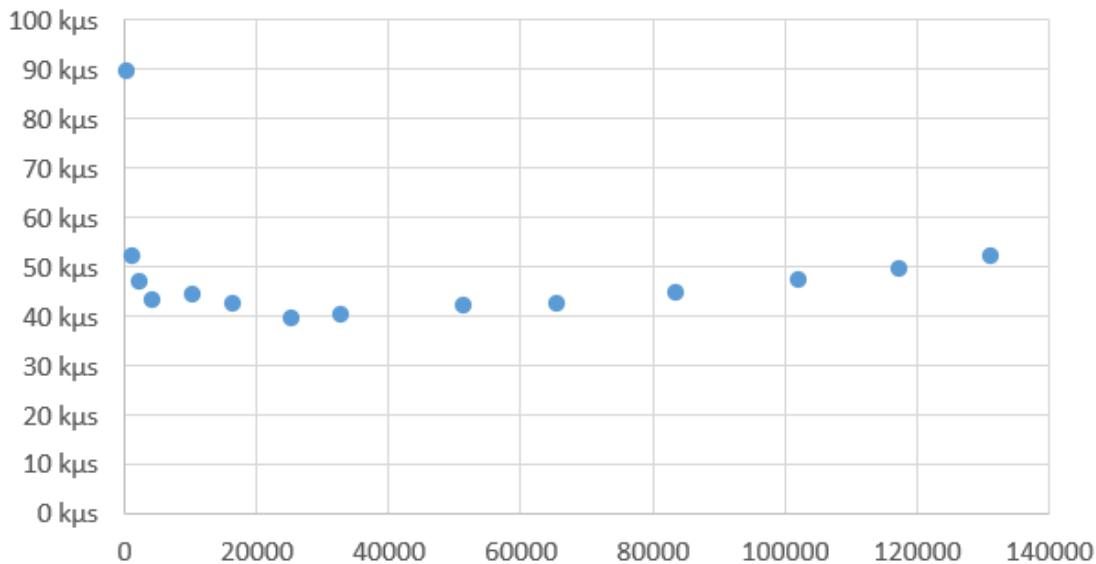
Ezen az optimalizálást nem számolom bele a végleges értékekbe, ugyanis az eddigiek nélkül is kevésbé konzisztens. Általánosan elmondható volt idáig, hogy a pontos teljesítmény nem fog megfelelni két rendszer között, azonban arányaiban megfelelhetető lesz, azaz egy kétszeres teljesítmény növelés nagyjából ugyan ekkora különbséggel jár majd minden rendszeren. A programnak megadható a -t kapcsoló használatával, hogy hány kulcsot töltön fel a memóriába feltörésre egy időben. Az optimális mennyisége sok faktortól függ és ami nálam jelentős teljesítmény növekedést eredményez, az egy másik rendszeren többszörös lassulást jelenthet. Ezért ahelyett hogy egy általános megoldást találnék a problémára, a felhasználónak meghagyom a lehetőséget hogy ő döntse el, vagy kísérletezze ki a megfelelő értéket. Az én rendszerem esetén a megfelelő érték nagyjából $t = 22\ 000$.

3.11 Az ábrán látható hogy $t = 5000$ alatt jelentősen csökken a teljesítmény, hiszen az idő egyre nagyobb részét teszi ki a bufferek mozgatása és a kernel elindítása. A teljesítménykülönbség az én rendszeremen $t = 1$ és $t = 22\ 000$ között 1102 szeres. $t = 22\ 000$ felett lassan romlik a teljesítmény.

3.6. OpenCL Optimalizálás

Az OpenCL oldalon is hasonlóan jelentős teljesítmény növelés érhető el a feltörő algoritmus refaktorálásával. Az eredeti kódban vannak felesleges kódok, amelyek

Futásidő eloszlása thread-size függvényében



3.11. ábra. Teljesítmény különbség szálméret arányában az én rendszeremen.

nem lesznek használva, illetve optimalizálatlan és egyszerűsíthető részek. Az egyik elsődleges szempont az elágazások eliminálása és balanszolása lesz, ugyanis ezeknél egy videókártyának minden lehetőségen végig kell mennie hogy ki tudja elégíteni a lehetőségek számát minden szálra.

3.6.1. Lokális Memória

A beérkező kulcsokat a globális memóriából kapjuk meg, ahol melléjük írjuk a salt-ot, majd onnan másoljuk át a kezdeti bufferbe. Ez azt eredményezi, hogy sok felesleges üres memória másolódik át, illetve többször a lassú globális területen dolgozunk. Erre megoldásként az eszközön létre kell hoznunk egy memóriaterületet, amely tárolja a kulcs és a hash összefűzött szöveget. A dinamikus memória foglalás azonban jelentősen lassítana a műveleten, ezért kihasználom azt, hogy előre ismerjük a kulcs maximális hosszát és a salt pontos méretét. Ennek köszönhetően létre lehet hozni előre minden szálon a szükséges területeket, amely minimális számítási költséggel jár. Az adatok átmásolásához végigiterál a program a kapott globális kulcson és átmásolja a karaktereket, majd mögéhelyezi a salt-ot.

```

1 //Local memory for the key (+1 for terminating null)
2 char key[KEY_LENGTH + SALT_LENGTH + 1];

```

```

3
4 //Get pointer to key string
5 char* globalKey = keys + get_global_id(0) * KEY_LENGTH;
6
7 //Copy key
8 for (length = 0; length < KEY_LENGTH && (globalKey[length] != 0 &&
9     globalKey[length] != '\n'); length++)
10 {
11     key[length] = globalKey[length];
12 }
13
14 //Append salt
15 #pragma unroll
16 for (uint i = 0; i < SALT_LENGTH; i++)
17 {
18     //Get salt from preprocessor definition
19     key[length + i] = XSTR(SALT_STRING)[i];
20 }
21
22 //Closing null
23 key[length] = 0;

```

3.12. forráskód. A kulcs lokális memóriába másolása az eszközön.

