VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE ALGORITMŮ KOMPRESE DAT NA PLATFORMĚ SONY PS3

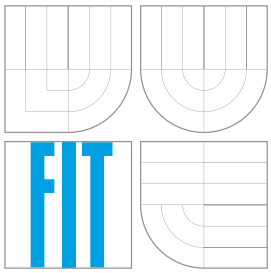
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DOMINIK BREITENBACHER

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE ALGORITMŮ KOMPRESE DAT NA PLATFORMĚ SONY PS3

ACCELERATION OF DATA COMPRESSION ALGORITHMS ON SONY PS3 PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DOMINIK BREITENBACHER

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VÁCLAV ŠIMEK

BRNO 2013

Abstrakt

Tato práce představuje použití zařízení PlayStationu 3 pro akceleraci kompresního algoritmu a snaží se tak prezentovat potenciál PlayStationu 3 pro použití na tyto úlohy. Pro demonstraci byla vybrána kompresní metoda založená na Burrows-Wheelerově transformaci. Výstup transformace je dále transformován pomocí Move-To-Front transformace a následně zakódován pomocí statického Huffmanova kódování. Kompresní algoritmus byl nazván PS3BWT. Ten vykonává kompresi po jednotlivých úlohách a snaží se vždy využít maximální počet dostupných procesorových jednotek tak, aby komprese byla provedena co nejrychleji.

Abstract

This paper presents the use of PlayStation 3 device for accelerating compression algorithms and tries to show the potencial of PlayStation 3 for use on these tasks. For a demonstration was selected compression method based on the Burrows-Wheeler transformation. The output of the transformation is further transformed by using the Move-To-Front transformation and subsequently encoded by the static Huffman encoding. The compression algorithm has been called PS3BWT. It performs compression by using each of tasks and tries to always use the maximum number of avaible processor units, so the compression is carried out as quickly as possible.

Klíčová slova

Sony PlayStation 3, bezztrátová komprese, Burrows-Wheelerova transformace, Move-To-Front transformace, Huffmanovo kódování, IBM Cell architektura.

Keywords

Sony PlayStation 3, lossless compression, Burrows-Wheeler transformation, Move-To-Front transformation, Huffman encoding, IBM Cell architecture.

Citace

Dominik Breitenbacher: Akcelerace algoritmů komprese dat
na platformě Sony PS3, bakalářská práce, Brno, FIT VUT v Brně, 2013

Akcelerace algoritmů komprese dat na platformě Sony PS3

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Václava Šimka.

.....
Dominik Breitenbacher
13. května 2013

Poděkování

Poděkovat bych chtěl svému vedoucímu bakalářské práce, Ing. Václavu Šimkovi, za to, že mi poskytl mnoho cenných rad a materiálů, ze kterých jsem mohl čerpat informace potřebné k vykonání této práce. Hlavním materiálem byla kniha [16], bez které by byl vývoj na PS3 velice obtížný. Také mi poskytl zařízení PlayStation 3, na kterém jsem mohl přímo zkoušet program tvořený v rámci této práce. Svými nápady mi dal opravdu mnoho možností, jakým směrem by se mohla práce ubírat a svým nadšením pro téma mě velmi motivoval k vypracování této práce.

Obsah

Kapitola 1

Úvod

Každý člověk pracující s počítači si potřebuje svá data někam uložit. Nejčastěji se jedná o pevný disk na daném počítači. Muže se však také jednat o USB flash disk, paměťovou kartu, e–mailovou schránku atd. Přestože má uživatel v dnešní době úložného prostoru a možností podstatně více než například před deseti lety, jsou tyto úložné prostory stále omezené a drahé. Může tak dojít k situaci, kdy na daném uložišti dojde místo a data není kam dále ukládat.

Tento problém řeší kompresní programy, které data zmenšují a napomáhají tak uživateli vyčerpat jeho úložný prostor pomaleji. Komprimovaná data však využívá mnoho lidí bez toho, aby věděli, že jsou tato data komprimovaná. Dnes je na internetu dostupný nespočet obrázků ve formátu jpg či hudba ve formátu mp3. Jedná se přitom o komprimovaná data. Naopak původní nekomprimovaná podoba těchto obrázků či hudby už se tak často nevyskytuje. Je to z toho důvodu, že původní podoba většinou obsahuje plno informací, které jsou podstatné jen pro určitou práci s danými daty a uživateli postačuje, když obrázek či hudební soubor splní svůj účel. Data je tedy možné zmenšit o tyto nepotřebné informace. Proces zmenšení souboru se nazývá ztrátová komprese. Díky tomu sice dojde ke snížení kvality, která však bude stále pro uživatele dostatečná, zároveň ale dojde k velké redukci velikosti komprimovaného obrázku či hudby.

Pokud ale uživatel pracuje s daty, kde vypuštění některých informací nepřipadá v úvahu, je nutné využít jednu z takzvaných bezztrátových kompresních metod. Toho může člověk využít, když si chce například zálohovat data na USB flash disk, kde je úložný prostor poměrně omezenější a není tedy úplně vhodné plýtvat tímto prostorem. Nebo také, když se člověk snaží poslat soubor dokumentů e–mailem a nechce příjemci úplně zaplnit schránku a zároveň nechce, aby se tento soubor dokumentů posílal příliš dlouho. Těchto metod je dnes dostupných opravdu mnoho a člověk může vybírat dle svých preferencí. Jedna z nejznámějších bezztrátových kompresních metod je ZIP, který je sice v mnoha ohledech překonaný, přesto se těší velké oblibě.

Kompresní algoritmy v dnešní době vykonávají svoji úlohu opravdu rychle a efektivně. I tak mnoho lidí stále přemýšlí, jak upravit tyto algoritmy, aby byly ještě rychlejší a efektivnější. Zastoupení vícejádrových procesorů mezi uživateli roste a tak se nabízí i úprava těchto algoritmů na variantu, která by dokázala použít vždy maximální počet jader dostupných na daném počítači. Toto se však netýká jenom počítačů, ale i chytrých telefonů, tabletů a dalších zařízení.

Jedním z takových zařízení je například i herní konzole PlayStation 3 od společnosti Sony. Jedná se o zařízení, které používá velmi ojedinělou architekturu. Díky svému výkonu

a relativně nízké ceně se stala velmi oblíbeným zařízením používaným pro vědecké účely. Nicméně od vydání PlayStation 3 již uběhla nějaká doba a tak je toto zařízení překonáváno například dnešními grafickými kartami. Přesto by bylo zajimavé zjistit, jak si toto zařízení vede s kompresí dat.

Proto vznikla tato bakalářská práce, která se snaží ukázat potenciál PlayStation 3 pro akceleraci kompresních algoritmů. Kapitola 2 nejdříve seznamuje čtenáře se samotnou herní konzolí PlayStation 3 a popisuje IBM Cell architekturu, která je srdcem PlayStationu. Kapitola 3 popisuje jednotlivé skupiny kompresních metod, některé algoritmy, které jsou součástí těchto metod a pak detailní pohled na Burrows–Wheelerovu transformaci, která byla zvolena jako základ kompresního algoritmu použitém v rámci této práce. Obsahem poslední kapitoly 4 je popis samotné implementace algoritmu a výsledky měření.

Kapitola 2

Architektura platformy Sony PlayStation 3

PlayStation 3 (PS3) je herní konzole, která byla představená firmou Sony v roce 2006. Jedná se o zařízení, které v té době mělo opravdu vysoký výkon a většina počítačů se mu v tomhle ohledu nemohla rovnat. Ovšem Sony tuto herní konzoli nevytvořila jen za účelem, aby se na ní hrály hry. PS3 konzole obsahovala tzv. funkci *OtherOS*, která umožňovala na disk PS3 nahrát a spouštět libovolný operační systém, který podporoval IBM Cell architekturu, jež je srdcem PS3.

Díky svému výkonu, funkci *OtherOS* a ceně, se PS3 konzole stala nástrojem mnoha vědeckých institutů. PS3 konzole se začala využívat v automobilizmu, letectví, ve finančních službách, ve vládních institutech atd. PS3 je možné zapojit i do takzvaných clusterů, což znamená, že se více konzolí PS3 k sobě paralelně připojí a mohou tak společně vykonávat velmi náročné úlohy. To předvedl Dr. Frank Mueller v roce 2007, když zapojil do clusteru 8 PS3. Následně však prohlásil, že pro jeho výpočty je velikost pamětí v PS3 velice limitujícím faktorem. V roce 2008 byl dokonce spuštěn první superpocítač založený na architektůře IBM Cell nazvaný IBM Roadrunner. Ten obsahoval 12960 Cell procesorů a 6480 dvoujádrových Opteronů, čímž dosáhl výpočetního výkonu 1.026 petaflopů. Tímto výkonem se dostal na první pozici v žebříčku TOP500 Linpack. PS3 konzole také poukázala na svůj výkon, když v roce 2007 Marc Stevens, Arjen K. Lenstra a Benne de Weger pomocí jedné konzole PS3 prolomili MD5 hrubou silou za pár hodin. V roce 2008 byly dokonce prolomeny SSL certifikáty využívající MD5 pomocí clusteru o velikosti 200 PS3 kontolí.

PS3 však není jediné zařízení, které využívá architekturu založenou na PowerPC procesorech. Konkurenční herní konzole XBox360 od firmy Microsoft Corporation využívá architekturu postavenou na 3 PPE (viz kapitola 2.1). Další konkurent Nintendo Wii obsahuje přímo PowerPC procesor a nástupce Wii U obsahuje IBM procesor velmi podobný Cell procesoru použitém v PS3, avšak byl na přání Nintenda upraven tak, aby více vyhovoval jejich požadavkům.

Procesor

PS3 obsahuje Cell procesor od společnosti IBM. Procesor obsahuje 1 PowerPC Procesor Element (PPE) a 8 kusů Synergistic Procesor Element (SPE). PPE je dvouvláknový vektorový procesor pro obecné použití bežící na frekvenci 3.2GHz s dvouúrovňovou cache, kdy cache první úrovně L1 má velikost 64kB a cache druhé úrovně L2 o velikosti 512kB. Proce-

sor je blíže popsán v sekci 2.1. SPE je procesor určený pro vysokorychlostní výpočty, který běží na frekvenci 3.1GHz. Narozdíl od PPE neobsahuje žádnou cache (viz sekce 2.1).

Paměť

Paměť, kterou má PS3 k dispozici je 256MB velká a jedná se o paměti typu XDR DRAM (eXtreme Data Rate Dynamic Random Access Memory). Jejími hlavními výhodami, oproti klasickým SDRAM, je vyšší rychlost a nízké latence. K této paměti (označovaná jako hlavní paměť) má přímý přístup pouze PPE. SPE v případě potřeby dat z hlavní paměti musí využívat DMA instrukce. Každá SPE však má vlastní paměť. Jedná se o uložiště SRAM o velikosti 256kB.

Přechod z PC na PS3 vyžaduje po programátorovi, aby si ve zvýšené míře hlídal paměť. Ještě více k tomu přispívá fakt, že abychom mohli vůbec na PS3 programovat, musíme pracovat v operačním systému, který také zabírá určité místo.

Grafický procesor

Grafický procesor vsazený do PS3 nese název RSX (Reality Synthesizer), který byl speciálně pro PS3 navržen firmou Nvidia. Jádro běží na frekvenci 550MHz. Velikost pamětí byla stanovena na 256MB a jsou typu GDDR3. Grafický čip podporuje DirectX 9.

2.1 IBM Cell architektura

Architektura vznikla ve spolupráci Sony s Toshibou a IBM. Tato architektura je rozšířením PowerPC architektury od IBM. Obrázek 2.1 ukazuje, z jakých bloků je architektura složená. Jsou to Kontrolér paměťového rozhraní (MIC), PowerPC Processor Element (PPE), 8 kusů Synergistic Processor Element (SPE), Propojovací sběrnice (EIB) a 2 vstupně/výstupní rozhraní (IOIF).

Propojovací sběrnice – EIB (Element Interconnect Bus)

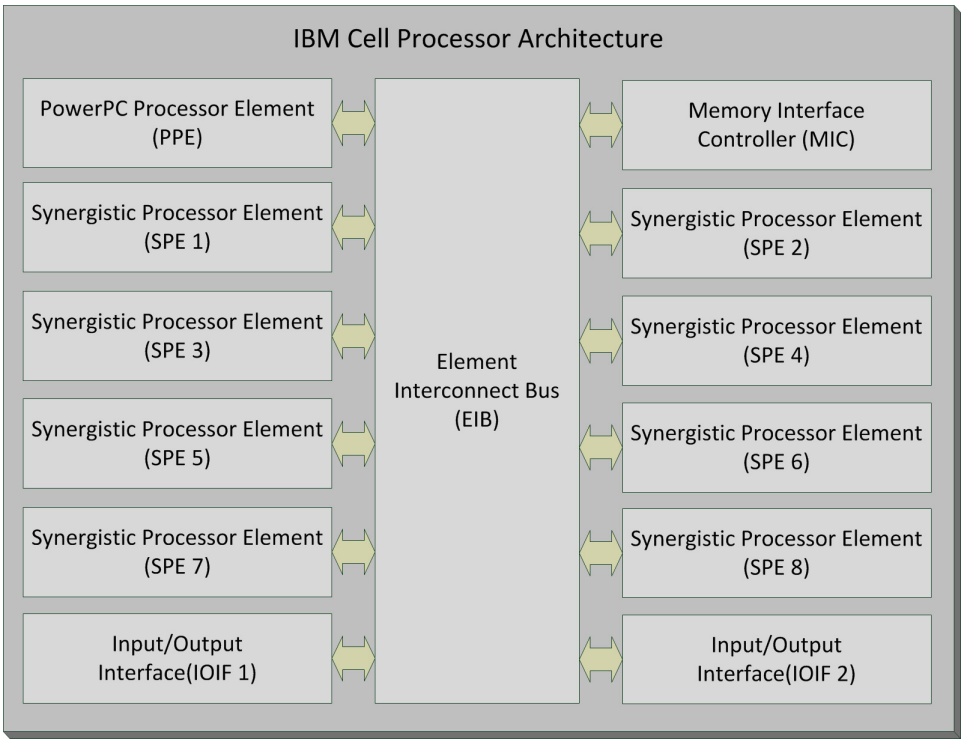
EIB propojuje všechny prvky procesoru a umožňuje jim tak vzájemnou komunikaci. Dále také EIB vyřizuje všechny požadavky DMA. Pro programátora je velmi důležité vědět, že každý DMA přenos může nést data o velikosti $1, 2, 4, 8, 16$ a násobek 16-ti bytů až do velikosti 16kB.

Vstupně/Výstupní rozhraní – IOIF (Input/Output Interface)

To má na starost propojení Cell procesoru s externími zařízeními. Jedním z externích zařízení je například grafický procesor RSX, klávesnice nebo myš.

Kontrolér paměťového rozhraní – MIC (Memory Interface Controller)

Kontrolér paměťového rozhraní propojuje hlavní paměť PS3 s Cell procesorem.



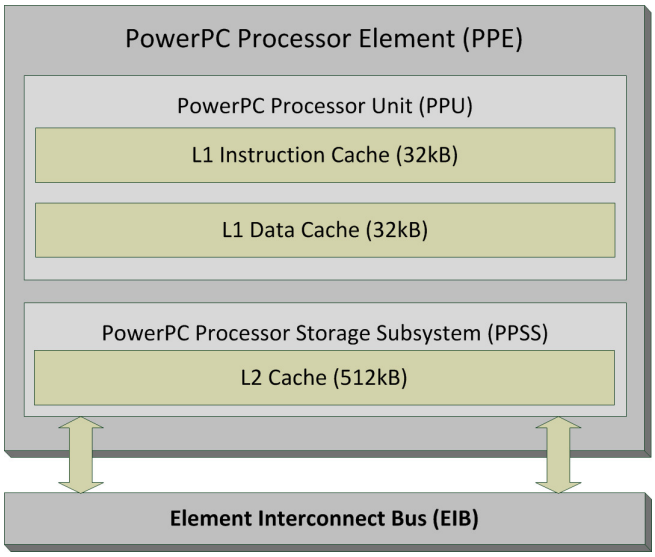
Obrázek 2.1: Architektura IBM Cell procesoru

PowerPC Processor Element - PPE

PPE se skládá z PPU jednotky (PowerPC Processor Unit) a PPSS (PowerPC Processor Storage Subsystem). PPU jednotka je procesor pro obecné použití. Proto slouží jako řídící jednotka, jejíž hlavní úlohou je řídit operace prováděné v SPE elementech a také je vhodná pro běh operačního systému. Jak bylo zmíněno v sekci 2, PPU jednotka je vektorová, což znamená, že dokáže pomocí jedné instrukce zpracovat více dat (například součet dvou sad 4 celých čísel).

Synergistic Processor Element - SPE

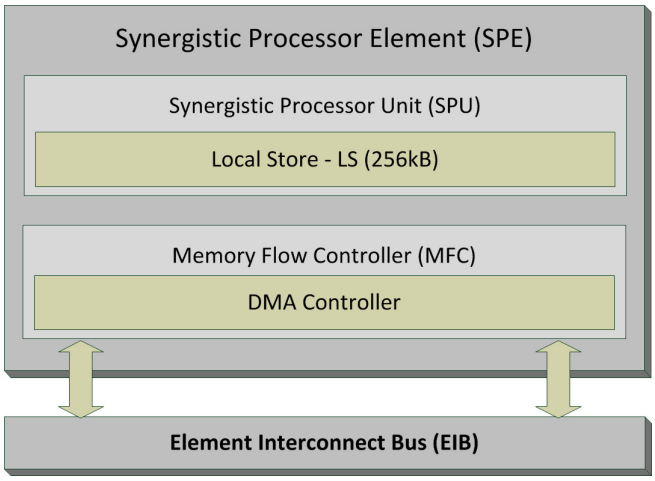
SPE element obsahuje SPU jednotku, která je zaměřená na velmi rychlý výpočet vektorových instrukcí. Každá SPU jednotka obsahuje dvě roury, které mohou paralelně zpracovávat instrukce. Toto je velmi výhodné, když chceme provádět nějaký výpočet, ten se bude zpracovávat v první rouře a zároveň žádáme o čtení dat, což obstará druhá roura. Každá SPU jednotka má svůj vlastní paměťový prostor Local Store (LS) o velikosti 256kB. Tento prostor sdílí jak program, tak data. Dále ale také poskytuje 128 128bitových registrů, které jsou využívány při vektorových operacích. SPU jednotka může pracovat pouze s daty umístěnými v lokální paměti. Lokální paměť je ovšem poměrně malá a tak pokud SPU jednotka potřebuje pracovat s daty větší velikosti, může využívat DMA přenosy. Jedná se o zasílání žádostí o přenos dat mezi LS a hlavní pamětí PS3. DMA přenosy jsou řízeny plánovačem a tak jsou požadavky vyřizovány efektivně. SPU jednotky nejsou tvořeny pro obecné použití a jsou tedy vhodné pouze pro výpočetní účely. Vyplývá to z toho, že SPU jednotka



Obrázek 2.2: Struktura PPE elementu

nemá žádnou cache, vlákna, stránkování ani paměťové segmenty. To vše dopomáhá k celkové rychlosti SPU jednotky. Jak bylo zmíněno, SPU jednotka je vektorový procesor a tedy při správném využití těchto instrukcí je zpracování dat ještě rychlejší. To platí i při práci s matematickými operacemi.

I když SPU jednotka nemá rozdělenou lokální paměť na segmenty, rozděluje tuto paměť pomyslně na zásobník a haldu. Zásobník se plní směrem shora dolů a halda naopak směrem z dola nahoru. Z tohoto dělení vyplývá, že SPU jednotka podporuje i dynamickou alokaci dat. Není zde však žádný mechanizmus, který by kontroloval, zda pro vytvoření nových dat, ať už v zásobníku nebo haldě, je v paměti dostatek místa. Tuto kontrolu musí vykonávat programátor.



Obrázek 2.3: Struktura SPE elementu

2.2 Programování na PS3

2.2.1 Přístup programátora

Programování na PS3 probíhá výhradně v jazyce C/C++. V rámci SPU jednotky je možné programovat výpočty také v jazyce Assembler. Myšlení při programování na PS3 musí být značně odlišné než při programování klasické PC architektury. Jak PPU jednotka, tak SPU jednotka jsou vektorovými procesory a pokud se programátor rychle adaptuje na používání vektorových instrukcí, jeho program poběží mnohem rychleji. Rychlejší bude jednak v tom, že vektorové instrukce provedou stejnou úlohu nad více daty rychleji než stejná skalární instrukce, ale také v tom, že data uložená ve vektoru jsou v paměti zarovnaná a neztrácí se tak čas při přístupu k datům.

Vektorové instrukce však lze využít pouze u úloh, ve kterých se nad bloky dat provádí stejné operace jako například posun 3D objektu o n pixelů. Pokud ovšem chceme provádět úlohy, kdy s bloky dat pracujeme v předem neznámém pořadí, pak musíme použít běžné skalární operace. Příkladem může být použití nějakého kryptografického algoritmu nebo třeba právě provádění komprese dat. V takovém případě vektorové instrukce nelze dobře použít a tak v takových úlohách vektorové procesory ztrácí svou výhodu.

2.2.2 Paralelizmus

Další věc, která vede k razantnímu urychlení vykonání programu, je zpracování úloh v co největší míře paralelně. To znamená použití SPU jednotek. Jelikož PS3 obsahuje 8 SPU jednotek, mohli bychom tedy teoreticky provádět až 8 úloh zároveň, které budou řízeny PPU jednotkou. Všech 8 SPU jednotek ovšem v praxi nelze pro zpracování programátorem naprogramovaného algoritmu použít. Je to z toho důvodu, že PS3 si jednu SPU jednotku rezervuje pro vlastní účely a pokud navíc programujeme pod operačním systémem Linux, pak ten si také rezervuje jednu SPU jednotku pro sebe. Maximální počet SPU jednotek, které můžeme v tomto případě použít, je 6.

Dostupné grafické karty dnes již překonávají svým výkonem PS3, avšak cena takového zařízení je mnohem vyšší než cena PS3. To by z PS3 stále dělalo velmi výhodné zařízení pro náročnější výpočty. Bohužel se podařilo prolomit ochranu proti použití nelegálně držených her pro PS3. Prolomení se provádělo přes funkci OtherOS a proto se Sony rozhodlo, že s vydáním upravené verze PS3, tzv. PS3 Slim, zablokuje funkci OtherOS. Kdo chtěl tuto funkci nadále využívat, nesměl PS3 aktualizovat na nový firmware. Tím zanikla šance na provádění poměrně náročných výpočtů na výkonném zařízení za velmi přijatelnou cenu.

2.2.3 Data a struktury

Přestože PS3 je pro programování paralelních procesů přímo předurčená, programování takových úloh je pro programátora, který přechází na architekturu IBM Cell, velmi náročné. Do verze IBM Cell SDK 3.0 bylo pro práci s SPU jednotkami možné využít pouze knihovnu `libspe2`. Průběh programu s využitím této knihovny je takový, že nejdříve PPU jednotka připraví data, která se budou posílat do SPU jednotky. Každá SPU jednotka však může dostat maximálně jeden ukazatel na data. Pokud chceme tedy posílat případně i přijímat různé druhy dat, musíme tuto situaci řešit vždy přes strukturu, která ponese příslušné informace a ukazatele na data. Ukazatele ve struktuře však nesmí být reprezentovány svým datovým typem, ale pro každý ukazatel ve strukture musí existovat proměnná typu integer,

která ponese hodnotu adresy uloženou v ukazateli. Dalším požadavkem na strukturu je, aby svou velikostí splňovala podmínku pro úspěšný DMA přenos do SPU jednotky (sekce 2.1). Pokud struktura nesplňuje tento požadavek, je nutné tuto strukturu upravit na požadovanou velikost. Posledním požadavkem na strukturu je, aby byla jak v PPU jednotce, tak v SPU jednotce nadefinována s parametrem `__attribute__((aligned(16)))`, která zajistí správné zarovnání struktury v paměti. Pokud programátor toto vše dodržel, pak při požadavku na načtení této struktury v SPU jednotce proběhne přenos úspěšně. Převezme-li programátor z načtené struktury ukazatel na data, která chce poslat z, případně odeslat do SPU jednotky, pak opět musí dodržet podmínky pro úspěšný přenos přes DMA a musí také brát ohled na to, že v jedné takto prováděné operaci nemůže přenést více jak 16kB dat. Chce-li přenést data většího objemu, musí sám zajistit příslušný počet požadavků na přenos dat a také posouvat ukazatele na data podle potřeby. Jakékoli pochybení vede k ukončení celého programu s chybou „`Bus Error`“. Programátor se více informací nedozví a je tedy na něm, aby zjistil, čím byla chyba zapříčiněna.

2.2.4 Paměť a vývoj

Jelikož je paměť PS3 velmi omezená, není vhodné, aby na ní byl provozován Linux s grafickým rozhraním a tak je programování na PS3 velice nepříjemné. Proto existuje možnost nainstalovat si IBM Cell SDK na architekturu x86 a programovat zde. Překladač pak zajistí, aby po přeložení programu na x86 šel tento program spustit přímo na PS3. Dalším velmi užitečným programem je simulátor Cell procesoru. Ten je od základu nastaven tak, že po jeho zapnutí, je možné spustit Linux a provádět tak všechny operace stejně jako na PS3. Zároveň je možné v simulátoru sledovat stav všech SPU jednotek a také stavy jednotlivých vláken PPU jednotky. Programátor tak má větší přehled o tom, co se v PS3 děje. Simulátor také dokáže ladit spuštěný program.

Další věc, kterou si musí programátor při programování úlohy v SPU jednotce hlídat je paměť. LS je velká jen 256kB a pokud programátor pracuje s větším objemem dat, může se stát, že mu paměť velmi rychle dojde.

2.2.5 ALF

Programování paralelních programů pro Cell procesor je pro programátora poměrně náročné, protože si musí hlídat velikosti dat, zarovnání paměti, musí řídit načítaní dat, což je ještě obtížnější, pokud chce využívat SPU jednotku naplno a tak provádět výpočetní operace nad dostupnými daty a zároveň načítat další, aby data byla co nejdříve dostupná pro zpracování. Existují úlohy, které kvůli těmto všem věcem, které programátor musí hlídat, jsou téměř nerealizovatelné. Matthew Scarpino, autor knihy *Programming the Cell Processor* [16] napsal:

> *„Když IBM vydalo verzi 2.1 Cell SDK, velmi dlouho jsem studoval jejich 16M rychlou Fourierovu transformaci(FFT). Normálně jsem velmi dobrý v rozluštění FFT, ale tenhle kód byl příšerný. Problém byl v ukládání dat a komunikaci. Každá FFT fáze posílala data se zarážející sérií nesouvislých operací nad seznamem a uložení dat do mnohonásobných bufferů. Kreslil jsem si diagramy, bloková schémata, ale sledovat kód bylo tak náročné, že jsem to prostě vzdal.“*

Proto od IBM Cell SDK verze 3.0 byla zavedena možnost využít místo knihovny `libspe2` knihovnu `ALF` (Accelerated Library Framework). Tato knihovna byla vytvořena pro prová-

dění úloh vyžadujících mnoho přenosů o velkých objemech dat. Programátor se již nemusí starat o řízení přenosů dat, vše za něj provede ALF, který přináší mnoho nových konceptů, funkcí a struktur dat. Ale jeho studium vyžaduje hodně času, než programátor plně pochopí funkčnost této knihovny. Funkčnost knihovny ovšem sahá za rámec této práce a nebude blíže popsána.

Kapitola 3

Kompresní algoritmy a metody

3.1 Dělení kompresních algoritmů

3.1.1 Slovníkové kompresní algoritmy

Kompresní algoritmy na bázi práce se slovníkem využívají faktu, že procházený soubor obsahuje mnoho opakujících se frází jako jsou slova, části slov anebo jiné kousky textu. Při průchodu si značí jednotlivé fráze a pokud právě zkoumanou frázi nezná, uloží si ji do svého slovníku. Naopak pokud se zkoumaná fráze již ve slovníku vyskytuje, nahradí ji referencí do slovníku. Reference typicky vyjadřuje vzdálenost prvního výskytu fráze od aktuální pozice. Pokud se nám podaří tuto referenci zakódovat do menší posloupnosti znaků než je nahrazovaná fráze, dochází k úspoře místa. Komprimovaná data se tedy skládají z frází, pro která nebyla nalezena žádná reference a referencí na fráze.

Nejznámějším zástupcem tohoto typu kompresních algoritmů je LZ77[18]. Ten pracuje na principu posouvajícího se okna. Okno je rozděleno na dvě části. První část obsahuje již zpracovaná data a slouží jako slovník. V pravé části se nachází data připravená ke zpracování. Slovníková část okna bývá v praktické implementaci mnohem větší než část s nezpracovanými daty. Čím je levá část okna větší, tím je větší pravděpodobnost, že bude nalezen řetězec, který bude v co největší míře odpovídat aktuálně zpracovávaným datům. Čím delší řetězec je nalezen, tím víc dochází ke kompresi. Shodný řetězec na vstupu je totiž nahrazen referencí na již existující stejný řetězec nacházející se ve slovníku. Reference se obvykle skládá ze tří informací. Na které pozici se shodný řetězec nachází, jak je dlouhý a jaký následuje znak, který porušil shodu. Může ovšem nastat situace, kdy nalezený řetězec bude tak malý, že jeho zakódování pomocí reference bude větší než samotný řetězec. Dále je také vhodné volit rozumnou velikost slovníkové části okna. Větší slovníková část sice poskytuje větší pravděpodobnost nalezení shodného řetězce na vstupu, ale také v případě takového nalezení může být vzdálenost tak velká, že při transformování řetězce na referenci dojde k prodloužení výstupního řetězce místo kýženého zmenšení.

3.1.2 Pravděpodobnostní kompresní algoritmy

Algoritmy patřící do této skupiny, jak název napovídá, pracují s pravděpodobnostmi. Pravděpodobnost se vztahuje vždy k symbolu z použité abecedy. Pravděpodobnost vztažená k symbolu se odvíjí od zvoleného kontextu. Pravděpodobnostní algoritmy se nejčastěji používají v kombinaci s aritmetickým kódováním (viz sekce 3.3.3), který při svém kódování také pracuje s pravděpodobnostmi.

U statické varianty se vždy pracuje se stejnou pravděpodobností pro daný znak. Ta může být určena například podle výskytu znaku v anglických textech či textech jiného jazyka.

Adaptivní varianta naopak přepočítává pravděpodobnost s každým načteným znakem. Jedná se sice o náročnější a pomalejší metodu, ale v konečném důsledku poskytuje lepší kompresi než statická varianta, protože používá přesné rozložení pravděpodobnosti.

3.1.3 Kompresní metody založené na Burrows-Wheelerově transformaci

Mnoho kompresních metod pracuje v *streaming módu*, kde se načte znak nebo pár znaků, ty se zpracují a pak následuje další čtení, a to tak dlouho, dokud jsou na vstupu nějaká data. Burrows-Wheelerova metoda (BW) [3] naopak pracuje v *blokovém módu*, kde na vstup jsou čteny bloky a každý blok je zpracován zvlášť jako jeden řetězec. Při obecném použití je Burrows-Wheelerova metoda vhodná pro zpracovávání obrázků, zvuku, textu a kompresní algoritmy založené na BW dokáží dosáhnout velmi vysoké komprese (až 1 bit na znak).

BW metoda je základem kompresního algoritmu, který je vytvořen v rámci této bakalářské práce, a proto je důkladněji vysvětlena v sekci 3.2.

3.2 Burrows-Wheelerova transformace

3.2.1 Základní popis BWT

Jak bylo zmíněno v sekci 3.1.3, BW metoda pracuje s bloky dat jako s řetězci. Výstižně a srozumitelně popsal princip BW metody David Salomon v knize [15]. Označme tedy vstupní řetězec jako S, který bude dlouhý n symbolů. Tento řetězec budeme transformovat pomocí BW na výstupní řetězec L. Transformací se zde myslí získání permutace, tedy uspořádané n-tice, která má stejný počet prvků jako původní řetězec.

Komprese

Jak se provede transformace ze vstupního řetězce S na řetězec L, ukáži na příkladu v několika krocích:

1. Vstupním řetězcem S bude „`swiss_miss`“. Pro tento řetězec se vytvoří matice o velikosti $n \times n$, kde na vrcholu je uložen řetězec S. Za ním se nachází $n-1$ obrazů řetězce S, kdy každý z nich je cyklicky orotován o jeden symbol doleva oproti předchozímu obrazu či původnímu řetězci S.

2. Dalším krokem v transformaci je lexikografické seřazení matice.

3. Z této seřazené matice se vybere poslední sloupec a ten je označen jako řetězec L. V našem příkladě, jak ukazuje obrázek 3.1, bude výstupním řetězcem L „`swm_siisss`“.

Má-li být BW transformace maximálně efektivní, je nutné použít takový třídící algoritmus, který bude dodržovat pravidlo, že pokud je symbol s na určité pozici v řetězci L, pak ostatní výskyty tohoto symbolu by se měly nacházet poblíž.

Aby bylo možné zpětně rekonstruovat řetězec S z L, samotný řetězec L nestačí. Potřebujeme ještě znát index původního řetězce S v seřazené matici. Tento index označíme I. V příkladě $I = 8$.

Index	Matice	Index	Seřazená matice	L
0	swiss_miss	5	_misssswiss	s
1	wiss_misss	2	iss_misssw	w
2	iss_misssw	7	issswiss_m	m
3	ss_missswi	6	missswiss_	_
4	s_missswis	4	s_missswis	s
5	_misssswiss	3	ss_missswi	i
6	missswiss_	8	ssswiss_mi	i
7	issswiss_m	9	sswiss_mis	s
8	ssswiss_mi	0	swiss_miss	s
9	sswiss_mis	1	wiss_misss	s

Obrázek 3.1: Transformace vstupního řetězce na výstupní

Výstupní řetězec L a index I by nyní bylo možné zakódovat například pomocí Huffmanova kódování. Tyto případy se ale v praxi moc nevyskytují. Výstupní řetězec lze ještě dále transformovat například pomocí metody Move-To-Front (viz sekce 3.2.4) anebo pomocí RLE (viz sekce 3.2.5), případně obojím. Tyto metody napomáhají ke zvýšení kompresního poměru, a proto je vhodné je zakomponovat do kompresního algoritmu, po nich se provádí samotné zakódování.

Dekomprese

Při dekódování je načten zakódovaný soubor, dekomprimován pomocí Huffmanova kódování, Move-To-Front nebo RLE případně obojím a následně je provedena rekonstrukce originálního řetězce S z řetězce L ve třech krocích:

1. První sloupec seřazené matice (sloupec F na obrázku 3.2) je sestaven z řetězce L tak, že se řetězec L lexikograficky seřadí.

2. Při třídění L je vytvořeno pomocné pole T, které bude obsahovat indexy řetězce L po seřazení.

3. Po získání pole T řetězec F již není nadále potřebný. Rekonstrukce dále postupuje podle:

$$\begin{aligned} &S[n-1-i] \leftarrow L[T^i[I]], \quad \text{pro } i = 0, 1, \ldots, n-1, \\ &\text{kde } T^0[j] = j, \text{ a } T^{i+1}[j] = T[T^i[j]]. \end{aligned} \tag{3.1}$$

Zde je několik kroků v rekonstrukci:

$$\begin{aligned} &S[10-1-0] = L[T^0[I]] = L[T^0[8]] = L[8] = s, \\ &S[10-1-1] = L[T^1[I]] = L[T[T^0[I]]] = L[T[8]] = L[7] = s, \\ &S[10-1-2] = L[T^2[I]] = L[T[T[T^0[I]]]] = L[T[T[8]]] = L[T[7]] = L[6] = i, \\ &\vdots \\ &S[10-1-9] = L[T^9[I]] = \ldots = L[9] = s. \end{aligned}$$

Index	F	L	T
0	_	s	4
1	i	w	9
2	i	m	3
3	m	_	0
4	s	s	5
5	s	i	1
6	s	i	2
7	s	s	6
8	s	s	7
9	w	s	8

Obrázek 3.2: Transformace výstupního řetězce na vstupní

3.2.2 Podpůrné algoritmy

Pokud bychom prováděli transformaci přesně tak, jak bylo popsáno v sekci 3.2.1, byla by BWT paměťově velmi náročná, protože bychom museli vytvořit a pracovat s maticí permutací vstupního řetězce. Proto existují algoritmy, které dokáží pracovat pouze s indexy počátečního znaku každé permutace, což je postačující. Tím se výrazně sníží nároky na paměť. Varianty těchto algoritmů jsou popsány v sekci 3.2.3.

Dalšími algoritmy, které je vhodné použít, jsou algoritmy podporující zvýšení kompresního poměru. Jedná se o algoritmy Run Length Encoding a Move-To-Front. První zmíněná metoda dokáže za správných podmínek výstup BWT zmenšit. Druhá metoda sice nedokáže výstup BWT zmenšit, avšak dokáže jej transformovat tak, že při použití algorimtu k zakódování dat, jako je například Huffmanovo kódování, a za vhodných podmínek, ovlivní konečnou velikost zakódovaných dat. Některé kompresní programy využívají kombinaci obou těchto metod. Jak tyto metody fungují je vysvětleno v sekcích 3.2.4 a 3.2.5.

Jako konečná fáze komprese dat přichází algorimty, které slouží k zakódování vstupních dat. Nejtradičnějším reprezentantem těchto algoritmů je Huffmanovo kódování. To lze vykonávat staticky (sekce 3.3.1) anebo adaptivně (sekce 3.3.2). Alternativou k Huffmanovu kódování je Aritmetické kódování, případně jeho vylepšená varianta Range encoding. Jejich funkce je vysvětlena v sekci 3.3.3.

3.2.3 Suffix-Tree a Suffix-Array

Suffix-Tree

Jedná se o datovou strukturu, která obsahuje všechny podřetězce vstupního řetězce včetně původního řetězce a je využívaná jako třídící algoritmus [17]. Jak název napovídá jedná se o strukturu tvaru stromu.

Velkou výhodou je, že při zvolení vhodného algoritmu, může být konstrukce stromu provedena v lineárním čase $O(n)$. Problémem při využívání Suffix-Tree na platformě PS3 je jeho příliš velká paměťová náročnost. Nejúspornějším řešením je pravděpodobně varianta Kurtze a Balkenhola, jejichž řešení má paměťovou náročnost v nejhorším případě $O(16n)$. Toto řešení by i přes svou paměťovou náročnost bylo možné na PS3 použít, avšak nebylo by možné ho použít v SPU jednotkách pro paralelní zpracování, kde je paměť 256kB a nebylo

by tedy možné použít dostatečně velké datové bloky, které by poskytovaly dobrý kompresní poměr. Proto jako alternativa byla zvolena konstrukce Suffix-Array, která dokáže také pracovat s lineární časovou složitostí a zároveň její paměťová náročnost je dostatečně nízká pro využití v paralelním zpracování na PS3.

Suffix-Array

Suffix-Array je datová struktura příbuzná struktuře Suffix-Tree [10]. Suffix-Array lze použít téměř ve všech případech, ve kterých se využívá Suffix-Tree. V nejjednodušším podání pracuje Suffix-Array tak, že pro určitý vstup si vytvoří pole indexů o stejné velikosti jako je délka vstupu. Každý index reprezentuje pozici znaku ze vstupu. Indexy jsou seřazeny podle lexikografické hodnoty znaků na vstupu.

Suffix-Array dělíme do svou skupin. Jedny jsou klasické a druhé tzv. lightweight. Lightweight jsou zaměřeny na to, aby měly co možná nejmenší paměťovou náročnost. Ta se aktuálně dostává na hodnotu $O(5n)$ [11], kdy $1n$ stojí samotný vstup a $4n$ je pole 4bytových integerů sloužících jako indexy na jednotlivé znaky vstupu. Na úkor takto nízké paměťové náročnosti se zvyšuje časová naročnost na $O(n^2 \log(n))$.

Na konci roku 2011 se objevila Suffix-Array konstrukce SA-IS [12], která pracuje s časovou náročností $O(n)$. Velkou výhodou ovšem je, že paměťová náročnost této konstrukce je $O(6.25n)$. Jedná se tedy o konstrukci, která svými vlastnostmi nejlépe odpovídá požadavkům pro použití na PS3.

3.2.4 Metoda Move-To-Front - MTF

MTF funguje takto[2]:

1. Stanoví se abeceda A, se kterou se bude pracovat.
2. Po stanovení abecedy je každý symbol z A uložen do seznamu.
3. Následně se začne provádět čtení ze vstupu. Čtení se provádí po znacích.
4. Načtený znak je vyhledán v seznamu symbolů.
5. Index, na kterém je symbol nalezen, slouží jako zakódování znaku a tento index je uložen na výstup.
6. Po uložení indexu na výstup je symbol v seznamu posunut na začátek.
7. kroky 3 až 6 se provádí tak dlouho, dokud není zpracován celý vstup.

Z kroků 5 a 6 lze vypozorovat, že index vyjadřuje počet symbolů, které předcházely načtený symbol. Pro maximální efektivitu této metody je nutné splnit předpoklad, že za právě zpracovávaným znakem bude na vstupu ten samý znak a nejlépe hned několikrát. Tím se totiž radikálně zvýší četnost jediného znaku, což napomáhá k zakódování například pomocí Huffmanova kódování. Pro lepší pochopení obrázek 3.3 ukazuje, jak se zakóduje řetězec „`swiss_miss`“ a řetězec „`sssssswwwww`“.

Obrázek 3.3a ukazuje, že předpoklad zmíněný výše, nebyl u řetězce `swiss_miss` zcela naplněn, a proto by využití MTF v tomto případě pravděpodobně nevedlo k zvýšení kompresního poměru při použití Huffmanova kódování. Naopak tomu je u řetězce `sssssswwwww`. Zakódováním tohoto řetězce se velmi výrazně zvýšila četnost jednoho znaku, což bude mít

Abeceda					Vstupu	Výstup
_	i	m	**s**	w	s	3
s	_	i	m	**w**	w	4
w	s	_	**i**	m	i	3
i	w	**s**	_	m	s	2
s	i	w	_	m	s	0
s	i	w	_	m	_	3
_	s	i	w	**m**	m	4
m	_	s	**i**	w	i	3
i	m	_	**s**	w	s	3
s	i	m	_	w	s	0

(a) `swiss_miss`

Abeceda					Vstupu	Výstup
_	i	m	**s**	w	s	3
s	_	i	m	w	s	0
s	_	i	m	w	s	0
s	_	i	m	w	s	0
s	_	i	m	w	s	0
s	_	i	m	**w**	w	4
w	s	_	i	m	w	0
w	s	_	i	m	w	0
w	s	_	i	m	w	0
w	s	_	i	m	w	0

(b) `sssssswwwww`

Obrázek 3.3: MTF transformace

za následek zvýšení kompresního poměru při využití Huffmanova kódování. Díky MTF může být nejčetnější znak zakódován i na 1bit.

3.2.5 Metoda Run Length Encoding - RLE

Jedná se o jednu z nejjednodušších metod pro zakódování dat [7]. Její princip spočívá v tom, že se prochází vstupní řetězec a každý n-násobný výskyt stejného symbolu ihned za sebou, je nahrazen za počet výskytů a samotný symbol. Na příkladu ukázáno, řetězec `abbcccddddeeeefffggh` se zakóduje pomocí RLE na `a2b3c4d4f3f2gh`.

Toto bude ovšem platit pouze v případě, kdy budeme kódovat řetězce, ve kterých se nebudou vyskytovat čísla. V opačném případě by totiž nebylo možné poznat, co reprezentuje počet výskytů a co je samotná součást vstupního řetězce. Proto je nutné zavést do kódování mechanizmus, díky kterému pak při dekódování půjde jednoznačně určit, co značí počet výskytů a co je číslo v rámci vstupního řetězce. Typicky se jedná o zavedení nějakého speciálního escape znaku. Tento znak se ovšem nesmí nacházet v abecedě, kterou používá vstupní řetězec. Zavedením escape znaku se musí zakódovávat místo nahrazením dvojicí trojicí. Když se zakódovávalo dvojicí, tak v nejhorším případě došlo k tomu, že se vstupní řetězec zakódováním neredukoval. Při použití trojice může už dojít k situaci, kdy po zakódování bude naopak výstupní řetězec delší než vstupní. Je to z toho důvodu, že nahrazení se provede už při výskytu dvou stejných symbolů. Řešením je využívat nahrazení pouze v případě, kdy počet stejných symbolů bude roven nebo větší než 3.

Dekódování je stejně jednoduché jako kódování. Dekodér ukládá znaky ze vstupního řetězce do výstupního tak dlouho, dokud nenarazí na escape znak. V takovém případě dekodér načte počet výskytů symbolu, ten si uloží a načte samotný symbol. Dekodér pak zkopíruje na výstup daný symbol tolikrát, kolik je uložený počet výskytů. Tento postup provádí dekodér tak dlouho, dokud není zpracován celý vstup.

3.3 Metody entropického kódování

3.3.1 Huffmanovo kódování statické

Jedná se o jednu z nejznámějších a nejpoužívanějších metod pro zakódování dat[8]. Její časová složitost je $O(n + |\Sigma| * \log |\Sigma|)$ a paměťová $O(|\Sigma|)$. Metoda pracuje dvouprůchodově. V prvním průchodu počítá četnost výskytu jednotlivých znaků abecedy. Ze získané četnosti pokračuje Huffmanovo kódování vybudováním binárního stromu. Vybudování probíhá následujícím způsobem:

1. Na začátku se vytvoří uzly stromu, kde každý uzel obsahuje jeden znak z abecedy, jeho četnost a ukazatele na přímé následníky. Ukazatel v tomto případě reprezentuje hranu. Tyto uzly jsou zároveň listmi vytvářeného stromu.

2. Jednotlivé listy jsou seřazeny dle četnosti.

3. Vždy se vezme první a druhý uzel s nejmenšími četnostmi. Vytvoří se nový uzel, který se stane uzlem nadřazeným pro tyto dva uzly. Četnosti vybraných uzlů se sečtou a výsledek se uloží do vytvořeného uzlu. Ukazatele ve vytvořeném uzlu jsou naplněny adresami na vybrané uzly. Vytvořený uzel nenese ani jednu hodnotu vybraných uzlů, ale místo této hodnoty se vloží hodnota, která symbolizuje, že daný uzel vznikl spojením dvou předcházejících uzlů.

4. Bod 3 provádíme tak dlouho, dokud nevznikne kořenový uzel

Každá hrana pak reprezentuje jeden bit kódu pro daný znak. Celý kód získáme průchodem stromu od kořene k listu obsahující hledaný znak. Pokud se ve vstupním řetězci nachází nějaký znak jehož četnost výskytu je mnohem vyšší než četnost ostatních, pak tento znak může být zakódován i do jednoho bitu. Alternativou k vyhledávání kódu pro znak může být vytvoření tabulky kódu z vytvořeného stromu, kdy ordinální hodnota hledaného znaku bude zároveň indexem do této tabulky. Při druhém průchodu hledáme kód aktuálně zpracovávaného znaku ve stromu případně v tabulce kódů a nalezený kód následně uložíme do výsledného řetězce.

Pro případné rozkódování zakódovaného souboru Huffmanovým kódováním nám zakódovaný řetězec nestačí. Bez dalších informací totiž není možné zjistit, jak jsou jednotlivé kódy dlouhé a který znak reprezentují. Proto je nutné k zakódovanému řetězci ještě uložit binární strom, podle kterého byl původní řetězec zakódován. S potřebou uložení stromu však roste i výsledný soubor. Může se tak stát, že zakódovaný soubor bude v konečném důsledku větší než původní řetězec. To se hlavně stává u souborů, kde výskyt jednotlivých znaků abecedy je poměrně vyrovnaný.

3.3.2 Adaptivní Huffmanovo kódování

Tato varianta byla popsána v [5],[6] s následnou úpravou popsanou v [9]. Metoda pracuje pouze jednoprůchodově. Paměťová a časová náročnost zůstává stejná. Rozdíl oproti statickému Huffmanovu kódování je v tom, že adaptivní kódování vytváří a upravuje strom podle aktuálně zpracovávaného znaku. Nepotřebuje tedy znát četnost znaků. Další výhodou oproti statickému kódování je fakt, že adaptivní kódování nepotřebuje do výstupního souboru ukládat binární strom, podle kterého by byla provedena případná rekonstrukce původního řetězce. Tím může být konečná velikost výstupního souboru menší než v případě

statického kódování. Pouhé uložení samotných zakódovaných dat by ovšem znemožňovalo zpětnou rekonstrukci. Dekodér by totiž neměl žádnou informaci o tom, kdy se v zakódovaném kódu poprvé objevil určitý znak. Proto se při kompresi znak, který se objevil ve vstupním řetězci poprvé uloží tak, že se do výstupního souboru uloží speciální escape znak a za něj se uloží zpracovávaný znak. Nevýhoda této metody je, že při chybě čí výpadku jednoho znaku by mohla být komprese či dekomprese znehodnocená.

3.3.3 Aritmetické kódování

Tato metoda kódování byla poprvé představena Peterem Eliasem. Praktické implementace se tato metoda dočkala až v roce 1976 pány Richardem Pascem [13] a Jormem Rissanenem [14]. Hlavní myšlenka této metody je v zakódování celého vstupního řetězce do jediného realného čísla. Díky této vlastnosti se aritmetické kódování více blíží k optimální velikosti kódu, který je použit pro zakódování daného znaku. Optimální délku kódu pro znak s lze spočítat dle vzorce „$-\log_2(P_s)$“, kde P_s je pravděpodobnost výskytu znaku s.

U statického kódování je možné získat pravděpodobnost jednotlivých znaků dvěma způsoby. Buď jsou pravděpodobnosti znaků předem pevně dány nebo se před kódováním spočte četnost výskytů jednotlivých znaků ve vstupním řetězci a z těchto četností se následně spočte pravděpodobnost výskytu pro každý znak. Metoda se více blíží k optimální délce kódu, protože například narozdíl od Huffmanova kódování nepotřebuje k zakódování znaku pevnou délku kódu.

Metoda funguje tak, že se na začátku stanoví interval $< 0, 1)$. Při zpracovávání vstupu se tento interval zmenšuje. Velikost omezení intervalu závisí na pravděpodobnosti právě zpracovávaného znaku. Čím vyšší pravděpodobnost znak má, tím méně omezuje interval a tím pádem vyžaduje menší počet bitů pro zakódování. Poté, co je nastaven počáteční interval, se tento interval rozdělí na menší intervaly podle pravděpodobnosti výskytu jednotlivých znaků. Například u řetězce „bratr“ bude mít znak b pravděpodobnost výskytu $P_b = 0.2$, znak r bude mít $P_r = 0.4$, $P_a = 0.2$ a $P_t = 0.2$. Počáteční interval se tedy rozdělí na intervaly $< 0, 0.2)$ pro P_a, $< 0.2, 0.4)$ pro P_b, $< 0.4, 0.8)$ pro P_r a $< 0.8, 1)$ pro P_t. Při druhém průchodu vstupním řetězcem, kdy dochází k samotnému zakódování, se postupně čtou jednotlivé znaky. Každý načtený znak omezí aktuální interval podle:

$$\begin{aligned} NovSpodHran &= AktSpodHran + RozAktInt * SpodHranZnak \\ NovHorHran &= AktHorHran + RozAktInt * HorHranZnak \end{aligned}$$

kde:

$NovSpodHran$	je nová spodní hranice intervalu,
$NovHorHran$	je nová horní hranice intervalu,
$AktSpodHran$	je stávající hodnota spodní hranice intervalu,
$AktHorHran$	je stávající hodnota horní hranice intervalu,
$RozAktInt$	je rozsah intervalu,
$SpodHranZnak$	je hodnota spodní hranice načteného znaku,
$HorHranZnak$	je hodnota horní hranice načteného znaku,

V uvedeném příkladu by tedy probíhalo zakódování takto: Jako první by kodér načetl znak b, jeho interval je $[0.2, 0.4)$. Dojde tedy k omezení intervalu podle postupu zmíněného výše na 20% u spodní hranice a na 40% u horní hranice. Výsledkem je interval $[0.2, 0.4)$. Další znak, r, omezí spodní hranici na 40% a horní hranici na 80%. Výsledný interval tedy

je $[0.28, 0.36)$. Znak a upraví interval na $[0.280, 0.296)$. Načtení znaku t vede k omezení intervalu na $[0.2928, 0.2960)$. Poslední načtený znak r omezí interval na $[0.29408, 0.29536)$.

Po zpracování vstupu se z výsledného intervalu vybere hodnota, která při uložení na výstup bude nejkratší. Při výběru hodnoty z intervalu se musí brát v potaz, že vybraná hodnota se bude ukládat binárně a tedy vybraná hodnota musí být nejkratší binárně. Jelikož výsledek bude vždy menší než 1, je možné z výsledného čísla vypustit část „0.“. V uvedeném příkladu byla vybrána hodnota „0.294921875“, ta bude binárně zakódována jako „010010111“. Při případné rekonstrukci samotný zakódovaný řetězec nestačí. Je ještě nutné uložit použitou abecedu, pravděpodobnost výskytu jednotlivých znaků abecedy a délku vstupního řetězce.

Dekomprese probíhá tak, že se načte abeceda, pravděpodobnosti a délka původního řetězce. Stanoví se stejný počáteční interval. Podle pravděpodobností se aktuální interval rozdělí na menší intervaly stejně jako při kódování. Následně dekódér načte zakódované číslo a převede jej z binární podoby do desítkové. Dekodér prochází jednotlivé intervaly a hledá, do kterého číslo spadá. Když je interval nalezen, dekodér uloží na výstup znak, kterému tento intrerval náleží. Dekodér poté přepočítá číslo podle vztahu:

$$NoveCislo \quad = \quad (Cislo - SpodHranZnak)/RozsahZnak$$

kde:

$NoveCislo$	je nová hodnota čísla,
$Cislo$	je aktuální hodnota čísla,
$SpodHranZnak$	je hodnota spodní hranice načteného znaku,
$RozsahZnak$	je rozsah intervalu načteného znaku

a následně opět prochází intervaly. Dekodér hledání intervalu a přepočet provádí tak dlouho, dokud není zrekonstruován řetězec o dané délce. V uvedeném příkladě by dekódování vypadalo následovně:

Číslo	**Znak**	**Přepočet**	**Nové číslo**
0.294921875	b	(0.294921875 - 0.2)/0.2	0.474609375
0.474609375	r	(0.474609375 - 0.4)/0.4	0.1865234375
0.1865234375	a	(0.1865234375 - 0)/0.2	0.9326171875
0.9326171875	t	(0.9326171875 - 0.8)/0.2	0.6630859375
0.6630859375	r	(0.6630859375 - 0.4)/0.4	0.6577148438

Aritmetické kódování může být také adaptivní. V takovém případě stačí, aby se provedl pouze jeden průchod přes vstupní řetězec. Adaptivní kódování pracuje na stejném principu jako statické, avšak pravděpodobnosti a dělení intervalů upravuje s každým zakódovaným znakem.

Range encoding

Range encoding je vylepšením aritmetického kódování[4]. Rozdíl oproti klasickému aritmetickému kódování je v tom, že Range encoding pracuje s čísly o jiném základu. Většinou pracuje se základem 256, což znamená, že místo práce s bity, pracuje s byty. Výhodou tohoto řešení je, že není třeba využívat binární operace. Díky tomuto bývá range encoding až 2-krát rychlejší než klasické aritmetické kódování pracující s celými čísly.

Kapitola 4

Paralelní verze kompresní metody BWT

4.1 Struktura implementace

PS3BWT je kompresní program založený na Burrows-Wheelerově transformaci (BWT). Ta je aplikována na vstup jako první, čímž se data transformují do formy, která při použití vhodného kódování napomůže k lepší kompresi. Existují ale i další transformace, které po aplikaci BWT dokáží výslednou kompresi ještě zlepšit.

Jednou z nich je například Move-To-Front transformace (MTF), která byla zmíněna v sekci 3.2.4. Ta dokáže změnit četnost výskytu znaků tak, že při zakódování se nejčetnější znak zakóduje i do 1bitu. V praxi se ukazuje, že tato metoda opravdu výrazně ovlivňuje výslednou velikost komprimovaného souboru. Jako další transformaci lze použít Run Length Encoding (RLE), který byl vysvětlen v sekci 3.2.5. PS3BWT RLE nevyužívá, protože podle naměřených hodnot v [1] se ukázalo, že RLE v kombinaci s MTF poskytuje v některých případech lepší výsledky než při použití pouze MTF a v ostatních případech výkazuje výsledky stejné případně horší.

K samotnému zakódování dat získaných z BWT a MTF transformací používá PS3BWT statické Huffmanovo kódování (sekce 3.3.1). Statické Huffmanovo kódování poskytuje sice o trochu horší kompresi než v případě adaptivního Huffmanova kódování (sekce 3.3.2), avšak je mnohem vhodnější k paralelizaci a tím pádem provedení této metody je podstatně rychlejší. Alternativou k Huffmanovu kódování je aritmetické kódování případně Range encoding, které poskytují lepší kompresní poměr než Huffmanovo kódování, nicméně kompresní poměr u Huffmanova kódování byl vyhodnocen jako dostatečný a navíc jak Aritmetické kódování, tak Range encoding jsou pomalejší při zpracování, a proto tyto metody kódování nebyly vybrány.

Pro každou úlohu, která se bude zpracovávat paralelně v SPU jednotkách je nutné předat potřebné informace do SPU jednotky. Při programování algoritmů pro IBM Cell je programátorovi umožněno přenést pouze jeden ukazatel do paměti do SPU jednotky, který bude následně jedním z parametrů funkce *main()* programu spouštěném v SPU jednotce. Proto ve všech případech je tento ukazatel naplněn adresou na strukturu, která nese potřebné informace jako jsou další ukazatele do hlavní paměti anebo důležité informace jako je velikost dat.

Aby byla komprese smysluplně účinná, je nutné vhodně zvolit velikost datových bloků,

které se budou zpracovávat. Samozřejmě velikost datového bloku by zase neměla být příliš velká, protože s velikostí datového bloku úměrně rostou i nároky na paměť. V případě programování aplikací pro IBM Cell procesor je také nutné, pokud pracujeme s SPU jednotkami, aby se při rozhodování o velikosti datového bloku bralo v potaz, kolik bytů je možno přenést z hlavní paměti do lokálního paměti v rámci jednoho čtení. A samozřejmě je také pro rozhodování důležitá samotná velikost lokální paměti SPU jednotky. Když jsem vzal v úvahu všechny tyto aspekty, rozhodl jsem se nastavit velikost datového bloku na 48 000B při použití sériové varianty BWT, 40 000B při použití paralelní varianty BWT, která pracuje s QuickSort třídícím algoritmem a 24 000B pro paralelní variantu BWT s využitím SA–IS třídícího algoritmu. Velikost přenášených dat v rámci jednoho čtení SPU jednotky jsem stanovil na 16 000B. Jak hodnotu velikosti datového bloku, tak maximální velikost dat v rámci jednoho čtení je možné měnit, avšak velikost přenášených dat v rámci jednoho čtení nesmí porušit podmínky pro úspěšný přenos, které byly zmíněny v sekci 2.1.

Zde je ukázka jedné z použitých struktur pro přenost informací z PPU jednotky do SPU jednotky. V tomto případě se jedná o strukturu přenášenou v rámci úlohy Huffmanova zakódování.

```
typedef struct huff_encode_io_data {
        /* ukazatel na vstupní data uložená v hlavní paměti */
        uintptr32_t             input;

        /* ukazatel do hlavní paměti pro výstupní data */
        uintptr32_t             output;

        /* ukazatel na tabulku kódů */
        uintptr32_t             codeTable;

        /* ukazatel na tabulku délek kódů */
        uintptr32_t             lengthTable;

        /* obsahuje hodnotu vyjadřující velikost datového bloku,
         * který je vstupem pro BWT */
        unsigned int            dataSize;

        /* je nositelem hodnoty vyjadřující jednoznačnou
         * identifikaci SPU jednotky */
        unsigned int            spe_id;

        /* velikost zakomprimovaných dat */
        unsigned int            compSize;

        /* Zarovnání dat */
        unsigned int            padding;
} huff_encode_io_data_t;
```

4.2 Zpracování vstupních dat

Při spuštění přebírá PS3BWT tři parametry. Vstupní soubor, výstupní soubor a mód zpracování. PS3BWT bere v potaz, že uživatel může požadovat zakomprimování souboru, jež by byl tak velký, že by při jeho zpracovávání došla zařízení, na kterém je PS3BWT pouštěn, paměť. Je zde tedy možnost nastavit, kolik datových bloků se bude maximálně najednou zpracovávat. PS3BWT lze použít na každém zařízení, které je založené na IBM Cell procesoru. Pokud bude PS3BWT přenesen na zařízení, které poskytuje více paměti než PS3, pak je vhodné změnit počet zpracovávaných bloků na vyšší číslo, díky čemuž se sníží celková velikost výstupu, protože se sníží počet řídích dat potřebných pro případnou dekompresi. PS3BWT podle nastavení načte příslušný počet bloků a zavolá funkci pro zpracování vstupu BWT metodou.

BWT může být provedeno dvěma způsoby, a to sériově nebo paralelně. Rozdíl je popsán v sekci 4.3. Po zpracování BWT získáme pro každý datový blok řetězec L. Tento řetězec je dále vstupem pro MTF transformaci, která je prováděna paralelně. Výstupem je opět řetězec. Ten slouží ke spočtení četnosti výskytu znaků pro statické Huffmanovo kódování. Ta je spočtena opět paralelně. Výsledek každého zpracovaného řetězce je připočten k celkovému součtu četností. Četnost dále slouží k vytvoření stromu pro Huffmanovo kódování. Za pomoci stromu s kódovou reprezentací jednotlivých znaků je provedeno kódování výstupních řetězců z MTF transformace. Zakódovaná data jsou uložena do výstupního souboru, čímž je úloha komprese dokončena.

PS3BWT umožňuje dva režimy výpisu statistik. Pokud je režim nastaven na „upovídaný“, pak po každé provedené úloze je vypsán čas, který byl potřebný k vykonání této úlohy. Není-li tento mód zapnut, PS3BWT vypíše pouze čas, který byl potřebný pro zpracování celého souboru.

4.3 Implementační detaily BWT

Princip provedení BWT byl vysvětlen v sekci 3.2. K BWT je potřeba použít vhodný třídící algoritmus. Nejčastěji bývá použit některý z Suffix Array algoritmů. PS3BWT nabízí dvě možnosti zpracování vstupu. Jednou je použití algoritmu Suffix Arraye SA-IS, druhou pak použití nerekurzivního QuickSortu, který si dokáže držet index znaku v řetězci.

4.3.1 SA–IS

Využití SA-IS vede k dobrým výsledkům komprese. Problémem však je jeho paměťová náročnost, která při nejhorší možné situaci bude $O(6.25n)$. Vzhledem k této situaci dává PS3BWT uživateli 2 možnosti. Pokud zvolí datový blok větší než 24 000B, pak z důvodu výše zmíněné paměťové náročnosti nelze SA-IS počítat v SPU jednotkách a výpočet se provádí v PPU jednotce. Pokud ovšem uživatel zvolí datové bloky menší velikosti, pak PS3BWT automaticky zvolí paralelní zpracování BWT. Jednotlivé bloky se tedy předají SPU jednotkám přes ukazatele umístěné v struktuře, jak bylo probráno v sekci 4.1.

Nevýhodou použití paralelní varianty je, že při paralelním zpracování se musí použít datové bloky menší velikosti a tím pádem je nutné zpracovávat více datových bloků. To má za následek zpomalení všech ostatních metod, které jsou pro vykonání komprese použity a jsou vykonány také paralelně. Další nevýhodou je výsledná velikost souboru. To

zapřičiňuje použitá velikost datového bloku. Rozdíl výsledné velikosti je však velmi malý s porovnáním rozdílu velikostí v případě použití QuickSortu, který je popsán níže.

4.3.2 QuickSort

Původní myšlenkou bylo využít variantu s QuickSortem v případě, kdy uživatel požaduje, aby byla komprese provedena co nejrychlejším způsobem. Vzhledem k vlastnostem QuickSortu není totiž tento algoritmus příliš vhodný k použití v kombinaci s BWT. BWT totiž vyžaduje, aby třídící algoritmy tvořily řetězec L způsobem, který je popsán v sekci 3.2.1 a QuickSort toto úplně nedodržuje. To má za následek, že permutace, které by měly být blízko sebe blízko nebudou, čímž se odstraní efekt BWT. Tím se výrazně sníží kompresní poměr a výstupní soubor je tak větší než v případě použití SA-IS. Zvýši se tím i doba vykonávání komprese, protože ostatní úlohy pak mají větší práci s daty, jak je popsáno v sekci 4.6.1. V konečném důsledku je pak metoda rychlejší jen v některých případech.

Výhodou QuickSort je jeho paměťová nenáročnost, díky níž je možné přenést do SPU jednotek větší blok dat než v případě paralelní varianty SA–IS.

4.3.3 Paralelní zpracování

Paralelní zpracování BWT vyžaduje vyšší režii, protože zde dochází k vytváření kontextu pro SPU jednotky, vytvoření struktury, která je nositelem potřebných proměnných a ukazatelů na data v hlavní paměti a po dokončení úlohy vyčištění SPU jednotky. Po dokončení operace v SPU jednotkách není nutné výstup nijak zpracovávat, protože je SPU jednotkou odeslán přímo do hlavní paměti. PS3BWT se snaží vždy využít maximální počet SPU jednotek, které jsou právě k dispozici. Pokud se vstupní soubor musí rozdělit do více bloků než kolik je dostupných SPU jednotek, pak po dokončení úlohy a vyčištění SPU se znovu vytvoří požadovaný kontext a úloha se provede znovu s novým datovým blokem. Toto se provádí tak dlouho, dokud jsou na vstupu nezpracované datové bloky. Jelikož při třídění musí být všechna data k dispozici, není možné využít vlastnosti SPU jednotek, kdy je možné provádět výpočet nad nějakými daty a čtení jiných dat z hlavní paměti do lokální paměti. SPU jednotka tedy musí nejdříve provést příslušný počet čtení a až poté provést třídění.

4.3.4 Postup

Ať už se provádí BWT s využitím SA-IS nebo QuickSortu, postupuje se vždy tak, že nejdříve se provede třídění. Poté se prochází seřazené permutace a od každé se uloží její poslední znak do nového řetězce. Tím získáme výsledek BWT metody, který bude dále sloužit jako vstup pro Move-To-Front transformaci. Velmi důležité také je uložit si index, na kterém se v seřazené matici permutací nachází originální řetězec. Bez tohoto údaje by případná dekomprese nebyla možná.

4.3.5 Paralelní SA–IS s využitím komunikace pomocí zpráv

Měření, která byla provedená za účelem získání informací o času stráveném v jednotlivých metodách, které je diskutováno v sekci 4.6, ukázalo, že režie pro práci s kontextem SPU jednotek je příliš časově náročná. Proto byla paralelní varianta BWT s použitím SA–IS třídícího algoritmu upravena tak, aby se vytvořil kontext pro každou SPU jednotku, pokud jich bude třeba využít maximální počet. Pak už se kontext vytvářet nebude. SPU jednotka dokončí úlohu a informuje PPU jednotku zasláním zprávy o svém dokončení. Pokud je

třeba zpracovat další bloky, PPU jednotka následující blok připraví a zašle SPU jednotce zprávu o tom, že má na vstupu další data. V opačném případě pošle PPU jednotka zprávu, na kterou SPU jednotka reaguje svým ukončením a PPU jednotka provede zničení kontextu. Tabulka 4.3 v sekci 4.6 prokázala, že tato úprava opravdu vedla u větších souborů k rychlejšímu zpracování.

4.4 Fáze Move-To-Front

Transformace je prováděna paralelně v SPU jednotkách. Každé SPU jednotce je přidělen jeden datový blok, který je výstupní řetězec BWT transformace. Pokud je datových bloků více než dostupných SPU jednotek, pak po dokončení zpracování datového bloku je SPU jednotce předán blok nový a proces se opakuje. SPU jednotka dostane ukazatel na strukturu obsahující ukazatele do paměti, kde se nachází vstup a kam se má uložit výstup. Navíc dostane údaj o tom, kolik bytů má vstup. Výstup MTF metody je zároveň vstupem pro spočtení četnosti znaků, která bude dále použita v Huffmanově kódování.

Program v SPU jednotce využívá vlastnost, kdy je možné zažádat o příjem dat z hlavní paměti a než jsou tato data načtena do lokální paměti (LS), SPU jednotka může provádět úlohy, které nepotřebují načítaná data. Dále taky program využívá pro samotnou transformaci vektorových instrukcí, které urychlují zpracování dat.

Jako první si SPU jednotka načte obsah struktury z hlavní paměti. Z něj si vezme ukazatel na vstup a požádá o načtení dat z hlavní paměti z dané adresy. Jak bylo zmíněno v sekci 2.1, SPU jednotka dokáže načíst maximálně 16 384B. Proto pokud jsou vstupní data větší, je nutné provést více čtení. Program je ovšem neprovádí za sebou. Program nejdříve načte prvních 16 000B a následně zažádá o načtení dalších vstupních dat. Mezitím, co se načítají data do bufferu, program provádí nad již načtenými daty z minulého čtení MTF transformace. Po dokončení transformace program odešle zpracovaná data na adresu uloženou v ukazateli pro výstup. Následně program zkontroluje, zda načítaná data jsou již přečtená. Je-li tomu tak, pak v případě potřeby požádá o další čtení dat a nad načtenými daty je opět provedena transformace. To se opakuje tak dlouho, dokud není celý vstup zpracován. Pokud je vstup kratší, provádí se pouze jedno čtení dat, provede se transformace a výsledek transformace je odeslán do hlavní paměti. Po dokončení úlohy již nejsou data v hlavní paměti nijak upravována.

4.5 Huffmanovo kódování

4.5.1 Četnost znaků pro Huffmanovo kódování

Tato úloha se opět provádí v SPU jednotkách. SPU jednotce je předán ukazatel na strukturu obsahující ukazatele na vstup a výstup a také obsahuje velikost vstupních dat. Program načte strukturu z hlavní paměti. Obdobně jako u MTF transformace PS3BWT využívá práci s daty během načítání dalších vstupních dat. Vektorové instrukce zde využity nejsou. Po načtení struktury se vezme ukazatel na vstupní data a požádá se o načtení vstupních dat z hlavní paměti do LS. Po načtení těchto dat je podána v případě potřeby další žádost o načtení dat, která se budou načítat do bufferu. Poté jsou předešlá načtená data zpracována. Proces je prováděn tak dlouho, dokud není zpracován celý vstup. Po zpracování vstupu jsou výstupní data odeslána do hlavní paměti. Když je úloha v SPU jednotkách dokončena PPU jednotka vezme výstupní data z dané SPU jednotky a připočte je do globálního pole

obsahující četnosti jednotlivých znaků. Tím je získána četnost pro každý znak ve všech zpracovávaných blocích. Z této četnosti je následně vytvořen strom pro Huffmanovo kódování.

4.5.2 Strom Huffmanova kódování

Ze získané četnosti výskytu znaků je vytvořen strom Huffmanova kódování. Ten je tvořen v PPU jednotce, protože se jedná o úlohu, kterou nelze rozumně paralelizovat. Ze stromu se dále vytvoří tabulka kódů. Každý index do této tabulky reprezentuje ordinální hodnotu znaku v ASCII. Pomocí tabulky kódu se vytvoří další tabulka, která obsahuje velikost jednotlivých kódů. Ta poslouží k rychlému získání délky kódu, protože opakovaně počítat délku kódu by bylo neefektivní. Když jsou tabulky vytvořeny a inicializovány, je možné započít úlohu samotného zakódování dat.

4.5.3 Zakódování souboru pomocí Huffmanova kódování

PPU jednotka

Zakódování probíhá z velké části opět paralelně. Jednotkám je předán ukazatel na strukturu obsahující ukazatele na vstupní datový blok, jimž je opět výstup z MTF transformace, a také na výstup. Dále ovšem ale také obsahuje ukazatele na tabulku kódů a na tabulku délek kódů. Jako poslední jsou ve struktuře položky obsahující velikost dat před kompresí a velikost dat po kompresi. Velikost dat po kompresi je naplněna až SPU jednotkou. Pokud je vstupních datových bloků více než dostupných SPU jednotek, pak po dokončení operací s datovým blokem je SPU jednotce předán nový datový blok a proces je zopakován.

Zpracování vstupu SPU jednotkami

SPU jednotka stejně jako v ostatních případech nejříve načte strukturu z hlavní paměti. Dále jsou načteny tabulky. Až poté se načte vstup případně čast vstupu do velikosti 16 000 bytů. Pokud je velikost vstupu větší než 16 000 bytů, pak se v průběhu programu data donačítají ve chvíli, kdy předchozí data byla již zpracována. Při samotném kódování jsou vstupní data procházená po znacích. Vždy se načte znak a jeho ordinální hodnota je použita jako index do tabulky kódu. Kód se uloží do pomocné proměnné. Dále se pomocí této hodnoty indexuje do tabulky délky kódů, hodnota se také uloží do pomocně proměnné. V konečném cyklu o velikosti délky kódu se pak kód zapisuje pomocí bitových operací do proměnné o velikosti jednoho znaku. Pokud je už v proměnné uložen předchozí znak případně jeho část a délka aktuálně zpracovávaného kódu by byla větší než zbylý prostor pro uložení kódu do proměnné, uloží se část kódu do proměnné tak, aby byla plně využita, ta se uloží do výstupního řetězce, vyčistí se a uloží se zbytek kódu či jeho další část. Tento proces probíhá tak dlouho, dokud se nezakódují všechna vstupní data. Při každem zapsaném bytu do výstupního řetězce se inkrementuje počitadlo. Hodnota počitadla po zpracování celého vstupu pak vyjadřuje, kolik bytů je potřeba pro zapsání zakódovaného řetězce. Tuto hodnotu při dokončení zakódování v SPU jednotce, předá SPU jednotka do PPU jednotky. Při zakódovávání může dojít k situaci, kdy výstup přesáhne hodnotu 16 000 bytů. V takovém případě po každém dosažení 16 000 bytů na výstupu se výstup odešle do hlavní paměti, posune se ukazatel, výstupní buffer se vyčistí a kódování pokračuje dál.

Dokončení úlohy – PPU jednotka

Když SPU jednotka dokončí kódování, PPU jednotka převezme ukazatel na výstupní data a velikost těchto dat. Následně provede uložení výstupních dat do souboru. Jedná se o místo, kde je při této úloze spotřebováno nejvíce času, avšak SPU jednotky nemají možnost komunikovat s okolím, pouze s PPU jednotkou a s ostatními SPU jednotkami, a tedy nemohou zapisovat do souboru. Po zpracování všech datových bloků je provedena kontrola, zda se zpracoval již celý soubor nebo je-li třeba celý proces opakovat. V případě, že vstupní soubor ještě celý zpracován nebyl, vyčistí se pole vstupních řetězců, pole výstupních řetězců, pole indexů na originální řetězec a počitadlo datových bloků je nastaveno na 0. Po vyčištění se proces komprimace opakuje. Když je celý soubor zpracován, je uvolněna veškerá zabraná paměť, uzavřeny vstupní a výstupní soubory a PS3BWT ukončen.

4.6 Měření a statistiky

Měření bylo prováděno na těchto variantách PS3BWT:

- Sériové zpracování BWT za využití SA–IS a velikostí datového bloku 48 000B
- Paralelní zpracování BWT za využití SA–IS a velikostí datového bloku 24 000B
- Paralelní zpracování BWT za využití QuickSort a velikostí datového bloku 40 000B

Velikost datového bloku byla u každé varianty volena tak, aby maximálně využila LS paměť v SPU jednotkách. Při měření se zjišťovalo, jak rychlé jsou jednotlivé variaty a jak velký bude výstupní soubor v případě jejich použití. Výsledky měření pro zjištění rychlostí jednotlivých variant jsou vypsány v tabulce 4.2 a výsledky pro zjištění velikosti zakódovaného souboru lze nalézt v tabulce 4.5. Tabulka obsahuje 4 sloupce, kdy první obsahuje použité soubory a zbylé obsahují naměřené hodnoty u dané varianty. Každá varianta je specifikována třídicím algoritmem, zda byla prováděná sériově či paralelně a jak velký datový blok byl použit. V tabulce je v každém řádku vyznačena varianta, která byla provedena nejrychleji či zakódovala soubor na nejmenší velikost. Nejrychlejší a nejúčinější metoda je vyznačena tučným písmem. Výsledky měření jsou váženým průměrem 3 běhů jednotlivých variant u daného souboru.

K testování variant PS3BWT byly vybrány tyto soubory:

The Artificial Corpus	**Kategorie**	**Velikost [B]**
a.txt	Znak 'a'	1
aaa.txt	Znak 'a' 100 000krát opakovaný	100000
alphabet.txt	Opakující se znaky abecedy	100000
random.txt	100 000 znaků náhodně vygenerovaných	100000
The Calgary Corpus	**Kategorie**	**Velikost [B]**
bib	Bibliografie	111261
book1	Kniha	768771
book2	Kniha	610856
news	USENET batch soubor	377109
paper1	Technický dokument	53161
paper2	Technický dokument	82199
progc	Zdrojový kód v C	39611
progl	Zdrojový kód v LISP	71646
progp	Zdrojový kód v Pascalu	49379
The Canterbury Corpus	**Kategorie**	**Velikost [B]**
alice29.txt	Anglický text	152089
asyoulik.txt	Shakespeare	125179
cp.html	HTML kód	24603
fields.c	Zdrojový kód v C	11150
grammar.lsp	Zdrojový kód v LISP	3721
lcet10.txt	Technické psaní	426754
plrabn12.txt	Poezie	481861
sum	SPARC	38240
Xargs.1	GNU manuál	4227
The Large Corpus	**Kategorie**	**Velikost [B]**
bible.txt	Bible	4047392
world192.txt	Kniha o CIA faktech	2473400
Ostatní	**Kategorie**	**Velikost [B]**
etext99	Soubor elektronických knih	105277340

Tabulka 4.1: Soubory použité pro měření

4.6.1 Rychlost

The Artificial Corpus	**SA-IS Sériově 48K**	**SA-IS Paralelně 24K**	**QuickSort Paralelně 40K**
aaa.txt	0.071sec	0.056sec	**0.046sec**
alphabet.txt	0.065sec	0.065sec	**0.054sec**
a.txt	**0.017sec**	**0.017sec**	0.018sec
random.txt	0.170sec	**0.113sec**	0.124sec
The Calgary Corpus			
bib	0.110sec	**0.087sec**	0.109sec
book1	0.534sec	0.397sec	**0.368sec**
book2	0.410sec	0.319sec	**0.301sec**
news	0.294sec	0.220sec	**0.215sec**
paper1	0.081sec	**0.063sec**	0.087sec
paper2	0.096sec	**0.074sec**	0.094sec
progc	**0.065sec**	**0.065sec**	0.084sec
progl	0.078sec	**0.060sec**	0.083sec
progp	0.067sec	**0.058sec**	0.087sec
The Canterbury Corpus			
alice29.txt	0.136sec	0.126sec	**0.112sec**
asyoulik.txt	0.125sec	**0.091sec**	0.109sec
cp.html	**0.048sec**	0.055sec	0.059sec
fields.c	**0.030sec**	0.033sec	0.036sec
grammar.lsp	0.022sec	**0.018sec**	0.024sec
lcet10.txt	0.307sec	**0.223sec**	0.235sec
plrabn12.txt	0.354sec	0.268sec	**0.257sec**
sum	0.070sec	**0.065sec**	0.086sec
Xargs.1	**0.023sec**	0.024sec	0.025sec
The Large Corpus			
bible.txt	2.457sec	1.795sec	**1.636sec**
world192.txt	1.553sec	1.138sec	**1.085sec**
Misc			
etext99	68.401sec	50.068sec	**49.415sec**

Tabulka 4.2: Naměřené rychlosti jednotlivých variant

Tabulka 4.2 ukazuje, že ve většině případů je soubor nejrychleji zpracován pomocí paralelní varianty BWT. Zrychlení je nejvíce viditelné u větších souborů. Přestože při měření bylo využíváno všech 6 dostupných SPU jednotek, byl nárust rychlosti zpracování průměrně pouze 37%. Bylo provedeno plno měření, které mělo osvětlit, proč není paralelní zpracování ještě rychlejší.

Vysoká režie operací pro práci s kontextem pro SPU jednotky

Jeden z důvodů je poměrně vysoká režie při vytváření, nahrávaní a rušení kontextu pro SPU jednotku. Tyto 3 operace jsou provedeny průměrně za 1.7ms. Důležitým faktem je, že

tyto operace jsou prováděny v PPU jednotce a tedy jsou jako celek prováděny sériově pro každou SPU jednotku. Pokud chceme zakódovat například `etext99`, který je při paralelním zpracování rozdělen na 2194 datových bloků o maximální velikosti 48 000B, znamená to, že se musí 2194krát vytvořit, nahrát a následně zrušit kontext pro každou SPU jednotku. Zpoždění oproti sériovému zpracování, kde tato režie není potřebná, je 3729.8ms, tedy 3.73s. Proto varianta paralelního zpracování BWT s využitím SA–IS byla experimentálně upravena tak, aby se tyto operace provedly maximálně tolikrát, jako je počet dostupných SPU jednotek. SPU jednotky tak pobeží nepřetržitě a když jsou nová data na vstupu připravena ke zpracování, bude SPU jednotka vyrozumněna zprávou. Ta úlohu vykoná a následně informuje PPU jednotku o dokončení úlohy. PPU jednotka nečeká na to, až bude zpráva vyřízená a přejde rovnou k čekání na zprávu od jedné z SPU jednotek, že dokončila úlohu a je připravena na zpracování dalšího bloku dat. Tabulka 4.3 porovnává rychlosti variat bez použití zpráv a s využitím zpráv.

Soubor	**SA-IS Paralelně 24K**	**SA-IS Paralelně 24K MSG**
a.txt	**0.017sec**	0.018sec
progp	**0.058sec**	0.059sec
Xargs.1	**0.024sec**	0.025sec
bible.txt	1.795sec	**1.729sec**
worl192.txt	1.138sec	**1.105sec**
etext99	50.068sec	**46.718sec**

Tabulka 4.3: Naměřené rychlosti variant SA–IS a SA–IS s posíláním zpráv

Tabulka 4.3 ukázala, že k zrychlení došlo u větších souborů. Naopak ke zpomalení většinou došlo u menších souborů. Je to způsobeno tím, že u souborů do 6 datových bloků, kdy tedy budou obsazeny všechny SPU jednotky, nemůže varianta s posíláním zpráv dosáhnout vyšší efektivity, protože se operace pro práci s kontextem SPU jednotek musí vytvořit. Naopak je ještě nutné po dokončení úloh v SPU jednotkách provést komunikaci pomocí zpráv, aby SPU jednotka byla informována o tom, že na vstupu nejsou žádná další data a má tedy ukončit svoji činnost.

Čtení dat z hlavní paměti do LS

Další zpoždění způsobuje čtení či zápis dat z/do hlavní paměti do/z LS. Při práci s 24 000B datovým blokem se zpoždění dostane na hodnotu asi 2ms. Jelikož se tyto operace provádějí v SPU jednotce, zpožďuje se pouze vykonání samotné úlohy. Pokud ale mnoho SPU jednotek žádá o data, může dojít k zahlcení plánovače DMA přenosů.

Vektorový procesor SPU

Vliv na zpoždění má však také samotná SPU jednotka. BWT vyžaduje provedení mnoho různých skalarních operací nad daty umístěnými na různých místech v datovém bloku. SPU jednotka je naopak zaměřena na vysokorychlostní vykonání vektorových operací, které se provádí nad souvislými bloky dat. Díky tomu SPU jednotka ztrácí svůj výkon. Naopak PPU jednotka je procesorem pro obecné použití a použití skalárních instrukcí tak nepředstavuje

ztrátu výkonu. Měření ukázalo, že BWT s využitím SA–IS nad datovým blokem o velikosti 24 000B je v PPU jednotce dokončena za zhruba 9ms. Naopak stejná úloha se stejně velkým objemem dat je v SPU jednotce dokončena za zhruba 16ms, pokud odečteme 2ms za zmíněné vykonání vstupně/výstupních operací, bude úloha vykonána o 5ms později. Na souboru `etext99`, kde by se muselo použít 4387 datových bloků o velikosti 24 000B, mohlo by se zpoždení díky pomalejšímu zpracování v SPU jednotce teoreticky vyšplhat až na 21940ms, tj. 21,9s.

Velikost datového bloku

Velikost bloku je dalším faktorem, který může vést k delšímu vykonání úlohy. Sériové zpracování dat pomocí BWT, jelikož má k dispozici přímý přístup k hlavní paměti, pracuje s největší velikostí datového bloku. Naopak paralelní varianta BWT s využitím SA–IS používá datové bloky o poloviční velikosti. Tím pádem SPU jednotky musí zpracovat 2krát více bloků, než PPU jednotka a dochází k vyšší režii spojené s operacemi starajícími se o kontext pro SPU jednotky. Jelikož paralelní BWT pracuje s menšími bloky, musí se stejně velkými bloky pracovat i ostatní metody. Ostatní metody tak nejsou schopny pracovat s maximální efektivitou.

Velikost datového bloku ovlivňuje samotnou efektivitu třídění dat. Čím větší blok je, tím efektivnější pak bude komprese. U menších bloků tak nemusí být stejné znaky vedle sebe tak často jako v případě větších bloků. Tím dojde k delšímu vykonávání MTF úlohy, protože se budou častěji posouvat znaky na začátek abecedy a navíc nemusí dojít k tak jednoznačnému navýšení četnosti nejčastěji se vyskytujících znaků. Výsledek MTF má přímý vliv na Huffmanovo kódování. Čím různorodější je výstup MTF kódování, tím déle bude kodéru trvat zakódování. Toto chování kompresního algoritmu lze nejvíce vypozorovat při použití BWT metody s využitím QuickSort třídícího algoritmu jež zachycuje tabulka 4.4.

Úloha	**SA-IS Sériově 48K**	**SA-IS Paralelně 24K**	**QuickSort Paralelně 40K**
BWT	9.1ms	16.3ms	6.4ms
MTF	4.7ms	3.8ms	5.2ms
Spočtení četnosti výskytů	2.0ms	2.1ms	2.7ms
Vytvoření binárního stromu	5.5ms	4.8ms	4.9ms
Zakódování dat	20.8ms	21.9ms	36ms

Tabulka 4.4: Naměřená doba vykonávání jednotlivých úloh

SA–IS vs. QuickSort

QuickSort je sám o sobě rychlejší třídící algoritmus než SA–IS. Nicméně jak bylo zmíněno v sekci 4.3.2, není příliš vhodný pro použití v kombinaci s BWT. Nastává tak efekt popsaný v sekci 4.6.1. Jak ukazuje tabulka 4.2, u souborů větší velikosti je QuickSort nejrychlejší variantou, ovšem u souborů menších bývá v některých případech rychlejší varianta s využitím SA–IS.

4.6.2 Velikost zakódovaného souboru

The Artificial Corpus	SA-IS Sériově 48K	SA-IS Paralelně 24K	QuickSort Paralelně 40K
aaa.txt	**13075B**	13091B	13083B
alphabet.txt	**13154B**	13228B	18824B
a.txt	522B	522B	522B
random.txt	**76059B**	76197B	76048B
The Calgary Corpus			
bib	**37721B**	40558B	69812B
book1	**311656B**	331199B	452322B
book2	**228760B**	228760B	361757B
news	**154392B**	157615B	241787B
paper1	**19120B**	20334B	32091B
paper2	**30250B**	32219B	48677B
progc	**14236B**	14899B	25727B
progl	**19926B**	20412B	36753B
progp	**13603B**	14349B	29692B
The Canterbury Corpus			
alice29.txt	**54434**	59400B	91941B
asyoulik.txt	**48780B**	51139B	75457B
cp.html	**9249B**	9389B	15693B
fields.c	**3926B**	**3926B**	6258B
grammar.lsp	**1890B**	**1890B**	2510B
lcet10.txt	**149274B**	154688B	249199B
plrabn12.txt	**205458B**	205458B	282341B
sum	**14326B**	15051B	24089B
Xargs.1	**2341B**	**2341B**	2953B
The Large Corpus			
bible.txt	**1244653B**	1313365B	2473400B
world192.txt	**839267B**	936147B	1528752B
Misc			
etext99	**39709238B**	41882394B	61369225B

Tabulka 4.5: Naměřené velikosti zakódovaných souborů jedotlivých variant

Výsledky velikosti zakódovaného souboru mnohem více odpovídají předpokladu než tomu bylo u výsledků měřených rychlostí. Takulka velikostí potvrdila předpoklad, že nejmenší zakódované soubory bude poskytovat varianta s využitím největších datových bloků a s použitím třídícího algoritmu, který nejvíce splňuje pravidlo pro třídění řetězce L v sekci 3.2.1. V případě PS3BWT se jedná o variantu sériového provedení BWT, která provádí tuto transformaci nad daty o maximální velikosti 48 000B dat. Za ní následuje její paralelní varianta. Varianta využívající QuickSort poskytuje nejhorší kompresi. Soubory `a.txt`, `fields.c`, `grammar.lsp` a `Xargs.1` ukazují, že pokud bude velikost původního souboru menší či rovna 24 000B, pak bude sériová i paralelní varianta SA–IS BWT generovat stejně velké zakódované soubory. Soubor `a.txt` také ukazuje minimální možnou velikost výstup-

ního souboru. Naopak soubor `aaa.txt` ukazuje, že velikost datového bloku má vliv na výslednou velikost výstupního souboru. U sériového BWT a BWT s použitím QuickSortu však používají datové bloky o takové velikosti, že u obou variant by došlo k rozdělení tohoto souboru na 3 bloky a tak by měl být výsledný soubor stejně veliký. QuickSort ale díky své nestabilitě nenechá při třídění datového bloku první znak na své pozici, ale posune ho někam hlouběji. Jeho konečná pozice je pak uložena do výstupního souboru, aby bylo možné zakódovaný soubor případně rozkódovat. Jelikož pozice může mít několikacifernou hodnotu, dochází tak ke zvětšení výstupního souboru.

Kapitola 5

Závěr

IBM Cell procesor přináší mnoho nových věcí, které je programátor nucen si osvojit, pokud chce pracovat s tímto procesorem maximálně efektivně. Jedná se hlavně o vektorový přístup k datům, komunikaci mezi PPU a SPU jednotkami a zajištění doručení dat SPU jednotce z hlavní paměti. Největší změny ve vývoji pro PS3 bylo možné pozorovat u samotných her. V době vydání PS3 sice hry vypadaly dobře, ale nedokázaly využít potenciálu procesoru. Postupem času, kdy vývojáři více a více rozuměli samotné PS3, se lepšila i vizuální stránka a mechanizmy her. Obdobně tomu bylo u tvorby PS3BWT. Po naimplementování a naměření hodnot, jsem mnohem lépe pochopil, jak IBM Cell pracuje a své poznatky na vylepšení popisuji níže. Vývoji pro IBM Cell také hodně pomohl přechod z `libspe` knihovny na `ALF` knihovnu, která byla představena v IBM Cell SDK 3.0. Ta výrazně ulehčila programátorovi práci s přenosem dat do SPU jednotky. Její prvotní pochopení je však poměrně náročné.

PS3BWT je kompresní algoritmus založený na BWT transformaci, kdy jsou převážně využívány skalární operace. Přestože IBM Cell architektura není přímo uzpůsobená na skalární operace, hlavně SPU jednotky, které jsou přímo konstruované na vysokorychlostní vektorové výpočty, dokáže tento procesor provést tento algoritmus v relativně rozumném čase, jak ukazují výsledky měření v sekci 4.6.

PS3BWT poskytuje mnoho prostoru pro vylepšení. Při měření se ukázalo, že režie spojená s přípravou SPU jednotky pro vykonání úlohy je časově velice náročná. Z tabulky 4.3 vyplývá, že pokud se využije mechanismu, kdy bude příprava SPU jednotky provedena maximálně jednou u každé jednotky a jednotka poběží tak dlouho, dokud bude mít na vstupu data, dojde k velké časové úspoře. Na souboru `etext99` se jednalo až o několik vteřin. Další prací na PS3BWT by tedy bylo převést jednotlivé úlohy na daný mechanismus. Pro opravdu malé soubory by však převedení úloh na tento mechanismus mohl mít opačný efekt a režie spojená s přípravou SPU jednotky by nevymizela. V takových případech by bylo vhodnější použít sériový přístup.

Dalším námětem na vylepšení by mohlo být převedení třídícího algoritmu tak, aby v maximální možné míře využíval vektorový přístup. Jedná se o velice náročný proces, hlavně v případě SA–IS, a proto převedení v rámci této práce nebylo provedeno. Nicméně toto převedení by mohlo přinést značné urychlení.

Správnou úpravou MTF úlohy by bylo možné v této úloze provádět rovnou spočtení četnosti výskytu jednotlivých znaků. Tím by došlo také k poměrně velké časové úspoře.

Poměrně zajímavou myšlenkou je použití úplně jiného přístupu k rozdělení SPU jednotek. Pokud by byla provedená zmíněná úprava o přesunutí výpočtu četnosti do MTF úlohy, dojde ke snížení počtu úloh prováděných v SPU jednotkách na 3. V případě provedení proudové komprese, kdy jedna SPU jednotka by prováděla BWT, druhá MTF a třetí adaptivní Huffmanovo kódování, pak se režie s obsluhou SPU jednotek sníží na minimum. PPU jednotka by se pouze starala o přísun dat do SPU jednotky, která provádí BWT a o uložení zakódovaných dat. Tímto přístupem bychom ovšem využili pouze 3 SPU jednotky. Jelikož jich má ale v PS3 programátor k dispozici 6, je možné provádět kompresi ve dvou proudech a proces komprese tak ještě urychlit. PPU jednotka avšak musí v takovém případě ještě zajistit synchronizaci uložení dat, aby nedošlo k jejich nekonzistenci.

Ovšem největším nedostatkem pro provedení paralelní komprese na PS3 je velikost paměti v SPU jednotkách. Bzip2, program pracující také na základě BWT, používá jako minimální velikost datového bloku 100kB. Maximem je pak blok o velikosti 900kB. Na PS3 není možné při paralelním zpracování použít ani polovinu minimální velikosti datového bloku u Bzip2 a tak PS3BWT nespokytuje tak dobrou kompresi. Větší počet bloků také s sebou nese vyšší režii na řízení SPU jednotek a celá komprese se tak zpomaluje.

Aplikace výše zmíněných úprav vede k razantnímu zrychlení komprese a využití potenciálu PS3 by bylo mnohem vyšší. Aplikace těchto úprav je však časově náročnou záležitostí a tedy nebylo možné je v rámci této práce provést.

Literatura

[1] Adjeroh, D.; Bell, T.; Mukherjee, A.: *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. New York, NY 10013, USA: Springer Science+Business Media, LLC, ISBN 978-0-387-78908-8.

[2] Bentley, J. L.; Sleator, D. D.; Tarjan, R. E.; aj.: A Locally Adaptive Data Compression Scheme. *Communications of the ACM* [online]. 1986, roč. 29, č. 4 [cit. 2013-05-04]. ISSN 0001-0782.
URL
http://www.cs.cmu.edu/~sleator/papers/adaptive-data-compression.pdf

[3] Burrows, M.; Wheeler, D. J.: A Block-sorting Lossless Data Compression Algorithm. *SRC Research Report* [online]. May 10, 1994, č. 124 [cit. 2013-05-04].
URL ftp://gatekeeper.research.compaq.com/pub/dec/SRC/research-reports/SRC-124.pdf

[4] Campos, A. S. E.: Range coder [online]. 1999 [cit. 2013-05-03].
URL http://www.arturocampos.com/ac_range.html

[5] Faller, N.: An Adaptive System for Data Compression. *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*: s. 593–597.

[6] Gallager, R. G.: Variations On a Theme By Huffman. *IEEE TRANSACTIONS ON INFORMATION THEORY* [online]. November 1978, roč. 24, č. 6, s. 668-674 [cit. 2013-05-04].
URL http://web.mit.edu/6.441/www/reading/IT-V24-N6.pdf

[7] Gottlieb, D.; Hagerth, S. A.; Lehot, P. G. H.; aj.: A classification of compression methods and their usefulness for a large data processing center. *Proceeding AFIPS 1975*: s. 453–458.

[8] Huffman, D. A.: A Method for the Construction of Minimum-Redundancy Codes. In *PROCEEDINGS OF THE I.R.E.* [online], 1952, s. 1098–1101 [cit. 2013–05–03].
URL http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf

[9] Knuth, D. E.: Dynamic Huffman Coding. *Journal of Algorithms*, ročník 6, č. 2: s. 163–180.

[10] Manber, U.; Myers, G.: Suffix arrays: a new method for on-line string searches. *In Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms* [online], 1990 [cit. 2013-05-03].

URL http://www.cs.washington.edu/education/courses/cse590q/00au/papers/manber-myers_soda90.pdf

[11] Manzini, G.; Ferragina, P.: Engineering a lightweight suffix array construction algorithm. *Algorithmica* [online]. 2004, roč. 40, č. 1, s. 33-50 [cit. 2013-05-04]. DOI: 10.1007/s00453-004-1094-1.
URL http://ilex.iit.cnr.it/manzini/papers/Algorithmica04.pdf

[12] Nong, G.; Zhang, S.; Chan, W. H.: Two Efficient Algorithms for Linear Suffix Array Construction. *IEEE Transactions on computers* [online], 2011 [cit. 2013-05-03], ISSN 0018-9340.
URL http://www.cs.sysu.edu.cn/nong/index.files/Two%20Efficient%20Algorithms%20for%20Linear%20Suffix%20Array%20Construction.pdf

[13] Pasco, R. C.: Source Coding Algorithms for Fast Data Compression [online]. May 1976 [cit. 2013-05-03].
URL http://www.ic.tu-berlin.de/fileadmin/fg121/Source-Coding_WS12/selected-readings/Pasco__1976.pdf

[14] Rissanen, J. J.: Generalized Kraft Inequality and Arithmetic Coding. *IBM Journal of Research and Development*, ročník 20, č. 3: s. 198–203.

[15] Salomon, D.; Motta, G.: *Handbook of Data Compression*. Springer-Verlag London Limited, ISBN 978-1-84882-902-2.

[16] Scarpino, M.: *Programming the Cell Processor*. Boston, MA 02116, USA: Pearson Education, Inc, ISBN 978-0-13-600886-6.

[17] Weiner, P.: Linear Pattern Matching Algorithms. *14th Annual IEEE Symposium on Switching and Automata Theory* [online], 1973 [cit. 2013-05-03].
URL http://airelles.i3s.unice.fr/files/Weiner.pdf

[18] Ziv, J.; Lempel, A.: A Universal Algorithm for Sequential Data Compression. *IEEE TRANSACTIONS ON INFORMATION THEORY* [online]. May 1977, roč. 23, č. 3, s. 337-343 [cit. 2013-05-04].
URL http://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1977_universal_algorithm.pdf

Příloha A

Obsah DVD

Corpus/
  corpus.zip
Src/
  PS3BWT.c
  spu_BWT_T.c
  spu_BWT_T_Speed.c
  spu_MTF.c
  spu_Huffman_frequencies.c
  spu_Huffman_encoding.c
  utils.h
  Makefile
Tools/
  cellsdkiso.zip
  CellSDK-Devel-Fedora_3.0.0.1.0.iso
  CellSDK-Extras-Fedora_3.0.0.1.0.iso
Tex/
  projekt.zip
Doc/
  Akcelerace_algoritmu_komprese_dat_na_platforme_sony_ps3.pdf