LATVIJAS UNIVERSITĀTE

EKSAKTO ZINĀTŅU UN TEHNOLOĢIJU FAKULTĀTE

DATORIKAS NODAĻA

GPU PROGRAMMĒŠANAS SALĪDZINĀJUMS CUDA, ROCM UN OPENCL SASKARNĒS

KURSA DARBS

Autors: **Artūrs Kļaviņš**

Studenta apliecības Nr.: ak21373

Darba vadītājs: profesors, Dr. dat. Leo Seļāvo

RĪGA, 2024

Anotācija

Darbā gan teorētiski, gan praktiski tiks apskatītas CUDA, ROCm un OpenCL GPU programmēšanas saskarnes. Tiks salīdzināta saskarņu dokumentācija, piedāvātās programmatūras iespējas un ierobežojumi atbalstītajā aparatūrā. Praktiski tiks ieviesta paroļu uzlaušanas un Džona Konveja dzīves spēles programmas katrā saskarnē, apskatīti īstenošanas apsvērumi un analizēta ātrdarbība uz vienas un tās pašas aparatūras.

Atslēgas vārdi: GPU, Nvidia, AMD, CUDA, ROCm, OpenCL

Abstract

Abstract body

Keywords: GPU, Nvidia, AMD, CUDA, ROCm

SATURS

1.IEVADS

Grafiskais procesors vai nu kā atsevišķa vai centrālajā procesorā integrēta komponente ir sastopama gandrīz visos modernajos datoros. Vēsturiski izmantota tikai grafisko elementu apstrādei, kur paralēli veicami daudzi līdzīgi darbi, piemēram, teksta renderēšana, pikseļu aizpildīšana uz ekrāna, 3D ēnotāju funkcijas.

Kļuva skaidrs, ka GPU augstās paralelizācijas iespējas varētu izmantot citos uzdevumos, kuri klasiski pildāmi uz CPU. Rezultātā tos būtu daudz efektīvāk pildīt uz GPU, palielinot programmu ātrdarbību.

Pirms moderniem ietvariem, saskarnēm un arhitektūras atbalstu, ar kuru palīdzību uz GPU iespējams skaitļot principā jebko, programmētājiem vajadzēja atrast 'nestandarta' risinājumus, lai pildītu ne-grafiskas problēmas. Piemēram, 2003. gadā radās risinājums kā skaitļot vispārējus lineārās algebras vienādojumus, pārveidojot matricu datus kā tekstūras un uz tām izpildot ēnotājus. [1]

Līdz ar to radās pieprasījums pēc plašlietojamas skaitļošanas uz grafiskajiem procesoriem. GPU ražotāji to sāka ņemt vērā un 2006. gadā Nvidia ieviesa CUDA API platformu ar tiešu tās atbalstu uz Nvidia videokartēm, sākot ar "Tesla" GPU mikroarhitektūru.[2]

Nvidia videokartes sākot jau no tā paša 2006. gada ir bijušas tirgus līderes, un tādas ir vēl joprojām. AMD nepalīdzēja fakts, ka savu ROCm platformu ieviesa daudz vēlāk, tikai 2016. gadā, kā tieši konkurentu CUDA, kad jau CUDA bija praktiski pierādīta un lietota 10 gadus.

Nvidia videokartes un CUDA dominē kā populārākā GPU izvēle un plašlietojuma GPU skaitļošanas (GPGPU) platforma. No digitālās izplatīšanas platformas "Steam" 2024. gada decembra aparatūras un programmatūras aptaujas var secināt, ka 75% "Steam" lietotāju izmanto Nvidia videokartes, bet tikai 16% - AMD. [3]

Bet ROCm programmatūras steks ar GPU programmēšanas, dziļās mašīnmācīšanās, HPC iespējām līdzinās pieejamajās iespējās ar CUDA. Tāpēc šī darbā mērķis ir salīdzināt abas platformas, to pieejāmas salīdzināmās funkcijas, ātrdarbību un iespējamās priekšrocības, izvēloties vienu vai otru.

2.PIERAKSTI PAR OPENCL

Izskatās, ka ir diezgan zems apjoms ar modernām OpenCL 3.0 pamācībām, izņemot tehniskās specifikācijas, kuras iesācējām varētu nebūtu piemērotas.

Jaunākā pamācības literatūra ir priekš opengl 2.0, iznāca 2015 gadā [4](debetable vai jaunākā, bet zināmākā)

OpenCL nav beginner friendly, pārsvarā paredzēts izstrādātājiem, kuri jau labi spējīgi orientēti precīzi tehniskās specifikācijās, lai rakstītu platform-neaktarīgas programmas priekš CPU, GPU u.c. hardware paātrinātājiem

Nvidia izskatās, ka ir OpenCL atbalsts

Izmanto SPIR starp-posma reprezentācijas (intermediate represenatation) valodu, kas izmantota arī citās Khronos Group valodās, ietvaros - Vulkan, SYCL

Labi atsaukties uz specifikāciju [5]

Varbūt derētu realizēt benchmarkingu ar profilēšanas rīkiem, lai precizāk noskaidrot dažādās 'veiktspējas' dažādos izpildes posmos, piemēram:

- atmiņas iedalīšana uz gpu,
- vendor atrašana un iespējams kodola kompilēšana (opencl gadījumā ig)
- kodola izpilde,
- atmiņas atbrīvošana

Kādas varētu būt atšķirības programmēšanas modelī starp CUDA, HIP, OpenCL? Idejiski jau mērķa arhitektūra ir apmēram vienāda, līdz ar to liekas, ka modelim ar tādam vajadzētu būt.

2.1 OpenCL platformas modelis - pieraksti pa taisno no dokumentācijas

Sastāv no saimnieka (CPU) ar vienu vai vairākām OpenCL iekārtām (GPU).

OpenCL iekārta ir iedalīta vienā vai vairākās *compute* vienībās (Compute Units), un tās ir iedalītas apstrādes elementos (processing elements)

Skaitļošana notiek šajos apstrādes elementos

OpenCL programma sastāv no saimnieka koda un iekārtas koda. Saimnieka koda daļa nodod kodola (kernel) kodu OpenCL iekārtai un iekārta to izpilda uz tās apstrādes elementiem.

Kad apstrādes elementi apstrādes vienībā izpilda to pašu secību ar priekšrakstiem (? varētu labāk izvārdod), tad vadības plūsma ir saucama par kopdarbīgu (converged)

Kopdarbīga vadības plūsma ir labi piemērota uz tādas aparatūrus kā GPU, kura ir specializēta vienas instrukciju kopas izpildei paralēli uz vairākiem apstrādes elementiem. Nav grūti izsecināt, ka uz šādas aparatūras noteiktas OpenCL programmas izpildīs konkrētus uzdevumus ātrāk par līdzīgu risinājumu uz saimnieka - CPU.

Iekārtas kodola kodu ir iespējams sniegt kā OpenCL C99 pirmkoda simbolu virkni, SPIR-V starpvalodu vai bināru objektu. OpenCL piedāvā kompilatoru, kas spējīgs no minētajiem formātiem izveidot izpildāmo programmas objektu.