Mivel a jelszó nem tartalmazhat sortörés karaktert, ezért addig vizsgáljuk a szöveget, amíg nem találunk egyet vagy egy szövegzáró null-t.

3.6.2. Kulcs Konvertálása

A kulcsot a hasheléshez egy uint tömbbe kell konvertálnunk. Mivel minden karakter 8 bit, ezért pontosan 4 karakter fér bele egy integer-be. Így amíg négyvel egészket osztható a kulcs hossza, pontosan belefér egész mennyiségű int-be.

```

1 //Copy whole uints
2 qua = length / 4;
3 mod = length % 4;
4 for (uint i = 0; i < qua; i++)
5 {
6     W[i] = (key[i * 4 + 0]) << 24;

```

```

7     W[i] |= (key[i * 4 + 1]) << 16;
8     W[i] |= (key[i * 4 + 2]) << 8;
9     W[i] |= (key[i * 4 + 3]);
10 }

```

3.13. forráskód. A kulcs négygyel egészen osztható részének az uint tömbbe másolása.

Külön kell kezelni azonban amikor a kulcs hossza nem osztható n-el hiszen akkor az utolsó uint utolsó karaktere mögé egy 1-es bitet kell helyezni.

Erre egy gyors megoldást biztosít a biteltolás és a négy esetre bontás. Legyen: K_l a kulcs hossza és S_l a salt hossza. Ekkor a négy eset:

1. $K_l + S_l \equiv 0 \pmod{4} \iff n * 4 + 0 = K_l + S_l \ (n \geq 0)$
2. $K_l + S_l \equiv 1 \pmod{4} \iff n * 4 + 1 = K_l + S_l \ (n \geq 0)$
3. $K_l + S_l \equiv 2 \pmod{4} \iff n * 4 + 2 = K_l + S_l \ (n \geq 0)$
4. $K_l + S_l \equiv 3 \pmod{4} \iff n * 4 + 3 = K_l + S_l \ (n \geq 0)$

Ezekben az esetekben másként kell kezelni az utolsó 0-3 karaktert. A szöveg végén egy 1-es bitet kell elhelyezni ezek mellett, amely az előző modulus alapú felbontás alapján rendre:

\equiv	Bin	Hex
0	1000 0000 0000 0000 0000 0000 0000 0000	80000000
1	1000 0000 0000 0000 0000 0000 0000	800000
2	1000 0000 0000 0000	8000
3	1000 0000	80

3.11. táblázat. Az utolsó uint értékének a padding-je a mod érték alapján

```

1 //Pad remaining uint with leading 1 bit
2 switch (mod)
3 {
4     //l = n * 4 + 0
5     case 0:
6         W[qua] = 0x80000000;

