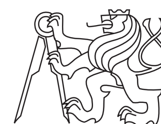
Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů



Diplomová práce

Přehled úložných formátů pro řídké matice pro paralelní systémy

Bc. Radek Bittara

Vedoucí práce: Ing. Daniel Langr

4. května 2015

Poděkování

Tímto bych chtěl poděkovat vedoucímu své diplomové práce, kterým byl Ing. Daniel Langr, za poskytnutí cenných rad a za celkové vedení mé práce. Dále bych chtěl poděkovat všem, kteří mě po dobu psaní této práce i po dobu celého mého studia podporovali.

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, 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 autorského zákona.

V Praze dne 4. května 2015 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2015 Radek Bittara. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Bittara, Radek. *Přehled úložných formátů pro řídké matice pro paralelní systémy.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Cílem této práce je poskytnout ucelený přehled dosud publikovaných formátů pro uložení řídké matice v paměti počítače. Práce obsahuje popis jejich principů a případné využití při násobení řídké matice s vektorem na moderních paralelních systémech.

Klíčová slova řídká matice, úložný formát, násobení řídké matice vektorem, SIMD, GPU

Abstract

The goal of this thesis is to provide a overview of published sparse matrix storage formats in computer memory. This thesis contains a description of the principles and potential use in a sparse matrix-vector multiplication for modern parallel systems.

Keywords sparse matrix, storage format, sparse matrix-vector multiplication, SIMD, GPU

Obsah

Úvod **1**

1 Základní problematika **3**
1.1 Řídká matice 3
1.2 Násobení řídké matice s vektorem (SpMV) 4
1.3 Ukládání řídkých matic 5
1.4 Moderní úložné formáty 6

2 Paralelní systémy **9**
2.1 Základní typy procesorů 9
2.2 Architektura SIMD 11
2.3 Architektura grafických procesorů 12
2.4 Moderní koprocesory 14

3 Základní formáty **17**
3.1 Coordinate Format (COO) 17
3.2 Compressed Row Storage (CSR/CRS) 18
3.3 Compressed Column Storage (CSC/CCS) 19
3.4 Diagonal Format (DIAG) 20
3.5 Compressed Daigonal Storage (CDS) 21
3.6 Jagged Diagonal Storage (JDS/JAD) 22
3.7 Transpose Jagged Diagnal Storage (TJDS) 24
3.8 Skyline Storage (SKS) 24
3.9 ELLPACK Format 25

4 Registrově-blokové formáty **29**
4.1 Blocked CSR/CSC (BCSR/BCSC) 29
4.2 Specified Machine parameters Using Register Blocking (SMURB) 30
4.3 Unaligned Block CSR (UBCSR) 30
4.4 Blocked-Based Compression Storage (BBCS) 32

4.5 Row Segmented Diagonal (RSDIAG) 33
4.6 Decomposed Block CSR (BCSR-DEC) 35
4.7 Decomposed Block Compressed Sparse Diagonals (BCSD-DEC) 36
4.8 Blocked Transpose Jagged Diagnal Storage (BTJDS) 36

5 CSR formáty určené pro GPU 39
5.1 Row-grouperd CSR (RgCSR) 39
5.2 Adaptive row-grouped CSR (ArgCSR) 40
5.3 Improved Compressed Row Storage (ICSR) 41
5.4 Compressed Multi-Row Storage (CMRS) 42
5.5 CSR with Segmented Interleave Combination (SIC) 43

6 ELLPACK formáty 45
6.1 Hybrid ELLPACK/COO (HYB) 45
6.2 Sliced ELLPACK (SELLPACK) 46
6.3 Blocked ELLPACK (BELLPACK) 47
6.4 ELLPACK-R (ELL-R) 47
6.5 ELLR-T 48
6.6 Sliced ELLR-T 49
6.7 ELLPACK-RP 50
6.8 Adaptive Warp-Balancing ELLPACK (AdELL) 51
6.9 Bisection ELLPACK (BiELL) 52
6.10 Padded Jagged Diagonal Storage (pJDS) 53
6.11 SELL-C-σ 54
6.12 ELLPACK Sparse Block (ESB) 55
6.13 Bisection JAD format (BiJAD) 57

7 Hierarchické formáty 59
7.1 Basic Hierarchical Format (BHF) 60
7.2 Advanced Hierarchical (AHF) 62
7.3 Adaptive Blocking Hierarchical Storage format (ABHSF) . . . 63
7.4 Hierarchical Sparse Matrix (HiSM) 64
7.5 Compressed Sparse Blocks (CSB) 65
7.6 Expanded Compressed Sparse Blocks (eCSB) 66
7.7 Extended Sparse Block Compressed Row Storage (SBCRSx) . . 67
7.8 Pattern-Based Representation (PBR) 67

8 Kompresní formáty 71
8.1 Arithmetical Coding Based format (ACB) 71
8.2 Delta-coded Sparse Row (DCSR) 72
8.3 CSR Delta Units (CSR-DU) 72
8.4 Row Pattern CSR (RPCSR) 73
8.5 Compressed Sparse eXtended (CSX) 74
8.6 CSR Values Indirect (CSR-VI) 76

9 Cache-blokové formáty **77**
9.1 Recursive CSR/CSC (RCSR/RCSC) 77
9.2 Cache-Adaptive heuristics-based Register Blocking (CARB) . . 78

10 Formáty založené na stromové struktuře **81**
10.1 Minimal Binary Tree (MBT) 81
10.2 Minimal Quadtree (MQT) 83

Závěr **85**
Návrh pro pokračování 86
Shrnutí 86

Literatura **87**

A Seznam použitých zkratek **95**

B Obsah přiloženého CD **99**

Seznam obrázků

1.1 Příklad řídké matice, kde černé čtverečky představují nenulové prvky 4

2.1 Ukázka vícejádrového procesoru 10
2.2 Zjednodušená ukázka mnohojádrového procesoru 10
2.3 SIMD architektura (zdroj: [1]) 11
2.4 Rozdíl mezi běžným a vektorovým výpočtem 12
2.5 Ukázka architektury NVIDIA GF100 Fermi (zdroj: [2]) 13
2.6 Architektura Knights Corner (zdroj: [3]) 14

3.1 Příklad reprezentace řídké matice ve formátu COO 17
3.2 Řídká matice ve formátu CSR 19
3.3 Řídká matice ve formátu CSC 20
3.4 Uložení řídké matice ve formátu DIAG 20
3.5 Princip ukládání diagonálního pásu ve formátu CDS 21
3.6 Zobecnění formátu CDS 22
3.7 Přeuspořádání řídké matice do formátu JDS (zdroj: [4]) 23
3.8 Přeuspořádání vstupní matice a vektoru do formátu TJDS 26
3.9 Příklad uložení a rozdělení matice ve formátu ELLPACK (zdroj: [5]) 27

4.1 BCSR struktura (zdroj: [6]) 30
4.2 Rozdělení matice do bloků pro potřeby UBCSR formátu (zdroj: [7]) 31
4.3 Použití formátu BBCS na řídkou matici (zdroj: [4]) 33
4.4 Princip rozdělení matice do řádkových segmentů v RSDIAG (zdroj: [8]) 34
4.5 Uložení matice a vstupního vektoru ve formátu BTJDS 38

5.1 Příklad matice neposkytující jednotný přístup pro CSR při GPGPU 39
5.2 Ukázka matice uložené ve formátu RgCSR (zdroj: [9]) 40
5.3 Ukázka matice uložené ve formátu ArgCSR (zdroj: [10]) 41
5.4 Zarovnání řádků v globální paměti GPU pro formát ICSR (zdroj: [11]) 42

5.5 Princip prokládání řádků použitý ve formátu SIC 44

6.1 Uložení řídké matice ve formátu HYB 46
6.2 Uložení matice do pásů Sliced ELLPACKu (zdroj: [12]) 47
6.3 Konstrukce BELLPACK (zdroj: [6]) 48
6.4 Uspořádání prvků v BELLPACK formátu (zdroj: [6]) 49
6.5 Uložení řídké matice pomocí ELLR-T pro $T = 2$ (zdroj: [13]) . . . 50
6.6 Příklad reprezentace AdELL formátu (zdroj: [5]) 52
6.7 Princip BiELL pro 8 vláken (zdroj: [14]) 53
6.8 Převod matice do formátu pJDS (zdroj: [15]) 54
6.9 Použití SELL-C-σ formátu pro různé σ (zdroj: [16]) 55
6.10 Struktura ESB (zdroj: [3]) 56
6.11 Převod seřazených řádků pro použití BiJAD formátu (zdroj: [14]) . 57

7.1 Princip hierarchických formátů (zdroj: [17]) 59
7.2 Princip hierarchického formátu COOCOO ($n = 16$, $c = 2$) 61
7.3 Princip reprezentace nenulových prvků v AHF formátu (zdroj: [18]) 62
7.4 Struktura formátu HiSM (zdroj: [19]) 65
7.5 Mortonův rekurzivní rozklad 66
7.6 Struktura formátu SBCRSx (zdroj: [20]) 68
7.7 Blokové vzory a jejich uložení ve formátu PBR (zdroj: [21]) 69

8.1 Převod matice z CSR do DCSR (zdroj: [22]) 72
8.2 Komprese indexů do formátu RPCSR (zdroj: [22]) 74
8.3 Detekce různých substruktur pro využití CSX (zdroj: [23]) 75
8.4 Nepřímé adresování unikátních hodnot ve formátu CSR-VI (zdroj: [24]) 76

9.1 Rekurzivní dělení matice v závislosti na počtu nenulových prvků v bloku u RCSR/RCSC pro různě velké cache (zdroj: [25]) 78
9.2 Ukázka diagonálních bloků a izolovaných prvků (zdroj: [26]) 79

10.1 Reprezentace vstupní matice (vlevo) pomocí stromu MBT (zdroj: [27]) 82
10.2 Reprezentace matice pomocí MQT (zdroj: [28]) 83

Seznam tabulek

1.1 Seznam moderních úložných formátů (zdroj: [29]) 7

8.1 Reprezentace jednotek ve formátu CSR-DU, kde u8 je 8 bitů a NR je nový řádek (zdroj: [24]) 73

Úvod

Násobení řídké matice s vektorem (SpMV) je považováno za jednu z nejdůležitějších numerických metod pro vědecké a technické obory. Má tedy smysl se výkonností a efektivností této operace zabývat. Dnešní doba paralelních systémů nám nabízí nespočet možností, jak tyto výpočty urychlovat, a dosahovat tak optimálních řešení v uspokojivém čase. Za předpokladu neefektivního využívání nám ovšem ani samotný supervýkonný paralelní systém sám o sobě nezajistí požadované výsledky. Úzkým hrdlem současných, a lze předpokládal i budoucích, systémů je paměťový subsystém, který omezuje výkon jednotlivých výpočetních jednotek pouze na zlomek jejich maximálních možností.

Mezi hlavní problémy patří otázka, jak uložit matici v paměti počítače, abychom výpočet co možná nejvíce zefektivnili a mohli dosahovat vysokého výkonu. Kromě základních formátů pro uložení matice (COO, CSR, JDS apod.), bylo od počátku moderní éry paralelních počítačů publikováno několik desítek nových formátů. Hlavní náplní této práce je představit jednotlivé formáty, rozdělit je do skupin podle jejich hlavních rysů a popsat jejich principy. Dále pak popsat důvody a návaznosti vzniku daného formátu, provést jeho analýzu a určit vhodnost pro různé současné paralelní platformy.

První kapitola se věnuje obecné problematice řídkých matic a jejich ukládáním do paměti. Druhá kapitola obsahuje stručný přehled paralelních systémů, které s těmito maticemi pracují a provádí SpMV výpočty. Následující kapitoly se již zabývají konkrétními úložnými formáty, od základních až po ty nově vznikající. Kapitoly jsou členěny podle principů a vlastností jednotlivých formátů a obsahují popis a analýzu struktury úložných formátů. V závěru práce se nachází celkové zhodnocení vývoje vzniku nových formátů.

Nové paměťově efektivní formáty jsou v současné době stále ve vývoji. Hlavním cílem této práce je poskytnout čtenáři ucelený přehled a principy dosud publikovaných formátů pro ukládání řídkých matic v paměti počítače.

Kapitola **1**

Základní problematika

1.1 Řídká matice

Matice $A_{m,n}$ $(m \times n)$, kde $m, n \in N^{+}$, je schématické uspořádání prvků do m řádek a n sloupců.

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & \cdots & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & \cdots & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & & & \vdots \\ \vdots & \vdots & & a_{i,j} & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & \cdots & \cdots & a_{m,n} \end{pmatrix}$$

Každý prvek matice $a_{i,j}$ je jednoznačně identifikován indexy (i, j), kde i je index řádku a j je index sloupce. Matice může mít následující tvar:

- $m \neq n$ – obdélníková matice
- $m = n$ – čtvercová matice

Pro jednoduchost bude v následujícím textu předpokládána čtvercová matice $A_{n,n}$ $(n \times n)$.

Speciálním typem matice A je matice značená jako A^T. Jedná se o transponovanou matici, kde jsou řádky přeuspořádány do sloupců a sloupce do řádků. Pro prvky matice A^T platí:

$$a^T_{i,j} = a_{j,i}$$

Prvky matice A se dělí do dvou skupin:

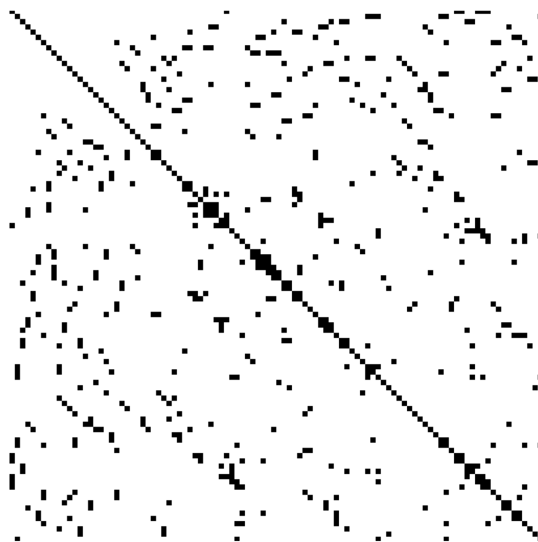
- **nenulový prvek** – takový prvek, pro který platí $a_{i,j} \neq 0$ (dále jako *nnz*)

- **nulový prvek** – takový prvek, pro který platí $a_{i,j} = 0$

Pokud počet *nnz* prvků dosahuje řádově n^2, jedná se o hustou matici. Tento text se zabývá opačným pólem, tedy řídkou maticí. Její definice se různí. Obecně lze za řídkou matici považovat matici, která má pouze malé procento *nnz* prvků.

Řídké matice se dělí na dva typy:

- strukturované
  - matice, kde nenulové prvky tvoří pravidelný vzor, typicky nenulové prvky tvoří malé diagonály nebo leží v blocích stejné velikosti
- nestrukturované
  - matice, kde umístění nenulových prvků je nepravidelné a netvoří žádný pravidelný vzor



Obrázek 1.1: Příklad řídké matice, kde černé čtverečky představují nenulové prvky

1.2 Násobení řídké matice s vektorem (SpMV)

Základní algebraickou operací, na kterou se nově vznikající úložné formáty zaměřují, je násobení vstupní matice $A_{n,n}$ s vstupním vektorem $\vec{x} = (x_1, x_2, \ldots, x_n)$ za účelem výpočtu výstupního vektoru $\vec{y}$. Vztah lze definovat jako:

$$\vec{y} = A\vec{x}$$

Samotné násobení je jednoduché, každý řádek matice A je skalárně vynásoben se vstupním vektorem $\vec{x}$. Princip je zobrazen na následujícím příkladu:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 \\ a_{3,1}x_1 + a_{3,2}x_2 + a_{3,3}x_3 \end{pmatrix}$$

Analogicky je prováděn výpočet SpMTV, kde je jako vstup transponovaná matice A^T.

Výkon SpMV závisí na mnoha parametrech [29]:

- zvolený úložný formát
- využití hardwarové architektury
- kvalita algoritmu, který pracuje s úložným formátem
- kvalita optimalizace pro danou architekturu
- kvalita paralelizace
- způsob reprezentace dat v počítači

1.3 Ukládání řídkých matic

Matice je řídká, pokud je procento nulových prvků nebo jejich distribuce taková, že lze jejich přítomnost ekonomicky využít. Jinými slovy pokud dokážeme využít nulové prvky k ušetření výpočetního času nebo paměti (obvykle obojího), je matice řídká [30].

Idea ukládání řídkých matic spočívá v tom, že nulové prvky není třeba ukládat do paměti. Reprezentace matice jako $2D$ pole značně snižuje výpočetní efektivitu a omezuje velikost výpočetního problému. Řídké matice, které jsou doporučeny k testování formátů pro ukládání řídkých matic (viz. [29]), mohou mít počet řádků a sloupců v řádech miliónů, počet nenulových prvků však u většiny z nich nepřesahuje jedno procento. Pro představu, matice o jednom milionu řádků a sloupců, kde jeden prvek má velikost 10B (i s řádkovým a sloupcovým indexem), by potřebovala k uložení 10TB prostoru.

Při provádění SpMV je třeba znát řádkový a sloupcový index nenulových prvků. K tomu je třeba definovat datovou strukturu, která se o ukládání informací a hodnot nenulových prvků postará.

Paměťová úspora není jediným cílem, kterého se autoři úložných formátů snaží dosáhnout. Přesun dat z hlavní paměti na výpočetní jednotku limituje ve velké míře jeho výkon. Vyššího výpočetního výkonu lze tedy dosáhnout snížením množství přenášených dat skrz paměťovou hierarchii. Roli může také hrát uspořádání dat v paměti a jejich efektivní využití při paralelním zpracování.

1.4 Moderní úložné formáty

S příchodem éry moderních paralelních výpočetních systémů se značně rozšířil výzkum v oblasti úložných formátů pro řídké matice. Vzniklo několik desítek nových formátů, jejich přehled je uveden v tabulce 1.1. Těmito uvedenými formáty se také zabývá tato práce. Nelze jednoznačně říct, že se jedná o všechny formáty, dalších nových, zde nezveřejněných, formátů může být stále hodně, bude se ale většinou jednat o různé lehčí modifikace formátů popisovaných v této práci.

Tabulka 1.1: Seznam moderních úložných formátů (zdroj: [29])

název	**autor a publikace**
Adaptive-Blocking Hierarchical Storage Format (ABHSF)	Langr a jiní [17], [31]
Adaptive ELL (AdELL)	Maggioni a Berger-Wolf [5]
Adaptive row-grouped CSR (ArgCSR)	Heller a Oberhuber [10]
Advanced Hierarchical (AH)	Šimeček a jiní [18]
Arithmetical Coding Based (ACB)	Šimeček a jiní [32], [27]
Basic Hierarchical (BH)	Šimeček a jiní [18]
Bisection ELLPACK (BiELL)	Zheng a jiní [14]
Bisection JAD (BiJAD)	Zheng a jiní [14]
Blocked-Based Compression Storage (BBCS)	Stathis [4]
Blocked ELLPACK (BELLPACK)	Choi a jiní [6]
Blocked Transposed Jagged Diagonal Storage (BTJDS)	Abu-Sufah a Karim [33]
Cache-adaptive heuristics-based register blocking (CARB)	Tvrdík a Šimeček [26]
Compressed Multi-Row Storage (CMRS)	Koza a jiní [34]
Compressed Sparse Blocks (CSB)	Buluc a jiní [35]
Compressed Sparse eXtended (CSX)	Karakasis a jiní [23], [36], [37]
COOCOO256	Šimeček a jiní [38]
COOCSR256	Šimeček a jiní [38]
CSR Delta Units (CSR-DU)	Kourtis a jiní [24], [39], [40]
CSR Values Indirect (CSR-VI)	Kourtis a jiní [24], [39], [40]
CSR with Segmented Interleave Combination (SIC)	Feng a jiní [41]
Decomposed Block CSR (BCSR-DEC)	Karakasis a jiní [42]
Decomposed Block Compressed Sparse Diagonals (BCSD-DEC)	Karakasis a jiní [42]
Delta-coded Sparse Row (DCSR)	Willcock a Lumsdaine [22]
ELLPACK-R (ELL-R)	Vázquez a jiní [43]
ELLPACK-RP	Cao a jiní [44]
ELLPACK Sparse Block (ESB)	Liu a jiní [3]
ELLR-T	Vázquez a jiní [45]
Expanded Compressed Sparse Block (eCSB)	Tao a jiní [46]
Extended Sparse Block Compressed Row Storage (SBCRSx)	Smailbegovic a jiní [20]
Hierarchical Sparse Matrix (HiSM)	Stathis a jiní [4], [19]
Hybrid EELPACK/COO (ELL/COO, HYB)	Bell a Garland [47]
Improved Compressed Sparse Row (ICSR)	Yang a jiní [11]
Minimal Binary Tree (MBT)	Šimeček a jiní [32], [27]
Minimal Quadtree (MQ/MQT)	Šimeček a jiní [28]
Padded Jagged Diagonal Storage (pJDS)	Kreutzer a jiní [15]
Pattern-Based Representation (PBR)	Belgin a jiní [21], [48]
Recursive CSR/CSC (RCSR/RCSC)	Martone a jiní [49], [50], [51], [25]
Row-grouped CSR (RgCSR)	Oberhuber a jiní [9]
Row Pattern CSR (RPCSR)	Willcock a Lumsdaine [22]
Row Segmented Diagonal (RSDIAG)	Vuduc [8]
SELL-C-σ	Kreutzer a jiní [16]
Sliced ELLPACK	Monakov a jiní [12]
Sliced ELLR-T	Dziekonski a jiní [52]
Specified Machine Parameters Using Register Blocking (SMURB)	Šimeček a jiní [53]
Transpose Jagged Diagonal Storage (TJDS)	Montagne a Ekambaram [54]
Unaligned Block CSR (UBCSR)	Vuduc a Moon [7]

Kapitola **2**

Paralelní systémy

Následující kapitola se zabývá paralelními systémy, které lze při výpočtu SpMV efektivně využít. Moderní HPC systémy jsou založeny na hybridní distribuované sdílené paměti, tzn. že na úrovni HW se fakticky jedná o systém s distribuovanou pamětí, tato skutečnost je však uživateli skryta a paměť se tváří jako globální sdílený paměťový prostor. Výpočetní a paměťové uzly jsou v HPC systémech většinou propojeny vysokorychlostními síťovými subsystémy. Mapování výpočetního problému je tedy nutné zařídit na aplikační úrovni. Výpočet se skládá z následujících fází:

- rozdělení vstupních dat mezi výpočetní uzly
- každý uzel provede lokální výpočet svých přidělených dat
- paralelní redukce mezivýsledků do konečného výsledku

Druhý krok je plně nezávislý v rámci každého výpočetního uzlu. Tato práce je zaměřena právě na efektivní využití úložných formátů pro řídké matice v rámci lokálního výpočtu. Pokud tedy v následujícím textu mluvíme o SpMV, předpokládáme SpMV v rámci jednoho výpočetního uzlu.

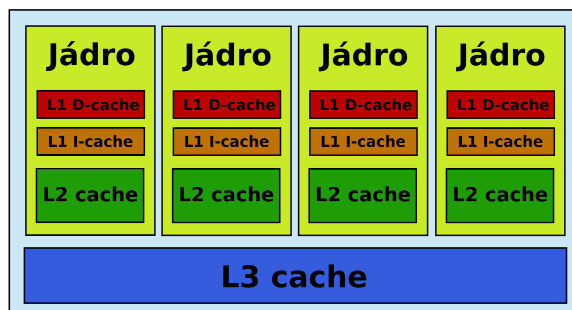
2.1 Základní typy procesorů

Současné procesory lze z hlediska paralelizace rozdělit do dvou skupin [1]:

- **vícejádrové** (*multi-core*)
- **mnohojádrové** (*many-core*)

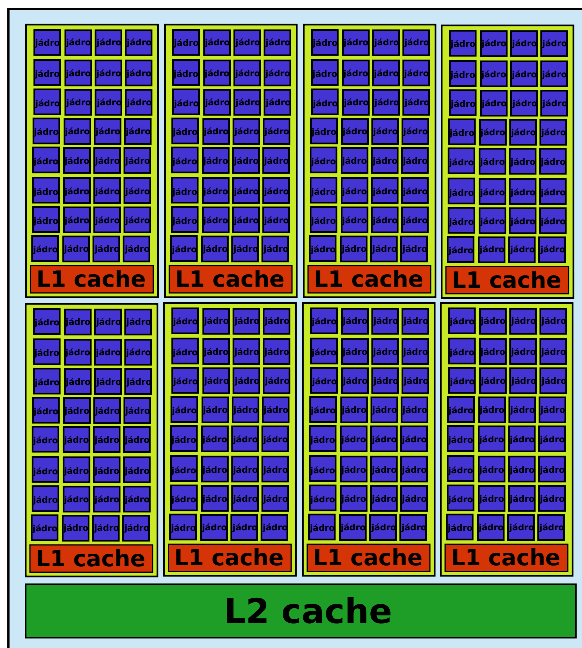
Jako vícejádrové jsou označovány klasické procesory, složené z menšího množství plnohodnotných jader. Vyznačují se poměrně složitou řídící logikou a obsahem větších hierarchicky poskládaných cache pamětí. Ukázka takového procesoru je na obrázku 2.1, nachází se na ní čtyři plnohodnotná jádra, z nichž

každé má vlastní L1 data a instrukční cache a L2 cache. L3 cache je pro všechna jádra společná. To značně limituje možná rozšíření procesoru o více dalších jader, s přibývajícím počtem bude efektivita klesat, protože jádra budou soupeřit o společnou L3 cache.



Obrázek 2.1: Ukázka vícejádrového procesoru

Za mnohojádrové systémy lze označit současné grafické procesory, které zpravidla obsahují stovky až tisíce jednoduchých výpočetních jader. Řídící logika je v jejich případě zjednodušená a cache paměti menší (viz. obrázek 2.2). Detailnější popis je uveden v sekci 2.3.



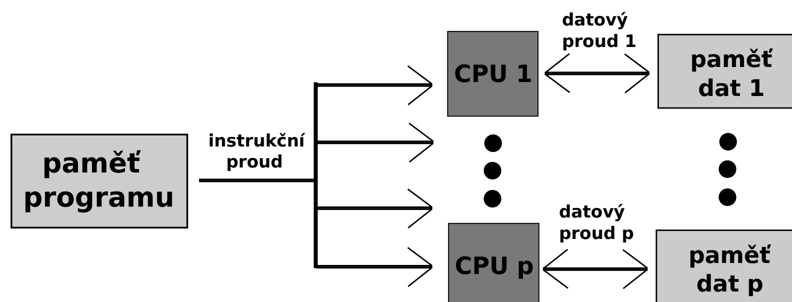
Obrázek 2.2: Zjednodušená ukázka mnohojádrového procesoru

2.2 Architektura SIMD

Podle Flynnova rozdělení jsou paralelní architektury děleny na následující kategorie:

- **Single Instruction Single Data (SISD):** sériové zpracování dat podle jednoho programu (typicky počítač von Neumannova typu)
- **Single Instruction Multiple Data (SIMD):** paralelní zpracování stejné instrukce, každý procesor pracuje nad různými daty
- **Multiple Instruction Single Data (MISD):** paralelní zpracování různých instrukcí z různých programů nad stejnými daty
- **Multiple Instruction Multiple Data (MIMD):** víceprocesorový systém, každý procesor má vlastní program a vlastní data

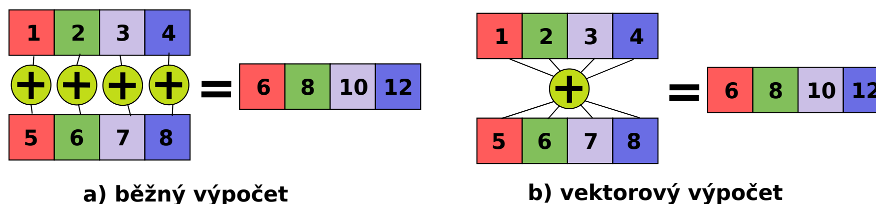
Pro paralelní řešení SpMV je klíčová architektura SIMD. Jedna instrukce je synchronně vykonávána množinou procesorů nad různými daty. Princip SIMD je zobrazen na obrázku 2.3. Tato architektura je vhodná pro zpracování datově paralelních úloh, tedy že lze úlohu rozdělit na podmnožiny nezávislých dat, které mohou být samostatně paralelně zpracovávány [1].



Obrázek 2.3: SIMD architektura (zdroj: [1])

2.2.1 Vektorové procesory

SIMD přístup je též využit při vektorovém zpracování, které operuje paralelně nad více datovými hodnotami uloženými ve vektorových registrech. Procesory obsahují kromě klasických jednotek jednotku SIMD, která tyto vektorové operace obstarává. Procesorové instrukční sady bývají rozšířeny o speciální vektorové operace, ty nejběžněji používané jsou MMX (*MultiMedia eXtension*) a SSE (*Streaming SIMD Extensions*) [1].



Obrázek 2.4: Rozdíl mezi běžným a vektorovým výpočtem

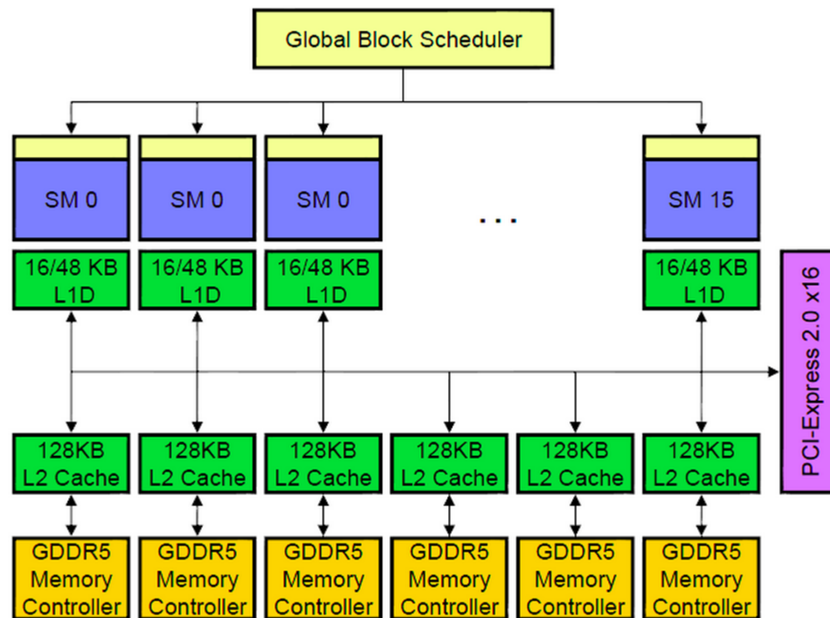
Na obrázku 2.4 je vidět rozdíl mezi běžným a vektorovým zpracováním. Zatímco u běžného výpočtu je třeba pro každou dvojici vstupních prvků použít jednu aritmetickou operaci (tedy provádět výpočet postupně v pipeline za sebou), vektorové zpracování nabízí možnost provedení jedné aritmetické operace nad více prvky paralelně v jednom taktu (počet zpracovaných hodnot závisí na velikosti SIMD jednotky).

Toto zpracování je možné vzhledem k výpočtu SpMV efektivně využít při násobení prvků jednoho řádku matice s prvky vstupního vektoru, je však třeba mít data v paměti ideálně uspořádána.

2.3 Architektura grafických procesorů

Grafické procesory (GPU) nabízí uživateli řádově až tisíce vláken, kde každé vlákno může operovat nad nezávislými datovými prvky. Vlákna jsou seskupována do bloků vláken, tyto bloky lze poté seskupovat do logických mřížek (v závislosti na řešeném problému). Architektura je označována jako SIMT (*Single Instruction Multiple Thread*), jedná se o specializovanou SIMD architekturu.

Základní jednotkou GPU jsou multiprocesory neboli SM (*Streaming Multiprocesor*). Každý SM obsahuje svoji logiku pro načítání, dekódování a vykonávání instrukcí a je schopný zpracovávat jeden nebo více bloků vláken. Rovnoměrné rozložení bloků mezi SM zajišťuje globální plánovač (tzv. *Global Block Scheduler*). Samotný SM si poté dělí bloky vláken na tzv. *warpy*, tj. skupinu 32 vláken, u kterých probíhá paralelní zpracování. *Warpy* jsou vykonávány po jednom, o plánování se stará plánovač (tzv. *warp scheduler*), který je součástí každého SM, čekající *warpy* jsou odloženy do instrukčního bufferu. *Warp* je plánovačem vybrán pouze v případě, že jsou jeho operandy načteny a potřebná paměť dostupná. Tento přístup může efektivně zakrýt latenci způsobenou přístupem do paměti [1]. Architektury jednotlivých grafických karet se mohou mírně lišit, SM mohou být např. seskupeny do clusterů, popsaný základ ale zůstává stejný, příklad karty NVIDIA GF100 Fermi je zobrazen na obrázku 2.5.



Obrázek 2.5: Ukázka architektury NVIDIA GF100 Fermi (zdroj: [2])

Z pohledu paměťového modelu má každé vlákno přístup do několika druhů pamětí, dostupné paměti jsou [1]:

- **Globální paměť:** největší a zároveň nejpomalejší, umístěna mimo GPU, přístup trvá řádově stovky hodinových cyklů, obsahuje subčásti určené jen pro čtení (konstantní paměť a paměť textur)
- **Registry:** nejrychlejší, přístup trvá jeden hodinový cyklus, privátní pro každé vlákno
- **Lokální paměť:** privátní paměť pro každé vlákno, nevýhodou je umístění v globální paměti
- **Sdílená paměť:** primárně pro výměnu dat mezi vlákny v rámci bloku, lze využít i jako cache

Z hlediska výpočtu SpMV je nejzajímavější sdílená paměť, která by měla být co nejlépe využita. Její fyzické umístění se nachází přímo na čipu GPU v rámci každého SM. Přístup trvá čtyři hodinové cykly, což je oproti globální paměti značný rozdíl.

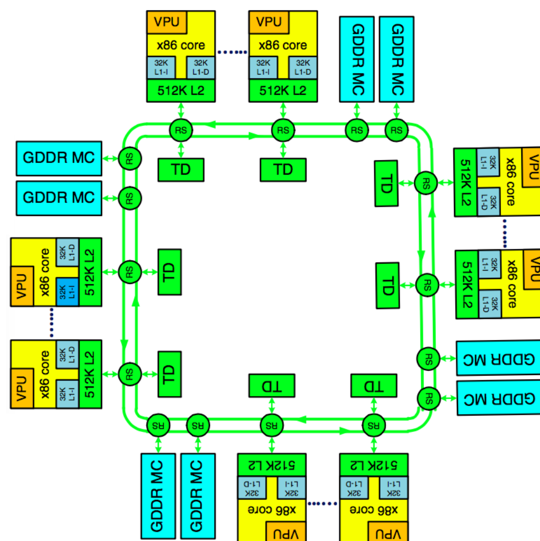
2.4 Moderní koprocesory

Nově vyvíjené mnohojádrové čipy se výhradně specializují na urychlení výpočtů a snaží se plně konkurovat současným grafickým procesorům. Hlavním iniciátorem v této oblasti je společnost Intel, která se několik let snažila vyvinout konkurenceschopný koprocesor.

Jako jeden z prvních byl v roce 2012 představen koprocesor Intel Xeon Phi s kódovým označením Knights Corner, který měl následující specifikace [3]:

- 57 až 61 jader s frekvencí až $1,1$ GHz
- GDDR5 paměti o možných kapacitách 3, 6 nebo 8 GB s frekvencí od 5 do 5.5 GHz.
- L1 cache až $1,9$ MB
- L2 cache až $30,5$ MB

Z důvodu zpětné kompatibility a snazšímu rozšíření byla zachována architektura jader x86. Každé jádro obsahuje vlastní L1 a L2 cache a 512-bitovou SIMD jednotku pro vektorové zpracování (VPU).



Obrázek 2.6: Architektura Knights Corner (zdroj: [3])

Na obrázku 2.6 je zobrazena architektura uvnitř koprocesoru. Jednotlivá jádra jsou propojena širokopásmovou obousměrnou sítí v kruhovém uspořádání. Koherenci[1] L2 cache zajišťují tzv. *tag directories* (TD) [55].

[1] Paměťová koherence znamená, že ve všech paměťových jednotkách jsou stejná data.

Hlavní výhoda této architektury by se měla skrývat ve velikosti cache pamětí a vektorového zpracování na každém jádře.

Intel později přináší na trh v rámci rodiny Xeon Phi další generace koprocesorů, a to:

- Knights Landing
- Knights Hill
- Xeon Phi

Jejich detailnější popis je nad rámec této práce, jako referenční model postačí generace Knights Corner.

Kapitola **3**

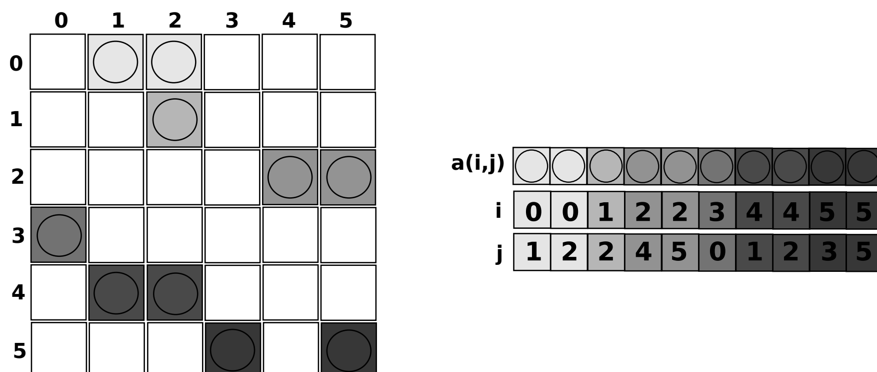
Základní formáty

3.1 Coordinate Format (COO)

Nejintuitivnější formát pro ukládání řídkých matic. Je postaven na principu jednoduchého uložení všech potřebných informací o *nnz* prvku [56]. Pro každý *nnz* prvek $a_{i,j}$ v matici A jsou uloženy následující informace:

- index řádku i
- index sloupce j
- hodnota prvku $a_{i,j}$

K uložení v paměti počítače stačí tři pole o velikosti počtu nenulových prvků. Princip uložení je zobrazen na obrázku 3.1. Tento formát se dobře hodí pro velmi řídké matice se spoustou prázdných řádků. Požadavky na prostor v paměti jsou přímo úměrné počtu *nnz* prvků.



Obrázek 3.1: Příklad reprezentace řídké matice ve formátu COO

Mezi hlavní výhody patří rychlý a přímý přístup ke konkrétní hodnotě. Formát je jednoduchý z hlediska implementace a je snadno převoditelný do/z formátu CSR/CSC. K dalším výhodám patří nulová režie pro výpočty nad prázdnými řádky.

Formát s sebou nese ale spoustu nevýhod. Tou hlavní je jeho neefektivnost pro výpočet SpMV. Jedním z faktorů, který může ovlivnit výkon SpMV je ten, že formát implicitně nezahrnuje informace o pořadí prvků uložených v paměti, to může vést k nutnosti častého střídání hodnot vstupního vektoru, při paralelním zpracování pak k zajištění atomicity při zápisu do výstupního vektoru [45]. Dalším problémem je jeho celková paměťová náročnost, která je, oproti dále popsaným formátům, stále vysoká.

3.2 Compressed Row Storage (CSR/CRS)

V praxi spolu s jeho modifikací CSC (3.3) nejpoužívanější formát pro ukládání řídkých matic. Nedělá žádné předpoklady o řídkosti matice a rozložení *nnz* prvků. Neukládá žádné přebytečné prvky navíc.

V případě nesymetrické matice jsou k uložení struktury potřeba tři pole:

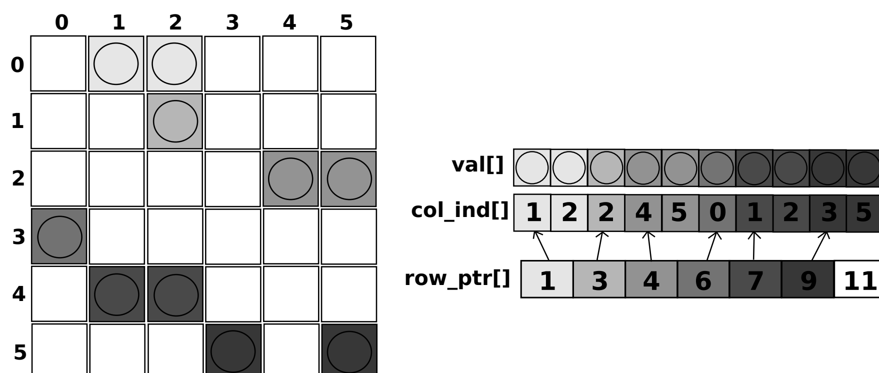
- *val*[] pole obsahující hodnoty prvků, velikost rovna počtu *nnz*
- *col_ind*[] pole se sloupcovými indexy pro každý *nnz*, velikost stejná jako pro *val*[]
- *row_ptr*[] pole obsahující pointery do pole *val*[] (příp. *col_ind*[]), které určují start každého řádku, velikost n (v praxi pro snazší implementaci většinou $n+1$, poslední prvek je definován jako $row_ptr[n+1] = nnz+1$)

Pokud je matice symetrická, stačí ukládat pouze dolní/horní trojúhelníkovou část matice. V tomto případě ovšem platíme za uspořené místo komplikovanějšími algoritmy s odlišným přístupem k datům [56].

Úspora místa při uložení matice v CSR je značná, v praxi je většinou počet prvků $nnz \gg n$, není tedy třeba tolik paměti jako v případě formátu COO [57]. Hodnoty jsou po řádcích uloženy do souvislého paměťového úseku, nicméně formát vyžaduje pro každou skalární operaci SpMV nepřímý adresovací krok, což nečiní formát moc efektivním.

Implementace SpMV založené na CSR formátu mají z pohledu paralelního výpočtu SpMV mnoho nedostatků, které brání optimalizaci výkonu na SIMD procesorech či grafických kartách, hlavní nedostatky jsou následující [47], [45], [12], [9], [16]:

- použití jednoho vlákna/jednotky na jeden řádek
    - *val*[] a *col_ind*[] uloženy v paměti souvisle, ale vlákna souvisle nepřistupují (tzn. začátky řádků jsou v paměti daleko od sebe)



Obrázek 3.2: Řídká matice ve formátu CSR

  - časová ani prostorová lokalita není zajištěna v důsledku nepřímého adresování
  - hrozba nevyváženosti výpočtu (husté řádky oproti řádkům s minimálním počtem *nnz*)

- použití *warpu* (resp. SIMD) na jeden řádek
  - efektivní výkon vyžaduje průměrný počet *nnz* na řádek podstatně větší než šířka *warpu* (resp. SIMD)
  - nutná implementace paralelní redukce

CSR formát se stal základem pro velkou část moderních formátů, které se snaží redukovat tyto hlavní nedostatky v rámci paralelního SpMV.

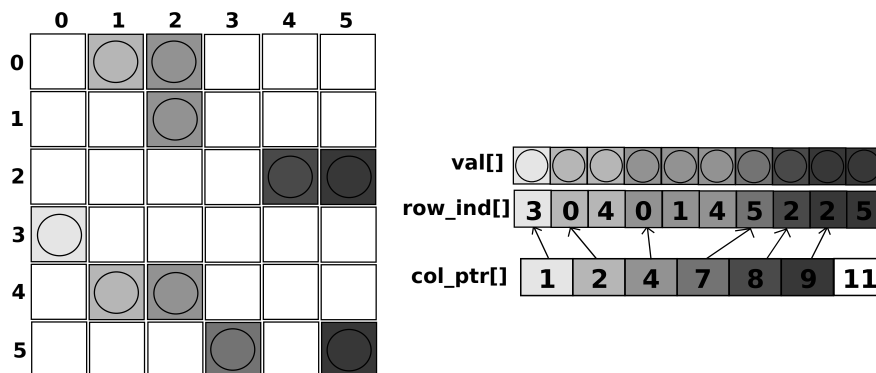
3.3 Compressed Column Storage (CSC/CCS)

Analogický formát k formátu CSR. Někdy nazývá také jako Harwell-Boeing format. Jedním z hlavních důvodů vzniku je použití ve specifických programovacích jazycích, zvláště pak FORTRAN [56].

Odlišnost od CSR je v opačném uspořádání prvků. CSC formát pro matici A je CSR formát pro matici A^T.

Struktura je téměř identická, dochází pouze k prohození rolí mezi řádky a sloupci:

- *val*[] pole obsahující hodnoty prvků
- *row_ind*[] pole s řádkovými indexy pro každý *nnz* prvek
- *col_ptr*[] pole obsahující pointery na *nnz* prvky, které určují start každého sloupce



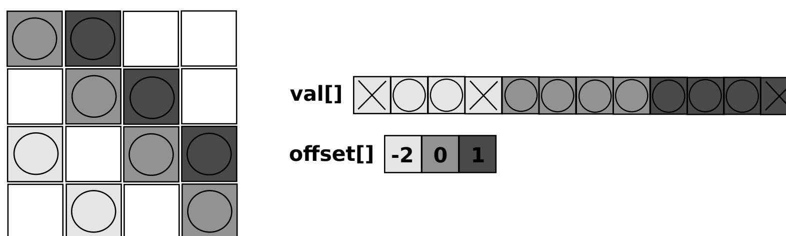
Obrázek 3.3: Řídká matice ve formátu CSC

3.4 Diagonal Format (DIAG)

Formát navržený speciálně pro matice, jejichž *nnz* prvky mají diagonální uspořádání. Není proto vhodný pro využití u všeobecných typů matic.

Struktura formátu vyžaduje pouze následující dvě pole [47]:

- *val*[] pole obsahující hodnoty prvků, velikost pole je rovna $n \times diag$, kde *diag* značí počet nenulových diagonál
- *offset*[] pole uchovávající posunutí každé diagonály oproti hlavní diagonále, velikost *diag*



Obrázek 3.4: Uložení řídké matice ve formátu DIAG

Příklad uložení je zobrazen na obrázku 3.4. Diagonály se ukládají za sebou postupně od nejmenší hodnoty *offset*. Hodnota *offset* může být negativní (diagonála se nachází pod hlavní diagonálou), pozitivní (nad hlavní diagonálou), či nulová (hlavní diagonála). Lze si všimnout, že formát ukládá i nulové prvky, a to v případě, pokud se v některé diagonále nachází alespoň jeden *nnz* prvek nebo je diagonála kratší než hlavní (což u čtvercové matice nastává pro všechny diagonály kromě hlavní).

Ukládání pomocných nul může potenciálně znamenat jednu z hlavních nevýhod tohoto formátu, zvláště v případě, kdy diagonála obsahuje velmi malý počet *nnz* prvků. Mezi výhody lze zařadit nepotřebnost ukládání řádkových a sloupcových indexů, souvislý přístup k datům je přesto zaručen. Podle hodnoty *offset* se snadno určí, která hodnota z vstupního vektoru $\vec{x}$ je právě potřeba.

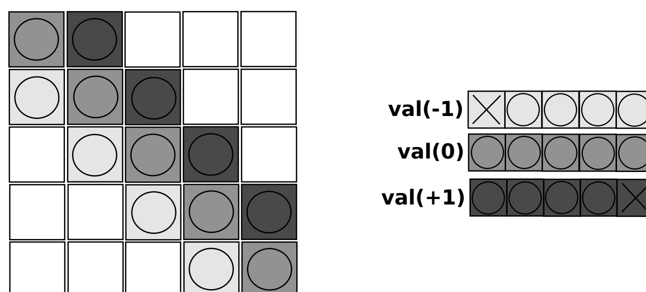
Paralelizace je intuitivní. Jedno vlákno (jednotka) bude přiřazeno konkrétnímu řádku, přes který bude iterovat. Přístup k hodnotám matice i vektoru $\vec{x}$ je tak souvislý.

3.5 Compressed Daigonal Storage (CDS)

Formát určený pro řídké matice, ve kterých jsou nenulové prvky zhuštěny okolo hlavní diagonály do tzv. pásu *nnz* elementů. Pás je specifikován svoji šířkou, která je určena parametry p a q, kde p a q je počet diagonálních pruhů směrem doleva, resp. doprava od hlavní diagonály. Celková šířka pásu $w = p + q + 1$ [56].

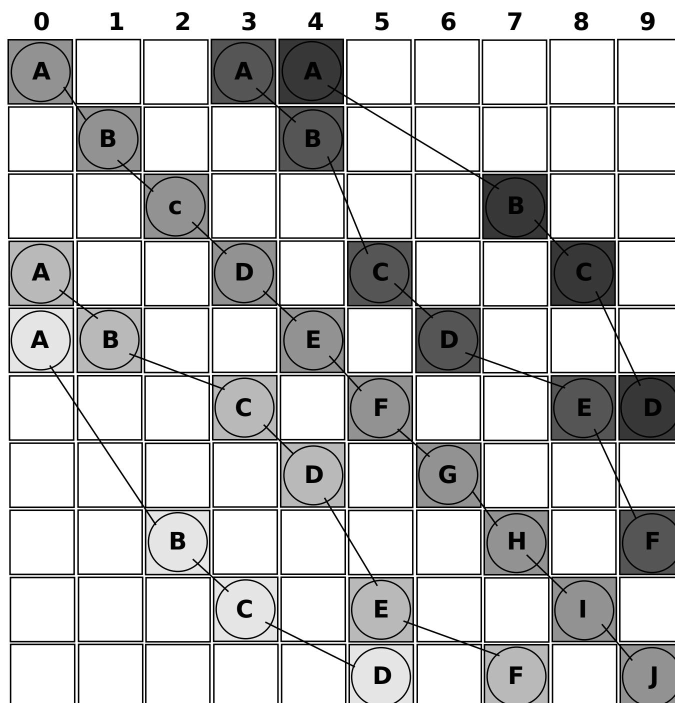
Diagonální pruhy se ukládají do jednoho pole $val[]$ postupně za sebou, princip lze vidět na obrázku 3.5. Velikost pole je $n \times w$. Lze si všimnout, že formát povoluje ukládání nulových prvků a dále prvků, jejichž indexy nekorespondují s indexy matice (ty můžou být považovány za nulové).

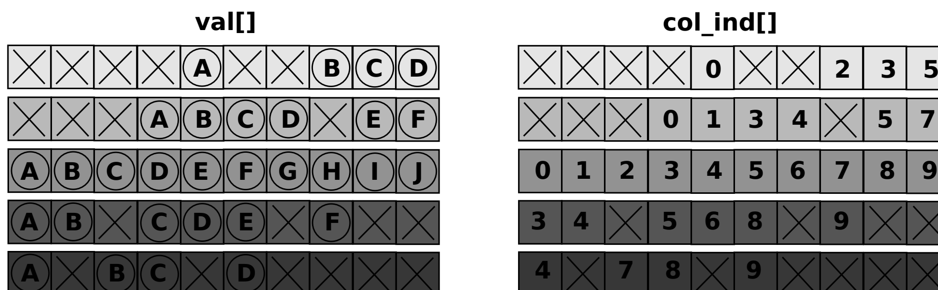
Výhodou formátu je eliminace indexových polí pro řádky a sloupce, na rozdíl od formátu DIAG nepotřebuje ani pole $offset[]$. V případě dostatečně velké hodnoty w je možné efektivní využití vektorového zpracování.



Obrázek 3.5: Princip ukládání diagonálního pásu ve formátu CDS

Použitelnost formátu je závislá na vstupní matici. Pokud některé řádky překročí ideální šířku diagonálního pásu, povede to k uložení přebytečného množství nulových prvků. Tento problém řeší zobecněná verze formátu (viz. [58]). Toto zobecnění je účinnější pro různé šířky pásma *nnz* prvků v jednotlivých řádkách. Princip je zobrazen na obrázku 3.6. Diagonální pruhy jsou sestavovány tak, aby pokryly *nnz* prvky a redukovaly zbytečné ukládání nul. Podmínkou je existence dalšího pole $col_ind[]$, které uchovává sloupcové indexy všech ukládaných prvků, velikost pole je stejná jako pro $val[]$.





Obrázek 3.6: Zobecnění formátu CDS

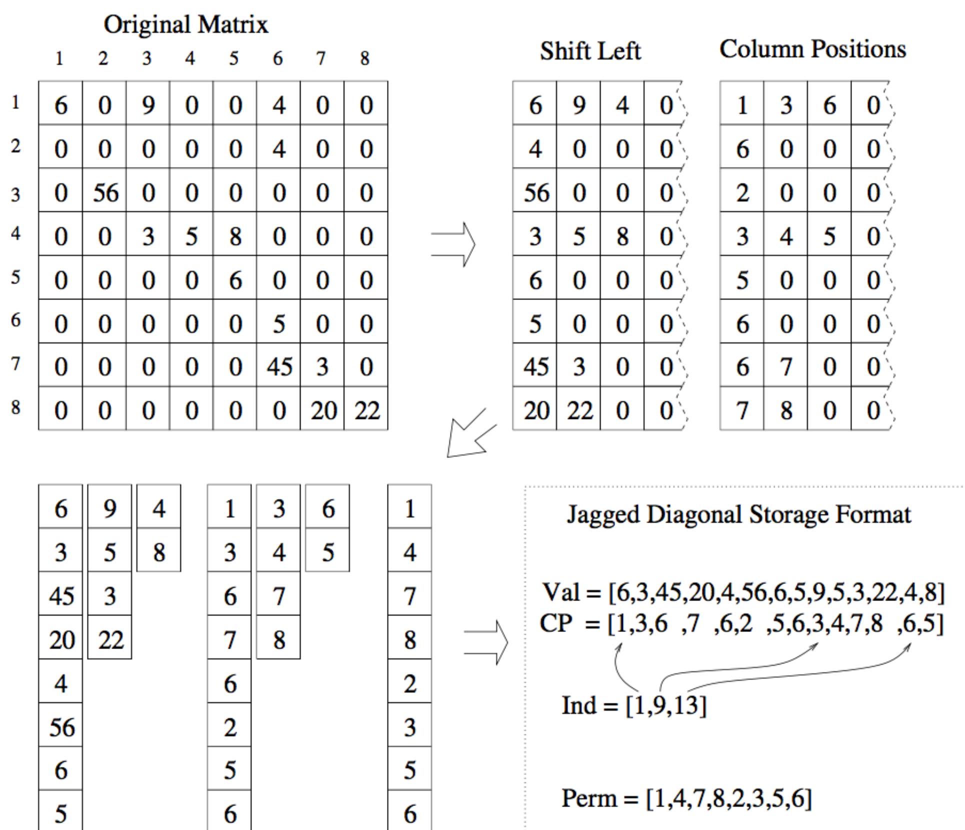
Ziskem může být snížení paměťových nároků a použitelnost na větší škále vstupních matic. SpMV bude ovšem výrazně zpomalovat fakt, že je nutné pracovat s polem sloupcových indexů, protože nová struktura neumožní jednoduše určit z pole *val*[] pozici prvku v konkrétní řádce.

3.6 Jagged Diagonal Storage (JDS/JAD)

Formát vycházející z principu přeuspořádání *nnz* prvků ve vstupní matici. Podobný formátu CDS, na rozdíl od CDS je ale prostorově efektivnější a

nevyžaduje strukturu nenulových prvků vstupní matice v diagonální formě. Struktura pro uložení *nnz* využívá principů CSR [56].

Po přeuspořádání vstupní matice vznikají tzv. „*jagged diagonals*“ (JD). Prvním krokem přeuspořádání je posunutí *nnz* směrem doleva. Počet JD je určen podle počtu *nnz* v nejhustším řádku. Řádky, které mají menší počet *nnz* jsou doplněny nulami. JD v tomto případě reprezentují jednotlivé sloupce, které se ukládají v paměti po sobě. Vznikají tak vektory, které se svoji délkou blíží hodnotě řádu matice. Může být značně nevyhovující, pokud se budou počty *nnz* v řádcích výrazně lišit. Z tohoto důvodu jsou řádky ještě seřazeny od nejpočetnějších až po řádky s nejmenším počtem *nnz*. Převod matice do JDS lze vidět na obrázku 3.7.



Obrázek 3.7: Přeuspořádání řídké matice do formátu JDS (zdroj: [4])

K uložení ve formátu JDS jsou zapotřebí následující pole [59]:

- *val*[] pole obsahující hodnoty prvků, velikost rovna počtu *nnz*
- *col_ind*[] pole se sloupcovými indexy pro každý *nnz*, velikost jako u *val*[]

- $jd_ptr[]$ pole obsahující pointer na nnz, který určuje start každé JD, velikost podle počtu JD
- $perm[]$ pole určující původní pozici každého řádku před permutací, velikost rovna n (počet řádek matice)

Formát nabízí výhodu v možnosti využití operací s velkými vektory, tedy ideální pro použití na vektorových procesorech. Nicméně výkon SpMV bude ovlivněn častým přístupem do paměti. Bude nutné často aktualizovat hodnoty vstupního vektoru $\vec{x}$, protože indexy vektoru nekorespondují s indexy jednotlivých JD. Každá výpočetní jednotka, počítající jeden řádek, může v jedné iteraci potřebovat různé části vstupního vektoru. Na tento problém se zaměřuje vylepšený formát TJDS (viz. 3.7).

3.7 Transpose Jagged Diagnal Storage (TJDS)

Formát vychází z původního JDS (3.6). Vznikl jako jeho upravená verze pro efektivnější výpočet SpMV. Nový formát má jako hlavní cíl redukovat přístupy do hlavní paměti, které výkon SpMV ve velké míře brzdí.

Principem je odlišné seskupování nnz prvků. Na rozdíl od JDS jsou nnz prvky seskupovány ve sloupcích, a to posunutím směrem nahoru. Seskupené sloupce jsou seřazeny podle hustoty zleva doprava (nejhustší vlevo). Ukládání do paměti probíhá po řádcích, každý takovýto řádek se nazývá *transpose jagged diagonal* (*TJD*) [54].

Struktura je podobná jako v případě JDS, pouze pole $col_ind[]$ nahrazuje $row_ind[]$ a pole $jd_ptr[]$ nahrazuje $tjd_ptr[]$. Pole $perm[]$ není pro TJDS nutné, stačí před výpočtem přeházet vstupní vektor $\vec{x}$ podle permutace sloupců vstupní matice. Příklad použití je zobrazen na obrázku 3.8.

Dostáváme tak formát, který lze efektivně využít např. pro vektorové zpracování bez zbytečných přístupů do hlavní paměti, protože sousední nnz prvky v paměti přesně korespondují se sousedními prvky permutovaného vektoru $\vec{x}$.

Formát nabízí i možnou paralelizaci na GPU mapováním vlákna na sloupec, kde si každé vlákno drží prvky vektoru $\vec{x}$, které potřebuje pro své sloupce. Problém je, že na každém řádku pracuje více vláken, a je tedy třeba zajistit atomicitu při zápisu do výstupního vektoru, případně implementovat redukci mezivýsledků každého vlákna. Obojí způsobuje velkou režii navíc. Redukce by musela být prováděna ve sdílené paměti, která je ovšem moc malá. Na tento problém se dále zaměřuje formát BTJDS 4.8.

3.8 Skyline Storage (SKS)

Formát SKS byl implementován hlavně za účelem použití v přímých metodách pro řešení lineárních systémů (převážně Gaussova eliminace) [56].

Principem je dekompozice matice A na tři části[1] [8]:

- diagonála (uložena do pole $diag[]$)
- dolní trojúhelníková matice (uložena ve formátu CSR)
- horní trojúhelníková matice (uložena ve formátu CSC)

V případě symetrické matice lze ukládat pouze dolní trojúhelníkovou matici.

3.9 ELLPACK Format

Formát vznikl za účelem komprese řídké matice pro využití při řešení velkých řídkých lineárních systémů za pomoci ITPACKV (viz. [60]) na vektorových počítačích [45]. Jedná se o odlehčenější verzi formátu JDS (viz. 3.6).

Struktura formátu se skládá ze dvou polí:

- $val[]$ pole obsahující hodnoty prvků, velikost rovna $n \times nnz_{max}$, kde nnz_{max} značí počet nnz v nejhustším řádku
- $col_ind[]$ pole se sloupcovými indexy pro každý nnz

Nároky na paměť se tedy odvíjí od řádku s největším počtem nnz prvků. Ostatní řádky jsou poté doplněny nulovými hodnotami. Na rozdíl od JDS již neprobíhá řazení řádků podle hustoty. Díky zhuštění nnz prvků lze ELLPACK využít na vektorových procesorech či pro GPGPU.

Při použití na GPGPU, za předpokladu mapování jednoho vlákna na jeden výstupní prvek spolu s ukládáním prvků matice A ve sloupcovém pořadí, nabízí ELLPACK několik výhod. Např. jednotný přístup do paměti, tzn. že vlákno t přistupuje k prvku v t. řádku jako $A[t + kn]$, kde $0 \leq k < nnz_{max}$, pak vlákno $t+1$ bude přistupovat k prvku v paměti následujícím, tedy $A[(t+1) + kn]$. Další výhodou je nepotřebnost synchronizace, každé vlákno se stará o jeden výstupní prvek, tím pádem nevznikají žádné datové závislosti [45].

Jelikož se ale jedná o základní návrh, obsahuje formát velké množství nedostatků, které mu nedovolují praktickou využitelnost [34]. Kromě velkých paměťových nároků lze narazit na extrémní případ, kdy vstupní matice bude obsahovat byť jen jedinou hustou řádku, to povede na uložení spousty zbytečných nul a nevyváženosti při paralelním zpracování. Při pokusu o odstranění těchto problémů vzniklo několik dalších formátů, které principiálně staví na základním ELLPACKu. Tyto formáty budou představeny v kapitole 6.

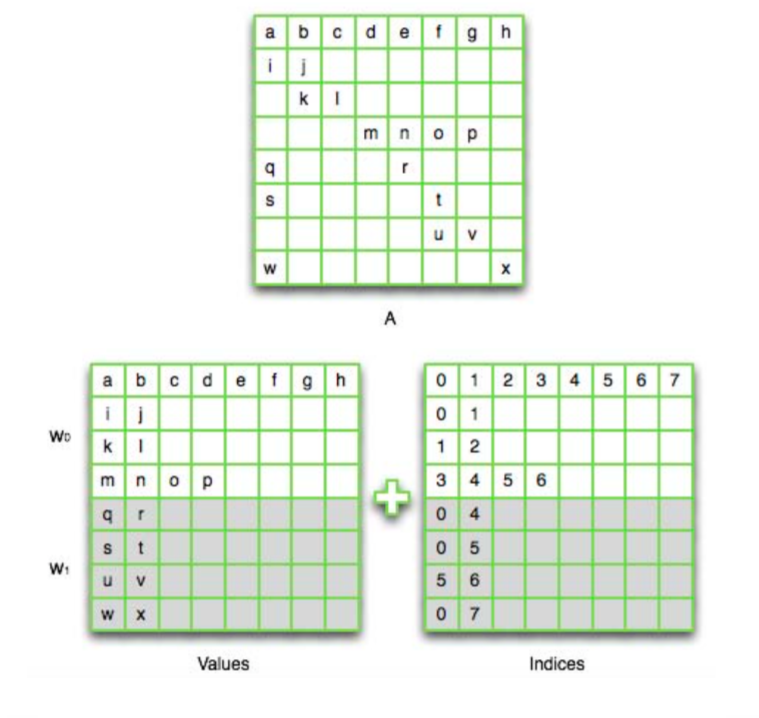
[1] V různých zdrojích se struktura formátu liší, zde je prezentována forma, která se nejčastěji využívá u přímých metod pro řídké matice.

a) originální vstupní matice a vektor

b) posunutí nenulových prvků směrem nahoru

c) přeskupení sloupců a prvků vstupního vektoru

Obrázek 3.8: Přeuspořádání vstupní matice a vektoru do formátu TJDS



Obrázek 3.9: Příklad uložení a rozdělení matice ve formátu ELLPACK (zdroj: [5])

Kapitola 4

Registrově-blokové formáty

Hlavním cílem formátů popisovaných v následující kapitole je snížit datový přenos mezi hlavní pamětí a výpočetní jednotkou. Vstupní matice je typicky rozdělena do bloků, které jsou v paměti uloženy za sebou. Důsledkem je rozdělení výpočetního problému na podproblémy, které lze pak řešit efektivněji. Registrově-blokovým formátem se může v podstatě stát jakýkoliv základní i pokročilý formát.

4.1 Blocked CSR/CSC (BCSR/BCSC)

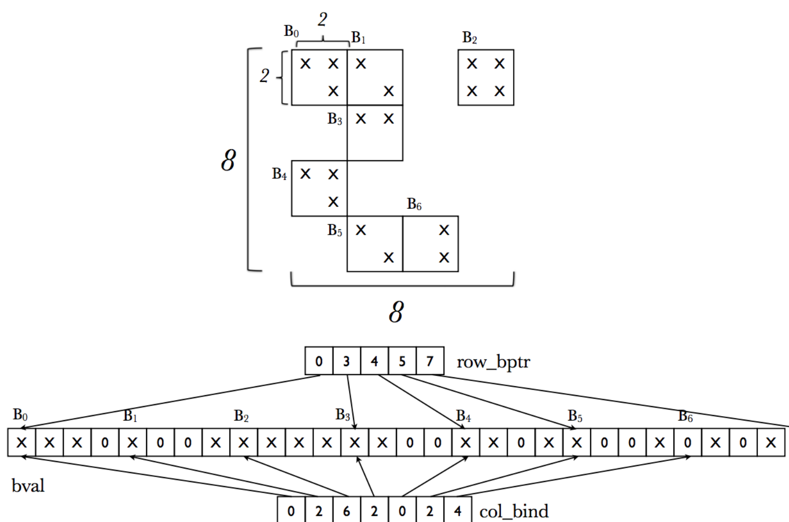
Formát CSR, který byl upraven pro efektivnější využití dat uložených v rámci procesorové paměti (registrech) [35]. Matice A je rozdělena do $\frac{n}{rw} \times \frac{n}{cl}$ bloků o velikosti $rw \times cl$. Úložné schéma pro nejvyšší úroveň je ve formátu CSR. Jednotlivé nenulové bloky[1] jsou jako husté matice uloženy za sebou v paměti. Princip uložení je zobrazen na obrázku 4.1.

Struktura BCSR je stejná jako CSR, liší se pouze v hodnotách, které jsou ukládány [56]:

- $block_col[]$ pole sloupcových indexů na první prvek nenulového bloku (tedy na pozici $[0, 0]$ v rámci každého bloku)
- $block_ptr_row[]$ pole pointerů, kde každý pointer ukazuje na první nenulový prvek v rámci blokového řádku
- $val[]$ pole hodnot každého bloku uložené v paměti po blocích v řádkovém pořadí

Využití vektorizace a paralelního zpracování je možné, nicméně nedosahuje optimálních výsledků (viz. [6]).

[1]Nenulový blok je blok, který obsahuje alespoň jeden nenulový prvek.



Obrázek 4.1: BCSR struktura (zdroj: [6])

4.2 Specified Machine parameters Using Register Blocking (SMURB)

Formát, který hledá optimální konfiguraci bloků s minimální paměťovou náročností. Jeho výhodou je kombinování všech typů bloků (horizontální, vertikální, diagonální).

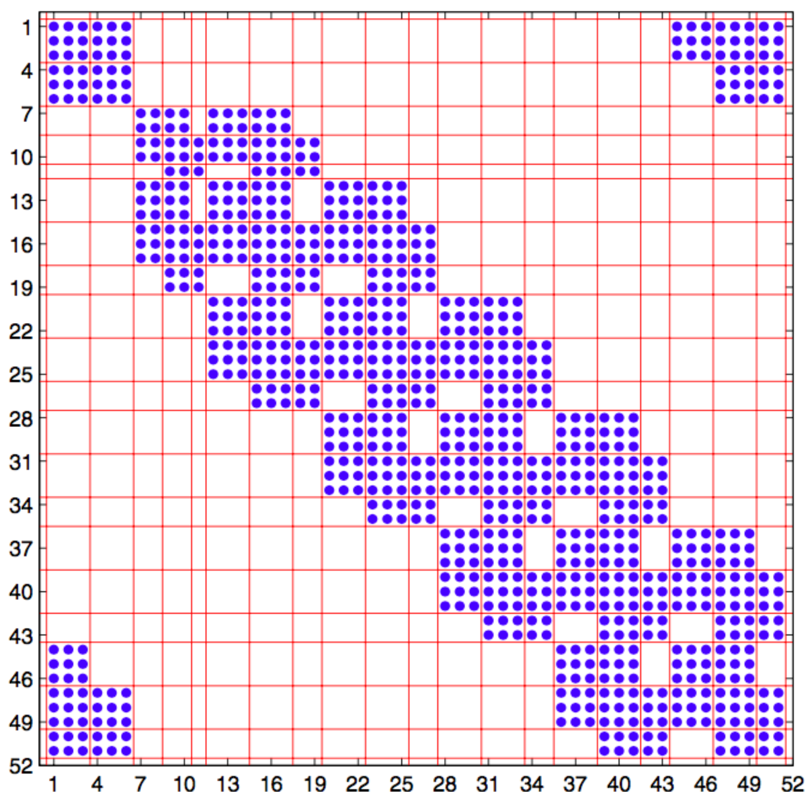
Vstupní matice A je rozdělena do sady čtvercových regionů. Region je dočasná datová struktura, která reprezentuje submatici matice A. Volba velikosti regionu by měla být volena s ohledem na prostorovou složitost dočasných datových struktur a pravděpodobností nalezení vhodných bloků *nnz* elementů v regionu [53].

Kromě parametru pro velikost region je nutné zvolit ještě mez, která bude určovat minimální počet *nnz*, pro který má smysl hledat optimální konfiguraci. V případě, že region obsahuje méně *nnz* než určená mez, uloží se region ve formátu CSR. V opačném případě proběhne výpočet prostorové náročnosti jednotlivých typů bloků (výpočet viz. [53]), v podstatě se jedná o zkoušku všech možností a následným výběrem té nejefektivnější.

4.3 Unaligned Block CSR (UBCSR)

Formát vychází z blokového CSR (BCSR viz. 4.1), jehož největším nedostatkem je ukládání zbytečných nulových prvků a následná režie při práci s nimi. Nový formát se soustředí hlavně na ušetření paměťových nároků a efektivitu ukládání, samotný výkon SpMV nemusí být výrazně lepší [7].

Princip je rozdělit matici A na s submatic $A_1 + A_2 + \ldots + A_s$, kde každá A_i může mít různou velikost.



Obrázek 4.2: Rozdělení matice do bloků pro potřeby UBCSR formátu (zdroj: [7])

Rozdělení matice probíhá nejprve po řádkách. Princip si lze jednoduše představit, řádek převedeme na bitový vektor, ve kterém 1 znamená umístění *nnz* prvku a 0 umístění nulového prvku, pokud se vektor následujícího řádku shoduje s předchozím řádkem, patří řádky do jednoho segmentu, v případě, že se vektor liší, začíná nový segment. Poté se pokračuje stejným způsobem, ale po sloupcích. Po rozdělení vzniká podobná struktura disjunktních submatic jako na obrázku 4.2.

Kvůli nezarovnaným blokům je nutné rozšířit strukturu BCSR o pole *A_row_ind*[], které obsahuje indexy začátků řádkových segmentů[1].

[1]Podle analýzy zveřejněné struktury by místo pole *A_row_ind*[] mělo stačil pole *block_ptr_row*[] z BCSR, které obsahuje pointer na první *nnz* prvek v rámci segmentu, jelikož bloky jsou v rámci řádků zarovnané.

4.4 Blocked-Based Compression Storage (BBCS)

Formát BBCS byl navržen pro řešení obecně známých problémů ukládání řídkých matic, mezi hlavní potíže patří:

krátký vektor většina matic má malý počet *nnz* v každém řádku/sloupci (při vektorovém zpracování značná režie)

indexovaný přístup pozice každého *nnz* ukládána explicitně (při přístupu k dalšímu prvku je vyžadován přístup do paměti indexů)

paměťová režie obecné formáty typicky ukládají sloupcový index pro každý *nnz* prvek

Kromě zmíněných problémů se zaměřuje i na iregularitu maticových vzorů, kde operace a přístupy do paměti nejsou kontinuální ani předvídatelné (nemají prostorovou lokalitu).

Principem je rozdělit vstupní matici A do $\lceil \frac{n}{s} \rceil$ vertikálních segmentů A^m, kde $m = 0, 1, \ldots, \lceil \frac{n}{s} \rceil - 1$, parametr s by měl být volen podle konkrétní architektury (ideálně velikost vektoru pro vektorizaci či velikost *warpu*). Každý segment A^m je uložen po řádcích, ukládány jsou pouze *nnz* prvky. Důvodem tohoto rozdělení je fakt, že při výpočtu $y = A\vec{x}$ jsou *nnz* elementy matice A použity pouze jednou, kdežto každému segmentu A^m připadá určitá část vektoru $\vec{x}$, ten lze tak znovu použít bez jeho zbytečného opětovného načítání do lokální paměti [4].

Každý segment A^m je v paměti uložen jako sekvence 6 vstupů, (viz. obrázek 4.3):

value hodnota *nnz* prvku

column index sloupcový index v rámci segmentu (tedy pro A^m je index j mod m)

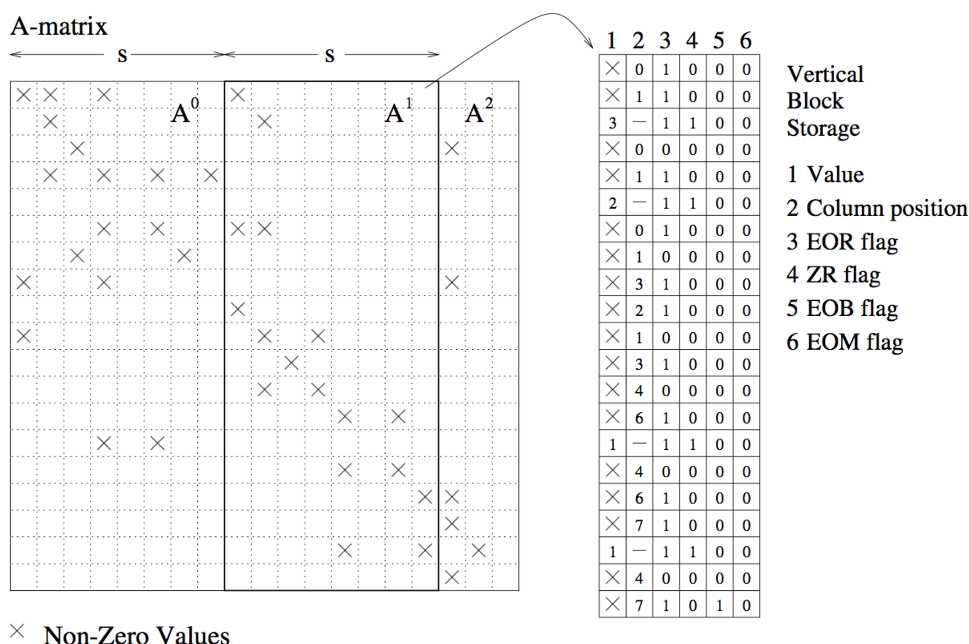
EOR flag neboli *end-of-row*, hodnota 1 je nastavena pouze pro poslední prvek v řádce

ZR flag neboli *zero-row*, hodnota 1 v případě, že řádek neobsahuje *nnz* prvky, v hodnotě *value* je pak počet následujících nulových řádků

EOB flag neboli *end-of-block*, hodnota 1 pro poslední prvek v segmentu

EOM flag neboli *end-of-matrix*, hodnota 1 pro poslední prvek v matici

Vstupní matice A je v paměti uložena po segmentech za sebou (nezáleží na pořadí). Celkově formát přináší menší paměťovou náročnost a možnost využití prostorové lokality dat. Vektorizace při SpMV výpočtu se využívá po sekvencích, kde každá výpočetní jednotka dostane šestici vstupů z matice a jeden prvek vstupního vektoru.



Obrázek 4.3: Použití formátu BBCS na řídkou matici (zdroj: [4])

4.5 Row Segmented Diagonal (RSDIAG)

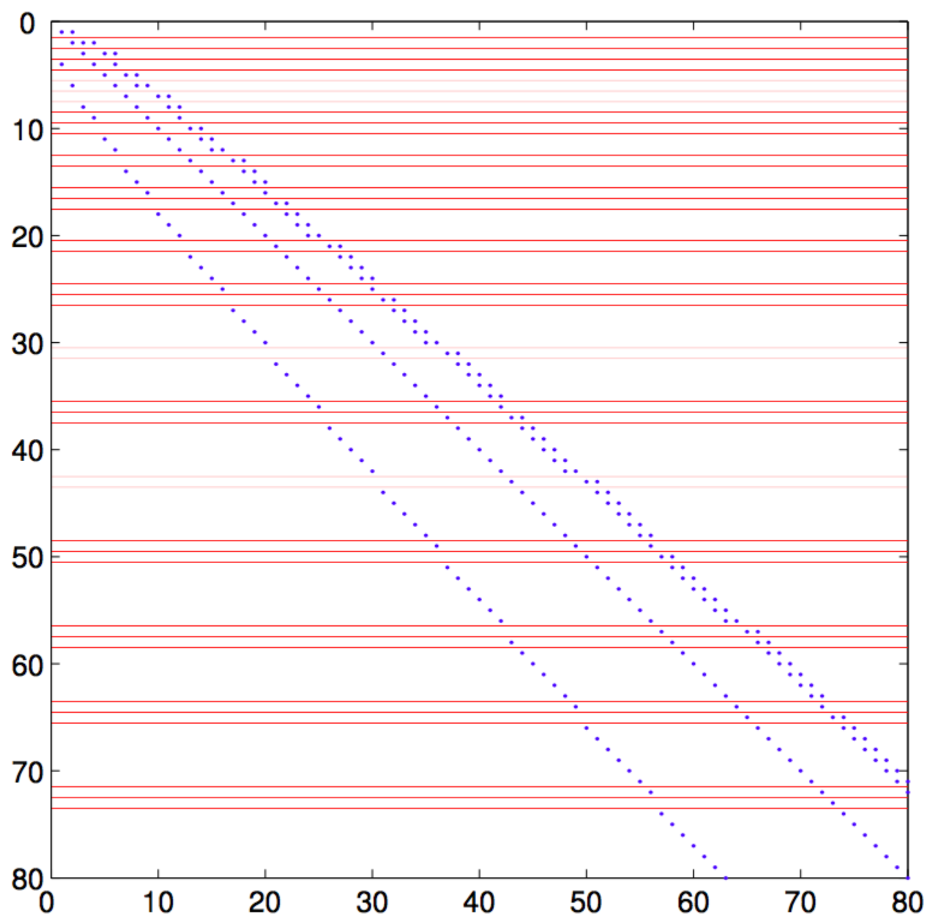
Formát určený pro vstupní matice s diagonálními substrukturami. Hlavní důvod jeho návrhu spočívá v tom, že původní DIAG (3.4) je uzpůsoben pouze na matice s úplnými nebo téměř úplnými diagonálami, problém nastává, pokud jsou diagonály pouze částečné a různě velké. Pro uložení takových diagonál by DIAG vyžadoval velké množství paměti pro uložení nulových prvků a výpočet by nebyl vůbec efektivní.

Principem RSDIAG je rozdělení vstupní matice do řádkových segmentů, kde každý segment obsahuje jednu nebo více diagonál o velikosti počtu řádků v segmentu.

Struktura formátu je následující:

- *val*[] – hodnoty *nnz* prvků
- *seq_starts*[] – obsahuje pointery na první řádky každého segmentu
- *src_ind*[] – sloupcové indexy ukazující na začátek každé diagonály
- *num_diags*[] – počet diagonál v každém segmentu

Princip rozdělení do segmentů je stejný jako u formátu UBCSR (viz. 4.3). Pokud se bitový vektor řádku shoduje s bitovým vektorem následujícího řádku, jedná se stále o jeden segment. Segmenty tak mohou mít různou velikost.



Obrázek 4.4: Princip rozdělení matice do řádkových segmentů v RSDIAG (zdroj: [8])

Nenulové prvky jsou do paměti ukládány prokládáním diagonál v každém segmentu pomocí parametru u (tzv. *unrolling depth*), tzn. že nejprve je uloženo prvních u nnz z první diagonály, poté prvních u nnz z druhé diagonály atd. dokud nejsou uloženy všechny nnz. Parametr u je vybírán na základě vstupní matice, případně podle architektury, na které je prováděn výpočet SpMV.

Rozdělení do segmentů je zobrazeno na následujícím příkladě, matice obsahuje dva segmenty (první má dva řádky, druhý řádky čtyři), parametr $u = 2$ [8]:

$$A = \begin{pmatrix} a_{00} & 0 & 0 & a_{03} & 0 & 0 & 0 \\ 0 & a_{11} & 0 & 0 & a_{14} & 0 & 0 \\ a_{20} & 0 & a_{22} & a_{23} & 0 & 0 & 0 \\ 0 & a_{31} & 0 & a_{33} & a_{34} & 0 & 0 \\ 0 & 0 & a_{42} & 0 & a_{44} & a_{45} & 0 \\ 0 & 0 & 0 & a_{53} & 0 & a_{55} & a_{56} \end{pmatrix}$$

$$val[] = \left[\; a_{00}, a_{11}, a_{03}, a_{14} | a_{20}, a_{31}, a_{22}, a_{33}, a_{23}, a_{34} | a_{42}, a_{53}, a_{44}, a_{55}, a_{45}, a_{56} \; \right],$$

$$seq_starts[] = \left[\; 0, 2, 6 \; \right],$$

$$src_ind[] = \left[\; 0, 3 | 0, 2, 3 \; \right],$$

$$num_diags[] = \left[\; 2, 3 \; \right].$$

Lze si všimnout, že formát neukládá žádné zbytečné nulové prvky. Díky lokalitě uložených dat (jedna diagonála potřebuje vždy část sousedících prvků vstupního vektoru $\vec{x}$) může dosahovat dobrých výsledků. Ideálně se nabízí využití SIMD jednotky o velikosti u.

4.6 Decomposed Block CSR (BCSR-DEC)

Blokový formát dekompozičního typu, který vychází z původního BCSR 4.1. Jeho cílem je vyhnout se zbytečného ukládání nulových prvků.

Dekompozice vstupní matice A se provádí do dvou submatic [42]:

- submatice s plnými bloky *nnz* prvků (žádné nulové výplně)
- submatice se zbývajícími prvky, uložení ve formátu CSR

Dekompozice přináší ovšem řadu problémů. Např. chybí časová a prostorová lokalita *nnz* elementů, dále pak druhá CSR submatice bude ve většině případů obsahovat velmi krátké řádky. Tyto faktory výrazně ovlivní celkový výkon SpMV a využití paralelních systémů bude neefektivní.

Bohužel pro tento formát autoři nezveřejňují bližší informace, a není tak zcela jasná jeho vnitřní struktura a její následné využití.

4.7 Decomposed Block Compressed Sparse Diagonals (BCSD-DEC)

Blokový formát dekompozičního typu, který vychází z formátu BCSD (blokové formy CDS 3.5).

Dekompozice vstupní matice A se provádí podobně jako u BCSR-DEC (4.6) do dvou submatic [42]:

- submatice s plnými bloky hustých diagonál s *nnz* prvky (žádné nulové výplně)
- submatice se zbývajícími prvky, uložení ve formátu CSR

Formát se potýká se stejnými problémy jako BCSR-DEC. I u tohoto formátu chybí v publikaci bližší popis celé struktury, a není tak možná detailnější analýza.

4.8 Blocked Transpose Jagged Diagnal Storage (BTJDS)

BTJDS vychází z formátu TJDS (3.7). Hlavním cílem tohoto nového formátu je zefektivnit vícevláknové zpracování GPGPU (využití rychlé sdílené paměti na GPU).

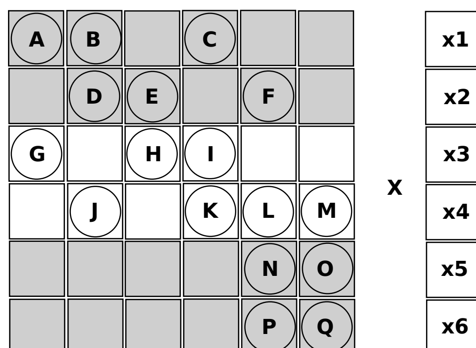
Princip spočívá v jednoduchém rozdělení vstupní matice do bloků. Každý blok obsahuje stejný počet řádků a je uložen ve formátu TJDS, tzn. že TJDS se aplikuje až po rozdělení matice do bloků. Do paměti jsou hodnoty ukládány po blocích za sebou. Struktura pro blokové TJDS zůstává stejná jako v případě klasického TJDS. Jediný rozdíl je, že do pole *row_ind*[] stačí ukládat pouze index řádku v rámci bloku, tzn. ušetření místa v případě velkých indexů. Výrazné změny doznal vstupní vektor $\vec{x}$, pro potřeby BTJDS obsahuje pro každý blok segment prvků, které jsou pro výpočet v konkrétním bloku potřeba. Vektor $\vec{x}$ může být tak mnohem větší než původní $\vec{x}$, navíc je přidáno pole *x_position*[], které obsahuje pointery na začátek segmentu x každého bloku. Princip uložení je vidět na obrázku 4.5.

Paralelizace se provádí způsobem jeden *warp* na určitý počet bloků (typicky 1 nebo 2). Každé vlákno se stará o přidělený počet sloupců a výsledky ukládá do sdílené paměti. Po dokončení svého výpočtů provede jedno vlákno redukci ve sdílené paměti a výsledek uloží do výstupního vektoru [33].

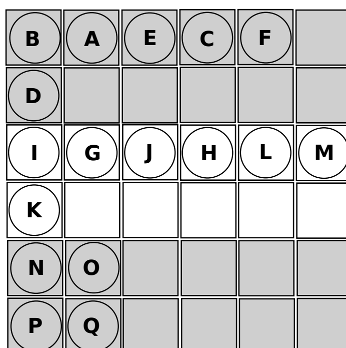
Souhrn výhod nového formátu:

- potřebné a vícekrát používané hodnoty jsou po dobu výpočtu uloženy v rychlé sdílené paměti
- přístup do globální paměti je souběžný

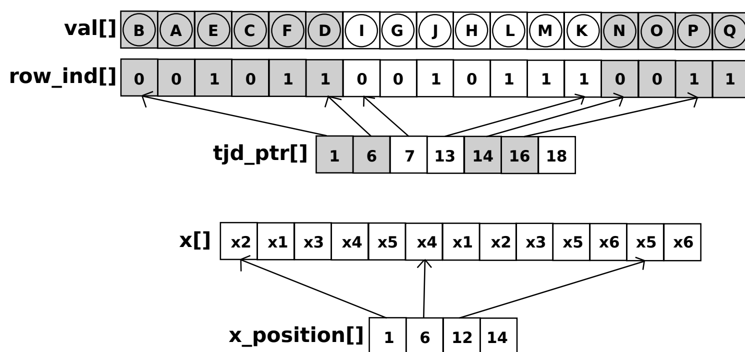
- *warp* má implicitní synchronizaci, synchronizace nutná pouze při globálním seskupování výsledků
- potřeba menšího množství bitů pro řádkové indexy



a) originální vstupní matice rozdělená do bloků a vektor



b) posunutí nenulových prvků směrem nahoru v rámci bloku a seřazení sloupců



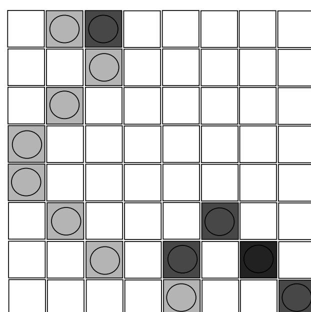
c) uložení prvků matice a přeuspořádaného vstupního vektoru

Obrázek 4.5: Uložení matice a vstupního vektoru ve formátu BTJDS

Kapitola **5**

CSR formáty určené pro GPU

Formáty v této kapitole jsou navrženy především z důvodu neefektivity klasického CSR (3.2) při GPGPU výpočtu SpMV. Jedním z problémů CSR na GPU je nejednotný přístup do hlavní paměti. Pro ukázku si lze představit řídkou matici, která má na každém řádku minimálně dva *nnz* prvky, při mapování jednoho vlákna na řádek pak jednotlivá vlákna v jedné iteraci přistupují do paměti na různě od sebe vzdálená místa, viz. obrázek 5.1, kde osm vláken v první iteraci přistupuje na prvky $0, 2, 3, 4, 5, 6, 8$ a 11 (indexováno od 0 v řádkovém uspořádání), které evidentně nejsou v paměti uloženy za sebou. Mezi další nevýhody patří neefektivní výpočet z důvodu např. nevyváženosti zátěže jednotlivých vláken, či velkého množství přístupů do globální paměti.



Obrázek 5.1: Příklad matice neposkytující jednotný přístup pro CSR při GPGPU

5.1 Row-grouperd CSR (RgCSR)

RgCSR se specializuje na lepší zarovnán hodnot v paměti pro efektivnější přístup. Principem je rozdělit vstupní matici do řádkových segmentů, kde každý segment obsahuje stejný počet řádků. Do paměti poté ukládáme prvky

po těchto segmentech za sebou. V rámci segmentu probíhá ukládání podle pořadí *nnz* prvku v řádku. Nejprve jsou uloženy všechny první *nnz* prvky každého řádku v segmentu, potom všechny druhé *nnz* každého řádku atd. Pokud nějaký řádek obsahuje méně *nnz* prvků než jiné řádky v segmentu, ukládají se nulové hodnoty [9].

Obrázek 5.2: Ukázka matice uložené ve formátu RgCSR (zdroj: [9])

Použitá pole:

- *val*[] – hodnoty prvků (včetně nulové výplně)
- *col_ind*[] – sloupcové indexy prvků z *val*[]
- *group_ptr*[] – pointer do *val*[]/*col_ind*[] na první prvek v každém segmentu
- *row_lenght*[] – pole s počty *nnz* prvků pro každý řádek

Při mapování *warpu* na segment pak přistupují vlákna v iterací k sousedícím hodnotám a přístup do paměti je tak efektivnější. Formát se hodně podobá formátu SELLPACK (představen v 6.2) a jemu podobným.

5.2 Adaptive row-grouped CSR (ArgCSR)

ArgCSR navazuje na formát RgCSR (5.1) a formát ELLPACK (3.9). Zaměřuje se hlavně na eliminaci neefektivního výpočtu v případech, že matice obsahuje jeden či několik silně hustých řádků. Při výpočtu je většina vláken v rámci *warpu* ve stavu, že musí provádět prázdné operace a čekat na dokončení vlákna, které operuje nad hustou řádkou. To vede ke zbytečnému plýtvání výpočetních zdrojů a celkové neefektivitě.

Struktura formátu je jednoduchá, obsahuje následující pole:

- *val*[] pole hodnot prvků (včetně nulové výplně)
- *col_ind*[] pole sloupcových indexů prvků z *val*[]

- *threads_mapping*[] – způsob mapování vláken na jednotlivé řádky

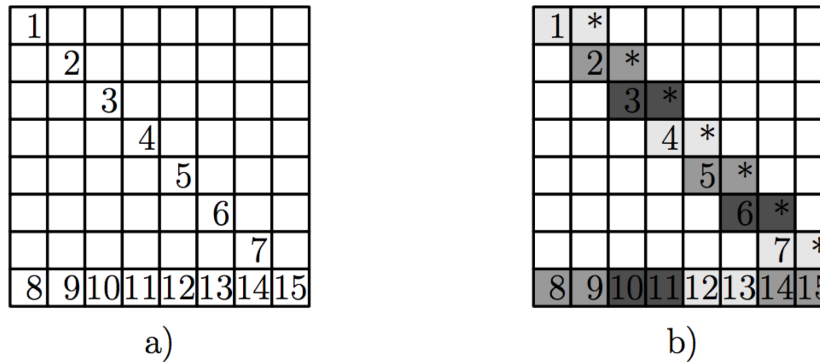
Nový formát rozděluje matici do skupin řádků, každá taková skupina se definuje pomocí následujících údajů [10]:

firstRow index na první řádek dané skupiny

size počet řádků ve skupině

offset první prvek skupiny v poli *val*[] popř. *col_ind*[]

chunkSize počet zpracovávaných prvků jedním vláknem



values[]	1	2	3	4	5	6	7	8	10	12	14	*	*	*	*	*	*	*	9	11	13	15
columns[]	0	1	2	3	4	5	6	0	2	4	6	-1	-1	-1	-1	-1	-1	-1	1	3	5	7

threadsMapping[]	0	1	2	3	4	5	6	7	12

Obrázek 5.3: Ukázka matice uložené ve formátu ArgCSR (zdroj: [10])

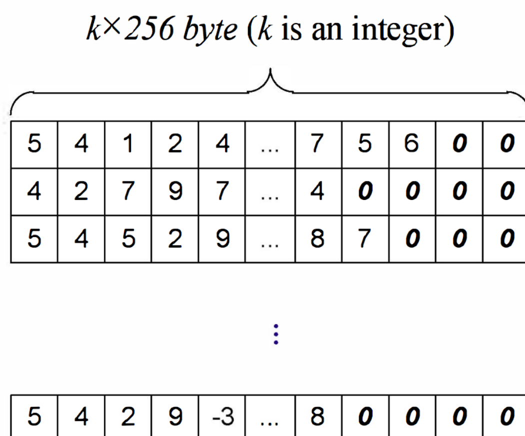
Na obrázku 5.3 je uveden příklad ukládání skupiny řádků a mapování 12 vláken na tzv. „*chunky*“, kde $chunkSize = 2$. Místo 49 zbytečných instrukcí se nyní provede pouze 7. Hodnoty jsou do paměti ukládány podobně jako RgCSR, tentokrát však nikoliv podle pozice v řádku, nýbrž podle pozice v *chunku*. To zajišťuje jednotný přístup na sousedící prvky v globální paměti v rámci jedné iterace. Mapování více vláken na husté řádky pak výrazně snižuje výpočetní režii a poskytuje efektivnější využití výpočetních zdrojů.

5.3 Improved Compressed Row Storage (ICSR)

Klasické CSR je na GPU neoptimální a jedním z důvodů je fakt, že je nutné řešit hodně nepřímých a nepravidelných přístupů do globální paměti pro jednu aritmetickou operaci, proto autoři formátu ICSR přichází s návrhem ideálního

zarovnání dat v globální paměti, které by mohlo vést k vyššímu výkonu SpMV [11].

Princip spočívá v zarovnání jednotlivých řádků na násobek 256, každý řádek tak startuje na pozici, jejíž index je násobkem 256. Toho je docíleno doplněním nulových prvků, viz. obrázek 5.4. Přesný důvod doplnění na násobek 256 autor neuvádí, hlavní roli by měla hrát architektura, na které bude matice uložena, formát lze tak jednoduše přizpůsobit změnou zarovnání na jinou hodnotu.



Obrázek 5.4: Zarovnání řádků v globální paměti GPU pro formát ICSR (zdroj: [11])

Doplnění nulových prvků mírně zvýší náročnost na úložný prostor (je nutné rozšířit i pole sloupcových indexů), ale výkonnost SpMV může růst [61]. Formát má hodně blízko ke klasickému ELLPACK formátu (3.9). Má tedy i stejné problémy, např. u matice s jednou velmi hustou řádkou rostou nároky na úložný prostor do extrémních hodnot a formát je v podstatě nepoužitelný.

5.4 Compressed Multi-Row Storage (CMRS)

Podstatou tohoto formátu je zpracovat řídkou matici ve větších kusech než po jednotlivých řádkách a zároveň zachovat typickou strukturu CSR formátu.

Vstupní matice je rozdělena do skupin řádků nazývaných *stripy*, počet řádků ve *stripu* je určen parametrem *height*. Pokud platí $height = 1$, jedná se o klasický CSR.

Struktura formátu obsahuje [34]:

- *val*[] pole hodnot *nnz* prvků

- *col_ind*[] pole sloupcových indexů *nnz* prvků
- *strip_ptr*[] pole pointerů na první *nnz* daného *stripu*
- *row_in_strip*[] pole indexů řádků každého *nnz* v rámci *stripu*

Příklad konkrétní reprezentace formátu je vidět na následující vstupní matici:

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 & 4 \\ 0 & 0 & 5 & 0 & 6 \\ 0 & 0 & 7 & 8 & 9 \\ 0 & 0 & 0 & 0 & 10 \end{pmatrix}$$

$$val[] = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{bmatrix},$$

$$col_ind[] = \begin{bmatrix} 0 & 1 & 3 & 4 & 2 & 2 & 3 & 4 & 4 & 4 \end{bmatrix},$$

$$row_in_strip[] = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix},$$

$$strip_ptr[] = \begin{bmatrix} 0 & 4 & 9 & 10 \end{bmatrix}.$$

Pro souvislý přístup do paměti jsou hodnoty do paměti ukládány po *stripech*, a to v pořadí podle sloupcového indexu[1]. Pole *row_in_strip*[] umožňuje dynamické přidělování vláken na řádek. Na rozdíl od dalších podobných formátů nevyžaduje ukládání zbytečných nulových prvků ani řazení řádků podle velikosti.

5.5 CSR with Segmented Interleave Combination (SIC)

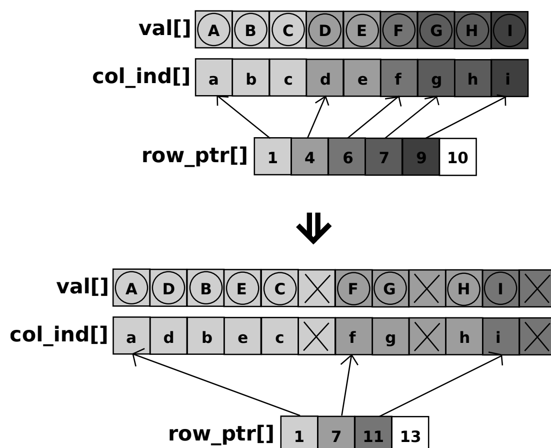
Formát SIC využívá prokládání řádků se snahou o snížení paměťových požadavků a zabránění zbytečného plýtvání výpočetní zdrojů na GPU.

Hlavním principem je kombinování určitého množství CSR řádek do nového SIC řádku. Vstupní matice A je rozdělena do $\lceil \frac{n}{c} \rceil$ skupin, kde c je počet

[1] Podle analýzy to nemusí být nejefektivnější řešení, např. v případě, kdy jeden řádek má všechny *nnz* prvky umístěny v první půlce matice a druhý řádek naopak v druhé půlce.

sousedících řádků v jedné skupině. Struktura k uložení formátu zůstává stejná jako u CSR [41].

Princip prokládání je zobrazen na obrázku 5.5. Pokud řádky v rámci skupiny nemají stejný počet *nnz* prvků, jsou nové řádky doplněny nulovými prvky. V případě, že n není násobkem c, jsou na konec matice doplněny celé nulové řádky.



Obrázek 5.5: Princip prokládání řádků použitý ve formátu SIC

Formát je závislý na rozmístění *nnz* prvků ve vstupní matici, pokud by matice obsahovala několik hustých řádků, znamenalo by to zbytečné ukládání nulových prvků. Mapováním jednoho *warpu* na jeden SIC řádek lze redukovat pomalé přístupy do globální paměti, nutná je v tomto případě ale implementace redukce mezivýsledků (díky rychlosti sdílené paměti ale nemusí být z hlediska režie problémová).

Kapitola 6

ELLPACK formáty

ELLPACK formáty mají základ v původním ELLPACK (viz. 3.9). Jeho modifikace si kladou za cíl redukovat nevýhody původního formátu a poskytnout efektivní zpracování při GPGPU výpočtu SpMV.

6.1 Hybrid ELLPACK/COO (HYB)

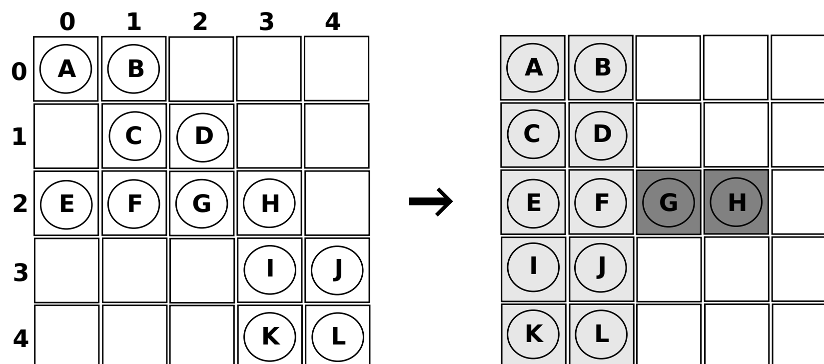
Formát založen na kombinaci ELLPACK formátu a COO (3.1). Důvodem jeho vzniku je jedna z největších nevýhod klasického ELLPACK, a to pokud nějaký řádek má znatelně větší počet *nnz* prvků než je průměr, je třeba zbytečně ukládat velké množství nulových prvků. HYB v těchto případech poskytuje menší prostorovou náročnost.

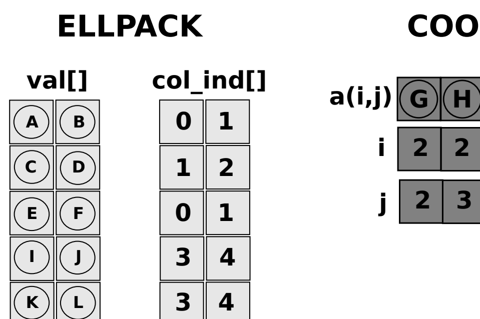
Struktura formátu je stejná jako struktura ELLPACK, ve které se ukládá $n \times k$ prvků, přibývá navíc ještě struktura formátu COO. Principem HYB je určení efektivní meze k_1, která určuje počet sloupců, které se budou ukládat v ELLPACK formátu. Prvky, které se do této meze nevejdou, jsou uloženy ve formátu COO. K uložení je tedy potřeba jen $n \times k_1 + 3c_{coo}$, kde $k_1 < k$ a c_{coo} je počet prvků ukládaných v COO [47].

Pro dosažení efektivního uložení je nutné zvolit optimální parametr k_1, protože můžou nastat dva extrémní případy [9]:

- k_1 je jen o málo menší než k – stále bude tedy potřeba ukládat velké množství nulových prvků
- k_1 je moc malé – nutnost ukládat většinu prvků v méně efektivním COO

Spojení dvou formátu s sebou nese určité nevýhody. U HYB je nutná implementace výpočtu pro oba typy formátů. Je zřejmé, že paralelní zpracování bude pro část s COO prvky neefektivní.





Obrázek 6.1: Uložení řídké matice ve formátu HYB

6.2 Sliced ELLPACK (SELLPACK)

Další z formátů, který se snaží redukovat režii a ukládání velkého množství nulových prvků ELLPACK formátu v případě nevyváženého počtu *nnz* prvků v řádcích vstupní matice.

Princip spočívá v rozdělení matice podle parametru S do pásu, tzv. „*stripu*“, který pokrývá S sousedících řádků. Každý *strip* je seřazen v ELLPACK formátu. V případě, že počet řádků matice není dělitelný parametrem S, doplňují se do matice nulové řádky. Kromě klasické struktury ELLPACKu obsahuje nový formát navíc pole, do kterého jsou ukládány indexy prvních prvků v jednotlivých *stripech*. Struktura je naznačena na obrázku 6.2.

Lze si všimnout, že struktura se hodně podobá CSR formátu. V případě $S = 1$ dostáváme klasický CSR (viz. 3.2). Pokud $S = n$, pak se jedná o klasický ELLPACK (viz. 3.9). S přibývající hodnotou S roste počet ukládaných nulových prvků. Pro optimální využití formátu volíme S nízké s ohledem na ušetřený paměťový prostor, zároveň ale dostatečně velké, aby bylo možné efektivní využití GPU architektury (ideální hodnota S je alespoň velikosti jednoho *warpu*) [12].

$$\begin{pmatrix} a & 0 & b & 0 \\ 0 & c & 0 & d \\ 0 & e & 0 & 0 \\ 0 & 0 & 0 & f \end{pmatrix}$$

value: $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ $\begin{pmatrix} e \\ f \end{pmatrix}$ column: $\begin{pmatrix} 0 & 2 \\ 1 & 3 \end{pmatrix}$ $\begin{pmatrix} 1 \\ 3 \end{pmatrix}$

slice_start	0				4	6
column	0	1	2	3	1	3
value	a	c	b	d	e	f

Obrázek 6.2: Uložení matice do pásů Sliced ELLPACKu (zdroj: [12])

6.3 Blocked ELLPACK (BELLPACK)

Formát se snaží vylepšit blokový formát BCSR (viz. 4.1), který sice zefektivňuje klasický CSR, nicméně se svoji strukturou nehodí pro GPU architekturu a efektivita výpočetu SpMV zde zaostává.

Postup konstrukce BELLPACK má následující fáze [6]:

blocking rozdělí matici do bloků $r \times c$

reordering přeuspořádá bloky, všechny nenulové bloky jsou posunuty doleva a následně seřazeny odshora podle počtu nenulových bloků (podobně jako u JDS, viz. 3.6)

ELLPACK seskupí sousedící bloky do stejně velkých submatic velikosti R bloků (počet submatic bude $\frac{n}{R}$)

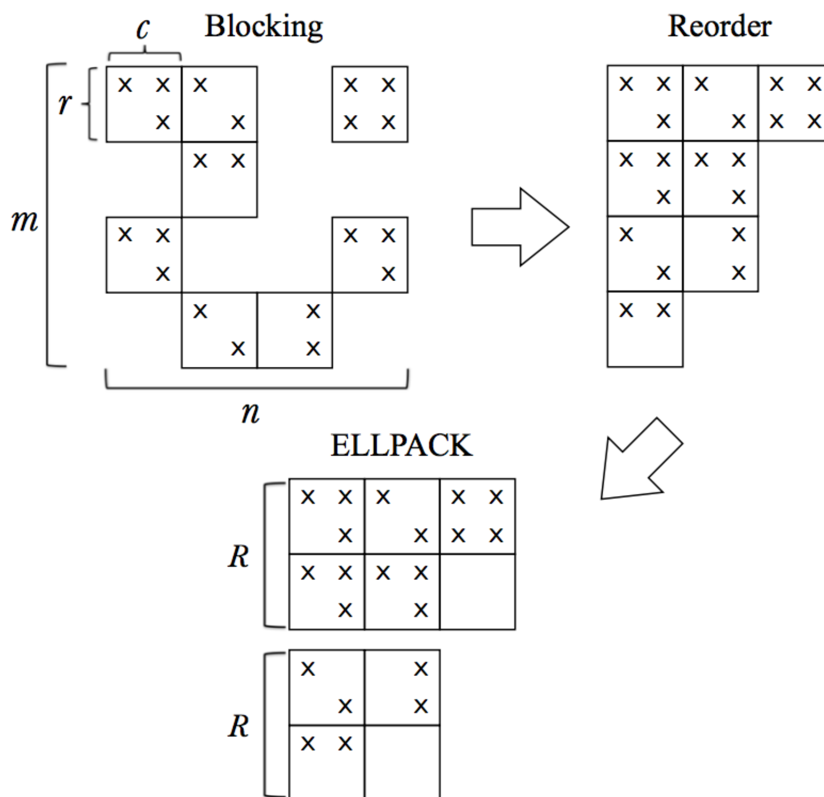
Obrázek 6.3 zobrazuje tvorbu BELLPACK formátu. Jednotlivé submatice jsou ukládány ve formátu ELLPACK. Princip ukládání je zobrazen na obrázku 6.4. Prvky jsou uspořádány do sloupců po řádcích blok po bloku, kde každý sloupec je určen pro jedno vlákno. Do paměti jsou tyto sloupce ukládány po řádcích, což zajistí sjednocený přístup vláken k souvislému paměťovému bloku. Pro určení potřebných prvků vstupního vektoru $\vec{x}$ je ještě třeba ukládat sloupcové pozice jednotlivých $r \times c$ bloků.

Pro nejvyšší výkon je vyžadována optimální volba parametrů r, c a R. Tyto parametry jsou závislé na vstupní matici.

6.4 ELLPACK-R (ELL-R)

ELL-R rozšiřuje původní ELLPACK o informaci, která udává počet *nnz* prvků v jednotlivých řádcích. Do struktury ELLPACKu je tedy přidáno pole $rl[]$ o velikosti n [45].

Informace o počtu *nnz* prvků omezí výpočet pouze na operace s *nnz*, navíc lze ušetřit režii vynecháním podmíněných větví (netřeba při každém přístupu



Obrázek 6.3: Konstrukce BELLPACK (zdroj: [6])

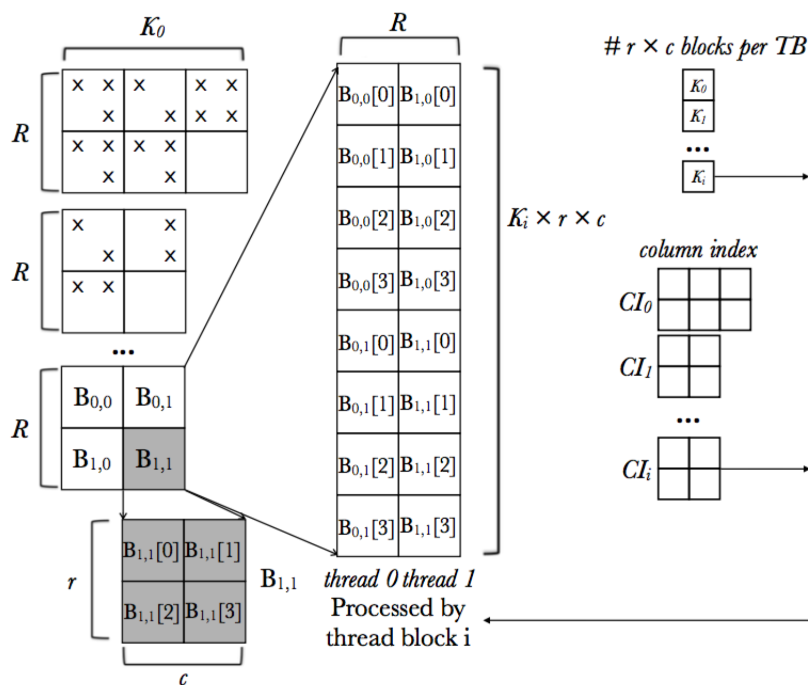
testovat, zda je prvek nulový). Každé vlákno ve *warpu*, starající se o jeden řádek, iteruje pouze přes daný počet prvků, poté se zastaví, ostatní vlákna ve *warpu* mohou pokračovat dál. Běh *warpu* tak závisí na nejdelším řádku [43].

6.5 ELLR-T

Rozšíření formátu ELL-R, cílem je efektivnější mapování na výpočetní bloky GPU.

Hlavní rozdíl je v systému výpočtu, v ELLR-T pracuje více vláken T na jednom řádku ($T = 1, 2, 4, 8, 16, 32$). Prvky jsou v paměti ukládány po řádkách v rámci T sloupců (viz. obrázek 6.5). Celá sekvence je doplněna nulovými prvky tak, aby celková velikost byla násobkem 16 [13].

Vlákna jsou rozdělena do skupin S_i, kde S_i obsahuje set T vláken řešících řádek i. Každé vlákno v S_i obstarává přesně $\lceil \frac{rl[i]}{T} \rceil$ prvků v i-tém řádku, své mezivýsledky si každé vlákno ukládá do sdílené paměti. Mapování více vláken



Obrázek 6.4: Uspořádání prvků v BELLPACK formátu (zdroj: [6])

na řádek s sebou přináší nutnost implementace paralelní redukce, která poté z mezivýsledků vytvoří konečný výsledek prvku i výstupního vektoru [45].

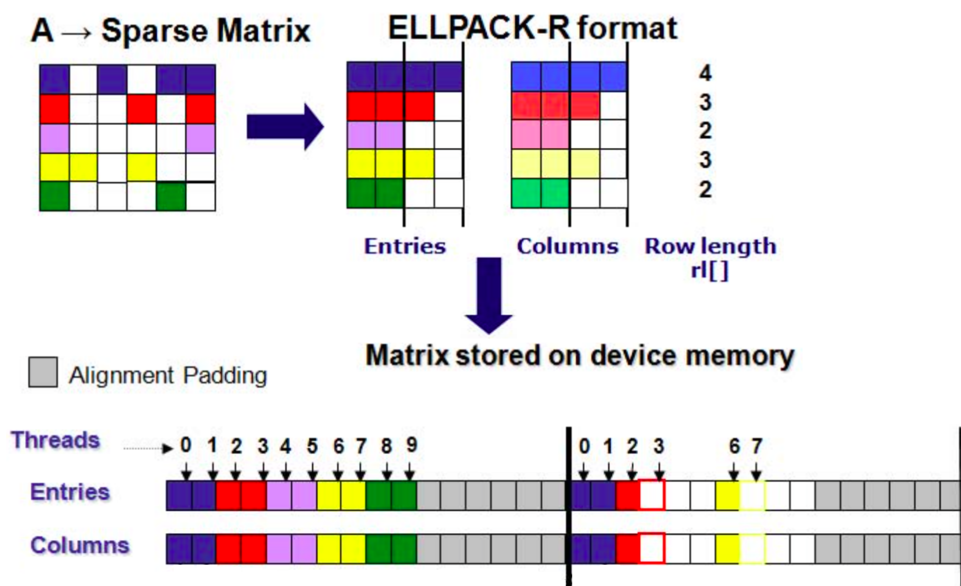
Formát poskytuje několik výhod:

- způsob uložení zajišťuje jednotný a seřazený přístup do globální paměti
- stejný kód pro všechna vlákna v rámci *warpu*
- neprovádí zbytečné iterace a vyhýbá se nevyváženosti výpočtu v rámci *warpu*

6.6 Sliced ELLR-T

Jednoduchá kombinace formátu Sliced ELLPACK (6.2) a ELLR-T (6.5). Vstupní matice je, stejně jako u Sliced ELLPACKu, rozdělena do řádkových segmentů obsahujících S sousedících řádek. Každý segment je poté uložen ve formátu ELLR-T [52].

Rozdělením vstupní matice do segmentů dochází k redukci paměťových požadavků. Využitím ELLR-T a sdílené paměti GPU je možné dosáhnout efektivního výkonu SpMV.



Obrázek 6.5: Uložení řídké matice pomocí ELLR-T pro $T = 2$ (zdroj: [13])

6.7 ELLPACK-RP

Formát vznikl s cílem lepšího vyvažování zátěže u formátu ELLPACK-R (6.4). Využívá přitom poznatky z formátu JDS (3.6).

ELLPACK-R je vhodný pro GPU architektury, pokud ale nastane případ, že nějaký řádek obsahuje velké množství *nnz* prvků než je průměrný počet, výpočet se stává silně nevyvážený a neefektivní (jedno vlákno iteruje přes hustý řádek, ostatní vlákna mají delší dobu dopočítáno a jsou v nečinném stavu). Formát JDS, podobně jako ELLPACK, zhušťuje *nnz* směrem doleva, navíc provádí permutaci řádků podle počtu jejich nenulových prvků.

Nechť NNZ_i určuje počet *nnz* prvků v řádku i, lze předpokládat následující [44]:

$$NNZ_1 \geq NNZ_2 \geq \ldots \geq NNZ_k \geq NNZ_{k-1} \geq \ldots \geq NNZ_{n-1} \geq NNZ_n$$

Zjednodušeně lze tedy předpokládat:

$$NNZ_1 + NNZ_n \approx NNZ_2 + NNZ_{n-1} \approx \ldots \approx NNZ_k + NNZ_{k-1}$$

Pro vyvážený výpočet by tak měly být jednomu vláknu přiřazeny dva řádky z opačných konců.

Struktura formátu je totožná se strukturou formátu ELLPACK-R, přibývá navíc pouze pole $perm[]$ o velikosti n, které pro každý řádek ukládá jeho původní pozici před permutací.

Příklad konkrétní reprezentace formátu je vidět na následující vstupní matici:

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 3 \\ 7 & 0 & 9 & 0 \\ 4 & 2 & 1 & 0 \end{pmatrix}$$

$$val[] = \begin{bmatrix} 4 & 2 & 1 \\ 2 & 3 & * \\ 7 & 9 & * \\ 1 & * & * \end{bmatrix} col_ind[] = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 3 & * \\ 0 & 2 & * \\ 3 & * & * \end{bmatrix}$$

$$rl[] = \begin{bmatrix} 3 \\ 2 \\ 2 \\ 1 \end{bmatrix} perm[] = \begin{bmatrix} 3 \\ 1 \\ 2 \\ 0 \end{bmatrix}$$

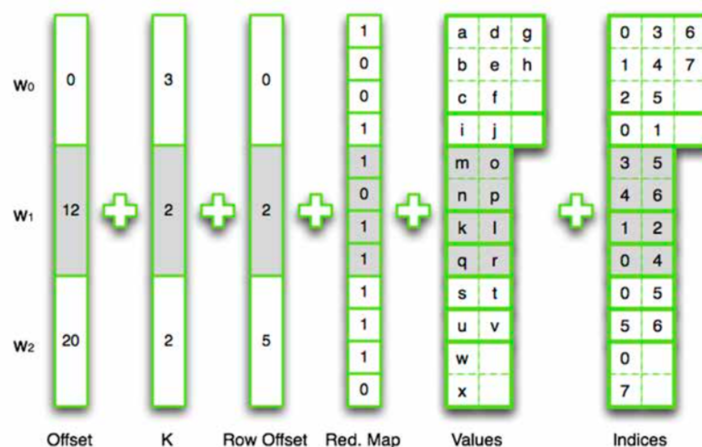
6.8 Adaptive Warp-Balancing ELLPACK (AdELL)

Formát navržený s myšlenkou vylepšení vyvažování zátěže v rámci jednotlivých *warpů*. Z důvodu nepravidelné distribuce *nnz* prvků v matici dochází k neefektivnímu využití možné paralelizace.

Formát funguje na podobném principu jako např. SIC (5.5), CMRS (5.4) nebo RgCSR (5.1), zmíněné formáty ovšem neberou v potaz nevyváženost pracovních úloh a přiřazují rovnoměrné počty vláken každému řádku, navíc jejich výkon závisí na parametrizaci. Dalším podobným formátem je ArgCSR (5.2), který již využívá jednoduchého vyvažování, je ale podmíněn blokovou synchronizací. V případě AdELL si vystačíme se synchronizací na úrovni *warpu*, která je více efektivnější.

Každý *warp* může zpracovávat od 1 do T řádek, kde T je počet vláken ve *warpu*. V každém řádku může použít různý počet vláken, díky této distribuci lze efektivně využít paralelní výpočet na dané architektuře. Struktura formátu byla částečně inspirována formátem SELLPACK (6.2). Příklad uložení konkrétní matice je vidět na obrázku 6.6 (jako vstupní matice je zde použita matice A z obrázku 3.9).

K uložení jsou potřeba následující struktury [5]:



Obrázek 6.6: Příklad reprezentace AdELL formátu (zdroj: [5])

- *val*[] pole hodnot *nnz* prvků
- *col_ind*[] pole sloupcových indexů *nnz* prvků
- $offset$[] pole o velikosti počtu *warpů*, každý *offset* určuje začátek dat v paměti pro každý *warp*
- row_offset[] pole o velikosti počtu *warpů*, obsažené pointery určují vždy první řádek pro každý *warp*
- *workload*[](K) pole s lokální zátěží každého *warpu* (počet *nnz* v řádku i dělený počtem přiřazených vláken na řádek i)
- *reduction_map*[] – bitová mapa, která určuje vlákno, které bude zapisovat konečný výsledek pro konkrétní řádku (bit nastaven na 1)

V případě $workload > 1$ musí vlákna spolupracovat na výpočtu jednoho řádku, nicméně není třeba žádné speciální synchronizace, každé vlákno si bude počítat svoji část, po dokončení celého *warpu* pak jedno z vláken (podle pole *reduction_map*[]) zpracuje ve sdílené paměti mezivýsledky. Formát je schopen vyrovnat se i s maticemi, které obsahují velmi husté řádky, které vybočují z průměru vstupní matice.

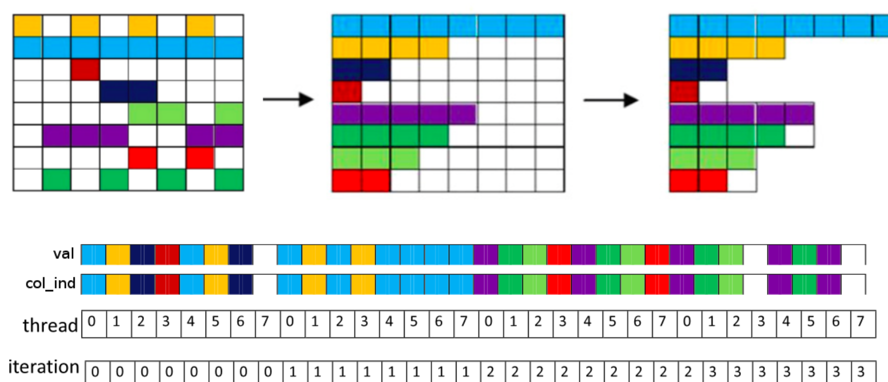
6.9 Bisection ELLPACK (BiELL)

Další z ELLPACK formátů, který se pokouší o vylepšení vyvažování zátěže při výpočtu SpMV. Hlavní myšlenka spočívá ve tom, že po výpočtu první poloviny řádků jsou vlákna přiřazena druhé polovině.

Principem je rozdělit vstupní matici do řádkových bloků (*stripů*), typicky o velikosti *warpu*. V rámci *stripu* jsou řádky seřazeny podle počtu *nnz* prvků, *nnz* jsou zarovnány doleva. Sloupce v jednotlivých *stripech* jsou poté rozděleny do $\log_2(warpsize) + 1$ skupin, prvky v každé skupině jsou v paměti uloženy za sebou pomocí ELLPACK formátu [14].

Pro formát je potřeba následující struktura:

- *val*[] – hodnoty *nnz* prvků
- *col_ind*[] – sloupcové indexy *nnz* prvků
- *perm*[] – uchování pozic řádků z důvodu permutace
- *ptr_group*[] – pointery na první prvek každé skupiny



Obrázek 6.7: Princip BiELL pro 8 vláken (zdroj: [14])

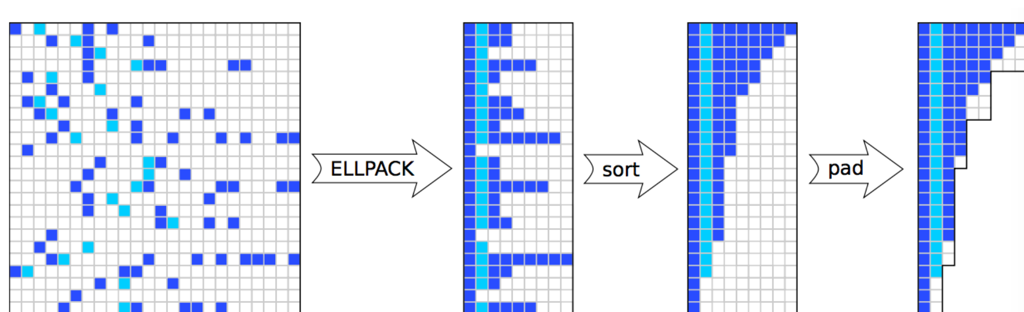
Výpočet probíhá postupně po skupinách.

6.10 Padded Jagged Diagonal Storage (pJDS)

Formát navržený speciálně pro výpočty na GPU. Jeho hlavním cílem je vylepšení formátu ELLPACK-R (6.4).

Princip nového formátu pochází ze základních formátů JDS 3.6 a ELLPACK 3.9. Nenulové prvky jsou seskupeny do levé části, poté jsou řádky seřazeny podle svojí hustoty, následně jsou řádky rozděleny do bloků velikosti b_r. Hodnota b_r by měla být ideálně velikost *warpu*. Postup převodu na pJDS je vidět na obrázku 6.8.

Struktura formátu je stejná jako v případě ELLPACK, tedy pole *val*[] a *col_ind*[] pro uložení hodnot a jejich sloupcové pozice. Hodnoty jsou v paměti



Obrázek 6.8: Převod matice do formátu pJDS (zdroj: [15])

ukládány po sloupcích, proto nastává situace, že jednotlivé sloupce jsou jinak dlouhé, z tohoto důvodu je přidáno ještě jedno pole, a to *col_start*[], které udává odkaz na první prvek v paměti pro daný sloupec [15].

Formát v podstatě udržuje strukturu formátu ELL-R, ale poskytuje potencionálně lepší využití paměti a hardwarové architektury (vlákna v rámci *warpu* budou přibližně rovnoměrně vytíženy). Permutace řádků mohou ale rozbít určité pravidelné struktury matice (diagonály, husté bloky apod.), což vede k neefektivnímu využití vstupního vektoru $\vec{x}$ (z pohledu uspořádání dat či znovu využití v cache). Výstupní vektor musí být navíc opět permutován do původní podoby, je tedy nutné držet informaci o pořadí řádků před permutací.

6.11 SELL-C-σ

Formát vychází ze Sliced ELLPACK (6.2), jeho modifikace se snaží snížit režii výpočtů nad nulovými prvky a zvýšit prostorovou lokalitu dat.

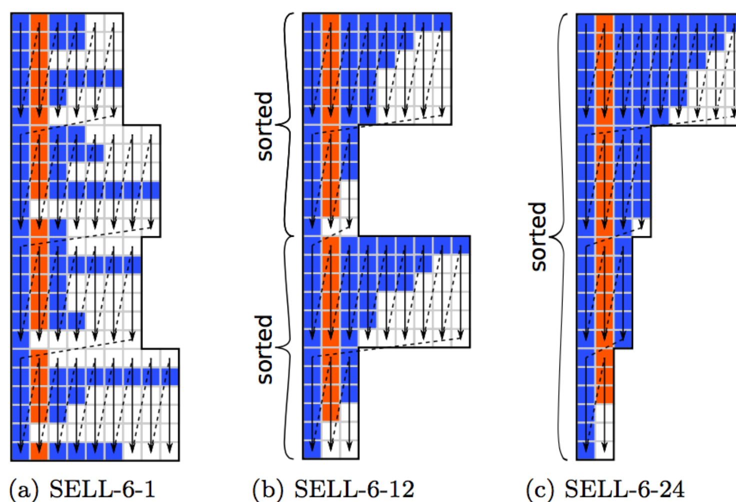
Formát je parametrizován pomocí následujících parametrů [16]:

- C – velikost bloku po sobě jdoucích řádek (tzv. „*chunk*“)
- σ – velikost třídícího segmentu, typicky násobek C

Struktura formátu je oproti struktuře Sliced ELLPACKu rozšířena o pole *cl*[], které uchovává počet *nnz* prvků v nejhustším řádku v rámci *chunku*. Ukázka formátu je zobrazena na obrázku 6.9.

Třídící parametr σ udává počet řádků v rámci *chunků*, které budou řazeny podle své hustoty, třídí se tak typicky přes více *chunků*. Ideální σ je známe pouze pro silně pravidelné matice, jeho volba může tedy výrazně ovlivnit efektivitu výpočtu. Z pohledu redukce nulových prvků je např. nejideálnější $\sigma = n$, to může ovšem úplně rozbít prostorovou lokalitu dat.

V vztahu s ostatními formáty lze (podobně jako u Sliced ELLPACK) tvrdit následující:



(a) SELL-6-1 (b) SELL-6-12 (c) SELL-6-24

Obrázek 6.9: Použití SELL-C-σ formátu pro různé σ (zdroj: [16])

- SELL-1-1 principiálně koresponduje s klasickým CSR (3.2)
- SELL-n-1 principiálně koresponduje s klasickým ELLPACK (3.9)
- SELL-C-n principiálně koresponduje s pJDS (6.10)

6.12 ELLPACK Sparse Block (ESB)

Formát navržený speciálně pro efektivní výpočet na koprocesoru Intel Xeon Phi (viz. 2.4).

Implementace se snaží řešit typické problémy SIMD architektury při práci s řídkými maticemi, mezi ty hlavní patří:

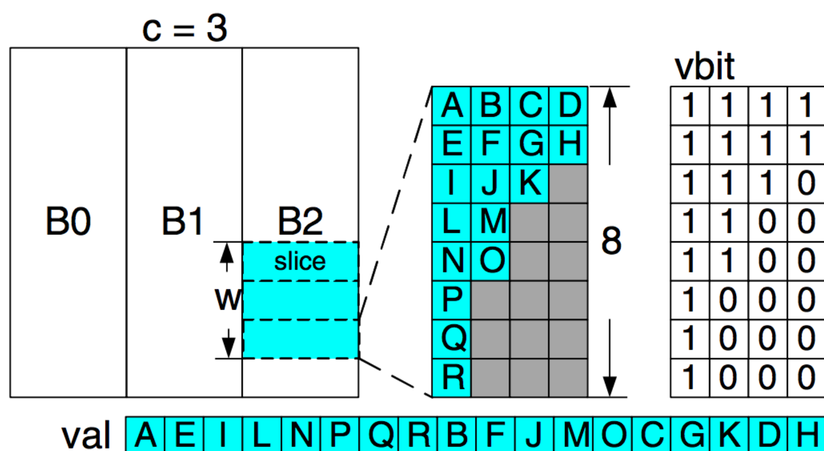
- malá SIMD efektivita kvůli řídkosti matice
- velká režie z důvodu nepravidelného přístupu do paměti
- nevyváženost výpočtu

Nově navržený formát upravuje klasický ELLPACK o několik specifických rozšíření [3]:

- konečný třídící segment w (podobně jako SELL-C-σ, viz. 6.11)
- bitové pole, které určuje pozici nnz prvků
- rozdělení matice do c sloupcových bloků

- vyvažování zátěže (implementováno na aplikační úrovni)

Vstupní matice je rozdělena na sloupcové bloky, v rámci každého sloupcového bloku jsou řádky rozděleny do řádkových bloků (tzv. „*slices*“), kde každý *slice* obsahuje 8 řádek. Přesně po w řádkových bloků jsou bloky tříděny podle hustoty jednotlivých řádků. Celá sekvence w *slices* je ukládána ve formátu SELLPACK (6.2). K SELLPACKu je dále přidána bitová mapa, která určuje pozici *nnz* prvků v rámci ukládaného bloku, bitová reprezentace je na zvolené architektuře výrazně efektivnější. Díky řazení pouze w bloků je udržována prostorová lokalita dat vůči vstupnímu vektoru $\vec{x}$. Princip uložení struktury je vidět na obrázku 6.10.



Obrázek 6.10: Struktura ESB (zdroj: [3])

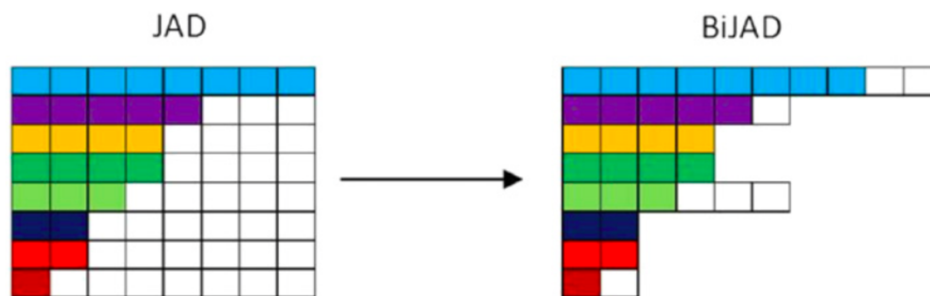
Potřebné pole pro uložení celé struktury jsou:

- *val*[] – hodnoty *nnz* prvků
- *col_ind*[] – sloupcové indexy *nnz* prvků
- *vbit*[] pole bitů určující pozici každého *nnz* v SELLPACKu
- *yorder*[] – pořadí řádků po provedení řazení v rámci w
- *slice_ptr*[] – pointery ukazující na první prvek každého *slice*
- *vbit_ptr*[] pole c pointrů do *vbit*[] ukazující na začátek bitového pole pro každý sloupcový blok

Z popisu vyplývá, že (za předpokladu dobře zvolených parametrů c a w) dokáže formát efektivně využít dostupnou cache s dobrou podporou lokality dat, což může SpMV výrazně urychlit.

6.13 Bisection JAD format (BiJAD)

BiJAD nebo též BiJDS. Jedná se o modifikaci formátu BiELL (6.9). Jeho nevýhodou, v případě nerovnoměrných hustot řádků, může být větší množství doplněných nulových prvků v rámci skupin [14]. Formát je v podstatě totožný jako BiELL, liší se pouze v jednom kroku, a to že v prvním kroku řadí všechny řádky podle počtu *nnz* prvků od nejhustších po nejřidší (podobně jako u formátu JDS/JAD), poté se s řádky zachází stejně jako u formátu BiELL, struktura pro uložení matice je tedy stejná. Princip je zobrazen na obrázku 6.13.



Obrázek 6.11: Převod seřazených řádků pro použití BiJAD formátu (zdroj: [14])

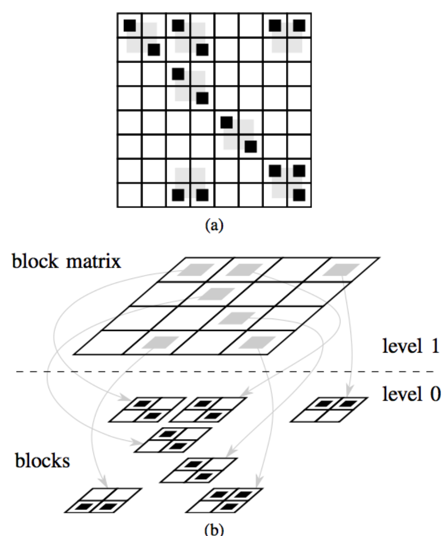
Kapitola 7

Hierarchické formáty

Formáty hierarchického typu, jejich principem je rozdělit vstupní matici A do menších disjunktních bloků. Indexy *nnz* elementů jsou rozděleny na dvě části:

- indexy bloků (level 0)
- indexy v rámci bloku (level 1)

Hlavní myšlenka tohoto typu formátů je, že sice nelze zredukovat počty elementů v indexových polích, ale můžeme zredukovat počty bitů pro každý řádkový/sloupcový index.



Obrázek 7.1: Princip hierarchických formátů (zdroj: [17])

Tento typ formátů se blízce podobá formátům typu RB (viz. kapitola 4), narozdíl od nich nejsou nenulové bloky ukládány v paměti postupně za sebou, ale je využíváno specifičtějších metod k určení jednotlivých *nnz* bloků.

7.1 Basic Hierarchical Format (BHF)

Vstupní matice A je rozdělena do $2^c \times 2^c$ disjunktních bloků, kde $c \in N^+$ je formální parametr. Na indexy nenulových bloků je tak potřeba pouze $log_2(\frac{n}{2^c})$ bitů. Na index v rámci bloku je potřeba c bitů.

Pro ukládání nenulových bloků a *nnz* prvků uvnitř každého bloku jsou využívány kombinace základních formátů COO (3.1) a CSR (3.2). BHF tak lze využít pro obecný typ matic. Mezi BHF patří následující kombinace [18]:

- COOCOO
- COOCSR
- CSRCOO
- CSRCSR

Formáty byly navrženy především pro ukládání velmi velkých řídkých matic na paralelních I/O systémech. Hlavním cílem je zmenšit počet přenášených dat. Nepřináší tak efektivní využití na vektorových počítačích ani grafických kartách (stejně jako formáty COO a CSR obecně).

7.1.1 COOCOO

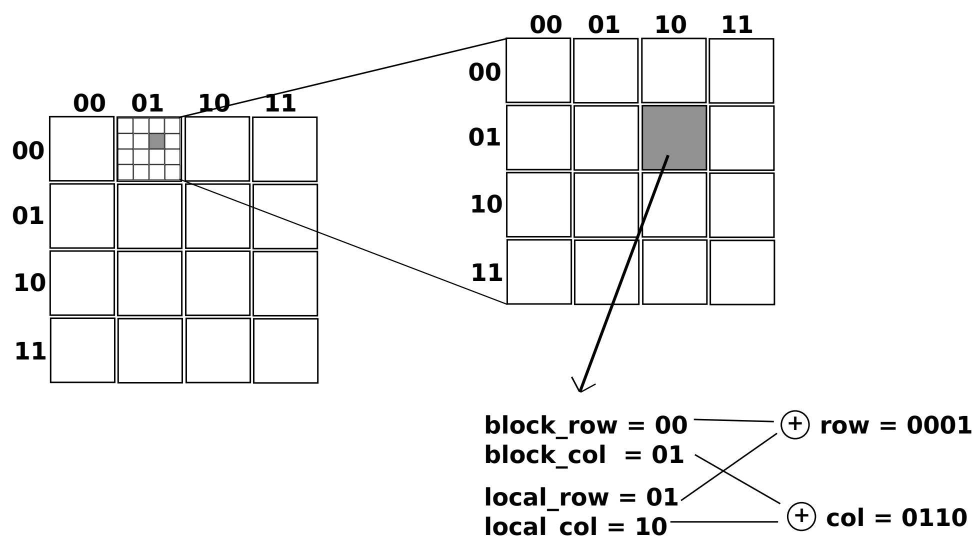
Nejjednodušší kombinace ukládá *nnz* prvky v rámci bloku formou COO, stejně tak pozice samotných nenulových bloků.

K uložení matice A v tomto formátu jsou zapotřebí následující informace:

- v rámci bloku:
  - *val*[] pole obsahující hodnoty všech *nnz* prvků
  - *local_row*[] pole s řádkovými indexy pro každý *nnz*, každý index má velikost c bitů, celkově $k \times c$, kde k je počet *nnz* prvků
  - *local_col*[] pole se sloupcovými indexy pro každý *nnz*, velikost je totožná jako u pole *local_row*[]
- v rámci matice:
  - *block_row*[] pole s řádkovými indexy všech nenulových bloků, každý index má velikost $log_2(\frac{n}{2^c})$, celkově $K \times log_2(\frac{n}{2^c})$, kde K je počet nenulových bloků

– *block_col*[] pole se sloupcovými indexy všech nenulových bloků, velikost stejná jako *block_row*[]

– *block_ptr*[] pole s pointery na první *nnz* každého bloku, počet pointrů je tedy K, každý pointer má velikost $log_2 k$ (je třeba namapovat všechny *nnz* prvky)



Obrázek 7.2: Princip hierarchického formátu COOCOO ($n = 16$, $c = 2$)

Konkrétní implementace formátu se nazývá COOCOO256. Parametr $c = 8$. Matice A je tak rozdělena na 256×256 ($2^8 \times 2^8$) submatic [38].

7.1.2 COOCSR

Formát odvozený od formátu COOCOO, jeho vznik byl podpořen faktem, že klasické CSR v porovnání s COO poskytuje viditelně paměťově efektivnější úložné schéma (viz. 3.2).

Struktura formátu na nejvyšší úrovni (level 0) je totožná jako u COOCOO, tedy pole *block_row*[], *block_col*[] a *block_ptr*[]. Na nižší úrovni (level 1) zůstávají stejná pole *val*[] a *local_col*[]. Pole *local_row*[] je nahrazeno polem *local_ptr_row*[], které obsahuje 2^c indexů na první *nnz* prvek v rámci každého řádku v bloku. Každý index v tomto poli má velikost $2c$ bitů (protože maximální počet prvků v bloku je $2^c \times 2^c$).

Konkrétní publikovaná implementace se nazývá COOCSR256. Parametr $c = 8$. Matice A je tak rozdělena na 256×256 ($2^8 \times 2^8$) submatic [38].

7.1.3 CSRCOO

Informace o nenulových blocích jsou uloženy ve formátu CSR. Level 0 má proto následující strukturu:

- *block_ptr_row*[] pole indexů na první nenulový blok v rámci každého řádku (zde je řádek myšlen jako seskupení 2^c řádek)
- *block_col*[] pole se sloupcovými indexy pro každý nenulový blok, velikost $K \times log_2(\frac{n}{2^c})$
- *block_ptr*[] pole s pointery na první *nnz* prvek každého bloku, velikost $K \times log_2 k$ (viz. 7.1.1)

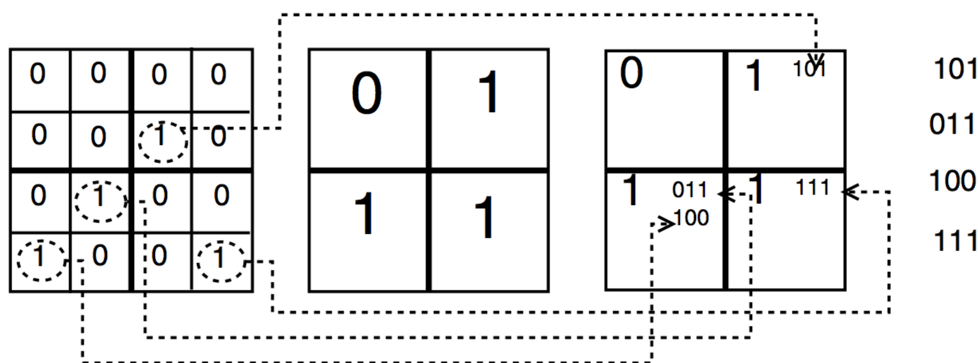
Struktura pro nižší level 1 zůstává stejná jako u COOCOO.

7.1.4 CSRCSR

Formát využívající CSR pro level 0 i level 1. Ze čtveřice BHF formátů je nejvíce paměťově úspornější. Struktura pro level 0 bude mít stejnou podobu jako CSRCOO, level 1 pak jako struktura COOCSR.

7.2 Advanced Hierarchical (AHF)

Hierarchický formát, který využívá bitmapové reprezentace a formátu COO. Vstupní matice A je rozdělena na $s \times s$ bloky. Každý bit v bitmapě reprezentuje jeden z těchto bloků (bit má hodnotu 1, pokud blok obsahuje alespoň jeden *nnz* prvek). Velikost bitmapy je tak $\lceil \frac{n}{s} \rceil \times \lceil \frac{n}{s} \rceil$. Prvky v rámci jednotlivých bloků jsou uloženy v COO formátu.



Obrázek 7.3: Princip reprezentace nenulových prvků v AHF formátu (zdroj: [18])

Princip zakódování nenulových prvků je zobrazen na obrázku 7.3. Pro indexy informující o pozici v bloku potřebuje každý prvek $2log_2s$ bitů. Každý *nnz* obsahuje navíc bitový příznak, který určuje, zda je prvek první ve svém bloku (bit nastaven na 1), pořadí se určuje po řádcích odshora dolů, zleva doprava.

Formát je, podobně jako BHF, určen primárně pro minimalizaci počtu přenášených dat na paralelních I/O systémech. V paralelním vektorovém zpracování nemá efektivní využití.

7.3 Adaptive Blocking Hierarchical Storage format (ABHSF)

Formát ABHSF vznikl za účelem zefektivnění původních hierarchických formátů BHF 7.1 a AHF 7.2, ty fungují na principu rozložení matice do submatic. Problém původních formátů je takový, že používají vždy fixní schéma pro všechny submatice, což nemusí být vždy efektivní. ABHSF používá různé schémata uložení pro různé bloky. Principem je, aby každý blok použil sobě optimální úložný formát [17].

Nevýhody použití fixního schématu:

- každý publikovaný hierarchický formát je prostorově optimální pro různé vstupní matice (tzn. žádný není optimální pro všechny matice)
- každý formát na nižším levelu je obecně prostorově optimální pro různé bloky (některé bloky mohou být uloženy neoptimálně v rámci zvoleného fixního formátu)

ABHSF rozděluje vstupní matici A do $\lceil \frac{n}{s} \rceil \times \lceil \frac{n}{s} \rceil$ bloků, o velikosti $s \times s$. Informace o blocích na levelu 0 jsou uloženy ve formátu COO. Pro bloky na levelu 1 je vybráno jedno z následujících schémat:

husté celý blok uložen jako hustá matice

bitmapa uložení *nnz* prvků v podobě bitmapy

COO blok uložen ve formátu COO

CSR blok uložen ve formátu CSR

Potřebná struktura pro uložení ABHSF pokrývající všechny výše zmíněná schémata obsahuje následující [57]:

- $block_row[]$ pole řádkových indexů bloků
- $block_col[]$ pole sloupcových indexů bloků
- $block_nnz[]$ pole s počtem *nnz* prvků v každém nenulovém bloku

- *type*[] pole s označením typu úložného schématu pro každý nenulový blok
- *bitmap*[] pole reprezentující bitmapový stream
- *local_row*[] pole s řádkovými indexy v rámci bloku
- *local_ptr_row*[] pole s pointery na prvky, které jsou první v jednotlivých řádcích bloku
- *local_col*[] pole se sloupcovými indexy v rámci bloku
- *val*[] pole s hodnotami *nnz* prvků

Kombinace různých formátů může značně zredukovat paměťové nároky (nejen v porovnání s COO/CSR, ale i s dalšími hierarchickými formáty). Formát lze případně rozšířit o další schémata pro ukládání bloků na levelu 1. Využití je podobné jako u ostatních hierarchických formátů, tedy primárně pro paralelní I/O počítače.

Použití formátu vyžaduje specifické algoritmy, které budou jednotlivým nenulovým bloků určovat optimální formát pro uložení.

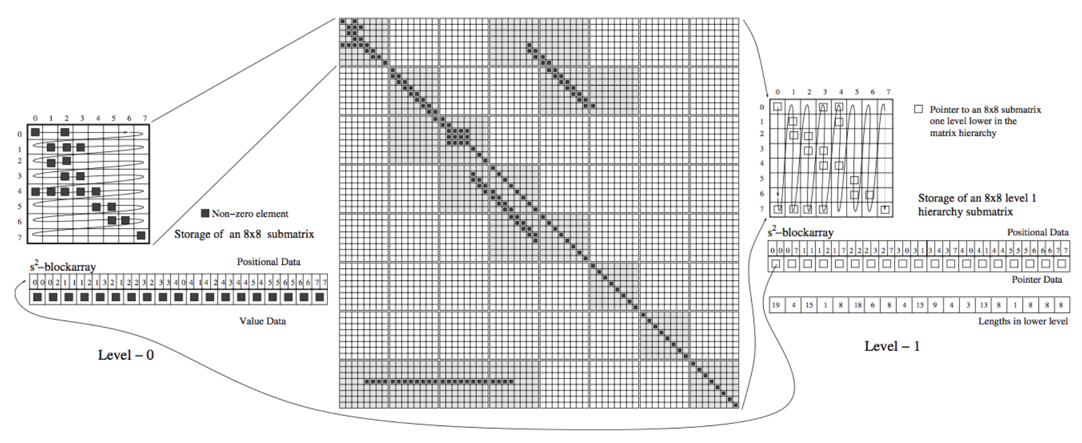
7.4 Hierarchical Sparse Matrix (HiSM)

Formát vznikl primárně jako formát pro práci s řídkými maticemi na vektorových procesorech. Snaží se řešit již dříve zmíněné problémy, hlavně tedy krátký vektor, indexovaný přístup a paměťová režie (viz. 4.4).

Struktura HiSM je kombinací upravené verze BBCS (viz. 4.4) a hierarchického způsobu ukládání řídkých matic. Princip je vidět na obrázku 7.4. Matice A je rozdělena do submatic $\lceil \frac{n}{s} \rceil \times \lceil \frac{n}{s} \rceil$, kde s je parametr cílové vektorové architektury (maximum elementů, které je možné zpracovat na dané architektuře v jedné instrukci). Každý blok $s \times s$ je v paměti uložen samostatně.

Nejnižší úroveň obsahuje po řádcích uložené hodnoty *nnz* prvků společně s pozičními údaji. Pro vektorové architektury se v praxi využívá typicky $s < 256$, tedy maximálně 2 bajty pro sloupcový a řádkový index dohromady. O stupeň vyšší úroveň obsahuje nenulové bloky, jejich pozici a pointer do paměti na nižší úroveň. Navíc je pro vyšší úrovně uložen v paměti vektor, který udává délku (počet *nnz* prvků v bloku nižší úrovně). Informace jsou uloženy po sloupcích, ale není to v této úrovni striktně vyžadováno. Úrovní lze mít i více než dvě [19].

Tento formát lze využít na klasických vektorových procesorech či SIMD jednotkách. Ovšem v případě, že se na nejnižší úrovni budou vyskytovat stále velmi řídké matice, nebude využití vektorových jednotek plně efektivní.



Obrázek 7.4: Struktura formátu HiSM (zdroj: [19])

7.5 Compressed Sparse Blocks (CSB)

Formát se zaměřuje na efektivní paralelní výpočet na many/multi-core architekturách. Prostorové požadavky jsou v podstatě stejné jako pro klasické CSR, které je ale pro paralelní zpracování nevhodné. Snahou je tedy zahladit nevýhody CSR popř. CSC. Formát nevyžaduje speciální vzory vstupní matice.

Matice A je rozdělena pomocí parametru β na čtvercové bloky o velikosti $\beta \times \beta$, kde submatice $A_{i,j}$ obsahuje:

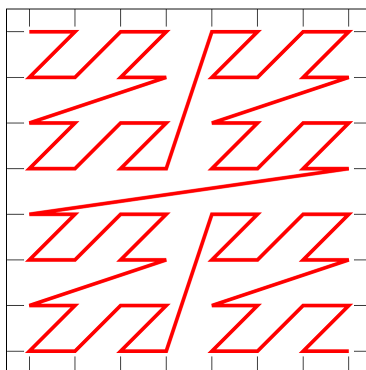
- $i\beta, i\beta + 1, \ldots (i + 1)\beta - 1$ řádky
- $j\beta, j\beta + 1, \ldots (j + 1)\beta - 1$ sloupce

Většina submatic $A_{i,j}$ je *hypersparse*[1]. Každý *nnz* prvek v submatici je uložen jako trojice (val, row_ind, col_ind). V rámci submatice je potřeba méně bitů k zakódování indexových hodnot. Submatice jsou v paměti uloženy souvisle za sebou. K určení začátku každé submatice je využito pole $block_ptr[]$, které ukazuje na první *nnz* prvky v rámci každé submatice.

Submatice a *nnz* v nich obsažené jsou ukládány pomocí Mortonova rozkladu (viz. obrázek 7.5). Toto uspořádání má pomoci k vyššímu výkonu a prostorové lokalitě při SpMV, navíc lze formát použít pro výpočet $A\vec{x}$ i $A^T\vec{x}$ [35].

Základní paralelismus je prováděn po jednotlivých segmentech řádků (tzv. „*blockrow*“), tedy submaticích, které jsou umístěny v řádkovém segmentu vedle sebe, neboli i-tý segment bude obsahovat submatice $A_{i,0}$ až $A_{i,n/\beta-1}$. Pro-

[1] Hypersparse znamená, že poměr mezi počtem nenulových prvků a řádem matice je následující: $nnz \ll n$



Obrázek 7.5: Mortonův rekurzivní rozklad

tože každý segment řádků pasuje na odlišnou část výstupního vektoru, nebudou vznikat konflikty při zápisu.

7.6 Expanded Compressed Sparse Blocks (eCSB)

Formát vycházející z CSB (viz. 7.5). Jeho rozšířená verze má za cíl zajistit efektivní využití pro použití SpMV i SpMTV při GPGPU.

Nedostatky CSB formátu při zpracování GPGPU:

- algoritmus provádí složitý proces rekurzivního rozkládání matice na malé kousky
- GPGPU požaduje stejný běh pro všechna vlákna
- každý blok má různý počet *nnz*, použití *warpu* na jednotlivé bloky může způsobovat plýtvání výpočetních zdrojů
- GPGPU nemá dynamické vyvažování zátěže

Upravená struktura pro eCSB obsahuje následující [46]:

- *val*[] pole hodnot *nnz* prvků
- *block_ptr*[] pole s pointery na první *nnz* prvek v rámci každého řádkového segmentu
- *block_ind*[] pole blokových indexů každého *nnz* prvku (zjednodušuje proces zpracování jednotlivými vlákny)
- *col_ind*[] pole sloupcových indexů v rámci bloku

- *row_ind*[] pole řádkových indexů v rámci bloku

Základní idea je zachovat CSB řazení *nnz* prvků. Hlavní problém je, že distribuce *nnz* se může výrazně lišit. K fyzickému uložení bloků je proto využíváno jednoho následujících formátů:

ELL distribuce *nnz* je poměrně pravidelná (viz. 3.9)

COO distribuce *nnz* je silně nepravidelná (viz. 3.1)

HYB pro zbývající případy (viz. 6.1)

Nenulové prvky jsou v rámci bloku uloženy ve formátu CSB (to zajišťuje sloupcový i řádkový přístup). V rámci *blockrow* jsou pak *nnz* prvky uloženy blok po bloku.

7.7 Extended Sparse Block Compressed Row Storage (SBCRSx)

Tento formát nenabízí prostorově efektivnější řešení a není to ani jeho cílem. Formát byl vyvinut z důvodu efektivního zpracování na procesorovém čipu. Vychází z formátu SBCRS, což je modifikace formátu BCSR (viz. 4.1), jediným rozdílem oproti BCSR je, že na nejnižší úrovni ukládá pouze *nnz* prvky [20].

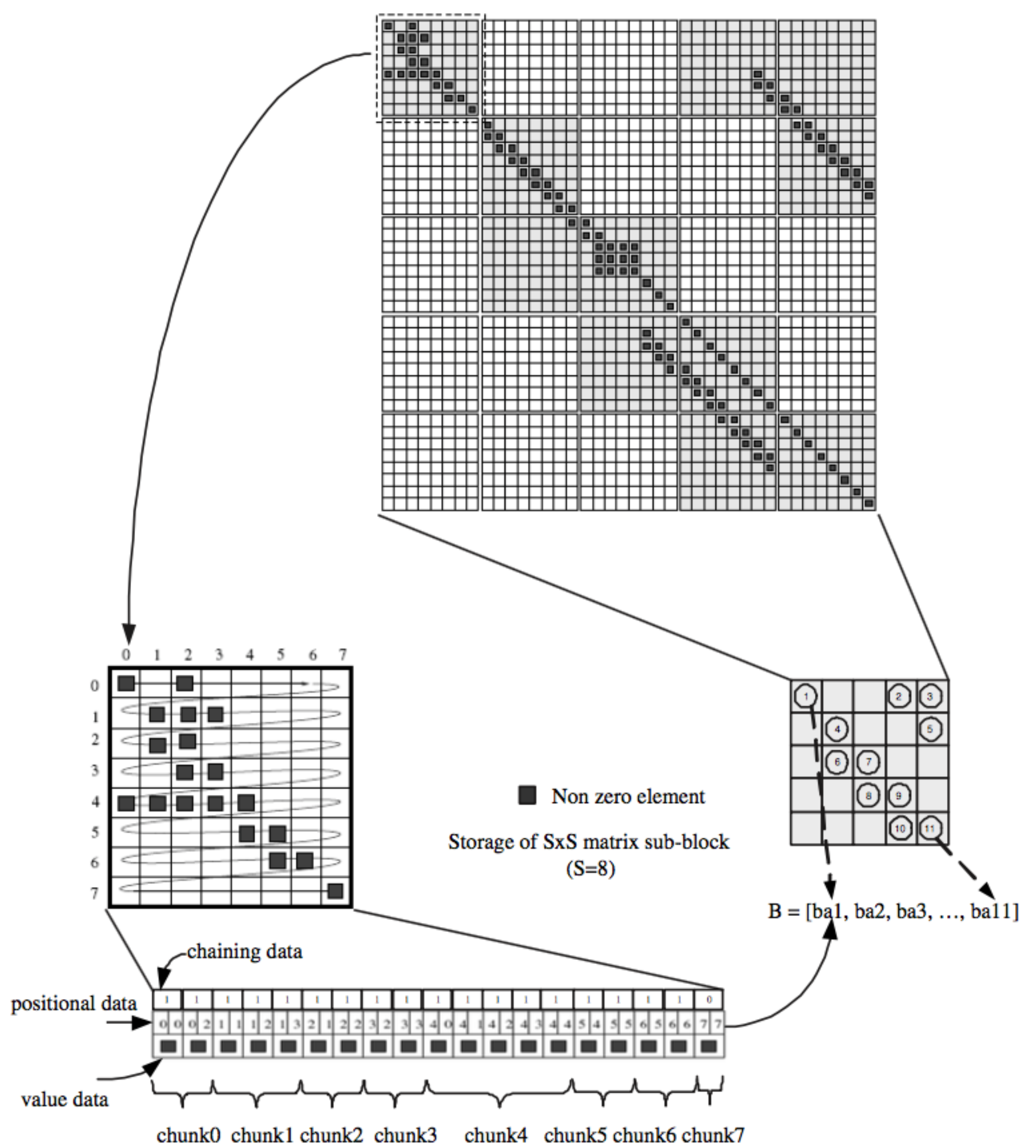
SBCRSx se snaží řešit problémy s přidanou režií při SpMV výpočtu a neefektivním vykonávání aritmetických operací, které jsou důsledkem nepřímé adresace operandů. Cílem je co největší znovuvyužití dat přímo na čipu.

Struktura je stejná jako pro BCSR (viz. 4.1), přibylo pouze pole *chain_bit*[], které obsahuje bity pro každý *nnz* prvek. Nenulové prvky jsou tak rozděleny do tzv. „*chunků*“. Všechny prvky v řádku, kde se nachází více než jeden *nnz*, mají *chain_bit* nastaven na hodnotu 1. Pozitivní bit říká, že se v řádku nachází další *nnz*, pokud je bit nulový, lze provést výpočet a výsledek uložit do výstupního vektoru. Toto zpracování by mělo zajistit zvýšení prostorové lokality dat.

Celkové prezentace formátu působí zmateně. V případě ohodnocování všech prvků v řádku, kde se nachází více než jeden *nnz*, *chain_bitem* s hodnotou 1, nebylo by možné jednotlivé *chunky* od sebe odlišit, je tedy nutné vždy poslednímu prvku v řádku přidělit *chain_bit* s nulovou hodnotou.

7.8 Pattern-Based Representation (PBR)

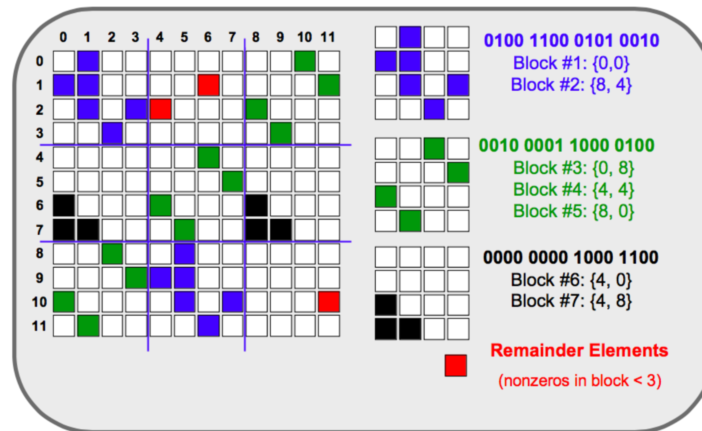
Motivací pro návrh formátu PBR je fakt, že většinu matic lze rozdělit do bloků, které sdílí malý počet různých vzorů (tzv. „*patterns*“), jádro pro násobení bloků se tak často opakuje. Autoři formátu se snaží zredukovat indexy, které stačí uchovávat v rámci každého vzoru.



Obrázek 7.6: Struktura formátu SBCRSx (zdroj: [20])

Princip spočívá v rozdělení matice A do $\lceil \frac{n}{R} \rceil \times \lceil \frac{n}{C} \rceil$ bloků. Poté je třeba v blocích nalézt opakující se vzory. Každý vzor je reprezentován pomocí COO (pozice prvního prvku v každém bloku se stejným vzorem) a bitovým vektorem o velikosti $R \times C$, který udává pozici *nnz* prvků v bloku daného vzoru, bitový řetězec je ukládán po řádcích (viz. obrázek 7.7). Bloky stejného vzoru jsou ukládány v paměti za sebou. Do vzorů neřadíme bloky, které mají méně než stanovený počet *nnz* prvků a dále vzory, které se moc často neopakují. Tyto

vyloučené vzory by přinesly minimální nebo žádné snížení režie pro ukládání indexů. Vyloučené bloky jsou ukládány pomocí CSR (viz. 3.2). Pro optimální využití formátu je třeba maximalizovat počet *nnz* prvků pokrytých formátem PBR. Pokrytí je závislé na volbě parametrů R a C [21].



Obrázek 7.7: Blokové vzory a jejich uložení ve formátu PBR (zdroj: [21])

Pro efektivní využití je třeba generátor kódu, který pro každý vzor generuje optimální výpočetní jádro. Paralelizace je možná, díky opakující se struktuře bloků lze snadno určit pracovní zátěž pro jednotlivé výpočetní jednotky, bude však nutná paralelní redukce (abychom se vyhnuli drahé atomizaci), protože výstupní vektor bude sdílený. Pro navržený formát je možné i využití vektorizace, je ale potřeba tomu přizpůsobit kódový generátor (viz. [48]).

Kapitola **8**

Kompresní formáty

V této kapitole jsou představeny formáty, které se primárně zaměřují na kompresi vstupní matice za cílem snížení celkové paměťové náročnosti a snížení přenášeného objemu dat mezi výpočetní jednotkou a hlavní pamětí. Součástí formátů jsou kódovací a dekódovací algoritmy. Využití pro GPGPU je z důvodu těchto složitějších algoritmů, které způsobují další režii a není možné je na GPU efektivně implementovat, značně neefektivní. Stejně tak pro SIMD jednotky není z důvodu poskládání dat v paměti tento typ formátů efektivní. Formáty lze rozdělit na:

- indexově-kompresní
  - Arithmetical Coding Based format
  - Delta-coded Sparse Row
  - CSR Delta Units
  - Row Pattern CSR
  - Compressed Sparse eXtended
- hodnotově-kompresní
  - CSR Values Indirect

8.1 Arithmetical Coding Based format (ACB)

Formát navržený pro snížení celkové paměťové náročnosti na ukládání struktury matice. Soustředí se tak pouze na kompresi indexů.

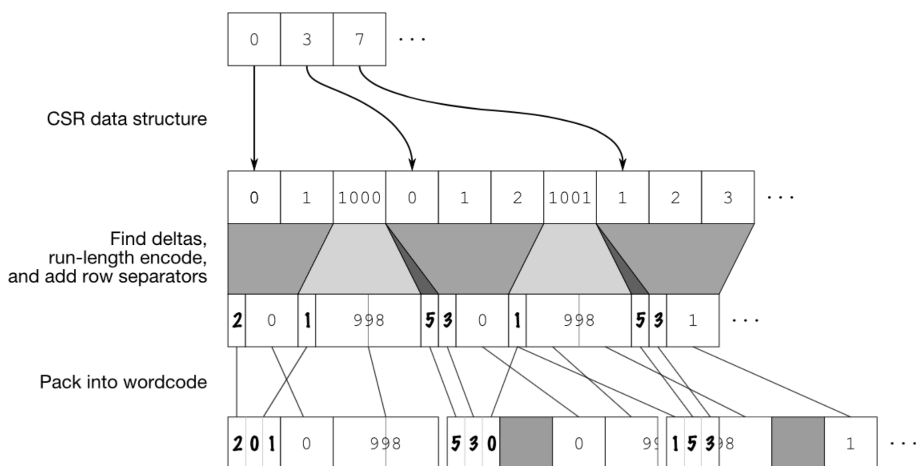
Vstupní matice A je reprezentována bitovým vektorem B velikosti $n \times n$. Každý bit pro *nnz* prvek je ohodnocen jako 1, ostatní nulové prvky hodnotou 0. Následně je vektor zkomprimován pomocí aritmetického kódování [27].

Nevýhodou formátu je vysoká režie na kódování a dekódování.

8.2 Delta-coded Sparse Row (DCSR)

Formát založen na bezztrátové kompresi indexů. Princip je postaven na kódování rozdílů mezi sloupcovými pozicemi *nnz* prvků v řádku matice. Snaží se využít minimální množství bajtů a poskytuje efektivní kompresi sousedních *nnz*. Navíc nabízí snadnou a rychlou kompresi a dekompresi [22].

Kódování vstupní matice je prováděno pomocí sady šesti příkazových kódů. Ke každému příkazu (kromě jednoho) je přiřazen jednobajtový argument. Na strukturu se tak lze dívat jako na seznam párů (příkaz, argument). Tyto páry se po třech ukládají do 32-bitového slova. Nejvyšší bajt slova obsahuje příkazové kódy, tři spodní bajty obsahují argumenty příkazů. Ukázka komprese je na obrázku 8.1.



Obrázek 8.1: Převod matice z CSR do DCSR (zdroj: [22])

8.3 CSR Delta Units (CSR-DU)

Formát navržený pro redukci *branchů*[1], které vznikají z důvodu velmi častého dekódování operací. Používá k tomu indexovou kompresi. Snaží se hledat pravidelnosti v řídké matici a využívat je speciálně přizpůsobenými *run-time* metodami.

Vstupní matice je rozdělena do logický jednotek (*units*), kde každá z nich je charakterizována typem pravidelnosti. Hlavním cílem je využít oblasti vykazující určitou úroveň hustoty. Každá jednotka je charakterizována požadovanou velikostí pro uložení *delta* hodnot, které vyjadřují sloupcovou vzdálenost mezi

[1] *branch* nebo *jump* je základní prostředek k větvení programu, při kterém se rozhoduje, jaká sekvence instrukcí se bude vykonávat

po sobě jdoucími *nnz* prvky (např. pokud je vzdálenost mezi dvěma *nnz* maximálně 256, formát vyžaduje jeden bajt k uložení *delta*). Každá jednotka může pokrývat maximálně jeden řádek.

Struktura každé jednotky obsahuje:

uflags typ jednotky (obsahuje informaci o velikosti *delta*, případně další informace jako např. *flag* značící začátek nové řádky)

usize počet *nnz* v jednotce

ujmp sloupcová vzdálenost od předchozí jednotky

ucis pole obsahující *delta* hodnoty pro zbývající sloupcové indexy

Příklad uložení vstupní matici A je reprezentován tabulkou 8.1. Zde každá jednotka představuje jeden řádek vstupní matice [24]. Vstupní matice A je následující:

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 0 & 6.3 & 0 & 7.7 & 0 & 8.8 \\ 0 & 0 & 1.1 & 0 & 0 & 0 \\ 0 & 0 & 2.9 & 0 & 3.7 & 2.9 \\ 9.0 & 0 & 0 & 1.1 & 4.5 & 0 \\ 1.1 & 0 & 2.9 & 3.7 & 0 & 1.1 \end{pmatrix}.$$

Tabulka 8.1: Reprezentace jednotek ve formátu CSR-DU, kde u8 je 8 bitů a NR je nový řádek (zdroj: [24])

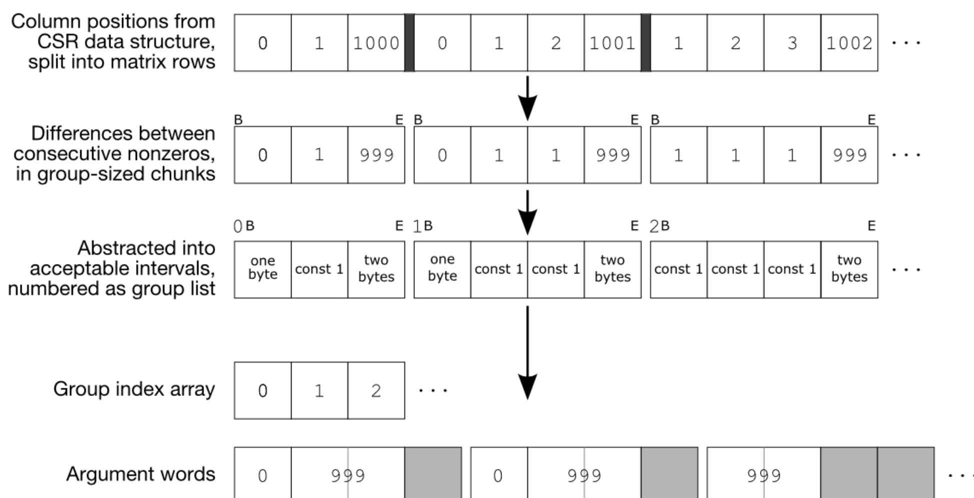
unit	**uflags**	**usize**	**ujmp**	**ucis**
0	u8,NR	2	0	1
1	u8,NR	3	1	2,2
2	u8,NR	1	2	-
3	u8,NR	3	2	2,1
4	u8,NR	3	0	3,1
5	u8,NR	4	0	2,1,2

8.4 Row Pattern CSR (RPCSR)

Formát nabízející kompresi indexů a jejich opětovnou *on-the-fly* dekompresi, při které nejsou dekompresovaná data ukládána do paměti. Částečně vychází

z formátu DCSR (8.2), nabízí efektivnější kompresi, nicméně za cenu větší časovou režie na kompresi [22].

Hlavní princip spočívá v rozdělení *delta* do intervalů, skupiny těchto intervalů (nazývány *patterns*) jsou uloženy v kompresované matici.



Obrázek 8.2: Komprese indexů do formátu RPCSR (zdroj: [22])

Základní jednotkou je index skupiny a pole jeho argumentů (viz. obrázek 8.2). Každá skupina obsahuje informace indikující pozici skupiny v řádku. Dekomprese formátu je jednodušší než samotná komprese.

8.5 Compressed Sparse eXtended (CSX)

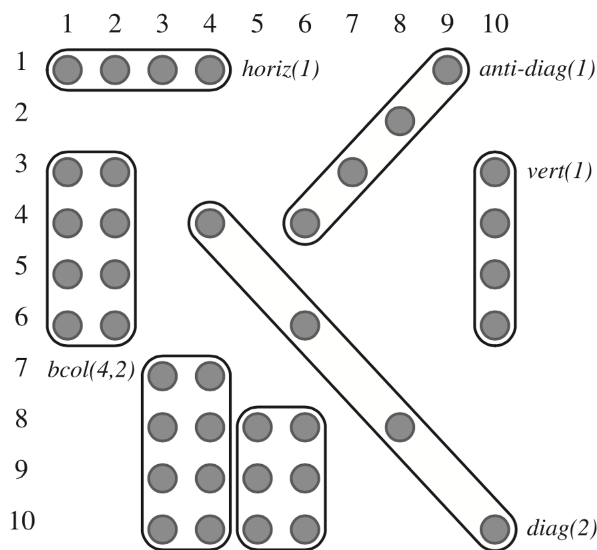
Formát navržen z důvodu minimalizace paměťových nároků na uložení struktury matice. Využívá kompresní schéma postavené na formátu CSR-DU (8.3).

CSX je schopen detekovat a zpracovat různé varianty substruktur (horizontální, vertikální, diagonální, antidiagonální a $2D$ bloky). To dovoluje značnou kompresi indexů vstupní matice [23]. Ukázka detekování substruktur viz. obrázek 8.3. Rozšíření o další typy substruktur je možné.

Struktura CSX nahrazuje původní CSR pole *col_ind*[] a *row_ptr*[] jedním polem *ctl*[] obsahujícím všechny požadované informace. CSX rozděluje (podobně jako CSR-DU) matici do jednotek. Každá jednotka představuje substrukturu uvnitř řídké matice.

Jednotka je rozdělena na dvě následující části:

head deskriptor jednotky



Obrázek 8.3: Detekce různých substruktur pro využití CSX (zdroj: [23])

body *delta* hodnoty jednotky

Deskriptor obsahuje:

id ID substruktury

size velikost jednotky (počet *nnz* prvků v jednotce)

nr bit, značící začátek nové řádky

ucol inicializační sloupcový index jednotky

rjmp bit, signalizující přítomnost prázdných řádek

ujmp počet prázdných řádek, které lze přeskočit

Hodnoty jsou ukládány standardně do pole *val*[], a to po jednotlivých substrukturách za sebou, v rámci substruktury probíhá ukládání v řádkovém pořadí.

Možná paralelizace se provádí rozdělením matice na vyvážený počet *nnz* prvků, až poté je prováděna detekce a kódování. Vzniká tak několik oddělených CSX struktur.

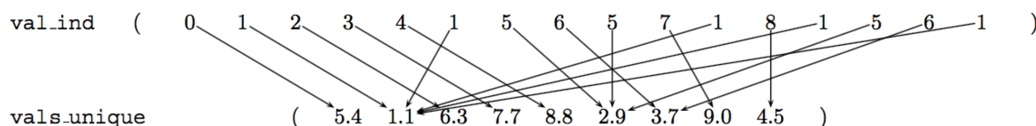
8.6 CSR Values Indirect (CSR-VI)

Formát specializující se na kompresi hodnot nenulových prvků. Je založený na klasickém CSR a je určený pouze pro matice, které obsahují malé množství unikátních *nnz* prvků.

Do paměti jsou ukládány pouze unikátní hodnoty nenulových prvků. Struktura odpovídá formátu CSR, pouze původní pole *val*[] je nahrazeno dvěma poli:

- *val_uniq*[] pole unikátních hodnot *nnz* prvků
- *val_ind*[] pole indexů do *val_uniq*[] pro každý *nnz* prvek

Aby mělo použití formátu smysl, měl by být každý index v poli *val_ind*[] znatelně menší než samotná hodnota *nnz* prvku. Velikost indexů je proto určena podle počtu unikátních hodnot *nnz*, které je potřeba adresovat (např. maximální počet unikátních *nnz* prvků bude 2^8, pak na adresaci stačí jeden bajt)



Obrázek 8.4: Nepřímé adresování unikátních hodnot ve formátu CSR-VI (zdroj: [24])

Paralelizace je možná, její nedostatky ale budou hodně podobné jako u formátu CSR. V případě, že bude počet unikátních hodnot malý, lze částečně snížit přístupy do hlavní paměti.

Kapitola 9

Cache-blokové formáty

Cache-blokové formáty se hodně podobají RB formátům (viz. 4), narozdíl od nich mají ale hlavní cíl zvýšení časové lokality dat [62]. Maximální využití cache pamětí výrazně snižuje přístup do hlavní paměti a zrychluje SpMV výpočet. Formáty jsou navržené pro *multi-core* systémy, jejich použití na GPU je proto nevhodné.

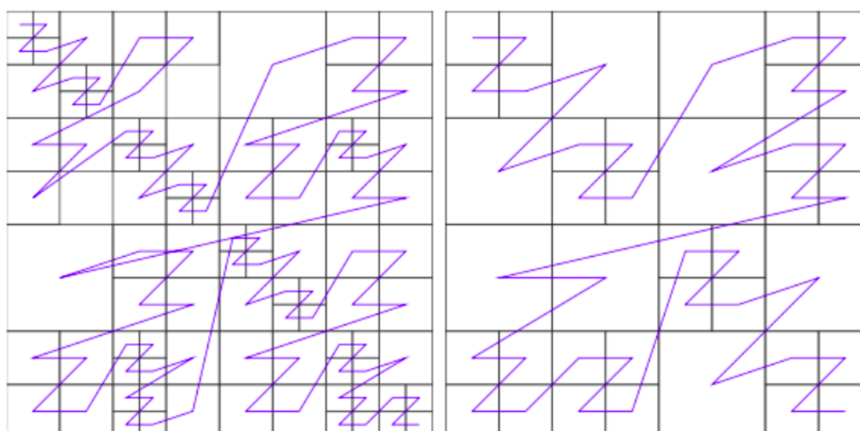
9.1 Recursive CSR/CSC (RCSR/RCSC)

Formát vznikl za účelem efektivnějšího zpracování řídké matice na *multicore* procesorech. Snaží se řešit nedostatečné využití výpočetních jednotek a nízké znovupoužití dat v lokální paměti procesoru. Při klasickém RB (viz. 4) je pro efektivní využití většinou nutná bloková struktura matice, tu ale spousta matic nemá.

Formát RCSR (příp. RCSC) dělí vstupní matici rekurzivně do submatic, ale pouze v případě, pokud obsahuje dostatečné množství *nnz* prvků. Formát je tak parametrizován hodnotou, která určuje hranici ukončení rekurzivního dělení. Při volbě tohoto parametru závisí hlavně na zvolené architektuře, v potaz se berou různé faktory, např. velikost číselné hodnoty, velikost pointeru, velikosti nejvzdálenější cache apod. [25].

Vstupní matice je dělena do kvadrátů (v případě obdélníkové matice jsou pro rekurzivní dělení potřeba speciální heuristiky), již nedělitelné submatice jsou poté v paměti ukládány v CSR/CSC pomocí Mortonova rozkladu (*Z-ordering*). Princip je přiblížen na obrázku 9.1.

Postupné zpracovávání blok po bloku zajišťuje lokalitu dat bez ohledu na řídkost matice. Pro každý blok je potřeba jen určitá část vstupního vektoru $\vec{x}$.

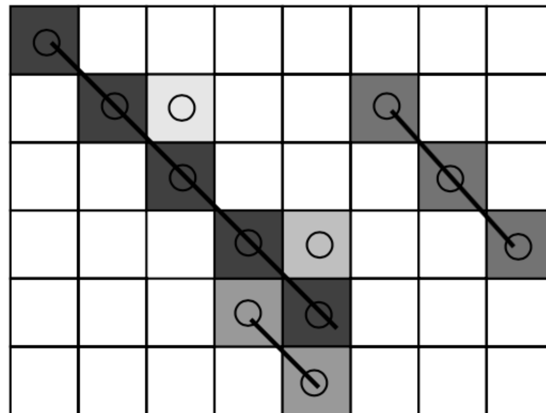


Obrázek 9.1: Rekurzivní dělení matice v závislosti na počtu nenulových prvků v bloku u RCSR/RCSC pro různě velké cache (zdroj: [25])

9.2 Cache-Adaptive heuristics-based Register Blocking (CARB)

Navržený formát staví na modifikaci formátu DRB (Daigonal register blocking), kde jsou ukládány husté diagonální bloky, jejichž velikost se může lišit. Vstupní matice může kromě diagonálních shluků obsahovat také *nnz* prvky, které jsou mimo diagonální pásy (viz. obrázek 9.2), jejich ukládání jako diagonály vede ke zbytečné paměťové náročnosti. Tento problém řeší C-DRB (Combined-DRB), který kombinuje formát DRB s formátem CSR, který zastřešuje všechny izolované *nnz* prvky. Potenciální problém C-DRB je takový, že počet DRB bloků může růst s velikostí a uspořádáním matice, s tím tedy rostou i nároky na uložení indexů každého bloku, problém řeší H-DRB (Heuristic DRB), který povoluje částečně husté bloky (tzv. „*H-dense blocks*“) podle heuristikou určeného parametru maximální řídkosti.

H-DRB je použit jako základ pro nový formát CARB. Nedostatkem H-DRB je fakt, že může obsahovat velmi dlouhé diagonální bloky, tedy pro efektivní využití cache nevhodné. CARB formát přidává parametr l (volený s ohledem na cache paměť dané architektury). Vstupní matice se tedy dělí na disjunktní horizontální pásy o počtu l řádků, diagonální bloky se pak tvoří pouze v rámci individuálních pásů a jsou zde řazeny podle pozice prvního horního prvku diagonály [26].



Obrázek 9.2: Ukázka diagonálních bloků a izolovaných prvků (zdroj: [26])

Kapitola 10

Formáty založené na stromové struktuře

Tento typ formátů využívá k reprezentaci řídké matice stromovou strukturu. Vnitřní uzly, reprezentující submatice, obsahují odkaz na další menší submatice, v listech stromu se pak nachází konkrétní hodnoty *nnz* prvků. Z důvodu složitější rekurzivní struktury není jejich použití pro GPU moc vhodné.

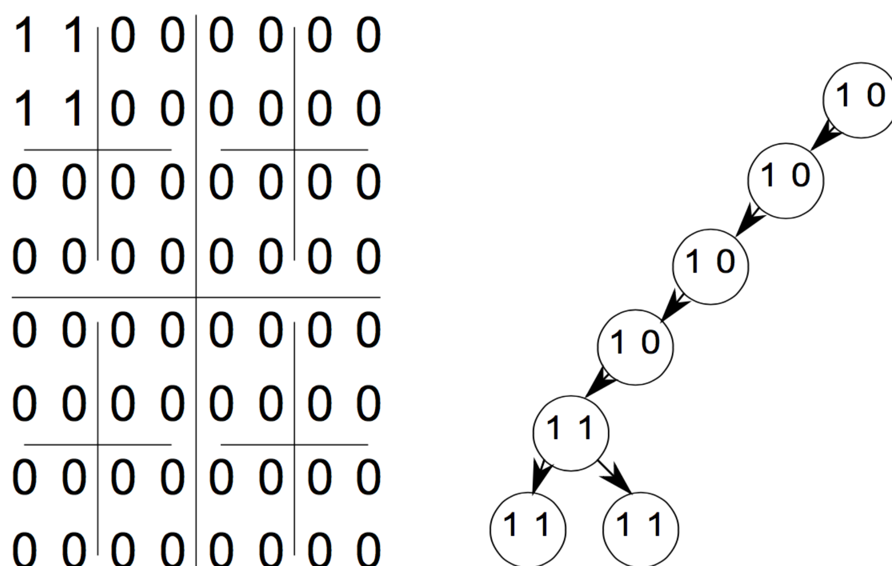
10.1 Minimal Binary Tree (MBT)

Formát vycházející z FBT (Full Binary Tree). FBT je hojně využívaná struktura, její princip je takový, že všechny vnitřní uzly mají dva potomky. Ve standardní implementaci má každý uzel dva pointery na následníky, v případě listů jsou pointery nahrazeny hodnotou *nnz* prvků. Při ukládání je řídká matice rozdělena na submatice, každý uzel stromu reprezentuje jednu submatici. Rozklad se provádí ve střídavých směrech (prvně horizontálně, poté vertikálně atd.):

- uzly v lichém stupni reprezentují rozdělení submatice na dvě poloviny podle osy x
- uzly v sudém stupni reprezentují rozdělení submatice na dvě poloviny podle osy y

FBT má z pohledu paměťové efektivity nevýhodu v ukládání pointerů, které způsobují zbytečnou režii. K eliminaci této nevýhody vznikl upravený MBT.

MBT ukládá všechny uzly jako bitový stream. S ohledem na fakt, že velikost matice je dána, lze snadno dopočítat pozice potomků, ukládání pointerů už tak není třeba. Všechny uzly obsahují pouze dva bity (1, pokud v následující submatici je alespoň jeden *nnz* prvek, jinak 0). Na obrázku 10.1 je vidět



Obrázek 10.1: Reprezentace vstupní matice (vlevo) pomocí stromu MBT (zdroj: [27])

princip rozdělení vstupní matice do stromové struktury, níže je uveden příklad přímé transformace do bitového streamu [27]:

$$S = MBT(A) = MBT\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} =$$

$$= \text{"11"} + MBT\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} + MBT\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} =$$

$$= \text{"11"}+\text{"01"}+\text{"11"}+MBT\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}+MBT\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}+MBT\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} =$$

$$= \text{"11"} + \text{"01"} + \text{"11"} + \text{"11"} + \text{"10"} + \text{"01"}+$$

$$+MBT\begin{pmatrix} 0 & 1 \end{pmatrix} + MBT\begin{pmatrix} 1 & 0 \end{pmatrix} + MBT\begin{pmatrix} 1 & 0 \end{pmatrix} + MBT\begin{pmatrix} 0 & 1 \end{pmatrix} =$$

$$= \text{"11"} + \text{"01"} + \text{"11"} + \text{"11"} + \text{"10"} + \text{"01"} + \text{"10"} + \text{"10"} + \text{"01"}.$$

Nevýhodou může být, že přesnou prostorovou náročnost nelze jednoznačně určit, vždy záleží na rozložení *nnz* prvků v matice.

Formát lze ještě mírně prostorově zefektivnit pomocí tzv. CBT (Compressed Binary Tree), který využívá skryté jedničky. Ta funguje na principu, že můžou nastat pouze situace 01, 10 a 11 (00 není pro neprázdnou submatici možná), pokud tedy přijde na vstup 0, zákonitě ji musí následovat 1, 1 lze tedy zanedbat (skrýt) a reprezentovat stav 01 pouze hodnotou 0. Jaké bude mít tato změna implementační dopady autoři nezmiňují, bitový stream totiž bude obsahovat 1-bitové i 2-bitové uzly.

Formát vznikl primárně za účelem minimalizace prostorové složitosti. Jeho využitelnost pro vektorové zpracování je téměř nulová.

10.2 Minimal Quadtree (MQT)

Formát MQT vychází z formátu QT, což je stromová datové struktura, kde všechny vnitřní uzly mají právě čtyři potomky. QT podobně jako BT rozděluje vstupní matici do submatic a také se potýká se stejným problémem prostorové náročnosti pro pointery na jednotlivé submatice. To vede k úpravě formátu na MQT.

V MQT jsou použity místo pointerů bity reprezentující jednotlivé submatice, jedničkový bit signalizuje neprázdnou submatici. Vše je v podstatě stejné jako v MBT (viz. 10.1). Jediný rozdíl je v použití čtyř bitů, které rozdělují submatici do čtyřech zón, není tedy třeba provádět rozklad střídavými směry. Princip je zobrazen na obrázku 10.2 [28].

Obrázek 10.2: Reprezentace matice pomocí MQT (zdroj: [28])

Uzly do bitového streamu jsou ukládány podle úrovně ve stromové struktuře, řazeno v řádkovém pořadí. Kompresi lze vylepšit pomocí CQT (Compressed Quadtree), kdy 0001 je reprezentováno jako 0, bude ovšem třeba řešit správné dekódování bitového streamu, podobně jako u CBT (viz. **??**).

Využitelnost formátu pro vektorové zpracování je stejná jako u formátu MBT.

Závěr

Od počátku éry vícejádrových a mnohojádrových systémů se významně rozrostl obor zabývající se ukládáním řídkých matic a jejich zpracováním na těchto systémech, o čemž svědčí více než čtyřicet nových moderních formátů, které se snaží ve větší míře pokročilé architektury efektivně využívat.

Z celkového přehledu nelze jednoznačně říct, které formáty jsou ty nejefektivnější a nejvhodnější pro výpočet SpMV. Vždy záleží na spoustě okolností, které mohou efektivitu formátů výrazně zvýšit, či úplně degradovat. Je třeba se soustředit nejen na typ výpočetní architektury a zvolený úložný formát, ale i na programátorský model, který by měl být schopen plně využít nabízeného potenciálu.

Moderní úložné formáty ve většině případů vycházejí ze základních formátů. Autoři se snaží původní struktury adaptovat na moderní systémy. Velkého rozmachu se dočkaly grafické akcelerátory, které se dostávají do popředí v ohledu paralelních výpočtů SpMV a patří jim pozornost cca poloviny všech popisovaných formátů v této práci.

Jedním z dalších cílů větší části formátů je dosažení časové, případně prostorové lokality dat (ideálně obojího). Typicky je toho dosaženo rozdělením vstupní matice do submatic, tímto způsobem lze i efektivně snížit datový přenos mezi hlavní pamětí a výpočetní jednotkou. O maximální snížení datového toku se pak starají speciální kompresní formáty, na druhou stranu ale obsahují většinou velkou vedlejší režii pro kompresi a dekompresi, a to znemožňuje jejich využití např. na zmíněných GPU.

Je nutné zmínit, že vývoj úložných formátů pro řídké matice je stále ještě v počátcích. Jedním z problémů je neexistence jednotné specifikace, podle které by bylo možné formáty porovnávat. Dosud publikované úložné formáty a jejich výkonnostní testy tak bývají většinou nevypovídající, jelikož každý autor provádí měření na různých vstupních maticích a různých HW architekturách. Někteří autoři ani neposkytují detailnější popis konkrétní implementace a reprezentace dat, což výzkum značně komplikuje. Přímé srovnání formátů mezi sebou tak není možné.

Návrh pro pokračování

Tato práce byla zaměřena na ucelený přehled úložných formátů pro řídké matice. Možností pro pokračování v oblasti těchto formátů je mnoho. Nabízí se např. zaměření na určitou konkrétní oblast, ať už konkrétní typy formátů, matic, či architektur. Dále pak možnost reálné implementace vybraných formátů, jejich testování a srovnání.

Práci lze také rozšířit o detailní analýzu srovnávacích kritérií, které byly nad rámec této práce. Srovnávací kritéria jsou uvedena v článku *Evaluation Criteria for Sparse Matrix Storage Formats* (viz. [29]).

Shrnutí

Všechny body zadání jsem splnil. Seznámil jsem se s principem ukládání řídkém matice v paměti počítače a s problémy, které tento proces obnáší. Dále jsem nastudoval a popsal principy úložných formátů od základních až po nově vznikající. Formáty jsme poté rozdělil podle jejich nejtypičtějších vlastností do charakteristických skupin.

U formátů, kde mělo smysl řešit paralelní výpočet SpMV, jsem uvedl vhodnou metodu a případně nastínil možnost efektivního řešení, v opačném případě jsem uvedl důvody, proč je paralelizace nevhodná.

Literatura

[1] Šimeček, I.; Sloup, J.: *Programování grafických akcelerátorů*. České vysoké učení technické v Praze, 2013, ISBN 978-80-01-05195-5.

[2] Kanter, D.: Inside fermi: NVIDIA's hpc push. září 2009. Dostupné z: `http://www.realworldtech.com/fermi/`

[3] Liu, X.; Smelyanskiy, M.; Chow, E.; aj.: Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, New York, NY, USA: ACM, 2013, s. 273–282.

[4] Stathis, P. T.: *Sparse matrix vector processing formats*. Dizertační práce, Technische Universiteit Delft, 2004.

[5] Maggioni, M.; Berger-Wolf, T.: AdELL: An adaptive warp-balancing ELL format for efficient sparse matrix-vector multiplication on GPUs. In *Proceedings of the 42nd International Conference on Parallel Processing (ICPP)*, 2013, s. 11–20.

[6] Choi, J. W.; Singh, A.; Vuduc, R. W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, ACM, leden 2010, s. 115–126.

[7] Vuduc, R. W.; Moon, H.-J.: Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High Performance Computing and Communications*, Heidelberg: Springer-Verlag, 2005, s. 807–816.

[8] Vuduc, R.: *Automatic performance tuning of sparse matrix kernels*. Dizertační práce, University of California, December 2003.

[9] Oberhuber, T.; Suzuki, A.; Vacata, J.: New row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA. In *Acta Technica*, ročník 56, 2011, s. 447–466.

[10] Heller, M.; Oberhuber, T.: Adaptive row-grouped CSR format for storing of sparse matrices on GPU. *CoRR*, 2012. Dostupné z: `http://arxiv.org/abs/1203.5737`

[11] Yang, M.; Sun, C.; Li, Z.; aj.: An improved sparse matrix-vector multiplication kernel for solving modified equation in large scale power flow calculation on CUDA. In *Proceedings of the 7th International Power Electronics and Motion Control Conference (IPEMC)*, ročník 3, 2012, s. 2028–2031.

[12] Monakov, A.; Lokhmotov, A.; Avetisyan, A.: Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *High Performance Embedded Architectures and Compilers*, Heidelberg: Springer-Verlag, 2010, s. 111–125.

[13] Vázquez, F.; Ortega, G.; Fernández, J.; aj.: Fast sparse matrix matrix product based on ELLR-T and GPU computing. In *10th IEEE International Symposium on Parallel and Distributed Processing with Applications*, IEEE Computer Society, 2012, s. 669–674.

[14] Zheng, C.; Gu, S.; Gu, T.-X.; aj.: BiELL: A bisection ELLPACK-based storage format for optimizing SpMV on GPUs. *Journal of Parallel and Distributed Computing*, ročník 74, č. 7, 2014: s. 2639–2647.

[15] Kreutzer, M.; Hager, G.; Wellein, G.; aj.: Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, Washington, DC, USA: IEEE Computer Society, 2012, str. 1696–1702.

[16] Kreutzer, M.; Hager, G.; Wellein, G.; aj.: A unified sparse matrix data format for modern processors with wide SIMD units. ArXiv e-prints, 2013.

[17] Langr, D.; Šimeček, I.; Tvrdík, P.; aj.: Adaptive-blocking hierarchical storage format for sparse matrices. In *Federated Conference on Computer Science and Information Systems (FedCSIS)*, NEW YORK, NY 10017 USA: IEEE Xplore Digital Library, září 2012, s. 545–551.

[18] Šimeček, I.; Langr, D.; Tvrdík, P.: Space-efficient sparse matrix storage formats for massively parallel systems. In *Proceedings of the 14th IEEE International Conference of High Performance Computing and Communications (HPCC)*, IEEE Computer Society, 2012, s. 54–60.

[19] Stathis, P.; Vassiliadis, S.; Cotofana, S.: A hierarchical sparse matrix storage format for vector processors. In *Proceedings of the 17th International*

Symposium on Parallel and Distributed Processing, IPDPS '03, Washington, DC, USA: IEEE Computer Society, 2003, str. 61.

[20] Smailbegovic, F.; Gaydadjiev, G. N.; Vassiliadis, S.: Sparse matrix storage format. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, 2005, s. 445–448.

[21] Belgin, M.; Back, G.; Ribbens, C. J.: Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, New York, NY, USA: ACM, 2009, str. 100–109.

[22] Willcock, J.; Lumsdaine, A.: Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, New York, NY, USA: ACM, 2006, s. 307–316.

[23] Karakasis, V.; Gkountouvas, T.; Kourtis, K.; aj.: An extended compression format for the optimization of sparse matrix-vector multiplication. In *IEEE Transactions on Parallel and Distributed Systems*, ročník 24, říjen 2013, s. 1930–1940.

[24] Kourtis, K.; Goumas, G.; Koziris, N.: Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression. In *37th International Conference on Parallel Processing*, ICPP '08, září 2008, s. 511–519.

[25] Martone, M.; Filippone, S.; Tucci, S.; aj.: Utilizing recursive storage in sparse matrix-vector multiplication—preliminary considerations. In *Proceedings of the ISCA 25th International Conference on Computers and Their Applications, CATA 2010*, 2010, s. 300–305.

[26] Tvrdík, P.; Šimeček, I.: A new diagonal blocking format and model of cache behavior for sparse matrices. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, PPAM'05, Heidelberg: Springer-Verlag, 2006, s. 164–171.

[27] Šimeček, I.; Langr, D.; Tvrdík, P.: Tree-based space efficient formats for storing the structure of sparse matrices. In *Scalable Computing: Practice and Experience*, ročník 15, 2014, s. 1–20.

[28] Šimeček, I.; Langr, D.; Tvrdík, P.: Minimal quadtree format for compression of sparse matrices storage. In *Proceedings of the 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, IEEE Computer Society, září 2012, s. 359–364.

[29] Langr, D.; Tvrdík, P.: Evaluation Criteria for Sparse Matrix Storage Formats. *IEEE Transactions on parallel and distributed systems.*

[30] Davis, T. A.; Hu, Y. F.: The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, ročník 38, č. 1, 2011.

[31] Langr, D.; Šimeček, I.; Tvrdík, P.: Storing sparse matrices in the adaptive-blocking hierarchical storage format. In *Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2013)*, IEEE Xplore Digital Library, září 2013, str. 479–486.

[32] Šimeček, I.; Langr, D.; Tvrdík, P.: Space efficient formats for structure of sparse matrices based on tree structures. In *Proceedings of the 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, IEEE Computer Society, září 2013, s. 344–351.

[33] Abu-Sufah, W.; Karim, A.: An effective approach for implementing sparse matrix-vector multiplication on graphics processing units. In *Proceedings of the IEEE 14th International Conference on High Performance Computing and Communication (HPCC)*, 2012, s. 453–460.

[34] Koza, Z.; Matyka, M.; Szkoda, S.; aj.: Compressed multirow storage format for sparse matrices on graphics processing units. *SIAM Journal on Scientific Computing*, ročník 36, č. 2, 2014: s. C219–C239.

[35] Buluc, A.; Fineman, J. T.; Frigo, M.; aj.: Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, 2009, s. 233–244.

[36] Kourtis, K.; Karakasis, V.; Goumas, G.; aj.: CSX: An extended compression format for SpMV on shared memory systems. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, 2011, s. 247–256.

[37] Meyer, J. C.; Cebrian, J. M.; Natvig, L.; aj.: Energy-efficient sparse matrix autotuning with CSX - a trade-off study. In *Proceedings of the 2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops & PhD Forum*, Los Alamitos, CA, USA: IEEE Computer Society, 2013, s. 931–937.

[38] Šimeček, I.; Langr, D.: Space-efficient sparse matrix storage formats with 8-bit indices. In *Proceedings of Seminar on Numerical Analysis (SNA)*, leden 2012, s. 161–164.

[39] Kourtis, K.; Goumas, G.; Koziris, N.: Optimizing sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, New York, NY, USA: ACM, 2008, s. 87–96.

[40] Kourtis, K.; Goumas, G.; Koziris, N.: Exploiting compression opportunities to improve SpMxV performance on shared memory systems. *ACM Trans. Archit. Code Optim.*, ročník 7, č. 3, prosinec 2010: str. 16:1–16:31.

[41] Feng, X.; Jin, H.; Zheng, R.; aj.: Optimization of sparse matrix-vector multiplication with variant CSR on GPUs. In *Proceedings of the IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, 2011, s. 165–172.

[42] Karakasis, V.; Goumas, G.; Koziris, N.: A comparative study of blocking storage methods for sparse matrices on multicore architectures. In *International Conference on Computational Science and Engineering*, CSE '09, srpen 2009, s. 247–256.

[43] Vázquez, F.; Fernández, J. J.; Garzón, E. M.: A new approach for sparse matrix vector product on NVIDIA GPUs. In *Concurrency and Computation: Practice and Experience*, ročník 23, 2011, s. 815–826.

[44] Cao, W.; Yao, L.; Li, Z.; aj.: Implementing sparse matrix-vector multiplication using CUDA based on a hybrid sparse matrix format. In *Proceedings of the International Conference on Computer Application and System Modeling (ICCASM)*, ročník 11, 2010, s. V11–161–V11–165.

[45] Vázquez, F.; Ortega, G.; Fernandez, J. J.; aj.: Improving the performance of the sparse matrix vector product with GPUs. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, Washington, DC, USA: IEEE Computer Society, 2010, s. 1146–1151.

[46] Tao, Y.; Deng, Y.; Mu, S.; aj.: GPU accelerated sparse matrix-vector multiplication and sparse matrix-transpose vector multiplication. In *Concurrency and Computation: Practice and Experience*, 2014.

[47] 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*, SC '09, New York, NY, USA: ACM, 2009, s. 18:1–18:11.

[48] Belgin, M.; Back, G.; Ribbens, C. J.: A library for pattern-based sparse matrix vector multiply. *International Journal of Parallel Programming*, ročník 39, č. 1, 2011: str. 62–87.

[49] Martone, M.; Filippone, S.; Paprzycki, M.; aj.: On BLAS operations with recursively stored sparse matrices. In *Proceedings of the 2010 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC'10, Washington, DC, USA: IEEE Computer Society, 2010, str. 49–56.

[50] Martone, M.; Filippone, S.; Paprzycki, M.; aj.: On the usage of 16 bit indices in recursively stored sparse matrices. In *12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, září 2010, s. 57–64.

[51] Martone, M.; Filippone, S.; Tucci, S.; aj.: Use of hybrid recursive CSR/COO data structures in sparse matrix-vector multiplication. In *Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on*, říjen 2010, str. 327–335.

[52] Dziekonski, A.; Lamecki, A.; Mrozowski, M.: A memory efficient and fast sparse matrix vector product on a GPU. In *Progress in Electromagnetics Research*, ročník 116, 2011, s. 49–63.

[53] Šimeček, I.; Tvrdík, P.: Sparse matrix-vector multiplication—final solution? In *Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics*, PPAM'07, Heidelberg: Springer-Verlag, 2008, s. 156–165.

[54] Montagne, E.; Ekambaram, A.: An optimal storage format for sparse matrices. In *Information Processing Letters*, ročník 90, 2004, s. 87–92.

[55] INTEL: Intel® Xeon Phi™ Coprocessor – the Architecture. listopad 2012. Dostupné z: `https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner`

[56] Barrett, R.; Berry, M.; Chan, T. F.; aj.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia: PA, USA: SIAM, druhé vydání, 1994.

[57] Langr, D.; Šimeček, I.; Tvrdík: Storing Sparse Matrices to Files in the Adaptive-Blocking Hierarchical Storage Format. In *Federated Conference on Computer Science and Information Systems*, IEEE, 2013, s. 479–486.

[58] Melhem, R.; Gannon, D.: Toward Efficient Implementation of Preconditioned Conjugate Gradient Methods On Vector Supercomputers. *International Journal of High Performance Computing Applications*, ročník 1, č. 1, 1987: s. 70–98.

[59] Shahnaz, R.; Usman, A.; Chughtai, I. R.: Review of storage techniques for sparse matrices. In *9th International Multitopic Conference*, IEEE Computer Society, prosinec 2005, s. 1–7.

[60] Grimes, R. G.; Kincaid, D. R.; Young, D. M.: ITPACKV 2D User's Guide. *CNA-232*, 1989. Dostupné z: `http://rene.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/`

[61] Bell, N.; Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, 2009.

[62] Nishtala, R.; Vuduc, R.; Demmel, J.; aj.: When cache blocking of sparse matrix vector multiply works and why. *Journal of Applicable Algebra in Engineering, Communication and Computing*, květen 2007: s. 297–311.

Příloha **A**

Seznam použitých zkratek

ABHSF Adaptive Blocking Hierarchical Storage Format

ACB Arithmetical Coding Based format

AdELL Adaptive warp-balancing ELLPACK

AHF Advanced Hierarchical Format

ArgCSR Adaptive row-grouped Compressed Row Storage

BBCS Blocked-Based Compression Storage

BCSC Blocked Compressed Column Storage

BCSD Blocked Compressed Daigonal Storage

BCSD-DEC Decomposed Block Compressed Sparse Diagonals

BCSR Blocked Compressed Row Storage

BCSR-DEC Decomposed Block Compressed Row Storage

BELLPACK Blocked ELLPACK

BHF Basic Hierarchical Format

BiELL Bisection ELLPACK

BiJAD Bisection Jagged Diagonal storage

BT Binary Tree

BTJDS Blocked Transpose Jagged Diagnal Storage

CARB Cache-Adaptive heuristics-based Register Blocking

CBT Compressed Binary Tree

C-DRB Combined Daigonal Register Blocking

CDS Compressed Daigonal Storage

CMRS Compressed Multi-Row Storage

COO Coordinate Format

CQT Compressed Quadtree

CSB Compressed Sparse Blocks

CSC Compressed Column Storage

CSR Compressed Row Storage

CSR-DU Compressed Row Storage Delta Units

CSR-VI Compressed Row Storage Values Indirect

CSX Compressed Sparse eXtended

DCSR Delta-Coded Sparse Row

DIAG Diagonal Format

DRB Daigonal Register Blocking

eCSB Expanded Compressed Sparse Blocks

ELL-R ELLPACK-R

EOB End Of Block

EOM End Of Matrix

EOR End Of Row

ESB ELLPACK Sparse Block

FBT Full Binary Tree

GPGPU General-Purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

H-DRB Heuristic Daigonal Register Blocking

HiSM Hierarchical Sparse Matrix

HPC High-Performance Computing

HW Hardware

HYB Hybrid ELLPACK/Coordinate Format

ICSR Improved Compressed Row Storage

JAD Jagged Diagonal storage

JD Jagged Diagonal

JDS Jagged Diagonal Storage

MBT Minimal Binary Tree

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

MMX MultiMedia eXtension

MQT Minimal Quadtree

nnz non-zero

PBR Pattern-Based Representation

pJDS Padded Jagged Diagonal Storage

QT Quad Tree

RB Register Blocking

RCSC Recursive Compressed Column Storage

RCSR Recursive Compressed Row Storage

RgCSR Row-grouperd Compressed Row Storage

RPCSR Row Pattern Compressed Row Storage

RSDIAG Row Segmented Diagonal

SBCRS Sparse Block Compressed Row Storage

SBCRSx Extended Sparse Block Compressed Row Storage

SELLPACK Sliced ELLPACK

SIC Compressed Row Storage with Segmented Interleave Combination

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SISD Single Instruction Single Data

SKS Skyline Storage

SM Streaming Multiprocesor

SMURB Specified Machine parameters Using Register Blocking

SpMTV Sparse Matrix-Transpose-Vector multiplication

SpMV Sparse Matrix-Vector multiplication

SSE Streaming SIMD Extensions

TD Tag Directories

TJD Transpose Jagged Diagonal

TJDS Transpose Jagged Diagnal Storage

UBCSR Unaligned Block Compressed Row Storage

VPU Vector Processing Unit

ZR Zero Row

Příloha B

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
src
  └── thesis ........................ zdrojová forma práce ve formátu LaTeX
text ........................ text práce
  └── thesis.pdf ........................ text práce ve formátu PDF
```