FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Diplomová práce

Akcelerace vyhodnocování výrazů pomocí vektorových instrukčních sad

Studijní program: N2612 – Elektrotechnika a informatika

Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Victor Trnka**

Vedoucí práce: doc. Mgr. Jan Březina, Ph.D.

Liberec 2023

FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Zadání diplomové práce

Akcelerace vyhodnocování výrazů pomocí vektorových instrukcí

Jméno a příjmení:	**Bc. Victor Trnka**
Osobní číslo:	M20000174
Studijní program:	N2612 Elektrotechnika a informatika
Studijní obor:	Informační technologie
Zadávající katedra:	Ústav nových technologií a aplikované informatiky
Akademický rok:	2021/2022

Zásady pro vypracování:

1. Proveďte rešerši knihoven pro využití vektorových instrukcí.
2. Aplikujte knihovnu (např. Vector Class library) na základní operace parseru. Demonstrujte zrychlení výpočtu.
3. Ověřte přenositelnost pilotní aplikace na různé instrukční sady.
4. Rozšiřte použití knihovny na všechny operace BParseru.
5. Ověřte přenositelnost pro kompletní sadu operací parseru.
6. Aplikujte koncept maskování pro efektivní vyhodnocování podmíněných výrazů.

Rozsah grafických prací:	dle potřeby dokumentace
Rozsah pracovní zprávy:	40-50 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] Březina, J.: BParser, online: https://github.com/flow123d/bparser (5.10. 2020)
[2] Shibata, N., Petrogalli, F., 2020. SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions. IEEE Transactions on Parallel and Distributed Systems 31, 1316–1327. https://doi.org/10.1109/TPDS.2019.2960333
[3] Karpiński, P. and McDonald, J., 2017: A High-Performance Portable Abstract Interface for Explicit SIMD Vectorization, In: PMAM'17: Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores, 21–28.
[4] https://dl.acm.org/doi/10.1145/3026937.3026939

Vedoucí práce:	doc. Mgr. Jan Březina, Ph.D. Ústav nových technologií a aplikované informatiky
Datum zadání práce:	12. října 2021
Předpokládaný termín odevzdání:	22. května 2023

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

Ing. Josef Novák, Ph.D.
vedoucí ústavu

V Liberci dne 19. října 2021

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

9. 1. 2023 Bc. Victor Trnka

Poděkování

Rád bych poděkoval všem, kteří přispěli ke vzniku této diplomové práce. Především svému vedoucímu práce za skvělé vedení.

Akcelerace vyhodnocování výrazů pomocí vektorových instrukčních sad

Abstrakt

Tématem této práce je popis způsobu zrychlení a optimalizace vyhodnocování matematických výrazů v BParseru pomocí přidání podpory vektorových instrukcí procesoru. Doposud nebylo možno provádět dostatečně efektivní vyhodnocování matematických výrazů, neboť nebylo možno ovlivnit optimalizaci při překladu související s vektorovým zpracováním a bylo zde i omezení na podporované instrukční sady jednotlivých procesorů z hlediska přenositelnosti. Práce obsahuje informace o tom, co se rozumí pod pojmem parser s tím, že je zahrnut i stručný výčet existujících parserů. Dále je zmíněno, na jakém principu funguje vektorové vyhodnocování výrazů, a připojen je rovněž přehled vektorových instrukčních sad. Práce navazuje výčtem C++ knihoven, které podporují vektorové výpočty, a výběrem nejvhodnější knihovny Vector Class Library. Poté jsou zmíněny všechny nutné úpravy potřebné k vyřešení překážek a chyb, které souvisí s implementací. Práce dále zahrnuje testy přenositelnosti a rychlosti vyhodnocování výrazů. Z výsledků v podobě přehledných grafů z provedených testů rychlosti je zřejmé, že knihovna Vector Class Library přináší velmi dobrý posun v rychlosti a optimalizaci vyhodnocování výrazů v BParseru. Zároveň bylo dosaženo i maximální přenositelnosti programu mezi procesory x86-64 s rozličnými instrukčními sadami.

Klíčová slova: C++, SIMD, vektorové instrukční sady, vyhodnocování výrazů

Abstract

The topic of this paper is to describe how to speed up and optimize the evaluation of mathematical expressions in BParser by adding support for vector processor instructions. Up to now, it has not been possible to perform sufficiently efficient evaluation of mathematical expressions, as the translation optimizations associated with vector processing could not be influenced, and there were limitations on the supported instruction sets of each processor in terms of portability. This paper provides information on what is meant by the term parser, with a brief listing of existing parsers included. Furthermore, the principle on which vector expression evaluation works is mentioned, and a survey of vector instruction sets is also included. The paper continues by listing the C++ libraries that support vector computation and selecting the most appropriate Vector Class Library. Then, any necessary modifications needed to resolve implementation-related obstacles and errors are mentioned. The work also includes tests of portability and speed of expression evaluation. From the results in the form of clear graphs from the speed tests performed, it is clear that the Vector Class Library brings a very good improvement in speed and optimization of expression evaluation in BParser. At the same time, maximum program portability between x86-64 processors with different instruction sets was also achieved.

Keywords: C++, SIMD, vector instruction sets, expression evaluation

Obsah

Seznam zkratek 10

1 Úvod **13**

2 Vektorové vyhodnocování výrazů **15**

2.1 Vyhodnocování výrazů 15

2.2 Přehled parserů 16

2.2.1 BParser 16

2.2.2 Expression Toolkit 16

2.2.3 MathPresso 17

2.2.4 muParser 17

2.2.5 muParserSSE 17

2.3 Vektorové instrukční sady 18

2.3.1 MMX 19

2.3.2 SSE 19

2.3.3 SSE2 19

2.3.4 SSE3 19

2.3.5 SSSE3 20

2.3.6 SSE4 20

2.3.7 AVX 20

2.3.8 AVX2 20

2.3.9 AVX512 20

2.4 Knihovny pro vektorové instrukce v C++ 21

2.4.1 UME::SIMD 21

2.4.2 Vector Class v2 . . . 22
2.4.3 Vector Code . . . 23
2.4.4 Další knihovny . . . 23
2.4.5 Zvolená knihovna . . . 24

3 Úpravy kódu **26**
3.1 Konfigurační změny . . . 27
3.1.1 Přípravné změny . . . 27
3.1.2 Kompilace . . . 29
3.1.3 Ostatní změny . . . 29
3.2 Šablony a specializace . . . 30
3.2.1 Testovací výpis vektoru . . . 31
3.3 Konverze a reprezentace boolean hodnot . . . 32
3.3.1 Problém s datovým typem boolean . . . 33
3.3.2 Ukládání hodnot typu boolean . . . 33
3.3.3 Chybné výsledky v podmíněných výrazech . . . 35
3.4 Řešení drobných problémů s knihovnou . . . 35
3.4.1 Matematické funkce . . . 35
3.4.2 Chyba operace zaokrouhlování . . . 36
3.4.3 Operace modulo . . . 36

4 Testy **38**
4.1 Testy korektnosti a přenositelnosti . . . 38
4.1.1 Testy překladačů a operačních systémů . . . 39
4.1.2 Testovací hardware . . . 39
4.2 Testy rychlosti . . . 40
4.2.1 Prvotní test při zahájení vývoje . . . 40
4.2.2 Sada testovacích výrazů . . . 41
4.2.3 Skript na zpracování dat . . . 43
4.2.4 Definice prováděných testů . . . 44
4.2.5 Výsledné grafy . . . 45

5 Závěr 56

Použitá literatura 59

Použité obrázky 60

Přílohy 61

Seznam zkratek

API	Application Programming Interface
AVX	Advanced Vector Extensions
CPU	Central Processing Unit
DSP	Digital Signal Processing
GCC	GNU Compiler Collection
GPU	Graphics Processing Unit
HPC	High-performance computing
JIT	Just In-Time
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SLEEF	SIMD Library for Evaluating Elementary Functions
SSE	Streaming SIMD Extensions
SVML	Short Vector Math Library
VCL2	Vector Class Library version 2

Seznam obrázků

2.1 Schéma skalárního (SISD) a vektorového (SIMD) zpracování [9] . . . 18
2.2 Závislost registrů ZMM, YMM a XMM. [10] 21

4.1 Porovnání rychlosti BParseru OLD, C++ a BParser AVX2 na „Novém notebooku“, zdroj: vlastní 41
4.2 Graf porovnání poměru rychlostí v závislosti na velikosti vstupního vektoru u instrukční sady AVX512, zdroj: vlastní 45
4.3 Graf porovnání poměru rychlostí v závislosti na velikosti vstupního vektoru u instrukční sady AVX2, zdroj: vlastní 46
4.4 Graf porovnání poměru rychlostí v závislosti na velikosti vstupního vektoru u instrukční sady SSE, zdroj: vlastní 47
4.5 Graf porovnání poměru rychlosti C++ a BParseru AVX512, zdroj: vlastní 48
4.6 Graf porovnání poměru rychlosti C++ a BParseru AVX2, zdroj: vlastní 49
4.7 Graf porovnání poměru rychlosti C++ a BParseru SSE, zdroj: vlastní 50
4.8 Graf porovnání poměrů rychlostí v závislosti na použité sadě instrukcí, zdroj: vlastní 51
4.9 Graf porovnání poměrů rychlostí v závislosti na použité sadě instrukcí, zdroj: vlastní 52
4.10 Graf porovnání poměrů rychlostí v závislosti na použité sadě instrukcí, zdroj: vlastní 53
4.11 Graf porovnání poměru rychlosti BParser OLD a BParser AVX2, zdroj: vlastní 54

4.12 Graf porovnání poměru rychlosti BParser OLD a BParser AVX2 na aritmetických operacích, zdroj: vlastní 55

1 Úvod

Při kompilaci programu pro vyhodnocení výrazu nebývá přesně znám výraz, který bude zpracováván. Kompilátor tedy nemůže provést optimalizaci výpočtu již ve fázi překladu. Vyhodnocení výrazu se děje až při běhu programu takzvaným syntaktickým analyzátorem, kterým je v případě této práce BParser. Ten by měl být pro tuto činnost optimalizován, aby při mnohokrát opakovaném vyhodnocování nad různými daty byl efektivní. To, že nemohl kompilátor provést optimalizaci výpočtu při překladu, není pro rychlost vyhodnocení výrazu tolik podstatné, neboť větší dopad na rychlost má mnohonásobné vyhodnocování stále stejného výrazu.

Jisté omezení BParseru doposud spočívalo v tom, že nepodporoval všechny dostupné vektorové instrukční sady. Specifickou vlastností BParseru je použití jednoho datového typu, což klade nároky při určitých operacích. Navíc žádné operace v parseru nebyly definované takovým způsobem, aby kompilátor při překladu vektorových instrukcí opravdu využil. Proto ke zrychlení vyhodnocení výrazů řízeně využijeme vektorových instrukčních sad obsažených v současných procesorech. Abychom toho dosáhli bez detailní znalosti jednotlivých instrukčních sad, vybereme vhodnou knihovnu, která instrukce z těchto sad efektivně používá. Knihovna navíc musí mít definované stejné operace s podporou vektorizace, jako má BParser.

Cílem rovněž je, aby se program dal přeložit na jakémkoliv počítači, který má procesor architektury x86-64. Dále by se měla provést maximální optimalizace a mělo zaručit použití nejvyšší možné instrukční sady. Zároveň požadujeme, aby ve zkompilovaném kódu byly podporovány i další instrukční sady a program z nich dynamicky zvolil tu, která je optimální pro použitý procesor architektury x86-64.

Následující text v kapitole 2 poskytuje teoretický úvod do vyhodnocování mate-

matických výrazů, podává přehled parserů, vektorových instrukčních sad a knihovny pro vektorové instrukce v C++. V kapitole číslo 3 jsou popsány a zdůvodněny úpravy, které byly provedeny v kódu BParseru. Konečně kapitola 4 se zabývá testováním a porovnáním rychlosti vyhodnocování matematických výrazů v různých podmínkách.

2 Vektorové vyhodnocování výrazů

V této kapitole se podíváme, jak probíhá vyhodnocování matematických výrazů. Dále si ukážeme princip jak takové výrazy vyhodnocovat pomocí vektorů. Poté se seznámíme se specializovanými instrukčními sadami, které podporují vektorové výpočty. V poslední části si uvedeme volně dostupné knihovny, které dokáží tento přístup aplikovat.

2.1 Vyhodnocování výrazů

Výrazem rozumíme nějaké seskupení hodnot, konstant, operátorů, funkcí, atd. Pro člověka je vcelku jednoduché pochopit, a pokud zná několik pravidel, tak i vyhodnotit předložený matematický výraz. Pro počítač to tak snadné není a musí se postupně udělat několik kroků, které ho dovedou od zadaného textu k výsledné hodnotě.

Syntaktický analyzátor je také nazýván parser. Nicméně často se výraz parser používá pro vyjádření celku, který zahrnuje více než jen syntaktický analyzátor. Pro nás bude parser označovat celek, který jednak udělá syntaktickou analýzu, ale následně provede i samotné vyhodnocení daného výrazu. Parser tedy plní úlohu syntaktického analyzátoru a vyhodnocovače výrazu.

Postup zpracování a vyhodnocení výrazu je následující: Výraz se převede na strukturu nazvanou syntaktický strom, tzv. „abstract syntax tree“, na základě gramatiky [1]. Výsledný syntaktický strom se následně převádí na vektorové operace nad „array“ a tyto operace se pak převádí na acyklický graf bytecode operací. Bytecode je přenositelný strojový kód, který je nezávislý na architektuře procesoru. Pro tento graf pak parser hledá vhodné „topologické uspořádání“ neboli kompilaci, která

mimo jiné zahrnuje alokaci dočasných proměnných. Bytecode musí být interpretován, což je v zásadě pomalejší než předem kompilovaný kód, ale při opakovaném provádění stejné operace nad různými daty začne být vektorový přístup (SIMD) efektivnější, a to tím více, čím větší množství dat se zpracovává.

2.2 Přehled parserů

Zde si uvedeme několik volně dostupných parserů napsaných v jazyce C++, které jsou k dispozici.

2.2.1 BParser

BParser [2] je součástí projektu Flow123d [3], který se vyvíjí na Technické Univerzitě v Liberci. Slouží k rozboru a vyhodnocení zadaného matematického výrazu se zaměřením na vysokorychlostní výpočty. Je napsán v jazyce C++. Syntaxe výrazů se používá stejná jako v knihovně Numpy pro jazyk Python. Výraz je vždy přeložen do bytecode. Aby se zrychlilo načítání z paměti, je zde implementován blokový alokátor paměti, který vytváří takzvané arény, které alokují části parseru a zpracovávaného výrazu v rámci předem alokované souvislé části paměti. Parser podporuje výpočty s vektory a s maticemi pomocí přístupu SIMD. Pro zrychlení výpočtů využívá vektorových instrukčních sad moderních CPU. Ke spuštění je potřeba mít kompilátor GCC 4.6 nebo jeho novější verzi a navíc knihovnu Boost alespoň ve verzi 1.58.0, která díky komponentě Spirit [4] použité pro tvorbu parseru z gramatiky pomocí složitého mechanizmu šablon generuje velmi rychlý parser, i když překlad trvá relativně dlouho.

2.2.2 Expression Toolkit

Tvůrce ExprTk [5] uvádí, že knihovna disponuje jednoduchým použitím s efektivními výpočty. Má zabudovanou podporu řetězců a několik operací, které s nimi dokáží snadno pracovat. Ze zmíněných knihoven nabízí nejvíce funkcí včetně numerické integrace a derivace. Knihovna s určitým omezením také nabízí vytvoření vlastních

funkcí, které pak dále můžeme použít ve vlastních výrazech. Dokáže taktéž pracovat s C vektory, avšak explicitně nevyužívá vektorového vyhodnocení výrazů. V principu je možné knihovnu přinutit pracovat tak, aby vektorové vyhodnocení používala, ale vzhledem k tomu, že na takovéto používání nebyla knihovna od počátku navržena, je obtížné dosáhnout korektní funkčnosti. Z důvodu složité úpravy stávajícího kódu a také kvůli absenci několika požadovaných vlastností začal vývoj BParseru.

2.2.3 MathPresso

Zadaný matematický výraz se v MathPressu [6] převádí do strojového kódu, jde tedy o JIT kompilátor. Tím se minimalizuje režie oproti parserům využívajícím přechodnou reprezentaci (např. bytecode). Pokud CPU podporuje instrukční sadu SSE4.1, knihovna ji využije pro zvýšení rychlosti. Avšak instrukční sady AVX a novější nejsou podporovány. MathPresso ve skutečnosti nedokáže počítat s vektory, ale jednotlivé vektorové instrukce používá jen ke zrychlení výpočtů.

2.2.4 muParser

MuParser [7] je vysoce výkonná C++ knihovna umožňující vyhodnocovat matematické výrazy. Tyto výrazy se převedou do bytecode, který je poté interpretován. Dokáže dosahovat vysokých rychlostí předvypočítáním konstantních částí výrazu a pomocí možnosti paralelizovat výpočet rozdělením na více vláken za podpory OpenMP API. Knihovna je navržena tak, aby mohla být spuštěna na co největším počtu zařízení, a k jejímu spuštění stačí mít vybraný C++ kompilátor.

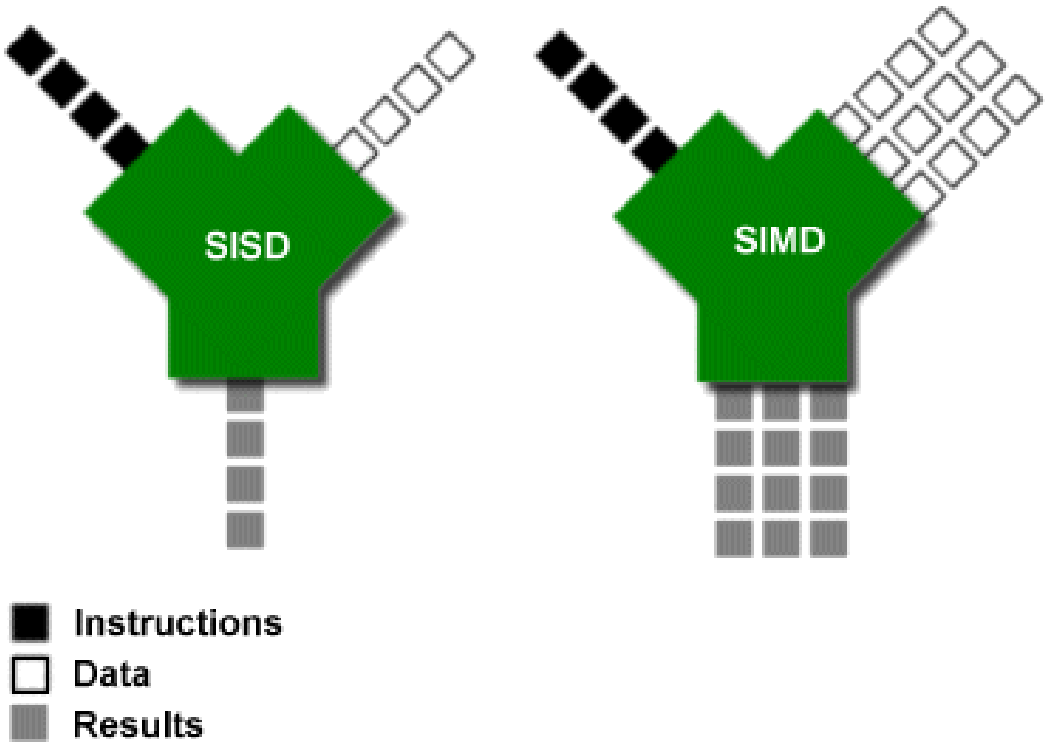
2.2.5 muParserSSE

Tato knihovna je založena na muParser s doplněním o JIT funkcionalitu, konkrétně s využitím AsmJit knihovny. Jak je z názvu muParserSSE [8] patrné, knihovna je schopna využívat pouze 128bitové instrukce z instrukčních sad SSE. V rámci velmi urychleného vývoje není ani implementovaná celá sada funkcí z výchozí knihovny muParser. Ačkoliv využívá vektorového vyhodnocování výrazů, tak nedosahuje

uspokojující rychlosti pro použití na náš problém. Přesnost výpočtu je také omezena kvůli použití datového typu float namísto původního double.

2.3 Vektorové instrukční sady

Vektorové vyhodnocování výrazů znamená, že pokud se v zadaném výrazu nějaká operace několikrát opakuje s různými daty, tak namísto opakovaného provádění té samé operace se jednotlivá data přesunou do vektoru jako jeho prvky a poté se pomocí vektorové instrukce vypočtou všechny hodnoty najednou. Avšak ve skutečném procesoru není možné dělat výpočty pro libovolnou délku vektoru. Vektor se musí buď rozdělit na několik menších, pokud je příliš velký (svou velikostí přesahuje šířku registru), nebo se doplní na odpovídající délku. Maximální velikost vektoru závisí na velikosti registru a množina dostupných vektorových operací závisí na podporované instrukční sadě daného procesoru.



Obrázek 2.1: Schéma skalárního (SISD) a vektorového (SIMD) zpracování [9]

Veškeré instrukční sady jsou závislé na typu a výrobci procesoru. Neustále se vyvíjí a vylepšují. Stejně tomu je tak i u vektorových instrukčních sad. V tomto případě se jedná především o rozšíření počtu operací, které lze vykonávat vektorově. Dále se jedná o rozšíření co do počtu prvků vektoru.

Do přehledu vezměme v potaz vývoj instrukčních sad pro řadu procesorů typu x86, který započal v roce 1997.

2.3.1 MMX

Vznikla ke zrychlení multimediálních aplikací a mělo se díky ní dosáhnout téměř dvojnásobného zrychlení. Používané registry jsou široké 64 bitů. Kvůli problémům s překladači jazyka C se kód musel psát v assembleru. Pro správný běh programu se musel zvolit buď celočíselný mód, nebo mód s datovým typem float.

2.3.2 SSE

V roce 1999 přišla firma Intel s nástupcem rozšíření MMX v podobě Streaming SIMD Extensions, který přidal 8 nových registrů a rozšířil je na 128 bitů. Dále se přidalo 70 nových instrukcí pro práci s float.

2.3.3 SSE2

S rokem 2001 přišla novější verze SSE, která umožnila použití datového typu double.

2.3.4 SSE3

Instrukční sada byla představena v roce 2004. Oproti SSE2 přidává několik matematických instrukcí spojených s prací v DSP. Dále bylo umožněno sčítat a násobit dvě čísla uložená ve stejném registru, což do té doby nebylo možné.

2.3.5 SSSE3

Vylepšení SSSE3 s sebou přineslo pouze 16 nových instrukcí. Například společnost AMD využívá tohoto rozšíření v procesorových řadách Cat, Heavy Equipment a Zen. Intel využívá tohoto vylepšení ve svých procesorových řadách Xeon, Core 2, Core iX.

2.3.6 SSE4

K představení došlo v roce 2006. Přidává 54 nových instrukcí. Instrukční sada se dělí na dvě části SSE4.1 a SSE4.2. SSE4.1 obsahuje 47 instrukcí a zbylých 7 instrukcí je obsaženo v SSE4.2. První část přidává instrukce pro přesuny a uspořádání dat v registru, kdežto druhá část je zaměřena na operace se znaky a s řetězci. AMD vytvořilo navíc SSE4a pro své procesory, které pak Intel vůbec nepoužívá.

2.3.7 AVX

S AVX přichází firmy Intel a AMD v roce 2008, ale první procesory se s ní objevily až v roce 2011. Největší změnou je zdvojnásobení šířky registrů oproti SSE na 256 bitů. Dolních 128 bitů jednotlivých registrů stále můžeme používat v režimu SSE. Díky tomu se zajistila zpětná kompatibilita s SSE. Neméně důležitým vylepšením je použití formátu instrukcí se třemi operandy.

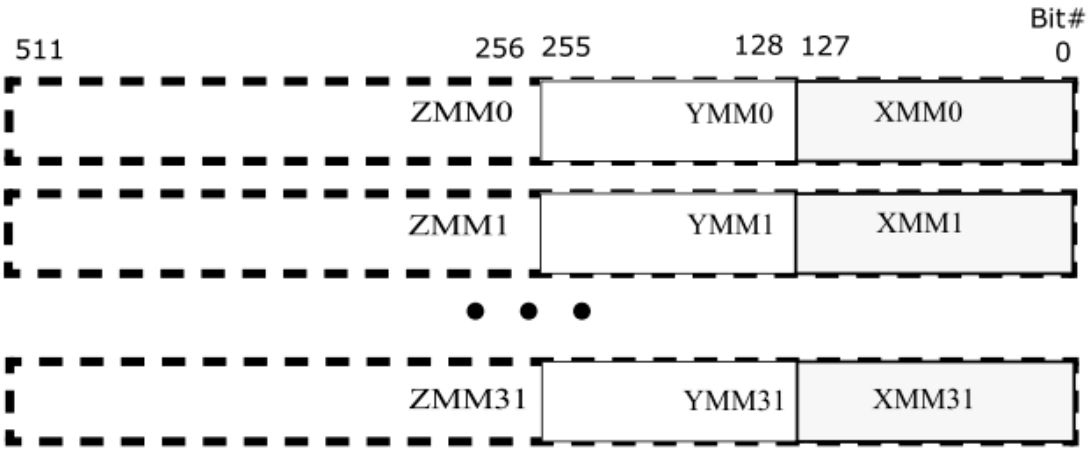
2.3.8 AVX2

Rozšiřuje AVX v oblasti instrukcí s operacemi na datovém typu integer pro šířku 256 bitů. Přidává bitovou manipulaci, násobení se třemi operandy a také vektorové posuny.

2.3.9 AVX512

Došlo k dalšímu rozšíření registru na 512 bitů a vzrostl počet registrů z 16 na 32 (ZMM0 až ZMM31). Všechny registry můžeme adresovat také jako 256bitové YMM (AVX) nebo 128bitové XMM (SSE). Instrukční sada AVX512 má mnoho

odlišných variant, které přidávají různé instrukce, většinou jsou určené pro nějaké více specializované použití.



Obrázek 2.2: Závislost registrů ZMM, YMM a XMM. [10]

2.4 Knihovny pro vektorové instrukce v C++

Pro náš účel potřebujeme knihovnu instrukcí s těmito vlastnostmi: Jde o knihovnu s otevřeným kódem spravovaným pod online verzovacím systémem (například GitHub) s vhodnou otevřenou licencí. Knihovna by měla poskytovat dostatečné spektrum instrukčních sad (historicky starší i moderní), měla by umožnit přiměřeně snadné použití v kódu (např. srozumitelné názvy datových typů, metod, funkcí, apod.). Knihovna musí umožňovat použití námi zvolených kompilátorů (GCC a CLang). Musí podporovat vektorizaci operací z BParseru. Důležité je, aby tyto operace byly podporovány všemi vektorovými instrukčními sadami knihovny. Preferujeme využití moderních šablon, a tedy kompatibilitu minimálně s verzí C++11.

2.4.1 UME::SIMD

Knihovna vyžaduje použití standardu C++11. UME::SIMD [11] v dnešní době plně podporuje 4 instrukční sady. Tyto sady jsou: AVX, AVX2, AVX512 a IMCI (speci-

ální 512bitová instrukční sada použitá ve vektorové výpočetní jednotce v koprocesorech Intel Xeon Phi). Instrukční sady SSE podporovány nejsou a ani se v budoucnu neuvažuje o jejich implementaci do knihovny. Ke zkompilování je možné použít kompilátorů CLang, GCC, Intel C++ nebo MS Visual C++. Je možné využít kolem 250 operací na přibližně 60 typech SIMD vektorů, které jsou dány pomocí šablon. Délka jednoho vektoru musí vždy být mocnina 2 a jeho maximální šířka může být 1024 bitů. To v praxi znamená maximálně 16krát double nebo 128krát int8. Výhodné pro přehlednou práci je možnost různého pojmenování tříd vektorů. Například pro vektor, který obsahuje čtyři floaty, je možnost zápisu následující:

```
1 SIMDVec<float, 4> <=> SIMDVec_f<float, 4> <=> SIMD4_32f
```

Pro zlepšení práce a ke zkrácení výsledného kódu je k tomu vytvořena navíc doplňující knihovna UME::Vector [12], která umožňuje vytvářet vektory libovolné délky.

2.4.2 Vector Class v2

Knihovna Vector Class version 2 [13] vyžaduje minimální standard jazyka C++17. Podporované instrukční sady jsou SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, AVX512F/VL/BW/DQ/ER, dále pak AVX512VBMI/VBMI2, XOP, FMA3, FMA4. Ke kompilaci je možné využití kompilátorů pro C++ jako CLang, GCC, Intel C++ nebo MS Visual C++, nicméně nejlepších výsledků se dosáhne s GCC a CLang. VCL2 nemá podporu na platformu ARM, jinak běží na 32 i 64bitových procesorech s operačními systémy Windows, Linux a MacOS. Knihovna definuje i vektory typu boolean v různých velikostech dle typu instrukčních sad. O detekci nejvyšší využitelné instrukční sady se stará samotná knihovna. V případech, kdy by se vektor nevešel do registru, rozdělí se automaticky na více vektorů. Jelikož kompilace probíhá bez znalosti cílového CPU, tak musí být vygenerován kód pro více variant a ten správný se vybírá až za běhu. Také je možné omezit si maximální šířku vektoru. Například pokud bychom měli procesor podporující AVX512, ale chtěli bychom využít jen maximálně 128 bitů, pak bychom to udělali takto:

```
1 #define MAX_VECTOR_SIZE 128
```

2.4.3 Vector Code

Knihovna Vc [14] ke svému fungování vyžaduje standard C++11. Podpora instrukčních sad je následující: SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2. Ve vývoji je i AVX512, NEON, NVIDIA GPU a CUDA. Kompilátory se mohou použít CLang, GCC, Intel C++ nebo MS Visual C++. Dále je potřeba mít CMake verze 3.0 a novější. Maximální šířka vektoru není uvedena, ale dokud nebude zavedena plná podpora pro AVX512, pak bychom měli počítat s šířkou 256 bitů. V této době je vývoj upnut na následující knihovnu STD - SIMD, která společně s touto knihovnou bude tvořit vylepšenou verzi Vc 2.

2.4.4 Další knihovny

- STD - SIMD: Tato experimentální knihovna std-simd [15] tvoří základ pro Vc 2. Vývoj byl ukončen, protože její funkce se vložila do knihovny v GCC verze 11. Funguje na platformě x86_64, aarch64 a arm.

- QuickVec: Knihovna QuickVec [16] se neobejde bez vývojového prostředí Microsoft Visual Studio. Potřebuje standard C++11 a cílová platforma je pouze x64_86. Instrukční sady, které podporuje tato knihovna, jsou AVX, SSE, SSE2, SSE4.1, SSE4.2. Knihovna byla vyvíjena pouze jako studentský projekt.

- The Tuesday C++ Vector Math and SIMD Library: Ke spuštění knihovny Tuesday [17] je potřeba mít minimálně C++14 a k tomu vhodný kompilátor GCC nebo CLang. Knihovna navíc vyžaduje použití Microsoft Visual Studio 2015 a novější. Zaměření zde není na jednorozměrné vektory, ale spíše na vícerozměrné vektory. Jsou zde rozděleny dimenze vektorů jako separátní typy. Maximální dimenze je 4.

2.4.5 Zvolená knihovna

Z uvedeného seznamu knihoven jsme vybrali Vector Class library verze 2 popsanou v kapitole 2.4.2. Splňuje především požadavek na podporu instrukčních sad, kdy je možné využít kromě AVX a AVX2 jak SSE, tak i AVX512. Stejně tak splňuje naše požadavky na použitelné kompilátory, na otevřenou licenci, na otevřený kód a na online verzovací systém. Podporuje všechny operace mimo operaci modulo. Operace modulo se v jedné z posledních verzí knihovny objevila, ale není ještě plně vektorizovaná. Pozitivem knihovny je její snadné použití v kódu, kdy si ji stačí stáhnout z GitHub repozitáře a vložit na požadované místo v projektu. V samotném kódu projektu se pak pouze vloží odkaz na hlavičkový soubor celé knihovny. Knihovna dále umí sama detekovat nejvyšší podporovanou instrukční sadu procesoru. Určitou nevýhodou knihovny je potřeba využití standardu C++17, kdy BParseru stačila pouze C++14. To klade omezení na použitelné verze kompilátorů.

Jako poměrně vhodná se jevila i knihovna UME::SIMD z kapitoly 2.4.1, ta však nepodporuje instrukční sady SSE. Její předností je naopak uživatelsky přívětivá syntaxe vektorových datových typů.

Knihovna Vector Code popsaná v kapitole 2.4.3 v současnosti nepodporuje instrukční sadu AVX512. Nadějným se jevil vývoj pro grafické procesory, byl však zastaven.

Níže demonstrujeme jednoduché použití knihovny VCL2 (řádek 1) a ukázku některých funkcí a datových typů.

```
1  #include "vectorclass.h"
2
3  int main() {
4      // Init vectors of 4
5      Vec4f f1(1.2, 2.4, 3.6, 4.8);
6      Vec4d d3(1.2, 2.4, 3.6, 4.8);
7      Vec4f res_f1;
8      Vec4d res_d3;
9
10     // Compute
```

```
11      res_f1 = sqrt(f1);
12      res_d3 = pow(d3, 3);
13
14      // Print results
15      printVector(res_f1, "res_f1:");
16      printVector(res_d3, "res_d3:");
17
18      return 0;
19  }
```

3 Úpravy kódu

První část kapitoly se věnuje přípravám pro vložení knihovny VCL2 do kódu a změnám souvisejících s nastavením pro úspěšnou kompilaci. Druhá část popisuje použití šablon a jejích specializací. Další část je věnována specifické oblasti reprezentace boolean hodnot. V závěru jsou řešeny drobné problémy, které se při implementaci vyskytly.

V knihovně Vector Class Library se nachází vektory, které se liší podle použitého datového typu, který v sobě ukládají. Ty se pak dále rozlišují podle počtu obsažených prvků. V BParseru pracujeme s datovým typem double. My pak využijeme vektory se dvěma, čtyřmi nebo osmi doubly. Ty se značí `Vec2d`, `Vec4d` a `Vec8d`. Kvůli některým operacím potřebujeme využít také datový typ boolean. K tomu má knihovna příslušné vektory `Vec2db`, `Vec4db` a `Vec8db`. V BParseru místo používání pevného vektoru o čtyřech prvcích definovaného jako `double4` je potřeba dynamicky využívat vektory z knihovny VCL2. To by mělo umožnit, aby se stejný zdrojový kód dal zkompilovat na různých počítačích s rozdílnými podporovanými instrukčními sadami. V BParseru je celkem 39 operací. Jedná se o aritmetické operace, matematické funkce, binární operace a vyhodnocení podmíněných výrazů. Vybraná knihovna VCL2 má v sobě implementovaných operací více, než jich parser potřebuje.

Prováděné úpravy kódu měly dvojí charakter. Jednak šlo o úpravy předpokládané (například vložení knihovny VCL2, použití šablon na vektorové datové typy z knihovny, aj.) a na druhé straně o úpravy vynucené zjištěnými chybami (například špatná interpretace hodnoty `True`, chybějící operace modulo, apod.).

3.1 Konfigurační změny

Pro větší přehlednost rozdělme provedené změny do tří skupin: změny, které jsou nutné pro vložení knihovny VCL2 do projektu, dále změny úzce související s kompilací a nakonec další nezbytné úpravy kódu.

3.1.1 Přípravné změny

V prvním kroku vytvořme v parseru adresář `third_party/VCL_v2` a do něj vložme staženou knihovnu z GitHub. Cestu na tento adresář je potřeba doplnit do souboru `CMakeLists.txt` jako další pravidlo, kde má kompilátor hledat hlavičkové soubory pro direktivu `#include`.

Dále vytvořme hlavičkový soubor `VCL_v2_math.hh`, protože budeme potřebovat vložit z knihovny matematické operace a funkce. Tento soubor vložme do adresáře `include` spolu s dalšími hlavičkovými soubory BParseru. Jelikož jsou nastavena striktní pravidla pro překlad, kdy i upozornění je považováno za chybu, a vzhledem k tomu, že nepotřebujeme využít celou knihovnu, která obsahuje velké množství nastavených parametrů a proměnných, musíme dočasně vypnout příznaky pro kompilátor GCC `-Wunused-but-set-parameter` a `-Wunused-but-set-variable`, abychom zajistili bezchybný překlad.

Ukázka ze souboru `VCL_v2_math.hh`, ve kterém se vloží tři matematické hlavičkové soubory z knihovny VCL2:

```
1  #ifdef __GNUC__
2  #if (__GNUC__ > 6)
3
4  // save diagnostic state
5  #pragma GCC diagnostic push
6
7  // turn off the specific warning.
8  #pragma GCC diagnostic ignored "-Wunused-but-set-parameter"
9  #pragma GCC diagnostic ignored "-Wunused-but-set-variable"
10
```

```
11 #endif
12 #endif
13
14 #include "vectormath_hyp.h"
15 #include "vectormath_exp.h"
16 #include "vectormath_trig.h"
17
18 #ifdef __GNUC__
19 #if (__GNUC__ > 6)
20
21 // turn the warnings back on
22 #pragma GCC diagnostic pop
23
24 #endif
25 #endif
```

Abychom v parseru mohli vytvořit procesor, který bude pracovat s příslušným datovým typem, nejprve musíme vytvořit nadřazenou strukturu, ze které se procesor bude dědit. Tuto strukturu jsme pojmenovali `ProcessorBase`. Obsahuje virtuální metody `run`, `set_subset` a virtuální destruktor. Konstruktor této struktury se využije k uložení ukazatele na arénu procesoru, kterou je potřeba znát při destrukci konkrétního procesoru. V rámci struktury je deklarována statická metoda `create_processor`, která rozhoduje o vytvoření správného typu procesoru na základě velikosti `simd_size`.

Všechny výskyty struktury `Processor` jsme nahradili nadřazenou strukturou `ProcessorBase`.

Dále jsme vytvořili funkci `get_simd_size`, která vrací datový typ `uint`. Funkce zjistí, jakou maximální instrukční sadu konkrétní CPU podporuje, a vrátí hodnotu, která odpovídá počtu double čísel, jež se vejdou do příslušných registrů. Například pro instrukční sadu AVX512 to znamená, že se vrátí číslo 8, protože 8 * 64 bitů je 512 bitů.

Podstatné změny se odehrály v souboru `processor.hh`. Z důvodu přehlednosti a kvůli oddělení deklarací a definic jsme ho rozdělili na tři soubory: `processor.hh`,

`eval_impl.hh` a `create_processor.hh`. Velkou část původního souboru jsme přesunuli do nového souboru `eval_impl.hh`, kde se nachází struktury vektorů, operací a vyhodnocování operací. V souboru `processor.hh` zůstala definice metody `create_processor_` a struktury `ProcessorBase` se zděděnou strukturou `Processor`. Uvnitř procesoru se nastavují vektory a jejich hodnoty na základě typu uzlu.

3.1.2 Kompilace

V adresáři `include` vytvořme čtyři `.cc` soubory, v nichž nadefinujeme speciální makra pro knihovnu VCL2 tak, aby se vždy použila maximální optimalizace pro jednotlivé instrukční sady podporující danou šířku vektoru. Tím chceme dosáhnout toho, aby se kód přeložil do jednoho spustitelného souboru čtyřikrát, pokaždé s jinými instrukcemi v závislosti na instrukční sadě, tedy bez vektorizace, s instrukční sadou SSE4, s AVX2, s AVX512. Tyto soubory je třeba přidat jako cíle k překladu do souboru `CMakeLists.txt`, kde se nachází pravidla pro překlad celého programu. V něm také nastavme obecný příznak optimalizace `-O3`, který značí nejvyšší míru optimalizace. Dále nastavme příznak použití standardu C++17. Pro jednotlivé `.cc` soubory zde nastavme pravidla pro cílovou instrukční sadu a optimalizaci výpočtů. V každém `.cc` souboru je dále obsažena funkce na vytvoření procesoru s konkrétním datovým typem z knihovny, který svojí šířkou odpovídá šířce registrů instrukční sady.

3.1.3 Ostatní změny

Dalším hlavičkovým souborem, který se dočkal důležitých změn, je `parser.hh`. Jde o přidání proměnné `simd_size` související s vytvářením správného typu procesoru pomocí rozhodovací metody `create_processor`. Ta se nachází v hlavičkovém souboru `create_processor.hh` a její funkcí je buď předat přednastavenou hodnotu `simd_size`, nebo zavolat druhou metodu ze souboru, a to `get_simd_size`. Obě tyto metody jsme sem přesunuli z hlavičkového souboru `processor.hh`.

V hlavičkovém souboru `arrena_alloc.hh` jsme přidali do konstruktoru struktury `ArenaAlloc` podmínku, aby nebyl ukazatel na začátek arény nulový. Pokud by se tak stalo, program ohlásí chybu a nepřiřadí hodnotu `nullptr` do ukazatele, který se pak ve struktuře používá v různých metodách.

V souboru `config.hh` jsme vymazali definici `SIMD_OPERATION_SIZE`, která již nemá význam.

3.2 Šablony a specializace

Šablony slouží k použití stejné struktury nebo metody s různými datovými typy, aniž bychom ji museli pro každý typ definovat znovu a opisovat tak stejný kód několikrát.

Aby byl splněn požadavek, že parser bude spustitelný na jakémkoliv počítači, tak kromě používání vektorových instrukcí z vektorových instrukčních sad je potřeba udělat i variantu nevektorovou. Tedy kromě využití vektorových datových typů z knihovny využijeme počítání pomocí typu double. Toho docílíme takzvanou specializací struktur nebo metod. Pokud máme šablonu (template) na nějaké metodě nebo struktuře, můžeme nadefinovat i speciální chování pro vybraný datový typ. Ukázka jednoduché specializace metody `f` pro double:

```
1  template <class T>
2  void f(T x) {
3      std::cout << "Templated f: " << x << std::endl;
4  }
5
6  template<>
7  void f<double>(double x) {
8      std::cout << "Specialized template for double type: "
9      << x << endl;
10 }
```

Tyto specializace provedeme v každém případě, kdy je rozdílný kód pro vektory z knihovny a pro double. Například některé operace jsou definované v knihovně VCL2 jinak než ve standardní matematické knihovně. Dále se specializace využívá v maskování boolean hodnot na double hodnoty. Rozřazující struktura `EvalImpl`,

která se dělí podle počtu operandů, má také specializaci pro double kvůli tomu, že knihovna potřebuje využít vlastní funkce načítání a ukládání. Další specializace se využije při vytváření jednotlivých variant procesoru podle velikosti `simd_size`.

Šablonu jsme použili na strukturu `Processor`, na metodu `create_processor_`, na struktury `Vec`, `Workspace` a `EvalImpl`, ve které použijeme specializace kvůli nutnosti použití funkcí pro načítání operandů a kvůli ukládání výsledku zpět do paměti u vektorů z knihovny VCL2. Dále pak využijeme šablon pro metodu `eval`, která vyhodnocuje jednotlivé operace, a na funkce k získání hodnot `True` a `False`. Šablony se specializacemi se také používají při aplikaci konceptu maskování zmíněného v kapitole 3.3.2.

3.2.1 Testovací výpis vektoru

Abychom zjistili, zda funguje správně šablona na operace, potřebovali jsme vektory vypsat do konzole. Knihovna žádnou takovou funkci nemá vytvořenou, a tak jsme ji vytvořili v souboru `test_tools.hh`. Vypisuje vektor s jednotlivými prvky a kvůli rozdílným vektorovým datovým typům vektorů je obalena šablonou a specializovaná pro výpis jednoprvkové hodnoty double. Abychom vyzkoušeli i jiné věci z knihovny, vytvořili jsme navíc testovací soubor s názvem `test_VCL_v2.cc`, ve kterém je možné libovolně zkoušet pokusy s vektory a zjistit, co se s nimi děje. Ukázka kódu této funkce:

```
1  template<typename VecType>
2  static void print_VCL_vector(const VecType & v,
3                               const char * prefix);
4
5  template<typename VecType>
6  void print_VCL_vector(const VecType & v,
7                        const char * prefix) {
8      bool first = true;
9      std::cout << prefix << "(";
10     for(int i = 0; i < VecType::size(); i++)
11     {
12         if (first)
```

```
13          {
14              std::cout << v[i];
15              first = false;
16              continue;
17          }
18
19          std::cout << " ; " << v[i];
20      }
21      std::cout << ")" << std::endl;
22  }
23  template<>
24  void print_VCL_vector<double>(const double & v,
25                                const char * prefix) {
26      std::cout << prefix << "(" << v << ")" << std::endl;
27  }
```

3.3 Konverze a reprezentace boolean hodnot

BParser je celý vytvořen na základě jednoho datového typu. Dříve to byl vytvořený datový typ `double4`. Nyní bychom kvůli některým operacím z knihovny potřebovali využít datový typ s boolean hodnotami, protože jako své parametry jiný typ nepřijmou.

V souboru `scalar_node.hh` připravme novou strukturu `ConstantBoolNode`, která vytvoří vrchol acyklického grafu výrazu speciálně pro boolean konstanty a funkci přebírá od klasických konstant. O tuto možnost rozšíříme výčtovou proměnou `ResultStorage`, která ukládá, jakého typu je výsledný vrchol. S tím je též spojena metoda na vytváření takového vrcholu. V souboru `expression_dag.hh` musíme přidat podmínku, aby se správně posouval index výsledku podle počtu boolean konstant. Další související změna je v souboru `array.hh`, kde opět po vzoru konstant vytvoříme stejnou metodu ve struktuře `Array` pro boolean konstanty. Nakonec tuto metodu využijeme pro získání polí s hodnotou `True` a `False`.

3.3.1 Problém s datovým typem boolean

Knihovna používá dva typy boolean vektorů. Takzvané `broad` a `compact`. Pokud není k dispozici instrukční sada AVX512, tak se používá `broad` varianta, kdy vektor obsahuje stejný počet prvků jako příslušný datový typ. Například pro 128 bitový vektor typu Vec2db by vypadal pro hodnotu `False` jako (0, 0). Pokud by byl ve variantě `compact`, pak by obsahoval pouze jednu hodnotu (0). Tyto dva typy jsou mezi sebou nekompatibilní, tedy nejde mezi nimi převádět. Navíc je těžké předem přesně určit, jestli se použije, samozřejmě při splnění podmínky instrukční sady, varianta `broad` nebo `compact`. Z tohoto důvodu chceme přinutit knihovnu, aby použila vždy `broad` variantu a bylo to tak stejné pro všechny velikosti vektorů. V knihovně navíc neexistuje přímá konverze mezi datovými typy obsahující double a boolean. Konverze funguje automaticky v knihovně mezi boolean a integer vektory. A dále funguje konverze mezi různými datovými typy s `broad` typem boolean vektoru, pokud mají odpovídající počet bitů. Abychom dokázali správně přiřazovat vektory typů double a boolean mezi sebou, musí mít vektory navzájem si odpovídající velikosti. Ale abychom určili například k datovému typu `Vec8d` datový typ `Vec8db`, musíme to explicitně definovat. K tomu využijeme konverzních funkcí.

3.3.2 Ukládání hodnot typu boolean

Pro ukládání hodnot typu boolean uděláme konverzní funkce, které budou znát odpovídající si datové typy. V celém parseru se bude každý datový typ tvářit a ukládat jako typ double. A jen když funkce bude potřebovat mít vstupní parametr vektor typu boolean, tak pomocí konverzní funkce z double na boolean jí z typu double vytvoříme příslušný vektor typu boolean. Pokud funkce bude vracet hodnotu jako boolean vektor, tak ji pomocí konverzní funkce z boolean na double převedeme na vektor double hodnot a s tím již zbytek parseru umí pracovat. Princip konverze spočívá ve výsledku v tom, že ukazatel do paměti na příslušné hodnoty přetypujeme z jednoho datového typu na druhý. Uvedeme zde části kódu z převodu double hodnot na boolean. Metoda má název `as_bool` a využívá prvně struktury `d_to_b`,

která je šablonovaná vybraným typem double vektoru. Pro každý typ je pak zavedena specializace a uvnitř je definovaný příslušný typ boolean. Například pro `Vec8d` vypadá specializace následovně:

```
1 template<>
2 struct d_to_b<Vec8d> {
3     typedef Vec8db bool_type;
4 };
```

Další částí konverzní funkce je struktura nazvaná `union`. Ta umožňuje v sobě uchovat oba druhy datových typů pro sobě si odpovídající vektory. Vytvoří se hodnota stejného datového typu, jako je vstup, v tomto příkladě double, tím bude i tento `union` šablonovaný a opět bude mít své specializace. Odpovídající datový typ v příslušném `union` bude použit jako maska. Příklad:

```
1 template<>
2 union d_to_b_mask<Vec8d> {
3         Vec8d  value;
4         Vec8db  mask;
5 };
```

Když jsou připravené obě tyto struktury a specializované pro všechny varianty double datových typů, které jsou přípustné, tak to umožní vzít double hodnotu a vrátit ji jako boolean hodnotu pod správným datovým typem.

```
1 template<typename double_type>
2 inline typename
3     d_to_b<double_type>::bool_type as_bool(double_type in)
4 {
5     d_to_b_mask<double_type> x = {in};
6     return x.mask;
7 }
```

Pro výše uvedený příklad by se za `double_type` dosadil `Vec8d` a vrácený typ by byl `Vec8db`. Pro opačnou konverzi by se použila inverzní metoda `as_double`, která je definovaná obdobným způsobem jako `as_bool`.

Standardní boolean (bool) hodnotu interpretujeme jako datový typ `int64_t`, ale u porovnávacích operací se s datovým typem double vyskytuje výsledek jako

bool, proto přetížíme metodu `as_double`, aby byla takto specializovaná jen jednou a omezilo se co nejvíc specializování jednotlivých operací.

3.3.3 Chybné výsledky v podmíněných výrazech

Při testování správnosti výsledků se nám vyskytl problém u operace podmíněného výrazu. Pokud byla podmínka nepravdivá a reprezentace `False` hodnot byla nula, tak výpočet a rozhodnutí bylo vždy správné. Chyba se projevila v případě, kdy podmínka vyšla pravdivá. Po prozkoumání funkčnosti vektorů z knihovny jsme zjistili, že hodnota `True` je reprezentovaná hodnotou `-1` a ne `1`. Je to tedy vlastnost knihovny VCL2, nikoliv chyba. Tudíž to není jako v jazyce C, kdy nula je reprezentace nepravdy a kterékoliv jiné číslo znamená pravdu. Proto striktně zaveďme konstanty pro `True` hodnotu `-1` a pro `False` hodnotu `0`.

3.4 Řešení drobných problémů s knihovnou

3.4.1 Matematické funkce

Knihovna umožňuje výběr při výpočtech standardních matematických funkcí: buď se funkce budou vkládat vždy jako kus kódu v místech, kde se má operace počítat, nebo se využije některá z externích knihoven. Knihovna VCL2 neumožňuje obě varianty kombinovat. Z několika možností jsme prozkoumali knihovnu od firmy Intel s názvem `Short Vector Math Library` a knihovnu pro vektorové výpočty matematických funkcí nazvanou SLEEF [18]. Pro projekt jsme nakonec vybrali možnost přímého vkládání kódu oproti použití externí knihovny, protože nechceme mít v projektu závislosti na dalších knihovnách.

3.4.2 Chyba operace zaokrouhlování

Jediná chyba v knihovně souvisela s problémem u operace zaokrouhlování [19] spojeným s verzemi překladače GCC 7.xx a 8.xx [20]. Vyřešena byla tak, že se zjistí, jestli je překladač v jedné z verzí, která tuto chybu obsahuje, a pak speciálně jen pro tuto funkci se nastaví ignorování příznaku `-Wattributes`. Tím se rozšíří podpora verzí překladače GCC od verze 7 a výše. Ukázka vyřešení problému:

```
1  // preventing gcc 7.x.x and 8.x.x bug
2  #ifdef __GNUC__
3  #if (__GNUC__ > 6 && __GNUC__ < 9)
4  #pragma GCC diagnostic push
5  #pragma GCC diagnostic ignored "-Wattributes"
6  #endif
7  #endif
8
9  // function with warning
10 static inline Vec2d round(Vec2d const a);
11
12 #ifdef __GNUC__
13 #if (__GNUC__ > 6 && __GNUC__ < 9)
14 #pragma GCC diagnostic pop
15 #endif
16 #endif
```

3.4.3 Operace modulo

Operace modulo (%) na začátku vývoje této práce v použité knihovně chyběla. Vývojář knihovny poskytl informaci, že má v plánu tuto operaci někdy v budoucnu implementovat. Přibližně po roce se funkce `fmodulo` v knihovně objevila. Jenže její parametry obsahují pouze vektory typu float nebo double jako dělenec, což je pro použití v parseru sice vhodné, ale dělitel není vektor, ale pouze skalár. Tato skutečnost je již problém, protože vstupními parametry do každé operace v parseru je v případě použití knihovny vektor doublů a nelze předpokládat, že by všechny

prvky vektoru, který je dělitelem, měly stejnou hodnotu. Takže se musíme uchýlit k náhradnímu výpočtu místo použití jedné funkce.

Operace modulo lze vypočítat pomocí vzorce. Pokud je `a` dělenec a `b` dělitel, pak je výpočet následující:

$$a \; mod \; b = a - b * \lfloor \frac{a}{b} \rfloor$$

Výpočet tímto způsobem je plně vektorizovaný, a tím pádem lépe splňuje aktuální používání datových typů v parseru.

4 Testy

K otestování funkčnosti řešení s implementovanou knihovnou využijme dvou testů. Jednak testy korektnosti v různých prostředích a s různými překladači, tedy test parseru a v něm jednoduché testy operací na malých vektorech. A jednak testy efektivity knihovny, tedy test rychlosti výpočtů jednotlivých matematických výrazů.

Předpoklady k provedení testů jsou následující: architektura procesoru počítače je `x86-64`, operační systém Linux, nainstalované knihovny Boost ve verzi alespoň 1.58 a CMake ve verzi minimálně 3.10.

4.1 Testy korektnosti a přenositelnosti

Knihovna VCL2 umožňuje využít možnost překladu a spuštění na starších počítačích, kde chybí podpora instrukční sady AVX2. Lze vzít úplně stejný zdrojový kód a teoreticky ho přeložit na jakémkoliv počítači splňujícím výše uvedené předpoklady. Přeložený spustitelný soubor pak lze spustit na kterémkoliv počítači, pokud má procesor architektury x86-64.

K otestování korektnosti použijme vytvořený test procesoru omezený na počítače využívající vektorové instrukční sady AVX2. Tím ověříme, zda jednotlivé operace a funkce dají správný výsledek. Poté přistupme k testování parseru, čímž ověříme přenositelnost a korektnost pro vybrané matematické výrazy, tedy že operace a funkce se šablonou a specializacemi fungují podle předpokladu a vracejí očekávané výsledky.

Ještě vytvořme testovací strukturu `ParserTest`, která se dědí ze struktury `Parser`, ale navíc jí lze ručně nastavit parametr `simd_size`. To nám umožňuje

vyzkoušet použití různých vektorových instrukčních sad, nejen nejvyšší sadu automaticky detekovanou.

4.1.1 Testy překladačů a operačních systémů

Nezávislost řešení na zvoleném překladači byla testována na platformě GitHub Actions [21]. Zde definujeme kombinace operačního systému a překladače a tím simulujeme testovací prostředí na nespecifikovaném hardware, na kterém pak spouštíme vybrané testy. Ty se mohou spouštět i automaticky podle zvolené akce. Například v repozitáři BParseru se pouští nový test při každém publikování změn.

Ke spuštění je potřeba kompilátor na jazyk C++. Knihovna by měla fungovat s překladači Intel C++, CLang, GCC a Microsoft Visual C++. Dle autora knihovny a jeho testů [22] je nejlepších výsledků dosahováno s kompilátory CLang a GCC, proto od testování zbylých kompilátorů upustíme. Kompilátor GCC musí být verze 7 a výše. Verze překladače CLang musí být minimálně 5.

Provedli jsme testy s překladači CLang a GCC na operačních systémech Linux. V Actions jsme ověřili fungování s překladačem GCC ve verzích 7, 8, 9 s operačním systémem Ubuntu 18.04, verze 10 a 11 s operačním systémem Ubuntu 20.04, u kompilátoru CLang ve verzích 5, 6, 7 a 8 s operačním systémem Ubuntu 18.04. Ve všech případech testy prokázaly plnou funkčnost.

Podrobnější dokumentace o provedených testech je dostupná v GitHub repozitáři BParseru [2] v záložce Actions.

4.1.2 Testovací hardware

Protože GitHub Actions nepodporuje volbu testovacího hardware, provedli jsme navíc testy na třech různých počítačích.

1. „Starý počítač“. Stolní počítač s dvoujádrovým procesorem AMD Athlon(tm) 64 X2 Dual Core Processor 5000+ CPU @ 2.6 GHz. Operační systém: Debian 12 (kódové označení bookworm). Kompilátor: GCC 10.4.0.

2. „Nový notebook“. Čtyřjádrový procesor Intel(R) Core(TM) i5-5300U CPU @ 2.30 GHz. Operační systém: Ubuntu 22.04. Kompilátor: GCC 11.3.0.

3. „Cluster Charon“. Čtyřicetijádrový procesor Intel(R) Xeon(R) Silver 4114 CPU @ 2.20 GHz. Operační systém: Debian 11 (kódové označení bullseye). Kompilátor: GCC 10.2.1.

Veškeré zbylé informace o procesorech jsou obsaženy v příloze.

„Starý počítač“ podporuje maximálně vektorovou instrukční sadu SSE2. „Nový notebook“ podporuje maximálně vektorovou instrukční sadu AVX2. „Cluster Charon“ podporuje maximálně vektorovou instrukční sadu AVX512.

Spuštění téhož testu na všech třech počítačích ověří přenositelnost: na kterémkoliv stroji lze provést kompilaci a vzniklý spustitelný soubor pak spustit na stoji jiném.

Tento test proběhl úspěšně. Testovací spustitelný soubor je v příloze spolu se snímky obrazovky z výše uvedených počítačů, na kterých byla možnost provádět testy.

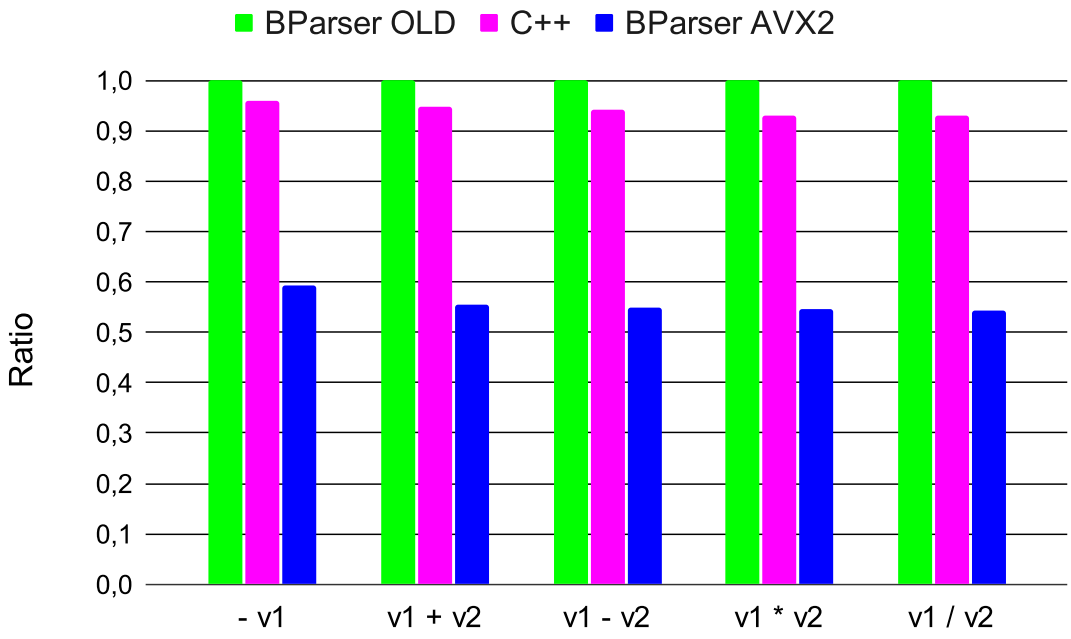
4.2 Testy rychlosti

Účel začlenění knihovny VCL2 do BParseru byl, kromě zajištění přenositelnosti, zrychlit vyhodnocování matematických výrazů. Už testy při zahájení vývoje naznačovaly, že ke zrychlení dochází.

4.2.1 Prvotní test při zahájení vývoje

Vybrali jsme 5 operací, které používají stejný zápis jak v původní implementaci, tak v knihovně VCL2. První operací je použití operátoru „−“. Dalšími operacemi jsou sčítání, odčítání, násobení a dělení. Všechny tyto operace fungují stejně pro double

i pro vektory. V testu každá z vybraných operací proběhla s milionem opakování a měřil se čas celkového zpracování pro každou operaci. Tento proces se provedl třikrát: pro BParser bez knihovny (označme BParser OLD), pro C++ a pro BParser s knihovnou VCL2 a s využitím vektorové instrukční sady AVX2 (označme BParser AVX2). Testy proběhly na stroji „Nový notebook“. Významné zrychlení operací při použití knihovny VCL2 tak potvrdilo její správný výběr.



Obrázek 4.1: Porovnání rychlosti BParseru OLD, C++ a BParser AVX2 na „Novém notebooku“, zdroj: vlastní

4.2.2 Sada testovacích výrazů

Po dokončení vývoje a ověření přenositelnosti můžeme provést testy na porovnání zrychlení. Abychom mohli provádět testy, musíme si stanovit testovací sadu výrazů. Ta bude obsahovat především všechny operace parseru a dále složené výrazy s více operacemi. Testovacích výrazů bude celkem 46. Z důvodu snazšího vyhodnocení a přehledného zobrazení je rozdělme do čtyřech kategorií:

- První kategorii pojmenujme `Arithmetic`. Obsahuje pět základních operací, jak byly definované dříve, doplněné o operaci modulo.
- Druhá kategorie se jmenuje `Boolean`. Obsahuje logické a porovnávací operace, ve kterých se v nějaké variantě používají datové typy boolean.
- Třetí kategorie obsahuje matematické funkce nebo výběrové funkce. Nazývá se `Function` a obsahuje nejvíc výrazů.
- Poslední kategorií jsou výrazy složené. Nazvěme tuto kategorii `Composed`.

Tyto kategorie budou použity jako hlavní rozdělení pro většinu testů, aby výsledné grafy byly přehledné. Přesné rozložení výrazů do kategorií je vidět v tabulce 4.1.

Arithmetic	Boolean	Function	Composed
-v1	(v1 == v2)	abs(v1)	v3 if (v1 == v2) else v4
v1 + v2	v1 != v2	sqrt(v1)	cv1 + v1 + v2 - v3
v1 - v2	v1 <v2	exp(v1)	cs1 - v1 + v2 * v3
v1 * v2	v1 <= v2	log(v1)	cs1 * v1 / v2
v1 / v2	v1 >v2	log10(v1)	cv1 / v1 * v2 / v3
v1 % v2	v1 >= v2	sin(v1)	v1 + v2 + v3 + v4
	not (v1 == v2)	sinh(v1)	3 * v1 + cs1 * v2 + v3 + 2.5 * v4
	v1 or v2	asin(v1)	
	v1 and v2	cos(v1)	
		cosh(v1)	
		acos(v1)	
		tan(v1)	
		tanh(v1)	
		atan(v1)	
		ceil(v1)	
		floor(v1)	
		isnan(v1)	
		isinf(v1)	
		sgn(v1)	
		atan2(v1, v2)	
		v1 ** v2	
		maximum(v1, v2)	
		minimum(v1, v2)	

Tabulka 4.1: Rozdělení testovacích operací do jednotlivých kategorií, zdroj: vlastní

4.2.3 Skript na zpracování dat

Kvůli rychlé replikaci testů a možnosti snadno spustit test s více opakováními a rovnou výsledky vyhodnotit a zobrazit v grafech byl vytvořen skript v jazyce Python.

4.2.4 Definice prováděných testů

Testy dle níže uvedených definic provedeme pro tři velikosti vstupního vektoru: 64, 256 a 1024 hodnot. Počet opakování testu je volen tak, aby celkový počet operací zůstal konstantní. Tedy vektor velikosti 64 se provede 1 600 000krát, pro velikost 256 bude opakování 400 000krát a pro velikost 1024 jen 100 000krát.

Každý test se všemi velikostmi vektorů provedeme pětkrát a pro další vyhodnocení použijeme medián z naměřených časů. Hodnotu pro celou kategorii testovaných výrazů pak stanovíme jako aritmetický průměr z hodnot mediánů jednotlivých výrazů.

BParser s knihovnou VCL2

- BParser AVX512: BParser s knihovnou VCL2 s použitou vektorovou instrukční sadou AVX512 na „Clusteru Charon“.
- BParser AVX2: BParser s knihovnou VCL2 s použitou vektorovou instrukční sadou AVX2 na „Novém notebooku“.
- BParser SSE: BParser s knihovnou VCL2 s použitou vektorovou instrukční sadou SSE na „Starém počítači“.

BParser bez knihovny

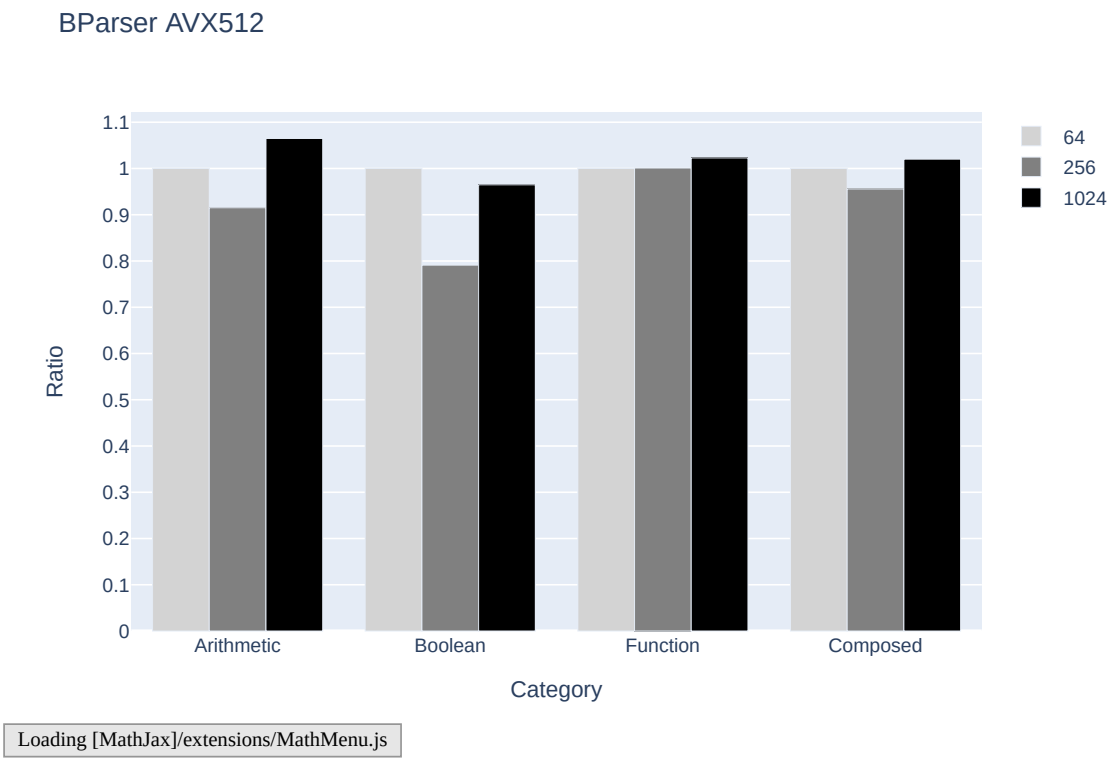
- BParser OLD: BParser bez knihovny, commit ze dne 27.07.2022 uživatele jbrezmorf, s použitou vektorovou instrukční sadou AVX2 na „Novém notebooku“.

C++

- C++: Čistý výpočet v jazyce C++ spuštěný na stejném počítači jako porovnávaný test BParseru.

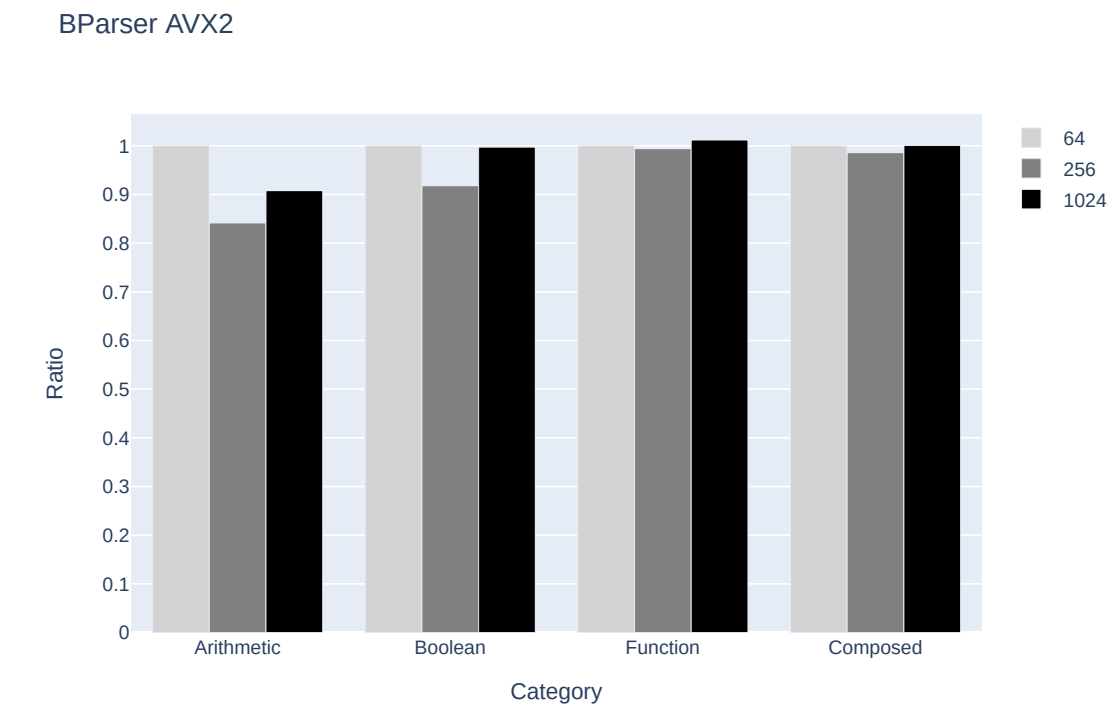
4.2.5 Výsledné grafy

Nejprve proveďme test pro různé velikosti vstupního vektoru zpracovávaného jednou instrukční sadou. Očekáváme, že průměrný čas zpracování nebude na velikosti vektoru závislý, tedy že pro vektory velikosti 64, 256 a 1024 budou zjištěné časy přibližně stejné.



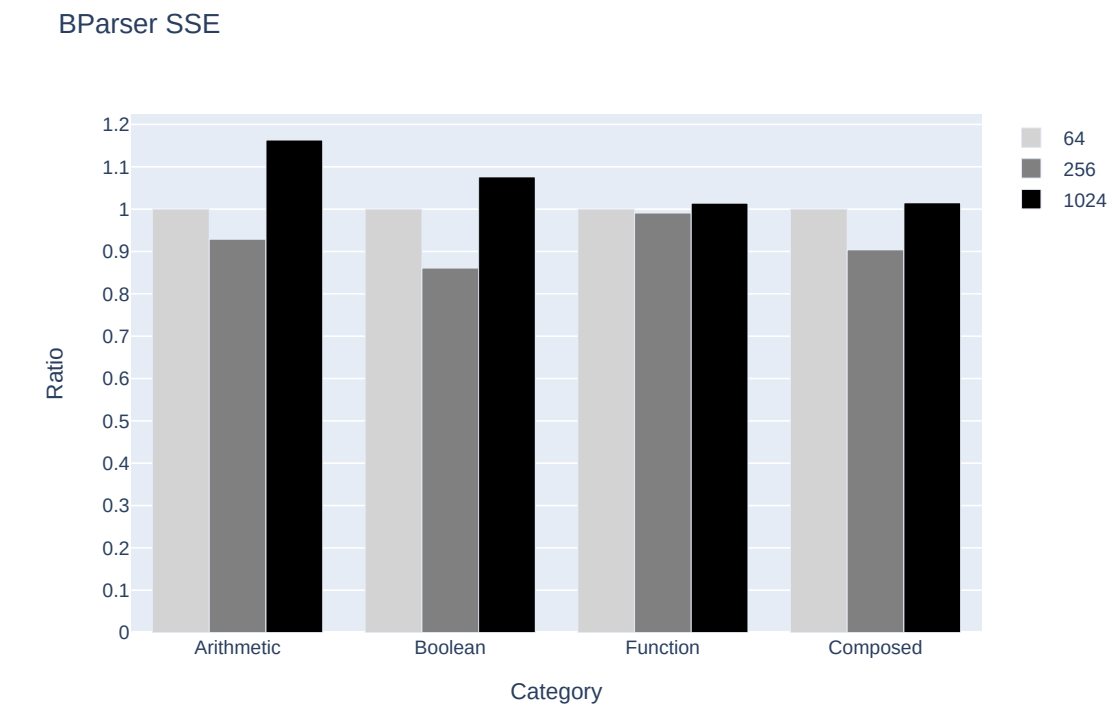
Obrázek 4.2: Graf porovnání poměru rychlostí v závislosti na velikosti vstupního vektoru u instrukční sady AVX512, zdroj: vlastní

Graf na obrázku 4.2 ukazuje, že pro instrukční sadu AVX512 jsou časy zpracování vektorů různých velikostí přibližně stejné pro všechny kategorie operací. Mírně rychleji probíhalo vyhodnocení výrazů při velikosti vektoru 256. Pro tento jev není zcela jasné vysvětlení.



Obrázek 4.3: Graf porovnání poměru rychlostí v závislosti na velikosti vstupního vektoru u instrukční sady AVX2, zdroj: vlastní

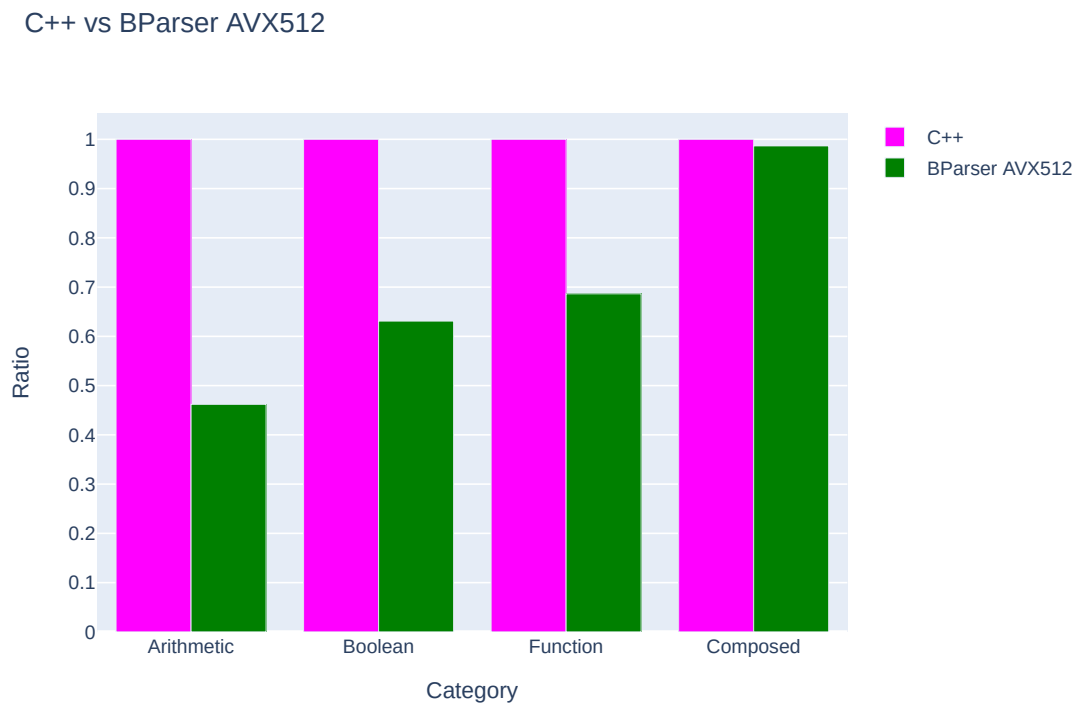
Na obrázku 4.3 vidíme graf porovnání průměrných časů vyhodnocení výrazů v závislosti na velikosti vstupního vektoru pro instrukční sadu AVX2. Obdobně jako pro AVX512 na grafu 4.2, tak i zde vychází jako nejlepší varianta velikost 256, zlepšení však je méně výrazné.



Obrázek 4.4: Graf porovnání poměru rychlostí v závislosti na velikosti vstupního vektoru u instrukční sady SSE, zdroj: vlastní

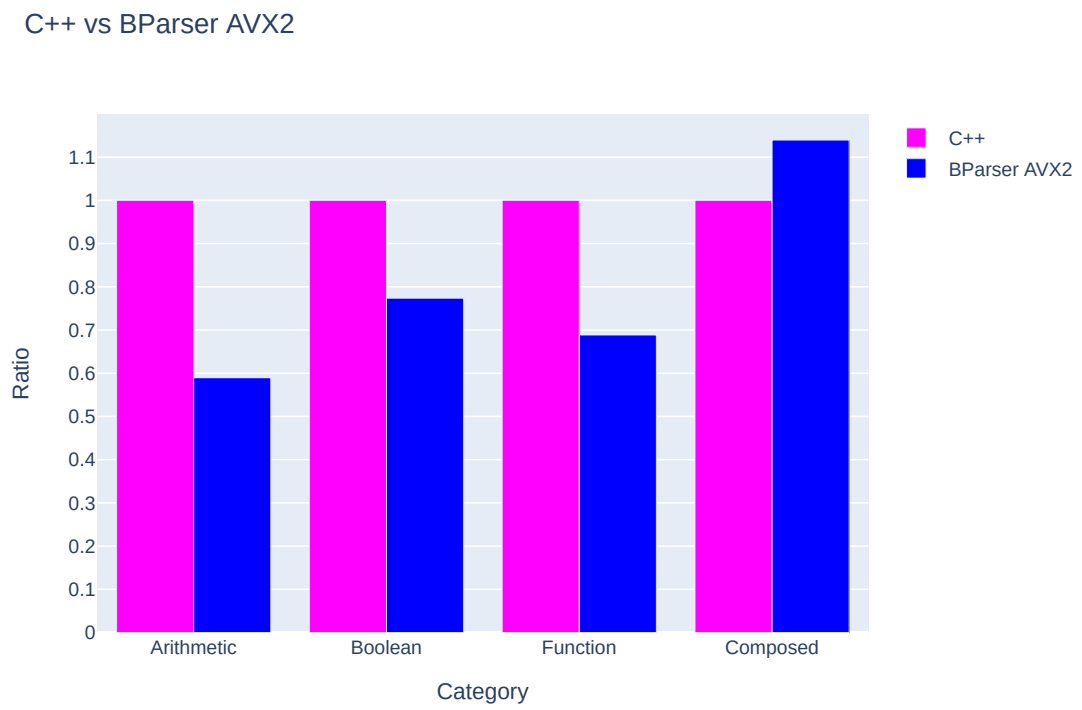
Z grafu porovnání velikostí vektoru na obrázku 4.4 lze opět vidět jako nejlepší variantu velikost 256. Takže nezáleží, na jaké instrukční sadě se test provádí, protože nejrychlejší velikost, byť ne o mnoho, je vždy 256. Jak bylo uvedeno již výše, pro tento jev nemáme zdůvodnění.

V dalším bloku testů porovnáváme čas vyhodnocení výrazů při zpracování pomocí C++ a BParseru s použitím různých instrukčních sad. Předpokládáme, že BParser bude ve vyhodnocování výrazů prvních tří kategorií rychlejší než C++, zatímco u složených výrazů rychlejší nebude, protože nemá podporu zřetězených operací a jednotlivé mezivýsledky ukládá do paměti a poté je zase načítá pro další výpočty.



Obrázek 4.5: Graf porovnání poměru rychlosti C++ a BParseru AVX512, zdroj: vlastní

Jak je na obrázku 4.5 vidět, zejména u aritmetických operací parser překračuje dvojnásobou rychlost oproti C++. Nejpomalejší je parser v případech složených výrazů, kdy není dobře optimalizované zřetězení operací a ukládání a načítání z a do paměti je pomalé. To odpovídá očekávaným výsledkům.



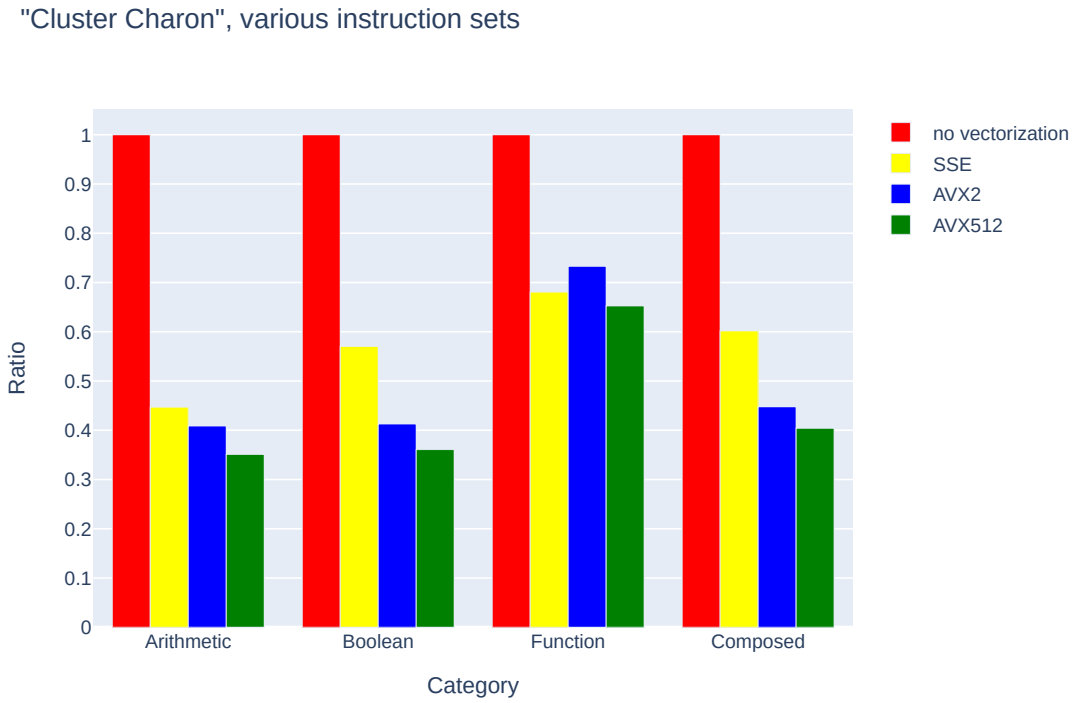
Obrázek 4.6: Graf porovnání poměru rychlosti C++ a BParseru AVX2, zdroj: vlastní

Porovnání rychlosti C++ s BParserem s využitím vektorové instrukční sady AVX2, na grafu z obrázku 4.6, pořád celkově vypadá rychlejší vyhodnocení pro BParser. V případě složených výrazů je již parser pomalejší. Instrukční sada AVX2 pracuje s poloviční šířkou vektoru než AVX512, tedy přístupy do paměti se zdvojnásobily. Celkově tedy došlo k poměrnému zpomalení BParseru oproti výsledku z grafu 4.5.

Obrázek 4.7: Graf porovnání poměru rychlosti C++ a BParseru SSE, zdroj: vlastní

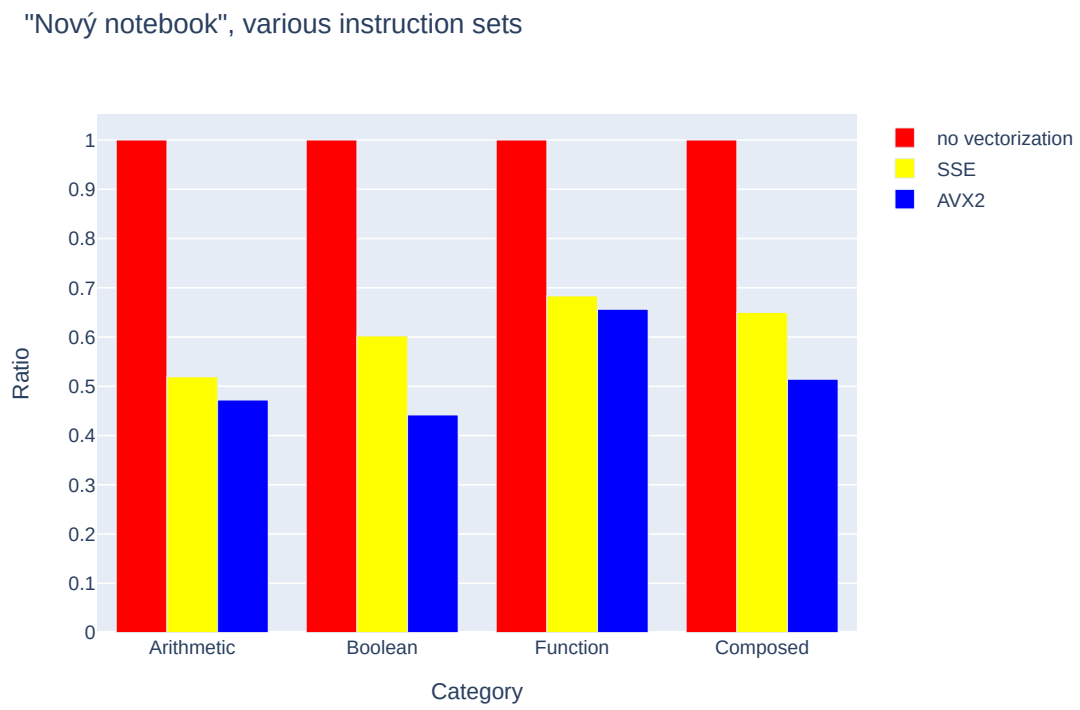
Porovnání C++ a BParseru s použitím instrukční vektorové sady SSE. Na obrázku 4.7 je vidět další zpomalení oproti variantě s AVX2 a s AVX512. U matematických funkcí je parser ještě stále o něco málo rychlejší než C++, zatímco u aritmetických a porovnávacích operací je pomalejší. Nejhorší výsledek je ale u složených operací, kdy je BParser pomalejší více jak dvakrát. Tím se potvrzuje trend, že absence zřetězených operací a ještě častější přístup do paměti BParser znevýhodňuje oproti C++.

Ve třetí skupině testů porovnáváme rychlosti vyhodnocení výrazů na použité sadě instrukcí na jednom počítači. Očekáváme, že každá novější instrukční sada bude zhruba 2x rychlejší než sada předchozí, protože operuje s dvakrát větším vektorem na dvojnásobné šířce registru.



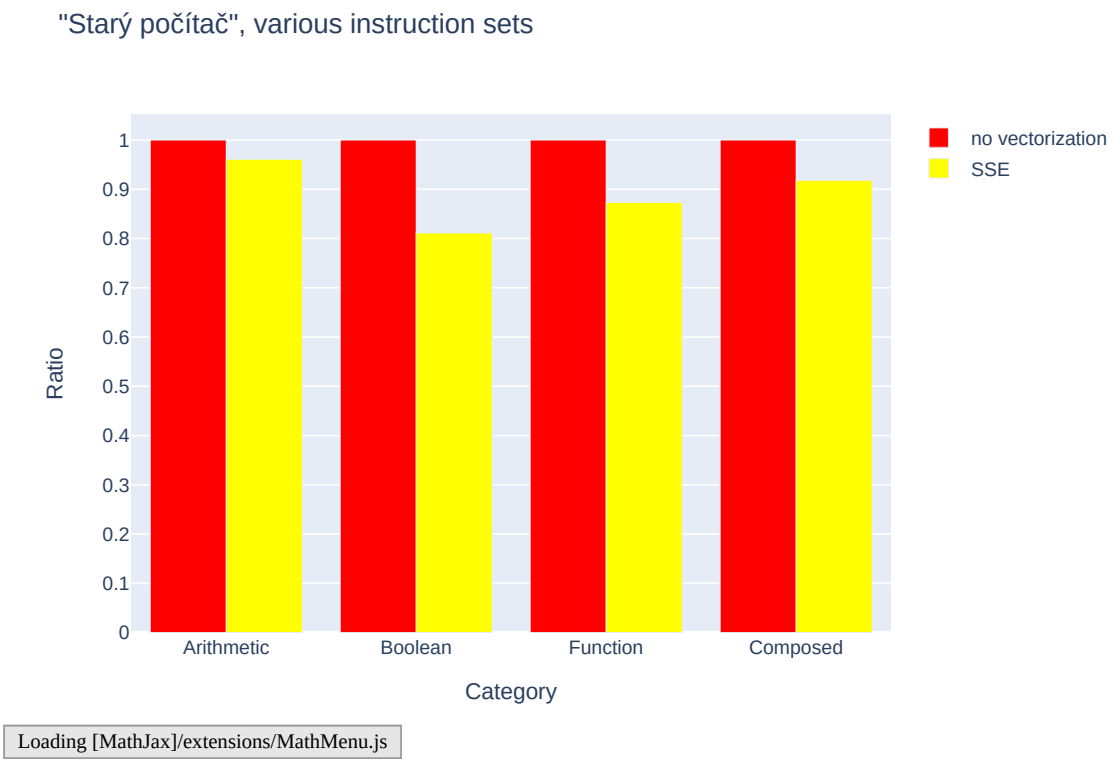
Obrázek 4.8: Graf porovnání poměrů rychlostí v závislosti na použité sadě instrukcí, zdroj: vlastní

Graf z obrázku 4.8 ukazuje velký rozdíl mezi vektorizovaným a nevektorizovaným výpočtem. Dle očekávání je rychlost vyšší, je-li využita vektorizace s většími registry. Výjimka je v kategorii funkcí, kde jsou celkově vektorizované varianty pomalejší než v ostatních kategoriích, a navíc varianta s instrukční sadou AVX2 je pomalejší než varianta s použitou instrukční sadou SSE. Při prozkoumání konkrétních časů u jednotlivých operací vychází všechny hyperbolické funkce u varianty AVX2 pomalejší než u ostatních variant. Zrychlení nedosahuje očekávaného poměru 1:2:4:8.



Obrázek 4.9: Graf porovnání poměrů rychlostí v závislosti na použité sadě instrukcí, zdroj: vlastní

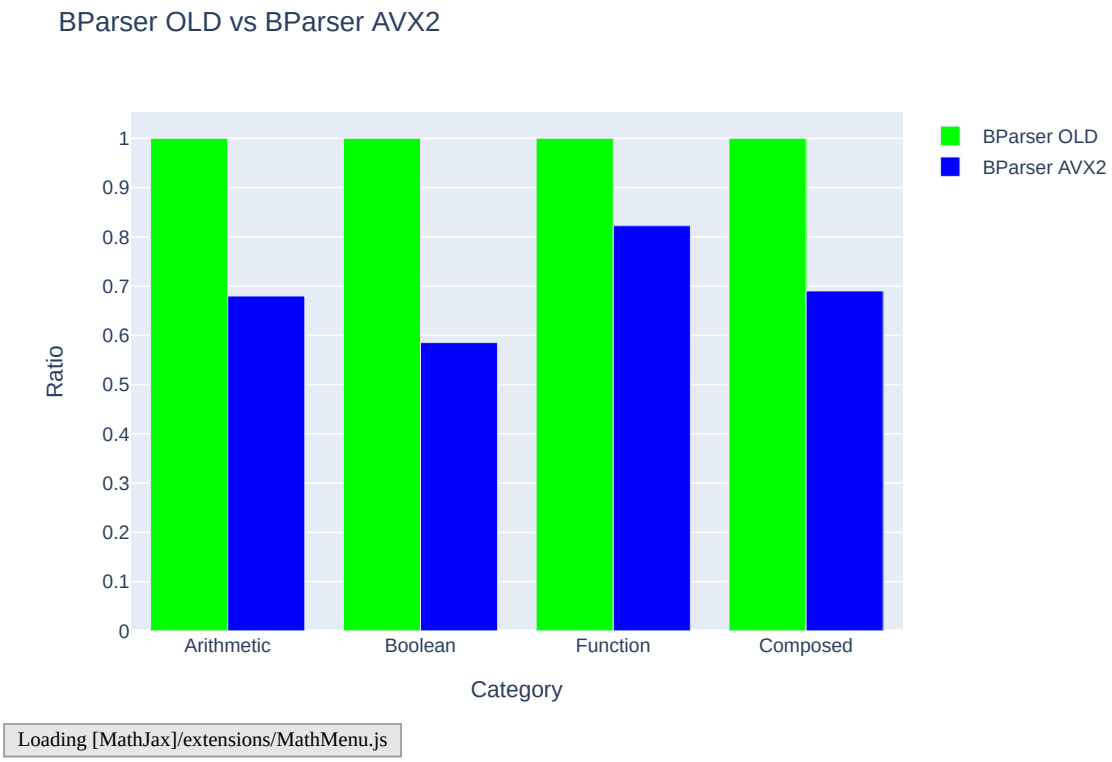
Na „novém notebooku“ nebylo možné testovat sadu instrukcí AVX512, zbylé dvě sady vyhodnocovaly výrazy dle očekávání rychleji než přístup bez vektorizace. U instrukční sady SSE se blížíme očekávanému dvojnásobnému zrychlení. Sada AVX2 výpočet nadále zrychlí, ale už ne dvojnásobně oproti SSE.



Obrázek 4.10: Graf porovnání poměrů rychlostí v závislosti na použité sadě instrukcí, zdroj: vlastní

Na obrázku 4.10 je porovnání nevektorizovaného přístupu a přístupu pomocí instrukční sady SSE. V tomto případě na tomto počítači je zrychlení u všech kategorií relativně malé. Toto malé zrychlení oproti zbylým dvěma počítačům je pravděpodobně způsobeno tím, že na „starém počítači“ bylo možné využít jen instrukční sady SSE2, nikoliv SSE4 jako na ostatních počítačích.

Poslední skupina grafů porovnává rychlost vyhodnocování výrazů původním BParserem bez použité knihovny a BParser s knihovnou VCL2 a instrukční sadou AVX2 na „novém notebooku“. Očekáváme, že dojde ke zrychlení vyhodnocení, jak bylo naznačeno už během prvotních testů při zahájení vývoje, jak zachycuje graf na obrázku 4.1.



Obrázek 4.11: Graf porovnání poměru rychlosti BParser OLD a BParser AVX2, zdroj: vlastní

Graf na obrázku 4.11 ukazuje zrychlení vyhodnocení ve všech kategoriích operací přibližně v míře, jak naznačoval původní test na základních operacích. Poměrně nejméně se projeví zrychlení u kategorií funkcí tak, jak tomu bylo v předchozích skupinách testů. To vše naznačuje, že některé funkce v knihovně nejsou navrženy optimálně. To ostatně odpovídá i informacím o účinnosti funkcí uvedeným v dokumentaci knihovny VCL2.

Obrázek 4.12: Graf porovnání poměru rychlosti BParser OLD a BParser AVX2 na aritmetických operacích, zdroj: vlastní

V tomto grafu jsme se detailně zaměřili na porovnání jednotlivých výrazů ze skupiny `Arithmetic`. Zrychlení u všech operací dosahuje podobné výše, pouze u sčítání došlo k poněkud překvapivému výraznému zrychlení.

5 Závěr

V rámci této diplomové práce bylo mým cílem prozkoumat možnosti vektorového vyhodnocování matematických výrazů. Dále jsem se měl seznámit s vektorovými instrukčními sadami používaných v procesorech architektury x86-64. Jako další úkol jsem měl vybrat C++ knihovnu, která podporuje vektorové operace pomocí vektorových instrukčních sad. Tuto knihovnu vyzkoušet na základní operace v BParseru. Poté jsem měl zajistit, aby bylo kód možné přenášet a spouštět na počítačích, jejichž procesory podporují různé vektorové instrukční sady. Dále jsem měl rozšířit použití knihovny tak, aby se zprovoznily veškeré operace parseru. A nakonec ověřit přenositelnost celého kódu.

Po prostudování seznamu knihoven jsem vybral vhodnou knihovnu Vector Class Library verze 2. Tato knihovna mi přišla vhodná zejména díky velké podpoře vektorových instrukčních sad. Dále podporuje mnoho překladačů jako CLang, GCC, Microsoft Visual C++ a Intel C++. A nejdůležitější byla její podpora všech operací, které se nachází v BParseru. Implementace knihovny na základní operace parseru si vyžádala změnu použitého datového typu. Tam jsem začal používat šablony na datové typy z knihovny tak, abych splnil možnost spouštět kód na různých CPU. Při řešení problémů, které se naskytly při vývoji, jsem využil specializaci struktur a konverzi hodnot typu double na typ boolean a obráceně. Je to z důvodu, že knihovna využívá jiné datové typy pro uložení vektorů s hodnotami double a boolean. Dále jsem se snažil, aby se v produkčním kódu použily maximální optimalizace pro jednotlivé varianty instrukčních sad.

Zajištění přnositelnosti jednoho binárního spustitelného souboru jsem docílil pomocí přidání pravidel pro kompilaci stejného kódu vícekrát, pokaždé s jinou instrukč-

ní sadou, a až za běhu se rozhodne podle konkrétního procesoru, která varianta se spustí. Pro test rychlosti vyhodnocení výrazů jsem vytvořil sadu testovacích výrazů, která obsahuje všechny operace BParseru doplněné o výrazy složené z několika operací. Abych mohl zobrazit výsledky rychlostních testů, vytvořil jsem si skript v jazyce Python.

Použitím knihovny VCL2 jsem dosáhl zrychlení vyhodnocení výrazů o jednu třetinu při použití na stejném hardware. V zadání uvažované využití principu maskování jsem nemusel využít, neboť daná funkcionalita je již obsažena v samotné knihovně. Dokázal jsem, že je testy možné spustit na jakémkoliv hardware. Jednak otestováním na běžně dostupných počítačích, na clusteru Charon, a také pomocí GitHub Actions.

Dalším vývojem by se dalo dosáhnout ještě většího zrychlení vyhodnocování výrazů pomocí víceúrovňových výpočtů, při nichž by se u podmíněných výrazů vypočítával jen ten výsledek, který se dále použije. Jinou možností zrychlení by bylo využití zřetězených operací a omezení tak časově náročného ukládání a načítání mezivýsledků z paměti.

Veškeré změny a vylepšení jsem prováděl na neustále vyvíjeném produkčním kódu. Vzhledem k jeho obsáhlosti a velkým závislostem je i malá úprava poměrně velkým zásahem v kódu a jeho ladění je tak velmi obtížné, zejména pak odstraňování chyb s alokací paměti.

Použitá literatura

[1] HAAS, Antonín. *Syntaktická analýza shora-dolů*. Olomouc, 2015. Bakalářská práce. Univerzita Palackého v Olomouci, Přírodovědecká fakulta, Katedra informatiky.

[2] BŘEZINA, Jan a David FLANDERKA. *BParser* [GitHub repository: https://github.com/flow123d/bparser]. 2019-2023.

[3] BŘEZINA, Jan et al. *Flow123d* [GitHub repository: http://github.com/flow123d/flow123d]. 2011–2023.

[4] GUZMAN, Joel de a Hartmut KAISER. *Spirit 2.5.5*. 2001-201. Dostupné také z: https://www.boost.org/doc/libs/1_67_0/libs/spirit/doc/html/index.html.

[5] PARTOW, Arash. *C++ Mathematical Expression Toolkit* [GitHub repository: https://github.com/ArashPartow/exprtk]. 2002–2021.

[6] KOBALICEK, Petr. *MathPresso* [GitHub repository: https://github.com/kobalicek/mathpresso]. 2015-2020.

[7] *Muparser - fast math parser library*. 2020. Dostupné také z: https://beltoforion.de/en/muparser/.

[8] BERG, Ingo. *muParserSSE* [GitHub repository: https://github.com/beltoforion/muparsersse]. 2011–2020.

[11] KARPIŃSKI, P. a J. MCDONALD. A High-Performance Portable Abstract Interface for Explicit SIMD Vectorization. In: *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*. Austin, TX, USA: Association for Computing Machinery, 2017,

s. 21–28. PMAM'17. ISBN 9781450348836. Dostupné z DOI: 10.1145/3026937.3026939.

[12] KARPINSKI, Przemyslaw. *UME::Vector* [GitHub repository: https://github.com/edanor/umevector]. 2016-2019.

[13] FOG, Agner. *version2* [GitHub repository: https://github.com/vectorclass/version2]. 2019-2023.

[14] KRETZ, Matthias. *Vc* [GitHub repository: https://github.com/VcDevel/Vc]. 2014-2022.

[15] KRETZ, Matthias. *std-simd* [GitHub repository: https://github.com/VcDevel/std-simd]. 2019-2021.

[16] KELLOGG, Matthew. *QuickVec* [Bitbucket repository: https://bitbucket.org/kellogg92/quickvec/src/master/]. 2015.

[17] CINCINESH. *The Tuesday C++ Vector Math and SIMD Library* [GitHub repository: https://github.com/Cincinesh/tue]. 2015.

[18] SHIBATA, Naoki a Francesco PETROGALLI. SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions. *IEEE Transactions on Parallel and Distributed Systems*. 2020, roč. 31, č. 6, s. 1316–1327. Dostupné z DOI: 10.1109/TPDS.2019.2960333.

[19] HOLYWU. *Warnings on gcc 7.1.0*. 2017. Dostupné také z: https://www.agner.org/optimize/vectorclass/read.php?i=160.

[20] HANS, L. *False warnings about "optimization attribute" on operators when -fno-ipa-cp-clone*. 2019. Dostupné také z: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=89325.

[21] *Features • GitHub Actions*. 2022. Dostupné také z: https://github.com/features/actions.

[22] FOG, Agner. *VCL C++ vector class library manual*. Denmark, 2022. Dostupné také z: https://raw.githubusercontent.com/vectorclass/manual/master/vcl_manual.pdf.

Použité obrázky

[9] STOKES, Jon. *Single Instruction stream, Single Data stream*. 2000. Dostupné také z: https://cdn.arstechnica.net/wp-content/uploads/archive/cpu/1q00/simd/figure6.gif.

[10] MIRALLES, Damian et al. *AVX512 ZMM registers for SIMD operations as an extension of the AVX YMM register and SSE XMM registers*. 2018. Dostupné také z: https://www.researchgate.net/profile/Damian-Miralles/publication/333609595/figure/fig1/AS:766068905758720@1559656468071/AVX512-ZMM-registers-for-SIMD-operations-as-an-extension-of-the-AVX-YMM-register-and-SSE.ppm.

A Přílohy

- Binární přenositelný soubor `test_bin`
- Kopie obrazovky a textové logovací soubory o spuštění `test_bin` na různých počítačích
- Detailní výsledky z provedených testů rychlosti