```

```

7      break;

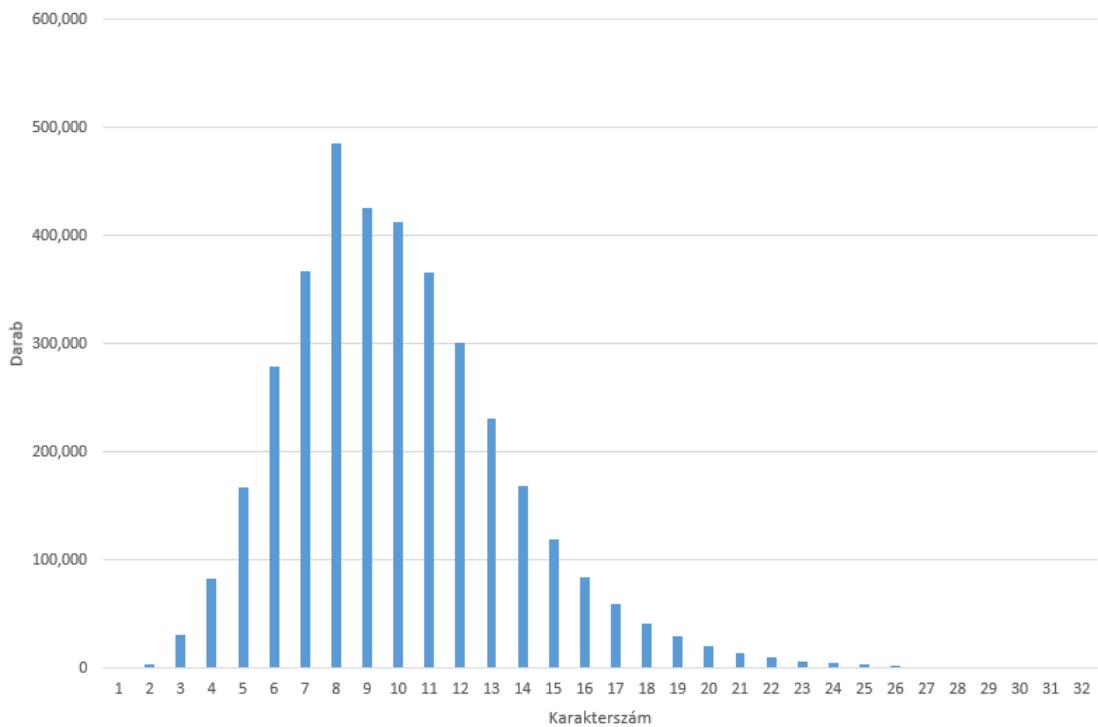
8

9      //l = n * 4 + 3
10
11     case 3:
12         W[qua] = (key[qua * 4 + 0]) << 24;
13         W[qua] |= (key[qua * 4 + 1]) << 16;
14         W[qua] |= (key[qua * 4 + 2]) << 8;
15         W[qua] |= 0x80;
16         break;
17
18     case 2:
19         W[qua] = (key[qua * 4 + 0]) << 24;
20         W[qua] |= (key[qua * 4 + 1]) << 16;
21         W[qua] |= 0x8000;
22         break;
23
24     case 1:
25         W[qua] = (key[qua * 4 + 0]) << 24;
26         W[qua] |= 0x800000;
27         break;
28 }
29
30
31 //Add key length
32 W[15] = length * 8;

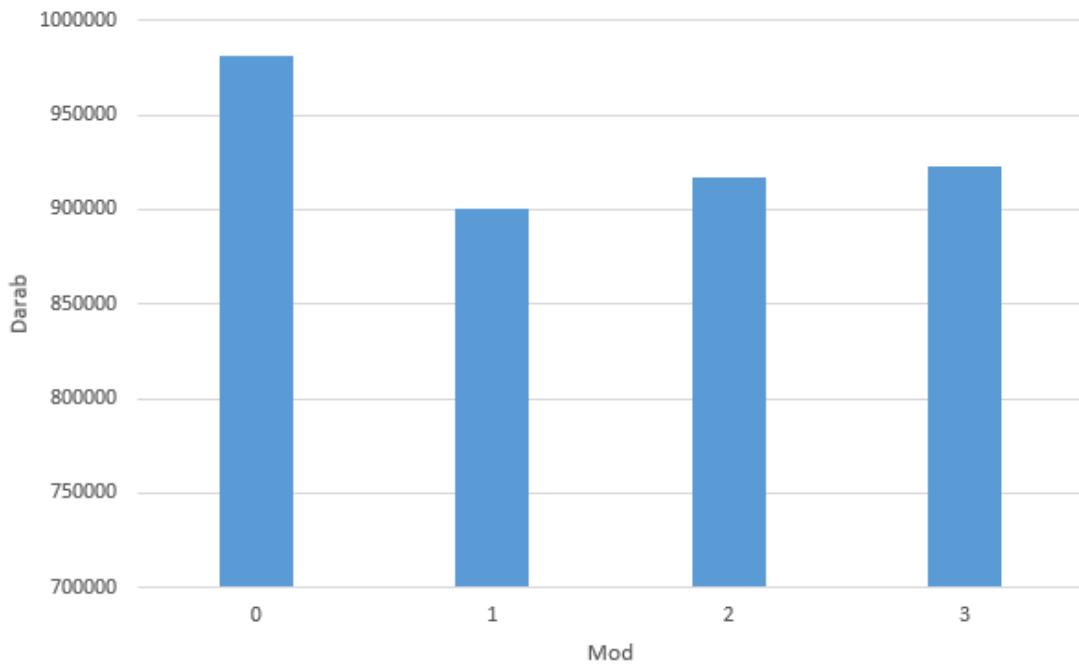
```

3.14. forráskód. A kulcs négygyel nem osztható részének az uint tömbbe másolása.

Megfigyelhető, hogy a feldolgozás sorrendje 0, 3, 2, 1 a megmaradó nem egész vagy üres uint esetén. Ennek az oka statisztikai gyakoriságból ered. Az általam használt majdnem 4 millió elemű gyakran használt listában a jelszavak adoptt hosszúságának a számát összeszámoltam.



3.12. ábra. Jelszavak hossza és gyakoriságuk majdnem 4 millió elemű listából.



3.13. ábra. Jelszavak hosszának gyakorisága mod 4-el számolva. A láthatóság kedvéért a grafikon 700 000-nél kezdődik.

A grafikonról látható, hogy a legtöbbször használt jelszavak hossza a mod 4 =

0 kategóriába esik. Ennek egy oka lehet, hogy sok weboldal megkövetel legalább 8 karaktert, és páran maximum 16 karaktert jelszónak.

Abban az esetben ha egy szál munkacsoportban (work group) minden elem például mod 0 és 1-re értékelődik ki, akkor nem szükséges a további két csoportot is végigszámolni. Annak érdekében hogy ez az eset a lehető leggyakrabban történjen meg, az elágazásokat a mod 4 sorrendjük alapján rendszereztem.

3.6.3. Kiterjesztés és Tömörítés

Az adatok konvertálása után a következő lépés az üzenet kiterjesztése majd tömörítése. Ezeket a legtöbb rendszerben ciklusok végezik, amelyen belül is elágazások találhatók bizonyos műveleteknek. Mivel csak egy blokkal dolgozunk, ezért ezeket statikusan is elvégezhetjük ciklusok kiterítésével. Ehhez szeparálni kell a különböző részeket és konstans hosszúságú ciklusként unroll-olni. Ennek a végeredménye egy különböztetett üzenetsor.

```

1 //Run message schedule
2 #pragma unroll
3 for (uint i = 16; i < 64; i++)
4 {
5     W[i] = sig1(W[i - 2]) + W[i - 7] + sig0(W[i - 15]) + W[i - 16];
6 }
```

3.15. forráskód. Az üzenetsor kiterjesztése a kezdeti kulcsból.

Jelen esetben a $\text{sig0}(x)$ és $\text{sig1}(x)$ függvények rendre a $\sigma_0(x)$ és $\sigma_1(x)$ függvényeknek felelnek meg. Az első 15 elem az üzenetet és annak hosszát tárolja, ezért a kiterjesztés 16-tól kezdődik.

Ennek a kódnak a kiemelésével a tömörírést szintén egy kiterített ciklusban tudjuk használni 3.16.

```

1 //Prepare compression
2 A = H0;
3 B = H1;
4 C = H2;
5 D = H3;
6 E = H4;
7 F = H5;
```

```

8 G = H6;
9 H = H7;
10
11
12 //Compress
13 #pragma unroll
14 for (uint i = 0; i < 64; i++)
15 {
16     //Alter instance
17     T1 = H + csig1(E) + ch(E, F, G) + K[i] + W[i];
18     T2 = csig0(A) + maj(A, B, C);
19
20     //Rotate over, override H
21     H = G;
22     G = F;
23     F = E;
24     E = D + T1;
25     D = C;
26     C = B;
27     B = A;
28     A = T1 + T2;
29 }

```

3.16. forráskód. Az üzenetsor tömörítése a kezdeti kulcsból.

Látható, hogy az unroll-ok használata miatt nincsen ciklus, sem elágazás a kód ezen részében. A kiterjesztés 5. sorában található kódot például egy időben kettesével is el lehet végezni, ugyanis nem befolyásolja az n . elem az $n+1$ -et. Ezzel szemben az adatok forgatása miatt továbbra is egyesével kell haladnunk a második kiterjesztett ciklusban.

3.6.4. Összehasonlítás

Az összehasonlításnál alap esetben az összes hashelt blokk elemeit összegezzük, majd ezeket összehasonlítjuk. Jelenleg mivel csak egy blokk lehetséges, ezért csak az első blokk hash értékét kell hozzáadni a kiindulási értékekhez és ezt összehasonlítani az előre ismert értékekkel.

A hashelés eredménye minden esetben egy 8 elemű uint tömb lesz, amit a kiíratáshoz hexadecimális karaktersorozattá alakítunk. Jelenleg felesleges ezt átalakítani, ezért a host oldalon előre kiszámoljuk a hash uint felírását és definiáljuk azt a kernelben fordítás előtt. Ezzel egyrészt nem lesz szükséges egy nagyobb karaktertömmbe átkonvertálni az uint elemeket, másrészt az összehasonlításnál 64 byte méretű terület összehasonlítása helyett 8 darab 32 bites területet hasonlítunk.

Bontsuk a hash-et 8 részre, ahol H a feltörésre szánt hash, h a jelenlegi blokkból számolt hash, D a kiindulási állapot és n a blokkok száma. Ekkor a hash-ek egyeznek, ha minden komponensük egyezik:

$$\begin{aligned} H_1 &= D_1 + \sum_{i=1}^n h_1^i, & H_5 &= D_5 + \sum_{i=1}^n h_5^i, \\ H_2 &= D_2 + \sum_{i=1}^n h_2^i, & H_6 &= D_6 + \sum_{i=1}^n h_6^i, \\ H_3 &= D_3 + \sum_{i=1}^n h_3^i, & H_7 &= D_7 + \sum_{i=1}^n h_7^i, \\ H_4 &= D_4 + \sum_{i=1}^n h_4^i, & H_8 &= D_8 + \sum_{i=1}^n h_8^i; \end{aligned}$$

A mi esetünkben kizárolag egy blokk lesz használva, ezért az egyenletek egyszerűsíthetőek.

$$\begin{aligned} H_1 &= D_1 + h_1^1, & H_5 &= D_5 + h_5^1, \\ H_2 &= D_2 + h_2^1, & H_6 &= D_6 + h_6^1, \\ H_3 &= D_3 + h_3^1, & H_7 &= D_7 + h_7^1, \\ H_4 &= D_4 + h_4^1, & H_8 &= D_8 + h_8^1; \end{aligned}$$

Ezen értékekből a kernel fordításának időpontjában ismert számunkra H és D egy konstans érték. Ezért ezen műveletek egy részét már a fordító elvégezheti, amennyiben átrendezzük a műveleteket. Akkor bizonyítottuk hogy a megadott kulcs meg-

egyezik a megkapott hash-el ha:

$$h_1^1 = H_1 - D_1 \wedge h_2^1 = H_2 - D_2 \wedge h_3^1 = H_3 - D_3 \wedge h_4^1 = H_4 - D_4 \wedge \\ h_5^1 = H_5 - D_5 \wedge h_6^1 = H_6 - D_6 \wedge h_7^1 = H_7 - D_7 \wedge h_8^1 = H_8 - D_8;$$

