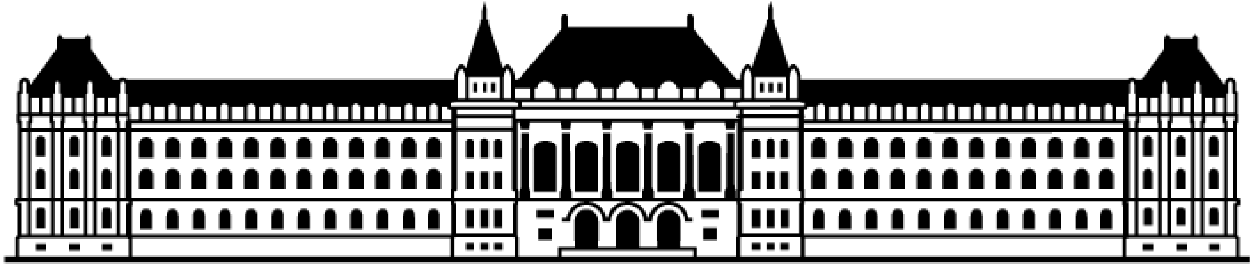


BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Hatékony mintakeresés kártékony programok azonosítására grafikus kártyán

Készítette:

Frendl Péter

Konzulens:

Dr. Dudás Ákos

Tudományos Diákköri Konferencia

2015 Október

Tartalomjegyzék

Összefoglaló **4**

Abstract **5**

1 Bevezetés **6**

2 Alapismeretek **8**

2.1 Vírus detektálási technikák 8

2.2 A mai GPU-k általános architektúrája 10

3 Kapcsolódó munkák **13**

4 A javasolt algoritmus **16**

4.1 Inspiráció 16

4.2 Az algoritmus megtervezése 16

4.2.1 Az alap algoritmus 17

4.2.2 Egyedi minták 22

4.2.3 Mintaillesztés bináris kereséssel 23

4.2.4 Mintaillesztés MPHF-el 25

4.2.5 Minta egyezések minimalizálása 29

4.2.6 A GPU és CPU feldolgozás párhuzamosítása 29

4.3 Rosszindulatú adatok elleni védelem 30

4.3.1 Véletlen mintavételezés 30

4.3.2 Rövidzár osztályozás 31

4.3.3 Nem kívánt minták elkerülése 33

5 Implementáció **34**

5.1 Az implementációs környezet 34

5.2 A szignatúra adatbázis 35

5.3 Device-Host memóriatranzakciók minimalizálása 35

5.4 Pinned memory használata 36

6 Mérések **38**

6.1 A tesztkörnyezet 38

6.2 Feldolgozási sebesség 39

6.2.1 Ofszetszám 39

6.2.2 Adatméret ... 41
6.3 CPU és GPU összehasonlítása ... 43
7 Konklúzió ... 47
Irodalomjegyzék ... 48

Összefoglaló

A grafikus hardverek fejlődésével az eredetileg grafikus feladatok ellátására tervezett vezérlőkártyák, vagy röviden GPU-k (Graphics Processor Unit) egyre nagyobb teret nyernek általános felhasználási területeken is. Ezen erősen párhuzamos eszközök egyre szélesebb körű felhasználásnak örvendenek számos tudományterületen, például a bioinformatikában (DNS-szekvenálás, protein folding), a gyógyszerkutatásban (molekuláris szimulációk) és a pénzügyekben (kockázatelemzés). A GPU-k erőforrásigényes algoritmusok körében való térhódítása jelentős, és a párhuzamos működés irányába való törekvés a központi feldolgozó egységek, vagy röviden CPU-k (Central Processing Unit) fejlődését követve is igen szembetűnő.

Az informatikai technológiák fejlődésével és a számítógépes hálózatok gyors elterjedésével az elmúlt évtizedekben fontos feladattá vált az ezen technológiákkal együtt megjelenő rosszindulatú programokkal szembeni védelem. A hálózati sávszélesség és a háttértárak gyors kapacitás-növekedésével egyre fontosabbá válnak az olyan eszközök, amelyek nagy mennyiségű adat gyors ellenőrzésére képesek.

Adódik a kérdés, hogy alkalmas-e a GPU a kártevő programokkal szembeni effektív védelemre. A GPU-k a jó tulajdonságaik mellett architekturális sajátosságaik miatt új kihívásokat rejtenek magukban implementációs szempontból, így a kérdés megválaszolása speciális algoritmus megtervezését, implementációját és tesztelését igényli.

A dolgozatomban egy olyan szignatúra alapú víruskereső algoritmust mutatok be, amely masszívan párhuzamos működéséből adódóan képes a GPU erőforrásainak magasfokú kihasználására, és ezáltal a víruskeresés hardveres gyorsítására.

Abstract

With the advancement of graphics hardware, graphics processor units (or GPUs) are becoming better suited for general purpose computing. These massively parallel computing devices are used in a wide array of applications, for example in bioinformatics (DNA-sequencing, protein folding), medicine (molecular dynamics) and finances (risk analysis). GPUs are becoming a prevalent solution for tackling time intensive computing problems, and a transition to high degree parallelism can be observed in the CPU (Central Processing Unit) industry, too.

With the advancement of information technology and the rapid expansion of computer networks, protection against malware (malicious software) has become an increasingly important task. With the fast increase of network bandwidth and data storage space, scanner applications capable of high data throughput are becoming extremely important.

The question emerges: is the GPU suited to provide effective protection against malicious software? GPUs, while providing tremendous computing capabilities, also introduce a new architecture, thus requiring new angles of approach for the efficient utilization of their capabilities. To answer this question, the specialized implementation of an effective GPU-specific algorithm is required.

This paper presents a signature-based detection algorithm, which – thanks to its construction – is capable of harnessing the massive parallelism of the GPU, thus hardware accelerating malware detection.

1 Bevezetés

Az elektronikus adatok védelme, és ezáltal a kártékony programok elleni védelem az internet megjelenése óta az informatika egyik legfontosabb területe. Ez a helyzet az elektronikusan tárolt adatok mennyiségének robbanásszerű növekedésével egyre csak súlyosbodik, és ezen adatok egyre nagyobb arányban tárolódnak felhő rendszerekben, amelyek napról napra támadásoknak vannak kitéve.

A kártevő programokkal szembeni védekezés hagyományosan a központi feldolgozó egységek feladata, és a hálózati végpontokban mindenképpen (de sok esetben akár a hálózati csomópontokban is) megtörténik. Az adatok hálózaton keresztül történő továbbítása sokszor például titkosított csatornán történik, amelyek az adatokat a továbbított formában nehezen ellenőrizhetővé teszik. Emiatt a dekódolt adatok ellenőrzése a cél csomópontokban mindenképp kulcsfontosságú.

Bár bizonyos víruskeresési módszerekhez szükséges lehet egy CPU komplexitása, a feldolgozandó adatmennyiség hatalmas, így felmerül a kérdés, hogy a központi feldolgozóegység helyett az ellenőrzés más hardver eszközökön is elvégezhető-e. Olyan hardverre van szükségünk, amely nagy adatmennyiség, akár párhuzamos feldolgozására is alkalmas, és nem igényel nagy beruházást.

A grafikus feldolgozó egységek a hagyományos feladataiknak köszönhetően teljesen más szögből közelítették meg az adatfeldolgozást, mint a CPU-k. A grafikus műveletek esetében több adaton kell többnyire azonos műveleteket elvégezni, így a hangsúly az adat párhuzamos feldolgozására esett. Az így kialakult architektúra a SIMD: Single Instruction, Multiple Data. Ennek keretében a GPU szálak csoportosan hajtják végre a feladataikat, szálanként csupán a kezelt adatok különböznek. Ez az architektúra jól párhuzamosítható algoritmus esetében a CPU-n elért sebességnek a több tízszeresét biztosíthatja.

A kártékony programok azonosítása komplex, többlépcsős feladat, így teljes egészében nem végezhető a GPU-n eredményesen. A dolgozatom célja e bonyolult feladatrendszer egy olyan elemének a kiragadása, amely alkalmas arra, hogy párhuzamosítva végezzük, és így megfelelő a GPU-n való futtatásra. Jó választás esetén jelentős gyorsulást tapasztalhatunk a CPU-n elérhető

sebességgel szemben, és hatékony megoldást találhatunk nagy mennyiségű adat gyors ellenőrzésére.

A dolgozatom további részének felépítése a következő. A második fejezetben bevezetem a használt GPU architektúráról való tudnivalókat, valamint a rendelkezésre álló fő víruskeresési módszereket. Ezen előismeretek feltétlen szükségesek a későbbi fejezetek megértéséhez és az algoritmus fejlesztése során hozott döntések elméleti alátámasztásához. A harmadik fejezetben egyéb elért eredményeket ismertetek a témában. A negyedik fejezetben az általam kifejlesztett algoritmust mutatom be kezdeti állapotában, valamint sorban az egyes elméleti megfontolásból bevezetett, majd tesztelt optimalizációkat követve. Ezt az ötödik fejezetben az algoritmus implementációjának tárgyalása követi, ahol bemutatom a folyamat során felfedezett optimalizációs lehetőségeket is. A hatodik fejezetben ismertetem a hardveresen gyorsított keresési algoritmus sebességét és összevetem azt a teljességében CPU-n futó változatéval. A hetedik fejezetben értékelem az elért eredményeket, majd zárásként a jövőbeli felhasználási és fejlesztési lehetőségeket tárgyalom.

2 Alapismeretek

Mielőtt belekezdenénk a GPU által gyorsított algoritmusok tárgyalásába, fontos, hogy megismerjük a ma rendelkezésre álló vírusdetektálási technikákat, és a GPU-k általános architektúráját. Ebben a fejezetben ezek bemutatása következik.

2.1 Vírus detektálási technikák

A kártevő programok első megjelenése óta a készítőik számos módszert találtak a szoftverek idegen rendszerekbe való bejuttatására. A védekezés fő eszköze a detektálás, azaz felismerés, amelyet a kártékony program eltávolítása követ. Dolgozatomban a detektálással foglalkozom, amely feladatot különböző eszközökkel, technikákkal lehetséges megvalósítani. A leggyakrabban használt módszerek a következők [1]:

- Szignatúra alapú detekció,
- Heurisztikák,
- Viselkedés alapú detekció,
- Sandbox detekció,
- Adatbányászati technikák.

Szignatúra alapú detekció: Az egyik legelső módszer, ami káros programok azonosítására képes. A technika lényege az ismert malware bájtkódjának eltárolása. Jövőbeli, potenciálisan veszélyes fájlokra a víruskereső szoftver ellenőrzi, hogy tartalmazzák-e az eltárolt bájtkódot. Amennyiben ez, valamint egyéb feltételek (például a vírusnak megfelelő fájl kiterjesztése, fájlbeli pozíció) teljesül, a szóban forgó fájl fertőzöttnek minősül. A bájtkódnak általában csak egy részét tároljuk, ezt a kártékony program szignatúrájának nevezzük, és innen származik a technika elnevezése. Amennyiben a szoftver támogatja, a szignatúrákban szerepelhetnek úgynevezett joker karakterek, amelyek funkciójukban a reguláris kifejezésekben szereplő speciális karakterekhez hasonlíthatók.

Heurisztikák: A módszer alapja a vizsgált fájl különböző kritériumok szerinti osztályzása, és a potenciális veszélyességének mérlegelése. Ilyenek lehetnek például, hogy mi a kiterjesztése, milyen utasításokat tartalmaz, és honnan származik (pl. internet). A szignatúra alapú detekcióval

ellentétben ez a módszer alkalmas eddig nem ismert vírusok detektálására is, amennyiben a kérdéses fájl vagy program rendelkezik a vizsgált ismertető jegyekkel.

Viselkedés alapú detekció: A heurisztikus analízishez hasonló, azonban ez a módszer a futó program működését és az az által hagyott nyomokat figyeli, ebből következtet a program esetleges veszélyességére. Ez a módszer igen hasznos az olyan programokkal szemben, amelyek titkosítják a saját bájtkódjukat, ezzel nehézkessé téve az ellenőrzést. A technika a polimorf [11] és metamorf [12] programok felismerésére is alkalmas lehet. A polimorf és metamorf programok terjedéskor logikailag ekvivalens módon megváltoztatják a saját bájtkódjukat, így a szignatúra és heurisztika alapú detekció alkalmatlan a felismerésükre. A két típus abban különbözik egymástól, hogy a polimorf programokban az átalakító motor érintetlen marad, míg metamorf esetben maga a motor is módosul. Az ilyen típusú programok verziói logikailag invariánsak, azaz az azonos típusú kártevők különböző példányai futásának mellékhatása azonos. Ez a detekciós módszer az önmagukat titkosító programokkal szemben is hatásos, hiszen a programok a bájtkódjuk titkosításának feloldásával kerülhetnek futtatható állapotba. A memóriába feloldott állapotban töltődnek be, azaz a futásuk során azonosíthatóak. Ennek a technikának a velejárója, hogy a kártékony program detekciója csak akkor lehetséges, ha a program már megkezdte a károkozást.

Sandbox detekció: A módszer lényegében egyezik a viselkedés alapú detekcióval, azonban ez esetben a vizsgált program egy virtuális környezetben van futtatva. Ebből az a jelentős előny származik, hogy a vizsgáló rendszer sértetlen marad a kártékony programok működésének ellenére is.

Adatbányászati technikák: A módszer lényege nagymennyiségű adatból való minták kinyerése, majd ezen minták detekcióra való felhasználása, például tanuló algoritmusok segítségével.

A fenti módszerek tág kategóriákat képeznek, ezeken belül is számos algoritmus és megoldás létezik a módszerek realizálására. Amennyiben ezen módszereket a GPU-n kívánjuk megvalósítani, azt kell vizsgálnunk, hogy milyen mértékben képezhetők az egyes megoldások sok, egymástól függetlenül futtatható, azonos futási szerkezetű szálra. Olyan algoritmusokat keresünk, amelyek szálai különböző adatokon azonos feladatokat végeznek. Az elágazások és feltételes ugrások kerülendők, és fontos a warp-on belüli memóriaelérés lokalitásának megőrzése.

