FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Efektivní násobení řídkých matic
Student:	Bc. Jaroslav Ryba
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

1) Analyzujte formáty ukládání řídkých matic [1,2,3,4].
2) Analyzujte základní algoritmy pro násobení řídkých matic a algoritmy používané v současných praktických implementacích.
3) Analyzujte algoritmy pro násobení matic uložených v pokročilých fornátech: ELL, BSR atd.
4) Vytvořte proof-of-concept implementaci v CUDA algoritmu pro násobení matic pro nejméně 3 různé formáty ukládání řídkých matic.
5) Proveďte testování předchozího bodu a zhodnoťte efektivitu řešení.

Seznam odborné literatury

[1] ELIÁŠ, Roman. Paralelní násobení řídkých matic. Praha, 2007. Diplomová práce. České vysoké učení technické v Praze Fakulta elektrotechnická. Vedoucí práce Ing. Ivan Šimeček.
[2] Raphael Yuster and Uri Zwick. Fast sparse matrix multiplication. ACM Transactions on Algorithms, 1(1):2–13, July 2005. ISSN 1549-6325 (print), 1549-6333 (electronic).
[3] DUFF, Iain S. Direct methods for sparse matrices / I.S. Duff, A.M. Erisman, J.K. Reid. Oxford: Clarendon Press, c1986. Numerical mathematics and scientific computation. ISBN 978-0198534211.
[4] OSTERBY, Ole a Zahari ZLATEV. Direct Methods for Sparse Matrices. Berlin: Springer-Verlag, 1983. Lecture notes in computer science.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 19. ledna 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Efektivní násobení řídkých matic

Bc. Jaroslav Ryba

Katedra softwarového inženýrství
Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

7. května 2019

Poděkování

Chtěl bych poděkovat svému vedoucímu diplomové práce doc. Ing. Ivanu Šimečkovi, Ph.D., za odborné vedení, za pomoc a rady při zpracování této práce. Děkuji také Mgr. Drahoslavě Rybové za pomoc při gramatické kontrole práce.

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 7. května 2019 .....

Odkaz na tuto práci

Ryba, Jaroslav. *Efektivní násobení řídkých matic.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato diplomová práce se zabývá implementací základu knihovny pro práci s řídkými maticemi. Dále je obsažena implementace a optimalizace masivně paralelizovaného násobení matic na GPU za využití technologie CUDA.

Práce také slouží k poskytnutí vhledu do problematiky násobení matic, řídkých matic a efektivní implementace algoritmů v technologii CUDA obecně.

Klíčová slova Násobení matic, řídké matice, CUDA, optimalizace, knihovna, GPU, C++.

Abstract

This master's thesis deals with implementation of the basis of sparse matrix library. It also contains implementation and optimalisations of massively parallel matrix multiplication on GPU in the CUDA technology.

This work is also intended to give basic level understanding of the matrix multiplication, sparse matrices and efficient implementation of algorithms under the limitations of the CUDA technology.

Keywords Matrix multiplication, sparse matrices, CUDA, optimization, library, GPU, C++.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Jedním ze základních algebraických problémů je násobení matic. Tato operace je základním stavebním kamenem mnoha v praxi významných algoritmů, například fyzikálních simulací, hledání nejkratší cesty. Často se také využívá v počítačové grafice.

Řešení tohoto problému se věnovalo již značné úsilí a za poslední století se podařilo snížit teoretickou časovou složitost vynásobení dvou matic z naivního kubického algoritmu ($O(n^3)$) až na úroveň Coppersmith-Winogradova algoritmu ($O(n^{2.372})$) a dosáhnout i značných zlepšení efektivity implementací těchto algoritmů.

Mnoho problémů však vede na násobení řídkých matic, tedy matic s velkým množstvím nulových prvků. Klasické algoritmy násobení matic nejsou schopné využít této vlastnosti matic a jejich časová složitost je nezávislá na množství nenulových prvků. Při využití této vlastnosti je však možné ušetřit značnou část výpočetních prostředků a dosáhnout tak mnohem vyšší rychlosti výpočtů.

Vzhledem k neustálému zvyšování výkonu grafických karet (GPU) a pomalému růstu výkonu CPU se v posledních letech stále častěji výpočty přesouvají na GPU a masivně paralelizují. Správně implementovaný a optimalizovaný algoritmus tak může teoreticky získat přístup k řádově sto i tisícinásobným výpočetním prostředkům. Tento trend je taktéž posílen relativní jednoduchostí implementace kódů za pomoc specializovaných technologií (CUDA, openCV).

Tato práce se zaměřuje na výše zmíněný problém násobení řídkých matic a využití výpočtů na grafické kartě pro jeho řešení. Cílem práce je poskytnout čtenáři vhled do základní problematiky násobení matic, problematiky řídkých matic a operací s nimi a současného stavu existujících knihoven pro práci s řídkými maticemi.

Dále je součástí práce navržení a implementace kostry knihovny pro práci s řídkými maticemi, která může v budoucnu sloužit jako základ pro testování

algoritmů a případně i jako základ větších projektů. V neposlední řadě je také součástí práce implementace a optimalizace několika implementací násobení řídkých matic v technologii CUDA (a integrace do vytvořené kostry knihovny).

V poslední části práce jsou vytvořené kódy testovány, vyhodnoceny a porovnány s efektivitou současné verze „konkurenční“ knihovny pro násobení řídkých matic na GPU.

Tato práce může sloužit jako studijní materiál pro zájemce o problematiku řídkých matic, praktická část pak může být využita jako základ pro vytvoření v praxi použitelné implementace efektivní knihovny pro práci s řídkými maticemi.

Cílem první kapitoly (Analýza) je seznámit čtenáře se základní problematikou násobení matic a algoritmy používanými pro násobení matic v hustých formátech. Dále se v této kapitole také čtenář dočte o formátech ukládání řídkých matic, jejich specifikách a silných stránkách. V neposlední řadě se zde také nachází rešerše současných knihoven pro práci s řídkými maticemi a informace o tom, jaké algoritmy a jaký interface tyto knihovny používají. Tyto informace budou v dalších částech využity k návrhu nové knihovny.

Ve druhé kapitole (Návrh) je nejdříve zdůvodněn výběr formátů řídkých matic pro implementaci, dále jsem navrženy a popsány sekvenční algoritmy a datové struktury pro práci s vybranými formáty a navrženy dva způsoby paralelizace algoritmů násobení matic. Je zde také popsán základ třídní struktury navržené knihovny a integrace paralelních algoritmů s knihovnou.

Ve třetí kapitole (Implementace) je popsán průběh implementace knihovny a paralelizovaného násobení. Je zde popsána škála využitých postupů pro optimalizaci CUDA (paralelizovaných) kódů a racionalizace těchto technik. Jsou zde zmíněna i některá omezení této technologie a jak se s nimi vypořádat.

V poslední kapitole (Testování) je popsán způsob testování správnosti a efektivity implementací, včetně využitého hardwaru a parametrů kompilace. Dále se zde nachází výsledky testování efektivity hlavních optimalizačních technik a verzí paralelizovaných algoritmů. V neposlední řadě se zde taktéž nachází srovnání efektivity s knihovnou cuSPARSE.

Kapitola 1

Analýza

Cílem této kapitoly je seznámení s problematikou, kterou se tato práce dále zabývá (násobení matic, řídké matice), a s přístupem „konkurenčních“ aplikací k řešení této problematiky.

1.1 Základní pojmy

V této sekci jsou definovány některé významné termíny využívané dále v práci. Není-li uvedeno jinak, je matematický základ čerpán z [2], definice jsou však parafrázovány, upraveny a zjednodušeny pro účely této práce.

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

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix},$$

kde a_{ij} jsou **prvky** matice. Číslu i říkáme **řádkový** a číslu j **sloupcový index**. [2]

Definice 1.2. Nechť $m, n, i \in \mathbb{N}$. Matici A typu $n \times 1$ budeme nazývat **sloupcový vektor** o délce n. Matici typu $1 \times m$ budeme nazývat **řádkový vektor** o délce m, zkráceně pak vektor o délce m, a jeho prvky $a_{1j}; j \leq m$ můžeme zkráceně psát a_j

Definice 1.3. Nechť $n \in \mathbb{N}$ a A, B jsou vektory o délce n. **Skalární součin** vektorů $c = A \cdot B$ definujeme takto: $c = \sum_{k=1}^{n} a_k \cdot b_k$.

Definice 1.4. Nechť $m, n \in \mathbb{N}$, $\alpha \in \mathbb{R}$ a A je matice typu $n \times m$. **Součin matice A s reálným číslem** α, značíme $C = \alpha \cdot A$, je matice typu $n \times m$, pro kterou platí

$$\forall i, j \in \mathbb{N}, i \leq n, j \leq m : c_{ij} = \alpha a_{ij}.$$

Definice 1.5. Nechť $m, n \in \mathbb{N}$ a A, B jsou matice typu $n \times m$. **Součet matic** $C = A + B$ je matice typu $n \times m$, pro kterou platí $\forall i, j \in \mathbb{N}, i \leq n, j \leq m : c_{ij} = a_{ij} + b_{ij}$.

Definice 1.6. Nechť $m, n \in \mathbb{N}$ a A je matice typu $n \times m$. **Transpozicí matice** A, značíme A^T, nazýváme matici typu $m \times n$, pro niž platí

$$\forall i, j \in \mathbb{N}, i \leq m, j \leq n : a^T_{ij} = a_{ji}.$$

Definice 1.7. Nechť $m, l, n \in \mathbb{N}$, A je matice typu $n \times l$ a B je matice typu $l \times m$. **Maticovým součinem** $C = A \cdot B$ nazýváme takovou matici typu $n \times m$, pro kterou platí

$$\forall i, j \in \mathbb{N}, i \leq n, j \leq m : C_{ij} = \sum_{k=1}^{l} a_{ik} \cdot b_{kj}.$$

Definice 1.8. Nechť $m, n, l \in \mathbb{N}$ a A je matice typu $n \times m$. **Diagonálou matice** A nazveme takový vektor D délky l, pro který $\exists c \in \mathbb{Z}$ tak, že platí

$$(\forall i \in \mathbb{N}, i \leq l : d_i = a_{i,i+c}, c \geq 0 \wedge c + l = m) \vee$$

$$(\forall i \in \mathbb{N}, i \leq l : d_i = a_{i-c,i}, c \leq 0 \wedge c + l = n)$$

. Hodnotu c pak nazýváme **offsetem diagonály**.

Definice 1.9. Řekneme, že matice $A = (a_{i,j})$ je **pásovou maticí**, pokud existují nezáporné konstanty p, q (nazývané levý a pravý polo-pás) takové, že $a_{i,j} \neq 0$ pouze pokud $i - p \leq j \leq i + q$.[3]

Definice 1.10. Nechť $m, n \in \mathbb{N}$, A je matice typu $n \times m$. **Hustotou matice** A nazveme $c \in \mathbb{R}$, pro něž platí

$$c = \frac{|\{a_{ij}; i, j \in \mathbb{N} \wedge i \leq n \wedge j \leq m \wedge a_{ij} \neq 0\}|}{m \cdot n}.$$

Neexistuje oficiální definice řídké matice. Pro naše účely však postačí tato pragmatická definice od J. H. Wilkinsona:

Definice 1.11. Nechť $m, n \in \mathbb{N}$, o matici A typu $n \times m$ řekneme, že je **řídká**, pokud můžeme využít faktu, že část jejích hodnot je nulová.

Definice 1.12. Nechť f, g jsou funkce na přirozených číslech. Řekneme, že g je **asymptotickou horní mezí** f, značíme $f = O(g)$ právě tehdy, když:

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

Definice 1.13. Nechť f, g jsou funkce na přirozených číslech a $x \in \mathbb{N}$. **Výpočetní složitostí** implementace algoritmu I nazveme takovou funkci $f(x)$, pro niž platí $g = O(f)$ a $g(x)$ je funkcí závislosti počtu provedených operací na velikosti vstupu.

Definice 1.14. Nechť f, g jsou funkce na přirozených číslech a $x \in \mathbb{N}$. **Paměťovou náročností** implementace algoritmu I nazveme takovou funkci $f(x)$ pro niž platí $g = O(f)$ a $g(x)$ je funkcí závislosti množství využité paměti na velikosti vstupu.

Definice 1.15. Časovou efektivitou E implementace algoritmu I (též zkráceně efektivitou algoritmu, nebo výkonem algoritmu) na hardwaru H se vstupem V rozumíme množství operací typu T, které při spuštění I na H průměrně proběhne za jednotku času.

Pro účely této práce (násobení řídkých matic) pak budou T z 1.15 operace s reálnými čísly (plovoucí řádovou čárkou), konkrétně jejich sčítání a násobení. Tuto hodnotu získáme jako podíl provedených operací a času spotřebovaného na provedení výpočtu. Pro tuto veličinu budeme používat jednotku FLOPS (počet operací s řádovou čárkou za sekundu).

1.2 Formáty řídkých matic

V této kapitole popíši některé základní způsoby ukládání řídkých matic. Pro ilustraci ukládání bude využita jednoduchá matice 4x4, kterou můžete vidět v tabulce 1.1.

Za desetiletí vývoje vzniklo nespočet různých formátů ukládání matic. Cílem této části tedy není popis všech, ale několika vybraných. Formáty byly vybírány dle jejich významnosti (rozšíření) a relevantnosti k dalšímu obsahu této práce.

Základní ukládání matice v hustém formátu je zřejmé a nebude v této práci dále rozebíráno.

Tabulka 1.1: Vzorová hustá matice

$$\begin{bmatrix} 0 & 1.1 & 0 & 2.0 \\ 2.3 & 0 & 0 & 2.4 \\ 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 0.4 \end{bmatrix}$$

1.2.1 COO

Informace v této kapitole vychází z [4].

Nejintuitivnějším způsobem ukládání řídkých matic je pomocí pozic a hodnot nenulových prvků (řádek, sloupec, hodnota). Tento formát se obvykle označuje jako COO (coordinate format - souřadnicový formát). Pro lepší práci s pamětí (cache) je vhodné ukládat tyto hodnoty pomocí tří polí. Dle způsobu využití se také může hodit hodnoty seřadit dle pozice.

Dříve zmíněná vzorová matice (1.1) se dá tímto způsobem zapsat tak, jak je vidět v tabulce 1.2.

Tabulka 1.2: Vzorová matice z tabulky 1.1 ve formátu COO

řádek	0	0	1	1	2	3
sloupec	1	3	0	3	2	3
hodnota	1.1	2.0	2.3	2.4	1.0	0.4

Lze snadno nahlédnout, že pro matici $n \times m$ s nnz nenulovými hodnotami k uspoření místa dojde, je-li

$$nnz \cdot (valueSize + 2 \cdot indexSize) < n \cdot m \cdot valueSize,$$

kde $valueSize$ je počet bitů k uložení hodnoty a $indexSize$ je počet bitů na uložení indexu. Pokud dále předpokládáme $valueSize = indexSize$ (například obě hodnoty 32, což je standardní velikost), dosáhneme úspory, je-li $\frac{nnz}{n \cdot m} < \frac{1}{3}$.

1.2.2 CRS

Definice v této kapitole částečně převzata z [5] a rozšířena o doplňující informace z [6] a [4].

Compressed row storage (CRS) lze označit za inkrementální vylepšení COO. Hodnoty prvků i sloupcové indexy jsou ukládány stejným způsobem (2 vektory), zde je však již přímo vyžadováno jejich seřazení (dle řádků a sloupců v tomto pořadí důležitosti). Řádkový vektor pak obsahuje informaci o počtu prvků v jednotlivých řádcích. Pro urychlení přístupu k prvkům (aby nemusel být tento vektor procházen od začátku při čtení prvku) je pak tato informace uložena ve formě prefixového součtu počtů prvků (tedy kolikátý prvek je první v tomto řádku, počítáno od nuly), tyto hodnoty budou dále nazývány offset řádku. Je standardem (pro zjednodušení algoritmů) rozšířit vektor offsetů řádků o jeden prvek, kterým je počet nenulových prvků v matici, tedy offset n+1. (neexistujícího) řádku.

