FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Komprese LZ77 pomocí OpenCL
Student:	Martin Bobek
Vedoucí:	Ing. Radomír Polách
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce zimního semestru 2019/20

Pokyny pro vypracování

Research LZ77 compression method and its parallel implementation. Research OpenCL for parallel processing using GPU.
Analyse and implement LZ77 method compression method using OpenCL.
Test LZ77 method compression method for speed and
compression ratio on both GPU and CPU and compare the results.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 6. března 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Komprese LZ77 pomocí OpenCL

Martin Bobek

Katedra teoretické informatiky
Vedoucí práce: Ing. Radomír Polách

14. května 2018

Poděkování

V první řadě bych rád bych poděkoval vedoucímu mé práce Ing. Radomírovi Poláchovi za užitečné rady a možnost otestovat běh implementace na jeho high-end výpočetních zařízeních. Dále bych rád poděkoval své rodině za jejich nepřetržitou podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 14. května 2018 ……

Odkaz na tuto práci

Bobek, Martin. *Komprese LZ77 pomocí OpenCL*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato bakalářská práce se zabývá implementací paralelní verze kompresního algoritmu LZ77 pro výpočet na grafických kartách s využitím frameworku OpenCL. V rámci řešení byly pro vyhledávání implementovány a porovnány Z-algoritmus, hešování a prohledávání hrubou silou a byla otestována výkonnost implementace při probíhajícím výpočtu na GPU a CPU.

Klíčová slova Kompresní algoritmus LZ77, framework OpenCL, paralelizace, GPGPU, grafická karta

Abstract

This bachelor thesis is focused on the implementation of a parallel version of the LZ77 compression algorithm to be computed on a graphics card using the OpenCL framework. Z-algorithm, hashing and brute force search algorithm are implemented and tested for the searching part of LZ77 within this thesis, and the achieved results on GPU and CPU are compared.

Keywords Compression algorithm LZ77, OpenCL framework, parallelization, GPGPU, graphics card

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Seznam algoritmů

Úvod

V dnešní stále více se digitalizující společnosti neustále dochází k přenosu a archivaci velkých objemů dat. Bezztrátová komprese je schopna tento objem zredukovat, aniž by došlo k narušení integrity dat, a může tak mimo jiné šetřit kapacitu úložných zařízení, či zefektivnit využití komunikačních kanálů. Cenou je provádění dodatečných výpočtů pro získání zkomprimované verze dat. To může být příliš výpočetně náročné a tedy může trvat příliš dlouho. Tato práce se zabývá přesunem výpočtu komprimace z běžně používaného CPU na obecně nepříliš zužitkovaný GPU a potenciálně i zrychlením výpočtu vhodným využitím vysoce paralelní architektury grafické karty.

Cílem rešeršní práce je seznámení s kompresním algoritmem LZ77, se způsoby, jakým jej lze implementovat a paralelizovat, dále stručně popsat architekturu grafické karty z pohledu frameworku OpenCL a jak pro tuto architekturu vyvíjet efektivní aplikace.

Cílem praktické části práce je ve frameworku OpenCL naimplementovat paralelní verzi algoritmu LZ77 pro grafickou kartu a otestovat rychlost a kompresní poměr implementované aplikace při výpočtu probíhajícím na GPU oproti CPU.

První kapitola se zabývá frameworkem OpenCL, jakým způsobem jej lze využít pro „general-purpose processing on graphics processing units“ (GPGPU), a zmiňuje osvědčené postupy při vývoji aplikací v tomto frameworku.

Druhá kapitola rozebírá komprimaci, princip, na kterém je založen algoritmus LZ77, možné způsoby jeho efektivní implementace na CPU a na základě těchto znalostí a znalostí z předchozí kapitoly odhaduje efektivitu odvozených implementací pro GPU.

Ve třetí kapitole je popsáno, jakým způsobem byly implementovány vybrané metody z druhé kapitoly.

Poslední čtvrtá kapitola obsahuje testování a zhodnocení dosažených výsledků implementace.

Kapitola 1

Framework OpenCL

Tato kapitola rozebírá framework OpenCL, který je využit v praktické části této práce pro přesunutí prováděných výpočtů na grafickou kartu. Zaměří se na informace užitečné pro vývoj aplikací v tomto frameworku. Hlavními body bude struktura dostupné paměti, výpočetní model, jazyk OpenCL C založený na dobře známém jazyce C a jeho omezení, a budou zmíněny vlastnosti dobře navržené a efektivní aplikace v tomto frameworku.

1.1 O OpenCL

Informace uvedené v následujících dvou odstavcích jsou vzaty z [1]. OpenCL (Open Computing Language) je framework k psaní programů na tzv. heterogenní platformy, tedy počítače obsahující kombinaci centrálních procesorových jednotek (CPU), grafických procesorů (GPU), a dalších procesorů. OpenCL se stal prvním průmyslovým standardem cílícím právě na tyto platformy na konci roku 2008, kdy byl vydán konsorciem Khronos. V této práci je rozebírána specifikace 1.2 tohoto frameworku, protože autorovi dostupná grafická karta pro vývoj podporuje nejvýše tuto verzi.

Výhodou programů v OpenCL je jejich portabilita, jediný program může beze změn nebo jen s malými úpravami běžet na mnoha platformách od běžných stolních počítačů, mobilů, programovatelných hradlových polí až po nody super počítačů. Placenou cenou za tuto portabilitu je nutnost v rámci programu přes API OpenCL vybírat platformu a výpočetní zařízení na dané platformě, přičemž takováto „nízkoúrovňová“ práce není při programování běžná.

V této práci je tento framework použit pro účely provádění obecných výpočtů na grafických kartách (GPGPU – General–purpose computing on graphics processing units), v jehož rámci je využita masivně paralelní architektura grafické karty ke komprimaci dat algoritmem LZ77.

1.2 Základní pojmy

Tato sekce obsahuje vysvětlení významu základních pojmů, které budou v následujícím textu používány a jejichž přehledný výčet je vhodný pro připomenutí významu některého z mnoha pojmů představených v této kapitole. Zdrojem pojmů je [2].

Globální ID Číslo, které je využíváno jako globální identifikátor jedné pracovní položky. Při spuštění kernelu je zadán celkový počet pracovních položek, ze kterého je globální ID odvozeno (pro každé číslo z rozsahu od 0 do celkového počtu bez jedné je vytvořena pracovní položka s daným ID).

Host/Hostitel Obvykle to bývá CPU na výpočetním zařízení, kde OpenCL aplikace běží. Host obstarává práci s OpenCL API, vybírá platformu, výpočetní zařízení, spouští kernely a další.

Kernel Kernelem je nazývána vstupní funkce programu v jazyce OpenCL C vykonávaného na OpenCL zařízení (obdobně jako funkce main v klasickém progrmau v C). V kódu je označena speciálním kvalifikátorem `__kernel`.

Lokální ID Lokální ID identifikuje pracovní položku v rámci pracovní skupiny, do které patří.

OpenCL kontext Kontext v OpenCL popisuje prostředí, ve kterém jsou spouštěny kernely. V rámci kontextu na OpenCL zařízení jsou vytvářeny OpenCL objekty reprezentující programy (obsahující zdrojový kód kernelu, jsou kompilovány za běhu programu), kernely a paměťové buffery na daném zařízení.

OpenCL platforma Prvním krokem v OpenCL aplikaci je vybrání platformy, na které bude probíhat výpočet. V rámci platformy může být více OpenCL zařízení (platformou může být např. procesor zahrnující dvě OpenCL zařízení, samotné CPU a integrované GPU).

OpenCL zařízení Složka OpenCL platformy, skládá se z množiny výpočetních jednotek. Na daném zařízení se vytvoří tzv. příkazová fronta (command-queue), do které lze vkládat příkazy jako spouštění kernelů, čtení ze nebo zápis do paměťových objektů.

Pracovní skupina (work-group / WG) Množina příbuzných pracovních položek, které jsou vypočítávány na jedné výpočetní jednotkce. V rámci pracovní skupiny mají všechny pracovní položky přístup k lokální paměti příslušné pracovní skupiny.

Pracovní položka (work-item / WI) Jedna z množiny paralelně běžících spuštění kernelu vyvovalných hostem na OpenCL zařízení. Má unikátní globální ID mezi všemi pracovními položkami pro daný kernel a unikátní lokální ID mezi všemi pracovními položkami z příslušné pracovní skupiny.

Příkazová fronta Objekt vytvářený v rámci OpenCL kontextu. Jsou do ní vkládány příkazy pro zařízení jako např. zápis a čtení z bufferů OpenCL zařízení a spouštění kernelů. Má více módů, ve kterých může běžet, více informací je uvedeno dále v sekci 1.5.2.

SIMT (single instruction, multiple threads) Model, podle kterého jsou vykonávány pracovní položky v jednom warpu. Výpočetní elementy provadějící skupinu pracovních položek odpovídající warpu v jedné chvíli vždy vykonávají stejnou instrukci. Kazdá pracovní položka však má vlastní instrukční ukazatel a v případě, že při vykonávání pracovních položek z jednoho warpu dojde k rozvětvení, je tato situace řešena serializací běhu a zneaktivněním výpočetních elementů provádějících pracovní položky, které se nenachází v aktuálně prováděné větvi programu. To může mít negativní dopad na výkon aplikace, viz sekce 1.6.

Výpočetní jednotka (compute unit / CU) Část OpenCL zařízení. Jediné zařízení má jednu nebo více výpočetních jednotek. Jedna výpočetní jednotka se skládá z jednoho nebo více výpočetních elementů a k nim přidružené lokální paměti.

Výpočetní element (processing element / PE) Virtuální skalární procesor. Jedna pracovní položka může být vykonávana jedním nebo vícero výpočtními elementy.

Warp Warp je pojem z technologie CUDA a nemá v rámci OpenCL přímý ekvivalent, avšak v této práci je pro zjednodušení popisu využíván. Warp je skupina pracovních položek, která je vykonávána na GPU na základě modelu SIMT (single instruction, multiple thread). Zdrojem tohoto pojmu je [3].

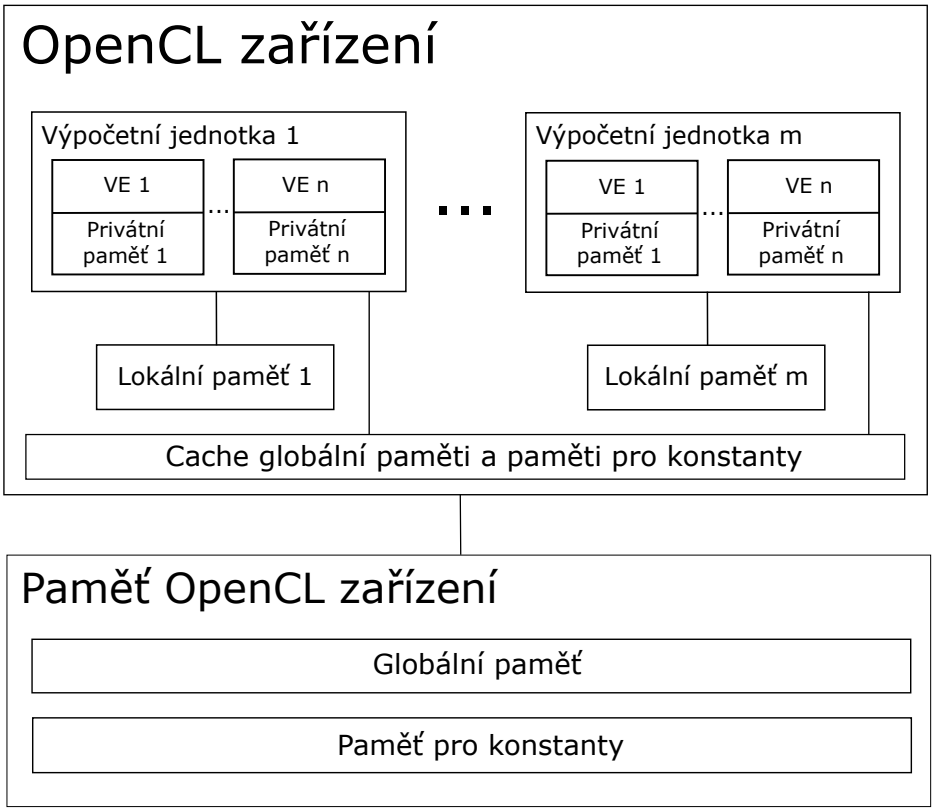
1.3 Paměťová struktura

Tato sekce se věnuje rúzným typům paměti, které lze využít na zařízeních podporujících OpenCL a jejich výhodám a nevýhodám [3].

Globální paměť Globální paměť je zdaleka největší dostupnou pamětí pro OpenCL zařízení. Velikost tohoto typu paměti je obvykle advertizována prodejci grafických karet a pohybuje se v řádu gigabajtů. Nevýhodou tohoto typu paměti je, že se nenachází na čipu a jedinéčtení z tohoto typu

paměti obvykle trvá 400–600 taktů. Tuto prodlevu lze však v některých případech snížit technikami, které budou popsány dále v sekci 1.6.

Paměť pro konstantní data Paměť pro konstantní data je rovněž mimo čip, a na rozdíl od zbývajících pamětí, do kterých lze za běhu zapisovat i číst, slouží pouze ke čtení. Narozdíl od globální paměti je pro tento typ paměti garantováno, že při čtení z ní se data uloží do cache, což může být užitečné k dosažení lepšího výkonu aplikace.



Obrázek 1.1: Diagram dostupných typů paměti na základě [2]

Privátní paměť Tento typ paměti se automaticky využívá na parametry funkcí (kromě vstupní kernelové funkce) a na lokální proměnné ve funkcích. Je viditelná pouze pro konkrétní pracovní položku a dokud je pro to místo, alokují se proměnné v privátní paměti do registrů na čipu. Přístup do těchto registrů je velmi rychlý, trvá typicky 1 takt. Při vyčerpání dostupných registrů se další proměnné alokují do paměti, která je mimo čip a prodleva stoupá na úroveň globální paměti, dochází řádově až k stonásobnému zpomalení.

Lokální paměť Lokální paměť je sdílená mezi pracovními položkami v jedné pracovní skupině. Stejně jako privátní paměť se nachází na čipu a je oproti globální paměti velmi rychlá, operace trvají řádově jednotky až desítky taktů v závislosti na tom, zda bylo využito sloučeného čtení z paměti.

1.4 Výpočetní model

informace v této a následující sekci byly čerpány z [1]. Z pohledu OpenCL se OpenCL zařízení (jako např. CPU nebo GPU) skládá z výpočetních jednotek, které se dále dělí na výpočetní elementy. Množství dostupných výpočetních jednotek a výpočetních elementů na OpenCL zařízení je dána jeho architekturou. V následující podsekci je popsáno, jak s nimi OpenCL nakládá.

1.4.1 Vykonávání kernelu

Když je na hostu spuštěn běh kernelu na OpenCL zařízení, vytvoří se množina indexů přirozených čísel z intervalu od 0 do velikosti odpovídající hodnotě parametru v příkazu bez jedné. Pro každý index z této množiny je spuštěna instance kernelu. Tyto instance se nazývají pracovní položky, a daným indexem (také nazývaným globální ID) se identifikují.

Všechny vytvořené pracovní položky vykonávají stejné, kernelem zadefinované instrukce. Navzdory tomu se však chování jednotlivých pracovních položek může lišit např. při větvení programu v závislosti na globálním ID, či vstupních datech.

Pracovní položky jsou seskupeny do pracovních skupin. Pracovní skupiny jsou hrubší dekompozicí množiny indexů, jsou všechny stejně velké a jejich velikost dělí velikost množiny všech indexů. Každá z pracovních skupina má svůj unikátní index mezi všemi pracovními skupinami, a zároveň každá pracovní položka má své unikátní index v rámci pracovní skupiny (také nazývaný lokální ID). Dvojice čísel skládající se z ID pracovní skupiny a lokálního ID dané pracovní položky je pro každou pracovní položku unikátní a lze ji používat jako alternativu ke globálnímu ID.

Pro pracovní položky v rámci jedné pracovní skupiny platí, že jsou všechny vykonávány zároveň výpočetními elementy jedné výpočetní jednotky. Paralelní běh těchto pracovních položek je neměnný fakt, narozdíl od vykonávání kernelů, či rozdílných pracovních skupin, kterézto mohou běžet zároveň, ale v závislosti na implementaci mohou být zeserializovány a nelze se na to z pozice programátora spolehnout. Serializace práce pracovních skupin může nastat např. v případě, kdy je v kernelu využívano příliš registrů nebo lokální paměti, a při paralelním běhu všech pracovních skupin by těchto prostředků nebyl dostatek.

1.4.2 Příkazové fronty

Způsob, jakým host komunikuje s OpenCL zařízeními je skrz příkazovou frontu. Příkazová fronta je vytvářena v programu na hostu v rámci OpenCL kontextu pro konkrétní zařízení a vkládají se tam příkazy určené pro toto zařízení. Zadávání příkazů je neblokující, tzn. program v hostu nečeká na dokončení zadané práce probíhající na OpenCL zařízení a pokračuje vykonáváním následujícího kódu. Příkazy mohou být těchto typů:

Spouštění kernelu Tyto příkazy spouští běh kernelu na OpenCL zařízení.

Práce s pamětí Příkazy tohoto typu slouží k přesunu dat mezi hostem a paměťovými buffery na OpenCL zařízení, přesunu dat mezi buffery a k dalším podobným účelům.

Synchronizace Synchronizační příkazy se využívají k zajištění, že před vykonáním následujících příkazů ve frontě je nejdříve dokončen běh příkazů předchozích. To se může zdát jako inherentní vlastnost této datové struktury, ale nemusí tomu tak být, viz dále.

Role příkazů pro spuštění kernelů a práci s pamětí je zřejmá. Synchronizační příkazy jsou užitečné v případech, kdy je v příkazové frontě zařazeno ke spuštění více kernelů, které spolu interagují. Typickým příkladem je situace, kdy výstup jedné skupiny kernelů je použit jako vstup druhé skupiny kernelů. V takovém případě potřebujeme zajistit, aby druhá skupina začala svoji práci až po skončení první, a toho dosáhneme právě použitím tohoto typu příkazů.

Pro vykonávání příkazů v příkazové frontě existují dva základní módy:

Běh v pořadí (in-order execution) Příkazy v příkazové frontě v tomto módu jsou vykonávány v pořadí, v jakém byly zadány, a v tomto pořadí jsou i dokončovány. Toho je dosaženo přísnou serializací, kdy práce předchozího příkaz ve frontě musí být dokončena před začátkem práce následujícího příkazu.

Neregulovaný běh (out-of-order execution) V tomto módu jsou příkazy stále spouštěny v zadaném pořadí, ale není regulováno, v jakém pořadí skončí. Pokud jsou mezi příkazy ve frontě závislosti, programátorovou odpovědností je, aby tyto závislosti s pomocí synchronizačních příkazů ošetřil.

Výhodou neregulovaného běhu je, že v některých případech může lépe balancovat zátěž na zařízení (např. výpočetní jednotka, která z nějakého důvodu skončí se svojí prací dříve, nemusí neaktivně čekat na ostatní vyýpočetní jednotky a může začít pracovat na dalším kernelu) a zlepšit tak výkon. Pro mnoho problémů však z jejich podstaty není možné neregulovaný běh příkazové fronty

využít. Dále, zatímco všechna OpenCL zařízení garantují podporu módu příkazové fronty běhu v pořadí, neregulovaný běh na některých zařízeních nemusí být podporován.

Alternativně, u zařízení, které tuto možnost podporuje, je možno v rámci kontextu nad daným zařízením vytvořit více příkazových front, které pracují nezávisle a souběžně bez možnosti je synchronizovat.

1.5 Aplikace v OpenCL

OpenCL aplikace se skládají ze dvou částí – části běžící na hostu a množiny kernelů. V první části se pracuje s OpenCL API za účelem výběru a přípravy OpenCL zařízení k práci. Kernely jsou funkce psané v jazyce OpenCL C, které provádí požadované výpočty na OpenCL zařízeních. Typicky provádějí výpočty nad daty ze vstupních bufferů a svůj výstup uloží do výstupnich bufferů.

1.5.1 Část vykonávaná na hostu

Jak už bylo řečeno, tato část obstarává práci s API frameworku OpenCL. To se obvykle sestává z následujících kroků:

1. Dotázání se API na dostupné OpenCL platformy a OpenCL zařízení na těchto platformách.

2. Výběr vhodného zařízení pro výpočty dané aplikace, často na základě buď výběru uživatele, nebo krátkého otestování dostupných zařízení.

3. Vytvoření tzv. OpenCL kontextu na vybraném zařízení, v rámci kontextu je dále vytvořena příkazová fronta pro dané zařízení.

4. Vytvoření OpenCL paměťových bufferů na OpenCL zařízení, které budou předány kernelu jako argumenty a v případě bufferů určených ke čtení se tyto buffery také nainicializují vstupními daty.

5. Načtení kódu kernelů, na jehož základě jsou vytvořeny OpenCL objekty reprezentující programy. Na základě těchto objektů se poté programy zkompilují pro vybrané OpenCL zařízení.

6. Vytvoření objektů OpenCL pro kernel na základě v pátém kroku zkompilovaných programů. Argumentům těchto kernelových objektů jsou dále přiřazeny jejich odpovídající OpenCL paměťové buffery, se kterými mají pracovat.

7. Spuštění běhu kernelů se specifikovaným počtem vytvořených pracovních položek.

8. Načtení výstupů prováděných výpočtů z OpenCL bufferů a další naložení s těmito daty dle aplikace.

1.5.2 Kernelová část

Kernely aplikací v OpenCL jsou psány v jazyce OpenCL C. Tento jazyk vychází ze standardu ISO C99, a je tedy velmi podobný jazyku C. Tato práce uvede jen některé změny mezi těmito dvěma jazyky. Významným omezením oproti jazyku C je, že není podporována rekurze a není možno použít část standardních knihoven C. Naopak mezi významné přidané funkcionality patří funkce k identifikování aktuální pracovní položky, prostředky pro synchronizaci výpočtu pracovních položek v rámci pracovní skupiny, kvalifikátory paměťového prostoru (pro globální, lokální, privátní pamět a pro paměť pro konstantní data), vektorové datové typy fixní velikosti, a mnoho dalšího.

