Zpracování medicínských dat pro výpočet typického dne pacienta s diabetem

Semestrální práce z předmětu KIV/PPR na ZČU v Plzni

Bc. Jiří Winter

31. ledna 2026

Obsah

Kapitola 1

Úvod a Struktura projektu

Cílem této práce je efektivní zpracování velkých objemů medicínských dat (Dexcom, HR, BVP) s různou frekvencí vzorkování (od 5 minut až po 15 ms). Aplikace počítá mediány naměřených hodnot v čase, provádí redukci dat (downsampling) a využívá k tomu metody paralelizace na CPU (OpenMP, SIMD) a GPU (OpenCL).

1.1 Medicínská data

Práce umí zpracovat tři typy dat:

- **Dexcom Data:** Glukózová data s vzorkováním po 5 minutách (288 záznamů/den).
- **HR Data:** Srdeční frekvence s vzorkováním po 1 sekundě (86,400 záznamů/den).
- **BVP Data:** Biovoltážní signál s vzorkováním po 15 ms (5,760,000 záznamů/den).

1.2 Adresářová struktura

Projekt je rozdělen do modulů podle zodpovědnosti:

- **include/**
  - `DataReader.h` - Deklarace tříd pro načítání a konverzi dat.
  - `ReadDexcomData.h` - Datové struktury pro uložení záznamů (`DexcomData`).
- **src/**
  - `DataReader.cpp` - Implementace parseru CSV a převodu/normalizace časových razítek.
  - `CPUSequential.cpp` - Referenční sekvenční implementace.
  - `CPUParallel.cpp` - Implementace využívající OpenMP a vektorizaci.
  - `GPUParallel.cpp` - Host kód pro OpenCL (správa bufferů a kernelů).
  - `kernel.cl` - OpenCL kernely pro běh na grafické kartě.
  - `main.cpp` - Vstupní bod, benchmarky a validace výsledků.

1.3 Spuštění programu

Program je určen pro běh z příkazové řádky. Uživatelská příručka je uvedena v Příloze A.

Kapitola 2

Algoritmus zpracování dat

Algoritmus zpracování dat je identický pro všechny tři implementační metody (sekvenční, paralelní i GPU), liší se pouze způsobem, jakým jsou jednotlivé kroky vykonávány.

Algoritmus pracuje se třemi dimenzemi dat:

1. **Time Slots (Časová okna):** Diskrétní časové úseky během dne (např. 00:00:00, 00:00:15, ...).
2. **Patients (Pacienti):** Jednotliví měření lidé.
3. **Days (Dny):** Naměřené hodnoty pro daného pacienta a daný čas v různých dnech (6-8 dní měření).

Celý proces se skládá ze tří hlavních fází:

2.1 Fáze 1: Medián pro každého pacienta

V této fázi se pro každou kombinaci **(TimeSlot, Patient)** vezmou všechna naměřená data za všechny dny (6-8 hodnot) a vypočítá se z nich medián.

- **Vstup:** Matice [TimeSlots × Patients × Days]
- **Výstup:** Matice [TimeSlots × Patients]
- **Operace:** Pro každého pacienta P v čase T se vezme vektor dní $D_1 \dots D_N$, seřadí se a vybere se střední hodnota.

2.2 Fáze 2: Medián napříč pacienty (Grand Median)

Následně se pro každý **TimeSlot** vezmou výsledné mediány všech pacientů z předchozí fáze a vypočítá se jeden reprezentativní medián pro tento časový okamžik.

- **Vstup:** Matice [TimeSlots × Patients]
- **Výstup:** Vektor [TimeSlots]
- **Operace:** Pro každý čas T se vezme vektor pacientů $P_1 \dots P_M$, seřadí se a vybere se střední hodnota.

2.3 Fáze 3: Downsampling (Redukce časových oken)

V případě potřeby (např. pro vizualizaci nebo redukci šumu) se provádí snížení vzorkovací frekvence. Původní jemná časová okna se slučují do větších bloků.

- **Příklad:** Redukce z 15ms samplingu na 5 minut.
- **Vstup:** Vektor [TimeSlots] (např. 5 760 000 prvků)
- **Výstup:** Vektor [ReducedTimeSlots] (např. 288 prvků)
- **Operace:** Pro každé nové (větší) okno se vezme úsek původních hodnot, seřadí se a vypočítá medián.

Kapitola 3

Načítání a Reprezentace dat

Pro efektivní zpracování na GPU je klíčové převést data z objektové reprezentace (vektor objektů) do lineárního pole (Flat Data Layout).

3.1 Flat Data Layout

Data jsou uložena v jediném spojitém vektoru `std::vector<float> flat_data`. Indexace probíhá podle vzorce:

$$Index = TimeSlot \times (NumPatients \times NumDays) + Patient \times NumDays + Day \quad (3.1)$$

Tento layout zajišťuje, že data pro jednoho pacienta v rámci jednoho časového slotu (např. 8 dní měření) leží v paměti bezprostředně za sebou, což je optimální pro SIMD instrukce a cache procesoru.

3.2 Parsování časových razítek

Pro každý typ dat je nutné převést textové časové razítko na index časového slotu pro zarovnání dat. Pro každou frekvenci vzorkování je implementována samostatná funkce spolu s funkcí pro normalizaci času:

- **Frekvence 5 minut:** minutes_since_midnight()
- **Frekvence 1 sekunda:** seconds_since_midnight()
- **Frekvence 15 ms:** units_15ms_since_midnight()

3.2.1 Frekvence 15ms

U vstupních dat je nutné převést textový čas na index slotu. Implementace využívá `std::from_chars` pro maximální rychlost a řeší i normalizaci milisekund.

```
std::optional<uint32_t> DataConverter::units_15ms_since_midnight(const std::
    string_view& ts) {
    // Rychlá cesta: předpokládáme formát "YYYY-MM-DD HH:MM:SS.ms"
    if (ts.size() >= 23) {
        // ... parsování HH, MM, SS ...

        // Helper funkce pro parsování milisekund
        auto to_int = [](std::string_view s) -> std::optional<int> {
            int out{};
```

```
            auto res = std::from_chars(s.data(), s.data() + s.size(), out);
            if (res.ec == std::errc{}) return out;
            return std::nullopt;
        };

        int digit_count = 0;
        // ... spočítání číslic v milisekundách -> digit_count

        // Parsování milisekund a normalizace
        if (digit_count > 0) {
            auto sv_ms = ts.substr(20, digit_count);
            auto oms = to_int(sv_ms);
            if (oms) {
                ms = *oms;
                // ISO 8601 pravidlo: .5 je 500ms, .05 je 50ms
                if (digit_count == 1) ms *= 100;
                else if (digit_count == 2) ms *= 10;
            }
        }

        // Výpočet celkových milisekund a slot indexu
        uint32_t total_ms = (*oh * 3600 + *om * 60 + *os) * 1000 + ms;
        return total_ms / 15;
    }
    return std::nullopt;
}
```

Listing 3.1: Metoda pro výpočet indexu slotu z času (DataReader.cpp)

3.3 Handling prázdných hodnot (Padding)

Jelikož jsou v měřeních prázdné úseky, kdy data chybí, používáme padding a vyplníme tyto prázdná data konstantou. Pro BVP data, která mohou nabývat záporných hodnot, nelze použít `-1.0f` a je tedy použito nejmenší možný float.

```
// Používáme velmi malé číslo, aby se nepletlo s reálnými daty
constexpr float EMPTY_VALUE = -1.0e30f;
constexpr float EPSILON = 100.0f; // Tolerance pro porovnání
```

Listing 3.2: Definice paddingu

Kapitola 4

Sekvenční zpracování (CPU)

