

Klasifikace dat do pravděpodobnostního rozdělení

KIV/PPR - semestrální práce

Mukanova Zhanel
A22N0130P

4. prosince 2022

Obsah

1	Zadání	5
2	Analýza	6
2.1	Původní nápad	6
2.2	Realizace	6
2.3	Maximum likelihood estimation	6
2.3.1	Gauss	7
2.3.2	Exponenciální rozdělení	7
2.3.3	Poissonovo rozdělení	7
2.3.4	Rovnoměrné rozdělení	7
2.4	Residual sum of squares (RSS)	8
3	Implementace	9
3.1	File Mapping	9
3.1.1	První řešení	9
3.1.2	První pokus optimalizaci	10
3.1.3	Výsledné řešení	10
3.1.4	Algoritmus	11
3.2	Vypočet statistik (První iterace)	15
3.2.1	SMP	15
3.2.2	SMP (optimalizace + vektorizace)	16
3.2.3	OpenCL	16
3.2.4	OpenCL kernel pro sběr statistik	17
3.3	Sestavení histogramu (Druhá iterace)	18
3.3.1	SMP	18
3.3.2	SMP (optimalizace + vektorizace)	18
3.3.3	Porovnání rychlostí běhu SMP	19
3.3.4	OpenCL kernel	19
3.4	Histogram hustoty (Density histogram)	20
3.5	Počítání hodnot RSS	20
3.6	Watchdog	20
4	Uživatelská dokumentace	21
4.1	Ovládání	21
4.2	Výstup	21

5	Analýza výsledků	23
6	Závěr	25
	Literatura	26

1 Zadání

Zadáním semestrální práce bylo vytvořit program, který dokáže přiřadit vstupní data do jednoho ze čtyř rozdělení: Normálního/Gaussovo, Poissonovo, Exponenciálního či rovnoměrného. Program musí na konci výpočtu vypsat hodnoty charakterizující rozdělení a zdůvodnění svého výsledku. Také zadání definuje následující limity, které musí být dodrženy:

- Testovaný soubor bude velký několik GB
- Paměť bude omezená na 1GB.
- Program musí skončit do 15 minut na iCore7 Skylake.

Jako vstupní parametry musí program přijímat:

- **soubor** - cesta k souboru, může být relativní k program.exe, ale i absolutní
- **procesor** - řetězce určující, na kterých procesorech výpočet proběhne, a to zároveň
 - all - použije CPU a všechny dostupné GPU
 - SMP - vícevláknový výpočet na CPU
- **názvy OpenCL zařízení** jako samostatné argumenty - pozor, v systému může být několik OpenCL platforem

Součástí programu je také watchdog vlákno, které hlídá správnou funkci programu. Rozšířila jsem vstupní argumenty o zadání počtu vláken, které budou využité na každém CPU, výběr optimalizovaného řešení a výběr času, jak často musí watchdog kontrolovat výsledky.

2 Analýza

2.1 Původní nápad

Původně jsem počítala, že budu řešit zadání semestrální práce pomocí statistické metody “Kolmogorovův–Smirnovův test” [wik22a]. Během samotné implementace jsem narazila na problém, že podobné metody typu „Goodness of fit tests“ se nehodí pro odhad pravděpodobnostního rozdělení. Nelze pomocí těchto testů zjistit parametry rozdělení, které byly využity pro generování sady dat, což by mi nepomohlo v řešení jednoho z bodů zadání semestrální práce:

Program vypíše hodnoty charakterizující rozdělení a zdůvodnění svého výsledku.

Navíc, algoritmus „Kolmogorovův–Smirnovův test“ vyžaduje provedení řazení dat, což je složitá úloha pro velkou sadu dat.

2.2 Realizace

Mojí druhou myšlenkou bylo vytvořit dvouprůchodový program, který by v prvním průchodu sbíral statistiky a v druhém sestavoval histogram dat. Statistiky jsou nutné, aby se v následujících krocích spočítaly parametry rozdělení, samotný histogram a také pro nalezení chyby mezi pozorovanou a očekávanou frekvencí dat pomocí metody „Residual sum of squares“.

2.3 Maximum likelihood estimation

Maximum likelihood estimation (MLE) je metoda odhadu, která nám umožňuje pomocí vzorku odhadnout parametry pravděpodobnostního rozdělení, které data vygenerovalo. Úkolem MLE je odhadnout skutečný parametr Θ který je spojen s neznámým rozdělením, pomocí kterého byl vytvořen vstupní vzorek dat. Vzorce pro každé rozdělení, které byly použité v této práci vypadají následujícím způsobem:

2.3.1 Gauss

$$\mu = \frac{1}{n} \sum_{i=1}^n x_i$$

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2$$

Kde μ odpovídá střední hodnotě a σ^2 rozptylu [MT21b].

2.3.2 Exponenciální rozdělení

$$\lambda = \frac{n}{\sum_{i=1}^n x_i}$$

Kde lambda λ je parametr exponenciálního rozdělení, který musí odpovídat $\lambda > 0$ podle definici rozdělení. Pomocí lambdy lze dále odvodit střední hodnotu rozdělení [MT21a].

2.3.3 Poissonovo rozdělení

$$\lambda = \frac{1}{n} \sum_{i=1}^n x_i$$

Kde λ je střední hodnota rozdělení a musí odpovídat $\lambda > 0$ podle definici rozdělení [MT21c].

2.3.4 Rovnoměrné rozdělení

$$a = \min(X_1..X_n)$$

$$b = \max(X_1..X_n)$$

Kde $X_1..X_n$ jsou náhodná data. Parametry rovnoměrného rozdělení odpovídají minimální (a) a maximální (b) hodnotám dat.

2.4 Residual sum of squares (RSS)

Ve statistice je **Residual sum of squares (RSS)** [\[Wik22b\]](#) součtem čtverců rozdílů (odchylky předpovězené od skutečných empirických hodnot dat). Je to míra nesouladu mezi daty a modelem odhadu, v našem případě jde o předpokládané rozdělení. Malá hodnota RSS indikuje, že predikovaný model odpovídá zkoumaným datům. Používá se jako optimální kritérium při výběru parametrů a výběru modelu (rozdělení pravděpodobnosti).

$$RSS = \sum_{i=1}^n (y_i - f(x_i))^2$$

V této práci používám RSS na sestaveném histogramu pro nalezení chyby mezi pozorované frekvenci data a očekávanou frekvenci.

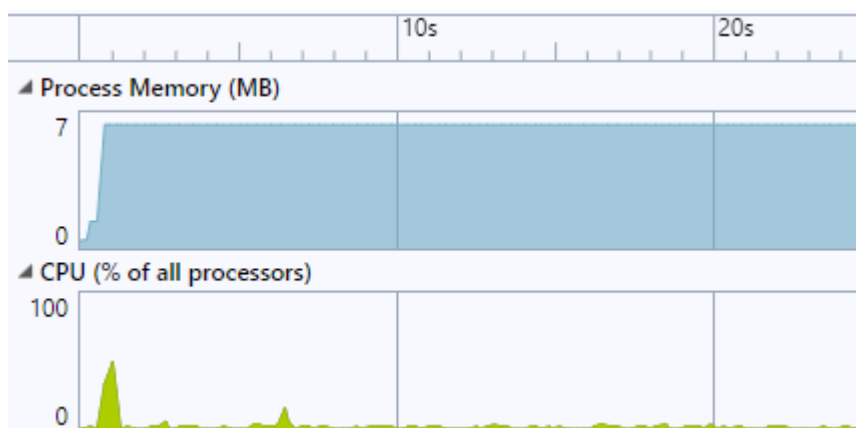
3 Implementace

3.1 File Mapping

S doporučením od pana docenta jsem rozhodla ve své práci využít mapování souboru [Wik21] do adresního prostoru procesu místo klasického čtení ze souboru. Mapování souboru umožňuje programu efektivně pracovat s velkým datovým souborem, aniž by bylo nutné mapovat celý soubor do paměti. Další výhodou “**File Mapping**” je to, že ke sdílené mapované stránce může také přistupovat více vláken. Celé řešení se nachází ve třídě `File_mapping`. Tato třída je mapuje soubor do paměti a rozděluje ho mezi výpočetními vlákny.

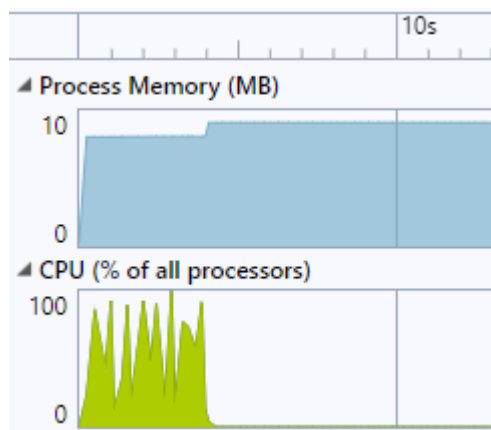
3.1.1 První řešení

V první verzi mapování a zpracování souboru jsem rozhodla dělit celý soubor na menší bloky ke zpracování. Program mapoval postupně každou menší část souboru a rozděloval ho mezi vlákna. Takové řešení mělo několik nevýhod. První bylo, že soubor ležící na disku typu HDD se načítal hodně pomalu (viz obrázek 3.1).



Obrázek 3.1: Ukázka využití CPU při prvním běhu. Zpracování souboru velikosti 7GB, první běh s dobou trvání 10 min

Mnou mapované bloky se v prvním běhu programu nenacházely v paměti RAM, a proto vlákna zbytečně čekala na přesun stránky paměti z disku. Další běhy programu byly o hodně rychlejší (viz obrázek 3.2), nicméně podstatou programu není mnohonásobný běh.



Obrázek 3.2: Ukázka využití CPU při druhém běhu. Zpracování souboru velikosti 7GB, každý další běh s dobou trvání cca 5 sec

Druhá nevýhoda byla v tom, že moje sekvenční řešení, které mapovalo celý soubor najednou, bylo vždycky rychlejší. Vzhledem k tomu, že paralelizace by měla urychlovat výpočet, nikoliv ho zpomalovat, jsem rozhodla řešit problém jiným způsobem.

3.1.2 První pokus optimalizací

Z příkladu výše vyplývá, že doba trvání 10 minut pro první běh programu se vstupními daty 7 GB téměř nesplňuje zadání, kde je definované omezení doby běhu na 15 min. Proto jsem rozhodla dynamicky měnit násobek “allocation granularity” pro výpočet mapovaného bloku. Pokud doba zpracování přesahuje dobu 5 sekund (stanoveno pomocí pokusného běhu), velikost bloku se zmenší. Místo cca 2 GB mapovacích bloků budou mapovány bloky velikosti odpovídající “allocation granularity”. Takový způsob prokázal většího využití CPU. Takto malé bloky stačí načítat pouze v prvním průchodu. Druhý průchod bude rychlejší, protože všechny stránky už budou uloženy v paměti RAM.

Bohužel tato optimalizace neprokázala vysokou spolehlivost výpočtu. Některá data se ztrácela a nedokázala jsem přijít na důvod.

3.1.3 Výsledné řešení

Ze zadání semestrální práce vyplývá omezení, že paměť bude omezená na 1 GB. V případě SMP režimu, mi toto omezení nevadí, protože není alokována žádná paměť pro vstupní data, která se neukládají. Proto jsem rozhodla namapovat celý soubor najednou a takový velký blok rozdělit mezi vlákna

ke zpracování. Příklad algoritmu pro režim SMP je ukázan na obrázku 3.5.

V případě režimu s využitím OpenCL zařízení je potřeba vytvářet buffer s daty, aby se mohl poslat na zařízení. Velikost jednoho bufferu nesmí přesahovat 1 GB podle zadání. Abych nemusela mapovat celý soubor do paměti a mohly se využít výhody mapování menších částí, musela jsem správně nastavit offsety. Z definici od Microsoftu offsety musí být násobkem “allocation granularity” systému, což odpovídá 2^{16} (nebo 32 na 64bitových systémech) bytům [Fre22].

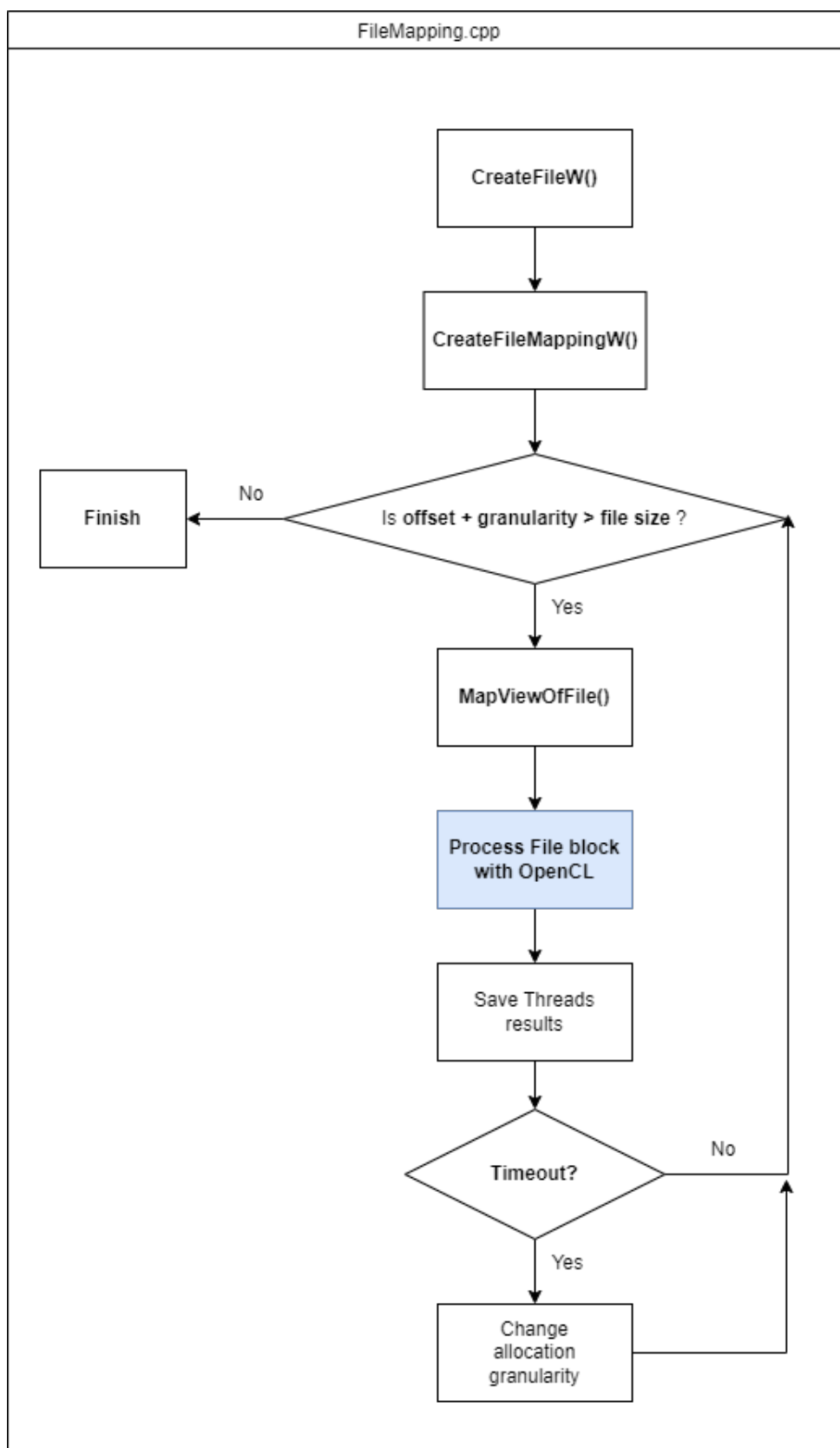
Také kopírování dat na OpenCL zařízení provádí CPU, které hodně zpomaluje výpočet. Proto jsem rozhodla načítat menší bloky s velikostí cca 0,5 GB a pak ho nerozdělovat mezi procesy. Příklad algoritmu pro režim SMP je ukázan na obrázcích 3.3 a 3.4 .