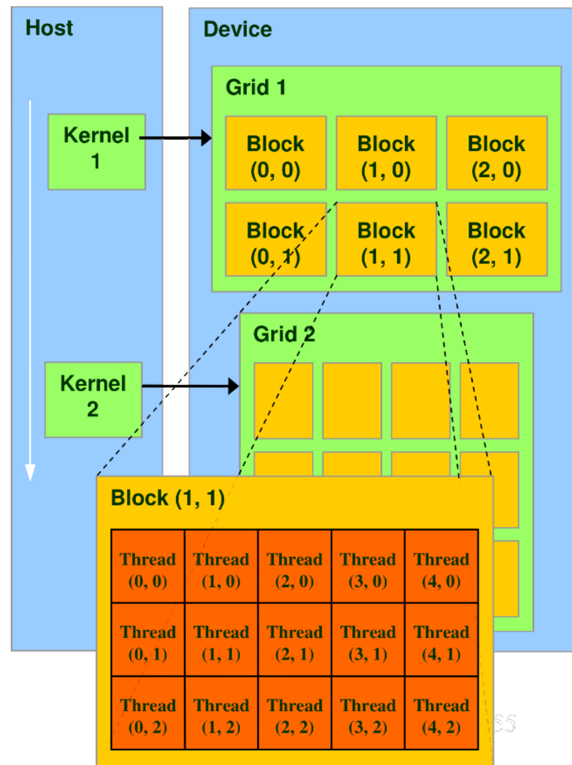
A fenti megoldások közül egyből szembetűnik, hogy a heurisztikus megoldások a feltételek használata miatt elágazásokban gazdagok, így nehézkesen valósíthatók meg a grafikus kártyán. A viselkedés alapú detekció, bár hatásos a ravaszabb vírusok felismerésére, bonyolult működés megvalósítását kívánja, és ez megint csak nem kívánt tulajdonság, amennyiben a feladatot sok egyforma egyszerű részfeladatra akarjuk osztani. A sandbox detekció csak tovább ront a helyzeten, így a további két megfontolandó módszer a szignatúra alapú detekció és az adatbányászatra épülő megoldások maradnak. Mindkét módszer alkalmaz olyan algoritmusokat, amelyek jól párhuzamosíthatók. Ezek közül ez a dolgozat a szignatúra alapú detekció GPU-val történő hardveres gyorsítását tárgyalja.

2.2 A mai GPU-k általános architektúrája

A grafikus kártyák a CPU-k MIMD (Multiple Instruction, Multiple Data) architektúrájával szemben a SIMD (Simple Instruction, Multiple Data) jellegű architektúra felett működnek. Ez azt jelenti, hogy az egyes szálak egy időben nem (illetve a gyakorlatban csak korlátozottan) képesek más-más utasítások végrehajtására. Az általam ezúttal alkalmazott platform a CUDA (Compute Unified Device Architecture), amelyen keresztül az NVIDIA grafikus kártyák programozhatók. Hasonló működés reprodukálható különböző shader nyelvek (pl. GLSL, HLSL) alkalmazásával, vagy OpenCL-el, amely heterogén feldolgozó egységeket futtató rendszerek programozásához fejlesztett eszköz.

CUDA környezetben párhuzamosan több szálon végrehajtott SIMD művelet logikailag egy úgynevezett grid-hez köthető. Egy grid blokkokból épül fel, a blokkok építőelemei pedig maguk a szálak, a párhuzamos végrehajtás atomi egységei. A szálak warp-okba csoportosulnak. Egy warp 16 vagy 32 szálat tartalmaz, a videokártya típusától függően. Az egy warpban levő szálak mindig csak azonos utasítást hajthatnak végre ugyanazon órajel alatt.

Az 1. ábrán is látható módon, a blokkokban a szálak logikailag 1, 2 vagy 3 dimenzióba szervezhetők. Az egy blokkon belüli szálak száma korlátos, így az egyszerre futó blokkok számát az határozza meg, hogy hány szálat akarunk futtatni a GPU-n.



Ábra 1 – A CUDA kernel felépítése

A warpok működéséből adódóan a feltételes szerkezetek körültekintően használandóak a GPU-n futtatott kódokban. Mivel a szálak különböző adatokon dolgoznak, elágazások (feltételes ugrások) esetén más-más szálaknál különböző értékekre értékelődhet ki a feltétel. Ha egy warpon belül nem minden szál ugyanarra az ágra kerül, akkor az ágak ennél a warpnál szekvenciálisan futnak, hiszen az egy warpon belüli szálak egyszerre csak ugyanazt az utasítást hajthatják végre. Körültekintéssel használandók emiatt `az if/else`, a `switch`, a rövidzár kiértékelés, és a `goto` még inkább kerülendő.

Az egy warpon belüli működés nem csak nem csak az utasítás végrehajtás, hanem memóriaelérés terén is összehangolandó. A GPU-n számos memóriatípus van különböző feladatok végrehajtására. A legáltalánosabb memória a globális memória, amely méretre a legnagyobb (GB nagyságrend), írható, és ezen keresztül végezhető kommunikáció a GPU-n (más néven device) futtatandó kód (kernel kód) és a CPU (avagy host) logika között. Az egy warpon belüli memóriaeléréseket a GPU minél kevesebb memóriaeléréssel próbálja megoldani a felhasznált DRAM sávszélesség csökkentésének érdekében. Ez azt jelenti, hogy az egy warpon belüli szálak egy időben, ideális esetben azonos vagy egymáshoz közeli memóriacímeket vizsgálnak.

A fentiekből adódóan fontos, hogy az elvégzendő feladatot elegendő mennyiségű, szerkezetileg egyforma és egymástól független részfeladatra bontsuk fel, az adatelérések lokalitását is észben tartva. Ezt az elvet a későbbiekben bemutatott saját implementációjú algoritmusomban is figyelembe fogom venni.

3 Kapcsolódó munkák

A grafikus hardveren történő általános célú programozás viszonylag új tudományterületnek számít, azonban több megközelítésből is léteznek már eredmények a témában. Ebben a fejezetben ezen eredményeket fogom röviden bemutatni.

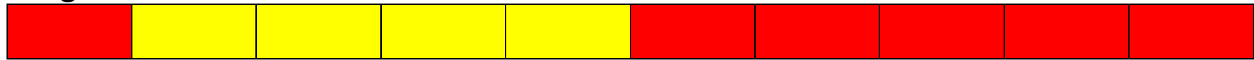
Hasznos olvasmány ebben a témában E. Seamans és T. Alexander munkája [2], amely a GPU-t, mint egy szűrőt használja fel a CPU-n végzendő ellenőrzések csökkentésének céljával.

Az alapötlet az, hogy a szignatúra adatbázis tulajdonképpen egy szótár, aminek a bejegyzéseit keressük az ellenőrzött adatokban, például fájlokban. Ha a fájl összes lehetséges ofszetjére ellenőrizzük, hogy valamely szignatúra illeszkedik-e oda, akkor minden egyezést regisztrálhatunk, így ez egy jó megoldás a kártékony fájlok azonosítására. A keresést úgy végezzük, hogy kiválasztunk egy mintahosszt, és minden szignatúrából veszünk egy ekkora mintát olyan módon, hogy minden minta egyedi legyen. Vegyünk egy példát 10 bájtos szignatúrákkal és 4 bájtos mintákkal.

szignatúra 1

minta 1

szignatúra 2



minta 2

feltétel: minta 1 ≠ minta 2

Ezeket a mintákat elküldjük a GPU-nak. Ezután az ellenőrzendő adatokat is átküldjük a GPU-nak, ami párhuzamosított módon dolgozza fel azokat. A GPU-n egy szálnak egy ofszetet osztunk ki az adatból ellenőrzésre. A szál az ofszetjére minden szignatúra mintát megpróbál illeszteni, és ha van találat, menti azt a GPU memóriájába.

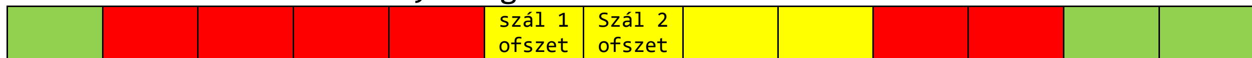
adatbuffer a GPU memóriában

szál 1 a saját pozícióján illeszteni próbálja minta 1-et és minta 2-t

közben szál 2 is ezt teszi a saját pozícióján

Az eredményt a CPU visszaolvassa, majd minden találat esetében megpróbálja illeszteni a vizsgált adatra a találathoz tartozó helyen a mintának megfelelő teljes szignatúrát (a hamis pozitív találatok kiszűrése érdekében). Amennyiben az illesztés sikerül, a fájl fertőzöttnek minősül.

adatbuffer a rendszermemóriában
minta 1 találat a GPU-n, szignatúra 1 illesztési kísérlet a CPU-n



találati ofszet

E. Seamans és T. Alexander [2] a szignatúra adatbázist a ClamAV [18] nyílt forráskódú vírusírtóból nyerték ki, és a teszteléshez is ezt a szoftvert használták. Az eredeti szignatúra detekciós algoritmust kicserélték a sajátjukkal, és az így kapott programot összehasonlították az eredeti ClamAV-val. Mindkét változat vírusadatbázisát korlátozták, hogy csak teljes szignatúrákat tartalmazzon. A módosított verzióval 8 MB-os fájlon 27-szeres gyorsulást értek el.

A munka három limitációt említ: maximum 64 000 szignatúrára tud egy futás alatt ellenőrizni, minden szignatúrából egyedi 6 bájtos mintára van szükség, és csak fix szignatúrákat támogat az algoritmus, azaz nincsenek joker karakterek.

A. N. V. Dias [3] megoldása szálanként egy fájlt ellenőriz a GPU-n, és állapotgépeket használ a szignatúrák illesztéséhez. Itt is, az előző megoldáshoz hasonlóan nem teljes szignatúrák, hanem az ezekből származó minták illesztéséről beszélhetünk. A GPU-n történő szűrés találatait az algoritmus a ClamAV-nak továbbítja ellenőrzésre. A ClamAV rendszerhez képest bizonyos esetekben 28-szoros gyorsulást mértek. Az említett limitációk közé tartozik, hogy a nagy fájlok lassítják a keresést, a hamis pozitívokat bizonyos esetekben nehéz feloldani, és a szignatúra adatbázis változásakor újra kell kalkulálni az automatákat, ami hosszú időt vesz igénybe.

Nem csak a szignatúra maga, hanem a kártékony programfájl hash értéke is alkalmas a szinatúra azonosítására. N. S. Kovach az MD5 hash-ek GPU-n történő gyors generálásával kísérelte meg gyorsítani a szkennelési folyamatot [4], és fájlmérettől függően 82-85%-os gyorsulást ér el a CPU-val szemben. A vizsgált fájlok méretben 192 kB-ra korlátozódnak.

A fentiekből látható, hogy a GPU-n történő víruskeresés ígéretes eredményeket mutat a teljesítmény szempontjából. Az is látható azonban, hogy egyelőre csak kísérleti megoldások születtek, ezért a problémát közelebbről is érdemes megvizsgálni.

4 A javasolt algoritmus

A szignatúra alapú detektálás egyszerű ötletre épül, azonban a hatékony megvalósítás nem triviális. Ebben a fejezetben bemutatom, az egyszerűbb algoritmusból milyen lépéseken keresztül juthatunk el a hatékony megvalósításig.

4.1 Inspiráció

Az algoritmust a [2] cikkben leírt, szignatúra alapú detekciót végző algoritmus inspirálta. Ebből a mintavétel alapú GPU-oldali szűrést és a szálak közötti munkamegosztás ötletét vettem át. Mivel a mintaméret minden szignatúránál egyezik, minden szál egyforma munkaadagot kap. E mellett az egy szál – egy ofszet munkamegosztás lehetővé teszi, hogy az egy warpban elhelyezkedő szálaknak szomszédos memóriaterületeket ellenőrizzenek. A megközelítésnek ez a két jó tulajdonsága ösztönzött arra, hogy egy ezen ötletekre alapuló alternatív algoritmust tervezzek, ami a mellett, hogy gyors, minél kevesebb limitációval rendelkezik.

4.2 Az algoritmus megtervezése

A detekcióhoz tehát szignatúrákra, illetve azokból származó mintákra van szükség. Ezt egy előzetes lépésben tudjuk előállítani a szignatúrák mintavételezésével. (Ezt a szignatúrahalmazunk változásakor kell legközelebb megismételnünk.)

A detekciót a GPU és a CPU együtt végzi. A GPU feladata a lehetséges találatok gyors megkeresése, melyet utána a CPU validál. A teljes folyamat a következő lépésekben zajlik.

1. Beolvassuk az adatokat egy adatbufferbe.
2. A beolvasott adatokat átmásoljuk a GPU memóriába.
3. Futtatjuk a GPU oldali szűrést végző kernel kódot (GPU-n futó kód).
4. Az eredményeket a GPU memóriából a rendszermemóriába olvassuk.
5. A mintaillesztéssel nyert találatokat validáljuk a mintákhoz tartozó teljes szignatúrákkal.
6. A szignatúraillesztésekkel nyert találatokat elmentjük, ezek tényleges pozitívak.

Az utolsó lépésben említett tényleges pozitív jelző itt a mintaillesztéssel kapott pozitívsághoz képest relatívan értendő. Mivel a szignatúra adatbázisokban a tárolt szignatúrák a hozzájuk tartozó káros programok bájtkódjának legtöbbször csak töredékei, hamis pozitívak természetesen előfordulhatnak abban az esetben is, ha teljes szignatúra illeszkedik a vizsgált adatra.

A következőkben bemutatom az alap algoritmust és annak optimalizált változatait.

4.2.1 Az alap algoritmus

Az algoritmus tehát szignatúra detekcióval azonosítja az általa ismert, azaz a szignatúra adatbázisában szereplő szignatúrákhoz tartozó vírusokat. Ennek hardveres gyorsításához egyenlő hosszú mintákat vesz a szignatúrákból, és ezeket keresi a vizsgált adatokban párhuzamosan. A CPU-nak ez után már csak az így kapott találatokat kell ellenőriznie, e miatt gyorsul a feldolgozás.

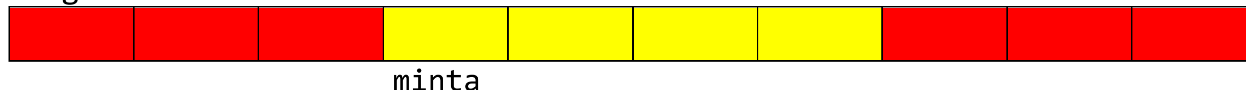
A vett minták mérete nagyban megszabja a GPU-n futó algoritmus pontosságát (minél hosszabb a minta, annál kisebb az esélye a fals pozitívnak), viszont rontja a sebességet (hiszen annál hosszabb egyezést kell vizsgálni). A kernelen a kis mintamérettel által nyert idő a CPU-n elvész, hiszen a fals pozitívok miatt sok munka hárul a CPU-ra, így nem cél a mintaméret minimalizálása. Mégis érdemes az értéket kellően alacsonyra választani, hiszen nagy minták esetén előfordulhat, hogy egyetlen minta egyezés vizsgálata sokkal tovább tart, mint a többié, és ez hosszan fenntartja a GPU-t, ezzel csökkentve az adatáteresztő képességét. Megoldásom a használt adatbázis legrövidebb szignatúrájának hosszát választja mintaméretnek. Ez általában pár bájtot jelent, tehát ennél kisebb mintával nem érdemes dolgozni, és egyben felső határ is a mintahosszra, hiszen a legrövidebb szignatúrából nem vehetünk nála nagyobb mintát.