Kompilators var būt 'tiešsaistes' vai 'bezsaistes' jeb

- Tiešsaistes kompilators ir pieejams saimnieka programmas izpildes laikā
- Bezsaistes tiek izsaukts atsevišķi un saimnieka programmai tiek nodots un ielādēts gatavs izpildāms fails

Izskatās, ka salīdzinot ar HIP un CUDA, te ir lielāka brīvība veidos kā realizē kodola programmas kompilēšana un palaišanu, bet derētu paskatīties vairāk par šo tajās platformās

Ņemot vērā kā GPU arhitektūru apraksta Nvidia un AMD, šis OpenCL platformas modeļa abstrakcijas slānis ir raksturojams kā 'tuvu dzelžiem'. Tomēr kodola kompilatoram ir diezgan brīva izvēle optimizācijās starp faktiskajiem arhitektūras elementiem un kā tos reprezentē OpenCL.

Piemērs, šim faktam ir situācijas, kur iekārtas kodola programma tiek kompilēta priekš CPU. Centrālais procesors arhitektūras līmenī var nesaturēt vairākus kodolus vai pavedienus, kurus programmētājs sagaida izstrādājot iekārtas kodolu.

Vai arī, palaižot kodolu, tiek definēts pavedienu grupas izmērs (piemēram, izmērā 128), kas uz visām iekārtām nebūs pieejams tādā skaitā. Kompilators šo situāciju atrisinātu, izsaucot kodolu vairākās partijās, bet no pirmkoda puses tiek definēta viena partija.

Te arī jāpārliecinās, bet man liekas, ka HIP un CUDA pie nederīga grupas/pavediena

izmēra izmestu izpildes laika kļūdu, OpenCL dod lielāku brīvību

3.ETALONUZDEVUMA REZULTĀTU LĪDZVĒRTĪBA

Salīdzinot dažādus ietvarus ar it kā vienu un to pašu uzdevumu nav tik vienkārši, jo ar naivu risinājumu ir grūti garantēt, ka attiecīgās izveidotās programmas ir pietiekami līdzvērtīgas, lai tās varētu godīgi salīdzināt.

Piemēram, salīdzinot dažādu programmēšanas valodu veiktspēju ar noteiktu etalonuzdevumu, varētu izvēlēties valodas C++ un Python, apstrādājot kādu failu caur standarta ievadi, aprēķinot cik rindas satur dotais fails (skatīt izdruku 3.1 un 3.2)

Izdruka 3.1: Vienkārša faila apstrāde valodā C++ caur standarta ievadi

```cpp
#include <cstdio>
#include <iostream>
#include <string>

int main()
{
  std::string line;
  size_t lineCount = 0;

  while (std::getline(std::cin, line))
  {
    lineCount++;
  }

  printf("Fails satur %zu rindas\n", lineCount);

  return 0;
}
```

Izdruka 3.2: Vienkārša faila apstrāde valodā Python caur standarta ievadi

```python
import sys

def main():
    line_count = 0
    input_line = ''

    for line in sys.stdin:
        input_line = line
        line_count += 1

    print("Fails satur " + str(line_count) + " rindas");

if __name__ == "__main__":
    main()
```

Pārbaudot izpildes laikus, piemēram, ar failu ar 100 miljoniem rindu, izmantojot time utilītu, var iegūt rezultātus, kuri skatāmi izdrukā 3.3.

Izdruka 3.3: Etalonuzdevuma rezultāti failam ar 100 miljoniem rindu

```
$ time ./cpp_benchmark < 100mil.txt
Fails satur 100000000 rindsas

real    0m25.787s
user    0m24.884s
sys     0m0.892s

$ time python py_benchmark.py < 100mil.txt
Fails satur 100000000 rindsas

real    0m7.129s
user    0m6.303s
sys     0m0.819s
```

Tātad pēc iegūtājiem datiem (25,787s priekš C++ un 7,129s priekš Python) varētu secināt, ka Python ir 3,6 reizes veiktspējīgāks nekā C++, kas sarežģītākos piemēros būtu nepiemērots apgalvojums. Līdz ar to, dotajam piemērēram kā vispārīgam dažādu valodu salīdzinājumam ir problēmas:

- reālos programmēšanas scenārijos reti pārādīsies nepieciešamība risināt minēto problēmu,
- etalonuzdevums nesatur darbības ar sarežģītākām datu struktūrām un algoritmiem, kuras drīzāk parādītos netriviālās programmās
- un iespējams vissvarīgāk - augsta līmeņa valoda kā Python abstraktē relatīvi sarežģītas darbības ar failiem, datu buferiem, standarta ievadi / izvadi, ieviešot optimizācijas izpildes laikā, tāpēc dotos risinājumus nevar uzskatīt par ekvivalentiem.

C++ risinājumā ieviešot pat ļoti vienkāršas optimizācijas konfigurācijā ar standarta ievadi, var iegūt krietni labāku rezultātu (skatīt izdruku 3.4 un 3.5).

Izdruka 3.4: Optimizēta vienkārša faila apstrāde valodā C++ caur standarta ievadi

```
#include <cstdio>
#include <iostream>
#include <string>

int main()
{
    std::ios::sync_with_stdio(false); // nesinhronize C un C++ stdio
```

```
    std::cin.tie(nullptr); // neflusho cout, kad tiek lietots cin

  std::string line;
  size_t lineCount = 0;

  while (std::getline(std::cin, line))
  {
    lineCount++;
  }

  printf("Fails satur %zu rindas\n", lineCount);

  return 0;
}
```

Izdruka 3.5: Optimizētā C++ etalonuzdevuma rezultāti failam ar 100 miljoniem rindu

```
$ time ./cpp_test < 100mil.txt
File contains 100000000 lines

real    0m2.002s
user    0m1.690s
sys     0m0.308s
```

varbūt jāmin ka kādu laiku atpakaļ soc tīklos bija populārs benchmarks apstrādāt 1 miljardu rindu garu failu - līdzīgs problemātisks benchmarks

Līdz ar to, darbojoties ar etalonuzdevumiem, jāņem vērā, ka var iegūt nepatiesu līdzvērtību

4.GPGPU ARHITEKTŪRA UN PROGRAMMĒŠANA

Procesi, kas izmanto GPU resursus, tipiski iedala atmiņu un sagatavo datus uz CPU, un tad nodod darbu grafiskajam procesoram. Atmiņas iedalīšanas varianti ir stipri atkarīgi no veicamā uzdevuma un, būtiskāk, no pieejamā GPU arhitektūras.

Vecākās videokartēs atmiņas iedalīšanu strikti veica CPU un to atmiņas ir pavisam atdalītas (RAM un VRAM - Video RAM). Modernāki risinājumi spējīgi pielietot vienotu atmiņas apgabalu (CPU un GPU abi var piekļūt viens otra atmiņai), piemēram, Nvidia kartes sākot ar Pascal mikroarhitektūru [2]. Rezultātā programmētājam nav jāpārvalda, kuras adreses ir centrālā un kuras grafiskā procesora. Vēl arī jāapsver, vai ir iespēja pa tiešo no GPU iedalīt atmiņu, vai to darīs tikai CPU.