Případné dělení výpočtů bloků mezi procesy s využitím OpenCL se totiž ukázalo jako nespolehlivá úloha, která vracela různé výsledky kvůli „race condition“.

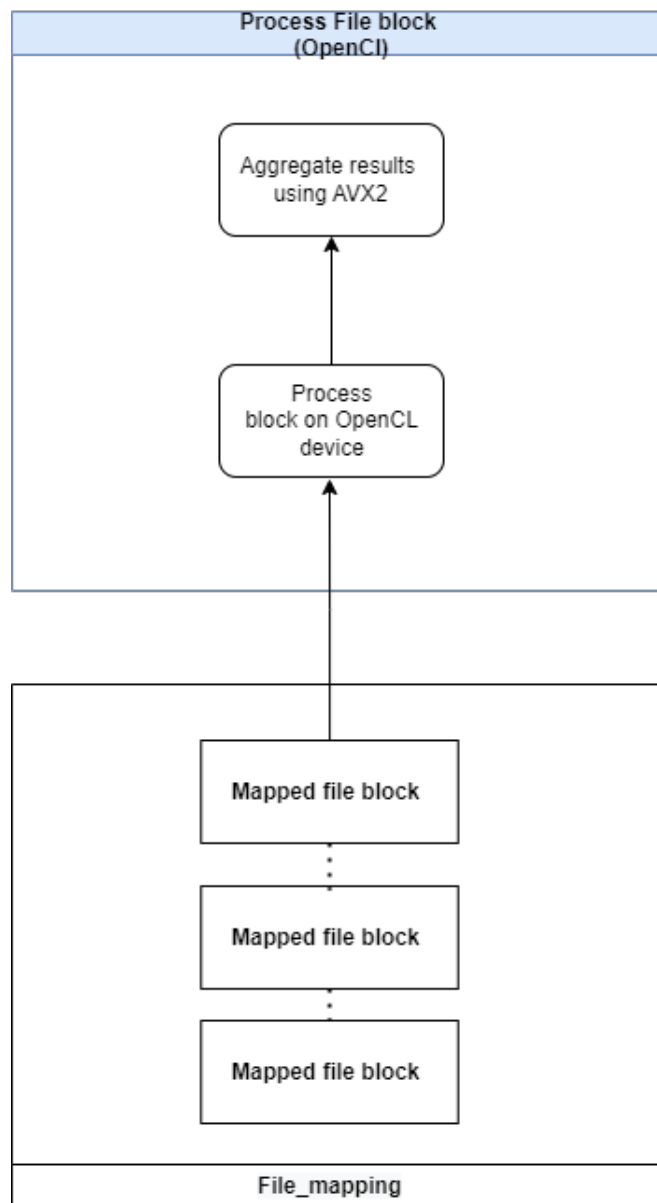
Tímto způsobem se mi podařilo dosáhnout výrazného zrychlení programu v režimu SMP. Režim OpenCL jsem rozhodla nechat v původní implementaci z důvodu omezení paměti programu.

3.1.4 Algoritmus

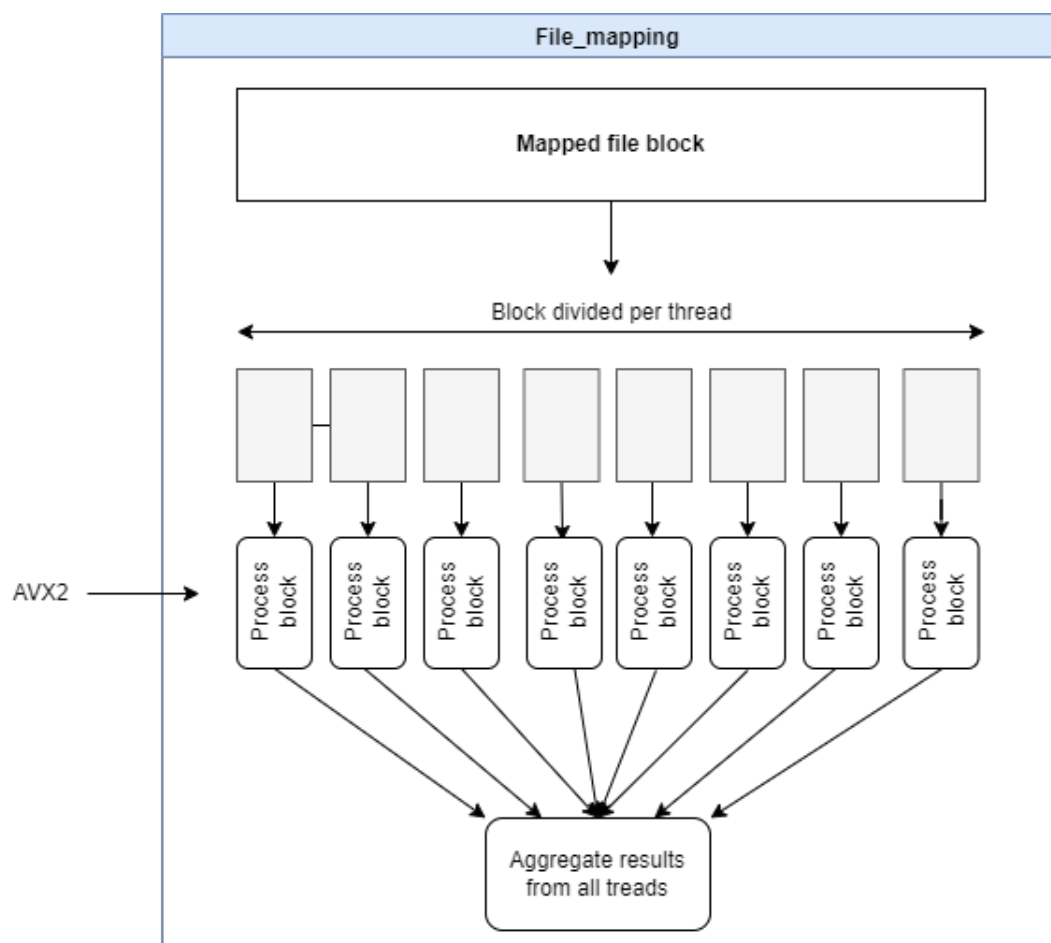
Algoritmus mapování bloků pro dvě iterace je stejný. Liší se pouze funkcí, která se přiděluje vláknům pro zpracování souboru. Vlákna se vytvářejí pomocí `std::async`: `std::async(std::launch::async, /* .. */);`. To zaručuje, že běh zadané funkce bude ve vlastním vlákně, a když vlákno doběhne, vrátí objekt typu `std::future` jako výsledek výpočtu. Výsledky se na konci sjednocují.



Obrázek 3.3: Ukázka 1 zpracování bloků souboru v režimu ALL



Obrázek 3.4: Ukázka 2 zpracování souboru pomocí OpenCL zařízení v režimu ALL



Obrázek 3.5: Ukázka zpracování souboru pomocí vláken v režimu SMP

3.2 Výpočet statistik (První iterace)

První průchod vstupním souborem sbírá potřebné statistiky z dat, které budou využité v dalších krocích. Statistiky se ukládají do následující struktury:

```
1 struct SDataStat
2 {
3     unsigned int n = 0;
4     double sum = 0.0;
5     double max = 0.0;
6     double min = 0.0;
7     double mean = 0.0;
8     double variance = 0.0;
9     bool isNegative = 0;
10 };
```

- **N** – počet čísel v souboru. Používá se hlavně ve Watchdogu, který kontroluje jestli počet prvků po paralelním výpočtu nepřesáhl skutečný počet prvků v souboru.
- **Sum** – suma všech prvků v souboru. Používá se pro odhad parametru pravděpodobnosti.
- **Min/Max** – minimální a maximální prvek v souboru. Používá se pro odhad parametru pravděpodobnosti a sestavení histogramu.
- **Mean** – střední hodnota dat
- **Variance** – odchylka dat (spočítá se během druhého průchodu souborem) **isNegative** – jestli vstupní data obsahují záporná čísla.

3.2.1 SMP

První moje řešení paralelního výpočtu bylo s použitím knihovny TBB od společnosti Intel. Během implementace řešení pomocí algoritmu `tbb::parallel_reduce()` jsem ale narazila na problém, že kontrolní počet čísel v souboru **N** se občas lišil. Ve většině počtu běhů byl správný, ale občas vracel chybné odpovědi. Implementace algoritmu pro sběr statistik se nachází ve třídě `Running_stat_parallel` v souboru `statistics.cpp`.

Také jsem nedokázala zjistit jak využít možnosti manuální vektorizace s použitím této knihovny. V dokumentaci chyběla informace, zda lze nastavovat vlastní intervaly zpracování pro každé vlákno tak, abych mohla zaručit, že každé vlákno bude zpracovávat určitý počet dat.

Po velkém počtu pokusů o využití manuální vektorizace s tímto algoritmem jsem se rozhodla jít jinou cestou a napsat si vlastní algoritmus s použitím manuální vektorizace.

3.2.2 SMP (optimalizace + vektorizace)

Soubor `smp_utils.cpp` obsahuje funkci `get_statistics_vectorized()` pro sběr statistik z dat pomocí AVX2 instrukcí. Tato funkce se přiřazuje každému vláknům ve třídě `File_mapping`, která také rozhoduje kolik dat každému vláknům přidělí. Každé vlákno pak pracuje se svojí částí bloku.

```
1 void get_statistics_vectorized(/* ... */)
2 {
3     __m256d min = _mm256_set1_pd(
4         std::numeric_limits<double>::max()
5     );
6
7     __m256d max = _mm256_set1_pd(
8         std::numeric_limits<double>::lowest()
9     );
10
11     for (int block = 0; block < data_count; block += 4)
12     {
13         // Count elements
14         stat.n += 4;
15
16         // Find sum of 4 vector element
17         __m256d vec = _mm256_load_pd(data + block);
18         stat.sum += hsum_double_avx(vec);
19
20         // Find Max and Min of 4 vector element
21         max = _mm256_max_pd(max, vec);
22         min = _mm256_min_pd(min, vec);
23     }
24
25     // Agregate results
26     /* ... */
27 }
```

3.2.3 OpenCL

Jak jsem popsala v sekci s implementací `File_mapping`, OpenCL u mě měl nevýhodu v tom, že se blok dat musel ukládat do paměti, aby byl přesměrován na OpenCL zařízení. Samotný proces zapsání dat do bufferu se provádí pomocí aktuálního vlákna, což hodně zpomaluje výpočet. Zkoušela jsem vy-

tvářet vlákna, která by zpracovávaly vlastní blok dat (podobně jako SMP) a zapisovala vlastní buffery do sdílené fronty `cl::CommandQueue`. Výsledky výpočtu ovšem nebyly správné. Proto se spouštěl OpenCL kernel pro každý blok dat bez použití vláken, což zaručilo konzistenci zpracovaných výsledků. Kernel vrací `std::vector` s velikostí počtů „work group“. To u velkého počtu dat může znamenat velkou hodnotu. Proto na agregaci výsledků používám funkci s využitím AVX2 instrukci ze souboru `smp_utils.cpp`.

3.2.4 OpenCL kernel pro sběr statistik

Rozhodla jsem využít algoritmus **Parallel Sum Reduction**, který je popsán na této stránce [\[DOU22\]](#), který jsem doplnila o vyhledání maxima a minima. Celý kód pro kernel se nachází v souboru `statistics_kernel.cl`.

3.3 Sestavení histogramu (Druhá iterace)

Druhé procházení souborem sestavuje histogram frekvencí výskytu konkrétních čísel. Frekvenční histogram je pouze mezikrok ve výpočtu RSS. Cílem je sestavit „Density histogram“. Stejně jako v první iteraci se bloky rozdávají vláknům ve třídě `File_Mapping`.

3.3.1 SMP

Jako první pokus o řešení problému jsem začala pracovat s algoritmem `tbb::parallel_reduce()`, který stejně jako v první iteraci vytvářel nespolehlivé výsledky. Implementaci lze nalézt ve třídě `Histogram_parallel` v souboru `histogram.cpp`. Rozhodla jsem provést optimalizaci, abych dokázala použít výhody autovektorizace a zaručit konzistenci výsledků.

3.3.2 SMP (optimalizace + vektorizace)

Algoritmus založený na AVX2 instrukcích, který sestavuje histogram, se nachází ve funkci `get_histogram_vectorized()` v souboru `smp_utils.cpp`.

```
1 void get_histogram_vectorized(/* .. */)
2 {
3     // Fill vector with mean/min and scale value
4     const __m256d mean = _mm256_set1_pd(stat.mean);
5     const __m256d min = _mm256_set1_pd(stat.min);
6     const __m256d scale = _mm256_set1_pd(hist.scaleFactor);
7
8     for (int block = 0; block < data_count; block += 4)
9     {
10         __m256d vec = _mm256_load_pd(data + block);
11         // Compute variance of 4 vector elements
12         variance += variance_double_avx(vec, mean);
13
14         // Find position for 4 elements
15         __m256d position = position_double_avx(vec, min,
16         scale);
17         double* pos = (double*)&position;
18
19         local_vector[static_cast<size_t>(pos[0])] += 1;
20         local_vector[static_cast<size_t>(pos[1])] += 1;
21         local_vector[static_cast<size_t>(pos[2])] += 1;
22         local_vector[static_cast<size_t>(pos[3])] += 1;
23     }
24 }
```

Stejně jako v první iteraci se tato metoda přiděluje každému vláknku ve třídě `File_Mapping` pro zpracování konkrétního bloku dat. Příklad vektorizace je znázorněn na obrázku 3.6.

```

Disassembly  x smp_utils.cpp
Address: ppr::parallel::hsum_double_avx(_m256d)
Viewing Options
00007FF7B8E44767 sub     rsp,278h
00007FF7B8E4476E lea     rbp,[rsp+30h]
00007FF7B8E44773 lea     rcx,[_7FEF4607_smp_utils@cpp (07FF7B8E9C4DEh)]
00007FF7B8E4477A call    __CheckForDebuggerJustMyCode (07FF7B8DC78C2h)
        __m128d vlow = _mm256_castpd256_pd128(v);
00007FF7B8E4477F mov     rax,qword ptr [&v]
00007FF7B8E44786 vmovups  xmm0,xmmword ptr [rax]
00007FF7B8E4478A vmovupd  xmmword ptr [rbp+160h],xmm0
00007FF7B8E44792 vmovupd  xmm0,xmmword ptr [rbp+160h]
00007FF7B8E4479A vmovupd  xmmword ptr [vlow],xmm0
        __m128d vhigh = _mm256_extractf128_pd(v, 1);
00007FF7B8E4479F mov     rax,qword ptr [&v]
00007FF7B8E447A6 vmovupd  ymm0,ymmword ptr [rax]
00007FF7B8E447AA vextractf128 xmm0,ymm0,1
00007FF7B8E447B0 vmovupd  xmmword ptr [rbp+190h],xmm0
00007FF7B8E447B8 vmovupd  xmm0,xmmword ptr [rbp+190h]
00007FF7B8E447C0 vmovupd  xmmword ptr [vhigh],xmm0
        vlow = _mm_add_pd(vlow, vhigh);
00007FF7B8E447C5 vmovupd  xmm0,xmmword ptr [vlow]
00007FF7B8E447CA vaddpd   xmm0,xmm0,xmmword ptr [vhigh]
00007FF7B8E447CF vmovupd  xmmword ptr [rbp+1C0h],xmm0
00007FF7B8E447D7 vmovupd  xmm0,xmmword ptr [rbp+1C0h]
00007FF7B8E447DF vmovupd  xmmword ptr [vlow],xmm0
        __m128d high64 = _mm_unpackhi_pd(vlow, vlow);
00007FF7B8E447E4 vmovupd  xmm0,xmmword ptr [vlow]
00007FF7B8E447E9 vunpckhpd xmm0,xmm0,xmmword ptr [vlow]
00007FF7B8E447EE vmovupd  xmmword ptr [rbp+1F0h],xmm0
00007FF7B8E447F6 vmovupd  xmm0,xmmword ptr [rbp+1F0h]
00007FF7B8E447FE vmovupd  xmmword ptr [high64],xmm0
        return _mm_cvtsd_f64(_mm_add_sd(vlow, high64)); // reduce to scalar
00007FF7B8E44803 vmovupd  xmm0,xmmword ptr [vlow]

```

Obrázek 3.6: Ukázka vektorizace programu

3.3.3 Porovnání rychlostí běhu SMP

Ukázalo se, že algoritmus `tbb` je o trochu rychlejší, když je vstupní soubor malý. Při práci s velkými soubory je implementace s využitím manuální vektorizace rychlejší.

3.3.4 OpenCL kernel

Kernel z druhé iterace se prochází skrz všechny prvky ve work group a ukládá spočítanou hodnotu rozptylu, a také provádí atomické operace pro ukládání

frekvenčních hodnot histogramu. Celý kód pro kernel se nachází v souboru `histogram_kernel.cl`.

3.4 Histogram hustoty (Density histogram)

Vzhledem k tomu, že počet prvků v histogramu není vysoký, rozhodla jsem provést tuto operaci sekvenčně. Na základě frekvenčního histogramu se vytváří histogram hustoty, který se pak využívá v počítání RSS.

3.5 Počítání hodnot RSS

V této části programu jsem požila algoritmus `tbb::parallel_reduce`. Celkové počítání RSS pomocí této funkce nezabíralo víc než 0.1 sekundy. Stejně jako jsem popsala výše, nepodařilo se mi využít algoritmus TBB pro počítání spolehlivých výsledků. Proto jsem využila standartní funkci pro vytváření vláken `std::async` pro paralelní počítání RSS pro všechny distribuce (funkce `calculate_histogram_RSS_cpu` v souboru `smp_utils.cpp`). V souboru `rss.cpp` se nachází implementace počítání chyby mezi pozorovanou frekvencí dat a očekávanou frekvencí pro každé rozdělení.

3.6 Watchdog

Watchdog je vlákno, které v nekonečné smyčce kontroluje správnost počítaných dat podle implementovaných pravidel. Jestli najde nesouvislost, ukončí program a upozorní na to uživatele. Celá implementace Watchdogu se nachází v souboru `watchdog.cpp`.

4 Uživatelská dokumentace

4.1 Ovládání

Pro spuštění programu musí být zadané povinné argumenty:

- Cesta k souboru s daty
- Režim programu (smp, all, seq)
- Názvy OpenCL zařízení

Mimo povinných vstupních argumentů programu jsem přidala následující nepovinné argumenty:

- **-o** argument musí být typu boolean. Nastavuje, zda se program spustí s provedenou optimalizací. Defaultní hodnota je nastavena na TRUE, tedy s běh s optimalizací.
- **-w** argument je celočíselného typu a ovlivňuje dobu v sekundách na jak dlouho se musí Watchdog uspat.

Příklad validního vstupu:

```
1 "C:\gauss" all "NVIDIA GeForce MX150" "Inter(R) UHD Graphics  
620" -o 0 -w 5
```

4.2 Výstup

Po zpracování souboru program vždycky vypíše následující informaci:

- Vstupní parametry, se kterými program byl spuštěn
- Spočítané statistiky
- Výsledky RSS
- Čas běhu
- Výsledné rozdělení s parametry spočítané na základě nejnižší hodnoty RSS

Příklad výstupu:

```
1                                     [Initial parameters]
2 -----
3 > Mode:                               smp
4 > Number of threads:                 8
5 > Optimalization:                   TRUE
6 > Watchdog timer:                   2 sec
7
8
9 > Started ..
10
11                                    [Statistics]
12 -----
13 > n:                                137524224
14 > sum:                              1.37528e+08
15 > mean:                             1.00003
16 > variance:                         1.00015
17 > min:                              0
18 > max:                              20.9085
19 > isNegative:                       0
20 > isInteger:                        0
21
22
23                                    [Results]
24 -----
25 > Gauss RSS:                        0.0837409
26 > Poisson RSS:                      0.160802
27 > Exponential RSS:                  0.126511
28 > Uniform RSS:                      0.538024
29
30
31
32                                    [Time]
33 -----
34 > Statistics computing time:         0.563849 sec.
35 > Histogram computing time:         1.25544 sec.
36 > RSS computing time:                0.0007262 sec.
37 > TOTAL TIME:                       1.82843 sec.
38
39 > Input data have 'Exponential distribution' with lambda
    =0.999971.00015
```

5 Analýza výsledků

Z přiložených tabulek 5.1 a 5.2 lze pozorovat, že první běh programu trvá vždy více času než každý následující běh. Kvůli prvnímu běhu se tedy střední hodnota času zhoršuje. Souvisí to s tím, že požadované stránky s mapovaným souborem v prvním běhu nejsou v rychlé paměti RAM. Jakmile se ovšem do paměti RAM načtou, program se výrazně zrychlí. Algoritmus mého programu prochází vstupní soubor dvakrát. Takže časy první a druhé iterace by měli být skoro totožné. Ale během prvního průchodu se stránky se souborem přemístí do RAM paměti a druhá iterace potom běží mnohem rychleji, jak lze pozorovat na tabulce 5.3.

#	Sekvenční výpočet	Paralelní - TBB	Paralelní - Vektorizace	Paralelní - ALL
1.	22.2017	28.9536	27.4167	48.2253
2.	11.2105	5.50204	4.27749	47.9483
3.	11.2373	4.37373	4.13617	53.8338
4.	11.2793	4.45375	4.1926	55.5232
5.	11.2817	4.6461	4.19529	55.5233
Průměr	13.4421 sek.	9.585844 sek.	8.84365 sek.	52.21078 sek.

Tabulka 5.1: Příklady času (sekundy) běhu programu v různých režimech se souborem velikosti 7 GB ležícím na SSD disku

#	Sekvenční výpočet	Paralelní - TBB	Paralelní - Vektorizace	Paralelní - ALL
1.	95.1375	539.369	358.862	183.742
2.	88.2781	8.34883	7.49745	47.122
3.	34.0248	8.05892	7.80384	48.4697
4.	33.9824	8.2241	7.7205	47.3133
5.	33.8996	8.29998	7.60831	52.4419
Průměr	57.06448 sek.	114.460166 sek.	77.89842 sek.	75.81778 sek.

Tabulka 5.2: Příklady času (sekundy) běhu programu v různých režimech se souborem velikosti 7 GB ležícím na HDD disku

Další pozorovanou věcí je fakt, že se časy zpracování souborů stejné velikosti liší v závislosti na typu pevného disku (SSD a HDD), kde jsou data uložena. Důvodem rychlost čtení, kdy se stránky se souborem přenášejí pomaleji do RAM z disku HDD.

#	Total	Statistics	Histogram	RSS
1.	27.4167	22.8254	4.49203	0.0014339
2.	4.27749	2.3742	1.90303	0.0001261
3.	4.13617	2.30001	1.8358	0.0001384
4.	4.1926	2.36215	1.83021	0.0001122
5.	4.19529	2.37862	1.81576	0.0001845
Průměr	8.84365 sek.	6.448076 sek.	2.375366 sek.	0.00039902 sek.

Tabulka 5.3: Příklad rozdílu času (sekundy) běhu každé iterace programu v režimu SMP se souborem velikosti 7 GB ležícím na SSD disku.

Po porovnání středních hodnot času výpočtu (tabulka 5.4 a 5.5) se zjistilo, že se mi podařilo dosáhnout maximálně 52% zrychlení oproti sekvenčnímu běhu, za předpokladu, že vstupní soubor se nacházel na SSD disku. Sekvenční program ve střední hodnotě času se ukázal jako rychlejší řešení pro zpracování souboru z HDD disku. Ale také se dá všimnout, že jakmile se soubor dostane do paměti RAM, paralelní implementace mého programu je mnohem efektivnější (tabulka 5.2).

Parallel - ALL	-74%
Parallel - TBB	40%
Parallel - SMP (opt)	52%

Tabulka 5.4: Porovnání rychlostí jednotlivých režimů programu oproti sekvenčnímu režimu (velikost souboru 7 GB, SSD disk)

Parallel - ALL	-25%
Parallel - TBB	-50%
Parallel - SMP (opt)	-27%

Tabulka 5.5: Porovnání rychlostí jednotlivých režimů programu oproti sekvenčnímu režimu (velikost souboru 7 GB, HDD disk)

Nepodařilo se mi dosáhnout rychlejšího výpočtu paralelizace s využitím OpenCL zařízení. Kopírování dat do zařízení zabírá moc času a snahy o paralelizaci selhaly, kvůli „race conditional“.

Nejrychlejším režimem programu se ukázal režim SMP s využitím manuální vektorizace. Všechny porovnání rychlostí výpočtů jsou dostupné v Excel tabulce ve složce s touto dokumentací.

6 Závěr

Semestrální práce se nepodařilo zcela dokončit. Program nevrací správné výsledky spočítaného RSS. Chyba je ve výpočtu RSS u normálního a rovnoměrného rozdělení. Exponenciální a rovnoměrné rozdělení se umí poznat.

Program byl testován na operačním systému Windows 11 s procesorem Intel Core i7 a grafickou kartou NVIDIA GeForce MX150.

Nejvíce času jsem strávila vymýšlením algoritmu na detekci správného rozdělení. Další výzvou bylo porozumět, proč se nedaří využít plného vytížení CPU.

Tato práce mi vytvořila praktický přehled o tom, jak fungují vlákna a stránky paměti. Největší zpomalení programu dělají chybějící stránky v paměti RAM, dále také kopírování dat do grafické karty. Rychlost programu také ovlivňuje, na jakém typu disku data leží.

Kód této práce lze najít v Git repositáři na této stránce <https://bitbucket.org/pwnsauce8/kiv-ppr/src>.

Literatura

- [DOU22] DOURNAC.ORG. *Parallel Sum Reduction*. 2022. Available at https://dournac.org/info/gpu_sum_reduction.
- [Fre22] Freepascal. *Freepascal*. 2022. Available at <https://www.freepascal.org/docs-html/prog/progsu171.html>.
- [MT21a] PhD. Marco Taboga. *Exponential distribution - Maximum Likelihood Estimation*. 2021. Available at <https://www.statlect.com/fundamentals-of-statistics/exponential-distribution-maximum-likelihood>.
- [MT21b] PhD. Marco Taboga. *Normal distribution - Maximum Likelihood Estimation*. 2021. Available at <https://www.statlect.com/fundamentals-of-statistics/normal-distribution-maximum-likelihood>.
- [MT21c] PhD. Marco Taboga. *Poisson distribution - Maximum Likelihood Estimation*. 2021. Available at <https://www.statlect.com/fundamentals-of-statistics/Poisson-distribution-maximum-likelihood>.
- [Wik21] Wikipedia. *Microsoft*. 2021. Available at <https://learn.microsoft.com/en-us/windows/win32/memory/creating-a-view-within-a-file>.
- [wik22a] wikipedia. *Kolmogorov–Smirnov test*. 2022. Available at https://en.wikipedia.org/wiki/Residual_sum_of_squares.