Dříve zmíněná vzorová matice (1.1) se dá tímto způsobem zapsat tak, jak je vidět v tabulce 1.3.

Paměťová náročnost tohoto formátu (se stejným značením a předpoklady jako u COO) je $2nnz + n + 1$, což je u matice bez prázdných řádků hodnota menší nebo rovna paměťové náročnosti COO (pokud bereme v úvahu, že i u COO je nezbytné uložit počet nenulových prvků). Čím hustší je matice, tím větší výhody CRS přináší, k rovnosti dojde u matice s přesně jedním prvkem na řádek.

Tabulka 1.3: Vzorová matice z tabulky 1.1 ve formátu CRS

ofset řádku	0	2	4	5	6
sloupec	1	3	0	3	2	3
hodnota	1.1	2.0	2.3	2.4	1.0	0.4

1.2.3 CCS

Základní popis přebrán z [6].

Compressed column storage (CCS), také nazývaný Harwell-Boeingův formát řídkých matic, je analogický formát k CRS, kde místo po řádcích je vektor procházen po sloupcích a jsou zaznamenávány řádkové indexy hodnot. Tento formát se dá také chápat jako CRS transponované matice.

Jak lze z výše uvedeného popisu pochopit, vlastnosti tohoto formátu jsou stejné jako u CRS a tento formát může být vhodný pro násobení s maticí v CRS formátu.

Tabulka 1.4: Vzorová matice z tabulky 1.1 ve formátu CCS

ofset sloupce	0	1	2	3	6
řádek	1	0	2	0	1	3
hodnota	2.3	1.1	1.0	2.0	2.4	0.4

Dříve zmíněná vzorová matice (1.1) se dá tímto způsobem zapsat tak, jak je vidět v tabulce 1.4.

1.2.4 BCRS

Definice využívá informace z [5] s rozšířením o [6] a [4].

Block compressed row storage (BCRS, někdy také BSR), je nadstavba nad CRS, která využívá vlastnosti matic majících nenulové hodnoty shlukované (prostorově) blízko u sebe.

Původní matice je nejdříve rozložena na podmatice (bloky) řádově menší velikosti (například 2x2) a následně je tato matice bloků uložena pomocí CRS, přičemž blok s pouze nulovými hodnotami je brán jako nulový element. Při ukládání hodnot je po řádcích zapsán celý blok, nikoliv jen jeho nenulové elementy.

Pro velikost bloku 2x2 nám tedy pro naši vzorovou matici (1.1) vznikne zápis v tabulce 1.5.

Tabulka 1.5: Vzorová matice z tabulky 1.1 ve formátu BCRS

ofset řádku			0	2	3							
sloupec			0	1	1							
hodnota	0	1.1	2.3	0	0	2.0	0	2.4	1.0	0	0	0.4

Adresovat je pak nutné dvoufázově (vyhledat adresu bloku a následovně přičíst pozici v bloku), což ovšem na moderním hardwaru nepřidává prakticky žádnou výpočetní složitost. Přínosy ukládání matic v tomto formátu jsou velmi závislé na struktuře matice a vhodně zvolené velikosti bloku. Čím více je matice tvořená shluky hodnot, tím lepších výsledků lze dosáhnout.

Ačkoliv tato metoda ukládání může (kvůli ukládání i nulových elementů) vést k navýšení paměťových nároků oproti CRS, je v praxi často využívána pro lepší využití cache a obecně lepší výkonnost některých algoritmů.

1.2.5 DIA

Definice parafrázována z [1].

Diagonal format (DIA), také známý jako compressed diagonal storage (CDS), je vhodný, pokud jsou nenulové hodnoty soustředěny do malého množství diagonál. Matice je uložena za pomoci dvou polí: pole hodnot a fsetů diagonál od hlavní diagonály. Diagonály nad a pod hlavní diagonálou mají negativní respektivě pozitivní offsety. Do pole hodnot jsou hodnoty ukládány postupně od diagonály s nejnižším offsetem k nejvyššímu a v rámci diagonál dle sloupcového indexu. Ukládány jsou jsou i nulové hodnoty a diagonály jsou doplněny jako by začínaly na prvním a končily na posledním řádku.

Ukázku tohoto formátu můžete vidět na obrázku 1.1. Neexistující prvky (*) mohou být nahrazeny například 0.

Obrázek 1.1: Diagonální formát, převzato z [1]

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

DIA

hodnoty	*	*	5	6	1	2	3	4	7	8	9	*
offsety diagonál					-2	0	1					

Výhodou tohoto formátu je vysoká úspornost pro specifické typy matic a efektivní paralelizace některých maticových algoritmů (např. násobení matice vektorem). Nevýhodou je vysoký nárůst náročnosti (paměťové i výpočetní) pro matice nemající vhodnou strukturu a s tím spojená omezená kategorie matic, na něž lze formát efektivně aplikovat.

1.2.6 ELL

Definice parafrázována z [1].

ELLPACK (ELL) je další specializovaný formát. Podmínka tohoto formátu je však volnější než u DIA. Místo nízkého počtu diagonál zde požadujeme, aby byly prvky rozloženy mezi řádky co nejrovnoměrněji. Přesněji chceme, aby řádek s nejvíce nenulovými elementy byl co nejkratší.

Matice o n řádcích s nejvíce m nenulovými prvky na řádek je uložena pomocí dvou hustých $n \times m$ matic. V první z těchto matic jsou ukládány nenulové hodnoty v pořadí dle indexů sloupců, tedy jako kdyby z původní matice byly vypuštěny nuly a ostatí hodnoty „sraženy“ doleva. Do druhé matice jsou ukládány původní sloupcové indexy těchto prvků. Obě tyto matice jsou zprava doplněny nulami (aby byla zachována stejná délka všech řádků).

Obrázek 1.2: Vzorová matice z tabulky 1.1 ve formátu ELL

$$hodnoty = \begin{bmatrix} 1.1 & 2.0 \\ 2.3 & 2.4 \\ 1.0 & 0 \\ 0.4 & 0 \end{bmatrix} \quad indexy = \begin{bmatrix} 1 & 3 \\ 0 & 3 \\ 2 & 0 \\ 3 & 0 \end{bmatrix}$$

Dříve zmíněná vzorová matice (1.1) se dá v ELL zapsat tak, jak je vidět na obrázku 1.2. I v tomto případě mohou být samozřejmě pro vyšší efektivitu matice rozepsány do dvou 1D polí.

ELL se dá považovat za rozšíření DIA. Zatímco u DIA je pozice prvku implicitní (dle pozice diagonály), u ELL je explicitně určená, což umožňuje jej využít pro mnohem širší škálu matic.

Tento formát je také velmi vhodný pro vektorové architektury a paralelizaci a efektivní práci s cache.

1.2.7 Hybridní formát

Tato část čerpá z [1].

Hybridní formát (HYB) je kombinace ELL a COO. Cílem využití tohoto formátu je zvýšit použitelnost ELL na obecnější případy a zároveň zachovat (částečně) jeho efektivitu.

Prvních N prvků každého řádku se uloží pomocí ELL, zbývající prvky se uloží pomocí COO. Cílem je vytvořit jakési „jádro“ matice, které bude uložené v ELL a bude obsahovat pouze malé množství nulových hodnot.

Složitějším problémem je určení vhodného N. Tato hodnota závisí na struktuře matice a výkonnosti algoritmů nad COO a ELL. Víme-li tento poměr výkonností, můžeme tuto hodnotu již přesně určit výpočtem z histogramu počtů nenulových elementů na řádek.

Příklad (vzorové) matice uložené v hybridním formmátu (s $N = 1$) si můžete prohlédnout na obrázku 1.3.

Obrázek 1.3: Vzorová matice z tabulky 1.1 v hybridním formátu

$$hodnotyELL = \begin{bmatrix} 1.1 \\ 2.3 \\ 1.0 \\ 0.4 \end{bmatrix} \; indexyELL = \begin{bmatrix} 1 \\ 0 \\ 2 \\ 3 \end{bmatrix}$$

řádek	0	1
sloupec	3	3
hodnota	2.0	2.4

1.2.8 Dynamické formáty

Tato část čerpá z [7] a [8].

Většina formátů ukládání řídkých matic je velmi neefektivní pro změny matice. Zvláště neefektivní jsou operace přidávání a odebírání hodnot, které u většiny formátů často vynutí přepis celé matice (což je operace se složitostí $O(nnz)$). Dynamické formáty vychází ze základních formátů pro ukládání řídkých matic (nejčastěji CRS, BCRS a COO) a snaží se odstranit tento problém při zachování nízké paměťové náročnosti.

Dynamické formáty mohou být realizovány pomocí:

- Spojových seznamů (jednosměrných, obousměrných i cyklických) - tento způsob umožňuje velmi rychlé přidávání a odebírání prvků a je snadno implementovatelný (často používán pro COO).
- Stromových struktur (například trie) - tento způsob je vhodný pro časté čtení hodnot matice s náhodným přístupem, je však významně komplikovanější na implementaci a zajištění efektivity algoritmů pracujících s celou maticí
- Pole polí (pole pointerů) - pole hodnot (indexů) je rozděleno na části (obvykle samostatné řádky/sloupce), které jsou ukládány do samostatně alokovaných polí a místo offsetů řádků je uchováván celý pointer, což umožňuje minimalizovat dopad přidání prvků pouze na jeden řádek ($O(nnz_m)$) místo celé matice (často používán pro CRS, BCRS).
- Rozvolnění formátů - mezi uložené hodnoty jsou přidány výplňové prvky (nulové), které jsou přepsány v případě přidání nových prvků a v případě odebírání prvků jsou prvky nahrazeny výplňovými hodnotami.

Případně mohou být využity kombinace několika z těchto technik (například dále zmíněný Eigen využívá 3. a 4.).

Výhodami těchto formátů jsou flexibilita pro vkládání a odebírání prvků, případně rychlejší přístup k nim. Nevýhodami pak jsou obtížnost zachování

paměťové lokality (využití cache), obtížnost vektorizace/paralelizace, vyšší paměťová náročnost a obecně nižší efektivita pro algoritmy pracující systematicky po řádcích/sloupcích (mezi které patří i násobení matic).

1.3 Algoritmy násobení hustých matic

1.3.1 Klasický algoritmus

Základní (a pro mnohé případy i nejlepší) postup vyplývá ze vzorce pro násobení matic a můžete jej vidět v algoritmu 1.

Algoritmus 1 Klasické násobení matic

```
1: procedure NaiveMultiply(A, B)                          ▷ Vynásob A*B
2:     C ← zeroes(A.height, B.width)                      ▷ Vytvoř 2D pole na ukládání
   výsledků a nastav jej na nuly
3:     for i ← 0 to A.height do
4:         for j ← 0 to B.width do
5:             for k ← 0 to A.width do
6:                 C[i][j] ← C[i][j] + A[i][k] * B[k][j]
7:     return C                                           ▷ Matice s výsledky
```

Pro zvýšení efektivity je samozřejmě možné vhodně změnit pořadí procházení a případně využit transpozice.

Výhodami tohoto algoritmu jsou jednoduchost jeho implementace, snadnost paralelizace, využitelnost pro specializované formáty ukládání (řídké matice) a vysoká efektivita pro relativně malé matice.

Jeho nevýhodou je pak jeho asymptotická složitost, která je $\Theta(n^3)$, a z toho plynoucí pomalost (vůči dále zmíněným algoritmům) při aplikaci na velké matice (řádově 1000×1000 prvků).

1.3.2 Strassenův algoritmus

Strassenův algoritmus [9] je založen na principu rozděl a panuj (rekurzivní přístup). Asymptotická složitost tohoto algoritmu je $O(n^{log_2 7}) \approx O(n^{2.808})$. Důkaz správnosti algoritmu je po dosazení do výsledného vzorce zřejmý, a proto nebude rozváděn.

Chceme-li vypočítat maticový součin $C = A \cdot B$, rozdělíme nejdříve každou matici na 4 stejně velké podmatice

$$C = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \cdot \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right].$$

Následně je nutné vypočítat hodnoty 7 mezivýsledků

$$\begin{aligned}
M_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}),\\
M_2 &= (A_{21} + A_{22}) \cdot B_{11},\\
M_3 &= A_{11} \cdot (B_{12} - B_{22}),\\
M_4 &= A_{22} \cdot (B_{21} - B_{11}),\\
M_5 &= (A_{11} + A_{12}) \cdot B_{22},\\
M_6 &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}),\\
M_7 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}).
\end{aligned}$$

Nakonec se za použití těchto pomocných matic vypočítá výsledná matice

$$C = \left[\begin{array}{c|c} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ \hline M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{array}\right].$$

Celkově je tedy nezbytné pro jednu úroveň Strassenova algoritmu provést 18 součtů ($\Theta(n^2)$) a 7 násobení ($\Theta(n^3)$). Pro dostatečně velké matice je jedna úroveň Strassenova algoritmu oproti klasické variantě rekurzivního násobení matic, která využívá 4 součty a 8 násobení, tedy až o 12.5 % efektivnější.

Nevýhodou tohoto algoritmu je nízká efektivita na menších maticích (kvůli většímu počtu sčítání a celkové větší komplexnosti) a zvýšená paměťová náročnost. I bez přepisování vstupních matic lze však algoritmus provést s paměťovou složitostí $O(\frac{2n}{3})$ [10].

1.3.3 Winogradova varianta Strassenova algoritmu

Winogradova varianta Strassenova algoritmu [11] je algoritmus založený na stejném principu jako původní Strassenův algoritmus, který však místo 18 využívá pouze 15 součtů. Ačkoliv toto zlepšení nemá žádný vliv na asymptotickou složitost, v praktické implementaci se projeví drobným zlepšení efektivity algortimu.

Celé schéma algoritmu je (značení A, B, C převzato z vysvětlení Strassenova algoritmu):

$$\begin{array}{lll}
S_1 = A_{21} + A_{22}, & M_1 = S_2 \cdot S_6, & V_1 = M_1 + M_2,\\
S_2 = S_1 - A_{11}, & M_2 = A_{11} \cdot B_{11}, & V_2 = V_1 + M_4,\\
S_3 = A_{11} - A_{21}, & M_3 = A_{12} \cdot B_{21}, & C_{11} = M_2 + M_3,\\
S_4 = A_{12} - S_2, & M_4 = S_3 \cdot S_7, & C_{12} = V_1 + M_5 + M_6,\\
S_5 = B_{12} - B_{11}, & M_5 = S_1 \cdot S_5, & C_{21} = V_2 - M_7,\\
S_6 = B_{22} - S_5, & M_6 = S_4 \cdot B_{22}, & C_{22} = V_2 + M_5,\\
S_7 = B_{22} - B_{12}, & M_7 = A_{22} \cdot S_8, & \\
S_8 = S_6 - B_{21}. & &
\end{array}$$

Například využitím 4 iterací tohoto algoritmu (pro matice 16384×16384) lze prakticky dosáhnout snížení počtu aritmetických operací o 41.3 % oproti klasickému algoritmu [10].

Paměťová náročnost zůstává stejná jako u původního Strassenova algoritmu[11].

1.3.4 Další algoritmy

V současné době existuje již řada algoritmů s nižší asymptotickou složitostí než Strassenův algoritmus ($O(n^{w=2.808})$). V roce 1978 vznikl Panův algoritmus ($w = 2.796$), o rok později algoritmus od Bimi a spol. ($w = 2.78$). Následovala řada dalších algoritmů, které překonávaly do té doby nejlepší dosažené výsledky. Z dalších známých algoritmů stojí za zmínku například Coppersmith-Winogradův algoritmus ($w = 2.376$) z roku 1989 (později sníženo na $w = 2.372$), který je dodnes nejrychlejším známým algoritmem.[12]

Ačkoliv tyto algoritmy jsou asymptoticky rychlejší, matice, pro něž jsou tyto algoritmy oproti klasickému (a také Strassenovu) algoritmu výhodné, jsou natolik velké, že nejsou zpracovatelné na současném hardwaru. Zvláště pak mnohé z novějších algoritmů jsou pouze teoretického rázu a v praxi (například kvůli nepřesnostem při výpočtech u čísel s plovoucí desetinnou čárkou) nepoužitelné.

1.3.5 Využití subkubických algoritmů pro řídké matice

Pro implementaci Strassenova algoritmu (a jeho variant), což je jediný v současnosti používaný subkubický algoritmus, je nutné dokázat matici horizontálně a vertikálně rozdělit. Tato operace je u husté matice jednoduchá (nemusí proběhnout žádné čtení, stačí upravit způsob indexování do matice a meze), na druhou stranu u řídkých matic se jedná o operaci velmi složitou, s časovou náročností $O(nnz)$, tedy přečtení celé matice a její znovuzapsání do několika nových, což značně zmenšuje přínos Strassenova algoritmu i u sekvenčních implementací.

Efektivita násobení matic na grafické kartě (masivně paralelizovaná) je obvykle mnohem více omezena využitím paměti, než výkonem výpočetních jednotek. Pro masivní paralelizaci je tedy tento algoritmus, který značně zvyšuje počet paměťových operací a množství využité paměti, naneštěstí ještě méně vhodný, než pro sekvenční verzi a proto nebude jeho praktická implementace uvažována.

1.4 Rešerše existujících knihoven

1.4.1 NIST Sparse BLAS

Informace v této sekci vycházejí z uživatelského manuálu [13] a zkoumání kódů dostupných z [14].

Touto knihovnou se budu nejvíce zabývat nejen kvůli její rozšířenosti, ale zároveň protože její kód je open-source, a lze ji proto snadno a důkladně analyzovat.

1.4.1.1 Interface

Celou funkcionalitu této knihovny lze rozdělit do 4 kategorií:

- Level 1 - operace typu *vektor* × *vektor*
- Level 2 - operace typu *matice* × *vektor*
- Level 3 - operace typu *matice* × *matice*
- Ostatní - inicializace, destrukce, getry, setry

Z výše uvedených kategorií je pro tuto práci nejvýznamnější Level 3, protože právě ten obsahuje násobení matic. Dále stojí za zmínku také životní cyklus matic.

Práce s řídkými maticemi sestává z těchto tří kroků:

- Vytvoření matice a získání handleru k ní (int)
- Použití handleru jako argumentu při výpočtech s maticí
- Když matice již není potřebná, je explicitně (uživatelem) zavolána čisticí funkce, která uvolní využívané prostředky

Pro nás nejvýznamnější funkce násobení matic má pak základní tvary:

$$C \leftarrow \alpha \cdot A \cdot B + C,$$

$$C \leftarrow \alpha \cdot A^T \cdot B + C,$$

a pro matice komplexních čísel

$$C \leftarrow \alpha \cdot A^H \cdot B + C,$$

kde A je řídká matice, B, C jsou husté matice a α je reálné číslo (float). Starší verze podporovaly navíc koeficient β pro sčítání s C. Tato funkce pak existuje ve 4 variantách pro různé typy prvků matic (jednoduchá, dvojitá přesnost atd.).

Násobení dvou řídkých matic knihovna neumožňuje. Při volání funkcí jsou řídké matice reprezentovány handlerem, husté matice 1D polem (pointer), způsobem, jak jsou uloženy (po řádcích, sloupcích - enum), a velikostí jejich hlavní dimenze (int). Výběr varianty s transpozicí, či konjungovanou transpozicí je reprezentován enumem.

Tato knihovna vnitřně (nedeklarováno v hlavičkových souborech) reprezentuje matice pomocí formátu BCRS, konkrétní parametry se nedají z vnějšku ovlivnit. Dále také umožňuje specializované algoritmy a ukládání pro matice s význačnými vlastnostmi (symetrické, triangulární).

Knihovna NIST Sparse BLAS vznikala původně v jazyce Fortran, což je poznat i ze způsobu práce s maticemi a z interface funkcí. Jak bylo však již zmíněno v [15], tento inferface má pro využití v C++ mnoho problémů a pro účely této práce by jej bylo vhodné vylepšit. Zejména se jedná o životní cyklus matic. Zde by bylo vhodné využít možností C++ práce s objekty a držet se obvyklejších standardů pro jejich vytváření a ničení (konstruktory, destruktory). Dále pak veškeré parametry týkající se jedné matice (pořadí prvků, délka dimenze, data) by měly být sjednoceny do jednoho objektu. Bude-li podporováno více typů prvků, měly by navíc být funkce přetíženy (nebo využity šablony) místo několika různých jmen funkcí.

1.4.1.2 Algoritmy

Algoritmy této knihovny vychází z přímé implementace klasického algoritmu násobení matic, samozřejmě s odlišným způsobem adresování (řídké matice obvykle nemají přímý přístup k prvkům). Ve zdrojových kódech jsem nenalezl žádné specifické triky a techniky sloužící k zlepšení efektivity kromě rozdělení funkcí dle parametrů na mnoho jednodužších funkcí, místo využití větvení uvnitř jádra funkcí.

1.4.2 Eigen

Informace o této knihovně pochází z dokumentace dostupné na [16] a studia kódu dostupného z [17].

Eigen je open-source knihovna napsaná v C++, zabývající se lineární algebrou. Pro nás významně se věnuje mimo jiné i řídkým maticím a operacím s nimi.

1.4.2.1 Interface

Interface této knihovny je mnohem bližší objektovému programování (a moderním standardům C++). S maticemi se tedy pracuje pro objekty obvyklým způsobem (konstruktory, destruktory, metody).

Z operací s řídkými maticemi umožňuje knihovna (mimo jiné) sčítání, násobení skalárem, transpozici (a získání hermitiánu matice pro komplexní čísla, vice o hermitianu například [18]), násobení dvou matic a operace aplikované po prvku (tedy výsledek získán jako $C_{i,j} = op(A_{i,j}, B_{i,j})$).

Významnou změnou oproti NIST Sparse BLAS je, že výsledkem operace se dvěma řídkými maticemi je znovu řídká matice, nikoliv matice hustá.

Pro ukládání řídkých matic je využíván upravený formát CRS/CCS. Úprava spočívá v ponechávání volných míst pro přidávání prvků. Navíc v některých

místech využívá nekomprimovaného formátu, ve kterém jsou jednotlivé řádky (případně sloupce) ukládány do samostatných polí (ve formátu odpovídajícím jednomu řádku/sloupci CRS/CCS) a na ně jsou uložené odkazy (což umožňuje jednodužší a rychlejší přidávání prvků do matice).

1.4.2.2 Algoritmy

I zde je využit základní algoritmus násobení matic. Výsledná matice je vytvářena sekvenčně po sloupcích. Celý algoritmus je významně komplikován tím, že výsledná matice je také v říkém formátu. Například je nutné odhadovat množství nenulových prvků ve výsledném sloupci (prováděno velmi jednoduchým odhadem na základě předpokladu, že max 1 index nenulového prvku se liší). Algoritmus vyžaduje mnoho dynamických alokací (a realokací) a velmi složitě paralelizovatelný.

1.4.3 cuSPARSE

Informace v této sekci vychází z dokumentace dostupné na [19].

Knihovna cuSPARSE obsahuje sadu low-level funkcí pro práci s řídkými maticemi na GPU a je určená pro jazyky C a C++.

Konkrétní algoritmy používané v cuSPARSE nebudou rozebírány, protože kódy knihovny nejsou veřejně k dispozici. Tato knihovna se však (kvůli podobné problematice, kterou se zabývá) dá využít pro porovnání efektivity s kódy implementovanými během této diplomové práce.

1.4.3.1 Interface

Interface této knihovny je značně podobný s NIST Sparse BLAS. V cuSPARSE se nachází mírně obecnější funkce typu

$$C \leftarrow \alpha op(A) \cdot op(B) + \beta \cdot C,$$

(cusparse<t>csrmm2) kde $op()$ značí identitu, transpozici, nebo hermitián. Kromě matice A (a $op(A)$), která je v řídkém formátu, jsou všechny ostatní matice husté a α, β jsou reálná čísla.

Tato knihovna ovšem navíc obsahuje i funkci násobení dvou řídkých matic. Konkrétně se jedná o funkce typu

$$C \leftarrow \cdot op(A) \cdot op(B)$$

(cusparse<t>csrgemm) a

$$C \leftarrow \alpha \cdot op(A) \cdot op(B) + \beta \cdot D$$

(cusparse<t>csrgemm2). V těchto dvou případech jsou všechny matice v CRS formátu. Tyto funkce však nejsou zcela implementovány, takže $op()$ může reprezentovat pouze identitu (tedy bez transpozic) a nejsou podporována žádná

pokročilejší ukládání matic, které knihovna v jiných částech umožňuje (symetrické, trojúhelníkové aj.). Před voláním těchto funkcí je navíc nutné volat další funkce pro výpočet správné alokace matice C a v druhém případě i alokaci bufferu (jehož účel není přesně specifikován). Tyto dvě funkce budou dále použity k porovnání efektivity s kódy vzniklými jako součást této práce.

Matice jsou předávány dle způsobu obvyklého v jazyce C, tedy po jednotlivých polích a proměnných. Toto však vede k značně velkému množství argumentů (například 27 pro csrgemm2), což zvláště v kombinaci s dříve zmíněným vícefázovým voláním funkcí není příliš uživatelsky přívětivé.

Kapitola 2

Návrh

V první fázi návrhu je nezbytné určit, s jakými formáty matic bude implementace pracovat a jaký interface budou mít metody pro práci s maticemi. Dále je nutné navrhnout základní sekvenční algoritmy pro práci s jednotlivými formáty a určit, jak tyto algoritmy paralelizovat. Nakonec je také nezbytné vytvořit základní návrh tříd.

2.1 Interface

Základ interface funkce násobení (a sčítání) jsem se rozhodl čerpat z knihovny NIST Sparse BLASS, ovšem za využití některých prvků používaných knihovnou Eigen (a obecně objektovým programováním). Navíc jsem se vzhledem k charakteru prototypu a malým přínosům opaku rozhodl podporovat pouze ukládání hodnot ve formě čísel s plovoucí řádovou čárkou s jednoduchou přesností (tedy v C++ float).

Bude tedy prováděna operace:

$$C \leftarrow \alpha \cdot A \cdot B + C,$$

$$C \leftarrow \alpha \cdot A^T \cdot B + C.$$

Argumenty operace budou konstantní pointery na matice A, B, hodnota α (float) a pointer na výstupní hustou matici C, jehož nulovost znamená, že má být alokována nová matice. Dále bude volání funkce obsahovat dvě výběrové hodnoty (enum), které určují provedení transpozice matice A před výpočtem a zda má být výpočet proveden paralelně.

2.2 Formáty matic

Z interface operace násobení je zřejmé, že musí být přítomný formát pro husté matice.

Z formátů pro ukládání řídkých matic byly dále vybrány:

- CRS - nejčastěji používaný, základní formát ukládání řídkýc matic
- BCRS - založený na CRS, vhodný pro blokové matice
- DIA - prakticky nejefektivnější formát, má však velmi omezené použití (pro matice s malým množstvím diagonnál)
- ELL - nadstavba nad DIA (a CRS), obecně považován za efektivní a značně obecný (pro matice s rovnoměrně rozloženými prvky na řádky)

Formát CCS nebyl specificky přidán, protože je prakticky shodný s CRS transponované matice. Formát COO je pro samotné násobení nevhodný a paměťově náročný, proto byl také z vnitřní reprezentace vyřazen, ačkoliv převody mezi ním a ostatními formáty se ukázaly být nezbytné pro získání testovacích matic (viz sekci testování). Dynamické formáty byly vynechány z důvodu nízké efektivity algoritmů násobení (tedy opaku cíle této práce) a toho, že tato práce necílí na snadné dynamické měnění matic. Hybridní formát nebyl přidán kvůli jeho složitosti a jeho přidáním by byl významně překročen zamýšlený rozsah práce.

2.3 Sekvenční algoritmy

Cílem je implementovat (a paralelizovat) funkci $\alpha \cdot A^{(T)} \cdot B + C$, kde A, B jsou dvě řídké matice uložené ve stejném formátu a C je hustá matice, sloužící zároveň k uložení výsledku. K implementaci této funkce (a ověření funkčnosti) je zapotřebí 4 dílčích funkcí. Tyto funkce jsou:

- Hustá matice $\rightarrow$ řídká matice (vytvoření)
- Transpozice
- Násobení (s akumulací do výsledkové matice)
- Řídká matice $\rightarrow$ hustá matice (pro kontrolní výsledky a testování)

Z nich poslední je obvykle velmi jednoduchá (a zřejmá), proto zde nebude specificky popisována. Jediné násobení pak má (asymptoticky) vyšší výpočetní složitost než složitost paměťovou, a proto je jej vhodné paralelizovat. Všechny ostatní funkce mají stejnou paměťovou a výpočetní složitost ($O(nnz)$), a proto jejich paralelizace pomocí GPU (kvůli přenosům dat) nepřinese žádné zrychlení.

Abych předešel exponenciálnímu nárůstu počtu nutných impelementací násobení, rozhodl jsem se implementovat pouze násobení matic ve stejném formátu (u BCRS navíc se stejnou velikostí bloku).

2.3.1 CRS

2.3.1.1 Datová struktura

Z definice formátu vyplývá, že pro uložení matice typu $n \times m$ s nnz nenulovými elementy budou potřeba nejméně 3 pole. Pole offsetů (int) o délce $n + 1$ (na pozici n je uloženo nnz), pole indexů sloupců (int) o délce nnz a pole hodnot (float) také o délce nnz. Aby bylo možné reprodukovat původní matici, je dále nezbytné mít uloženou nejen hodnotu n, ale i m.

2.3.1.2 Převod z husté matice

Jak můžete vidět na algoritmu 2, převod je poměrně intuitivní (algoritmus přesto přepsán, protože se bude hodit při popisu dalších formátů). Za zmínku stojí především řádek 4. Tuto operaci lze odebrat a využít hodnoty *count* z hlavního for cyklu. Problém tohoto přístupu však je, že bychom museli využívat dynamické alokace polí (taktéž $O(n \cdot m)$, ovšem pravděpodobně s vyššími konstantami), takže výsledný kód by byl pomalejší a zároveň složitější.

Algoritmus 2 Hustá matice → CRS

```
1:  procedure CRS(M : DenseMatrix)
2:      C : CRS                                          ▷ Tvořená matice
3:      C.height ← M.height; C.width ← M.width
4:      nnz ← count_nnz(M)                               ▷ O(n · m)
5:      C.offsets ← zeroes(C.height)
6:      C.indexes, C.values ← zeroes(nnz)
7:      count ← 0
8:      for i ← 0 to M.height do
9:          C.offsets[i] ← count
10:         for j ← 0 to M.width do
11:             if notEmpty(M[i][j]) then
12:                 C.values[count] ← M[i][j]
13:                 C.indexes[count++] ← j
14:     C.offsets[M.height] ← nnz
15:     return C
```

2.3.1.3 Transpozice

Je zřejmě nežádoucí provádět tuto transpozici přes hustou matici ($O(n \cdot m)$). Jako druhá možnost se jeví přeřazení prvků. Obvyklé řadící algoritmy mají složitot $O(nnz * \log nnz)$, což je významně lepší, avšak ne ideální. Můžeme se však inspirovat z counting sortu. Je podstatné, že hodnoty jsou již seřazené dle řádků. Stačí tedy, abychom je counting sortem seřadili podle sloupců, a díky

stabilnosti tohoto algoritmu získáme požadované seřazení, a tedy i transponovanou matici. Pole offsetů získáme přímo při provádění counting sortu (prefixový součet počtů jednotlivých hodnot), jen jej oproti standardní variantě musíme posunout o jedno pole doprava, tedy tak, aby

$$\forall x \in \mathbb{N} \cup \{0\}, x \leq m : \mathit{offset}_x = \sum_{k=0}^{x-1} nnz_k,$$

kde nnz_k je počet nenulových prvků ve sloupci k.

Složitost celé transpozice je (ze složitosti counting sortu) $O(nnz + m)$ a za předpokladu, že $nnz \geq m$ (tedy, že v každém sloupci je průměrně alespoň jeden prvek) je tato složitost $O(nnz)$, což je optimální složitost, protože celou matici musíme přepsat.

2.3.1.4 Násobení

Algoritmus násobení vychází z klasického postupu násobení (viz algoritmus 1, dále indexy značeny dle tohoto algoritmu). Důležité však je, v jakém pořadí budeme matice procházet. Formát CRS je vhodný pouze k procházení po řádcích. Můžeme však využít vlastnosti, že transpozicí uložené matice získáme stejnou matici uloženou ve formátu CCS (zřejmé z definice CRS/CCS), který naopak dobře pracuje po sloupcích. Dále budeme způsob procházení násobení matic $A \cdot B$ značit dle užitého formátu, tedy například $CRS \cdot CCS$.

Z definice maticového součtu jsou zřejmé tyto dva fakty:

- Mezi sebou násobíme všechny prvky $a_{i,k}$ s prvky $b_{k,j}$, tedy k-tý sloupec z matice A s k-tým řádkem z matice B
- Hodnotu prvku c_{ij} matice C získáme jako skalární součin i-tého řádku matice A s j-tým sloupcem matice B

Tyto dva pohledy nás vedou na dva logické způsoby násobení, což jsou $CCS \cdot CRS$ (dále také CR) a $CRS \cdot CCS$ (dále také RC). Výhodou prvního je, že nemusíme porovnávat hodnoty indexů a jediná prováděná operace je tedy samotné násobení. Tento způsob se tudíž zřejmě hodí pro sériovou verzi a vyžaduje nejméně provedených operací. Druhý způsob vyžaduje porovnávání indexů a je významně náročnější na výpočetní prostředky, na druhou stranu však umožňuje samostatný výpočet hodnot jednotlivých prvků matice. Tato vlastnost umožňuje $CRS \cdot CCS$ masivně paralelizovat, aniž bychom museli využít nákladnou synchronizaci zápisů do výsledné matice (která by pohltila téměř všechny výhody paralelizace), nebo zápis do více pomocných matic a následnou redukci.

2.3.2 BCRS

2.3.2.1 Datová struktura

Datová struktura BCRS je totožná s CRS. Rozdílem je, že jsou zde indexy blokových sloupců (nikoliv samotných prvků) a bude mít tedy pouze délku $nnzb$, což je počet nenulových bloků. Navíc je třeba uložit velikost bloku. Pro jednoduchost (a malé výhody opaku) jsem se rozhodl podporovat pouze čtvercové bloky, takže tato velikost může být uložena za pomocí jediné hodnoty bs.

2.3.2.2 Převod z husté matice

Algoritmus 3 Hustá matice → BCRS

```
 1: procedure CRS(M : DenseMatrix, bs : int)
 2:     C : BCRS                                              ▷ Tvořená matice
 3:     C.height ← M.height; C.width ← M.width; bs2 ← bs · bs
 4:     max_len ← C.width ∗ C.height/bs2
 5:     C.offsets ← zeroes(C.height)
 6:     C.indexes ← zeroes(max_len); C.values ← zeroes(max_len ∗ bs2)
 7:     count ← 0
 8:     for i ← 0 to M.height/blok do
 9:         C.offsets[i] ← count
10:         for j ← 0 to M.width do
11:             if notEmpty(M[i ∗ bs : (i + 1) ∗ bs][j ∗ bs : (j + 1) ∗ bs]) then
12:                 C.values[count ∗ bs2 : (count + 1) ∗ bs2] ←
13: M[i ∗ bs : (i + 1) ∗ bs][j ∗ bs : (j + 1) ∗ bs]
14:                 C.indexes[count + +] ← j
15:     C.offsets[M.height] ← nnz
16:     return C
```

Jak si můžeme všimnout na algoritmu 3, lze tento převod provést prakticky stejně jako u CRS (viz algoritmus 2). Rozdíly jsou ve velikosti alokovaných polí (po dokončení převodu mohou být pole zmenšena), a dále pak na řádcích 11, kde místo testování jednoho prvku testujeme celý blok a na řádku 12, kde kopírujeme celý blok hodnot.

2.3.2.3 Transpozice

I zde můžeme využít již hotovou transpozici z CRS. Celý postup může proběhnout stejně, místo prvků však pracujeme s celými bloky prvků. Samotné bloky musí být také transponovány způsobem obvyklým pro husté matice, což lze provést například při jejich přesunu v counting sortu.

2.3.2.4 Násobení

Vynásobením dvou bloků vždy vznikne nový blok hodnot (který může být přímo ukládán do husté matice, případně ukládán během výpočtu po hodnotách). I zde tedy lze násobit dvěma způsoby, tedy $BCCS \cdot BCRS$ (CR) a $BCRS \cdot BCCS$ (RC). Prvky uvnitř bloků však nemá velký smysl násobit jinak, než pomocí nezávislých skalárních součtů (řádek × sloupec) a postupného ukládání, protože bloky samy o sobě jsou husté, a tedy nevzniká dodatečná náročnost s porovnáváním indexů.

2.3.3 ELL

2.3.3.1 Datová struktura

Pro uložení matice ($n \times m$) v ELL formátu je nezbytné uchovat dvě matice velikosti $n \times ell\_width$, tedy matici hodnot a matici sloupcových indexů, kde ell_width je největší počet nenulových prvků na řádek. Přebytečné místo v řádcích s méně nenulovými prvky může být pro jednoduchost práce s maticí vyplněno nulami. Dále je samozřejmě nezbytné také uložit ell_width a šířku a výšku původní matice.

2.3.3.2 Převod z husté matice

Algoritmus 4 Hustá matice → ELL

```
1:  procedure CRS(M : DenseMatrix)
2:      C : ELL                                          ▷ Tvořená matice
3:      C.height ← M.height; C.width ← M.width
4:      C.ell_width ← max(count_nnz(M.rows))             ▷ O(n · m)
5:      C.indexes, C.values ← zeroes(C.ell_width ∗ C.height)
6:      for i ← 0 to M.height do
7:          row_index ← 0
8:          for j ← 0 to M.width do
9:              if notEmpty(M[i][j]) then
10:                 C.values[i ∗ C.ell_width + row_index] ← M[i][j]
11:                 C.indexes[i ∗ C.ell_width + row_index] ← j;
12:                 row_index + +;
13:     return C
```

Ačkoliv je struktura tohoto formátu značně rozdílná od CRS, i zde lze převod provést podobným způsobem. Pseudokód tohoto algoritmu si můžete prohlédnout na algoritmu 4.

2.3.3.3 Transpozice

I u tohoto formátu bude transpozice probíhat podobně jako u základního CRS. Rozdílem je nutnost explicitně vynechávat nulové prvky a navíc vypočítat novou *ell_width*. Kromě těchto dvou rozdílů je algoritmus obdobný.

2.3.3.4 Násobení

Algoritmus násobení je v zásadě shodný s CRS, výhodou tohoto formátu ovšem je, že všechny řádky nenulových elementů jsou stejně dlouhé (s výplňovými nulami), a proto může být zaručena lepší práce s pamětí a využití vláken při paralelizaci.

2.3.4 DIA

2.3.4.1 Datová struktura

Pro ukládání v tomto formátu je třeba pole hodnot (float) o velikosti $n \cdot$ *dia_count*, kde *dia_count* je počet nenulových diagonál a n je výška matice, tedy i diagonály doplněné nulami (na délku nejdelší možné). Dále je potřebné pole indexů diagonál (int), logicky dlouhé *dia_count* a kromě obvyklé výšky a šířky matice musí být uložen také již zmiňovaný počet nenulových diagonál.

2.3.4.2 Převod z husté matice

Převod do tohoto formátu je nejsložitější ze všech implementovaných formátů. Nejdříve je nezbytné zjistit počet nenulových diagonál a jejich indexy. Toto lze provést jednoduchým průchodem matice například se zaznamenáváním přítomnosti diagonál do pole flagů (booleovských hodnot) v $O(n \cdot m)$ a následným průchodem tohoto pole a vypsáním přítomných diagonál v $O(m)$. Následně je nutné tyto diagonály postupně (od nejnižšího indexu) znovu projít a do pole hodnot zapisovat veškeré hodnoty z diagonály (včetně nulových a výplňových hodnot pro prvky mimo matici). I tato operace tedy lze provést v $O(n \cdot m)$. Pro jeho přílišnou délku (převážně kvůli kontrole mezí a přepočtům indexů) zde nebude pseudokód uveden.

2.3.4.3 Transpozice

Tato operace je naopak nejjednodušší ze všech formátů. Transpozice totiž znamená pouhé vynásobení indexů diagonál -1 (a zapsání do pole indexů v opačném pořadí). Aby bylo zachováno zarovnání diagonál (a výplňových prvků), je navíc nutné posunout hodnoty diagonály o původní index doleva (pro záporné tedy o absolutní hodnotu doprava), čímž je dosaženo toho, že transponované diagonály znovu začínají na řádku 0.

2.3.4.4 Násobení

Zde je znovu použitelný základní algoritmus. Po řádcích je možné matici procházet s konstantním krokem (m). Tento způsob je vhodnější než procházení po sloupcích, které lze sice také provést s konstantním krokem, ale vyžaduje kontrolu mezí matice a především samotných diagonál (protože ne všechny diagonály obsahují všechny sloupce). S výplňovými prvky lze pro jednoduchost (a efektivitu) nakládat jako se všemi ostatními, protože kvůli jejich nulovosti výsledek nemění.

2.4 Základní struktura paralelních algoritmů

Jak si lze snadno povšimnout z předchozí sekce, úpravou sekvenčních algoritmů násobení můžeme získat paralelní verze a to hned dvěma způsoby dle způsobu procházení matic.

Algoritmus 5 Paralelní násobení $col \times row$

```
1:  procedure CUDA_CR(AT, B : Matrix, C : DenseMatrix, α : float)
2:      for all k ← 0 to B.height − 1 do in parallel
3:          k ← fork(B.height)
4:          for e1 : Element in A[k] do
5:              for e2 : Element in B[k] do
6:                  x ← e2.column_index
7:                  y ← e1.column_index
8:                  value ← α · e1.value · e2.value
9:                  atomic C[y][x] ← C[y][x] + value;
10:     return C
```

Pokud budeme procházet matice způsobem $col \times row$ (tedy například u CRS matic $CCS \times CRS$, dále jen CR), dostaneme v základu postup, který si lze prohlédnout v algoritmu 5. Procházíme zde veškeré prvky, které mají být spolu vynásobeny, a výsledek tohoto násobení zapisujeme do výsledkové matice. Zřejmou výhodou je tedy jednoduchost a i menší náročnost procházení, nevýhodou je pak nutnost atomické operace, protože výsledky jsou zapisovány nepravidelně a mohlo by docházet k vzájemnému přepisování.

Pokud budeme procházet matice způsobem $row \times col$ (tedy u CRS matic $CRS \times CCS$, dále RC), získáme složitější algoritmus, který si můžete prohlédnout v algoritmu 6. V tomto algoritmu přiřadíme každému vláknu jednu pozici ve výsledkové matici a následně provedeme skalární součin příslušného řádku z A a sloupce z B (tedy řádku z BT) se zapsáním do výsledkové matice až po dokončení součinu. Tento způsob procházení má zřejmou výhodu odstranění atomické operace, protože přístup k výsledkové matici je řízený a na jednu pozici přistupuje pouze jedno vlákno. Navíc se významně sníží počet zápisů. Na druhou stranu je ovšem nutné provádět velké množství kon-

Algoritmus 6 Paralelní násobení $row \times col$

```
1:  procedure CUDA_RC(A, BT : Matrix, C : DenseMatrix, α : float)
2:      for all y ← 0 to A.height − 1 do in parallel
3:          for all x ← 0 to BT.height − 1 do in parallel
4:              e1 ← A[y].firstElement()
5:              e2 ← B[x].firstElement()
6:              sum ← 0
7:              while e1.notEnd() ∧ e2.notEnd() do
8:                  if e1.column_index < e2.column_index then
9:                      e1 ← e1.next()
10:                 else if e1.column_index > e2.column_index then
11:                     e2 ← e2.next()
12:                 else      ▷ Sl. indexy se rovnají, prvky mají být vynásobeny
13:                     sum ← α · e1.value · e2.value
14:                     e1 ← e1.next()
15:                     e2 ← e2.next()
16:             C[y][x] ← C[y][x] + sum;
17:             return C
```

trol sloupcových indexů, což zvyšuje složitost a u výpočtů na GPU může vést k divergenci vláken ve warpu.

Velkou výhodou druhého algoritmu je mnohem lepší použitelnost na značně velké matice. Pokud totiž velikost výsledkové matice přesahuje množství paměti GPU (a násobené matice jsou dostatečně řídké, aby se vešly do globální paměti GPU), je možné rozdělit výsledkovou matici na části a následně provádět výpočet pouze nad jednotlivými bloky, aniž by došlo k výraznému navýšení časové náročnosti.

Oba tyto algoritmy je samozřejmě nutné specializovat pro jednotlivé formáty ukládání matic. Specificky je nutné zakomponovat různé způsoby přístupu k prvkům a jejich sloupcovým indexům a různé způsoby procházení řádků matice. Pro formát BCRS je navíc nezbytné rozšířit i operaci násobení na násobení mezi bloky (které může být provedeno základním algoritmem pro násobení hustých matic).

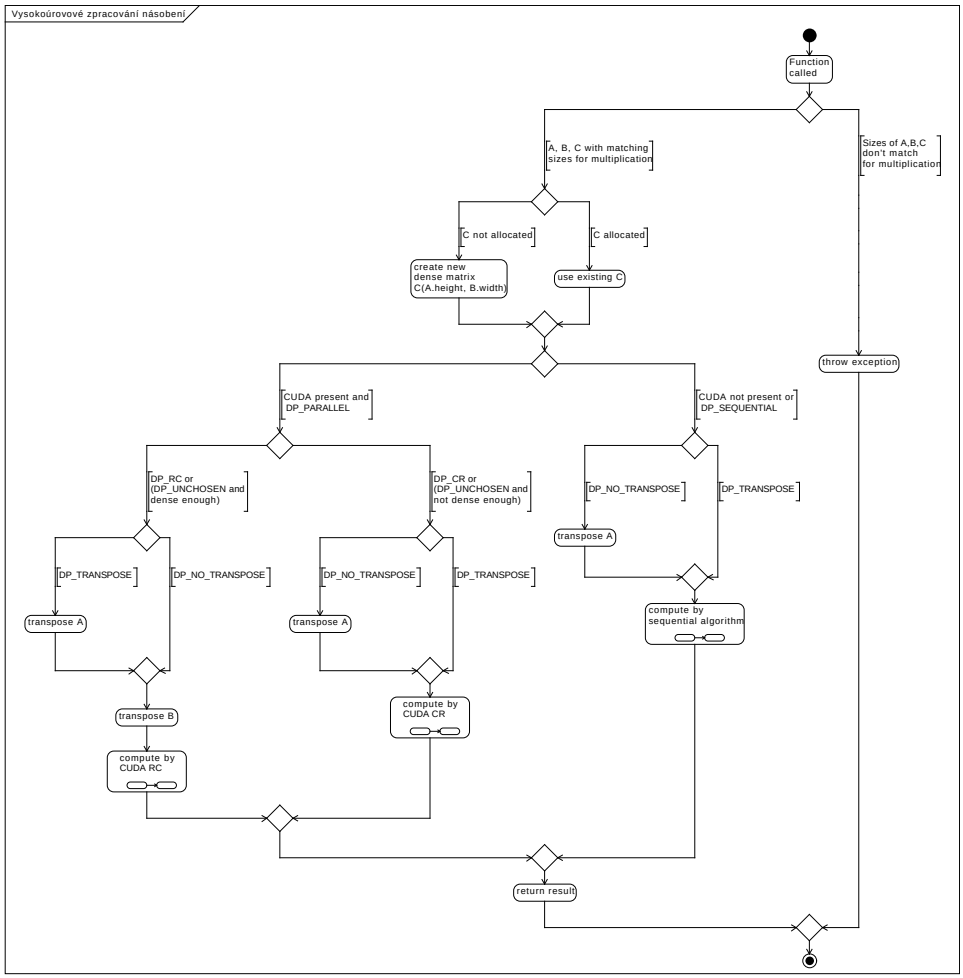
Dále je samozřejmě možné algoritmy dále optimalizovat použitím různých nízkoúrovňových optimalizačních technik a zvláště pak u druhého algoritmu můžeme narazit na limitaci CUDA (například počet vláken v bloku), viz dále.

2.5 Přechod mezi high level a low level funkcemi

Jak bylo zmíněno v sekci sekvenčních algoritmů, bude implementováno pouze násobení matic ve stejném formátu uložení. Navíc budou implementovány 3 různé způsoby výpočtu (2 paralelní a jeden sekvenční), které na vstupu

Obrázek 2.1: Vysokoúrovňová funkce násobení matic

.



vyžadují různé kombinace transponování matic A a B. Aby bylo usnadněno používání a také umožněno v budoucnu snadno měnit implementaci nízkoúrovňových funkcí nezávisle na interface, rozhodl jsem se implementovat „rozcestníkovou“ funkci, která bude obalovat nízkoúrovňové funkce, a mechanismus, zajišťující převod na stejný formát uložení.

„Rozcestníková“ funkce bude na vstupu přijímat 2 libovolné matice A a B, hodnotu α a výstupní hustou matici C, dále pak informace zda má být provedena transpozice matice A před násobením (DP_TRANSPOSE, nebo DP_NO_TRANSPOSE), zda má výpočet být proveden sekvenčně, či paralelně (DP_SEQUENTIAL, DP_PARALLEL) a preferovanou metodu procházení matice. Pro usnadnění používání budou mít poslední tři výběry zadanou defaultní hodnotu (bez transpozice, paralelně, nevybráno) a proto je bude

možné vynechat (DP_RC, DP_CR, DP_UNCHOSEN). Navíc místo matice C bude možné zadat nulový pointer, což znamená, že se má alokovat nová nulová matice odpovídajících rozměrů. Výsledek bude uložen v matici C (nebo nové v případě nulového pointeru). Tato funkce navíc zajistí kontrolu chybných velikostí matic. Celou funkci si můžete prohlédnout na diagramu 2.1.

V diagramu si lze všimnout, že při nevybraném způsobu procházení matice (RC,CR) je způsob vybrán automaticky dle vlastností matice. Tento automatický výběr je založen na předpokladu, že verze $row \times col$ je efektivnější pro hustší matice a naopak $col \times row$ je efektivnější pro matice řidší. Tento předpoklad bude ověřen a odhadnutí vhodné hranice bude experimentálně provedeno v rámci testování a získané hodnoty zpětně použity pro dokončení mechanismu.

Mechanismus převodu na stejný formát využívá operací převodu na hustou matici a z husté matice. V případě různých formátů je druhá matice (matice B) nejdříve převedena na hustou matici a následně z ní zpět na stejný formát jako matice A. Kvůli limitacím dědičnosti u jazyka C++ je nutné, aby byla tato operace prováděna na začátku každé z nízkoúrovňových funkcí a ne přímo u sdílené „rozcestníkové“ funkce. V opačném případě bychom buď museli implementovat tuto funkci pro veškeré formáty matic (se zřejmým porušením DRY) a nebo by musela funkce obsahovat switch s různými formáty, což taktéž závažně porušuje pravidla správného objektového návrhu.

2.6 Návrh tříd

Po navrhnutí většiny dílčích částí je konečně možné pustit se do samotného návrhu třídní struktury.

Základním stavebním prvkem bude abstraktní třída pro ukládání matice (Matrix). Tato třída bude taktéž obsahovat deklarace všech společných metod. Třída může obsahovat výšku a šířku matice (protože matice v jakémkoliv formátu má tyto parametry), abstraktní deklarace nízkoúrovňových metod pro násobení a transpozici, implementovanou vysokoúrovňovou metodu pro násobení a případné další wrappery této metody sloužící ke zjednodušení používání. Dále bude obsahovat základ mechanismu pro převod matic na stejný formát.

Jednotlivé třídy pro ukládání ve specifických formátech budou dědit z třídy Matrix a dále budou obsahovat konstruktory z husté matice (třída pro hustou matici bude přijímat pole dat a velikost matice), destruktory, konkrétní implementace násobení, transpozice a převodu na hustou matici. Dále budou obsahovat také konkrétní mechanismus pro převod na stejný formát. Samozřejmě budou také obsahovat data uložená v daném formátu.

Neabstraktní třídy dědící od třídy Matrix budou: DenseMatrix, CRSMatrix, BCRSMatrix, DIAMatrix, ELLMatrix. Formáty dat v těchto třídách jsou zřejmé z názvu a byly již rozebrány výše.

Obrázek 2.2: Základní návrh tříd pro matice

Kapitola 3

Implementace

V této části práce je podrobněji popsána implementace základní funkcionality (sekvenční část) a prvky, které byly přidány navíc oproti předchozímu návrhu. Dále jsou v této sekci také řešena specifika výpočtů na GPU (zvláště pak samotné technologie CUDA) a optimalizace nutné pro zajištění dobré efektivity implementací. Implementované kódy si lze prohlédnout na přiloženém CD.

3.1 Implementace základní funkcionality

Implementace základní funkcionality knihovny probíhala dle návrhu tříd a za použití rozebraných algoritmů pro práci s řídkými maticemi. Vzhledem k zadání práce byla implementace prováděna v jazyce C++ a kvůli přenositelnosti byl zvolen standard C++11, který je v současnosti podporován většinou kompilátorů.

Při implementaci bylo dbáno na to, aby bylo knihovnu možno přeložit (a spustit) i na zařízeních nekompatibilních s CUDA technologií. Toto bylo zajištěno implementací sekvenčních verzí všech algoritmů, oddělením CUDA kódu do samostatných souborů a pomocí direktiv preprocesoru.

Aby se předešlo velkému nárůstu rozsahu práce (mimo obor zadání) a komplexity kódu, nebyly implementovány funkce, které nebyly nutné pro implementaci operace násobení matic, případně testování správnosti a efektivnosti této operace. Mezi tyto úmyslně vynechané funkce patří například přidávání (a měnění) hodnot načtených matic. Návrh a implementace však byly prováděny takovým způsobem, aby bylo v budoucnu snadné funkcionalitu v případě potřeby rozšířit.

3.2 Uživatelský interface

Ačkoliv byl program navržen ve stylu knihovny (bez přímého uživatelského rozhraní), pro testování bylo rozhraní nutné vytvořit. Pro potřeby této práce

byl tedy navržen jednoduchý interface, který umožňuje ze vstupních souborů načíst dvě matice, uložit je ve vybraném formátu, provést jejich vynásobení a zjistit časovou náročnost poslední operace. Tento interface nevyužívá všech možností implementace a jeho jediným účelem je sloužit k testování knihovny a ověření její funkčnosti (správné kompilace atd.).

3.3 Implementace CUDA

Implementace paralelních verzí algoritmů v jazyce CUDA s sebou nesla různé problémy a nutná rozhodnutí.

Nejdříve se bylo nutné rozhodnout, pro jakou verzi technologie CUDA a jaké minimální specifikace hardwaru (CUDA capability) bude vyvíjeno. Byla zvolena verze jazyka (a runtime environmentu) 9.1, protože se v době počátku práce na implementaci jednalo o nejnovější verzi, pro niž bylo možné sehnat kompilátor pro systém Linux. Jako rozumný kompromis mezi schopnostmi a podporou staršího hardwaru (a s ohledem na dostupná testovací zařízení) byla pak zvolena CUDA capability 3.5 (Kepler). Veškerá další tvrzení a implementační rozhodnutí budou vztahována k této konfiguraci.

Druhým problémem byla enkapsulace dat. CUDA kernel musí být mimo třídu, zároveň však potřebuje přímý přístup k datům třídy. Pro vyřešení tohoto problému bylo nejdříve spuštění kernelu „obaleno“ metodou třídy. Pro uspořádání práce s daty byla navíc v každé třídě definována struktura, která slouží pro načítání dat z třídy na GPU, jejich uchovávání, případné zpětné uložení a následné uvolnění prostředků. Přímo k datům třídy je tedy přistupováno pouze konstruktorem struktury (definované uvnitř třídy).

Další významné rozhodnutí souviselo s omezením technologie CUDA. Výpočetní kernel je možné spustit paralelně na maximálně $2^{31}-1$ blocích, z nichž každý může obsahovat maximálně 1024 vláken. První hodnota zřejmě není významným omezením (protože převyšuje obvyklé množství RAM u většiny GPU), na druhou hodnotu je nutné brát zřetel.

U verze $col \times row$ můžeme například přidělit každému vláknu jednu hodnotu k (tedy sloupec z A a odpovídající řádek z B). Vlákna pak můžeme jednoduše rozdělit například po 1024 na blok, případně po menším počtu (dělitelném 32) pro lepší výkon u menších matic. Tato varianta byla zvolena jako základní způsob rozdělení. Druhou možností je pak přidělit jedno k celému bloku vláken. Tato varianta může mít vyšší overhead a pro malé (případně velmi řídké) matice může být množství práce pro jednotlivá vlákna příliš malé. Na druhou stranu umožňuje využití sdílené paměti a lepší práci s globální pamětí.

U verze $row \times col$ je situace složitější. Přirozeným rozdělením je přidělit každému vláknu jednu výstupní pozici v matici C, a tedy jeden řádek z matice A a jeden sloupec z matice B, na výpočet skalárního součinu, kdy blok určuje řádek a pozice vlákna v bloku určuje slupec. Toto umožňuje snadné adre-

sování i využití sdílené paměti pro uchování celého řádku. Zde je však problém s maximálním počtem vláken na blok. Toto lze obejít rozdělením řádku do více bloků, ale jako efektivnější varianta bylo nakonec zvoleno přidělení více pozic každému vláknu, s cyklením do vyčerpání velikosti řádku. Například vlákno s indexem 4 v bloku 100 bude u matice velikosti 2048×2048 a velikostí bloku 1024 počítat všechny výstupní pozice, pro něž platí $y = 100 \ \wedge \ x \equiv 4 \mod 1024$, tedy prvky řádku 100 ve sloupcích 4 a 1028.

3.4 Využití prostředků CUDA

Pro napsání efektivní implementace algoritmů (dále jen algoritmů) je důležité využít co nejvíce možností GPU. Jednou z hlavních je schopnost GPU částečně skrývat latence operací (převážně paměťových) využitím současného provádění více vláken, než je skutečný počet výpočetních jednotek.

Nejvíce může na multiprocesoru teoreticky být 2048 vláken a efektivita algoritmu je logicky nejvyšší, pokud je tohoto čísla prakticky dosaženo. K tomu je třeba nepřesáhnout žádný z dalších limitů, kterými jsou počet bloků (bloky nesmí být příliš malé), množství sdílené paměti na blok a počet využitých registrů (obvykle hlídán kompilátorem). Rozhodne-li se tedy hodnota jednoho z těchto parametrů, jsou maxima všech ostatních hodnot přesně určena. Pro dosažení optimální efektivity byly tedy hodnoty (počet vláken na blok a velikosti přidělené sdílené paměti) určovány dle těchto limitů.

3.5 Optimalizace

V této sekci budou popsány techniky, které se ukázaly být účinné pro zvýšení efektivity paralelizovaných algoritmů. Zkoušené techniky, které nevykazovaly významné zvýšení efektivity (například loop unrolling), budou vynechány. Detailnější popis testování efektivity je v kapitole Testování.

Kromě celkové efektivity algoritmu byl pro návrh optimalizací používán taktéž profiler (Nvidia Visual Profiler), pomocí kterého bylo možné snadněji se orientovat v problémech jednotlivých implementací a navrhovat směr dalšího vylepšování.

Optimalizace byly nejdříve prováděny na algoritmech CRS formátu a následně přeneseny na ostatní.

3.5.1 Využití sdílené paměti u RC verze

Hlavním důvodem snižení efektivity u kernelů jsou paměťové operace. Tento fakt, který platí obecně, byl potvrzen i v rámci profilování. Paměť s nejpomalejším přístupem (a nejnižším throughputem) je globální paměť zařízení. Pro zvýšení efektivity násobení je tedy nutné tuto paměť využívat co nejméně a co nejefektivněji.

Pro zvolenou verzi CUDA capability je k dispozici 48 KB sdílené paměti na multiprocesor. Ve srovnání s globální pamětí umožňuje tato paměť vyšší průtok dat, má nižší odezvu a umožňuje obsluhovat více vláken zároveň. Tato paměť je sdílená v rámci jednoho bloku.

U verze RC (*row* × *column*) pracuje celý blok s jedním řádkem matice AT, proto je intuitivní nahrát tento řádek do sdílené paměti. Všechna vlákna v bloku nejdříve společně načtou data (indexování dle formátu a hodnoty nenulových prvků) z globální paměti do sdílené (kvůli zamezení race conditions je nutná synchronizace na úrovni bloku před a po načtení) a poté se provádí výpočet již s těmito přednačtenými daty.

Kvůli omezené velikosti paměti je nutné v případě dlouhých řádků nutné řádek rozdělit na úseky, které se do paměti vydají, a data tak načítat opakovaně. Toto však kvůli využití řídkých formátů nelze provést pomocí indexu sloupce, ale pouze pořadím nenulového prvku v řádku. To vede k problému s indexováním v matici B. Po dokončení úseku máme nalezený index prvku v BT, který je větší nebo roven poslednímu prvku z tohoto úseku (viz algoritmus 6) (jedno vlákno musí zpracovávat více sloupců). S tímto indexem lze nakládat třemi způsoby. Můžeme jej zahodit a v příštím úseku (u tohoto sloupce) znovu začít hledat od začátku. Tento způsob ale vede k častému procházení stejných částí, je velmi neefektivní, a proto nevhodný. Druhou možností je tento index uložit (do globální paměti) a po načtení nových dat jej znovu načíst. Třetí způsob je načítání nového úseku nikoliv poté, co jsou projity všechny sloupce, ale poté, co všechna vlákna v bloku dokončí práci s ním pro jeden sloupec (tedy přesunutí načítání o úroveň níže). Nevýhodou druhého způsobu je zvýšená paměťová náročnost a využití globální paměti. Nevýhodou třetího pak nutnost vícenásobného načítání stejného úseku u velkých matic (kde velké množství řádků je děleno na mnoho úseků). Při testování a profilování však nebyl zjištěn znatelný rozdíl mezi těmito variantami, a proto byla pro jednoduchost vybrána varianta třetí.

Kvůli mnohem většímu množství operací porovnávání indexů (vůči operacím násobení) je vhodné načítat do sdílené paměti pouze indexy prvků, zatímco samotné hodnoty mohou být načteny až při výpočtu (čímž je výrazně zvýšena délka úseků, bez významné ztráty efektivity výpočtu v rámci úseku).

Velikost úseku načítaného do sdílené paměti lze snadno vypočítat za běhu programu. K tomu lze využít znalost velikosti paměti pro multiprocesor, maximální množství vláken uložených na multiprocesoru (2048), které se snažíme zachovat kvůli efektivitě, a znalost velikosti dat jednoho prvku pro jednotlivé formáty ukládání. Není-li sdílená paměť využívána i pro další účely, pak například pro CRS se jedná o $\frac{48\ KB}{2048 \cdot 4}$, tedy 6 prvků na vlákno v bloku a pro blok o velikosti 1024 (maximální) je velikost jednoho úseku 6144 prvků.

3.5.2 Zarovnání přístupů k paměti

Při načítání z globální paměti (případně její cache) se nenačítá pouze 1 prvek, ale celý blok 128 B (případně 32 z L2 cache), tedy 32 prvků pro int (float). Pokud více po sobě jdoucích vláken chce přistupovat k těmto datům, řídící jednotka je schopná celé toto načtení provést v jedné operaci. Pokud je načítání posunuté (například nulté vlákno načítá první prvek), musí být provedena dvě načtení. Aby byla zvýšena efektivita, je proto vhodné vynutit zarovnání načítání indexů na 32 prvků (i za cenu „prázdných“ cyklů a kontrolování mezních podmínek pod hranicí začátku načítání).

3.5.3 Využití registrů u RC verze

Efektivita kernelů je dále snižována dlouhou latencí nahrávání indexů sloupců matice BT z globální paměti. Pro snížení této latence byla využita technika přednahrávání dat těchto inexů do registrů. Tím je nejen umožněno zařízení lépe maskovat latenci globální paměti za běhu programu, ale také je kompilátoru objasněna nezávislost těchto dat na zbytku instrukcí v cyklu a je mu tak umožněno provádět radikálnější úpravy kódu při optimalizaci (loop unrolling, přehazování pořadí instrukcí aj.).

Testováním bylo zjištěno, že nejvyšší přínos přinese ukládání pouze jedné hodnoty dopředu a ukládání více než tří již (kvůli zhoršené práci s registry v jiných částech z důvodu jejich omezeného počtu) naopak efektivitu snižuje (dvě hodnoty dopředu je mírně méně efektivní než jedna).

3.5.4 Operace nad daty více vláken

Hlavní operací blokující vyšší efektivitu se po předchozích optimalizacích stalo načítání indexů prvků matice B. Ačkoliv lepší využití registrů snížilo dopad této operace, stále se jednalo o bottleneck. Důvodem tohoto problému je špatná práce s globální pamětí. Z načtených 128 B je totiž v cyklu využito pouze 4 B (jeden prvek) a zbytek zahozen. K využití větší části načtených dat je nezbytné, aby více vláken ve warpu využívalo indexová data jednoho sloupce. Abychom tohoto docílili, je však nutné přeuspořádat přidělení sloupců vláknům.

Místo toho, aby každý sloupec (a výsledný prvek) byl počítán jedním vláknem, musí na tomto výpočtu spolupracovat vláken více. Vzhledem k sekvenčnímu charakteru vyhledávání stejných indexů je však nemožné, aby toto bylo prováděno samostatně. Můžeme však využít operace nad daty vláken ve warpu, konkrétně ballot (operace, která určí, pro která vlákna ve warpu platí podmínka), all (operace zjistí, zda podmínka platí pro všechna vlákna) a shift (posune data vybrané proměnné ve warpu nižším, nebo vyšším vláknům warpu). U všech těchto operací lze maskou specifikovat, která vlákna warpu se mají operace účastnit. Například pro 4 vlákna na sloupec tak mohou být naráz načteny 4 indexy, následně je pomocí operace all zjištěno, zda není

index v A menší (případně posunuto o jedna a opakováno), pomocí operace ballot je zjištěno, o kolik pozic je nutné posunout index v B, než bude větší nebo roven indexu v A (0-4), a pomocí operace shift je posun proveden. Tímto posunem o více prvků částečně kompenzujeme neefektivitu více vláken provádějících stejné operace. Samotné násobení (a ukládání) pak může být provedeno například pouze jedním z vláken (nejoptimálnější způsob závislý na formátu ukládání).

Takto upravené načítání již může být rozšířeno o načítání indexů B do sdílené paměti, aby bylo zachováno zarovnání a nejlépe využita data v rámci jedné operace.

Přestože tento způsob vede k tomu, že část vláken neprovádí výpočty a slouží pouze k načítání prvků (kvůli tomu, že bottleneck efektivity není provádění výpočtů, ale práce s globální pamětí), dojde ve výsledku ke zlepšení efektivity.

Jako nejoptimálnější se ukázalo využití pouze 2 vláken na sloupec(s cache 4 prvků). Toto je logické vzhledem k exponenciálně klesající pravděpodobnosti posunů o více než 2 prvky v matici B. Za předpokladu posunu alespoň o jeden prvek má posun o 2 prvky téměř 50% pravděpodobnost, zatímco o 4 pouze 12.5% (téměř, protože rovnost indexů je u náhodných dostatečně řídkých matic oproti nerovnosti řádově menší). Data ostatních vláken jsou tedy prakticky nevyužita.

3.5.5 Optimalizace CR verze

Stejně jako u RC verze i zde byl bottleneck v efektivitě využití globální paměti. K zlepšení její efektivity bylo tedy nezbytné využít paměti sdílené. Aby toto mohlo být provedeno, musela být nejdříve změněna distribuce výpočtu mezi vlákna a bloky.

Místo přidělení každého řádku (hodnoty k) jednomu vláknu bylo zvoleno přidělení jednoho řádku celému bloku. Pro větší a řidší matice by bylo možné přidělit více řádků každému bloku, to ale nebylo vzhledem k limitaci velikosti výstupní matice potřebné.

Celý blok tedy sdílí sloupec matice A a řádek matice B. Pro vyřešení limitace velikosti sdílené paměti (počet hodnot v řádku může být větší než velikost sdílené paměti) byla využita technika blokování, často používaná při násobení hustých matic. Hodnoty (a indexy) řádku a slupce jsou rozděleny do bloků stejné velikosti a každý blok ze sloupce A je vynásoben každým blokem z řádku B (prvky uvnitř bloků také tímto způsobem každý s každým). Tento způsob umožňuje postupně nahrávat do sdílené paměti vždy pouze dva bloky a počet přístupů do globální poměti je významně snížen z původních $O(n^2)$ operací na řádek (konkrétní k).

Množství operací načítání z globální paměti po aplikaci této metody opti-

malizace pro stejně velké bloky stoupá dle funkce

$$O\left(\min(x,n)\cdot\left\lceil\frac{n}{x}\right\rceil+\min(x,m)\cdot\left\lceil\frac{n}{x}\right\rceil\cdot\left\lceil\frac{m}{x}\right\rceil\right),$$

kde x je počet prvků načítaných do sdílené paměti (velikost bloku) a n, m jsou délky řádků (sloupců). Pokud pro zjednodušení předpokládáme stejný počet prvků ve sloupci matice A jako řádku matice B (tedy $n = m$), dostaneme vzorec

$$O\left(\min(x,n)\cdot\left\lceil\frac{n}{x}\right\rceil\cdot\left(1+\left\lceil\frac{n}{x}\right\rceil\right)\right).$$

Tato funkce je zřejmě lineární pro $n \leq x$, tedy pokud množství prvků v řádku nepřesahuje kapacitu sdílené paměti. V případě, že toto množství je přesaženo (do sdílené paměti je třeba načítat více bloků v řádku), roste tato funkce sice kvadraticky, ale s mnohem nižším koeficientem (vyděleno x), což je na menších maticích téměř nerozpoznatelné od lineární funkce.

Další výhodou tohoto postupu je, že nedochází ke kolizi zápisů do globální paměti v rámci jednoho výpočetního bloku, protože index výsledku je určen řádkovým indexem prvku z A a sloupcovým indexem prvku z B (a tato dvojice je pro dané k jedinečná). Může tedy docházet pouze ke kolizím mezi bloky, čímž se snižuje časová náročnost atomického přičtení do výsledkové matice.

Práce je mezi vlákna v bloku rozdělena pomocí 2D souřadnic vláken (pro 1024 vláken 32×32), určujících přidělené elementy z jednotlivých bloků.

Pro načítání do sdílené paměti byly využity techniky z optimalizací RC verze algoritmu (zarovnání, využití celého 128B čtení).

Během optimalizace byly také vyzkoušeny další techniky (změna velikosti bloků, přidělení více práce jednotlivým blokům aj.), tyto techniky však neměly na efektivitu (bez ohledu na hustotu a velikost matice) pozorovatelný kladný vliv a proto nebudou dále rozebírány.

3.5.6 Přenesení optimalizací na ostatní formáty řídkých matic

V první fázi byly veškeré optimalizace prováděny na CRS formátu (kvůli jednoduchosti debugování). Všechny tyto optimalizace jsou však poměrně obecné (a algoritmy pro většinu formátů navrženy v zásadě stejně), takže je bylo možné využít i u zbývajících formátů.

Přenesení optimalizací na BCRS bylo nejjednodušší. Kromě úpravy velikostí dat prvků (z jednoho floatu na celý blok) a změny způsobu násobení mohla být celá funkce prakticky zkopírována. Velkou výhodu má pak tento formát u RC verze. Za využití techniky s prací nad daty více vláken warpu (poslední hlavní optimalizace RC verze zmíněná výše) jsou u CRS ostatní vlákna při násobení nevyužita. U BCRS je však možné přidělit každému vláknu část násobeného bloku, a tak neztrácet žádný výkon. Navíc se tímto přístupem k datům bloku lépe využije operací načítání dat bloků z globální

paměti, protože při správném rozložení vláken přistupují sousední vlákna vždy k sousedním prvkům (omezení globální paměti viz výše). Díky využití těchto původně neaktivních vláken je taktéž možné pro lepší výkon zvýšit množství spolupracujících vláken ze dvou na čtyři.

I přenesení optimalizací na ELL formát je jednoduché. Jediný problém, který je nutný vyřešit, je práce s výplňovými nulami (v indexech sloupců), které narušují předpoklady provedené u operací s warpy (monotonii – konkrétně neklesání – sloupcových indexů v řádku), proto je nutné je explicitně nahrazovat nejvyššími možnými přirozenými čísly. Tento formát navíc z optimalizací těží mnohem méně než CRS, protože mnohé optimalizace byly navrženy takovým způsobem, že odstraňují nevýhody CRS formátu vůči ELL (například zarovnání prvků).

Ačkoliv i poslední formát (DIA) může z některých optimalizačních technik těžit, jejich využití je zde značně omezené a problematické. Pro ideální využití výpočetních prostředků by zde tedy bylo vhodné navrhnout zcela specializovaný způsob optimalizací.

Kapitola 4

Testování

Během celého vývoje probíhalo samozřejmě testování vzniklých kódů. V prvních fázích se jednalo zvláště o unit testy a další testování věcné správnosti implementací (tedy zda program skutečně provádí to, co bylo specifikováno). V pozdějších fázích se dále přidalo testování aplikace jako celku (z velké části manuální), testování efektivity implementací a optimalizací.

Při testování efektivity byl měřen čas pouze samotného výpočtu násobení matic (bez přesunů dat mezi GPU-CPU a pomocných operací na CPU).

4.1 Testovací hardware

Testování probíhalo na přístroji s těmito technickými specifikacemi:

- Procesor: Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz
- Výpočetní paměť: SODIMM DDR3 Synchronous 1600 MHz, 8GiB
- GPU: GK107M [GeForce GT 650M]
  - CUDA jader: 384
  - výpočetní frekvence: až 735-900 (boost) MHz
  - CUDA capability: 3.5 (Kepler)
  - Globální paměť: 1024 MB
  - Maximální výpočetní výkon (float): 729.6 GFLOPS

4.2 Nastavení kompilace

Pro kompilaci byl využíván Nvidia CUDA Compiler (NVCC), což je proprietární překladač přímo od společnosti Nvidia. Minimální předpokládaná CUDA capability ponechána na verzi 3.0, protože žádné z rozdílů mezi těmito architekturami nebyly v implementaci využity a efektivita kompilovaných

verzí je tedy stejná jako u architektury 3.5 (což bylo i experimentálně ověřeno). Verze 3.5 je pouze funkcionálním rozšířením verze 3.0 (unified memory, dynamický paralelismus), a proto je možné program kompilovaný pro 3.0 spustit i na zařízení s CUDA capability 3.5. Úroveň optimalizací ponechána na nejvyšší -O3 pro kód běžící na GPU a -O2 pro sériový C++ kód.

Při debugování byly dále využity přepínače -g a -lineinfo (které dodají debugovací informace).

4.3 Unit testy

Pro tento druh testování (definice unit testů a více o nich například v [20]) jsem zvolil knihovnu CppUnit (verze 1.14) a to z důvodu její rozšířenosti a zvláště pak z důvodu její jednoduché integrace s použitým vývojovým prostředím (Netbeans).

Instalaci jsem provedl podle [21], protože přímá instalace z repozitáře pro Ubuntu 18.04 (k 19.2.2019) není v kombinaci s Netbeans funkční. Pro následné spuštění testů je navíc, jak jsem zjistil z [22], nezbytné přidat direktivu „$-lcppunit$“.

Osvědčilo se také starší unit testy uchovávat a tento soubor testů používat jako smoke testy.

4.4 Věcná správnost implementace

Pro dostatečně malé celočíselné matice lze správnost kontrolovat pomocí jednoduchého porovnání výsledku s výsledkem očekávaným. Pro velké matice s reálnými čísly je však tento postup neúčinný, protože operace s čísly s omezenou přesností nejsou komutativní, ale při paralelních výpočtech není pořadí operací přesně určeno (zvláště u CR verze). Prováděním operací v rozdílném pořadí tedy vzniknou rozdílné výsledky. Proto bylo pro zjištění správnosti nutno využít jiných postupů.

Prvním způsobem ověření správnosti je kontrola počtu provedených operací násobení, protože tento počet je stejný u sekvenčních verzí i u všech verzí paralelních (všechny jsou založené na stejném principu).

Druhý postupem pak bylo využití pro tento účel navržené operace porovnávání rozdílnosti matic, která vypočítává průměrnou relativní odchylku nenulových prvků obou matic (odchylka dvou prvků počítána jako $\frac{|a-b|}{|a|+|b|}$), a omezení této odchylky vybranou hodnotou. Při měření byla maximální naměřená průměrná odchylka řádově 10^{-6}, což je vzhledem k přesnosti float zanedbatelnná hodnota.

4.5 Testovací data

Základní testovací data (pro zjištění správnosti implementace) byla vytvořena ručně. Pro získání větších matic na testování časové náročnosti, efektivity algoritmů a optimalizačních technik již bylo nutné použít jiného přístupu. V první fázi jsem využíval matice získané z [23]. Pro získání matic se specifickými parametry a tvorbu grafů s těmito maticemi byl však výběr matic příliš hrubou technikou a generátory, které se mi podařilo nalézt (například matgen) byly obvykle značně nedostačující funkcionalitou. Rozhodl jsem se proto vytvořit si vlastní generátor náhodných řídkých matic (se specifikací parametrů).

Vytvořený generátor umožňuje generovat matice libovolné velikosti se zadanou hustotou nenulových hodnot. Navíc kromě základních náhodných matic umožňuje i generovat matice s hodnotami shlukovanými do bloků, či omezeného počtu diagonál.

Pro testování efektivity (času) byly vždy využity 2 různé matice (stejné velikosti). Pro snadnost zobrazování byly voleny čtvercové matice.

4.6 Vstupní formát matic

Vzhledem k původnímu zdroji matic a také k rozšířenosti tohoto formátu jsem se rozhodl využívat pro vstupní (vnější) formát matic formát MM (Matrix Market), přesněji jeho zjednodušenou verzi (tedy pouze umožňující zápis řídkých matic s reálnými čísly). Jedná se o jednoduchý formát založený na COO. První řádek po hlavičce (která není pro naše účely významná) obsahuje tři údaje: n m nnz, kde n je výška matice, m je šířka matice a nnz je počet nenulových hodnot. Zbylých nnz řádků obsahuje vždy y souřadnici, x souřadnici a hodnotu tohoto prvku. Vše je odděleno mezerami.

Dále tento formát umožňuje řádkové komentáře začínající znakem % (a pro naše účely zjednodušeného formátu je i hlavička pouhým komentářem).

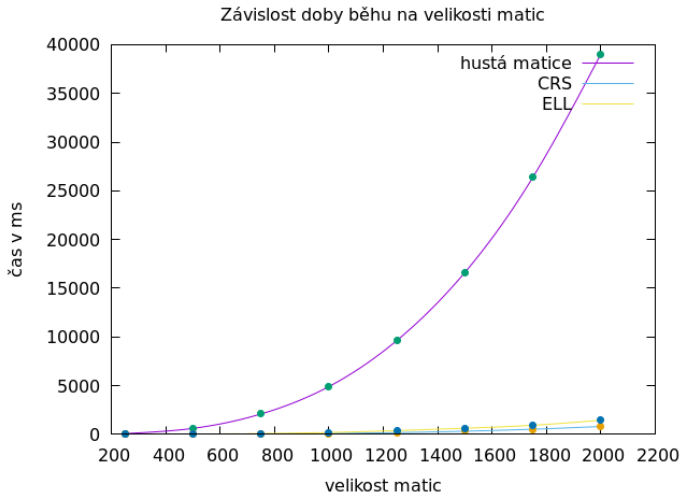
Veškeré vygenerované testovací matice jsou kvůli opakovatelnosti nejdříve uloženy v tomto formátu a následně teprve nahráty do aplikace a použity.

4.7 Sekvenční verze

První otázkou, na kterou chceme znát odpověď, je, zda (a za jakých podmínek) se využití řídkých matic časově vyplatí vůči standardnímu algoritmu násobení hustých matic. K tomuto ověření byly provedeny dvě série měření závislostí času a to na hustotě a velikosti matic.

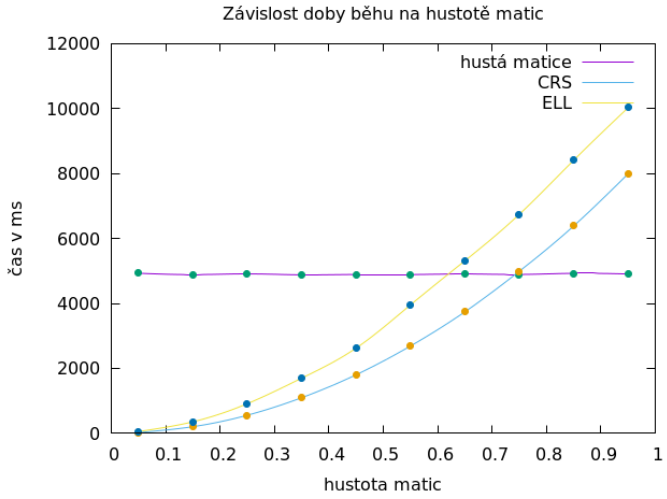
Na grafu 4.1 je vidět, že za konstantní hustoty matic je doba běhu algoritmu (dle očekávání) u všech formátů kubicky závislá na velikosti matic. Řídké formáty jsou však (pro tuto hustotu) významně rychlejší. Konkrétně CRS 50x, a ELL 25x. Důvodem tohoto zrychlení je vynechání nutnosti načítání

Obrázek 4.1: Graf závislosti doby běhu na velikosti matice s hustotou 10 %



.

Obrázek 4.2: Graf závislosti doby běhu na hustotě matice o velikosti 1000 × 1000



.

z paměti (a násobení) nulových prvků, zároveň je však výpočet jedné hodnoty významně časově náročnější.

Důsledek zvýšené složitosti jednotlivých výpočtů je vidět na grafu 4.2. Zatímco hustý formát je na hustotě matic nezávislý, řídké formáty jsou na hustotě matic kubicky závislé. Z výsledků měření také můžeme přibližně aproximovat, že (pro tyto konkrétní implementace a zcela náhodné matice) je CRS výhodné do hustoty 77 % a ELL 65 % (tedy poměrně vysoké hustoty).

Formát ELL si vedl významně hůře kvůli vyšší paměťové náročnosti a nevyužití jeho výhod v měřených neoptimalizovaných verzích.

4.7.1 Výběr optimálního algoritmu

Během návrhu byl proveden předběžný předpoklad, že efektivita CR a RC verzí (a zvolení lepší z těchto verzí pro konkrétní matice) je závislá na hustotě matic. Na základě tohoto předpokladu byl vytvořen mechanismus automatického výběru algoritmu.

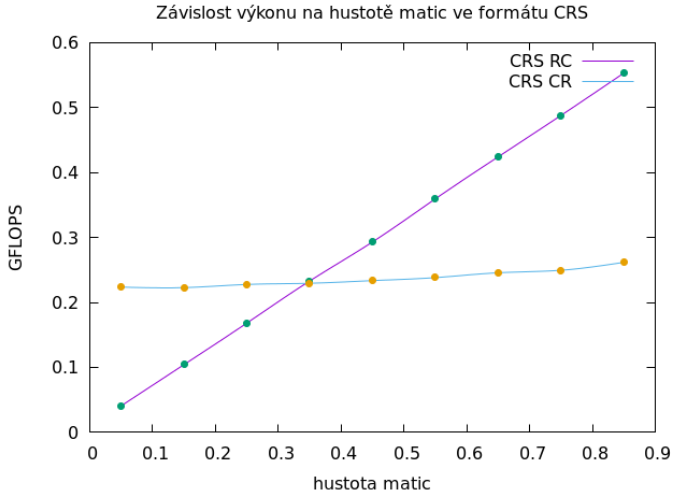
Pro ověření správnosti tohoto předpokladu a také odhad této hranice bylo tedy nutné provést testování efektivity jednotlivých verzí. Veškerá testování probíhala vždy na násobení dvou různých náhodných matic o velikosti 2048×2048 prvků a přičítání k nulové matici. Efektivita byla měřená pomocí GFLOPS, tedy miliard provedených aritmetických operací s reálnými čísly za sekundu. Kvůli přesnosti (počet násobení a sčítání je částečně závislý na rozmístění nenulových prvků) byl tento počet dopředu změřen na sekvenční verzi (nikoliv tedy odhadnut z velikosti matice a hustoty) a nebyly zahrnuty operace nad nulovými prvky, i když jsou v daném formátu prováděny (například ELL). Efektivita byla poté získána vydělením počtu nutných operací časem výpočtu.

Prvním testovaným formátem byl formát CRS. Výsledky tohoto testování můžete vidět na grafu 4.3. Na grafu je vidět, že u tohoto formátu se hypotéza potvrdila. Zatímco RC verze významně (lineárně) získává na efektivitě kvůli nižšímu procentu porovnávání indexů, efektivita CR verze je změnou hustoty ovlivněna jen velmi málo. Důvodem pro nárůst efektivity RC verze je snižování relativních rozdílů v délce běhu jednotlivých vláken, a tedy i snižování procenta času, kdy jsou rychlejší vlákna neaktivní (čekající na zbylá vlákna z warpu). Tento rozdíl zdá se převýšil efekt zvýšeného množství simultánních zápisů v atomické operaci (konflikt zápisů prodlužuje operaci).

Z grafu také můžeme odhadnout přibližnou mez, nad kterou je již RC verze efektivnější, a to 35 %.

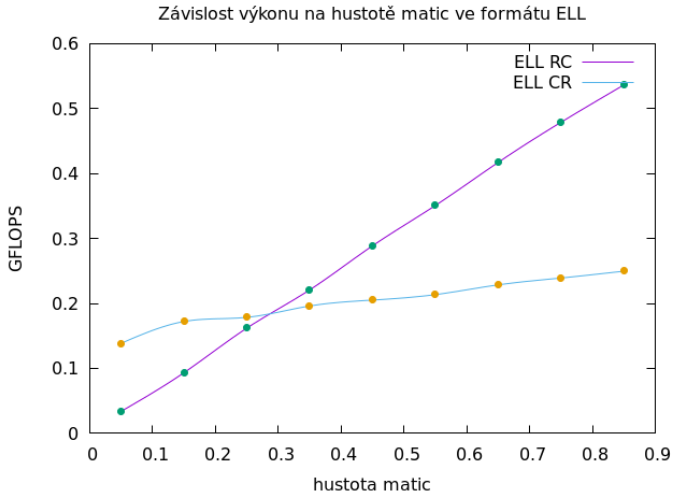
Pro ELL formát jsou naměřené výsledky na grafu 4.4. Snadno si lze povšimnout, že RC verze se chová stejně jako u formátu CRS. Je také vidět, že i neoptimalizovaná paralelní verze, na rozdíl od sekvenční, nemá významně rozdílnou efektivitu oproti CRS formátu (kvůli rozdílnému způsobu práce s pamětí a minimalizaci neaktivních vláken). Na druhou stranu CR verze

Obrázek 4.3: Graf závislosti doby běhu na hustotě matice formátu CRS



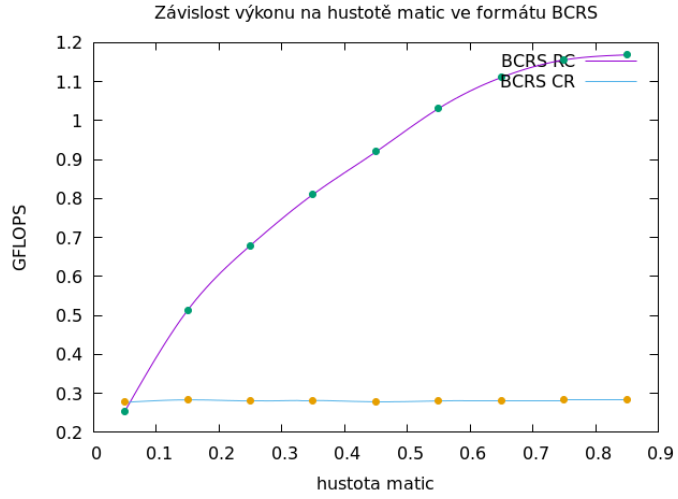
.

Obrázek 4.4: Graf závislosti doby běhu na hustotě matice formátu ELL



.

Obrázek 4.5: Graf závislosti doby běhu na hustotě matice formátu BCRS



.

získává mnohem významněji na efektivitě než u CRS formátu. Toto je způsobeno rozdílnou délkou řádků a z toho plynoucími operacemi s nulovými prvky.

I zde se potvrdila závislost na hustotě matic, dolní mez lepší efektivity RC verze je však nižší: 27 %.

Pro BCRS formát byla zvolena hodnota zaplněnosti bloků na 100 % (tedy bloky neobsahují žádné nulové hodnoty) a velikost bloku 2 prvky.

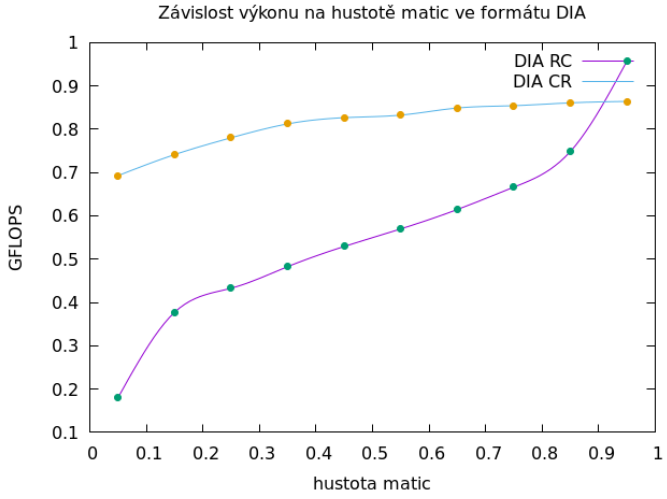
Na grafu 4.5 si můžeme všimnout, že CR verze se podle očekávání chová stejně jako u CRS a i efektivita je jen lehce vyšší (kvůli nižšímu množství atomických operací). Na druhou stranu RC verze zcela zřejmě velmi profituje ze sníženého množství operací s indexy, a proto je její efektivita několikanásobně vyšší. Ve zvýšení efektivity navíc hraje efekt lepší využití cache i načítání hodnot z globální paměti při samotném násobení dvou bloků (k hodnotám vedle sebe v paměti je přistupováno v po sobě následujících cyklech, nikoliv náhodně).

I u tohoto formátu je možné nalézt dolní hranici lepší efektivity RC verze, je to však pouhých 6 % hustoty matice.

Pro tento formát byl omezen maximální počet nenulových diagonál na $2m \cdot d$ (kde m je šířka matice d hustota matice), tedy procento nenulových diagonál je maximálně stejné s hustotou matice (při vyšší hustotě diagonál není matice příliš vhodná pro tento formát kvůli vysokému počtu nulových prvků v diagonálách).

Výsledky měření formátu DIA se nachází na grafu 4.6. Chování tohoto formátu je však na první pohled odlišné od ostatních zkoušených formátů. CR verze algoritmu je významně efektivnější než u všech ostatních formátů. Pro tuto vysokou efektivitu existuje hned několik významných důvodů. Zaprvé se jedná o způsob práce s pamětí, kde sousední vlákna přistupují vždy

Obrázek 4.6: Graf závislosti doby běhu na hustotě matice formátu DIA



.

k sousedním prvkům v pmaěti (protože přistupují k sousedním prvkům stejné diagonály). Dalším důvodem je nízké množství kolizí zápisů. Pokud by všechna vlákna pracovala synchronně, každé by zapisovalo přesně o řádek výsledkové matice dále než předchozí. Protože značná část vláken pracuje synchrhonně (případně je práce na nich zpuštěna až po dokončení práce na předchozích), jsou tímto způsobem kolize zápisů minimalizovány. Posledním důvodem je minimální neaktivita vláken, protože veškerá vlákna mají stejnou zátěž (množství nutných operací).

I RC verze algoritmů je pro tento formát téměř dvakrát efektivnější než CRS. Hlavním důvodem této efektivity je snížená práce s pamětí (kvůli implicitnímu indexování pomocí offsetu diagonály) a z toho plynoucí lepší využití cache. Se zvětšujícím se počtem diagonál se využití cache zhoršuje, to je však více než kompenzováno sníženým množstvím operací s indexy (souhra těchto dvou jevů pak vede k ne zcela lineární křivce efektivity).

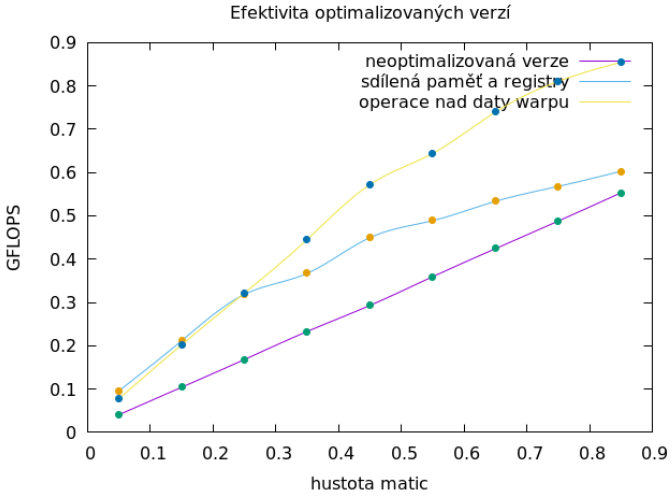
Ačkoliv i u tohoto formátu se předpoklad existence meze splnil, tato mez se (kvůli vysoké optimálnosti CR verze) nachází až na 90% hustotě.

4.8 Efektivita optimalizací RC verze

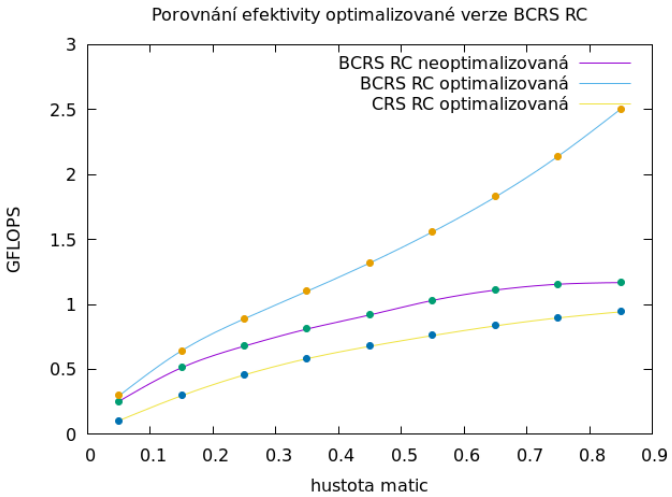
Na grafu 4.7 si lze prohlédnout výsledky optimalizací. Testování optimalizací bylo prováděno na CRS formátu a náhodných řídkých maticích o velikosti 2048×2048.

Efektivita neoptimalizované verze algoritmu je na první pohled lineárně závislá na hustotě. Verze s využitím sdílené paměti a přednahráváním hodnot do registrů si vede jednoznačně lépe na všech testovaných maticích. Přibližně do hustoty 25 % roste efektivita lineárně (s vyšším koeficientem), u vyšších

Obrázek 4.7: Graf efektivity optimalizací RC verze



Obrázek 4.8: Graf efektivity optimalizací RC verze na BCRS



hustot však nízká efektivita načítání indexů matice B není dobře maskována (při častém střídání vyšších indexů A a B není ani ideálně využitá cache), a proto je ukončen lineární růst efektivity a přinosy se snižují.

Poslední testovaná varianta je verze s operacemi warpů (a spoluprací více warpů na jednom výsledku). Tato verze je kvůli vyššímu overheadu (a nižšímu množství aktivních vláken) pomalejší na maticích s méně prvky (méně hustých), ale její efektivita roste lineárně až do hustoty 45 %. I dále jsou její přínosy v efektivitě značné a obecně se dá prohlásit za nejlepší variantu pro dostatečně velké matice.

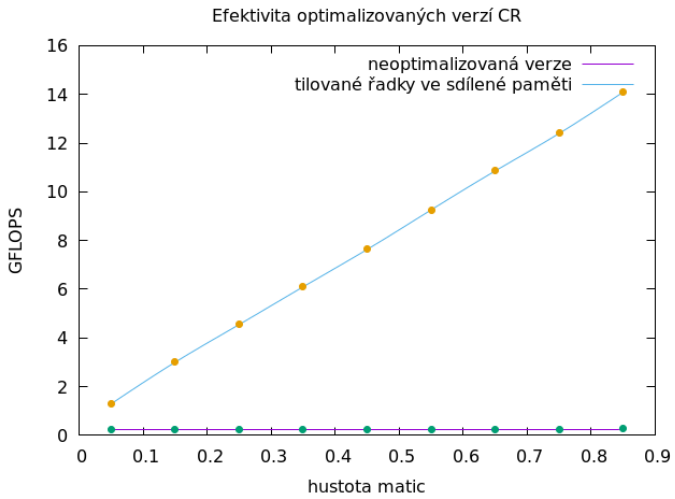
Tyto optimalizace dosáhly ještě významnějších přínosů pro BCRS formát.

Porovnání optimalizované verze CRS a BCRS pro blokovou matici (s velikostí bloku 2×2) si lze prohlédnout na obrázku 4.8. Významně jiný tvar křivky efektivity je způsoben větším množstvím vlivů působících na efektivitu implementace, zejména se (navíc oproti CRS formátu) jedná o efektivitu násobení jednotlivých bloků.

4.9 Efektivita optimalizací CR verze

Výsledek optimalizace CR verze algoritmu je vyobrazen na grafu 4.9. První, čeho si lze všimnout je, že optimalizovaná verze je výrazně efektivnější ve všech testovacích případech oproti neoptimalizované verzi (díky menší zátěži globální paměti, která je obvyklým limitujícím faktorem).

Obrázek 4.9: Graf efektivity optimalizací CR verze



.

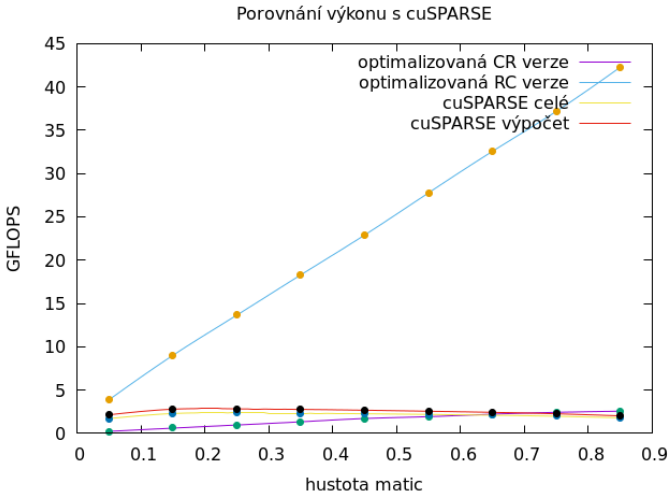
Dále si lze všimnout, že efektivita optimalizovaného algoritmu stoupá téměř lineárně s hustotou matice (a tedy počtem prvků na řádek). Toto je způsobeno rozdílnou rychlostí růstu funkce množství operací na řádek ($O(n^2)$) a funkce určující počet načítání z globální paměti na řádek ($O(n)$ pro dostatečně malé matice a $O(n^2)$ pro větší, ale s malým koeficientem u kvadratického členu, více v subsekci 3.5.5), protože počet načítání z globální paměti je i zde bottleneckem výkonu implementace. U dostatečně malých matic je rozdílem právě lineární funkce. Pro ještě větší matice (než se bez problémů vejdou do globální paměti testovací GPU) bude výkon algoritmu konvergovat (shora omezen) k podílu kvadratických členů těchto dvou funkcí.

Dá se očekávat, že existuje maximum efektivity, kterého algoritmus dosáhne pro dostatečně velké a husté matice (když náročnost operace násobení a sčítání přesáhne vliv paměťových operací). Takto velké matice jsou však kvůli omezení velikosti výstupní matice nepraktické.

4.10 Porovnání s cuSPARSE

V neposlední řadě bylo taktéž nutné porovnat efektivitu vytvořených implementací s jinými dostupnými knihovnami. Pro kontrolu byla vybrána knihovna cuSPARSE, konkrétně její operace cusparseScsrgemm, která je blízká implementované operaci (vrací ovšem matici v řídkém formátu). Tato knihovna umožňuje pouze výpočet netransponované matice krát netransponovaná (tedy rozdílný od obou zmíněných v této práci). Před voláním této operace je nutné zavolat operaci cusparseXcsrgemmNnz, která zjistí, které pozice výsledkové matice budou nenulové a následně je nutné alokovat výsledkovou matici, proto byly výsledky rozděleny na výsledky s předvýpočtem (celá procedura) a pouhé násobení.

Obrázek 4.10: Porovnání efektivity optimalizovaných algoritmů s cuSPARSE



Tabulka 4.1: Porovnání doby běhu výpočtu v ms

hustota v %	CR verze	**RC verze**	cuSPARSE
0.05	827	**50**	90 | 113
0.15	2864	**193**	623 | 758
0.25	5002	**354**	1705 | 2049
0.35	7084	**518**	3420 | 4067
0.45	9117	**684**	5856 | 6902
0.55	12097	**842**	9143 | 10677
0.65	14717	**1003**	13434 | 15552
0.75	17878	**1169**	19203 | 21988
0.85	21778	**1321**	27554 | 31091

Výsledky tohoto porovnání můžete vidět na grafu 4.10. Přesné naměřené

časy si pak můžete prohlédnout v tabulce 4.1. Veškeré časy v tabulce jsou bez přesunů mezi pamětí procesoru a grafické karty a jiné režie CPU operací. U cuSPARSE jsou uvedeny časy samotného výpočtu a také časy provedení celé procedury, tedy se zjištěním struktury výsledkové matice (v tomto pořadí, oddělené |). Z dat je vidět, že CR verze byla významně efektivnější než cuSPARSE ve všech měřených bodech. Ačkoliv RC verze si u řidších matic vede hůře než cuSPARSE, s rostoucí hustotou matic (a množstvím prvků na řádek) však postupně efektivita RC verze cuSPARSE překonává. Vyšší efektivita cuSPARSE algoritmu pro nižší hustoty matic (a způsob použití této funkce) taktéž ukazuje možnost případné budoucí optimalizace RC verze využitím algoritmu pro určení struktury výstupní matice před samotným násobením a vyhnutí se tak nutnosti výpočtů skalárích součinů pro nulové prvky.

Závěr

V práci jsem se zabýval analýzou stavu řešení problému násobení (hustých) matic a jednotlivými subkubickými algoritmy. Dále jsem se zabýval formáty pro ukládání řídkých matic, jejich specifiky a výhodami. Také jsem provedl rešerše „konkurenčních“ knihoven pro práci s řídkými maticemi (NIST Sparse BLAS a Eigen).

V druhé části práce jsem provedl návrh základního interface (a funkcionality) funkce pro násobení matic a vybral čtyři formáty (CRS, BCRS, ELL, DIA) pro implementaci. Dále jsem detailněji popsal sekvenční algoritmy pro převod matic v těchto formátech na husté matice a zpět, transpozici a samotné násobení. Posléze jsem navrhl dva způsoby paralelizace násobení matic (CR, RC) a vyslovil hypotézu, že existuje taková hustota matic, že méně husté matice jsou efektivněji násobené prvním způsobem, zatímco více husté matice jsou efektivněji násobené způsobem druhým. V neposlední řadě jsem taktéž navrhl strukturu základní knihovny pro násobení řídkých matic.

Ve třetí části jsem implementoval navrženou kostru knihovny. Dále jsem také během této fáze vývoje implementoval paralelní násobení v CUDA a to dvěma způsoby pro každý formát (CR, RC). Tyto algoritmy byly dále pečlivě optimalizovány pro využití co nejvyšší části potenciálu GPU. V textové části jsou rozepsány tyto postupy optimalizace, důvody jejich provádění a specifika technologie CUDA, se kterými se v takovýchto případech je nutné potýkat. Také jsem krátce zmínil některé další nutné úpravy, které byly provedeny nad rámec návrhu.

V poslední části práce jsem popsal způsoby testování, hardware, na němž bylo testování prováděno, a nastavení kompilace kódů. V této části byl taktéž popsán způsob získávání testovacích dat (a implementace generátoru) a formát ukládání vstupních dat. Provedl jsem také testování správnosti implementací. Bylo provedeno testování efektivity optimalizačních technik, jejich porovnání s neoptimalizovanými verzemi a zhodnocení těchto výsledků. Hypotéza existence bodu vyšší efektivity RC byla potvrzena pro neoptimalizované verze. Pro optimalizované verze algoritmů však již nebylo tento bod možné najít kvůli

významně lepší optimalizovatelntosti CR verze a z toho plynoucí lepší celkové efektivitě (tento bod by se teoreticky mohl nacházet až za hranicí 100% hustoty). Optimalizovaná verze RC algoritmu však přesto může nalézt uplatnění při násobení matic, u nichž velikost výstupní matice přesahuje velikost paměti GPU. V neposlední řadě bylo také v této části práce provedeno srovnání s knihovnou cuSPARSE (zabývající se taktéž násobením řídkých matic na GPU) a bylo zjištěno, že zvláště RC verze algoritmu významně překonává efektivitou algoritmus, který tato knihovna používá (2-20x dle hustoty vstupních matic).

Tato práce splnila všechny body zadání. Za zvláště dobrý výsledek pak považuji překonání efektivity v současnosti využívaných implementací násobení řídkých matic.

Téma, jemuž jsem se věnoval, je velmi rozsáhlé, a proto na něm v budoucnu lze provést ještě mnoho další práce. Například by bylo vhodné rozšířit knihovnu o další formáty ukládání řídkých matic, přidat další funkcionalitu práce s řídkými maticemi, rozšířit algoritmy o možnost ukládání výsledkové matice v řídkém formátu a dále experimentovat s optimalizacemi a využitím jiných algoritmů pro násobení a předvýpočty. Tato vylepšení a úpravy však již nespadají do rozsahu mé diplomové práce.

Literatura

[1] Bell, N.; Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov 2009, ISSN 2167-4329, s. 1–11, doi:10.1145/1654059.1654078.

[2] Dombek, D.; Kalvoda, T.; Kleprlík, L.; aj.: Lineární algebra. 2018. Dostupné z: `http://sagemath.fit.cvut.cz/deploy/bi-lin/lin-text.pdf`

[3] Stanimirovič, I. P.; Tasič, M. B.: PERFORMANCE COMPARISON OF STORAGE FORMATSFOR SPARSE MATRICES. *FACTA UNIVERSITATIS (NIŠ)*, ročník 24, 2009: s. 39–51.

[4] Grossman, M.: 101 Ways to Store a Sparse Matrix. 2015. Dostupné z: `https://medium.com/@jmaxg3/101-ways-to-store-a-sparse-matrix-c7f2bf15a229`

[5] Smailbegovic, F.; Gaydadjiev, G.; Vassiliadis, S.: Sparse matrix storage format. 01 2005.

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

[7] Østerby, O.; Zlatev, Z.: *Direct Methods for Sparse Matrices*, ročník 9. 01 1983, ISBN 3-540-12676-7.

[8] King, J.; Gilray, T.; Kirby, R. M.; aj.: Dynamic-CSR : A Format for Dynamic Sparse-Matrix Updates. 2016.

[9] Strassen, V.: Gaussian Elimination is Not Optimal. *Numer. Math.*, ročník 13, č. 4, Srpen 1969: s. 354–356, ISSN 0029-599X, doi:10.1007/BF02165411. Dostupné z: `http://dx.doi.org/10.1007/BF02165411`

[10] Li, J.; Ranka, S.; Sahni, S.: Strassen's Matrix Multiplication on GPUs. *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, 12 2011: s. 157–164, doi:10.1109/ICPADS.2011.130.

[11] C. Douglas, C.; Heroux, M.; Slishman Roger M, G.: Gemmw: A Portable Level 3 Blas Winograd Variant Of Strassen's Matrix–Matrix Multiply Algorithm. *Journal of Computational Physics*, ročník 110, 07 1997, doi: 10.1006/jcph.1994.1001.

[12] Williams, V. V.: Multiplying matrices inO(n2.373)time. 2014. Dostupné z: https://people.csail.mit.edu/virgi/matrixmult-f.pdf

[13] Remington, K. A.; Pozo, R.: NIST Sparse BLAS User's guide. 1996. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.54.8763&rep=rep1&type=pdf

[14] Pozo, R.: Sparse Basic Linear Algebra Subprograms (BLAS) Library. 2006. Dostupné z: https://math.nist.gov/spblas/

[15] Ronald Pozo, A. L., Karin Remington: Sparse BLASS Issues in C/C++. 2006. Dostupné z: http://www.netlib.org%2Futk%2Fpapers%2Fsblas-slides%2Fpozo.ps&usg=AOvVaw0cwtXNHq4xLtVgTPNz9EDS

[16] Jacob, B.; a spol., G. G.: Documentation Eigen 3.3.7. 2019. Dostupné z: http://eigen.tuxfamily.org

[17] Jacob, B.; a spol., G. G.: Eigen Bitbucket. 2019. Dostupné z: https://bitbucket.org/eigen/eigen/

[18] Brown, K. S.: Hermitian Matrices. 2019. Dostupné z: https://www.mathpages.com/home/kmath306/kmath306.htm

[19] NVIDIA: cuSPARSE :: CUDA Toolkit Documentation. 2019. Dostupné z: https://docs.nvidia.com/cuda/cusparse/index.html

[20] Fundamentals, S. T.: Unit Testing. 2019. Dostupné z: http://softwaretestingfundamentals.com/unit-testing/

[21] freedesktop.org: cppunit test framework. 2018. Dostupné z: https://freedesktop.org/wiki/Software/cppunit/

[22] amertkara: How to launch cppunit test? 2014. Dostupné z: https://stackoverflow.com/a/26836387

[23] of Florida, U.: SuiteSparse Matrix Collection. 2019. Dostupné z: https://sparse.tamu.edu/

Kapitola 5

Seznam použitých zkratek

BCRS Block Compressed Row Storage

CCS Compressed Column Storage

COO Coordinate format

CPU Central processing unit

CR Algoritmus násobící matice způsobem $column \times row$

CRS Compressed Row Storage

CUDA Compute Unified Device Architecture

DIA Diagonal format

DRY Don't Repeat Yourself

ELL ELLPACK

FLOPS Floating point operations per second

GPU Graphics processing unit

HYB Hybrid format

RAM Random-access memory

RC Algoritmus násobící matice způsobem $row \times column$

Kapitola 6

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD a instalační instrukce
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF