Semestrální práce z KIV/PPR

Paralelizace výpočtu variačního koeficientu a mediánu absolutní odchylky

Dominik Zappe – A23N0011P
(zapped99@students.zcu.cz)

Prosinec 2024

Obsah

1 Zadání

Z **BIG IDEAs Lab Glycemic Variability and Wearable Device Data** si stáhněte datovou sadu subjektů, ze které si vytáhněte všechny hodnoty akcelerometrů - soubory `ACC*.csv`. Tím získáte 3 množiny hodnot (X, Y a Z) pro každého pacienta, a pro ně spočtěte koeficient variace a medián absolutní odchylky.

Výpočet budete realizovat pro:

1. sériový kód,
2. vektorizovaný kód,
3. vícevláknový, ale nevektorizovaný kód,
4. vícevláknový a vektorizovaný kód,
5. GPU kód.

Každý typ výpočtu spusťte 10x a do výsledku uveďte medián z naměřených časů vlastního výpočtu. Každá množina bude mít 74381 hodnot. Uvedené výpočty postupně spouštějte pro prvních 1000, 2000, 3000 až všechny hodnoty. Z naměřených a vypočtených hodnot pak sestrojte 3 grafy ve formátu `.svg`:

- časy výpočtů pro všechny typy výpočtů - uvidíte, jak se mění urychlení s rostoucí velikostí dat,
- hodnoty koeficientu variace - uvidíte, jak rychle bude konvergovat k výsledné hodnotě, a při správné implementaci by hodnoty měly být stejné pro všechny typy výpočtů,
- hodnoty mediánu absolutní odchylky - obdobně jako výše.

Rady a omezení

- Nepoužívejte algoritmy průběžného výpočtu mediánu, standardní odchylky apod. Pokud přesto ano, pak to přidejte jako extra implementaci navíc.
- Sériový kód a kód OpenCL může mít hodně sdíleného - můžete si tak vyzkoušet pohodlný způsob ladění OpenCL kódu.
- Vektorizaci provádějte manuálně pro AVX2 double - inkludujte `immintrin.h`, v nastavení projektu zvolte AVX2, abyste měli správné zarovnání struktur v paměti.
- Paralelizaci můžete provést pomocí `std::for_each` a `std::execution::par_unseq`.

- Při výpočtu na GPU spočítejte redukční operace také na GPU, ale můžete to udělat pomocí několika kernelů.
- Pokud si správně napíšete prototyp vektorizovaného a sériového výpočtu, popř. pár wrapperů, můžete je rovnou volat z `std::for_each` jen změnou parametrů (tj. výpočetní funkce a execution policy). Tak si na pár řádcích realizujete všechny 4 varianty výpočtu na CPU.
- SYCL by měl umožnit udělat totéž i pro variantu GPU.

Informace o samostatné práci

Samostatná práce využije alespoň dvě ze celkem tří možných technologií:

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

Pokyny pro zpracování

Co musí být splněno pro základní bodování do 15 bodů:

- Program musí jít přeložit pomocí souboru `checker.exe` s konfigurací v souboru `checker.ini`, tak, jak ho dodal cvičící - je-li to pro daný rok relevantní.
- Opravné odevzdání musí obsahovat popis toho, co se změnilo.
- Zdrojové kódy budou přeložitelné bez warningů (u překladače jazyka C/C++ přepínač `-Wall`, u ostatních ekvivalentním způsobem).
- V referátu musí být popsán použitý algoritmus a způsob jeho implementace, případné zdůvodnění postupu, zdůvodnění strukturování programu, uživatelský "minimanuál"(tj. jak se překládá ze zdrojových souborů, požadavky na výpočetní prostředí, jak se ovládá ap.), v závěru pak (sebekritické) hodnocení vykonané práce ap.
- V příloze referátu může být výpis vytvořených programů (není nutný, ale hodí se u zkoušky - alespoň nějaké "zajímavé"pasáže).

- Zdrojové kódy by měly být dobře čitelné (samovysvětlující identifikátory, strukturování na funkce, strukturované konstrukce uvnitř funkcí, komentáře).
- Data budou dělena mezi procesy dynamicky (LOAD-BALANCE), tj. model farmer-worker a rozdělení celé práce na malé kousky. Pokud ne (např. je to nevhodné pro zadaný typ úlohy, byly použity globální operace v MPI...), tak řešení správně popsat a odůvodnit v dokumentaci.
- Vytvořené programy musí být správné, tj. pro zadaná vstupní data musí dát (vždy) správný výsledek.
- Správnost programu bude testována na referenčních strojích.
- Není-li určeno zadáním, pak pro základní bodování stačí konzolový I/O interface, tj. zadávání vstupních dat z klávesnice a výpisy výsledků na displej ve stylu řádkového terminálu.
- Analýza výkonnosti programu a pokus o její zlepšení, včetně podrobného zhodnocení dosažených výsledků.

Co se dále hodnotí:

- Statická analýza kódu provedená Visual Studiem 2019 při překladu v release mode.
- Statický analyzátor má vždy pravdu, protože máte vždy možnost napsat kód čitelně tak, aby negeneroval false-positive.
- Názvy proměnných a funkcí jsou oddělovány podtržítky, dále viz JSF coding standard.
- Úroveň DTP (berte zpracování referátu jako trénink na zpracování diplomky).
- Modulární strukturování programu: Snažte se vytvořit univerzálněji využitelné funkce (nebo třídy) jako samostatný modul (knihovnu). Funkčnost vytvořeného modulu demonstrujte prostřednictvím hlavního programu aplikace na vzorových vstupních datech (použijte několik různých množin vstupních dat). Hlavní program by měl číst vstupní data ze souboru nebo umožňovat jejich nastavení prostřednictvím klávesnice, dále by pro realizaci paralelního výpočtu volal funkce (instaloval objekty), a nakonec (po ukončení paralelního výpočtu) zobrazoval výsledky.
- Řešení I/O a způsob (přehlednost) zobrazení výsledků.
- Rozšíření zadání tak, že program funguje i pro jinou třídu (příbuzných úloh).
- Vše, co se ještě bude vedoucímu cvičení líbit.

2 Návrh z průběžné prezentace

V průběžné prezentaci bylo navrženo řešení pomocí standardního výpočtu mediánu absolutní odchylky:

$$MAD(X) = median(|x_i - median(X)|)$$

a pro výpočet variačního koeficientu byl zvolen výpočetní tvar pro výpočet standardní odchylky:

$$CV(X) = \frac{\sigma}{\mu} = \frac{\sqrt{Var(X)}}{E(X)} = \frac{\sqrt{E((X-\mu)^2)}}{E(X)} = \frac{\sqrt{E(X^2) - E^2(X)}}{E(X)}$$

(v prezentaci bylo taktéž upozorněno na možnou numerickou nestabilitu výpočetního tvaru standardní odchylky – pro dodaná data to není problém).

Dále byl v prezentaci zmíněn návrh datové struktury pro uchování dat – tři vektory – X, Y, Z.

V neposlední řadě byly v návrhu zmíněny dvě možné optimalizace:

- Při řazení je možné rovnou nasčítat sumy (u *merge sortu* např. při posledním spojování pod-polí – iteruje se přes všechny prvky)
- Při výpočtu MAD není nutno řadit dvakrát – odečtením mediánu od seřazeného pole se z mediánu stává 0, pole je stále seřazené. Záporná část (vlevo od původního mediánu (nyní 0)) se překlopí do kladných čísel, ale tato posloupnost je stále seřazená – jen pozpátku. Tento problém vede na bitonické spojení dvou bitonicky seřazených posloupností.

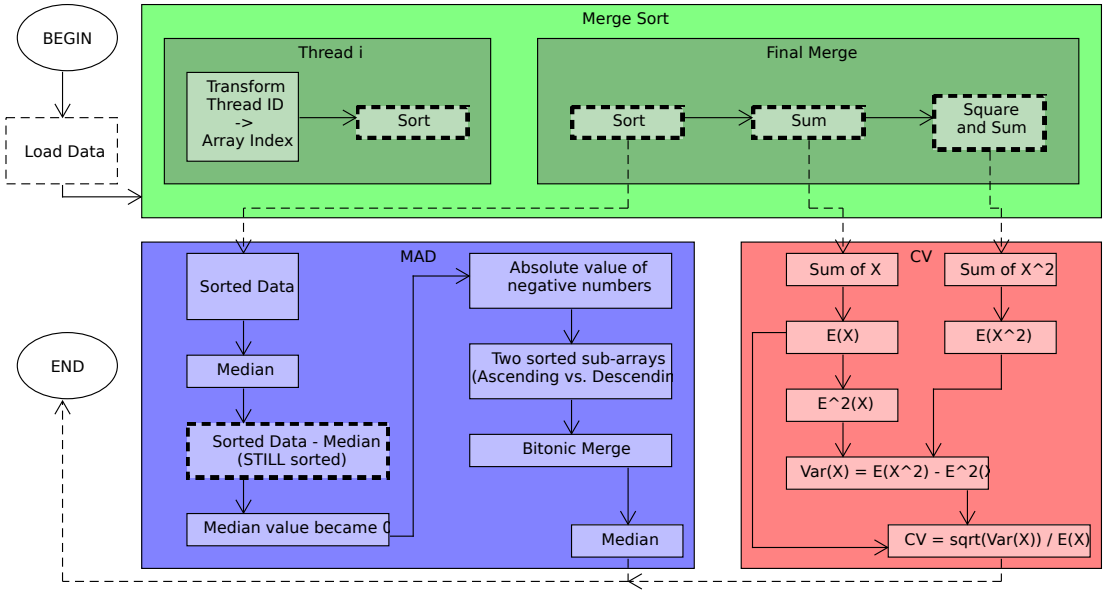
Na následujícím Obrázku 1 je vidět Control/Data-flow diagram návrhu z prezentace. Přerušovaně (čárkovaně) jsou označeny paralelizovatelné úseky; tučně jsou označeny vektorizovatelné úseky.

2.1 Změny návrhu

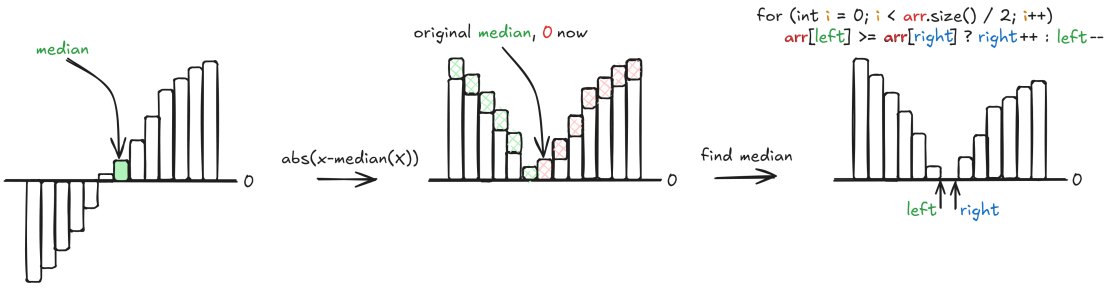
Jedinou zásadní změnou oproti návrhu z průběžné prezentace je nevyužití možnosti nasčítat sumy při posledním průchodu polem při jeho řazení. Při implementaci manuální vektorizace bylo toto rozhodnutí přehodnoceno, neboť kód se stal zbytečně složitým a nečitelným. Tato skutečnost pouze v Obrázku 1 vysouvá rámečky „Sum“ a „Square and Sum“ ze zelené oblasti do červené.

Druhou méně zásadní změnou je uvědomění si, že vyloučené druhé řazení – původně řešené bitonickým spojením dvou bitonických posloupností – lze řešit ještě jednodušeji. Není vůbec

nutné posloupnosti spojovat, ale lze užít dvou ukazatelů – levý a pravý – a postupně s nimi šoupat přesně $\frac{n}{2}$ krát (n = velikost pole), vyobrazeno na Obrázku 2.



Obrázek 1: Původní Control/Data-flow diagram návrhu



Obrázek 2: Nalezení druhého mediánu při výpočtu mediánu absolutní odchylky

3 Implementace

Celá semestrální práce je implementována v jazyce C++ ve standardu C++17. Kromě dovolených technologií nebylo využito žádné externí knihovny (pouze u grafického vykreslení bylo využito doporučeného SVG vykreslovače z `https://github.com/SmartCGMS/common.git`)

3.1 Načítání dat

Při načítaní dat bylo zjištěno, že standardní *file streamy* jazyka C++ jsou pomalejší než staré C konstrukce pro práci se soubory. Nejspíše za to může „zbytečný OOP overhead“. Měřené a zmiňované časy se týkají souboru *ACC_001.csv*.

Při standardním načtení použitím *std::ifstream* a *std::getline* trvá načtení v průměru 45 sekund.

Při přepsání do standardních bezpečných konstrukcí do starších, méně bezpečných, C konstrukcí dojde k urychlení na průměrné načtení za cca 20 sekund. Přepsáno bylo hlavně *std::ifstream* na *FILE ** a *fopen()*, *std::getline* na *fscanf()*.

K dalšímu urychlení došlo při jiném přístupu k vektorům uvnitř struktury pro pacientova data. Namísto vkládání do dynamického pole se nejdříve vektor zvětší na empiricky odhadnutou velikost přibližně jednoho milionu – přesně 2^{20} (tedy $1 << 20$). Díky této změně velikosti (opravdu *.resize()*, nikoliv *.reserve()*) je později možné pole přímo indexovat namísto vkládání pomocí *.push_back()*, resp. *.emplace_back()*. Další urychlení je načítání řádek pomocí *fgets()* a ruční parsování pomocí *std::strtod()*. Pole je v průběhu parsování dynamicky zvětšováno na dvojnásobek své původní velikosti – vede k amortizaci. Tato urychlení dosahují načtení za průměrných 8 sekund.

Závěrečné urychlení zahrnuje načtení souboru jednorázově do RAM a následné paralelní zpracování jeho řádek. Zde by mohl nastat problém s nedostatkem paměti, ale dovolil jsem si toto udělat, neboť největší vstupní soubor má 966 MB. Jelikož je pak soubor načten v paměti RAM, dá se jednoduše rozdělit na pole řádek. Nad řádky se již dá jednoduše pracovat více vláknově, navíc je předem známo, kolik řádek bylo načteno – je možné vektory ve struktuře pacientských dat zvětšit na přesnou velikost počtu řádek a dále již jednoduše indexovat. Indexy řádek se rozdělí na kusy podle počtu dostupných jader procesoru. Každé vlákno pak zpracovává daný interval řádek, které jsou vzájemně disjunktní – nemůže dojít k souběhu a není nutno synchronizovat. Tato závěrečná urychlení dosahují načtení souboru sériově za průměrně 4.5 sekundy, paralelně za průměrně 2.1 sekundy.

V kódu byly všechny zmíněné funkce ponechány – pro názornost vývoje. Jedinou využitou funkcí je však ta poslední – paralelní s využitím RAM. Zmiňované funkce jsou implementovány v */src/dataloader/dataloader.h* a */src/dataloader/dataloader.cpp*.

3.2 Výpočty

Výpočetní zdrojové kódy jsou hierarchicky rozdělené do podadresářů */src/calculations/cpu/* a */src/calculations/gpu/*. Ve společném adresáři */src/calculations/* je k nalezení třída *computations*, která slouží jako *rozhraní* (spíše *abstraktní třída*) pro budoucí statický polymorfismus. Třída definuje dvě funkce – každá pro jeden výpočet – *compute_mad()* a *compute_coef_var()*.

Ve funkci *compute_mad()* nejprve dochází k seřazení vstupního pole (zde využit statický polymorfismus). Dále je jednoduše vybrán medián ze seřazeného pole. Následuje výpočet absolutní hodnoty z rozdílu ve formě funkce, kterou opět bude definovat potomek (zde opět statický polymorfismus). Následně je vybrán z výsledného pole medián pomocí optimalizace zmiňované v Sekci 2.1.

Funkce *compute_coef_var()* prakticky pouze provolává funkci budoucích potomků pro výpočet sumy elementů vstupního pole (naráz také suma čtverců elementů vstupního pole) (zde je opět užito statického polymorfismu). Následně je užit vzoreček pro výpočet variačního koeficientu z vypočtených sum.

Díky zmíněnému statickému polymorfismu je pak možné si někdy na začátku běhu programu vybrat konkrétní výpočet z dostupných možností – *sériový sekvenční*, *sériový vektorizovaný*, *paralelní sekvenční*, *paralelní vektorizovaný* a *GPU*. Není tedy nutné pro každý výpočet zvlášť mít ve funkcích nějaké podmínky, které by za běhu rozhodovaly vícekrát o způsobu výpočtu. Rozhodne se pouze jednou a to na začátku programu pomocí návrhového vzoru *visitor* užitím *std::variant*.

Přes statický polymorfismus je vyřešen pouze rozdíl mezi *sekvenčním*, *vektorizovaným* a *GPU* výpočtem. Rozdíl mezi *sériovým* a *paralelním* během je vyřešen mnohem elegantněji a jednodušeji – v implementaci se prakticky pouze jedná o záměnu *std::execution::seq* za *std::execution::par*. Bohužel tyto třídy ze standardní knihovny nemají žádného předka, od kterého by dědily, bylo tedy nutné tuto skutečnost vyřešit přes *šablony* a opět *std::variant* spolu s návrhovým vzorem *visitor*.

Ve finále je nutné implementovat tři funkce pro každou variantu – *sort*, *výpočet absolutní hodnoty rozdílu* a *suma elementů pole (+ suma čtverců)*.

Automaticky se pro výpočty předpokládá plovoucí řádka s přesností 8-byte (*double*). Překladači lze však předat příznak „D_USE_FLOAT“, pak bude pro výpočty použita přesnost na 4-byte (*float*). Tato skutečnost je k vidění v */src/utils/utils.h*.

3.2.1 CPU

Zdrojové kódy jsou k nalezení v */src/calculations/cpu/merge_sort.** a */src/calculations/cpu/cpu_comps.**.

MAD

Pro tento výpočet je zapotřebí jak řazení, tak výpočet absolutní hodnoty rozdílu.

Algoritmem řazení byl zvolen *merge sort*. Při přístupu „zdola nahoru“ je algoritmus vhodný pro paralelizaci – iterování vstupním polem s exponenciálně rostoucím *stridem* – nemůže ani dojít k souběhu. Vnitřní smyčka je paralelizovatelná, uvnitř níž dochází k výpočtu levé a pravé hranice – v rámci těchto hranic dochází k řazení a spojování pod-polí. Manuální vektorizace v rámci řazení nebyla vůbec využita, neboť *merge sort* není vhodným algoritmem pro vektorizaci.

Výpočet absolutní hodnoty rozdílu je v rámci semestrální práce úplně nejjednodušším, neboť zde již z definice výpočtu nemůže dojít k souběhu – paralelizace je zde prakticky zadarmo. Při sekvenčním výpočtu se pouze jedná o jeden *std::for_each()*, který buď bude sériový, nebo paralelní (záleží na volbě uživatele). U vektorizované verze je výpočet zajímavější – nejprve se načte vstupní medián do AVX2 registru. Následně se v *std::for_each()* načítají čtveřice *doublů* (resp. osmice *floatů*) do AVX2 vektorů. Následně je užito vektorového rozdílu. Absolutní hodnota je vypočtená následujícím „trikem“ – nejprve se vytvoří záporné znaménko pomocí *_mm256_set1_pd(-0.0)*, následně je na rozdíl tato „maska“ aplikována pomocí *AND* a *NOT* – *_mm256_andnot_pd(sign_bit, diff_vec)* – tím se efektivně provede absolutní hodnota.

CV

Pro tento výpočet je zapotřebí pouze výpočet sumy elementů a sumy čtverců elementů pole.

Výpočet těchto dvou sum je vyřešen v rámci jednoho průchodu polem – není nutné mít dva *std::for_each()*. Vstupní pole je rozděleno do kusů podle počtu dostupných vláken. Tyto kusy jsou pak procházeny paralelně – to ještě nezajišťuje nemožnost dojít k souběhu. Za předpokladu, že by všechna vlákna přičítala do společné sdílené proměnné, docházelo by k souběhu. Je proto nutné si před smyčkou definovat pole lokálních sum (pole má velikost podle počtu vláken). Pak již k souběhu dojít nemůže, každé vlákno si přičítá do své lokální sumy. Po paralelním nasčítání v rámci kusů je ještě zapotřebí sečíst lokální sumy do jedné globální (to je již záležitost jednoho vlákna). U vektorizované verze tohoto výpočtu je hlavním rozdílem nasčítávání po čtveřicích (resp. osmicích pro *float*). Hlavní věcí je zde zaručit, že jednotlivé kusy jsou velikosti násobku čtyř (resp. osmi). Mimo užití AVX2 funkcí je vektorizovaný součet shodný se sekvenční verzí.

3.2.2 GPU

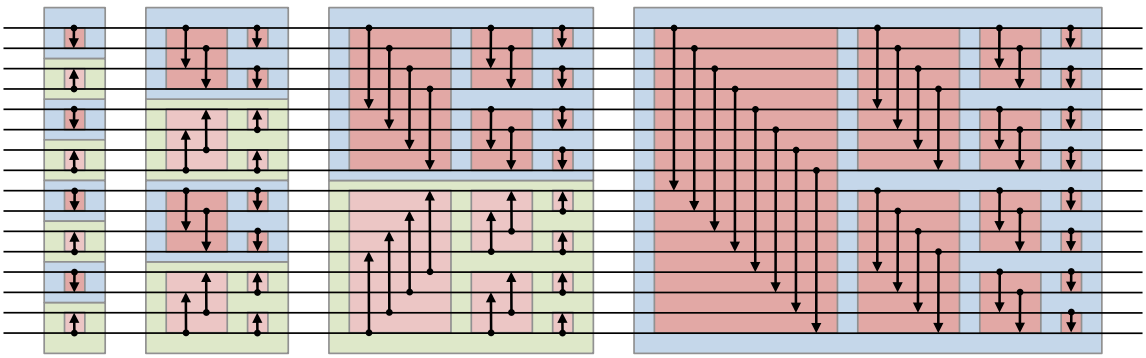
Zdrojové kódy jsou v tomto případě rozděleny do „dvou“ – *host side code* a *kernel code* – tedy kód pro naplánování výpočtů na *GPU* a *kernel* (kód pro *GPU*) samotný. Oba zdrojové kódy lze nalézt v adresáři */src/calculations/gpu/**.

Aby byla zajištěna konzistentnost s příznakem překladače „D_USE_FLOAT“, jsou veškeré *kernely* napsané „dvojím způsobem“ přes podmínky preprocesoru – viz */src/computations/gpu/gpu.h*.

MAD

Pro tento výpočet je zapotřebí jak řazení, tak výpočet absolutní hodnoty rozdílu.

Pro dobré porovnání s *CPU* výpočty byl nejprve algoritmem řazení zvolen *merge sort*. Tento způsob řazení však není příliš pro *GPU* vhodný. Tento výpočet byl největším úzkým hrdlem celé práce – pro zlepšení *GPU* výpočtů byl tedy implementován *bitonic sort* přes tzv. *sorting networks*. Přístup přes *sorting networks* je na GPU ideální, neboť se jedná o masivně paralelizovatelný přístup k řazení. Na straně hostitele je však nejprve nutné zajistit velikost pole jako mocninu dvou – *padding*. Dále je nutné iterovat přes tzv. *stages* a *passes of stages* – zde se již plánuje samotný kernel. Globální velikost je vždy velikost *paddovaného* pole dělená dvěma. Na straně *GPU* – v kernelu – se pak složitým postupem musí zvolit vhodná dvojice prvků k porovnání – tento výběr probíhá na základě následujícího schématu na Obrázku 3. Tímto je výpočet na *GPU* také optimální – nelze tedy v budoucnu porovnávat *CPU* a *GPU* co se týče algoritmů, ale lze porovnávat dosažené časy jednotlivých platforem, neboť obě mají „optimální“ implementaci (v rámci možností).



Obrázek 3: *Sorting network* s viditelnými *stages* (červené bloky) a *passes* (modré a zelené bloky) a směry (šipky) – převzato z `https://en.wikipedia.org/wiki/Bitonic_sorter`

Výpočet absolutní hodnoty rozdílu je opět nejjednodušší částí výpočtů. Kernel přebírá dvě pole – vstupní, výstupní – a číslo k odečtení (medián). Každé vlákno na základě svého globálního indexu pouze odečte medián od vstupního prvku, který odpovídá indexu vlákna, a zapíše výsledek po provedení absolutní hodnoty na příslušný index výstupního pole.

CV

Pro tento výpočet je zapotřebí pouze výpočet sumy elementů a sumy čtverců elementů pole.

Výpočet sum je opět řešen v rámci jednoho průchodu polem. Řešení využívá lokální velikosti (zvoleno 256) – tolik prvků mají lokální pole mezivýsledků. Jednotlivé *work group* nesčítají všechny prvky pole, ale pouze „lokální“ část původního pole. Pro redukční sumu je zvolen přístup „shora dolů“ – tedy logaritmicky zmenšující se *stride* – tím je zajištěn lokální přístup k poli. Pole mezivýsledků je ve finále zpracováno na *CPU* – součet 256 čísel.

3.3 Vykreslení

Na základě domluvy s vyučujícím bylo pro implementaci vykreslení výsledných grafů užito SVG vykreslovače z `https://github.com/SmartCGMS/common.git`. Vlastní vykreslovač grafů je implementován v */src/my_drawing/svg_generator.**. Funkce vykreslující grafy využívají nabízená grafická primitiva z výše zmíněného SVG vykreslovače.

3.4 Ošetření uživatelských vstupů

První možností uživatelského vstupu je příznak překladače „D_USE_FLOAT“ – přepínač mezi *double* a *single* přesností (v kódu užití *double* a *float* datových typů). Definice tohoto příznaku je v */src/utils/utils.h*.

Zbylé uživatelské vstupy jsou možné přes příkazovou řádku nad již přeloženým programem – ošetření těchto vstupů zajišťuje třída *arg_parser* implementovaná v */src/utils/arg_parser.**. Zde je zjevná inspirace v parseru argumentů z Pythonu – *argparse*. Program očekává nutně jeden z následujících vstupů:

- -f ⟨soubor⟩ – definice vstupního souboru.
- -d ⟨adresář⟩ – definice adresáře, který obsahuje vstupní soubory.

Tímto způsobem program ví, odkud čerpat data. Lze tedy zpracovat soubor jeden, případně celý adresář plný datových souborů (v očekávaném formátu ze zadání). Příznaky jsou navzájem výlučné, ale právě jeden je povinný.

Další možné argumenty z příkazové řádky už nejsou povinné; vypadají následovně:

- -r ⟨počet opakování experimentu⟩ – ze všech experimentů se do výsledného grafu propisuje *medián* výsledků (pro splnění zadání nutno spustit s „-r 10“).
- -n ⟨počet „batchů“ dat⟩ – počet kusů, do kolika se vstupní data mají roztrhat, nad nimiž se bude r-krát provádět daný výpočet. Prakticky se jedná o granularitu osy X ve výstupních grafech.
- --par – za tímto příznakem se neočekává žádná hodnota. Jedná se o pouhý přepínač mezi *sériovým* a *paralelním* výpočtem.
- --vec – za tímto příznakem se opět neočekává hodnota. Jedná se opět o přepínač, tentokrát mezi *sekvenčním* a *vektorizovaným* výpočtem.
- --gpu – opět se jedná o přepínač bez hodnoty – přepíná se mezi *CPU* a *GPU* výpočtem.
- --all – výše popsané tři přepínače umožňují spustit pouze jeden typ výpočtu nad daným datovým souborem (opět se jedná o přepínač neočekávající hodnotu). Pro splnění zadání ještě existuje tento přepínač, díky kterému budou iterativně provedeny všechny kombinace typů výpočtů. Tento přepínač mění grafický výstup na grafy s pěti čarami – každá křivka odpovídá jinému typu výpočtu (pokud je program spuštěn v režimu jednoho konkrétního typu výpočtu grafy obsahují tři křivky – čára pro každý sloupec vstupních dat – X, Y a Z).
- –no_graphs – za tímto příznakem opět není očekávána žádná hodnota. Tento přepínač umožňuje zamezit generování obrázků na konci běhu programu (výhodné hlavně při vývoji pro ladění).
- -h – vypsání nápovědy.
- --help – vypsání nápovědy.

4 Výsledky a diskuze

4.1 Načítání dat

Jak již bylo zmíněno v Sekci 3.1, měřen byl i čas načtení dat. Funkce pro načtení dat také přebírá *execution policy* a je tedy možné data načítat sériově a paralelně. Pro názornost urychlení paralelním načtením byl proveden experiment na „nejrychlejší“ funkci – 30 opakování pro každý typ načtení – paralelní / sériový. Výsledné časy jsou vidět v Tabulce 1. Všechny hodnoty v Tabulce 3.1 jsou v milisekundách – zaokrouhleno na jedno desetinné místo. Z Tabulky 3.1 je zjevné, že paralelní načtení je více než dvakrát rychlejší oproti svému sériovému protějšku.

	Medián (ms)	**Průměr (ms)**	**Standardní odchylka (ms)**
Sériové načtení	4478.0	4563.7	193.2
Paralelní načtení	1988.5	1986.7	45.0

Tabulka 1: Porovnání sériového a paralelního načtení dat

4.2 Výpočty

Hlavním experimentem v rámci semestrální práce bylo měření času trvání jednotlivých typů výpočtů – *Sériové*/*Paralelní* × *Sekvenční*/*Vektorizované* + *GPU*. V následujících grafech není započtený výše zmiňovaný čas pro načtení dat – jedná se o čisté časy trvání jednotlivých výpočtů. Program byl spuštěn s následujícími příznaky:

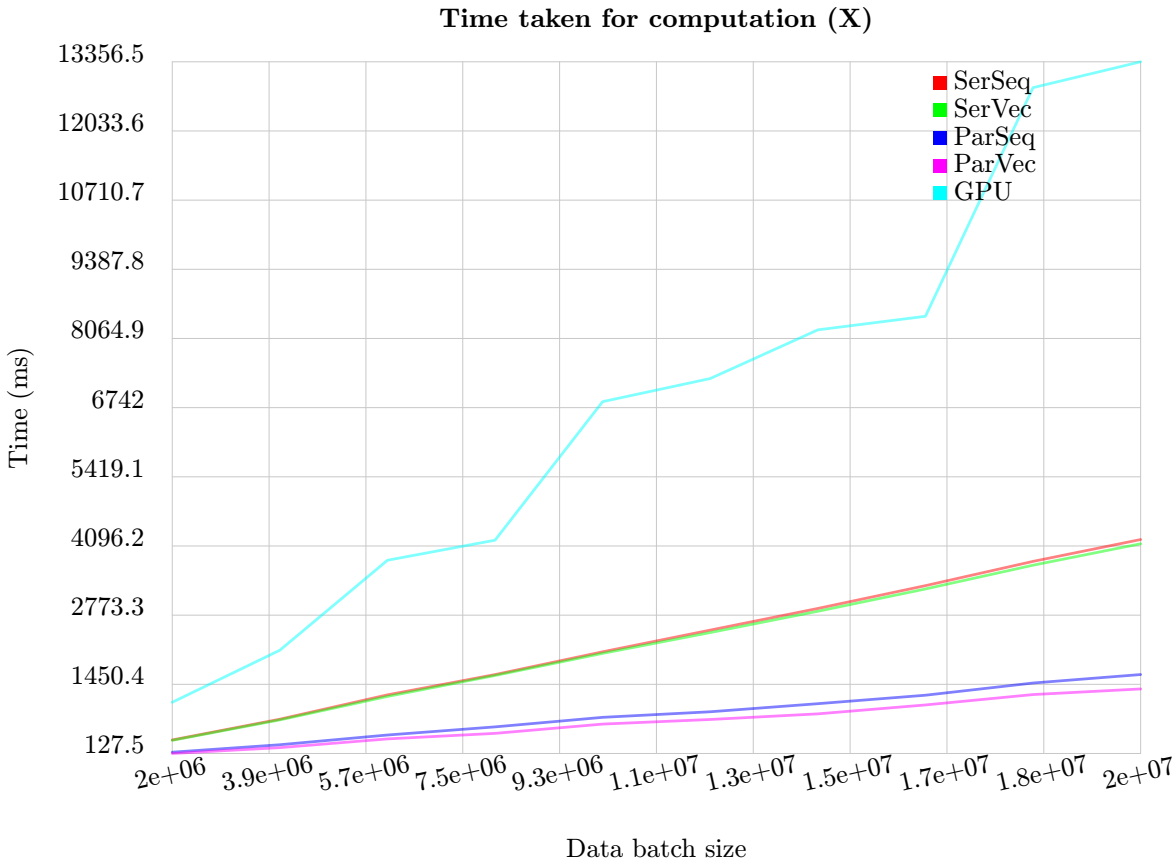
-d data --all -r 10 -n 10

Jelikož by se sem všechny grafy nevešly, jsou zobrazeny grafy pouze pro soubor *ACC_001.csv* (sloupec X) – tedy pro replikaci spuštění s argumenty:

-f data/ACC_001.csv --all -r 10 -n 10

Křivky na následujících grafech jsou konstruovány z *mediánu* opakování (-r), granularita osy X je určena počtem využitých prvků pro dané výpočty (-n).

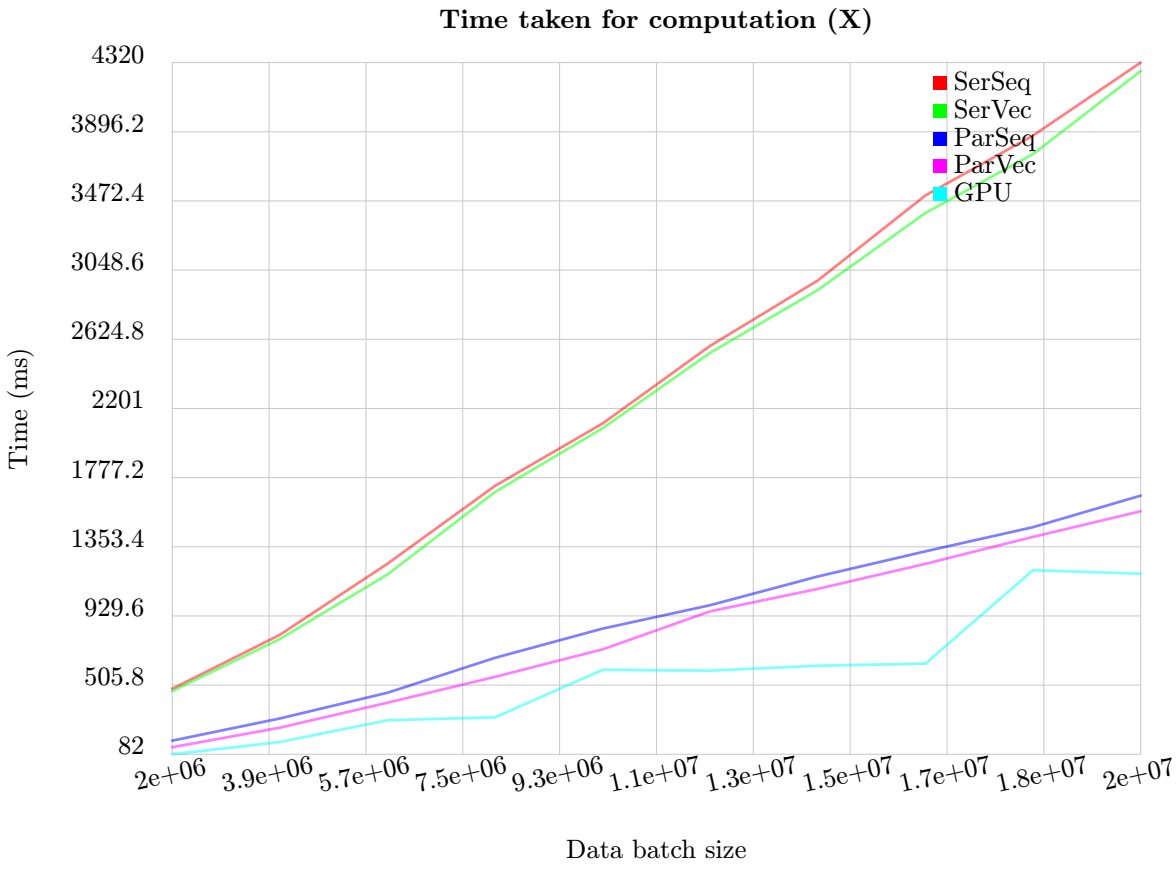
4.2.1 Porovnání jednotlivých typů výpočtů



Obrázek 4: Původní naměřené časy s *merge sortem* na *GPU* pro jednotlivé metody výpočtů na *ACC_001.csv*, sloupec *X*

Na Obrázku 4 jsou vidět původní naměřené časy (v milisekundách, osa Y) pro jednotlivé kusy dat (osa X). Vykreslené křivky reprezentují jednotlivé metody výpočtů. Tento obrázek pochází z doby, kdy na *GPU* byl implementován *merge sort*. Je vidět, že největší rozdíl byl obecně mezi *CPU* a *GPU* – to je právě způsobeno volbou špatného řadicího algoritmu.

Na Obrázku 5 jsou vidět finální naměřené časy pro jednotlivé metody výpočtů. Je vidět, že změna řadícího algoritmu na *GPU* najednou z *GPU* výpočtů udělala ty nejrychlejší (původně nejpomalejší). Z obrázku je krásně vidět „schodovitý“ časový průběh pro *GPU* výpočty, protože nejvíce trvající částí je řazení – *bitonic sort*, který pro svůj běh potřebuje vstupní data velikosti mocniny dvou. Tedy co *schod*, to překročení nové mocniny dvou.

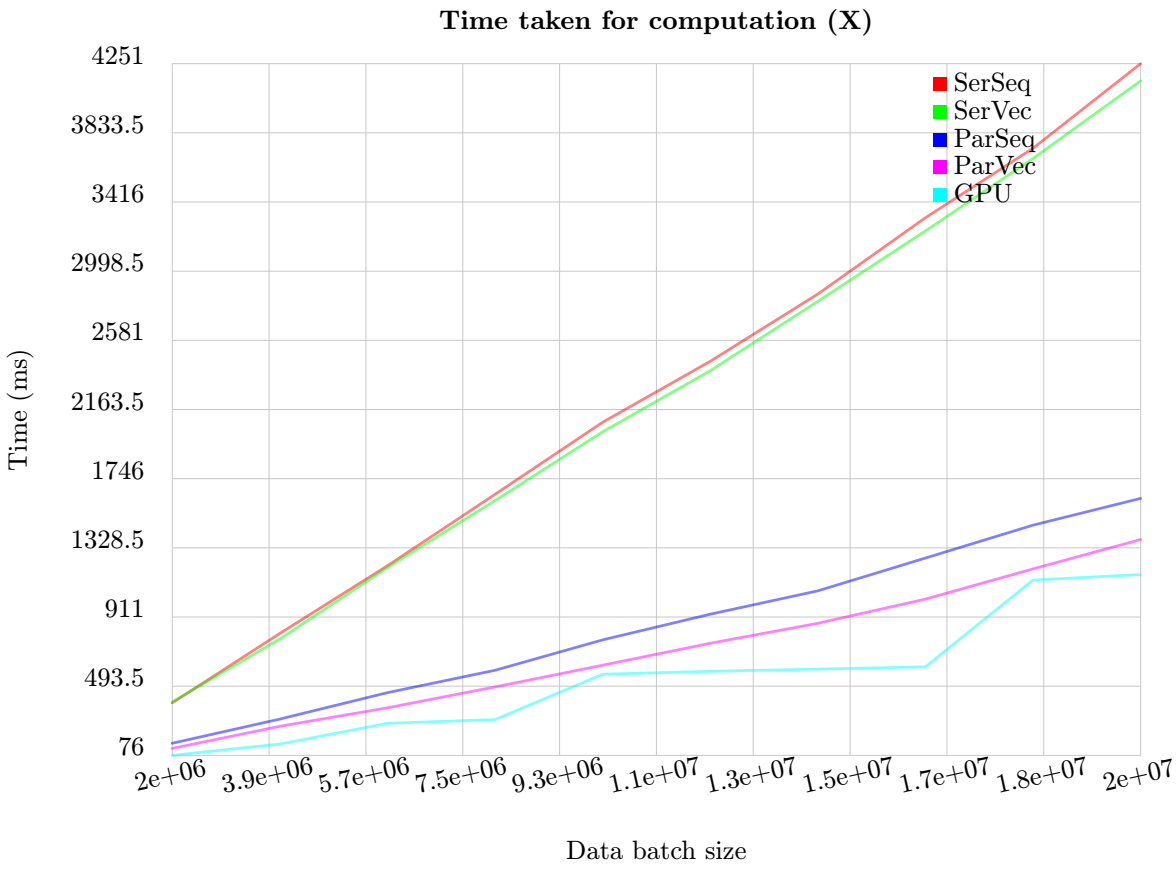


Obrázek 5: Naměřené časy pro jednotlivé metody výpočtů na *ACC_001.csv*, sloupec *X*

Velký skok je dále vidět mezi *sériovým* a *paralelním* během – to bude hlavně tím, že *merge sort* je paralelizován a jedná se o největší úzké hrdlo celého *CPU* kódu. Jelikož se *merge sort* nedá dobře vektorizovat, nejsou patrné příliš velké rozdíly mezi *sekvenčním* a *vektorizovaným* během. Vektorizované jsou pouze výpočty sum a absolutních hodnot rozdílů – opět je ale nejdéle trvající částí kódu *merge sort*, který není vektorizován.

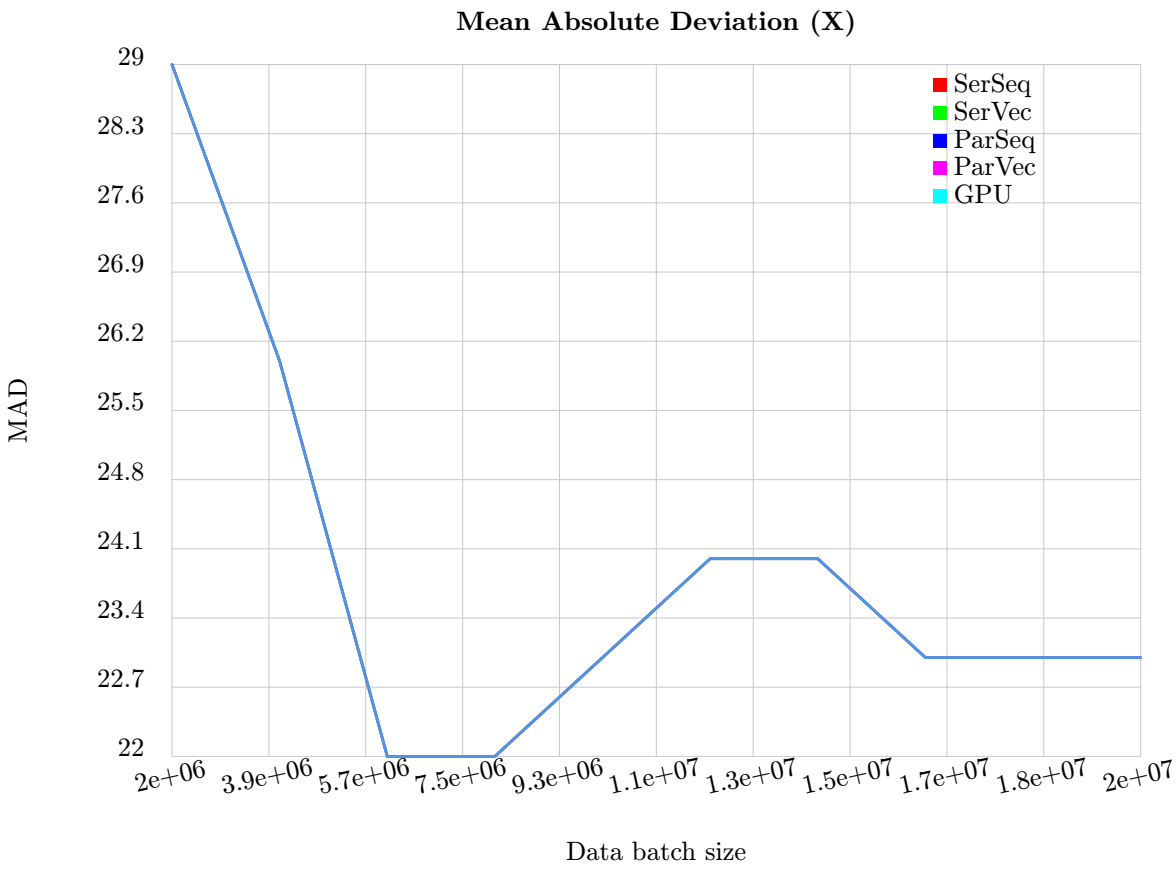
Příčinou malého rozdílu vektorizovaného kódu oproti sekvenčnímu je především auto-vektorizace, která je v nové verzi MSVC 2022 automaticky zapnuta. Bohužel se v rámci semestrální práce nepodařilo vypnout auto-vektorizaci pomocí *pragma* (z CourseWare), protože ta je aplikovatelná pouze na obyčejné *for* smyčky – nikoliv na *std::for_each()*. Pro tyto smyčky je však auto-vektorizace přímo řízená předávanou *execution policy* – je zajištěno, aby byla vždy vypnuta. Napříč tomu však byly vytvořeny grafy, kde je celkově vypnuta optimalizace levelu 3 (*O3*) – byl využit přepínač překladače *O2* – viz Obrázek 6. Hlavně mezi *modrou* a *fialovou* křivkou je vidět větší mezera – to může být zapříčeněno auto-vektorizací kódu mimo *std::for_each()* Lze očekávat, že při

implementaci vektorizovatelného řazení by měly křivky *sekvenčních* a *vektorizovaných* běhů mít mezi sebou větší mezeru – vektorizace by měla urychlovat kód více, než je to vidět z obrázků. Další možností malé efektivity vektorizace by mohlo být využití vektorových funkcí na nezarovnanou paměť (načítání a zápis). V rámci práce nebyl implementován vlastní alokátor pro práci s *std::vector*, který je ze základu paměťově nezarovnaný, neboť uložená data jsou na haldě.

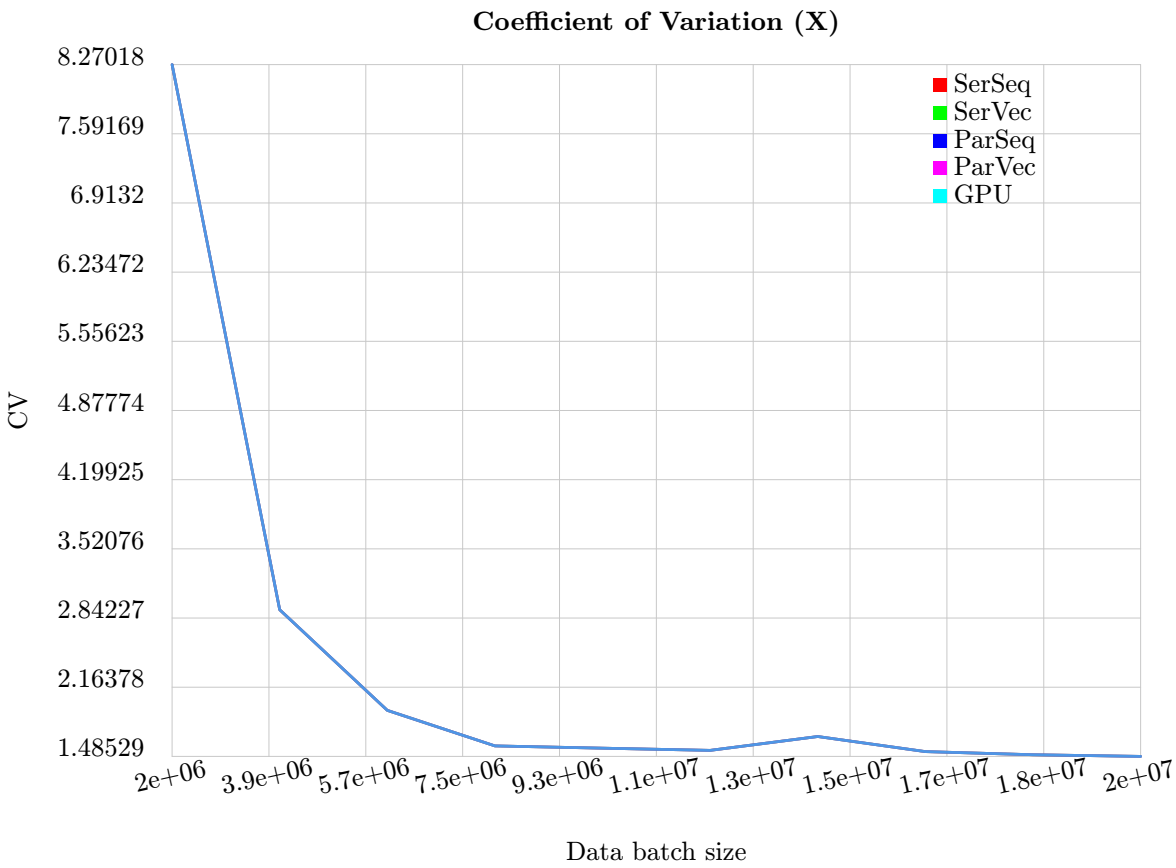


Obrázek 6: Naměřené časy pro jednotlivé metody výpočtů s vypnutými optimalizacemi (*O2*) na *ACC_001.csv*, sloupec *X*

K dodržení zadání program rovněž vykresluje následující grafy – konvergence vypočtených hodnot v závislosti na počtu dat užitých k jejich výpočtu. Z těchto grafů však není vidět příliš informací, neboť všechny metody výpočtů nepřekvapivě dospěly ke stejným výsledkům, tedy všech pět křivek leží kompletně na sobě. Konvergence mediánu absolutní odchylky a variačního koeficientu pro data *X* z *ACC_001.csv* jsou vidět na Obrázku 7 a 8.



Obrázek 7: Konvergence vypočtených hodnot *MAD* na *ACC_001.csv*, sloupec *X*



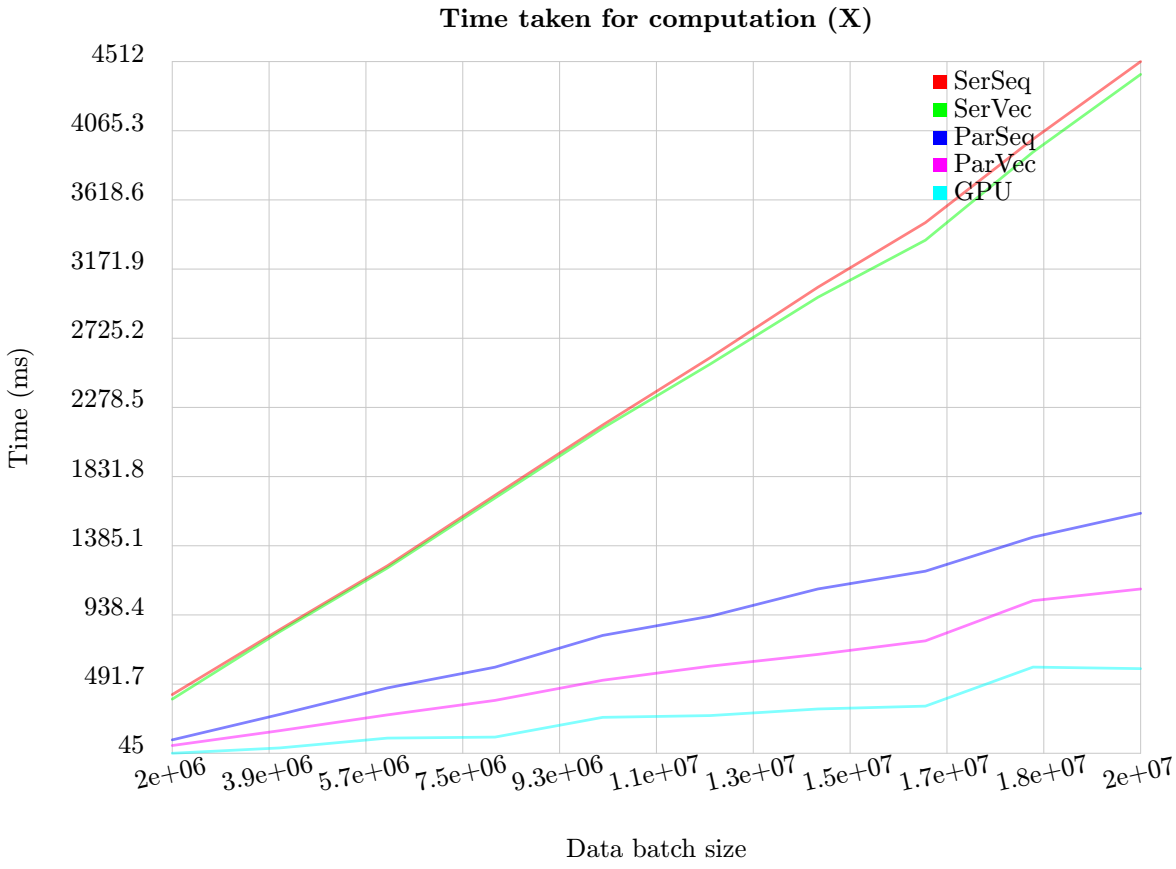
Obrázek 8: Konvergence vypočtených hodnot CV na *ACC_001.csv*, sloupec X

4.2.2 Porovnání *Double* a *Single* přesnosti

Hlavní rozdíl při užití *float* oproti *double* by měla být rychlost a přesnost výpočtu – veškeré výpočty by měly být znatelně rychlejší, přesnost by však měla být znatelně nižší. Velký rozdíl by také měl být ve vektorizaci – do jednoho vektorového registru se nyní vejde 8 *float* čísel (původně 4 *double* čísla).

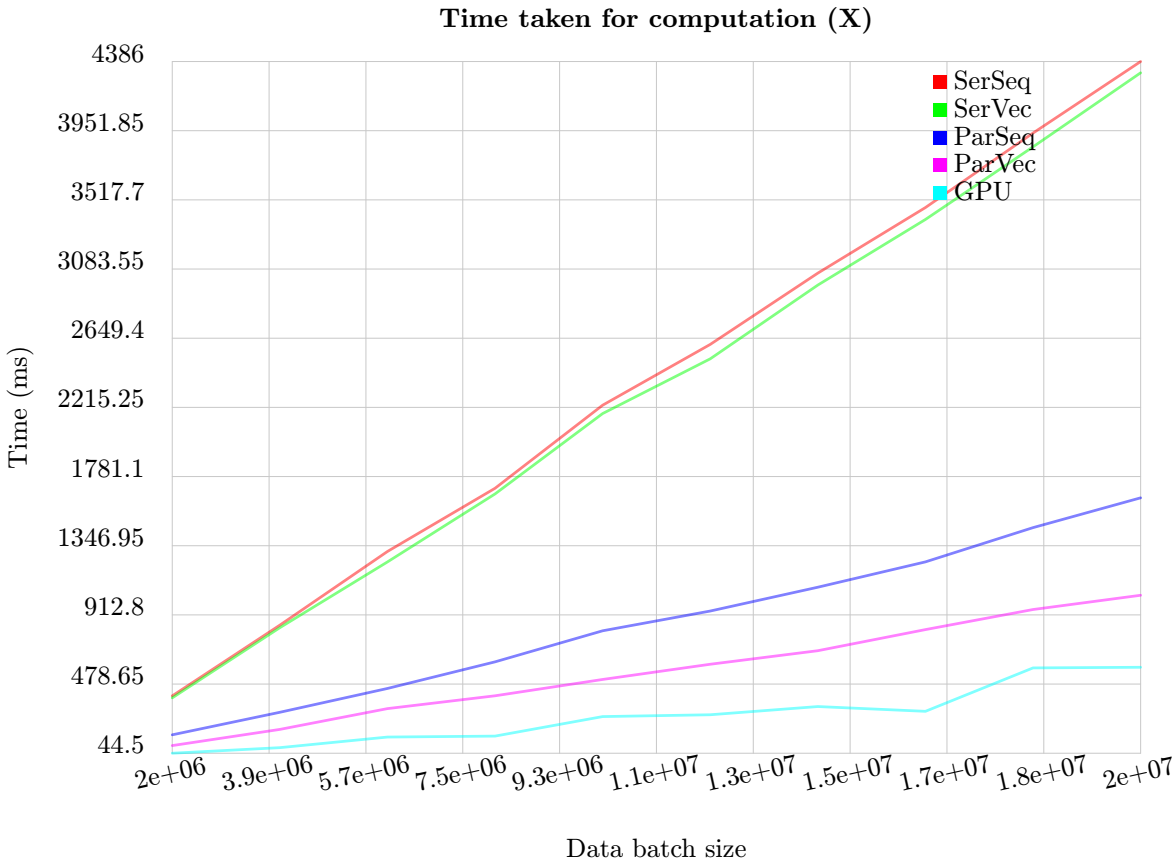
Na Obrázku 9 je opět vidět, že *GPU* je nejrychlejší metodou – hlavně díky implementaci *bitonic sortu* pomocí *sorting network*.

Další velký skok je opět mezi *sériovými* a *paralelními* výpočty – opět způsobeno tím, že implementovaný *merge sort* je paralelizován. Jelikož však toto řazení není vektorizované, nejsou patrné příliš velké rozdíly mezi *sekvenčními* a *vektorizovanými* běhy, ale rozdíly oproti Obrázky 5 jsou lépe vidět (zejména *modrá* a *fialová* křivka).



Obrázek 9: Naměřené časy pro jednotlivé metody výpočtů s užitím *single* precision na *ACC_001.csv*, sloupec *X*

Na následujícím Obrázku 10 je stejný výpočet pouze s vypnutými optimalizacemi (příznak překladače *O2*). Z grafu je opět jemně patrné větší oddálení *sekvenčních* a *vektorizovaných* výpočtů – ale opět to není tak znatelné z důvodu užití *std::for_each()*, které jsou řízeny přímo *execution policy*, která zajišťuje vypnutí auto-vektorizace. Kód mimo tyto smyčky však mohl podléhat auto-vektorizaci.



Obrázek 10: Naměřené časy pro jednotlivé metody výpočtů s užitím *single* precision a s vypnutými optimalizacemi (*O2*) na *ACC_001.csv*, sloupec *X*

4.3 Výsledné hodnoty *MAD* a *CV*

V následující Tabulce 2 jsou vidět všechny vypočtené hodnoty *MAD* a *CV* pro každý ze vstupních souborů ze zadání (využita přesnost *double*).

Soubor	**MAD X**	**CV X**	**MAD Y**	**CV Y**	**MAD Z**	**CV Z**
ACC_001.csv	23	1.485	18	36.406	26	1.497
ACC_002.csv	13	-0.558	19	149.964	20	1.588
ACC_003.csv	29	-2.364	31	-14.762	23	1.565
ACC_004.csv	30	-2.198	24	5.894	21	-1.094
ACC_005.csv	26	1.760	23	8.645	25	1.576
ACC_006.csv	19	1.026	12	-14.759	27	1.410
ACC_007.csv	27	2.150	24	23.061	20	1.502
ACC_008.csv	37	5.112	24	3.964	21	1.309
ACC_009.csv	15	-0.956	23	12.257	23	2.214
ACC_010.csv	32	-2.703	21	-7.378	29	3.303
ACC_011.csv	22	2.056	21	13.554	17	1.201
ACC_012.csv	15	-0.759	18	-8.139	20	1.510
ACC_013.csv	22	-3.153	22	7.225	31	1.923
ACC_014.csv	20	-1.215	23	19.718	22	1.415
ACC_015.csv	23	-13.898	21	5.091	37	-183.608
ACC_016.csv	24	1.414	24	2.992	21	1.520

Tabulka 2: Výsledné vypočtené hodnoty *MAD* a *CV* s využitím přesnosti *double*

Při využití *single precision* čísel se výsledky liší až od třetího desetinného místa, tedy tabulka by vypadala téměř totožně – z tohoto důvodu zde není uvedena.

5 Závěr

V rámci semestrální práce byla zhotovena aplikace umožňující analýzu různých typů výpočtů variačního koeficientu a mediánu absolutní odchylky. Řešené kombinace typů výpočtů jsou: *Sériové/Paralelní* × *Sekvenční/Vektorizované* + *GPU*.

Grafické výstupy práce demonstrují získané urychlení při paralelizaci a vektorizaci výpočtů. Díky implementaci *bitonic sort* se *sorting networks* bylo také v rámci práce možné správně optimalizovat *GPU* výpočty pro získání očekávaného urychlení.

Hlavní možné vylepšení do budoucnosti je optimalizace *CPU* řazení, neboť implementovaný *merge sort* je znatelně pomalejší než např. dostupné standardní řazení knihovny C++ (std::sort()).

Dalším vylepšením by mohla být implementace vlastního alokátoru pro *std::vector* – tím by se dala zarovnat paměť na 4 byty a bylo by možné využít zarovnané načítání a ukládání dat v rámci vektorizace.

A Uživatelská příručka

Předpoklady

Pro správný překlad a běh programu je zapotřebí mít:

- CMake verze 3.21 a vyšší
- C++ překladač (např. clang, gcc nebo msvc)
- Standardní C++ knihovny standardu C++17 a novější

Nástroj CMake si lze stáhnout z oficiálních stránek `https://cmake.org`. Standardní knihovny jazyka C++ jsou součástí překladače.

Překlad

Pro účely překladu slouží nástroj CMake spolu s překladačem jazyka C++. V adresáři projektu si např. vytvořte adresář s názvem *build*. Výpis 1 ukazuje postup, jak si aplikaci v tomto adresáři sestavit pod operačním systémem Windows. Pro Linux je postup ukázán ve Výpisu 2.

Nezapomeňte na přepínač **Release** u obou operačních systémů! Z hlediska optimalizací je velice důležité sestavit aplikaci v tomto módu.

Rovněž nezapomeňte sestavit aplikaci **64-bitovým** překladačem! Je to nezbytné z důvodu způsobu načítání dat – ve 32-bitovém režimu by nemuselo stačit dostupných 4 GB adresovatelné paměti! Načtení dat je paralelní a načítá celý soubor do hlavní paměti RAM – nejvíce paměťově náročný úsek aplikace.

Pro využití příznaku „D_USE_FLOAT“ a pro kompilaci v režimu *single precision* je možné modifikovat první příkaz volání nástroje *cmake* přidáním -DCMAKE_CXX_FLAGS=”-D_USE_FLOAT”. Pro Windows by pak první příkaz vypadal takto:

cmake .. –DCMAKE_CXX_FLAGS=”–D_USE_FLOAT”

```
C:\SP\build>cmake ..
... (zde by se mel objevit dlouhy vypis)
-- Configuring done
-- Generating done
-- Build files have been written to: C:\SP\build

C:\SP\build>cmake --build . --config Release
... (zde by se mel objevit dlouhy vypis)

C:\SP\build>
```

Listing 1: Překlad na Windows

```
user@pc:/SP/build$ cmake -DCMAKE_BUILD_TYPE=Release ..
... (zde by se mel objevit dlouhy vypis)
-- Configuring done
-- Generating done
-- Build files have been written to: /SP/build
user@pc:/SP/build$ make
... (zde by se mel objevit dlouhy vypis)
user@pc:/SP/build$
```

Listing 2: Překlad na Linux

Spuštění

Po překladu dle výše uvedeného návodu by se spustitelný soubor pod operačním systémem Windows měl nacházet ve složce SP/build/Release; spustitelný soubor pro operační systém Linux by se měl nacházet ve složce SP/build.

Aplikace předpokládá vstupní parametry z příkazové řádky – povinný je buď příznak -f nebo -d. Příklad úspěšného spuštění předvádí Výpisy 3 a 4.

```
C:\SP\build\Release>SP.exe -f data/ACC_001.csv --all -r 10 -n
```

Listing 3: Spuštění na Windows

```
user@pc:/SP/build$ ./SP -d data --par --vec --gpu --no_graphs
```

Listing 4: Spuštění na Linux

Následuje seznam možných příznaků a přepínačů s jejich vysvětlením:

- -f ⟨soubor⟩ – definice vstupního souboru.
- -d ⟨adresář⟩ – definice adresáře, který obsahuje vstupní soubory.

Tímto způsobem program ví, odkud čerpat data. Lze tedy zpracovat soubor jeden, případně celý adresář plný datových souborů (v očekávaném formátu ze zadání). Příznaky jsou navzájem výlučné, ale právě jeden je povinný.

Další možné argumenty z příkazové řádky už nejsou povinné; vypadají následovně:

- -r ⟨počet opakování experimentu⟩ – ze všech experimentů se do výsledného grafu propisuje *medián* výsledků (pro splnění zadání nutno spustit s „-r 10“).
- -n ⟨počet „batchů“ dat⟩ – počet kusů, do kolika se vstupní data mají roztrhat, nad nimiž se bude r-krát provádět daný výpočet. Prakticky se jedná o granularitu osy X ve výstupních grafech.
- --par – za tímto příznakem se neočekává žádná hodnota. Jedná se o pouhý přepínač mezi *sériovým* a *paralelním* výpočtem.
- --vec – za tímto příznakem se opět neočekává hodnota. Jedná se opět o přepínač, tentokrát mezi *sekvenčním* a *vektorizovaným* výpočtem.
- --gpu – opět se jedná o přepínač bez hodnoty – přepíná se mezi *CPU* a *GPU* výpočtem.
- --all – výše popsané tři přepínače umožňují spustit pouze jeden typ výpočtu nad daným datovým souborem (opět se jedná o přepínač neočekávající hodnotu). Pro splnění zadání ještě existuje tento přepínač, díky kterému budou iterativně provedeny všechny kombinace typů výpočtů. Tento přepínač mění grafický výstup na grafy s pěti čarami – každá křivka odpovídá jinému typu výpočtu (pokud je program spuštěn v režimu jednoho konkrétního typu výpočtu grafy obsahují tři křivky – čára pro každý sloupec vstupních dat – X, Y a Z).
- –no_graphs – za tímto příznakem opět není očekávána žádná hodnota. Tento přepínač umožňuje zamezit generování obrázků na konci běhu programu (výhodné hlavně při vývoji pro ladění).
- -h – vypsání nápovědy.
- --help – vypsání nápovědy.

B Zajímavé úryvky kódu

Úryvky mají odstraněné komentáře pro úsporu místa. Pro okomentovaný kód je nutno nahlédnout do projektu. Nejzajímavější kód z pohledu autora je výkonný kód stojící za výpočty – *Sériové/Paralelní* × *Sekvenční/Vektorizované* + *GPU*.

```
#ifdef _USE_FLOAT
using decimal = float;
#else
using decimal = double;
#endif
```

Listing 5: utils.h pro následné chápání „decimal“

```cpp
template<typename derived>
class computations {
public:
    template <typename exec_policy>
    [[nodiscard]] decimal compute_mad(exec_policy policy, std::vector<decimal> &arr) {
        static_cast<derived *>(this)->sort(policy, arr);

        const auto median = static_cast<decimal>((arr[arr.size() / 2] + arr[(arr.size() - 1) / 2]) / 2.0);

        std::vector<decimal> diff(arr.size());

        static_cast<derived *>(this)->compute_abs_diff(policy, arr, median, diff);

        size_t left = diff.size() / 2 - 1, right = diff.size() / 2;
        decimal prev = 0, curr = 0;
        for (size_t i = 0; i <= diff.size() / 2; i++) {
            prev = curr;
            curr = diff[left] >= diff[right] ? diff[right++] : diff[left--];
        }

        return static_cast<decimal>((arr.size() & 1) ? curr : (prev + curr) / 2.0);
    }

    template <typename exec_policy>
    [[nodiscard]] decimal compute_coef_var(exec_policy policy, const std::vector<decimal> &arr) {
        const auto n = static_cast<double>(arr.size());

        decimal sum = 0, sum_sq = 0;
        static_cast<derived *>(this)->compute_sums(policy, arr, sum, sum_sq);

        return static_cast<decimal>(std::sqrt((sum_sq - sum * sum / n) / n) / (sum / n));
    }
};
```

Listing 6: Třída pro výpočty (statický polymorfismus)

```cpp
class seq_comp : public computations<seq_comp> {
public:
    template<typename exec_policy>
    static void sort(exec_policy policy, std::vector<decimal> &arr) {
        merge_sort(policy, arr);
    }

    template<typename exec_policy>
    static void compute_abs_diff(exec_policy policy, const std::vector<
        decimal> &arr, decimal median, std::vector<decimal> &diff) {
        std::for_each(policy, arr.begin(), arr.end(), [&](const auto &val)
            {
            diff[&val - &arr[0]] = std::abs(val - median);
        });
    }

    template<typename exec_policy>
    static void compute_sums(exec_policy policy, const std::vector<decimal>
         &arr, decimal &sum, decimal &sum_sq) {
        const auto max_num_threads = std::thread::hardware_concurrency();
        const auto chunk_size = arr.size() / max_num_threads;
        std::vector<decimal> sums(max_num_threads, 0);
        std::vector<decimal> sums_sq(max_num_threads, 0);
        std::vector<size_t> chunk_indices(max_num_threads);
        std::iota(chunk_indices.begin(), chunk_indices.end(), 0);

        std::for_each(policy, chunk_indices.begin(), chunk_indices.end(),
            [&](const auto &i) {
            const size_t start = i * chunk_size;
            const size_t end = (i == max_num_threads - 1) ? arr.size() :
                start + chunk_size;

            for (size_t j = start; j < end; j++) {
                sums[i] += arr[j];
                sums_sq[i] += arr[j] * arr[j];
            }
        });

        for (size_t i = 0; i < max_num_threads; i++) {
            sum += sums[i];
            sum_sq += sums_sq[i];
        }
    }
};
```

Listing 7: Třída pro sekvenční výpočty (sériové / paralelní)

```
class vec_comp : public computations<vec_comp> {
public:
    template<typename exec_policy>
    static void sort(exec_policy policy, std::vector<decimal> &arr) {
        merge_sort(policy, arr);
    }

    template<typename exec_policy>
    static void compute_abs_diff(exec_policy policy, const std::vector<
        decimal> &arr, decimal median, std::vector<decimal> &diff) {
        const size_t n = arr.size();

        #ifndef _USE_FLOAT
        const size_t vec_capacity = sizeof(__m256d) / sizeof(decimal);
        #else
        const size_t vec_capacity = sizeof(__m256) / sizeof(decimal);
        #endif

        #ifndef _USE_FLOAT
        __m256d median_vec = _mm256_set1_pd(median);
        #else
        __m256 median_vec = _mm256_set1_ps(median);
        #endif

        std::vector<size_t> indices(n / vec_capacity);
        for (size_t i = 0; i < indices.size(); i++)
            indices[i] = i * vec_capacity;

        std::for_each(policy, indices.begin(), indices.end(), [&](const
            auto &i) {
            #ifndef _USE_FLOAT
            __m256d arr_vec = _mm256_loadu_pd(&arr[i]);

            __m256d diff_vec = _mm256_sub_pd(arr_vec, median_vec);
            __m256d sign_bit = _mm256_set1_pd(-0.0);
            diff_vec = _mm256_andnot_pd(sign_bit, diff_vec);

            _mm256_storeu_pd(&diff[i], diff_vec);
            #else
            __m256 arr_vec = _mm256_loadu_ps(&arr[i]);

            __m256 diff_vec = _mm256_sub_ps(arr_vec, median_vec);
            __m256 sign_bit = _mm256_set1_ps(-0.0);
            diff_vec = _mm256_andnot_ps(sign_bit, diff_vec);

            _mm256_storeu_ps(&diff[i], diff_vec);
            #endif
        });

```

```
48          for (size_t i = n - n % vec_capacity; i < n; i++)
49              diff[i] = std::abs(arr[i] - median);
50      }
51
52      template<typename exec_policy>
53      static void compute_sums(exec_policy policy, const std::vector<decimal>
            &arr, decimal &sum, decimal &sum_sq) {
54          const size_t n = arr.size();
55
56          #ifndef _USE_FLOAT
57          const size_t vec_capacity = sizeof(__m256d) / sizeof(decimal);
58          #else
59          const size_t vec_capacity = sizeof(__m256) / sizeof(decimal);
60          #endif
61
62          const auto max_num_threads = std::thread::hardware_concurrency();
63          auto chunk_size = arr.size() / max_num_threads;
64          chunk_size = chunk_size - chunk_size % vec_capacity;
65          std::vector<decimal> sums(max_num_threads, 0);
66          std::vector<decimal> sums_sq(max_num_threads, 0);
67          std::vector<size_t> chunk_indices(max_num_threads);
68          std::iota(chunk_indices.begin(), chunk_indices.end(), 0);
69
70          std::for_each(policy, chunk_indices.begin(), chunk_indices.end(),
                [&](const auto &i) {
71              const size_t start = i * chunk_size;
72              const size_t end = start + chunk_size;
73
74              #ifndef _USE_FLOAT
75              __m256d sum_vec = _mm256_setzero_pd();
76              __m256d sum_sq_vec = _mm256_setzero_pd();
77              #else
78              __m256 sum_vec = _mm256_setzero_ps();
79              __m256 sum_sq_vec = _mm256_setzero_ps();
80              #endif
81
82              for (size_t j = start; j < end; j += vec_capacity) {
83                  #ifndef _USE_FLOAT
84                  __m256d arr_vec = _mm256_loadu_pd(&arr[j]);
85
86                  sum_vec = _mm256_add_pd(sum_vec, arr_vec);
87                  sum_sq_vec = _mm256_add_pd(sum_sq_vec, _mm256_mul_pd(
                        arr_vec, arr_vec));
88                  #else
89                  __m256 arr_vec = _mm256_loadu_ps(&arr[j]);
90
91                  sum_vec = _mm256_add_ps(sum_vec, arr_vec);
92                  sum_sq_vec = _mm256_add_ps(sum_sq_vec, _mm256_mul_ps(
                        arr_vec, arr_vec));
93                  #endif
```

```
94              }
95
96              std::vector<decimal> sum_arr(vec_capacity, 0.0);
97              std::vector<decimal> sum_sq_arr(vec_capacity, 0.0);
98              #ifndef _USE_FLOAT
99              _mm256_storeu_pd(sum_arr.data(), sum_vec);
100             _mm256_storeu_pd(sum_sq_arr.data(), sum_sq_vec);
101             #else
102             _mm256_storeu_ps(sum_arr.data(), sum_vec);
103             _mm256_storeu_ps(sum_sq_arr.data(), sum_sq_vec);
104             #endif
105
106             for (size_t j = 0; j < vec_capacity; j++) {
107                 sums[i] += sum_arr[j];
108                 sums_sq[i] += sum_sq_arr[j];
109             }
110         });
111
112         for (size_t i = 0; i < max_num_threads; i++) {
113             sum += sums[i];
114             sum_sq += sums_sq[i];
115         }
116
117         for (size_t i = chunk_size * max_num_threads; i < n; i++) {
118             sum += arr[i];
119             sum_sq += arr[i] * arr[i];
120         }
121     }
122 };
```

Listing 8: Třída pro vektorizované výpočty (sériové / paralelní) (je zde vidět i užití příznaku překladače _USE_FLOAT – potom „decimal“ je float

```
__kernel void bitonic_sort(__global double *arr, const uint stage, const
    uint pass) {
    int gid = get_global_id(0);

    uint pair_distance = 1 << (stage - pass);
    uint block_width = 2 * pair_distance;

    uint left_id = (gid & (pair_distance - 1)) + (gid >> (stage - pass)) *
        block_width;
    uint right_id = left_id + pair_distance;

    double left = arr[left_id];
    double right = arr[right_id];

    uint same_dir_block_width = gid >> stage;
    uint same_dir = same_dir_block_width & 0x1;

    if (same_dir) {
        uint temp = right_id;
        right_id = left_id;
        left_id = temp;
    }

    if (left < right) {
        arr[left_id] = left;
        arr[right_id] = right;
    } else {
        arr[left_id] = right;
        arr[right_id] = left;
    }
}

__kernel void my_abs_diff(__global const double *arr, __global double *diff
    , const double median) {
    int i = get_global_id(0);
    diff[i] = fabs(arr[i] - median);
}

__kernel void reduce_sum(__global const double *arr, __global double *sums,
     __global double *sums_sq, const int n) {
    int gid = get_global_id(0);
    int local_id = get_local_id(0);
    int group_id = get_group_id(0);

    __local double partial_sums[256];
    __local double partial_sums_sq[256];

    partial_sums[local_id] = (gid < n) ? arr[gid] : 0.0;
    partial_sums_sq[local_id] = (gid < n) ? arr[gid] * arr[gid] : 0.0;
```

```
46      barrier(CLK_LOCAL_MEM_FENCE);
47
48      for (int stride = get_local_size(0) / 2; stride > 0; stride /= 2) {
49          if (local_id < stride) {
50              partial_sums[local_id] += partial_sums[local_id + stride];
51              partial_sums_sq[local_id] += partial_sums_sq[local_id + stride
                    ];
52          }
53
54          barrier(CLK_LOCAL_MEM_FENCE);
55      }
56
57      if (local_id == 0) {
58          sums[group_id] = partial_sums[0];
59          sums_sq[group_id] = partial_sums_sq[0];
60      }
61  }
```

Listing 9: GPU výpočty (kernel code) (pouze pro double – řešení float obdobné jako u vektorizace)

```
constexpr size_t local_size = 256;

class gpu_comps : public computations<gpu_comps> {
private:
    /* Initialized by the constructor -- not present here for space reasons */
    cl::Platform platform;
    cl::Device device;
    cl::Context context;
    cl::CommandQueue queue;
    cl::Program program;
    cl::Buffer input_buffer;

public:
    template<typename exec_policy>
    void sort(exec_policy policy, std::vector<decimal> &arr) {
        (void) policy;  /* Supress warning about unused policy */

        const auto n = arr.size();

        size_t pow = 1, num_stages = 0;
        while (pow < n) {
            pow <<= 1;
            num_stages++;
        }
        arr.resize(pow, std::numeric_limits<decimal>::max());

        /* Create buffers -- input buffer is set by the sums function, but we need padded array here */
        this->input_buffer = cl::Buffer(this->context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sizeof(decimal) * pow, arr.data());

        cl::Kernel bitonic_sort_kernel(program, "bitonic_sort");
        bitonic_sort_kernel.setArg(0, this->input_buffer);

        size_t global_size = (pow / 2 + local_size - 1) / local_size * local_size;

        for (size_t stage = 0; stage < num_stages; stage++) {
            bitonic_sort_kernel.setArg(1, static_cast<int>(stage));
            for (size_t pass = 0; pass < stage + 1; pass++) {
                bitonic_sort_kernel.setArg(2, static_cast<int>(pass));

                this->queue.enqueueNDRangeKernel(bitonic_sort_kernel, cl::NullRange, cl::NDRange(global_size), cl::NDRange(local_size));
                this->queue.finish();
            }
        }
```

```
44
45          this->queue.enqueueReadBuffer(this->input_buffer, CL_TRUE, 0,
                sizeof(decimal) * pow, arr.data());
46
47          arr.resize(n);
48      }
49
50      template<typename exec_policy>
51      void compute_abs_diff(exec_policy policy, const std::vector<decimal> &
            arr, decimal median, std::vector<decimal> &diff) {
52          (void) policy;  /* Supress warning about unused policy */
53
54          const auto n = arr.size();
55
56          cl::Buffer buffer_diff(this->context, CL_MEM_WRITE_ONLY, sizeof(
                decimal) * n);
57
58          cl::Kernel kernel(this->program, "my_abs_diff");
59          kernel.setArg(0, this->input_buffer);   /* Use the input buffer --
                it should be set by the sort function */
60          kernel.setArg(1, buffer_diff);
61          kernel.setArg(2, median);
62
63          this->queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange
                (n));
64          this->queue.finish();
65
66          this->queue.enqueueReadBuffer(buffer_diff, CL_TRUE, 0, sizeof(
                decimal) * n, diff.data());
67      }
68
69      template<typename exec_policy>
70      void compute_sums(exec_policy policy, const std::vector<decimal> &arr,
            decimal &sum, decimal &sum_sq) {
71          (void) policy;  /* Supress warning about unused policy */
72
73          const auto n = arr.size();
74
75          std::vector<decimal> sums((n + local_size - 1) / local_size);
76          std::vector<decimal> sums_sq((n + local_size - 1) / local_size);
77
78          /* Copy the array to the GPU once and keep it there */
79          this->input_buffer = cl::Buffer(this->context, CL_MEM_READ_ONLY |
                CL_MEM_COPY_HOST_PTR, sizeof(decimal) * n, const_cast<decimal
                *>(arr.data()));
80          cl::Buffer buffer_sums(this->context, CL_MEM_WRITE_ONLY, sizeof(
                decimal) * sums.size());
81          cl::Buffer buffer_sums_sq(this->context, CL_MEM_WRITE_ONLY, sizeof(
                decimal) * sums_sq.size());
82
```

```
83          cl::Kernel kernel(this->program, "reduce_sum");
84          kernel.setArg(0, this->input_buffer);  /* This function creates the
                 input buffer -- order matters (CV -> MAD) */
85          kernel.setArg(1, buffer_sums);
86          kernel.setArg(2, buffer_sums_sq);
87          kernel.setArg(3, static_cast<int>(n));
88
89          size_t global_size = ((n + local_size - 1) / local_size) *
                local_size;
90          this->queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange
                (global_size), cl::NDRange(local_size));
91          this->queue.finish();
92
93          this->queue.enqueueReadBuffer(buffer_sums, CL_TRUE, 0, sizeof(
                decimal) * sums.size(), sums.data());
94          this->queue.enqueueReadBuffer(buffer_sums_sq, CL_TRUE, 0, sizeof(
                decimal) * sums_sq.size(), sums_sq.data());
95
96          for (size_t i = 0; i < sums.size(); i++) {
97              sum += sums[i];
98              sum_sq += sums_sq[i];
99          }
100     }
101 };
```

Listing 10: Třída pro GPU výpočty (host-side code)