

# **Datově orientovaný přístup při vývoji software**

Bc. Tomáš Janečka

---

Diplomová práce  
2023



Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky  
Ústav informatiky a umělé inteligence

Akademický rok: 2022/2023

## ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: Bc. Tomáš Janečka  
Osobní číslo: A20131  
Studijní program: N0613A140022 Informační technologie  
Specializace: Softwarové inženýrství  
Forma studia: Prezenční  
Téma práce: Datově orientovaný přístup při vývoji software  
Téma práce anglicky: Data-Oriented Software Design

### Zásady pro vypracování

1. Definujte pojem datově orientovaný návrh a seznamte se s touto problematikou.
2. Porovnejte tento způsob návrhu s objektově orientovaným návrhem.
3. Popište vliv mikroarchitektury počítače na rychlosť běhu programu.
4. Demonstrařte jednotlivé principy na příkladech.
5. Ověřte efektivitu programů pomocí nástrojů pro výkonnostní testy a profilování.
6. Sestavte sadu doporučení pro využití datově orientovaného přístupu.

Forma zpracování diplomové práce: **tištěná/elektronická**

**Seznam doporučené literatury:**

1. FABIAN, Richard. Data-Oriented Design: Software engineering for limited resources and short schedules [online]. Richard Fabian, 2018, 307 s. ISBN 9781916478701.
2. STROUSTRUP, Bjarne. The C++ Programming Language. 4th Edition. Addison-Wesley Professional, 2013, 1376 s. ISBN 0275967301.
3. NESTERUK, Dmitri. Design Patterns in Modern C++: Reusable Approaches for Object-Oriented Software Design. New York: APress, 2018. ISBN 978-1484236024.
4. KUSSWURM, Daniel. Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX. Apress, 700 s. ISBN 1484200659.
5. BRYANT, Randal a David O'HALLORON. Computer Systems: A Programmer's Perspective. 3rd Edition. Pearson, 1128 s. ISBN 013409266X.

Vedoucí diplomové práce:

**Ing. Peter Janků, Ph.D.**

Ústav informatiky a umělé inteligence

Datum zadání diplomové práce:

**2. prosince 2022**

Termín odevzdání diplomové práce: **26. května 2023**

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.  
děkan



prof. Mgr. Roman Jašek, Ph.D., DBA v.r.  
ředitel ústavu

Ve Zlíně dne 7. prosince 2022

**Prohlašuji, že**

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

**Prohlašuji,**

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 15.5.2023

Tomáš Janečka, v.r.

## **ABSTRAKT**

Diplomová práce se zabývá tématem datově orientovaného vývoje software. V teoretické části student popisuje jednotlivé techniky návrhu software, charakteristiku používaného hardware a vybrané aspekty ovlivňujících výkonnost implementovaných algoritmů. V praktické části jsou srovnány jednotlivé způsoby implementace algoritmů včetně praktických příkladů doplněné o výstupy nástrojů pro výkonnostní testy a profilování.

Klíčová slova: datová orientace, C++, optimalizace, výkon, benchmarking, profilování, vyrovnávací paměť, operační paměť

## **ABSTRACT**

This diploma thesis is concerned with the topic of data-oriented software design. In the theoretical part, the student is tasked with describing software design techniques, the characteristics of the hardware used and pointing out selected aspects that determine the performance of the implemented algorithms. The practical part shows the comparison of the different ways of algorithm implementations, along with examples including outputs from performance measuring tools and profilers.

Keywords: data-oriented, C++, optimization, performance, benchmarking, profiling, cache, memory

Děkuji vedoucímu práce Ing. Peterovi Janků, Ph.D. za předmětné konzultace a poskytnutí hodnotných rad zejména ohledně formálního zpracování. Rovněž oceňuji dostatek volnosti při vypracovávání a výběru použitých technologií.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

# OBSAH

<b>ÚVOD.....</b>	<b>10</b>
<b>1 TEORETICKÁ ČÁST.....</b>	<b>11</b>
<b>1 DATOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ .....</b>	<b>12</b>
1.1 DEFINICE .....	12
1.2 HISTORICKÝ VÝSKYT .....	12
1.3 HLAVNÍ MYŠLENKY .....	13
1.3.1 Je to o datech.....	13
1.3.2 Data nejsou problémová doména .....	13
1.3.3 Statistika.....	14
1.3.4 Data se mění .....	14
1.4 VYUŽITÍ .....	14
<b>2 DALŠÍ PARADIGMATA.....</b>	<b>16</b>
2.1 OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ.....	16
2.2 FUNKCIONÁLNÍ PROGRAMOVÁNÍ.....	16
2.3 POROVNÁNÍ.....	17
<b>3 HARDWARE JAKO PLATFORMA.....</b>	<b>19</b>
3.1 VYROVNÁVACÍ PAMĚŤ .....	19
3.1.1 Struktura a adresování.....	20
3.1.2 Pojmy .....	21
3.1.3 Vyrovnávací paměť s přímým mapováním.....	21
3.1.4 Set associative vyrovnávací paměť .....	21
3.1.5 Plně asociativní vyrovnávací paměť .....	22
3.1.6 Substituční strategie .....	22
3.1.7 Způsoby zapisování.....	22
3.2 OPERAČNÍ PAMĚŤ .....	23
3.3 CPU PIPELINING .....	23
3.3.1 Fáze instrukčního kanálu.....	23
3.3.2 Sekvenční zpracování.....	24
3.3.3 Skalární pipeline.....	24
3.3.4 Superskalární pipeline .....	24
3.3.5 Ideální vs reálné řešení .....	25
3.3.6 Závislosti .....	25
3.4 PREFETCHING .....	27
3.4.1 Softwarový prefetching .....	27
3.4.2 Hardwarový prefetching .....	27
3.4.3 Execution based prefetching .....	27
3.4.4 Address correlation based prefetching .....	28
3.4.5 Content directed prefetching .....	28
3.5 LOKALITA .....	28
3.6 JAZYK SYMBOLICKÝCH ADRES .....	29
3.6.1 Vnitřní architektura x86-32.....	29
3.6.2 Adresovací módy .....	30
3.6.3 Instrukce pro přesun dat .....	31

3.6.4	Instrukce pro aritmetické operace .....	31
3.6.5	Instrukce pro bitovou rotaci a posuv .....	31
3.6.6	Instrukce pro předání kontroly .....	31
3.7	PŘEKLADAČE .....	32
3.7.1	GCC.....	32
3.7.2	Clang .....	33
3.7.3	MSVC .....	34
3.8	SIMD.....	34
3.9	PARALELIZACE.....	38
3.9.1	Instruction-level paralelismus .....	38
3.9.1.1	Loop unrolling .....	39
3.9.1.2	Předpovídání větvení .....	40
3.9.2	Data-level paralelismus .....	40
3.9.2.1	Strip mining .....	41
3.9.2.2	Výpočty na GPU .....	41
3.9.2.3	Loop-level paralelismus.....	41
3.9.3	Thread-level paralelismus .....	42
3.9.3.1	Amdahlův zákon .....	43
3.9.3.2	Koherence vyrovnávacích pamětí.....	44
3.9.3.3	Synchronizační primitiva .....	44
3.9.3.4	POSIX vlákna .....	45
3.9.3.5	Semafor.....	47
4	<b>MĚŘENÍ VÝKONU APLIKACÍ.....</b>	<b>49</b>
4.1	BENCHMARKING .....	49
4.1.1	Microbenchmark .....	49
4.2	PROFILOVÁNÍ .....	50
4.2.1	Vzorkovací profilery .....	50
4.2.2	Instrumentační profilery .....	52
5	<b>POUŽITÉ TECHNOLOGIE.....</b>	<b>58</b>
5.1	PROGRAMOVACÍ JAZYK C++ .....	58
5.2	VISUAL STUDIO CODE .....	59
5.3	VISUAL STUDIO 2019 .....	60
5.4	CMAKE .....	61
5.5	HARDWAROVÉ SPECIFIKACE.....	62
<b>II</b>	<b>PRAKTICKÁ ČÁST .....</b>	<b>65</b>
6	<b>PŘÍKLADY OPTIMALIZACÍ .....</b>	<b>66</b>
6.1	ITERACE POLEM.....	66
6.1.1	Iterace dvourozměrným polem .....	66
6.1.2	Závislost iterací .....	69
6.1.3	Loop unrolling.....	70
6.1.4	Projevy asociativity vyrovnávacích paměti.....	71

6.2	PŘEDVÍDATELNOST OPERACÍ .....	71
6.3	HOT VS COLD DATA .....	72
6.4	SOA vs AoS .....	73
6.5	SIMD .....	75
6.6	ZAROVNÁNÍ DAT .....	76
6.7	DATOVÉ A KONTROLNÍ ZÁVISLOSTI.....	76
6.8	PRAVÉ A FALEŠNÉ SDÍLENÍ .....	78
6.9	ALIASING PAMĚTI.....	79
6.10	NÁVRHOVÉ VZORY Z POHLEDU DOP.....	80
6.10.1	Existence based processing .....	80
6.10.2	Component based objects.....	81
6.10.3	Adapter .....	82
6.10.4	Composite .....	82
6.10.5	Memento .....	82
<b>7</b>	<b>OVĚŘENÍ VÝKONU POMOCÍ NÁSTROJŮ .....</b>	<b>83</b>
7.1	MĚŘENÍ DOBY VYKONÁVÁNÍ UKÁZKOVÝCH PŘÍKLADŮ .....	83
7.2	ANALÝZA UKÁZKOVÝCH PŘÍKLADŮ PROFILOVACÍMI NÁSTROJI.....	98
7.2.1	Zdrojový kód 1 .....	98
7.2.2	Zdrojový kód 4 .....	102
7.2.3	Zdrojový kód 7 .....	106
7.2.4	Zdrojový kód 9 .....	109
<b>8</b>	<b>SADA DOPORUČENÍ.....</b>	<b>112</b>
8.1	OBECNÁ DOPORUČENÍ .....	112
8.2	DOPORUČENÍ PRO OPTIMALIZACE.....	113
<b>ZÁVĚR .....</b>	<b>116</b>	
<b>SEZNAM POUŽITÉ LITERATURY.....</b>	<b>117</b>	
<b>SEZNAM POUŽITYCH SYMBOLŮ A ZKRATEK.....</b>	<b>120</b>	
<b>SEZNAM OBRÁZKŮ .....</b>	<b>121</b>	
<b>SEZNAM TABULEK.....</b>	<b>124</b>	
<b>SEZNAM PŘÍLOH.....</b>	<b>126</b>	

## ÚVOD

Objektově orientované programování je jedno z nejrozšířenějších paradigm mezi programátory. Mnoho z nich, včetně autora diplomové práce, se s ním setkali na úplném začátku své programátorské kariéry. Jedná se o velmi užitečný nástroj. Každý nástroj má ale svůj účel, a není určen k řešení každého problému. Jedním ze zásadních rozdílů mezi objektově orientovaným a datově orientovaným návrhem je ta věc, kterou tyto způsoby programování považují jako hlavní. Objektově orientovaný návrh se soustředí na vytvoření abstraktního, idealizovaného a také co nejobecnějšího modelu reálného problému.

Naproti tomu datově orientovaný návrh považuje data za to nejvýznamnější. Zároveň se toto paradigma soustředí na charakteristiky hardware, na kterém se software vykonává a dbá na efektivní využívání zdrojů. Dostupnost výpočetních zdrojů je v porovnání s minulostí nesrovnatelná, což ale neznamená, že je pro určité aplikace dostačující. Z tohoto důvodu je tento způsob tvorby programů mimo jiné využíván v herním průmyslu. Právě herní vývojáři tvoří naučné podklady o tomto tématu, ve kterých často poukazují na podstatné nuance při souhře hardware a software. Tyto zdánlivé detaily však často mají zásadní vliv na rychlosť běhu programu a také na využití operační paměti. Autor tyto nápady považuje za velmi zajímavé a aktuální, a proto tato diplomová práce poskytne shrnutí nejzásadnějších myšlenek tohoto paradigmata.

V teoretické části bude představena definice pojmu datově orientované programování a budou popsány hlavní myšlenky. Rovněž je třeba se zabývat tématem mikroarchitektury počítače, jelikož jeho znalost je pro využití v této oblasti kritická.

V praktické části bude představeno množství praktik, které lze aplikovat na tvorbu programu a budou porovnána běžná řešení často řešených problémů a také zhodnoceny výstupy výkonnostních testů a profilovacích nástrojů. Na základě empirických dat bude sestavena sada doporučení pro čtenáře, kteří by se chtěli zdokonalit v této oblasti a využít benefitů, které psaní programů tímto způsobem přináší.

## **I. TEORETICKÁ ČÁST**

## 1 DATOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ

V této sekci bude představen pojem datově orientované programování a datově orientovaný návrh. Kromě dále uvedených definic je možné o tomto paradigmatu říci, že se jedná o způsob, jakým vyvíjet software. Zároveň ale může koexistovat s kódem, který byl napsán způsobem jiným ve stejném projektu. Tento nástroj se neváže ke konkrétní oblasti problémů či programovacích jazyků. V žádném případě se nejedná o něco, co by nebylo použito v minulosti, byť třeba pod jiným jménem. Ačkoliv nejde o nový koncept, samotný pojem „data-oriented“ se ve vývojářských kruzích začal vyskytovat teprve nedávno. I z tohoto důvodu je třeba při této diplomové práci využít omezeného množství literatury, která se zabývá tímto tématem, a také většího množství záznamů přednášek z programátorských konferencí. Navíc si může člověk při studiu této problematiky všimnout, že každý řečník či autor si pod tímto pojmem představuje něco trochu jiného. Některé koncepty může zcela zanedbat a také může představit něco, o čem nikdo před ním nemluvil.

### 1.1 Definice

Definovat toto paradigma lze mnoha různými způsoby. Je uvedena definice nalezená v literatuře.

„Datově orientovaný návrh je dovednost navrhnut software pomocí vývoje transformací pro data v řádné formě, kde řádná forma je řízena cílovým hardwarem a transformacemi, které na něm běží.“ [1]

„Datově orientovaný návrh si nechává napovědět daty, která jsou pozorovatelná nebo očekávaná. Na rozdíl od uvažování všech možných scénářů nebo plánování adaptability, využíváme nejpravděpodobnější vstupy pro směřování algoritmu. Na rozdíl od plánování rozšířitelnosti je jednoduchý a má za cíl plnit úkoly.“ [1]

### 1.2 Historický výskyt

Článek na téma „data-oriented“, který jako jeden z prvních použil tento termín a také měl za cíl seznámit čtenáře s touto myšlenkou, vyšel v roce 2009 v časopisu Game Developer, jehož autorem je Noel Llopis. Jedná se o příspěvek od herního vývojáře o způsobu vývoje her v časopisu pro herní vývojáře. Není divu, že toto paradigmata pramení právě z oblasti, kde je souhra software a hardware klíčem k úspěchu. V tomto článku Llopis pojednává o tom, jak by všudypřítomné objektově orientované programování mohlo být příčinou nízkého výkonu

her. Je v něm uvedeno, na kterou věc se různé programovací přístupy soustředí a jak se od nich datově orientovaný přístup liší. Autor článku dále uvádí svůj výčet výhod tohoto přístupu, a to paralelizace, využití vyrovnávací paměti, modularita a jednoduchost testování. Závěrem jsou představeny rady, jak začlenit tento přístup do aktuálně vyvíjené aplikace a jak data získat a co je důležité sledovat. Llopis se ještě vyjadřuje k tomu, že pro objektově orientovaný návrh rozhodně existuje místo a nechce ho démonizovat. Například „v systémech, které byly navrženy tímto způsobem nebo výkonově nekritické aplikace.“ [2]

Odkazy z tohoto příspěvku směřují na Mika Actona a Jima Tilandera. Oba jsou vlastníci blogů, kde v minulosti publikovali články na různá téma, které zjevně ovlivnily Llopisův přístup k vývoji aplikací a her. Z toho je zřejmé, že myšlenky datově orientovaného návrhu pramení z prvních let druhého tisíciletí. Jelikož se zde také klade důraz na vývoj software s ohledem na hardware, dalo by se říct, že byl tento přístup používán ještě mnohem dříve. V raných dobách výpočetní techniky byly, v porovnání s dnešní dobou, všechny dostupné prostředky velmi vzácné, a proto bylo nezbytné s nimi nakládat co nejfektivněji.

## 1.3 Hlavní myšlenky

### 1.3.1 Je to o datech

Všechny aplikace, co kdy byly napsány, slouží k poskytnutí výstupu v závislosti na vstupních datech. Grafické aplikace pracují s obrázky. Textové editory pracují s textem. Každá z nich očekává určitý formát dat. Ten může být velmi složitý, nebo velmi jednoduchý. Programátoři si také často neuvědomují, že instrukce jsou také data, protože se rovněž nachází v operační paměti. Za všech okolností je třeba myslet na to, že data nikdy neexistují jen tak, ale pokaždé se nachází na nějakém hardware, ať už na virtuálním stroji, nebo konkrétním procesoru. „Data jsou vše, co máme“. [1]

### 1.3.2 Data nejsou problémová doména

Na rozdíl od objektově orientovaného přístupu, datově orientovaný přístup obětovává čitelnost kódu, což umožňuje nezatěžovat počítač lidskými koncepty. Umístěním dat do třídy umožní dát těmto datům kontext, to ale může mít následek existence velkého množství dat, které spolu nesouvisí. Proto se v tomto paradigmatu uvažuje o datech jako o „faktech, o kterých lze uvažovat tak, jak je třeba pro získání výstupních dat v požadovaném formátu. Datově orientovaný návrh nezabudovává problém z reálného světa do kódu“. [1]

### 1.3.3 Statistika

Nejen vstupní data programu jsou zahrnuta do pojmu „data“. Data o datech mohou být stejně nebo i více významná. Mohou mít zásadní vliv na to, jak se píše kód. Pokud je k dispozici prvotní verzi funkční aplikace, která pracuje s produkčními daty, nebo se disponuje daty, která byla shromážděna libovolným způsobem, má programátor více kontextu a dokáže lépe uvažovat o problému, který řeší. „Data jsou typ, frekvence, množství, tvar a pravděpodobnost.“ [1]

Analýza dat může mít podobu prostého výpisu hodnoty proměnné. Stačí, když zvolíme libovolnou proměnnou, která nás zajímá a budeme sledovat vývoj jejích hodnot v čase. „Pokud porozumíme datům, porozumíme problému.“ [3]

Jelikož se tento přístup opírá o data, pro dosažení lepšího než dostatečného řešení problému je třeba získat co nejvíce dat o problému ještě před začátkem vývoje. Prvním krokem tvorby programů tímto způsobem by měl být návrh struktury dat za podpory informací o hardware a překladači, protože tyto prvky se významným způsobem podílí na transformaci dat. [4]

Jelikož vývoj aplikací může být velmi náročný na čas a prostředky, je rozumné investovat naše úsilí do 20 % kódu, ve kterém je tráveno 80 % času a má zásadní vliv na výkon programu.

### 1.3.4 Data se mění

Významná myšlenka je držet data a operace nad těmito daty zvlášť a neshlukovat je do tříd nebo jiných konstruktů. Díky tomu dokáže programátor lépe reagovat na změny a minimálnizovat náklady potřebného přepisu kódu. „Datově orientovaný návrh může pozorovat změnu v architektuře aplikace díky porozumění změn v datech.“ [1]

## 1.4 Využití

Jak již bylo zmíněno, datově orientovaný návrh je hojně využíván v herním průmyslu. Je to jedna z oblastí, kde se vývojáři snaží vytěžit co možná největší výkon ze své aplikace a zároveň musí respektovat omezení jednoho nebo více druhů hardware, na kterém bude běžet. Po seznámení s tímto paradigmatem a jeho hlavními myšlenkami a způsoby implementace je zřejmé, že použití tohoto přístupu má pro programátory jako jednotlivce i další zajímavé implikace. Jedna z nich je zařazení dalšího užitečného nástroje mezi své dovednosti. Jelikož pro aplikování datově orientovaného návrhu je důležitá znalost hardware, je programátor

nucen se vzdělávat v oblasti počítačové architektury, mikroarchitektury, operačních systémů a strojového kódu. Zároveň je zde potenciál lepšího porozumění problému, který je zrovna řešen programátorem. Protože pokud se porozumí datům, porozumí se problému a programátor může zjistit řadu hodnotných informací a podle toho může v budoucnu vylepšit kód. Mezi specifické informace by se dala zařadit frekvence volání určitých funkcí, studium hodnot proměnných měnících se v čase nebo vyzozorování vzorce opakování hodnot proměnných.

## 2 DALŠÍ PARADIGMATA

Programovací paradigmata je způsob nebo styl, kterým se píše kód. Nemusí se nutně vztahovat ke konkrétnímu programovacímu jazyku, i když různé programovací jazyky mají blíže k jednomu paradigmatu než ke druhému. V následující sekci se nachází srovnání paradigmata, kterým se zabývá tato diplomová práce, s dalšími paradigmaty, které se běžně používají, nebo začínají používat v aktuální době. Byla vybrána pouze ta, která jsou pro srovnání relevantní, protože mají například řešit nedostatky druhého.

### 2.1 Objektově orientované programování

„Objektově orientované programování je o modelování systému jako kolekci objektů, kde každý objekt představuje určitý aspekt systému. Objekty obsahují jak funkce, tak data. Objekt poskytuje veřejné rozhraní, které je přístupné v kódu, a také obsahuje svůj privátní, vnitřní stav; ostatní části systému se nemusí zajímat o to, co se děje uvnitř objektu.“ [5]

V rámci tohoto paradigmata se hojně využívá tříd. Třída je předpis pro vytváření instancí. Každá třída může mít vlastnosti, které charakterizují instance, a metody, které popisují chování. Mezi třídami mohou existovat vztahy. Jedním z nich je dědičnost, díky které jedna třída dědí vlastnosti a metody třídy druhé. [5]

Pokud rodičovská třída definuje metody jako virtuální, pak třídy, které od ní dědí, mohou měnit chování v závislosti na typu instance. Toto umožňuje mít kolekci objektů různého typu, ale pro interakci s nimi používat jednotný přístup. Ačkoliv se jedná o užitečnou věc, jsou to právě volání virtuálních funkcí, které v určitých případech mohou být příčinou zásadního zpomalení aplikace.

Zapouzdření je myšlenka vymezení veřejného rozhraní a zároveň schování detailů fungování objektu. V případě potřebné změny nám toto umožní změnit kód pouze na jednom místě, jelikož navenek je přístupné pouze veřejné rozhraní, které nebylo třeba měnit. [5]

### 2.2 Funkcionální programování

„Funkcionální programování je přístup k vývoji software, který používá ryzí funkce pro vytvoření udržitelného software. Jinými slovy se jedná o tvorbení programů, aplikací a kompozicí funkcí.“ [6]

Jak vyplývá z názvu, funkce je zde základní stavební jednotka. Kromě jejich běžného použití jsou funkce využívány taky jako proměnné, argumenty funkcí nebo návratové hodnoty

funkcí. Na rozdíl od ostatních paradigm se zde preferuje použití proměnných, jejichž hodnota se po deklaraci nemění. Základní myšlenky tohoto programovacího stylu pochází z matematického nástroje zvaného lambda kalkul, který byl popsán ve 30. letech minulého století Alonzem Churchem. Mezi funkcionální programovací jazyky se řadí Haskell, Erlang, Clojure, LISP, Scala a Elixir. Postupem času i běžné programovací jazyky zařazují do svého arzenálu nástroje, které pramení z tohoto způsobu programování. [6]

Ve funkcionálním programování je významné používání rekurze. V závislosti na implementaci toto může způsobit značné zpomalení aplikace v porovnání s implementací pomocí použití klasické iterace ve smyčce.

### 2.3 Porovnání

„Objektově orientovaný návrh je soustředěn na problém a jeho řešení. Objekty, abstraktní reprezentace věcí, které tvoří návrh řešení problému představeného v návrhovém dokumentu aplikace. Objekty manipulují pouze s těmi daty, které jsou potřeba pro jejich reprezentaci bez jakéhokoliv ohledu na hardware nebo na data z reálného světa nebo jejich množství. Z tohoto důvodu nám objektově orientovaný návrh umožní rychle sestavit první verze aplikací a tím pádem také první podobu kódu. Datově orientovaný návrh se k problému staví jinak. Na rozdíl od předpokladu, že nevíme nic o hardware, usuzujeme, že nevíme nic o řešeném problému.“ [1]

Rozdíl mezi objektově orientovaným a funkcionálním přístupem je ten, že zatímco OOP využívá imperativní přístup, který spočívá ve specifikaci kroků potřebných k vyřešení problému, FP využívá deklarativní přístup, který pracuje s výsledkem operace, nehledě na to, jak jsme k němu přišli. Dalším rozdílem je využití proměnných a konstantních proměnných. V FP se v případě přepisu vytvoří zcela nová proměnná, do které se překopíruje původní hodnota. Díky tomu se kód v případě potřeby snáze mění a lépe testuje a lépe se v něm hledají chyby. Doporučuje se využívat OOP pro standardizované a přímočaré projekty a FP pro aplikace, které je třeba škálovat a musí být flexibilní. [6]

Jedna věc, kterou sdílí funkcionální programování s datově orientovaným programováním je skutečnost, že obě paradigmata identifikovala nedostatky objektově orientovaného programování. Jejich společná překážka je potřeba uvažovat zcela odlišně, než jak člověk dosud myslí při psaní programu. Pro programátora odchovaného na objektově orientovaném návrhu mohou být myšlenky aplikované v těchto alternativních programovacích

přístupech relativně složité, ba i zpočátku nepochopitelné. Obě paradigmata se mohou vyskytovat po boku objektově orientovaného kódu ve stejné aplikaci a být použity tam, kde dávají smysl. Zatímco funkcionální návrh si klade za cíl zlepšit robustnost a modularitu aplikace, datově orientovaný přístup se soustředí jak na modularitu, tak na zvýšení výkonu programu.

### 3 HARDWARE JAKO PLATFORMA

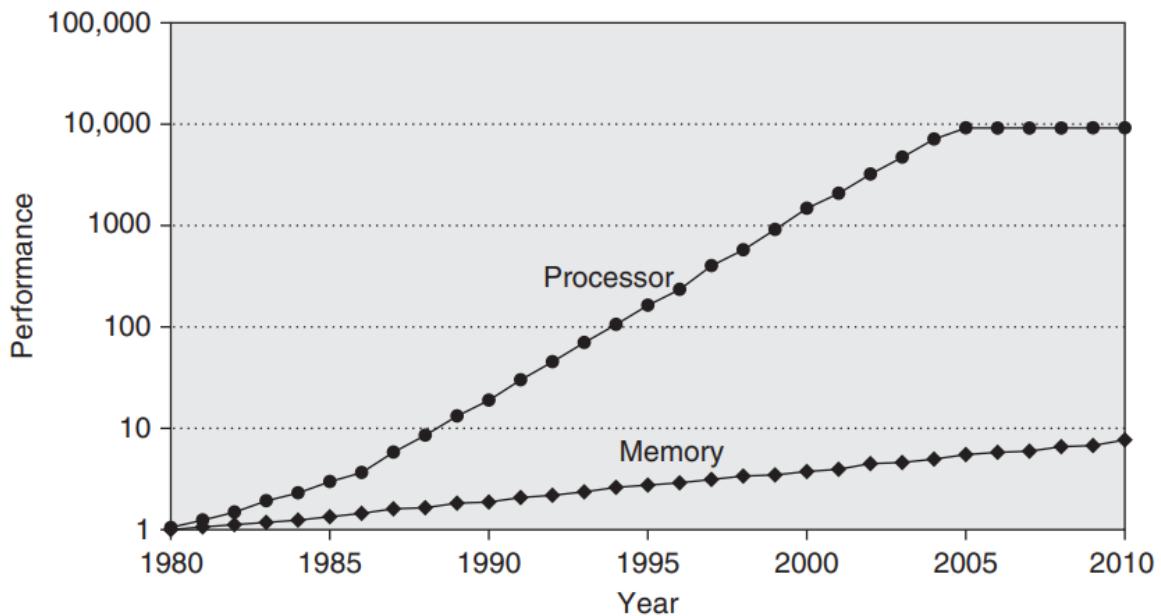
Datově orientovaný vývoj dbá na efektivní využívání hardware. V této sekci budou popsány části počítače a jednotlivých komponentů, které mají zásadní vliv na rychlosť běhu programu.

#### 3.1 Vyrovnávací paměť

Vyrovnávací paměť (cache) je rychlé paměťové zařízení s malou kapacitou, které slouží k ukládání malých částí dat z paměťových zařízení nižších úrovní paměťové hierarchie. Pokud se mluví o cache pamětech, myslí se zpravidla uložiště, které se nachází na procesorovém čipu a je k dispozici výpočetním jádrům. Jako vyrovnávací paměť se ale může považovat i operační paměť ve vztahu k hard-disku nebo SSD. [7]

Cache paměť pro ukládání dat využívá technologie SRAM. V moderních procesorech se vyskytuje v několika úrovních. Každá úroveň má různou velikost a přístupovou dobu. Nejblíže k výpočetnímu jádru je úroveň L1. Tu ještě výrobci CPU separují na paměť pro instrukce a data, nazývané *i-cache* a *d-cache*. Tato úroveň bývá privátní pro každé jádro. Na další úrovni se nachází úroveň L2. Ta je unifikovaná, takže obsahuje jak instrukce, tak data. Může být privátní pro jedno jádro nebo sdílená mezi všemi jádry. Nejvýše postavená je úroveň L3. Ta je společná pro všechna jádra a má největší kapacitu. [7]

Vývoj vyrovnávacích pamětí je jeden z přispěvatelů k relativně obrovským výkonům dnešních CPU. Tento růst bohužel mnohonásobně převyšuje ten u operačních pamětí, což je zobrazeno v Obr. 1. I z tohoto důvodu by se měli programátoři naučit, jak efektivně využívat cache paměť. [8]



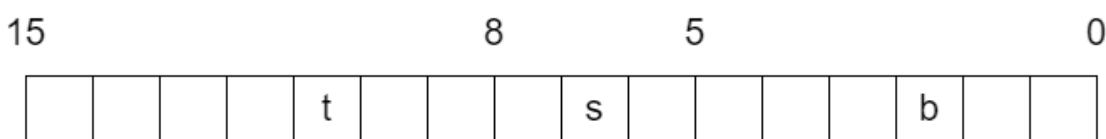
Obr. 1: Vývoj doby trvání přístupu do procesorové a operační paměti. [8]

### 3.1.1 Struktura a adresování

Cache paměti uspořádávají data do sad (set). Každá sada obsahuje jeden nebo více řádků (line). Jeden řádek kromě samotných dat, kterým se také říká blok (block), obsahuje ještě jeden bit  $v$ , který signalizuje validitu záznamu, a také štítek (tag), který se používá při adresování a načítání dat z cache. [7]

K adresování do vyrovnávací paměti se využívá celá adresa dat rozdelená na části. Počet bitů každé části je dán počtem bitů, které počítač používá pro adresování  $m$  (typicky 32 nebo 64 bitů), počtem sad  $S$  a velikostí bloku  $B$  (typicky 64 B). Parametry  $S$  a  $B$  jsou stanoveny výrobcem a každá konfigurace má své výhody i nevýhody. Abychom mohli získat počet bitů použitých pro tag  $t$ , vypočítáme počet bitů pro index sady  $s = \log_2(S)$  a počet bitů pro offset v bloku  $b = \log_2(B)$ . Počet bitů pro tag  $t = m - (s + b)$ . [7]

$$\begin{array}{ll} S = 8 & s = 3 \\ B = 64 & b = 6 \\ m = 16 & t = 16 - (3 + 6) = 7 \end{array}$$



Obr. 2: Příklad rozdělení adresy. [Zdroj vlastní]

Bity  $s$  slouží pro nalezení odpovídající sady v cache paměti. Bity  $t$  slouží pro identifikaci odpovídající cache line v sadě. Bity  $b$  říkají, kde v rámci je bloku začátek dat, o která se žádá. [7]

### 3.1.2 Pojmy

Při popisování funkce vyrovnávacích pamětí je třeba si popsat několik běžně používaných pojmu. *Cache hit* označuje nalezení požadovaných dat v první sousední úrovni směrem dolů v paměťové hierarchii. Při optimalizaci programů je snaha uchovat data, se kterými se operuje, v jakémkoliv úrovni cache na procesoru, jelikož i přístup do L3 je rychlejší než přístup do operační paměti. *Cache miss* označuje absenci požadovaných dat v první sousední úrovni směrem dolů v paměťové hierarchii. Pokud je cache paměť plná, je třeba využít zvolené substituční strategie a nahradit vybranou cache line. Toto může být označeno jako *block eviction*. Existuje několik druhů *cache miss*. V případě, že je vyrovnávací paměť prázdná, při žádosti o data nastává *compulsory miss*. Pokud by velikost vyrovnávací paměti byla příliš malá, nebo bychom se do ní pokoušeli zapisovat data, která musí být uložena na stejné místo, jedná se o *conflict miss*. Jestliže pracujeme obrovskou sadou dat, iterujeme přes ni ve smyčce a tím dojde k vyčerpání kapacity cache, pozorujeme *capacity miss*. Když dochází k opakováným konfliktům na stejném místě v cache paměti, označujeme to jako *thrashing*. Poměr počtu *cache miss* a počtu dotazů na data nám dává *miss rate*. *Hit rate* je vyjádřeno jako  $1 - Miss rate$ . Doba přesunu dat z paměti do CPU se označuje jako *hit time*. Doba čekání na data v případě *cache miss* se jmenuje *miss penalty*. Bavíme-li se o pojmu *write hit*, máme na mysli to, že se data, která chceme aktualizovat, vyskytují v nejbližší cache paměti. Naproti tomu *write miss* označuje absenci dat, která aktualizujeme. [7]

### 3.1.3 Vyrovnávací paměť s přímým mapováním

Direct-mapped cache je typ vyrovnávací paměti, který má v každé sadě právě jednu cache line. Tyto paměti jsou jednoduché na implementaci a používání, ale je zde zvýšené riziko *thrashingu*. [7]

### 3.1.4 Set associative vyrovnávací paměť

Tento typ pamětí se redukuje problém existující v paměti s přímým mapováním pomocí navýšení počtu cache line v rámci jedné sady. *Asociativní* paměť ukládá data jako pole dvojic *klíč-hodnota*. Každou sadu v této paměti si lze představit jako asociativní paměť, jejímž klíčem je spojení bitů  $v$  a  $t$  a hodnotou je obsah bloku. [7]

Při návrhu stupně asociativity je třeba zvolit vhodný kompromis mezi větším a nižším. Vyšší stupeň asociativity je pomalejší a složitější na implementaci. [7]

Cache		
L1 Data	4 x 32 KBytes	8-way
L1 Inst.	4 x 32 KBytes	8-way
Level 2	4 x 256 KBytes	4-way
Level 3	6 MBytes	12-way

Obr. 3: Příklad uspořádání a typu cache pamětí – snímek programu CPU-Z. [Zdroj vlastní]

Obr. 3 je příkladem různých úrovní cache pamětí. V levém sloupci vidíme *počet jader x velikost paměti*. V pravém sloupci je uveden stupeň asociativity. *N-way* nám říká, že každá sada pojme *N* cache line.

### 3.1.5 Plně asociativní vyrovnnávací paměť

Jedná se o asociativní vyrovnnávací paměť s jednou sadou. Tato sada obsahuje všechny cache line. Při adresování je možné vypustit bity *s*, jelikož se vždy pracuje s hodnotou 0. Plně asociativní vyrovnnávací paměť je vhodná pro malé kapacity, jelikož pro větší kapacity by bylo zapotřebí značné množství hardware a zároveň by byla pomalá. [7]

### 3.1.6 Substituční strategie

Pokud nastane *conflict miss*, je třeba umístit žádaná data na vhodné místo. Toto místo může být zvoleno náhodně. Alternativně je možné zvolit sofistikovanější postupu, jako je nahrazení dat, která byla použita nejdále v minulosti (LRU), případně nejméně často využita ve stanoveném časovém okně (LFU). [7]

### 3.1.7 Způsoby zapisování

Na rozdíl od čtení dat, zápis dat je o něco složitější. Jsou-li data, která jsou aktualizována, v cache paměti, mohou být nová data buď rovnou zapsána do cache paměti nižší úrovně (*write-through*) nebo upravena a zapsána je do paměti až v době vyřazení cache line z paměti (*write-back*). Nastane-li *write miss*, může se opět vybrat z více přístupů. Způsob *write-allocate* si nejdříve načte blok z nižší paměťové vrstvy a poté upraví data. Na rozdíl od toho, *no-write-allocate* přímo zapíše data do nižší úrovně. Výše uvedené způsoby lze kombinovat mezi sebou a každá kombinace je vhodná pro jiný cíl. [7]

## 3.2 Operační paměť

Operační paměť na rozdíl od vyrovnávacích pamětí využívá technologie DRAM pro ukládání dat. Jeden bit je uchován pomocí kondenzátoru. Kvůli pokročilému stupni integrace a zároveň náchylnosti na rušení, každý kondenzátor musí být pravidelně dobíjen (refresh). Tento způsob ukládání dat je využit pro operační paměti kvůli možnosti dosažení vyšší kapacity na stejnou plochu čipu v porovnání s SRAM a také kvůli ceně. Čas přístupu je ale delší. [7]

Data na paměťovém čipu jsou uspořádána do dvourozměrného pole označovaného jako superbuňka (supercell) a ta zase obsahuje určitý počet paměťových buněk (cell). Čip je propojen s paměťovým kontrolérem pomocí adresových a datových vodičů, které slouží k vyhledání požadovaných buněk a zápisu nebo čtení dat. Při adresování se nejprve na vodiče zapíše adresa řádku, který se následně celý zkopiuje do interního bufferu řádku, a následně se na stejně vodiče zapíše adresa sloupce, což zapříčiní zapsání dat z požadované paměťové lokace z bufferu řádku na datové vodiče. [7]

Paměťové čipy jsou dále uspořádány do paměťových modulů. Přístup jednotlivým čipům na modulu probíhá paralelně. V případě čtení dat z paměti paměťový kontrolér obdrží adresu, kterou rozdělí podle počtu čipů na modulu pro získání adresy superbuňky. Každý čip zapíše hledaná data zpět na výstup modulu, který data poskládá do správného pořadí a poté je pošle kontroléru. Data jsou pomocí sběrnice následně zaslána do CPU. [7]

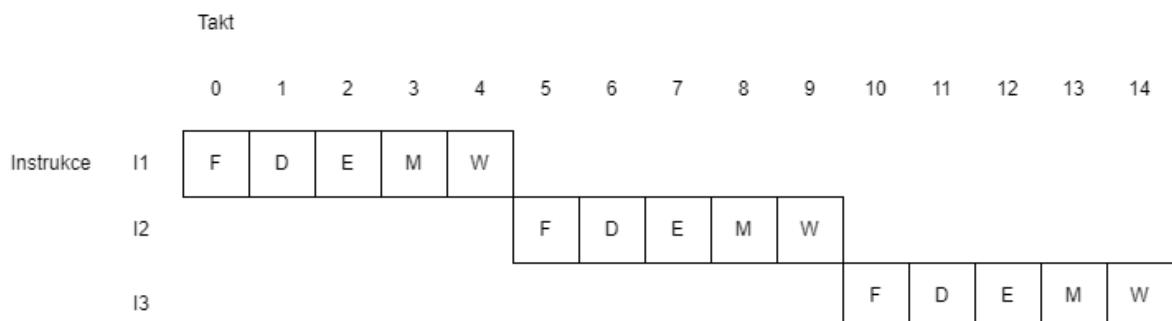
## 3.3 CPU pipelining

### 3.3.1 Fáze instrukčního kanálu

Zpracování jedné instrukce procesorem je rozdeleno do několika fází a bývá popsáno instrukční *pipeline*. První je fáze *fetch*, ve které se načte instrukce z paměti pomocí adresy, která je uchována v registru jménem *program counter*. Následující úroveň *decode* se zabývá načtením operandů. Krok *execute* provede operaci, kterou popisuje instrukce, pomocí aritmeticko-logické jednotky. Může se jednat o matematické nebo logické operace, výpočet adresy, vyhodnocení podmínky nebo směru větvení. Zápis a čtení paměti se děje ve fázi *memory*. Konečný krok *write-back* zapisuje vypočtené výsledky do registrů. Aby mohla být hodnocena vhodnost různých přístupů k provádění instrukcí, jsou využívány pojmy *latency*, což je doba vykonání operace, a *throughput*, který popisuje počet provedených operací za jednotku času. [7]

### 3.3.2 Sekvenční zpracování

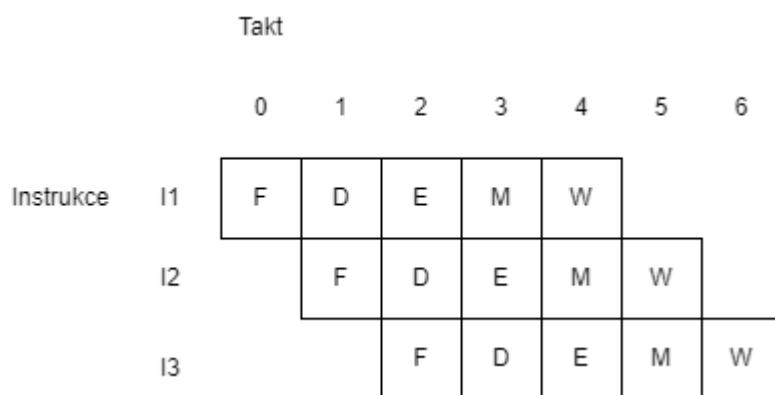
Jestliže se vykonává instrukce jedna po druhé a každá instrukce projde kanálem jako celek, hovoří se o sekvenčním vykonávání.



Obr. 4: Sekvenční vykonávání instrukcí. [Zdroj vlastní]

### 3.3.3 Skalární pipeline

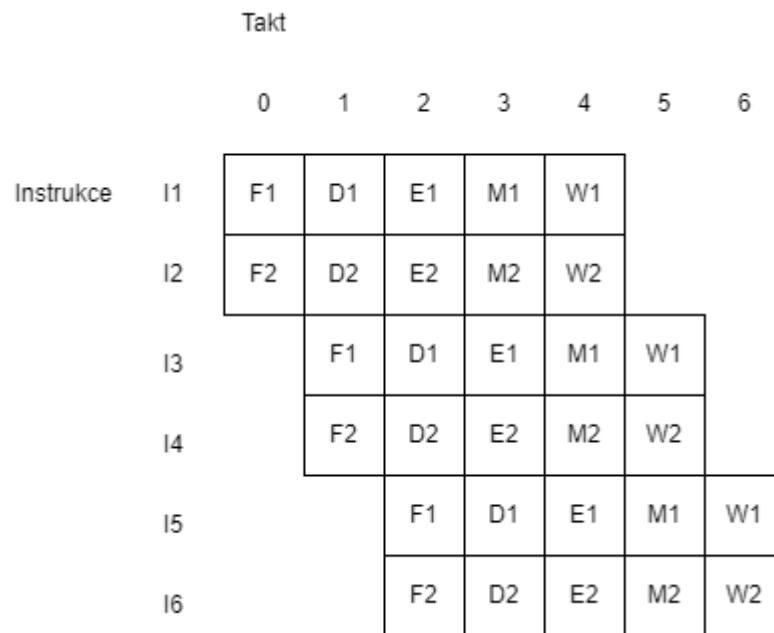
Jiný způsob provádění instrukcí, který by měl lépe využívat dostupný hardware, se nazývá *pipelined*.



Obr. 5: Skalární instrukční pipeline. [Zdroj vlastní]

### 3.3.4 Superskalární pipeline

Vylepšení skalárního *pipelined* přístupu je způsob superskalární. Ten je zdokonalen pomocí přidaného hardware, díky kterému se může během jednoho taktu nacházet více instrukcí v té samé fázi.



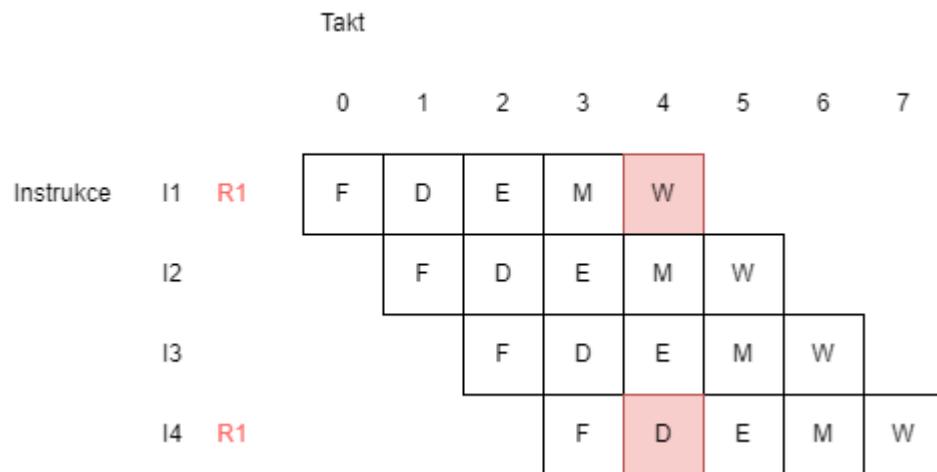
Obr. 6: Superskalární instrukční pipeline. [Zdroj vlastní]

### 3.3.5 Ideální vs reálné řešení

Ačkoliv je *pipelining* vhodným způsobem pro zvýšení počtu vykonaných instrukcí za jednotku času, může mít za následek mírný nárůst latence každé instrukce. Ve dříve popisovaných příkladech *pipeline* jsou uvažovány uniformní časy strávené v každé fázi. Ve skutečnosti však mohou různé kroky trvat různě dlouho a tím pádem způsobovat prodlevy při přechodu mezi kroky. Toto může být částečně vyřešeno zvýšením počtu fází a snížením doby trvání každé fáze, což, jako každé rozhodnutí při návrhu hardware, má taky své nevýhody. [7]

### 3.3.6 Závislosti

Doposud byla uvažována nezávislost dvou po sobě jdoucích instrukcí. V reálném světě k takovému ideálnímu případu nemusí docházet často. Rozlišuje se *datovou závislost*, kde se dvě instrukce odkazují na to samé místo v paměti nebo ten samý registr, a *kontrolní závislost*, ke které dochází při skoku nebo návratu z rutiny. Nesprávná implementace řešení těchto stavů může zapříčinit chyby ve výpočtech, které bývají označovány jako *data a control hazard*. [7]



Obr. 7: Data hazard. [Zdroj vlastní]

*Data hazard* může mít podobu načtení staré hodnoty aktuální instrukcí, zatímco předchozí instrukce už měla hodnotu změnit na novou. Obr. 7 ukazuje vznik chyby ve fázi D u instrukce I4. Jelikož je hodnota R1 aktualizována ve stejném taktu, jako je načítání hodnot operandů instrukce, načtená hodnota není správná. Tento problém lze vyřešit vícero způsoby. Ten nejjednodušší je pozastavení (stalling) pipeline. Spočívá ve vložení takzvaných bublin (bubble) mezi problémové instrukce, aby se posunuly překrývající fáze v rámci cyklu a tím pádem byla načtena správná hodnota. Tento přístup může mít negativní vliv na počet provedených instrukcí za jednotku času. Další způsob se nazývá *data forwarding*. Namísto toho, aby byla nová data zapsána ve fázi W do souboru registrů, aby mohla být v dalším taktu načtena fází D, aktualizovaná hodnota se předá přímo z W na požadované místo. [7]

Pokud ve fázi F nedokážeme jednoznačně určit adresu další instrukce, mluvíme o *control hazard*. Příkladem u instrukce *ret*, po jejímž zavolání by se obecně měl vrátit proud vykonávání na místo předchozího volání funkce, je adresa další instrukce známá až ve fázi W a dochází k pozastavení pipeline. V případě instrukce podmíněného skoku se nemusí čekat na výsledek porovnání, ale lze odhadnout, kterým směrem větve se bude pokračovat. Jen tehdy, pokud se nesprávně zvolená instrukce nachází ve fázích F nebo D je můžeme po detekci bez problémů odstranit z pipeline, protože ještě neměly čas změnit aktuální kontext. [7]

Dalším příkladem je *structure hazard*, který je zapříčiněn omezeným počtem exekučních jednotek. Může nastat, jestliže dvě navzájem se překrývající instrukce bojují o tu samou funkční jednotku. Obr. 7 také zobrazuje možný *structure hazard* v taktu 3, kdy se překrývá fáze F s fází M. V případě, že by obě tyto fáze využívaly stejný paměťový modul a ten

podporoval pouze sekvenční přístupy, nastává *structure hazard*. Tento případ by se dal opět vyřešit vložením bubliny. [8]

### 3.4 Prefetching

Prefetching je jeden ze způsobů tolerance latence načítání dat z paměti. Zabývá se načítáním dat z paměti předtím, než jsou potřeba, například pomocí predikce adres. Tímto způsobem můžeme eliminovat *compulsory cache miss*. V moderních systémech je velikost dat, která jsou přednačtena, stejná jako velikost cache bloku. Tato technika může být provedena hardwarem, na úrovni překladu nebo uživatelsky. Existuje více druhů prefetcherů a ty se mohou lišit tím, který přednačítací algoritmus je použit. Přednačítací algoritmus udává, co se bude načítat. Většina moderních systémů ukládá přednačtená data do cache paměti. [9]

#### 3.4.1 Softwarový prefetching

Prefetching pomocí software je proveden instrukcemi pro přednačítání dat, které poskytuje instrukční architektura. Pracuje s nimi programátor nebo kompilátor a jsou vhodné pro přístupy do paměti, které jsou pravidelné. Je možné vybrat si, do které úrovně cache paměti budou data zapsána. [9]

#### 3.4.2 Hardwarový prefetching

Prefetching pomocí hardware spočívá v monitorování přístupů do paměti a nalezení opakujících se vzorů. Předpovězené adresy jsou generovány automaticky. Na rozdíl od softwarového přístupu, hardwarový způsob může být lépe přizpůsoben systému, který používáme, a také může představovat menší zatížení. Funkce takového prefetcheru může být taková, že při čtení určitého cache bloku z paměti automaticky načte i jeden nebo více dalších, které po něm následují (*next-line prefetcher*). Jestliže je zaznamenáno několik opakujících se instrukcí, které načítají z paměti s pravidelným offsetem od počáteční adresy, může v určitém případě dojít k přednačtení dalších pravděpodobných dat (*stride prefetcher*). [9]

#### 3.4.3 Execution based prefetching

Univerzální způsob, který může být implementován jak hardwarově, tak softwarově, je *execution based prefetching*. Zde probíhá přednačítání dat v jiném vlákně, než ve kterém je prováděn hlavní program. [9]

Ve speciálním vlákně stačí provést ty části kódu, při kterých by jinak nastal *cache miss*. Mezi ně patří výpočet adres, předpovídání větvení a předpovídání hodnoty. Toto vlákno může existovat jak na stejném jádře, jako je vykonávaný kontext, tak na zcela separátním jádře. Rovněž nemusí existovat po celou dobu běhu programu, ale může dynamicky vznikat a zanikat. Pokud to ISA nabízí, může programátor využít poskytované instrukce pro zahájení výpočtu problémového bloku kódu. [10]

#### 3.4.4 Address correlation based prefetching

Tvrzení, že se snáze odhadují pravidelně se posouvající přístupy do paměti, není překvapující. Ne u všech datových struktur, použitých pro práci s daty, je toto možné. Proto existují i různé přístupy k prefetchingu, které se snaží zrychlit i nepravidelné přístupy do paměti. Tento způsob využívá korelační tabulkou pro zaznamenání přímého přístupu do paměti, a zároveň také těch, které následovaly hned po něm. Lze totiž vypozorovat, že některé přístupy mohou nastat s větší pravděpodobností než ostatní. Ke každému odkazu do paměti může být zapamatován jeden nebo více dalších následujících odkazů a ty budou v budoucnu přednačteny. Tento druh prefetchingu ovšem nedokáže eliminovat *compulsory cache miss*. [10]

#### 3.4.5 Content directed prefetching

Při programování v jazycích jako C/C++ je často využíváno práce s ukazateli do paměti, zejména pak u některých datových struktur. Jelikož ukazatel se může odkazovat na jakékoliv místo v paměti, jen těžko by mohl programátor využít předpovídání adres s pravidelným krokem. Právě pro zefektivnění práce s ukazateli řeší tento přístup. Snaží se identifikovat hodnoty v cache bloku, které jsou potencionálními ukazateli, a načíst pro ně odkazovanou hodnotu z paměti do cache. Není zde třeba žádné tabulky pro sledování přístupů v minulosti, ale také je zde možnost zahlcování cache paměti nepotřebnými daty, protože tento algoritmus načte data pro všechny ukazatele v cache bloku. [10]

### 3.5 Lokalita

Pojem lokalita se týká odkazování na data v paměti. Je to vlastnost dobře napsaných programů, které díky práci s ní mohou běžet rychleji, a to díky faktu, že hardware počítače i operační systémy jsou navrženy pro zužitkování lokality. Lze se s ní setkat v různé podobě v každé úrovni paměťové hierarchie. Zásadní myšlenka, která mohla podnítit vznik vyrovnávacích pamětí, byla, že pokud se jednou odkážeme na určité místo v paměti, je zde pravděpodobnost, že se na něj odkážeme vícekrát v blízké budoucnosti. Toto bývá označováno

jako *časová lokalita*. Jiným případem je *prostorová lokalita*, která říká, že pokud je odkázanou na určité místo v paměti, nejspíše se bude odkazovat i na další místa v paměti v blízkosti původního. [7]

### 3.6 Jazyk symbolických adres

V dnešní době masivního rozšíření high-level programovacích jazyků, jako je C#, Java, Python či JavaScript, se snáze zapomene na to, jak hardware počítače vlastně vykonává nás program. Je třeba si uvědomit, že pohodlnost používání těchto jazyků je výhodná zejména pro jeho uživatele. CPU počítače však mnohdy tíží abstrakce, které do návrhu vnáší programátor, čímž i do určité míry zakrývá realitu toho, co se skutečně pod pokličkou děje. Skutečnost je taková, že procesor rozumí pouze konečné sadě instrukcí a umí je vykonat rychle. Nejblíže této úrovni, nebore-li se v potaz strojový kód, je jazyk symbolických adres. Porozumět kódu v podobě jazyku symbolických adres je velmi užitečné a v kombinaci s pochopením architektury počítačového systému se jedná o velmi mocný nástroj při optimalizaci kódu.

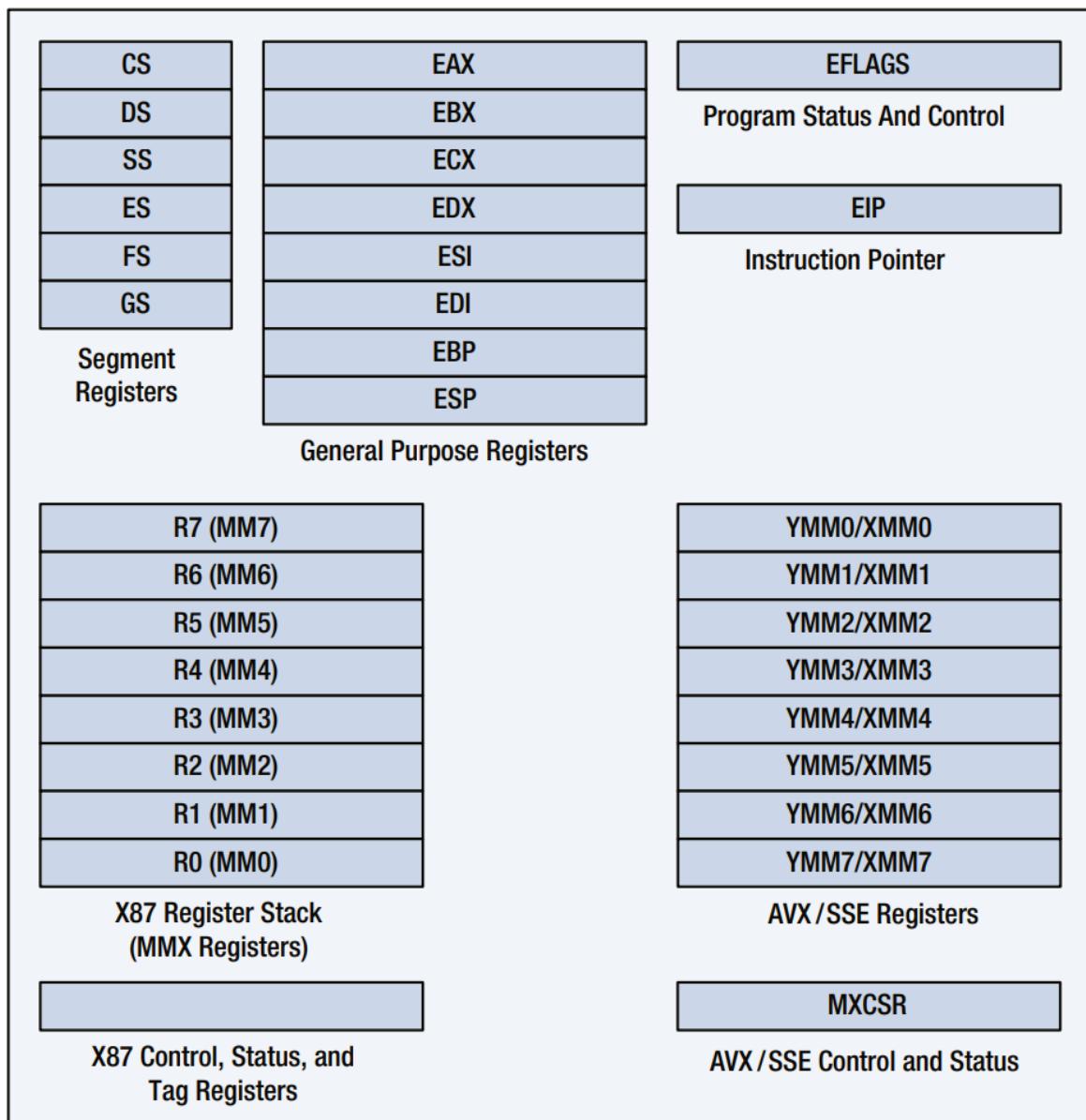
Počítače vykonávají strojový kód, což je sekvence bytů popisující operace pro manipulaci s daty, čtení a zápis dat nebo síťovou komunikaci. Z high-level jazyku vznikne strojový kód pro konkrétní stroj s použitím kompilátoru. V jazyce symbolických adres se programuje pomocí skládání instrukcí za sebe jednu po druhé. Dostupné instrukce jsou definovány abstrakcí nazývanou *instruction set architecture*. Jedny z těch nejvyužívanějších jsou x86-32 a x86-64. [7]

#### 3.6.1 Vnitřní architektura x86-32

Název instrukční sady x86 se odkazuje na procesory společnosti Intel, které byly pojmenovány 8086, 80286, 80386 a 80486. Postupem času docházelo k modernizaci parametrů jako zvětšení velikosti registrů a dalšímu rozšiřování instrukční sady. V dnešní době jsou definovány fundamentální datové typy jako *byte* (8 bitů), *word* (16 bitů), *doubleword* (32 bitů) či *quintword* (80 bitů). Bity každého datového typu jsou uspořádány v pořadí *little-endian*. [11]

Programátorovi je k dispozici několik sad registrů. *Segmentové registry* jsou použity pro definování logického paměťového modelu pro vykonávání programu a ukládání dat. *General-purpose registry* primárně slouží pro vykonávání logických, aritmetických a paměťových operací. Každý z nich může být dále rozdělen pro výpočty s 8 nebo 16-bitovými operandy. Ačkoliv mají sloužit k obecným účelům, v určitých případech slouží k uložení předem

daných výsledků operací. Rovněž existuje konvence, která stanovuje, jaký registr by se měl k čemu použít. *EFLAGS registr* obsahuje kolekci bitů převážně využívanou operačním systémem. Tyto byty indikují výsledky logických a aritmetických operací. *Instrukční ukazatel* udává, která další instrukce se bude vykonávat. Je využit zejména instrukcemi pro volání rutin, skoky nebo návraty z funkce. Obr. 8 ukazuje grafické znázornění. [11]



Obr. 8: Vnitřní architektura x86-32. [11]

### 3.6.2 Adresovací módy

Popisovaná instrukční sada nabízí 4 adresovací módy. U instrukcí, které čtou nebo zapisují do registrů nebo paměti, se vždy vypočítává efektivní adresa. Efektivní adresa může být dána *bázovým registrum*, za který může být dosazen libovolný *general-purpose registr*. Dále se

k adrese přičítá *displacement*, což je číselná hodnota udávající konstantní posuny zakódované v instrukci. K adrese může být dále přičtena hodnota *indexového registru*, za který může být dosazen libovolný *general-purpose registr* kromě ESP. *Indexový registr* se pak ještě může vynásobit *škálovacím faktorem*, který nabývá hodnot 1, 2, 4 a 8. Pro stanovení efektivní adresy lze použít libovolné kombinace těchto hodnot. [11]

### 3.6.3 Instrukce pro přesun dat

Instrukce *mov* je používána pro kopírování dat mezi registry a mezi paměťovými lokacemi. Rovněž se zde nachází instrukce podmíněného přesunu *cmovecc*, která spolupracuje s instrukcí pro porovnávání *cmp*. Instrukce *push* umístí stanovenou hodnotu na zásobník a instrukce *pop* vrátí hodnotu na vrcholu zásobníku. [11]

### 3.6.4 Instrukce pro aritmetické operace.

Aritmetický součet je proveden pomocí instrukce *add*, která dokáže sečít dva operandy. Obdobně funguje instrukce *sub* pro odečítání. Instrukce pro násobení existuje ve variantě se znaménkem a bez znaménka. Verze *mul* pro násobení bez znaménka provede vynásobení dosazené hodnoty s hodnotou registru EAX. Následně uloží výsledek ve dvou částech do registrů EAX a EDX. Obdobně pro operaci dělení *div*. [11]

### 3.6.5 Instrukce pro bitovou rotaci a posuv

Tato skupina provádí operace bitového posuvu a rotace. Rotovat a posouvat lze vlevo nebo vpravo a také lze zvolit, zda je operace logická či aritmetická. [11]

Rozdíl mezi aritmetickým a logickým posuvem je ten, že zatímco u logické rotace vpravo je posloupnost bitů doplněna nulami zleva, u aritmetické rotace je posloupnost bitů doplněna tou hodnotou, kterou nabývá nejvíce významný bit. Kompilátory často v rámci optimalizace substituují operaci dělení bitovým posuvem. [7]

### 3.6.6 Instrukce pro předání kontroly

Instrukce *jmp* provede skok na specifikované návštěstí. Instrukce *call* slouží pro začátek vykonávání rutiny. Nejprve je obsah registru EIP zapsán na zásobník a poté se provede nepodmíněný skok na vybrané místo. Opačným způsobem postupuje instrukce *ret*, která slouží k návratu na místo, za kterým došlo k volání rutiny. Obr. 9 ukazuje příklad programu v jazyce symbolických adres. [11]

```

SignedIsEQ_ proc
    push ebp
    mov ebp,esp

    xor eax,eax
    mov ecx,[ebp+8]
    cmp ecx,[ebp+12]
    sete al

    pop ebp
    ret
SignedIsEQ_ endp

```

Obr. 9: Příklad programu v jazyce symbolických adres. [11]

### 3.7 Překladače

Překlad z high-level jazyka do strojového kódu se odehrává v několika krocích, které bývají označovány jako *kompilační systém*. V první části se odehrává *preprocessing*, kde dochází k odstranění komentářů a ostatních znaků, které nejsou důležité pro běh programu, a také nahrazení direktiv pro vkládání knihoven kódem. V *kompilační* fázi dojde k převodu podoby kódu z high-level jazyka do jazyka symbolických adres. Zde může dojít k optimalizačním krokům, jako je například reorganizace kódu. Následující *assembly* část provádí překlad z jazyka symbolických adres do strojového kódu a zabalení do podoby *relocatable object programu*. Na závěr probíhá *linkování* dalších objektových souborů knihoven s naším souborem. [7]

Jelikož je v rámci práce využit jazyk C/C++, následuje popis překladačů právě tohoto jazyka.

#### 3.7.1 GCC

Zkratka GCC znamená *GNU Compiler Collection* a jedná se o integrovanou distribuci překladačů pro programovací jazyky jako je C, C++ či Fortran. [12]

Překladač se při překladu programu drží dříve popsaného postupu. Uživatel však může specifikovat množství nabízených možností. Běžná komplilace programu s jedním zdrojovým souborem by mohla vypadat následovně.

```
gcc -o hello.exe hello.c
```

Obr. 10: Příklad běžné komplilace programu pomocí GCC. [Zdroj vlastní]

Uživatel může proces komplikace zastavit po každé fázi. Pro zastavení po fázi *preprocessingu* slouží možnost *-E*. Výsledek je zobrazen na standardním výstupu, nebo ho můžeme zapsat do souboru pomocí možnosti *-o*. Pro zastavení po fázi *assembly* slouží možnost *-S*. Do souboru s příponou *.s* je uložena podoba našeho programu v jazyce symbolických adres. Pro zastavení po fázi *linkování* slouží možnost *-c*. [12]

GCC překladač dovoluje specifikovat používaný standard jazyka C/C++ pomocí možnosti *-std*. Mezi podporované hodnoty patří *c90*, *c89*, *c11*, *gnu90*, *c++98*, či *iso9899:2017*. Dále je možné si vybrat, jaké uspořádání bitů struktur a unionů má být použito pomocí *-fssostuct*, za který lze dosadit *big-endian*, *little-endian* nebo *native*. [12]

Mezi důležité možnosti patří výběr agresivity optimalizace kódu. S využitím těchto možností se komplilátor snaží vylepšit výkon programu či snížit počet použitých instrukcí s daní delší doby komplikace. Optimalizace je možné zcela vypnout možností *-O0*. Mezi základní volby se řadí tři úrovně optimalizace, dané možnostmi *-O1*, *-O2* a *-O3*. Tyto příkazy zastřešují sadu dalších možností pro specifické případy. Mezi ně patří například možnost *-finline-functions*, která hodnotí vhodnost každé funkce programu pro *inlining*. *Inlining* nahrazuje kód pro volání funkce prostým vložením těla funkce do kódu. Další je možnost *-auto-inc-dec*, která kombinuje inkrementaci nebo dekrementaci adresy s přístupem do paměti. Zde je třeba myslit na to, že některé možnosti optimalizace jsou dostupné pouze na určitých architekturách. [12]

### 3.7.2 Clang

„Překladač Clang je open-source komplilátor pro rodinu programovacích jazyků příbuzných s C a soustředí se na jejich nejlepší dostupnou implementaci. Staví na LLVM optimalizéru a generátoru kódu pro poskytnutí optimalizace vysoké kvality a generace kódu pro různé cíle.“ [13]

```
clang -o hello.exe hello.c
```

Obr. 11: Příklad běžné komplikace programu pomocí Clang. [Zdroj vlastní]

Clang nabízí rovněž extenzivní počet možností pro úpravu výsledku komplikace. Uživatel si opět volí několik optimalizačních úrovní, jak bylo popsáno dříve. Optimalizační schopnosti jsou dále rozšířeny o *optimalizace vedené profilováním*. Vylepšení spočívá ve sběru dat při prvotní komplikaci a vygenerování tabulky s těmito daty ve formátu podporovaném profilery. Profiler vygeneruje výstup, který popisuje, v jakém místě v kódu je stráveno nejvíce času a

tím pádem místo, kde mají optimalizace smysl. Rovněž mohou být zaznamenány informace o pravděpodobnostech větvení. [13]

### 3.7.3 MSVC

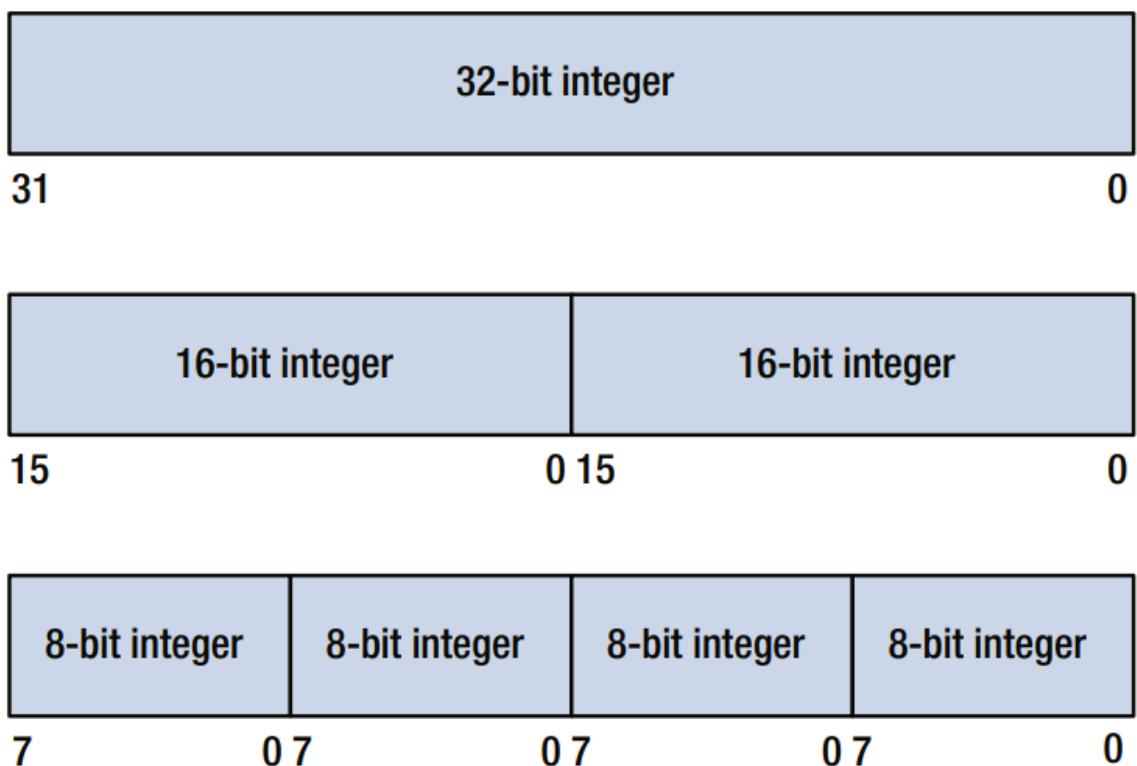
Microsoft Visual C++ je překladač vyvinutý společností Microsoft pro překlad programů psaných v C/C++ na platformě Windows. Překladač generuje soubory v Common Object File formátu a linker zase spustitelné soubory. [14]

```
cl hello.c
```

Obr. 12: Příklad běžné komplikace programu pomocí MSVC. [Zdroj vlastní]

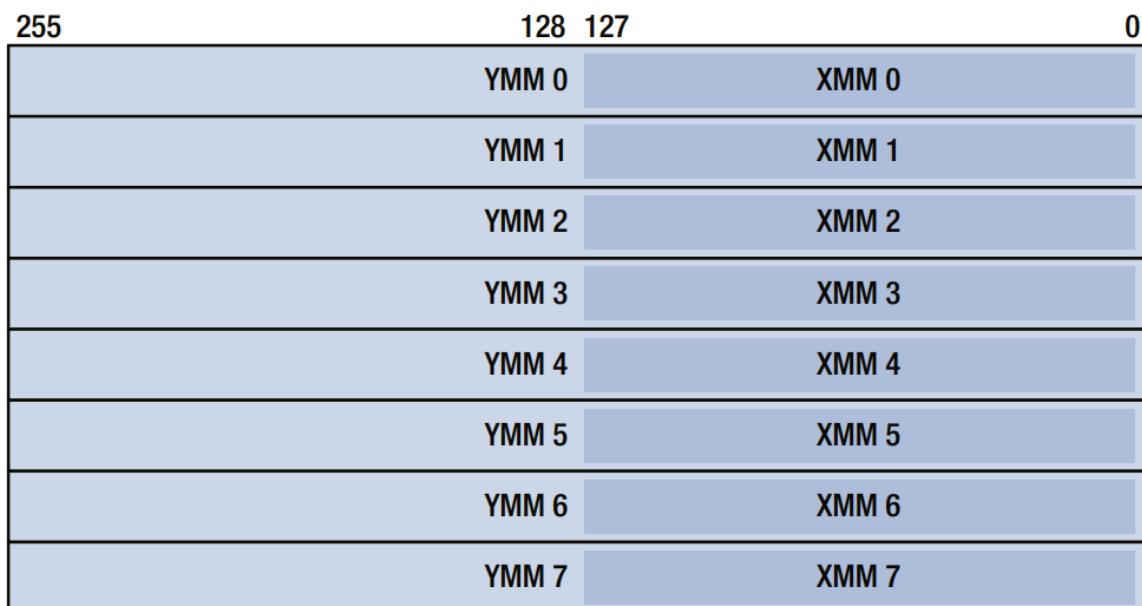
## 3.8 SIMD

SIMD je způsob zpracování dat, kde je stejná operace aplikována na datovou sadu. Dnešní procesory dokáží s daty takto pracovat a programátorům nabízí celou řadu instrukcí pro aritmetické, logické operace a převody mezi typy. V roce 1997 byl do mikroarchitektury P5 od společnosti Intel zařazen výpočetní zdroj jménem MMX, který konceptu SIMD využívá a sloužil původně ke zrychlení operací nad multimedialními daty. [11]



Obr. 13: Možnosti využití registru v SIMD. [11]

Obr. 13 znázorňuje možnosti rozdělení SIMD registru. V případě že je k dispozici 32-bitový registr, lze na něj nahlížet několika způsoby. Může obsahovat jedno 32-bitové číslo, dvě 16-bitová čísla nebo čtyři 8-bitová čísla. Díky tomu lze provést tu samou operaci na více datech paralelně. Technologie MMX operuje s registry o velikosti 64 bitů. Později představená sada SSE rozšiřuje velikost na 128 bitů a v aktuální době nabízí největší velikost sada AVX, AVX2 a AVX-512, kde registry nabývají velikostí 256 a 512 bitů. [11]



Obr. 14: Sada registrů AVX. [11]

Významnou roli u SIMD operací hraje zarovnání operandů v paměti. Data jsou zarovnána v případě, že jejich adresa je dělitelná velikostí jejich datového typu v bytech beze zbytku. Na rozdíl od fundamentálních datových typů, kde není zarovnání vyžadováno, při použití instrukčních sad SSE a AVX zarovnání dat vyžadováno je. I přesto se doporučuje zarovnávat fundamentální typy z důvodu možného zlepšení výkonu při přístupu do paměti. [11]

X86-SSE využívá pro výpočty 8 128-bitových registrů pojmenovaných *XMM0* až *XMM7*. Dále také obsahuje kontrolní a stavový registr *MXCSR*, který obsahuje příznaky po operacích s čísly s plovoucí řádovou tečkou. Jsou zde k dispozici instrukce pro zpracování jak skalárních *single* a *double floating-point* čísel, tak takzvaných složených (*packed*) hodnot. Právě při využití složených hodnot lze využít paralelního zpracování více dat. Předtím je třeba je ovšem vložit do dříve zmíněných registrů. K tomu existují instrukce *movaps* či *movups*, které na rozdíl od MMX umožňují pracovat i s nezarovnanou pamětí. Poté můžeme použít instrukce pro běžné aritmetické a logické operace jako *addps*, *mulps*, *sqrtps* či *maxps*. Navíc máme k dispozici další užitečné operace jako například horizontální součet

sousedících hodnot ve zdrojovém a cílovém registru (*haddps*), změna pozic hodnot v rámci SSE registru pomocí bezprostřední hodnoty (*shufps*) nebo sloučení zdrojového a cílového registru pomocí bitové masky (*blendps*). [11]

```
#include <stdint.h>
#include <stdlib.h>

#define N 8

int16_t* vectorAdd() {
    int16_t* a = aligned_alloc(16, sizeof(int16_t) * N);
    int16_t* b = aligned_alloc(16, sizeof(int16_t) * N);
    int16_t* res = aligned_alloc(16, sizeof(int16_t) * N);

    for (int i = 0; i < N; ++i) {
        res[i] = a[i] + b[i];
    }

    return res;
}

void main() {
    int16_t* res = vectorAdd();
}
```

Obr. 15: Program sečtení dvou polí a uložení do výsledného pole. [Zdroj vlastní]

V mnohých případech dokáže překladač sám vyzporovat, kde by mohl aplikovat SIMD přístup ke zrychlení kódu. Příkladně scítání dvou polí s celými čísly ve smyčce v Obr. 15. Velikost datového typu (16), velikost pole (8) a hodnota zarovnání (16) byly zvoleny záměrně k demonstraci využití SSE. Tyto hodnoty zajistí, že se celé pole s celými čísly vejde do jednoho *XMM* registru a také, že bude zdrojové pole adekvátně zarovnáno. Nyní lze pomocí adekvátních přepínačů u překladače GCC docílit využití operací nad složenými typy.

```
.L2:
    movzx   ecx, WORD PTR [rbx+rdx]
    add     cx, WORD PTR [rbp+0+rdx]
    mov     WORD PTR [rax+rdx], cx
    add     rdx, 2
    cmp     rdx, 16
    jne     .L2
```

Obr. 16: Kód varianty s přepínači *-O1 -msse*. [Zdroj vlastní]

```
movdqa  xmm0, XMMWORD PTR [rbp+0]
paddw   xmm0, XMMWORD PTR [rbx]
movaps  XMMWORD PTR [rax], xmm0
```

Obr. 17: Kód varianty s přepínači *-O2 -msse*. [Zdroj vlastní]

Jak je zřejmé z Obr. 16, úroveň optimalizace 1 ještě nestačí na exploataci SIMD dovedností. Teprve při využití druhé úrovně překladač zjistí, že tělo smyčky lze při využití složeného datového typu provést pomocí těchto 3 instrukcí.

```
.L2:
    movzx   ecx, WORD PTR [rbx+rdx]
    add     cx, WORD PTR [rbp+0+rdx]
    mov     WORD PTR [rax+rdx], cx
    add     rdx, 2
    cmp     rdx, 14
    jne     .L2
```

Obr. 18: Kód varianty s přepínači *-O2 -msse* a lichým počtem prvků v poli. [Zdroj vlastní]

V případě, že počet prvků v poli je lichý, překladač nedokáže využít SIMD přístupu a výsledek je podobný jako v případě *-O1*, což ukazuje Obr. 18.

Ve více komplikovaných příkladech je možné, že překladač nebude schopen převést kód na použití vektorových operací. V takovém případě je možné sáhnout po *wrapper* knihovnách, které poskytují abstrakce, které docílí vynucení SIMD operací, jako například Boost.SIMD. O krok níže k hardware je také možnost využití takzvaných intrinsicských funkcí, které poskytují výrobci procesorů a existuje v nich řada funkcí jazyka C/C++, které se téměř jednou jedné mapují na SIMD instrukce procesoru. Jedna z takových knihoven je *immintrin.h*, která nabízí speciální datové typy a funkce s následující signaturou:

```
_m128i _mm_add_epi16 (_m128i a, _m128i b). [15]
```

Jedná se o funkci, která využívá stejné instrukce jako v předchozím obrázku a slouží pro součet hodnot ve dvou *XMM* registrech, který obsahuje 8 16-bitových hodnot. Finální způsob spočívá v programování přímo pomocí jazyka symbolických adres a využívání požadovaných SIMD instrukcí.

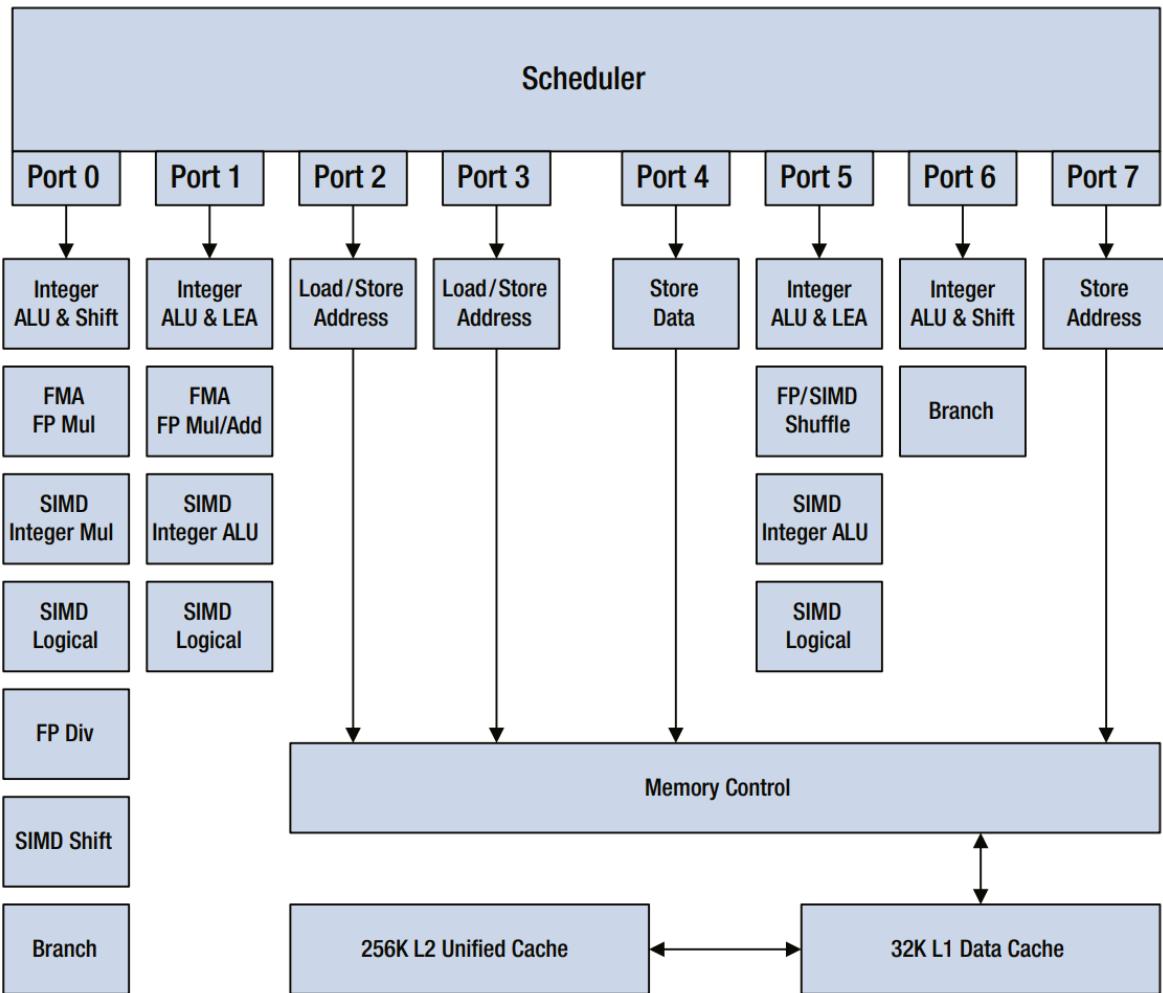
## 3.9 Paralelizace

Paralelizace ve smyslu provádění výpočtů na více jádrech procesoru v jeden okamžik je v současné době hojně využívaná a výrobci procesorů ji postupně rozvíjí z důvodu přiblížení k fyzikálním hranicím integrace na čipu. Oproti vývoji software v minulosti může poskytnout značný posun ve výkonech aplikací, ale také s sebou nesou svá specifika a potíže. Je však třeba si uvědomit, že v rámci výpočetního systému se paralelizace vyskytuje v různých podobách na různých úrovních.

### 3.9.1 Instruction-level parallelismus

Využití paralelismu na úrovni instrukcí je umožněno existencí a využitím pipeline při vykonávání instrukcí. K potenciálnímu zrychlení dochází kvůli faktu, že instrukce mohou být prováděny paralelně. Využití paralelismu na této úrovni může být zprostředkováno jak hardwarově, v podobě dynamického hledání využití paralelismu při běhu programu, tak softwarově, kde je potenciál paralelizace stanoven staticky při psaní kódu. Ukazatelem efektivity pipeline může být *CPI*, který udává počet cyklů potřebných pro provedení instrukce. Pro výpočet se využívá také hodnoty potencionálních pozastavení z důvodu *data* či *control hazard*. Mnoho optimalizací je zprostředkováno překladačem, který se snaží efektivně využít pipeline. [8]

Pro pochopení této úrovně paralelismu je třeba si uvědomit, že skutečný způsob vykonávání instrukcí je dán mikroarchitekturou daného procesoru, který podporuje i více než 100 operací *in-flight* v jednom časovém okamžiku díky využití technologií jako CPU pipelining či *out-of-order execution*. [7]



Obr. 19: Exekuční engine Haswell CPU. [11]

Z důvodu vícenásobného počtu exekučních jednotek, znázorněných v Obr. 19 pro tu samou operaci můžeme docílit souběžného vykonávání více mikro-operací stejného typu.

### 3.9.1.1 Loop unrolling

Jedná se o přístup replikace těla smyčky s cílem zvýšit počet užitečných instrukcí, které skutečně provádí nějaký pro nás významný výpočet, a také lépe usporádat po sobě následující instrukce do pipeline s využitím navzájem nezávislých operací. [8]

Po překladu programu, ve kterém se nachází iterace ve smyčce, do jazyka symbolických adres lze běžně spatřit využití instrukcí podmíněného skoku a dekrementace počítadla pro zprostředkování očekávaného chování. Tyto dvě instrukce však mohou mít za následek po-zastavení pipeline. Několikanásobnou replikací těla smyčky vzniká potenciál vstupu operací z po sobě jdoucích iterací do pipeline, pokud jsou na sobě nezávislé. Tato optimalizace navíc otevírá dveře pro aplikaci dalších optimalizací, jako například přeskládání výrazů. [8]

Při používání tohoto způsobu pro zrychlení běhu programu je třeba uvažovat nad tím, kolikrát by mělo být tělo smyčky replikováno. Od určitého počtu už by mohlo docházet k poklesu výkonu. Rovněž dojde k nárůstu velikosti souboru a následně k potencionálním *cache miss* v instrukční vyrovnávací paměti. Dalším limitujícím faktorem je počet registrů, který musí výt využít. Z důvodu zachování nezávislosti instrukcí musí být mezivýsledky ukládány do odlišných registrů, jejichž počet je omezený. [8]

### 3.9.1.2 Předpovídání větvení

Větvení v programu může mít za následek snížení výkonu z důvodu špatného odhadu cesty. V předchozím případě bylo toto riziko sníženo díky snížení počtu podmíněných skoků. Z důvodu potřeby lepšího odhadování při skocích jsou využívány různé druhy předpovídání. Užitečné informace mohou být zjištěny ze statické analýzy kódu, pro razantní snížení penalizace za špatnou předpověď je však zapotřebí využít také dynamické předvídání. Zkoumání výsledků nedávných větvení je aplikováno *korelačním prediktorem*, který podle posbíraných výsledků rozhodne cestu aktuální větve. Je možné také sledovat celou historii větvení pomocí zápisu do posuvného registru, jehož každý bit určuje, kterým směrem jsme se vydali v odpovídající větvi programu. [8]

Zlepšit výkon při větvení můžeme několika způsoby. První z nich lze aplikovat na podmínky a spočívá v provádění porovnání, jehož výsledek lze snadno odhadnout. Alternativně lze upravit způsob programování a využít přístup podmíněných instrukcí, jako je například *cmove*. Této instrukce předchází porovnání hodnot dvou registrů a přesun dat je proveden pouze v případě, že hodnota zdrojového registru je větší nebo rovna hodnotě cílového registru. Díky tomuto přístupu lze dosáhnout konstantního výkonu nezávisle na složitosti odhadu výsledku porovnání. [7]

### 3.9.2 Data-level parallelismus

Mluví-li se o paralelismu na úrovni dat, myslí se tím především SIMD výpočetní model. Mohou sem být zařazeny výpočty na procesorech s vektorovou architekturou, či GPU. Podnětem pro rozšíření SIMD exekučních jednotek a instrukcí byl nárůst počtu aplikací, které by právě takové zpracování dat mohly využít. Mezi ně lze zařadit práci s multimediálními daty a maticemi. SIMD má potenciál být energeticky úsporné, jelikož jedna instrukce zpracuje více dat najednou. Navíc náročnost hardwarové implementace takovýchto operací není velká. [8]

### 3.9.2.1 *Strip mining*

Jak je zřejmé z Obr. 18, nelze vždy spoléhat na překladač pro využití vektorových instrukcí. Je třeba myslet na to, že tyto vektorové instrukce pracují s registry o určité délce, která je zpravidla mocnina dvou, a také na zarovnání v paměti. Zarovnání je důležité i z toho důvodu, aby se vektor v hodnot paměti nenacházel na dvou různých paměťových stránkách. Délka pole, se kterým programátor pracuje, zřídka odpovídá požadavkům. Navíc ne vždy je možné ji dopředu odhadnout. V takových případech však může programátor vynaložit úsilí, aby překladači práci usnadnil. [8]

Pokud je známa maximální povolenou délku registru pro vektorové operace, mohou být data a smyčky nachystána tak, aby je bylo možné na registry namapovat. Takový postup je označován jako *strip mining*. Pokud se v původním programu vyskytovala jedna smyčka, bude rozdělena na dvě. Hlavní smyčka zpracuje data, kterými lze plně naplnit vektorové registry a vedlejší smyčka, která zpracuje zbytek. [8]

### 3.9.2.2 *Výpočty na GPU*

Společnost Nvidia, zabývající se převážně vývojem grafických čipů, představila programovací jazyk CUDA pro využití paralelního potenciálu svých karet. Programátor zde využívá SIMD principů na masivní úrovni paralelizace, ale musí se také seznámit se specifikami GPU platformy. Programátorům jsou zde k dispozici abstrakce s názvy *thread*, *block* a *grid*. Tyto pojmy se dají přirovnat ke běžně používaným termínům ze všedních programovacích jazyků. *Thread* si můžeme představit jako jednu právě probíhající iteraci smyčky. Ty se dále seskupují do *bloku*, který můžeme přirovnat tělu smyčky. *Bloky* jsou dále uspořádávány do *gridu*, který představuje vektorizovatelnou smyčku. Počty vláken v bloku a bloků v mřížce si při výpočtu stanovuje programátor. Takovéto bloky jsou posílány plánovačem na vícevláknový SIMD procesor pro vykonání. [8]

Dalším způsobem psaní programů pro grafické procesory je OpenACC. Výhoda tohoto přístupu je aplikování stejného přístupu pro psaní programů pro CPU i GPU. Jedná se o „programovací model, který využívá direktiv překladače pro umožnění spuštění kódu na paralelních akcelerátorech.“ [16]

### 3.9.2.3 *Loop-level parallelismus*

Konstrukt označovaný jako smyčka je v mnoha ohledech vhodným kandidátem na paralelizaci na více různých úrovních souběžnosti. To ovšem platí pouze při dodržení určitých

pravidel, které do velké míry diktuje hardware. Značné množství typů závislostí může být odhaleno statickou analýzou kódu pro smyčku, mezi které patří například *loop-carried dependence*. Tato závislost popisuje využití dat vypočítaných v předchozích iteracích iteracemi následujícími. [8]

```
for (i=0; i<100; i=i+1) {
    A[i] = A[i] + B[i]; /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */
}
```

Obr. 20: Ukázka *loop-carried* závislosti. [8]

Příklad v Obr. 20 ukazuje závislost na předchozí iteraci u pole B. První výpočet závisí na hodnotě, která byla vypočtena v předchozí iteraci. Jelikož neexistuje cyklická závislost mezi těmito dvěma výpočty, smyčka může být paralelizována. [8]

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Obr. 21: Přepis smyčky. [8]

Obr. 21 zobrazuje přepis smyčky pro umožnění paralelizace. Jelikož oba výpočty v rámci jedné iterace nejsou závislé na sobě, můžeme změnit jejich pořadí. V nulté iteraci se počítá s hodnotou pole B na pozici nula. Tento výpočet je možné přesunout před smyčku. Překládače kontrolují existující smyčky v programu a nalézají závislosti pomocí stanovení, zda jsou indexy pro přístup k prvkům pole *afinní*. *Afnní* index lze vyjádřit ve formátu  $a * i + b$ , kde  $a$  a  $b$  jsou konstanty, a  $i$  je iterační proměnná. Jestliže mají dvě affinní funkce stejnou hodnotu při různých hodnotách indexu, existuje závislost. Pokud je jiný způsob přístupu do toho samého pole hodnot vyjádřen jako  $c * i + d$ , může být závislost odhalena také pomocí největšího společného dělitele. V případě, že  $GCD(c,a)$  dělí  $(d - b)$ , existuje *loop-carried* závislost. [8]

### 3.9.3 Thread-level parallelismus

Postupem vývoje výpočetní techniky začali vývojáři narážet na fyzikální hranice tehdejších procesorů a také klesající výtěžnost z exploatace ILP. Kvůli nikdy nekončícímu hledání vyššího výpočetního výkonu byly vyvinuty jak procesory obsahující více než jedno

výpočetní jádro (*multicore*), tak systémy obsahující více procesorů (*multiprocessor*). Multi-procesorové systémy jsou charakterizovány úzce svázanými procesory, jejichž koordinace je zajištěna operačním systémem a využívají sdíleného paměťového prostoru. U tohoto přístupu je využíván princip paralelního zpracování ve smyslu spolupráce více vláken na jednom úkolu, nebo existence navzájem nezávislých procesů, což je označováno jako *request-level parallelismus*. [8]

### 3.9.3.1 Amdahlův zákon

Amdahlův zákon popisuje možné zrychlení běhu algoritmu při představení vylepšení v libovolné podobě. Dosažitelné zrychlení je omezeno podílem času, ve kterém může být vylepšení využito. [8]

Zrychlení je vyjádřeno jako:

$$Zrychlení = \frac{Doba\ běhu\ algoritmu\ bez\ vylepšení}{Doba\ běhu\ algoritmu\ s\ vylepšením\ tam,\ kde\ může\ být\ využito} \quad (1)$$

Celkové zrychlení závisí na dvou faktorech. Prvním z nich je podíl času běhu částí s vylepšením a bez vylepšení, označován jako  $Zrychlení_V$ . Dalším z nich je určení, jak velká část programu může toto vylepšení využít, což označíme jako  $Podíl_V$ . Nová doba běhu algoritmu je tedy dána jako:

$$Doba_N = Doba_S * \left( (1 - Podíl_V) + \frac{Podíl_V}{Zrychlení_V} \right), \text{kde} \quad (2)$$

- $Doba_N$  je doba běhu algoritmu s vylepšením
- $Doba_S$  je doba běhu algoritmu bez vylepšení

Celkové zrychlení je vyjádřeno jako:

$$Zrychlení_C = \frac{Doba_S}{Doba_N} = \frac{1}{(1 - Podíl_V) + \frac{Podíl_V}{Zrychlení_V}} \quad (3)$$

Tento zákon vyjadřuje klesající výtěžnost implementace nových vylepšení. Velikost části programu, na kterou může být aplikováno vylepšení, diktuje maximální možné dosažitelné zrychlení. [8]

### 3.9.3.2 Koherence vyrovnávacích pamětí

U *multicore* systému můžeme pozorovat asymetrický přístup do paměti. Načítání dat z vyrovnávací paměti je rychlejší než načítání dat paměti operační. Při využití více jader procesoru k provádění výpočtů můžeme data rozdělit na *privátní* a *sdílená*. *Sdílená* data jsou jednou z forem komunikace mezi jádry. Ukládání takových dat do vyrovnávací paměti jader představuje problém koherence vyrovnávacích pamětí. Koherence vyrovnávacích pamětí zajišťuje, že více procesorů má konzistentní pohled na paměť. Paměťový systém je označen za koherentní v případě, že čtením libovolných dat získáme neaktuálnější data. Pro zajištění této vlastnosti jsou aplikovány dva přístupy. První je označován jako *snooping*, kde každá vyrovnávací paměť, která obsahuje určitý blok paměti sleduje, zda je tento blok sdílen, pomocí čtení sdíleného vysílacího média. Další se jmenuje *directory based*, který soustřeďuje informace o tom, zda jsou bloky paměti sdíleny, do jednoho místa. Při využití *snooping* protokolu, první procesor, který zapíše do paměťového bloku, se stane jeho *vlastníkem*. Tento způsob bývá označován jako *write-invalidate*. Pokud jádra chtějí zapisovat do sdíleného bloku, nebo se jádro pokusí o čtení modifikovaných dat v bloku, může dojít k takzvanému *true sharing miss*. Podobné chování nastává také při *false sharing miss*, kdy zápis jednoho jádra do *cache bloku* způsobí invalidaci dat jádra jiného, aniž by se odkazovala na ta samá data. [8]

### 3.9.3.3 Synchronizační primitiva

Jedním z přístupů pro vynucení sekvenčních zápisů je využití synchronizačních primitiv poskytovaných hardwarem. Toto řešení může mít za následek zvýšení latence operací s vysokou úrovni sdílených dat. Pokud to ISA nabízí, lze využít párových instrukcí *load linked* a *store conditional*. Jestliže je po vykonání načtení změněna hodnota v paměti, odkud načtení proběhlo, pak instrukce pro zápis selže. To samé se stane i v případě změny kontextu procesoru. [8]

```
try:    MOV R3,R4 ;mov exchange value
        LL  R2,0(R1);load linked
        SC  R3,0(R1);store conditional
        BEQZR3,try ;branch store fails
        MOV R4,R2 ;put load value in R4
```

Obr. 22: Příklad atomické výměny hodnot. [8]

Běžně používaným primitivem je zámek. Jelikož je při žádání o zámek možné, že je aktuálně zabraný jiným jádrem, musí se jádro, které zámek žádá, točit ve smyčce a zkoušet žádat opakováně. Proto se tomuto přístupu také říká *spin lock*. Při využití koherenčního protokolu využíváme dvou principů. Prvním z nich je, že kontrola stavu zámku může být prováděna na lokální kopii dat v cache jádra. Druhým je, že jestliže jádro využilo zámku v současnosti, je pravděpodobné, že tak učiní znovu. [8]

```
lockit: LDR2,0(R1)      ;load of lock
        BNEZR2,lockit   ;not available-spin
        DADDUIR2,R0,#1   ;load locked value
        EXCHR2,0(R1)     ;swap
        BNEZR2,lockit   ;branch if lock wasn't 0
```

Obr. 23: Spin lock. [8]

Obr. 23 znázorňuje rutinu spin lock. V případě současné existence tří jader J0, J1 a J2 bude postup vypadat následovně:

J0 vlastní zámek a J1 a J2 ve smyčce kontrolují své lokální kopie sdíleného zámku. Jakmile J0 zapíše hodnotu 1, zámek se uvolní a J0 se stane majitelem zámku, což invaliduje lokální kopie v cache zbylých jader. Obě jádra vygenerují *cache miss* a následně požádají o opětovné načtení zámku do cache, který bude nyní opět sdílený. Jedno z jader je obsluženo jako první, a právě to bude moci zámek zamknout. Toto jádro zapíše 1 do zámku pomocí atomické výměny a jestliže je načtená hodnota rovna 0, zámek může být zamčen aktuálním jádrem, a to může pokračovat do kritické sekce. Druhé jádro opět zapíše 1 do zámku, ale protože mu atomická výměna vrátí hodnotu 1, vrací se na začátek smyčky. [8]

### 3.9.3.4 POSIX vlákna

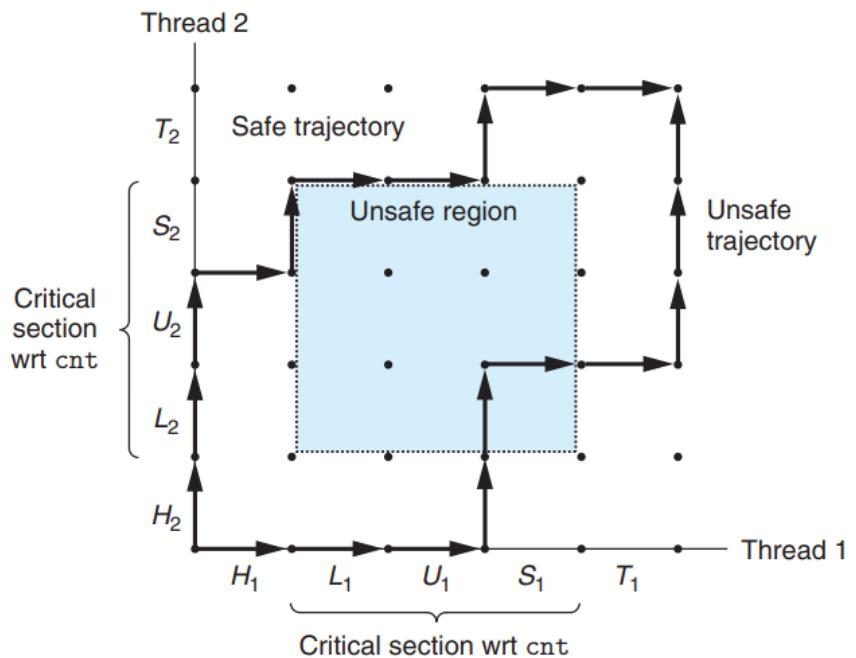
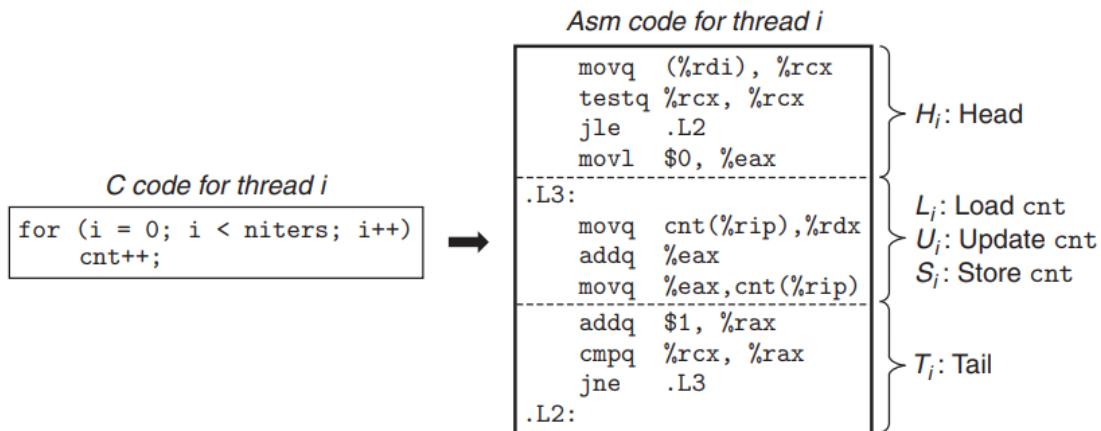
Proces je abstrakce operačního systému pro právě vykonávající se program. Pracuje s iluzí, že mu náleží všechny prostředky procesoru a také paměti. Zároveň se jeví, že jeho instrukce jsou vykonávány jedna po druhé bez přerušení. Právě vykonávající se program má svůj *kontext*, který je mimo jiné určen daty a kódem programu v paměti a hodnotami v CPU registroch. *Vlákno* je logický proud instrukcí vykonávající se v kontextu procesu. Každé vlákno má svůj kontext, identifikátor, zásobník, ukazatel na vrchol zásobníku, stavový registr a registry k obecnému použití. [7]

Mezi vlákny nemusí existovat rodičovská hierarchie. Můžeme je rozdělit na *hlavní* a *vrstevnická*. Každý proces začíná s jedním hlavním vláknem a po čas existence může vytvářet

vrstevnická vlákna. Po dobu jejich existence dochází ke střídání vykonávání těchto vláken.  
[7]

POSIX vlákna poskytují standardní rozhraní pro práci s vlákny v jazyce C. Funkce pro vytvoření vláka `pthread_create` slouží k vytvoření jádra. Můžeme jí jako argumenty poslat ukazatel na funkci, kterou chceme ve vlákně vykonávat, spolu s argumenty pro tuto funkci. Výsledkem je obdržen identifikátor vláka. Všechna existující vlákna mohou být zrušena voláním funkce `pthread_exit` a pro jednotlivá vlákna slouží `pthread_cancel`. Jestliže se od vlákna očekává návratová hodnota, po dokončení jeho běhu je obdržen výsledek pomocí `pthread_join`, které se předá identifikátor vlákna a ukazatel pro data. [7]

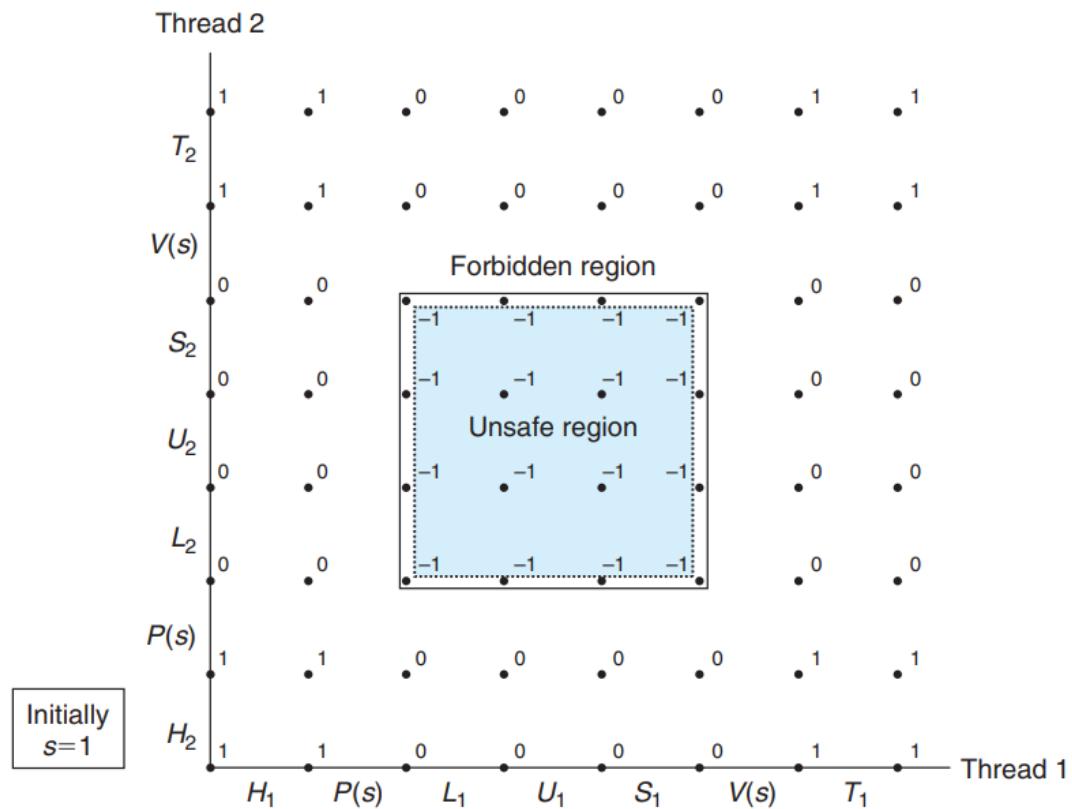
Opět je třeba myslet na potíže nastávající při využití sdílených proměnných ve vláknech. Mezi ně se řadí globální proměnné, statické proměnné a také lokální proměnné předané jako argument pomocí ukazatele. Po dobu současné existence více vláken nelze předpovědět, kdy dojde ke změně kontextu. Pokud více vláken zapisuje do sdílené proměnné, její hodnoty s velkou pravděpodobností nebudou odpovídat očekávání a také budou v různých bězích programu nekonzistentní. V posloupnosti instrukcí programu se nachází skupiny navazujících instrukcí, při jejichž vykonávání by nemělo dojít ke změně kontextu, protože tyto instrukce nějakým způsobem pracují se sdílenou proměnnou. Těmto skupinám se říká *kritická sekce* a můžeme si ji zobrazit v *grafu průběhu* v Obr. 24. [7]



Obr. 24: Graf průběhu. [7]

### 3.9.3.5 Semafor

Jedním z nástrojů synchronizace vláken je *semafor*. Jedná se o globální proměnnou, která obsahuje číselnou hodnotu. Tato hodnota je manipulována funkcemi  $P(s)$  a  $V(s)$ . Jestliže se volá funkci  $P$  a proměnná  $s$  je větší než 0, žádající vlákno ji sníží o 1 a pokračuje. Jinak čeká, až proměnná  $s$  nabude hodnoty 0. Při volání funkce  $V$  se hodnota  $s$  zvyšuje o 1, čímž dochází k odblokování právě jednoho vlákna. V případě, že je iniciální hodnota  $s$  rovna 1, mluví se o *binárním semaforu*, či *mutexu*. Využití takového synchronizačního objektu lze promítнуть do grafu průběhu v Obr. 25. [7]



Obr. 25: Graf průběhu s promítnutými hodnotami binárního semaforu. [7]

Tuto synchronizační strukturu lze také využít pro signalizaci dostupnosti sdílených prostředků. Toho se docílí inicializací semaforu na hodnotu větší než 1. Toto lze ilustrovat na příkladu *producent-konzument*, kde mimo výlučný přístup ke sdílenému datovému bufferu je třeba také zajistit uspořádaný přístup. Proto se zde ještě musí použít další dva semafory. Jeden semafor pro počítání volných míst v bufferu a jeden pro počítání položek umístěných v bufferu. Pokud chce producent přidat položku do bufferu, musí jako první zkontolovat semafor signalizující počet volných míst. Jakmile se uvolní místo, musí zkontolovat semafor pro přístup ke sdílenému bufferu. Poté může položku přidat a signalizovat odemčení semaforu bufferu a také navýšení počtu položek v bufferu pomocí druhého semaforu. Konzument na rozdíl od toho jako první kontroluje semafor pro počet položek v bufferu a poté inkrementuje semafor pro volná místa. [7]

## 4 MĚŘENÍ VÝKONU APLIKACÍ

Bude-li se hovořit o výkonu počítačového programu, různí lidé si pod tímto pojmem mohou představit odlišné metriky. Významná metrika označovaná jako *doba vykonávání* (*execution time*) udává čas mezi začátkem a koncem události. Doba vykonávání programu je spolehlivým ukazatelem výkonu a používání jiných metrik může vést k zavádějícím závěrům při optimalizacích. [8]

Na úrovni procesoru sledujeme časové jednotky pojmenované *cykly*, které se využívají k řízení a synchronizování operací. Délka cyklu může být odvozena z taktu procesoru, který v současnosti nabývá hodnot větších než 3 GHz na běžně dostupných procesorech. Je-li znám počet cyklů pro provedení celého programu, zjistí se *procesorový čas*. [8]

$$\text{Procesorový čas} = \frac{\text{Počet cyklů programu}}{\text{Takt procesoru}} \quad (4)$$

V případě, že je znám počet instrukcí v programu, zjistí se průměrný počet cyklů na instrukci, označováno jako CPI. [8]

$$CPI = \frac{\text{Počet cyklů programu}}{\text{Počet instrukcí programu}} \quad (5)$$

### 4.1 Benchmarking

Návrháři výpočetního hardware využívají zátěžové testy, zvané *benchmark*, pro posouzení vhodnosti navrženého hardware pro určité operace. Benchmark má podobu krátkého programu, který obsahuje malou část programu z reálné aplikace, či většího programu imitujícího chování reálné aplikace. Společnosti, jako například SPEC, se soustředí na vývoj benchmarkových sad, které by měly co nejlépe a nejobjektivněji posoudit výkon hardware. [8]

#### 4.1.1 Microbenchmark

Microbenchmark měří výkon malé části programu. Nemohou být použity pro posouzení celkového výkonu aplikace, ale jsou vhodné pro posouzení latence operací a propustnosti. [17]

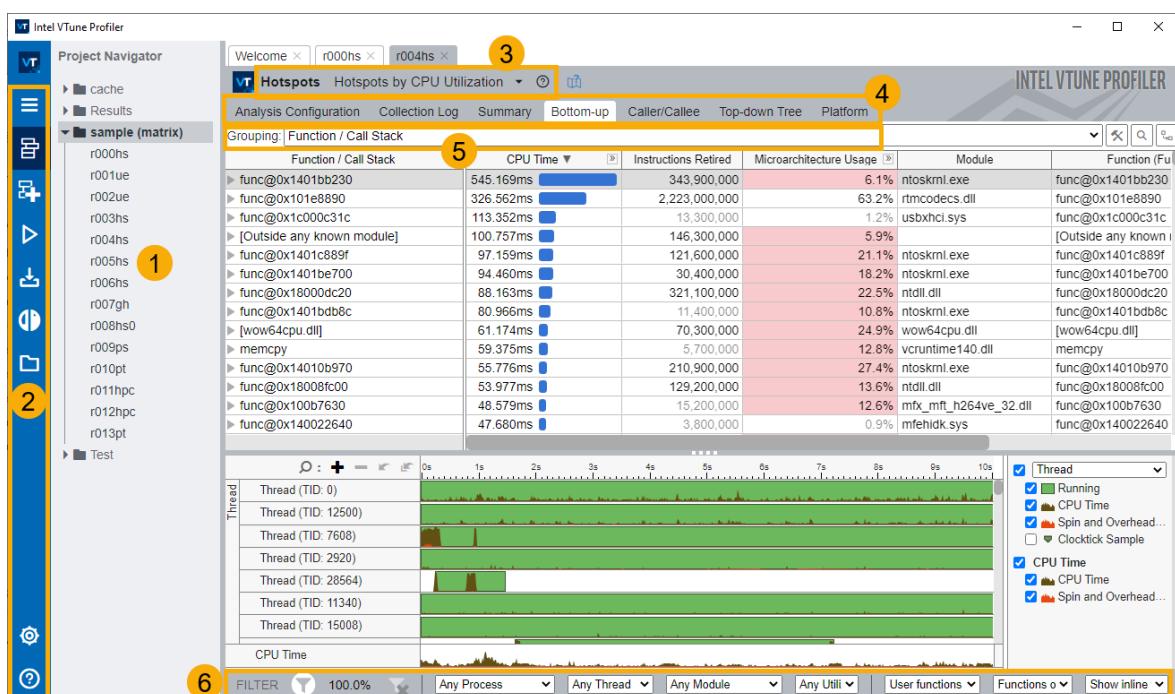
V rámci této práce je využita knihovna Google Benchmark, která slouží k ohodnocení výkonu funkcí v programu, podobně jako u jednotkových testů. Jedná se o rozšířenou knihovnu pro provádění měření a pro demonstraci výkonu ukázkových příkladů je dostačující.

## 4.2 Profilování

Optimalizace se mimo jiné zabývá snižováním doby běhu programu. Aby se dalo zjistit, proč program neběží tak rychle, jak by mohl, je využíváno profilování. K tomu slouží nástroje zvané *profilery*, které umožňují měřit určité metriky a díky nim lze pochopit výkonnostní charakteristiky programu. Tyto nástroje se využijí jak pro měření výkonu, tak při hledání kritických míst v programu při optimalizaci. Významnou částí těchto nástrojů je možnost vizualizace průběhu programu. [18]

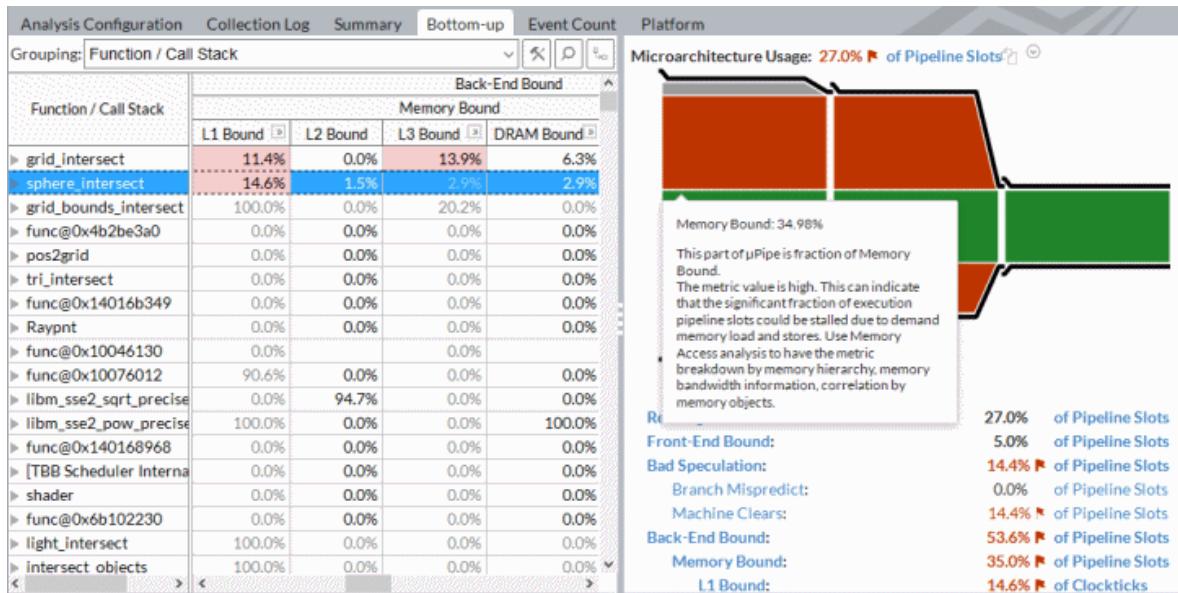
### 4.2.1 Vzorkovací profilery

Vzorkovací profilery fungují na principu periodického čtení informací ze zásobníku volání. Frekvence přerušování může být dána uživatelem. Tyto nástroje pracují s předpokladem, že pokud je doba běhu funkce delší než doba běhu ostatních funkcí, při načtení dat profilerem se budeme v této funkci nacházet častěji než v ostatních funkcích. [18]



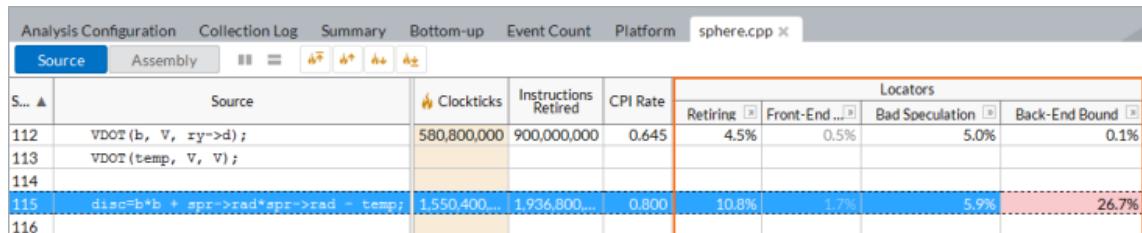
Obr. 26: Grafické prostředí programu Intel VTune. [19]

Profilovací nástroj Intel VTune nám nabízí množství funkcí pro analýzu různých metrik. V adresárové struktuře Obr. 26 u čísla 1 se nachází záznamy jednotlivých profilovacích relací. Aplikace nabízí různé druhy profilovacích běhů, které mohou být dopodrobna zkoumány přepínáním mezi záložkami označenými u čísla 4. V comboboxu níže lze zvolit předem nastavené skupiny dat, které chceme sledovat, a také granularitu těchto skupin pro zjištění různých pro nás významných informací. [19]



Obr. 27: Okno s μpipe zobrazením. [19]

Jednou z metod analýz je prozkoumání využití mikroarchitektury procesoru. Aplikace využívá metrik získaných na základě událostí z PMU. Hodnocení je dáno na základě sledování využití pipeline procesoru. Tuto pipeline lze rozdělit na *front-end*, ve kterém se odehrává načítání a dekódování instrukcí, a *back-end*, který popisuje samotné vykonávání instrukcí za použití funkčních jednotek procesoru. Za předpokladu, že je známo, že front-end v každém cyklu může vygenerovat až 4 operace, lze porovnat naměřená data s daty ideálními. Jestliže volná místa v pipeline nebyla zcela naplněna, protože například nedošlo ke včasnému načtení instrukcí, je tato skutečnost označena jako *front-end bound*. Pokud nemohl back-end přijmout více operací stejného typu, mluvíme o *back-end bound*. V barevném grafu na Obr. 27 aplikace ukazuje, jak dobře nebo špatně je pipeline využita a v jaké části je problém. Je zobrazeno, jaké procento instrukcí bylo načteno a provedeno (*retired*), načteno a zahozeno (*bad speculation*) a také front-end a back-end bound. Tyto kategorie jsou popsány *top-down* modelem a každá z nich má stanovenou své limitní procento, které je při překročení označeno a může pomoci identifikovat potencionální problém. Alternativní pohled *bottom-up* poskytuje výčet identifikovaných objektů ze zásobníku volání, jako třeba programové funkce nebo moduly, a uvádí dostupné naměřené metriky. Opět jsou zvýrazněny překročené hranice předem nastavených limitů. Odsud lze identifikovat i konkrétní řádek kódu, který má na svědomí zpomalení. [19]



Obr. 28: Pohled source code analysis. [19]

Aplikace pro získávání profilovacích dat využívá jednotek pro monitorování výkonu (*PMU*), což jsou hardwarové jednotky nacházející se na čipu a mají za úkol sledovat počty určitých událostí, jako je třeba počet cache-miss. Tyto jednotky spouštějí řadu událostí, které lze odposlouchávat. Aplikace označuje funkce, které zabírají nejvíce procesorového času, jako *hotspots*, na jejichž optimalizaci bychom se měli zaměřit. Pro detailnější identifikaci problémů lze využít dalších typů analýzy. Dále se dají sledovat *memory bound* potíže, způsobené například cache-miss, nebo *core bound*, což popisuje suboptimální využití exekučních jednotek procesoru. [20]

#### 4.2.2 Instrumentační profilery

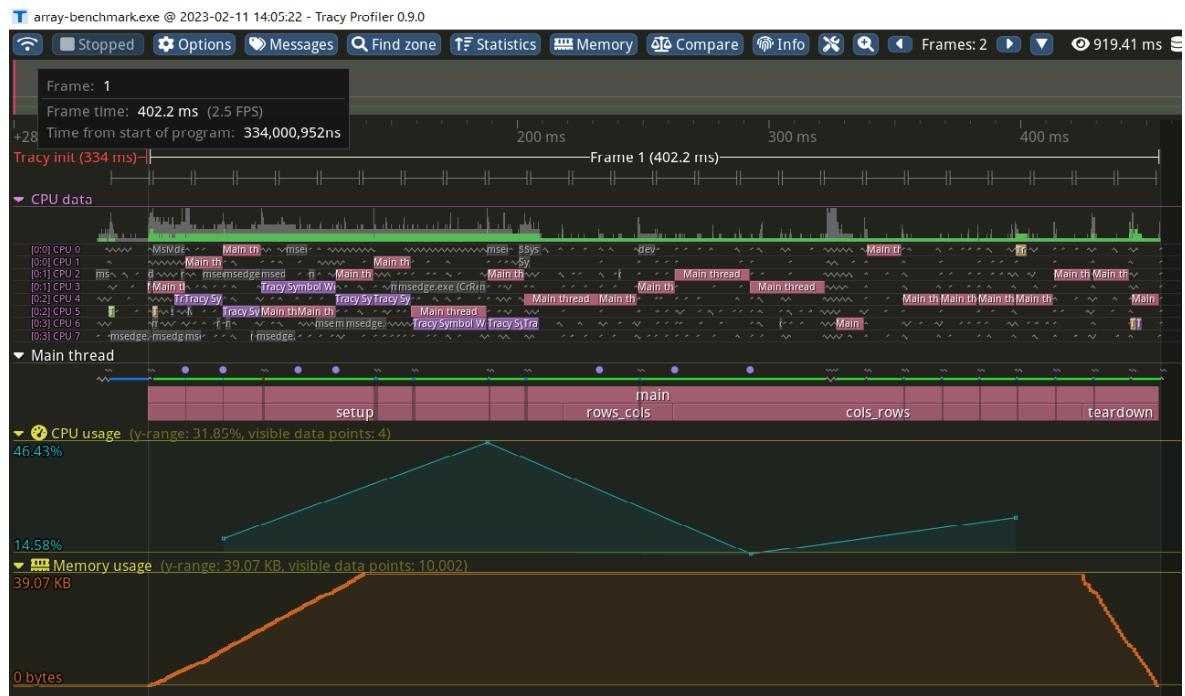
Instrumentační profilery využívají speciálních částí kódu, často v podobě maker, pro vlastní označení sekcí kódu. Na rozdíl od vzorkovacích profilerů jsou tyto nástroje přesnější, protože nepracují s průměrnými výsledky. Jelikož je tato metoda v rukou uživatele, může profileru mimo jiné poskytnout další data, která považuje za významná ke sledování. [18]

Zástupcem této kategorie profilovacích nástrojů je program Tracy. Jedná se o hybridní snímkový a vzorkovací profiler s nanosekundovým rozlišením. Na rozdíl od ostatních instrumentačních nástrojů, dokáže také poskytnout statistické informace na základě dat ze zásobníku volání, podobně jako vzorkovací profilery. Tento nástroj jen velmi málo zatěžuje profilovaný program voláním svých funkcí, protože zaznamenání profilovací události trvá jenom pár nanosekund. Tracy považuje *snímek (frame)* jako základní pracovní jednotku při profilování. Pojem snímek je použit proto, že se od programu očekává, že bude použit pro profilování výkonu her. Snímky však mohou být aplikovány univerzálně a jejich použití navíc není vynuceno. Program dovoluje uživateli používat jak rozhraní v příkazové řádce, tak grafické rozhraní. V obou způsobech použití komunikuje profilovaná aplikace s profilovacím nástrojem prostřednictvím síťové komunikace. Aplikaci lze sledovat v reálném čase, také po jejím skončení, nebo může uživatel do aplikace načíst dříve zaznamenané profilovací logy. [21]

Knihovna Tracy poskytuje programátorovi množství způsobů, jak zaznamenat důležitá aplikaciční data. Funkce *TracyMessageL* bere jako argument textový řetězec, který následně zobrazí v profileru v odpovídajícím čase. Zprávy je možné barevně odlišit. Snímek je označen pomocí *FrameMark*. K volání by mělo dojít ihned po vyrenderování snímku. Mimo funkce užitečné pro profilování her, existuje také příkaz *ZoneScoped*, který sleduje danou zónu. Tento příkaz funguje pomocí sledování speciální profilovací proměnné umístěné na zásobníku. Běžné použití je na začátku vykonávání funkcí, které chceme sledovat, ale příkaz může být umístěn také do smyček. Pomocí *TracyPlot* můžeme profilovací aplikaci posílat data a sledovat jejich postupný vývoj v čase. Rovněž dokážeme sledovat využití paměti pomocí funkce *TracyAlloc* a *TracyFree*. Těmto funkcím je poskytována proměnná typu ukazatel. Při používání tohoto mechanismu je třeba myslet na to, kdy ukazatel předat. Profilovací funkci zaznamenávající alokaci je třeba zavolat po alokaci paměti a uvolňovací funkci zase před samotným uvolněním paměti. Dále existují funkce pro profilování aplikace využívající technologií OpenGL, Vulkan, Direct3D 11, Direct3D 12 a OpenCL. Výše uvedené příkazy mohou být obohaceny o data ze zásobníku volání v případě přidání písmene *S* za název funkce. Pak je třeba specifikovat hloubku, která se má sledovat. Jestliže je profilovací aplikace spuštěna pod zvýšenými právy, dochází k automatickému sledování rozšířených informací. Sem patří například využití a topologie CPU, změny kontextu, vzorkování zásobníku volání, sledování uspaných vláken, čtení hardwareových PMU nebo zobrazení programu v podobě jazyku symbolických adres. [21]

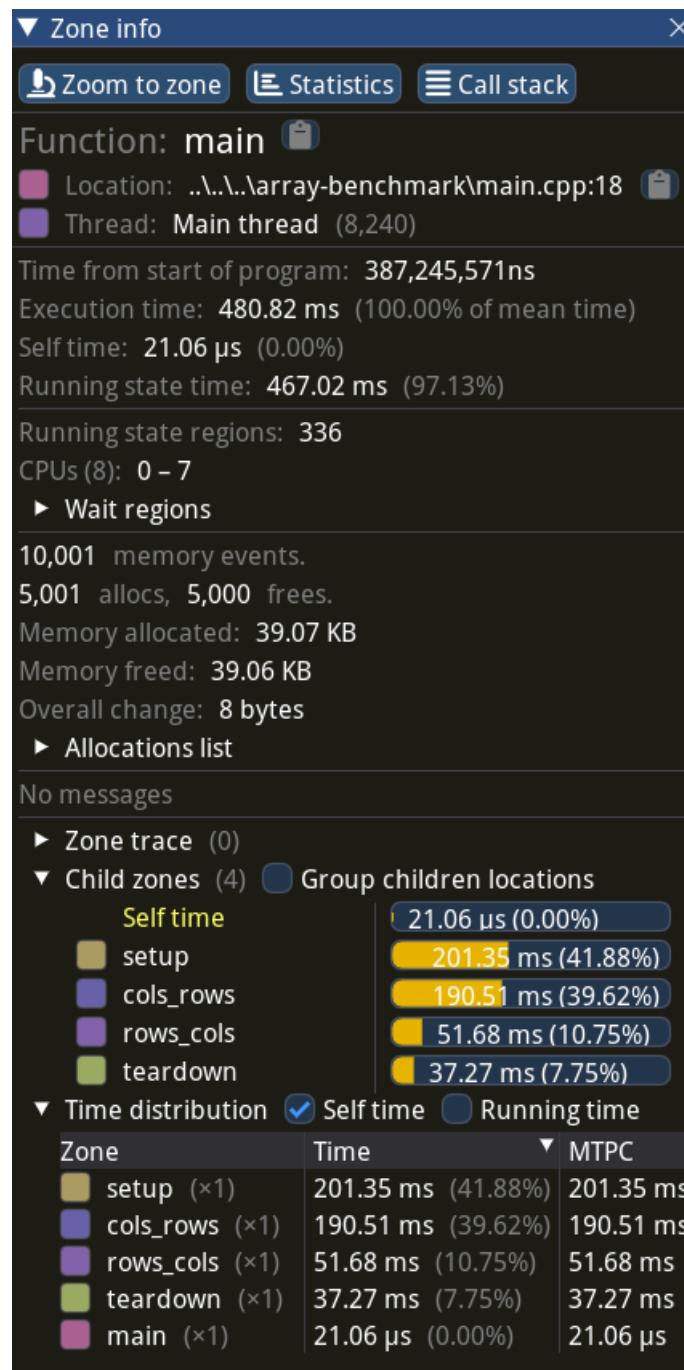
```
D:\Projekty\diplomka\Practice\libs\tracy\capture\build\win32\x64\Release>capture.exe -a 127.0.0.1 -o arrayTrace.tracy
Connecting to 127.0.0.1:8086...
Queue delay: 113 ns
Timer resolution: 12 ns
  18.10 Mbps /  8.4% = 215.97 Mbps | Tx: 23.03 MB | 154.74 MB | 727.22 ms
Frames: 2
Time span: 727.22 ms
Zones: 5
Elapsed time: 13.53 s
Saving trace... done!
Trace size 62.74 MB (47.49% ratio)
```

Obr. 29: Použití programu Tracy z příkazové řádky. [Zdroj vlastní]



Obr. 30: Grafické prostředí programu Tracy. [Zdroj vlastní]

Grafické rozhraní programu Tracy nabízí velké množství funkcí a také získaných informací podaných v různé formě. Bezprostředně pod horní lištou se nachází timeline se zaznamenáými snímky. V tomto případě je zde pouze jeden, který obsahuje všechna data. Jelikož je program spuštěn s administrátorskými právy, lze vidět rozšířené informace, jako například využití jednotlivých vláken a jader procesoru. Hlavní vlákno profilované aplikace je rozděleno na několik částí. Každá odpovídá funkci jazyka C++ a zastřešuje je funkce *main*. V grafu jsou také znázorněny časové okamžiky, ve kterých došlo ke čtení zásobníku volání. Pod tímto oknem se nachází průběh využití CPU a operační paměti.



Obr. 31: Informace o zóně. [Zdroj vlastní]

Po kliknutí na část *main* v grafu jsou prezentovány zónové informace, které zobrazují dodatečná detailní data.

```
62     void rows_cols()
63     @6 {
64     @5     ZoneScopes(5);
65     @1     int sum = 0;
66     @7     for (int i = 0; i < ROWS; i++)
67     {
68     @7         for (int j = 0; j < COLS; j++)
69     {
70     @9             sum += arr[i][j];
71     @1         }
72     @1     }
73     @8 }
```

Obr. 32: Pohled na funkci ze zdrojového souboru. [Zdroj vlastní]

Užitečná funkce tohoto programu spočívá ve sledování zásobníku volání a následné zobrazení samotného zápisu programu, který se vykonává. Jsou nabízeny dva pohledy. Jeden je načten ze zdrojového souboru a zobrazen, a druhý ve formě jazyka symbolických adres. Tento pohled mimo jiné také poskytuje informace o využití registrů a závislostech následujících instrukcí, nebo také informace o vlastnostech využitych instrukcí pro danou mikroarchitekturu. V případě použití podporovaného operačního systému dokáže program také zaznamenat informace z PMU a zobrazit je k odpovídající instrukci. Díky tomu lze přímo vidět *hotspot* v aplikaci a zaměřit se na optimalizace tam, kde dávají smysl. Rovněž dokážeme identifikovat závislosti mezi instrukcemi. Po kliknutí na vybranou instrukci se zobrazí názvy používaných registrů a také je naznačeno, do kterého se zapisuje a ze kterého se čte.

```

+0      push    rdi
+2      sub     rsp, 0x40
+6      lea     rdi, [rsp + 0x20]
+11     mov     ecx, 8
+16     mov     eax, 0xffffffff
+21     rep stosd dword ptr [rdi], eax
+23     mov     r9b, 1
+26     mov     r8d, 5
+32     lea     rdx, [rip + 0x40b71]
+39     lea     rcx, [rsp + 0x24]
+44     <call   0x7ff72dbc1d16
+49     mov     dword ptr [rsp + 0x34], 0
+57     mov     dword ptr [rsp + 0x38], 0
+65     jmp    .L1 -> [rows_cols]
+67     mov     eax, dword ptr [rsp + 0x38] ; .L0
+71     inc     eax
+73     mov     dword ptr [rsp + 0x38], eax
+77     >cmp   dword ptr [rsp + 0x38], 0x1388 ; .L1
+85     jge    .L5 -> [rows_cols]
+87     mov     dword ptr [rsp + 0x3c], 0
+95     jmp    .L3 -> [rows_cols]
+97     >mov   eax, dword ptr [rsp + 0x3c] ; .L2
+101    inc     eax
+103    mov     dword ptr [rsp + 0x3c], eax
+107    >cmp   dword ptr [rsp + 0x3c], 0x1388 ; .L3
+115    jge    .L4 -> [rows_cols]
+117    movsxd rax, dword ptr [rsp + 0x38]
+122    movsxd rcx, dword ptr [rsp + 0x3c]
+127    mov     rdx, qword ptr [rip + 0x528ea]
+134    mov     rax, qword ptr [rdx + rax*8]
+138    mov     eax, dword ptr [rax + rcx*4]
+141    mov     ecx, dword ptr [rsp + 0x34]
+145    add     ecx, eax
+147    mov     eax, ecx
+149    mov     dword ptr [rsp + 0x34], eax
+153    jmp    .L2 -> [rows_cols]
+155    >jmp   .L0 ; .L4 -> [rows_cols]
+157    >lea   rcx, [rsp + 0x24] ; .L5
+162    <call   0x7ff72dbc1e56
+167    mov     rcx, rsp
+170    lea     rdx, [rip + 0x3fa9f]
+177    <call   0x7ff72dbc1a96
+182    add     rsp, 0x40
+186    pop     rdi
+187    <ret

```

Obr. 33: Pohled na program v podobě jazyka symbolických adres. [Zdroj vlastní]

## 5 POUŽITÉ TECHNOLOGIE

### 5.1 Programovací jazyk C++

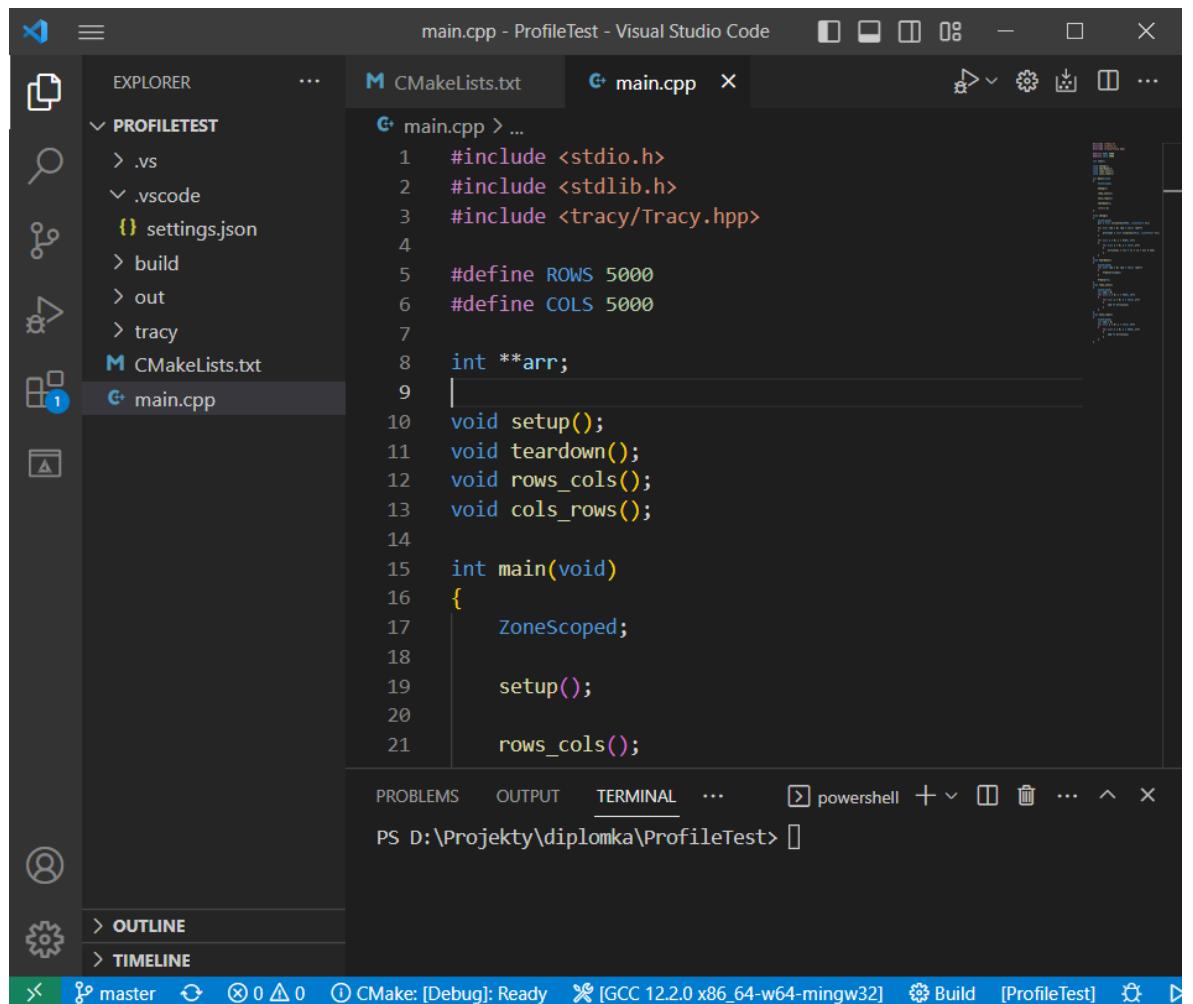
„C++ je programovací jazyk pro obecné použití, který nabízí přímý a účinný hardwarový model a je vybaven pro definici odlehčených abstrakcí“. [22]

Tento jazyk může být využit k programování jak mikrokontrolerů, tak obrovských distribuovaných aplikací. Tvorce tohoto jazyka zdůrazňuje jeden z principů, kterého se držel. Kromě C++ by se mezi něj a hardware neměl vejít další jazyk. Z toho důvodu je tento jazyk oblíbený pro programování systémů, protože dokáže přímo manipulovat s hardwarovými zdroji. Programátor zde může využít několik různých programovacích přístupů. *Procedurální* přístup se zaměřuje na zpracování a návrh vhodných datových struktur. Pro jeho využití slouží vestavěné datové typy jazyka, operátory, funkce a datové struktury. *Objektově orientovaný* přístup je podporován díky existenci tříd a možnostem dědičnosti a zapouzdření. *Obecný* přístup se soustředí na implementaci a použití obecných algoritmů. Pro zobecnění lze v C++ využít šablon. Pro vykonání programu napsaného v jazyku C++ se používá převod pomocí komplikace. Jeden z důvodů výběru tohoto jazyku pro diplomovou práci je existence více překladačů, čímž vzniká možnost porovnání vhodnosti každého z nich pro různé úkony. Tento jazyk je *staticky typovaný*, což znamená, že typ každé entity musí být známý komplátorem v okamžiku používání. Datové typy odpovídají fundamentálním datovým typům využívaných počítačem. Každý datový typ má předem danou velikost, která je dána hardwarom. Patří mezi ně například *bool*, pro uchování pravdivostní hodnoty, *char*, pro zaznamenání znaku, *int*, pro kladná i záporná celá čísla, a *double*, pro čísla s desetinnou čárkou. Datové typy se využívají při *deklaraci* proměnných nebo funkcí. Deklarace představuje jméno do programu. Na rozdíl od běžné definice pojmu objekt, jazyk C++ jako *objekt* považuje kus paměti, která obsahuje hodnotu a má typ. *Proměnná* je poté pojmenovaný objekt. Pokud je třeba s proměnnými pracovat, zpravidla se k tomu využijí *funkce*. Ta specifikuje, jak má být operace provedena. Při její deklaraci je třeba uvést návratový typ, jméno a případně argumenty této funkce, každý se svým datovým typem. Deklarovaná funkce sama o sobě nemá žádné chování. To je třeba specifikovat v *definici* funkce. [22]

Jazyk C++ byl zvolen k implementaci z důvodu rozšířenosti použití v praxi a možnosti relativně jednoduše uvažovat nad převodem kódu do jazyka symbolických adres. Rovněž tento jazyk můžeme překládat více překladači a díky tomu porovnat vhodnost každého z nich pro různé úlohy.

## 5.2 Visual Studio Code

Jako primární vývojové prostředí bylo zvoleno Visual Studio Code. Jedná se o odlehčený a výkonný editor zdrojového kódu dostupný v podobě desktopové aplikace. Je dostupný pro Windows, Linux a macOS. Obsahuje vestavěnou podporu jazyků JavaScript, TypeScript a runtime prostředí Node.js. Množství dalších jazyků je podporováno v podobě rozšíření. [23]



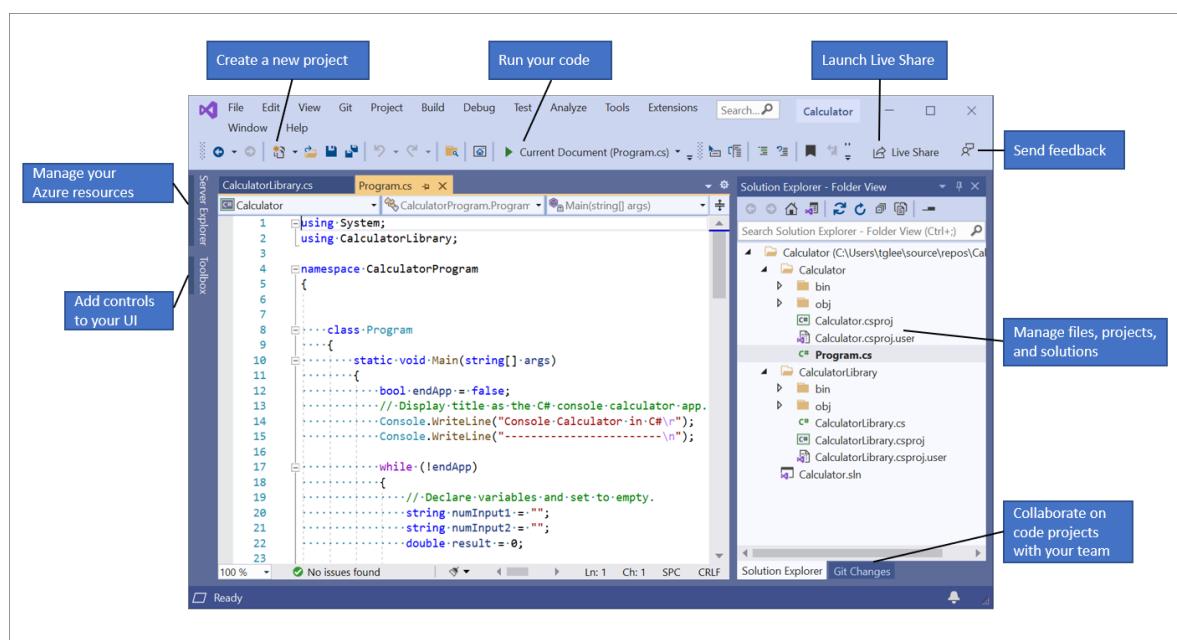
Obr. 34: Grafické prostředí programu Visual Studio Code. [Zdroj vlastní]

Tuto aplikaci lze vnímat zejména jako textový editor, který je posunut o několik úrovní výše díky volitelným rozšířením. Mimo ně se může pyšnit častými aktualizacemi, které jej obohacují o různé funkce zlepšující použitelnost. Aplikace se skládá z modulárních částí, jejichž pozici může uživatel měnit. Prostředí disponuje prohlížečem souborů, hlavní plochou pro editaci textu, a také vestavěný terminál. Dále je grafické rozhraní obohaceno o ikony, tlačítka nebo okna podle nainstalovaných rozšíření. Toto můžeme vidět v obrázku v pravém horním rohu, kde se nachází ikony z rozšíření *Code Runner*. To slouží ke zjednodušení spouštění a ladění aplikace přímo z grafického rozhraní.

Podle ankety uživatelů z roku 2018 bylo toto vývojové prostředí nejpopulárnější mezi vývojáři. Obdrželo vysoké hodnocení zejména v oblasti UX, rozšiřitelnosti a celkovém hodnocení. Na rozdíl od jiných editorů je první setkání s ním hodnoceno kladně, ale za některými zase zaostává po výkonové stránce. [24]

### 5.3 Visual Studio 2019

Dalším použitým vývojovým prostředím je Visual Studio 2019 ve verzi Community. Kromě editoru zdrojového kódu a nástrojů ladění toto vývojové prostředí nabízí také vlastní překladače, nástroje pro doplňování kódu, grafické návrháře a další nástroje pro ulehčení návrhu software. Jednou ze zajímavých funkcí je *Live Share*, která dovoluje více vývojářům kolaborovat na projektu a v reálném čase editovat a ladit kód. Vývojové prostředí je vybaveno dovedností navigace mezi zdrojovými soubory více různými způsoby, což programátorovi napomáhá při orientaci ve větších projektech. K usnadnění vývoje také přispívá technologie *IntelliSense*, kterou programátor využije při potřebě doplnit kód. Tento nástroj se snaží nabídnout nejrelevantnější výsledky na základě rozepsaného textu a také kontextu. Podobně jako předchozí vývojové prostředí, také Visual Studio 2019 nabízí množství možností úpravy vzhledu aplikace a rozložení oken. [25]



Obr. 35: Vývojové prostředí Visual Studio 2019. [25]

## 5.4 CMake

V rámci praktické části je pro sestavování ukázkových aplikací využit systém CMake. Jedná se o open-source systém, který spravuje proces sestavování programu způsobem nezávislým na překladači. Pracuje s konfiguračním souborem s pevně daným názvem *CMakeLists*, který se nachází v každém zdrojovém adresáři. Díky těmto souborům dokáže generovat výsledné soubory. CMake dokáže podporovat složité adresářové hierarchie a také aplikace závislé na více knihovnách. Díky jednoduchému návrhu tohoto systému je snadno rozšířitelný o nové funkce. [26]

V krátkém příkladu je nastíněna struktura konfiguračních souborů systému CMake. Uvažujeme hierarchii, ve které existuje jeden kořenový adresář, který obsahuje adresář *Hello* a adresář *Demo*. V kořenovém konfiguračním souboru specifikujeme minimální požadovanou verzi systému CMake, dále pak název projektu a specifikace podadresářů. Příklad dále ukaže, jak vyjádříme, že adresář *Hello* je knihovna. Naproti tomu adresář *Demo* definuje spustitelný soubor s přilinkovanou knihovnou. [27]

```
# CMakeLists files in this project can
# refer to the root source directory of the project as ${HELLO_SOURCE_DIR} and
# to the root binary directory of the project as ${HELLO_BINARY_DIR}.
cmake_minimum_required (VERSION 2.8.11)
project (HELLO)

# Recurse into the "Hello" and "Demo" subdirectories. This does not actually
# cause another cmake executable to run. The same process will walk through
# the project's entire directory structure.
add_subdirectory (Hello)
add_subdirectory (Demo)
```

Obr. 36: Konfigurační soubor kořenového adresáře. [27]

```
# Create a library called "Hello" which includes the source file "hello.hxx".
# The extension is already found. Any number of sources could be listed here.
add_library (Hello hello.hxx)

# Make sure the compiler can find include files for our Hello library
# when other libraries or executables link to Hello
target_include_directories (Hello PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

Obr. 37: Konfigurační soubor knihovny Hello. [27]

```
# Add executable called "helloDemo" that is built from the source files
# "demo.cxx" and "demo_b.hxx". The extensions are automatically found.
add_executable (helloDemo demo.cxx demo_b.hxx)

# Link the executable to the Hello library. Since the Hello library has
# public include directories we will use those link directories when building
# helloDemo
target_link_libraries (helloDemo LINK_PUBLIC Hello)
```

Obr. 38: Konfigurační soubor adresáře se spustitelným souborem. [27]

Podle konvence se výsledky sestavení zapisují do adresáře *build*. Tento adresář se vyskytuje ve stejné rovině, jako kořenový adresář. Jestliže se uživatel nachází v adresáři *build*, pak spustí samotný proces sestavování pomocí volání `cmake ..` z příkazové řádky. V případě úspěšného sestavení jsou výsledné soubory vytvořeny v adresáři *build*.

Tento sestavovací systém byl vybrán kvůli svému rozšíření, podpoře a přenositelnosti.

## 5.5 Hardwarové specifikace

Následující programy byly spuštěny na počítači autora diplomové práce. Výsledky charakterizují tuto konkrétní hardwarovou kombinaci. Jedná se o notebook Acer Aspire 5, konkrétně variantu A515-51G-55X7. Použitý operační systém je Microsoft Windows 10. Ačkoliv se nejedná o operační systém stejně flexibilní jako Linux, tento operační systém byl zvolen z důvodu svého rozšíření na PC v domácnostech a ve společnostech. Pro představení následujících příkladů, které demonstrují vliv tvorby programů na výkon, je dostačující. Následující údaje byly pořízeny pomocí programu CPU-Z.

Processor			
Name	Intel Core i5 8250U		
Code Name	Kaby Lake-R	Max TDP	15.0 W
Package	Socket 1356 FCBGA		
Technology	14 nm	Core VID	1.043 V
Specification			
Family	6	Model	E
Ext. Family	6	Ext. Model	8E
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3		
Clocks (Core #0)			
Core Speed	3091.66 MHz		
Multiplier	x 31.0 (4.0 - 34.0)		
Bus Speed	99.73 MHz		
Rated FSB			
Cache			
L1 Data	4 x 32 KBytes	8-way	
L1 Inst.	4 x 32 KBytes	8-way	
Level 2	4 x 256 KBytes	4-way	
Level 3	6 MBytes	12-way	
Selection	Socket #1	Cores	4
		Threads	8

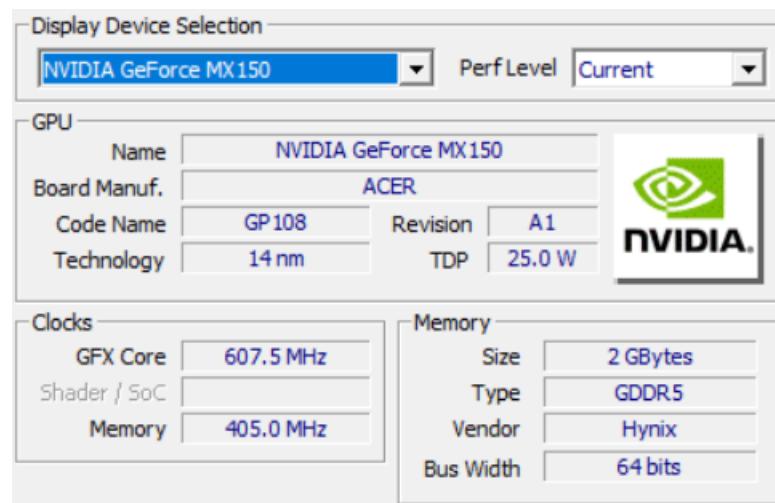
Obr. 39: Informace o CPU. [Zdroj vlastní]

General	
Type	DDR4
Size	20 GBytes
Channel #	Dual
DC Mode	
Uncore Frequency	
498.5 MHz	
Timings	
DRAM Frequency	1197.1 MHz
FSB:DRAM	3:36
CAS# Latency (CL)	17.0 clocks
RAS# to CAS# Delay (tRCD)	17 clocks
RAS# Precharge (tRP)	17 clocks
Cycle Time (tRAS)	39 clocks
Row Refresh Cycle Time (tRFC)	420 clocks
Command Rate (CR)	2T
DRAM Idle Timer	
Total CAS# (tRDRAM)	
Row To Column (tRCD)	

Obr. 40: Informace o operační paměti. [Zdroj vlastní]

Display Device Selection	
Intel(R) UHD Graphics 620	Perf Level
GPU	
Name	Intel® UHD Graphics 620
Board Manuf.	ACER
Code Name	
Technology	
Revision	7
TDP	
	
Clocks	
GFX Core	
Shader / SoC	
Memory	
Memory	
Size	
Type	
Vendor	
Bus Width	

Obr. 41: Informace o grafické kartě #1. [Zdroj vlastní]



Obr. 42: Informace o grafické kartě #2. [Zdroj vlastní]

## **II. PRAKTICKÁ ČÁST**

## 6 PŘÍKLADY OPTIMALIZACÍ

V následující sekci budou přestaveny vzorové programy, které znázorňují jeden nebo více principů nastíněných v teoretické části. Jedná se pouze o konceptuální podobu programů, které budou posléze přepsány do spustitelné formy. Představené varianty kódu mohou mít různý vliv na výkon v závislosti na platformě a na konkrétní aplikaci pracující s daty. Dosažené výsledky budou kvantifikovány a zhodnoceny v následující sekci.

### 6.1 Iterace polem

Většina programovacích jazyků nabízí mechanismus, který dovoluje opakování provádět stejnou posloupnost operací nad daty. V jazyce C++ k tomu slouží konstrukty jako *for*, *while* či *do-while*. Při využití smyčky *for* může programátor provést potřebná nastavení v inicializační části, stanovit ukončovací podmínu a také specifikovat velikost kroku od iterace k iteraci. Běžně je pro realizaci opakování využití instrukce podmíněného skoku. Nejprve se provede vyhodnocení ukončovací podmínky a posléze bud' pokračuje do další iterace, nebo smyčka končí.

#### 6.1.1 Iterace dvourozměrným polem

Velmi jednoduchá ukázka spolupráce s charakteristikami hardware jsou odlišné způsoby iterace přes dvourozměrné pole. Zdrojový kód 1 ukazuje dva základná přístupy. Jeden z nich bude rychlejší i přesto, že jejich asymptotická složitost je totožná.

Rychlejší způsob využívá faktu, že „prvky pole jsou uspořádány v paměti po řádcích.“ [7]

Dále je třeba si uvědomit, že při čtení i jednoho bytu z operační paměti dorazí celý cache blok. V případě, že jsou sousedící prvky ve vyrovnavací paměti, CPU k nim může přistoupit rychleji.

```

1 #define ROWS 5000
2 #define COLS 5000
3
4     double **array;
5
6     // Array initialization
7
8     double sum = 0.0;
9     for (int i = 0; i < ROWS; i++)
10    {
11        for (int j = 0; j < COLS; j++)
12        {
13            sum += array[i][j];
14        }
15    }
16
17    sum = 0.0;
18    for (int j = 0; j < COLS; j++)
19    {
20        for (int i = 0; i < ROWS; i++)
21        {
22            sum += array[i][j];
23        }
24    }

```

Zdrojový kód 1: Průchod dvourozměrným polem. [Zdroj vlastní]

Tento princip lokality je možné exploarovat při násobení matic. Prvky matice mohou být iterovány v různém pořadí, a toto pořadí může mít vliv na rychlosť běhu programu. Zdrojový kód 2 znázorňuje různé přístupové vzorce.

Ačkoliv je opět stejná asymptotická složitost pro všechny varianty, některé z nich lépe využívají lokality a dosahují různé míry cache miss. Liší se v tom, na které pole je odkazováno v nejvíce zanořené smyčce. [7]

```

1     double **A;
2     double **B;
3     double **C;
4
5     double sum = 0.0;
6     int n = // A number large enough to ensure one row does not fit
7 the L1 cache
8     // Version 1
9     for (int i = 0; i < n; i++)
10    {
11        for (int j = 0; j < n; j++)
12        {
13            sum = 0.0;
14            for (int k = 0; k < n; k++)
15            {
16                sum += A[i][k] * B[k][j];
17            }
18        }

```

```
19
20         C[i][j] = sum;
21     }
22 }
23
24 // Version 2
25 for (int j = 0; j < n; j++)
26 {
27     for (int i = 0; i < n; i++)
28     {
29         sum = 0.0;
30
31         for (int k = 0; k < n; k++)
32         {
33             sum += A[i][k] * B[k][j];
34         }
35
36         C[i][j] = sum;
37     }
38 }
39
40 // Version 3
41 double r = 0.0;
42 for (int j = 0; j < n; j++)
43 {
44     for (int k = 0; k < n; k++)
45     {
46         r = B[k][j];
47         for (int i = 0; i < n; i++)
48         {
49             C[i][j] += A[i][k] * r;
50         }
51     }
52 }
53
54 // Version 4
55 for (int k = 0; k < n; k++)
56 {
57     for (int j = 0; j < n; j++)
58     {
59         r = B[k][j];
60         for (int i = 0; i < n; i++)
61         {
62             C[i][j] += A[i][k] * r;
63         }
64     }
65 }
66
67 // Version 5
68 for (int k = 0; k < n; k++)
69 {
70     for (int i = 0; i < n; i++)
71     {
72         r = A[i][k];
73         for (int j = 0; j < n; j++)
74         {
```

```

75             C[i][j] += r * B[k][j];
76         }
77     }
78 }
79
80 // Version 6
81 for (int i = 0; i < n; i++)
82 {
83     for (int k = 0; k < n; k++)
84     {
85         r = A[i][k];
86         for (int j = 0; j < n; j++)
87         {
88             C[i][j] += r * B[k][j];
89         }
90     }
91 }
```

Zdrojový kód 2: Různé způsoby průchodu maticí při sčítání matic. [7]

### 6.1.2 Závislost iterací

Iterace, následující po sobě ve smyčce, mohou mít datové závislosti na výsledcích iterací předchozích, čímž dojde k vynucené sekvenčnímu provádění a tím pádem nevyužití potenciálu paralelizace smyčky. Jak je zřejmé z představené problematiky v teoretické sekci, v některých případech je možné smyčku přepsat tak, aby mohla být provedena paralelně. Zdrojový kód 3 zobrazuje smyčku se závislostí a bez závislosti.

```

1   double *A;
2   double *B;
3   double *C;
4   double *D;
5
6   int n = // A number large enough to show the impact
7
8   // With dependence - current value of B depends on previous iteration
9   for (int i = 0; i < n; i++)
10  {
11      A[i] = A[i] + B[i];
12      B[i+1] = C[i] + D[i];
13  }
14
15
16 // Without dependence
17 A[0] = A[0] + B[0];
18 for (int i = 0; i < n - 1; i++)
19 {
20     B[i+1] = C[i] + D[i];
21     A[i+1] = A[i+1] + B[i+1];
22 }
23 B[n] = C[n-1] + D[n-1];
```

Zdrojový kód 3: Smyčka se závislostí a bez závislosti. [8]

### 6.1.3 Loop unrolling

Pro zvýšení počtu instrukcí provádějících pro nás užitečnou práci lze využít několikanásobné duplikování operací v těle smyčky pro několik po sobě následujících iterací. Zdrojový kód 4 demonstruje *loop unrolling 2x1*.

```
1 void loop_unrolling_slow(double* array, int n, double* result) {
2
3     for (int i = 0; i < n; i++)
4     {
5         *result += array[i];
6     }
7 }
8
9 void loop_unrolling_fast(double* array, int n, double* result) {
10
11    int i = 0;
12    int limit = n - 1;
13    double accumulator = 0.0;
14
15    // Unrolled loop
16    for (; i < limit; i+=2)
17    {
18        accumulator += array[i] + array[i+1];
19    }
20
21    // Finalizing loop - process remaining elements
22    for (; i < n; i++)
23    {
24        accumulator += array[i];
25    }
26
27    *result = accumulator;
28 }
```

Zdrojový kód 4: Loop unrolling. [7]

Funkce disponující unrolled smyčkou využívá mimo jiné také optimalizaci v podobě akumulace hodnot do lokální proměnné, čímž potenciálně eliminuje opakování přístupy do paměti z důvodu využití registru procesoru. K zapsání výsledné hodnoty do výsledné proměnné se provede až na samotném konci funkce. Rovněž je zde uveden obecný tvar pro unrolling, což může být zřejmé z druhé, finalizační smyčky. V závislosti na míře unrollingu a hodnotě proměnné  $n$  nemusí být všechny prvky pole zpracovány v hlavní smyčce. Jak bylo zmíněno v teoretické části, lokálních akumulačních proměnných můžeme deklarovat více, ale je třeba myslet na fyzické možnosti procesoru v podobě omezeného počtu využitelných registrů. Rovněž můžeme zvýšit stupeň unrollingu vícenásobnou duplikací těla smyčky a odpovídajícího navýšení kroku iterační proměnné.

### 6.1.4 Projevy asociativity vyrovnávacích paměti

Při konstrukci vyrovnávacích pamětí procesoru je zpravidla využito set-associative vyrovnávacích pamětí s daným stupněm asociativity. Dále bývá nejvyšší úroveň vyrovnávací paměti sdílena mezi všemi jádry. Je třeba myslet na to, že počet míst pro data v jedné sadě je omezený a že existuje možnost thrashingu při opakovém odkazování na data, která náleží do stejné sady. Zdrojový kód 5 se snaží znázornit tuto skutečnost.

V mnoha případech používají programátoři při tvorbě smyček s nejednotkovým krokem krok, který je mocnina dvou. Při opakové inkrementaci můžeme při převodu výsledné iterační hodnoty do dvojkové soustavy sledovat, že ke změně dochází zejména při vyšších bitech. Ovšem k výběru sady bývají používány bity nižší, a ty se v tomto případě nemění.

[28]

```
1 // Steps: 30, 32, 60, 64, 120, 128, 250, 256, 510, 512
2 void cache_associativity_limit(int step) {
3
4     double *array;
5     int n = // A number to ensure the same number of memory accesses
6 for every step
7
8     // Array initialization
9
10    double sum = 0.0;
11
12    for (int i = 0; i < n; i += step)
13    {
14        sum += array[i];
15    }
16 }
```

Zdrojový kód 5: Průchod smyčkou s různým krokem. [Zdroj vlastní]

## 6.2 Předvídatelnost operací

Moderní hardware využívá prefetchingu pro spekulativní načtení dat do paměti. Dokáže také vyzkoušet různé pravidelné vzory přístupu k datům. Ne vždy je ale využíván dostatečně předvídatelný postup. Limitace hardware pro předpovídání přístupu k datům bude představena v následujících příkladech. Zdrojový kód 6 prezentuje různé způsoby přístupu do pole.

```
1  double *array;
2  int n = // A number to ensure the same number of memory accesses for
3 every variant
4
5  // Array initialization
6
7  double sum = 0.0;
8
9  // Sequential
10 for (int i = 0; i < n; i++)
11 {
12     sum += array[i];
13 }
14
15 // Strided
16 for (int i = 0; i < n; i += step)
17 {
18     sum += array[i];
19 }
20
21 // Pattern
22 for (int i = 0; i < n; i++)
23 {
24     int index = (2 * i + 5) % n;
25     sum += array[index];
26 }
27
28 // Random
29 srand(seed);
30
31 for (int i = 0; i < n; i++)
32 {
33     int index = rand() % n;
34     sum += array[index];
35 }
```

Zdrojový kód 6: Přístupy do pole s různým krokem. [Zdroj vlastní]

### 6.3 Hot vs cold data

Následující příklad se týká uspořádání dat v datové struktuře. Je zde využito faktu, že nevšechny proměnné jsou odkazovány stejně často. Některé jsou čteny či zapisovány častěji než jiné.

Frekventovaně odkazovaná data jsou nazývána jako *hot*. Naproti tomu málo odkazovaná data jsou označována jako *cold*. [1]

Jestliže se často volá nějaká operace, která pracuje pouze s částí dat z datové struktury, nevyužívá se celé kapacity cache bloku a pro načtení potřebných dat je třeba se častěji odkazovat do paměti. V případě, že je tato operace volána často, dopad na čas jejího běhu může

být značný. Zdrojový kód 7 porovnává klasickou deklaraci a rozdělní dat podle frekvence používání.

```

1 // Original
2 struct Data {
3     double a, b, c, d, e, f, g, h, result;
4 };
5
6 Data* data;
7 int n = // Number of elements in data array
8
9 // Array initialization
10
11 for (int i = 0; i < n; i++)
12 {
13     data[i].result = data[i].a + data[i].b * data[i].c;
14 }
15
16 // Hot and cold data separated
17 struct DataHot {
18     double a, b, c, result;
19 };
20
21 struct DataCold {
22     double d, e, f, g, h;
23 };
24
25 struct Data {
26     DataHot* hot;
27     DataCold* cold;
28 };
29
30 Data data;
31 int n = // Number of elements in data array
32
33 // Array initialization
34
35 for (int i = 0; i < n; i++)
36 {
37     data.hot[i].result = data.hot[i].a + data.hot[i].b *
38 data.hot[i].c;
39 }
```

Zdrojový kód 7: Separace často odkazovaných a málo odkazovaných dat. [Zdroj vlastní]

## 6.4 SoA vs AoS

Typickým příkladem rozdílu mezi OOP a DOP je využití *structure of arrays*. Zdrojový kód 8 zobrazuje deklaraci datové struktury AoS, která značí *array of structures*. Tento způsob je běžně využíván při použití OOP. Je vytvořeno pole těchto datových struktur a je použito podle příkladu ve smyčce. V paměti jsou jednotlivé proměnné ve struktuře poskládány za

sebe. Jestliže některé z nich nejsou využity při často volané operaci, mohou se kapacitou cache bloku a je třeba častěji číst data z paměti. Naproti tomu se při DOP využije *structure of arrays*. V tomto případě se vytvoří jedna proměnná pro data, která ale obsahuje pole pro jednotlivé proměnné. Alokujeme se tedy každé z nich. Poté se ve smyčce indexuje až do polí v této struktuře. Je tak lépe využito kapacity cache bloku. Potenciálně by mohlo být AoS využito v kombinaci s *loop unrollingem* či SIMD instrukcemi pro využití prostorové lokality. Zdrojový kód 8 poukazuje na rozlišností těchto přístupů.

```
1 // Array of structures
2 struct AoS {
3     double a, b, c, d, e, f, g, h, result;
4 }
5
6 AoS* data;
7 int n = // Number of elements in data array
8
9 // Array initialization
10
11 for (int i = 0; i < n; i++)
12 {
13     data[i].result = data[i].a + data[i].b * data[i].c;
14 }
15
16 // Structure of arrays
17 struct SoA {
18     double* a;
19     double* b;
20     double* c;
21     double* d;
22     double* e;
23     double* f;
24     double* g;
25     double* h;
26     double* results;
27 };
28
29 SoA data;
30 int n = // Number of elements in each array of data
31
32 // Array initialization
33
34 for (int i = 0; i < n; i++)
35 {
36     data.results[i] = data.a[i] + data.b[i] * data.c[i];
37 }
```

Zdrojový kód 8: AoS a SoA přístup. [Zdroj vlastní]

Praktickým příkladem použití je při implementaci hash tabulky. V případě, že je statisticky nejčastěji používaná operace vyhledávání mezi klíči, pak je strukturování hash tabulky jako páru klíč-hodnota nehospodárné, když se bere v potaz utilizaci cache bloku. Dává tedy smysl

mít separátní pole pro klíče a hodnoty. Po vyhledání klíče a indexaci do pole hodnot nastane cache miss, ale jelikož se jedná o operaci, která nastává méně často než vyhledávání, je to přijatelné řešení. [3]

## 6.5 SIMD

S využitím SIMD instrukcí lze zpracovávat několik dat najednou. V příkladu je ukázka využití knihovny `intrin.h`. Zdrojový kód 9 ukazuje využití intrinsicských funkcí, které se mapují na SIMD instrukce. Jsou využity SSE instrukce. Jelikož jsou kladený nároky na zarovnání dat, je třeba využít funkce `_aligned_malloc`. Načtení existující proměnné do vektorové proměnné provedeme pomocí `_mm_loadu_pd`. Jelikož je zpracováno více dat najednou, počet iterací je nižší. Na závěr je třeba sečít hodnoty obsažené v proměnné pro akumulaci, jelikož obsahuje dvě skalární hodnoty. Příklad uvažuje, že počet prvků v poli je zpracován kompletně hlavní smyčkou. Reálně by pak byla přidána další smyčku, která by zpracovala zbytek s využitím buď vektorového, či skalárního přístupu.

```
1 #define N // Number of elements in array
2
3     // Scalar
4     double* array;
5     double sum = 0.0;
6
7     // Array initialization
8
9     for (int i = 0; i < N; i++)
10    {
11        sum += array[i];
12    }
13
14    // Vector
15    double* array = (double*)_aligned_malloc(sizeof(double) * N, 16);
16
17    // SSE specific step
18    int step = 128 / (sizeof(double) * 8);
19    int sumSize = step;
20    double* sum = (double*)malloc(sizeof(double) * sumSize);
21
22    __m128d sumVector = _mm_loadu_pd(sum);
23    __m128d* arrayVector = (__m128d*)array;
24    int vecIterations = N / step;
25
26    for (int i = 0; i < vecIterations; i++)
27    {
28        sumVector = _mm_add_pd(arrayVector[i], sumVector);
29    }
30
31    _mm_store_pd(sum, sumVector);
32
```

```

33     double sumTotal = 0.0;
34
35     for (int i = 0; i < sumSize; i++)
36     {
37         sumTotal += sum[i];
38     }

```

Zdrojový kód 9: Suma hodnot v poli: skalární a SIMD způsob. [Zdroj vlastní]

## 6.6 Zarovnání dat

Zdrojový kód 10 ukazuje, že dvě datové struktury, které mají identické členské proměnné, ale jejich pořadí se liší, mají různou velikost.

```

1   struct Data {
2       bool a;
3       int32_t b;
4       bool c;
5       int64_t d;
6       bool e;
7   };
8
9   sizeof(Data) // 32
10
11  struct DataOrdered {
12      int64_t a;
13      int32_t b;
14      bool c, d, e;
15  };
16
17  sizeof(DataOrdered) // 16

```

Zdrojový kód 10: Vliv pořadí proměnných v datové struktuře na její velikost.

[Zdroj vlastní]

Velikost datového typu proměnné klade určité nároky na umístění v paměti. Zarovnání dat musí být dodrženo jak pro jednotlivé proměnné, tak pro celou datovou strukturu. To je řešeno vkládáním prázdných proměnných. Tento fakt lze pozorovat při deklaraci datové struktury. Důvodem zarovnání je, že instrukce jsou navrženy pro práci nad uniformními daty. Obecné doporučení je řadit členské proměnné podle velikosti jejich datového typu sestupně. [28]

## 6.7 Datové a kontrolní závislosti

Zdrojový kód 9 obsahuje datovou závislost. Proměnná `sum` je odkazována v každé iteraci a její aktuální hodnota je závislá také na její hodnotě předchozí. Jedná se tedy o *loop-carried dependence*.

Odstranit tento nedostatek lze pomocí rozvinutí této proměnné do vektoru a sumu každé iterace uložit na odpovídající pozici v jednorozměrném poli. Ovšem poté bude třeba tyto mezivýsledky sečít v další smyčce. [8]

Zdrojový kód 4 představuje kontrolní závislosti, které se dají redukovat pomocí aplikace *loop unrollingu*. Větvení programu může mít mnohdy zásadní vliv na výkon programu a díky snížení množství jejich výskytu se jedná o vhodnou optimalizaci.

Hardware se dále snaží snižovat dopad větvení pomocí předvídání, zda se kód větve provede (*taken*), nebo ne (*not taken*). Programátor může pomocí hardware tím, že podmínky ve větvích budou co nejvíce předvídatelné. [8]

```
1   double* array = (double *)malloc(sizeof(double) * N);
2   srand(0);
3
4   // Initialize
5   for (int i = 0; i < N; ++i)
6   {
7       array[i] = rand() % 10;
8   }
9
10  // Process predictable
11  for (int i = 0; i < N; ++i)
12  {
13      if (array[i] > 20.0) {
14          printf("%f", array[i]);
15      }
16  }
17
18  // Process predictable
19  for (int i = 0; i < N; ++i)
20  {
21      if (array[i] >= 0.0 && array[i] < 10.0) {
22          printf("%f", array[i]);
23      }
24  }
25
26  // Process not so predictable
27  for (int i = 0; i < N; ++i)
28  {
29      if (array[i] > 5.0) {
30          printf("%f", array[i]);
31      }
32  }
```

Zdrojový kód 11: Předvídatelnost podmínek. [Zdroj vlastní]

Zdrojový kód 11 ukazuje příklady různě předvídatelných podmínek. Alokuj se zde pole s počtem prvků  $N$  a nastaví se hodnoty pomocí funkce *rand* tak, že žádný prvek nemá hodnotu větší než 10. První zpracovávací smyčka testuje, zda je hodnota prvku větší než 20.

Tato podmínka v tomto případě nikdy nenastane. Kompilátor by tuto skutečnost mohl zaznamenat a tuto sekci kódu by vůbec nemusel překládat. Pokud kód odstraněn nebude, ale spoň pomůžeme hardware pro spekulaci tím, že větev bude pokaždě *not taken*. Další zpracovávací smyčka má naopak podmínku, která bude pokaždě kladná. V tomto případě bude větev pokaždě *taken*. Vhodná optimalizace by zde byla zcela odstranit instrukce pro kontrolu a využít *loop unrolling*. Poslední zpracovávací smyčka není zdaleka tak předvídatelná, jako ty předchozí. Prostor pro optimalizace je zde výrazně nižší.

## 6.8 Pravé a falešné sdílení

Zdrojový kód 12 znázorňuje vliv udržování koherence vyrovnávacích pamětí na výkon aplikace. První příklad je ukázka pravého sdílení dat. Čtyři vlákna procesoru přistupují ke sdílené proměnné. Další ukázka představuje falešné sdílení. Každé vlákno nyní inkrementuje jinou proměnnou, ale jelikož jsou tyto proměnné umístěny v paměti hned vedle sebe, je zde pravděpodobnost, že se vyskytují ve stejném cache bloku.

```
1 #define N // Number of iterations
2
3     // True sharing
4     int shared = 0;
5     mutex m;
6
7     // Do on each core
8     for (int i = 0; i < N; i++)
9     {
10         // Each core increments a shared variable
11         mutex.lock();
12         ++shared;
13         mutex.unlock();
14     }
15
16     // False sharing
17     int32_t dataA = 0;
18     int32_t dataB = 0;
19     int32_t dataC = 0;
20     int32_t dataD = 0;
21
22     // Do on each core
23     for (int i = 0; i < N; i++)
24     {
25         // Each core increments a not shared variable
26         ++data(A|B|C|D);
27     }
28
29     // False sharing fix
30 #define ALIGNMENT (64)
31     alignas(ALIGNMENT) int32_t dataAlignedA = 0;
32     alignas(ALIGNMENT) int32_t dataAlignedB = 0;
```

```

33     alignas(ALIGNMENT) int32_t dataAlignedC = 0;
34     alignas(ALIGNMENT) int32_t dataAlignedD = 0;
35
36     // Do on each core
37     for (int i = 0; i < N; i++)
38     {
39         // Each core increments a not shared variable
40         ++dataAligned(A|B|C|D);
41     }

```

Zdrojový kód 12: Pravé a falešné sdílení. [Zdroj vlastní]

Při přístupu každého vlákna k proměnné dojde k invalidaci bloku a následný cache miss pro ostatní vlákna. Způsob opravy tohoto jevu je umístit každou proměnnou do odlišného cache bloku pomocí zarovnání. [29]

## 6.9 Aliasing paměti

Při vytváření funkce v jazyce C++ lze předávat jak hodnotu, tak odkaz do paměti. Při využití ukazatelů jako argumentů funkce vzniká omezení pro překladač, který nemůže aplikovat různé optimalizace. Ukazatele by totiž mohly ukazovat na stejně místo v paměti. Toto bývá označováno jako *memory aliasing*. Pokud oba argumenty obsahují stejnou hodnotu, je třeba dodržet určité pořadí operací.

```

1 void write_read(double* src, double* dest, int n) {
2     int count = n;
3     int val = 0;
4
5     while (count) {
6         *dest = val;
7         val = (*src) + 1;
8         count--;
9     }
10 }
11
12 // Usage
13 #define N // Number of writes and reads
14 double *array;
15
16 // Array initialization
17
18 // Independent
19 write_read(&array[0], &array[1], N);
20 // Aliased
21 write_read(&array[0], &array[0], N);

```

Zdrojový kód 13: Demonstrace aliasingu paměti. [7]

Zdrojový kód 13 ilustruje *write/read dependency*, kde výsledek čtení závisí na nedávné operaci zápisu. Pro pochopení charakteristiky rozdílné doby běhu mezi voláním funkce *write-*

*\_read* s argumenty, které ukazují buď na odlišné nebo stejné místo v paměti, je třeba si uvědomit způsob využití funkčních jednotek v CPU. Jednotky pro načítání a čtení dat z paměti mohou pracovat nezávisle na sobě. To platí pouze v případě, že čtení nezávisí na předchozím zápisu. Jednotka pro čtení při své operaci kontroluje *store buffer* jednotky pro zápis pro zjištění, zda není požadavek na zápis na určité místo v paměti. Jestliže se ve *store bufferu* nachází odpovídající záznam, jednotka pro čtení musí vyčkat na provedení zápisu, jinak může pokračovat ve čtení. Jedná se o další ukázku ILP. [7]

## 6.10 Návrhové vzory z pohledu DOP

Návrhové vzory reprezentují „strategie a řešení aktivit, se kterými se často setkáváme.“ [1] Mohou být použity v různých programovacích jazycích, ale neexistují jako šablonová řešení z důvodu, že „komplexita nepramení z jazyka samotného, ale kvůli faktu, že řešení pochází z problémové domény. Pokud bychom přestali řešený problém modelovat jako objekty a dívali se na něj v podobě technického návrhu, mnoho návrhových vzorů by nebylo zapotřebí.“ [1]

### 6.10.1 Existence based processing

Tento způsob práce s daty „zpracovává každý prvek v homogenní sadě dat.“ [1]

Existence based processing respektuje způsob funkce hardware a uvědomuje si výhody možné aplikace různých úrovní paralelismu, stejně jako implikace operování nad prvky v poli, které jsou v paměti umístěny za sebou. Zásadní myšlenka je snížit počet všudypřítomných last-minute podmínek, podle kterých se data budou zpracovávat nebo ne. Toto je docíleno použitím polí s implicitní booleovskou hodnotou. Stejná logika je aplikována pro odstranění rozhodování na základě hodnot enumerací. Rovněž lze tímto způsobem zaregistrovat obsluhu události, kterou je třeba sledovat a zareagovat na ni. [1]

Zdrojový kód 14 ukazuje dvě pole hodnot. Jedno uchovává hodnoty větší nebo rovny nule a druhé hodnoty menší než nula. Implicitní proměnná je v tomto případě „kladná“ a „záporná“ čísla. V některých případech je výhodnější podoba smyčky na řádku číslo 6. Pokud se ale tato logika opakuje, může se vyplatit udržovat si separovaná pole hodnot a následně je pouze zpracovat a neprovádět redundantní kontrolu hodnot. Opět se tedy snižuje počet instrukcí, které jsou potřeba pouze z důvodu režie smyčky, a tedy zvyšuje se množství pro nás přínosné vykonané práce.

```
1 #define N // Number of elements in array
2     double *array;
3
4     // Array initialization
5
6     for (int i = 0; i < N; ++i)
7     {
8         if (array[i] >= 0.0)
9         {
10             // Positive logic
11         }
12     else
13     {
14         // Negative logic
15     }
16 }
17
18 double *positive;
19 double *negative;
20 int positiveCount;
21 int negativeCount;
22
23 for (int i = 0; i < positiveCount; ++i)
24 {
25     // Positive logic on positive[i]
26 }
27
28 for (int i = 0; i < negativeCount; ++i)
29 {
30     // Negative logic on negative[i]
31 }
```

Zdrojový kód 14: Ukázka existence based processing. [Zdroj vlastní]

### 6.10.2 Component based objects

Tento způsob pohledu na objekty, se kterými program pracuje, zcela převrací přístup použitý v objektově orientovaném programování. Místo specifikace objektu jako kontejneru, který obsahuje různé, spolu logicky nesouvisející proměnné, lze vnímat objekt jako výsledek souhrnu *komponent*. Komponentu si lze představit jako třídu nebo strukturu obsahující proměnné, které je logické umístit k sobě. Ta potom zpravidla existuje spolu s ostatními komponentami stejného typu v poli. Na rozdíl od iniciálního návrhu, je zde využito principů lokality ku prospěchu. Původní objekt je nyní popisován pouze spojením odpovídajících částí. Pro práci s ním stačí využít ID pro indexaci nebo vyhledávání v poli komponentů. Tímto vzniká pomocný objekt pro identifikaci zvaný *entita*, který obsahuje pouze unikátní ID. Je-likož komponenty obsahují pouze data, je třeba implementovat způsob práce s těmito daty. K tomu slouží pomocné objekty zvané *managery*, které iterují přes pole komponent a

provádějí odpovídající operace. Tento koncept má nejen výkonové implikace, ale také umožnuje triviálním způsobem rozšířit funkcionality existujícího kódu. [1]

#### 6.10.3 Adapter

Návrhový vzor „adapter“ slouží k práci s rozhraním, které není možné upravit například z důvodu nepřístupného zdrojového kódu. Pokud si přejeme využít rozhraní jiného, je třeba vytvořit takovýto pomocný adaptér, který vytvoří z dat, se kterými se pracuje, nová data v požadovaném formátu. [30]

V případě DOP pracujeme s daty, která se nachází v poli nebo podobném kontejneru. Data sama o sobě nemají rozhraní, tudíž není co adaptovat. V případě potřeby lze snadno vytvořit transformaci pro práci s potřebnými daty. [1]

#### 6.10.4 Composite

Návrhový vzor „composite“ vytváří identické rozhraní pro práci s jednotlivými objekty a se skupinami objektů. [30]

Jelikož data se při použití Component based objects nesoustředí do jednoho objektu, nevzniká problém při práci se skupinami objektů. [1]

#### 6.10.5 Memento

Návrhový vzor „memento“ je zástupcem behaviorálních návrhových vzorů a je využíván k uchovávání stavu systému pro případnou potřebu návratu do tohoto stavu. Rovněž ho můžeme využít pro implementaci chování *undo* a *redo*. [30]

DOP vidí využití pro tento návrhový vzor, specificky při implementaci *level-of-detail* ve videohrách. Při změnách úrovně detailu se vytvoří komprimovaná sada dat, které mohou být v budoucnu využity pro rekonstrukci obrazu s vyššími detaily. [1]

## 7 OVĚŘENÍ VÝKONU POMOCÍ NÁSTROJŮ

Následující sekce prezentuje výsledky implementací jednotlivých zdrojových kódů. Tyto výsledky byly získány za použití knihovny Google Benchmark. Rovněž se zde porovnává různé překladače jazyka C++. V příkladech, kde se pracuje s jednorozměrným polem, jsou stanoveny 3 různé počty prvků těchto polí. Tyto počty byly zvoleny podle velikosti vyrovnávací paměti počítače, který byl použit pro spuštění benchmark. Toto umožňuje lépe pochopit, pro jaké případy budou mít následující optimalizace smysl, a pro které zase ne. Nejmenší velikost je stanovena tak, aby se celé pole vešlo do L1. Následující velikost zaručuje, že se pole vejde do L3. Největší velikost zajistí, že se celé pole nevejde do vyrovnávací paměti. Stejná idea je aplikována u polí dvourozměrných. Dále se u každého překladače porovnávají dva stupně optimalizace. Tyto stupně optimalizace jsou stanoveny výrobcem překladačů a aplikují se při překladu programu.

### 7.1 Měření doby vykonávání ukázkových příkladů

Tabulka 1 ukazuje rozdíly mezi iterací dvourozměrným polem po řádku a po sloupci. Z výsledků je zjevné, že se rozdíl nejvíce projevuje v případě, že se celé pole nevejde do vyrovnávací paměti. Bližší informace jsou uvedeny v sekci 3.5.

Tabulka 1: Výsledky implementace zdrojového kódu 1

GCC						
	O0			O2		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>Matice 45x45</b>						
<b>Po řádku</b>	112000	0.005468	0.005441	298667	0.002370	0.002407
<b>Po sloupcí</b>	100000	0.005479	0.005469	280000	0.002386	0.002344
<b>Matice 170x170</b>						
<b>Po řádku</b>	8960	0.079244	0.078474	20364	0.035140	0.033761
<b>Po sloupcí</b>	8960	0.080019	0.080218	20364	0.034778	0.035295
<b>Matice 900x900</b>						
<b>Po řádku</b>	320	2.302900	2.294920	747	1.052660	1.045850
<b>Po sloupcí</b>	154	4.568580	4.565750	448	1.517130	1.534600
<b>Clang</b>						
	O0			O2		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]

	<b>Matice 45x45</b>					
<b>Po řádku</b>	100000	0.005491	0.005625	280000	0.002388	0.002344
<b>Po sloupci</b>	100000	0.005460	0.005469	280000	0.002430	0.002400
<b>Matrice 170x170</b>						
<b>Po řádku</b>	8960	0.079068	0.076730	17231	0.036205	0.035365
<b>Po sloupci</b>	8960	0.084000	0.080218	19478	0.037783	0.037703
<b>Matrice 900x900</b>						
<b>Po řádku</b>	299	2.309190	2.299330	640	1.198340	1.196290
<b>Po sloupci</b>	160	4.916930	4.882810	448	1.613330	1.639230
<b>MSVC</b>						
	<b>O0</b>			<b>O2</b>		
	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>
<b>Matrice 45x45</b>						
<b>Po řádku</b>	100000	0.005514	0.005469	298667	0.002386	0.002354
<b>Po sloupci</b>	112000	0.005481	0.005441	298667	0.002396	0.002407
<b>Matrice 170x170</b>						
<b>Po řádku</b>	8960	0.078667	0.078474	19478	0.034756	0.034494
<b>Po sloupci</b>	8960	0.078929	0.078474	20364	0.035150	0.035295
<b>Matrice 900x900</b>						
<b>Po řádku</b>	299	2.291930	2.299330	640	1.040020	1.025390
<b>Po sloupci</b>	160	4.349880	4.394530	448	1.642000	1.639230

Tabulka 2 demonstруje vliv pořadí iterace dvourozměrným polem na dobu běhu programu. Lze pozorovat, že u různých stupňů optimalizace dochází ke zrychlení u odlišných způsobů iterování.

Tabulka 2: Výsledky implementace zdrojového kódu 2

	<b>GCC</b>					
	<b>O0</b>			<b>O2</b>		
	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>
<b>Matrice 45x45</b>						
<b>IJK</b>	2635	0.262455	0.254981	8960	0.076323	0.076730
<b>JIK</b>	2800	0.254998	0.256696	8960	0.081097	0.081962
<b>JKI</b>	2240	0.320907	0.320871	8960	0.072310	0.071498
<b>KJI</b>	1792	0.309481	0.305176	11200	0.070737	0.071150
<b>KIJ</b>	2489	0.291873	0.295048	11200	0.056337	0.055804
<b>IKJ</b>	2358	0.336317	0.337945	11200	0.058206	0.058594
<b>Matrice 170x170</b>						
<b>IJK</b>	45	16.197400	16.319400	112	5.847800	5.719870

<b>JIK</b>	45	15.479200	15.625000	112	5.763180	5.719870
<b>JKI</b>	32	22.867500	22.949200	112	5.824770	5.859380
<b>KJI</b>	34	21.510900	21.599300	100	5.768720	5.781250
<b>KIJ</b>	45	18.397500	18.402800	224	3.095780	3.138950
<b>IKJ</b>	34	16.335100	16.084600	224	3.035120	2.999440
<b>Clang</b>						
<b>O0</b>			<b>O2</b>			
Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]	
<b>Matrice 45x45</b>						
<b>IJK</b>	2800	0.257850	0.256696	8960	0.078620	0.078474
<b>JKI</b>	2635	0.253384	0.254981	8960	0.084917	0.085449
<b>KJI</b>	2240	0.309895	0.306920	8960	0.089540	0.088937
<b>KIJ</b>	2240	0.334814	0.334821	6400	0.079679	0.078125
<b>KIJ</b>	2489	0.294384	0.295048	11200	0.057587	0.057199
<b>IKJ</b>	2133	0.348725	0.351617	11200	0.057673	0.057199
<b>Matrice 170x170</b>						
<b>IJK</b>	41	16.251000	16.006100	112	5.709020	5.580360
<b>JKI</b>	45	15.195100	14.930600	112	6.544400	6.277900
<b>JKI</b>	32	21.779200	21.484400	90	6.886200	6.770830
<b>KJI</b>	34	21.037400	21.139700	75	7.195780	7.291670
<b>KIJ</b>	32	17.267200	17.089800	204	3.343830	3.293500
<b>IKJ</b>	41	16.353300	16.387200	224	3.554320	3.557480
<b>MSVC</b>						
<b>O0</b>			<b>O2</b>			
Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]	
<b>Matrice 45x45</b>						
<b>IJK</b>	2800	0.252479	0.256696	8960	0.074493	0.073242
<b>JKI</b>	2800	0.252359	0.251116	8960	0.078194	0.078474
<b>JKI</b>	2240	0.311413	0.313895	8960	0.075970	0.076730
<b>KJI</b>	2240	0.311146	0.306920	8960	0.074303	0.073242
<b>KIJ</b>	2240	0.318085	0.320871	16000	0.044134	0.043945
<b>IKJ</b>	2240	0.328351	0.334821	10000	0.050240	0.050000
<b>Matrice 170x170</b>						
<b>IJK</b>	45	15.108800	14.930600	112	5.790410	5.719870
<b>JKI</b>	45	14.159400	14.236100	112	5.902340	5.859380
<b>JKI</b>	34	20.858300	21.139700	100	6.682410	6.718750
<b>KJI</b>	34	19.920000	19.761000	100	5.662600	5.781250
<b>KIJ</b>	37	17.865600	17.736500	204	3.130420	3.063730
<b>IKJ</b>	37	19.116700	18.581100	224	2.765810	2.720420

Tabulka 3 ukazuje, že v tomto případě nemá odstranění *loop-carried* závislost kladný vliv na výkon. Dobu běhu buď nesníží, nebo dokonce zhorší.

Tabulka 3: Výsledky implementace zdrojového kódu 3

	<b>GCC</b>					
	<b>O0</b>			<b>O2</b>		
	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>
<b>n=2000</b>						
<b>Se závislostí</b>	89600	0.007625	0.007673	344615	0.001966	0.001995
<b>Bez závislosti</b>	89600	0.007843	0.007673	298667	0.002280	0.002302
<b>n=30000</b>						
<b>Se závislostí</b>	6400	0.112079	0.112305	19478	0.035356	0.035296
<b>Bez závislosti</b>	5600	0.118491	0.117188	22400	0.032006	0.032087
<b>n=800000</b>						
<b>Se závislostí</b>	160	5.690670	5.664060	264	2.682430	2.663350
<b>Bez závislosti</b>	100	5.259620	5.156250	299	3.180580	3.187710
<b>Clang</b>						
	<b>O0</b>			<b>O2</b>		
	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>
<b>n=2000</b>						
<b>Se závislostí</b>	89600	0.007758	0.007673	320000	0.002134	0.002148
<b>Bez závislosti</b>	89600	0.007876	0.007847	320000	0.002128	0.002100
<b>n=30000</b>						
<b>Se závislostí</b>	4480	0.124237	0.125558	22400	0.032281	0.032087
<b>Bez závislosti</b>	6400	0.132115	0.131836	19478	0.034193	0.033692
<b>n=800000</b>						
<b>Se závislostí</b>	160	4.385400	4.394530	166	3.904420	3.859190
<b>Bez závislosti</b>	172	4.711280	4.723840	249	2.551100	2.572790
<b>MSVC</b>						
	<b>O0</b>			<b>O2</b>		
	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>
<b>n=2000</b>						
<b>Se závislostí</b>	89600	0.007308	0.007324	407273	0.001739	0.001726
<b>Bez závislosti</b>	89600	0.008181	0.008196	373333	0.001878	0.001883
<b>n=30000</b>						
<b>Se závislostí</b>	6400	0.112364	0.114746	23579	0.032773	0.032471
<b>Bez závislosti</b>	6400	0.112317	0.112305	23579	0.033749	0.033796
<b>n=800000</b>						
<b>Se závislostí</b>	179	4.029610	4.102650	280	2.990420	2.957590

<b>Bez závislosti</b>	172	4.068720	4.087940	345	2.609300	2.626810
-----------------------	-----	----------	----------	-----	----------	----------

Tabulka 4 prezentuje nezanedbatelný vliv *loop unrolling* na dobu potřebnou ke zpracování smyčky. Lze také sledovat, jak náskok dále roste s vyšším stupněm rozbalování smyčky. Bližší informace jsou uvedeny v sekci 3.9.1.1.

Tabulka 4: Výsledky implementace zdrojového kódu 4

<b>GCC</b>						
	<b>O0</b>			<b>O2</b>		
	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>
<b>n=2000</b>						
<b>Suma</b>	100000	0.005824	0.005781	298667	0.002334	0.002302
<b>Suma 2x unroll</b>	248889	0.002862	0.002762	560000	0.001173	0.001172
<b>Suma 4x unroll</b>	344615	0.002382	0.002312	1000000	0.000592	0.000594
<b>n=30000</b>						
<b>Suma</b>	8960	0.089388	0.090681	19478	0.035896	0.036098
<b>Suma 2x unroll</b>	16593	0.043729	0.043317	37333	0.019078	0.018834
<b>Suma 4x unroll</b>	20364	0.045772	0.042968	74667	0.009114	0.009208
<b>n=800000</b>						
<b>Suma</b>	299	2.409830	2.403850	640	1.041110	1.049800
<b>Suma 2x unroll</b>	498	1.564720	1.537400	1000	0.526749	0.531250
<b>Suma 4x unroll</b>	320	2.222300	2.246090	1948	0.368726	0.368968
<b>Clang</b>						
	<b>O0</b>			<b>O2</b>		
	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>
<b>n=2000</b>						
<b>Suma</b>	100000	0.005468	0.005313	280000	0.002350	0.002344
<b>Suma 2x unroll</b>	248889	0.002722	0.002700	560000	0.001188	0.001200
<b>Suma 4x unroll</b>	448000	0.001603	0.001604	1120000	0.000601	0.000600
<b>n=30000</b>						
<b>Suma</b>	7467	0.086408	0.085794	19478	0.036800	0.036901
<b>Suma 2x unroll</b>	17920	0.041068	0.040981	37333	0.019074	0.019252
<b>Suma 4x unroll</b>	28000	0.025870	0.025670	74667	0.009427	0.009417
<b>n=800000</b>						
<b>Suma</b>	308	2.325740	2.333600	640	1.077740	1.074220
<b>Suma 2x unroll</b>	560	1.231520	1.227680	1120	0.598752	0.599888
<b>Suma 4x unroll</b>	1120	0.797982	0.795201	1600	0.436668	0.439453
<b>MSVC</b>						
	<b>O0</b>			<b>O2</b>		

	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=2000</b>						
<b>Suma</b>	100000	0.005473	0.005469	298667	0.002371	0.002354
<b>Suma 2x unroll</b>	235789	0.002822	0.002783	640000	0.001172	0.001172
<b>Suma 4x unroll</b>	448000	0.001591	0.001604	1120000	0.000596	0.000600
<b>n=30000</b>						
<b>Suma</b>	8960	0.089534	0.088937	19478	0.039054	0.038505
<b>Suma 2x unroll</b>	16186	0.049649	0.050198	34462	0.018420	0.018136
<b>Suma 4x unroll</b>	28000	0.025758	0.025670	74667	0.009151	0.009208
<b>n=800000</b>						
<b>Suma</b>	280	2.457370	2.455360	747	1.072300	1.066770
<b>Suma 2x unroll</b>	560	1.282680	1.283480	1120	0.535022	0.530134
<b>Suma 4x unroll</b>	747	0.815184	0.815763	1723	0.426264	0.426219

Tabulka 5 zaznamenává výsledky ukázku iterace polem s určitým krokem. Z výsledků je zřejmé, že ve všech případech, kdy je krok mocninou čísla dvě, dochází k degradaci výkonu iterace smyčkou v porovnání s číslem blízkým. Pro každý krok je počet přístupů do paměti totožný, a sice 10000. Jak roste krok, přirozeně se zvětšuje velikost iterovaného pole. I přesto se v některých případech krok 510 výkonově blíží kroku 64, ačkoliv pracují na různě velkých polích. Bližší informace jsou uvedeny v sekci 3.1.

Tabulka 5: Výsledky implementace zdrojového kódu 5

	GCC					
	O0			O2		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>Krok 30</b>	14544	0.047967	0.048345	44800	0.015145	0.015346
<b>Krok 32</b>	16593	0.043093	0.043317	34462	0.020374	0.020403
<b>Krok 60</b>	10000	0.051955	0.053125	37333	0.018398	0.018415
<b>Krok 64</b>	7467	0.133257	0.133923	10000	0.054394	0.053125
<b>Krok 120</b>	6400	0.090447	0.090332	40727	0.017791	0.017648
<b>Krok 128</b>	4978	0.146661	0.147524	8960	0.079215	0.076730
<b>Krok 250</b>	11200	0.078024	0.078125	28000	0.023816	0.023996
<b>Krok 256</b>	4073	0.166413	0.164958	7467	0.111273	0.110905
<b>Krok 510</b>	8960	0.077134	0.076730	14867	0.049476	0.049396
<b>Krok 512</b>	2800	0.283579	0.284598	8960	0.094164	0.094169
<b>Clang</b>						
	O0			O2		

	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>Krok 30</b>	24889	0.030523	0.030762	44800	0.015832	0.015695
<b>Krok 32</b>	18667	0.035627	0.035993	34462	0.020935	0.020856
<b>Krok 60</b>	20364	0.034083	0.034528	37333	0.017853	0.017578
<b>Krok 64</b>	11200	0.058433	0.058594	21333	0.039157	0.039551
<b>Krok 120</b>	18667	0.035990	0.035993	34462	0.018179	0.017683
<b>Krok 128</b>	4480	0.133772	0.132533	8960	0.079723	0.080218
<b>Krok 250</b>	18667	0.037055	0.036830	29867	0.025596	0.025635
<b>Krok 256</b>	4978	0.143369	0.141246	6400	0.110949	0.112305
<b>Krok 510</b>	15448	0.047575	0.047539	15448	0.045173	0.045516
<b>Krok 512</b>	4073	0.156649	0.153450	6400	0.091926	0.092773
<b>MSVC</b>						
<b>O0</b>				<b>O2</b>		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>Krok 30</b>	23579	0.033870	0.033796	44800	0.015700	0.015695
<b>Krok 32</b>	23579	0.036080	0.035784	34462	0.022156	0.022217
<b>Krok 60</b>	16725	0.031400	0.030830	37333	0.017593	0.017578
<b>Krok 64</b>	8960	0.065058	0.066267	17920	0.038149	0.037493
<b>Krok 120</b>	20364	0.035885	0.036062	37333	0.020320	0.020508
<b>Krok 128</b>	5600	0.122398	0.122768	8960	0.089561	0.087193
<b>Krok 250</b>	20364	0.036871	0.036830	29867	0.023736	0.023542
<b>Krok 256</b>	4978	0.144097	0.144385	6400	0.117186	0.117188
<b>Krok 510</b>	14933	0.044799	0.044993	15448	0.043739	0.044504
<b>Krok 512</b>	4480	0.162265	0.163923	7467	0.103638	0.102534

Tabulka 6 představuje výsledky různých způsobů iterace polem. Jak již bylo nastíněno v teoretické části, počítač dokáže rozpoznat pravidelné iterační vzory. Proto pravidelná změna iterační proměnné s malým krokem dosahuje nejlepších výsledků. Díky tomu lze totiž explotovat hardwarový prefetching a také prostorovou lokalitu. Počet přístupů do paměti je stejný, jako v předchozím případě. Bližší informace jsou uvedeny v sekci 3.4.

Tabulka 6: Výsledky implementace zdrojového kódu 6

	GCC					
	O0			O2		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>Sekvenční</b>	16593	0.044742	0.044258	64000	0.012063	0.012207
<b>Vzor</b>	7467	0.080983	0.077424	56000	0.012564	0.012556
<b>Krok 2</b>	14933	0.046413	0.043946	56000	0.013553	0.013672

<b>Krok 32</b>	9249	0.092866	0.092916	34462	0.020953	0.021310
<b>Krok 1024</b>	2635	0.259092	0.254981	5600	0.114843	0.114397
<b>Náhodný</b>	1545	0.384611	0.374191	4480	0.147870	0.146484
<b>Clang</b>						
<b>O0</b>				<b>O2</b>		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>Sekvenční</b>	24889	0.027092	0.026995	56000	0.012328	0.012277
<b>Vzor</b>	18667	0.036433	0.035993	56000	0.012214	0.012277
<b>Krok 2</b>	23579	0.027527	0.027832	56000	0.011948	0.011998
<b>Krok 32</b>	21333	0.030001	0.030030	37333	0.020366	0.020508
<b>Krok 1024</b>	4978	0.142663	0.144385	6400	0.110224	0.112305
<b>Náhodný</b>	4073	0.182676	0.184139	4480	0.147655	0.146484
<b>MSVC</b>						
<b>O0</b>				<b>O2</b>		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>Sekvenční</b>	24889	0.027496	0.027623	56000	0.012170	0.011998
<b>Vzor</b>	22400	0.031034	0.030692	56000	0.012578	0.012556
<b>Krok 2</b>	24889	0.027972	0.028250	56000	0.012413	0.012277
<b>Krok 32</b>	20364	0.030201	0.029924	34462	0.021451	0.021310
<b>Krok 1024</b>	4073	0.156119	0.157286	7467	0.126843	0.125552
<b>Náhodný</b>	1000	0.503371	0.500000	4480	0.167598	0.167411

Tabulka 7 ukazuje vliv uspořádání dat v datové struktuře na rychlosť běhu programu. Ačkoliv se v obou případech provádí ty samé výpočetní operace, jeden přístup je ve většině případů rychlejší než ten druhý. Díky separaci frekventovaně odkazovaných dat lépe využíváme kapacitu cache bloku, což umožní snížit počet odkazů do operační paměti, které mohou být velmi nákladné.

Tabulka 7: Výsledky implementace zdrojového kódu 7

	<b>GCC</b>					
	<b>O0</b>			<b>O2</b>		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=2000</b>						
<b>Klasický</b>	37333	0.017660	0.017160	298667	0.002443	0.002407
<b>Hot/cold</b>	49778	0.010256	0.010359	320000	0.001764	0.001758
<b>n=30000</b>						
<b>Klasický</b>	2133	0.264469	0.256388	11200	0.066862	0.065569
<b>Hot/cold</b>	4978	0.149977	0.150663	21333	0.035865	0.035157

	<b>n=800000</b>					
<b>Klasický</b>	64	11.754600	10.986300	112	6.326750	6.277900
<b>Hot/cold</b>	149	4.206700	4.194630	249	4.099590	4.016060
<b>Clang</b>						
	<b>O0</b>			<b>O2</b>		
	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>
<b>n=2000</b>						
<b>Klasický</b>	89600	0.007400	0.007324	298667	0.002269	0.002250
<b>Hot/cold</b>	100000	0.005391	0.005313	373333	0.001968	0.001967
<b>n=30000</b>						
<b>Klasický</b>	5600	0.114235	0.114397	11200	0.060423	0.059989
<b>Hot/cold</b>	7467	0.080343	0.079517	20364	0.032415	0.032226
<b>n=800000</b>						
<b>Klasický</b>	112	6.384970	6.417410	100	6.232610	6.250000
<b>Hot/cold</b>	224	2.965980	2.999440	236	3.518550	3.509000
<b>MSVC</b>						
	<b>O0</b>			<b>O2</b>		
	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>
<b>n=2000</b>						
<b>Klasický</b>	112000	0.006736	0.006696	263529	0.002399	0.002372
<b>Hot/cold</b>	100000	0.005613	0.005625	344615	0.002030	0.001995
<b>n=30000</b>						
<b>Klasický</b>	5600	0.118182	0.117188	11200	0.060231	0.059989
<b>Hot/cold</b>	8960	0.091771	0.090681	19478	0.036183	0.036098
<b>n=800000</b>						
<b>Klasický</b>	90	8.823340	8.680560	112	7.406960	7.393970
<b>Hot/cold</b>	224	3.184240	3.208710	224	2.935240	2.929690

Tabulka 8 demonstruje další vliv rozdílu uspořádání dat na výkon. V tomto případě se porovnává *array of structures* se *structure of arrays*. Důvod zrychlení je stejný, jako pro předchozí příklad.

Tabulka 8: Výsledky implementace zdrojového kódu 8

	<b>GCC</b>					
	<b>O0</b>			<b>O2</b>		
	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>
<b>n=2000</b>						
<b>AoS</b>	64000	0.011130	0.010742	320000	0.002342	0.002344

<b>SoA</b>	89600	0.007212	0.006975	448000	0.001366	0.001325
<b>n=30000</b>						
<b>AoS</b>	4480	0.163360	0.163923	11200	0.065206	0.065569
<b>SoA</b>	6400	0.109482	0.109863	32000	0.023603	0.023438
<b>n=800000</b>						
<b>AoS</b>	90	8.748240	8.506940	90	6.262720	6.250000
<b>SoA</b>	166	4.169010	4.141570	373	2.421480	2.303950
<b>Clang</b>						
<b>00</b>			<b>02</b>			
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=2000</b>						
<b>AoS</b>	89600	0.006702	0.006627	320000	0.002315	0.002246
<b>SoA</b>	112000	0.005343	0.005301	497778	0.001375	0.001350
<b>n=30000</b>						
<b>AoS</b>	6400	0.108680	0.109863	11200	0.060649	0.061384
<b>SoA</b>	8960	0.080381	0.080218	29867	0.022837	0.023019
<b>n=800000</b>						
<b>AoS</b>	112	6.338810	6.277900	75	7.897190	7.916670
<b>SoA</b>	280	2.540010	2.566960	373	1.817650	1.843160
<b>MSVC</b>						
<b>00</b>			<b>02</b>			
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=2000</b>						
<b>AoS</b>	89600	0.006721	0.006627	280000	0.002355	0.002400
<b>SoA</b>	112000	0.005526	0.005580	896000	0.000921	0.000924
<b>n=30000</b>						
<b>AoS</b>	6400	0.112860	0.112305	10000	0.058099	0.057813
<b>SoA</b>	8960	0.082518	0.081962	37333	0.018105	0.017997
<b>n=800000</b>						
<b>AoS</b>	90	6.693550	6.770830	112	6.167340	6.138390
<b>SoA</b>	264	2.601510	2.604170	448	1.743440	1.743860

Tabulka 9 obsahuje výsledky porovnání výpočtu sumy pole klasickým způsobem a s použitím SIMD instrukcí. Vektorový přístup provádí výpočet se dvěma prvky najedou místo jednoho, což lze pozorovat v porovnání vykonané práce. Ta se blíží dvojnásobku. Stále ovšem existuje určitá SIMD režie, která může nárůst výkonu limitovat. Pokud bychom mohli pracovat s datovým typem, který je menší, šlo by docílit ještě razantnějšího rozdílu v době běhu

programu. Rovněž lze pozorovat, že rozdíl mezi klasickou a vektorovou variantou je značnější při využití vyššího stupně optimalizace. Bližší informace jsou uvedeny v sekci 3.8.

Tabulka 9: Výsledky implementace zdrojového kódu 9

	GCC					
	O0			O2		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=2000</b>						
<b>Suma</b>	89600	0.007109	0.006975	298667	0.002349	0.002354
<b>Suma SIMD</b>	112000	0.004771	0.004743	560000	0.001276	0.001256
<b>n=30000</b>						
<b>Suma</b>	7467	0.092128	0.092072	18667	0.036989	0.036830
<b>Suma SIMD</b>	11200	0.071560	0.071150	37333	0.018362	0.018415
<b>n=800000</b>						
<b>Suma</b>	264	2.642930	2.663350	747	1.089000	1.087680
<b>Suma SIMD</b>	345	2.080920	2.083330	896	0.591000	0.592913
<b>Clang</b>						
	O0			O2		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=2000</b>						
<b>Suma</b>	112000	0.005379	0.005301	280000	0.002339	0.002344
<b>Suma SIMD</b>	165926	0.004263	0.004238	560000	0.001246	0.001256
<b>n=30000</b>						
<b>Suma</b>	7467	0.081255	0.079517	19478	0.035856	0.036098
<b>Suma SIMD</b>	11200	0.063047	0.062779	40727	0.018202	0.018415
<b>n=800000</b>						
<b>Suma</b>	320	2.249300	2.246090	640	1.012690	1.000980
<b>Suma SIMD</b>	407	1.756450	1.765970	1000	0.530646	0.531250
<b>MSVC</b>						
	O0			O2		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=2000</b>						
<b>Suma</b>	100000	0.005418	0.005469	298667	0.002367	0.002354
<b>Suma SIMD</b>	160000	0.004375	0.004395	560000	0.001249	0.001256
<b>n=30000</b>						
<b>Suma</b>	8960	0.083043	0.081962	20364	0.038496	0.038364
<b>Suma SIMD</b>	8960	0.063457	0.062779	40727	0.018297	0.018415
<b>n=800000</b>						
<b>Suma</b>	299	2.249860	2.247070	747	1.035730	1.024930

<b>Suma SIMD</b>	407	1.786230	1.765970	1000	0.533066	0.531250
------------------	-----	----------	----------	------	----------	----------

Tabulka 10 představuje vliv pořadí členských proměnných v datové struktuře na dobu vykonávání programu. Seřadí-li se podle velikosti, dosahuje se lepších výsledků. Z tohoto důvodu je dobré myslit na tuto potenciální optimalizaci, jelikož je velmi snadné ji provést.

Tabulka 10: Výsledky implementace zdrojového kódu 10

<b>Klasicky</b>	11200	0.070558	0.071150	23579	0.029729	0.029820
<b>Seřazeně</b>	11200	0.068243	0.068359	34462	0.019760	0.019496
<b>n=800000</b>						
<b>Klasicky</b>	249	2.887060	2.886550	249	2.860520	2.823800
<b>Seřazeně</b>	345	2.114200	2.083330	498	1.305330	1.317770

Tabulka 11 ukazuje různorodou povahu překladačů k jejich přístupu řešení větvení. První optimalizační stupeň naznačuje, že špatně předvídatelné větvení má vliv na výkon v případě přerostení velikosti L1. Naproti tomu druhý stupeň optimalizace demonstruje, jak špatná předvídatelnost větvení v porovnání s variantami s dobrou předvídatelností může mít negativní, žádný, nebo dokonce pozitivní vliv na rychlosť běhu programu. Návrh hardware pro řešení větví neustále prochází vývojem a je na tvůrcích překladačů, jak s ním naloží. Rovněž by mohlo být dosaženo zcela odlišných výsledků na různých architekturách. Bližší informace jsou uvedeny v sekci 3.9.1.2.

Tabulka 11: Výsledky implementace zdrojového kódu 11

GCC						
O0				O2		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=2000</b>						
<b>Nikdy</b>	112000	0.005484	0.005580	448000	0.001615	0.001639
<b>Vždy</b>	112000	0.006444	0.006278	448000	0.001688	0.001674
<b>x &gt; 5</b>	112000	0.005499	0.005441	448000	0.001635	0.001639
<b>n=30000</b>						
<b>Nikdy</b>	8960	0.079046	0.080218	28000	0.025361	0.025112
<b>Vždy</b>	6400	0.095181	0.095215	28000	0.026221	0.025670
<b>x &gt; 5</b>	2987	0.234673	0.230164	28000	0.025243	0.025112
<b>n=800000</b>						
<b>Nikdy</b>	264	2.356660	2.367420	747	0.833842	0.815763
<b>Vždy</b>	264	2.826350	2.781720	896	0.835550	0.837054
<b>x &gt; 5</b>	112	6.692410	6.696430	1120	0.757284	0.753348
Clang						
O0				O2		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=2000</b>						
<b>Nikdy</b>	149333	0.004680	0.004604	407273	0.001589	0.001573
<b>Vždy</b>	100000	0.005427	0.005469	448000	0.001630	0.001639
<b>x &gt; 5</b>	149333	0.005620	0.005650	448000	0.001596	0.001604

	<b>n=30000</b>					
<b>Nikdy</b>	8960	0.070663	0.069755	28000	0.023817	0.023438
<b>Vždy</b>	8960	0.081458	0.081962	28000	0.024931	0.025112
<b>x &gt; 5</b>	3200	0.213839	0.209961	29867	0.023978	0.024065
<b>n=800000</b>						
<b>Nikdy</b>	373	1.961200	1.968830	896	0.693744	0.697545
<b>Vždy</b>	299	2.327190	2.351590	896	0.741221	0.732422
<b>x &gt; 5</b>	112	5.770770	5.719870	896	0.704136	0.714983
<b>MSVC</b>						
	<b>00</b>			<b>02</b>		
	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>
<b>n=2000</b>						
<b>Nikdy</b>	203636	0.003357	0.003376	640000	0.001097	0.001099
<b>Vždy</b>	144516	0.004730	0.004649	497778	0.001380	0.001381
<b>x &gt; 5</b>	165926	0.004060	0.004049	640000	0.001172	0.001172
<b>n=30000</b>						
<b>Nikdy</b>	10000	0.051328	0.051563	40727	0.016218	0.015730
<b>Vždy</b>	8960	0.071835	0.071498	32000	0.020576	0.020508
<b>x &gt; 5</b>	3733	0.196489	0.196725	7467	0.104737	0.104627
<b>n=800000</b>						
<b>Nikdy</b>	498	1.447820	1.443270	1000	0.495667	0.500000
<b>Vždy</b>	345	2.012090	1.992750	1120	0.727234	0.739397
<b>x &gt; 5</b>	112	5.620060	5.580360	187	3.669930	3.676470

Tabulka 12 představuje vliv falešného sdílení při vícejádrových výpočtech. Lze pozorovat, že prosté zarovnání dat v paměti může mít nezanedbatelný vliv na výkon. V tomto případě  $n$  označuje počet iterací hlavní smyčky programu. Bližší informace jsou uvedeny v sekci 3.9.3.

Tabulka 12: Výsledky implementace zdrojového kódu 12

	<b>GCC</b>					
	<b>00</b>			<b>02</b>		
	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>	<b>Počet iterací</b>	<b>Čas [ms]</b>	<b>Procesorový čas [ms]</b>
<b>n=1048576</b>						
<b>Pravé sdílení</b>	10	195.374000	85.937500	16	99.129200	47.851600
<b>Falešné sdílení</b>	90	13.397800	10.243100	264	3.300750	3.018470
<b>Oprava FS</b>	373	2.493230	2.262060	407	1.780200	1.727580
<b>Clang</b>						
	<b>00</b>			<b>02</b>		

	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=1048576</b>						
<b>Pravé sdílení</b>	10	150.570000	71.875000	14	99.927900	39.062500
<b>Falešné sdílení</b>	75	12.747600	10.416700	204	3.371230	2.833950
<b>Oprava FS</b>	407	1.835190	1.765970	373	1.792620	1.717490
<b>MSVC</b>						
<b>00</b>			<b>02</b>			
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=1048576</b>						
<b>Pravé sdílení</b>	4	191.076000	191.406000	5	137.905000	140.625000
<b>Falešné sdílení</b>	56	12.411800	12.276800	112	4.761430	4.743300
<b>Oprava FS</b>	249	2.792380	2.823800	407	1.720780	1.689190

Tabulka 13 uzavírá sekci měření výkonu ukázkou vlivu mikroarchitektury na rychlosť běhu programu. Z výsledků je zřejmé, že nárůst výkonu u pomalejší varianty programu mezi stupni optimalizace není srovnatelný s variantou rychlejší. Program je totiž limitován samotným hardwarem, který kvůli posloupnosti instrukcí a z nich pramenící datové závislosti nemůže pracovat v paralelním režimu. V tomto případě  $n$  označuje počet iterací hlavní smyčky programu. Blížší informace jsou uvedeny v sekci 3.3.6.

Tabulka 13: Výsledky implementace zdrojového kódu 13

	<b>GCC</b>					
	<b>00</b>			<b>02</b>		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=1048576</b>						
<b>Zápis čtení</b>	280	2.17556	2.12054	2240	0.313822	0.313895
<b>Zápis čtení alias</b>	90	7.82592	7.98611	112	5.307320	5.301340
	<b>Clang</b>					
	<b>00</b>			<b>02</b>		
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=1048576</b>						
<b>Zápis čtení</b>	320	2.157670	2.148440	2240	0.317840	0.313895
<b>Zápis čtení alias</b>	90	7.814450	7.812500	100	5.346000	5.312500
	<b>MSVC</b>					
	<b>00</b>			<b>02</b>		

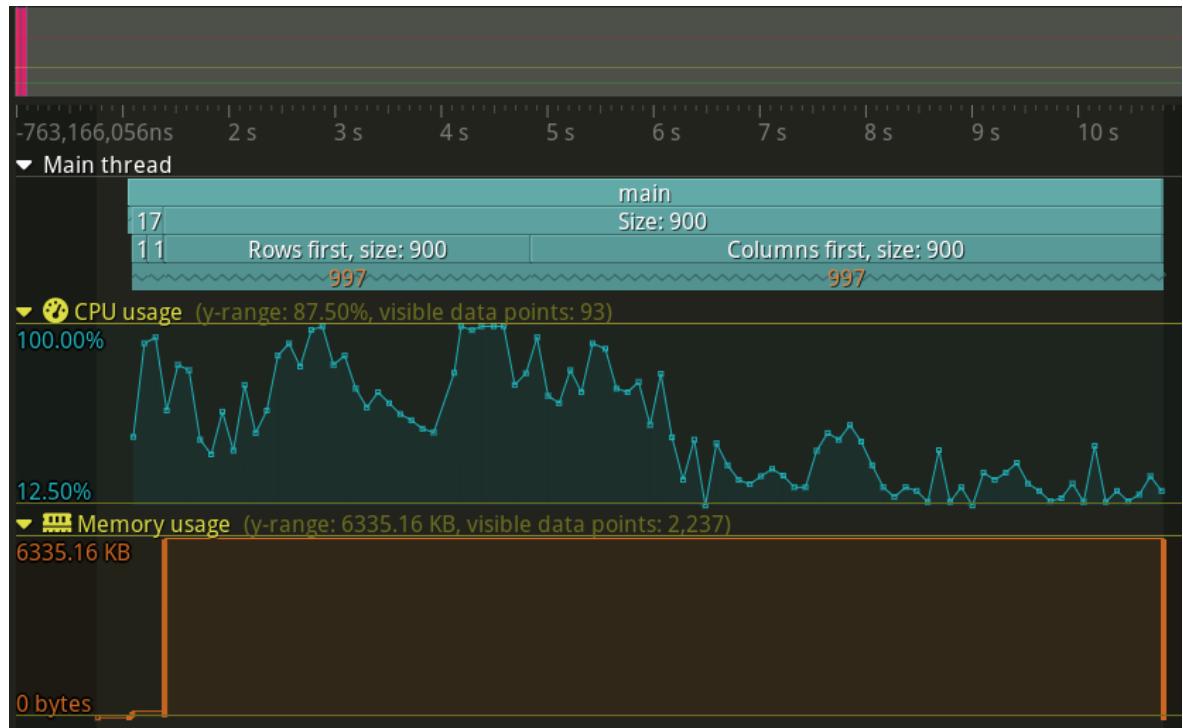
	Počet iterací	Čas [ms]	Procesorový čas [ms]	Počet iterací	Čas [ms]	Procesorový čas [ms]
<b>n=1048576</b>						
Zápis čtení	345	2.140600	2.128620	747	0.883918	0.878514
Zápis čtení alias	75	7.855130	7.708330	100	5.366580	5.468750

## 7.2 Analýza ukázkových příkladů profilovacími nástroji

Vybrané ukázkové příklady byly dále upraveny do podoby, která dokáže poskytnout požadované informace profilovacím nástrojům. Tyto příklady byly sestaveny překladačem MSVC s optimalizačním stupněm *O0* z důvodu potřeby zachování dostupnosti zdrojového kódu v podobě jazyka symbolických adres a poskytnutí požadovaných dat pro vzorkovací profiler.

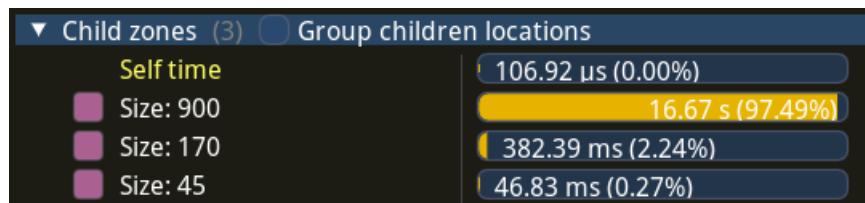
### 7.2.1 Zdrojový kód 1

Instrumentační profiler Tracy zobrazuje průběh programu pro iteraci dvouozměrného pole po sloupcích a po řádcích v Obr. 43. Velikosti polí jsou stejné jako v tabulce 1. Aplikace nám umožňuje graficky znázornit využití CPU a také alokované a uvolněné paměti.



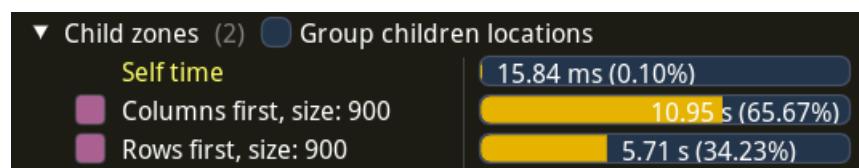
Obr. 43: Program Tracy – analýza zdrojového kódu 1. [Zdroj vlastní]

Implementace příkladu určeného pro profilování je členěna podle velikost dat, na kterých se pracuje. Obr. 44 vyčítá celkový čas potřebný pro vykonání celé části.



Obr. 44: Profilování částí podle velikosti dat. [Zdroj vlastní]

U každé části pak lze vidět rozdíl v době trvání jednotlivých přístupů. Aby měl profilovací nástroj dostatečné množství dat a aby se snáze připojil k profilované aplikaci, provádí se sledovaná metoda opakováně ve smyčce.



Obr. 45: Pohled na srovnání doby běhu pro velikost 900x900. [Zdroj vlastní]

Obr. 46 ukazuje, že průměrný čas exekuce funkce pro iteraci po řádcích zhruba odpovídá hodnotám naměřeným benchmarkovacím nástrojem.

<b>45x45</b>	Group mean: 16.92 µs	Group median: 6.58 µs
<b>170x170</b>	Group mean: 80.66 µs	Group median: 77.82 µs
<b>900x900</b>	Group mean: 2.99 ms	Group median: 2.51 ms

Obr. 46: Průměrná doba běhu iterace po řádcích. [Zdroj vlastní]

Tracy rovněž poskytuje funkci analýzy programu v podobě jazyka symbolických adres a pomáhá identifikovat potenciálně problémová místa v aplikaci. Obr. 47 zobrazuje tělo hlavní smyčky programu. V obou případech se využívá identických instrukcí, v jednom případě je stráveno více času u instrukce pro převod dat z registru do paměti.

**Po řádcích**

0.10%	+253	movsxsd	rax, dword ptr [rsp + 0x90]
0.01%	+261	movsxsd	rcx, dword ptr [rsp + 0x94]
8.11%	+269	mov	rdx, qword ptr [rip + 0x53a5c]
0.11%	+276	mov	rax, qword ptr [rdx + rax*8]
0.40%	+280	movsd	xmm0, qword ptr [rsp + 0x88]
17.00%	+289	addsd	xmm0, qword ptr [rax + rcx*8]
56.43%	+294	movsd	qword ptr [rsp + 0x88], xmm0

**Po sloupcích**

0.03%	+253	movsxsd	rax, dword ptr [rsp + 0x94]
0.00%	+261	movsxsd	rcx, dword ptr [rsp + 0x90]
3.47%	+269	mov	rdx, qword ptr [rip + 0x538bc]
0.08%	+276	mov	rax, qword ptr [rdx + rax*8]
0.27%	+280	movsd	xmm0, qword ptr [rsp + 0x88]
5.99%	+289	addsd	xmm0, qword ptr [rax + rcx*8]
82.18%	+294	movsd	qword ptr [rsp + 0x88], xmm0

Obr. 47: Tělo hlavní smyčky obou variant v jazyce symbolických adres. [Zdroj vlastní]

Tento jev je způsoben čekáním na obsluhu cache miss. Potvrzuje to i profilovací nástroj vTune v Obr. 48. Jedná se o informace získané z PMU monitorující události týkající se vyrovnávací paměti.

**Po řádcích**

MEM_LOAD_RETIREDFB_HIT_PS	63,001,890
MEM_LOAD_RETIREDL1_HIT_PS	35,220,052,830
MEM_LOAD_RETIREDL1_MISS_PS	<u>9,000,270</u>
MEM_LOAD_RETIREDL2_HIT_PS	3,000,090
MEM_LOAD_RETIREDL3_HIT_PS	1,500,630
MEM_LOAD_RETIREDL3_MISS_PS	<u>3,000,210</u>

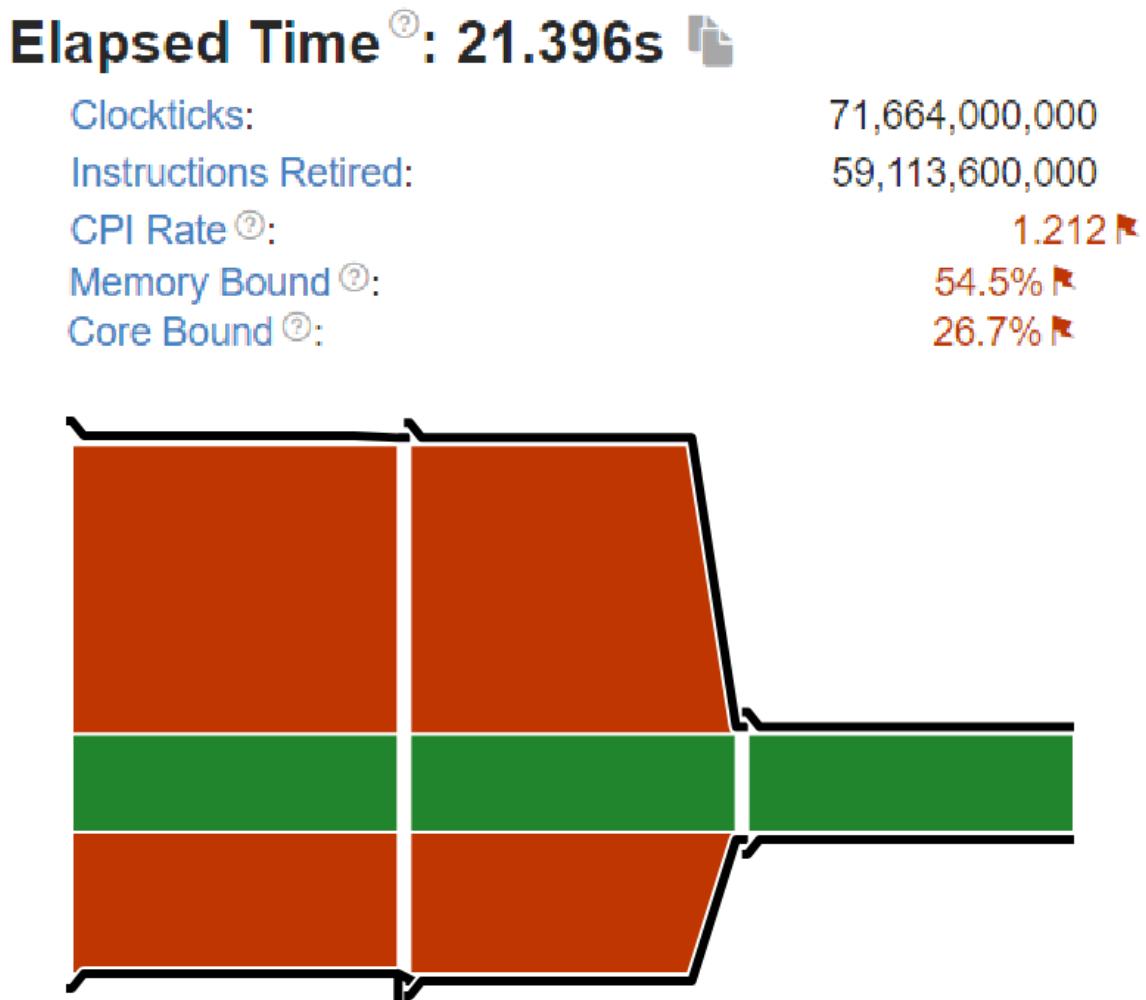
**Po sloupcích**

MEM_LOAD_RETIREDFB_HIT_PS	33,000,990
MEM_LOAD_RETIREDL1_HIT_PS	34,440,051,660
MEM_LOAD_RETIREDL1_MISS_PS	<u>3,783,113,490</u>
MEM_LOAD_RETIREDL2_HIT_PS	2,913,087,390
MEM_LOAD_RETIREDL3_HIT_PS	645,270,900
MEM_LOAD_RETIREDL3_MISS_PS	<u>180,012,600</u>

Obr. 48: Srovnání hardwarových událostí obou variant. [Zdroj vlastní]

Obr. 49 je shrnutí, které poskytuje vTune pro iteraci po sloupcích. Mimo detailního popisu jednotlivých částí pipeline lze pozorovat také grafickou reprezentaci efektivity programu.

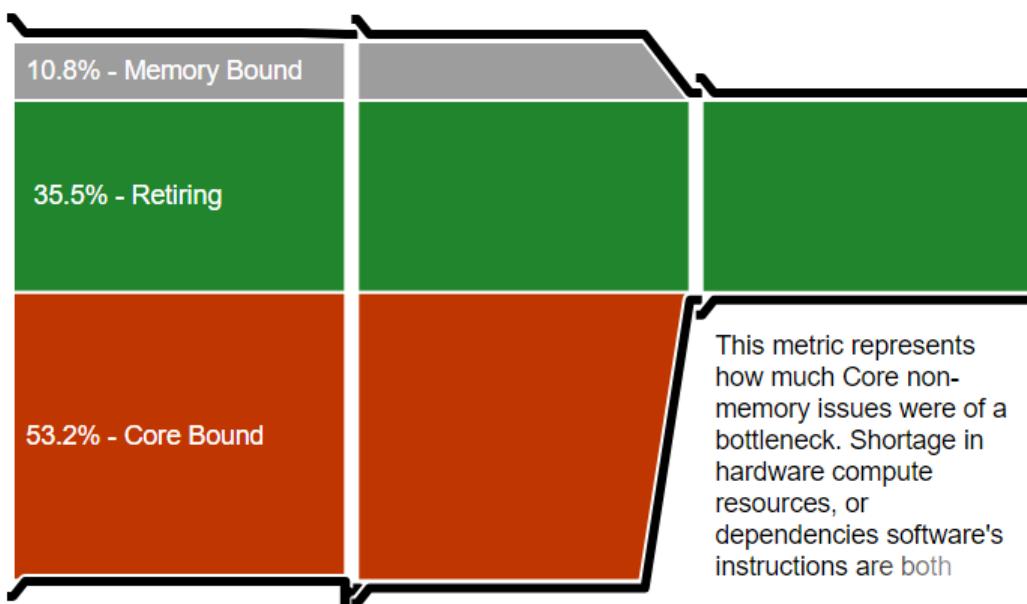
Při porovnání Obr. 50 s Obr. 49, je rozdíl v omezující oblasti zjevný. Iterace po řádcích eliminuje *memory bound* a následně nalézá další příležitost k potenciální optimalizaci, tedy *core bound*. Příznivou informací je, že zelená plocha znázorňující počet vykonalých instrukcí se zvětšila.



Obr. 49: Shrnutí výkonnostních charakteristik programu vTune pro variantu po sloupcích. [Zdroj vlastní]

## Elapsed Time <sup>?</sup>: 12.291s

Clockticks:	40,856,000,000
Instructions Retired:	59,025,600,000
CPI Rate <sup>?</sup> :	0.692
Memory Bound <sup>?</sup> :	10.8%
Core Bound <sup>?</sup> :	53.2% 



Obr. 50: Shrnutí výkonnostních charakteristik programu vTune pro variantu po řádcích. [Zdroj vlastní]

### 7.2.2 Zdrojový kód 4

Obr. 51 ukazuje tělo hlavní smyčky programu. Lze pozorovat, že při zvyšování stupně unrollingu se zvyšuje počet instrukcí pro provádění samotného výpočtu a přesun dat, a tím pádem se snižuje počet režijních instrukcí pro implementaci smyčky.

### Suma hodnot pole

0.30%	+195	movsx	rax, dword ptr [rsp + 0x74]
0.21%	+200	mov	rcx, qword ptr [rsp + 0xb0]
8.95%	+208	mov	rdx, qword ptr [rsp + 0xa0]
0.09%	+216	movsd	xmm0, qword ptr [rcx]
16.74%	+220	addsd	xmm0, qword ptr [rdx + rax*8]
55.45%	+225	mov	rax, qword ptr [rsp + 0xb0]
0.46%	+233	movsd	qword ptr [rax], xmm0

### Suma hodnot pole, 2x unrolling

0.03%	+242	movsx	rax, dword ptr [rsp + 0x94]
	+250	mov	ecx, dword ptr [rsp + 0x94]
7.86%	+257	inc	ecx
0.08%	+259	movsx	rcx, ecx
0.03%	+262	mov	rdx, qword ptr [rsp + 0xd0]
0.01%	+270	mov	rdi, qword ptr [rsp + 0xd0]
8.32%	+278	movsd	xmm0, qword ptr [rdx + rax*8]
22.83%	+283	addsd	xmm0, qword ptr [rdi + rcx*8]
3.36%	+288	movsd	xmm1, qword ptr [rsp + 0xa0]
2.61%	+297	addsd	xmm1, xmm0
39.33%	+301	movaps	xmm0, xmm1
0.21%	+304	movsd	qword ptr [rsp + 0xa0], xmm0

### Suma hodnot pole, 4x unrolling

0.05%	+243	movsx	rax, dword ptr [rsp + 0x94]
0.01%	+251	mov	ecx, dword ptr [rsp + 0x94]
6.50%	+258	inc	ecx
0.03%	+260	movsx	rcx, ecx
0.03%	+263	mov	rdx, qword ptr [rsp + 0xd0]
0.03%	+271	mov	rdi, qword ptr [rsp + 0xd0]
7.05%	+279	movsd	xmm0, qword ptr [rdx + rax*8]
0.64%	+284	addsd	xmm0, qword ptr [rdi + rcx*8]
2.22%	+289	mov	eax, dword ptr [rsp + 0x94]
0.05%	+296	add	eax, 2
5.93%	+299	cdqe	
0.03%	+301	mov	rcx, qword ptr [rsp + 0xd0]
0.64%	+309	addsd	xmm0, qword ptr [rcx + rax*8]
22.63%	+314	mov	eax, dword ptr [rsp + 0x94]
5.49%	+321	add	eax, 3
0.01%	+324	cdqe	
	+326	mov	rcx, qword ptr [rsp + 0xd0]
1.35%	+334	addsd	xmm0, qword ptr [rcx + rax*8]
12.11%	+339	movsd	xmm1, qword ptr [rsp + 0xa0]
0.07%	+348	addsd	xmm1, xmm0
21.45%	+352	movaps	xmm0, xmm1
0.05%	+355	movsd	qword ptr [rsp + 0xa0], xmm0

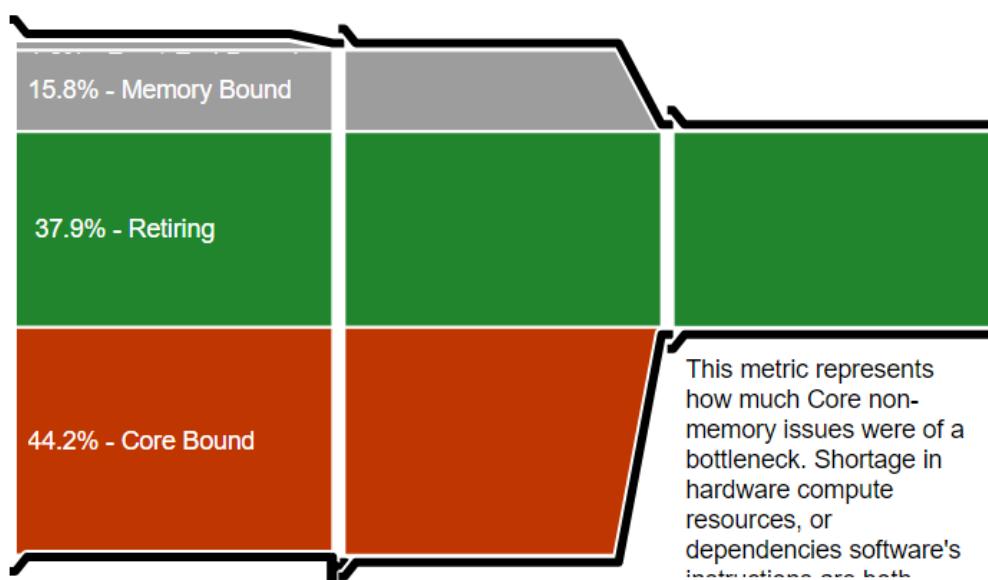
Obr. 51: Tělo hlavní smyčky různých stupňů unrollingu v jazyce symbolických adres. [Zdroj vlastní]

Nahlédnutí na Obr. 52, Obr. 53 a Obr. 54 nám poskytuje znázornění růstu výkonu při zvyšování stupně unrollingu. Nejvýznamnější změna je v rostoucím počtu vykonalých instrukcí. Dále si lze všimnout, že procentuální hodnota u pole *back-end bound* s rostoucím stupněm unrollingu klesá. Podle vysvětlivek programu nižší hodnoty znamenají lepší využití

funkčních jednotek procesoru. Do určitého bodu má smysl zvyšovat stupeň unrollingu, pak je ovšem riziko naražení na limity hardware a možnou degradaci výkonu aplikace.

## Elapsed Time <sup>?</sup>: 10.984s

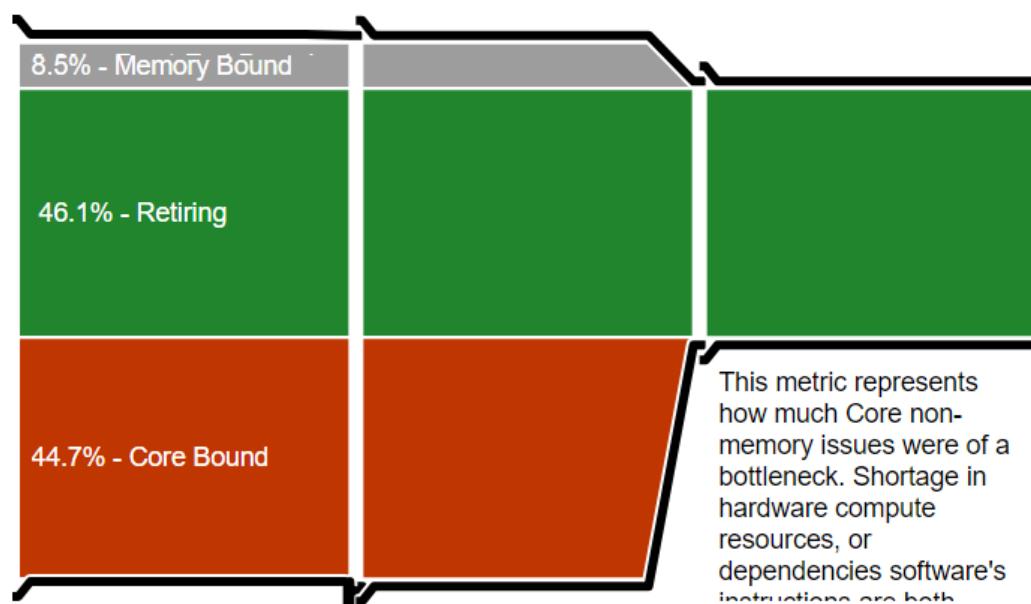
Clockticks:	36,284,800,000
Instructions Retired:	52,750,400,000
CPI Rate <sup>?</sup> :	0.688
Retiring <sup>?</sup> :	37.9%
Back-End Bound <sup>?</sup> :	59.9% 



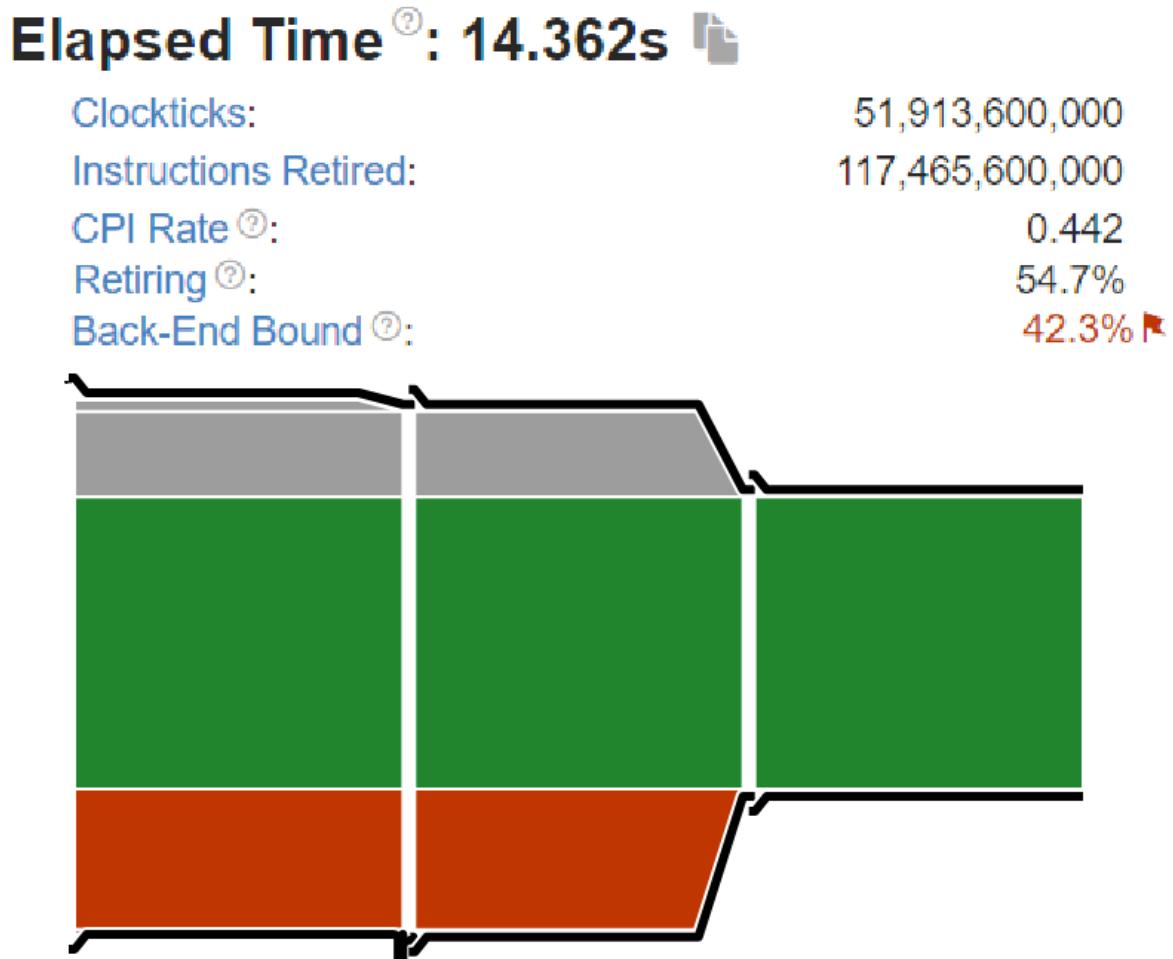
Obr. 52: Shrnutí výkonnostních charakteristik programu vTune pro variantu bez unrollingu. [Zdroj vlastní]

## Elapsed Time (?): 16.996s

Clockticks:	16,326,400,000
Instructions Retired:	32,635,200,000
CPI Rate <small>(?)</small> :	0.500
Retiring <small>(?)</small> :	46.1%
Back-End Bound <small>(?)</small> :	53.2% 



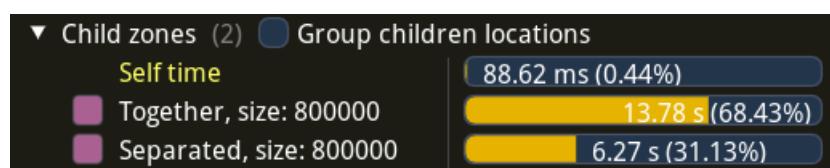
Obr. 53: Shrnutí výkonnostních charakteristik programu vTune pro variantu s dvojnásobným unrollingem. [Zdroj vlastní]



Obr. 54: Shrnutí výkonnostních charakteristik programu vTune pro variantu s čtyřnásobným unrollingem. [Zdroj vlastní]

### 7.2.3 Zdrojový kód 7

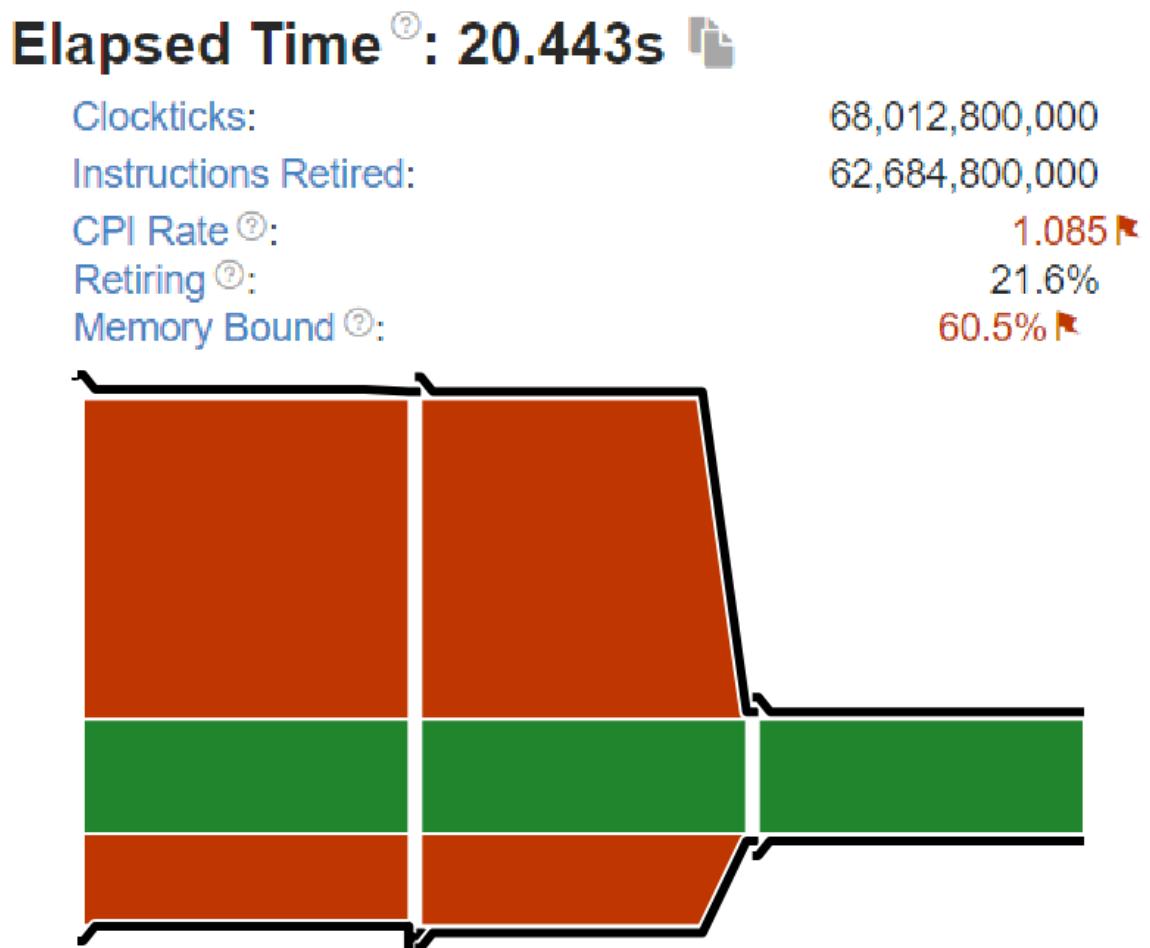
Obr. 55 upozorňuje na to, jak může rozložení dat v datové struktuře, a tím pádem v operační paměti, zásadně ovlivnit výkon programu. Při dostatečně velkém objemu dat lze pozorovat, že je možné vykonat stejně množství práce za poloviční čas.



Obr. 55: Pohled na srovnání doby běhu pro velikost 800000. [Zdroj vlastní]

Při porovnání Obr. 56, zobrazující výsledek klasického rozložení, a Obr. 57, zobrazující výsledek separace *hot* a *cold* dat vidíme, jak se zvýšil počet vykonaných instrukcí, snížil se počet potřebných cyklů CPU na jednu instrukci a snížilo se procento *memory bound*. Jelikož

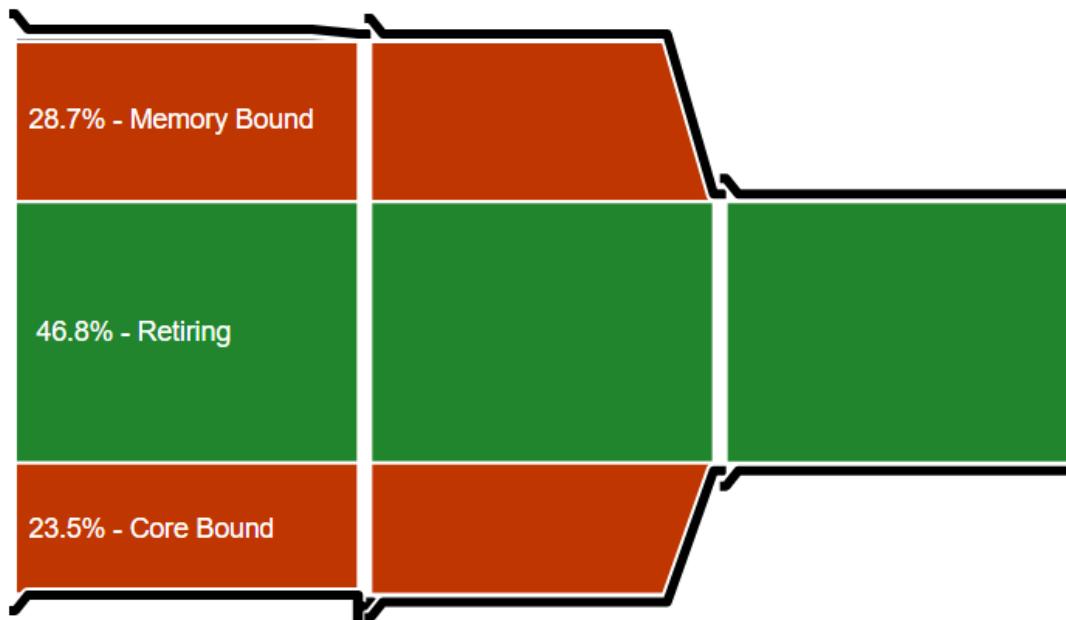
se potřebná data většinou nacházejí v cache paměti, sníží se počet odkazů do operační paměti.



Obr. 56: Shrnutí výkonnostních charakteristik programu vTune pro klasickou variantu. [Zdroj vlastní]

## Elapsed Time : 9.329s

Clockticks:	31,132,800,000
Instructions Retired:	62,632,000,000
CPI Rate  :	0.497
Retiring  :	46.8%
Memory Bound  :	28.7% 



Obr. 57: Shrnutí výkonnostních charakteristik programu vTune pro variantu se separovanými daty. [Zdroj vlastní]

Obr. 58 zobrazuje rozdíl v počtu zastavení exekuční pipeline z různých důvodů. Celkový počet zastavení je několikanásobně snížen při využití separace dat. Je možné konstatovat, že cache miss jsou důvodem zvýšeného počtu zastavení pipeline.

Klasicky	Hot/cold
CYCLE_ACTIVITY.CYCLES_L1D_MISS	58,800,088,200
CYCLE_ACTIVITY.CYCLES_MEM_ANY	67,380,101,070
CYCLE_ACTIVITY.STALLS_L1D_MISS	36,600,054,900
CYCLE_ACTIVITY.STALLS_L2_MISS	13,800,020,700
CYCLE_ACTIVITY.STALLS_L3_MISS	9,240,013,860
CYCLE_ACTIVITY.STALLS_MEM_ANY	37,260,055,890
CYCLE_ACTIVITY.STALLS_TOTAL	37,740,056,610
	CYCLE_ACTIVITY.CYCLES_L1D_MISS 9,180,013,770
	CYCLE_ACTIVITY.CYCLES_MEM_ANY 31,380,047,070
	CYCLE_ACTIVITY.STALLS_L1D_MISS 6,120,009,180
	CYCLE_ACTIVITY.STALLS_L2_MISS 300,000,450
	CYCLE_ACTIVITY.STALLS_L3_MISS 120,000,180
	CYCLE_ACTIVITY.STALLS_MEM_ANY 7,080,010,620
	CYCLE_ACTIVITY.STALLS_TOTAL 6,780,010,170

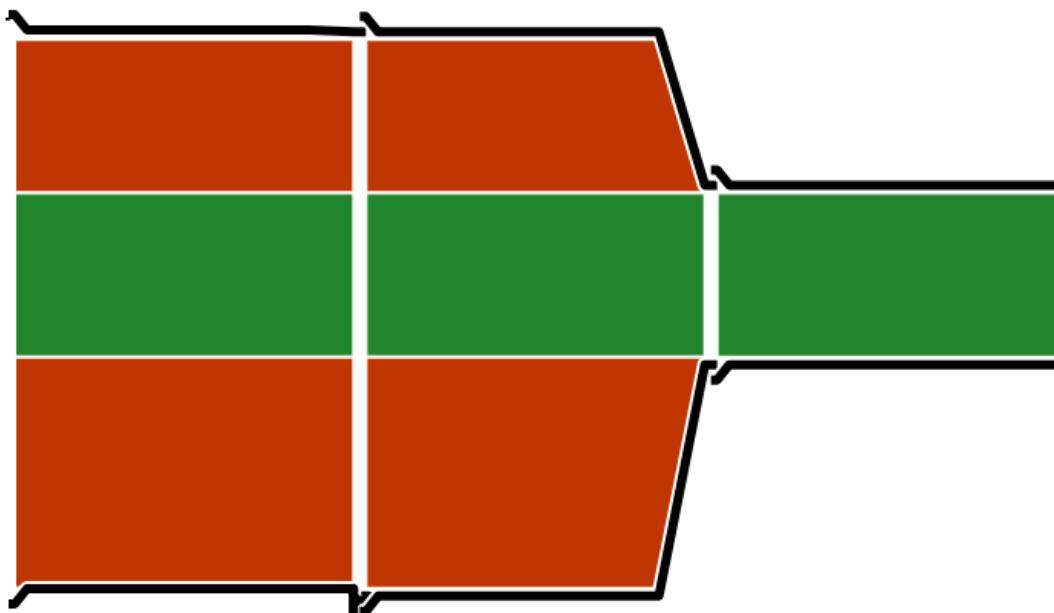
Obr. 58: Srovnání hardwarových událostí obou variant. [Zdroj vlastní]

#### 7.2.4 Zdrojový kód 9

Při prozkoumání Obr. 59 s výsledky klasické sumy hodnot v poli a Obr. 60 s výsledky sumy hodnot pole s využitím SIMD instrukcí je možné pozorovat nepatrné rozdíly. Překvapující informací je snížený počet vykonaných instrukcí. Je však nutné si uvědomit, že SIMD přístup zpracovává více dat pomocí jedné operace. Dále je možné si všimnout, že Obr. 60 uvádí zvýšení utilizace výpočetních jednotek pro vektorové operace, z 25 % na 50 %.

### Elapsed Time ?: 44.706s

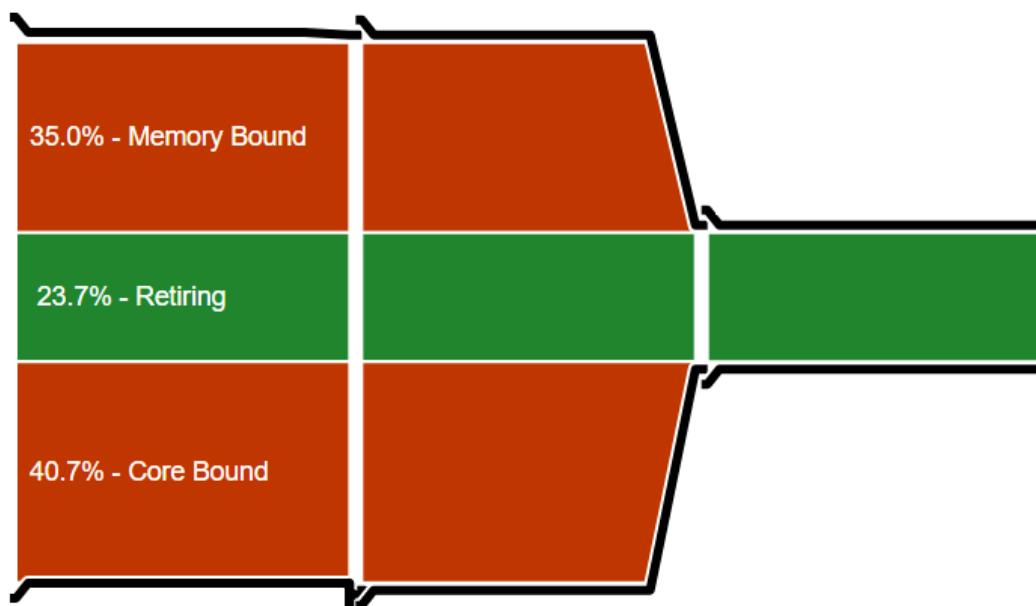
Clockticks:	149,134,400,000
Instructions Retired:	189,977,600,000
CPI Rate <small>?</small> :	0.785
Retiring <small>?</small> :	29.8%
Vector Capacity Usage (FPU) <small>?</small> :	25.0% 



Obr. 59 Shrnutí výkonnostních charakteristik programu vTune pro skalární vari-  
antu. [Zdroj vlastní]

## Elapsed Time <sup>(?)</sup>: 35.394s

Clockticks:	118,145,600,000
Instructions Retired:	118,774,400,000
CPI Rate <sup>(?)</sup> :	0.995
Retiring <sup>(?)</sup> :	23.7%
Vector Capacity Usage (FPU) <sup>(?)</sup> :	50.0% 



Obr. 60: Shrnutí výkonnostních charakteristik programu vTune pro vektorovou variantu. [Zdroj vlastní]

Porovnání těla hlavní smyčky programů v Obr. 61 lze pozorovat, že při využití SIMD intrinsicských funkcí skutečně dojde k převodu na odpovídající vektorové instrukce. Lze pozorovat instrukci *addpd*, která seče dva vektorové registry obsahující dvě hodnoty datového typu *double*. Následující instrukce *movaps* představují režii SIMD operací, jejichž následkem může být nedosažení očekávaného zrychlení. Toho by podle tabulky 9 mohlo být dosaženo použitím vyššího stupně optimalizace překladače.

## Suma hodnot pole

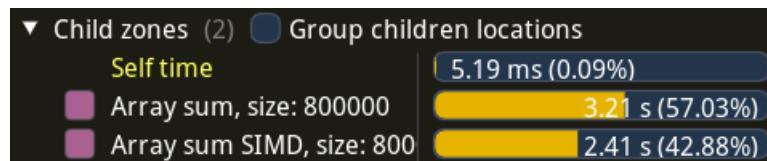
0.08%	+208	movsxd	rax, dword ptr [rsp + 0x90]
0.01%	+216	mov	rcx, qword ptr [rip + 0x53ab1]
9.23%	+223	movsd	xmm0, qword ptr [rsp + 0x88]
14.16%	+232	addsd	xmm0, qword ptr [rcx + rax*8]
57.49%	+237	movsd	qword ptr [rsp + 0x88], xmm0

## Suma hodnot pole SIMD

0.01%	+322	movsxd	rax, dword ptr [rsp + 0xcc]
	+330	imul	rax, rax, 0x10
6.48%	+334	mov	rcx, qword ptr [rsp + 0xc0]
0.02%	+342	movups	xmm0, xmmword ptr [rcx + rax]
7.00%	+346	addpd	xmm0, xmmword ptr [rsp + 0xa0]
39.82%	+355	movaps	xmmword ptr [rsp + 0xd0], xmm0
6.38%	+363	movaps	xmm0, xmmword ptr [rsp + 0xd0]
20.91%	+371	movaps	xmmword ptr [rsp + 0xa0], xmm0

Obr. 61: Tělo hlavní smyčky skalární a vektorové varianty v jazyce symbolických adres. [Zdroj vlastní]

Je vhodné si připomenout, že tou nejvýznamnější metrikou pro posouzení úspěchu optimalizace je doba trvání vykonávání programu. Obr. 62 potvrzuje, že tohoto cíle bylo dosaženo. Je však potřeba zamyslet se nad tím, kolik úsilí programátor obětuje takovéto netriviální optimalizaci. Tato nemalá investice by měla poskytnout odpovídající zrychlení.



Obr. 62: Pohled na srovnání doby běhu pro velikost 800000. [Zdroj vlastní]

## 8 SADA DOPORUČENÍ

Následující sekce představí doporučení pro programátory, kteří by chtěli aplikovat datově orientovaný přístup při tvorbě software.

### 8.1 Obecná doporučení

- **Využít DOP tam, kde dává smysl**

Ačkoliv výhoda DOP spočívá v aplikování různých prostředků pro tvorbu software a členění programu zejména s cílem dosáhnout lepšího výkonu, nemusí to být vhodné všude. Pomocí profilovacích nástrojů lze identifikovat problémová místa existující aplikace, a po zvážení všech dopadů připadá přepis tímto způsobem v úvahu. Pokud by vývojář začínal s aplikováním DOP principů již od začátku projektu, mohl by zjistit, že iniciální část návrhu je v porovnání s ostatními paradigmaty poskytuje hmatatelné výsledky později.

- **Využít OOP tam, kde dává smysl**

Jelikož se jedná o jedno z nejrozšířenějších paradigmat ve světě software, v některých případech není možné se vyhnout tomuto způsobu myšlení. Například při využívání knihoven třetích stran. Rovněž může být tento přístup vhodnější, pokud chce programátor rychle zhmatnit svou ideu a zjistit, zda má smysl se jí dále věnovat. Jelikož mohou tyto dvě paradigmata koexistovat ve stejně aplikaci, není třeba jedno z nich zcela vyřadit.

- **Profilovat**

Profilování umožňuje softwarovým vývojářům identifikovat místa aplikace s možností potenciálního zlepšení. Dovoluje tvůrcům software lépe zaměřit své úsilí a nemarnit čas nad optimalizacemi nesoucími pouze marginální výkonové přírůstky.

- **Testovat**

Jakmile profilovací nástroj odhalí problémovou funkci, je vhodné pro ni vytvořit sadu jednotkových testů. Na základě reálného používání aplikace si nechat napovědět daty a specifikovat množinu vstupních testovacích dat od běžně používaných, přes okrajové hodnoty, až po hodnoty vyvolávající kritické situace. Při optimalizaci kódu se možné se setkat s nepředvídatelnými situacemi, kdy překladač aplikuje tak agresivní optimalizační metody, že se může očekávané chování funkce změnit. Na to lze reagovat pomocí testů

- **Porozumět hardware**

Rozsáhlá část DOP se věnuje kooperací software s hardware a snahou využít jeho plný potenciál. Není na škodu se seznámit s hardwarem, pro který programátor vyvíjí prostřednic-tvím studia dokumentace.

- **Porozumět datům**

Primárním cílem programů je pracovat s daty a transformovat je z jedné podoby do druhé. Sama o sobě nemají žádný kontext. Ten jim je udělen až v případě jejich použití. Vývojář by se měl snažit zjistit, která část programu nastává statisticky nejčastěji a na tu se zaměřit. Čím více dat je k dispozici, tím lépe může programátor uvažovat o problému. [3]

- **Lépe komunikovat s překladačem**

Tvorbu software si lze představit jako formu komunikace s překladačem. Vývojář se mu snaží sdělit, jaký problém chce vyřešit, a z tohoto popisu se překladač snaží co nejlépe poskládat posloupnost instrukcí. Čím více kontextu překladač má, tím lépe dokáže program přeložit. Aby člověk pochopil, jak lépe komunikovat s překladačem, je vhodné využít prostředky pro zkoumání přeloženého kódu do jazyka symbolických adres a poté se snažit v preferovaném vysokoúrovňovém programovacím jazyce nasimulovat posloupnosti instrukcí. Aby překladač fungoval spolehlivě, v některých situacích musí uvažovat široké množství možností, které by mohly nastat a přizpůsobit se té nejvíce omezující. Jako následek toho může dojít ke snížení množství aplikovatelných optimalizací. Snížením počtu možností, které by mohly nastat, pomocí specifikace kontextu, jako například zarovnání proměnných nebo předem známý počet elementů v poli, lze zvýšit počet aplikovatelných optimalizací.

- **Využívat optimalizačních stupňů překladače**

Většina překladačů disponuje možností výběru úrovně optimalizace ať už pro zlepšení výkonu či dosažení menší velikosti spustitelného souboru. Jedná se o jednoduše aplikovatelné vylepšení.

- **Profilovat znovu**

Po aplikování optimalizací je třeba verifikovat, zda došlo k dosažení původního cíle. Opět by měl programátor sledovat zásadní metriku, kterou je celková doba běhu programu.

## 8.2 Doporučení pro optimalizace

- **Loop unrolling**

Pro zrychlení provádění smyčky je dobrou optimalizační technikou loop unrolling. Některé překladače se ji mohou pokusit aplikovat na úrovni jazyka symbolických adres, nicméně pro zaručení implementace tohoto vylepšení je vhodné tuto techniku použít na úrovni programovacího jazyka. Je třeba myslit na to, že existuje stupeň unrollingu, nad který už nemá smysl jít, jelikož dojde k naražení na limity hardware.

- **Stanovení vhodného kroku**

Tabulka 5 dokazuje, že nevhodně zvolený krok při iteraci polem může významně degradovat dobu běhu programu. Pokud to nevyžaduje logika programu, je dobrý nápad vyhnout se kroku, který je mocninou čísla dvě.

- **Organizovat členské proměnné v datové struktuře / třídě**

V rámci ukázkových příkladů se několik z nich zabývalo vlivem uspořádání členských proměnných v datové struktuře na dobu trvání programu. Vzhledem k tomu, že se jedná o nepatrny zásah a není třeba do značné míry upravovat logiku programu, stojí za to zvážit správnou organizaci. Kvůli požadavkům operačního systému na zarovnání dat je vhodné seřadit v datové struktuře členské proměnné podle velikosti sestupně. Dále je třeba stanovit, na které proměnné se bude program odkazovat častěji než na jiné. Takovéto proměnné je dobré umísťit v paměti blízko sebe, aby bylo možné využít prostorové lokality.

- **Používat předvídatelné operace**

Usnadnit práci hardwarovým jednotkám pro předpovídání větvení může programátor používáním předvídatelných operací. V některých případech může díky snadné předvídatelnosti do značné míry upravit kód i překladač. Ačkoliv naměřené výsledky nemusí reflektovat toto tvrzení, je dobré jej mít na paměti a uvědomit si, že hardware preferuje práci nad homogenními daty, mezi kterými neexistují datové závislosti.

- **Nesdílet cache blok mezi jádry**

Tabulka 12 prezentuje vliv falešného sdílení na výkon programu. Pro odhalení tohoto nedostatku je nezbytná znalost hardwarových záležitostí a nemusí být každému zřejmá. Využitím profilovacích nástrojů může být změřena doba běhu programu a dokáže osvětlit, proč po paralelizaci programu nedošlo k očekávanému zrychlení. V představeném příkladu oprava spočívala v úpravě zarovnání odkazovaných proměnných.

- **Odstranit závislosti**

Stejně jako u předchozího doporučení, pro pochopení projevu datových závislostí by měl vývojář mít znalosti hardware. Napomoci k odhalení závislostí může pomoci prozkoumání zdrojového kódu aplikace v podobě jazyka symbolických adres. Součástí profilovacího programu Tracy je náhled na přeložený program doplněný o další informace, jako o procentuální vyjádření času stráveného v instrukci nebo naznačení závislosti registrů mezi instrukcemi.

- **Zarovnávat data**

V případě práce se SIMD instrukcemi je vyžadováno zarovnání dat. Ačkoliv jinak to vyžadováno není, doporučuje se zarovnat data alespoň podle velikosti cache bloku. Tímto by mělo být zabráněno možnosti rozprostření dat na dvě paměťové stránky.

## ZÁVĚR

Diplomová práce se zabývala seznámením s programovacím paradigmatem zvaným datově orientované programování. V rámci teoretické části bylo pojednáno o historii tohoto pojmu, oblasti využití a byly představeny fundamentální myšlenky tohoto způsobu tvorby programů.

Došlo ke srovnání s ostatními rozšířenými paradigmaty a byly uvedeny výhody a nevýhody. Jelikož jedním z ústředních motivů datově orientovaného programování je efektivní využití hardware, byla podstatná část této práce věnována popisu hardware, jehož správné využití může mít zásadní dopad na výkon aplikace. Mezi tento hardware patří vyrovnávací paměti, operační paměť a také exekuční pipeline procesoru. Pro lepší pochopení výkonnostních charakteristik programu je vhodné zkoumat výstup překladače v podobě jazyka symbolických adres. Pokud je vhodně komunikován řešený problém v programovacím jazyce, překladač dokáže generovat množství optimalizací. Rovněž byla přiblížena instrukční architektura x86, která je v současnosti velmi rozšířená na běžných počítačích. Postupem času se tvůrci hardware soustředí na rozšíření možností paralelizace na různých úrovních. Z toho důvodu byly tyto úrovně představeny a bylo pojednáno o jejich exploataci. Následně byly představeny nástroje, které nám umožní kvantifikovat podstatné metriky udávající výkon programu a také lépe uvažovat nad výkonovými charakteristikami. Předmětem praktické části bylo představit koncepty nastíněné v teoretické části, implementovat je v jazyce C++ a zhodnotit jejich vliv na dobu vykonávání programu pomocí profilovacích nástrojů.

Tato práce představuje ucelený přehled různých druhů optimalizací od položení teoretického základu přes implementaci až k výsledkům měření. Optimalizace byly implementovány pro demonstraci zlepšení některé z významných metrik. Výsledky všech ukázkových programů jsou představeny v podobě tabulek, které dále srovnávají různé překladače a různé optimizační stupně těchto překladačů. Vybrané příklady byly dále analyzovány pomocí profilovacích nástrojů dvou typů. Na základě měření a analýz je výstupem zhodnocení dosažených výsledků a na jejich základě byla sestavena sada doporučení pro softwarové vývojáře, kteří by chtěli využít benefitů, které toto programovací paradigma přináší.

Nahlédnutím na tabulky s výsledky měření lze zjistit, že většina optimalizací přináší očekávaný výsledek a mohou být použity ke snížení doby běhu programu. Tuto skutečnost dále validují profilovací nástroje. Využití těchto optimalizací v reálných aplikacích je opodstatněno a stojí za zvážení.

## SEZNAM POUŽITÉ LITERATURY

- [1] FABIAN, Richard. *Data-oriented design: software engineering for limited resources and short schedules*. B.m.: Richard Fabian, 2018. ISBN 978-1-916478-70-1.
- [2] *Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP) – Games from Within* [online]. 4. prosinec 2009 [vid. 2023-03-10]. Dostupné z: <https://gamesfromwithin.com/data-oriented-design>
- [3] *CppCon 2014: Mike Acton „Data-Oriented Design and C++“* [online]. 2014 [vid. 2023-03-10]. Dostupné z: <https://www.youtube.com/watch?v=rX0ItVEVjHc>
- [4] BAYLISS, Jessica D. The Data-Oriented Design Process for Game Development. *Computer* [online]. 2022, 55(5), 31–38. ISSN 1558-0814. Dostupné z: doi:10.1109/MC.2022.3155108
- [5] *Object-oriented programming - Learn web development | MDN* [online]. 24. únor 2023 [vid. 2023-03-10]. Dostupné z: [https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented\\_programming](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming)
- [6] MITCHELL, Brad. What Is Functional Programming and Why Use It? *Coding Dojo* [online]. 13. červenec 2022 [vid. 2023-03-10]. Dostupné z: <https://www.coding-dojo.com/blog/what-is-functional-programming>
- [7] BRYANT, Randal E. a David R. O'HALLARON. *Computer systems: a programmer's perspective*. Third edition. Boston: Pearson, 2016. ISBN 978-0-13-409266-9.
- [8] HENNESSY, John L., David A. PATTERSON a Andrea C. ARPACI-DUSSEAU. *Computer architecture: a quantitative approach*. 4th ed. Amsterdam ; Boston: Morgan Kaufmann, 2007. ISBN 978-0-12-370490-0.
- [9] *Lecture 25: Prefetching - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu* [online]. 2015 [vid. 2023-03-10]. Dostupné z: <https://www.youtube.com/watch?v=ibPL7T9iEwY>
- [10] *Lecture 26. More Prefetching and Emerging Memory Technologies - CMU - Comp. Arch. 2015 - Onur Mutlu* [online]. 2015 [vid. 2023-03-10]. Dostupné z: <https://www.youtube.com/watch?v=TUFins4z6o4>

- [11] KUSSWURM, Daniel. *Modern x86 assembly language programming: 32-bit, 64-bit, SSE, and AVX*. New York, NY: Apress, 2014. The expert's voice in programming. ISBN 978-1-4842-0065-0.
- [12] STALLMAN, Richard, M. *Using the GNU Compiler Collection: For gcc version 12.2.0* [online]. B.m.: GNU Press. 2022 [vid. 2023-03-10]. Dostupné z: <https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc.pdf>
- [13] *Clang Compiler User's Manual — Clang 17.0.0git documentation* [online]. [vid. 2023-03-10]. Dostupné z: <https://clang.llvm.org/docs/UsersManual.html>
- [14] TYLERMSFT. *MSVC Compiler Options* [online]. 3. srpen 2021 [vid. 2023-03-10]. Dostupné z: <https://learn.microsoft.com/en-us/cpp/build/reference/compiler-options>
- [15] Intel® Intrinsics Guide. *Intel* [online]. [vid. 2023-03-10]. Dostupné z: <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>
- [16] OpenACC Programming and Best Practices Guide. nedatováno.
- [17] *What is Micro-benchmarking?* [online]. [vid. 2023-03-10]. Dostupné z: <https://adservio.fr/post/what-is-microbenchmarking>
- [18] *The Basics of Profiling - Mathieu Ropert - CppCon 2021* [online]. 2022 [vid. 2023-03-10]. Dostupné z: <https://www.youtube.com/watch?v=dToaepIXW4s>
- [19] Intel® VTune™ Profiler. *Intel* [online]. [vid. 2023-03-10]. Dostupné z: <https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html>
- [20] Intel® VTune™ Profiler Performance Analysis Cookbook. *Intel* [online]. [vid. 2023-03-10]. Dostupné z: <https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top.html>
- [21] *OpenACC Programming and Best Practices Guide* [online]. 2022 [vid. 2023-02-06]. Dostupné z: <https://www.openacc.org/sites/default/files/inline-files/openacc-guide.pdf>
- [22] STROUSTRUP, Bjarne. *The C++ Programming Language, 4th Edition*. 4th edition. Westport, Conn: Addison-Wesley Professional, 2013. ISBN 978-0-275-96730-7.
- [23] *Documentation for Visual Studio Code* [online]. [vid. 2023-03-10]. Dostupné z: <https://code.visualstudio.com/docs>

- [24] *Code Time Data: Ranking the Top 5 Code Editors in 2019 - /src/ blog* [online]. [vid. 2023-03-10]. Dostupné z: <https://www.software.com/src/ranking-the-top-5-code-editors-2019>
- [25] ANANDMEG. *Overview of Visual Studio* [online]. 10. březen 2023 [vid. 2023-03-10]. Dostupné z: <https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide>
- [26] *Overview | CMake* [online]. [vid. 2023-03-10]. Dostupné z: <https://cmake.org/overview/>
- [27] *Examples | CMake* [online]. [vid. 2023-03-10]. Dostupné z: <https://cmake.org/examples/>
- [28] *CPU Cache Effects - Sergey Slotin - Meeting C++ 2022* [online]. 2022 [vid. 2023-03-10]. Dostupné z: [https://www.youtube.com/watch?v=mQWuX\\_KgH00](https://www.youtube.com/watch?v=mQWuX_KgH00)
- [29] Performance Implications of False Sharing. *CoffeeBeforeArch.github.io* [online]. 28. prosinec 2019 [vid. 2023-03-17]. Dostupné z: <https://coffeebeforearch.github.io/2019/12/28/false-sharing-tutorial.html>
- [30] NESTERUK, Dmitri. *Design patterns in modern C++: reusable approaches for object-oriented software design*. New York, NY: Springer Science+Business Media, 2018. ISBN 978-1-4842-3602-4.

## SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AVX	Advanced Vector Extensions
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DOD	Data-oriented design
DOP	Datově orientované programování
DRAM	Dynamic Random Access Memory
FP	Funkcionální programování
GPU	Graphics Processing Unit
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
LFU	Least Frequently Used
LRU	Least Recently Used
MMX	Multi Media Extenstions
OOP	Objektově orientované programování
PC	Personal Computer
PMU	Performance Monitoring Unit
SIMD	Single Instruction Multiple Data
SRAM	Static Random Access Memory
SSD	Solid State Drive
SSE	Streaming SIMD Extensions

## SEZNAM OBRÁZKŮ

Obr. 1: Vývoj doby trvání přístupu do procesorové a operační paměti. [7] .....	20
Obr. 2: Příklad rozdělení adresy. [Zdroj vlastní] .....	20
Obr. 3: Příklad uspořádání a typu cache paměti – snímek programu CPU-Z. [Zdroj vlastní] .....	22
Obr. 4: Sekvenční vykonávání instrukcí. [Zdroj vlastní].....	24
Obr. 5: Skalární instrukční pipeline. [Zdroj vlastní].....	24
Obr. 6: Superskalární instrukční pipeline. [Zdroj vlastní] .....	25
Obr. 7: Data hazard. [Zdroj vlastní].....	26
Obr. 8: Vnitřní architektura x86-32. [10] .....	30
Obr. 9: Příklad programu v jazyce symbolických adres. [10] .....	32
Obr. 10: Příklad běžné komplikace programu pomocí GCC. [Zdroj vlastní].....	32
Obr. 11: Příklad běžné komplikace programu pomocí Clang. [Zdroj vlastní] .....	33
Obr. 12: Příklad běžné komplikace programu pomocí MSVC. [Zdroj vlastní] .....	34
Obr. 13: Možnosti využití registru v SIMD. [10] .....	34
Obr. 14: Sada registrů AVX. [10].....	35
Obr. 15: Program sečtení dvou polí a uložení do výsledného pole. [Zdroj vlastní] ...	36
Obr. 16: Kód varianty s přepínači <i>-O1 -msse</i> . [Zdroj vlastní] .....	36
Obr. 17: Kód varianty s přepínači <i>-O2 -msse</i> . [Zdroj vlastní] .....	37
Obr. 18: Kód varianty s přepínači <i>-O2 -msse</i> a lichým počtem prvků v poli. [Zdroj vlastní] .....	37
Obr. 19: Exekuční engine Haswell CPU. [10] .....	39
Obr. 20: Ukázka <i>loop-carried</i> závislosti. [7] .....	42
Obr. 21: Přepis smyčky. [7] .....	42
Obr. 22: Příklad atomické výměny hodnot. [7] .....	44
Obr. 23: Spin lock. [7] .....	45
Obr. 24: Graf průběhu. [6] .....	47
Obr. 25: Graf průběhu s promítnutými hodnotami binárního semaforu. [6] .....	48
Obr. 26: Grafické prostředí programu Intel VTune. [18] .....	50
Obr. 27: Okno s μpipe zobrazením. [18] .....	51
Obr. 28: Pohled <i>source code analysis</i> . [18] .....	52
Obr. 29: Použití programu Tracy z příkazové řádky. [Zdroj vlastní] .....	53
Obr. 30: Grafické prostředí programu Tracy. [Zdroj vlastní].....	54

Obr. 31: Informace o zóně. [Zdroj vlastní] .....	55
Obr. 32: Pohled na funkci ze zdrojového souboru. [Zdroj vlastní] .....	56
Obr. 33: Pohled na program v podobě jazyka symbolických adres. [Zdroj vlastní] ..	57
Obr. 34: Grafické prostředí programu Visual Studio Code. [Zdroj vlastní] .....	59
Obr. 35: Vývojové prostředí Visual Studio 2019. [24].....	60
Obr. 36: Konfigurační soubor kořenového adresáře. [26] .....	61
Obr. 37: Konfigurační soubor knihovny Hello. [26] .....	61
Obr. 38: Konfigurační soubor adresáře se spustitelným souborem. [26] .....	62
Obr. 39: Informace o CPU. [Zdroj vlastní].....	62
Obr. 40: Informace o operační paměti. [Zdroj vlastní] .....	63
Obr. 41: Informace o grafické kartě #1. [Zdroj vlastní] .....	63
Obr. 42: Informace o grafické kartě #2. [Zdroj vlastní] .....	64
Obr. 43: Program Tracy – analýza zdrojového kódu 1. [Zdroj vlastní].....	98
Obr. 44: Profilování částí podle velikosti dat. [Zdroj vlastní] .....	99
Obr. 45: Pohled na srovnání doby běhu pro velikost 900x900. [Zdroj vlastní] .....	99
Obr. 46: Průměrná doba běhu iterace po řádcích. [Zdroj vlastní] .....	99
Obr. 47: Tělo hlavní smyčky obou variant v jazyce symbolických adres. [Zdroj vlastní]	
.....	100
Obr. 48: Srovnání hardwarových událostí obou variant. [Zdroj vlastní].....	100
Obr. 49: Shrnutí výkonnostních charakteristik programu vTune pro variantu po sloupcích. [Zdroj vlastní] .....	101
Obr. 50: Shrnutí výkonnostních charakteristik programu vTune pro variantu po řádcích. [Zdroj vlastní] .....	102
Obr. 51: Tělo hlavní smyčky různých stupňů unrollingu v jazyce symbolických adres.	
[Zdroj vlastní].....	103
Obr. 52: Shrnutí výkonnostních charakteristik programu vTune pro variantu bez unrollingu. [Zdroj vlastní] .....	104
Obr. 53: Shrnutí výkonnostních charakteristik programu vTune pro variantu s dvojnásobným unrollingem. [Zdroj vlastní] .....	105
Obr. 54: Shrnutí výkonnostních charakteristik programu vTune pro variantu s čtyřnásobným unrollingem. [Zdroj vlastní] .....	106
Obr. 55: Pohled na srovnání doby běhu pro velikost 800000. [Zdroj vlastní] .....	106

Obr. 56: Shrnutí výkonnostních charakteristik programu vTune pro klasickou variantu. [Zdroj vlastní].....	107
Obr. 57: Shrnutí výkonnostních charakteristik programu vTune pro variantu se separovanými daty. [Zdroj vlastní] .....	108
Obr. 58: Srovnání hardwarových událostí obou variant. [Zdroj vlastní] .....	108
Obr. 59 Shrnutí výkonnostních charakteristik programu vTune pro skalární variantu. [Zdroj vlastní].....	109
Obr. 60: Shrnutí výkonnostních charakteristik programu vTune pro vektorovou variantu. [Zdroj vlastní].....	110
Obr. 61: Tělo hlavní smyčky skalární a vektorové varianty v jazyce symbolických adres. [Zdroj vlastní] .....	111
Obr. 62: Pohled na srovnání doby běhu pro velikost 800000. [Zdroj vlastní] .....	111

**SEZNAM TABULEK**

Tabulka 1: Výsledky implementace zdrojového kódu 1 .....	83
Tabulka 2: Výsledky implementace zdrojového kódu 2 .....	84
Tabulka 3: Výsledky implementace zdrojového kódu 3 .....	86
Tabulka 4: Výsledky implementace zdrojového kódu 4 .....	87
Tabulka 5: Výsledky implementace zdrojového kódu 5 .....	88
Tabulka 6: Výsledky implementace zdrojového kódu 6 .....	89
Tabulka 7: Výsledky implementace zdrojového kódu 7 .....	90
Tabulka 8: Výsledky implementace zdrojového kódu 8 .....	91
Tabulka 9: Výsledky implementace zdrojového kódu 9 .....	93
Tabulka 10: Výsledky implementace zdrojového kódu 10 .....	94
Tabulka 11: Výsledky implementace zdrojového kódu 11 .....	95
Tabulka 12: Výsledky implementace zdrojového kódu 12 .....	96
Tabulka 13: Výsledky implementace zdrojového kódu 13 .....	97

## SEZNAM ZDROJOVÝCH KÓDŮ

Zdrojový kód 1: Průchod dvourozměrným polem. [Zdroj vlastní] .....	67
Zdrojový kód 2: Různé způsoby průchodu maticí při sčítání matic. [6] .....	69
Zdrojový kód 3: Smyčka se závislostí a bez závislosti. [7] .....	69
Zdrojový kód 4: Loop unrolling. [6].....	70
Zdrojový kód 5: Průchod smyčkou s různým krokem. [Zdroj vlastní] .....	71
Zdrojový kód 6: Přístupy do pole s různým krokem. [Zdroj vlastní] .....	72
Zdrojový kód 7: Separace často odkazovaných a málo odkazovaných dat. [Zdroj vlastní] .....	73
Zdrojový kód 8: AoS a SoA přístup. [Zdroj vlastní] .....	74
Zdrojový kód 9: Suma hodnot v poli: skalární a SIMD způsob. [Zdroj vlastní] .....	76
Zdrojový kód 10: Vliv pořadí proměnných v datové struktuře na její velikost. [Zdroj vlastní] .....	76
Zdrojový kód 11: Předvídatelnost podmínek. [Zdroj vlastní] .....	77
Zdrojový kód 12: Pravé a falešné sdílení. [Zdroj vlastní] .....	79
Zdrojový kód 13: Demonstrace aliasingu paměti. [6] .....	79
Zdrojový kód 14: Ukázka existence based processing. [Zdroj vlastní] .....	81

## **SEZNAM PŘÍLOH**

Příloha P I: CD s diplomovou prací včetně zdrojového kódu a výsledků měření

## **PŘÍLOHA P I: CD**

Přiložené CD obsahuje:

Diplomovou práci ve formátu .pdf: fulltext.pdf

Komprimovaný zdrojový kód a výsledky  
měření: prilohy.zip