Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Teoretické informatiky

Bakalářská práce

Vliv formátu uložení řídké matice na výkonnost násobení řídkých matic

Tomáš Nesrovnal

Vedoucí práce: Ing. Ivan Šimeček, Ph.D

1. května 2014

Poděkování

Děkuji vedoucímu práce Ing. Ivanu Šimečkovi, Ph.D. a bc. Štefanu Šafárovi za cenné rady.

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. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či spracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 1. května 2014 ....

Odkaz na tuto práci

Nesrovnal, Tomáš. *Vliv formátu uložení řídké matice na výkonnost násobení řídkých matic*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2014.

Abstrakt

V několika větách shrňte obsah a přínos této práce v češtině. Po přečtení abstraktu by se čtenář měl mít čtenář dost informací pro rozhodnutí, zda chce Vaši práci číst.

Klíčová slova Nahraďte seznamem klíčových slov v češtině oddělených čárkou.

Abstract

Sem doplňte ekvivalent abstraktu Vaší práce v angličtině.

Keywords Nahraďte seznamem klíčových slov v angličtině oddělených čárkou.

Obsah

Úvod **1**

1 Úvod do problematiky **3**
1.1 Použití násobení matic 3
1.2 Matice 5
1.3 Vektor 5
1.4 Definice násobení matice maticí 5
1.5 Složitosti 6
1.6 Řídké matice 6

2 Algoritmy násobení matic **7**
2.1 Podle definice 7
2.2 Násobení transponovanou maticí 8
2.3 Násobení po řádcích 8
2.4 Rekurzivní násobení 9
2.5 Strassenův algoritmus 10
2.6 Rychlé algoritmy 14
2.7 Algoritmus podle definice upravený pro řídké matice 14
2.8 Rychlé násobení řídkých matic 18
2.9 Další algoritmy pro řídké matice 18

3 Formáty uložení řídkých matic **19**
3.1 COO - Coordinate list 20
3.2 CSR - Compressed sparse row 21
3.3 BSR - Block Sparse Row 23
3.4 Quadtree 26
3.5 ? 27

4 Modifikace formátu quadtree **29**
4.1 KAT - k-ary tree matrix 29

4.2 Typy listů 29
4.3 Tvoření stromu 30
4.4 Násobení 30

5 Analýza a návrh **33**
5.1 ATLAS 33
5.2 Možnosti řešení 34
5.3 Zvolené řešení 34

6 Realizace **35**
6.1 Prostředí 35
6.2 Design implementace 35
6.3 Implementace KAT 36
6.4 Testování 38
6.5 MatrixMarket 38
6.6 Optimalizace 40
6.7 Měření 40

Závěr **41**

Literatura **43**

A Seznam použitých zkratek **47**

Seznam obrázků **47**

B Obsah přiloženého CD **51**

Seznam obrázků

1.1 3D neorientovaný graf ve tvaru helikoptéry, jeho reprezentace v řídké matici a výsledek simulace 4

2.1 Strassen (TODO: převzato z wikipedie: předělat?) 11
2.2 Ukázka numerické stability Strassenova algoritmu 14
2.3 Matice orsirr_1 před vynásobením 16
2.4 Matice orsirr_1 po vynásobením sama se sebou 17

3.1 Matice uložená ve formátu CSR 22
3.2 Matice uložená ve formátu BSR 24
3.3 Rozdělení matice 26
3.4 Strom matice uložené ve formátu Quadtree 27

6.1 UML diagram tříd programu 36
6.2 Matice vygenerovaná generátorem 40

Seznam tabulek

3.1 Matice uložená ve formátu COO 20

Úvod

Kapitola 1

Úvod do problematiky

Při řešení problémů často hledáme způsob, jak interpretovat data v takovém formátu, se kterým již umíme pracovat. Jedním ze základních prvků je matice. Ve spoustě případů takové matice obsahují nemalý počet nulových prvků. Takové matice obecně navýváme řídké a při práci s nimi tuto vlastnost využíváme.

Pro výstavu v roce 2011 s názvem Art in Enginnering v Harnově Muzeu představila floridská univerzita kolekci ilustrací řídkých matic s názvem The Beauty of Mathematics: As Illustrated by the University of Florida Sparse Matrix Collection [11]. Pro estetické zobrazení řídkých matic byla provedena simulace. Každému uzlu byl přiřazen elektrický náboj a každá hrana představovala pružinu. V simulaci byla celá tato kontrukce položena na tvrdou podložku. Simulace byla zastavena v okamžiku, když se konstrukce přestala hýbat. Pro ilustraci přikládáme výsledek simlulace 1.1 na modelu helikoptéry, tedy neorientovaném 3D grafu uloženém v řídké matici.

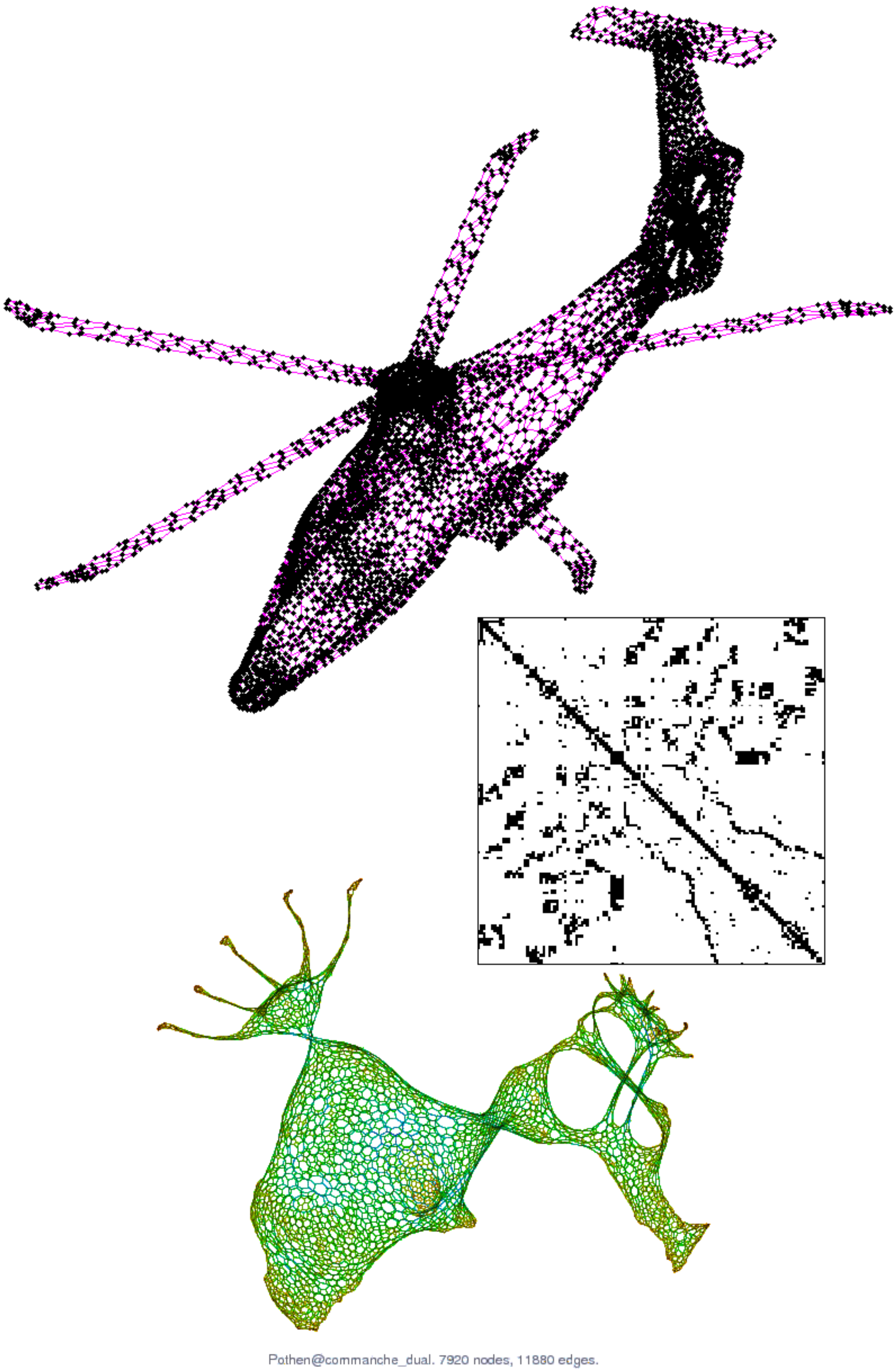
Řídké matice jsou používané ve velké škále oblastí, například výpočty diferenciálních rovnic, Google page rank, 3D grafika, statistika, těžení z dat, kryptografie a další.

1.1 Použití násobení matic

Násobení maticí může znázorňovat nějakou transformaci jednoho stavu do druhého. Největší uplatnění řídkých matic je tedy v simulaci jevů, například fyzikálních, biologických, chemických, ekonomických a dalších.

Jako příklad simulace můžeme uvést metodu konečných prvků [12][6].

Násobení matice maticí má menší praktické využití než násobení matice vektorem. V praxi ale můžeme složit více vektorů v jednu matici a tím si zrychlit výpočet. To se obzvláště hodí pro real-time aplikace.



Obrázek 1.1: 3D neorientovaný graf ve tvaru helikoptéry, jeho reprezentace v řídké matici a výsledek simulace

1.2 Matice

Matice **A** typu (m, n) je mn uspořádaných prkvů z množiny **R**. O prvku $a_{r,s} \in \mathbf{R}, r \in \{1, 2, \ldots, m\}, s \in \{1, 2, \ldots, n\}$ říkáme, že je na r-tém řádku a s-tém sloupci matice **A**. Matici **A** zapisujeme do řádků a sloupců takto:

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \tag{1.1}$$

Matici **M** typu (m, n), kde všechny její prvky jsou rovny nule, nazýváme *nulovou maticí.*

O matici typu (m, n) budeme říkat, že je m široká a n vysoká. Pokud o matici řekneme že má velikost n, myslíme tím, že je typu (n, n).

1.3 Vektor

Matici **V** typu (1, n) nazveme vektorem.

1.4 Definice násobení matice maticí

Buď **A** matice typu (m,n) s prvky $a_{i,j}$ a **B** matice typu (n,p) s prvky $b_{j,k}$. Definujeme součin matic $\mathbf{A} \cdot \mathbf{B}$ jako matici **C** typu (m,p) s prvky $c_{i,k}$ které vypočteme jako:

$$c_{row,col} = \sum_{k=1}^{N} a_{row,k} b_{k,col} \tag{1.2}$$

Výsledek součinu matic se nezmění, pokud matice doplníme o libovolný počet nulových řádků a nebo sloupců. Této vlastnosti můžeme využít pro získání potřebných rozměrů:

1. Při násobení matice A typu (m,n) s maticí B typu (o,p), kde $n \neq o$.
2. Pokud potřebujeme matice stejné velikosti.
3. Pokud potřebujeme matice určité velikosti, například $2^{\mathbf{N}}$.

Násobení matice vektorem je pouze případem násobení matice typu (m,n) s maticí typu (1,n).

1.5 Složitosti

K označení složitostí, ať již časových nebo prostorových (paměťových) používáme $\mathcal{O}$-notaci, která označuje množinu funkcí, asymptoticky roustoucí řádově stejně rychle, nebo rychleji.

$$\mathcal{O}(g(n)) = \Big\{f(n) : \exists c \in R^{+} \exists n_0 \in N \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)\Big\} \tag{1.3}$$

$\mathcal{O}$notací budeme popisovat nejhorší možný případ. Obdobně Θ značí funkce roustoucí stejně rychle a Ω stejně rychle, nebo pomaleji.

