VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE BURROWS-WHEELEROVY TRANSFORMACE S VYUŽITÍM GPU

ACCELERATION OF BURROWS-WHEELER TRANSFORM USING GPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — TOMÁŠ ZAHRADNÍČEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. VÁCLAV ŠIMEK
SUPERVISOR

BRNO 2019

Ústav počítačových systémů (UPSY) Akademický rok 2018/2019

Zadání diplomové práce

21785

Student: **Zahradníček Tomáš, Bc.**
Program: Informační technologie Obor: Počítačové a vestavěné systémy
Název: **Akcelerace Burrows-Wheelerovy transformace s využitím GPU**
Acceleration of Burrows-Wheeler Transform Using GPU
Kategorie: Algoritmy a datové struktury
Zadání:

1. Pečlivě prostudujte rysy architektury a principy činnosti moderních grafických karet (GPU).
2. Prostudujte softwarové knihovny CUDA a OpenCL pro implementaci obecných výpočtů na platformě GPU.
3. Prostudujte Burrows-Wheelerovu transformaci (BWT) a její využití pro kompresi dat.
4. Proveďte detailní rozbor sekvenční varianty této komprimační metody a identifikujte časově nejnáročnější části.
5. Na základě zjištěných poznatků navrhněte řešení využívající platformu GPU k akceleraci časově náročných částí BWT.
6. Implementujte navržené řešení v jazyce C/C++ s využitím knihoven CUDA či OpenCL. Proveďte detailní analýzu výkonnosti s pomocí dostupných nástrojů.
7. Zhodnoťte dosažené výsledky a diskutujte možná vylepšení projektu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 5 zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Šimek Václav, Ing.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 31. října 2018

Abstrakt

Tato práce se zabývá Burrows-Wheelerovou transformací (BWT) a možnostmi akcelerace této transformace na grafickém procesoru (GPU). Jsou představeny metody komprese založené na BWT. Pro práci s GPU jsou představeny softwarové knihovny CUDA a OpenCL. Jsou implementovány paralelní varianty BWT i následných kroků potřebných ke kompresi, s použitím knihovny CUDA. Je testována míra komprese použitých přístupů a paralelní verze jsou porovnány s jejich sekvenčními implementacemi.

Abstract

This thesis deals with Burrows-Wheeler transform (BWT) and possibilities of acceleration of this transform on graphics processing unit (GPU). Methods of compression based on BWT are introduced, as well as software libraries CUDA and OpenCL for writing programs for GPU. Parallel variants of BWT are implemented, as well as following steps necessary for compression, using CUDA library. Amount of compression of used approaches are tested and parallel versions are compared to their sequential counterparts.

Klíčová slova

Burrows-Wheelerova transformace, BWT, GPU, CUDA, komprese, akcelerace, paralelní algoritmy

Keywords

Burrows-Wheeler transform, BWT, GPU, CUDA, compression, acceleration, parallel algorithms

Citace

ZAHRADNÍČEK, Tomáš. *Akcelerace Burrows-Wheelerovy transformace s využitím GPU.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Šimek Václav.

Akcelerace Burrows-Wheelerovy transformace s využitím GPU

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Václava Šimka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Tomáš Zahradníček
21. května 2019

Obsah

1 Úvod **3**

2 Burrows-Wheelerova transformace **4**
2.1 Podstata transformace 4
2.2 Efektivní implementace 6
2.2.1 Dopředná BWT 6
2.2.2 Zpětná BWT 6

3 Komprese pomocí BWT **9**
3.1 Další transformace 9
3.1.1 Move-to-front 9
3.1.2 Run-length encoding 9
3.2 Entropické kodéry 10
3.2.1 Huffmanovo kódování 10
3.2.2 Aritmetické kódování 12

4 Grafické procesory **13**
4.1 CUDA 13
4.1.1 Aplikace kernelu 14
4.1.2 Výpočetní model 14
4.1.3 Paměťový model 15
4.1.4 Paralelní metody 16
4.2 OpenCL 16
4.2.1 Aplikace kernelu 16
4.2.2 Model 17

5 Paralelní operace **19**
5.1 Scan 19
5.2 Segmentovaný scan 20
5.3 Suffix sum 21

6 Návrh akcelerace BWT pomocí GPU **23**
6.1 Dopředná transformace 23
6.2 Zpětná transformace 23

7 Implementace **25**
7.1 dopředná BWT 25
7.1.1 thrust varianta 25

7.1.2 Merge-splitting varianta . . . 26
7.1.3 Bucket sort varianta . . . 26
7.1.4 Řazení po částech . . . 27
7.1.5 Obecné optimalizace . . . 29
7.2 Zpětná BWT . . . 30
7.3 Move-to-front . . . 31
7.4 Huffmanovo kódování . . . 31
7.5 Komprimovaný formát . . . 33

8 Testování **34**
8.1 Dopředná BWT . . . 34
8.1.1 Základní implementace . . . 34
8.1.2 Agregované porovnávání . . . 36
8.2 Zpětná BWT . . . 36
8.3 Komprese . . . 37
8.3.1 Náročnost jednotlivých částí . . . 37
8.3.2 Kompresní poměr . . . 37

9 Závěr **39**

Literatura **40**

A Obsah přiloženého paměťového média **41**

Kapitola 1

Úvod

Jedním z využití Burrows-Wheelerovy transformace (BWT) je komprese dat. Při kompresi a dekompresi větších souborů na sekvenčním CPU může i tato relativně rychlá metoda zabrat výrazné množství času. Cílem této práce je identifikovat časově náročné části BWT a vytvořit akcelerovanou implementaci pro grafické procesory (GPU).

První část práce se zabývá Burrows-Wheelerovou transformací. Je představen samotný koncept transformace a proč je s kombinací s ostatními transformacemi a kompresními metodami tak účinná. Následně je podrobně popsána implementace BWT a její inverzní varianty (zpětná BWT). V kapitole 3 jsou zhruba představeny nejpoužívanější kompresní metody s kombinací s BWT. Kapitola 4 se zabývá architekturou GPU a možnostmi vytváření programů pro běh na nich – knihovny CUDA a OpenCL.

Kapitola 5 popisuje velmi časté a efektivní paralelní operace, které je vhodné použít i na paralelní implementaci BWT.

V kapitole 7 je popsána implementace několika variant sekvenční a paralelní BWT, využívající různých druhů řazení a způsobu paralelizací na různých úrovních. Také jsou zmíněny použité obecné optimalizace, které mohou BWT zrychlit na úkor větší paměťové náročnosti. Ačkoliv je BWT hlavním tématem, jsou implementovány sekvenční i paralelní varianty dalších potřebných transformací, díky kterým lze dohromady provádět kompresi dat.

Tyto různé implementace jsou navzájem porovnávány v kapitole 8. Podrobněji jsou porovnávané samotné varianty BWT bez dalších částí pro kompresi. Následně jsou také porovnávány jednotlivé části komprese, jak z hlediska časové náročnosti tak dosažené komprese.

Kapitola 2

Burrows-Wheelerova transformace

V této kapitole bude představena základní myšlenka Burrows-Wheelerovy transformace, abychom pochopili její užitečné vlastnosti pro kompresi dat. Následuje podrobný popis implementace a možných modifikaci, a analýza časově nejnáročnějších částí algoritmu.[2]

2.1 Podstata transformace

Samotná Buttows-Wheelerova transformace (BWT) nezmění velikost vstupních dat. Dokáže ale přeuspořádat data takovým způsobem, aby značně zvýšila kompresní poměr následujících kompresních metod (popsáno v kapitole 3). BWT dokáže pracovat nad libovolnými daty (nejen textovými), ale příklady budou uvažovat právě textový vstup.

Uvažujme text "mississippi$", jako vstup pro BWT. Znak $ je pomocný symbol pro reprezentaci konce řetězce, a lexikograficky bude uvažován jako poslední znak abecedy. Pro BWT je potřeba první vytvořit všechny rotace vstupního řetězce (obr. 2.1a). Následně jsou všechny tyto rotace lexikograficky seřazeny (obr. 2.1b). Výsledkem BWT je řetězec z posledního sloupce seřazených rotací, tady v tomto případě "ssmp$pissiii".

```
mississippi$
ississippi$m
ssissippi$mi
sissippi$mis
issippi$miss
ssippi$missi
sippi$missis
ippi$mississ
ppi$mississi
pi$mississip
i$mississipp
$mississippi
```

(a)

```
ippi$mississ
issippi$miss
ississippi$m
i$mississipp
mississippi$
pi$mississip
ppi$mississi
sippi$missis
sissippi$mis
ssippi$missi
ssissippi$mi
$mississippi
```

(b)

Obrázek 2.1: (a) Všechny rotace vstupního řetězce "mississippi". (b) Seřazené rotace.

V tomto jednoduchém případě není užitečnost transformace příliš zřejmá. Pro to je potřeba transformovat co možná nejdelší řetězce (blok). U delších řetězců se projeví shlukování

symbolů s podobným kontextem (podobné následující znaky). Toto je zapříčíněno tím, že znaky z posledního sloupce jsou následovány podřetězci začínajícími v prvním sloupci, a právě tyto jsou seřazeny, a tedy stejně začínající podřetězce jsou seskupeny k sobě.

K tomu aby byla transformace prakticky použitelná musí být možné její výstup transformovat zpět na původní data. Na začátku je znám pouze výstup BWT, tedy poslední sloupec(obr. 2.2a). Výstup BWT je pouze přeuspořádání vstupu, jsou tedy zachovány všechny symboly (a jejich počty), což jsou všechny podřetězce původního textu o délce 1. Protože v prvním sloupeci jsou tyto symboly pouze seřazené, je možné z výstupu BWT (posledního sloupce) získat sloupec první (obr. 2.2b). Jakmile je získán první, jsou známy všechny podřetězce o délce 2, protože symboly z prvního sloupce následují bezprostředně za symboly z posledního sloupce (obr. 2.2c). Tyto podřetězce je možné opět seřadit (obr. 2.2d), a získat tak opět další sloupec. Takto je možné pokračovat dokud není zrekonstruována celá tabulka 2.1b. Z této lze odvodit původní text, například z řádku ve kterém je $ posledním symbolem.

s
s
m
p
$
p
i
s
s
i
i
i

(a)

i	s
i	s
i	m
i	p
m	$
p	p
p	i
s	s
s	s
s	i
s	i
$	i

(b)

s	i	s
s	i	s
m	i	m
p	i	p
$	m	$
p	p	p
i	p	i
s	s	s
s	s	s
i	s	i
i	s	i
i	$	i

(c)

ip	s
is	s
is	m
i$	p
mi	$
pi	p
pp	i
si	s
si	s
ss	i
ss	i
$m	i

(d)

Obrázek 2.2: Podstata zpětné BWT: (a) Vstup zpětné BWT je použit jako poslední sloupec. (b) Seřazení symbolů do prvního sloupce. (c) Nově známé podřetězce délky 2. (d) Seřazení podřetězců délky 2.

2.2 Efektivní implementace

Postup popsaný v předchozí části je vhodný na pochopení principu BWT, ale pro efektivní implementaci nevhodný. Při bloku velikosti N by bylo potřeba N^2 paměti pro všechny rotace, a minimálně tedy N^2 operací – na pouhé zapsání každé hodnoty v tabulce. Navíc pro zpětnou BWT bylo potřeba provést N seřazení řetězců o délce až N. Protože užitečnost BWT se značně zvyšuje se zvyšováním délky bloku (N), prakticky jsou používány efektivnější metody s prostorovou složitostí $O(n)$ a časovou složitostí mezi $O(n)$ a $O(n\ log\ n)$ (podle zvolených možných modifikací a také vlastnostech vstupního textu).

2.2.1 Dopředná BWT

Pro dopřednou BWT, namísto ukládání každého rotovaného řetězce, stačí pomocné pole $R[1..N]$, které bude obsahovat, kolikátým symbolem původního textu začíná odpovídající řádek tabulky (obr. 2.3). Není zde použit pomocný symbol $, což znamená, že do výstupu BWT musí být přidáno pořadí symbolu ve výstupu, kterým původní text začínal. Oproti tomu není kladeno omezení na vstupní data – s unikátním symbolem $ nesmí být tento symbol obsažen v původním textu. Vstupní řetězec je označen T (v tomto případě T = "mississippi"). Pro vytvoření tabulky pomocí pole R musí být pole inicializováno tak, aby obsahovalo všechny indexy (např. $R[i] = i$) a následně jej seřadit – Pro řazení prvku i jsou vždy použity podřetězce začínající $T[R[i]]$. Vzhledem k tomu, že výstupem BWT je poslední sloupec (označen L), lze jej získat jako $L[i] = T[R[i] - 1]$. Výraz $R[i] - 1$ v tomto případě musí nabývat hodnot 1..N, tedy cyklicky $i - 1 = N$ pro $i = 1$.

R	T	
1	m	mississippi
2	i	ississippim
3	s	ssissippimi
4	s	sissippimis
5	i	issippimiss
6	s	ssippimissi
7	s	sippimissip
8	i	ippimissipp
9	p	ppimississi
10	p	pimississip
11	i	imississipp

(a)

R	T	F L
11	m	imississipp
8	i	ippimississ
5	s	issippimiss
2	s	ississippim
1	i	mississippi
10	s	pimississip
9	s	ppimississi
7	i	sippimissis
4	p	sissippimis
6	p	ssippimissi
3	i	ssissippimi

(b)

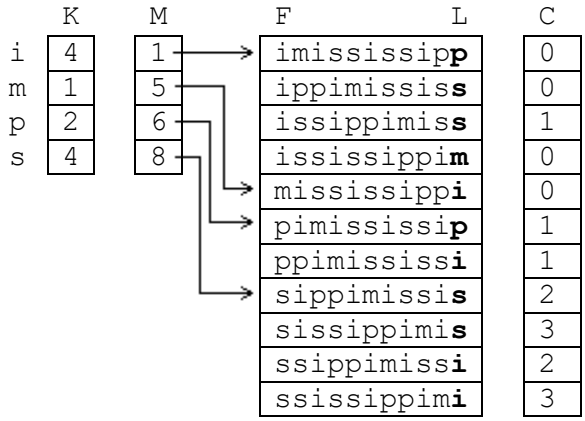
Obrázek 2.3: Pomocné pole R, vstup T = "mississippi"a implicitní reprezentace tabulky (a) Po inicializaci. (b) Po seřazení

2.2.2 Zpětná BWT

Zpětná BWT potřebuje některé další pomocné struktury, ale je možné ji provést v prostorové i časové složitosti $O(N)$. Pro rekonstrukci textu je potřeba u každého znaku z F (prvního sloupce) zjistit jeho pozici v L (posledním sloupci), a tím i následující znak, který je nyní na tomto novém řádku ve sloupci F. Vzhledem k tomu že znaky se mohou opakovat, je nutné různé výskyty stejného znaku rozlišovat. Znaky v L jsou uspořádány podle

následujících znaků a stejné znaky v F, tedy podřetězec jimi začínající sdílí alespoň první znak, jsou seřazeny také podle následujících znaků. Díky tomu jsou stejné znaky v L mezi sebou uspořádány ve stejném pořadí jako stejné znaky v F (např. první výskyt znaku p ve sloupci L i jeho první výskyt v F budou ty samé výskyty tohoto znaku v kontextu vstupního řetězce).

Při zpětné BWT musí být známa pozice v L posledního znaku vstupního řetězce T. Je vhodnější rekonstruovat původní text pozpátku, při využití poznatku, kterým lze jednoduše spárovat znaky z L a F. Ke znaku který je právě dekódován je nalezen výskyt toho znaku v F, a od něj odvozen předcházející znak (v tomto novém řádku ve sloupci L). Od nově zjištěného předcházejícího znaku se pokračuje stejným způsobem. Pro tento postup jsou zavedeny nové pomocné struktury (obr. 2.4): pole $K[c]$ (c jsou postupně lexikograficky všechny znaky vstupní abecedy), pro počítání kolikrát se daný znak již vyskytl při procházení sloupce F, pole $C[1..N]$ pro přiřazení pořadí výskytu daného symbolu ve sloupci L, a $M[c]$ pro uložení indexu prvního výskytu každého symbolu ve sloupci F.



Obrázek 2.4: Implicitní sestavení správného pořadí rotovaných bloků pomocnými strukturami.

Algoritmus 1 popisuje získání dekódovaného textu Q. Vstupem algoritmu je sloupec L a index a, který označuje index symbolu v L, který je posledním symbolem v původním řetězci T. Po inicializaci pole K je postupně procházen vstup zpětné BWT, tedy L, a ke každému výskytu symbolu přiřazován počet výskytů daného symbolu před tímto. Potom jsou (na řádku 11) díky spočítaným výskytům každého znaku odvozeny první výskyty ve sloupci F. Sloupec F nemusí být ani explicitně konstruován, protože je implicitně obsažen v poli M (stejné znaky jsou zde díky seřazení vždy v jednom shluku). Nakonec je s těmito pomocnými strukturami odzadu naplněn výsledek Q. Zde je (na řádku 16) využito zmíněné získávání odpovídajícího předešlého znaku pomocí párování znaků v F a L.

Algorithm 1 rekonstrukce původního textu [2]

BWT-Decode(L, a)

```
for c ← 1 to |Σ| do
    K[c] ← 0

for i ← 1 to N do
    C[i] ← K[L[i]]
    K[L[i]] ← K[L[i]] + 1

sum ← 1
for c ← 1 to |Σ| do
    M[c] ← sum
    sum ← sum + K[c]

i ← a
for j ← n downto 1 do
    Q[j] ← L[i]
    i ← C[i] + M[L[i]]
```

Kapitola 3

Komprese pomocí BWT

Samotná BWT nesníží množství zakódovaných dat, ale přeuspořádá symboly tak, že stejné symboly se vyskytují velmi často ve shlucích (ne nezbytně bez přerušení). Proto se BWT používá jako první z několika kroků komprese. Navazující kroky jsou jednoduché transformace, které dokáží dále zjednodušit strukturu komprimovaného textu (kapitola 3.1), a nakonec se takto předzpracovaný text komprimuje entropickými kodéry, které se snaží každý symbol reprezentovat co nejméně bity (kapitola 3.2).

3.1 Další transformace

Ačkoliv existují komplikovanější metody pro zpracování textu pro kompresi, jednoduché transformace move-to-front nebo run-length encoding použity na výstup BWT nabízí jedny z nejlepších kompresních poměrů, a zároveň jsou oproti ostatním metodám rychlé.

3.1.1 Move-to-front

Move-to-front list (MTF) je obvyklá metoda spojována s BWT, navrhnuta již při první publikaci jejími autory [3]. Výstupem této transformace jsou namísto znaků v textu čísla, které reprezentují kolik unikátních odlišných znaků se v textu vyskytlo mezi stávajícím znakem a nejbližším předchozím výskytem stejného znaku. Pokud tedy výstup BWT obsahuje převážně znaky ve shluku, většina výsledných čísel budou nuly (za sebou následující výskyty stejného znaku) s občasnými vyššími čísly při výskytu nového znaku.

Při zpracování je udržován seznam všech symbolů ve vstupním textu. Text je postupně procházen a pro každý symbol je výstupem jeho stávající index v seznamu. Takto zpracovaný symbol je následně přesunut na začátek seznamu, čímž se sníží indexy dalších výskytů toho samého symbolu. Výstupní čísla mohou být dále kódovány fixním kódem, který potřebuje méně bitů pro nižší hodnoty, nebo ponechán pro entropické kodéry (kapitola 3.2).

3.1.2 Run-length encoding

Run-length encoding (RLE) má také velký přínos z předcházející BWT, jejíž výstup obsahuje velké množství sledů stejného symbolu. RLE kóduje takové sledy sekvencí *specialní symbol, počet opakování, opakovaný symbol.* Krátké sledy se nemusí kódovat vůbec - jsou ponechány stejné jako na vstupu. Právě v tom případě musí zakódovaný sled začínat symbolem, označující že právě RLE kódování následuje.

3.2 Entropické kodéry

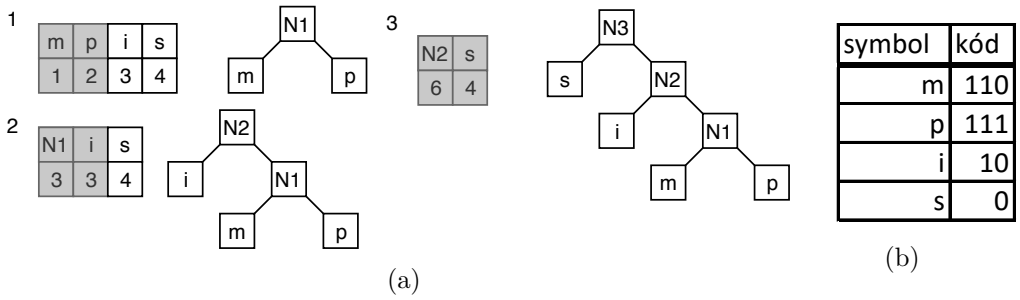
Entropické kodéry vychází ze statistické pravděpodobnosti výskytu symbolů pro jejich zakódování. Symboly s vyšší pravděpodobností jsou zakódovány na méně bitů a naopak. Jednou možností pro odvození pravděpodobnost je z počtu výskytů jednotlivých znaků v kódovaném souboru, Na což jsou potřeba dva průchody vstupním textem - pro spočítání pravděpodobností a pro samotné kódování. Tímto přístupem bude mít pro daný vstup jeden symbol po celou dobu stejný kód. Další možností je adaptivní kódování, které odvozování pravděpodobností a kódování provádí současně. Pravděpodobnosti se mohou postupně měnit, a tedy se mohou měnit i kódy přiřazené jednotlivým symbolům.

Obě tyto techniky lze využít pro různý způsob kódování. Uvedeny budou Huffmanovo kódování a aritmetické kódování.

Poměr výskytů symbolů ve výstupu BWT se v různých částech značně mění, je vhodné aby odvozené pravděpodobnosti a tedy i kódy symbolů vycházeli jen z relativně nedávno zpracovaných symbolů. Proto se v souvislosti s BWT používají spíše varianty adaptivních kódů. Při využití neadaptivních variant by kódování výstupu BWT nemělo žádný přínos oproti kódování přímo vstupního textu. Pokud ale mezi BWT a kódováním je použita například metoda move-to-front, která vhodně využije shluků stejných znaků, je možné u neadaptivních variant dosáhnout lepší komprese.

3.2.1 Huffmanovo kódování

Při Huffmanovém kódování je ze získaných četností kódovaných symbolů zkonstruován Huffmanův strom, kde každý symbol je reprezentován listovým uzlem, a každá hrana reprezentuje určitý bit pro výsledný kód symbolu. Strom je konstruován seřazením symbolů do fronty tak, aby symbol s nejmenší četností byl ve frontě první. Postupně jsou odebírány z fronty vždy 2 prvky (uzly), a ty jsou spojeny pomocí nového rodičovského uzlu. Tento nový uzel je ohodnocen součtem četností jeho potomků, a je zařazen zpět do fronty podle nové četnosti. Takto se pokračuje dokud je ve frontě dostatek prvků. Ve výsledném stromu je možné zjistit kód pro každý symbol jako cestu z kořenového uzlu do daného listu (obr. 3.1). [8]



Obrázek 3.1: (a) postupná konstrukce huffmanova stromu na zakódování řetězce "mississippi"s pomocí fronty podle četnosti uzlů (b) kódy odvozené z vytvořeného stromu, za předpokladu že hrana doleva značí bit 0 a doprava 1

Listové uzly náležícím frekventovaným symbolům jsou umístěny blízko kořene, a tedy jim náleží kratší kód. Pro některé málo frekventované symboly může být výsledný kód delší než jejich původní reprezentace – pokud je v procesu vytváření stromu nový uzel umístěn do fronty opět jako první, může délka kódu v bitech růst až na celkový počet symbolů.

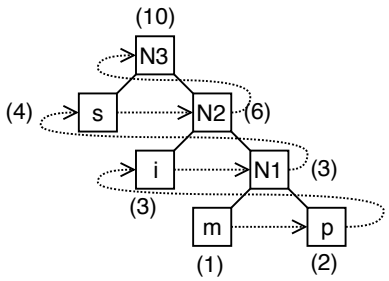
Huffmanův kód se dekóduje postupným čtením bitů a posouváním v huffmanově stromu. Počínaje v kořenovém uzlu je vždy vybrán jeden z potomků, podle přečteného bitu, dokud není dosaženo listového uzlu. Tehdy je ukončeno dekódování jednoho symbolu, náležícímu danému listovému uzlu, a pokračuje se opět od kořene. Není tedy možné jednoduše začít dekódování uprostřed zakódovaného souboru, protože by nebylo možné zjistit stávající uzel v procesu dekódování.

Protože není možné ze zakódovaného souboru implicitně zjistit frekvence symbolů ze kterého by bylo možné vytvořit Huffmanův strom, je potřeba tento strom explicitně přidat k zakódovanému řetězci – není potřeba přidávat ke každému symbolu přesnou frekvenci, postačí propojení uzlů ve stromu, a reprezentace symbolů pro každý listový uzel.

Při adaptivním Huffmanově kódování není strom vytvořen předem podle frekvencí symbolů, ale je postupně vytvářen a upravován při čtení symbolů k zakódování. Na začátku je strom prázdný a žádnému symbolu není přiřazen kód. Při prvním výskytu každého symbolu je tento symbol zakódován nezměněný, a odpovídající listový uzel je přidaný do Huffmanova stromu. Při každém dalším výskytu je symbol zakódován podle stávajícího stromu (cesta od kořene k symbolu), a četnost symbolu je zvýšena. Tato úprava stromu může mít za následek porušení podmínky stromu, že uzly na nižší úrovni (dále od kořene) musí mít nižší četnost než uzly ve vyšší úrovni. Při každém zakódovaném symbolu musí být tato podmínka zkontrolována, a pokud je porušena, jsou uzly ve stromě přeuspořádány tak aby opět platila. Tímto mohou být jednomu symbolu postupně přiřazovány různé kódy.

Pro jednodušší kontrolování pravidla, že na nižších úrovních jsou vždy uzly s nižší četností je zavedeno striktnější pravidlo: v průchodu uzly zleva doprava a zdola nahoru (ke kořeni)(obr. 3.2) musí být četnosti uzlů seřazené vzestupně. Při každé úpravě četnosti (kvůli právě zakódovanému/dekódovanému symbolu) je počínaje upravovaným uzlem (označen X) jsou provedeny následující kroky:

- porovnání X s četností s jeho následníkem (v průchodu zleva doprava, zdola nahoru). Pokud má následník stejnou nebo nižší četnost, jsou prohozeny uzel X a jeho nejvzdálenější následník se stejnou nebo nižší četností
- uzlu X je zvýšena četnost
- opakováno pro rodiče uzlu X, pokud rodičem není kořen



Obrázek 3.2: Uspořádání na huffmanově stromě. Tečkovaně je naznačena zmíněná posloupnost zleva doprava, zdola nahoru. U každého uzlu je v závorce uvedena četnost, pro nelistové uzly je to součet četností jeho 2 potomků.

Při dekódování jsou tyto kroky identické. Dekodér začíná z prázdným stromem a postupně podle dekódovaných symbolů strom upravuje. Musí ale rozeznat kdy očekává zakódovaný symbol, a kdy neupravený symbol (ještě neobsažený ve stromu). Pro tento účel

strom vždy obsahuje pomocný listový uzel s nulovou četností, který značí symbol který se ještě v souboru nevyskytl. Kód tohoto pomocného listového uzlu je vždy následován neupravenou reprezentací symbolu, který má být do stromu přidán.

Díky tomu, že stav stromu záleží pouze na již zakódovaných/dekódovaných symbolech, není potřeba strom explicitně kódovat jako u neadaptivní verze.

Četnosti jednotlivých uzlů mohou být periodicky snižovány (např. půleny), čímž je dávána větší váha právě zakódovaným symbolům ke změně stromu a tedy i délky kódů. Toto je vhodné pro kódování výstupu BWT, protože pravděpodobnost výskytu symbolů se v různých částech výrazně mění – stejné symboly jsou často shlukovány.

3.2.2 Aritmetické kódování

Aritmetické kódování obecně dosahuje lepšího kompresního poměru než Huffmanovo kódování, na úkor rychlosti. Lepší kompresní poměr je dosažen tím, že není přiřazován jeden kód jednotlivým symbolům. Celá zpráva zakódována jako binární číslo. Při kódování se přičítá kód (odpovídajícímu kódovanému symbolu) na stávající konec čísla. Oproti Huffmanovu kódování nejde však o pouhou konkatenaci kódů – při přičítání se mohou kódy překrývat a tak dojít k přenosu do vyšších řádu. Kódy pro jednotlivé symboly jsou opět kratší, čím je předpokládaná pravděpodobnost jeho výskytu větší. [7]

Stejně jako u Huffmanova kódování je možné, a kontextu BWT i vhodné aby kódy pro jednotlivé symboly nebyli stejné pro celý vstup, ale postupně se měnili na základě nedávno kódovaných symbolů. Odvozená pravděpodobnost je tedy opět nepřímo úměrná četnosti symbolu, přičemž tato počítaná četnost se periodicky snižuje pro snížení vlivu příliš vzdálených přečtených symbolů.

Kapitola 4

Grafické procesory

Grafické procesory GPU dokáží rychle provést výpočty při generování obrazu zadané scény. Nejčastějšími součástmi těchto výpočtů – grafického řetězce (pipeline) – je zpracování vrcholů, rasterizace polygonů a úprava výsledného rastru.

Starší GPU svou hardwarovou strukturou blízce odpovídají tomuto pořadí operací, které se při generování scény provádí: každému kroku odpovídá odlišná část GPU, která je speciálně konstruována na výpočet nad odpovídajícími daty (vertex shader, fragment shader).

Novější GPU stále podporují toto logické rozdělení, avšak hardwarově jsou shadery sjednoceny do jader které jsou schopny provádět libovolné operace (unifikované shadery). Protože při generování scény je potřeba zpracovat velké množsví dat (mnoho vrcholů a, ve výsledku, pixelů), obsahují GPU několik jader. Každé jádro má svůj řídící hardware, různá jádra tedy mohou vykonávat různé, na sobě nezávislé sekvence příkazů. Paralelizace se může provádět i na úrovní jednotlivých jader – každé jádro může obsahovat několik aritmetickologických jednotek a může tak provádět jednu operaci nad vektorem dat (SIMD – single instruction multiple data). Jádra (stream procesory / CUDA jádra) jsou dále hierarchicky seskupeny do multiprocesorů (NVIDIA - Stream multiprocessor), což umožňuje rychlou komunikaci mezi seskupenými jádry. Multiprocesory navíc obsahují vyrovnávací paměti pro data a instrukce, a logiku pro plánování úloh mezi jádry.

Další součástí GPU je paměť spolu s případnými cache pro rychlejší přístup k datům mezi jednotlivými jádry a pamětí. Pro zpracování dat na grafickém procesoru se musí data nahrát z hostitelského systému - CPU spolu s hlavní operační pamětí. Rychlost přenosu mezi hlavní paměti a paměti GPU je několikanásobně pomalejší oproti přenosům mezi GPU pamětí a grafickými jádry, proto je pro efektivitu programu důležité nepřenášet zbytečná data mezi CPU a GPU. GPU obecně nabízí vysokou propustnost na úkor zvýšené latence, je tedy vhodná pro zpracování různých na sobě nezávislých objektů.

4.1 CUDA

CUDA je rozšíření pro jazyky C/C++ současně s modelem pro paralelní výpočty, které umožňuje programování NVIDIA grafických procesorů. CUDA nabízí několik API, každá z různými úrovněmi abstrakce – Thrust API, runtime API a driver API. Thrust API je nejvyšším z nich – je možné jím programovat algoritmy nezávisle na specifickém hardwaru. Nižší API nabízí větší kontrolu nad hardwarem, ale vyžaduje podrobnější znalost i podrobnější výsledný kód.

4.1.1 Aplikace kernelu

Jedním z hlavních prvků CUDA programování je kernel. Jedná se o subrutinu, která je volána na hostitelském zařízení (CPU), a provedena na CUDA zařízení (GPU). Volání je asynchronní – vykonání kernelu je pouze naplánováno pro provedení na GPU.

Další abstrakci CUDA jsou vícerozměrné pole. V případě kdy má být každý prvek pole zpracován kernelem, může být na každý prvek vytvořeno odpovídající vlákno a díky tomu jednoduše namapováno na paralelní architekturu.

Rozšíření jazyků C/C++ spočívá kromě API také v nových klíčových slovech, většinou pro specifikaci paměti pro uložení daného objektu. Následuje ukázka zápisu kernelu, který přiřadí každému prvku v poli $a[i] = i$. Klíčové slovo `__global__` je použito pro definici kernelu (bude uložen v globální paměti GPU). Při paralelním zpracováním více prvků je tělo kernelu vykonáno na každém prvku. V jeho těle se využívají speciální proměnné `blockIdx`, `blockDim` a `threadIdx`, které specifikují vlastnosti bloku a vlákna v rámci tohoto bloku, a díky tomu lze přiřadit vláknu různé prvky na zpracování.

```
__global__ void fillKernel(int *a, int n)
{
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    if (tid < n) a[tid] = tid;
}
```

Výpis 4.1: Definice kernelu v CUDA.

Tento kernel je následně možné použít pro naplnění alokovaného pole (výpis 4.2). `thrust::device_vector` alokuje odpovídající pole na GPU. Při volání `fillKernel` ve funkci `fill` je uvnitř <<< >>> specifikována výpočetní konfigurace – počet bloků a počet vláken na blok. Pomocí jednoduchého dělení a operací modulo je odvozen minimální počet bloků s daným počtem vláken, aby bylo upraveno celé alokované pole. [4]

```
const int N=50000;
thrust::device_vector<int> a(N);
int nThreadsPerBlock= 512;
int nBlocks= n/nThreadsPerBlock + ((n%nThreadsPerBlock)?1:0);
fillKernel <<< nBlocks, nThreadsPerBlock >>> (thrust::
   raw_pointer_cast(&a[0]), n);
```

Výpis 4.2: Příklad volání kernelu v CUDA.

4.1.2 Výpočetní model

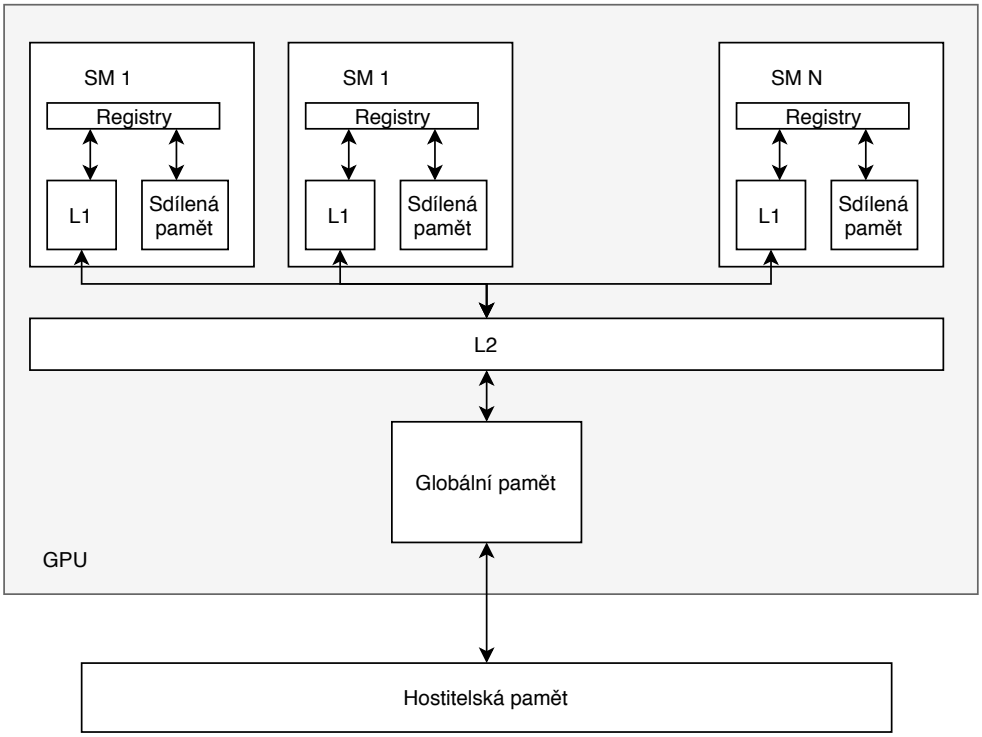
Hlavní myšlenkou CUDA v možnosti zvyšování výkonů algoritmů paralelizací je výpočetní model, který rozděluje výpočet na vlákna a bloky. GPU mají často za úlohu provádět opakovaně operaci (kernel) na velkém množství nezávislých dat. Každý prvek z těchto dat může být tedy zpracován v jednom vlákně. Blok obsahuje několik vláken. Celý blok je předán na zpracování jednomu multiprocesoru v GPU. Díky tomu můžou vlákna v jednom bloku spolupracovat. O rozdělování těchto úloh (bloků) multiprocesorům se stará plánovač na GPU. Každý multiprocesor si potom sám plánuje přiřazování vláken jednotlivým svým jádrům. Multiprocesory obsahují většinou 32 jader, mohou tedy obsluhovat v jednu chvíli 32 vláken, ale multiprocesoru by mělo být přiřazeno více vláken aby nebyl tak častý přístup k relativně pomalé paměti GPU. Multiprocesor má všechny potřebné data ke vláknům v jeho

bloku uložen ve svojí paměti, a dokud nedokončí celý blok, nemusí komunikovat se zbytkem GPU. Blok vláken rozdělí na tzv. warpy - dohromady 32 (počet jader v multiprocesoru) vláken, které se provádí současně.

Protože jednotlivá jádra v multiprocesoru vykonávají vždy stejné instrukce, může nastat problém při podmíněném větvení v kódu – část vláken v jednom warpu musí vykonat tělo `if` jiná část `else` – tzv. warp divergence. V tomto případě musí multiprocesor projít přes obě větve podmíněného příkazu a odpovídající příkazy provedou jen vlákna s odpovídajícím výsledkem podmínky. Toto může značně zpomalit výpočet (až 32krát, kde každé vlákno vykoná svoji větev). Proto je vhodné se takovým případům pokud možno vyhnout již při samotné formulaci paralelního algoritmu, nebo problematickou část algoritmu ponechat na výpočet pro CPU. [4]

4.1.3 Paměťový model

GPU podporující CUDA mají několik úrovní paměti (obr. 4.1) a nástroj CUDA k nim poskytuje programátorovi kontrolovaný přístup. Hlavní (globální) paměť GPU je největší (řádově GB) ale zároveň s nejpomalejším přístupem. Proto je při běhu algoritmů na GPU žádoucí znovupoužití dat, které už jsou jednou poslány na jednotlivé multiprocesory (SM - streaming multiprocessor). Nejvýkonnější (v propustnosti a latenci) pamětí jsou registry v jednotlivých SM, ale kapacitně velmi omezeny. Registry mohou být využívány pouze v rámci jednoho vlákna. Sdílená paměť, řádově desítky kB, umožňuje sdílení v rámci bloku vláken, tedy vláken běžícím na stejném SM. Je přímo na SM, nabízí tedy větší větší rychlost než globální paměť.



Obrázek 4.1: Hierarchie paměti CUDA zařízení.[4]

4.1.4 Paralelní metody

Map a reduce jsou základní metody pro modelování paralelního algoritmu pro jejich obecnost a jednoduchou paralelizaci.

Map značí aplikování funkce na každý prvek z kolekce. V rámci GPU je analogií jedno vlákno, které na daném vstupu (jednom prvku) aplikuje svůj kernel (funkci). Použití lze vidět ve výpisu 4.2, jako volání `fillKernel`.

Reduce dostane jako vstup celé pole prvků, na které je postupně aplikován binární asociativní operátor (funkce) a výsledkem je jedna hodnota. Typickými příklady *reduce* jsou suma, minimum nebo maximum (u číselných hodnot). Pro tyto typické příklady jsou v CUDA již připravené funkce pro celou redukci, případně `thrust::reduce` nabízí možnost uživatelsky definovaného binárního operátoru pro efektivní redukci.

Thrust API dále poskytuje řadu funkcí nad kolekcemi optimalizovanou pro paralelní výpočet. Relevantní pro BWT jsou například `scan`, což je metoda podobná reduce, ale místo jediného výsledku vrátí celé pole částečných výsledků. `sort_by_key` seřadí pole na základě pole klíčů, uložené odděleně od pole řazených hodnot. [1]

4.2 OpenCL

OpenCL je framework pro psaní paralelních algoritmů pro heterogení platformy. Oproti CUDA není mířena pouze na specifický typ (NVIDIA) GPU, ale GPU nezávisle na výrobci, případně FPGA nebo jiný specializovaný hardware. OpenCL poskytuje abstrakci nad různými vlastnostmi hardwaru.

OpenCL se dělí na následující části. *OpenCL platform API* obsahuje funkce pro zjištění možností připojených OpenCL zařízení, a nastavení kontextu pro aplikaci. *OpenCL runtime API* obsahuje funkce pro zadávání příkazů nad vytvořeným kontextem, tedy pomocí něj se zadává samotný paralelní výpočet na OpenCL zařízeních. Poslední částí, OpenCL programovací jazyk, je rozšíření jazyka C, který slouží pro psaní kernelů.

Paralelní operace jako např. metoda *reduce* nebo *scan* nejsou přímo v OpenCL podporovány, musí být tedy celé implementovány v uživatelském kódu pomocí základních prvků OpenCL – volání kernelu a případnými synchronizacemi mezi vlákny.

4.2.1 Aplikace kernelu

Kernel musí být definován jazykem OpenCL C v odděleném souboru od hlavního zdrojového kódu – ten je v neupraveném jazyce C, pouze jsou z něj volány funkce OpenCL. Kód pro kernel musí být kompilován při běhu hlavního programu (na začátku), pomocí následující dvojice funkcí (výpis 4.3) – první sestaví program ze zdrojového kódu OpenCL C (výpis 4.4), druhý vytvoří objekt pro kernel s daným názvem (jeden soubor/program může mít více definicí kernelů). Tento kernel, podobně jako v příkladu pro CUDA, naplní předané pole hodnotami $a[i] = i$. Při volání kernelu musí být parametry předem připraveny pomocí funkce `clSetKernelArg` 4.5.

```
program = CreateProgram(context, device, "kernel_source.cl")
kernel = clCreateKernel(program, "fillKernel")
```

Výpis 4.3: Příprava kernelu na začátku hlavního programu v OpenCL.

```
__kernel void fillKernel(__global int *a)
{
```

```
    int gid = get_global_id(0);
    a[tid] = tid;
}
```

Výpis 4.4: Zápis kódu pro kernel v OpenCL.

```
cl_mem memObject;
// ...
clSetKernelArg(kernel, 0, sizeof(cl_mem), memObject);
clEnqueueNDRangeKernel(commandQueue, kernel, /*...*/);
```

Výpis 4.5: Spuštění kernelu v OpenCL.

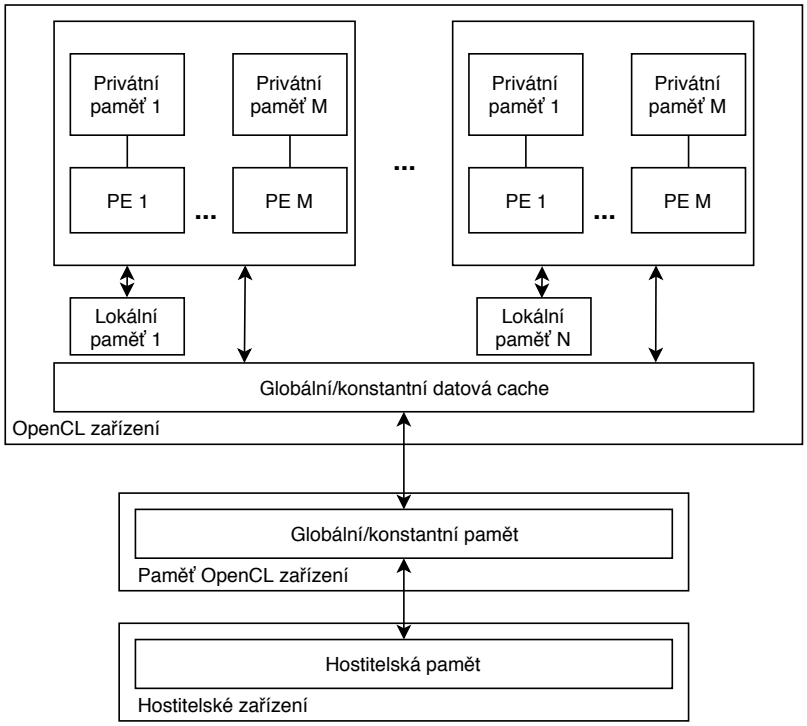
Při zadávání práce jednotlivým OpenCL zařízením (např. spuštění kernelu nad kolekcí dat) se manipuluje s frontou příkazů (command-queue) asociovanou s daným zařízením odpovídajícími funkcemi runtime API. Tato fronta ve výchozím nastavení nezačne vykonávat následující zadaný příkaz, dokud předchozí není úplně hotový. Toto může zpomalit rychlost výpočtu, pokud data těchto příkazů na sobě nezávisí. Návaznost jednotlivých příkazu lze zajistit pomocí událostí (event) – na konci příkazu nastane událost která spustí závislé příkazy. Tím lze docílit asynchronního provádění nezávislých příkazů.

Při spouštění kernelu (přesněji vložení tohoto spuštění do fronty, výpis 4.5) jsou zadávány parametry `globalWorkSize` a `localWorkSize`, které specifikují rozdělení dané úlohy mezi jednotlivé procesory na OpenCL zařízení. V CUDA jsou podobnými parametry počet bloků a vláken při spouštění kernelu. V OpenCL je však možné každé work-group přiřadit rozdílný počet work-item (zmíněné parametry jsou zadávány jako pole), a je tedy možné připravit takovou konfiguraci, že v kernelu není potřeba kontrolovat velikost pole (v porovnání s CUDA příkladem).

4.2.2 Model

V modelu OpenCL je vždy jedno hostitelské zařízení které voláním API OpenCL ovládá zbývající OpenCL zařízení. Každé OpenCL zařízení se dále dělí na jednu nebo více výpočetních jednotek (compute unit), a ty se dále dělí na jednu nebo více processing element (PE). U GPU mohou compute unit odpovídat multiprocesorům (SM) a processing element jednotlivým jádrům uvnitř SM. Při vykonávání kernelu jsou jednotlivé nezávislé prvky označovány work-item (podobně jako v CUDA thread), kde každý má svoje globální ID.

Této hierarchii odpovídá také několik úrovní pamětí(obr. 4.2). Globální paměť může být čtena/zapisována libovolným work-item, přičemž tento přístup může být podle možností zařízení cachován. Konstantní paměť je pouze pro čtení a musí být inicializována hostitelským zařízením. Lokální pamět náleží vždy jedné work-group (skupina work-item, podobně jako blok v CUDA), a může tedy obsahovat sdílené proměnné mezi více work-item. Privátní paměť náleží jedinému work-item.



Obrázek 4.2: Hierarchie paměti v OpenCL.[5]

Kapitola 5

Paralelní operace

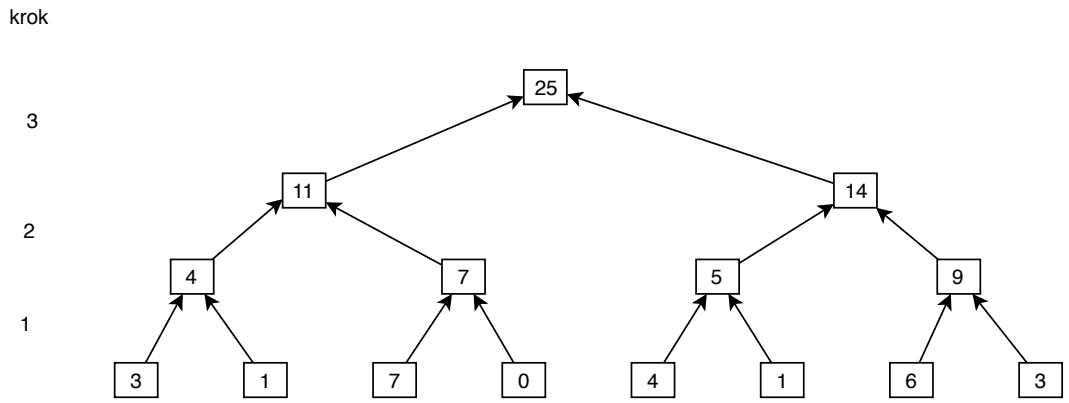
Pro paralelní algoritmy existují obecné paralelní operace aplikovatelné na různé problémy. V této kapitole budou uvedeny takové operace, které mohou být využity pro BWT nebo kompresní metody následující po BWT.

5.1 Scan

Operace scan (prefix sums) přijímá pole n prvků $[a_0, a_1, ..., a_n]$ a binární asociativní operátor $\oplus$. Výstupem operace je $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$. [6]

Velmi častými používanými asociativními operátory je součet (+), minimum a maximum. Tato operace je součástí mnoha paralelních algoritmů, například i paralelního řazení quick sort a radix sort.

V sekvenčním prostředí je tato operace jednoduše implementovatelná s časovou složitostí $O(n)$ přičítáním postupně každého prvku do průběžné sumy, která je uložena na odpovídající místo do výstupu. Pro paralelizaci je využita asociativita operátoru, díky kterému lze změnit pořadí aplikace na jednotlivé prvky ve smyslu binárního stromu.



Obrázek 5.1: Provádení plus-reduce na stromě

První částí této operace je jednodušší operace, reduce, jejíž výsledkem je pouze jedna hodnota, suma všech prvků $(a_0 \oplus a_1 \oplus ... \oplus a_{n-1})$. Pokud je pro každý pár prvků (tedy listové uzly odpovídajícího binárního stromu) přiřazen jeden procesor/vlákno a každý procesor může nezávisle přistoupit ke svým prvkům i zapisovat do paměti nezávisle (model PRAM), je možné tuto fázi provést v čase $O(log\ n)$, protože $\lceil log_2\ n \rceil$ je hloubka stromu, po

kterém se operace $\oplus$ provádí (obr. 5.1). Pro provádění této části není potřeba paměť navíc – mezivýsledky je možné ukládat přímo do vstupního pole vhodným indexováním (obr. 5.2).

krok	prvky v poli							
	3	1	7	0	4	1	6	3
1	3	4	7	7	4	5	6	9
2	3	4	7	11	4	5	6	14
3	3	4	7	11	4	5	6	25

Obrázek 5.2: Provádění plus-reduce v poli. Označeny jsou prvky které jsou v každém kroku modifikovány. Výsledek reduce je na konci v posledním prvku pole.

Pokud není k dispozici dostatek procesorů na každý prvek, ale jen p procesorů, kde $p < n$, může každému procesoru být přiřazeno n/p prvků, a každý procesor na své části spočítá svou sumu sekvenčně, tedy s časovou složitostí $O(n/p)$. Složitost operace reduce je tedy potom $O(n/p + log\ p)$.

Druhá část operace scan využívá částečných součtů, které byli vytvořeny v průběhu reduce. Je využita stejná stromová struktura, ale tentokrát se začíná od kořene. Do kořenového uzlu je vložen nulový prvek asociativního operátoru, a následně v každém kroku aktivní uzel uloží do svého levého potomka svou hodnotu, a do pravého potomka součet (operátorem $\oplus$) své hodnoty a hodnoty levého potomka (před přiřazením) (obr. 5.3).

Po provedení všech kroků až po listové uzly obsahuje každý prvek součet ($\oplus$) všech předchozích prvků ve vstupním poli.

Výsledkem je prescan – operace se vstupem stejným jako scan a výstupem $[0_\oplus, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$, kde $0_\oplus$ je nulovým prvkem operace $\oplus$. Pro získání výsledku operace scan je potřeba navíc posunout všechny prvky doleva a do posledního prvku uložit výsledek operace reduce – součet všech prvků.

Celková časová složitost je logaritmická – obě části výpočtu podle stromu jsou logaritmické, a případný posun (z výsledku prescan na scan) konstatní (každý procesor přečte hodnotu z vedlejšího prvku).

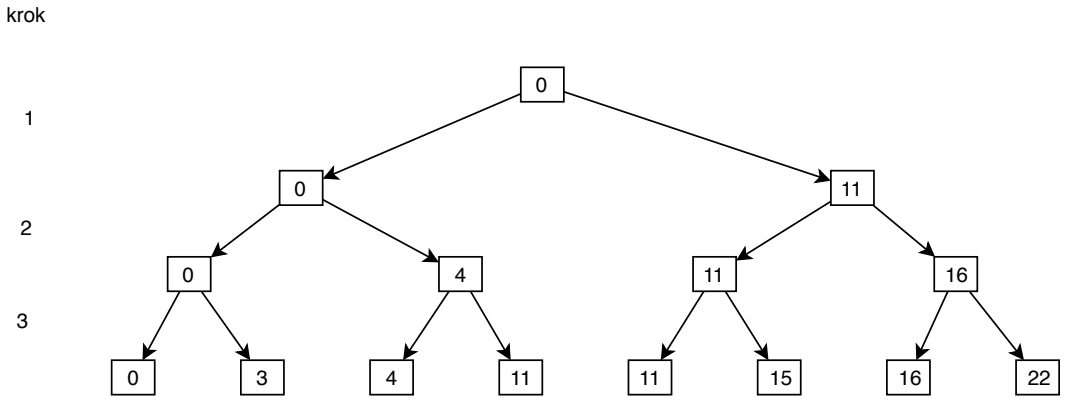
5.2 Segmentovaný scan

Některé paralelní algoritmy vyžadují provádět scan na nezávislých skupinách prvků. Pro tuto operaci je potřeba aby prvky ve stejné skupině byli seskupeny v sousedních pozicích, protože je potřeba pole příznaků, které označuje prvky, které jsou začátkem segmentu. Pokud jsou skupiny prvků označeny unikátním číslem, může takový příznak být vytvořen porovnáním z číslem skupiny předchozího prvku každým procesorem.

Pro operaci segmentovaného scanu je využita operace obyčejného scanu, ale za použití speciálního binárního asociativního operátoru, který bere v úvahu příznaky segmentu. Označme takový operátor $\odot$. Každý z operandů tohoto operátoru je dvojce (příznak, hodnota) a je definován:

$$(f_a, a) \odot (f_b, b) = (f_a\ or\ f_b\ , (f_b \times a) \oplus b)$$

or je logická disjunkce, $\oplus$ je původní operátor na kterém se provádí segmentovaný scan a $\times$ je definován:



(a)

krok	prvky v poli							
výsledek reduce	3	4	7	11	4	5	6	25
nulový prvek	3	4	7	11	4	5	6	0
1	3	4	7	0	4	5	6	11
2	3	0	7	4	4	11	6	16
3	0	3	4	11	11	15	16	22

(b)

Obrázek 5.3: Druhá část operace plus-scan (a) na stromě (b) v poli. Zvýrazněné prvky jsou v daném kroku změněné.

$$f \times x = \begin{cases} 0_{\oplus} & f = 1 \\ x & f = 0 \end{cases}$$

$0_{\oplus}$ je nulový prvek operace $\oplus$. Operátor $\odot$ tedy při výpočtu hodnoty přičítá levý operand pouze, pokud pravý operand není začátkem segmentu (nemá příznak 1). Při výpočtu nového příznaku je díky použití logického *or* udržována informace zda součet hodnot již zahrnuje prvek začínající segment a tedy prvky vlevo od začátku segmentu nejsou do sumy hodnot zahrnuty (obr. 5.4).

Oproti jednoduchému scanu je segmentovaný scan náročnější pouze v použitém operátoru, časová složitost je tedy stále logaritmická.

5.3 Suffix sum

Operace suffix sum je podobná operaci scan, ale pracuje nad jednosměrně vázaným seznamem, kde každý prvek drží kromě své hodnoty jen odkaz na svého následníka, případně žádného následníka pokud se jedná o poslední prvek v seznamu. Následníkem posledního prvku je sám tento prvek.

Cílem operace je zjistit součet (libovolný asociativní operátor $\oplus$) všech prvků od daného prvku až do konce seznamu. Podobného principu jako u operace scan využít nelze, protože

krok		prvky v poli							
	F (příznaky segmentu)	1	0	1	0	0	1	0	0
	X (hodnoty)	3	1	7	0	4	1	6	3
	(F,X)	(1,3)	(0,1)	(1,7)	(0,0)	(0,4)	(1,1)	(0,6)	(0,3)
1		(1,3)	(1,4)	(1,7)	(1,7)	(0,4)	(1,1)	(0,6)	(0,9)
2		(1,3)	(1,4)	(1,7)	(1,7)	(0,4)	(1,1)	(0,6)	(1,10)
3		(1,3)	(1,4)	(1,7)	(1,7)	(0,4)	(1,1)	(0,6)	(1,10)
nulový prvek		(1,3)	(1,4)	(1,7)	(1,7)	(0,4)	(1,1)	(0,6)	(0,0)
1		(1,3)	(1,4)	(1,7)	(0,0)	(0,4)	(1,1)	(0,6)	(1,7)
2		(1,3)	(0,0)	(1,7)	(1,4)	(0,4)	(1,7)	(0,6)	(1,1)
3		(0,0)	(1,3)	(1,4)	(1,7)	(1,7)	(1,11)	(1,1)	(1,7)
posun		(1,3)	(1,4)	(1,7)	(1,7)	(1,11)	(1,1)	(1,7)	(1,10)
výsledek		3	4	7	7	11	1	7	10

Obrázek 5.4: Výpočet segmentovaného plus-scanu v poli pomocí $\odot$-scanu. Označeny jsou prvky které jsou v každém kroku modifikovány.

není známo ani pořadové číslo prvku. Je využita metoda zdvojování cesty, kde je v každém kroku zjištěn prvek seznamu 2krát dále než stávající následník. Přitom je ke stávající hodnotě přičtena hodnota přeskakovaného prvku (algoritmus 2).

Algorithm 2 suffix sort [6]

```
1:  for i ← 1 to N do in parallel
2:      if succ(i) = i then
3:          rank(i) ← 0
4:      else
5:          rank(i) ← value(i)
6:
7:      for j ← 1 to ⌈log_2 N⌉ do
8:          rank(i) ← rank(i) + rank(succ(i))
9:          succ(i) ← succ(succ(i))
10:
11:     if value(last) ≠ 0 then
12:         rank(i) ← rank(i) + value(last)
13:
```

Pokud je cílem pouze zjistit pozici každého prvku v seznamu, je k tomu tento algoritmus možný využít, přičemž každý prvek má na začátku $value(i) = 1$. Výsledkem je pro každý prvek vzdálenost od konce, což lze případně jednoduše přepočítat na pozici: $N - rank(i)$.

Stejně jako u operace scan je složitost logaritmiká, ale narozdíl od scan pro takovou složitost vyžaduje možnost číst více procesory ze stejného paměťového místa (CREW PRAM). Navíc jsou celou dobu aktivní všechny procesory. Tyto skutečnosti mohou na reálném systému způsobovat zpomalení oproti teoretické složitosti.

Kapitola 6

Návrh akcelerace BWT pomocí GPU

Pro akceleraci BWT je možné využít vlastnosti BWT, že jednotlivé bloky jsou navzájem naprosto nezávislé. Zpracování jednotlivých bloků může tedy probíhat zároveň. Efektivita tohoto přístupu bude velmi záviset na velikosti bloku pro BWT a také na velikosti vstupních dat. Pokud vstupní data budou o velikosti několika jednotek bloků, bude možná paralelizace taktéž nízká. Tento způsob je také velmi náročný na paměť, což způsobí využívání větších ale pomalejších pamětí obsažených na GPU. Výhodou je možnost použití na oba směry transformace, a také na případné další transformace při dalším zpracování pro kompresi/dekompresi.

6.1 Dopředná transformace

Jedinou operací v dopředné transformaci je řazení vstupu na základě klíče (popsáno v kapitole 2.2). Žádoucí je tedy využít některé z paralelních metod řazení (v CUDA dostupné funkce, např `sort_by_key`). Problémem zde ale je, že klíčem je řetězec o délce stejné jako je vstup. Jednou možností je brát v úvahu jen omezený počet znaků, a případné kolize vyřešit odděleně, přičemž předpoklad je, že těchto kolizí bude relativně málo.

6.2 Zpětná transformace

U zpětné transformace (algoritmus 1) jsou kritické dva cykly o délce vstupního bloku (N).

První z cyklů přiřadí každému symbolu ve vstupu počet stejných symbolů vyskytujících se před ním do pole C. Tento cyklus je možné nahradit paralelním algoritmem scan pro sumu, který ke každému prvku pole přiřadí částečný součet všech prvků od začátku pole po daný prvek. Pro tento přístup by bylo potřeba předzpracovat vstupní blok jednou pro každý symbol abecedy $c \in \Sigma$, tak aby pole o velikosti N obsahovalo 1 právě pokud odpovídající prvek v L obsahuje daný znak, jinak 0. Následně se pro každý symbol provede scan nad takto připraveným polem a tím je vytvořeno několik polí C – jeden pro každý symbol z abecedy.

Druhý cyklus délky N je mnohem obtížnější paralelizovat, protože následující krok vždy vychází z výsledku předchozího. Namísto jednoho dekódování od pozice a, tedy od posledního symbolu, je možné provádět dekódování od několika různých míst současně. Není

však možné zjistit jak daleko budou tyto nová počáteční místa daleko od sebe ve výsledném dekódovaném textu. Bylo by nutné tedy provádět periodické kontroly, zda více vláken nedekóduje stejnou část textu, což by ve výsledku mohlo zabrat více času než sekvenční implementace.

Kapitola 7

Implementace

7.1 dopředná BWT

Bylo vytvořeno několik implementací dopředné BWT. Sekvenční varianty slouží hlavně pro porovnání výsledného zrychlení paralelních variant. Všechny varianty načítají specifikovaný soubor z disku postupně po blocích dané velikosti a po transformaci je výstup opět zapsán do výstupního souboru. Pokud možno je paměť, která je opakovaně potřebná pro každý blok alokována pouze jednou na začátku programu, protože případné alokace jak v hlavní paměti, tak v paměti grafické karty je časově náročná. Navíc v grafické kartě opakovaná alokace paměti koliduje s možností využívat streamy pro specifikaci a běh několika nezávislých úloh.

Pro sekvenční implementaci BWT stačí seřadit pole všech indexů, které je inicializováno $R[i] = i$, kde klíčem pro řazení je řetězec (rotace původního řetězce) začínající na indexu i. K tomu je použita funkce `std::sort` se specifikovanou vlastní porovnávací funkcí, která postupně porovnává znaky na porovnávaných indexech do první neshody. Funkce `std::sort` využívá řadící metodu, která kombinuje quicksort jako primární algoritmus, a na menší části pole heapsort a insertion sort.

Pro kontrolu byl navíc velmi podobně implementován vlastní quicksort s výběrem prostředního prvku pro rozdělení pole.

Pro paralelní zpracování existuje několik způsobů jak rozdělit práci mezi procesory na GPU. Následuje několik implementací, které rozdělují práci na různých úrovních a lišící se způsobem řazení.

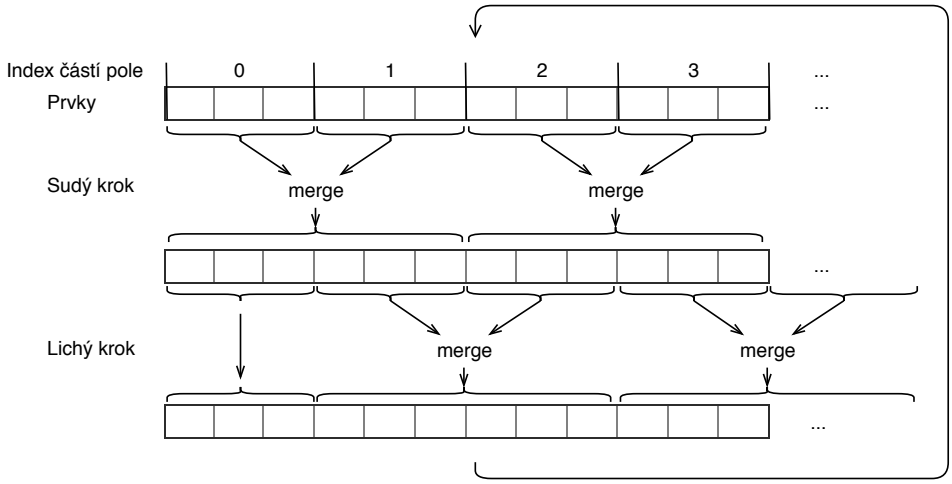
7.1.1 thrust varianta

Tato varianta využívá pouze funkce z knihovny thrust. Všechny volání kernelu na grafické kartě jsou tedy prováděny skrze tuto knihovnu. Místo přímé implementace kernelu jsou většinou jako parametr specifikovány funkce popisující transformaci jednoho prvku. Díky tomu muže být úloha optimálně rozdělena na prostředky grafické karty. Nevýhodou je, že knihovna thrust nenabízí možnost aplikovat funkci na několik nezávyslých dat – např. spuštění řazení na několika nezávyslých blocích.

Stežejní v implementaci je funkce `thrust::sort`. Stejně jako v `std::sort` je možné specifikovat již zmíněnou porovnávací funkci. `thrust::sort` podle velikosti předaného pole určí počet počet (GPU) bloků, mezi které je vstupní pole rozděleno. Každý blok, obsahující 256 vláken paralelně seřadí přiřazenou část pole, a následné jsou takto seřazené části slučovány mezi bloky pomocí merge-splitting sort.

7.1.2 Merge-splitting varianta

V této implementaci je každý blok vstupního souboru přiřazen jednomu GPU bloku. V GPU bloku obsahujícím až 1024 vláken je vstupní blok rovnoměrně rozdělen mezi jednotlivá vlákna. Každé vlákno svou část pole sekvenčně seřadí heapsortem. Heapsort byl zvolen, protože kromě dobré časové složitosti nepotřebuje další pomocnou pamět na zásobníku (oproti quicksortu, který ji potřebuje na rekurzivní zanoření), což by mohl být na GPU problém. Po seřazení jednotlivých částí pole jsou sousední seřazené části postupně slučovány a opět půleny (obr. 7.1). Pokaždé se spojují (merge) 2 kratší posloupnosti, a toto slučování mohou současně provádět 2 vlákna bez ztráty výkonu – jedno vlákno vytvoří jen nižší polovinu sloučené seřazené posloupnosti, druhé vlákno začíná od největších prvků a vytvoří jen vyšší polovinu



Obrázek 7.1: Merge-splitting sort po seřazení každé z částí

Pro seřazení celého pole tímto způsobem je potřeba celkem stejný počet kroků jako je částí na které je pole rozděleno. Více vláken tedy rychleji zvládne přípravnou část kde si každý seřadí svoji část posloupnosti, ale znamená to na druhou stranu více kroků. Mezi každým krokem je potřeba všechna vlákna synchronizovat a to při větším počtu představuje jistou časovou náročnost.

Všechna vlákna mají tedy celou dobu práci, a to relativně rovnoměrně rozloženou (až na samotné porovnávání, ve kterém se může lišit počet porovnaných znaků). Polovina vláken slučuje nižší polovinu 2 posloupností, druhá polovina vyšší polovinu. Je tedy vhodné rozdělit vlákna tak, aby vlákna v jednom warpu prováděla pouze jedno z těchto slučování (obr. 7.2)

7.1.3 Bucket sort varianta

Další implementace využívá podobný způsob paralelního řazení – každé vlákno si první seřadí jemu přiřazenou část. Poté jsou seřazené části postupně slučovány po 2 do stále větších částí (obr. 7.3). Každý krok je tedy slučováno méně částí, a každá část je dvojnásobně velká, dokud výsledkem není jedno celé seřazené pole.

Každý krok se počet aktivních vláken půlí. Nejnáročnější jsou poslední kroky, kde jsou slučovány největší části pole malým počtem vláken. Proto je i zde vhodné rozdělit vlákna

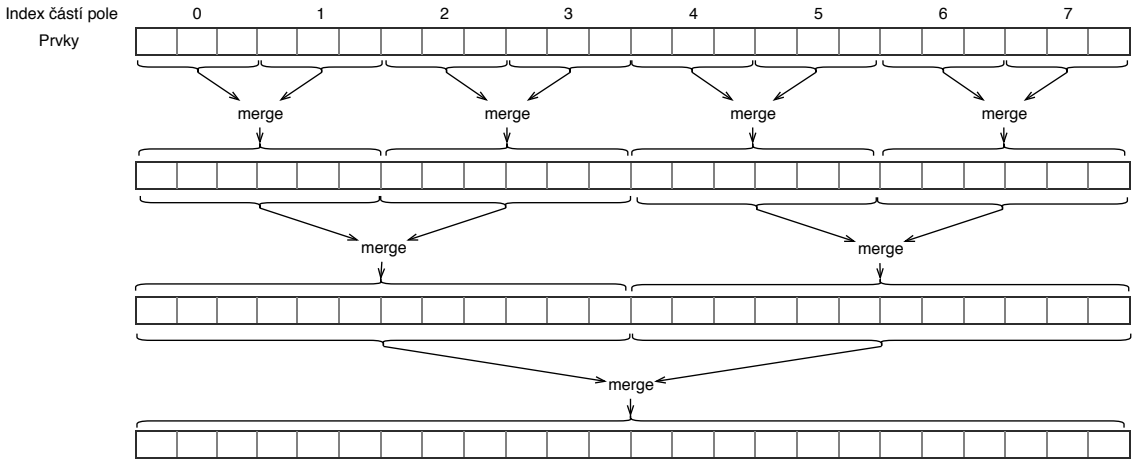
Index vlákna	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Přiřazená část pole	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Způsob slučování	L	H	L	H	L	H	L	H	L	H	L	H	L	H	L	H

(a)

Index vlákna	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Přiřazená část pole	0	2	4	6	8	10	12	14	1	3	5	7	9	11	13	15
Způsob slučování	L	L	L	L	L	L	L	L	H	H	H	H	H	H	H	H

(b)

Obrázek 7.2: Rozdělení částí řazeného pole jednotlivým vláknům. Typ řazení odpovídá půlce které vlákno při slučování vytváří při prvním kroku: L - nižší, H - vyšší (a) stejné indexy (b) pro stejný typ slučování ve warpu.



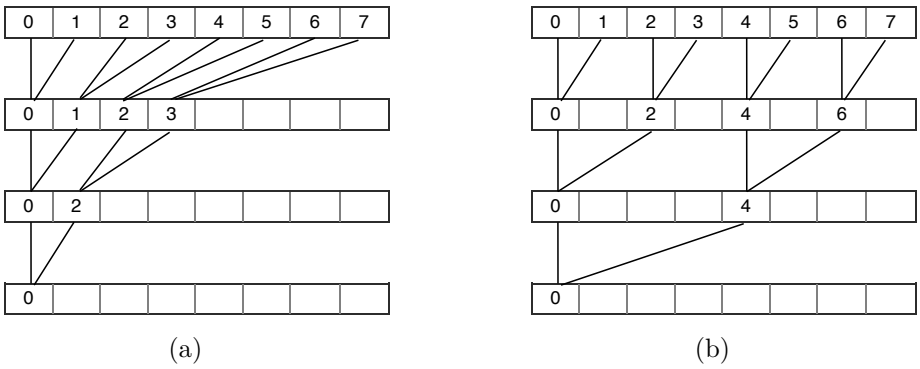
Obrázek 7.3: Bucket sort po seřazení každé z částí

tak, aby se minimalizovala divergence warpu. Pokud jsou vlákna rovnoměrně rozdělována mezi aktivní warpy, je v každém warpu aktivní méně vláken a dochází k divergenci méně (obr. 7.4).

7.1.4 Řazení po částech

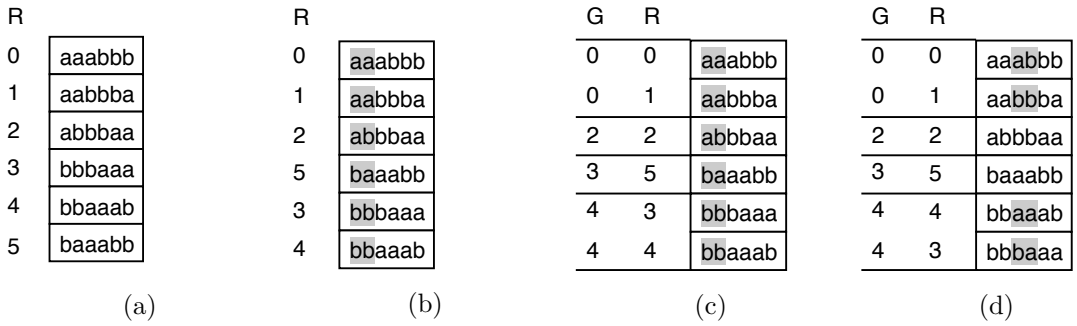
Předchozí implementace při řazení a porovnávání rotací prochází odpovídající části vstupního řetězce tak dlouho dokud u porovnávaného páru není rozhodnuta nerovnost – mezi všemi rotacemi je úplné uspořádání.

Tato implementace porovnává rotace vždy po předem zadaných částech a může se tedy stát že některé rotace budou při řazení vyhodnoceny jako rovny. Pokud se toto stane, jsou tyto rotace seskupeny a v další fázi, kde se uvažuje jejich další část, jsou porovnávány pouze rotace v rámci stejné skupiny. Rotace které jsou jedinými členy své skupiny, a jsou tedy už úplně seřazeny, se nemusí v následujících fázích řadit znovu, a díky tomu postupně klesá počet porovnávaných rotací.



Obrázek 7.4: Rozdělení vláken při jednotlivých krocích bucket sortu. (a) vždy vlákna s nejnižším indexem (b) vlákna rovnoměrně rozděleny. V druhém případě by pro velikost warpu 4 byly vlákna v posledních 2 krocích samotné ve warpu.

Příkladem nechť je vstupní řetězec `aaabbb`, kde se rotace řadí postupně po 2 znacích. Po prvotním řazení jsou rotace seřazeny jen podle jejich prvních 2 znaků (obr. 7.6b). Rotace, které tuto dvojici znaků mají stejnou se sousedními rotacemi jsou seskupeny (obr. 7.6c). Uvnitř každé skupiny (s více než 1 prvkem) zvlášť je v další fázi provedeno řazení podle dalšího páru znaků (obr. 7.5d). Následně je provedeno další rozdělení skupin podle právě porovnávaného páru znaků. Protože po dalším vytvoření skupin už jsou všechny skupiny jednočlenné, je řazení ukončeno.

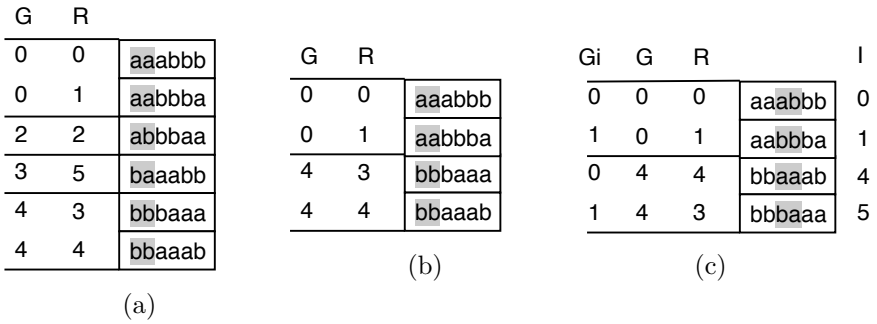


Obrázek 7.5: Fáze řazení po částech (po 2 znacích) na vstupním řetězci `aaabbb`. (a) po inicializaci, (b) seřazení podle prvních 2 znaků, (c) vytvoření skupin (pole G - číslo skupiny ke každému prvku), (d) Po seřazení podle druhých 2 znaků.

Paralelní řazení v první fázi je prováděno knihovní funkcí `thrust::sort_by_key`, kde klíčem pro řazení je právě relevantní část rotace (skupina znaků o pevně dané velikosti). Pro paralelní seskupení rotací po částečném seřazení je každý element (stále stejná skupina znaků, podle které se právě řadilo) porovnán se svými bezprostředně sousedními elementy. Pokud se od obou liší, je element úplně seřazen a dalších fází už se neúčastní. Pokud je odlišný od svého levého souseda, je začátkem skupiny a zapíše svůj index do odpovídajícího pole jako číslo skupiny. Pokud má stejného levého souseda, zapíše jako své číslo skupiny číslo 0. Následně je číslo skupiny rozdistribuováno pomocí operace max-scan (`thrust::inclusive_scan`). Řazení v každé další fází je prováděno opět pomocí `thrust::sort_by_key`, ale tentokrát je součástí klíče navíc i číslo skupiny. Jsou tedy sice

řazeny všechny doposud neseřazené rotace současně, ale díky číslu skupiny nejsou prvky mezi různými skupinami promíchávány.

Protože jsou pro řazení ve vyšších fázích prvky přesunuty do menšího pole (obr. 7.6b), je potřeba na konci každé fáze rotace seřazené ve své skupině přesunout na odpovídající místo v poli se všemi rotacemi. Číslo skupiny odpovídá indexu prvního prvku ve skupině. Pomocí operace segmentovaného plus-prescanu (`thrust::exclusive_scan_by_key`) nad polem jedniček, kde číslo skupiny odpovídá číslu segmentu, je získán posun od prvního prvku v rámci skupiny. Pomocí součtu těchto 2 čísel je tedy možné zjistit cílový index právě seřazené rotace (obr. 7.6c).



Obrázek 7.6: Řazení pouze neseřazených prvků (a) stav po 1. řazení (b) extrakce prvků, které ještě nejsou seřaeny (nejsou samotné ve své skupině) (c) vytvoření indexu každého prvku v rámci své skupiny (Gi) a výpočet indexu (pole I) těchto prvků v původním poli se všemi rotacemi

7.1.5 Obecné optimalizace

Agregované porovnávání

Základní operace porovnávání dvou rotací vstupního řetězce postupně od daných indexů porovnává 8bitové znaky, dokud se některý neliší. Pro delší bloky textu, případně pro specifické vstupní řetězce se může porovnávat příliš dlouhý prefix pro rozhodnutí o nerovnosti. Jednou z možností urychlení je porovnávat např. 8 znaků současně, které jsou agregovány do jednoho 64bitového čísla tak, aby výsledek porovnání odpovídajících prefixů byl stejný. Pole těchto agregovaných čísel může být vytvořen předem před samotným řazením rotací. Takové pole zvětší paměťovou náročnost pro řazení rotací bloku – pro každý 8 bitový znak je vytvořen 64bitový prvek, ale může značně přispět ke zrychlení řazení. Kromě samotného 64bitového porovnávání je také zaručeno zarovnání na adresu násobku 8 bytů a tedy rychlejší přístup k samotným prvkům.

Zpracování více bloků

Pro paralelní varianty, které to připouští – nevyužívající pro řazení thrust funkci – je možné každý blok vstupního souboru zpracovávat v samostatném GPU bloku. Toho je možné docílit buď implementací kernelu, který na základě aktuálního GPU bloku vybere zpracovávanou část dat. Další možnost je zadávat spouštění kernelu na různých GPU streamech. Úlohy na rozdílných streamech mohou být prováděny současně na rozdílných multiprocesorech. V obou případech je potřeba úměrně více paměti, což může být pro větší zvolenou velikost bloku problém.

7.2 Zpětná BWT

Algoritmus zpětně transformace se skládá ze 2 základních částí: příprava pomocných struktur, pomocí které lze v konstantním čase najít pro každý symbol jeho předchůdce v původním textu, a následné využití struktury pro průchod takto odkazovaných symbolů v pořadí původního textu (případně přesně ve zpětném pořadí, jak bylo ukázáno v kapitole 2.2.2). Pro implementaci sekvenční varianty byl pouze přepsán algoritmus ukázán ve stejné kapitole.

V cyklech obou zmíněných částí jsou závislosti na předchozí iteraci, stěžující paralelní provádění. Při přípravě pomocných struktur je potřeba počítat počet výskytů každého symbolu v zakódovaném řetězci a přiřazovat k nim stávající počet. Ve druhé části je potřeba dodržet pořadí odkazovaných prvků pro zjištění pozice v dekódovaném řetězci. Obě závislosti jdou ale vyřešit pomocí základních paralelních operací.

Při počítání výskytů každého prvku (v algoritmu 1 v kapitole 2.2.2 naplňování pole C) je možné pro paralelní výpočet přiřadit každý symbol jednomu vláknu a provést operaci prescan, jednou pro každý symbol abecedy. V každé takové iteraci prescanu je potřeba připravit příznaky jedniček na místa kde se právě počítaný symbol vyskytuje, a do výsledného pole C výsledek přiřadit také pouze pokud na odpovídající pozici vstupu vyskytuje daný symbol. Pro takovýto přístup je tedy potřeba 256krát provést operaci prescan, spolu s kopírováním výsledku prescanu do pole C.

Algoritmus 1 provádí vlastně counting sort, spolu s ukládáním dalších několika pomocných ukazatelů na začáteky segmentu symbolů v seřazeném poli v jendom průchodu. Pro paralelní výpočet je vhodnější než několikrát provádět prescan jednou seřadit paralelním radix sortem . Díky stabilitě radix sortu je uchováno pořadí symbolů, což je pro následný průchod, v druhé části transformace, jednotlivými odkazy stěžejní. Pokud jsou před řazením označeny všechny vstupní symboly jejich stávajícími indexy ($A[i] = i$), a následné řazení spolu s řazenými symboly posouvá tyto indexy (`thrust::stable_sort_by_key`), je výsledkem odkaz u každého symbolu k jeho původní pozici. To je tedy odkaz ze sloupce F na odpovídající symbol ve sloupci L (notace z kapitoly 2.2 a je možný podobný průchod jako v původním sekvenčním přístupu).

Pro paralelní průchod připravené struktury je vhodné ji uvažovat jako seznam – každý prvek má kromě své hodnoty (symbol) i odkaz. Odkazem ze sloupce F do L (kde je stejný výskyt symbolu) lze získat pro každý symbol jeho následníka (následující symbol v původním textu), protože v rámci stejného řádku je symbol v F následníkem symbolu v L. Pro symbol na nově získaném řádku (pomocí odkazu) je možné opět zjistit následníka. Struktura je tedy cyklický seznam, ale je možné ji jednoduše přerušit na symbolu daným indexem pro zpětnou BWT, který je uložen na začátku vstupního bloku. Pomocí operace suffix sum je možné zjistit vzdálenost od konce seznamu, a po zjištění přesné pozice může každé vlákno již nezávisle na ostatních svůj symbol na správné místo ve výstupním řetězci.

Inverzní kroky navazujících metod, tedy inverzní move-to-front a Huffmanovo dekódování bylo implementováno pouze sekvenčně. Důvodem je rychlost provedení těchto inverzních kroků již v sekvenční variatně. Tyto kroky jsou také obtížně paralizovatelné. U inverzní MTF je sekvenční závislost na symbolech které byli již dekódovány. U Huffmanova dekódování nelze určit kde v zakódovaném řetězci začíná zakódovaný symbol bez dekódování předchozích.

7.3 Move-to-front

V sekvenční variantě se ukládá do pomocného pole na každý vstupní symbol jeho transformovaná hodnota – právě zakódované znaky mají hodnotu nastavenou na 0 a ostatní jsou podle potřeby posunuty o 1 výše. Vzhledem k tomu že po BWT transformaci jsou očekávány převážně nízké hodnoty, není velký problém při vyhledávání sybmolu procházet pomocné pole po symbolech se vzestupnou zakódovanou hodnotou.

Pro paralelní verzi je problém že stav pomocného pole, a tedy zobrazení ze vstupního symbolu na jeho transformovanou hodnotu, je závislé na předchozích symbolech. Není ale závislý na přesném pořadí. Důležitý je pro každý symbol pouze pozice jeho posledního výskytu. Tohoto je v implementaci využito tak, že je vstup rozdělen na menší části, a pro každou část je zjištěna poslední pozice každého symbolu. Každá část (256 bytů) je přiřazena jednomu vláknu, a to zjistí poslední index každého znaku pouze v jeho části. Tato informace je rozšířena do všech částí pomocí max-prescanu nad jednotlivými částmi. 256 bytů bylo zvoleno, protože každé vlákno si musí uchovávat až 256 indexů, takže je na takový přístup potřeba pomocná pamět o velikosti 1 index na 1 vstupní symbol.

Prescan je teoreticky možný provádět na libovolných typech s libovolným asociativním operátorem. Při pokusu s `thrust::exclusive_scan` byl ale problém s velikostí celých struktur s 256 indexy jako jednotlivé prvky pro scan. Místo toho je prováděn max-prescan pro každý symbol zvlášť. To má za následek pouze jiné uspořádání výsledků prescanu, časová složitost je stejná.

Po tom co jsou známy poslední indexy každého symbolu pro každý začátek 256 bytové části, každé vlákno si vytvoří vlastní pomocné move-to-front pole pomocí seřazení (sekvenční - každé vlákno provádí nezávisle svou část). Symboly s největším indexem budou mít nejnižší pořadí v move-to-front poli. Následně každý procesor sekvenčně projde svou část a upravuje seznam podle symbolů které zakódoval, stejně, jako v sekvenční variantě.

7.4 Huffmanovo kódování

Pro jednodušší paralelizaci byla implementována sekvenční a paralelní verze neadaptivního Huffmanova kódování. Každý BWT blok je zpracováván nezávisle, tedy pro každý blok je vytvářen nový Huffmanův strom. Pro kódování je strom využit pro inicializaci pomocného pole s kódy pro každý symbol, společně s délkou kódu – strom už nemusí být procházen při kódování každého symbolu zvlášť, místo toho je kód symbolu přímo vyhledán v pomocném poli. Naopak při dekódování zůstává nejlepší reprezentací Huffmanův strom, který je procházen podle vstupních bitů pro dekódování symbolu.

Huffmanovo kódování se skládá ze 4 částí: zjištění četnosti každého symbolu, vytvoření Huffmanova stromu, vytvoření kódu pro každý symbol, a samotné kódování vstupu.

U paralelní implementace může pro výpočet četnosti každé vlákno obstarat jeden vstupní symbol, a k odpovídajícímu symbolu přičíst jedničku. Pokud více vláken stejný symbol, pokouší se zvýšit číslo na stejném paměťovém místě. Obyčejné přičtení hodnoty do paměti znamená kroky čtení z paměti, inkrementaci v registru a zápis zpět do paměti, a nesynchronizovaná vlákna by si tedy mohla navzájem přepisovat zapsanou hodnotu. Prostředí CUDA poskytuje atomickou funkci `atomicAdd`, která hodnotu v paměti zvýší, jakoby tato operace byla atomická. Nejedná se o atomickou instrukci procesoru – pouze je zajištěna synchronizace mezi vlákny tak, aby nedošlo k nesprávnému pořadí čtení/zápisu mezi vlákny. Může se tedy jednat o časově náročnou režii.

Výsledná četnost je následně poslána z GPU do hostující paměti na zpracování CPU. CPU z četností vytvoří Huffmanův strom a následně kódy pro každý ze symbolů. Tyto kódy jsou poslány na GPU kde je prováděno samotné kódování vstupu.

Paralelní kódování je možné do jisté míry provádět nezávisle na okolních symbolech – u neadaptivního Huffmanova kódu je po celou dobu kód pro daný symbol stejný. Odpovídající kód ale musí být uložen na správné bity ve výstupním řetězci. Není potřeba znát přesně zakódované předcházející symboly, ale je potřeba celková délka těchto zakódovaných symbolů. Při přiřazení každého symbolu jednomu vláknu každé vlákno zná délku kódu svého symbolu, a tuto informaci lze rozšířit pomocí plus-prescanu. Výsledkem je pro každé vlákno počáteční bit, na který má zapisovat svůj kód. Toto může být prováděno relativně nezávisle, ale vzhledem k tomu, že vlákna nezapisují data zarovnaná na celé byty, je potřeba řešit podobný problém jako s atomickým sčítáním u počítání četnosti symbolů.

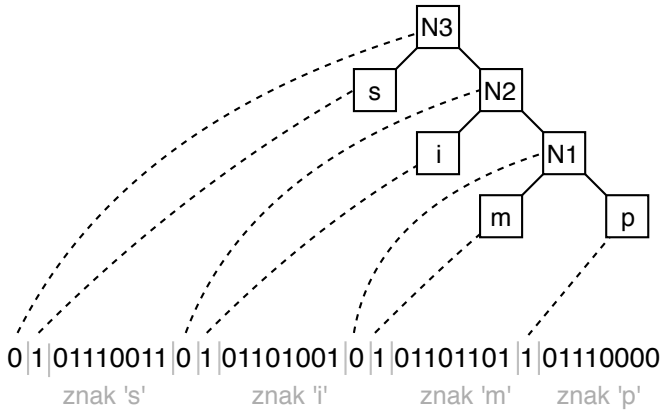
Každou operaci lze převést na ekvivalentní, která se chová atomicky pomocí funkce `atomicCAS` (Compare And Swap)[1]. Tato funkce přijímá kromě adresy paměťového místa, které má být upraveno, hodnoty *compare* a *value*. Pokud je na dané adrese v paměti hodnota rovna *compare*, je nahrazena hodnotou *value*. V opačném případě je paměťové místo ponecháno beze změny. V každém případě funkce vrací hodnotu v paměti před provedením této operace. Operace převedená pomocí `atomicCAS` na atomickou se tedy opakovaně snaží vyměnit (swap) hodnotu v paměti, kde při každém pokusu zadává očekávanou hodnotu před výměnou, a požadovanou novou hodnotu (výpis 7.1). Do paměti je hodnota zapsána pouze pokud se při výpočtu v lokální paměti (v lokálních registrech) nezměnila jiným vláknem. V každé iteraci tedy úspěšně provede operaci alespoň 1 vlákno ze všech které se snaží zapsat na stejné paměťové místo).

```
old_value = *adress;
do {
  assumed = old_value;
  // výpočet nové hodnoty na zakladě předchozí,
  // v~lokální paměti
  new_value = f(old_value);
  old = atomicCAS(adress, assumed, new_value);
} while (assumed != old)
```

Výpis 7.1: Použití `atomicCAS` k implementaci vlastní atomické operace. `f()` je požadovaná neatomická operace.

Tímto postupem je implementován atomický zápis jen několika bitů (zakódovaného symbolu) na správné místo výstupního řetězce.

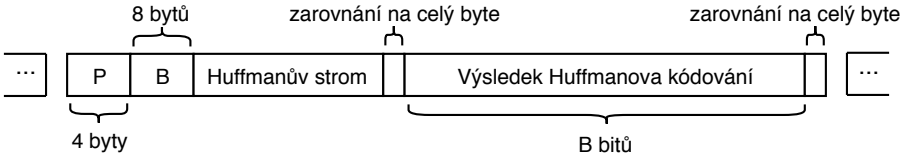
Do výstupu musí být zakódován kromě samotného zakódovaného vstupního řetězce i Huffmanův strom. Bit 0 značí pomocný (nelistový) uzel a je následovaný jeho dvěma potomky, v prefixové notaci (první levý, po hraně reprezentující kódový bit 0, pak pravý s bitem 1). Bit 1 značí listový uzel a je následovaný odpovídajícím nezakódovaným symbolem. Příklad zakódovaného stromu je znázorněn v obr. 7.7. Reálně je kódovaný výstup move-to-front transformace (čísla od 0 do 255, ne znaky), ale princip je stejný. Díky prefixové notaci uzlů je možné v každé části dekódování stromu znát hloubku zanoření (úroveň stromu), a není tedy potřeba explicitně specifikovat délku zakódovaného stromu – dekódování končí jakmile není očekáván žádný další uzel.



Obrázek 7.7: Zakódovaný Huffmanův strom z obr. 3.1

7.5 Komprimovaný formát

Při kompresi jsou na každý blok nezávisle postupně aplikovány BWT, Move-to-front transformace a Huffmanovo kódování. Každý zakódovaný blok obsahuje na začátku 2 čísla pevné velikosti: Index od kterého je potřeba začít dekódovnání BWT a délka zakódovaného řetězce v bitech (bez délky zakódovaného stromu). Následuje zakódovaný strom a nakonec výstup Huffmanova kódování (obr. 7.8).



Obrázek 7.8: Formát 1 zakódovaného bloku. P je index pro začátek dekódování ve zpětné BWT, B je počet zakódovaných bitů.

Kapitola 8

Testování

Testy byli prováděny na procesoru Intel Core i5-6300HQ s frekvencí 2,3GHz a grafické kartě NVIDIA GeForce GTX 960M, která má 8 multiprocesorů, každý plně vytížitelný 1024 vlákny. Při měření časů jednotlivých implementací je měřen čas operací opakující se pro každý blok. Jednorázová inicializace, např. alokace paměti a inicializace grafické karty nejsou brány v potaz. Na druhou stranu jsou do měřeného času jsou zahrnuty kromě samotné transformace i čtení a zápis vstupního a transformovaného souboru a u paralelních verzí přenos dat z hlavní paměti na grafickou kartu.

8.1 Dopředná BWT

V následujících sekcích jsou porovnány jednotlivé sekvenční (1 vlákno na CPU) a paralelní (GPU) implementace dopředné BWT, bez navazujících kompresních metod.

8.1.1 Základní implementace

Následující testy byli provedeny na implementacích bez optimalizace pro agregované porovnávání znaků (kapitola 7.1.5), tedy znaky jsou při řazení rotací porovnávány postupně po jednom. V tabulce 8.1 jsou zobrazeny časy zpracování souboru `bible.txt`, 3.85MB anglického textu bible. Tento soubor je dostatečně velký na testování různé velikosti bloku a obsahuje relativně mnoho znaků s rovnoměrnou četností (všechna písmena anglické abecedy, mezery a interpunkce), je tedy pro BWT jednoduchý na zpracování. To je vidět i v tabulce 8.2, která ukazuje že průměrně je potřeba asi 4 znaky na seřazení všech rotací. Toto číslo se samozřejmě zvětšuje při větších blocích, protože při větším množství porovnávaných rotací se zvětšuje počet s podobným prefixem.

Ze všech variant rychlostí značně převažuje použití řazení z knihovny `thrust`. Vzhledem k tomu, že se v této variantě zpracovává v jednu chvíli pouze 1 blok, mohou jeho části být přiřazeny různým GPU blokům a tedy různým multiprocesorům. Tak se lépe využívají rychlé paměti (vyrovnávací paměť a sdílená paměť).

Varianty bucket_sort a merge_split_sort zpracovávají v jednu chvíli 8 bloků (testováno na grafické kartě s 8 multiprocesory). merge_split_sort obsahuje příliš mnoho kroků, ve kterém se slučují seřazené části pole, a přitom dochází k velkému počtu zbytečných porovnání. Použití více vláken ještě více uškodí, protože těchto kroků je více a navíc do globální paměti přistupuje více vláken. Při větších blocích není možné už vstupní soubor nepřesahuje velikost 8 bloků a je tedy navíc vidět velká časová ztráta zpracováním příliš malého počtu bloků

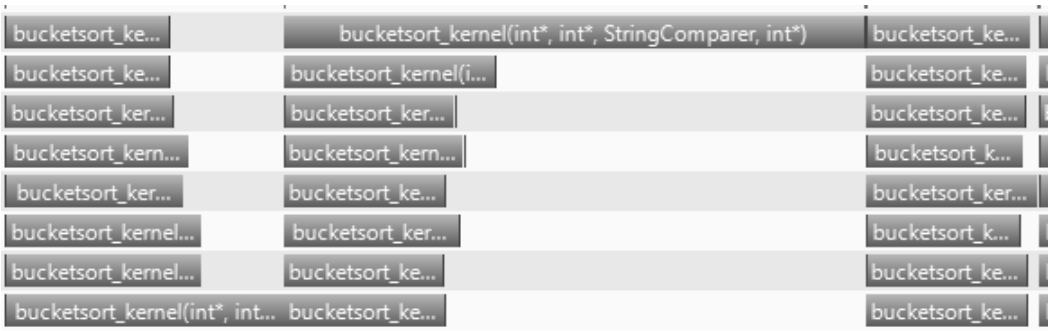
varianta \ velikost bloku	64kB	256kB	1MB	4MB
std::sort (CPU)	1517	1826	2327	2846
quicksort (CPU)	1433	1645	2051	2448
thrust::sort (GPU)	201	184	189	303
bucket_sort, 256 vláken / blok (GPU)	2288	2597	5056	17467
bucket_sort, 1024 vláken / blok (GPU)	3081	3498	4662	15995
merge_split_sort, 256 vláken / blok (GPU)	3105	3572	6490	23917
merge_split_sort, 1024 vláken / blok (GPU)	11147	23159	18694	45843

Tabulka 8.1: Časy transformace (v milisekundách) souboru `bible.txt` při rozdělení na různě velké bloky pro BWT transformaci.

varianta \ velikost bloku	64kB	256kB	1MB	4MB
std::sort (CPU)	102mil. (3.63)	116mil. (4.21)	129mil. (4.79)	141mil. (5.34)
quicksort (CPU)	90mil. (3.2)	100mil. (3.74)	111mil. (4.25)	123mil. (4.73)
thrust::sort (GPU)	103mil. (3.11)	118mil. (3.61)	132mil. (4.12)	147mil. (4.63)
bucket_sort, 256 vláken / blok (GPU)	85mil. (2.99)	101mil. (3.29)	117mil. (3.57)	133mil. (3.84)
bucket_sort, 1024 vláken / blok (GPU)	77mil. (3.23)	92mil. (3.62)	109mil. (3.75)	125mil. (4.01)
merge_split_sort, 256 vláken / blok (GPU)	1084mil. (2.4)	1100mil. (2.58)	1116mil. (2.8)	1133mil. (3.05)
merge_split_sort, 1024 vláken / blok (GPU)	4177mil. (2.77)	4193mil. (2.83)	4209mil. (2.93)	4225mil. (3.11)

Tabulka 8.2: počet porovnání párů rotací při zpracování souboru `bible.txt`. V závorce je uveden průměrný počet porovnaných znaků na každé porovnání párů rotací.

U varianty bucket_sort je jeden z problému nedostatečné využití rychlých pamětí, protože je na celý blok přiřazen pouze jeden multiprocesor. Dalším problémem je při zpracování více bloků současně. Některé bloky mohou být zpracovány rychleji a multiprocesor potom musí čekat než jeho výsledek přijde na řadu pro zapsání na disk a je mu přiřazen další blok vstupu (obr. 8.1). Některé vstupní bloky mohou být značně jednodušší pro zpracování BWT – záleží hlavně na průměrném počtu znaků k porovnání dvou rotací.



Obrázek 8.1: Výstup z profilovacího nástroje NVIDIA visual profiler. Vytížení jednotlivých streamů zpracovávajících různé vstupní bloky u bucket sort varianty. Vertikálně jsou zobrazeny různé streamy, horizontálně čas.

8.1.2 Agregované porovnávání

Při agregování vstupních znaků do 64 bitových čísel na rychlejší porovnání jsou všechny implementace téměř 2x rychlejší (viz tabulka8.3). Pole těchto 64 bitových čísel se musí pro každý blok předem připravit a vyžaduje další pomocnou paměť (navíc 8 bytů na 1 vstupní byte), ale čas strávený samotným řazením je u všech implementací je stále převládající.

Díky porovnáváním 8 znaků je převážná většina porovnání rotací vyřešena v prvním vyhledaní. Tabulka 8.4 ukazuje průměrný počet porovnání skupin 8 znaků (agregovány pro porovnávání) na uspořádání páru rotací. Pro testovaný vstupní soubor tedy už není samotné porovnávání rotací příliš zrychlit.

varianta \ velikost bloku	64kB	256kB	1MB	4MB
std::sort (CPU)	929	1101	1426	1862
quicksort (CPU)	910	1090	1366	1742
thrust::sort (GPU)	143	88	110	178
bucket_sort, 256 vláken / blok (GPU)	822	1021	1792	6101
bucket_sort, 1024 vláken / blok (GPU)	1031	1349	1764	5774
merge_split_sort, 256 vláken / blok (GPU)	2338	2755	4504	9022
merge_split_sort, 1024 vláken / blok (GPU)	17569	26745	23294	25893
řazení po částech	973	483	383	385

Tabulka 8.3: Časy (v milisekundách) transformace souboru `bible.txt`, při připravení vstupu na porovnávání 8 znaků současně (64bitové porovnávání).

varianta \ velikost bloku	64kB	256kB	1MB	4MB
std::sort (CPU)	1.08	1.13	1.18	1.23
quicksort (CPU)	1.01	1.04	1.08	1.14
thrust::sort (GPU)	1.06	1.1	1.14	1.18
bucket_sort, 256 vláken / blok (GPU)	1.07	1.1	1.13	1.16
bucket_sort, 1024 vláken / blok (GPU)	1.09	1.13	1.14	1.17
merge_split_sort, 256 vláken / blok (GPU)	1.02	1.02	1.03	1.05
merge_split_sort, 1024 vláken / blok (GPU)	1.03	1.03	1.03	1.03

Tabulka 8.4: Průměrný počet porovnaných skupin 8 znaků na každé porovnání párů rotací při transformaci souboru `bible.txt`

.

8.2 Zpětná BWT

V tabulce 8.5 jsou uvedeny časy sekvenční (CPU) a paralelní (GPU) varianty zpětné BWT. CPU varianta je implementována podle algoritmu 1 v kapitole 2.2.2, GPU varianta pomocí operace suffix sum popsané v kapitole 7.2. GPU varianta je pro hlavně pro větší testované velikosti bloku mírně rychlejší.

varianta \ velikost bloku	64kB	256kB	1MB	4MB
CPU	66	98	204	356
GPU	90	124	231	290

Tabulka 8.5: Časy (v milisekundách) zpětné BWT transformovaného souboru `bible.txt`, pro různé velikost bloku pro BWT.

8.3 Komprese

V následujících sekcích jsou porovnávány sekvenčním a paralelní implementace celého kompresního nástroje složeného z BWT, move-to-front a Huffmanova kódování. Jsou porovnány vliv jednotlivých částí jak na časovou náročnost tak na jejich účinek na výsledný kompresní poměr.

8.3.1 Náročnost jednotlivých částí

V tabulce 8.6 jsou uvedeny časy jednotlivých částí komprese. Většina volání funkcí a kernelů pro provedení na GPU jsou vůči CPU asynchronní, takže pro účely měření času na CPU jsou před každým měřením jsou oba procesory (CPU, GPU) synchronizovány pomocí `cudaDeviceSynchronize`, což může vytvářet menší zkreslení měření, hlavně při kompresi s malými bloky.

Jediná část, která je v CPU implementaci rychlejší je dopředná BWT (použita varianta thrust sort, z kapitoly 7.1.1, společně s agregovaným porovnáváním). Zrychlení této části se projeví hlavně při zpracovávání větších bloků.

Move-to-front transformace i její zpětná transformace je v sekvenčním prostředí velmi jednoduchá a rychlá, hlavně po BWT, díky které je většina kódovaných znaků blízko vrcholu fronty (nízké zakódované číslo). V Paralelním prostředí MTF vyžaduje prescan pro každý znak a následné sekvenční řazení v každém vláknu, než může samotné kódování začít. Navíc při malé velikosti bloku nejsou ani vytíženy všechna vlákna – při velikosti 64KB se využívá pouze 256 vláken, každé kódující 256 znaků.

Podobný případ je u Huffmanova kódování. Paralelní verze používá synchronizaci mezi vlákny v podobně atomických operací jak při počítání četnosti znaků tak při samotném kódování. Čekání na tuto synchronizaci zabírá převážnou část času Huffmanova kódování na GPU.

Nejlepší způsob s těmito implementacemi je na GPU vypočítat pouze dopřednou BWT, a zbytek provádět na CPU.

8.3.2 Kompresní poměr

Jediná metoda která komprimuje svůj vstup je ve stávájícím programu Huffmanovo kódování. Její výstup a kompresní poměr se ale liší na základě předchozího zpracování. Cílem

	velikost bloku	BWT	MTF	Huffman	Celkem
komprese (CPU)	64kB	872 (90%)	38 (4%)	40 (4%)	970
komprese (GPU)		135 (5%)	2252 (91%)	66 (3%)	2465
dekomprese (CPU)		52 (30%)	32 (18%)	88 (50%)	176
komprese (CPU)	4096kB	1810 (97%)	25 (1%)	34 (2%)	1874
komprese (GPU)		170 (39%)	216 (50%)	38 (9%)	434
dekomprese (CPU)		320 (73%)	37 (8%)	80 (18%)	440

Tabulka 8.6: Časy (v milisekundách) jednotlivých částí komprese souboru `bible.txt`. Pro dekompresi jsou uvedeny časy odpovídajících inverzních transformací. V závorce je uvedena procentuální část dané části z celého času komprese. Pokud v řádku není součet 100%, může to kromě zaokrouhlovací chyby být čas přístupu k souboru a kopírování dat z/na GPU.

následujících testů je ukázat účinek předzpracování pomocí BWT a MTF na výsledný kompresní poměr (tabulka 8.7).

	velikost	Huffman	BWT 64kB	BWT 4096kB
bible.txt	3953	2167 (45.2%)	1196(69.7%)	1030 (73.9%)
e.coli.txt	4530	1133 (75.0%)	1257 (72.3%)	1251 (72.4%)
enwik8.txt	97657	62303 (36.2%)	37082 (62.0%)	31506 (67.7%)

Tabulka 8.7: Původní a komprimovaná velikost (v kB) různých souborů. Sloupec Huffman reprezentuje velikost při použití samotného Huffmanova kódování. Další sloupce značí použití BWT a MTF před samotným Huffmanovým kódováním, s různými velikostmi bloku. V závorce je uveden ušetřený prostor kompresí *(1-(komprimovaná velikost/původní velikost))*.

Při použití samotného Huffmanova kódování je ke kompresi využito jen statistické zastoupení jednotlivých znaků pro reprezentaci menším počtem bitů. Soubor `e.coli` obsahuje genom bakterie E-coli. Tento genom je do souboru zapsán jen pomocí 4 různých znaků. V tomto případě je samotné Huffmanovo kódování lepší než s předzpracováním BWT + MTF – Každý ze 4 relativně frekventovaných znaků může být reprezentován 2 bity z původních 8. Při předzpracování je přidáno množství znaků (ačkoliv některé jen s minimální četností), takže musí být i některé časté znaky reprezentovaný více než 2 bity. Toto nekompenzuje ani předchozí BWT, která pravděpodobně nemůže dostatečně využít častých vzorů v genomu, který je oproti přirozenému jazyku méně pravidelný.

Soubor `enwiki8` je prvních 10^8 bytů anglické wikipedie z roku 2006, obsahující kromě přirozeného jazyka také HTML a další formátovací značky. Kvůli větší abecedě je komprese bez předzpracování horší než u `bible.txt`, ale s BWT a MTF je výsledná komprese podobná. S větší velikostí bloku pro BWT je výsledná komprese lepší, díky delšímu kontextu které může BWT uvažovat.

Kapitola 9

Závěr

Byly vytvořeny programy pro kompresi pomocí BWT, Move-to-front transformace a Huffmanova kódování. Všechny tyto kroky jsou implementovány jak v sekvenční variantě tak s paralelní využívající GPU. Jediný krok komprese, který je na GPU rychlejší je samotná dopředná BWT. Nejrychlejší implementací stávající komprese je tedy provádění dopředné BWT na GPU, a zbývající kroky sekvenčně na CPU.

Pro dekompresi jedinný krok implementovaný paralelně pro GPU je zpětná BWT, která je jen mírně rychlejší pro větší bloky velikosti bloku. Pro dopřednou BWT bylo však dosažené výraznější zrychlení – sekvenční dopředná BWT je výrazně časově náročnější než zpětná BWT. Ostatní kroky dekomprese byli implementovány pouze v sekvenční variantě, protože jejich časová náročnost je oproti ostatním krokům zanedbatelná a sekvenční závislosti v těchto krocích stěžují paralelní implementaci.

Další rozšíření by mohlo spočívat v paralelní implementaci některé další transformace používané s BWT. Adaptivní Huffmanovo kódování by mělo zlepšit výslednou kompresi s možnosti vynechání MTF, ale má závislost na stávajícím stavu Huffmanova stromu při kódování, takže efektivní paralelní implementace může být problematická.

Run-length encoding by také měl výrazně pomoci s výslednou kompresí. V paralelní implementaci může být délka běhu stejných znaků rozšířena pomocí segmentovaného scanu. Výsledná pozice může být spočítána podobě jak u paralelního Huffmanova kódovaní, takže zrychlení oproti sekvenční variantě je možné.

V samotné BWT je možné vylepšení řazení rotací. Samotné obecné řazení je efektivně implementováno pomocí knihovny `trhust`, ale porovnávání dlouhých shodných prefixů může porovnávání zpomalit. Tento problém je řešen pomocí porovnávání po částech rotací v kapitole 7.1.4, ale případné zrychlení nepřeváží režii na vytváření pomocných struktur, takže je pomalejší než přímé řazení pomocí `thrust`. Možné zlepšení je neporovnávat vždy stejně dlouhé části, ale postupně tuto délku zdvojnásobovat (podobný postup je, sekvenčně, popisován v [2]). Tím je počet iterací snížen z M na $log\ M$, kde M je délka nejdelší shody prefixů rotací.

Pro řazení po částech by také bylo vhodné efektivnější implementace segmentovaného řazení, které je momentálně řešeno řazením párů pomocí `thrust` řazení.

Ačkoliv je zpětná BWT značně rychlejší než dopředná BWT, je u paralelní verze možné zlepšení sumy prefixů implementací, ve které nejsou celou dobu aktivní všechna vlákna. Díky tomu by menší vytížení hlavní paměti na GPU mohlo vést k dalšímu zrychlení této metody.

Literatura

[1] *Cuda toolkit documentation.*
URL https://docs.nvidia.com/cuda/

[2] Adjeroh, D.; Bell, T.; Mukherjee, A.: *THE BURROWS-WHEELER TRANSFORM: Data Compression, Suffix Arrays, and Pattern Matching.* Springer Science+Business Media, LLC, 2008, ISBN 978-0-387-78908-8.

[3] Burrows, M.; Wheeler, D. J.: *A block-sorting lossless data compression algorithm.* 1994.

[4] Farber, R.: *CUDA application design and development.* Amsterdam : Boston: Elsevier ; Morgan Kaufmann, 2011, ISBN 978-0-12-388426-8.

[5] Munshi, A.; Gaster, B. R.; Mattson, T. G.; aj.: *OpenCL programming guide.* Addison-Wesley, 2012, ISBN 978-0-321-74964-2.

[6] Reif, J. H.: *Synthesis of parallel algorithms.* Morgan Kaufman Publishers, 1991.

[7] Rissanen, J.; Langdon, G. G.: *Arithmetic Coding. IBM Journal of Research and Development*, ročník 23, 1979: s. 149–162.

[8] Salomon, D.: *Data compression: the complete reference.* Springer, 2006, ISBN 0-387-40697-2.

Příloha A

Obsah přiloženého paměťového média

- `/src` – zdrojové soubory programů a `CMakeLists.txt` pro jejich sestavení pomocí `CMake`.
- `/pdf` – text této práce, barevně i verze pro tisk.
- `/latex` – zdrojové soubory k vytvoření textu této práce.
- `/README.md` – soubor s popisem sestavení programů a jejich použití.