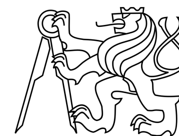
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	ešení problému diskrétního logaritmu použitím index calculu na GPU
Student:	Bc. Dominik Plíšek
Vedoucí:	Dr.-Ing. Martin Novotný
Studijní program:	Informatika
Studijní obor:	Po íta ová bezpe nost
Katedra:	Katedra po íta ových systém
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Seznamte se s algoritmem Index Calculus, který se používá pro ešení problému diskrétního logaritmu.

St žejním krokem Index Calculu je ešení rozsáhlých ídkých soustav lineárních rovnic v modulární aritmetice. Nastudujte z literatury vhodné paralelní algoritmy pro tuto úlohu.

Cílem práce je analýza vlastností t chto soustav, rozbor algoritm z hlediska jejich použití v kone ném t lese a pr zkum p evoditelnosti vybraného algoritmu na GPU. Je-li to možné, prove te alespo áste nou implementaci a m ení výkonnosti.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

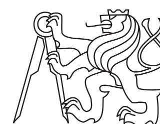
prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 28. ledna 2016

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra počítačových systémů



Diplomová práce

Řešení problému diskrétního logaritmu použitím index calculu na GPU

Bc. Dominik Plíšek

Vedoucí práce: Dr.-Ing. Martin Novotný

9. května 2017

Poděkování

Děkuji svému vedoucímu panu Dr.-Ing. Martinu Novotnému za jeho odborné rady i podporu. Děkuji panu Mgr. Michalu Kupsovi, Ph.D. a panu Ing. Ivu Petrovi, Ph.D. za konzultace v oblasti matematiky. Děkuji panu Dr. Timu Davisovi za jeho celoživotní výzkum v oblasti řídkých matic a GPU, který byl pro tuto práci klíčový. V neposlední řadě děkuji své ženě i dětem za trpělivost.

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 2017 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Dominik Plíšek. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Plíšek, Dominik. *Řešení problému diskrétního logaritmu použitím index calculu na GPU.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Práce se zabývá možností využít grafickou kartu k urychlení jednoho kroku index calculu, algoritmu pro hledání diskrétního logaritmu. V tomto kroku je řešena rozsáhlá řídká soustava lineárních kongruencí. Práce popisuje tři metody řešení řídkých lineárních soustav a zkoumá možnost jejich úpravy pro modulární aritmetiku a pro specifické vlastnosti problému. Následuje implementace multifrontální LU faktorizace, zčásti určená pro grafickou kartu, a její popis. Nakonec je změřen výkon oproti verzi bez použití grafické karty a je učiněn závěr, že tímto způsobem nelze dosáhnout velkého zrychlení.

Klíčová slova index calculus, problém diskrétního logaritmu, řídká matice, soustava lineárních kongruencí, LU faktorizace, UMFPACK, GPU

Abstract

This thesis attempts to use the GPU to speed up one step of Index Calculus, an algorithm for finding the discrete logarithm. The step consists of solving a large sparse system of linear congruences. The thesis describes three methods of solving sparse linear systems, investigating the viability of their modification for modular arithmetic and for the specific characteristics of the problem. An implementation and documentation of a multifrontal LU factorization follows, utilizing in part the GPU. Finally, performance measurements show that great speedup cannot be achieved using the GPU in this situation.

Keywords Index Calculus, discrete logarithm problem, sparse matrix, linear system of congruences, LU factorization, UMFPACK, GPU

Obsah

Úvod **1**

1 Index calculus **3**
1.1 Problém diskrétního logaritmu 3
1.2 Co je index calculus 4
1.3 Varianty 4
1.4 Algoritmus 5
1.5 Složitost 7
1.6 Neprvočíselný modul 9

2 Řešení řídkých soustav lineárních kongruencí **11**
2.1 Řídká matice 11
2.2 Datové struktury 12
2.3 Základní operace 14
2.4 Postup řešení soustavy 18
2.5 Specifika modulární aritmetiky 34

3 Vlastnosti soustav vznikajících v index calculu **37**
3.1 Co soustava vyjadřuje 37
3.2 Rozměry a hodnost matice soustavy 38
3.3 Statistické vlastnosti faktorizace polynomu 38
3.4 Příklad v $GF(2^{256})$ 41
3.5 Model pro náhodné generování soustav 42

4 Použitelnost algoritmů pro problém daných vlastností **43**
4.1 Choleského faktorizace 43
4.2 LU faktorizace 43
4.3 QR faktorizace 45

5 Realizace LU faktorizace **49**

5.1 Existující implementace 49
5.2 Zvolený postup 56
5.3 GPU kernel 57
5.4 Úprava UMFPACK pro LU faktorizaci 68
5.5 Implementace řešení soustavy 77
5.6 Testování 78

6 Měření 81
6.1 Parametry grafické karty 81
6.2 Parametry procesoru 82
6.3 Měření kernelu samostatně 83
6.4 Vstupní data 84
6.5 Metodika měření 84
6.6 Výsledky 88

Závěr 97

Literatura 99

A Seznam použitých zkratek 103

B Obsah přiloženého CD 105

Seznam obrázků

2.1 Reprezentace řídké matice pomocí spojového seznamu 15
2.2 Doplnění při řešení řídké dolní trojúhelníkové soustavy s řídkým vektorem pravé strany [1] 16
2.3 Ořez grafu L, vznik eliminačního stromu [1] 20
2.4 Příklad Choleského faktorizace a eliminačního stromu [1] 21
2.5 Výplň sloupce k za základě předchozích sloupců 24
2.6 Budoucí výplň dosud nefaktorizované části A na základě nenulového vzoru pivotního řádku a sloupce i 25
2.7 Multifrontální metoda faktorizace symetrické matice [1] 26
2.8 Multifrontální metoda faktorizace asymetrické matice [1] 27
2.9 Krok k QR faktorizace 29
2.10 Výplň A' v jednom kroku QR faktorizace 32
2.11 Aplikace předchozích Householderových reflexí na řádek k 33

4.1 Multifrontální kontribuční strom osmi frontálních matic a jeho faktorizace jednotlivými spouštěními GPU kernelu [2] 46
4.2 Faktorizace frontální matice [2] 47

5.1 Náčrtek dělení frontální matice 51
5.2 Vstupní a výstupní bloky GPU kernelu 59
5.3 Jeden krok paralelní husté LU faktorizace [3] 61
5.4 Pivotní blok ve sdílené paměti a vlákna zodpovědná za jeho hodnoty 62
5.5 Komunikace mezi vlákny v nejjemnějším řešení dolní trojúhelníkové soustavy 64
5.6 Ukázka cyklického mapování po řádcích s více vektory pravých stran 65
5.7 Ukázka mapování na transponované soustavě 65
5.8 Mapování vláken na segment C 69

6.1 Zrychlení faktorizace frontální matice pomocí GPU v závislosti na její velikosti 85

6.2 Zrychlení faktorizace frontální matice pomocí GPU v závislosti na jejím tvaru 86
6.3 Zrychlení faktorizace frontální matice pomocí GPU v závislosti na počtu pivotních řádků a sloupců 87
6.4 Zrychlení faktorizace frontální matice pomocí GPU v závislosti na počtu prvočíselných modulů – souběžně řešených úloh 87
6.5 Čas běhu algoritmu řešení soustavy v závislosti na velikosti tělesa . 91
6.6 Zrychlení algoritmu řešení soustavy pomocí GPU v závislosti na velikosti tělesa 91
6.7 Čas běhu algoritmu řešení soustavy v závislosti na počtu prvočíselných modulů – souběžně řešených úloh 92
6.8 Zrychlení algoritmu řešení soustavy pomocí GPU v závislosti na počtu prvočíselných modulů – souběžně řešených úloh 92
6.9 Čas běhu algoritmu řešení soustavy v závislosti na velikosti tělesa při rovnoměrném rozložení nenul 95
6.10 Zrychlení algoritmu řešení soustavy pomocí GPU v závislosti na velikosti tělesa při rovnoměrném rozložení nenul 95
6.11 Čas běhu algoritmu řešení soustavy v závislosti na počtu prvočíselných modulů – souběžně řešených úloh při rovnoměrném rozložení nenul 96
6.12 Zrychlení algoritmu řešení soustavy pomocí GPU v závislosti na počtu prvočíselných modulů – souběžně řešených úloh při rovnoměrném rozložení nenul 96

Seznam tabulek

6.1 Výsledky měření samostatného kernelu podle velikosti (ms) 85
6.2 Výsledky měření samostatného kernelu podle tvaru (ms) 85
6.3 Výsledky měření samostatného kernelu podle počtu pivotních řádků a sloupců (ms) 86
6.4 Výsledky měření samostatného kernelu podle počtu souběžně řešených úloh (ms) 86
6.5 Měření nad soustavami patřičných vlastností (ms) 90
6.6 Měření nad soustavami s rovnoměrně rozloženými nenulovými prvky (ms) 94

Úvod

Žijeme v digitální éře, čím dál větší část našich životů se přesouvá do kyberprostoru. Přes internet a digitální média komunikujeme, sdílíme své zážitky, obchodujeme, plánujeme, uzavíráme smlouvy, v kyberprostoru ukládáme důvěrný obsah, osobní údaje, kompromitující materiály, přístupová hesla, hledáme tu zábavu. Čím dál větší počet lidí tráví čím dál více času u počítače, aniž by mnohdy věděli, jak počítač funguje, jak funguje internet, jaké jsou hrozby internetu a jaká jsou nutná bezpečnostní opatření při přístupu k němu. Proto je čím dál důležitější dbát na bezpečnost digitálních údajů, a to nejen osobních, ale i firemních a státních.

Jedním ze základních kamenů digitální bezpečnosti je šifra, tedy úprava údaje tak, aby bylo možné jej rozluštit jen při znalosti určitého tajného klíče. Přestože je únik citlivého údaje mnohdy zapříčiněn lidským faktorem – prozrazením údaje nebo klíče, je třeba dbát na to, aby nebylo možné klíč odvodit na základě znalosti původního a šifrovaného údaje. Neoprávněné získání klíče by znamenalo možnost klíč použít k odhalení jiných údajů, případně vystupovat pod cizí identitou apod.

Šifrovacích algoritmů je celá řada a jsou založeny na různých matematických principech. Významnou roli v této oblasti hrají jednosměrné funkce. Takové, kde nalézt funkční hodnotu je snadné (na straně šifrujícího), zatímco nalézt vzor k funkční hodnotě je v rozumném čase nemožné (na straně útočníka).

Známou jednosměrnou funkcí je modulární umocňování. V některých konečných cyklických grupách platí, že nalézt exponent při známém základu (generátoru grupy) a výsledné hodnotě představuje velmi obtížný problém. Nazýváme jej problémem diskrétního logaritmu, neboť je hledán exponent. Na problému diskrétního logaritmu stojí několik známých šifer a jeho prolomením (nalezením efektivního algoritmu pro jeho řešení) bychom narušili jejich bezpečnost.

Pro řešení problému diskrétního logaritmu bylo vyvinuto několik algoritmů s exponenciální složitostí, ale existuje jeden, jehož složitost je subexponenci-

ální, patří mezi nejstarší a jmenuje se index calculus (indexem je myšlen právě hledaný exponent). Jedná se o nejefektivnější známý algoritmus. Jeho nevýhodou je, že lze použít jen v některých grupách. Subexponenciální složitosti totiž dosahuje využitím znalosti jejich struktury. Například nelze použít v často používaných grupách nad eliptickými křivkami. V jednom z kroků index calculu je třeba řešit rozsáhlou soustavu lineárních kongruencí. Tato soustava je velice řídká, to znamená, že většina hodnot v matici soustavy je nulová. Pro zacházení s řídkými maticemi existují specializované postupy a datové struktury. Tato práce se jim bude věnovat a bude hledat způsob, jak postup řešení takové soustavy urychlit pomocí grafické karty počítače.

Využití grafické karty pro výpočty jiné než grafické je poměrně nový přístup, který se začal využívat až v 21. století. Vžilo se označení *GPGPU*, tedy *General Purpose computing on Graphics Processing Units*. Grafická karta (dále budeme psát jen GPU) se od standardního procesoru liší především tím, že obsahuje mnohonásobně větší počet výpočetních jader, které jsou ale relativně pomalé. Proto je vhodná pro takové problémy, kde lze využít masivního paralelismu a kde dílčí úlohy nejsou těžké.

Možnosti urychlení index calculu je dobré zkoumat proto, abychom odhalili případnou možnost prolomení problému diskrétního logaritmu dříve, než se to podaří útočníkovi.

Kapitola 1

Index calculus

Základem této práce je algoritmus zvaný index calculus. Jedná se o jeden ze způsobů řešení problému diskrétního logaritmu, tedy hledání exponentu, na který je třeba umocnit daný prvek konečné grupy, abychom získali prvek jiný.

1.1 Problém diskrétního logaritmu

Diskrétním logaritmem nazýváme celé číslo k v roli exponentu, který byl použit pro umocnění jednoho prvku konečné grupy a získání jiného:

$$a^k = b. \tag{1.1}$$

Problémem diskrétního logaritmu nazýváme hledání tohoto exponentu, tedy řešení následující rovnosti pro k:

$$k = log_a b. \tag{1.2}$$

Diskrétní logaritmus je tak obdobou logaritmu známého z oboru reálných čísel. Pracujeme-li v konečné cyklické grupě a za prvek a volíme nějaký její generátor, pak bude diskrétní logaritmus vždy existovat. V této práci se soustřeďme pouze na tuto možnost.

V závislosti na volbě grupy může být hledání diskrétního logaritmu snadné nebo naopak v rozumném čase neproveditelné. V případě grupy celých čísel s násobením modulo prvočíslo p, psáno $\mathbb{Z}_p^\times$, představuje funkce 1.1 takzvanou jednosměrnou funkci, tedy takovou funkci, jejíž funkční hodnotu b v bodě k lze snadno spočítat modulárním mocněním základu a, zatímco nalézt vzor k k hodnotě b je těžké.

Jednosměrné funkce mají využití především v kryptografii, neboť umožňují utajit informaci pro přenos po nezabezpečeném kanálu tak, aby její odtajnění bylo obtížné. Několik asymetrických šifer je založeno na obtížnosti problému diskrétního logaritmu. Prolomit problém diskrétního logaritmu by například

znamenalo prolomit Diffieho-Hellmanův problém a šifrovací systém ElGamal [4].

1.2 Co je index calculus

Nalézt diskrétní logaritmus hrubou silou by vyžadovalo vyzkoušet všechny hodnoty exponentu $1, \ldots, N-1$, kde N je řád grupy, dokud bychom nedosáhli rovnosti 1.1. Složitost takového útoku je $O(N)$, neboť násobíme opakovaně základem a.

Je známých několik algoritmů, které řeší diskrétní logaritmus efektivněji. Například Babystep-Giantstep algoritmus, popsaný Danielem Shanksem v roce 1971, řeší problém diskrétního logaritmu v $O(\sqrt{N})$ krocích [5]. Takzvaná ρ-metoda popsaná Johnem Pollardem v roce 1978 má menší paměťovou náročnost při stejné asymptotické složitosti [6].

Asymptoticky nejrychlejší algoritmus se nazývá index calculus. Jeho počátky sahají až do roku 1922, ale publikace tohoto postupu a analýza jeho složitosti se objevila až v roce 1979 [7]. Zatímco výše zmíněné algoritmy fungují v libovolné grupě, index calculus těží z toho, že se soustředí jen na vybrané grupy a využívá znalosti jejich struktury.

1.3 Varianty

Index calculus není popsán obecně pro libovolnou grupu. V tomto textu se soustřeďme na dvě varianty: Variantu pro grupu $\mathbb{Z}_p^\times$ a variantu pro multiplikativní grupu konečného tělesa $GF(p^m)^*$, které se používají v kryptografii. Obě jsou vhodnými volbami grupy například pro šifrovací systém ElGamal.

1.3.1 Multiplikativní grupa $\mathbb{Z}_p^\times$

$\mathbb{Z}_p^\times$ je cyklická grupa řádu $N = \varphi(p) = p - 1$, kde p je prvočíslo, a obsahuje všechna přirozená čísla od 1 do $p - 1$. Operace definovaná na této grupě je operace násobení přirozených čísel modulo p. Tato grupa má $\varphi(N) = \varphi(p-1)$ generátorů, libovolný z nich můžeme zvolit pro demonstraci algoritmu.

1.3.2 Multiplikativní grupa konečného tělesa $GF(2^n)^*$

Množina prvků tělesa $GF(p^m)$ bez nuly spolu s operací násobení tvoří cyklickou grupu, které říkáme multiplikativní grupa tělesa. Operace sčítání a násobení na tomto tělese a tedy i operace na multiplikativní grupě jsou definovány modulo ireducibilní polynom stupně m, který budeme značit $f(x)$. Index calculus je popsán i pro tuto grupu. Pro účely této práce budeme uvažovat $p = 2$ a budeme pracovat v grupě $GF(2^n)^*$.

1.4 Algoritmus

Popišme nyní celý postup index calculu platný pro obě zmíněné varianty. Vstupem algoritmu je cyklická grupa G řádu N, nějaký její generátor $g \in G$ a prvek $h \in G$. Výstupem algoritmu je diskrétní logaritmus prvku h, tedy přirozené číslo $log_g h \in \{1, \ldots, N-1\}$.

1.4.1 Faktorová báze

Prvním krokem algoritmu je určení faktorové báze, podmnožiny grupy G takové, aby bylo možné co největší počet prvků G vyjádřit jako součin prvků z této množiny. Faktorovou bázi označíme $S \subset G$.

V případě varianty v $\mathbb{Z}_p^\times$ volíme za množinu S všechna prvočísla z G, která jsou menší nebo rovna předem zvolenému přirozenému číslu B. Prvky G, které je možno rozložit na součin prvků S, jsou pak ty prvky, které jsou B-hladké, tj. mají všechny prvočíselné dělitele menší nebo rovny B. Zmenšováním čísla B získáváme rychlejší ověření, zda je náhodně vybraný prvek B-hladký, zvětšováním získáváme větší pravděpodobnost, že takový bude.

V případě varianty v $GF(2^n)^*$ opět volíme přirozené číslo B a prvky S jsou ireducibilní polynomy stupně nejvýše B. Prvky G, které je možné rozložit na součin prvků S, jsou B-hladké polynomy, tedy takové, které lze rozložit na součin polynomů stupně nejvýše B. Při vyšším B je vyšší pravděpodobnost, že náhodně vybraný polynom z G je B-hladký, ale zároveň se tak zvětšuje soustava lineárních kongruencí ve fázi 1.4.2.

1.4.2 Výpočet logaritmů

Nyní potřebujeme vyčíslit logaritmus každého prvku faktorové báze S. Máme $|S|$ neznámých, vyžadujeme tedy soustavu alespoň $|S|$ kongruencí.

Zvolíme náhodně exponent $l \in \{1, \ldots, N-1\}$ a pokusíme se rozložit prvek g^l (kde g je vstupní generátor grupy) na součin prvků faktorové báze. Pokud to nelze, volíme jiné l. Pokud to lze, získáme t faktorů označených $p_1, \ldots, p_t$ násobností $c_1, \ldots, c_t$ a máme v grupě $\mathbb{Z}_p^\times$ kongruenci

$$g^l \equiv \prod_{i=1}^{t} p_i^{c_i} \pmod p \tag{1.3}$$

nebo v grupě $GF(2^n)^*$ obdobně

$$g^l(x) \equiv \prod_{i=1}^{t} p_i^{c_i}(x) \pmod{f(x)}, \tag{1.4}$$

kde $f(x)$ je zadaný ireducibilní polynom stupně n.

Provedeme logaritmus obou stran kongruence 1.3 či 1.4 a získáme následující kongruenci pro využití v soustavě:

$$
\begin{aligned}
\log_g g^l &\equiv log_g \prod_{i=1}^{t} p_i^{c_i} \pmod N, \\
l \log_g g &\equiv \sum_{i=1}^{t} log_g p_i^{c_i} \pmod N, \\
l &\equiv \sum_{i=1}^{t} c_i \log_g p_i \pmod N.
\end{aligned}
\tag{1.5}
$$

Kongruence 1.5 je nezávislá na volbě grupy G. Není totiž počítána v původní grupě G, nýbrž v okruhu celých čísel modulo N, kde N je řád grupy G. Logaritmizací obou stran kongruence 1.3 či 1.4 jsme se přesunuli k práci s exponenty, které nejsou prvky G, nýbrž celá čísla. Máme zde k dispozici množinu celých čísel 0 až $N - 1$ spolu se standardním sčítáním a násobením modulo N.

Tento postup musíme opakovat tolikrát, dokud nemáme dostatek kongruencí 1.5 pro vyjádření všech neznámých logaritmů prvků faktorové báze S. Budeme potřebovat najít více kongruencí než $|S|$, protože některé mohou být vzájemně lineárně závislé. Výsledkem je soustava lineárních kongruencí pro $|S|$ neznámých logaritmů, kde sloupec pravých stran tvoří nalezená l.

Soustavu kongruencí vyřešíme vyjádřením všech neznámých logaritmů.

1.4.3 Hledání řešící kongruence

V poslední fázi algoritmu opět volbou náhodného exponentu hledáme prvek z G rozložitelný na součin prvků faktorové báze. Na rozdíl od fáze 1.4.2 nám postačí nalézt jeden takový exponent k. Lze-li rozložit prvek hg^{-k} na u různých faktorů $p_1, \ldots, p_u$ násobností $d_1, \ldots, d_u$, máme v grupě $\mathbb{Z}_p^\times$ kongruenci

$$
hg^{-k} \equiv \prod_{i=1}^{u} p_i^{d_i} \pmod p \tag{1.6}
$$

nebo v grupě $GF(2^n)^*$ kongruenci

$$
hg^{-k}(x) \equiv \prod_{i=1}^{u} p_i^{d_i}(x) \pmod{f(x)}. \tag{1.7}
$$

Nyní opět provedeme logaritmizaci obou stran a získáme jednotný výsledek

$$\begin{aligned} \log_g hg^{-k} &\equiv \log_g \prod_{i=1}^{u} p_i^{d_i} \pmod N \\ \log_g h - k\log_g g &\equiv \sum_{i=1}^{u} \log_g p_i^{d_i} \pmod N \\ \log_g h - k &\equiv \sum_{i=1}^{u} d_i \log_g p_i \pmod N \\ \log_g h &\equiv \sum_{i=1}^{u} d_i \log_g p_i + k \pmod N. \end{aligned} \tag{1.8}$$

Pravá strana kongruence 1.8 neobsahuje žádné neznámé. Jejím prostým sečtením získáme hledaný výstup algoritmu $\log_g h$.

1.5 Složitost

Index calculus je první známý algoritmus řešící problém diskrétního logaritmu se subexponenciální složitostí. Algoritmus má subexponenciální složitost, jestliže počet operací potřebných k jeho provedení je $O(e^{f(k)})$, kde k je binární délka vstupu a $f(k) = o(k)$ [4].

Odvození složitosti je poměrně náročné, proto jej zde uveďme zjednodušeně. Věnujme se výpočtu jen pro grupu $GF(2^n)^*$, která je pro tuto práci klíčová.

1.5.1 Odvození nejvyššího stupně polynomu v S

V grupě $GF(2^n)^*$ sestává faktorová báze S z ireducibilních polynomů stupně nejvýše nějaké m. Správná volba tohoto stupně má vliv na výslednou složitost algoritmu. Volíme-li větší m, rychleji nalezneme kongruence ve fázi 1.4.2, protože je větší pravděpodobnost, že náhodně vybraný polynom je m-hladký. Volíme-li naopak menší m, rychleji vyřešíme soustavu těchto kongruencí, protože je méně neznámých.

Ireducibilních polynomů stupně j je přesně

$$I_j = \frac{1}{j} \sum_{k|j} \mu(k) 2^{(j/k)}, \tag{1.9}$$

kde μ je Möbiova funkce. Velikost faktorové báze je dána počtem ireducibilních polynomů stupně menšího nebo rovného m, zajímá nás tedy suma přes všechna j od 1 do m a ta je dle [4] přibližně rovna

$$|S| \approx \frac{1}{m} 2^{m+1} \tag{1.10}$$

Dále pravděpodobnost, že náhodně vybraný polynom stupně nejvýše $n-1$ lze faktorizovat jako součin ireducibilních polynomů stupně nejvýše m, je přibližně rovna

$$P_{fakt} \approx \left(\frac{m}{n}\right)^{(1+o(1))\frac{n}{m}} \tag{1.11}$$

za předpokladu, že $n^{\frac{1}{100}} \leq m \leq n^{\frac{99}{100}}$. Složitost řešení soustavy kongruencí je obecně $O(|S|^3)$. Protože musíme provést průměrně $1/P_{fakt}$ pokusů, než nalezneme jeden takový polynom, a protože jich potřebujeme nalézt (o něco více než) $|S|$, můžeme psát, že potřebujeme provést přibližně

$$|S|\frac{1}{P_{fakt}} + |S|^3 \approx \frac{1}{m}2^{m+1}\left(\frac{n}{m}\right)^{(1+o(1))\frac{n}{m}} + \frac{1}{m}2^{3m+3} \tag{1.12}$$

operací, abychom připravili a následně vyřešili soustavu kongruencí. Pravou stranu 1.12 lze dle [4] minimalizovat volbou

$$m = c_1\sqrt{n \log n}, \tag{1.13}$$

kde $c_1 \approx 0,57$.

1.5.2 Složitost algoritmu

Pro výpočet asymptotické složitosti můžeme zanedbat výrazy $1/m$ a multiplikativní konstanty a dosadit m spočtené v 1.13. Příprava soustavy kongruencí má složitost

$$\begin{aligned} |S|\frac{1}{P_{fakt}} &\approx \frac{1}{m}2^{m+1}\left(\frac{n}{m}\right)^{(1+o(1))\frac{n}{m}} \\ &\approx 2^m\left(\frac{n}{m}\right)^{(1+o(1))\frac{n}{m}} \\ &= O\left(exp\left((c_2 + o(1))\sqrt{n \log n}\right)\right). \end{aligned} \tag{1.14}$$

Řešení soustavy má složitost

$$\begin{aligned} |S|^3 &\approx \frac{1}{m}2^{3m+3} \\ &\approx 2^{3m} \\ &= O\left(exp\left(c_3\sqrt{n \log n}\right)\right). \end{aligned} \tag{1.15}$$

Na závěr hledáme jednu řešící kongruenci se složitostí

$$\begin{aligned} \frac{1}{P_{fakt}} &\approx \left(\frac{n}{m}\right)^{(1+o(1))\frac{n}{m}} \\ &= O\left(exp\left(\left(c_4 + o\left(1\right)\right)\sqrt{n \log n}\right)\right). \end{aligned} \tag{1.16}$$

Pro ukázku, že se skutečně jedná o subexponenciální složitost, se podívejme na funkci $L_q[\alpha, c]$, kterou se subexponenciální složitost často odhaduje:

$$L_q[\alpha, c] = exp\left(c\left(\log q\right)^{\alpha}\left(\log\log q\right)^{1-\alpha}\right). \tag{1.17}$$

Jak vidíme, všechny tři odvozené složitosti 1.14, 1.15 i 1.16 odpovídají funkci $L_{2^n}[\frac{1}{2}, c]$, liší se jen konstanta c.

Algoritmus se můžeme několika způsoby pokusit dále zrychlit. Jednou z možností je využití skutečnosti, že soustava kongruencí je řídká. Algoritmy pracující s řídkými maticemi mohou dosahovat asymptoticky nižší složitosti. Právě řešením řídkých soustav lineárních kongruencí se budeme v této práci dále zabývat. Pokud by se podařilo řešení soustavy asymptoticky urychlit, změnilo by to i výpočet optimálního největšího stupně polynomu ve faktorové bázi, který byl představen v sekci 1.5.1, protože bychom si mohli dovolit větší soustavu a tedy usnadnit stavbu báze. V rovnosti 1.13 by se však zvětšila jen konstanta c_1.

1.6 Neprvočíselný modul

Modul, v kterém řešíme soustavu lineárních kongruencí ve fázi výpočtu algoritmů 1.4.2, je roven řádu grupy, v které pracujeme, protože exponenty generátorů grupy má smysl počítat pouze modulo řád grupy. V případě $GF(2^n)^*$ je řád roven $2^n - 1$, protože těleso obsahuje 2^n prvků včetně nuly, která do jeho multiplikativní grupy nepatří. V případě $\mathbb{Z}_p^{\times}$ je řád grupy a tedy použitý modul roven $\varphi(p) = p - 1$, nikoli p, protože grupa opět neobsahuje nulu. Takové moduly nejsou vždy prvočíselné. To představuje problém, protože v neprvočíselném modulu nemá každý prvek inverzi, kterou potřebujeme pro řešení soustavy kongruencí.

V této sekci popíšeme, jak lze problém rozložit na podproblémy tak, aby podproblémy byly řešeny v modulech prvočíselných. Je třeba podotknout, že řád grupy, v které je šifrováno, je obyčejně volen tak, aby alespoň jeden z těchto modulů byl dost velký (nelze-li zvolit řád prvočíselný).

1.6.1 Čínská věta o zbytcích

Pro správné porozumění rozkladu je třeba připomenout, co říká čínská věta o zbytcích. Čínská věta o zbytcích tvrdí, že při zadaných k vzájemně nesouděl-

ných přirozených číslech $m_1, \dots, m_k$ v roli modulů a celých číslech $a_1, \dots, a_k$ existuje právě jedno $x \pmod{M = \prod_{i=1}^{k} m_i}$ takové, že platí

$$x \equiv a_i \pmod{m_i} \tag{1.18}$$

pro všechna $i \in \{1, \dots, k\}$.

1.6.2 Výpočet v neprvočíselném modulu

Algoritmus popsaný Stephenem Pohlingem a Martinem Hellmanem v roce 1978 dokáže nalézt diskrétní logaritmus v grupě řádu N s prvočíselným rozkladem $\prod_{i=1}^{k} q_i^{e_i}$ v $O\left(\sum_{i=1}^{k} e_i \left(S(q_i) + \log q_i\right)\right)$ krocích, kde $S(q_i)$ je složitost nalezení diskrétního logaritmu v grupě řádu q_i [8]. Algoritmus spočívá v řešení $e_i k$ problémů diskrétního logaritmu v grupách prvočíselných řádů q_i. Člen $\log q_i$ popisuje přípravné operace.

Nejprve potřebujeme pro každé $i \in 1, \dots, k$ nalézt y_i takové, že platí

$$\left(g^{N/q_i^{e_i}}\right)^{y_i} = h^{N/q_i^{e_i}}, \tag{1.19}$$

kde g je zadaný generátor grupy a h je zadaný prvek grupy. Výsledný diskrétní logaritmus x pak spočítáme pomocí čínské věty o zbytcích jako

$$x \equiv y_i \pmod{q_i^{e_i}} \tag{1.20}$$

pro všechna $i \in \{1, \dots, k\}$. Čínskou větu o zbytcích lze použít, protože čísla $q_i^{e_i}$ jsou jistě vzájemně nesoudělná, mají prázdný průnik prvočíselných rozkladů.

Řád prvku $g^{N/q_i^{e_i}}$ je ale $q_i^{e_i}$, což stále ještě nemusí být prvočíslo a tedy pro problém diskrétního logaritmu v rovnosti 1.19 stále ještě nemůžeme použít index calculus. Ve skutečnosti budeme pro nalezení jednoho y_i muset vyřešit e_i diskrétních logaritmů v grupách řádů q_i. Pro snadnější zápis přepišme rovnost 1.19 jako

$$g^x = h, \tag{1.21}$$

kde řád prvku g je q^e (pozor, jedná se o jiné g než je zadané). Exponent x přepíšeme jako

$$x = x_0 + x_1 q + x_2 q + \ldots + x_{e-1} q^{e-1} \tag{1.22}$$

a postupně hledáme všechna x_i pomocí rovnosti

$$\left(g^{q^{e-1}}\right)^{x_i} = \left(h g^{-x_0 - \ldots - x_{i-1} q^{i-1}}\right)^{q^{e-i-1}}. \tag{1.23}$$

Zde se konečně jedná o diskrétní logaritmy v grupách prvočíselného řádu q, na které můžeme použít index calculus.

Kapitola **2**

Řešení řídkých soustav lineárních kongruencí

V rámci index calculu je třeba řešit rozsáhlou soustavu lineárních kongruencí. Jak podrobněji zdůvodníme v kapitole 3, je tato soustava velice řídká. Řídké soustavy mají zvláštní vlastnosti, kterých lze využít pro urychlení výpočtů. Tato kapitola se věnuje datovým strukturám a algoritmům nad těmito soustavami. Prozatím se budeme věnovat řešení soustavy rovnic v reálných číslech, které je běžnější, a až poté specifikujeme, v čem se řešení kongruencí v celých číslech liší.

Řídké soustavy lineárních rovnic jsou mnohdy řešeny iterativními algoritmy, které pracují na základě iterativního zpřesňování výsledku. Čím více iterací, tím přesnějšího výsledku dosáhnou. V případě modulární aritmetiky ovšem nelze hovořit o vzdálenosti čísel tak jako u čísel reálných a nelze výsledek postupně zpřesňovat. Proto se musíme omezit na metody přímé.

Řídká soustava je vyjádřena řídkou maticí soustavy. Proto v této kapitole pracujeme s řídkými maticemi a vektory.

2.1 Řídká matice

Řídká matice je matice obsahující dostatek nul, aby se vyplatilo toho využít. To je poněkud vágní definice J. H. Wilkinsona, která je nicméně docela praktická. Opakem řídké matice je matice hustá.

Při práci s obecnou maticí jsme zvyklí ukládat všechny její hodnoty například v dvourozměrném poli a operace provádět nad všemi těmito hodnotami. Jak paměťová, tak výpočetní složitost algoritmů bývá úměrná rozměrům matice $m \times n$. V případě řídké matice by takový přístup představoval značné plýtvání, protože většina paměti by byla zaplněna nulami. Co se týče algoritmů, většina operací lze přeskočit, protože neznamenají žádnou úpravu dat. Při práci s řídkými maticemi se tedy snažíme, aby paměťová i výpočetní slo-

žitost byla úměrná počtu nenulových hodnot. Nenulovým hodnotám budeme nadále říkat jen „nenuly“.

Například násobení dvou čtvercových matic velikosti n má obecně složitost $O(n^3)$, jelikož máme n^2 hodnot výsledné matice a pro každou tuto hodnotu musíme provést n násobení a sčítání v řádku první a sloupci druhé násobené matice. Řešení soustavy rovnic (například Gaussovou eliminací matice soustavy) má složitost taktéž $O(n^3)$, jak bylo zmíněno v sekci 1.5. Při práci s řídkými soustavami, potažmo maticemi, lze tuto složitost asymptoticky snížit.

2.2 Datové struktury

Abychom neukládali do paměti všechny hodnoty, které jsou většinou nulové, a abychom mohli efektivně procházet jen nenuly, používají se komprimované datové struktury.

V této sekci budeme různými způsoby reprezentovat tuto ukázkovou matici:

$$A = \begin{pmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{pmatrix}. \tag{2.1}$$

2.2.1 Nenulové prvky

Nejprve je třeba stanovit, co je to nenula. Nenulový prvek nebo nenula je dvojice souřadnic, na kterých leží hodnota, kterou je třeba uvažovat v algoritmech. Při práci s řídkými maticemi nás mnohdy nezajímá, jaká číselná hodnota na tomto místě leží, neboť pracujeme s maticemi pouze strukturálně. Algoritmy stojí z velké části na práci s nenulovými vzory, zkoumají umístění nenul, vztahy mezi nimi a vznik nových. Nenula je tedy jakási černá skříňka, která se v matici pohybuje, má nějaké souvislosti s jinými skříňkami, ale hodnota v ní může být libovolná, dokonce i nulová.

Nastane-li situace, že strukturální nenula obsahuje numerickou nulu, obyčejně se tím v paměti ani v algoritmech nezabýváme, protože by to bylo příliš výpočetně náročné.

2.2.2 Souřadnicový formát

Vzhledem k řídkosti hodnot v matici je výhodnější ukládat nenuly spolu s jejich souřadnicemi než celou matici. Implementačně nejjednodušší způsob uložení hodnot je doslova takový. Spočívá v použití tří stejně velkých polí, kde první obsahuje indexy řádků, druhé indexy sloupců a třetí hodnoty. Zde je ukázka reprezentace matice A z 2.1:

```
int i[] =    {0,   1,   3,   1,   2,   3,   0,   2,   1,   3};
int j[] =    {0,   0,   0,   1,   1,   1,   2,   2,   3,   3};
double a[] = {4.5, 3.1, 3.5, 2.9, 1.7, 0.4, 3.2, 3.0, 0.9, 1.0};
```

Tento formát se snadno sestavuje, ale zabírá zbytečně moc místa v paměti a není vhodný ani pro algoritmy, protože iterace po řádcích i sloupcích je obtížná. Používá se tedy například pro vstup či výstup algoritmů.

2.2.3 Komprimované sloupce

Ukázka souřadnicového formátu v 2.2.2 napovídá, že je zde přebytečná informace. Pole `j` indexů sloupců nese v řadě se opakující hodnoty. Ze souřadnicového formátu snadno sestavíme tzv. komprimované sloupce, které tento nedostatek odstraňují. Každý sloupec je zde reprezentován dvojicemi index řádku a hodnota.

Formát používá tři pole. První pole `p` o velikosti $n + 1$, kde n je počet sloupců, obsahuje ukazatele na začátky sloupců, tedy indexy jejich počátků v druhých dvou polích. Poslední hodnota tohoto pole, ono $+1$, ukazuje za konec posledního sloupce, a tedy v něm lze nalézt celkový počet nenulových hodnot v matici. Velikost sloupce j je vždy `p[j+1] - p[j]`. První hodnota pole `p` je vždy nula.

Druhé a třetí pole `i` a `a` mají velikost rovnu počtu nenulových hodnot. Pole `i` obsahuje indexy řádků, pole `a` hodnoty. Indexy řádků hodnot ve sloupci j jsou uloženy v `i[p[j]]` až `i[p[j+1] - 1]` a hodnoty jako takové jsou na stejných pozicích v poli `a`.

```
int p[] =    {0,             3,             6,        8,        10};
int i[] =    {0,   1,   3,   1,   2,   3,   0,   2,   1,   3};
double a[] = {4.5, 3.1, 3.5, 2.9, 1.7, 0.4, 3.2, 3.0, 0.9, 1.0};
```

Převod souřadnicového formátu na formát komprimovaných sloupců lze provést v lineárním čase. Prvním průchodem spočítáme hodnoty v jednotlivých sloupcích, v druhém průchodu pak tyto hodnoty vyplníme [1].

Použitím tohoto formátu máme rychlý přístup ke sloupcům, můžeme je snadno procházet a snadno přidávat nové. Průchod řádku je naopak náročný.

2.2.4 Komprimované řádky

Formát komprimovaných řádků je obdobný 2.2.3, jen z pohledu řádků místo sloupců. Pole `p` zde představuje indexy počátků řádků, pole `i` indexy sloupců a pole `a` hodnoty jako takové. Ukázka pro matici A z 2.1:

```
int p[] =    {0,        2,             5,        7,             10};
int i[] =    {0,   2,   0,   1,   3,   1,   2,   0,   1,   3};
double a[] = {4.5, 3.2, 3.1, 2.9, 0.9, 1.7, 3.0, 3.5, 0.4, 1.0};
```

Tato verze je výhodnější pro průchod a přidávání nových řádků, naopak iterace po sloupci je pomalá.

2.2.5 Spojový seznam

Řídkou matici lze uložit i pomocí ukazatelů. Zde existuje několik možností. Můžeme uchopit matici po řádcích, po sloupcích nebo obojí současně.

Chceme-li uchovávat matici po řádcích, uložíme seznam ukazatelů na první nenulové prvky řádků. Každý nenulový prvek je vyjádřen strukturou, která v sobě drží jednak číselnou hodnotu a jednak ukazatel na příští, případně i předchozí nenulu na stejném řádku. Stejně tak můžeme uložit seznam prvních prvků sloupců.

Můžeme mít v paměti jak seznam prvních prvků řádků, tak seznam prvních prvků sloupců. Každý prvek pak drží dva ukazatele: Jeden na příští nenulu na řádku, jeden na příští nenulu ve sloupci.

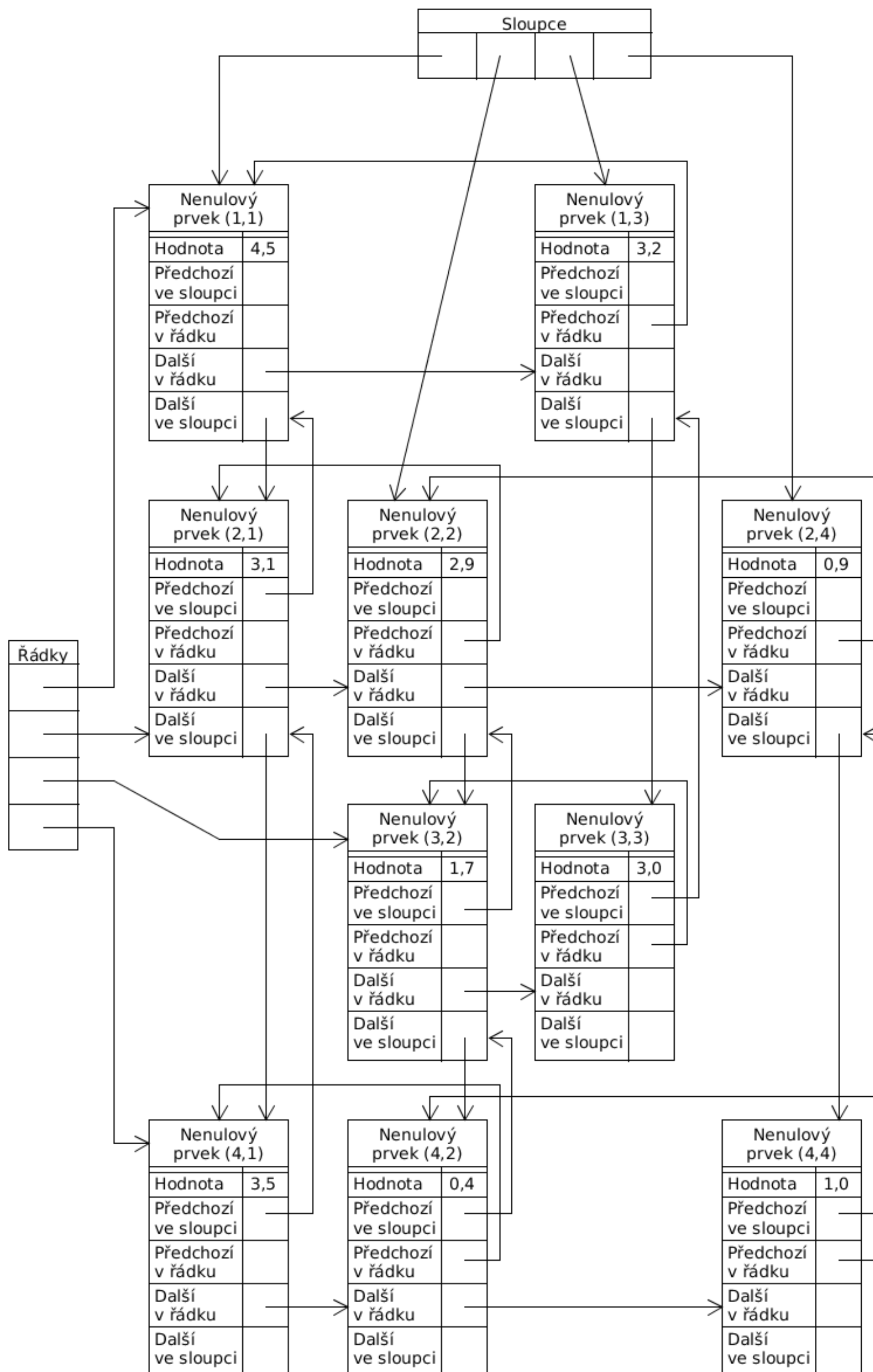
Na obrázku 2.1 je ukázka takové sloupcořádkové obousměrné reprezentace matice A. Výhoda této reprezentace je v rychlém odstraňování prvků, průchodu řádku i sloupce. Nevýhoda je v náročnější implementaci, větším objemu dat a pomalejší tvorbě. Reprezentace pomocí plnohodnotného spojového seznamu zabírá $m + n + 5k$ paměti, kde k je počet nenulových prvků. Oproti tomu souřadnicový formát zabere $3k$ paměti a komprimované sloupce jen $n + 1 + 2k$.

2.3 Základní operace

Při práci s řídkými maticemi hraje významnou roli tzv. nenulový vzor, tedy strukturální pohled na matici, kde nás zajímá jen umístění nenul, nikoli jejich numerické hodnoty. Pro některé algoritmy je nutné znát nenulový vzor výsledku dříve, než je spuštěn výpočet. Například v algoritmu QR faktorizace paradoxně známe dopředu nenulový vzor výsledné matice R a díky tomu víme, které sloupce v každém kroku procházet. Neznalost nenulového vzoru R by nás nutila procházet sloupce všechny, na což je třeba dát si pozor, ztratila by se výhoda řídké matice.

2.3.1 Dolní trojúhelníková soustava s hustou pravou stranou

Nejprve si ukážeme jeden z nejjednodušších algoritmů nad řídkou maticí, kterým je řešení dolní trojúhelníkové soustavy rovnic $Lx = b$, kde L je řídká dolní trojúhelníková matice a b je hustý vektor pravé strany. Matice L je uložena ve formátu komprimovaných sloupců, což je pro tento algoritmus nejvýhodnější, jelikož procházíme postupně sloupce. Pseudokód je k vidění v algoritmu 2.1. Řídkosti matice L využíváme tak, že procházíme ve vnitřním cyklu jen ty řádky, kde je v aktuálním sloupci j nenulová hodnota. Nemusíme procházet všechny řádky a tázat se na rovnost nule, protože ve formátu komprimovaných sloupců iterujeme přes nenulový vzor.



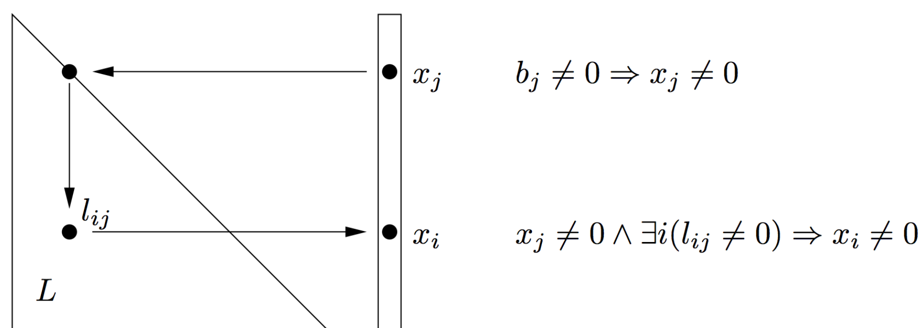
Obrázek 2.1: Reprezentace řídké matice pomocí spojového seznamu

Algoritmus 2.1 Řešení dolní trojúhelníkové soustavy $Lx = b$ s hustým vektorem pravé strany

Vstup: L, b
Výstup: x
  $x \leftarrow b$
  for $j = 1$ to n **do**
    $x_j \leftarrow x_j / l_{jj}$
    for all $i > j$ where $l_{ij} \neq 0$ **do**
      $x_i \leftarrow x_i - l_{ij} x_j$
    end for
  end for

2.3.2 Dolní trojúhelníková soustava s řídkou pravou stranou

Nyní se podívejme opět na algoritmus řešení dolní trojúhelníkové soustavy $Lx = b$, tentokrát ale s řídkým vektorem pravé strany. Pomůže nám to zavést představu řídké matice jakožto grafu. V případě této varianty se setkáme s problematikou vzniku nových nenul, tedy tzv. výplně (z anglického *fill-in*). Obrázek 2.2 zobrazuje princip vzniku nenul v řídkém vektoru výsledku x. Jsou zde zobrazena dvě pravidla. První plyne z úvodní kopie vektoru pravé strany do vektoru výsledku a říká, že byla-li nenula na určité pozici vektoru pravé strany, bude i na stejné pozici ve výsledku. Druhé pravidlo říká, že vznikne nenula v jakémkoli dalším řádku, kde je v aktuálním sloupci matice L nenula. To je proto, že od této pozice výsledku x, kde mohla být původně nula, bude odečten nenulový skalár $l_{ij} x_j$, viz algoritmus 2.1.



Obrázek 2.2: Doplnění při řešení řídké dolní trojúhelníkové soustavy s řídkým vektorem pravé strany [1]

Z tohoto jednoduchého příkladu by mělo být patrné, že lze rozhodnout o nenulovém vzoru výsledku x jen na základě nenulových vzorů vstupů L a b, nepotřebujeme znát hodnoty. Toho můžeme využít k značnému urych-

lení následného numerického výpočtu. Této přípravné strukturální fázi říkáme symbolická analýza.

2.3.3 Představa grafu

Šipky na obrázku 2.2 vyjadřují vznik nenuly na pozici i na základě existence nenuly na pozici j. Tento jeden případ vzniku nenuly je určen existencí nenuly l_{ij}. Jednotlivé pozice si můžeme představit jako vrcholy orientovaného acyklického grafu, kde hrana z j do i vyjadřuje existenci nějaké nenuly l_{ij}. Pro matici L velikosti n tak získáme graf o n vrcholech s počtem hran rovným počtu nenul pod diagonálou, kde hrana říká: Je-li ve zdrojovém vrcholu nenula, bude nenula ve vrcholu cílovém. Takový graf je jen jiným vyjádřením matice L.

Jak bude vypadat nenulový vzor výsledku x, známe-li tento graf? Označíme v grafu všechny vrcholy, kde jsou nenuly ve vstupním vektoru pravé strany b, čímž učiníme zadost prvnímu pravidlu z obrázku 2.2. Dále, podle druhého pravidla, označíme všechny vrcholy dostupné z označených vrcholů. Označené vrcholy teď tvoří hledaný nenulový vzor x. Označíme-li tento vzor $\mathcal{X}$, získáme algoritmus 2.2. Od algoritmu 2.1 se liší tím, že úplně přeskakuje výpočet hodnot x_j, o kterých dopředu víme, že budou nulové.

Algoritmus 2.2 Řešení dolní trojúhelníkové soustavy $Lx = b$ s řídkým vektorem pravé strany

---

Vstup: $L, b, \mathcal{X}$
Výstup: x
  $x \leftarrow b$
  for all $j \in \mathcal{X}$ **do**
    $x_j \leftarrow x_j / l_{jj}$
    for all $i > j$ where $l_{ij} \neq 0$ **do**
      $x_i \leftarrow x_i - l_{ij} x_j$
    end for
  end for

Algoritmus 2.2 je asymptoticky dokonalý, pracuje v čase přímo úměrném počtu potřebných aritmetických operací.

2.3.4 Horní trojúhelníková soustava

Řešení horní trojúhelníkové soustavy je obdobné, jen je konáno zdola nahoru, a tedy i výplň nenul v případě řídkého vektoru pravé strany probíhá opačně. Algoritmus 2.2 lze přepsat pro horní trojúhelníkovou soustavu prostým otočením postupu, viz algoritmus 2.3.

Algoritmus 2.3 Řešení horní trojúhelníkové soustavy $Ux = b$ s řídkým vektorem pravé strany

Vstup: $U, b, \mathcal{X}$
Výstup: x
  $x \leftarrow b$
  for all $j \in reverse(\mathcal{X})$ **do**
    $x_j \leftarrow x_j / l_{jj}$
    for all $i < j$ where $l_{ij} \neq 0$ **do**
      $x_i \leftarrow x_i - l_{ij} x_j$
    end for
  end for

2.4 Postup řešení soustavy

Řešení soustavy rovnic znamená řešit rovnici $Ax = b$ pro neznámý vektor x, kde A je čtvercová matice a b je vektor pravé strany.

Jak bylo zmíněno na začátku této kapitoly, má smysl zabývat se jen přímými algoritmy, tedy ne iterativními, protože pracujeme v konečném okruhu, kde nás zajímá jen přesný výsledek.

Postup se zpravidla skládá z dvou fází. V první fázi je třeba matici soustavy faktorizovat, tedy rozložit na součin dvou nebo více jiných matic. V druhé fázi pak tohoto rozkladu využijeme ke konečnému vyřešení soustavy. V této sekci si ukážeme tři nejznámější faktorizace i řešení a jejich průběh při řídké vstupní matici A. Jedná se o Choleského faktorizaci, LU faktorizaci a QR faktorizaci. Budeme je prozatím popisovat obecně v reálných číslech, specifika modulární aritmetiky odložíme na sekci 2.5.

Faktorizace obyčejně způsobuje výplň nových nenul. Čím více nových nenul vznikne, tím pracnější je řešení. Proto se před faktorizací matice permutuje. Můžeme aplikovat jak řádkovou, tak sloupcovou permutaci, a to pokud možno tak, aby při faktorizaci vzniklo co nejméně nových nenul. Bohužel je minimalizace výplně NP-úplný problém [9], proto se používají různé heuristiky. Navíc je třeba například v případě LU faktorizace spolu s minimalizací výplně usilovat i o to, aby se na místech pivotů objevila co největší čísla. Omezí se tak chyba výpočtu s čísly v plovoucí řádové čárce (netýká se však této práce, protože ta pracuje s čísly celými).

2.4.1 Choleského faktorizace

André-Louis Cholesky byl francouzský matematik, který je znám hlavně díky vynálezu podle něj nazvané Choleského faktorizace, která spočívá v rozložení symetrické pozitivně definitní matice na součin dolní a horní trojúhelníkové

matice, z nichž jedna je transpozicí druhé:

$$A = LL^T. \tag{2.2}$$

Problém $Ax = b$ pak lze přepsat jako

$$LL^T x = b \tag{2.3}$$

a v druhé fázi řešit jedním řešením dolní trojúhelníkové soustavy a jedním řešením horní trojúhelníkové soustavy:

$$\begin{aligned} Ly &= b \\ L^T x &= y. \end{aligned} \tag{2.4}$$

2.4.1.1 Algoritmus

Choleského faktorizaci, tedy hledání dolní trojúhelníkové matice L, lze konstruovat například po řádcích. Tento postup můžeme definovat matematickou indukcí. Indukční krok znázorníme pomocí blokových matic:

$$\begin{bmatrix} L_{11} & \\ l_{12}^T & l_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & l_{12} \\ & l_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} \\ a_{12}^T & a_{22} \end{bmatrix} \tag{2.5}$$

Rovnost 2.5 předpokládá, že jsme již faktorizovali prvních $k-1$ řádků a máme připravenu podmatici L_{11} velikosti $k-1 \times k-1$. Indukčním krokem je sestavení k-tého řádku spočtením vektoru l_{12} a skaláru l_{22}. Vektor l_{12} zjistíme vyřešením řídké dolní trojúhelníkové soustavy s řídkou pravou stranou

$$L_{11} l_{12} = a_{12}, \tag{2.6}$$

což umíme řešit algoritmem 2.2. Skalár l_{22} spočítáme vyřešením rovnice

$$\begin{aligned} l_{12}^T l_{12} + l_{22}^2 &= a_{22} \\ l_{22}^2 &= a_{22} - l_{12}^T l_{12} \\ l_{22} &= \sqrt{a_{22} - l_{12}^T l_{12}}, \end{aligned} \tag{2.7}$$

kde se vyskytuje skalární součin řídkého vektoru získaného v 2.6 se sebou sama, což je operace se složitostí přímo úměrnou počtu nenul ve vektoru, a následně několik operací se skaláry.

Pro úplnost je třeba doplnit indukční předpoklad, tedy schopnost spočítat první řádek matice L, což je jediný skalár l_{11}. Tato zmenšenina Choleského faktorizace vypadá takto:

$$l_{11}^2 = a_{11} \tag{2.8}$$

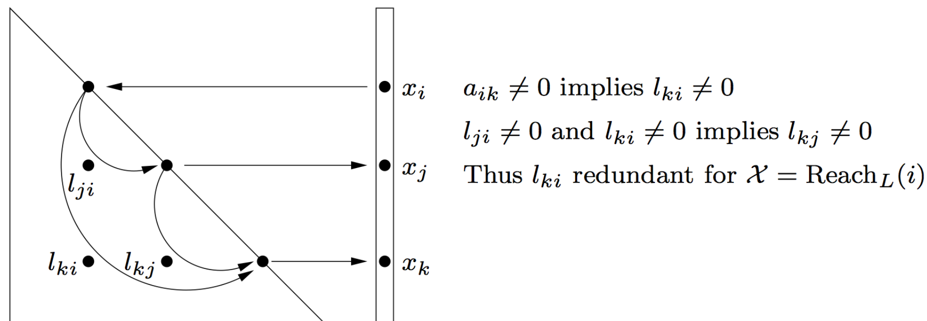
a hledaným skalárem l_{11} je odmocnina z a_{11}.

Algoritmus si můžeme představit rekurzivně, kde v kroku k vyřešíme rekurzivně Choleského faktorizaci matice velikosti $k-1$ a následně spočítáme 2.6 a 2.7 pro získání faktorizace matice velikosti k.

2.4.1.2 Výplň nenul v Choleského faktorizaci

Existují i jiné algoritmy, které skládají matici L z jiných směrů, ale mají jeden společný problém. Potřebují předem znát počty nenulových hodnot v řádcích a sloupcích výsledné matice L, tedy znát její nenulový vzor $\mathcal{L}$. Pro efektivní zjištění $\mathcal{L}$ je třeba definovat pojem eliminační strom.

Podívejme se nejprve na obrázek 2.3, který je rozšířením obrázku 2.2 a zobrazuje řešení dolní trojúhelníkové soustavy popsané rovnicí 2.6 v k-tém kroku Choleského faktorizace. Všimněme si, že vektor pravé strany, zde označený x, je k-tým sloupcem matice A a po jeho transformaci v řešení soustavy se stane k-tým řádkem matice L. Zajímá nás nenulový vzor řešení, tedy k-tého řádku, který označme $\mathcal{L}_k$.



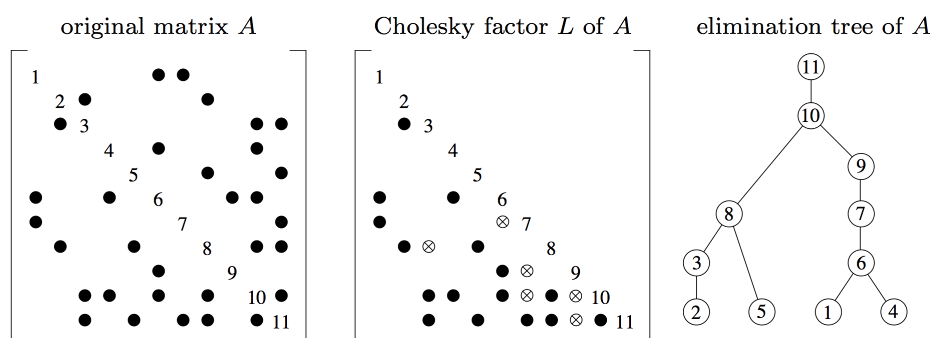
Obrázek 2.3: Ořez grafu L, vznik eliminačního stromu [1]

Řekněme, že existuje nějaká nenula na pozici a_{ik} neboli x_i. Protože se x stane $k$tým řádkem L, bude nenula i na pozici l_{ki}. Dále si představme, že existuje další nenula na pozici l_{ji}, tedy přímo nad l_{ki}. Podle 2.2 platí, že vznikne nenula na pozici výsledku x_j. Ta se ale poté přesune na pozici l_{kj}. Výsledkem tedy budou tři nenuly v matici L na pozicích l_{ji}, l_{ki} a l_{kj} [10]. Jak je vidět na obrázku 2.3, stává se tak při hledání budoucích vzorů řádků za pomoci grafu L a vrcholů dostupných z vrcholu i hrana l_{ki} redundantní, protože do vrcholu k se dostaneme i přes vrchol j.

Z předchozího odstavce plyne, že lze graf L ořezat tak, aby z každého vrcholu i vycházela nejvýše jedna hrana, a to ta příslušící první nenule pod diagonálou ve sloupci i. Z grafu se tak stane strom, kterému říkáme eliminační strom matice A. (Může se jednat i o les, přesto se nazývá eliminačním stromem.) Nalezení tohoto stromu v rámci symbolické analýzy jen na základě

matice A a jeho využití pro zjištění počtů prvků v řádcích a sloupcích L je proveditelné v čase $O(|A|)$, kde $|A|$ je počet nenul v matici A [11] [12]. Postup hledání počtů prvků je ale komplikovaný a nad rámec této práce. Lze jej nastudovat například v [1].

Na obrázku 2.4 ještě vidíme čistě strukturální příklad vstupní matice A, Choleského faktoru L a eliminačního stromu A. Doplněné nenuly jsou označeny kroužky s křížkem, původní kroužky plnými.



Obrázek 2.4: Příklad Choleského faktorizace a eliminačního stromu [1]

Choleského faktorizace je ze tří zde uvedených nejméně obecná, neboť vyžaduje speciálně symetrickou pozitivně definitní matici.

2.4.2 LU faktorizace

LU faktorizace libovolné regulární matice A je rozklad na dolní trojúhelníkovou matici L a horní trojúhelníkovou matici U takto:

$$A = LU. \tag{2.9}$$

V této práci budeme uvažovat variantu, kde má matice L jednotkovou diagonálu. Obvykle je třeba permutovat alespoň řádky matice A pomocí řádkové permutace P (*partial pivoting*), ale pro účely minimalizace výplně nenul (a hledání co největších pivotů v reálném případě) se zpravidla provádí i sloupcová permutace Q (*full pivoting*). Celý rozklad je pak

$$PAQ = LU. \tag{2.10}$$

Problém pak můžeme přepsat za použití rozkladu 2.10:

$$\begin{aligned} Ax &= b \\ PAx &= Pb \\ PAQQ^{-1}x &= Pb \\ LUQ^{-1}x &= Pb. \end{aligned} \tag{2.11}$$

Označíme-li $z = Q^{-1}x$ a $y = Uz$, pak postup řešení vyžaduje čtyři kroky: Permutaci vektoru pravé strany b podle řádkové permutace P, řešení dolní a horní trojúhelníkové soustavy a nakonec permutaci výsledku podle sloupcové permutace Q:

$$\begin{aligned} b' &= Pb \\ Ly &= b' \\ Uz &= y \\ x &= Qz. \end{aligned} \tag{2.12}$$

Bez sloupcové permutace by byl vynechán poslední krok, a kdybychom se obešli i bez řádkové permutace, vynechali bychom i ten první.

2.4.2.1 Algoritmus

Rozklad permutované matice PAQ na faktory L a U lze provést mnoha různými postupy. Zde uvedeme postup, který staví faktory po sloupcích zleva doprava. Tak jako v případě Choleského faktorizace, i zde postup popíšeme matematickou indukcí. Algoritmus pak lze odvodit přirozeně rekurzí.

Řekněme, že umíme nalézt prvních $k-1$ sloupců matic L a U. Indukční krok nám ukáže, že dovedeme nalézt $k$tý sloupec. Pomůžeme si blokovými maticemi. První řádek a sloupec mají velikost $k-1$. Prostřední řádek i sloupec jsou $k$té a mají velikost 1. Poslední řádek a sloupec představují zbytek matice.

$$\begin{bmatrix} L_{11} & & \\ l_{21} & 1 & \\ L_{31} & l_{32} & L_{33} \end{bmatrix} \begin{bmatrix} U_{11} & u_{12} & U_{13} \\ & u_{22} & u_{23} \\ & & U_{33} \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} & A_{13} \\ a_{21} & a_{22} & a_{23} \\ A_{31} & a_{32} & A_{33} \end{bmatrix} \tag{2.13}$$

Podle pravidel pro násobení matic (které platí i pro matice blokové) odvodíme jednu rovnici pro každou z neznámých u_{12}, u_{22} a l_{32}:

$$L_{11}u_{12} = a_{12} \tag{2.14}$$

$$l_{21}u_{12} + u_{22} = a_{22} \tag{2.15}$$

$$L_{31}u_{12} + l_{32}u_{22} = a_{32} \tag{2.16}$$

Z rovnice 2.14 můžeme vyjádřit u_{12}, dosadit do 2.15 a vyjádřit u_{22}, dosadit do 2.16 a vyjádřit l_{32}. Tím jsme získali celý sloupec k matic L i U a dokázali indukční krok.

Indukčním předpokladem je schopnost nalézt první sloupec L a U. To ale není problém, z blokových matic vypustíme první řádky i sloupce, neznámá u_{12} nám úplně vypadne spolu s rovnicí 2.14 a druhé dvě rovnice přijdou o první člen.

Máme tedy dokázáno, že tímto způsobem lze najít celý rozklad. Jak ale ukázali Gilbert a Peierls v [13], k-tý sloupec lze nalézt jediným řešením řídké dolní trojúhelníkové soustavy s řídkým vektorem pravé strany algoritmem 2.2. Vezměme si následující soustavu vyjádřenou blokově:

$$\begin{bmatrix} L_{11} & & \\ l_{21} & 1 & \\ L_{31} & 0 & I \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \end{bmatrix}, \tag{2.17}$$

kde I je jednotková matice, $x_1 = u_{12}$, $x_2 = u_{22}$ a $x_3 = l_{32}u_{22}$. Pak sestava 2.17 vyjadřuje právě rovnice 2.14, 2.15 a 2.16. Dolní trojúhelníkovou matici z 2.17 nemusíme od k-tého sloupce dál držet v paměti, příslušné nuly a jedničky můžeme předpokládat v algoritmu 2.2 implicitně.

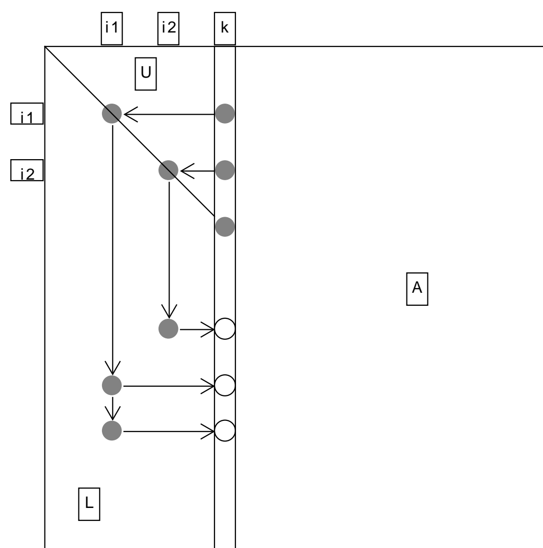
2.4.2.2 Výplň nenul v LU faktorizaci

Tak jako v ukázce algoritmu výše budeme i v této sekci předpokládat již permutovanou matici, kde se tzv. pivotním prvkem v každém kroku stane vždy další prvek na diagonále. V praxi jsou permutace většinou prováděny za běhu tak, že se v každém kroku vybere dosud nevybraný řádek a dosud nevybraný sloupec jako pivotní na základě nějaké heuristiky. Pivotní prvek je jejich průsečík. To by ale v tuto chvíli jen komplikovalo popis.

Podívejme se podrobněji na dolní trojúhelníkovou soustavu 2.17. Jaký je nenulový vzor sloupce k, který tato soustava počítá? Dle obrázku 2.2 bude mít stejný vzor jako sloupec k matice A plus výplň. Pro každou nenulu a_{ik} (potažmo x_i) nad diagonálou se podíváme do již hotového předchozího sloupce i matice L a kdekoli najdeme nenulu l_{ji} pod diagonálou, vytvoříme (novou) nenulu i v počítaném sloupci x_j. Výplň je znázorněna prázdnými kolečky na obrázku 2.5. Nefaktorizovaná část matice A a faktorizovaná část matic L a U jsou zde zobrazeny pro lepší názornost v jedné matici.

Jinak řečeno z opačného pohledu směrem do nefaktorizované části A, nenulový vzor pivotního sloupce i pod diagonálou se rozšíří do právě těch budoucích sloupců k matic L i U, kde byla nenula v matici A na řádku i a pozici a_{ik}. Co je méně patrné je, že i nenulový vzor pivotního řádku i napravo od diagonály se rozšíří do těch budoucích řádků k, kde byla nenula v matici A ve sloupci i na pozici a_{ki}. Toto sjednocení nenulových vzorů je zobrazeno na obrázku 2.6 a dává základ k jinému přístupu k faktorizaci zvanému multifrontální.

Nenulový vzor matice U je shora omezen nenulovým vzorem matice R v QR faktorizaci (2.4.3), který je dán tzv. sloupcovým eliminačním stromem matice A^TA. Jinými slovy, není možné, aby v matici U vznikla nenula, která není popsána sloupcovým eliminačním stromem A^TA [14]. Toho lze využít při přípravě struktury matice U. Toto omezení rozebereme podrobněji v sekci 2.4.3.3.



Obrázek 2.5: Výplň sloupce k za základě předchozích sloupců

2.4.2.3 Multifrontální přístup

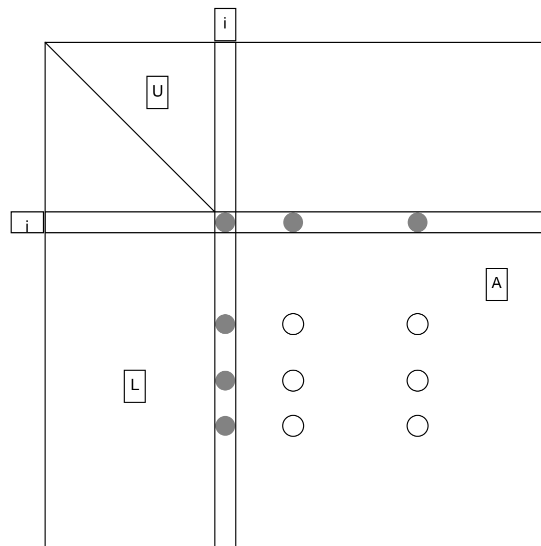
Multifrontální metoda byla poprvé navržena v roce 1983 [15]. Těží z poznatku viditelného na obrázku 2.6, že výběrem pivotu v určitém kroku i faktorizace vytvoříme v grafu kliku. Představme si plná kolečka v řádku a sloupci i jako hrany grafu matice A. Výběrem pivotu a_{ii} a jeho odstraněním z grafu vznikne klika mezi všemi zbývajícími sousedy vrcholu i.

Tato klika tvoří hustou podmatici matice A a označíme ji maticí frontální. Sloučíme-li ji do jednoho vrcholu grafu a sjednotíme-li hrany z ní vycházející, zjistíme, že v ní můžeme udělat jeden krok faktorizace samostatně metodami pro faktorizaci husté matice. Nefaktorizovanou, ale upravenou část frontální matice pak zakomponujeme do příslušných příštích frontálních matic podle hran grafu. Této upravené části frontální matice říkáme kontribuční blok, protože kontribuuje do příštích frontálních matic.

Multifrontální metoda tedy v rámci symbolické analýzy připraví seznam malých hustých frontálních matic a vztahů mezi nimi a veškerou numerickou faktorizaci provádí uvnitř metodami pro řešení hustých matic.

Ukáže se, že kontribuční strom frontálních matic je tzv. sloupcovým eliminačním stromem matice $A^T A$. V případě strukturálně symetrické matice A je tento strom rovný eliminačnímu stromu matice A již popsanému v sekci 2.4.1.

Dále se ukáže, že je možné do jedné frontální matice sloučit i několik kroků faktorizace, mají-li příslušné pivotní řádky a sloupce podobné nenulové vzory. Čím volnější podobnost zde umožníme, tím více vznikne ve frontální matici explicitních nul, ale současně získáme rychlejší výpočet díky většímu počtu



Obrázek 2.6: Budoucí výplň dosud nefaktorizované části A na základě nenulového vzoru pivotního řádku a sloupce i

kroků faktorizace provedených najednou v jedné husté podmatici.

Ukázka multifrontální metody na strukturálně symetrické řídké matici A je na obrázku 2.7.

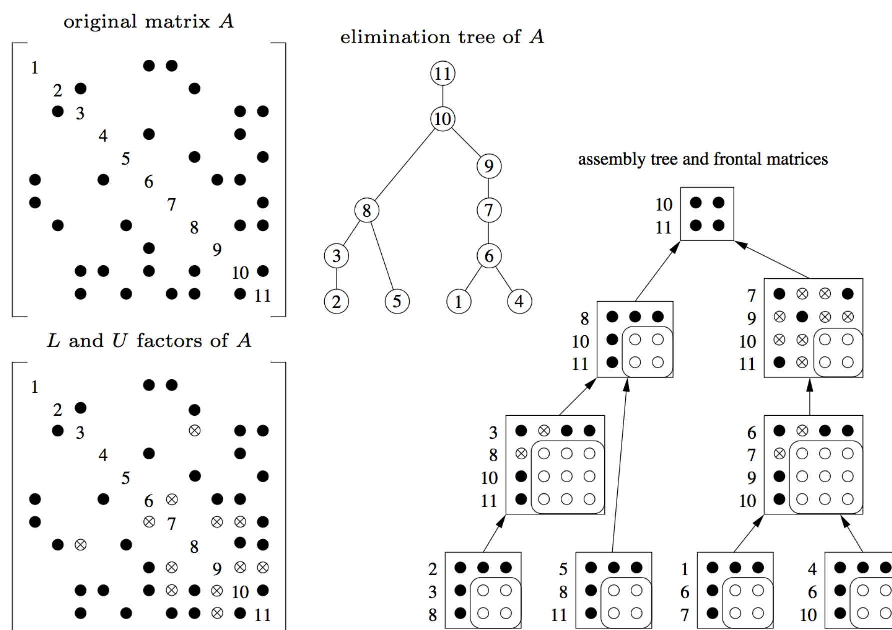
Nesymetrický případ je poněkud komplikovanější, protože jsou frontální matice obdélníkové a navíc mohou kontribuovat do více budoucích podmatic. Kontribuční strom frontálních matic tedy není stromem, nýbrž orientovaným acyklickým grafem. Analýza této problematiky je dílem především prof. Tima Davise, jehož práce je pro tento text hlavním zdrojem. Ukázka asymetrického případu je na obrázku 2.8.

2.4.2.4 Heuristika hledání pivotů

V předchozím rozboru jsme předpokládali již permutovanou matici PAQ, kde jsme pro větší názornost za pivoty brali postupně prvky na diagonále. Jak ale nalézt vhodnou permutaci?

Jak jsme ukázali v sekci 2.4.2.2, výběr pivotu má zásadní vliv na výplň nenul. Nalézt ideální posloupnost výběru pivotů tak, aby byla výplň minimální, je NP-úplný problém [9], proto je třeba použít nějakou heuristiku.

Rozumnou možností by se zdál být výběr pivotu na základě nejnižšího stupně vrcholu. V každém kroku faktorizace by se za pivotní zvolila ta nenula, která má v aktuálním grafu zbývajících nenul nejmenší stupeň, tedy ta, v jejíž řádku i sloupci je v dosud nefaktorizované části A nejméně nenul. V každém kroku bychom tak vytvořili co nejmenší kliku v grafu, tedy co nejméně nových



Obrázek 2.7: Multifrontální metoda faktorizace symetrické matice [1]

hran – nenul.

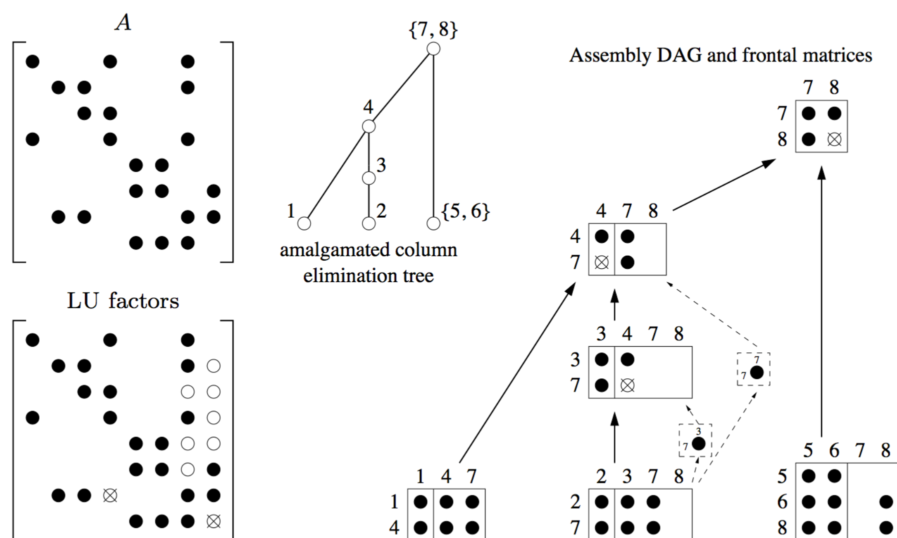
Problém heuristiky na základě nejnižšího stupně je v tom, že je příliš výpočetně náročná, vyžaduje neustálé přepočítávání nenul ve sloupcích a řádcích. Proto se často používá heuristika na základě přibližného nejnižšího stupně (*AMD – approximate minimum degree*). Místo aby analytický algoritmus počítal explicitně stupně vrcholů, udržuje v paměti jen horní mez sloupcového i řádkového stupně každého vrcholu, která lze spočítat rychleji. Například v případě multifrontální metody lze horní mez sloupcového stupně vrcholu j spočítat v čase $O(|\mathcal{A}^k_{*j}| + |\mathcal{C}_j|)$, kde $|\mathcal{A}^k_{*j}|$ je počet nenul ve sloupci j nefaktorizované části matice A a $|\mathcal{C}_j|$ je počet předchozích frontálních matic kontribuujících do tohoto sloupce [16].

2.4.3 QR faktorizace

QR faktorizace je ze tří zde uvedených nejobecnější, protože nevyžaduje regulární vstupní matici A. Umí pracovat s nedourčenou soustavou a řešit problém nejmenších čtverců v případě soustavy přeurčené (minimalizovat chybu) [1]. Zachází tedy s maticí soustavy obecně obdélníkovou velikosti $m \times n$.

Vstupní matici A (s jejími řádkovými a sloupcovými permutacemi, tak jako u LU faktorizace) rozložíme na součin matic Q a R

$$QR = A, \tag{2.18}$$



Obrázek 2.8: Multifrontální metoda faktorizace asymetrické matice [1]

kde Q je čtvercová ortogonální matice velikosti $m \times m$ a R je matice horní trojúhelníková stejných rozměrů jako matice A, tedy $m \times n$. Ortogonální matice je taková matice, pro níž platí $Q^T Q = I$, kde I je jednotková matice. Jinými slovy, inverzi matice Q získáme pouhou transpozicí.

Ortogonální matice má tu vlastnost, že násobíme-li jí vektor, nezmění vektor svou normu, jen se otáčí v prostoru. Platí rovnosti

$$||x||_2 = ||Qx||_2, \tag{2.19}$$

$$||x||_2 = ||Q^T x||_2. \tag{2.20}$$

Ukažme si jejich význam na problému nejmenších čtverců, kde místo rovnosti $Ax = b$ hledáme nejmenší chybu této rovnosti, tedy

$$\min ||Ax - b||_2. \tag{2.21}$$

Protože platí 2.20 a protože $R = Q^{-1} A = Q^T A$ můžeme psát

$$||Ax - b||_2 = ||Q^T (Ax - b)||_2 = ||Rx - b'||_2. \tag{2.22}$$

Pro lepší představu rozepíšeme $Rx - b'$ pomocí blokových matic. Protože R vyplňuje jen horní trojúhelník obdélníkové matice, tvoří její spodní řádky nulové vektory, a proto platí

$$\begin{bmatrix} R_1 \\ 0 \end{bmatrix} \begin{bmatrix} x \end{bmatrix} - \begin{bmatrix} b'_1 \\ b'_2 \end{bmatrix} = \begin{bmatrix} R_1 x - b'_1 \\ -b'_2 \end{bmatrix}, \tag{2.23}$$

čili

$$||Rx - b'||_2 = ||R_1x - b'_1||_2 + || - b'_2||_2 \tag{2.24}$$

Člen $||R_1x - b'_1||_2$ umíme minimalizovat až k jeho úplnému odstranění vyřešením horní trojúhelníkové soustavy $R_1x = b'_1$ a obdržíme řešení problému

$$\min ||Ax - b||_2 = || - b'_2||_2. \tag{2.25}$$

Volbou jiného vektoru x než toho, který řeší soustavu $R_1x = b'_1$, bychom nemohli dosáhnout menší chyby než 2.25, protože jeden člen 2.24 jsme minimalizovali na nulu a v druhém se proměnná x nevyskytuje.

2.4.3.1 Hledání Householderovy reflexe

Horní trojúhelníkovou matici R můžeme tvořit například po sloupcích, kde v každém kroku k eliminujeme všechny nenuly ve sloupci k pod diagonálou. Eliminaci provedeme tzv. Householderovou reflexí sloupce k. To znamená, že vynásobíme již částečně faktorizovanou matici A^k (označení pro částečně faktorizovanou matici A v k-tém kroku) ortogonální maticí H_k, která otočí sloupec k tak, aby pod jeho diagonálou byly jen nuly. Jak již bylo řečeno, vynásobením ortogonální maticí se nezmění norma tohoto (ani jiného) sloupce, jen je otočen v prostoru tak, aby všechny souřadnice větší než k byly nulové.

Vynásobením matice A ortogonální maticí H_k v každém kroku $k = 1, \ldots, n$ získáme

$$R = H_n \cdot H_{n-1} \cdots H_1 \cdot A = Q^T A. \tag{2.26}$$

Pro matici Q tedy platí

$$Q = H_1^T \cdot H_2^T \cdots H_n^T, \tag{2.27}$$

ale vzhledem k tomu, že ve většině případů Q na nic nepotřebujeme, nemusíme obyčejně zaznamenávat ani matice H_k v jednotlivých krocích.

Podívejme se nyní konkrétněji na krok k faktorizace. Žádáme ve sloupci k vynulovat všechny hodnoty pod diagonálou a současně nechceme, aby Householderova reflexe ovlivnila již faktorizovanou část. Ukáže se, že lze za H_k zvolit matici v blokovém tvaru

$$H_k = \begin{bmatrix} I & 0 \\ 0 & H'_k \end{bmatrix}, \tag{2.28}$$

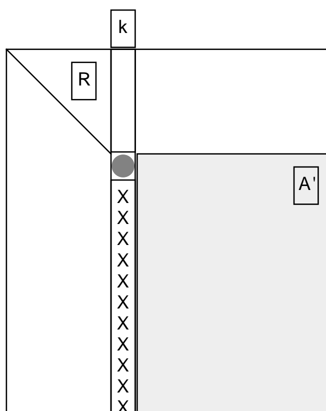
kde H'_k je matice Householderovy reflexe velikosti $m - k + 1 \times n - k + 1$. Pro důkaz, že matice 2.28 neovlivní již faktorizovanou část, máme

$$H_k A^k = \begin{bmatrix} I & 0 \\ 0 & H'_k \end{bmatrix} \begin{bmatrix} X & Y \\ 0 & Z \end{bmatrix} = \begin{bmatrix} X & Y \\ 0 & H'_k Z \end{bmatrix}. \tag{2.29}$$

A pro úplnost dokažme, že je matice H_k ve tvaru 2.28 ortogonální za předpokladu ortogonality matice H'_k, tedy za předpokladu, že $H'^T_k H'_k = I$:

$$H_k^T H_k = \begin{bmatrix} I & 0 \\ 0 & H'^T_k \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & H'_k \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & H'^T_k H'_k \end{bmatrix} = I. \tag{2.30}$$

Nyní tedy v každém kroku k hledáme zmenšenou Householderovu reflexi H'_k tak, aby vynulovala ve vektoru prvního sloupce dosud nefaktorizované podmatice A' velikosti $m - k + 1 \times n - k + 1$ všechny souřadnice až na první. Jinými slovy, v každém kroku řešíme stejný problém, jen menšího a menšího rozměru. Nákres je na obrázku 2.9.



Obrázek 2.9: Krok k QR faktorizace

Matice Householderovy reflexe má obecně tvar

$$H = I - v\beta v^T, \tag{2.31}$$

kde $\beta = \frac{2}{v^T v}$. Je tedy zcela určena jediným vektorem v, který budeme nazývat Householderovým vektorem a ukážeme si, jak jej nalézt. Aplikujeme-li takovou reflexi na první sloupec aktuální podmatice A' označený y, získáme

$$Hy = y - v\beta v^T y, \tag{2.32}$$

kde $\beta v^T y$ je skalár. Od vektoru y odčítáme nějaký násobek vektoru v a přejeme si, aby výsledek měl jen první souřadnici nenulovou. Je třeba si uvědomit, že dopředu známe výsledek, jelikož víme, že vzhledem k ortogonalitě H musí platit $||Hy||_2 = ||y||_2$, tedy že

$$Hy = \begin{pmatrix} ||y||_2 \\ 0 \\ \dots \\ 0 \end{pmatrix}. \tag{2.33}$$

Implementačně nejjednodušší je položit implicitně $\beta v^T y = 1$ a stanovit vektor v tvaru

$$v = \begin{pmatrix} y_1 - ||y||_2 \\ y_2 \\ y_3 \\ \cdots \\ y_{m-k+1} \end{pmatrix}, \tag{2.34}$$

aby po odečtení v od y první souřadnice vyšla $||y||_2$ a ostatní byly vynulovány. Víme, že tato volba plní požadavky, protože vede ke správnému výsledku, nicméně pro úplnost ještě ukažme, že platí $\beta = \frac{2}{v^T v}$, tedy že se skutečně jedná o Householderovu reflexi:

$$\begin{aligned} \beta v^T y = \frac{2v^T y}{v^T v} &= \frac{2 \cdot \Big((y_1 - ||y||_2) \cdot y_1 + y_2^2 + y_3^2 + \ldots + y_{m-k+1}^2 \Big)}{(y_1 - ||y||_2)^2 + y_2^2 + y_3^2 + \ldots + y_{m-k+1}^2} \\ &= \frac{2 \cdot \Big(y_1^2 + y_2^2 + \ldots + y_{m-k+1}^2 - ||y||_2 \cdot y_1 \Big)}{y_1^2 + y_2^2 + \ldots + y_{m-k+1}^2 - 2 \cdot ||y||_2 \cdot y_1 + ||y||_2^2} \\ &= \frac{2 \cdot ||y||_2^2 - 2 \cdot ||y||_2 \cdot y_1}{2 \cdot ||y||_2^2 - 2 \cdot ||y||_2 \cdot y_1} = 1. \end{aligned} \tag{2.35}$$

Tato volba vektoru v je vhodná pro implementaci, protože nám v kroku k stačí spočítat $||y||_2$, tedy normu sloupce k od diagonály níže. Vzhledem k tomu, že by v paměti ve sloupci k pod diagonálou zbyly samé nuly, můžeme zde nechat původní hodnoty a bez práce máme uložený téměř celý Householderův vektor v (až na diagonálu).

2.4.3.2 Algoritmus

Nejjednodušším algoritmem QR faktorizace je algoritmus 2.4. V každém kroku k nalezne Householderovu reflexi pro část sloupce k od diagonály níže a následně tuto reflexi aplikuje na celou aktuální podmatici A', čímž vynuluje její první sloupec. Nakonec je v matici A výsledek, tak jej přesuneme do výstupní matice R. Householderovy vektory jsou uloženy pro pořádek zvlášť v matici V a parametry β jsou ve vektoru Beta.

Je třeba upozornit na důležitý poznatek ohledně složitosti výpočtu. Násobíme-li nejprve $v\beta v^T$, získáme matici, kterou pak násobíme maticí $A[k:m, k:n]$ a dopouštíme se násobení matice maticí se složitostí obecně $O(n^3)$. Pokud ale nejprve vynásobíme $v^T A[k:m, k:n]$, násobíme maticí vektor a pak vektorem vektor a složitost nepřesáhne $O(n^2)$.

Algoritmus 2.4 v každém kroku upravuje všechny sloupce matice napravo. Druhá možnost je upravovat sloupec k až ve chvíli, kdy je to potřeba, tedy před krokem k. Tento postup najdeme v algoritmu 2.5. Zacházíme zde navíc

Algoritmus 2.4 QR faktorizace s okamžitými úpravami

Vstup: $A[m,n]$
Výstup: $R[m,n], V[m,n]$, Beta$[n]$
  for $k = 1$ to n **do**
    $v, \beta \leftarrow$ householder$(A[k:m,k])$
    $V[k:m,k] \leftarrow v$
    Beta$[k] \leftarrow \beta$
    $A[k:m,k:n] \leftarrow A[k:m,k:n] - v\beta v^T A[k:m,k:n]$
  end for
  $R \leftarrow A$

s hodnotou σ, která značí velikost vektoru, podle nějž počítáme Householderovu reflexi, a tedy zároveň hodnotu, která má tvořit diagonálou matice R.

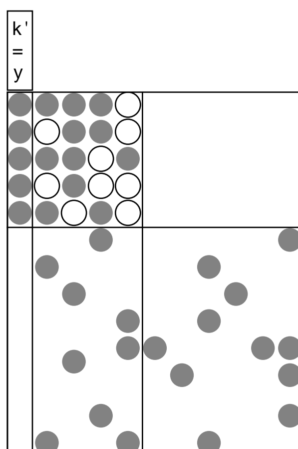
Algoritmus 2.5 QR faktorizace s úpravami dle potřeby

Vstup: $A[m,n]$
Výstup: $R[n,n], V[m,n]$, Beta$[n]$
  for $k = 1$ to n **do**
    $x \leftarrow A[*,j]$
    for $j = 1$ to $k-1$ **do**
      $v \leftarrow V[j:m,j]$
      $\beta \leftarrow$ Beta$[j]$
      $x[j:m] \leftarrow x[j:m] - v\ \beta v^T x[j:m]$
    end for
    $v, \beta, \sigma \leftarrow$ householder$(x[k:m])$
    $V[k:m,k] \leftarrow v$
    Beta$[k] \leftarrow \beta$
    $R[1:k-1,k] \leftarrow x[1:k-1]$
    $R[k,k] \leftarrow \sigma$
  end for

2.4.3.3 Výplň nenul v QR faktorizaci

Podívejme se nyní na řídkou QR faktorizaci. Householderův vektor v volíme až na první souřadnici rovný prvnímu sloupci podmatice A'. To znamená, že má i stejný nenulový vzor. Tento poznatek je zásadní pro určení průběžného nenulového vzoru a pro výběr pivotního sloupce v každém kroku (sloupcové permutace). Klíčový je obrázek 2.10, který zobrazuje aktuální podmatici A' v libovolném kroku k faktorizace.

Podmatice A' je zde zobrazena permutovaná po řádcích tak, aby sloupec k' (aktivní část sloupce k) měl všechny nenuly na začátku, a zároveň po sloup-



Obrázek 2.10: Výplň A' v jednom kroku QR faktorizace

cích tak, aby byly více vlevo ty sloupce, které sdílí se sloupcem k' libovolnou nenulu. Jsou tak lépe vidět čtyři vzniklé bloky upravované matice, permutace ve skutečnosti neprovádíme. Horní blok představuje řádky, které mají nenulu ve sloupci k', dolní blok ty, které nemají. Levý blok představuje sloupce, které sdílí nenulu se sloupcem k', pravý blok ty, které nesdílí.

Levý horní blok je zcela vyplněn nenulami, ostatní tři zůstávají touto Householderovou reflexí nedotčeny, strukturálně i numericky. Levý horní blok je celý doplněn, protože úprava sloupců se řídí rovnicí 2.32 a od těchto sloupců tedy odčítáme násobek Householderova vektoru v, který má v těchto souřadnicích samé strukturální nenuly (připomeňme, že má stejný nenulový vzor jako sloupec k'). Dolní bloky zůstávají nedotčeny, protože od každého sloupce odčítáme skalární násobek vektoru v, který má v těchto souřadnicích nuly. Pravé bloky zůstávají nedotčeny, protože $v^T y$ je zde rovno nule (není sdílena žádná nenulová hodnota), čili od celých těchto sloupců odčítáme nulový násobek vektoru v, neboli neodčítáme nic.

Srovnáme-li tuto výplň s výplní v LU faktorizaci (viz sekci 2.4.2.2), vidíme, že zatímco v LU faktorizaci byl nenulový vzor pivotního sloupce rozkopírován do těch sloupců, které s ním sdílely nenulu v pivotním řádku, zde je rozkopírován do těch, které s ním sdílejí libovolnou nenulu. Proto je výplň v QR faktorizaci horní mezí výplně v LU. Jako kdybychom za pivotní řádky v tomto kroku zvolili najednou všechny ty, které mají ve sloupci k nenulu.

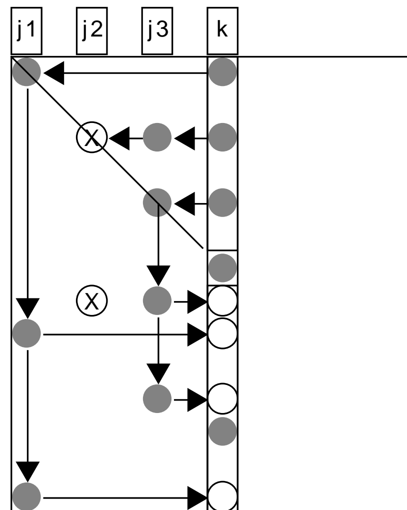
V každém kroku algoritmu 2.5 potřebujeme vědět, které předchozí Householderovy reflexe na sloupec k aplikovat, abychom využili řídkosti. k tomu potřebujeme znát dopředu nenulový vzor R – paradoxně, protože R je až výsledkem faktorizace. Jelikož víme, že $A = QR$ a také $A^T = R^T Q^T$, zjistíme,

že

$$A^T A = R^T Q^T Q R = R^T R, \tag{2.36}$$

protože Q je ortogonální a proto $Q^T Q = I$. $A^T A$ je symetrická, pozitivně definitní matice a $R^T \cdot R$ je tedy její Choleského faktorizace. Nenulový vzor Choleského faktoru najít umíme, a to na základě eliminačního stromu vstupní matice. To znamená, že nenulový vzor matice R stojí na základě eliminačního stromu matice $A^T A$, takzvaného sloupcového eliminačního stromu. Spočítat $A^T A$ by bylo zdlouhavé, naštěstí existuje postup, jak odvodit sloupcový eliminační strom přímo z matice A v téměř lineárním čase vůči počtu nenul $|A|$ [17]. Tím je vysvětleno i tvrzení ze sekce o LU faktorizaci, že je nenulový vzor matice U shora omezen sloupcovým eliminačním stromem $A^T A$.

Nenulový vzor R tedy dopředu známe a v kroku k víme, které předchozí Householderovy reflexe se sloupce dotknou. Jsou to ty z předchozích kroků j, kde v grafu R vede hrana přímo z k do j. Je-li j nepřímým potomkem k, aplikaci vynecháme. Na obrázku 2.11 je vykřížkována předchozí reflexe j_2, která není aplikována, protože není přímým potomkem. Její nenulový vzor se totiž již dříve rozšířil do sloupce j_3 a ten jsme aplikovali.



Obrázek 2.11: Aplikace předchozích Householderových reflexí na řádek k

Pěknou dodatečnou vlastností tohoto výběru je, že jsou nenulové vzory vybraných reflexí vždy disjunktní (jinak by nebyly přímými potomky). To znamená, že počet nenul sloupce k bude přesně roven součtu počtů nenul aplikovaných sloupců j a existujících nenul ve sloupci k. Víme-li tento počet dopředu, můžeme správně připravit řídké datové struktury.

2.4.3.4 Multifrontální přístup

Multifrontální pohled na problém QR faktorizace je obdobný tomu z LU faktorizace rozebranému v sekci 2.4.2.3. Hustou frontální podmatici zde tvoří levý horní blok obrázku 2.10. Do jedné frontální matice opět můžeme vtěsnat více kroků faktorizace, mají-li podobný nenulový vzor ve výsledném R. Datové závislosti jsou dány sloupcovým eliminačním stromem $A^T A$.

2.5 Specifika modulární aritmetiky

Kapitola 2 se dosud zabývala řešením soustav obecně v $\mathbb{R}$ nebo v $\mathbb{C}$. Nyní se podívejme na to, jaká specifika do problematiky přinášejí výpočty v modulární aritmetice. Je patrné, že velká část práce spočívá v symbolické neboli strukturální analýze, která je na numerických výpočtech nezávislá. Nicméně v konečném důsledku je třeba matici soustavy faktorizovat i numericky. Řešený problém má nyní podobu

$$Ax \equiv b \pmod p \tag{2.37}$$

Hlavní potíží je nemožnost použít iterativní metody. Ty jsou založeny na postupném zpřesňování výsledku, tedy postupném přibližování k němu. Okruh celých čísel s prvočíselným modulem ale není spojitý jako $\mathbb{R}$ a není zde snadno definována vzdálenost dvou prvků. Potřebujeme skutečně přesný výsledek a musíme se proto omezit na metody přímé.

Ze stejného důvodu nemá smysl hledat minimalizaci chyby v případě přeurčené matice soustavy pomocí QR faktorizace, jak to bylo popsáno v sekci 2.4.3 – není definována vzdálenost prvků a nelze ji minimalizovat.

Pohybujeme se v okruhu celých čísel modulo prvočíslo s definovanými operacemi sčítání a násobení. Po každém sčítání a násobení dvou čísel je třeba nalézt zbytek po dělení daným modulem. Obě tyto operace jsou tedy výpočetně náročnější, ovšem jen o $O(1)$, asymptoticky se složitost nemění.

Dělení je násobení inverzním číslem. Vzhledem k tomu, že je modul prvočíselný, máme k dispozici inverzi ke všem číslům kromě nuly, stejně jako v $\mathbb{R}$ nelze nulou dělit. Nalézt inverzní prvek čísla a pomocí rozšířeného Euklidova algoritmu má obecně složitost $O(\log a)$, tedy srovnatelnou s $\mathbb{R}$.

Mocnění je opakované násobení a lze provádět například algoritmem binárního umocňování.

Odmocnina představuje problém. Je definována jen pro kvadratická rezidua. V případě prvočíselného modulu p jich je $(p+1)/2$, tedy asi polovina. Na druhou stranu v případě $\mathbb{R}$ je odmocnina definována také jen asi pro polovinu čísel – pro ta nezáporná. Nicméně jak se ukáže, hledaní Householderového vektoru v QR faktorizaci stojí na možnosti spočítat odmocninu ze součtu druhých mocnin, která je v $\mathbb{R}$ možná spočítat vždy, zatímco v modulární aritmetice jen v polovině případů.

O něco snadnější je výběr pivotních prvků. Tento výběr je totiž v případě $\mathbb{R}$ vyvažováním dvou často protichůdných požadavků. Jednak chceme, aby v pivotním řádku a sloupci bylo co nejméně nenul ve snaze minimalizovat výplň. Zároveň se ale snažíme, aby měl pivot numericky co největší absolutní hodnotu pro minimalizaci chyby výpočtu v plovoucí řádové čárce. Druhý požadavek v modulární aritmetice vypadává a tak můžeme o to lépe plnit ten první. Navíc lze teoreticky symbolická analýza lépe oddělit, protože výběr pivotu nezávisí na jeho numerické hodnotě.

Další výhodou modulární aritmetiky je nemožnost přetečení. Za předpokladu, že máme k dispozici datový typ dostatečné velikosti, aby se do něj v případě modulu p vešla druhá mocnina čísla $p - 1$, můžeme provádět libovolné operace bez ztráty přesnosti, přetečení nebo podtečení. Jen je třeba dávat pozor při odčítání. Chceme-li odčítat $a - b$, jedná se o přičítání záporného čísla, tedy čísla $p-b$, a výpočet by měl být $(a+p-b) \mod p$. Kdybychom počítali $(a - b) \mod p$, mohlo by dojít k podtečení.

Kapitola 3

Vlastnosti soustav vznikajících v index calculu

V kapitole 2 jsme se zabývali řešením řídké soustavy lineárních kongruencí $Ax \equiv b \pmod{p}$ obecně. Matici A jsme přisuzovali jen vlastnost řídkosti, tedy konstantní počet nenulových hodnot v každém sloupci, celkem $O(n)$ nenul. Vektoru pravé strany b jsme nepřiřazovali vlastnosti žádné, mohl být řídký i hustý, lišily by se jen algoritmy použité pro řešení dolních a horních trojúhelníkových soustav. Modul p jsme předpokládali prvočíselný.

Tato kapitola se věnuje konkrétním specifickým vlastnostem soustav, které mohou vzniknout ve fázi 1.4.2 index calculu. Soustředit se budeme na variantu pracující v multiplikativní grupě konečného tělesa $GF(2^n)^*$. Znalost těchto vlastností pak v kapitole 4 využijeme k analýze možnosti využití popsaných algoritmů.

3.1 Co soustava vyjadřuje

Jak bylo popsáno v sekci 1.4.2, sestavujeme soustavu kongruencí rozkladem náhodně vybraných polynomů tělesa na ireducibilní polynomy. Pro každý nalezený polynom, který lze rozložit na součin prvků faktorové báze, máme kongruenci

$$l \equiv \sum_{i=1}^{t} c_i \log_g p_i \pmod{N}, \tag{3.1}$$

kde $\log_g p_i$ je i-tá neznámá a c_i její násobnost. Z každé kongruence 3.1 vznikne jeden řádek matice soustavy a jedna hodnota vektoru pravé strany.

Neznámý vektor soustavy x sestává z logaritmů prvků faktorové báze, jeho velikost je tedy rovna velikosti faktorové báze. Sloupce matice soustavy A představují násobnosti jednotlivých faktorů, tedy jich je rovněž tolik, kolik je prvků faktorové báze. Řádků A je tolik, kolik jsme nalezli kongruencí 3.1,

a musí jich být o něco více než neznámých, protože některé mohou být vzájemně lineárně závislé. Matice A je tedy obdélníková s větším počtem řádků než sloupců a může obsahovat lineárně závislé řádky. Vektor pravé strany b odpovídá též počtu kongruencí a obsahuje nalezené hodnoty l z 3.1.

Soustavu je třeba řešit v několika modulech současně, jak bylo rozebráno v sekci 1.6. Každý z těchto modulů plní roli p v řešeném problému $Ax \equiv b \pmod{p}$ a je prvočíselný.

3.2 Rozměry a hodnost matice soustavy

Matice soustavy je obdélníková a má velikost $m \times n$, kde m je počet vygenerovaných kongruencí a n je velikost faktorové báze $|S|$. Kongruencí generujeme více než neznámých, protože nevíme, které jsou vzájemně lineárně závislé, a je nutné, aby měla matice hodnost rovnu počtu neznámých a soustava tak měla jen jedno řešení. Situace, kdy by byla matice přeurčená a soustava neměla žádné řešení, nastat nemůže, protože generované kongruence si nemohou protiřečit.

Ověřovat u každé nalezené kongruence její lineární nezávislost by bylo příliš náročné a zvyšovalo by asymptotickou složitost. Jednalo by se v podstatě o $O(n)$ pokusů řešení soustavy, celková asymptotická složitost by vzrostla $n$krát.

Jak tedy zjistit, kdy je kongruencí dostatek? Pro začátek víme, že v každém sloupci musí být alespoň jedna nenulová hodnota, jinak pro příslušnou neznámou nelze nalézt řešení (hodnost je vždy menší než počet neznámých). Jak se ukáže, vlastnosti níže popsané způsobí, že dosáhnout tohoto stavu obyčejně znamená generovat několikanásobně více řádků než sloupců. V takovém nepoměru řádků a sloupců se v našem případě již málokdy stane, že by matice byla nedourčená. Kdyby přeci jen tato vzácná situace nastala, řešič soustavy nahlásí chybu a vygenerujeme více kongruencí. Soustavu tedy budeme řešit vícekrát, ale jen $O(1)$ násobně.

3.3 Statistické vlastnosti faktorizace polynomu

Vygenerovat jednu kongruenci znamená nalézt polynom $g^l(x)$ stupně $k \in 1, \ldots, n-1$ tělesa $GF(2^n)$ rozložitelný na součin polynomů faktorové báze S. Ve faktorizaci tohoto polynomu mají různé faktory různou pravděpodobnost výskytu a to určí řídkost a další vlastnosti soustavy.

3.3.1 Počet ireducibilních faktorů polynomu stupně k

V první řadě nás zajímá rozdělení náhodné veličiny, kterou je počet faktorů náhodného polynomu stupně k, protože tolik budeme očekávat nenul v kaž-

dém řádku matice soustavy. Odvození tohoto rozdělení vyžaduje znalost tzv. generujících funkcí.

Generující (nebo též vytvořující) funkce určité posloupnosti je taková polynomiální funkce $A(x)$, jejíž koeficienty u jednotlivých členů tvoří právě tuto posloupnost. Jednoduchým příkladem budiž posloupnost, kde pro prvek a_n platí

$$a_n = 2^n - 1. \tag{3.2}$$

Generující funkce takové posloupnosti je

$$A(x) = (2-1)x + (2^2-1)x^2 + (2^3-1)x^3 + \ldots = \frac{x}{(1-x)(1-2x)}, \tag{3.3}$$

kde, jak vidíme, koeficienty u jednotlivých členů tvoří prvky posloupnosti. To znamená, že chceme-li znát například $i$tý prvek posloupnosti, podíváme se na koeficient členu x^i.

Pojem jsme definovali poněkud naruby, smysl generujících funkcí je totiž právě v tom, že není třeba znát přímo vzorec pro $n$tý prvek, jako jsme jej popsali v 3.2. Konečnou podobu generující funkce zapsanou v 3.3 lze odvodit například z induktivního zápisu $a_{n+1} = 2a_n + 1$ a pak lze s generujícími funkcemi různě zacházet, aniž bychom se přímý vzorec 3.2 kdy dozvěděli [18].

Generující funkce může mít i více proměnných. U funkce $A(x, y)$ nás například může zajímat koeficient u členu $x^i y^j$.

Vraťme se k hledanému rozdělení. Pro počet polynomů stupně n majících k ireducibilních faktorů využijeme generující funkce

$$P(x, y) = \prod_{j \geq 1} \left(1 - xy^j\right)^{-I_j}, \tag{3.4}$$

kde I_j je počet monických ireducibilních polynomů stupně j (1.9). Hledaný počet vyčteme z generující funkce 3.4 jako koeficient u členu $x^k y^n$.

Na základě dvojí derivace generující funkce 3.4, zafixování $x = 1$ a asymptotické analýzy funkce lze odvodit střední hodnotu hledaného počtu rovnu $\log k$ se směrodatnou odchylkou $\sqrt{\log k}$ a normálním rozdělením [19].

Střední hodnota $\log k$ je vztažena ke stupni náhodného polynomu, který může být libovolný od 1 až do $n-1$. Nicméně $k = n-1$ s pravděpodobností $1/2$ a například $k = n-7$ s pravděpodobností přes 99%, proto můžeme pro velká n nerovnost mezi k a n zanedbat a za odhad počtu nenul v řádku A považovat $\log n$.

3.3.2 Pravděpodobnost výskytu faktoru stupně l

Zjistili jsme, že v každém řádku můžeme očekávat asi $\log n$ nenulových hodnot. Dále nás zajímá, jak jsou tyto nenuly v řádku rozmístěny. Ukáže se, že rozložení není rovnoměrné.

S daným stupněm k náhodného monického polynomu a s rostoucí prvočíselnou charakteristikou tělesa q se rozdělení ireducibilních faktorů stupňů $1, 2, 3, \ldots$ asymptoticky blíží rozdělení cyklů délek $1, 2, 3, \ldots$ v náhodné permutaci k prvků [20]. Pro $k \to \infty$ se toto rozdělení cyklů asymptoticky blíží nezávislým Poissonovým rozdělením s parametry $1, \frac{1}{2}, \frac{1}{3}, \ldots$. V kombinaci tedy platí, že pro $q, k \to \infty$ jsou rozdělení ireducibilních faktorů stupňů $1, 2, 3, \ldots$ asymptoticky nezávislá Poissonova rozdělení s parametry $1, \frac{1}{2}, \frac{1}{3}, \ldots$. V našem případě je $q = 2$, ale lepší odhad neznáme.

Poissonovo rozdělení veličiny X lze vyjádřit pro všechny hodnoty $x = 0, 1, 2, \ldots$ jako

$$P(X = x) = \frac{\lambda^x}{x!} e^{-\lambda}. \tag{3.5}$$

Nás zajímá pravděpodobnost, že má náhodný polynom alespoň jeden faktor stupně l, tedy opačného jevu, než že nemá žádný faktor stupně l. Pravděpodobnost, že náhodný polynom nemá faktor stupně l je přibližně $P(X = 0) = e^{-(1/l)}$, čili pravděpodobnost opačného jevu je přibližně

$$P(X > 0) = 1 - e^{-(1/l)}. \tag{3.6}$$

Pro velká l je 3.6 přibližně rovno $1/l$, což je pravděpodobnost tohoto jevu zmíněná v [21].

Vidíme, že faktory vyšších stupňů se vyskytují s větší pravděpodobností než faktory nižších stupňů. Ireducibilních polynomů vyššího stupně je navíc větší počet. Nenulové hodnoty jsou tedy v matici výrazně sesypané k jedné straně. Pravděpodobnost výskytu faktoru stupně jedna je například více než poloviční. V $GF(2^n)$ jsou ireducibilní faktory stupně jedna přesně dva, z nichž každý se vyskytne častěji než v každém čtvrtém řádku.

Díky heuristice přibližného nejnižšího stupně sloupce by ale mělo být možné permutovat matici tak, aby hustší sloupce byly voleny později a výplň byla co nejmenší.

Co se týče hodnot v matici, tedy násobností faktorů, budou většinou rovny jedné. Pravděpodobnost, že má náhodný polynom alespoň dva faktory stupně l je

$$P(X > 1) = 1 - e^{-(1/l)} - \frac{1}{l} e^{-(1/l)}, \tag{3.7}$$

což v případě faktorů stupně jedna znamená asi čtvrtinovou pravděpodobnost, ale ještě se musejí oba faktory shodovat na jednom ze dvou možných, čili pravděpodobnost větší hodnoty v matici než jedna je v tomto případě $1/6$. U faktorů vyšších stupňů se toto číslo prudce snižuje. Například pro $l = 3$ jsme již na pravděpodobnosti okolo 4% a ještě se opět musejí faktory shodnout na jednom ze dvou možných, dohromady je tedy pravděpodobnost jen okolo 3%.

3.3.3 Nejvyšší stupně faktoru

Větší řídkost matice soustavy k jedné straně podporuje také očekávaný $i$tý nejvyšší stupeň faktoru ve faktorizaci náhodného polynomu stupně k. Ten je roven $c_i k$, kde c_i je konstanta, která pro je první tři i rovna

$$\begin{aligned} c_1 &\approx 0.62433 \\ c_2 &\approx 0.20958 \\ c_3 &\approx 0.08831 \end{aligned} \tag{3.8}$$

[21].

3.4 Příklad v $GF(2^{256})$

Ukažme si statistické vlastnosti na konkrétním příkladu v $GF(2^{256})$. Podle 1.13 bude nejvyšší stupeň faktoru ve faktorové bázi ideálně přibližně

$$\approx 0.57 \cdot \sqrt{256 \cdot \log 256} \approx 21, \tag{3.9}$$

sloupce matice tedy budou představovat všechny ireducibilní polynomy stupně nejvýše 21. Těch je podle 1.9

$$\sum_{j=1}^{21} I_j = 210871, \tag{3.10}$$

matice tedy bude mít 210871 sloupců. Počet řádků bude větší a bude záležet na tom, kdy dosáhneme plné hodnosti.

Co se týče počtu nenul na řádku, platí v $GF(2^{256})$ pro počet faktorů náhodného polynomu normální rozdělení se střední hodnotou přibližně

$$\log 256 \approx 5.54518, \tag{3.11}$$

čili očekáváme počet nenul na řádku v řádu jednotek.

Střední hodnoty tří nejvyšších stupňů faktorů náhodného polynomu vycházejí v tomto případě

$$\begin{aligned} c_1 n &\approx 0.62433 \cdot 256 = 159.82848 \\ c_2 n &\approx 0.20958 \cdot 256 = 53.65248 \\ c_3 n &\approx 0.08831 \cdot 256 = 22.60736 \end{aligned} \tag{3.12}$$

Každá ze středních hodnot je větší než m, čili průměrný náhodně vybraný polynom nepůjde faktorizovat na faktory z faktorové báze a střední hodnoty nám o matici nic neříkají. Víme jen to, že četnost výskytu jednotlivých stupňů faktorů l se bude řídit přibližně 3.6.

3.5 Model pro náhodné generování soustav

Na základě statistických vlastností rozebraných v této kapitole lze sestrojit algoritmus, který generuje náhodné soustavy rovnic. Ty pak můžeme používat pro testování řešících algoritmů. Výhodou náhodného algoritmu oproti skutečnému generování patřičných kongruencí je jeho podstatně menší složitost.

V každém kroku algoritmu 3.1 je generován jeden řádek i matice soustavy a jedna souřadnice vektoru pravé strany. Na základě 3.5 algoritmus rozhodne, kolik faktorů kterého stupně bude existovat (c). Dále tento počet (většinou roven nule, výjimečně jedničce a téměř nikdy více) rovnoměrně rozloží mezi sloupce příslušící faktorům daného stupně. Souřadnice vektoru pravé strany je libovolná.

Vstupem algoritmu je n určující pracovní těleso $GF(2^n)$. Algoritmus generuje řádky, dokud neexistuje alespoň jedna nenulová hodnota v každém sloupci.

Algoritmus 3.1 Generování náhodné soustavy rovnic vlastností příslušících index calculu

Vstup: n
Výstup: A, b
  $m \leftarrow \lceil 0.57 \cdot \sqrt{n \log n} \rceil$
  $i \leftarrow 1$
  while $i \leq I_{1:m}$ || ne každý sloupec obsahuje hodnotu **do**
    for $l = 1, \ldots, m$ **do**
      $c \leftarrow random_poisson(1/l)$
      $A[i, I_{1:l-1} + 1 : I_{1:l-1} + I_l] \leftarrow uniformly_distributed(c)$
    end for
    $b[i] = random_integer()$
    $i \leftarrow i + 1$
  end while

Počet sloupců matice (v algoritmu 3.1 označen $I_{1:m}$) je podle 1.10 přibližně roven $\frac{1}{m}2^{m+1}$, čili složitost alogirtmu 3.1 je

$$O(|S| \cdot m) = O\left(\frac{1}{m}2^{m+1} \cdot m\right) = O(2^m), \tag{3.13}$$

což je o hodně méně než složitost generování skutečné soustavy pro index calculus, která je dle 1.14 rovna

$$O\left(2^m \left(\frac{n}{m}\right)^{(1+o(1))\frac{n}{m}}\right). \tag{3.14}$$

Kapitola 4

Použitelnost algoritmů pro problém daných vlastností

V kapitole 2 jsme si ukázali tři hlavní algoritmy pro řešení soustavy lineárních rovnic: Choleského faktorizaci, LU faktorizaci a QR faktorizaci. Algoritmy jsme představovali obecně, bez ohledu na postup index calculu. V kapitole 3 jsme se soustředili na konkrétní vlastnosti soustav, které jsou v případě index calculu potřeba řešit. Nyní je čas dát tyto dvě kapitoly do souvislosti a analyzovat možnost použití každého ze tří algoritmů ve fázi 1.4.2 index calculu.

4.1 Choleského faktorizace

Základním předpokladem použití Choleského faktorizace je symetrická pozitivně definitní vstupní matice A. Jak se ukázalo v kapitole 3, matice je hustší směrem k jedné straně a řidší směrem k druhé. Vzhledem k tomu, že každý řádek je generován za stejných podmínek, vzniká podobnost spíše mezi řádky než mezi protilehlými hodnotami přes diagonálu. Symetrie podél diagonály bude velice nízká a Choleského faktorizaci není možné použít. Její zmínka byla nicméně nutná pro úvod do problematiky, jelikož na ní lze nejnázorněji popsat využití řešiče dolních a horních trojúhelníkových soustav, výplň nenul a vznik a význam eliminačního stromu.

4.2 LU faktorizace

Jedinou podmínkou funkční LU faktorizace je regulární čtvercová vstupní matice. Pomocí řádkových permutací je pak vždy možné nalézt řešení. Sloupcové permutace dále pomáhají nalézt řešení, které představuje co nejmenší výplň nenul a je tedy nejrychlejší.

4.2.1 Úpravy

Matice soustavy v index calculu je ale obdélníková, má více řádků než sloupců. Postup LU faktorizace je tedy třeba upravit. V každém kroku symbolické analýzy (je-li oddělena, jinak v hlavním cyklu) hledá libovolný LU algoritmus nejvhodnějšího kandidáta na pivotní prvek, čili hledá pivotní řádek a pivotní sloupec v jejichž průniku leží v daném kroku nenulový prvek (vytváří řádkovou a sloupcovou permutaci). Nabízí se možnost vybírat pivotní řádky z množiny celkového většího počtu řádků a nepoužité řádky nakonec zahodit. Za předpokladu, že vždy vybereme správně i příslušnou souřadnici vektoru pravé strany, vybíráme prostě dostatečný počet rovnic, které všechny současně platí, a výsledek bude správný (měla-li vstupní matice hodnost rovnu počtu neznámých – jinak v určitém kroku nastane situace, kdy algoritmus nenalezne žádný nenulový pivotní prvek).

Algoritmy LU faktorizace by musely být dále upraveny tak, aby pracovaly současně v několika různých modulech. To znamená, že by řešily současně několik úloh strukturálně totožných, ale numericky odlišných. V praxi to znamená v každé řídké datové struktuře ukládat pro dané souřadnice několik hodnot, jednu pro každý modul. Symbolická analýza je prováděna jen jednou, výpočty jednou pro každý modul.

4.2.2 Používané operace

Ještě zbývá ověřit, zda některý krok algoritmu nenarazí na překážku způsobenou zvláštnostmi počtu v modulární aritmetice (viz 2.5).

Použijeme-li verzi algoritmu popsanou v 2.4.2.1, během samotné faktorizace provádíme opakovaně řešení dolní trojúhelníkové soustavy. To obnáší operace odčítání, násobení a dělení. Dáme-li si pozor na podtečení při odčítání a přijmeme-li logaritmickou složitost dělení (hledání inverzního prvku), není zde žádný problém.

Multifrontální metoda je v principu strukturální. Všechny numerické výpočty jsou prováděny uvnitř hustých frontálních podmatic. Zde dochází vždy k několika krokům husté LU faktorizace. Ta spočívá též v dělení, násobení a odčítání. I tento algoritmus by tedy mělo být možné použít.

Použití faktorů L a U pro řešení soustavy pak spočívá v řešení jedné dolní a jedné horní trojúhelníkové soustavy. Ani zde tedy není problém.

4.2.3 Vhodnost použití vzhledem k vlastnostem soustavy

Co se týče vhodnosti metody pro soustavu vzniklou v index calculu, víme, že matice soustavy bude mít nenulové hodnoty sesypané k jedné straně a že řádky budou vzájemně podobného charakteru. Použitím heuristiky přibližného nejmenšího stupně by mělo být možné vybírat pivotní sloupce v pořadí od nejřidšího po nejhustší. V těch prvních vybraných by pravděpodobně byla vždy jen jedna nenulová hodnota. Tím by mělo být možné dosáhnout toho,

aby výplň nenul byla velice pozvolná a posílila až v posledních sloupcích, které jsou již na vstupu mezi nejhustšími. Řádky jsou vzájemně podobného charakteru, tedy by postup jejich výběru neměl mít velký vliv, jen je třeba, aby pro každý sloupec byl vybrán ten řádek, kde je v daný okamžik nenulový prvek.

4.3 QR faktorizace

QR faktorizace je ještě obecnější než LU faktorizace, protože je stavěná pro práci s nedourčenými i přeurčenými maticemi včetně obdélníkových. Pro index calculus by se tedy mohla jevit jako nejvhodnější.

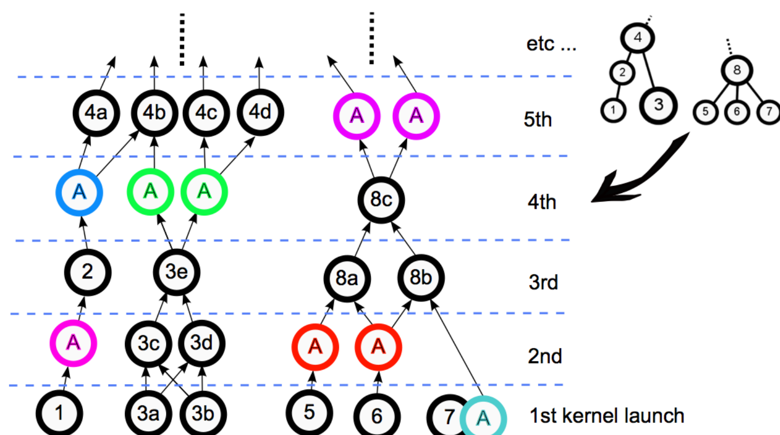
4.3.1 Existující GPU implementace

Tato práce má zkoumat možnost využití GPU pro urychlení výpočtu. Ke QR faktorizaci existuje pečlivě propracovaná a zdokumentovaná implementace pro GPU, jejíž autorem je prof. Tim Davis a další [2]. Je součástí balíčku SuiteSparse dostupného volně z internetu.

Davisova GPU implementace je založena na multifrontální metodě. Na základě připraveného kontribučního stromu malých hustých frontálních podmatic vytváří algoritmus na CPU seznamy úloh, které pak hromadně předává do jednotlivých spuštění GPU kernelu. Plánovač tyto úlohy navrhuje tak, aby postupně frontální matice řešily a předávaly kontribuční bloky do dalších na nich závislých frontálních matic. Frontální matici Davisův algoritmus rozděluje na dlaždice velikosti 32×32 hodnot, nahlíží na ni jako na matici velikosti $m \times n$ dlaždic a faktorizuje ji po částech hledáním částečných Householderových reflexí.

Úlohy pro GPU kernel jsou tří druhů. Jednu skupinu tvoří úlohy faktorizační, které faktorizují blok v ideálním případě 3×1 dlaždic. Druhou tvoří úlohy upravovací, které upravují části sloupců napravo od nedávno faktorizovaného bloku. Třetí skupinu tvoří úlohy skládací, které do aktuální frontální matice skládají kontribuční bloky matic předcházejících. Ukázka postupné paralelní faktorizace kontribučního stromu v několika spuštěních GPU kernelu je na obrázku 4.1. Faktorizační a upravovací úlohy jsou zobrazeny černě, skládací úlohy jsou barevné s písmenkem A uprostřed. Menší frontální matice, mj. 5 a 6, jsou faktorizovány v jenom spuštění kernelu, větší matice jako např. 3 a 4 vyžadující více spuštění.

Plánování úloh pro jednu frontální matici probíhá tak, aby v jeden okamžik vždy probíhala práce na největším možném počtu dlaždic a aby byl co nejvíce využíván masivní paralelismus GPU. Příklad faktorizace ve 12 spuštění GPU kernelu je na obrázku 4.2. Hustá matice je nejprve implicitně permutována tak, aby byla co nejblíže hornímu trojúhelníkovému tvaru. V prvním spuštění jsou řešeny tři úlohy typu faktorizace 3×1 dlaždic, na obrázku označené červenou, fialovou a zelenou barvou. V dalším spuštění jsou Householderovy reflexe z těchto faktorizací aplikovány na zbytek sloupců napravo a současně je



Obrázek 4.1: Multifrontální kontribuční strom osmi frontálních matic a jeho faktorizace jednotlivými spouštěními GPU kernelu [2]

prováděna další faktorizace (světle modrá), jejíž dva řádky nejsou navazující. Ta je aplikována ve třetím spuštění, zatímco vznikají další faktorizace (žlutá a tmavě modrá). Takto se postupuje, dokud není faktorizována celá frontální matice.

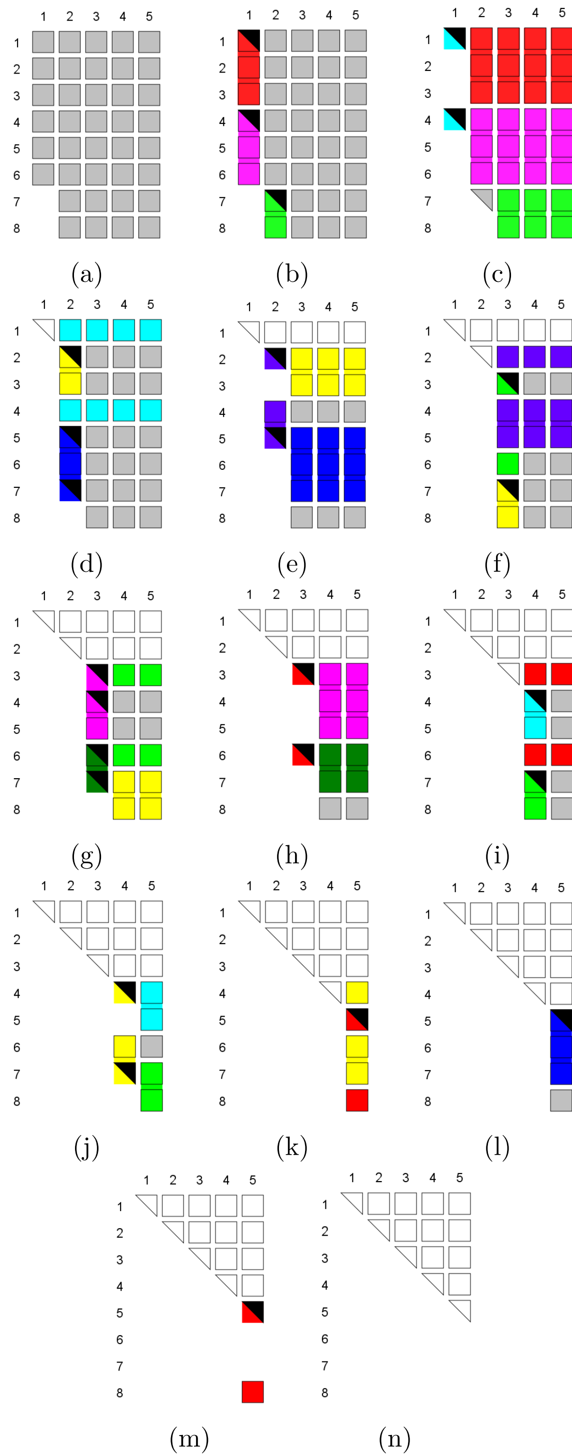
4.3.2 Neúspěšný převod do modulární aritmetiky

Davisův algoritmus pracuje nad reálnými čísly, ale převod do práce s čísly celými modulo prvočíslo se zdá přímočarý. V rámci své semestrální práce pro předmět *MI-PRC* na *FIT ČVUT* jsem se zabýval převedením jednoho z GPU podkernelů, konkrétně toho faktorizačního, do modulární aritmetiky s cílem v budoucnu převést celý algoritmus. Jenže je zde nepřekonatelná překážka již zmíněná v sekci 2.5.

Problém je v generování Householderových reflexí. Operaci je třeba připravit tak, aby vynulovala v daném vektoru všechny souřadnice až na tu první, aniž by změnila jeho normu. Tato norma se potom stane první souřadnicí. Jenže normu počítáme tak, že sečteme druhé mocniny (nenulových) prvků a pak spočítáme odmocninu. A jak jsme rozebrali v 2.5, odmocnina bude definována jen v polovině případů. V každém kroku je poloviční pravděpodobnost, že druhá odmocnina nebude existovat, čili pravděpodobnost, že taková situace nastane někde v průběhu algoritmu, se blíží 100%.

Kdyby operace měnila normu vektoru, její matice by nebyla ortogonální a neplatily by vzorce pro řešení pomocí QR faktorizace popsané v sekci 2.4.3. Navíc by pak neplatil vzorec 2.36 popisující nenulový vzor R a QR metoda by nemohla snadno využít řídkosti.

QR faktorizaci tedy v okruhu celých čísel modulo prvočíslo patrně použít nemůžeme a nezbývá nám, než se uchýlit k LU faktorizaci.



Obrázek 4.2: Faktorizace frontální matice [2]

Kapitola **5**

Realizace LU faktorizace

V kapitole 4 jsme rozebrali, proč je LU faktorizace pro řešení soustavy v index calculu ze tří možností jediná použitelná. V této kapitole nejprve popíšeme existující implementace LU řešičů a možnost jejich převedení do modulární aritmetiky a na GPU. Druhá část kapitoly pak bude sloužit jako dokumentace k provedené realizaci.

5.1 Existující implementace

Věnovat se budeme tří existujícím implementacím LU faktorizace. CSparse, relativně jednoduchému balíku funkcí nad řídkými maticemi. UMFPACK, robustní implementaci multifrontální metody používané v MATLABu. A nakonec se budeme věnovat teoretické práci zabývající se CPU-GPU hybridní implementací multifrontální LU faktorizace.

5.1.1 CSparse

CSparse je knihovna funkcí v jazyce C, která vznikla jako doprovod knihy *Direct Methods for Sparse Linear Systems* of Tima Davise [22]. Jedná se o jednoduchou, čistou a přehlednou implementaci základních operací s řídkými maticemi, včetně Choleského, LU i QR faktorizace. Využívá pouze CPU a nepoužívá multifrontální metody. Je zde dobře čitelný vznik eliminačního stromu, odvození a práce s nenulovými vzory a počítání nenul v řádcích a sloupcích.

Ačkoli je implementace CSparse asymptoticky dokonalá, chybí v ní mnoho mechanismů, které řešení urychlují konstantně, nicméně značně. Mimo jiné proto nebyla zvolena pro MATLAB – tam je použita knihovna UMFPACK popsaná v sekci 5.1.2.

LU faktorizace v CSparse používá metodu popsanou v sekci 2.4.2.1, tedy opakované řešení dolní trojúhelníkové soustavy rovnic, kde část matice soustavy vždy tvoří implicitní nuly a část implicitní jednotková podmatice. Tato

metoda je možná bez problému převést do modulární aritmetiky, ale není vhodná pro implementaci na GPU.

GPU má omezenou paměť, nahrát do ní celý problém není možné, je třeba nahrávat do ní krok po kroku. Algoritmus je navržen tak, že pro každý krok faktorizace potřebuje celý dosavadní faktor L, to znamená, že ke konci algoritmu by musela v globální paměti GPU být nahraná téměř celá matice L a paměť by nestačila. Navíc bychom prováděli obrovské množství kopírování z CPU do GPU, což je pomalá operace.

5.1.2 UMFPACK

UMFPACK je zkratka pro *Unsymmetric Multifrontal Package*. Jedná se o robustní implementaci multifrontální LU faktorizace. Základní postup je zdokumentován v doprovodné práci [16] a zde jej popíšeme, protože je na něm založena implementace, která je součástí této práce. Zabývat se budeme jen faktorizací, následní řešení je poměrně přímočaré a bude popsáno v rámci sekce 5.5.

5.1.2.1 Popis algoritmu

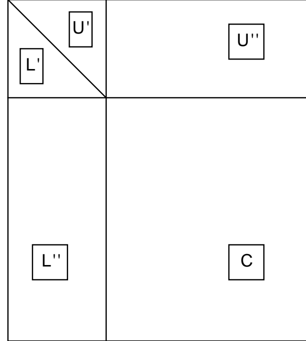
Nejprve je třeba zavést několik pojmů.

Aktivní matice, kterou budeme značit A', je ta část vstupní matice A, která ještě nebyla faktorizována. Skládá se z hodnot nacházejících se ve vstupní matici a hodnot v kontribučních blocích předcházejících frontálních matic, které dosud nebyly sestaveny do matic dalších. Aktivní matice A' není samostatně uložena v paměti, je implicitním složením jiných, které uloženy jsou.

A^k budeme značit zbývající řádky a sloupce matice A v kroku k, které dosud nebyly smazány (přesunuty do nějaké frontální matice).

Aktuální frontální maticí budeme označovat tu, které je právě připravována k faktorizaci, a budeme ji značit F.

Frontální matice, obecně obdélníková, je rozdělena na čtyři bloky dané dělením řádků i sloupců na pivotní a nepivotní. Kolik chceme provést uvnitř frontální matice kroků faktorizace, tolik má pivotních řádků i sloupců. Máme zde podmatici L', což je ta část průniku pivotních řádků a sloupců, která se na výstupu stane součástí faktoru L. Stejně tak máme podmatici U', kterou tvoří druhá část průniku a stane se na výstupu součástí faktoru U. L'' značíme hodnoty pivotních sloupců, které jsou mimo pivotní řádky, a U'' značíme hodnoty pivotních řádků, které jsou mimo pivotní sloupce. Tyto dva bloky se též na výstupu stanou součástí faktoru L a U. Zbytek matice tvoří *kontribuční blok*, který budeme značit C a který bude uložen pro sestavení do pozdějších frontálních matic. Náčrtek frontální matice je na obrázku 5.1. Pivotní a nepivotní řádky a sloupce jsou zde řazeny za sebou, v praxi mohou být ve frontální matici rozmístěny libovolně.



Obrázek 5.1: Náčrtek dělení frontální matice

Nenulový vzor čili strukturu matice nebo vektoru budeme značit kaligrafickým písmem. Ve frontální matici máme $\mathcal{L}$ značící indexy řádků vybrané do frontální matice, jejichž podmnožiny $\mathcal{L}'$ a $\mathcal{L}''$ tvoří pivotní a nepivotní indexy. To samé platí pro $\mathcal{U}, \mathcal{U}'$ a $\mathcal{U}''$ v případě sloupců. Nenulový vzor vstupní matice A značíme $\mathcal{A}$.

Dosud dále nesestavený kontribuční blok spolu s informací o indexech jeho řádků a sloupců nazýváme elementem a číslujeme podle pořadového čísla kroku k, v kterém vznikl. Seznam aktivních elementů udržujeme v množině značené $\overline{V}$.

Tento algoritmus nemá oddělenou symbolickou analýzu od výpočtu. Frontální matice tedy vznikají za chodu. Najdeme vždy jeden globální nenulový pivotní prvek v aktivní matici A', na jeho základě vytvoříme frontální matici, sestavíme do ní řádky a sloupce A^k a předcházející kontribuční bloky, a dále se pokoušíme uvnitř frontální matice nalézt další lokální pivoty, abychom docílili lepšího výkonu provedením několika kroků uvnitř jedné husté matice.

Zjednodušený popis je k vidění v algoritmu 5.1.

Hledání globálního pivotu je založeno na heuristice přibližného nejnižšího stupně. Horní mez stupně každého řádku a sloupce průběžně udržujeme v paměti a upravujeme v každém kroku faktorizace. Pivot najdeme tak, že vybereme čtyři sloupce s nejnižší horní mezí stupně a v nich najdeme pivotní prvek, který má nejnižší Markowitzovu cenu [23] $(\overline{d_r}(r) - 1)(\overline{d_c}(c) - 1)$, kde $\overline{d_r}(r)$ je horní mez stupně řádku r a $\overline{d_c}(c)$ je horní mez stupně sloupce c. Stupněm zde myslíme počet nenul, které ještě nebyly vybrány jako pivotní.

Množina řádků $\mathcal{L}$ a sloupců $\mathcal{U}$ frontální matice F je rovna po řadě nenulovému vzoru vybraného globálního pivotního sloupce a řádku. Frontální matici alokujeme v paměti například dvakrát větší, aby se do ní vešly nenulové vzory případných dalších lokálních pivotů. Tato multiplikativní konstanta udává volnost v míře podobnosti nenulových vzorů pivotních řádků a sloupců,

Algoritmus 5.1 Osnova algoritmu UMFPACK [16]

Vstup: A
Výstup: L, U
1: inicializace
2: **while** faktorizace A nedokončena **do**
3: Najdi globální pivot.
4: Sestav frontální matici F.
5: **while** v matici F je lokální pivot **do**
6: Sestav kontribuční bloky, řádky a sloupce do F.
7: Spočti stupně řádků a sloupců v kontribučním bloku C.
8: Spočti část C.
9: Pokus se najít další lokální pivot.
10: **end while**
11: Dopočti C.
12: Ulož C.
13: **end while**

které jsou vybrány do jedné frontální matice. Větší konstanta by znamenala více práce uvnitř jedné husté matice, ale vzniklo by více zbytečných nenul.

Sestavení frontální matice spočívá v sestavení pivotního řádku a sloupce z A^k a předchozích kontribučních bloků. Ostatní hodnoty jsou prozatím ponechány na svých původních místech a jsou součástí F implicitně, kromě těch, které lze do F sestavit plně, tzn. všechny jejich souřadnice jsou v $\mathcal{L}$ a $\mathcal{U}$, nebo těch, v nichž potřebujeme spočítat přesně počet nenul (především při ověření, zda se vejdou do frontální matice, budou-li vybrány dalšími lokální pivoty).

Numerické úpravy na základě pivotních řádků a sloupců jsou prováděny podle potřeby a pokud možno po větších skupinách.

Detailnější popis je v algoritmu 5.2.

Písmeno $\hat{L}$ v zápisu značí podmnožinu L'', která ještě nebyla aplikována na kontribuční blok C. Stejně tak $\hat{U}$ značí dosud neaplikovanou podmnožinu U''.

Proměnná k značí krok faktorizace a je inkrementována po každém jeho dokončení. V proměnné $\overline{V}$ udržujeme seznam aktivních elementů. Proměnné s a t udávají maximální velikost dané frontální matice a jsou dány multiplikativní konstantou g, která určuje volnost v podobnosti nenulových vzorů pivotních řádků a sloupců v jedné frontální matici.

Pro každý řádek a sloupec matice A' udržujeme seznam elementů, které do něj kontribuují. Tento seznam pro řádek i značíme $\mathcal{R}_i$ a pro sloupec j značíme $\mathcal{C}_j$. Horní meze stupňů sloupců upravujeme v každém kroku pro všechny

Algoritmus 5.2 Základní algoritmus UMFPACK [16]

Vstup: A
Výstup: L, U
inicializace
$k \leftarrow 1$
$\overline{V} \leftarrow \{\}$
while $k \leq n$ **do**
  $e \leftarrow k$
  Najdi $k$tý globální pivot a'_{rc} v aktivní matici A'.
  $\mathcal{L} \leftarrow Struct(A'_{*c})$
  $\mathcal{U} \leftarrow Struct(A'_{r*})$
  $s \leftarrow g|\mathcal{L}|$
  $t \leftarrow g|\mathcal{U}|$
  Alokuj frontální matici F velikosti $s \times t$
  while cyklus nebyl ukončen **do**
    Sestav k-tý pivotní řádek a sloupec do F.
    Projdi seznamy elementů a spočítej externí stupně řádků a sloupců.
    Sestavuj řádky a sloupce A^k, kde to lze kompletně.
    Sestavuj řádky a sloupce z kontribučních bloků, kde to lze kompletně.
    Spočítej nové horní meze stupňů řádků a sloupců.
    Spočti hodnoty v L' a L''
    $k \leftarrow k + 1$
    if $|\mathcal{U}'| \mod b = 0$ **then**
      $C \leftarrow C - \hat{L}\hat{U}$
    end if
    if $|\mathcal{U}''| = 0$ **then**
      Ukonči cyklus.
    end if
    Najdi kandidáta na lokální pivotní sloupec $c \in \mathcal{U}''$.
    $C_{*c} \leftarrow C_{*c} - \hat{L}\hat{U}_{*c}$
    if $\overline{d_c}(c) \neq |\mathcal{L}''|$ **then**
      Sestav sloupec c a spočti přesně $\overline{d_c}(c)$.
    end if
    if $\overline{d_c}(c) > s - |\mathcal{U}'|$ **then**
      Ukonči cyklus.
    end if
    Najdi kandidáta na lokální pivotní řádek $r \in \mathcal{L}''$.
    if kandidát nenalezen **then**
      Ukonči cyklus.
    end if
    if $\overline{d_r}(r) \neq |\mathcal{U}''|$ **then**
      Sestav řádek r a spočti přesně $\overline{d_r}(r)$.
    end if
    if $\overline{d_r}(r) > t - |\mathcal{U}'|$ **then**
      Ukonči cyklus.
    end if
    $\mathcal{L} \leftarrow \mathcal{L} \cup Struct(A'_{*c})$
    $\mathcal{U} \leftarrow \mathcal{U} \cup Struct(A'_{r*})$
    $C_{r*} \leftarrow C_{r*} - \hat{L}_{r*}\hat{U}$
  end while
  Ulož $L', L'', \mathcal{L}, U', U'', \mathcal{U}$
  $C \leftarrow C - \hat{L}\hat{U}$
  $C_e = C$
  $\mathcal{L}_e = \mathcal{L}''$
  $\mathcal{U}_e = \mathcal{U}''$
  Ulož element s indexem e obsaující $C_e, \mathcal{L}_e, \mathcal{U}_e$.
  Dealokuj F.
  $\overline{V} = \overline{V} \cup e$
  Přidej e do seznamů elementů.
end while

sloupce $j \in \mathcal{U}''$ na základě funkce

$$\overline{d_c}(j) = \min \begin{cases} n - k, \\ |\mathcal{L}''| + \overline{d_c}(j), \\ |\mathcal{L}''| + (|\mathcal{A}^k_{*j}| - \alpha_j) + \left(\sum_{e \in \mathcal{C}_j} |\mathcal{L}_e \setminus \mathcal{L}|\right), \end{cases} \tag{5.1}$$

kde α_j je počet řádků, které jsme prošli ve sloupci j matice A^k, než jsme našli první hodnotu, jejíž řádek není ve frontální matici. $\left(\sum_{e \in \mathcal{C}_j} |\mathcal{L}_e \setminus \mathcal{L}|\right)$ značí součet externích sloupcových stupňů všech elementů kontribuujících do sloupce j. Externí sloupcový stupeň je počet řádků, které daný element kontribuuje a nejsou v aktuální frontální matici, proto $\mathcal{L}_e \setminus \mathcal{L}$. Pro výpočet externího stupně existuje efektivní algoritmus popsaný v [16].

Ukažme nyní, že funkce 5.1 skutečně určuje horní mez stupně sloupce $j \in \mathcal{U}''$. První případ omezuje hodnotu tak, že nelze, aby ve sloupci zbývalo více hodnot, než je jeho velikost minus počet již vybraných pivotních řádků, jedná se tedy o případ, kdy je sloupec zcela vyplněný nenulami, o absolutní horní mez. Druhý případ říká, že nemohl počet nenul ve sloupci vzrůst o více než o výplň vzniklou v aktuální frontální matici. Poslední případ je přímým výpočtem a tvoří jej součet tří členů. První člen popisuje přispění aktuální frontální matice. Druhý člen značí nejhorší případ, kdy všechny nenuly zbývající v matici A^k ve sloupci j, kromě těch, které jsme od kraje prošli a shledali, že jsou počítány v $|\mathcal{L}''|$, jsou mimo $\mathcal{L}''$. Poslední popisuje přispění kontribučních bloků předcházejících frontálních matic (elementů).

Pro horní mez stupně řádku $i \in \mathcal{L}''$ platí obdobná funkce

$$\overline{d_r}(i) = \min \begin{cases} n - k, \\ |\mathcal{U}''| + \overline{d_r}(i), \\ |\mathcal{U}''| + (|\mathcal{A}^k_{i*}| - \alpha_i) + \left(\sum_{e \in \mathcal{R}_i} |\mathcal{U}_e \setminus \mathcal{U}|\right). \end{cases} \tag{5.2}$$

Numerickou úpravu kontribučního bloku frontální matice provádíme pouze pro řádky a sloupce, které se mají stát pivotními, nebo pokud máme ve frontální matici počet pivotů dělitelný nějakým parametrem b, jehož výchozí hodnota je 16. Důvodem je, že je numerický výpočet rychlejší, pracuje-li nad větším počtem hodnot současně. Numerická práce je odkládána, jak to jen lze.

Nastane-li situace, kdy po označení nového lokálního pivotu v aktuální frontální matici nepřibudou do matice žádné nové řádky ani sloupce, podstatně se zjednoduší obsah vnitřního cyklu. V takové chvíli stačí snížit horní meze stupňů řádků v $\mathcal{L}''$ a sloupců v $\mathcal{U}''$ o jedna, protože byl vybrán nový pivot, ale nedošlo ke změně výplně nenul.

5.1.2.2 Zvláštní případy a konstantní zrychlení

UMFPACK je v praxi složitější než algoritmus 5.2. Asymptoticky sice funguje stejně, ale používá několik mechanismů navíc, které běh mohou konstantně urychlit.

Především dokáže využít zvláštních případů matic. Například hned odhalí, je-li v nějakém řádku a sloupci jen jedna hodnota, takzvaný jedináček (z anglického *singleton*). Všichni jedináčci jsou přednostně zvoleni jako pivotní a nejsou pro ně formovány frontální matice. Při zvláštním případu (permutované) diagonální matice toto zjednodušení vede k lineární složitosti.

Dále zkoumá míru symetrie vstupní matice a na základě toho umí rozhodnout, zda není vhodnější jiný algoritmus nebo například jiná heuristika volení pivotů.

Symbolická analýza není tak úplně sloučená, jako je tomu v algoritmu 5.2. Část symbolické analýzy se odehrává napřed, jsou provedeny předběžné permutace, jsou odhaleni jedináčci, vybírá se strategie hledání pivotů a nastavuje se mnoho dalších parametrů pro hlavní algoritmus.

5.1.3 CPU-GPU hybridní implementace

Implementací LU faktorizace pro GPU se zabývá práce *A CPU–GPU hybrid approach for the unsymmetric multifrontal method* [24]. Práce se věnuje otázce, jak lze zrychlit algoritmus UMFPACK pomocí GPU. Chce využít skutečnosti, že je GPU vhodná pro základní funkce lineární algebry (*BLAS – Basic Linear Algebra Subroutines*) a že existuje implementace těchto funkcí v programovacím jazyce CUDA pro NVIDIA GPU přímo od výrobce s názvem CUBLAS. Jedná se především o funkce řešení trojúhelníkové soustavy rovnic s jedním nebo více vektory pravých stran, násobení matice vektorem a násobení matice maticí.

Tato úprava algoritmu 5.1 spočívá v tom, že na základě prahové hodnoty počtu pivotů ve frontální matici rozhodne, zda její hustou faktorizaci provést na CPU nebo na GPU. To znamená, že operace na řádku 11 je v některých případech kopírována a provedena na GPU.

Práce dále rozebírá, jak je možné urychlit kopírování mezi CPU a GPU. Věnuje se čtyřem aspektům poskytovaným jazykem CUDA.

Page-locked memory Alokace paměti na straně CPU tak, aby nemohla být odstránkována a GPU mohla znát přesnou fyzickou adresu. Alokovat tímto způsobem velké množství paměti by znamenalo zpomalit práci na straně CPU, nicméně autoři ve svých pokusech sledovali asi 150% zrychlení.

Zero-copy memory Od CUDA verze 3.0 je možné vynechat explicitní kopírování mezi pamětí na straně CPU a pamětí na straně GPU a z CUDA kódu přistupovat zdánlivě přímo do paměti na straně CPU. Jedná se ale

jen o pomůcku, data jsou i tak kopírována a není vhodné tento postup volit, jedná-li se o taková data, která budou znovu používána.

Asynchronous memory copy Použitím různých datových proudů (*stream*) je možné docílit toho, aby GPU prováděla kernel nad jedněmi daty, zatímco jsou jiná data kopírována.

Memory reusing Jedná se o snahu co nejvíce přepoužívat společné bloky paměti mezi jednotlivými úkoly za účelem omezení kopírování.

Na závěr autoři zmiňují možnost úpravy hybridního algoritmu tak, aby od určité prahové velikosti frontální matice provedl veškerý numerický výpočet na GPU, to znamená, aby zkopíroval bloky L', L'', U', U'' a C do globální paměti GPU a numerickou faktorizaci provedl kompletně tam. Toto je postup, kterému se budeme dále věnovat.

5.2 Zvolený postup

Součástí této práce je implementace LU řešiče v modulární aritmetice, který umí pracovat s více prvočíselnými moduly současně a je rozšířen o možnost pracovat nad více řádky než sloupci.

Faktorizační algoritmus vychází z CPU-GPU hybridního návrhu popsaného v sekci 5.1.3. Frontální matici umí faktorizovat jak na CPU, tak na GPU. Funguje tak, že nejprve připraví celou frontální matici k faktorizaci, pak všechny její bloky pošle do GPU kernelu (je-li zvolena GPU faktorizace) a na výstupu kernelu očekává faktorizované bloky.

Řešící algoritmus čerpá z knihovny CSparse zmíněné v sekci 5.1.1 a GPU nevyužívá. Sestává ze dvou permutací, jednoho řešení dolní trojúhelníkové a jednoho horní trojúhelníkové soustavy. I tento algoritmus pracuje s více moduly současně.

Následuje dokumentace k této implementaci.

5.2.1 Datové typy

Protože algoritmy pracují v přirozených číslech s nulou, datový typ, do kterého jsou ukládány hodnoty, je typu celého čísla bez znaménka, jehož velikost je 64 bitů. I indexy matice jsou v principu přirozená čísla (řádky i sloupce číslujeme od nuly) a jsou ukládány ve stejném datovém typu. Totéž platí pro index nebo počet nenulových hodnot v matici. Zde je výňatek ze souboru *types.h*, který základní datové typy definuje.

```
typedef uint64_t ValueIndex;
typedef uint64_t MatrixIndex;
typedef uint64_t Value;
```

Modul, v kterém algoritmus může pracovat, nesmí být větší než 2^{32}, aby žádná hodnota nepřesáhla $2^{32} - 1$ a při násobení dvou hodnot nedošlo k přetečení 64bitového datového typu.

Nedostatkem (a možným budoucím rozšířením) algoritmu je jeho neschopnost pracovat s většími moduly. Pro řád grupy, v které se šifruje, je obyčejně voleno velké prvočíslo nebo číslo složené obsahující velké prvočíslo pro zamezení snadné faktorizace.

5.3 GPU kernel

Tato sekce popisuje GPU kernel pro částečnou LU faktorizaci frontální matice až s šestnácti pivotními prvky. Kernel je implementován v jazyce CUDA pro NVIDIA GPU spolu s jednoduchou sekvenční CPU verzí pro srovnání výkonu. Frontální matici převezme jakožto čtyři bloky a po faktorizaci odpovídající bloky vrátí nazpět. Pracuje s více moduly současně.

5.3.1 Vstupy

Vstupem kernelu je seznam úkolů, jeden pro každý pracovní modul. Úkol má strukturu

```
typedef struct task_descriptor_t {

        Value *pivotBlock;
        Value *pivotRowBlock;
        Value *pivotColBlock;
        Value *contributionBlock;
        int pivotCount;
        MatrixIndex contributionBlockRows;
        MatrixIndex contributionBlockCols;
        Value module;
        int *result;

} TaskDescriptor;
```

První čtyři pole obsahují čtyři vstupní bloky hodnot frontální matice. Na obrázku 5.1 odpovídá proměnná `pivotBlock` čtvercové matici $L'|U'$. Proměnná `pivotRowBlock` obsahuje pivotní řádky bez pivotních sloupců, tedy U'', proměnná `pivotColBlock` je L'', `contributionBlock` je C.

Další tři celá čísla popisují velikost. Počet pivotů je v `pivotCount` (nejvýše 16) a proměnné `contributionBlockRows` a `contributionBlockCols` určují počet řádků a sloupců kontribučního bloku C.

Následuje pracovní modul a ukazatel na celé číslo, kam má být uložena informace o úspěchu faktorizace.

5.3.2 Výstupy

Čtyři vstupní bloky hodnot jsou upraveny na místě, tedy přímo v dodané struktuře, a zde jsou po skončení kernelu k přečtení. Informace o úspěchu či neúspěchu je uložena v paměti, kam ukazuje ukazatel `*result` vstupní struktury, jako celé číslo. Úspěch je značen hodnotou 0. Chyba algoritmu, například nedostatek paměti, je značena zápornou hodnotou. Kladná hodnota i znamená, že při i-tém kroku faktorizace byl pivotní prvek shledán jako numericky nulový. Protože je numerická faktorizace prováděna jen uvnitř kernelu, volající algoritmus dopředu neví, zda tato situace může nastat. Nastane-li, algoritmus na straně CPU označí ve frontální matici pivoty pořadových čísel $\geq i$ jako nepivotní. Tyto kroky faktorizace pak budou provedeny v některé budoucí frontální matici.

Nulový pivot se nemůže objevit v nultém kroku, protože volající algoritmus ví, jaká hodnota zde je, a kontrolu na její nulovost provede sám. Proto lze o úspěchu informovat hodnotou 0.

5.3.3 Popis algoritmu

Algoritmus částečné LU faktorizace vstupní frontální matice spočívá v jejím rozkladu na následující součin, psáno blokově:

$$\begin{bmatrix} P & Q \\ R & C \end{bmatrix} = \begin{bmatrix} L' & 0 \\ L'' & C - L''U'' \end{bmatrix} \begin{bmatrix} U' & U'' \\ 0 & I \end{bmatrix}, \tag{5.3}$$

kde jsme původní nefaktorizované bloky pro přehlednost přeznačili na P, Q, R a C. Oba faktory můžeme ponechat v původní matici ve formě

$$\begin{bmatrix} L'|U' & U'' \\ L'' & C - L''U'' \end{bmatrix}, \tag{5.4}$$

což je výstupní matice algoritmu kernelu. Dolní trojúhelníková matice L' a horní trojúhelníková matice U' mohou sídlit v jedné matici čtvercové s tím, že diagonála patří matici U', protože L' má diagonálu implicitně jednotkovou.

Z rozkladu 5.3 odvodíme vzorce pro výpočet faktorů. Na pivotním bloku P je provedena plnohodnotná malá hustá LU faktorizace na faktory L' a U':

$$L'U' = P. \tag{5.5}$$

Pro blok pivotních řádků Q platí

$$L'U'' = Q, \tag{5.6}$$

čili U'' získáme řešením husté dolní trojúhelníkové soustavy s více vektory pravých stran (tvořenými sloupci matice Q).

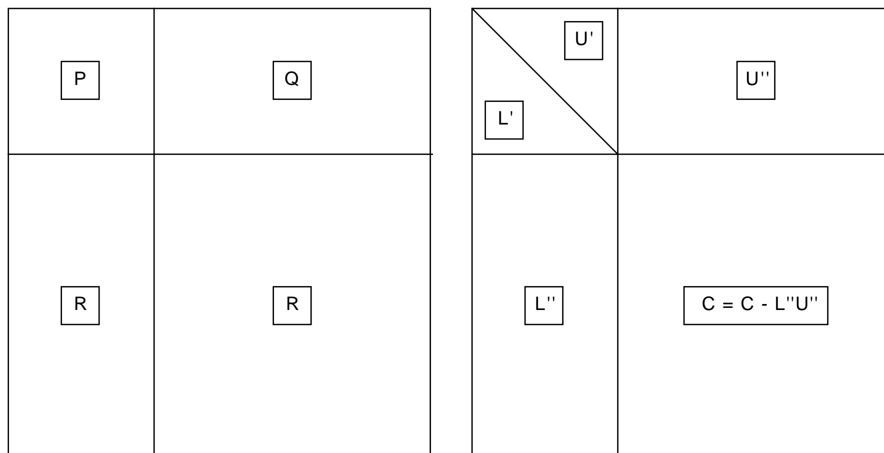
Pro blok pivotních sloupců R platí

$$\begin{aligned} L''U' &= R \\ U'^T L''^T &= R^T \end{aligned} \tag{5.7}$$

a L'' spočítáme opět řešením husté dolní trojúhelníkové soustavy s více vektory pravých stran. Transpozice jsou implicitní.

Na závěr od vstupního kontribučního bloku C odečteme součin matic $L''U''$.

Umístění a transformace bloků v paměti je načrtnuto na obrázku 5.2.



Obrázek 5.2: Vstupní a výstupní bloky GPU kernelu

5.3.4 Sekvenční verze

Sekvenční verze pro každý pracovní modul jednoduše vyřeší čtyři výpočty popsané v sekci 5.3.3, tedy hustou LU faktorizaci, dvě dolní trojúhelníkové soustavy s více vektory pravých stran a jedno násobení matice maticí. Postup je popsán v algoritmu 5.3.

5.3.5 Osnova GPU kernelu

GPU kernel je spouštěn s parametrem počtu bloků vláken a s parametrem počtu vláken v každém bloku. Každý blok vláken má vlastní sdílenou paměť, ke které mají přístup všechna vlákna bloku. Kernel je naprogramován tak, že jeden úkol vždy řeší jeden blok vláken. Bloky vláken vykonávají podobný objem práce, jen každý v jiném modulu. Pro jednoduchost budeme postup popisovat pro jeden úkol – jeden modul.

Algoritmus 5.3 Sekvenční algoritmus GPU kernelu

Vstup: P, Q, R, C pro každý prvočíselný modul p
Výstup: L', L'', U', U'', C a signalizace úspěchu r pro každé p
for all p **do**
  LU faktorizací převeď $P \rightarrow L'|U' \pmod{p}$.
  for $j \leftarrow 1, \ldots, |Q|$ **do**
    Vyřeš dolní trojúhelníkovou soustavu $L'U''_{*j} \equiv Q_{*j} \pmod{p}$
  end for
  for $i \leftarrow 1, \ldots, |R|$ **do**
    Vyřeš dolní trojúhelníkovou soustavu $U'^T L''^T_{*i} \equiv R^T_{*i} \pmod{p}$
  end for
  $C \leftarrow C - L''U'' \pmod{p}$
end for

Počet vláken je nastaven na maximálních 1024 a práce je mezi ně pokud možno rovnoměrně rozdělována. Počet vláken lze snížit, ale je pak nutné upravit i další parametry. Experimentálně byl ověřen nejvyšší výkon při tomto nastavení.

Osnova kernelu je v algoritmu 5.4.

Algoritmus 5.4 Osnova algoritmu GPU kernelu

Vstup: P, Q, R, C, prvočíselný modul p
Výstup: L', L'', U', U'', C, signalizace úspěchu r
1: První vlákno uloží úkol do sdílené paměti.
2: Bariéra.
3: Faktorizace pivotního bloku P.
4: **if** chyba faktorizace v kroku k **then**
5:   První vlákno uloží na výstup $r \leftarrow k$.
6:   Blok vláken končí.
7: **end if**
8: Úprava pivotních bloků L'' a U''.
9: Bariéra.
10: Úprava kontribučního bloku C.

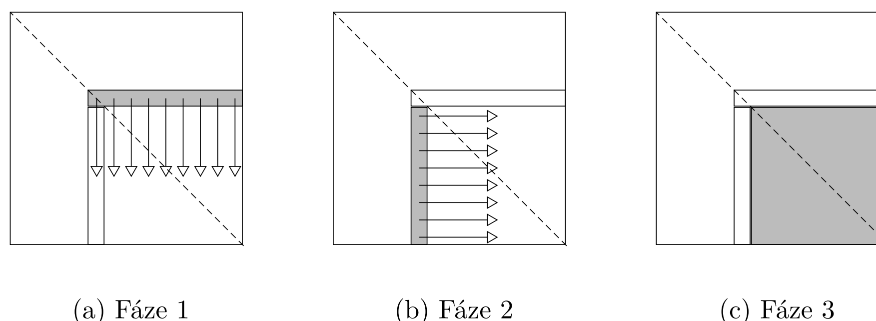
Bariéra na řádku 2 synchronizuje vlákna, protože první z nich právě poslalo informace o úkolu do sdílené paměti a všechna ostatní vlákna jej potřebují mít k dispozici.

Ačkoli úprava na řádku 8 potřebuje znát faktorizovaný pivotní blok P, není zde třeba synchronizace, protože implementace faktorizace P zaručuje, že všechna potřebná data již jsou připravena ve sdílené paměti, kde jsou očekávána. Oproti tomu bariéra na řádku 9 potřeba je, protože úprava L'' a U'' stejnou jistotu neposkytuje.

V následujících sekcích rozebereme podrobněji jednotlivé kroky na GPU.

5.3.6 Faktorizace pivotního bloku P

Na čtvercovém pivotním bloku P velikosti nejvýše 16 je provedena úplná LU faktorizace standardním paralelním algoritmem pro hustou LU faktorizaci, jehož krok k je znázorněn na obrázku 5.3.



(a) Fáze 1 (b) Fáze 2 (c) Fáze 3

Obrázek 5.3: Jeden krok paralelní husté LU faktorizace [3]

V první fázi vlákna, která mají na starosti hodnoty pivotního řádku k, tyto hodnoty vyšlou broadcastem vláknům, která mají na starosti ostatní hodnoty v dosud nefaktorizované podmatici, viz nákres 5.3a. Výjimku tvoří vlákno, které má na starosti pivotní hodnotu na pozici k, k. To spočítá inverzi této hodnoty a posílá inverzi. Toto je úprava pro modulární aritmetiku, kde pro dělení potřebujeme spočítat inverzi.

V druhé fázi vlákna, která mají na starosti hodnoty pivotního sloupce k kromě pivotu jako takového, vynásobí tyto hodnoty přijatou inverzí pivotu z první fáze (obdoba dělení v $\mathbb{R}$). Součin pak vyšlou broadcastem vláknům, která mají na starosti hodnoty napravo, viz obrázek 5.3b.

V poslední fázi, znázorněné na obrázku 5.3c, vlákna mající na starost ostatní hodnoty od každé z těchto hodnot odečtou vždy součin hodnoty přijaté přímo shora a hodnoty přijaté přímo zleva.

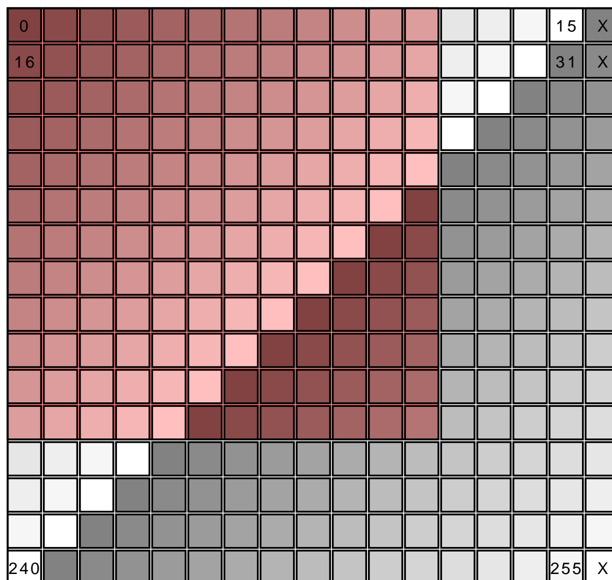
5.3.6.1 Rozdělení práce

Při rozdělování práce mezi vlákna je třeba dávat pozor na konflikty paměťových bank. Ve skupině 16 vláken (*halfwarp*) běžících v jeden okamžik na jednom SM (*streaming multiprocessor*) musí každé přistupovat k jiné bance sdílené paměti, jinak dojde k serializaci paměťových požadavků (výjimku tvoří broadcast, kdy více vláken čte ze stejné banky stejnou adresu). Paměťových bank je 32 (od architektury Fermi) a jsou prokládány s jemností 32 bitů, čili stejná banka je použita každých 16 po sobě jdoucích 64bitových hodnot. Přistupuje-li 16 vláken jednoho halfwarpu k 16 po sobě jdoucím číslům, ke konfliktu nedojde.

Pivotní blok ve sdílené paměti definujeme o jeden sloupec širší, aby hodnoty, které jsou ve stejném sloupci, nebyly ve stejné paměťové bance. Díky

tomu může více vláken jednoho halfwarpu číst různé hodnoty jednoho sloupce. Tato výhoda nebude využita při faktorizaci pivotního bloku, později ale ano (pivotní blok ve sdílené paměti zůstane pro další fázi algoritmu).

GPU kernel je spuštěn s 1024 vlákny, hodnot v P je nejvýše 256, to znamená, že během faktorizace pivotního bloku bude část vláken spát a žádné nebude mít na starosti více než jednu hodnotu. Hodnoty přiřazujeme vláknům po řádcích, a to jak pro práci s globální pamětí, tak pro práci se sdílenou. Nákres pivotního bloku ve sdílené paměti a zodpovědných vláken je na obrázku 5.4. Šestnáct odstínů šedi označuje 16 dvojic paměťových bank, kde jedna dvojice má 64 bitů pro jednu hodnotu matice. Čísla vláken jsou jen naznačena. Na obrázku je patrný i zmíněný sloupec navíc. Červená barva udává příklad pivotního bloku, který je menší (obsahuje jen 12 pivotů namísto 16). Číslování vláken se v takovém případě nemění, jinak by došlo ke konfliktům (druhou hodnotu v prvním sloupci by četlo vlákno 12, které je ve stejném halfwarpu jako vlákno 1, a to by četlo z též paměťové banky druhou hodnotu v prvním řádku).



Obrázek 5.4: Pivotní blok ve sdílené paměti a vlákna zodpovědná za jeho hodnoty

5.3.6.2 Algoritmus

Obecný postup popsaný na začátku této kapitoly je třeba přizpůsobit specifikům grafické karty. Broadcasty znázorněné na obrázku 5.3 se například na GPU řeší zápisem do sdílené paměti a následným hromadným čtením této hod-

noty cílovými vlákny, přičemž nedochází ke konfliktu paměťové banky, jedná se o výjimku zmíněnou výše. Zjednodušený popis je k vidění v algoritmu 5.5.

Algoritmus 5.5 Faktorizace pivotního bloku na GPU

Vstup: P, prvočíselný modul p
Výstup: $P = L'|U'$, signalizace úspěchu r
1: $P_{i,j} \leftarrow global(P_{i,j})$ ▷ Všechna vlákna držící hodnoty z P
2: **for** $k \leftarrow 1, \ldots, |P|$ **do**
3: $shared(pinv[k]) \leftarrow inv(P_{k,k}, p)$ ▷ Vlákno držící $P_{k,k}$
4: Bariéra.
5: **if** inverze selhala **then**
6: $r \leftarrow k$
7: Vlákna bloku končí.
8: **end if**
9: $shared(P_{k,j}) \leftarrow P_{k,j}$ ▷ Vlákna držící $P_{k,k+1:|P|}$
10: $P_{i,k} \leftarrow P_{i,k} \cdot shared(pinv[k]) \pmod p$ ▷ Vlákna držící $P_{k+1:|P|,k}$
11: $shared(P_{i,k}) \leftarrow P_{i,k}$ ▷ Vlákna držící $P_{k+1:|P|,k}$
12: Bariéra.
13: $P_{i,j} \leftarrow P_{i,j} - shared(P_{i,k})shared(P_{k,j}) \pmod p$ ▷ Vlákna držící $P_{k+1:|P|,k+1:|P|}$
14: **end for**
15: $global(P_{i,j}) \leftarrow P_{i,j}$ ▷ Všechna vlákna držící hodnoty z P

Nejprve každé vlákno načte svou hodnotu z globální paměti do registru. Celá matice P je tak uložena v registrech vláken. Poté následuje $|P|$ kroků faktorizace, během nichž čím dál menší počet vláken vykonává práci. Nakonec každé vlákno vrátí svou novou hodnotu zpět do globální paměti.

Pro rychlost přístupů do globální paměti je důležité, aby bylo přistupováno do co největších spojitých oddílů. Zde je matice P z globální paměti a zpět přesouvána celá najednou, což je dobré.

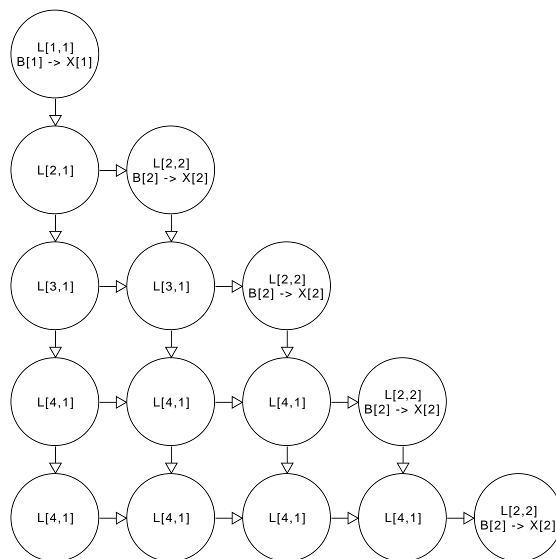
V kroku k faktorizace nejprve vlákno mající na starosti pivotní prvek $P_{k,k}$ spočítá jeho inverzi a uloží ji do sdílené paměti. To je obdoba jednoho z broadcastů na obrázku 5.3a. Následuje bariéra, protože jiná vlákna budou potřebovat číst tento broadcast. Dále vlákna v pivotním řádku uloží svou hodnotu do sdílené paměti, to je též obrázek 5.3a. Tato akce sice mohla začít už před bariérou, ale provádí ji stejný warp jako řádek 3, takže by stejně nezačala, navíc je výhodné ji spustit po boku následující akce pivotního sloupce, která trvá podobně dlouho. Vlákna držící pivotní sloupec zde přečtou ze sdílené paměti inverzi (dokončení broadcastu), vynásobí jím svoji hodnotu a výsledek uloží opět do sdílené paměti, což je obdoba broadcastu na obrázku 5.3b. Následuje další bariéra, protože je třeba, aby byl výsledek předchozích dvou akcí bezpečně uložen ve sdílené paměti. V poslední fázi každé vlákno mající na starost některou z hodnot zbytku dosud nefaktorizovaného pivotního bloku přečte ze sdílené paměti příslušnou hodnotu pivotního řádku a příslušnou hodnotu pivotního sloupce (dokončení broadcastu) a odečte od své hodnoty jejich součin. To je obdoba obrázku 5.3c.

5.3.7 Aktualizace bloků Q a R

Máme-li faktorizovaný pivotní blok P, přijde na řadu úprava bloků pivotních řádků a sloupců Q a R. Tyto dvě operace jsou velice podobné. Jedná se o dvě řešení dolní trojúhelníkové soustavy s více vektory pravých stran. V případě Q tvoří vektory pravých stran sloupce, v případě R je tvoří řádky.

5.3.7.1 Mapování husté dolní trojúhelníkové sestavy

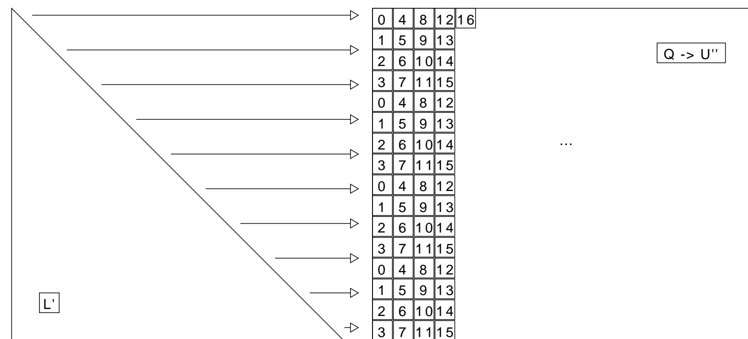
Pro obecný popis budeme používat notaci $Lx = b$. Nejjemnější algoritmus, kde každou hodnotu trojúhelníkové matice soustavy drží právě jedno vlákno, spočívá v kombinaci broadcastů a redukcí. Redukce je prováděna mezi vlákny v jednom řádku k, kde pro spočítání neznámé x_k musíme odečíst od původní hodnoty b_k všechny členy $l_{kj}x_j$. Spočítaná hodnota neznámé v $k$tém kroku x_k je broadcastem vysílána vláknům, která mají na starost hodnoty ve sloupci k, tedy hodnoty l_{ik}, protože budou potřeboval násobit $l_{ik}x_k$. Náčrtek komunikace mezi vlákny je na obrázku 5.5. Broadcasty jsou vysílány shora dolů, redukce prováděny zleva doprava.



Obrázek 5.5: Komunikace mezi vlákny v nejjemnějším řešení dolní trojúhelníkové soustavy

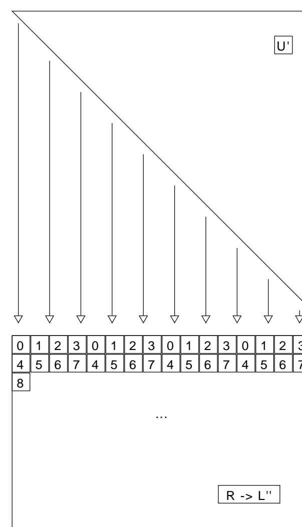
Práci lze shlukovat například po sloupcích, kdy celý sloupec řeší jedno vlákno (vlákna jsou rozdělena do jednorozměrné mřížky). Tím se zbavíme svislých broadcastů. Nebo může jedno vlákno řešit celý řádek, tím se naopak zbavíme redukcí. V obou případech lze více sloupců nebo řádků přiřadit jednomu vláknu, buď blokově, cyklicky nebo blokově cyklicky.

Pro případ bloku pivotních řádků, tedy hledání U'' řešením dolní trojúhelníkové soustavy $L'U'' = Q$, se jeví jako vhodné cyklické mapování po řádcích s tím, že navíc řešíme mnoho soustav současně (je víc vektorů pravých stran a tedy i vektorů řešení). Toto mapování je patrné z obrázku 5.6, kde je každá soustava rozdělena po řádcích cyklicky mezi čtyři vlákna a každé vlákno má na starosti čtyři řádky matice L'.



Obrázek 5.6: Ukázka cyklického mapování po řádcích s více vektory pravých stran

Mapování soustavy $U'^T L''^T = R^T$ je prakticky totožné. Soustavu je snadnější představit si tak, jak je uložena v paměti, viz obrázek 5.7. Vektory pravých stran zde tvoří řádky R.



Obrázek 5.7: Ukázka mapování na transponované soustavě

5.3.7.2 Algoritmus

Matice Q může mít mnoho sloupců, tak jako matice R může mít mnoho řádků. Vzhledem k omezené kapacitě sdílené paměti i registrů je třeba obě matice řešit po segmentech. Kdyby navíc jedno vlákno mělo na starosti více hodnot než čtyři, experimentálně bylo ověřeno, že by se začal výpočet zpomalovat. Je to proto, že je třeba udržovat ideální poměr výpočtů a paměťových přenosů, aby se jedno skrylo za druhé. Za velikost segmentu proto bylo zvoleno 128 sloupců Q nebo řádků R. Iterace po segmentech je popsána v algoritmu 5.6.

Algoritmus 5.6 Aktualizace bloků Q a R po segmentech

Vstup: Q, R, prvočíselný modul p
Výstup: U'', L''
1: **while** existuje další segment **do**
2: Přednačti příští segment z globální paměti do registru.
3: Spočti aktuální segment.
4: Bariéra.
5: Pošli výsledek aktuálního segmentu zpět do globální paměti.
6: **end while**

Výhoda tohoto postupu je v tom, že řádky 5, 2 a 3 mohou probíhat souběžně. Řádek 5 naplánuje odeslání segmentu 1 do globální paměti, řádek 2 naplánuje přijetí segmentu 3 do registru a řádek 3 hned začne počítat segment 2. Zároveň to znamená, že musíme mít dost místa v registrech pro uložení tří segmentů v jeden okamžik.

Výpočet segmentu probíhá pomocí mapování na obrázcích 5.6 a 5.7. Broadcasty spočtených hodnot jsou prováděny za pomoci pole stejného tvaru jako segment ve sdílené paměti. Tato dvě pomocná pole (jedno pro Q, jedno pro R) jsou rozšířena o jeden sloupec pro zamezení konfliktům v přístupu k paměťovým bankám. Popis aktualizace jednoho segmentu Q i R současně je v algoritmu 5.7.

Při ukládání hodnoty na řádku 2 do sdílené paměti pracují dle obrázku 5.6 čtyři vlákna jednoho halfwarpu, a protože jsou paměťové buňky po sobě jdoucí, jsou každá v jiné bance, tedy nedojde ke konfliktu.

Při ukládání hodnoty na řádku 3 pracují dle 5.7 též čtyři vlákna jednoho halfwarpu, ale tentokrát přistupují k buňkám ve sloupci. Ty však budou též každá v jiné paměťové bance, protože je toto pomocné pole ve sdílené paměti rozšířeno o jeden sloupec (podobný efekt jako na obrázku 5.4).

Následuje synchronizace, protože jiná vlákna budou vzápětí tyto broadcasty chtít přijmout.

Na řádku 6 přijímají čtyři vlákna stejného halfwarpu broadcast z jedné paměťové buňky obsahující $P_{i,k}$. Jiná čtyři vlákna téhož halfwarpu současně přijímají broadcast z jiné buňky obsahující $P_{i+1,k}$, atd. Ke konfliktu nedojde, protože přestože buňky jsou nad sebou, jsou každá v jiné paměťové bance.

Algoritmus 5.7 Aktualizace jednoho segmentu Q a R

Vstup: Q, R, prvočíselný modul p
Výstup: U'', L''
1: **for** $k \leftarrow 1, \ldots, |P|$ **do**
2: $shared(Q_{k,j}) \leftarrow Q_{k,j}$ ▷ Vlákna držící $Q_{k,1:|Q|}$
3: $shared(R_{i,k}) \leftarrow R_{i,k} \cdot pinv[k]$ ▷ Vlákna držící $R_{1:|R|,k}$
4: Bariéra.
5: **for all** $Q_{i,j}$, které má vlákno na starosti **do**
6: **if** $i > k$ **then** $Q_{i,j} \leftarrow Q_{i,j} - P_{i,k} \cdot shared(Q_{k,j}) \pmod p$
7: **end if**
8: **end for**
9: **for all** $R_{i,j}$, které má vlákno na starosti **do**
10: **if** $j > k$ **then** $R_{i,j} \leftarrow R_{i,j} - P_{k,j} \cdot shared(R_{i,k}) \pmod p$
11: **end if**
12: **end for**
13: **end for**

Více různých broadcastů v rámci jednoho halfwarpu též není problém (od CUDA compute capability 2.0). Analogické tvrzení lze vyslovit o řádku 10.

5.3.8 Aktualizace kontribučního bloku C

Jedná se o odečtení součinu v předchozím kroku spočtených matic L'' a U'' od matice C. To znamená, že od každé hodnoty c_{ij} musíme odečíst skalární součin vektorů $l''_{i*}u''_{*j}$. Vzhledem k neznámé velikosti C musíme opět postupovat po segmentech, tentokrát ale ve směru řádků i sloupců. Protože chceme co nejvíce přepoužívat načtené segmenty L'' a U'', postupujeme přednostně tou dimenzí, která je větší. Čili pokud je více řádků než sloupců, postupujeme nejprve dolů, pak doprava, a naopak.

Vnější popis práce se segmenty C je v algoritmu 5.8.

Na řádcích 12 a 15 je naplánováno stahování dat z globální paměti do registru, které probíhá souběžně s výpočty na řádku 18. Tím se snažíme docílit co největšího ukrytí přenosů z globální paměti za výpočetní čas.

Řádek 18 pracuje se segmenty L'' a U'' ve sdílené paměti a současně řádky 4 a 7 nahrávají do sdílené paměti příští potřebné segmenty (mezi těmito operacemi není bariéra), čili je třeba používat alternující bloky sdílené paměti.

Velikost segmentu C byla na základě velikosti sdílené paměti, počtu registrů a především praktických experimentů nastavena na 64×32, kde každé vlákno má na starosti dvě hodnoty v jednom sloupci. Velikost segmentu L'' je tedy 64×16 a velikost segmentu U'' je 16×32. Na přesunech segmentů L'' a U'' z globální paměti do registrů a následně z registrů do sdílené paměti spolupracují všechna vlákna a každé musí mít na starost přesun jedné hodnoty segmentu L'' a jedné hodnoty segmentu U''. Mapování výpočetních vláken na

Algoritmus 5.8 Aktualizace C po segmentech

Vstup: C, L'', U'', prvočíselný modul p
Výstup: C
1: Přednačti první segment L'' a U'' z globální paměti do registru.
2: **while** existuje další segment **do**
3: **if** ve sdílené paměti je uložen nesprávný segment L'' **then**
4: Ulož aktuální segment L'' z registru do sdílené paměti.
5: **end if**
6: **if** ve sdílené paměti je uložen nesprávný segment U'' **then**
7: Ulož aktuální segment U'' z registru do sdílené paměti.
8: **end if**
9: Bariéra.
10: **if** existuje další segment **then**
11: **if** ve sdílené paměti je jiný segment L'' než bude příště třeba **then**
12: Přednačti příští potřebný segment L'' z globální paměti do registru.
13: **end if**
14: **if** ve sdílené paměti je jiný segment U'' než bude příště třeba **then**
15: Přednačti příští potřebný segment U'' z globální paměti do registru.
16: **end if**
17: **end if**
18: Aktualizuj aktuální segment C přímo v globální paměti na základě segmentu L'' a U'' v paměti sdílené.
19: **end while**

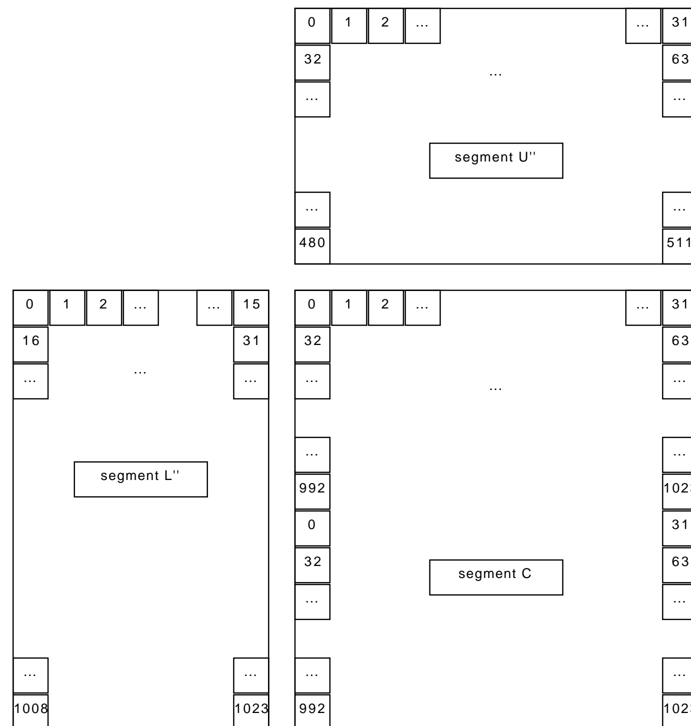
segment C a vláken zajišťujících paměťové operace na segmenty L'' a U'' je naznačeno na obrázku 5.8.

Každé výpočetní vlákno má na starost dvě hodnoty ve sloupci segmentu C. Pro každou z nich musí provést skalární součin příslušného řádku L'' a sloupce U'', které jsou nahrané ve sdílené paměti, a poté tento součin odečíst od odpovídající hodnoty C přímo v globální paměti. Hodnoty matice C po celou dobu aktualizace zůstávají v globální paměti. Tento postup je inspirován aktualizačním kernelem multifrontální QR faktorizace na GPU [2].

V každém kroku skalárního součinu čtou vlákna jednoho halfwarpu každé jinou hodnotu U'', každé z jiné paměťové banky. Hodnotu L'' čte celý halfwarp stejnou, jedná se tedy o broadcast.

5.4 Úprava UMFPACK pro LU faktorizaci

GPU kernel popsaný v sekci 5.3 je třeba zasadit do kontextu multifrontální faktorizace, abychom ji mohli vyzkoušet na problémech vlastností popsaných v kapitole 3. k dispozici máme UMFPACK popsaný v sekci 5.1.2. Jenže UMFPACK je až příliš bohatý nástroj. Čistý počet řádků C kódu knihovny je asi 35000. V symbolické analýze i numerických výpočtech je zapojených několik dodatečných specifických mechanismů urychlujících výpočty. Navíc jsou numerické kroky prováděny již v průběhu stavby frontální matice, my je potřebujeme provést všechny až v kernelu. Upravit tak velkou knihovnu pro modulární aritmetiku, pro práci s více prvočíselnými moduly současně a tak,



Obrázek 5.8: Mapování vláken na segment C

aby uměla pracovat s více řádky než sloupci, je nepoměrně složitý úkol vůči našemu cíli, kterým je srovnání GPU kernelu se sekvenční CPU verzí v daném kontextu. Nepotřebujeme absolutní čas běhu celého algoritmu, jen relativní.

Z těchto důvodů se ukázalo, že je výhodnější napsat od začátku zjednodušený UMFPACK založený na popisu algoritmu ze sekce 5.1.2.1, pracující v modulární aritmetice a schopný přijmout obdélníkovou matici soustavy. Tento zjednodušený UMFPACK má pro srovnání asi 2500 řádků C a CUDA kódu.

5.4.1 Vstupy

Vstupem LU faktorizace v upraveném UMFPACK je především matice A určená k faktorizaci. Ta je očekávána v souřadnicovém formátu, protože jak bylo řečeno v sekci 2.2.2, je tento formát vzhledem k jednoduchosti jeho konstrukce vhodný pro vstupy algoritmů. O matici A navíc potřebujeme vědět, jaké má rozměry v řádcích a sloupcích a kolik je v ní nenulových hodnot. Matice smí mít více řádků než sloupců.

Dalším vstupem je seznam prvočíselných modulů, v kterých se má souběžně pracovat. Protože je více modulů, musí i hodnoty A v souřadnicovém

formátu být specifikovány jednou pro každý modul.

5.4.2 Výstupy

Výstupem faktorizace jsou faktory L a U. Faktor L je na výstupu ve formátu komprimovaných sloupců (viz sekci 2.2.3), protože je sestavován po sloupcích, faktor U je ve formátu komprimovaných řádků (2.2.4), protože je sestavován po řádcích. Tyto formáty očekává i následný řešič popsaný v sekci 5.5.1. Každá hodnota obou faktorů je tolikanásobná, pro kolik byl výpočet prováděn modulů.

Dalšími výstupy jsou řádková a sloupcová permutace, které byly na vstupní matici A aplikovány. Permutace jsou dodány obousměrně, tj. jak pro převod starých indexů na nové, tak nových na staré. I tyto permutace jsou vstupem následného řešiče.

5.4.3 Přehled algoritmu

Algoritmus vychází ze zjednodušeného popisu UMFPACK v sekci 5.1.2.1. Zde se budeme soustředit na potřebné změny oproti němu. Algoritmus 5.9 zobrazuje původní algoritmus 5.2 s přidanými komentáři o změnách.

V první řadě potřebujeme na vstup přidat seznam modulů, jak bylo zmíněno v sekci 5.4.1. Všechny operace sestavení, které jsou skládáním různých řádků a sloupců tak, aby tam, kde se překrývají, došlo k součtu hodnot, jsou třeba provádět v jednotlivých modulech. Všechny numerické úpravy frontální matice, které byly prováděny průběžně, je nyní třeba odložit a provést uvnitř GPU kernelu až po sestavení celé frontální matice.

Faktorizace celé frontální matice GPU kernelem může selhat v $i$tém kroku kvůli numericky nulovému pivotu, to je nevýhoda zcela oddělené výpočetní fáze. V takovém případě je třeba označit pivoty $\geq i$ za nepivotní a spustit kernel znovu. Tyto pivotní řádky a sloupce budou pak faktorizovány v rámci některé pozdější frontální matice.

Není-li nalezen příští lokální pivotní řádek, velí podmínka na řádku 35 ukončit cyklus. Podmínka je takto oddělena od podmínky neexistujícího lokálního pivotního sloupce na řádku 23, protože klasický UMFPACK potřeboval vybírat pivotní prvek i na základě jeho numerické hodnoty, potřeboval, aby tato hodnota byla co největší, kvůli chybě operací s plovoucí řádovou čárkou. Naše modulární verze tímto problémem netrpí, tak může rozhodnout o existenci lokálního pivotu celkově už na řádku 23.

Pro naše účely upravený algoritmus je k vidění v algoritmu 5.10.

5.4.4 Datové struktury

Jak bylo zmíněno v sekci 5.4.1, vstupní matice A je v souřadnicovém formátu, což je nevhodný formát pro práci, proto je potřeba ji konvertovat. V jednotlivých krocích algoritmu se z této matice postupně odstraňují řádky a sloupce,

Algoritmus 5.9 Úpravy algoritmu UMFPACK [16]

Vstup: A ▷ Potřebujeme navíc seznam modulů
Výstup: L, U
1: inicializace
2: $k \leftarrow 1$
3: $\overline{V} \leftarrow \{\}$
4: **while** $k \leq n$ **do**
5: $e \leftarrow k$
6: Najdi $k$tý globální pivot a'_{rc} v aktivní matici A'.
7: $\mathcal{L} \leftarrow Struct(A'_{*c})$
8: $\mathcal{U} \leftarrow Struct(A'_{r*})$
9: $s \leftarrow g|\mathcal{L}|$
10: $t \leftarrow g|\mathcal{U}|$
11: Alokuj frontální matici F velikosti $s \times t$
12: **while** cyklus nebyl ukončen **do**
13: Sestav $k$tý pivotní řádek a sloupec do F.
14: Projdi seznamy elementů a spočítej externí stupně řádků a sloupců.
15: Sestavuj řádky a sloupce A^k, kde to lze kompletně. ▷ Operace sestavení v různých modulech
16: Sestavuj řádky a sloupce z kontribučních bloků, kde to lze kompletně. ▷ Operace sestavení v různých modulech
17: Spočítej nové horní meze stupňů řádků a sloupců.
18: Spočti hodnoty v L' a L'' ▷ Numerické úpravy odložit pro GPU kernel
19: $k \leftarrow k + 1$
20: **if** $|\mathcal{U}'| \mod b = 0$ **then**
21: $C \leftarrow C - \hat{L}\hat{U}$ ▷ Numerické úpravy odložit pro GPU kernel
22: **end if**
23: **if** $|\mathcal{U}''| = 0$ **then** ▷ Nebo $|\mathcal{L}''| = 0$
24: Ukonči cyklus.
25: **end if**
26: Najdi kandidáta na lokální pivotní sloupec $c \in \mathcal{U}''$.
27: $C_{*c} \leftarrow C_{*c} - \hat{L}\hat{U}_{*c}$ ▷ Numerické úpravy odložit pro GPU kernel
28: **if** $\overline{d_c}(c) \neq |\mathcal{L}''|$ **then**
29: Sestav sloupec c a spočti přesně $\overline{d_c}(c)$. ▷ Operace sestavení v různých modulech
30: **end if**
31: **if** $\overline{d_c}(c) > s - |\mathcal{U}'|$ **then**
32: Ukonči cyklus.
33: **end if**
34: Najdi kandidáta na lokální pivotní řádek $r \in \mathcal{L}''$.
35: **if** kandidát nenalezen **then** ▷ Již pokryto řádkem 23
36: Ukonči cyklus.
37: **end if**
38: **if** $\overline{d_r}(r) \neq |\mathcal{U}''|$ **then**
39: Sestav řádek r a spočti přesně $\overline{d_r}(r)$. ▷ Operace sestavení v různých modulech
40: **end if**
41: **if** $\overline{d_r}(r) > t - |\mathcal{U}'|$ **then**
42: Ukonči cyklus.
43: **end if**
44: $\mathcal{L} \leftarrow \mathcal{L} \cup Struct(A'_{*c})$
45: $\mathcal{U} \leftarrow \mathcal{U} \cup Struct(A'_{r*})$
46: $C_{r*} \leftarrow C_{r*} - \hat{L}_{r*}\hat{U}$ ▷ Numerické úpravy odložit pro GPU kernel
47: **end while**
48: Ulož $L', L'', \mathcal{L}, U', U'', \mathcal{U}$ ▷ Nejprve spustit GPU kernel, který tyto bloky upraví. Selže-li v kroku i, bude třeba označit pivoty $\geq i$ za nepivotní a kernel opakovat.
49: $C \leftarrow C - \hat{L}\hat{U}$ ▷ Numerická faktorizace již byla provedena.
50: $\mathcal{U}_e = \mathcal{U}''$
51: Ulož element s indexem e obsaující $C_e, \mathcal{L}_e, \mathcal{U}_e$.
52: Dealokuj F.
53: $\overline{V} = \overline{V} \cup e$
54: Přidej e do seznamů elementů.
55: **end while**

Algoritmus 5.10 Upravený algoritmus UMFPACK [16]

Vstup: A, seznam prvočíselných modulů p
Výstup: L, U
inicializace
$k \leftarrow 1$
$\overline{V} \leftarrow \{\}$
while $k \leq n$ **do**
  $e \leftarrow k$
  Najdi $k$tý globální pivot a'_{rc} v aktivní matici A'.
  $\mathcal{L} \leftarrow Struct(A'_{*c})$
  $\mathcal{U} \leftarrow Struct(A'_{r*})$
  $s \leftarrow g|\mathcal{L}|$
  $t \leftarrow g|\mathcal{U}|$
  Alokuj frontální matici F velikosti $s \times t$
  while cyklus nebyl ukončen **do**
    Sestav $k$tý pivotní řádek a sloupec do F.
    Projdi seznamy elementů a spočítej externí stupně řádků a sloupců.
    Sestavuj řádky a sloupce A^k v každém modulu, kde to lze kompletně.
    Sestavuj řádky a sloupce z kontribučních bloků v každém modulu, kde to lze kompletně.
    Spočítej nové horní meze stupňů řádků a sloupců.
    $k \leftarrow k + 1$
    if $|\mathcal{U}''| = 0 \;||\; |\mathcal{L}''| = 0$ **then**
      Ukonči cyklus.
    end if
    Najdi kandidáta na lokální pivotní sloupec $c \in \mathcal{U}''$.
    if $\overline{d_c}(c) \neq |\mathcal{L}''|$ **then**
      Sestav sloupec c v každém modulu a spočti přesně $\overline{d_c}(c)$.
    end if
    if $\overline{d_c}(c) > s - |\mathcal{U}'|$ **then**
      Ukonči cyklus.
    end if
    Najdi kandidáta na lokální pivotní řádek $r \in \mathcal{L}''$.
    if $\overline{d_r}(r) \neq |\mathcal{U}''|$ **then**
      Sestav řádek r v každém modulu a spočti přesně $\overline{d_r}(r)$.
    end if
    if $\overline{d_r}(r) > t - |\mathcal{U}'|$ **then**
      Ukonči cyklus.
    end if
    $\mathcal{L} \leftarrow \mathcal{L} \cup Struct(A'_{*c})$
    $\mathcal{U} \leftarrow \mathcal{U} \cup Struct(A'_{r*})$
  end while
  Proveď numerickou faktorizaci pomocí GPU kernelu.
  if faktorizace selhala v i-tém kroku **then**
    Označ pivoty $\geq i$ za nepivotní.
    Proveď numerickou faktorizaci pomocí GPU kernelu.
  end if
  Ulož $L', L'', \mathcal{L}, U', U'', \mathcal{U}$
  $\mathcal{U}_e = \mathcal{U}''$
  Ulož element s indexem e obsahující $C_e, \mathcal{L}_e, \mathcal{U}_e$.
  Dealokuj F.
  $\overline{V} = \overline{V} \cup e$
  Přidej e do seznamů elementů.
end while

což je operace, která je těžká provádět ve formátu komprimovaných sloupců či řádků. Proto matici A konvertujeme do formátu spojového seznamu a udržujeme odkazy jak na řádky, tak na sloupce matice, abychom obojí mohli efektivně procházet (viz sekci 2.2.5). Tento formát umožňuje velmi rychlé odstraňování prvků i celých řádků i sloupců a přirozenou aktualizaci sloupcového pohledu při odstranění z pohledu řádků a naopak, tedy je pro matici A (a následné její zbytky A^k) jako stvořený a využijeme jej i přes jeho větší paměťovou náročnost. V každém prvku spojového seznamu (nenulovém prvku matice) udržujeme jednu numerickou hodnotu pro každý pracovní modul.

Výstupní matici L konstruujeme postupně po sloupcích, kde vždy včas víme, kolik bude mít další sloupec nenulových hodnot, ideálním formátem jsou tedy komprimované sloupce. Matici U konstruujeme stejným způsobem, ale po řádcích, čili zde je výhodný formát komprimovaných řádků. Tyto formáty L a U jsou navíc dobré i pro následný řešící algoritmus, který bude nad těmito faktory provádět řešení dolní a horní trojúhelníkové soustavy. Numerické hodnoty jsou i zde vícenásobné, udržujeme jednu pro každý modul.

Pro výstupní permutace používáme jednorozměrná pole. Normální permutací nazýváme pole, které nám určuje staré indexy nových indexů, přístup k poli tedy vypadá takto:

```
P[new] = old,
```

zatímco inverzní permutací nazýváme pole, které pro staré indexy vrací indexy nové, čili

```
Pinv[old] = new.
```

Pole inverzní permutace musí mít velikost počtu řádků, z nichž některé nakonec nejsou vybrány. Na indexy nepoužitých řádků nemusíme ukládat speciální hodnoty. Stačí, když při jeho použití vždy pomocí normální permutace zkontrolujeme, zda daný index značí správné mapování. Ověření lze provést rovností

```
P[Pinv[old]] = old.
```

Hustou frontální matici ukládáme klasicky do dvourozměrného pole. Neroste dynamicky, je alokována jednou, pak je jen vyplňována. Stejně tak indexy řádků a sloupců ve frontální matici ukládáme do jednorozměrných polí. Řádky ani sloupce neudržujeme seřazené, to je v klasickém poli příliš náročná operace, každé vložení indexu by stálo $O(n)$, kde n je počet indexů, protože bychom museli posunovat ostatní indexy. Na druhou stranu tím ale zavádíme pomalejší hledání indexů ($O(n)$ místo $O(\log n)$). Dále ještě alokujeme dvě jednorozměrné bitové mapy, jednu pro indexy řádků, jednu pro indexy sloupců, které říkají, zda jsou dané indexy pivotní nebo ne.

Seznam elementů značený $\overline{V}$ je pole ukazatelů na strukturu elementu, kde index v poli značí krok faktorizace, v kterém element vznikl. Na počátku jsou všechny ukazatele nulové. Vnitřek elementu obsahuje jednorozměrná pole indexů řádků a sloupců kontribučního bloku elementu, bitová pole značící, zda

tyto řádky či sloupce již byly složeny do budoucí frontální matice, a samozřejmě kontribuční blok jako takový ve formě dvourozměrného pole.

Seznamy elementů kontribuujících do každého řádku matice A^k udržujeme v jednorozměrném poli obsahujícím spojové seznamy struktur, které tyto kontribuce popisují. To nám umožňuje kontribuce snadno přidávat a odstraňovat. Druhé pole obsahuje spojové seznamy pro sloupce.

Horní meze stupňů řádků a sloupců A^k uchováváme ve dvou jednorozměrných polích podle indexů řádků a sloupců.

Pro rychlý výběr čtyř sloupců s nejmenšími horními mezemi stupně a odstranění jednoho z nich používáme kombinaci dvou binárních hald. Více o tom v sekci 5.4.5.

5.4.5 Vybrané zajímavosti implementace

Pseudokódový popis multifrontální LU faktorizace, která je součástí této práce, je v algoritmu 5.10. Implementace algoritmu v jazyce C je k této práci přiložena. Nicméně si zde ukažme alespoň několik jejích zajímavostí.

5.4.5.1 Alokace paměti

Práce s pamětí je pečlivě ošetřena, aby ani při chybových stavech nedošlo k situaci, kdy by některý paměťový blok nebyl uvolněn. Pro snadnější zachování paměťové integrity bylo použito (s drobnými úpravami) několik obalujících funkcí pro standardní knihovní alokační a dealokační funkce jazyka C, jejichž autorem je Tim Davis a které použil ve své knihovně CSparse popsané v sekci 5.1.1.

První z těchto funkcí, funkce `safe_malloc`, obaluje funkci `malloc` a zajišťuje, aby vždy byl alokován alespoň jeden byte paměti. Funkce `safe_calloc` řeší totéž, ale vnitřně používá funkci `calloc`, která paměť nuluje.

```
void *safe_malloc (size_t n, size_t size)
{
    size_t total = MAX(n * size, 1);
    return (malloc (total)) ;
}

void *safe_calloc (size_t n, size_t size)
{
    return (calloc (MAX (n,1), size)) ;
}
```

Funkce `safe_free` má velký význam pro zajištění bezchybové práce s pamětí. Uvolní paměť, kam ukazatel ukazuje, jen pokud je ukazatel nenulový. Zároveň vrací nulu, aby bylo možné voláním funkce snadno ukazatel vynulovat.

```
void *safe_free (void *p)
{
    if (p) free (p) ;
```

```
    return 0 ;
}
```

Příklad použití:

```
p = safe_free(p).
```

Zavoláme-li tento příkaz dvakrát po sobě, nedojde k chybě. Navíc, selže-li předchozí volání

```
p = safe_malloc(count, size),
```

funkce `safe_free` neskončí s chybou, protože odhalí, že je ukazatel nulový. Díky tomu lze definovat jednu funkci pro uvolnění všech používaných ukazatelů a tu volat vždy před jakýmkoli ukončením běhu, v jakékoli fázi. Uvolněny jsou jen ty bloky paměti, které již existují a ještě nebyly uvolněny.

Poslední z těchto obalujících funkcí je funkce `safe_realloc`. Známý problém klasické funkce `realloc`, která mění velikost alokované paměti, je v tom, že nepovede-li se alokace, funkce vrátí nulový ukazatel, ale původně alokovanou paměť neuvolní. To znamená, že přirozený způsob volání

```
p = realloc(p, newSize)
```

při chybě způsobí ztrátu ukazatele na stále ještě alokovanou paměť a dojde nevyhnutelně k úniku. Funkce `safe_realloc` oproti tomu vrací vždy nenulový ukazatel, původní nebo nový. Informaci o úspěchu vrací ve vstupním ukazateli `*ok`. Zároveň funkce zajišťuje i minimální alokaci jednoho bytu.

```
void *safe_realloc (void *p, size_t n, size_t size, Bool *ok)
{
    void *pnew ;
    pnew = realloc (p, MAX (n,1) * size) ;
    *ok = (pnew != NULL) ;
    return ((*ok) ? pnew : p) ;
}
```

5.4.5.2 Binární haldy

Další implementační zajímavostí, která stojí za zmínku, je hledání čtyř sloupců s nejnižší horní mezí stupně. Kdyby nás zajímal vždy jen jeden sloupec s nejnižší horní mezí, přirozenou volbou datové struktury by byla binární halda. Prvek haldy by byl index sloupce a H-vlastnost by spočívala v nižší horní mezi stupně předka než potomka. Při výběru pivotního sloupce by došlo k odstranění vrchního prvku.

Jenže nás zajímají vrchní čtyři prvky, nebo i více, pokud změníme parametry algoritmu. Proto zavedeme druhou binární haldu rozhodující o tom, které prvky hlavní haldy jsou nejvíce na vrchu. Postup nalezení k vrchních prvků haldy H je popsán algoritmem 5.11. Druhá halda je zde označena H' a její prvky jsou indexy prvků v haldě H. Začíná prázdná. Nejprve do ni vložíme index vrchního prvku H, tedy 1, ten je jistě nejvíce vrchní. Poté postupujeme

nejvýše v k krocích tak, že vždy odstraníme vrchní prvek H' odkazující na k-tý nejvrchnější prvek H a přidáme do H' oba syny tohoto prvku H (pokud existují).

Algoritmus 5.11 Nalezení k vrchních prvků haldy H

Vstup: H, k
Výstup: x ... Pole k vrchních prvků
  $H' \leftarrow empty_heap()$
  $H' \leftarrow H'.insert(1)$
  while $|x| < k$ && H' je neprázdná **do**
    $a \leftarrow H'.extract_top()$
    $x[|x|+1] \leftarrow H[a]$
    if $left_child(a) \leq |H|$ **then**
      $H' \leftarrow H'.insert(left_child(a))$
      if $right_child(a) \leq |H|$ **then**
        $H' \leftarrow H'.insert(right_child(a))$
      end if
    end if
  end while

Tento algoritmus má složitost $O(k \log k)$, v daném kontextu je toto ovšem rovno $O(1)$, jelikož k je vůči vnějšímu algoritmu zanedbatelná konstanta.

Když pak z k kandidátů na pivotní sloupec vybereme toho pravého, odstraníme jej z haldy H. Nebude vždy vrchní, proto se nejedná o standardní operaci *extract_top*(), ale o operaci obdobnou, kde na místo odstraněného prvku vložíme poslední prvek haldy a spustíme z něj algoritmus udržení H-vlastnosti směrem dolů.

H-vlastnost haldy H je založena na horních mezích stupňů sloupců. Ty se v průběhu algoritmu mění. Změní-li se kdykoli horní mez stupně nějakého sloupce, je třeba na jemu odpovídajícím prvku haldy spustit algoritmus udržení H-vlastnosti, a to buď vzhůru, pokud horní mez klesla, nebo dolů, pokud horní mez stoupla. Najít tento odpovídající prvek chceme v konstantním čase, proto udržujeme ještě inverzi haldy H, která nám pro index sloupce dodá index v H. Údržba inverze přidává jen malou konstantní složitost operacím nad haldou.

5.4.5.3 Iterátor

V průběhu algoritmu v různých situacích potřebujeme procházet seznam elementů kontribuujících do určitého řádku nebo sloupce. Všechny tyto průchody mají společný postup, kde vždy kontrolujeme, zda daný element ještě existuje, a pokud ne, odstraníme jej ze seznamu. Co se liší je akce, která je s elementy prováděna.

Pro omezení duplicitního kódu se ukázalo jako nejvhodnější řešení definovat jednu funkci, která seznam prochází, nazvěme ji iterátorem, a předávat ji adresy jiných funkcí, které má iterátor pro každý nalezený element zavolat. Potřebujeme také zavolané funkci dopravit dodatečné informace, ty předáváme pomocí ukazatele typu `void`.

5.5 Implementace řešení soustavy

Výstupem LU faktorizace popsané v sekci 5.4 jsou faktory L a U, které je následně možné použít pro vyřešení soustavy $Ax = b$. Tato část implementace je inspirovaná více knihovnou CSparse popsanou v sekci 5.1.1 než knihovnou UMFPACK. Je velice jednoduchá a nepoužívá GPU.

5.5.1 Vstupy

Vstupem algoritmu řešení je jednak výstup algoritmu LU faktorizace, který je popsán v sekci 5.4.2, a jednak vektor pravých stran tolikanásobný, kolik je prvočíselných modulů.

5.5.2 Výstupy

Výstupem algoritmu je vektor řešení x, který je opět tolikanásobný, kolik je prvočíselných modulů. Vektor x je vracen ve stejné datové struktuře, v které byl na vstupu vektor b, čili vektor b je nahrazen vektorem x.

5.5.3 Popis algoritmu

Algoritmus implementuje řešení soustavy pomocí faktorů L a U, jak je popsáno rovnicí 2.12. Jedná se o čtyři kroky. Prvním je permutace vektoru pravé strany b podle řádkové permutace P. Druhým je řešení dolní trojúhelníkové soustavy s faktorem L. Třetím je řešení horní trojúhelníkové soustavy s faktorem U. Posledním je opět permutace, tentokrát podle té sloupcové (Q). Popis v pseudokódu je v algoritmu 5.12.

Algoritmus 5.12 Řešení soustavy pomocí faktorů z LU faktorizace

Vstup: L, U, b, P, Q, seznam prvočíselných modulů p
Výstup: x

$x \leftarrow b$
$x \leftarrow P(x)$
$x \leftarrow lower_solve(L, x, p)$
$x \leftarrow upper_solve(U, x, p)$
$x \leftarrow Q(x)$

Permutace P v prvním kroku převádí staré indexy na nové (soustavy jsou řešeny v nových), tedy se jedná o inverzní permutaci. Potřebujeme zde ale i permutaci normální, protože jak bylo řečeno v sekci 5.4.4, inverzní permutace v našem případě potřebuje ověření. Permutace Q v posledním kroku převádí nové indexy zpět na staré, jedná se tedy o normální permutaci.

5.6 Testování

Přiložená knihovna a testovací programy je nejprve třeba sestavit – v hlavní složce programu spustit příkaz `make`. Příkaz je možné konfigurovat parametry:

CUDA=false Žádost o použití sekvenčního CPU kernelu místo CUDA GPU kernelu.

INFO=true Při spuštění bude zapnut strohý informační výpis.

DEBUG=true Při spuštění bude zapnut detailní výpis. Pro větší soustavy představuje tato volba velmi dlouhou dobu běhu.

TIME=true Po skončení programu budou vypsány podrobné informace o době běhu různých jeho částí.

Sestavení je v tuto chvíli možné jen na Linuxu a na macOS. Nevyžaduje žádné neobvyklé balíky kromě funkčního CUDA kompilátoru `nvcc`.

Binární knihovna se objeví ve složce `Lib`, testovací programy ve složce `Demo`. Smazat binární soubory a vrátit se do výchozího stavu je možné příkazem `make clean`.

GPU kernel lze testovat samostatně obalujícím algoritmem, který spustí částečnou faktorizaci jedné frontální matice dané velikosti a daného počtu pivotů nad zadaným počtem prvočíselných modulů. Stejný algoritmus je možné použít i pro testování sekvenční CPU verze kernelu při parametru sestavení `CUDA=false`. Popis testovacího programu je v algoritmu 5.13. Spustitelný program se nachází ve složce `Demo`, jmenuje se `factorFrontalMatrix` a očekává následující parametry:

Počet řádků Počet řádků frontální matice.

Počet sloupců Počet sloupců frontální matice.

Počet pivotů Počet pivotních řádků a sloupců z celkového počtu.

Počet úkolů Počet náhodných úkolů s náhodnými moduly, které mají být spuštěny souběžně.

Řešení soustavy multifrontální LU faktorizací lze testovat pomocí programu `luSolveDemo` ve složce `Demo`. Program je třeba volat se šesti parametry:

Algoritmus 5.13 Test GPU kernelu faktorizací jedné frontální matice

Vstup: $|P|$, $|F|$, $|p|$
Výstup: ověřovací Mathematica notebook, čas běhu kernelu

1: Vygeneruj $|p|$ náhodných úkolů daných rozměrů s náhodnými moduly.
2: Vypiš bloky úkolů do Mathematica notebooku.
3: Spusť algoritmus faktorizace těchto úkolů.
4: Vypiš faktorizované bloky úkolů do Mathematica notebooku.

Soubor obsahující matici soustavy Cesta k souboru obsahujícímu vstupní matici A v souřadnicovém formátu. Jeden řádek obsahuje jednu hodnotu matice zadanou trojicí index řádku, index sloupce, hodnota. Celá čísla v trojici jsou oddělena mezerami. Indexy musí být menší než 2^{32}, hodnoty mohou být až $2^{64} - 1$, ale budou z nich počítány zbytky po dělení zadanými moduly a ty smí být nejvýše 2^{32}.

Soubor obsahující vektor pravé strany Cesta k souboru obsahujícímu vstupní vektor pravé strany b se souřadnicemi oddělenými mezerami nebo novými řádky. Hodnoty smí být opět až $2^{64} - 1$, ale budou z nich počítány zbytky po dělení zadanými moduly a ty smí být nejvýše 2^{32}.

Počet hodnot Celé číslo udávající počet nenulových hodnot matice A, musí být menší než 2^{64}.

Počet řádků Celé číslo udávající počet řádků vstupní matice A, musí být menší než 2^{32}.

Počet sloupců Celé číslo udávající počet sloupců vstupní matice A, musí být menší než 2^{32}.

Seznam modulů Seznam modulů, v kterých má být soustava řešena. Moduly jsou odděleny mezerami. Žádný modul nesmí být větší než 2^{32}.

Popis programu je v algoritmu 5.14.

Algoritmus 5.14 Test řešení soustavy multifrontální LU faktorizací

Vstup: A, b, seznam prvočíselných modulů p
Výstup: ověřovací Mathematica notebook, čas běhu kernelu

Proveď multifrontální LU faktorizaci matice A v modulech p.
for all p **do**
    Zapiš do Mathematica notebooku matici A.
    Zapiš do Mathematica notebooku faktory L a U.
    Zapiš do Mathematica notebooku vektor pravých stran b.
end for
Zapiš do Mathematica notebooku řádkovou permutační matici P.
Zapiš do Mathematica notebooku sloupcovou permutační matici Q.
Vyřeš soustavu $Ax = b$ pomocí faktorů L a U v modulech p.
for all p **do**
    Zapiš do Mathematica notebooku vektor výsledku x.
end for

Kapitola 6

Měření

V této kapitole se budeme věnovat měření výkonnosti implementace. Vzhledem k tomu, že byl GPU kernel zasazen do zjednodušené verze multifrontální LU faktorizace, která je jistě pomalejší než knihovna UMFPACK představená v sekci 5.1.2, nemá smysl srovnávat čas běhu algoritmu s UMFPACK nebo s jinými implementacemi. Smyslem je analyzovat zrychlení pomocí GPU. Budeme tedy vzájemně srovnávat zjednodušenou verzi s GPU kernelem a bez GPU kernelu.

Měření bude provedeno na grafické kartě *Tesla K40c* a procesoru *Intel(R) Xeon(R) CPU E5-2620 v2*.

6.1 Parametry grafické karty

Grafická karta *Tesla K40c*, na které budeme měřit, má dle programu `deviceQuery` následující parametry:

CUDA Driver Version / Runtime Version 8.0 / 8.0

CUDA Capability Major/Minor version number 3.5

Total amount of global memory 11439 MBytes (11995054080 bytes)

(15) Multiprocessors, (192) CUDA Cores/MP 2880 CUDA Cores

GPU Max Clock rate 745 MHz (0.75 GHz)

Memory Clock rate 3004 Mhz

Memory Bus Width 384-bit

L2 Cache Size 1572864 bytes

Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)

Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers

Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers

Total amount of constant memory 65536 bytes

Total amount of shared memory per block 49152 bytes

Total number of registers available per block 65536

Warp size 32

Maximum number of threads per multiprocessor 2048

Maximum number of threads per block 1024

Max dimension size of a thread block (x,y,z) (1024, 1024, 64)

Max dimension size of a grid size (x,y,z) (2147483647, 65535, 65535)

Maximum memory pitch 2147483647 bytes

Texture alignment 512 bytes

Concurrent copy and kernel execution Yes with 2 copy engine(s)

Run time limit on kernels No

Integrated GPU sharing Host Memory No

Support host page-locked memory mapping Yes

Alignment requirement for Surfaces Yes

Device has ECC support Enabled

Device supports Unified Addressing (UVA) Yes

Device PCI Domain ID / Bus ID / location ID 0 / 2 / 0

6.2 Parametry procesoru

Procesor *Intel(R) Xeon(R) CPU E5-2620 v2*, na kterém budeme měřit, má dle programu `lscpu` následující parametry:

Architecture x86_64

CPU op-mode(s) 32-bit, 64-bit

Byte Order Little Endian

CPU(s) 24

On-line CPU(s) list 0-23

Thread(s) per core 2

Core(s) per socket 6

Socket(s) 2

NUMA node(s) 2

Vendor ID GenuineIntel

CPU family 6

Model 62

Model name Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz

Stepping 4

CPU MHz 2100.091

BogoMIPS 4200.18

Virtualization VT-x

L1d cache 32K

L1i cache 32K

L2 cache 256K

L3 cache 15360K

NUMA node0 CPU(s) 0-5,12-17

NUMA node1 CPU(s) 6-11,18-23

6.3 Měření kernelu samostatně

Nejprve budeme spouštět test osamostatněného GPU kernelu s různými parametry a budeme srovnávat čas běhu oproti sekvenční CPU verzi kernelu. Tento test spustí faktorizační kernel se sadou $|p|$ úkolů velikosti $m \times n$ s k pivoty. Sledovat budeme čas běhu samotného kernelu, celkový čas faktorizace včetně kopírování dat na grafickou kartu a zpět a odpovídající čas sekvenční verze. Podíl času sekvenčního ku času na grafické kartě včetně kopírování nazveme zrychlením a bude nás zajímat jeho vývoj při určitých změnách parametrů.

Výsledky srovnávající výkon podle různých velikostí frontální matice jsou v tabulce 6.1. Jak je vidět též z grafu 6.1, zrychlení je tím větší, čím je větší matice. Je to protože je lépe využito masivního paralelismu GPU. Je možné, že pro větší problémy by zrychlení přestalo růst, nicméně tak velké frontální matice jsou vzácné.

Různou dobu běhu algoritmu faktorizace při změně jen tvaru matice je možné pozorovat v tabulce 6.2. Sekvenční algoritmus tvar matice příliš neovlivňuje, počet operací je při stejném počtu hodnot stejný. GPU kernel je naopak při větším poměru stran znatelně zpomalován, protože aktualizace pivotních řádků a sloupců není rovnoměrně rozdělena mezi vlákna. Je-li navíc

jedna strana matice menší než segment kontribučního bloku C, pracuje při aktualizaci tohoto bloku jen část vláken. Vývoj zrychlení je zobrazen v grafu 6.2.

Srovnání výkonu při označení více nebo méně řádků a sloupců jako pivotní je k vidění v tabulce 6.3 a grafu 6.3. Při větším počtu pivotů je zrychlení větší, protože faktorizace pivotního bloku a úprava bloků pivotních řádků a sloupců využije více GPU vláken.

V neposlední řadě je možné sledovat i vývoj v závislosti na počtu souběžně řešených úkolů (počtu pracovních prvočíselných modulů). Sekvenční verze tyto úkoly serializuje, GPU kernel je provádí jakožto bloky vláken. Výsledky jsou v tabulce 6.4 a zrychlení je zobrazeno v grafu 6.4. Jak je vidět, zrychlení se příliš nemění. Je to protože i GPU kernel bloky vláken na svých streaming multiprocesorech serializuje.

6.4 Vstupní data

Účelem měření celého řešení soustavy pomocí LU faktorizace je srovnat rychlost při zapojeném GPU kernelu s rychlostí při zapojeném sekvenčním CPU kernelu. Jako vstupní data použijeme několik náhodných matic různých velikostí, jejichž statistické vlastnosti odpovídají těm, které byly popsány v kapitole 3. Pro srovnání budeme algoritmus měřit i na maticích s rovnoměrným rozdělením nenul a většími hodnotami než jedna.

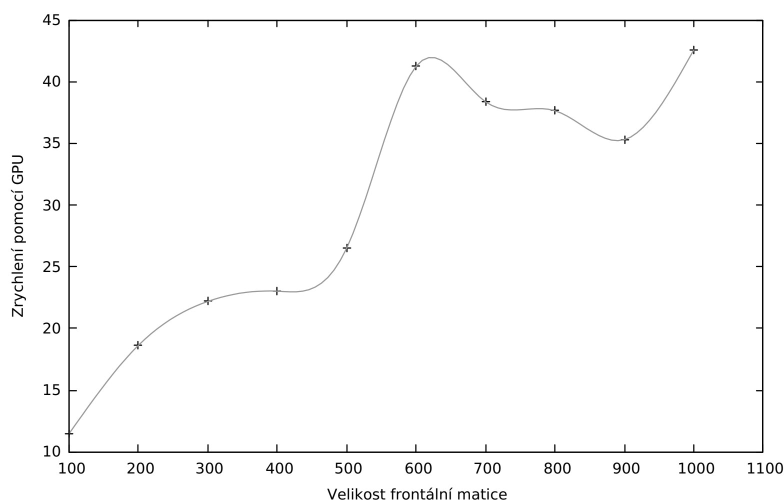
6.5 Metodika měření

V první řadě nás zajímá srovnání času celého řešení soustavy ve dvou nastaveních: Za použití GPU kernelu a bez použití GPU kernelu. Algoritmus obalující kernel ale sestává z několika fází a ačkoli nemá velký význam srovnávat absolutní čas běhu algoritmu s plnohodnotnějšími implementacemi jako je UMFPACK, můžeme alespoň změřit, které části našeho zjednodušeného algoritmu trvají déle než jiné. Proto při zapnutém sestavovacím přepínači `TIME=true` knihovna vypisuje celou strukturu informací o časech běhu jednotlivých částí algoritmu.

Měříme zvlášť trvání algoritmu LU faktorizace a zvlášť řešení pomocí faktorů L a U. Při měření LU faktorizace pozorujeme tři úrovně detailu. Nejprve nás zajímá celkový čas. Poté máme k dispozici rozpad na pět hlavních částí: Inicializace, příprava frontální matice, cyklus konstrukce frontální matice, faktorizace frontální matice (zde je použit GPU kernel nebo jeho sekvenční obdoba) a dokončení. Při ještě detailnějším pohledu můžeme měřit jednotlivé fáze konstrukce frontální matice. Jedná se o umístění frontálního řádku a sloupce, výpočet externích stupňů řádků a sloupců, skládání jiných řádků a sloupců, hledání příštího lokálního pivotu a na závěr jeho složení. Doby trvání těchto fází jsou zobrazeny v součtu přes všechny iterace.

$\|p\|$	m	n	k	GPU Čas kernelu	Čas vč. kopírování	CPU	Zrychlení
100	100	100	16	5.597	39.260	450.297	11.469
100	200	200	16	13.058	103.911	1936.639	18.637
100	300	300	16	26.562	201.024	4461.810	22.195
100	400	400	16	41.339	349.249	8039.634	23.019
100	500	500	16	70.189	476.547	12631.585	26.506
100	600	600	16	99.663	443.397	18290.026	41.249
100	700	700	16	129.540	653.232	25067.779	38.375
100	800	800	16	168.718	870.806	32781.827	37.645
100	900	900	16	206.283	1175.538	41497.005	35.300
100	1000	1000	16	255.601	1205.873	51336.821	42.572

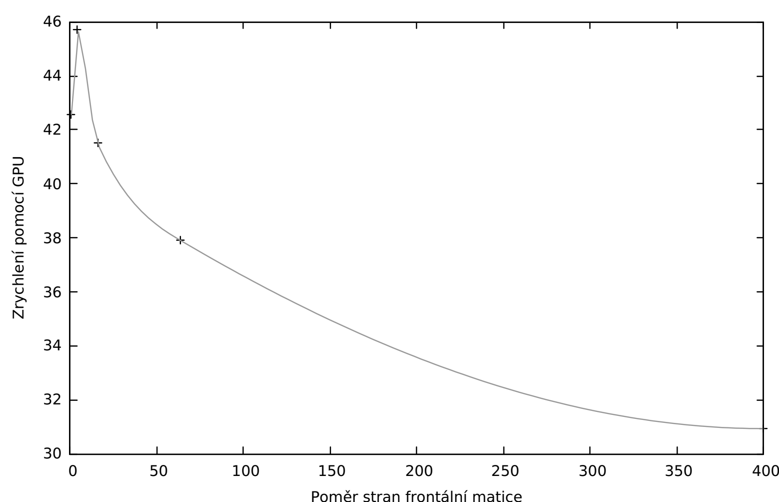
Tabulka 6.1: Výsledky měření samostatného kernelu podle velikosti (ms)



Obrázek 6.1: Zrychlení faktorizace frontální matice pomocí GPU v závislosti na její velikosti

$\|p\|$	m	n	k	GPU Čas kernelu	Čas vč. kopírování	CPU	Zrychlení
100	1000	1000	16	255.601	1205.873	51336.821	42.572
100	2000	500	16	269.771	1119.453	51163.924	45.704
100	4000	250	16	288.289	1217.387	50521.756	41.500
100	8000	125	16	319.195	1296.142	49121.223	37.898
100	20000	50	16	478.361	1450.470	44878.162	30.940

Tabulka 6.2: Výsledky měření samostatného kernelu podle tvaru (ms)



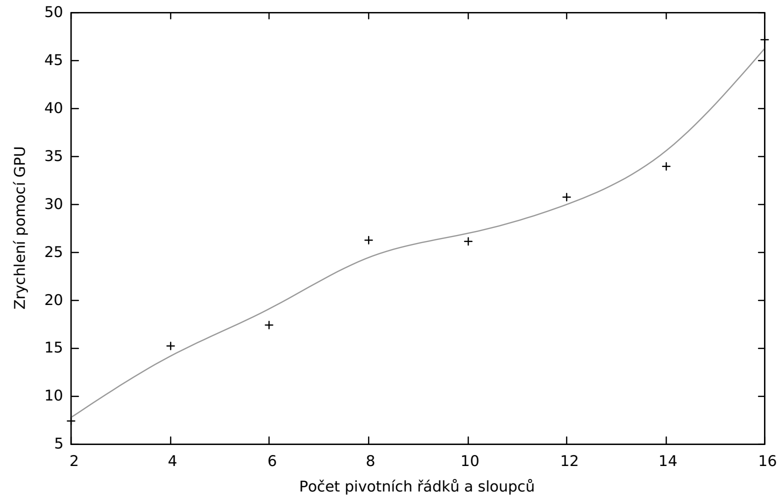
Obrázek 6.2: Zrychlení faktorizace frontální matice pomocí GPU v závislosti na jejím tvaru

$\|p\|$	m	n	k	GPU Čas kernelu	GPU Čas vč. kopírování	CPU	Zrychlení
100	1000	1000	2	51.237	1118.264	8289.123	7.412
100	1000	1000	4	82.224	960.429	14570.782	15.171
100	1000	1000	6	113.269	1192.173	20773.248	17.424
100	1000	1000	8	136.662	1021.177	26840.341	26.283
100	1000	1000	10	167.831	1263.619	32938.806	26.067
100	1000	1000	12	196.641	1295.712	39770.211	30.693
100	1000	1000	14	227.304	1332.005	45180.413	33.919
100	1000	1000	16	255.353	1087.431	51308.374	47.183

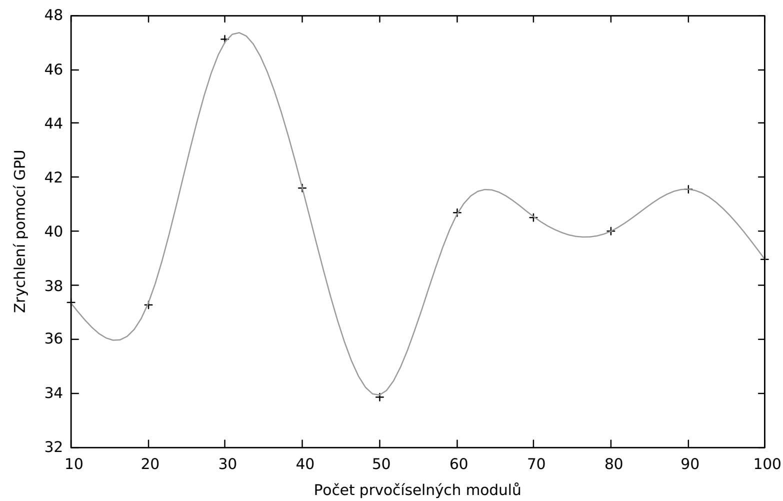
Tabulka 6.3: Výsledky měření samostatného kernelu podle počtu pivotních řádků a sloupců (ms)

$\|p\|$	m	n	k	GPU Čas kernelu	GPU Čas vč. kopírování	CPU	Zrychlení
10	1000	1000	16	36.761	137.928	5152.462	37.356
20	1000	1000	16	72.585	275.287	10255.552	37.254
30	1000	1000	16	73.242	327.051	15403.479	47.098
40	1000	1000	16	109.669	492.967	20507.219	41.599
50	1000	1000	16	144.652	757.067	25633.611	33.859
60	1000	1000	16	146.322	756.245	30761.277	40.676
70	1000	1000	16	182.508	890.451	36079.022	40.517
80	1000	1000	16	217.207	1031.091	41233.476	39.990
90	1000	1000	16	219.643	1119.713	46559.114	41.581
100	1000	1000	16	255.373	1333.405	51921.631	38.939

Tabulka 6.4: Výsledky měření samostatného kernelu podle počtu souběžně řešených úloh (ms)



Obrázek 6.3: Zrychlení faktorizace frontální matice pomocí GPU v závislosti na počtu pivotních řádků a sloupců



Obrázek 6.4: Zrychlení faktorizace frontální matice pomocí GPU v závislosti na počtu prvočíselných modulů – souběžně řešených úloh

Řešení soustavy pomocí připravených faktorů L a U měříme jednak celkově a jednak rozdělené na čtyři fáze – na permutaci podle řádkové permutace P, řešení dolní trojúhelníkové soustavy s maticí L, řešení horní trojúhelníkové soustavy s maticí U a permutaci podle sloupcové permutace Q.

6.6 Výsledky

Podívejme se nyní na výsledky všech provedených měření, nejprve nad soustavami odpovídajícími vlastnostem popsaným v kapitole 3, poté nad soustavami s rovnoměrným rozložením hodnot. Rozměrné tabulky zobrazené v této sekci neposkytují dost prostoru pro vysvětlení jednotlivých sloupců, proto nejprve uveďme jejich popis:

n Pracovní těleso $GF(2^n)$

|p| Počet prvočíselných faktorů – počet souběžně řešených úloh

h Počet nenulových hodnot

r Počet řádků

c Počet sloupců

GPU faktorizace Čas faktorizací frontálních matic pomocí GPU kernelu

GPU celkem Celkový čas řešení soustavy se zapojeným GPU kernelem

CPU celkem Celkový čas řešení soustavy se zapojenou sekvenčních CPU obdobou GPU kernelu

CPU ostatní Detailní doby trvání jednotlivých fází algoritmu (při zapojené sekvenční obdobě kernelu)

Jednotlivé fáze algoritmu jsou ve sloupcích označeny jen čísly a písmeny, zde je legenda:

1 Inicializace

2 Příprava frontální matice

3 Vložení pivotního řádku a sloupce

4 Výpočet externích stupňů řádků a sloupců

5 Sestavení jiných řádků a sloupců

6 Hledání příštího lokálního pivotu

7 Sestavení příštího lokálního pivotu

8 Faktorizace frontální matice

9 Dokončení

a Permutace podle řádkové permutace

b Řešení dolní trojúhelníkové soustavy

c Řešení horní trojúhelníkové soustavy

d Permutace podle sloupcové permutace

Tabulka 6.5 ukazuje výsledky měření nad soustavami statistických vlastností odpovídajících soustavám vznikajícím v index calculu. V prvním sloupci vidíme n určující těleso $GF(2^n)$, kde by se taková soustava mohla vyskytnout. V horní části tabulky je zanesen vývoj podle zvětšujícího se n. Ve spodní části pak můžeme pozorovat vývoj v závislosti na počtu prvočíselných modulů, v kterých je soustava řešena.

Z horní části tabulky 6.5 je patrné, že pro větší tělesa čas běhu prudce roste, viz také graf 6.5. Podstatné zjištění je to, že je GPU verze pomalejší. Až při $n = 96$ se GPU verze blíží té sekvenční. Jak ukazuje graf 6.6, zrychlení roste a pravděpodobně by pro větší problémy dále rostlo, nicméně jsme daleko pod zrychlením, které jsme pozorovali při měření samostatného kernelu. Problém je v tom, že, jak je možné pozorovat při zapnutých informativních výstupech programu, frontální matice jsou poměrně malé. Matice soustav jsou tak řídké, že po většinu algoritmu pracujeme s frontálními maticemi s jedním pivotním řádkem a sloupcem a jednotkami řádků a sloupců nepivotních. To je pro použití GPU kernelu velice nevýhodné. Pustíme-li se do kopírování dat na grafickou kartu a chceme-li využít jejího výkonu, potřebujeme mít dostatek dat, aby pracovala pokud možno všechna vlákna co nejefektivněji. Jinak je lepší použít CPU.

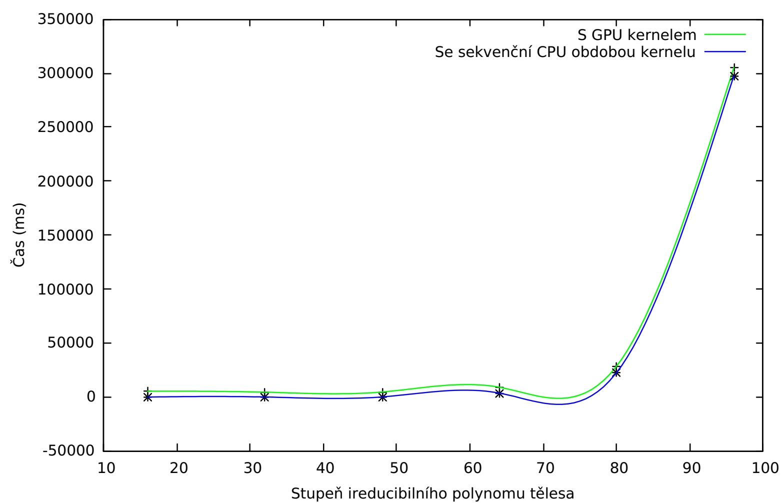
Dolní část tabulky zobrazuje vývoj podle počtu prvočíselných modulů, v kterých souběžně pracujeme. Jak je vidět z grafu 6.7, časy obou způsobů řešení se vyvíjejí více méně lineárně, a graf 6.8 potvrzuje, že zrychlení (v tomto případě zpomalení) příliš neroste ani neklesá. Je to proto, že jak CPU tak GPU musí úkoly do velké míry serializovat.

Dále je v tabulce 6.5 možné dočíst se detailních údajů o průběhu algoritmu. Je patrné, že nejvíce času (pomineme-li numerickou faktorizaci na GPU) algoritmus tráví ve fázi sestavování lokálních řádků a sloupců (fáze 7). To je dáno především tím, že pro každý řádek či sloupec, který sestavujeme, musíme projít všechny řádky či sloupce frontální matice, abychom zjistili, zda už je přítomen. Bylo by možné zkusit udržovat řádky a sloupce ve frontální matici seřazené, aby toto hledání bylo logaritmické (na druhou stranu by bylo pomalejší vkládání nových).

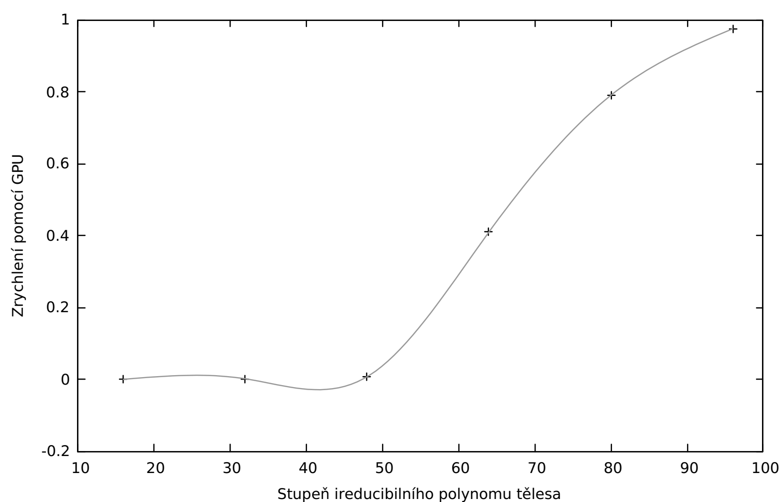
Tabulka 6.6 obsahuje výsledky měření nad soustavami se stejným počtem sloupců a podobným počtem hodnot na řádek, ale s hodnotami rozmístěnými

					GPU		CPU																
							LU faktorizace										Řešení						
									Stavba F														
n	$\|p\|$	h	r	c	Faktorizace	Celkem	1	2	3	4	5	6	7	8	9	Celkem	a	b	c	d	Celkem	Celkem	Zrychlení
16	10	70	36	8	5366.597	5366.946	0.043	0.139	0.016	0.007	0.013	0.006	0.008	0.145	0.004	0.437	0.002	0.005	0.005	0.001	0.016	0.453	0.000
32	10	1122	427	41	4467.379	4476.802	0.427	2.845	0.198	0.108	0.152	0.039	4.760	1.953	0.021	10.781	0.006	0.020	0.029	0.003	0.059	10.840	0.002
48	10	2242	812	71	4571.934	4601.031	0.778	6.639	0.439	0.206	0.323	0.065	18.713	4.096	0.039	31.789	0.009	0.033	0.060	0.005	0.112	31.901	0.007
64	10	18239	6209	226	5227.153	9189.905	6.578	180.740	3.928	1.905	2.905	0.264	3529.572	36.450	0.254	3764.192	0.043	0.138	0.170	0.014	0.369	3764.561	0.410
80	10	34454	11378	412	5961.204	28268.012	12.185	571.064	8.411	5.103	6.621	0.497	21659.013	72.017	0.635	22338.561	0.089	0.254	0.311	0.023	0.680	22339.241	0.790
96	10	94858	30709	747	7459.498	305323.823	34.662	4301.262	38.517	17.381	20.867	1.050	292989.193	208.041	1.662	297618.439	0.267	0.650	0.561	0.063	1.546	297619.985	0.975
64	10	18239	6209	226	5227.153	9189.905	6.349	180.742	3.948	1.945	2.920	0.275	3529.070	36.368	0.255	3763.644	0.059	0.139	0.170	0.014	0.386	3764.030	0.410
64	20	18239	6209	226	6862.581	10862.443	10.477	215.758	6.800	1.940	3.543	0.274	3730.326	67.895	0.296	4038.947	0.072	0.201	0.337	0.025	0.639	4039.586	0.372
64	30	18239	6209	226	6233.110	10359.683	13.890	252.857	12.306	2.287	4.471	0.288	4056.009	104.108	0.333	4448.366	0.189	0.315	0.513	0.041	1.063	4449.429	0.429
64	40	18239	6209	226	6669.876	11050.477	19.933	290.829	18.576	2.327	5.250	0.281	3998.275	139.335	0.377	4477.004	0.239	0.594	0.810	0.063	1.710	4478.714	0.405
64	50	18239	6209	226	7126.791	11787.510	24.615	327.190	26.113	2.385	6.093	0.266	4063.647	172.462	0.440	4624.912	0.267	0.609	0.944	0.079	1.904	4626.816	0.393
64	60	18239	6209	226	8563.732	13379.429	30.777	372.015	34.935	2.692	7.164	0.287	4286.383	208.894	0.481	4945.414	0.355	1.073	1.318	0.105	2.854	4948.268	0.370
64	70	18239	6209	226	8047.444	13069.722	36.055	411.547	44.881	2.686	8.250	0.294	4606.754	244.272	0.539	5357.125	0.448	1.377	1.609	0.130	3.568	5360.693	0.410
64	80	18239	6209	226	8441.194	13718.988	41.595	452.177	55.147	2.749	9.225	0.272	4635.608	287.252	0.593	5486.303	0.454	1.082	1.668	0.145	3.353	5489.656	0.400
64	90	18239	6209	226	10019.358	15513.543	46.627	494.207	65.759	2.969	10.569	0.296	4778.788	330.709	0.637	5732.446	0.601	1.865	2.200	0.176	4.847	5737.293	0.370
64	100	18239	6209	226	10442.996	16176.431	51.935	537.154	78.034	2.955	11.525	0.280	5038.540	374.282	0.687	6097.093	0.590	1.384	2.121	0.220	4.320	6101.413	0.377

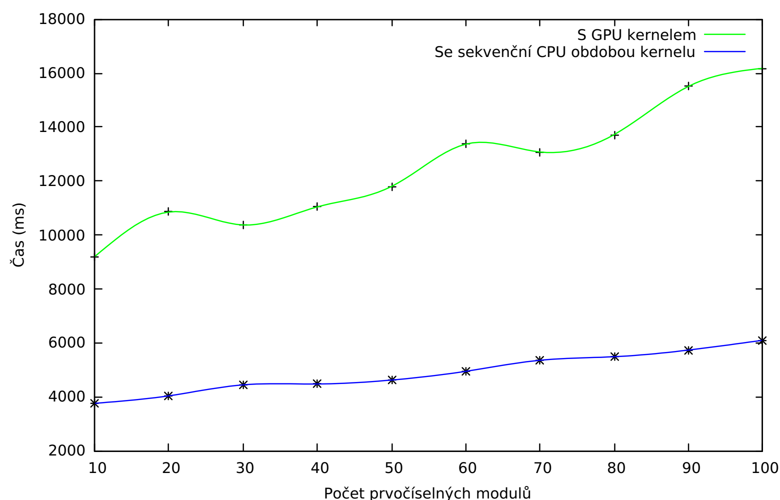
Tabulka 6.5: Měření nad soustavami patřičných vlastností (ms)



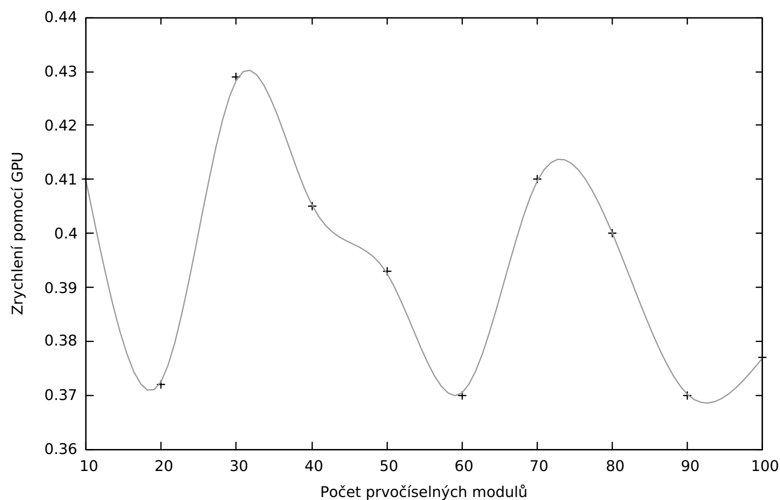
Obrázek 6.5: Čas běhu algoritmu řešení soustavy v závislosti na velikosti tělesa



Obrázek 6.6: Zrychlení algoritmu řešení soustavy pomocí GPU v závislosti na velikosti tělesa



Obrázek 6.7: Čas běhu algoritmu řešení soustavy v závislosti na počtu prvočíselných modulů – souběžně řešených úloh



Obrázek 6.8: Zrychlení algoritmu řešení soustavy pomocí GPU v závislosti na počtu prvočíselných modulů – souběžně řešených úloh

rovnoměrně. Vývoj hodnot je velmi podobný. Časy jsou ale kratší, rychlost tedy větší.

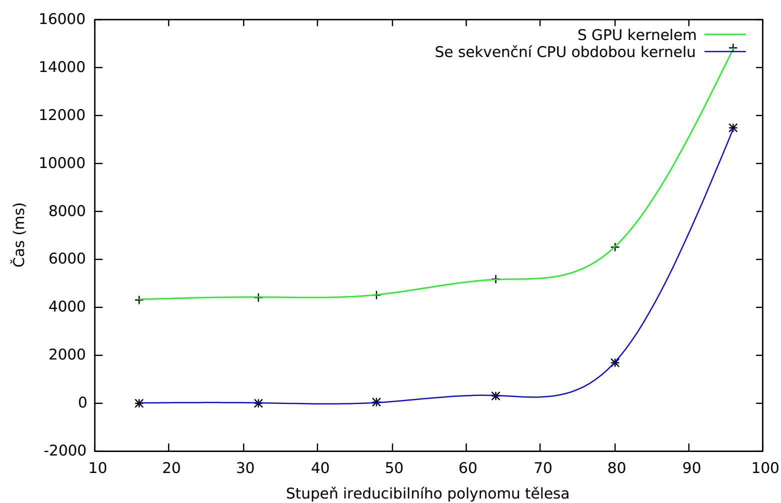
Jedním z důvodů rychlejšího řešení soustav s rovnoměrným rozmístěním jsou menší frontální matice v pozdějších fázích algoritmu. Husté sloupce u jedné strany matic soustav vyskytujících se v index calculu totiž způsobují velkou výplň nenul, i když jsou vybrány co nejpozději.

Druhým urychlovačem jsou větší hodnoty v maticích. Zatímco v maticích příslušících index calculu jsou většinou jedničky, v těchto maticích jsou libovolná 32 bitová čísla. Jsou-li v matici samé jedničky (nebo jakékoli jiné hodnoty všechny stejné), dochází často k jejich vzájemnému odečtení během faktorizace frontální matice a k nutnosti faktorizaci opakovat s menším počtem pivotů.

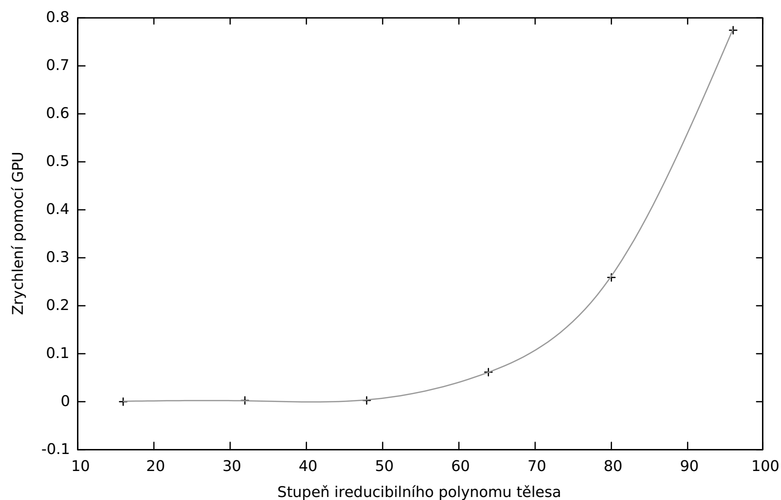
Grafy 6.9 až 6.12, založené na tabulce 6.6, odpovídají formou grafům 6.5 až 6.8 a mají podobný vývoj s výjimkou grafu 6.12, který na rozdíl od grafu 6.8 jednoznačně roste. Zdá se, že u soustavy s náhodným rozmístěním dokáže GPU lépe využít výkonu napříč bloky vláken. Důvodem jsou pravděpodobně menší frontální matice.

					GPU		CPU																		
							LU faktorizace												Řešení						
									Stavba F																
n	$\|p\|$	h	r	c	8	Celkem	1	2	3	4	5	6	7	8	9	Celkem	a	b	c	d	Celkem	Celkem	Zrychlení
16	10	70	36	8	4316.332	4316.622	0.039	0.083	0.015	0.008	0.024	0.008	0.027	0.177	0.006	0.441	0.002	0.010	0.053	0.001	0.069	0.510	0.000
32	10	1122	427	41	4410.640	4413.051	0.147	0.719	0.113	0.082	0.380	0.044	0.350	2.660	0.029	4.780	0.005	0.065	0.311	0.004	0.387	5.167	0.001
48	10	2242	812	71	4504.494	4509.663	0.264	1.458	0.218	0.155	1.188	0.083	0.679	8.787	0.059	13.295	0.006	0.139	0.556	0.004	0.709	14.004	0.003
64	10	18239	6209	226	5049.299	5147.650	0.888	21.281	1.996	1.545	49.575	0.532	5.985	229.008	0.487	312.984	0.019	1.036	2.456	0.013	3.529	316.513	0.061
80	10	34454	11378	412	5966.358	6494.190	1.557	114.568	6.511	4.678	353.809	1.475	15.483	1168.779	1.383	1671.794	0.037	2.907	5.226	0.024	8.199	1679.993	0.259
96	10	94858	30709	747	9751.079	14785.099	2.756	886.580	40.051	17.925	3572.061	6.288	57.191	6827.343	3.999	11425.261	0.093	8.081	12.043	0.047	20.267	11445.528	0.774
64	10	18239	6209	226	6097.894	6197.369	0.911	21.347	1.981	1.541	49.647	0.533	6.009	230.140	0.493	314.292	0.033	1.035	2.452	0.013	3.537	317.829	0.051
64	20	18239	6209	226	5425.952	5587.606	1.416	38.819	4.379	1.609	84.170	0.494	9.451	462.420	0.817	605.179	0.038	1.982	4.877	0.027	6.928	612.107	0.110
64	30	18239	6209	226	5806.091	6064.582	1.814	58.336	6.974	1.742	128.614	0.534	13.865	691.604	1.452	906.636	0.098	3.214	7.339	0.044	10.699	917.335	0.151
64	40	18239	6209	226	6139.680	6483.705	2.477	81.114	9.927	1.827	175.372	0.500	18.446	922.999	2.681	1216.989	0.121	4.165	9.920	0.067	14.276	1231.265	0.190
64	50	18239	6209	226	7578.228	8018.980	3.025	104.508	13.340	1.953	226.052	0.532	23.594	1157.906	3.421	1536.046	0.255	6.350	13.346	0.090	20.045	1556.091	0.194
64	60	18239	6209	226	6883.061	7440.556	3.530	128.133	16.917	2.039	273.924	0.507	28.480	1396.909	3.956	1856.049	0.266	6.847	15.433	0.110	22.660	1878.709	0.252
64	70	18239	6209	226	7249.256	7900.237	4.009	151.760	20.816	2.122	327.591	0.517	33.892	1627.467	4.970	2174.754	0.320	8.178	18.282	0.134	26.918	2201.672	0.279
64	80	18239	6209	226	7631.483	8343.062	4.499	208.993	29.454	2.531	451.537	0.516	44.114	1896.457	7.026	2646.907	0.510	11.663	21.151	0.159	33.487	2680.394	0.321
64	90	18239	6209	226	8022.113	8904.034	5.003	200.139	29.090	2.337	436.302	0.521	45.021	2105.092	6.424	2831.532	0.454	11.292	23.818	0.198	35.767	2867.299	0.322
64	100	18239	6209	226	8377.467	9374.602	5.662	225.868	33.844	2.474	496.022	0.538	51.027	2335.215	7.222	3159.684	0.659	16.608	29.319	0.236	46.825	3206.509	0.342

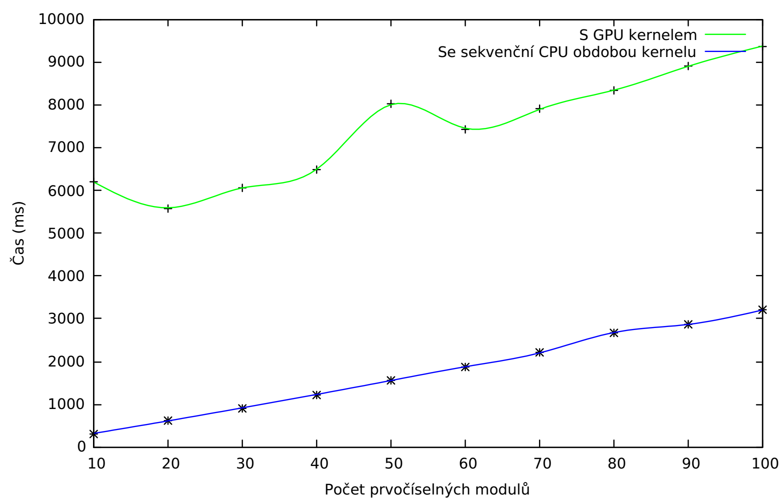
Tabulka 6.6: Měření nad soustavami s rovnoměrně rozloženými nenulovými prvky (ms)



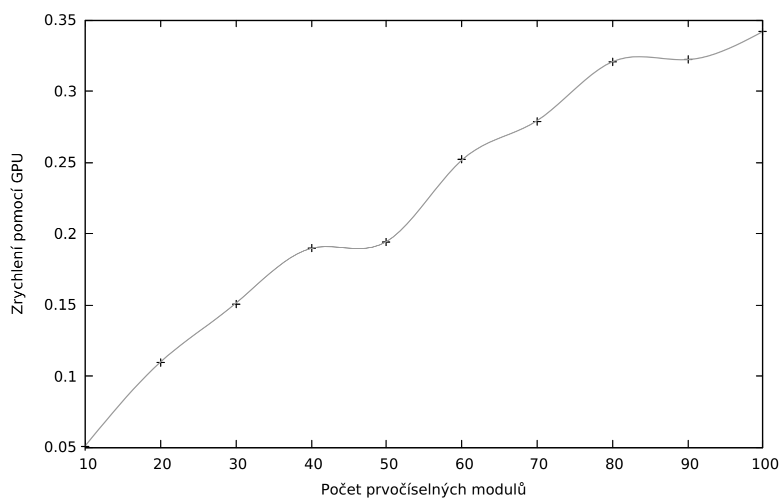
Obrázek 6.9: Čas běhu algoritmu řešení soustavy v závislosti na velikosti tělesa při rovnoměrném rozložení nenul



Obrázek 6.10: Zrychlení algoritmu řešení soustavy pomocí GPU v závislosti na velikosti tělesa při rovnoměrném rozložení nenul



Obrázek 6.11: Čas běhu algoritmu řešení soustavy v závislosti na počtu prvočíselných modulů – souběžně řešených úloh při rovnoměrném rozložení nenul



Obrázek 6.12: Zrychlení algoritmu řešení soustavy pomocí GPU v závislosti na počtu prvočíselných modulů – souběžně řešených úloh při rovnoměrném rozložení nenul

Závěr

Cílem práce bylo popsat problém diskrétního logaritmu a algoritmus zvaný index calculus jakožto nejefektivnější známý způsob jeho řešení. Dalším cílem bylo prozkoumat možnost zrychlení jedné z fází tohoto algoritmu pomocí grafické karty počítače, konkrétně kroku, v kterém je třeba řešit rozsáhlou řídkou soustavu lineárních kongruencí.

Existuje několik známých postupů, jak řešit soustavu lineárních rovnic. Většina z nich sestává ze dvou hlavních fází: Fáze faktorizační, která hledá rozklad matice soustavy na součin dvou nebo více matic, a fáze řešící, která tento rozklad využije k nálezu řešení. První z těchto fází je většinou výrazně náročnější. V této práci se podařilo nalézt a popsat tři takové postupy založeny na třech různých dekompozicích matice soustavy: Choleského faktorizaci, LU faktorizaci a QR faktorizaci.

Práce se dále zabývala specifickými úpravami, které je třeba do těchto známých algoritmů zavést, aby bylo možné jejich prostřednictvím řešit nejen lineární rovnice, ale i lineární kongruence. Jak se ukázalo, QR faktorizaci nelze použít, protože pro polovinu hodnot, které se v matici soustavy mohou objevit, není v modulární aritmetice definována druhá odmocnina.

Prozkoumány byly i statistické vlastnosti soustav, které jsou v tomto kroku index calculu třeba řešit. Na jejich základě byla vyřazena možnost použití Choleského faktorizace, protože je soustava příliš málo symetrická. Díky znalosti statistických vlastností se podařilo vyvinout algoritmus, který generuje ukázkové náhodné soustavy rychleji, než je generuje index calculus. To je výhodné pro testování.

Z popsaných postupů zbylo jen řešení pomocí LU faktorizace, tj. rozkladu matice soustavy na součin dolní a horní trojúhelníkové matice (a případných permutačních matic). Práce se dále zabývala existujícími implementacemi LU faktorizace a možnostmi jejich převodu do modulární aritmetiky a na grafickou kartu. Podařilo se nalézt tři existující implementace: CSparse (jednoduchou knihovnu pracující s řídkými maticemi), UMFPACK (podstatně složitější knihovnu pro tzv. multifrontální metodu asymetrické LU faktorizace použitou

v MATLABu) a vědeckou práci zabývající se možností převodu numerického jádra knihovny UMFPACK na grafickou kartu.

Na základě výzkumu byl zvolen postup pokusit se implementovat novou zjednodušenou podobu algoritmu použitého v knihovně UMFPACK. Implementace se podařila a je plně funkční včetně několika potřebných úprav, především pro modulární aritmetiku, pro práci současně v několika prvočíselných modulech (to je nutné v případě, že řád grupy je složený) a pro obdélníkovou matici soustavy (protože se v index calculu může stát, že je třeba vygenerovat více kongruencí než je počet neznámých).

Pro grafickou kartu byl úspěšně implementován GPU kernel částečně faktorizující hustou frontální podmatici vstupní matice soustavy, což je dílčí krok multifrontální LU faktorizace, do nějž se soustředí velká většina numerických operací. Spolu s GPU kernelem byla pro srovnání vyvinuta i jeho sekvenční podoba určena pro běžný procesor. Při sestavení přiložené zjednodušené implementace multifrontální LU faktorizace je možné zvolit, která verze výpočetního kernelu bude zapojena.

Jak pro samotný GPU kernel, tak pro celý algoritmus multifrontální LU faktorizace byly vyvinuty testovací programy. Jejich prostřednictvím bylo otestováno řešení ukázkových soustav kongruencí, jednak takových, jejichž statistické vlastnosti se blíží vlastnostem soustav, které skutečně vznikají v postupu index calculu, a jednak takových, kde je rozložení nenulových prvků více rovnoměrné.

Při testování bylo odhaleno, že ačkoli je implementovaný GPU kernel v běžném případě podstatně výkonnější než jeho sekvenční podoba (zrychlení je v řádu desítek), při zapojení do konkrétního problému multifrontální LU faktorizace matic soustav patřičné řídkosti představuje GPU kernel oproti sekvenční podobě spíše drobné zpomalení. Hlavní příčinou je příliš malý počet kroků faktorizace uvnitř každé frontální matice, což je dáno extrémní řídkostí matic. GPU kernel představuje zrychlení tehdy, když se v něm odehrává větší množství práce, jinak je kopírování dat do paměti grafické karty a zpět neadekvátně pomalé.

Cílem práce bylo zjistit, zda je vhodné jeden vybraný krok index calculu urychlit pomocí grafické karty. Jak se ukázalo, přinejmenším při tomto zvoleném postupu to vhodné není. Dá se říci, že se jedná o dobrou zprávu, protože problém diskrétního logaritmu zůstává i nadále těžkým problémem a bezpečnost šifer, které na něj spoléhají, není touto prací snížena.

Literatura

[1] Davis, T. A.; Rajamanickam, S.; Sid-Lakhdar, W. M.: A survey of direct methods for sparse linear systems. *Acta Numerica*, ročník 25, 2016: s. 383–566, doi:10.1017/S0962492916000076.

[2] YERALAN, S. N.; DAVIS, T. A.; RANKA, S.: Algorithm 9xx: Sparse QR Factorization on the GPU. 2015.

[3] Tvrdík, P.: Paralelní algoritmy pro lineární algebru. *KPS FIT ČVUT*, 2015.

[4] Tomáš Kalvoda, Š. S., Ivo Petr: Matematika pro kryptologii. *KAM FIT ČVUT*, květen 2016.

[5] Shanks, D.: Class number, a theory of factorization, and genera. In *1969 Number Theory Institute (Proc. Sympos. Pure Math., Vol. XX, State Univ. New York, Stony Brook, N.Y., 1969)*, Providence, R.I., 1971, s. 415–440.

[6] Pollard, J.: Monte Carlo methods for Index Computation (mod p). *Mathematics of Computation*, ročník 32, 1978: s. 918–924.

[7] Adleman, L.: A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, SFCS '79, Washington, DC, USA: IEEE Computer Society, 1979, s. 55–60, doi:10.1109/SFCS.1979.2. Dostupné z: `http://dx.doi.org/10.1109/SFCS.1979.2`

[8] Pohlig, S.; Hellman, M.: An improved algorithm for computing logarithms overGF(p)and its cryptographic significance (Corresp.). *IEEE Transactions on Information Theory*, ročník 24, č. 1, January 1978: s. 106–110, ISSN 0018-9448, doi:10.1109/TIT.1978.1055817.

[9] Yannakakis, M.: Computing the Minimum Fill-In is NP-Complete. *SIAM Journal on Algebraic Discrete Methods*, ročník 2, č. 1, 1981: s. 77–79, doi:10.1137/0602010, http://dx.doi.org/10.1137/0602010. Dostupné z: http://dx.doi.org/10.1137/0602010

[10] Parter, S.: The Use of Linear Graphs in Gauss Elimination. *SIAM Review*, ročník 3, č. 2, 1961: s. 119–130, doi:10.1137/1003021, http://dx.doi.org/10.1137/1003021. Dostupné z: http://dx.doi.org/10.1137/1003021

[11] Liu, J. W.: A Compact Row Storage Scheme for Cholesky Factors Using Elimination Trees. *ACM Trans. Math. Softw.*, ročník 12, č. 2, Červen 1986: s. 127–148, ISSN 0098-3500, doi:10.1145/6497.6499. Dostupné z: http://doi.acm.org/10.1145/6497.6499

[12] Schreiber, R.: A New Implementation of Sparse Gaussian Elimination. *ACM Trans. Math. Softw.*, ročník 8, č. 3, Září 1982: s. 256–276, ISSN 0098-3500, doi:10.1145/356004.356006. Dostupné z: http://doi.acm.org/10.1145/356004.356006

[13] Gilbert, J. R.; Peierls, T.: Sparse Partial Pivoting in Time Proportional to Arithmetic Operations. *SIAM Journal on Scientific and Statistical Computing*, ročník 9, č. 5, 1988: s. 862–874, doi:10.1137/0909058, http://dx.doi.org/10.1137/0909058. Dostupné z: http://dx.doi.org/10.1137/0909058

[14] Gilbert, J. R.; Ng, E. G.: Predicting Structure In Nonsymmetric Sparse Matrix Factorizations. In *GRAPH THEORY AND SPARSE MATRIX COMPUTATION*, Springer-Verlag, 1992, s. 107–139.

[15] Duff, I. S.; Reid, J. K.: The Multifrontal Solution of Indefinite Sparse Symmetric Linear. *ACM Trans. Math. Softw.*, ročník 9, č. 3, Září 1983: s. 302–325, ISSN 0098-3500, doi:10.1145/356044.356047. Dostupné z: http://doi.acm.org/10.1145/356044.356047

[16] Davis, T. A.; Duff, I. S.: An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization. *SIAM Journal on Matrix Analysis and Applications*, ročník 18, č. 1, 1997: s. 140–158, doi:10.1137/S0895479894246905, http://dx.doi.org/10.1137/S0895479894246905. Dostupné z: http://dx.doi.org/10.1137/S0895479894246905

[17] Gilbert, J. R.; Li, X. S.; Ng, E. G.; aj.: Computing Row and Column Counts for Sparse QR and LU Factorization. *BIT Numerical Mathematics*, ročník 41, č. 4, 2001: s. 693–710, ISSN 1572-9125, doi: 10.1023/A:1021943902025. Dostupné z: http://dx.doi.org/10.1023/A:1021943902025

[18] Wilf, H.: *Generatingfunctionology*. Academic Press, 1994, ISBN 9780127519562. Dostupné z: https://books.google.cz/books?id=5iTvAAAAMAAJ

[19] Flajolet, P.; Soria, M.: Gaussian limiting distributions for the number of components in combinatorial structures. *Journal of Combinatorial Theory, Series A*, ročník 53, č. 2, 1990: s. 165 – 182, ISSN 0097-3165, doi:http://dx.doi.org/10.1016/0097-3165(90)90056-3. Dostupné z: http://www.sciencedirect.com/science/article/pii/0097316590900563

[20] Granville, A.: The anatomy of integers and permutations. *Départment de Mathématiques et Statistique, Université de Montréal*, 2011.

[21] Panario, D.: Counting Polynomials over Finite Fields: Random Properties and Algorithms. *School of Mathematics and Statistics, Carleton University*, May 2010.

[22] Davis, T.: *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006, doi:10.1137/1.9780898718881, http://epubs.siam.org/doi/pdf/10.1137/1.9780898718881. Dostupné z: http://epubs.siam.org/doi/abs/10.1137/1.9780898718881

[23] Markowitz, H. M.: The Elimination Form of the Inverse and Its Application to Linear Programming. *Management Science*, ročník 3, č. 3, 1957: s. 255–269, ISSN 00251909, 15265501. Dostupné z: http://www.jstor.org/stable/2627454

[24] Yu, C. D.; Wang, W.; Pierce, D.: A CPU–GPU hybrid approach for the unsymmetric multifrontal method. *Parallel Computing*, ročník 37, č. 12, 2011: s. 759 – 770, ISSN 0167-8191, doi:https://doi.org/10.1016/j.parco.2011.09.002, 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10). Dostupné z: http://www.sciencedirect.com/science/article/pii/S0167819111001293

Příloha A

Seznam použitých zkratek

GPU Graphical processing unit

CPU Central processing unit

UMFPACK Unsymmetric multifrontal package

Příloha B

Obsah přiloženého CD

```
readme.txt ............... obsah CD a pokyny pro sestavení knihovny
src
  impl ..................................... zdrojové kódy implementace
    lib ............................................................ knihovna
    matrixgen ..................................... generátory matic
  thesis ................... zdrojová forma práce ve formátu LaTeX
text ........................................................... text práce
  thesis.pdf ................................ text práce ve formátu PDF
  thesis.ps .................................... text práce ve formátu PS
```