VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

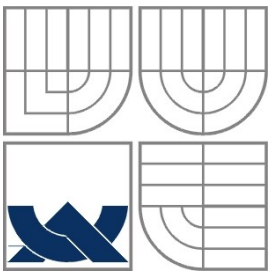
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE BURROWS-WHEELEROVY TRANSFORMACE S VYUŽITÍM GPU

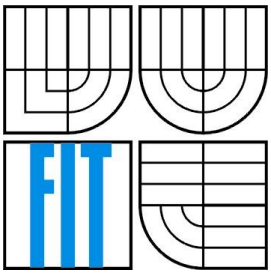
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Radek Iša
AUTHOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE BURROWS-WHEELEROVY TRANSFORMACE S VYUŽITÍM GPU

ACCELERATION OF BURROWS-WHEELER TRANSFORM USING GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Radek Iša
AUTHOR

VEDOUCÍ PRÁCE — Ing. Václav Šimek
SUPERVISOR

BRNO 2014

Abstrakt

Tato bakalářská práce se zabývá burrows-wheelerovou transformací a urychlení této transformace pomocí grafické karty. Předvedením akcelerace burrows-wheelerovy transformace na komprimační metodě. Program je napsán v jazyce C s rozšířením o syntaxi CUDA toolkitu.

Abstract

This paper presents accelerating of burrows-wheeler transform using graphic cards. Present this acceleration with commpression method. This program is written in C language with extension of CUDA toolkit syntax.

Klíčová slova

Burrows-wheelerova transformace, BWT, CUDA

Keywords

Burrows-wheeler transform, BWT, CUDA

Citace

Radek Iša: Akcelerace Burrows-Wheelerovy transformace s využitím GPU, **bakalářská práce, Brno, FIT VUT v Brně**, 2014

Akcelerace Burrows-wheelerovy transformace s využitím GPU

Prohlášení

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

……
Radek Iša
21.května.2014

Poděkování

Děkuji svému vedoucímu práce panu inženýrovi Václavu Šimkovi, za jeho trpělivost, poskytnutí rad a materiálů. Dále bych rád poděkoval všem, kteří mě poradili se zpracováním bakalářské práce.

Obsah

1 Úvod

Pomocí komprimace můžeme na stejné úložiště nahrát více dat. I když dnes existují velká úložiště dat v řádech jednotek Terabitů, vzrostl i počet dat, který chceme na těchto úložištích uchovávat. Proto je potřeba i nadále data komprimovat a vylepšovat komprimační algoritmy. Existují dva druhy komprimace: ztrátová a bezztrátová. Ztrátová komprimace má lepší komprimační poměr, toho je dosaženo nezvratným odstraněním některých přebytečných informací z dat. Jelikož ztrátové komprimace odstraní některé informace, nelze takto komprimovaná data již rekonstruovat do zcela původní podoby, ale do podoby přibližné. Proto se ztrátové komprimace využívají u dat jako je video, hudba, fotografie. U těchto mediálních dat se využívá nedokonalého lidského vnímání a nevadí nám ztráta některých informací. U dat, kde se nám nesmí ztratit žádný bit, jako je text, program a jiná data, používáme bezztrátové komprimace. Příkladem bezztrátové komprimace může být inkrementální zálohování, kde se data nezálohují celá, ale jen provedené změny od poslední zálohy. Dnes je k dispozici mnoho metod komprese dat, ať ztrátových nebo bezztrátových. Některé nejpoužívanější komprimační formáty jsou: mp3, bzip2, jpeg, zip, rar, mpeg, atd. Dnes pracujeme s komprimovanými daty, a ani o tom nevíme. Většina multimediálních dat je již komprimovaná. Ať jde o video nebo hudbu. Na druhou stranu nekomprimované formáty se již nepoužívají kromě specifických aplikací. To je dáno potřebou určitého okruhu lidí. Například hudební skladatel bude pracovat s nekomprimovanou podobou hudebního souboru (aby slyšel i ty nejmenší detaily zvuku), ale obyčejnému posluchači bude stačit komprimovaná podoba, ve které je ochuzen o tyto detaily. Nevýhodou komprimované podoby média je potřeba toto médium nejdříve dekódovat. V případě rozsáhlých dat to může trvat velice dlouhou dobu. Pokud tato rozsáhlá data chceme dekódovat v reálném čase budeme k tomu potřebovat rychlejší algoritmu nebo výkonnější stroj. Algoritmy využívající Burrows-wheelerovu transformaci(BWT) spotřebují nejvíce času v této transformaci. Proto je vhodné tuto transformaci urychlovat.

Možností urychlení je několik. Jednou možností je zvýšení taktu procesoru. Tato technika se uplatňovala v době pentium 4. Toto řešení má jednu zásadní nevýhodu. Se zvětšující se frekvencí se kvadraticky zvětšuje spotřeba elektrické energie. Proto se dnešní urychlování ubírá směrem k většímu počtu jader. Důvodem pro to je fakt, dva procesory s poloviční frekvencí spotřebují daleko méně elektrické energie než jeden procesor s dvojnásobnou frekvencí. Bohužel ne všechny aplikace jdou naprogramovat do vícevláknové podoby. Vetšinou ve vícevláknových aplikacích je nutné mít jedno vlákno, které bude koordinovat práci ostatních vláken. Problém vícevláknového programování také spočívá v režii v meziprocesorové komunikaci. Při velkém počtu vláken může meziprocesorová komunikace fatálně zpomalit algoritmus. Takovým víceprocesorovým systémem, který umožňuje současný běh několika tisíc vláken je grafická karta. Grafická karta obsahuje obrovské množství málo výkonných aritmeticko-logických jednotek, kdežto klasický procesor(CPU) obsahuje malé množství velice výkonných aritmeticko-logických jednotek.

Proto se v této bakalářské práci zabývám Burrows-Wheelerovou transformací a jejím urychlení pomocí grafické karty pomocí technologie CUDA. Cílem je značně urychlit Burrows-Wheelerovu transformaci a toto urychlení demonstrovat na vybraných testech. Druhý oddíl této práce popisuje samotnou burrows-Wheelerovu transformaci. Třetí oddíl se zabývá aplikovanými komprimačními algoritmy. Ve čtvrtém oddílu je popsána architektura grafické karty. A Nakonec je popsána implementace mého řešení.

2 Burrows-Wheelerova transformace

Burrows-Wheelerova transformace (BWT) byla zveřejněna v roce 1994 Davidem Wheelerem a Mikem Burrowsem ve článku „works in block mode“. BWT je jednou z nejvíce efektivních podpůrných metod pro kompresi dat, která byla objevena ve dvacátém století. Jedná se o metodu hojně využívanou v kompresních algoritmech, samotná metoda však žádnou kompresi neprovádí. Pouze transformuje text do lépe komprimovatelné podoby. Tato transformace pracuje v blokovém módu, pokud je na vstupu proud dat, potom se načte část tohoto proudu do paměti a zpracuje jako jeden blok. Výstup je permutace vstupních znaků, kde jsou stejné znaky s vysokou pravděpodobností výskytu řazeny blízko sebe. Tímto způsobem můžeme vylepšit kompresní poměr některých kompresních algoritmů a nemusíme měnit jejich kód. Nejvíce se tato vlastnost projevuje u adaptivních komprimačních metod, kde stejný znak na jiném místě je reprezentován jiným kódem.

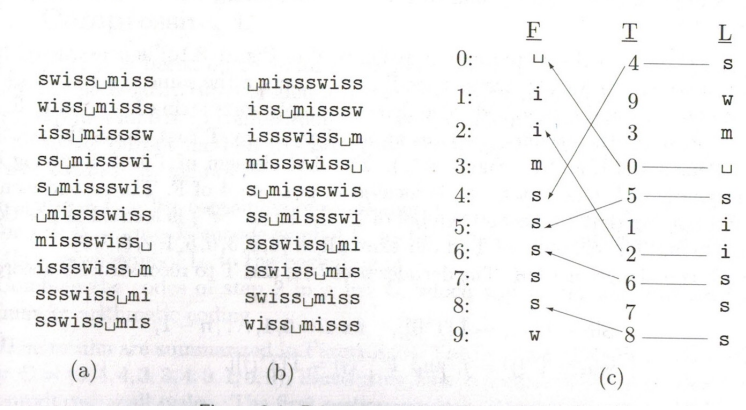
Výběr velikosti bloku BWT není vůbec jednoduchý. Pokud zvolíme příliš velký blok, algoritmus zabere příliš paměti a naopak pokud zvolíme příliš malé bloky algoritmus bude neúčinný. Měli bychom se snažit aby velikost bloku byla co největší. Ale je zde ještě jeden parametr, který ovlivňuje velikost bloku a tím je velikost vyrovnávací paměti. Jelikož přístup BWT do paměti je náhodný, je vhodné volit velikosti bloků vzhledem k těmto vyrovnávacím pamětem. Pokud bychom tak nečinili, mohlo by docházet k častému přímému přístupu do paměti, který je pomalý. Algoritmus se považuje za účinný pokud velikost bloku je více jak několik tisíc symbolů.

Kompresní algoritmy používající BWT mají dobrý kompresní poměr. Nejčastěji se BWT používá s algoritmem Move-to-front a následně se aplikují komprimační algoritmy Return-to-length a nakonec Huffmanovo kódování. BWT transformace má časovou složitost podle použitého algoritmu, ale většinou tato časová složitost je horší než časová složitost zpětné transformace. To má jednu zásadní výhodu: Kódování můžeme provádět na mnohem výkonnějším stroji a přesto nebude problém data dekódovat na ostatních strojích.

Některé algoritmy vylepšují časovou složitost algoritmu neporovnáváním celého textu, ale jenom jeho části. BWT se využívá v indexování textu, kompresi, analyzování DNA a dalších oborech. Z kompresních algoritmů BWT využívá například JPEG, mp3, Bzip2.

2.1 Transformace

Transformace se provádí vytvořením dvojrozměrného pole, kde velikosti obou rozměrů je stejná jako velikost textu. Toto pole se zaplní rotacemi původního textu, kde každý řádek je vždy o jeden znak posunutý oproti předchozímu řádku (viz obr. 2.1). Tyto rotace se setřídí a poslední sloupec v poli je požadovaný výsledek. Transformace funguje na principu nejčastějších slov v textu. Pokud předpokládáme opakování slov v textu, potom se v některé permutaci rotací nachází každé z těchto stejných slov na začátku textu. Po setřídění těchto permutací se tyto permutace se stejnými slovy nachází za sebou. Pokud tedy rozdělíme slovo „the“ hned za „t“ na začátku textu bude „he“ a na konci „t“. Po setřídění rotací textů budou všechny texty začínající na „he“ uložena blízko sebe a „t“ bude na konci tohoto textu. Tedy ve sloupci L budou stejná písmena řazena relativně za sebou, a proto se výstup z BWT dobře komprimuje.



Obr 2.1: Burrows-wheelerova transofmace [11]

na obrázku 2.1 je ukázáno jak BWT funguje nad textem S = „swiss miss“.

A. vytvoří se pole o velikosti n*n, do kterého se uloží všechny rotace vstupního textu.

B. řádky výstupu (A) se setřídí podle klíče (lexikograficky nebo podle ascii tabulky...).

C. výsledný text L je posledním sloupcem ve výstupu (B)

Transformace jen přeuspořádá znaky. Tedy písmena a jejich počet je v textu stejný jen jsou ve výstupním textu jinak uspořádány.

2.2 Zpětná transformace

Jak znovu získat z výsledného textu (L) zase původní text (S)? K tomu je třeba znát pozici originálního textu (S) v setříděném seznamu rotací (B) a výsledný text tedy sloupec (L). Jelikož sloupec (L) obsahuje stejná písmena jako první sloupec (F), získáme setříděním sloupce L sloupec F. Poté se vytvoří pole T, které reprezentuje vztah mezi sloupcem F a L. Pro příklad z obrázku pole T obsahuje (4,9,3,0,5,1,2,6,7,8) - na první pozici pole T je čtyřka, protože v L na první pozici se nachází písmeno „s“, které se v poli F nachází na čtvrté pozici. Pozice nerotovaného textu (originálního textu) slouží jako index posledního písmena. Zde je to 8. Nyní se podíváme na pozici 8 v poli L a získáme písmeno „s“, potom zjistíme předcházející písmeno, pomocí pole T. Na pozici 8 v poli T je index 7, nyní zjistíme, že na indexu 7 v poli L je také „s“ a získali jsme text „ss“ další iterací získáme písmeno „i“ a text „iss“. Zpětná transformace se ve většině případů provádí pozpátku, ovšem pro některé aplikace je toto chování nevýhodné, a proto používají o něco náročnější transformaci směrem dopředu.

Pragma:

```
L[i] = F[T[i]];
```

Rekonstrukce:

```
S[10 -1 -0] = L[8];
S[10 -1 -1] = L[ T[8]];
S[10 -1 -2] = L[ T[ T[8]]];
…
S[10 -10] = L[ T[...T[ T[8]]...]];
```

2.3 Významné datové struktury BWT

Suffixový strom a suffixové pole reprezentují lexikálně seřazené pole podřetězců daného řetězce. Tyto struktury mají mnoho využití v porovnání textu, analýze lidského genomu a textové kompresi, vyhledávání podřetězců a mnoho dalšího. Algoritmus pro konstrukci suffixového stromu byl představen v roce 1973. Jako alternativa s menší paměťovou náročností bylo v roce 1990 přestaveno suffixové pole. V roce 2003 představil Karkainen se Sendersem suffixové pole s lineární časovou složitostí. Ferragina se v roce 2005 zabýval problémem třídění textu v externí paměti.

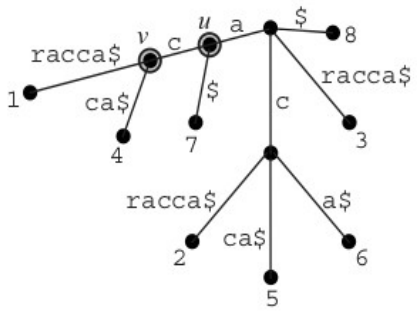
BWT bylo vynalezeno nezávisle na suffixovém stromu, ale později se zjistilo, že sufixové stromy jsou podobné struktury. Pokud k třídění rotací textu použijeme klasické třídicí metody bez znalosti suffixového pole nebo suffixového stromu, bude se časová složitost třídícícho algoritmu O(n * log n). Velkou nevýhodou tohoto přístupu jsou dlouhé opakující se sekvence znaků, které mohou časovou složitost zhoršit na O(n*n). Proto se pro BWT používají algoritmy pro suffixové stromy a suffixová pole, které zase snižují časovou složitost na O(n) s větší paměťovou náročností. Pokud tedy použijeme suffixový strom nebo suffixové pole můžeme pomocí BWT třídit až několik megabitů v celkem rozumném čase. Časově nejnáročnější částí BWT je třídění textu, proto se pro tento účel začaly využívat suffixové stromy nebo pole. Na konec textu se vloží znak, který není v použité abecedě a zároveň je nejmenším možným znakem. Tím se problém třídění textu v BWT změní na suffixový strom nebo suffixové pole. Trend v tomto odvětví je využít co nejméně paměti jak je možné, aby mohly být využívány větší bloky dat. Bzip2 používá algoritmus Larson-Sadakane, který má časovou složitost O(n * log(n)), pokud tento algoritmus bude příliš pomalý Bzip2 použije variantu fallbacksortu. Existuje spousta algoritmů vytváření suffixového stromu a suffixového pole s různou časovou složitostí. Několik algoritmů pro suffixové stromy a pole jsou uvedeny v tabulce 2.1.

Vztah BWT a suffixového pole :

BWT[i] = T[SA[i] – 1], pro i = 0..n a SA[i] != 0
BWT[i] = T[n –1], pro SA[i] == 0
kde T je vstupní text, SA je suffix array, n je velikost textu T

2.3.1 Suffixový strom

Tato struktura má tvar obecného stromu. Jedna z knih zabývajících se suffixovými stromy je (Gusfield, 1997) „provides a comprehensive treatment of suffix trees, their construction, and applications“. Paměťová i časová složitost konstrukce suffixového stromu je lineární. Ze sufixového stromu lze vytvořit sufixové pole.



Obr. 2.2: suffixový strom. [6]

2.3.2 Suffixové pole

Suffixové pole je struktura velice podobná suffixovému stromu. Suffixové pole obsahuje lexikograficky seřazené ukazatele do textu. Tedy pokud na pozici „i“ se nachází element „j“, potom se v textu nachází „i“ podřetězců předcházející podřetězec „j“. Ve většině problémů kde se využívá suffixový strom, lze využít suffixové pole, ale neplatí to pro všechny případy. Hlavní motivací použití suffixového pole místo suffixového stomu je nižší paměťová složitost.

Některé algoritmy a jejich porovnání podle … (převzato z) jednalo se o 200MB soubory

algoritmus	angličtina(s)	DNA (s)	Zdrojové kódy ()	RAM (MB)
K	244	246	293	481
NZC	58	59	41	1000
FGM	405	421	324	481

Tabulka 2.1: některé algoritmy pro konstrukci suffixového pole. Jednalo se o 200MB soubory [5]

2.3.3 Fast lightweight Suffix array construction

Tento algorimtus je popsán v publikacích fast lighweight suffix array cosntruction od Juha Karkkainenem a Stefanem Burhardtem [3]. Základní myšlenkou tohoto algoritmu je setřídění seznamu vybraných podřetězců, a využití těchto podřetězců pomocí matematické funkce difference covers. Jednoduchá metoda pro vytváření difference covers je popsána Colbounem a Lingem. Pro takové tříděni platí: Pro každé dva podřetězce (i,j) existují dva již seřazené podřetězce (i+k, j+k), kde „k“ je menší než „v“. Pokud podřetězce „i“ a „j“ setřídíme do hloubky „v“, můžeme pomocí podřetězců „i + k“ a „j + k“ určit, který z podřetězců (i,j) je menší. Matematická struktura different cover slouží k tomu aby se tento stav odehrál vždy do určitého maximálního počtu porovnání , které je menší než vybrané „v“. Vybrané vzorky se setřídí a vytvoří se invertované ISA pole, kde index do tohoto pole znamená začátek podřetězce a číslo v tomto poli reprezentuje pozici podřetězce v seřazeném poli. Takovéto pole by muselo mít velikost 4n a přitom počet tříděných podřetězců je mnohem menší, z těchto důvodů se používá přepočtová funkce. Nakonec se setřídí všechny podřetězce do maximální hloubky „v“. Pokud není možné rozhodnout, který z podřetězců je lexikograficky větší, je využito ISA pole. Tento algortmus je nenáročný na dodatečnou paměť a zároveň také rychlý. Algoritmus má časovou složitost O(n * log(n)) a potřebuje navíc O(N/sqrt(n)) paměti. Pokud bychom chtěli využít nízkou paměťovou náročnost tohoto algoritmu, museli bychom použít algoritmus Longest Common Prefix (LCP). Nízká paměťová náročnost je dána počítáním BWT jen pro část zadaného textu v aktuálním čase. Pomocí LPC se vyberou podřetězce, které patří do aktuálně řazeného bloku. Tedy se nacházejí mezi dvěma zvolenými podřetězci. Tyto podřetězce se seřadí a vytvoří se část BWT, které se již může zapsat na úložiště nejčastěji na pevný disk.

3 Komprese založené na BWT

Účelem komprimačních algoritmů je zmenšit velkost vstupu změnou reprezentace dat. Poměr mezi velikostí vstupu a výstupu se nazývá komprimační poměr. Tento komprimační poměr v podstatě udává efektivitu kompresních metod. U některých komprimačních algoritmů se před aplikací samotného komprimačního algoritmu použijí jiné algoritmy jako je BWT nebo Move-to-front, které upraví vstup do lépe komprimovatelné podoby, tedy zvýší kompresní poměr.

Pro mnoho kompresních algoritmů existují dvě varianty adaptivní a neadaptivní. Neadaptivní algoritmus projde celý vstup a vytvoří statickou tabulku kódování jednotlivých znaků, kdežto adaptivní kódování pracuje proudově. Tabulku upravuje v průběhu kódování, proto může mít jeden znak na výstupu různý kód na různých pozicích.

3.1 Move to front (MTF)

Tato transformace není komprimační metodou, ale metodou podpůrnou, která přesune nejčastěji používané znaky na začátek abecedy za účelem zvýšení kompresního poměru. Transformace pracuje proudově, proto je nutné data kódovat a dekódovat ve stejném pořadí. Transformace funguje takto:

1) Vytvoří se seznam znaků ze zadané abecedy, tento seznam musí být pro transofrmaci i pro zpětnou transformaci předem známý.

2) Načte se ze vstupu znak.

3) Prohledá se seznam, a na výstup se vypíše pozice kde byl znak nalezen.

4) List s tímto znakem je přesunut na začátek seznamu. Pokud následující znak bude stejný jako aktuální bude tento znak v kroku 3 nalezen na první pozici v seznamu

5) Dokud jsou na vstupu data skoč na krok 2.

Z Algoritmu vyplývá, že pozice jednotlivých znaků jsou určeny počtem typů znaků, které byly přečteny ze vstupu od posledního přečtení aktuálního znaku. Nevýhodou tohoto algoritmu je, pokud se na vstupu nenachází žádné identické znaky, potom nedojde ke snížení četnosti některých znaků s vyšší četností znaků. Algoritmus sice může zvýšit počet různých znaků na výstupu, ale výstup má nesporně menší směrodatnou odchylku než vstup, data s nízkou směrodatnou odchylkou jsou velice vhodná pro komprimační algoritmy. Existuje několik modifikací algoritmu Move-to-front. Jednou z variant je varianta pracující s celými slovy. Další zase nepřesouvá znak z abecedy dokud těchto znaků nebylo za sebou určité množství. Příkladem MTF mějme text (aaabbcaa) a abecedu (a,b,c). Výstupem tohoto textu bude sekvence čísel (00010220) neboli ve znacích (aaabacca). Z tohoto příkladu to není příliš patrné, ale algoritmus změní sekvenci stejných znaků na sekvenci nul, kromě prvního znaku z této sekvence.

Zpětná transformace se provádí jednoduše. Hodnota ze vstupu se použije jako index do seznamu. Na zadaném indexu se nachází odpovídající znak. Tento znak je vypsán na výstup a v seznamu je přesunut na začátek.

Časová složitost transformace i zpětné transformace je lineární. Paměťová složitost je konstantní. Dokonce lze provést na místě, tedy přepisovat původní data.

3.2 Run length encoding (RLE)

Je to nejjednodušší metoda komprese. Jedná se o proudovou kompresi dat, kde výsledný komprimační poměr silně závisí na počtu a velikosti opakujících se sekvencí stejných znaků. Základní myšlenkou je: možnost zapsat opakující se sekvenci stejného znaku jako jeden znak a počet opakování tohoto znaku. Pokud se ve vstupu „*d*“ krát opakuje vstupní znak „*n*“ potom na výstup můžeme zapsat sekvenci jako „*dn*“, kde číslo „*d*“ určí počet opakování znaku „*n*“. Tento způsob komprese má dvě hlavní nevýhody. Není způsob jak říct, že „*d*“ není součástí dat, ale udává počet opakování, proto některé algoritmy využívají speciální znak, který rozhodne o tom jestli následující znak je znak nebo číslo určující počet opakování. Druhou nevýhodou je nedostatečný počet opakování znaků v obyčejném textu. To může způsobit, že komprimovaná data budou větší než nekomprimovaná, proto některé algoritmy RLE začínají komprimovat až od sekvence tří a více stejných znaků. V metodě RLE může být výstup v nejhorším případě stejně dlouhý jako vstup. Z těchto důvodů nemusí být tato komprese pro obyčejný text vůbec výhodná. Naopak je výhodné používat tuto kompresi tam, kde jsou dlouhé sekvence stejných znaků, jako je tomu například ve výstupu BWT.

Dekomprese je velice jednoduchá. Algoritmus přepisuje vstup na výstup, dokud nenarazí na sekvenci označující počet opakování. Ze vstupu se načte počet opakování a znak, který se opakuje. Dekomprimační algoritmus vypíše na výstup několik těchto znaků, podle počtu opakování.

Časová složitost tohoto algoritmu je lineární. Požadovaná paměť je pouze pro výstupní pole.

3.3 Huffmanovo kódování

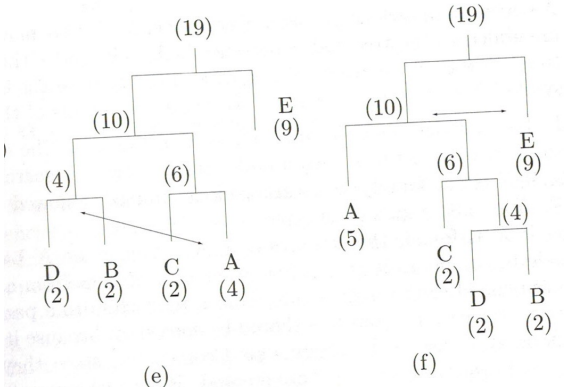
Tato metoda byla vynalezena D. Huffmanem v roce 1952. Je to hojně používaná blokově orientovaná kompresní metoda. Jedná se o algoritmus s proměnou délkou kódu. Základní myšlenkou celého algoritmu je překódovat vstupní abecedu tak aby nejfrekventovanější znaky zabíraly méně bitů. Podobnou myšlenku také používá kódování UTF-8, kdy jsou méně frekventované znaky zakódovány jako posloupnost několika bajtů. Komprese začíná vytvořením histogramu abecedy v zadaném textu. Potom se podle četnosti každého znaku vytvoří kódové sekvence pro každý znak ve vstupu. Pokud se ve vstupním textu nachází dostatečný počet různých znaků, mohou mít některé málopočetné znaky delší zakódování jako jeden bajt. Dekompresor potřebuje zakódovaný text a informaci kódu jednotlivých znaků. Dekomprese se provádí skenováním bit po bitu a prochází se stromem, který byl vytvořen kompresním algoritmem, až se dostane na konec. Potom se vypíše písmeno na konci tohoto stromu a začne se zase od začátku stromu. Pro ještě lepší zakódování lze použít aritmetické kódování, které se více blíží ideální velikosti znaku log2(p), kde p je pravděpodobnost výskytu znaku v textu. Aritmetické kódování je mnohem pomalejší než Huffmanovo kódování, proto se pro některé aplikace aritmetické kódování nehodí. Aritmetické kódování může více znaků zakódovat do jedné posloupnosti bitů.

Adaptivní Huffmanovo kódování

Adaptivní Huffmanovo kódování není blokově orientovaná metoda. V této metodě není potřeba znát četnost jednotlivých znaků předem. Algoritmus funguje jednoduše. Nejdříve se vytvoří strom s

jednou hodnotou NULL. Pokud na vstupu se bude nacházet znak, který se ještě nenachází v kódovacím stromě, bude mu na výstupu předcházet hodnota NULL. Následně se znak umístí do kódovacího stromu. Velikost kódu a kód se u jednoho znaku v průběhu komprese mění podle aktuální četnosti znaku na vstupu. Tato dynamičnost může vytvářet lepší výsledky komprese.

Při dekompresi se také strom inicializuje hodnotou NULL pokud se v textu objeví tato hodnota dekompresor ví, že následující hodnota je hodnota znaku, který je nutné přidat do kódovacího stromu. Tato metoda pracuje v proudovém módu a není potřeba znát před dekompresí zakódování jednotlivých znaků. Časová složitost toto algoritmu je konstantní.

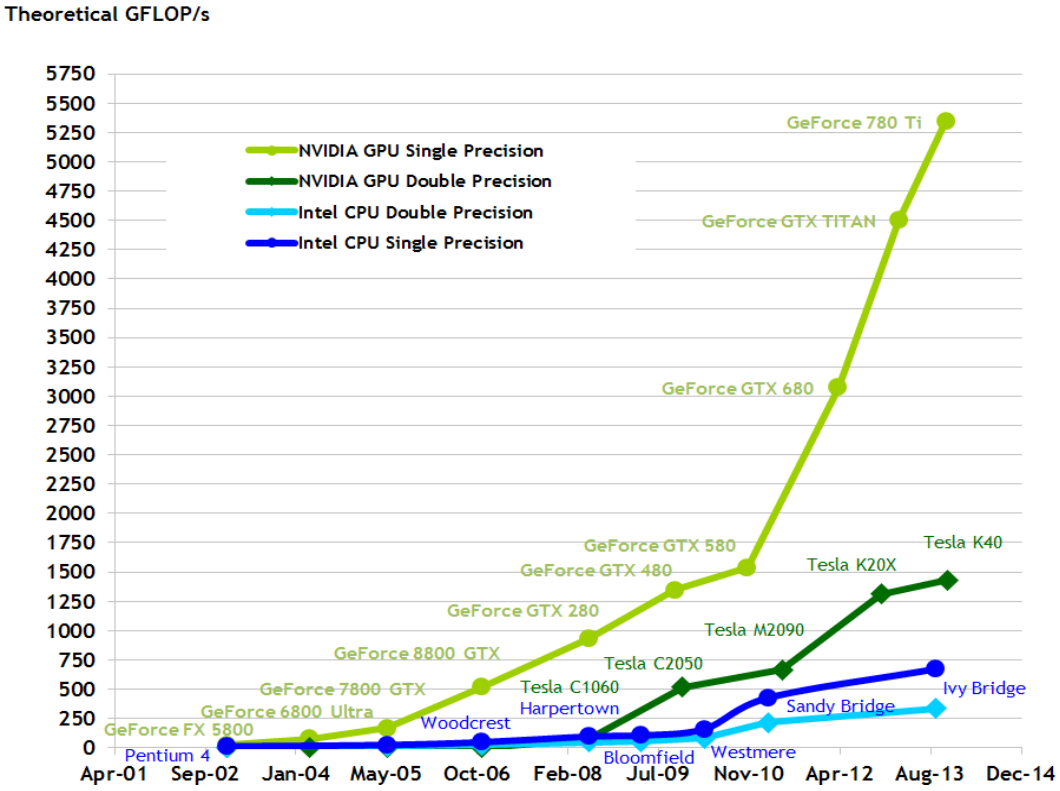


Obr 3.1: změna kódovacího stromu adaptivního Huffmanova kódování. [11]

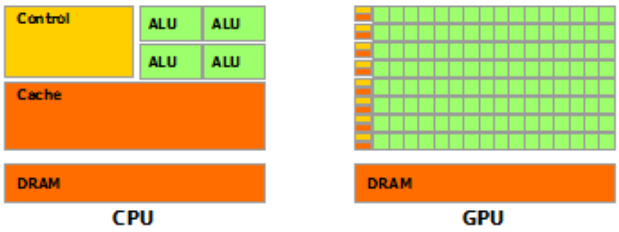
4 Akcelerace výpočtu s využitím GPU platformy

Původním úkolem grafických karet bylo zobrazovat grafická data na monitor. Pro tuto práci je potřeba vysoký výpočetní výkon, který se dosahuje masivním paralelizmem. Po určitém čase se objevila myšlenka využít tohoto paralelizmu grafických karet pro některé náročné výpočty.

V roce 2006. představila NVIDIA svoji technologii CUDA pro paralelní programování na grafických kartách NVIDIA. Compute Unified Device Architecture (CUDA) je masivně paralelní architektura s vysokou propustností vnitřních pamětí. Jak je vidět na obrázku 4.1 i s nižší frekvencí jádra dokáže grafická karta, díky masivnímu paralelizmu, provést několikanásobně více aritmetickologických operací než CPU. Samotná grafická karta je navrhována pro programy vykonávající stejný algoritmus pro obrovské množství dat. Myšleno jako vykonávání vždy stejné instrukce nad všemi daty. To je výhodné u některých algoritmů jako jsou simulace, maticové operace, zpracování obrazu atd. Tedy pro algoritmy s vysokou možností pralelizace, programy kde je možné pustit několik tisíc až několik desítek tisíc vláken. Druhý obrázek 4.2 ukazuje rozdílné uspořádání aritmetickologických jednotek pro grafickou kartu a pro CPU. Důvodem proč je možné dosáhnout na grafické kartě daleko více aritmetickologických operací je snížení počtu řídících jednotek na úkor aritmetickologickým jednotkám, jak je vidět na obrázku 4.3. Tento přístup má nevýhodu v omezení programu na stejný kód pro velký počet aritmetickologických jednotek.



Obr 4.1: Operace v řádové desetinné čárce za sekundu pro procesor a grafickou kartu [13]



Obr 4.2: Rozdíl v ALU jednotek v procesoru a grafické kartě [13]

V dnešní době každá grafická karta od nvidie podporuje CUDA, tedy všechny novější grafické karty jako GEFORCE 8800GTX vyrobené v roce 2006. Seznam všech grafických karet podporujících CUDA a verze jejich architektur je na stránkách www.nvidia.com/CUDA. Zde je ke stažení i samotný toolkit nezbytný pro programování CUDA aplikací.

4.1 Architektura grafické karty

GPU se skládá z globální paměti a multiprocesoru. Globální pamětí je myšlena paměť fyzicky umístěna na grafické kartě. Multiprocesor se skládá ze streamprocesorů, lokální paměti a registrů. Tyto registry jsou potom přidělovány jednotlivým vláknům. Rychlost pamětí je různá. Nejrychlejší pamětí jsou registry. Druhou nejrychlejší pamětí je sdílená paměť na multiprocesoru, jelikož je blíže streamprocesorům než globální paměť. Sdílenou paměť mohou pouze využívat vlákna, která běží na multiprocesoru. Jako třetí nejrychlejší pamětí je globální paměť umístěná na grafické kartě. Nejpomalejší pamětí je paměť hosta, ke které se přistupuje pomocí PCI-E sběrnice. Streamprocesory vykonávají program jednotlivých vláken. Různé typy grafických karet obsahují různý počet multiprocesorů a různé verze architektur těchto multiprocesů. Na jedné grafické kartě jsou vždy multiprocesory se stejnou verzí architektury. Liší se jenom počet multiprocesorů, podle toho o jak výkonnou grafikou kartu se jedná. Sdílená paměť je společná vždy pro jeden blok. Dnes existují tyto architektury multiprocesoru 1.0, 1.1, 2.0, 2.1, 3.0 a 3.5. Nedávno přibyla nová architektura 5.0.

4.2 Programovací model

Pro překlad je potřeba mít nainstalován „CUDA toolkit“, „gcc“ a „g++“. Tedy pro překlad kódu je zapotřebí mít dva kompilátory. Jeden pro překlad kódu pro grafickou kartu , tedy v našem případě „nvcc“, a jeden pro překlad kódu pro CPU (hosta). Zatímco pro spuštění přeloženého kódu je potřeba mít pouze nainstalován ovladač od nvidie a CUDA toolkit. CUDA toolkit podporuje vývoj v jazycích C/C++, Fortran, java, python atd. CUDA je dostupná na architekturách linux, OS X a microsoft windows. Jelikož grafická karta a host (CPU) pracuje asynchronně, je nutné explicitní synchronizace.

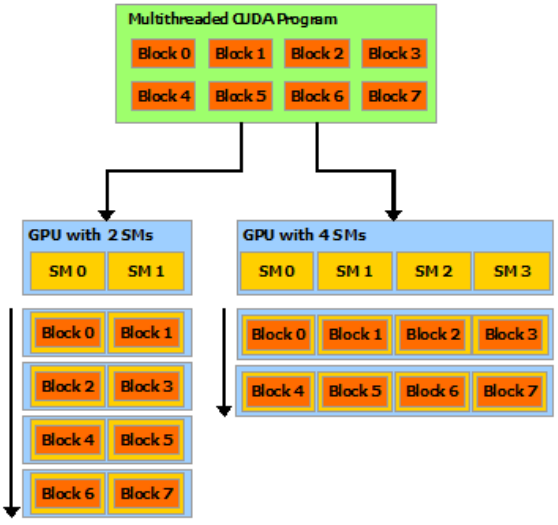
V CUDA se programuje pomocí bloků a vláken, kde několik vláken patří jednomu bloku. Bloky jsou omezeny v synchronizaci. Bloky lze pouze synchronizovat pomocí spin-locku a atomických instrukcí v globální paměti, které jsou k dispozici od verze multiprocesoru 1.1. Potom je

možné přistupovat ke společným globálním proměnným, ale není možné pomocí spin locku čekat na ostatní bloky jelikož některé bloky nebudou spuštěny dokud se jiný blok neukončí. Pokud tyto bloky budou čekat na ještě nespuštěné bloky nikdy neuvolní multiprocesor a dojde k uváznutí (deathlock). Kdežto vlákna z jednoho bloku je možné synchronizovat pomocí instrukce „*__syncthreads()*“. Tato instrukce způsobí čekání, dokud ji ostatní vlákna v bloku také nevykonají. Existuje ještě jedna možnost synchronizace a tou je warp. Warp je popsán v hardwarové implementaci. Plánovač, který plánuje vykonávaní kódu pro warp při rozvětvení kódu nejdříve provede jednu větev, následně druhou větev programu, a na konci obou větví se opět warp sjednotí. Multiprocesor dokáže zpracovávat zároveň jeden a více bloků podle své kapacity. Grafické karty nepodporují multitasking, proto blok, který se začne vykonávat na jednom multiprocesoru také skončí na tomto multiprocesoru. Některé grafické karty umožňují konkurentní vykonávání dvou kernelů. Kód pro grafickou kartu se ve většině literatury nazývá kernel, a proto i já níže budu tento kód nazývat kernelem. Jelikož grafická karta je asynchronní vůči hostovi, může host zadat práci grafické kartě a následně mohou obě platformy pokračovat ve výpočtech samostatně.

Přenosy dat přes jakoukoliv sběrnici jsou časově náročné, proto existuje pár jednoduchých pravidel pro programování periferií.

1. Přeneseme data do periferie.
2. Vytvoříme dostatek práce pro periferii.
3. Získáme dokončený výpočet.

Pokud chceme programovat efektivně pro grafické karty, musíme hostem rozdělit problém na podproblémy, které mohou být paralelně zpracované na grafické kartě. Tyto podproblémy necháme řešit bloky, které se postupně mapují do multiprocesorů umístěných na grafické kartě. Pokud grafická karta nemá dostatečný počet volných prostředků pro spuštění bloku, nezačne se tento blok vykonávat, dokud se neukončí jiný blok kódu, který uvolní dostatek prostředků. Při mapování bloků na multiprocesor nezávisí na počtu multiprocesorů ani na jejich rozložení na grafické kartě viz obr. 4.3. Pokud existuje dostatek zdrojů na multiprocesoru, dojde ke spuštění dalšího bloku. Přitom každý blok může být spuštěn na jakémkoliv volném multiprocesoru. Tato vlastnost umožňuje spouštět kernel na grafické kartě s různým počtem a uspořádáním multiprocesorů. Na jednom multiprocesoru může být vykonáváno více bloků.



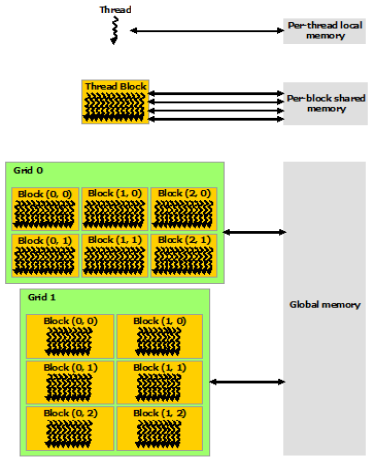
Obr. 4.3: Mapování výpočetních bloků pro grafické karty s různým počtem multiprocesorů. [13]

CUDA dále přidává do jazyka C další syntaxi, jako je odlišení sdílené a globální paměti nebo pro odlišení kódu běžícího na grafické kartě a hostu. Seznam těchto změn se nachází na internetových stránkách nvidie. Kód obsahující rozšíření CUDA nemůže být přeložen jiným kompilátorem než NVCC. Do kernelu lze přidat inline assembler, tedy jen CUDA instrukce dostupné pro danou architekturu multiprocesoru. Více o inline instrukcích se nachází na webových stránkácha nvidie.

Důležitou technikou při programování kernelu ke snížení náročnosti na přenosové pásmo je využití vyrovnávacích pamětí a sdruženého přístupu do globální paměti. Různé možnosti sdruženého přístupu jsou zobrazeného na obrázku 4.4. Pokud má dojít ke sdruženému přístupu do globální paměti je nutné dodržet podmínky: První podmínkou je přístup k proměnné o minimální velikosti 4 bajty (int) nebo k násobkům této velikosti. Druhou podmínkou je přístup Warpu k proměnným umístěných v paměti za sebou. Grafické karty obsahují dvě úrovně vyrovnávacích pamětí L1 a L2. L1 vyrovnávací paměť se nacházejí přímo na multiprocesoru a jejich velikost je nastavitelná na 16kB nebo 48kB. Pokud nechceme používat vyrovnávací paměti, překladač „*nvcc*“ obsahuje přepínač, kterým vypne využívání vyrovnávacích pamětí, nebo pokud chceme vypnout vyrovnávací paměti jen pro jednu proměnou stačí při zápisu do této proměnné používat patřičnou instrukci.

4.3 Hierarchie pamětí

Existuje několik druhů pamětí fyzických a logických. Tyto paměti jsou rozděleny podle toho kde se nachází a pro co jsou určeny. Fyzické paměti jsou: sdílená paměť, globální paměť a paměť hosta (CPU, RAM paměť). V těchto pamětech jsou uloženy paměti logické, jako je texturovací, globální, konstantní, sdílená a stránkově uzamčená paměť (angl. Page-locket). K pamětem, které se nacházejí v globální paměti nebo v paměti hosta, může fyzicky přistupovat každé vlákno z jakéhokoliv bloku. K paměti na multiprocesoru mohou fyzicky přistupovat jenom vlákna nacházející se na tomto multiprocesoru. Logická lokální paměť se nachází ve fyzické globální paměti a každé vlákno má svou vlastní. Jiné vlákno nemůže k této paměti přistupovat. Logická sdílená paměť se fyzicky nachází v multiprocesoru a každý blok má svou vlastní. K této paměti mohou přistupovat pouze vlákna ze stejného bloku. Jiná vlákna běžící v jiném bloku na témže multiprocesoru, nemohou k této paměti přistupovat. Logická texturová a constatntní paměť se nachází ve fyzické globální paměti a jejich určení je trochu rozdílné. Nachází se zde konstanty a kernel nesmí do těchto pamětí zapisovat. Proto může multiprocesor drasticky využívat vyrovnávací paměť.



Obr. 4.4: Hierarchie pamětí [13]

4.3.1 Sdílená paměť

Sdílená paměť je mnohem rychlejší než globální paměť. Proto se její část využívá pro vyrovnávací paměti L1. I z tohoto důvodu je možné přepínat mezi velikostmi sdílené paměti a L1 vyrovnávací paměti podle preference. Kdy je možné mít 16kB vyrovnávací paměť a 48kB sdílenou paměť nebo naopak 16kB sdílenou a 48kB vyrovnávací (je udáno pro architektu 2.x). Jelikož na jednom multiprocesoru může být vykonáváno více bloků jsou tyto hodnoty udány pro jeden multiprocesor nikoliv pro jeden blok. Hlavním důvodem rychlosti těchto pamětí je jejich konstrukce a jejich umístění na multiprocesoru. Tedy hardwarově blíže než globální paměť.

4.3.2 Texturovací paměť a paměť pro konstanty

Na grafické kartě se nachází velké množství aritmetickologických jednotek, jelikož každá jednotka potřebuje přísun dat, může se stát přenos dat mezi globální pamětí a aritmetickologickými jednotkami úzkým místem, které snižuje podstatně výkon. Proto grafické karty obsahují paměti, které za určitých podmínek mnohonásobně urychlují přístup do globální paměti (paměť konstant a texturovací paměť). Tento druh paměti se sice nachází v globální paměti, ale silně využívá svých vlastních vyrovnávacích pamětí, které jsou odděleny od ostatních vyrovnávacích pamětí. Tyto logické paměti mají svoji vyrovnávací paměť na multiprocesoru.

Paměť konstant nemůže být jakkoliv dynamicky alokována, a její hodnoty přetrvají celý běh kernelu. Maximální velikost konstantní paměti je 64kB. Naproti tomu texturovací paměti mohou být dynamicky alokovány hostem před spuštěním kernelu. Ačkoliv jsou texturovací paměti navrženy pro klasické použití v openGL a DirectX renderování, má texturovací paměť výhody, které lze uplatnit i v obecných výpočtech prováděných na grafické kartě. Pro využití texturovací paměti se předpokládá využívání dat blízko sebe, které vlákna budou číst v co nejkratší době po sobě. Tedy pokud vlákna čtou nahodile po paměti, jako tomu je v algoritmu BWT, může texturovací paměť ještě operace prodlužovat. Proto je nutné využívat tuto paměť opatrně.

4.3.3 Stránkově uzamčená paměť

Stránkově uzamčená paměť je paměť hosta, tedy nejčastěji paměť RAM, kterou vyžívá CPU. Tato paměť může být rovnou alokována pomocí CUDA toolkitu nebo pomocí standartní funkce pro alokaci paměti a následně tuto paměť můžeme registrovat jako stránkově uzamčenou. Tedy systém nemůže tuto paměť odložit na disk nebo jinam přesunout. Stránkově uzamčená paměť se nejčastěji používá pro komunikaci s periferními zařízeními, kdy se přes tuto paměť pomocí stanoveného protokolu předávají data. Některým zařízením může být tato paměť namapována do adresového prostoru a jejich fyzická operační paměť může být tímto způsobem rozšířena. Pokud alokujeme příliš stránkově uzamčené paměti, může se tím snížit celkový výkon hosta. Kopírování pomocí stránkově uzamčené paměti můžeme urychlit kopírování mezi pamětí hosta a pamětí grafické karty. Toto kopírování může být také prováděno konkurentně vzhledem ke kódu běžícímu na CPU i vzhledem ke kódu běžícímu na grafické kartě.

Zápisem-kombinovaná paměť pro stránkově uzamčenou paměť CPU implicitně využívá vyrovnávací paměti. Pokud bychom zapsali do této paměti nějaká data, nemusel by je kernel správně přečíst, jelikož by tyto data byla zapsaná jenom ve vyrovnávacích pamětech hosta, a do hlavní paměti RAM by se nezapsaly. Proto je nutné pro tuto část adresového prostoru vypnout využívání

vyrovnávacích pamětí. Potom se výsledky vypočítané hostem okamžitě uloží do paměti. Tato vlastnost má jednu zásadní nevýhodu a tou je pomalé čtení z této paměti. Proto je vhodné do této paměti jenom zapisovat. Použitím této paměti pro přenos mezi hostem a grafickou kartou můžeme tento přenos urychlit až o 40%.

Mapovaná paměť od grafické karty s verzí multiprocesoru 1.1 může být stránkově uzamčená paměť, mapována do grafické karty. Pokud zápisem-kombinovanou paměť namapujeme do zařízení grafické karty, odpadá nutnost explicitně kopírovat data z pamětí do grafické karty, ale grafická karta může přímo pracovat s touto pamětí. Nevýhodou tohoto přístupu je malá rychlost PCI-E sběrnice, pomocí které se přistupuje do globální paměti. Jelikož s touto pamětí může pracovat více zařízení, nelze operace provedené atomickými instrukcemi v této paměti považovat za atomické. Pokud vytvoříme takovou paměť, potřebujeme dva ukazatele: jeden pro hosta a jeden pro grafickou kartu. Tyto ukazatele se mohou lišit, jelikož klasické PC používá stránkování.

4.4 Kompilace

Existují dvě možnosti kompilace. Jedna při překladu kódu, druhá při spuštění kernelu. U kompilace při překladu se zdrojový kód přeloží přímo do strojového kódu grafické karty pro danou verzi multiprocesoru. Přitom kód pro různé verze multiprocesorů nemusí být kompatibilní. U kompilace při překladu se zdrojový kód přeloží do mezikódu PTX a následně při každém spuštění se mezikód přeloží do strojového kódu aktuální grafické karty. Překlad nemusí probíhat vždycky, neboť ovladač grafické karty výsledný strojový kód ukládá pro příští použití. Výhodou PTX kódu je využití všech novinek při aktualizaci ovladače, lepší kompatibilita mezi verzemi multiprocesorů. PTX kód lze vždy spustit na novějším hardware, kdežto spuštění přeloženého kódu při překladu lze pouze na verzích multiprocesoru se stejným nebo vyšším minoritním číslem verze multiprocesoru. Tedy kód přeložený při překladu pro verzi X.y lze spustit na verzi X.z kde platí y < z. Jelikož u kompilace při překladu není nutný sekundární překlad při spuštění kernelu na grafické kartě, je tato možnost rychlejší při prvním spuštění kernelu. Při překladu je možné vygenerovat kód pro několik verzí multiprocesoru. Například můžeme vytvořit kernel pro několik verzí multiprocesoru kompilovaný při překladu a několik verzí PTX kódu pro vyšší verze multiprocesoru. Pokud máme více grafických karet s různou verzí multiprocesoru, bude se nám tato vlastnost hodit. Když se budeme rozhodovat na které grafické kartě kernel poběží těsně před jeho spuštěním. Překlad má omezení pro překlad kódu pro různou datovou šířku, je-li kód pro hosta (CPU) přeložen v 64-bitovém módu musí být kód pro grafickou kartu také přeložen v 64-bitovém módu. To samé platí o 32-bitovém módu, pro ten musí být také kódy jak pro hosta tak pro grafickou kartu přeloženy v 32-bitovém módu.

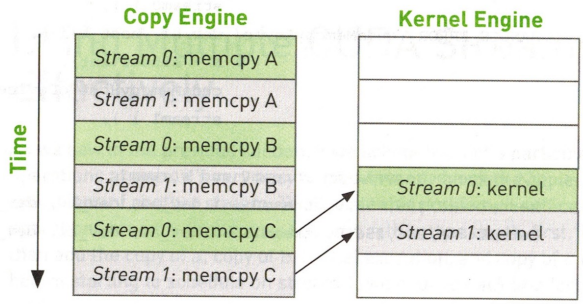
4.5 Konkurentní provádění kódu

Konkurentně se může vykonávat:

+ vykonávání kernelu na grafické kartě od programu na běžícím na hostovi (CPU).
+ kopírování paměťového bloku 64 kB a menších.
+ kopírování pamětí pomocí funkcí s příponou „asinc“
+ některé grafické karty s verzí multiprocesoru 2.x a vyšší mohou vykonávat více kernelů konkurentně

4.5.1 Stream

Stream umožňuje kopírovat data zcela nezávisle na práci grafické karty i hosta. Pokud se stream dobře použije, může urychlit celou aplikaci až o několik procent. To ale závislí na tom, kolik procent z celkového výkonu zabere kopírování potřebných dat. Pro urychlení aplikace je potřeba vytvořit minimálně dva streamy a dva kernely. Zatímco jeden kernel se vykonává, data pro druhý se kopírují do paměti grafické karty. Na obrázku X.y je vidět správné použití streamu. Pokud je kopírování oproti samotnému výpočtu velice drahé, je tato metoda tou správnou k urychlení aplikace. Jelikož některé funkce jsou implicitně synchronizovány, je nutné dávat pozor při programování na jejich uspořádání. Ke streamu je možné přidat také takzvanou callback funkci, která se vykoná po ukončení kopírování z paměti grafické karty do paměti hosta.



Obr. 4.5: Správně použité dva streamy pro dva stejné kernely. [13]

4.6 Systém s více grafickými kartami

Některé systémy obsahují i více grafických karet než jednu. Těmto grafickým kartám je možné zadávat různé kernely a také mohou spolupracovat na vyřešení stejného problému. Také je možné pomocí sběrnice PCI-E přistupovat k paměti ostatních grafických karet. Tato metoda se nazývá rovný s rovným paměťový přístup (angl. peer-to-peer memory acces). Pokud je aplikace 64-bitová, může být využit jeden adresový prostor pro všechny grafické karty. Tato paměť je přístupná přes PCI-E sběrnici. Pokud bude více zařízení připojených pomocí PCI-E sběrnice bude přes tuto sběrnici komunikovat. Zařízení mohou zahltit sběrnici požadavky pro přenos dat. Tím se může komunikace s externí pamětí tak zpomalit, že algoritmus stráví většinu času čekáním na data. Což má za následek neúnosné zpomalení aplikace i paradoxní možnost rychlejšího běhu aplikace na méně externích zařízení.

4.7 Warp a multiprocesor

Multiprocesory jsou navrženy jako architektura SIMT (single isntruction multiple thread). Tato architektura je velice podobná architektuře SIMD (single instruction multiple data). Rozdíl spočívá ve větší volnosti programátora, jelikož každé vlákno nemusí vykonávat stejnou instrukci.

Vlákna se sdružují do warpu tedy do 32 vláken následujících za sebou. Warp vždy začíná vláknem s identifikačním číslem dělitelným 32. Maximální efektivity je docíleno, když všech 32 vláken ve warpu vykonávají jednu instrukci a tedy nedochází k divergenci. Důvodem je vykonávání pouze jedné instrukce warpem v aktuálním čase. Pokud tedy dojde k divergenci musí se mezi těmito

vlákny přepínat. Warp, který je před rozvětvením programu, vykoná nejdříve jednu větev programu a potom druhou větev program a na konci rozvětvení bude opět warp vykonávat jednu instrukci pro všech 32 vláken. Tím je docíleno synchronizace, a tuto synchronizaci můžeme využít v programu. Pokud warp zapisuje neatomickou instrukcí do stejné proměnné je chování definováno architekturou, na které je kernel spuštěn (různé architektury mohou dávat různé výsledky). Multiprocesory jsou také ochuzeny o predikci skoku. Predikce skoku nemusí být v této architektuře nutná, pokud bude jeden streamprocesor vykonávat jeden warp jedno vlákno po druhém pro všech 32 vláken. Může tak být skok pro první vlákno rozhodnut ještě před dokončením instrukce pro poslední vlákno z warpu.

4.8 Možnosti optimalizace

Jednou možností optimalizace je optimalizace přístupu do pamětí. Aplikace by měla co nejvíce dat přesouvat k pamětem umístěným co nejblíže streamprocoseorům v multiprocesoru. Existují možnosti asynchronní komunikace mezi hostem a grafickou kartu nezávisle na provádění kódu kernelu nebo kódu hosta. Pro architektury 2.x a vyšší je možné spustit kód kernelů konkurentně.

Druhou možností optimalizace je optimalizace proudu instrukcí, minimalizování instrukcí s malou propustností, snížením divergence warpu na minimum a minimalizováním instrukce pro synchronizování. Jednou z možností jak toho dosáhnout odstranění synchronizačních instrukcí je využití implicitní synchronizace warpu.

5 Implementační detaily

Výsledný program pro CPU i pro grafickou kartu je naprogramován v jazyce C s rozšířenou syntaxí toolkitu CUDA. Pro implementaci třídění v BWT jsem se rozhodl použít suffixové pole, které je jednou z takzvaných lightweight metod. Tyto metody se snaží využívat co nejméně paměti. Lightweight metodu jsem vybral z důvodu velkého počtu vláken na grafické kartě, pokud používáme tolik vláken stává se paměťová náročnost velice závažnou. Tedy pokud budeme mít 1000 vláken, každé z těchto vláken bude zpracovávat data o velikosti 20kB. Pokud algoritmus bude potřebovat pro vykonání desetinásobek paměti oproti vstupním datům. Zaberou všechny struktury v globální paměti grafické karty zhruba 200MB. Což je obrovské množství paměti, a proto je nutné dbát na co nejmenší velikost všech datových struktur.

Program komprimuje text pomocí algoritmů BWT, MTF, RLE a Huffmanova kódování. Nejnáročnější částí celého programu je třídění rotací textu v algoritmu BWT, které spotřebuje asi 70% celkového času programu. Proto jsem hlavně urychloval tuto část programu. Pokud tedy urychlíme tuto část urychlíme tedy celý algoritmus komprese. Program má dvě varianty algoritmu BWT jedna se provádí na CPU a druhá na GPU. Výběr mezi těmito varianty se provádí pomocí přepínače „-A“ z příkazové řádky.

Program je schopen zpracovat jakákoliv data, nemusí jít jen o text. Do výstupního souboru je vždy před každým blokem uložena hlavička. Tato hlavička určuje velikost vstupního bloku a pozici počátečního bajtu v BWT transformaci.

5.1 Implementace komprimačních algoritmů

Po aplikaci BWT na vstup program transformuje text pomocí Move-to-front transformace. Tento algoritmus přesune nejčastěji používané znaky na začátek abecedy, a tím sníží standardní chybovou odchylku dat. Jako první komprimační algoritmus jsem použil upravenou verzi Return-to-lengt, která komprimuje jen posloupnosti nul. Po použití algoritmu Move-to-front je totiž velice velká pravděpodobnost opakujících se znaků s hodnotou nula. Algoritmus Return-to-lenght pracuje jednoduše, pokud bude na vstupu nula spočítá se délka sekvence a na výstup se vypíše znak s hodnotou nula a počet následujících znaků s hodnotou nula. Tento přístup má nevýhodu pokud počet následujících nul bude větší než 255. Tento problém jsem vyřešil uložením nuly a počtu nul do výstupu a opětovným spuštěním komprese. Jiné posloupnosti jako posloupnosti nul implementovaný algoritmus vůbec nekomprimuje.

```
bwt_out = bwt(input);
mtf_out = mtf(bwt_out);
output = rle(mtf_out);
```

Kód 5.1: Pseudokód celého komprimačního programu

5.2 Sériová implementace BWT

Algoritmus BWT používá některé neměnné parametry, jako je parametr „V“. Tento parametr dokáže měnit paměťovou náročnost algoritmu BWT. V programu je nastaven na pevnou hodnotu 1024. Paměťová složitost algoritmu potom je 5,1565*n. Kde n je počet znaků na vstupu. Toto číslo se dá jednoduše spočítat. Pokud v dcs struktuře bude 40 indexů a velikost indexu jsou 4 bajty, potom při generování vzorků bude potřeba 40 * (N/1024) indexů. Základní algoritmus vytvoří dcs strukturu. Podle dcs struktury se následně vygenerují vzorky indexů, které je nutné setřídit. Z tohoto setříděného pole se následně vytvoří ISA pole, které se využívá v hlavním třídění všech podřetězců . Pro třídění podřetězců je využito indexů do původního textu. Tato možnost šetří paměť, jelikož není nutné uchovávat všechny podřetězce v paměti. V hlavním třídění se setřídí text do maximální hloubky „v“ a poté se pomocí dcs struktury určí které dva podřetězce jsou již setříděny a jsou podřetězci porovnávaných podřetězců textu.

```
dcs=gen_dcs(v = 1024);
sample_arr = gen_sample(dcs); //vygenerováni vzorku pod?et?zc?
sort_sample = sort(sample_sort, text); //set?íd?ní vzork?
ISA = gen_isa(sort_sample); //vytvo?ení ISA pole
index = gen_index(); //vygenerování všech pod?et?zc?
sort_index = sort(index, text, ISA, v);//set?íd?ní všech pod?et?zc?
output = gen_output(sort_index, text);
```

Kód 5.2: Pseudokód algoritmu BWT

Jako třídící algoritmy jsem vybral MSB-radixsort [12] pro pole větší než 4000 prvků. Pro pole v rozmezí od 16 do 4000 prvků jsem použil 3-cestný quicksort [10] a pro nejmenší pole, tedy pole o velikosti 16 prvků a méně jsem použil insertsort. Radixsort i quicksort dělí text podle prvního zadaného bajtu. Medián u quicksortu je vybrán porovnáním prvního, náhodného a posledního porovnávaného bajtu.

Následující kód zobrazuje všech podřetězců pomocí ISA pole. Je-li blok je dostatečně velký a setříděn do hloubky „v“, tedy do takové hloubky kde můžeme využít ISA pole, potom se tento algoritmus spustí a třídí jen na základě ISA pole. Druhým algoritmem je insertsort, který třídí pomocí ISA pole.

```
maxdepth = V;
push(stack, {0,size_text,0});
while(is_empty(stack)){
  block = pop(stack);
  if(block.dept >= maxdepth){
    //setridi text jen na zaklade ISA pole
    qsort(index, block, ISA);
  }
  else if(block.size > TRESHOLD_1){
    radixsort(&index, text, block, &stack);
  }
  else if(block.size > TRESHOLD_2){
    quicksort3way(&index, text, block, &stack);
  }
  else{
    dcs_insertsort(&index, text, block, ISA);
 }
}
```

Kód 5.3: Pseudokód třídění podřetězců v hlavním tříděni algoritmu BWT

5.3 Paralelní implementace

Základní algoritmus funguje úplně stejně jako sériová varianta. Program nejdříve alokuje potřebnou paměť na grafické kartě pro data, index a dcs strukturu. Následně vytvoří dcs strukturu a nahraje ji do grafické karty. Potom je načten určitý počet bloků dat do grafické karty. Jeden blok vždy zpracovává jeden warp. Načítaní a transformace bloků probíhá dokud jsou na vstupu data. Samotná komprimace dat se provádí zase sériově pomocí CPU. Aby byl paralelní algoritmus rychlejší je potřeba mít na vstupu dostatečný počet bloků data. Při práci s bloky o velikost 100000 znaků je potřeba mít na vstupu soubor o velikosti okolo 9MB pro plné zatížení dvou multiprocesorů s architekturou 2.x.

```
dcs =gen_dcs(v = 1024);
inint_gpu(dcs);
while(foef(soubor)){
  blocks = cpu_read_blocks(soubor) //na?te n?kolik blok? (98)
  transform_blocks = gpu_bwt_transform(blocks);
  komprim_blocks = cpu_mtf_rle_huff_kompress(transform_blocks);
  cpu_write(output_file, komprim_blocks);
}
```

Kód 5.4: Pseudokód akcelerovaného bwt pomocí gpu.

Třídění textů probíhá vektorově díky implicitní synchronizaci warpu. Každé vlákno z warpu kontroluje odpovídající čtveřici bajtů dvou porovnávaných textů. Tato čtveřice bajtů odpovídá pozici vlákna ve warpu. Důležitou funkcí pro všechny třídící metody použité v kernelu je „get_data()“, tato funkce přistupuje do globální paměti zarovnaným způsobem a načte čtveřici bajtů. Pokud tedy warp

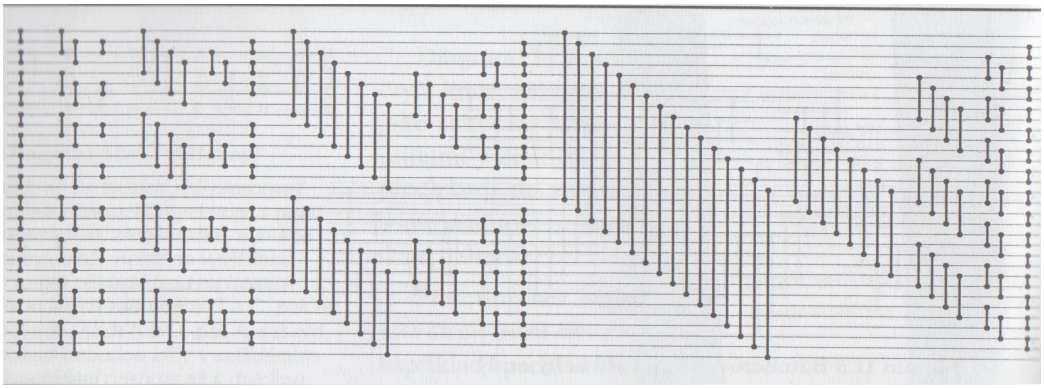
začne vykonávat tuto funkci dojde k vektorovému načtení načtení 128 bajtů dat. Následkem této implementace je nutnost dělitelnosti velikosti vstupního pole čtyřmi.

```
index = get_index + warp_id;
index /= size(int);
//zajišt?ní rotace textu ve vstupním poli.
index %= (len(text)/size(int));
out1 = ((int *)text)[index] >> (8* (get_index % size(int)));
index++;
index %= (len(text)/size(int));
out2 =((int*)text)[index]<<(size(int)(8* (get_index % size(int))));
out = out1 | out2;
```

Kod 5.1: Pseudokód funkce „ged_data()“

Pro grafickou kartu jsem zvolil jiné uspořádání třídících algoritmů. Pro pole větší jak 5000 prvků používám 3-cestný quicksort. Pro menší pole jsem zvolil Mergesort na místě. Oproti sériové implementaci jsem nepoužil radixsort. Nevýhodou radixsortu bylo využití velkého množství sdílené paměti, a spouštění příliš málo vláken. Výhoda implementace quicksortu oproti mergesortu na místě je využití sdílené paměti pro dělící prvek v quicksortu. Tímto krokem se sníží počet přístupů do globální paměti a uvolní se část přenosového pásma globální paměti pro ostatní vlákna.

Mergesort na místě je implementován tak aby co nejvíce využíval aritmeticko-logické jednotky grafické karty. Pracuje tímto způsobem: Mějme x vláken. Tato vlákna porovnávají vždy znak odpovídající jejich „id“ z x textů. Potom každé vlákno může porovnávat, který text je menší a který větší. Tabulka je ukázána na příkladu quicksortu. Kdy každé vlákno „vl.“ porovná odpovídající bajt v textu a výsledek uloží do určené proměnné (řádky). Nejdříve se porovná text_1, následně text_2 až poslední text_x. V druhé fázi bude každé vlákno kontrolovat jedno ze slov (sloupce). Ukázka třídící metody merge sort na místě, kterou jsem použil v paralelní implementaci.



Obr. 5.1: Batcherova třídící síť lichá-sudá. Ukázka třídící sítě mergesort pro 32 prvků. [10]

Při implementaci jsem od začátku nevyužíval implicitní synchronizace warpu. Pro využití synchronizace bylo nutné přepsat všechny třídící algoritmy. Následkem bylo odstranění většiny synchronizačních funkcí „__syncthreads()“ Zrychlení algoritmu bylo pouze o 10%.

5.4 Testy

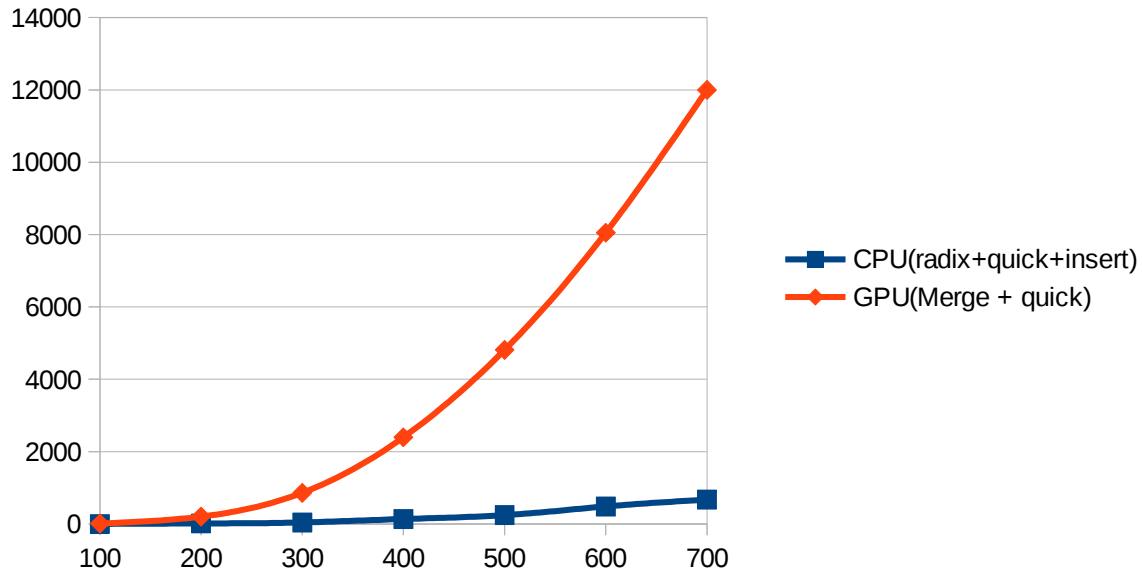
Program je rozdělen na moduly a to i kód pro grafickou kartu. Testy probíhaly na stroji s procesorem core 2 duo E4800, s pamětí DDR2 o velikosti 8GB, grafickou kartou „nvidia gt630“ s dvěma multiprocesory, tato grafická karta byla umístěna ve sběrnici PCI-E x16 verze 1. Překlad byl proveden za pomocí překladačů NVCC a GCC, přičemž kód pro grafickou kartu byl přeložen překladačem NVCC z toolkitu CUDA 5.5 a kód pro CPU byl přeložen překladačem GCC 4.6.3.

Kombinované algoritmy pro třídění textu mají prahy určující který algoritmus se zrovna použije popsané výše. Měření bylo provedeno na více variantách velikostí bloků, aby bylo vidět časovou složitost algoritmů a náklady vynaložené na tuto složitost. Výsledky jednotlivých měření jsou patrné v tabulkách a grafech.

Jelikož časově nejsložitější částí v BWT je třídění textu zaměřil jsem testování tímto směrem. Pro první jsem oddělil třídící algoritmy od komprese a ostatních úloh, proto jsem testoval každý algoritmus jen na porovnávání textu. Toto porovnání textu bylo provedeno do maximální hloubky textu. Algoritmy běžící na grafické kartě byly otestovány jen pro jeden blok s 32 vlákny, tedy jedním Warpem. Je nutné si uvědomit, kolik má grafická karta multiprocesorů a kolik bloků může běžet na jednom multiprocesoru. Tedy pro mou grafickou kartu by měly být všechny algoritmy běžící na GPU 96x rychlejší, jelikož se jedná o grafickou kartu se dvěma multiprocesory, a každý multiprocesor obsahuje 48 Warp Rychlosti jsou zapsané v tabulce a následně zobrazené do grafu. Z tabulky vyplývá: Aby došlo k urychlení musí být zároveň spuštěno více jak 20 bloků.

	100	200	300	400	500	600	700
CPU(radix+quick + insert)	0	14	41	137	244	484	673
GPU(quick + merge)	12	203	860	2392	4812	8051	12000
Podíl GPU/CPU	-	14	20	17	19	16	17

Tabulka 5.1: Doba vykonání jednotlivých třídících algoritmů(hodnoty jsou v sekundách)



Graf 5.1: Doba vykonání jednotlivých třídících algoritmů

Při druhém testování jsem zjistil chybu v programu. Testoval jsem rychlost celého algoritmu. Pro druhé testování byl program přeložen s parametry „--maxregcount=20“. Program při testování spouštěl 12 bloků na gpu v každém bloku bylo 256 vláken. Rychlost algoritmu běžícího na GPU byla vždy mnohonásobně vyšší než algoritmu běžícího na CPU. Tabulka ukazuje některá měření. Odrážka znamená vypnutí programu pro jeho příliš dlouhé vykonávání. Podle Předpokladu by mělo třídění 1GB souboru zhruba trvat 12s * (10240/96) = 20 minut. Bohužel třídění trvalo více jak 50Minut. Což je nepřiměřené.

Velikost Bloku	100 000	200 000	300 000
Aaaa.txt.300Kb	-/17sec	Neměřil	Neměřil
English.1024Mb	50min/8min	-/10min	-/12min
s_2,8MB	20sec/2sec	30sec/8sec	40sec/12sec

Tabulka 5.2: Ukázka doby běhu sériové a paralelní implementace.

Třetí testování je testování rozdílu ve velikostech jednotlivých souborů před a po kompresi. Jelikož komprese pro adaptivní Huffmanovo kódování není hotová(zvětšuje soubor). Proto jsem otestoval jen kompresní metodu RLE, která je funkční. V tabulce první sloupec udává název soubor druhý velikost souboru. V dalších sloupcích jsou zadány výsledné velikosti souboru s kompresním poměrem v závorce.

Soubor	**Velikost**	100 000	300 000	500 000	900 000
AAAA.txt	300KB	2,6Kb(115)		-	-
English.1024	1024MB	853MB(1,2)	810MB(1,26)	794MB(1,29)	781MB(1,31)
S_2,8MB	2,8MB	2,1MB(1,33)	1,4MB(2)	987KB(2,83)	497KB(5,633)

Tabulka 5.3: Účinnost komprimační metody.

5.4.1 Amdahlovo pravidlo

Pokud přejdeme od sériového algoritmu k paralelnímu algoritmu, Amdahlovo pravidlo nám říká maximální urychlení tohoto algoritmu. Bohužel tohoto urychlení není možné dosáhnout vzhledem k potřebné komunikaci mezi vlákny. Samotné Amdahlovo pravidlo neurčí přesně urychlení, jelikož je nutné brát v potaz synchronizaci mezi vlákny, rozdílné kmitočty aritmetickologických jednotek a rozdílnou hardwarovou implementaci (pipelining, přístup do paměti, RISC, CISC).

$$S(n)=\frac{1}{(1-P)+\frac{P}{N}}$$

Vzorec 5.1: Amdahalovo pravidlo. [8]
Kde:
N..... je počet procesorů vykonávající tuto činnost.
P..... je poměrná část algoritmu, která je urychlena v procentech.

Pro náš projekt budeme uvažovat urychlení BWT zhruba 70% aplikace, a 96 procesorů.

$$S(n)=\frac{1}{(1-P)+\frac{P}{N}}=\frac{1}{(1-0,7)+\frac{0,7}{96}}=3,254237285$$

Bohužel tohoto urychlení se mi nepovedlo dosáhnout.

5.4.2 Překážky při implemetaci

Další překážkou při programováni v CUDA je vyrovnávací paměť. Pokud se dobře nevyužije může být zápis i čtení z globální paměti velice pomalé, zvláště pokud v programu pracujeme s bajty a ne se čtveřicemi bajtů. Někdy je výhodné tuto vyrovnávací paměť vypnout při překladu.

Velké problémy činí práce se zásobníkem v kernelu, a nemožnost jeho realokace realokace. Proto musí být v kernelu implementována metoda, která nepotřebuje zásobník ani jinou dodatečnou paměť. Takovou metodou je mergesort na místě, který se spustí pokud dojde k naplnění zásobníku. Další problémovou částí je složitá implementace atomického přístupu k tomuto zásobníku.

5.4.3 Možná vylepšení

Prvním možným vylepšením je předělat program, tak aby urychlil transformaci několika gigabytů velkého textu. Bohužel toto řešení nese sebou riziko nedostatku paměti. Jedno z možných řešení je mít text mimo paměť grafické karty, a použít tedy externí třídění. Takové řešení nemusí být ideální, jelikož přenosy po PCI-E sběrnici jsou velice drahé.

Dalším vylepšením by bylo použít algoritmus LCP místo radixsortu, tak jak je popsané v [1]. Každý blok v kernelu by třídil jeden blok BWT o dané velikosti. Program by byl rozdělen na dvě části. V první částí by každý blok vytvořil ISA tabulku nezbytnou pro třídění podřetězců. Druhá část se bude opakovat dokud každý blok nesetřídí svůj blok BWT. Každý blok třídí blok pro který vytvářel pole ISA, ale třídění bude probíhat pro omezenou velikost díky LCP algoritmu. Dejme tomu že budeme třídit blok o velikost 900kb. Každý blok setřídí pouze část například do 1000 prvků. Tato část se zapíše na disk a algoritmus setřídí další část do 1000 prvků. To se bude opakovat dokud neskončí všechny bloky svoji konverzi. Nevýhodou algoritmu který je představen v [1] je, že nevíme kolikrát je potřeba spustit kernel jelikož velikost výstupních bloků se může lišit.

Jednoduché vylepšení lze implementovat i pro algoritmus MTF. Pokud sekvence nul bude větší jak 255. Představme si sekvenci 300 znaků s hodnotou nula, ty se uloží na výstup jako posloupnost znaků (0,255,0,45). Optimalizací bychom mohli jednu nulu odstranit, a předpokládat pokud počet nul přesáhne 255 další číslo značí pokračující sekvenci. Podle příkladu bychom dostali sekvenci (0,255,45).

Vylepšení které jsem chtěl implementovat, pokud by program dosahoval požadované rychlosti. Tento kód by se dal využít k synchronizaci bloků vláken, a spolupráci toho bloku vláken na setřídění jednoho bloku dat. Tím snížit paměťovou náročnost k udržení ISA pole v paměti. Kód je uveden v příloze B. Jeden blok pro každý warp je pro grafickou kartu paměťově neúnosné.

5.4.4 Nedostatky a jejich odstranění

Abych mohl uvažovat s polem suffixů, tak se ve vstupních datech nesmí vyskytovat znak s ordinální hodnotou nula. Tento nedostatek jsem odstranit použitím operace modulo na všechny ukazatele (indexi) do původního textu. Dále je nutné ošetřit ISA pole, tak aby byly záznamy pro setřídění podřetězců na konci pole. Proto jsem do ISA pole přidal ukazatele, které ukazují za konec textu. Ukazatel za konec textu se pomocí operace modulo převede zpátky do textu.

Bloky k třídění textu se ukládají na zásobník v pořadí nikoliv podle velikosti. To má za následek potřebu větší paměti pro zásobník. Řešení problému není nijak složité jen u quicksortu stačí ověřit který se tří bloku je nejmenší a ten vložit na zásobník jako poslední.

6 Závěr

Burrows-wheelerova transformace může podstatně zlepšit kompresní poměr. Bohužel tato vlastnost je vykoupena daleko větší dobou běhu programu. Samotné BWT spotřebuje kolem 70% času celého algoritmu. Tohoto nepříjemného efektu se můžeme zbavit akcelerováním výpočtu. Použitím akcelerace BWT můžeme zvětšit bloky dat, a tím zvětšit kompresní poměr, nebo provádět dekompresi větších souborů v reálném čase. Jednou takovou možností akcelerace je akcelerace pomocí grafické karty. Při akceleraci je důležité brát na vědomí: pro jakýkoliv algoritmus je lepší snížit časovou složitost než využívat paralelizace.

Programování pro architekturu CUDA je velice rozdílné paralelní programování od paralelního programování pro CPU. S takovým počtem vláken, kterými disponuje grafická karta je velice náročná implementace. Pro co největší paralelizmus se zde používá speciální hardwarová architektura, která se chová jako kdyby každé vlákno vykonávalo svůj program nezávisle na ostatních. S rostoucím počtem vláken se zvedá i paměťová náročnost algoritmu. Aby bylo možné co nejefektivněji zásobovat několik tisíc vláken dat, je nutné velice dobře rozumět synchronizačním mechanizmům v CUDA. Každé vlákno nemůže třídit data pro jeden blok BWT kvůli vyrovnávacím pamětem, paměťové náročnosti a hardwarové implementaci warpu.

Bohužel se mi žádného urychlení se mi dosáhnout nepovedlo, spíše naopak algoritmu běžící na GPU je vždy několikanásobně pomalejší.

Díky této bakalářské práci jsem se dozvěděl mnoho nového o možnostech implementace paralelizmu jak hardwarové tak softwarové, a v neposlední řadě i o architektuře grafické karty.

7 Literatura

[1] Juha Kärkkäinen. Fast bwt in small space by blockwise suffix sorting. Theoretical Computer Science. Issue 3. Essex: Elsevier,20. listopadu 2007,s. 249-257. ISSN 0303-3975. DOI 10.1016/j.tcs.2007.07.018

[2] Juha Kärkkäinen, Peter Sanders a Stefan Burkhardt. Linear Work Suffix Arraz Construction. Journal of the ACM. Issue 6. New York: ACM, listopad 2006,[s. 918-936] . ISSN 0004-5411. DOI 10.1145/1217856.1217858

[3] Juha Kärkkäinen a Stefan Burkhardt. Fast Lighweight Suffix Array Construction and Checking. Proceedings of the 14th annual conference of Combinatorial pattern matching, s. 55-69. 25.-27. červen 2003. Morelia, Michoacán, Mexiko

[4] Juha Kärkkäinen a Peter Sanders. Simple Linear Work Suffix Array Construction. Proceedings of the 30th internation conferenco on Automata, languages and programing, 30. června – 4. července 2003. Eindhoven, The Netherlands

[5] Jasbir Dhaliwal, Simon J. Puglisi a Anrew Turpin. Trends in Suffix Sorting: A Survey of Low Memory Algorithms. Proceedings of the Thirty-fifth Australasian Computer Science Conference, s. 91-98, 30. leden – 3. únor 2012, Melborne, Australia

[6] D. Adjeroh, T. Bell, A. Mukherjee. The burrow-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. New York: Springer Science+Business Media. ©2008. ISBN 978-0-387-78908-8.

[7] D. E. Knuth, The Art of Computer Programing volume 3 sorting and searching. 2. vydání. USA: Addison-Wesley, 1998, 780s. ISBN 0-201-8968-0

[8] Rob Farber. CUDA application design and development. Waltham: Elsevier, ©2011. ISBN 978-0-12-388426-8

[9] Jason Sanders a Edward Kandrot. CUDA by Example: an introduction to general-purpose GPU programing. Boston: Addison-wesley. ISBN 978-0-13-138768-3.

[10] R. Sedgewick. Algoritmy v C. Přel. Ing. Jiří Gree SortPress s.r.o./ Addison-wesley, 2003. 688 s. ISBN 80-86497-56-9.

[11] David Salomon. Data Compression: the complete reference. 3rd edition. New York: Springer-verlag, 2004. ISBN 0-387-40697-2.

[12] Peter M. McIlroy, Keith Bostic M. Douglas McIlroy. Engineering Radix Sort. Computing systems. 6. 1993, s.5-27. Doi 10.1.1.22.6990

[13] NVIDIA: CUDA Toolkit Documentation. [online]. [cit. 2014-03-29]. Dostupné z: http://docs.nvidia.com/cuda/

Příloha A
Obsah DVD

src – zdrojové soubory
Readme.txt – manuál k programuje
zprava – zprava ve formátu odf, pdf

Příloha B
Zdrojový kód

```
//q_run musí být volatilní prom?nná. Jinak nemusí dojít k
//zapsání prom?nné do sdílené pam?ti.
//block m?že být __syncthreads(); nebo
//jenom spin lock while(stack.top == 0)

POP(stack){
 do{
  block = false;
  lock(mux); //provedeni atomickými instukcemi ve sdilene pameti
    //kontrola zda každé vlákno provádí n?jakou ?innost
  if(is_any_block() && !is_empty(stack)){ //musi byt n?co v zasob
    unlock(mux);
    block(); //__syncthreads();
    block = true;
  }

  if(is_empty(stack)){
    unlock(mux);
    q_run[warp_id] = 1; //vlakno nemá co d?lat.
    block(); //__syncthreads();
    block = true;
    //kontrola zda vsechny q_run[NUM_THREAD/WARP_SIZE]jsou 1;
    if(is_end()){
      return false; // exit program (zasobnik je prazdny)
    }
    q_run[warp_id] = 0;
  }
 while(block);

 next_block = stack.arr[stack.top--];
 //vsechny data zm?nená data se presuno do pameti
 __threadfence_block();

 unlock(mux);
 return true;
}
```

Kód: Zamýšlený kód pro synchronizaci více vláken varpu. Metoda Pop vyzvednutí dat ze zásobníku.