Referenční implementace slouží pro ověření správnosti paralelních algoritmů. Iteruje přes časové sloty, pacienty a dny sekvenčně s potlačením automatické vektorizace/paralelizace kompilátorem.

```
// Při implemetaci a benchmarku byl použit clang-kompilátor
#pragma clang loop vectorize(enable) interleave(disable)
for (...) {}
```

Listing 4.1: Definice paddingu

4.1 Ukázka výpočtu mediánu

Následující kód ukazuje výpočet mediánu, který filtruje padding a řeší případ, kdy se sudým počtem prvků. V takovém případě je medián průměrem dvou prostředních hodnot.

```
inline float calculateMedian(const std::vector<float>& data) {
    if (data.empty()) return EMPTY_VALUE;

    // Filtrace: Vytvoříme buffer jen pro platná data
    // Ignorujeme EMPTY_VALUE padding, ale bereme i záporná čísla (BVP)
    std::vector<float> valid_data;
    valid_data.reserve(data.size());

    for (float val : data) {
        // Podmínka pokud je hodnota větší než EMPTY_VALUE + odchylka floatu
        if (val > EMPTY_VALUE + EPSILON) {
            valid_data.push_back(val);
        }
    }

    if (valid_data.empty()) return EMPTY_VALUE;

    // Samotný výpočet mediánu (nth_element)
    size_t n = valid_data.size() / 2;
    std::nth_element(valid_data.begin(), valid_data.begin() + n, valid_data.
    end());
    float median = valid_data[n];

    // Průměrování pro sudý počet prvků
    // Pokud je počet prvků sudý, medián je průměr n-tého a (n-1)-tého prvku
    .
    // nth_element nám zaručil n-tý, pro (n-1)-tý musíme najít maximum v lev
    é části.
    if (valid_data.size() % 2 == 0) {
```

```
27          auto max_it = std::max_element(valid_data.begin(), valid_data.begin
    () + n);
28          median = (*max_it + median) / 2.0f;
29      }
30      return median;
31  }
```

Listing 4.2: Sekvenční výpočet mediánu s filtrací (CPUSequential.cpp)

4.2 Ukázka sekvenčního algoritmu

Následující kód ukazuje hlavní smyčku sekvenčního výpočtu mediánů a redukce časových slotů. Nejprve se spočítají mediány pro všechny pacienty napříč dny, poté se spočítá medián pro každé časové okno napříč pacienty a poté se provede downsampling. Výsledky jsou uloženy do `result_medians_seq` a `updated_timeslots_seq`.

```
1  void DexcomData::processSequential(int32_t num_wanted_time_slots) {
2  #pragma clang loop vectorize(enable) interleave(disable)
3      // smyčka napříc časovými sloty
4      for (auto i = 0; i < num_time_slots; i++) {
5          std::vector<float> patients_medians;
6          patients_medians.reserve(num_patients);
7          // smyčka napříč pacienty
8          for (auto j = 0; j < num_patients; j++) {
9              std::vector<float> vectorData;
10             vectorData.reserve(num_days);
11             auto data = getPatientDataPtr(i, j);
12             for (auto k = 0; k < num_days; k++) {
13                 vectorData.push_back(*(data+k));
14             }
15             // medián pro pacienta j v časovém slotu i napříč dny
16             float median = calculateMedian(vectorData);
17             patients_medians.push_back(median);
18         }
19         // medián pro časový slot i napříč pacienty
20         float median = calculateMedian(patients_medians);
21         result_medians_seq.push_back(median);
22     }
23     // downsampling časových slotů, pokud je požadováno
24     // num_wanted_time_slots je počet cílových slotů po redukci
25     if (num_wanted_time_slots > 0) {
26         updated_timeslots_seq.reserve(num_wanted_time_slots);
27         std::vector<float> vectorData;
28         for (int i = 0; i < num_wanted_time_slots; ++i) {
29             // Výpočet rozsahu pro aktuální cílový slot
30             // Rozdíl v end_idx a start_idx určuje počet prvků ke zpracování
31             size_t start_idx = (static_cast<size_t>(i) * num_time_slots) /
    num_wanted_time_slots;
32             size_t end_idx = (static_cast<size_t>(i + 1) * num_time_slots) /
     num_wanted_time_slots;
33
34             if (end_idx > num_time_slots) end_idx = num_time_slots;
35             if (start_idx >= end_idx) {
36                 updated_timeslots_seq.push_back(EMPTY_VALUE);
37                 continue;
38             }
39
40             for (size_t k = start_idx; k < end_idx; ++k) {
41                 vectorData.push_back(result_medians_seq[k]);
```

```
42              }
43              // výpočet mediánu pro aktuální cílový slot
44              float median = calculateMedian(vectorData);
45              updated_timeslots_seq.push_back(median);
46          }
47      }
48  }
```

Listing 4.3: Sekvenční výpočet mediánu s filtrací (CPUSequential.cpp)

4.3 Výběr algoritmu: std::nth_element

Pro výpočet mediánu byla zvolena funkce `std::nth_element` namísto klasického řazení `std::sort`. Důvodem je algoritmická složitost:

- `std::sort`: Složitost $\mathcal{O}(N \log N)$, seřadí celý vektor.
- `std::nth_element`: Složitost $\mathcal{O}(N)$ v průměrném případě. Používá algoritmus Quickselect, který pouze přehází prvky tak, aby na n-té pozici byl prvek, který by tam patřil v seřazené posloupnosti.

Jelikož nás zajímá pouze jedna hodnota uprostřed (medián), je plné řazení zbytečné a `std::nth_element` je výrazně rychlejší.

Kapitola 5

Paralelizace a SIMD Optimalizace

Pro zvýšení výkonu na moderních procesorech byla využita kombinace vláknové paralelizace (OpenMP) a vektorizace (SIMD). V rámci projektu byly implementovány a změřeny dvě varianty paralelního zpracování na CPU. Obě využívají knihovnu OpenMP pro distribuci práce mezi vlákna, ale liší se v úrovni optimalizace výpočtu uvnitř jednotlivých vláken. Jedna metoda používá běžné sekvenční výpočty, zatímco druhá využívá ručně optimalizovaný kód pro výpočet mediánu z pevného počtu 8 hodnot pomocí SIMD instrukcí.

1. **Parallel (Bez SIMD):** Jednoduchá paralelizace smyčky, která pro výpočet mediánu používá alokaci `std::vector` a funkci `std::nth_element`.
2. **Parallel + SIMD:** Ručně optimalizovaná varianta pomocí SIMD instrukcí, která používá řazení přímo v registrech procesoru - Sorting Network.

5.1 Paralelizace algoritmu

Data pro různé časové sloty jsou na sobě nezávislá, takže může být snadno paralelizována. Každý časový slot tak může být vypočítán paralelně. Paralelizace probíhá ve dvou fázích:

1. **Fáze 1:** Výpočet mediánů pro jednotlivé TimeSloty. Smyčka napříč časovými sloty je rozdělena mezi vlákna pomocí `#pragma omp parallel for`. Každé vlákno zpracovává svůj blok časových slotů.
2. **Fáze 2:** Redukce (Downsampling) časových oken. Paralelizace přes cílové (redukované) sloty.

5.2 Varianta 1: Paralelizace bez SIMD

Tato metoda (reprezentovaná funkcí `processParallelCPUNonVectorized`) paralelizuje vnější smyčku přes časové sloty pomocí `#pragma omp parallel for`. Uvnitř vlákna se však chová stejně jako sekvenční kód: pro každého pacienta dynamicky alokuje vektor, naplní jej daty a zavolá obecnou funkci pro medián.

5.3 Varianta 2: Paralelizace se SIMD

Tato varianta (funkce `processParallelCPU`) se snaží o ruční optimalizaci pomocí vektorových instrukcí SIMD. Využívá faktu, že pro daný dataset máme maximálně 8 dní měření. To umožňuje řazení pomocí Sorting Network nad vectory o 4 floatech - `float32x4_t`. Pro každý den

se vytvoří jeden vektor a do jednoho vektoru se načtou hodnoty pro 4 pacienty. Jedna operace řazení tak zpracuje 4 pacienty najednou.

```
// Funkce pro transpozici 4x4 matice uložené ve 4 vektorech
inline void transpose_matrix(float32x4_t& r0, float32x4_t& r1, float32x4_t& r2, float32x4_t& r3) {
    float32x4_t t0 = vtrn1q_f32(r0, r1);    // bere všechny sudé indexy
    float32x4_t t1 = vtrn2q_f32(r0, r1);    // bere všechny liché indexy
    float32x4_t t2 = vtrn1q_f32(r2, r3);
    float32x4_t t3 = vtrn2q_f32(r2, r3);
    r0 = vcombine_f32(vget_low_f32(t0), vget_low_f32(t2));
    r1 = vcombine_f32(vget_low_f32(t1), vget_low_f32(t3));
    r2 = vcombine_f32(vget_high_f32(t0), vget_high_f32(t2));
    r3 = vcombine_f32(vget_high_f32(t1), vget_high_f32(t3));
}

void DexcomData::processParallelCPU(int32_t num_wanted_time_slots) {
    // ... inicializace promenych a vektoru
    const uint32_t aligned_patients = (num_patients / 4) * 4;
    const uint32_t remainder = num_patients % 4;
    const uint32_t padding_patients = (remainder == 0) ? 0 : (4 - remainder);

    for (auto i = 0; i < padding_patients; i++) {
        for (auto j = 0; j < num_time_slots; j++) {
            flat_data.push_back(-1);
        }
    }
    const uint32_t patients_with_padding = aligned_patients + padding_patients;
    #pragma omp parallel for
    {
        for (uint32_t ts = 0; ts < num_time_slots; ++ts) {
            for (uint32_t p = 0; p < patients_with_padding; p += 4) {
                float* ptr = getPatientDataPtr(ts, p);

                // Načtení dat pro 4 pacienty a 4 dny měření najednou do vektorů
                // d0 obsahuje první 4 dny pro pacienta p
                float32x4_t d0 = vld1q_f32(ptr);
                // d1 obsahuje první 4 dny pro pacienta p+1
                float32x4_t d1 = vld1q_f32(ptr + 1*8);
                float32x4_t d2 = vld1q_f32(ptr + 2*8);
                float32x4_t d3 = vld1q_f32(ptr + 3*8);
                // Transpozice matic pro správné zarovnání
                // d0 obsahuje den 1 pro pacienty p, p+1, p+2, p+3
                transpose_matrix(d0, d1, d2, d3);

                // Načtení dat pro dny 5-8
                float32x4_t d4 = vld1q_f32(ptr + 4);
                float32x4_t d5 = vld1q_f32(ptr + 1*8 + 4);
                float32x4_t d6 = vld1q_f32(ptr + 2*8 + 4);
                float32x4_t d7 = vld1q_f32(ptr + 3*8 + 4);
                // Transpozice matic pro správné zarovnání
                transpose_matrix(d4, d5, d6, d7);

                // Výpočet mediánu pro 4 pacienty najednou ...
            }
        }
    }
```

```
}
```

Listing 5.1: Inicializace SIMD vektorů pro paralelní výpočet mediánu (CPUParallel.cpp)

5.3.1 Princip SIMD řazení - Sorting Network

Namísto obecného řazení používáme tzv. **Sorting Network** (řadicí síť). Jde o algoritmus s pevně daným počtem porovnání a prohození, který nezávisí na hodnotách dat (data-independent). Hledání využívá paralelní porovnání a prohození hodnot v registrech procesoru **"Compare and Swap"** pomocí instrukcí `vminq_f32` a `vmaxq_f32`.

```
#define SORT_VEC(A, B) { \
    float32x4_t min_v = vminq_f32(A, B); \
    float32x4_t max_v = vmaxq_f32(A, B); \
    A = min_v; \
    B = max_v; \
}
```

Listing 5.2: Makro pro porovnání a prohození dvou SIMD vektorů (CPUParallel.cpp)

Seřazení 8 hodnot pro 4 pacienty probíhá ve 21 krocích (Compare and Swap).

```
void DexcomData::processParallelCPU(int32_t num_wanted_time_slots) {
    for (uint32_t ts = 0; ts < num_time_slots; ++ts) {
        for (uint32_t p = 0; p < patients_with_padding; p += 4) {
            // Načtení a transpozice dat (viz předchozí kód) ...

            // Ruční řazení 8 hodnot pomocí Sorting Network
            SORT_VEC(d0, d1); SORT_VEC(d2, d3); SORT_VEC(d4, d5); SORT_VEC(
    d6, d7);
            SORT_VEC(d0, d2); SORT_VEC(d1, d3); SORT_VEC(d4, d6); SORT_VEC(
    d5, d7);
            SORT_VEC(d1, d2); SORT_VEC(d5, d6); SORT_VEC(d0, d4); SORT_VEC(
    d3, d7);
            SORT_VEC(d1, d5); SORT_VEC(d2, d6);
            SORT_VEC(d1, d4); SORT_VEC(d3, d6);
            SORT_VEC(d2, d4); SORT_VEC(d3, d5);
            SORT_VEC(d3, d4); SORT_VEC(d1, d2); SORT_VEC(d5, d6);

            // Filtrace paddingu a extrakce mediánu z registrových vektorů
    ...
        }
    }
}
```

Listing 5.3: Makro pro porovnání a prohození dvou SIMD vektorů (CPUParallel.cpp)

5.3.2 Extrakce mediánu ze seřazených registrů

Po seřazení pomocí Sorting Network jsou hodnoty v registrech uspořádány od nejmenšího k největšímu. Následuje filtrace paddingu (`EMPTY_VALUE`) a výpočet mediánu. Počet platných hodnot se spočítá pomocí porovnání s `EMPTY_VALUE + EPSILON` a na základě toho se určí pozice mediánu.

```
    // Filtrace paddingu a počítání platných hodnot po Sorting Network - viz
    . výše
    float32x4_t limit = vdupq_n_f32(EMPTY_VALUE + EPSILON);
    uint32x4_t counts = vdupq_n_u32(0);

    // Sčítáme masky pro nalezení neplatných hodnot
```

```
    counts = vsubq_u32(counts, vcgtq_f32(d0, limit));
    counts = vsubq_u32(counts, vcgtq_f32(d1, limit));
    counts = vsubq_u32(counts, vcgtq_f32(d2, limit));
    counts = vsubq_u32(counts, vcgtq_f32(d3, limit));
    counts = vsubq_u32(counts, vcgtq_f32(d4, limit));
    counts = vsubq_u32(counts, vcgtq_f32(d5, limit));
    counts = vsubq_u32(counts, vcgtq_f32(d6, limit));
    counts = vsubq_u32(counts, vcgtq_f32(d7, limit));

    uint32_t cnt[4];
    vst1q_u32(cnt, counts);

    // extrakce seřazených hodnot do pole
    float sorted_cols[32];
    vst1q_f32(sorted_cols + 0, d0);
    vst1q_f32(sorted_cols + 4, d1);
    vst1q_f32(sorted_cols + 8, d2);
    vst1q_f32(sorted_cols + 12, d3);
    vst1q_f32(sorted_cols + 16, d4);
    vst1q_f32(sorted_cols + 20, d5);
    vst1q_f32(sorted_cols + 24, d6);
    vst1q_f32(sorted_cols + 28, d7);

    // Výpočet mediánu pro každého z 4 pacientů
    for(int k=0; k<4; ++k) {
        int n = cnt[k];
        float median = EMPTY_VALUE;
        if(n > 0) {
            int start = 8 - n;
            int mid = n / 2;

            int idx1 = (start + mid) * 4 + k;
            median = sorted_cols[idx1];

            // Sudý počet prvků - průměr dvou prostředních hodnot
            if(n % 2 == 0) {
                int idx2 = (start + mid - 1) * 4 + k;
                median = (sorted_cols[idx2] + median) / 2.0f;
            }
            grand_median_buffer.push_back(median);
        }
        result_medians_par_per_pat[(ts * num_patients) + p + k] = median;
    }
```

Listing 5.4: Extrakce mediánu ze seřazených SIMD registrů (CPUParallel.cpp)

5.4 Medián napříč pacienty

Optímální algoritmus pro vektorové řazení se mi nepodařilo implementovat. Níže je ukázka implementace bitonic sort pro 16 prvků, ale to neseřadilo hodnoty napříč všemi vektory - jen v rámci jednoho vektoru o 4 prvcích. Z maticového pohledu to seřadilo sloupce i řádky.

```
inline void bitonic_sort_16(float32x4_t& q0, float32x4_t& q1, float32x4_t&
    q2, float32x4_t& q3) {

    SORT_VEC(q0, q1); SORT_VEC(q2, q3); SORT_VEC(q0, q2); SORT_VEC(q1, q3);
    SORT_VEC(q0, q3); SORT_VEC(q1, q2);

    transpose_matrix(q0, q1, q2, q3);
```

```
    SORT_VEC(q0, q1); SORT_VEC(q2, q3); SORT_VEC(q0, q2); SORT_VEC(q1, q3);
    SORT_VEC(q0, q3); SORT_VEC(q1, q2);

    transpose_matrix(q0, q1, q2, q3);
}
```

Listing 5.5: Bitonic Sort pro 16 prvků ve 4 vektorech (CPUParallel.cpp)

Proto je v této fázi využita standardní funkce `std::nth_element`, která sice není vektorizovaná, ale celá smyčka přes časové sloty je stále paralelizována pomocí OpenMP.

```
    // Výpočet Grand Median pro každý časový slot
    float gm = EMPTY_VALUE;
    if (!grand_median_buffer.empty()) {
        size_t n = grand_median_buffer.size() / 2;
        std::nth_element(grand_median_buffer.begin(), grand_median_buffer.
    begin() + n, grand_median_buffer.end());
        gm = grand_median_buffer[n];

        if (grand_median_buffer.size() % 2 == 0) {
            auto max_it = std::max_element(grand_median_buffer.begin(),
    grand_median_buffer.begin() + n);
            gm = (*max_it + gm) / 2.0f;
        }
    }
```

Listing 5.6: Bitonic Sort pro 16 prvků ve 4 vektorech (CPUParallel.cpp)

5.5 Downsampling časových slotů

Downsampling časových slotů je implementován paralelně pomocí OpenMP. Každé vlákno zpracovává jeden cílový (redukovaný) časový slot. Pro každý cílový slot se vypočítá rozsah původních slotů, které do něj spadají, a z těchto hodnot se spočítá medián pomocí `std::nth_element`. Implementace je bez manuální SIMD optimalizace.

```
    #pragma omp for schedule(static)
    for (int i = 0; i < num_wanted_time_slots; ++i) {
        size_t start_idx = (static_cast<size_t>(i) * num_time_slots) /
    num_wanted_time_slots;
        size_t end_idx = (static_cast<size_t>(i + 1) * num_time_slots) /
    num_wanted_time_slots;

        if (end_idx > num_time_slots) end_idx = num_time_slots;
        size_t count = end_idx - start_idx;
        if (count <= 0) {
            updated_timeslots_par[i] = (start_idx < num_time_slots) ?
    result_medians_par[start_idx] : EMPTY_VALUE;
            continue;
        }

        // Lokální buffer pro hodnoty v aktuálním rozsahu
        std::vector<float> local_buffer;
        // ... naplnění local_buffer hodnotami z result_medians_par ...

        float median = EMPTY_VALUE;
        if (!local_buffer.empty()) {
            auto n = local_buffer.size() / 2;
            std::nth_element(local_buffer.begin(), local_buffer.begin() + n,
     local_buffer.end());
```

```
            median = local_buffer[n];

            if (n % 2 == 0) {
                auto max_it = std::max_element(local_buffer.begin(),
    local_buffer.begin() + n);
                median = (*max_it + median) / 2.0f;
            }
        }
        updated_timeslots_par[i] = median;
    }
```

Listing 5.7: Paralelní downsampling časových slotů (CPUParallel.cpp)

Kapitola 6

GPU (OpenCL)

Pro zpracování větších dat (zejména BVP s **740 milionů záznamů** (5.7 milionu slotů × 16 pacientů × 8 dní)) byla implementována pipeline skládající se ze 3 kernelů. Rozdělení do tří kernelů bylo zvoleno z důvodu synchronizace paralelních operací. Implementace je realizována v OpenCL.

Celá pipeline je kontrolována z C++ hostitelského kódu, který připraví data, nastaví OpenCL prostředí, nahraje data na GPU, spustí jednotlivé kernely a stáhne výsledky zpět na CPU - `GPUParallel.cpp`. Mezi jednotlivými kernely zůstávají data uložena v globální paměti GPU a zpět do paměti programu jsou přenesena jen finální výsledky po skončení celého výpočtu.

```
void DexcomData::processGPU(int32_t num_wanted_time_slots, bool
    read_all_outputs, int8_t num_kernels_use, const std::string& kernel_file)
     {
    // nastavení OpenCL prostředí, načtení kernelů z kernel_file ...

    // alokace bufferů v globální paměti GPU
    // input data
    cl_mem d_data = clCreateBuffer(context, CL_MEM_READ_ONLY |
        CL_MEM_COPY_HOST_PTR, flat_data.size() * sizeof(float),
        flat_data.data(), &err);
    checkErr(err, "CreateBuffer Input Data");

    // medians per patients
    cl_mem d_medians_per_patients_results = clCreateBuffer(context,
        CL_MEM_WRITE_ONLY, total_items * sizeof(float), NULL, &err);
    checkErr(err, "CreateBuffer medians per patients");

    // medians per time_slots
    cl_mem d_time_slots_results = clCreateBuffer(context, CL_MEM_READ_WRITE,
        num_time_slots * sizeof(float), NULL, &err);
    checkErr(err, "CreateBuffer medians per time_slots");

    size_t total_items = num_time_slots * num_patients;

    // kernel 1: Výpočet mediánu napříč dny pro každého pacienta
    cl_kernel kernel = clCreateKernel(program, "median_kernel", &err);

    // nastavení argumentů kernelu 1 ...
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_data);
    clSetKernelArg(kernel, 1, sizeof(cl_mem),
        &d_medians_per_patients_results);
    clSetKernelArg(kernel, 2, sizeof(int), &n_days);
    clSetKernelArg(kernel, 3, sizeof(int), &n_total);
    size_t global_work_size = total_items;

```

```
    err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_work_size,
        NULL, 0, NULL, NULL);

    // kernel 2: Výpočet mediánu napříč pacienty pro každý časový slot
    if (num_kernels_use >= 2) {
        cl_kernel timeslots_kernel = clCreateKernel(program,
            "reduce_patients_kernel", &err);

        // nastavení argumentů kernelu 2 ...
        clSetKernelArg(timeslots_kernel, 0, sizeof(cl_mem),
            &d_medians_per_patients_results);
        clSetKernelArg(timeslots_kernel, 1, sizeof(cl_mem),
            &d_time_slots_results);
        clSetKernelArg(timeslots_kernel, 2, sizeof(int), &num_patients);

        size_t global_work_size2 = num_time_slots;

        err = clEnqueueNDRangeKernel(queue, timeslots_kernel,
            1, NULL, &global_work_size2, NULL, 0, NULL, NULL);

        // release kernel 2
    }
    // kernel 3: Downsampling časových slotů
    if (num_kernels_use >= 3 && num_wanted_time_slots > 0) {

        // buffer pro výsledky downsamplingu
        cl_mem d_wanted_time_slots_results = clCreateBuffer(context,
            CL_MEM_READ_WRITE, num_wanted_time_slots * sizeof(float),
            NULL, &err);
        checkErr(err, "CreateBuffer medians per wanted_time_slots");

        cl_kernel reduce_slots_kernel = clCreateKernel(program,
            "reduce_slots_kernel", &err);

        // nastavení argumentů kernelu 3 ...
        clSetKernelArg(reduce_slots_kernel, 0, sizeof(cl_mem),
            &d_time_slots_results);
        clSetKernelArg(reduce_slots_kernel, 1, sizeof(cl_mem),
            &d_wanted_time_slots_results);
        clSetKernelArg(reduce_slots_kernel, 2, sizeof(int),
            &num_time_slots);
        clSetKernelArg(reduce_slots_kernel, 3, sizeof(int),
            &num_wanted_time_slots);

        size_t global_work_size3 = num_wanted_time_slots;

        err = clEnqueueNDRangeKernel(queue, reduce_slots_kernel,
            1, NULL, &global_work_size3, NULL, 0, NULL, NULL);

        // stáhnutí výsledků z GPU
        rr = clEnqueueReadBuffer(queue, d_wanted_time_slots_results,
            CL_TRUE, 0, num_wanted_time_slots * sizeof(float),
            updated_timeslots_gpu.data(), 0, NULL, NULL);

        // release kernel 3
    }
    // release ostatních zdrojů GPU ...

}
```

6.1 Kernel 1: Median per Patient

Tento kernel počítá medián napříč 8 dny pro každého pacienta v každém časovém okně.

- **Vstup:** Raw data [TimeSlots × Patients × Days]
- **Implementace:** Každé vlákno (Work-item) zpracovává jednoho pacienta v jednom čase. Protože počet dnů je fixní (8), data se načtou do privátních registrů a seřadí pomocí **Sorting Network** (série instrukcí `CAS`).

```
__kernel void median_kernel(
    __global const float* data,
    __global float* result_medians,
    const int num_days,
    const int total_patients_slots
) {
    int gid = get_global_id(0);
    long offset = (long)gid * (long)num_days;

    // Načtení 8 dnů do registrů
    float v0 = data[offset + 0];
    float v1 = data[offset + 1];
    float v2 = data[offset + 2];
    float v3 = data[offset + 3];
    float v4 = data[offset + 4];
    float v5 = data[offset + 5];
    float v6 = data[offset + 6];
    float v7 = data[offset + 7];

    // Sorting Network (Compare-And-Swap)
    CAS(v0, v1); CAS(v2, v3); CAS(v4, v5); CAS(v6, v7);
    // ... další kroky sítě ...

    // Filtrace a výpočet mediánu
    int valid_count = 0;
    valid_count += (v0 > VALID_THRESHOLD);
    // ...

    if (valid_count > 0) {
        // ... výběr mediánu z pole 'sorted' a uložení do result_medians ...
    }
}
```

Listing 6.1: Median Kernel s CAS sítí (kernel.cl)

6.2 Kernel 2: Reduce Patients

Tento kernel počítá medián napříč všemi pacienty pro každý časový slot na základě výstupu z prvního kernelu.

- **Vstup:** Výstup z Kernelu 1 (mediány jednotlivých pacientů).
- **Úkol:** Pro každý časový slot spočítat medián ze všech pacientů.
- **Důvod oddělení:** Výpočet mediánu přes pacienty vyžaduje přístup k datům, která byla vyprodukována různými vlákny v Kernelu 1. Spojení do jednoho kernelu by vyžadovalo globální bariéry nebo atomické operace, což by degradovalo výkon. Spuštěním nového kernelu je synchronizace zaručena globálně.

- **Implementace:** Každé vlákno načte mediány všech pacientů pro daný čas do lokálního bufferu a provede řazení.

```
__kernel void reduce_patients_kernel(
    __global const float* input_matrix,
    __global float* output_medians,
    const int num_patients
) {
    int ts = get_global_id(0); // time_slot

    #define MAX_BUFFER_SIZE 16
    float buffer[MAX_BUFFER_SIZE];

    int valid_count = 0;
    long offset = (long)ts * (long)num_patients;

    for (int p = 0; p < num_patients; ++p) {
        float val = input_matrix[offset + p];
        if (val > VALID_THRESHOLD && valid_count < MAX_BUFFER_SIZE) {
            buffer[valid_count++] = val;
        }
    }

    float result = EMPTY_VALUE;
    if (valid_count > 0) {
        sort(buffer, valid_count);

        int mid = valid_count / 2;
        result = buffer[mid];

        // sudý počet prvků - průměr dvou prostředních hodnot
        if (valid_count % 2 == 0) {
            result = (buffer[mid - 1] + result) / 2.0f;
        }
    }

    output_medians[ts] = result;
}
```

Listing 6.2: reduce_patients_kernel pro medián napříč pacienty (kernel.cl)

Pro řazení jsou implementované 2 funkce - **Bubble Sort** a **Shell Sort**. Bubble Sort by měl být vhodnější pro malý počet pacientů (16), zatímco Shell Sort je efektivnější pro větší množiny dat. V další kapitole jsou změřeny výkony obou přístupů. Shell sort provádí řazení přímo v globální paměti bez alokace další paměti, zatímco Bubble Sort využívá lokální buffer.

```
void shell_sort_global(__global float* data, int start_idx, int count) {
    for (int gap = count / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < count; i += 1) {
            float temp = data[start_idx + i];
            int j;
            for (j = i; j >= gap && data[start_idx + j - gap] > temp; j -=
   gap) {
                data[start_idx + j] = data[start_idx + j - gap];
            }
            data[start_idx + j] = temp;
        }
    }
}

void sort(float* arr, int n) {
```

```
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                float temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Listing 6.3: Řazení: Bubble Sort a Shell Sort (kernel.cl)

6.3 Kernel 3: Downsampling (Shell Sort)

Tento kernel provádí downsampling časových slotů na základě výstupu z druhého kernelu. Jako parametr přijímá počet cílových časových slotů a pro každý cílový slot spočítá medián z příslušných původních slotů.

- **Vstup:** Výstup z Kernelu 2 (jeden medián pro každý časový slot).
- **Úkol:** Spočítat medián z K po sobě jdoucích časových slotů a tím provést downsampling.
- **Implementace:** Protože K může být velké (tisíce hodnot), nelze použít registry. Kernel využívá **In-Place Shell Sort** přímo v globální paměti grafické karty.

```
__kernel void reduce_slots_kernel(
    __global float* input_slots,
    __global float* output_final,
    const int num_total_slots,
    const int num_wanted_slots
) {
    // 1. Komprese dat (přesun validních hodnot na začátek úseku)
    int valid_count = 0;
    int write_ptr = start_idx;
    for (int read_ptr = start_idx; read_ptr < end_idx; ++read_ptr) {
        if (input_slots[read_ptr] > VALID_THRESHOLD) {
            if (read_ptr != write_ptr)
                input_slots[write_ptr] = input_slots[read_ptr];
            write_ptr++;
            valid_count++;
        }
    }

    // 2. Řazení přímo v globální paměti (bez alokace)
    if (valid_count > 0) {
        shell_sort_global(input_slots, start_idx, valid_count);
        // ... výběr mediánu ...
    }
}
```

Listing 6.4: Downsampling s Shell Sortem

6.4 Řešení pro BVP data (740 milionů záznamů)

Při zpracování BVP dat se objevily specifické problémy spojené s jejich objemem:

- **Přetečení indexů:** Počet prvků ($5.7 \times 10^6 \times 16 \times 8 \approx 740 \times 10^6$) se vejde do 32-bitového `int`, ale při výpočtu indexů (např. násobení) může dojít k přetečení mezivýsledku. V kernelech byla proto důsledně zavedena 64-bitová aritmetika (`long`) pro adresaci paměti.
- **Filtrace:** BVP data mohou být záporná. Byla zavedena konstanta `EMPTY_VALUE = -1e30` pro padding, aby se odlišila chybějící data od platných záporných hodnot.

Kapitola 7

Naměřené výsledky a Vyhodnocení

V této kapitole jsou prezentovány výsledky benchmarků pro různé sady dat. Měření byla prováděna opakovaně a výsledky byly validovány křížovou kontrolou (Sequential vs. Parallel vs. GPU), aby bylo zaručena, že jsou výsledky shodné (s definovanou tolerancí).

7.1 Hardwarová konfigurace

Měření probíhalo na následující konfiguraci:

- **CPU:** M1 Pro (8 jader)
- **GPU:** M1 Pro (14 jader)
- **RAM:** 16GB unifikované paměti
- **OS:** macOS Tahoe 26.2

7.2 Časy načítání dat

Vstupem pro algoritmy jsou CSV soubory. Měření zahrnuje čas potřebný pro přečtení souborů z disku a jejich konverzi do `flat_data` formátu. Kategorie dat DEXCOM, HR a BVP mají různé velikosti souborů kvůli rozdílným vzorkovacím intervalům.

Tabulka 7.1: Časy načítání vstupních souborů (15 souborů pro typ dat DEXCOM, 16 souborů pro typy dat HR a BVP)

Typ Dat (vzorkovací interval)	**Průměrná velikost jednoho souboru**	**Čas načítání [ms]**
Dexcom (5 min)	139 KB	166 ms
HR (1 sec)	15.8 MB	18 435 ms (18s)
BVP (15 ms)	1.28 GB	1 568 830 ms (26m 9s)

7.3 Dexcom Data (Nízká zátěž)

- **Charakteristika:** Vzorkování po 5 minutách (288 slotů/den).
- **Velikost:** Malá data, minimální paměťová náročnost.

Tabulka 7.2: Výsledky pro Dexcom Data

Metoda	**Čas [ms]**
CPU Sequential	5.45483 ms
CPU Parallel + SIMD	0.8895 ms
CPU Parallel (No Vectorization)	4.79196 ms
GPU (Full Pipeline)	59.51 ms

Zhodnocení: U malých dat je režie spojená s přenosem dat na GPU (PCIe latence) a spouštěním kernelů větší než samotný výpočet. Zde dominuje CPU Parallel verze díky vektorovým instrukcím, efektivnímu využití cache a absence overheadu.

7.4 HR Data (Střední zátěž)

- **Charakteristika:** Vzorkování po 1 sekundě (86 400 slotů/den).
- **Velikost:** Středně velká data.

Tabulka 7.3: Výsledky pro HR Data

Metoda	**Čas [ms]**
CPU Sequential	1544.08 ms
CPU Parallel + Vectorized	226.49 ms
CPU Parallel (No Vectorization)	1375.42 ms
GPU (Full Pipeline)	19.66 ms

Zhodnocení: S rostoucím objemem dat je rychlost GPU znatelnější. Paralelní CPU se SIMD instrukcemi stále nabízí dobrý výkon, ale pravděpodobně je limitována implementací, která vektorizuje jen první část výpočtu (medián napříč dny). GPU verze těží z paralelizace napříč všemi částmi výpočtu.

Zajímavé je, že GPU zde běží kratší dobu než GPU u Dexcom dat. Důvodem je, že GPU běží první nad Dexcom daty a následně na HR daty. Režie inicializace OpenCL a JIT kompilace kernelu se započítá do prvního běhu a při druhém běhu již není potřeba.

7.5 BVP Data (Vysoká zátěž)

- **Charakteristika:** Vzorkování po 15 ms (cca 5 760 000 slotů/den).
- **Velikost:** Velká data, miliony záznamů. Zde se projevují limity paměti a cache.
- **Specifika:** Data obsahují záporné hodnoty, což vyžadovalo robustní handling paddingu (`EMPTY_VALUE = -1e30`).

Tabulka 7.4: Výsledky pro BVP Data

Metoda	**Čas [ms]**
CPU Sequential	80440.40 ms
CPU Parallel + Vectorized	16506.80 ms
CPU Parallel (No Vectorization)	68084.00 ms
GPU (Full Pipeline)	1583.48 ms

Zhodnocení: Toto je scénář, pro který byla GPU optimalizace navržena. GPU je více než 36x rychlejší než sekvenční CPU a více než 7.5x rychlejší než paralelní CPU s vektorizací. Velká data plně využívají paralelní architekturu GPU a minimalizují režii spojenou s přenosem dat díky pipeline.

7.6 Analýza fází GPU výpočtu

Pro detailní pochopení výkonu GPU bylo provedeno měření po jednotlivých krocích (kernelech). To umožňuje identifikovat úzká hrdla pipeline.

- **1 Kernel:** Spuštění pouze `median_kernel` (výpočet mediánů pro každého pacienta a časový slot). Zpět do RAM se přenáší kompletní matice výsledků (časové sloty × pacienti).
- **2 Kernely:** Navíc spuštění `reduce_patients_kernel`. Redukce pacientů na GPU, do RAM se přenáší jen jeden medián pro každý časový slot.
- **3 Kernely (Full):** Kompletní pipeline včetně `reduce_slots_kernel` (downsampling). Zpět do RAM se přenáší pouze finální downsampled výsledky.

Tabulka 7.5: Časy GPU podle počtu aktivních kernelů (BVP data)

Konfigurace GPU Pipeline	**Čas [ms]**
Pouze Kernel 1 (Median per Patient)	1094.36
Kernel 1 + Kernel 2 (Reduce Patients)	534.659
Kernel 1 + 2 + 3 (Full Pipeline)	1024.54

Analýza: Čas pro "1 Kernel"(1094 ms) je dvojnásobný oproti "2 Kernelům"(535 ms). To indikuje **Memory Bottleneck** v přenosu dat z GPU. Výstupem prvního kernelu je cca 368 MB dat (92 milionů hodnot), které se musí přenést přes sběrnici do RAM. Při použití dvou kernelů zůstávají tato data v paměti GPU a přenáší se pouze výsledek po 2. kernelu (cca 23 MB). Časová úspora na přenosu dat zde výrazně převyšuje výpočetní čas druhého kernelu.

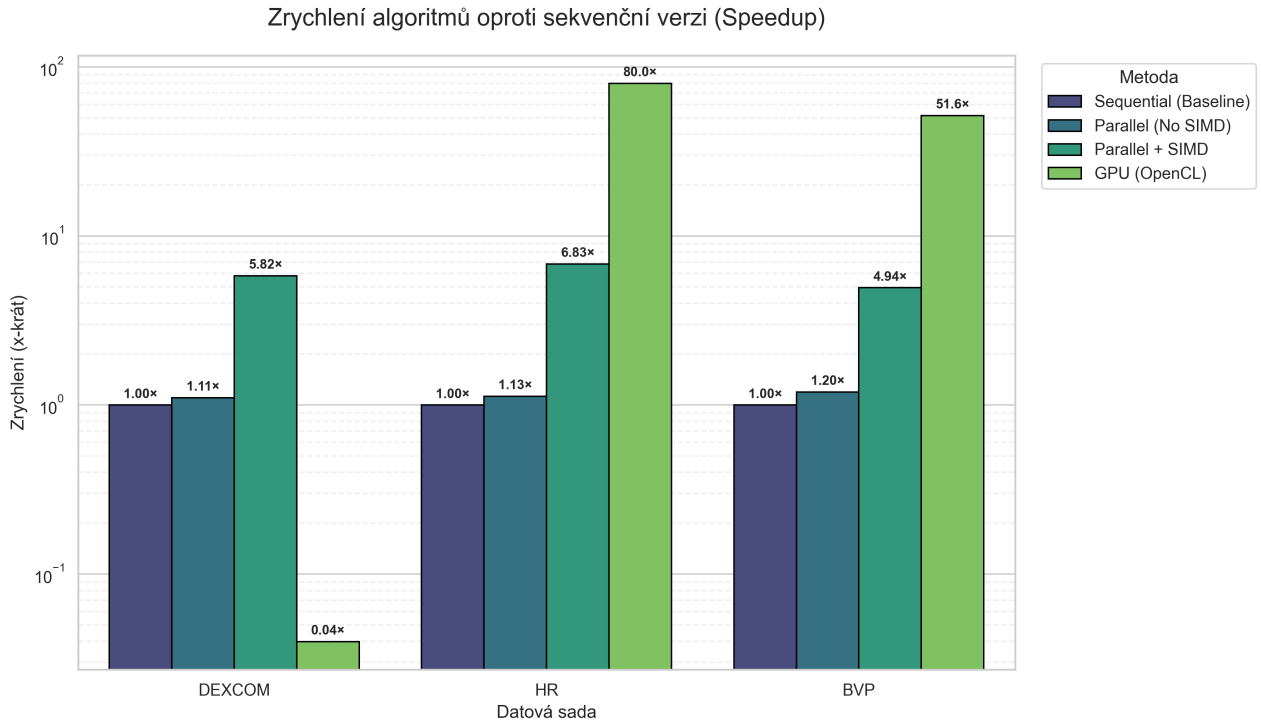
Kapitola 8

Analýza a Diskuze výsledků

Cílem této kapitoly je vyhodnotit efektivitu implementovaných algoritmů na základě naměřených časů. Zaměřujeme se na porovnání absolutního výkonu (škálovatelnost) a efektivity paralelizace (Karp-Flattova metrika).

8.1 Zrychlení a efektivita (Speedup)

Zrychlení (S) je definováno jako poměr času sekvenčního algoritmu k času paralelního algoritmu: $S = T_{seq}/T_{par}$. Z naměřených dat je patrné, že charakteristika zrychlení se dramaticky mění s velikostí dat.



Obrázek 8.1: Graf zrychlení (Speedup) jednotlivých metod oproti sekvenční verzi (osa Y je logaritmická). U malých dat (Dexcom) je GPU neefektivní, zatímco u velkých (BVP) dominuje se zrychlením přes 50×.

Z grafu 8.1 vyplývají tři klíčové poznatky:

1. **Režie GPU u malých dat:** U sady Dexcom dosahuje GPU zrychlení $S < 1$ (zpomalení). Režie spojená s přenosem dat a spouštěním kernelu zde převyšuje samotný výpočet.

2. **Dominance GPU u větších dat:** U sady HR a BVP je GPU výrazně efektivnější. Pro BVP dosahuje GPU zrychlení $51,6\times$.

3. **Role SIMD instrukcí:** U CPU implementace je viditelný rozdíl mezi čistou paralelizací (NoVect) a vektorizací. Zatímco 'Parallel (No SIMD)' dosahuje u BVP zrychlení pouze $1,2\times$, optimalizovaná verze 'Parallel + SIMD' dosahuje zrychlení $4,9\times$.

8.2 Karp-Flattova metrika

Pro hlubší analýzu efektivity paralelizace na CPU byla použita Karp-Flattova metrika (e), která odhaduje sériový podíl kódu včetně režie paralelizace. Výpočet byl proveden pro procesor Apple M1 Pro s počtem jader $p = 8$.

$$e = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Nízká hodnota e značí efektivní paralelizaci. Hodnoty v Tabulce 8.1 ukazují, jak efektivně dokáže algoritmus využít dostupná jádra.

Tabulka 8.1: Karp-Flattova metrika pro CPU implementace ($p = 8$)

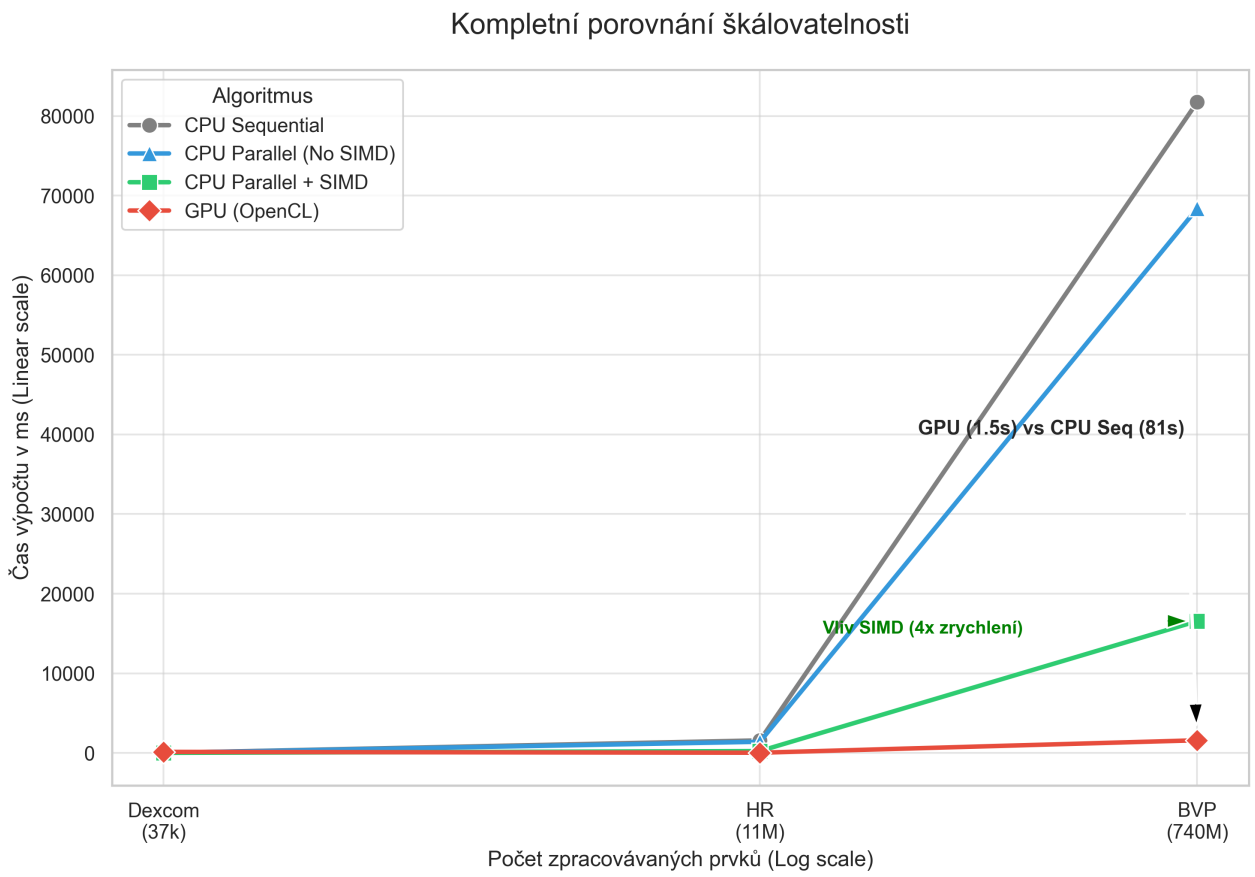
Dataset	**Metoda**	**Speedup (S)**	**Karp-Flatt (e)**	**Efektivita**
Dexcom	CPU Par + SIMD	5,82	0,053	Vysoká
HR	CPU Par + SIMD	6,83	0,024	Velmi vysoká
BVP	CPU Par + SIMD	4,94	0,089	Střední
BVP	**CPU Par (No SIMD)**	**1,19**	**0,817**	**Kritická**

Interpretace:

- U sady **HR** dosahuje metrika hodnoty $0,024$, což znamená, že pouze 2,4 % času tvoří sériová část nebo režie. Paralelizace je zde téměř ideální.
- U sady **BVP** (s vektorizací) hodnota stoupá na $0,089$ (8,9 % režie). To je pravděpodobně způsobeno saturací paměťové sběrnice (Memory Bound), kdy 8 jader soutěží o přístup k 1,2 GB dat.
- Metoda **BVP bez SIMD** vykazuje $e = 0,817$. To znamená, že 81,7 % výpočtu se chová sériově. Bez SIMD instrukcí, které zpracují 4 hodnoty najednou a efektivně využívají registry, je CPU zahlceno režií správy vláken a čekáním na data, což degraduje výkon téměř na úroveň sekvenčního kódu.

8.3 Škálovatelnost v čase

Poslední graf zobrazuje absolutní čas běhu v závislosti na velikosti vstupu.



Obrázek 8.2: Závislost doby výpočtu na velikosti vstupních dat (log-linear měřítko).

Graf 8.2 ilustruje, že zatímco sekvenční a čistě paralelní řešení rostou lineárně s velikostí dat (strmý nárůst u BVP), GPU řešení (červená křivka) si zachovává nízký čas zpracování i pro 740 milionů záznamů. Zelená křivka (SIMD) představuje ideální volbu pro středně velká data, kde je rychlejší než GPU (díky absenci přenosu dat), ale u větších dat již na GPU ztrácí.

Kapitola 9

Diskuze nad implementací

Projekt demonstruje efektivní zpracování velkých datových sad pomocí různých přístupů - sekvenčního CPU, paralelního CPU s vektorizací a GPU akcelerace. Každý přístup má své výhody a nevýhody v závislosti na velikosti dat a dostupném hardwaru. Při větších datových sadách (BVP) je GPU akcelerace jasně nejefektivnější, zatímco pro menší datové sady (Dexcom) může být optimalizovaná paralelní verze na CPU výhodnější díky nižší režii.

Práce úspěšně řeší několik technických výzev, včetně efektivního využití SIMD instrukcí na CPU a implementace robustní pipeline na GPU. Nicméně, existují oblasti pro další zlepšení:

- **Vektorizace redukční fáze na CPU:** Současná implementace využívá SIMD pouze pro výpočet mediánu napříč dny. Vektorizace dalších částí výpočtu (medián napříč pacienty, medián při downsamplingu) by mohla dále zlepšit výkon paralelní CPU verze. Efektivní implementace řazení dynamicky dlouhého pole pomocí vektorových instrukcí je však náročná. Já jsem se o to pokoušel (viz. kapitola 5.4), ale bez úspěchu.
- **Dynamická volba strategie:** Vývoj adaptivního systému, který by na základě velikosti dat a dostupného hardwaru automaticky zvolil nejefektivnější zpracovatelský přístup (CPU vs. GPU).

Vzhledem k vellikosti některých datových sad (BVP) trvá načítání dat z disku značný čas (více než 26 minut). Pro reálné aplikace by bylo vhodné zvážit optimalizace v této oblasti, například paralelním načítáním dat.

Kapitola 10

Závěr

Projekt úspěšně implementuje zpracování velkých medicínských datových sad 3 způsoby - sekvenční CPU, paralelní CPU+SIMD a GPU. Během vývoje byly vyřešeny klíčové technické výzvy:

- **Filtrace dat:** Specifika BVP signálu (záporné hodnoty) si vynutila změnu logiky pro detekci prázdných míst. Zavedení konstanty `EMPTY_VALUE = -1e30` a sjednocení podmínek ve všech metodách zajistilo konzistenci výsledků.
- **Optimalizace:**
  - **GPU:** Využití Sorting Networks v prvním kernelu a in-place Shell Sortu ve třetím kernelu minimalizovalo nároky na paměť.

Výsledná aplikace prokazuje, že pro masivní data (BVP) je GPU akcelerace vysoce efektivní, zatímco pro menší datové sady (Dexcom) je vhodnější optimalizovaná paralelní verze na CPU.

Příloha A: Uživatelský manuál

Aplikace je navržena jako nástroj příkazové řádky (CLI), který umožňuje flexibilní konfiguraci benchmarků. Uživatel může volit, které datové sady se mají zpracovat, jaké výpočetní metody (sekvenční, paralelní, GPU) se mají použít a kam se mají uložit výsledky. **V aplikaci jsou použity vektorové instrukce NEON SIMD specifické pro ARM architekturu.**

Překlad a Sestavení

Projekt využívá sestavovací systém CMake. Pro sestavení aplikace je nutné mít nainstalovaný kompilátor podporující C++23 a knihovnu OpenCL.

```
mkdir build
cd build
cmake ..
make
```

Po úspěšném sestavení vznikne v adresáři `build` spustitelný soubor `ppr_semestralka`. Příkazem `./ppr_semestralka -h` lze zobrazit nápovědu k ovládání.

Přidání složky s daty

Vstupní data by měla být umístěna ve složce `data` ve kořenovém adresáři projektu (ne ve složce `build`). Struktura složky `data` by měla být následující:

```
data/
    001/
        DEXCOM_001.csv
        HR_001.csv
        BVP_001.csv
    002/
        DEXCOM_002.csv
        HR_002.csv
        BVP_002.csv
    003/
        DEXCOM_003.csv
        HR_003.csv
        BVP_003.csv
    ...
```

Parametry příkazové řádky

Program přijímá následující přepínače pro konfiguraci běhu:

`-d, -datasets <list>` Specifikuje čárkou oddělený seznam datových sad ke zpracování.

- Možnosti: `dexcom`, `hr`, `bvp`
- Výchozí: Všechny (`dexcom,hr,bvp`)

`-m, -mode <list>` Specifikuje čárkou oddělený seznam výpočetních metod.

- Možnosti: `seq` (Sekvenční), `par` (Paralelní CPU), `gpu` (OpenCL), `all` (Všechny)
- Výchozí: `all`

`-k, -kernels <num>` Nastavuje počet aktivních fází (kernelů) v GPU pipeline. Slouží primárně pro analýzu úzkých hrdel.

- `1`: Pouze výpočet mediánů pro pacienta (masivní přenos dat).
- `2`: Přidána redukce pacientů na GPU (minimalizace přenosu).
- `3`: Kompletní pipeline včetně downsamplingu (vysoká výpočetní zátěž).
- Výchozí: `3`

`-o, -output <file>` Cesta k souboru pro logování výsledků benchmarku.

- Výchozí: `benchmark_logs.txt`

`-h, -help` Vypíše nápovědu k ovládání.

Příklady použití

1. Základní spuštění (všechny testy): Spustí všechny algoritmy nad všemi datovými sadami s výchozím nastavením.

```
./ppr_semestralka
```

2. Benchmark pouze GPU na velkých datech: Spustí pouze GPU implementaci nad BVP daty a výsledky uloží do specifického souboru.

```
./ppr_semestralka -d bvp -m gpu -o gpu_results.txt
```

3. Analýza GPU pipeline (Memory Bound test): Spustí GPU výpočet nad BVP daty pouze s prvním kernelem pro otestování propustnosti paměti.

```
./ppr_semestralka -d bvp -m gpu -k 1
```

4. Porovnání CPU metod: Spustí sekvenční a paralelní verzi nad HR a Dexcom daty pro ověření zrychlení na procesoru.

```
./ppr_semestralka -d hr,dexcom -m seq,par
```