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

9. května 2014

Poděkování

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

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 zpracováním 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 9. května 2014 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

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

Tato práce popisuje formáty pro uložení řídkých matic COO, CSR, BSR, Quadtree a jeho modifikace v podobě snížení výšky stromu. Dále popisuje algoritmy pro násobení matic v těchto formátech. Součástí práce je i implementace těchto formátů a algoritmů v jazyce C a experimentální porovnání výkonnosti s teoretickými předpoklady.

Klíčová slova řídká matice, kvadrantový strom, násobení řídkých matic

Abstract

We describe sparse matrix storage formats COO, CSR, BSR, Quadtree and its modification in form of a lowering its height in this thesis. We also describe alogirthms for matrix multiplication in these formats. An implementation of these formats and algorithms in the C language with an experimental comparsion of measured results with theoretical assumptions is a part of this thesis too.

Keywords sparse matrix, quadtree, sparse matrix multiplication

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Při řešení problémů často hledáme způsob, jak interpretovat data v takovém formátu, se kterým již umíme pracovat. Jedním z takových základních prvků je matice. V některých případech takové matice obsahují nemalý počet nulových prvků. Takové matice obecně nazýváme řídké a při práci s nimi této vlastnost využíváme.

Binární operace násobení nad množinou matic patří mezi základní operace z lineární algebry. Pomocí násobení matic lze skládat lineární transformace. Tato operace najde uplatnění v mnohých vědeckých disciplínách.

Formát uložení řídké matice má vliv na výkonnost násobení řídkých matic. V této práci popíšeme formáty COO, CSR, BSR, Quadtree a navrhneme modifikaci formátu Quadtree snížením výšky stromu. Implementujeme tyto formáty spolu s algoritmy pro násobení matic v těchto formátech a porovnáme teoretické předpoklady s naměřenými hodnotami.

V první kapitole ukážeme některé příklady použití násobení řídkých matic v praxi a definujeme některé pojmy. Ve druhé kapitole ukážeme obecné algoritmy pro násobení matic a ve třetí kapitole popíšeme řídké formáty COO, CSR, BSR, Quadtree a jeho modifikaci a algoritmy pro násobení matic v těchto formátech. Ve čtvrté kapitole ukážeme některé současné programy pro práci s řídkými maticemi. V páté kapitole popíšeme naší implementaci a ukážeme naměřené hodnoty.

Kapitola 1

Úvod do problematiky

1.1 Řídké matice

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 [13]. Pro estetické zobrazení řídkých matic byla provedena simulace[12]. Každému uzlu byl přiřazen elektrický náboj a každá hrana představovala pružinu. V simulaci byla celá tato konstrukce položena na tvrdou podložku. Simulace byla zastavena v okamžiku, když se konstrukce přestala hýbat. Pro ilustraci přikládáme výsledek simulace 1.1 na modelu helikoptéry, tedy neorientovaném 3D grafu uloženém v řídké matici.

Řídké matice jsou používané ve velké škále oblastí[11], například výpočty diferenciálních rovnic, Google page rank, 3D grafika, statistika, těžení z dat, kryptografie a další.

1.2 Použití násobení matic

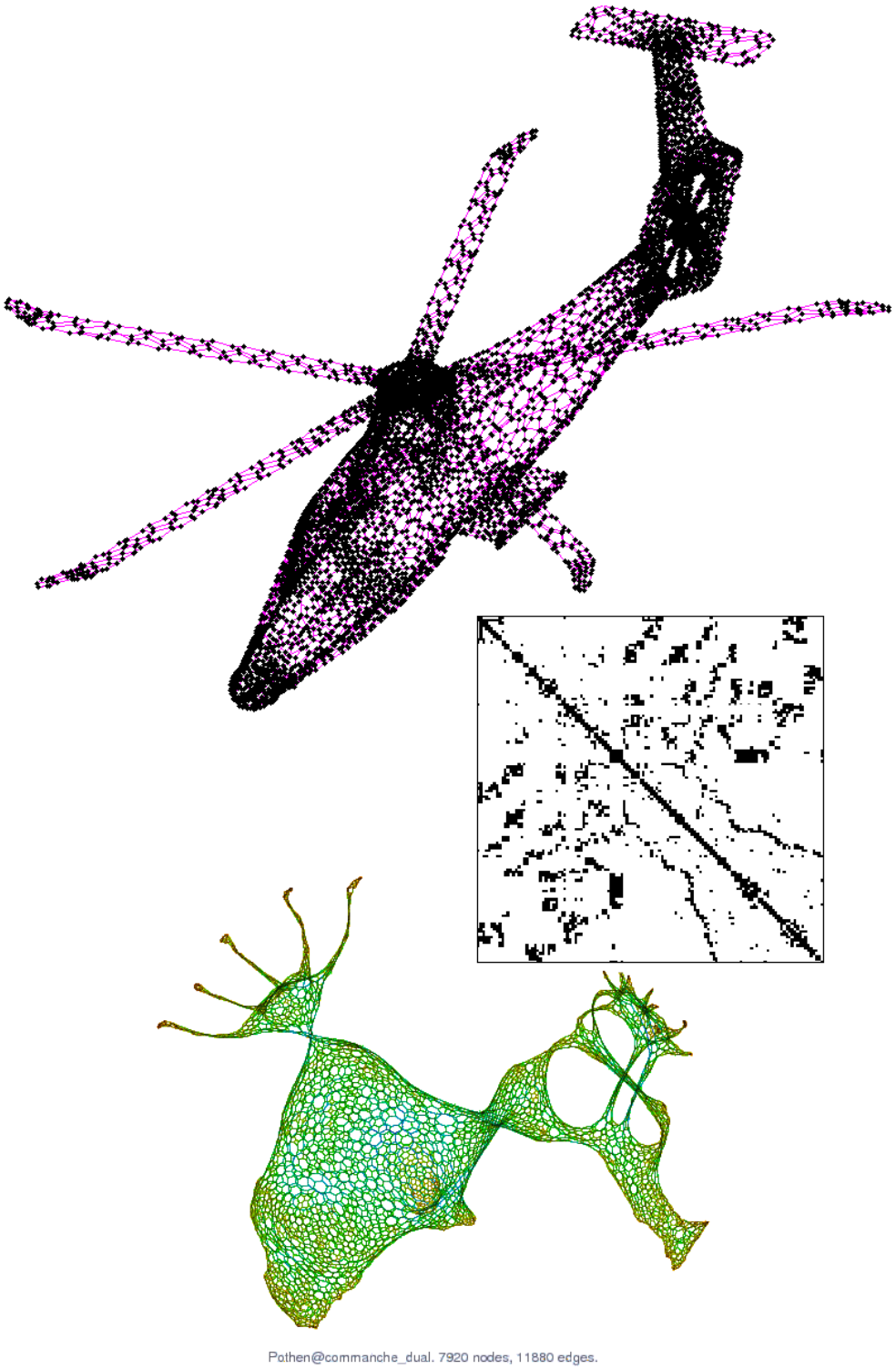
Násobení maticí může znázorňovat nějakou transformaci jednoho stavu do druhého. Největší uplatnění řídkých matic je tedy v simulaci jevů, například fyzikálních, biologických, chemických, ekonomických a dalších.

Jako příklad simulace můžeme uvést metodu konečných prvků [15][5].

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 obzvlášť hodí pro real-time aplikace.

1.3 Matice

Matice **A** typu (m, n) je mn uspořádaných prvků 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



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

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).

1.4 Vektor

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

1.5 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).

1.6 Složitosti

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

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

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

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

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

má asymptotickou 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 $t(n) = \Theta(f(n))$.

1.7 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).

Kapitola 2

Algoritmy násobení matic

V této kapitole představíme některé základní a pokročilejší algoritmy pro násobení matic. Základní algoritmy mají stejnou asymptotickou složitost $\mathcal{O}(n^3)$ 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 konstanty.

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

```
1:  procedure MMM-DEFINITION(A, B, C)                ▷ A,B,C jsou matice
2:      for row ← 0 to A.height do                    ▷ řádky
3:          for col ← 0 to B.width do                 ▷ sloupce
4:              sum ← 0;
5:              for i ← 0 to A.height do
6:                  sum += A[row][i] * B[i][col];
7:              end for
8:              C[row][col] ← sum;
9:          end for
10:     end for
11: end procedure
```

Z pseudokódu je vidět, že ve dvou for cyklech provádíme N násobení a N sčítaní. Asymptotická složitost je tedy $\mathcal{O}(n^2(n+n)) = \mathcal{O}(2n^3)$. 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

```
1:  procedure MMM-TRANSPOSE(A, B, C)                ▷ A,B,C jsou matice
2:      B ← transpose(B)
3:      for rowA ← 0 to A.height do                          ▷ řádky
4:          for rowB ← 0 to B.height do                      ▷ sloupce
5:              sum ← 0;
6:              for i ← 0 to A.height do
7:                  sum += A[rowA][i] * B[i][rowB];
8:              end for
9:              C[rowA][rowB] ← sum;
10:         end for
11:     end for
12: 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 náhodných 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

```
1: procedure MMM-BY-ROWS(A, B, C)                    ▷ A,B,C jsou matice
2:     for r ← 0 to A.height do                      ▷ řádky matice A i B
3:         for cA ← 0 to A.width do                  ▷ sloupce matice A
4:             for c ← 0 to B.width do               ▷ sloupce matice B
5:                 C[r][cA] += A[r][cA] * B[r][cB];
6:             end for
7:         end for
8:     end for
9: 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

```
1: procedure MMM-RECURSIVE(A, B, C, ay, ax, by, bx, cy, cx, n)
2:     if n = 1 then
3:         C[cy][cx] ← C[cy][cx] + A[ay][ax] · B[by][bx];
4:         return;
5:     end if
6:     for all r ∈ {0, n/2} do
7:         for all c ∈ {0, n/2} do
8:             for all i ∈ {0, n/2} do
9:                 MMM-recursive(A, B, C, ay + i, ax + r, by + c, bx + i, cy +
   c, cx + r, n/2);
10:            end for
11:        end for
12:    end for
13: 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.

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

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

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

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

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 mistrovské metody je `MMM-recursive`(n) = $\mathcal{O}(n^3)$.

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.

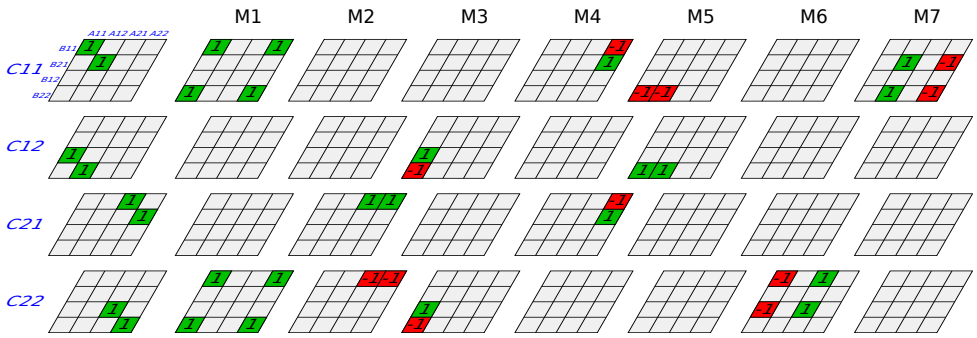
2.5 Strassenův algoritmus

V roce 1969 Volker Strassen v časopise Numerische Mathematik publikoval článek [22], ve kterém jako první představil algoritmus násobení dvou matic s menší asymptotickou složitostí než algoritmus podle definice, tedy $O(n^3)$.

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

Volker Strassen tedy využil jisté symetrie [17] 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 seskládány. V ilustračních maticích o velikosti čtyři ukazujeme, které sčítance pomocná matice do výsledku přičítá a které odečítá.



Obrázek 2.1: Strassen (TODO: [predelat na vlastni, cernobilou verzi])

Zápis Strassenova algoritmu vypadá následovně:

$$
\begin{aligned}
A \cdot B &= \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \\
M_1 &= (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2}) \cdot B_{1,1} \\
M_3 &= A_{1,1} \cdot (B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2} \cdot (B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2}) \cdot B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}) \\
C &= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix}
\end{aligned} \tag{2.2}
$$

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`. Parametry obou funkcí jsou: `offset-*`$(A, B, C, ay, ax, by, bx, cy, cx, n)$.

Algorithm 5 Strassenův algoritmus

```
1: procedure MMM-strassen(A, B, C, ay, ax, by, bx, cy, cx, n)
2:     if n = 1 then
3:         C[cy][cx] ← C[cy][cx] + A[ay][ax] · B[by][bx];
4:         return;
5:     end if
6:     h ← n/2;                                              ▷ čtvrtina
7:     m[9] ←init-matrices(9, h);                  ▷ devět pomocných matic
8:     offset-add(a, a, m[8], ay, ax, ay + h, ax + h, 0, 0, h);        ▷ M1
9:     offset-add(b, b, m[9], by, bx, by + h, bx + h, 0, 0, h);
10:    MMM-strassen(m[8], m[9], m[1], 0, 0, 0, 0, 0, 0, h);
11:    offset-add(a, a, m[8], ay + h, ax, ay + h, ax + h, 0, 0, h);    ▷ M2
12:    MMM-strassen(m[8], b, m[2], 0, 0, bx, by, 0, 0, h);
13:    offset-sub(b, b, m[8], by, bx + h, by + h, bx + h, 0, 0, h);    ▷ M3
14:    MMM-strassen(a, m[8], m[3], ay, ax, 0, 0, 0, 0, h);
15:    offset-sub(b, b, m[8], by + h, bx, by, bx, 0, 0, h);            ▷ M4
16:    MMM-strassen(a, m[8], m[4], ay + h, ax + h, 0, 0, 0, 0, h);
17:    offset-add(a, a, m[8], ay, ax, ay, ax + h, 0, 0, h);            ▷ M5
18:    MMM-strassen(m[8], b, m[5], 0, 0, by + h, bx + h, 0, 0, h);
19:    offset-sub(a, a, m[8], ay + h, ax, ay, ax, 0, 0, h);            ▷ M6
20:    offset-add(b, b, m[9], by, bx, by, bx + h, 0, 0, h);
21:    MMM-strassen(m[8], m[9], m[6], 0, 0, 0, 0, 0, 0, h);
22:    offset-sub(a, a, m[8], ay, ax + h, ay + h, ax + h, 0, 0, h);    ▷ M7
23:    offset-add(b, b, m[9], by + h, bx, by + h, bx + h, 0, 0, h);
24:    MMM-strassen(m[8], m[9], m[7], 0, 0, 0, 0, 0, 0, h);
25:    offset-add(m[1], m[4], m[8], 0, 0, 0, 0, 0, 0, h);              ▷ c1,1
26:    offset-sub(m[8], m[5], m[8], 0, 0, 0, 0, 0, 0, h);
27:    offset-add(m[8], m[7], c, 0, 0, 0, 0, cy, cx, h);
28:    offset-add(m[3], m[5], c, 0, 0, 0, 0, cy, cx + h, h);           ▷ c1,2
29:    offset-add(m[2], m[4], c, 0, 0, 0, 0, cy + h, cx, h);           ▷ c2,1
30:    offset-sub(m[1], m[2], m[8], 0, 0, 0, 0, 0, 0, h);              ▷ c2,2
31:    offset-add(m[8], m[3], m[8], 0, 0, 0, 0, 0, 0, h);
32:    offset-add(m[8], m[6], c, 0, 0, 0, 0, cy + h, cx + h, h);
33: 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) = $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.8})$.

Stejně jako v předešlém algoritmu, 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 následovně:

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

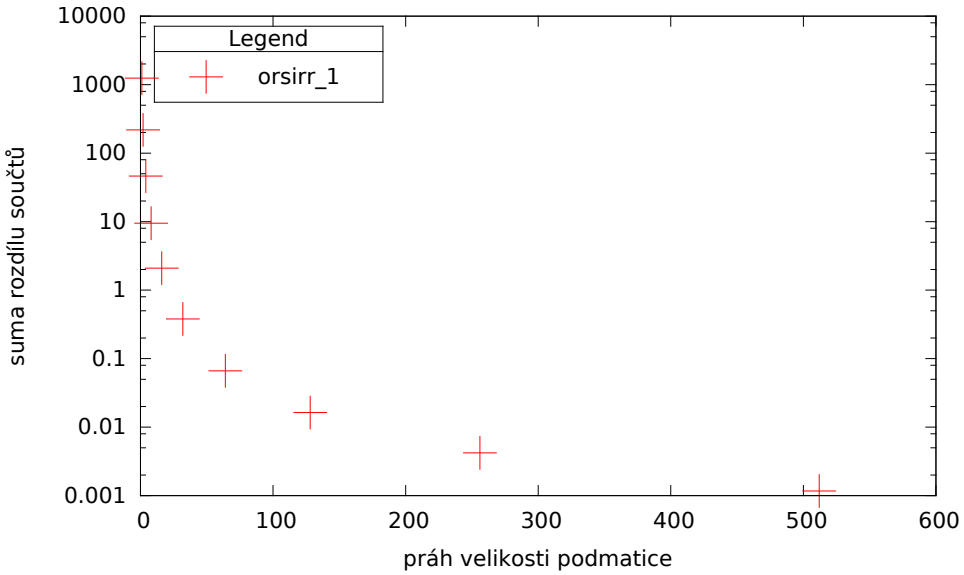
Správný výsledek je $30.234 \times 0.8912 + 0.5678 \times 0.7891 = 26.9445408 + 0.44805098 = 27.39259178$.

Nyní výpočet provedeme pomocí Strassenova algoritmu:

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

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 přesnosti, tedy v datovém typu `double` jazyka C99. Z grafu 2.2 je vidět exponenciální 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 [6]. Pro sedm operací násobení je možné snížit počet sčítání a odečítání. Pro jednoduchost zde uvádíme pouze originální algoritmus.

2.6 Rychlé algoritmy

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

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 [28][27], 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 [21], 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:

$$\sum_{i=1}^{n} nnzr_{A,i} nnzc_{B,i} \tag{2.3}$$

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.

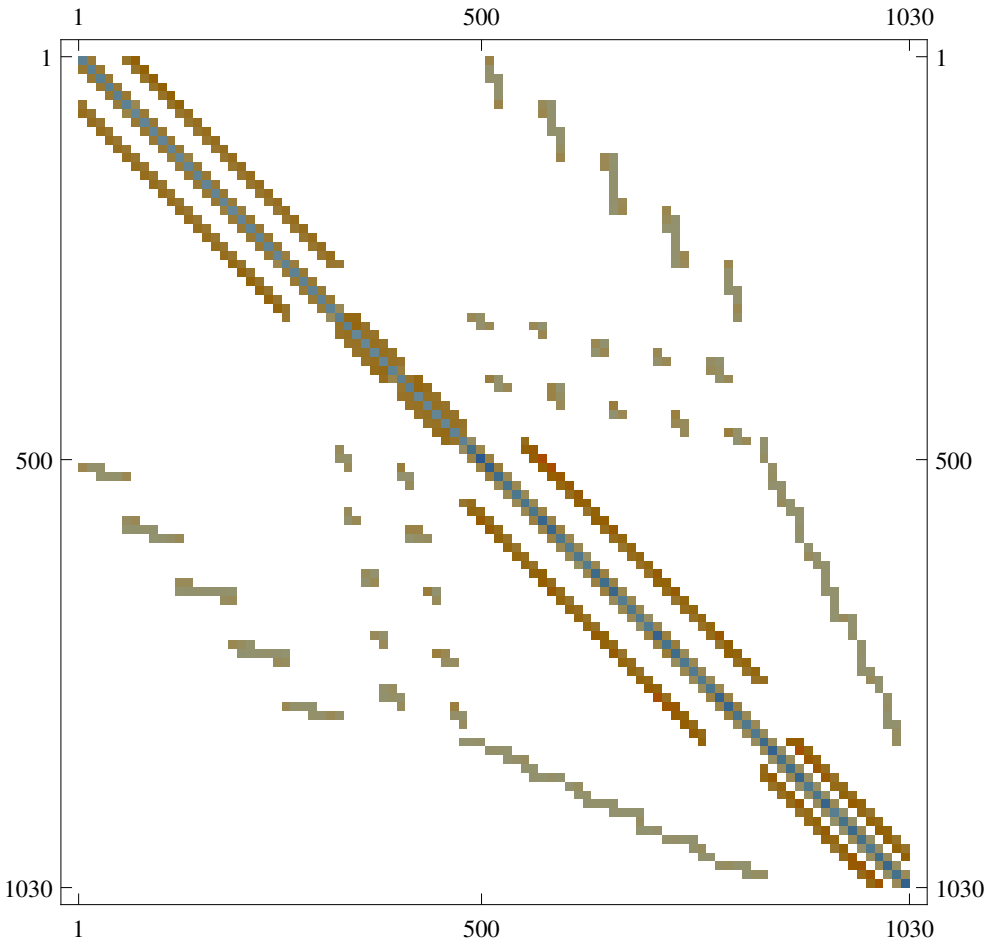
Pro reálnou představu demonstrujeme násobení dvou stejných matic. Opět se jedná o dvě matice orsirr_1. Matice orsirr_1 má velikost $n = 1030$ a $m = 6858$ nnz. Rozmístění nnz před respektive po vynásobení ukazují obrázky 2.3 respektive 2.4.

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 $\omega \approx 2.807354$ od $nnz < n^{1.8}$ a algoritmus Virginie Williamsové $\omega < 2.3727$ od $nnz < n^{1.37}$ jsou asymptoticky stejně rychlé jako algoritmus podle definice upravený pro řídké matice $\mathcal{O}(mn)$. 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.

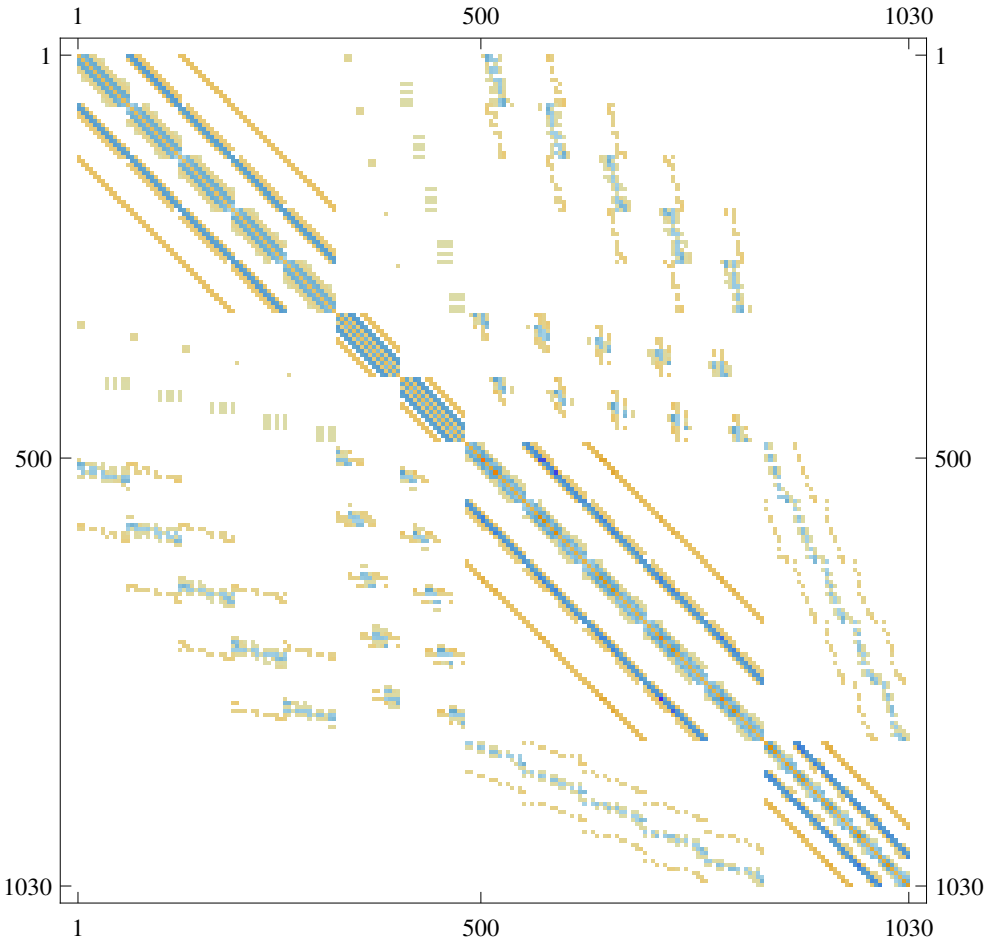
Raphael Yuster a Uri Zwick ukázali algoritmus [32] s asymptotickou složitostí $\mathcal{O}(m^{0.7}n^{1.2} + n^{2+o(1))}$, který rozdělí permutace řádků a sloupců na řídké a husté. Řídké permutace násobí algoritmem podle definice v úpravě pro řídké matice $\mathcal{O}(mn)$. Husté permutace vynásobí v té době nejznámějším nejrychlejším algoritmem a to od Dona Coppersmitha a Shmuela Winograda s asymptotickou složitostí $\omega = 2.375477$ [9].



Obrázek 2.3: Matice orsirr_1 před vynásobením

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 [25].



Obrázek 2.4: Matice orsirr_1 po vynásobením sama se sebou

Kapitola 3

Formáty uložení řídkých matic

Formáty uložení řídkých matic obecně ukládají jednotlivé prvky zvlášť a tedy nemusí ukládat ty nulové. To ale přináší řadu nevýhod. Za prvé se musí ukládat informace o souřadnicích. Za druhé, ztrácíme možnost přístupu k prvku na libovolných souřadnicích 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 mnoho formátů, jak řídkou matici efektivně uložit a pracovat s ní. 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 umožňujícími přidávání nebo odebírání prvků nebudou součástí práce. Stejně tak při násobení matice A vektorem B je výsledek C vektor.

Dalším kritériem pro rozdělení formátů je uspořádanost nenulových prvků v řídké matici. Pro uspořádané matice bude efektivnější takový formát, který využije určitý vzor, v jakým jsou prvky matice uloženy. 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. Pole `v` s hodnotami prvků, pole `r` s `y` souřadnicemi a pole `c` s `x` souřadnicemi.

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

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

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 redundantní.

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

```
1: procedure COO-MVM(COO,V,C)
2:     for i←0 to COO.nnz do
3:         V.v[COO.r[i]] += COO.v[i] * V.v[COO.c[i]];
4:     end for
5: 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 procházení matice se tak v poli s prvky mezi indexy $row\_pointers[i]$ a $row\_pointers[i+1]$ nachází prvky na řádku i. Proto je pole právě o jedna delší než výška matice, abychom mohli určit konec posledního řádku.

Algorithm 7 Násobení dvou COO matic

```
1: procedure COO-MMM(A,B,C)
2:     arp ← ComputeRowBeginnings(A, A.nnz + 1);
3:     brp ← ComputeRowBeginnings(B, B.nnz + 1);
4:     for i ← 0 to A.height do                          ▷ násobení
5:         for ac←arp[i] to arp[i + 1] do
6:             for bc←brp[A.c[ac]] to brp[A.c[ac] + 1] do
7:                 C.v[r][A.c[bc]] += A.v[ac] * B.v[bc];
8:             end for
9:         end for
10:    end for
11: end procedure
```

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

3.1.1 Formát MatrixMarket

MatrixMarket [4] je internetová sada matic s vlastním formátem pro uložení řídkých matic `.mtx`. Tato sada obsahuje skoro pět set matic z různých oblastí. Obsahuje i generátory řídkých matic, jejichž výstupy jsou matice různých vlastností.

Formát MatrixMarket je velice podobný formátu COO. Navíc umožňuje nastavit matici jako symetrickou a uložit pouze její polovinu.

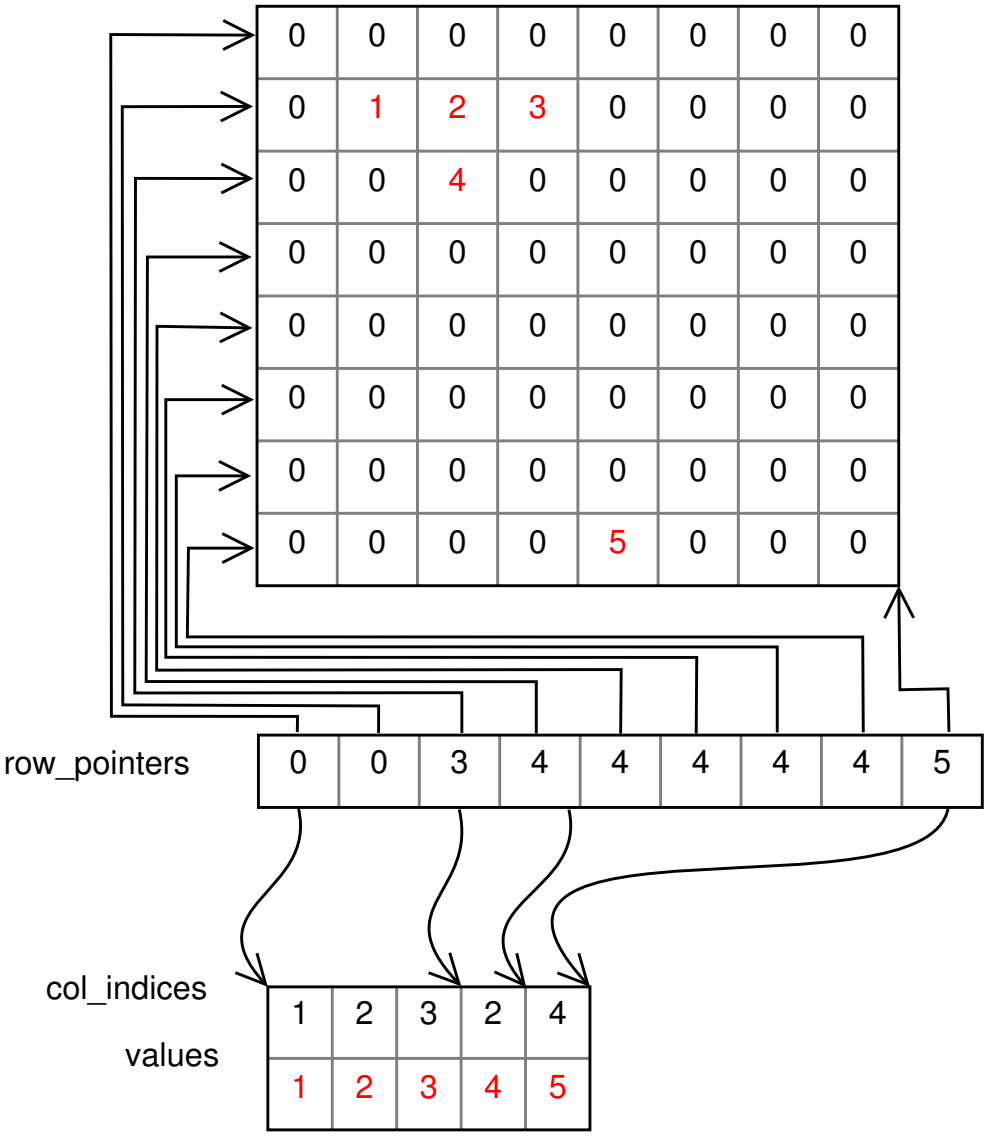
3.2 CSR - Compressed sparse row

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 to, co jsme si předpočítali v algoritmu 7. K poli s hodnotami je další pole *col_indices*, přiřazující ke každému prvku informaci o sloupci.

Jak je vidět z ilustrace 3.1, řádek s více prvky je uložen efektivně. Kvůli mnoho prázdným řádkům se ale pole *row_pointers* nezdá 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.

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.



Obrázek 3.1: Matice uložená ve formátu CSR

Pro uložení matice ve formátu CSR potřebujeme pole o velikost $n + 1$ pro uložení informací o řádcích a $2 * nnz$ pro prvky a jejich sloupce. Paměťová složitost formátu CSR je $\mathcal{O}(n + 1 + 2 * nnz)$.

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

Algorithm 8 Násobení matice CSR s vektorem

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

Algorithm 9 Násobení dvou CSR matic

```
1: procedure CSR-MMM(A,B,C)
2:     for i ← 0 to A.height do                          ▷ násobení
3:         for ac ← A.rp[i] to A.rp[i + 1] do
4:             for bc ← B.rp[A.ci[ac]] to B.rp[A.ci[ac] + 1] do
5:                 C.v[r][B.ci[bc]] += A.v[ac] * B.v[bc];
6:             end for
7:         end for
8:     end for
9: end procedure
```

3.3 BSR - Block Sparse Row

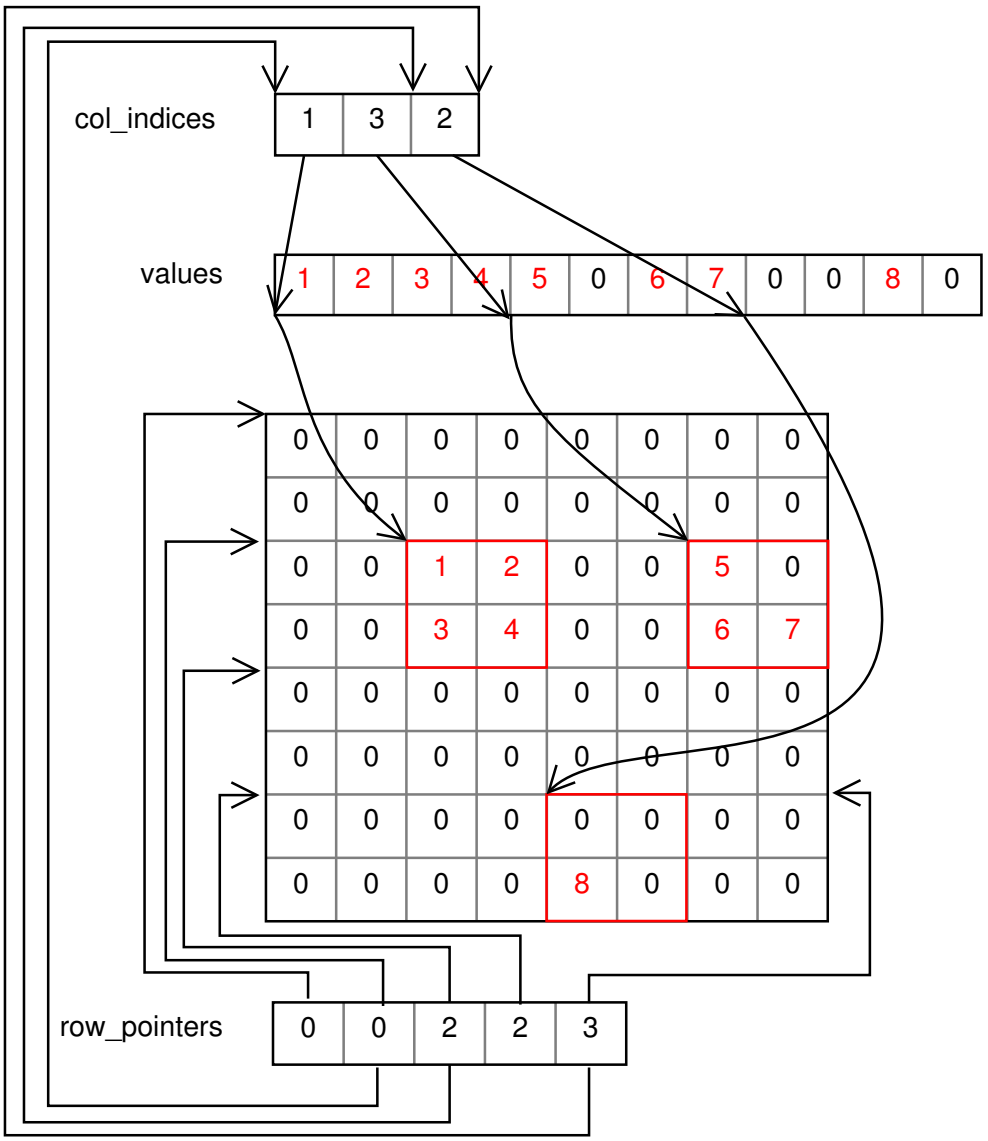
Jako formát CSR využívá uložení prvků v řádku, formát BSR[7][18] ještě navíc detekuje a ukládá prvky v blocích.

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

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

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



Obrázek 3.2: Matice uložená ve formátu BSR

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[29].

Algorithm 10 Násobení matice BSR s vektorem

```
1: procedure BSR-MVM(A,B,C)
2:     bs ← A.block_size;
3:     for i ← 0 to A.height / bs do
4:         for ac ← A.rp[i] to A.rp[i + 1] do
5:             for l ← 0  to A.bs do                    ▷ násobení bloku
6:                 for m ← 0 to bs do
7:                     C.v[(i * bs) + l] += A.v[ac * (bs * bs) + (l *
   bs) + m] * B.v[A.ci[ac] * bs + m];
8:                 end for
9:             end for
10:        end for
11:    end for
12: end procedure
```

Algorithm 11 Násobení dvou BSR matic

```
1: procedure BSR-MMM(A,B,C)
2:     bs ← A.block_size;
3:     for i ← 0 to A.height / bs do
4:         for ac ← A.rp[i] to A.rp[i+1] do
5:             for bc ← B.rp[A.ci[ac]] to B.rp[A.ci[ac]+1] do
6:                 for l ← 0 to A.bs do                 ▷ násobení bloku
7:                     for m ← 0 to bs do
8:                         for n ← 0 to bs do
9:                             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];
10:                        end for
11:                    end for
12:                end for
13:            end for
14:        end for
15:    end for
16: end procedure
```

Paměťová složitost je závislá na počtu bloků b, velikosti bloků $block\_size$ a počtu řádek matice n. Pole $row\_pointers$ má velikost $n+1$, pole $col\_indices$ má velikost b a pole hodnot $values$ má velikost $b * sm\_size^2$. Celková velikost matice uložené ve formátu BSR je $\mathcal{O}(n + b + b * sm\_size^2)$.

3.4 Quadtree

Předchozí popsáné formáty uložení řídkých matic jsou velmi přímočaré. Neumožňují rekurzivní přístup, který potřebný například pro Strassenův algoritmus. Tento problém řeší formát Quadtree [20][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, E. Pokud obsahuje nějaké prvky, je list označen jako hustý, D. Vnitřní uzly jsou označeny jako smíšené, M.

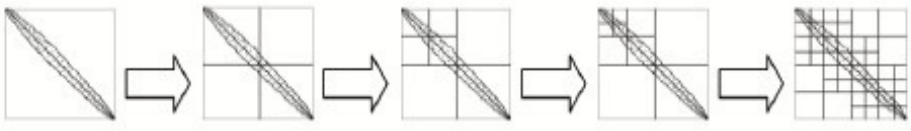


Figure 1. Recursive division on matrix $BCSSTK15$(Model of an offshore platform).

Obrázek 3.3: Rozdělení matice

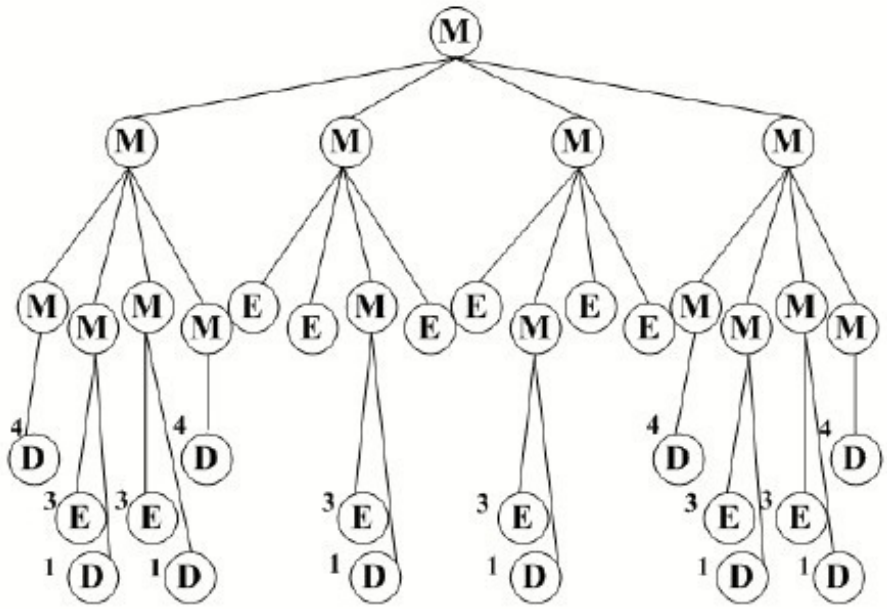


Figure 2. Abstract the divided matrix in Figure 1 into a quadtree based storage format.

Obrázek 3.4: Strom matice uložené ve formátu Quadtree

Výška stromu Quadtree je při reprezentaci matice o velikosti n a při velikosti podmatice sm_size:

$$\left\lceil \log_4 \frac{n^2}{sm\_size^2} \right\rceil \tag{3.1}$$

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

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

3.5 Modifikace formátu quadtree

V této práci navrhujeme místo 4-árního stromu obecný k-ární strom. Výška stromu se tím sníží. Pro předchozí příklad 3.4 by tedy pro 16-ární strom byla

výška pouze 4. 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 $KAT.n = 2$. Pro snadnější rekurzivní přístup budeme jako $KAT.n$ uvažovat pouze mocniny dvou.

Výška stromu pro KAT matici označíme jako `KAT.height` a vypočítáme ji jako výšku Quadtree, jen s parametrem k:

$$KAT.height = \left\lceil \log_k \frac{n^2}{sm\_size^2} \right\rceil \tag{3.2}$$

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

3.5.1 Typy listů

KAT matice obsahuje 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.

[TODO: ilustrace kat matrix]

3.5.2 Násobení

Pro násobení budeme používat algoritmus rekurzivního násobení popsaného v 2.4. Popsaný algoritmus 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:

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 asymptotická složitost násobení dvou KAT matic $\mathcal{O}((\frac{n^2}{sm\_size} * (height + sm\_size^3))$.

3.5.3 Paměťová složitost

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

Algorithm 12 Násobení matice KAT s vektorem

```
1: procedure KAT-MVM(KAT, KAT_node, VB, VC)
2:     for i ← 0 to KAT.n do
3:         for j ← 0 to KAT.n do
4:             if KAT_node.childs[i][j] ≠ NIL then
5:                 if KAT_node.childs[i][j].type = "submatrix" then
6:                     multiplyNode(KAT_node.childs[i][j], VB, VC);
7:                     coninue;
8:                 end if
9:                 if KAT_node.childs[i][j].type = "inner" then
10:                    KAT-MVM(KAT,KAT_node.childs[i][j], VB, VC);
11:                    coninue;
12:                end if
13:            end if
14:        end for
15:    end for
16: end procedure
```

Algorithm 13 Násobení dvou KAT matic

```
1: procedure KAT-MMM(KATa, KATa_node, KATb, KATb_node, C)
2:     for i ← 0 to KAT.n do
3:         for j ← 0 to KAT.n do
4:             for k ← 0 to KAT.n do
5:                 if KATa_node.childs[i][k] ≠ NIL and
   KATb_node.childs[k][j] ≠ NIL then
6:                     if KAT_node.childs[i][j].type = "submatrix" then
7:                         multiplyNode(KAT_node.childs[i][j],
   KAT_node.childs[i][j], VC);
8:                         coninue;
9:                     end if
10:                    if KAT_node.childs[i][k].type = "inner" then
11:                        KAT-MMM(KATa,KATa_node.childs[i][k],
   KATb,KATb_node.childs[k][j], C);
12:                        coninue;
13:                    end if
14:                end if
15:            end for
16:        end for
17:    end for
18: end procedure
```

Kapitola 4

Analýza a návrh

V této kapitole představíme některé známé systémy, které umí pracovat s řídkými maticemi. Popíšeme možnosti implementace 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. Některé nástroje jsou ale specificky optimalizované a i přes jejich obecnost mají dobrou výkonnost. Maximální efektivnosti dosáhneme, pokud co nejvíce využijeme hardware a náš program bude řešit konkrétní úkol.

4.1 Software pro práci s řídkými maticemi

4.1.1 SciPy.sparse

SciPy je knihovna skriptovacího jazyka Python [16]. 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++.

4.1.2 Boost

Pro práci s řídkými maticemi v jazyce C++ je možné použít knihovnu uBLAS[26] ze sady knihoven Boost[14]. Knihovna uBLAS obsahuje algoritmy pro řešení základních ukolů z lineálrní algebry. Protože je kód napsaný pomocí C++ šablon, je vygenerovaný kód výkonný.

4.1.3 ATLAS

ATLAS [23] je v překladu zkratkou pro automatické generování vylazeného kódu pro software řešící úlohy z lineární 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 [2], nebo pro násobení matice s maticí je to například PHiPAC [24][3].

4.1.4 Wolfram Mathematica

Wolfram Mathematica [30] 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` 3.1.1. 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", %]` [31].

4.2 Řešení implementace

Jedna z možných variant pro porovnání všech zvolených formátů byla doimplementovat formát Quadtree, respektive KAT do knihovny SciPy.sparse. Další možnost byla pokračovat v semestrální práci z předmětu Efektivní implementace algoritmů s tématem násobení matic ve formátu CSR.

Protože se v této práci zabýváme pouze vlivem uložení řídké matice pro operaci násobení, rozhodli jsme se vytvořit program pouze 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. Navíc pokud nebudou použity žádné jiné implementace, budou všechny algoritmy stejně optimalizované. Navážeme tedy na semestrální práci. Pro každý formát vytvoříme ve zdrojovém kódu třídu, v níž implementujeme operaci násobení s maticí a násobení s vektorem.

Kapitola 5

Realizace

5.1 Prostředí

Popsané algoritmy jsme implementovali v jazyce C99[todo: zdroj]. Implementace probíhala v operačním systému Lubuntu[todo: zdroj], v IDE Eclipse CDT[todo: zdroj]. Zdrojový kód byl kompilován pomocí překladače GNU C[todo: zdroj], s optimalizačním přepínačem `-Ofast`.

K lazení chyb pro práci s pamětí jsme použili nástroj Valgrind[todo: zdroj] s přepínači `--leak-check=full --show-reachable=yes`. Běžící program jsme krokovali pomocí nástroje GNU Debugger[todo: zdroj]. Zdrojový kód pro lazení chyb byl kompilován s přepínači `-Wall -pedantic -Og -ggdb`.

5.2 Design implementace

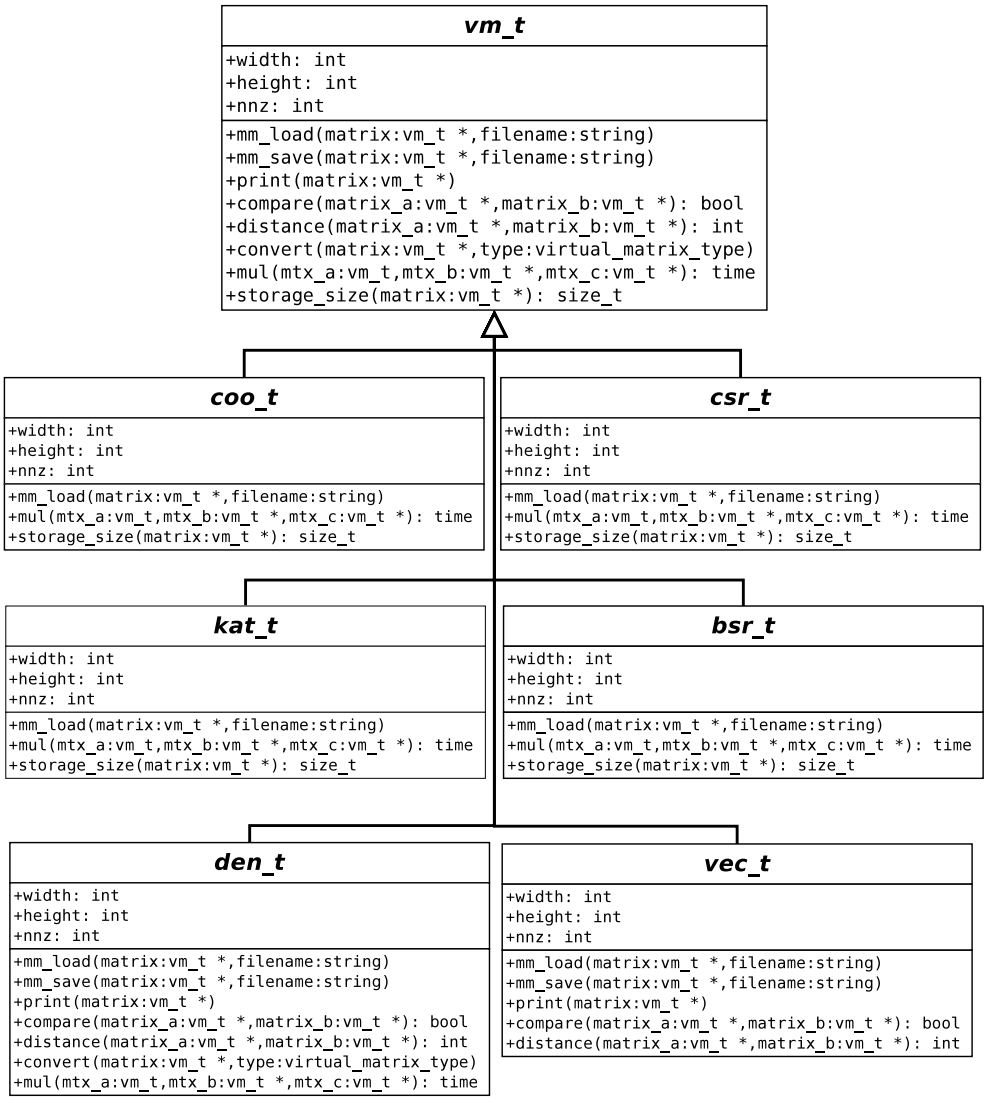
Po různých formátech požadujeme stejné operace, proto v programu zavádíme abstraktní matici a jednoduchý objektově orientovaný model [19], kde formáty dědí z virtuální matice. Obrázek 5.1 ukazuje jednoduchý návrh tříd v UML.

Program běží v příkazovém řádku bez GUI. Program načte jednu, nebo dvě matice v určitém formátu a vynásobí je. Pomocí přepínačů lze zobrazit, nebo uložit výslednou matici, popřípadě vypsat některé údaje o načtených maticích. Podrobnější popis nastavení programu je v příloze.

5.3 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 vnitřních 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



Obrázek 5.1: UML diagram tříd programu

konstanty, rozbalí. Překladači i explicitně sdělujeme, ať cykly rozbalí přes atribut `__attribute__((optimize("unroll-loops")))`.

V bakalářské práci Rozšíření implementace formátu kvadrantového stromu od Tomáše Karabely[todo: zdroj] 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`

5.3.1 Tvoření stromu

Při tvoření matice v KAT formátu, pro každý prvek procházíme strom a hledáme správný list. Protože výška stromu je daná třemi parametry, tedy velikostí matice n, velikostí podmatice sm_size a počtu větvení uzlu k, nejefektivnější využití bude, pokud je n bezezbytku dělitelné $k * sm\_size$. Pokud toto neplatí, rodiče listů budou mít část synů nevyužitých i při uložení husté matice. Protože se prvky do matice načítají po řádcích zleva doprava, je velká pravděpodobnost, že další načtený prvek bude patřit do stejného listu. Z tohoto důvodu si pamatujeme poslední list a pokud prvek patří do něj, vrátíme jej. Algoritmus 14 ukazuje, jakým způsobem vyhledáváme list pro prvek.

5.3.2 Násobení listů matice KAT

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 2, ukážeme zde pouze násobení CSR listu s hustým listem.

5.4 Testování

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

Pro přehled o nefungujících konfiguracích jsme použili testovací framework Cassertion [todo: zdroj].

Při testování jsme i na relativně malých maticích narazili na problém s numerickou stabilitou. Část z testovacích matic má velký desetinný rozvoj a protože čísla jsou v různých formátech násobena v jiném pořadí, dochází k

Algorithm 14 Vyhledání listu pro KAT matici

```
1: procedure KAT-getNode(KAT, y, x)
2:     if {y, x} ∈ lastLeaf.area then
3:         return lastLeaf;
4:     end if
5:     tmpNode ← KAT.root;
6:     blockY ← 0;                                          ▷ výřez matice
7:     blockX ← 0;
8:     blockS ← KAT.sm_size;
9:         ▷ až do předposledního vnitřního uzlu traverzujeme podle velikosti
   KAT.n
10:    while blockS > (KAT.n * KAT.sm_size) do
11:        nodeY ← (y - blockY) / (blockS / KAT.n);
12:        nodeX ← (x - blockX) / (blockS / KAT.n);
13:        tmpNode ← tmpNode.childs[nodeY][nodeX];
14:        blockY += nodeY * (blockS / KAT.n);
15:        blockX += nodeX * (blockS / KAT.n);
16:        blockS /= KAT.n;
17:    end while
18:            ▷ do posledního vnitnřního uzlu traverzujeme podle velikosti
   KAT.sm_size
19:    nodeY ← (y - blockY) / KAT.sm_size;
20:    nodeX ← (x - blockX) / KAT.sm_size;
21:    tmpNode ← tmpNode.childs[nodeY][nodeX];
22:                                              ▷ nyní je tmpNode list
23:    tmpNode.y ← ⌊y / KAT.sm_size⌋ * KAT.sm_size ;
24:    tmpNode.x ← ⌊x / KAT.sm_size⌋ * KAT.sm_size ;
25:    lastLeaf ← tmpNode;
26:    return tmpNode;
27: end procedure
```

rozdílu mezi výsledky. Při kompilaci programu lze nastavit přesnost výpočtu pomocí typu proměnné pro uchovaní hodnot buď na `float`, `double` nebo `long double`. Při porovnávání dvou matic neporovnáváme hodnoty přímo, ale sledujeme velikost jejich rozdíl. V defaultní konfiguraci je povolená odchylka 0.001, při kompilaci lze změnit. Pokud spustíme testy pouze s přesností `float`, část testů selže právě kvůli numerické stabilitě.

5.5 Práce s formátem MatrixMarket

Pro náš program jsme zvolili právě formát `.mtx` 3.1.1. Podporujeme více druhů tohoto formátu. Nesymetrický, s banerem `%%MatrixMarket matrix`

Algorithm 15 Násobení hustého KAT listu s CSR listem

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

Algorithm 16 Testování

```
1: procedure TestFormats(PairList, FormatList)
2:     for all pair ∈ PairList do
3:         denseA ← vm_load(pair.a, DENSE);
4:         denseB ← vm_load(pair.b, DENSE);
5:         denseC ← vm_mul(denseA, denseB, denseC);
6:         for all format ∈ FormatList do
7:             sparseA ← vm_load(pair.a, format);
8:             sparseB ← vm_load(pair.b, format);
9:             sparseC ← vm_mul(sparseA, sparseB, sparseC);
10:            if vm_compare(denseC, sparseC) = NOT_SAME then
11:                print("Error:", pair.a, pair.b, format);
12:            end if
13:        end for
14:    end for
15: end procedure
```

`coordinate real symmetric` a symetrický s banerem `%%MatrixMarket matrix coordinate real general`. Místo reálných čísel podporujeme i druh `pattern`, kdy nenulové prvky nabývají pouze hodnoty jedna.

Přestože náš program neumí pracovat s komprimovanými soubory, v prostředí unixového systému předáváme programu pojmenovanou rouru do které komprimovanou matici rozbalujeme příkazem `<(gzip -cd matrix.mtx.gz)` .

5.5.1 Generátor řídkých matic

Generátory z MatrixMarketu běží v internetovém prohlížeči a v jazyce Java. Protože takto není jednoduché matice generovat ve skriptu, abychom při distribuci našeho programu nemuseli přikládat velké testovací matice, implemen-

tovali jsme jednoduchý generátor řídkých matic. Parametry předáváme programu informace o výsledné matici a seznam objektů, tedy buď podmatic nebo diagonál, které mají být do matice zahrnuty. Manuál ke generátoru je možné vypsat zavoláním `./tests/bin/matrix_generator -h`.

Algorithm 17 Generování řídkých matic

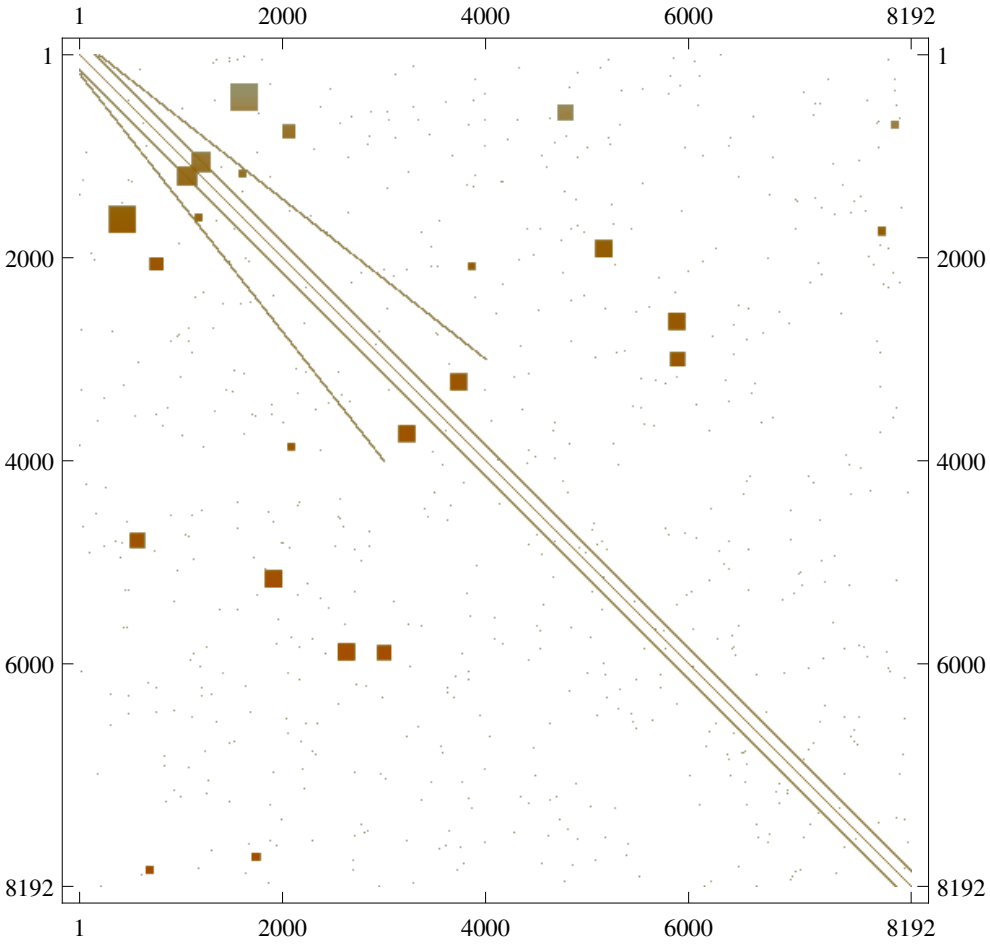
```
1: procedure SparseMatrixGenerator(file, width, height, ItemList)
2:     MtxWrapper ← InitMtxWrapper();
3:     MtxWrapper.PositionVector ← InitVector();
4:     for all Item ∈ ItemList do
5:         MtxWrapper.addItem(Item.y, Item.x, Item.properties);
6:         if Item.type == Mirrored then
7:             MtxWrapper.addItem(Item.x, Item.y, Item.properties);
8:         end if
9:     end for
10:    MtxWrapper.PositionVector.sort();
11:    MtxWrapper.PositionVector.removeDuplicates();
12:    MtxWrapper.write(file);
13: end procedure
```

5.6 Měření

Měření probíhalo na serveru `star.fit.cvut.cz`.

TODO: popsat jak a kde to budu měřit, cas z omp a cachegrind/callgrind/velikost struktur/pocet uzlu ve strome, zrychleni oproti csr*csr>den, mensi velikost oproti den

5.7 Zhodnocení



Obrázek 5.2: Matice vygenerovaná generátorem

Závěr

Závěr.

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

[3] 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.

[4] 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`

[5] 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`

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

[7] community, T. S.: scipy.sparse.bsr_matrix. 2014. Dostupné z: `http://docs.scipy.org/doc/scipy-0.13.0/reference/generated/scipy.sparse.bsr_matrix.html`

[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.: SPARSE MATRIX ALGORITHMS AND SOFTWARE. 2014. Dostupné z: `http://www.cise.ufl.edu/~davis/research.html`

[12] Davis, T.; Hu, Y.: Visualizing Sparse Matrices. 2014. Dostupné z: `http://www.cise.ufl.edu/research/sparse/matrices/synopsis/`

[13] 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`

[14] Dawes, B.; Abrahams, D.; Rivera, R.: Boost. 2007. Dostupné z: `http://www.boost.org/`

[15] 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.

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

[17] 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.

[18] Intel: Intel® Math Kernel Library 11.0.5 Reference Manual. 2014. Dostupné z: `https://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mklman/GUID-9FCEB1C4-670D-4738-81D2-F378013412B0.htm`

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

[20] 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.

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

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

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

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

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

[26] Walter, J.; Koch, M.; Winkler, G.; aj.: Basic Linear Algebra Library. 2010. Dostupné z: `http://www.boost.org/doc/libs/1_55_0/libs/numeric/ublas/doc/index.htm`

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

[28] 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.

[29] Wolf, M. E.; Lam, M. S.: A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, New York, NY, USA: ACM, 1991, ISBN 0-89791-428-7, s. 30–44, doi:10.1145/113445.113449. Dostupné z: `http://doi.acm.org/10.1145/113445.113449`

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

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

[32] 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

Seznam algoritmů

*

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