Analýza statistických rozdělení dat KIV/PPR

David Markov

Fakulta aplikovaných věd
Západočeská univerzita v Plzni
7. ledna 2023

Obsah

Kapitola 1

Zadání

1.1 Standardní zadání

Program semestrální práce dostane, jako jeden z parametrů, zadaný soubor, přístupný pouze pro čtení. Bude ho interpretovat jako čísla v plovoucí čárce - 64-bitový *double*. Program rozhodne, ke kterému rozdělení mají data nejblíže - zda k normálnímu/Gauss, Poissonovu, exponenciálnímu či rovnoměrnému - viz látka *KMA/PSA* a *KIV/VSS*. Program vypíše hodnoty charakterizující rozdělení a zdůvodnění svého výsledku.

Program se bude spouštět následovně:
`pprsolver.exe <soubor> <procesor>`

- *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 samostané argumenty - pozor, v systému může být několik OpenCL platforem

Součástí programu bude *watchdog* vlákno, které bude hlídat správnou funkci programu.

Testovaný soubor bude velký několik GB, ale paměť bude omezená na 1GB. Zařídí validátor. Program musí skončit do 15 minut na *iCore7 Skylake*.

1.2 Zpracování

Samostatná práce využije alespoň dvě z celkem tří možných technologií:

- Paralelní program pro systém se sdílenou pamětí - *C++* vč. *PSTL C++17*, popř. *WinAPI*; program musí využít buď autovektorizaci nebo manuální *AVX2* vektorizaci výpočtu zadaného problému.
- Program využívající asymetrický multiprocesor - konkrétně x86 CPU a *OpenCL* kompatibilní *GPGPU*. Po domluvě lze použít *SYCL*.
- Po domluvě - Paralelní program pro systém s distribuovanou pamětí - *C++ MPI*

Kapitola 2

Způsoby určení pravděpodobnostního rozdělení

Existuje mnoho způsobů, kterými lze přiřadit datovou sadu ke konkrétním pravděpodobnostním rozdělením. Každý se hodí pro data s jinými vlastnostmi, navíc jsou zadáním předloženy omezující podmínky na výkonnost. Všechny tyto prerekvizity bude nutno zahrnout jako kritéria výběru.

2.1 Chí-kvadrát test dobré shody

Test dobré shody je statistická metoda, umožňující ověřit, zda-li má náhodná veličina předem dané pravděpodobnostní rozdělení. Tato metoda je založena na transformaci multinomické náhodné veličiny na veličinu, která má rozdělení přibližně chí kvadrát [2]. Postup testu je následující:

1. Rozdělení oboru hodnot náhodné veličiny na k nepřekrývajících se částí
2. Určení pravděpodobností p_i nabytí hodnoty z i-té části náhodnou veličinou
3. Provedení N pokusů a určení četností X_i nabytí hodnoty z i-té části náhodnou veličinou
4. Porovnání očekávaných četností Np_i se skutečnými četnostmi X_i

Při porovnání četností jsou vypočtena náhodná veličina χ^2 podle vzorce 2.1.

$$\chi^2 = \sum_{i=1}^{k} \frac{(X_i - Np_i)^2}{Npi} \tag{2.1}$$

Pokud má testovaná veličina předpokládané rozdělení, má veličina χ^2 přibližně rozdělení chí kvadrát.

2.2 Srovnání na základě kurtózy

Jako nejjednodušší způsob srovnání jednotlivých rozdělení se jeví srovnání jejich vlastností. Pokud by každé z pravděpodobnostních rozdělení náhodné veličiny mělo nějakou specifickou vlastnost, či vlastnost společnou, nabývající specifické hodnoty, bylo by možné určit rozdělení náhodné veličiny pouze výpočtem této vlastnosti, resp. hodnoty. Přesně touto vlastností je tzv. *kurtóza*.

Kurtóza, často nazývaná „koeficient špičatosti“, je vlastnost náhodného rozdělení, související s jeho čtvrtým momentem. Konkrétně se jedná o škálovanou hodnotu čtvrtého momentu náhodného rozdělení [6]. Označení „špičatost“ je tedy nesprávné, jelikož kurtóza souvisí s krajními percentily, nikoli s rozložením dat kolem střední hodnoty. Kurtóza je tím vyšší, čím extrémnější jsou hodnoty odchylek, tzv. *outlierů*. Její hodnotu lze vyjádřit pomocí vzorce 2.2.

$$Kurt\left[X\right] = E\left[\left(\frac{X-\mu}{\sigma}\right)^4\right] = \frac{E\left[(X-\mu)^4\right]}{\left(\left[(X-\mu)^2\right]\right)^2} = \frac{\mu_4}{\sigma^4}, \tag{2.2}$$

kde μ_4 je čtvrtý centrální moment a σ je střední odchylka.

Hodnoty kurtózy náhodného rozdělení jsou specifické pro každé jednotlivé rozdělení (viz tabulka 2.1). Jsou navíc dostatečně odlišné na to, aby bylo možné na jejím základě rozhodnout o příslušnosti náhodné veličiny ke konkrétnímu pravděpodobnostnímu rozdělení.

Rozdělení	Kurtóza
Rovnoměrné	$-\frac{6}{5}$
Poissonovo	$\frac{1}{\lambda}$
Exponenciální	6
Normální	0

Tabulka 2.1: Specifické hodnoty kurtózy jednotlivých náhodných rozdělení.

2.3 Výběr vhodné klasifikační metody

Obě navržené metody klasifikace jsou validní pro zadanou úlohu. Každá má ale své výhody i nevýhody, které se mohou projevit při klasifikaci různých datových sad.

Test dobré shody je spolehlivou a ověřenou metodou klasifikace. Hlavní výhodou je stabilita výpočtu na různých datových sadách a ověřenost algoritmu jakožto statistické metody. Dílčí kroky této metody jsou navíc poměrně snadno paralelizovatelné i vektorizovatelné. Nevýhodou testu dobré shody je jeho náchylnost na *outliery* a nutnost dvojího průchodu dat, pokud je oborem hodnot náhodné veličiny prostor reálných čísel. V prvním průchodu by bylo třeba zjistit minimum, maximum a střední hodnotu vstupních dat, ve druhém pak začít s rozdělením na nepřekrývající se části. Navíc by bylo třeba vyfiltrovat extrémní hodnoty, které by mohly určení jednotlivých *bucketů* ztížit.

Výpočet kurtózy množiny dat je poněkud ad-hoc metodou, v praxi není využití výpočtu této vlastnosti k určení příslušnosti k pravděpodobnostnímu rozdělení běžné. Algoritmus je ovšem jednoduchý a ve standardních podmínkách stabilní. Existují takové metody výpočtu, které umožňují tuto vlastnost spočíst v jediném průchodu vstupních dat, navíc s možností paralelizace či vektorizace výpočtu. Ze vrozce je ale zřejmé, že při nevhodné množině vstupních dat může tento algoritmus selhat z důvodu přetečení registrů při operaci násobení (viz čtvrtá mocnina ve vzorci 2.2).

