



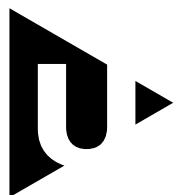
FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Semestrální práce

Filtrace EEG signálu lineární 1D konvolucí

Rastislav Lipták



FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Semestrální práce

Filtrace EEG signálu lineární 1D konvolucí

Bc. Rastislav Lipták

Obsah

1	Zadání	3
2	Užitý hardware a software	4
2.1	Hardware	4
2.2	Software	5
3	Popis implementace	6
3.1	Referenční implementace	8
3.2	Naivní implementace	8
3.2.1	Implementace na CPU (Sekvenční a Paralelní)	8
3.2.2	Implementace na GPU	10
3.3	Optimalizovaná CPU implementace	13
3.3.1	Skalární a autovektorizovaná implementace	13
3.3.2	Manuálně vektorizovaná implementace	16
3.4	Optimalizovaná GPU implementace	18
4	Uživatelské rozhraní a ovládání	21
4.1	Konfigurace a vstupy	21
4.2	Prezentace výstupů a metrik	22
5	Validace a správnost výsledků	24
5.1	Metodika porovnání	24
5.2	Výsledky validace	24
5.2.1	Analýza odchylek	27
6	Výkonnostní analýza	28
6.1	Metodika měření	28
6.1.1	Sběr dat	28
6.1.2	Statistické zpracování a metriky	29
6.2	Porovnání implementací	29
6.2.1	Souhrnné výsledky	29

6.2.2	Analýza CPU výkonu a konfrontace s teorií	30
6.2.3	Analýza GPU výkonu a hardwarové limity	32
6.2.4	Teoretická analýza paralelizace	33
6.3	Analýza škálování s velikostí dat	35
6.3.1	Linearita výpočtu	35
6.3.2	Vliv paměťové hierarchie a System Level Cache	37
6.4	Analýza škálování s velikostí poloměru jádra	38
6.4.1	Charakteristika GPU škálování	39
6.4.2	Limity naivních algoritmů	39
6.4.3	Stabilizace optimalizovaných implementací	40
6.4.4	Srovnání se systémovou knihovnou Apple vDSP	41
6.5	Výkon na odlišných architekturách Apple Silicon	41
6.5.1	Specifikace testovacích zařízení	42
6.5.2	Adaptace metodiky pro srovnávací měření	42
6.5.3	Výsledky srovnávacího měření	43
7	Závěr	45
Bibliografie		47
Seznam obrázků		50
Seznam tabulek		51
Seznam výpisů		52

Zadání

1

Cílem práce je realizace a optimalizace výpočtu lineární 1D konvoluce nad EEG daty.

Matematická definice: Algoritmus realizuje diskrétní konvoluci vstupního signálu x s konvolučním jádrem h o poloměru R . Výstupní hodnota $y[i]$ je definována jako vážený součet prvků v okolí bodu i :

$$y[i] = \sum_{j=-R}^R x[i+j] \cdot h[j] \quad (1.1)$$

Kde:

- x představuje vstupní diskrétní signál,
- h je symetrické konvoluční jádro o velikosti $2R + 1$,
- R značí poloměr konvoluce.

Požadavky na implementaci:

- **CPU varianta:** Vytvoření implementace, která efektivně kombinuje sekvenční zpracování, vícevláknové zpracování a SIMD vektorizaci.
- **GPU varianta:** Realizujte výpočet na grafickém akcelerátoru.
- **Referenční implementace:** Pro účely srovnání a ověření optimalizace bude zahrnuta i naivní (neoptimalizovaná) verze algoritmu.

Užitý hardware a software

— 2

Pro implementaci, experimentální část a měření výkonu byl využit počítač **Apple MacBook Pro 14 (2021)** osazený čipem (SoC) Apple M1 Pro. Tato kapitola shrnuje technické specifikace tohoto zařízení se zaměřením na parametry ovlivňující výpočetní výkon a popisuje softwarové prostředí, ve kterém probíhal překlad a běh aplikace.

2.1 Hardware

Čip Apple M1 Pro je založený na architektuře ARM (ISA ARMv8.5-A [Son]). Procesorová část (CPU) disponuje celkem 8 jádry, která jsou rozdělena do dvou clusterů dle architektury big.LITTLE:

- **Výkonná jádra:** 6 jader s kódovým označením **Firestorm**. Tato jádra pracují na frekvenci 600 až 3228 MHz. Disponují 192 KB instrukční cache, 128 KB datové cache a sdílenou 24 MB L2 cache [Notb]. Mikroarchitektura Firestorm se vyznačuje šírkou pipeline 8 instrukcí na cyklus [Joha].
- **Úsporná jádra:** 2 jádra s kódovým označením **Icestorm**, pracující na frekvenci 600 až 2064 MHz. Jsou vybavena menší 128 KB instrukční a 64 KB datovou cache a sdílenou 4 MB cache [Notb].

Oba typy jader podporují vektorové instrukce technologie NEON a disponují 32 vektorovými registry o šířce 128 bitů [Arm24]. Paměťová hierarchie je dále rozšířena o systémovou cache (System Level Cache) o velikosti 24 MB [Wika].

Jelikož výrobce oficiální hodnoty výkonu ve FLOPS pro procesor neuvádí, je nutné je odvodit z parametrů mikroarchitektury. Výkonné jádro **Firestorm** dokáže zpracovat 32 operací s plovoucí rádovou čárkou za takt (díky čtveřici 128bitových NEON jednotek) [Joha]. Při maximální frekvenci 3,22 GHz lze teoretický výkon jednoho jádra vypočítat následovně:

$$3,22 \text{ GHz} \times 32 \text{ FLOPs/cyklus} \approx 103 \text{ GFLOPS}$$

V případě zapojení všech jader (multicore) se celkový teoretický výkon pohybuje v rozmezí 660 až 700 GFLOPS (součet výkonu šesti jader **Firestorm** a dvou úsporných jader **Icestorm**).

Systém využívá jednotnou paměť (Unified Memory) typu LPDDR5-6400 o kapacitě 16 GB. Paměť je připojena přes 256bitovou sběrnici, což zajišťuje propustnost 200 GB/s [Notb].

Integrovaný grafický akcelerátor (GPU) obsahuje 14 aktivních jader, která celkově disponují 1792 aritmeticko-logickými jednotkami (ALU). GPU využívá SIMD architekturu se šírkou 32 vláken a má hardwarově omezenou sdílenou paměť (tzv. **threadgroup memory**) o velikosti 32 kB [Appd]. Na základě analýzy třetí strany GPU dosahuje teoretického výpočetního výkonu v plovoucí řádové čárce (FP32) přibližně 4,6 TFLOPS [Nota].

Zařízení využívá aktivní systém chlazení.

2.2 Software

Vývoj, testování a měření probíhalo v prostředí operačního systému **macOS** (verze 26.1, 25B78). Jako vývojové prostředí (IDE) byl použit **Xcode** verze 26.2 (17C52).

Projekt je implementován v jazyce **C++** (standard C++23) s využitím grafického API **Metal** (verze 4). Pro překlad kódu byly využity následující optimalizační parametry (flagy) kompilátoru, nastavené pro konfiguraci *Release*:

- **Obecné C++ flagy:**

Zdrojový kód 2.1: Kompilační flagy C++

```
1 -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -O3 -  
2 ffast-math
```

- **Metal compiler flagy:**

Zdrojový kód 2.2: Kompilační flagy Metal

```
1 -O3 -ffast-math  
2
```

Aplikace využívá následující systémové frameworky a knihovny:

- **Metal.framework**, **Accelerate.framework**,
- **Foundation.framework**, **CoreFoundation.framework**,
- **QuartzCore.framework**, **IOKit.framework**,
- **libcurl.tbd**.

Popis implementace

3

Struktura zdrojového kódu je navržena s ohledem na striktní oddělení vstupních a výstupních operací (modul `src/io`) a výpočetní logiky (`src/processors`). Architekturu doplňuje integrovaná knihovna `edflib` pro manipulaci se soubory EDF a pro komunikaci s grafickým akcelerátorem na platformě Apple Silicon je využitý C++ wrapper `metal-cpp`, umožňující přímé volání Metal API bez nutnosti Objective-C vrstvy.

Zpracování dat probíhá v sekvenční pipeline navržené pro minimalizaci paměťové náročnosti. Celý proces zpracování dat, vizualizovaný na diagramu 3.1, se skládá ze čtyř na sebe navazujících fází:

1. **Inicializace:** Na základě parametrů od uživatele se připraví konfigurační struktury. Následně se vygeneruje konvoluční jádro (Gaussova křivka), které je uloženo ve standardním `std::vector`.
2. **Načtení dat:** Funkce `load_edf_data` zajistí načtení signálů a jejich konverzi na typ `float`. Data všech kanálů jsou "zploštěna" do jediného spojitého 1D vektoru. Tato struktura dat byla zvolena proto, že umožňuje procesoru i grafické kartě efektivněji rozdělovat práci na menší bloky bez nutnosti složité adresace vícerozměrných polí.

Pro uložení dat je definován vlastní typ `NeonVector`, který využívá zarovnání `aligned_allocator` na 16384 bajtů.

- Toto zarovnání odpovídá velikosti paměťové stránky, což je nezbytná podmínka pro vytvoření tzv. *zero-copy* bufferů v rámci rozhraní Metal API [Appe].
- Zároveň toto zarovnání automaticky splňuje požadavky na zarovnání pro CPU cache line (128 B [Appb]) a SIMD registry (16 B), čímž je zajištěna minimalizace penalizací při přístupu do paměti.

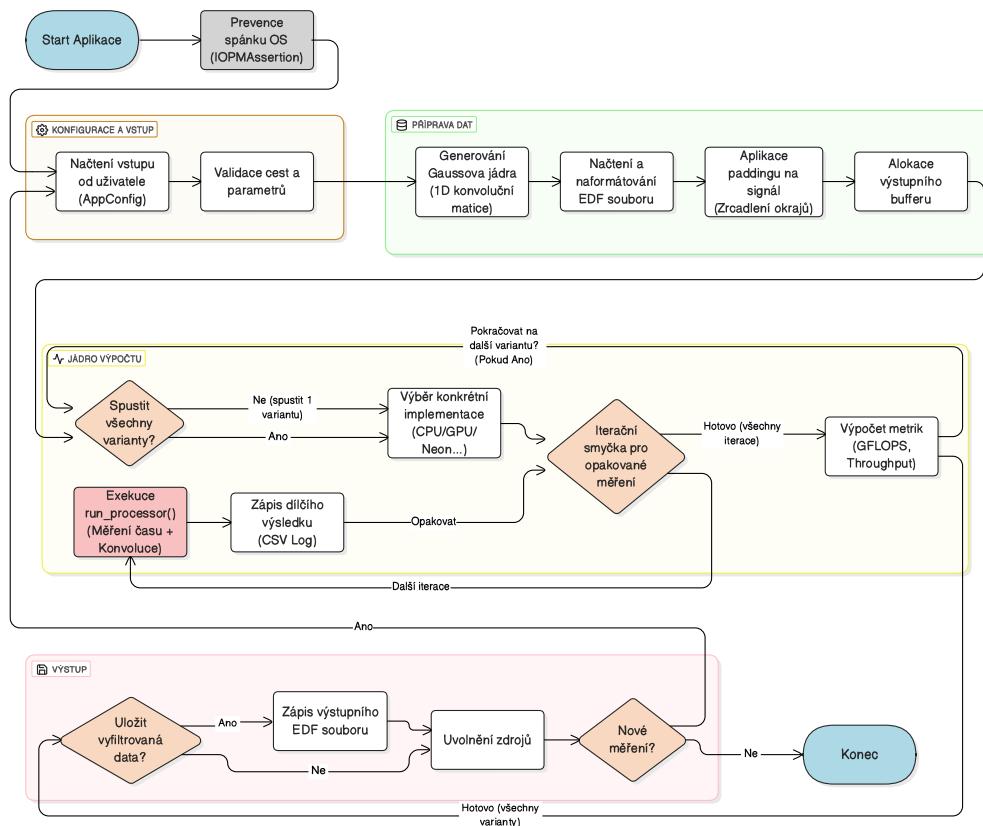
3. **Výpočetní jádro:** Funkce `run_processor` řídí spouštění konkrétních implementací algoritmu. Aby se předešlo degradaci výkonu na sběrnici, pracuje se

striktně se dvěma oddělenými buffery (vstupní `const` a výstupní). Při vývoji paralelních algoritmů se totiž ukázalo, že zápis a čtení do stejného paměťového bloku (*in-place* operace) vede k problému, kdy vlákna na různých jádrech si navzájem zneplatňovala cache lines, což zahltalo sběrnici synchronizační reží a výrazně zpomalilo výpočet.

Specifikem GPU implementace je práce s pamětí:

- **Data signálu:** Díky použití NeonVector a SoC architektuře je možné využít techniku *Zero-copy*.
- **Konvoluční jádro:** Jelikož je jádro uloženo v běžném `std::vector` (bez specifického zarovnání), musí být do paměti GPU explicitně zkopirováno. Vzhledem k malé velikosti jádra (řádově kB) je však režie tohoto kopírování zanedbatelná.

4. **Export:** Po dokončení výpočtu funkce `save_data` převede výsledky z linearizovaného vektoru zpět do struktury formátu EDF a uloží je na disk.



Obrázek 3.1: Control Flow Diagram aplikace.

3.1 Referenční implementace

Pro ověření správnosti výstupů a srovnání výkonu byla implementována referenční verze využívající systémovou knihovnu **Accelerate framework** (modul vDSP). Tato knihovna je na platformě Apple Silicon implementována přímo nad instrukční sadou NEON (SIMD) [Appa].

Pro výpočet byla zvolena funkce vDSP_conv, která realizuje klouzavý skalární součin [Appf]. Z hlediska architektury aplikace je zásadní, že je tato funkce synchronní a neimplementuje interní paralelizaci.

Matematicky funkce počítá korelací mezi jádrem A a signálem B :

$$C[n] = \sum_{k=0}^{N-1} A[k] \cdot B[n+k] \quad (3.1)$$

Díky symetrii použitého Gaussova jádra je tento výpočet ekvivalentní konvoluci bez nutnosti inverze vektoru filtru.

3.2 Naivní implementace

Tato část popisuje prvotní funkční verze algoritmů pro CPU i GPU. Tyto implementace, označované jako „naivní“, představují doslovny přepis matematické definice konvoluce (rovnice 1.1) do programového kódu.

Hlavním účelem těchto verzí není dosažení maximálního výkonu, ale stanovení základní úrovně výkonu, vůči které je měřeno zrychlení (speedup) optimalizovaných implementací.

V těchto implementacích není využita explicitní vektorizace či manuální rozbalování smyček. Optimalizace je ponechána čistě na schopnostech kompilátoru.

3.2.1 Implementace na CPU (Sekvenční a Paralelní)

Základem obou CPU variant je dvojitý vnořený cyklus. Vnější smyčka iteruje přes prvky signálu, zatímco vnitřní smyčka provádí skalární součin mezi částí vstupu a konvolučním jádrem.

Zdrojový kód 3.1: Jádro naivní konvoluce na CPU

```

1 for (size_t i = Radius; i < dataSize - Radius; ++i) {
2     float sum = 0.0f;
3     for (int j = -Radius; j <= Radius; ++j) {
4         sum += data[i + j] * convolutionKernel[j + Radius];
5     }
6     outputBuffer[i - Radius] = sum;
7 }
```

Rozdíl mezi sekvenční a paralelní verzí spočívá v řízení toku výpočtu. Zatímco sekvenční verze iteruje přes data pomocí standardního cyklu `for`, paralelní varianta deleguje tuto úlohu na systémové API ***Grand Central Dispatch*** (GCD). Konkrétně volání funkce `dispatch_apply` nahrazuje vnější smyčku a automaticky rozděluje výstupní pole na menší nezávislé bloky.

Klíčovým parametrem optimalizace je velikost tohoto bloku. Cílem bylo nalézt nejmenší možnou granularitu pro ideální rozložení zátěže, u které se ještě neprojevuje režie správy vláken. Při testování menších bloků (2048 prvků) bylo naměřeno konzistentní zpomalení výpočtu, způsobené režíí správy velkého množství úloh. Opačný extrém představovala hodnota 32768 prvků (typu `float`), jejíž paměťový nárok 128 kB přesně odpovídá celkové kapacitě L1 cache jader ***Firestorm***. Zde docházelo k poklesu výkonu vlivem saturace cache, kdy nezbýval prostor pro konvoluční jádro a překrývající se vstupní data. V pásmu mezi těmito limity (konkrétně pro hodnoty 4096, 8192 a 16384) však měření vykazovala statisticky totožné výsledky. Jako finální parametr tak byla zvolena hodnota `ChunkSize = 8192`, která leží uprostřed tohoto stabilního výkonnostního intervalu.

Dalším parametrem ovlivňujícím výkon je priorita fronty GCD, která je nastavena na `QOS_CLASS_USER_INTERACTIVE`. Tento příznak informuje plánovač operačního systému, že se jedná o úlohu vyžadující okamžitou odezvu, což na hybridní architektuře Apple Silicon vede k prioritnímu přidělování práce na výkonné jádra a zvyšování jejich taktu na maximální úroveň [Appc; Oak].

Zdrojový kód 3.2: Ukázka paralelizace pomocí GCD

```

1 const size_t numChunks = (totalWork + ChunkSize - 1) /
    ChunkSize;
2 dispatch_apply(numChunks, dispatch_get_global_queue(
    QOS_CLASS_USER_INTERACTIVE, 0), ^(size_t chunkIdx) {
3     size_t chunkStart = start + chunkIdx * ChunkSize;
4     size_t chunkEnd = std::min(chunkStart + ChunkSize, end);
5
6     for (size_t i = chunkStart; i < chunkEnd; ++i) {
7         // ... výpočet konvoluce (stejný jako u sekvenční
8         // verze) ...
9     }
9 });

```

Výkonnostní limity

Ačkoliv je tento kód funkční a snadno čitelný, trpí několika zásadními nedostatkami, které brání plnému využití hardwaru Apple Silicon:

1. **Silná datová závislost:** Výpočet využívá jediný akumulátor (`sum`), do kterého se v každé iteraci přičítá výsledek. To vytváří sériovou závislost, kdy proce-

sor nemůže zahájit další krok výpočtu, dokud není dokončen ten předchozí. Vzhledem k tomu, že instrukce FMA (Fused Multiply-Add) má na architektuře **Firestorm** latenci 4 cykly, vznikají v pipeline procesoru prázdná místa a výpočetní jednotky nejsou plně vytíženy [Johb].

2. **Neefektivní práce s registry:** Při posunu konvolučního okna dochází k zahazování dat z registrů. Protože se okno posouvá vždy jen o jeden prvek, je drtivá většina vstupních hodnot z předchozího kroku potřebná i v tom aktuálním (dochází k výraznému překryvu dat). Jelikož však naivní implementace tato data v registrech neudrží, musí být v každé iteraci znova načtena z L1 cache. I přes rychlosť cache tento proces zbytečně brzdí výpočet.

3.2.2 Implementace na GPU

Přenos dat mezi CPU a GPU je optimalizován pro architekturu sdílené paměti. Díky tomu, že jsou vstupní data zarovnaná na potřebnou velikost paměťové stránky (využitím **NeonVector**), není nutné je fyzicky překopírovávat. Při vytváření bufferu na GPU tak nedochází k alokaci další paměti, ale API pouze převeze od procesoru ukazatel na paměťový blok vstupních dat a výstupního bufferu. Tento přístup (**Zero-Copy**) ovšem není použitý pro konvoluční jádro, které je paměťově zanedbatelné a jehož hodnoty se fyzicky překopírovávají [Appe].

Zdrojový kód 3.3: Vytváření bufferů: Zero-Copy vs. standardní alokace

```

1 // Zero-Copy přenos vstupních dat
2 MTL::Buffer* dataBuffer = ctx.device->newBuffer(
3     (void*)data.data(),
4     data.size() * sizeof(float),
5     MTL::ResourceStorageModeShared,
6     nullptr
7 );
8
9 // Standardní alokace s kopírováním dat pro konvoluční jádro
10 MTL::Buffer* kernelBuffer = ctx.device->newBuffer(
11     convolutionKernel.data(),
12     convolutionKernel.size() * sizeof(float),
13     MTL::ResourceStorageModeShared
14 );

```

Konfigurace dlaždic a vláken. Při samotném zpracování signálu pak dochází k rozdelení vstupních dat na menší nezávislé bloky zvané dlaždice (**Tiles**). Každá dlaždice představuje úsek dat, který je zpracován jednou skupinou vláken (**Thread-group**). Velikost této dlaždice není náhodná, ale je určena součinem parametrů

THREADS_PER_GROUP (počet výpočetních vláken ve skupině) a ITEMS_PER_THREAD (počtem vzorků signálu, které zpracuje jedno vlákno).

$$\text{TILE_SIZE} = \text{THREADS_PER_GROUP} \times \text{ITEMS_PER_THREAD} \quad (3.2)$$

Pro implementaci byla zvolena konfigurace THREADS_PER_GROUP = 256. Tato hodnota je násobkem hardwarové šířky SIMD jednotky (32 vláken), což zajišťuje plné vytížení výpočetních cyklů. Současně objem dat, který musí takto velká skupina načíst, bezpečně nepřekračuje limit 32 kB vyhrazený pro sdílenou paměť. Ačkoliv experimentální měření s hodnotami 128, 512 a 1024 ukázala, že výkon této verze je limitován spíše paměťovou propustností než nastavením paralelizace, byla hodnota 256 zachována pro konzistenci s navazující optimalizovanou implementací.

Druhým parametrem je konfigurace ITEMS_PER_THREAD = 4, která byla zvolena s ohledem na využití datového typu float4. Ten umožňuje efektivně zpracovávat čtyři vzorky signálu v jediném kroku, což se promítá do celé struktury výpočetního kernelu.

Výpočet v rámci kernelu. Výpočet v rámci jednoho kernelu začíná fází kooperativního načítání, kdy skupina vláken společně přenese přidělenou dlaždici z globální paměti do rychlé lokální paměti (**threadgroup memory**). Vlákna načítají data po vektorech o 4 vzorcích a naplní cache nejen daty pro výpočet, ale i nezbytnými okraji.

Zdrojový kód 3.4: Kooperativní načtení dat do sdílené paměti

```

1 int groupStartPixel = group_id * (THREADS_PER_GROUP * 4);
2 int startVectorIdx = groupStartPixel / 4;
3 int totalPixels = (THREADS_PER_GROUP * 4) + kSize - 1;
4 int vectorsNeeded = (totalPixels + 3) / 4;
5 threadgroup float4* cacheVec = (threadgroup float4*)cache;
6
7 for (int i = tid; i < vectorsNeeded; i += THREADS_PER_GROUP)
8 {
9     int globalVecIdx = startVectorIdx + i;
10    cacheVec[i] = data[globalVecIdx];
}
```

Po synchronizaci bariérou následuje výpočetní smyčka, kde již každé vlákno pracuje samostatně: čte připravená data z lokální paměti, provádí aritmetické operace nad čtvericemi hodnot a výsledný vektor zapisuje zpět.

Zdrojový kód 3.5: Naivní vektorizovaný výpočet na GPU

```

1 threadgroup_barrier(mem_flags::mem_threadgroup);
2
```

```

3 int localPixelIndex = tid * 4;
4 float4 sum = float4(0.0f);
5
6 for (int k = 0; k < kSize; ++k) {
7     float d0 = cache[localPixelIndex + k + 0];
8     float d1 = cache[localPixelIndex + k + 1];
9     float d2 = cache[localPixelIndex + k + 2];
10    float d3 = cache[localPixelIndex + k + 3];
11
12    sum = fma(float4(d0, d1, d2, d3), float4(convKernel[k]), sum);
13 }
14
15 int globalOutputVecIndex = (groupStartPixel + localPixelIndex
16 / 4;
16 output[globalOutputVecIndex] = sum;

```

Výkonnostní limity

V rámci této implementace byly nasazeny některé optimalizační techniky, jako je Zero-Copy přenos dat a práce s vektorovými typy. Přesto naráží na výkonnostní limity způsobené samotnou architekturou kódu:

- Silná datová závislost:** Závislost na jediném akumulátoru `sum` vynucuje striktně sériové zpracování v rámci jednoho výpočetního kernelu. Další instrukce FMA nemůže začít před dokončením předchozí, což vede k nevyužitým cyklům (pipeline stalls). Jelikož vlákno nemá žádnou jinou nezávislou prací, nelze tyto prostoje skrýt na úrovni instrukčního paralelismu.
- Redundantní čtení a nízká aritmetická intenzita:** Limitujícím faktorem je nutnost opakovaného načítání čtyř skalárních hodnot (vstupního vektoru) uložených ve sdílené paměti. Protože implementace neudrží data v registrech mezi iteracemi, musí být při posunu konvolučního okna načítána znova, ačkoliv se většina hodnot překrývá. To vede k extrémnímu nepoměru: na jednu výpočetní instrukci (FMA) připadá pět čtení z paměti (4x sestavení vektoru + 1x koeficient). Výpočetní jednotky tak většinu času pouze čekají na data.
- Limit velikosti konvolučního jádra:** Tato implementace omezuje maximální délku konvolučního jádra, kterou program dokáže zpracovat. Pokud totiž bude jádro příliš velké dojde k pádu výpočtu, způsobeném nutností načíst do limitované sdílené paměti (32 kB) nejen zpracovávanou dlaždici, ale i kompletní okolí definované poloměrem jádra. Zatímco vstupní signál je efektivně rozdělen na bloky, paměťový nárok na toto okolí roste s velikostí jádra a

v určitém bodě kapacitu paměti vyčerpá. Je však nutné podotknout, že tento limit se pohybuje v řádech tisíců vzorků. V kontextu zpracování EEG signálu, kde se typicky využívají kratší filtry, tak toto omezení nepředstavuje pro běžné úlohy reálnou překážku.

3.3 Optimalizovaná CPU implementace

Optimalizace CPU implementace byla primárně cílena na konvoluci s většími jádry (řádově desítky až stovky prvků). Ačkoliv se v praxi často využívají i velmi malá jádra (např. 3–7 prvků [Tan+21]), přínos pokročilých optimalizačních technik (jako je vektorizace či manuální rozbalování smyček) se nejlépe demonstruje na úlohách s vyšší aritmetickou náročností. Právě v těchto případech, kdy výpočet není limitován propustností paměti, lze efektivně maximalizovat využití výpočetních jednotek architektury **Apple Silicon**.

Z hlediska řízení toku výpočtu vycházejí tyto optimalizované implementace přímo z logiky naivní verze, přičemž schéma paralelizace zůstalo nezměněno. To znamená, že sekvenční verze využívá pro iteraci nad daty standardní cyklus `for`, zatímco paralelní varianta deleguje rozdělení práce na systémové API **Grand Central Dispatch** (GCD). Volání `dispatch_apply` je zde opět realizováno s prioritou `QOS_CLASS_USER_INTERACTIVE`. Stejně tak zůstala zachována strategie dělení dat do nezávislých bloků o velikosti definované konstantou `CHUNK_SIZE` (8192 prvků), která se stejně jako u naivní implementace osvědčila jako ideální kompromis mezi granularitou paralelizace a režimem správy vláken.

3.3.1 Skalárni a autovektorizovaná implementace

Skalárni i autovektorizovaná varianta sdílejí totožný zdrojový kód a liší se výhradně instrukcemi pro komplilátor **Clang**, což umožňuje přímo porovnat přínos vektorizace oproti čistě skalárnímu zpracování. Rozdíl spočívá v konfiguraci direktivy `#pragma clang loop vectorize`. Zatímco u skalárni verze je vektorizace explicitně zakázána parametrem `disable`, u autovektorizované verze je naopak povolena parametrem `enable` s nápovědou pro prokládání `interleave_count(4)`. Absence automatické vektorizace u verze bez explicitních direktiv byla ověřena během překladu pomocí diagnostických flagů `-Rpass=loop-vectorize` (vypíše informace o smyčkách, které komplilátor vektorizoval) a `-Rpass-missed=loop-vectorize` (vypíše informace o smyčkách, kde komplilátor vektorizaci zvažoval, ale neprovědl ji). Tuto skutečnost potvrdil i experiment, kdy komplikace kódu s globálně zakázanou vektorizací vedla k identickému výkonu.

Algoritmické optimalizace

Provedené optimalizace se přímo zaměřují na výkonné problémy naivní implementace. Primárním cílem je rozbití silných datových závislostí, maximalizace vytížení instrukční pipeline a optimalizace správy registrů pro minimalizaci redundantních přístupů do paměti. K dosažení těchto cílů byly provedeny následující úpravy:

Změna pořadí smyček. Tato technika nejvýrazněji ovlivnila výkon optimalizované implementace. Zatímco v naivní verzi vnější smyčka prochází data a vnitřní smyčka konvoluční jádro, v optimalizované verzi jsou smyčky otočené, tedy vnější smyčka iteruje přes prvky konvolučního jádra a vnitřní přes vstupní data. Prakticky to znamená, že v naivní implementaci je vnější smyčka „velká“ (iteruje přes data) a vnitřní smyčka „malá“ (iteruje přes jádro). Kvůli tomu, že je vnitřní cyklus krátký, dochází k velmi rychlé obměně hodnot v registrech. Naopak v optimalizované verzi je vnější smyčka malá a vnitřní velká. Díky tomu platí, že jakmile vnější smyčka načte hodnotu koeficientu do registru, provede se s ní velké množství výpočtů (průchod celým blokem dat), než je vyměněna za jinou. Tím se drasticky snižuje frekvence komunikace mezi pamětí L1 a registry, která v původní verzi brzdila výpočet.

Blokové zpracování jádra. Zavedení blokového zpracování bylo nutným krokem pro stabilizaci výkonu u velkých konvolučních jader. Ačkoliv změna pořadí smyček optimalizovala toku dat, při testování velkých poloměrů jader (např. R=512 a více) docházelo k výrazným propadům výkonu. Analýza ukázala, že nejpravděpodobnější příčinou je způsob, jakým se komplilátor snaží distribuovat data mezi RAM, cache a registry. Aby byl výpočet efektivní, pravděpodobně se komplilátor snaží udržet průběžné součety vregistrech, avšak pokud je konvoluční jádro příliš velké, všechna data se už do registrů nevejdou a dojde k přetečení, kvůli kterému je komplilátor nucen využít pomalejší L1 cache.

Funkčním řešením problému s padajícím výkonem se ukázalo rozdělení jádra na fixní bloky o velikosti K_BATCH = 32. Tato hodnota byla zvolena na základě experimentů, kdy jak vyšší, tak nižší hodnoty vykazovaly horší výkon.

Zdrojový kód 3.6: Struktura blokového zpracování

```

1 float* __restrict o_chunk = outputPtr + start;
2 const float* __restrict d_chunk = dataPtr + start;
3
4 size_t k = 0;
5 // Vnější smyčka iteruje přes jádro po blocích (KBatch)
6 for (; k + KBatch <= KernelSize; k += KBatch) {
7
8     // Načtení bloku jádra do lokálního bufferu (registry)

```

```

9   float k_vals[KBatch];
10  for(int i=0; i<KBatch; ++i) k_vals[i] = kernelPtr[k+i];
11
12  // Vnitřní smyčka iteruje přes data (Output Stationary)
13 #pragma clang loop vectorize(enable) interleave_count(4)
14  for (size_t out = 0; out < actualChunkSize; ++out) {
15      // ... výpočet konvoluce ...
16  }
17 }
```

Nezávislost instrukcí a manuální rozbalení smyčky. Pro zajištění nezávislosti instrukcí a maximální vytížení pipeline využívá výpočet **čtverečí** nezávislých akumulátorů (acc0 až acc3). Tento počet byl zvolen s ohledem na latenci instrukce FMA (**Fused Multiply-Add**), která na procesorech **Firestorm** trvá **4 cykly** [Johb]. Rozdelení práce mezi více sčítaců umožňuje efektivní překrytí této latence, kdy pipeline může začít zpracovávat instrukci pro acc1, zatímco výpočet acc0 stále probíhá.

Stejný faktor je využit i pro manuální rozbalení smyčky. Krok smyčky tak přesně koresponduje s kapacitou 128bitového registru **NEON**, který pojme právě čtyři hodnoty typu **float**. Díky tomu může kompilátor snadno nahradit skupinu skalárních operací jedinou vektorovou instrukcí.

Zdrojový kód 3.7: Vnitřní smyčka s unrollingem

```

1 float* __restrict o_chunk = outputPtr + start;
2
3 for (size_t out = 0; out < actualChunkSize; ++out) {
4     float acc0 = 0.0f;
5     float acc1 = 0.0f;
6     float acc2 = 0.0f;
7     float acc3 = 0.0f;
8
9     const float* __restrict current_d = d_chunk + out + k;
10
11    for (int i = 0; i < KBatch; i += 4) {
12        acc0 += current_d[i + 0] * k_vals[i + 0];
13        acc1 += current_d[i + 1] * k_vals[i + 1];
14        acc2 += current_d[i + 2] * k_vals[i + 2];
15        acc3 += current_d[i + 3] * k_vals[i + 3];
16    }
17
18    o_chunk[out] += (acc0 + acc1 + acc2 + acc3);
19 }
```

Prokládání iterací. Direktiva `#pragma clang loop interleave_count(4)` instruuje kompilátor, aby zpracovával 4 iterace smyčky současně. Volba faktoru 4

je přímo svázána s dostupným počtem hardwareových registrů. Pro výpočet jednoho bodu se používají 4 nezávislé akumulátory. Při **prokládání** 4 iterací tak procesor musí udržovat v běhu celkem $4 \times 4 = 16$ akumulátorů současně. Vzhledem k tomu, že architektura **NEON** disponuje celkem 32 registry, je tato hodnota optimální. Polovina registrů (16) je využita pro udržení stavu paralelních výpočtů a druhá polovina zůstává k dispozici pro načítání koeficientů jádra (**k_vals**) a vstupních dat. Například faktor prokládání 6 by již vedl k požadavku na 24 registrů pouze pro akumulátory, což by ponechalo kriticky málo prostoru pro operandy. To by nutilo komplilátor odkládat data do paměti a degradovalo výkon, což potvrdila i experimentální měření.

Zdrojový kód 3.8: Prokládání iterací na CPU

```

1 size_t k = 0;
2 for (; k + KBatch <= KernelSize; k += KBatch) {
3     // ...
4     #pragma clang loop vectorize(enable) interleave_count(4)
5     for (size_t out = 0; out < actualChunkSize; ++out) {
6         // ...
7         for (int i = 0; i < KBatch; i += 4) {
8             acc0 += current_d[i + 0] * k_vals[i + 0];
9             // ...
10            acc3 += current_d[i + 3] * k_vals[i + 3];
11        }
12    }
13 }
```

3.3.2 Manuálně vektorizovaná implementace

Cílem této implementace je pomocí vstavených funkcí pro sadu instrukcí **NEON** co nejpřesněji reprodukovat optimalizační strategii, kterou aplikuje komplilátor u automaticky vektorizované varianty. Kód je strukturován tak, aby explicitně realizoval stejné principy práce s registry, k jakým dochází při automatickém překladu.

Logika výpočtu proto kopíruje prokládání iterací a rozbalení smyčky v identickém rozsahu. Vnitřní smyčka zpracovává 16 výstupních prvků současně, rozdělených do čtyř nezávislých skupin (vektory A, B, C, D). Pro každou skupinu je aplikován unrolling s krokem 4, což odpovídá matici 4×4 vektorových akumulátorů:

- **Vektory A (out 0...3):** registry acc0_A až acc3_A
- **Vektory B (out 4...7):** registry acc0_B až acc3_B
- **Vektory C (out 8...11):** registry acc0_C až acc3_C

- **Vektory D (out 12...15):** registry acc0_D až acc3_D

Tato konfigurace, stejně jako automatická verze, vyhrazuje 16 vektorových registrů pro mezivýsledky. Zbývajících 16 registrů architektury **Apple Silicon** pak slouží pro vstupní hodnoty, konkrétně čtyři registry uchovávají koeficienty jádra a zbytek je využit pro načítání datových bloků pro skupiny A až D.

Pro realizaci samotné vektorizace je nutné použít instrukce vld1q_f32 (načtení bloku dat z paměti do registru), vmulq_f32 (vektorové násobení), vfmaq_f32 (vektorové násobení s přičtením k akumulátoru), vaddq_f32 (sčítání dvou vektorů) a vst1q_f32 (uložení výsledného vektoru zpět do paměti) [Arm]. Protože se však prvky konvolučního jádra v cyklu vybírají po jednom (jsou to skaláry), musí být nejprve použita funkce vdupq_n_f32, která tuto jednu hodnotu replikuje do všech pozic pomocného vektoru $[k, k, k, k]$. Díky tomu může následně instrukce FMA zpracovat vektor vstupních dat a v jediném instrukčním cyklu spočítat 4 hodnoty najednou, jak je ukázáno na schématu 3.3.

$$\underbrace{\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix}}_{\text{Vstupní data}} \times \underbrace{\begin{bmatrix} k \\ k \\ k \\ k \end{bmatrix}}_{\text{Replikovaný koeficient}} = \underbrace{\begin{bmatrix} d_0 \cdot k \\ d_1 \cdot k \\ d_2 \cdot k \\ d_3 \cdot k \end{bmatrix}}_{\text{Vektorový výsledek}} \quad (3.3)$$

Zdrojový kód 3.9: Manuální vektorizace pomocí vstavěných funkcí

```

1 for (int i = 0; i < KBatch; i += 4) {
2     // Replikace koeficientu na všechny pozice vektoru (např.
3     // [k0, k0, k0, k0])
4     float32x4_t k0 = vdupq_n_f32(k_ptr_base[i + 0]);
5     // ... k1, k2, k3
6
6     // Výpočet FMA pro skupinu A (registry acc0_A až acc3_A)
7     acc0_A = vfmaq_f32(acc0_A, vld1q_f32(current_d + i + 0),
8     k0);
9     acc1_A = vfmaq_f32(acc1_A, vld1q_f32(current_d + i + 1),
10    k1);
11    // ... acc2_A, acc3_A
12
13    // ... Identicky pro skupiny B, C a D (posunuté v datech)
14    ...
15
16 // Sečtení 4 dílčích vektorů a přičtení výsledku k celkovému
17 // výsledku
18 float32x4_t sum_A = vaddq_f32(vaddq_f32(acc0_A, acc1_A),
19     vaddq_f32(acc2_A, acc3_A));

```

```
16 vst1q_f32(o_chunk + out, vaddq_f32(vld1q_f32(o_chunk + out),
    sum_A));
```

Vzhledem k fixní velikosti bloku (16 prvků) obsahuje kód dvě úrovně „dočišťovacích“ smyček (vektorovou po 4 prvcích a skalární), které řeší okrajové případy nedělitelné velikostí bloku.

3.4 Optimalizovaná GPU implementace

Hlavním cílem optimalizované implementace bylo maximalizovat využití výpočetního výkonu grafického akcelerátoru a překonat limity paměťové propustnosti, které byly úzkým hrdlem naivní verze. Součástí optimalizačního procesu bylo i odstranění omezení pro maximální velikost konvolučního jádra.

Jelikož tato verze vychází z naivní implementace, sdílí s ní klíčové prvky jako přenos vstupních dat a výstupního bufferu pomocí *Zero-Copy* přístupu a konfiguraci vláken, tedy jak moc je úloha rozdělená na pracovní skupiny (`ITEMS_PER_THREAD` a `THREADS_PER_GROUP`) a jaká je velikost jedné dlaždice (`TILE_SIZE`).

Zásadní změny se odehrály v logice výpočetního kernelu (shaderu), který byl kompletně přepracován s cílem maximalizovat aritmetickou intenzitu a eliminovat redundantní přístupy do paměti. K dosažení těchto vlastností byly implementovány tyto změny:

Zvýšení granularity vlákna. Prvním krokem k vyššímu výkonu bylo radikální zvýšení objemu práce, kterou vykonává jedno vlákno. Oproti naivní verzi, kde vlákno zpracovává pouze jeden vektor (4 hodnoty), optimalizovaná verze počítá konvoluci pro **16 hodnot současně**. To je v kódu realizováno pomocí čtyř nezávislých akumulátorů typu `float4` (`sumA`, `sumB`, `sumC`, `sumD`). Tento počet byl zvolen na základě experimentálního ověření. Testované konfigurace se 3 a 5 akumulátory vykazovaly konzistentně horší výkon než varianta se čtyřmi.

Posuvné okno v registrech. Implementace posuvného okna vregistrech představuje optimalizaci s nejvýznamnějším dopadem na výsledný výkon kernelu. Zatímco naivní verze byla limitována propustností paměti kvůli opakovanému načítání stejných dat, optimalizovaná verze radikálně mění přístup zavedením privátní cache přímo vregistrech vlákna.

Implementace využívá 16 skalárních registrů (v0 až v15), které tvoří posuvné okno nad vstupními daty. Před vstupem do hlavní smyčky se tyto registry naplní prvními 16 hodnotami ze sdílené paměti. V každém kroku se pak provede konvoluční součin mezi daty uloženými vregistrech a příslušnou vahou konvolučního jádra, přičemž výsledky se sčítají do výstupních akumulátorů. Jakmile je výpočet pro daný krok hotov, okno se posune, což znamená, že se hodnota v nejstarším

registru v0 zahodí a obsah ostatních registrů se přesune o jednu pozici vlevo (v1 → v0, v2 → v1 atd.). Tím se uvolní poslední registr v15, do kterého se nače **jediná** nová hodnota ze sdílené paměti.

Zdrojový kód 3.10: Princip posuvného okna v hlavní smyčce

```

1 float v0 = scalarCache[localBaseIndex + 0];
2 float v1 = scalarCache[localBaseIndex + 1];
3 // ...
4 float v15 = scalarCache[localBaseIndex + 15];
5
6 for (int k = 0; k < kernelSize; ++k) {
7     float w = convKernel[k];
8     sumA = fma(float4(v0, v1, v2, v3),      float4(w), sumA);
9     sumB = fma(float4(v4, v5, v6, v7),      float4(w), sumB);
10    sumC = fma(float4(v8, v9, v10, v11),    float4(w), sumC);
11    sumD = fma(float4(v12, v13, v14, v15),   float4(w), sumD);
12
13    float next = scalarCache[localBaseIndex + k + 16];
14
15    v0 = v1;  v1 = v2;  v2 = v3;  v3 = v4;
16    v4 = v5;  v5 = v6;  v6 = v7;  v7 = v8;
17    v8 = v9;  v9 = v10; v10 = v11; v11 = v12;
18    v12 = v13; v13 = v14; v14 = v15; v15 = next;
19 }
```

Díky tomu, že se data do registrů opakovaně nenačítají, ale pouze se v nich přesouvají, dosahuje tato metoda výrazně vyšší aritmetické intenzity než naivní implementace. Zatímco u naivní verze vyžadovala každá instrukce FMA samostatný přístup do sdílené paměti (poměr 1 čtení : 1 FMA), v optimalizované verzi připadá na **jedno čtení celkem 16 instrukcí FMA**. Úzké hrdlo výpočtu se tak přesouvá z paměťové sběrnice na aritmetické jednotky GPU.

Segmentace konvolučního jádra. Aby bylo možné zpracovávat konvoluční filtry libovolné délky a nedocházelo k vyčerpání limitované kapacity sdílené paměti (32 kB na skupinu vláken), byl do optimalizované verze zaveden princip segmentace konvolučního jádra. Vnější smyčka kernelu tak nejde přes celé jádro najednou, ale iteruje po blocích o fixní velikosti KERNEL_SEGMENT_SIZE.

Velikost tohoto bloku nelze volit náhodně. Musí být nastavena v přímé závislosti na počtu vláken ve skupině (THREADS_PER_GROUP) tak, aby celkový objem dat potřebný pro jednu iteraci výpočtu nepřekročil hardwarový limit paměti. Celková paměťová náročnost je dána vztahem 3.4.

$$M_{req} \approx (TILE_SIZE + KERNEL_SEGMENT_SIZE) \times \text{sizeof}(float) \quad (3.4)$$

Pro implementaci byla zvolena konfigurace KERNEL_SEGMENT_SIZE = 1024 a THREADS_PER_GROUP = 256. Vzhledem k tomu, že velikost TILE_SIZE je dána rovníkem 3.2 (kde ITEMS_PER_THREAD = 16), je hodnota 256 nejvyšší možnou, protože 512 vláken by obsadilo celou dostupnou kapacitu sdílené paměti. Jedinou proměnnou, kterou tak lze ovlivnit, je KERNEL_SEGMENT_SIZE, u které se experimentálně ověřilo, že při jejím zvýšení na 2048 a 4096 i snížení na 512 dochází k poklesu výkonu.

V každé fázi smyčky se tak do sdílené paměti načte pouze nezbytný úsek vstupních dat a provede se výpočet. Před přechodem na další segment je ale nutná synchronizace bariérou (threadgroup_barrier). Bez ní by rychlejší vlákna začala do sdílené cache načítat data pro nový segment a přepsala by obsah sdílené paměti dříve, než by ji pomalejší vlákna stihla využít pro výpočet aktuálního segmentu.

Zdrojový kód 3.11: Implementace segmentovaného zpracování konvoluce

```

1 for (int k_base = 0; k_base < (int)kernelSize; k_base +=  
     KERNEL_SEGMENT_SIZE) {  
2     // Určení délky aktuálního segmentu (bude plný segment,  
     // nebo zbytek)  
3     int currentSegmentLen = min((int)KERNEL_SEGMENT_SIZE, (int)kernelSize - k_base);  
4  
5     // Výpočet potřebného množství dat  
6     int elementsNeeded = TILE_SIZE + currentSegmentLen - 1;  
7     int vectorsNeeded = (elementsNeeded + 3) / 4;  
8     int dataOffset = groupStartGlobal + k_base;  
9  
10    // ... Kooperativní načítání dat do sdílené paměti ...  
11  
12    threadgroup_barrier(mem_flags::mem_threadgroup);  
13  
14    // ... Vlastní výpočet nad aktuálním segmentem ...  
15  
16 }
```

Uživatelské rozhraní a ovládání

4

Program je navržen jako konzolová aplikace (CLI). Interakce s uživatelem probíhá formou textového průvodce, který postupně konfiguruje parametry pro zpracování EEG signálu.

4.1 Konfigurace a vstupy

Po spuštění aplikace je uživatel vyzván k zadání série konfiguračních parametrů. Ovládání se řídí následujícími pravidly:

- **Potvrzení volby:** Pro potvrzení zadанé hodnoty nebo pro výběr výchozí (defaultní) hodnoty slouží klávesa **[Enter]**. Výchozí hodnoty jsou vždy uvedeny v závorce u příslušné výzvy.
- **Návrat zpět:** Pokud uživatel udělá chybu nebo chce změnit předchozí volbu, může zadat znak **[b]** (back), čímž se vrátí o krok zpět v konfiguračním procesu.
- **Validace vstupů:** Vstupy jsou automaticky kontrolovány. U číselných voleb (např. výběr módu) se kontroluje rozsah povolených hodnot. Při zadávání cesty k výstupnímu souboru aplikace ověří, zda se jedná o validní cestu ke složce. Pokud složka neexistuje, program se ji pokusí vytvořit.

Aplikace umožňuje konfigurovat následující parametry běhu:

1. **Vstupní dataset:** Cesta k souboru ve formátu EDF. Specifickou funkcionalitou je, že když výchozí dataset Siena Scalp EDF není na disku nalezen, program automaticky využije nástroj curl k jeho stažení z internetu.
2. **Mód zpracování:** Výběr konkrétní implementace algoritmu (sekvenční/parallelní verze na CPU, různé úrovně vektorizace, nebo implementace na GPU).
3. **Počet iterací:** Nastavení počtu opakování konvoluce pro získání statisticky relevantních výsledků měření (výchozí hodnota je 10).

4. **Uložení výstupu:** Volba, zda se mají vyfiltrovaná data uložit, a specifikace cílové složky.

4.2 Prezentace výstupů a metrik

Během zpracování dat aplikace průběžně informuje uživatele o stavu výpočtu pomocí standardního výstupu. Před zahájením benchmarku jsou vypsány parametry konvolučního jádra (typ Gaussovo jádro, velikost, poloměr a sigma) a parametry načtených dat (počet kanálů, vzorků a celková velikost v bajtech).

Po dokončení sady měření program vypíše detailní souhrn, který obsahuje:

- **Time Breakdown:** Průměrný celkový čas běhu, čistý čas výpočtu (compute) a čas strávený paměťovými operacemi pro CPU i GPU. U GPU verze je navíc vyčíslen overhead spojený s API a spuštěním kernelu.
- **Metrics:** Výkonnostní metriky zahrnující propustnost (Throughput) v MSamples/s a výpočetní výkon v GFLOPS. Obě metriky jsou počítané z času výpočtu.

Ukázka z reálného běhu programu, demonstrující konfiguraci, spuštění benchmarku pro CPU a GPU varianty a využití funkce návratu v menu, je uvedena ve výpisu 4.1.

Výpis 4.1: Ukázka běhu programu s konfigurací a výsledky benchmarku

```

1 =====
2          Welcome to
3      EEG Linear Filter Benchmark Suite
4 =====
5 Controls:
6  [ENTER]: Confirm default value
7  'b'     : Go back to previous step
8
9 Enter path to the input EDF file:
10 (Default: Siena Scalp EEG - 1.0.0/PN01/PN01-1)
11 >
12
13 Select benchmark mode:
14 -1 - WHOLE_BENCHMARK_SUITE (Default)
15  0 - CPU_SEQ_APPLE
16 ...
17 10 - GPU_32BIT
18 > 0
19
20 Enter number of benchmark iterations
21 (Default: 10)
22 >
```

```
23
24 Do you want to save the results? (y/n):
25 (Default n)
26 >
27 =====
28 Configuration complete. Starting...
29 =====
30 Convolution kernel: Gaussian
31 Size: 513 | Radius: 256 | Sigma: 1
32 Loading file: EegLinearFilter/data/PN01-1.edf
33 ...
34 =====
35 Starting benchmark suite
36 =====
37 Mode: CPU_SEQ_APPLE
38
39 Run 1: 14.1087s
40 ...
41 Run 10: 12.7284s
42
43 AVG results over 10 runs:
44 Time Breakdown:
45   Total: 13.1627s
46   Compute: 12.9699s
47   Mem Ops: 0.192749s (CPU)
48 Metrics:
49   Throughput: 67.0905 MSamples/s
50   Performance: 68.8349 GFLOPS
51 =====
52 Done!
53 =====
54 Do you want to run another benchmark? (y/n):
55 > n
56 Exiting application. Goodbye!
```

Validace a správnost výsledků

5

Vzhledem k absenci referenčního analytického řešení byla pro ověření korektnosti implementace zvolena metoda křížové validace. Předpokladem je, že pokud různé implementace algoritmu (sekvenční, paralelní, vektorizované, GPU) dospějí ke shodným výsledkům nad stejnými daty, lze implementaci považovat za správnou.

Významnou roli při validaci hraje také referenční implementace využívající optimalizovanou knihovnou funkci vDSP_conv z frameworku Apple Accelerate (mód CPU_SEQ_APPLE).

5.1 Metodika porovnání

Pro automatizované porovnání výstupů všech implementovaných módů byl vytištěn validační skript `results_similarity_by_mode.py` v jazyce Python. Ten provádí komparaci "každý s každým" na úrovni jednotlivých vzorků uložených ve výstupních souborech EDF.

Jelikož formát EDF ukládá data jako 16-bitová celá čísla (integer) [Kem], porovnává skript přímo tyto hodnoty. Vlivem odlišného průběhu výpočtu v plovoucí rádové čárce při různých úrovních optimalizace se nepodařilo zcela eliminovat drobné odchylinky. Rozdíl o hodnotě 1 (na úrovni celočíselné reprezentace) je proto klasifikován jako akceptovatelná zaokrouhlovací chyba.

Validační skript generuje tři typy výstupů:

1. **Souhrnný report v konzoli:** Tabulka shody mezi všemi páry módů.
2. **CSV reporty:** Detailní seznam míst, kde došlo k neshodě.
3. **Vizualizace (PDF):** Grafické znázornění distribuce chyb v signálu.

5.2 Výsledky validace

Jako testovací dataset byl použit Siena Scalp EEG - 1.0.0/PN01/PN01-1 který má následující parametry:

- Počet signálů: 35
- Vzorků na signál: 24 861 184 (celkem cca 870 milionů vzorků)
- Velikost dat: 1659 MB

Níže uvedený výpis 5.1 zobrazuje kompletní tabulkou porovnání. Sloupec *MATCH %* indikuje procentuální shodu, *DIFF COUNT* absolutní počet odlišných vzorků a *MAX DIFF* maximální nalezený rozdíl v hodnotě některého ze vzorků.

Tabulka 5.1: Tabulka porovnání podobnosti výsledků pro dataset PN01-1

File A	File B	Match %	Diff Count	Max Diff
CPU_PAR_AUTO_VEC	CPU_PAR_MANUAL_VEC	100.00%	9577	1
CPU_PAR_AUTO_VEC	CPU_PAR_NAIVE	100.00%	10795	1
CPU_PAR_AUTO_VEC	CPU_PAR_NO_VEC	100.00%	2240	1
CPU_PAR_AUTO_VEC	CPU_SEQ_APPLE	100.00%	11530	1
CPU_PAR_AUTO_VEC	CPU_SEQ_AUTO_VEC	100.00%	0	0
CPU_PAR_AUTO_VEC	CPU_SEQ_MANUAL_VEC	100.00%	9577	1
CPU_PAR_AUTO_VEC	CPU_SEQ_NAIVE	100.00%	10795	1
CPU_PAR_AUTO_VEC	CPU_SEQ_NO_VEC	100.00%	2240	1
CPU_PAR_AUTO_VEC	GPU_32BIT	100.00%	26325	1
CPU_PAR_AUTO_VEC	GPU_NAIVE	100.00%	26325	1
CPU_PAR_MANUAL_VEC	CPU_PAR_NAIVE	100.00%	9018	1
CPU_PAR_MANUAL_VEC	CPU_PAR_NO_VEC	100.00%	10349	1
CPU_PAR_MANUAL_VEC	CPU_SEQ_APPLE	100.00%	10073	1
CPU_PAR_MANUAL_VEC	CPU_SEQ_AUTO_VEC	100.00%	9577	1
CPU_PAR_MANUAL_VEC	CPU_SEQ_MANUAL_VEC	100.00%	0	0
CPU_PAR_MANUAL_VEC	CPU_SEQ_NAIVE	100.00%	9018	1
CPU_PAR_MANUAL_VEC	CPU_SEQ_NO_VEC	100.00%	10349	1
CPU_PAR_MANUAL_VEC	GPU_32BIT	100.00%	25994	1
CPU_PAR_MANUAL_VEC	GPU_NAIVE	100.00%	25994	1
CPU_PAR_NAIVE	CPU_PAR_NO_VEC	100.00%	10533	1
CPU_PAR_NAIVE	CPU_SEQ_APPLE	100.00%	10855	1
CPU_PAR_NAIVE	CPU_SEQ_AUTO_VEC	100.00%	10795	1
CPU_PAR_NAIVE	CPU_SEQ_MANUAL_VEC	100.00%	9018	1
CPU_PAR_NAIVE	CPU_SEQ_NAIVE	100.00%	0	0
CPU_PAR_NAIVE	CPU_SEQ_NO_VEC	100.00%	10533	1
CPU_PAR_NAIVE	GPU_32BIT	100.00%	26298	1

(pokračování na další straně)

Tabulka 5.1 – pokračování z předchozí strany

File A	File B	Match	Diff Count	Max Diff
CPU_PAR_NAIVE	GPU_NAIVE	100.00%	26298	1
CPU_PAR_NO_VEC	CPU_SEQ_APPLE	100.00%	11870	1
CPU_PAR_NO_VEC	CPU_SEQ_AUTO_VEC	100.00%	2240	1
CPU_PAR_NO_VEC	CPU_SEQ_MANUAL_VEC	100.00%	10349	1
CPU_PAR_NO_VEC	CPU_SEQ_NAIVE	100.00%	10533	1
CPU_PAR_NO_VEC	CPU_SEQ_NO_VEC	100.00%	0	0
CPU_PAR_NO_VEC	GPU_32BIT	100.00%	26499	1
CPU_PAR_NO_VEC	GPU_NAIVE	100.00%	26499	1
CPU_SEQ_APPLE	CPU_SEQ_AUTO_VEC	100.00%	11530	1
CPU_SEQ_APPLE	CPU_SEQ_MANUAL_VEC	100.00%	10073	1
CPU_SEQ_APPLE	CPU_SEQ_NAIVE	100.00%	10855	1
CPU_SEQ_APPLE	CPU_SEQ_NO_VEC	100.00%	11870	1
CPU_SEQ_APPLE	GPU_32BIT	100.00%	25439	1
CPU_SEQ_APPLE	GPU_NAIVE	100.00%	25439	1
CPU_SEQ_AUTO_VEC	CPU_SEQ_MANUAL_VEC	100.00%	9577	1
CPU_SEQ_AUTO_VEC	CPU_SEQ_NAIVE	100.00%	10795	1
CPU_SEQ_AUTO_VEC	CPU_SEQ_NO_VEC	100.00%	2240	1
CPU_SEQ_AUTO_VEC	GPU_32BIT	100.00%	26325	1
CPU_SEQ_AUTO_VEC	GPU_NAIVE	100.00%	26325	1
CPU_SEQ_MANUAL_VEC	CPU_SEQ_NAIVE	100.00%	9018	1
CPU_SEQ_MANUAL_VEC	CPU_SEQ_NO_VEC	100.00%	10349	1
CPU_SEQ_MANUAL_VEC	GPU_32BIT	100.00%	25994	1
CPU_SEQ_MANUAL_VEC	GPU_NAIVE	100.00%	25994	1
CPU_SEQ_NAIVE	CPU_SEQ_NO_VEC	100.00%	10533	1
CPU_SEQ_NAIVE	GPU_32BIT	100.00%	26298	1
CPU_SEQ_NAIVE	GPU_NAIVE	100.00%	26298	1
CPU_SEQ_NO_VEC	GPU_32BIT	100.00%	26499	1
CPU_SEQ_NO_VEC	GPU_NAIVE	100.00%	26499	1
GPU_32BIT	GPU_NAIVE	100.00%	0	0

Z tabulky je patrné, že maximální chyba (MAX DIFF) nikdy nepřekročila hodnotu 1. Počet neshod se pohybuje v řádech desítek tisíc, což je v kontextu celkového počtu 870 milionů vzorků statisticky zanedbatelné a potvrzuje funkční ekvivalenci všech implementací.

5.2.1 Analýza odchylek

Pro hlubší zkoumání chyb generuje skript CSV soubory mapující přesnou polohu a hodnotu odchylek. Ukázka obsahu takového souboru je ve výpisu 5.1. Sloupce identifikují kanál, časový index vzorku, surové hodnoty z obou porovnávaných souborů a velikost rozdílu.

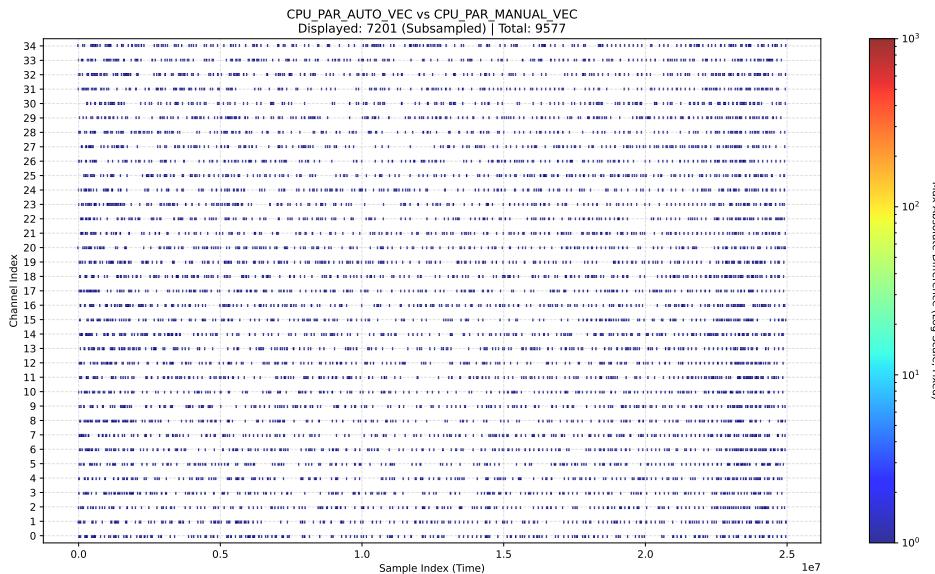
Výpis 5.1: Ukázka CSV reportu s detekovanými rozdíly

```

1 Channel , Sample Index , Value A (Raw) , Value B (Raw) , Diff
2 34 , 2914 , 15312 , 15313 , 1
3 20 , 3159 , 347 , 346 , 1
4 31 , 4096 , 3955 , 3956 , 1
5 31 , 7513 , 3981 , 3980 , 1
6 1 , 11624 , 17480 , 17481 , 1
7 ...

```

Pro vizuální kontrolu, zda se chyby nevyskytují v shlucích (což by mohlo indikovat systematickou chybu algoritmu), jsou generovány vizualizace ve formátu PDF. Obrázek 5.1 ukazuje srovnání automatické a manuální vektorizace. Body v grafu reprezentují místa, kde došlo k neshodě. Z rovnoměrného rozložení chyb napříč kanály i časem lze usuzovat, že se jedná o náhodné zaokrouhlovací chyby bez lokálních anomalií.



Obrázek 5.1: Porovnání výstupů módů CPU_PAR_AUTO_VEC a CPU_PAR_MANUAL_VEC.

Kompletní výsledky provedené analýzy jsou k dispozici v elektronické příloze této práce.

