České vysoké učení technické v Praze
Fakulta elektrotechnická

Bakalářská práce

Paralelizace revidované simplexové metody na GPU

Praha, 2014 **Autor: Jakub Hvězda**

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Jakub Hvězda**

Studijní program: Kybernetika a robotika
Obor: Systémy a řízení

Název tématu: **Paralelizace revidované simplexové metody na GPU**

Pokyny pro vypracování:

1. Seznamte se s problematikou a prostudujte literaturu dodanou vedoucím práce.
2. Navrhněte paralelní revidovaný simplexový algoritmus k řešení lineárních úloh na grafické kartě.
3. Implementujte navrženou metodu na grafické kartě GeForce GTX Titan.
4. Vyhodnoťte přínos paralelizace na vhodně zvolených dense a sparse příkladech.

Seznam odborné literatury:

[1] D. G. Spampinato and A. C. Elster, "Linear optimization on modern GPUs", in Parallel & Distributed Processing, IPDPS 2009, International Symposium on. IEEE, 2009, pp. 1–8.
[2] J. Bieling, P. Peschlow, and P. Martini, "An efficient gpu implementation of the revised simplex method", in Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on. IEEE, 2010, pp. 1–8.
[3] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA", NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.
[4] NVIDIA CUDA Programming Guide 5.0, NVIDIA, 2014. http://docs.nvidia.com/cuda/cuda-c-programming-guide/

Vedoucí: Ing. Jan Zábojník

Platnost zadání: do konce letního semestru 2014/2015

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 16. 1. 2014

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne _____ _____

podpis

Poděkování

Rád bych poděkoval především vedoucímu bakalářské práce Ing. Janu Zábojníkovi za pomoc, trpělivost a hlavně rady potřebné pro vypracování této práce.

Abstrakt

Cílem této bakalářské práce je implementace paralelního simplexového algoritmu pracujícího s řídkými(sparse) maticemi pro platformu Nvidia CUDA. Práce obsahuje popis platformy CUDA spolu s jejím programovacím modelem. Dále se práce věnuje popisu simplexového algoritmu spolu s jeho revidovanou verzí. Je uveden postup implementace a její efektivita je vyhodnocena na skupině sparse lineárních optimalizačních problémů.

Abstract

Goal of this bachelor thesis is implementation of parallel simplex algorithm that works with sparse matrices for Nvidia CUDA platform. Work includes description of CUDA platform with its programming model. This work also includes the description of simplex algorithm with its revised version. Subsequently the work describes implementation procedure and its effectivity is evaluated on a set of linear optimization problems.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Cílem této bakalářské práce je popis a implementace simplexové metody, přesněji její revidované verze, která byla zvolena pro svoji úspornost na počet operací, které jsou potřeba při každé iteraci algoritmu. Revidovaná simplexová metoda byla vybrána, neboť efektivně pracuje s řídkými(sparse) maticemi, které jsou pro implementaci využívány z důvodu jejich paměťové nenáročnosti v porovnání s hustými(dense) maticemi. Celý algoritmus je implementován na platformě Nvidia CUDA a testován na grafických kartách(dále jen GPU) Nvidia GeForce GTX570 a Nvidia GeForce GTX TITAN.

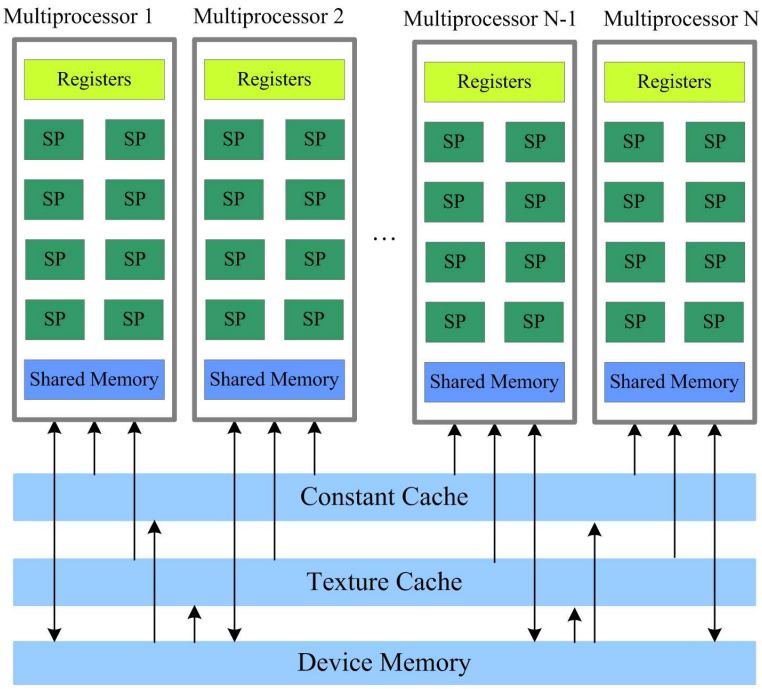
Práce je strukturována do pěti kapitol. První kapitola se věnuje popisu platformy CUDA, jejimu obecnému popisu a programovacím jazykům, které podporuje. Tato kapitola také pojednává o paměťové hierarchii NVIDIA CUDA, kde uvádí jednotlivé paměťové úrovně spolu s jejich stručným popisem. Dále je v této kapitole uveden popis CUDA knihoven, které byly použity pro implementaci. Závěr této kapitoly je věnován programovacímu modelu této platformy, přesněji vláknové hierarchii. Je vysvětleno, jak jednotlivá vlákna dokáží identifikovat svoji pozici v pracovní mřížce, aby nedocházelo ke kolizím při přístupu k datům. Druhá kapitola se věnuje popisu lineárního programování (dále jen LP). Na začátku této kapitoly je vysvětlen pojem lineárního programování, jaké typy úloh lze LP formulovat a jaké výsledky tyto úlohy mohou mít. V další části kapitoly jsou uvedeny obecné formy LP úloh a detailní popis kanonické formy, jež je vyžadována simplexovým algoritmem. Na závěr druhé kapitoly je vysvětlen princip jedno a dvoufázového simplexového algoritmu spolu s jejich jednotlivými kroky. Úvod třetí kapitoly je věnován srovnání formátů sparse matic. Druhá část této kapitoly je věnována revidované verzi simplexového algoritmu, jehož jednotlivé kroky jsou zde podrobně uvedeny. Čtvrtá kapitola je zasvěcena samotné implementaci, konkrétně: formátu vstupního souboru, zpracování načtených dat a jejich převod do kanonického

tvaru, jednotlivým krokům algoritmu a interpretaci výstupu. V páté kapitole je otestována základní a rozšířená verze implementovaného paralelního algoritmu na standardních testovacích LP problémech. V závěru jsou shrnuty dosažené výsledky a problémy, které bylo nutné v rámci této práce vyřešit.

Kapitola 2

NVIDIA CUDA

CUDA [3] je paralelní výpočetní platforma a také programovací model poprvé představený společností NVIDIA v roce 2006. Tato platforma umožňuje využití výpočetní síly grafických karet(GPU) pro klasické výpočty. Platforma CUDA podporuje řadu programovacích jazyků jako například C,C++, Java, Python. Pro další popis je použit jazyk CUDA C++, který se liší o jazyka C++ přidáním nových device funkcí a klíčových slov.



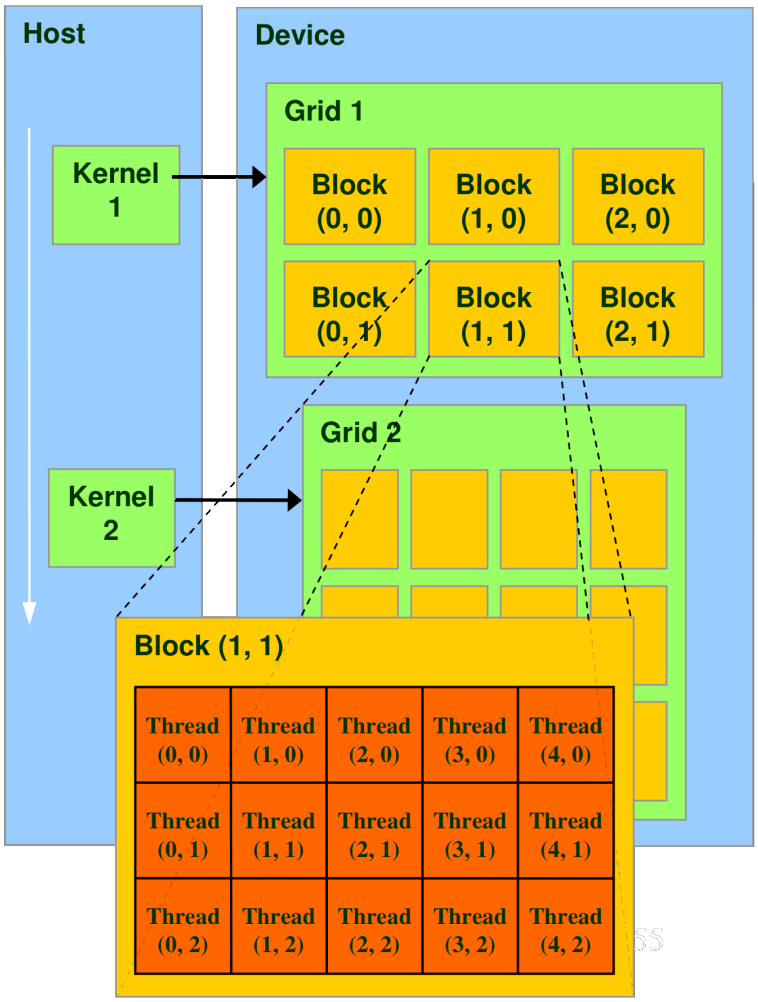
Obrázek 2.1: CUDA hardwarový model [1]

2.1 Streaming multiprocesory

Na obrázku 2.1 je uvedena typická architektura GPU. Hlavní výpočetní jednotkou GPU jsou tzv. Streaming multiprocesory, které jsou schopny plánovat a řídit tisíce paralelních výpočetních vláken. Každý multiprocesor má typicky 8, 32 nebo 48 streaming procesorů, na kterých jsou spouštěna jednotlivá vlákna. Tyto procesory jsou řádově pomalejší, než klasické jádro CPU. Streaming multiprocesory mají alespoň jeden scheduler, který zajišťuje přiřazení spouštění paralelních vláken ve skupinách po 32, tzv. warpech.

2.2 Paměťová hierarchie

Na obrázku 2.1 je uvedena typická paměťová hierarchie GPU. Paměťový prostor na platformě CUDA je rozdělen na 2 hlavní adresní prostory a to na host část, klasickou paměť hostujícího počítače a část device, paměti grafické karty. V důsledku tohoto rozdělení lze používat ukazatele na data na GPU(alokované funkcí *cudaMalloc()*) pouze v části kódu, určeném pro GPU výpočet a ukazatele do host paměti pouze v kódu určeném pro CPU. Data lze mezi prostorem host a device kopírovat pomocí funkce *cudaMemcpy()*. Takto zkopírovaná data jsou přesunuta do globální paměti(na obrázku označené jako device memory). Tato paměť je největší pamětí na GPU, ale také nejpomalejší. Tato nevýhoda se dá zmírnit využitím tzv. coalesced technik. Jelikož přístupy do globální paměti grafické karty jsou relativně pomalé, jsou při každém přístupu načteny i přilehlé adresy. Coalesced techniky spočívají ve využití dat i z těchto přilehlých adres. Příkladem například čtení sloupce v 2D poli. V klasickém případě je 2D pole uloženo po řádcích, čili v případě čtení jednoho prvku jsou načteny i přilehlé prvky z tohoto řádku. Při použití coalesced přístupů do paměti je vhodné pracovat s transponovaným polem, protože jsou načítány velké části řádků, které je možné využít. Dále GPU nabízí také paměť lokální a sdílenou. Lokální paměť je viditelná a využitelná jen pro ukládání lokálních proměnných konkrétního vlákna. Sdílená paměť je sdílená všemi vlákny v jednom bloku. Skrze tuto paměť se mohou vlákna v rámci jednoho bloku omezeně synchronizovat.



Obrázek 2.2: Vláknová hierarchie [2]

2.3 Vláknová hierarchie

Vláknová hierarchie je uvedena na obrázku 2.2. Funkce spouštěné na GPU se nazývají Kernely. Tyto kernely se spouštění na daném počtu vláken, která jsou seskupena do bloků, jež tvoří pracovní mřížku. Každé individuální vlákno zná svoji pozici jak vůči mřížce(*threadIdx.x/y/z*), tak i pozici bloku ve kterém se nachází vůči pracovní mřížce(*blockIdx.x/y/z*). Dále také zná velikost bloku(*blockDim.x/y/z*) a velikost pracovní mřížky(*gridDim.x/y/z*). Jelikož je každé vlákno tímto způsobem jednoznačně schopné určit svojí pozici, nedochází ke konfliktu při přístupu k datům.

Kapitola 3

Lineární programování a Simplexová metoda

Simplexový algoritmus byl poprvé představen Georgem Dantzigem v roce 1951. Jako podklad pro tuto kapitolu byla použita skripta Tomáše Wernera [4].

3.1 Úloha lineárního programování(LP)

Úlohou lineárního programování je minimalizace či maximalizace lineární (účelové) funkce za podmínek lineárních rovností a nerovností. Množina řešení je v případě LP tvořena konvexním polyedrem. V závislosti na tvaru polyedru a zvolené účelové funkci mohou nastat 3 případy:

- Úloha je řešitelná.
- Úloha je nepřípustná(podmínky si navzájem odporují nebo neexistuje řešení).
- Úloha je neomezená (účelová funkce lze neomezeně zvyšovat).

3.2 Tvar LP úloh

Obecnou LP úlohu lze zapsat jako:

$$\begin{aligned} \min\ & c_1x_1 + \cdots + c_nx_n \\ \text{za podmínek } & a_{i1} + \cdots + a_{in}x_n \geq b_i,\ i \in I_+ \\ & a_{i1} + \cdots + a_{in}x_n \leq b_i,\ i \in I_- \\ & a_{i1} + \cdots + a_{in}x_n = b_i,\ i \in I_0 \\ & x_j \geq 0, j \in J_+ \\ & x_j \leq 0, j \in J_- \\ & x_j \in R, j \in J_0, \end{aligned}$$

kde

$$\begin{aligned} I &= \{1, \ldots, m\} = I_0 \cup I_+ \cup I_- \\ J &= \{1, \ldots, n\} = J_0 \cup J_+ \cup J_- \end{aligned}$$

jsou rozklady indexových množin.

Pro potřeby simplexového algoritmu je potřeba převést tento tvar na standardní tvar, ve kterém jsou povolena pouze omezení typu '='. Transformaci omezení lze provést zavedením tzv. slackových proměnných pro omezení typu '$\leq$' a surplus proměnných pro omezení typu '$\geq$'. Tyto pomocné proměnné vyrovnávají rozdíl od vektoru pravé strany a nevyskytují se v účelové funkci.

3.3 Simplexová metoda

Simplexová metoda je algoritmus pro řešení úlohy lineárního programování. Algoritmus prohledává tzv. bázová řešení úloh lineárního programování přechodem mezi sousedními vrcholy polyedru (stále přípustná řešení), přičemž dvojice sousedních bází odpovídá dvojici vrcholů spojených hranou. Postupným procházením hran polyedru ve směru, ve kterém se účelová funkce zmenšuje, je nalezeno optimální řešení.

3.3.1 Inicializace algoritmu

Na začátku základního simplexového algoritmu je úloha zadána ve tvaru:

$$\min\{\boldsymbol{c^T x} | \boldsymbol{Ax} \leq \boldsymbol{b},\ \boldsymbol{x} \geq 0\}, \tag{3.1}$$

kde matice $\boldsymbol{A}$ obsahuje koeficienty podmínek ze standardního tvaru LP (spolu se standardní bází), $\boldsymbol{b} \geq 0$ je vektor pravých stran z podmínek a $\boldsymbol{c^T}$ je vektor koeficientů příslušných proměnných v účelové funkci.

3.3.2 Kroky algoritmu

Prvním krokem je vytvoření simplexové tabulky:

$$\begin{bmatrix} \boldsymbol{c^T} & \boldsymbol{0} & 0 \\ \boldsymbol{A} & \boldsymbol{I} & \boldsymbol{b} \end{bmatrix} \tag{3.2}$$

V této tabulce tvoří sloupce slackových proměnných tvoří počáteční bázi. Pokud tomu tak není, je nutné použit dvoufázový simplexový algoritmus, který ve své první fázi přípustnou počáteční bázi nalezne.

Nyní, aby se přešlo k sousední bázi a tedy novému řešení, je potřeba najít pivot. Prvním krokem krokem k nalezení pivotu je určení proměnné, která se stane aktivní (přejde do báze). Pro tento krok existuje velké množství heuristik. V této práci jsou uvažovány 2 způsoby výběru a to výběr proměnné, která má nejmenší koeficient ve vektoru redukovaných cen, a tzv. Blandovo pravidlo, které vybírá proměnnou se záporným koeficientem redukované ceny a zároveň nejmenším sloupcovým indexem. Toto pravidlo také zabraňuje možnému cyklení algoritmu. Následně zvolená proměnná je označena indexem q. Pokud jsou všechny koeficienty nezáporné, tak hodnotu účelové funkce již nelze zvýšit, algoritmus nalezl optimum a končí. V případě, že není splněna tato podmínka, algoritmus pokračuje hledáním proměnné, která se stane neaktivní (opustí bázi) a jejíž hodnota bude nastavena na 0. Ta je zjištěna nalezením řádku, kde:

$$\frac{b_i}{a_{ij}}, \quad a_{ij} > 0$$
$$i \in 1...n, \quad j \in 1...m,$$

nabývá nejmenší nezáporné hodnoty (označení indexem p). Pokud se takových proměnných vyskytne více, je zvolena ta, která přísluší proměnné s nejmenším sloupcovým indexem. Tento výběr říká, na jakou hodnotu bude vstupující proměnná nastavena. Prvek a_{pq} je tedy pivot. Nyní je pomocí Gauss-Jordanovy eliminace pozice pivotu nastavena na 1, a pozice nad ním a pod ním na 0, čímž je proměnná vpuštěna do báze. Tomuto kroku se říká ekvivalentní úprava okolo pilotu. Tímto krokem končí jedna iterace simplexového algoritmu.

3.3.3 Přepsaný algoritmus

1. Výběr indexu q pivotu takového, že $c_j < 0$
2. Výběr indexu p pivotu takového, že

$$p \in \operatorname{argmin} \frac{b_{i'}}{a_{i'j}} \tag{3.3}$$

3. Ekvivalentní úprava okolo pivotu a_q^p

Algoritmus končí, pokud nelze provést další iteraci buď z toho důvodu, že koeficienty $\boldsymbol{c}_j$ jsou nezáporné (jsme v optimu) nebo v některém sloupci j je $\boldsymbol{c}_j < 0$ a $\boldsymbol{a}_{ij} \leq 0$ pro všechna i, což značí, že úloha je neomezená.

3.3.4 Dvoufázová simplexová metoda

V případě, že je úloha zadaná v obecném tvaru, tedy neobsahuje pouze nerovnosti typu '$\leq$', může nastat situace, že matice $\boldsymbol{A}$ v sobě neobsahuje standardní bázi, tedy počáteční přípustné řešení úlohy. Případně není ani známé, zda je úloha řešitelná. Pokud tento případ nastane, je nejdříve nutné vyřešit pomocnou LP úlohu, jejímž řešením je libovolné přípustné počáteční řešení. Nejprve, pokud se ve vektoru pravých stran $\boldsymbol{b}$ vyskytne záporné číslo, je potřeba příslušný řádek v matici $\boldsymbol{A}$ vynásobit -1 a otočit nerovnost. Pomocnou úlohu definujeme jako:

$$\min\{\mathbf{1}^{\boldsymbol{T}}\boldsymbol{u} | \boldsymbol{A}\boldsymbol{x} + \boldsymbol{u} = \boldsymbol{b}, \boldsymbol{x} \geq 0, \boldsymbol{u} \geq 0\} \tag{3.4}$$

a k ní korespondující simplexovou tabulkou:

$$\begin{bmatrix} \mathbf{0} & \mathbf{1}^{\boldsymbol{T}} & 0 \\ \boldsymbol{A} & \boldsymbol{I} & \boldsymbol{b} \end{bmatrix} \tag{3.5}$$

Původní úloha je přípustná právě tehdy, pokud optimální hodnota účelové funkce pomocné úlohy je rovna 0. Této hodnoty se dosáhne, když $\boldsymbol{u} = 0$, tedy pokud nalezené řešení neporušuje žádné omezení. Na začátku algoritmu tvoří sloupce příslušící proměnným $\boldsymbol{u}$ standardní bázi. Na pomocnou úlohu tedy lze pustit simplexový algoritmus, který může skončit dvěma způsoby:

- Optimum je větší než 0, původní úloha je tedy nepřípustná.
- Optimum je rovno 0, původní úloha je tedy přípustná. Pokud není optimální řešení $(\boldsymbol{x}, \boldsymbol{u}) = 0$ pomocné úlohy degenerované, tedy v bázi nejsou žádné proměnné s hodnotou nastavenou na 0, pak jsou všechny bázové proměnné kladné. Pokud je tedy po ukončení algoritmu hodnota účelové funkce rovna 0 a všechny proměnné $\boldsymbol{u}$ jsou nebázové, pak mezi sloupci, které přísluší proměnným $\boldsymbol{x}$ existuje standardní báze.

Nalezení této počáteční báze se nazývá první fáze simplexového algoritmu. Pokud je řešení pomocné úlohy degenerované, tedy v bázi jsou některé proměnné nulové, může se stát, že po skončení první fáze zůstanou některé proměnné $\boldsymbol{u}$ v bázi. Tyto proměnné je nutné v bázi nahradit a to tak, že na řádku příslušící této proměnné najdeme nějakou proměnnou x, která není v bázi a jejíž koeficient na řádce je nenulový. Následně na této proměnné provedeme ekvivalentní úpravy, čímž proměnnou dostaneme do báze.

Kapitola 4

Sparse matice a Revidovaná Simplexová metoda

4.1 Formáty sparse matic

Sparse(tj. řídké) matice jsou matice v nichž převažují nulové prvky. Pro nižší paměťovou náročnost algoritmů jsou pro jejich uchování využity speciální formáty. V této práci jsou použity formáty CSR (Compressed sparse row) a COO (Coordinate list).

4.1.1 Coordinate list

Nejjednodušším formátem zápisu řídkých matic je formát COO, který je tvořen 3 vektory:

- Vektor $\boldsymbol{r}$ řádkových indexů
- Vektor $\boldsymbol{c}$ sloupcových indexů
- Vektor $\boldsymbol{v}$ hodnot

Tyto vektory jsou ideálně seřazeny podle řádkového indexu daného prvku pro rychlejší náhodný přístup. Hlavní předností matic v tomto formátu je jejich jednoduchá konstrukce, která probíhá pouhým přidáním hodnoty a řádkového a sloupcového indexu nenulového prvku do příslušných vektorů.

Například matici:

$$\begin{bmatrix} 0 & 1 & 5 \\ 0 & 0 & 4 \\ 1 & 0 & 0 \end{bmatrix} \tag{4.1}$$

je možné převést do COO formátu jako:

$$\boldsymbol{r} = (0, 0, 1, 2)$$

$$\boldsymbol{c} = (1, 2, 2, 0)$$

$$\boldsymbol{v} = (1, 5, 4, 1)$$

4.1.2 Compressed sparse row

Druhým formátem řídkých matic, který je v této práci využíván je formát CSR. Jeho základem jsou opět 3 vektory:

- Vektor $\boldsymbol{v}$ hodnot.
- Vektor $\boldsymbol{c}$ sloupcových indexů.
- Vektor $\boldsymbol{o}$ který ukazuje na pozici prvku ve vektoru $\boldsymbol{v}$, kterým začíná řádek v matici. Poslední prvek tohoto vektoru je vždy ukazatel na poslední prvek matice.

Tento formát je využit pro jednoduchý a rychlý přístup k požadovanému prvku, neboť pro nalezení prvku se při známém řádkovém indexu prochází pouze daný řádek a nikoli celá matice, na rozdíl od COO. Matice 4.1 vypadá po převedení do CSR formátu jako:

$$\boldsymbol{v} = (1, 5, 4, 1)$$

$$\boldsymbol{c} = (1, 2, 2, 0)$$

$$\boldsymbol{o} = (0, 2, 3, 3)$$

4.2 Revidovaná simplexová metoda

Hlavní výhodou revidované metody oproti klasické verzi je menší počet výpočetních operací, pokud je počet podmínek menší v porovnání s počtem proměnných. Dále také dokáže efektivně pracovat se sparse formátem matic. Navíc při každé iteraci neupravuje celou původní matici, ale pouze inverzní bázovou matici [5].

4.2.1 Inicializace algoritmu

Na začátku máme úlohu ve tvaru:

$$\min\{\boldsymbol{c^T x} | \boldsymbol{Ax} \leq \boldsymbol{b},\ \boldsymbol{x} \geq 0\}, \tag{4.2}$$

kde matice $\boldsymbol{A}$ obsahuje koeficienty podmínek ze standardního tvaru LP (spolu se standardní bází), $\boldsymbol{b} \geq 0$ je vektor pravých stran z podmínek a $\boldsymbol{c^T}$ je vektor koeficientů příslušných proměnných v účelové funkci. Podmínkou je znalost počáteční báze.

V případě, že zadaná úloha v tomto tvaru není, je potřeba použít dvoufázovou revidovanou simplexovou metodu 3.4, abychom nalezli počáteční bázi. Pokud počáteční báze je známa, je počáteční simplexová tabulka 3.2 rozdělena na potřebné matice a vektory:

1. Matice $\boldsymbol{B}$ určující sloupce proměnných jež jsou v bázi z původní matice $\boldsymbol{A}$
2. Matice $\boldsymbol{R}$ určující sloupce proměnných jež nejsou v bázi z původní matice $\boldsymbol{A}$
3. Vektor $\boldsymbol{b}$ koeficientů pravých stran
4. Vektor $\boldsymbol{c^T}$ koeficientů proměnných z účelové funkce
5. Vektor $\boldsymbol{x_b}$ určující indexy bazických proměnných
6. Vektor $\boldsymbol{x_r}$ určující indexy nebazických proměnných

4.2.2 Bázové řešení

Původní tvar můžeme přepsat podle námi vytvořených matic jako:

$$\begin{aligned} \boldsymbol{Bx_B} + \boldsymbol{Rx_R} &= \boldsymbol{b} \\ \boldsymbol{Bx_B} &= \boldsymbol{b} - \boldsymbol{Rx_R} \\ \boldsymbol{x_B} &= \boldsymbol{B^{-1}b} - \boldsymbol{B^{-1}Rx_R} \end{aligned}$$

a jelikož všechny proměnné, které nejsou v bázi, mají nulovou hodnotu, je $\boldsymbol{x_R}$ nulový vektor. Je tedy možné psát:

$$\boldsymbol{\beta} := \boldsymbol{B^{-1}b} \tag{4.3}$$

Což je vektor pravých stran v aktuální iteraci, neboli bázové řešení.

4.2.3 Podmínky optimality

Účelová funkce může být rozdělena: $\boldsymbol{c^T x} = \boldsymbol{c_B^T x_B} + \boldsymbol{c_R^T x_R}$, kde

- $\boldsymbol{c_B^T}$ jsou ceny bazických proměnných
- $\boldsymbol{c_R^T}$ jsou ceny nebazických proměnných

Touto úpravou jsou získány vektory vyjadřující koeficienty účelové funkce, které jsou a nejsou v bázi.

4.2.3.1 Redukované ceny

Pro vyjádření redukovaných cen je nejdříve účelová funkce vyjádřena z hlediska nebazických proměnných:

$$\begin{aligned} \boldsymbol{c^T x} = & \\ & = \boldsymbol{c_B^T x_B} + \boldsymbol{c_R^T x_R} \\ & = \boldsymbol{c_B^T}(\boldsymbol{B^{-1}b} - \boldsymbol{B^{-1}R x_R}) + \boldsymbol{c_R^T x_R} \\ & = \boldsymbol{c_B^T B^{-1} b} - \boldsymbol{c_B^T B^{-1} R x_R} + \boldsymbol{c_R^T x_R} \\ & = \boldsymbol{c_B^T B^{-1} b} + (\boldsymbol{c_R^T} - \boldsymbol{c_B^T B^{-1} R}) \boldsymbol{x_R} \end{aligned} \tag{4.4}$$

Nechť $\boldsymbol{a}_j$ je sloupcem z $\boldsymbol{A}$, který přísluší nebázové proměnné x_j. Poté z 4.4 je možné vyjádřit redukovanou cenu jako

$$d_j := c_j - \boldsymbol{c_B^T B^{-1} a_j} \tag{4.5}$$

Pokud je 4.5 dosazeno do 4.4 je možné napsat

$$\boldsymbol{c^T x} = \boldsymbol{c_B^T B^{-1} b} + \boldsymbol{d_R^T x_R} \tag{4.6}$$

4.2.3.2 Hodnota účelové funkce

K získání hodnoty účelové funkce je využito faktu, že všechny nebazické proměnné jsou nastavené na 0, tedy $\boldsymbol{x_R} = 0$. Z 4.6 tedy plyne

$$z := \boldsymbol{c_B^T B^{-1} b} \tag{4.7}$$

4.2.3.3 Simplexový multiplikátor

Z 4.5 a 4.7 lze vidět, že výpočet

$$\boldsymbol{c_B^T B^{-1}} \tag{4.8}$$

se objevuje v obou rovnicích. Pro vyhnutí se opakovanému výpočtu tohoto vektoru je zaveden tzv. simplexový multiplikátor

$$\boldsymbol{\pi^T} := \boldsymbol{c_B^T B^{-1}} \tag{4.9}$$

s jehož pomocí je možno rovnice přepsat jako

$$\begin{aligned} d_j &= c_j - \boldsymbol{\pi^T a_j} \\ z &= \boldsymbol{\pi^T b} \end{aligned}$$

4.2.3.4 Podmínka optimality

Podmínkou optimality je, aby všechny redukované ceny proměnných, které nejsou v bázi, byly nezáporné:

$$d_j \geq 0 \quad \forall j \in \boldsymbol{R} \tag{4.10}$$

4.2.4 Vylepšení neoptimálního řešení

Pokud některá nebazická proměnná x_q nesplňuje podmínku optimality 4.10, tedy:

$$d_q \geq 0, \tag{4.11}$$

pak tato proměnná vstoupí do báze. Jelikož je proměnná x_q nebazická, je její současná hodnota 0. Cílem je zvýšení této hodnoty při stálém splnění znaménkového omezení bazických proměnných. Potom nechť $t \geq 0$ je nová hodnota pro x_q a je možné napsat:

$$\boldsymbol{x_B(t) = B^{-1}b - B^{-1}Rx_R = B^{-1}b - B^{-1}a_q t = \beta - t\alpha_q} \tag{4.12}$$

Nyní, pro stále splnění znaménkového omezení bazických proměnných je třeba, aby $\forall i \in \{1, \ldots, m\}$ platilo $\boldsymbol{\beta_i - t\alpha_q^i} \geq 0$. To je zaručené ve dvou případech:

- Pokud je $\alpha_q^i \leq 0$, pak je omezení splněno pro každé $t \geq 0$.
- Pokud je $\alpha_q^i > 0$, pak potřebujeme aby $\frac{\beta_i}{\alpha_q^i} \geq t$.

Jelikož tyto podíly značí, na jakou hodnotu povolí dané omezení zvolenou proměnnou zvýšit, je potřeba zvolit tuto hodnotu jako

$$\theta := \min\{\frac{\beta_i}{\alpha_q^i} | \alpha_q^i > 0\} \tag{4.13}$$

Pokud by byla zvolena hodnota vyšší, než minimální, nalezené řešení nebude přípustné. V případě, že se těchto hodnot vyskytne více, tak dle Blandova pravidla je zvolena vždy ta, která přísluší proměnné s nižším sloupcovým indexem. Při výběru proměnné, která opouští bázi tedy mohou nastat dvě situace:

- Pokud by $\theta = +\infty$, neboli neexistuje i takové, že $1 \leq i \leq m$ a $\alpha_q^i > 0$. Pak je možné hodnotu účelové funkce nekonečně zlepšovat a řešení LP je neomezené.
- Pokud je $\theta < +\infty$, pak je možné nastavit $x_q = \theta$ bez toho, aby bylo porušeno znaménkové omezení bazických proměnných. x_q vstoupí do báze a x_{k_p} bázi opustí.

Pokud řešení není neomezené, potom nalezená proměnná, která opustí bázi, je označena indexem p, α_q^p je tedy pivot.

4.2.5 Aktualizace

Nyní je potřeba aktualizovat vektor pravých stran $\boldsymbol{\beta}$, vektory určující které proměnné jsou/nejsou v bázi ($\boldsymbol{x_b}$, $\boldsymbol{x_r}$) a inverzní bázovou matici $\boldsymbol{B^{-1}}$ pro další iteraci.

4.2.5.1 Aktualizace $\boldsymbol{\beta}$

Pro získání nového bázového řešení je nastavena proměnná, která nově vstoupila do báze, na hodnotu θ. Tedy:

$$\beta_p = \theta \tag{4.14}$$

Poté jsou zbylé proměnné nastaveny, aby odpovídaly hodnotám, které by vznikli po ekvivalentních pivotových operacích.

$$\beta_i = \beta_i - \theta\alpha_q^i, i \neq p \tag{4.15}$$

4.2.5.2 Aktualizace bázové matice

Cílem je náhrada sloupce v bázové matici, který odpovídá proměnné, která opouští bázi, za sloupec proměnné, která do báze vstupuje. Tedy:

$$\boldsymbol{B} = \boldsymbol{B} + \left(\boldsymbol{a_q} - \boldsymbol{a_{k_p}}\right) \boldsymbol{e_p^T} \tag{4.16}$$

4.2.5.3 Aktualizace hodnoty účelové funkce

Aktualizace hodnoty účelové funkce je provedena tak, že ke stávající hodnotě je přičten výsledek součinu redukované ceny proměnné d_q, která vstupuje do báze a hodnoty θ, na níž je tato proměnná nastavena. Tedy:

$$z = z + \theta d_q \tag{4.17}$$

4.2.6 Algoritmický zápis

1. Inicializace:
   - Nalezení počáteční splnitelné báze $\boldsymbol{B}$
   - Výpočet:

$$\begin{aligned} \boldsymbol{B^{-1}} \\ \boldsymbol{\beta} = \boldsymbol{B^{-1}b} \\ z \ = \boldsymbol{c_B^T \beta} \end{aligned}$$

2. Ocenění:
   - Výpočet:

$$\begin{aligned} \boldsymbol{\pi} = \boldsymbol{c_B^T B^{-1}} \\ d_j = c_j - \boldsymbol{\pi^T a}_j \end{aligned}$$

   - Pokud $\forall j \in \boldsymbol{R}, d_j \geq 0$ pak OPTIMUM, je nalezena optimální hodnota a algoritmus končí.
   - Jinak výběr $d_q < 0$ a výpočet:

$$\boldsymbol{\alpha_q} = \boldsymbol{B^{-1} a_q} \tag{4.18}$$

3. Poměrový test:
   - Výpočet:

$$I \ = \{i | 1 \leq i \leq m, \alpha_q^i > 0\}$$

- Pokud $I = \emptyset$, pak je řešení NEOMEZENÉ a algoritmus končí
- Jinak $\theta = \min_{i \in I}(\frac{\beta_i}{\alpha_q^i})$ a p tak, že $\theta = \frac{\beta_p}{\alpha_q^p}$

4. Aktualizace:

- Nové bazické řešení: $\beta_p = \theta$, $\beta_i = \beta_i - \theta\alpha_q^i$, pokud $i \neq p$
- Nová báze: $B = B + (a_q - a_{k_p})e_p^T$
- Nová hodnota účelové funkce: $z = z + \theta d_q$

Kapitola 5

Implementace

Tato kapitola se věnuje samotné implementaci revidované simplexové metody pro GPU. Nejprve jsou popsány knihovny, které byly zvažovány a následně vybrány pro implementaci. Poté jsou popsány jednotlivé části programu, bez ohledu na jejich konkrétní implementaci, a jsou vysvětleny. Celý program je implementován v jazyce CUDA C++ s balíčkem CUDA 5.5

5.1 Využité knihovny

Pro implementaci algoritmu bylo potřeba najít knihovny, které umožňují rychlou a efektivní práci s vektory a zároveň implementují formáty a elementární operace se sparse maticemi. Tyto podmínky splňují knihovny Thrust, CUSP a cuSPARSE.

5.1.1 Thrust

Knihovna Thrust [6] poskytuje vysokou úroveň abstrakce pro GPU programování. Mezi hlavní přednosti této knihovny patří rychlá implementace algoritmů jako je sort, scan nebo reduction.

5.1.2 cuSPARSE

Pro potřeby implementace formátu sparse matic a operací na nich byla zvažována knihovna cuSPARSE [7]. Jedná se o knihovnu společnosti NVIDIA dodávanou spolu s CUDA instalací. Využití této knihovny bylo ale posléze zamítnuto, z důvodu náročnější

implementace. Pro operace se sparse maticemi byla místo této knihovny použita knihovna CUSP.

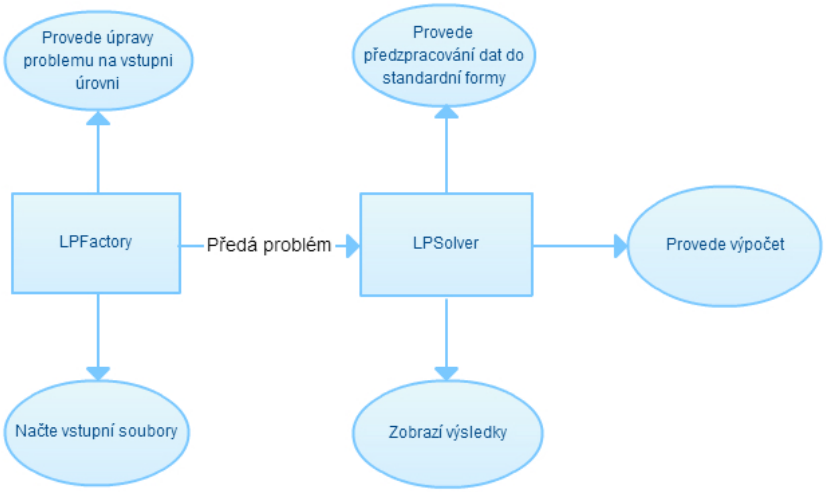
5.1.3 CUSP

CUSP [8] je knihovnou vyvíjenou společností NVIDIA, která je zároveň dostupná jako open-source projekt. Poskytuje vysokou úroveň abstrakce pro manipulaci se sparse maticemi a implementuje na nich základní operace. Mezi její hlavní přednosti patří implementace formátů COO a CSR a rutin pro jejich vzájemné konverze. Dále také fakt, že se jedná o nadstavbu výše zmíněné knihovny Thrust, což umožňuje standardní konverzi vektorů těchto knihoven a zároveň současné využití algoritmů poskytovaných oběma knihovnami.

5.2 Architektura programu

Na obrázku 5.1 lze vidět architekturu programu, jehož kostru tvoří 2 třídy:

- Třída *LPFactory* zprostředkovává načtení souboru a jeho úpravu na vstupní úrovni
- Třída *LPSolver*, která od třídy *LPFactory* převezme načtený problém, předzpracuje jej, provede jeho výpočet a zobrazí získané řešení.



Obrázek 5.1: Architektura programu

5.2.1 Načítání vstupních souborů

Vstupem do programu jsou 2 soubory. První soubor obsahuje seznam všech proměnných, které se v programu vyskytují a také určuje v jakém pořadí budou přiřazeny jednotlivé hodnoty proměnným po skončení algoritmu. Druhý soubor obsahuje samotný optimalizační problém v upraveném .lp formátu. Rozdíly oproti konvenci:

- Koeficient se znaménkem před každou proměnnou
- Pokud je proměnná omezena na konstantu, je převedena na omezení.

Takto upravený formát vstupních LP souborů byl zvolen z důvodu zaměření na odladění algoritmu a jeho testování. Postup načítání je zobrazen na obr.5.2.

5.2.2 Úpravy problému při načítání

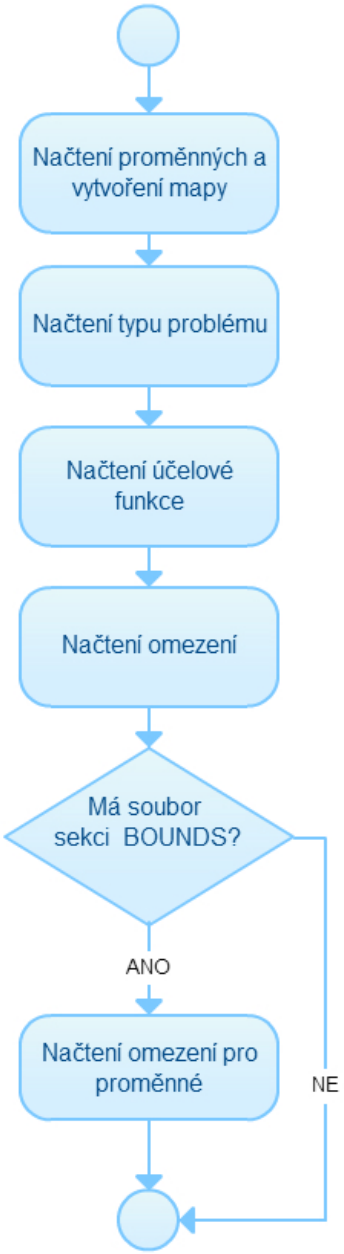
Při načítání problému se může stát, že pravá strana některého omezení je záporná. V tomto případě se celému načtenému řádku změní znaménko, aby bylo zachováno znaménkové omezení proměnných.

5.2.2.1 Škálování problému

Po načtení dat je problém "vyškálován", aby se předešlo výskytu velkého rozdílu v řádu minimálního a maximálního načteného koeficientu. Toto vyškálování problému má vliv na akumulaci numerických chyb v algoritmu a spočívá ve zjištění v absolutní hodnotě minimálního a maximálního koeficientu, následném vypočtení střední hodnoty jejich řádu r, kde r je celé číslo, a vynásobení všech načtených hodnot koeficientem 10^{-r}. Rozsah koeficientů ve škálovaném problému zůstane zachován, dojde pouze k posunutí jejich řádů.

5.2.3 Předzpracování dat

Pro potřeby simplexového algoritmu je nutné, aby byla vstupní data v daném standardním tvaru. Načtená data ze souboru je tedy nutné upravit, aby tomuto formátu vyhovovala. Následující postup je přejat z [9].



Obrázek 5.2: Průběh načítání souborů

5.2.3.1 Úprava mezí proměnných a matice A

Ve standardním tvaru jsou všechny proměnné kladné, tj.$\boldsymbol{x} \geq 0$. Ve většině reálných případů tomu ovšem tak nebývá a je tedy proto nutné je do tohoto tvaru převést následujícími kroky:

- Pokud má proměnná meze ve tvaru $x \leq a$, je tato mez přidána do seznamu omezení.

- Meze typu $x \geq a$ jsou do standardního tvaru převedeny substitucí $x' = x - a$ do původního problému.
- Meze proměnných, které jsou ve tvaru $a \leq x \leq b$ jsou do standardního tvaru převedeny posunutím mezí na $0 \leq x \leq (b - a)$ a následném přidání $x \leq (b - a)$ do seznamu omezení úlohy.
- Meze typu $x \leq -a$ jsou do standardního tvaru převedeny substitucí $x' = -(x + a)$.
- Posledním případem jsou neomezené proměnné, pro které platí $-\infty \leq x \leq \infty$. Tento problém je vyřešen substitucí proměnné $x = (x_1 - x_2)$, kde $x_1, x_2 \geq 0$.

Jelikož byly při úpravě mezí proměnných některé meze posunuty, je potřeba po této substituci tyto posuny odečíst od vektoru pravé strany $\boldsymbol{b}$.

5.2.3.2 Vytvoření rozšířené matice $\boldsymbol{A}$

Protože všechna omezení musí být ve tvaru rovností, jsou do matice $\boldsymbol{A}$ přidány slackové a surplus proměnné, čímž je vytvořena rozšířená matice $\boldsymbol{A}$.

- Nerovnost typu '$\leq$' je nahrazena přidáním slackové proměnné
- Nerovnost typu '$\geq$' je nahrazena přidáním surplus proměnné.

Dále je také pro potřeby první fáze revidovaného simplexového algoritmu připojit za matici $\boldsymbol{A}$ jednotkovou matici umělých proměnných, která bude tvořit počáteční standardní bázi. Jako názorný příklad je do standardní formy převeden následující příklad:

$$
\begin{aligned}
&\text{Minimalizuj} \\
&3x - 3y + z \ + 8w \\
&\text{S podmínkami} \\
&x + 2y - z \ \geq 5 \\
&-3z + 2w \leq -10 \\
&-3x - y + z \ \leq 4 \\
&-y - 3z \leq -5 \\
&\text{Omezení proměnných} \\
&-\infty \leq x \leq \infty \\
&y \geq 5 \\
&-3 \leq z \ \leq 3 \\
&w \leq -1
\end{aligned}
$$

Základní načtení matice $\boldsymbol{A}$ tohoto problému je uvedeno v tabulce 5.1. Řádky 2 a 4 jsou načteny s opačným znaménkem, neboť jim náleží záporná pravá strana.

Tabulka 5.1: Počáteční načtení matice A

x	y	z	w
1	2	-1	0
0	0	3	-2
-3	-1	1	0
0	1	3	0

Následně jsou upraveny jednotlivé proměnné dle 5.2.3.1. Tabulka 5.2 reprezentuje stav matice po těchto úpravách. Po úpravě proměnných jsou nahrazeny nerovnosti rovnostmi, toho je dosaženo přidáním slackových a surplus proměnných do problému. Výsledek tohoto kroku je uveden v tab.5.3.

Tabulka 5.2: Matice A po úpravě mezí proměnných

x_1	x_2	y	z	w
1	-1	2	-1	0
0	0	0	-3	-2
3	-3	1	-1	0
0	0	1	3	0
0	0	0	1	0

Tabulka 5.3: Matice A po přidání slackových a surplus proměnných

x_1	x_2	y	z	w	s_1	s_2	s_3	s_4	s_5
1	-1	2	-1	0	-1	0	0	0	0
0	0	0	-3	-2	0	1	0	0	0
3	-3	1	-1	0	0	0	-1	0	0
0	0	1	3	0	0	0	0	-1	0
0	0	0	1	0	0	0	0	0	1

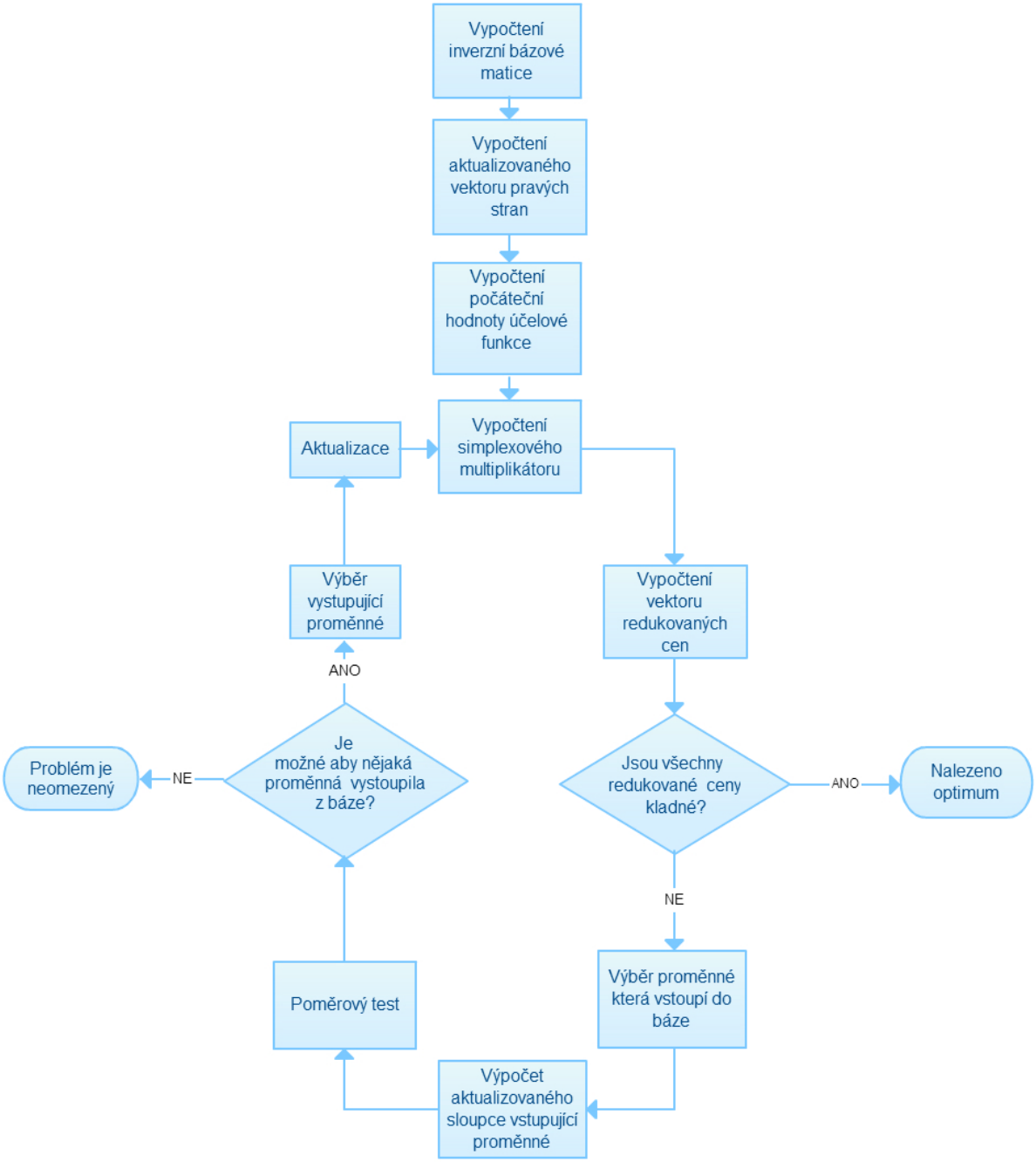
Posledním krokem pro získání výsledné matice $\boldsymbol{A}$ je přidání umělých proměnných, které budou tvořit standardní bázi na začátku první fáze simplexového algoritmu. Výsledná matice $\boldsymbol{A}$ tedy vypadá 5.4.

Tabulka 5.4: Konečná rozšířená matice $\boldsymbol{A}$

x_1	x_2	y	z	w	s_1	s_2	s_3	s_4	s_5	u_1	u_2	u_3	u_4	u_5
1	-1	2	-1	0	-1	0	0	0	0	1	0	0	0	0
0	0	0	-3	-2	0	1	0	0	0	0	1	0	0	0
3	-3	1	-1	0	0	0	-1	0	0	0	0	1	0	0
0	0	1	3	0	0	0	0	-1	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	0	0	0	0	1

5.2.4 Implementace revidovaného simplexového algoritmu

Jelikož je simplexový algoritmus z podstaty sekvenční, byla zvolena paralelizace na úrovni jednotlivých operací. Tato možnost byla zvolena, neboť se jednotlivé kroky skládají z maticových operací, které lze snadno paralelizovat. Samotný postup výpočtu je uveden na obr.5.3.



Obrázek 5.3: Průběh simplexového algoritmu

5.2.4.1 Inicializace

Pro potřeby revidovaného simplexového algoritmu je zadaný problém rozdělen na následující matice a vektory:

- Matice $\boldsymbol{A}$, která je předem transponována a převedena do formátu CSR pro rychlejší přístup při čtení sloupců.
- Vektor $\boldsymbol{b}$ pravých stran.
- Vektor reprezentující redukované ceny $\boldsymbol{d}$.
- Vektor reprezentující poměry z poměrového testu $\frac{\boldsymbol{\beta}}{\boldsymbol{\alpha_q}}$.
- Vektor simplexového multiplikátoru $\boldsymbol{\pi}$.
- CSR matice reprezentující inverzní bázovou matici $\boldsymbol{B^{-1}}$.
- COO matice, která je použita na konci iterace pro aktualizaci $\boldsymbol{B^{-1}}$.
- Vektory $\boldsymbol{x_b}$ a $\boldsymbol{x_r}$ indexů proměnných které jsou a nejsou v bázi.
- Vektory $\boldsymbol{c_b}$ a $\boldsymbol{c_r}$ koeficientů proměnných z účelové funkce, které jsou a nejsou v bázi.

Po inicializaci těchto matic a vektorů je alokován požadovaný prostor na GPU a vektory s maticemi jsou překopírovány.

5.2.4.2 Výpočet inverzní bázové matice

Jelikož v první fázi algoritmu je inverzní bázová matice jednotková a ve druhé fázi je využita výstupní matice z první fáze, nebylo potřeba tento krok nijak implementovat.

5.2.4.3 Výpočet aktualizovaného vektoru pravých stran

Pro implementaci tohoto kroku byla využita funkce knihovny *cusp::multiply()*, která byla aplikována na inverzní bázovou matici $\boldsymbol{B^{-1}}$ a vektor pravých stran $\boldsymbol{b}$.

5.2.4.4 Výpočet počáteční účelové hodnoty

Výpočet počáteční hodnoty účelové funkce je implementován v metodě *getValueZVector()*, která spouští Kernel na grafické kartě, který získá vektor jednotlivých součinů vektoru účelové funkce $\boldsymbol{c_b}$ a aktualizovaného vektoru pravých stran $\boldsymbol{\beta}$. Tento vektor součinů je následně paralelně sečten pomocí funkce knihovny *thrust::reduce()*.

5.2.4.5 Výpočet simplexového multiplikátoru

Jelikož funkce knihovny *cusp::multiply()* podporuje pouze násobení typu Matice × Vektor a transpozice inverzní bázové matice by byla paměťově náročná operace, byla implementována metoda *multiplyVectorByCsrMatrix()* nahrazující tuto operaci, která spouští kernel o n vláknech, kde n je velikost výsledného vektoru. Každé z těchto vláken vypočte příslušný prvek výsledného vektoru. Tato metoda byla použita na vektor $\boldsymbol{c_b}$ a inverzní bázovou matici $\boldsymbol{B^{-1}}$ pro zisk výsledného vektoru $\boldsymbol{\pi}$ 4.9.

5.2.4.6 Výpočet vektoru redukovaných cen

Pro výpočet redukovaných cen 4.5 byla implementována metoda *getReducedCostVector()*, která spouští Kernel o n vláknech, kde n je délka vektoru redukovaných cen. Každé z těchto vláken vypočte součin vektorů $\boldsymbol{\pi}$ a dané redukované ceny příslušného sloupce z matice $\boldsymbol{A}$. Tento součin následně odečte od koeficientu proměnné, který přísluší redukované ceně v účelové funkci a výsledek zapíše do výsledného vektoru.

5.2.4.7 Výběr vstupující proměnné

Pro výběr proměnné, která vstoupí do báze je implementována metoda *getIndexQ()*, která vrací 2 indexy proměnných zvolených podle 2 strategií.

1. Proměnná je zvolena podle Blandova pravidla. To je provedeno filtrací pouze záporných redukovaných cen, pro kterou byly implementovány příslušející metody, a následným vrácením té redukované ceny, která má nejnižší sloupcový index

2. Proměnná je zvolena podle nejnižší redukované ceny. Ta je nalezena použitím funkce knihovny *thrust::minElement()*.

Pokud nelze podle pravidel zvolit žádnou proměnnou, tedy všechny proměnné mají nezápornou redukovanou cenu, je vrácena první proměnná z vektoru nebazických proměnných.

Výběr proměnné s nezápornou redukovanou cenou je detekován a signalizuje nalezení optimální hodnoty a ukončení algoritmu. Pokud se jednalo o druhou fázi algoritmu, jsou následně provedeny zpětné substituce provedené při předzpracování.

5.2.4.8 Výpočet aktualizovaného sloupce vstupující proměnné

Výpočet aktualizovaného sloupce vstupující proměnné je implementován v metodě *getAlphaQ()*, která spustí Kernel na grafické kartě o n vláknech, kde n je velikost sloupce z matice $\boldsymbol{A}$. Každé z těchto vláken vypočte příslušející hodnotu výsledného vektoru vynásobením inverzní bázové matice $\boldsymbol{B^{-1}}$ a sloupce vstupující proměnné.

5.2.4.9 Poměrový test

Tento krok byl proveden implementací metody *getRatioVector()*, která spustí Kernel o n vláknech, kde n je velikost vektoru $\boldsymbol{b}$ pravé strany. Každé z těchto vláken vypočte hodnotu příslušející dané pozici 3. Pokud je hodnota vektoru $\boldsymbol{\alpha_q}$ na dané pozici ≤ 0 je na výslednou pozici dána konstanta ∞.

5.2.4.10 Výběr vystupující proměnné

Výběr vystupující proměnné 4.13 je proveden dle Blandova pravidla. Nejprve jsou odfiltrovány z vektoru poměrů hodnoty ∞. Následně je ze zbylých hodnot nalezeno minimum pomocí funkce knihovny *thrust::minElement()*. Poté jsou vybrány proměnné, které mají jako hodnotu toto minimum a je vrácena proměnná, které přísluší nejnižší index ve vektoru bazických proměnných $\boldsymbol{x_b}$, a je označena jako θ.

Pokud není nalezena žádná proměnná, která je schopna vstoupit do báze (všechny hodnoty vektoru poměrů jsou ∞), je vrácena první proměnná z vektoru $\boldsymbol{x_b}$ bazických proměnných. Výběr proměnné, jejíž hodnota v poměrovém testu je ∞, je zachycen a signalizuje, že problém je neomezený. Program je následně ukončen.

5.2.4.11 Aktualizace hodnoty účelové funkce

Aktualizace hodnoty účelové funkce je provedena přičtením součinu redukované ceny vybrané proměnné a hodnoty na kterou tato proměnná je nastavena. Tedy:

$$z = z + d_q\theta$$

5.2.4.12 Aktualizace vektorů bazických a nebazických proměnných

Pro tuto aktualizaci byla implementována metoda *updateBaseIndices()*, která zamění vstupující proměnnou z vektoru $\boldsymbol{x_r}$ za vystupující z vektoru $\boldsymbol{x_b}$. Tato operace je provedena i na vektorech $\boldsymbol{c_r}$ a $\boldsymbol{c_b}$, tedy na koeficientech daných proměnných v účelové funkci.

5.2.4.13 Aktualizace vektoru pravé strany

Aktualizace je implementována v metodě *updateBeta()*, která na grafické kartě spustí kernel o počtu vláken příslušící velikosti vektoru pravé strany. Vlákno s indexem, který přísluší proměnné vystupující z báze, aktualizuje svoji hodnotu na:

$$\beta_p = \theta,$$

všechna ostatní vlákna aktualizují jim příslušnou hodnotu na:

$$\beta_i = \beta_i - \theta\alpha_q^i.$$

5.2.4.14 Aktualizace inverzní bázové matice

Jelikož výpočet inverzní bázové matice v každé iteraci by byl výpočetně náročný, byla implementována metoda, která stávající $\boldsymbol{B^{-1}}$ upraví do tvaru vhodného pro další iteraci. Tato metoda spočívá ve výpočtu aktualizační matice, která po vynásobení stávající $\boldsymbol{B^{-1}}$ vytvoří novou $\boldsymbol{B^{-1}}$ pro další iteraci. Následující postup je převzat z [5].

Aktualizační matice je vytvořena pomocí vektoru $\boldsymbol{\alpha_q}$ a reprezentuje řádkové operace na tomto vektoru pro získání 1 na pozici výstupní proměnné p a 0 na všech ostatních pozicích. Tvar této matice tedy lze zapsat jako:

$$\boldsymbol{E} = (\boldsymbol{e1}, \ldots, \boldsymbol{e_{p-1}}, \boldsymbol{\eta}, \boldsymbol{e_{p+1}}, \ldots, \boldsymbol{e_m}),$$

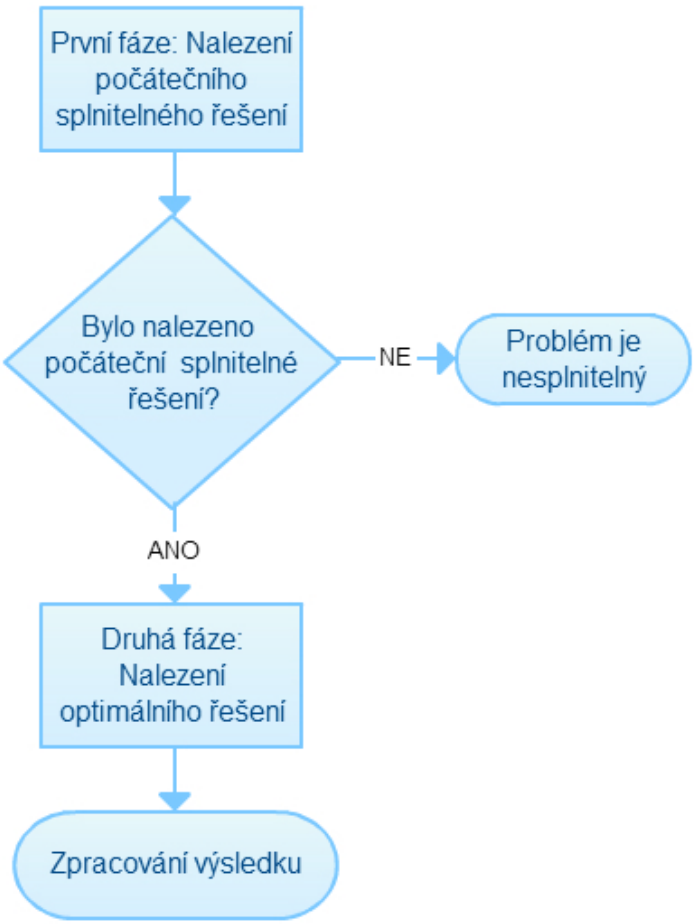
kde vektory označené jako $\boldsymbol{e}$ jsou sloupce jednotkové matice a vektor $\boldsymbol{\eta}$ má tvar

$$\boldsymbol{\eta^T} = \left(\frac{-\alpha_q^1}{\alpha_q^p}, ..., \frac{-\alpha_q^{p-1}}{\alpha_q^p}, \frac{1}{\alpha_q^p}, \frac{-\alpha_q^{p+1}}{\alpha_q^p}, ..., \frac{-\alpha_q^m}{\alpha_q^p} \right)$$

Vygenerování aktualizační matice probíhá v metodě *generateUpdateMatrix()*, která spustí kernel o počtu vláken odpovídajícímu počtu řádků výsledné matice. Každé z těchto vláken zapíše odpovídající řádek do předalokované paměti.

5.2.5 Postup výpočtu

Pro vyřešení vstupního problému je nejdříve nutné zjistit zda je problém řešitelný a pokud ano, tak najít libovolné počáteční řešení.



Obrázek 5.4: Průběh výpočtu

5.2.5.1 První fáze simplexového algoritmu

V této fázi je vytvořena účelová funkce odpovídající součtu umělých proměnných, které tvoří počáteční bázi, a vektory určující bazické a nebazické proměnné. Tedy

- $\boldsymbol{x_b}$ obsahuje indexy umělých proměnných.
- $\boldsymbol{x_r}$ obsahuje indexy zbylých proměnných problému.
- $\boldsymbol{c_b}$ je na začátku tvořen vektorem 1, který odpovídá součtu umělých proměnných.
- $\boldsymbol{c_r}$ je na začátku tvořen vektorem 0, jelikož všechny původní proměnné jsou nebazické.

Dále je pro tuto fázi inicializována inverzní bázová matice jako jednotková matice. Takto připravená data jsou poslána do simplexového algoritmu 5.2.4.

5.2.5.2 Rozhodnutí o řešitelnosti problému

Pokud po skončení simplexového algoritmu je hodnota účelové funkce rovna 0, je daný problém řešitelný a algoritmus nalezl počáteční řešení. V případě, že hodnota účelové funkce není 0, program signalizuje neřešitelnost problému.

Je-li problém degenerovaný, tedy v bázi jsou některé proměnné, které mají hodnotu 0, je možné, že v bázi zůstaly umělé proměnné s hodnotou 0, které je z báze potřeba odstranit.

5.2.5.3 Odstranění umělých proměnných z počátečního řešení

Odstranění probíhá podle kroků simplexového algoritmu. Je dán index výstupní umělé proměnné p a je potřebné najít vstupní proměnnou. Ta je nalezena postupnou aktualizací sloupců, nebazických proměnných a v případě že v některém z těchto sloupců je na pozici p nenulová hodnota, je proměnná sloupci příslušející určena jako vstupní. Následně jsou provedeny kroky aktualizace, čímž je umělá proměnná z báze odstraněna a nahrazena. Tento postup byl přejat z [4].

5.2.5.4 Druhá fáze simplexového algoritmu

Pro druhou fázi musí být na datech, které prošli první fází, provedeny následující úpravy:

- Z matice $\boldsymbol{A}$ jsou odstraněny sloupce příslušející umělým proměnným.
- Velikost vektoru redukovaných cen je změněna, aby odpovídala počtu nebazických proměnných po odstranění umělých proměnných.
- Z vektoru $\boldsymbol{x_r}$ nebazických proměnných jsou odfiltrovány indexy obsahující umělé proměnné.
- Z původní účelové funkce jsou do vektoru $\boldsymbol{c_r}$ přiřazeny koeficienty příslušející nebazickým proměnným z vektoru $\boldsymbol{x_r}$.
- Z účelové funkce problému jsou přiřazeny do vektoru $\boldsymbol{c_b}$ koeficienty bazických proměnných ve vektoru $\boldsymbol{x_b}$.

Ostatní vektory a matice, na kterých nebyly provedeny žádné úpravy, zůstávají pro tuto fázi stejné. Takto připravená data jsou poslána do simplexového algoritmu, po jehož skončení je buď známo optimum, nebo je zjištěno, že problém je neomezený.

5.2.6 Zpracování výsledku

Pokud je po druhé fázi nalezeno optimum, jsou z GPU zkopírovány do paměti počítače vektory $\boldsymbol{x_b}$ a $\boldsymbol{\beta}$, které reprezentují bazické proměnné a jim příslušné hodnoty. Následně je v paměti alokován vektor o velikosti počtu proměnných v problému načteném ze souboru. Vektor je poté procházen a u každé proměnné je nahlédnuto do vektoru $\boldsymbol{x_b}$, zda se v něm daná proměnná nevyskytuje. Pokud se v něm nachází, je převedena podle následujících pravidel do svého původního tvaru:

- Je-li proměnná posunutá, je její hodnota z vektoru $\boldsymbol{\beta}$ posunuta zpět do původních mezí.
- Má-li proměnná změněné znaménko, tak je jí znaménko změněno zpět.
- V případě neomezené proměnné, je ve vektoru $\boldsymbol{x_b}$ vyhledávána i proměnná s indexem o 1 větším, pokud nalezena nebyla, je považována její hodnota za 0. Tyto 2 proměnné jsou podle 5.2.3.1 od sebe odečteny pro získání výsledné hodnoty proměnné.

Kapitola 6

Testování programu

Tato kapitola je věnována porovnávání výsledků vytvořeného programu. Testování bylo prováděno na 2 grafických kartách a to Nvidia GeForce GTX570 (dále jen GPU1) a Nvidia GeForce GTX TITAN (dále jen GPU2). Jako vstupní data byly zvoleny veřejně dostupné LP problémy 'NETLIB LP' [10]. V první části je program otestován ve své původní verzi na GPU1. Následně jsou rozebrány problémy, které v průběhu výpočtu nastávají, zvolený způsob jejich ošetření a také důsledky způsobené ošetřením. Přínos úprav je následně vyhodnocen na GPU1 a úspěšně vyřešené problémy na GPU2. Program není testován na dense problémech, jelikož je ze své podstaty koncipován pro řešení sparse problémů a řešení dense problémů by tedy bylo neefektivní.

Tabulka 6.1: Srovnání GPU1 a GPU2

Model	GeForce GTX 570	GeForce GTX Titan
Frekvence jádra	732 MHz	837 MHz
Frekvence shaderů	1464 MHz	837 MHz
Unifikovaných shaderů	480	2688

6.1 Soubor testovaných problémů

Program byl testován na vybraných problémech ze souboru problémů 'NETLIB LP'. V tabulce 6.2 jsou uvedeny názvy jednotlivých problémů spolu s počtem proměnných, omezení a nenulových prvků.

Tabulka 6.2: Seznam testovaných problémů

Název	Proměnných	Omezení	Počet NNZ	%NNZ	Optimum
25fv47	1571	821	10400	0,81	5501,84
80bau3b	9799	2735	21500	0,08	9,87E+05
adlittle	97	56	383	7,05	2,25E+05
afiro	32	27	83	27,00	-464,753
agg	163	488	2410	488,00	-3,60E+07
agg2	302	516	4284	2,75	-2,02E+07
agg3	472	305	2494	1,73	1,03E+07
bandm	472	305	2494	1,73	-1,59E+02
beaconfd	262	173	3375	7,45	3,36E+04
blend	83	74	491	7,99	-3,08E+01
bnl1	1175	632	5121	0,69	1,98E+03
bnl2	3489	2280	13999	0,18	1,81E+03
bore3d	315	234	1430	1,94	1,37E+03
brandy	249	182	2148	4,74	1,52E+03
capri	353	287	1783	1,76	2,69E+03
cycle	2857	1886	20720	0,38	-5,23E+00
czprob	3523	1156	10898	0,27	2,19E+06
d2q06c	5167	2171	32417	0,29	1,23E+05
d6cube	6184	404	37704	1,51	3,15E+02
degen2	534	444	3978	1,68	-1,44E+03
degen3	1818	1503	24646	0,90	-9,87E+02
dfl001	12230	6071	35632	0,05	1,13E+07
e226	282	223	2578	4,10	-1,88E+01
etamacro	688	482	2491	0,75	-7,56E+02

6.2 Výsledky původní verze programu

Nejprve byla otestována standardní implementace simplexového algoritmu na GPU1 a jejíž výsledky jsou vidět v tabulce 6.3.

Tabulka 6.3: Výsledky původní implementace na GTX570

Název	Stav	#iterací 1.řeš	1.řeš[s]	# iterací optima	Optimum	Optimum [s]	Nejdelší iterace[ms]
25fv47	neřešitelné	13458	898,2	-	-	-	113,4
80bau3b	ukončeno	7000+	-	-	-	-	1122
adlittle	vyřešené	110	0,814	82	225495	0,639	11,32
afiro	vyřešené	27	0,18	0	-464,753	0	7,604
agg	neřešitelné	653	10,39	-	-	-	20,94
agg2	vyřešené	543	8,23	52	-2,02E-07	1.00	19,16
agg3	ukončeno	563	8,99	cykl(1000)	-	2,27	26,12
bandm	ukončeno	30 000+	-	-	-	-	21,11
beaconfd	ukončeno	cykl(1000)	-	-	-	-	11,94
blend	nesprávně	488	3,87	31	-276,97	0,25	9,54
bnl1	neřešitelné	7723	429	-	-	-	71,1
bnl2	ukončeno	30 000+	-	-	-	-	508,37
bore3d	ukončeno	cykl(7000)	-	-	-	-	17,22
brandy	nesprávně	816	8,69	141	2338,34	1,88	14,68
capri	neřešitelné	1784	57,32	215	2690,02	3,12	45,67
cycle	ukončeno	30 000+	-	-	-	-	107,01
czprob	ukončeno	1683	43,58	30 000+	-	-	89,1
d2q06c	ukončeno	30 000+	-	-	-	-	557,65
d6cube	vyřešeno	10113	338,6	14126	315,947	631	48,5
degen2	ukončeno	2077	81,64	30 000+	-	-	53
degen3	ukončeno	30 000+	-	-	-	-	94,99
dfl001	ukončeno	5000+	-	-	-	-	1047
e226	nesprávně	482	4,9	649	-24,46	10,1	17,96
etamacro	neřešitelné	3072	126,84	-	-	-	59,02

Z 6.3 je zřejmé, že základní verze implementace umožňuje vyřešit v rozumném čase pouze malé problémy. Ve zbylých problémech se vyskytly problémy: dlouhá doba výpočtu a případné zacyklení. Spousta problémů se také jevila jako neřešitelná, protože po první fázi byla účelová hodnota blízká nule, ale ne nulová, případně v bázi zůstaly nenulové umělé proměnné. Průběžné výpisy byli prováděny každých 1000 iterací programu. Pokud byl počet iterací vyšší než 30 000 byl výpočet ukončen. Dále pokud se hodnota počtu

umělých proměnných v bázi v první fázi algoritmu neměnila po několik tisíc iterací, bylo uvažováno zacyklení řešení (v tabulkách označeno jako cykl(#iterace 1. výskytu)).

Z tabulky 6.3 je zřejmé, že základní verze implementace si s většími problémy neporadila. Je tedy nutné rozšířit program tak, aby eliminoval či potlačil hlavní problémy. Prvním citelným problémem je cyklení algoritmu. Dalším problémem je numerická nestabilita algoritmu, která způsobovala například výběr neplatných pivotů. Posledním velkým problémem je zvyšování délky výpočtu jednotlivých iterací, způsobené postupným zaplňováním inverzní bázové matice.

6.2.1 Cyklení

V první verzi algoritmu byl výběr sloupce vstupní proměnné prováděn na základě nejnižší redukované ceny. Tento způsob může při degenerovaném řešení vést k zacyklení, tedy k opakovanému návratu k jednomu bázovému řešení.

Aby se předešlo tomuto zacyklení bylo implementováno Blandovo pravidlo, které má garantované ukončení algoritmu. Jeho nevýhodou ovšem je potenciálně značný počet iterací, které je třeba provést před nalezením optima. Počet iterací při aplikování Blandova pravidla je z principu větší než při výběru minimální redukované ceny, proto byla implementována úprava, kdy je při výběru proměnné s malou redukovanou cenou zvolena proměnná s minimální redukovanou cenou.

6.2.2 Numerická nestabilita

Největším identifikovaným problémem při implementaci představovala numerická nestabilita navrženého algoritmu.

Prvním příznakem numerické nestability byl výběr vstupní proměnné k výběru proměnné s redukovanou cenou blízkou nule, případně příliš malé nebo naopak příliš velké hodnoty při poměrovém testu. Tento problém byl vyřešen zaokrouhlováním malých hodnot (řádově v absolutní hodnotě menších než 10^{-7}) ve všech krocích algoritmu na nulu.

Dalším opatřením potlačujícím numerickou nestabilitu je implementace škálování vstupních hodnot postupem popsaným v kapitole 5.2.2.1. Ta zamezí výskytu příliš velkého rozdílu v řádů v absolutní hodnotě maximálního a minimálního koeficientu od 0.

6.2.3 Zvyšování délky výpočtu iterace

Tento problém byl detekován ve fázi výpočtu simplexového multiplikátoru $\boldsymbol{\pi}$ 5.2.4.5 a při aktualizaci inverzní bázové matice $\boldsymbol{B^{-1}}$ 5.2.4.14, přičemž tyto dva kroky měly společné násobení $\boldsymbol{B^{-1}}$. Po vypsání této matice bylo zjištěno, že se v ní po každé iteraci vyskytují prvky nulové a blízké nule. Výskyt nulových hodnot byl dosledován k chybě ve funkci *cusp::multiply()* knihovny CUSP, kdy po výpočtu hodnoty na pozici, kde se dříve vyskytovala nenulová hodnota vznikla hodnota nulová a nebyla odfiltrována. Z důvodu výskytu těchto prvků bylo potřeba implementovat jejich filtraci, pro zrychlení výpočtu jednotlivých iterací.

6.3 Výsledky upravených verzí algoritmu

Po implementaci výše zmíněných úprav spolu s filtrací nulových a nule blízkých prvků z $\boldsymbol{B^{-1}}$ byl program opět otestován na použité sbírce problémů při použití dvou možností výběru pivotu. Prvním způsobem byl, jako v původní verzi bez úprav, výběr podle minimální ceny 6.5 6.7, druhým způsobem byla kombinace Blandova pravidla a výběru podle minimální ceny 6.4 6.6. Tato kombinace spočívá ve výběru podle Blandova pravidla a pouze v případě, kdy má vybraný sloupec malou hodnotu ceny (řádově tisíciny), je vybrán sloupec s minimální redukovanou cenou. Touto kombinací byla kompenzována možná numerická nestabilita, kdy Blandovo pravidlo vybíralo redukované ceny blízké nule, které ale ještě nebyly pod hranicí zaokrouhlení na nulu.

Tabulka 6.4: Výsledky po úpravách s Blandovým pravidlem na GTX570

Název	Stav	#iterací 1.řeš	1.řeš[s]	# iterací optima	Optimum	Optimum [s]	Nejdelší iterace[ms]
25fv47	ukončeno	30 000+	-	-	-	-	12,0
80bau3b	neřešitelné	15701	1986,0	-	-	-	492,0
adlittle	vyřešené	265	1,4	62	2,255E+05	0,3	6,0
afiro	vyřešené	38	0,2	0	-4,648E+02	0,0	6,1
agg	neřešitelné	874	9,1	-	-	-	12,1
agg2	vyřešené	668	7,3	172	-2,024E-07	2,0	13,0
agg3	vyřešené	680	7,4	208	1,031E+07	2,3	12,5
bandm	ukončeno	30 000+	-	-	-	-	14,2
beaconfd	vyřešené	343	1,8	61	3,359E+04	1,6	6,5
blend	vyřešené	584	3,2	32	-3,080E+01	0,2	6,5
bnl1	neřešitelné	19894	783,2	-	-	-	52,8
bnl2	ukončeno	30 000+	-	-	-	-	80,4
bore3d	ukončeno	30 000+	-	-	-	-	9,4
brandy	ukončeno	30 000+	-	-	-	-	7,4
capri	vyřešené	789	8,9	215	2,690E+03	3,1	16,8
cycle	ukončeno	30 000+	-	-	-	-	17,2
czprob	vyřešené	4992	61,4	10353	2,185E+06	146,0	17,8
d2q06c	neřešitelné	27897	4398,2	-	-	-	308,7
d6cube	ukončeno	30 000+	-	-	-	-	20,6
degen2	ukončeno	30 000+	-	-	-	-	18,9
degen3	ukončeno	30 000+	-	-	-	-	28,0
dfl001	neřešitelné	23330	6220,0	-	-	-	519,3
e226	ukončeno	30 000+	-	-	-	-	10,5
etamacro	ukončeno	30 000+	-	-	-	-	30,1

Tabulka 6.5: Výsledky po úpravách s výběrem minima na GTX570

Název	Stav	#iterací 1.řeš	1.řeš[s]	# iterací optima	Optimum	Optimum [s]	Nejdelší iterace[ms]
25fv47	neřešitelné	18109	1352	-	-	-	87,8
80bau3b	neřešitelné	10181	388	-	-	-	84,7
adlittle	vyřešeno	100	0	6	2,255E+05	0,1	6,2
afiro	vyřešeno	27	0	0	-4,648E+02	0,0	5,5
agg	vyřešeno	677	7	54	-3,599E+07	0,5	10,6
agg2	vyřešeno	555	6	43	-2,024E+07	0,5	12,2
agg3	vyřešeno	566	6	55	1,031E+07	0,6	12,9
bandm	ukončeno	30 000+	-	-	-	-	14,6
beaconfd	vyřešeno	209	1,1	33	3,359E+04	0,2	6,4
blend	vyřešeno	127	0,7	19	-3,081E+01	0,1	6,3
bnl1	neřešitelné	6191	219,6	-	-	-	52,5
bnl2	ukončeno	30 000+	-	-	-	-	93,8
bore3d	ukončeno	30 000+	-	-	-	-	7,7
brandy	vyřešeno	543	4	74	1,519E+03	0,7	10,5
capri	neřešitelné	1147	18	-	-	-	18,4
cycle	ukončeno	30 000+	-	-	-	-	17,3
czprob	vyřešeno	1644	18,8	786	2,185E+06	10,5	16,7
d2q06c	neřešitelné	12399	2009,4	-	-	-	282,8
d6cube	ukončeno	30 000+	-	-	-	-	41,0
degen2	ukončeno	30 000+	-	-	-	-	28,0
degen3	ukončeno	30 000+	-	-	-	-	120,2
dfl001	neřešitelné	3931	831,0	-	-	-	513,6
e226	vyřešeno	412	2,3	397	-1,862E+01	3,1	7,4
etamacro	neřešitelné	926	9,3	-	-	-	40,6

Tabulka 6.6: Výsledky po úpravách s Blandovým pravidlem na GTX TITAN

Název	#iterací 1.řeš	1.řeš[s]	#iterací optima	Optimum	Optimum[s]	Nejdelší iterace[ms]
adlittle	265	3,0	62	2,255E+05	0,7	11,8
afiro	38	0,4	0	-4,648E+02	0,0	10,1
agg2	668	13,9	172	-2,024E+07	3,7	23,3
agg3	680	14,2	208	1,031E+07	4,4	24,3
beaconfd	343	3,4	61	3,359E+04	0,7	11,9
blend	584	6,4	32	-3,081E+01	0,4	13,2
capri	789	14,4	215	2,690E+03	5,8	30,2
czprob	4992	110,5	10353	2,185E+06	279,4	35,3

Tabulka 6.7: Výsledky po úpravách s výběrem minima na GTX TITAN

Název	#iterací 1.řeš	1.řeš[s]	#iterací optima	Optimum	Optimum[s]	Nejdelší iterace[ms]
adlittle	100	1,0	6	2,255E+05	9,9	11,8
afiro	27	0,2	0	-4,648E+02	0,0	9,3
agg	677	11,9	54	-3,599E+07	0,9	19,0
agg2	555	10,4	43	-2,020E+07	0,8	20,3
agg3	566	10,7	55	1,030E+07	1,0	21,6
beaconfd	209	2,3	36	3,359E+04	0,4	16,3
blend	127	1,3	19	-3,081E+01	0,2	11,3
brandy	543	6,9	74	1,519E+03	1,3	19,2
czprob	1644	30,8	786	2,185E+06	18,9	31,6

Z tabulek je patrné, že došlo ke zlepšení stability a výpočetních časů. U většiny předem ukončených výpočtů byla zřejmá konvergence k řešení, bohužel velmi pomalu. U neřešitelných případů došlo k tomu, že vlivem nestability byly ve výsledku první fáze obsaženy umělé proměnné, které neměli nulovou hodnotu, ale hodnotu blízkou nule, což způsobilo klasifikaci problému jako neřešitelného, případně hodnota účelové funkce nebyla nulová ani blízká nule.

Zpomalení výpočtu jednotlivých iterací je jednoznačně způsobeno přibývajícími prvky v matici $\boldsymbol{B^{-1}}$. Sparse formát této matice tedy snižuje paměťové nároky, ale jelikož se tato matice s rostoucím počtem iterací postupně zaplňuje, rostou výpočetní časy kroků algoritmu, které s ní pracují. Zároveň úprava zaokrouhlování a filtrování hodnot umožnila v některých případech dosáhnout správného výsledku, v jiných případech ale způsobila akumulace nepřesností způsobených častým zaokrouhlováním ke klasifikaci problému jako neřešitelného.

Z tabulek 6.7, 6.6 je patrný nárůst výpočetních časů GPU2 oproti GPU1, který je způsobený rozdílnou rychlostí výpočetních jednotek (837MHz vs. 1464MHz). Použitý algoritmus násobení sparse matic předpočítává počet a pozici nenulových prvků, tato operace je závislí především na rychlosti shader jednotek. Z tab.6.1 je možné vidět, že v tomto ohledu GPU1 předčí GPU2, jelikož má frekvenci shader jednotek téměř dvojnásobnou.

Kapitola 7

Závěr

V průběhu této práce byl vytvořen program řešící úlohy lineárního programování využívající výpočetního výkonu GPU. Sekvenční simplexový algoritmus byl paralelizován na úrovni jednotlivých operací a základní implementace algoritmu byla otestována na souboru problémů 'NETLIB LP'.

Z výsledků bylo patrné, že program nalezl správné řešení v daném limitu iterací pouze pro relativně malé problémy, v případě větších problémů byly identifikovány problémy se zacyklením, numerickou nestabilitou a zpomalením při pozdějších iteracích z důvodu nárůstu počtu nenulových prvků v inverzní bázové matici, přičemž je možné, že se tato matice stane až dense a tudíž se její uložení ve sparse formátu stane neefektivním. Část z těchto problémů byla potlačena implementací Blandova pravidla v kombinaci s pravidlem výběru minimální redukované ceny, zaokrouhlováním ve všech operacích, škálováním vstupního problému a implementací filtrace inverzní bázové matice.

Následně byly porovnány výsledky programu s implementovaným kombinovaným Blandovým pravidlem a výběrem minimální redukované ceny. Z výsledků bylo patrné, že u některých problémů došlo k potlačení výše uvedených nepříznivých jevů a došlo k nalezení správného řešení, ale u více než poloviny problémů stále přetrval problém s klasifikací problému jako neřešitelného. Jediné co se u těchto případů zlepšilo byly výpočetní časy jednotlivých iterací.

Například implementace kombinovaného Blandova pravidla přinesla značné navýšení počtu iterací, které způsobilo zpomalení v krocích algoritmu, kde je použita inverzní bázová matice. U výběru vstupní proměnné pomocí minimální redukované ceny bylo možné vidět rychlejší konvergenci ke konečnému řešení, tedy některé problémy, které předchozí výběr nedokázal v rozumném čase vyřešit byly vyřešeny.

Při testování implementace na dvou různých grafických kartách byly zjištěny delší

výpočetní časy na výkonnějším GPU GTX Titan. Tyto rozdíly byly způsobeny rozdílnou frekvencí shader jednotek obou testovaných grafických karet. Pro plné využití potenciálu GPU GTX Titan by bylo nutné testovat implementaci na větších problémech, kde by se projevil zvýšený počet pomalejších shader jednotek.

Jelikož bylo zjištěno, že heuristika pro výběr pivotu má zásadní vliv na rychlost získání výsledku, měla by se budoucí práce zaměřit na nalezení heuristiky, která dokáže rychle a efektivně zkonvergovat k řešení bez rizika zacyklení, které poskytuje Blandovo pravidlo. Dále by bylo vhodné zaměřit se na efektivnější uložení inverzní bázové matice a především způsob její aktualizace po každé iteraci, který by zredukoval výpočetní čas. Dalšími body, které by zredukovaly výpočetní čas a zároveň vyřešily některé problémy jsou počáteční předzpracování souboru omezení daného problému a také využití slackových proměnných, příslušejících omezení typu '$\leq$' efektivnější nalezení počáteční báze.

Literatura

[1] http://www.biomedcentral.com/content/supplementary/1756-0500-2-73-s3.png.

[2] http://www.ixbt.com/video3/images/cuda/cuda3.png.

[3] NVIDIA. *NVIDIA CUDA Programming Guide 5.5*, 2014. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[4] Tomáš Werner. Optimalizace, 2014. https://cw.felk.cvut.cz/wiki/_media/courses/a4b33opt/opt.pdf.

[5] Enric Rodriguez-Carbonell Javier Larrosa, Albert Oliveras. The revised simplex method. http://www.lsi.upc.edu/~erodri/webpage/cps/theory/lp/revised/slides.pdf.

[6] NVIDIA Corporation. *Thrust*. http://docs.nvidia.com/cuda/thrust/#axzz31QBwCwzo.

[7] NVIDIA Corporation. *cuSPARSE*. http://docs.nvidia.com/cuda/cusparse/#axzz31QBwCwzo.

[8] NVIDIA Corporation. *CUSP*. https://code.google.com/p/cusp-library/wiki/QuickStartGuide.

[9] David Galvin. Simplex method - summary. https://www3.nd.edu/~dgalvin1/30210/30210_F07/presentations/simplex_full.pdf.

[10] N. Bell and M. Garland. The final netlib-lp results. Technical report, Konrad-Zuse-Zentrum fur Informationstechnik Berlin, 2003. http://www.zib.de/Publications/Reports/ZR-03-05.pdf.

Příloha A

Obsah přiloženého CD

1. Text bakalářské práce ve formátu pdf.
2. Projekt ve Visual Studiu 2012 spolu se všemi použitými soubory.
3. Soubory knihovny CUSP.
4. Jednoduchý návod na zprovoznění.