

LATVIJAS UNIVERSITĀTE  
EKSAKTO ZINĀTŅU UN TEHNOLOĢIJU FAKULTĀTE  
DATORIKAS NODAĻA

**GPU PROGRAMMĒŠANA NVIDIA CUDA UN  
AMD ROCM SASKARNĒS**

KURSA DARBS

Autors: **Artūrs Kļaviņš**

Studenta apliecības Nr.: ak21373

Darba vadītājs: profesors, Dr. dat. Leo Seļāvo

RĪGA, 2024

## **Anotācija**

Anotācijas saturs

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

## **Abstract**

Abstract body

**Keywords:** GPU, Nvidia, AMD, CUDA, ROCm

# SATURS

<b>1</b>	<b>Ievads</b>	<b>4</b>
<b>2</b>	<b>GPGPU arhitektūra un programmēšana</b>	<b>5</b>
<b>3</b>	<b>Platformu salīdzinājums</b>	<b>7</b>
<b>4</b>	<b>Uzdevums</b>	<b>10</b>
4.1	Izstrādājamās programmas definīcija . . . . .	11
4.1.1	CUDA risinājums . . . . .	11
4.1.2	Portēšana uz HIP caur WSL 2 . . . . .	13
4.1.3	Analīze un secinājumi . . . . .	13
<b>5</b>	<b>Platformneatkarīgi risinājumi</b>	<b>14</b>
	<b>Bibliogrāfija</b>	<b>15</b>
	<b>Pielikumi</b>	<b>16</b>

## 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 ģeometru 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 vispārīgus lineārās algebras vienādojumus, pārveidojot matricu datus kā tekstūras un uz tām izpildot ģeometru funkcijas. [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 pieejamas salīdzināmās funkcijas, ātrdarbību un iespējamās priekšrocības, izvēloties vienu vai otru.

## 2.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 pavadieniem 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

### 3.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, modificēt, kompilēt pats, kā arī dot savu pienesumu gan dokumentācijā, gan kodā.[4]

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*).[5]

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 3.1).

**Izdruka 3.1:** CUDA un HIP funkciju definīciju salīdzinājums

```
1 // CUDA:
2 __host__ myCpuFunction() { /*...*/ }
3 __device__ myGpuFunction() { /*...*/ }
4 __global__ kernel() {}
5
6 // HIP:
7 __host__ myCpuFunction() { /*...*/ }
8 __device__ myGpuFunction() { /*...*/ }
9 __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 funkciju API, tās saturošās funkcijas ir tieši atbilstošas CUDA funkcijām.[6, 7]

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'. [8].

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'. [9]

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. [9]

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 instalācijas 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 overrideot 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 programmatūrā 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 pāreju 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. [10]

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

## 4.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.[11, 12]

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 daudzās mazās daļās (vēlams skaitā  $\geq 1$  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 parolu laužējs.

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

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

## 4.1 Izstrādājamās programmas definīcija

Programma ir domāta kā komandrindas utilīta ar CLI argumentiem:

- testu palaišanas arguments
- ievades faila ceļš,
- paroles jaucejvērtība.

Apstrādājamais fails saturēs potenciālās paroles, katra savā rindā, kurām tiks izrēķināta jaucejvērtība un salīdzināta pret doto. Izmantojamais jaukšanas algoritms - SHA256. Jaucejvērtību 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 jaucejvērtībām un to attiecīgajiem ziņojumiem, kā arī, lai pārbaudītu GPU kodola programmas darbību, platformas pieejamību uz šī datora apertūras
- `$ pwCracker <paroļu faila ceļš> <paroles hash vērtība>`, lai veiktu galveno programmas izpildi

### 4.1.1 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 vispārīgi reti lietota, vai tā nav īsti paredzēta lietošanai iekš GPU.

Piemēram, nav pieejamas `std::string`, `std::vector` struktūras, jo tās dinamiski iedala atmiņā, 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 parolu 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 parolu buferi un katrai parolei fiksēt garumu, tām liekot galā nulļu 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  $13 \cdot 2048 = 26\text{K}$  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.

```
1 __global__ void kernel(  
2     const cuda::std::uint8_t *passwords,  
3     const int *pwLengths,  
4     int pwCount,  
5     cuda::std::uint64_t maxPwLength,  
6     const cuda::std::uint8_t *targetHash,  
7     int *resultIndex  
8 )  
9 {  
10     int idx = blockIdx.x * blockDim.x + threadIdx.x;  
11  
12     if(idx >= pwCount)  
13     {  
14         return;
```

```

15     }
16
17     const cuda::std::uint8_t *password = passwords + (idx *
18         maxPwLength);
19     int pwLength = pwLengths[idx];
20
21     cuda::std::uint8_t hash[32];
22
23     sha256(password, pwLength, hash);
24
25     // ja hashi īsakrt, tad āierakstm paroles indeksu šiek '
26     // resultIndex',
27     // āt āk āt īāmaing adrese atrodas device īākopgaj ņāatmi, ājlieto
28     // atomiska funkcija
29     if(compareHashes(targetHash, hash))
30     {
31         // ja resultIndex āglabjas ēīvrtba -1, tad āaizstj to ar idx
32         // ar šo tiek īar ēreiz šānodroints, ka ja ākds pavediens
33         // ēāvlk ētomr ānonk īldz šim āstvoklim,
34         // tad āmodifikcija netiks veikta, jo ātaj īībrd jau '
35         // resultIndex' != -1
36         atomicCAS((unsigned int*)resultIndex, -1, (unsigned int)idx);
37     }
38 }

```

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

#### 4.1.2 Portēšana uz HIP caur WSL 2

#### 4.1.3 Analīze un secinājumi

Ir jāpievērš uzmanība mērķa videokartei un mikroarhitektūrai, jo izstrādājot programmu uz personīgā datora ar RTX 3060 videokarti bija pieejamas funkcijas, kuras testējot uz cita datora ar Quattro sērijas 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

## **5.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] Advanced Micro Devices, *What is ROCm?*, <https://rocm.docs.amd.com/en/latest/what-is-rocm.html>, 2024, (Piekļūts: 05.01.2025).
- [5] *HIP documentation*, (Piekļūt: 04.01.2025).
- [6] Advanced Micro Devices, *HIP math API*, [https://rocm.docs.amd.com/projects/HIP/en/latest/reference/math\\_api.html](https://rocm.docs.amd.com/projects/HIP/en/latest/reference/math_api.html), 2024, (Piekļūts: 05.01.2025).
- [7] Nvidia, *CUDA Math API Reference Manual*, <https://docs.nvidia.com/cuda/cuda-math-api/index.html>, 2024, (Piekļūts: 05.01.2025).
- [8] Advanced Micro Devices, *HIP Compilers*, <https://rocm.docs.amd.com/projects/HIP/en/latest/understand/compilers.html>, 2024, (Piekļūts: 05.01.2025).
- [9] ROCm, *HIPIFY pirmkoda krātuve*, <https://github.com/ROCm/HIPIFY>, 2025, (Piekļūts: 05.01.2025).
- [10] TensorFlow komūna, *TensorFlow pirmkoda krātuve*, <https://github.com/tensorflow/tensorflow>, 2025, (Piekļūts: 05.01.2025).
- [11] Nvidia, *CUDA on WSL User Guide*, <https://docs.nvidia.com/cuda/wsl-user-guide/index.html>, 2024, (Piekļūts: 06.01.2025).
- [12] 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](https://rocm.docs.amd.com/projects/radeon/en/latest/docs/install/wsl/howto_wsl.html), 2024, (Piekļūts: 06.01.2025).

# PIELIKUMI

Izdruka 5.1: Paroļu laužēja 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, konkrētu 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,
nemt vera zinojuma garumu un padding,
136     // zinojuma garums nedrīkst 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
bool lengthOk = length <= (440/8);
138     assert(lengthOk);
139
140     cuda::std::uint32_t state[8] = {
141         h0,h1,h2,h3,h4,h5,h6,h7
142     };
143
144     cuda::std::uint8_t chunk[64];
145
146     // sakuma nonullejam bloku
147     for(int i = 0; i < 64; i++) {
148         chunk[i] = 0;
149     }
150
151     // ierakstam pasu zinojumu
152     for(int i = 0; i < length; i++) {
153         chunk[i] = input[i];
154     }
155
156     // pec prasibam ir japieliek '1' bits, parejas baita vertibas
attiecigi ir nulles, atbilstosi SHA mainiga 'K' prasibam
157     chunk[length] = 0b10000000;
158
159     // padding gala japieliek zinojuma garums ka 64 bitu big-endian
skaitlis
160     for(int i = 1; i <= 8; i++)
161     {
162         chunk[64-i] = ((length * 8) >> ((i-1) * 8)) & 0xFF;
163     }
164
165     sha256ProcessChunk(state, chunk);
166
167     // sadalam 32 bitu vertibas 4as 8 bitu un ierakstam output masiva
168     for(int i = 0; i < 8; i++) {
169         cuda::std::uint32_t currentStateValue = state[i];
170
171         output[i * 4] = (cuda::std::uint8_t)(currentStateValue >> 24)
172         ;

```

```

173         output[i * 4 + 1] = (cuda::std::uint8_t)(currentStateValue >>
174             16);
175         output[i * 4 + 2] = (cuda::std::uint8_t)(currentStateValue >>
176             8);
177         output[i * 4 + 3] = (cuda::std::uint8_t)(currentStateValue);
178     }
179 }
180
181 __device__ bool compareHashes(const cuda::std::uint8_t *h1, const
182     cuda::std::uint8_t *h2)
183 {
184     for(int i = 0; i < 32; i++)
185     {
186         if(h1[i] != h2[i])
187         {
188             return false;
189         }
190     }
191     return true;
192 }
193
194 __global__ void kernel(
195     const cuda::std::uint8_t *passwords,
196     const int *pwLengths,
197     int pwCount,
198     cuda::std::uint64_t maxPwLength,
199     const cuda::std::uint8_t *targetHash,
200     int *resultIndex
201 )
202 {
203     int idx = blockIdx.x * blockDim.x + threadIdx.x;
204
205     if(idx >= pwCount)
206     {
207         return;
208     }
209
210     const cuda::std::uint8_t *password = passwords + (idx *
211         maxPwLength);
212     int pwLength = pwLengths[idx];
213
214     cuda::std::uint8_t hash[32];
215
216     sha256(password, pwLength, hash);
217
218     // ja hashi sakrit, tad ierakstam paroles indeksu ieks '
219     // resultIndex',
220     // ta ka ta mainiga adrese atrodas device kopigaja atmina,
221     // jalieto atomiska funkcija
222     if(compareHashes(targetHash, hash))
223     {

```

```

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

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādī-  
tie 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: