FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

<KIV> KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

PPR - Semestrální práce
Paralelizace a vektorizace výpočtu koeficientu variance a mediánu absolutní odchylky pro medicínská data

Západočeská Univerzita v Plzni
Katedra Informatiky a výpočetní techniky
Semestrální práce - PPR

Vladimíra Kimlová
zimní semestr
29. května 2025

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:

- sériový kód,
- vektorizovaný kód,
- vícevláknový, ale nevektorizovaný kód,
- vícevláknový a vektorizovaný kód,
- GPU kód.

Každý typ výpočtu spusťte 10× 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`:

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

Rady a omezení:

- Nepoužívejte algoritmy průběžného výpočtu mediánu, standardní odchylky apod. Pokud přece jen 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 redukční operace spočítejte také na GPU, ale můžete to udělat pomocí několika kernelů.
- Když 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), čímž si na pár řádcích realizujete všechny 4 varianty výpočtu na CPU.
- SYCL by měl umožnit udělat to samé i pro variantu GPU.

2 Analýza problému

V rámci semestrální práce bylo pracováno s medicínským datasetem obsahujícím data z akcelerometru pro 16 pacientů, pro každého pacienta pak 3 množiny hodnot (`X, Y, Z`). Velikosti jednotlivých množin se pohybují v řádech desítek milionů hodnot – hodí se zde proto paralelní přístup ke zpracování těchto dat. Úkolem je výpočet koeficientu variance a mediánu absolutní odchylky pro každou množinu dat.
Koeficient variance lze obecně získat jako poměr směrodatné odchylky σ a průměru μ takto

$$CV = \frac{\sigma}{\mu} = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}}{\mu}, \quad \text{kde } \mu = \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{1}$$

Tento vzorec lze ještě dále převést do tzv. výpočtového tvaru, kde jsou nejprve napočteny suma (respektive průměr) x_i a suma čtverců x_i

$$CV = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2 - \left(\frac{1}{n}\sum_{i=1}^{n} x_i\right)^2}}{\frac{1}{n}\sum_{i=1}^{n} x_i}. \tag{2}$$

První vzorec 1 vychází z definice koeficientu variance, respektive variance jako takové. V programovém řešení vyžaduje tato podoba vzorce 2 průchody daty; v prvním průchodu je napočítán průměr, ve druhém průchodu je pak napočítán rozdíl čtverců od průměru. Hlavní výhodou tohoto vzorce je numerická stabilita při výpočtech v plovoucí čárce.
Druhý vzorec 2 představuje určité zjednodušení při jeho programové realizaci, neboť vyžaduje pouze jeden průchod daty, v rámci tohoto průchodu jsou napočítány výše zmíněné sumy a je tak eliminována nutnost druhého průchodu. Tato varianta však představuje riziko tzv. **Catastrophic cancellation**, tj. ztráty přesnosti operandů vlivem zaokrouhlování. Vzhledem k neextrémním hodnotám dat byl zvolen tento přístup.

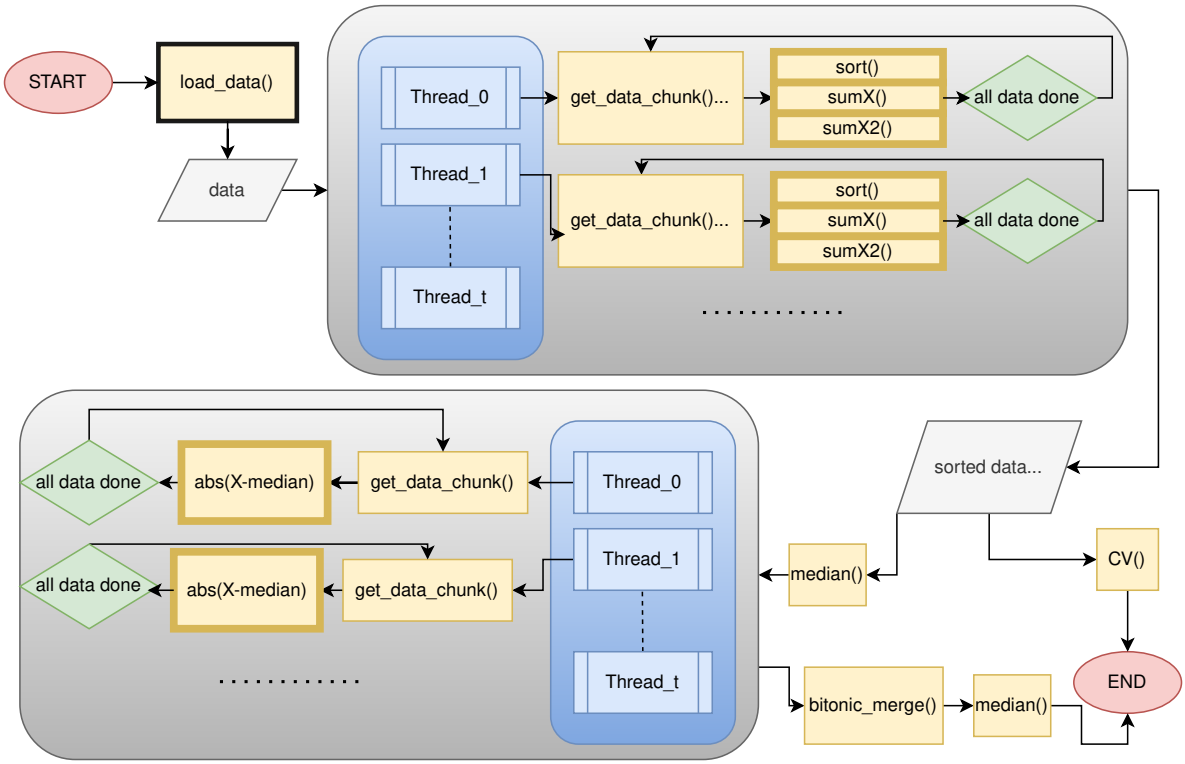
Medián absolutní odchylky lze pak získat následovně

$$MAD = \text{median}(|x_i - \text{median}(x)|), \quad i = 1, \ldots, n. \tag{3}$$

Při programové realizaci to pak vyžaduje data nejprve seřadit, vybrat medián, hodnotu mediánu od dat odečíst v absolutní hodnotě, tato data opět seřadit a vybrat finální medián.

3 Návrh řešení a implementace

Obrázek 1 ilustruje základní návrh řešení, jenž byl představen během průběžné prezentace. Tučně zvýrazněné ohraničení označuje bloky, které lze paralelizovat, zatímco barevné tučné ohraničení identifikuje části, jež jsou paralelizovatelné i vektorizovatelné. Pro běh na GPU opět platí, že tučně ohraničené oblasti odpovídají kódu vykonávanému na GPU. Úpravou oproti původnímu návrhu je skutečnost, že řadicí algoritmus není kompletně vektorizovaný – vektorizace se uplatňuje pouze v posledním průchodu (kde se zároveň provádí i napočítání sumy a sumy druhých mocnin). Dále byl bitonický merge pole (spojení 2 seřazených podpolí) nahrazen pouze na výběr mediánu, neboť celé seřazené pole jako takové již není potřeba.



Obrázek 1: Control flow diagram navrženého řešení z průběžné prezentace

Pro uložení dat je pro každého pacienta využita datová struktura sestávající ze tří vektorů.
Veškeré výpočty jsou organizovány prostřednictvím dvou samostatných tříd – jedné určené pro výpočty na CPU a druhé pro výpočty na GPU. Tyto třídy využívají konceptu „statického polymorfismu“, přestože nezdědily společného předka a fungují jako nezávislé entity, jelikož jejich implementace vychází z odlišných principů (například pro GPU zahrnují kód běžící na straně hostitelského procesoru).
„Statický polymorfismus“ je zde realizován díky tomu, že obě třídy poskytují totožné rozhraní – implementují stejné funkce se shodnými signaturami. Díky této shodě je možné mezi nimi přepínat prostřednictvím statického přetypování, což umožňuje flexibilní použití těchto tříd v kódu bez nutnosti vytvářet explicitní hierarchii tříd nebo abstraktních rozhraní.
Rozhodování o způsobu provádění výpočtů bylo řízeno uživatelem prostřednictvím parametrů příkazové řádky. Uživatel měl možnost specifikovat, zda výpočty poběží na CPU nebo GPU, zda budou prováděny paralelně či sériově, a zda bude využita manuální vektorizace. Tyto volby bylo nutné zpracovat za běhu programu, což přineslo potřebu flexibilní a efektivní správy variant běhu.
Pro výběr zařízení (CPU vs GPU) byla využita třída `device_type`, která mapuje typ zařízení na odpovídající varianty zpracování, jako například `CPU_data_processing` nebo `GPU_data_processing`. Podobně byla navržena třída `execution_policy`, která spravuje volbu politiky smyček, jako je sekvenční (`std::execution::seq`) nebo paralelní (`std::execution::par`) provádění. Obě třídy využívají mapování a varianty (`std::variant`) pro flexibilní správu možností. Konstrukce `std::visit` pak zajišťuje dynamický výběr a správné nastavení algoritmů během běhu programu.

3.1 Měření času běhu jednotlivých metod

Měření veškerého času běhu jednotlivých výpočtů je provedeno pomocí jednotné variadické šablonové funkce 1, která dokáže přijmout jako argument jakoukoli jinou funkci, jejíž čas běhu má změřit. Návratem této funkce je pak čas běhu a návratové hodnoty vstupní funkce.

```
/**
 * @brief Measure the time taken by a function to execute
 * Using variadic templates to accept any function and its arguments
 *
 * @tparam Func Function type
 * @tparam Args Argument types
 * @param f Function to measure
 * @param args Arguments to pass to the function
```

```
 * @return std::pair<double, decltype(f(args...))> Time taken and
     return value of the function
 */
template<typename Func, typename... Args>
auto measure_time(Func &&func, Args &&... args) {
    auto start = std::chrono::high_resolution_clock::now();
    auto result = std::forward<Func>(func)(std::forward<Args>(args)
        ...);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    return std::make_tuple(duration.count(), result);
}
```

Úryvek kódu 1: Funkce pro měření času běhu libovolné funkce

3.2 Zpracování uživatelského vstupu

Pro zpracování uživatelského vstupu a konfiguraci běhu programu byl implementován vlastní parser argumentů z příkazové řádky. Každému argumentu pak přísluší jeho název, nápověda, je-li povinný, má-li hodnotu a jeho defaultní hodnota. Dále byla přidána možnost nadefinovat si tzv. vzájemně exkluzivní skupiny argumentů, tj. argumenty, které se nesmí vyskytovat společně (například nedává smysl pouštět program na GPU a zároveň požadovat manuální AVX vektorizaci).
Kromě konfigurace pomocí příkazové řádky je navíc umožněno přidat překladači flag `/D_FLOAT`, který umožňuje veškeré výpočty provést ve *single precision* aritmetice. Tohoto je docíleno pomocí klíčového slova `using`.

3.3 Paralelní načtení dat

Vzhledem k velkému objemu dat představovalo načtení dat značné úzké hrdlo celého problému, neboť při jednoduchém načtení pomocí `ifstream` trvalo načtení přibližně 2 × déle než při současné implementaci pomocí `fopen_s` a `fread`. Data jsou takto načtena do pomocného vektoru (`std::vector<std::string_view>`) řádků. Samotná paralelizace se pak uplatňuje při parsování těchto řádků do jednotlivých vektorů **X, Y, Z**.

3.4 Výběr mediánu v bitonickém poli

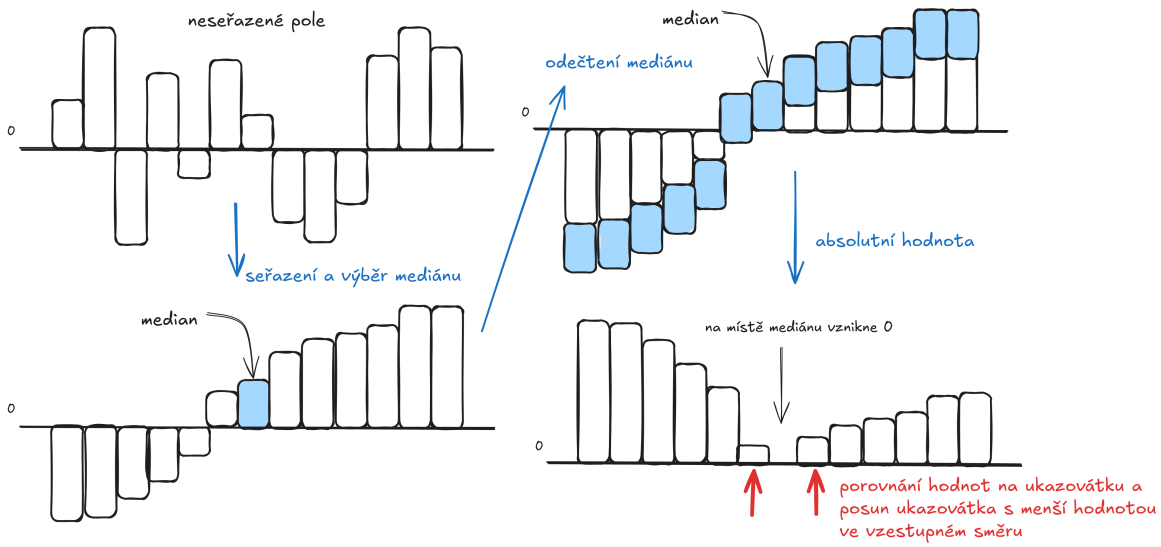
S druhým řazením dat, po odečtení mediánu a výpočtu absolutní hodnoty, již není nutno zacházet jako s řazením neznámého pole, data v tomto poli jsou již z poloviny seřazena a navíc pro výběr mediánu již není nutné měnit pořadí prvků tohoto pole, stačí

pouze výběr mediánu v tomto částečně seřazeném poli. Toto zjednodušení je nastíněno na Obrázku 2.

1. Nejprve je pole seřazeno,
2. dále je nalezena hodnota mediánu (prostřední prvek nebo průměr ze „dvou prostředních“ v závislosti na sudosti počtu prvků),
3. nalezená hodnota mediánu je odečtena od všech prvků,
4. po aplikování absolutní hodnoty na jednotlivé prvky vznikají 2 seřazená podpole (jedno sestupně a druhé vzestupně). Výslednou hodnotu mediánu lze jednoduše vybrat za pomocí dvou ukazovátek jednoduchým algoritmem: obě ukazovátka začínají v prostředku pole, hodnoty ukazovátek jsou porovnány a ukazovátko s menší hodnotou se posouvá ve vzestupném směru seřazeného podpole, toto je opakováno k krát, kde k je polovina délky pole. Podrobnější detaily lze vyčíst v pseudokódu 1 níže.

Algorithm 1 Nalezení mediánu v bitonickém poli

```
1:  function FindMedian(arr, n)
2:      Input: arr: vstupní pole, n: délka pole
3:      Output: median of the array
4:      left_middle ← ⌊(n − 1)/2⌋, right_middle ← ⌊n/2⌋      ▷ Nalezení prostředních indexů
5:      prev ← 0, curr ← 0
6:      for i ← 0 to ⌊n/2⌋ do
7:          prev ← curr
8:          if arr[left_middle] ≥ arr[right_middle] then
9:              curr ← arr[right_middle]
10:             right_middle ← right_middle + 1
11:         else
12:             curr ← arr[left_middle]
13:             left_middle ← left_middle - 1
14:         end if
15:     end for
16:     if n is odd then
17:         return curr
18:     else
19:         return (prev + curr) / 2
20:     end if
21: end function
```



Obrázek 2: Nalezení mediánu absolutní odchylky

3.5 Paralelizace a vektorizace výpočtů na CPU

Ukázky některých částí zdrojového kódu popisované v této části jsou dostupné v příloze 2. Pro řazení jednotlivých vektorů byla v implementaci na CPU použita metoda **Bottom-up merge sort**. Tento algoritmus byl zvolen pro svou stabilitu, efektivitu při práci s většími objemy dat a jednoduchou paralelizovatelnost. Samotná paralelizace pak byla provedena ve vnitřních, spojovacích, smyčkách řadicího algoritmu. V rámci závěrečné iterace, která spočívá ve spojování dílčích seřazených částí do finálního celistvého pole, byly současně vypočítány potřebné sumy pro stanovení koeficientu variance. Tímto postupem bylo dosaženo vyšší efektivity díky eliminaci nutnosti samostatného průchodu daty pro tyto výpočty.

Pro umožnění paralelizace výpočtu sum (sumy a sumy čtverců) bylo nezbytné implementovat tzv. redukční sumu, viz pseudokód 2. Tento přístup eliminuje potřebu nákladného zamykání sdílené proměnné při zapisování mezivýsledků, čímž se výrazně zvyšuje efektivita výpočtu. Namísto přímého zapisování do sdílené paměti je každému výpočetnímu vláknu přidělena část dat a výsledek pro tuto část je zapsán na příslušnou pozici pole lokálních sum nezávisle na ostatních vláknech. V závěrečném kroku jsou tyto lokální sumy sloučeny (redukce) do jednoho finálního výsledku. Tento postup minimalizuje synchronizační režii a maximalizuje využití paralelního výpočetního výkonu.

V případě vektorizovaného přístupu byla navíc využita schopnost SIMD instrukcí zpracovávat více prvků současně. V rámci každého bloku navíc vlákno načítalo data po skupinách odpovídajících šířce SIMD registru, přičemž byly současně počítány

Algorithm 2 Paralelní výpočet redukční sumy

Input: arr: vstupní pole, n: délka pole
Output: sum: celková suma
$max\_threads \leftarrow$ Počet vláken na CPU ▷ Získání dostupného počtu vláken
$chunk\_size \leftarrow n / max\_threads$
$local\_sums \leftarrow [0.0, 0.0, \dots]$ ▷ Pole lokálních sum pro každé vlákno
for (každé vlákno $thread\_id$ od 0 do $max\_threads - 1$ **v paralelním režimu**) **do**
    $start \leftarrow thread\_id \cdot chunk\_size$
    $end \leftarrow \min((thread\_id + 1) \cdot chunk\_size, n)$
    $local\_sum \leftarrow 0.0$ ▷ Inicializace lokální sumy pro vlákno
    for $i \leftarrow start$ to $end - 1$ **do**
        $local\_sum \leftarrow local\_sum + arr[i]$ ▷ Akumulace hodnot
    end for
    $local\_sums[thread\_id] \leftarrow local\_sum$ ▷ Uložení výsledku pro vlákno
end for
$sum \leftarrow$ Accumulate($local\_sums$) ▷ Redukce lokálních sum do globální sumy

suma a suma čtverců přímo ve vektorových registrech. Horizontální redukce uvnitř registrů umožnila rychlé získání dílčích výsledků (lokálních sum), zatímco zbytkové prvky, které nebyly násobkem šířky registru, byly zpracovány skalárně. Vektorový výpočet sumy pole je naznačen v pseudokódu 3.

Výpočet koeficientu variance byl realizován na základě vypočítaných sum, přičemž tento krok je z hlediska implementace již triviální. Pro stanovení mediánu absolutní odchylky (MAD) bylo nejprve nezbytné extrahovat medián z již seřazeného pole hodnot, což opět představovalo jednoduchou operaci. Následný výpočet absolutní odchylky, spočívající v odečtení mediánu od každého prvku pole a aplikaci absolutní hodnoty, bylo možné efektivně provést jak paralelně, tak vektorizovaně. Vzhledem k sekvenční povaze operace odečtu a aplikace absolutní hodnoty nebylo nutné implementovat složité synchronizační mechanismy ani redukční kroky. Paralelní zpracování bylo tedy realizováno prostřednictvím standardní knihovní funkce `std::for_each` s použitím paralelní politiky, která umožnila jednoduchou distribuci výpočtů mezi vlákna. Pro podporu vektorizace bylo zapotřebí optimalizovat výpočet v souladu se šířkou SIMD registrů, detailní implementace nastíněna v pseudokódu 4 níže. Pro výpočet absolutní hodnoty je zde využito maskování pomocí bitové operace ANDNOT se speciálně vytvořenou maskou, která má všechny bity nulové, kromě znaménkového bitu nastaveného na jedničku.

Pro výpočet výsledné hodnoty MAD byl využit navržený algoritmus výběru mediánu v bitonickém poli, viz Sekce 3.4.

Algorithm 3 Výpočet sumy pomocí SIMD

Input: arr: vstupní pole, n: délka pole
Output: sum: celková suma
$stride \leftarrow$ Šířka SIMD registru ▷ Počet prvků zpracovávaných najednou
$vec\_sum \leftarrow$ SetZero() ▷ Inicializace SIMD registru pro akumulaci
for $i \leftarrow 0$ to $n - stride$ **step** $stride$ **do**
    $vec\_vals \leftarrow$ LoadSIMD($arr[i]$) ▷ Načtení hodnot do SIMD registru
    $vec\_sum \leftarrow$ AddSIMD(vec_sum, vec_vals) ▷ Akumulace v SIMD registru
end for
$temp\_sum \leftarrow [0.0, 0.0, \dots]$ ▷ Dočasné pole pro horizontální sumu
Store($temp\_sum, vec\_sum$) ▷ Uložení výsledku SIMD registru
$sum \leftarrow 0.0$ ▷ Inicializace výsledku
for $k \leftarrow 0$ to $stride - 1$ **do**
    $sum \leftarrow sum + temp\_sum[k]$ ▷ Horizontální suma pro konečný výsledek
end for
pro zbylé prvky, které nevyplní celý SIMD registr:
for $i \leftarrow n - (n\%stride)$ to $n - 1$ **do**
    $sum \leftarrow sum + arr[i]$ ▷ Akumulace zbylých prvků
end for

Algorithm 4 Výpočet absolutní odchylky pomocí SIMD a paralelizace

Input: arr: vstupní pole, n: délka pole, $median$: medián pole
Output: abs_diff: pole absolutních odchylek
$med \leftarrow$ Broadcast($median$) ▷ Replikace mediánu přes SIMD registr
$sign\_mask \leftarrow$ CreateMask(-0.0) ▷ Maska pro odstranění znaménka
$step \leftarrow$ SIMD_width ▷ Šířka SIMD registru (počet prvků)
$indices \leftarrow []$ ▷ Inicializace pole indexů
for $i \leftarrow 0$ **to** $n - step$ **step** $step$ **do**
    Append($indices, i$)
end for
▷ Paralelní zpracování pomocí standardní knihovny
$policy \leftarrow$ GetParallelPolicy()
ForEach($policy, indices,$ **do**
    let i **in**
    $vec \leftarrow$ LoadSIMD($arr[i]$) ▷ Načtení $step$ prvků
    $diff \leftarrow vec - med$ ▷ Odečtení mediánu
    $abs\_diff\_vec \leftarrow$ ClearSignBit($diff, sign\_mask$) ▷ Aplikace absolutní hodnoty
    StoreSIMD($abs\_diff[i], abs\_diff\_vec$) ▷ Uložení výsledku
EndForEach)
return abs_diff

3.6 Výpočty na GPU

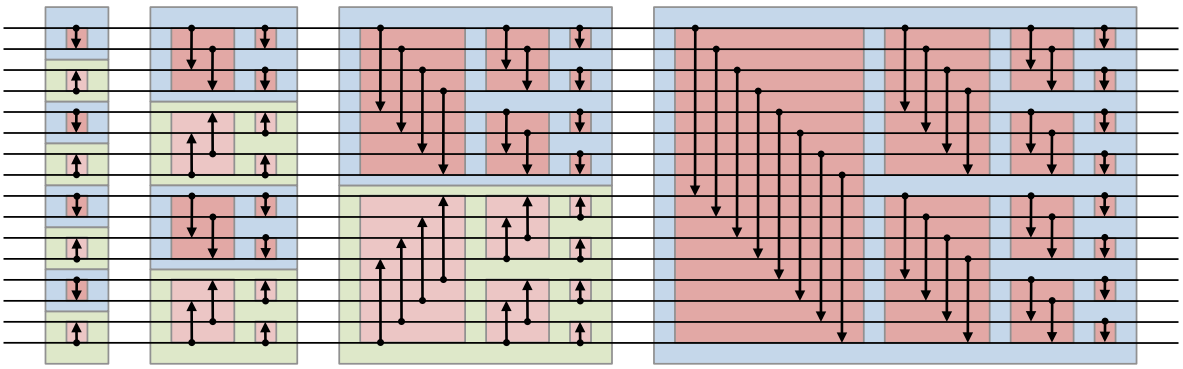
V původní implementaci byl pro řazení na GPU využit algoritmus **merge sort**, analogicky k implementaci na CPU. Nicméně doba výpočtu přesahovala 20 sekund, což bylo v porovnání s verzí na CPU neúnosně pomalé. Vzhledem k charakteru výpočtů na GPU a jejich efektivitě při práci s paralelními datovými strukturami byl místo toho zvolen algoritmus **bitonic merge sort**, který je lépe optimalizován pro architekturu GPU.

Tato změna vedla k výraznému zrychlení, avšak byla ztracena možnost přímého porovnání běhů identických algoritmů na obou platformách. Na druhou stranu tak bylo možné lépe srovnat výkon CPU a GPU při řešení úlohy optimalizované pro jednotlivé platformy. Změna algoritmu a nutnost redukce pro výpočet sum také vedly k tomu, že sumy nebyly nadále počítány během řazení, ale byly vypočítány separátně jako oddělený modul.

3.6.1 Bitonic merge sort

Bitonický merge sort je sofistikovaný algoritmus pro paralelní řazení dat, využívá specifické vlastnosti tzv. *bitonických posloupností*, což jsou sekvence, které nejprve monotónně rostou (nebo klesají) až do určitého vrcholu (nebo minima), a následně monotónně klesají (nebo rostou).

Bitonický merge sort je úzce spjat s konceptem řadících sítí (**sorting networks**). Tyto sítě pracují tak, že každý prvek vstupuje na jednom konci sítě, postupuje skrze jednotlivé úrovně komparátorů a na konci vystupuje na správné pozici. Komparátory ve řadících sítích zajišťují, že při každém průchodu jsou porovnány dva prvky a případně prohozeny, aby odpovídaly požadovanému pořadí. Diagram bitonického řazení sítě se 16 vstupy je na Obrázku 3.



Obrázek 3: Bitonická řadící síť se 16 vstupy, převzato z `https://en.wikipedia.org/wiki/Bitonic_sorter`

Řadící síť pro bitonický merge sort se skládá z následujících prvků:

- **Červené bloky**: Tyto bloky porovnávají odpovídající prvky z horní a dolní poloviny vstupu a zašťují, že horní polovina obsahuje menší prvky než dolní (pro směr dolů), nebo naopak (pro směr nahoru).
- **Modré a zelené bloky**: Modré bloky řadí vstupy do rostoucího pořadí, zatímco zelené bloky je řadí do klesajícího pořadí. Tyto bloky využívají hierarchickou strukturu červených bloků, aby postupně řadily prvky do stále menších podskupin, dokud není každý prvek na správné pozici.

Algoritmus postupuje následujícím způsobem:

1. **Inicializace**: Každý prvek vstupní sekvence je považován za seřazenou posloupnost délky jedna.
2. **Tvorba bitonických posloupností**: Prvky jsou iterativně seskupovány do bitonických posloupností pomocí modrých a zelených bloků. Každý sloupec těchto bloků spojuje N seřazených sekvencí do $N/2$ bitonických sekvencí.
3. **Řazení bitonických posloupností**: Bitonické posloupnosti jsou dále řazeny pomocí červených bloků, dokud nezůstane jediná seřazená posloupnost. Proces řazení je rozdělen do několika *fází* (**stages**), přičemž každá fáze odpovídá zdvojnásobení velikosti řazených posloupností. Každá fáze je dále rozdělena na *podfáze* (**passes**), které řadí prvky v rámci aktuálního bloku.

   Například:

   - V první fázi (stage 0) jsou dvojice prvků seřazeny tak, aby tvořily posloupnosti délky 2.
   - V druhé fázi (stage 1) jsou tyto posloupnosti spojeny do bitonických posloupností délky 4.
   - Tento proces pokračuje, dokud není vstupní sekvence plně seřazena (poslední stage).

3.6.2 Realizace výpočtů na GPU

Veškeré zdrojové kódy jádra popisované v této sekci jsou dostupné v příloze 3.
Jak bylo uvedeno výše, výpočet sum potřebných pro stanovení koeficientu variance byl na GPU proveden odděleně od procesu řazení.
Výpočet sumy a sumy čtverců prvků v poli byl implementován opět pomocí techniky redukce. Každá pracovní jednotka (work-item) načetla odpovídající prvek ze vstupního pole a uložila ho do lokální paměti. V této lokální paměti probíhala redukce pro součet a součet čtverců v rámci pracovního bloku (workgroup). Redukce pak probíhala ve

smyčce, která postupně zkracovala velikost kroku (stride), čímž se hodnoty v každém bloku postupně spojovaly do jednoho součtu a součtu čtverců. Během tohoto procesu byla zajištěna synchronizace mezi pracovními jednotkami. Po dokončení redukce v každé pracovní skupině byl výsledek uložen do globální paměti, kde byla na hostitelském systému provedena finální redukce.
Ziskem těchto sum byl triviálně, na CPU, vypočten koeficient variance.
Řazení bylo implementováno pomocí algoritmu **bitonic merge sort**. Pro správnou funkci tohoto algoritmu bylo nezbytné zajistit, že velikost řazeného pole je mocninou čísla 2. Pokud velikost pole tuto podmínku nesplňovala, bylo vstupní pole upraveno přidáním maximální hodnoty příslušného datového typu (double nebo float). Po provedení řazení byly tyto pomocné hodnoty odstraněny. Proces řazení byl rozdělen do dvou úrovní iterací: stages a passes. Vnější cyklus iteroval přes jednotlivé fáze (stages) bitonického řazení, kde každá fáze odpovídala zvětšování velikosti řazených skupin. Vnitřní cyklus pak iteroval přes jednotlivé průchody (passes) v rámci dané fáze, což zajistilo správné porovnání a výměnu prvků podle pravidel bitonického algoritmu. Pro každou kombinaci fáze a průchodu pak byl provolán kód jádra s příslušnými argumenty.
Z takto seřazeného pole byla jednoduše vybrána hodnota mediánu. Jeho absolutní odchylky od každé hodnoty v poli bylo možno opět provést paralelně na GPU. Zbytek výpočtu MAD probíhal již analogicky jako u výpočtů na CPU.

3.7 Vykreslení

Vykreslení bylo implementováno za pomoci připravených utilit dostupných zde: `https://github.com/SmartCGMS/common.git`, využity byly knihovny `SVGRenderer.h` a `Drawing.h` pro vykreslení grafických prvků. Implementovaná funkce `plot_graph` pak slouží k vykreslení grafu ve formátu SVG.
Funkce nejprve ověřuje platnost vstupních dat (bodů pro osy X a Y). Následně připraví plátno pro vykreslení grafu a spočítá minimální a maximální hodnoty pro osy X a Y, přičemž přidá okraje a upraví rozsah os pro lepší zobrazení. Poté vykreslí osy a mřížku a přidá značky a popisky na osy. Pro každou sérii dat (sadu bodů) vytvoří křivku, kterou přidá do grafu s definovanými barvami a vlastnostmi. Funkce také přidá název grafu, popisky os a legendu pro jednotlivé linie. Na závěr vygeneruje SVG kód, který uloží do souboru.

4 Výsledky a diskuze

Tato sekce prezentuje výsledky výpočtů koeficientu variance a mediánu absolutní odchylky získaných z dat akcelerometrů, a srovnává jejich efektivitu a konzistenci při použití různých přístupů (sériový, vektorizovaný, paralelní a GPU kód).

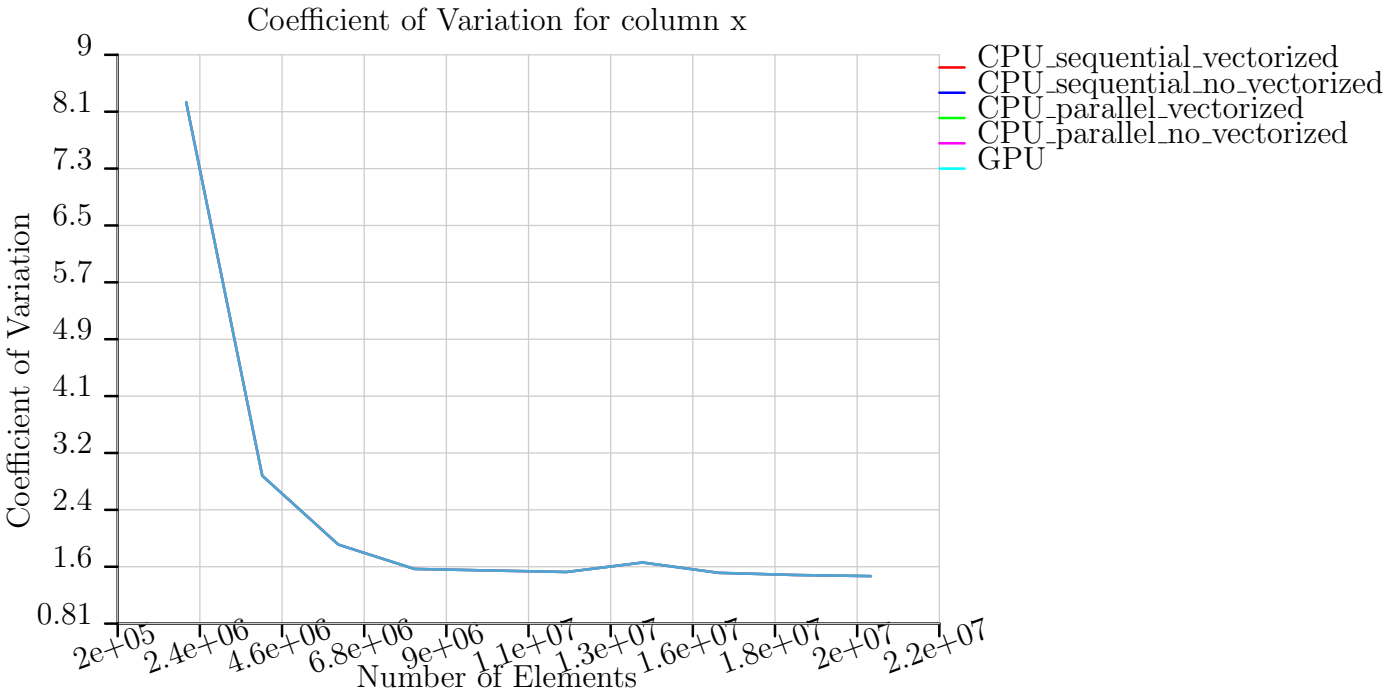
Na základě konzultace se cvičícím nebyly v této části zahrnuty grafy pro všech 16 pacientů a všechny tři složky dat. Místo toho byly vybrány reprezentativní vzorky, které dostatečně vystihují obecné trendy zkoumaného problému, čímž bylo minimalizováno riziko redundance a zajištěna přehlednost prezentovaných výsledků. Všechny prezentované grafy byly generovány na základě dat ze sloupce **X** prvního pacienta (soubor `ACC_001.csv`), přičemž granularita osy x byla nastavena na hodnotu 10. To znamená, že velikost dat, nad kterými byly prováděny výpočty, se zvyšovala po jedné desetině původní velikosti.

Obrázky 4 a 5 ilustrují konvergenci vypočítaných statistických ukazatelů v závislosti na velikosti datové množiny. Na obou grafech je jasně patrné, že hodnoty získané jednotlivými implementovanými výpočetními variantami jsou shodné, což potvrzuje správnost a konzistenci implementace. U koeficientu variability je zřejmá hladká a stabilní konvergence k finální hodnotě, což je očekávané vzhledem k jeho statistické povaze (při malé množině dat je variance náhodného vzorku vyšší, a s rostoucím množstvím dat se hodnota stabilizuje). Naopak u mediánu absolutní odchylky tento trend není patrný, což je způsobeno specifiky této statistiky – medián absolutní odchylky je silně závislý na konkrétní struktuře a charakteristikách vybraných datových podmnožin. Tato variabilita tedy odráží citlivost této veličiny na fluktuace v datech.

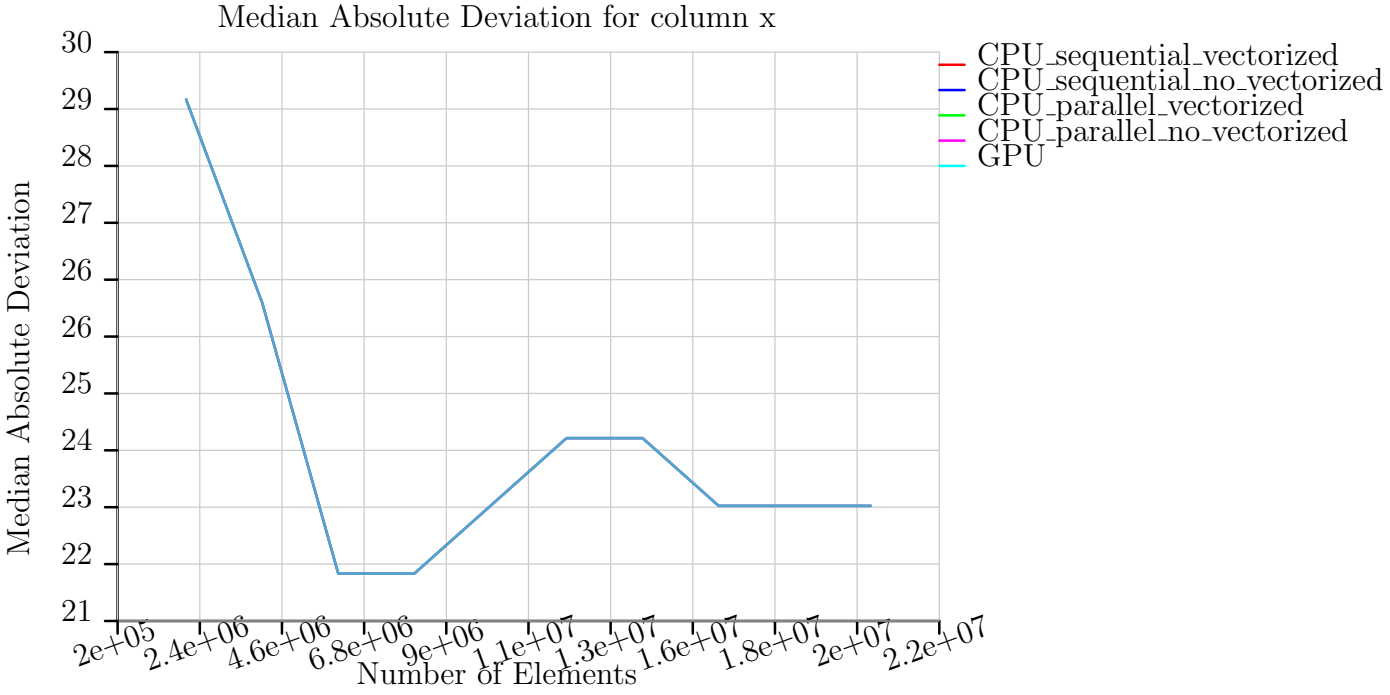
Identický experiment byl proveden i pro *single precision* aritmetiku (float). Ve výsledných grafech se však rozdíly v přesnosti neprojevily výrazně, a proto je pro porovnání v tabulce 1 uveden koeficient variance vypočtený pro oba datové typy (double a float). Pro medián absolutní odchylky (MAD) se výsledky mezi těmito dvěma typy nelišily.

Column	**Double Coefficient of Variance**	**Float Coefficient of Variance**
x	1.48529	1.50418
y	36.4063	35.8386
z	1.49689	1.51202

Tabulka 1: Hodnoty koeficientu variance pro různé desetinné datové typy



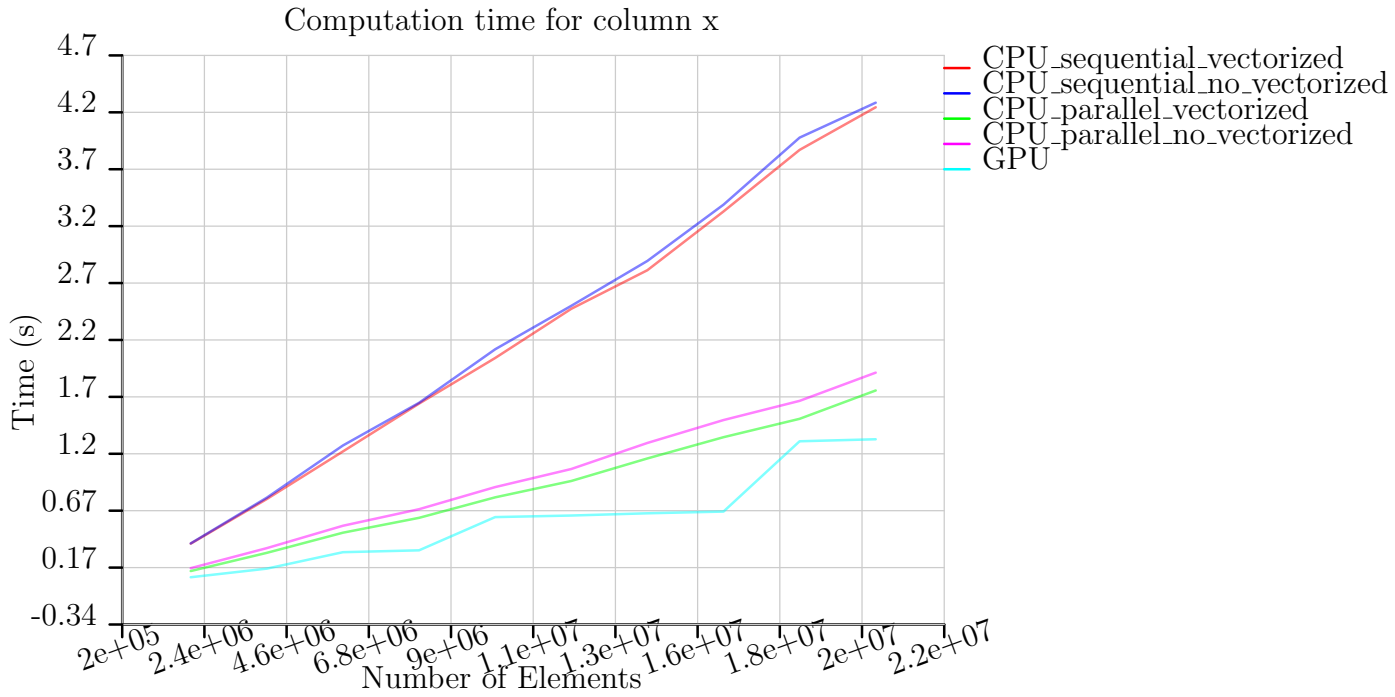
Obrázek 4: Konvergence hodnot koeficientu variance



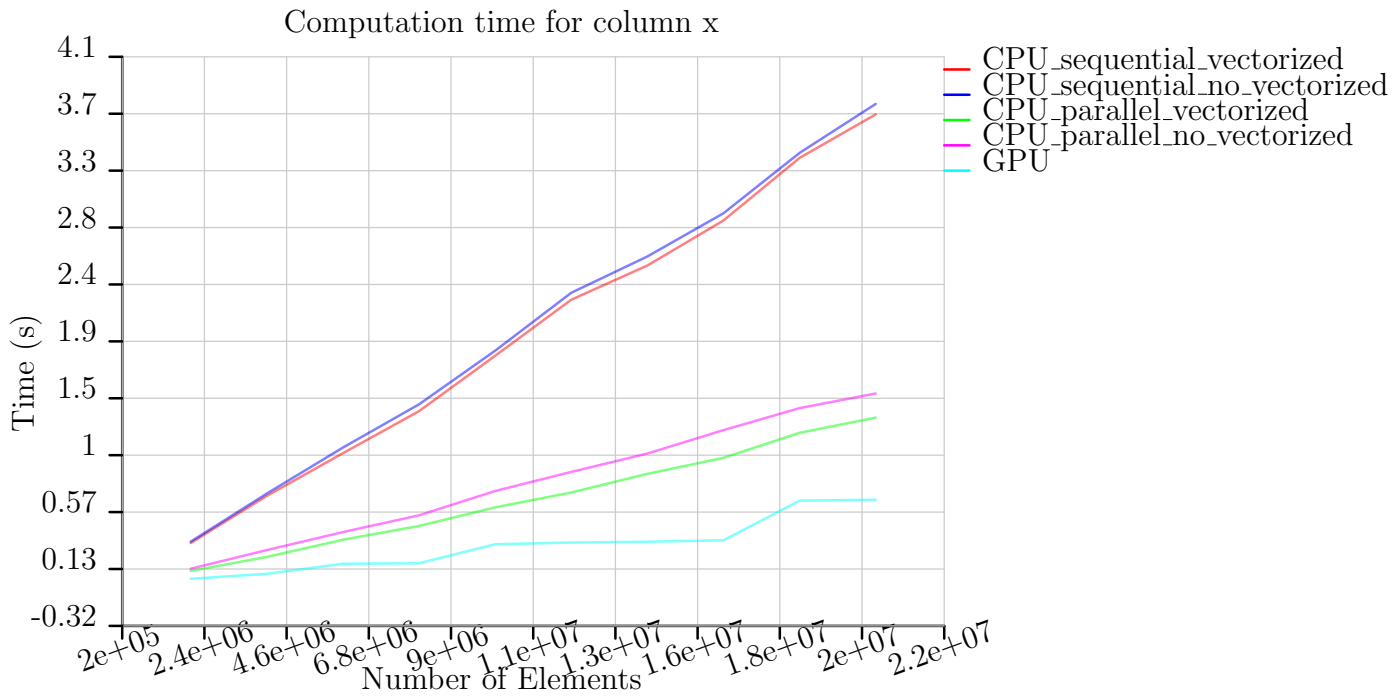
Obrázek 5: Konvergence hodnot mediánu absolutní odchylky

Analýza časové náročnosti výpočtů byla rovněž provedena pro oba datové typy. Naměřené hodnoty jsou zobrazeny na Obrázcích 6 a 7. Z těchto grafů je zřejmé, že nejrychlejší výkon byl dosažen při použití GPU, zatímco nejpomalejší běh vykazoval sekvenční nevektorizovaný algoritmus. U křivky reprezentující GPU výpočet je patrný

schodovitý trend, což je způsobeno povahou řadicího algoritmu, který vyžaduje, aby bylo vstupní pole vždy zarovnáno na nejbližší vyšší mocninu čísla 2.
Dále je z grafů jasně patrný přínos vektorizace: vektorizovaná varianta vykazuje vždy lepší časový výkon než její nevektorizovaná obdoba. Při použití *single precision* (float) se rozdíl mezi vektorizovanými a nevektorizovanými variantami stává výraznějším, přičemž zejména u paralelního výpočtu je tento efekt patrnější. Tento jev lze vysvětlit tím, že snížená přesnost u *single precision* umožňuje ukládat do vektorových registrů větší množství čísel najednou, což činí vektorizaci efektivnější. Obecně lze také konstatovat, že běh programu je u *single precision* rychlejší.
Omezený přínos vektorizace je důsledkem toho, že vektorizace není aplikována na celý řadicí algoritmus, přičemž právě samotný řadicí proces je výpočetně nejnáročnější částí výpočtů. Lze tedy očekávat, že implementace jiného řadicího algoritmu, který by byl lépe přizpůsoben pro vektorizaci, by vedla k výraznějšímu zrychlení.



Obrázek 6: Časová náročnost výpočtů při postupném navyšování datového rozsahu pro všechny realizované varianty, *double precision*



Obrázek 7: Časová náročnost výpočtů při postupném navyšování datového rozsahu pro všechny realizované varianty, *single precision*

Závěrem je v tabulce 2 uveden kompletní přehled vypočtených statistických ukazatelů (MAD a CV) pro všechny pacienty a jejich odpovídající množiny hodnot (**X, Y, Z**). Tato tabulka poskytuje souhrn výsledků, který umožňuje detailní srovnání mezi jednotlivými pacienty a různými složkami dat.

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

Tabulka 2: Hodnoty MAD a CV pro všechny pacienty a množiny hodnot

5 Závěr

Tato semestrální práce se zaměřila na analýzu a implementaci výpočtů nad rozsáhlými datovými sadami s využitím paralelních technik na CPU i GPU. Byly důkladně zkoumány různé přístupy k řazení dat, výpočtu součtů, koeficientů variability a mediánu absolutních odchylek, přičemž byly zohledněny jak teoretické aspekty, tak praktická implementace. Výsledky experimentů ukázaly, že výkonnostní rozdíly mezi CPU a GPU implementacemi nejsou dány pouze hrubým výpočetním výkonem, ale také efektivitou zvolených algoritmů a způsobem práce s paměťovou hierarchií.

Prezentované výsledky jednoznačně potvrzují správnost a konzistenci implementovaných výpočtů statistických ukazatelů. Analýza časové náročnosti odhalila, že GPU implementace dosahovala nejlepších časů díky vysokému stupni paralelizace, zatímco paralelní CPU varianty také vykazovaly výrazné zlepšení výkonu oproti sekvenčním implementacím.

Pro budoucí rozšíření práce se nabízí několik směrů optimalizace. Na straně CPU by mohlo být přínosné zavedení efektivnějšího algoritmu pro řazení, ideálně s plným využitím vektorových instrukcí, což by mohlo výrazně snížit dobu zpracování této klíčové operace. U GPU implementací by se výzkum měl zaměřit na optimalizaci

výpočetních jader, zejména těch určených pro součet a další redukční operace. Významným zlepšením by také mohlo být zavedení vlastního paměťového alokátoru pro optimální zarovnání dat, čímž by bylo možné dále zlepšit efektivitu paměťového přístupu.

6 Uživatelská příručka

Pro správnou kompilaci a bezproblémový běh programu je nezbytné mít následující prostředí:

- CMake ve verzi 3.21 nebo novější
- Překladač **MSVC** 2022
- Standardní C++ knihovny odpovídající standardu C++17 nebo novější

Nástroj CMake lze pohodlně stáhnout z oficiálních webových stránek na adrese `https://cmake.org`. Standardní knihovny C++ jsou součástí většiny moderních C++ překladačů a obvykle je není třeba instalovat samostatně.
Pro správné sestavení aplikace je nezbytné využít **64-bitový** překladač, a to z důvodu způsobu načítání dat. V 32-bitovém režimu by mohl být limit 4 GB adresovatelné paměti nedostatečný pro správné fungování aplikace.
Pro kompilaci v režimu *single precision* je možné upravit první příkaz volání nástroje *cmake* přidáním přepínače `-DCMAKE_CXX_FLAGS="-D_FLOAT"`. Alternativně lze upravit soubor `CMakeLists.txt` tím, že se odkomentuje řádek 7 a zakomentuje řádek 8, jak je ukázáno níže:

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} /W4 /arch:AVX2 /D_FLOAT")
#set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} /W4 /arch:AVX2")
```

Spuštění programu z příkazové řádky

Program lze spustit z příkazové řádky s různými volbami a parametry, které modifikují běh aplikace. Následuje popis možností, které je možné použít při spuštění.

Povinné argumenty:

- `--input`: Cesta k souboru nebo adresáři obsahujícímu data ve formátu `.csv`. Toto je jediný povinný argument.
  - Příklad: `--input data/ACC_001.csv`

Volitelné argumenty:

- `--output`: Cesta k adresáři pro uložení výsledků. Tento argument je volitelný, a pokud není uveden, výstup bude uložen do adresáře `results`.
  - Příklad: `--output results`
- `--repetitions`: Počet opakování experimentu (do výsledku uložen medián z těchto opakování). Výchozí hodnota je 1.
  - Příklad: `--repetitions 10`
- `--num_partitions`: Počet dělení dat. Výchozí hodnota je 1.
  - Příklad: `--num_partitions 4`
- `--gpu`: Určuje typ zařízení - buď CPU nebo GPU. Defalutně program poběží na CPU.
  - Použití: `--gpu`
- `--parallel`: Určuje, zda se výpočty mají provádět paralelně nebo sekvenčně. Pokud není tento argument uveden, program použije výchozí nastavení sekvenčního běhu.
  - Použití: `--parallel`
- `--vectorized`: Aktivuje manuální vektorizaci s využitím AVX2.
  - Použití: `--vectorized`
- `--all_variants`: Spustí všechny varianty algoritmu; sériové spuštění, sériové spuštění s vektorizací, paralelní spuštění, paralelní spuštění s vektorizací, spuštění na GPU.
  - Použití: `--all_variants`

A Zajímavé úkázky kódu pro výpočty na CPU

```
#ifdef _FLOAT
using real = float;
#define LOAD _mm256_loadu_ps
#define STORE _mm256_storeu_ps
#define SETZERO _mm256_setzero_ps
#define ADD _mm256_add_ps
#define SUB _mm256_sub_ps
#define MUL _mm256_mul_ps
#define ANDNOT _mm256_andnot_ps
#define STRIDE __m256
#define SET1 _mm256_set1_ps
#define str_to_real std::strtof
#else
using real = double;
#define LOAD _mm256_loadu_pd
#define STORE _mm256_storeu_pd
#define SETZERO _mm256_setzero_pd
#define ADD _mm256_add_pd
#define SUB _mm256_sub_pd
#define MUL _mm256_mul_pd
#define ANDNOT _mm256_andnot_pd
#define STRIDE __m256d
#define SET1 _mm256_set1_pd
#define str_to_real std::strtod
#endif

void abs_diff_calc(std::vector<real> &arr, std::vector<real> &abs_diff
    , real median, size_t n,
                   const bool is_vectorized, const execution_policy &
                       policy) {
    size_t i = 0;
    if (is_vectorized) {

        // broadcast median to all elements of the vector
        auto med = SET1(median);

        // create a mask with the sign bit cleared
        auto sign_mask = SET1(-0.0);

        size_t step = sizeof(STRIDE) / sizeof(real);

        std::vector<size_t> indices;
        for (i = 0; i <= n - step; i += step) {
```

```
42              indices.push_back(i);
43          }
44
45          // process 4 elements at a time using AVX2
46          std::visit([&](auto &&exec_policy) {
47              std::for_each(exec_policy, indices.begin(), indices.end(),
                    [&](size_t i) {
48                  auto vec = LOAD(&arr[i]); // load 4 elements
49                  auto diff = SUB(vec, med); // subtract median from
                        elements
50                  auto abs_diff_vec = ANDNOT(sign_mask, diff); // clear
                        the sign bit
51                  STORE(&abs_diff[i], abs_diff_vec); // store result
52              });
53          }, policy.get_policy());
54      }
55
56      // process remaining elements
57      int j = static_cast<int>(i);
58      std::visit([&](auto &&exec_policy) {
59          std::for_each(exec_policy, arr.begin() + j, arr.begin() +
                static_cast<int>(n), [&](real &value) {
60              abs_diff[&value - &arr[0]] = std::abs(value - median);
61          });
62      }, policy.get_policy());
63  }
64
65  void sum_and_copy(const std::vector<real> &arr, std::vector<real> &
        halve_arr,
66                    size_t start, size_t size, real &sum, real &sum2,
67                    const execution_policy &policy) {
68
69      size_t max_num_threads = std::thread::hardware_concurrency();
70      size_t chunk_size = size / max_num_threads;
71
72      // local sums and sum of squares for each thread
73      std::vector<real> local_sums(max_num_threads, 0.0);
74      std::vector<real> local_sums2(max_num_threads, 0.0);
75
76      // vector to hold the chunk indices for each thread
77      std::vector<size_t> chunk_indices(max_num_threads);
78      std::iota(chunk_indices.begin(), chunk_indices.end(), 0); // pre-
            calculate chunk indices
79
80      std::visit([&](auto &&exec_policy) {
```

```
81          std::for_each(exec_policy, chunk_indices.begin(),
                chunk_indices.end(),
82                        [&](const auto &chunk_id) {
83                            size_t start_chunk = chunk_id * chunk_size;
84                            size_t end_chunk = (chunk_id ==
                                max_num_threads - 1) ? size : start_chunk
                                 + chunk_size;
85
86                            for (size_t i = start_chunk; i < end_chunk;
                                i++) {
87                                real val = arr[i + start];  // get value
                                     from original array
88                                halve_arr[i] = val;  // copy value to
                                    halve array
89
90                                // accumulate sum and sum of squares for
                                     this chunk
91                                local_sums[chunk_id] += val;
92                                local_sums2[chunk_id] += val * val;
93                            }
94                        });
95      }, policy.get_policy());
96
97      // combine results from all threads - reduction of local sums
98      sum += std::accumulate(local_sums.begin(), local_sums.end(),
            static_cast<real>(0.0));
99      sum2 += std::accumulate(local_sums2.begin(), local_sums2.end(),
            static_cast<real>(0.0));
100 }
101
102 void sum_and_copy_vec(const std::vector<real> &arr, std::vector<real>
        &halve_arr,
103                       size_t start, size_t size, real &sum, real &sum2
                              ,
104                       const execution_policy &policy) {
105
106     size_t max_num_threads = std::thread::hardware_concurrency();
107     size_t chunk_size = size / max_num_threads;
108     size_t stride = sizeof(STRIDE) / sizeof(real);
109
110     // local sums and sum of squares for each thread
111     std::vector<real> local_sums(max_num_threads, static_cast<real
            >(0.0));
112     std::vector<real> local_sums2(max_num_threads, static_cast<real
            >(0.0));
```

```
113
114     // vector to hold the chunk indices for each thread
115     std::vector<size_t> chunk_indices(max_num_threads);
116     std::iota(chunk_indices.begin(), chunk_indices.end(), 0); // pre-
            calculate chunk indices
117
118     std::visit([&](auto &&exec_policy) {
119         std::for_each(exec_policy, chunk_indices.begin(),
                chunk_indices.end(),
120                       [&](const auto &chunk_id) {
121                           size_t start_chunk = chunk_id * chunk_size;
122                           size_t end_chunk = (chunk_id ==
                                max_num_threads - 1) ? size : start_chunk
                                 + chunk_size;
123
124                           auto vec_sum = SETZERO();
125                           auto vec_sum2 = SETZERO();
126
127                           for (size_t i = start_chunk; i < end_chunk;
                                i += stride) {
128                               auto vec_vals = LOAD(&arr[i + start]);
                                    // load elements
129                               STORE(&halve_arr[i], vec_vals); // store
                                     elements
130                               vec_sum = ADD(vec_sum, vec_vals); //
                                    accumulate sum
131                               vec_sum2 = ADD(vec_sum2, MUL(vec_vals,
                                    vec_vals)); // accumulate sum of
                                    squares
132                           }
133
134                           // horizontal sum - sum of vector elements
135                           std::vector<real> temp_sum(stride,
                                static_cast<real>(0.0));
136                           std::vector<real> temp_sum2(stride,
                                static_cast<real>(0.0));
137                           STORE(temp_sum.data(), vec_sum);
138                           STORE(temp_sum2.data(), vec_sum2);
139                           for (size_t k = 0; k < stride; k++) {
140                               local_sums[chunk_id] += temp_sum[k];
141                               local_sums2[chunk_id] += temp_sum2[k];
142                           }
143
144                           // handle any remaining elements (less than
                                stride) in the tail
```

```
145                             for (size_t i = end_chunk - (end_chunk %
                                    stride); i < end_chunk; i++) {
146                                 real val = arr[i + start];  // get value
                                        from original array
147                                 halve_arr[i] = val;  // copy value to
                                        halve array
148
149                                 // accumulate sum and sum of squares for
                                        this chunk
150                                 local_sums[chunk_id] += val;
151                                 local_sums2[chunk_id] += val * val;
152                             }
153                         });
154     }, policy.get_policy());
155
156     // combine results from all threads - reduction of local sums
157     sum += std::accumulate(local_sums.begin(), local_sums.end(),
            static_cast<real>(0.0));
158     sum2 += std::accumulate(local_sums2.begin(), local_sums2.end(),
            static_cast<real>(0.0));
159 }
```

Úryvek kódu 2: Útržky kódu pro paralelní a vektorizované výpočty

B OpenCL kernel kód

```
1     __kernel void abs_diff_calc(__global double *arr, double median) {
2         int i = get_global_id(0);
3         arr[i] = fabs(arr[i] - median);
4     }
5
6     __kernel void vector_sums(
7         __global const double* input,
8         __global double* partial_sums,
9         __global double* partial_sums_squares,
10        __local double* local_sums,
11        __local double* local_sums_squares) {
12
13        uint global_id = get_global_id(0);
14        uint local_id = get_local_id(0);
15        uint group_size = get_local_size(0);
16        uint group_id = get_group_id(0);
17
18        // initialize local memory
```

```
19          double value = (global_id < get_global_size(0)) ? input[
                global_id] : 0.0;
20          local_sums[local_id] = value;
21          local_sums_squares[local_id] = value * value;
22
23          // synchronize to ensure all work-items have written to local
                memory
24          barrier(CLK_LOCAL_MEM_FENCE);
25
26          // reduction within the workgroup
27          for (uint stride = group_size / 2; stride > 0; stride /= 2) {
28              if (local_id < stride) {
29                  local_sums[local_id] += local_sums[local_id + stride];
30                  local_sums_squares[local_id] += local_sums_squares[
                        local_id + stride];
31              }
32              barrier(CLK_LOCAL_MEM_FENCE); // ensure updates are
                    visible to all work-items
33          }
34
35          // write the results of this workgroup to the partial sums
                arrays
36          if (local_id == 0) {
37              partial_sums[group_id] = local_sums[0];
38              partial_sums_squares[group_id] = local_sums_squares[0];
39          }
40      }
41
42      __kernel void bitonic_sort_kernel(__global double *arr, const uint
            stage, const uint pass_of_stage) {
43          uint thread_id = get_global_id(0);
44          uint pair_distance = 1 << (stage - pass_of_stage); // distance
                between elements to compare
45          uint block_width = 2 * pair_distance; // width of the block
                being compared
46          uint temp;
47          bool compare_result;
48          uint left_id = (thread_id & (pair_distance - 1)) + (thread_id
                >> (stage - pass_of_stage)) * block_width;
49          uint right_id = left_id + pair_distance;
50
51          // get the elements to compare
52          double left_element = arr[left_id];
53          double right_element = arr[right_id];
54
```

```
55          uint same_direction_block_width = thread_id >> stage;
56          uint same_direction = same_direction_block_width & 0x1; //
                check if the block is in the same direction
57
58          // swap the elements if they are not in the same direction
59          temp = same_direction ? right_id : temp;
60          right_id = same_direction ? left_id : right_id;
61          left_id = same_direction ? temp : left_id;
62
63          compare_result = (left_element < right_element);
64
65          // swap the elements if they are not in the correct order
66          double greater = compare_result ? right_element : left_element
                ;
67          double lesser = compare_result ? left_element : right_element;
68
69          arr[left_id] = lesser;
70          arr[right_id] = greater;
71      }
```

Úryvek kódu 3: Zdrojové kódy jádra