Protams, noteiktus abstrakcijas slāņus zemāk, procesoru atmiņas tomēr dzīvo dažādās vietās un nav apvienojami, izņemot procesorus, kuros CPU un GPU dzīvo vienā čipā (integrētās videokartes).

Uz GPU izpildāmā programma tiek saukta par kodolu (no angļu val. *kernel*), un no CPU ar draiveru palīdzību tiek nodots:

- uz GPU pavedieniem izpildāmais kodols,
- pavedienu skaits,
- kodola funkcijas argumenti (visbiežāk tās būs apstrādājamo datu atmiņas adreses - rādītāji).

Jāpārveido, lai nav grāmatas tulkojums: GPU sastāv no daudziem kodoliem un katrs kodols izpilda SIMT (*Single instruction, multiple threads*) modelim atbilstošu iedoto izpildāmo kodolu

Derētu tad minēt šādas lietas

- Augsta darbu paralelizācija lielo kodolu skaitu dēļ
- Īpašas instrukcijas konkrētu datu apstrādei
- Darbus nodod procesors (kaut kā īsti nezinu kā) GPU un GPU izmet atpakaļ rezultātu vai prasīto uzzīmē uz ekrāna

- Darbus var nodot GPU caur saskarnēm kā Nvidia CUDA vai AMD ROCm

SIMT kodoli - single instruction, multiple threads. Sadalās SIMT frontendā un SIMD (multiple data) backendā

SIMT steks, lai atbilstītu zarošanos

SIMT deadlock - paveidiens gaida uz atomicCAS, tālāk neies kamēr neizpildīs (while (!atomicCAS) ...), kad to dara vairāki paveidieni, tad var notikt deadlock

5.PLATFORMU SALĪDZINĀJUMS

Lai piesaistītu GPU programmētājus, AMD jau no sākuma dizainēja ROCm, lai tā līdzinātos CUDA. Protams, abas platformas ir paradzētas GPU programmēšanai un apakšējā videokaršu arhitektūra nebūs tik atšķirīga. Galvenā atšķirība, neskaitot platformu mērķa grafiskos procesorus, ir fakts, ka atšķirībā no CUDA, kura ir slēgtā, ROCm ir atklātā pirmkoda programmatūra, līdz ar to, ja nepieciešams, visu programmatūras saturu var izpētīt, modificiēt, kompilēt pats, kā arī dot savu pienesumu gan dokumentācijā, gan kodā.[6]

ROCm dokumentācija ir ar savām problēmām, piemēram, lai atrastu instalāciju nākas diezgan dziļi meklēt un 'lēkāt' starp lapām, lai atrastu konkrēto instalācijas failu. Instalācijas pamācības un lejupielādes lapas ir diezgan sadalītas. CUDA šis process ir vienkāršāks, kā arī CUDA ir pastāvējusi daudz ilgāku laiku, līdz ar to pieejamā literatūra, forumu diskusiju skaits ārpus oficiālajām dokumentācijām ir daudz lielāks nekā priekš ROCm.

ROCm satur vairākas programmas, bibliotēkas un ietvarus dažādiem darbiem ar augstas veiktspējas, paralelizācijas skaitļošanu, bet konkrētais C++ API priekš GPU programēšanas ir HIP (no angļu val. *Heterogeneous-computing Interface for Portability*).[7]

Noteiktas HIP dizaina izvēles ir tieši aizņemtas no CUDA, lai CUDA vidē pieredzējušajiem izstrādātājiem pāriet uz ROCm būtu vieglāk. Piemēram, C++ dekoratori, kuri norāda vai funkcija ir CPU vai GPU, vai GPU kodola funkcija ir vienādi (skatīt izdruku 5.1).

Izdruka 5.1: CUDA un HIP funkciju definīciju salīdzinājums

```
// CUDA:
__host__ myCpuFunction() {/*...*/}
__device__ myGpuFunction() {/*...*/}
__global__ kernel() {}

// HIP:
__host__ myCpuFunction() {/*...*/}
__device__ myGpuFunction() {/*...*/}
__global__ kernel() {/*...*/}
```

HIP ir diezgan liels atbalsts ne tikai AMD videokartēm, bet arī Nvidia. Tas iespējams tāpēc, ka daudzas HIP saskarnes ir CUDA savietojamas, piemēram, GPU matemātisko funckiju API, tās saturošās funkcijas ir tieši atbilstošas CUDA funkcijām.[8, 9]

Kompilēšanas līmenī šis atbalsts ir iespējams, jo HIP izmanto kompilatoru draiveri 'hipcc', kurš, atkarībā no platformas, veiks pirms-apstrādi un izsauks attiecīgo kompilatoru - AMD videokartes gadījumā 'amdclang++' un Nvidia CUDA - 'nvcc'. [10].

Tā kā varētu interpretēt, ka CUDA ir tieša apakškopa ROCM un HIP platformai, bet tomēr pilnīgs atbalsts visām CUDA funkcijām nav pieejams.

Tā kā, rakstot CUDA kodu, arī tiek izmantots tas pats 'nvcc' kompilators, ROCm piedāvā utilītu pirmkoda migrēšanai - 'HIPIFY'. [11]

Potenciāls mīnuss HIP saistītām utilītām un bibliotēkam ir fakts, ka vairākām nav pieejama "*out of the box*" instalācija. Ir nepieciešamība kompilēt pirmkodu pašam, šo papildus soli sarežģī atkarīgo pakešu pārvaldīšana. Windows gadījumā arī rodas sarežģījumi, jo ROCm utilītu noklusētā vide ir Linux un visa ROCm izmantotais kompilators ir Clang/LLVM, kam uz Windows ir nepieciešama papildus konfigurēšana. Piemērs šādai utilītai ir "HIPIFY", ar kuru iespējams pārveidot CUDA pirmkodu uz HIP. [11]

Tā kā CUDA ir slēgtā pirmkoda, tad, protams, visa programmatūra un rīki pieejami caur instalācijām un rezultātā šis process ir daudz vienkāršāks.

Lai gan HIP atbalsta ir diezgan liels ierobežojums atbalstītajām videokartēm

ROCm paradzētā vide ir Linux, bet CUDA tā ir Windows. ROCm dokumentācijā, kur nepieciešams darbs ar komandrindas rīkiem, pamācības ir tikai Linux operētājsistēmai. CUDA gadījumā problēmas rodas jau instalācījas brīdī darbā ar Linux, jo nepieciešamas papildus darbības, konfigurācijas atkarībā no distributīva, izmantotā pakotņu pārvaldnieka sistēmas.

Atšķirībā no AMD, CUDA neizplata informāciju par savām videokartēm un tā kā Linux pats par sevi ir atklātā pirmkoda programmatūra, tā nenāk ar Nvidia draiveriem. Tomēr projekts nouveau ar reversēs inženierijas metodēm cenšas piedāvāt atklātā pirmkoda draiverus nvidia videokartēm

Bet, lai izmantotu CUDA šis risinājums neder, jebkurā gadījumā nāksies overridot visus ne-Nvidia izlaistus draiverus. Linus |TOrvalds par arī ir izteicies, ka nosoda šādu kompānijas politiku.

