FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Paralelní algoritmy pro maximální toky v sítích na architektuře CUDA
Student:	Kamil Červený
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

1. Nastudujte a proveďte rešerši algoritmů hledajících maximální toky v sítích (Fordův–Fulkersonův, Edmondsův–Karpův, Dinicův, Goldbergův a jeho vylepšení Ahuja-Orlin) [1,2]

2. Navrhněte efektivní paralelizace Goldbergova a Ahuja-Orlin algoritmu pomocí architektury CUDA.

3. Navrhnuté algoritmy implementujte, otestujte a porovnejte naměřené zrychlení pro různé typy a velikosti sítí.Porovnejte váš přístup s publikací [3] a jinými open-souce knihovnami.

Seznam odborné literatury

[1] GOLDBERG, Andrew; TARJAN, Robert. A New Approach to the Maximum-Flow Problem. J. ACM. 1988, roč. 35, s. 921–940. Dostupné z DOI:
10.1145/48014.61051.;

[2] K. AHUJA, Ravindra; ORLIN, James. A Fast and Simple Algorithm for the Maximum Flow Problem. Operations Research. 1989, roč. 37. Dostupné z
DOI: 10.1287/opre.37.5.748

[3] WU, Jiadong; HE, Zhengyu; HONG, Bo. GPU Computing Gems Jade Edition. Applications of GPU Computing Series. 2012, s 55-66. Dostupné z DOI: 10.1016/B978-0-12-385963-1.00005-8.

doc. Ing. Jan Janoušek, Ph.D. vedoucí katedry	doc. RNDr. Ing. Marcel Jiřina, Ph.D. děkan

V Praze dne 14. ledna 2020

Diplomová práce

Paralelní algoritmy pro maximální toky v sítích na architektuře CUDA

Bc. Kamil Červený

Katedra teoretické informatiky
Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

12. února 2021

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu práce panu docentovi Ivanovi Šimečkovi, který nade mnou nezlomil hůl navzdory mé do nebe volající neschopnosti a sílícímu úpadku morálky doprovázejícího konec studia umocněného současnou pandemií. Dále chci poděkovat svým spolustudentům, které jsem za dobu studia na fakultě poznal, bez jejichž vzájemné pomoci by pro mě studium bylo o mnoho náročnější. Nakonec chci zmínit pana Pavla Korbeláře z ÚTVS, ke kterému jsem poctivě každý semestr svého bakalářského i magisterského studia rád docházel na tělocvik, kde jsem si každou středu nebo čtvrtek mohl ode všeho při hraní badmintonu odpočinout.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 12. února 2021 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Kamil Červený. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Červený, Kamil. *Paralelní algoritmy pro maximální toky v sítích na architektuře CUDA*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Práce se zabývá možnostmi paralelizace algoritmů hledajících maximální tok v síti pro masivně paralelní architekturu CUDA. Na úvod práce je zaveden nezbytný teoretický základ z teorie grafů a algoritmů, na který navazuje představení existujících sekvenčních a masivně paralelních algoritmů, které vycházejí výhradně z původního sekvenčního Goldbergova algoritmu. Dále je uveden nový masivně paralelní algoritmus využívající obarvení hran sítě. Uvedené paralelní algoritmy jsou implementovány v programovacím jazyce `C++` rozšířeném o CUDA. Na závěr práce je výkonnost implementovaných algoritmů porovnána na sadě různých typů sítí.

Klíčová slova maximální tok v síti, CUDA, GPU, masivně paralelní algoritmus, paralelní BFS, Goldbergův algoritmus, `C++`

Abstract

This thesis explores the possibilities of parallelizing algorithms finding maximum flow in a network for massively parallel CUDA architecture. At the beginning of the work, a necessary theoretical basis from the graph and algorithm theory is introduced, followed by the introduction of existing sequential and massively parallel algorithms based exclusively on the original sequential Goldberg algorithm. Next, a new massively parallel algorithm using network edge coloring is introduced. All these parallel algorithms are implemented in the `C++` programming language extended by CUDA. At the end of the work, the performance of implemented algorithms is compared on a set of different types of networks.

Keywords maximum flow in network, CUDA, GPU, massively parallel algorithm, parallel BFS, Goldberg algorithm, `C++`

Obsah

Úvod **1**
Struktura práce . . . 1

1 Základní pojmy a poznatky **3**
1.1 Základní definice . . . 3
1.2 Hlavní věta o tocích . . . 6

2 Sekvenční algoritmy **11**
2.1 Fordův-Fulkersonův algoritmus . . . 11
2.1.1 Edmonsův-Karpův algoritmus . . . 13
2.2 Dinitzův algoritmus . . . 13
2.3 Goldbergův algoritmus . . . 18
2.3.1 Vylepšení výběrem nejvyššího vrcholu . . . 24
2.3.2 Vylepšení škálováním přebytků . . . 25
2.3.3 Heuristika globálního přepočítání výšek . . . 26
2.3.4 Minimální řez a velikost maximálního toku . . . 27

3 Paralelní algoritmy na architektuře CUDA **29**
3.1 Základní pojmy složitosti paralelních algoritmů . . . 29
3.2 Výběr vhodného algoritmu pro paralelizaci . . . 30
3.3 Metoda pulzu . . . 30
3.4 Paralelní algoritmy pro CPU . . . 32
3.5 Představení technologie CUDA . . . 32
3.5.1 Divergence warpu . . . 33
3.5.2 Atomické operace . . . 34
3.6 Paralelní BFS na CUDA . . . 36
3.7 Existující algoritmy hledající maximální tok . . . 37
3.7.1 Synchronizace atomickými operacemi . . . 37
3.7.1.1 Vlákna přiřazována vrcholům . . . 37

3.7.1.2 Vlákna přiřazována hranám 39
3.7.2 Synchronizace fázemi 41
3.7.2.1 Statické převádění přebytků 41
3.7.2.2 Dynamické převádění přebytků 43
3.8 Nový algoritmus využívající obarvení hran 44
3.8.1 Důkaz správnosti 45

4 Detaily implementace 49
4.1 Technologie 49
4.2 Reprezentace datových struktur v paměti 49
4.2.1 Graf 49
4.2.2 Lattice 50
4.3 Implementace algoritmů 50
4.3.1 Paralelní globální přepočítání výšek 51
4.4 Parametry algoritmů 52

5 Testování a měření algoritmů 53
5.1 Vstupní instance 53
5.1.1 Sítě DIMACS 53
5.1.2 Sítě KaHIP 53
5.1.3 Sítě z počítačového vidění 54
5.2 Testování správnosti 54
5.3 Měření 55
5.3.1 Prostředí 55
5.3.2 Výsledky měření 56
5.3.2.1 Algoritmus využívající *lattice* 56
5.3.2.2 Algoritmus využívající obarvení hran 56

Závěr 59

A Seznam použitých zkratek 61

B Obsah přiloženého flash disku 63

Literatura 65

Seznam obrázků

3.1 Schématický rozdíl mezi tranzistory na CPU a GPU 33
3.2 Hierarchie vláken v CUDA pro dvoudimenzionální mřížku a bloky 34
3.3 Hierarchie paměti v CUDA 35

Seznam tabulek

2.1 Asymptotická časová složitost uvedených sekvenčních algoritmů hledajících maximální tok 28

3.1 Porovnání hlavních vlastností uvedených paralelních algoritmů – 1. typ synchronizace vláken při přístupu do sdílené paměti, 2. způsob přiřazování vláken při převádění přebytků a 3. jak se lokálně mění výšky vrcholů 48

5.1 Přehled parametrů vstupních instancí 54
5.2 Detaily struktury *lattice* během vykonávání algoritmu 3.7.10 57
5.3 Detaily obarvování hran jednotlivých sítí v algoritmu 3.8.1 57
5.4 Přehled naměřených časů běhu v sekundách 58

Úvod

Tématem této práce jsou algoritmy pro hledání maximálního toku v síti na architektuře CUDA, jedná se tedy o masivně paralelní řešení tohoto důležitého problému oboru informatiky a teorie grafů. Sekvenčních algoritmů hledajících maximální tok existuje několik a některé jsou známy již desítky let. Nacházení stále lepších algoritmlů vedlo k čím dál lepší asymptotické časové složitosti, i když nižší časová složitost nevede vždy nutně k rychlejšímu běhu algoritmu v reálném světě na reálném počítači. Proto stojí za úvahu pro některé z nich využít i různé heuristiky snižující jejich reálnou dobu běhu. V dnešní době se vzhledem k dostupnosti GPU nabízí otázka, jak takové algoritmy paralelizovat a dosáhnout jejich ještě rychlejšího běhu využitím výpočetního systému s masivně paralelní architekturou, jako je například CUDA. Existující paralelní algoritmy pro CPU i GPU jsou výhradně variantami původního sekvenčního Goldbergova algoritmu [6] díky vlastnostem jeho interních operací.

Nalezení maximálního toku v síti má mnoho aplikací v reálném světě například při plánování zdrojů nebo zjišťování propustnosti železniční, silniční nebo vodovodní sítě. Některé problémy lze dokonce na problém nalezení maximálního toku převést, což se využívá například v počítačovém vidění při segmentaci obrazu, proto existují implementace speciálně navrženy pouze pro určitý typ sítě, tato práce se ovšem zabývá algoritmy pro obecné různorodé typy sítí.

Struktura práce

Tato práce se dělí na na sebe logicky navazující kapitoly, nejdříve je uveden nezbytný teoretický základ zejména z teorie grafů spolu s některými existujícími sekvenčními algoritmy hledajícími maximální tok, na které navazuje uvedení paralelních algoritmů pro architekturu CUDA spolu s návrhem jednoho nového. Závěrečné dvě kapitoly se zabývají implementací uvedených algoritmů a porovnání jejich výkonosti na různých typech sítí.

Kapitola 1

Základní pojmy a poznatky

V této kapitole jsou uvedeny definice základních pojmů z teorie grafů spolu se souvisejícími poznatky a větami, které budou dále v práci využívány. Konkrétnější pojmy související s jednotlivými algoritmy budou uvedeny v příslušných dalších kapitolách. Uvedené definice a věty jsou převzaty z [1] a [2].

1.1 Základní definice

Definice 1.1.1 (Orientovaný graf)**.** *Orientovaným grafem* rozumíme uspořádanou dvojici (V, E), kde V je neprázdná konečná množina vrcholů a E je množina hran. *Hrana* v orientovaném grafu je uspořádaná dvojice (u, v), kde $u, v \in V$. *Výstupní stupeň* vrcholu $u \in E$ definujeme jako $\text{DEG}^{\text{OUT}}(u) := |\{v : (u, v) \in E\}|$. △

Definice 1.1.2 (Orientovaná cesta)**.** *Orientovanou cestou* z vrcholu v_0 do vrcholu v_n v orientovaném grafu (V, E) rozumíme konečnou posloupnost $(v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n)$, kde $v_0, \ldots, v_n \in V$ jsou různé vrcholy a pro každé i platí $e_i = (v_{i-1}, v_i) \in E$. *Délkou* takové cesty rozumíme počet hran e_i v cestě, pokud existuje, pokud neeixstuje, její délka je ∞. *Vzdáleností* vrcholů v_0 a v_n rozumíme délku nejkratší cesty z v_0 do v_n a značíme ji $d(v_0, v_n)$. △

Definice 1.1.3 (Síť)**.** *Síť* je uspořádaná pětice (V, E, z, s, c), kde:

- (V, E) je orientovaný graf,
- $c : E \to \mathbb{R}_0^+$ je funkce přiřazující hranám jejich kapacity,
- $z, s \in V$ jsou dva různé vrcholy nazývané *zdroj* a *stok* (*spotřebič*). △

Předpokládáme, že síť je souvislý graf a pro každou její hranu $(u, v) \in E$ existuje i hrana opačná $(v, u) \in E$. Toho lze docílit bez újmy na obecnosti tím, že pokud by nějaká opačná hrana v síti chyběla, přidáme ji do sítě s nulovou kapacitou.

Definice 1.1.4 (Tok)**.** *Tok* v síti je funkce $f : E \to \mathbb{R}_0^+$, pro kterou platí:

- $\forall e \in E : f(e) \leq c(e)$, neboli tok po hraně je shora omezen její kapacitou,
- *Kirchoffův zákon*:

$$\forall v \in V \backslash \{z, s\} : \sum_{u:(u,v)\in E} f((u,v)) = \sum_{u:(v,u)\in E} f((v,u)),$$

  neboli do každého vrcholu kromě zdroje a stoku přitéká stejné množství, které z něj odtéká. △

Pro sumy podobné těm v Kirchhoffově zákoně zadefinujeme následující pohodlné značení.

Definice 1.1.5 (Přítok, odtok, přebytek)**.** Pro libovolnou funkci $f : E \to \mathbb{R}$ definujeme:

- $f^+(v) := \sum_{u:(u,v)\in E} f((u,v))$ – celkový *přítok* do vrcholu v,
- $f^-(v) := \sum_{u:(v,u)\in E} f((v,u))$ – celkový *odtok* z vrcholu v,
- $f^\Delta(v) := f^+(v) - f^-(v)$ – *přebytek* ve vrcholu v. △

Z Kirchhoffova zákona tedy plyne, že pro všechny vrcholy kromě zdroje a stoku platí $f^\Delta(v) = 0$.

Definice 1.1.6 (Velikost toku)**.** *Velikost toku* f definujeme jako $|f| := f^\Delta(s)$, neboli jaké množství přitéká do stoku. △

Lemma 1.1.1. Pro každou síť (V, E, z, s, c) a tok f v ní platí

$$f^\Delta(s) = -f^\Delta(z).$$

Důkaz: Označme $S = \sum_{v\in E} f^\Delta(v)$. Podle Kirchhoffova zákona může být přebytek ve vrcholu nenulový pouze pro zdroj a stok, takže $S = f^\Delta(z)+f^\Delta(s)$. Zároveň je tato suma rovna nule, protože každá hrana k ní přispěje jednou kladně (ve vrcholu do kterého vede) a jednou záporně (ve vrcholu ze kterého vede). □

Definice 1.1.7 (Řez)**.** *Řezem* mezi zdrojem z a stokem s v síti (V, E, z, s, c) nazveme množinu hran $R \subseteq E$ takovou, že v síti $(V, E\backslash R, z, s, c)$ neexistuje žádná orientovaná cesta ze zdroje do stoku. *Kapacitou řezu* rozumíme $c(R) := \sum_{e\in R} c(e)$. △

Řez je důležitým pojmem z oblasti toků, který v následující sekci dá do souvislosti jeho kapacitu a velikost toku v síti. Uvědomme si, že toků v libovolné síti může být nekonečně mnoho, takže nelze triviálně tvrdit, že existuje nějaký maximální (existují nekonečné množiny nemající maximum). Následující lemma však říká, že tomu tak skutečně je.

Věta 1.1.1 (O existenci maximálního toku)**.** Pro každou síť existuje maximální tok.

Důkaz: Nechť $S = (V, E, z, s, c)$ je síť. Označme množinu $W = \{|f| : f$ je tok v síti $S\}$ a $M = sup(M)$ supremum této množiny.

Protože funkce přiřazující každé hraně 0 je tok, platí $M \geq 0$. Zároveň platí, že každý tok má nejvýše velikost $c(\{z\}, V\backslash\{z\}) = \sum_{(z,x)\in E} c((z,x)) \leq \infty$, takže $M \leq \infty$. Stačí tedy ukázat, že existuje tok velikost M, ten je pak jistě maximální.

Protože M je supremum W, tak pro každé $\varepsilon > 0$ existuje tok f takový, že $|f| \geq M - \varepsilon$, neboli existuje posloupnost toků $(f_n)_{n=1}^{\infty}$ taková, že $\lim_{n\to\infty} |f_n| = M$. Z této posloupnosti nyní vybereme podposloupnost $(f_n^{'})_{n=1}^{\infty}$ takovou, že $(f_n^{'}(e))_{n=1}^{\infty}$ je konvergentní pro každou hranu $e \in E$.

Zvolme nějaké pevné pořadí všech hran sítě $e_1, e_2, e_3, \ldots, e_{|E|}$ a položme $(f_n^{(0)})_{n=1}^{\infty} = (f_n)_{n=1}^{\infty}$ pro všechna $i \in \mathbb{N}$. Nyní budeme postupně pro $j = 1, 2, 3, \ldots, |E|$ z posloupnosti toků $(f_n^{(j-1)})_{n=1}^{\infty}$ vybírat podposloupnost toků $(f_n^{(j)})_{n=1}^{\infty}$ tak, aby $(f_n^{(j)}(e_j))_{n=1}^{\infty}$ byla konvergentní. Neboli pro každou hranu vybereme konvergentní podposloupnost z předchozí. To lze udělat, protože $(f_n^{(j-1)}(e_j))_{n=1}^{\infty}$ je zdola omezená nulou a shora $c(e_j)$, takže vybíráme konvergentní podposloupnost z omezené posloupnosti. Poté položíme $(f_n^{'})_{n=1}^{\infty} = (f_n^{(|E|)})_{n=1}^{\infty}$. Tím docílíme toho, že $(f_n^{'}(e_j))_{n=1}^{\infty}$ je pro každé $j = 1, 2, 3, \ldots, |E|$ konvergentní, neboli existuje $\lim_{n\to\infty} f_n^{'}(e)$ pro každou hranu $e \in E$.

Definujeme funkci $g : E \to \mathbb{R}$ předpisem $g(e) = \lim_{n\to\infty} f_n^{'}(e)$ a ukážeme, že g je tok. Pro spor předpokládáme nejprve, že existuje hrana $e \in E$, pro kterou $g(e) > c(e)$. Pak existuje $n_0 \in \mathbb{N}$ takové, že pro všechny $n > n_0$ platí $f_n^{'}(e) > c(e)$, což je spor s tím, že $f_n^{'}$ je tok. Obdobně pokud $g(e) < 0$, tak existuje $n_0 \in \mathbb{N}$ takové, že pro všechny $n > n_0$ platí $f_n^{'}(e) < 0$, což je také spor s tím, že $f_n^{'}$ je tok. Dále pro spor předpokládáme neplatnost Kirchhoffova zákona, tedy že existuje vrchol $u \in V, u \neq z, s$ splňující

$$\sum_{(x,u)\in E} g((x,u)) \neq \sum_{(u,y)\in E} g((u,y)).$$

Pak mají posloupnosti $\left(\sum_{(x,u)\in E} f_n^{'}((x,u))\right)_{n=1}^{\infty}$ a $\left(\sum_{(u,y)\in E} f_n^{'}((u,y))\right)_{n=1}^{\infty}$ různé limity. Takže existuje $n_0 \in \mathbb{N}$ takové, že pro všechny $n > n_0$ platí

$$\sum_{(x,u)\in E} f_n^{'}((x,u)) > (<) \sum_{(u,y)\in E} f_n^{'}((u,y)),$$

což je v obou případech (< a >) spor s tím, že f_n' je tok. Ukázali jsme tedy, že g je tok. Pro jeho velikost platí

$$\begin{aligned}
|g| &= \sum_{(z,x)\in E} g((z,x)) - \sum_{(x,z)\in E} g((x,z)) = \\
&= \sum_{(z,x)\in E} \lim_{n\to\infty} f_n'((z,x)) - \sum_{(x,z)\in E} \lim_{n\to\infty} f_n'((x,z)) = \\
&= \lim_{n\to\infty} \Big(\sum_{(z,x)\in E} f_n'((z,x)) - \sum_{(x,z)\in E} f_n'((x,z)) \Big) = \\
&= \lim_{n\to\infty} |f_n'| = M,
\end{aligned}$$

takže jsme našli tok g hledané velkosti M. □

1.2 Hlavní věta o tocích

Následující důležitá věta dává do souvislosti toky a řezy v síti. Narozdíl od toků, je počet řezů sítě shora omezen počtem podmnožin množiny hran dané sítě, takže jistě existuje nejaký s nejmenší kapacitou.

Věta 1.2.1 (Hlavní věta o tocích)**.** Pro každou síť se velikost maximálního toku rovná kapacitě minimálního řezu.

Důkaz věty postupně vyplyne z následujících pomocných lemmat.

Definice 1.2.1. Nechť A, B jsou dvě podmnožiny vrcholů nějaké sítě. Pro množinu hran vedoucích z vrcholů A do vrcholů B zavedeme značení:

$$S(A,B) := \{(u,v) : u \in A, v \in B, (u,v) \in E\},$$

hrany vedoucí z B do A do množiny nepatří. Dále:

$$\begin{aligned}
c(A,B) &:= \sum_{e\in S(A,B)} c(e), \\
f(A,B) &:= \sum_{e\in S(A,B)} f(e), \\
f^{\Delta}(A,B) &:= f(A,B) - f(B,A).
\end{aligned}$$

△

Lemma 1.2.1. Rozdělíme-li vrcholy nějaké sítě na dvě množiny A, B tak, že $z \in A$ a $s \in B$, potom $S(A,B)$ je řez dané sítě, takový řez nazveme *elementární řez.*

Důkaz: Sporem. Nechť v síti $(V, E \backslash S(A,B), z, s, c)$ existuje orientovaná cesta z z do s, pak v dané cestě nutně existuje hrana vedoucí z A do B, každá taková hrana ovšem byla ze sítě odstraněna, což je spor. □

Lemma 1.2.2. Každý řez sítě obsahuje jako svou podmnožinu elementární řez.

Důkaz: Nechť R je řez sítě $S_1 = (V, E, z, s, c)$ a množina A obsahuje všechny vrcholy, do kterých vede ze zdroje orientovaná cesta v síti $S_2 = (V, E \backslash R, z, s, c)$. Pak $S(A, V \backslash A) \subseteq R$ (což je elementární řez), protože kdyby nějaká hrana $e = (u, v) \in S(A, V \backslash A)$, kde $u \in A$ a $v \notin A$, a zároveň $e \notin R$, existovala by orientovaná cesta ze zdroje do vrcholu v v síti S_2 a tedy vrchol $v \in A$, což neplatí. □

Lemma 1.2.3. Pro každý řez R sítě platí, že pokud $R \backslash \{e\}$ není řez pro žádnou hranu $e \in R$, tak R je elementární.

Důkaz: Plyne jednoduše z předcházejícího lemmatu 1.2.2. □

Lemma 1.2.4. Nechť $A \subseteq V$ je podmnožina vrcholů nějaké sítě obsahující zdroj a ne stok a f je libovolný tok v této síti, pak platí $|f| = f(A, V \backslash A) - f(V \backslash A, A)$.

Důkaz: Nechť A je libovolná množina a f tok splňující předpoklady. Pro každý vrchol u z A mimo zdroj platí Kirchhoffův zákon

$$\sum_{(u,x)\in E} f((u,x)) - \sum_{(x,u)\in E} f((x,u)) = 0,$$

pro zdroj platí

$$\sum_{(z,x)\in E} f((z,x)) - \sum_{(x,z)\in E} f((x,z)) = |f|.$$

Sečtením levých a pravých stran obou rovnic do jedné dostáváme

$$\sum_{u\in A} \Big(\sum_{(u,x)\in E} f((u,x)) - \sum_{(x,u)\in E} f((x,u)) \Big) = |f|.$$

Levou stranu upravíme na

$$\begin{aligned} &\sum_{(u,x)\in E, u\in A} f((u,x)) - \sum_{(x,u)\in E, u\in A} f((x,u)) = \\ &= \sum_{(u,x)\in E, u\in A, x\in A} f((u,x)) - \sum_{(x,u)\in E, u\in A, x\in A} f((x,u)) + \\ &+ \sum_{(u,x)\in E, u\in A, x\notin A} f((u,x)) - \sum_{(x,u)\in E, u\in A, x\notin A} f((x,u)), \end{aligned}$$

kde první dvě sumy jsou stejné jen s opačnými znaménky, takže dostáváme

$$\sum_{(u,v)\in E, u\in A, v\notin A} f((u,v)) - \sum_{(v,u)\in E, u\in A, v\notin A} f((v,u)) = |f|,$$

což je přesně $f(A, V\backslash A) - f(V\backslash A, A) = |f|$. □

Následující lemma říká část Hlavní věty o tocích – velikost maximálního toku je nejvýše roven kapacitě minimálního řezu.

Lemma 1.2.5. Pro každý tok f a řez R v nějaké síti platí $|f| \leq c(R)$.

Důkaz: Daný řez R obsahuje jako svou podmnožinu podle lemmatu 1.2.2 nějaký elementární řez $S(A, V\backslash A)$ takový, že $z \in A$. Pro velikost toku f potom platí

$$|f| = f(A, V\backslash A) - f(V\backslash A, A) \leq f(A, V\backslash A) \leq c(A, V\backslash A),$$

kde poslední nerovnost platí, protože $f(e) \leq c(e)$ pro všechny hrany sítě. V řezu R mohou být ještě nějaké hrany navíc oproti elementárnímu řezu $S(A, V\backslash A)$, takže dostáváme $|f| \leq c(A, V\backslash A) \leq c(R)$. □

Definice 1.2.2 (Neorientovaná cesta)**.** *Neorientovanou cestou* v síti (V, E, z, s, c) rozumíme konečnou posloupnost $(v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n)$, kde $v_0, \ldots, v_n \in V$ jsou různé vrcholy a pro každé i platí $e_i = (v_{i-1}, v_i) \in E$ nebo $e_i = (v_i, v_{i-1}) \in E$ (ignorujeme orientaci hran). △

Definice 1.2.3 ((Ne)nasycená cesta, zlepšující cesta, nasycený tok)**.** Nechť f je tok v nějaké síti. Neorientovaná cesta v síti se nazývá *nasycená* vzhledem k toku f, pokud v dané cestě existuje hrana (v_{i-1}, v_i) orientovaná po směru cesty splňující $f((v_{i-1}, v_i)) = c((v_{i-1}, v_i))$ a k ní opačná hrana (v_i, v_{i-1}) orientovaná proti směru cesty splňuje $f((v_i, v_{i-1})) = 0$. Pokud taková hrana neexistuje, říkáme cestě *nenasycená*. Nenasycené cestě ze zdroje do stoku budeme říkat *zlepšující cesta*. Tok f nazveme *nasycený*, pokud každá cesta vedoucí ze zdroje do stoku je nasycená. △

Lemma 1.2.6. Tok f v síti je maximální právě tehdy, když je nasycený. Pro každý maximální tok f existuje řez R takový, že $|f| = c(R)$.

Důkaz: Nejdříve dokážeme sporem, že maximální tok je nasycený. Nechť tok f je maximální a nenasycený. Existuje tedy zlepšující cesta P, podle níž nalezneme vylepšení toku f. Zavedeme

$$\begin{aligned}
\varepsilon_1 &:= \text{MIN}\{c(e) - f(e) : e \in P \text{ je orientovaná po směru } P\},\\
\varepsilon_2 &:= \text{MIN}\{f(e) : e \in P \text{ je orientovaná proti směru } P\},\\
\varepsilon_P &:= \text{MIN}\{\varepsilon_1, \varepsilon_2\}.
\end{aligned}$$

Platí, že $\varepsilon_P > 0$ protože P je nenasycená. Tok f změníme na tok f' následujícím způsobem:

$$f'(e) = \begin{cases} f(e) + \varepsilon_P & e \in P \text{ orientovaná po směru } P, \\ f(e) - \varepsilon_P & e \in P \text{ orientovaná proti směru } P, \\ f(e) & e \notin P. \end{cases}$$

S takto zvoleným ε_P se nikde nepřekročí kapacita hrany a $f'(e)$ není nikde záporné. Kirchhoffův zákon stále platí, protože ε_P se pro vrcholy na cestě jednou přičte i odečte, jedná se tedy o validní tok. Pro f' tedy platí $|f'| = |f| + \varepsilon_P > |f|$, což je spor.

Nyní dokážeme, že pokud je tok nasycený, pak je maximální. Zvolme A jako množinu všech vrcholů v, pro které existuje nenasycená cesta ze zdroje do v. Protože f je nasycený, platí $z \in A$ a $s \notin A$. Pro každou hranu $e \in S(A, V \backslash A)$ platí $f(e) = c(e)$ a pro každou hranu $e \in S(V \backslash A, A)$ platí $f(e) = 0$. Platí tedy

$$|f| = f(A, V \backslash A) - f(V \backslash A, A) = c(A, V \backslash A) - 0 = c(A, V \backslash A).$$

Z lemmatu 1.2.5 víme, že $|f| \leq c(R)$ pro jakýkoli řez a tok. Pro řez $R = S(A, V \backslash A)$ dokonce platí $c(R) = |f|$, tok f je tedy určitě maximální a my k němu našli i příslušný řez. □

Tímto důkazem je dokázána i Hlavní věta o tocích 1.2.1 ze začátku této sekce. Víme, že pro každou síť existuje maximální tok a minimální řez o stejné kapacitě. Důkaz také dává jednoduchý algoritmus na hledání maximálního toku v síti, který je představen v další kapitole.

Kapitola **2**

Sekvenční algoritmy

V této kapitole budou představeny některé existující sekvenční algoritmy hledající maximální tok v síti. Na vybrané z nich bude navázáno v další kapitole. Některé algoritmy využívají velmi různé přístupy a jiné jsou „pouze“ vylepšení jejich předchůdců. Algoritmy jsou uvedeny v pořadí, aby na sebe logicky navazovaly. Pomocné definice a lemmata uvedené v jednotlivých sekcích pochazí z [1].

2.1 Fordův-Fulkersonův algoritmus

První představený algoritmus byl publikován v roce 1956 v [3]. Jeho myšlenka spočívá v postupném vylepšování toku začínajícího tokem o velikosti nula, jak bylo naznačeno na konci minulé kapitoly. Dokud existuje nějaká nenasycená cesta ze zdroje do stoku, algoritmus stávající tok podél této cesty vylepší o nejvyšší možnou hodnotu. Pro jeho snadnější popis závádíme následující definici, pseudokód je uveden v algoritmu 2.1.1.

Definice 2.1.1 (Rezerva hrany)**.** Nechť f je tok v síti (V, E, z, s, c), *rezervou hrany* $(u, v) \in E$ vzhledem k toku f rozumíme

$$r((u, v)) := c((u, v)) - f((u, v)) + f((v, u)).$$

△

Lemma 2.1.1. Pokud se Fordův-Fulkersonův algoritmus zastaví, vydá maximální tok.

Důkaz: Nechť (V, E, z, s, c) je síť pro kterou se algoritmus zastaví. Označme množiny vrcholů

$$A := \{v \in V : \text{existuje nenasycená cesta z } z \text{ do } v\} \text{ a } B := V \setminus A.$$

Zdroj z leží v A, protože pro něj existuje cesta nulové délky, která je tedy nenasycená. Stok s leží v B, protože algoritmus skončil, takže na řádku 2 byla podmínka vyhodnocena jako false. Mnzožina hran $S(A,B)$ je tedy řez.

Dále víme, že všechny hrany řezu mají nulovou rezervu. Kdyby totiž pro nějakou $(u,v) \in S(A,B)$ platilo $r((u,v)) > 0$, tak by existovala nenasycená cesta z z do v (spojením nenasycené cesty z z do u a hrany (u,v)), takže pro vrchol v by platilo $v \in A$, což je spor s tím, že $v \in B$.

Takže po všech hranách v $S(A,B)$ teče tok rovný kapacitě hrany a po hranách $S(B,A)$ teče tok rovný nule, tento tok označme f. Tudíž pro řez $S(A,B)$ platí $c(A,B) = f^{\Delta}(A,B)$. Takže jsme našli řez o kapacitě rovné velkosti toku, podle věty 1.2.1 je daný tok maximální. □

Algoritmus 2.1.1: Fordův-Fulkersonův algoritmus

input : síť $S = (V, E, z, s, c)$
output: maximální tok f sítě S

```
1 f ← nulový tok
2 while existuje nenasycená cesta P z z do s do
3     ε ← MIN{r(e) : e ∈ P}                          ◁ o kolik lze zvýšit tok
4     forall (u, v) ∈ P do
5         δ ← MIN{f((v, u)), ε}          ◁ kolik lze odečíst v protisměru cesty
6         f((v, u)) ← f((v, u)) − δ                ◁ snížení toku v protisměru
7         f((u, v)) ← f((u, v)) + ε − δ            ◁ zbytek zvýší tok ve směru
8 return f
```

Lemma 2.1.2. Pro každou síť s racionálními kapacitamy hran se Fordův-Fulkersonův algoritmus zastaví v čase $\mathcal{O}(|\text{MAX}\{f : f \text{ je tok sítě}\}| \cdot |E|)$.

Důkaz: Pro síť (V, E, z, s, c) nejdříve přenásobíme všechny kapacity hran nejmenším společným násobkem jmenovatelů všech kapacit hran ve zlomku v základním tvaru, kapacity tedy budou od teď pouze celá kladná čísla.

Z toho vyplívá, že $\varepsilon \geq 1$, takže v každé iteraci cyklu na řádcích 2 až 7 se f zvýší alespoň o 1. Počet těchto iterací lze shora omezit například sumou $\sum_{(z,u)\in E} c((z,u))$, což je konečné číslo, takže algoritmus jistě skončí. Nenasycenou cestu ze zdroje do stoku lze najít pomocí BFS v čase $\mathcal{O}(|E|)$. Cesta se bude hledat, dokud f není maximální, a zvýší se vždy alespoň o 1, takže algoritmus pracuje v čase $\mathcal{O}(|\text{MAX}\{f : f \text{ je tok sítě}\}| \cdot |E|)$. □

Na druhém řádku algoritmu, kde se vybírá nenasycená cesta, není nijak specifikováno, která se má vybrat, pokud cest existuje víc. Tato skutečnost částečně algoritmus limituje, ten totiž nemusí pro reálné kapacity hran sítě vůbec doběhnout, sami autoři takovou síť uvedli. Tento nedostatek lze obejít jednoduchou úpravou uvedenou v následující podsekci.

2.1.1 Edmonsův-Karpův algoritmus

Pokud by se na druhém řádku algoritmu 2.1.1 namísto libovolné vybírala nejkratší nenasycená cesta ze zdroje do stoku, jednalo by se o algoritmus známý pod jménem „Edmonsův-Karpův“, ten nezávisle na sobě objevili v roce 1970 Yefim Dinitz [5] a o dva roky později Jack Edmons s Richardem Karpem [4]. Tato jednoduchá modifikace asymptoticky zrychluje běh na $\mathcal{O}(|V| \cdot |E|^2)$, ale také zajisťuje, že se algoritmus vždy zastaví a najde maximální tok i pro reálné kapacity hran.

2.2 Dinitzův algoritmus

Podstata Fordova-Fulkersonova algoritmu spočívá v postupném vylepšování toku podél libovolné nenasycené cesty ze zdroje do stoku, to za prvé vůbec nemusí vést k nalezení maximálního toku v síti s reálnými kapacitami hran a za druhé to může trvat velmi dlouho i pro malé jednoduché sítě.

Oba tyto nedostatky řeší Dinitzův algoritmus publikovaný v roce 1970 Yefimem Dinitzem v [5], ten najde maximální tok v libovolné síti a s nižší asymptotickou složitostí. Jeho myšlenka spočívá v tom, že narozdíl od vylepšování toku pomocí cesty, vylepšuje tok pomocí toku. Pro popis algoritmu nejprve zavedeme několik definic usnadňujících jeho popis.

Definice 2.2.1 (Průtok hranou)**.** Nechť f je tok v síti (V, E, z, s, c), *průtokem* hranou $(u, v) \in E$ rozumíme $f^*((u, v)) := f((u, v)) - f((v, u))$. △

Lemma 2.2.1. Průtok hranou (u, v) sítě má následeující vlastnosti:

1. $f^*((u, v)) = -f^*((v, u))$,
2. $f^*((u, v)) \leq c((u, v))$,
3. $f^*((u, v)) \geq -c((v, u))$,
4. pro všechny vrcholy $v \neq z, s$ platí $\sum_{u:(u,v)\in E} f^*((u, v)) = 0$.

Důkaz: Vlastnosti 1 a 2 okamžitě vyplývají po přepsání z definice průtoku. Vlastnost 3 pak vyplývá z prvních dvou. U vlastnosti 4 si stačí přepsat sumu dle definice průtoku a uvědomit si, že se jedná o Kirchhoffův zákon z definice 1.1.4. □

Lemma 2.2.2 (O průtoku)**.** Nechť $S = (V, E, z, s, c)$ je síť a $f^* : E \to \mathbb{R}$ je funkce splňující vlastnosti 1, 2 a 4 Pak existuje tok f v síti S s průtokem f^*.

Důkaz: Stačí pro každou hranu $(u, v) \in E$ a k ní opačnou (v, u) najít tok f. Předpokládejme bez újmy na obecnosti, že pro všechny hrany (u, v) platí $f^*((u, v)) \geq 0$, díky vlastnosti 1 stačí kdyžtak vrcholy u a v prohodit. Nyní

stačí pro každou takovou (u, v) položit $f((u, v)) = f^*((u, v))$ a pro hranu opačnou $f((v, u)) = 0$. Z vlastnosti 2 plyne, že f nikdy nepřekročí kapacitu hrany a z vlastnosti 4 vyplývá platnost Kirchhoffova zákona, takže f je tok. □

Definice 2.2.2 (Síť rezerv)**.** Nechť f je tok v síti $S = (V, E, z, s, c)$. *Sítí rezerv* k toku f rozumíme síť $R(S, f) := (V, E, z, s, r)$, kde $r((u, v))$ je rezerva hrany (u, v) dle definice 2.1.1 pro všechny $(u, v) \in E$. △

Lemma 2.2.3 (O zlepšování toků)**.** Pro každý tok f v nějaké síti $S = (V, E, z, s, c)$ a pro každý tok g v síti $R(S, f)$ lze v čase $\mathcal{O}(|E|)$ nalézt tok h v síti S splňující $|h| = |f| + |g|$.

Důkaz: Pro každou hranu $e \in E$ položme $h^*(e) = f^*(e) + g^*(e)$ a ukažme, že funkce h^* splňuje všechny tři vlastnosti z předpokladu lemmatu 2.2.2. Vlastnost 1 platí okamžitě, protože když platí pro jednotlivé toky f^* a g^*, tak triviálně platí i pro jejich součet. Aby g byl tok sítě $R(S, f)$, tak jistě platí $g^*((u, v)) \leq c((u, v)) - f^*((u, v)) = r((u, v))$ (vzhledem k f), takže přičtením $f^*((u, v))$ k oboum stranám nerovnosti dostáváme $h^*((u, v)) = f^*((u, v)) + g^*((u, v)) \leq c((u, v))$. Tedy vlastnost 2 platí také. Protože vlastnost 4 platí pro toky f^* a g^*, tak jistě pro jejich součet $0 + 0$ platí také.

Dle lemmatu 2.2.2 tedy existuje tok h s průtokem h^*. Pro jeho velikost platí

$$|h| = \sum_{(u,s) \in E} f(u, s) - f(s, u) + \sum_{(u,s) \in E} g(u, s) - g(s, u) = |f| + |g|.$$

Nakonec si stačí uvědomit, že $h^*(e)$ jsme našli jako součet $f^*(e)$ a $g^*(e)$ pro každou hranu $e \in E$ a v důkazu lemmatu 2.2.2 se hledaný tok f našel tak, že každé hraně se v konstantním čase přiřadila jedna ze dvou hodnot. Asymptotická složitost nalezení toku je tedy $\mathcal{O}(|E|)$. □

Definice 2.2.3 (Blokující tok)**.** Nechť f je tok v nějaké síti. Řekneme, že tok f je *blokující*, pokud na každé orientované cestě ze zdroje do stoku existuje alespoň jedna hrana e, pro kterou platí $f(e) = c(e)$. △

Definice 2.2.4 (Vrstevnatá síť)**.** Řekneme, že síť (V, E, z, s, c) je *vrstevnatá* (*pročištěná*), pokud pro všehny hrany $e \in E$ platí, že hrana e leží na nějaké nejkratší cestě z z do s. △

S takto zavedenými definicemi lze chod algoritmu popsat takto. Algoritmus začne s nulovým tokem a ten bude postupně vylepšovat pomocí blokujících toků v síti rezerv dané tokem z předchozí iterace.

Blokující tok budeme hledat jen ve vrstevnaté síti, tedy podsíti tvořené pouze nejkratšími nenasycenými cestami ze zdroje do stoku. Před hledáním blokujícího toku tedy musíme nejdřív síť zbavit hran, které určitě neleží na nějaké nejkratší nenasycené cestě ze zdroje do stoku. To jsou hrany vedoucí uvnitř

jedné vrstvy nebo do nižších vrstev. Dále to jsou „slepé uličky“, což jsou vrcholy, do kterých vede cesta ze zdroje, ale ze kterých neexistuje cesta do stoku.

Blokující tok se pak ve vrstevnaté síti hledá snadno, protože víme, že každá hrana takové sítě s kladnou rezervou leží na nějaké nenasycené cestě ze zdroje do stoku. Stačí tedy postupně zlepšovat nulový tok podél libovolné takové cesty, dokud nějaká existuje. Každé takové vylepšení, lze chápat jako jednu iteraci Fordova-Fulkersonova algoritmu. Při tom je třeba dát pozor na to, že během vylepšování toku, mohly nasycením nějaké hrany vzniknout nové „slepé uličky“, ty je potřeba pokaždé odstranit (pročistit síť) pomocí fronty, aby síť zůstala vrstevnatá.

Tímto postupem se docílí kompromisu, že jedna iterace Dinitzova algoritmu nebude trvat příliš dlouho a zároveň, že hodnota, o kterou se tok zlepší, bude co nejvyšší. Pseudokód je uveden v algoritmu 2.2.1.

Lemma 2.2.4 (O korektnosti)**.** Pokud se Dinitzův algoritmus zastaví, vydá maximální tok.

Důkaz: Z lemmatu o zlepšování toků 2.2.3 plyne, že f je po celou dobu korektní tok. Algoritmus se zastaví, pokud neexistuje nenasycená cesta ze zdroje do stoku (řádek 4), to je stejná podmínka jako u Fordova-Fulkerosonova algoritmu a ten při jejím splnění našel maximální tok. □

Lemma 2.2.5 (O složitosti iterací)**.** Každá iterace Dinitzova algoritmu (řádky 2 až 10) trvá $\mathcal{O}(|V| \cdot |E|)$ času.

Důkaz: Síť je souvislá, takže $|V| = \mathcal{O}(|E|)$. Sestrojení sítě rezerv, mazání hran s nulovou rezervou, hledání nejkratší cesty i konečné zlepšování toku trvají $\mathcal{O}(|E|)$ času.
Při čištění sítě (procedura 2.2.2) včetně čištění během hledání blokujícího toku se každá hrana i vrchol smaže nejvýše jednou, což pro hranu i vrchol trvá konstantní čas. Takže čištení zabere $\mathcal{O}(|V| + |E|) = \mathcal{O}(|E|)$ času.
Při hledání blokujícího toku (procedura 2.2.3) se projde nejvýše $|E|$ cest, protože pro každou nalezenou cestu se v síti nasytí alespoň jedna její hrana, která se odstraní. Hledání cesty ve vrstevnaté síti zabere $\mathcal{O}(|V|)$ času. Celkem hledání blokujícího toku tedy zabere $\mathcal{O}(|V| \cdot |E|)$.
Jedna iterace tedy zabere $\mathcal{O}(|E|) + \mathcal{O}(|E|) + \mathcal{O}(|V| \cdot |E|) = \mathcal{O}(|V| \cdot |E|)$ času. □

Lemma 2.2.6 (O délce cest)**.** Délka l nejkratší nenasycené cesty ze zdroje do stoku vypočtená Dinitzovým algoritmem na řádku 5 vzroste při každé iteraci alespoň o jedna.

Důkaz: Označme R_i jako síť rezerv v i-té iteraci Dinitzova algoritmu zbavená hran s nulovou rezervou a před pročištěním. Nechť nejkratší nenasycená cesta ze zdroje do stoku v R_i má délku l. Ukážeme, že v sítí R_{i+1} neexistuje nenasycená cesta ze zdroje do stoku délky l.

V síti R_i se pro každou nenasycenou cestu ze zdroje do stoku odstraní (nasytí) alespoň jedna její hrana (řádek 7 procedury 2.2.3). Takže v síti R_{i+1} žádná taková cesta neexistuje.
Zvýšením toku po nějaké hraně (řádek 5 procedury 2.2.3) se ovšem mohla zvýšit rezerva opačné hrany z nuly na kladné číslo, takže taková hrana je v R_{i+1} navíc oproti R_i. Stačí ukázat, že každá nenasycená cesta ze zdroje do stoku, která je teď v R_{i+1} navíc je delší než l.
Rozdělme vrcholy R_i do vrstev podle jejich vzdálenosti od zdroje. V této síti ze zvyšoval tok pouze na hranách vedoucích z vrstvy do vrstvy o jedna vyšší, takže hrany, které mohou být v R_{i+1} navíc jsou pouze ty vedoucí do vrstvy o jedna nižší. Pak každá nenasycená cesta ze zdroje do stoku vedoucí přes takovou hranu má jistě délku alespoň $l+2$, protože stok je v l-té vrstvě a žádná nenasycená hrana nevede do vrstvy o víc než jedna vyšší. □

Lemma 2.2.7. Dinitzův algoritmus provede nejvýš $|V|$ iterací.

Důkaz: Cesta ze zdroje do stoku obsahuje nejvýš $|V|$ hran, takže nejkratší taková cesta se prodlouží nejvýše $|V|-2$-krát, v první iteraci se cesta neprodlužuje (žádná totiž nebyla ještě nalezena) a v poslední iteraci už žádná cesta neexistuje. Dohromady to je tedy $|V|$ iterací. □

Věta 2.2.1 (Časová složitost Dinitzova algoritmu)**.** Dinitzův algoritmus najde maximální tok v síti v čase $\mathcal{O}(|V|^2 \cdot |E|)$.

Důkaz: Podle lemmatu 2.2.7 proběhne nejvýše $|V|$ iterací, každá z nich podle lemmatu 2.2.5 trvá $\mathcal{O}(|V| \cdot |E|)$ času, což dává dohromady $\mathcal{O}(|V|^2 \cdot |E|)$ času. Takže algoritmus se vždy zastaví a podle lemmatu 2.2.4 najde maximální tok. □

Algoritmus 2.2.1: Dinitzův algoritmus

```
   input  : síť S = (V, E, z, s, c)
   output: maximální tok f sítě S
 1 f ← nulový tok v síti S
 2 repeat
 3 |  R ← síť rezerv S vzhledem k f bez hran z nulovou rezervou
 4 |  if existuje nenasycená cesta z z do s v síti R then
 5 |  |  l ← délka nejkratší nenasycené cesty z z do s
 6 |  else
 7 |  |  return f
 8 |  R ← ČIŠTĚNÍ_SÍTĚ(R, l)                      ◁ procedura 2.2.2
 9 |  g ← BLOKUJÍCÍ_TOK(R)                         ◁ procedura 2.2.3
10 |  f ← zlepšení toku f o tok g
```

Procedura 2.2.2: Čištění sítě

```
   input  : síť R = (V, E, z, s, r), přirozené číslo l
   output: pročištěná síť R
 1 pomocí BFS rozděl vrcholy sítě do vrstev podle jejich vzdálenosti od z
 2 z R odstraň všechny hrany kromě těch vedoucích do vyšší vrstvy
 3 z R odstraň všechny vrcholy v : d(z, v) > l a k nim příslušné hrany
 4 F ← fronta vrcholů v ∈ V\{z, s} : DEG^OUT(v) = 0
 5 while F ≠ ∅ do
 6 |  odeber vrchol v z F                  ◁ vrchol na konci „slepé uličky“
 7 |  ze sítě R odeber vrchol v a všechny hrany, které do něj vedou
 8 |  pokud nějakému vrcholu v R klesl DEG^OUT na 0, zařaď ho do F
 9 return R
```

Procedura 2.2.3: Blokující tok

```
   input  : síť R = (V, E, z, s, r)
   output: blokující tok g v síti R
 1 g ← nulový tok v síti R
 2 while v síti R existuje nenasycená cesta P z z do s do
 3 |  ε ← MIN{r(e) − g(e) : e ∈ P}
 4 |  forall e ∈ P do
 5 |  |  g(e) ← g(e) + ε
 6 |  |  if g(e) = c(e) then
 7 |  |  |  smaž e z R
 8 |  dočisti síť R pomocí fronty      ◁ odstranění nových „slepých uliček“
 9 return g
```

2.3 Goldbergův algoritmus

Poslední představený algoritmus na hledání maximálního toku byl navržený Andrewem Goldbergem a publikovaný v roce 1988 v [6]. Tento algoritmus využívá odlišný přístup od předchozích. Po dobu běhu se nevylepšuje tok, ale začíná se vlnou vycházející ze zdroje, která se postupně převádí až do stoku. Algoritmus je podstatně jednodušší než Dinitzův a ukáže se, že může mít dokonce lepší časovou složitost.

Definice 2.3.1 (Vlna)**.** Nechť $S = (V, E, z, s, c)$ je síť. *Vlna* v síti S je funkce $f : E \to \mathbb{R}_0^+$, pro kterou platí:

- $\forall e \in E : f(e) \leq c(e)$, vlna nepřekročí kapacitu hrany,
- $\forall v \in V \setminus \{z, s\} : f^\Delta(v) \geq 0$, přebytek v každém vrcholu kromě zdroje a stoku je nezáporný. △

Definice 2.3.2 (Výška vrcholu)**.** Nechť $S = (V, E, z, s, c)$ je síť. *Výška* vrcholu je funkce $h : V \to \mathbb{N}_0$, pro kterou platí:

- $h(z) = |V|$, výška zdroje je vždy rovna počtu vrcholů,
- $h(s) = 0$, výška stoku je vždy rovna nule,
- $\forall (u, v) \in E : r((u, v)) > 0 \Rightarrow h(u) \leq h(v) + 1$, výška vrcholu po hraně s kladnou rezervou nikdy neklesá o více než jedna.

Pokud budeme chtít zdůraznit, že vrcholy sítě S mají výšku h, budeme síť značit $S = (V_h, E, z, s, c)$. △

Definice 2.3.3 (Převedení přebytku)**.** Nechť $S = (V_h, E, z, s, c)$ je síť, $e = (u, v) \in E$ je hrana splňující $r(e) > 0$ a $f^\Delta(u) > 0$ a $\delta = \textsc{min}\{r(e), f^\Delta(u)\}$. *Převedením přebytku* po hraně e rozumíme poslání δ jednotek toku po této hraně, což je odečtení $\varepsilon = \textsc{min}\{\delta, f((v, u))\}$ od toku v protisměru hrany e a přičtení $\delta - \varepsilon$ k toku v jejím směru. Takto zavedené převádění přebytků po hranách pak změní tok, příslušné přebytky vrcholů a rezervy hran následovně:

$$\begin{aligned}
f((v, u)) &= f((v, u)) - \varepsilon, \\
f((u, v)) &= f((u, v)) + \delta - \varepsilon, \\
f'^\Delta(u) &= f^\Delta(u) - \delta, \\
f'^\Delta(v) &= f'^\Delta(v) + \delta, \\
r'((u, v)) &= r((u, v)) - \delta, \\
r'((v, u)) &= r((v, u)) + \delta.
\end{aligned}$$

△

Algoritmus bude fungovat tak, že vlna se bude postupně šířit (převádět přebytky) ze zdroje směrem ke stoku a případné množství, které se nepodaří převést do stoku bude převedeno zpět do zdroje, aby se z vlny stal tok. Aby toto fungovalo korektně, budeme převádět přebytky pouze do vrcholů s nižší výškou. Pokud z nějakého vrcholu s kladným přebytkem nebude vést hrana z kladnou rezervou do nižšího vrcholu, budeme tento vrchol zvedat – zvyšovat výšku. To je celá myšlenka Goldberova algoritmu, jeho pseudokód je uveden v algoritmu 2.3.1.

Lemma 2.3.1. Po celou dobu běhu Goldbergova algoritmu platí:

- funkce f je vlna,
- pro všechny vrcholy výška $h(u)$ nikdy neklesá,
- pro všechny vrcholy kromě zdroje je $f^{\Delta}(u) \geq 0$.

Důkaz: Po inicializaci na řádku 1 všechny tři části triviálně platí. Při převedení přebytku na řádku 4 z definice 2.3.3 plyne, že f nepřekračuje kapacitu hrany a přebytek není v žádném vrcholu kromě zdroje záporný, přebytek ve stoku může pouze růst díky podmínce na řádku 2. Výšky se v tomto kroku nemění. Při zvedání vrcholu na řádku 6 se výšky pouze zvětšují a to jen u vrcholů různých od zdroje a stoku, funkce f se v tomto kroku nemění. □

Algoritmus 2.3.1: Goldbergův algoritmus

```
   input  : síť S = (V_h, E, z, s, c)
   output: maximální tok f sítě S
1  S, f ← INIT(S)                                        ◁ procedura 2.3.2
2  while ∃u ∈ V_h\{z, s} : f^Δ(u) > 0 do
3      if ∃(u, v) ∈ E : r((u, v)) > 0 ∧ h(u) > h(v) then
4          převeď přebytek po hraně (u, v)              ◁ definice 2.3.3
5      else
6          h(u) ← h(u) + 1
7  return f
```

Lemma 2.3.2 (O výšce)**.** Během celého běhu algoritmu je $h(v)$ platná výška pro každý vrchol (splňuje definici 2.3.2).

Důkaz: Výška zdroje a stoku se díky podmínce na řádku 2 nikdy nemění. Indukcí dle běhu algortmu ukážeme, že se v žádném jeho kroku neporuší ani podmínka rozdílů výšek. Při inicializaci na řádku 1 se výšky a počáteční vlna inicializují a všechny hrany s kladnou rezervou vedou z vrcholů s výškou 0,

Procedura 2.3.2: Inicializace Goldbergova algoritmuú

input : síť $S = (V_h, E, z, s, c)$
output: síť S a vlna f v síti S inicializované pro Goldbergův alg.

1 $h(z) \leftarrow |V_h|$ ◁ výška zdroje
2 **forall** $v \in V_h \setminus \{z\}$ **do**
3 $\quad h(v) \leftarrow 0$ ◁ počáteční výška ostatních vrcholů
4 $f \leftarrow$ nulový tok v síti S
5 **forall** $(z, u) \in E$ **do**
6 $\quad f((z, u)) \leftarrow c((z, u))$ ◁ počáteční vlna
7 **return** S, f

do vrcholů s výškou 0 nebo do zdroje s výškou $|V|$. Pak se může podmínka rozdílů výšek porušit pouze ve dvou případech:

- Zvednutím vrcholu $u \in V$, ze kterého vede hrana (u, v) s kladnou rezervou, kde $h(u) = h(v) + 1$. To ale nemůže nastat, protože algoritmus na řádku 3 dá v takovém případě přednost převedení přebytku po této hraně před zvednutím vrcholu u.
- Zvětšením rezervy hrany $(u, v) \in E$, kde $h(u) > h(v) + 1$. To ovšem nemůže nastat, protože rezerva této hrany se může zvýšit pouze převedením přebytku po opačné hraně (v, u) a to nenastane, protože algoritmus převádí přebytky pouze z vrcholů vyšších do nižších. □

Lemma 2.3.3 (O korektnosti)**.** Pokud se Goldbergův algoritmus zastaví, f je maximální tok.

Důkaz: Nejprve ukážeme, že f je tok. Jelikož po celou dobu běhu algoritmu je f vlna, stačí ukázat, že pro ni na konci platí Kirchhoffův zákon. Protože se algoritmus zastavil neexistuje kvůli podmínce na řádku 2 vrchol (kromě zdroje a stoku) s kladným přebytkem, což je přeformulovaný Kirchhoffův zákon z definice 1.1.4.
Teď sporem ukážeme, že f je maximální. Pro spor nechť f není maximální. Z lemmatu 1.2.6 tedy plyne, že existuje nenasycená cesta P ze zdroje do stoku. Tato cesta má nejvýše $|V|$ vrcholů a tedy $|V| - 1$ nenasycených hran a podle lemmatu 2.3.1 začína ve zdroji ve výšce $|V|$ a končí ve stoku ve výšce 0. Protože výšky vrcholů cesty P klesají o $|V|$ výšek, tak alespoň pro jednu její hranu (u, v) musí platit, že $h(u) > h(v)+1$, což je spor s lemmatem o výšce 2.3.2. □

Lemma 2.3.4. Po celou dobu běhu Goldbergova algoritmu platí

$$\sum_{u \in V} f^{\Delta}(u) = 0.$$

Důkaz: Po inicializaci lemma platí, protože po každé hraně (z, u) vedoucí ze zdroje se pošle vlna rovna kapacitě $c((z, u))$, přebytek se pak ve zdroji sníží o $c((z, u))$ a ve vrcholu u zvýší o $c((z, u))$. Všechny ostatní vrcholy mají přebytek roven nule. Přebytky ve vrcholech se pak mění pouze při jejich převádění po hraně (u, v) na řádku 4 a to podle definice 2.3.3 jednou odečte hodnotu δ od $f^\Delta(u)$ a jednou přičte hodnotu δ k $f^\Delta(v)$, takže suma přebytků ve všech vrcholech se nezmění a zůstane nulová. □

Lemma 2.3.5. Pokud během Goldbergova algoritmu v nějaký okamžik platí $-f^\Delta(z) = f^\Delta(s)$, pak pro všechny vrcholy sítě $v \in V \backslash \{z, s\}$ platí $f^\Delta(v) = 0$ a f je tok.

Důkaz: Sporem. Nechť existuje vrchol $v \in V \backslash \{z, s\}$ pro který $f^\Delta(v) > 0$. Podle lemmatu 2.3.4 platí $\sum_{u \in V} f^\Delta(u) = 0$ a kvůli předpokladům tohoto lemmatu dokonce $\sum_{u \in V \backslash \{z,s\}} f^\Delta(u) = 0$. Podle lemmatu 2.3.1 jsou přebytky ve všech vrcholech kromě zdroje nezáporné, takže

$$\sum_{u \in V \backslash \{z,s\}} f^\Delta(u) = f^\Delta(v) + \sum_{u \in V \backslash \{z,s,v\}} f^\Delta(u) > 0, \text{ což je spor.}$$

Protože vlna f má nulové přebytky ve všech vrcholech kromě zdroje a stoku, tak splňuje definici 2.2.3 a je tokem. □

Lemma 2.3.6. Nechť v je vrchol sítě v nějaké fázi Goldbergova algoritmu, pro který $f^\Delta(v) > 0$. Pak z tohoto vrcholu existuje nenasycená cesta do zdroje.

Důkaz: Nechť v je vrchol splňující předpoklady a označme množinu vrcholů

$$A := \{u \in V : \text{existuje nenasycená cesta z } v \text{ do } u\}.$$

Ukážeme, že A obsahuje zdroj. Sečteme přebytky všech vrcholů v množině A. Do tohoto součtu budou přispívat pouze hrany vedoucí ven z množiny A a k nim opačné, hrany vedoucí mezi vrcholy v A do součtu přispějí nulou, což vychází z definice 2.3.3. Takže dostáváme:

$$\sum_{u \in A} f^\Delta(u) = \sum_{e \in S(V \backslash A, A)} f(e) - \sum_{e \in S(A, V \backslash A)} f(e).$$

Pro všechny hrany (u, v) vedoucí z venku do množiny A v první sumě platí, že $f((u, v)) = 0$, protože kdyby tomu tak nebylo, hrana k ní opačná (v, u) by byla nenasycená a vrchol u ležící mimo množinu A by se do A mohl přidat. První suma je tedy rovna nule. Druhá suma je jistě větší nebo rovna nula, protože se jedná o součet nezáporných čísel. Takže součet přebytků všech vrcholů v A je menší nebo roven nule. Vrchol v má však z předpokadu kladný přebytek, takže v A musí existovat i vrchol ze zápornáým přebytkem a to může být pouze zdroj, takže zdroj patří do množiny A. □

Lemma 2.3.7. Pokud během Goldbergova algoritmu pro nějaký vrchol v platí $h(v) \geq |V|$, tak z něj neexistuje nenasycená cesta do stoku.

Důkaz: Sporem. Nechť P je nenasycená cesta z vrcholu v do stoku, kde $h(v) \geq |V|$. Cesta P obsahuje nejvýše $|V|$ vrcholů a tedy $|V| - 1$ hran. Každá hrana $(u, v) \in P$ je nenasycená, takže $h(u) \leq h(v) + 1$, neboli po každé hraně cesty klesne výška nejvíce o jedna. Výška po celé cestě P klesne o alespoň $|V|$, takže musí obsahovat alespoň $|V|$ hran, což je spor. □

Lemma 2.3.8 (O maximální výšce)**.** Po celou dobu běhu Goldbergova algoritmu pro všechny vrcholy $v \in V_h$ platí $h(v) \leq 2 \cdot |V|$.

Důkaz: Sporem. Nechť vrchol v má výšku $h(v) > 2 \cdot |V|$, do této výšky se mohl dostat pouze zvednutím z výšky alespoň $2 \cdot |V|$. Před tímto zvednutím měl vrchol v jistě kladný přebytek (aby ho algoritmus mohl zvednout) a dle lemmatu 2.3.6 z něj v tu chvíli existovala nenasycená cesta do zdroje. Ta klesala alespoň o $|V|$ výšek (z výšky alespoň $2 \cdot |V|$ do výšky zdroje $|V|$) a obsahovala maximálně $|V| - 1$ hran, takže alespoň pro jednu její hranu (u, v) musí platit, že $h(u) > h(v) + 1$, což je spor s lemmatem o výšce 2.3.2. □

Lemma 2.3.9 (O počtu zvednutí)**.** Goldbergův algoritmus provede dohromady nejvýš $2 \cdot |V|^2$ zvednutí vrcholu.

Důkaz: Z předchozího lemmatu 2.3.8 plyne, že každý vrchol může být zvednut nejvýše $2 \cdot |V|$-krát, takže všechny vrcholy dohromady $2 \cdot |V|^2$-krát. □

Definice 2.3.4 ((Ne)nasycené převedení přebytku)**.** Řekneme, že převedení přebytku po hraně e je *nasycené*, pokud po jeho provedení klesla rezerva hrany e na nulu. V opačném případě se jedná o *nenasycené* převedení. △

Lemma 2.3.10 (O počtu nasycených převedení)**.** Goldbergův algoritmus provede nejvýše $|V| \cdot |E|$ nasycených převedení přebytku.

Důkaz: Nechť (u, v) je hrana sítě, ukážeme kolikrát nejvíce mohlo dojít k nasycenému převedení přebytku po této hraně. Při prvním nasyceném převedení přebytku po hraně (u, v) klesla rezerva této hrany na nulu a dle lemmatu 2.3.2 platilo $h(u) = h(v) + 1$. Aby došlo k dalšímu nasycenému převedení po této hraně, musí se zvýšit její rezerva. K tomu může dojít pouze převedením přebytku po hraně k ní opačné (v, u). Aby k tomuto převedení mohlo dojít musí mít vrchol v výšku o jedna větší než u. Aby se poté mohl znovu převádět přebytek po hraně (u, v) musí se vrchol u zvednout nejdříve do výšky vrcholu v a pak o jedna výš, celkově tedy alespoň dvakrát.
Takže mezi každými dvěmi nasycenými převedeními po hraně (u, v) musel být vrchol u alespoň dvakrát zvednut. A podle lemmatu 2.3.9 k tomu mohlo dojít nejvíce $|V|$-krát. Takže všech nasycených převedení se mohlo pro všechny hrany dohromady provést nejvýše $|V| \cdot |E|$. □

Lemma 2.3.11 (O počtu nenasycených převedení)**.** Goldbergův algoritmus provede $\mathcal{O}(|V|^2 \cdot |E|)$ nenasycených převedení přebytku.

Důkaz: Nechť $S = (V_h, E, z, s, c)$ je síť. Zaveďme následující funkci (potenciál):

$$\Phi := \sum_{v \in V \setminus \{z,s\}, f^{\Delta}(v) > 0} h(v),$$

ta bude sloužit k popisu stavu algoritmu. Každá operace potenciál buď zvýší nebo sníží, ukáže se, že tato funkce je vždy nezáporná a operací, které ji snižují je výrazně méně než těch, které ji zvyšují. Na konci běhu algoritmu pomocí této funkce odvodíme asymptotický počet nenasycených převedení přebytků. Ponteciál se v průběhu algoritmu mění takto:

- Na začátku běhu algoritmu má hodnotu 0.
- Po celou dobu jistě platí $\Phi \geq 0$, protože se jedná o součet nezáporných čísel.
- Zvednutí vrcholu zvýší její hodnotu o jedna. Aby mohl být vrchol zvednut, musí mít kladný přebytek, takže do sumy už přispíval svou výškou a po zvednutí přispěje ještě o jedna víc. Z lemmatu 2.3.9 víme, že nastane nejvýše $2 \cdot |V|^2$ zvednutí vrcholu, takže zvedáním všech vrcholů se zvýší potenciál nejvýše o $2 \cdot |V|^2$.
- Nasycené převedení přebytku po hraně (u, v) zvýší Φ nejvýše o $2 \cdot |V|$. Vrchol u má před převedením jistě kladný přebytek a po převedení může mít stále kladný nebo nulový. Pokud má vrchol v před převedením nulový přebytek, tak po převedení má jistě kladný a do potenciálu bude přispívat svou výškou, ta je nejvýše $2 \cdot |V|$. Potenciál se tedy zvýší nejvýše o $2 \cdot |V|$ a podle lemmatu 2.3.10 k tomu dojde nejvýše $|V| \cdot |E|$-krát, takže nasycené převedení přebytku zvýší potenciál za celý běh algoritmu nejvýše o $2 \cdot |V|^2 \cdot |E|$.
- Při nenasyceném převedení přebytku po hraně (u, v) se od potenciálu jistě odečte $h(u)$. Pokud má vrchol v před převedením nulový přebytek, tak po převedení má jistě kladný a do potenciálu bude přispívat svou výškou, pro tu dle lemmatu 2.3.2 platí $h(v) = h(u) - 1$. Takže každé nenasycené převedení sníží potenciál alespoň o $h(u) - h(v) = h(u) - h(u) + 1 = 1$.

Potenciál tedy celkově stoupne o nejvýše $2 \cdot |V|^2 + 2 \cdot |V|^2 \cdot |E| = \mathcal{O}(|V|^2 \cdot |E|)$ a při každém nenasyceném převedení klesne o alespoň jedna, takže všech nenasycených převedení je $\mathcal{O}(|V|^2 \cdot |E|)$. □

Věta 2.3.1 (Složitost Goldbergova algoritmu)**.** Goldbergův algoritmus najde maximální tok v čase $\mathcal{O}(|V|^2 \cdot |E|)$.

Důkaz: Inicializace na řádku 1 trvá triviálně $\mathcal{O}(|E|)$ času. Lemma 2.3.9 tvrdí, že se provede nejvýše $2 \cdot |V|^2$ zvednutí vrcholu, dle lemmatu 2.3.10 nejvýše $|V| \cdot |E|$ nasycených převedení přebytku a dle lemmatu 2.3.11 $\mathcal{O}(|V|^2 \cdot |E|)$ nenasycených převedení přebytku. Předpokládáme, že všechny operace mají konstantní složitost. Celková složitost algoritmu tedy je $\mathcal{O}(|E| + |V| \cdot |E| + |V|^2 \cdot |E| + |V|^2) = \mathcal{O}(|V|^2 \cdot |E|)$ a dle lemmatu 2.3.3 je nalezený tok maximální. □

2.3.1 Vylepšení výběrem nejvyššího vrcholu

Základní verze Goldbergova algoritmu dosahuje stejné asymptotické složitosti jako Dinitzův algoritmus. Jednoduchou úpravou lze však dosáhnout složitosti řádově lepší. Goldbergův algoritmus nespecifikuje, jaký vrchol s kladným přebytkem se má na řádku 2 vybrat, pokud jich existuje víc. Stačí namísto libovolného vrcholu s kladným přebytkem vždy vybírat ten s největší výškou.

Lemma 2.3.12. Goldbergův algoritmus, který na řádku 2 vybírá nejvyšší vrchol s kladným přebytkem, provede $\mathcal{O}(|V|^3)$ nenasycených převedení přebytku.

Důkaz: Vrcholy sítě rozdělme do hladin podle jejich výšek a označme nejvyšší hladinu s kladným přebytkem jako:

$$H := \textsc{max}\{h(v) : v \in V \land v \neq z, s \land f^{\Delta}(v) > 0\}.$$

Běh algoritmu rozdělme na fáze. Každá fáze začíná a končí změnou H, to se může zvýšit o jedna, pokud se vrcholu v nejvyšší hladině zvýší výška, a snížit o jedna, pokud se z nejvyššího vrcholu převede přebytek do vrcholu s výškou o jedna menší. H je vždy nezáporné. Z lemmatu 2.3.9 víme, že nastane $\mathcal{O}(|V|^2)$ zvednutí vrcholu, takže počet zvýšení H je shora omezen a protože se H zvyšuje a snižuje vždy o jedna, je počet snížení omezen počtem zvýšení. Tudíž nastane $\mathcal{O}(|V|^2)$ fází.
V každé fázi algoritmus na řádku 2 vybírá pouze vrcholy z nejvyšší hladiny, a pokud se z nějakého takového vrcholu v provede nenasycené převedení přebytku, vrchol má poté jistě nulový přebytek a v této fázi z něj určitě k žádnému dalšímu převedení přebytku nedojde. Kdyby z takového vrcholu v k dalšímu převedení přebytku totiž došlo, musel by se do vrcholu nejdříve nějaký přebytek dostat a to je možné pouze z vrcholu u s o jedna větší výškou. K tomu ale může dojít v jiné fázi, protože vrchol u má kladný přebytek a větší výšku než vrchol v. Z toho plyne, že v každé fázi dojde k nejvýše jednomu nenasycenému převedení přebytku z každého vrcholu, takže v každé fázi dojde k nejvýše $|V|$ nenasyceným převedení přebytku.
Jelikož fází je $\mathcal{O}(|V|^2)$ a v každé dojde k nejvýše $|V|$ nenasyceným převedení

přebytku, dohromady dojde k $\mathcal{O}(|V|^3)$ převedení za dobu běhu celého algoritmu. □

Tento odhad je i tak příliš pesimistický a dá se ukázat, že počet nenasycených převedení přebytku lze tímto vylepšením dokonce omezit $\mathcal{O}(|V|^2 \cdot \sqrt{|E|})$, jak je uvedeno v[1] na straně 343.

Lemma 2.3.13. Goldbergův algoritmus využívající výběr nejvyššího vrcholu najde maximální tok v čase $\mathcal{O}(|V|^2 \cdot \sqrt{|E|})$.

Důkaz: V důkazu věty 2.3.1 o složitosti Goldbergova algoritmu měl největší podíl na výsledné složitosti člen $\mathcal{O}(|V|^2 \cdot |E|)$ udávající počet nenasycených převedení přebytku, ten se uvedenou úpravou změní na $\mathcal{O}(|V|^2 \cdot \sqrt{|E|})$, takže složitost Goldbergova algoritmu s výběrem nejvyššího vrcholu je $\mathcal{O}(|E| + |V| \cdot |E| + |V|^2 \cdot \sqrt{|E|} + |V|^2) = \mathcal{O}(|V|^2 \cdot \sqrt{|E|})$. □

2.3.2 Vylepšení škálováním přebytků

Myšlenka tohoto vylepšení Goldbergova algoritmu spočívá v upřednostňování převádění přebytku z vrcholů s velkým přebytkem do vrcholů s malým přebytkem, čímž se sníží jejich celkový počet. Tím se dosahuje ještě lepší časové složitosti než v předchozím uvedeném vylepšení výběrem nejvyššího vrcholu, algoritmus byl publikován v roce 1989 v [7] a předpokládá celočíselné kapacity hran, jeho pseudokód je uveden v algoritmu 2.3.3.

Algoritmus na začátku zvolí proměnnou U jako maximum z kapacit hran vedoucích ze zdroje. Pomocí U se zvolí proměnná $K := \lceil \log_2 U \rceil$ udávající počet škálovacích iterací, a proměnná $\alpha := 2^K$ udávající horní mez přebytků ve všech vrcholech. V každé iteraci se budou přebytky převádět pouze z vrcholů s dostatečně velkým přebytkem konkrétně větším než $\frac{\alpha}{2}$ a menším nebo rovným α, až takové vrcholy nebudou existovat, α se zmenší na polovinu a pokračuje se další iterací. Při převádění přebytku se dává přednost převádění z vrcholu s nejnižší výškou, tím zajistíme, že převáděné množství bude co největší, protože vrchol, do kterého se bude převádět má jistě přebytek menší nebo roven $\frac{\alpha}{2}$. Aby se přebytek nějakého vrcholu nezvýšil nad hodnotu α, upravíme množství, které lze po hraně (u, v) převést (definice 2.3.3):

$$\delta = \begin{cases} \textsc{min}\{r((u,v)), f^{\Delta}(u)\} & \text{pokud } v = z, s, \\ \textsc{min}\{r((u,v)), f^{\Delta}(u), \alpha - f^{\Delta}(v)\} & \text{jinak.} \end{cases}$$

Lemma 2.3.14. Každé nenasycené převedení přebytku po hraně (u, v) zvýší její průtok o více než $\frac{\alpha}{2}$.

Důkaz: Pokud $v = z, s$, lemma triviálně platí, protože $f^\Delta(u) > \frac{\alpha}{2}$. Pokud $v \neq z, s$, pak jistě $f^\Delta(v) < \frac{\alpha}{2}$, takže

$$\delta = \text{MIN}\{f^\Delta(u), \alpha - f^\Delta(v)\} > \text{MIN}\{\frac{\alpha}{2}, \alpha - \frac{\alpha}{2}\} = \frac{\alpha}{2}.$$

□

Lemma 2.3.15. Po celou dobu běhu algoritmu má každý vrchol kromě stoku přebytek nejvýše α.

Důkaz: Po inicializaci na řádcích 1 až 3 lemma triviálně platí. Přebytek ve vrcholu v se může zvýšit pouze převedením na řádku 10. Po převedení po hraně (u, v) pro přebytek vrcholu v platí

$$f'^\Delta(v) = f^\Delta(v) + \text{MIN}\{(r((u,v)), f^\Delta(u), \alpha - f^\Delta(v)\} \leq f^\Delta(v) + \alpha - f^\Delta(v) = \alpha.$$

□

Lemma 2.3.16. Goldbergův algoritmus využívající škálování přebytků najde maximální tok v čase $\mathcal{O}(|V| \cdot |E| + |V|^2 \cdot \log_2 U)$.

Důkaz: Je uveden v [7] na straně 755. □

2.3.3 Heuristika globálního přepočítání výšek

Touto heuristikou publikovnou v [8] lze docílit menšího počtu zvyšování výšek vrcholů.

Za běhu Goldbergova algoritmu může pro nějakou hranu (u, v) s kladnou rezervou nastat, že $h(v)$ je výrazně větší než $h(u)$, takže aby se po této hraně mohl převést přebytek, musí se nejdříve vrchol u zvednout do výšky $h(v) + 1$, zvedání vrcholu u však pouze prodlužuje dobu běhu a vlnu v síti nijak nemění. Z lemmatu 2.3.17 plyne, že $d(u, s) \geq h(u) - h(s) = h(u)$, takže každému vrcholu kromě zdroje můžeme přiřadit výšku rovnou jeho vzdálenosti od stoku a stále se bude jednat o platnou výšku. Takové nastavení výšek způsobí, že vrcholy jsou ihned připravené převádět svůj přebytek směrem ke stoku bez zvyšování jejich výšek. Vrcholům u, ze kterých neexistuje nenasycená cesta do stoku, lze přiřadit výšku $|V| + d(u, z)$, z nich se pak budou přebytky vracet zpět směrem do zdroje. Výšky vrcholům lze přiřadit pomocí zpětného BFS začínajícího ve stoku respektive ve zdroji, který se vydá z vrcholu u po hraně (u, v), pokud má k ní opačná hrana (v, u) kladnou rezervu.

Takové přepočítání výšek lze provádět vždy po provedení nějakého daného počtu zvednutí vrcholů nebo provedení cyklu mezi řádky 2 a 6.

Algoritmus 2.3.3: Goldbergův algoritmus – škálování přebytků

```
input  : síť S = (V_h, E, z, s, c)
output: maximální tok f sítě S
1  S, f ← INIT(S)                                         ◁ procedura 2.3.2
2  U ← MAX{c((z, u)) : (z, u) ∈ E}
3  K ← ⌈log_2 U⌉
4  lists ← množina 2 · |V| prázdných seznamů vrcholů
5  for i in K ... 0 do
6  |   α ← 2^i
7  |   forall v ∈ V_h\{s} : f^Δ(v) > α/2 do
8  |   |   přidej v do lists[h(v)]          ◁ seznam vrcholů se stejnou výškou
9  |   while ∃ neprázdný seznam v množině lists do
10 |   |   level ← nejmenší n ∈ ℕ : lists[n] ≠ ∅
11 |   |   odeber vrchol u ze začátku seznamu lists[level]
12 |   |   if ∃ hrana (u, v) ∈ E : r((u, v)) > 0 ∧ h(u) > h(v) then
13 |   |   |   převeď přebytek po hraně (u, v)     ◁ upravená definice 2.3.3
14 |   |   |   if f^Δ(u) > α/2 then                 ◁ stále přípustný vrchol
15 |   |   |   |   přidej u do seznamu lists[level]
16 |   |   |   if f^Δ(v) > α/2 ∧ v ≠ z, s then      ◁ nový přípustný vrchol
17 |   |   |   |   přidej v do seznamu lists[level − 1]
18 |   |   else
19 |   |   |   h(u) ← MIN{h(v) : (u, v) ∈ E ∧ r((u, v)) > 0} + 1
20 |   |   |   přidej u do seznamu lists[h(u)]
21 return f
```

Lemma 2.3.17. Nechť h je výška vrcholů v síti (V_h, E, z, s, c) v nějaké fázi Goldbergova algoritmu, pak pro každou nenasycenou cestu z vrcholu u do vrcholu v platí $d(u, v) \geq h(u) - h(v)$.

Důkaz: Pokud nenasycená cesta z vrcholu u do vrcholu v neexistuje, pak její délka je ∞ a lemma platí. Pokud existuje, označme ji P, každá hrana $(x, y) \in P$ je nenasycená, takže $h(x) \leq h(y) + 1$, tedy po každé hraně klesne výška nejvýše o jedna. Počet hran v cestě P je roven $d(u, v)$, takže výška po celé cestě klesne nejvýše o $d(u, v)$. □

2.3.4 Minimální řez a velikost maximálního toku

Goldbergův algoritmus najde vždy maximální tok sítě, ovšem může se stát, že nám stačí znát pouze minimální řez nebo velikost maximálního toku a jeho konkrétní rozložení po hranách sítě nás nezajímá.

Pokud v nějakém okamžiku běhu Goldbergova algoritmu pro všechny vrcholy v s kladným přebytkem platí $h(v) \geq |V|$, tak z lemmatu 2.3.7 plyne, že z těchto vrcholů neexistuje nenasycená cesta do stoku, takže z nich nelze převést přebytek až do stoku. Přebytek se z těchto vrcholů bude podle lemmatu 2.3.6 pouze vracet zpět do zdroje, čímž se upraví vlna na tok, ale velikost výsledného toku se nijak nezmění.
Takže pokud v nějakém okamžiku mají všechny vrcholy s kladným přebytkem (kromě stoku) výšku alespoň $|V|$, může se algoritmus zastavit a hodnota $f^{\Delta}(s)$ udává velikost maximálního toku v síti. Minimální řez lze v ten moment najít pomocí zpětného BFS (viz podsekce 2.3.3) začínajícího ve stoku, který do minimálního řezu přidá všechny hrany (u, v), pokud BFS nalezl vrchol u a hrana (u, v) má nulovou rezervu – z vrcholu v se neexistuje nenasycená cesta do stoku. Pokud budeme chtít vědět, jak maximální tok vypadá, necháme algoritmus běžet dál, dokud se sám nezastaví po nesplnění podmínky na řádku 2. Běh Goldbergova algoritmu lze tedy rozdělit na dvě fáze, v té první se vždy nejdříve najde velikost maximálního toku, případně i minimální řez, a ve druhé se najde onen maximální tok. Platí, že první fáze zabírá asymptoticky více času [18].

algoritmus (sekce)	**asymptotická časová složitost**
Fordův-Fulkersonův (2.1)	$\mathcal{O}(\textsc{max}\{\lvert f\rvert : f \text{ je tok sítě}\} \cdot \lvert E\rvert)$
Edmonsův-Karpův (2.1.1)	$\mathcal{O}(\lvert V\rvert \cdot \lvert E\rvert^2)$
Dinitzův (2.2)	$\mathcal{O}(\lvert V\rvert^2 \cdot \lvert E\rvert)$
Goldbergův (2.3)	$\mathcal{O}(\lvert V\rvert^2 \cdot \lvert E\rvert)$
– výběr nejvyššího vrcholu (2.3.1)	$\mathcal{O}(\lvert V\rvert^2 \cdot \sqrt{\lvert E\rvert})$
– škálování přebytků (2.3.2)	$\mathcal{O}(\lvert V\rvert \cdot \lvert E\rvert + \lvert V\rvert^2 \cdot \log_2 U)$,
	$U = \textsc{max}\{c((z, u)) : (z, u) \in E\}$

Tabulka 2.1: Asymptotická časová složitost uvedených sekvenčních algoritmů hledajících maximální tok

Kapitola 3

Paralelní algoritmy na architektuře CUDA

Na úvod této kapitoly jsou uvedeny základní měřítka efektivity algoritmů, následuje úvod do paralelních algoritmů pro hledání maximálního toku spolu s uvedením existujících paralelních variant Goldbergova algoritmu pro CPU, na který navazuje krátký úvod do architekrury CUDA. Dále jsou představeny existující paralelní algoritmy pro tuto architekturu spolu s paralelní verzí algoritmu BFS. Na závěr je navrhnut nový paralelní algoritmus využívající obarvení hran. Implementacemi uvedených algoritmů pro CUDA se zabývá následující kapitola.

3.1 Základní pojmy složitosti paralelních algoritmů

V této sekci jsou uvedeny základní pojmy a měřítka složiosti sekvenčních a paralelních algoritmů pocházející z [19].

- T_A^K : doba výpočtu sekvenčního algoritmu A, který řeší problém K na vstupních datech velikosti n.
- $SL^K(n)$: nejhorší časová složitost nejlepšího možného sekvenčního algoritmu pro řešení problému K.
- $SU^K(n)$: nejhorší časová složitost nejrychlejšího známého sekvenčního algoritmu pro řešení problému K.
- $T(n,p)$: čas, který uplynul od začátku paralelního výpočtu do okamžiku, kdy poslední (nejpomalejší) z p procesorů/vláken/jader skončil výpočet. Skláda se z výpočetních kroků a komunikačních kroků.
- $S(n,p) = \frac{SU(n)}{T(n,p)}$: *paralelní zrychlení*, které je vždy menší nebo rovno p. V případě, že by stoupl počet procesorů k-krát a zároveň klesl $T(n,p)$

k-krát, tedy $S(n,p) = \Theta(p)$, jedná se o *lineární zrychlení*, kterého je ve skutečnosti obtížné dosáhnout.

- $L^K(n,p) = \frac{SL^K(n)}{p}$: spodní mez na paralelní čas pro p procesorů.
- $C(n,p) = p \cdot T(n,p)$: *paralelní cena.*
- Algoritmus nazveme *cenově optimální*, pokud $C(n,p) = \mathcal{O}(SU(n))$.
- $E(n,p) = \frac{SU(n)}{C(n,p)}$: *paralelní efektivnost*, která je menší nebo rovna 1.
- Pokud pro danou konstantu $0 < E_0 < 1$ platí, že $E(n,p) \geq E_0$, říkáme, že algoritmus má *konstantní efektivnost.*
- Pro paralelní algoritmus jsou tvrzení, že 1. je *cenově optimální*, 2. má *lineární zrychlení* a 3. má *konstantní efektivnost*, ekvivalentní.

3.2 Výběr vhodného algoritmu pro paralelizaci

Goldbergův algoritmus 2.3.1 se nabízí jako nejvhodnější kandidát na paralelizaci pro více vláken díky jeho vlastnosti, že v hlavní smyčce na řádcích 2 až 6 jsou všechny operace − převádění přebytků a zvedání vrcholů − lokálního charakteru a jejich provádění není závislé na stavu v jiných částech sítě, proto ji lze paralelizovat bez vysoké nadbytečné režie.

Při návrhu paralelního algoritmu pro velký počet vláken, se nabízí každému vrcholu nebo hraně přiřadit jedno vlákno a nechat vlákna paralelně převádět přebytky a zvedat vrcholy. To ovšem bez nějaké formy synchronizace může vést k časově závislým chybám, v jeden moment může totiž jedno vlákno převádět přebytek do vrcholu v, do kterého jiné vlákno také převádí přebytek, a hodnota $f^\Delta(v)$ je tedy upravována více vlákny současně. Tudíž je nutné navrhnout i nějakou synchronizační metodu, aby algoritmus fungoval správně. Mimo jiné právě různé přístupy k synchronizaci vláken od sebe odlišují jednotlivé dále uvedené algortimy a jsou jedním z faktorů ovlivňujících jejich efektivitu. Je důležité poznamenat, že synchronizace by neměla používat kritické sekce a zámky, protože to není v CUDA podporováno [12] a muselo by se to implementovat například atomickými operacemi s pamětí, což by při velkém počtu vláken ssebou neslo vysokou režii.

3.3 Metoda pulzu

Paralelní verzi Goldbergova algoritmu pro libovolný počet procesorů uvedli sami jeho autoři v původním článku [6]. Předpokládá se distribouvaný model bez sdílené paměti s možností posílání zpráv mezi procesory.
Nejdříve se každému vrcholu přiřadí procesor, který ho bude zpracovávat. Procesor může komunikovat se všemi procesory sousedních vrcholů a má lokálně k

dispozici data pouze o svém vrcholu a jeho incidentních hranách. Algoritmus pak funguje na základě opakovaného provádění takzvaného *pulzu*, ten se skládá ze čtyř fází: v první se posílají přebytky z vrcholů do jejich sousedů, ve druhé se vrcholy zvedají, ve třetí vrcholy rozesílají svou novou výšku svým sousedům a ve čtvrté vrcholy načítají přebytky poslané od jejich sousedů v první fázi. Provádění *pulzu* se opakuje, dokud existuje vrchol různý od stoku s kladným přebytkem, což je stejná podmínka jako v sekvenčním algoritmu 2.3.1. Všechny tyto fáze jsou prováděny každým procesorem paralelně. Aby algoritmus fungoval korektně, před začátkem provádění každé fáze procesory čekají, až všechny ostatní procesory dokončí tu současnou – synchronizace bariérou. Pseudokód *pulzu* je uveden v algoritmu 3.3.1.

Tento algoritmus může sloužit jako startovní bod při návrhu paralelního algoritmu určeného jak pro CPU tak pro architekturu CUDA a jeho synchronizační metodu fázemi využívají některé dále uvedené algoritmy.

Algoritmus 3.3.1: Pulz

```
input  : síť S = (V_h, E, z, s, c); vlna f v síti S
output: síť S a vlna f v síti S po provedení jednoho pulzu

1  forall u ∈ V_h paralelně do              ◁ jeden procesor ↔ jeden vrchol
       1. FÁZE
2      forall (u, v) ∈ E : r((u, v)) > 0 do
3          převeď přebytek po hraně (u, v) podle definice 2.3.3, ale
           nezvyšuj hodnotu f^Δ(v)
       2. FÁZE
4      if f^Δ(u) > 0 ∧ u ≠ z, s then
5          h'(u) ← MIN{h(v) : (u, v) ∈ E ∧ r((u, v)) > 0} + 1
6      else
7          h'(u) ← h(u)
       3. FÁZE
8      if f^Δ(u) > 0 ∧ h(u) ≠ h'(u) then
9          h(u) ← h'(u)
10         broadcast h(u) všem v ∈ V_h : (u, v) ∈ E
       4. FÁZE
11     zvyš hodnotu f^Δ(u), podle převedeného množství do vrcholu u z
       1. fáze na řádku 3
12 return S, f
```

3.4 Paralelní algoritmy pro CPU

Porovnáváním paralelních algoritmů pro hledání maximálního toku v síti na CPU se zabývá bakalářská práce Jana Groschafta [17] z roku 2019. V ní jsou uvedeny tři paralelní algoritmy.

Prvním z nich s názvem „Parallel Push–Relabel“ je implementace algoritmu využívající pulzu (algoritmus 3.3.1) ze začátku této kapitoly. Vrcholy jsou v každé fázi udržovány ve frontě, ze které je jednotlivá vlákna vybírají a provádějí nad nimi operace daného pulzu, dokud není fronta prázdná.

Zbylé dva algortimy paralelizují Goldbergův algoritmus rozdělením sítě na segmenty, v práci jsou pojmenovány „Push–Relabel Segment“ a „Ahuja–Orlin Segment“. Oba fungují tak, že vrcholy sítě nejdříve rozdělí podle jejich výšek do disjunktních množin (segmentů) a každé vlákno CPU pak sekvenčně převádí přebytky pouze uvnitř svého segmentu. Druhý jmenovaný algoritmus se od prvního liší tím, že při převádění přebytlů uvnitř segmentu využívá vylepšení škálováním přebytků uvededené v podsekci 2.3.2.

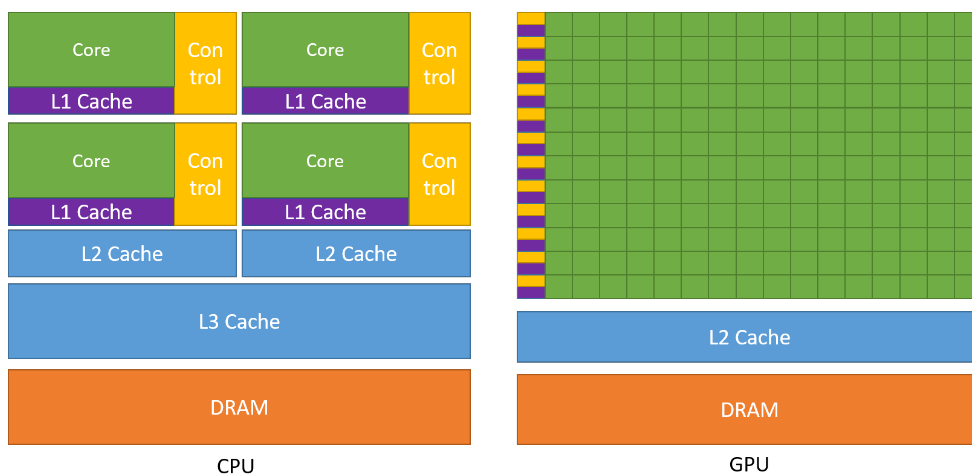
V práci je popsán také paralelní algoritmus pro globální přepočítání výšek na CPU, který všechny tři uvedené algoritmy využívají a periodicky provádí.

3.5 Představení technologie CUDA

Informace a obrázky celé této sekce pochází z příručky [9], kde je možné najít podrobnější popis této technologie. Hlavním rozdílem mezi CPU a GPU je, že CPU je navržený a určený k co nejrychlejšímu vykonávání posloupnosti operací v jednom vlákně a může souběžně provádět nejvýše desítky takových vláken, předností GPU je naopak vykonávání až tisíců vláken souběžně, které nejsou tak výkonné, ale díky jejich velkému počtu dokáží skrýt latenci přístupu do paměti. Obrázek 3.1 ukazuje schématický rozdíl mezi GPU a CPU, kde GPU disponuje výrazně vyšším počtem tranzistorů pro práci z daty.

CUDA® byla uvedena společností NVIDIA® v listopadu 2006 jako platforma pro obecné paralelní výpočty a programovací model pro svá GPU. Ve své podstatě CUDA poskytuje tři klíčové abstrakce: hierarchie skupin vláken, sdílené paměti a synchronizace bariérou. To je při psaní aplikací realizováno pouze minimálním rozšířením jazyka C. Tyto abstrakce vedou programátora k rozdělení řešeného problému na menší subproblémy řešitelné nezávisle a paralelně bloky vláken a tyto subproblémy na ještě menší řešitelné jednotlivými vlákny bloku.

CUDA rozšiřuje jazyk C o možnost definovat funkci zvanou *kernel*, která se spouští na M vláknech v každém z N stejně velkých bloků souběžně na GPU, narozdíl od prostých funkcí vykonávaných sekvenčně jedním vláknem na CPU. Při spuštění kernelu se každému vláknu přiřadí unikátní ID, které je z kernelu přístupné z vestavěných proměnných CUDA. Vlákna jsou organizována do bloků a bloky do mřížky (grid). Existuje omezení, že každý blok



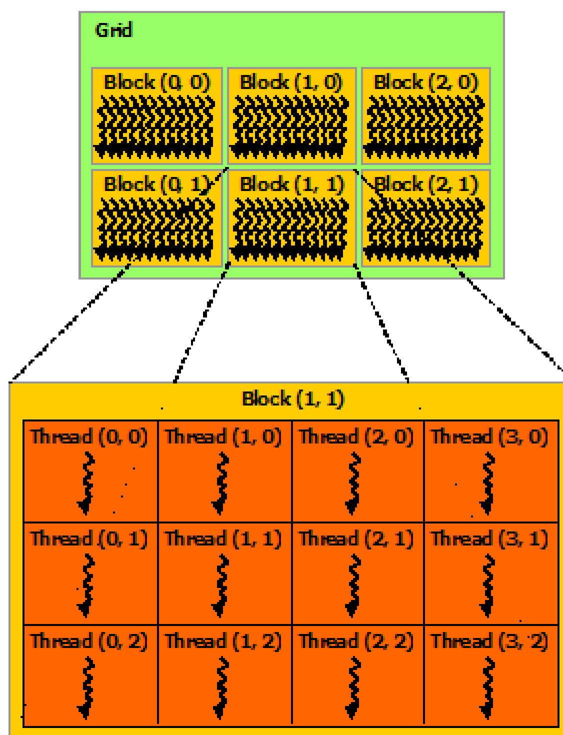
Obrázek 3.1: Schématický rozdíl mezi tranzistory na CPU a GPU

může obsahovat nejvýše 1024 vláken, to je z důvodu, že tyto vlákna jsou vykonávána jediným procesorem a dělí se o omezenou velikost sdílené paměti. Mřížka stejně jako bloky může mít jednu, dvě nebo tři dimenze, což při zavolání kernelu v kódu určují parametry v závorkách `<<<blocks_per_grid, threads_per_block>>>`. Obrázek 3.2 popisuje hierarchii vláken v CUDA pro konkrétní počet bloků a vláken v bloku.

Vlákna mohou přistuovat k různým typům paměti. Každé vlákno má k dispozici vlastní privátní lokální paměť, každý blok vláken má sdílenou paměť viditelnou pouze pro všechna vlákna daného bloku a všechna vlákna mřížky mají přístup do globální paměti. Dále existují další dvě read-only typy paměti přístupné všemi vlákny mřížky – paměť konstant a paměť textur.

3.5.1 Divergence warpu

CUDA je příkladem takzvané SIMT (single instruction multiple threads) architektury, která se vyznačuje tím, že v jeden okamžik mohou vlákna vykonávat pouze jednu instrukci najednou. V CUDA jsou vlákna každého bloku mřížky při vykonávání kernelu rozdělena na takzvané *warpy*, což jsou skupiny 32 vláken, která vykonávají vždy stejnou instrukci. Pokud by nějaké vlákno jednoho *warpu* například kvůli podmínce `if` v kódu mělo vykonávat jinou instrukci než ostatní, dochází k divergenci uvnitř warpu. Warp musí provést obě větve, ale v jeden okamžik může provádět pouze jednu z nich, provádění instrukcí druhé větve se tak musí střídat s první, čímž dochází k pomalejšímu vykonávání kódu kernelu. To může v extrémním případě vést k tomu, že každé vlákno warpu je vykonáváno sekvenčně. Při návrhu aplikací pro CUDA je tedy důležité co nejvíce minimalizovat divergence warpů. Architektura Volta přináší



Obrázek 3.2: Hierarchie vláken v CUDA pro dvoudimenzionální mřížku a bloky

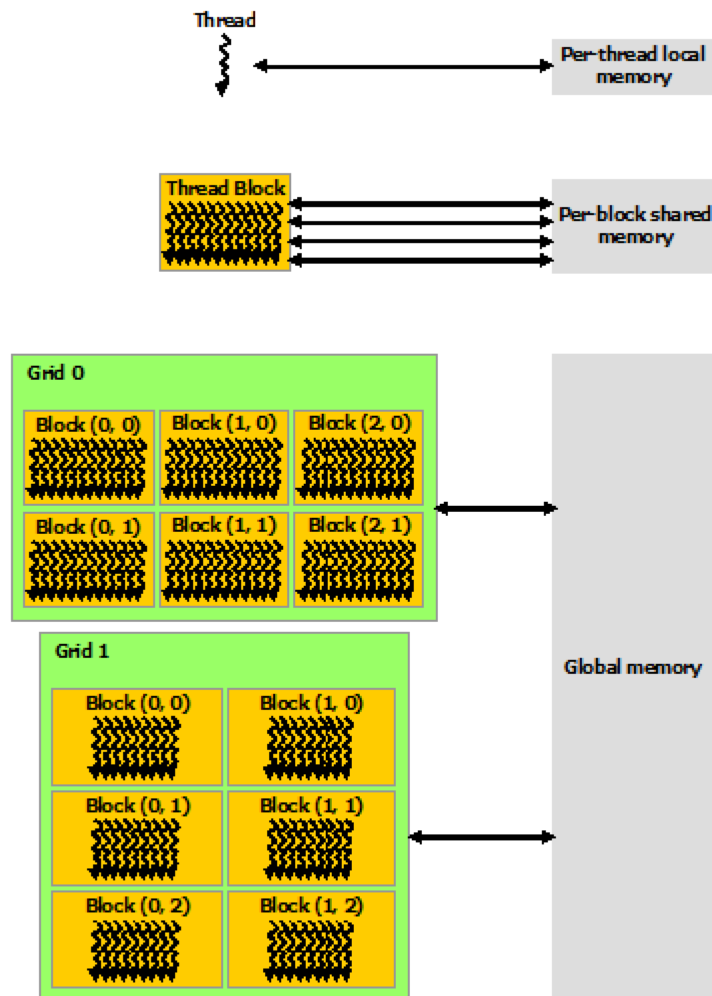
vylepšení, které optimalizuje provádění vláken uvnitř warpu, které vykonávají odlišné instrukce, jak je uvedeno v [20].

3.5.2 Atomické operace

CUDA podporuje řadu atomických aritmetických funkcí [9]. Jsou to funkce do jejichž provádění žádné jiné vlákno nemůže zasahovvat, neboli je zaručené, že k operandům atomické funkce nemá žádné jiné vlákno přístup a nemůže je změnit před dokončením atomické funkce. Zde je uvedeno několik atomických aritmetických funkcí včetně jejich významu.

- ATOMICADD($address, val$) – načte hodnotu *old* uloženou na adrese *address*, spočítá $old + val$ a výsledek uloží zpět na adresu *address*. Funkce vrací *old*.

- ATOMICSUB($address, val$) – načte hodnotu *old* uloženou na adrese *address*, spočítá $old - val$ a výsledek uloží zpět na adresu *address*. Funkce vrací *old*.



Obrázek 3.3: Hierarchie paměti v CUDA

- ATOMICMIN$(address, val)$ – načte hodnotu *old* uloženou na adrese *address*, spočítá minimum z *old* a *val* a výsledek uloží zpět na adresu *address*. Funkce vrací *old*.

- ATOMICMAX$(address, val)$ – načte hodnotu *old* uloženou na adrese *address*, spočítá maximum z *old* a *val* a výsledek uloží zpět na adresu *address*. Funkce vrací *old*.

- ATOMICINC$(address, val)$ – načte hodnotu *old* uloženou na adrese *address*, spočítá $(old >= val)$? $0 : (old + 1)$ a výsledek uloží zpět na adresu *address*. Funkce vrací *old*.

3.6 Paralelní BFS na CUDA

Jak bylo uvedeno v podsekci 2.3.3 Goldbergův algoritmus lze urcyhlit prováděním globálního přepočítání výšek pomocí BFS, proto při jeho paralelní modifikaci pro architekturu CUDA se nabízí využít i pralelní algoritmus BFS pro stejnou architekturu. Ten může zaprvé pracovat rychleji než jeho sekvenční verze pro CPU a hlavně se tím eliminuje nutnost kopírování dat mezi hlavní pamětí CPU a pamětí CUDA před a po provedení globálního přepočítání výšek.

Několik paralelních variant BFS pro CUDA je uvedeno spolu s jejich měřením v [11]. Relativně dobře ve srovnání s ostatními uvedenými z měření vyšel algoritmus ze sekce „Vertex Frontier with Iteration Counter" pocházející z [10], jeho pseudokód je uveden v algoritmu 3.6.1.

Algoritmus 3.6.1: Paralelní BFS

input : graf $G = (V, E)$; počáteční vrchol $v \in V$
output: funkce d přiřazující vrcholům jejich vzdálenost od v

```
1  forall u ∈ V\{v} do
2  |  d(u) ← ∞                                        ◁ inicializace vzdáleností
3  d(v) ← 0
4  level ← 0                                          ◁ udává současnou hloubku hranice
5  do
6  |  stop ← TRUE                                     ◁ globální sdílená proměnná
7  |  d, stop ← SHIFT_FRONTIER(G, d, level, stop)     ◁ procedura 3.6.2
8  |  level ← level + 1
9  while stop = FALSE;
10 return d
```

Algoritmus využívá pojem *hranice*, což je množina vrcholů se stejnou určitou vzdáleností od počátečního vrcholu. Jeho myšlenka spočívá v opakovaném posouvání *hranice*. Na začátku *hranici* tvoří pouze počáteční vrchol, jehož vzdálenost je 0, ostatní vrcholy mají iniciální vzdálenost ∞. Následně se provádí iterace počínaje iterací nula. V každé i-té iteraci se každému vrcholu u přiřadí jedno vlákno, které zkontroluje, zda vzdálenost u je rovna i, a pokud ano, vlákno pro všechny jeho nenalezené sousedy (ty se vzdáleností ∞) nastaví jejich vzdálenost na $i + 1$, čímž posune *hranici* o jednu úroveň dál. Triviálně platí, že počet iterací je roven excentricitě počátečního vrcholu, neboli jeho vzdálnosti k nejvzdálenějšímu dosažitelnému vrcholu. Důležité je si uvědomit, že navzdory případným vícenásobným současným zápisum vláken do stejné sdílené proměnné na řádcích 6 a 7 procedury 3.6.2, operace nemusí být atomická a nevede k chybnému výpočtu, protože vlákna daným proměnným vždy přiřazují stejnou správnou hodnotu.

Procedura 3.6.2: Posun hranice

input : graf $G = (V, E)$; funkce vzdálenosti d; číslo $level \in \mathbb{N}_0$; bool příznak *stop*

output: funkce d s posunutou *hranicí* o jednu úroveň; bool příznak *stop* udávající zda byly nalezeny všechny dosažitelné vrcholy

```
1 forall u ∈ V paralelně do                    ◁ jedno vlákno ↔ jeden vrchol
2   if d(u) = level then
3     forall (u, v) ∈ E do
4       if d(v) = ∞ then
5         d(v) ← level + 1
6         stop ← FALSE                         ◁ byl nalezen dosud nenalezený vrchol
7 return d, stop
```

3.7 Existující algoritmy hledající maximální tok

V této sekci jsou představeny čtyři existující algoritmu pro CUDA hledající maximální tok v síti, liší se především způsobem synchronizace a přiřazování vláken.

3.7.1 Synchronizace atomickými operacemi

Algoritmy této podsekce se vyznačují použitím atomických operací k synchronizaci vláken, které CUDA podporuje [9].

3.7.1.1 Vlákna přiřazována vrcholům

První představený algoritmus pochází z [12]. Algoritmus uvedený v publikaci hledá pouze hodnotu maximálního toku a případně minimální řez, ale lze ho jednoduše upravit i pro hledání maximálního toku, jak bylo uvedeno v podsekci 2.3.4. Pro heuristiku globálního přepočítání výšek využívá CPU, argumentem je nízká výkonnost BFS prováděného na GPU. Vzhledem k tomu, že publikace vyšla roku 2012, takové tvrzení na novějších a výkonnějších GPU už nemusí platit, jak je uvedeno v [22], kde sekvenční BFS zaostává za variantami pro GPU ve většině případů.

Algoritmus po inicializaci provádí hlavní smyčku na řádcích 2 až 5 algoritmu 3.7.1, kde se nejdříve provede globální přepočítání výšek a poté se několikrát provádí procedura 3.7.2, ve které se převádějí přebytky a zvyšují výšky vrcholů. Tato hlavní smyčka se opakuje, dokud se přebytek ve stoku nerovná odtoku ze zdroje, tedy vlna se změní na tok.

V proceduře 3.7.2 se vlákna přiřadí vrcholům tak, že jedno vlákno má na starosti jeden vrchol. Každý vrchol kromě zdroje a stoku s kladným přebytkem se poté pokusí převést co největší část svého přebytku do jednoho ze svých

sousedních vrcholů pomocí atomických aritmetických operací. Pokud do žádného sousedního vrcholu přebytek nelze převést, tedy z vrcholu nevede žádná klesající hrana, zvýší vrchol svou výšku o jedna. Toto paralelní převádění přebytků a zvedání vrcholů lze chápat jako jako paralelní provádění smyčky na řádcích 2 až 6 sekvenčního Goldbergova algoritmu 2.3.1. Jednou nevýhodou této procedury může být vysoká divergence vláken při vykonávání smyčky na řádcích 4 až 7, pokud jsou stupně vrcholů výrazně odlišné.

Algoritmus 3.7.1: Maximální tok - CUDA: atomic, vrcholově centr.

```
input  : síť S = (V_h, E, z, s, c)
output: maximální tok f sítě S

1  S, f ← INIT(S)                                   ◁ procedura 2.3.2
2  while −f^Δ(z) ≠ f^Δ(s) do
3      globální přepočítání výšek algoritmem BFS
4      repeat limit times
5          S, f ← VERTEX_TRANSFER_RELABEL(S, f)     ◁ procedura 3.7.2
6  return f
```

Procedura 3.7.2: Převedení přebytků: atomic, vrcholově centrický

```
input  : síť S = (V_h, E, z, s, c); vlna f v síti S
output: síť S a vlna f v síti S, kde vrcholy s kladným přebytkem
        převedly přebytek po jedné hraně, nebo zvýšily svou výšku

1  forall u ∈ V_h paralelně do                      ◁ jedno vlákno ↔ jeden vrchol
2      if f^Δ(u) > 0 ∧ u ≠ z, s then
3          h' ← ∞
4          forall (u, v) ∈ E : r((u, v)) > 0 do
5              if h' > h(v) then
6                  v' ← v                           ◁ vrchol do kterého lze převádět
7                  h' ← h(v)
8          if h(u) > h' then
9              δ ← MIN{f^Δ(u), r((u, v'))}          ◁ kolik lze převést
10             ε ← MIN{δ, f((v', u))}               ◁ kolik lze odečíst v protisměru
11             ATOMICSUB(f((v', u)), ε)
12             ATOMICADD(f((u, v')), δ − ε)
13             ATOMICSUB(f^Δ(u), δ)
14             ATOMICADD(f^Δ(v'), δ)
15         else
16             h(u) ← h(u) + 1
17 return S, f
```

3.7.1.2 Vlákna přiřazována hranám

Další algoritmus využívající atomické operace je popsán v [13]. Algoritmus se od předchozího liší hlavně tím, že vlákna při převádění přebytku nepřiřazuje vrcholům ale hranám, což má podle autorů pozitivní vliv na efektivitu z důvodu nízké divergence vláken. U grafů, které mají odlišné stupně vrcholů, může při vrcholové centricitě (přiřazování vláken vrcholům) docházet například k různě dlouhým `for` smyčkám, když vlákno iteruje přes všechny své sousední vrcholy. Článek neuvádí použití globálního přepočítání výšek, avšak uvádí paralelní variantu BFS pro nalezení minimálního řezu velmi podobnou uvedenému algoritmu ze sekce 3.6.

Algoritmus 3.7.3 po inicializaci a globálním přepočítání výšek opakovaně provádí operaci převedení přebytků po hranách následovanou operací zvedání vrcholů, to je realizováno pomocí dvou procedur 3.7.4 a 3.7.5. Algoritmus končí, když se odtok ze zdroje rovná přítoku do stoku.

Algoritmus 3.7.3: Maximální tok - CUDA: atomic, hranově centr.

input : síť $S = (V_h, E, z, s, c)$
output: maximální tok f sítě S

```
1 S, f ← INIT(S)                                  ◁ procedura 2.3.2
2 forall v ∈ V_h do
3     u.lift ← TRUE                               ◁ atribut vrcholu
4 while −f^Δ(z) ≠ f^Δ(s) do
5     globální přepočítání výšek algoritmem BFS
6     repeat limit times
7         S, f ← EDGE_TRANSFER(S, f)              ◁ procedura 3.7.4
8         S ← EDGE_RELABEL(S, f)                  ◁ procedura 3.7.5
9 return f
```

Operace převedení přebytků je popsána v proceduře 3.7.4. V té se každé hraně (u, v), přiřadí jedno vlákno, které se pokusí převést přebytek z prvního vrcholu do druhého, pokud jsou splněny podmínky pro převedení ($h(u) > h(v)$ a $r((u, v)) > 0$), za použití atomických operací. Vlákna nejdříve zjistí, kolik mohou ve skutečnosti převést, protože z jednoho vrcholu může vést více hran. Proto na řádku 4 každé vlákno atomicky odečte od přebytku výchozího vrcholu hrany rezervu dané hrany a podle vrácené hodnoty *old* pak vlákno zjistí, zda může převádět, nebo ho ostatní vlákna v převádění „předběhla“. Je důležité si uvědomit, že přebytek se nikdy nepřevádí po jedné hraně oběmi směry najednou, protože by to porušovalo podmínku výšek (přebytek se vždy převádí do nižšího vrcholu), úprava hodnot $f((u, v))$ a $f((v, u))$ tak nemusí být atomická. V proceduře se dále aktualizuje bool příznak $u.lift$ udávající, zda z vrcholu u nevede klesající hrana s kladnou rezervou, a tudíž vrcholu u

lze zvýšit výšku.

Druhá operace zvedající vrcholy je popsána v proceduře 3.7.5. Ta je rozdělena na tři fáze, v první se inicializuje proměnná $u.new_height$ udávající novou možnou výšku vrcholu u. Ve druhé se každé hraně (u, v) přiřadí jedno vlákno, které zkontroluje, zda vrcholu u lze zvýšit výšku na hodnotu $h(v) + 1$ (kladný přebytek vrcholu u a rezerva hrany (u, v) a pravdivý příznak $lift$), a pokud ano tak mu pomocí atomických operací upraví proměnnou $u.new_height$ na minimum ze současné hodnoty $u.new_height$ a $h(v) + 1$. Po skončení druhé fáze tak mají všechny vrcholy u, které lze zvednout, hodnotu $u.new_height$ nastavenou na nejvyšší možné výšce ale takové, aby výška všech klesajících hran z vrcholu u klesala nejvíce o jedna. Ve třetí fázi se vrcholy paralelně zvedají do výšek určených hodnotou $u.new_height$ ze druhé fáze.

Procedura 3.7.4: Převedení přebytků: atomic, hranově centrický

input : síť $S = (V_h, E, z, s, c)$; vlna f v síti S
output: síť S a vlna f v síti S, kde se po hranách poslalo největší možné množství přebytku

```
 1 forall (u, v) ∈ E paralelně do                     ◁ jedno vlákno ↔ jedna hrana
 2   if r((u, v)) > 0 ∧ u ≠ z, s ∧ h(u) > h(v) then
 3     old ← atomicSub(f^Δ(u), r((u, v)))
 4     if old ≥ r((u, v)) then                        ◁ lze převést množství r((u, v))
 5       atomicAdd(f^Δ(v), r((u, v)))
 6       příslušně uprav tok f po hranách (u, v) a (v, u)
 7     else if old > 0 then                           ◁ lze převést množství old
 8       atomicAdd(f^Δ(u), r((u, v)) − old)
 9       atomicAdd(f^Δ(v), old)
10       příslušně uprav tok f po hranách (u, v) a (v, u)
11     else                                           ◁ nelze nic převést
12       atomicAdd(f^Δ(u), r((u, v)))
13     if r((u, v)) > 0 then
14       u.lift ← false                               ◁ vrchol u nelze zvednout
15 return S, f
```

Procedura 3.7.5: Zvednutí vrcholů: atomic, hranově centrický

input : síť $S = (V_h, E, z, s, c)$; vlna f v síti S
output: síť S se zvednutými vrcholy

```
1  forall v ∈ V_h\{z, s} paralelně do            ◁ jedno vlákno ↔ jeden vrchol
2      v.new_height ← ∞
3  forall (u, v) ∈ E paralelně do                ◁ jedno vlákno ↔ jedna hrana
4      if u.lift = TRUE ∧ f^Δ(u) > 0 ∧ r((u, v)) > 0 ∧ u ≠ z, s then
5          ATOMICMIN(u.new_height, h(v) + 1)
6  forall v ∈ V_h\{z, s} paralelně do            ◁ jedno vlákno ↔ jeden vrchol
7      if v.lift = TRUE ∧ f^Δ(v) > 0 then
8          h(v) ← v.new_height
9      v.lift ← TRUE
10 return S
```

3.7.2 Synchronizace fázemi

Atomické operace využívané algoritmy předchozí podsekce mohou při vícenásobné úpravě stejné proměnné dosahovat značného zpomalení, protože vlákna musí být při aplikování atomické operace na jedno paměťové místo serializována. Tento nedostatek lze vyřešit použitím synchronizační strategie, ve které v žádném kroku nemůže dojít k vícenásobné úpravě jednoho místa v paměti a operace s pamětí nemusí být atomické. Na začátku této kapitoly v algoritmu 3.3.1 byla uvedena metoda synchronizace pomocí *pulzu*. Podobnou synchronizaci vláken používají algoritmy uvedené v následujících podsekcích, ovšem díky tomu, že všechna vlákna mají přístup do globální paměti, je to podstatně jednodušší.

3.7.2.1 Statické převádění přebytků

Algoritmus popsaný v [14], se specializuje na hledání minimálního toku pro mřížkové grafy, což využívá při segmentaci obrazu, ale lze ho použít i na obecné grafy.

Algoritmus 3.7.6 nejdřív inicializuje síť a vlnu a začne provádět hlavní smyčku na řádcích 2 až 9. V té se nejdříve globálně přepočítají výšky všech vrcholů. Pak se m-krát provede pulz (viz sekce 3.3) o dvou fázích, což je realizováno procedurami 3.7.9 a 3.7.7. Poté se lokálně zvýší výšky vrcholů v proceduře 3.7.8, to se spolu s cyklem pulzu opakuje k-krát a potom se znovu globálně přepočítají výšky vrcholů a celý cyklus se opakuje. Algoritmus končí při splnění rovnosti odtoku ze zdroje a přítoku do stoku. Autoři uvádí, že nejlepších výsledků dosahovaly hodnoty $m = 1$ a $k = 6$.

V proceduře 3.7.9 se všem vrcholům kromě zdroje a stoku přiřadí jedno

vlákno. Pak se pro všechny vrcholy kromě zdroje a stoku převede největší možné množství přebytku do jejich sousedních vrcholů, ovšem bez zvýšení hodnoty přebytku v cílovém vrcholu. Poslané množství po hraně (u, v) se uloží do sdíleného pole p o délce $|E|$.

V proceduře 3.7.7 se poté všem vrcholům přiřadí jedno vlákno a z pole p se k přebytku příslušných vrcholů přičte množství poslané od sousedních vrcholů z první fáze pulzu.

Procedura 3.7.8 má na starosti zvyšování výšek vrcholů. V té se opět každému vrcholu přiřadí jedno vlákno, a pokud má příslušný vrchol kladný přebytek a je různý od zdroje a stoku zvýší se mu potenciálně výška. Novou výšku vrcholu u pak určuje nejnižší sousední vrchol, do kterého vede hrana s kladnou rezervou. Může se stát, že takový nejnižší vrchol má výšku $h(u) - 1$, v tom případě se výška vrcholu u nezmění.

Algoritmus 3.7.6: Maximální tok - CUDA: fáze, statické převádění

input : síť $S = (V_h, E, z, s, c)$
output: maximální tok f sítě S

```
1  S, f ← INIT(S)                                          ◁ procedura 2.3.2
2  while −f^Δ(z) ≠ f^Δ(s) do
3  │ globální přepočítání výšek pomocí BFS algoritmu
4  │ repeat k times
5  │ │ repeat m times
6  │ │ │ p ← vynulované pole délky |E|
7  │ │ │ f, p ← VERTEX_PUSH(S, f, p, V_h)                  ◁ procedura 3.7.9
8  │ │ └ f ← VERTEX_PULL(S, f, p, V_h)                     ◁ procedura 3.7.7
9  │ └ S ← VERTEX_LOCAL_RELABEL(S, f)                      ◁ procedura 3.7.8
   └
10 return f
```

Procedura 3.7.7: Načtení přebytků do vrcholů: vrcholově centrický

input : síť $S = (V_h, E, z, s, c)$; vlna f v síti S; vyplněné pole poslaných přebytků p; množina vrcholů $M \subseteq V_h$
output: vlna f v síti S, kde vrcholy $v \in M$ načetly přebytek poslaný od sousedních vrcholů

```
1  forall v ∈ M paralelně do                  ◁ jedno vlákno ↔ jeden vrchol
2  │ forall (u, v) ∈ E do
3  │ └ f^Δ(v) ← f^Δ(v) + p[(u, v)]
   └
4  return f
```

Procedura 3.7.8: Zvednutí vrcholů: vrcholově centrický

input : síť $S = (V_h, E, z, s, c)$; vlna f v síti S
output: síť S se zvednutými vrcholy

```
1 forall u ∈ V_h paralelně do                          ◁ jedno vlákno ↔ jeden vrchol
2   if f^Δ(u) > 0 ∧ u ≠ z, s then
3     u.new_height ← ∞
4     forall (u, v) ∈ E : r((u, v)) > 0 do
5       u.new_height ← MIN{u.new_height, h(v) + 1}
6     h(u) ← u.new_height
7 return S, f
```

Procedura 3.7.9: Poslání přebytků z vrcholů: vrcholově centrický

input : síť $S = (V_h, E, z, s, c)$; vlna f v síti S; pole p poslaných přebytků k vyplnění; množina vrcholů $M \subseteq V_h$
output: vlna f v síti S, kde vrcholy $v \in M$ převedly svůj přebytek; vyplněné pole poslaných přebytků p

```
1 forall u ∈ M paralelně do                            ◁ jedno vlákno ↔ jeden vrchol
2   if u ≠ z, s then
3     forall (u, v) ∈ E : r((u, v)) > 0 ∧ h(u) = h(v) + 1 do
4       if f^Δ(u) = 0 then
5         break
6       δ ← MIN{f^Δ(u), r((u, v))}                     ◁ kolik lze převést
7       převeď δ jednotek toku po hraně (u, v), ale nezvyšuj f^Δ(v)
8       p[(u, v)] ← δ                                  ◁ pole poslaných přebytků
9 return f, p
```

3.7.2.2 Dynamické převádění přebytků

Předchozí algoritmus převáděl přebytky vždy ze všech vrcholů najednou, algoritmus uvedený v [15] volí strategii postupného dynamického převádění od nejvyšších vrcholů k nižším. K tomu používá jednoduchou datovou strukturu zvanou *lattice*, kterou si lze představit jako seznam množin vrcholů, kde každá množina obsahuje všechny vrcholy sítě se stejnou výškou. Ze struktury lze tedy jednoduše vybírat a přidávat do ní vrcholy se stejnou výškou. Algoritmus se liší od všech předešlých tím, že výšky vrcholů mění pouze pomocí globálního přepočítání výšek.

Algoritmus 3.7.10 začíná inicializací a v hlavní smyčce na řádcích 2 až 8 nejprve globálně přepočítá výšky vrcholů pomocí paralelního BFS 3.6.1. Autoři algoritmu v sekci IV na straně 3 uvádí metodu, jak při každém posunu

hranice rovněž zkonstruovat množinu všech vrcholů tvořící hranici, která se pak vkládá do *lattice*, k tomu využívají paralelní prefixový součet. Po doběhnutí BFS se vykonává smyčka na řádcích 4 až 8, ve které se postupuje od vrcholů s největší výškou k nejnižším. Z *lattice* se v každé iteraci odebere množina vrcholů s největší výškou lvl a z těchto vrcholů se dvoufázově převede přebytek do nižších vrcholů. Nejdříve se pomocí procedury 3.7.9 pošle přebytek vrcholům s výškou $lvl-1$, tyto vecholy pak načtou poslaný přebytek v proceduře 3.7.7. V další iteraci se posílá přebytek z vrcholů s výškou $lvl-1$ do vrcholů s výškou $lvl-2$...nakonec se takto převede přebytek z vrcholů o výšce 1 do stoku. Převádění přebytků je tedy opět realizováno bez použití atomických operací. Poté se opět globálně přepočítají výšky vrcholů a znovu zkonstruuje struktura *lattice*, aby odpovídala nynějšímu stavu sítě. Celý tento cyklus se opakuje, dokud se z vlny nestane tok.

Algoritmus 3.7.10: Maximální tok - CUDA: fáze, dynam. převádění

input : síť $S = (V_h, E, z, s, c)$; vlna f v síti S
output: síť S a tok f v síti S s převedenými přebytky

```
1 S, f ← INIT(S)                                        ◁ procedura 2.3.2
2 while −f^Δ(z) ≠ f^Δ(s) do
3     lattice ← BFS konstruující lattice
4     while lattice obsahuje alespoň dva seznamy vrcholů do
5         z lattice odeber seznam list s největšími výškami vrcholů lvl
6         p ← vynulované pole délky |E|
7         f, p ← VERTEX_PUSH(S, f, p, list)             ◁ procedura 3.7.9
8         f ← VERTEX_PULL(S, f, p, lattice[lvl − 1])    ◁ procedura 3.7.7
9 return S, f
```

3.8 Nový algoritmus využívající obarvení hran

V této sekci je představený nový algoritmus pro CUDA, využívající obarvení hran sítě.

Algoritmus 3.8.1 po inicializaci sítě a počáteční vlny nastaví proměnné všech vrcholů $u.lift$ na true, tato proměnná udává, zda lze vrcholu u zvýšit výšku. Poté se obarví hrany sítě tak, že se každé hraně přiřadí barva (přirozené číslo) tak, že pro všechny vrcholy platí, že všechny k nim incidentní hrany mají různou barvu, pokud neuvažujeme orientaci hran. Následuje hlavní smyčka algoritmu na řádcích 6 až 11, ve které se nejdříve provádí globální přepočítání výšek, následuje převádění přebytků v proceduře 3.8.2 pro všechny barvy sítě. Po proiterování všech barev následuje procedura 3.8.3 zvyšující výšky vrcholů. Po nějakém daném počtu opakování této smyčky (převádění přebytků a zvýšení výšek) se opět globálně přepočítají výšky, aby lépe odpo-

vídaly současnému stavu sítě a celá hlavní smyčka se opakuje až do doby, než se z vlny stane tok.

V proceduře 3.8.2 při převádění přebytků jsou vlákna přiřazována hranám. Každé vlákno převede přebytek po hraně (u, v), pouze pokud hrana a incidentní vrcholy splňují podmínky pro převedení a pokud má hrana (u, v) barvu dané iterace. Pokud i po převedení má hrana (u, v) kladný přebytek nastavuje se proměnná $u.lift$ na FALSE, protože vrcholu v nelze zvýšit výšku (vede z něj klesající hrana (u, v) s kladnou rezervou). Obarvení hran způsobuje, že v žádný okamžik dvě různá vlákna nikdy neupravují současně jedno paměťové místo, takže převádění pebytku lze provést bez použití atomických operací. To platí, přestože dvě různá vlákna mohou ve stejný okamžik provádět tuto proceduru pro hrany (u, v) a k ní opačnou (v, u), ve skutečnosti se díky podmínce výšek na řádku 2 procedury 3.8.2 nikdy nepřevádí v obou směrech najednou a nejvýše jedno vlákno ve skutečnosti přebytek převede. Díky tomu, že vlákna jsou při převádění přebytků přiřazována hranám a ne vrcholům, mají všechna vlákna vyvážené množství práce, i pokud jsou stupně vrcholů výrazně rozdílné.

V proceduře 3.8.3 se každému vrcholu přiřadí jedno vlákno, které zkontroluje, zda je možné vrcholu zvýšit výšku. Pokud lze, vlákno zvýší výšku vrcholu o jedna. Nakonec se nastaví proměnnná vrcholu $u.lift$ na true, která se bude moct měnit v další iteraci při převádění přebutku a bude se kontrolovat při dalším vykonávání této procedury.

Na obarvení hran grafu lze použít Misra-Grieův algoritmus [16]. Ten dokáže obarvit hrany libovolného neorientovaného grafu v čase $\mathcal{O}(|V| \cdot |E|)$ za použití nejvíce $\Delta + 1$ barev, kde Δ je největší stupeň vrcholu obarvovaného grafu. Nebo lze použít naivní „hladový“ algoritmus obarvující hrany uvedený například zde [23], který může běžet rychleji, ovšem může použít více barev. Protože síť je orientovaný graf, budou všechny hrany (u, v) a k ní opačná (v, u) mít stejnou barvu.

Algoritmus nepotřebjue žádné atomické operace a jeho dílčí procedury by měly vykazovat nízkou míru divergence vláken – žádné vlákno CUDA nevykonává `for` smyčku. Naproti tomu jeho úzké hrdlo by mohla být inicializace konkrétně sekvenční obarvení hran grafu, které může trvat nezanedbatelné množství času.

3.8.1 Důkaz správnosti

Na řádcích 1 až 5 algoritmu 3.8.1 se inicializuje síť a obarví hrany. Ukončovací podmínka na řádku 6 vychází z lemmatu 2.3.5. Tělo smyčky na řádcích 7 až 11 se skládá z heuristiky globálního přepočítání výšek popsané v podsekci 2.3.3 a smyčky na řádcích 8 až 11, kterou lze opakovat libovolně-krát.

Ve této smyčce se nejprve pro každou barvu sítě provádí procedura 3.8.2. V té se pro všechny hrany (u, v) obarvené barvou dané iterace zkontrolují podmínky: 1. rezerva hrany (u, v) je kladná, 2. vrchol u je různý od z, s a 3. výška

$h(u)$ je větší než $h(v)$, což zahrnuje stejné podmínky pro převedení přebytku,k teré jsou v původním Goldbergově algoritmu 2.3.1, až na podmínku kladného přebytku vrcholu u, ale nulový přebytek z vrcholu lze převádět, i když to nemá žádný efekt. Pokud hrana všechny podmínky splňuje, převede se po ní přebytek z vrcholu u do vrcholu v podle definice 2.3.3. Pokud dále platí, že rezerva hrany (u, v) je i po převedení přebytku kladná, vrchol u nelze zvednout podle definice 2.3.2 (vede z něj klesající hrana (u, v) s kladnou rezervou) a proto se na řádku 5 nastavuje příznak $lift$ na false. Ačkoli se převádění přebytků provádí paralelně, díky obarvení hran žádné dvě převádějící hrany nikdy nemají žádný společný vrchol a to i přesto, že hrana (u, v) a k ní opačná (v, u) mají stejnou barvu a stejné krajní vrcholy, ovšem přebytek se vždy převádí pouze do nižšího vrcholu, takže vždy nejvýše po jedné z této dvojice hran. Převádění přebytků má tedy stejný efekt, jako kdyby se pro všechny hrany dané barvy provádělo sekvenčně. Procedura 3.8.2 tedy primárně převádí přebytky stejně jako původní sekvenční Goldbergův algoritmus na řádku 4, výšky vrcholů zůstavají po celou dobu stejné a vlna splňuje definici 2.3.1.

Po převedení přebytků se na řádku 11 vykonává procedura 3.8.3. V té se paralelně pro všechny vrcholy u zkontroluje 1. zda jsou různé od zdroje a stoku, 2. mají kladný přebytek a 3. proměnná $u.lift$ vrcholu je nastavena na true. Všimněme si, jak bylo popsáno v předchozím odstavci, že proměnná $lift$ každého vrcholu je nastavena na false, pokud z něj vede alespoň jedna klesající hrana s kladnou rezervou, a nastavena na true v opačném případě. Připomeňme, že v původním Goldbergově algoritmu se vrchol zvedá, pouze pokud 1. je různý od zdroje a stoku, 2. má kladný přebytek a 3, nevede z něj klesající hrana s kladnou rezervou, což jsou stejné tři podmínky pro zvednutí vrcholu. Vrcholu u se tedy při splnění daných tří podmínek zvětšuje výška o jedna, čímž výšky vrcholů zůstanou korektní podle definice 2.3.2. Vlna se v proceduře nijak nemění. □

Algoritmus 3.8.1: Maximální tok - CUDA: obarvené hrany

input : síť $S = (V_h, E, z, s, c)$
output: maximální tok f sítě S

1 $S, f \leftarrow$ INIT(S) ◁ procedura 2.3.2
2 **forall** $v \in V_h$ **do**
3   $u.lift \leftarrow$ TRUE ◁ atribut vrcholu
4 obarvy hrany sítě S ◁ Misra-Griesův algoritmus [16]
5 $colors \leftarrow$ množina barev hran sítě S
6 **while** $-f^{\Delta}(z) \neq f^{\Delta}(s)$ **do**
7   globální přepočítání výšek pomocí BFS algoritmu
8   **repeat** *limit* **times**
9   **forall** $c \in colors$ **do**
10       $S, f \leftarrow$ EDGE_TRANSFER(S, f, c) ◁ procedura 3.8.2
11     $S \leftarrow$ LIFT_VERTICES(S, f) ◁ procedura 3.8.3
12 **return** f

Procedura 3.8.2: Převedení přebytků: obarvené hrany, hranově c.

input : síť $S = (V_h, E, z, s, c)$; vlna f v síti S; barva c
output: síť S a vlna f v síti S, kde se po hranách obarvených barvou c poslalo největší možné množství přebytku

1 **forall** $(u, v) \in E : col((u, v)) = c$ **paralelně do** ◁ jed. vl.↔ jed. hrana
2   **if** $r((u, v)) > 0 \wedge u \neq z, s \wedge h(u) > h(v)$ **then**
3   převeď přebytek po hraně (u, v) ◁ definice 2.3.3
4   **if** $r((u, v)) > 0$ **then**
5       $u.lift \leftarrow$ FALSE
6 **return** S, f

Procedura 3.8.3: Zvednutí vrcholů: obarvené hrany, vrcholově c.

input : síť $S = (V_h, E, z, s, c)$; vlna f v síti S
output: síť S se zvednutými vrcholy

1 **forall** $v \in V_h$ **paralelně do** ◁ jedno vlákno ↔ jeden vrchol
2   **if** $u \neq z, s \wedge f^{\Delta}(u) > 0 \wedge u.lift = TRUE$ **then**
3     $h(u) \leftarrow h(u) + 1$
4   $u.lift \leftarrow$ TRUE
5 **return** S

algoritmus (sekce)	**synchronizace vláken**	**přiřazování vláken**	**lokální změna výšek**
alg. z [12](3.7.1.1)	atomickými operacemi	vrcholům	vždy o jedna
alg. z [13](3.7.1.2)	atomickými operacemi	hranám	podle výšek soused. vrcholů
alg. z [14](3.7.2.1)	fázemi	vrcholům (staticky)	podle výšek soused. vrcholů
alg. z [15](3.7.2.2)	fázemi	vrcholům (dynamicky)	—
nový alg.: CUDA coloured edges (3.8)	obarvením hran	hranám	vždy o jedna

Tabulka 3.1: Porovnání hlavních vlastností uvedených paralelních algoritmů – 1. typ synchronizace vláken při přístupu do sdílené paměti, 2. způsob přiřazování vláken při převádění přebytků a 3. jak se lokálně mění výšky vrcholů

Kapitola 4

Detaily implementace

V této kapitole je uveden způsob a detaily implementace uvedených paralelních algoritmů. Zdrojový kód implementace je dostupný pro přihlášené uživatele na fakultním GitLabu FIT ČVUT na adrese [1] a flash disku přiloženém k fyzické kopii této práce.

4.1 Technologie

Jako programovací jazýk byl zvolen `C++` ve standardu `C++11`, je využívána standardní knihovna jazyka. Pro psaní kódu vykonávaného na GPU je použita CUDA, což je z pohledu psaní kódu pouze rozšíření jazyka `C/C++` o nové konstrukty. Dále jsou použity knihovny Boost [24] a Thrust [25].

4.2 Reprezentace datových struktur v paměti

4.2.1 Graf

Kvůli typu přístupu, kde mnoho vláken za běhu algoritmu přistupuje současně ke stejnému atributu různých vrcholů nebo hran, jsou atributy uloženy jako struktura polí (Structure of Arrays), kde vlákna jednoho warpu mohou přistupovat k sousedícím indexům stejného pole rychleji, jak je uvedeno zde [26]. Všechna tato pole mají prvky o velikosti 32 bitů kromě pole odpovídajícího booleovskému atributu vrcholu *lift* v algoritmech 3.7.3 a 3.8.1, které má prvky 8-bitové z důvodu úspory místa. Všechna data jsou uložena v globální paměti GPU.

Atributy hran (rezerva hrany, zdrojový vrchol hrany, barva hrany. . .) včetně hran zpětných jsou uložené jako struktura polí a platí, že indexy hran (u, v) a k ní opačné (v, u) se liší o 1, tedy jsou uložené v poli vedle sebe, tak že první hrana grafu má index 0 k ní zpětná index 1, další hrana má index 2 k ní zpětná

[1] https://gitlab.fit.cvut.cz/cerveka2/mi-dip_implementation

index 3 atd. Určování indexu zpětné hrany k libovolné hraně lze díky tomu jednoduše implementovat bitovými operacemi bez nutnosti použití příkazu `if`. Každá hrana má své atributy vždy na stejném indexu v několika různých polích. Hranu lze tedy jednoznačně identifikovat jejím indexem a hranu k ní opačnou indexem o jedna větším či menším.

Atributy vrcholů (přebytek vrcholu, výška vrcholu...) jsou uložené také jako struktura polí. Ke každému vrcholu u je udržováno pole indexů, což jsou indexy hran vycházejících z daného vrcholu u. Z daného vrcholu u nelze tedy přistupovat přímo k jeho sousedním vrcholům, ale je nutné index sousedního vrcholu nejdříve dopočítat jako cílový vrchol hrany vycházející z u. Vrchol lze jako hrany jednoznačně určit svým indexem ve všech polích udržujících atributy vrcholů.

4.2.2 Lattice

Jak bylo uvedeno, algoritmus ze sekce 3.7.2.2 využívá datovou strukturu zvanou *lattice*. Ta je implementována jako dvě pole velikosti $2 \cdot |V|$, které se vyplňují při globálním přepočítání výšek. První pole 32-bitových prvků udržuje počty vrcholů s danou výškou tak, že na indexu 0 je počet vrcholů z výškou 1, na indexu 1 je počet vrcholů s výškou 2 atd. Druhé pole ukládá samotné vrcholy jako dynamicky alokované pole 32-bitových indexů vrcholů, což jsou jednotlivé úrovně této struktury. Po globálním přepočítání výšek je druhé uvedené pole vyplněno tak, že na indexu 0 má uložené pole všech vrcholů s výškou 1, na indexu 1 má pole všech vrcholů s výškou 2 atd. Struktura tedy vždy po její konstrukci obsahuje všechny vrcholy sítě, ale jejich umístění dané jejich výškami se může v čase lišit. V této struktuře jsou postupně procházeny jednotlivé úrovně (jednotlivá pole vrcholů o stejné výšce) od nejvyšší po nejnižší, jak je uvedeno v algoritmu 3.7.10 ve smyčce na řádcích 4 až 8. Struktura se po každém tomto průchodu uvede dealokací alokovaných polí do prázdného stavu, kdy neobsahuje žádné vrcholy.

4.3 Implementace algoritmů

Všechny algoritmy používají stejnou reprezentaci grafů v paměti uvedenou v předchozí sekci, což u některých může vést k neefektivnímu přístupu do paměti kvůli častým výpadkům paměti cache, což je například při iteraci přes sousedy nějakého vrcholu v, kde přístupy ke hranám vycházejícím z v vedou na náhodná nesouvislá místa v paměti. Daná reprezentace na druhou nevyžaduje při žádné editaci atributu sítě úpravu více míst v paměti, protože každý atribut hran i vrcholů je v paměti uložen právě jednou.

Všechny algortimy hledající maximální tok vykonávají smyčku `while`, dokud se odtok ze zdroje nerovná přítoku do stoku, ta je u všech implementována jako samostatný kernel vykonávaný jedním vláknem CUDA. Uvnitř tohoto kernelu pak vlákno vykonává funkce nebo spouští další kernely, jedná

se tedy o využití dynamického paralelismu [27]. Procedury jednotlivých algoritmů, které vykonávají nějaké příkazy paralelně pro hrany nebo vrcholy sítě, jsou implementovány jako kernel spuštěný pro daný počet vláken, který je roven buď počtu vrcholů nebo počtu hran sítě. Jednotlivým vláknům se v kernelu podle jejich unikátní pozice, dané pozicí vlákna v bloku a pozicí bloku v celé mřížce vláken CUDA, přiřadí vrchol nebo hrana, pro které pak vykonávají daný kód. Přiřazování vláken funguje tak, že vlákno na pozici i obstarává vrchol nebo hranu s indexem i. V některých případech je spouštení kernelu pro velké množství vláken nahrazeno použitím funkce `memset`, pokud se má pouze vyplnit nějaké pole stejnou hodnotou jako například na řádku 2 procedury 3.7.5.

V algoritmu 3.8.1 se hrany obarvují sekvenčně buď pomocí knihovny Boost implementující Misra-Griesův algoritmus, nebo „hladovým“ algoritmem [23]. Po obarvení se hrany seřadí podle jejich barev pomocí knihovny Thrust, po jejich uložení do vnitřní reprezentace jsou pak pole atributů hran rozdělené na souvislé úseky, kde v každém úseku jsou pouze hrany stejné barvy. Když se pak na řádcích 9 a 10 iteruje přes všechny barvy hran, může se kernel převádějící přebytky spouštět přímo pro potřebný počet vláken daný počtem hran dané barvy a vlákna jednoho warpu přistupují vždy k sousedícím prvkům jednotlivých polí atributů hran.

4.3.1 Paralelní globální přepočítání výšek

Aby se minimalizovalo kopírování mezi paměťmi GPU a CPU je globální přepočítání výšek a tedy algoritmus BFS jako jeho součást vykonáván na GPU. Implementace kopíruje pseudokód uvedeného BFS algoritmu ze sekce 3.6. Všechny řádky algoritmu 3.6.1 vykonává jedno CUDA vlákno, které na řádku 7 spouští kernel pro daný počet vláken posouvající hranici o jednu úroveň a čeká na jeho dokončení všemi vlákny.

V případě, kdy se vykonává globální přepočítání výšek v algoritmu ze sekce 3.7.2.2, je nutné za jeho běhu vyplňovat strukturu *lattice*. To je implementováno podle popisu uvedeného v [15] na obrázku 3 na straně 3. Kernelu realizujícímu proceduru 3.6.2, se přidává argument *traversedNodes*, což je vynulované pole délky $|V|$. V tomto kernelu se při nalezení dosud nenalezeného vrcholu v zapíše na index vrcholu v v poli *traversedNodes* hodnota 1. Po skončení kernelu posouvajícího hranici se in-place paralelně spočítá inkluzivní prefixový součet pole *traversedNodes*, což je realizované funkcí knihovny Thrust. Z pole *traversedNodes* lze pak jednoduše sestavit pole nově nalezených vrcholů se stejnou výškou, protože prvek pole *traversedNodes* na indexu těchto nově nalezených vrcholů má o jedna vyšší hodnotu než prvek na indexu o jedna menším. Tyto vrcholy jsou pak přidány do struktury *lattice* spolu s jejich počtem, který je roven prvku pole $traversedNodes$ na jeho posledním indexu.

4.4 Parametry algoritmů

Pro kernely, které jsou vykonávány více než jedním CUDA vláknem, byla jako velikost jednoho bloku mřížky cuda vláken zvolena hodnota 128, která spolu s hodnotou 256 dosahovala nejlepších výsledků. Některé algoritmy lze parametrizovat délkami jejich vnitřních smyček, zde jsou uvedeny použité hodnoty parametrů.

V algoritmu ze sekce 3.7.1.1 se jako efektivní délka smyčky na řádcích 4 a 5 jevila hodnota v nižším řádu stovek, nakonec byla zvolena hodnota 200.

Algoritmu ze sekce 3.7.1.2 lze měmit délku smyčky na řádcích 6 až 8, použitá délka činí 20.

Jak uvádějí autoři algoritmu ze sekce 3.7.2.1, nejefektivnější hodnotou parametru k je 6 a m je 1 a tyto hodnoty byly použity.

Poslední uvedený algoritmus ze sekce 3.8 lze parametrizovat délkou smyčky na řádcích 8 až 11, zvolená hodnota, která vykazovala relativně nízký čas běhu, je 100.

Kapitola 5

Testování a měření algoritmů

V této kapitole jsou uvedeny výsledky měření implementovaných algoritmů a jejich porovnání.

5.1 Vstupní instance

Měření probíhalo na osmi sítích různého typu popsaných níže.

5.1.1 Sítě DIMACS

Sítě jsou vygenerovány programy, které jsou součástí prvního ročníku implementační soutěže DIMACS [31]. Generátory jsou dostupné z [28].

- **ac_4000:** graf vygenerovaný programem `ac`. Jedná se o úplný acyklický orientovaný graf, jehož hrany mají náhodné kapacity v rozmezí 1 až 10 000.

- **genrmf_128_32:** graf vygenerovaný programem `genrmf`. Skládá z 32 čtvercových mřížek velikosti 128 $\times$ 128, které jsou ve vrstvách za sebou. Z každého vrcholu v i-té vrstvě vede hrana do náhodného vrcholu ve vrstvě následující. Kapacity hran jsou náhodná čísla v rozmezí 10 a 100.

5.1.2 Sítě KaHIP

Sítě založené na grafech z desátého ročníku soutěže DIMACS [32]. Všechny tyto sítě mají jednotkové kapacity všech hran, pouze hrany vedoucí ze zdroje a do stoku mají kapacitu 2 147 483 647, což je nejvyšší možná kladná hodnota, kterou lze uložit znaménkově do 32-bitového celého čísla. Aby nedocházelo k přetečení hodnoty vnitřní reprezentace přebytku ve vrcholech, je graf upraven, aby všechny hrany měly kapacitu 1. Instance jsou dostupné z [29].

- **af_shell9:** síť ze sekce „Based on Other Graphs (Road Networks, Scientific Computing, ...)“ v [29]
- **del_strip20:** síť založená na Delaunayho triangulaci náhodných bodů rozmístěných ve čtverci
- **grid_strip20:** mřížkový graf
- **rgg_strip20:** náhodný geometrický graf

5.1.3 Sítě z počítačového vidění

Mnoho problémů z oboru počítačového vidění lze převést na problém nalezení minimálního řezu v síti, což se využívá například v grafice nebo biomedicínské analýze obrazu. Sítě z této sekce reprezentují právě takové problémy. Instance jsou mřížkového typu a jsou dostupné z [30].

- **BL06-camel-sml:** Síť je odvozená z problému rekonstrukce 3D objektu z několika 2D obrázků. Má rozměry $40 \times 30 \times 42 \times 24$ vrcholů.
- **LB07-bunny-sml:** Síť reprezentuje problém vznikající při rekonstrukci 3D objektu z mnoha bodů v prostoru. Má rozměry $102 \times 100 \times 79$ vrcholů.

Parametry jednotlivých sítí jsou uvedeny v tabulce 5.1. Počet hran nezahrnuje zpětné hrany s nulovou kapacitou.

název sítě	**počet vrcholů**	**počet hran**	**velikost max. toku**
ac_4000	4 000	7 998 000	1 986 298 448
af_shell9	151 457	5 107 415	1 490
BL06-camel-sml	1 209 602	5 963 582	6 413 363
del_strip20	629 147	3 769 500	1 482
genrmf_128_32	524288	2 588 672	895 521
grid_strip20	838 862	3 352 684	1 024
LB07-bunny-sml	805 802	5 040 834	961 163
rgg_strip20	629 147	8 273 741	1 525

Tabulka 5.1: Přehled parametrů vstupních instancí

5.2 Testování správnosti

Maximální tok v síti narozdíl od jeho velikosti není vždy jednoznačný, protože rozložení toku po hranách sítě může vést na maximální tok ve více případech.

Kvůli této nejednoznačnosti se při testování správnosti kontrolovala pouze velikost maximálního toku.

Při testování správnosti byly výstupy implementovaných algoritmů porovnány s výstupem implementační části bakalářské práce [17] zabývající se hledáním maximálního toku v síti, která implementuje hledání velikosti maximálního toku. Testování proběhlo na všech sítích uvedených v tabulce 5.1.

5.3 Měření

Tato sekce uvádí detaily měření algoritmů.

5.3.1 Prostředí

Měření probíhalo na fakultním serveru *STAR*, který má následující hardwarové parametry.

- **CPU:**
  - název modelu: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
  - počet fyzických jader: 6 (logických 12)
  - dostupná paměť: 32GB
  - frekvence: 2 100 MHz (turbo 2 600 MHz)[2]
- **GPU:**
  - název modelu: GeForce RTX 2080 Ti
  - počet jader CUDA: 4 352
  - dostupná paměť: 11 GB typu GDDR6
  - frekvence: 1 350 MHz (boost 1 545 MHz)
  - compute capability: 7.5 (Turing)[3]

Překladačem zdrojového kódu v `C++` je `g++` ve verzi 4.8.5, pro CUDA je to `nvcc` ve verzi 10.0.130. Použité přepínače při kompilaci jsou: `-g -x cu -std=c++11 -O3 -Xcompiler -Wall,-Wno-pedantic -c -dc -arch=sm_70 -gencode=arch=compute_70,code=sm_70 -rdc=true`.

[2] https://ark.intel.com/content/www/us/en/ark/products/75789/intel-xeon-processor-e5-2620-v2-15m-cache-2-10-ghz.html

[3] https://www.nvidia.com/cs-cz/geforce/graphics-cards/rtx-2080-ti

5.3.2 Výsledky měření

Naměřené časy běhů algoritmů pro vstupní instance jsou uvedené v tabulce 5.4. Tabulka uvádí časy pro všechny implementované paralelní algoritmy a sekvenční Goldbergův algoritmus, jehož implementace pochází z [17]. Pokud doba běhu překročila jednu hodinu, je v tabulce uvedená hodnota +3600.

Na první pohled je vidět, že sekvenční Goldbergův algoritmus má relativně stabilní časy a předhání všechny paralelní implementace i o několik řádů, pouze algoritmus ze sekce 3.7.1.1 je rychlejší pro instanci s názvem „af_shell9“. Z paralelních implementací nejnižších časů dosáhl algoritmus uvedený ve druhém sloupci následovaný algoritmem ze třetího sloupce, oba využívají k synchronizaci vláken při přístupu do sdílené paměti atomické operace. Algoritmy ze čtvrtého a pátého sloupce využívající k synchronizaci metodu fází doběhly s až o řády horšími časy.

Doba běhu algoritmů uvedených především v posledních dvou sloupcích vykazuje vysokou závislost na typu vstupní sítě, což je rozebráno v následujících podsekcích.

5.3.2.1 Algoritmus využívající *lattice*

V tabulce 5.2 jsou uvedeny detaily struktury *lattice* během vykonávání algoritmu 3.7.10 pro vstupní instance. Druhý sloupec určuje, kolikrát se struktura *lattice* tvořila při globálním přepočítání výšek, dokud se nenašel maximální tok. Třetí sloupec udává aritmetický průměr počtů úrovní vrcholů ve struktuře po jejím vytvoření. Pro instanci s názvem „genrmf_128_32“ algoritmus nedoběhl v rozumném čase a data pro ni nejsou známa.

Lze si všimnout závisloti mezi rychlostí běhu algoritmu a hodnotami v tabulce 5.2. Algoritmus měl nejnižší časy pro instance „ac_4000“ a „LB07-bunny-sml“, ve kterých se převáděl přebytek z jedné úrovně *lattice* do nižší úrovně dohromady $22 \cdot 1.8 = 39.6$-krát, respektive $28 \cdot 82 = 2\,296$-krát. Naopak největší naměřené časy jsou pro instance „del_strip20“ a „rgg_strip20“, kde se převádělo $87 \cdot 1110.5 = 96\,613.5$-krát, respektive $110 \cdot 648.4 = 71\,324$-krát.

5.3.2.2 Algoritmus využívající obarvení hran

V tabulce 5.3 jsou uvedeny detaily obarvování jednotlivých vstupních instancí v algoritmu 3.8.1. Pro obarvování sítě byl použitý naivní hladový „algoritmus“ z [23], protože Misra-Griesův algoritmus poskytovaný knihovnou *Boost* běžel výrazně pomaleji.

Z tabulky lze vyčíst, že doba obarvování sítí nevykazuje na vstupních instancích výrazné rozdíly. Naproti tomu je vidět, že celková doba běhu algoritmu silně závisí na průměrném počtu hran obarvených stejnou barvou, což je nejvíce vidět na síti s nejnižší dobou běhu „genrmf_128_32“, na jejíž obarvení bylo použito pouze sedm barev a průměrně 221 184 hran bylo obarveno stejnou barvou. Pro sítě „BL06-camel-sml“ a „LB07-bunny-sml“, na jejichž obarvení

bylo použito nejvíce barev s průměrným počtem 5.66 respektive 15.77 hran obarvených stejnou barvou, algoritmus vůbec nedoběhl v rozumném čase.

název sítě	**kolikrát se *lattice* tvořila**	**prům. počet úrovní vrcholů v *lattice***
ac_4000	22	1.8
af_shell9	144	204.6
BL06-camel-sml	85	209.2
del_strip20	87	1 110.5
genrmf_128_32	—	—
grid_strip20	12	1 532
LB07-bunny-sml	28	82.0
rgg_strip20	110	648.4

Tabulka 5.2: Detaily struktury *lattice* během vykonávání algoritmu 3.7.10

název sítě	**použitý počet barev**	**prům. počet hran na barvu**	**doba barvení hran sítě**
ac_4000	6 547	1 221.62	6.23s
af_shell9	1 755	1 456.02	2.04s
BL06-camel-sml	654 271	5.66	8.70s
del_strip20	2 753	685.58	4.53s
genrmf_128_32	7	221 184.00	3.37s
grid_strip20	1 024	1 638.05	4.93s
LB07-bunny-sml	167 944	15.77	5.23s
rgg_strip20	1 575	2 627.57	5.59s

Tabulka 5.3: Detaily obarvování hran jednotlivých sítí v algoritmu 3.8.1

název sítě	**sekv. Goldbergův alg.** (s. 2.3)	**alg. z** [12] (s. 3.7.1.1)	**alg. z** [13] (s. 3.7.1.2)	**alg. z** [14] (s. 3.7.2.1)	**alg. z** [15] (s. 3.7.2.2)	**alg. s obarvenými hranami** (s. 3.8)
ac_4000	0.08	17.91	0.20	13.82	23.93	129.84
af_shell9	0.81	0.38	3.26	63.32	2 759.8	62.81
BL06-camel-sml	1.49	9.34	49.65	348.77	2 843.97	+3 600.00
del_strip20	1.27	4.64	65.64	229.31	+3 600.00	238.39
genrmf_128_32	9.02	17.74	135.48	902.722	+3 600.00	14.45
grid_strip20	1.80	7.07	125.70	156.02	1 388.32	361.04
LB07-bunny-sml	0.57	0.79	6.12	25.56	262.88	+3 600.00
rgg_strip20	1.75	2.79	33.91	265.98	+3 600.00	122.20

Tabulka 5.4: Přehled naměřených časů běhu v sekundách

Závěr

V práci byly uvedeny existující sekvenční algoritmy hledající maximální tok v síti spolu s nezbytným teoretickým základem. Dále byly představeny čtyři existující masivně paralelní varianty Goldbergova algoritmu pro architekturu CUDA spolu s uvedením nového masivně paralelního algoritmu pro stejnou architekturu využívajícího obarvení hran sítě. Popsán byl také existující paralelní BFS algoritmus využívaný pro heuristiku Globílního přepočítání výšek.

Všechny paralelní algoritmy byly implementovány v programovacím jazyku `C++` rozšířeném o CUDA pro psaní kódu vykonávaného na GPU. Implementované algoritmy byly měřeny a testovány na sadě různorodých instancí na fakultním serveru *STAR* spolu s existující implementací sekvenčního Goldbergova algoritmu. Měření ukázalo, že uvedené implementace paralelních algoritmů výrazně zaostávají za sekvenční verzí a také, že závislost efektivity algoritmu na typu vstupní sítě je pro jednotlivé paralelní algoritmy výrazně odlišná.

Do budoucna se nabízí prozkoumat, zda by jiné implementace uvedených paralelních algoritmů dosahovaly lepších výsledků. Mohlo by se například jednat o odlišný způsob vnitřní reprezentace sítě, jiný způsob práce s pamětí v kombinaci s jinou volbou parametrů algoritmů, jako je například délka jejich interních smyček.

Příloha A

Seznam použitých zkratek

BFS Breadth-first search (prohledávání do šířky)

CPU Central processing unit (centrální procesorová jednotka)

CUDA Compute unified device architecture

GPU Graphics processing unit (grafická procesorová jendotka)

SIMT Single instruction multiple threads

Příloha **B**

Obsah přiloženého flash disku

- readme.txt ...... popis obsahu tohoto flash disku
- DP_Cerveny_Kamil_2021.pdf ...... tato práce ve formátu PDF
- src/ ...... adresář se zdrojovými kódy
  - thesis/ ...... adresář se zdrojovým kódem této práce
  - implementation/ ...... adresář se zdrojovým kódem implementace

Literatura

[1] M. Mareš, T. Valla. *Průvodce labyrintem algoritmů.* Praha: CZ.NIC, z. s. p. o., 2017. s: 319–346. ISBN: 978-80-88168-22-5.

[2] O. Suchý, T. Valla. *Sítě, toky v sítích, Ford-Fulkersonův algoritmus* [handout k přednášce]. Praha: České vysoké učení technické v Praze, Fakulta informačních technoglogií, Katedra teoretické informatiky. 2019.

[3] L. R. Ford, D. R. Fulkerson. *Maximal Flow Through a Network.* Canadian Journal of Mathematics. 1956, roč. 8, s: 399–404. DOI: 10.4153/CJM-1956-045-5.

[4] J. Edmonds, R. M. Karp. *Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems.* Journal of the ACM. 1972, roč. 19, s: 248–264. DOI: 10.1145/321694.321699.

[5] Y. Dinitz. *Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation.* Soviet Math. Dokl. 1970, roč. 11, s: 1277–1280.

[6] A. V. Goldberg, R. Tarjan. *A New Approach to the Maximum-Flow Problem.* J. ACM. 1988, roč. 35, s: 921–940. DOI: 10.1145/48014.61051.

[7] R. K. Ahuja, J. B. Orlin. *A Fast and Simple Algorithm for the Maximum Flow Problem.* Operations Research. 1989, roč. 37, s: 748-759. DOI: 10.1287/opre.37.5.748.

[8] B. V. Cherkassky, A. V. Goldberg. *On Implementing the Push—Relabel Method for the Maximum Flow Problem.* Algorithmica. 1997, roč. 19, s: 390–410. DOI: 10.1007/PL00009180.

[9] NVIDIA Corporation. *CUDA C++ Programming Guide* [online]. © 2007-2020 [cit. 03.07.2020]. verze: v11.0. Dostupné z: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[10] S. Hong, S. K. Kim, T. Oguntebi, K. Olukotun. *Accelerating CUDA Graph Algorithms at Maximum Warp*. Computer Systems Laboratory, Stanford University. 2011. DOI: 10.1145/2038037.1941590.

[11] M. Springer. *Breadth-first Search in CUDA*. Tokyo Institute of Technology. 6.11.2017. Dostupné z: `https://m-sp.org/downloads/titech_bfs_cuda.pdf`.

[12] J. Wu, Z. He, B. Hong. *Efficient CUDA Algorithms for the Maximum Network Flow Problem*. In *GPU Computing Gems Jade Edition*. 2012, s: 55-66. DOI: 10.1016/B978-0-12-385963-1.00005-8.

[13] J. Kolomazník, J. Horáček, V. Krajíček, J. Pelikán. *Implementing Interactive 3D Segmentation on CUDA Using Graph-Cuts and Watershed Transformation*. The 20th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision in cooperation with EUROGRAPHICS. Plzeň. 2012, s: 35-38. ISBN: 978-80-86943-80-0.

[14] V. Vineet, P. J. Narayanan. *CUDA Cuts: Fast Graph Cuts on the GPU*. 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops. 2008. DOI: 10.1109/CVPRW.2008.4563095.

[15] M. Hussein, A. Varshney, L. Davis. *On implementing graph cuts on cuda*. In *First Workshop on General Purpose Processing on Graphics Processing Units*. Northeastern University. 2007.

[16] J. Misra, D. Gries. *A constructive proof of Vizing's theorem*. Information Processing Letters. 1992, roč. 41, s: 131-133. DOI: 10.1016/0020-0190(92)90041-S.

[17] J. Groschaft. *Paralelizace algoritmů pro toky v síti*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

[18] Z. He, B. Hong. *Dynamically Tuned Push-Relabel Algorithm for the Maximum Flow Problem on CPU-GPU-Hybrid Platforms*. 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). 2010. DOI: 10.1109/IPDPS.2010.5470401.

[19] P. Tvrdík. *Úvod do paralelního a distribuovaného programování* [handout k přednášce]. Praha: České vysoké učení technické v Praze, Fakulta informačních technoglogií, Katedra počítačových systémů. 2020.

[20] NVIDIA Corporation. *Inside Volta: The World's Most Advanced Data Center GPU* [online] [cit. 15.07.2020]. 2017. Dostupné z: `https://developer.nvidia.com/blog/inside-volta`.

[21] J. Groschaft. *Maxflow* [online] [cit. 20.07.2020]. GitHub repository. 2019. Dostupné z: https://github.com/Zagrosss/maxflow.

[22] D. Tödling, M. Winter. *Breadth-First Search on Dynamic Graphs using Dynamic Parallelism on the GPU*. Proceedings of CESCG 2019: The 23rd Central European Seminar on Computer Graphics (non-peer-reviewed). 2019. Dostupné z: https://cescg.org/cescg_submission/breadth-first-search-using-dynamic-parallelism-on-the-gpu.

[23] P. Richard. *C++ Program to Perform Edge Coloring of a Graph* [online] [cit. 25.01.2021]. 2019. Dostupné z: https://www.tutorialspoint.com/cplusplus-program-to-perform-edge-coloring-of-a-graph.

[24] Boost. *Boost C++ Libraries* [online] [cit. 26.01.2021]. 2021. Dostupné z: https://www.boost.org.

[25] Thrust. *Thrust Thrust - Parallel Algorithms Library* [online] [cit. 26.01.2021]. Dostupné z: https://thrust.github.io.

[26] The Cornell University Center for Advanced Computing. *Introduction to GPGPU and CUDA Programming: Memory Coalescing* [online] [cit. 31.01.2021]. Dostupné z: https://cvw.cac.cornell.edu/gpu/coalesced.

[27] NVIDIA Corporation. *Adaptive Parallel Computation with CUDA Dynamic Parallelism* [online] [cit. 01.02.2021]. Dostupné z https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism.

[28] Center for Discrete Mathematics and Theoretical Computer Science. *The First DIMACS Implementation Challenge: 1990-1991* [online] [cit. 02.02.2021]. Dostupné z http://archive.dimacs.rutgers.edu/pub/netflow/generators/network.

[29] Ch. Schulz. *KaHIP - Karlsruhe High Quality Partitioning* [online] [cit. 02.02.2021]. Dostupné z https://kahip.github.io.

[30] Cheriton School of Computer Science. *Max-flow problem instances in vision* [online] [cit. 02.02.2021]. Dostupné z https://vision.cs.uwaterloo.ca/data/maxflow.

[31] D. S. Johnson, C. C. McGeoch. *Network Flows and Matching: First DIMACS Implementation Challenge*. Boston, MA, USA: American Mathematical Society, 1993. ISBN: 0821865986.

[32] D. Bader, H.Meyerhenke, P. Sanders, D. Wagner. *Graph partitioning and graph clustering. Proceedings of the 10th DIMACS implementation*

challenge workshop. Atlanta, GA, USA, February 13–14, 2012. 2013. Dostupné z DOI: 10.1090/conm/588.