Výkonnostní analýza

6

Tato kapitola analyzuje přínos implementovaných optimalizačních technik na základě výsledků syntetických testů. Analýza se zaměřuje na porovnání výkonu v závislosti na velikosti konvolučního jádra a objemu dat. Dále hodnotí efektivitu využití hardwarových prostředků a škálovatelnost řešení na platformě Apple Silicon. Součástí vyhodnocení je i analýza limitů a efektivity paralelizace s využitím Amdahlova, Gustafsonova a Karp-Flattova zákona. Veškeré naměřené hodnoty a detailní analýzy jsou k dispozici v příloze této práce.

6.1 Metodika měření

Pro zajištění reprodukovatelnosti výsledků byl stanoven striktní testovací protokol. Referenčním zařízením je **MacBook Pro 14 (2021)** s čipem **Apple M1 Pro** (8 CPU jader, 14 GPU jader), napájený ze sítě pro vyloučení vlivu správy energie na stabilitu výkonu.

Eliminace vnějších vlivů na měření proběhla následovně:

- Ukončení všech uživatelských aplikací.
- Deaktivace síťových rozhraní a automatických aktualizací.
- Absence jakékoliv uživatelské interakce během testu (zamezení I/O přerušení).

6.1.1 Sběr dat

Měření výkonu probíhalo napříč všemi implementovanými verzemi kódu (zahrnující CPU varianty s různou úrovní vektorizace i GPU implementace). Testy sledovaly chování algoritmů při změně velikosti konvolučního jádra i objemu vstupních dat.

Aby byly výsledky statisticky průkazné, byla každá konfigurace změřena celkem 40krát. Schéma měření bylo následující:

1. Algoritmus se spustil v **10 iteracích** bezprostředně po sobě.

2. Tento desetinnásobný cyklus se zopakoval ve **4 nezávislých spuštěních**.

Každý jednotlivý běh byl zaznamenán, čímž vznikl dataset obsahující 7040 záznámů, které byly následně analyzovány.

6.1.2 Statistické zpracování a metriky

K eliminaci odlehlých hodnot způsobených nedeterminismem operačního systému byla použita dvoustupňová agregace mediánem:

1. **Agregace běhu:** Medián z 10 bezprostředně za sebou proběhlých iterací (eliminace vlivu studeného startu hardwaru).
2. **Finální hodnota:** Medián ze 4 agregovaných běhů (výsledná hodnota konfigurace).

Jako hlavní srovnávací metrika byl zvolen výpočetní výkon v **GFLOPS**. Tato volba umožňuje nejen porovnání nezávisle na délce trvání úlohy, ale také vztažení výsledků k teoretickému maximálnímu výkonu hardwaru, čímž lze přímo kvantifikovat efektivitu využití výpočetních jednotek. Výpočet vychází z čistého času výpočtu $T_{compute}$:

$$P = \frac{N_{samples} \times (2 \cdot R + 1) \times 2}{T_{compute} \times 10^9} \quad (6.1)$$

Kde $N_{samples}$ značí počet vzorků, R poloměr jádra a násobitel 2 zohledňuje operaci FMA (Fused Multiply-Add) jako dvě operace v plovoucí řádové čárce.

6.2 Porovnání implementací

Tato sekce srovnává všechny realizované přístupy na fixní konfiguraci úlohy. Pro testování byl zvolen reprezentativní scénář nad datasetem PN01-1.edf (870 milionů vzorků, 1659 MB) s konvolučním jádrem o poloměru $R = 256$ (celková velikost 513 prvků). Zvolená konfigurace představuje dostatečně robustní úlohu, kde se již plně projevují benefity vektorizace, paralelizace i akcelerace na GPU a minimalizuje se vliv režie spouštění vláken či kernelů.

6.2.1 Souhrnné výsledky

Naměřené výsledky v tabulce 6.1 odhalují tři zřetelné výkonné úrovně: implementace na GPU, paralelní CPU implementace a sekvenční CPU algoritmy. Toto rozvrstvení potvrzuje i graf 6.1. Z výsledků je patrné, že pro úlohu 1D lineární konvoluce je paralelizace výpočtu velmi přínosná, neboť i naivní paralelní implementace svým výkonem předčily veškerá sekvenční řešení.

Tabulka 6.1: Srovnání výkonnostních parametrů jednotlivých implementací ($R = 256$, Dataset PN01-1)

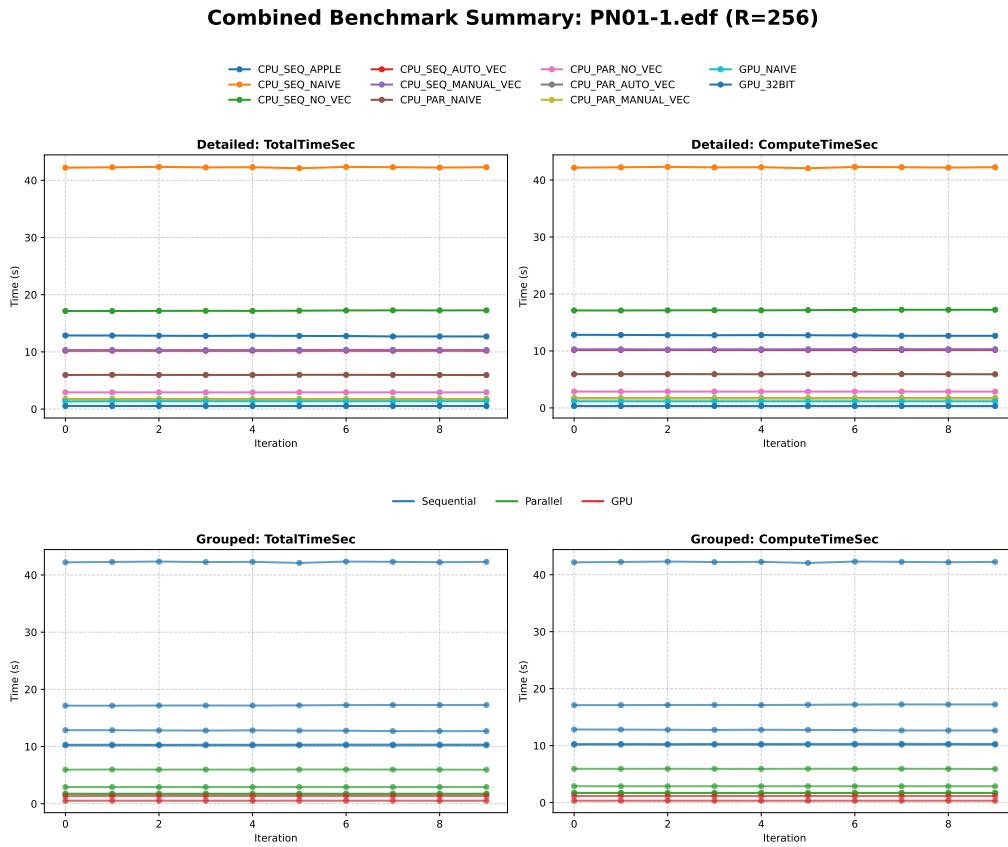
Mód	Medián celkového času	Čas výpočtu	Výkon
	[s]	[s]	[GFLOPS]
<i>GPU Implementace</i>			
GPU_32BIT	0.5333	0.3312	2695.52
GPU_NAIVE	1.3660	1.1615	768.66
<i>CPU Paralelní implementace (8 jader)</i>			
CPU_PAR_MANUAL_VEC	1.7247	1.6871	529.17
CPU_PAR_AUTO_VEC	1.7385	1.6996	525.29
CPU_PAR_NO_VEC	2.9227	2.8713	310.93
CPU_PAR_NAIVE	5.9735	5.9230	150.73
<i>CPU Sekvenční implementace (1 jádro)</i>			
CPU_SEQ_AUTO_VEC	10.2102	10.1744	87.75
CPU_SEQ_MANUAL_VEC	10.3050	10.2692	86.94
CPU_SEQ_APPLE (vDSP)	12.7113	12.6721	70.45
CPU_SEQ_NO_VEC	17.1507	17.1123	52.17
CPU_SEQ_NAIVE	42.2808	42.2409	21.14

Detailní pohled na data odhaluje zajímavý fenomén při srovnání automatické (AUTO_VEC) a manuální (MANUAL_VEC) vektorizace. Ačkoliv jsou si výkonnostně velmi blízké, měření vykazují konzistentní trend. Zatímco v sekvenčním režimu mírně vítězí automatická optimalizace (87,75 GFLOPS oproti 86,94 GFLOPS), při paralelním zapojení všech jader se situace obrací ve prospěch manuálního přístupu (529,17 GFLOPS oproti 525,29 GFLOPS). Tento jev je pravděpodobně způsoben rozdíly v efektivitě distribuce dat a práce s pamětí. Zatímco při paralelném zpracování se drobné zdržení jednoho jádra ztratí v plynulém výkonu ostatních, v sekvenčním režimu tento kompenzační efekt chybí. Každá neefektivita se tak naplno projeví ve výsledném čase, což nahrává preciznosti optimalizací komplikátoru.

Vzájemné poměry zrychlení mezi všemi testovanými metodami jsou vizualizovány formou matice na obrázku 6.2. Zde je jasně vidět dominantní postavení optimalizované GPU verze, která dosahuje zrychlení 128× oproti základní naivní implementaci na CPU.

6.2.2 Analýza CPU výkonu a konfrontace s teorií

Základní naivní implementace (CPU_SEQ_NAIVE) dosahuje výkonu 21 GFLOPS. Díky optimalizacím stoupá výkon sekvenčního zpracování na cca 87 GFLOPS, což představuje **4,15násobné zrychlení**. Teoretický maximální výkon jádra Firestorm je odhadován na 103 GFLOPS a tak optimalizované sekvenční verze dosahují přibližně



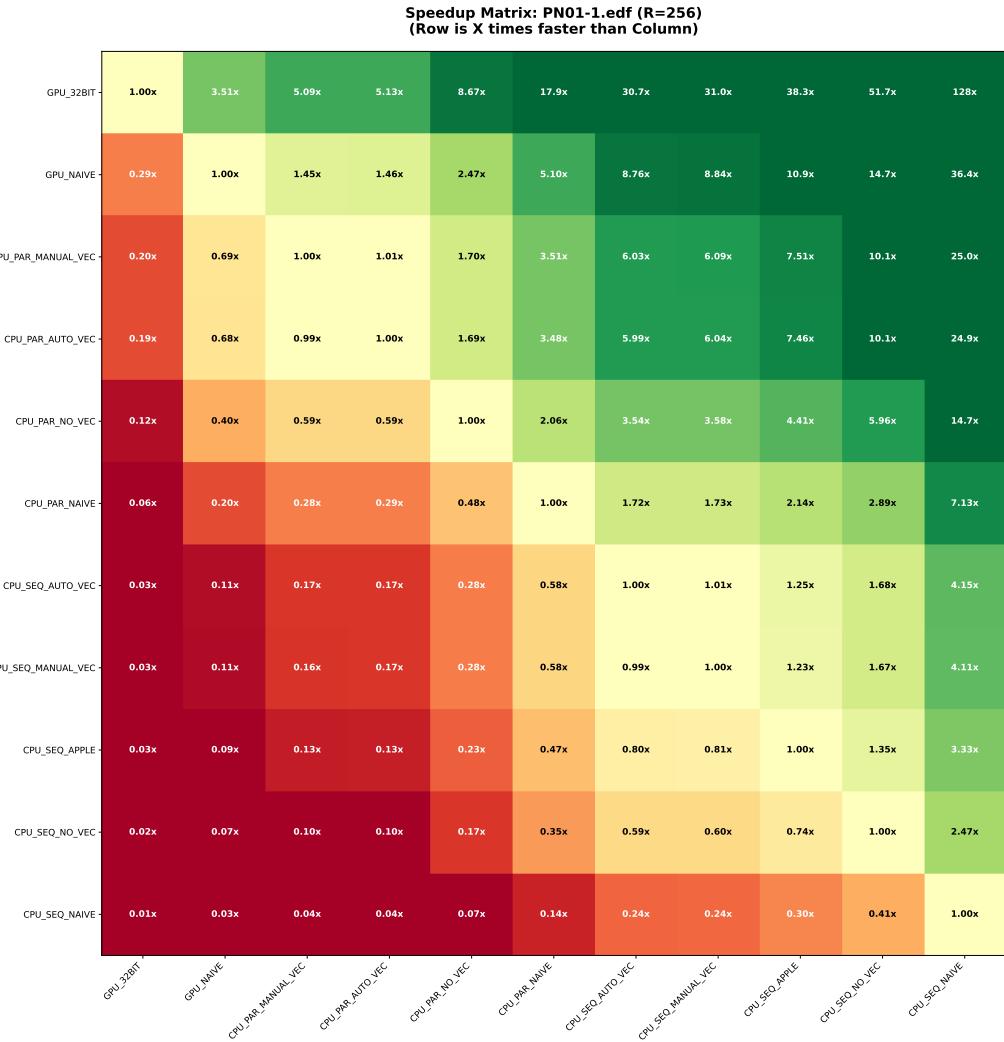
Obrázek 6.1: Souhrnné porovnání celkového a výpočetního času všech implementací ($R = 256$, Dataset PN01-1).

85 % teoretického maxima. Zajímavé je srovnání s knihovnou Apple Accelerate (CPU_SEQ_APPLE), která dosahuje 70,45 GFLOPS. Vlastní implementace optimalizovaná pro specifický typ konvoluce tak v tomto případě překonává obecnou knihovnou funkci vDSP_conv.

Při zapojení všech 8 jader škáluje výkon na 525 GFLOPS – 529 GFLOPS. Teoretický limit celého CPU (multicore) se pohybuje v rozmezí 660–700 GFLOPS. Dosažená hodnota tedy odpovídá přibližně **75–80 % teoretického maxima** celého čipu.

Dalším důležitým ukazatelem je, že stabilita optimalizovaného sekvenčního i paralelního výpočtu je vysoká (viz obr. 6.3). Rozptyl naměřených časů je minimální a pohybuje se v řádu desítek milisekund (odchylka pod 2 %).

6.2.3 Analýza GPU výkonu a hardwareové limity

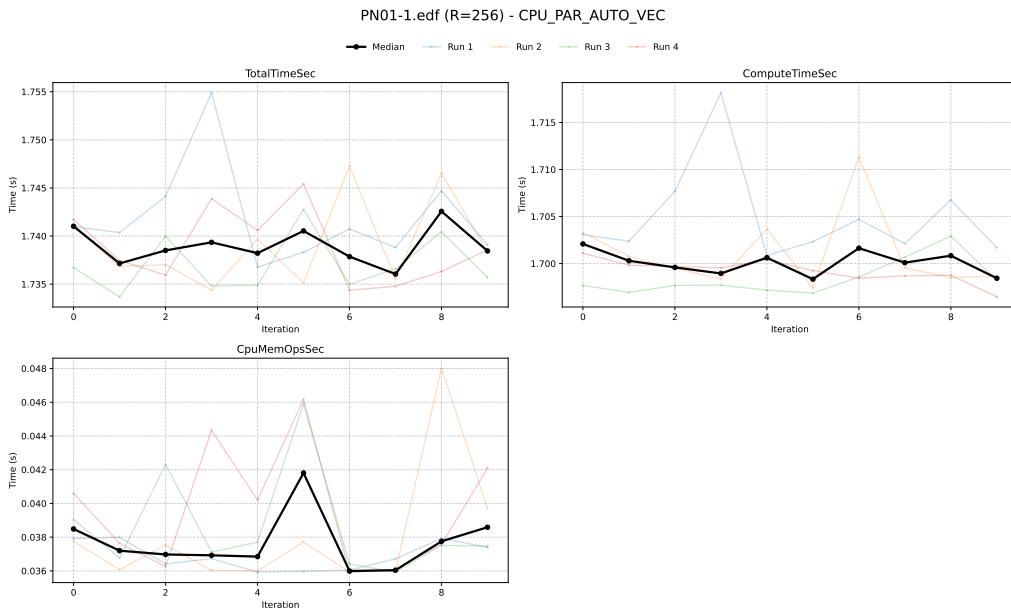


Obrázek 6.2: Matice zrychlení (Speedup Matrix) pro dataset PN01-1 a jádro $R = 256$. Hodnoty udávají, kolikrát je metoda v řádku rychlejší než metoda ve sloupci.

