FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Paralelní násobení řídkých matic
Student:	Bc. Lukáš Simulík
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Násobení matice maticí je jednou z nejběžnější operací v numerické lineární algebře. Objevuje se rovněž v grafových algoritmech, například pro počítání cyklů délky tři či prohledávání do šířky s více zdroji.

1. Nastudujte si algoritmy pro paralelní platformy zabývající se násobením dvou řídkých matic [1, 2, 3, 4, 5, 6].
2. Zaměřte se na vnitřní architekturu a použité maticové formáty v aktuálních state-of-the-art algoritmech.
3. Popište použité optimalizace ve zmíněných algoritmech s ohledem na specifika, která přináší paralelizace pro CPU a GPU [1, 5, 6].
4. Zaměřte se na situaci při násobení obecných matic. (Novější algoritmy pro porovnávání se navzájem používají pouze jednu malou podmnožinu maticových operací: C = A*A případně C = A * A^T.)
5. Navrhněte a implementujte vlastní řešení.
6. Změřte výkonnost na sadě testovacích dat (vybraných po dohodě s vedoucím), srovnejte s referenční implementací a diskutujte výsledky.

Zdroje:

[1] BRMerge https://arxiv.org/abs/2206.06611
[2] bhSPARSE https://www.sciencedirect.com/science/article/pii/S0743731515001185
[3] nsparse https://ieeexplore.ieee.org/document/8025284
[4] spECK https://dl.acm.org/doi/10.1145/3332466.3374521
[5] AC-SpGEMM https://people.mpi-inf.mpg.de/~rzayer/pappe/AC_SpGEMM.pdf
[6] OpSparse https://arxiv.org/abs/2206.07244

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 28. září 2022 v Praze.

Diplomová práce

PARALELNÍ NÁSOBENÍ ŘÍDKÝCH MATIC

Bc. Lukáš Simulík

Fakulta informačních technologií
Katedra teoretické informatiky
Vedoucí: doc. Ing. Ivan Šimeček, Ph.D.
29. června 2023

Odkaz na tuto práci: Simulík Lukáš. *Paralelní násobení řídkých matic.* Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat především docentovi Šimečkovi za jeho přínosy a ochotu, se kterou odpovídal na mé dotazy a ukazoval mi další pohledy na danou problematiku.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při 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, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 29. června 2023 .....

Abstrakt

Práce se zabývá paralelním násobením řídkých matic a obsahuje analýzu a porovnání výkonu vlastního implementovaného algoritmu v OpenMP s existujícími algoritmy. Byly popsány již existující algoritmy a jejich postupy a optimalizace, díky kterým dokáží dosáhnout svých časů. Tyto myšlenky byly použity ve vlastní implementaci. Pro porovnání byla provedena sada měření, při kterých byly testovány různé matice ze souboru matic SuiteSparse. Naměřené výsledky byly poté analyzovány a porovnány, aby se zjistila relativní efektivita implementovaného algoritmu v porovnání s existujícími algoritmy.

Klíčová slova algoritmus, řídká matice, násobení, OpenMP, paralelizace

Abstract

The work focuses on parallel sparse matrix multiplication and includes analysis and comparison of performance between a custom implemented algorithm in OpenMP and existing algorithms. The existing algorithms and their approaches and optimizations, which enable them to achieve their respective times, have been described. These ideas were utilized in the custom implementation. For comparison, a set of measurements was performed, testing various matrices from the SuiteSparse matrix collection. The obtained results were then analyzed and compared to determine the relative efficiency of the implemented algorithm compared to the existing algorithms.

Keywords algorithm, sparse matrix, multiplication, OpenMP, parallelization

Kapitola 1
Úvod

1.1 Základní terminologie

V textu se bude používat značení, které je silně inspirováno citovanými pracemi, liší se jen v drobných detailech. Matice budou značeny velkými písmeny, rozměry matice malými.

Maticové násobení $C = A * B$, kde A má rozměry $m \times n$ a B $n \times p$, je definováno jako $C_{ij} = \sum_{k=1}^{n} A_i k * B_{kj}$ pro každé $i = 1, 2, ..., m-1, m$; $j = 1, 2, 3, ..., p-1, p$. Písmena A, B, C budou mít vždy stejný význam, tedy C bude výslednou maticí, A a B budou levou, respektive pravou stranou násobení.

Pokud se bude pracovat se čtvercovými maticemi, kde $m = n$, bude se pro jednoduchost používat pouze n. Mocnění se bude zapisovat jako $C = A^2 = A * A$, v případě nevyhovujících rozměrů dojde nejdříve k transpozici pravé strany, $C = A * A^T$.

1.2 Řídká matice

Termín řídká (sparse) matice je poměrně vágně definován. Počet nenulových prvků musí být dostatečně malý v porovnání s $O(mn)$, aby se dalo využít formátů řídké matice. [1] Každý formát má jinou paměťovou náročnost, proto zlom, kdy začíná být formát relevantní, je individuální.

Jedna ze známých matic, *webbase-1M*, značící propojení webů, obsahuje jen 3 105 536 nenulových prvků, hustá matice by jich měla mít 8500krát více. [2]

Vedle řídké matice se dá zadefinovat i pojem velmi řídké matice (hypersparse). Zde je počet nenulové prvků shora omezen $O(n)$, může tedy dojít k tomu, že řádky jsou bez nenulových prvků. [3]

Ve vzorcích, pro zkrácení zápisu, se počet nenulových prvků v matici bude označovat anglickou zkratkou tohoto termínu – *nnz*.

1.3 Kompresní poměr matice

Jedním ze způsobů, jak jedním číslem popsat vlastnost řídké matice, je takzvaný kompresní poměr. Jde o metriku spojenou s počítáním mocniny matice. Spočítá se jako podíl počtu násobení, co muselo být provedeno, a počtu nenulových prvků ve výsledné matici. Spolu se znalostí maximální a průměrné délky nenulového řádku to poukazuje na pravidelnost matice. [4]

Tato terminologie není jednotná napříč pracemi, jinde se nepracuje s poměrem ale faktorem, a dělencem se nazývá počet operací s plovoucí čárkou – *FLOP* – což vede k identickému pohledu na danou problematiku. [5]

1.4 Původ matic

Matice, které budou v práci používané, budou pocházet ze dvou zdrojů:

- veřejné kolekce,
- generátor grafů.

Tou nejvýznamnější kolekcí řídkých matic je *SuiteSparse Matrix Collection*. Matice, které se zde vyskytují, jsou používané pro porovnávání výkonu mezi jednotlivými algoritmy. Popisují různé problémy, díky čemuž je sada dostatečně rozmanitá, obsahuje například symetrické grafy, kombinatorické problémy, chemické procesy. Většina matic je čtvercových a různě velkých, proto se ve většině případů provádí při měření výkonu pouze mocnění, nenásobí se různé druhy problémů mezi sebou. Částečnou odpovědí na tento problém může být generátor grafů. Zde je možné vytvořit celou řadu čtvercových matic, které mají různé vlastnosti. Generátory pouze s danou pravděpodobností rozhodují, zda mezi vrcholy bude hrana, což negativně ovlivní pestrost výsledků v porovnání s kolekcí – hodnoty budou vždy 1. Posledním způsobem, jak matice vytvořit, je úprava již existujících. Může jít o transpozici, či zúžení na několik sloupců. Tato myšlenka není nová, byla použita například v práci z roku 2018, kdy bylo prováděno násobení matice stejnou maticí, ze které bylo vybráno jen několik málo sloupců [5].

1.5 Ukládání matice na disk

Pro uchovávání matic na disku se zřejmě nejčastěji používá formát nazývaný Matrix Market. Nejde o jediný způsob, na webové stránce – kolekci matic – se vyskytují ještě další dva [2].

První myšlenky pocházejí z roku 1996 [6]. Formát je dvojího druhu, v jednoduché podobě je možné jej rozdělit na variantu pro hustou a řídkou matici. Hustá matice je zapsána jako pole a dále bude popisována pouze varianta pro matice řídké.

Celý formát lze rozdělit do tří spolu souvisejících částí. Na prvním řádku musí být vždy text začínající dvěma procenty a frází *MatrixMarket*, za kterou jsou další informace, čtyři klíčová slova dále upřesňující co za data se v souboru nachází.

První dvě slova určují objekt a způsob uložení. Pro mé účely jde svým způsobem o konstanty, vždy se bude pracovat s maticemi, dalšími možnostmi je graf či vektor, v souřadnicovém formátu. Třetí definuje datový typ uchovávaných hodnot. Vedle celých, desetinných, a komplexních čísel se zde může ještě vyskytovat *pattern*, který nespecifikuje hodnotu. Lze na něj narazit u popisu grafů kde se takto popisuje existence hrany. Poslední slovo je vyhrazeno pro popis symetričnosti matice. Pokud je matice symetrická, jsou uloženy pouze hodnoty pod diagonálou a diagonála. Zbylá čísla jde jednoznačně určit.

Další částí je jeden řádek se třemi celými čísly. První dvě udávají výsku a šířku matice, třetí počet nenulových hodnot. Popisem nenulových hodnot se zabývá poslední část. Ve většině případů jde opět o trojici, kde první číslo určuje řádek, druhé sloupec matice. U *pattern* je třetí číslo, jinak určující obsah buňky, vynecháno.

Formát podporuje komentáře, používá se pro ně symbolu procenta.

1.6 Generátor grafů

Pro vytváření matic lze použít generátory grafů. Jejich výstupem je matice sousednosti, kde nenulová hodnota (jedna) značí existenci hrany mezi dvěma vrcholy. Matice nemusí být nutně symetrická, směr hrany se u orientovaných grafů respektuje. Výsledek je jistou podmnožinou obecné matice. Vždy bude čtvercová, a možné hodnoty jsou omezeny pouze na nulu či jedničku.

Pro vytváření grafů je používáno různých modelů. Všechny pracují nějakým způsobem na pravděpodobnosti, že hrana bude mezi dvěma vrcholy vést. Díky různým strategiím jak graf

generovat, lze tvořit rozdílně vypadající struktury. Asi tím nejjednodušším je Erdős-Rényi model, kdy se vezme graf o N vrcholech, a pak se každá hrana, kterých je $\frac{N*(N-1)}{2}$, s pravděpodobností p do grafu buď umístí, či ne. [7]

Další variantou je například model R-Mat. Ten má celkem čtyři parametry, jejich součet musí být roven jedné ($a+b+c+d=1$). Generuje se rekurzivně, pro zjednodušení bude popsána jen matice s rozměry $N=2^k$. U $k=1$ se každý z parametrů umístí do svého kvadrantu. Jde o graf o dvou vrcholech, může mít proto čtyři hrany (orientovaný graf dovolující smyčky). Každá hrana má tak jasně danou pravděpodobnost výskytu danou parametrem. Pro větší graf, $k=2$, se každá buňka zvětší a místo jedné hrany bude reprezentovat čtyři, podobně, jako je tomu v případě $k=1$. Pravděpodobnost nové hrany vyplyne z původní pravděpodobnosti a součinu s parametrem spadajícího do daného kvadrantu. [8]

Popularita R-Mat plynula z jeho použití ve známém benchmarku Graph 500 a přímočará paralelizace. Pro každou hranu ale spotřebuje ve své základní variantě logaritmický čas. Kvůli tomu se do popředí dostávají modely pracující v čase lineárním, které stejně jako R-Mat jdou velmi dobře škálovat. [8]

Kapitola 2

Formáty pro ukládání řídkých matic

Formát matice slouží dvěma účelům. Tím hlavním je omezit paměťovou náročnost a druhým data umístit takovým způsobem, aby docházelo k nejmenšímu množství *cache miss*.

Již v roce 1998 došlo k pozorování, že je důležitější vyhýbat se *cache miss* než omezit počet zbytečných operací, kterou může být násobení nulou. K této operaci nedochází zas tak často díky způsobu, jakým se matice reprezentují [9].

U násobení matice maticí se používají s formáty matic příliš neexperimentuje, a autoři zůstavají věrní základním třem formátům, a to i přesto, že většína algoritmů k problému přistupuje jako k mnohonásobnému násobení matice vektorem [4, 10].

I přesto, že se na problém často nahlíží jako na mnohonásobné násobení matice vektorem, formáty zmiňované u algoritmů zabývající se násobením matice vektorem se v implementacích neobjevují, a autoři zůstávají věrní těm, které lze zařadit mezi ty základní.

V této kapitole budou zmíněny tyto běžně používané, jedna z upravených variant pro speciální podkategorii řídkých matic, a některé složitější formáty zmiňované u násobení s vektorem.

2.1 COO

Zřejmě nejjednodušším formátem pro pochopení je takzvaný souřadnicový formát, kde se pro každou nenulovou buňku ukládá její hodnota a pozice v matici, řádek a sloupec. Dovoluje náhodný přístup na jakoukoliv položku, hodnoty lze vkládat libovolně, nezáleží na pořadí.

Jeho implementace je snadná, jde o tři pole, kde jedno z nich ukládá řádkový index, druhý sloupcový index, a poslední slouží pro hodnotu položky. Všechny tři pole mají stejnou délku, a tou je počet nenulových hodnot v matici, $O(nnz) = O(nnz + nnz + nnz)$.

V algoritmech, které budou dále zmiňovány, nejde o formát nejběžnější, a to i přesto, že má všchny výhody zmiňované výše. Nelze totiž tvrdit, že jde o relevantní výhodu, je to spíše o benevolenci v ukládání a její využívání by vše komplikovalo či rovnou znemožňovalo. Převody mezi formáty nelze brát jako velkou výhodu, k načtení matice dojde jednou, a algoritmus dále pracuje jen s jedním formátem. Libovolné uspořádání prvků je přímo nežádoucí, algoritmy pracují s celými řádky či sloupci a v jednotlivých mezikrocích ve výpočtech se počítá s jakousi lokalitou, dochází k řazení a shlukování. S tím lehce souvisí další nepoužívaná výhoda v náhodném přístupu; cílem je pracovat s celými řádky, a ukazuje se, že stačí v konstantním čase být schopný přistoupit na první položku v řádku, respektive sloupci. To je jeden z bodů, se kterým pracuje *CSR* 2.2. [11]

2.2 CSR a CSC

CSR a jeho obdobná varianta *CSC* jde brát jako ty hlavní formáty. Každý z algoritmů, který bude v této prác zmíněn, s jedním z nich pracuje. Popis bude odpovídat tomu, jak by se dal zjednodušit souřadnicový formát2.1. Bude popsána pouze varianta komprimovaných řádků, *CSR*, varianta se sloupci se dá popsat analogicky.

Pokud se seřadí hodnoty v souřadnicovém formátu podle řádku, není nutné u každé hodnoty zmiňovat její řádkový index. Díky tomu je možné toto pole plné indexů nahradit jiným, které bude evidovat začátky jednotlivých řádků, index na první položku řádku v poli hodnot. Rozdíl počátečních indexů mezi sousedy pole odpovídá počtu hodnot v jednotlivých řádcích. Aby se dalo s polem indexů dobře pracovat v algoritmech, na poslední pozici v poli se umístí počet všech nenulových prvků v matici.

Není poté možné jednoduše zjistit řádkový index jakékoliv hodnoty, ale protože se při násobení pracuje s celými řádky, algoritmy tuto informaci nad jednotlivými prvky nepotřebují. Protože se očekává, že výška matice je menší než počet všech nenulových hodnot, je tento formát úspornější než souřadnicový. Navíc v sobě nepřímo obsahuje informaci o délce řádků a na start řádku jde přistupovat pomocí jediného indexu, který je jasně definován. [12]

2.3 DCSR

Pro grafy sociálních sítí je běžné, že spousta řádků a sloupců jsou samé nuly. To vede k velmi nízkému počtu nenulových prvků – $nnz < n$ – a pro ty není formát komprimovaných řádků tím nejefektivnějším způsob, jak data uložit.

Právě pro tento účel vznikl formát *DCSR*, což je speciální varianta již zmiňovaného *CSR*, respektive *CSC*. Písmeno *D* na začátku zkratky značí *double*, jde tedy o dvojitou komprimaci.

Základní varianta ve všech případech ukládá index na začátek řádku. U matic s mnoha prázdnými řádky obsahuje toto pole ve spoustě případů irelevantní informaci. Zde se ukládají pouze řádky, které nejsou prázdné. Tento postup způsobuje s indexováním, již není možné v konstantním čase přistupovat k první položce v řádku. Aby se tento problém eliminoval, zavádí se další pole. To je dlouhé jako počet řádků matice vydělené počtem nenulových řádků. Funguje podobně jako původní pole pro řádky, rozdíl mezi sousedními položkami udává počet nenulových řádků, které se v daném intervalu vyskytují.

Díky tomu se nenulové řádky umístily do buněk, a pomocí půlení intervalu je stále možné přistupovat ke každému řádku v matici. Nejdříve se zjistí, do které buňky vyhledávaný řádek patří, a pak se na daném intervalu hledá řádek, který má stejný index.

Pokud by matice prázdné řádky neobsahovala, byl by tento formát horší, než standardní přístup – bylo by zde pouze pole navíc. [3]

2.4 BSR

Další způsob, jak ukládat matice, je po blocích, ne po samostatných hodnotách. Díky tomu má formát jasně definované rozměry a je možné vylepšit paměťovou lokalitu.

Na bloky lze nahlížet stejným způsobem, jako na komprimované formáty. Jen se na daném sloupcovém indexu nenachází pouze jedna hodnota, ale nějaký blok předem definovaných rozměrů. Rozměry mohou být různé, platí ale, že rozměry jsou poměrně malé, délka strany nebude větší než šest. [13]

Pokud by velikost bloku byla jedna, jednalo by se o standardní formát komprimovaných řádků 2.2. Zvolit správné rozměry není jednoduché – hrozí, že spousta hodnot bude nulových, což degraduje efektivitu formátu a následně i algoritmů, které jej budou používat. [14]

2.5 Diagonální

Pokud čtvercová matice obsahuje velké množství diagonál, které jsou plné nul, je možné použít tento specializovaný formát. Místo na řádky či sloupce, zaměřuje se na diagonály s nenulovými prvky.

Registruje se velikost strany matice a počet nenulových diagonál. Dále se používají dvě pole. Jedno z nich slouží pro ukládání vzdálenosti diagonály od hlavní diagonály. Kladné číslo znamená, že uložená diagonála je nad hlavní diagonálou a naopak. Hlavní diagonála odpovídá nulové vzdálenosti. Pro ukládání hodnot se používá 2D pole, jeho výška odpovídá výšce matice, šířka počtu nenulových diagonál.

Sloupce odpovídají jednotlivým diagonálám. Data se do pole ukládají na základě vzdálenosti od diagonály. Pokud je vzdálenost kladná, hodnoty se ukládají od počátku sloupce, u záporných čísel se začínat indexovat od vzdálenosti. Počet hodnot je rozdílem mezi délkou strany matice a vzdálenosti od hlavní diagonály.

Jeho kvalitu nejde jednoduše shora omezit, vše záleží na struktuře matice, nejenom počtu nenulových prvků – například symetrické matice jde uložit velmi výhodně – stačí evidovat pouze jednu polovinu diagonál. Protože se používá 2D pole, je dobré, aby většina diagonál byla blízko hlavní diagonály. Čím dále od hlavní diagonály, tím méně hodnot diagonála obsahuje a sloupec obsahuje čím dál tím méně relevantních hodnot. [15]

2.6 Skyline

Existují i další formáty, které jako svou referenci používají hlavní diagonálu. Takovým je například Skyline. Podobně jako diagonální formát 2.5, jde o poměrně specializovaný formát, které najde své uplatnění zejména u symetrických či triangulárních matic, protože se ukládají pouze hodnoty na diagonále a nad/pod diagonálou. Pokud by se tedy ukládala obecná čtvercová matice, je nutné uložit zvlášť horní a dolní část matice, což není praktické, ani efektivní – hlavní diagonála bude uložena dvakrát.

Ukládají se celé řetězce hodnot od nejvzdálenějšího prvku od hlavní diagonály po hlavní diagonálu včetně. Díky tomu není třeba ukládat žádné informace o sloupcovém indexu. Stačí vědět, které sloupec se aktuálně zpracovává, a délku uchovávaného řetězce. Pracuje se se dvěma poli – první z nich ukládá indexy na začátky řetězců hodnot, druhé tyto hodnoty. Pole s indexy je dlouhé jako velikost strany matice plus jedna. Poslední položka obsahuje počet uložených hodnot. Díky tomy jde délka řetězce jednoduše určit, stačí spočítat rozdíl mezi dvěma hodnotami v tomto poli (pro zjištění délky řetězce sloupce 3, odečte se počáteční index sloupce 4 od sloupce 3).

Formát tedy trpí na podobný problém, jako diagonální formát; pokud se v matici objevují nenulové prvky dosti vzdálené od hlavní diagonály, hrozí, že se bude ukládat příliš mnoho nulových prvků, které jiné formáty ignorují, což zhorší paměťovou efektivitu. [16]

2.7 Formáty pro násobení matice vektorem

I přesto, že lze na problematiku násobení dvou matic nahlížet jako vícenásobné násobení matic s vektorem, přichází vektorová varianta problému s vlastními, nepřenositelnými formáty. Díky tomu, že jedna ze stran má pouze jeden sloupec/řádek, je možné matici přizpůsobit na míru tomuto vektoru, což je něco, co se nedá v obecném násobení použít.

Navzdory jejich nepoužitelnosti pro násobení dvou matic zde budou zmíněny pro kompletní přehled formátů. Zároveň mohou být zdrojem inspirace pro generické formáty.

2.7.1 Ellpack

Jedním ze specializovaných formátů je Ellpack. Základní ideou je ukládat pouze nenulové prvky a uchovávat jejich sloupcové indexy. Vznikají tak dvě dvourozměrná pole, jedno pro sloupcové indexy, druhé pro hodnoty.

Tento způsob ukládání má jednu vadu, a tím je nemožnost adresovat jednotlivé řádky a číst celou matici nerušeně po sloupcích. Proto se tady krátké řádky doplňují zprava nulami na délku nejdelšího řádku. Kvůli tomu může být formát neefektivní, a jeho další iterace se snaží tento problém mírnit zarovnáváním jen několika řádků v bloku a v některých případech navíc seřazením řádků, pro podporu této myšlenky zarovnávání v rámci bloku. [17]

2.7.2 AMB

Algoritmus *nsparse* přichází ve dvou variantách – jedna slouží pro násobení matice vektorem, druhá je využívána pro obecné násobení dvou matic.

Hlavní rozdíl, který lze u vektorového přístupu pozorovat, je výběr formátu. Místo CSR se používá formátu AMB (Adaptive Multi-level Blocking), který vychází z formátu Ellpack 2.7.1. Ten byl ještě předtím, než autoři algoritmu tento formát definovat, několikrát upraven tak, aby se zmenšil počet nenulových prvků a s tím související vyšší paměťová náročnost.

Jednou z takových úprav je možnost zkrátit řádky. Místo toho, aby se zarovnávalo podle nejdelšího řádku, bude se zarovnávat podle největšího řádku v daném bloku. Velikost bloku je daná parametrem, který se předem určí. Aby bylo možné stále přistupovat k jakémukoliv řádku, je nutné ukládat indexy na začátky jednotlivých bloků. Podle indexů a počtu řádků v bloku jde určit délku řádků. Tento formát se nazývá *SELL-C*, kde C je argument, pod kterým se skrývá právě počet řádků v jedné sekci. [17, 18]

V AMB se matice nejdříve „rozpůlí“ – vznikají dvě matice se stejným počtem řádků. Těmto dvěma polovinám se seřadí řádky podle počtu nenulových prvků v nich. Seřazení pomáhá snížit paměťovou náročnost – v dalším kroku se hodnoty uloží stejným způsobem, jako v SELL-C – a díky seřazení bude počet nulových prvků sloužící jako pouhé zarovnání bude nižší, než v neupraveném případě.

Díky půlení lze sloupcové indexy ukládat v menším datovém formátu. Zde již nejde o celé indexy, ale o zbytky po dělení. Podobné tvrzení platí o indexech řádků. Neřadí se totiž celá matice, ale jen jasně definovaný blok, tudíž opět nejde o indexy, ale o permutaci řádků – pro index je nutné permutaci přenásobit číslem bloku.

Se znalostí délky řádků není nutné ukládat všechny sloupcové indexy, sousedící jde zkomprimovat. Řádky se proto přestávají doplňovat pouze zprava, zachovává se původní struktura a lze proto tvrdit, že existuje sousední hodnota. Tato úspora není tak významná, protože se vše nejdříve seřadilo a řádky v blocích nejsou příliš dlouhé. Jde tak o další referenci, podle které se dá rozhodnout, jak velké bloky pro uložení matice zvolit. [18]

Kapitola 3

Způsoby násobení matice

K maticovému násobení jde přistupovat několika způsoby. Tím nejpřirozenějším je násobení po jednotlivých buňkách:

$C_ij = \sum_{k=1}^{n} A_ik * B_{kj}$

Další variantou je výsledek skládat po řádcích:

$C_{*}j = \sum_{k=1}^{n} A_{*}k * B_{kj}$

Tento způsob je v algoritmech hojně využíván [5]. Je pro to hned několik důvodů – efektivně ignoruje nulové prvky, počítání řádků je na sobě zcela nezávislé, a akumulace hodnot má velmi dobrou lokalitu dat. [19]

Tento přístup má mnoho kladů, není možné jej ale přímočaře použít. Problémem je neznalost velikosti výsledné matice, natož její struktury.

Každý algoritmus musí nějakým způsobem tento problém vyřešit. Jsou tři hlavní způsoby, jak se k tomuto postavit. Žádný z nich není věrný základní myšlence, kdy se počítá právě jedna konkrétní buňka. Pokaždé se jich vypočítá větší množství, což klade nároky na efektivní náhodný přístup k jednotlivým mezivýsledkům a případný souběh při vícevláknovém zpracovávání.

3.1 ESC

Pod těmito třemi písmeny se skrývají tři anglická slova vysvětlující fungování algoritmu:

- E – Expand
- S – Sort
- C – Compress

Ve své základní podobě jde o myšlenku velmi úzce spojenou s formátem COO 2.1. V prvním kroku se provádí veškeré násobení. Neprovádí se žádná akumulace nad jednou položkou, jak je běžné u naivního $O(n^3)$ algoritmu, ale postupně se vybírají položky z matice A a vynásobí se s hodnotami na odpovídajících indexech v matici B.

Předchozí způsob byl přímočarý, avšak tímto způsobem nelze dosáhnout rekonstrukce matice. Je proto na čase přijít s druhým krokem, kterým je řazení. Mezivýsledky se seřadí podle svého řádkového a sloupcového indexu. Cílem zařídit, aby položky se stejnou dvojicí indexů byly u sebe.

Díky tomu může dojít k poslednímu kroku, kompresi. Stejná dvojice identifikátorů znamená, že hodnoty by se měly sečíst, protože patří do stejné buňky v kýžené matici.

3.2 Merge

Myšlenka řazení, která se objevila u algoritmu ESC 3.1, není neobvyklá. Podobnou cestou se vydává i druhá zmiňovaná metodika, která se inspiruje algoritmem *Mergesort*.

Zde je lepší na algoritmus nahlížet jako na m násobení matice vektorem, kde m je počet řádků matice A.

Na vektor a je dobré nahlížet pouze jako na věc co má index a nějakou hodnotu. Indexem se vybere řádek v matici B, a každá hodnota se jím přeškáluje. Výsledkem je n vektorů s rozdílnou délkou, na základě délky řádku v matici B, kde n je počet nenulových prvků ve vektoru a.

Nyní, s několika vektory, bude docházet ke slévání vektorů. Položky se stejným indexem se sečtou a zároveň se každá hodnota správně seřadí. Tímto způsobem vznikl jeden řádek výsledné matice. [4]

3.3 Akumulátory

Poslední kategorie je nejvíce věrná standardní myšlence násobení matic. V jeho pozadí se neskrývá žádný řadicí algoritmus, místo toho se pracuje s nějakou variantou akumulátoru, do kterého se mezivýsledky, které povedou k tvorbě celého vektoru, ukládají. Podobně jako u Merge 3.2 základní jednotkou pro jeden krok výpočtu je vektor.

Existuje několik typů akumulátorů. Tím nejrozšířenějším je hashovací tabulka. Před vložením/přičtení hodnoty na daný index se vypočítá jeho hash. Pokud se na dané pozici nachází hodnota s jiným indexem, najde se pro ni jiné umístění, například prostým přičtením jedničky, nebo opětovným použitím hashovací funkce. [5] Velikost tabulky závisí na implementaci, nejjednodušší je použít počet násobení, které bude v jednom kroku provedeno – sečtou se nenulové řádky matice, na které ukazují indexy ve vektoru.

Atypickou variantou této hashovací tabulky je hashovací vektor, *HashVector*, který využívá specifických vlastnostní architektury procesoru. Místo toho, aby se pracovalo s jedním polem s daty, dochází zde k rozdělení do bloků, nad kterými lze provádět rychle operace detekce výskytu prvku. Co se týká alokované paměti, hodnota by měla zůstat stejné, jako v případě hashovací tabulky. V obou případech [5] volí mocniny dvojky, a na jimi zvolené architektuře je vektor schopný uchovat osm položek, tedy dvě na třetí.

Pro některé problémy, například AMG, u kterého je jedna z matic úzká, se nemusí používat hashování. Lze používat vektor tak dlouhý, jako je šířka menší z matic. Tento přístup se nazývá perfektní hashování a díky tomu nikdy nebude docházet k hledání volného místa, kam nový mezivýsledek při potenciálním konfliktu uložit. Problémem je jeho omezené využití a s tím související znalost domény. [20]

Poslední typ využívá prioritní fronty-haldy. Protože se pracuje s haldou, vystupuje zde logaritmická časová složitost místo optimistické konstantní v případě hashování. Jde o kompaktnější variantu hashovací tabulky, kdy velikost haldy není určena počtem násobení, které bude provedeno, ale pouze počtem nenulových prvků ve vybraném vektoru. [21, 20, 5] Pokud by se ale akumulace prováděla stejným způsobem, jako tomu bylo u hashování, paměťová náročnost by byla stejná. Proto se zde volí o něco složitější postup. Každá hodnota z vektoru ukazuje na některý z řádků matice. Pro inicializaci se vynásobí první hodnota z řádku matice s hodnotou na dané pozici ve vektoru a vloží se do haldy. Pamatujeme si hodnotu, sloupcový index a řádek, ze kterého součin vznikl. Poté se, dokud je to možné, střídají operace odebírání z fronty a vkládání nové položky. Odebírání je přímočaré, vkládání záleží na odebraném prvku. Provede se další výpočet ve stejném řádku, který byl zmíněn u právě odebraného prvku. Pokud již nejde dále násobit odeberou se všechny položky z fronty. [22]

Tento způsob podobně jako perfektní hashování respektuje sloupcové indexy a nezmění jejich pořadí ve výsledné řídké matici. Obecné hashování jako jediné tuto vlastnost nemá.

V haldě by mohlo docházet k akumulaci, kdy by se položky se stejným sloupcovým indexem

sečetly. Algoritmus v práci zmíněný [22] tuto možnost nepopisuje, tedy ta opravdová akumulace se provádí až po odebrání z haldy. Ta pouze zaručuje, že se bude držet správná posloupnost. Spadá proto spíše do kategorie řadicích algoritmů.

Kapitola 4

Způsoby alokace paměti

Práce s pamětí je jedno z významných témat, které se musí u násobení řešit. Alokovat hustou matici bude ve spoustě případů nemožné pro svou velikost (citation needed), musí se proto volit způsoby, které provádějí alokaci skromněji.

Jsou celkem čtyři hlavní proudy, jak s touto problematikou pracovat. Některé způsoby lze kombinovat. Liší se od sebe časovou náročností a množstvím alokované paměti. Objevují se tu oba extrémy, vedoucí k postupným realokacím, nebo podobně jako u příkladu s hustou maticí k nereálnému požadavku.

4.1 Přesná alokace

Jak již název naznačuje, cílem je přesně určit velikost výsledné matice. Výpočet se provádí nad zjednodušeným vstupem, místo nenulových desetinných čísel se používají příznaky (booleans) [23].

Tento postup se v literatuře nazývá symbolickou fází [5]. Jde o myšlenku, které je k vidění například u MKL či cuSPARSE [23]. V RMerge se spočítají délky řádků a díky prefixovému součtu se vypočítají začátky řádků pro formát komprimovaných řádků [4].

Postup je časově poměrně náročný, protože se stejná myšlenka – násobení dvou matic – provádí dvakrát.

4.2 Progresivní metoda

Tento způsob je jediným ze zmiňovaných, který je ve své implementaci smířený s tím, že bude třeba zvětšovat velikost alokované paměti.

Jsou operace, u kterých lze počet nenul v matice jasně určit; jde například o Choleského dekompozici [24].

Vše začíná nějakým základním odhadem velikosti. Toto záleží na implementaci algoritmu – lze použít odhadu za použití pravděpodobnosti či případných dalších znalostech řešeného problému [25, 24]. Není nutné odhadovat velikost celé matice najednou, většina algoritmů pracuje po menších částech. Alokovaným blokem bude proto jen jeden řádek/sloupec, který se v případě, že bude příliš malý, rozšíří. Provede se alokace nového bloku, obvykle větší o nějaký malý násobek (například 1,5 předchozího bloku), a data se překopírují [24].

Jde o myšlenku, která přistupuje citlivě k alokování, avšak za poměně vysokou cenu v podobě nižšího výkonu, který je způsoben realokacemi [23].

4.3 Horní odhad

Tato myšlenka má velmi blízko k naivní alokaci paměti, jen se místo alokování celé husté matice pracuje se součinem počtu nenulových prvků v obou maticích. Je náchylný ke stejnému neduhu jako již zmíněná naivní práce s pamětí. Pokud nenastanou žádné problémy, alokace tímto optimistickým způsobem povede k nejrychlejšímu běhu programu. Problém nastává, pokud je odhadované množství tak velké, že se nevejde do hlavní paměti a negativně tak ovlivnit celkový čas. Zcela se ignoruje možnost „vynulování prvků“, kdy součet součinů nenulových prvků skončí nulou, a nebude ve výsledné matici. [23]

Podobnou myšlenku používají během svého běhu algoritmy založené na myšlence ESC 3.1 [23], kdy každý součin má své místo v paměti. Ne vždy ale pracují s tímto horním odhadem, aby se obešel problém s alokací příliš velké paměti se pracuje s bloky s jasně danou velikostí [26].

4.4 Pravděpodobnostní metoda

Práce s pravděpodobností se objevuje hned v několika oblastech okolo násobení matic. Práce z roku 1998 odhaduje strukturu matice, počet nenulových prvků ve sloupcích matice či výhodné pořadí operací násobení sekvence matic [25]. Nejde o první práci na toto téma od tohoto autora.

Práce ukazuje několik způsobů, jak odhadnout velikost matice. Liší se od sebe svou komplikovaností, předpoklady a možnostmi. Zmíněn tu bude podrobně ten nejjednodušší, další pouze rámcově.

Jde o naivní odhad, který lze vypočítat v konstantním čase. Pracuje s předpokladem, že nenulové prvky jsou v matici rozmístěny náhodně a nezávisle. Výsledkem je $\frac{a*b}{n}$, kde a a b jsou počty nenulových prvků v levé, respektive pravé matici. V podílu se nachází „společná“ strana matice, tedy šířka levé strany a výška pravé strany.

Další varianta rozšiřuje naivní postup a pracuje o poznání jemněji, počítá se pravděpodobnost, se kterou bude prvek na dané pozici nenulový. Vzorec, ke kterému došli, má velkou časovou náročnost, a tak se výpočet pravděpodobnosti pouze odhaduje podobným způsobem, jakým se počítala pravděpodobnost u naivního odhadu ($(n, r, s)min\{1, \frac{r*s}{n}\}$). r značí počet nenulových hodnot v řádku i matice A, s nenulové hodnoty ve sloupci j matice B. [25]

$$p_{ij} \equiv \text{Prob}\{[A]_{ij} \neq 0\} = p(n, r, s) = 1 - \frac{(n-r)!(n-s)!}{(n-r-s)!n!}$$

Kapitola 5
Existující algoritmy

V této kapitole budou zmíněny všechny novější algoritmy, které se označují jako varianty, které jsou mnohem lepší než ty, které lze nazvat tím nejjednodušším, co lze nalézt. Jde o knihovní funkce vše CUDA, Intel MKL.

Kapitoly budou seřazeny chronologicky, protože je běžné, že jsou některé kroky argumentovány algoritmy předchozími. Podobné tvrzení jde uplatnit u porovnávání výkonnosti, zde rovněž platí, že je lepší řadit podle data vydání, a ne podle použitých technologií a kategorií, do kterých algoritmus spadá.

Cílem každého z nich je jakýmkoliv způsobem se zhostit problematiky alokace paměti, která byla zmíněna dříve v samostatné kapitole 4, a pokusit se co nejlépe rozložit práce mezi jednotlivá vlákna.

5.1 RMerge

Základní myšlenkou za tímto algoritmem je technika *merge* 3.2. Úprava alogitmu je provedena tak, aby byla výhodná pro SIMD architektury, v některých oblastech volící neefektivní výpočty na místo používání datových struktur. Písmeno *R* je zkratkou pro *row*, slévají se k sobě výsledky přeškálovaných řádků.

Algoritmus se skládá ze tří fází. V první se zjistí délky řádků výsledné matice. Na základě toho lze dojít k druhé fázi, alokace paměti a vypočítání začátků jednotlivých řádků ve formátu CSR. Posledním krokem je samotný výpočet, který odpovídá prvnímu kroku, jde tedy o algoritmus provádějící přesnou alokaci, provádějící numerickou a symbolickou fázi.

Aby byla zátěž mezi vlákny v algoritmu dobře rozložena, dojde k úpravě levé strany. Vzhledem k tomu, jak funguje slévání, počet nenulových prvků ve vektoru ovlivňuje, kolik polí se bude nakonec slévat do výsledku, do jednoho řádku matice *C*. Úprava spočívá v rozložení matice na dvě, A1 a A2 tak, aby A1 * A2 = A. Cílem je matice na levé straně upravovat tak, dokud délka řádků nebude nižší než námi vybraná horní hranice. Výsledkem tohoto rozkladu jsou jednoduché matice – A2 obsahuje stejné hodnoty, ve stejných sloupcích, jen se zvýšil počet řádků. Za každý delší řádek vzniká několik dalších řádků. Počet hodnot v řádku se vydělí kýženým cílovým počtem nenulových hodnot; horní celá část z tohoto čísla vede k počtu řádků. Matice A1 má v této první fázi dva řádky, hodnotou je buď jednička, nebo nula. V této fázi se používá rekurze, matice A1 totiž může trpět stejným problémem jako původní A, a dělba tak může pokračovat.

Délka řádků je silně ovlivněna hardwarovými omezeními, souvisí se sub-warpy na GPU, proto jsou v práci [4] zmíněny mocniny dvojky od čísla dva po třicet dva.

Samotné násobení se provádí zprava, od první rozložené s nepozměněnou *B*. Každé vlákno přeškáluje svůj odppovídající řádek na pravé straně. Pak dojde k paralelnímu výběru minima,

aby se zjistil nejmenší sloupcový index. Pak každé vlákno vrátí hodnotu, kterou má na tomto indexu, nebo nulu. Nad těmito čísly se provede suma a tato hodnota se uloží do matice C. Toto je zdroj neefektivity, kterou zmiňují i autoři, ale GPU je pro tyto numerické operace mnohem lépe připraveno, než pro práci se strukturami.

Každé další násobení by mělo být jednodušší, protože jde pouze o úpravu formátu a správné uspořádání hodnot do výsledné matice C. Matice se postupně zmenšují, a k násobení nemusí docházet, násobí se jedničkou. Kvůli tomu, jak se matice změnšují, se omezují možnosti, jak vlákna zaměstnat.

5.2 bhSPARSE

Autoři algoritmu si dali za cíl celkem tři body. Všechny z nich jsou poměrně přímočaré, patří sem snaha o správnou alokaci paměti, úspornější metoda pro vkládání prvku (v porovnání s naivní ideou), a load balancing.

V jejich podání jde o jednofázový hybridní algoritmus, skládající se ze čtyř fází. Každá z nich nějakým způsobem buď míří k řešení problémů, na které byla práce zaměřena, nebo je přímo adresuje.

První fáze vypočítává maximální počet nenulových prvků v jednotlivých řádcích matice C. Jde o sumu násobení, které se bude muset provést. Tento výpočet autoři provádí pomocí GPU.

Tyto výsledky, horní odhady, se použijí v další fázi, která řádky kategorizuje podle jejich náročnosti. Tato část se pro změnu provádí na CPU, jde pouze o porovnání jedné hodnoty a vložení indexu do pole. Cílem této kategorizace je určit, jakým způsobem bude který řádek zpracován. Každá buňka, *bin*, patří do nějaké sady a nějakým způsobem určuje náročnost každého řádku. Do první sady spadají prázdné řádky. Kvůli těmto není nutné alokovat pamět, nic počítat. Další v pořadí musí provést právě jednu operaci. Toto je podobný případ jako první sada, jen je nutné pamatovat na alokaci paměti o velikosti jedna.

Ani u jednoho z výše uvedených případů není nutné počítat, nebudou proto ani kopírovány na GPU. Další skupiny již budou na kartě zpracovávány a kategorizace plynou z hardwarových omezení. Autoři pracovali s nVidia, která měla *thread bunch* velikosti třicet dva. Proto je další skupina v rozsahu dva až třicet dva. Jde tak o řádky, které je nutné pronásobit, ale jsou stále příliš triviální na to, aby byly efektivně zpracovány pomocí *thread bloku* GPU.

Řádky spadající do další skupiny jsou již dostatečně velké na to, aby zatížily celý thread blok, jsou ale stále dostatečně malé na to, aby se celé vešly do lokální paměti. Dalším společným jmenovatelem je jednoduchý odhad na výslednou paměť – nepůjde o číslo větší, než jakým je horní odhad, a ten není tak velký, aby způsobil problémy při alokaci paměti.

Poslední buňky již nejsou shora omezené, jde o všechny, které byly pro předchozí příliš velké. Používá se jeden celý thread blok, neliší se proto od druhé skupiny, ale jsou tu problémy s pamětí. Použití předchozí strategie by mohlo vést k alokaci až příliš velkého bloku paměti.

Další fáze se zabývá výpočtem matice. Hodnoty z prvních dvou skupin se jednoduše překopírují, přeškálují. Pro další již je třeba použít GPU. Každá z nich používá rozdílnou strategii. Postupně, podle velikosti skupiny, se používá akumulátorová metoda s haldou, ESC, metoda slévání. V metodě s haldou každé vlákno zpracovává právě jeden řádek, nepracuje na každém z nich celý thread blok. Poslední metoda jako jediná pracuje s progresivní metodou alokace paměti, autoři začínají s kapacitou dvě stě padesát šest prvků, v případě, že tato hodnota je nedostatečná, zdvojnásobí se.

Nakonec se všechny hodnoty překopírují do výsledné matice C. Hodnoty pocházející z druhé skupiny, s právě jednou hodnotou, se překopírují pomocí jednoho vlákna. Pro ostatní netriviální se použije celý thread blok. [23]

Výsledkem snahy o vyřešení všech neduhů, které naivní implementace má, je program, který využívá všech základních druhů způsobů, jak k násobení přistupovat, a tří způsobů, jak alokovat pamět. Jediný vynechaný je pravděpodobnostní, progresivní metoda pracuje s fixní startovní hodnotou daná velikostí předchozí skupiny.

5.3 nsparse

Tento algoritmus vznikal v době, kdy ostatní zrychlování dosahovali použitím velkého množství paměti. Na tuto problematiku se zaměřují autoři nsparse, a plyne z toho algoritmus, jehož hlavní výhodou je právě paměťová střídmost. V práci jako protipříklad paměťové efektivity je zmiňován přístup ESC 3.1, který musí alokovat místo pro všechny mezivýpočty.

Výsledkem je dvoufázový akumulátorový algoritmus, používající pro ukládání mezivýsledků hashovací tabulky.

Podobně jako bhSPARSE 5.2, i zde se používá kategorizace řádků matice A.

Ke kategorizování dochází celkem dvakrát, před symbolickou fází se řádky umisťují do skupin podle počtu násobení. Po provedení symbolické fáze již není třeba používat tohoto hrubého odhadu a lze použít skutečný počet nenulových prvků na jednotlivých řádcích výsledné matice C.

Kategorie nerozhodují o použitých algoritmech jako tomu je u bhSPARSE, pouze se omezují prostředky, se kterými daná skupina může pracovat. Buď může být použit celý thread blok, nebo jen subwarp o čtyřech vláknech. Číslo čtyři je empiricky změřená hodnota, autoři díky tomuto počtu docházeli k nejlepším výsledkům. Dalšími zmiňovanými možnostmi jsou mocniny dvojky v rozsahu jedna až šestnáct.

Algoritmus je rozdělen do dvou částí, velmi silně ovlivněn dvoufázovým přístupem k počítání matice. Nejdříve se zjistí, kolik násobení je nutné provést pro spočítání jednotlivých řádků C. Na základě těchto hodnot se provede první kategorizace a může se přejít k symbolické fázi.

Zde se vypočítá, kolik hodnot se bude muset do výsledné matice uložit. Do tabulky stačí ukládat sloupcové indexy, násobení se neprovádí a buňka je neměnná – buď obsahuje sloupcový index, nebo hodnotu například mínus jedna, značící volné pole. Protože v rámci jednoho řádku pracuje větší počet vláken, je možné, že při operaci vkládání dojde ke konfliktům. Musí se proto použít atomického přístupu. Pro výpočet hash se použije sloupcového indexu, který se přenásobí konstantně zvolenou hodnotou. V hashovací tabulce se používá nejjednoduššího způsobu, jak konflikty řešit. Vypočítaná hash se pouze zvětší o jedna a pokus o vložení prvku se opakuje.

Díky těmto součtům je možné zpřesnit kategorizaci řádků A, správně naalokovat pamět pro výslednou matici a díky paralelnímu prefixovému součtu nastavit indexy symbolizující začátky jednotlivých řádků.

Část zabývající se numerickou fází má před sebou tři dílčí úkoly. První z nich odpovídá symbolické fázi, jen je zde třeba ještě pracovat s dalším polem, do kterého se ukládají výsledky. Druhým úkolem je z hashovací tabulky získat nenulové hodnoty. A na konec je každému vláknu přiřazena nenulová hodnota. Cílem vlákna je zjistit pozici ve výsledné matici, jde tedy o porovnání sloupcového indexu se všemi ostatními prvky.

Kategorie byly utvořeny tak, aby odpovídaly hardwarovým omezením. Například poslední kategorie, která není shora omezena, používá pro tabulku vždy globální paměť. Další kategorie je utvořena tak, aby šlo pracovat kompletně se sdílenou pamětí. A pak se postupně počet používaných thread bloků zvyšuje, velikost thread bloku snižuje. Protože opearce modulo je náročná, provádí se místo ni bitový posun. Kvůli tomu jsou rozměry tabulky vždy mocninou dvojky. [27]

5.4 AC-SpGEMM

Jde o jednofázový algoritmus navržený primárně pro grafické karty. Spadá do kategorie ESC, ale protože naivní myšlenka zcela ignoruje problémy spojené s alokací paměti či možné paralelní zpracování, dochází zde k řadě vylepšení.

Některá programovací rozhodnutí plynou ze znalosti řešených problémů, například průměrný počet nenulových prvků v Sparse Matrix Collection [2] je dvě stě. Dohromady s menšími Thread bloky je možné uchovávat celkem asi čtyři tisíce mezivýpočtů se v ideálním případě vejde celý výsledný řádek do lokální paměti.

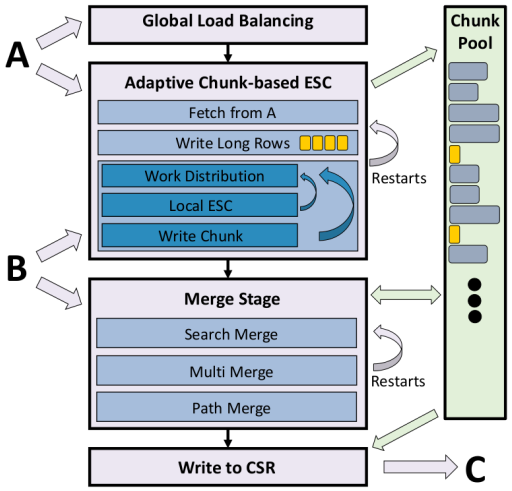
Vyvažování zátěže bývá v algoritmech náročné. Například bhSPARSE (viz sekce 5.2) řádky kategorizuje podle počtu nenulových řádků, nsparse (viz sekce 5.3) analyzuje řádky podle počtu operací násobení, které se musí provést. Zde se jde jinou cestou, mnohem jednodušší, každému bloku se přiřadí fixní počet hodnot z matice A. Nejnáročnější operací je pak přiřazení řádku z B ke každé hodnotě, což je v porovnání s kroky prováděnými u algoritmů výše zanedbatelné. Nevýhodou je nižší kvalita rozložení.

Při tvoření této vazby se navíc kontroluje délka řádku B. Pokud je příliš dlouhý, a mohl by při zpracovávání způsobovat problémy (zbytečné rozkouskovávání, aby se vešel do omezené lokální paměti), vyjme se z výpočtu a vytvoří vlastní blok, ve kterém dojde k přeškálování hodnotou z A.

Každý blok provádí nad svými hodnotami A ESC (viz sekce 3.1). Nepracuje se zde se všemi hodnotami zároveň, či podle řádků; struktura je zde volnější a cílem je pracovat pouze s lokální pamětí, ignorují se zde ve většině případů vazby na řádky. Po té, co se dokončí celý jeden řádek v lokální paměti, nebo bude paměť celá zaplněná, odešle se jako jedna velká sada do globální paměti. Odsud pochází první dvě písmena algoritmu, AC značí Adaptive Chunk, a *chunk* je právě ten kus zpracovaných dat na daném řádkovém indexu, který je odeslán dál. Operací ESC může být v rámci bloku prováděno více, vše záleží na počtu hodnot v matici B a řádkových indexech hodnot matice A. Vědět, jaké mezivýpočty do lokální paměti poslat jsou tím nejdůležitějším, čím se algoritmus musí zabývat. Toto je daní za hrubé vyvažování zátěže na globální úrovni. Zde jde ale o něco, čemu se žádný algoritmus vyhnout nemůže – hodnoty z B musí být načtěny vždy, teď se pouze na jejich podmnožině určí, které to budou.

Řazení v ESC je prováděno Radix sortem, proto je výhodné pracovat s co nejkratšími indexy. Zde se ale kombinují řádkové a sloupcové indexy, proto omezenování indexů není přímočaré. Řádkové nezpůsobují velký problém, ty jsou unikátní napříč blokem, ale u sloupcových se musí pracovat s neúplnou znalostí, omezí se na interval nejmenšího a největšího indexu z B.

Kusy v globální paměti jsou seřazeny podle pořadí, ve kterém vznikly. Protože každý kus nemusí reprezentovat jeden jediný řádek, dojde k jejich slévání. Jsou různé způsoby, jak k jejich spojení dojít, liší se od sebe zda umí pracovat s jakýmkoliv počtem bloků, nebo zda pouze se dvěma. Cílem je najít průnik indexů a nad hodnotami v tomto intervalu provádět spojení.



Obrázek 5.1 Fáze algoritmu ACSpGEMM

Poté je možné všechna data zkopírovat do výsledné řídké matice. Během vytváření bloků a slévání se zjistila délka všech řádků, pomocí paralelního prefixového součtu se zjistí začátky jednotlivých řádků a vlákna překopírují své bloky na dané místo výsledné matice C.

Algoritmus až na detekci dlouhých řádků, různé strategie při spojování bloků podle jejich počtu je podobně jako RMerge věrný své kategorii. [26]

5.5 spECK

Za touto zkratkou se skrývá poměrně dlouhý název – *SpGEMM achieving Efficient Computation for all Kinds of matrices*. Adresují neduhy, které mají algoritmy předchozí, jako například již zmiňované AC-SpGEMM 5.4, nsparse 5.3, bhsparse 5.2.

Snahou je vytvořit algoritmus, který bude provádět přípravu na násobení, ale nebude tak náročná, aby se na ni strávila velká část celkového výpočtu, a pozmění se způsob, jakým se práce rozděluje mezi bloky a vlákna GPU.

Algoritmus se skládá z šesti částí. Cílem prvních čtyř je nasbírat dostatek informací o řešeném problému tak, aby se zátěž mezi vlákna dala dobře rozložit. Začíná se analýzou řádků. Nejde zde pouze o zkoumání počtu násobení, zjišťuje se navíc maximální délky řádků, počet násobení u nejdelších řádků, rozsah sloupcových indexů u jednotlivých řádků. Všechny tyto statistiky jsou vztaženy k jednotlivým řádkům matice A.

Díky těmto statistikám jde rozhodnout, zda je třeba provádět globální optimalizaci. Pod touto frází se skrývá možné rozdělení řádků do kategorií, jako tomu bylo u nsparse 5.3. Stejným způsobem se tvoří kategorie, zlomovým bodem je velikost sdílené paměti na grafické kartě. Práce s kategoriemi se ale provádí trošku jinak, proaktivně se shlukují řádky tak, aby odpovídaly velikosti sdílené paměti. Do jisté míry se proto postup otočil a thread blok si sám sobě přiřazuje řádky. Jde o hladový způsob, kdy se snaží přiřazovat sousední řádky do limitu hashovací tabulky. Nedochází k přeuspořádání řádků podle délky, může tedy trpět na případy, kdy například dva řádky vedle sebe vyžadují hashovací tabulku jen lehce větší, než je polovina celkové kapacity paměti.

Protože tato kategorizace není nejlevnější operací, pro malé matice, či matice s rovnoměrným rozložením prvků, se přeskakuje.

Další tři části budou spolu velmi úzce souviset a bude řádně popsána pouze jedna z nich. Jde o symbolický a numerický výpočet matice a mezi těmito dvěma výpočty je umístěna případná globální optimalizace. Ta díky předchozímu numerickému výpočtu může pracovat s větším množstvím informací.

Násobení je do jisté míry inspirováno algoritmem bhsparse 5.2. Podle struktury se vybere způsob, jakým se bude počítat. Výběr je tu omezen na některou z akumulátorových variant – hashovací tabulka, perfektní hashování jako hustý vektor, a prosté překopírování dat v případě, že matice A má pouze jeden nenulový prvek.

Globální optimalizace přiřadila řádky thread blokům. Na úrovni bloků se dále provádí lokální load balancing. Vlákna se rozdělí do skupin a těm se rovnoměrně přiřadí nenulové prvky A. Protože doba běhu pak silně závisí na pravé straně, přiřazování dále respektuje průměrnou délkou řádku B. Výjimečně dlouhé řádky tento odhad mohou špatně vychýlit, použije se ještě jedna heuristika, která dává do souvislosti počet zpracovávaných řádků a maximální počet iterací, které by každé vlákno mohlo být donuceno provádět. Počet iterací je podílem nejdelšího řádku a počtu vláken ve skupině. Pokud je poměr počet iterací a počet zpracovávaných řádků příliš velký, upraví se počet skupin. Pracuje se s tím předpokladem, že práce na velkém množství řádků je zhruba stejně náročná, jako zpracovávání dlouhého řádku a zde je pouze snaha tyto dvě hodnoty dostat do vhodného intervalu.

Pro řídké řádky výsledné matice se použije hashovací tabulky s otevřeným adresováním a *linear probing*. Protože jedna skupin může pracovat s větším množstvím řádků, výpočet indexu do hashovací tabulky je o něco náročnější, musí se v těchto případech navíc zapracovat do indexu řádek A. Jde o skládaný klíč, pět bitů je vyhrazeno řádku, zbytek sloupcovému indexu B. Pro velké řádky je lepší použít celého pole, bez hashování. Tato struktura má několik výhod, například je kompaktní – není třeba evidovat sloupcové indexy. S tím souvisí absence kolizí a ten

fakt, že není nutné položky řadit do výsledné matice. Posledním případem je triviální přenásobení hodnot, pokud řádek v levé matici obsahuje pouze jeden nenulový prvek.

V posledním kroku stačí do matice C, vytvořené podle výsledků ze symbolické fáze, seřadit hodnoty, které vznikly za pomoci hashování. [28]

5.6 OpSparse

Další popisovaným algoritmem bude OpSparse. Za jeho zrodem stojí oprava sedmi neefektivností, které byly objeveny u nsparse 5.3 a spECK 5.5. Jde tedy o podobnou vizi, způsob, jak vytvářet nový algoritmus, kterou razil již zmiňovaný spECK 5.5.

Algoritmus jde rozdělit do pěti částí. Autoři uvádí počet o jedna větší, ale protože tou poslední zmiňovanou je úklid prostředku, což je krok, který musí provádět každý, budu jej ignorovat.

Jednotlivé kroky dále silně připomínají předchozí algoritmy. Za prvé je nutné spočítat, kolikrát se bude násobit na kterém řádku, díky čemuž jde pak v další fázích provádět kategorizaci všech řádků.

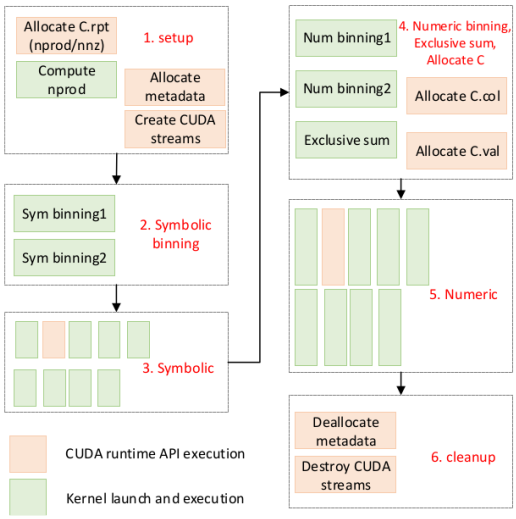
Stejně jako u algoritmů, kterými se autoři inspirovali, i zde se matice počítá dvakrát, v symbolické, a numerické fázi. Pro každou z těchto fází se provádí zvlášť kategorizace, liší se pouze používanou proměnou, symbolická fáze používá počet násobení, numerická počet nenulových prvků. Používá se k tomu pouze sdílené paměti, díky čemuž používání atomických operací není tak drahé. Evidence toho, kdo patří do své kategorie, využívá tří polí. Struktura připomíná CSR, ale protože není potřeba sloupcových indexů, je formát velmi kompaktní. Dvě z nich jsou dlouhé jako počet kategorií, autoři pracovali s osmi. Jedno z nich udává počet řádků v jednotlivých kategoriích, druhé offset, kde začínají prvky kategorií. Jde o exkluzivní prefixový součet pole předchozího. Třetí pole obsahuje indexy jednotlivých řádků.

Plnění této strukturu se provádí ve dvou krocích. Nejdříve je nutné zjistit, jak velké jsou jednotlivé skupiny. Během toho se také kontroluje maximální délka řádku. Pokud by se stalo, že všechno by spadlo do první kategorie pro nejkratší řádky, je veškerá kategorizace zbytečná a druhá fáze se dá zjednodušit. V případě většího rozptylu hodnot pak v druhé fázi dochází k umístění řádků na správná místa pole. Díky počtu nenulových hodnot a vypočítanému offsetu je tento krok přímočarý. Velký důraz se zde klade právě na omezení paměti využívané na těchto pomocných výpočtech. Ty navíc v průběhu výpočtu ztrátí význam, a je tak možné je znovu použít později. Pro kategorizace v numerické fázi jde použít stejná pole ze symbolické fáze, a ještě předtím se alokovala paměť uchovávající počet násobení v jednotlivých řádcích, což je podstatné jen pro symbolickou fázi.

Pro hashování se používá otevřené adresace s lineárním řešením konfliktů. Je zde jedna malá optimalizace pro omezení přístupu do pole. Indexy a hodnoty jsou ve svých polích, a pro přístup do pole s indexy se používá operace *compare-and-swap*. Ta spojuje tři operace do jedné; zde se využijí dvě její vlastnosti – oznámí, zda jde do buňky vložit/přičíst, a zároveň sloupcový index nastaví, což je operace, která se standardně provádí v případě, kdy je pole volné. Dá se proto tvrdit, že počet instrukcí nad hashovací tabulkou může být až o třetinu menší. nsparse 5.3 místo operace modulo používal bitový posun, zde pro dosažení stejného cíle používají logický AND. V numerické fázi se ale této myšlenky nepoužívá, nezaokrouhluje se na mocniny dvojky, a modulo je tak stále v kódu k nalezení. Toto rozhodnutí bylo provedeno ve prospěch lepšího zatížení jednotlivých *thread bloků*. Hashovacích tabulek na jednom bloku může být více a mohou být různě dlouhé. Tabulka by měla být v numerické fázi alespoň dvakrát větší, než je počet násobení, které bude provádět. Dvojnásobek je empiricky zjištěná hodnota, experimentovalo se i se zvětšením o padesát a tři sta procent. Kvůli této volnější tvorbě hashovacích tabulek dochází k horšímu využívání hardware. Na druhou stranu zde není žádná fixní velikost, ke které se rozměry tabulky zaokrouhlují, a je tak možné mít počet konfliktů částečně pod kontrolou.

V pětifázovém plánu jsou některé kroky, které nejsou závislé na globální paměti. Díky tomu je možné některé kernely spustit, a zároveň alokovat pamt. Ihned v první části algoritmu zde tehoto postupu využít; lze zjistit, kolik násobení se bude celkem provádět, a také alokovat paměť

pro pomocné výpočty. Při práci s hashovací tabulkou se kvůli některým řádkům musí alokovat globální paměť. Předtím je ale možné spustit kategorie řešící menší problémy, díky čemuž se čas, během kterého se alokuje, využije k výpočtům. Ke správnému načasování dochází i při spouštění kernelů. Ty, které mají spočítat dlouhé řádky, se spustí dříve, než ostatní. Existují totiž matice, kde jeden tento složitý problém trvá stejně dlouho, jako výpočet všech ostatních řádků, což by mohlo vést ke dvojnásobnému zpomalení.



Obrázek 5.3 Fáze algoritmu OpSparse

Až na několik výjimek je tento algoritmus totožný s jeho vzory, ale díky malým optimalizacím, se kterými přichází, dosahuje i v těch nejhorších případech desetiprocentního zrychlení nad ostatními. [29]

5.7 BRMerge

Posledním popisovaným bude BRMerge. Jako jediný algoritmus je zaměřen na CPU. Podobně jako RMerge 5.1, použije techniku slévání. Hlavním cílem bylo vyřešit neefektivní práce s pamětí, která je spojena s předchozími variantami těchto přístupů. U GPU je například kritizován přístup jednoho vlákna k většímu množství přeškálovaných řádků, k těm není v rámci sub-warpu přímočarý přístup, vedoucí k poklesu výkonu.

Násobení se skládá ze dvou částí, které jsou pro slévání typické. Každý řádek z B, na který poukazuje hodnota z A, se přeškáluje. Tím hlavním rozdílem zde je, že se jednotlivé řádky umisťuji do jedoho pole. Začátky řádků, respektive jejich konce, hlídá další pole. Způsob uložení je tak totožný s ukládáním matice v CSR. Pro alokaci paměti se používá horní odhad, kterým je počet všech násobení. Toto pole se alokuje celkem dvakrát, to druhé se použije později.

V druhé fázi, kterou nazývají akumulační, se ze všech mezivýpočtů stane jeden řádek. Naprogramována je pomocí dvou cyklů. Vnější pouze kontroluje počet listů, které se stále ještě musí zpracovat. Pokud se eviduje pouze jedno pole, rekonstrukce řádku matice C byla úspěšně dokončena.

Poté se postupně berou dvě pole a slévají se dohromady, výsledky se umisťují do dalšího pole. Myšlenka je identická s první fází, evidují se sloupcové indexy a délky řádků. Tento krok se provede pro každý pár. Počet polí se tak sníží o polovinu. Nejde o slévání výsledku prvních dvou polí s třetím a tak podobně, postupuje se po dvojicích. Na konci jedné této iterace se prohodí ukazatele na jednotlivá pole. Tato dvě pole nazývají pracovně ping-pong buffer.

Byly navrženy dvě varianty tohoto přístupu, lišící se od sebe myšlenkou alokace paměti. Jak již bylo zmiňováno dříve (kde), lze matici spočítat v dostatečně velkém poli a pak kopírovat data do výsledné matice, nebo matici spočítat dvakrát. Prvním výpočtem zjistit, jak matice má vypadat, a provést alokaci. Druhé násobení pak může výpočty vkládat rovnou do výsledné matice, a nemusí se provádět kopírování.

Kombinace symbolické a numerické fáze se ukazuje jako ta lepší varianta, na autory vybraných maticích dosahuje v průměru pětiprocentního zrychlení. Nejde o jednoznačný výsledek, byly matice, na kterých se jednofázový přístup s kopírováním dat ukázal jako ten lepší. [19]

Kapitola 6
Knihovny

Pro práci na CPU i GPU existují knihovny, které přináší prostředky využitelné u násobení maticemi. Může jít o načítání matic v řídkém formátu, vlastní dynamická alokace paměti [5, 32], či nějaká jednoduchá varianta celého procesu násobení, se kterým se algoritmy [38, 32] porovnávaly.

6.1 OpenMP

Pro zjednodušení práce s vlákny na CPU se používá OpenMP. Jde o průmyslový standard, jehož začátky se datují do roku 1997. Nejde o programovací jazyk, pouze pomocí direktiv rozšiřuje ty již existující.

Se zjednodušeným pohledem na knihovnu jde brát OpenMP jen jako sadu direktiv, které se umisťují nad bloky a teprve s přiložením knihovny se mění chování programu. Direktivami se označují místa, která jsou vhodná pro paralelizaci, nebo v případě již paralelního kódu potřebují speciální péči, například kritická sekce, místo, do kterého by mělo vstoupit pouze jedno vlákno a tak podobně.

Před každým paralelním regionem se vytvoří daný počet vláken, jinak se používá jedno hlavní *master* vlákno. Tomuto způsobu se říká *fork/join*. V každé direktivě může být počet vláken definován, jinak se hodnota může nastavit funkcí, nebo se použije hodnota proměnného prostředí.

Tím hlavním, pro co OpenMP bude využíváno, je paralelizace cyklů. Jsou zde dva hlavní způsoby, jak naplánovat dělbu práce. Statické plánování jednoduše rozdělí cyklus rovnoměrně podle počtu iterací a počtu vláken. Druhým způsob je dynamický, které jednotlivé iterace nepřiřazuje ihned na začátku, ale pracující vlákna po dokončení práce žádají o další cyklus. Na konci každého takového paralelního regionu je implicitní bariéra, na které vlákna bez práce čekají na ostatní, ještě počítající. [33]

6.2 Intel Math Kernel Library a oneAPI

Pro práci na vícevláknových procesorech přichází společnost Intel s poměrně rozsáhlou knihovnou, nazývanou Math Kernel Library (MKL). Lze ji dále volně kategorizovat do sekcí podle zaměření na vybrané matematické problémy či hardware. Je zde k nalezení sada funkcí pro rychlou Fourierovu transformaci, v případě maticových operací jde například o faktorizaci, singulární rozklad. K těmto algoritmům existují jejich vícevláknové varianty, či varianta pro distribuované systémy. Pro paralelizaci se využívá knihovna OpenMP. [34]

Listopadu roku 2019 vzniká vize unifikovaného programovacího modelu pro různé architektury procesorů. Skládá se z několika částí, základ tvoří knihovny, co mají pomáhat s vývojem

výkonných aplikací zabývající se primárně zpracováváním dat. Další oblasti jsou více specializované, jde o vizualizaci, stavbu neuronových sítí či nástrojů využívající strojového učení. [35]

V prosinci dalšího roku dochází k prvnímu oficiálnímu vydání sady nástrojů-knihoven nazývané oneAPI. Od první zprávy došlo například k přidání matematické knihovny do tohoto balení knihoven a ze samostatného MKL se stává část oneAPI nazývané oneMKL. [36]

Matematická knihovna není jedinou oblastí API, která je pro maticové násobení zajímavá. Protože cílem je pomoci při vývoji výkonných aplikací, přináší s sebou také prostředky, pomáhající při vývoji paralelně pracujících programů. Thread Building Blocks, oneTBB, obsahuje paralelní varianty běžných algoritmů či programovacích konstrukcí. Jde o *for* cykly, řazení, redukce. Alokování paměti má rovněž svou paralelní variantu, nazývá se *scalable_allocator*, a práce s pamětí se provádí v takovém stylu, aby byla škálovatelná s počtem vláken. [32] Dle měření prováděného v práci z roku 2018 (Asi lepší odkaz) paralelní alokace a dealokace jsou od jisté velikosti paměti řádově lepší v porovnání s běžným sekvenčním přístupem. TBB je na intervalu 1 GB až 4 GB horší než přístup C++ s *new* a *delete*. U 8GB hranice přichází zlom a paralelní C++ přístup je srovnatelný se sekvenčním řešením. Toto zhoršení nastává u druhého porovnávaného až u čísla 64 GB. [5]

6.3 CUDA, Thrust, cuSPARSE, cuBLAS

V roce 2006 společnost NVIDIA představila platformu umožňující využívat prostředky grafických karet od stejné společnosti. Využívat sílu této technologie je možné v jazycích *C*, *C++* či *Java*. Jde o celý ekosystém, přináší vlastní profiler a celou řadu knihoven, které budou zmíněně později. [**cuda**]

CUDA pouze rozšiřuje standardní *C* o možnost pracovat s grafickou kartou. Nenachází se zde žádné složitější datové struktury či funkce. Pro tyto účely je nutné použít další knihovny.

Zmíněny tu budou tři, které se nějakým způsobem týkají matic, nebo nabízí funkce, které by se daly využít v algoritmech pro maticové násobení.

Thrust připomíná standardní C++ knihovnu. Je plná šablon, struktur a algoritmů, které ve většině případů lze nalézt i ve jmenném prostoru *std*. Struktury jsou dvojího druhu, *device* pro GPU, *host* pro CPU. Hosta je možné nahradit strukturami ze standardní knihovny. Některé z nich nejsou přímo implementovány, je ale zaručena kompatibilita. Tyto dva druhy je možné mezi sebou kopírovat (operátor =, *thrust::copy*), není proto nutné používat *cudaMemcpy*. Vektory a další struktury jde vedle kopírování ještě transformovat, například pomocí funktorů; provádět s nimi redukce, inicializovat pomocí sekvencí. Většina algoritmů má svou reprezentaci i ve standardní knihovně, Thrust pouze přináší její *kopii* pro paralelní zpracování na grafické kartě. v této oblasti je nejzajímavější algoritmus paralelní redukc, který se dá velmi dobře použít pro výpočet indexu pro jednotlivé starty řádků v řídké matici. Tento algoritmus má svou reprezentaci v C++, ale jeho využití je spojené zejména s paralelizací, například v řadicím algoritmu *radix sort* a již zmíněném ukládání mezivýsledků matice do výsledného komprimovaného formátu. [37]

CuSPARSE je knihovna zaměřena na násobení řídkých matic. Přináší proto podporu pro celou řadu formátů, většina z nich byla zmíněna v kapitole o formátech řídkých matic (2). Algoritmus pro násobení dvou řídkých matic zde implementovaný je tou první referencí, se kterou se další implementace musí srovnávat.

Přináší velké množství svých datových typů a konstant. Například pro číselné datové typy přichází s dalšími osmi variantami. Ze základních datových typů se tu používá pouze *float* a *double*, i celočíselné operace dostávají vlastní speciální datové typy. Vedle celých čísel se zde používají také komplexní čísla a šestnáctibitové varianty celých a desetinných čísel.

Každá funkce z této knihovny vrací nějaký svůj status. Datový typ je jiný než ten, který se používá v CUDA, což může komplikovat práci.

Dále bude popsán postup, jakým se pomocí funkcí nabízených cuSPARSE provede maticové násobení. Jsou zde i další operace, například násobení řídké matice hustou maticí, násobení

řídké matice vektorem a další kombinace. Předpisy funkcí si jsou velmi podobné, oproti dvěma řídkým maticím jde o předpisy vždy jednodušší – některé argumenty ztrácejí v jiných operacích na smyslu.

Než se začnou řídké matice násobit, musí se vytvořit pomocí funkcí, které knihovna nabízí. Postup tak připomíná Intel MKL 6.2, i zde se řídká matice převáděla do vnitřní reprezentace, které knihovna rozumí. Všechna data musí být v paměti grafické karty. I přesto, že knihovna umí celou řadu formátů, u násobení řídké matice řídkou maticí je možné používat pouze formát komprimovaných řádků. Násobení probíhá ve dvou krocích. Nejdříve se odhadne, jak velký buffer bude pro násobení potřeba. Tak velkou pamět programátor naalokuje a pak může spustit samotný výpočet. Pomocí dalších funkcí se dají zjistit rozměry matice, počet hodnot, a nastaví se správně všechny ukazatele, aby bylo možné překopírovat výsledek zpět na hosta. [38]

Poslední zmíněnou knihovnou bude cuBLAS. Ta pro tuto práci neposlouží jako reference, ani jako zdroj algoritmů. Je ale dobré ji zmínit, protože jde o sadu funkcí použitelných pro lineární programování. Dá se proto o ni hovořit jako o cuSPARSE, který je zaměřen na husté matice. Některé základní myšlenky jsou totožné – jsou tu datové typy totožné s cuSPARSE, a pracuje se s jinými stavy, pro knihovnu specifické.

V dokumentaci jsou funkce rozděleny do tří kategorií, podle druhu operací. V první sadě jsou vektorové operace – například hledání největší hodnoty, výpočet sumy, skalárního součinu, L2 normy, či aplikace operátoru rotace.

Další v pořadí je násobení matice vektorem. Je zde celá řada funkcí, od obecného násobení po operace nad specializovanými maticemi, u kterých je možné řešení najít rychleji. Stejným tvrzením jde popsat funkce pracující s dvěma maticemi. U matic se navíc řeší, zda některá z nich není transponovaná, a navíc je zde například funkce pro LU faktorizaci. [39]

Kapitola 7

Implementace

Vlastní algoritmizace bude popsána nějakým způsobem chronologicky. Nejdříve se popíše první naivní implementace, na které se popíší slabá místa a v dalších sekcích budou jejich vylepšení a rozšíření. Nehledě na různé iterace, na začátku implementace muselo dojít k rozhodnutí, jakou cestou se vydat.

Jako programovací jazyk byl vybrán *C*. Pro paralelizaci byla použita knihovna OpenMP6.1, která byla popsaná v samostatné sekci. Pro násobení matice se použije akumulátorový způsob, konkrétně hashovací tabulka. Algoritmus bude jednoprůchodový – provede se prvotní horní odhad paměti, ve které se provede násobení, následně se data překopírují do další, tentokrát již přesné paměti.

Rozhodnutí pro jednoprůchodový algoritmus jsem provedl po pozorování časové náročnosti novějších algoritmů jako spECK 5.5, OpSparse 5.6 či BRMerge 5.7, ve kterých byl časový rozdíl mezi numerickou a symbolickou fází minimální. Proto hned na počátku jsem usoudil, že bude lepší pouze lehkou analýzu vstupu a vypočítaná data překopírovat, než matice počítat dvakrát. Zároveň jde o jakýsi kontrast se zmíněnými algoritmy z posledních let, každý z nich zvolil dvoufázový přístup.

7.1 První varianta

Algoritmus jde rozdělit do pěti fází, při velmi jemném dělení do sedmi. Tři z nich se zabývají alokací, respektive dealokací paměti.

Nejdříve se provede alokace paměti, aby šlo provést základní analýzu součinu. Paměti není třeba mnoho, stačí jedno pole pro uložení počtu prvků, které lze v každém řádku výsledné matice očekávat.

Na základě této analýzy je možné si připravit paměť, která se použije pro hashování. Odhadnutý počet hodnot nabývá dalšího významu, půjde o index do pole, na kterém začíná hashovací tabulka pro daný řádek. Před tím, než se začne takto používat, musí se provést prefixový součet. Během hashování se spočítají všechny mezivýpočty. Používá se jednoduché lineární hashování s otevřenou adresací. Poté se hodnoty v tabulce přesunou na začátek pole, aby řazení neprobíhalo nad zbytečně velkým irelevantních dat.

Po seřazení se data překopírují do výsledné matice. Protože se během hashování hlídalo skutečné množství hodnot, není obtížné tuto alokaci a následné kopírování dat provést.

7.2 Používání vektoru

I přesto, že jsem zvolil hashování jako svou metodu pro akumulaci hodnot, není od věci se tomuto způsobu vyhnout. Pokud to bude možné, bude se používat vektoru. Vektor je tak velký, aby se hashovat nemuselo, sloupcový index jednoznačně určuje pozici ve vektoru. Tato idea má spousty výhod, nedochází ke konfliktům, zachovává se vzestupná posloupnost sloupcových indexů, díky čemuž se pak vektor nemusí řadit. Nemusí se přistupovat k poli s mezivýsledky, které je určeno pro sloupcové indexy, ty lze odvodit z kontextu.

Problémovým bodem je odhalování takového momentu, kdy je dobré začít používat vektor. Protože chci používat jednoduchou hashovací funkci – identitu, od které se případně jen odečte hodnota prvního nenulového sloupcového indexu – musí vektor tvořit posloupnost indexů. Vektor se použije tehdy, když jsou sloupcové indexy mezi nejmenším indexem a největším menší, než velikost hashovací tabulky. Tedy během analýzy součinu se vedle počtu součinů pozorují sloupcové indexy jednotlivých řádků pravé strany násobení. Jednoduché sčítání v cyklu je rázem zkomplikováno o celou řadu podmínek, avšak výhody převažují nad nevýhodami. Protože řazení je tou nejkomplikovanější operací, jakýkoliv způsob, jak se mu vyhnout, je vylepšením.

7.3 Řazení

Nejjednodušším způsobem, jak seřadit data pro, je použití knihovní funkce *sort*. Ta má ale jednu velkou nevýhodu, a tou je její předpis. Vstupem je jediné pole a řadicí funkce. V případě formátu komprimovaných řádků je ale pole jako argument nevýhodné, protože zde je cílem pomocí pole sloupcových indexů seřadit nejen tyto indexy, ale zároveň hodnoty, nacházející se v druhém poli, které s danými indexy souvisí. Tento problém jde vyřešit přímočaře použitím struktury. Protože se řadí pouze mezivýpočet, nejde o velkou komplikaci – předpis funkce zůstává stejný, struktura se správně umístí do výsledné matice, která používá tři pole.

Jde však o bezkonkurenčně nejslabší článek algoritmu. Čas, strávený řazením, tvořil většinu běhu programu. Hlavním cílem bylo zbavit se struktury. Protože se pracovalo primárně se sloupcovými indexy, je výhodné, že jsou u sebe v jednom poli. Tuto strukturu paměti přímočará metoda narušuje, v té byla data uložena v páru *sloucový index-hodnota*. Po implementaci vlastní varianty *quicksort* bylo možné pracovat s tímto lepším rozložením dat v paměti a čas se zlepšil o polovinu.

Mergesort tvoří s *quicksort* pomyslnou dvojici standarních řadicích algoritmů. Algoritmus slévání má ale jeden významný neduh, a tím je potřeba pomocné paměti pro výpočty. Ale protože můj algoritmus má již v této fázi naalokovanou pamět navíc, nezpůsobuje mi tato nevýhoda problémy, v některých případech by se dalo s trochou nadsázky tvrdit, že jde rovnou o výhodu. Mým pomocným polem zde bude výsledná matice, díky čemuž si v některých případech (pro seřazení bude třeba lichého počtu slévání) mohu ušetřit kopírování dat do finální matice. Má varianta není rekurzivní, postupuje se opačně. Místo půlení intervalů na triviální případy a jejich následné slévání si zde neefektivním kvadratickým algoritmem seřadím osmice prvků (můj triviální případ), ze kterých sléváním vytvořím sekvence delší. Tento přístup by měl zaručit, že se bude respektovat zarovnání dat v paměti a práce s cache bude efektivní. Podobně jako *quicksort*, i zde došlo k velkému zrychlení, a to opět dvojnásobnému. Oproti první naivní implementaci tak jde o čtyřnásobné zrychlení.

7.4 Půlení pravé strany násobení

Velké množství prvků vstupující do násobení negativním způsobem ovlivňuje všechny aspekty algoritmu. Protože horní odhad je příliš pesimistický, naznačuje, že bude potřeba velkých hashovacích tabulek a ty negativně ovlivní rychlost řazení.

Výpis kódu 7.1 Ukázka branchless kódu

```
#ifdef BRANCHLESS
        t += (t > HASH_TABLE_THRESHOLD) * t;
        t = (t > r->width) * r->width + (t <= r->width) * t;
#else
        if (t > HASH_TABLE_THRESHOLD)
        {
                t *= 2;
        }
        if (t > r->width)
        {
                t = r->width;
        }
#endif
```

Přepínání na hustý vektor dokáže eliminovat pouze primitivní případy, ve kterých jsou sloupcové indexy začátku a konce blízko sebe. Zároveň do prvotní analýzy přináší nemalé množství dalších podmínek, které tuto první část, která by měla být jednoduchá, abych byl věrný jednofázovému přístupu, značně komplikuje.

Proto jsem se rozhodl experimentovat s rozpůlením pravé matice na dvě menší. Místo B_{mn} se bude pracovat s $B1_{m\frac{n}{2}}$ a $B_{m\frac{n}{2}}$. Cílem je zmenšit pravou stranu tak, aby počet násobení v jednotlivých hashovacích tabulkách menší. Díky tomu je možné hashovací tabulku zmenšit, což vede na menší počet konfliktů a dále jednodušší řazení. Před začátkem přípravy se vypočítají poloviny řádků – upraví se rozsahy, do kterých patří který index. Pokud by se měla použít terminologie z CSR, dalo by se přeneseně tvrdit, že jde o přidávání dalších řádků.

7.5 Optimalizace

Zde budou popsány menší optimalizace, které si buď nezaslouží vlastní kapitolu, nebo se netýkají výlučně některé z fází algoritmu.

Jednou z takových úprav je nastavení velikosti hashovací tabulky. Podobně jako v OpSparse 5.6, i zde po horním odhadu hashovací tabulku přenásobím, aby měla dvojnásobné rozměry. Díky tomu by mělo docházet k menšímu množství konfliktů. Je tu zavedena horní hranice v podobně šířky vstupní matice – není důvodu, aby tabulka byla větší, než matice, tolik unikátních sloupcových indexů zde není. K přenásobení dojde pouze tehdy, když je hashovací tabulka větší než čtrnáct. Tuto hodnotu jsem empiricky zvolil při prvotních pokusech, kdy se ukázalo, že je lepší mít víc konfliktů, než pomalejší řazení. Příliš velká tabulka negativně ovlivňuje přípravu před řazením, protože je zde teoreticky dvojnásobek prázdných polí.

Slabým místem je alokace paměti. Standardní *malloc* a *free* používá pouze jedno vlákno, což je v případě paralelního programování úzké hrdlo implementace. Díky použití *scalable_allocator* z OneAPI **??** se čas běhu v částech programu, které jsem bral jako neměnné, velmi pozitivně zlepšil. Měření nebude prováděno bez této technologie. OneAPI má jednu „nevýhodu“, která poukazuje na můj případný nevhodný výběr programovacího jazyka. Kód jde dále kompilovat jen za použití jejich kompilátoru, který vše kompiluje jako *C++* kód.

Další ze způsobů, jak lze upravit kód, je eliminace podmínek, což je technika, která si zasloužila samostatnou sekci v optimalizacích (viz **??**). Jednou z ukázek může být například úprava velikosti hashovací tabulky. Zde se místo dvou podmínek, které vedou k podmíněnému skoku, využijí podmínky tři, které pouze nastaví správně příznak 0/1, aby násobení mělo kýžený efekt pouze tehdy, kdy je splněna podmínka, což vede ke stejným výsledkům.

Tato eliminace podmínek byla provedena jen v první fázi, ve které se analyzuje vstup. Nebylo pozorováno zrychlení.

V prvních variantách se místo paralelního prefixového součtu používá sekvenční varianta s jedním vláknem. Protože na stanici, na které se provádí měření, nepoužívá verzi OpenMP, která umí tento součet vypočítat, použil jsem implementaci z OneAPI. Výsledky jsou rozporuplné, podobně jako *branchless* i zde je těžké pozorovat zrychlení, což je v případě převodu sekvenční verze na paralelní velmi kontroverzní. Vysvětlení může být poměrně prosté – sekvenční varianta vše prováděla *in-place* a další paměť zkrátka vede k horším časům.

Místo toho, aby OpenMP přidělovalo řádky matice vláknu vlastním způsobem, vypočítal jsem si vlastní dělící body. Některé z matic mají z počátku příliš krátké řádky a ke konci jde o neúměrné zatížení, ve kterém poslední vlákno provádí většinu výpočtů. OpenMP na tento problém odpověď má, dynamické přidělování práce. Dynamické má ale větší režii a může narušit prostorovou lokalizaci.

Kapitola 8
Měření

Všechny algoritmy dříve zmiňované pracovaly ve vší obecnosti. Avšak v sekcích, které se zabývaly měřením výkonu, se pozornost zaměřila zejména na mocnění matice. Jsou zde dvě výjimky, které budou sloužit jako inspirace pro výběr dalších testovacích případů.

Práce z roku 2018 pro své testovací případy použila matici grafu, z něj vybrala některé sloupce, a s nimi provedla maticové násobení. [5]

Cílem této kapitoly je vedle standardních čtvercových matic, na kterých se algoritmy běžně porovnávají, přidat i další případy a rozšířit tak rozsah případů, na kterých se algoritmy porovnávají.

8.1 Testovací prostředí

Pro testování bude použit školní cluster *star*. Na jednom z uzlů, pojmenovaný *gpu-02*, se nachází grafická karta GeForce RTX 2080 Ti, dva procesory Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10 GHz, který má šest jader a dvanáct vláken, spolu s 32 GB paměti.

8.2 Matice

Pro mocnění čtvercových matic byla vybrána podmnožina, která se objevovala napříč všemi pracemi.

V kolekci matic se ve většině případů objevují matice čtvercových rozměrů. Dá se tu ale narazit i na obdélníkové matice, avšak ty se nikdy neobjevily v grafech naměřených časů. Aby šlo u těchto matic provádět mocninu, pravá strana se transponovala. Validní výraz je ale možné vytvořit i transponováním levé strany, což v tomto případě mění rozměry výsledné matice a klade zcela jiné nároky na vyvažování zátěže. [2]

U nesymetrických čtvercových matic jde pro větší variabilitu provádět $C = A * A^T$ a $C = A^T * A$.

Do kolekce matic se umisťují sady problémů, které spadají do stejné kategorie, mají stejného autora, a dokonce i stejné rozměry. Zde je proto jednoduché provést násobení $C = A * B$.

Pro transpozici se použilo jednoduchého skriptu v PHP, který ma celou řadu nevýhod; například nemožnost transponovat velmi velké matice z důvodu paměťové náročnosti.

Tabulka 8.1 Použité matice

Název	Počet řádků	Počet sloupců	NNZ	Symetrie
2cubes-sphere	101492	101492	874378	Ano
144	144649	144649	1074393	Ano
amazon0312	400727	400727	3200440	Ne
bibd-19-9	171	92378	3325608	Ne
cage12	130228	130228	2032536	Ne
cage13	445315	445315	7479343	Ne
landmark	71952	2704	1151232	Ne
stat96v2	29089	957432	2852184	Ne
webbase-1M	1000005	1000005	3105536	Ne

8.3 Měřené algoritmy

Implementovaný algoritmus se bude měřit v poslední iteraci – s půlením pravé matice. V tabulkách budou obsaženy pouze výsledky běhů, které použily dvanáct vláken a všechny optimalizace.

CPU algoritmy zde budou zastoupeny vedle implementovaného navíc knihovním Intel MKL a BRMerge. MKL je spuštěn pokaždé, kdy se počítá implementovaný algoritmus jako reference.

GPU algoritmy zde budou zastoupeny spECK a ACSpGEMM. Ostatní buď využívají buď příliš starou CUDA knihovnu, nebo byl zdrojový kód tak upraven na míru grafické karty, se kterou autoři pracovali tak, že nebylo možné ani po úpravách program úspěšně spustit. Každý z algoritmů měl předpis upraven tak, že nepřijímal jakékoliv matice, ať už jde o cestu k nim, nebo o formát. Zároveň zde byla zatajena režie kopírování dat z/na GPU, díky čemuž časy vypadaly skvěle, ale nebylo by možné je porovnat s řešením používající standardní procesor.

Pokud to bude možné, budou porovnávány CPU a GPU programy navzájem. Právě díky měřené řežii kopírování je možné pozorovat, kdy se vyplatí k takto drahé operaci přistoupit.

8.4 Naměřené hodnoty

Bohužel nebylo možné z důvodu paměťové náročnosti provádět součit všech transpozicí. V tabulkách níže budou zmíněné všecny naměřené hodnoty. Název součinu odpovídá použitým vstupům. Všechna měření jsou průměrem minimálně padesáti hodnot.

Tabulka 8.2 Naměřené hodnoty CPU

Součin	Vlastní (v ms)	BRMerge (v ms)	MKL (v ms)
2cubes-sphere^2	185,381	79,554	133,263
144^2	236,535	124,929	190,678
amazon0312 * amazon0312^T	1123,251	404,442	1361,899
amazon0312^T $*$ $amazon0312$	286,968	141,031	357,538
bibd-19-9 * bibd-19-9^T	176,402	409,940	140,848
cage12^2	247,422	104,078	252,105
cage12 * cage12^T	274,768	121,170	259,696
cage12^T $*$ $cage12$	241,875	118,052	252,038
cage12^T $*$ $cage12^T$	275,051	103,009	253,360
cage13^2	903,134	452,019	1078,064
cage13 * cage13^T	920,239	456,641	1085,056
cage13^T $*$ $cage13$	948,127	449,464	1026,071
cage13^T $*$ $cage13^T$	927,481	456,093	1059,856
landmark^T $*$ $landmark$	52,938	22,801	15,597
stat96v2 * stat96v2^T	73,968	11,960	18,396
webbase-1M^2	658,763	176,728	1029,172

Tabulka 8.3 Naměřené hodnoty GPU

Součin	ACSpGEMM (v ms)	spECK (v ms)
2cubes-sphere^2	8,268	4,784
144^2	9,895	5,452
amazon0312 * amazon0312^T	33,618	45,2407
amazon0312^T $*$ $amazon0312$	10,6555	13,723
bibd-19-9 * bibd-19-9^T	17,629	14,004
cage12^2	10,388	8,446
cage12 * cage12^T	10,653	8,446
cage12^T $*$ $cage12$	10,388	8,449
cage12^T $*$ $cage12^T$	10,403	8,448
cage13^2	37,155	25,450
cage13 * cage13^T	37,085	26,066
cage13^T $*$ $cage13$	37,086	26,219
cage13^T $*$ $cage13^T$	37,419	24,821
landmark^T $*$ $landmark$	4,101	1,906
stat96v2 * stat96v2^T	3,634	3,152
webbase-1M^2	19,707	37,820

Kapitola 9
Závěr

Ukázalo se, že BRMerge je až na jeden případ v průměru dvojnásobně rychlejší, u některých instancí dokonce čtyřnásobně. Ve zmíněném jediném případě je vlastní implementace více než dvakrát rychlejší.

Protože kopírování dat z CPU na CPU je dosti drahou operací, trvající v některých případěch přes půl sekundy, je BRMerge ve spoustě případů obstojným soupeřem pro algoritmy využívající grafické karty.

SpECK až na tři součiny nabízí čtvrtinové až poloviční zrychlení.

Bibliografie

1. YANG, Carl; BULUC, Aydin; OWENS, John D. *Design Principles for Sparse Matrix Multiplication on the GPU*. arXiv, 2018. Dostupné z DOI: 10.48550/ARXIV.1803.08601.
2. DAVIS, Timothy A.; HU, Yifan. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 2011, roč. 38, č. 1. ISSN 0098-3500. Dostupné z DOI: 10.1145/2049662.2049663.
3. BULUC, Aydin; GILBERT, John R. On the representation and multiplication of hypersparse matrices. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008, s. 1–11. Dostupné z DOI: 10.1109/IPDPS.2008.4536313.
4. GREMSE, Felix; HÖFTER, Andreas; SCHWEN, Lars Ole; KIESSLING, Fabian; NAUMANN, Uwe. GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging. *SIAM Journal on Scientific Computing*. 2015, roč. 37, s. C54–C71. Dostupné z DOI: 10.1137/130948811.
5. NAGASAKA, Yusuke; MATSUOKA, Satoshi; AZAD, Ariful; BULUÇ, Aydın. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. 2018. Dostupné z DOI: 10.48550/ARXIV.1804.01698.
6. BOISVERT, Ronald; POZO, Roldan; REMINGTON, Karin. The Matrix Market Exchange Formats: Initial Design. 1997.
7. FUNKE, Daniel; LAMM, Sebastian; MEYER, Ulrich; SANDERS, Peter; PENSCHUCK, Manuel; SCHULZ, Christian; STRASH, Darren; LOOZ, Moritz von. *Communication-free Massively Distributed Graph Generation*. arXiv, 2017. Dostupné z DOI: 10.48550/ARXIV.1710.07565.
8. HÜBSCHLE-SCHNEIDER, Lorenz; SANDERS, Peter. Linear work generation of R-MAT graphs. *Network Science*. 2020, roč. 8, č. 4, s. 543–550.
9. SULATYCKE, P.D.; GHOSE, K. Caching-efficient multithreaded fast multiplication of sparse matrices. In: *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. 1998, s. 117–123. Dostupné z DOI: 10.1109/IPPS.1998.669899.
10. DALTON, Steven; OLSON, Luke; BELL, Nathan. Optimizing Sparse Matrix—Matrix Multiplication for the GPU. *ACM Trans. Math. Softw.* 2015, roč. 41, č. 4. ISSN 0098-3500. Dostupné z DOI: 10.1145/2699470.
11. CORPORATION, Intel. Sparse BLAS Coordinate Matrix Storage Format. In: 2023. Dostupné také z: https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2023-0/sparse-blas-coordinate-matrix-storage-format.html.

12. CORPORATION, Intel. Sparse BLAS CSR Matrix Storage Format. In: 2023. Dostupné také z: https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2023-0/sparse-blas-csr-matrix-storage-format.html.

13. EUN-JIN IM, Katherine Yelick. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. In: 2001. Dostupné také z: https://bebop.cs.berkeley.edu/pubs/im2001-regreuse-iccs.pdf.

14. CORPORATION, Intel. Sparse BLAS BSR Matrix Storage Format. In: 2023. Dostupné také z: https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2023-0/sparse-blas-bsr-matrix-storage-format.html.

15. CORPORATION, IBM. Compressed-Diagonal Storage Mode. In: 2021. Dostupné také z: https://www.ibm.com/docs/en/essl/6.2?topic=representation-compressed-diagonal-storage-mode.

16. CORPORATION, Intel. Sparse BLAS Skyline Matrix Storage Format. In: 2023. Dostupné také z: https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2023-0/sparse-blas-skyline-matrix-storage-format.html.

17. ANZT, Hartwig; TOMOV, Stanimire; DONGARRA, Jack J. Implementing a Sparse Matrix Vector Product for the SELL-C / SELL-C-σ formats on NVIDIA GPUs. In: 2014. Dostupné také z: https://library.eecs.utk.edu/files/ut-eecs-14-727.pdf.

18. NAGASAKA, Yusuke; NUKADA, Akira; MATSUOKA, Satoshi. Adaptive Multi-level Blocking Optimization for Sparse Matrix Vector Multiplication on GPU. *Procedia Computer Science*. 2016, roč. 80, s. 131–142. ISSN 1877-0509. Dostupné z DOI: https://doi.org/10.1016/j.procs.2016.05.304. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

19. DU, Zhaoyang; GUAN, Yijin; GUAN, Tianchan; NIU, Dimin; ZHENG, Hongzhong; XIE, Yuan. Accelerating CPU-Based Sparse General Matrix Multiplication With Binary Row Merging. *IEEE Access*. 2022, roč. 10, s. 79237–79248. Dostupné z DOI: 10.1109/ACCESS.2022.3193937.

20. ELLIOTT, James J.; SIEFERT, Christopher M. Low Thread-Count Gustavson: A Multithreaded Algorithm for Sparse Matrix-Matrix Multiplication Using Perfect Hashing. In: *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*. 2018, s. 57–64. Dostupné z DOI: 10.1109/ScalA.2018.00011.

21. BULUÇ, Aydin; GILBERT, John. On the representation and multiplication of hypersparse matrices. In: 2008, s. 1–11. Dostupné z DOI: 10.1109/IPDPS.2008.4536313.

22. AZAD, Ariful; BALLARD, Grey; BULUÇ, Aydin; DEMMEL, James; GRIGORI, Laura; SCHWARTZ, Oded; TOLEDO, Sivan; WILLIAMS, Samuel. Exploiting Multiple Levels of Parallelism in Sparse Matrix-Matrix Multiplication. *SIAM Journal on Scientific Computing*. 2016, roč. 38, č. 6, s. C624–C651. Dostupné z DOI: 10.1137/15m104253x.

23. LIU, Weifeng; VINTER, Brian. A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors. *CoRR*. 2015, roč. abs/1504.05022. Dostupné z arXiv: 1504.05022.

24. GILBERT, John; MOLER, Cleve; SCHREIBER, Robert. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*. 1997, roč. 13. Dostupné z DOI: 10.1137/0613024.

25. COHEN, Edith. Structure prediction and computation of sparse matrix products. *Journal of Combinatorial Optimization*. 1998, roč. 2, č. 4, s. 307–332.

26. WINTER, Martin; MLAKAR, Daniel; ZAYER, Rhaleb; SEIDEL, Hans-Peter; STEINBERGER, Markus. Adaptive Sparse Matrix-Matrix Multiplication on the GPU. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. Washington, District of Columbia: Association for Computing Machinery, 2019, s. 68–81. PPoPP '19. ISBN 9781450362252. Dostupné z DOI: 10.1145/3293883.3295701.

27. NAGASAKA, Yusuke; NUKADA, Akira; MATSUOKA, Satoshi. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. In: *2017 46th International Conference on Parallel Processing (ICPP)*. 2017, s. 101–110. Dostupné z DOI: 10.1109/ICPP.2017.19.

28. PARGER, Mathias; WINTER, Martin; MLAKAR, Daniel; STEINBERGER, Markus. SpECK: Accelerating GPU Sparse Matrix-Matrix Multiplication through Lightweight Analysis. In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. San Diego, California: Association for Computing Machinery, 2020, s. 362–375. PPoPP '20. ISBN 9781450368186. Dostupné z DOI: 10.1145/3332466.3374521.

29. DU, Zhaoyang; GUAN, Yijin; GUAN, Tianchan; NIU, Dimin; HUANG, Linyong; ZHENG, Hongzhong; XIE, Yuan. OpSparse: A Highly Optimized Framework for Sparse General Matrix Multiplication on GPUs. *IEEE Access*. 2022, roč. 10, s. 85960–85974. Dostupné z DOI: 10.1109/ACCESS.2022.3196940.

30. ELMASRY, Amr; KATAJAINEN, Jyrki. Branchless search programs. In: *International Symposium on Experimental Algorithms*. 2013, s. 127–138.

31. WILLIAMS, Samuel; OLIKER, Leonid; VUDUC, Richard; SHALF, John; YELICK, Katherine; DEMMEL, James. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. 2007, s. 1–12. Dostupné z DOI: 10.1145/1362622.1362674.

32. CORPORATION, Intel. oneAPI Specification. In: 2022. Dostupné také z: https://spec.oneapi.io/versions/latest/index.html.

33. CHANDRA, Rohit; DAGUM, Leo; KOHR, David; MENON, Ramesh; MAYDAN, Dror; MCDONALD, Jeff. *Parallel programming in OpenMP*. Morgan kaufmann, 2001. Dostupné také z: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=45faf2f1b9119079beda1383ea78497f499a649a.

34. WANG, Endong; ZHANG, Qing; SHEN, Bo; ZHANG, Guangyong; LU, Xiaowei; WU, Qing; WANG, Yajuan. Intel Math Kernel Library. In: *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*. Cham: Springer International Publishing, 2014, s. 167–188. ISBN 978-3-319-06486-4. Dostupné z DOI: 10.1007/978-3-319-06486-4_7.

35. CORPORATION, Intel. Fact Sheet: oneAPI. In: 2020. Dostupné také z: https://newsroom.intel.com/articles/fact-sheet-oneapi/.

36. CORPORATION, Intel. New Intel oneAPI Toolkits for XPU Software Development. In: 2020. Dostupné také z: https://www.intel.com/content/www/us/en/newsroom/news/oneapi-toolkits-xpu-software-development.html.

37. AFFILIATES, NVIDIA Corporation. Thrust. In: 2023. Dostupné také z: https://docs.nvidia.com/cuda/thrust/index.html.

38. AFFILIATES, NVIDIA Corporation. cuSPARSE. In: 2023. Dostupné také z: https://docs.nvidia.com/cuda/cusparse/index.html.

39. AFFILIATES, NVIDIA Corporation. cuBLAS. In: 2023. Dostupné také z: https://spec.oneapi.io/versions/latest/index.html.

Obsah přiloženého média