A programkódban a $(H_1 \dots H_8) \Rightarrow (\text{HASH_0} \dots \text{HASH_7})$, $(h_1 \dots h_8) \Rightarrow (\text{A} \dots \text{H})$, $(D_1 \dots D_8) \Rightarrow (\text{H0} \dots \text{H7})$. Mivel az egyenlőségjelek jobb oldalán található kivonás minden két eleme ismert fordítási időben preprocesszor definícióként, ezért a végeleges kódba annak a kiszámolt verziója kerül be. Ennek köszönhetően összesen 1..8 darab egész szám egyenlőségvizsgálat fog megtörténni. Amennyiben az adott szalon bizonyítottuk a kulcsok egyezését, a szál száma plusz egy belekerül az eredmény uint-be, amelyet később a c++ oldalon kiolvasunk. A szálak számozása 0-tól indul, azonban jelenleg a 0 érték (amelyet host-tól kapunk) azt fogja jelenti, hogy nem találtuk az értéket, ezért egyezés esetén a szál számához egyet hozzá kell adnia a kernelnek.

```

1 //Verify result
2 if (A == HASH_0 - H0 && B == HASH_1 - H1 &&
3     C == HASH_2 - H2 && D == HASH_3 - H3 &&
4     E == HASH_4 - H4 && F == HASH_5 - H5 &&
5     G == HASH_6 - H6 && H == HASH_7 - H7)
6 {
7     *result = globalID + 1;
8 }
```

3.17. forráskód. Egyszerűsített hash egyenlőség vizsgálata.

Lépés	Futásidő	Teljesítmény	Különbség	Javítás
CPU	$116\ 019\ 598\ \mu s \approx 116.0s$	$\approx 32\ 196\ H/s$		
GPU 1	$21\ 490\ 658\ \mu s \approx 21.5s$	$\approx 173\ 814\ H/s$	+550.2%	
GPU 2	$20\ 201\ 236\ \mu s \approx 20.2s$	$\approx 190\ 322\ H/s$	+9.5%	Double Buffer
GPU 3	$20\ 112\ 915\ \mu s \approx 20.1s$	$\approx 191\ 091\ H/s$	+0.4%	Preprocessor
GPU 4	$3\ 034\ 589\ \mu s \approx 3.0s$	$\approx 1\ 157\ 085\ H/s$	+665.7%	Cstdio
GPU 5	$1\ 083\ 921\ \mu s \approx 1.0s$	$\approx 3\ 239\ 422\ H/s$	+279.9%	Release/x64
GPU 5	$308\ 774\ \mu s \approx 0.3s$	$\approx 12\ 097\ 414\ H/s$	+373.4%	Kernel opt.

3.12. táblázat. Teljesítmény táblázat az OpenCL kernel optimalizálások
hozzáadásával.

Az eredmények alapján az én rendszeremen elérte végleges maximális teljesítmény 12 MH/s, amely egy jelentős 376 szoros előrelépés az első CPU verzióhoz képest és 70 szeres teljesítmény növekedés az első GPU verzióhoz képest.

4. fejezet

Tesztelés

4.1. Memóriaszivárgás

A C++ nyelv sajátosságai közé tartozik, hogy örökölte a C nyelvből a memóriaterületekre való mutatók használatát és azok lefoglalásának és felszabadításának manuális feladatát. Memóriaszivárgásról (memory leak) akkor beszélhetünk, ha egy adott memóriaterület lefoglalásra kerül a program futása során, azonban az sosem szabadul fel, vagy túl későn. Ez hibát okozhat egy dinamikusabb memóriakezelésű környezetben, ahol sokszor hozunk létre objektumokat majd távolítjuk el őket. Amennyiben a felhasználható memória elfogy, a program futása a következő memóriahívás létrehozásakor leáll, vagy a számítógépen a többi párhuzamosan futó program futását akadályozza.

Ezen program minimális memóriaszivárgási kockázattal jár, ugyanis fix bufferekkel dolgozik és azokat a futás első szakaszában egyszer hozza létre. Ennek ellenére amennyiben egy ilyen hiba megjelenik, az biztosan mutatja hogy valami nem megfelelően működik a belső rendszerben, ezért egy tökéletes hibakezelési módszer.

A memóriaszivárgásokat manuálisan megkeresni szinte lehetetlen, amennyiben elég kis területekről van szó, hogy ne látsszon a memóriahasználati grafikonon. Ezért egy Microsoft által fejlesztett könyvtárat alkalmaztam, amely a program futásának befejeztével kilistázza a nem felszabadított memóriaterületek címeit és azok tartalmát, amelyet segítséget nyújtanak azok megtalálásában.