6.2.3 Analýza GPU výkonu a hardwareové limity

Nejvýkonnější implementací je GPU_32BIT s výkonem **2695 GFLOPS** (2.7 TFLOPS). Oproti naivní GPU verzi (768 GFLOPS) přináší provedené optimalizace nárůst výkonu o 350 %.

Dle analýz třetích stran [Nota] dosahuje GPU čipu M1 Pro teoretického výkonu až 4,6 TFLOPS. Naměřená hodnota 2,7 TFLOPS tak představuje přibližně 59 % tohoto odhadovaného maxima. Pro ověření, zda je tento limit dán neefektivitou implementace či vlastnostmi hardwaru, byl proveden srovnávací benchmark pomocí knihovny **PyTorch** (MPS backend). Tento test (skript `measure_gpu_power.py`) realizoval konvoluci nad synteticky generovanými daty o shodných rozměrech a na-



Obrázek 6.3: Průběh výpočtu módu CPU_PAR_AUTO_VEC v čase (10 iterací, 4 běhy) pro $R = 256$ a dataset PN01-1.

měřil trvalý výkon **2561 GFLOPS**.

Vlastní implementace v jazyce Metal tedy mírně překonává (o cca 5 %) vysoce optimalizovanou knihovnu PyTorch. To naznačuje, že hodnota kolem 2.7 TFLOPS je pro tento typ operace (1D konvoluce) na daném hardwaru reálným maximem.

6.2.4 Teoretická analýza paralelizace

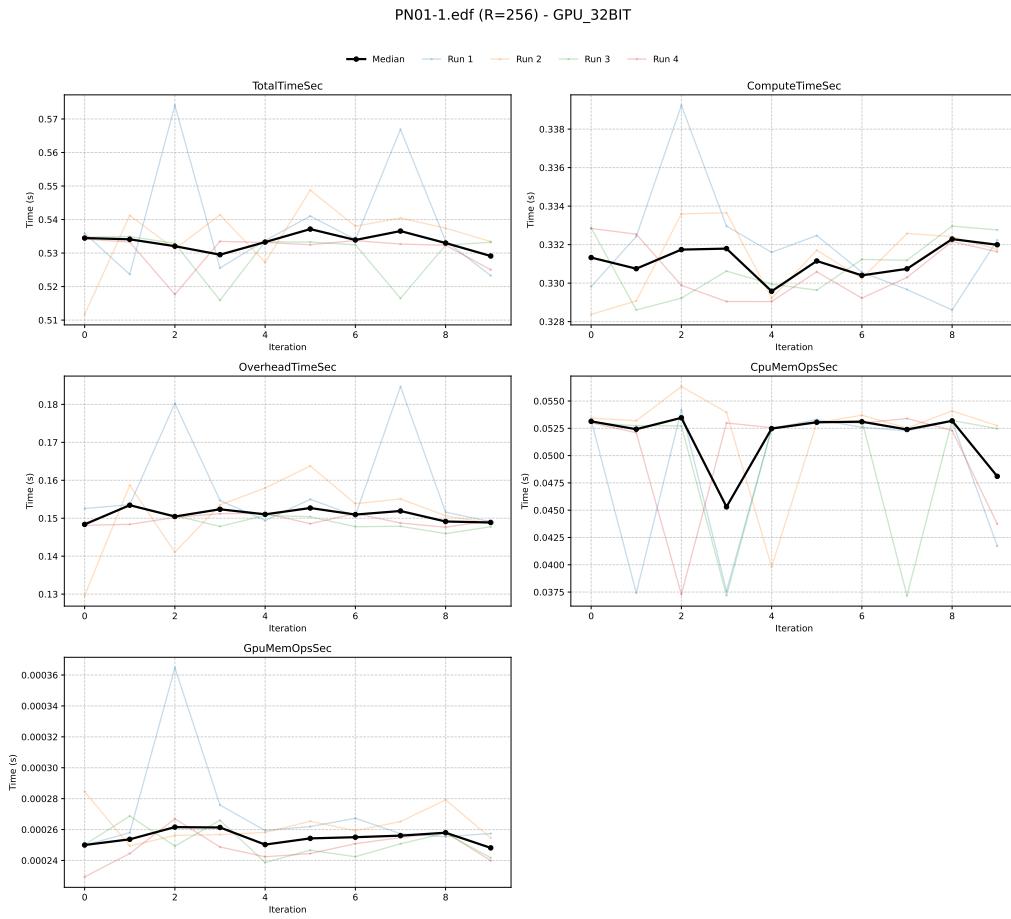
Pro kvantifikaci efektivity paralelního zpracování byly na implementované algoritmy aplikovány teoretické metriky zrychlení (S), Karp-Flattova metrika (e) a Gustafsonův zákon (S_{scaled}). Tyto metriky jsou pro počet jader $P = 8$ definovány vztahy:

$$S = \frac{T_s}{T_p}, \quad e = \frac{\frac{1}{S} - \frac{1}{P}}{1 - \frac{1}{P}}, \quad S_{scaled} = P - (P - 1)e \quad (6.2)$$

Dále byl pomocí Amdahlova zákona odvozen teoretický limit maximálního zrychlení (S_{limit}) pro hypotetický nekonečný počet jader:

$$S_{limit} = \lim_{P \rightarrow \infty} \frac{1}{e + \frac{1-e}{P}} = \frac{1}{e} \quad (6.3)$$

Pro přesné zhodnocení byly do těchto vztahů dosazeny časy čistého výpočtu ($T_{compute}$), které nezahrnují alokaci a nulování paměti. Tím je odfiltrována režie přípravy dat a izolována efektivita samotného konvolučního algoritmu včetně režie



Obrázek 6.4: Průběh výpočtu módu GPU_32BIT v čase (10 iterací, 4 běhy) pro $R = 256$ a dataset PN01-1.

správy vláken. Výsledné hodnoty pro všechny implementační varianty shrnuje tabulka 6.2.

Tabulka 6.2: Metriky paralelizace vypočtené z časů výpočtů ($P = 8$)

Implementace	T_s [s]	T_p [s]	Zrychlení (S)	Karp-Flatt (e)	Gustafson (S_{scaled})	Limit (S_{limit})
NAIVE	42,24	5,92	7,14	0,017 (1,7 %)	7,88	58,8
NO_VEC	17,11	2,87	5,96	0,049 (4,9 %)	7,66	20,4
AUTO_VEC	10,17	1,70	5,98	0,048 (4,8 %)	7,66	20,8
MANUAL_VEC	10,27	1,69	6,07	0,045 (4,5 %)	7,68	22,2

Nejvyšší hodnoty zrychlení ($S = 7, 13$) a nejnižší sériovou režii ($e = 1, 7\%$) vykazuje naivní implementace. Kvůli silné datové závislosti a nevyužívání registrů je výpočet limitován rychlosťí samotného výpočtu, nikoliv propustností paměti. Jed-

notlivá vlákna tak nezahlcují sběrnici a nebrzdí se navzájem, díky čemuž se tato verze blíží ideálnímu lineárnímu škálování.

Naopak u optimalizovaných variant (NO_VEC, AUTO_VEC a MANUAL_VEC) dochází k poklesu zrychlení na úroveň $S \approx 6,0$ a nárůstu sériové režie na cca 4,5–4,9 %. Tento jev ukazuje, že algoritmické optimalizace zkrátily dobu výpočtu natolik, že se úzkým hrdlem stala propustnost paměťové sběrnice a tak škálovatelnost s přidáváním dalších jader klesá rychleji než u naivní, výpočetně náročné implementace.

6.3 Analýza škálování s velikostí dat

Pro ověření stability a efektivity implementací byl proveden experiment sledující výpočetní výkon v závislosti na rostoucím objemu vstupních dat. Velikost konvolučního jádra byla v tomto testu fixována na $R = 256$. Testované spektrum velikosti vstupů sahá od 1,9 milionu prvků (cca 7,6 MB) až po 1,1 miliardy prvků (cca 4,4 GB). Naměřené hodnoty (GFLOPS) pro tři reprezentativní velikosti dat jsou shrnutы v tabulce 6.3.

Tabulka 6.3: Vývoj výpočetního výkonu [GFLOPS] v závislosti na velikosti vstupních dat ($R = 256$)

Implementace	Malá data	Střední data	Velká data
	(1,9 mil. prvků / 7,6 MB)	(175 mil. prvků / 700 MB)	(1,1 mld. prvků / 4,4 GB)
GPU_32BIT	2 625,92	2 727,01	2 701,68
GPU_NAIVE	700,44	772,24	769,27
CPU_PAR_MANUAL_VEC	516,39	530,25	528,45
CPU_PAR_AUTO_VEC	510,18	526,35	524,66
CPU_PAR_NO_VEC	307,46	312,03	310,77
CPU_PAR_NAIVE	210,91	156,73	150,58
CPU_SEQ_MANUAL_VEC	86,39	87,06	85,97
CPU_SEQ_AUTO_VEC	87,42	87,64	87,01
CPU_SEQ_APPLE	69,85	70,47	70,52
CPU_SEQ_NO_VEC	51,50	51,91	51,29
CPU_SEQ_NAIVE	32,95	21,44	21,13

6.3.1 Linearita výpočtu

Pro analýzu škálovatelnosti bylo sledováno, zda nárůst doby výpočtu odpovídá nárůstu objemu vykonané práce. Jako srovnávací metrika byl zvolen počet operací v plovoucí řádové čárce (G-Ops), nikoliv prostý počet záznamů v datasetu. Tento přístup přesněji vyjadřuje skutečnou zátěž procesoru, jelikož není zkreslen vlivem okrajových podmínek vzniklých při aplikaci konvolučního jádra na hrany vstupních dat a odpovídá reálnému množství aritmetických operací, které musí ALU zpracovat. Objem vykonané práce W v jednotkách G-Ops je definován vztahem:

$$W = \frac{N_{out} \cdot (2R + 1) \cdot 2}{10^9} \quad (6.4)$$

kde N_{out} představuje počet vypočítaných výstupních prvků (zohledňující zkrácení o okraje), R je poloměr konvolučního jádra a člen $(2R + 1)$ odpovídá celkové šířce jádra. Faktor 2 reprezentuje dvojici operací násobení a sčítání (FMA – Fused Multiply-Add), které se provádějí pro každý prvek jádra nad každým vzorkem signálu.

Referenční hodnotou pro toto srovnání je nejmenší naměřená sada dat o velikosti 1,9 milionu prvků (7,6 MB), která vyžaduje provedení 2,0 G-Ops. V tabulce 6.4 jsou časy výpočtu větších úloh vyjádřeny jako násobky této základní hodnoty.

Tabulka 6.4: Faktory nárůstu času výpočtu vztažené k počtu operací při srovnání vlivu velikosti vstupních dat (Baseline = 2,0 G-Ops)

Implementace	10,9 G-Ops (Ideál 5,5×)	394 G-Ops (Ideál 197×)	1081 G-Ops (Ideál 541×)	Prům. odchylka
GPU_32BIT	5,0×	184,4×	500,1×	-7,8 %
GPU_NAIVE	5,2×	182,0×	501,9×	-6,7 %
CPU_PAR_MANUAL_VEC	5,5×	195,6×	538,3×	-0,5 %
CPU_PAR_AUTO_VEC	5,4×	192,0×	528,3×	-2,4 %
CPU_PAR_NO_VEC	5,5×	197,5×	543,5×	+0,2 %
CPU_PAR_NAIVE	5,5×	277,2×	771,9×	+29,9 %
CPU_SEQ_MANUAL_VEC	5,5×	197,9×	551,5×	+0,9 %
CPU_SEQ_AUTO_VEC	5,5×	199,2×	552,2×	+1,5 %
CPU_SEQ_APPLE	5,5×	198,2×	543,6×	+0,6 %
CPU_SEQ_NO_VEC	5,5×	197,1×	550,3×	+0,6 %
CPU_SEQ_NAIVE	7,3×	309,4×	855,3×	+51,9 %

Naměřená data odhalují tři zásadní trendy. Prvním je vysoká stabilita optimalizovaných CPU implementací, které si udržují nízkou průměrnou odchylku (pod 2,5 %) a jejichž výkon roste předvídatelně bez ohledu na to, zda data leží v cache nebo v operační paměti.

Druhým trendem je rozdílný nástup degradace u naivních verzí. Zatímco sekvenční verze výrazně zpomaluje již u úlohy s 10,9 G-Ops (cca 42 MB dat), paralelní implementace si v tomto bodě drží efektivitu a k propadu výkonu u ní dochází až u řádově větších datasetů. Tento posun naznačuje, že paralelizace oddaluje okamžik nasycení paměti.

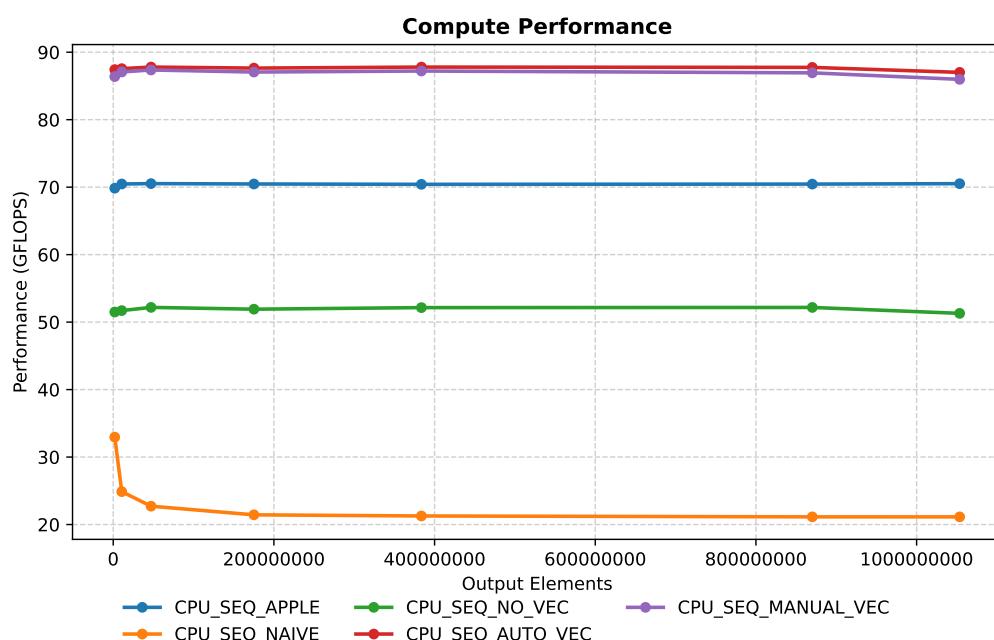
Třetím specifikem je „záporná“ odchylka u GPU, která potvrzuje, že grafické akcelerátory pracují efektivněji s velkými objemy dat, kdy se fixní režie spojená se startem výpočtu stává v celkovém čase zanedbatelnou.

6.3.2 Vliv paměťové hierarchie a System Level Cache

Detailní pohled na výpočetní propustnost (GFLOPS) odhaluje příčinu rozdílného chování naivních algoritmů. Srovnání grafů sekvenčních a paralelních implementací (Obrázek 6.5 a 6.6) ukazuje, jak zásadní roli hraje velikost dat vzhledem k dostupné cache paměti.

Kritický zlom nastává u sekvenční naivní implementace (CPU_SEQ_NAIVE) oka-mžitě při přechodu z referenční na úlohu s 10,9 G-Ops. Tato úloha odpovídá zpraco-vání cca **42 MB** dat. Protože výkonná jádra procesoru sdílejí **L2 cache o velikosti 24 MB**, objem dat přesahuje její fyzickou kapacitu. To vede k rapidnímu nárůstu výpadků v cache, kdy procesor musí pro chybějící data sahat do pomalejší System Level Cache (SLC) nebo přímo do RAM. Důsledkem je strmý propad výkonu z 33 na 21 GFLOPS, viditelný na obrázku 6.5.

Data GFLOPS (Sequential) - Fixed R=256

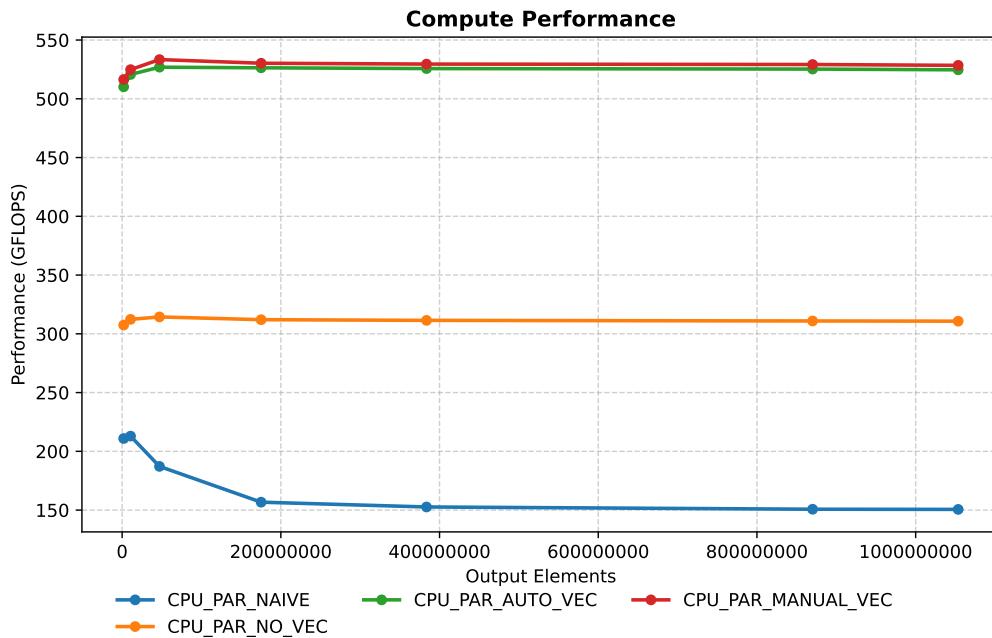


Obrázek 6.5: Srovnání škálování sekvenčních implementací v závislosti na velikosti vstupních dat.

U paralelní naivní verze je degradace pozvolnější (viz obrázek 6.6), ačkoliv součet požadavků osmi jader (42 MB) rovněž přesahuje kapacitu sdílené L2 cache. Rozdíl v rychlosti propadu výkonu je dán schopností paralelizace skrývat latenci paměti, kdy je čekání jednoho vlákna na data maskováno výpočtem ostatních. Tento pokles

se zastavuje u datasetu o velikosti 1,5 GB (394 G-Ops), kde se stabilizuje na hodnotě přibližně 150 GFLOPS.

Data GFLOPS (Parallel) - Fixed R=256



Obrázek 6.6: Srovnání škálování paralelních implementací v závislosti na velikosti vstupních dat.

6.4 Analýza škálování s velikostí poloměru jádra

V této části se analyzuje vliv poloměru konvolučního jádra (R) na efektivitu výpočtu. Změnou tohoto parametru totiž dochází ke změně počtu aritmetických operací nutných pro získání jedné výstupní hodnoty, zatímco množství vstupních dat, nad kterými se provádí konvoluce, zůstává stejné. Tím pádem se na rozdíl od předchozího případu, kdy se měnil objem dat a testovalo se využití cache a propustnost paměti, se testuje efektivita samotného výpočtu a manipulace s daty. Co se experimentálních měření týče, tak ta byla provedena nad datasetem PN01-1.edf (1659 MB) s poloměry konvolučního jádra v rozsahu $R = 2$ až 1024 , čímž se otěstovalo, jak jednotlivé algoritmy škálují svůj výkon s rostoucí aritmetickou intenzitou úlohy. Naměřené hodnoty výkonu pro vybrané poloměry jader shrnuje tabulka 6.5.

Tabulka 6.5: Závislost výpočetního výkonu [GFLOPS] na poloměru jádra R (Dataset PN01-1)

Implementace	R=2	R=16	R=32	R=64	R=256	R=1024
<i>GPU Implementace</i>						
GPU_32BIT	206,16	1198,32	1801,55	2302,98	2695,52	2752,72
GPU_NAIVE	223,38	704,36	733,68	754,71	768,66	765,97
<i>CPU Paralelní implementace</i>						
CPU_PAR_MANUAL_VEC	75,97	362,73	485,96	516,31	529,17	531,67
CPU_PAR_AUTO_VEC	68,51	366,01	491,14	501,40	525,29	530,09
CPU_PAR_NO_VEC	33,28	239,61	279,10	298,92	310,93	318,03
CPU_PAR_NAIVE	147,77	207,96	201,60	202,82	150,73	179,05
<i>CPU Sekvenční implementace</i>						
CPU_SEQ_MANUAL_VEC	25,93	76,66	81,81	84,03	86,94	87,68
CPU_SEQ_AUTO_VEC	23,35	80,82	83,38	85,11	87,75	88,00
CPU_SEQ_APPLE	47,25	88,14	84,87	78,86	70,45	67,73
CPU_SEQ_NO_VEC	5,88	41,99	46,44	49,00	52,17	52,81
CPU_SEQ_NAIVE	61,41	34,64	31,83	32,79	21,14	26,08

6.4.1 Charakteristika GPU škálování

Nejvýraznější nárůst výkonu je naměřený u implementace na grafickém akcelerátoru. Varianta GPU_32BIT startuje na hodnotě 206 GFLOPS ($R = 2$) a stoupá až k hranici 2700 GFLOPS ($R = 256$), kde se již výkon drží na konzistentní úrovni. Tento rozdíl je způsoben tím, že u malých poloměrů je objem výpočtů příliš malý a doba exekuce příliš krátká na to, aby se dokázala efektivně zakrýt latence přístupů do globální paměti a hardwarová režie spojená se spouštěním vláken. S rostoucím poloměrem se ale poměr aritmetických operací vůči načítání dat zvyšuje, čímž dochází k plnému vytížení výpočetních jednotek a potlačení vlivu paměťové latence.

6.4.2 Limity naivních algoritmů

U naivních implementací dosahuje výkon maxima při velmi malých velikostech konvolučního jádra a s jeho rostoucí velikostí postupně degraduje. U sekvenční varianty (CPU_SEQ_NAIVE) činí tento pokles přibližně 65 %, a to z maximálně dosažených 64,79 GFLOPS při $R = 4$ na minimální hodnotu 21 GFLOPS při $R = 128$. U paralelní varianty je propad mírnější, přibližně 53 %, konkrétně z 315,47 GFLOPS při $R = 8$ na 147,30 GFLOPS při $R = 128$.

Hlavní příčinou tohoto chování je vnitřní smyčka implementace, která iteruje přes prvky konvolučního jádra. Kompilátor se při optimalizaci snaží tuto smyčku

plně rozbalit, což je efektivní na malých jádrech. V těchto případech je naivní implementace dokonce rychlejší než optimalizované varianty. S rostoucí velikostí jádra se však pracovní data již nevejdou do 32 vektorových registrů, což vede k častějším přístupům do pomalejší paměti L1 a následnému poklesu výkonu.

U paralelní varianty se navíc projevuje režie spojená se správou vláken, což potvrzuje i výsledky optimalizované implementace. Tato režie má největší dopad na menších jádrech a s jejich rostoucí velikostí se její vliv postupně snižuje. Důsledkem je posun maximálního dosaženého výkonu paralelní varianty směrem k větším velikostem jader ve srovnání se sekvenční implementací.

6.4.3 Stabilizace optimalizovaných implementací

Zavedení blokového zpracování konvolučního jádra vedlo k stabilizaci výkonu u všech optimalizovaných verzí (sekvenčních i paralelních). Díky postupnému zpracování po blocích o velikosti $K_BATCH = 32$ je kompilátoru garantováno, že se potřebná data vejdou do hardwarových registrů. Tím se eliminuje jejich přetečení a následné propady výkonu, které jsou vidět u velkých jader při zpracování naivní implementací.

Paradoxně však u velmi malých jader ($R < 16$) optimalizované implementace zaostávají. Příčinou je právě struktura blokového zpracování, která vyžaduje existenci tzv. „dočišťovací“ smyčky pro zpracování zbytků jádra (části nedělitelné 32). U malých jader není splněna podmínka pro vstup do hlavní optimalizované větve (jelikož $2R + 1 < 32$) a celý výpočet tak spadá do této záložní sekce.

Problém spočívá v tom, že tato sekce využívá prohozené pořadí smyček (vnější iteruje přes prvky jádra, vnitřní přes data). To neumožňuje akumulovat průběžné výsledky konvoluce v registru (jako v naivní implementaci proměnnou `sum`), ale vynucuje přičítání mezivýsledků přímo do výstupního pole v paměti. To vede k nárůstu počtu paměťových operací z jedné (u naivní verze) na $2 \times (2R + 1)$ (čtení i zápis pro každý prvek jádra) na každý vzorek, což způsobuje dramatický propad výkonu.

Zdrojový kód 6.1: Struktura blokového zpracování a neefektivní dočišťovací smyčky

```

1 size_t k = 0;
2 for (; k + KBatch <= KernelSize; k += KBatch) {
3     // ... Efektivní výpočet ...
4 }
5
6 // Dočišťovací smyčka
7 for (; k < KernelSize; ++k) {
8     const float kv = kernelPtr[k];
9     for (size_t out = 0; out < actualChunkSize; ++out) {
10         o_chunk[out] += d_chunk[out + k] * kv;

```

```

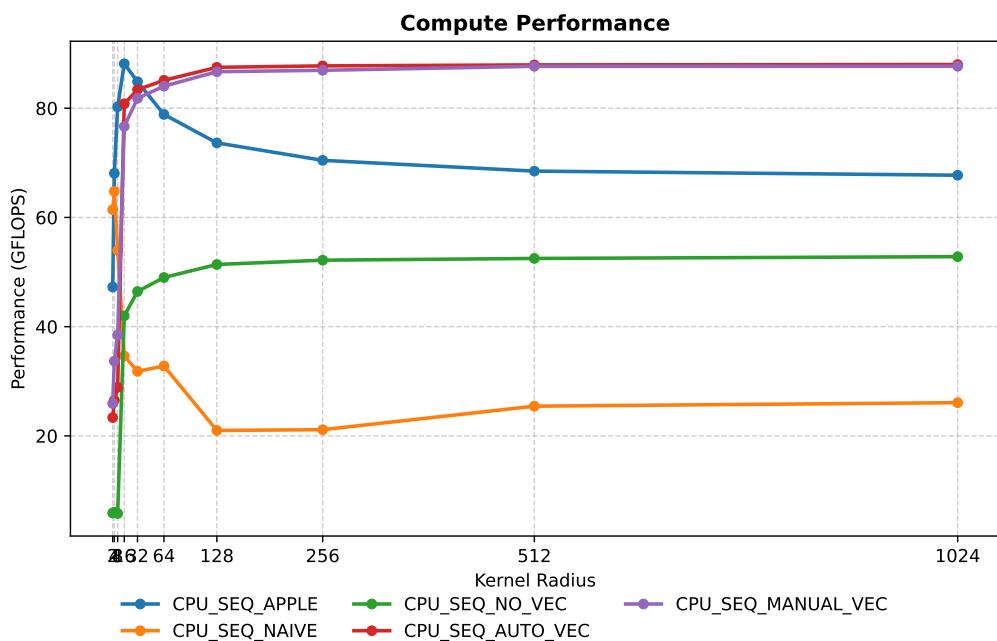
11     }
12 }
```

6.4.4 Srovnání se systémovou knihovnou Apple vDSP

Naměřená data u systémové funkce vDSP_conv naznačují, že tato funkce je primárně optimalizována pro zpracování krátkých konvolučních jader.

Své výkonné maximum 88,14 GFLOPS dosahuje tato funkce u jádra s poloměrem $R = 16$ a s rostoucí velikostí jádra je však patrný pozvolný úbytek efektivity. Výkon se postupně snižuje z maxima na stabilní úroveň okolo 70,45 GFLOPS (pro $R = 256$), což představuje pokles o přibližně 20 %.

Radius GFLOPS (Sequential): PN01-1.edf



Obrázek 6.7: Porovnání výkonu sekvenčních CPU implementací při změně poloměru konvolučního jádra.

6.5 Výkon na odlišných architekturách Apple Silicon

Pro ověření robustnosti implementace a potvrzení, že naměřené charakteristiky nejsou specifické pouze pro jednu konfiguraci hardwaru, byly experimenty repli-

kovány na druhém zařízení s odlišnou konfigurací. Cílem tohoto srovnání je analyzovat chování algoritmů v prostředí s odlišným poměrem výpočetního výkonu a paměťové propustnosti.

6.5.1 Specifikace testovacích zařízení

Jako srovnávací zařízení byl zvolen **Apple MacBook Air 13 (2024)** osazený čipem **Apple M4**. Rozdíly mezi tímto zařízením a referenčním Apple MacBook Pro (M1 Pro) jsou v tabulce 6.6.

Tabulka 6.6: Srovnání hardwarových parametrů testovacích zařízení

Parametr	MacBook Pro (M1 Pro)	MacBook Air (M4)
Generace čipu	Apple M1 (2021)	Apple M4 (2024)
ISA	ARMv8.5-A	ARMv9.4-A [Notc]
CPU Konfigurace	8 Jader (6P + 2E)	10 Jader (4P + 6E) [Wikb]
Frekvence P-Core	3,2 GHz	~4,4 GHz [Notc]
Frekvence E-Core	2,0 GHz	~2,9 GHz [Notc]
Vektorové registry	32 × 128-bit (NEON)	32 × 128-bit (NEON)
Šířka pipeline	8-wide	10-wide [Jon]
Teor. výkon (Single)	~103 GFLOPS	~141 GFLOPS*
Teor. výkon (Multi)	~700 GFLOPS	~841 GFLOPS*
Instr. Cache (P-Core)	192 KB	192 KB [Eve]
Data Cache (P-Core)	128 KB	128 KB [Eve]
L2 Cache (P-Cluster)	24 MB	16 MB [Eve]
System Level Cache	24 MB	8 MB [Low]
Paměť (RAM)	16 GB LPDDR5	16 GB LPDDR5X [Notc]
Propustnost paměti	200 GB/s	120 GB/s [Wikb]
GPU Jádra	14 jader	10 jader [Notc]
GPU ALU	1792 ALU	1280 ALU [Nan]
GPU Cache	32 kB (Threadgroup)	32 kB (Threadgroup) [Appd]
GPU Výkon (FP32)	~4,6 TFLOPS	~4,3 TFLOPS [CPU]
Chlazení	Aktivní	Pasivní

*Teoretický CPU výkonu pro M4 je dopočítán na základě frekvencí a architektury NEON.