V této práci bude využita metoda klasifikace na základě kurtózy. Jedná se nejen o jednoduchý algoritmický výpočet, ale i o neotřelý přístup k řešení daného problému. Konkrétní metodou výpočtu kurtózy vstupních dat bude *Welfordův online algoritmus*. Jedná se o inkrementální výpočet, určený primárně k výpočtu prvních dvou momentů náhodné veličiny, tj. střední hodnoty a rozptylu, resp. střední odchylky [9]:

$$\begin{aligned} M_{1,n} &= x1 + \frac{x_n - M_{1,n-1}}{n} \\ M_{2,n} &= M_{2,n-1} + (x_n - M_{1,n-1})(x_n - M_{1,n}) \\ \sigma_n^2 &= \frac{M_{2,n}}{n} \\ s_n^2 &= \frac{M_{2,n}}{n-1} \end{aligned} \tag{2.3}$$

Tento algoritmus lze ale dále rozšířit pro výpočet třetího a čtvrtého momentu náhodné veličiny, tedy k výpočtu kurtózy [7].

$$\begin{aligned} \delta &= x - m \\ m_n &= m_{n-1} + \frac{\delta}{n} \\ M_{2,n} &= M_{2,n-1}\delta^2\frac{n-1}{n} \\ M_{3,n} &= M_{3,n-1} + \delta^3\frac{(n-1)(n-2)}{n^2} - \frac{3\delta M_{2,n-1}}{n} \\ M_{4,n} &= M_{4,n-1} + \frac{\delta^4(n-1)\left(n^2-3n+3\right)}{n^3} + \frac{6\delta^2 M_{2,n-1}}{n^2} - \frac{6\delta M_{3,n-1}}{n} \\ kurtosis &= \frac{(nM_{4,n})}{M_{2,n}^2} - 3 \end{aligned} \tag{2.4}$$

Welfordův algoritmus je navíc speciálním případem algoritmu, který spojuje výsledky výpočtu pro dvě arbitrární množiny A a B. Terriberry formuloval tyto

rovnice pro všechny čtyři momenty náhodné veličiny [8]:

$$
\begin{aligned}
n_C &= n_A + n_B \\
\delta &= M_{1,A} - M_{1,B} \\
M_{1,C} &= M_{1,A} + \delta \frac{n_B}{n_C} \\
M_{2,C} &= M_{2,A} + M_{2,B} + \delta^2 \frac{n_A n_B}{n_C} \\
M_{3,C} &= M_{3,A} + M_{3,B} + \delta^3 \frac{n_A n_B \left(n_A - n_B\right)}{n_C^2} + 3\delta \frac{n_A M_{2,B} - n_B M_{2,A}}{n_C} \\
M_{4,C} &= M_{4,A} + M_{4,B} \\
&\quad + \delta^4 \frac{n_A n_B \left(n_A^2 - n_A n_B + n_{_B}^{2}\right)}{n_C^3} \\
&\quad + 6\delta^2 \frac{n_A^2 M_{2,B} + n_B^2 M_{2,A}}{n_C^2} + 4\delta \frac{n_A M_{3,B} - n_B M_{3,A}}{n_C}
\end{aligned}
\tag{2.5}
$$

Kapitola 3

Volba zadaných technologií

V této práci bude využit paralelní výpočet pomocí *PSTL C++ 17* včetně autovektorizace výpočtu, spolu s využitím *OpenCL* kompatibilního *GPGPU*.

Kapitola 4

Sériová implementace

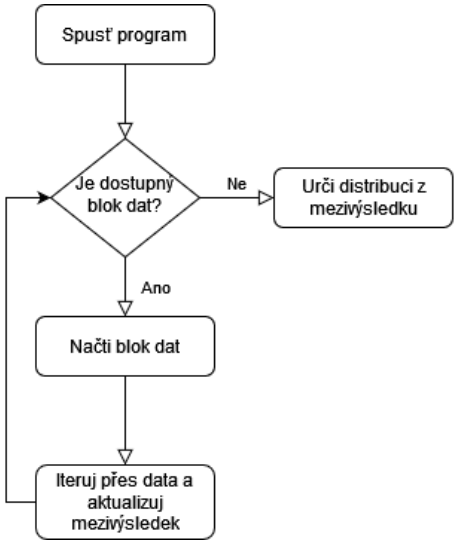
Před zahájením úvah o paralelizaci výpočtu bude nejprve provedena implementace sériového výpočtu. Sériový výpočet využije sekvenční varianty Welfordova online algoritmu. Program bude postupně iterovat přes vstupní data a s každým prvkem aktualizuje mezivýsledek výpočtu podle vzorce 2.4, viz pseudokód 2 [3]. Jelikož je ze zadání omezena maximální dostupná paměť programu a lze očekávat vstupní data o velikosti větší než je tento limit, nelze uvažovat o načtení všech dat do paměti.

Program bude postupně načítat bloky dat, které iterativně zpracuje a uloží mezivýsledek. Tuto akci bude opakovat tak dlouho, dokud jsou na vstupu dostupné bloky dat. Posledním mezivýsledkem bude konečné řešení výpočtu, ze kterého bude vypočtena kurtóza a určeno pravděpodobnostní rozdělení vstupních dat. Výpočet je vizualizován na obr. 4.1.

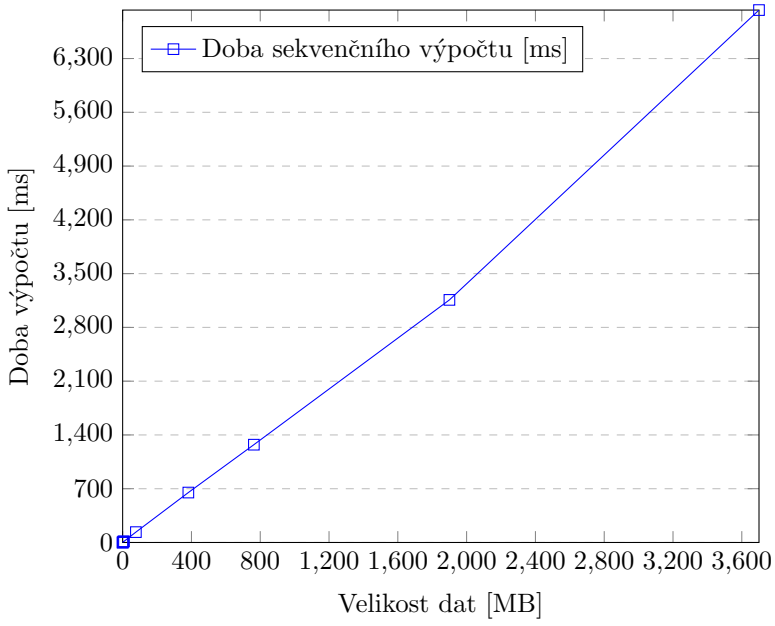
4.1 Benchmarking

Výpočet byl spuštěn na množinách dat o různých velikostech od desítek kilobytů až po jednotky giga-bytů. Program byl testován na zařízení s operačním systémem Windows 10 Home verze 21H2, build 19044.2364. Zařízení disponuje 8-jádrovým procesorem AMD Ryzen 7 5800U s integrovanou grafickou kartou Radeon Graphics. Měření byla realizována několikanásobným spuštěním výpočtu na stejných datech a určením aritmetického průměru výsledných dob běhu.