ROCm nav iekļauts populārākajās pakotņu pārvaldnieku sistēmās, lai gan ar automātisko instalāciju tiek iekļauts visa nepieciešamā programatūrā darbam ar ROCm programmām, problēmas rodas saskarnē ar CUDA, jo nākas atrisināt atkarību problēmas. Noklusēti tiks meklēta cuda atkarība distributīva izmantotajā pārvaldniekā, piemēram, Ubuntu "apt". Bet, šī atkarība satur tikai CUDA draiverus, nevis izstrādes rīkus, lai veiktu darbu ar CUDA Toolkit.

Līdz ar to ir manuāli jāatrisina neatbilstošas pakotņu atkarības.

Kopumā AMD mērķis ar ROCm, atbalstot konkurent-kompāniju, ir sniegt gala lietotājiem, izstrādātājiem vieglāku pārēju uz AMD platformu un videokartēm. Uzturot funkcionalitāti platform-neatkarīgu un programmatūru rakstot HIP platformā, izstrādātāji ir spējīgi atbalstīt abu platformu videokartes.

Rezultātā kompānijas un gala lietotāji, sastādot datoru, serveru specifikāciju, varētu neuztraukties par platform-atkarību, jo uzņēmumam svarīgā programmatūra, piemēram, mašīnmācīšanās bibliotēkas strādātu bez problēmām gan uz Nvidia, gan AMD videokartēm.

AMD gadījumā, protams, ka labāka situācija būtu, ka tiktu izvēlēta viņu ražota videokarte. Un šādā teorētiskā scenārijā tāds arī būtu iznākums, jo aptuveni vienādas specifikācijas videokartes starp abiem ražotājiem ir ar lielu cenas starpību.

TABULA AR SPECIEM UN CENU

Bet, tā kā CUDA parādījās pirmā, kļuva par industrijas standartu un agrāk citu risinājumu nebija, mūsdienās plaši lietota programmatūra ir rakstīta uz CUDA. Piemēram, mašīnmācīšanās bibliotēka TensorFlow. [12]

Migrēšana lielos projektos tomēr nav tik vienkāršs process, un ar šobrīdējo lielo AI pieprasījumu

6.UZDEVUMS

Praktiskai platformu salīdzināšanai tomēr būtu nepieciešama videokarte. Darba izstrādes laikā bija pieejams portatīvais dators ar:

- AMD Ryzen 5600H procesoru ar Radeon Graphics integrēto videokarti
- Nvidia GeForce RTX 3060 Laptop ārējo videokarti

Lai gan it kā viena datora ietvaros būtu pieejamas gan Nvidia, gan AMD videokartes, diemžēl integrētajai Radeon Graphics kartei nav ROCm atbalsts, kā arī HIP nav oficiāls atbalsts Nvidia videokārtēm uz Windows, tikai Linux. Tāpēc par pamata platformu analīzē tiks izmantota CUDA.

Iespējams risinājums ir WSL - *Windows Subsystem for Linux*, ar kuru iespējams izmantot Linux paredzētās programmas uz Windows. Gan CUDA, gan ROCm ir atbalsts un pamācības kā sakonfigurēt šīs platformas darbam uz WSL.[13, 14]

Bet ņemot vērā, ka HIP atbalsta arī CUDA, būs iespējams apskatīt HIP iespējas uz Nvidia videokartes.

Jāizvēlas tāds uzdevums, kuru iespējams 'augsti' paralelizēt, tas ir, sadalīt uzdevumu daudzos mazos gabalos (vēlams skaitā >= 1 miljons), lai būtu pievienotā vērtība (ātrdarbība) to pildīt uz GPU, ņemot vērā papildus darbu un nepieciešamās zināšanas, lai ieviestu GPU risinājumu.

Relatīvi vienkāršs, bet pietiekams uzdevums, lai parādītu ietvaru atšķirības un vispārīgi atšķirīgās paradigmas starp CPU un GPU programmēšanu būtu paroļu lauzējs.

Nobriedušāki paroļi lauzēji, jeb precīzāk - paroļu atkopēji kā HashCat ir spējīgi ņemt vērā vairākas paroļu variācijas, maskas, faktus, ka parole, piemēram, iesākusies ar "123" un citas sarežģītākas potenciālo paroļu iegūšanas metodes. Demonstrācijas vajadzībām pietiktu izstrādāt pašu 'kodolu', kas, ar jau saņemtu iespējamo paroļu sarakstu, tās pārbaudīs.

Tātad jāizstrādā programma, kas ņemot vērā ieejas failu ar potenciālajām parolēm, no kādas paroles jaucējvērtības spētu noskaidrot 'hešoto' paroli. Uzdevumu būtu vērts risināt uz GPU, jo pie liela paroļu skaita, katrs GPU kodols varētu rēķināt savas paroles jaucējvērtību un pārbaudīt to pret uzlaužamo.

6.1 Izstrādājamās programmas definīcija

Programma ir domāta kā komandrindas utilīta ar CLI argumentiem:

- testu palaišanais arguments
- ievades faila ceļš,
- paroles jaucējvērtība.

Apstrādājamais fails saturēs potenciālās paroles, katra savā rindā, kurām tiks izrēķināta jaucējvērtība un salīdzināta pret doto. Izmantojamais jaukšanas algoritms - SHA256. Jaucējvērtību rēķināšana un salīdzināšana jārealizē izpildei uz GPU.

SHA256 izvēlēts tā tīri tā popularitātes un relatīvi ātrās izpildes dēļ, paroles uzlaušanas demonstrācijas vajadzībām ar šo pietiek, nepieciešamības gadījumā iespējams ieviest citu jaukšanas algoritmu un aizstāt ar esošo.

Programmu iespējams palaist ar:

- `$ pwCracker --test`, lai vispārīgi pārbaudītu programmas darbību pret zināmām jaucējvērtībām un to attiecīgajiem ziņojumiem, kā arī, lai pārbaudītu GPU kodola programmas darbību, platformas pieejamību uz šī datora apartūras
- `$ pwCracker <paroļu faila ceļs> <paroles hash vērtība>`, lai veiktu galveno programmas izpildi

6.1.1 CUDA risinājums

Lai gan CUDA ir domāta izstrādei uz C++, izpildāmais kods uz GPU ir ar saviem ierobežojumiem, kas neļauj pielietot C++ standarta bibliotēkas funkcijas, jo tās iekš 'CUDA device code' nav implementētas. Vai nu tāpēc, ka kāda konkrēta funkcija ir visparīgi reti lietota, vai tā nav īsti paredzēta lietošanai iekš GPU.

Piemēram, nav pieejamas std::string, std::vector struktūras, jo tās dinamiski iedala atmiņu, kas darbībā iekš CUDA kerneļa būtu ļoti lēna darbība. Tā kā paroles par nelaimi ir simbolu virknes, tad tās nāksies apstrādāt C stilā (char*), vai arī kā baitu masīvus (uint8_t*).