Pro výpočet složitostí rekurzivních algoritmu používáme Mistrovksou metodu [10]. Pokud $a \geq 1, b > 0$ jsou konstanty a $f(n)$ je funkce o jednné proměnné, tak rekurence

$$t(n) = at(n/b) + f(n) \tag{1.4}$$

má asyptototickou složitost:

1. Pro $f(n) = \mathcal{O}(n^{log_b a - \epsilon})$, kde $\epsilon > 0$ je konstanta platí, že $t(n) = \Theta(n^{log_b a})$.
2. Pro $f(n) = \Theta(n^{log_b a})$ platí, že $t(n) = \Theta(n^{log_b a} log n)$.
3. Pro $f(n) = \Omega(n^{log_b a + \epsilon})$, kde $\epsilon > 0$ je konstanta a $af(n/b) \leq cf(n)$ pro konstantu $c < 1$ a $\forall n \geq n_0$, tak platí, že $t(n) = \Theta(f(n))$.

1.6 Řídké matice

Matice, které obsahují velké množství nulových prvků, nazýváme řídké. Nebudeme přesně uvádět, kolik procent z celkového počtu prvků musí být nulových, abychom matici nazývali řídkou. Stejně jako řídkou matici můžeme uložit do formátu pro husté matice, můžeme hustou matici uložit do formátu pro řídké matice.

Řídkost matice budeme vyjadřovat pomocí nnz (Number of NonZero elements), tedy počtem nenulových prvků z celkových mn, pro matici A typu (m, n).

TODO: typy ridkych matic (pasova, atd, pattern, real)

Kapitola 2

Algoritmy násobení matic

V této kapitole představíme některé základní a pokročilejší algoritmy pro násobení matic. Základní algoritmy mají stejnou asymptotickou složitost $O(n^3)$ a liší se pouze přístupům k prvkům, což je důležité pro řídké formáty. Pokročilejší algoritmy jsou sice asymptoticky rychlejší, ale přinášejí nevýhodu ve formě numerické stability a velké skryté konstantě.

2.0.1 Pseudokódy

Pseudokódy v této práci jsou ve stylu syntaxe jazyka Fortran, ale budou představovat zápis jazyka C99. Pro pole platí, že jsou indexovaná od nuly a for cyklus `for i ← 0 to 10` bude iterovat od nultého do devátého prvku včetně.

2.1 Podle definice

Základním algoritmem násobení dvou matic je podle definice. Ve třech for cyklech postupně vybíráme řádky matice A, sloupce matice B a v N krocích násobíme. N je jak šířka matice A, tak i výška matice B.

Algorithm 1 Násobení matic podle definice

```
procedure MMM-DEFINITION(A, B, C)                 ▷ A,B,C jsou matice
    for row ← 0 to A.height do                    ▷ řádky
        for col ← 0 to B.width do                 ▷ sloupce
            sum ← 0;
            for i ← 0 to A.height do
                sum += A[row][i] * B[i][col];
            end for
            C[row][col] ← sum;
        end for
    end for
end procedure
```

Z pseudokódu je vidět, že ve dvou for cyklech provádíme N násobení a N sčítaní. Asymptotická složitost je tedy $O(n^2(n+n)) = O(2n^3)$. V ukázkových výpočtech je násobení pouze $N-1$ krát, to proto, že neuvádíme přičítání k nule (řádek 6).

2.2 Násobení transponovanou maticí

Pokud nám formát uložení matice nedovolí procházet prvky po sloupcích, je řešením druhou matici transponovat. Poté můžeme násobit řádky matice A s řádky transponované matice B.

Algorithm 2 Násobení transponovanou maticí

```
procedure MMM-TRANSPOSE(A, B, C)                ▷ A,B,C jsou matice
    B ← transpose(B)
    for rowA ← 0 to A.height do                 ▷ řádky
        for rowB ← 0 to B.height do             ▷ sloupce
            sum ← 0;
            for i ← 0 to A.height do
                sum += A[rowA][i] ∗ B[i][rowB];
            end for
            C[rowA][rowB] ← sum;
        end for
    end for
end procedure
```

Podobný algoritmus můžeme použít i pokud nám formát nedovolí procházet prvky po řádcích, ale pouze po sloupcích. Například v této práci neuvedený Compressed Sparse Columns.

2.3 Násobení po řádcích

Další možností jak násobit dvě matice, kde nám formát uložení nedovolí procházet po sloupcích je procházet současně řádky matice A i B a přičítat jednotlivé součiny na správné místo ve výsledné matici C.

Nevýhodou tohoto řešení je velký počet přístupů do pole C. Protože k prvkům přičítáme, tedy načítáme a sčítáme, je potřeba před samotným násobením nastavit všechny prvky matice C na hodnotu nula.

Algorithm 3 Násobení po řádcích

```
procedure MMM-BY-ROWS(A, B, C)                    ▷ A,B,C jsou matice
    for r ← 0 to A.height do                      ▷ řádky matice A i B
        for cA ← 0 to A.width do                  ▷ sloupce matice A
            for c ← 0 to B.width do               ▷ sloupce matice B
                C[r][cA] += A[r][cA] * B[r][cB];
            end for
        end for
    end for
end procedure
```

2.4 Rekurzivní násobení

Pro matice A i B o stejné velikosti $2^{\mathbf{N}}$ můžeme použít rekurzivní přístup. Tedy programovací techniku rozděl a panuj, kdy rozdělíme větší problémy na menší podproblémy.

Každou z matic rozdělíme na čtvrtiny a jednotlivé podmatice násobíme algoritmem podle definice, tedy jako matice o velikosti dva.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix} \tag{2.1}$$

Tento postup opakujeme, dokud velikostí podmatic nenarazíme na práh, tedy hodnotu, při které opustíme rekurzivní algoritmus a použijeme algoritmus lineární. V ukázkovém pseudokódu dělíme podmatice až na velikost prahu jedna, podmatice tedy obsahují pouze jeden prvek.

Algorithm 4 Rekurzivní násobení

```
procedure MMM-RECURSIVE(A, B, C, ay, ax, by, bx, cy, cx, n)
    if n = 1 then
        C[cy][cx] ← C[cy][cx] + A[ay][ax] · B[by][bx];
        return;
    end if
    for all r ∈ {0, n/2} do
        for all c ∈ {0, n/2} do
            for all i ∈ {0, n/2} do
                MMM-recursive(A, B, C, ay + i, ax + r, by + c, bx + i, cy + c, cx + r, n/2);
            end for
        end for
    end for
end procedure
```

Pro ilustraci jako příklad uvádíme výpočet horního levého prvku v násobení dvou matic o velikosti 2^2. Pro větší přehlednost značíme prvky malým písmem z názvu matice a indexy o jejich pozicích.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{pmatrix} = \tag{2.2}$$

$$\begin{pmatrix} \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} + \begin{pmatrix} a_{1,3} & a_{1,4} \\ a_{2,3} & a_{2,4} \end{pmatrix} \cdot \begin{pmatrix} b_{3,1} & b_{3,2} \\ b_{4,1} & b_{4,2} \end{pmatrix} & \dots \\ \dots & \dots \end{pmatrix} = \tag{2.3}$$

$$\begin{pmatrix} \begin{pmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & \dots \\ \dots & \dots \end{pmatrix} + \begin{pmatrix} a_{1,3}b_{3,1} + a_{1,4}b_{4,1} & \dots \\ \dots & \dots \end{pmatrix} & \dots \\ \dots & \dots \end{pmatrix} = \tag{2.4}$$

$$\begin{pmatrix} \begin{pmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1} + a_{1,4}b_{4,1} & \dots \\ \dots & \dots \end{pmatrix} & \dots \\ \dots & \dots \end{pmatrix} \tag{2.5}$$

Asymptotická složitost je samozřejmě stejná jako u algoritmu podle definice. Asymptotickou složitost rekurzivního algoritmu můžeme spočítat pomocí mistrovské metody.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T(n/2) + \Theta(1) & \text{if } n > 1 \end{cases}$$

Protože platí, že $a = 8, b = 2, r = \log_2 8, n^r = n^{\log_2 8} = n^3 = \Omega(1)$, tak asymptotická složitost podle mistrovké metody je `MMM-recursive`(n) $= O(n^3)$.

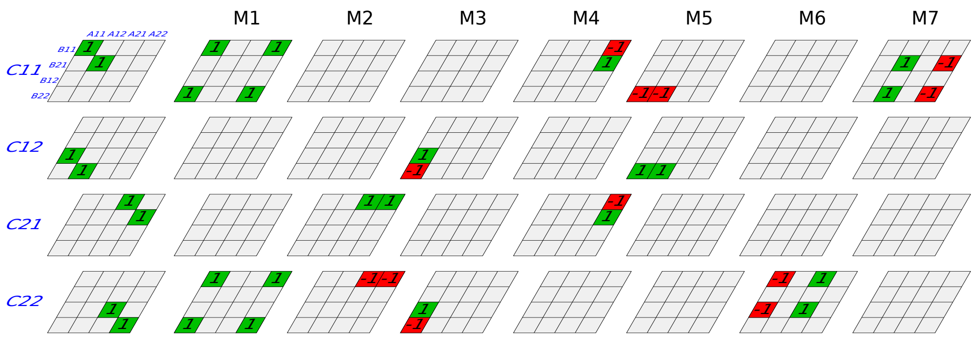
Kvůli režii rekurzivního dělení v praxi nezmenšujeme podmatice až na velikost jedna. Vhodný práh velikosti podmatice je například takový, co se vejde do L1 cache.

2.5 Strassenův algoritmus

V roce 1969 Volker Strassen v časopise Numerische Mathematik publikoval článek [19], ve kterém jako první představil algoritmus násobení dvou matic s menší asymptotickou složitostí než algotimus podle definice, tedy $O(n^3)$.

Algoritmus je založen na myšlence, že sčítání je operace méně náročnejší než operace násobení. Respektive dvě matice umíme sečíst nebo odečíst v složitosti $O(n^2)$, ale vynásobit v $O(n^3)$.

Volker Strassen tedy využil jisté symetrie [14] v násobení dvou matic A a B o velikosti dva a výslednou matici C seskládal pomocí sedmi pomocných matic. Obrázek 2.1 ukazuje, z čeho se pomocné matice skládají a jak jsou do výsledné matice seskládany. V ilustračních maticích o velikosti čtyři ukazujeme, které sčítance pomocná matice do výsledku přičítá a které odečítá.



Obrázek 2.1: Strassen (TODO: převzato z wikipedie: předělat?)

Zápis Strassenova algoritmu vypadá následnovně:

$$A \cdot B = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \tag{2.6}$$

$$M_1 = (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}) \tag{2.7}$$

$$M_2 = (A_{2,1} + A_{2,2}) \cdot B_{1,1} \tag{2.8}$$

$$M_3 = A_{1,1} \cdot (B_{1,2} - B_{2,2}) \tag{2.9}$$

$$M_4 = A_{2,2} \cdot (B_{2,1} - B_{1,1}) \tag{2.10}$$

$$M_5 = (A_{1,1} + A_{1,2}) \cdot B_{2,2} \tag{2.11}$$

$$M_6 = (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2}) \tag{2.12}$$

$$M_7 = (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}) \tag{2.13}$$

$$C = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix} \tag{2.14}$$

V pseudokódu používáme procedury `offset-add` respektive `offset-sub`. Slouží ke sčítání respektive odečítání bloku prvků o velikosti n v maticích od nějakého offsetu `y` a `x`. Paremetry obou funkcí jsou: `offset-*`$(A, B, C, ay, ax, by, bx, cy, cx, n)$.

Algorithm 5 Strassenův algoritmus

```
procedure MMM-STRASSEN(A, B, C, ay, ax, by, bx, cy, cx, n)
    if n = 1 then
        C[cy][cx] ← C[cy][cx] + A[ay][ax] · B[by][bx];
        return;
    end if
    h ← n/2;                                                  ▷ čtvrtina
    m[9] ← init-matrices(9, h);                               ▷ devět pomocných matic
    offset-add(a, a, m[8], ay, ax, ay + h, ax + h, 0, 0, h);  ▷ M1
    offset-add(b, b, m[9], by, bx, by + h, bx + h, 0, 0, h);
    MMM-strassen(m[8], m[9], m[1], 0, 0, 0, 0, 0, 0, h);
    offset-add(a, a, m[8], ay + h, ax, ay + h, ax + h, 0, 0, h);  ▷ M2
    MMM-strassen(m[8], b, m[2], 0, 0, bx, by, 0, 0, h);
    offset-sub(b, b, m[8], by, bx + h, by + h, bx + h, 0, 0, h);  ▷ M3
    MMM-strassen(a, m[8], m[3], ay, ax, 0, 0, 0, 0, h);
    offset-sub(b, b, m[8], by + h, bx, by, bx, 0, 0, h);      ▷ M4
    MMM-strassen(a, m[8], m[4], ay + h, ax + h, 0, 0, 0, 0, h);
    offset-add(a, a, m[8], ay, ax, ay, ax + h, 0, 0, h);      ▷ M5
    MMM-strassen(m[8], b, m[5], 0, 0, by + h, bx + h, 0, 0, h);
    offset-sub(a, a, m[8], ay + h, ax, ay, ax, 0, 0, h);      ▷ M6
    offset-add(b, b, m[9], by, bx, by, bx + h, 0, 0, h);
    MMM-strassen(m[8], m[9], m[6], 0, 0, 0, 0, 0, 0, h);
    offset-sub(a, a, m[8], ay, ax + h, ay + h, ax + h, 0, 0, h);  ▷ M7
    offset-add(b, b, m[9], by + h, bx, by + h, bx + h, 0, 0, h);
    MMM-strassen(m[8], m[9], m[7], 0, 0, 0, 0, 0, 0, h);
    offset-add(m[1], m[4], m[8], 0, 0, 0, 0, 0, 0, h);        ▷ c1,1
    offset-sub(m[8], m[5], m[8], 0, 0, 0, 0, 0, 0, h);
    offset-add(m[8], m[7], c, 0, 0, 0, 0, cy, cx, h);
    offset-add(m[3], m[5], c, 0, 0, 0, 0, cy, cx + h, h);     ▷ c1,2
    offset-add(m[2], m[4], c, 0, 0, 0, 0, cy + h, cx, h);     ▷ c2,1
    offset-sub(m[1], m[2], m[8], 0, 0, 0, 0, 0, 0, h);        ▷ c2,2
    offset-add(m[8], m[3], m[8], 0, 0, 0, 0, 0, 0, h);
    offset-add(m[8], m[6], c, 0, 0, 0, 0, cy + h, cx + h, h);
end procedure
```

Výpočet asymptotické složitosti vypočteme podobně jako u `MMM-recursive`, tedy mistrovskou metodou.

V každém kroku rekurze počítáme $\Theta(n^2)$ operací na vytvoření pomocných matic.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

Protože platí, že $a = 7, b = 2, r = \log_2 7, n^r = n^{\log_2 7} = \Omega(n^2)$, tak asymptotická složitost podle mistrovské metody je `MMM-strassen`(n) $= O(n^{\log_2 7}) \approx O(n^{2.8})$.

Stejně jako v předešlém algorimu, i zde demonstrujeme výpočet levého horního prvku matice z násobení dvou matic o velikosti dva. Místo parametrické matice použijeme konkrétní desetinná čísla, abychom ukázali numerickou stabilitu Strassenova algoritmu. Pro ukázku budeme uvažovat počítač, který u čísel ukládá pouze pět cifer, znaménko a desetinnou čárku.

Pomocí algoritmu podle definice, by takový počítač vypočítal součin dvou matic následnovně:

$$\begin{pmatrix} 30.234 & 0.5678 \\ 0.9123 & 10.456 \end{pmatrix} \cdot \begin{pmatrix} 0.8912 & 0.3456 \\ 0.7891 & 9.999 \end{pmatrix} = \begin{pmatrix} 27.392 & \dots \\ \dots & \dots \end{pmatrix} \tag{2.15}$$

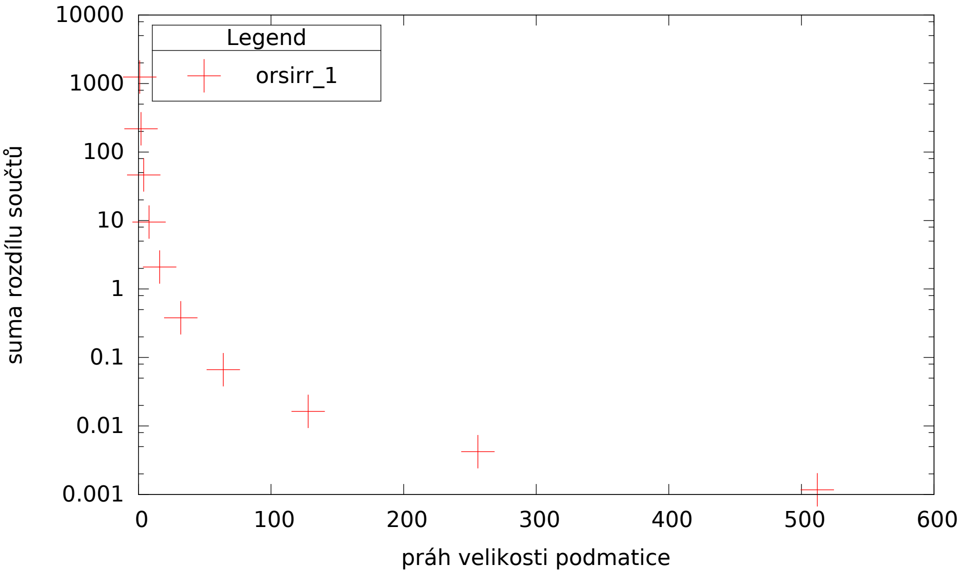
Správný výsledek je $30.234 \times 0.8912 + 0.5678 \times 0.7891 = 26.9445408 + 0.44805098 = 27.39259178$.

Nyní výpočet provedeme pomocí Strassenova algoritmu:

$$\begin{aligned}
\begin{pmatrix} 30.234 & 0.5678 \\ 0.9123 & 10.456 \end{pmatrix} \cdot \begin{pmatrix} 0.8912 & 0.3456 \\ 0.7891 & 9.999 \end{pmatrix} && (2.16) \\
M_1 = (30.234 + 10.456) \cdot (0.8912 + 9.999) = 443.12 && (2.17) \\
\dots && (2.18) \\
M_4 = 10.456 \cdot (0.7891 - 0.8912) = -1.067 && (2.19) \\
M_5 = (30.234 + 0.5678) \cdot 9.999 = 307.98 && (2.20) \\
\dots && (2.21) \\
M_7 = (0.5678 - 10.456) \cdot (0.7891 + 9.999) = -106.67 && (2.22) \\
\begin{pmatrix} M_1 + M_4 - M_5 + M_7 & \dots \\ \dots & \dots \end{pmatrix} = \begin{pmatrix} 27.403 & \dots \\ \dots & \dots \end{pmatrix} && (2.23)
\end{aligned}$$

Jak můžeme vidět, zatímco u algoritmu podle definice jsme pouze ztratili desetinnou přesnost, výsledek Strassenova algoritmu se lišil už v prvním desetinném čísle.

Pro reálnou představu stability Strassenova algoritmu jsme provedli experiment, ve kterém jsme vynásobili dvě stejné matice (matice orsirr_1, oříznuta na velikost 1024) algoritmem podle definice a Strassenovým algoritmem s různými prahy a sečetli všechny rozdíly mezi výsledky. Násobení probíhalo ve dvojté desetinné přesenosti, tedy v datovém typu `double` jazyka C99. Z grafu 2.2 je vidět exponencionální závislost mezi velikostí prahu a celkovou chybou.



Obrázek 2.2: Ukázka numerické stability Strassenova algoritmu

Strassenův algorimus lze ještě vylepšit. Algoritmům na stejném principu se říká Strassen-like [7]. Pro sedm operací násobení je monžné snížit počet sčítání a odečítání. Pro jednoduchost zde ovšem uvádíme originalní algoritmus.

2.6 Rychlé algoritmy

Po tom, co Strassen ukázal, že existují rychlejší algoritmy než $O(n^3)$, ještě rychlejší algoritmy než ten jeho na sebe nenechaly dlouho čekat. Složitost násobení matic pro jednoduchost označíme jako $O(n^\omega)$.

Nejpomalejší algoritmus s $\omega = 3$ je podle definice. Strassenův algoritmus se sedmi násobeními má $\omega \approx 2.807354$. Jeden z nejrychlejších algoritmů je algoritmus Virginie Williamsové [24][23], pro který je $\omega < 2.3727$.

Hranicí nejlepší možné složitosti může být $\omega = 2$, protože každý prvek z matice musíme nějak započítat. Existují z velké části podložené domněnky na základě teorie grup [18], že $\omega = 2$ skutečně platí, ale přímý důkaz ještě neexistuje.

2.7 Algoritmus podle definice upravený pro řídké matice

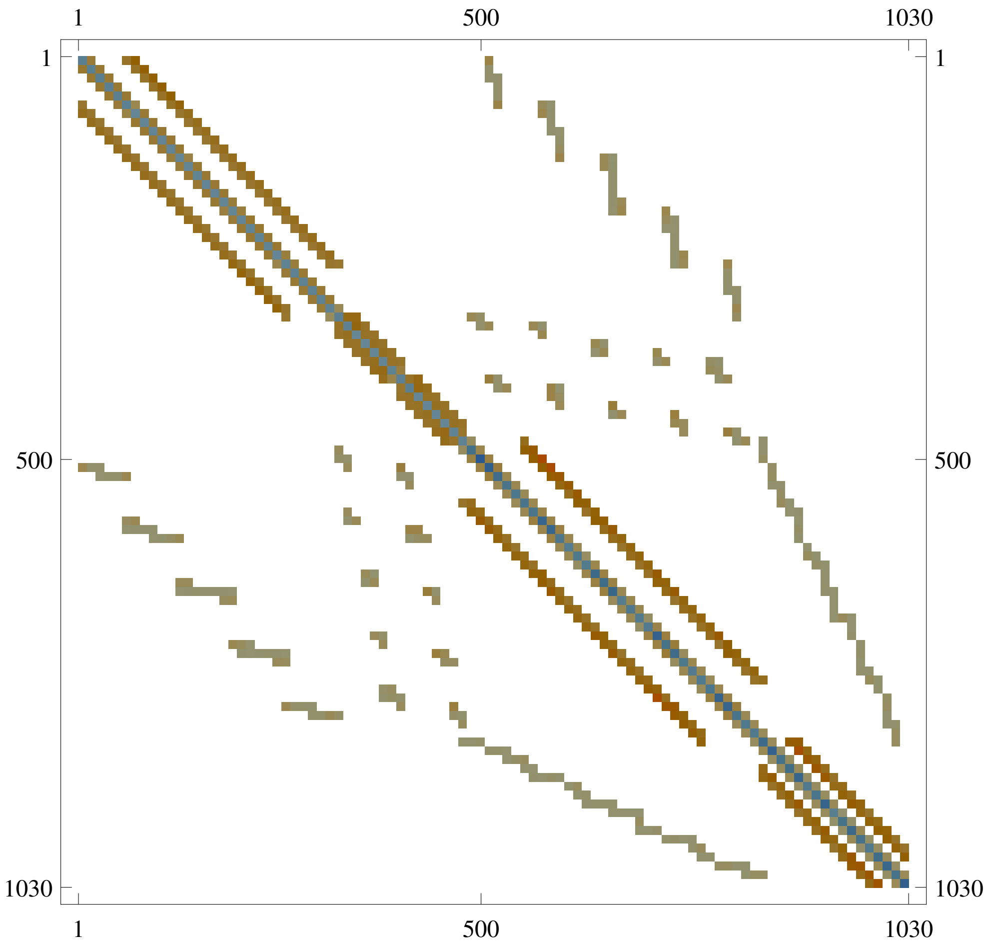
Pokud násobíme řídké matice A a B o velikosti N, můžeme vynechat násobení takových dvou prvků, z nichž je alespoň jeden nulový. Označme $nnzr_{M,i}$

jako počet nenulových prvků v i-tém řádku matice M a $nnzc_{M,i}$ jako počet nenulových prvků v i-tém sloupci matice M. Protože násobíme každý řádek s každým sloupcem, bude celkový počet operací násobení dán vzorcem:

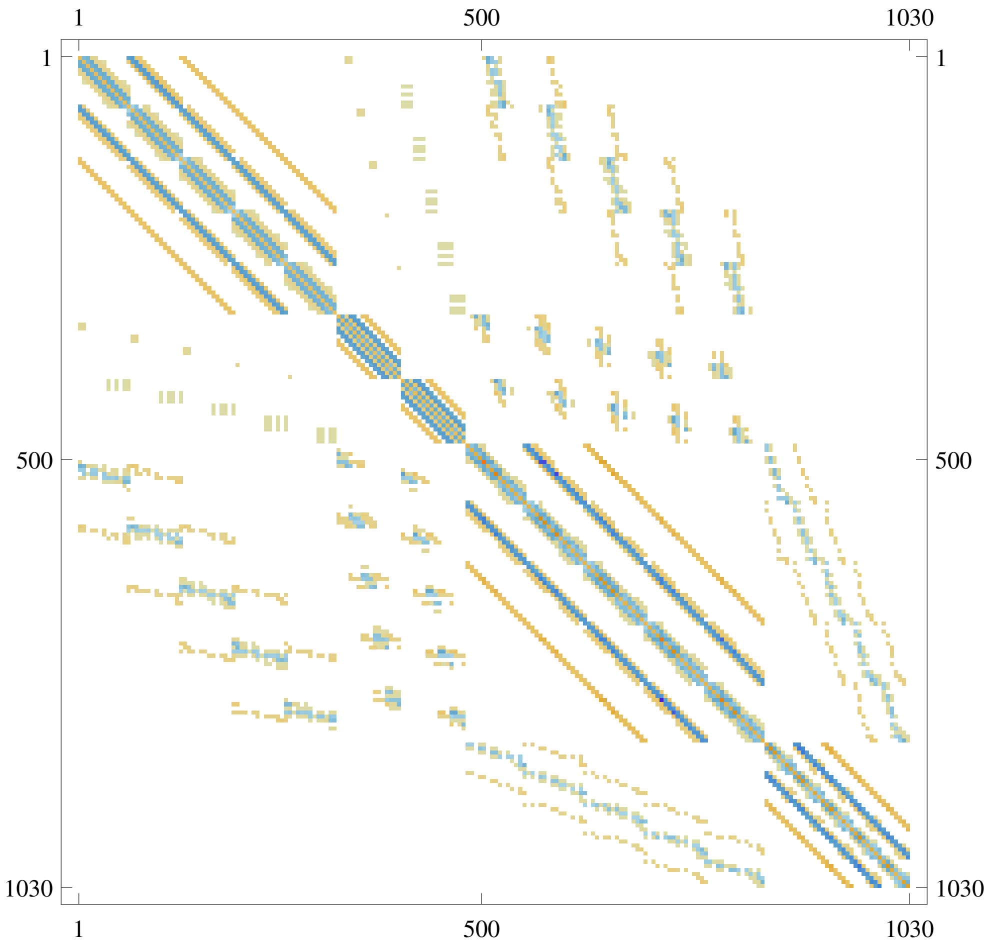
$$\sum_{i=1}^{N} nnzr_{A,i} nnzc_{B,i} \tag{2.24}$$

Složitost tohoto algoritmu pro násobení dvou matic A a B o velikosti n tedy můžeme vyjádřit jako $O(mn)$, kde $m = max(nnz(A), nnz(B)$. Pro husté matice bez jediného nulového prvku samozřejmě platí, že $m = n^2$. Nutno podotknout, že $O(mn)$ je nejhorší případ, kdy se všechny nenulové prvky matice A budou násobit se všemy nenulovými prvky matice B.

Pro reálnou představu demonstrujeme násobení dvou stejných matic. Opět se jedná o dvě matice orsirr_1. Matice orsirr_1 má velikost $n = 1030$ a $m = 6858$ nnz. Rozmístění nnz před respektive po vynásobení ukazují obrázky 2.3 respektive 6.2.



Obrázek 2.3: Matice orsirr_1 před vynásobením



Obrázek 2.4: Matice orsirr_1 po vynásobením sama se sebou

Kdyby nastal nejhorší možný případ, počet operací násobení při použití algoritmu podle definice pro řídké matice by byl $1030 \times 6858 = 7.063740 \times 10^6$. Pro tento případ ovšem stačí 4.6976×10^4 operací násobení. Oproti nejhoršímu možnému případu nastalo $7.063740 \times 10^6 - 4.6976 \times 10^4 = 7.016764 \times 10^6$ situací, kdy jeden ze dvou prkvů byl nulový a operace násobení nemusela být provedena. Při násobení algoritmem podle definice pro husté matice by v 1.092680024×10^9 případech operace násobení nemusela být provedena, protože alespoň jeden ze dvou prvků by byl nula. Po vynásobení matice orsirr_1 sama se sebou, stoupl její počet nnz z 6858 na 23532. V tomto případě to bylo především z důvodu, že všechny prvky z diagonály matice jsou nenulové, každý prvek se tedy započítá.

2.8 Rychlé násobení řídkých matic

Strassenův algoritmus $\omega = 2.8$ od $nnz < n^{1.8}$ a algoritmus Virginie Williamsové $\omega = 2.3$ od $nnz < n^{1.37}$ jsou asymptoticky stejně rychlé jako algoritmus podle definice upravený pro řídké matice $O(mn)$. Například pro $n = 1000$ je tato hranice pro Strassenův algoritmus 251189 (40 %) nnz a pro algoritmus Virginie Williamsové 12882 (78 %) nnz z celkových možných 1000000 prvků.

Raphael Yuster a Uri Zwick ukázali algoritmus [27] s asymptotickou složitostí $O(m^{0.7}n^{1.2} + n^{2+o(1)})$, který rozdělí permutace řádků a sloupců na řídké a husté. Řídké permutace násobí algoritmem podle definice v úpravě pro řídké matice $O(mn)$. Husté permutace vynásobí v té době nejznámějším nejrychlejším algoritmem a to od Dona Coppersmitha a Shmuela Winograda s asymptotickou složitostí $\omega = 2.3$ [9].

2.9 Další algoritmy pro řídké matice

Algoritmů násobení řídkých matic je mnoho. Často se odvíjejí od typu řídkých matic a formátu v jakém jsou uloženy. Příkladem může být formát, ve kterém se ukládají diagonály [22].

Kapitola 3

Formáty uložení řídkých matic

Formáty uložení řídkých matic obecně ukládají jednotlivé elementy zvlášť a tedy nemusí ukládat ty nulové. To ale přináší řadu nevýhod. Za prvé se musí ukládat informace o souřadicních jednotliých prvcích. Za druhé, ztrácíme možnost přístupu k prvku na libovolných šouřadnicích v čase $\Theta(1)$, protože prvky nemáme přímo indexované podle jejich umístění v řádku a sloupci.

Protože řídké matice můžeme rozdělit do mnoha kategorií a provádět nad nimi mnoho operací, existuje hodně formátů, jak řídkou matici efektivně uložit a pracovat s ní.

`http://www.cs.colostate.edu/~mroberts/toolbox/c++/sparseMatrix/sparse_matrix_compression.html`

3.0.1 Modifikace řídké matice

Formáty uložení řídkých matic můžeme také rozdělit podle toho, zda-li je možné do nich přidávat nebo odebírat prvky.

Při násobení matic $C = A \cdot B$ se matice A ani B nemění. V této práci budeme předpokládat, že matice C bude hustá a formáty umožujícími přidávání nebo odebírání prvků nebudou součástí práce. Stejně tak při násobení matice A vektorem B je výsledek C vektor.

3.0.2 Uspořádanost řídké matice

Dalším kritériem pro rozdělení formátů je uspořádanost nenulových prvků v řídké matici. Pro uspořádané prvky bude efektivnější takový formát, který využije určitý vzor. V řídkých maticích takovým vzorem může být například diagonála, nebo blok prvků. Efektivně lze za vzor považovat i prvky v řádku, nebo ve sloupci.

Uspořádáním může být také symetrie matice, kdy nám stačí uložit pouze polovinu matice. Často řídké matice bývají symetrické podle hlavní diagonály.

3.1 COO - Coordinate list

Formát COO, česky seznam souřadnic, je základní formát řídkých matic. Ke každému nenulovému prvku ukládá jeho souřadnice `y` a `x`. Implementovat tento formát můžeme například jako tři pole, jedno s hodnotami prvků, druhé s `y` souřadnicemi a třetí s `x` souřadnicemi.

Pro ukázku v tomto formátu uložíme matici o velikosti $n = 8$ s $nnz = 5$ nenulovými prvky.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{3} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{4} & \mathbf{5} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{3.1}$$

values[5]	1	2	3	4	5
y-coords[5]	1	1	4	7	7
x-coords[5]	0	1	1	5	6

Tabulka 3.1: Matice uložená ve formátu COO

Jak můžeme vidět, délka polí je závislá pouze na nnz. Pro velké matice s malým počtem neuspořádaných nenulových prvků je tento formát velmi efektivní. Pokud by bylo prvků velké množství, informace o uložení `y` nebo `x` souřadnic by byla často redundatní.

Paměťová náročnost formátu COO je $O(3 * nnz)$.

Formát COO je velmi jednoduchý a přímočarý. Při procházení jeho prvků nám stačí jedna iterace přes tři stejně dlouhé pole. Tím je tedy například algoritmus násobení řídké matice ve formátu COO s vektorem velmi jednoduchý:

Algorithm 6 Násobení matice COO s vektorem

```
procedure COO-MVM(COO,V,C)
    for i←0 to COO.nnz do
        V.v[COO.r[i]] += COO.v[i] * V.v[COO.c[i]];
    end for
end procedure
```

Při násobení dvou matic narazíme na problém. Při této operaci se každý prvek násobí dvakrát. Potřebujeme způsob, jak se v matici vrátit zpátky na určité místo. Takový naivní algoritmus pro násobení dvou COO matic by byl složitý. Museli bychom si pamatovat začátky řádků a neustále kontrolovat, jestli jsme nepřesáhli další řádek. Lepším řešením je dopředu si předpočítat, kde který řádek začíná a končí. Předpočítáme si tedy pole, nazvané *row_pointers*, o délce $M.height + 1$, obsahující indexy začátků a konců řádek.

Při procháchezní matice se tak v poli s prvky mezi indexy $row\_pointers[i]$ a $row\_pointers[i+1]$ nachází prkvy na řádku i. Proto je pole právě o jedna delší než výska matice, abychom mohli určit konec posledního řádku.

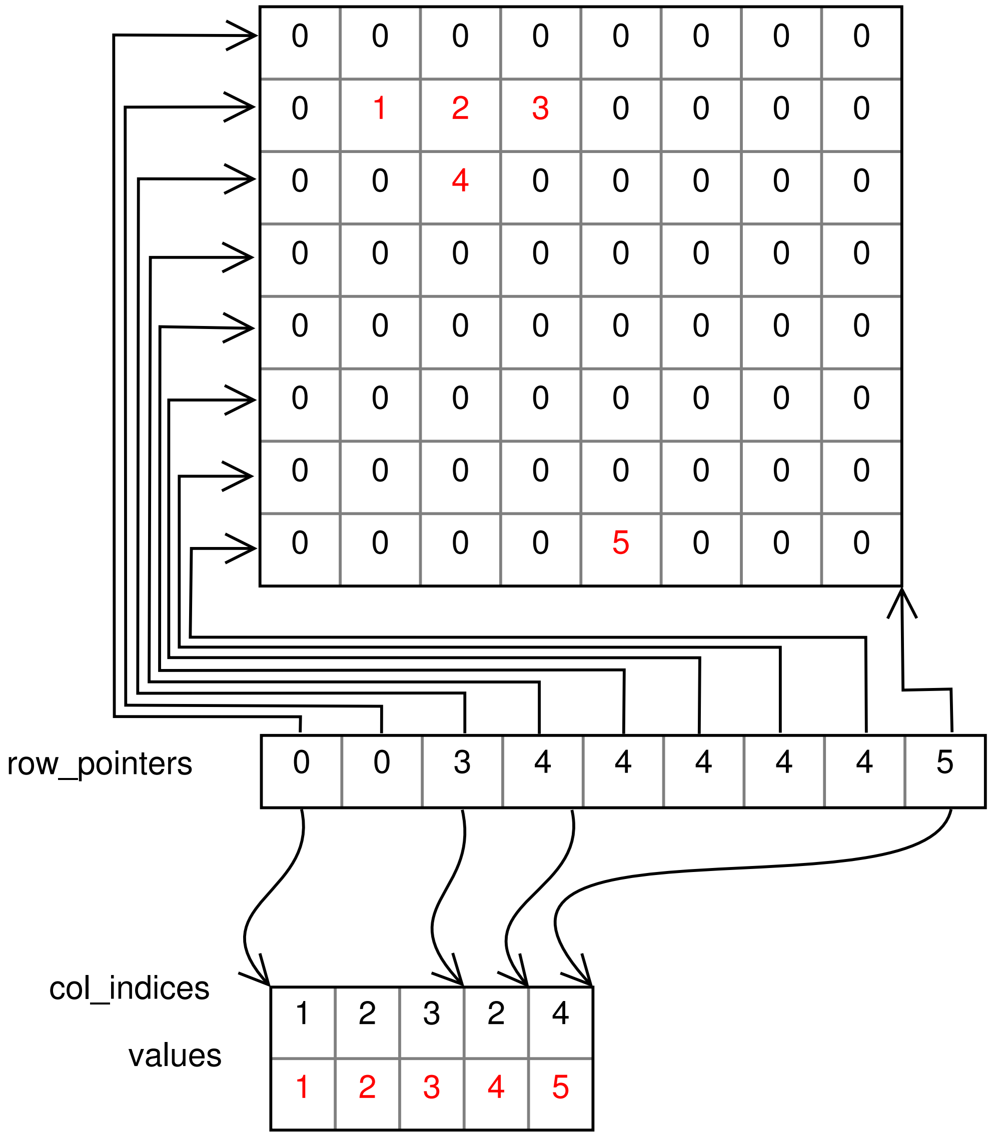
Algorithm 7 Násobení dvou COO matic

```
procedure COO-MMM(A,B,C)
    arp ← InitArray(A.nnz + 1);
    brp ← InitArray(B.nnz + 1);
    for i ← 0 to A.nnz do                ▷ předpočítání prvků v řádcích matice A
        arp[A.r[i] + 1] = arp[A.r[i] + 1] + 1;
    end for
    for i ← 0 to A.height do
        arp[i + 1] = arp[i + 1] + arp[i];
    end for
    for i ← 0 to B.nnz do                ▷ předpočítání prvků v řádcích matice B
        brp[B.r[i] + 1] = brp[B.r[i] + 1] + 1;
    end for
    for i ← 0 to A.height do
        brp[i + 1] = brp[i + 1] + brp[i];
    end for
    for i ← 0 to A.height do                                         ▷ násobení
        for ac←arp[i] to arp[i + 1] do
            for bc←brp[A.c[ac]] to brp[A.c[ac] + 1] do
                C.v[r][A.c[bc]] += A.v[ac] * B.v[bc];
            end for
        end for
    end for
end procedure
```

V části s násobením již pole s `y` souřadnicemi nepotřebujeme. Lepší formát uložení řídkých matic pro násobení by byl takový, který namíto pole s `y` souřadnicemi obsahuje předpočítané začátky a konce řádků. Takovým formátem je CSR.

3.2 CSR - Compressed sparse row

Problém efektivnosti formátu COO pro větší množství prvků řeší formát CSR, česky komprimované řídké řádky. Formát CSR obsahuje pole *row_pointers*, které ukládá informace o tom, kolik se v daném řádku nachází prvků, tedy přesně to, co jsme si přespočítali v algoritmu 7. K poli s hodnotami je další pole *col_indicies*, přiřazující ke každému prvku informaci o sloupci.



Obrázek 3.1: Matice uložená ve formátu CSR

Jak je vidět z ilustrace 3.1, řádek s více prvky je uložen efektivně. Díky prázdným řádkům se nezdá pole *row_pointers* rozumně využité.

Při násobení matice CSR s vektorem potřebujeme o jeden for cyklus více než v případě násobení matice COO s vektorem. Důvodem je ztráta informace o řádku prvku.

Algorithm 8 Násobení matice CSR s vektorem

```
procedure CSR-MVM(CSR,V,C)
    for i ← 0  to CSR.h do
        for ci ← CSR.rp[i] to CSR.rp[i + 1] do
            C.v[r] += CSR.v[ci] * V.v[A.ci[ci]];
        end for
    end for
end procedure
```

Násobení dvou CSR matic je stejné jak v případě násobení dvou COO matic 3.1. Jediný rozdíl je, že předpočíné začátky a konce řádků jsou součástí formátu.

Algorithm 9 Násobení dvou CSR matic

```
procedure CSR-MMM(A,B,C)
    for i ← 0 to A.height do                                    ▷ násobení
        for ac ← A.rp[i] to A.rp[i + 1] do
            for bc ← B.rp[A.ci[ac]] to B.rp[A.ci[ac] + 1] do
                C.v[r][B.ci[bc]] += A.v[ac] * B.v[bc];
            end for
        end for
    end for
end procedure
```

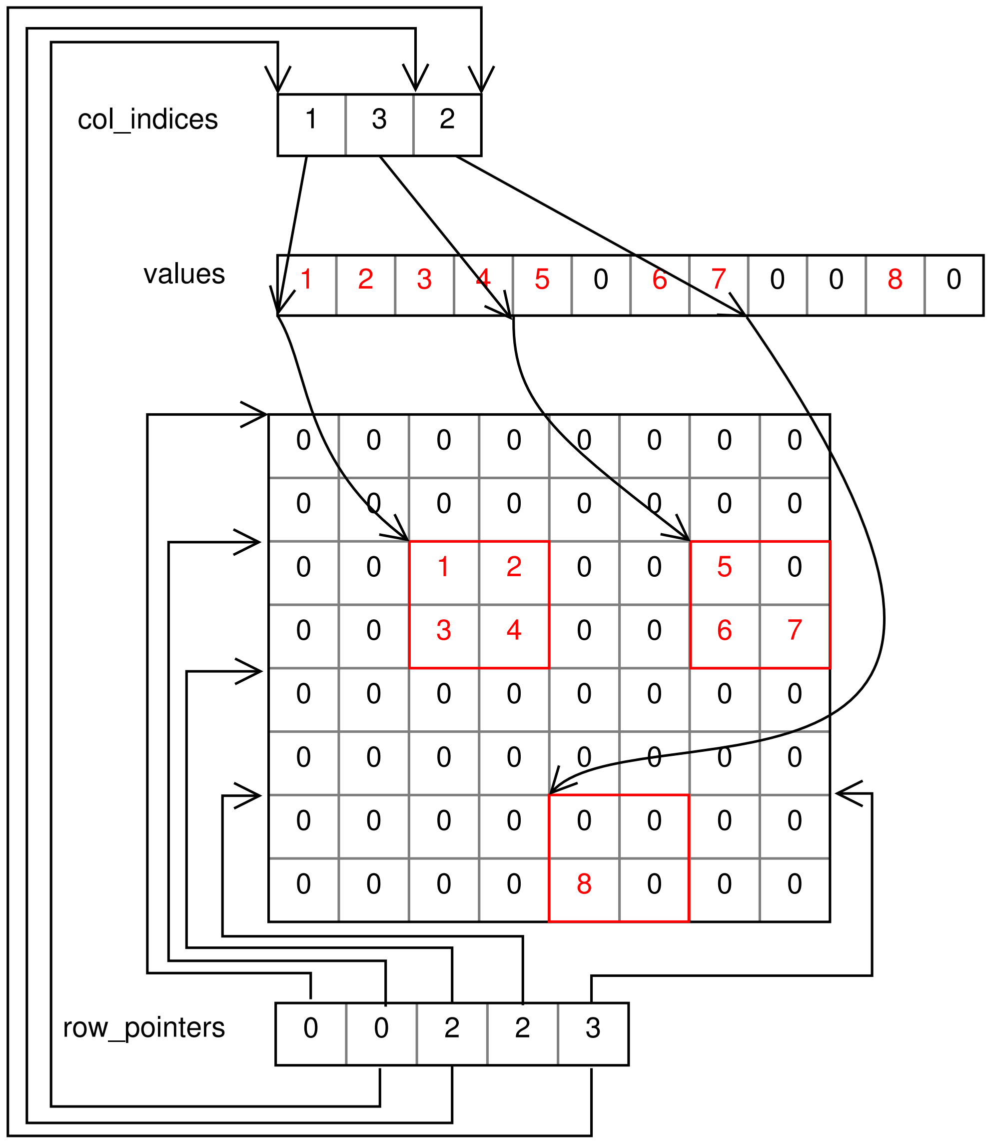
Existuje varianta tohoto formátu, nazvaná CSC - compressed sparse columns, která místo ukládání řádku ukládá sloupce.

3.3 BSR - Block Sparse Row

Jako formát CSR využívá uložení prvků v řádku, formát BSR ještě navíc detekuje a ukládá prvky v blocích.

Protože při násobení matic násobíme každý prvek dvakrát, bylo by dobré tyto dvě operace provést co nejdříve, abychom při druhém znovunačtení prvku mohli sáhnout pro prvek do cache. Pokud jsou prkvy procházené po menších blocích, dostaneme se k prvku podruhé dříve, než jej z cache přemaže jiný prvek.

Matici A ve formátu BSR uklákáme, velice podobně jako u formátu CSR, pomocí tří polí. Je potřeba i jedna proměnná, která uchovává velikost bloku. Tuto proměnnou nazveme *block_size*. Matici rozdělíme do bloků Pole *col_indices* označuje sloupec, ve kterém se blok nachází. Sloupcem rozumíme $A.width/A.block\_size$.



Obrázek 3.2: Matice uložená ve formátu BSR

Uložení matice ve formátu BSR ilustruje obrázek 3.4. Pole *row_pointers* obsahuje informace o tom, na kterém řádku je kolik bloků. Na řádku i je $row\_pointers[i+1] - row\_pointers[i]$ bloků. Ve kterém sloupci se blok prvků nachází udává pole *col_indices*. První blok z řádku i je ve sloupci $col\_indices[row\_pointers[i]]$ a poslední blok je ve sloupci $col\_indices[row\_pointers[i+1]]$. Pole *values* obsahuje prvky v blocích, včetně nulových prvků.

Pokud vezmeme algoritmus násobení dvou CSR matic respektive CSR matice s vektorem, jen místo prvků násobíme bloky, vznikne nám algoritmus pro násobení dvou BSR matric respektive BSR matice s vektorem. Za povšimnutí stojí větší počet for cyklů s menším rozsahem iterace než u CSR. To nám v nejvnitřnějších cyklech dovoluje lépe využívat cache. Jedná se tedy o přístup podobný optimalizační technice loop tiling.

Algorithm 10 Násobení matice BSR s vektorem

```
procedure BSR-MVM(A,B,C)
    bs ← A.block_size;
    for i ← 0 to A.height / bs do
        for ac ← A.rp[i] to A.rp[i + 1] do
            for l ← 0  to A.bs do                              ▷ násobení bloku
                for m ← 0 to bs do
                    C.v[(i * bs) + l] += A.v[ac * (bs * bs) + (l * bs) + m] * B.v[A.ci[ac] * bs + m];
                end for
            end for
        end for
    end for
end procedure
```

Algorithm 11 Násobení dvou BSR matic

```
procedure BSR-MMM(A,B,C)
    bs ← A.block_size;
    for i ← 0 to A.height / bs do
        for ac ← A.rp[i] to A.rp[i+1] do
            for bc ← B.rp[A.ci[ac]] to B.rp[A.ci[ac]+1] do
                for l ← 0 to A.bs do                           ▷ násobení bloku
                    for m ← 0 to bs do
                        for n ← 0 to bs do
                            C.v[(i * bs) + l][(B.ci[bc] * bs) + m] += A.v[ac * (bs * bs) + (l * bs) + n] * B.v[bc * (bs * bs) + (n * bs) + m];
                        end for
                    end for
                end for
            end for
        end for
    end for
end procedure
```

http://docs.scipy.org/doc/scipy-0.13.0/reference/generated/scipy.sparse.bsr_matrix.ht
https://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mklman/GUID-9FCEB1C4-670D-4738-81D2-F378013412B0.htm

3.4 Quadtree

Předchozí popsáné formáty uložení řídkých matic jsou velmi přímočaré. Neumožnují rekurzivní přístup, který je pro velké matice vhodnější. Tento problém řeší formát Quadtree [17][1]. Jedná se o matici uloženou v 4-árním stromě.

Tento formát dělí matici na čtvrtiny do té doby, než se dosáhne velikosti podmatice *sm_size*. Pokud je celá čtvrtina prázdná, je uzel označen jako prázdný, označen E. Pokud obsahuje nějaké prvky, je list označen jako hustý, D. Vnitřní uzly jsou označeny jako smíšené, tedy M.

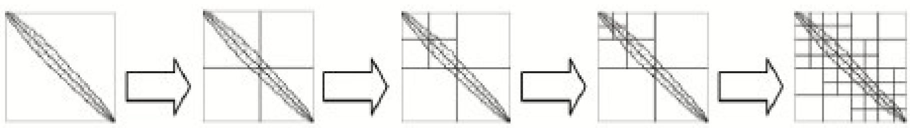
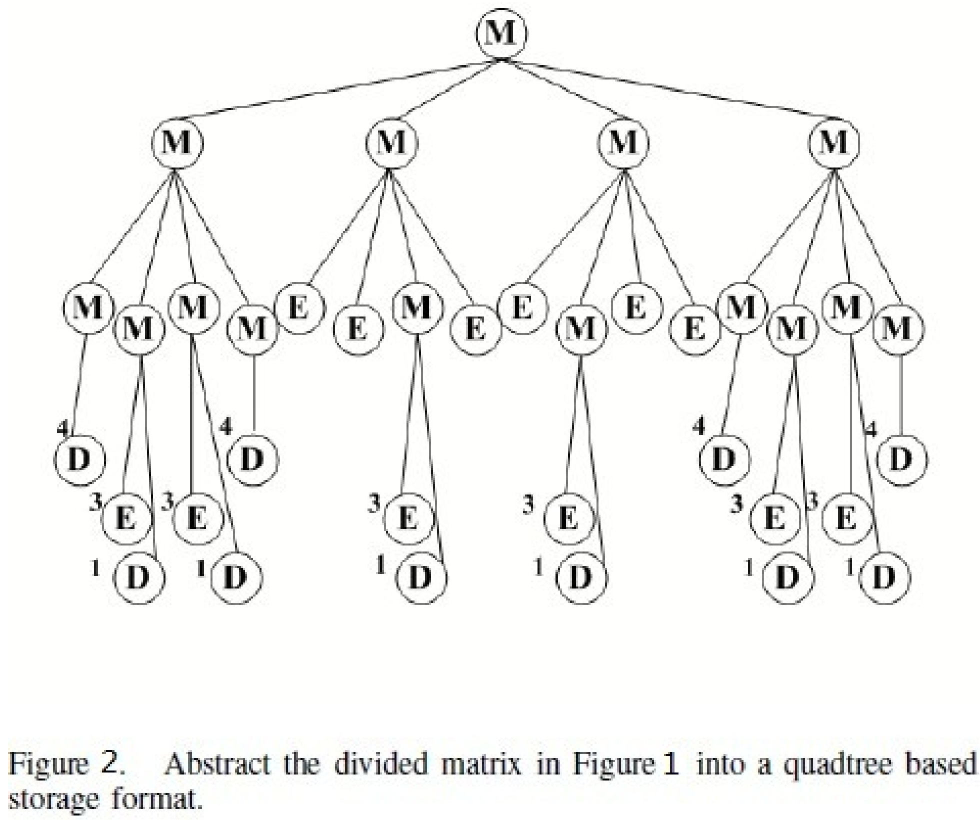


Figure 1. Recursive division on matrix $BCSSTK15$(Model of an offshore platform).

Obrázek 3.3: Rozdělení matice



Obrázek 3.4: Strom matice uložené ve formátu Quadtree

Algoritmy pro práci s formátem Quadtree budou ukázány v kapitole o obecnějším formátě 3.5.

3.5 ?

TODO: tady jsem chtel spocictat kdy se vyplati mit ridkou matici, ale lepsi bude tabulka. Pokud například uložíme matici o rozměrech 100x100 v dvojté přestnosti, bude zabírat `M x N x sizeof(double) = 100 x 100 x 8 = 80000B = 80kB`. Pokud zvolíme řídký formát matice, kde ke každému elementu uložíme i jeho x a y souřadnici, tak do 80kB uložíme `80000 / (sizeof(int)+sizeof(int)+sizeof(double)) = 80000/16= 5000` elementů. Pokud matice obsahuje více jak 50 % nulových elementů, vyplatí se nám ji uložit do řídkého formátu.

Kapitola 4

Modifikace formátu quadtree

Protože formát Quadtree je 4-ární strom, jeho výška při reprezentaci matice o velikosti n a při velikosti podmatice sm_size je:

$$\left\lceil \log_4 \frac{n^2}{sm\_size^2} \right\rceil \tag{4.1}$$

Například pro matici o velikosti $n = 16384$ s velikostí bloku $sm\_size = 128$ je výška stromu 7.

4.1 KAT - k-ary tree matrix

V této práci navrhneme místo 4-árního stromu obecný k-ární strom. Výška stromu se tím sníží. Pro předchozí příklad by tedy pro 32-ární strom byla výška pouze 3. Tento formát nazveme `k-ary tree matrix` se zkratkou KAT. Pro přehlednost nebudeme uvádět k, ale $KAT.n = sqrt(k)$. Formát Quadtree můžeme prohlásit za matici KAT s $KAT.n = 2$.

Výška stromu pro KAT matici označíme jako `KAT.height` a vypočítáme ji jako výsku quadtree, jen s paremetrem k:

$$KAT.height = \left\lceil \log_k \frac{n^2}{sm\_size^2} \right\rceil \tag{4.2}$$

Pokud to bude z kontextu jasné, budeme výšku KAT stromu značit pouze *height*.

4.2 Typy listů

Formát Quadtree definovaný v (cituj) označí všechny nenulové listy jako husté a ukládá je v COO formátu. KAT matice obasahuje proměnnou $KAT.dense\_treshold$, která udává maximální počet prvků, aby se s listem pracovalo některým z řídkých formátů, tedy COO, CSR, BSR. Při překročení této hranice je list uložen ve formátu husté matice, tedy bude obsahovat i nulové prvky.

4.3 Tvoření stromu

Pro každý prvek procházíme strom a hledáme správný list. Protože výška stromu je daná třemy parametry, tedy velikostí matice n, velikostí podmatice sm_size a počtu větvení uzlu k, čelíme problému, nejefektivnější využití bude, pokud je n bezezbytku dělitelné $k \ast sm_{s}ize$. Pokud toto neplatí, rodiče listů budou mít část synů nevyužitých i při uložení husté matice.

Algorithm 12 Vyhledání listu pro KAT matici

```
procedure KAT-getNode(KAT, y, x)
    tmpNode ← KAT.root;
    blockY ← 0;                                        ▷ výřez matice
    blockX ← 0;
    blockS ← KAT.sm_size;
        ▷ až do předposledního vnitřního uzlu traverzujeme podle velikosti KAT.n
    while blockS > (KAT.n * KAT.sm_size) do
        nodeY ← (y - blockY) / (blockS / KAT.n);
        nodeX ← (x - blockX) / (blockS / KAT.n);
        tmpNode ← tmpNode.childs[nodeY][nodeX];
        blockY += nodeY * (blockS / KAT.n);
        blockX += nodeX * (blockS / KAT.n);
        blockS /= KAT.n;
    end while
        ▷ do posledního vnitnřního uzlu traverzujeme podle velikosti KAT.sm_size
    nodeY ← (y - blockY) / KAT.sm_size;
    nodeX ← (x - blockX) / KAT.sm_size;
    tmpNode ← tmpNode.childs[nodeY][nodeX];
                                                       ▷ nyní je tmpNode list
    tmpNode.x ← ⌊y / KAT.sm_size⌋ * KAT.sm_size ;
    tmpNode.x ← ⌊x / KAT.sm_size⌋ * KAT.sm_size ;
    return tmpNode;
end procedure
```

4.4 Násobení

Pro násobení budeme používat algoritmus rekurzivního násobení popsaného v 2.4. Popsaný algormus dělí matice na čtvrtiny, jde tedy aplikovat na formát Quadtree. Při násobení matice uložené v k-árním stromě budeme násobit matici podmatic o velikosti k:

Algorithm 13 Násobení matice KAT s vektorem

```
procedure KAT-MVM(KAT, KAT_node, VB, VC)
    for i ← 0 to KAT.n do
        for j ← 0 to KAT.n do
            if KAT_node.childs[i][j] ≠ NIL then
                if KAT_node.childs[i][j].type = "submatrix" then
                    multiplyNode(KAT_node.childs[i][j], VB, VC);
                    coninue;
                end if
                if KAT_node.childs[i][j].type = "inner" then
                    KAT-MVM(KAT,KAT_node.childs[i][j], VB, VC);
                    coninue;
                end if
            end if
        end for
    end for
end procedure
```

Algorithm 14 Násobení dvou KAT matic

```
procedure KAT-MMM(KATa, KATa_node, KATb, KATb_node, C)
    for i ← 0 to KAT.n do
        for j ← 0 to KAT.n do
            for k ← 0 to KAT.n do
                if KATa_node.childs[i][k] ≠ NIL and KATb_node.childs[k][j] ≠ NIL then
                    if KAT_node.childs[i][j].type = "submatrix" then
                        multiplyNode(KAT_node.childs[i][j], KAT_node.childs[i][j], VC);
                        coninue;
                    end if
                    if KAT_node.childs[i][k].type = "inner" then
                        KAT-MMM(KATa,KATa_node.childs[i][k], KATb,KATb_node.childs[k][j] C);
                        coninue;
                    end if
                end if
            end for
        end for
    end for
end procedure
```

Přesnější asymptotická složitost násobení matic v KAT formátu je vyšší než u předchozích formátů, protože je potřeba připočítat průchod stromem. Pokud převedeme hustou matici do formátu KAT se složitostí násobení dvou listů $\mathcal{O}(sm\_size^3)$, bude asyptotická složistost násobení dvou KAT matic $\mathcal{O}((\frac{n^2}{sm\_size} * (height + sm\_size^3))$.

Paměťová složitost celé KAT matice záleží na typu a hustotě uložení dat. Uvedeme zde pouze paměťovou složitost stromu. Velikost uzlu je pole k odkazů na syny. Počet vnitřních uzlů spočítáme pomocí geometrické posloupnosti. Paměťová složitost stromu KAT matice tedy je $\mathcal{O}(\frac{k^{height-1}-1}{k-1} * k)$

Kapitola 5

Analýza a návrh

V této kapitole představíme některé známé systémy, které umí pracovat s řídkými maticemi. Popíšeme možnosti implemetntace a zvolené řešení.

Pokud potřebujeme rychle vynásobit dvě malé matice, můžeme použít obecný nástroj se kterým se snadno pracuje. Čím je ale nástroj obecnější, tím spíš bude pomalejší. Maximální výkonnosti tedy doáshneme, pokud co nejvíce využijeme hardware a náš program bude řešit konkrétní úkol.

5.1 ATLAS

ATLAS [20] je v překladu zkratkou pro automatické generování vylazeného kódu pro sofware řešící úlohy z lienální algebry. Je tak dosaženo pomocí jak obecných optimalizací kódu, tak optimalizací pro konkrétní architektury.

Podobných projektů je více, konkrétně pro násobení matice s vektorem je to například Sparsity [3], nebo pro násobení matice s maticí je to například PHiPAC [21][4].

5.1.1 SciPy.sparse

SciPy je knihovna skriptovacího jazyka Python [13]. Obsahuje modul pro základní práci s řídkými maticemi, nazvaný sparse [8]. Samotné výpočty nad řídkými maticemi jsou v této knihovně z důvodu rychlosti napsány v jazyce C++.

5.1.2 Wolfram Mathematica

Wolfram Mathematica [25] je mocný výpočetní software. Práce v tomto software je snadná, interaktivní a přitom dokáže efektivně rozdělovat práci nejen na více procesorových jader, ale taky využít i grafické karty. Nechybí tedy ani implementace operací pro řídké matice.

V této práci pomocí Wolfram Mathematicy vizualizujeme řídké matice z formátu `.mtx`, který bude popsán v 6.5. Mathematica umí pracovat i s maticemi v souborech `.mtx.gz`, tedy komprimovanými soubory. Příkaz pro importování řídké matice a její vizualizaci je `Import["/var/tmp/matrix.mtx", "Graphics"]`, následné uložení matice lze provést například příkazem `Export["/var/tmp/matrix.pdf", %]` [26].

5.1.3 Boost

Pro práci s řídkými maticemi v jazyce C++ je možné použít knihovnu uBLAS[15] ze sady knihoven Boost[2]. Knihovna uBLAS obsahuje algoritmy pro řešení základních ukolů z lineálrní algebry. Protože je kód napsaný pomocí C++ šablon, je vygenerovaný kód výkonný.

5.2 Možnosti řešení

Jedna z možných variant pro porovnání všech zvolených formátů byla doimplementovat formát Quadtree, respektive KAT do knihovny SciPy.sparse. Takový program by byl ale příliš komplexní a nepřímočarý.

Další možnost byla pokračovat s semestrální práci z předmětu Efektivní implementace algoritmů s tématem násobení matic ve formátu CSR.

5.3 Zvolené řešení

Protože se v této práci zabýváme pouze vlivem uložení řídké matice pro operaci násobení, rozhodli jsme se vytvořit program jen pro tuto operaci. Obecná knihovna pro práci s řídkými maticemi, jakým například SciPy.sparse je, nemůže využít některých vlastností matic při jejich násobení. Hlavně zacházení s násobenými maticemi jako s konstantami.

Navážeme tedy na semestrální práci. Vytvoříme více druhů matic a pro každou implementujeme operaci násobení s maticí a násobení s vektorem.

Kapitola 6

Realizace

6.1 Prostředí

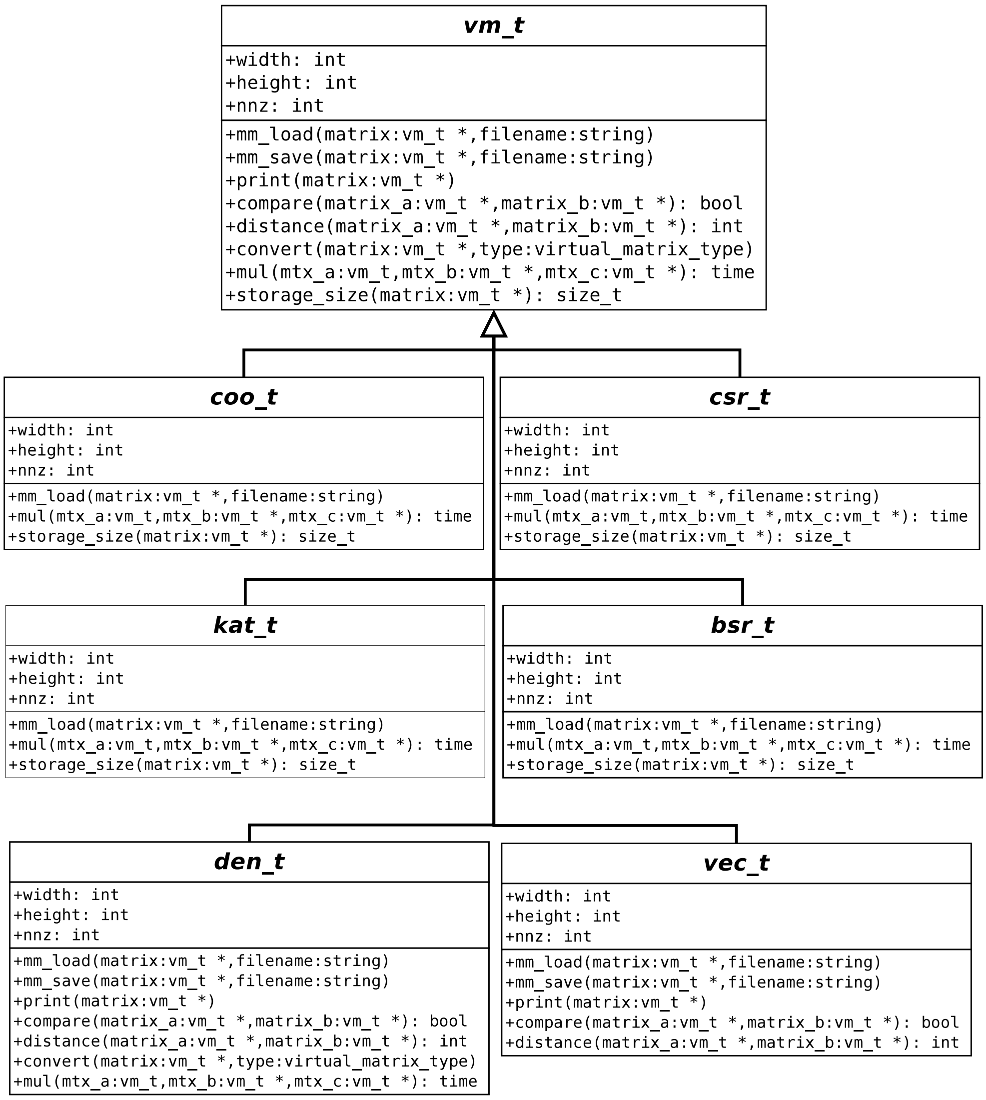
Popsané algoritmy jsme implementovali v jazyce C99. Implementace probíhala v operačním systému Lubuntu, v IDE Eclipse CDT. Zdrojový kód byl kompilován pomocí překladače GNU C.

K lazení chyb pro práci s pamětí jsme použili nástroj Valgrind s přepínači `- - leak-check=full`. Běžící progam jsme krokovali pomocí nástoje GNU Debugger.

6.2 Design implementace

Popsané algoritmy byly implementovány v progrovacím jazyce C99.

Protože matice mají některé vlastnosti stejné a lze s podobně pracovat, byl zvolen jednoduchý objektově orientovaný model [16]. Nadřazeným objektem je virtuální matice vm_t, obsahující virtualní tabulku funkcí.



Obrázek 6.1: UML diagram tříd programu

Výsledný program načte dvě matice v určitém formátu a vynásobí je. Pomocí přepínačů lze zobrazit, nebo uložit výslednou matici, popřípadě vypsat některé údaje o načtených maticích.

6.3 Implementace KAT

U formátů COO, CSR, BSR je implementace přímočará podle pseudokódů. U implementace KAT máme více možností, proto zde popíšeme naší implementaci.

Jeden z důležitých parametrů je k, tedy maximální počet synů vnitřích uzlů. V naší implementaci je tento parametr uložen v konstantě `KAT.n`, pro připomenutí $KAT.n = sqrt(k)$. Překladač poté cykly, kde iterujeme do této konstanty, rozbalí. Překladači i explicintě zdělujeme, ať cykly rozbalí přes atribut `__attribute__((optimize("unroll-loops")))`.

V bakalářské práci Rozšíření implementace formátu kvadrantového stromu od Tomáše Karabely [CITACE] je Quadtree implementován jako strom, jehož listy tvoří virtuální matice podobným těm v naší práci. Protože jeho formát byl určený pro algoritmus LU rozklad, jeho virtuální matice musely umět přijímat i další prvky. Protože v naší práci se s formáty uložení řídké matice zachází jako s konstantami, rozhodli jsme se hodnoty prvků a informace v polích `row_pointers` a `col_indices` uložit mimo listy stromu. V listech se nachází ukazatel do velkého pole na příslušné místo pro daný list. Ušetříme tak práci knihovně libc s mnohonásobným volání funkce `malloc`.

Protože dovolujeme dva druhy listů, tedy hustý a řídký ve formátu CSR, bylo potřeba implementovat následující algoritmy:

1. hustý list * hustý list
2. hustý list * CSR list
3. hustý list * vektor
4. CSR list * hustý list
5. CSR list * CSR list
6. CSR list * vektor

Protože tyto algoritmy jsou velice podobné již popsaným algoritmům, ukážeme zde pro představu násobení CSR listu s hustým listem.

Algorithm 15 Násobení hustého KAT listu s CSR listem

```
procedure KAT-MMM-DEN-CSR(ka,kb,na,nb,c)    ▷ ka,kb=KAT matice; na,nb=listy, c = hustá matice
    for i ← 0 to ka.sm_size do
        for j ← na.rp[i] to na.rp[i] do
            for k ← 0 to ka.sm_size do
                C.v[na.y + i][nb.x + k] += na.v[j] * nb.v[ka.sm_size * na.ci[j] + j];
            end for
        end for
    end for
end procedure
```

6.4 Testování

Pro ověření správnosti algoritmů je potřeba testovací software. Strategie testování spočívá ve výběru testovacích matic, vynásobení v hustém formátu uložení, vynásobení v některém z řídkých formátů uložení a porovnaní výsledků.

Algorithm 16 Testování

```
procedure TestFormats(PairList, FormatList)
    for all pair ∈ PairList do
        denseA ← vm_load(pair.a, DENSE);
        denseB ← vm_load(pair.b, DENSE);
        denseC ← vm_mul(denseA, denseB, denseC);
        for all format ∈ FormatList do
            sparseA ← vm_load(pair.a, format);
            sparseB ← vm_load(pair.b, format);
            sparseC ← vm_mul(sparseA, sparseB, sparseC);
            if vm_compare(denseC, sparseC) = NOT_SAME then
                print("Error:", pair.a, pair.b, format);
            end if
        end for
    end for
end procedure
```

Pro přehled o nefungujících konfiguracích jsme použili testovací framework Cassertion.

6.5 MatrixMarket

MatrixMarket [5] je internetová sada matic s vlastním formátem pro uložení řídkých matic `.mtx`. Tato sada obsahuje skoro pětset matic z různých oblastí. Obsahuje i generátory řídkých matic, jejichž výstupy jsou matice různých vlastností.

Pro náš program jsme zvolili právě formát `.mtx`. Podporujeme dva druhy tohoto formátu. Nesymetrický, s banerem `%%MatrixMarket matrix coordinate real symmetric` a symetrický s banerem `%%MatrixMarket matrix coordinate real general`.

6.5.1 Generátor řídkých matic

Generátory z MatrixMarketu běží v internetovém prohlížeči a v jazyce Java. Protože takto není jednoduché matice generovat ve skriptu, aby jsme při distribuci našeho programu nemuseli přikládat velké testovací matice, implementovali jsme jednoduchý generátor řídkých matic. Parametry předáváme programu informace o výsledné matici a seznam objektů, tedy buď podmatic nebo diagonál, které se mají být do matice zahrnuty. Manuál ke generátoru je možné vypsát zavoláním `./tests/bin/matrix_generator -h`.

Algorithm 17 Generování řídkých matic

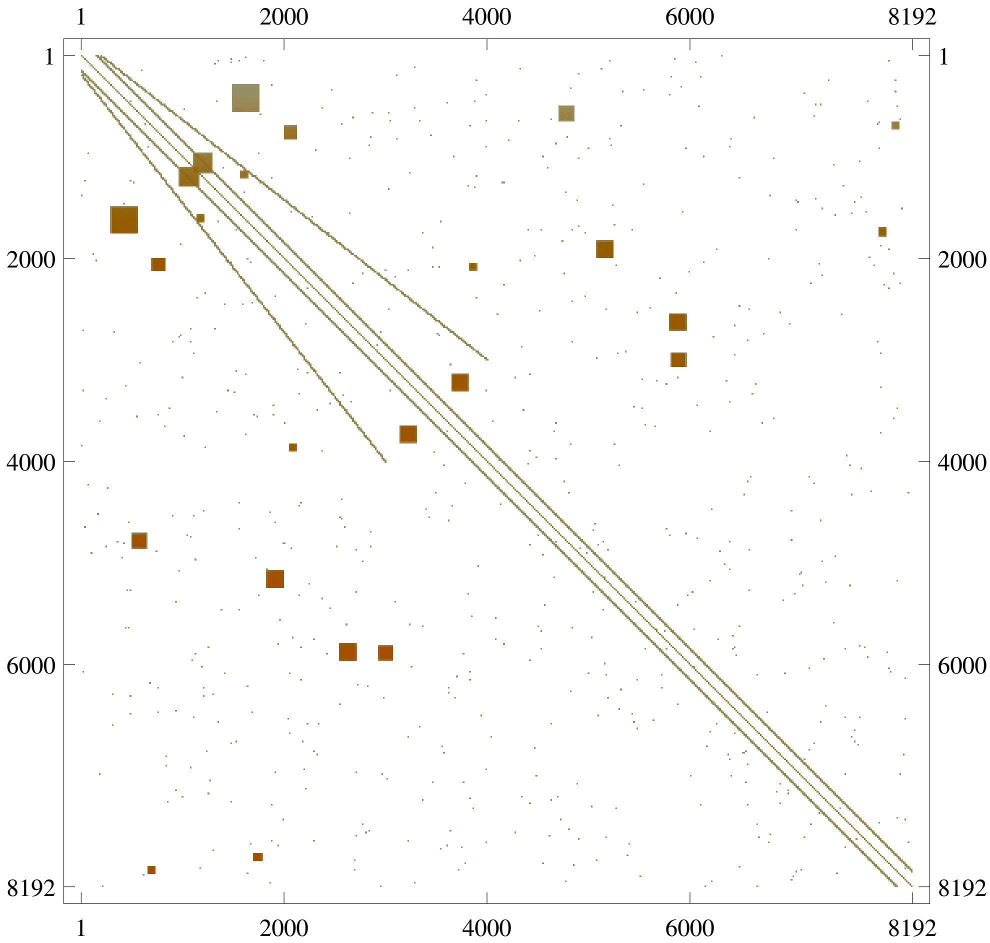
procedure SPARSEMATRIXGENERATOR($file, width, height, ItemList$)
  $MtxWrapper \leftarrow InitMtxWrapper()$;
  $MtxWrapper.PositionVector \leftarrow InitVector()$;
  for all $Item \in ItemList$ **do**
    $MtxWrapper.addItem(Item.y, Item.x, Item.properties)$;
    if $Item.type == Mirrored$ **then**
      $MtxWrapper.addItem(Item.x, Item.y, Item.properties)$;
    end if
  end for
  $MtxWrapper.PositionVector.sort()$;
  $MtxWrapper.PositionVector.removeDuplicates()$;
  $MtxWrapper.write(file)$;
end procedure

Generátor řídkých matic byl implementován v jednom souboru. Lze spouštět s následujícími parametry:

- `-c` matice bude obsahovat hlavní diagonálu
- `-H <celé číslo>` výška matice
- `-i <typ,a,b,c,d,...>` seznam objektů, které se do matice přidají
  - `diagonal,ay,ax,by,bx,sparsity` prvky v přímce od bodu `[ax,ay]` do bodu `[bx,by]` s řídkostí `sparsity`
  - `block,ay,ax,by,bx,sparsity` blok prvků v obdelníku ohraničiného body `[ax,ay]` a `[bx,by]` s řídkostí `sparsity`
- `-n <celé číslo>` velikost matice
- `-o <soubor>` cílový soubor (lze použít i stdout)
- `-s <desetinné číslo>` řídkost matice (`sparsity`)
- `-S <desetine číslo>` startovací číslo
- `-W <celé číslo>` šířka matice
- `-h` zobraz nápovědu
- `-o <soubor>` cílový soubor (lze použít i `stdout`)

- `-v` vypisuj průběh generování (`verbose`)

$./tests/bin/matrix_generator-n8192-s0.00001-imdiagonal,150,0,8192,8042,0.95,mdiagonal,2$
$o/tmp/matrix2.mtx$



Obrázek 6.2: Matice vygenerovaná generátorem

6.6 Optimalizace

TODO: popsat moznosti optimalizace

6.7 Měření

TODO: popsat jak to budu měřit, tedy cas z omp a cachegrind/callgrind

Závěr

Zaver.

Literatura

[1] *A Quadtree Based Storage Format and Effective Multiplication Algorithm for Sparse Matrix [online].* [cit. 2014-04-27]. Dostupné z: http://quardtree.sourceforge.net/

[2] Beman Dawes, R. R., David Abrahams: Boost. 2007. Dostupné z: http://www.boost.org/

[3] of California at Berkeley, U.: Sparsity. 2014. Dostupné z: http://www.cs.berkeley.edu/~yelick/sparsity/

[4] Bilmes, J.; Asanović, K.; Demmel, J.; aj.: PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology and its application to Matrix Multiply. LAPACK working note 111, University of Tennessee, 1996.

[5] Boisvert, R. F.; Pozo, R.; Remington, K.; aj.: Matrix Market: A Web Resource for Test Matrix Collections. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement*, London, UK, UK: Chapman & Hall, Ltd., 1997, ISBN 0-412-80530-8, s. 125–137. Dostupné z: http://dl.acm.org/citation.cfm?id=265834.265854

[6] Borsic, A.; Halter, R.; Wan, Y.; aj.: Sensitivity study and optimization of a 3D electric impedance tomography prostate probe. *Physiological Measurement*, ročník 30, č. 6, 2009: str. S1. Dostupné z: http://stacks.iop.org/0967-3334/30/i=6/a=S01

[7] Cenk, M.; Hasan, M. A.: On the Arithmetic Complexity of Strassen-Like Matrix Multiplications. *IACR Cryptology ePrint Archive*, ročník 2013, 2013: str. 107.

[8] community, T. S.: Sparse matrices (scipy.sparse). 2014. Dostupné z: http://docs.scipy.org/doc/scipy/reference/sparse.html

[9] Coppersmith, D.; Winograd, S.: Matrix Multiplication via Arithmetic Progressions. *J. Symb. Comput.*, ročník 9, č. 3, 1990: s. 251–280.

[10] Cormen, T. H.; Stein, C.; Rivest, R. L.; aj.: *Introduction to Algorithms*. McGraw-Hill Higher Education, druhé vydání, 2001, ISBN 0070131511.

[11] Davis, T. A.; Hu, Y.: The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, ročník 38, č. 1, Prosinec 2011: s. 1:1–1:25, ISSN 0098-3500, doi:10.1145/2049662.2049663. Dostupné z: http://www.cise.ufl.edu/research/sparse/matrices

[12] El-Kurdi, Y.; Gross, W.; Giannacopoulos, D.: Sparse Matrix-Vector Multiplication for Finite Element Method Matrices on FPGAs. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, April 2006, s. 293–294, doi:10.1109/FCCM.2006.65.

[13] Foundation, P. S.: Python. 2014. Dostupné z: https://www.python.org/

[14] Gates, A. Q.; Kreinovich, V.: Strassen's Algorithm Made (Somewhat) More Natural: A Pedagogical Remark. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, ročník 73, 2001: s. 142–145.

[15] Joerg Walter, G. W. D. B., Mathias Koch: Basic Linear Algebra Library. 2010. Dostupné z: http://www.boost.org/doc/libs/1_55_0/libs/numeric/ublas/doc/index.htm

[16] Schreiner, A.: *Objektorientierte Programmierung mit ANSI C*. Hanser, 1994, ISBN 9783446174269. Dostupné z: http://books.google.cz/books?id=sgA2AAAACAAJ

[17] Šimeček, I.: Sparse Matrix Computations with Quadtrees. In *Seminar on Numerical Analysis*, Liberec: Technical University, 2008, ISBN 978-80-7372-298-2, s. 122–124.

[18] Stothers, A. J.: *On the Complexity of Matrix Multiplication*. diploma thesis, University of Edinburgh, 2010.

[19] Strassen, V.: Gaussian Elimination is not Optimal. *Numerische Mathematik*, ročník 13, č. 4, Prosinec 1967: s. 354–355.

[20] of Tennessee, U.: Automatically Tuned Linear Algebra Software. 2014. Dostupné z: http://acts.nersc.gov/atlas/

[21] of Tennessee, U.: PHiPAC. 2014. Dostupné z: http://www1.icsi.berkeley.edu/~bilmes/phipac/

[22] Tvrdík, P.; Šimeček, I.: A new diagonal blocking format and model of cache behavior for sparse matrices.

[23] Williams, V. V.: Breaking the Coppersmith-Winograd barrier. 2011.

[24] Williams, V. V.: Multiplying matrices faster than coppersmith-winograd. In *STOC*, editace H. J. Karloff; T. Pitassi, ACM, 2012, ISBN 978-1-4503-1245-5, s. 887–898.

[25] Wolfram: Wolfram Mathematica 9. 2014. Dostupné z: `http://www.wolfram.com/mathematica/`

[26] Wolfram: Wolfram Mathematica 9 Documentation Center: MTX. 2014. Dostupné z: `http://reference.wolfram.com/mathematica/ref/format/MTX.html`

[27] Yuster, R.; Zwick, U.: Fast sparse matrix multiplication. *ACM Transactions on Algorithms*, ročník 1, č. 1, 2005: s. 2–13.

Příloha A

Seznam použitých zkratek

COO Coordinate

BSR Block sparse row

CSC Column sparse column

CSR Column sparse row

KAT k-ary tree

Seznam obrázků

1.1 3D neorientovaný graf ve tvaru helikoptéry, jeho reprezentace v řídké matici a výsledek simulace 4

2.1 Strassen (TODO: převzato z wikipedie: předělat?) 11
2.2 Ukázka numerické stability Strassenova algoritmu 14
2.3 Matice orsirr_1 před vynásobením 16
2.4 Matice orsirr_1 po vynásobením sama se sebou 17

3.1 Matice uložená ve formátu CSR 22

3.2 Matice uložená ve formátu BSR 24
3.3 Rozdělení matice 26
3.4 Strom matice uložené ve formátu Quadtree 27

6.1 UML diagram tříd programu 36
6.2 Matice vygenerovaná generátorem 40

Seznam algoritmů

1 Násobení matic podle definice 7
2 Násobení transponovanou maticí 8
3 Násobení po řádcích 9
4 Rekurzivní násobení 9
5 Strassenův algoritmus 12
6 Násobení matice COO s vektorem 20
7 Násobení dvou COO matic 21
8 Násobení matice CSR s vektorem 23
9 Násobení dvou CSR matic 23
10 Násobení matice BSR s vektorem 25
11 Násobení dvou BSR matic 25
12 Vyhledání listu pro KAT matici 30
13 Násobení matice KAT s vektorem 31
14 Násobení dvou KAT matic 32
15 Násobení hustého KAT listu s CSR listem 37
16 Testování 38
17 Generování řídkých matic 39

*

Příloha B

Obsah přiloženého CD

```
 readme.txt ..................... stručný popis obsahu CD
 exe ................ adresář se spustitelnou formou implementace
 src
   impl ....................... zdrojové kódy implementace
   thesis .............. zdrojová forma práce ve formátu LaTeX
 text ........................................... text práce
   thesis.pdf ................... text práce ve formátu PDF
   thesis.ps ..................... text práce ve formátu PS
```