

# Klasifikace dat do pravděpodobnostního rozdělení

KIV/PPR - semestrální práce

Mukanova Zhanel  
A22N0130P

11. ledna 2023

# Obsah

<b>1</b>	<b>Zadání</b>	<b>5</b>
<b>2</b>	<b>Analýza</b>	<b>6</b>
2.1	Původní nápad . . . . .	6
2.2	Realizace . . . . .	6
2.3	Maximum likelihood estimation . . . . .	7
2.3.1	Normální rozdělení . . . . .	7
2.3.2	Exponenciální rozdělení . . . . .	7
2.3.3	Poissonovo rozdělení . . . . .	8
2.3.4	Rovnoměrné rozdělení . . . . .	8
2.4	Hustota pravděpodobnosti (Probability Density Function, PDF)	9
2.4.1	Hustota pravděpodobnosti pro normální rozdělení . .	9
2.4.2	Hustota pravděpodobnosti pro exponenciální rozdělení	9
2.4.3	Hustota pravděpodobnosti pro rovnoměrné rozdělení	9
2.4.4	Hustota pravděpodobnosti pro poissonovo rozdělení .	9
2.5	Frekvenční histogram (frequency histogram) . . . . .	10
2.6	Histogram hustoty (density histogram) . . . . .	11
2.7	Residual sum of squares (RSS) . . . . .	12
<b>3</b>	<b>Implementace</b>	<b>13</b>
3.1	File Mapping . . . . .	13
3.1.1	První řešení . . . . .	13
3.1.2	První pokus optimalizaci . . . . .	15
3.1.3	Výsledné řešení . . . . .	15
3.2	Vypočet statistik (První iterace) . . . . .	20
3.2.1	Režim SMP . . . . .	20
3.2.2	Režim SMP (optimalizace + vektorizace) . . . . .	21
3.2.3	OpenCL kernel pro sběr statistik . . . . .	21
3.3	Sestavení histogramu (Druhá iterace) . . . . .	22
3.3.1	Režim SMP . . . . .	22
3.3.2	Režim SMP (optimalizace + vektorizace) . . . . .	22
3.3.3	Porovnání rychlostí běhu SMP . . . . .	23
3.3.4	OpenCL kernel . . . . .	23
3.4	Histogram hustoty (Density histogram) . . . . .	24
3.5	Počítání hodnot RSS . . . . .	24
3.6	Watchdog . . . . .	24

<b>4</b>	<b>Uživatelská dokumentace</b>	<b>25</b>
4.1	Ovládání . . . . .	25
4.2	Výstup . . . . .	25
<b>5</b>	<b>Analýza výsledků</b>	<b>31</b>
<b>6</b>	<b>Závěr</b>	<b>37</b>
	<b>Literatura</b>	<b>38</b>

# 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 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 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  $\Theta$  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  $\mu$  odpovídá střední hodnotě,  $\sigma^2$  rozptylu a  $SD$  směrodatné odchylce [MT21b].

### 2.3.2 Exponenciální rozdělení

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

Kde lambda  $\lambda$  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 následujícím způsobem:

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

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

### 2.3.3 Poissonovo rozdělení

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

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

### 2.3.4 Rovnoměrné rozdělení

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

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

$$SD = b - a$$

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 spojitě náhodné veličiny  $X$ , pak platí:

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

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

Dále budou ukázány funkce hustoty pravděpodobností pro každé rozdělení, který byly využity 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

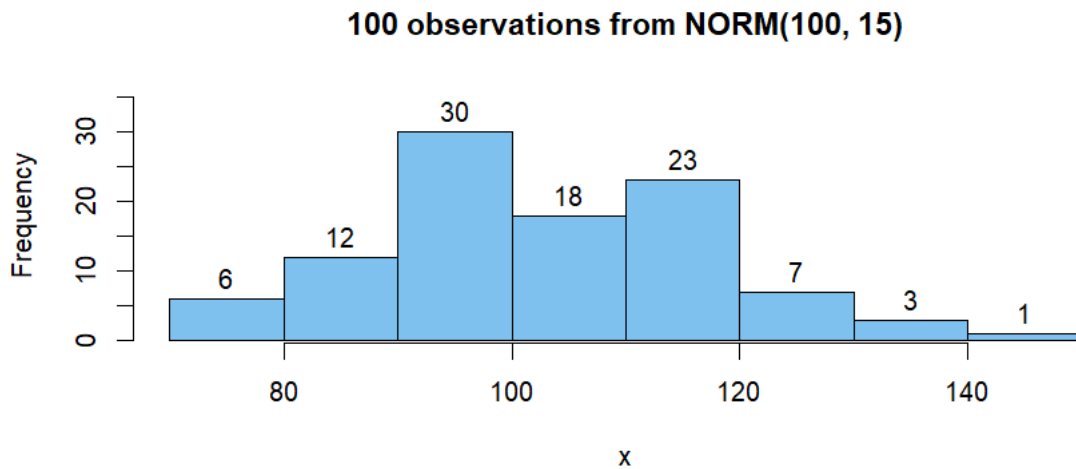
$$f(x) = \frac{e^{-\mu} \mu^x}{x!},$$

kde  $\mu$  je střední hodnota Poissonovo rozdělení a  $X$  je náhodná 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 počítám pomocí:

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

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

a pro diskretní (poissonovo rozdělení):

$$binCount = max - min$$

$$w = 1,$$

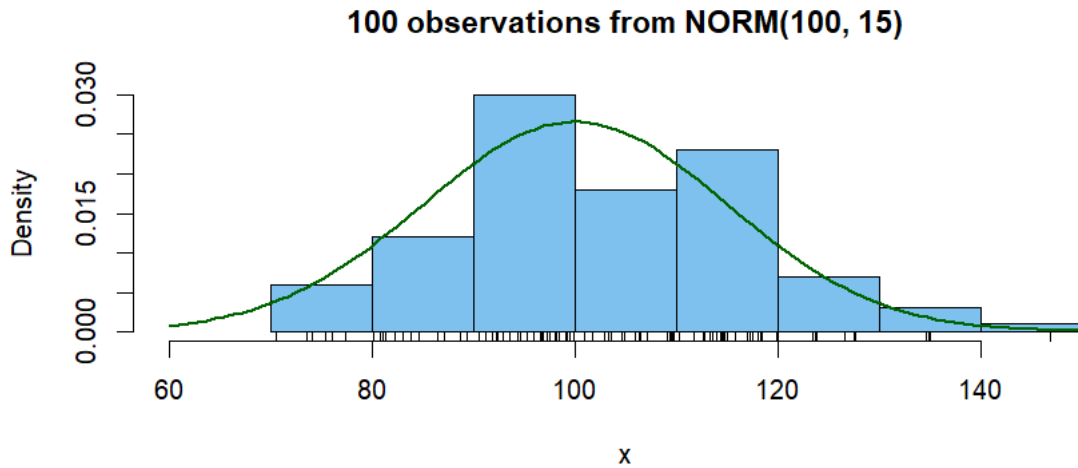
kde  $binCount$  je počet sloupců v histogramu a  $w$  je šířka sloupce.

## 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 funkcí hustoty pravděpodobnosti (PDF) rozdělení, jejíž integrál 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).

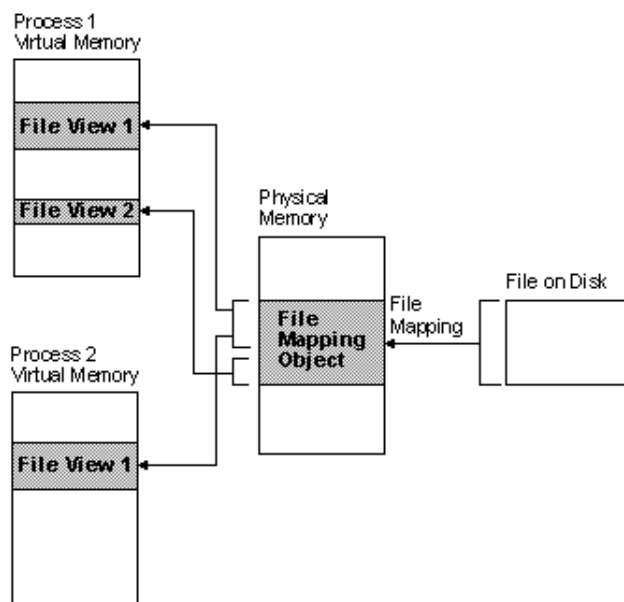
$$RSS = \sum_{i=1}^n (y_i - f(x_i))^2$$

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.

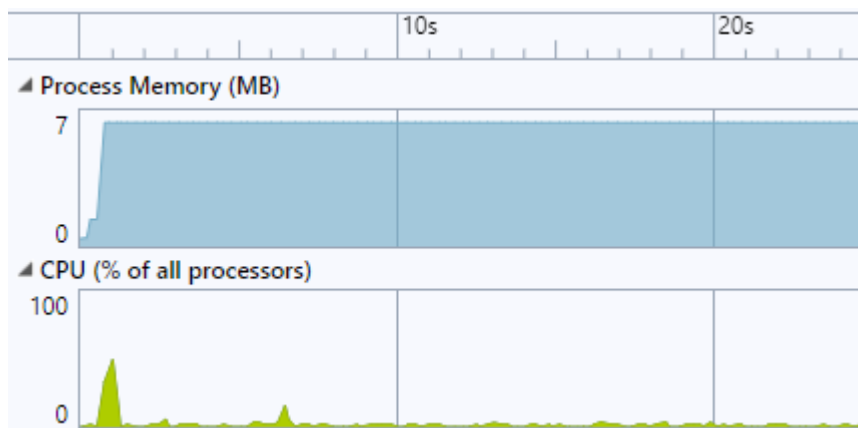


Obrázek 3.1: Ukázka mapování souboru do adresního prostoru procesu

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

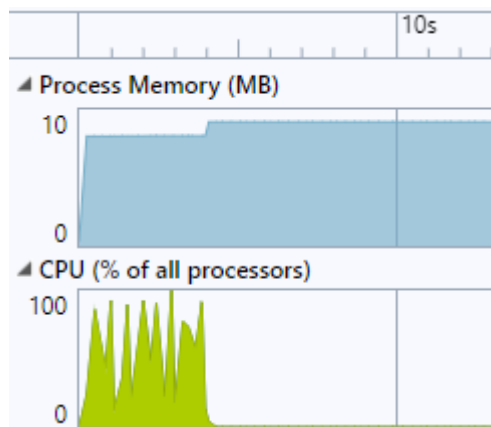
#### 3.1.1 První řešení

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 3.2).



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 výpočet mapovaného bloku. Pokud doba zpracování přesahuje dobu 5 sekund (stanoveno pomocí pokusného běhu), velikost bloku se zmenší tak, aby vlákno nemuselo čekat příliš 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 popsáno 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). Třída `File_mapping` obsahuje celkem dvě řídicí 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 zadání semestrální práce vyplývá omezení využití paměti 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 tento blok rozdělit mezi vlákna ke zpracování. V každé iteraci programu je stejný způsob rozdělování dat, liší se pouze funkce, která data zpracovává. 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

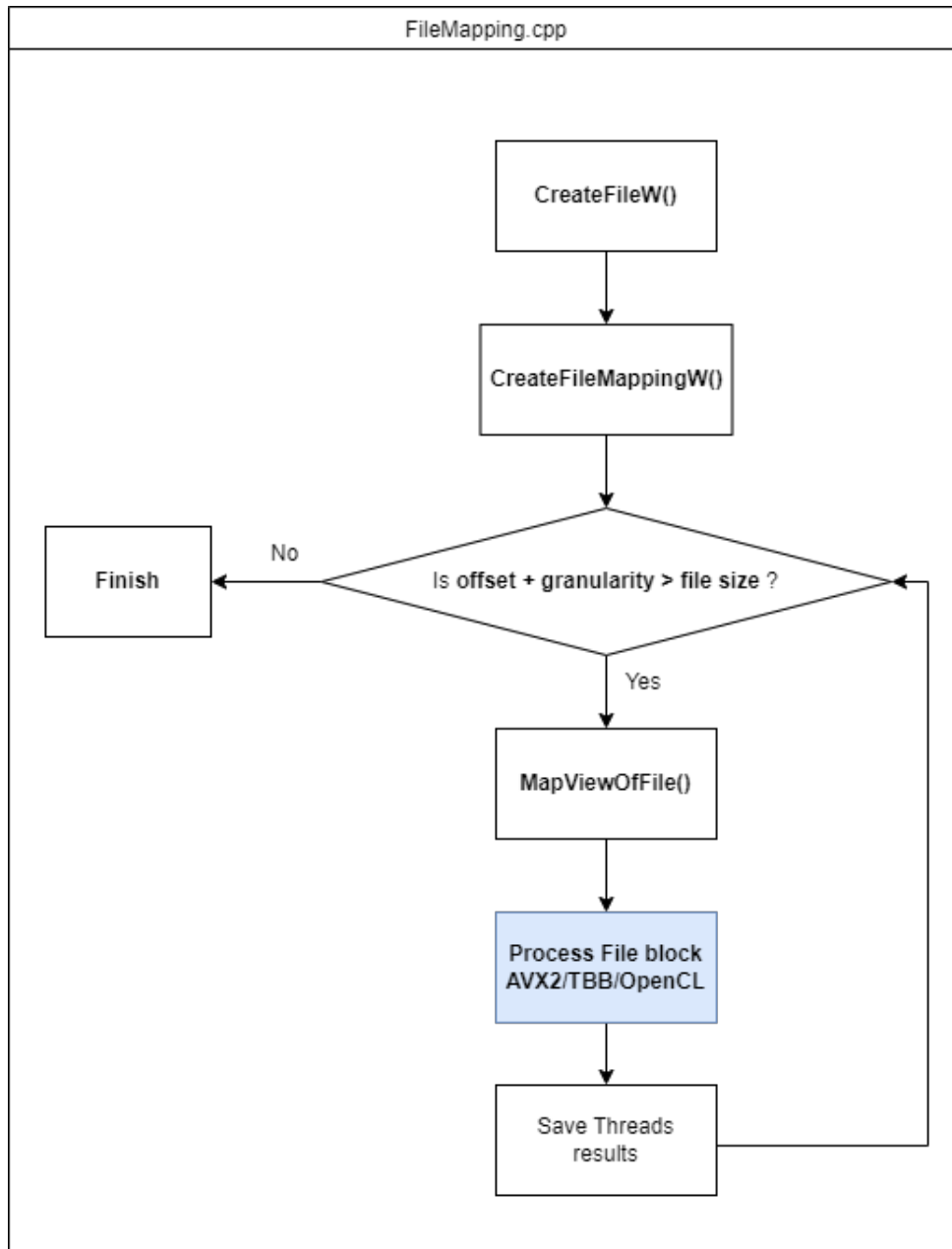
doběhne, vrátí objekt typu `std::future` jako výsledek výpočtu. Výsledky se na konci výpočtu vláken sjednocují.

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

### Režim ALL

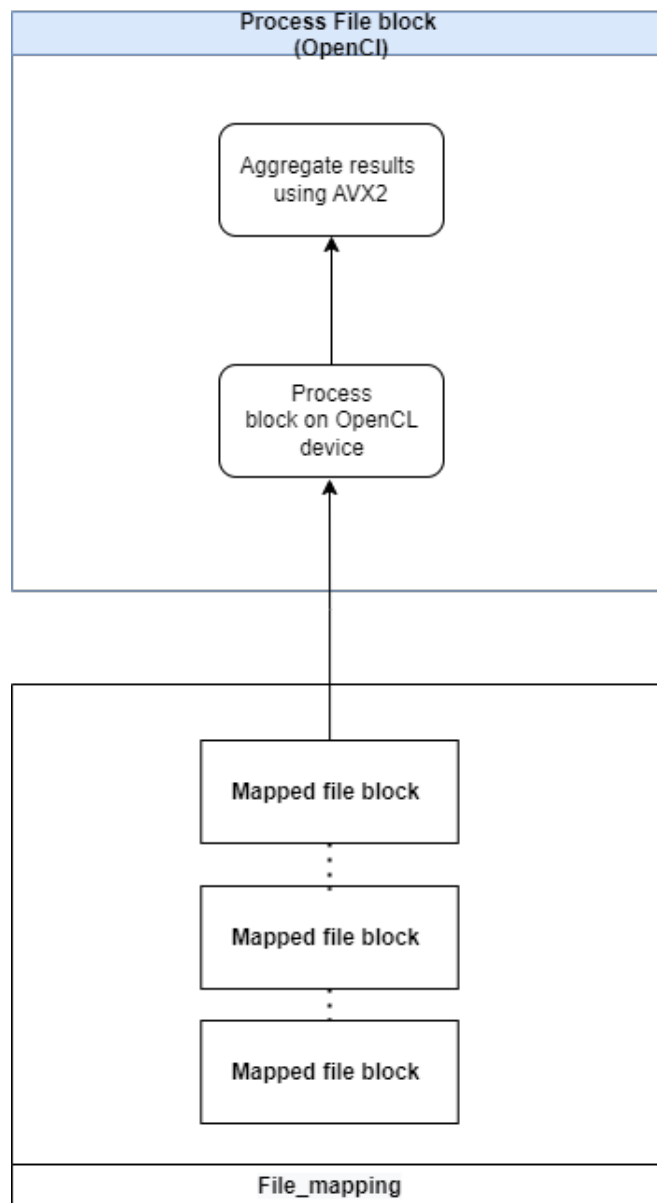
V případě režimu s využitím OpenCL zařízení je potřeba vytvářet buffer s daty, aby se mohl poslat na zařízení. Velikost jednoho bufferu nesmí přesahovat 1 GB podle zadání. Abych nemusela mapovat celý soubor do paměti a mohly se využít výhody mapování menších částí, musela jsem správně nastavit offsety.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

Také kopírování dat na OpenCL zařízení provádí CPU, to hodně zpomaluje výpočet. Proto jsem rozhodla načítat menší bloky s velikostí cca 0,5 GB. Dále je tento blok s daty rozdělen mezi jednotlivá OpenCL zařízení v různých vláknech pro zpracování. Příklad algoritmu mapování souboru pro režim ALL je ukázan na obrázku 3.5 .

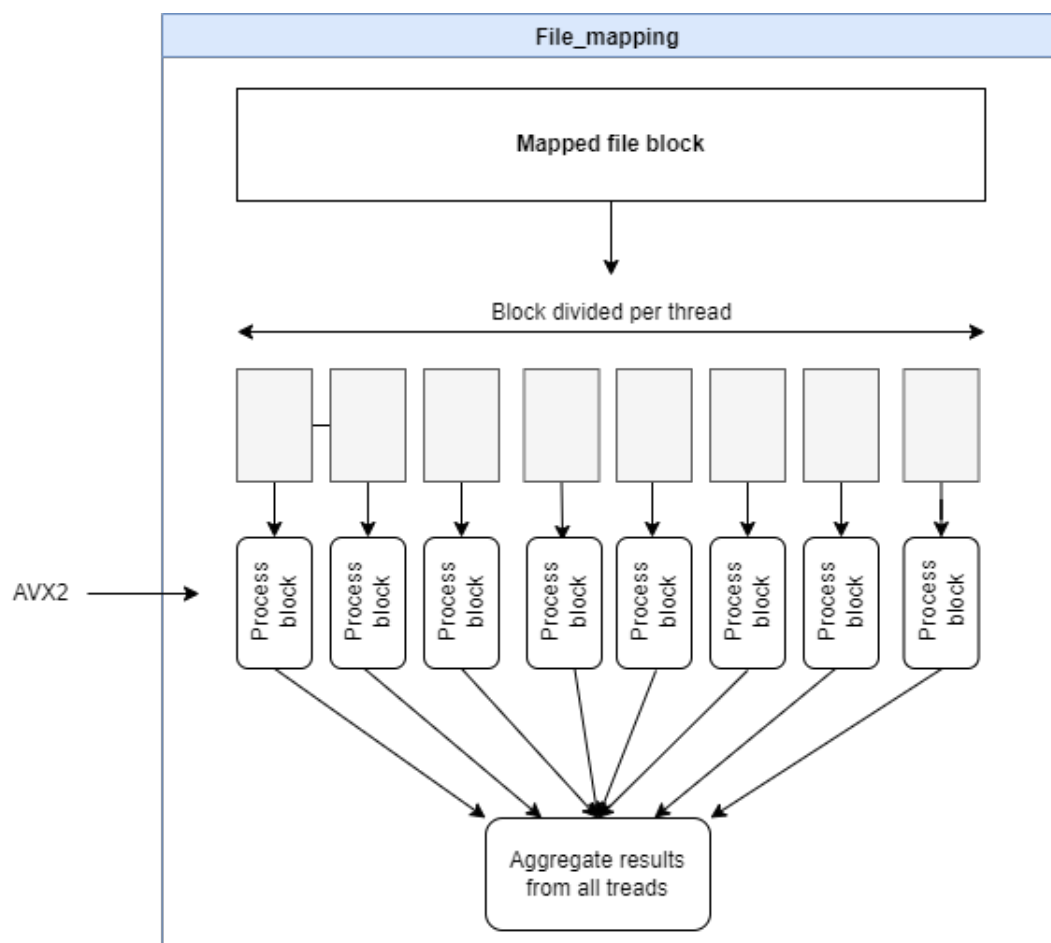


Obrázek 3.4: Ukázka algoritmu mapování souboru pro režim režimy





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 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 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 vlákně ve třídě `File_mapping`, která také rozhoduje kolik dat každému vlákně 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 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“ (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 použít 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,
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 3.7.

```

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 3.7: 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. Podrobnější porovnání bude ukázáno 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 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 (viz sekci 2.6), 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 RSS a také implementace funkce 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 zadane povinné argumenty:

- Cesta k souboru s daty
- Režim programu
  - **smp**
  - **seq**
  - **all**
  - Názvy jednotlivých 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 uspaní Watchdogu, zadáno v sekundách.

Příklad validního vstupu, který zapne program v režimu SMP bez optimalizace a watchdog vlákno bude kontrolovat běh programu každých 6 sekund:

```
1 "C:\gauss" smp -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**:

```
1 [Initial parameters]
2 -----
3 > File: C:\delete\gauss
4 > Mode: smp
5 > Number of threads: 8
6 > Optimization: TRUE
7 > Watchdog timer: 2 sec
8
9
10 > Started ..
11
12 [Statistics]
13 -----
14 > n: 1000000000
15 > sum: -3572.2
16 > mean: -3.5722e-06
17 > variance: 1.00005
18 > min: -6.10922
19 > max: 5.87261
20 > isNegative: 1
21 > isInteger: 0
22
23
24 [Results]
25 -----
26 > Gauss RSS: 0.00079459
27 > Poisson RSS: -nan(ind)
28 > Exponential RSS: 0.588353
29 > Uniform RSS: 0.477286
30
31
32
33 [Time]
34 -----
35 > Statistics computing time: 3.02456 sec.
36 > Histogram computing time: 4.37617 sec.
37 > RSS computing time: 0.0003251 sec.
38 > TOTAL TIME: 7.40135 sec.
39
40 > 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**:

```
1 [Initial parameters]
2 -----
3 > File: C:\delete\exp
4 > Mode: smp
5 > Number of threads: 8
6 > Optimization: TRUE
7 > Watchdog timer: 2 sec
8
9
10 > Started ..
11
12 [Statistics]
13 -----
14 > n: 137524224
15 > sum: 1.37528e+08
16 > mean: 1.00003
17 > variance: 1.00015
18 > min: 0
19 > max: 20.9085
20 > isNegative: 0
21 > isInteger: 0
22
23
24 [Results]
25 -----
26 > Gauss RSS: 0.24959
27 > Poisson RSS: 0.17973
28 > Exponential RSS: 0.121527
29 > Uniform RSS: 0.558112
30
31
32 [Time]
33 -----
34 > Statistics computing time: 4.54918 sec.
35 > Histogram computing time: 0.803888 sec.
36 > RSS computing time: 0.0002501 sec.
37 > TOTAL TIME: 5.35362 sec.
38
39 > 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**:

```
1 [Initial parameters]
2 -----
3 > File: C:\delete\poisson
4 > Mode: smp
5 > Number of threads: 8
6 > Optimization: TRUE
7 > Watchdog timer: 2 sec
8
9
10 > Started ..
11
12 [Statistics]
13 -----
14 > n: 141318656
15 > sum: 1.41323e+08
16 > mean: 1.00003
17 > variance: 0.999858
18 > min: 0
19 > max: 10
20 > isNegative: 0
21 > isInteger: 1
22
23
24 [Results]
25 -----
26 > Gauss RSS: 0.0203551
27 > Poisson RSS: 2.58775e-09
28 > Exponential RSS: 0.402114
29 > Uniform RSS: 0.208506
30
31
32 [Time]
33 -----
34 > Statistics computing time: 3.32517 sec.
35 > Histogram computing time: 0.688783 sec.
36 > RSS computing time: 0.0001357 sec.
37 > TOTAL TIME: 4.01441 sec.
38
39 > 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**:

```
1 [Initial parameters]
2 -----
3 > File: C:\delete\uniform
4 > Mode: smp
5 > Number of threads: 8
6 > Optimization: TRUE
7 > Watchdog timer: 2 sec
8
9
10 > Started ..
11
12 [Statistics]
13 -----
14 > n: 133684224
15 > sum: 6.68395e+07
16 > mean: 0.49998
17 > variance: 0.0833297
18 > min: 0
19 > max: 1
20 > isNegative: 0
21 > isInteger: 0
22
23
24 [Results]
25 -----
26 > Gauss RSS: 3.82184
27 > Poisson RSS: 7.97942
28 > Exponential RSS: 7.39103
29 > Uniform RSS: 0.0357194
30
31
32 [Time]
33 -----
34 > Statistics computing time: 4.0244 sec.
35 > Histogram computing time: 0.654257 sec.
36 > RSS computing time: 0.0001364 sec.
37 > TOTAL TIME: 4.67911 sec.
38
39 > Input data have 'Uniform distribution' with a=0 and b=1
```

## 5 Analýza výsledků

Příložená tabulka 5.1 ukazuje pozorování pěti běhu programu ve třech různých režimech s využitím CPU, tedy režim sekvenčního výpočtu, režim SMP - TBB a režim SMP s vektorizací. Vstupní data byla uložena na SSD disku. Lze pozorovat delší první běh programu v libovolném režimu. Souvisí to s tím, že požadované stránky s mapovaným souborem v prvním běhu nejsou v rychlé paměti RAM. Jakmile se do paměti RAM stránky načtou, program se v dalších bězích výrazně zrychlí.

#	Sekvenční výpočet	SMP - TBB	SMP - Vektorizace
1.	22.2017	28.9536	27.4167
2.	11.2105	5.50204	4.27749
3.	11.2373	4.37373	4.13617
4.	11.2793	4.45375	4.1926
5.	11.2817	4.6461	4.19529
Průměr	13.4421 sek.	9.585844 sek.	8.84365 sek.

Tabulka 5.1: Naměřené časy (sekundy) běhu programu v různých režimech s využitím CPU se souborem o velikosti 7 GB uloženém na SSD disku

Následující tabulka 5.2 ukazuje pět běhů programu s využitím GPU, tedy v režimu ALL nebo v režimu jednotlivých OpenCL zařízení). První běhy jsou opět pomalejší vlivem popsaného faktoru neuložených stránek v paměti RAM. V porovnání s tabulkou 5.1 je řešení s využitím GPU mnohem pomalejší. Důvodem je omezená paměť programu na 1 GB. Není možné posílat na OpenCL zařízení větší objem dat, protože by se neměl držet v paměti buffer s větší velikostí.

#	All	"NVIDIA GeForce MX150"	"Intel(R) UHD Graphics 620"
1.	98.7556	71.3399	130.496
2.	70.7089	47.9483	85.5481
3.	57.143	53.8338	80.7535
4.	84.0632	55.5232	80.0557
5.	61.269	55.5233	80.1668
Průměr	74.38794 sek.	56.8337 sek.	91.40402 sek.

Tabulka 5.2: Naměřené časy (sekundy) běhu programu v různých režimech s využitím GPU se souborem o velikosti 7 GB uloženém na SSD disku

Další tabulky 5.3 a 5.4 ukazují porovnání časů běhu programu v různých režimech (podobně jako předchozí tabulky 5.1 a 5.2). Rozdíl je ale v tom, že vstupní data byly uloženy na disku typu HDD.

#	Sekvenční výpočet	SMP - TBB	SMP - Vektorizace
1.	95.1375	539.369	358.862
2.	88.2781	8.34883	7.49745
3.	34.0248	8.05892	7.80384
4.	33.9824	8.2241	7.7205
5.	33.8996	8.29998	7.60831
Průměr	57.06448 sek.	114.460166 sek.	77.89842 sek.

Tabulka 5.3: Naměřené časy (sekundy) běhu programu v různých režimech s využitím CPU se souborem o velikosti 7 GB uloženém na HDD disku

#	All	"NVIDIA GeForce MX150"	"Intel(R) UHD Graphics 620"
1.	306.869	130.892	232.378
2.	99.0408	50.3058	106.945
3.	75.9644	53.9913	106.47
4.	90.0549	57.002	112.469
5.	222.29	91.7597	84.6309
Průměr	158.84382 sek.	76.79016 sek.	128.57858 sek.

Tabulka 5.4: Naměřené časy (sekundy) běhu programu v různých režimech s využitím GPU se souborem o velikosti 7 GB uloženém na HDD disku



Algoritmus mého programu prochází vstupní soubor dvakrát. Časy první a druhé iterace by měly být skoro totožné. Ale během prvního průchodu (sběr statistik ze vstupních dat) se stránky se souborem přemístí do paměti RAM, a druhý průchod souborem (sestavování histogramu) potom běží mnohem rychleji, jak lze pozorovat na tabulce 5.5. Zde je vidět, že první iterace se sběrem statistik trvá 22 sekund 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 5.5: Naměřené rozdíly časů (sekundy) běhu každé iterace programu v režimu SMP se souborem o velikosti 7 GB uloženém 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.

Po porovnání středních hodnot časů výpočtu (tabulka 5.6 a 5.7) se zjistilo, že se mi podařilo dosáhnout maximálně 52% zrychlení oproti sekvenčnímu běhu, za předpokladu, že vstupní soubor se nacházel na SSD disku. Nejrychlejším řešením byl režim SMP s využitím vektorizace a nejhorší čas předvedl režim s využitím pouze jednoho OpenCL zařízení ("Intel(R) UHD Graphics 620"). Režim ALL je pomalejší z důvodu čekání na pomalejší zařízení ("Intel(R) UHD Graphics 620") (obrázek 5.1).

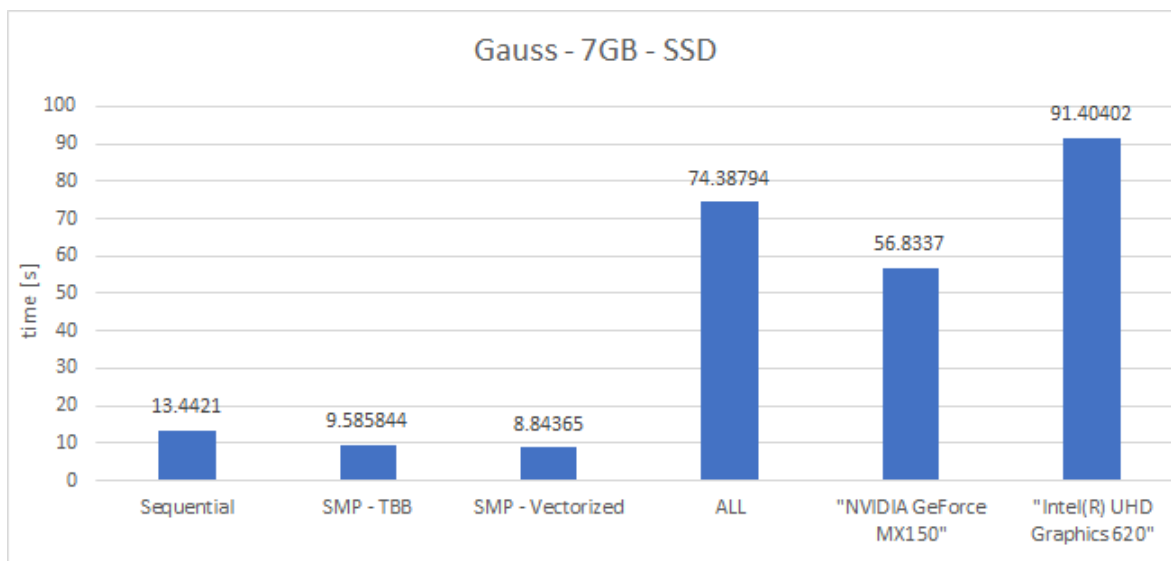
Sekvenční program ve střední hodnotě času se ukázal jako rychlejší řešení pro zpracování souboru z HDD disku (obrázek 5.2).

ALL	-82%
SMP - TBB	40%
SMP - Vectorized	52%
"NVIDIA GeForce MX150"	-76%
"Intel(R) UHD Graphics 620"	-85%

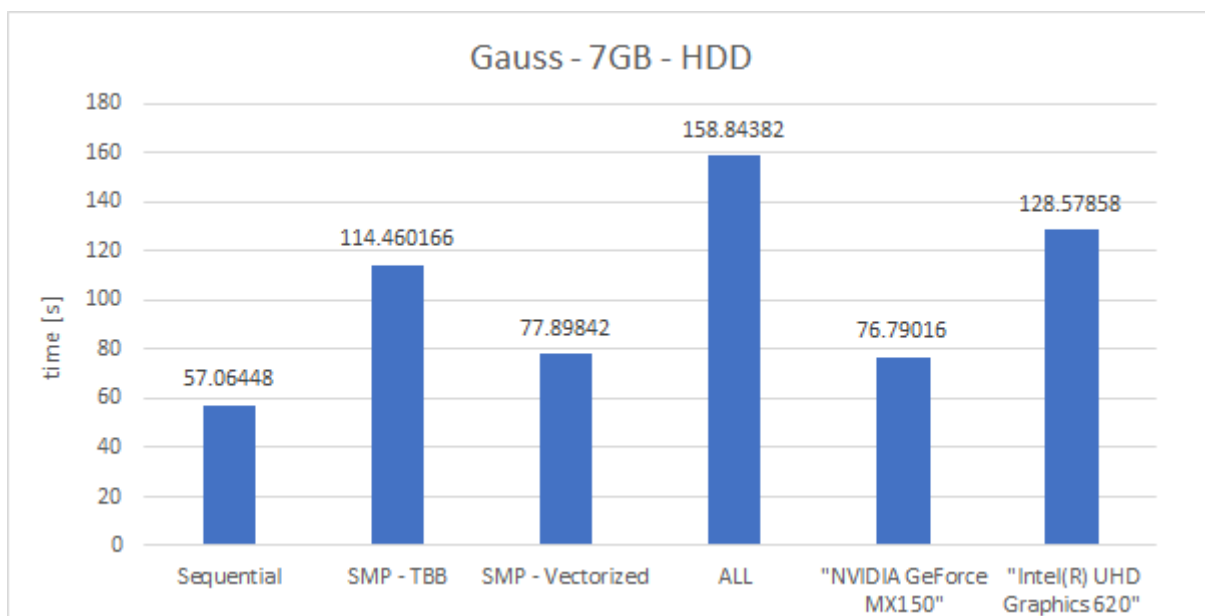
Tabulka 5.6: Porovnání rychlostí jednotlivých režimů programu oproti sekvenčnímu režimu (velikost souboru 7 GB, SSD disk)

ALL	-64%
SMP - TBB	-50%
SMP - Vectorized (opt)	-27%
"NVIDIA GeForce MX150"	-26%
"Intel(R) UHD Graphics 620"	-56%

Tabulka 5.7: Porovnání rychlostí jednotlivých režimů programu oproti sekvenčnímu režimu (velikost souboru 7 GB, HDD disk)



Obrázek 5.1: Porovnání rychostí jednotlivých režimů programu (velikost souboru 7 GB, SSD disk)



Obrázek 5.2: Porovnání rychostí jednotlivých režimů programu (velikost souboru 7 GB, HDD disk)

## 6 Závěr

Zadání semestrální práce bylo splněno. Program vrací správný odhad pravděpodobnosti pro vstupní data na základě spočítané 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 vymyšlením 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 výpočtu paralelizace s využitím OpenCL zařízení. Kopírování dat do zařízení zabírá moc času a paměti.

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](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 test*. 2022. Available at [https://en.wikipedia.org/wiki/Residual\\_sum\\_of\\_squares](https://en.wikipedia.org/wiki/Residual_sum_of_squares).
- Wikipedia. *Residual sum of squares*. 2022. Available at [https://en.wikipedia.org/wiki/Residual\\_sum\\_of\\_squares](https://en.wikipedia.org/wiki/Residual_sum_of_squares).