A main fájlban kell importálni a könyvárat, majd a main futásának befejeztével kiiratni az esetleges hibákat.

```

1 //Memoryleak test
2 #ifdef _DEBUG
3     #define _CRTDBG_MAP_ALLOC
4     #include <stdlib.h>
5     #include <crtDBG.h>
6 #endif // _DEBUG
7
8 //Main
9 int main(int argc, char* argv[])
10 {
11     RelayResult result = MainCommandRelay::relay(argc, argv);
12
13 #ifdef _DEBUG
14     _CrtDumpMemoryLeaks();
15 #endif // _DEBUG
16
17     return static_cast<int>(result);
18 }
```

4.1. forráskód. Microsoft CRT könyvtár implementációja a rendszerben

Látható, hogy csak akkor importáljuk a könyvtárat, amikor debug üzemmódban fordítjuk a programot, illetve a kiiratás metódusa is csak ebben az esetben lesz meghívva.

```

1 Detected memory leaks!
2 Dumping objects ->
3 {18} normal block at 0x0178FE61, 32 bytes long.
4 Data: <REDACTED> 00 00 00 00 00 00 00 00
5 Object dump complete.
```

4.2. forráskód. Memóriaszivárgás üzenet példa

4.2. Egységesztelés

A program tesztelésének a fő módszere az egységesztelés (unit test) jelen esetben. Az egységeket a program lefutásának kis szakaszaira lehet alkalmazni, ezáltal kideríthető, hogy a hiba pontosan melyik egységből származik.

Az egységek tesztelése a Microsoft Visual Studio beépített eszközeivel történik, amelyet képesek futtatni az előre megírt tesztelő kódot és azok eredményét egy jól érthető és gyorsan áttekinthető felületen prezentálni. Erre egy külön teszt projekt lett létrehozva a program főkönyvtárában. Fontos, hogy a tesztelő helyes működéséhez a fordítási beállításoknak Debug módban kell lenniük.

A tesztek elkészítéséhez módosítanom kellett a függvények visszatérési értékén, ahogy azok megfelelően tesztelhetővé váljanak, az eredményeket ne csak a kimeneti konzolban jelenítsék meg. Ezen felül többször előfordult, hogy az OpenCL optimalizáció során hibás eredményeket generált a hash funkció, amelyeket az egységetesztnek sikerült jelezni.

```

1 //Load
2 GPUController* gc = new GPUController();
3 gc->attachDevice({});
4
5 //Hash
6 Assert::AreEqual(std::string(
    "b493d48364afe44d11c0165cf470a4164d1e2609911ef998be868d46ade3de4e
    "), gc->hashSingle("banana"));
7 Assert::AreEqual(std::string(
    "c79c99dded78b97103916e94e5bc052d0b881ad2da896674b177bda1b1830e35
    "), gc->hashSingle("encloses"));
8 Assert::AreEqual(std::string("9
    c9e82db146a9bfe73b43aebfa89cd889a4fccc6fe916a66dcd497ecc4c182a2"
    ), gc->hashSingle("enclosesHf45DD"));
9 Assert::AreEqual(std::string("59557
    cf1890bf0b7458c1e66119ab01c3a796fd09df296ef7e70745d29934777"),
    gc->hashSingle("ex-wethouder"));
10
11 //Length mods (4)
12 Assert::AreEqual(std::string(
    "ca978112ca1bbdcfac231b39a23dc4da786eff8147c4e72b9807785af0ee48bb
    "), gc->hashSingle("a")); //1
13 Assert::AreEqual(std::string("961
    b6dd3ede3cb8ecbaacbd68de040cd78eb2ed5889130cce4c49268ea4d506"),
    gc->hashSingle("aa")); //2
14 Assert::AreEqual(std::string("9834876
    dcfb05cb167a5c24953eba58c4ac89b1adf57f28f2f9d09af107ee8f0"), gc

```

```

15     ->hashSingle("aaa")); //3
15 Assert::AreEqual(std::string("61
16     be55a8e2f6b4e172338bddf184d6dbbe29c98853e0a0485ecee7f27b9af0b4")
16     , gc->hashSingle("aaaa")); //0
16 Assert::AreEqual(std::string("
17     ed968e840d10d2d313a870bc131a4e2c311d7ad09bdf32b3418147221f51a6e2
17     ") , gc->hashSingle("aaaaa")); //1
18 delete gc;

```

4.3. forráskód. A GPUController hashSingle metódusának tesztelése.

