Klasifikace dat do pravděpodobnostního rozdělení

KIV/PPR - semestrální práce

Mukanova Zhanel
A22N0130P

11. prosince 2022

Obsah

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 vypočtu vypsat hodnoty charakterizující rozdělení a zdůvodnění svého výsledku. Také zadání definuje následující limity, které musí být dodržené:

- 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
- n**ázvy 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 k řešení semestrální práce použiji 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. Tento bod zadání by nebyl splněn:

> *Program vypíše hodnoty charakterizující rozdělení a zdůvodnění svého výsledku.*

Navíc, algoritmus „Kolmogorovův–Smirnovův test" vyžaduje 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 průchodu 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 frekvenci 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 Normální rozdělení

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

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

$$SD = \sqrt{\sigma^2}$$

Kde μ odpovídá střední hodnotě, σ^2 rozptylu a SD směrodatné odchylce [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 definice rozdělení [MT21a].

Střední hodnotu ($Mean$) a směrodatnou odchylku (SD) pro exponenciální rozdělení se počítá následujicím způsobem:

$$Mean = min(X_1, X_2, \cdots, X_n)$$

$$SD = \frac{1}{n}\sum_{i=1}^{n} x_i - min(X_1 + X_2 + \cdots + X_n)$$

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

$$a = Mean = min(X_1, X_2, \cdots, X_n)$$

$$b = max(X_1, X_2, \cdots, X_n)$$

$$SD = b - a$$

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

2.4 Hustota pravděpodobnosti (Probability Density Function, PDF)

Hustota pravděpodobnosti (Probability Density Function, PDF) je v teorii pravděpodobnosti funkce, jejíž integrací na kterémkoli vzorku vyjde relativní pravděpodobnost. Je-li $\rho(x)$ hustota pravděpodobnosti spojité náhodné veličiny X, pak platí:

$$\int_{\Omega} \rho(x)\mathrm{d}x = 1$$

kde Ω je definiční obor veličiny X. Pro hodnoty x mimo definiční obor Ω je hustota pravděpodobnosti nulová, takže $\rho(x) = 0$ pro $x \notin \Omega$.

Dále budou ukázany funkce hustoty pravděpodobností pro každé rozdělení, který byly vuyžity v této práci.

2.4.1 Hustota pravděpodobnosti pro normální rozdělení

$$f(x) = \frac{e^{-X^2/2}}{\sqrt{2\pi}}$$

2.4.2 Hustota pravděpodobnosti pro exponenciální rozdělení

$$f(x) = e^{-X}$$

2.4.3 Hustota pravděpodobnosti pro rovnoměrné rozdělení

$$f(x) = 1$$

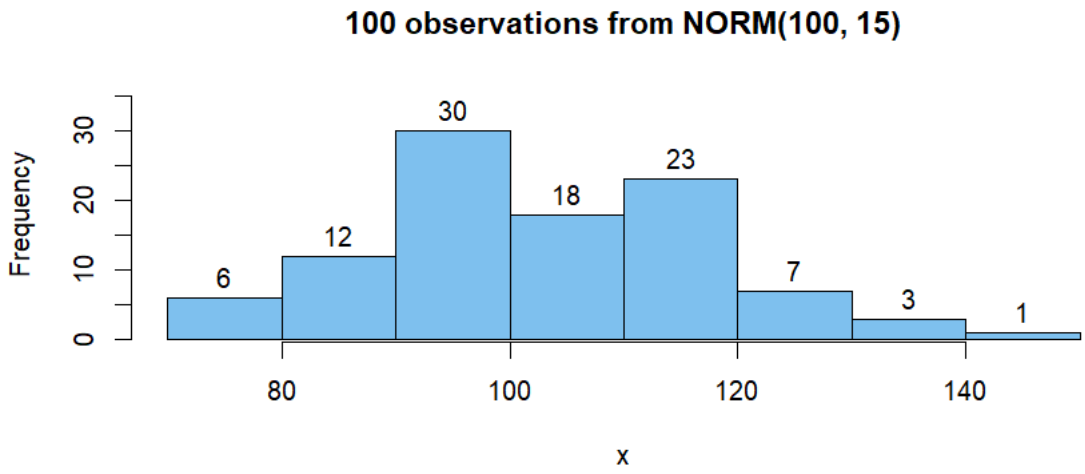
2.4.4 Hustota pravděpodobnosti pro poissonovo rozdělení

$$f(x) = \frac{e^{-\mu}\mu^X}{X!},$$

kde μ je střední hodnota Poissonovo rozdělení a X je náhodna hodnota.

2.5 Frekvenční histogram (frequency histogram)

Frekvenční histogram je grafické znázornění distribuce dat pomocí sloupcového grafu se sloupci stejné šířky, vyjadřující šířku intervalů (tříd), přičemž výška **sloupců vyjadřuje četnost** sledované veličiny v daném intervalu. Je důležité zvolit správnou šířku intervalu, neboť nesprávná šířka intervalu může snížit informační hodnotu diagramu.



Obrázek 2.1: Ukázka frekvenčního histogramu

V této práci využívám konstantní šířku histogramu, kterou pro spojitá rozdělení počítam pomocí:

$$binCount = log(n) + 2$$

$$w = (max - min)/binCount$$

a pro diskretní (poissonovo rozdělení):

$$binCount = max - min$$

$$w = 1,$$

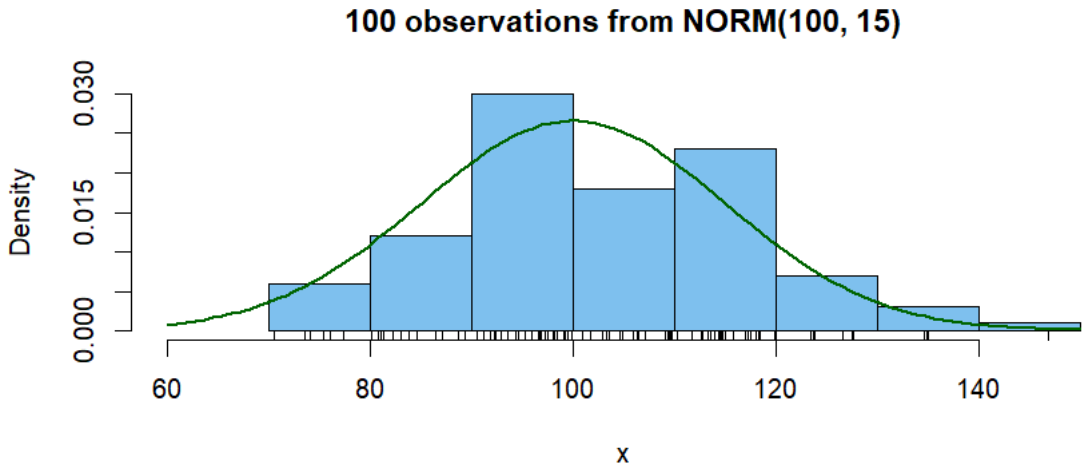
kde $binCount$ je počet sloupců v histogramu a w je šířka sloupce.

2.6 Histogram hustoty (density histogram)

Histogram hustoty se liší od frekvenčního tím, že jeho sloupce ukazují jednotky, díky nimž se suma všech hodnot sloupců je ekvivalentní 1 [Bru18]. To umožňuje porovnávat hodnoty s funkci hustoty pravděpodobnosti (PDF) rozdělení, jejíž integral se také roven 1.

Pokud f_i je frekvence i-tého sloupce, pak jeho relativní frekvence je $r_i = f_i/n$, kde n je velikost pozorovaných dat. Jeho hustota je potom $d_i = r_i/w_i$, kde w_i je jeho šířka. Vysledná formula pro převod frekvenčního histogramu na histogram hustoty je:

$$d_i = f_i/w_i/n$$



Obrázek 2.2: Ukázka histogramu hustoty

2.7 Residual sum of squares (RSS)

Ve statistice je **Residual sum of squares (RSS)** [Wik22b] součtem čtverců rozdílů mezi daty a modelem odhadu, v našem případě jde o předpokládané rozdělení. Čím je menší hodnota RSS, tím je menší rozdíl mezi pozorovanými a očekávanými daty. Tato hodnota se používá jako optimální kritérium při výběru parametrů a modelu (rozdělení pravděpodobnosti).

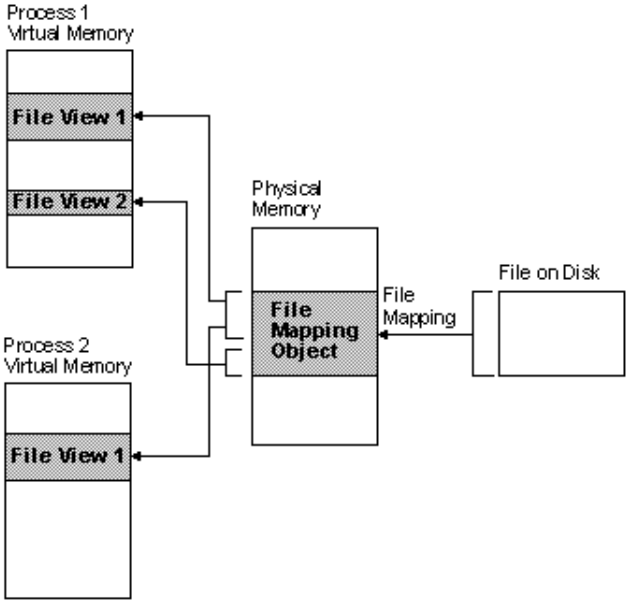
$$RSS = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Kde za y_i dosazují hodnoty d_i (viz sekci 2.6) a za $f(x_i)$ funkci hustoty pravděpodobnosti (viz sekci 2.4).

3 Implementace

3.1 File Mapping

S doporučením od pana docenta jsem rozhodla ve své práci využít mapování souboru [Mic21] do adresního prostoru procesu místo klasického čtení ze souboru (viz obrázek 3.1). 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.

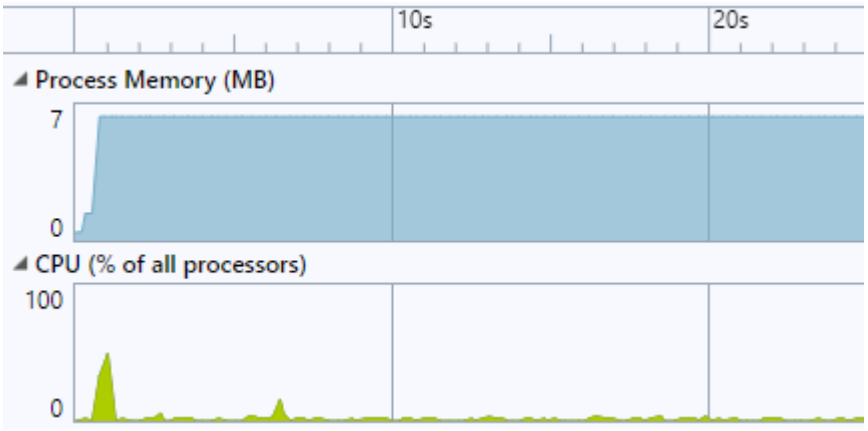


Obrázek 3.1: Ukázka mapování souboru do adresního prostoru procesu

Celé řešení se nachází ve třídě `File_mapping`. Tato třída mapuje soubor do paměti a rozděluje ho mezi výpočetními vlákny.

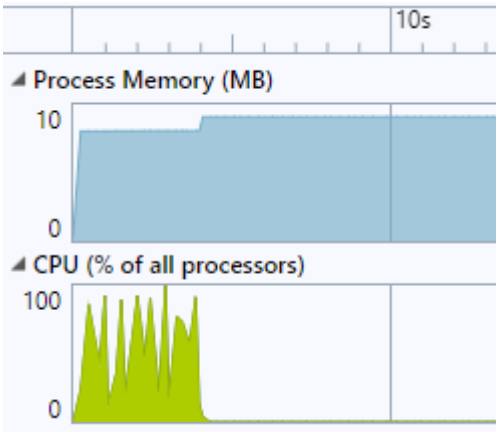
3.1.1 První řešení

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 vlákny. Takové řešení mělo několik nevýhod. První bylo, že soubor ležící na disku typu HDD se načítal hodně pomalu (viz obrázek 3.2).



Obrázek 3.2: Ukázka využití CPU při prvním běhu. Zpracování souboru o 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 3.3), nicméně podstatou programu není mnohonásobný běh.



Obrázek 3.3: Ukázka využití CPU při druhém běhu. Zpracování souboru o 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 optimalizaci

Z příkladu výše vyplývá, že doba trvání 10-ti 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" (viz sekci 3.1.3) pro vypočet mapovaného bloku. Pokud doba zpracování přesahuje dobu 5 sekund (stanoveno pomocí pokusného běhu), velikost bloku se změnší tak, aby vlákno nemuselo čekat přiliš dlouho na přesun stránky z disku na RAM. Místo cca 2 GB mapovacích bloků budou mapovány bloky velikosti odpovídající "allocation granularity". Takový způsob prokázal většího využití CPU. Navíc takto malé bloky by stačilo načítat pouze v prvním průchodu souborem. Druhý průchod by byl rychlejší, protože všechny stránky už by byly 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

V této sekci je popsané výsledné řešení mapování souboru a přidělování ho vláknům. 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.

Algoritmus mapování bloků pro dvě iterace je stejný. Liší se pouze funkcí, která se přiděluje vláknům pro zpracování souboru (viz obrázek 3.4). Třida `File_mapping` obsahuje celkem dvě řidící metody. První je `read_in_one_chunk_cpu` pro výpočet iterace s AVX2 instrukcí a druhá metoda `read_in_chunks_tbb` pro výpočet pomocí algoritmů TBB.

Režim SMP

Ze zadaní 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 vlákny ke zpracování. Příklad algoritmu pro režim SMP je ukázan na obrázku 3.6.

Pro výpočet s využitím AVX2 instrukci (metoda `read_in_one_chunk_cpu()`) se vlákna vytvářejí pomocí `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 vypočtu. Výsledky se na konci vypočtu vláken sjednocují.

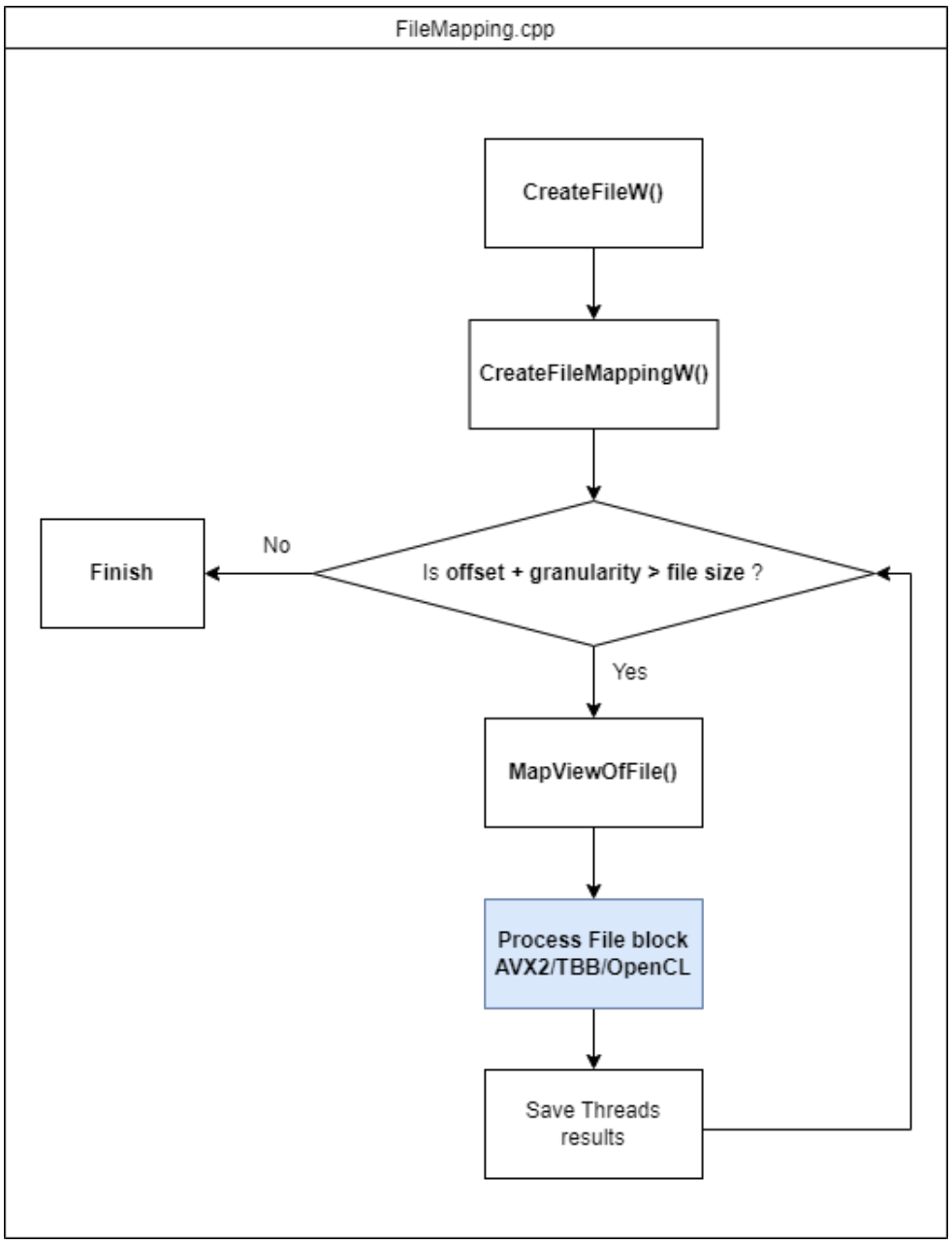
Pro výpočet s využitím algoritmů TBB (metoda `read_in_chunks_tbb`), se vlákná vytvářejí samotnou knihovnou.

Režim ALL

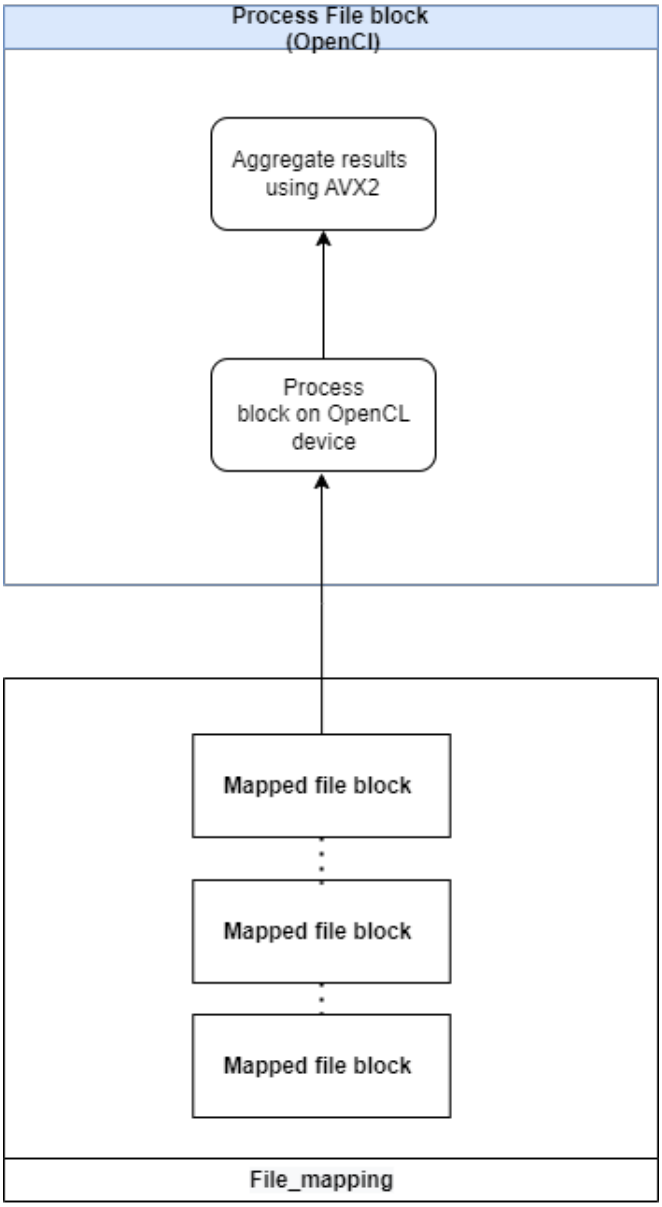
V případě režimu s využitím OpenCL zařízení je potřeba vytvářet buffer s daty, aby se mohl poslat na zařízení. Velikost jednoho bufferu nesmí přesahovat 1 GB podle zadání. Abych nemusela mapovat celý soubor do paměti a mohly se využit výhody mapování menších částí, musela jsem správně nastavit offsety. Z definici od Microsoftu offsety musí být násobkem **"allocation granularity"** systému, což odpovídá 2^{16} (nebo 2^{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 mapování souboru pro režim ALL je ukázan na obrázku 3.5 .

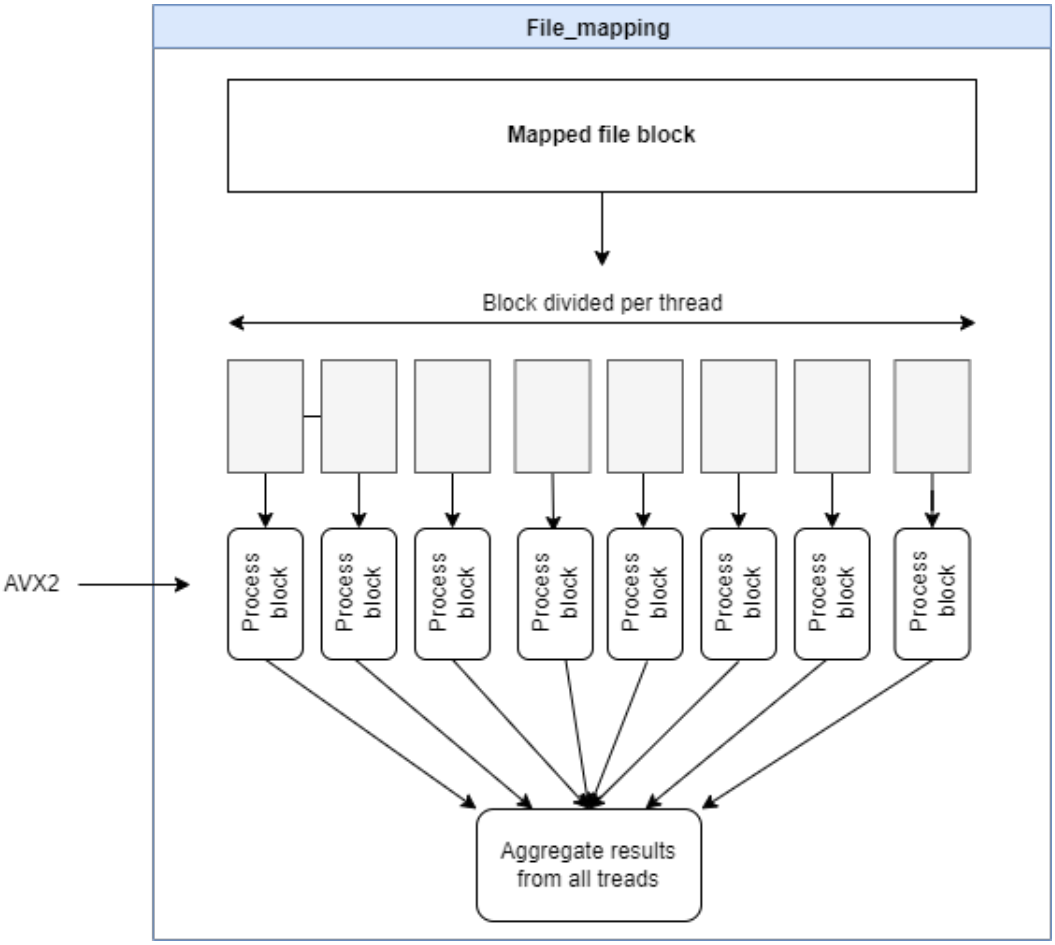
Případné dělení výpočtů bloků mezi procesy s využitím OpenCL se totiž ukázalo jako nespolehlivá úloha, která vracela různé výsledky kvůli „race condition".



Obrázek 3.4: Ukázka algoritmu mapování souboru pro režim režimy



Obrázek 3.5: Ukázka zpracování souboru pomocí OpenCL zařízení v režimu ALL



Obrázek 3.6: Ukázka zpracování souboru pomocí vláken v režimu SMP

3.2 Vypočet statistik (První iterace)

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

```
struct SDataStat
{
    unsigned int n = 0;
    double sum = 0.0;
    double max = 0.0;
    double min = 0.0;
    double mean = 0.0;
    double variance = 0.0;
    bool isNegative = 0;
};
```

- **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 Režim SMP

První moje řešení paralelního výpočtu bylo s použitím knihovny TBB od společností 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 Režim 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 vláknu ve třídě `File_mapping`, která také rozhoduje kolik dat každému vláknu přidělí. Každé vlákno pak pracuje se svojí částí bloku.

```
void get_statistics_vectorized(/* ... */)
{
    __m256d min = _mm256_set1_pd(
        std::numeric_limits<double>::max()
    );

    __m256d max = _mm256_set1_pd(
        std::numeric_limits<double>::lowest()
    );

    for (int block = 0; block < data_count; block += 4)
    {
        // Count elements
        stat.n += 4;

        // Find sum of 4 vector element
        __m256d vec = _mm256_load_pd(data + block);
        stat.sum += hsum_double_avx(vec);

        // Find Max and Min of 4 vector element
        max = _mm256_max_pd(max, vec);
        min = _mm256_min_pd(min, vec);
    }

    // Agregate results
    /* ... */
}
```

3.2.3 OpenCL

Jak jsem popsala v sekci s implementací `File_mapping`, OpenCL 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 vypočet. Zkoušela jsem vy-

tvářet vlákna, která by zpracovávaly vlastní blok dat (podobně jako SMP) a zapisovaly vlastní buffery do sdílené fronty `cl::CommandQueue`. Výsledky vypočtu 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 velikosti počtů „work group“. To u velkého počtu dat může znamenat velkou hodnotu. Proto na agregaci výsledků používám funkci s využitím AVX2 instrukcí ze souboru `smp_utils.cpp`.

3.2.4 OpenCL kernel pro sběr statistik

Rozhodla jsem využit algoritmus **Parallel Sum Reduction**, který je popsaný 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 vypočtu RSS. Cílem je sestavit „Density histogram“ (viz sekci 2.6). Stejně jako v první iteraci se bloky rozdávají vláknům ve třídě `File_Mapping`.

3.3.1 Režim SMP

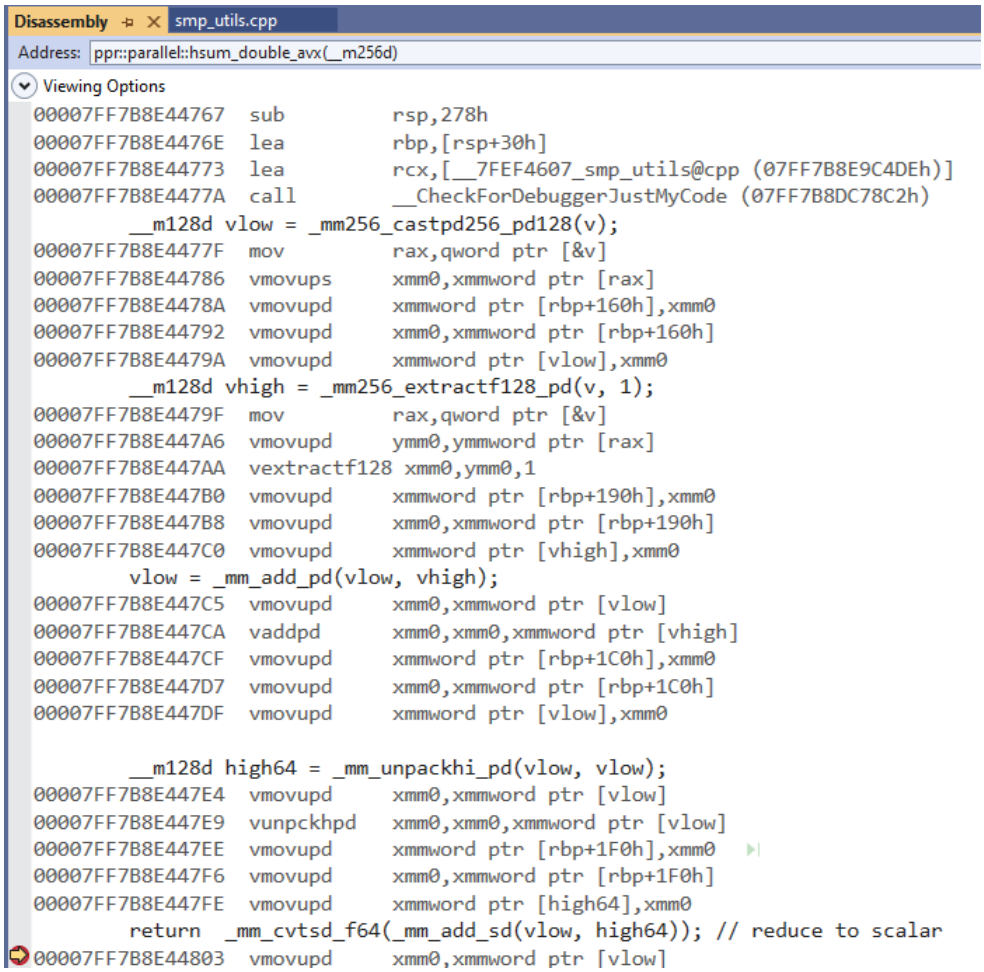
Jako první pokus o řešení problému jsem začala pracovat s algoritmem `tbb::parallel_reduce()`, který stejně jako v první iteraci vytvářel nespolehlivé výsledky. Implementaci lze nalézt ve třídě `Histogram_parallel` v souboru `histogram.cpp`. Rozhodla jsem provést optimalizaci, abych dokázala použit výhody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

3.3.2 Režim 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,
       scale);
16         double* pos = (double*)&position;
17
18         local_vector[static_cast<size_t>(pos[0])] += 1;
19         local_vector[static_cast<size_t>(pos[1])] += 1;
20         local_vector[static_cast<size_t>(pos[2])] += 1;
21         local_vector[static_cast<size_t>(pos[3])] += 1;
22     }
23 }
```

Stejně jako v první iteraci se tato metoda přiděluje každému vláknu ve třídě `File_Mapping` pro zpracování konkrétního bloku dat. Příklad vektorizace je znázorněn na obrázku 3.7.



Obrázek 3.7: Ukázka vektorizace progamu

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. Podrobnější porovnání bude ukázano v sekci 5.

3.3.4 OpenCL kernel

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

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 (viz sekci 2.6), který se pak využívá v počítaní RSS.

3.5 Počítaní hodnot RSS

V této části programu jsem požila algoritmus `tbb::parallel_reduce`. Celkové počítaní RSS pomocí této funkce nezabíralo víc než 0.1 sekundy. Stejně jako jsem popsala výše, nepodařilo se mi využit 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 počítaní RSS a také implementace funkci PDF pro každé rozdělení (viz sekci 2.4).

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** je argument celočíselného typu a ovlivňuje dobu uspání Watchdogu, zadáno v sekundách.

Příklad validního vstupu:

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

4.2 Výstup

Po zpracování souboru program vždy vypíše následující informace:

- 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 programu v režimu SMP se vstupním souborem (7 GB) obsahující data s **normálním rozdělením**:

```
                        [Initial parameters]
--------------------------------------------------
> File:                           C:\delete\gauss
> Mode:                           smp
> Number of threads:              8
> Optimalization:                 TRUE
> Watchdog timer:                 2 sec


> Started ..

                        [Statistics]
--------------------------------------------------
> n:                              1000000000
> sum:                            -3572.2
> mean:                           -3.5722e-06
> variance:                       1.00005
> min:                            -6.10922
> max:                            5.87261
> isNegative:                     1
> isInteger:                      0


                        [Results]
--------------------------------------------------
> Gauss RSS:                      0.00079459
> Poisson RSS:                    -nan(ind)
> Exponential RSS:                0.588353
> Uniform RSS:                    0.477286



                        [Time]
--------------------------------------------------
> Statistics computing time:      3.02456 sec.
> Histogram computing time:       4.37617 sec.
> RSS computing time:             0.0003251 sec.
> TOTAL TIME:                     7.40135 sec.

> Input data have 'Gauss/Normal distribution' with mean
   =-3.5722e-06 and variance=1.00005
```

Příklad výstupu programu v režimu SMP se vstupním souborem (1 GB) obsahující data s **exponenciálním rozdělením**:

```
                        [Initial parameters]
-------------------------------------------------
> File:                           C:\delete\exp
> Mode:                           smp
> Number of threads:              8
> Optimalization:                 TRUE
> Watchdog timer:                 2 sec


> Started ..

                        [Statistics]
-------------------------------------------------
> n:                              137524224
> sum:                            1.37528e+08
> mean:                           1.00003
> variance:                       1.00015
> min:                            0
> max:                            20.9085
> isNegative:                     0
> isInteger:                      0


                        [Results]
-------------------------------------------------
> Gauss RSS:                      0.24959
> Poisson RSS:                    0.17973
> Exponential RSS:                0.121527
> Uniform RSS:                    0.558112


                        [Time]
-------------------------------------------------
> Statistics computing time:      4.54918 sec.
> Histogram computing time:       0.803888 sec.
> RSS computing time:             0.0002501 sec.
> TOTAL TIME:                     5.35362 sec.

> Input data have 'Exponential distribution' with lambda
   =0.99997
```

Příklad výstupu programu v režimu SMP se vstupním souborem (1 GB) obsahující data s **poissonovým rozdělením**:

```
                        [Initial parameters]
-------------------------------------------------
> File:                           C:\delete\poisson
> Mode:                           smp
> Number of threads:              8
> Optimalization:                 TRUE
> Watchdog timer:                 2 sec


> Started ..

                        [Statistics]
-------------------------------------------------
> n:                              141318656
> sum:                            1.41323e+08
> mean:                           1.00003
> variance:                       0.999858
> min:                            0
> max:                            10
> isNegative:                     0
> isInteger:                      1


                        [Results]
-------------------------------------------------
> Gauss RSS:                      0.0203551
> Poisson RSS:                    2.58775e-09
> Exponential RSS:                0.402114
> Uniform RSS:                    0.208506


                        [Time]
-------------------------------------------------
> Statistics computing time:      3.32517 sec.
> Histogram computing time:       0.688783 sec.
> RSS computing time:             0.0001357 sec.
> TOTAL TIME:                     4.01441 sec.

> Input data have 'Poisson distribution' with lambda=1.00003
```

Příklad výstupu programu v režimu SMP se vstupním souborem (1 GB) obsahující data s **rovnoměrným rozdělením**:

```
                        [Initial parameters]
-------------------------------------------------
> File:                           C:\delete\uniform
> Mode:                           smp
> Number of threads:              8
> Optimalization:                 TRUE
> Watchdog timer:                 2 sec


> Started ..

                        [Statistics]
-------------------------------------------------
> n:                              133684224
> sum:                            6.68395e+07
> mean:                           0.49998
> variance:                       0.0833297
> min:                            0
> max:                            1
> isNegative:                     0
> isInteger:                      0


                        [Results]
-------------------------------------------------
> Gauss RSS:                      3.82184
> Poisson RSS:                    7.97942
> Exponential RSS:                7.39103
> Uniform RSS:                    0.0357194


                        [Time]
-------------------------------------------------
> Statistics computing time:      4.0244 sec.
> Histogram computing time:       0.654257 sec.
> RSS computing time:             0.0001364 sec.
> TOTAL TIME:                     4.67911 sec.

> Input data have 'Uniform distribution' with a=0 and b=1
```

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 pamětí RAM. Jakmile se ovšem do paměti RAM načtou, program se výrazně zrychlí.

#	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

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. Zde je vidět, že první iterace se sběrem statistik trvá 22 sekundy a druhá iterace, kde se sestavuje histogram, zabere pouze 4 sekundy.

#	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.

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 typu HDD.

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 můžeme 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 rychostí 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 rychostí jednotlivých režimů programu oproti sekvenčnímu režimu (velikost souboru 7 GB, HDD disk)

6 Závěr

Zadání semestrální práce bylo splněno. Program vrací správný odhad pravděpodobnosti pro vstupní data na základě spočitané hodnoty RSS.

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. Rychlost programu v režimu SMP také ovlivňuje, na jakém typu disku data leží.

Nepodařilo se mi dosáhnout rychlejšího vypočtu 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. Všechna jednotlivá data a jejich porovnání, včetně porovnání rychlostí výpočtu, jsou dostupné v Excel tabulce ve složce s touto dokumentací.

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

Literatura

[Bru18] BruceET. *Difference between Frequency and Density in a Histogram.* 2018. Available at `https://math.stackexchange.com/questions/2666834/what-is-the-difference-between-frequency-and-density-in-a-histogram`.

[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`.

[Mic21] Microsoft. *File Mapping.* 2021. Available at `https://learn.microsoft.com/en-us/windows/win32/memory/creating-a-view-within-a-file`.

[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`.

[wik22a] wikipedia. *Kolmogorov–Smirnov$_t$est.* 2022. *Availableat*.

Wikipedia. *Residual sum of squares.* 2022. Available at `https://en.wikipedia.org/wiki/Residual_sum_of_squares`.