Výsledkem měření je zřejmá lineární závislost doby výpočtu na velikosti vstupních dat. Jednotlivé naměřené hodnoty jsou uvedeny v tabulce A.1, závislost je vizualizována v grafu na obr. 4.2.



Obrázek 4.1: Vývojový diagram sekvenčního výpočtu



Obrázek 4.2: Závislost doby sekvenčního výpočtu na velikosti vstupních dat

Kapitola 5

Paralelizace výpočtu

Sériový výpočet sice poskytl uspokojivé výsledky, ale obsahuje několik míst pro možnou optimalizaci. Samotný cyklus iterace přes vstupní data lze totiž vektorizovat, tedy využít pro výpočet rozšířené registry a instrukce architektury *AVX2*. Welfordův algoritmus lze navíc i dále paralelizovat.

5.1 Vektorizace

Ve stávající podobně iterativního algoritmu není automatická vektorizace překladačem možná. Ve smyčce totiž výsledek i-té iterace závisí na výsledku předchozího kroku. Dochází zde k netriviální redukci vektoru dat do jediné hodnoty. Překladač tudíž nedokáže cyklus automaticky vektorizovat. Tento stav rozpozná a vypíše odpovídající upozornění:

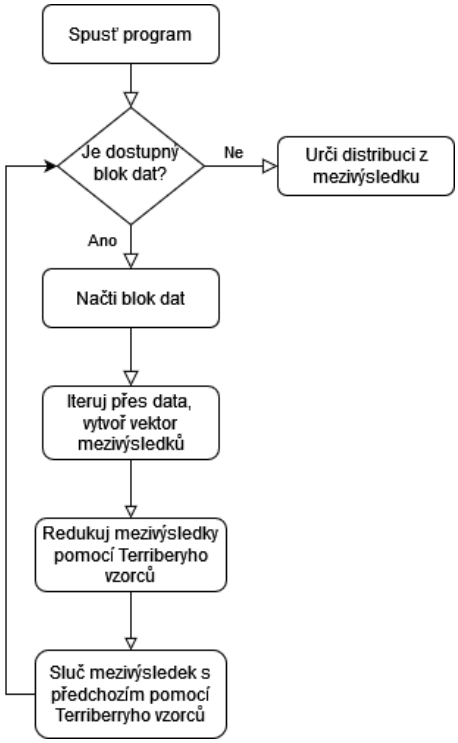
```
1>--- Analyzing function: virtual class CStatistics
__cdecl cpu::CSeq_Stats_Calculator::Analyze_Vector(class
std::vector <double,class std::allocator<double>> const &
__ptr64) __ptr64
1>C:\Users\Marko\Skola\repos\ppr\src\cpu\cpu_statistics.cpp(16) :
info C5002: loop not vectorized due to reason '1105'
```

Listing 1: Selhání automatické vektorizace Welfordova algoritmu z důvodu '1105' - neznámá operace redukce.

Je tedy potřeba algoritmus upravit tak, aby byla i-tá iterace závislá jedině na mezivýsledku $(i-k)$-té iterace. Toho lze snadno docílit rozdělením algoritmu na dva cykly - vnější cyklus, který bude iterovat vždy po konstantní velikosti bloku k a vnitřní cyklus, který bude iterovat přes každý prvek v daném bloku. Mezivýsledky i-té operace v n-tém bloku se budou redukovat s mezivýsledky předchozích i-tých operací v každém bloku. Tyto dílčí mezivýsledky lze nakonec zredukovat do jediného, díky již zmíněné vlastnosti Welfordova algoritmu - je

pouze speciální variantou algoritmu, který funguje pro dvě arbitrární množiny A a B, jejichž charakteristiky lze sloučit pomocí rovnice 2.5. Pseudokódem zapsáno viz B. Vývojový diagram algoritmu viz obr. 5.1. Úspěšnost automatické vektorizace lze ověřit kontrolou výpisů překladače:

```
1>--- Analyzing function: virtual class CStatistics
__cdecl cpu::CPar_Stats_Calculator::Analyze_Vector(class
std::vector <double,class std::allocator<double>> const &
__ptr64) __ptr64
1>C:\Users\Marko\Skola\repos\ppr\src\cpu\cpu_statistics.cpp(49)
: info C5001: loop vectorized
1>C:\Users\Marko\Skola\repos\ppr\src\cpu\cpu_statistics.cpp(70)
: info C5002: loop not vectorized due to reason '1305'
```

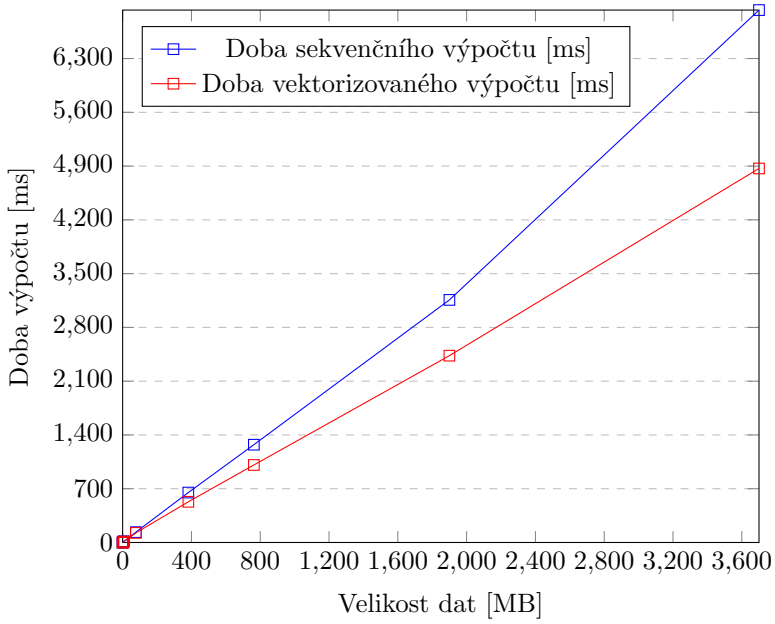


Obrázek 5.1: Vývojový diagram vektorizovaného výpočtu.

5.2 Srovnání se sériovou verzí

Testování vektorizovaného algoritmu je realizováno na totožném stroji jako měření algoritmu sekvenčního. Totéž platí i o různých velikostech vstupních dat. Jednotlivé naměřené hodnoty jsou uvedeny v tabulce A.1.

V grafu 5.2 lze vidět srovnání těchto svou implementací. Závislost doby výpočtu na velikosti vstupních dat je opět lineární funkcí, ovšem roste značně pomaleji než u sekvenčního výpočtu. Na malém vzorku dat ($< 100MB$ je zrychlení výpočtu zanedbatelné, ovšem u giga-bytových velikostí vstupních dat již pozorujeme znatelné urychlení.

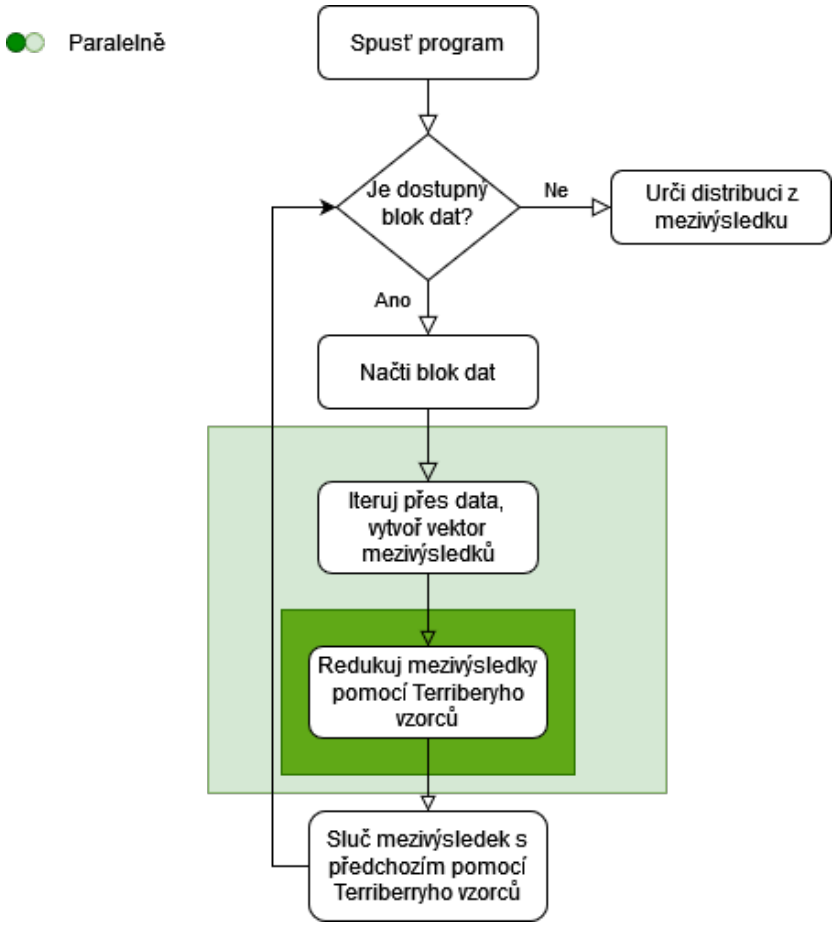


Obrázek 5.2: Srovnání dob sekvenčního a vektorového výpočtu v závislosti na velikosti vstupních dat.

5.3 Vícevláknový výpočet

Vektorizováním sekvenčního výpočtu bylo dosaženo značného urychlení. Nicméně, zatím byl uvažován výpočet pouze na jednom jádře CPU. Nabízí se tedy uvažovat zavedení paralelního vícevláknového výpočtu.

Data jsou analyzována pomocí Welfordova/Chanova algoritmu (rovnice 2.4) a mezivýsledky redukovány sekvenčně pomocí Terriberryho rovnic (2.5) na dvou místech - uvnitř vektorového výpočtu, kde je vektor mezivýsledků redukován do jediného výsledku a na konci iterace algoritmu, kdy je nový mezivýsledek sloučen s původním. V obou zmíněných místech se nabízí zavedení paralelního výpočtu, viz obr. 5.3.



Obrázek 5.3: Paralelizační příležitosti vektorového algoritmu.

Redukce vektoru mezivýsledků vektorového výpočtu je prováděna algoritmem `std::reduce`[1] standardní knihovny jazyka C++. Ta od verze C++17 nabízí ovšem i paralelní verze většiny implementovaných algoritmů v rámci tzv. *PSTL*[2]. K redukci mezivýsledků bude tedy využita paralelní varianta `std::reduce`, kterou lze spustit předáním parametru `std::execution::par`.

Samotný vektorový výpočet lze provádět v samostatném vlákně. Podobně jako uvnitř výpočtu, mezivýsledky je možné ukládat samostatně a nakonec je všechny, opět paralelně, redukovat do jediného finálního výsledku. Abychom nemuseli manuálně implementovat celé schéma *Worker - Farmer*, využijeme další části *PSTL*, kterou je funkce `std::async`.

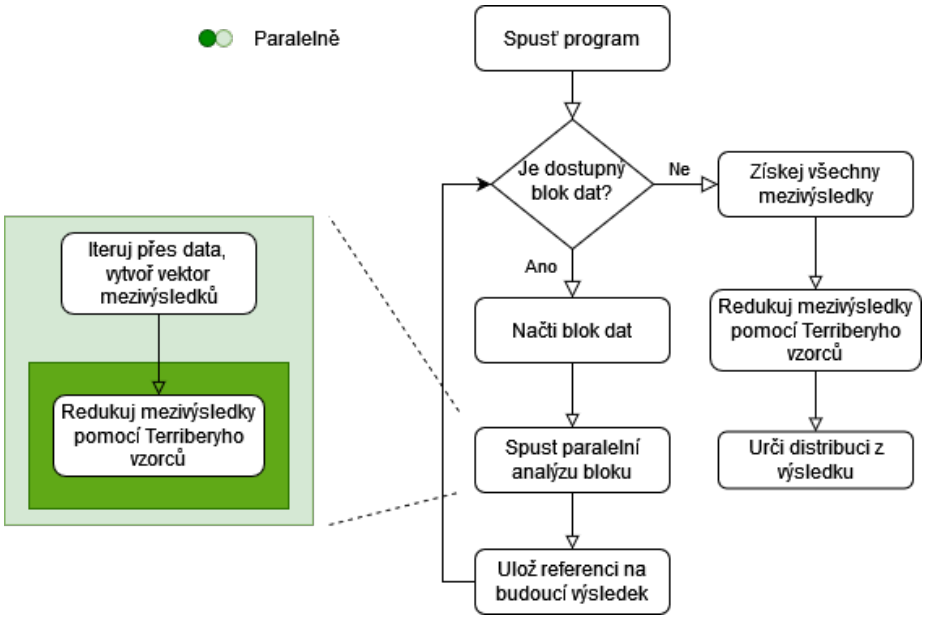
Tato funkce umožňuje spuštění funkce předané parametrem jinak než sekvenčně, v závislosti na nastavení spouštěcí politiky. Návratovou hodnotou je poté instance `std::future`, reprezentující sdílený stav volajícího a výkonného vlákna, který uchovává potenciální výsledek asynchronní operace [4]. Funkci lze momentálně volat s jednou ze dvou spouštěcích politik - `std::launch::async`, která spustí výpočet na novém vlákně, které může být potenciálně součástí *poolu* vláken, nebo `std::launch::deferred`, která spustí výpočet ve stejném vlákně, ale až v moment, kdy bude přistoupeno v potenciálnímu výsledku (tzv. *lazy evaluace*) [5].

Pro potřeby tohoto výpočtu bude využita politika `std::launch::async`, jelikož chceme využít co nejvíce jader CPU. Analýza každého bloku dat bude tedy volána pomocí `std::async(std::launch::async, ...)` a reference na potenciální výsledky budou uchovávány ve vektoru, který bude na konci výpočtu paralelně zredukován do jediného výsledku. Výpočet je vizualizován ve vývojovém diagramu 5.4.

5.4 Benchmark

Po vynesení naměřených časů do grafu 5.5 bylo zjištěno, že paralelizace vektorového výpočtu již další urychlení nepřinesla. Naopak došlo ke zpomalení běhu programu ve srovnání s čistě vektorovým výpočtem. Je tedy třeba zaměřit se na místa, ve kterých byla paralelizace zavedena a provést analýzu jednotlivých interagujících částí programu. Je například pravděpodobné, že načtení bloku dat ze souboru je časově náročnější než analýza tohoto bloku, tudíž asynchronní spouštění výpočtu nepřinese žádné urychlení. Naopak může pouze zavést další režii na případné vytvoření vlákna a následné získání výsledku, která je potenciálně časově náročnější, než samotný výpočet, a tedy způsobí celkové zpomalení běhu programu.

[1] https://en.cppreference.com/w/cpp/algorithm/reduce
[2] Parallel Standard Template Library



Obrázek 5.4: Paralelní vektorový algoritmus.

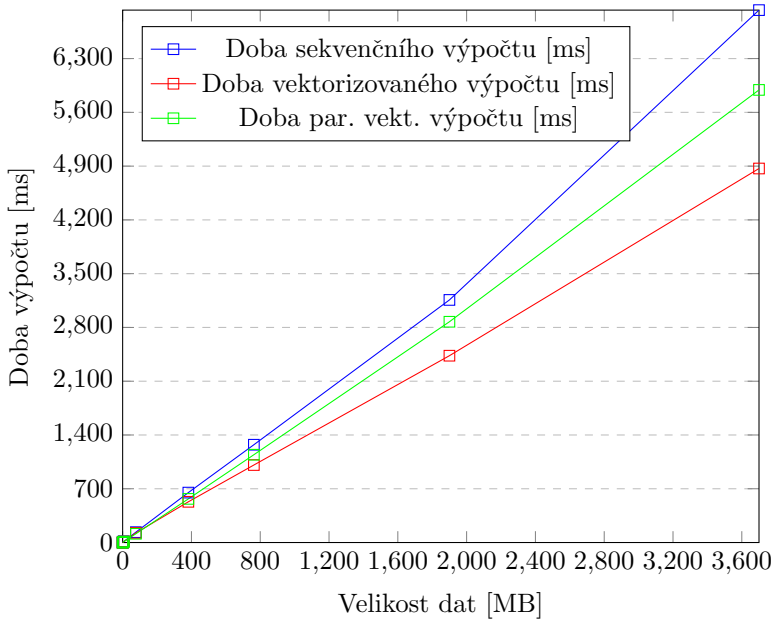
5.5 Výpočet pomocí grafického OpenCL akcelerátoru

Druhou zvolenou technologií paralelního výpočtu je běh na *GPGPU* podporujícím *OpenCL*.

Nejprve se zaměříme na bod zadání, který určuje možnost spuštění výpočtu na konkrétních OpenCL zařízeních. Jejich názvy budou načteny z *CLI*. Program projde všechny přítomné OpenCL platformy[3], a vyhledá dané zařízení (viz pseudokód B). Pokud je alespoň jedno zařízení nalezeno, dojde k sestavení OpenCL programu pro nalezená zařízení s algoritmem, analogickým tomu pro CPU. Ten bude následně na *GPGPU* spuštěn s každým načteným datovým blokem již v rámci paralelizace, provedené v sekci 5.3. Algoritmus je znázorněn v diagramu 5.6.

Samotný algoritmus analýzy je modifikací paralelního algoritmu výpočtu na CPU, která je navržena pro běh na OpenCL zařízení. Akcelerátoru jsou jako parametry předány ukazatel na vektor vstupních dat, počet prvků v tomto vektoru a ukazatele na paměť, kam má zapsat výsledek analýzy. Ukazatele jsou mapovány z paměti hostujícího zařízení do paměti akcelerátoru a zápis je tedy prováděn přímo do paměti hosta. Vnitřní cyklus výpočtu pracuje obdobně jako algoritmus po vektorizaci, ačkoli je paralelní krok prováděn po „širších“ blocích

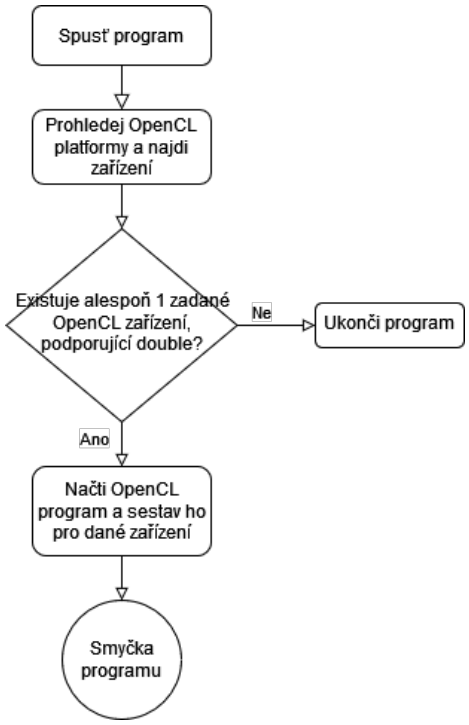
[3]Na zařízení může být přítomno několik OpenCL platforem. Např. pokud je instalováno CPU od výrobce *Intel* a dedikovaná grafická karta od výrobce *NVidia* nebo *AMD*.



Obrázek 5.5: Srovnání dob sekvenčního, vektorového a paralelního výpočtu v závislosti na velikosti vstupních dat.

- o velikosti *NDRange*, v tomto případě nastaveno na `256`. Mezivýsledky jsou akumulovány do lokální paměti akcelerátoru. Jakmile je průchod vektorem dokončen, následuje redukce mezivýsledků v lokální paměti pomocí Terriberryho rovnic. Levá polovina mezivýsledků je vždy paralelně sloučena s pravou polovinou, čímž získáme poloviční počet dalších mezivýsledků. Tento krok se opakuje tak dlouho, dokud nezůstane jeden finální výsledek, který je následně zapsán do paměti hosta. Algoritmus je vizualizován v diagramu 5.7. Výkonný kód OpenCL kernelu je uveden v příloze C.

Jelikož může být zadaných OpenCL zařízení pro výpočet několik, je třeba jim přiřazovat datové bloky k výpočtu. Toto rozdělení práce bude realizováno analogií algoritmu *Round Robin* plánovače operačního systému. Při přiřazení práce bude seznam zařízení procházen od začátku do konce a ukazatel na aktuální zařízení bude podle toho posouván. Jakmile se ukazatel posune za poslední zařízení v seznamu, bude přesunut opět na začátek.



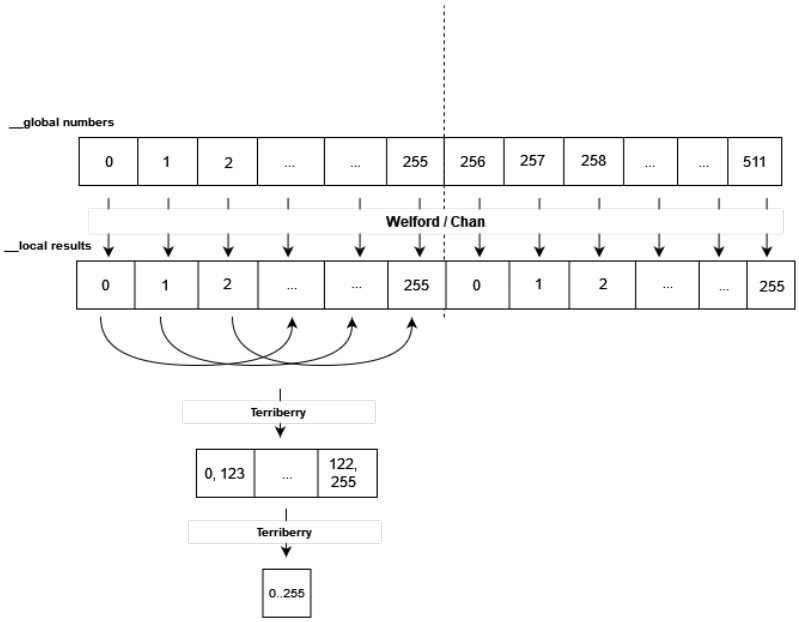
Obrázek 5.6: Vývojový diagram výpočtu na OpenCL zařízení. Smyčkou programu se rozumí totožná smyčka, jako znázorněna v diagramu 5.4.

5.6 Výsledky měření

Výsledky, vynesené v grafu 5.8, ukazují propastné rozdíly mezi výpočty na testovaných platformách. Ačkoli je opět vidět lineární závislost doby výpočtu na velikosti dat, výpočet na GPGPU (konkrétně na integrované grafické kartě *AMD Radeon Graphics*) je řádově pomalejší než výpočet na CPU. Příčina tkví s vysokou pravděpodobností ve dvou faktorech.

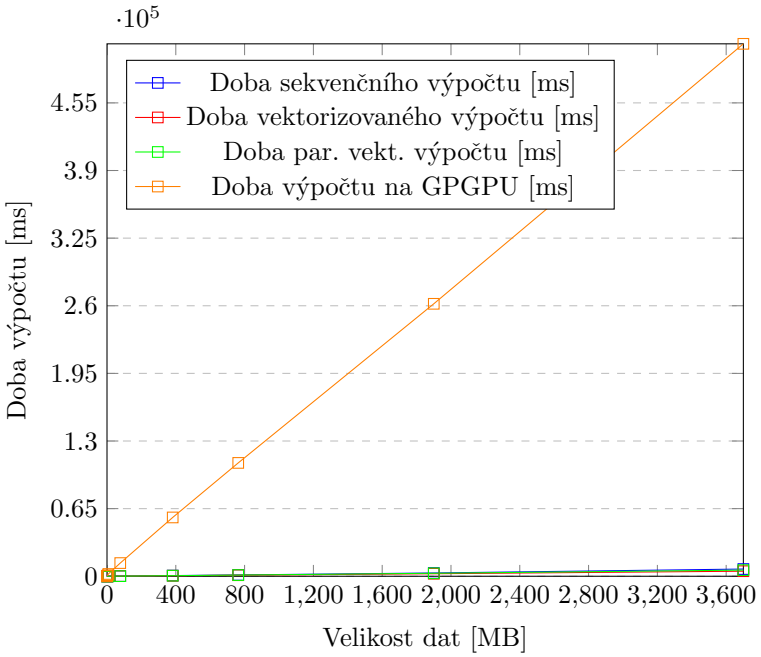
Welfordův/Chanův algoritmus je ve své podstatě navržen jako algoritmus iterativní. Ačkoli je využita jeho paralelní verze, algoritmus inherentně benefituje ze sekvenčního iterativního přístupu, který efektivně využívá vektorizovaný algoritmus v sekci 5.1.

Druhým problémem je zvolený přístup jako takový. OpenCL zařízení pracují nejefektivněji na velkém množství dat, nad kterým je možno využít co největší paralelizace jednoduchého výpočtu. Vhodnějším přístupem, než využití jediné *NDRange* k iteraci přes blok dat, by pravděpodobně bylo rozdělení bloku dat na několik *NDRange*. Každá by akumulovala mezivýsledky do paměti GPU, které by se na konci analýzy bloku nevracely na CPU (zde je vidět další neefektivnost zvoleného přístupu - časté přesuny paměti mezi GPU a CPU jsou časově náročné a neefektivní), nýbrž by sloužily jako základ pro další iterace. Na konci analýzy

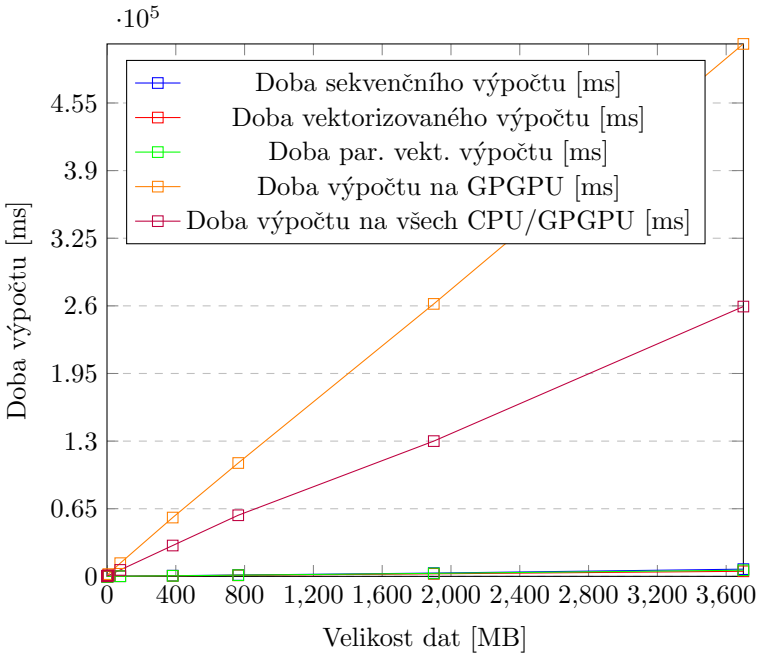


Obrázek 5.7: Vektorem dat je iterováno po blocích velikosti 256, mezivýsledky jsou poté postupně po polovinách redukovány do jediného výsledku.

celé datové sady by byl výsledek vrácen zpět na CPU. Tento přístup ovšem vyžaduje značné přepracování návrhu programu.



Obrázek 5.8: Srovnání dob sekvenčního, vektorového výpočtu v závislosti na velikosti vstupních dat.



Obrázek 5.9: Srovnání dob běhu všech algoritmů v závislosti na velikosti dat.

5.7 Kombinace CPU a GPGPU výpočtu

Pro spuštění výpočtu na obou platformách je třeba opět implementovat způsob distribuce práce těmto platformám. Jelikož je z předchozích výsledků zřejmé, že je výpočet na CPU několikanásobně efektivnější než výpočet na grafickém akcelerátoru, nabízí se implementovat tento mechanismus způsobem, který bude upřednostňovat CPU. Pro jednoduchost se ale omezíme na algoritmus *Round Robin*, který již byl využit pro distribuci práce mezi OpenCL zařízení v sekci 5.5.

Naměřené výsledky jsou uvedeny v grafu 5.9. Předáním části výpočtu na CPU bylo docíleno značného zrychlení oproti výpočtu čistě na *OpenCL* zařízení. Rychlost a efektivita výpočtu na CPU dokázala značně vybalancovat neefektivnost výpočtu na *GPGPU*. Při využití efektivnějšího algoritmu rozřazování práce by toto urychlení bylo ještě znatelnější. U malých datových sad lze naopak pozorovat velmi malé až zanedbatelné rozdíly mezi během na všech zařízeních a výpočtem čistě na *GPGPU*. To je pravděpodobně způsobeno nutností vyhledání *OpenCL* zařízení a překladem programu za běhu. U větších datových sad je tato zátež již neznatelná. Graf opět vyjadřuje značnou neefektivnost implementovaného algoritmu výpočtu na *GPGPU*.

Kapitola 6

Watchdog vlákno

Posledním bodem zadání je implementace tzv. *watchdog* vlákna. V praxi je pojmem *watchdog* často míněn hardwarový časovač, který periodicky inkrementuje interní čítač až do předem určeného maxima. Běžící vlákna pak periodicky tento čítač nastavují na 0 - tím dávají najevo, že nedošlo k zacyklení a výpočet korektně pokračuje dále. Pokud čítač dosáhne kritické hodnoty, program je watchdogem automaticky ukončen [1].

V této práci jsme ovšem omezeni pouze na softwarovou implementaci *wathcdog* vlákna. To s sebou inherentně přináší fakt, že taková implementace by v praxi nemohla být ověřitelná. I toto vlákno by totiž mělo být monitorováno, čehož lze docílit pouze odděleným hardwarovým časovačem. Softwarovou implementaci tedy navrhneme obdobně. Monitorovací vlákno bude obsahovat interní čítač. Ten nebude ovšem inkrementován automaticky samotným vláknem, ale bude periodicky inkrementován vlákny výpočtu o hodnotu n - velikost zpracovaného bloku. *Watchdog* se bude periodicky uspávat na dobu t (v implementaci *1000ms*). Po každém probuzení monitor zkontroluje, zda-li byl interní čítač během jeho spánku inkrementován. Pokud ne, je *watchdog* uspáván dále až na maximální počet šesti uspání. Pokud ani poté není čítač inkrementován oproti počáteční hodnotě, program je ukončen pro uvíznutí. Kontrolní smyčka je vizualizována v diagramu 6.1.

Obrázek 6.1: Vývojový diagram popisující implementaci Watchdog vlákna.

Kapitola 7

Zhodnocení

V této práci se podařilo navrhnout algoritmus, využívající výpočet čtyř centrálních momentů náhodného rozdělení k určení jeho kurtózy. Na základě toho bylo určeno, ke kterému ze zadaných rozdělení pravděpodobnosti mají vstupní data nejblíže. Nejprve byl navržen a implementován sekvenční, jedno-vláknový algoritmus. Následně se podařilo tento výpočet vektorizovat a dosáhnout značného urychlení.

Navržená paralelizace tohoto přístupu již další urychlení nepřinesla. Naopak, byla do výpočtu zanesena časová náročnost vytváření vláken tak, že uškodila celému výpočtu, který se tímto částečně zpomalil. Závěrem sekce 5.3 byl navržen alternativní přístup pro možné urychlení výpočtu.

Dále se povedlo navrhnout a implementovat modifikaci paralelního výpočtu, který využívá OpenCL kompatibilních *GPGPU* a grafických akcelerátorů. Ze zaznamenaných měření byla vypozorována značná neefektivita navrženého algoritmu a obecná nevhodnost využití grafických akcelerátorů pro použitý algoritmus výpočtu kurtózy. š Nakonec bylo do programu zavedeno kontrolní *watchdog* vlákno, které monitoruje běh výpočetních vláken a kontroluje jejich postup pomocí inkrementace čítače. Pokud *watchdog* vyhodnotí, že v programu došlo k uvíznutí či zacyklení výpočtu, ukončí celý program.

Literatura

[1] Barr, M.: Introduction to Watchdog Timers. [Online], [cit. 2023-07-01].
URL <https://www.embedded.com/introduction-to-watchdog-timers/>

[2] Pearson's chi-squared test. [Online], [cit. 2023-07-01].
URL <https://en.wikipedia.org/wiki/Pearson%27s_chi-squared_test>

[3] Cook, J.: Computing skewness and kurtosis in one pass. [Online], [cit. 2023-07-01].
URL <https://www.johndcook.com/blog/skewness_kurtosis/>

[4] cppreference.com: std::async. [Online], [cit. 2023-07-01].
URL <https://en.cppreference.com/w/cpp/thread/async>

[5] cppreference.com: std::launch. [Online], [cit. 2023-07-01].
URL <https://en.cppreference.com/w/cpp/thread/launch>

[6] Kurtosis. [Online], [cit. 2023-07-01].
URL <https://en.wikipedia.org/wiki/Kurtosis>

[7] Parallel algorithm. [Online], [cit. 2023-07-01].
URL <https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Parallel_algorithm>

[8] Higher order statistics. [Online], [cit. 2023-07-01].
URL <https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Higher-order_statistics>

[9] Welford's online algorithm. [Online], [cit. 2023-07-01].
URL <https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_online_algorithm>

Příloha A

Tabulky s měřeními

Doby výpočtu (ms)					
Velikost dat (MB)	Sekvenční	Vektorový	Paralelní	OCL	All
0.016	0	0	0	312	285
0.112	0	0	0	333	315
0.572	1	1	1	381	396
3.9	13	7	7	985	715
7.7	14	14	13	2164	829
77	135	125	114	12,676	6177
382	650	530	568	56,527	29,639
763	1273	1009	1143	10,8947	58,738
1900	3158	2432	2874	261,880	130,005
3700	6931	4869	5892	511,766	259,334

Tabulka A.1: Tabulované naměřené hodnoty všech průběžných a finálních implementací algoritmu.

Příloha B

Pseudokódy

```
void analyze(std::vector data) {
    double m_n = 0.0, m_M1 = 0.0, m_M2 = 0.0, m_M3 = 0.0, m_M4 = 0.0;

    for (const auto& x : data) {
        double delta, delta_n, delta_n2, term1;

        double n1 = m_n;
        m_n++;

        delta = x - m_M1;
        delta_n = delta / m_n;
        delta_n2 = delta_n * delta_n;
        term1 = delta * delta_n * n1;

        m_M1 += delta_n;
        m_M4 += term1 * delta_n2 * (m_n * m_n - 3 * m_n + 3)
            + 6 * delta_n2 * m_M2 - 4 * delta_n * m_M3;
        m_M3 += term1 * delta_n * (m_n - 2) - 3 * delta_n * m_M2;
        m_M2 += term1;
    }
}
```

Listing 2: Sekvenční výpočet čtvrtého momentu.

```
void analyze(vector numbers) {
    static const std::size_t count = 256;
    std::array<double, count> n{ 0 };
    std::array<double, count> M1{ 0 };
    std::array<double, count> M2{ 0 };
    std::array<double, count> M3{ 0 };
    std::array<double, count> M4{ 0 };

    for (std::size_t j = 0; (j + count) < numbers.size(); j += count) {

        for (std::size_t i = 0; i < count; ++i) {
            const auto n1 = n[i];
            const auto n2 = n1 + 1;
            // vektorizovany Welford
            const double delta = numbers[j + i] - M1[i];
            const double delta_n = delta / n2;
            const double delta_n2 = delta_n * delta_n;
            const double term1 = delta * delta_n * n1;

            M1[i] += delta_n;
            M4[i] += term1 * delta_n2 * (n2 * n2 - 3 * n2 + 3)
                + 6 * delta_n2 * M2[i] - 4 * delta_n * M3[i];
            M3[i] += term1 * delta_n * (n2 - 2) - 3 * delta_n * M2[i];
            M2[i] += term1;

            n[i] = n2;
        }
    }

    std::vector results(count);
    for (std::size_t i = 0; i < count; i++) {
        results[i] = { n[i], M1[i], M2[i], M3[i], M4[i] };
    }
    // redukce vsech mezivysledku pomoci Terriberryho rovnic
    result = std::reduce(results.cbegin(), results.cend());
}
```

Listing 3: Pseudokód vektorizovaného Welfordova algoritmu.

```
// projde vsechny platformy a vyhleda zarizeni s danym nazvem
void lookup_device(std::string device_name) {
  std::vector platforms = cl::Platform::get();
  for (auto& platform : platforms) {
    std::vector devices = platform.getAllDevices();
    for (auto&device : devices) {
      if (!(device.name() == device_name)) {
        continue;
      }
      // Zarizeni musi podporovat doublovou aritmetiku
      if (!device.supportsDoubles()) {
        std::cout <<
          "Zarizeni nepodporuje aritmetiku datoveho typu double";
        return;
      }

      return device;
    }
  }
}

// sestavi program pro zarizeni a vrati ho pro pozdejsi pouziti
// - spusteni algoritmu
void compile_program(cl::Device device) {
  std::string kernel_code = load_kernel_code();
  cl::Sources sources { kernel_code };
  cl::Program program { device, sources };

  try {
    program.build();
  } catch (e) {
    print_error(e);
  }

  return program;
}
```

Listing 4: Pseudokód vyhledání konkrétního OpenCL zařízení a sestavení programu kernelu.

Příloha C

Zdrojové kódy

```
#define LOCAL_GROUP_XDIM 256

__kernel __attribute__((reqd_work_group_size(LOCAL_GROUP_XDIM, 1, 1)))
void reduce_kernel(__global const double* numbers,
                   const double number_count,
                   __global double* g_n,
                   __global double* g_M1,
                   __global double* g_M2,
                   __global double* g_M3,
                   __global double* g_M4) {
    uint local_id = get_local_id(0);
    uint id = get_global_id(0);
    uint work_group_id = get_group_id(0);

    __local double n[LOCAL_GROUP_XDIM];
    __local double M1[LOCAL_GROUP_XDIM];
    __local double M2[LOCAL_GROUP_XDIM];
    __local double M3[LOCAL_GROUP_XDIM];
    __local double M4[LOCAL_GROUP_XDIM];

    n[local_id] = 0;
    M1[local_id] = 0;
    M2[local_id] = 0;
    M3[local_id] = 0;
    M4[local_id] = 0;

    n[local_id] = 1;
    M1[local_id] = numbers[id];
```

```
    uint other_id = id + LOCAL_GROUP_XDIM;
    while (other_id < number_count) {
        uint n1 = n[local_id];
        uint n2 = n1 + 1;

        double delta = numbers[other_id] - M1[local_id];
        double delta_n = delta / n2;
        double delta_n2 = delta_n * delta_n;
        double term1 = delta * delta_n * n1;

        M1[local_id] += delta_n;

        M4[local_id] += term1 * delta_n2 * (n2 * n2 - 3 * n2 + 3) +
          6 * delta_n2 * M2[local_id] - 4 * delta_n * M3[local_id];
        M3[local_id] += term1 * delta_n * (n2 - 2) - 3 *
          delta_n * M2[local_id];
        M2[local_id] += term1;
        n[local_id] = n2;

        other_id += LOCAL_GROUP_XDIM;
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    reduce_work_group(n, M1, M2, M3, M4, g_n, g_M1, g_M2, g_M3, g_M4);
}
```

Listing 5: Zdrojový kód Welfordova/Chanova algoritmu pro OpenCL kernel.

```
void reduce_work_group(double* local_n,
                       double* local_M1,
                       double* local_M2,
                       double* local_M3,
                       double* local_M4,
                       double* result_n,
                       double* result_M1,
                       double* result_M2,
                       double* result_M3,
                       double* result_M4) {
    uint local_id = get_local_id(0);
    uint id = get_global_id(0);
    uint work_group_id = get_group_id(0);

    uint dist = LOCAL_GROUP_XDIM;   // get_local_size(0)
    while (dist > 1) {
        dist >>= 1;
        if (local_id < dist) {
            uint other_id = local_id + dist;

            double prev_n = local_n[local_id];
            double new_n = prev_n + local_n[other_id];
            double prev_M2 = local_M2[local_id];
            double prev_M3 = local_M3[local_id];

            double delta = local_M1[local_id] - local_M1[other_id];
            double delta2 = delta * delta;
            double delta3 = delta * delta2;
            double delta4 = delta2 * delta2;
```

```
            local_M1[local_id] = (local_n[other_id] * local_M1[other_id] +
              prev_n * local_M1[local_id]) / new_n;
            local_M2[local_id] = local_M2[other_id] + local_M2[local_id] +
              delta2 * local_n[other_id] * prev_n / new_n;

            local_M3[local_id] = local_M3[other_id] + local_M3[local_id] +
                delta3 * local_n[other_id] * prev_n *
                (local_n[other_id] - prev_n) / (new_n * new_n);

            local_M3[local_id] += 3.0 * delta * (local_n[other_id] *
              prev_M2 - prev_n * local_M2[other_id]) / new_n;

            local_M4[local_id] = local_M4[other_id] + local_M4[local_id] +
              delta4 * local_n[other_id] * prev_n *
              (local_n[other_id] * local_n[other_id] - local_n[other_id] *
              prev_n + prev_n * prev_n) / (new_n * new_n * new_n);

            local_M4[local_id] += 6.0 * delta2 * (local_n[other_id] *
              local_n[other_id] * prev_M2 + prev_n * prev_n *
              local_M2[other_id]) / (new_n * new_n) + 4.0 * delta *
              (local_n[other_id] * prev_M3 - prev_n * local_M3[other_id]) /
              new_n;

            local_n[local_id] = new_n;
        }

        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if (local_id == 0) {
        result_n[work_group_id] = local_n[local_id];
        result_M1[work_group_id] = local_M1[local_id];
        result_M2[work_group_id] = local_M2[local_id];
        result_M3[work_group_id] = local_M3[local_id];
        result_M4[work_group_id] = local_M4[local_id];
    }
}
```

Listing 6: Zdrojový kód s paralelní redukcí pomocí Terriberryho rovnic pro OpenCL kernel.