6.5.2 Adaptace metodiky pro srovnávací měření

Pro zachování konzistence výsledků byla u srovnávacího zařízení aplikována metoda popsaná v sekci 6.1, avšak s dvěma nezbytnými modifikacemi. První úpravou je redukce počtu opakování ze čtyř sad po deseti iteracích na jednu sadu. Druhá

úprava zohledňuje absenci aktivního chlazení u modelu MacBook Air, kdy byly mezi jednotlivými měřeními zařazeny časové prodlevy pro vychladnutí čipu, aby se předešlo ovlivnění výkonu teplotou.

6.5.3 Výsledky srovnávacího měření

Validace proběhla na stejném datasetu PN01-1.edf (1659 MB) s identickým konvolučním jádrem o poloměru $R = 256$. Souhrnné výsledky pro obě platformy jsou uvedeny v tabulce 6.7.

Tabulka 6.7: Srovnání výkonu a času výpočtu mezi čipy M1 Pro a M4 ($R = 256$, Dataset PN01-1)

Implementace	Čas výpočtu [s]		Výkon [GFLOPS]		Změna výkonu (M4 vs M1)
	M1 Pro	M4	M1 Pro	M4	
<i>GPU Implementace</i>					
GPU_32BIT	0,331	0,346	2695,52	2581,10	- 4,2 %
GPU_NAIVE	1,162	1,196	768,66	746,52	- 2,9 %
<i>CPU Paralelní implementace</i>					
CPU_PAR_AUTO_VEC	1,700	1,378	525,29	647,97	+ 23,4 %
CPU_PAR_MANUAL_VEC	1,687	1,511	529,17	591,03	+ 11,7 %
CPU_PAR_NO_VEC	2,871	2,283	310,93	391,13	+ 25,8 %
CPU_PAR_NAIVE	5,923	3,136	150,73	284,68	+ 88,9 %
<i>CPU Sekvenční implementace</i>					
CPU_SEQ_AUTO_VEC	10,174	8,053	87,75	110,86	+ 26,3 %
CPU_SEQ_MANUAL_VEC	10,269	8,914	86,94	100,15	+ 15,2 %
CPU_SEQ_APPLE (vDSP)	12,672	9,652	70,45	92,50	+ 31,3 %
CPU_SEQ_NO_VEC	17,151	13,512	52,17	66,23	+ 27,0 %
CPU_SEQ_NAIVE	42,241	19,566	21,14	45,63	+ 115,8 %

6.5.3.1 Efektivita využití hardwaru

Pro základní validaci kvality implementace byla naměřená data konfrontována s teoretickými maximy obou čipů, což potvrdilo vysokou hardwarovou přenositelnost optimalizovaných algoritmů. V případě grafických akcelerátorů je efektivita využití FP32 jednotek téměř identická, kdy M1 Pro dosahuje 58,6 % a M4 60,0 % teoretického výkonu. Podobně konzistentní chování vykazuje i paralelní CPU zpracování, kde i přes výrazně odlišnou propustnost paměti dosahují oba čipy srovnatelného vytížení (75,6 % u M1 Pro oproti 77,0 % u M4). Efektivita mírně klesla pouze u jednovlákновého výkonu (78,7 % u M4 vs. 84,5 % u M1 Pro).

Naměřená data zároveň odhalila neefektivitu manuální optimalizace na novější architektuře. Zatímco u čipu M1 Pro byl výkonnostní rozdíl mezi automatickou a manuální vektorizací zanedbatelný (pod 1 %), u M4 tento rozdíl vzrostl na přibližně 10 % ve prospěch automatické vektorizace. Tento jev naznačuje, že fixní struktura manuálně vektorizovaného kódu pravděpodobně brání plnému využití širší pipeline (10-wide u M4 oproti 8-wide u M1 Pro).

Závěr

7

Cílem této práce byla implementace a optimalizace algoritmů pro filtraci EEG signálu pomocí lineární 1D konvoluce na platformě Apple Silicon. Výsledkem je sada algoritmů využívajících moderní standardy C++ a grafické API Metal, které pokrývají spektrum od referenčních naivních řešení až po vysoce optimalizované varianty využívající vektorizaci (NEON) a hardwarovou akceleraci (GPU).

Experimentální část prokázala, že navržené implementace dokáží efektivně využít dostupný výpočetní výkon hardwaru. Optimalizovaná varianta pro GPU dosáhla na čipu M1 Pro výkonu přibližně 2,7 TFLOPS, což odpovídá využití 60 % teoretického maxima čipu a překonává i měření provedená pomocí optimalizované knihovny PyTorch (MPS backend). V případě CPU implementace se podařilo dosáhnout výkonu přes 530 GFLOPS (cca 80 % teoretického maxima při využití všech jader). Významným výsledkem je skutečnost, že vlastní optimalizovaná sekvenční implementace překonala v testech výkonnosti i nativní systémovou funkci vDSP_conv z knihovny Apple Accelerate.

Testování potvrdilo robustnost řešení vůči změnám parametrů úlohy i hardwarové konfigurace:

- **Škálovatelnost:** Implementace vykazují stabilní výkon při zpracování různě velkých objemů dat i změně velikosti konvolučního jádra.
- **Přenositelnost:** Srovnávací měření na novějším čipu Apple M4 potvrdilo, že navržené optimalizace nejsou specifické pouze pro architekturu M1, ale efektivně škálují i na novějších generacích procesorů.
- **Validita:** Správnost výstupů byla ověřena křížovou validací všech implementací a srovnáním s referenční funkcí, přičemž maximální odchylka se pohybovala na úrovni statisticky nevýznamné zaokrouhlovací chyby.

I přes dosažené výsledky byly identifikovány oblasti pro budoucí rozvoj a optimalizaci. Prvním nedostatkem je pokles efektivity u konvolučních jader s malým poloměrem ($R < 16$), kde se negativně projevuje režie blokového zpracování a ne-

efektivita dočišťovacích smyček. Pro tyto případy by bylo vhodné implementovat specializovanou větev algoritmu.

Druhým a zásadním omezením pro nasazení do produkčního prostředí je práce s pamětí. Současná implementace načítá kompletní dataset do operační paměti RAM. Ačkoliv je toto řešení pro běžné EEG záznamy dostačující, u extrémně dlouhých měření naráží na fyzické limity zařízení. Budoucí vývoj by se měl zaměřit na implementaci streamovaného zpracování, které by umožnilo načítat a filtrovat data po částech, a tím zpracovávat soubory přesahující kapacitu dostupné operační paměti.

I přes uvedená omezení představuje výsledná aplikace vysoce efektivní nástroj, který již v současné podobě umožňuje zpracovávat rozsáhlá EEG data řádově rychleji než běžně dostupné prostředky.

Bibliografie

- [Appa] APPLE INC. *Accelerate Framework Documentation* [online]. Apple Developer Documentation. [cit. 2025-12-29]. Dostupné z: <https://developer.apple.com/documentation/accelerate>.
- [Appb] APPLE INC. *Addressing Architectural Differences in Your macOS Code* [online]. Apple Developer Documentation. [cit. 2025-12-28]. Dostupné z: <https://developer.apple.com/documentation/apple-silicon/addressing-architectural-differences-in-your-macos-code>.
- [Appc] APPLE INC. *DispatchQoS.QoSClass.userInteractive* [online]. Apple Developer Documentation. [cit. 2025-12-29]. Dostupné z: <https://developer.apple.com/documentation/dispatch/dispatchqos/qosclass/userinteractive>.
- [Appd] APPLE INC. *Metal Feature Set Tables* [online]. [B.r.]. [cit. 2026-01-01]. Dostupné z: <https://developer.apple.com/metal/Metal-Feature-Set-Tables.pdf>.
- [Appe] APPLE INC. *MTLDevice: newBufferWithBytesNoCopy:length:options:deallocator:* [online]. Apple Developer Documentation. [cit. 2025-12-28]. Dostupné z: <https://developer.apple.com/documentation/metal/mltdevice/1433382-newbufferwithbytesnocopy>.
- [Appf] APPLE INC. *vDSP_conv* [online]. Apple Developer Documentation. [cit. 2025-12-29]. Dostupné z: https://developer.apple.com/documentation/accelerate/vdsp_conv.
- [Arm24] ARM LIMITED. *Arm Architecture Reference Manual for A-profile architecture*. Arm Developer, 2024. Dostupné také z: <https://developer.arm.com/documentation/ddi0487/latest/>.
- [Arm] ARM LIMITED. *Arm Intrinsics Reference: NEON* [online]. Arm Developer. [cit. 2026-01-02]. Dostupné z: <https://developer.arm.com/architectures/instruction-sets/intrinsics/#f:@navigationhierarchiesimdisa%5BNeon%5D&q=>.

- [CPU] CPU-MONKEY. *Apple M4(10 Cores) GPU Benchmark* [online]. [cit. 2026-01-01]. Dostupné z: https://www.cpu-monkey.com/en/igpu-apple-m4_10_core.
- [Eve] EVERYMAC. *MacBook Pro 14-Inch "M4" 10 CPU/10 GPU Specs* [online]. [cit. 2026-01-01]. Dostupné z: https://everymac.com/systems/apple/macbook_pro/specs/macbook-pro-m4-10-core-cpu-10-core-gpu-14-2024-specs.html.
- [Joha] JOHNSON, Dougall. *Apple Firestorm Microarchitecture* [online]. [cit. 2025-12-27]. Dostupné z: <https://dougallj.github.io/applecpu/firestorm.html>.
- [Johb] JOHNSON, Dougall. *Firestorm SIMD and FP Instructions* [online]. [cit. 2025-12-29]. Dostupné z: <https://dougallj.github.io/applecpu/firestorm-simd.html>.
- [Jon] JON PEDDIE RESEARCH. *Apple introduces M4 Pro and M4 Max* [online]. [cit. 2026-01-01]. Dostupné z: <https://www.jonpeddie.com/news/apple-introduces-m4-pro-and-m4-max/>.
- [Kem] KEMP, Bob. *Full specification of EDF* [online]. EDFplus.info. [cit. 2025-12-28]. Dostupné z: <https://www.edfplus.info/specs/edf.html>.
- [Low] LOW END MAC. *M4 MacBook Air (Early 2025) - Technical Specifications* [online]. [cit. 2026-01-01]. Dostupné z: <https://lowendmac.com/2025/m4-macbook-air-early-2025/>.
- [Nan] NANOREVIEW. *Apple M4 10-Core GPU* [online]. [cit. 2026-01-01]. Dostupné z: <https://nanoreview.net/en/gpu/apple-m4-gpu>.
- [Nota] NOTEBOOKCHECK. *Apple M1 Pro 14-Core GPU - Benchmarks and Specs* [online]. NotebookCheck.net. [cit. 2025-12-27]. Dostupné z: <https://www.notebookcheck.net/Apple-M1-Pro-14-Core-GPU-Benchmarks-and-Specs.576651.0.html>.
- [Notb] NOTEBOOKCHECK. *Apple M1 Pro 8-Core Processor - Benchmarks and Specs* [online]. NotebookCheck.net. [cit. 2025-12-27]. Dostupné z: <https://www.notebookcheck.net/Apple-M1-Pro-8-Core-Processor-Benchmarks-and-Specs.576648.0.html>.
- [Notc] NOTEBOOKCHECK. *Apple M4 10-Core Processor - Benchmarks and Specs* [online]. [cit. 2026-01-01]. Dostupné z: <https://www.notebookcheck.net/Apple-M4-10-cores-Processor-Benchmarks-and-Specs.835975.0.html>.

- [Oak] OAKLEY, Howard. *Making the most of Apple silicon power: 3 Controls* [online]. The Eclectic Light Company. [cit. 2025-12-29]. Dostupné z: <https://eclecticlight.co/2022/10/13/making-the-most-of-apple-silicon-power-3-controls/>.
- [Son] SON, Choonho. *Apple Silicon, State-of-the-art ARM CPU* [online]. DEV Community. [cit. 2026-01-02]. Dostupné z: <https://dev.to/choonho/apple-silicon-state-of-the-art-arm-cpu-4131>.
- [Tan+21] TANG, Wensi et al. Omni-Scale CNNs: A simple and effective kernel size configuration for time series classification. In: *International Conference on Learning Representations (ICLR)*. 2021. Dostupné také z: <https://openreview.net/forum?id=PDYs7Z2XFGv>.
- [Wika] WIKIPEDIA CONTRIBUTORS. *Apple M1 — Wikipedia, The Free Encyclopedia* [online]. [cit. 2026-01-01]. Dostupné z: https://en.wikipedia.org/wiki/Apple_M1.
- [Wikb] WIKIPEDIA CONTRIBUTORS. *Apple M4* [online]. [cit. 2026-01-01]. Dostupné z: https://en.wikipedia.org/wiki/Apple_M4.

Seznam obrázků

3.1	Control Flow Diagram aplikace.	7
5.1	Porovnání výstupů módů CPU_PAR_AUTO_VEC a CPU_PAR_MANUAL_VEC.	27
6.1	Souhrnné porovnání celkového a výpočetního času všech implementací ($R = 256$, Dataset PN01-1).	31
6.2	Matice zrychlení (Speedup Matrix) pro dataset PN01-1 a jádro $R = 256$. Hodnoty udávají, kolikrát je metoda v řádku rychlejší než metoda ve sloupci.	32
6.3	Průběh výpočtu módu CPU_PAR_AUTO_VEC v čase (10 iterací, 4 běhy) pro $R = 256$ a dataset PN01-1.	33
6.4	Průběh výpočtu módu GPU_32BIT v čase (10 iterací, 4 běhy) pro $R = 256$ a dataset PN01-1.	34
6.5	Srovnání škálování sekvenčních implementací v závislosti na velikosti vstupních dat.	37
6.6	Srovnání škálování paralelních implementací v závislosti na velikosti vstupních dat.	38
6.7	Porovnání výkonu sekvenčních CPU implementací při změně poloměru konvolučního jádra.	41

Seznam tabulek

5.1	Tabulka porovnání podobnosti výsledků pro dataset PN01-1	25
6.1	Srovnání výkonnostních parametrů jednotlivých implementací ($R = 256$, Dataset PN01-1)	30
6.2	Metriky paralelizace vypočtené z časů výpočtů ($P = 8$)	34
6.3	Vývoj výpočetního výkonu [GFLOPS] v závislosti na velikosti vstupních dat ($R = 256$)	35
6.4	Faktory nárůstu času výpočtu vztažené k počtu operací při srovnání vlivu velikosti vstupních dat (Baseline = 2,0 G-Ops)	36
6.5	Závislost výpočetního výkonu [GFLOPS] na poloměru jádra R (Dataset PN01-1)	39
6.6	Srovnání hardwarových parametrů testovacích zařízení	42
6.7	Srovnání výkonu a času výpočtu mezi čipy M1 Pro a M4 ($R = 256$, Dataset PN01-1)	43

Seznam výpisů

2.1	Kompilační flagy C++	5
2.2	Kompilační flagy Metal	5
3.1	Jádro naivní konvoluce na CPU	8
3.2	Ukázka paralelizace pomocí GCD	9
3.3	Vytváření bufferů: Zero-Copy vs. standardní alokace	10
3.4	Kooperativní načtení dat do sdílené paměti	11
3.5	Naivní vektorizovaný výpočet na GPU	11
3.6	Struktura blokového zpracování	14
3.7	Vnitřní smyčka s unrollingem	15
3.8	Prokládání iterací na CPU	16
3.9	Manuální vektorizace pomocí vstavěných funkcí	17
3.10	Princip posuvného okna v hlavní smyčce	19
3.11	Implementace segmentovaného zpracování konvoluce	20
4.1	Ukázka běhu programu s konfigurací a výsledky benchmarku	22
5.1	Ukázka CSV reportu s detekovanými rozdíly	27
6.1	Struktura blokového zpracování a neefektivní dočišťovací smyčky	40