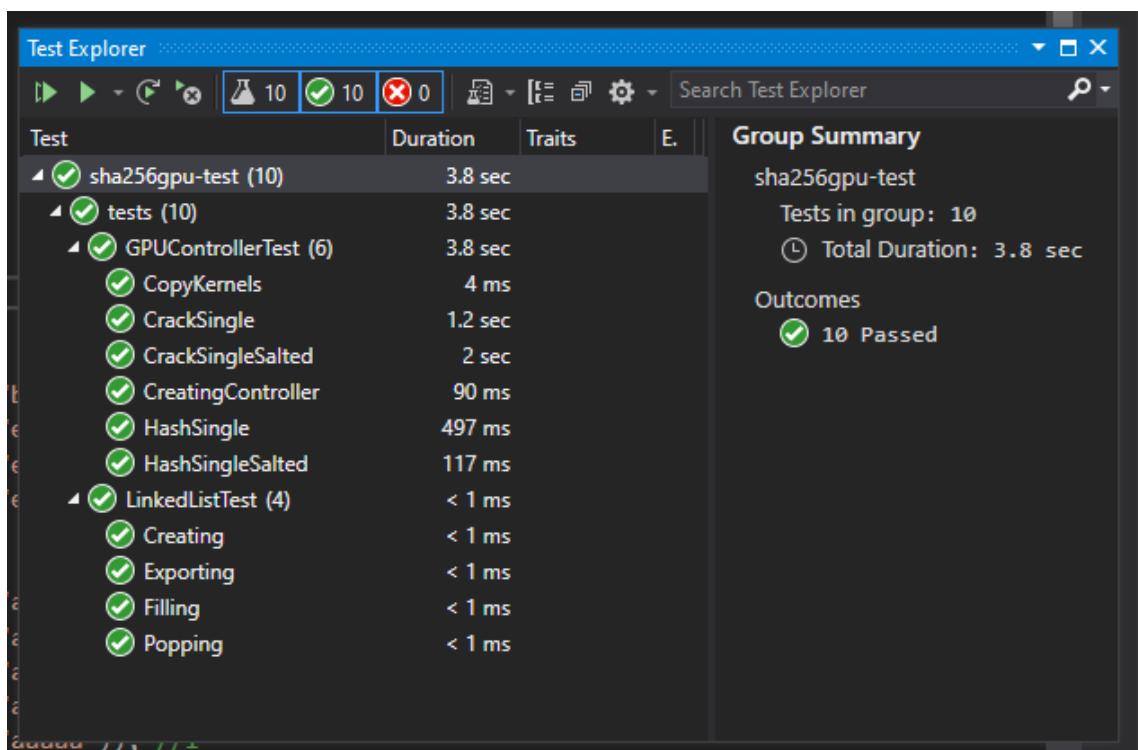
Látható, hogy a teszt első lépésében létrehozunk egy GPUController objektumot és rakkörök a rendszer alapértelmezett eszközére. Ezután elvégzünk egy néhány előre ismert hashelést és ellenőrizzük az eredményüket. A második részben tesztelünk különböző karakterhosszúságú sorokat, hiszen a program más ágon megy keresztül amennyiben ezek hossza maradékosan osztva 4-vel különbözik.

Ez a teszt lépés a GPU oldalon épp használt hashelés helyességére fókuszál, amely különösen előnyösnek bizonyult az eszköz oldali optimalizáció közben. A többi teszt is hasonlóképp épül fel, amelyek a program működésének különböző elemeire fókuszálnak rá.

- GPUController
 - **CopyKernels** - minden kernelkód létezik-e és a friss verziók átmásolódnak a tesztelő mappájába.
 - **Creating Controller** - A controller helyesen létrejön-e és rá tud-e csatlakozni az alapértelmezett eszközre.
 - **HashSingle** - Egy jelszó hashelés helyes-e.
 - **HashSingleSalted** - Egy jelszó és salt hashelés helyes-e.
 - **CrackSingle** - Egy jelszó feltörése helyes-e egy rövid jelszófájl használatával.
 - **CrackSingleSalted** - Egy jelszó salt-os feltörése sikeres-e egy rövid jelszófájl használatával. Azt hogy a megadott hash salt-al van ellátva a helyesen program állapítja-e meg.

- `LinkedList`
 - **Creating** - Helyesen elkészül-e egy példány.
 - **Filling** - Helyesen hozzáadhatók-e elemek.
 - **Popping** - Helyesen eltávolíthatók-e elemek.
 - **Exporting** - Helyesen exportálhatók-e az elemek egy `std::vector`-ba.

A beépített tesztelő felületet a Visual Studio főablakának főmenüjében a Test > Test Explorer gombbal érhetjük el.



4.1. ábra. A tesztek példa futtatása.

4.3. Nagy Adattömeg

A program egyik fő szempontja volt a skálázhatóság, ezért jól kell teljesítenie bár-mekkora adattömegben, lineáris teljesítményromlással. A tesztelést a 4 millió soros adatfolyam másolásával végeztem, ahol minden sor betűit véletlenszerűen összekevertem és a fájl mögé illesztettem 100-szor, minden alkalommal újra összekeverve. Ezáltal egy majdnem 400 000 000 elemű jelszó fájl jött létre, melynek lemezen elfoglalt mérete 4.367 GB.

Hibahatárnak nagyjából 5%-ot jelöltem ki, hiszen a számítógép sosem neutrális állapotban van és ezek befolyásolhatják a futási időt. Ennek a futási ideje 98.18 szorosa volt az eredeti majdnem 4 millió elemű fájlénak, amely hibahatáron belül helyezkedik el, így látható, hogy a program futási ideje lineárisan növekszik nagyobb adatfolyamok feldolgozása esetén.

5. fejezet

Konklúzió

5.1. Összegzés

Elkészítettem C++-ban egy SHA256 jelszó hash feltörő programot, amely a konfiguráció feldolgozása után elkészít egy OpenCL kernelt és lefordítja azt az adott rendszerre optimalizálva. Ezek után elkezdi a jelszó feltörését egy megkapott gyakran használt jelszavakat tartalmazó tetszőlegesen nagy fájl használatával. A program képes megkülönböztetni salt-al ellátott és alap SHA256 hash-eket.

Emellett a futtatásnál kapcsolókkal beállítható a használt eszköz, az egyszerre feltörenő adatok mennyisége és a maximális kulcsméret. minden egyes parancs egyértelműen és olvashatóan meghatározza a feladatot, amely elindítása után a program részletes leírást ad a felhasználónak a futás jelenlegi állapotáról.

5.2. Fejlesztési Lehetőségek

A program egy használható és szükségesen személyre szabható formában van, számos érdekes fejlesztési lehetőséget tartogat. Felsoroltam néhány lehetőséget, a sorrend jelentőségteljessége nélkül.

