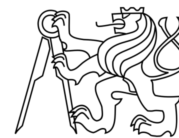
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název:	Efektivní násobení ídkých matic
Student:	Thanh Quang Mai
Vedoucí:	doc. Ing. Ivan Šime ek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce zimního semestru 2018/19

Pokyny pro vypracování

1) Seznamte se s r znými formáty uložení ídkých matic v [1,2].
2) Seznamte se s algoritmy pro násobení matic a diskutujte jejich využití pro násobení ídkých matic v [3,4].
3) Implementujte v C/C++ algoritmy z bodu 2) pro násobení matic uložených ve formátu z bodu 1) , nejmén tyto: triviální a Strassen v algoritmus.
4) Algoritmy optimalizujte a paralelizujte pomocí technologie OpenMP.
5) Algoritmy otestujte na vlastních vygenerovaných ídkých maticích a na ídkých maticích z ve ejn dostupných zdroj (nap . MatrixMarket, ...).
6) Výkonnost algoritm zm te na fakultním serveru STAR.
7) Porovnejte výkonnost s existujícími knihovnami ešící efektivní násobení ídkých matic (nap . knihovnou Eigen [5]).

Seznam odborné literatury

[1] DONGARRA, Jack. Survey of Sparse Matrix Storage Formats [online]. 1995 [cit. 2017-02-22]. Dostupné z: http://www.netlib.org/linalg/html_templates/node90.html
[2] LICHÝ, Lukáš. Efektivní násobení ídkých matic [online]. ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE, 2015. Dostupné také z: https://dip.felk.cvut.cz/browse/pdfcache/lichyluk_2014bach.pdf. Bakalá ská práce.
[3] ZVONÍK. Rychlé násobení matic a Strassen v algoritmus ve Winogradov úprav [online]. 2010. Dostupné z: http://people.fjfi.cvut.cz/pelanedi/tema/skriptatemakonecsemestru/strassenzvonik.pdf
[4] YUSTER, Raphael a Uri ZWICK. Fast Sparse Matrix Multiplication. ACM Transactions on Algorithms (TALG) [online]. 2005. Dostupné z: http://dl.acm.org/citation.cfm?id=1077466
[5] Eigen: Dostupné z: https://eigen.tuxfamily.org/dox-devel/group__TutorialSparse.html

doc. Ing. Jan Janoušek, Ph.D. vedoucí katedry	prof. Ing. Pavel Tvrdík, CSc. d kan

V Praze dne 28. února 2017

Bakalářská práce

Efektivní násobení řídkých matic

Thanh Quang Mai

Katedra teoretické informatiky
Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

12. února 2019

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 12. února 2019 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Thanh Quang Mai. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Mai, Thanh Quang. *Efektivní násobení řídkých matic.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato bakalářská práce se zabývá formáty pro uložení řídkých matic COO, CSR, CSC a algoritmy pro násobení matic v těchto formátech. Cílem práce je i implementace těchto algoritmů a formátů v programovacím jazyce C++ a změření výkonu algoritmů na fakultním klasteru STAR.

Klíčová slova řídké matice, Strassenův algoritmus, násobení matic, COO, CSR, C++

Abstract

This bachelor's thesis deals with the sparse matrix storage formats COO, CSR, CSC and algorithms for matrix multiplication in these formats. The goal of this thesis is the implementation of said formats and algorithms in the programming language C++ and testing their performance on the faculty cluster STAR.

Keywords sparse matrices, Strassen's algorithm, matrix multiplication, COO, CSR, C++

Obsah

Úvod **1**

1 Definice základních pojmů **3**
1.1 Matice . . . 3
1.1.1 Vektor . . . 3
1.2 Typy matic . . . 4
1.2.1 Obdélníková matice . . . 4
1.2.2 Čtvercová matice . . . 4
1.2.3 Jednotková matice . . . 4
1.2.4 Regulární a inverzní matice . . . 4
1.2.5 Řídká matice . . . 5
1.3 Maticové operace . . . 5
1.3.1 Násobení matice skalárem . . . 5
1.3.2 Součet a rozdíl matic . . . 6
1.3.3 Násobení matice maticí . . . 6
1.3.4 Transpozice matice . . . 7
1.4 Asymptotické složitosti . . . 7
1.4.1 $\mathcal{O}$-notace . . . 8
1.4.2 Θ-notace . . . 8
1.4.3 Ω-notace . . . 8
1.4.4 Mistrovská metoda . . . 9

2 Formáty ukládání řídkých matic **11**
2.1 Hustý formát . . . 11
2.2 COO - Coordinate List . . . 11
2.3 CSR - Compressed Sparse Row . . . 12
2.4 CSC - Compressed Sparse Column . . . 13

3 Algoritmy násobení matic **15**
3.1 Klasický algoritmus násobení matic . . . 15

3.2 Strassenův algoritmus 16
3.3 Další algoritmy pro rychlé násobení matic 18
3.4 Násobení matic v řídkém formátu 18
3.4.1 Násobení ve formátu COO 19
3.4.2 Násobení ve formátu CSR 20
3.4.3 Násobení ve formátu CSC 20
3.4.4 Násobení ve formátu CSC × CSR 20

4 Analýza a návrh 23
4.1 Existující řešení 23
4.1.1 Akademické práce 23
4.1.2 Knihovny 23
4.2 Strassenův algoritmus s řídkými maticemi 26

5 Implementace 27
5.1 Použité nástroje 27
5.1.1 Nastavení překladače 27
5.1.2 Optimalizační techniky 28
5.1.2.1 Vektorizace 28
5.1.2.2 Rozbalení cyklů 29
5.1.2.3 Proložení cyklů 29
5.1.3 Knihovna OpenMP 30
5.1.3.1 Direktiva pro překladač 31
5.1.3.2 Knihovní funkce 32
5.1.3.3 Proměnné prostředí 32
5.1.4 Formát uložení MatrixMarket 34
5.2 Popis tříd 35
5.2.1 Třída Matrix 35
5.2.2 Třída MM_IO 36
5.2.3 Třída Convert 37
5.2.4 Třída Mult 39
5.3 Paralelizace 40

6 Měření 43
6.1 Hranice přepnutí Strassenova algoritmu 44
6.2 Doba výpočtu v závislosti na hustotě matice 44
6.3 Doba výpočtu v závislosti na velikosti matice 46
6.4 Zrychlení algoritmů při použití více vláken 48
6.5 Porovnání s knihovnou Eigen 50

Závěr 53

Literatura 55

A Seznam použitých zkratek 59

B Obsah přiloženého CD **61**

Seznam obrázků

5.1 Fork-join model 30
5.2 Matrix 36
5.3 MM_IO 36
5.4 Convert 37
5.5 Mult 39

6.1 Distribuce nenulových prvků v matici PSMIGR 2 44
6.2 Distribuce nenulových prvků v matici WEST2021 44
6.3 Čas výpočtu pro různé hranice přepnutí Strassenova algoritmu, matice WEST20216.2 45
6.4 Doba výpočtu při změně hustoty matice o rozměrech 1000×1000, osa y v logaritmickém měřitku 46
6.5 Doba výpočtu při změně velikosti matice (hustota matice 10%), osa y v logaritmickém měřitku 47
6.6 Doba výpočtu při změně velikosti matice (hustota matice 20%), osa y v logaritmickém měřitku 48
6.7 Zrychlení při využití více vláken, matice PSMIGR 26.1 49
6.8 Zrychlení při využití více vláken, matice 1000×1000 se 100% hustotou 50
6.9 Porovnání vybraných algoritmů s knihovnou Eigen pro různé hustoty matic o velikosti 2000×2000 51

Seznam tabulek

1.1 Obvyklé třídy složitosti 9

2.1 Matice uložená ve formátu COO 12
2.2 Matice uložená ve formátu CSR 13
2.3 Matice uložená ve formátu CSC 14

4.1 Uložení řídké matice v upraveném formátu CSC ve knihovně Eigen[1] 24

Úvod

V dnešní době je v mnoha oblastech potřeba rychle a efektivně provádět výpočty s maticemi různých typů a velikostí. Matice se využívají např. v oblastech matematiky, fyziky a informačních technologií. V IT se s maticemi často setkáváme v grafice (práce s vektory), teorii grafů (Floyd-Warshallův algoritmus), umělé inteligenci, šifrování a mnohých dalších.

Abychom se při práci s maticemi dostali k nějakému konečnému výsledku, tak je většinou zapotřebí s maticemi provádět nějaké maticové operace. Ve této práci se zabýváme především operací násobení dvou matic, konkrétněji násobením dvou **řídkých matic**.

Pojem řídká matice není přesně definován. Neformálně by se dalo říct, že řídké matice jsou takové matice, ve kterých je více nulových než nenulových prvků. V praxi jde většinou o matice s rozměry v řádech tisíců, desetitisísů a vyšších. Práce s s takto velkými maticemi je náročná jak paměťově (nutnost ukládat velké množství nulových prvků), tak i z hlediska výpočetního času (délka trvání číselné operace v počítačích nezávisí na konkrétních číslech, se kterými se počítá, trvá vždy stejně dlouho). Z tohoto důvodu došlo ke vzniku formátů (pro ukládání) a algoritmů násobení pro řídké matice, které jsou pro tento typ matic mnohem efektivnější.

Cílem mé práce je analýza formátů pro ukládání řídkých matic, implementace vybraných algoritmů pro násobení matic v programovacím jazyce C++ (klasický a Strassenův algoritmus) a změření výkonnosti těchto implementovaných algoritmů.

Kapitola **1**

Definice základních pojmů

1.1 Matice

Definice 1.1. Nechť $m, n \in \mathbb{N}$. Uspořádaný soubor mn čísel zapsaný do tabulky o m řádcích a n sloupcích nazýváme **matice**[2] typu $m \times n$. Matici obvykle značíme takto:

$$\mathbb{A} := \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix},$$

kde $a_{i,j}$ jsou **prvky** matice (někdy je značíme taky jako $\mathbb{A}_{i,j}$ a nazýváme je ij**té prvky**). Číslu i říkáme **řádkový** a číslu j **sloupcový** index.

1.1.1 Vektor

Definice 1.2. Nechť $m, n \in \mathbb{N}$. Matice $\mathbb{A} \in \mathbb{R}^{m,1}$, resp. $\mathbb{B} \in \mathbb{R}^{1,n}$ nazveme mprvkové sloupcové, resp. nprvkové řádkové **vektory**.

$$\mathbb{A} := \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{m,1} \end{pmatrix}, \mathbb{B} := \begin{pmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,n} \end{pmatrix}$$

1.2 Typy matic

V této sekci si představíme některé základní typy matic.

1.2.1 Obdélníková matice

Definice 1.3. Nechť $m, n \in \mathbb{N}$. Matici $\mathbb{A} \in \mathbb{R}^{m,n}$ nazveme **obdélníkovou**, právě tehdy když $m \neq n$.

Př.

$$\mathbb{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

1.2.2 Čtvercová matice

Definice 1.4. Nechť $m, n \in \mathbb{N}$. Matici $\mathbb{A} \in \mathbb{R}^{m,n}$ nazveme **čtvercovou**, právě tehdy když $m = n$.

Př.

$$\mathbb{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

1.2.3 Jednotková matice

Definice 1.5. Čtvercovou matici $\mathbb{E} \in \mathbb{R}^{n,n}$ nazýváme **jednotkovou maticí**[3], pokud pro její prvky $e_{i,j}$ platí: $e_{i,j} = 0$ pro $i \neq j$ a $e_{i,j} = 1$ pro $i = j$.

Př.

$$\mathbb{E} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

Jednotková matice je speciální případ tzv. **diagonální matice**.

1.2.4 Regulární a inverzní matice

Definice 1.6. Nechť $\mathbb{A} \in \mathbb{R}^{n,n}$. Existuje-li $\mathbb{B} \in \mathbb{R}^{n,n}$ taková, že platí

$$\mathbb{A}\mathbb{B} = \mathbb{B}\mathbb{A} = \mathbb{E}^{12},$$

nazýváme matici $\mathbb{A}$ **regulární**[2] a $\mathbb{B}$ **inverzní maticí**[2] k matici $\mathbb{A}$. Značíme $\mathbb{B} = \mathbb{A}^{-1}$. Pokud $\mathbb{A}$ není regulární, nazýváme matici $\mathbb{A}$ **singulární**[2].

[1] jednotková matice
[2] součin matic definován v 1.3.3

Př.

$$\mathbb{A} = \begin{pmatrix} 4 & 3 \\ 3 & 2 \end{pmatrix}, \mathbb{A}^{-1} = \begin{pmatrix} -2 & 3 \\ 3 & -4 \end{pmatrix}$$

$$\mathbb{A}\mathbb{A}^{-1} = \begin{pmatrix} 4 & 3 \\ 3 & 2 \end{pmatrix} \begin{pmatrix} -2 & 3 \\ 3 & -4 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

1.2.5 Řídká matice

Pojem řídká matice není přesně definován. Neformálně by se dalo říct, že řídké matice jsou takové matice, ve kterých je více nulových než nenulových prvků. V praxi jde většinou o matice s rozměry v řádech tisíců, desetitisísů a vyšších. Práce s s takto velkými maticemi je náročná jak paměťově (nutnost ukládat velké množství nulových prvků), tak i z hlediska výpočetního času (délka trvání číselné operace v počítačích nezávisí na konkrétních číslech, se kterými se počítá, trvá vždy stejně dlouho). Z tohoto důvodu došlo ke vzniku formátů (pro ukládání) a algoritmů násobení pro řídké matice, které jsou pro tento typ matic mnohem efektivnější.

„Řídká matice“ není matematický pojem, proto neexistuje jednoznačná definice, tento pojem je spojen se způsobem práce s touto maticí v počítači. Pro násobení řídkých matic existují efektivnější algoritmy a úspornější způsoby ukládání těchto matic v paměti než pro klasické husté matice. Řídkou maticí bychom tedy mohli nazvat takovou matici, u kterých se díky počtu jejích nulových prvků vyplatí tyto efektivnější algoritmy a úspornější formáty pro uložení využít.

Řídkost matice se vyjadřuje pomocí ***nnz*** (**n**umber of **n**on**z**ero matrix elements), což je počet nenulových prvků z celkového počtu $m \times n$ prvků matice z $\mathbb{R}^{m,n}$.