Amennyiben nagy fájlokat is sikeresen szeretnénk kezelni, ezeket nem egészben, hanem részletekben kell beolvasnunk és ellenőriznünk. Egy részről a GPU kernel hívások nagy overhead-del rendelkeznek (~20μs), így előnyös egyszerre minél több adatot tárolni a memóriában. Másrészt a számítógépekben rendelkezésre álló rendszermemória korlátos, így ezt csak mértékkel tehetjük. Ez még inkább jellemző a grafikus kártyákra, ahol általában kevesebb a rendelkezésre álló memória (általában 1-4 GB, és nincs virtuális memória támogatás, azaz a fizikai méret egyben felső korlát is). Az adatokat tartalmazó buffer méretének meghatározása így fontos lépés. Kezdjük tehát az algoritmus definiálását a GPU memória és a rendszermemória között egyszerre mozgatott adatmennyiség meghatározásával.

A vizsgált adatszeletnek legalább akkorának kell lennie, hogy a vizsgált szignatúrák elférjenek benne. Míg a GPU-n mintaillesztést végzünk, a CPU teljes szignatúrákat illeszt. A minták szignatúrákból származó kivonatok, így kisebb méretűek. Ebből fakad, hogy a GPU-nak a vizsgált adat egyetlen pontjának, azaz ofszetjének az ellenőrzéséhez kevesebb adatra van szüksége, mint a CPU-nak.

Vegyünk egy példát. Álljon a rendelkezésünkre egy 10 bájtos szignatúra, valamint ebből egy 4 bájtos minta a 3. bájt ofszettől.

szignatúra



minta

A GPU-n így egy fájlban két bájtpozíció ellenőrzéséhez 5 bájtra van szükség.

adat

szál 1 ellenőriz minden szignatúra mintára

szál 2 ellenőriz minden szignatúra mintára

A fenti két művelet párhuzamosan zajlik.

A GPU-n `n` darab szomszédos ofszet ellenőrzéséhez `n + mintaméret – 1` méretű adatra van szükség. Így az utolsó ofszeten még épp rendelkezésre áll egy mintányi adat, ahogy az a fenti ábrákon is látszik. Ebből adódik a GPU adatbuffer szükséges mérete `n` szomszédos ofszet ellenőrzéséhez.

Minden találati adat legyen a talált minta helyének indexe a minták tárolójában.

A GPU-val szemben a CPU-n több kontextusra van szükség a teljes szignatúrákkal való tesztelés érdekében.

teljes adat

minta 1 találat a GPU-n, szignatúra 1 ellenőrzése a CPU-n

találati ofszet

Ahogy az a fenti ábrán is látszik, egy minta találat teljes szignatúrával való ellenőrzéséhez szükség van a találati ofszet körüli akkora adat-kontextusra, amelyben a szignatúra elfér. Ennek a paraméterei a szignatúra méretétől és a szignatúra mintavételezési ofszetétől függenek. Szükség van a találati pont előtt a mintavételi ofszetnek megfelelő mennyiségű bájtra, valamint a találati ponttól számítva még a szignatúraméret és a szignatúra mintavételezési ofszetje különbségének megfelelő számú bájtra.

`n` darab szomszédos ofszet esetén annyi adatra van szükség a szignatúraillesztéshez, hogy bármely ofszeten bármelyik szignatúra minta illeszkedése esetén a CPU illeszteni tudja ezen pontra a mintához tartozó teljes szignatúrát a szignatúra mintavételi pontja mentén. Ezen adatmennyiség kiszámolásához vizsgáljuk meg az összes szignatúránál, hogy hol van bennük a mintavételi pont. Vegyük a szignatúrák elejétől számolt távolságok maximumát legyen ez `elejétől_max`. Ez után vegyük a szignatúrák végétől számolt távolságok maximumát, legyen ez `végétől_max`. Ezek után az adatbuffer szükséges mérete:

`elejétől_max + n + végétől_max – 1`, `n ≥ 1`.

A fenti elrendezésben a középső n ofszetre igaz, hogy a legelső (legkisebb memóriacímen elhelyezkedő) ofszeten elfér az a szignatúra, aminek a mintavételi pontja a legtávolabb van a szignatúra elejétől, és a legutolsó (legnagyobb memóriacímen elhelyezkedő) ofszeten elfér a szignatúra, aminek a mintavételi pontja a legtávolabb van a szignatúra végétől. Következésképpen a középső `n` ofszetre minden szignatúra illeszkedése ellenőrizhető a szignatúra mintavételi pontjával, mint illesztési ponttal.

Ebből adódik a minimum szükséges bufferméret, ami `n` szomszédos bit ellenőrzéséhez szükséges mind a CPU-n, mint a GPU-n. A buffert csúszóablakként fogjuk használni az adatok feldolgozásához. Válasszuk ki a kívánt bitmennyiséget (`n`-et), amit egyszerre ellenőrizni akarunk. Ehhez határozzuk meg a CPU oldali, valamint a GPU oldali bufferméretet. Ezt a következő lépések követik.

1. A bufferbe olvassunk adatot az `elejétől_max` ofszettől a buffer végéig, vagy ameddig tudunk. Hívjuk ezt a szakaszt hasznos adatnak, és tartsuk számon.
2. A buffer `elejétől_max` ofszetjétől elérhető hasznos adatokkal töltsük meg a GPU adatbuffert a buffer végéig, vagy ameddig tudjuk.
3. GPU mintaillesztő kernel indítása, a GPU a bufferében ellenőrzi az adatokat. Legyen a bufferben tárol hasznos adat mérete `m`, ekkor az ellenőrzendő ofszetek száma `offsetCount = m – szignatúraméret + 1`.
4. Olvassuk az eredményt a CPU memóriába (lásd később).
5. A CPU-n találatokat vizsgáljuk meg a hozzájuk tartozó teljes szignatúrákkal, majd mentsük az igaz pozitívokat. Ezzel készen vagyunk az `n` darab ofszet ellenőrzésével.
6. Ha van még a CPU buffer végén adat, amit nem ellenőriztünk a GPU-n:
    - 6.1. Toljuk el a bufferben az adatokat `n`-el.
    - 6.2. Töltsük fel a bufferben így felszabaduló részt további adattal, amennyiben lehetséges, nem értük még el az ellenőrzendő adat végét.
    - 6.3. Ugorjunk a 2-es pontra.

A CPU buffer végéről természetesen akkor fogy el az adat, amikor a vizsgálandó adategység, például fájl vagy TCP csomag végére érünk. Ezen túl az adatfeldolgozás jellemzésekor a fájl kifejezést fogom használni, de természetesen az elv minden összefüggő bájtsorozatra ugyanaz. Amennyiben egymás után több fájlt szeretnénk megvizsgálni, az új fájl megnyitása után az 1-es ponttól indítsuk a feldolgozást.

A fenti algoritmust egészítsük ki azzal, hogy az adatbufferben mindig csak a hasznos adatot toljuk el, és dolgozzuk fel. Mivel a buffert csúszóablakként használjuk, a fájlok végére éréskor nem lesz már tele adattal. Tartsuk számon, hogy mettől meddig van benne hasznos adat, és csak ezen végezzük el a fenti műveleteket (csúsztatás, adat mozgatás a GPU-ba ellenőrzésre). A fenti algoritmusban megadott manipulálandó mennyiségek maximumokat jelölnek.

A bufferek mellett egyéb adatok is tárolandók a CPU és GPU oldalon. A GPU-nak a szignatúrákból nyert mintákhoz, a CPU-nak pedig a szignatúrákhoz is hozzá kell férnie, valamint ahhoz az információhoz, hogy melyik mintához melyik szignatúra tartozik. Ez utóbbi történhet például indexek formájában, hiszen a szignatúráink adottak, így egy listában tárolhatjuk őket valamilyen sorrendben.

A mintaillesztő kernelben egy GPU szál egy ofszetre való minden szignatúra minta illesztéséért felelős, így `n` ofszet esetén `n` darab szálra van szükség a teljes ellenőrzés megvalósításához. Ez azt jelenti, hogy `n` darab kernelt kell indítanunk, hiszen egy szál egy kernel példányt futtat. Egy találat értéke a talált minta egyedi azonosítója, például a minta helye a mintákat tartalmazó tárolóban. Ahhoz, hogy adott szál az összes ellenőrzött minta illeszkedését regisztrálni tudja, egy `mintaszám + 1` méretű tömbre van szüksége, ahová kizárólag ő írhat. Ebben elfér az összes találat abban a szélsőséges esetben is, ha adott ofszetre minden minta illeszkedik. Az extra adatmező lehetővé teszi, hogy az adatszerkezet első pozíciójában tároljuk, hogy az adott ofszetre hány minta illeszkedett. Ha az adott szál az indexének megfelelő indexű tömbbe írja a találatait, akkor ezen tömb indexe egyben a találat adatban elfoglalt helyét is jelöli. `n` darab ofszet esetén `n` ilyen tömbre van szükség. A kernel szerkezete a következő.

1. Számoljuk ki a szál indexét (`index = blokk_index * blokk_méret + szál_index_`blokkon_belül), ez egyben az általa ellenőrzendő ofszet indexe is a GPU bufferben.
2. A GPU adatbufferében az `index` ofszettől számított `mintaméret` darab bájtra próbáljuk meg sorban illeszteni az összes szignatúra kivonatunkat, és a találatokat írjuk egymás után a szál indexének megfelelő találatokat tároló tömbbe, a második pozíciótól indulva, közben számolva a találatok számát.
3. Írjuk a találatok számát a találatokat tartalmazó tömb első pozíciójába.

A fenti működésből következik, hogy a GPU-n `n` darab ofszet feldolgozásához `n * (mintaszám + 1)` mennyiségű memóriára van szükségünk, azaz a szükséges memória arányos az ofszetek számának négyzetével. Ez nem csak azt jelenti, hogy a memóriaigény rosszul skálázódik az egy körben feldolgozott ofszetek számának emelésével, hiszen a kernel futása után ezt az adatot vissza kell mozgatni a rendszermemóriába. A host-device memória tranzakciós sávszélesség nem túl magas, modern hardver esetén 8 GBps. Ez azt jelenti, hogy a fent vázolt módszer által mozgatott memóriamennyiség jelentős mértékben lassítja az algoritmust nagy ofszet mennyiség esetén. Ez probléma, hiszen a kernelindítás overhead igen nagy, ezért egyszerre minél több adatot szeretnénk ellenőrizni a GPU-n.

E. Seamans és T. Alexander [2] úgy oldotta meg ezt a problémát, hogy minden szignatúrához tartozó mintát egyedinek választott. Ez megoldja a memóriaszükséglet problémáját,

hiszen egyedi minták esetén minden kernel futásának eredménye a talált mintához tartozó szignatúra indexe, vagy egy negatív szám, például -1, ha nem illeszkedett az egyik szignatúra sem. Másfelől szignatúra adatbázistól függően előfordulhatnak olyan esetek, amikor nincs lehetőség minden szignatúrából egyedi mintát venni. Az én megoldásom eltérő, és a következő pontban ismertetem.

A GPU oldali szűrés eredményének rendszermemóriába olvasása után a következő lépés a találatok teljes ellenőrzése, azaz minden talált minta esetén a hozzá tartozó szignatúra `találati_pozíció – mintavételi_offset` pozícióra való illesztése. Amennyiben ez sikeres, a fájl fertőzött a szignatúra által reprezentált káros programmal. Mivel az illesztendő szignatúrákhoz tartozó mintákat már sikeresen illesztette a GPU, ezen szakaszokat átugorhatjuk a teljes illeszkedés ellenőrzése során.

4.2.2 Egyedi minták

Az előző pontban leírt problémára a megoldás abban rejlik, hogy a vizsgált adatra ofszetenként maximum egy szignatúra minta illeszkedhessen, hiszen így minden ofszet vizsgálatának eredménye a talált minta azonosítója lehet, például a minta indexe a mintákat tartalmazó tárolóban. Amennyiben nincs találat, az eredmény legyen negatív szám, például -1. Ezen eredményeket a szálak egy tömbbe az indexeikkel egyező indexbe írhatják, és a CPU-nak ezt a tömböt kell átmásolnia a rendszermemóriába a találatok validálásának érdekében.

E. Seamans és T. Alexander [2] megoldása úgy biztosítja a minták egyediségét, hogy megköveteli, hogy minden szignatúrából vett minta egyedi legyen. Ez limitálja az algoritmussal kompatibilis szignatúrahalmazokat, ahogy ezt az alábbi példa is mutatja.

Vegyünk egy olyan szignatúra adatbázist, amiben a legrövidebb szignatúra x bájt, és legyen 2^x darab egyedi szignatúránk az adatbázisban. E mellett legyen egy szignatúránk, ami több mint x bájtból áll. A szignatúráink természetesen egyediek. A maximum lehetséges mintaméret x, hiszen ez a legrövidebb szignatúránk hossza. Egy bájt két értéket vehet fel ({0, 1}), tehát x bájt Descartes-szorzata 2^x. Ez azt jelenti, hogy x mintahossz mellett 2^x darab egyedi minta létezik az x mintahossz mellett. Mivel az adatbázis $2^x + 1$ darab mintát tartalmaz, nem biztosítható a minták egyedisége. Kisebb mintahossz mellett a Descartes-szorzat is kisebb, tehát a helyzet csak romlik.

A megoldásomban nem követelem meg, hogy egyediek legyenek a minták, azonban mintavétel után halmazt képzek a mintákból, azaz eltávolítom a duplumokat, és tárolom minden mintára, hogy mely szignatúrák tartoznak hozzá. Így minden minta egyedi, és `mintaszám ≤ szignatúraszám`. Ezzel a megoldással szintén GPU szálanként egy egész szám tárolására alkalmas memóriamennyiségre van szükség az eredmények tárolásához. Mivel egy mintához több szignatúra tartozhat, a CPU-ra minta illeszkedés esetén valamivel több munka hárul. Ez a futási idő szempontjából többnyire nem jelentős overhead, valamint nem függ a szignatúra adatbázis összetételétől és a mintavételi technikától. Természetesen az az ideális, ha ezzel a sebességcsökkenéssel egyáltalán nem kell számolni. A *4.2.5.* fejezetben megmutatom, hogy jó mintavételezési technika esetén ez valóban nem is probléma. Előtte azonban következzenek az ennél sokkal jelentősebb optimalizációk.

4.2.3 Mintaillesztés bináris kereséssel

Vegyük észre, hogy az előző pontokban ismertetett algoritmus esetében a mintaillesztő kernel lineáris keresést végez a minták között: minden szál a saját ofszetjéhez tartozó adatszeletet keresi a mintahalmazban, szekvenciálisan ellenőrizve az összes mintát. A kernel futási ideje így egyenesen arányos a minták számával, azaz a futási idő `n` minta esetén `O(n)`. Általánosan megfogalmazható, hogy nagy adathalmazokon dolgozó algoritmusok esetén a lépésszám adathalmaz méretétől való egyenesen arányos függése nem kívánatos jelenség, ezért minimalizálandó.

Amennyiben nagy adathalmazon történő keresést szeretnénk gyorsítani, jó választás a bináris keresés. Ezen keresési módszer előfeltétele, hogy a vizsgált adatszerkezet rendezett legyen. A rendezett adatszerkezetek akkor ideálisak a rendezetlenekkel szemben, ha sok olvasást és kevés beszúrást végzünk rajtuk, hiszen mindkét művelet komplexitása `O(log(n))`, míg rendezetlen adathalmazok esetén a keresés `O(n)`, a beszúrás pedig `O(1)` lépésben végezhető el.

A fentiek alapján a mintahalmazunk tökéletes jelölt arra, hogy rendezetten tároljuk, hiszen a létrehozása után nem módosítjuk a keresés folyamán, viszont rengeteget olvasunk belőle. Rendezzük a mintahalmazt növekvő (akár csökkenő) sorrendbe, és módosítsuk úgy a szűrésért felelős kernel kódot, hogy az bináris keresést végezzen a mintahalmazon. A minták egyedi bájtsorozatok, tehát lexikografikusan egyértelműen rendezhetők. Legyen minden minta 0. indexe a MSB, és a mintavételt követően végezzük el az e szerinti rendezést.

Ahogy azt az *2.1.* fejezetben ismertettem, a kernelek esetén fontos az elágazások körültekintéssel való használata. A bináris keresés legelterjedtebb megoldása [5]:

```
int binary_search(int A[], int key, int imin, int imax)
{
  // continue searching while [imin,imax] is not empty
  while (imin <= imax)
    {
      // calculate the midpoint for roughly equal partition
      int imid = midpoint(imin, imax);
      if(A[imid] == key)
        // key found at index imid
        return imid;
      // determine which subarray to search
      else if (A[imid] < key)
        // change min index to search upper subarray
        imin = imid + 1;
      else
        // change max index to search lower subarray
        imax = imid - 1;
    }
  // key was not found
  return KEY_NOT_FOUND;
}
```

A fenti pszeudokódban látható, hogy a keresést végző ciklus három elágazást tartalmaz. Ezek az elágazások problémásak, hiszen a szomszédos szálak által keresett különböző bejegyzések többnyire más-más pozíciókban találhatók (vagy nem találhatók) a mintahalmazban. Alternatív megoldásként az egyenlőségvizsgálat a cikluson kívül is végezhető [5]:

```
// inclusive indices
//   0 <= imin when using truncate toward zero divide
//     imid = (imin+imax)/2;
//   imin unrestricted when using truncate toward minus infinity divide
//     imid = (imin+imax)>>1; or
//     imid = (int)floor((imin+imax)/2.0);
int binary_search(int A[], int key, int imin, int imax)
{
  // continually narrow search until just one element remains
  while (imin < imax)
    {
      int imid = midpoint(imin, imax);

      // code must guarantee the interval is reduced at each iteration
      assert(imid < imax);
      // note: 0 <= imin < imax implies imid will always be less than imax

      // reduce the search
      if (A[imid] < key)
        imin = imid + 1;
      else
```

```
        imax = imid;
    }
  // At exit of while:
  //   if A[] is empty, then imax < imin
  //   otherwise imax == imin

  // deferred test for equality
  if ((imax == imin) && (A[imin] == key))
    return imin;
  else
    return KEY_NOT_FOUND;
}
```

Ez a kód rendkívül előnyös abból a szempontból, hogy a keresést végző ciklusban csak két elágazás található. Ez a tulajdonság kernelben való futás esetén különösen fontos. A lineáris keresés fenti algoritmussal való lecserélését követően egy 1781 bejegyzést tartalmazó szignatúra adatbázis esetén a teljes algoritmus 8-szoros gyorsulását tapasztaltam. Az `n/log(n)` hányados `n` növekedésével nő, így nagyobb adatbázisok esetén a sebességkülönbség még inkább szembetűnő.

4.2.4 Mintaillesztés MPHF-el

Bár a fenti eredmény imponáló, az `O(log(n))`-es lépésszám még mindig nem optimális: a keresés lépésszáma jelentős mértékben függ a mintahalmaz méretétől. Az olvasáshoz szükséges lépésszám szempontjából jobb megoldást biztosítanak a vödrös hash táblák. Ezen adatszerkezetek kulcs-érték párokat tárolnak. Több tároló elemmel (vödörrel) rendelkeznek, és adott bejegyzés esetén a kulcsból a táblához tartozó hash függvény által generált érték határozza meg, hogy az adott érték melyik tárolóba kerül. Ezen táblák általában `O(1)` időben érik el a kívánt elemet. Ez az érték romlik, ha a több bejegyzés is ugyanazon tárolóba kerül a kulcsok hash értéke alapján. Ezt ütközésnek nevezzük. Legrosszabb esetben az elérés `O(n)` időt vesz igénybe. Ez az az eset, amikor minden tárolt elem ütközik. Az ilyen mértékű degradáció azonban csak nagyon rosszul szóró hash függvény hatására vagy nagyon specifikus adathalmaz esetén történik. Jó szórású a hash függvényünk, ha a használt kulcsokat egyenletesen tudja elosztani a rendelkezésre álló vödrök között.

A hash táblák némileg több memóriát kívánnak, mint a bináris kereséshez szükséges adatszerkezetek, de jó hash függvény esetén ez a plusz költség nem jelentős, figyelembe véve, hogy a mai grafikus kártyák a vírusadatbázisoknál nagyságrendekkel több memóriával rendelkeznek. Az extra memóriaszükséglet a tárolandó adathalmaz méretének konstans szorzója.

A jó hash függvénnyel ellátott hash táblák általános esetben is jól teljesítenek, de a szignatúraminták esete igen speciális, hiszen ez esetben konstans mintahalmazról beszélünk. Ilyen esetekre szolgálnak a MPHF-ek (Minimal Perfect Hash Function). A „Perfect Hash”, azaz tökéletes hash kifejezés azt takarja, hogy ezen függvények garantálják az ütközésmentes tárolást. Ez garantált elérési időt jelent, az általános hash táblákkal ellentétben. A minimálisság arra utal, hogy a függvény a hash tábla elemeit a lehető legkevesebb „vödörben”, azaz tárolóban helyezi el. Mivel a hash függvény tökéletes, egy vödörhöz egy kulcs tartozik, és természetesen minden kulcshoz tartozik egy vödör. Következésképpen `n` kulcs `n` vödörben tárolódik. Az MPHF-ek tehát olyan hash függvények, amelyek O(1) elérési időt biztosítanak, és a vödörszámuk megegyezik a tárolt elemek számával.

Az általam használt MPHF a [6]-ban leírtakra épül. A cikkben ismertetett algoritmus elméleti háttere a [7] munkában található. A hash táblát egyszer kell felépítenünk, hiszen a szignatúráink víruskeresés közben nem változnak. A tábla létrehozása tehát a keresési időn kívül történik.

Az adatszerkezet felépítéséhez és használatához szükségünk lesz egy tetszőleges méretű bájtsorozatok hash-elésére szolgáló gyors és jól szóró algoritmusra. Az implementációmban az FNV-1 algoritmust [8] használom. Ennek a pszeudokódja a következő.

```
hash = FNV_offset_basis
  for each byte_of_data to be hashed
        hash = hash × FNV_prime
        hash = hash XOR byte_of_data
  return hash
```

Azt FNV-1 egy gyors és egyszerű hash függvény, amely tetszőleges bájtsorozat hash-elésére képes, így a mi esetünkben ideális. Az `FNV_offset_basis` egy speciálisan megválasztott szám a jobb szórás érdekében, ez az FNV-1 előrelépése az FNV-0-val szemben. Az `FNV_prime` egy prim szám, ami ugyancsak speciálisan megválasztott, és jó szórást biztosít. A jó szórás a MPHF építési fázisában fontos, ugyanis ezután az MPHF definíció szerint nem szór.

Természetesen fenti függvény által generált értékeknek a hash tábla vödörszámával való maradéka képzendő. Az így kapott értékekkel lesz címezhető a tábla. A teljes függvény tehát a következő.

```
hash = FNV_offset_basis
for each byte_of_data to be hashed
    hash = hash × FNV_prime
    hash = hash XOR byte_of_data
return hash MOD NUMBER_OF_BUCKETS // a hash tábla vödörszáma
```

A hash tábla két altáblából fog állni. Az egyik a kulcs-érték párokat tárolja, nevezzük ezt `érték_tábla`-nak. A másik tábla egy számértékeket tároló köztes táblára, nevezzük ezt az `offset_tábla`-nak.

Egy MPH tábla a következőképpen áll elő.

1. Hozzunk létre egy `mintaszám` darab vödörből álló hash táblát, ami az FNV-1-gyel hash-el. Töltsük fel ezt a táblát a kulcs-érték párjainkkal a kulcsok hash értéke alapján.
2. Rendezzünk az 1. pontban létrehozott hash tábla vödreit elemszám szerinti csökkenő sorrendbe.
3. Elemszám szerinti csökkenő sorrendben minden vödörre, ahol a vödör elemszáma nagyobb, mint 1:
   3.1. `offset = 1`
   3.2. Képezzük a vödörben szereplő értékek hash-jeit az `FNV_offset_basis = offset` alternatív FNV-1 függvénnyel.
   3.3. Ha a képzett hash-ek valamelyike az `érték_tábla` valamely nem üres vödrére mutat, vagy bármely két hash ugyanarra a vödörre mutat, akkor `offset = offset + 1`, és ugorjunk a 3.2.-es pontra.
   3.4. A vödrünk összes elemét helyezzük az `érték_tábla`-ba a hash-jeik által vödrökbe úgy, hogy minden elem abba a vödörbe kerül, amelyikre a belőle képzett hash mutat.
   3.5. `offset_tábla[index] = offset`, ahol `index` a vördünk rendezés előtti indexe.
4. Minden vödörre, ahol az elemszám 1:
   4.1. Legyen `index` az első üres vödör indexe az `érték_tábla`-ban, legyen `hash` a vödörben található elem FNV-1 hash-e, majd `offset_tábla[hash] = -index`.

A fenti algoritmus futási ideje a [7] szerint lineáris, nálam bőven egy másodperc alatt fut 22053 bejegyzéses szignatúra adatbázisra egy szálon egy i7 4790k 4.4Ghz-es processzoron. Ez azt mutatja, hogy az előfeldolgozás nem ró túl nagy terhet a rendszerre, és mindenképen érdemes alkalmazni. A fenti algoritmust persze elég szignatúra adatbázisonként egyszer futtatni és a generált adatszerkezetet eltárolni, így a víruskeresési időt nem befolyásolja.

A táblánkban a szignatúraminták a kulcsok. Minden kulcshoz tartozó érték azon szignatúrákból és az ezekhez tartozó mintavételi ofszetekből áll, ahol a szignatúrából vett minta egyezik a kulcs mintával.

Egy mintahalmazból a fenti módon generált tábla segítségével konstans idő alatt meghatározhatjuk, hogy egy `mintahossz` méretű bájtsorozat benne van-e a mintahalmazban. Ezt a következőképpen tehetjük meg.

1. Legyen `hash1` a bájtsorozat FNV-1 hash-e.
2. `hash2 = offset_tábla[hash1]`.
3. Amennyiben a bájtsorozat egyezik az `érték_tábla[hash2]` értékben szereplő kulccsal, a bájtsorozat tagja a mintahalmaznak.

Ezzel a módszerrel egy GPU szál az MHPF segítségével a mintahalmaz méretétől függetlenül, konstans idő alatt meg tudja határozni, hogy az ellenőrzendő adatból a rá kiosztott ofszettől mintavételezett mintahossz méretű bájtsorozat benne van-e a mintahalmazban.

GPU oldalon az `érték_tábla`-ból csak a kulcsok tárolandók, így a fenti lépéssorozat már elvégezhető. Ha egy szál a fenti műveletet végezve egyezést talált, akkor futásának eredménye a fent említett `hash2` index, ellenkező esetben pedig -1. CPU oldalon az `érték_tábla[hash2]` bejegyzés tartalmazza a talált mintához tartozó szignatúrákat és a hozzájuk tartozó mintavételi ofszeteket is, ezekkel végzendő a találati ponton a teljes ellenőrzés.

Az ismertetett MPHF alapú mintaillesztés egy 1781 bejegyzést tartalmazó szignatúra adatbázist használva a bináris kereséshez képest 15%-os sebességnövekedést biztosított. Erről az adatbázisról később egy 22053 bejegyzéses adatbázisra váltottam, és a váltás után némi növekedést tapasztaltam az algoritmus futási idejében. Ennek a mértéke az ellenőrzött fájlok tartalmától függ, és a legrosszabb esetben 50-80% körül mozog. Ez a jelenség a minta alapú szűrés velejárója. A nagyobbik adatbázis bejegyzéseinek száma a kisebbikének több, mint 12-szerese, és a nagyobb szignatúrahalmazból nagyobb mintahalmaz adódik. Ez, bár az MPHF sebességét nem befolyásolja, azt eredményezi, hogy a mintaillesztés során több találat keletkezik. Így a CPU-nak több szignatúraillesztést kell végeznie, és ez okozza a sebességcsökkenést.

4.2.5 Minta egyezések minimalizálása

Az algoritmus kezdeti állapotában minden szignatúra közepéből mintavételeztem. Ahogy azt a *4.2.2.* fejezetben is említettem, ha a szignatúrákból vett minták közül sok az egyező, az megnövelheti a CPU által végzendő feladatok számát. Érdemes ezért olyan módszerrel mintavételeznünk, ami minimalizálja az egyezéseket. A legegyszerűbb, jól működő módszer:

Minden szignatúrára:

1. Legyen `offset = 0`.
2. Vegyünk mintát a szignatúra `offset`-edik pozíciójából.
3. Ha a vett minta nem egyezik egyik eddig mentett mintával sem, mentsük el, és készen vagyunk. Ellenkező esetben `offset = offset + 1` és ugorjunk a 2. pontra.

A minták egyezésének minimalizálására alternatív megoldást mutat be a `3.3.1.` pont.

4.2.6 A GPU és CPU feldolgozás párhuzamosítása

A CPU oldali ellenőrzés függ a GPU általi szűrés eredményétől, de ezek a műveletek az algoritmus kis módosításával párhuzamosíthatók. Ehhez arra van szükség, hogy a GPU mindig eggyel előrébb járjon az adatok feldolgozásában. Ebből az következik, hogy a GPU és a CPU által egy időben ellenőrzött adatok nem egyezőek. A GPU ellenőrzi a buffer elejét, a CPU pedig az ez után következő részt. Az ellenőrzéshez szükséges adatkontextusokat ezúttal is biztosítani kell a bufferben.

Egy olyan adatbufferre lesz szükségünk, amiben elfér két akkora adatblokk, amennyi ofszetet egy körben ellenőrizni szeretnénk. A *4.2.1.* fejezet jelöléseit használva `n` ofszet ellenőrzéséhez a következő bufferre van szükség.

```
vége := n > végétől_max ? 0 : végétől_max – n
```

A buffer a szokásos csúszóablak módszerrel töltődik fel a végétől az eleje felé, a képen jelölt módon. Először a `GPU scan` és a `CPU scan` szakasz határáig töltjük fel. Itt végzünk egy GPU szűrést, majd mentjük a szűrés eredményeit és tovább toljuk az ablakot `n`-el. Innentől minden iterációban fut a CPU és a GPU oldali keresés is. A GPU oldali keresés mindig egy lépéssel előrébb

jár, és a CPU oldali keresés a GPU-szűrés előző iterációból származó eredményeit használja az ellenőrzés felgyorsításához. Az adat végét elérve lesz egy iteráció, amikor csak a `CPU scan` oldalon lesz értelmes adat, így ekkor csak a CPU oldali keresést kell elvégeznünk.

A fenti módszerrel a GPU és CPU oldali keresés egymással párhuzamosan futtatható, így az algoritmusnak optimálisabbá válik az időkihasználása.

4.3 Rosszindulatú adatok elleni védelem

A keresőmotor megbízhatóságának érdekében fontos feladat, hogy az az ellene irányuló adatokat a megfelelő módon kezelje.

Ha egy támadó ismeri a keresési algoritmust és a vírusadatbázist, szándékosan előállíthat rosszindulatú fájlokat. Még egyszerűbb a helyzet, ha a személy ismeri a szignatúra mintákat, vagy a mintavételezési eljárást. Ekkor előállíthat olyan adatot, amiben nagy számban fordulnak elő CPU oldali ellenőrzést előidéző bájtsorozatok, ezzel erősen lelassítva az algoritmust.

A keresés lassítása, mint támadás, elsősorban a bejövő hálózati forgalom ellenőrzésében hátrányos. Implementációm a háttértáron található fájlokat ellenőrzi, de a program módosítható úgy, hogy a bejövő hálózati forgalmat ellenőrizze, így fontos minimalizálni az algoritmus támadhatóságát.

4.3.1 Véletlen mintavételezés

Amennyiben egy támadó ismeri az algoritmus által használt szignatúramintákat, lelassíthatja az ezen algoritmussal védekező rendszert az által, hogy ezen mintákból készített adatokkal árasztja el az algoritmust futtató végpontot. Ha az algoritmus a *4.2.5*-ös fejezetben leírt módszert használja a mintavételhez, a támadó a szignatúra adatbázis ismeretében a mintákat könnyen meghatározhatja, és sikeres támadást indíthat a rendszer ellen.

Ezt elkerülendő, biztonságosabb mintavételi eljárásra van szükségünk a mellett, hogy az egy mintához tartozó vírusok számát alacsonyan tartjuk.

Az általam használt eljárás a *4.2.5*-ös fejezetben leírtaktól annyiban különbözik, hogy a lehetséges mintavételi pozíciókon véletlen sorrendben történik mintavételi próbálkozás. A mintavételi algoritmus egy szignatúrára tehát a következő.

1. Legyen `offset` azon pozíciók egyike a szignatúrában, ahol még nem történt mintavételi kísérlet.
2. Vegyünk mintát a szignatúra `offset`-edik pozíciójából.
3. Ha a vett minta nem egyezik egyik eddig mentett mintával sem, mentsük el, és készen vagyunk. Ellenkező esetben ugorjunk a 2. pontra.

Ha a fenti algoritmussal veszünk mintát a szignatúráinkból, akkor a minták reprodukálásához a keresőmotort futtató rendszerhez való hozzáférés, vagy bájtkombinációk próbálgatása és a rendszer válaszidejének figyelése szükséges.

Kivételt képeznek ez alól azon szignatúrák, amelyek hossza egyezik a szignatúrahosszal. A mintahossz egyenlő a legrövidebb szignatúra hosszával, így ilyenek feltételül előfordulnak. Ezek segítségével sikeres támadások indíthatók a keresőmotor ellen. Ezt orvosolja a *4.3.2* fejezetben bemutatott módszer.

4.3.2 Rövidzár osztályozás

Könnyen előállíthatók az adatáteresztő képességet jelentősen csökkentő (a keresési időt növelő) adatfolyamok azon szignatúrákból, amelyek mérete megegyezik a kereső algoritmus által használt mintamérettel. Ezen szignatúrák rövidek, így sűrűn előfordulhatnak az ellenőrzött adatfolyamban, és CPU oldali ellenőrzést igényelnek. Ilyen szignatúrák minden szignatúra adatbázis esetén előfordulnak, hiszen a választott mintaméret egyenlő a legrövidebb szignatúra méretével.

A mintaméret csökkentése természetesen nem megoldás, hiszen ez esetben a legrövidebb szignatúrával való támadás hatékonysága nem csökken, ha pedig a támadó rájön a mintavételi pontra, akkor a hatékonyság növelhető, hiszen elég ezt a rövidebb mintát ismételnie az adatfolyamában.

Ezzel szemben a mintaméret növelése nem lehetséges, hiszen a lehetséges mintaméretek maximuma a használt szignatúrák méretének minimuma. Ez abból következik, hogy egyenlő méretű mintákkal dolgozunk, és egyik szignatúrából sem tudunk a méreténél nagyobb mintát venni.

Vegyük észre, hogy ha egy támadó mintahossz méretű szignatúrákkal támad, akkor védekezhetünk ez ellen úgy, hogy egy szignatúra jelenlétét csak egyszer regisztráljuk az ellenőrzött adategységben (fájlban). Így, ha megtaláltunk egy szignatúrát, kivonhatjuk azt a későbbiekben a fájlban keresendő szignatúrák közül.

Azzal, hogy egy káros bájtsorozat jelenlétét maximum egyszer regisztráljuk fájlonként, az algoritmus elveszti a lehetőséget, hogy kijavítsa a fertőzött fájlt, hiszen nem ismeri az összes fertőzést a fájlban. Ha javítás után újraszkenneli a fájlt, megtalálhatja a következő javítandó pontot, és ezt addig ismételheti, amíg végig nem ért a fájlon, de ha mindenképp meg ki akarjuk javítani a fájlt, akkor tegyünk nála kivételt, és regisztráljunk benne minden találatot.

Mivel a fentebb ismertetett, minden szignatúrát csak az első előfordulásakor regisztráló megoldás nem képes a fájlok effektív helyreállítására, többnyire felesleges információ a fertőzés helye és típusa. Arra vagyunk kíváncsiak, hogy fertőzött-e egy fájl, vagy tiszta. Az előbb ismertetett megoldás helyett tehát, ha találunk egy szignatúrát egy fájlban, akkor jelöljük a fájlt fertőzöttnek az ezen szignatúrához tartozó vírussal, és térjünk át a következő ellenőrzendő fájlra. Ezzel a megoldással hatástalaníthatók az algoritmus elleni azon támadások, amelyek teljes szignatúrákat alkalmaznak, hiszen ez esetben az adategység egyből fertőzöttnek minősül, és nem történik rajta további vizsgálat. Hívjuk a módszert rövidzár osztályozásnak.

Elképzelhető olyan adatfolyam is, amelyben nem teljes szignatúrák szerepelnek, hogy elkerüljék a rosszindulatú adat rövidzár kiértékelését. Ha ezek a szignatúrarészletek kis részét képezik az eredeti szignatúrának, akkor a szignatúrából vett minta nagy valószínűséggel nem szerepel bennük, így nem jelentenek gondot. Ha a szignatúrarészletek nagy részét képzik a teljes szignatúráknak, akkor nagy eséllyel tartalmazzák a szignatúrákból vett mintákat, így ilyen adatokkal lassítható a rendszer.

Ezzel szemben szóba jöhet az a védekezési módszer, hogy ha az egy fájlban a GPU által talált fals pozitívak száma elér egy határértéket, akkor rosszindulatúnak minősítjük a fájlt, és

leállítjuk a fájl vizsgálatát. Ezen funkció pontos meghatározása további kutatást igényel, erre ez a munka nem terjed ki.

4.3.3 Nem kívánt minták elkerülése

A szignatúrákban egyes bájtsorozatok gyakrabban előfordulnak, mint mások. Ezen bájtsorozatok a mellett, hogy átlagos programokban is gyakran előforduló utasítások lehetnek, nagyobb eséllyel mintavételeződnek. Ezen okokból ezek a bájtsorozatok lassíthatják az algoritmust ártatlan fájlok vizsgálata során, valamint támadási pontok is.

Jó példa erre a csupa 0 bájtsorozat, ami gyakran előfordul különféle szignatúrákban. A gyakori előfordulás miatt könnyen kaphatunk olyan mintát, ami csak 0-kból áll. Ha ilyen mintával szűrünk a GPU-n, egy 0-ás bájtokkal megtöltött nagyobb fájl a sokszorosára lassíthatja az algoritmus futási sebességét. Innen is látszik milyen fontos, hogy ne tudódjon ki, milyen mintákat használ az algoritmus az adatok szűréséhez.

Szükséges tehát, hogy a szignatúrákban és fájlokban gyakran előforduló bájtsorozatokat hátrányban részesítsük a szignatúrákból való mintavételezés során. Mondhatni a szignatúra szignatúráját keressük.

Megoldásomban a *4.3.1* fejezetben bemutatott véletlen mintavételezést azzal egészítettem ki, hogy nem fogadok el olyan mintát, amiben végig ugyanazon bájt érték ismétlődik. Kivételt képez az az eset, amikor egy ilyen szempontból elfogadható mintát sem találtam, ekkor az utolsó mintavételi próbálkozás eredménye lesz a használt minta.

5 Implementáció

Az algoritmus elméleti működése az előző fejezetekben bemutattam és sok helyen az implementációra is kitértem, azonban hátra maradt néhány implementációs szempontból fontos kérdés. Ezek tárgyalása következik.

5.1 Az implementációs környezet

Az algoritmus megvalósításához egy gyors kódot generáló programozási nyelvre volt szükségem. E mellett feltétel volt, hogy alacsony szintű műveleteket tudjak használni a nyelvben, hiszen a GPU kernel gyors működéséhez primitív, effektív adatszerkezetekre volt szükségem. Természetes választás volt tehát a C++, hiszen ez egyben a legelterjedtebb CUDA programozási környezet is (illetve ennek egy kibővített részhalmaza).

A Visual Studio 2013-mat [13] használtam az implementációhoz, ez telepíthető támogatást biztosít a CUDA-ban való programozáshoz. Ehhez a CUDA Toolkit [14] nevű program telepítendő, én ebből a 7.5-ös verziót használtam. A CUDA platform igénybe vételéhez szükség van a használt gépen egy CUDA kompatibilis GPU-ra.

Általánosan hasznos eszköz még C++-hoz a Boost [15], ami sok hasznos C++ könyvtár gyűjteménye. Az implementációm konzolos felületen működik, ehhez a `boost::program_options` könyvtárat használtam. A help utasításra az alábbi szöveg generálódik.

```
C:\Folder1>Folder2\virso --help
Allowed options:
  -h [ --help ]                         produce help message
  -s [ --signature-file ] arg (=C:\Folder1\Folder2\signatures.sig)
                                        signature file path
  -t [ --target ] arg (=C:\Folder1)
                                        scan targets
  -o [ --offset-count ] arg (=1048576)  number of offsets to be
sent to the gpu per round
```

A programnak paraméterként adható a használt szignatúra adatbázis fájl neve és az ellenőrzendő mappák, fájlok címe. Szignatúra adatbázisnak alapértelmezetten a saját mappájában

levő `signatures.sig` nevű fájlt veszi, az ellenőrzendő mappa pedig alapértelmezetten a working directory. Megadható még, hogy a GPU hány adat ofszetet ellenőrizzen egyszerre. Ez az érték alapértelmezetten 2^{20}, azaz bájtok esetén egy megabájt.

Az operációs rendszer mappaszerkezetében való navigálásra a `boost::filesystem`-et vettem igénybe.

5.2 A szignatúra adatbázis

Az algoritmus a pontos bájtkódjuk által meghatározott szignatúrákat keres a vizsgált adatokban. Nem terjed ki tehát a vizsgálat olyan szignatúrákra, amelyekben joker karakterek szerepelnek. A használt vírusadatbázisok szöveges fájlok, amelyekben a kártékony programok nevei és szignatúrái párosával tárolódnak. A szignatúra bájtkódok hexadecimális formában vannak ábrázolva. A programneveket a hexadecimális karaktersorozatoktól az „=” határolókarakter választja el. A szintaxis tehát a következő.

```
név=szignatúra
```

Példa:

```
Skew.445 (Clam)=51525653558bec33c98ec180fc4b7413fa26c706
```

A szignatúrák a ClamAV open source vírusirtó szignatúra fájljaiból származnak. A fenti szintaxis a szignatúrák legegyszerűbb ábrázolási módja. Ez a forma nem tartalmaz joker karaktereket, információt a szignatúra fájlban való lehetséges pozícióiról, valamint arról, hogy milyen típusú fájlokban vizsgálandó ez a szignatúra.

5.3 Device-Host memóriatranzakciók minimalizálása

Ahogy már említettem, a rendszermemória és a GPU memória (host és device memória) közötti adatátvitel, azaz memóriatranzakció nem túl gyors, jó mai hardver esetében az elméleti sávszélesség 8 GBps. E mellett a tranzakciók indítása némi overhead-del jár. Érdemes tehát a tranzakciós hívásokat és a mozgatott adatmennyiséget alacsonyan tartani.

Az ideális eset az lenne, ha egy, a találati index-találati pont párokat folytonosan tároló tömböt tudnánk visszaolvasni a GPU-ból a rendszermemóriába, ezzel lenne minimális az átlagos

tranzakciós mennyiség. A GPU-n azonban ehhez a szálak között annyi szinkronizációs lépésre lenne szükség, hogy a végeredmény jóval lassabb lenne, mint az előző bekezdésben leírt módszer.

A *4*-es fejezetben leírt algoritmusban a GPU-n végzett adatszűrés eredménye a vizsgált adat minden bájt ofszetjére az arra az ofszetre illeszkedő szignatúra minta indexe a minták tárolójában, vagy `-1`, ha az egyik minta sem illeszkedik. Kellően nagy, például 100 000 bejegyzéses adatbázis esetén a minták indexei nem férnek el 1 és 2 bájton, így 4 bájton tárolom őket. Ebből az következik, hogy `n` adat ofszetre a GPU `4n` méretű eredményt generál, és ezt mind vissza kell olvasni a rendszermemóriába.

Az implementáció során kísérleti úton jobb sebességet értem el úgy, ha ofszetenként csak egy igaz/hamis boolean értéket adtam vissza, ahol igaz = találat, hamis = nincs találat. Ezzel a módszerrel a CPU azt az információt kapja meg, hogy hol található szignatúra minta az adatban. Ezután a CPU-nak is el kell végeznie ezen pozíciókon az MPHF alapú mintaillesztést. Ez több munka, mint ha készen kapná a talált minták indexeit, de így ofszetenként csak egy bájt a GPU-n végzett szűrés eredménye. A device-host adatátvitel ilyen módon rövidebb időt vesz igénybe, így végeredményben az algoritmus gyorsabban fut.

Egy bájton természetesen 8 darab igaz/hamis érték is tárolható, azonban egy bájtot nem képes több GPU szál párhuzamosan manipulálni. A szálak közti szinkronizáció nagy overhead-del járna, és a CPU oldali dekódolás is extra időt igényel, így ez a megközelítés nem szerencsés.

5.4 Pinned memory használata

A CUDA platform lehetőséget biztosít olyan, úgynevezett pinned memóriaterületek foglalására a rendszermemóriában, amik nem swap-elhetők. Ezen memóriaterületeket a rendszer nem írhatja ki a virtuális memóriába, így mindig garantáltan a fizikai memóriában tartózkodnak, és az elérésük mindig gyors.

Ahogy M. Boyer mérései [16] is mutatják, ez a memória alkalmas a memóriatranzakciók gyorsítására. E mellett M. Boyer [17] említi, hogy ezen típusú memóriafoglalás és felszabadítás igen időigényes, tehát körültekintéssel használandó.

Az ebben a munkában ismertetett algoritmus rengeteg memóriatranzakciót végez. A használt bufferek egyszer allokálandók, a futás elején, ezért a pinned memória foglalási és deallokálási overhead-je könnyen megtérül. Így foglalt memóriaterület a host oldali adattároló

buffer és az a buffer, amibe a GPU-ból a szűrés eredményeként kapott bool értékeket olvasom. E kettő közül a második buffer pinnelése okozza a jelentősebb sebességnövekedést (~8%), mivel ezt a buffert device-host tranzakcióra használom, de a másik buffer is még mérhető gyorsulást eredményez (~2%). Ezen megállapítások összhangban vannak M. Boyer méréseivel [16], amik azt mutatják, hogy device-host tranzakciók esetén adott adatmennyiség átvitelekor nagyobb gyorsulás érhető el, mint host-device esetben.

6 Mérések

A munkám többek között arra irányult, hogy egy olyan algoritmust fejlesszek ki, ami gyorsítani képes a kártékony programok azonosításán. Ahhoz, hogy a CPU-n futó algoritmussal szembeni gyorsulást megtudhassuk, a CPU-n futó és a GPU gyorsított algoritmusok futási idejének összehasonlítására van szükség.

CPU-n futó algoritmus alatt a bemutatott algoritmusnak azon módosítása értendő, amely a minta alapú szűrést a GPU helyett a CPU-n végzi ofszetenként szekvenciálisan.

A vizsgálandó paraméterek a mérések során a fájlméret, az egyszerre ellenőrzendő ofszetek száma és a fájlokban található teljes és részleges szignatúrák száma. Az egyik mért érték az abszolút áteresztőképesség, másik pedig a speedup. A speedup a tisztán CPU-n futó és a hardveresen gyorsított algoritmusok futási idejének a hányadosa.

Minden mérési eredmény a fejezetben 10 mérés átlagaként adódik, ez a CPU-n zajló egyéb folyamatok által okozott mérési pontatlanságokat hivatott minimalizálni.

6.1 A tesztkörnyezet

A mérési eredmények ismertetése előtt bemutatom a használt tesztrendszert.

```
CPU:  Intel Core i7-4790K
RAM:  Kingston 16GB DDR3-1866MHZ NON-ECC CL9 DIMM (KIT OF 2) XMP SAVAGE
GPU:  Gigabyte PCIe N970G1 GAMING-4GD GTX 970 4GB DDR5 (PCIe 3.0 x16)
SSD:  Samsung 500GB SATA3 2,5" 850 EVO Basic SSD
OS:   Windows 10
```

A mérések során gondot jelentett a fenti listában szereplő SSD, aminek az elméleti szekvenciális olvasási sebessége 540 MBps. Amennyiben az ellenőrzött adatokat ellenőrzés közben az SSD-ről olvastam be, az algoritmuson 530 MBps körüli áteresztőképességet mértem, és az SSD folyamatosan 100% körüli használatban volt a rendszer szerint.

Ebből arra következtettem, hogy az SSD igen jelentős szűk keresztmetszet az algoritmus adatáteresztő képességét illetően. Ez gond, hiszen így az az algoritmus valódi sebessége, amit például hálózati adat ellenőrzése során érhetne el, nem mérhető. Emiatt az implementációmban a működést úgy módosítottam, hogy az ellenőrzendő adatokat a program ellenőrzés előtt a

memóriába olvassa. Ezzel a módosítással az SSD, mint szűk keresztmetszet megszűnik az adatok ellenőrzése során.

A fenti megoldásnak a víruskeresés szempontjából nincs gyakorlati haszna, sőt, az ellenőrzendő adatméretet jelentősen korlátozza. Másfelől a megoldás nem kerülheti ki tökéletesen az SSD-t, hiszen memóriában tárolt adatról az operációs rendszer dönthet úgy, hogy swap-eli. Ebben az esetben ismét számolnunk kell az SSD-vel, mint szűk keresztmetszettel.

Az algoritmus csak a mérések kedvéért olvassa tehát a tesztelendő adatokat előre a memóriába. Ez a megoldás, bár nem tökéletes, segít egy jobb közelítést adni az algoritmus adatáteresztő képességére.

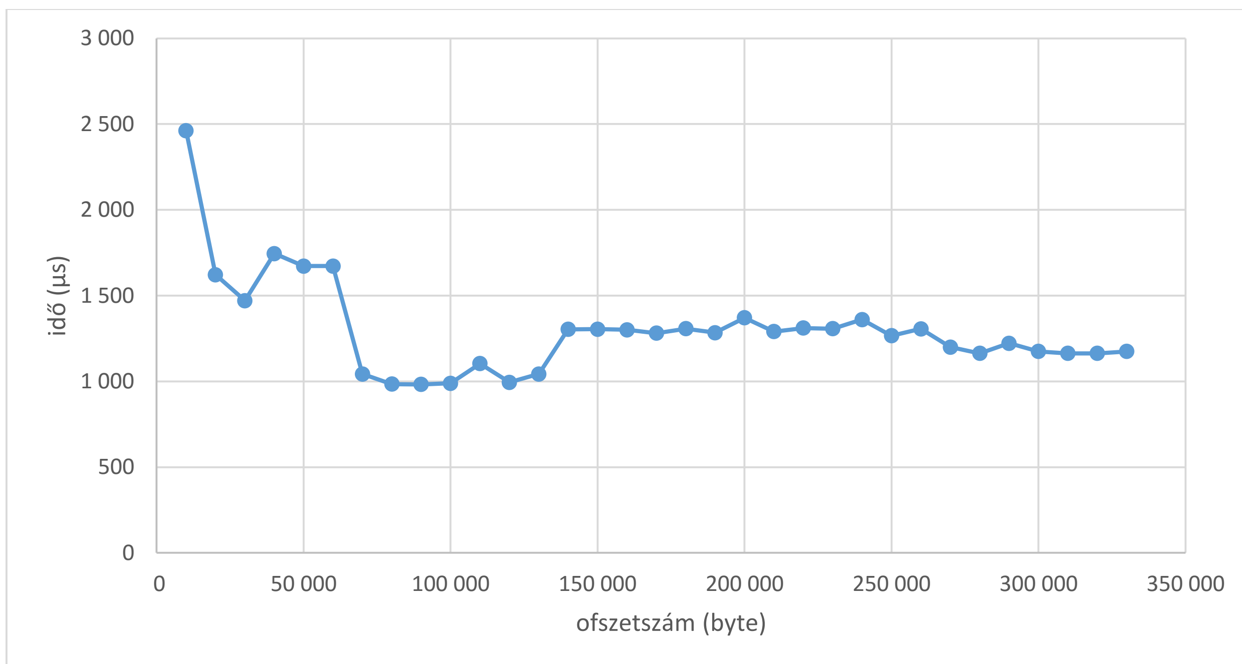
A valós felhasználási esetben, amikor nem tárolunk egyszerre minden adatot a memóriában, az algoritmus memóriaszükséglete arányos a szignatúra adatbázis méretének és az egyszerre ellenőrzendő ofszetek számának az összegével. Egy nagyobb vírusadatbázis mérete pár MB, az egyszerre ellenőrzendő ofszetek számára pedig jó érték a 10^{20}, így pár MB-nyi adatot kell egyszerre a rendszermemóriában tárolni. Ez nagyságrendekkel kevesebb, mint a manapság rendelkezésre álló rendszer- és GPU memória, ezért a memóriaszükségletre nem fordítok hangsúlyt a mérések során.

6.2 Feldolgozási sebesség

Ez a fejezet az algoritmus teljesítményét mutatja be különböző paraméterezések és munkaadagok esetében. Az ebben a fejezetben vizsgált fájlok nem fertőzöttek, tehát átlagosnak tekinthetők. A víruskeresés során ilyenek a leggyakrabban előforduló fájlok, ezért a feldolgozási sebességet elsősorban ezeken érdemes mérni.

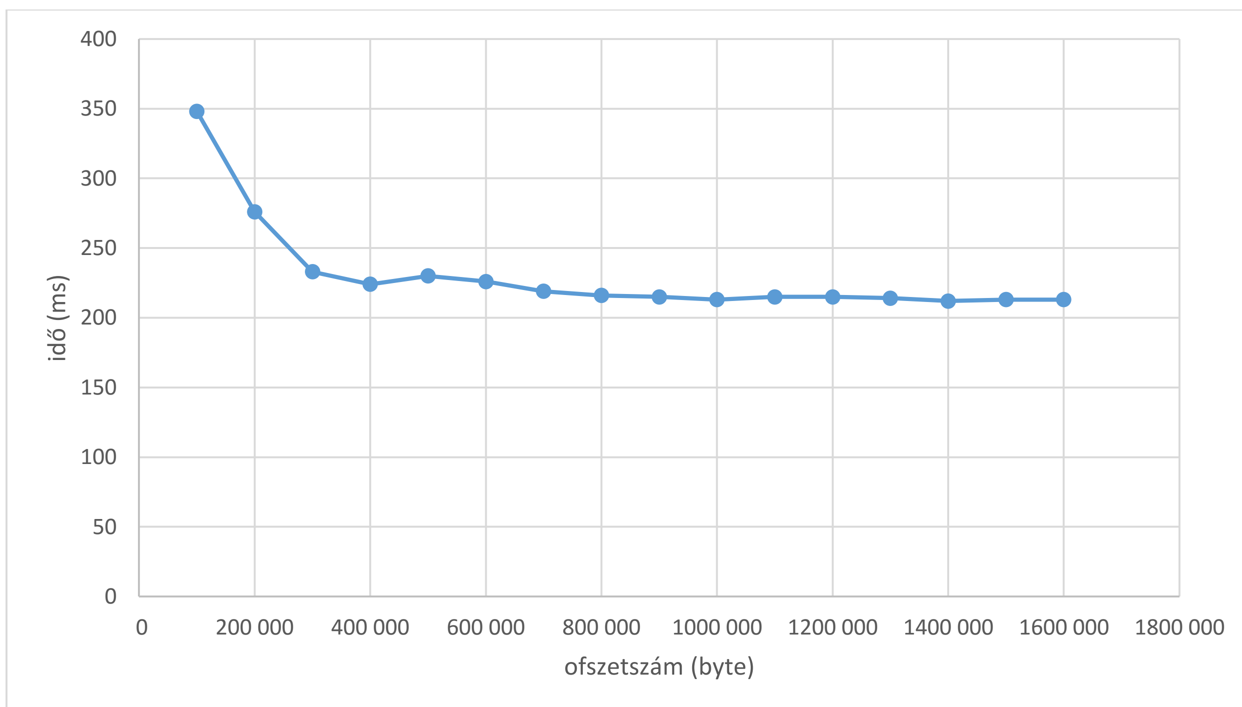
6.2.1 Ofszetszám

Fontos paramétere az algoritmusnak az, hogy egyszerre hány ofszeten keresünk szignatúrákat. Ezen fejezet témája az ofszetszám érték különböző méretű fájlokon való tesztelése. A 2. ábra mutatja az ellenőrzési időt változó ofszetszámokra 128kB méretű bemeneti fájlok esetén.



Ábra 2 – Futási idő különböző ofszetszámokkal 128 KB-os fájlon

Ahogy az a fenti ábrán is látható, az ofszetszám jelentősen befolyásolja a futási időt. Ez elsősorban akkor jelentős, ha a szám túl kicsi.



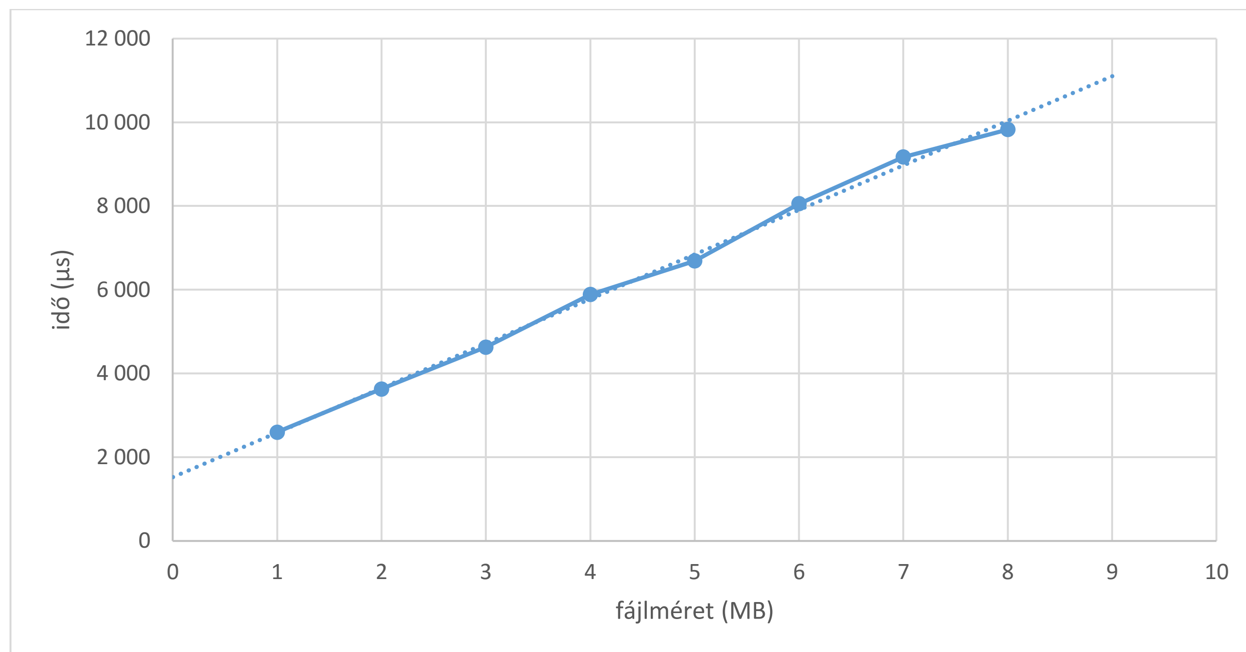
Ábra 3 – Futási idő különböző ofszetszámokkal 200 MB-os fájlon

A 3. ábrán látható, hogy 200 MB-os fájl esetén jóval nagyobb ofszetszámon érhető el az optimális sebesség. Ahogy a fenti ábrán is látszik, 1 MB-nyi ofszet már ideális sebességet biztosít. A 10 MB-os ofszet körülbelül 10%-os lassulást okoz az 1 MB-oshoz képest.

A túl alacsony ofszetszám azért lassítja a keresést, mert ez által a fájlt feldolgozó ciklus többször fut le, mint magasabb ofszetszám esetén. Ez több bufferművelettel és kernelhívással jár, ezek overhead-je mutatkozik meg a mérési eredményben. A szükségesnél magasabb ofszetszám által okozott overheadet az okozza, hogy az algoritmusnak több memóriát kell lefoglalnia az ofszetek eredményeinek a tárolására. Az egyik foglalás GPU globális memória foglalás, a másik pedig pinned rendszermemória foglalás. Ezek a műveletek időigényesebbek, mint az átlagos foglalások, ennek az overhead-je mutatkozik meg az eredményekben.

6.2.2 Adatméret

Ez a fejezet azt teszteli, hogy milyen sebességgel ellenőrizhetők különböző méretű fájlok. A használt ofszetszám 2^{20}.

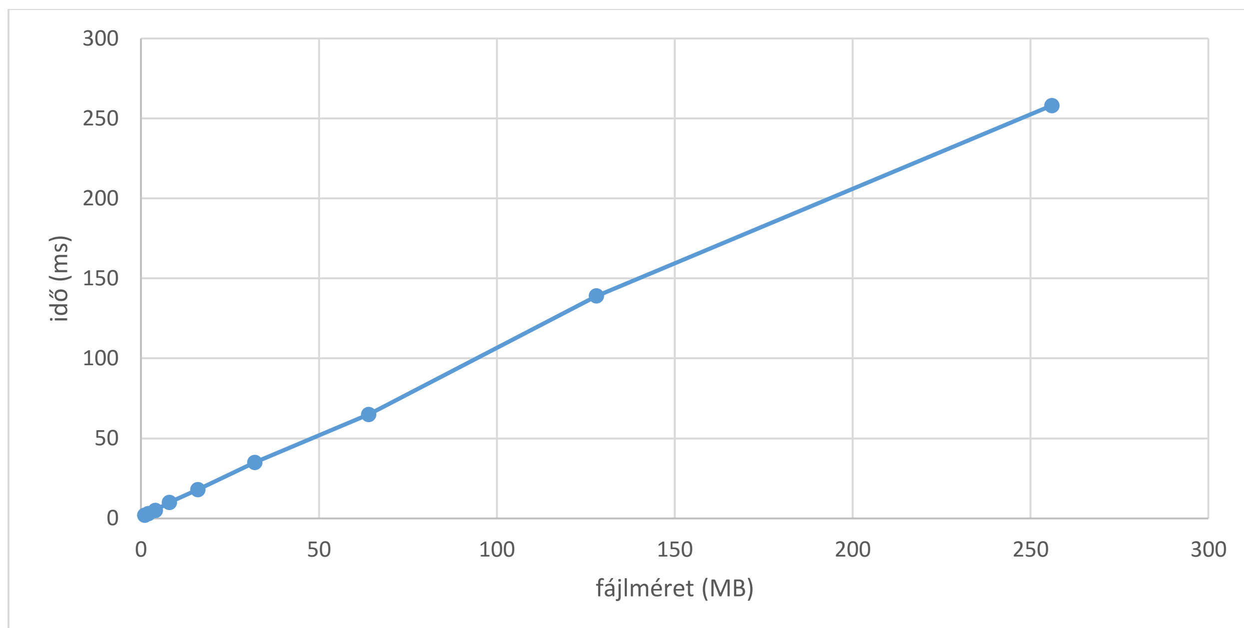


Ábra 4 – 10 MB alatti fájlok

A 4. ábrán látható a futási idő változó fájlméretekre, amely ebben a tartományban lineárisnak tűnik a fájlméret függvényében. A használt ofszetszám 1 MB, azonban a kisebb fájloknál valamivel alacsonyabb az optimális érték. A trendvonal idő tengellyel való

metszéspontján az algoritmus inicializációs overhead-je látszik. Ez kis fájlméreteknél jelentős overhead-et jelent. 1 MB-os fájl esetén a mért feldolgozási idő 2596 µs, azaz az adatáteresztési sebesség 3 Gbps.

16 darab 1 MB-os fájl esetén az ellenőrzési idő 18 774 µs, 128 darab 1 MB-os fájl esetén pedig az érték 132 648 µs. Az előbbi 6.66 Gbps, az utóbbi 7.54 Gbps feldolgozási sebességnek felel meg. Ebből arra következtethetünk, hogy a kis fájloknál jelentkező adatáteresztési sebességcsökkenés az algoritmus inicializációs overhead-jéből adódik, és pár 10 MB-nyi adat ellenőrzése esetén már minimális.



Ábra 5 – Ellenőrzési idő 1, 2, 4, 8, 16, 32, 64, 128, 256 MB fájlméretekre

Nagyob bemeneti adatméret tartományon megismételve a mérést (lásd 5. ábra) bebizonyosodott, hogy a végrehajtási idő közel lineárisan függ a fájlmérettől. Nagyobb, fertőzésmentes fájlok esetén a sebesség 7 Gbps körül mozog. A fenti grafikonban is jelölt 256 MB-os fájlra 258 milliszekundumos ellenőrzési időt mértem. Ez 7.75 Gbps adatáteresztési sebességet jelent. Az 1 MB-os fájl esetében a mért idő 2596 µs, az adatáteresztési sebesség így 3 Gbps. 300 000-es ofszetszámmal ez 3.45 Gbps-re emelhető.

A 4. ábrán láthattuk, hogy az algoritmus nagy relatív overhead-et okoz kis fájlok esetén. Megvizsgáltam tehát egy olyan esetet, amikor sok kis méretű fájlt kell feldolgoznia az

algoritmusnak. Abban az esetben, ha 1024 darab 1 KB méretű fájlt vizsgálunk, a futási idő 70 ms. Ez 114.28 Mbps feldolgozási sebességet jelent.

A mért eredmény megmagyarázható. Az algoritmus úgy akadályozza meg, hogy fájlokon átnyúló szignatúrákat találjon, hogy nem tárol egyszerre több fájlból adatot az adatbufferben. Ebből adódóan 1024 darab kernelhívás szükséges 1024 darab 1 KB-os fájl feldolgozásához. Ehhez persze 2*1024 memóriatranzakció is társul (1024 oda-vissza), így végeredményben hatalmas overhead-et kapunk.

Bár a pár KB méretű fájlok esetén az algoritmus jelentősen lelassul, egy átlagos háttértár esetében ez nem okoz a fenti mértékben lassulást. Ezt a következő példával illusztrálom.

A Windows mappa számos, különböző méretű fájlt tartalmaz, így ezt a mappát használom egy átlagos rendszer háttértárján tárolt fájlok átlagos méretének becsléséhez. A tesztrendszer Windows mappája 27.9 GB, és 109 538 darab fájlt tartalmaz. Ez azt jelenti, hogy a mappában az átlagos fájlméret 267 KB. Létrehoztam 1024 darab 267 KB-os fájlt, és ellenőriztem rajtuk a futási időt. A fájlok mérete együtt 267 MB, az ellenőrzési idő pedig 334 ms. Ez 6.25 Gbps adatáteresztési sebességet jelent. Átlagos méretű fájlok esetén az áteresztőképesség tehát a maximumhoz képest 17%-ot esik.

A mérés során azért nem magát a Windows mappát használtam, mert mivel a mérések erejéig az algoritmus futás előtt a rendszermemóriába tölti az ellenőrzendő adatokat, a program nem tud egy bizonyos méretnél nagyobb fájlokat kezelni. Ez a méret 500 MB körül van, a Windows mappában pedig van egy 779 MB-os Windows patch installer.

A kis fájlok esetén megfigyelt teljesítménycsökkenés természetesen orvosolandó probléma, aminek a megoldásához az algoritmus további módosítása szükséges. A lehetséges megoldások a munka végén, a *7.2* fejezetben olvashatók.

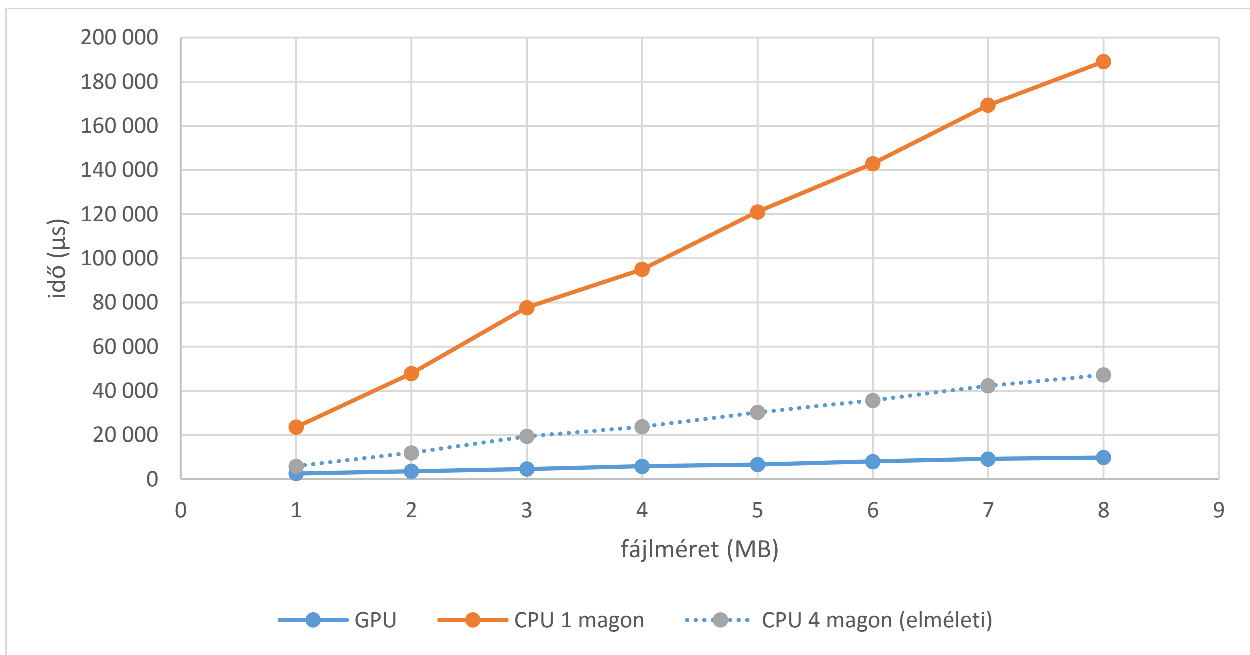
6.3 CPU és GPU összehasonlítása

A munka fő célja egy olyan algoritmus bemutatása, ami a GPU által sikeresen gyorsítható. Ez a fejezet összehasonlítja a hardveresen gyorsított algoritmust a tisztán CPU-n futóval.

A tisztán CPU-n futó algoritmus egy szálon fut. A tesztkörnyezet CPU-ja 4 feldolgozó maggal bír, ezek jó párhuzamosítás esetén megközelítően 4-szeresére növelhetik az algoritmus

CPU-n futó verziójának feldolgozási sebességét. Ezt a teszteléskor úgy fogom figyelembe venni, hogy a mérési eredmények között szerepeltetni fogom a tisztán CPU-n futó algoritmus hipotetikus, a 4 magra 100%-osan párhuzamosított verzióját.

Vessük össze a *6.2.2* fejezetben kapott eredményeket a CPU-n futó algoritmus eredményeivel (6. ábra).



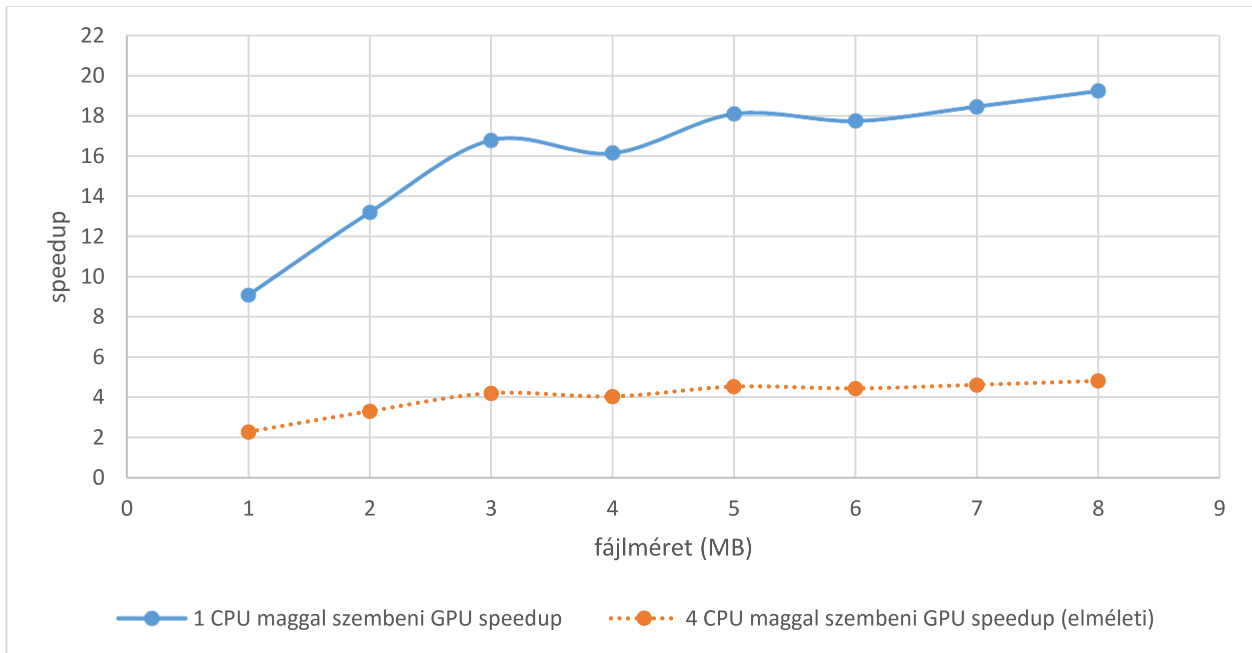
Ábra 6 – 10 MB alatti fájlok – CPU vs GPU

A fenti ábrán láthatók a mérési eredmények, valamint a CPU-n futó egy szálas algoritmus teljesítményéből kiszámolt CPU-ra elméletileg maximális 100%-osan párhuzamosított algoritmus futási ideje. Látható, hogy a GPU valójban gyorsabb a CPU-nál. Természetesen az algoritmus nem párhuzamosítható 100%-osan (például inicializációs overhead, adatbuffer feltöltése). Az összehasonlítás csak arra szolgál, hogy egy körülbelüli kép kialakulhasson a hardveresen gyorsított és a CPU-n futó algoritmus viszonyáról.

Fontos jellemző a két algoritmus futási idejének arányát mutató speedup érték. Ez azt mutatja meg, hogy az egyik algoritmus sebessége hányszorosa a másikénak. Ha a CPU-n futó algoritmusnak a hardveresen gyorsított algoritmussal szembeni speedup-ját akarjuk megtudni, a két algoritmus futási idejének a hányadosát kell képeznünk, azaz

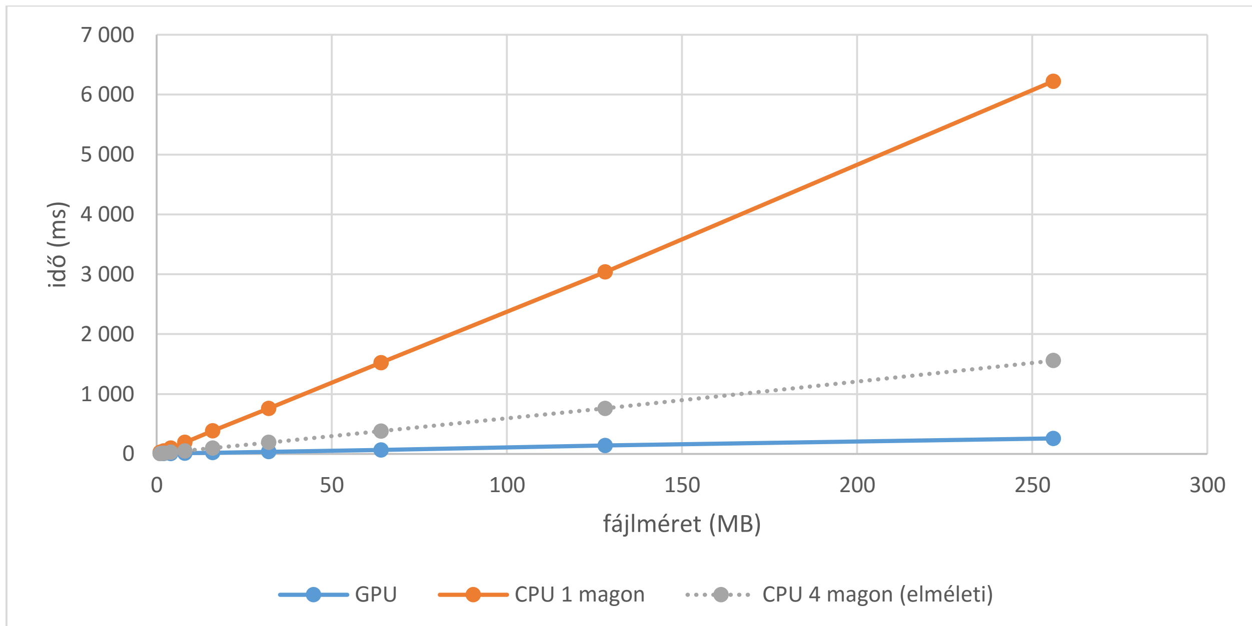
```
speedup = GPU_idő / CPU_idő.
```

Így kapható a 7. ábra.



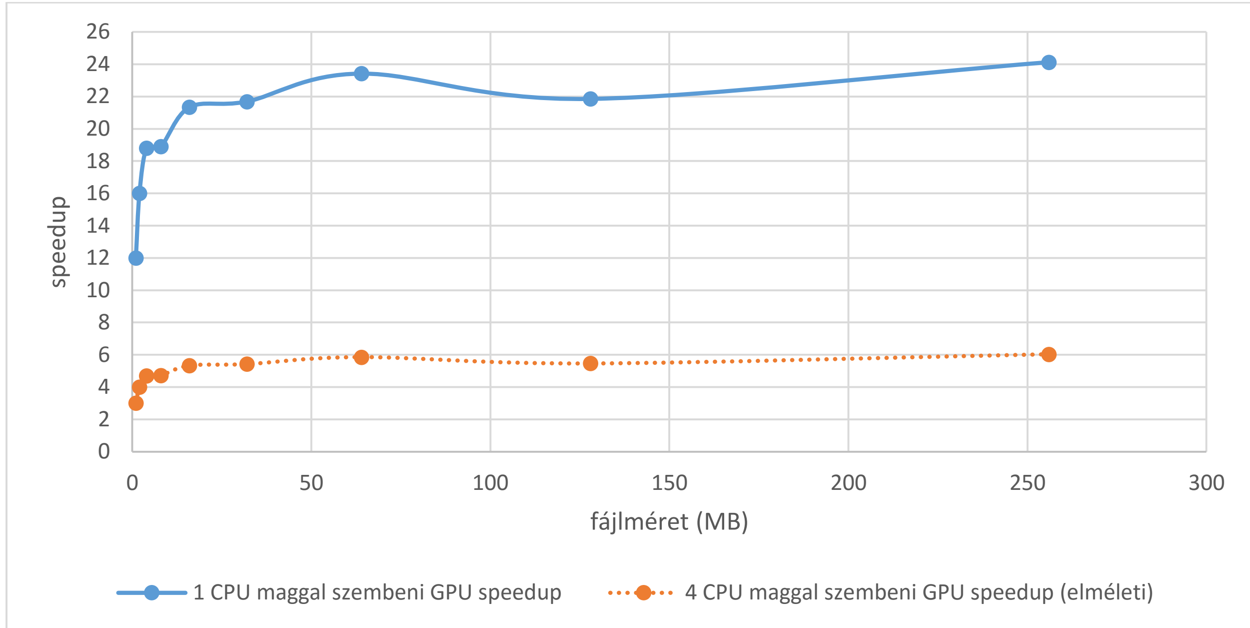
Ábra 7 – 10 MB alatti fájlok – GPU speedup

Látható, hogy a hardveresen gyorsított algoritmus sebessége a sokszorosa a CPU-n futó algoritmusénak. Megvizsgálandó az is, hogy a különböző fájlméretek hogyan befolyásolják a speedup-ot.



Ábra 8 – 1, 2, 4, 8, 16, 32, 64, 128, 256 MB fájlméretek – CPU vs GPU

A 8. ábrán látható eltérő fájlméretekre a CPU és GPU megoldások futási ideje. Ahogy az a fenti ábrán is látható, a GPU nagyobb fájlméreteknél is jobban teljesít. Nézzük a speedup-ot (9. ábra). Az eredmények azt mutatják, hogy nagy fájlok esetén érhető el a maximum speedup.



Ábra 9 – 1, 2, 4, 8, 16, 32, 64, 128, 256 MB fájlméretek – GPU speedup

Az ofszetszám függvényében nem vizsgáltam speedup-ot. Mivel az alacsony ofszetszám növeli a GPU által okozott overhead-et, ez egyértelműen rossz hatással van a speedup-ra, viszont a legtöbb esetben nem érdemes 1 MB ofszet alá menni, úgyhogy ez nem jelent különösebb problémát.

A *6.2.2* fejezetben említett azon szélsőséges esetben, mikor 1024 darab 1 KB-os fájlt kell ellenőriznie az algoritmusnak, a CPU-n futó változat 24 ms alatt teljesítette a vizsgálatot. Ez egyezik azzal az eredménnyel, amit a CPU-s változatnál az 1 MB-os fájl vizsgálata esetén mértem. Ez is azt mutatja, hogy a hardveresen gyorsított algoritmusnál a kis fájlok esetében megfigyelt teljesítménycsökkenést az algoritmus működéséből adódó GPU overhead okozza.

7 Konklúzió

A tervezési fázisban sikerült kifejlesztenem egy olyan szignatúra detekció alapú víruskereső algoritmust, ami nagyszámú szignatúra keresésére képes tetszőleges méretű adathalmazban, és a futási ideje csak minimálisan függ a szignatúra adatbázis méretétől.

Az implementációs fázisban realizáltam az algoritmust, és további optimizációkat hajtottam rajta végre. Az implementációs fázis terméke egy olyan program lett, ami gyors szignatúra alapú detekcióra képes a tesztrendszer háttértárában, és nem rendelkezik egyetlen, más megoldásokban található limitációval sem.

Az algoritmus adatáteresztési sebessége nagy fájlok esetén 7.5 Gbps körül mozog. Ha nagyon sok kis méretű fájlt (pár KB) ellenőriz az algoritmus, az adatáteresztési sebesség a fájlmérettől függően csökkenhet. Átlagos fájlméretek esetén 6 Gbps feletti sebességet mértem.

A program grafikus kártyán futó része jól párhuzamosított, azonban a CPU oldali feldolgozás a jelenlegi implementációban teljesen soros. Több processzor mag felhasználásával jelentős sebességnövekedés érhető el, ennek megvizsgálása a jövőbeli tervek közé tartozik.

Az algoritmus pár KB-os fájlok esetén jelentősen lelassul, az erre vonatozó mérések és az indoklás a *6.2.2* fejezetben olvasható. Az erre a problémára való megoldás keresése jelenleg egyik legfontosabb feladat további munkám szempontjából.

További fontos előrelépés lenne az algoritmus CPU oldali részének olyan fejlesztése, amely lehetővé tenné a joker karaktereket tartalmazó szignatúrák azonosítását.

Irodalomjegyzék

[1] Wikipedia. (2015, October 13). Antivirus software – Identification methods [Online]. Available: https://en.wikipedia.org/wiki/Antivirus_software#Identification_methods

[2] E. Seamans and T. Alexander. (2007). Fast Virus Signature Matching on the GPU [Online]. Available: https://a248.e.akamai.net/f/248/10/10/http.developer.nvidia.com/GPUGems3/gpugems3_ch35.html

[3] A. N. V. Dias. (2012, November). Detecting Computer Viruses using GPUs [Online]. Available: https://fenix.tecnico.ulisboa.pt/downloadFile/395144992995/tese.pdf

[4] N. S. Kovach (2010, September). Accelerating Malware Detection via a Graphics Processing Unit [Online]. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a529467.pdf

[5] Wikipedia. (2015, October 11). Binary search algorithm [Online]. Available: https://en.wikipedia.org/wiki/Binary_search_algorithm

[6] S. Hanov. (2011). Throw away the keys: Easy, Minimal Perfect Hashing [Online]. Available: http://stevehanov.ca/blog/index.php?id=119

[7] D. Belazzougui et al. Hash, displace, and compress [Online]. Available: http://cmph.sourceforge.net/papers/esa09.pdf

[8] L. C. Noll. (2015, January 7). FNV Hash [Online]. Available: http://www.isthe.com/chongo/tech/comp/fnv/index.html

[9] M. Zahran. (2012). Graphics Processing Units (GPUs): Architecture and Programming Lecture 5: CUDA Threads [Online]. Available: http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture5.pdf

[10] M. Harris. (2012, December 4). How to Optimize Data Transfers in CUDA C/C++ [Online]. Available: http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

[11] Wikipedia. (2015, April 22). Polymorphic code [Online]. Available: https://en.wikipedia.org/wiki/Polymorphic_code

[12] Wikipedia. (2015, September 3). Metamorphic code [Online]. Available: https://en.wikipedia.org/wiki/Metamorphic_code

[13] Microsoft (2015). Microsoft Visual Studio [Online]. Available: https://www.visualstudio.com/

[14] NVIDIA (2015). CUDA Toolkit 7.5 [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[15] Boost (2015). Available: http://www.boost.org/

[16] M. Boyer. Choosing Between Pinned and Non-Pinned Memory [Online]. Available: https://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html

[17] M. Boyer. Memory Allocation Overhead [Online]. Available: https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_management_overhead.html

[18] ClamAV [Online]. Available: http://www.clamav.net/