- Automatikus sebesség-optimalizálás és konfiguráció. Egy további parancs (pl. „-t” kapcsoló) egy referencia jelszó fájl használatával variálja a beállításokat, amíg szükségesen megközelíti a leggyorsabb konfigurációt az adott rendszerre. A kapott konfigurációt mint ajánlást átadja a felhasználónak, aki további finomhangolást eszközölhet.

- Egy felhasználói felület elkészítése, ahol a felhasználó távolról figyelheti és irányíthatja a program működését. Pl. egy websocket kliensen keresztül böngészőből, tetszőleges eszközről.
- Több számítási eszköz egyidejű használata. Ez több-videókártyás szerverek esetén lehet hasznos.
- Jelszó metamorfizmus támogatása. minden, az adatbázisban lévő jelszón kissébb változtatásokat kipróbálni, amelyeket emberek gyakran használnak a jelszavak bonyolítására. Például (i) betű (!)-re cserélése, kötőjel szóköz helyett, vagy születési dátum jelszó mögé helyezése.
- Több hash feltörése egy időben.

6. fejezet

Köszönetnyilvánítás

Szeretném megköszönni témavezetőmnek, Eichhardt Ivánnak a sok segítséget és ötletet, amit a dolgozat elkészítése alatt nyújtott.

Hálás vagyok az Eötvös Lóránd Tudományegyetem Informatikai Karának és minden tanárnak aki engem tanított az évek során, a remek felkészítésért amit kaptam és a sok fejlődési lehetőségért mind informatika, mind matematika terén.

Köszönnettel tartozom továbbá Daniel Miessler biztonságtechnikai szakembernek, akik publikussá tette a leggyakrabban használt jelszavak listáit GitHub oldalán.

Ábrák jegyzéke

2.1. Use-case diagram.	16
3.1. A program UML osztálydiagramja.	22
3.2. Adatmozgazás egy fájl tartalmának hashelése során.	34
3.3. Adatmozgazás parallel módon egy fájl tartalmának hashelése során. .	35
3.4. Az adatmozgatás és futtatás szekvenciadiagramja feltörés során. . .	36
3.5. Szálak párhuzamosan dolgoznak a bemeneti kulcs bufferen és a kimeneti hash bufferen.	38
3.6. Szálak párhuzamosan dolgoznak a bemeneti kulcs bufferen és a megadott hash és salt értékeken.	39
3.7. CPU és GPU alapesetben és salt-al történő összehasonlítása az első szimuláció során.	41
3.8. GPU Salt Feltörés Futásidő Eloszlása milliomod másodpercben számolva.	42
3.9. Az első optimalizálási lépések utáni teljesítmények összehasonlítása. .	47
3.10. A fordítási mód kiválasztása a Visual Studio-ban.	48
3.11. Teljesítmény különbség szálméret arányában az én rendszeremen. . .	50
3.12. Jelszavak hossza és gyakoriságuk majdnem 4 millió elemű listából. . .	54
3.13. Jelszavak hosszának gyakorisága mod 4-el számolva. A láthatóság kedvéért a grafikon 700 000-nél kezdődik.	54
4.1. A tesztek példa futtatása.	64

Irodalomjegyzék

- [1] Jonathan Katz és Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.
- [2] Claire Ellis. “Exploring the Enigma”. <http://plus.maths.org/content/os/isue34/features/ellis/index>. Acesso em 5 (2005), 1939–1945. old.
- [3] Quynh H Dang. “Secure hash standard”. (2015).
- [4] Bart Preneel. “Analysis and design of cryptographic hash functions”. Dissz. Katholieke Universiteit te Leuven, 1993.
- [5] Shay Gueron, Simon Johnson és Jesse Walker. “SHA-512/256”. *2011 Eighth International Conference on Information Technology: New Generations*. IEEE. 2011, 354–358. old.
- [6] Puru Joshi Ankit Kumar Jain Rohit Jones. “Survey of Cryptographic Hashing Algorithms for Message Signing”. <http://www.ijcst.com/vol8/8.2/3-rohit-jones.pdf> (2017), 22. old.
- [7] Yindong Chen, Liping Li és Ziran Chen. “An approach to verifying data integrity for cloud storage”. *2017 13th International Conference on Computational Intelligence and Security (CIS)*. IEEE. 2017, 582–585. old.
- [8] Kerry E Dungan és Lee C Potter. “Classifying sets of attributed scattering centers using a hash coded database”. *Algorithms for Synthetic Aperture Radar Imagery XVII*. 7699. köt. International Society for Optics és Photonics. 2010, 76990Q.
- [9] Simson Garfinkel és tsai. “Using purpose-built functions and block hashes to enable small block and sub-file forensics”. *digital investigation* 7 (2010), S13–S23.

- [10] Kurt Guntheroth. *Optimized C++: proven techniques for heightened performance.* " O'Reilly Media, Inc.", 2016.
- [11] Nicolas T Courtois, Marek Grajek és Rahul Naik. "Optimizing sha256 in bitco-in mining". *International Conference on Cryptography and Security Systems.* Springer. 2014, 131–144. old.
- [12] Xinxin Mei és Xiaowen Chu. "Dissecting GPU memory hierarchy through microbenchmarking". *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2016), 72–86. old.
- [13] Aaftab Munshi és tsai. *OpenCL programming guide.* Pearson Education, 2011.