Př. $nnz = 5$

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 \end{pmatrix}$$

1.3 Maticové operace

1.3.1 Násobení matice skalárem

Definice 1.7. Nechť $m, n \in \mathbb{N}$,$\alpha \in \mathbb{R}$ a $\mathbb{A} \in \mathbb{R}^{m,n}$ matice s prvky a_{ij}. Součin matice $\mathbb{A}$ a reálného čísla (skaláru)[2] α definujeme takto:

$$\alpha\mathbb{A} := \begin{pmatrix} \alpha a_{1,1} & \alpha a_{1,2} & \cdots & \alpha a_{1,n} \\ \alpha a_{2,1} & \alpha a_{2,2} & \cdots & \alpha a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha a_{m,1} & \alpha a_{m,2} & \cdots & \alpha a_{m,n} \end{pmatrix}.$$

1.3.2 Součet a rozdíl matic

Definice 1.8. Buďte $m, n \in \mathbb{N}$, $\mathbb{A}, \mathbb{B} \in \mathbb{R}^{m,n}$ matice s prvky a_{ij} resp. b_{ij}. **Součet matic**[2] $\mathbb{A}$ a $\mathbb{B}$ definujeme jako:

$$\mathbb{A} + \mathbb{B} := \begin{pmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & \cdots & a_{1,n} + b_{1,n} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & \cdots & a_{2,n} + b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} + b_{m,1} & a_{m,2} + b_{m,2} & \cdots & a_{m,n} + b_{m,n} \end{pmatrix}.$$

Analogicky lze definovat i rozdíl matic.

Jelikož je sčítání reálných čísel komutativní, resp. asociativní, je i sčítání matic komutativní, resp. asociativní. Pro matice $\mathbb{A}, \mathbb{B}, \mathbb{C} \in \mathbb{R}^{m,n}$ tedy platí:

$$\mathbb{A} + \mathbb{B} = \mathbb{B} + \mathbb{A} \qquad \text{(Komutativita)}$$

$$(\mathbb{A} + \mathbb{B}) + \mathbb{C} = \mathbb{A} + (\mathbb{B} + \mathbb{C}) \qquad \text{(Asociativita)}$$

1.3.3 Násobení matice maticí

Definice 1.9. Buďte $m, n, p \in \mathbb{N}$, $\mathbb{A} \in \mathbb{R}^{m,n}$ matice s prvky $a_{i,j}$ a $\mathbb{B} \in \mathbb{R}^{n,p}$ matice s prvky $b_{i,j}$. **Součinem matic**[2] $\mathbb{A}$ a $\mathbb{B}$ je matice $\mathbb{D} \in \mathbb{R}^{m,p}$ s prvky $d_{i,j}$, pro kterou platí

$$d_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j},$$

značíme $\mathbb{D} = \mathbb{A}\mathbb{B}$.

Všimněme si, že součin matic $\mathbb{D} = \mathbb{A}\mathbb{B}$ je definován pouze tehdy, pokud počet sloupců matice $\mathbb{A}$ je roven počtu řádků matice $\mathbb{B}$, výsledná matice $\mathbb{D}$ bude mít stejný počet řádků, jako matice $\mathbb{A}$ a stejný počet sloupců, jako matice $\mathbb{B}$.

Př.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1\cdot 1 + 2\cdot 4 + 3\cdot 7 & 1\cdot 2 + 2\cdot 5 + 3\cdot 8 & 1\cdot 3 + 2\cdot 6 + 3\cdot 9 \\ 4\cdot 1 + 5\cdot 4 + 6\cdot 7 & 4\cdot 2 + 5\cdot 5 + 6\cdot 8 & 4\cdot 3 + 5\cdot 6 + 6\cdot 9 \end{pmatrix} =$$

$$= \begin{pmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \end{pmatrix}$$

Součin matic **není** obecně komutativní. Může se stát, že součin $\mathbb{A}\mathbb{B}$ je definován ale součin $\mathbb{B}\mathbb{A}$ definován není. V případě, kdy jsou oba součiny definovány, mohou jako výsledek vzniknout dvě matice různého typu. I v případě, kdy by vyšly matice stejného typu, tak si nemusí být rovny.

Výsledkem součinu dvou čtvercových matic z $\mathbb{R}^{n,n}$ je opět matice z $\mathbb{R}^{n,n}$. Je tedy alespoň součin dvou čtvercových matic komutativní? Na následujícím příkladu je vidět, že není.

Př.

$$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 2 \\ -2 & -2 \end{pmatrix}$$

Jak je to tedy s asociativitou? Za předpokladu, že rozměry matic jsou takové, že jsou součiny definované, tak asociativita platí.

Nechť $m, n, s, t \in \mathbb{N}$. Pro libovolné matice $\mathbb{A} \in \mathbb{R}^{m,n}$, $\mathbb{B} \in \mathbb{R}^{n,s}$ a $\mathbb{D} \in \mathbb{R}^{s,t}$ platí:

$$\mathbb{A}(\mathbb{B}\mathbb{D}) = (\mathbb{A}\mathbb{B})\mathbb{D}.$$

Důkaz tohoto tvrzení nalezneme například v [2], [3].

1.3.4 Transpozice matice

Definice 1.10. Nechť $m, n \in \mathbb{N}$ a $\mathbb{A} \in \mathbb{R}^{m,n}$ matice s prvky $a_{i,j}$. **Transpozicí**[2] matice $\mathbb{A}$ je matice $\mathbb{A}^T \in \mathbb{R}^{n,m}$, jejíž prvek v jtém řádku a itém sloupci je roven $a_{i,j}$.

Transpozice tedy není aritmetická operace, je to vlastně přeskládání prvků. Neformálně by se dalo říci, že transponovaná matice $\mathbb{A}^T$ vznikne přepsáním řádků z matice $\mathbb{A}$ do sloupců matice $\mathbb{A}^T$.

Př.

$$\text{Je-li např. } \mathbb{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}, \text{ potom } \mathbb{A}^T = \begin{pmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{pmatrix}.$$

Pro každou matici $\mathbb{A}$ také platí $(\mathbb{A}^T)^T = \mathbb{A}$.

1.4 Asymptotické složitosti

Při porovnávání efektivity algoritmů používáme tzv. asymptotickou složitost. Ta vyjadřuje, jak roste náročnost algoritmus (doba výpočtu nebo potřebná paměť) s rostoucím množstvím vstupních dat. Doba výpočtu se však neměří v sekundách ale počtem provedených elementárních operací. Mezi tyto operace patří například základní aritmetické operace, přiřazení či porovnávání dvou hodnot.

Asymptotická složitost nám rozděluje algoritmy do tříd složitostí, která nám umožňuje porovnávat efektivitu algoritmů mezi sebou, a to nezávisle na imeplementaci či použité platformě.[4] K určování asymptotické složitosti používáme tři základní notace - $\mathcal{O}$-notaci, Θ-notaci a Ω-notaci.

1.4.1 $\mathcal{O}$-notace

$\mathcal{O}$ označuje množinu funkcí, které asymptoticky rostou řádově rychleji nebo stejně rychle. Notaci $\mathcal{O}$ lze tedy chápat jako odhad složitosti v nejhoršího případě.

Definice 1.11. Jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je nejvýše řádu $g(n)$, psáno $f(n) = \mathcal{O}(g(n))$, jestliže:

$$\exists c \in \mathbb{R}^+ \, \exists n_0 \in \mathbb{N} \, \forall n \geq n_0 : f(n) \leq c \cdot g(n)[5]$$

1.4.2 Θ-notace

Θ označuje množinu funkcí, které asymptoticky rostou stejně rychle. U některých algoritmů nelze určit. Používá se k odhadu složitosti v průměrném případě.

Definice 1.12. Jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je stejného řádu jako $g(n)$, psáno $f(n) = \Theta(g(n))$, jestliže:

$$\exists c_1, c_2 \in \mathbb{R}^+ \, \exists n_0 \in \mathbb{N} \, \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)[5]$$

1.4.3 Ω-notace

Ω označuje množinu funkcí, které asymptoticky rostou stejně rychle nebo pomaleji. Notaci $\mathcal{O}$ lze tedy chápat jako odhad složitosti v nejlepším případě.

Definice 1.13. Jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je nejméně řádu $g(n)$, psáno $f(n) = \Omega(g(n))$, jestliže:

$$\exists c \in \mathbb{R}^+ \, \exists n_0 \in \mathbb{N} \, \forall n \geq n_0 : c \cdot g(n) \leq f(n)[5]$$

Při určování složitosti algoritmů se zanedbávají aditivní a multiplikativní konstanty, také se využívá toho, že pro velká vstupní data bude dominovat asymptoticky nejsložitější člen.

Př. Máme-li například funkci $f(n) = 3n^3 + 2n^2 + n + 1$, pak časovou složitost určuje hlavní člen n^3 a složitost v nejhorším případě bude tedy $\mathcal{O}(n^3)$.

Tabulka 1.1: Obvyklé třídy složitosti

Asymptotická složitost	Vyjádření
konstantní	$\mathcal{O}(1)$
logaritmická	$\mathcal{O}(\log n)$
lineární	$\mathcal{O}(n)$
lineárně logaritmická	$\mathcal{O}(n \log n)$
kvadratická	$\mathcal{O}(n^2)$
polynomiální	$\mathcal{O}(n^k), k \in \mathbb{R}$
exponenciální	$\mathcal{O}(k^n), k \in \mathbb{R}$
faktoriálová	$\mathcal{O}(n!)$

1.4.4 Mistrovská metoda

Složitosti mnoha rekurentních algoritmů lze vyjádřit ve formě rekurentní rovnice ve tvaru

$$t(n) = at(n/b) + f(n),$$

kde $a \geq 1, b > 0$, a $f(n)$ je funkce jedné proměnné. Rovnice odpovídá rekurzivnímu algoritmu, který dělí problém velikosti n na a částí velikosti n/b a na dělení a zpětné kombinování výsledků potřebuje $f(n)$ času. Nejjednodušší způsob jak záskat přesný odhad chování $t(n)$ je použít takzvanou **Mistrovskou metodu** (angl. Master theorem)[5].

Nechť $a \geq 1$ a $b > 1$ jsou konstanty a $f(n)$ je funkce jedné proměnné. Potom rekurentní rovnice

$$t(n) = at(n/b) + f(n)$$

má následující asymptotické chování:

1. Pokud $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ pro nějakou konstantu $\epsilon > 0$, pak $t(n) = \Theta(n^{\log_b a})$.
2. Pokud $f(n) = \Theta(n^{\log_b a})$, pak $t(n) = \Theta(n^{\log_b a} \log n)$.
3. Pokud $f(n) = \Omega(n^{\log_b a + \epsilon})$ pro nějakou konstantu $\epsilon > 0$ a pokud $af(n/b) \leq cf(n)$ pro nějakou konstantu $c < 1$ a $\forall n \geq n_0$, pak $t(n) = \Theta(f(n))$.

Př. Pro algoritmus *Mergesort* vypadá rovnice takto:

$$t(n) = 2t(n/2) + n$$

$$a = 2, b = 2, f(n) = n, n^{\log_2 2} = n = \Theta(n) \rightarrow t(n) = \Theta(n \log n). \quad \text{(2. případ)}$$

Kapitola 2

Formáty ukládání řídkých matic

V této kapitole bylo čerpáno z těchto zdrojů [6][7][8].

2.1 Hustý formát

Klasickým způsobem, jak uložit obecnou matici o rozměrech $m \times n$, je do 2D pole o mn prvcích. Do tohoto pole se ukládá každý prvek matice bez ohledu na to, zda je prvek nulový, či nenulový. Tento formát je vhodný pro ukládání hustých matic (např. 90%+ nenulových prvků) a také nám poskytuje rychlý přístup k jakémukoliv prvku matice. Avšak při práci s řídkými maticemi o velkých rozměrech, u kterých je počet nulových prvků mnohem vyšší než prvků nenulových, se tento způsob ukládání stává vysoce neefektivní.

Řídké matice lze v paměti uložit úsporněji. Vzhledem k jejich struktuře je žádoucí ukládat do paměti pouze nenulové prvky, avšak využitím tohoto přístupu nám vzniká nutnost ukládat do paměti také indexy těchto prvků. A právě formáty, jak lze uložit řídké matice, se zabývat v této kapitole.

2.2 COO - Coordinate List

Coordinate list, česky seznam souřadnic, je základní formát pro ukládání řídkých matic. Formát COO je velmi jednoduchý a intuitivní. Tento formát ukládá nenulové prvky matice jako uspořádané trojice (i, j, a_{ij}), tedy řádkový index, sloupcový index a hodnotu prvku. Tento formát by při ukládání husté matice spotřeboval více paměti než předchozí formát, protože k hodnotám prvků se navíc do paměti ukládají indexy prvků. Avšak pro řídké matice, kde je počet nenulových prvků v řádu jednotek procent, tento problém mizí a formát se stává efektivní.

K uložení matice do paměti potřebujeme tři pole *row, col* a *val* o velikosti *nnz*[3]. V polích *row*, resp. *col* jsou uloženy řádkové, resp. sloupcové indexy

[3]počet nenulových prvků

prvků. V poli *val* jsou uloženy hodnoty nenulových prvků v lexikografickém pořadí, prvek hodnoty *val[i]* je tedy v matici na pozici $(row[i], col[i])$.

Formát COO je vhodný pro rychlou konstrukci řídkých matic a pro rychlý převod do ostatních formátů pro ukládání řídkých matic. Avšak je nevhodný pro aritmetické operace a oproti formátům CSR a CSC zabírá více paměti.[9] Paměťová složitost COO je $\mathcal{O}(3nnz)$.

Př. Mějme čtvercovou matici $\mathbb{A}$ řádu $n = 6$ a $nnz = 5$.

$$\mathbb{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{pmatrix}$$

Uložení této matice ve formátu COO by bylo:

row	1	2	2	5	6
col	2	2	3	4	6
val	1	2	3	4	5

Tabulka 2.1: Matice uložená ve formátu COO

2.3 CSR - Compressed Sparse Row

Formát CSR, česky komprimované řídké řádky, je modifikovaný souřadnicový formát pro vyšší efektivitu při ukládání více nenulových prvků. K uložení matice také využívá tři jednorozměrná pole *row_ptr, col_ind* a *val.* Stejně jako v souřadnicovém formátu jsou v poli *val* uloženy hodnoty nenulových prvků a v poli *col_ind* sloupcové indexy nenulových prvků v lexikografickém pořadí. Komprimace spočívá v tom, že místo toho, aby se ukládaly řádkové indexy pro každý nenulový prvek matice, se v poli *row_ptr* ukládají ukazatele (indexy) do pole *val*, které specifikují začátky a konce řádků. Při procházení matice se tedy v poli *val* mezi indexy *row_ptr[i-1]* a *row_ptr[i]* nacházejí hodnoty prvků na *i-tém* řádku matice.[4] Pole *row_ptr* je o jedna delší než výška matice, aby bylo možno určit konec posledního řádku.

Pokud bychom předpokládali obecnou matici o rozměrech $m \times n$ s nnz nenulovými prvky, tak velikosti polí *val* a *col_ind* budou nnz a velikost pole *row_ptr* bude $m+1$. Z toho vyplývá, že paměťová složitost CSR je $\mathcal{O}(2nnz + m + 1)$ pro matici o rozměrech $m \times n$ a $\mathcal{O}(2nnz + n + 1)$ pro čtvercovou matici řádu n.

[4] Předpokládáme indexování polí od 0 a indexování matic od 1.

Př. Mějme čtvercovou matici $\mathbb{A}$ řádu $n = 6$ a $nnz = 8$.

$$\mathbb{A} = \begin{pmatrix} 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 \\ 4 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 & 8 \end{pmatrix}$$

Uložení této matice ve formátu CSR by bylo:

Tabulka 2.2: Matice uložená ve formátu CSR

Index	0	1	2	3	4	5	6	7
row_ptr	0	2	3	5	6	7	8	
col_ind	2	3	6	1	3	5	6	6
val	1	2	3	4	5	6	7	8

2.4 CSC - Compressed Sparse Column

CSC, česky komprimované řídké sloupce, je analogický formát k formátu CSR. Narozdíl od formátu CSR je matice ukládána a procházena po sloupcích. Matice je v paměti opět reprezentována třemi jednorozměrnými poli *col_ptr, row_ind* a *val*. V polích *val*, respektive *row_ind* jsou uloženy hodnoty, respektive řádkové indexy nenulových prvků matice v kolexikografickém pořadí[5]. V poli *col_ptr* jsou ukládány indexy do pole *val*, které indikují začátky a konce sloupců. Hodnoty prvků matice v i-tém sloupci najdeme v poli *val* mezi indexy *col_ptr[i-1]* a *col_ptr[i]*.

Velikosti polí *row_ind, val* jsou podobně jako v předchozím případě nnz a velikost pole *col_ptr* je $n+1$. Z toho vyplývá paměťová složitost $\mathcal{O}(2nnz+n+1)$ pro matici o rozměrech $m \times n$ a $\mathcal{O}(2nnz + n + 1)$ pro čtvercovou matici řádu n.

[5]pořadí shora dolů, zleva doprava

Př. Mějme čtvercovou matici $\mathbb{A}$ řádu $n = 6$ a $nnz = 8$.

$$\mathbb{A} = \begin{pmatrix} 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 \\ 4 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 & 8 \end{pmatrix}$$

Uložení této matice ve formátu CSC by bylo:

Tabulka 2.3: Matice uložená ve formátu CSC

Index	0	1	2	3	4	5	6	7
col_ptr	0	1	2	4	4	5	8	
row_ind	3	1	1	3	4	2	5	6
val	4	1	2	5	6	3	7	8

Kapitola **3**

Algoritmy násobení matic

Při násobení matic $\mathbb{C} = \mathbb{A}\mathbb{B}$ budeme předpokládat, že výsledná matice je vždy hustá. Ve všech algoritmech jsou matice indexovány od 1 a pole jsou indexována od 0. V této kapitole bylo čerpáno z těchto zdrojů [6][8][10][11].

3.1 Klasický algoritmus násobení matic

Klasický algoritmus pro násobení dvou matic vychází z definice uvedené 1.3.3. Pseudokód pro tento algoritmus je popsán v 1.

Algorithm 1 Klasický algoritmus podle definice

```
1: procedure MM_DEF(A,B)
2:     Alokuj matici D a vyplň 0
3:     for i ← 0 to A.height do
4:         for j ← 0 to B.width do
5:             for k ← 0 to A.width do
6:                 D[i][j] += A[i][k] * B[k][j]
7:     return D
```

V pseudokódu je vidět, že algoritmus ve třech cyklech postupně vybírá řádky z matice $\mathbb{A}$, sloupce z matice $\mathbb{B}$ a poté postupně vynásobí a sečtě n[6] prvků i-tého řádku matice $\mathbb{A}$ a j-tého sloupce matice $\mathbb{B}$. Pro výpočet je potřeba projít kombinace všech řádků matice $\mathbb{A}$ a všech sloupců matice $\mathbb{B}$. Pokud předpokládáme matice $\mathbb{A} \in \mathbb{R}^{m,n}$ a $\mathbb{B} \in \mathbb{R}^{n,p}$, potom to znamená mp kombinací a u každé z nich provádíme n násobení a n sčítání. Z toho nám vyplývá časová složitost $\mathcal{O}(mp(n+n)) = \mathcal{O}(mpn)$ a v případě násobení čtvercových matic řádu n je časová složitost $\mathcal{O}(n^3)$.

[6] n je šířka $\mathbb{A}$ i výška $\mathbb{B}$

3.2 Strassenův algoritmus

Strassenův algoritmus je algoritmus pro násobení matic, který byl v roce 1969 publikován německým matematikem Volkerem Strassenem. Ve své publikaci dokázal, že klasický algoritmus s asymptotickou složitostí $\mathcal{O}(n^3)$ není optimální. Algoritmus využívá toho, že operace sčítání/odčítání jsou rychlejší než operace násobení. Strassenův algoritmus pro násobení matic z $\mathbb{R}^{2,2}$ potřebuje celkem 7 operací násobení a 18 operací sčítání/odčítání narozdíl od klasického algoritmu, který potřebuje 8 operací násobení a 4 operace sčítání.[10]

Počítáme-li maticový produkt $\mathbb{C} = \mathbb{AB}$, tak musí platit, že matice $\mathbb{A}, \mathbb{B} \in \mathbb{R}^{n,n}$, kde $n = 2^k$ a $k \in \mathbb{N}_0$. Pokud tato podmínka neplatí, doplníme sloupce a řádky matic nulami, tak aby byla podmínka splněna. Nyní lze každou z matic rozdělit do 4 blokových matic o rozměrech $\frac{n}{2} \times \frac{n}{2}$. Matice označíme podle následujícího schématu:

$$\mathbb{A} = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}, \mathbb{B} = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}, \mathbb{C} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}.$$

Pokud bychom použili klasický algoritmus, matici $\mathbb{C}$ bychom získali takto:

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Tento algoritmus provede celkem

$$T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$$

výpočetních operací. Celková složitost operací sčítání je $\Theta(n^2)$ (složitost součtu matic o velikosti $\frac{n}{2} \times \frac{n}{2}$ je $\frac{n^2}{4} = \Theta(n^2)$). Člen $8T(\frac{n}{2})$ nám značí osm rekurzivních volání maticového násobení na matice polovičního řádu. Aplikováním mistrovské metody 1.4.4 dostáváme asymptotickou složitost

$$T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3).$$

Volker Strassen ale dokázal přijít na nový způsob, jak vyjádřit blokové matice $C_{1,1}..C_{2,2}$.

$$
\begin{aligned}
C_{1,1} &= P1 + P4 - P5 + P7 \\
C_{1,2} &= P3 + P5 \\
C_{2,1} &= P2 + P4 \\
C_{2,2} &= P1 - P2 + P3 + P6
\end{aligned}
$$

K vyjádření matice $\mathbb{C}$ použil sedm pomocných matic $P1..P7$, které definoval takto:

$$
\begin{aligned}
P1 &= (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}) \\
P2 &= (A_{2,1} + A_{2,2}) \cdot B_{1,1} \\
P3 &= A_{1,1} \cdot (B_{1,2} - B_{2,2}) \\
P4 &= A_{2,2} \cdot (B_{2,1} - B_{1,1}) \\
P5 &= (A_{1,1} + A_{1,2}) \cdot B_{2,2} \\
P6 &= (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2}) \\
P7 &= (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2})
\end{aligned}
$$

Pseudokód výše popsaného algoritmu je možné vidět v 2.

Algorithm 2 Strassenův algoritmus

```
 1: procedure MM_STRASSEN(A,B,n)
 2:     if n = 1 then
 3:         return A × B
 4:     else
 5:         m ← n/2
 6:         Výpočet submatic A1,1, B1,1..A2,2, B2,2 za použití m
 7:         MM_Strassen(A1,1 + A2,2, B1,1 + B2,2, m)        ▷ P1
 8:         MM_Strassen(A2,1 + A2,2, B1,1, m)               ▷ P2
 9:         MM_Strassen(A1,1, B1,2 - B2,2, m)               ▷ P3
10:         MM_Strassen(A2,2, B2,1 - B1,1, m)               ▷ P4
11:         MM_Strassen(A1,1 + A1,2, B2,2, m)               ▷ P5
12:         MM_Strassen(A2,1 - A1,1, B1,1 + B1,2, m)        ▷ P6
13:         MM_Strassen(A1,2 - A2,2, B2,1 + B2,2, m)        ▷ P7
14:         C1,1 ← P1 + P4 - P5 + P7
15:         C1,2 ← P3 + P5
16:         C2,1 ← P2 + P4
17:         C2,2 ← P1 - P2 + P3 + P6
18:         Spojení C1,1, C1,2, C2,1, C2,2 do výsledné matice C
19:         return C
```

Jak je vidět v pseudokódu, algoritmus provádí sedm rekurzivních volání (řádky 7-13). Operace na řádcích 14 až 17 mají časovou složitost $\Theta(n^2)$.

Ostatní operace jsou konstantní. Vychází nám tedy

$$T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$$

výpočetních operací. Použitím mistrovské metody získáváme asymptotickou složitost Strassenova algoritmu

$$T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81}).$$

3.3 Další algoritmy pro rychlé násobení matic

Strassen dokázal, že klasický algoritmus se složitostí $\mathcal{O}(n^3)$není optimální. Na jeho práci navázaly další algoritmy, které se snažily snížit časovou složitost násobení matic. **Shmuel Winograd** vylepšil Strassenův algoritmus, jeho vylepšený algoritmus vyžaduje pro výpočet součinu matic z $\mathbb{R}^{2,2}$ 7 operací násobení a 15 operací sčítání/odčítání[10] (oproti originálu, který potřebuje 7 násobení a 18 sčítání/odčítání), jeho asymptotická složitost $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.81})$ ale zůstala nezměněná.

Algoritmus Coppersmith-Winograd[12] s asymptotickou složitostí $\mathcal{O}(n^{2.376})$ byl představen v roce 1990 a až do roku 2010 to byl algoritmus s nejlepší složitostí. V roce 2010 **Andrew Stothers**[13] svou úpravou C-W algoritmu získal mírně lepší složitost $\mathcal{O}(n^{2.37369})$. V následujícím roce dosáhla dalšího zlepšení **Virginia Vassilevska Williamsová**[14] se složitostí $\mathcal{O}(n^{2.372873})$. V současné době je nejlepší algoritmus **François Le Galla**[15], který v roce 2014 přišel se svým algoritmem se složitostí $\mathcal{O}(n^{2.3728639})$. Tyto algoritmy jsou zřídka využívány v praxi, kvůli své vysoké multiplikativní konstantě, jejich rychlost se uplatňuje až u matic velmi vysokých řádů. Většina těchto pokročilých algoritmů je založena na reprezentaci maticového násobení pomocí trilineárních forem nebo využívají teorie grup (algoritmus **Cohn-Umans**[16]).

3.4 Násobení matic v řídkém formátu

Mějme matice A, B uložené v řídkém formátu. Jelikož v této práci předpokládáme výslednou matici hustou, tak budeme matici $C = A \cdot B$ ukládat v klasickém hustém formátu. Z definice násobení matic1.3.3 je zřejmé, že prvky, které se násobení zúčastní musí splňovat podmínku

$$a.col_index == b.row_index,$$

kde a, b jsou prvky matice A, respektive B a *col_index*, resp. *row_index* jsou sloupcový, resp. řádkový index. Násobení se tedy zúčastní takové prkvy, u kterých je sloupcová souřadnice z matice A rovna řádkové souřadnici z matice B. Výsledky těchto částečných součinů potom přičteme na příslušné místo ve výsledné matici C, na $C[a.row_index][b.col_index]$. Proto je nutné si výslednou matici C předpřipravit a vyplnit ji nulami.

Při násobení řídkých čtvercových matic A a B řádu n můžeme vynechat násobení takových prvků, kdy jeden z nich je nulový. Protože násobíme každý řádek s každým sloupcem, tak bude počet operací při tomto násobení dán vzorcem:

$$\sum_{i=1}^{n} nnzr_{A,i} \cdot nnzc_{B,i} [8],$$

kde $nnzr_{A,i}$ je počet nenulových prvků v i-tém řádku matice A a $nnzc_{B,i}$ je počet nenulových prvků v i-tém sloupci matice B. Složitost tohoto algoritmu by se dala označit jako $\mathcal{O}(mn)$, kde $m = max(A.nnz, B.nnz)$. Pro matice, které neobsahují žádný nulový prvek platí $m = n^2$. Tento odhad složitosti je však pro řídké matice velmi pesimistický, protože předpokládá, že se všechny prvky matice A násobí se všemi prvky matice B.

3.4.1 Násobení ve formátu COO

Vstupní matice uložené ve formátu COO2.2 si označme A_{COO} a B_{COO}. Algoritmus pro násobení matic v tomto formátu bude fungovat tak, že postupně budeme procházet prvky matice A_{COO} a ke každému prvku budeme hledat prvky z matice B_{COO}, tak aby splňovaly podmínku 3.4. Pro prvky účastnících se násobení tedy musí platit podmínka

$$A_{COO}.col[i] == B_{COO}.row[j],$$

kde $0 \leq i < A_{COO}.nnz$ a $0 \leq j < B_{COO}.nnz$. Algoritmus postupně prochází všechny nenulové prvky matice A_{COO}, kouká na jejich sloupcový index a pak v poli $B_{COO}.row$ prochází řádkové indexy prvků matice B_{COO}. Pokud se indexy rovnají, prvky vynásobí a přičte do výsledné matice C na souřadnice $C[A_{COO}.row[i]][B_{COO}.col[j]]$. Pokud v poli $B_{COO}.row$ narazíme na řádkový index vyšší než sloupcový index prvku $A_{COO}.col[i]$, tak můžeme vyhledávání přerušit, protože prvky v poli $B_{COO}.row$ jsou seřazeny vzestupně[7]. Pseudokód tohoto algoritmu lze vidět v 3.

Algorithm 3 Násobení ve formátu COO

```
1: procedure MM_COO(A,B)
2:     Alokuj matici C a vyplň 0
3:     for i ← 0 to A.nnz do
4:         for j ← 0 to B.nnz do
5:             if A.col[i] < B.row[j] then break
6:             if A.col[i] = B.row[j] then
7:                 C[A.row[i]-1][B.col[j]-1] += A.val[i] * B.val[j]
8:     return C
```

[7]V této práci předpokládáme lexikografické pořadí prvků ve formátu COO2.2.

3.4.2 Násobení ve formátu CSR

Protože formát CSR2.3 vychází z formátu COO, je algoritmus pro násobení matic uložených v tomto formátu velmi podobný předchozímu algoritmu. Díky předpočítaným začátkům a koncům řádků v poli *row_ptr* víme, kde přesně se nachází první prvek řádku a také počet prvků v každém řádku. Tyto informace nám značně ulehčují vyhledávání násobitelných prvků matic A a B.

Algoritmus postupně prochází nenulové prvky matice A_{CSR} a kouká na jejich sloupcový index, označme ho $a.col_idx$. Díky tomu, že je matice B_{CSR} uložena ve formátu CSR víme, že indexy hodnot prvků matice B_{CSR} které se budou násobit s hodnotou vybraného prvku matice A_{CSR} budou v poli $B_{CSR}.val$ mezi indexy $B_{CSR}.row_ptr[a.col_idx-1]$ a $B_{CSR}.row_ptr[a.col_idx]$. Prvky v cyklu vynásobíme a přičteme na správné místo do matice C. Pseudokód algoritmu je uveden v 4.

Algorithm 4 Násobení ve formátu CSR

```
1: procedure MM_CSR(A,B)
2:     Alokuj matici C a vyplň 0
3:     for i ← 0 to A.height do
4:         for arp ← A.row_ptr[i] to A.row_ptr[i+1] do
5:             for brp ← B.row_ptr[A.col_ind[arp]-1] to B.row_ptr[A.col_ind[arp]] do
6:                 C[i][B.col_ind[brp]-1] += A.val[arp] * B.val[brp]
7:     return C
```

3.4.3 Násobení ve formátu CSC

Násobení matic ve formátu CSC je velmi podobné násobení předchozímu. Postupně procházíme prvky matice B a koukáme na jejich řádkové indexy, poté díky poli $A.col_ptr$ rychle najdeme prvky v matici A, které se s vybraným prvkem z B násobí a přičteme na správné místo ve výsledné matici C. Pseudokód je možné vidět v 5.

Algorithm 5 Násobení ve formátu CSC

```
1: procedure MM_CSC(A,B)
2:     Alokuj matici C a vyplň 0
3:     for j ← 0 to B.width do
4:         for bcp ← B.col_ptr[j] to B.col_ptr[j+1] do
5:             for acp ← A.col_ptr[B.row_ind[bcp]-1] to A.col_ptr[B.row_ind[bcp]] do
6:                 C[A.row_ind[acp]-1][j] += A.val[acp] * B.val[bcp]
7:     return C
```

3.4.4 Násobení ve formátu CSC × CSR

V této variantě násobení máme matici A uloženou ve formátu CSC a matici B ve formátu CSR. V algoritmu procházíme „zároveň“ i-tý sloupec matice A a i-tý řádek matice B. Jelikož je splněna podmínka, že sloupcový index matice A se rovna řádkovému indexu matice B, tak tyto prvky můžeme vynásobit a

přičíst na správné místo ve výsledné matici C. Pseudokód toho algoritmu by mohl vypadat následovně:

Algorithm 6 Násobení ve formátu CSC x CSR

```
1: procedure MM_CSC_CSR(A,B)
2:     Alokuj matici C a vyplň 0
3:     for i ← 0 to A.width do
4:         for acp ← A.col_ptr[i] to A.col_ptr[i+1] do
5:             for brp ← B.row_ptr[i] to B.row_ptr[i+1] do
6:                 C[A.row_ind[acp]-1][B.col_ind[brp]-1] += A.val[acp] * B.val[brp]
7:     return C
```

Kapitola 4

Analýza a návrh

4.1 Existující řešení

4.1.1 Akademické práce

Lukáš Lichý ve své bakalářské práci[6] popsal formáty pro ukládání řídkých matic COO, CSR/CSC a MSR/MSC a algoritmy pro násobení matic v těchto formátech. Po tom co analyzoval existující open-source knihovny řešící problém násobení řídkých matic, svoje algoritmy a formáty implementoval v jazyce C++ a provedl měření jejich výkonu na platofrmách AMD, Intel a SPARC. Výkonnost své implementace změřil dokonce i na DLX simulátoru.

V bakalářské prácí „**Vliv formátu uložení řídké matice na výkonnost násobení řídkých matic**“[8] nastudoval Tomáš Nesrovnal formáty uložení řídkých matic COO, CSR, BSR a Quadtree. Navrhnul modifikaci formátu Quadtree, který nazval KAT. V jazyce C poté implementoval násobení matice vektorem a násobení dvou matic ve formátech COO, CSR, BSR, Quadtree i KAT. Srovnal paměťové a časové složitosti implementovaných algoritmů a došel k závěru, že jeho formát KAT je paměťově náročnější než formát CSR, ale je vhodný pro matice, které obsahují hustší bloky nebo tolik prvků, že začne lépe využívat cache než formát CSR.

4.1.2 Knihovny

Eigen[17] je knihovna pro jazyk C++, která obsahuje mnoho funkcí a tříd řešící problémy z lineární algebry, šířena od verze 3.1.1 pod open-source licencí MPL2[18]. Knihovna je pravidelně aktualizována. Eigen podporuje jak malé matice o fixní velikost, tak velké husté matice, nechybí ani podpora pro práci s řídkými maticemi. Eigen využívá pouze standardní knihovnu C++ a pro jeho použití není nutné instalovat žádné externí knihovny. Knihovna obsahuje pouze hlavičkové soubory (lze je stáhnout z oficiálních stránek projektu), které stačí při kompilaci přilinkovat pomocí odpovídajícího přepínače, v případě překladače GNU GCC např. takto:

```
g++ -I /path/to/eigen/ muj_prog.cpp -o muj_prog
```

Pro práci s řídkými maticemi slouží třída `SparseMatrix`[1]. Tato třída pro reprezentaci řídkých matic implementuje mírně poupravené formáty CSR/CSC popsané v sekcích 2.3 a 2.4. Defaultně je využit formát ve sloupcovém uspořádání (tedy upravený CSC), který obsahuje tato čtyři pole:

- **`Values`** (původně *`val`*) - obsahuje hodnoty nenulových prvků
- **`InnerIndices`** (původně `row_ind`) - obsahuje řádkové indexy nenulových prvků
- **`OuterStarts`** (původně `col_ptr`) - obsahuje ukazatele do dvou předchozích polí, které označují přechod na další sloupec
- **`InnerNNZs`** - ukládá počet nenulových prvků v jednotlivých sloupcích

Př. Mějme čtvercovou matici $\mathbb{A}$ řádu $n = 5$ a $nnz = 8$. Přepokládejme indexaci matic i polí od 0.

$$\mathbb{A} = \begin{pmatrix} 0 & 3 & 0 & 0 & 0 \\ 22 & 0 & 0 & 0 & 17 \\ 7 & 5 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 8 \end{pmatrix}$$

Uložení této matice je pak znázorněno v tabulce 4.1.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Values	22	7	_	3	5	14	_	_	1	_	17	8
InnerIndices	1	2	_	0	2	4	_	_	2	_	1	4
OuterStarts	0	3	5	8	10	12						
InnerNNZs	2	2	1	1	2							

Tabulka 4.1: Uložení řídké matice v upraveném formátu CSC ve knihovně Eigen[1]

Prkvy v polích **`Values`** a **`InnerIndices`** označené jako '_' znázorňují volná místa pro efektivnější vkládání nových prvků. Při předpokladu, že nedojde k realokaci, tzn. že přidáváme do sloupce s volným místém '_', je složitost vložení nového prvku $\mathcal{O}(nnz_j)$, kde nnz_j je počet nenulových prvků v j-tém sloupci. Oproti původnímu formátu CSC bylo nutné zavést pole InnerNNZs, kvůli tomu, že počet nenulových prvků v jednotlivých sloupcích již nelze zjistti vztahem **`OuterStars[j + 1] - OuterStarts[j]`** jako v původním formátu. Důvodem jsou právě vynechaná místa pro rychlejší vkládání prvků. Aby byla zachování kompatibilita s ostatními knihovnama, tak Eigen samozřejmě implementuje i původní verze formátů CSR/CSC. Pro převod do původního formátu

slouží metoda `SparseMatrix::makeCompressed()`. Nechybí ani podpora základních aritmetických operací přetíženými operátory jako například sčítání a odčítání (i s hustými maticemi), transpozice či součin matic (řídká × řídká i řídká × hustá).

Armadillo[19] je open-source knihovna určená pro řešení problémů v lineární algebře a vědecké výpočty pro jazyk C++ šířena pod licencí Apache 2.0[20]. Knihovna poskytuje API podobné programovacímu jazyku MATLAB. K využití této knihovny je zapotřebí mít nainstalované knihovny OpenBLAS[21] a LAPACK[22]. Mnoho populárních Linuxových distribucí poskytuje předpřipravené balíčky, které lze nainstalovat z oficiálních repozitářů. Knihovna podporuje práci s hustými i řídkými maticemi. Samozřejmostí je podpora operací s celými, reálnými a komplexními čísly. Narozdíl od Eigenu je pro práci s řídkými maticemi využit klasický formát CSC. Nechybí podpora maticového násobení jak v hustém formátu, tak v řídkém formátu CSC, formáty lze násobit i mezi sebou, ale je třeba brát v potaz, že rychlost záleží na pořadí činitelů (hustá × řídká je zpravidla rychlejší než řídká × hustá)[23].

Blaze[24] je open-source C++ knihovna pro práci s hustými a řídkými maticemi šířena pod licencí BSD. První verze knihovny byla vydána v roce 2012. Podle benchmarků na oficiálních stránkách by knihovna měla být rychlejší než Armadillo i Eigen. Knihovna implementuje několik tříd pro práci s maticemi, pro práci s řídkými maticemi slouží třída `CompressedMatrix`, která matici reprezentuje ve formátech CSR/CSC.

uBLAS[25] je C++ knihovna zaměřená na řešení základních problémů v lineární algebře. Je součástí rozsáhlejší knihovny *Boost*. Knihovna již od roku 2008 nebyla aktualizována, takže jí chybí modernější funkce z C++11. Z tohoto důvodu také není tato knihovna nejrychlejší, existují modernější a rychlejší alternativy. Na druhou stranu stačí na řešení většiny jednoduchých problémů, funguje na různých platformách a její použití je přímočaré. Pro práci s řídkými maticemi jsou implementovány formáty COO, CSR/CSC a $MappedMatrix$. Formát $MappedMatrix$ ukládá matici v kontejneru $std :: map$. Jsou podporovány základní aritmetické operace jako sčítání, odčítání či násobení se skalárem. Pro součin matic jsou implementovány funkce $prod()$ a $sparse_prod()$.

SciPy[26] je knihovna ve skriptovacím jazyce Python pro vědecké a technické výpočty. Součástí knihovny je modul scipy.sparse[27], který slouží pro práci s řídkými maticemi. Pro práci s řídkými maticemi implementuje mimo jiné i formáty COO, CSR a CSC. Knihovna podporuje různé aritmetické operace v těchto formátech i převody mezi těmito formáty. Operace nad řídkými maticemi jsou kvůli rychlosti implementovány v jazyce C++.[28]

4.2 Strassenův algoritmus s řídkými maticemi

Strassenův algoritmus jsem implementoval pro hustý formát, formát COO a formát CSR. Pro řídké formáty jsem implementoval jednoúrovňový Strassenův algoritmus, kdy jsou vstupní matice rozděleny do čtvrtin a uložené v řídkém formátu. Poté je proveden výpočet pomocných matic $P1$-$P7$ pomocí klasického algoritmu pro násobení matic a nakonec je z $P1$-$P7$ vypočtena výsledná matice C. Rozhodl jsem se algoritmus implementovat touto cestou kvůli domněnce, že algoritmus není vhodný pro řídké formáty.

Jeden z problémů je ten, že při násobení, sčítání a odečítání matic může vzniknout matice hustá, v tomto případě se ztrácí výhody využití řídkých formátů. Při těchto operacích také dopředu nevíme počet nenulových prvků ve výsledné matici, proto musíme velikosti polí během výpočtů dynamicky měnit, což vede k vyšší režii. Další problémy při použití tohoto algoritmu souvisí s nutností rekurzivně dělit vstupní matice do menších submatic a s tím spojená režie. Při použití klasického hustého formátu toto není problém, protože matice můžeme dělit na úrovni indexace polí a nemusíme tak rekurzivně vytvářet nová pole. Naopak při použití řídkých formátů je zapotřebí při každém dělení vytvářet nové submatice, z tohoto důvodu se zvyšuje režie a požadavky na paměť.[6] Algoritmus má již ve verzi pro hustý formát vyšší paměťové nároky a vyšší multiplikativní konstantu než klasický algoritmus, přizpůsobením algoritmu na řídké formáty dochází k dalšímu zvýšení těchto hodnot. Implementací jednoúrovňového Strassena jsem se snažil minimalizovat tyto problémy, ale podle výsledků měření v kapitole 6 jsou implementované algoritmy pomalejší než klasické násobení v řídkém formátu, klasické algoritmy začínají dohánět až při vyšším počtu nenulových prvků v násobených maticích.

Kapitola **5**

Implementace

Rozhodl jsem se implementovat všechny formáty uvedené v této práci tzn. hustý formát, formát COO, formát CSR a formát CSC. Implementoval jsem i všechny popsané algoritmy pro násobení matic tzn. klasické násobení v hustém formátu, násobení ve formátu COO, násobení ve formátu CSR/CSC, násobení ve formátech CSC$\times$CSR a Strassenův algoritmus. Strassenův algoritmus byl implementován pouze pro formát hustý, COO a CSR a podporuje pouze čtvercové matice. Implementace obsahuje i jednoduchou funkci pro generování matic o určitých rozměrech s daným počtem nenulových prvků.

5.1 Použité nástroje

Algoritmy byly implementovány v programovacím jazyce C++. Kromě standardní knihovny a knihovny STL jsem v programu využil funkce z knihovny Matrix Market I/O, která je napsána v jazyce C a je dostupná na oficiálních stránkách projektu Matrix Market[8].

Při implementaci byl program zkompilován překladačem GNU GCC[29] verze 8.2.1 na operačním systému Fedora 29 (Workstation Edition). K analýze a ladění chyb při práci s pamětí byl využit nástroj Valgrind[30] s přepínači `-v -leak-check=full -show-leak-kinds=all`.

5.1.1 Nastavení překladače

Pro kompilaci programu byl použit překladač GNU GCC s následujícími přepínači:

```
g++ -std=c++11 -fopenmp -O3 -ftree-vectorize -ffast-math -mavx
-fopt-info-vec
```

[8]http://math.nist.gov/MatrixMarket/mmio-c.html

- **std=c++11** - zapíná podporu standardu C++11
- **fopenmp** - aktivuje knihovnu OpenMP5.1.3 pro paralelizaci programu
- **O3** - optimalizační přepínač, který aktivuje řadu dalších přepínačů zlepšující výkonnost výsledného programu
- **ftree-vectorize** - aktivace automatické vektorizace pomocí překladače
- **ffast-math** - zapíná skupinu přepínačů k co nejrychlejšímu vyhodnocení výrazů (změna pořadí operací, násobení převrácenou hodnotou místo dělení)
- **mavx** - využití AVX instrukcí při vektorizaci
- **fopt-info-vec** - vypíše informace o tom, jaké části kódu byly vektorizovány

5.1.2 Optimalizační techniky

5.1.2.1 Vektorizace

Vektorizace je technika, kdy je matematická či logická operace provedena na více dat najednou v jedné instrukci. Moderní procesory poskytují podporu pro vektorové operace, kdy je jedna instrukce aplikována na více dat najednou pomocí speciálních SIMD (Single instruction, multiple data) instrukcí. Vektorizace je výhodná zejména v cyklech, kdy jsou nad daty opakovaně prováděny stejné operace. V C/C++ lze vektorizaci aktitovat 4 způsoby[31]:

- Samotný ASM soubor, který je na závěr připojen k výsledku.
- Assemblerový kód vložený přímo ve zdrojovém C/C++ kódu.
- Pomocí intrinsic instrukcí, které jsou namapovány přímo na SIMD instrukce. Překladač se poté postará o generování kódu a použití registrů.
- Automatická vektorizace pomocí kompilátoru.

V této práci je použita automatická vektorizace pomocí kompilátoru. Aby cyklus mohl být vektorizován, měl by splňovat následující podmínky:

- Cyklus je nejvnitřnější.
- Cyklus neobsahuje datové závislosti, které by mohly narušit vektorizaci.
- Cyklus by neměl obsahovat podmínky.
- Data by měla být v paměti vhodně umístěny (měla by navazovat).
- Je proveden dostatek iterací.
- Uvnitř cyklu by neměly být volány žádné funkce či procedury, ani by cyklus neměl být předčasně ukončen.

5.1.2.2 Rozbalení cyklů

Moderní CPU mají hluboko pipeline a dosahují maximální výkonnosti pokud vykonávají delší úsek kódu bez podmíněných skoků. Rozbalením cyklu (loop unrolling) se sníží počet iterací a tím klesne i počet podmíněných skoků. Vnitřek cyklu se prodlouží a díky tomu mohou být instrukce uvnitř lépe naplánovány[32]. Jak rozbalení cyklu vypadá si ukážeme na následujícím příkladu 7.

Algorithm 7 Rozbalení cyklu

```
 1: for i ← 0 to (n - 1) do                          ▷ původní cyklus
 2:     DoStuff(i)
 3:
 4: for i ← 0 to (n - Uf) step Uf do ▷ cyklus po rozbalení s faktorem Uf
 5:     DoStuff(i)
 6:     DoStuff(i + 1)
 7:     DoStuff(i + 2)
 8:     ...
 9:     DoStuff(i + (Uf - 1))
10:
11: for i ← n - (n mod Uf) to (n - 1) do
12:     DoStuff(i)     ▷ clean-up cyklus v případě, kdy n není dělitelné Uf
```

Rozbalení cyklu je určeno parametrem Uf (unrolling factor). Kontrolující proměnná cyklu se při každé iteraci inkrementuje o Uf a v těle cyklu se provede kód pro hodnoty mezi aktuální a následující iterací. V případě, že n není násobek Uf následuje další tzv. clean-up cyklus, který se postará o zbylé iterace. Rozbalením cyklu počet podmíněných skoků klesá na $(n/\texttt{Uf})+(n \mod \texttt{Uf})+1$.

5.1.2.3 Proložení cyklů

Proložení cyklů (loop tiling, loop blocking) je technika, která transformuje cyklus tak, aby byla co nejlépe využita pamět cache. Pokud je rozsah cyklu větší než kapacita cache paměti, vyplatí se cyklus rozdrobit na 2 cykly. Vnitřní cyklus má Bf iterací a data v něm použitá se vejdou do paměti cache, vnější cyklus zajišťuje sémantickou shodu[32]. Loop tiling si ukážeme na následujícím příkladu 8.

Algorithm 8 Proložení cyklů

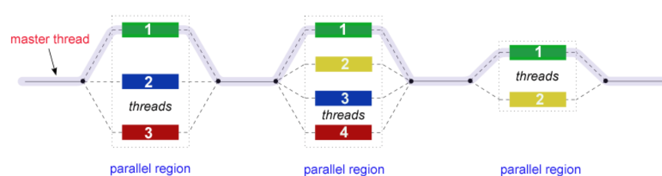
```
1: for i ← 0 to (m - 1) do                                  ▷ původní cyklus
2:     for j ← 0 to (n - 1) do
3:         DoStuff(i, j)
4:
5: for i1 ← 0 to (m - 1) step Bf do                         ▷ po proložení
6:     for j1 ← 0 to (n - 1) step Bf do
7:         for i2 ← i1 to min(i1 + Bf - 1, m) do            ▷ vejdou se do cache
8:             for j2 ← j1 to min(j1 + Bf - 1, n) do
9:                 DoStuff(i2, j2)
```

Proložení je dáno parametrem `Bf` (block factor), který závisí na parametrech cache paměti.

5.1.3 Knihovna OpenMP

OpenMP[33] (Open Multi-Processing) je multiplatformní API pro vícevláknové programování nad sdílenou pamětí v programovacích jazycích C, C++ a Fortran na operačních systémech Linux, Windows, macOS, Solaris a dalších. OpenMP je spravována neziskovou organizací OpenMP ARB, jejíž součástí jsou například firmy jako AMD, Intel, Nvidia, IBM, Oracle a další[34].

OpenMP využívá k paralelizaci tzv. fork-join model5.1. Každý program nejdříve začíná sekvenčně, dokud hlavní vlákno(master thread) nenarazí na paralelní region. Hlavní vlákno při vstupu do paralelního regionu vytvoří skupinu vedlejších vláken (fork) a kód uvnitř paralelní oblasti je vykonán paralelně, na konci paralelního regionu jsou vlákna synchronizována pomocí bariéry a poté všechna vlákna kromě hlavního zanikají (join).



Obrázek 5.1: Fork-join model[35]

OpenMP se skládá ze tří hlavních komponent:

- direktiva pro překladač
- knihovní funkce
- proměnné prostředí

5.1.3.1 Direktiva pro překladač

OpenMP direktiva jsou využívána pro různé účely jako například vytvoření paralelního regionu, rozdělení částí kódu mezi vlákna, synchronizaci vláken apod. Pokud tyto direktiva chceme využívat například s překladačem GNU GCC, tak musíme program zkompilovat s přepínačem `-fopenmp`, jinak budou direktiva ignorována.

Uvedeme si zde některé základní direktiva:

- `#pragma omp parallel [klauzule...]` - specifikuje části kódu, které mají být vykonány paralelně
- `#pragma omp for [klauzule...]` - rozdělí iterace for cyklu mezi jednotlivá vlákna, defaultně jsou iterace rozděleny mezi vlákna rovnoměrně 1
- `#pragma omp sections + section [klauzule...]` - označuje části kódu, které mohou být vykonány paralelně, tyto nezávislé části jsou označeny direktivou section, každá sekce je provedena pouze jednou 2
- `#pragma omp task [klauzule...]` - vytváří nezávislé úlohy(většinou funkce s určitými parametry), které mohou být vykonány jakýmkoliv vláknem ve skupině 3
- `#pragma omp single [klauzule...]` - označuje část kódu uvnitř paralelního regionu, která má být provedena pouze jedním vláknem
- `#pragma omp critical (jméno)` - určuje sekce kódu, které mohou v danou chvíli být prováděny pouze jedním vláknem
- `#pragma omp atomic` - zaručuje, že daná operace bude provedena atomicky(bez přerušení), jsou podporovány pouze jednoduché operace jako například $++, +, -, >>$ atd.

Klauzule specifikují mnoho věcí, jako například co se stane s proměnnými. Některé základní klauzule:

- `shared(proměnné)` - deklaruje proměnné, které jsou mezi vlákny sdíleny
- `private(proměnné)` - každé vlákno si vytvoří lokální neinicializovanou kopii daných proměnných
- `firstprivate(proměnné)` - každé vlákno si vytvoří lokální kopii daných proměnných a inicializuje jí hodnotou z hlavního vlákna
- `lastprivate(proměnné)` - po skončení paralelního regionu do proměnné v hlavním vlákně zkopíruje poslední známou hodnotu (například hodnotu z poslední iterace for cyklu)

- `nowait` -potlačuje implicitní bariéru u direktiv sections, for či single
- `num_threads(číslo)` - určuje kolik vláken má být v paralelním regionu vytvořeno
- a další...

5.1.3.2 Knihovní funkce

Pro použití knihovních funkcí a datových typů je zapotřebí do zdrojového kódu zahrnout hlavičkový soubor omp.h.

- `omp_set_num_threads(číslo)` - určí počet vláken, které mají být vytvořeny v paralelních regionech
- `omp_get_max_threads()` - zjistí maximální počet dostupných vláken
- `omp_in_parallel()` - zjistí zda se část kódu nachází v paralelní oblasti
- `omp_get_wtime()` - vrací čas v sekundách, který uběhl od nějakého bodu v minulosti
- a další...

5.1.3.3 Proměnné prostředí

Nastavují různé parametry pro běh programu.

- `OMP_NUM_THREADS` - určí maximální počet použitých vláken při provedení programu
- `OMP_NESTED` - povoluje/zakazuje vnořený paralelismus
- `OMP_STACKSIZE` - kontroluje velikost zásobníku pro vytvořená vedlejší vlákna
- a další...

Více o OpenMP si můžete přečíst například z těchto webových zdrojů[33][35][36].

Ukázka kódu 1 Příklad využití direktivy for

```
int main() {
	int arr[100];
	#pragma omp parallel
	{
		#pragma omp for
		for (int i = 0; i < 100; i++) {
			arr[i] += i;
		}
	}

	// Je ekvivalentni s

	#pragma parallel omp for
	for (int i = 0; i < 100; i++) {
		arr[i] += i;
	}

	return 0;
}
```

Ukázka kódu 2 Příklad využití direktivy sections

```
#pragma omp parallel
{
	function_a(); // Funkce provedena vsemi vlakny
	#pragma omp sections
	{
		#pragma omp section
		{
			function_b();
			function_c();
		}
		#pragma omp section
			function_d();
	}
}
```

Ukázka kódu 3 Příklad využití direktivy task

```
#pragma omp parallel
{
        int i, j;
        #pragma omp single
        {
           #pragma omp task
           i = foo(1);
           #pragma omp task
           j = bar(2);
           #pragma omp taskwait
           printf("%d\n", i + j);
        }
}
```

5.1.4 Formát uložení MatrixMarket

Matrix Market (MM)[37] je veřejně dostupný zdroj matic na internetu spravovaný americkým institutem NIST (National Institute of Standards and Technology). Poskytuje matice z různých vědeckých disciplín či generátory matic o daných parametrech. Matice jsou ukládány v souborech ve stejnojmenném formátu. Jsou definovány 2 typy tohoto formátu - formát **array**, který slouží pro ukládání obecných hustých matic a formát **coordinate**, který slouží k uložení řídkých matic, v této sekci se budeme zabývat formátem **coordinate**.

Formát MM se skládá ze čtyř částí:

1. Hlavička - vždy začíná řetězcem „`%%MatrixMarket`", poté následuje o jaký objekt se jedná „`matrix`" či „`vector`", v dalším políčku je určen typ formátu, buďto „`coordinate`" nebo „`array`", následuje datových typ prvků v objektu „`real`", „`double`", „`complex`", „`integer`", „`pattern`" a nakonec je určen typ symetrie - „`general`", „`symmetric`", „`skew-symmetric`", „`hermitian`" (jen pro „`complex`").

2. Volitelné komentáře, které musí začínat znakem '%'.

3. Počet řádků, počet sloupců a počet nenulových prvků v matici ve tvaru `m n nnz`.

4. Samotná data ve tvaru `i j value`, kde `i` je řádkový index, `j` je sloupcový index a `value` je hodnotu prvku matice. U symetrických matic jsou vypsány pouze prvky na diagonále a pod diagonálou.

Př. Mějme čtvercovou matici $\mathbb{A}$ řádu $n = 6$ a $nnz = .5$

$$\mathbb{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{pmatrix}$$

Uložení této matice ve formátu MM by vypadalo takto:

```
%%MatrixMarket matrix coordinate integer general
%Volitelny komentar
%Matice radu 6 s 5 nenulovymi prvky
6 6 5
1 2 1
2 2 2
2 3 3
5 4 4
6 6 5
```

5.2 Popis tříd

Program obsahuje osm tříd, které jsou v této sekci popsány.

5.2.1 Třída Matrix

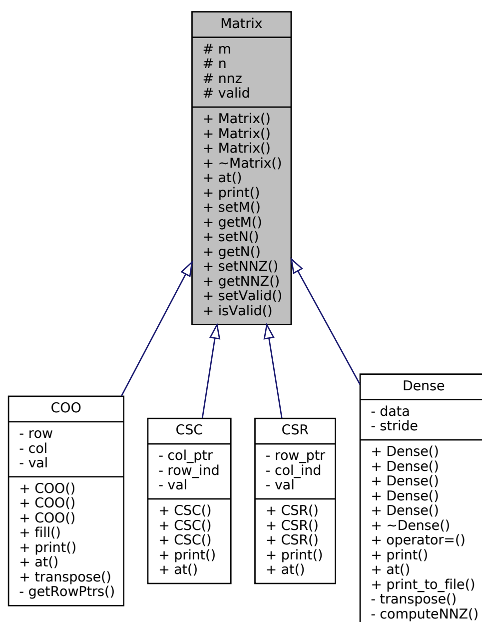
Třída `Matrix` je třída obsahující proměnné a funkce, ze které ostatní třídy implementující formáty pro uložení matic, dědí. Třída obsahuje proměnné `m` a `n`, které určují rozměry matice, proměnnou `nnz`, která určuje počet nenulových prvků v matici a proměnnou `valid` určující, zda-li je matice platná. Třída deklaruje virtuální funkci `print()`, která slouží k výpisu nenulových prvků matice do terminálu. Diagram dědičnosti lze vidět na obrázku 5.2.

Třída `Dense` implementuje hustý formát pro uložení matic popsaný v 2.1. Data jsou uloženy v dynamicky alokovaném jednorozměrném poli `data` o velikosti $m \times n$. Pro jednorozměrné pole jsem se rozhodl proto, že oproti dvourozměrnému poli je jednorozměrné pole alokováno v jednom souvislém bloku v paměti, takže je lépe využívána pamět cache. Prvek matice na `i`-tém řádku a `j`-tém sloupci je uložen v poli `data` na indexu $(i-1) \cdot n + (j-1)$. Kvůli dynamické alokaci paměti bylo nutné naimplemetovat vlastní konstruktor, destruktor, kopírovací konstruktor a operátor=. Metoda `transpose()` vrací transponovanou matici a metoda `computeNNZ()` spočítá v poli `data` počet nenulových prvků.

Třída `COO` je třída implementující formát COO popsaný v 2.2. Obsahuje tři pole o velikosti `nnz` - `row`, `col` a `val` pro uložení řádkového indexu, sloupcového indexu a hodnoty prvku. Metoda `fill()` je využita pro vyplnění matice při načítání ze souboru, metoda `transpose()` vrací transponovanou matici a metoda `getRowPtrs()` je používána při převodu do formátu CSR k získání pole `row_ptr`.

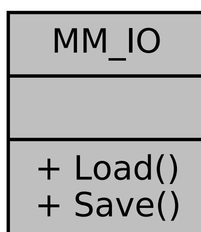
Třída `CSR` implementuje formát CSR popsaný v 2.3. Třída obsahuje dvě pole `col_ind` a `val` o velikosti `nnz` a jedno pole `row_ptr` o velikosti `m + 1`.

Třída `CSC` implementuje formát CSC popsaný v 2.4. Třída obsahuje dvě pole `row_ind` a `val` o velikosti `nnz` a jedno pole `col_ptr` o velikosti `n + 1`.



Obrázek 5.2: Diagram třídy Matrix

5.2.2 Třída MM_IO



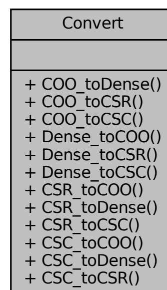
Obrázek 5.3: Diagram třídy MM_IO

Třída MM_IO slouží k načítání a ukládání matic do souborů ve uložených formátu Matrix Market5.1.4. Implementuje dvě statické metody Load() a Save().

Metoda Load() načte soubor a za pomocí funkcí v knihovně Matrix Market I/O přečtě hlavičku souboru. Prvky matice jsou nejdříve načteny do STL kontejneru map<int, double> a poté načteny do matice formátu COO. Metoda nepodporuje načítání souborů typu array a ani matice obsahující komplexní čísla.

Metoda Save() slouží k zápisu matice v hustém formátu do vybraného souboru. Hlavička souboru je zapsána pomocí funkcí z knihovny Matrix Market I/O a poté jsou prvky matice do souboru zapsány funkcí fprintf(). Soubory jsou vždy uloženy v typu coordinate real general.

5.2.3 Třída Convert



Obrázek 5.4: Diagram třídy Convert

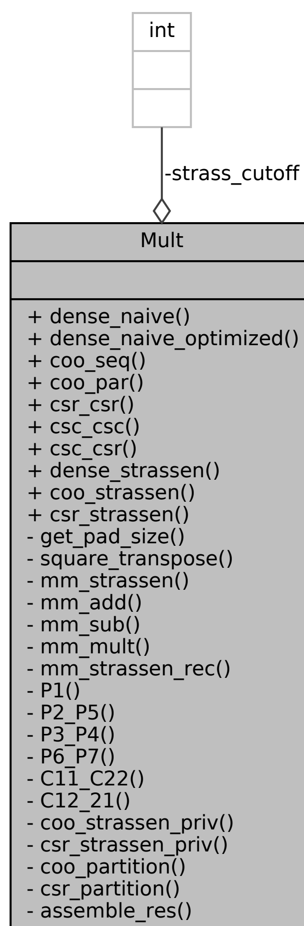
Třída Convert implementuje 12 statických funkcí pro převod mezi všemi implementovanými formáty navzájem. Seznam funkcí lze vidět na obrázku 5.4. Ukázka kódu funkce pro převod formátu COO na CSC je na 4.

Ukázka kódu 4 Převod z formátu COO do CSC

```
bool Convert::COO_toCSC(const COO &coo, CSC &csc) {
        if (!coo.isValid()) {
                return false;
        }

        csc = CSC(coo.m, coo.n, coo.nnz);

        // Spocitam si pocet prvku v~jednotlivych sloupcich
        for (int i = 0; i < coo.nnz; i++) {
                csc.col_ptr[coo.col[i] - 1]++;
        }

        // Spocitani ukazatelu do poli row_ind, val
        for (int i = 0, sum = 0; i < csc.n; i++) {
                int tmp = csc.col_ptr[i];
                csc.col_ptr[i] = sum;
                sum += tmp;
        }
        csc.col_ptr[csc.n] = coo.nnz;

        // Prekopirovani prvku z~COO formatu do CSC
        for (int i = 0; i < csc.nnz; i++) {
                int col = coo.col[i];
                int dest = csc.col_ptr[col - 1];

                csc.row_ind[dest] = coo.row[i];
                csc.val[dest] = coo.val[i];

                csc.col_ptr[col - 1]++;
        }

        // Oprava pole col_ptr, ktere jsme pri kopirovani zmenili
        for (int i = 0, last = 0; i < csc.n + 1; i++) {
                int tmp = csc.col_ptr[i];
                csc.col_ptr[i] = last;
                last = tmp;
        }

        return true;
}
```

5.2.4 Třída Mult

Třída `Mult` je třída implementující 10 hlavních statických metod pro násobení matic v různých formátech. Implementované metody lze vidět na obrázku 5.5. Třída obsahuje jednu statickou proměnnou `strass_cutoff`, která je využita při násobení Strassenovým algoritmem a udává nám hranici, kdy je místo Strassenova algoritmu použit klasický triviální algoritmus. Tuto proměnnou lze při zavolání metody `dense_strassen()` specifikovat, defaultně je nastavena na 1. Algoritmy jsou implementovány podle pseudokódů z kapitoly 3.



Obrázek 5.5: Diagram třídy Mult

Metoda `dense_strassen()` implementuje Strassenův algoritmus z 3.2. Algoritmus je implementován pouze pro čtvercové matice uložené v hustém formátu. Vstupní matice jsou nejdříve doplněny nulami tak, aby po určitém počtu dělení do submatic bylo možné matice vynásobit triviálním algoritmem.

Ukázka kódu 5 Paralelizace násobení ve formátu CSR

```
/*...*/
#pragma omp parallel for
for (int i = 0; i < a.m; i++) {
  for (int arp = a.row_ptr[i]; arp < a.row_ptr[i + 1]; arp++) {
    for (int brp = b.row_ptr[a.col_ind[arp] - 1]; brp < b.row_ptr[a.col_ind[arp]]; brp++) {
      d.data[i * d.stride + (b.col_ind[brp] - 1)] += a.val[arp] * b.val[brp];
    }
  }
}
/*...*/
```

Strassenův algoritmus je rekurzivně volán do té doby, než je velikost matice menší nebo rovna proměnné `strass_cutoff`, poté jsou matice vynásobeny triviální algoritmem. Snažil jsem se minimalizovat počet pomocných polí využitím pointerové aritmetiky a uložením některých pomocných polí do výsledné matice C.

5.3 Paralelizace

K paralelizaci algoritmů jsem použil knihovnu OpenMP popsanou v 5.1.3. Jedna z výhod knihovny OpenMP je, že k paralelizaci programu nemusíme kód nějak výrazně přepisovat. Jak je vidět například v ukázce kódu 5, paralelizace triviálního násobení, násobení ve formátu CSR a násobení ve formátu CSC byla jednoduchá, stačilo pomocí direktivy `#pragma omp parallel for` paralelizovat vnější for cyklus. Při paralelizaci násobení ve formátu COO a CSC $\times$ CSR však nastal problém, jednotlivá vlákna mohla v danou chvíli modifikovat stejná data ve výsledné matici a násobení tak dávalo špatné výsledky. Zkusil jsem do násobení přidat kritickou sekci, ale ta kvůli častému přístupu ke sdíleným datům algoritmus výrazně zpomalila. Algoritmy se mi nepodařilo upravit tak, aby tzv. race condition nevznikaly, a proto jsem se nakonec rozhodl k paralelizaci vnitřního cyklu, což zaručilo, že výsledky byly správné, protože jednotlivá vlákna vždy ve výsledné matici modifikovala různá místa.

Strassenův algoritmus jsem paralelizoval pomocí direktivy `#pragma omp task`, jak je vidět v ukázce kódu 6. Pro výpočet pomocných matic P1 až P7 jsem si vytvořil funkce. Jelikož je výpočet těchto pomocných matic nezávislý, lze ho jednoduše provést vícevláknově. Direktiva `#pragma omp single` je potřeba proto, aby byl každý task vytvořen pouze jednou, jinak by tasky byly vytvořené všemi vlákny z týmu, což v tomto případě nechceme. Jelikož je algoritmus rekurzivní, tak paralelní region vytváříme mimo tuto funkci, rekurzivním vytvářením paralelních regionů by rychle došlo k vyčerpání výpočetních prostředků.

Ukázka kódu 6 Paralelizace Strassenova algoritmu

```
void strassen(a, b, c, n) {
  /*pre-processing, doplneni matic nulami atd.*/
  #pragma omp paralel
  {
    #pragma omp single nowait
      strassen_rec(a, b, c);
  }
  /*...*/
}

void strass_rec(a, b, c, n) {
  /*...*/
  #pragma omp task
    P1(a11, a22, b11, b22, p1, half_n);
  #pragma omp task
    P2_P5(a21, a22, b11, p2, half_n);
  #pragma omp task
    P3_P4(b12, b22, a11, p3, half_n);
  #pragma omp task
    P3_P4(b21, b11, a22, p4, half_n);
  #pragma omp task
    P2_P5(a11, a12, b22, p5, half_n);
  #pragma omp task
    P6_P7(a21, a11, b11, b12, half_n);
  #pragma omp task
    P6_P7(a12, a22, b21, b22, p7, half_n);
  #pragma omp taskwait // synchronizace pred vypoctem vysledne matice
  /*...*/
}
```

Kapitola 6

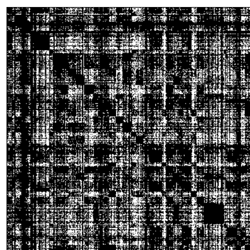
Měření

Měření jsem provedl na fakultním svazku STAR s následující konfigurací:

- CPU: 2x Intel Xeon E5-2630 v4 @ 2.20GHz (dohromady 20 fyzických jader, se zapnutou technologií Hyper-Threading až 40 vláken)
- CPU Cache L1 - L2 - L3: 32KB - 256KB - 25600KB
- RAM: 4x 16GB DDR4 2400MHz RAM
- OS: CentOS 7.4.1708
- Překladač: GCC 4.8.5

Algoritmy jsem změřil na vlastních vygenerovaných maticích i na maticích z repozitáře Matrix Market[37]. Pro ověření správnosti algoritmů jsou matice nejdříve vynásobeny klasickým algoritmem v hustém formátu. Poté jsou vynásobeny ve vybraném řídkém formátu a nakonec jsou tyto dvě výsledné matice porovnány. Kvůli použití datového typu `double` nejsou prvky matic porovnávány přímo, ale sledujeme velikost rozdílu s odchylkou 0.0001. Do časů měření není započítávána doba generování matice či doba načítání matice ze souboru. Každé měření bylo zopakováno 3x a výsledné časy zprůměrovány. Z webu MatrixMarket jsem měření provedl na těchto maticích:

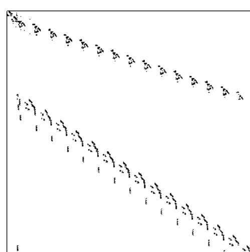
PSMIGR 2: Inter-county migration[38]



Obrázek 6.1: Distribuce nenulových prvků v matici PSMIGR 2

- **Disciplína:** Vnitrostátní migrace v USA
- **Rozměry:** 3140×3140
- **NNZ:** 540022
- **Hustota:** 5.5%

WEST2021: Chemical engineering plant models[39]



Obrázek 6.2: Distribuce nenulových prvků v matici WEST2021

- **Disciplína:** Chemické inženýrství
- **Rozměry:** 2021×2021
- **NNZ:** 7310
- **Hustota:** 0.18%

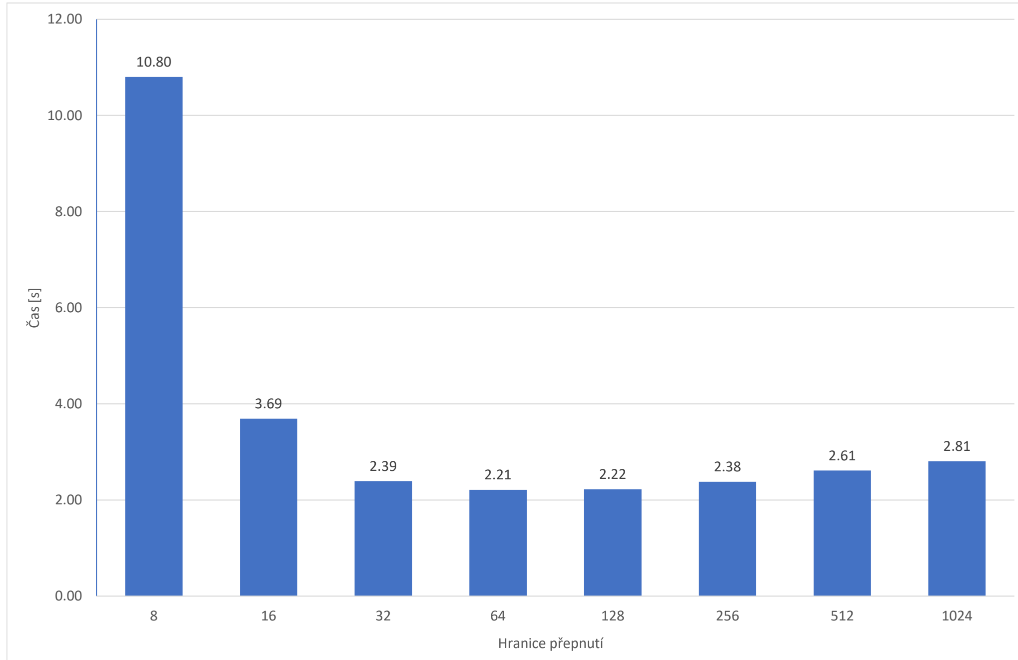
6.1 Hranice přepnutí Strassenova algoritmu

V grafu 6.3 je vidět, že vhodná hranice pro přepnutí ze Strassenova algoritmu na algoritmus klasický je pro výše uvedenou konfiguraci $n = 64$. Tato hranice byla použita u všech následujících měření.

6.2 Doba výpočtu v závislosti na hustotě matice

Graf 6.4 znázorňuje dobu výpočtu algoritmů při změně hustoty vstupních matic. Z grafu je na první pohled zřejmé, že čas výpočtu pro algoritmy využívající hustý formát (Triv, Triv-opt, Dense Strass) se při změně hustoty matice nemění. Výkonnost těchto algoritmů nezávisí na počtu nenulových prvků v matici, ale na rozměrech vstupních matic.

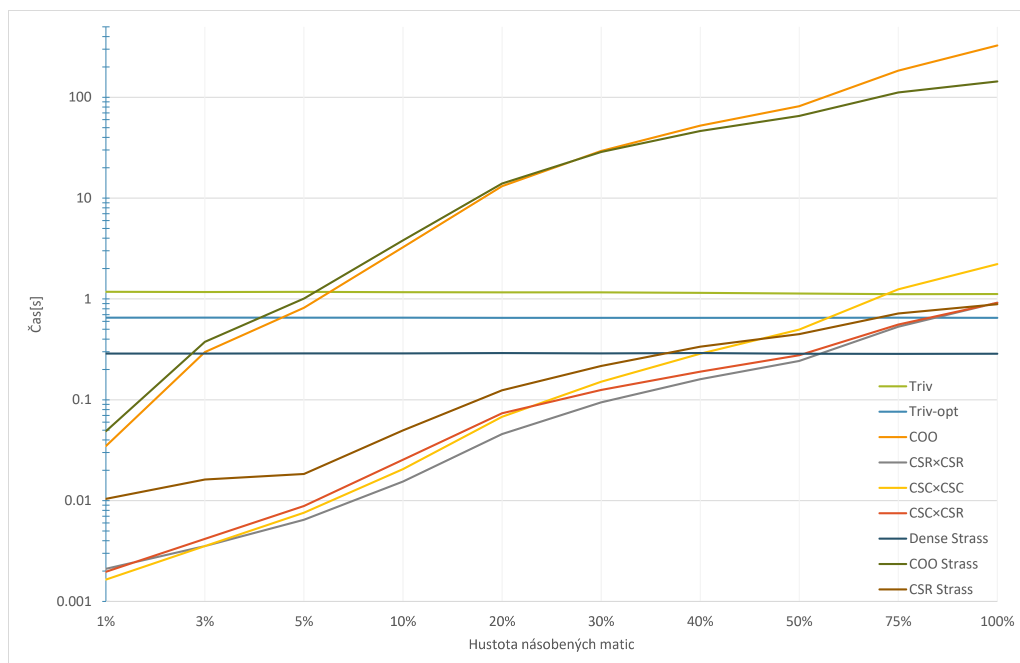
Algoritmy využívající formát COO jsou vhodné pouze pro velmi řídké matice do hustoty 5%. Poté jsou překonány dokonce i klasickým neoptimalizovaným algoritmem podle definice. Jejich doba výpočtu se stoupající hustotou



Obrázek 6.3: Čas výpočtu pro různé hranice přepnutí Strassenova algoritmu, matice WEST20216.2

násobených matic rapidně stoupá a při 100% zaplnění dosahuje řádově stovky vteřin. Strassenův algoritmus pro tento formát zpočátku nedosahuje takových časů jako klasický algoritmus, ale od hustoty matic 20% začíná být rychlejší.

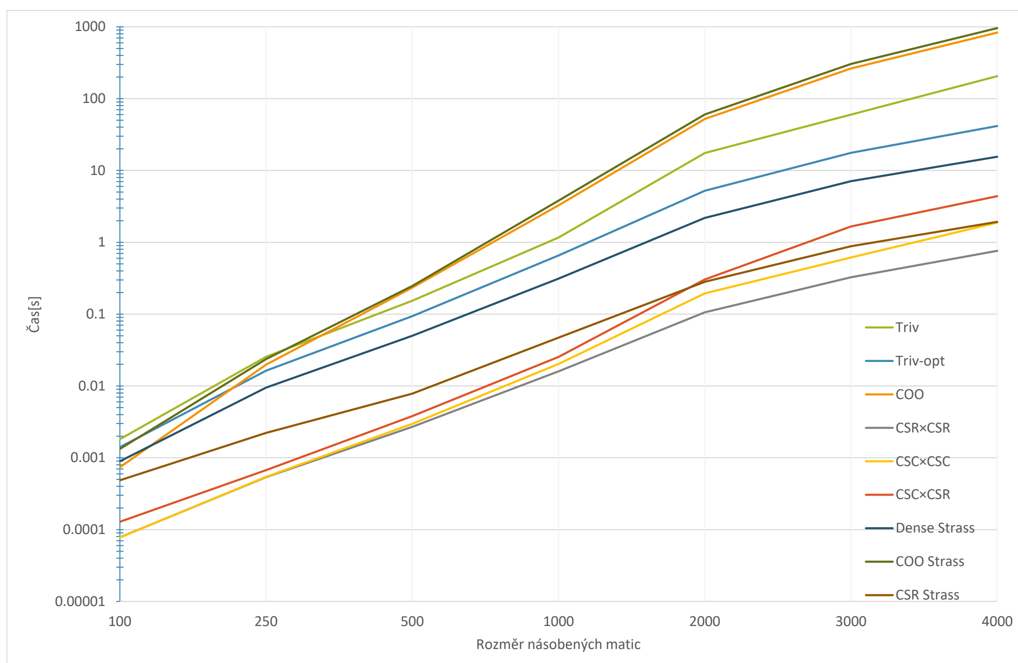
Algoritmy využívající komprimované formáty si po celou dobu vedou dobře, všechny varianty jsou výkonnější než násobení ve formátu COO. Při násobení ve formátu COO by se nejspíše vyplatilo převést do jednoho z komprimovaných formátů a poté vynásobit než provádět násobení ve formátu COO. Nejlepších časů dosahuje násobení ve formátech CSR×CSR, který je překonán Strassenovým algoritmem pro husté matice až při hustotě 50%. I když jsou algoritmy CSC×CSC a CSR×CSR téměř totožné, násobení ve formátu CSC je znatelně pomalejší. Důvodem bude nejspíše horší využití skryté paměti cache. Narozdíl od CSR×CSR vyplňuje CSC×CSC výslednou matici uloženou v hustém formátu po sloupcích, čímž je hůře využita pamět cache, protože prvky v sloupci nejsou v paměti uloženy souvisle za sebou narozdíl od prvků v řádku.



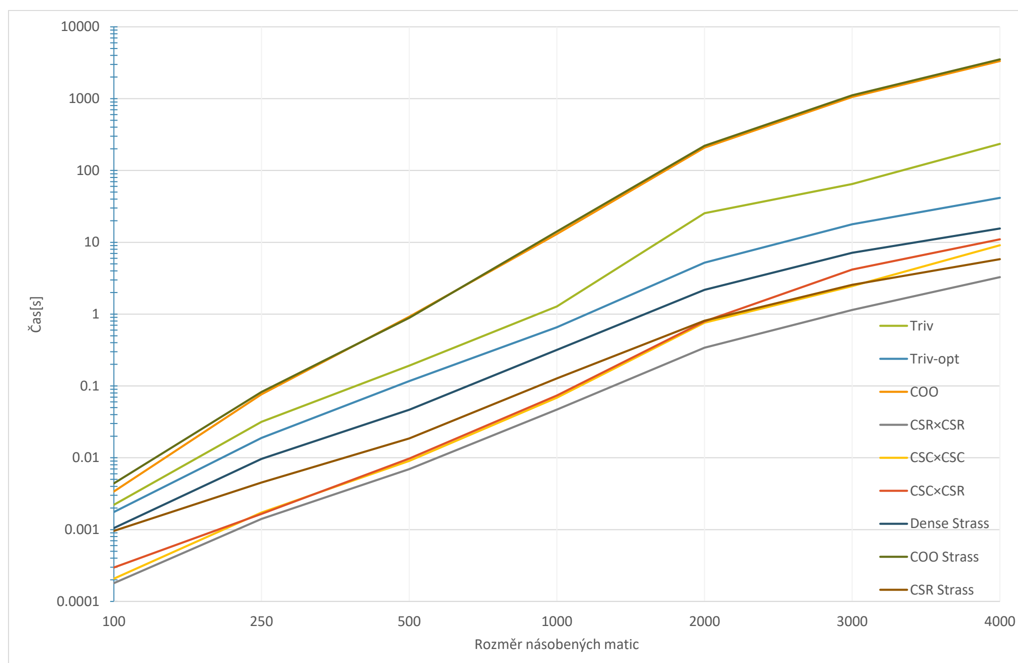
Obrázek 6.4: Doba výpočtu při změně hustoty matice o rozměrech 1000×1000, osa y v logaritmickém měřitku

6.3 Doba výpočtu v závislosti na velikosti matice

Grafy v 6.5 a 6.6 ukazují dobu výpočtu při rostoucí velikosti při 10%, resp. 20% hustotě vstupních matic. Je zde znovu vidět, že formát COO pro násobení matic není vhodný. Při větším počtu nenulových prvků jsou velice neefektivní a časy se dostávají až do tisíců vteřin. Násobení v komprimovaných formátech se po celou dobu drží pod násobeními v hustých formátech a je vhodné je pro takovéto matice využít. Nejlepších časů opět dosahuje násobení CSR×CSR.



Obrázek 6.5: Doba výpočtu při změně velikosti matice (hustota matice 10%), osa y v logaritmickém měřitku



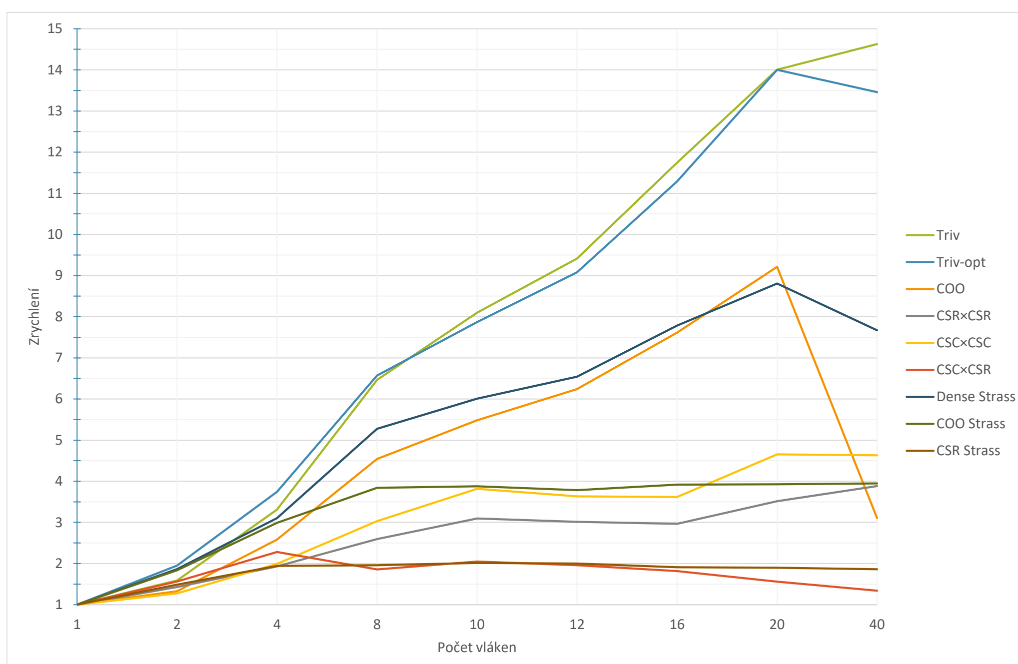
Obrázek 6.6: Doba výpočtu při změně velikosti matice (hustota matice 20%), osa y v logaritmickém měřitku

6.4 Zrychlení algoritmů při použití více vláken

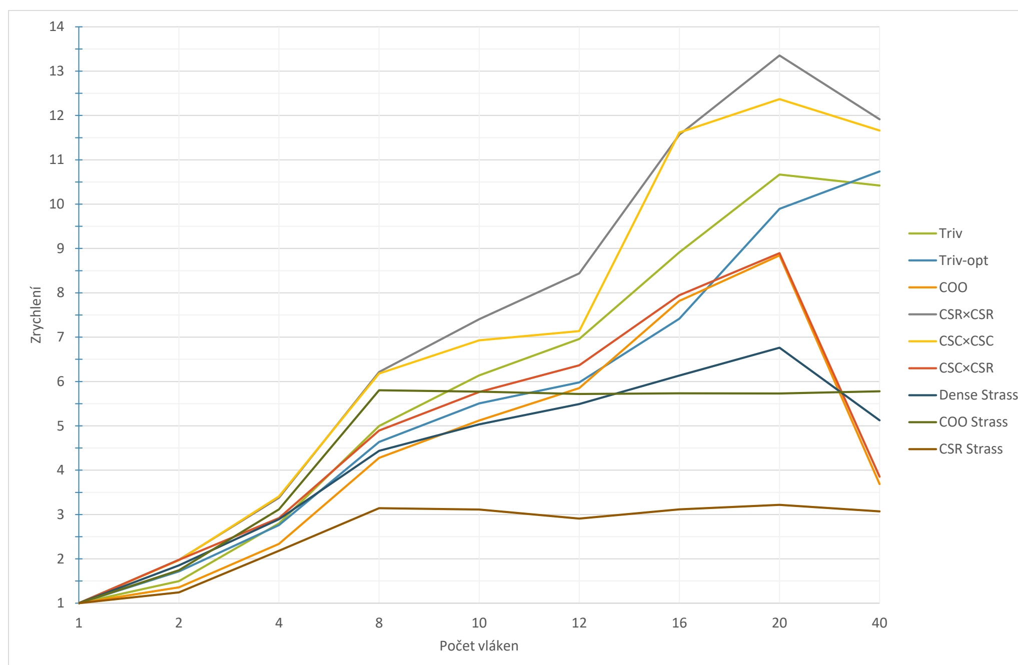
V grafech 6.7 a 6.8 můžeme vidět škálování algoritmů při využití více vláken. Podle grafů dosahují největšího zrychlení triviální algoritmy pro násobení v hustém formátu. Algoritmy v komprimovaných formátech v prvním grafu moc dobře neškálují. To bude způsobeno tím, že testovaná matice obsahuje malý počet nenulových prvků a režie + synchronizace spojená s použitím více vláken zabírají značnou část času. Nízký počet nenulových prvků způsobuje i nerovnoměrnou zátěž jednotlivých vláken. V druhém grafu lze vidět škálování při větším počtu nenulových prvků, algoritmy pro řídké formáty zde škálují o něco lépe. Strassenův algoritmus pro řídké formáty při více než 8 vláknech už neškáluje, protože je v něm paralelizován jen výpočet sedmi pomocných matic a ostatní vlákna zůstávají nevyužita.

Za zmínku stojí propad výkonnosti u některých algoritmů při využití 40 vláken. Domnívám se, že je to způsobeno tím, že testovací hardware obsa-

huje pouze 20 fyzických jader. Pro využití 40 vláken je zapnuta technologie Hyper-Threading, kdy každé dvě vlákna sdílejí prostředy jednoho fyzického jádra. Vlákna si nejspíše přepisují svoji sdílenou cache a to způsobuje propad výkonnosti.



Obrázek 6.7: Zrychlení při využití více vláken, matice PSMIGR 26.1



Obrázek 6.8: Zrychlení při využití více vláken, matice 1000 × 1000 se 100% hustotou

6.5 Porovnání s knihovnou Eigen

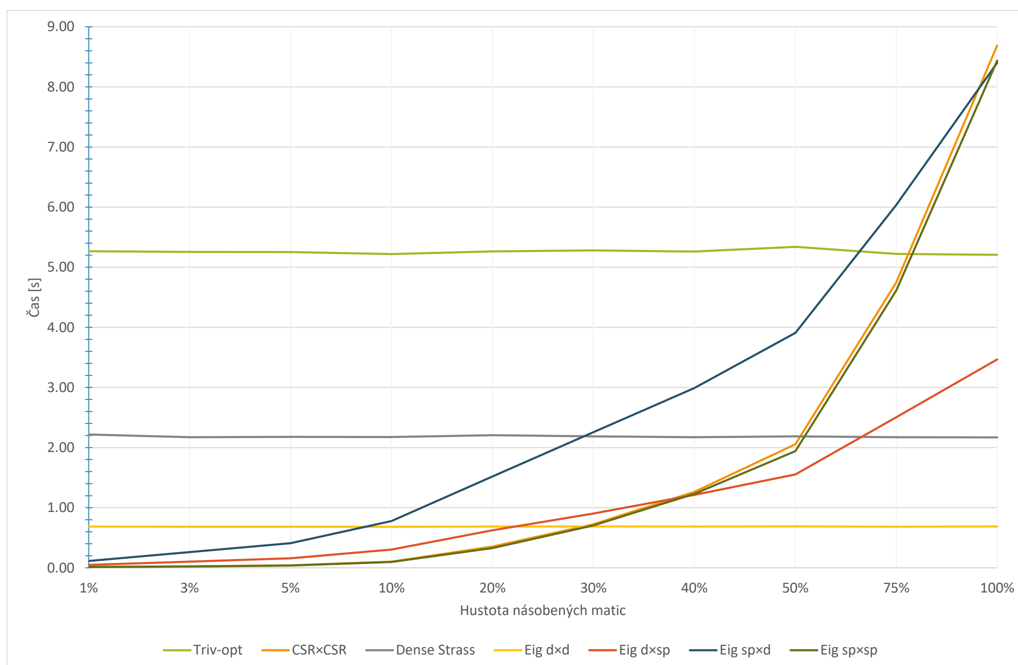
Výkonnost algoritmů jsem porovnal s veřejně dostupnou knihovnou Eigen4.1.2. Bylo provedeno pouze měření při využití jednoho vlákna, protože Eigen nepodporuje paralelní zpracování při práci s řídkými maticemi[9] (kromě násobení sparse×dense v řádkovém uspořádání, ale sekvenční algoritmy byly pomalejší než při využití defaultního formátu). Otestoval jsem algoritmy násobící husté i řídké formáty. Jak je vidět v 7, použití knihovny je přímočaré. Pro dosažení co nejlepšího výkonu při měření byl vypnut debugovací mód definováním direktivy `EIGEN_NO_DEBUG`.

Výsledky je možné si prohlédnou v grafu 6.9. Násobení hustých matic matic je v Eigenu mnohonásobně rychlejší než algoritmy implemetované v této práci. Násobení v řídkém formátu jsou přibližně stejně výkonné, ale při vyšší hustotě matic Eigen začíná vyhrávat.

[9] https://eigen.tuxfamily.org/dox/TopicMultiThreading.html

Ukázka kódu 7 Násobení matic s knihovnou Eigen

```
#include <Sparse>
#define EIGEN_NO_DEBUG
int main() {
  /*...*/
  Eigen::SparseMatrix<double> sp1(m1, n1), sp2(m2, n2); //upraveny format csc
  Eigen::MatrixXd d1(m1,n1), d2(m1,n1);
  /*Inicializace matic*/
  Eigen::MatrixXd d_d(d1 * d2); // dense*dense
  Eigen::MatrixXd d_sp(d1 * sp2); // dense*sparse
  Eigen::MatrixXd sp_d(sp1 * d2); // sparse*dense
  Eigen::MatrixXd sp_sp(sp1 * sp2); // sparse*sparse
  /*...*/
  return 0;
}
```



Obrázek 6.9: Porovnání vybraných algoritmů s knihovnou Eigen pro různé hustoty matic o velikosti 2000 × 2000

Závěr

V této práci jsem nejdříve popsal formáty pro uložení řídkých matic COO, CSR a CSC. Poté byly představeny některé základní i pokročilejší algoritmy pro násobení matic.

Tyto formáty poté byly implementovány. Z představených algoritmů jsem implementoval triviální algoritmus a Strassenův algoritmus pro hustý formát, formát COO a formát CSR. Algoritmy jsem optimalizoval pomocí transformací kódů a paralelizoval pomocí knihovny OpenMP.

Implementované algoritmy byly otestovány na vygenerovaných maticích i na maticích z veřejně dostupných zdrojů. Výkonnost sekvenčních i paralelních algoritmů byla změřena na fakultním svazku STAR a porovnána mezi sebou. Nejlepších výsledků pro řídké matice dosahovalo násobení ve formátu CSR×CSR. Nakonec byly implementované algoritmy porovnány s veřejně dostupnou knihovnou Eigen.

Literatura

[1] Eigen. Dostupné z: https://eigen.tuxfamily.org/dox-devel/group__TutorialSparse.html

[2] Dombek, D.; Klouda, K.: Studijní text k předmětu BI-LIN. 2017. Dostupné z: https://edux.fit.cvut.cz/courses/BI-LIN/_media/buildbot/lin-text.pdf

[3] Olšák, P.: *Úvod do algebry, zejména lineární*. Praha: FEL ČVUT v Praze, vyd. 1. vydání, 2007, ISBN 978-80-01-03775-1.

[4] Horděǰčuk, V.: Asymptotická složitost. Dostupné z: http://voho.eu/wiki/asymptoticka-slozitost/

[5] Dombek, D.; Kolář, J.: Materiály k předmětu BI-ZDM. ZS 2015/16. Dostupné z: https://edux.fit.cvut.cz/courses/BI-ZDM

[6] Lichý, L.: Efektivní násobení řídkých matic. 2014. Dostupné z: https://dip.felk.cvut.cz/browse/pdfcache/lichyluk_2014bach.pdf

[7] Dongarra, J.: Survey of Sparse Matrix Storage Formats. 1995. Dostupné z: http://www.netlib.org/linalg/html_templates/node90.html

[8] Nesrovnal, T.: Vliv formátu uložení řídké matice na výkonnost násobení řídkých matic. 2014. Dostupné z: https://dip.felk.cvut.cz/browse/pdfcache/nesrotom_2014bach.pdf

[9] SciPy. c2008-2016. Dostupné z: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html#scipy.sparse.coo_matrix

[10] Zvoník: Rychlé násobení matic a Strassenův algoritmus ve Winogradově úpravě. 2010. Dostupné z: http://people.fjfi.cvut.cz/pelanedi/tema/SkriptaTEMAKonecSemestru/StrassenZvonik.pdf

[11] Nguyen, P. T.: Divide-and-Conquer algorithsm for matrix multiplication. 2009. Dostupné z: http://cs.mcgill.ca/~pnguyen/251F09/matrix-mult.pdf

[12] Coppersmith, D.; Winograd, S.: Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, ročník 9, č. 3, 1990: s. 251–280, ISSN 07477171, doi:10.1016/S0747-7171(08)80013-2. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/S0747717108800132

[13] Davie, A. M.; Stothers, A. J.: Improved bound for complexity of matrix multiplication. *Proceedings of the Royal Society of Edinburgh: Section A Mathematics*, ročník 143, č. 02, 2013: s. 351–369, ISSN 0308-2105, doi:10.1017/S0308210511001648. Dostupné z: http://www.journals.cambridge.org/abstract_S0308210511001648

[14] Williams, V. V.: Breaking the Coppersmith-Winograd barrier. 2011. Dostupné z: https://people.csail.mit.edu/virgi/matrixmult-f.pdf

[15] Gall, F. L.: Powers of tensors and fast matrix multiplication. 2014. Dostupné z: https://arxiv.org/abs/1401.7714

[16] Cohn, H.; Umans, C.: A group-theoretic approach to fast matrix multiplication. *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings*, 2003: s. 438–449, doi:10.1109/SFCS.2003.1238217. Dostupné z: http://ieeexplore.ieee.org/document/1238217/

[17] Eigen. Dostupné z: http://eigen.tuxfamily.org/index.php?title=Main_Page

[18] Mozilla Public License. 2012. Dostupné z: https://www.mozilla.org/en-US/MPL/2.0/

[19] Armadillo. Dostupné z: http://arma.sourceforge.net/

[20] Apache License. 2004. Dostupné z: http://www.apache.org/licenses/LICENSE-2.0.html

[21] OpenBLAS. Dostupné z: http://www.openblas.net/

[22] LAPACK. 1992. Dostupné z: http://www.netlib.org/lapack/

[23] Hong, O.: Performance considerations with sparse matrices in Armadillo. Dostupné z: http://gallery.rcpp.org/articles/armadillo-sparse-matrix-performance/

[24] Blaze. Dostupné z: https://bitbucket.org/blaze-lib/blaze/

[25] UBLAS. Dostupné z: http://www.boost.org/doc/libs/1_66_0/libs/numeric/ublas/doc/index.html

[26] SciPy. c2008-2016. Dostupné z: https://scipy.org/

[27] SciPy. c2008-2016. Dostupné z: https://docs.scipy.org/doc/scipy/reference/sparse.html

[28] SciPy. c2008-2016. Dostupné z: https://github.com/scipy/scipy/tree/master/scipy/sparse/sparsetools

[29] GCC. 2018. Dostupné z: https://www.gnu.org/software/gcc/

[30] Valgrind. c2000-2018. Dostupné z: http://www.valgrind.org/

[31] Šimeček, I.: BI-EIA: Použití vektorizace v C/C++ (GCC a ICC). 2016. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/vektorizace.pdf

[32] Šimeček, I.: Kompilátorové optimalizace I. 2016. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/kompilator.pdf

[33] OpenMP Specifications. c2012-2018. Dostupné z: https://www.openmp.org/specifications/

[34] Members - OpenMP. 2012-2018. Dostupné z: https://www.openmp.org/about/members/

[35] Barney, B.: OpenMP Tutorial. Dostupné z: https://computing.llnl.gov/tutorials/openMP/

[36] Helsgaun, K.: Shared Memory Parallel Computing. 2010. Dostupné z: http://akira.ruc.dk/~keld/teaching/IPDC_f10/Slides/

[37] Matrix Market. Dostupné z: https://math.nist.gov/MatrixMarket/

[38] PSMIGR 2. 1983. Dostupné z: https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/psmigr/psmigr_2.html

[39] WEST2021. 1983. Dostupné z: https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/chemwest/west2021.html

[40] Cícha, T.: Řídká matice a jejich použití v numerické matematice. 2009. Dostupné z: https://is.muni.cz/th/207863/prif_b/sparse_matrices.pdf

Příloha **A**

Seznam použitých zkratek

nnz Number of nonzero matrix elements

COO Coordinate List

CSR Compressed Sparse Row

CSC Compressed Sparse Column

MPL Mozilla Public License

SIMD Single Instruction Multiple Data

OpenMP Open Multi-Processing

Příloha B

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
impl_src .......................... zdrojové kódy implementace
  matrix .......................... testovací data
text
  text_src ........................ zdrojový kód textu ve formátu LaTeX
  BP_Mai_Thanh_Quang_2019.pdf ..... text práce ve formátu PDF
```