Izstrādē ir jāvelta arī palielināta uzmanība datu struktūrām, to izvietojumam atmiņā, piemēram, problēma kā glabāt pārbaudāmo paroļu sarakstu. Naivais risinājums būtu masīvs ar adresēm, kuras norāda uz pirmo simbolu konkrētās paroles simbolu virknē.

Problēma šajā risinājumā ir, ka netiek nodrošināta atmiņas blakusnodalīšana, tas ir, vienā nepārtrauktā atmiņas apgabalā, jo konkrētā parole var teorētiski atrasties jebkur, paroles savstarpēji nav obligāti viena otrai blakus. Līdz ar to varētu rasties lēndarbība no konkrētā pavediena izpildes kodolā - katrs pavediens savu paroli meklētu teorētiski patvaļīgā vietā, kas var radīt kešatmiņas netrāpījumus.

Arī datu kopēšanas uz GPU atmiņu būs lēna, nāksies izmantot vairākas cudaMemcpy instrukcijas, katrai parolei.

Labāks risinājums ir izmantot nepārtrauktu paroļu buferi un katrai parolei fiksēt garumu, tām liekot galā nuļļu papildinājumus.

Jāņem vērā arī definētie kerneļa bloku un pavedienu skaiti ..

A reasonable minimum target is to launch a total number of threads of at least of SM * 2048.

https://forums.developer.nvidia.com/t/cuda-sha256-calculations-improvements/56757/4

These should be split between blocks with usually something in the range of 128,256, or 512 threads per block. It might be that 1024 threads per block is “OK”, it just requires some analysis to confirm.

Your GTX970 has 13 SMs, so target 13*2048 = 26K threads, ballpark, minimum. If you put 1024 threads per block, that would be 26 blocks. I’m not saying I know how to transform your code from 4 blocks to 26, but that is a reasonable performance goal, to maximize throughput.

```
__global__ void kernel(
    const cuda::std::uint8_t *passwords,
    const int *pwLengths,
    int pwCount,
    cuda::std::uint64_t maxPwLength,
    const cuda::std::uint8_t *targetHash,
    int *resultIndex
)
{
    int idx = blockIdx.x * blockDim.x +threadIdx.x;

    if(idx >= pwCount)
    {
        return;
```

```
15      }
16
17      const cuda::std::uint8_t *password = passwords + (idx *
            maxPwLength);
18      int pwLength = pwLengths[idx];
19
20      cuda::std::uint8_t hash[32];
21
22
23      sha256(password, pwLength, hash);
24
25      // ja hashi īsakrt, tad āierakstm paroles indeksu šiek '
            resultIndex',
26      // āt āk āt īāmaing adrese atrodas device īākopgaj ņāatmi, ājlieto
            atomiska funkcija
27      if(compareHashes(targetHash, hash))
28      {
29          // ja resultIndex āglabjas ēīvrtba -1, tad āaizstj to ar idx
30          // ar šo tiek īar ēreiz šānodroints, ka ja ākds pavediens
                ēāvlk ētomr ānonk īldz šim āstvoklim,
31          // tad āmodifikcija netiks veikta, jo ātaj īībrd jau '
                resultIndex' != -1
32          atomicCAS((unsigned int*)resultIndex, -1, (unsigned int)idx);
33      }
34  }
```

```
nvcc -O2 kernel.cu -o main.exe
```

6.1.2 Portēšana uz HIP caur WSL 2

6.1.3 Analīze un secinājumi

Ir jāpievērš uzmanība mērķa videokartei un mikroarhitektūrai, jo izstrādājot programmu uz personīgā datora ar RTX 3060 videokarti bija pieejamas funkijas, kuras testējot uz cita datora ar Quattro sērījas videokarti, programma nestrādāja.

Ir situācijas, kad šo problēmu var atrisināt, kompilācijas brīdī definējot mērķa arhitektūras, bet, ja izmantota kāda konkrēta funkcionalitāte, kura vienkārši nav pieejama uz mērķa, programma nestrādās.

Veiktspējas metrikas

Vēl ir jāņem vērā virsdarbe, kopējot datus no iekārtas uz CPU un atpakaļ, šāda darbība ir relatīvi dārga

7.PLATFORMNEATKARĪGI RISINĀJUMI

ZLUDA, OpenCL

BIBLIOGRĀFIJA

[1] J. Krüger un R. Westermann, ACM Trans. Graph. **22**, 908—916 (2003).

[2] Nvidia, *NVIDIA Tesla P100*, (2016) https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, (Piekļūts: 03.01.2025).

[3] Valve Corporation, *Steam Hardware & Software Survey: December 2024*, https://store.steampowered.com/hwsurvey, 2024, (Piekļūts: 05.01.2025).

[4] D. Kaeli, P. Mistry, D. Schaa un D. P. Zhang, izdev., *Heterogeneous Computing with OpenCL 2.0* (Morgan Kaufmann, Boston, 2015).

[5] Khronos OpenCL™Working Group, *The OpenCL™Specification*, https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html, 2024, (Piekļūts: 02.04.2025).

[6] Advanced Micro Devices, *What is ROCm?*, https://rocm.docs.amd.com/en/latest/what-is-rocm.html, 2024, (Piekļūts: 05.01.2025).

[7] *HIP documentation*, (Piekļut: 04.01.2025).

[8] Advanced Micro Devices, *HIP math API*, https://rocm.docs.amd.com/projects/HIP/en/latest/reference/math_api.html, 2024, (Piekļūts: 05.01.2025).

[9] Nvidia, *CUDA Math API Reference Manual*, https://docs.nvidia.com/cuda/cuda-math-api/index.html, 2024, (Piekļūts: 05.01.2025).

[10] Advanced Micro Devices, *HIP Compilers*, https://rocm.docs.amd.com/projects/HIP/en/latest/understand/compilers.html, 2024, (Piekļūts: 05.01.2025).

[11] ROCm, *HIPIFY pirmkoda krātuve*, https://github.com/ROCm/HIPIFY, 2025, (Piekļūts: 05.01.2025).

[12] TensorFlow komūna, *TensorFlow pirmkoda krātuve*, https://github.com/tensorflow/tensorflow, 2025, (Piekļūts: 05.01.2025).

[13] Nvidia, *CUDA on WSL User Guide*, https://docs.nvidia.com/cuda/wsl-user-guide/index.html, 2024, (Piekļūts: 06.01.2025).

[14] Advanced Micro Devices, *WSL How to guide - Use ROCm on Radeon GPUs*, `https://rocm.docs.amd.com/projects/radeon/en/latest/docs/install/wsl/howto_wsl.html`, 2024, (Piekļūts: 06.01.2025).

PIELIKUMI

Izdruka 7.1: Paroļu lauzēja implementācija CUDA vidē

```
1
2  // SHA 256 implementacija CUDA vide
3  // https://en.wikipedia.org/wiki/SHA-2
4
5  #include <cuda_runtime.h>
6  #include <device_launch_parameters.h>
7  #include <cuda/std/cstdint> // analogs C/C++ <cstdint>, bet nodrosina
       fiksetus datu tipu lielumus uz device
8  #include <string>
9  #include <vector>
10 #include <stdio.h>
11 #include <fstream>
12 #include <iostream>
13 #include <stdlib.h>
14 #include <string.h>
15 #include <sstream>
16 #include <algorithm>
17 #include <assert.h>
18 #include <iomanip>
19 #include <cstdint>
20
21 // forward deklaracija funkcijam, lai nav intelisense warningi, ka
       tas nav definetas (ir pieejamas uz device bez header include)
22 __device__ unsigned int __funnelshift_r(unsigned int lo,unsigned int
       hi,unsigned int shift);
23 unsigned int atomicCAS(unsigned int* address,unsigned int compare,
       unsigned int val);
24
25 // macro prieks katra cuda API izsaukuma rezultata parbaudes
26 // nemts no https://stackoverflow.com/questions/14038589/what-is-the-
       canonical-way-to-check-for-errors-using-the-cuda-runtime-api
27 #define CUDA_CHECK(ans) { gpuAssert((ans), __FILE__, __LINE__); }
28 inline void gpuAssert(cudaError_t code, const char *file, int line,
       bool abort=true)
29 {
30    if (code != cudaSuccess)
31    {
32       fprintf(stderr,"GPU assert: %s %s %d\n", cudaGetErrorString(
          code), file, line);
33       if (abort) exit(code);
34    }
35 }
36
37 // ROTR(x,n) rote x-a bitus pa labi pa n pozicijam, izmantojam cuda
       iebuveto funnelshift funkciju:
38 // https://docs.nvidia.com/cuda/cuda-math-api/cuda_math_api/
       group__CUDA__MATH__INTRINSIC__INT.html
```

```
39 // __device__ unsigned int __funnelshift_r(unsigned int lo, unsigned
       int hi, unsigned int shift)
40 // Concatenate hi : lo , shift right by shift & 31 bits, return the
       least significant 32 bits.
41 // Ja konkatene x ar pasu sevi, tad pec nobides, mazakie 32 biti
       satures attiecigo ROTR no x
42 #define ROTR(x, n) __funnelshift_r(x, x, n)
43
44 // makro funkcijas attieciga sha bloka apstradei
45 #define SS0(x) (ROTR(x, 7) ^ ROTR(x, 18) ^ (x >> 3))
46 #define SS1(x) (ROTR(x, 17) ^ ROTR(x, 19) ^ (x >> 10))
47 #define S0(x) (ROTR(x, 2) ^ ROTR(x, 13) ^ ROTR(x, 22))
48 #define S1(x) (ROTR(x, 6) ^ ROTR(x, 11) ^ ROTR(x, 25))
49 #define CH(x, y, z)  ((x & y) ^ (~x & z))
50 #define MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
51
52 // pirmie 32 biti kv. saknei no pirmajiem 8 pirmskaitliem 2 - 19 (no
       dalas aiz komata)
53 __device__ cuda::std::uint32_t h0 = 0x6a09e667;
54 __device__ cuda::std::uint32_t h1 = 0xbb67ae85;
55 __device__ cuda::std::uint32_t h2 = 0x3c6ef372;
56 __device__ cuda::std::uint32_t h3 = 0xa54ff53a;
57 __device__ cuda::std::uint32_t h4 = 0x510e527f;
58 __device__ cuda::std::uint32_t h5 = 0x9b05688c;
59 __device__ cuda::std::uint32_t h6 = 0x1f83d9ab;
60 __device__ cuda::std::uint32_t h7 = 0x5be0cd19;
61
62 // pirmie 32 biti no kubsaknem pirmajiem 64 pirmskaitliem 2 - 311
63 __device__ __constant__ cuda::std::uint32_t k[] =
64 {
65     0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5,0x3956c25b,0x59f111f1
          ,0x923f82a4,0xab1c5ed5,
66     0xd807aa98,0x12835b01,0x243185be,0x550c7dc3,0x72be5d74,0x80deb1fe
          ,0x9bdc06a7,0xc19bf174,
67     0xe49b69c1,0xefbe4786,0x0fc19dc6,0x240ca1cc,0x2de92c6f,0x4a7484aa
          ,0x5cb0a9dc,0x76f988da,
68     0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7,0xc6e00bf3,0xd5a79147
          ,0x06ca6351,0x14292967,
69     0x27b70a85,0x2e1b2138,0x4d2c6dfc,0x53380d13,0x650a7354,0x766a0abb
          ,0x81c2c92e,0x92722c85,
70     0xa2bfe8a1,0xa81a664b,0xc24b8b70,0xc76c51a3,0xd192e819,0xd6990624
          ,0xf40e3585,0x106aa070,
71     0x19a4c116,0x1e376c08,0x2748774c,0x34b0bcb5,0x391c0cb3,0x4ed8aa4a
          ,0x5b9cca4f,0x682e6ff3,
72     0x748f82ee,0x78a5636f,0x84c87814,0x8cc70208,0x90befffa,0xa4506ceb
          ,0xbef9a3f7,0xc67178f2
73 };
74
75 // apstrada vienu, konkretu 512 bitu bloku
76 // 'state' ir 8 skaitlu masivs, kuram apstrades sakuma jasatur h0-h7
       konstantes, apstrades beigas satures hash vertibu
77 // 'chunk' satur apstradajamo bitu bloku
```

```
78 __device__ void sha256ProcessChunk(cuda::std::uint32_t *state, cuda::
       std::uint8_t *chunk)
79 {
80     cuda::std::uint32_t w[64];
81 
82     // iekope visus 512 bitus ieks w masiva (512/32 = 16 vertibas)
83     // baiti jaieliek ieks 32 bitu vardiem, lai pirmais baits butu
           pirmais (skatoties no kreisas uz labo pusi),
84     // tas japabida pa kreisi pa 24, nakamie pa 16, 8, 0
85     // attieciga soli nakamie 'mazaksvarigie' biti ir nulles, tapec
           baitus sos baitus var konkatenet ar OR (|) operatoru
86     for(int i = 0; i < 16; i++)
87     {
88         w[i] = chunk[i * 4 + 0] << 24;
89         w[i] |= chunk[i * 4 + 1] << 16;
90         w[i] |= chunk[i * 4 + 2] << 8;
91         w[i] |= chunk[i * 4 + 3];
92     }
93 
94     // aizpilda parejas 'w' vertibas
95     for(int i = 16; i < 64; i++)
96     {
97         w[i] = w[i-16] + SS0(w[i-15]) + w[i-7] + SS1(w[i-2]);
98     }
99 
100    cuda::std::uint32_t a = state[0];
101    cuda::std::uint32_t b = state[1];
102    cuda::std::uint32_t c = state[2];
103    cuda::std::uint32_t d = state[3];
104    cuda::std::uint32_t e = state[4];
105    cuda::std::uint32_t f = state[5];
106    cuda::std::uint32_t g = state[6];
107    cuda::std::uint32_t h = state[7];
108 
109    for(int i = 0; i < 64; i++)
110    {
111        cuda::std::uint32_t temp1 = h + S1(e) + CH(e,f,g) + k[i] + w[
               i];
112        cuda::std::uint32_t temp2 = S0(a) + MAJ(a,b,c);
113        h = g;
114        g = f;
115        f = e;
116        e = d + temp1;
117        d = c;
118        c = b;
119        b = a;
120        a = temp1 + temp2;
121    }
122 
123    state[0] += a;
124    state[1] += b;
125    state[2] += c;
126    state[3] += d;
```

```
127     state[4] += e;
128     state[5] += f;
129     state[6] += g;
130     state[7] += h;
131 }
132 
133 __device__ void sha256(const cuda::std::uint8_t *input, cuda::std::
    uint64_t length, cuda::std::uint8_t *output)
134 {
135     // vienkarsibas pec apstradasim viena bloka ietvaros, tapec,
        nemot vera zinojuma garumu un padding,
136     // zinojuma garums nedrikst but lielaks par 440 bitiem, lai viss
        ietilpstu viena 512 bitu bloka
137     // https://crypto.stackexchange.com/questions/54852/what-happens-
        if-a-sha-256-input-is-too-long-longer-than-512-bits
138     bool lengthOk = length <= (440/8);
139     assert(lengthOk);
140 
141     cuda::std::uint32_t state[8] = {
142         h0,h1,h2,h3,h4,h5,h6,h7
143     };
144 
145     cuda::std::uint8_t chunk[64];
146 
147     // sakuma nonullejam bloku
148     for(int i = 0; i < 64; i++) {
149         chunk[i] = 0;
150     }
151 
152     // ierakstam pasu zinojumu
153     for(int i = 0; i < length; i++) {
154         chunk[i] = input[i];
155     }
156 
157     // pec prasibam ir japieliek '1' bits, parejas baita vertibas
        attiecigi ir nulles, atbilstosi SHA mainiga 'K' prasibam
158     chunk[length] = 0b10000000;
159 
160     // padding gala japieliek zinojuma garums ka 64 bitu big-endian
        skaitlis
161     for(int i = 1; i <= 8; i++)
162     {
163         chunk[64-i] = ((length * 8) >> ((i-1) * 8)) & 0xFF;
164     }
165 
166     sha256ProcessChunk(state, chunk);
167 
168     // sadalam 32 bitu vertibas 4as 8 bitu un ierakstam output masiva
169     for(int i = 0; i < 8; i++) {
170         cuda::std::uint32_t currentStateValue = state[i];
171 
172         output[i * 4] = (cuda::std::uint8_t)(currentStateValue >> 24)
            ;
```

```
173         output[i * 4 + 1] = (cuda::std::uint8_t)(currentStateValue >>
                16);
174         output[i * 4 + 2] = (cuda::std::uint8_t)(currentStateValue >>
                8);
175         output[i * 4 + 3] = (cuda::std::uint8_t)(currentStateValue);
176     }
177 }
178
179 __device__ bool compareHashes(const cuda::std::uint8_t *h1, const
       cuda::std::uint8_t *h2)
180 {
181     for(int i = 0; i < 32; i++)
182     {
183         if(h1[i] != h2[i])
184         {
185             return false;
186         }
187     }
188
189     return true;
190 }
191
192 __global__ void kernel(
193     const cuda::std::uint8_t *passwords,
194     const int *pwLengths,
195     int pwCount,
196     cuda::std::uint64_t maxPwLength,
197     const cuda::std::uint8_t *targetHash,
198     int *resultIndex
199 )
200 {
201     int idx = blockIdx.x * blockDim.x +threadIdx.x;
202
203     if(idx >= pwCount)
204     {
205         return;
206     }
207
208     const cuda::std::uint8_t *password = passwords + (idx *
           maxPwLength);
209     int pwLength = pwLengths[idx];
210
211     cuda::std::uint8_t hash[32];
212
213
214     sha256(password, pwLength, hash);
215
216     // ja hashi sakrit, tad ierakstam paroles indeksu ieks '
           resultIndex',
217     // ta ka ta mainiga adrese atrodas device kopigaja atmina,
           jalieto atomiska funkcija
218     if(compareHashes(targetHash, hash))
219     {
```

```
220         // ja resultIndex glabajas vertiba -1, tad aizstaj to ar idx
221         // ar so tiek ari reize nodrosinats, ka ja kads pavediens
               velak tomer nonak lidz sim stavoklim,
222         // tad modifikacija netiks veikta, jo taja bridi jau '
               resultIndex' != -1
223         atomicCAS((unsigned int*)resultIndex, -1, (unsigned int)idx);
224     }
225 }
226
227 static uint8_t parseHexByte(const std::string &hash, size_t offset)
228 {
229     std::string byteString = hash.substr(offset, 2);
230     return static_cast<uint8_t>(std::stoi(byteString, nullptr, 16));
231 }
232
233 std::vector<uint8_t> hexStringToBytes(const std::string &hash)
234 {
235     assert(hash.size() == 64); // 256 biti => 64 hex skaitli
236
237     std::vector<uint8_t> result(32);
238
239     for(size_t i = 0; i < 32; i++)
240     {
241         result[i] =  parseHexByte(hash, i * 2);
242     }
243
244     return result;
245 }
246
247
248 std::string parseBytesToHexString(const uint8_t* data, size_t length)
249 {
250     std::ostringstream ss;
251
252     for(size_t i = 0; i < length; i++)
253     {
254         ss << std::hex << std::setw(2) << std::setfill('0') << (int)
               data[i];
255     }
256
257     return ss.str();
258 }
259
260 void hashCheck(std::vector<std::string>& passwords, std::vector<
    uint8_t>& hash, int *cracked_idx)
261 {
262     assert(*cracked_idx == -1); // te sakuma jau jabut vertibai -1,
        padota no main
263
264     const int passwordCount = passwords.size();
265     const int maxPwLength = 55; // maksimalais zinojuma garums, lai
        tas ietilptu viena sha bloka
```

```
266     const size_t passwordsSize = passwordCount * maxPwLength; // ar
           pienemumu, ka viens simbols ir 1 baits
267
268     std::vector<uint8_t> pwBuffer(passwordsSize, 0);
269     std::vector<int> pwLengths(passwordCount);
270
271     for(size_t i = 0; i < passwordCount; i++) {
272         const std::string pw = passwords[i];
273         int pwLength = pw.size();
274
275         assert(pwLength <= maxPwLength);
276
277         pwLengths[i] = pwLength;
278
279         for(size_t j = 0; j < pwLength; j++)
280         {
281             // ja parole ir isaka par maxPwLength, tad 'tuksie'
                   masiva elementi bus aizpilditi ar nullem
282             pwBuffer[i * maxPwLength + j] = (uint8_t)pw[j];
283
284         }
285     }
286
287     cuda::std::uint8_t *d_passwords;
288     cuda::std::uint8_t *d_hash;
289     int *d_pwLengths;
290     int *d_cracked_idx;
291
292     *cracked_idx = -1;
293
294     CUDA_CHECK(cudaSetDevice(0));
295
296     CUDA_CHECK(cudaMalloc(&d_passwords, pwBuffer.size() * sizeof(
           uint8_t)));
297     CUDA_CHECK(cudaMemcpy(d_passwords, pwBuffer.data(), pwBuffer.size
           () * sizeof(uint8_t), cudaMemcpyHostToDevice));
298
299     CUDA_CHECK(cudaMalloc(&d_hash, 32));
300     CUDA_CHECK(cudaMemcpy(d_hash,hash.data(), 32,
           cudaMemcpyHostToDevice));
301
302     CUDA_CHECK(cudaMalloc(&d_pwLengths, passwordCount * sizeof(int)))
           ;
303     CUDA_CHECK(cudaMemcpy(d_pwLengths, pwLengths.data(),passwordCount
            * sizeof(int), cudaMemcpyHostToDevice));
304
305     CUDA_CHECK(cudaMalloc(&d_cracked_idx, sizeof(int)));
306     CUDA_CHECK(cudaMemcpy(d_cracked_idx, cracked_idx, sizeof(int),
           cudaMemcpyHostToDevice));
307
308
309     int numThreads = 256;
310     int numBlocks = (passwordCount + numThreads - 1 ) / numThreads;
```

```
311
312     kernel<<<numBlocks, numThreads>>>(d_passwords, d_pwLengths,
            passwordCount, maxPwLength, d_hash, d_cracked_idx);
313
314     CUDA_CHECK(cudaGetLastError());
315     CUDA_CHECK(cudaDeviceSynchronize());
316
317     CUDA_CHECK(cudaMemcpy(cracked_idx, d_cracked_idx, sizeof(int),
            cudaMemcpyDeviceToHost));
318
319     cudaFree(d_passwords);
320     cudaFree(d_hash);
321     cudaFree(d_cracked_idx);
322 }
323
324
325 void processFile(const std::string& fileName, std::vector<std::string
        >& buffer)
326 {
327     std::ifstream file(fileName);
328
329     if(!file.is_open())
330     {
331         throw std::runtime_error("Error opening file\n");
332     }
333
334     std::string line;
335     unsigned int currentOffset = 0;
336
337     while(std::getline(file,line))
338     {
339         buffer.push_back(line);
340     }
341
342     file.close();
343 }
344
345 // sha funkcijas testa device kodols
346 __global__ void testKernel(
347     const cuda::std::uint8_t *input,
348     cuda::std::uint64_t length,
349     cuda::std::uint8_t *calculatedHash
350 )
351 {
352     // testam pietieks ar pirmo pavedienu
353     if(threadIdx.x != 0 || blockIdx.x != 0) {
354         return;
355     }
356
357     sha256(input,length,calculatedHash);
358 }
359
```

```
360 void testSha(const std::string &password,const std::string
        hexExpectedHash)
361 {
362     std::vector<uint8_t> expectedHash = hexStringToBytes(
            hexExpectedHash);
363
364     size_t pwLength = password.size();
365     std::vector<uint8_t> passwordBytes(pwLength);
366
367     for(size_t i = 0; i < pwLength; i++)
368     {
369         passwordBytes[i] = password[i];
370     }
371
372     cuda::std::uint8_t *d_password;
373     cuda::std::uint8_t *d_hash;
374     cuda::std::uint8_t *d_calculatedHash;
375
376     CUDA_CHECK(cudaSetDevice(0));
377
378     CUDA_CHECK(cudaMalloc(&d_password,pwLength));
379     CUDA_CHECK(cudaMemcpy(d_password, passwordBytes.data(),pwLength,
            cudaMemcpyHostToDevice));
380
381     CUDA_CHECK(cudaMalloc(&d_calculatedHash,32));
382     CUDA_CHECK(cudaMemcpy(d_calculatedHash, std::vector<uint8_t
            >(32,0).data(), 32, cudaMemcpyHostToDevice));
383
384
385     int numThreads = 1;
386     int numBlocks = 1;
387
388     testKernel<<<numBlocks, numThreads>>>(d_password,pwLength,
            d_calculatedHash);
389
390     CUDA_CHECK(cudaGetLastError());
391     CUDA_CHECK(cudaDeviceSynchronize());
392
393     cuda::std::uint8_t h_calculatedHash[32];
394
395     CUDA_CHECK(cudaMemcpy(&h_calculatedHash, d_calculatedHash, 32,
            cudaMemcpyDeviceToHost));
396
397     cudaFree(d_password);
398     cudaFree(d_calculatedHash);
399
400     std::cout << "Expected:\t" << hexExpectedHash << "\nActual:\t\t"
            << parseBytesToHexString(h_calculatedHash,32) << '\n';
401 }
402
403
404 int main()
405 {
```

```
406     std::cout << "SHA Tests\n";
407
408     testSha("","
            e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
            ");
409     testSha("123456","8
            d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92
            ");
410
411
412     std::cout << "Hash Converison Tests\n";
413
414     std::string testHexHash = "8
            d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92
            ";
415     auto hexbytes = hexStringToBytes(testHexHash);
416     std::cout << "Original hash:\t\t" << testHexHash << "\nRoundtrip
            converted: \t" << parseBytesToHexString(hexbytes.data(),
            hexbytes.size()) << '\n';
417
418     std::cout << "Tests complete\n";
419
420     const std::string INPUT_FILE_NAME = "C:\\Users\\hazya\\Desktop
            \\10-million-password-list-top-1000000.txt";
421
422     std::vector<std::string> buffer;
423
424     processFile(INPUT_FILE_NAME, buffer);
425
426     std::string hexHash = "701402
            a369ed3107a22195f5d570ed29df71f39e2ce01123ea0c564bc8333270";
427
428     std::vector<uint8_t> hash = hexStringToBytes(hexHash);
429
430     int cracked_idx = -1;
431
432     hashCheck(buffer, hash, &cracked_idx);
433
434     if (cracked_idx != -1)
435     {
436         std::cout << "Yipeee! Found the password at index:" <<
                cracked_idx << std::endl;
437         std::cout << buffer[cracked_idx] << std::endl;
438     }
439     else
440     {
441         std::cout << "No matching passwords" << std::endl;
442     }
443
444     CUDA_CHECK(cudaDeviceReset());
445
446     return 0;
447 }
```

Ja darbam nepieciešams, dažādus palīgmateriālus var ievietot pielikumā. Tajā parasti iekļauj aprēķinu starprezultātus, ilustrācijas, anketu paraugus, kartes, aparātu un ierīču aprakstus u. c.

Kursa darbs "GPU programmēšanas salīdzinājums CUDA, ROCm un OpenCL saskarnēs" izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti.

Autors: Artūrs Kļaviņš _____

Rekomendēju/nerekomendēju darbu aizstāvēšanai (nevajadzīgo izsvītrot)

Darba vadītājs: profesors, Dr. dat. Leo Seļāvo _____

Darbs iesniegts Datorikas fakultātē

Dekāna pilnvarotā persona:

Darbs aizstāvēts kursa darbu komisijas sēdē

Komisija: