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 atbalsta, 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]

Pie līdzīga secinājuma nonāca ASV kompānija Apple 2009. gadā, izstrādājot pirmo OpenCL versiju. Vēlāk to sniedzot Khronos darba grupā kopā ar citām IT kompānijām, tai skaitā AMD un Nvidia. [3] AMD pie sava risinājuma modernā risinājuma nonāca 2016. gada, apvienojot iepriekšējos kompānijas GPU projektus, tika izveidota ROCm platforma kā tieša konkurente CUDA platformai.

Ņemot vērā, ka šie trīs projekti ir paredzēti līdzīgiem mērķiem, to pieejamās funkcionalitātes līdzinās viena otrai, CUDA un OpenCL ir iesaistītas vairākos lielos atvērta pirmkoda projektos, šī darba mērķis ir salīdzināt visas trīs platformas, to pieejamās salīdzināmās funkcijas, ātrdarbību un iespējamās priekšrocības, izvēloties kādu no tām.

2.PLATFORMU GPGPU PROGRAMMĒŠANAS MODEĻI UN LIETOJUMPROGRAMMU SASKARNE

GPU ir dizainēti ar domu izpildīt tūkstošiem pavedienus (vienādus sarakstus ar procesorā izpildāmām darbībām) paralēli. Atšķirībā no CPU, kur viens pavediens šīs darbības izpildās ātrāk, GPU kopumā dos lielāku caurlaidību.

Arhitektūras līmenī GPGPU kodoli satur SIMD (*Single Instruction, Multiple Data*) elementus, kuri spējīgi veikt paralēlu datu apstrādi ar vienu instrukciju, katrs elements strādājot ar saviem teorētiski patvaļīgiem datiem (atmiņas adresēm).

SIMD nav pieejami tiešā veidā caur instrukciju kopu, programmētājam ir jādarbojas ar pavedienu saskarnes SIMT (*Single Instruction, Multiple Threads*) darbību izpildes modeli. [4] Arhitektūrā tie ir ieviesti kā GPGPU kodoli un satur SIMT elementus. Līdz ar to SIMT kodols tiek saukt par priekšgalu (*front-end*) un SIMD par aizmugursistēmu (*back-end*).

SIMT abstrakcija ļauj programmētājam neuztraukties par pavedienu implementācijas detaļām, un tos var uzskatīt par pilnībā neatkarīgiem - katrs pavediens izpildās paralēli un ir spējīgs izpildīt patvaļīgu instrukciju sarakstu.

Rezultātā programmētājs uz GPU izpildāmo kodu, kuru nodod noteiktam skaitam pavedienu, saucamu par GPGPU izpildāmo kodolu (*device kernel*), ir spējīgs definēt kā MIMD (*Multiple Instructions, Multiple Data*) modelim līdzīgu kodu, tāpēc funkciju definēšana ir praktiski identiska tam kā to darītu priekš CPU, ar noteiktiem specifiskiem saskarnes API izsaukumiem un kontekstu, ka šis pavediens izpildās paralēli. [5]

Kursa darbā jau tika apskatītas galvenās atšķirības un kopējais starp CUDA un ROCm.[5] 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 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] CUDA gadījumā viss pieejamais programmatūras steks tiek palikts zem jau minētā, plašā termina - CUDA.

Pats programmēšanas modelis ir diezgan līdzīgs un vietām pat identisks ar CUDA. 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, ir vienādi (skatīt izdruku 2.1).

Izdruka 2.1: CUDA un HIP funkciju definīciju salīdzinājums

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

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

Tā pat ir arī ar GPGPU kodola izsaukumiem, minimālas atšķirības atmiņas iedalīšanā un aizpildīšanā (skatīt izdruku 2.2).

Izdruka 2.2: CUDA un HIP kodola darbināšanas, atmiņas API izsaukumu salīdzinājums

```
int *host_dati; // pienemam, ka ir aizpilditi ar datiem
int *host_rezultats;
int *device_dati;
int *device_rezults;
size_t size = 1234;

// CUDA:
cudaMalloc(&device_dati, size);
cudaMalloc(&device_rezultats, size);
cudaMemcpy(device_dati, host_dati, cudaMemcpyHostToDevice);
kernel<<<(size + 256 - 1)/256, 256>>>(device_dati, device_rezulats);
cudaMemcpy(host_rezultats, d_rezultats, cudaMemcpyDeviceToHost);

// HIP:
hipMalloc(&device_dati, size);
hipMalloc(&device_rezultats, size);
hipMemcpy(device_dati, host_dati, hipMemcpyHostToDevice);
kernel<<<(size + 256 - 1)/256, 256>>>(device_dati, device_rezulats);
hipMemcpy(host_rezultats, d_rezultats, hipMemcpyDeviceToHost);
```

Ņemot vērā šīs minimālās atšķirības, var secināt, ka vienkāršām situācijām portēt no vienas saskarnes uz otru nebūtu pārāk sarežģīti. Darba atvieglošanai, ROCm ietver

portēšanas rīku "HIPIFY" [8] no CUDA uz HIP, kas spējīgs arī sarežģītākām situācijām analizēt doto CUDA pirmkodu un aizstāt attiecīgos CUDA header failus, funkciju izsaukumus, dekoratorus ar HIP atbilstošajiem.

HIP ir 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 un bieži vien pat ar vienādu funkcijas nosaukumu. [9, 10]

Kompilēšanas līmenī šāds 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'. [11].

Tā kā varētu interpretēt, ka CUDA ir tieša apakškopa ROCM un HIP platformai, bet tomēr pilnīgs atbalsts visām CUDA funkcijām nav pieejams. Ir funkcijas, kuras HIP nav implementējusi, piemēram, 'cyl_bessel_i0f' [9]. Papildus nianses ir specifiskām funkcijām kā 'nextafterf', kura CUDA ir pieejama gan CPU, gan GPU kodā, bet HIP gadījumā tikai uz CPU. [9]

GPGPU izpildāmais kodols tiek 'nodots' no CPU caur saskarņu API izsaukumiem, parametros definējot:

- pavedienu skaitu blokā,
- bloku skaitu,
- kodola funkcijas argumentus (visbiežāk tās būs apstrādājamo datu atmiņas adreses - rādītāji). [4]

Khronos Group definētā un izstrādātā paralēlās programmēšanas platforma un ietvars - OpenCL ir paredzēta plašākam iekārtu klāstam, fokuss nav tieši uz videokartēm. OpenCL GPGPU kodolus ir iespējams darbināt uz CPU, GPU, aparatūras paātrinātājiem kā FPGA (no angļu val. *Field Programmable Gate Array*), ciparsignālu un mākslīgā intelekta AI procesoriem. [12]

Atšķirībā no CUDA un HIP, kur kodolu definēšana notiek C++ kodā līdzās ar CPU puses kodu, OpenCL ir definēta sava C lietojumprogrammu saskarnes valoda, kura balstīta uz C99.[13] Tā domāta tieši GPGPU kodolu definēšanai, un to iespējams kā teksta simbolu virkni ielādēt un kompilēt CPU puses kodā. Noteiktās situācijas tas var būt mīnuss - programmai jāvelta laiks kompilējot kodolus, ieviešot ātrdarbības zudumus.

Izdruka 2.3: OpenCL GPGPU kodols un izsaukšana no C++

```
// Kodols
__kernel void saxpy(__private float alpha,
                    __global const float *x,
                    __global float *y,
                    __private const unsigned int n)
{
    const int i = get_global_id(0);
    if (i >= n)
    {
        return;
    }
    y[i] = alpha * x[i] + y[i];
}

// CPU kods

clGetPlatformIDs(1, &platform, &numPlatforms);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, &numDevices)
    ;
context = clCreateContext(nullptr, 1, &device, nullptr, nullptr, &
    clResult);
queue = clCreateCommandQueueWithProperties(context, device,
    properties, &clResult);

std::string kernelSource = "kodola kods";
const char *kernelSourceCstring = kernelSource.c_str();
size_t kernelSize = kernelSource.length();

cl_program program = clCreateProgramWithSource(context, 1, &
    kernelSourceCstring, &kernelSize, &clResult);
clBuildProgram(program, 1, &device, "-cl-std=CL3.0", NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "saxpy", &clResult);

unsigned int vector_size = VECTOR_SIZE;
size_t global_size = VECTOR_SIZE;
size_t local_size = 64;

clSetKernelArg(kernel, 0, sizeof(float), &alpha);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_x);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_y);
clSetKernelArg(kernel, 3, sizeof(unsigned int), &vector_size);

clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size, &
    local_size, 0, NULL, NULL);
clEnqueueReadBuffer(queue, d_y, CL_TRUE, 0, VECTOR_SIZE * sizeof(
    float), h_y, 0, NULL, NULL);
```

Salīdzinot ar CUDA un HIP, OpenCL ir 'vārdīgāks' kodolu darbināšanā, jo no vienas puses kodola argumentu padošana CUDA un HIP ir vienkāršota caur «<...»> makro funkciju, no otras puses OpenCL piedāvā plašākas iespējas iekārtas vai iekārtu, kodolu,

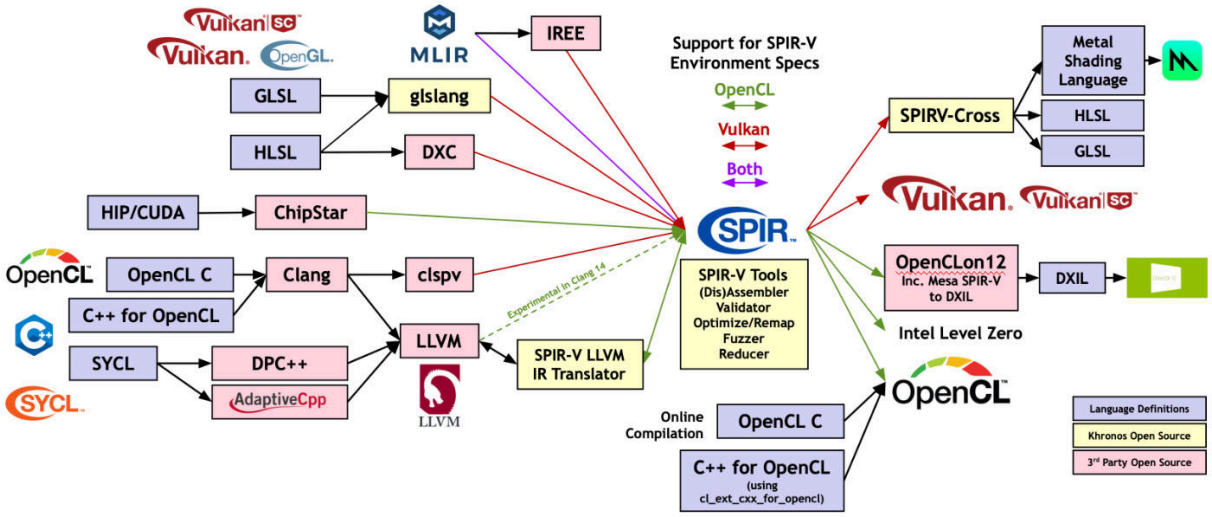
atmiņas organizēšanā: [12]

- Var vaicāt pēc OpenCL platformu, kura satur CPU un paralēlās izpildes iekārtas,
- Viena vai vairākas iekārtas, kuras var būt dažādu tipu,
- Konteksts, kurš satur:
  - Vienu vai vairākas iekārtas,
  - GPGPU kodola objektus jeb funkcijas, kuras izsauktas galvenajā kodola funkcijā
  - Programmas objektus - GPGPU kodolu pirmkodu vai kompilētus kodola bināros failus
  - Atmiņas objektus, kurus pārvalda saimnieka CPU un padod kodolu funkcijām (iekārtas atmiņas norādes, kuras uzturētas saimnieka CPU RAM)
  - Rindu - OpenCL kontekstam ir iespējams asinhroni nodot izpildei daudzus kodolus ar dažādām prioritātēm, izpildes kārtība un tās stāvoklis tiek pārvaldīts konteksta rindā

Rezultātā OpenCL programmēšanas modelis piedāvā jau gatavas struktūras un funkcionalitāti darbam ar vairākām iekārtām, vairākiem kontekstiem, vairākām kontekstu rindām un kodoliem, kuras noderīgas sarežģītās konveijerapstrādes tipa programmatūrās.

Kodoli nav kompilējami tikai izpildes laikā, bet arī tā sauktajā bezsaistes režīmā uz SPIR-V starp-posma reprezentācijas valodu; tā tiek lietota arī citos Khronos Group projektos kā Vulkan, SYCL[14] (skatīt attēlu 2.1). Bet ar šādu risinājumu SPIR-V fails ir kompilēts uz konkrētā datora un vairs netiek nodrošināta platform-neaktarīga. OpenCL iekārtas puses kompilators līdz ar to ir pieejams divos režīmos:

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



Att. 2.1: Khronos Group izstrādāto projektu ekosistēma[15]

Tiešsaistes kodolu kompilēšana nodrošina lielāku savietojamību ar dažādām aparatūrām, jo OpenCL kompilēs kodu tiešai attiecīgajai aparatūrai, jo OpenCL nodrošina izpildes laika kompilēšanu visos gadījumos un API versijās.

3.ETALONUZDEVUMU IZVEIDES APSVĒRUMI

Etalonuzdevumu izveidē (programmatūras vai algoritmu salīdzināšanas vajadzībām) jābūt pārliecinātam, ka izveidotā programma, izmantotie profilēšanas rīki un jebkāda cita programmatūra, kas kopumā paredzēta mērījumu iegūšanai, noformēšanai un analīzei, ir uzticama un atkārtojuma.

Apskatot šo problēmu vispārīgāk kā kāda eksperimenta veikšanu, darbības rezultātam, secinājumiem jābūt neatkarīgi atkārtojamiem. Lai labāk atdalītu un līdz ar to izprastu dažādas atkārtojamības pakāpes, ASV bāzētā, starptautiskā Skaitļošanas tehnikas asociācija (ACM) definē šādu terminoloģiju eksperimentiem (tulkojot no angļu val.): [16]

- Atkārtojamība (repeatability) - tā pati komanda[1], tāds pats uzstādījums,
- Reproducējamība (reproducability) - cita komanda, tāds pats uzstādījums,
- Atdarināmība (replicability) - cita komanda, cits uzstādījums.

Praktisko ierobežojumu dēļ garantēt pat reproducējamību šī darba ietvaros nav iespējams, bet uz to var tiekties precīzi definējot noteikumus, kuri tiks ievēroti programmu un etalonuzdevumu izstrādē, iegūstot atkārtojamību, tā lai to pašu varētu veikt arī citi.

Lai izveidotu etalonuzdevumu CUDA, ROCm HIP un OpenCL platformām, būs nepieciešams definēt kādu problēmu, kuras risinājumu ir jēga realizēt izpildei uz videokartes.

Jāizvēlas tāds uzdevums, kuru iespējams 'augsti' paralelizēt, tas ir, sadalīt uzdevumu daudzos mazos un (it īpaši) neatkarīgos gabalos, lai būtu pievienotā vērtība (ātrdarbība) to pildīt uz GPU, ņemot vērā papildus darbu un nepieciešamās zināšanas, lai ieviestu GPU risinājumu. Autora kursa darbā[5] tāds jau tika definēts, bet netika pievērsta papildus uzmanība etalonuzdevuma uzticamībai.

Salīdzinot dažādus ietvarus ar it kā vienu un to pašu uzdevumu, jāņem vērā, ka ar naivu risinājumu ir grūti garantēt, ka attiecīgās izveidotās programmas ir pietiekami līdzvērtīgas, lai iegūtos datus un līdz ar to ietvarus 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)

[1] Šajā kontekstā ar komandu saprotami cilvēki, eksperimenta veicēji.

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, kas satur 100 miljons rindu, izmantojot time[17] 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 rindas

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

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

real    0m7.129s
```

```
user    0m6.303s
sys     0m0.819s
```

Tātad no iegūtajiem rezultātiem (25,787s ilgs izpildes laiks 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: augsta līmeņa valoda kā Python abstraktē relatīvi sarežģītas darbības ar failiem, datu buferiem, straumēm, standarta ievadi un 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

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

int main()
{
    std::ios::sync_with_stdio(false); // nesinhronize C un C++ stdio
    std::cin.tie(nullptr); // neiztira 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
Fails satur 100000000 rindas

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

Līdz ar to, izstrādājot etalonuzdevumu programmas, jāņem vērā, ka var iegūt nepatiesu līdzvērtību starp risinājumiem dažādās valodās un neuzticamus datus, rezultātus.

CUDA, ROCm HIP un OpenCL platformās šī problēma ir mazāk izteikta, jo ietvari ir diezgan līdzīgā "abstrakcijas" pakāpē, bet tāpat tiek izvirzīti šādi ierobežojumi:

- Netiek lietota OpenCL automātiskā atmiņas pārvaldīšana, izdalīšana, tā vietā visos risinājumos konkrētai programmai tiek definēts vienāda izmēra buferis datu straumes ielasīšanai, apstrādei. Lai gan varētu kritizēt faktu, ka netiek lietota kāda potenciāla optimālāka vai ērtāka funkcionalitāte, kura netiek lietota tīri salīdzināšanas vajadzībām, no OpenCL atmiņas iespējām tāpat nāktos daļēji atteikties tādu datu apstrādē, kuri pārsniedz gan VRAM, gan RAM ierobežojumus,
- Līdzīgu iemeslu dēļ netiek lietoti vairāki OpenCL konteksti, kontekstu rindas,
- Starp visām platformām tiek lietoti līdzīgi vai ekvivalenti GPGPU puses atmiņu tipi,
- Ieviestais algoritms un paštaisītās datu struktūras ir vienādas pseidokoda līmenī.

Konkrētas programmatūras etalonuzdevuma izpildē, jādefinē kā tiks mērīti resursi - laiks un atmiņa. Vienkāršākais risinājums laika mērīšanai ir mērīt kopējo CPU laiku (lietotāja un sistēmas), kas paterēts programmas izpildei, kā, piemēram, ar jau iepriekš minēto time utilītu.

Tā kā mērķis ir analizēt GPGPU platformas un to konkrētu funkcionalitāšu ātrdarbību, tad tikai ar kopējo CPU laiku nepietieks (kā tas tika darīts autora kursa darbā analizējot CUDA un ROCm HIP platformas[5]). Tiks pieskaitīts laiks, kas iespējams nav tik svarīgs vai vispār saistīts ar konkrēto platformu, kādu tās API funckiju, piemēram, CPU puses faila apstrāde.

Kā min viena publikācija [18], mērīt pilno programmas izpildes laiku var nebūt pietiekoši, piemēram, ar textittime utilītu, it īpaši vairākpavedienu scenārijos. Lai gan minētajā pētījumā etalonuzdevumu sastādīšana apspriesta tīri CPU izpildes kontekstā, vairākpavedienu izpildi un to ietekmi no I/O darbībām var attiecināt arī priekš GPU.

Ja gala mērķis ir sasniegt etalonuzdevuma atdarināmību, tad faila apstrādes, datu sagatavošana un vispārīgi darbības ar disku var negatīvi ietekmēt rezultātus. Uzstādījumā var tikt izmantota neproprocionāli lēna datu uzglabāšanas iekārta attiecībā pret GPU.

Lai izvairītos no šādām problēmām, uzstādījumā jādefinē plāšāks analizējamo datu

klāsts. Lai izprastu konkrētās platformas stiprās un vājās puses, tiek noteikts, ka individuāli jāmēra:

- CPU puses datu apstrāde, sagatavošana
- GPU iekārtas, vides sagatavošana, platformas inicializēšana
- GPGPU kodolu izpilde
- Datu pārsūtīša no/uz GPU

Uzticamam etalonuzdevumam jānodrošina konsekventa un ideālā gadījumā deterministiska vide un programmas izpilde. Kā minēts Dirk Beyer, Stefan Löwe un Philipp Wendler darbā "Reliable benchmarking: requirements and solutions" [18], ir vairāki parametri (aparatūras arhitektūra, I/O darbības, laika mērīšanas rīku nianses, atmiņas iedalīšana), kuri var pēc nejaušības principa ietekmēt rezultātus.

Autori analizē arī citas potenciālās problēmas un izvirza 6 galvenās prasības uzticamākiem etalonuzdevumiem: [18]

- Precīzi jāmēra un jāierobežo resursi,
- Procesi jāizbeidz uzticami,
- Procesiem apzināti jāpiesķir kodoli,
- Jāņem vērā nevienveida atmiņas piekļuve (NUMA) vairākprocesoru situācijās,
- Jāizvairās no mijmaiņas,
- Jāizolē individuāli laidieni.

Izstrādātās prasības ir, galvenokārt, domātas CPU izpildes analīzei. GPU patēriņa mērīšana ir sarežģītāka tīri tā iemesla dēļ, ka nav pieejama vienota arhitektūras saskarne, līdz ar to ārpus daudzu etalonuzdevumu un profilēšanas rīku tvēruma. [18]

Risinājums būtu lietot konkrēto videokaršu izstrādātāju piedāvātos rīkus (Nvidia Nsights[19] Systems un AMD ROCm Systems Profiler[20]), bet, analizējot OpenCL programmas caur tiem, tiks iegūta tikai zemā līmeņa informācija par to ko OpenCL ir "nodevis" GPU.

Diemžēl ar Nvidia Nsights un AMD ROCm Systems Profiler nav iespējams iegūt konkrētu informāciju (kaut vai par GPGPU kodolu izpildes ilgumu) no OpenCL programmām, šāds atbalsts vispār netiek sniegts.[21] OpenCL gadījumā ārējie profilēšanas rīki nav pieejami, nākas pašam apieties ar pieejamajām API funkcijām.

Lai nodrošinātu kaut cik līdzīgu uz attiecīgajām platformām izstrādāto programmu konsekvenci, visām programmām jāsatur attiecīgie ekvivalentie GPU notikumu profilēšanas izsaukumi, kuri tiek vienādi apstrādāti un žurnalēti programmas izpildes laikā. Piemēram, ar platformu API funkcijām *cudaEventRecord*, *hipEventRecord*, *clGetEventProfilingInfo*.

Lai ievērotu 6 iepriekšminētās prasības, tiek piedāvāti šādi risinājumi:

3.1 Jāizolē individuāli laidieni

Izolācijai tiks izmantots programmatūras virtualizācijas rīks - Docker. [22]

3.2 Precīzi jāmēra un jāierobežo resursi

Mērīšanai tiks izmantotas attiecīgo platformu GPUGPU notikumu profilēšanas funkcijas. Maksimālais RAM apjoms piešķirts ar Docker (skatīt izdruku 3.6) un laika ierobežojumu apstrādās etalonuzdevuma Python skripts.

Izdruka 3.6: Docker konteinera palaišana, piešķirot tam konkrēti 12GiB RAM

```
1 $ docker run --memory="12g" ...
```

3.3 Procesi jāizbeidz uzticami

Etalonuzdevuma Python skriptam pēc definētā maksimālā izpildes laika ilguma jāapstādina laidienā mērāmā programma.

3.4 Procesiem apzināti jāpiesķir kodoli

Ar Docker tiek iedots konkrēts kodols ar komndarindas opciju (skatīt izdruku 3.7).

Izdruka 3.7: Docker konteinera palaišana, piešķirot tam konkrēti pirmo CPU kodolu

```
1 $ docker run --cpuset-cpus="0" ...
```

3.5 Jāņem vērā nevienveida atmiņas piekļuve (NUMA) vairākprocesoru situācijās

Tā kā CPU puses kods neizmanto pavedienus un tā kā šo etalonuzdevumu ietvaros izmantotā aparatūra satur tikai vienu CPU, par šo problēmu nav jāuztraucas.

3.6 Jāizvairās no mijmaiņas

Lai nodrošinātu mijmaiņas neizmantošanu Linux operētājsistēmā var īslaicīgi izslēgt mijmaiņas krātuvi (skatīt izdruku 3.8).

Izdruka 3.8: Mijmaiņas krātuves izlēgšana uz Linux

```
1 $ swapoff -a
```

Kā konfigurācijas redundanci, Docker konteineriem iespējams arī definēt kā pats Docker dzinējs veiks vai neveiks mijmaiņu ar konkrētu Docker konteineri (skatīt izdruku 3.9).

Izdruka 3.9: Mijmaiņas krātuves izlēgšana Docker konteinerim

```
1 $ docker run --memory="12g" --memory-swap="12g" --memory-swappiness=0
      ...
```

Izmantojot iepriekš definēto atmiņas ierobežojumu, ja tiek noteikts mijmaiņas krātuves un RAM atmiņas kopējais ierobežojums (*–memory-swap*) vienādā lielumā ar RAM ierobežojumu, tad būtībā mijmaiņas krātuves lielums ir 0.

4.UZDEVUMI

4.1 Uzdevumu izvēles pamatojums

Praktiskai saskarņu salīdzināšanai nepieciešams definēt programmēju problēmu, kuru iespējams izstrādāt uz visām platformām. Jāizvēlas tāds uzdevums, kuru iespējams 'augsti' paralelizēt, tas ir, sadalīt uzdevumu daudzos mazos un (it īpaši) neatkarīgos gabalos, lai būtu pievienotā vērtība (ātrdarbība) to pildīt uz GPU, ņemot vērā papildus darbu un nepieciešamās zināšanas, lai ieviestu GPU risinājumu.

Tā kā uzdevuma risinājumu primārais mērķis ir tos izmantot etalonuzdevumā, tad tam ir jābūt tādam, ar kuru iespējams notestēt kādu biežu lietotu GPGPU platformu funkciju apakškopu.

Viena uzdevuma ietvaros varētu rasties šaubas par etalonuzdevuma rezultātu ticamību, tas ir, iespējams radītie risinājumi, uzstādījums vai pats uzdevums radījis kādu īpašu situāciju, kurā kāda konkrēta platforma noteiktos mērījumos izvirzās vadībā vai tieši pretēji, kas nebūtu tiesa citās reālās programmās ar pievienoto vērtību. Tāpēc tiek izvirzīta prasība pēc vismaz diviem dažādiem uzdevumiem.

Par vienu uzdevumu tiks izmantots jau autora kursa darbā definētais un izstrādatais uzdevums (uz CUDA un ROCm, bet ne OpenCL) - paroļu lauzējs, atguvējs priekš SHA-256 jauktām, šifrētām parolēm.[5]

Nobriedušāki paroļi lauzēji, jeb precīzāk - paroļu atkopēji kā "hashcat"[23] 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, jeb tā sauktais 'vārdnīcas' tipa uzbrukums.[24]

Šādā uzdevuma implementācijās būs nepieciešamība izmantot datu straumi parolēm, pie atbilstošas paroles atrašanas, jānodrošina atomiska datu ierakstīšana no attiecīgā veiksmīgā GPGPU pavediena. Kā arī šim uzdevumam, ņemot vērā definētu straumes bufera izmēru un apstrādājamo paroļu skaitu, ir fiksēts izpildāmo GPGPU kodolu un laidienu skaits. Daudzos praktiskos scenārijos GPU resursi tiek izmantoti ilgtermiņa darbībām, kur izpildāmo kodolu skaits varētu nebūt definēts, piemēram, mākslīgā intelekta mašīn-

mācīšanās modeļi, kuri tiek trenēti kamēr sasniegts kāds kvalitātes kritērijs.

Noteiktajos etalonuzdevuma prasību ietvaros tomēr vajadzētu kādu statiski nosakāmu izpildes apjomu (cik reizes kodols tiks darbināts), tāpēc, lai notestētu šādu ilgtermiņa izpildi tiek noteikts otrs uzdevums - Džona Konveja dzīves spēles [25] realizācija, kurā izpildes apjoms definējams kā šūnu automāta darbības soļi.

Šīs šūnu automāts ir viens no zināmākajiem tā vienkāršo noteikumu un iespējamās kompleksās šūnu mijiedarbības dēļ (ar to iespējams uzkonstruēt jebkuru Tjūringa mašīnu).[25] Šūnas ir izkārtotas 2 dimensionālā režģī, virmsā. Katra šūna ir vai nu dzīva vai mirusi un nākamajā solī šūnas stāvoklis ir atkarīgs no tās tiešajiem 8 kaimiņiem:

- ja šūna ir mirusi:
  - ja tai ir precīzi 3 kaimiņi, tā nākamajā solī būs dzīva,
  - jebkurā citā situācijā tā paliks mirusi.
- ja šūna ir dzīva:
  - ja tai ir 2 vai trīs kaimiņi, tā paliks dzīva,
  - ja tai vai nu mazāk par 2 vai vairāk par 3 kaimiņiem, tā nākamajā solī nomirs (it kā vai nu no nepietiekamas apdzīvotības vai pārapdzīvotības).

Uzdevumu ir vērts pildīt uz GPU, jo katru automāta šūnu iespējams apstrādāt savā pavedienā gandrīz vai kā 2D bildi, kur katrs pikselis atbilst katrai automāta šūnai. Papildus sarežģītība un iespējamās platformu ātrdarbības atšķirības meklējamas GPU atmiņas datu izveidē un apmaiņā pie liela spēles soļu skaita, jo ar katru soli nākas mainīt ieejas datus, izejas datus.

4.2 Programmatūras, aparatūras uzstādījums

Tā kā programmas notikumu ilgumi dažādās konfigurācijās nāks no CPU, CUDA, HIP un OpenCL pusēm, tad lai būtu vienota un viegli apstrādājama datu kopa, notikumi tiks žurnalēti failā, izmantojot, spdlog[26], C++ žurnalēšanas bibliotēku.

Ērtai datu apstrādei, izmantojamais faila formāts ir CSV, un datu formāts skatāms izdrukā 4.1.

Izdruka 4.1: Etalonuzdevuma žurnālfaila ieraksta formāts

```
1 <Platformas tips>,<Notikuma apraksts>,<Ilgums, ms>
```

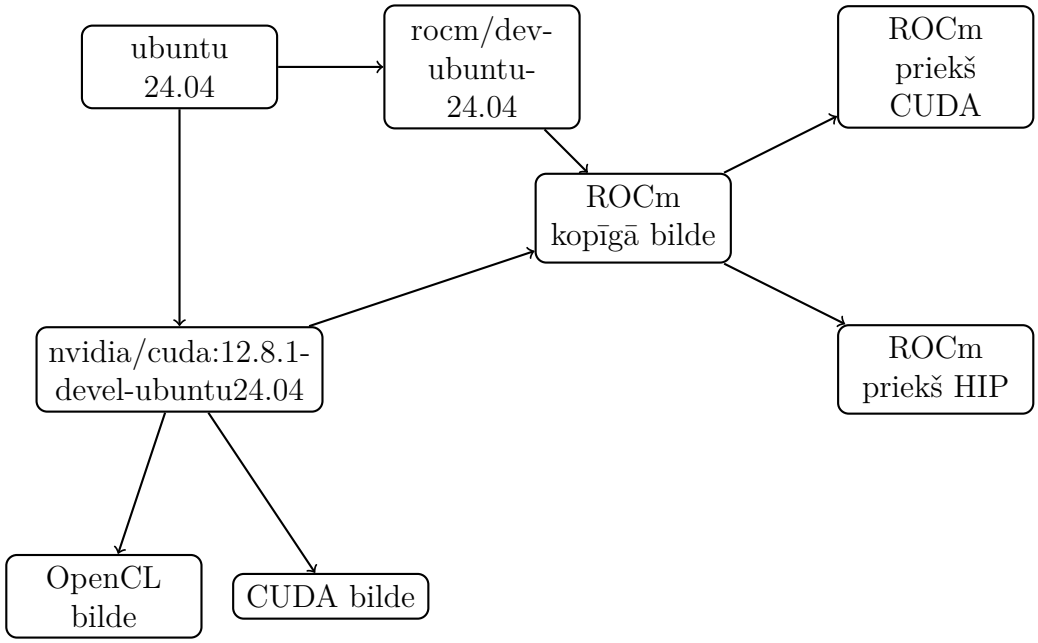
```
OpenCL, Kodola izpildes laiks, 2.0736
CUDA, Datu parraide uz GPU, 1.83927
HIP, Atminas malloc uz GPU, 6.42058
```

Notikuma ilgums norādāms milisekundēs, jo, galvenokārt, tādā mērvienībā notikumu ilgumu noklusēti atgriež gan CUDA, gan HIP profilēšanas funkcijas *cudaEventElapsedTime*, *hipEventElapsedTime*. OpenCL funkcija *clGetEventProfilingInfo* atgriež notikuma ilgumu nanosekundēs, šajā gadījumā rezultāti tiks pārveidoti, lai nodrošinātu konsekvenci starp pārējām platformām.

Uzdevumu izstrādei, testēšanai un etalonuzdevumu izpildei izmantots dators ar:

- Arch Linux operētājsistēmu ar Linux 6.14.5-arch1-1 OS kodolu
- AMD Ryzen 5600H procesoru ar Radeon Graphics integrēto videokarti,
- Nvidia GeForce RTX 3060 Laptop ārējo videokarti.

Lai nodrošinātu risinājumu izolāciju caur Docker, visu konteineru izveidē par bāzes Docker bildi izmantota Ubuntu 24.04. Par pamata Docker failu HIP risinājumam izmantots AMD izstrādatais [27] ar sagatavotu vidi ROCm un CUDA platformām. Līdzīgi priekš CUDA izmantota Nvidia sagatavotā bilde[28], kurai pamatā ir Ubuntu (pilnās Docker bilžu atkarības skatīt attēlā 4.1).



Att. 4.1: Docker pamatbilžu atkarības

Pašu programmu izstrādei un kompilēšanai izmantots sekojošs programmatūras steks ar fiksētām versijām:

- Make 4.4.1-2
- CMake 4.0.1-2
- gcc 15.1.1
- CUDA, nvcc 12.8.97
- ROCm HIP, hipcc 6.3.4
- spdlog 1.15.2-1

4.3 Paroļu atguvēja īstenojums

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

Programma ir paredzēta kā komandrindas utilīta ar CLI argumentiem:

- ievades faila ceļš,
- paroles jaucējvērtība,
- žurnālfaila ceļs

Apstrādājamais fails saturēs potenciālās paroles (piemēru skatīt izdrukā 4.2, katra savā rindā, kurām tiks izrēķināta jaucējvērtība un salīdzināta pret doto. Izmantojamais jaukšanas algoritms - SHA256. Jaucējvērtību rēķināšana un salīdzināšana jārealizē izpildei uz GPU. [5]

Izdruka 4.2: Paroļu ieejas faila piemērs ar nejauši ģenerētām parolēm

```
5QW8ZywwXSQ8I
YIqBz6PgEbDg3AK
qs0xxkco3AYaX
zRX5l1
qyGgl8Dg
cvJTiLCw7e
ySBinJCvd
7cDTl7RsdLFhD
...
```

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.

Lai notestētu SHA-256 aprēķinu darbību pret zināmām jaucējvērtībām un to attiecīgajiem ziņojumiem, kā arī, lai pārbaudītu vispārīgu GPGPU kodola programmas darbību, platformas pieejamību uz šī datora apartūras, un veiktu galveno programmas izpildes, programma jādarbina no komandrinas, padodot atbilstošos argumentus (skatīt izdruku 4.3). [5]

Izdruka 4.3: Programmas galvenā izpilde

```
$ pwcracker <parolu faila cels> <paroles hash vertiba> <zurnalfaila
    cels>
```

Algoritmu skatīt izdrukās 4.4.

Izdruka 4.4: Paroļu atguvēja CPU puses pseidokods

```
passwords, offsets, count =  readPasswordFromFile(inputFile)
log(fileLoadTime())

passwordsPinnedMemory = mallocPinnedMemory(sizeof(passwords))
offsetsPinnedMemory = mallocPinnedMemory(sizeof(offsets))

devicePasswordsBuffer = deviceMalloc(sizeof(passwords))
deviceOffsetsBuffer = deviceMalloc(sizeof(offsets))
deviceHash = deviceMalloc(sizeof(32)) // sha 256 biti => 32 baiti
deviceOutputIdx = deviceMalloc(sizeof(4)) // int izmers ieks C/C++
log(bufferCreationTime())

hostMemcpy(passwordsPinnedMemory, passwords)
hostMemcpy(offsetsPinnedMemory, offsets)
deviceMemcpy(devicePasswordsBuffer, passwordsPinnedMemory)
deviceMemcpy(deviceOffsetsBuffer, offsetsPinnedMemory)
deviceMemcpy(deviceHash, hash)
deviceMemcpy(deviceOutputIdx, -1)
log(hostToDeviceMemoryTransferTime())


PwCracker(devicePasswordsBuffer, deviceOffsetsBuffer, passwords.count
    (), deviceHash, deviceOutputIdx)
log(kernelExecTime())

if outputIdx != -1
    print(passwords[outputIdx])
```

Izdruka 4.5: Paroļu atguvēja GPGPU kodola pseidokods

```
PwCracker(passwords, pwOffsets, count, hash, outputIdx):
    i = blockIdx.x * blockDim.x + threadIdx.x
```

```
3
4       password = passwords + pwOffsets[i]
5       if i >= pwCount:
6           return
7
8       if sha256(password) == hash:
9           atomicStore(outputIdx, i)
```

4.4 Dzīves spēles šūnu automāta īstenojums

Tāpat kā paroļu atguvējam, programma darbināma no komandrindas, padodot CLI argumentus:

- ievades režģa faila ceļš,
- izejas režģa faila ceļš,
- šūnu automāta izpildāmo soļu skaits,
- žurnālfaila ceļs. (skatīt izdruku 4.6).

Izdruka 4.6: Programmas galvenā izpilde

```
$ gol <ieejas rezga faila cels> <izejas rezga faila cels> <automata
   solu skaits> <zurnalfaila cels>
```

Ieejas un izejas režģu failiem jāsastāv no vieniniekiem un nullēm, reprezentējot faktu vai attiecīgā šūna ir dzīva (1) vai mirusi (0), izdruka 4.7.

Izdruka 4.7: Ieejas, izejas faila piemērs (ar tā saukto planiera rakstu)

```
// ieejas fails:
0000000
0001000
0000100
0011100
0000000

// izejas fails pec viena sola:
0000000
0000000
0010100
0001100
0001000
```

Algoritma pseidokodu skatīt izdrukās 4.8 un 4.9.

Izdruka 4.8: Dzīves spēles šūnu automāta CPU puses pseidokods

```
data, width, height = readGridFromFile(inputFile)
```

```
log(gridReadTime())

inputGridPinnedMemory = mallocPinnedMemory(sizeof(data))
outputGridPinnedMemory = mallocPinnedMemory(sizeof(data))

deviceInputBuffer = deviceMalloc(sizeof(data))
deviceOutputBuffer = deviceMalloc(sizeof(data))
log(bufferCreationTime())

hostMemcpy(inputGridPinnedMemory, data)
deviceMemcpy(deviceInputBuffer, inputGridPinnedMemory)
log(hostToDeviceMemoryTransferTime())

for i in range(0, gameSteps):
    GameOfLife(deviceInputBuffer, deviceOutputBuffer, width, height)
    log(kernelExecTime())

    swap(deviceInputBuffer, deviceOutputBuffer)

deviceMemcpyToHost(outputGridPinnedMemory, deviceInputBuffer)
log(deviceToHostMemoryTransferTime())

writeGridToFile(outputGridPinnedMemory, outputFile)
log(gridWriteTime())
```

Izdruka 4.9: Dzīves spēles šūnu automāta GPGPU kodola pseidokods

```
GameOfLife(inputGrid, outputGrid, width, height):
    x = blockIdx.x * blockDim.x + threadIdx.x
    y = blockIdx.y * blockDim.y + threadIdx.y

    if x >= width || y >= height:
        return

    neighbors = neighborCount(inputGrid[y][x])

    cell = 0

    if input[y][x] == 1:
    if neighbors == 2 || neighbors == 3:
        cell = 1
  else if neighbors == 3:
        ncell = 1;

    outputGrid[y][x] = cell
```

4.5 Risinājumu kopējie ierobežojumi un skaidrojumi

Abu programmu pseidokodi rakstīti pēc iespējas platform-neatkarīgi, bet, lai būtu lielāka skaidrība par implementācijas detaļām, tabulā 4.1 minētas atbilstošās CUDA, HIP, OpenCL funkcijas.

Tabula 4.1: Pseidokodā minēto funkciju atbilstošās funkcijas katrā platformā

Pseidokoda funkcija	**CUDA**	**ROCm HIP**	**OpenCL**
mallocPinnedMemory	cudaMallocHost	hipHostMalloc	clCreateBuffer ar CL_MEM_READ_WRITE, CL_MEM_ALLOC_HOST_PTR karogiem
hostMemcpy	std::memcpy	std::memcpy	std::memcpy
deviceMalloc	cudaMalloc	hipMalloc	clCreateBuffer ar CL_MEM_READ_WRITE karogu
deviceMemcpy	cudaMemcpy	hipMemcpy	clEnqueueWriteBuffer

Mērāmie notikumu ilgumi kā pseidokodā norādīts ar, piemēram, *bufferCreationTime*, *deviceToHostMemoryTransferTime* u. tml., realizējami ar prioritāri attiecīgajām GPGPU API notikumu profilēšanas funkcijām, ja kādā vai visās kādu notikumu nav iespējams izmērīt, vai tas neskaitās kā notikums, tad lietojama C++ standarta bibliotēkas datumu un laiku manipulācijas bibliotēka "chrono".[29]

Izdruka 4.10: "chrono" laika mērīšanas piemērs

```
auto start = std::chrono::steady_clock::now();
// meramais koda gabals
auto end = std::chrono::steady_clock::now();

std::chrono::duration<double, std::milli> elapsedTime = end - start;
```

Izdruka 4.11: CUDA laika mērīšanas piemērs

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
// CUDA API izsaukumi
cudaEventRecord(stop);

float elapsedTime = 0;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

Izdruka 4.12: ROCm HIP laika mērīšanas piemērs

```
hipEvent_t start, stop;
hipEventCreate(&start);
hipEventCreate(&stop);

hipEventRecord(start);
// HIP API izsaukumi
hipEventRecord(stop);
```

```
8
9  float elapsedTime = 0;
10 hipEventElapsedTime(&elapsedTime, start, end);
```

Izdruka 4.13: OpenCL laika mērīšanas piemērs

```
1  cl_event event;
2
3  clResult = clEnqueueNDRangeKernel(kernel, 1, nullptr, globalSize,
       localSize, 0, nullptr,
4
5  clFunkcijaKuraSanemCl_EventParametru(/*...*/, &event);
6
7  cl_ulong start;
8  cl_ulong end;
9
10 clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(
       start), &start, nullptr);
11 clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_COMPLETE, sizeof(
       end), &end, nullptr);
12
13 float elapsedTime = static_cast<double>(end - start);
```

4.6 Ievaddatu ģenerēšana un etalonuzdevuma darbināšana

4.6.1 Ģenerēšanas

Paroļu atguvēja programmām, lai iegūtu nepieciešamos paroļu failus, izmantots Python skripts[5], kas:

- Nejauši ģenerē paroles pēc lietotāja ievadītā paroļu skaita,
- Katra parole ir garumā 6 - 16 un sastāv no ASCII simboliem.

Izvēlētas šādas prasības, jo paroles pārsvarā tiek definētas, izmantojot ASCII simbolus, un tās tiek rakstītas dotajā simbolu diapazonā. [30]

Skripts darbināms caur komandrindu, padodot CLI argumentus paroļu skaitam un faila nosaukumam, ceļam (skatīt izdruku 4.14).

Izdruka 4.14: Paroļu faila ģenerēšanas skripta darbināšana

```
1  $ python3 ./pwgen.py <parolu skaits> <izejas faila cels>
```

Līdzīgi izveidots Dzīves spēles režģa ievadfaila ģenerēšanas skripts, kurš katru šūnu nejauši aizpilda ar 1 vai 0. Skripts darbināms caur komandrindu, padodot CLI arguments režģa platumam un garumam, faila nosaukumam (skatīt izdruku 4.15).

Izdruka 4.15: Režģa failaģenerēšanas skripta darbināšana

```
$ python3 ./gridfile_gen.py <platums jeb x> <garums jeb y> <izejas
    faila cels>
```

4.6.2 Etalonuzdevumu skripts

Ņemot vērā izveidotās programmas un etalonuzdevumu izveides apsvērumus, definētās katra uzdevuma risinājuma dažādas laidienu konfigurācijas.

Paroļu atguvējam noteikti sekojoši parametri: [5]

- Risinājuma platforma:
  - CUDA
  - HIP
  - OpenCL
- Paroļu apjoms failā:
  - 10 000 paroles
  - 1 miljons paroles
  - 100 miljonu paroļu
- Laušanas rezultāts:
  - Veikmsīgi atrasta parole
  - Neatrasta parole
- Uzlaužamās paroles atrašanās vieta failā:
  - 1. kvartile (pirmie 25%)
  - 2. kvartile
  - 3. kvartile
  - 4. kvartile

Tā kā tiek lietota ievaddatu straumēšana, tomēr kaut kādā apmērā notiek secīga elementu apstrāde, dati tiek ielasīti un izņemti no GPU atmiņas, un kodoli laisti pēc

kārtas. Līdz ar to izpildes ilgums pie lieliem failu izmēriem ir atkarīgs no paroļu skaita. Lai redzētu šo ietekmi un atšķirības starp platformām, noteikta laidiena konfigurācija ar dažādiem paroļu garumiem un paroles atrašanos failā, tās pozīciju.

Līdzīgi definēta Dzīves spēles laidiena konfigurācija:

- Risinājuma platforma:
  - CUDA
  - HIP
  - OpenCL
- Ieejas režģa izmērs
  - 100x100
  - 1000x1000
  - 10000x10000
- Automāta soļu skaits:
  - 1
  - 100
  - 1000
  - 10000

Viena no etalonuzdevumu prasībām ir lieki nenoslogota sistēma, tāpēc, lai etalonuzdevuma skripts mazinātu savu ietekmi uz sistēmu un attiecīgi pēc iespējas mazāk ietekmētu programmas, mērķtiecīgi tiek iegūti visi iespējamie dati darbam pirmsapstrādes solī, kurš beidzas ar piespiedu kārtā izsauktu Python drazu savācēja, mazinot lieko RAM patērīņu.

Pēc laidienu konfigurācijām etalonuzdevuma pirmsapstrādes solī ar atbilstošajiem failu ģenerēšanas skriptiem tiek izveidoti ievadfaili, un failu direktorijas, ja tādas neeksistē. Failu struktūra sākot no skripta palaišanas saknes ir šāda:

- Ģenerētie ieejas dati: ./benchmark_data/input
- Programmu izveidoti izejas dati: ./benchmark_data/output

- Programu izveidotie žurnālfaili: ./benchmark_data/logs

Tālāk katram paroļu failam pēc atbilstošajām pozīcijām tiek atrastas un nošifrētas meklējamās paroles, katrai laidiena konfigurāciju permutācijai sagatavota komandrindas teksta virkne, ar kuru tiks palaists atbilstošais Docker konteineris, ar atbilstošo ievadfailu, žurnālfailu un izejas failu vai paroles jaucējvērtību.

Katra laidiena konfigurācija tiek darbināta vairākas reizes un operētājsistēmas kešatmiņa tiek piespiedu kārtā iztīrīta pēc katra laidiena. Laidienu reižu skaits tiek padods caur komandrindu.

Izdruka 4.16: Etalonuzdevumu darbināšanas Python skripta darbināšana

```
$ python3 benchmark.py <konfiguracijas darbinasanas skaits>
```

Lai atvieglotu tālāku žurnālfailu apstrādi, skriptā to failu nosaukumi tiek veidoti pēc aprakstoša šablona, saturot galveno informāciju par konkrēto laidienu (skatīt izdruku 4.17).

Izdruka 4.17: Žurnālfailu nosaukumu šabloni

```
# Parolu atguvejs
sha256_<parolu skaits>_<pozicija>_<platforma>_{laidiena nr}.log

# Dzives spele
gol_{platums}x{garums}_steps_{soli}_{platforma}_{laidiena nr}.log
```

4.7 Risinājumu repozitorijs, failu struktūra, kompilēšana, darbināšana

Bakalaura darba repozitorijs uzdevumu risinājumiem, ievadfailu ģenerēšanas un etalonuzdevuma Python skriptiem publicēts un pieejams GitHub.[31] Projekta saknē atrodas 7 direktorijas (katra uzdevuma risinājumi katrā platformā un Python skripti).

Attiecīgo projektu no tā direktorijas saknes iespējams kompilēt ar *cmake* vai ar *Docker* (skatīt izdrukas 4.18 un 4.19).

Izdruka 4.18: OpenCL un CUDA risinājumu kompilēšana

```
# Kompilesana

$ cmake -S . -B build
$ cmake --build build

# Uz Docker

$ docker buildx build . -t golcl
```

Izdruka 4.19: ROCm HIP risinājumu kompilēšana

```
# Kompilesana
$ cmake -S . -B build # (prieks ROCm)
$ cmake -S . -B build -D GPU_RUNTIME=CUDA # (prieks CUDA)
$ cmake --build build

# Uz Docker
$ docker buildx build . --target cuda -t golhip:cuda #(prieks CUDA)
$ docker buildx build . --target rocm -t golhip:rocm #(prieks ROCM)
```

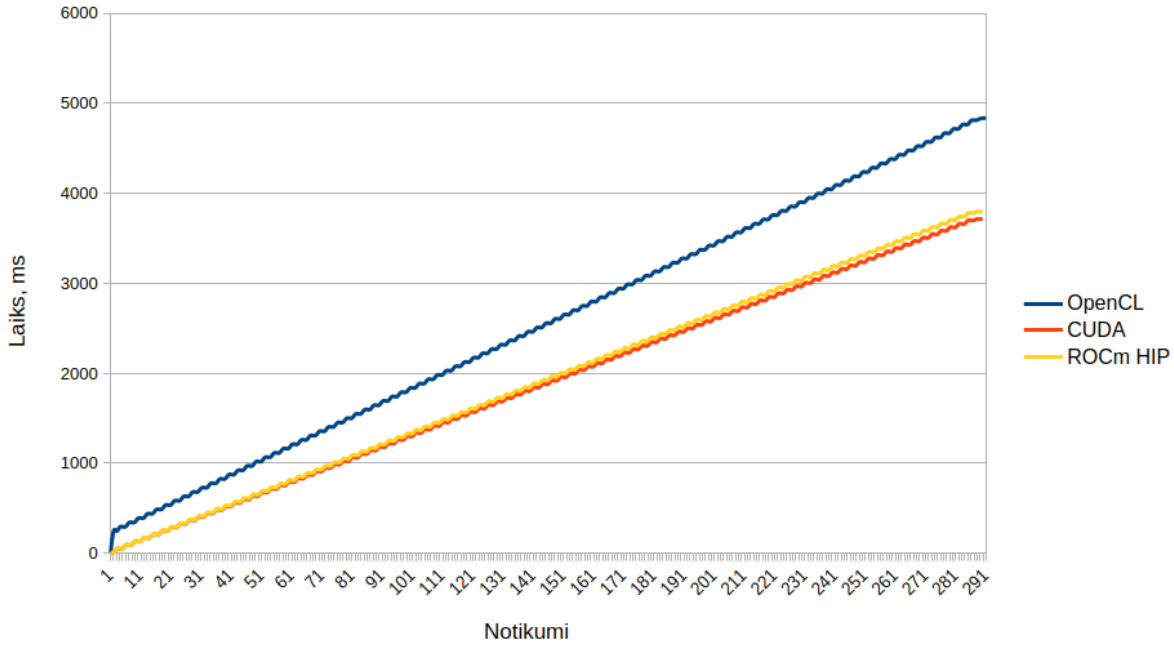
Ja programmas kompilētas pa taisno ar CMake, tad caur komandrindu, padodot CLI arguments, var darbināt kā norādīts 4.3 un 4.4 nodaļās. Darbināšanai ar Docker sākumā jānorāda GPU izmantošana, un jāmontē Docker sējums ievaddatiem un izvaddatiem (skatīt izdruku 4.20).

Izdruka 4.20: Docker konteinera darbināšanas konfigurācija

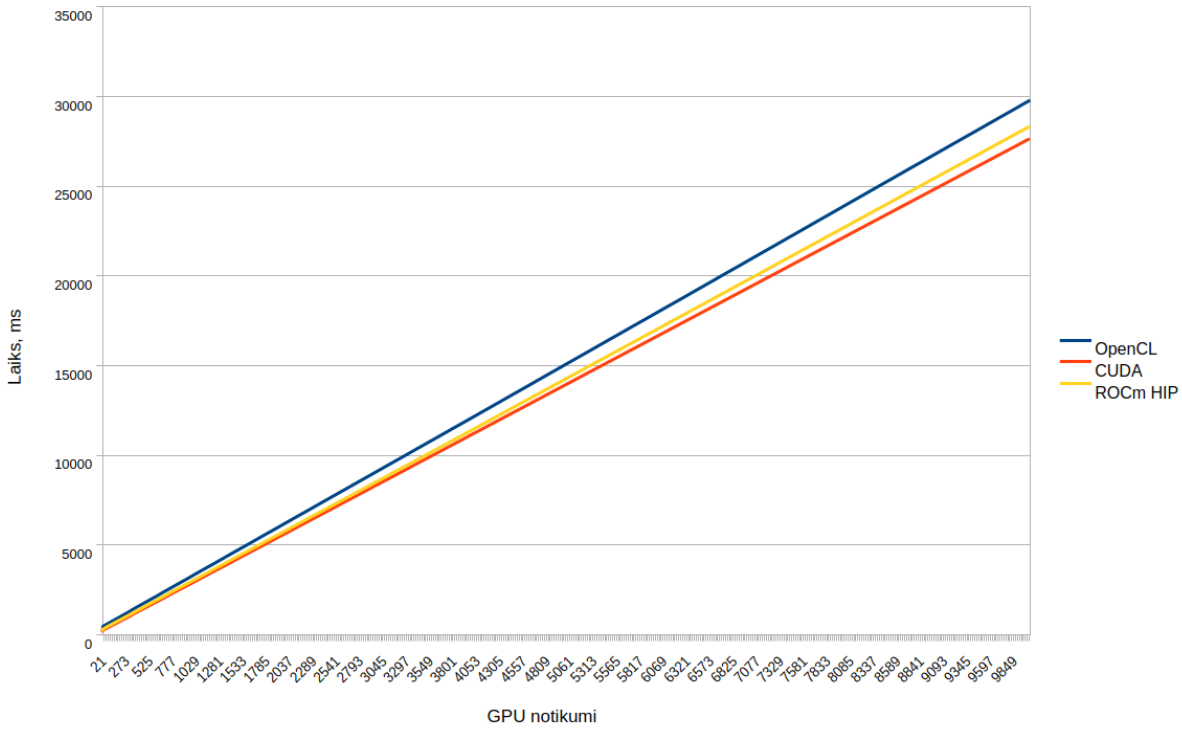
```
$ docker run --rm --gpus all -v <datu direktorija>/:/data <konteinera
    nosaukums> /data/<apstradajamais fails> /data/<izvada fails> <
   citas opcijas ...>

# Parolu atguvejs
$ sudo docker run --rm --gpus=all -v <datu direktorija>:/data <
   konteinera nosaukums> /data/<parolu faila cels> /data/<paroles
   hash vertiba> /data<zurnalfaila cels> /data/<zurnafails>

# Dzives spele
$ sudo docker run --rm --gpus=all -v <datu direktorija>:/data <
   konteinera nosaukums> /data/<ieejas rezga faila cels> /data/<
   izejas rezga faila cels> /data<automata solu skaits> <zurnalfaila
   cels>
```

5.DATU ANALĪZE

Tā kā no žurnālfailiem ir iegūti dažādi ar GPU izpildi saistītu notikumu izpildes laiki, kuri starp risinājumiem ir pēc iespējas ielikti ekvivalentās vietās, tad vienkārši apskatot tekošo kopējo uzkrāto (kumulatīvo) summu, var vizualizēt katras platformas izpildes laikus pret pietiekami līdzīgiem atskaites punktiem (skatīt attēlus 5.1, 5.2).



Att. 5.1: Paroļu atguvēja izpildes laiki 100m parolēm (parole netika atrasta), datu straumes izmērs $2^{20} = 1048576$



Att. 5.2: 10000x10000 Dzīves spēle ar 10000 soļiem

Pēc grafiem var apstiprināt to, kas jau tika noskaidrots kursa darbā saistībā ar CUDA un ROCm HIP - ROCm HIP ir neliela virsdarbe, salīdzinot ar CUDA.[5] Saistībā ar OpenCL platformu, katrs notikums ir, relatīvi runājot, ar daudz lielāku virsdarbi.

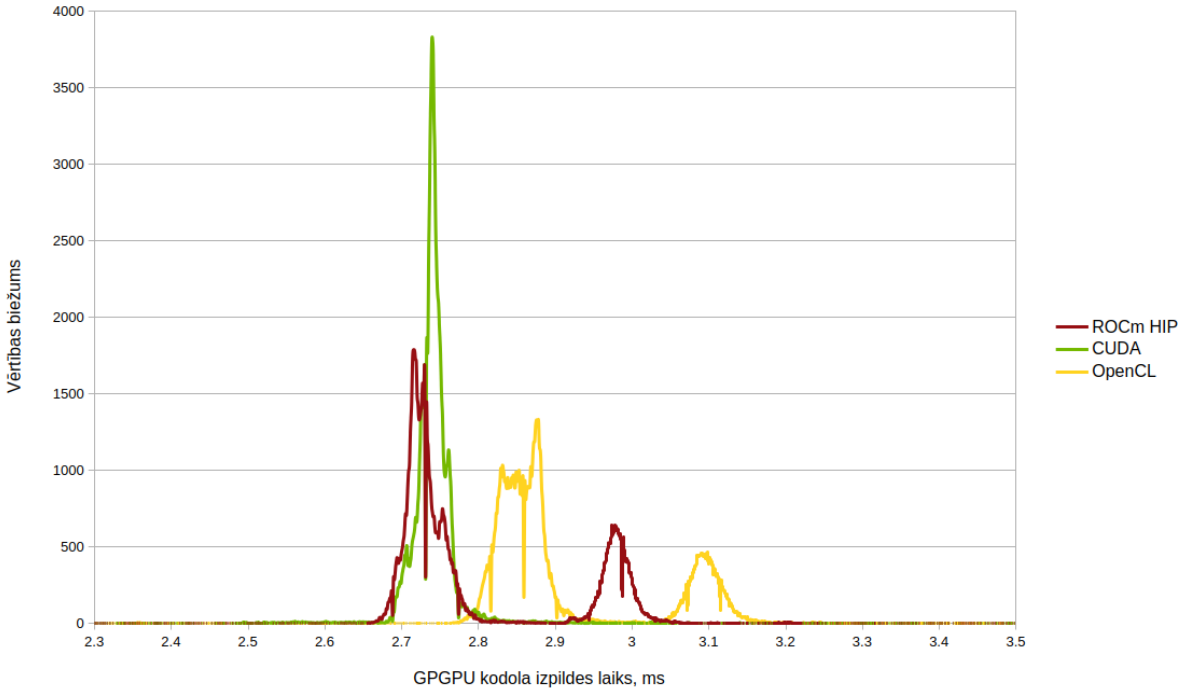
Pēc jaunajiem etalonuzdevumiem var secināt, ka vidēji paroles atgūšanas uzdevumā OpenCL ir par 30.18% lēnāks nekā CUDA un Dzīves spēles uzdevumā par 7.71% lēnāks. Bet, salīdzinot HIP ar CUDA, AMD platforma ir par 2.25% un 2.47% lēnāka.

Atsevišķi apskatot specifiskus GPGPU notikumus kā kodolu izpilde, starp platformām varētu atrast kodolu izpildes nestabilitāti, tas ir, izpildes laiki varētu būt ļoti dažādi, ar lielu variāciju, līdz ar to, ja apskata viena veida laidienu konfigurāciju starp platformām kā izpildes laiku histogrammu, varētu noskaidrot cik stabila ir kodolu izpilde.

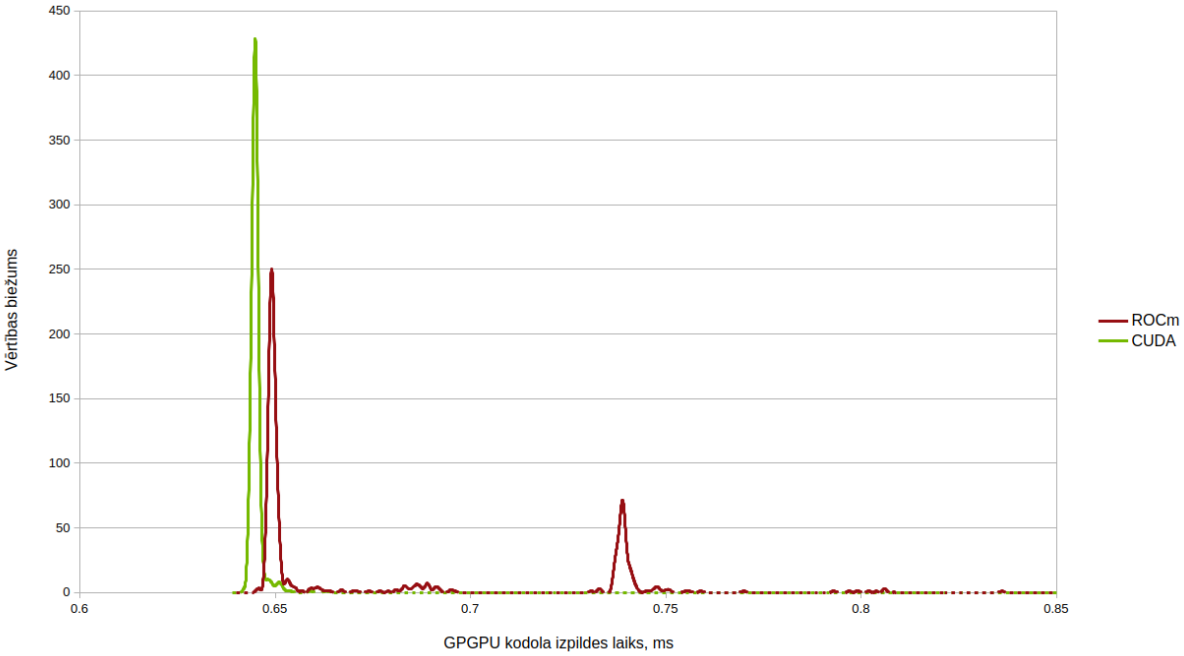
Tā kā izmantotās videokartes (un CUDA, HIP platformu definētā) notikumu precizitāte ir līdz ±0.5µs, histogrammas "spaiņa" (angļu val. *Histogram Bin*) platums noteikts kā 1µs - nav pārāk šaura, lai nerastos kļūdas dēļ datu neprecizitātes, un nav pārāk plata, ka tiktu zaudēta izšķirtspēja. [32] No analizējamajiem datiem izņemtas pašas pirmās kodolu izpildes programmu dzīves ciklā, kuras var saturēt inicializācijas un kešatmiņas virsdarbes.

Līdzīgs risinājums izmantots priekš paroļu atguvēja žurnālfailu datiem, ar papildus

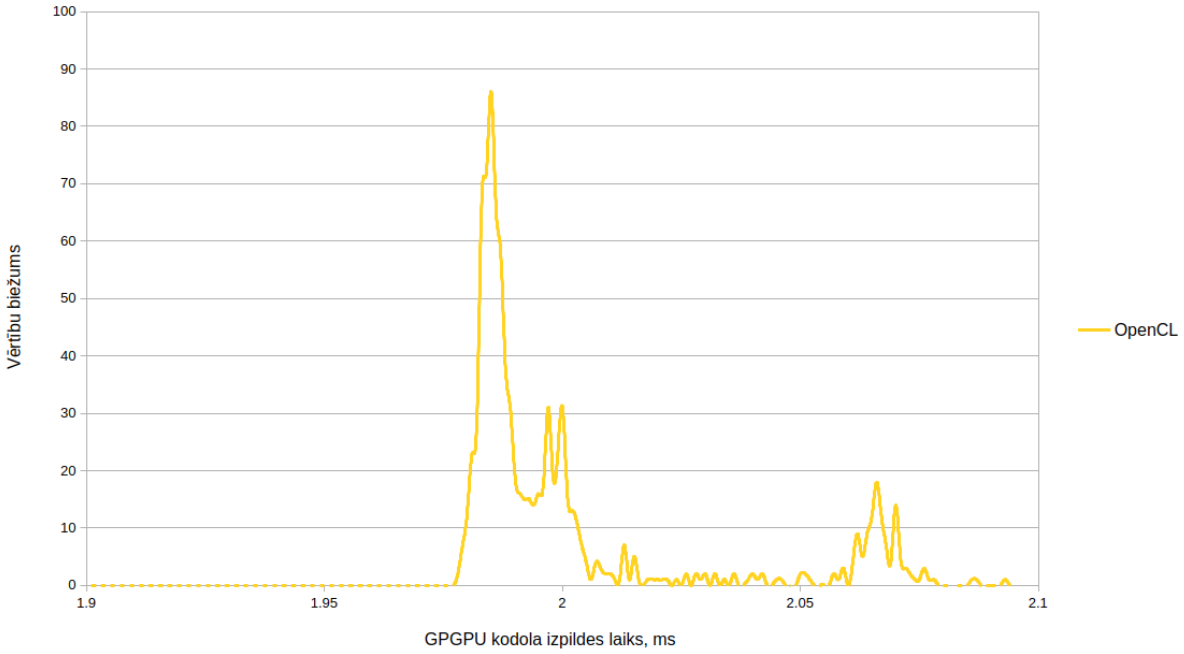
datu filtrēšanu, tā lai tiktu iekļautas tās kodolu izpildes, kuras pilnībā aizņem visu datu straumes buferi, pretēji kodolu izpildes laiki var būt neparedzami mainīgi.



Att. 5.3: Dzīves spēles kodolu izpildes histogramma (10000x10000 ar 10000 soļiem, 10 laidieni)



Att. 5.4: Paroļu atguvēja izpildes histogramma CUDA un ROCm platformām (100 000 000 paroles, 10 laidieni)



Att. 5.5: Paroļu atguvēja izpildes histogramma OpenCL platformai (100 000 000 paroles, 10 laidieni)

Apskatot histogrammu bildes 5.3, 5.4, 5.5, var redzēt, ka mazākā izkliedētība ir CUDA platformai. Pieņemot, ka dati veido normālo sadalījumu, tabulās 5.1, 5.2 redzamas atbilstošās Standartnovirzes un variācijas.

Platforma	**Standartnovirze**	**Variāciju koeficients**
CUDA	0.0015	0.0023
HIP	0.0408	0.0604
OpenCL	0.0275	0.0137

Tabula 5.1: Platformu paroļu atguvēja kodolu izpildes laiku variācijas

Platforma	**Standartnovirze**	**Variāciju koeficients**
CUDA	0.1613	0.0587
HIP	0.1132	0.0405
OpenCL	0.1228	0.0422

Tabula 5.2: Platformu Dzīves spēles kodolu izpildes laiku variācijas

Interesants novērojums ir fakts, ka abos uzdevumu risinājumos HIP un OpenCL izpildes laiku līknes satur divus pīķus ar diezgan līdzīgu relatīvo attālumu starp šo pīķu

modu. Ar esošo datu kopu diemžēl nav iespējams pietiekami izskaidrot kāpēc šāda veidojas struktūra. Bet iespējama teorija ir GPU aparatūras droselēšana.

Pie lielas noslodzes, kura laika ziņā ir palielināta HIP un OpenCL, varēja iestāties aparatūras takts frekvences samazināšana, lai limitētu sistēmas karšanu. Rezultātā veidojas divi pīķi - viens pie vienas takts frekvences un viens pie otras.

6.REZULTĀTI

Dažādos griezumos apskatīti iegūtie dati no žurnālfailiem. Kopsummā apstrādāti un analizēti 816 faili par CUDA, HIP, OpenCL dažādajiem GPGPU notikumu izpildes ilgumiem

7.SECINĀJUMI

BIBLIOGRĀFIJA

[1] J. Krüger un R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, 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] The Khronos™Group, Khronos Launches Heterogeneous Computing Initiative, https://www.khronos.org/news/press/khronos_launches_heterogeneous_computing_initiative, (Piekļūts: 21.05.2025).

[4] T. M. Aamodt, W. W. L. Fung, T. G. Rogers un M. Martonosi, General-purpose graphics processor architectures (Springer, 2018).

[5] Artūrs Kļaviņš, GPU programmēšana Nvidia CUDA un AMD ROCm saskarnēs, 2024, kursa darbs.

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

[7] Advanced Micro Devices, HIP documentation, (Piekļūts: 04.01.2025).

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

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

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

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

[12] Khronos OpenCL™Working Group, The OpenCL™Specification, (2025) (Piekļūts: 02.04.2025).

[13] Khronos OpenCL™Working Group, The OpenCL™C Specification, (2025) https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_C.html, (Piekļūts: 02.04.2025).

[14] The Khronos SPIR™Working Group, SPIR-V Specification, (2025) https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html, (Piekļūts: 02.04.2025).

[15] The Khronos SPIR™Working Group, SPIR-V: The Standard IR for Parallel Compute and Graphics, (2025) https://www.khronos.org/spirv/, (Piekļūts: 21.05.2025).

[16] Association for Computing Machinery, Artifact Review and Badging - Current, (2020) https://www.acm.org/publications/policies/artifact-review-and-badging-current, (Piekļūts: 12.04.2025).

[17] M. Kerrisk, time(1) — Linux manual page, (2024) https://www.man7.org/linux/man-pages/man1/time.1.html, (Piekļūts: 09.01.2025).

[18] D. Beyer, S. Löwe un P. Wendler, Reliable benchmarking: requirements and solutions, International Journal on Software Tools for Technology Transfer **21**, 1—29 (2019).

[19] Nvidia, Nsight Systems - User Guide, (2025) https://docs.nvidia.com/nsight-systems/UserGuide/index.html, (Piekļūts: 24.04.2025).

[20] Advanced Micro Devices, ROCm Systems Profiler documentation, (2025) https://rocm.docs.amd.com/projects/rocprofiler-systems/en/latest/index.html, (Piekļūts: 28.04.2025).

[21] Advanced Micro Devices, ROCm Systems Profiler features and use cases, (2025) https://rocm.docs.amd.com/projects/rocprofiler-systems/en/latest/conceptual/rocprof-sys-feature-set.html, (Piekļūts: 18.05.2025).

[22] Docker Inc., Docker Engine, https://docs.docker.com/engine/, 2025, (Piekļūts: 17.04.2025).

[23] "hashcat" izstrādātāju komūna, Hashcat, (2024) https://hashcat.net/hashcat/, (Piekļūts: 17.12.2024).

[24] "hashcat" izstrādātāju komūna, Dictionary Attack, (2024) https://hashcat.net/wiki/doku.php?id=dictionary_attack, (Piekļūts: 17.12.2024).

[25] J. Conway, Conway's game of life, Scientific American (1970).

[26] gabime, Fast C++ logging library. https://github.com/gabime/spdlog, 2024, (Piekļūts: 23.04.2025).

[27] Advanced Micro Devices, A collection of examples for the ROCm software stack, Dockerfiles for the examples, https://github.com/ROCm/rocm-examples/tree/amd-staging/Dockerfiles, (Piekļūts: 07.05.2025).

[28] Nvidia, nvidia/cuda:12.8.1-devel-ubuntu24.04, https://hub.docker.com/layers/nvidia/cuda/12.8.1-devel-ubuntu24.04/images/sha256-4b9ed5fa8361736996499f64ece
(Piekļūts: 20.05.2025).

[29] "cppreference" komūna, Date and time library, (2025) https://en.cppreference.com/w/cpp/chrono, (Piekļūts: 19.05.2025).

[30] C. Shen, T. Yu, H. Xu, G. Yang un X. Guan, User practice in password security: An empirical study of real-life passwords in the wild, Computers & Security **61**, 130—141 (2016).

[31] A. Kļaviņš, Risinājumu un etalonuzdevumu repozitorijs, https://github.com/clavinsh/cuda_hip_cl_benchmark, (Piekļūts: 19.05.2025).

[32] D. Freedman un P. Diaconis, On the histogram as a density estimator:L2 theory, Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete **57**, 453—476 (1981).

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: