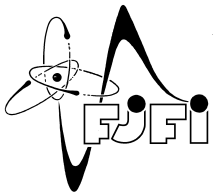
České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská



Paralelní algoritmy lineární algebry pro GPU

Parallel algorithms of linear algebra on GPU

Diplomová práce

Autor: **Bc. Matouš Fencl**

Vedoucí práce: **Ing. Tomáš Oberhuber, Ph.D.**

Akademický rok: 2019/2020

České vysoké učení technické v Praze, Fakulta jaderná a fyzikálně inženýrská

Katedra: matematiky

Akademický rok: 2019/2020

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Matouš Fencl
Studijní program:	Aplikace přírodních věd
Obor:	Aplikované matematicko-stochastické metody
Název práce (česky):	Paralelní algoritmy lineární algebry pro GPU
Název práce (anglicky):	Parallel algorithms of linear algebra on GPU

Pokyny pro vypracování:

1. Implementujte a otestujte paralelní algoritmus pro násobení symetrických řídkých matic s vektorem pomocí grafového obarvení.
2. Porovnejte výše odvozený algoritmus s algoritmem založeným na atomických instrukcích.
3. Implementujte paralelní algoritmus pro Gaussovu eliminační metodu na GPU.
4. Implementujte distribuovanou verzi paralelní Gaussovy eliminační metody.
5. Naměřte urychlení implementovaných algoritmů.

Doporučená literatura:

1. Y. Saad, Iterative Methods for Sparse Linear Systems. Second Edition: Society for Industrial and Applied Mathematics, 2003, ISBN 978-0898715347.
2. J. Cheng, M. Grossman, T. McKercher, Professional CUDA C Programming. First Edition: Worx, 2014, ISBN 978-1118739327.
3. D. B. Kirk, W. W. Hwu, Programming Massively Parallel Processors, Third Edition: A Hands-on Approach. Morgan Kaufmann, 2016, ISBN 978-0128119860.
4. N. Bell, M. Garland, Efficient Sparse Matrix-Vector Multiplication on CUDA, NVIDIA Technical Report NVR-2008-004, 2008.
5. D. Langr and P. Tvrdík, Evaluation Criteria for Sparse Matrix Storage Formats, in IEEE Transactions on Parallel and Distributed Systems 27, 2, 428-440, 2016.

Jméno a pracoviště vedoucí diplomové práce:

Ing. Tomáš Oberhuber, Ph.D.
Katedra matematiky, FJFI ČVUT v Praze, Trojanova 13, 120 00 Praha 2

Jméno a pracoviště konzultanta:

Datum zadání diplomové práce: 31.10.2019

Datum odevzdání diplomové práce: 4.5.2020

Doba platnosti zadání je dva roky od data zadání.

V Praze dne 15. října 2019

.....
garant oboru

.....
vedoucí katedry

.....
děkan

Poděkování:

Chtěl bych zde poděkovat především svému školiteli Ing. Tomáši Oberhuberovi, Ph.D. za pečlivost, ochotu, vstřícnost a odborné i lidské zázemí při vedení mé diplomové práce.

Čestné prohlášení:

Prohlašuji, že jsem tuto práci vypracoval samostatně a uvedl jsem všechnu použitou literaturu.

V Praze dne 24. července 2020 Matouš Fencl

Název práce:

Paralelní algoritmy lineární algebry pro GPU

Autor: Bc. Matouš Fencl

Obor: Aplikované matematicko-stochastické metody

Druh práce: Diplomová práce

Vedoucí práce: Ing. Tomáš Oberhuber, Ph.D., České vysoké učení technické v Praze, Fakulta jaderná a fyzikálně inženýrská, Katedra matematiky

Abstrakt: Cílem této práce je návrh algoritmu pro násobení řídkých symetrických matic na GPU a výpočet soustav lineárních rovnic rovněž na GPU. Jako numerická metoda pro řešení soustav lineárních rovnic byla zvolena Gaussova Eliminační metoda. Implementace je provedena pomocí jazyka C++ na CPU a pomocí nVidia CUDA a knihovny MPI na GPU. Dále se pro implementaci použila knihovna TNL a zároveň se výsledné algoritmy stávají součástí této knihovny. V textu jsou prezentovány časy výpočtů Gaussovy Eliminační metody a urychlení implementovaných algoritmů pro CPU i GPU. Dále jsou prezentovány výsledky SpMV ve formě tabulek a grafů s časy výpočtů včetně urychlení na CPU i GPU.

Klíčová slova: Gaussova Eliminační metoda, GEM, TNL, MPI, nVidia, nVidia CUDA, GPU, paralelizace, SpMV, Ellpack, symetrický Ellpack, atomické instrukce, speciální maticové obarvení

Title:

Parallel algorithms of linear algebra on GPU

Author: Bc. Matouš Fencl

Abstract: The aim of this work is to suggest algorithm for multiplication of sparse symmetric matrices with vectors on GPU and numerical calculation of systems of linear equations. Gauss Elimination method is a chosen numerical method used to solve systems of linear equations. Its implementation is accomplished using TNL library. Final algorithms are part of this library. The implementation is attained via C++ programming language on CPU and via nVidia CUDA toolkit and MPI library on GPU. Acquired results are presented in a form of figures, tables of computation times and efficiency of implemented algorithms for CPU and GPU. Further, results of SpMV are presented in a form of tables and figures with computational time and efficiency for CPU and GPU.

Key words: Gauss Elimination Method, GEM, TNL, MPI, nVidia, nVidia CUDA, GPU, parallelization, SpMV, Ellpack, symmetrick Ellpack atomic instructions, special matrix colorization

Obsah

1 Nástroje **13**
1.1 CUDA . . . 13
1.2 OpenMP . . . 16
1.3 TNL . . . 16
1.4 MPI . . . 17

2 Násobení řídké symetrické matice s vektorem **19**
2.1 Ellpack formát . . . 19
2.1.1 Symetrický Ellpack formát . . . 22
2.2 Paralelní SpMV . . . 25
2.2.1 SpMV s atomickými instrukcemi . . . 25
2.2.2 SpMV s maticovým obarvením . . . 27

3 Výsledky násobení matice s vektorem **39**
3.1 CPU SpMV s OpenMP . . . 41
3.1.1 CPU SpMV s jednoduchou přesností a OpenMP . . . 41
3.1.2 CPU SpMV s dvojitou přesností a OpenMP . . . 43
3.2 SpMV s jednoduchou přesností . . . 46
3.3 SpMV s dvojitou přesností . . . 48

4 Gaussova eliminační metoda **53**
4.1 Sekvenční Gaussova eliminační metoda . . . 53
4.2 Paralelní Gaussova eliminační metoda . . . 55
4.3 Paralelní Gaussova eliminační metoda s MPI . . . 58

5 Výsledky Gaussovy Eliminační metody **63**
5.1 GEM s jednoduchou přesností . . . 65
5.2 GEM s dvojitou přesností . . . 67

5.3 MPI GEM 68

Úvod

Numerické výpočty jsou důležitou součástí aplikované matematiky v oblastech, kde se analytické výsledky nedají vůbec nebo jen velmi obtížně najít. Z toho důvodu je snaha využít maximálního výpočetního výkonu dnešních počítačů i počítačových klastrů, jenž jsou pro tyto výpočty určeny. Klasické počítačové procesory již v dnešní době dosahují maximálního možného výkonu, který jim fyzikální zákony dovolí a proto je snaha vyvíjet paralelní programy fungující na grafických akcelerátorech. Jednou z možností je použití grafických akcelerátorů od značky nVidia za pomoci programovací architektury CUDA.

V první kapitole této práce si rychle představíme nástroje pro implementaci dvou základních operací, které jsou v numerické matematice řešeny velmi často. Jedná se o násobení řídké symetrické matice s vektorem a řešení soustav lineárních rovnic. V mnoha případech je mezivýsledek zachován v řídké symetrické matici, kterou je nutno následně násobit s vektory. Ve druhé kapitole tedy ukážeme dva algoritmy pro násobení řídké symetrické matice s vektorem, kde jako první způsob zvolíme atomické instrukce a jako druhý způsob zvolíme speciální obarvení matice pro předejití konfliktů v přístupech do paměti. Dále ve druhé kapitole porovnáme obě metody na testovacích maticích. Ve třetí a poslední kapitole ukážeme implementaci nejjednodušší verze Gaussovy eliminační metody na CPU a naši paralelní verzi pro GPU, kterou následně rozšíříme pomocí MPI pro výpočetní klastry. Na konci této kapitoly také ukážeme výsledky obou implementací na testovacích maticích.

Kapitola 1

Nástroje

V této kapitole ukážeme nástroje, které jsme pro řešení algebraických úloh pro CPU a GPU použili. Jedná se o CUDA toolkit, který umožňuje programování na grafických akcelerátorech značky nVidia. Následně si rychle představíme OpenMP, které se využívá pro paralelní programování se sdílenou pamětí. Dále si ve zkratce představíme knihovnu TNL, do které se implementace prováděly a jejíchž součástí se výsledné algoritmy stanou. A nakonec představíme Message Passing Iterface jenž se používá pro programování s distribuovanou pamětí.

1.1 CUDA

Společnost nVidia v současné době vyrábí jedny z nejlepších grafických karet na světě. Grafické karty obecně se vyznačují hlavně velkým počtem jader (též CUDA jader), jejichž počet se na každém modelu liší. Počet jader je u nejlepších karet větší než 5000. Jedno CUDA jádro sice není tak výkonné jako jedno jádro klasických procesorů CPU, ale kvůli jejich počtu je celkový výkon GPU mnohonásobně vyšší.

Grafické karty od nVidia se vyznačují velkým množstvím výpočetních jednotek (ALU - arithmetic logic unit), které jsou uspořádány do SM jednotek (Streaming Multiprocessor). Například grafická karta P100, na které se budou výpočty provádět, obsahuje 28 SM jednotek, kde každá má 128 CUDA jader. Tedy celkem obsahuje 3584 CUDA jader. Každý multiprocesor je navržen tak, aby na něm byly spouštěny stovky paralelních vláken. Tato vlákna jsou obecně řazena do warpů, kde jeden warp obsahuje 32 synchronních vláken. Na multiprocesoru se poté spouští jednotlivé warpy, jejichž vlákna jsou připravena na výpočet (mají načtena data z paměti). Z toho důvodu je nutné program spouštět s větším množstvím vláken než je CUDA jader dané grafické karty. Tímto způsobem se využívá maximální výpočetní výkon multiprocesoru a GPU tímto snižuje efekt latencí pamětí.

CUDA toolkit, neboli Compute Unified Device Architecture dále jen CUDA, je nástroj umožňující paralelní programovaní na grafických akcelerátorech společnosti nVidia. Jak je uvedeno v [3, 4], v CUDA je možno programovat pomocí jazyků C/C++, Fortranu a dalších.

CUDA rozšiřuje programovací jazyk C o kernelové funkce. Z obecné definice kernelové funkce ,která je znázorněna níže, je vidět, že se pro definici používá identifikátor `__global__`. Zároveň je si můžeme všimnout, že z kernelové funkce není možné vracet hodnoty. Jedinou možností jak dostat hodnoty z kernelové funkce je pomocí parametrů.

```
__global__ void KernelName( type parameter1, ... )
{
int i = threadIdx.x;
}
```

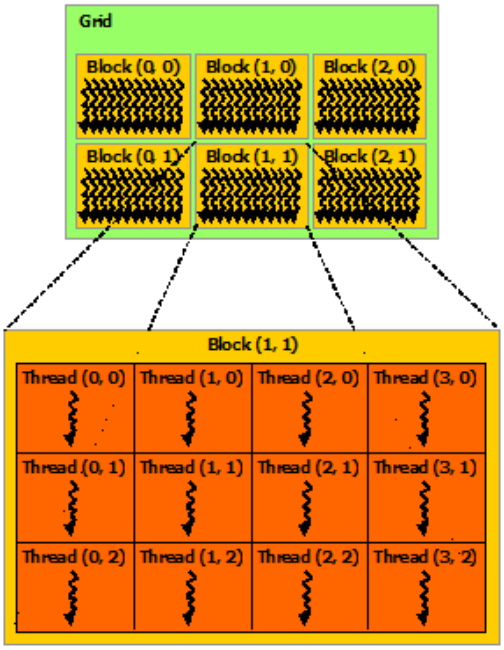
Volání kernelové funkce poté vypadá následovně.

```
KernelName<<< gridSize, blokSize, sharedMemory >>>( type parameter1, ...);
```

Kernelové funkce se spouštějí paralelně na GPU a to s počtem vláken, který si zvolíme pomocí proměnných `blockSize` a `gridSize`. Proměnná `blockSize` nám určuje počet vláken v jednom bloku, viz obrázek 1.1 žluté pole. Jak již bylo řečeno, jakmile jsou data pro všech 32 vláken jednoho warpu připravena ke zpracování prochází kódem synchronně. Jeden blok může být až tří-dimenzionální, což může ulehčit přepočítávání indexů u vícerozměrných úloh. Počet bloků v jednom gridu poté určí proměnná `gridSize`, viz obrázek 1.1 zelené pole. Grid může být opět jedno-dimenzionální až tří-dimenzionální, stejně jako tomu bylo u bloků.

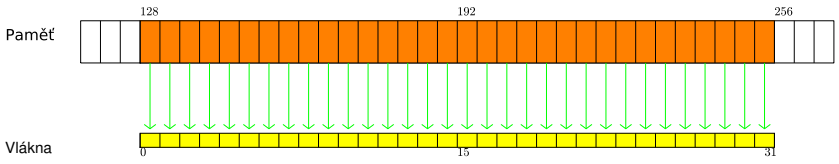
Důležitou součástí programování na GPU je možnost ukládání proměnných do různých pamětí grafické karty. Grafické karty od nVidia obsahují obvykle tři druhy pamětí. Prvním jsou registry, které jsou připojeny přímo ke CUDA jádrům a ukládají se do nich lokální proměnné z kernelových funkcí. Tyto proměnné má každé vlákno svoje tzn. nejsou sdílené. Druhou pamětí je sdílená paměť. V této paměti se musí předem alokovat místo pomocí speciálního parametru při volání kernelové funkce. Parametr `sharedMemory` obsahuje velikost místa v bytech, které chceme přidělit jednomu bloku. Uvnitř kernelové funkce se k této paměti poté přistupuje pomocí identifikátoru `__shared__`. Sdílená paměť není velká, proto by se do této paměti měly ukládat pouze nejpoužívanější sdílené proměnné. Tato paměť funguje podobně jako L1 cache na CPU, je stejně rychlá jako registry. Posledním možným úložištěm na GPU je globální paměť, která má obvykle velikost několika GB. Všechny ostatní proměnné, které se do kernelové funkce předávají pomocí parametrů musí být v globální paměti. Pro alokaci místa v globální paměti grafické karty slouží funkce `cudaMalloc()` a k uvolnění této paměti funkce `cudaFree()`. Do této paměti je často potřeba kopírovat data z paměti CPU, k tomu slouží funkce `cudaMemcpy()`. Tato funkce slouží i ke kopírování dat v opačném směru, tedy z GPU na CPU. Všechny tyto funkce jsou podrobně popsány v [4].

Více se o programování pomocí CUDA můžete dočíst v [3, 4, 5].

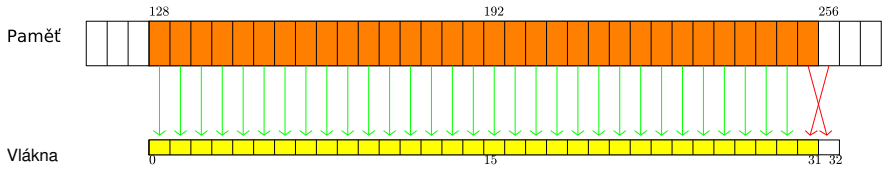


Obr. 1.1: Grid, bloky a vlákna, převzato z [4].

Při programování na GPU je nutné myslet i na podmínky spojené s přístupy do paměti. Nejen, že každá paměť je jinak rychlá a je tedy nutné rozmyslet, které proměnné kde uložit, ale i v jakém pořadí se bude do dané paměti přistupovat. Globální paměť je dobře navržena pro sekvenční přístup po 128-bytech na warp. Této podmínce se také říká podmínka sloučených přístupů do paměti. Je tedy důležité aby 32 vláken jednoho warpu přistupovalo do paměti o 128-bytech najednou a jednotlivá vlákna nesahala do různých částí paměti. Na obrázku 1.2 je znázorněno jakým způsobem by měla vlákna jednoho warpu přistupovat do paměti aby splnila podmínku sloučených přístupů, zatímco na obrázku 1.3 je znázorněno nenaplnění této podmínky, kde prvních 31 vláken warpu je nuceno čekat dokud 32. vlákno nenačte data z dalšího bloku paměti.



Obr. 1.2: Naplnění podmínky na sloučené přístupy do globální paměti GPU.



Obr. 1.3: Nenaplnění podmínky pro sloučené přístupy do globální paměti GPU.

1.2 OpenMP

OpenMP je API (Application Programming Interface), které umožňuje vytvářet paralelní programy v C, C++ a Fortranu, určené pro architektury se sdílenou pamětí. OpenMP je založeno na direktivách, které umožňují jednoduchou tvorbu paralelního kódu. Nejnovější standard verze 5.0 pro C, C++ i Fortran byl vydán roku 2018.

OpenMP umožňuje pomocí jednoduchých direktiv spouštět paralelně části kódu či for-cyklus. V této práci se využije právě paralelní for-cyklus, kde každé vlákno provede část iterací a to v pořadí, v jakém jsou vytíženy. Příklad volání takového for-cyklu je znázorněn níže.

```
#pragma omp parallel for
for( int i = 0; i < 10; i++ )
{
...
}
```

Více o direktivách můžete nalézt v [1].

1.3 TNL

Template Numerical Library, dále jen TNL, je numerická knihovna, která je vyvíjena pod vedením doktora T. Oberhubera na Katedře matematiky Fakulty jaderné a fyzikálně inženýrské Českého vysokého učení technického v Praze. V TNL jsou implementovány nejrůznější řešiče a datové struktury s nimiž se jednoduše pracuje v rámci pamětí. TNL umožňuje jednoduché vytváření těchto datových struktur v rámci paměti CPU i GPU a dále s nimi umožňuje pracovat nezávisle na paměti, ve které jsou uloženy. Tato vlastnost umožňuje jednoduchou přípravu dat pro výpočet. Jak bylo řečeno v předchozí části, některé z těchto datových struktur si zde blíže popíšeme, viz [2].

Zásadní výhodou kontejnerů implementovaných v TNL je automatické alokování paměti, možnost automatické realokace a snadné kopírování mezi CPU a GPU. Obě základní struktury, které zde budeme popisovat, mají tuto funkci implementovanou.

Základním kontejnerem v TNL je pole, neboli Array. Jelikož se jedná o šablonovou knihovnu, tak i tento základní kontejner je implementován následující šablonou:

`Array< Element, Device, Index, Allocator >`. Zde `Element` značí datový typ uložený v poli, `Device` může nabývat hodnot `Host` pro CPU nebo `Cuda` pro GPU a udává zařízení, na kterém se budou operace s polem provádět. Dále `Index` značí datový typ indexování a `Allocator` značí typ alokace a dealokace v paměti. Tento parametr se běžně volí `Default`, což značí typ alokace odpovídající parametru `Device`. Kontejner Array má metody typu `setSize()`,`getElement()`, `setElement()`, funkci `swap()` pro záměnu dvou polí a další. Zároveň jsou pro kontejner Array přetypovány operátory indexování, přiřazení, porovnání a další, což značně ulehčuje práci s touto základní strukturou.

Druhým základním kontejnerem je `Vector< Real, Device, Index, Allocator >`. Tento kontejner je nadstavbou Array. Přidává vektorové násobení a sčítání. Jedná se o kontejner, který reprezentuje vektory v matematice.

V této práci se omezíme výhradně na tyto dvě datové struktury i přes možnost využití dalších základních struktur v TNL implementovaných. Více se o TNL můžete dočíst v [2].

1.4 MPI

MPI, neboli Message Passing Interface je knihovna, která byla vytvořena v roce 1994. V současné době je MPI standard pro Message Passing model v paralelním programování pro distribuované architektury. Message Passing model je model, který umožňuje posílání zpráv mezi jednotlivými procesy paralelního programu. MPI tedy umožňuje vytvářet paralelní algoritmy pro vícejádrové procesory, více procesorové počítače nebo celé výpočetní klastry. Paralelizace pomocí MPI je možná i pro více grafických akcelerátorů, kterými jsou výpočetní klastry vybaveny. V MPI je možné programovat pomocí jazyků C/C++ a Fortranu.

Pomocí základních funkcí je možné rozesílat data mezi všechny či jen některé procesy daného programu. Tímto způsobem je docílena komunikace mezi jednotlivými procesy. Některé ze základních funkcí si nyní ukážeme.

Důležitou funkcí je `MPI_Comm_size( MPI_Comm communicator, int* size)`, která ukládá počet spuštěných procesů v komunikátoru `communicator` do proměnné `size`. `MPI_Comm` je označení pro komunikátor tedy skupinu procesů, kterou si sami na začátku zvolíme. Tato proměnná zjednodušuje posílání informace a příjem informací v rámci skupin procesů. K identifikaci jednotlivých procesů slouží funkce `MPI_Comm_rank(MPI_Comm communicator,int* rank)`, která do proměnné `rank` ukládá číslo procesu v daném komunikátoru. Tato identifikační čísla začínají 0 a zvětšují se do `size` − 1. Jednotlivé procesy mohou mít i vlastní jména. O tomto přístupu se můžete dočíst více v [7].

Funkce

```
MPI_Send( void* data, int count, MPI_Datatype datatype, int destination,
          int tag, MPI_Comm communicator)
```

a

```
MPI_Recv( void* data, int count, MPI_Datatype datatype, int source,
          int tag, MPI_Comm communicator, MPI_Status* status)
```

jsou hlavním nástrojem pro posílání zpráv z jednoho procesu do druhého. Obě funkce obsahují parametry `data`, jako ukazatel na data, `count`, počet prvků v datech, `datatype`,

jako typ dat který může být `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE` a další, `tag`, což je další volný parametr typu int a `communicator`, je, jak už bylo řečeno, skupina procesů. `MPI_Send` má navíc parametr `destination`, který určuje `rank` proces, kam se informace posílá. `MPI_Recv` má obdobný parametr `source`, což je `rank` odesílajícího procesu a navíc `status`, který slouží k ovládání programu.

Další funkce, které z MPI využijeme v kapitole o Gaussově Eliminační metodě, jsou

```
MPI_Bcast( void* data, int count, MPI_Datatype datatype, int root,
        MPI_Comm  communicator)
```

a

```
MPI_Allgather( void* send_data, int send_count, MPI_Datatype send_datatype,
        void* recv_data, int recv_count, MPI_Datatype recv_datatype,
        MPI_Comm communicator, MPI_Status* status).
```

Funkce `MPI_Bcast` rozesílá informace obsažené v `data` z procesu `root` do všech ostatních procesů v dané skupině procesů `communicator`. Druhou funkcí je `MPI_Allgather`, která vezme z každého procesu data `send_data`. Tato data následně urovná do jednoho velkého bloku v pořadí odpovídající číslu `rank` dané skupiny procesů v `communicator` a následně tento velký blok rozešle všem procesům zpět. MPI nabízí i další funkce, o kterých je možné dozvědět se v [7].

Kapitola 2

Násobení řídké symetrické matice s vektorem

Násobení matice s vektorem, také SpMV (Sparse Matrix Vector multiplication), tj.

$$\mathbb{A} \cdot \boldsymbol{x} = \boldsymbol{b},$$

pro $\mathbb{A} \in \mathbb{R}^{n \times m}, \boldsymbol{x} \in \mathbb{R}^m$ a $\boldsymbol{b} \in \mathbb{R}^n$ je jedna ze základních operací, kterou provádíme s maticemi. Je důležitou součástí iterativních řešičů, které na této operaci tráví mnoho času. Proto je důležité udělat násobení matice s vektorem maximálně rychlé s minimálními požadavky na paměť, viz [10].

Matice, které je nutné s vektory násobit, jsou často řídké a navíc symetrické, z toho důvodu se v této kapitole zaměříme na formát pro řídké matice Ellpack a jeho symetrickou verzi. Ukážeme jakým způsobem probíhá násobení řídké symetrické matice s vektorem, dále jen SpMV, a jaké tato operace přináší výhody a nevýhody. Dále si ukážeme dva možné způsoby řešení těchto nevýhod v podobě atomických instrukcí a speciálního maticového obarvení. Nakonec porovnáme tato dvě řešení na testovacích maticích z [14].

2.1 Ellpack formát

Pro uložení matic do paměti existuje velké množství formátů. Cílem této práce je zkoumat symetrický Ellpack formát pro uložení řídké symetrické matice, tedy symetrické matice, která má málo nenulových prvků, a její násobení s vektorem. Nejprve tedy ukážeme způsob ukládání matice pomocí Ellpack formátu a následně jeho symetrickou verzi.

Ellpack je jeden z formátů pro ukládání a práci s řídkými maticemi, jak je popsáno v [9] a [10]. V Ellpack formátu se matice $n \times m$ s maximálně K nenulovými prvky na řádku ukládá do vektoru `data` o velikosti $n \cdot K$ s doplněním nul na konci řádků, které mají méně než K nenulových prvků. Analogickým způsobem se ukládají i indexy sloupců do vektoru `columns`, kde se jako zarovnávací znak pro uměle vytvořené nulové prvky (paddingIndex) používá -1. Příklad pro CPU je zobrazen v rovnici (2.1).

$$\begin{aligned} \mathbb{A} &= \begin{pmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{pmatrix}, \\ K &= 3, \\ \texttt{data} &= [1\ 7\ 0\ 2\ 8\ 0\ 5\ 3\ 9\ 6\ 4\ 0], \\ \texttt{columns} &= [0\ 1\ -1\ 1\ 2\ -1\ 0\ 2\ 3\ 1\ 3\ -1]. \end{aligned} \tag{2.1}$$

Paměť na CPU s sebou nenese žádné dodatečné podmínky pro přístupy, zatímco na GPU se snažíme dosáhnout sloučených přístupů popsaných v kapitole o CUDA. Z toho důvodu se uložení ve formátu Ellpack liší pro CPU a GPU.

Je dobré zmínit, že přístup k prvkům matice pomocí dotázání na i-tý řádek a j-tý sloupec znamená nejprve prohledávání vektoru `columns` a zjišťování zda tento prvek je v matici nenulový a poté hledání prvku ve vektoru `data`. Při SpMV ale není nutné počítat s nulovými prvky a proto bude stačit procházet vektory postupně a připočítávat přírůstky k výslednému vektoru.

SpMV pro Ellpack matici na CPU v paralelní formě pomocí OpenMP je znázorněno v kódu 1. V tomto kódu se paralelně prochází všechny řádky a sekvenčně se pro každý řádek matice $\mathbb{A}$ počítá výsledek do vektoru $\boldsymbol{b}$.

Algoritmus 1 Funkce pro SpMV s Ellpack formátem na CPU.

```
1:  void EllpackSpMV( const Ellpack* A,
2:                    const Real* x,
3:                    Real* b)
4:  #pragma omp parallel for
5:  for Index row = 0; row < n; row ++ do
6:      IndexType i = row*A.K
7:      IndexType rowEnd = (row+1)*A.K
8:      IndexType columnIndex
9:      while i < rowEnd ∧ ( columnIndex = A.columns[ i ] ) ≠ -1 do
10:         b[ row ] += A.data[ i ] *x[ columnIndex ]
11:         i += 1
```

Pro správné uložení matice $\mathbb{A}$ na GPU je nutné rozmyslet v jakém pořadí se bude k prvkům matice $\mathbb{A}$ přistupovat. Je zřejmé, že vhodným mapováním bude jedno CUDA vlákno na jeden řádek matice $\mathbb{A}$. Prvních 32 vláken tedy přistoupí k prvním prvkům na prvních 32 řádcích matice $\mathbb{A}$. Z toho důvodu je vhodné matici na GPU uložit způsobem splňujícím podmínku pro sloučené přístupy do paměti. Je tedy nutné ukládat nejprve první nenulové prvky pro všechny řádky, následně druhé nenulové prvky pro všechny řádky až do K, což je maximální počet nenulových prvků na řádku. Zároveň je nutné zarovnat počet řádků na násobek 32, což je velikost jednoho warpu popsaného v kapitole o CUDA, z důvodu podmínky sloučených přístupů. Toto číslo označíme jako $\tilde{n} = (\lfloor n/32 \rfloor + 1) * 32$, kde n značí počet řádků matice

$\mathbb{A}$. Toto číslo $\tilde{n}$ také určuje vzdálenost následujícího prvku na řádku matice $\mathbb{A}$ ve vektoru `data`. Ukázka uložení matice ve formátu Ellpack je znázorněna v rovnici (2.2). Zde bylo, jako zarovnávací znak ve vektoru `columns` pro řádky, které mají méně nenulových prvků než ostatní řádky, použito číslo -1. Tímto číslem se také doplní nulové řádky, které jsme do paměti uměle přidali z důvodu podmínky na sloučené přístupy. Vektory `data` i `columns` mají nyní velikost $K \cdot \tilde{n}$, kde K je opět maximální počet nenulových prvků na jednom řádku. Pro tak malý příklad jako je (2.2) se zdá zarovnání paměti na násobek 32 jako zbytečná alokace paměti, tento faktor je ale u matic s miliónovým počtem řádků zanedbatelný.

$$
\begin{aligned}
\mathbb{A} = & \begin{pmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{pmatrix}, \\
\tilde{n} = & \ 32, \\
K = & \ 3, \\
\texttt{data} = [& \ 1,\ 2,\ 5,\ 6,\ 0, \ldots, 0, \\
& \ 7,\ 8,\ 3,\ 4,\ 0, \ldots, 0, \\
& \ 0,\ 0,\ 9,\ 0,\ 0, \ldots, 0\ ], \\
\texttt{columns} = [& \ 0,\ 1,\ 0,\ 1,\ -1, \ldots, -1, \\
& \ 1,\ 2,\ 2,\ 3,\ -1, \ldots, -1, \\
& -1,\ -1,\ 3,\ -1,\ -1, \ldots, -1\ ].
\end{aligned}
\tag{2.2}
$$

Kernelová funkce pro SpMV pomocí Ellpack formátu na GPU je znázorněn v kódu 2. Tato kernelová funkce se spouští s počtem vláken rovným počtu řádků matice $\mathbb{A}$ a jedno CUDA vlákno počítá s jedním řádkém této matice.

Algoritmus 2 Kernelová funkce pro SpMV s Ellpack formátem.

```
1: ___global___
2: void EllpackSpMVKernel( const Ellpack* 𝔸,
3:                         const Real* x,
4:                         Real* b)
5:
6: const IndexType rowIdx = blockIdx.x * blockDim.x + threadIdx.x
7: if rowIdx >= n then
8:     return
9: IndexType i = rowIdx
10: IndexType el( 0 )
11: Real result( 0.0 )
12: IndexType columnIndex
13: while el++ < 𝔸.K ∧
14:       (columnIndex = 𝔸.data[ i ] ) < m ∧
15:        columnIndex ≠ −1 do
16:     result += 𝔸.data[ i ] *x[ columnIndex ]
17:     i += 𝔸.ñ                                  ▷ i = i+(⌊numColumns/32⌋ + 1) · 32
18: b[ rowIdx ] = result
```

2.1.1 Symetrický Ellpack formát

Symetrický Ellpack formát pro matici $\mathbb{A} \in \mathbb{R}^{n \times n}$ je analogický ke klasickému Ellpack formátu, jedinou změnou je ukládání pouze pod-diagonálních a diagonálních prvků. Pro CPU má vektor `data` velikost $n \cdot K$, kde K je maximální počet nenulových pod-diagonálních a diagonálních prvků na jednom řádku. Příklad uložení matice $\mathbb{A}$ do symetrického Ellpack formátu na CPU je znázorněn v rovnici (2.3).

$$\begin{aligned} \mathbb{A} &= \begin{pmatrix} 1 & 7 & 0 & 0 \\ 7 & 2 & 8 & 6 \\ 0 & 8 & 3 & 9 \\ 0 & 6 & 9 & 0 \end{pmatrix}, \\ K &= 2, \\ \texttt{data} &= [1\ 0\ 7\ 2\ 8\ 3\ 6\ 9], \\ \texttt{columns} &= [0\ -1\ 0\ 1\ 1\ 2\ 1\ 2]. \end{aligned} \tag{2.3}$$

Uložením pouze pod-diagonálních a diagonálních prvků jsme sice snížili paměťové nároky, ale při výpočtu se musí počítat s pod-diagonálními prvky i jako s nad-diagonálními. V rovnici 2.4 je znázorněn i-tý řádek výsledku násobení matice $\mathbb{A}$ s vektorem $\boldsymbol{x}$. Zde tečkovaná část odpovídá pod-diagonálním a diagonálním prvkům, ke kterým se v symetrickém Ellpack formátu přistupuje přímo, a podtržená část odpovídá nad-diagonálním prvkům matice $\mathbb{A}$, k nimž se přistupuje jako k prvkům transponovaným, tedy pod-diagonálním.

$$(\mathbb{A} * \boldsymbol{x})_i = \sum_{j=0}^{m-1} \mathbb{A}_{i,j} * \boldsymbol{x}_j = \sum_{j=0}^{i} \mathbb{A}_{i,j} * \boldsymbol{x}_j + \sum_{j=i+1}^{m-1} \mathbb{A}_{i,j} * \boldsymbol{x}_j = \\ = \sum_{j=0}^{i} \mathbb{A}_{i,j} * \boldsymbol{x}_j + \sum_{j=i+1}^{m-1} \mathbb{A}^T_{j,i} * \boldsymbol{x}_j = \underbrace{\sum_{j=0}^{i} \mathbb{A}_{i,j} * \boldsymbol{x}_j}_{\cdots\cdots\cdots\cdots} + \underline{\sum_{j=i+1}^{m-1} \mathbb{A}_{j,i} * \boldsymbol{x}_j} \tag{2.4}$$

Jednoduchý pseudokód 3 znázorňuje počítání SpMV za pomoci pouze pod-diagonálních a diagonální prvků matice $\mathbb{A}$, které jsou v symetrickém Ellpack formátu uloženy. Řádky 4 a 5 tedy reprezentují přírůstky nad-diagonálních prvků matice $\mathbb{A}$ k výslednému vektoru $\boldsymbol{b}$.

Algoritmus 3 SpMV se symetrickým Ellpack formátem pomocí pod-diagonálních a diagonálních prvků.

```
1: for i = 0; i < n; i ++ do
2:     for j = 0; j ≤ i; j++ do
3:         b_i += A_{i,j} * x_j
4:         if j < i then
5:             b_j += A_{i,j} * x_i
```

Sekvenční kód pro symetrický Ellpack formát je znázorněn v algoritmu 4. Paralelní verze tohoto kódu za použití `#pragma omp parallel for` by bez dalších úprav fungovat nemohla z důvodu konfliktů v paměti výstupního vektoru. Doposud jedno vlákno počítalo vždy přírůstky pouze k jednomu řádku výstupního vektoru. V symetrické verzi Ellpack formátu musí jednotlivá vlákna přistupovat do různých řádků výstupního vektoru z nutnosti počítání nad-diagonálních prvků a tak by se mohlo stát, že do výstupního vektoru bude zapisovat více než jedno vlákno najednou a tím by mohl vzniknout nedefinovaný výraz. Tímto problémem se budeme zabývat v následující kapitole o atomických instrukcích a v kapitole o speciálním maticovém obarvení.

Algoritmus 4 SpMV se symetrickým Ellpack formátem na CPU.

```
1: void EllpackSymmetricSpMVFunction( const EllpackSymmetric* A,
2:                                    const Real* x,
3:                                    Real* b)
4: for IndexType row = 0; row < A.n; row ++ do
5:     IndexType i = row*A.K
6:     const IndexType rowEnd = (row+1)*A.K
7:     IndexType colIdx
8:     while i < rowEnd ∧ ( colIdx = A.columns[i] ) ≠ -1 do
9:         b[row] += A.data[i]*x[colIdx]          ▷ pod-diagonální a diagonální prvky
10:        if row ≠ colIdx then
11:            b[colIdx] += A.data[i]*x[row]      ▷ nad-diagonální prvky
12:        i += 1
```

Ekvivalentním způsobem jako klasický Ellpack na GPU se ukládá i symetrický Ellpack na GPU. Je opět nutné nejprve zvolit mapování vláken při násobení matice s vektorem v

CUDA kódu, abychom následně mohli rozmyslet, v jakém pořadí prvky ukládat. Je zřejmé, že mapování jednoho CUDA vlákna na jeden řádek matice $\mathbb{A}$ bude opět fungovat pokud při počítání s pod-diagonálními prvky budeme zároveň počítat i s jejich transpozicí, tedy s nad-diagonálními prvky, a tudíž je nebudeme z paměti číst dvakrát. Tento způsob je jistě jednodušší než přistupovat k nad-diagonálním prvkům zvlášť. Pokud bychom totiž chtěli přistupovat k nad-diagonálním prvkům, je nutné prohledávat celé vektory `data` a `columns`, protože přímý přístup k prvku na pozici (i,j) je ve formátu Ellpack pomalý. Příklad uložení matice $\mathbb{A}$ na GPU je znázorněn v rovnici (2.5).

$$\begin{aligned}
\mathbb{A} = &\quad \begin{pmatrix} 1 & 7 & 0 & 0 \\ 7 & 2 & 8 & 6 \\ 0 & 8 & 3 & 9 \\ 0 & 6 & 9 & 0 \end{pmatrix}, \\
\tilde{n} = &\quad 32, \\
K = &\quad 2, \\
\texttt{data} = [&\quad 1,\ 7,\ 8,\ 6,\ 0,\ldots,0, \\
&\quad 0,\ 2,\ 3,\ 9,\ 0,\ldots,0\ ], \\
\texttt{columns} = [&\quad 0,\ 0,\ 1,\ 1,\ -1,\ldots,-1, \\
&\quad 32,\ 1,\ 2,\ 2,\ -1,\ldots,-1\ ].
\end{aligned} \tag{2.5}$$

Velikost vektorů `data` a `columns` je opět $K \cdot \tilde{n}$, kde K opět označuje maximální počet prvků na jednom řádku matice $\mathbb{A}$ pod diagonálou včetně diagonály a $\tilde{n} = (\lfloor n/32 \rfloor + 1) * 32$, kde n opět značí počet řádků matice $\mathbb{A}$.

Nefunkční kernelová funkce pro výpočet na GPU je znázorněna v algoritmu 5. Tento kód opět nemůže fungovat z důvodu konfliktů v paměti popsaných výše. Daný kód budeme upravovat v následujících kapitole o atomických instrukcích a kapitole o speciálním maticovém obarvení.

Algoritmus 5 Kernelová funkce pro SpMV se symetrickým Ellpack formátem na GPU.

```
1: ___global___
2: void EllpackSymmetricSpMVKernel( const EllpackSymmetric* 𝔸,
3:                                  const Real* x,
4:                                  Real* b)
5: const IndexType rowIdx = blockIdx.x * blockDim.x + threadIdx.x
6: if rowIdx >= 𝔸.n then
7:     return
8: IndexType i = rowIdx
9: IndexType el = 0
10: IndexType colIdx
11: while el++ < 𝔸.K ∧
12:       ( colIdx = 𝔸.columns[i] ) < 𝔸.m ∧
13:       colIdx ≠ 𝔸.ñ do
14:     b[rowIdx] += 𝔸.data[i]*x[colIdx]          ▷ pod-diagonální a diagonální prvky
15:     if rowIdx ≠ colIdx then
16:         b[colIdx] += 𝔸.data[i]*x[rowIdx]          ▷ nad-diagonální prvky
17:     i += 𝔸.ñ
```

2.2 Paralelní SpMV

Jak bylo řečeno v minulé části práce, při paralelním násobení matice $\mathbb{A}$ s vektorem $\boldsymbol{x}$ budeme mapovat jedno vlákno na jeden řádek matice $\mathbb{A}$. Při tomto přístupu k prvkům za podmínky symetrického uložení matice ve formátu Ellpack se musí každý pod-diagonální prvek také transponovat a přispět na patřičné místo výstupního vektoru $\boldsymbol{b}$. Každé vlákno tak pracuje s několika řádky výstupního vektoru $\boldsymbol{b}$. Může se tak stát, že na jedno místo výstupního vektoru bude zapisovat více než jedno vlákno najednou a tak mohou vznikat konflikty v paměti. Tyto konflikty poté vedou k nedefinovaným výrazům a proto je nutné konflikty odstranit. V následujících sekcích ukážeme dvě možná řešení tohoto problému a to pomocí atomických instrukcí a speciálního maticového obarvení.

2.2.1 SpMV s atomickými instrukcemi

Atomické instrukce zabraňují konfliktům v paměti následujícím způsobem. Pokud nějaké vlákno na dané místo již zapisuje, potom ostatní vlákna mají zakázaný přístup do dané paměti a musí nečinně čekat. Atomické instrukce jsou implementovány jak v OpenMP tak v CUDA. Obě funkce jsou znázorněny níže v algoritmech 6, 7. Výhodou atomických instrukcí je jednoduchá implementace. Zřejmou nevýhodou je pak nutnost zastavení výpočtu v některých vláknech paralelního kódu, dokud ostatní vlákna neukončí svůj přístup do paměti. Samotné konflikty v paměti ale také zpomalují kód.

Paralelní verze kódu pro CPU, ve kde se využije paralelního programování pomocí OpenMP a jednoduchého paralelního for-cyklu, je znázorněna v algoritmu 6. Atomické instrukce v takovém případě mají tvar pouhého volání `#pragma omp atomic`.

Algoritmus 6 Funkce pro SpMV se symetrickým Ellpack formátem a atomickými instrukcemi na CPU.

```
1: void EllpackSymmetricSpMVFunction( const EllpackSymmetric* 𝔸,
2:                                    const Real* x,
3:                                    Real* b)
4: #pragma omp parallel for
5: for IndexType row = 0; row < 𝔸.n; row ++ do
6:     IndexType i = row∗𝔸.K
7:     const IndexType rowEnd = (row+1)∗𝔸.K
8:     IndexType colIdx
9:     while i < rowEnd ∧ ( colIdx = 𝔸.columns[i] ) ≠ -1 do
10:        #pragma omp atomic
11:        b[row] += 𝔸.data[i]∗x[colIdx]          ▷ pod-diagonální a diagonální prvky
12:        if row ≠ colIdx then
13:            #pragma omp atomic
14:            b[colIdx] += 𝔸.data[i]∗x[row]          ▷ nad-diagonální prvky
15:        i += 1
```

Paralelní verze kódu pro GPU je znázorněna v algoritmu 7. Globální funkce z algoritmu 7 se spouští s počtem vláken rovným $\tilde{n}$, což je počet řádků n matice $\mathbb{A}$ zarovnaný na násobek 32. Následně každé vlákno přistupuje do všech pod-diagonálních prvků a diagonálního prvku svého řádku a atomicky přičítá jejich hodnotu s násobkem vstupního vektoru $\boldsymbol{x}$ do výsledného vektoru $\boldsymbol{b}$. S pod-diagonálními prvky se opět počítá i jako s nad-diagonálními prvky, ale konflikty jsou již vyřešeny pomocí atomických instrukcí.

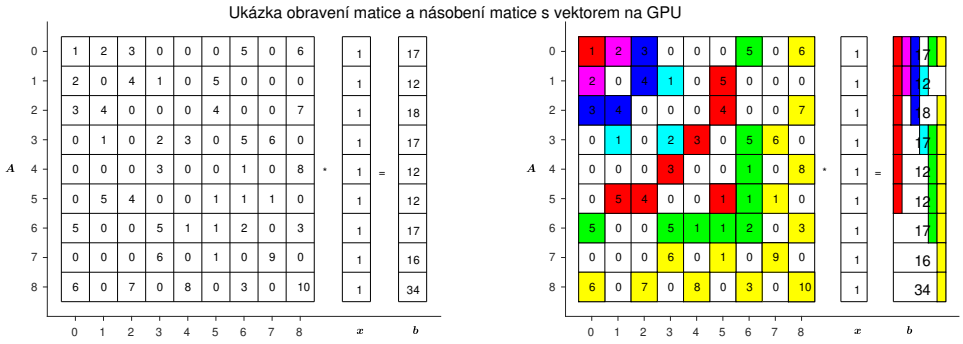
Algoritmus 7 Kernelová funkce pro SpMV se symetrickým Ellpack formátem a atomickými instrukcemi na GPU.

```
1: ___global___
2: void EllpackSymmetricSpMVKernel( const EllpackSymmetric* 𝔸,
3:                                  const Real* x,
4:                                  Real* b)
5: const IndexType rowIdx = blockIdx.x ∗ blockDim.x + threadIdx.x
6: if rowIdx >= 𝔸.n then
7:     return
8: IndexType i = rowIdx
9: IndexType el = 0
10: IndexType colIdx
11: while el++ < 𝔸.K ∧
12:       ( colIdx = 𝔸.columns[i] ) < 𝔸.m ∧
13:       colIdx ≠ 𝔸.ñ do
14:     atomicAdd( b[rowIdx], 𝔸.data[i]∗x[colIdx] )     ▷ pod-diagonální a diagonální prvky
15:     if rowIdx ≠ colIdx then
16:         atomicAdd( b[colIdx], 𝔸.data[i]∗x[rowIdx] )          ▷ nad-diagonální prvky
17:     i += 𝔸.ñ
```

2.2.2 SpMV s maticovým obarvením

Druhým přístupem, který vyzkoušíme k odstranění konfliktů v paměti výsledného vektoru $\boldsymbol{b}$ je speciální maticové obarvení. Jelikož jsou konflikty v paměti způsobeny paralelními přístupy do stejného místa v paměti a pouze zápis do tohoto místa z více vláken najednou způsobuje nedefinovaný výsledek, je na místě zkusit tyto přístupy udělat sekvenčně. Tím bychom odstranili konflikty v paměti výsledného vektoru $\boldsymbol{b}$. Jak bylo řečeno již v kapitole o symetrickém Ellpack formátu, konflikty vznikají z důvodu počítání nad-diagonálních prvků. Existují-li skupiny řádků, které přispívají na různá místa výstupního vektoru, konflikty by nemusely vznikat. Tyto skupiny poté můžeme procházet jednu po druhé a tím se vyhneme všem konfliktům v paměti. Důležitou podmínkou je, aby všechna vlákna zpracovávala v jeden okamžik pouze elementy ze stejné skupiny. Pokud by tomu tak nebylo, muselo by se vlákno v části výpočtu zastavit a počkat až přijde na řadu další skupina prvků matice $\mathbb{A}$, aby dané vlákno dokončilo svůj výpočet.

Příklad rozdělení do barevných skupin je na obrázku 2.1. Z obrázku si můžeme všimnout, že třeba 2. řádek, jehož pod-diagonální prvky a k nim odpovídající nad-diagonální prvky jsou obarveny modrou barvou, přispívá na tři řádky výstupního vektoru a to na řádky 0,1 a 2. Kdybychom chtěli obdobně modrou barvou obarvit další řádek, museli bychom najít takový řádek, který na tato místa výstupního vektoru nepřispívá. Takový řádek se v matici vyskytuje a je jím řádek 7. Řádek 7 samotný přispívá na místa 3,5 a 7 výstupního vektoru. Při obarvení tohoto řádku na modro by poté modrá barva přispívala na místa 0-3, 5 a 7 výstupního vektoru. Je tedy zřejmé, že rozdělení do skupin není jednoznačné.



Obrázek 2.1: Ukázka speciálního obarvení matice pro násobení na GPU.

K rozdělení matice do skupin můžeme využít speciální maticové obarvení, k jehož definici však nejprve zavedeme klasické grafové obarvení, ukážeme jak přejít od grafu k matici a zpět a nakonec dané obarvení upravíme pro naše účely.

Definice grafového obarvení je zobrazena v definici 1.

Definice 1. Nechť $G = (V, E)$ je graf, $k \in \mathbb{N}$. k-vrcholovým obarvením grafu G nazveme zobrazení $\varphi : V \to \hat{k}$. φ se nazývá vlastní k-vrcholové obarvení grafu G, jestliže platí

$$(\forall u, v \in V)(\varphi(u) = \varphi(v) \implies \{u, v\} \notin E)$$

tj. jestliže stejně barvené vrcholy nejsou spojené hranou.

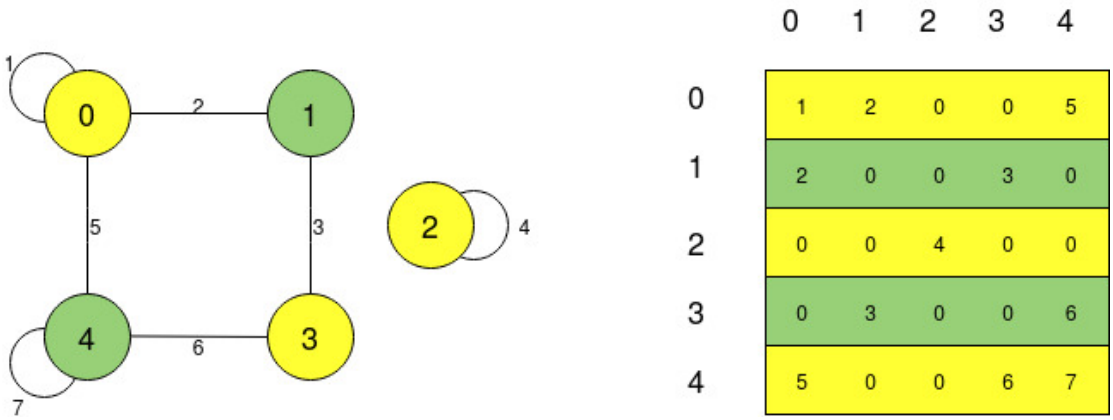
Každý neváženy neorientovaný graf lze poté uložit ve tvaru adjacenční matice, jež je definovaná v definici 2. Tuto definici lze rozšířit pro vážený orientovaný graf jednoduchým přepisem jedniček na váhy obsažené v jednotlivých hranách s doplněním nemožné váhy hrany (př. $+\infty$) tam, kde žádná hrana nevede.

Definice 2. Buď $G = (V, E)$ graf, n = $\#V$. Adjacenční maticí grafu G rozumíme matici $\mathcal{A}_G \in \{0,1\}^{n,n}$, pro jejíž prvky platí

$$(\mathcal{A}_G)_{i,j} = \begin{cases} 1 & pro\ \{v_i, v_j\} \in E \\ 0 & jinak \end{cases}$$

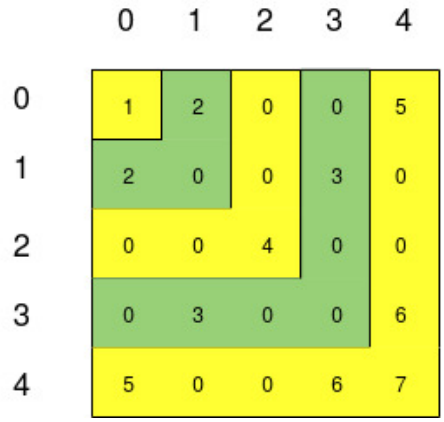
Tedy přechod od symetrické matice přes adjacenční matici ke grafu a jeho obarvení není nijak složitá záležitost. Bohužel klasické grafové obarvení, které má bohatý teoretický základ, se v naší úloze neuplatní. Nicméně je možné definici grafového obarvení upravit tak, aby rozdělovala matici do barev, se kterými bude možné provádět SpMV bez konfliktů. Tato úprava i postup k jejímu odvození je popsána v následujících odstavcích.

V obrázku 2.2 je ukázáno klasické grafové obarvení a matice daného obarvení. Je zřejmé, že toto obarvení není pro výpočet SpMV použitelné, protože prvky pod diagonálou mohou mít jinou barvu než prvky nad diagonálou.



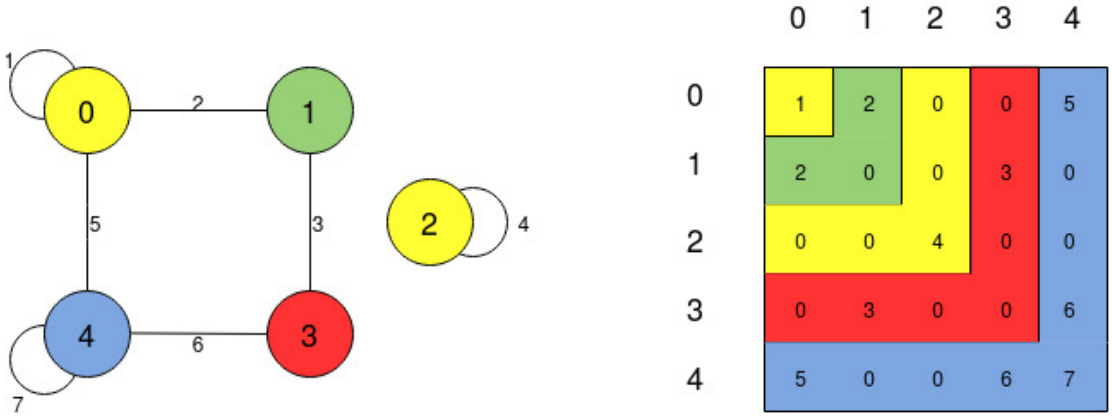
Obr. 2.2: Klasické obarvení matice

Kdybychom obarvovali pouze pod-diagonální prvky a odpovídajícím nad-diagonálním prvkům bychom přiřadili odpovídající barvu, vypadalo by obarvení jako na obrázku 2.3. Takové obarvení by již odpovídalo počítání SpMV, protože pro jeden řádek vlákno počítá s pod-diagonálními, diagonálním a nad-diagonálními prvky, jenž jsou nyní obarveny stejnou barvou. Při bližším prozkoumání nám ale toto obarvení nepomůže z důvodu konfliktů v paměti. V příkladu se totiž budou počítat řádky 0 a 4 (žlutá barva) najednou a v 0 řádku výstupního vektoru tak může opět vzniknout konflikt v paměti.



Obr. 2.3: Klasické obarvení matice s obarvením nad-diagonalních prvků podle pod-diagonálních.

Je tedy nutné obarvit tyto řádky různou barvou. Tomuto obarvení by odpovídalo grafové obarvení, kde nejen že žádné dva vrcholy stejné barvy nejsou spojeny hranou, ale navíc žádné dva vrcholy stejné barvy nemají společný sousední vrchol. Tzn. mezi vrcholy obarvenými stejnou barvou neexistují žádná cesta o délce dva jenž by je spojovala. (NEBO: neexistují žádné dvě hrany jenž by je spojovaly.) Takové obarvení je opět ukázáno na grafu na obrázku 2.4. Z příkladu je již vidět, že takovéto obarvení by bylo možné využít při paralelním výpočtu SpMV, ale počet barev je i na takto jednoduchém příkladě vysoký. Totiž 3. řádek by bylo možné obarvit žlutou barvou a stále by konflikty nevznikaly.



Obr. 2.4: Obarvení v závislosti na sousedech sousedů.

Pro vylepšení tohoto grafového obarvení musíme vzít v úvahu i směr hran. Jelikož jsou konflikty v paměti způsobeny nad-diagonálními prvky na jednom řádku, orientovaný graf by měl vycházet právě z nich. Díky symetričnosti matice ale můžeme vycházet z pod-diagonálních prvků matice a prvků v jednom sloupci. Vytvoříme-li graf pouze z těchto prvků s orientací od řádku ke sloupci, pak matice z našeho příkladu bude odpovídat grafu na obrázku 2.5. Lze si všimnout, že do 0 řádku výstupního vektoru budou přispívat řádky 0,1 a 4. Tyto řádky je tedy nutné obarvit různými barvami. 2. řádek nehraje v rámci obarvení roli, jelikož přistupuje pouze na druhou pozici výstupního vektoru, do které již žádný jiný řádek nepřispívá. Změna

ale nastává u řádku číslo 3, který přispívá do 1. řádku výstupního vektoru, takže nemůže být obarven stejnou barvou jako 1. řádek matice a zároveň do 3. řádku přispívá řádek 4. Tedy barva 3. a 4. řádku se musí lišit.

Výsledné pravidlo pro obarvení v takto vytvořeném orientovaném grafu je uvedeno v definici 3.

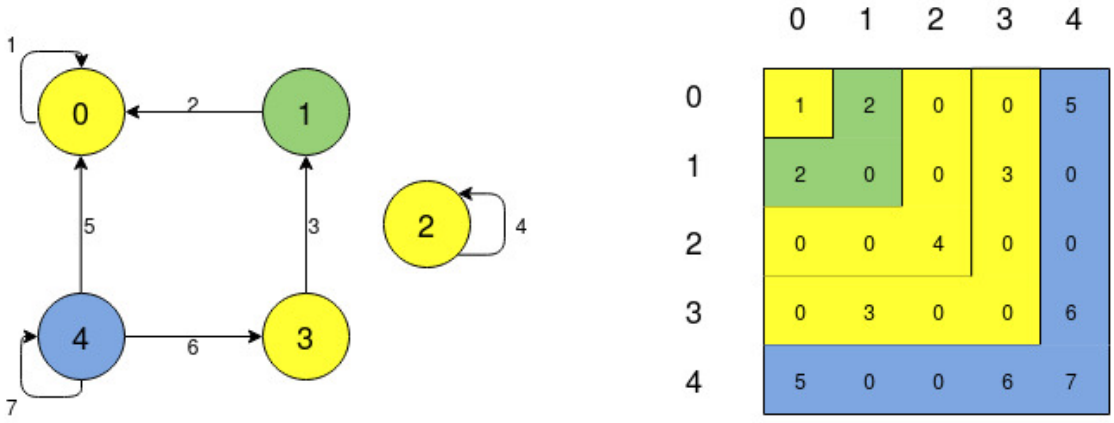
Definice 3. Buď $G = (V, E)$ orientovaný graf. k-vrcholovým obarvením grafu G nazveme zobrazení $\varphi : V \to \hat{k}$. φ nazveme speciální obarvení grafu, jestliže platí

$$(\forall u, v \in V)(\varphi(u) = \varphi(v) \implies \{u, v\} \notin E \wedge \{v, u\} \notin E)$$

a

$$(\forall u, v \in V)(\forall z \in V)(\varphi(u) = \varphi(v) \implies \{u, z\} \notin E \wedge \{v, z\} \notin E)$$

tj. stejně barevné vrcholy nejsou spojeny hranou nezávisle na orientaci hrany a ze stejně barvených vrcholů nevedou orientované hrany do jiného společného vrcholu.



Obr. 2.5: Speciální obarvení orientovaného grafu.

Rozdělení prvků matice $\mathbb{A}$ do skupin budeme nazývat speciální obarvení matice $\mathbb{A}$ a každé skupině prvků přiřadíme jednu barvu.

Existence obarvení matice popsaného výše je zřejmá, stačilo by totiž aby každé vlákno mělo jinou barvu. Tím by počet barev byl roven počtu řádků n matice $\mathbb{A}$ a výpočet by přešel na jednoduchý sekvenční kód z kapitoly o symetrickém Ellpack formátu. Tímto způsobem bychom ale nevyužili paralelní výkon ani klasického procesoru, natož pak grafické karty. Další podmínka je tedy zřejmá, je nutné vytvořit obarvení s minimálním počtem barev. Barvy se sice budou procházet sekvenčně, ale vlákna uvnitř barev budou moci počítat paralelně.

Příklad obarvení matice $\mathbb{A}$ je znázorněn na obrázku 2.1. Krom minimálního počtu barev použitých na obarvení je nutné snažit se pokud možno maximálně vyvážit počty řádků obarvených jednotlivými barvami. Tím se dosáhne alespoň podobné paralelizace pro všechny barvy, což je žádoucí při maximálním využití paralelizace.

Následující pseudo-algoritmus 8 ukazuje jakým způsobem lze matici obarvit tak, aby graf vytvořený z jejich pod-diagonálních a diagonálních prvků splňoval definici speciálního obarvení 3. Do tohoto algoritmu vstupuje matice $\mathbb{A}$, vektor `colorsVector`, o velikosti n do něhož

se budou ukládat barvy, a počet barev $\#colors$. Algoritmus se poté spouští dvakrát, kdy v prvním průchodu se udělá obarvení, které nebude příliš náhodné ale dostaneme dobrý odhad počtu barev $\#colors$ nutných k obarvení matice. Následný druhý průchod této funkce poté lépe znáhodní použití barev na řádky a tím dojde k vyvážení počtů řádků pro jednotlivé barvy.

Algoritmus 8 Způsob obarveni matice $\mathbb{A}$.

```
1:  function ColorizationSimple(A, colorsVector, #colors = 1)
2:      usedColors_{a,b}   a ∈ n − 1, b ∈ #colors − 1          ▷ S počátečními hodnotami 0
3:      for i = n − 1; i ≥ 1; i− do
4:          newColor_a   a ∈ #colors − 1                       ▷ S počátečními hodnotami 0
5:          for k = 0; k < #colors; k++ do
6:              newColor_k = usedColors_{i,k}
7:          for j = 0; j < n; j++ do
8:              if A_{i,j} ≠ 0 then
9:                  for k = 0; k < #colors; k++ do
10:                     newColor_k = |newColor_k - usedColors_{j,k}|
11:         count = Σ_{k=0}^{#colors−1} newColor_k
12:         if count = #colors then
13:             extendColorsArraysWithOne(#colors, newColor, usedColors)
14:         color = findRandomColor( newColor )
15:         colorsVector_i = color
16:         for j = 0; j < n; j++ do
17:             if A_{i,j} ≠ 0 then
18:                 usedColors_{j,color} = 1
19:         usedColors_{i,color} = 1
```

V samotném algoritmu se tedy nejprve inicializuje dvourozměrné pole usedColors samými nulami, do něhož se budou ukládat barvy, které jsou pro daný i-tý řádek již obsazené. Nulové hodnoty znamenají možnost použití dané barvy. Příkladem, bude-li usedColors obsahovat na pozici i, k hodnotu 1 znamená to, že na i-tém řádku již není možné použít barvu k. Poté se pro každý řádek inicializuje speciální pole newColor o velikosti počtu barev $\#colors$. Do tohoto pole se budou ukládat nuly a jedničky opět odpovídající barvám, které se na i-tém řádku již vyskytovat nemohou. For-cyklus na řádku 5 tedy odpovídá načtení všech nepřípustných barev pro i-tý řádek matice $\mathbb{A}$. Řádky 7-10 poté rozšíří pole newColor o použité barvy odpovídající transponovaným nenulovým prvkům i-tého řádku matice $\mathbb{A}$.

Podíváme-li se zpět na obrázek 2.1, u kterého si můžeme představit, že jsme v kroku $i = 2$, tedy na druhém řádku matice a řádky 3-8 jsou již obarveny, pak for-cyklus z pátého řádku algoritmu nám říká, že nemůžeme použít červenou a žlutou barvu, protože prvek 4 a 7 z druhého řádku matice jsou již těmito barvami obarvené. For-cyklus z řádku 7 pak říká, že nelze použít ani barvu tyrkysovou z důvodu prvku 1 z prvního řádku a zároveň nelze použít zelenou barvu z důvodu prvku 5 z nultého řádku matice $\mathbb{A}$.

Po průchodu těchto for-cyklů pole newColor obsahuje celkovou informaci o tom, které barvy nelze použít k obarvení i-tého řádku matice. Následující krok v algoritmu (řádek 11)

počítá, kolik barev se na daném i-tém řádku vyskytovat nesmí a pokud se toto číslo rovná celkovému počtu barev $\#colors$, tak je nutné rozšířit obě pole o další možnou barvu. Tyto kroky (řádky 11-13) jsou v algoritmu výrazné zvláště v prvním průchodu, ale i ve druhém průchodu tato možnost nastat může, protože obarvení nebude v obou průchodech algoritmu stejné a je možné, že nedokonalé obarvení ještě navýší počet nutných barev. V následujícím kroku se poté náhodně vybere jedna z možných barev (řádek 14), která se následně danému řádku přiřadí (řádek 15). Funkce EXTENDCOLORSARRAYSWITHONE a funkce FINDRANDOMCOLOR z řádků 13 a 14 jsou znázorněny v pseudoalgoritmu 9. Nakonec je nutné tuto barvu vyřadit z ostatních řádků, které se budou procházet v následujících krocích.

Algoritmus 9 Pomocné funkce k obarvení matice.

```
 1: function findRandomColor( newColor )
 2:     countPossibleColors = #colors - ∑_{k=0}^{#colors-1} newColor_k
 3:     newColorIndex = rand()  mod countPossibleColors + 1      ▷ rand() vrací náhodnou
    celočíselnou hodnotu z ⟨0, RAND_MAX)
 4:     color = 0
 5:     newColorCount = 0
 6:     while color < #colors do
 7:         if newColor_color = 0 then
 8:             newColorCount++
 9:             if newColorCount = newColorIndex then
10:                 break
11:         color++
       return color
```

Následující algoritmus 10 znázorňuje způsob obarvení matice $\mathbb{A}$ o rozměrech $n \times n$ uložené v symetrickém Ellpack formátu na CPU. Tento algoritmus je vysoce sekvenční a velmi náročný na výpočet, jedinou výhodou je nutnost počítání obarvení pouze dvakrát a následné násobení matice může probíhat s libovolnými vektory. Z toho důvodu se také nebude samotné obarvení zahrnovat do měření při zpracování výsledků.

Algoritmus 10 Způsob obarveni matice $\mathbb{A}$ v symetrickém Ellpack formátu.

```
1: function Colorization(𝔸, colorsVector)
2:     numOfColors = 1
3:     usedColors[𝔸.n][numOfColors]                          ▷ S počáteční hodnotou 0
4:     for row = 0; row < 𝔸.n; row ++ do
5:         rowBegin = row∗K
6:         rowEnd = (row+1)∗K
7:         newColor[numOfColors]                             ▷ S počáteční hodnotou 0
8:         for color = 0; color < numOfColors; color++ do
9:             newColor[color] |= usedColors[row][color]   ▷ Podivame se které barvy již byly
   v daném řádku použity (nad-diagonální prvky)
10:        for j = rowBegin; j < rowEnd; j++ do
11:            if 𝔸.columns[j] ≠ −1 then
12:                for color = 0; color < numOfColors; color++ do
13:                    newColor[color] |= usedColors[𝔸.columns[j]][color]   ▷ Podivame
   se, které barvy již byly v řádcích, odpovídajících sloupcům prvků tohoto řádku, použity
   (pod-diagonální prvky)
14:            else break
15:        count = negationOfnewColor( newColor ) ▷ Negace pole nám udá barvy, které je
   možné použít k obarvení daného řádku a spočte počet použitých barev
16:        if count == numberOfColors then
17:            numOfColors += 1
18:            newColor.resize( numOfColors )                ▷ Poč. hodnoty 0
19:            newColor[ numOfColors-1 ] = 1      ▷ nová barva je možná použít k obarvení
20:            usedColors.resize( 𝔸.n, numOfColors ) = usedColors ▷ zvětšení usedColors o 1
   barvu v každém řádku a překopírování starých hodnot
21:        color = findRandomColor( newColor )    ▷ vrací index náhodně vybrané 1 z pole
   newColor
22:        colorsVector[ row ] = color
23:        for j = rowBegin; j < rowEnd; j++ do
24:            if 𝔸.columns[j] ≠ −1 then
25:                usedColors[𝔸.columns[j]][color] = 1          ▷ nad-diagonální prvky
26:            else break
27:        usedColors[row][color] = 1    ▷ Na daný řádek se také nastaví barva jako použitá
```

V kódu 10 se funkce Colorization volá na matici $\mathbb{A}$ a vektor `colorsVector`. Tento vektor bude opět obsahovat výsledek o obarvení matice. Tedy každý řádek bude mít přiřazenou barvu tak, aby nevznikaly konflikty při násobení.

Je dobré zdůraznit, že obarvení zde ukázané funguje pro matice uložené na CPU v symetrickém Ellpack formátu. Tato matice se následně překopíruje na GPU a bude připravena k výpočtům SpMV. Než se tak stane je nutné se opět zamyslet nad podmínkou sloučených přístupů do paměti, kterou by bylo vhodné na GPU dodržet. V následujících odstavcích je tento problém blíže vysvětlen.

Uložení prvků v paměti CPU pro matici $\mathbb{A}$ z obrázku 2.1 je znázorněno v rovnici (2.6). Jelikož se kernelová funkce pro SpMV se speciálním maticovým obarvením bude na GPU spouštět s počtem vláken rovnu počtu řádků obarvených danou barvou, je nutné přeskládat prvky z vektoru `data` tak, aby byla splněna podmínka sloučených přístupů.

$$\begin{aligned} K = \quad & 5, \\ \texttt{data} = [\quad & 1,\ 0,\ 0,\ 0,\ 0, \\ & 2,\ 0,\ 0,\ 0,\ 0, \\ & 3,\ 4,\ 0,\ 0,\ 0, \\ & 1,\ 2,\ 0,\ 0,\ 0, \\ & 3,\ 0,\ 0,\ 0,\ 0, \\ & 5,\ 4,\ 1,\ 0,\ 0, \\ & 5,\ 5,\ 1,\ 1,\ 2, \\ & 6,\ 1,\ 9,\ 0,\ 0, \\ & 6,\ 7,\ 8,\ 3,\ 10\ ], \end{aligned} \tag{2.6}$$

Z důvodu malého počtu řádků matice $\mathbb{A}$ z obrázku 2.1 není přímo patrný důvod přehazování prvků tak, aby byly stejně obarvené řádky v paměti u sebe, ale pro větší matice toto dává dobrý význam. Představme si řídkou matici o jednom milionu řádků a třeba 10 barev použitých k obarvení této matice. Pokud bychom v tomto případě nepřehazovali řádky matice podle barev k sobě, museli bychom buďto spouštět kernelovou funkci s počtem vláken rovnu počtu řádků, což by znamenalo, že pro každou barvu by kernelovou funkcí prošlo 900 000 nečinných vláken a pouze 100 000 by jich provádělo nějakou práci. Druhou možností by bylo omezit se na 100 000 činných vláken, ale ta by poté přistupovala do paměti zcela náhodně a podmínku na sloučené přístupy do paměti bychom nemohli splnit. Vektor `data` s přeházenými prvky podle obarvení z obrázku 2.1 je znázorněn v rovnici (2.7). Nyní by červenou barvu počítala vlákna 0-2, fialovou barvu by zpracovávalo vlákno 3, modrou vlákno 4, tyrkysovou vlákno 5 a zelenou vlákno 6, nakonec na žlutou barvu zbudou vlákna 7 a 8.

Je zřejmé, že obarvení není ideální. Může se stát, že počet použitých barev by mohl být nižší či rovnoměrněji rozložený. Kód pro dosažení ideálního obarvení jsme ale nenašli ani nevymysleli, takže se spokojíme s tímto způsobem.

$$
\begin{aligned}
\tilde{n} = &\quad 32, \\
K = &\quad 5, \\
\texttt{data} = [&\quad 1,\ 0,\ 0,\ 0,\ 0, \\
&\quad 3,\ 0,\ 0,\ 0,\ 0, \\
&\quad 5,\ 4,\ 1,\ 0,\ 0, \\
&\quad 2,\ 0,\ 0,\ 0,\ 0, \\
&\quad 3,\ 4,\ 0,\ 0,\ 0, \\
&\quad 1,\ 2,\ 0,\ 0,\ 0, \\
&\quad 5,\ 5,\ 1,\ 1,\ 2, \\
&\quad 6,\ 1,\ 9,\ 0,\ 0, \\
&\quad 6,\ 7,\ 8,\ 3,\ 10\ ],
\end{aligned}
\tag{2.7}
$$

Tím, že přeházíme řádky matice podle barev, nám vzniknou další dvě důležitá pole a sice `permutationArray`, které obsahuje informaci o přeházení řádků. Příkladem, při prohození řádku 1 za řádek 2 bude `permutationArray` obsahovat číslice 2 a 1 v tomto pořadí. Tedy 1. řádek v přeházené matici $\mathbb{A}$ je vlastně druhý řádek z původní matice $\mathbb{A}$. Druhým polem je `colorsPointer` o velikosti numOfColors + 1, které obsahuje informace, od kterého řádku a do kterého řádku je daná barva v přeskládané matici $\mathbb{A}$.

Vektory `permutationArray` a `colorsPointer` jsou identické pro matici uloženou na CPU i GPU. Tyto vektory odkazují na matici a nikoli na vektor `data`. Nyní by byla matice připravena k výpočtu a je tedy čas na již zmíněné kopírování na GPU, které nijak neovlivní vektory `permutationArray` a `colorsPointer`.

Funkce pro výpočet SpMV na CPU s maticí $\mathbb{A}$ uloženou v přerovnaném symetrickém Ellpack formátu je znázorněna v algoritmu 11.

Algoritmus 11 Funkce pro SpMV se symetrickým Ellpack formátem a speciálním maticovým obarvením na CPU.

```
1: for color = 0; color < 𝔸.numOfColors; color++ do
2:     colorStart = colorsVector[colors]
3:     colorEnd = colorsVector[colors+1]
4:     numRows = colorsVector[colors+1] - colorsVector[colors]
5:     #pragma omp parallel for
6:     for j = 0; j < numRows; j++ do
7:         row = colorStart + j
8:         if row ≤ colorEnd then
9:             break
10:        i = row∗𝔸.K
11:        rowEnd = (row+1)∗𝔸.K
12:        colIdx = 0
13:        while i < rowEnd ∧ ( colIdx = 𝔸.columns[i] ) ≠ -1 do
14:            b[ 𝔸.permutationArray[rowIdx] ] += 𝔸.data[i]∗x[colIdx]
15:            if 𝔸.permutationArray[rowIdx] ≠ colIdx then
16:                b[colIdx] += 𝔸.data[i]∗x[ 𝔸.permutationArray[rowIdx] ] )
17:            i += 1
```

Kernelová funkce pro násobení řídké symetrické matice s vektorem pomocí speciálního maticového obarvení je znázorněna v algoritmu 12. Pro názornost je uvedeno gridSize rovno 1 ale je zřejmé, že se dá kód upravit i pro větší počet řádků na jednu barvu než je 1024, což je maximální počet vláken v jednom bloku.

Algoritmus 12 Kernelová funkce pro SpMV se symetrickým Ellpack formátem a speciálním maticovým obarvením na GPU.

```
1: for IndexType color = 0; color < 𝔸.numOfColors; color++ do
2:     IndexType blockSize = colorsVector[colors+1] - colorsVector[colors]
3:     IndexType gridSize = 1
4:     SpMVColorFunction⋘ gridSize, blockSize ⋙(𝔸, x, b, color)

5: function ___global___ SpMVColorFunction(const EllpackSymmetric* 𝔸,
                                            const Real* x,
                                            Real* b,
                                            IndexType colors )
6:     rowIdx = blockIdx.x ∗ blockDim.x + threadIdx.x + 𝔸.colorPointers[colors]
7:     IndexType i = rowIdx
8:     IndexType el = 0
9:     IndexType colIdx
10:    while el++ < 𝔸.K ∧
11:      ( colIdx = 𝔸.columns[i] ) < 𝔸.m ∧
12:       colIdx ≠ 𝔸.ñ do
13:       b[ 𝔸.permutationArray[rowIdx] ] += 𝔸.data[i]∗x[colIdx]        ▷ pod-diagonální a
   diagonální prvky
14:       if 𝔸.permutationArray[rowIdx] ≠ colIdx then
15:           b[colIdx] += 𝔸.data[i]∗x[ 𝔸.permutationArray[rowIdx] ] )    ▷ nad-diagonální
   prvky
16:       i += 𝔸.ñ
```

V obou algoritmech 11, 12 se sekvenčně prochází barvy a paralelně řádky obsažené v těchto barvách. Následné násobení je již obdobné klasickému SpMV se symetrickým Ellpack formátem.

Kapitola 3

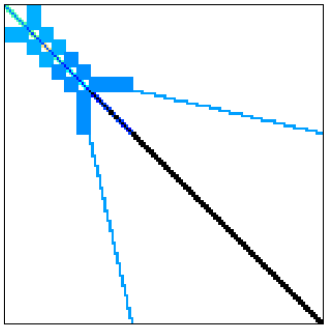
Výsledky násobení matice s vektorem

V této kapitole ukážeme výsledky získané při násobení řídkých symetrických matic uložených v symetrickém Ellpack formátu s vektorem jedniček a to pro CPU i GPU s přesností float i double s řešením konfliktů pomocí atomických instrukcí i speciálního maticového obarvení.

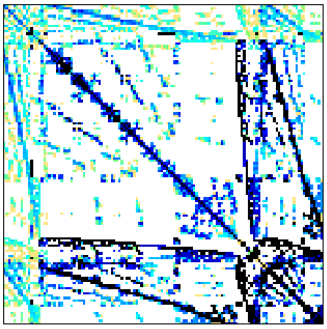
Výpočty SpMV s OpenMP se prováděly na školním počítači Teslon, který je vybaven 10-jádrovým procesorem Intel® Xeon® E5-2640 v4 se základní frekvencí 2.4GHz a cache pamětí 20 MB.

Výpočty SpMV s jednoduchou i dvojitou přesností se poté prováděly na fakultním klastru Helios, který je vybaven 16-jádrovým procesorem Intel® Xeon® Gold 6130 se základní frekvencí 2.1GHz a cache pamětí 22 MB. Výpočty se na tomto procesoru prováděly bez OpenMP. Dále se na Heliosu nachází grafický akcelerátor NVIDIA® TESLA V100 (Volta) s globální pamětí o velikosti 16GB. Na Heliosu je nainstalován GCC překladač verze 6.5.0, CUDA 10.2 a MPI 2.1.5.

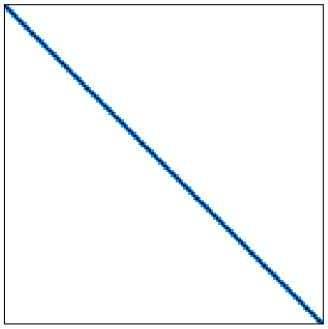
Matice použité při násobení byly převzaty z [14] a jejich náhledy jsou zobrazeny na obrázcích 3.1-3.5. Popis těchto matic je zaznamenán v tabulce 3.1.



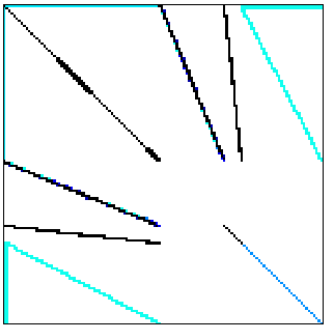
Obrázek 3.1: Matice G3_circuit o 1 585 478 řádcích s 4 623 152 nenulovými prvky.



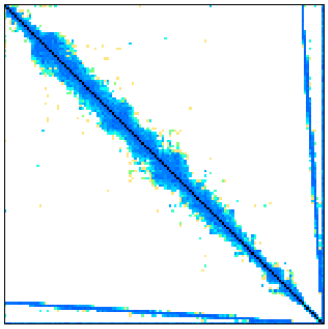
Obrázek 3.2: Matice crankseg_2 o 63 838 řádcích s 7 106 348 nenulovými prvky.



Obrázek 3.3: Matice ecology1 o 1 000 000 řádcích s 2 998 000 nenulovými prvky.



Obrázek 3.4: Matice kkt_power o 2 063 494 řádcích s 8 130 343 nenulovými prvky.



Obrázek 3.5: Matice thermal2 o 1 228 045 řádcích s 4 904 179 nenulovými prvky.

Tabulka 3.1: Tabulka parametrů matic použitých pro výpočet GEM.

matice	#řádků×#sloupců	#nenul. prvků	#barev	Ellpack #MB	sym. Ellpack #MB
crankseg_2	63 838 × 63 838	14 148 858	3 423	53,97	27,11
ecology1	1 000 000 × 1 000 000	4 996 000	4	19,06	11,44
thermal2	1 228 045 × 1 228 045	8 580 313	12	32,73	18,71
G3_circuit	1 585 478 × 1 585 478	7 660 826	4	29,22	17,64
kkt_power	2 063 494 × 2 063 494	12 771 361	106	48,71	27,50

V tabulce 3.1 si můžeme všimnout velikosti použitých matic spolu s počty nenulových prvků a přibližným počtem barev nutných ke speciálnímu maticovému obarvení. Počty barev jsou přibližné, protože obarvení je z části náhodné a proto může při výpočtu dojít k různým obarvením matice. Dále je ke každé matici udána velikosti paměti, kterou zabírají samotné hodnoty matice, bez vektoru `columns` a zarovnání paměti pro GPU. Skutečné velikosti matic v paměti by byly škálované stejným způsobem.

3.1 CPU SpMV s OpenMP

V této kapitole představíme výsledky pro SpMV na CPU za použití OpenMP pro testovací matice z tabulky 3.1. Pro výpočty byl výpočetní počítač Teslon s 10-jádrovým procesorem Xeon popsaným výše, který je schopen pomocí hyper-threading dosáhnout 20 vláken. Výpočty jsme prováděly pro 1, 2, 4, 8 a 10 vláken.

3.1.1 CPU SpMV s jednoduchou přesností a OpenMP

V tabulce 3.2 jsou zobrazeny časy výpočtů SpMV pro Ellpack na CPU v jednoduché přesnosti za použití OpenMP s 1, 2, 4, 8 a 10 vlákny. Z tabulky si můžeme všimnout, že nejrychlejší výpočet pro testovací matice proběhl s 10 vlákny OpenMP. Stejný závěr lze vypozorovat i z tabulky urychlení SpMV pro Ellpack 3.3.

Tabulka 3.2: Porovnání časů výpočtů SpMV s jednoduchou přesností pro Ellpack na CPU s 1, 2, 4, 8 a 10 vlákny OpenMP.

	Časy výpočtů SpMV pro formát Ellpack s OpenMP				
	1 vl.	2 vl.	4 vl.	8 vl.	10 vl.
crankseg_2	0,02995	0,03892	0,02330	0,01515	0,01245
ecology1	0,00457	0,00444	0,00380	0,00236	0,00127
thermal2	0,01781	0,01401	0,00855	0,00665	0,00566
G3_circuit	0,00914	0,00751	0,00628	0,00456	0,00466
kkt_power	0,08326	0,06558	0,05458	0,04109	0,03639

Tabulka 3.3: Tabulka urychlení SpMV s jednoduchou přesností pro Ellpack na CPU s 1×2, 1×4, 1×8 a 1×10 vlákny OpenMP.

	Urychlení SpMV pro formát Ellpack s OpenMP			
	1 vl.×2 vl.	1 vl.×4 vl	1 vl.×8 vl	1 vl.×10 vl
crankseg_2	0,77	1,29	1,98	2,41
ecology1	1,03	1,20	1,94	3,60
thermal2	1,27	2,08	2,68	3,15
G3_circuit	1,22	1,46	2,00	1,96
kkt_power	1,27	1,53	2,03	2,29

V tabulce 3.4 jsou znázorněny časy výpočtu SpMV s jednoduchou přesností pro symetrický Ellpack formát s atomickými instrukcemi na CPU s 1, 2, 4, 8 a 10 vlákny OpenMP. Opět si můžeme všimnout, že nejlepší časy výpočtu dosahuje SpMV na CPU s 10 vlákny OpenMP. Dále si můžeme všimnout, že výpočty SpMV pro Ellpack formát byly rychlejší než pro symetrický Ellpack formát s atomickými instrukcemi. V tabulce 3.5 je zobrazeno urychlení napočítané z tabulky 3.4.

Tabulka 3.4: Porovnání časů výpočtů SpMV s jednoduchou přesností pro symetrický Ellpack formát s atomickými instrukcemi na CPU s 1, 2, 4, 8 a 10 vlákny OpenMP.

	Časy výpočtů SpMV pro symetrický Ellpack formát s atom. instr. a OpenMP				
	1 vl.	2 vl.	4 vl.	8 vl.	10 vl.
crankseg_2	0,10099	0,08711	0,06842	0,03077	0,02932
ecology1	0,03504	0,02589	0,01459	0,01063	0,00903
thermal2	0,06496	0,04906	0,03087	0,01865	0,01769
G3_circuit	0,05413	0,04210	0,02282	0,01268	0,01060
kkt_power	0,14198	0,19611	0,12475	0,06087	0,06161

Tabulka 3.5: Tabulka urychlení SpMV s jednoduchou přesností pro symetrický Ellpack formát s atomickými instrukcemi na CPU s 1×2, 1×4, 1×8 a 1×10 vlákny OpenMP.

	Urychlení SpMV pro symetrický Ellpack formát s atom. instr. s OpenMP			
	1 vl.×2 vl.	1 vl.×4 vl	1 vl.×8 vl	1 vl.×10 vl
crankseg_2	1,16	1,48	3,28	3,44
ecology1	1,35	2,40	3,30	3,88
thermal2	1,32	2,10	3,48	3,67
G3_circuit	1,29	2,37	4,27	5,11
kkt_power	0,72	1,14	2,33	2,30

Nakonec v tabulce 3.6 se nacházejí časy výpočtů SpMV pro symetrický Ellpack formát se speciálním maticovým obarvením. Tyto časy dopadly opět nejlépe pro 10 vláken OpenMP, což potvrzuje i tabulka urychlení 3.7.

Tabulka 3.6: Porovnání časů výpočtů SpMV s jednoduchou přesností pro symetrický Ellpack formát se speciálním maticovým obarvením na CPU s 1, 2, 4, 8 a 10 vlákny OpenMP.

	Časy výpočtů SpMV pro sym. Ellpack se spec. mat. obar. a OpenMP				
	1 vl.	2 vl.	4 vl.	8 vl.	10 vl.
crankseg_2	0,03320	0,04524	0,02347	0,01633	0,01238
ecology1	0,00849	0,01248	0,00841	0,00848	0,00897
thermal2	0,03831	0,05089	0,02192	0,02816	0,02526
G3_circuit	0,02078	0,02313	0,02148	0,01954	0,01847
kkt_power	0,10940	0,10400	0,08611	0,06390	0,04238

Tabulka 3.7: Tabulka urychlení SpMV s jednoduchou přesností pro symetrický Ellpack formát se speciálním maticovým obarvením na CPU s 1×2, 1×4, 1×8 a 1×10 vlákny OpenMP.

	Urychlení SpMV pro sym. Ellpack se spec. mat. obar. s OpenMP			
	1 vl.×2 vl.	1 vl.×4 vl	1 vl.×8 vl	1 vl.×10 vl
crankseg_2	0,73	1,41	2,03	2,68
ecology1	0,68	1,01	1,00	0,95
thermal2	0,75	1,75	1,36	1,52
G3_circuit	0,90	0,97	1,06	1,13
kkt_power	1,05	1,27	1,71	2,58

Celkově z tabulek 3.3, 3.5 a 3.7 plyne, že pro více vláken se čas výpočtů SpMV zkracuje. OpenMP zkracuje výpočet přibližně tolikrát, kolik se použije vláken. Tento závěr není potvrzen u některých matic z tabulky 3.7 s urychlením pro sym. Ellpack formát a speciální maticové obarvení z důvodu paralelizace přes počet barev. Některé testovací matice totiž nemají dostatečný počet barev pro využití všech použitých vláken OpenMP.

3.1.2 CPU SpMV s dvojitou přesností a OpenMP

V této kapitole představíme výsledky SpMV s OpenMP na CPU ve dvojité přesnosti pro Ellpack, symetrický Ellpack s atomickými instrukcemi a symetrický Ellpack se speciálním maticovým obarvením. Výpočty se opět prováděly na fakultním počítači Teslon, jehož parametry byly uvedeny na začátku této kapitoly.

V tabulce 3.8 jsou uvedeny časy výpočtů SpMV s dvojitou přesností na CPU pro 1, 2, 4, 8 a 10 vláken OpenMP pro Ellpack formát. Napočítané urychlení je poté zobrazeno v tabulce 3.9. Z obou tabulek plyne, že výpočet nejrychleji probíhal pro 10 vláken OpenMP.

Tabulka 3.8: Porovnání časů výpočtů SpMV s dvojitou přesností pro Ellpack na CPU s 1, 2, 4, 8 a 10 vlákny OpenMP.

	Časy výpočtů SpMV pro formát Ellpack s OpenMP				
	1 vl.	2 vl.	4 vl.	8 vl.	10 vl.
crankseg_2	0,03083	0,03735	0,02090	0,01184	0,01123
ecology1	0,00586	0,00637	0,00349	0,00279	0,00258
thermal2	0,02168	0,02235	0,01153	0,00878	0,00675
G3_circuit	0,01343	0,01116	0,01022	0,00821	0,00862
kkt_power	0,09209	0,08526	0,06471	0,03544	0,03482

Tabulka 3.9: Tabulka urychlení SpMV s dvojitou přesností pro Ellpack na CPU s 1×2, 1×4, 1×8 a 1×10 vlákny OpenMP.

	Urychlení SpMV pro formát Ellpack s OpenMP			
	1 vl.×2 vl.	1 vl.×4 vl	1 vl.×8 vl	1 vl.×10 vl
crankseg_2	0.83	1.48	2.60	2.75
ecology1	0.92	1.68	2.10	2.27
thermal2	0.97	1.88	2.47	3.21
G3_circuit	1.20	1.31	1.64	1.56
kkt_power	1.08	1.42	2.60	2.64

V tabulce 3.10 jsou znázorněny časy výpočtů SpMV na CPU pro symetrický Ellpack formát s atomickými instrukcemi opět pro 1, 2, 4, 8 a 10 vláken OpenMP. Odpovídající urychlení je v tabulce 3.11. Opět platí závěr, že nejkratší čas výpočtu byl dosažen pro 10 vláken OpenMP.

Tabulka 3.10: Porovnání časů výpočtů SpMV s dvojitou přesností pro symetrický Ellpack formát s atomickými instrukcemi na CPU s 1, 2, 4, 8 a 10 vlákny OpenMP.

	Časy výpočtů SpMV pro symetrický Ellpack formát s atom. instr. a OpenMP				
	1 vl.	2 vl.	4 vl.	8 vl.	10 vl.
crankseg_2	0,10314	0,07986	0,05721	0,03965	0,03326
ecology1	0,03516	0,02662	0,01496	0,00892	0,00760
thermal2	0,06724	0,06405	0,03326	0,02560	0,01948
G3_circuit	0,05534	0,04602	0,02487	0,01506	0,01313
kkt_power	0,17318	0,22806	0,15809	0,06511	0,07476

Tabulka 3.11: Tabulka urychlení SpMV s dvojitou přesností pro symetrický Ellpack formát s atomickými instrukcemi na CPU s 1×2, 1×4, 1×8 a 1×10 vlákny OpenMP.

	Urychlení SpMV pro sym. Ellpack formát s atom. instr. s OpenMP			
	1 vl.×2 vl.	1 vl.×4 vl	1 vl.×8 vl	1 vl.×10 vl
crankseg_2	1.29	1.80	2.60	3.10
ecology1	1.32	2.35	3.94	4.63
thermal2	1.05	2.02	2.63	3.45
G3_circuit	1.20	2.23	3.67	4.21
kkt_power	0.76	1.10	2.66	2.32

V tabulkách 3.12 a 3.13 se nachází časy výpočtů SpMV s dvojitou přesností na CPU pro symetrický Ellpack formát se speciálním maticovým obarvením opět pro 1, 2, 4, 8 a 10 vláken OpenMP. Z obout tabulek plyne stejný závěr jako v předchozích příkladech, totiž, že nejrychleji výpočet proběhne pro 10 vláken OpenMP.

Tabulka 3.12: Porovnání časů výpočtů SpMV s dvojitou přesností pro symetrický Ellpack formát se speciálním maticovým obarvením na CPU s 1, 2, 4, 8 a 10 vlákny OpenMP.

	Časy výpočtů SpMV pro sym. Ellpack se spec. mat. obar. a OpenMP				
	1 vl.	2 vl.	4 vl.	8 vl.	10 vl.
crankseg_2	0,03582	0,05710	0,02773	0,01803	0,01957
ecology1	0,01304	0,01806	0,01141	0,01139	0,01084
thermal2	0,04783	0,07732	0,03844	0,02846	0,02380
G3_circuit	0,03570	0,05340	0,03363	0,02552	0,02458
kkt_power	0,13011	0,13361	0,07236	0,04386	0,05065

Tabulka 3.13: Tabulka urychlení SpMV s dvojitou přesností pro symetrický Ellpack formát se speciálním maticovým obarvením na CPU s 1×2, 1×4, 1×8 a 1×10 vlákny OpenMP.

	Urychlení SpMV pro sym. Ellpack se spec. mat. obar s OpenMP			
	1 vl.×2 vl.	1 vl.×4 vl	1 vl.×8 vl	1 vl.×10 vl
crankseg_2	0.63	1.29	1.99	1.83
ecology1	0.72	1.14	1.14	1.20
thermal2	0.62	1.24	1.68	2.01
G3_circuit	0.67	1.06	1.40	1.45
kkt_power	0.97	1.80	2.97	2.57

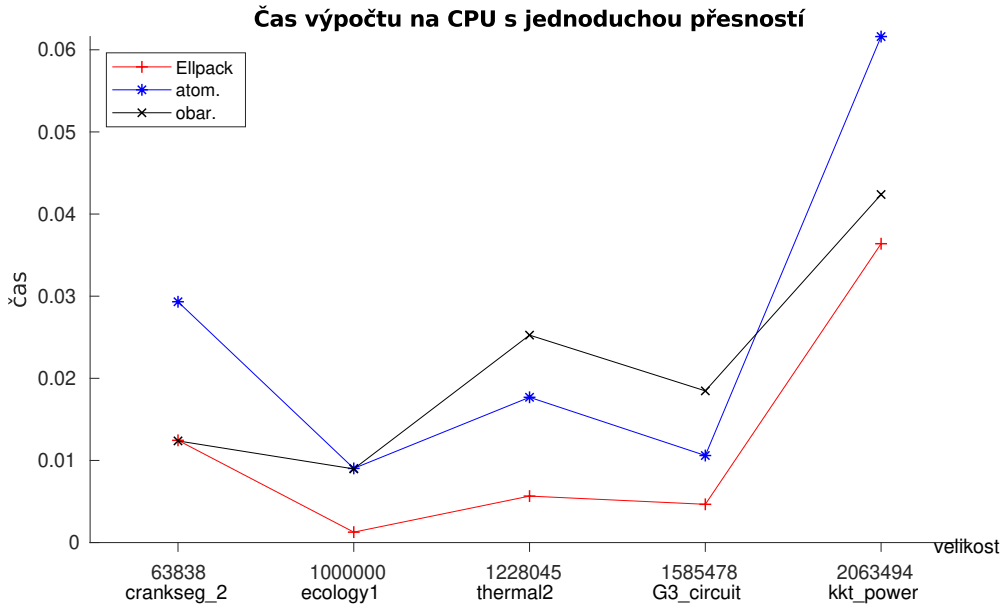
3.2 SpMV s jednoduchou přesností

V této kapitole představíme výsledky pro testovací matice z tabulky 3.1. Výpočty pro GPU se prováděly na klastru Helion, zatímco pro výsledky na CPU se použil fakultní počítač Teslon. Parametry obou zařízení jsou popsány na začátku této kapitoly. Pro CPU se použilo 10 vláken OpenMP z důvodu nejlepších výsledků dosažených na CPU.

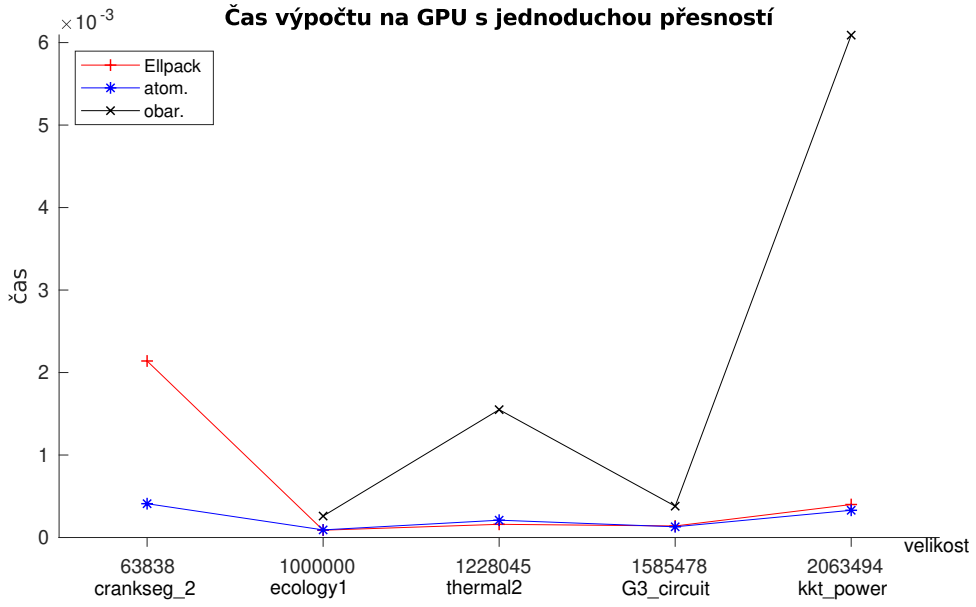
V tabulce 3.14 jsou znázorněny časy výpočtů pro matice uložené pomocí formátu Ellpack a symetrického Ellpack formátu s atomickými instrukcemi a symetrického Ellpack formátu se speciálním maticovým obarvením na CPU a GPU. Již z tabulky si můžeme všimnout, že výpočty provedené na CPU jsou obecně delší než výpočty na GPU. Zajímavá hodnota byla naměřena pro speciální maticové obarvení na GPU pro matici crankseg_2. Tato hodnota je řádově jinde, než ostatní z důvodu větší hustoty matice a tedy i velkého počtu barev nutného k obarvení této matice. Hodnoty z tabulky 3.14 jsou zobrazeny i na obrázku 3.6 pro CPU a na obrázku 3.7 pro GPU. Dalším zajímavým výsledkem je porovnání GPU výpočtů pro matici crankseg_2, kde symetrický Ellpack formát s atomickými instrukcemi výrazně předčil výpočet klasického Ellpack formátu i přes fakt, že vyšší hustota matice znamená více konfliktů v paměti výstupního vektoru. Pro ostatní matice má výpočet na GPU pro Ellpack formát a symetrický Ellpack formát s atomickými instrukcemi podobné výsledky.

Tabulka 3.14: Porovnání časů výpočtů SpMV s jednoduchou přesností pro Ellpack na CPU a GPU a symetrický Ellpack formát s atomickými instrukcemi a speciálním maticovým obarvením na CPU a GPU.

	čas SpMV Ellpack		čas SpMV Sym. Ellpack			
			atomické instrukce		speciální obarvení	
	CPU	GPU	CPU	GPU	CPU	GPU
crankseg_2	0,01245	0,00214	0,02932	0,00041	0,01238	0,54604
ecology1	0,00127	8,88e-05	0,00903	9,22e-05	0,00897	0,00026
thermal2	0,00566	0,00016	0,01769	0,00021	0,02526	0,00155
G3_circuit	0,00466	0,00014	0,01060	0,00013	0,01847	0,00038
kkt_power	0,03639	0,00040	0,06161	0,00033	0,04238	0,00609



Obrázek 3.6: Výsledky SpMV pro testovací matice z tabulky 3.1 s jednoduchou přesností na CPU.



Obrázek 3.7: Výsledky SpMV pro testovací matice z tabulky 3.1 s jednoduchou přesností na GPU.

Na základě měření z tabulky 3.14 a obrázků 3.6 a 3.7 lze říci, že GPU výpočty jsou rychlejší než CPU. Z CPU výpočtů se jako výhodné ukazuje použití klasického Ellpack formátu bez konfliktů v paměti, zatímco z GPU výpočtů se jedná o symetrický Ellpack formát s použitím atomických instrukcí. Ekvivalentní závěry se dají vyvodit i z tabulek urychlení 3.15 a 3.16.

Tabulka 3.15: Tabulka urychlení CPu×GPU pro Ellpack, symetrický Ellpack s atomickými instrukcemi a symetrický Ellpack se speciálním maticovým obarvením.

	urychlení CPU×GPU		
	Ellpack	Sym. Ellpack atom.	Sym. Ellpack obarveni
crankseg_2	5,82	71,51	0,02
ecology1	14,3	97,94	34,5
thermal2	35,38	84,24	16,30
G3_circuit	33,29	81,54	48,61
kkt_power	90,98	186,70	6,96

Tabulka 3.16: Tabulka urychlení CPU×CPU a GPU×GPU pro Ellpack a symetrický Ellpack se speciálním maticovým obarvením oproti symetrickému Ellpack formátu s atomickými instrukcemi.

	urychlení CPU×CPU		urychlení GPU×GPU	
	Ellpack	obarveni	Ellpack	obarveni
crankseg_2	0,42	0,42	5,220	1331,805
ecology1	0,14	0,99	0,963	2,820
thermal2	0,32	0,51	0,762	7,381
G3_circuit	0,44	1,74	1,077	2,923
kkt_power	0,59	0,69	1,212	18,455

Z tabulky 3.15 opět plyne, že CPU výpočty jsou pomalejší než GPU výpočty.

Z tabulky 3.16 si můžeme všimnout, že Ellpack i symetrický Ellpack s atomickými instrukcemi mají podobné výsledky, zatímco symetrický Ellpack se speciálním maticovým obarvením se ukázal jako neefektivní.

V následující kapitole provedeme stejnou analýzu pro výpočty s dvojitou přesností.

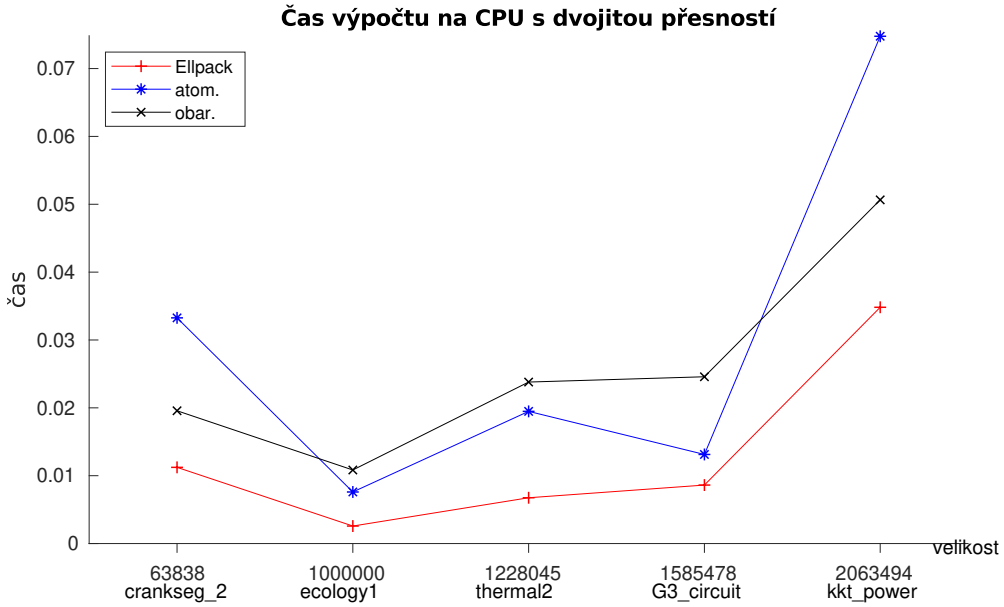
3.3 SpMV s dvojitou přesností

V této kapitole ukážeme výsledky pro SpMV s dvojitou přesností a to pro Ellpack, symetrický Ellpack s atomickými instrukcemi a symetrický Ellpack se speciálním maticovým obarvením na CPU i GPU. Pro výpočty na CPU se použilo 10 vláken OpenMP na fakultním počítači Teslon, zatímco pro GPU výpočty byl využit klastr Helios.

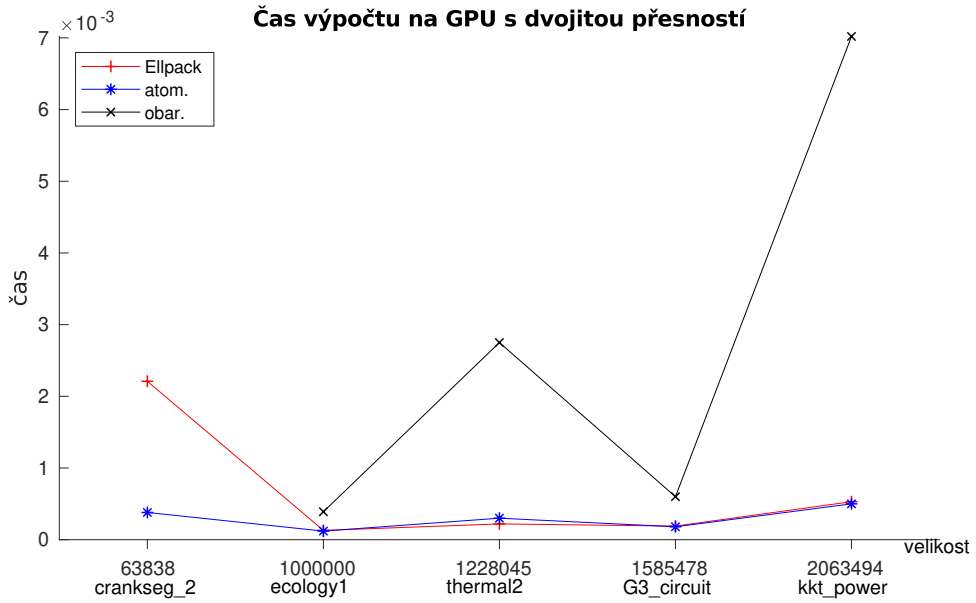
V tabulce 3.17 jsou zobrazeny časy výpočtů pro matice z tabulky 3.1. Tyto časy jsou poté zobrazeny pro CPU na obrázku 3.8 a pro GPU na obrázku 3.9, kde není vynesená hodnota pro matici cranseg_2 se speciálním maticovým obarvením z důvodu řádově vyšší hodnoty.

Tabulka 3.17: Porovnání časů výpočtů SpMV s dvojitou přesností pro Ellpack na CPU a GPU a symetrický Ellpack formát s atomickými instrukcemi a speciálním maticovým obarvením na CPU a GPU.

	čas SpMV Ellpack		čas SpMV Sym. Ellpack			
			atomické instrukce		speciální obarvení	
	CPU	GPU	CPU	GPU	CPU	GPU
crankseg_2	0,01123	0,00221	0,03326	0,00038	0,01957	0,57573
ecology1	0,00258	0,00013	0,00760	0,00012	0,01084	0,00039
thermal2	0,00675	0,00022	0,01948	0,00030	0,02380	0,00275
G3_circuit	0,00862	0,00019	0,01313	0,00018	0,02458	0,00060
kkt_power	0,03482	0,00053	0,07476	0,00050	0,05065	0,00702



Obrázek 3.8: Výsledky SpMV pro testovací matice z tabulky 3.1 s dvojitou přesností na CPU.



Obrázek 3.9: Výsledky SpMV pro testovací matice z tabulky 3.1 s dvojitou přesností na GPU.

Z tabulky 3.17 i obrázků 3.8, 3.9 si opět můžeme všimnout, že časy CPU výpočtů jsou delší než časy GPU výpočtů. Z CPU se opět jako lepší volba jeví Ellpack formát, naopak pro GPU vypadají Ellpack formát a symetrický Ellpack formát s atomickými instrukcemi jako vhodné volby. Symetrický Ellpack formát se speciálním maticovým obarvením ve výpočtech zaostává.

Ekvivalentní závěry se dají vyvodit i z tabulky 3.18 urychlení pro CPU vs GPU a z tabulky 3.19 urychlení pro CPU vs CPU a GPU vs GPU pro jednotlivé formáty a algoritmy. Z těchto tabulek opět plyne, že nejlepší pro výpočty na CPU je Ellpack formát, pokud máme dostatek paměti. Naopak pro výpočty na GPU je vhodnější symetrický Ellpack formát s atomickými instrukcemi, který má podobný čas výpočtu jako klasický Ellpack formát, ale zároveň ušetří část paměti.

Tabulka 3.18: Tabulka urychlení CPU×GPU pro Ellpack, symetrický Ellpack s atomickými instrukcemi a symetrický Ellpack se speciálním maticovým obarvením.

	urychlení CPU×GPU		
	Ellpack	Sym. Ellpack atom.	Sym. Ellpack obarveni
crankseg_2	5,08	87,53	0,03
ecology1	19,85	63,33	27,79
thermal2	30,68	64,93	8,65
G3_circuit	45,37	72,94	40,97
kkt_power	65,70	149,52	7,22

Tabulka 3.19: Tabulka urychlení CPU×CPU a GPU×GPU pro Ellpack a pro symetrický Ellpack se speciálním maticovým obarvením oproti symetrickému Ellpack formátu s atomickými instrukcemi.

	urychlení CPU×CPU		urychlení GPU×GPU	
	Ellpack	obarveni	Ellpack	obarveni
crankseg_2	0,34	0,59	5,816	1515,079
ecology1	0,34	1,43	1,083	3,250
thermal2	0,35	1,22	0,733	9,167
G3_circuit	0,66	1,87	1,056	3,333
kkt_power	0,47	0,68	1,060	14,040

Kapitola 4

Gaussova eliminační metoda

V této kapitole představíme základní verzi sekvenční Gaussovy eliminační metody a její paralelní verzi na GPU, jenž jsme implementovali pomocí knihovny TNL. Paralelní verzi Gaussovy eliminační metody poté rozšíříme pomocí nástroje MPI pro výpočty na výpočetních klastrech. Nakonec ukážeme výsledky, kterých jsme pomocí paralelních verzí dosáhli.

4.1 Sekvenční Gaussova eliminační metoda

Gaussova eliminační metoda, dále jen GEM, je nejjednodušší metoda pro řešení soustavy lineárních rovnic. Metoda se skládá ze dvou základních částí. V první části se matice soustavy převádí do horního trojúhelníkového tvaru pomocí elementárních řádkových úprav prohození řádků a odečtení násobku jednoho řádku od násobku druhého řádku. Druhou částí je následná zpětná substituce, pomocí které se dopočítává řešení soustavy, viz [12]. Obě části GEM jsou vysoce sekvenční a dohromady dávají složitost algoritmu $\mathcal{O}(n^3)$, viz [13].

Algoritmus jednoduchého sekvenčního GEM pro matici soustavy $\mathbb{A}$ o rozměrech $n \times n$ a vektorem pravé strany $\boldsymbol{b}$ o rozměrech $n \times 1$ jsou znázorněny v algoritmu 13. V tomto algoritmu se pro každý sloupec k nejprve znormuje k-tý řádek prvkem $A(k,k)$. Tento řádek nazýváme hlavním nebo také pivotním řádkem a prvek $A(k,k)$ nazýváme pivotem. Po znormování hlavního k-tého řádku se pro všechny řádky pod hlavním řádkem, tedy řádky $k+1$ až n, provádí ekvivalentní řádková úprava a to odečtení od každého řádku řádek hlavní s násobkem prvního nenulového prvku na každém z těchto řádků $A(i,k)$. Touto úpravou vzniknou všude ve sloupci k pod diagonálou nuly, viz 4.1. Opakováním těchto jednoduchých kroků pro všechna $k \in \langle 0, n)$ se matice převede na horní trojúhelníkový tvar. Po převedení matice do tohoto tvaru se dále provádí zpětná substituce, což odpovídá výpočtu řešení.

Hlavní nevýhodou tohoto jednoduchého algoritmu, krom sekvenčnosti, je numerická stabilita. Představíme-li si, že prvek $A(k,k)$ je nulový, pak algoritmus nelze provádět z důvodu nedefinované operace a to dělení nulou. Druhou nevhodnou možností je prvek $A(k,k)$ blízký nule. V tomto případě se dělení tímto prvkem při normování hlavního řádku vkládá do výpočtu numerická chyba, která se dále propaguje do všech ostatních řádků a tím i do výsledku. Z tohoto důvodu se do jednoduchého algoritmu vkládá částečné nebo celkové pivotování. Pivotováním se zajišťuje lepší numerická stabilita algoritmu.

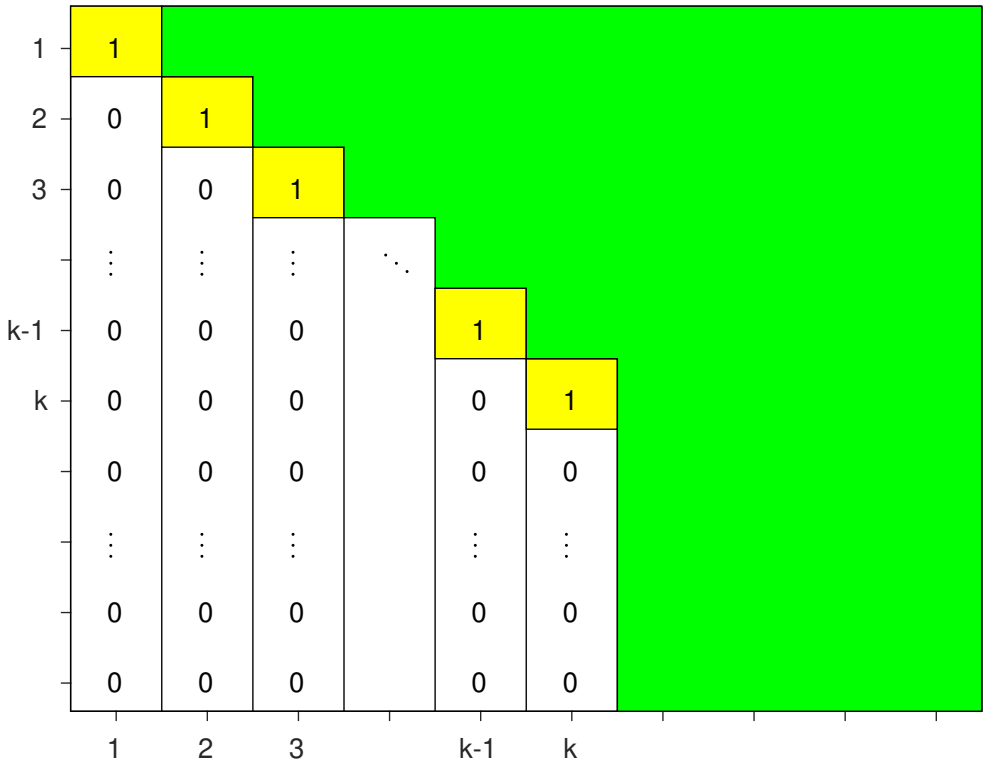
Částečné pivotování je hledání maximálního prvku v absolutní hodnotě pod diagonálou v daném sloupci k, který odpovídá kroku výpočtu. Následné přehození k-tého řádku a řádku s maximem v k-tém sloupci pod k-tým řádkem se nám dostane toto maximum na pivotní prvek, který jsme dříve označili jako $A(k,k)$. Následně algoritmus pokračuje normováním hlavního řádku a odečítáním od ostatních řádků, jak již bylo zmíněno výše. Částečné pivotování je nutnou součásti GEM metody protože nenulovost prvku $A(k,k)$ v k-tém kroku algoritmu nelze na začátku nijak zaručit. Pro obecné matice tento prvek ale nemusí být tím nejvhodnějším.

Lepším adeptem na pivotní prvek v k-tém kroku algoritmu je maximální prvek v absolutní hodnotě z části matice $\mathbb{A}$ ohraničené k-tým sloupcem a k-tým řádkem. Jedná se o prvky matice $A(i,j)$, kde $i \in \langle k+1, n)$ a $j \in \langle k+1, n)$. Po nalezení tohoto maxima je nutné prohodit nejen řádky ale i sloupce matice $\mathbb{A}$ tak, aby se maximum dostalo na pivotní prvek $A(k,k)$. Prohozením sloupců ale dojde i k prohození výsledného vektoru. Tyto permutace je nutné v průběhu algoritmu ukládat abychom při zpětné substituci následně provedli inverzní permutaci výsledného vektoru a tím dostali správné řešení. Z důvodu složitosti permutování se omezíme pouze na částečné pivotování při výpočtu GEM.

Z důvodu sekvenčnosti a složitosti se nejjednodušší verze GEM při výpočtech nepoužívá. Metodu je možné paralelizovat několika způsoby. Na jeden z těchto způsobů se v této práci zaměříme.

Algoritmus 13 Sekvenčí algoritmus Gaussovy metody pro matici $\mathbb{A}$ s rozměry $n \times n$, vektorem pravé strany b s rozměrem $n \times 1$ a vektorem řešení x se stejným rozměrem jako vektor b.

```
1:  for k = 0; k < n; k++ do
2:      if A(k,k) ≠ 0 then                          ▷ kontrola nenulovosti pivota
3:          for i = k; i < n; i++ do
4:              A(k,i) = A(k,i)/A(k,k)              ▷ normování hlavního řádku
5:      for i = k+1; i < n; i++ do
6:          for j = k+1; j < n; j++ do
7:              A(i,j) = A(i,j) − A(i,k) * A(k,j)   ▷ elimiace řádku od pivotního řádku
8:          b(k) = b(k) − A(i,k) * b(k)
9:          A(i,k) = 0
10: for k = n−1; k > −1; k−− do                     ▷ zpětná substituce
11:     x(k) = b(k)
12:     for j = k+1; j < n; j++ do
13:         x(k) = x(k) − x(j) * A(k,j)
```



Obrázek 4.1: k-tý krok sekvenčního GEM, kde zelené pole značí nenulové prvky.

4.2 Paralelní Gaussova eliminační metoda

V této části se zaměříme na algoritmus paralelní verze GEM, který jsme implementovali pomocí CUDA a knihovny TNL. Z toho důvodu budou následující pseudokódy obsahovat prvky programovací architektury CUDA.

Matici $\mathbb{A}$ o rozměrech $n \times n$ lze pomocí elementárních řádkových úprav převést do diagonálního tvaru bez nutností počítání zpětné substituce. Po normalizaci v k-tém kroku algoritmu lze jednoduchým odečtením hlavního k-tého řádku od řádků nad hlavním řádkem dosáhnout vynulování i prvků v k-tém sloupci nad hlavním řádkem. Tímto způsobem lze eliminovat všechny nad-diagonální prvky matice soustavy $\mathbb{A}$ a získat tak řešení soustavy bez nutnosti počítání zpětné substituce. Tento fakt, jak se ukáže níže, je velmi důležitý pro paralelní výpočty na GPU.

Z algoritmu 13 je zřejmé, že dělení pivotního řádku (řádek 4) není nutnou součástí algoritmu, pokud dělení pivotem budeme provádět na ostatních řádcích matice (ř. 7). Této vlastnosti nemá smysl v sekvenčním GEM využívat z důvodu navýšení počtu operací. Při paralelizaci nám ale zmizí jeden z nutných sekvenčních kroků. Dále si můžeme všimnout, že operace na řádcích i prvky v nich jsou na sobě nezávislé, za předpokladu, že se pivotní řádek nemění. Tedy for-cykly na řádcích 5 a 6 jsou plně paralelizovatelné. Dále využijeme možnosti

počítat i s nad-diagonálními prvky, čímž se nesníží složitost algoritmu ale zvýší se nám možnost využití paralelizace a to rozšířením for-cyklu z řádku 5 na rozsah $\langle 0,n)$ bez pivotního řádku k, jak je vidět v algoritmu 4.2. Bez tohoto kroku by algoritmus ztrácel v každém kroku $2n + 2k + 1$ paralelních vláken. Tím, že budeme upravovat zároveň i nad-diagonální prvky dokážeme tuto ztrátu snížit až na $n - 1$ paralelních vláken. Navíc část zpětné substituce, která je těžko paralelizovatelná úplně zmizí. Výsledná matice bude totiž v diagonálním tvaru s nenulovými prvky na diagonále. Pro výpočet řešení tedy stačí na konci programu podělit vektor $\boldsymbol{b}$ diagonálními prvky matice $\mathbb{A}$.

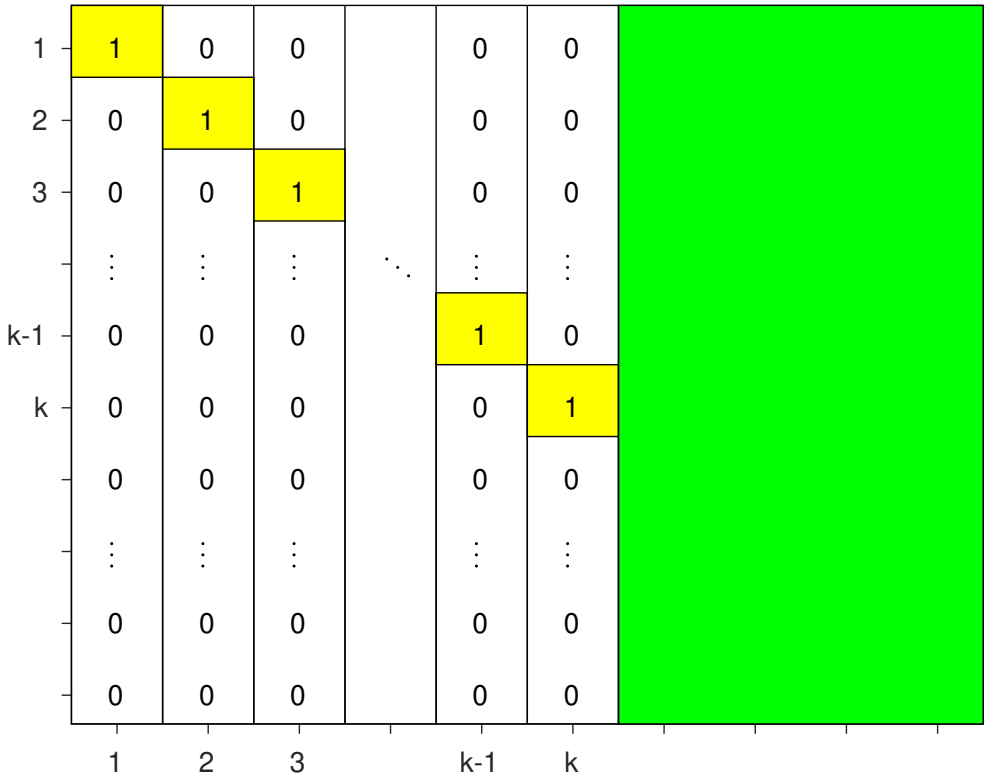
Hlavní část algoritmu v k-tém kroku, která se spouští s počtem bloků stejným jako počet řádků matice $\mathbb{A}$ a počtem vláken stejným jako zbývající sloupce tedy $n - k$ je znázorněna v algoritmu 14. Důležitou součástí je uložení matice v paměti po řádcích, čímž lze alespoň částečně naplnit podmínka sloučených přístupů do paměti, kterou jsme popsali v kapitole Nástroje CUDA.

Algoritmus 14 Paralelní algoritmus Gaussovy Eliminační metody pro matici $\mathbb{A}$ s rozměry $n \times n$, vektorem pravé strany b s rozměrem $n \times 1$ a vektorem řešení x se stejným rozměrem jako vektor b.

```
1:  for k = 0; k < n; k++ do
2:      MainKernel<<< n, n − k >>>(A, b, k)
3:  CalculateResult<<< 1, n >>>(A, b, x)

4:  function ___global___MainKernel(A, b, k)
5:      row = blockIdx.x
6:      col = threadIdx.x
7:      if col > k ∧ row ≠ k ∧ A(k, k) ≠ 0 then
8:          pivot = A(k, k)
9:          firstElemInRow = A(row, k)
10:         if firstElemInRow ≠ 0 then
11:             A(row,col) = A(row,col)−firstElemInRow∗A(k,col)/pivot
12:     if row ≠ k ∧ col== k then
13:         b(row) = b(row) − A(row, k) ∗ b(k)/A(k, k)

14: function ___global___CalculateResult(A, b, x)
15:     row = threadIdx.x
16:     x(row) = b(row)/A(row,row)
```



Obrázek 4.2: k-tý krok paralelního GEM, kde zelené pole značí nenulové prvky.

Samozřejmou součástí tohoto kódu je také částečný pivoting. Hledání maxima v absolutní hodnotě se nám podařilo implementovat pomocí paralelní redukce fungující v registrech GPU. Tato paralelní redukce je založena na funkci ___shfl_down_sync(), která byla přestavena v CUDA 9. Tato funkce umožňuje vláknu sahat do registrů vedlejších vláken stejného warpu. Více o paralelní redukci založenou na této funkci se můžete dočíst v [5]. Kód pro pivoting s využitím `MainKernel()` funkce z algoritmu 14 je znázorněn v algoritmu 15.

Algoritmus 15 Paralelní algoritmus Gaussovy Eliminační metody s pivotingem pro matici $\mathbb{A}$ s rozměry $n \times n$, vektorem pravé strany b o rozměru $n \times 1$ a vektorem řešení x se stejným rozměrem jako vektor b.

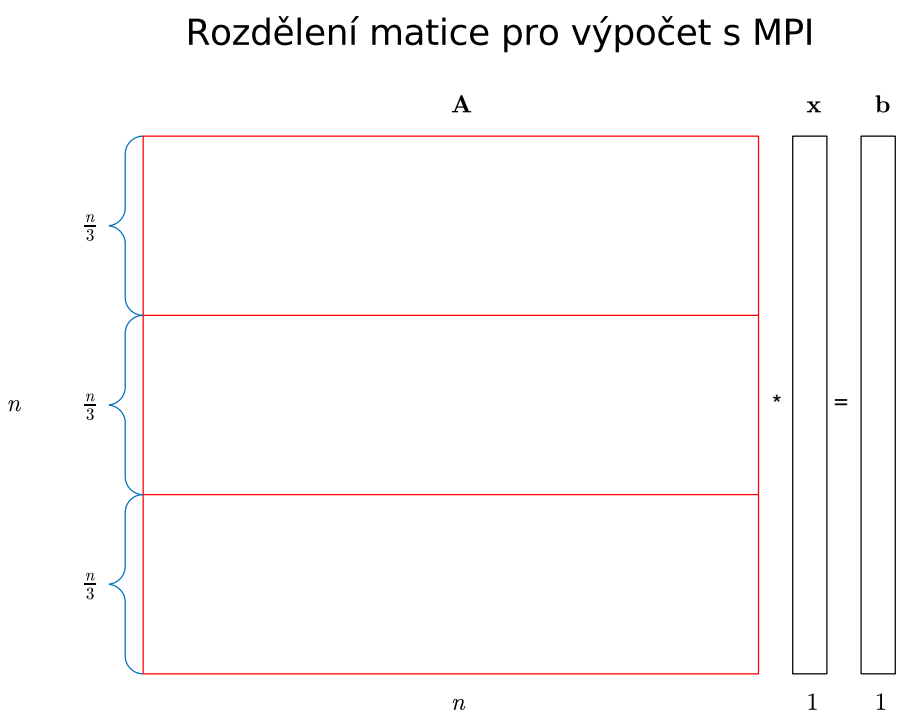
1: **for** $k = 0; \quad k < n; \quad k++$ **do**
2: pos = -1
3: FindRowPivot$\lll 1, n-k \ggg(\mathbb{A}, k$,pos) ▷ Najdi pivotní řádek s paralelní redukcí
4: **if** pos $\neq k$ **then**
5: SwapRows$\lll 1, n-k \ggg(\mathbb{A}, k$,pos) ▷ Vyměň pivotní řádek za k-tý řádek
6: MainKernel$\lll n, n-k \ggg(\mathbb{A}, b, k)$
7: CalculateResult$(\mathbb{A}, b, x)$

4.3 Paralelní Gaussova eliminační metoda s MPI

Důležitou součástí výpočtů je jejich rozšíření na výpočetní klastry, které jsou schopné počítat úlohy, jenž by se na jednom zařízení počítaly obtížně. Zatímco mírně upravený kód z předchozí kapitoly zvládá efektivně úlohy střední velikost (500-8000 řádků matice v závislosti na použitém GPU), pro větší úlohy není vhodný. MPI nám pomůže řešit úlohy, které jsou pro jedno GPU příliš veliké. V této podkapitole si tedy ukážeme implementaci pomocí MPI a to pouze na grafické karty.

Při použití MPI je nutné dbát na omezení týkající se přenosu dat. Přenos dat mezi jednotlivými komponenty výpočetního klastru je obecně pomalý. Z toho důvodu je nutné kopírování dat mezi jednotlivými GPU maximálně omezit.

Návrh vycházející z předchozího algoritmu je jednoduchý. Jedná se o rozdělení matice na vodorovné pásy podle počtu procesů. Toto rozdělení je znázorněno na obrázku 4.3. Při počtu procesů m poté každý proces pracuje s pásem matice $\mathbb{A}$ o velikosti $\frac{n}{m} \times n$, kde zlomek označíme jako $\tilde{n} = \frac{n}{m}$. V každém kroku k algoritmu dostává každý proces s použitím funkce `MPI_Bcast()` celý pivotní řádek o velikosti $1 \times n-k+1$, kde $n-k$ je počet sloupců matice $\mathbb{A}$ s nimiž se bude počítat a jeden prvek vektoru $\boldsymbol{b}$. S tímto řádkem poté každý proces upravuje svoji část matice a to pomocí ekvivalentí řádkové úpravy odečtení násobku povitálního řádku od svých řádků. Tato ekvivalentní řádková úprava již probíhá na GPU. Kód pro jednoduchý GEM s MPI bez pivotingu je znázorněn v algoritmu 16. Pseudokód obsahuje funkce z MPI, které jsme již popsali v kapitole o MPI.



Obr. 4.3: Příklad rozdělení matice a vektoru pravé strany na 3 bloky pro výpočet GEM s MPI

Algoritmus 16 Paralelní algoritmus Gaussovy Eliminační metody s MPI bez pivotingu pro matici $\mathbb{A}$ s rozměry $n \times n$ rozdělenou pomocí m processů na m pásů velikosti $\frac{n}{m} \times n$, vektorem pravé strany b o rozměru $n \times 1$ a vektorem řešení x se stejným rozměrem jako vektor b.

```
1: processID = -1
2: numOfProcesses = 0
3: MPI_Comm_rank( &processID )
4: MPI_Comm_size( &numOfProcesses )
5: k = 0
6: while k < n do
7:     rowPointer = k - ⌊k/ñ⌋ * ñ        ▷ řádek v části matice A odpovídající hlavnímu k-tému řádku
8:     mainRowVec(n − k + 1)                                  ▷ vektor o velikosti n − k + 1
9:     if ⌊k/ñ⌋ == processID then
10:        A.getRow( rowPointer, k, mainRowVec );
11:        mainRowVec[n − k] = b[rowPointer]
12:    MPI_Bcast( mainRow, size, k/ñ)
13:
14:    MainKernelMPI⋘ ñ, n − k ⋙(A, b, mainRowVec, k, rowPointer,
15:                                  processID, numOfProcesses)
16:    k = k + 1
17: CalculateResult(A, b, x)

18: function ___global___MainKernelMPI(A, b, mainRowVec, k, rowPointer,
19:                                    processID, numOfProcesses)
20:    row = blockIdx.x
21:    col = threadIdx.x
22:    if col > k ∧ mainRowVec[0] ≠ 0 ∧ rowPointer ≠ row then
23:        pivot = mainRowVec(0)
24:        firstElemInRow = A(row, k)
25:        if firstElemInRow ≠ 0 then
26:            A(row,col) = A(row,col)−firstElemInRow∗mainRowVec[col-k])/pivot
27:    if row + processID ∗ñ ≠ k ∧ col== k ∧ row == 0 ∧ row < ñ ∧ mainRowVec[0] ≠ 0 then
28:        b(row) = b(row) − A(row, k)∗mainRowVec[n − k]/mainRowVec[0]
```

Funkce `MainKernel` se musela z verze bez MPI rozšířit. Hlavní řádek již nemusí být obsažen v matici, přičemž každý proces s ním pracuje jako s externím vektorem mainRowVec. Zajímavostí algoritmu je také nutnost určení hlavního procesu, který tento mainRowVec plní hodnotami a následně rozesílá ostatním procesům pomocí funkce `MPI_Bcast()`. Funkce `CalculateResult()` vypadá také obdobně jako stejnojmenná funkce z algoritmu bez MPI. Rozšířením v této funkci je složení výsledku ze všech procesů do nultého procesu. Je tedy opět nutné identifikovat který proces má kterou část výsledku ve výsledném vektoru $\boldsymbol{b}$, což odpovídá pásům matice $\mathbb{A}$, a poté poslání těchto čísel nultému procesu, který je složí do celkového vektoru $\boldsymbol{x}$ o velikosti n.

Nyní je zřejmé, že již tento přístup vyžaduje velké množství komunikace mezi jednotlivými procesy. V každém kroku k algoritmu se totiž rozesílá hlavní řádek ostatním procesů, to znamená n-krát `MPI_Bcast()` a v něm musí každé vlákno přijmout hodnoty z hlavního procesu, který naopak hodnoty rozesílá. Tento kód ale není numericky stabilní tak, jako verze bez MPI, chybí totiž částečný pivoting.

Před tím než popíšeme kód pro pivoting, pojďme nejprve rozmyslet, co vše bude nutné v pivotingu udělat. Nejprve bude nutné nalézt pivotní řádek, který se může ale i nemusí nacházet v hlavním procesu, tedy procesu který reálně obsahuje k-tý řádek matice. Navíc ke zjištění, kde se pivotní řádek nachází, bude nutné nalézt lokální maximum v absolutní hodnotě v každém z pásů matice $\mathbb{A}$ a následné rozeslání těchto lokálních maxim všem procesům. Každý proces poté vyhodnotí v jakém procesu se nalézá globální maximum a na jakém řádku v pásu daného procesu se řádek nachází. Pokud se jako proces s maximem ukáže hlavní proces, pak kód pivotingu přejde na verzi kódu bez MPI, tedy přehodí se dva řádky pomocí funkce `SwapRows()` a načte se mainRowVec, který se rozešle ostatním procesům. Pokud ale maximum neleží v hlavním procesu, pak musí daný proces prohodit řádek s hlavním procesem a následně bude moci hlavní proces vytvořit mainRowVec a rozeslat ho ostatním procesům. Tento přístup lze pomocí MPI ještě zjednodušit, protože když už se posílá řádek pro výměnu je vhodné jej rozeslat i ostatním procesům, které mohou začít s výpočty. Tento popsaný kód je znázorněn níže v algoritmech 17 a 18.

Algoritmus 17 Paralelní algoritmus Gaussovy Eliminační metody s MPI s pivotingu pro matici $\mathbb{A}$ s rozměry $n \times n$ rozdělenou pomocí m processů na m pásů velikosti $\frac{n}{m} \times n$, vektorem pravé strany b o rozměru $n \times 1$ a vektorem řešení x se stejným rozměrem jako vektor b.

```
1: processID = -1
2: numOfProcesses = 0
3: MPI_Comm_rank( &processID )
4: MPI_Comm_size( &numOfProcesses )
5: k = 0
6: while k < n do
7:     rowPointer = k - ⌊k/ñ⌋ * ñ        ▷ řádek v části matice 𝔸 odpovídající hlavnímu k-tému řádku
8:     [MAX, POS, processMax ]=FINDMAX( 𝔸, processID, numOfProcesses)
9:     mainRowVec(n − k + 1)                                  ▷ vektor o velikosti n − k + 1
10:    if processMax ≠ ⌊k/ñ⌋ then
11:        if processID == processMax then
12:            𝔸.getRow( rowPointer, k, mainRowVec );
13:            mainRowVec[n − k] = b[k]
14:    else if ⌊k/ñ⌋ == processID then
15:        if POS ≠ rowPointer then
16:            SWAPROWS⋘ 1, n − k ⋙(𝔸, k,POS)
17:        𝔸.getRow( rowPointer, k, mainRowVec )
18:        mainRowVec[n − k] = b[rowPointer]
19:    MPI_Bcast( mainRow, size, k/ñ)
20:    if processMax ≠ ⌊k/ñ⌋ then
21:        mainRowSwap(n − k + 1)     ▷ vektor o velikosti n − k + 1 pro výměnu pivotního řádku
22:        if processID == processMax then
23:            MPI_Recv( mainRowSwap, n − k + 1, ⌊k/ñ⌋,0)
24:            𝔸.setRow(POS,k,mainRowSwap)
25:            b[POS] = mainRowSwap(n − k)
26:        else if processID == ⌊k/ñ⌋ then
27:            𝔸.getRow( rowPointer, k, mainRowSwap )
28:            mainRowSwap[n − k] = b[rowPointer]
29:            MPI_Send(mainRowSwap, n − k + 1, processMax, 0)
30:            𝔸.setRow(rowPointer,k,mainRowVec)
31:            b[rowPointer] = mainRowVec(n − k)
32:    MAINKERNELMPI⋘ ñ, n − k ⋙(𝔸, b, mainRowVec, k, rowPointer,
33:                                  processID, numOfProcesses)
34:    k = k + 1
35: CALCULATERESULT(𝔸, b, x)
```

Algoritmus 18 Funkce findMax() z algoritmu 17.

```
1: function [MAX, POS, processMax]=FINDMAX( 𝔸, processID, numOfProcesses)
2:     fromRow = 0
3:     if ⌊k/ñ⌋ > processID then
4:         fromRow = ñ
5:     else if ⌊k/ñ⌋ = processID then
6:         fromRow = rowPointer
7:     max = 0
8:     pos = -1
9:     if fromRow ≠ ñ then
10:        FINDROWPIVOT⋘ 1, ñ − fromRow ⋙(A, k,pos,max)
11:    data[0] = max; data[1] = pos
12:    dataRecv[2∗numOfProcesses]                    ▷ alokace paměti pro příchozí data
13:    MPI_Allgather( data, 2, dataRecv, 2)
14:    for i = 0;   i < 2∗numOfProcesses;   i=i+2 do
15:        if dataRecv[i+1] ≠ −1∧ MAX < dataRecv[i]  then
16:            MAX = dataRecv[i]
17:            POS = dataRecv[i+1]
18:            processMax = ⌊i/2⌋
```

Kapitola 5

Výsledky Gaussovy Eliminační metody

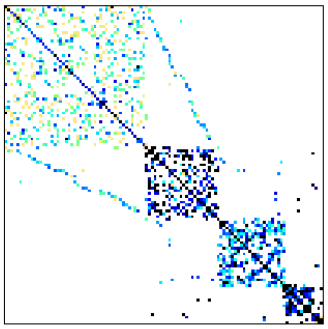
V této podkapitole si ukážeme výsledky GEM implementované na CPU, GPU bez MPI a GPU s MPI. Všechny implementace byly představeny v této kapitole.

Výpočty byly prováděny na školním počítači Teslon, který je vybaven 10-jádrovým procesorem Intel® Xeon® E5-2640 v4 se základní frekvencí 2.4GHz a cache pamětí 20 MB. Dále se na Teslonu nachází 2 grafické akcelerátory NVIDIA® Tesla P100-SXM2 globální pamětí o velikosti 16GB pro každý. Na Teslonu je nainstalován GCC překladač verze 8.2.0, CUDA 10.2 a a MPI 3.1.3.

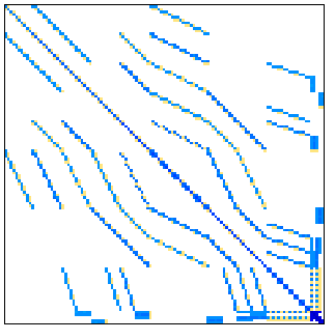
Všechny výpočty se byly provedeny na testovacích maticích z [14], jejichž parametry jsou znázorněny v tabulce 5.1 a jejich rozložení prvků na obrázcích 5.1 - 5.6. Matice vybrané pro výpočet by měly být různých velikostí i počtu nenulových prvků. Pro CPU kód se použilo jedno jádro procesoru Intel® Xeon® E5-2640 v4 a pro GPU kód se použila jeden grafický akcelerátor NVIDIA® Tesla P100-SXM2. MPI výpočty se poté prováděly pouze pro GPU kód za použití obou grafických akcelerátorů NVIDIA® Tesla P100-SXM2.

Samotný výpočet GEM pro soustavu rovnic $\mathbb{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ se poté prováděl s pravou stranou $\boldsymbol{b}$ rovnou součtu prvků v řádcích matice $\mathbb{A}$. Tento přístup nám umožnil ruční počítání výsledného vektoru $\boldsymbol{x}$, který by se měl rovnat vektoru samých jedniček. Z toho důvodu se ve výsledcích objeví L_2 norma definovaná vztahem (5.1), kde $\boldsymbol{x}$ označuje správné řešení soustavy $\mathbb{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ a $\hat{\boldsymbol{x}}$ označuje výsledek GEM.

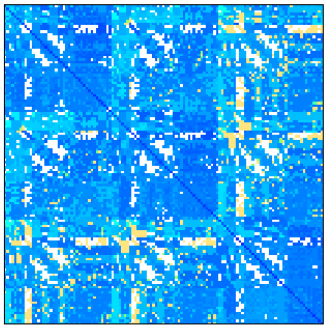
$$L_2 = \sum_{i=0}^{n} (x_i - \hat{x}_i)^2, \quad \boldsymbol{x} \in \mathbb{R}^n, \hat{\boldsymbol{x}} \in \mathbb{R}^n. \tag{5.1}$$



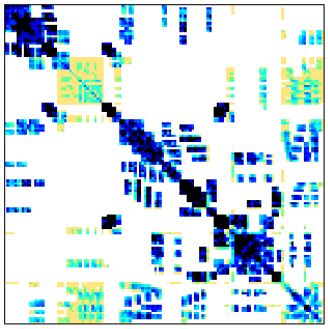
Obrázek 5.1: Matice 662_bus o 662 řádcích.



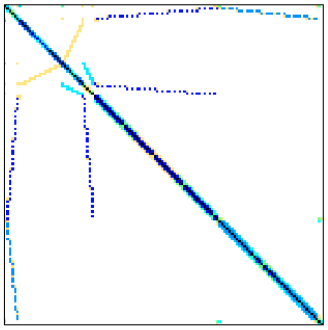
Obrázek 5.2: Matice msc01050 o 1050 řádcích.



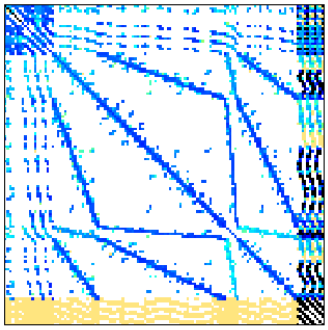
Obrázek 5.3: Matice comsol o 1500 řádcích.



Obrázek 5.4: Matice heart1 o 3557 řádcích.



Obrázek 5.5: Matice s3rmt3m3 o 5357 řádcích.



Obrázek 5.6: Matice fp o 7548 řádcích.

Tabulka 5.1: Tabulka parametrů matic použitých pro výpočet GEM.

matice	#řádků×#sloupců	#nenul. prvků
662_bus	662×662	1 568
msc01050	1050×1050	15 103
comsol	1500×1500	97 645
losmoc	1500×1500	97 645
heart1	3557×3557	1 387 773
s3rmt3m3	5357×5357	207 123
fp	7548×7548	848 553

V tabulce 5.1 si lze všimnout dvou matic se stejnými parametry. Matice losmoc byla námi vytvořena za účelem vyzkoušení maximálního pivotingu. Jedná se tedy o matici comsol, u které jsme experimentálně zjistili, že pivoting využije při výpočtu dvakrát a to pouze pro sousední řádky. Matice losmoc tedy vznikla z matice comsol u níž jsme přesunuli poslední řádek na nultý a zbylé řádky jsme o posunuli o jeden. U matice losmoc se tedy využívá pivotingu v každém kroku výpočtu.

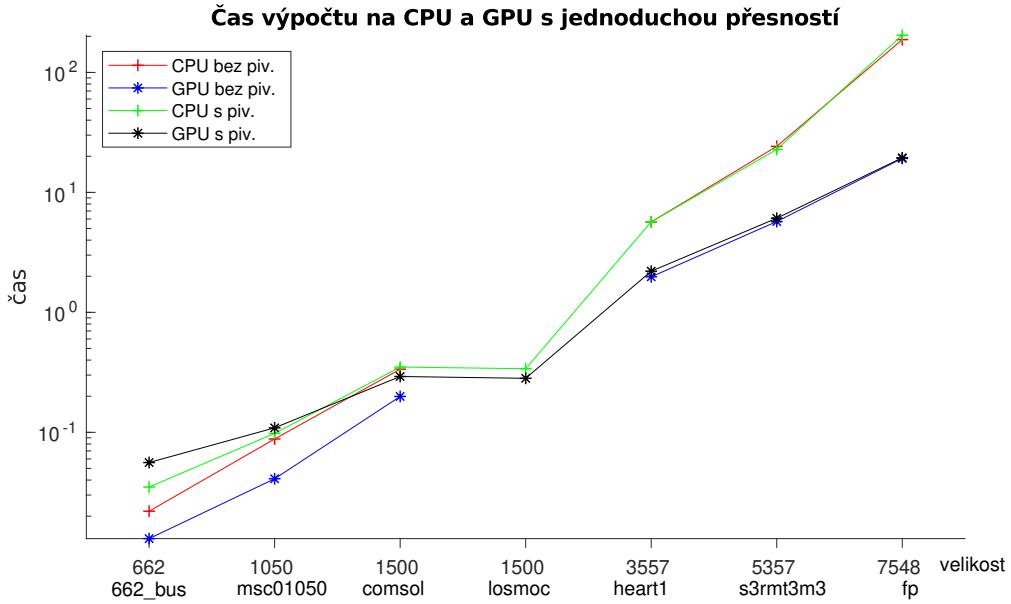
5.1 GEM s jednoduchou přesností

V této kapitole ukážeme výsledky dosažené na testovacích maticích z [14] s parametry z tabulky 5.1.

V tabulce 5.2 jsou vyobrazeny časy výpočtů a chyby oproti správnému řešení pro dané testovací matice na CPU a GPU bez pivotingu a na CPU a GPU s pivotingem s jednoduchou přesností. Lze si všimnout, že chyby řešení jsou u dané matice přibližně stejné pro všechny kombinace výpočtu. Z tabulky 5.4, kde jsou vyobrazeny obdobné výpočty pouze s dvojitou přesností, jsou chyby výpočtu oproti správnému řešení minimální či nulové. Z toho plyne, že algoritmy počítají správná řešení a chyby v 5.2 jsou způsobeny pouze nedostatečnou numerickou přesností. Dále si lze všimnout, že již na nejmenší matici 662_bus trvá výpočet GEM kratší dobu na GPU než na CPU bez pivotingu, naopak s pivotingem je lepší CPU než GPU. Hledání pivotálního řádku, i přes fakt, že v matici 662_bus pivoting nehraje roli, protože jsou maxima na diagonále v každém kroku výpočtu, zvyšuje výpočetní čas. Výsledky z tabulky 5.2 jsou znázorněny i na obrázku 5.7, kde na ose x jsou jednotlivé matice a na ose y je logaritmická hodnota času výpočtu.

Tabulka 5.2: Porovnání časů výpočtů GEM s jednoduchou přesností na CPU a GPU bez pivotingu a s pivotingem.

	GEM bez pivotingu				GEM s pivotingem			
	CPU	L_2 error	GPU	L_2 error	CPU	L_2 error	GPU	L_2 error
662_bus	0,022	0,009	0,015	0,007	0,035	0,009	0,058	0,007
msc01050	0,088	30,964	0,044	30,965	0,098	30,961	0,098	30,963
comsol	0,336	3,435	0,182	3,394	0,350	3,433	0,272	3,390
losmoc	-	-	-	-	0,339	3,837	0,275	3,825
heart1	5,675	0,042	1,850	0,041	5,687	33,835	2,067	39,583
s3rmt3m3	24,189	49,096	5,467	82,098	22,828	727,530	5,794	2578,237
fp	187,727	2,746	18,085	2,712	204,146	2,719	18,417	2,914



Obrázek 5.7: Výsledky GEM metody pro testovací matice s jednoduchou přesností.

V tabulce 5.3 je znázorněno naměřené urychlení mezi CPU×GPU bez pivotingu a mezi CPU×GPU s pivotingem. Tyto hodnoty opět potvrzují závěry vyvozené z tabulky 5.2. Hodnoty ukazují, že GPU má na větších maticích kratší výpočetní čas než CPU.

Tabulka 5.3: Tabulka urychlení CPU×GPU pro GEM v jednoduché přesnosti pro časy bez pivotingu a pro časy s pivotingem.

	ur. CPU×GPU GEM bez pivotingu	ur. CPU×GPU GEM s pivotingem
662_bus	1,47	0,60
msc01050	2,00	1,00
comsol	1,85	1,29
losmoc	-	1,23
heart1	3,07	2,75
s3rmt3m3	4,42	3,94
fp	10,28	11,08

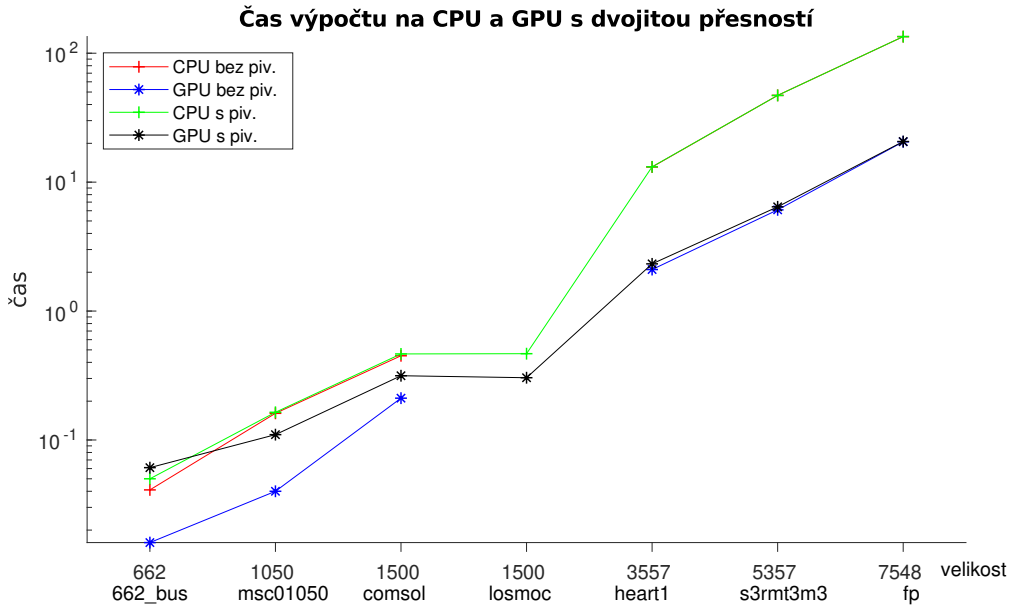
5.2 GEM s dvojitou přesností

V této kapitole ukážeme naměřený čas výpočtů a chybu oproti přesnému řešení pro GEM s dvojitou přesností. Dále napočítáme urychlení CPU×GPU bez pivotingu a CPU×GPU s pivotingem.

V tabulce 5.4 jsou znázorněny časy a chyby výpočtů GEM s dvojitou přesností. Hodnoty ukazují pro většinu testovacích matic nulovou chybu. Tento fakt svědčí o správnosti výsledku soustavy lineárních rovnic. Časy výpočtu z tabulky 5.4 jsou znázorněny i v obrázku 5.8.

Tabulka 5.4: Porovnání časů výpočtů GEM s dvojitou přesností na CPU a GPU bez pivotingu a s pivotingem.

	GEM bez pivotingu				GEM s pivotingem			
	CPU	L_2 error	GPU	L_2 error	CPU	L_2 error	GPU	L_2 error
662_bus	0,041	0,000	0,018	0,000	0,050	0,000	0,058	0,000
msc01050	0,161	0,283	0,043	0,286	0,164	0,218	0,106	0,215
comsol	0,450	0,000	0,214	0,000	0,465	0,000	0,299	0,000
losmoc	-	-	-	-	0,467	0,000	0,305	0,000
heart1	13,105	0,000	2,128	0,000	13,147	0,000	2,357	0,000
s3rmt3m3	47,089	0,000	6,203	0,000	47,066	0,000	6,563	0,000
fp	134,485	0,000	21,073	0,000	134,573	0,000	21,187	0,000



Obrázek 5.8: Výsledky GEM pro testovací matice s dvojitou přesností.

Z obrázku 5.8 je zřejmé, že GPU potřebuje pro výpočet GEM kratší čas než CPU. Na větší maticí se dosahuje až 7 násobného urychlení. Tento fakt je opět znázorněn v tabulce urychlení 5.5.

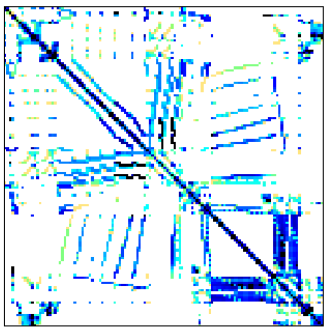
Tabulka 5.5: Tabulka urychlení CPU×GPU pro GEM s dvojitou přesností pro časy bez pivotingu a pro časy s pivotingem.

	ur. CPU×GPU GEM bez pivotingu	ur. CPU×GPU GEM s pivotingem
662_bus	2,28	0,86
msc01050	3,74	1,55
comsol	2,10	1,56
losmoc	-	1,53
heart1	6,16	5,58
s3rmt3m3	7,59	7,17
fp	6,38	6,35

5.3 MPI GEM

V této kapitole představíme výsledky pro Gaussovu Eliminační metodu s MPI. Algoritmus představen v předchozí kapitole jsme spustili na výpočetním klastru Teslon pro 1 a 2 procesy pro testovací matice z 5.1 a jednu větší matici ship_001 s 34 920 jejíž náhled je zobrazen na obrázku 5.9.

Výsledky pro výpočty provedené s jednoduchou přesností jsou v tabulce 5.6. Zajímavostí je rychlejší výpočet pro větší matice pro jeden proces MPI než pro kód bez MPI.



Obrázek 5.9: Matice ship_001 o 34 920 řádcích.

Tabulka 5.6: Porovnání časů výpočtů GEM s jednoduchou přesností na GPU s MPI pro 1 a 2 procesy bez pivotingu a s pivotingem.

	GEM bez pivotingu		GEM s pivotingem	
	1×GPU	2×GPU	1×GPU	2×GPU
662_bus	0,056	0,084	0,132	0,148
msc01050	0,100	0,148	0,212	0,240
comsol	0,232	0,240	0,374	0,386
losmoc	-	-	0,398	0,394
heart1	1,542	1,260	1,930	1,632
s3rmt3m3	4,178	3,314	4,834	3,922
fp	13,358	8,280	14,156	9,016
ship_001	1121,290	647,978	1125,806	651,414

V tabulce 5.7 je znázorněno naměřené urychlení pro výpočetní časy GEM s jednoduchou přesností. Zajímavostí je, že již 2 procesy mají určité urychlení oproti 1 procesu. Důvodem může být rozdělení výpočtů na více grafických akcelerátorů a tím lze lépe využít jejich výpočetní výkon.

Tabulka 5.7: Tabulka urychlení GEM s jednoduchou přesností na GPU s MPI 1×2 procesy bez pivotingu a s pivotingem.

	GEM bez pivotingu 1×2 GPU	GEM s pivotingem 1×2 GPU
662_bus	0,67	0,89
msc01050	0,68	0,88
comsol	0,97	0,97
losmoc	-	1,01
heart1	1,22	1,18
s3rmt3m3	1,26	1,23
fp	1,61	1,57
ship_001	1,73	1,73

V tabulce 5.8 jsou znázorněny časy výpočtů GEM s dvojitou přesností. Napočítané urychlení je poté znázorněno v tabulce 5.9. Lze si opět všimnout, že větší matice mají kratší výpočet pro 2 procesy než pro 1 proces.

Tabulka 5.8: Porovnání časů výpočtů GEM s dvojitou přesností na GPU s MPI pro 1 a 2 procesy bez pivotingu a s pivotingem.

	GEM bez pivotingu		GEM s pivotingem	
	1×GPU	2×GPU	1×GPU	2×GPU
662_bus	0,052	0,082	0,120	0,140
msc01050	0,100	0,152	0,210	0,254
comsol	0,260	0,288	0,412	0,438
losmoc	-	-	0,428	0,444
heart1	1,810	1,668	2,204	2,042
s3rmt3m3	4,838	4,306	5,496	4,916
fp	16,288	10,962	17,020	11,630
ship_001	1350,311	813,344	1354,717	816,638

Tabulka 5.9: Tabulka urychlení GEM s dvojitou přesností na GPU s MPI 1×2 procesy bez pivotingu a s pivotingem.

	GEM bez pivotingu 1×2 GPU	GEM s pivotingem 1×2 GPU
662_bus	0,63	0,86
msc01050	0,66	0,83
comsol	0,90	0,94
losmoc	-	0,96
heart1	1,09	1,08
s3rmt3m3	1,12	1,12
fp	1,49	1,46
ship_001	1,66	1,66

Závěr

V této práci jsme se seznámili s programovacími nástroji nVidia CUDA pro grafické akcelerátory značky nVidia, MPI pro počítačové architektury s distribuovanou pamětí, OpenMP pro počítačové architektury se sdílenou pamětí a knihovnou TNL, pomocí níž se implementace prováděly.

Představili jsme klasický Ellpack formát pro uložení řídké matice a jeho symetrickou verzi pro řídké symetrické matice. Spolu s formáty jsme představili implementace násobení matice s vektory a to, jak na CPU, tak na GPU.

Následně jsme prezentovali výsledky násobení vybraných řídkých symetrických matic s vektory. Výsledky, které jsme prezentovali pomocí časů výpočtů a urychlení jednotlivých algoritmů ukazují, že pro CPU je nejméně časově náročný Ellpack formát a pro GPU jsou nejméně časově náročné Ellpack formát a symetrický Ellpack formát s použitím atomických instrukcí, který navíc, díky symetričnosti, šetří místo v paměti. Naopak sofistikovaný algoritmus speciálního maticového obarvení pro symetrický Ellpack formát se ukázal jako neefektivní, jak pro CPU, tak pro GPU.

Pro řešení soustav lineárních rovnic pomocí Gaussovy Eliminační metody jsme nejprve ukázali jednoduchý sekvenční kód pro CPU, který jsme následně rozšířili pro výpočty na GPU a pomocí MPI pro výpočty na výpočetních klastrech pro více GPU.

Výsledky Gaussovy Eliminační metody ukazují správnost použité metody pro výpočet soustavy lineárních rovnic zadané maticí. Výpočetní čas je pro větší matice až několikanásobně nižší na GPU než na CPU a to pro výpočet bez pivotingu i s pivotingem. Výsledky pro Gaussovu Eliminační metodu s MPI ukazují vhodnost použití této metody pouze pro matice náročnější na paměť než je globální paměť jednoho grafického akcelerátoru, nikoliv menší.

Literatura

[1] OpenMP Architecture Review Board: *OpenMP Application Programming Interface*. Last update November, 2018 [cit. 2019-26-08] Dostupné z: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>

[2] T. Oberhuber, J. Klinkovský, V. Žabka, V. Klementa, R. Fučík *TNL: Framework for rapid development of numerical solvers for modern parallel architectures*. odeslano do ACM TOMS

[3] D. B. Kirk, W. W. Hwu: *Programming Massively Parallel Processors*. Third Edition: A Hands-on Approach. Morgan Kaufmann, 2016.

[4] NVIDIA Developer Zone: *CUDA C Programming Guide* [online]. Last updated May 15, 2018 [cit.2018-07-06] Dostupné z: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

[5] J. Cheng, M. Grossman, T. McKercher, *Professional CUDA C Programming*. First Edition: Worx, 2014, ISBN 978-1118739327

[6] *Optimizing Parallel Reduction in CUDA*. Nvidia [online]. [cit.2019-15-08]. Dostupné z: <https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf>

[7] W. Gropp, E. Lusk, A. Skjellum: *Using MPI*. Second Edition: Portable Parallel Programming with the Message-Passing Interface, The MIT Press, 1999, ISBN 0-262-57134-3

[8] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Second Edition: Society for Industrial and Applied Mathematics, 2003, ISBN 978-0898715347

[9] N. Bell, M. Garland: *Efficient Sparse Matrix-Vector Multiplication on CUDA* [online]. 11. 12. 2008. [online] Dostupné z: <https://www.nvidia.com/docs/IO/66889/nvr-2008-004.pdf>

[10] T. Oberhuber, A. Suzuki, J. Vacata: *New Row-grouped CSR format forstoring sparse matrices on GPUwith implementation in CUDA*. Acta Technica, [cit.2019-06-06] Dostupné z: <http://geraldine.fjfi.cvut.cz/ oberhuber/data/vyzkum/publikace/11-oberhuber-suzuki-vacata-rgcsr-format-in-cuda.pdf>

[11] D. Langr and P. Tvrdík, *Evaluation Criteria for Sparse Matrix Storage Formats*, in IEEE Transactions on Parallel and Distributed Systems 27, 2, 428-440, 2016

[12] Bareiss, E. H. *Multistep Integer-Preserving Gaussian Elimination.* Argonne National Laboratory Report ANL-7213, May 1966

[13] X. Xia ,J.C. Lee: *CUDA Linear Equations Solver Based on Modified Gaussian Elimination*, Arizona University. [online] [cit 2020-20-3] Dostupné z: <https://xinggaox.wordpress.com/2010/05/25/cuda-linear-equations-solver-based-on-modified-gaussian-elimination/>

[14] the University of Florida Sparse Matrix Collection: *The SuiteSparse Matrix Collection* [online] [cit. 2019-29-08] Dostupné z: <https://sparse.tamu.edu/>