Kernelová část se zcela odvíjí od konkrétních výpočtů, které chceme na OpenCL zařízeních provádět. Prvním krokem v kódu kernelu je obvykle získání unikátní identifikace aktuální pracovní položky (její globální ID, nebo kombinace ID pracovní skupiny a ID této položky v rámci této skupiny), na základě které se přistupuje do paměťových bufferů a provádí se požadovaná práce nad daty příslušející dané pracovní položce. V této části je důležitý dobrý návrh provádění výpočtů, který je co možná nejlépe přizpůsoben pro běh na OpenCL zařízeních. Následující sekce 1.6 popisuje, jak toho lze dosáhnout.

1.6 Osvědčené postupy při vývoji

Tato sekce je věnována specifickým doporučením, jak navrhovat a implementovat OpenCL aplikace, aby dosahovaly optimálního výkonu. Byla napsána na základě [4].

1.6.1 Divergence warpů

Větvení v rámci kernelů může mít zásadní negativní dopad na výkon v případě, kdy při průchodu programem divergují pracovní položky z jednoho warpu (typicky při použití příkazů `if`, `switch`, `do`, `for`, `while` s podmínkami, jejichž vyhodnocení není v rámci warpu jednotné). V takovém případě musí být běh serializován, a je proto vhodné se při vytváření kernelů aktivně snažit možnou divergenci minimalizovat.

1.6.2 Slučování čtení z paměti (Memory coalescing)

Důležitou optimalizací je využívání automatizovaného slučování čtení z paměti, především v případě čtení z globální paměti, které způsobuje největší prodlevu. V situaci, kdy více pracovních položek z jednoho warpu přistupuje do paměti do segmentu 32 bajtů v případě čtení 8bitových slov, do segmentu

64 bajtů v případě čtení 16bitových slov, nebo segmentu 128 bajtů v případě čtení 32bitových nebo 64bitových slov, pak je místo jednotlivých čtení pro každou pracovní položku provedena jediná paměťová transakce a výrazně se tím zvyšuje efektivita využití šířky pásma pro přenos dat mezi hostem a OpenCL zařízením.

Segmenty, které mohou být přečteny jedinou paměťovou transakcí, začínají na násobku jejich délky (např. není možné načíst potřebna data jedinou 64B transakcí v případě, kdy by byla čtena data z adres 32 až 96, protože 32 není násobkem 64).

Od specifikace OpenCL 1.2 byla zrušena podmínka, že k. výpočetní element ve warpu musí přistupovat ke k. prvku v daném segmentu, a je tedy možné, aby libovolný procesní element přistupoval k libovolnému prvku z daného segmentu, včetně vícenásobných čtení stejného prvku vícero elementy.

1.6.3 Využívání lokální paměti

Lokální paměť, která je sdílená mezi pracovní elementy v pracovní skupině, je značně rychlejší než globální paměť a je vhodné ji využívat jako programátorem ovládanou cache. Typické využití nachází v případě, kdy provádíme opakované výpočty nad stejnými daty; nejdříve jsou načtena data z globální paměti do lokální, v rámci pracovní skupiny jsou provedeny potřebné výpočty a výsledná data jsou nahrána zpátky do globální paměti. Dobrým příkladem takového výpočtu je násobení matic, při kterém jsou jednotlivé prvky matice mnohokrát použity pro výpočet a opětovná čtení z necachované globální paměti při každé operaci by měla znatelný negativní dopad na rychlost programu.

Tato paměť se dále dělí na více tzv. bank, přičemž ke všem bankám lze přistupovat zároveň. Banky jsou organizovány tak, aby po sobě jdoucí 32bitová slova byla uložena v po sobě jdoucích bankách. V případě, že všechny pracovní elementy z jednoho warpu přistupují do lokální paměti bez bankovních konfliktů (každý do jiné banky), je tato paměť stejně rychlá jako lokální registry, a její účelné použití tedy může značně pomocí výkonu aplikace.

Kapitola 2

O kompresi dat a algoritmu LZ77

Tato kapitola se zaobírá kompresí dat, popisuje algoritmus LZ77 a nalezené způsoby, jakými jej lze zparalelizovat při výpočtu na GPU.

2.1 Úvod do komprese dat

Komprese dat je proces, jehož účelem je zakódování informace do takové podoby, která využívá méně bitů paměťového místa než původně. Datová komprese se dělí na dva základní druhy [5]:

bezztrátová Tento typ komprese umožňuje přesnou reprodukci původních, nezkomprimovaných dat ze zkomprimovaných, a nedochází tedy ke ztrátě informace.

ztrátová Oproti bezztrátové při procesu komprimace dochází ke ztrátě nedůležitých dat tak, aby si člověk nevšiml rozdílu. To umožňuje dosáhnout znatelně větší komprese a tedy menších výsledných souborů, ale v mnoha případech ji není možno použít. Často se využívá např. při kompresi zvukových stop, či videí.

Algoritmus LZ77, jímž se tato práce zabývá, spadá mezi bezztrátové kompresní algoritmy.

2.2 Lempel-Ziv 77

Informace v úvodu této sekce pochází z [5]. LZ77 je algoritmus vyvinutý Lempelem a Zivem v roce 1977 a je základním stavebním kamenem pro další z něj odvozené algoritmy jako LZ78, LZSS a LZW. Komprimace dosahuje tak, že postupně zpracovává vstupní soubor, resp. řetězec bajtů, zleva doprava,

a nahrazuje sekvence dat, které se v limitované vzdálenosti (dáno konkrétní implementací) zopakují, odkazem na jejich předchozí výskyt. Tento odkaz se skládá ze trojice dvou čísel a jednoho znaku, kde první číslo udává vzdálenost od předchozího výskytu v bajtech, druhé číslo značí délku opakující se sekvence v bajtech a poslední člen trojice je znak, který ve vstupním řetězci následuje za opakující se sekvencí. Třetí člen trojice je užitečný k zakódování znaku, který se v řetězci dosud nevyskytl, nebo je příliš vzdálený k zakódování.

Hledání zopakovaných sekvenci bajtů lze dosáhnout mnoha způsoby, kterými se v tato kapitola ještě bude zabývat. Společným prvkem je, že v průběhu své práce algoritmus využívá dopředný buffer a relativně delší vyhledávací buffer. Dopředný buffer bývá obvykle velikosti v řádech desítek až stovek bajtů a obsahuje data, která budou v nejbližší době zkomprimována. Vyhledávací buffer mívá nejčastěji velikost v jednotkách až desítkách kB a obsahuje naposledy zpracovaná data v nezkomprimované formě. Při navrhování velikostí bufferů je nutno vzít v potaz, že zvětšením bufferů můžeme naleznout delší shody a můžeme zakódovat větší délku shody, ale zároveň tím zvětšujeme délku odkazů. Dané rozsahy velikostí, které jsou nejčastěji používány, vychází z toho, že při vhodné kombinaci velikostí bufferů lze odkaz zakódovat do přesně dvou či tří bajtů a zjednodušit si tak implementaci algoritmu.

Při daném obsahu dopředného bufferu a vyhledávacího bufferu se pracuje s dobře známým problémem vyhledávání v textu – je hledán výskyt co nejdelší předpony daného řetězce v jiném řetězci. Konkrétně je cílem naleznout co nejdelší předponu obsahu dopředného bufferu ve vyhledávacím bufferu, aby se algoritmus odkázal do vyhledávacího bufferu dříve popsanou trojicí a dosáhl tak komprese.

2.2.1 Formální popis

Tato sekce je založena na formálním popisu z [6]. Nejprve si zadefinujeme základní notaci:

- $X = x[1..N] = x_1...x_N$: řetězec určený ke zkomprimování, kde x_i je znak z abecedy Σ,
- $S_p = x[p..N]$: p-átá přípona řetězce X,
- $D_p = x[p{-}\mathrm{DSIZ}..p-1]$: vyhledávácí buffer velikosti DSIZ, když část $x[1..p-1]$ už byla zkomprimována.

V situaci, kdy už máme část vstupního řetězce $x[1..p-1]$ zkomprimovánu, postupujeme tak, že nalezneme nejdelší řetězec $x[j..j+l-1]$ začínající ve vyhledávacím bufferu D_p ($p{-}\mathrm{DSIZ} \leq j \leq p-1$) a který je identický s předponou $x[p..p+l-1]$ přípony S_p, a ten zakódujeme pomocí trojice $(\mathrm{DSIZ}{-}j, l, x[p+l])$, kterou připojíme za dosud zakódované trojice. V případě, že žádný takový řetězec nenalezneme, označíme tak nulovou hodnotou offsetu, nebo délky (v závis-

losti na implementaci) a do třetího členu trojice vložíme první bajt dopředného bufferu.

Formálně, LZ faktorizací řetězce S nazveme sekvenci trojic (i, j, k), kde i značí vzdálenost začátku dopředného bufferu od nalezeného shodného podřetězce (offset), j značí délku shodného podřetězce a k je následující znak za nalezenou shodou.

Následné dekomprimace ze sekvence zakódovaných trojic dosáhneme jednoduše iterací přes trojice zleva doprava, kdy nejdříve zkopírujeme sekvenci dat nacházející se v již dekomprimovancých datech o i bajtů zpět o délce j (popřípadě tento krok přeskočíme, pokud je offsetem nebo délkou signalizováno, že shoda nebyla nalezena), za které vložíme znak k a pokračujeme další trojicí.

2.2.2 Příklad

Pro lepší ilustraci popisu z předchozí sekce si uveďme příklad LZ faktorizace řetězce „ABRAKADABRA“ a pro zjednodušení použijme vyhledávací buffer velikosti pouze 7 B a dopředný buffer velikosti pouze 4 B [7].

Tabulka 2.1: Ukázka komprimace

Krok	**Vyhled. b.**	**Dopředný b.**	**Zb. vstupu**	**Výstup**	**Posun**
1	- - - - - - -	ABRA	KADABRA	(0, 0, A)	1 B
2	- - - - - - A	BRAK	ADABRA	(0, 0, B)	1 B
3	- - - - - AB	RAKA	DABRA	(0, 0, R)	1 B
4	- - - - ABR	AKAD	ABRA	(3, 1, K)	2 B
5	- - ABRAK	ADAB	RA	(2, 1, D)	2 B
6	ABRAKAD	ABRA		(7, 3, A)	4 B

Z tabulky 2.1 znázorňující práci algoritmu při komprimaci lze snadno pochopit princip, na kterém je založen. V počáteční fázi je vyhledávací buffer prázdný a postupně se zaplňuje již zakódovanými daty. Ve čtvrtém a pátém kroku se jíž algoritmus odkazuje na jeden bajt z vyhledávacího bufferu a v šestém kroku se odkázuje dokonce na 3 bajty – „ABR“, zakončené znakem „A“.

Tabulka 2.2: Ukázka dekomprimace

Krok	**Vstup**	**Výstup před**	**Přidáno**	**Výstup po**
1	(0, 0, A)		A	A
2	(0, 0, B)	A	B	AB
3	(0, 0, R)	AB	R	ABR
4	(3, 1, K)	ABR	AK	ABRAK
5	(2, 1, D)	ABRAK	AD	ABRAKAD
6	(7, 3, A)	ABRAKAD	ABRA	ABRAKADABRA

Dekomprimace (viz tabulka 2.2) je výpočetně jednodušší než komprimace a řídí se popisem v 2.2.1.

2.3 Způsoby implementace LZ77

Tato sekce rozebírá vybrané způsoby implementace vyhledávání podřetězce ve vyhledávacím bufferu identického s předponou dopředného bufferu.

2.3.1 Naivní vyhledávání hrubou silou

Nejjednodušší metodou je vyhledávání hrubou silou. Tedy, z každého indexu vyhledávacího bufferu spustit proceduru, která porovná řetězec začínající v daném bodě s řetězcem v dopředném bufferu a vrátí délku nalezené shody. Z těchto hodnot si stačí uložit tu nejdelší s indexem, na kterém začínala, načež tyto hodnoty využít k zakódování dat z dopředného bufferu. Výpočetní složitost tohoto algoritmu je O(nm), kde m je délka vyhledávaného řetězce a n je délka prohledávaného řetězce. Vychází z nejhoršího případu, kdy se vyhledávaný i prohledávaný řetězec skládají ze stejného, opakovaného znaku. Paměťová složitost je konstantní.

2.3.2 Z-algoritmus

Naivní přístup nijak nevyužívá faktu, že každý následující porovnávaný řetězec je příponou předchozího. Z porovnávání řetězce na dané pozici tedy můžeme získat nějakou užitečnou informaci pro vyhledávání na následujících indexech. Z této myšlenky vychází mnoho podobných algoritmů jako Knuth–Morris–Pratt (KMP), Boyer–Moore (BM), Z-algoritmus, a další, přičemž každý ji využívá ke zvýšení efektivity trochu jiným způsobem. V této práci je popsán Z-algoritmus na základě [8].

Algoritmus pro řetězec S délky n sestrojí pole Z takové, že $Z[i]$ je délka nejdelšího podřetězce začínajícího v $S[i]$, který je zároveň předponou S. Formálně vzato je tedy v $Z[i]$ maximální přirozené číslo k takové, aby $S[j] = S[i + j]$ pro všechna přirozená j z intervalu $[0, k)$. Pro naše účely nebude nutno vytvářet celé pole Z, stačí si udržovat nejdelší nalezenou shodu, která bude využita k zakódování.

Základním kamenem tohoto algoritmu je, že si za průchodu řetězcem (i iteruje od 1 do n) zleva doprava udržuje interval $[L, R]$, což je interval s největším možným R takovým, že $1 \leq L \leq i \leq R$ a $S[L..R]$ je příponou S. Pokud žádný takový interval pro dané i neexistuje, položíme $L = R = -1$.

Funkci algoritmu lze popsat indukcí. Pro $i = 1$ spočteme L a R porovnáním $S[0..]$ a $S[1..]$ a získáme tak i $Z[1]$. Nyní předpokládejme, že máme korektní interval $[L, R]$ pro $i - 1$ a všechny hodnoty pole Z v indexech od 0 do délky dopředného bufferu. Pak můžeme následujícím způsobem spočítat $Z[i]$ a $[L, R]$ pro aktuální i:

- Pokud $i > R$, pak neexistuje podřetězec, který je předponou S, začíná před i a končí za i. Pokud by takový podřetězec existoval, interval $[L, R]$ by tento podřetězec ohraničoval. V této situaci spočítáme nový $[L, R]$ porovnáním $S[0..]$ a $S[i..]$ a nastavíme $Z[i]$ na hodnotu $R - L + 1$.

- Jinak, když $i \leq R$, dosahuje $[L, R]$ až k alespoň i. Položme $k = i - L$. Víme, že $Z[i] \geq min(Z[k], R - i + 1)$ z definice pole Z a protože řetězec $S[i..]$ je identický s $S[k..]$ po alespoň $R - i + 1$ znaků.

- Pokud $Z[k] < R - i + 1$, pak nemůže znakem $S[i]$ začínat podřetězec, který je zároveň předponou a zároveň končí za R, protože jinak by $Z[k]$ muselo být větší. $Z[i]$ tedy nastavíme na hodnotu $Z[k]$ a interval $[L, R]$ neměníme.

- Pokud $Z[k] \geq R - i + 1$, pak je možné, aby aby znakem $S[i]$ začínal podřetězec, který je předponou a zároveň končí za R, a musíme tuto možnost ověřit. Nastavíme tedy $L = i$ a začneme porovnávat řetězce $S[R + 1..]$ s $S[R - L]$, abychom získali nové R a $Z[i]$.

V rámci jednoho průchodu popsaným cyklem postupně spočítáme všechny hodnoty pole Z, ze kterých použijeme maximální hodnotu společně s indexem, na kterém se nacházela.

Tento přístup vylepšuje výpočetní složitost naivního přístupu v nejhorším případě na $O(m+n)$, důkaz viz [8]. Pro funkci algoritmu je nutno si v paměti udržovat pole Z pro vyhledávaný řetězec, paměťová složitost je tedy $O(m)$.

2.3.3 Hešování

Odlišným přístupem od předchozích dvou je hešování předpon všech podřetězců, kdy kolize jsou řešeny zřetězením. Tato metoda však obvykle vyžaduje již $O(v)$ paměti na hešovací tabulku. Za průchodu vstupním algoritmem si udržujeme hešovací tabulku, kde máme uloženy odkazy na všechny přípony S_i, které jsou aktuálně ve vyhledávacím bufferu. Index je spočten vybranou hešovací funkcí z prvních několika znaků $x_i, x_{i+1}, ..$ a hodnotou na daném indexu hešovací tabulky je index i dané přípony S_i. Přípony se stejnou hodnotou heše jsou uloženy do spojového seznamu, kdy nejnovější shoda je ukládána na začátek. Tento způsob vyhledávání shod je využit i v známém programu gzip. Následující popis se založen na jeho implementaci, jak je popsána v [6].

Abecedou Σ jsou všechny možné hodnoty bajtu – $\{0, 1, .., 255\}$ a jsou zadefinovány hranice M_1 a M_2 pro délku l nalezené shody následovně:

- M_1: Pokud je $l < M_1$, pak je shoda vyhodnocena jako nedostatečně dlouhá a znak x_p nacházející se na začátku vyhledávacího bufferu je zakódován jako samostatný znak. Zároveň M_1 opdpovídá počtu znaků, který využívá hešovací funkce h ke spočtení heše.

- M_2: Pokud je $l > M_2$, pak je shoda omezena na délku M_2 za účelem zjednodušení implementace. M_2 je tedy velikost dopředného bufferu.

V gzipu jsou spojové seznamy jednotlivých heší implementovány pomocí dvou polí celých čísel *head* a *prev*, která mají velikost HSIZ, resp. DSIZ. HSIZ a DSIZ musí být mocninou dvou. Pole *head* a *prev* jsou definovány takto:

- $head[h_1]$ = maximální index i takový, aby $h(x_i, x_{i+1}, .., x_{i+M_1-1}) = h_1$.
- $prev[i\&DMASK]$ = maximální index j menší než i takový, aby $h(x_i, x_{i+1}, .., x_{i+M_1-1}) = h(x_j, x_{j+1}, .., x_{j+M_1-1})$, přičemž číslo DMASK $= DSIZ - 1$ je bitová maska, která efektivně aplikuje na index i operaci zbytku po dělení číslem $DSIZ$ (platí díky tomu, že DSIZ je mocninou dvou) a omezuje tak hodnotu indexu do množiny $\{0, 1, ..,$DSIZ$-1\}$.

Nyní si ukážeme hešovací funkci, kterou využívá gzip, ve výpisu kódu 1. Stejně jako u DMASK, i HMASK = HSIZ−1.

```
int hash(const char text[])
{
    int h = 0;
    for (int i = 0; i < M_1; ++i)
    {
        h = (h << d) | text[i];
        h = h & HMASK;
    }

    return h;
}
```

Výpis kódu 1: Hešovací funkce

S takto zavedenými strukturami už lze jednoduše pracovat, vkládání lze provést pomocí pár řádků, viz výpis kódu 2. V tomto výpisu jsou `data` ukazatel na vstupní řetězec ke zkomprimování a `p` je aktuální index začátku dopředného bufferu. Mazání z hashovací tabulky je nepotřebné, protože kvůli využívání zbytku po dělení (resp. operace AND se stejným účinkem) se hodnoty samy přepisují. Pouze při procházení spojového seznamu shod pro daný heš musíme kontrolovat, zda už index odkazovaného předchozího řetězce se stejným hešem není mimo vyhledávací buffer.

Nyní už zbývá jen popsat, jak tuto techniku použít v rámci LZ77. Pro danou pozici p dopředného bufferu spočteme heš S_p, s ní se odkážeme do hešovací tabulky, projdeme celý odpovídající spojový seznam a porovnáme délku shody všech odkazovaných řetězců s S_p. Nejdelší nalezenou shodu použijeme k zakódování trojice v LZ faktorizaci. Dále spočteme heše a vložíme do tabulky

```
int h = hash(data+p);
prev[p & DMASK] = head[h];
head[h] = p;
```

Výpis kódu 2: Vkládání do tabulky

indexy podřetězců, které se nacházejí mezi aktuální pozicí p a pozicí $p+l+1$, kde l je délka nejdelší nalezené shody, a konečně posuneme dopředný buffer o $l+1$ bajtů dopředu a opakujeme tentýž postup, dokud není LZ faktorizace celého vstupního řetězce hotova.

Výpočetní složitost při využití hešování je v průměrném případě pouze $O(m)$ pro porovnání malého množství nalezených shod. Pro vstupní soubory, na kterých však bude docházet k mnoha kolizím hešovací funkce, může dojít až k zpomalení na $O(nm)$. V rámci programu gzip i v rámci praktické části této práce je velikost hešovacích tabulek totožná s velikostí vyhledávacího bufferu, paměťová složitost se tedy v tomto případě zvyšuje na $O(n)$.

2.3.4 Příponové pole algoritmem skew

Dalším algoritmem, který se dá pro účely LZ77 využít, je skew. Následující popis tohoto algoritmu pochází z [9]. S jeho pomocí můžeme efektivně získat tzv. příponové pole (suffix array), které lze využít dalšími algoritmy k získání LZ faktorizace.

2.3.4.1 Definice příponového pole

Příponovým polem řetězce $X = X[1..N]$ nazveme pole $SA = SA[1..N]$, kde na pozici $SA[k]$ je uložen index i přípony S_i řetězce X takové, která je k. ve vzestupném lexikografickém uspořádání všech přípon řetězce X. Dále definujeme inverzní příponové pole $ISA = ISA[1..N]$, kde na pozici $ISA[k]$ je uložen index i právě tehdy, když $SA[i] = k$. Pro lepší ilustraci je uveden příklad vypočteného příponového a inverzního příponového pole pro řetězec „banana“ v tabulce 2.3.

Tabulka 2.3: Vypočtené příponové a inverzní příponové pole

i	Přípona S_i	Seřazené přípony	SA[i]	ISA[i]
1	banana	a	6	4
2	anana	ana	4	3
3	nana	anana	2	6
4	ana	banana	1	2
5	na	na	5	5
6	a	nana	3	1

2.3.4.2 Algoritmus skew, využití příponového pole

Tento algoritmus v lineárním čase zkonstruuje pro vstupní řetězec příponové pole. Skládá se ze tří kroků:

1. Zkonstruuje příponové pole pro přípony začínající na pozicích s indexem $i \bmod 3 \neq 0$. To provede rekurzivním zavoláním sebe sama na řetězec o dvou třetinách původní velikosti a výsledkem je příponové pole SA_{12}.

2. Zkonstruuje příponové pole pro zbývající přípony na pozicích s indexem $i \bmod 3 = 0$, výsledek uloží jako příponové pole SA_0.

3. Ve třetím kroku jsou příponová pole SA_{12} a SA_0 sloučeny do výsledného příponového pole SA.

Algoritmus 1 Skew

Vstup: pole Vstup - pole celých čísel reprezentujících znaky, resp. v rekurzivních voláních indexy trojic

1: $i12 \leftarrow$ přípony pole Vstup začínající na indexech = 1 nebo 2 mod 3
2: $i0 \leftarrow$ přípony pole Vstup začínající na indexech = 0 mod 3
3: $RadixSort(i12)$ // podle prvních tří znaků
4: $OcislujTrojice(i12)$
5: **if** !unikatniOcislovani **then**
6: $\quad Skew(ocislovani)$ // rekurzivní volání
7: $RadixSort(i12)$ // podle prvního znaku
8: $Merge(i0, i12)$
9: **end**

Výstup: Příponové pole pro pole Vstup

První krok je výpočetně nejnáročnější a je v něm zahrnuto rekurzivní volání. V tomto kroku jsou řazeny přípony na indexech $i \bmod 3 \neq 0$ algoritmem radix sort na základě prvních tří znaků. Poté se iteruje přes seřazené trojice a unikátní nalezené trojice jsou postupně od jedné očíslovány. Pokud bylo všem trojicím přiřazeno unikátní číslo, první krok je hotov. V opačném případě je algoritmus rekurzivně spuštěn nad polem oindexovaných trojic místo vstupního řetězce. Výsledkem prvního kroku je hotové příponové pole všech řetězců začínajících na indexech $i \bmod 3 \neq 0$.

V druhém kroku získáme příponové pole SA_0 tak, že pro indexy $i \bmod 3 = 0$ řadíme páry $(X[i], S_{i+1})$, přičemž lexikografické pořadí přípon začínajících na indexech $i + 1$ již známe z předchozího kroku z SA_{12}, a stačí tedy provést jeden běh radix sortu.

Konečně, ve třetím kroku algoritmu jsou slučovány dílčí příponová pole SA_{12} a SA_0 do finálního výsledku, k čemuž lze využít dobře známé procedury merge z řadicího algoritmu mergesort. Nyní stačí již nez popsat, jakým způsobem lze morovnávat přípony již vytvořených příponových polí. Porovnávání se dělí na dva případy:

- Pokud porovnáváme $S[j]$, kde j mod 3 = 0 s $S[i]$, kde i mod 3 = 1, potřebujeme porovnat dvojice $(X[i], S_{i+1})$ s $(X[j], S_{j+1})$. Platí, že $i+1$ a $j+1$ mod $3 \neq 0$, a tedy informace o pořadí přípon S_{i+1} a S_{j+1} je obsažena v již vytvořeném poli SA_{12} z prvního kroku. Jejich pozice v SA_{12} lze získat z inverzního příponového pole ISA_{12}, které z SA_{12} snadno dopočteme.

- V druhém případě, kdy i mod 3 = 2, porovnáváme trojice $(X[i], X[i+1], S_{i+2})$ s $(X[j], X[j+1], S_{j+2})$, kde přípony ve třetím členu trojice můžeme nahradit lexikografickým rankem z pole ISA_{12} stejně jako v prvním případě.

2.3.4.3 Následné využití příponového pole

V případě, kdy nám jde jen o sekvenční algoritmus, je situace podstatně jednodušší. Na základě hotového příponového pole můžeme snadno sestrojit pole LCP (longest common prefix – nejdelší společná předpona). Nejdříve si zadefinujme funkci lcp (pojmenovanou na základě stejné zkratky), která pro dva řetězce vrací délku jejich maximální společné předpony. Např. tedy platí, že lcp(aabb,aabc) = 3. S touto funkcí už můžeme jednoduše zadefinovat pole LCP[2..N] pro příponové pole SA[1..N] tak, že prvek LCP[i] = lcp($S_{SA[i-1]}, S_{SA[i]}$).

Pro sestrojení pole LCP na základě hotového příponového pole můžeme použít lineární algoritmus z práce [10].

Nyní zadefinujeme pole PrevOcc[1..N] (previous occurrence – předchozí výskyt) a pole LPF[1..N] (longest previous factor – nejdelší předchozí shoda). V poli PrevOcc je na indexu i uložen index j takový, aby $j < i$, $i - j \leq DSIZ$ a zároveň aby lcp(S_i, S_j) byla co největší. Pokud pro žádné j není lcp(S_i, S_j) ≥ 1, pak do $PrevOcc[i]$ uložíme -1. V poli LPF je pak na indexu i uložena hodnota lcp(S_i, S_j), popřípadě také -1.

Nyní můžeme využít algoritmus z [11], který v lineárním čase na základě polí SA a LCP spočte pole PrevOcc a LPF. Na základě těchto polí již pro daný řetězec můžeme snadno sestrojit LZ faktorizaci za pomoci algoritmu 2.

Závěrem je nutno zmínit, že výše uvedený postup je využit pro podřetězce délky $(1+k)$ násobku DSIZ, které postupně bereme ze vstupu. Poté, co je vytvořena LZ faktorizace pro prvních $(1+k)*$DSIZ bajtů vstupního řetězce, používáme podřetězec o $k*$DSIZ bajtů posunutý doprava a vytváříme LZ faktorizace pouze pravých $k*$DSIZ bajtů, dokud není zakódován celý řetězec.

Algoritmus 2 LZ faktorizace

Vstup: pole LPF, pole PrevOcc, pole Vstup

1: $i \leftarrow 1$
2: *list LZ faktorizace* $\leftarrow NULL$
3: **while** $i < n$ **do**
4: *Offset* $\leftarrow$ *i-PrevOcc[i]*
5: *Délka* $\leftarrow$ *max(LPF[i],0)*
6: Připoj trojici *(Offset,Délka,Vstup[Délka+1])* do LZ faktorizace
7: $i \leftarrow$ *i+Délka+1*
8: **end**

Výstup: LZ faktorizace

2.4 Paralelizovatelnost jednotlivých algoritmů na GPU

2.4.1 Hrubou silou, Z-algoritmus a hešování

Tyto algoritmy jsou inherentně sekvenční. Paralelizace můžeme dosáhnout tím, že vstup rozdělíme na kusy, které zkomprimujeme každý zvlášť klasickým sekvenčním způsobem, čímž ale dochází k zhoršení kompresního poměru. Otázkou je, zda jednotlivé kusy dat komprimovat na úrovni pracovních skupin, nebo pracovních položek.

V případě využití pracovních skupin a zvolení malého (4 kB) vyhledávacího bufferu je možné využít lokální paměť jako rychlejší buffer pro zpracovávaná data, dále je možné data do této paměti načíst sloučenými čteními z globální paměti, což jsou ale nepodstatné výhody oproti faktu, že kvůli sekvenčnosti těchto algoritmů lze při následné práci nad daty využít jen jeden výpočetní element z výpočetní jednotky. To činí tento přístup značně neefektivním.

Druhou možností je paralelizovat kompresi již na úrovni pracovních položek. Tento přístup již plně využije všechny dostupné výpočetní elementy OpenCL zařízení. V tomto případě již však nelze využít lokální paměť z důvodu její nízké kapacity, a musí být využívána pomalejší globální paměť. Dalším problémem je, že při kompresi dochází k častému porovnávání jednotlivých bajtů, na základě kterých se program větví. Když tedy výpočetní elementy z jednoho warpu komprimují každý jiný kus dat, nepochybně často dochází k divergenci při průchodu programem, která má negativní dopad na výkon. V neposlední řadě může také tento přístup mít poměrně vysokou hranici velikosti souboru, aby využil všechny dostupné výpočetní jednotky, resp. elementy. Např. v případě, kdy jsou v rámci jedné pracovní položky komprimovány 4 kB dat a kdy je na zařízení k dispozici celkem 1024 výpočetních elementů, musí být velikost souboru alespoň 4 MB, pokud mají být všechny výpo-

četní elementy využity. Tento problém lze při použití naivního algoritmu a Z-algoritmu kompenzovat tím, že snížíme objem dat komprimovaných v rámci jedné pracovní položky, čímž se ale dále zhoršuje úroveň dosažené komprese. U hešování lze objem komprimovaných dat také znížit, ale v tomto případě je před začátkem komprese vždy nutno naplnit hešovací tabulku daty z vyhledávacího bufferu, aby bylo možné hledat shody, a tudíž se zvýší množství redundantních výpočtů (oproti čistě sekvenční verzi).

V rámci praktické části práce jsou všechny tři tyto algoritmy implementovány, přičemž byla na základě výše popsaných výhod a nevýhod zvolena paralelizace komprese na úrovni pracovních položek.

2.4.2 Skew a využití SA

Postup s využitím algoritmu skew je vhodně granulární pro výpočet na GPU. V rámci výpočetních je možné efektivně pracovat na nezávislých kusech vstupu s využitím všech pracovních elementů v dané jednotce pro jednotlivé mezikroky. Implementace efektivní paralelní s sebou ale dále popsané komplikace.

2.4.2.1 Paralelizace algoritmu skew

Všechny kroky algoritmu skew jsou na GPU efektivně paralelizovatelné [9].

Radix sort, který je na začátku prováděn k seřazení dvou třetin přípon na základě prvních tří znaků, a později v algoritmu k seřazení zbývající třetiny, je dobře paralelizovatelný na GPU [12].

Dále, testování unikátnosti prvních tří znaků seřazených dvou třetin přípon lze zparalelizovat tak, že porovnáme každou příponu oproti jejímu předchůdci a uložíme do nového pole příznak 1 v případě, že se liší (jinak 0). K získání očíslování trojic pak stačí provést prefixový součet tohoto pole. Porovnávací krok lze snadno rozdělit mezi jednotlivé pracovní elementy a výpočet prefixového součtu je podobně jako radix sort možno pro GPU efektivně implementovat [13].

Na základě zjištění, zda jsou trojice unikátní, či nikoliv, se algoritmus rozhoduje o rekurzivním zavolání sebe sama. Tento krok představuje značný implementační problém, protože v jazyce OpenCL C není rekurze podporovaná. Originální implementace z [9] je sice také vytvořena v OpenCL, ovšem v jejich případě je počítáno jediné příponové pole na celém GPU, a rekurzivní zavolání je tedy možno vyřešit z programu na hostu opětovným zavolním hlavního kernelu. V našem případě však potřebujeme vypočíst příponové pole pro jiná data v rámci každé pracovní skupiny, a tento přístup nelze využít. Možným řešením je rekurzi simulovat zásobníkem na globální paměti. V literatuře nebyla nalezena práce, která by se již tímto problémem zabývala, a není tedy jasné, zda existuje lepší řešení.

Poslední částí výpočtu skew algoritmu je spojení dvou dílčích příponových polí do finálního celku. Efektivní přístup pro slučování více seřazených polí na GPU, který by byl vhodný pro použití v této práci, je popsán v [14].

2.4.2.2 Paralelizace využití SA

Prvním krokem stejně jako u sekvenčního algoritmu zůstává výpočet pole LCP, jehož výpočet lze vyřešit stejným algoritmem, pouze tentokrát rozdělíme spočtené příponové pole na stejně velké kusy, jejichž počet odpovídá velikosti pracovní skupiny, a v každé pracovní položce využijeme lineárního algoritmu pro spočtení daného kusu.

Poté už se situace komplikuje. Již nelze použít stejný algoritmus jako v minulém případě pro spočtení polí PrevOcc a LPF. Místo toho budeme potřebovat další pole jako mezikrok, které si nyní zadefinujeme na základě problému ANSV.

Problém ANSV (All nearest smaller values – všechny nejbližší menší hodnoty) je zadefinován následovně [15]: Nechť $A = (a_1, a_2, ..., a_n)$ je uspořádaná n-tice přirozených čísel. Pak pro každé $a_i, 1 \leq i \leq n$ chceme naleznout dvě nejbližší čísla v A taková, že jsou menší než a_i (pokud existují). Levé nejbližší menší číslo $a_j, j < i$ nazveme psv (previous smaller value – předchozí menší hodnota), a pravé nejbližší menší číslo $a_k, k > i$ nazveme nsv (next smaller value – následující menší hodnota). Pro naše účely si spočteme pole PSV a NSV, která obsahují na indexu i index hodnoty psv, resp. nsv pro i. prvek v příponovém poli. Ty spočteme na základě postupu popsaného v [9], kde se využívá sekvenčního algoritmu pro výpočet LPF a PrevOcc již zmíněného v sekci 2.3.4.3, ale v tomto případě je prováděn paralelně výpočetními elementy nad kusy SA, a je využit jen pro výpočet polí PSV a NSV.

Konečně, práce [16] nabízí dva způsoby, kterými lze z polí PSV a NSV efektivně paralelně spočíst pole LPF a PrevOcc, se kterými již naložíme stejně jako v sekci 2.3.4.3 ke spočtení LZ faktorizace.

Tímto přístupem bylo v [5] dosaženo 4–5násobného zrychlení oproti nejrychlejším sekvenčním implementacím LZ77.

Kapitola 3

Implementace

Tato kapitola popisuje způsoby, jakými byl LZ77 v praktické části této práce implementován. V rámci práce byly za tímto účelem vytvořeny tři oddělené aplikace využívající hešování, Z-algoritmus a vyhledávání hrubou silou pro vyhledávní shodných sekvencí dat. Dále byla implementována varianta pro otestování paralení dekomprimace, která je založena na metodě využívající hešování.

3.1 Implementační rozhodnutí ohledně granularizace problému

Společným prvkem všech implementovaných metod je, že soubor rozdělí na bloky, které jsou za běhu programu zkomprimovány jedním spuštěním kernelu. Tyto bloky jsou komprimovány nezávisle na předchozích blocích. V důsledku je možné tyto bloky později paralelně dekomprimovat, ovšem dochází také k mírnému zhoršení úrovně dosažené komprese.

Jeden blok dat se dále dělí na podčásti, které jsou faktorizovány v rámci jedné pracovní položky. Množství komprimovaných dat v rámci jedné pracovní položky bylo stanoveno na 4 kB jako kompromis mezi dvěma faktory – zvýšením tohoto atributu dojde ke zlepšení dosažené komprese, ale také se zvýší hranice velikosti souboru, při které už budou využity všechny výpočetní elementy daného zařízení. Zmenšení má opačný efekt. V případě implementace využívající hešování je navíc v rámci pracovní položky nejdříve nutné naplnit hešovací tabulku na základě předchozích dat. Snížením objemu zpracovávaných dat se tedy zvyšuje celkový objem těchto výpočtů.

3.2 Implementace části aplikace běžící na OpenCL hostiteli

Při spuštění aplikace jsou části na hostiteli předány kromě souboru ke zkomprimování další dva paraemtry:

ID OpenCL zařízení ID zařízení, na kterém budou prováděny výpočty. Tato umělá ID jsou programem přiřazena všem dostupným OpenCL zařízením a jsou zobrazitelná spuštěním se speciálním parametrem.

Počet vytvářených pracovních položek V rámci jednoho spuštění kernelu je počet vytvářených pracovních položek předávaným parametrem při spouštění aplikace, protože optimální počet k dosažení nejlepšího výkonu je závislý na architektuře využívaného zařízení. Zadaný počet se dále seskupí do pracovních skupin po 64 pracovních položkách.

Část aplikace běžící na hostitelském zařízení se řídí uvedeným nástinem v sekci 1.5.1. OpenCL kontext a příkazová fronta jsou vytvořeny nad OpenCL zařízením uvedeným v parametru. V dalším kroku jsou vytvořeny paměťové buffery a je načten a zkompilován kernel v závislosti na konkrétní implementaci.

3.2.1 Komprimace

Ve všech třech typech implementací jsou vytvářeny buffery v globální paměti pro vstupní data, výstupní data, dále pro velikost aktuálně zpracovávaného bloku dat a buffer pro uložení velikostí LZ faktorizací vypočtených v jednotlivých pracovních položkách. Výstupní buffer má trojnásobnou velikost oproti vstupnímu bufferu pro nejhorší možný případ, kdy nejsou nalezeny žádné / téměř žádné shody. Předposlední buffer je nutnou doplňující informací v případě, kdy už není dostatek vstupních dat pro všechny vytvořené pracovní položky a některé pracovní položky na základě této informace rovnou skončí, popřípadě komprimují menší blok dat než 4 kB. Poslední buffer je důležitou výstupní informací pro program na hostu pro složení jednotlivých dílčích LZ faktorizací do finální LZ faktorizace celého bloku.

V implementaci hešování jsou dále dva buffery v globální paměti navíc, které jsou v pracovních položkách využity pro jejich hešovací tabulky.

3.2.2 Dekomprimace

V případě varianty umožňující paralelní dekomprimaci je pozměněn způsob komprese, aby před zkomprimovaná data umístil hlavičku zanedbatelné velikosti (typicky 8 B na 4 MB komprimovaných dat) s potřebnými informacemi navíc. V hlavičce je uveden počet bloků, ze kterých se zkomprimovaný soubor skládá, velikost jednoho bloku v nezkomprimované formě a indexy, na

kterých začínají zkomprimované bloky. Dekomprimačnímu kernelu jsou poté předány paměťové buffery pro vstupní data, výstupní data, velikost nezkomprimovaného bloku, indexy, kde začínají zkomprimované bloky pro aktuálně dekomprimované bloky, index poslední spuštěné pracovní položky a velikost posledního dekomprimovaného bloku.

3.3 Implementace OpenCL kernelů

Velikost vyhledávacího bufferu je zvolena na 4096 B a dopředného bufferu na 16 B. Díky tomu je velikost odkazů zarovnána na celé bajty (12 bitů pro offset, 4 bity pro délku), což usnadňuje implementaci. U metody hešování jsou rozměry hešovacích tabulek HSIZ a DSIZ zvoleny po vzoru programu gzip na velikost totožnou s velikostí vyhledávacího bufferu.

3.3.1 Komprimace

Samotné kernely se obvykle skládají z úvodní části, hlavního komprimačního cyklu a závěrečného ukládaní potřebných metadat pro program na hostu. V úvodní části se určuje přesný kus zpracovávaných dat v dané pracovní položce na základě jeího globálního ID, v případě hešování je pak i dopočítávan obsah hešovací tabulky na základě předchozích dat. Základní komprimační cyklus vytvořených kernelů je vidět ve zkráceném výpisu kódu 3. Uvnitř cyklu probíhá výpočet LZ faktorizace, přičemž implementace vyhledávání shod je závislá na dané implementaci. Pro simulaci dopředných a vyhledávacích bufferů jsou využity indexy do vstupních dat. V závěrečné fázi kernelu je uložena délka vypočtené LZ faktorizace dané pracovní položky do příslušného argumentu kernelu.

3.3.2 Dekomprimace

Struktura dekomprimačního kernelu je stejná jako u komprimačních kernelů; v úvodu jsou pro danou pracovní položku určí vstupní data, se kterými bude pracovat, a posléze jsou v hlavním cyklu data dekomprimovány klasickým sekvenčním postupem tak, jak je popsán v 2.2.1.

```
while (frontBufferInd < end)
{
    searchBufferInd = ...; // spočtení začátku vyhledávacího
                           // bufferu s ohledem na začátek bloku
    frontBufferLen = ...; // spočtení délky dopředného bufferu
                          // s ohledem na konec vstupních dat
                          // dané prac. položky

    substringSearch(...); // vyhledání shody ve vyhledávacím
                          // bufferu; buďto hrubou silou,
                          // Z-algoritmem, nebo hešováním

    writeLZ77Codeword(...); // přidání další trojice
                            // k dosavadní LZ faktorizaci
                            // na základě nalezené shody

    frontBufferInd += foundLen + 1; // posunutí vyhledávacího
                                    // bufferu
}
```

Výpis kódu 3: Hlavní cyklus komprimačního kernelu

Kapitola 4

Testování

V této kapitole jsou uvedeny parametry implementované kompresní aplikace z běhu na vybraných korpusech a je zde popsáno, jakým způsobem byly tyto hodnoty naměřeny.

4.1 Způsob měření

Měření bylo prováděno nad soubory z korpusů Calgary [17], Canterbury [18], Silesia [19] a Prague Corpus [20]. Na použitých zařízeních byly komprimovány všechny soubory z vybraných korpusů všemi implementovanými metodami s proměnlivým počtem vytvořených pracovních položek (dále PVPP) při jednom spuštění kernelu. Tato kapitola bude rozebírat naměřené hodnoty nad korpusem Silesia, protože narozdíl od ostatních je tvořen relavitně velkými soubory (o velikosti z rozmezí 6–51 MB). To je pro naše účely vhodné z toho důvodu, že soubory z ostatních korpusů často nedosahují minimální velikosti potřebné k využití všech dostupných výpočetních elementů na použité grafické kartě, a zároveň při výpočtu na procesoru často dosahují hodnot hraničících s možnou chybou měření. V příloze A je uvedena obdoba všech tabulek o běhu na korpusu Silesa v této kapitole pro korpus Prague Corpus. Dále jsou v této příloze uvedeny pro soubory z korpusů Calgary a Canterbury jen dosažené kompresní poměry.

Před změřením kompresního poměru byla LZ faktorizace zakódována aritemtickým kodérem z [21]. Každá část trojic z LZ faktorizace byla kódována zvlášť do stejného souboru. Velikostí tohoto souboru je následně vydělena velikost původního souboru, čímž je získán naměřený kompresní poměr.

Kromě porovnání výsledků implementace pro CPU a GPU jsou uvedeny nekomentované doby běhu známého komprimačního programu gzip, který byl spouštěn s výchozím nastavením míry komprese (parametr –6).

Pro měření doby běhu komprese a dekomprese byly použity prostředky poskytované OpenCL popsané v následující sekci 4.1.1.

4.1.1 OpenCL časoměřič

OpenCL poskytuje objekty nazvané `cl_event`, tedy v překladu události, které se dají vytvořit k synchronizaci zadávaných funkcí do příkazové fronty, popřípadě ale i k profilování. Při vytvoření příkazové fronty se speciální vlastností `CL_QUEUE_PROFILING_ENABLE` je následně možno při zadávaní příkazů do příkazové fronty zadat jako argument událostní objekt, a následně z něj funkcí `clGetEventProfilingInfo()` vyčíst různé informace včetně toho, jak dlouho daný příkaz na OpenCL zařízení odpovídající dané frontě běžel. Tímto způsobem byly získány časové údaje ohledně trvání paměťových přesunů mezi hostem a OpenCL zařízením a doby běhu kernelů.

4.2 Použitý hardware

Testování bylo prováděno na grafické kartě AMD Vega Frontier Edition Liquid a procesoru AMD Ryzen 7 1700X s 32 GB dostupné operační paměti. Kód byl kompilován Microsoft C/C++ kompilátorem zahrnutým v rámci Microsoft Visual Studio 2017. Aplikace byla vyvíjena se starší specifikací OpenCL 1.2, protože autorovi dostupná grafická karta pro vývoj podporuje nejvýše tuto verzi. Testovaná grafická karta podporuje i novější verze, je možné, že dosažený výkon by mohl být vylepšen s využitím přidaných vlastností novějších verzí.

4.3 Naměřené hodnoty

4.3.1 Vliv počtu vytvářených pracovních položek

Nejdříve se v naměřených hodnotách zaměříme na optimalizaci PVPP při jednom spuštění kernelu. V rámci testování jsou soubory komprimovány s 1024, 1536, 2048, 2560, 3072, 4096 a 5120 pracovními položkami na jedno spuštění. V tabulce 4.1 lze vidět, jak tento parametr ovlivňuje dobu trvání komprese na GPU, v tabulce 4.2 totéž pro CPU.

U všech testovaných metod (nejen na znázorněných datech z hešování) je při výpočtu na grafické kartě pozorovatelné zrychlení při zvyšování PVPP. Na tomto zrychlení se významně podíli dva faktory. Prvním je, že námi využívaná grafická karta disponuje 4096 výpočetními elementy, pokud je tedy PVPP menší než tento počet, nelze ani všechny výpočetní elementy využít. Druhým faktorem je fakt, že zvyšováním PVPP roste také velikost dílčích bloků, na které se vstupní soubor rozdělí a pokud se po zvýšení PVPP zmenší celkový počet těchto dílčích bloků, sníží se objem režijních nákladů způsobený spouštěním běhu kernelů. Dále v této kapitole a v příloze A budou všechna data z měření na GPU naměřena s hodnotou PVPP 4096.

V případě výpočtu probíhajícího na CPU je situace méně jednoznačná. Stále lze u většiny pozorovat zkracování doby běhu při zvyšování PVPP, ale při navyšování přes 4096 pracovních položek z nejasných důvodů doba běhu

Tabulka 4.1: Doba trvání [s] výpočtu komprimace souborů z korpusu Silesia na GPU s použitím hešování pro rozdílné PVPP

Soubor	**Počet vytvářených pracovních položek**						
	1024	**1536**	**2048**	**2560**	**3072**	**4096**	**5120**
dickens	1,548	1,122	1,082	0,586	0,584	0,580	0,580
mozilla	136,396	99,635	80,337	58,251	60,879	51,013	39,014
mr	19,789	14,084	14,034	7,312	7,312	7,231	7,311
nci	37,558	28,435	19,037	19,247	14,435	9,876	10,106
ooffice	7,976	5,952	6,029	6,033	6,026	6,029	6,026
osdb	0,951	0,649	0,641	0,338	0,338	0,338	0,337
reymont	1,712	1,703	1,051	1,051	1,051	1,051	1,051
samba	36,205	24,705	24,304	20,147	16,197	16,200	16,198
sao	1,701	1,491	0,956	0,958	0,937	0,954	0,943
webster	4,037	2,900	2,124	1,718	1,743	1,352	1,027
xml	3,989	3,345	3,302	3,302	3,348	3,301	3,345
x-ray	0,378	0,287	0,252	0,162	0,157	0,159	0,158

Tabulka 4.2: Doba trvání [s] výpočtu komprimace souborů z korpusu Silesia na CPU s použitím hešování pro rozdílné PVPP

Soubor	**Počet vytvářených pracovních položek**						
	1024	**1536**	**2048**	**2560**	**3072**	**4096**	**5120**
dickens	0,146	0,092	0,076	0,059	0,053	0,105	0,075
mozilla	1,988	1,980	1,837	1,680	1,771	1,479	1,719
mr	0,518	0,367	0,374	0,403	0,487	0,552	0,504
nci	0,979	1,181	1,069	1,028	1,094	0,985	0,959
ooffice	0,162	0,088	0,093	0,081	0,077	0,077	0,078
osdb	0,089	0,054	0,051	0,041	0,036	0,029	0,077
reymont	0,122	0,085	0,065	0,047	0,046	0,045	0,069
samba	0,539	0,510	0,395	0,495	0,521	0,339	0,352
sao	0,117	0,080	0,065	0,056	0,048	0,048	0,120
webster	0,293	0,173	0,295	0,182	0,320	0,180	0,307
xml	0,059	0,044	0,034	0,031	0,025	0,023	0,019
x-ray	0,066	0,056	0,041	0,034	0,034	0,030	0,030

v mnoha případech roste, a některé soubory (např. mr, webster) se tomuto trendu vymykají úplně. Dále se u menších souborů doba výpočtu pohybuje v řádu desítek milisekund a hodnoty naměřené u těchto souborů začínají ztrácet výpovědní hodnotu kvůli nepřesnostem měření. Po normalizaci hodnot pro jednotlivé soubory běží komprese v průměru přes všechny soubory nejkratší dobu při využití PVPP rovnému 4096, a z tohoto důvodu jsou následující data z měření na CPU v této kapitole a v příloze A s touto hodnotou, stejně jako

u GPU.

4.3.2 Rychlost běhu jednotlivých implementací

Doba běhu komprese na grafické kartě při využití vyhledávání s pomocí hešování, hrubé síly a Z-algoritmu na souborech z korpusu Silesia je zaznamenána v následující tabulce 4.3.

Tabulka 4.3: Doba trvání [s] výpočtu komprimace souborů z korpusu Silesia na GPU. V značí dobu výpočtu, PP dobu trvání paměťových přesunů mezi hostem a OCL zařízením

Soubor	**Hešování**		**Hrubou silou**		**Z-algoritmus**	
	V	**PP**	**V**	**PP**	**V**	**PP**
dickens	0,580	0,015	3,502	0,004	4,171	0,004
mozilla	51,013	0,060	45,767	0,020	36,785	0,018
mr	7,231	0,015	9,521	0,004	5,509	0,005
nci	9,876	0,030	9,735	0,010	9,979	0,010
ooffice	6,029	0,014	8,363	0,005	7,608	0,004
osdb	0,338	0,015	6,036	0,004	7,014	0,005
reymont	1,051	0,015	6,519	0,004	7,943	0,006
samba	16,200	0,030	24,043	0,010	16,539	0,009
sao	0,954	0,015	5,720	0,004	6,540	0,004
webster	1,352	0,046	9,307	0,014	11,007	0,014
xml	3,301	0,014	4,364	0,004	4,431	0,005
x-ray	0,159	0,015	4,792	0,005	5,652	0,004

Na většině souborů si hešování vede jednoznačně nejlépe, výjimkami jsou hlavně soubory mozilla, nci, samba a některé další. Problémem těchto souborů při hešování je ten, že se v těchto souborech v krátkém horizontu často vyskytuje jen malá podmnožina všech možných hodnot bajtu. Při ukládání do hešovací tabulky pak dochází k mnoha kolizím a při následném vyhledávání předchozích shod se zdlouhavě prohledává řetízek odpovídající dané hodnotě hešovací funkce, což do velké míry degraduje výkon. Překvapením je, že metoda vyhledávání hrubou silou na mnoha souborech běží rychleji než asymptoticky lepší Z-algoritmus. To je pravděpodobně zapříčiněno tím, teoretický nejhorší scénář pro vyhledávání hrubou silou, který Z-algoritmus optimalizuje, prakticky příliš často nenastává, a přidané režijní náklady na některých souborech tento algoritmus oproti vyhledávání hrubou silou zpomalují.

Hodnoty z totožných měření, ale tentokrát provedených na CPU, jsou vypsány v tabulce 4.4. Při výpočtu probíhajícím na CPU je hešování zdaleka nejlepší, v průměru je komprese testovaných souborů 40krát rychlejší než vyhledávání hrubou silou. Dále lze z naměřených hodnot vyvodit, že Z-algoritmus

není vhodným algoritmem pro zrychlení vyhledávání shod pro účely LZ faktorizace na CPU; výpočet s využitím Z-algoritmu je pomalejší než vyhledávání hrubou silou na všech souborech, a to v průměru 1,31krát.

Tabulka 4.4: Doba trvání [s] výpočtu komprimace souborů z korpusu Silesia na CPU. V značí dobu výpočtu, PP dobu trvání paměťových přesunů mezi hostem a OCL zařízením

Soubor	**Hešování**		**Hrubou silou**		**Z-algoritmus**	
	V	**PP**	**V**	**PP**	**V**	**PP**
dickens	0,105	0,062	2,337	0,025	2,916	0,025
mozilla	1,479	0,097	11,929	0,034	15,901	0,034
mr	0,552	0,065	1,801	0,025	2,321	0,025
nci	0,985	0,075	3,185	0,029	3,654	0,028
ooffice	0,077	0,061	2,009	0,024	2,669	0,024
osdb	0,029	0,057	4,165	0,021	5,666	0,021
reymont	0,045	0,063	1,473	0,026	1,797	0,027
samba	0,339	0,072	4,748	0,027	6,032	0,026
sao	0,048	0,059	2,918	0,022	3,892	0,022
webster	0,180	0,087	6,053	0,033	8,480	0,033
xml	0,023	0,059	2,329	0,022	3,560	0,022
x-ray	0,030	0,063	1,011	0,027	1,276	0,026

Poslední tabulka 4.5 poskytuje závěrečné srovnání dob běhu jednotlivých implementací na GPU a CPU společně i s uvedením doby běhu gzipu. Na všech testovaných souborech při využití implementovaných metod na GPU probíhal výpočet delší dobu, ve většině případů v rozmezí od 2násobného do 10násobného zpomalení. Výjimkou je hešování některých souborů, u kterých dochází k mnoha hešovacím kolizím, při kterých je zpomalení oproti CPU až 40násobné, pravděpodobně kvůli extrémní divergenci warpů.

Běh aplikací na dvou natolik rozdílných zařízeních jako GPU a CPU lze obecně těžko srovnávat, v tomto případě je ale vzhledem k míře, do jaké je běh na GPU pomalejší, pravděpodobně bezpečně možné usoudit, že implementované metody nevyužívají výpočetní model GPU efektivně kvůli problémům popsaným v sekci 2.4.

Při porovnávání naměřených časů programu gzip s naměřenými časy implementovaných kompresních metod je nutno vzít v potaz, že v rámci uvedené doby běhu gzipu je navíc zahrnuto zakódování gzipem spočtené LZ faktorizace do Huffmanova kódu, zatímco u dob u implementovaných kompresních metod se jedná o dobu výpočtu samotné LZ faktorizace.

Tabulka 4.5: Doba trvání [s] výpočtu komprimace souborů z korpusu Silesia jednotlivými implementovanými metodami na GPU a CPU + programem Gzip na CPU

Soubor	Hešování		Hrubou silou		Z-algoritmus		Gzip
	GPU	CPU	GPU	CPU	GPU	CPU	
dickens	0,595	0,167	3,506	2,362	4,176	2,941	0,808
mozilla	51,073	1,576	45,787	11,963	36,803	15,935	2,683
mr	7,246	0,618	9,525	1,826	5,513	2,346	0,731
nci	9,906	1,060	9,745	3,214	9,988	3,683	0,644
ooffice	6,043	0,138	8,367	2,033	7,613	2,693	0,431
osdb	0,353	0,087	6,041	4,185	7,019	5,686	0,459
reymont	1,066	0,109	6,523	1,499	7,949	1,824	0,477
samba	16,230	0,411	24,053	4,775	16,548	6,058	0,757
sao	0,969	0,107	5,725	2,941	6,544	3,915	0,598
webster	1,398	0,267	9,321	6,086	11,021	8,513	2,044
xml	3,316	0,082	4,368	2,351	4,436	3,582	0,497
x-ray	0,173	0,093	4,797	1,037	5,656	1,302	0,166

4.3.3 Paralelní dekomprese

Naměřené doby běhu dekomperese z varianty hešování s paralelní dekompresí jsou společně s dobou běhu sekvenční dekomprese a dobou dekomprese gzipu uvedeny v tabulce 4.6. I v tomto případě je tímto upozorněno, že v rámci času uvedeného u programu gzip je navíc zahrnuto dekódování z Huffmanova kódu.

Paralelní dekomprese prováděná s uvedenými parametry běží při nastavení PVPP na 1024 v průměru 9,48krát pomaleji, při vyšších hodnotách PVPP se zpomalení dále zvyšuje. Na zpomalení se pravděpodobně podílí především tyto faktory:

- V rámci jedné pracovní položky je nutné dekomprimovat příliš mnoho dat – celý blok vstupu. Protože se všechny soubory z korpusu Silesia pro jakýkoliv PVPP z testovaných hodnot rozdělí nejvíce na řádově desítky bloků, vytvoří se maximálně desítky pracovních položek. To je problém, protože pro optimální využití 4096 výpočetních elementů testované GPU jsou potřebné tisíce pracovních položek. Tento problém nelze jednoduše vyřešit:
  - Snížením PVPP při kompresi nastane identický problém při kompresi a dojde ke zpomalení komprese.
  - Snížením množství dat komprimovaného v jedné pracovní položce se při využití hešování, které v rychlosti komprimace dosahuje nejlepších výsledků, zvýší množství prováděných redundantních výpočtů (před začátkem faktorizace v pracovní položce je nutné dopočí-

Tabulka 4.6: Doba trvání [s] výpočtu dekomprimace souborů z korpusu Silesia klasickým sekvenčním přístupem, paralelním přístupem + doba dekomprimace programu Gzip

Soubor	**Sekv.**	**Paralelní s různým PVPP**				**Gzip**
		1024	**2048**	**3072**	**4096**	
dickens	0,172	1,657	2,492	2,800	2,795	0,109
mozilla	0,734	6,690	10,101	11,117	13,131	0,433
mr	0,140	1,189	2,262	2,292	2,286	0,103
nci	0,297	2,245	4,406	6,696	6,654	0,166
ooffice	0,125	1,159	1,180	1,184	1,183	0,091
osdb	0,219	2,200	2,272	3,300	3,300	0,108
reymont	0,094	1,147	1,125	1,189	1,186	0,092
samba	0,297	3,351	4,418	5,507	5,546	0,170
sao	0,172	1,169	2,223	2,223	2,220	0,095
webster	0,578	4,487	7,721	9,921	9,923	0,342
xml	0,203	1,128	1,147	1,144	1,148	0,114
x-ray	0,063	1,183	2,256	2,256	2,258	0,049

tat obsah hešovacích tabulek na základě předchozích dat) a dojde ke zpomalení komprese.

  - Využíváním paralelní dekomprimace pouze na dostatečně velké soubory (v řádech GB) se podstatně snižuje její využitelnost.

- Dále má dekomprese podobné problémy jako komprese:

  - Každá pracovní položka pracuje s velkým množstvím paměti najednou a nelze tedy využít omezenou a rychlejší (lokální a privátní) paměť, která se nachází přímo na čipu, pro zrychlení přístupů do paměti.
  - Při dekompresi může často docházet k divergenci warpů při kopírování odkazovaných dat (při dekódování trojice z LZ faktorizace je délka shody pravděpodobně často různá u výpočetních elementů v rámci jednoho warpu).

Výše popsané faktory činí výpočet paralelní dekomprimace na GPU nepříliš efektivním.

4.3.4 Kompresní poměr

V této práci je porovnán kompresní poměr implementované komprese s kompresním poměrem dosahovaným programem gzip. Kompresní poměry dosahované jednotlivými implementacemi jsou velmi podobné, v tabulce 4.7 jsou uvedeny naměřené hodnoty dosažené s využitím hešování.

Tabulka 4.7: Velikosti souborů [B] z korpusu Silesia před a po zkomprimování programem gzip a implementovaným LZ77 s využitím hešování a následným zakódováním aritmetickým kodérem, kompresní poměry

Soubor	Velikost	Gzip	GZ KP	Hešování	Heš. KP
dickens	10 192 446	3 868 778	2,635	5 033 074	2,025
mozilla	51 220 480	19 048 745	2,689	22 415 651	2,285
mr	9 970 564	3 690 572	2,702	3 897 342	2,558
nci	33 553 445	3 199 453	10,487	5 070 332	6,618
ooffice	6 152 192	3 097 081	1,986	3 658 555	1,682
osdb	10 085 684	3 739 578	2,697	6 905 043	1,461
reymont	6 627 202	1 858 629	3,566	2 565 026	2,584
samba	21 606 400	5 460 537	3,957	7 712 926	2,801
sao	7 251 944	5 332 826	1,360	5 852 712	1,239
webster	41 458 703	12 201 980	3,398	16 603 720	2,497
xml	5 345 280	691 992	7,724	1 279 340	4,178
x-ray	8 474 240	6 037 794	1,404	6 230 467	1,360

Na všech souborech korpusu Silesia dohromady dosahují implementované metody 1,2krát menšího kompresního poměru než gzip. Tato ztráta je pravděpodobně způsobena převážně následujícími důvody:

- Velikosti vyhledávacího a dopředného bufferu jsou v praktické části této práce nastaveny vždy na 4096, resp. 16 za účelem zjednodušení implementace, zatímco v implementaci gzipu je velikost vyhledávacího bufferu proměnlivá od 4096 do 32768 a velikost dopředného bufferu je 256, což může pomoci dosáhnout lepšího kompresního poměru.
- Kvůli rozdělení vstupního řetězce na části dochází při faktorizací částí k neefektivní faktorizaci začátku a konce těchto částí.
- Blok dat, který je zkomprimován jedním spuštěním kernelu, je komprimován samostatně, tedy bez využití dat, které mohly předcházet tomuto bloku. V důsledku může být začátek tohoto bloku o velikosti vyhledávacího bufferu neefektivně zfaktorizován.

Závěr

Cílem rešeršní části práce práce byla rešerše algoritmu LZ77, možných paralelních implementací a rešerše frameworku OpenCL za účelem jeho využití pro výpočet na GPU. V rámci praktické části práce bylo cílem algoritmus LZ77 s pomocí frameworku OpenCL naimplementovat pro běh na GPU a otestovat rychlost běhu a kompresní poměr při výpočtu na CPU a GPU a porovnat naměřené hodnoty.

Cíle byly úspěšně splněny. V rámci praktické části nebylo při testování dosaženo s žádnou z implementovaných metod zrychlení při běhu na GPU oproti CPU. Běh komprese na grafické kartě byl u většiny souborů z korpusu Silesia 2–10krát pomalejší než na CPU. Dosažený kompresní poměr implementace byl totožný na obou zařízeních, konkrétně na korpusu Silesia byl celkový kompresní poměr 2,58. Známý kompresní program gzip s výchozím nastavením dosahoval na tomto korpusu o 20 % lepšího kompresního poměru.

Implementace algoritmu Skew a souvisejících procedur, pro které v rámci této práce nebyl prostor, ale jsou v rešeršní části popsány, by byly vhodným rozšířením praktické části této práce a dle [5] s nimi lze dosáhnout až 4–5násobného zrychlení oproti nejrychlejším sekvenčním implementacím na CPU.

Bibliografie

1. MUNSHI, Aaftab et al. *OpenCL Programming Guide*. Ann Arbor, Michigan: Addison-Wesley, 2011. ISBN 978-0-321-74964-2.
2. MUNSHI, Aaftab (ed.). The OpenCL Specification. 2012, č. 1.2. Dostupné také z: `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf`.
3. OpenCL Programming Guide for the CUDA Architecture. 2009, č. 2.3. Dostupné také z: `http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf`.
4. NVIDIA OpenCL Best Practices Guide. 2009. Dostupné také z: `https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf`.
5. CHING, Bryan. *Optimizing Lempel-Ziv factorization for the GPU architecture*. San Luis Obispo, California, USA, 2014. Diplomová práce. California Polytechnic State University.
6. SADAKANE, Kunihiko; IMAI, Hiroshi. Improving the Speed of LZ77 Compression by Hashing and Suffix sorting. *IEICE TRANS. FUNDAMENTALS*. 2000, roč. E83-A, č. 12, s. 2690–2691. ISSN 1745-1337.
7. HORDĚJČUK, Vojtěch. Kompresní algoritmus LZ77 [online]. 2017 [cit. 2018-04-15]. Dostupné z: `http://voho.eu/wiki/algoritmus-lz77/`.
8. WANG, Jeffrey. Z Algorithm [online]. 2011 [cit. 2018-04-16]. Dostupné z: `http://codeforces.com/blog/entry/3107`.
9. DEO, Mrinal; KEELY, Sean. Parallel Suffix Array and Least Common Prefix for the GPU. *ACM SIGPLAN Notices*. 2013, roč. 48, s. 197–206.
10. KASAI, Toru; LEE, Gunho; ARIMURA, Hiroki; ARIKAWA, Setsuo; PARK, Kunsoo. Linear-time longest-common-prefix computation in suffix arrays and its applications. In: *Proc. of CPM'01*. Springer-Verlag, 2001, s. 181–192.

11. CROCHEMORE, Maxime; ILIE, Lucian. Computing Longest Previous Factor in linear time and applications. *Information Processing Letters*. 2008, roč. 106, č. 2, s. 75–80. Dostupné z DOI: 10.1016/j.ipl.2007.10.006.

12. MERRILL, Duane G.; GRIMSHAW, Andrew S. Revisiting Sorting for GPGPU Stream Architectures. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. Vienna, Austria: ACM, 2010, s. 545–546. PACT '10. ISBN 978-1-4503-0178-7. Dostupné z DOI: 10.1145/1854273.1854344.

13. HARRIS, Mark; SENGUPTA, Shubhabrata; OWENS, John D. Parallel prefix sum (scan) with CUDA. *GPU gems*. 2007, roč. 3, č. 39, s. 851–876.

14. SATISH, N.; HARRIS, M.; GARLAND, M. Designing efficient sorting algorithms for manycore GPUs. In: *2009 IEEE International Symposium on Parallel Distributed Processing*. 2009, s. 1–10. ISSN 1530-2075. Dostupné z DOI: 10.1109/IPDPS.2009.5161005.

15. BERKMAN, Omer; SCHIEBER, Baruch; VISHKIN, Uzi. Optimal Doubly Logarithmic Parallel Algorithms Based On Finding All Nearest Smaller Values. 1993, roč. 14, s. 344–370.

16. SHUN, Julian; ZHAO, Fuyao. Practical Parallel Lempel-Ziv Factorization. In: *Proceedings of the 2013 Data Compression Conference*. Washington, DC, USA: IEEE Computer Society, 2013, s. 123–132. DCC '13. ISBN 978-0-7695-4965-1. Dostupné z DOI: 10.1109/DCC.2013.20.

17. WITTEN, Ian; BELL, Timothy; CLEARY, J. Calgary corpus. *University of Calgary, Canada*. 1987.

18. ARNOLD, R.; BELL, T. A corpus for the evaluation of lossless compression algorithms. In: *Data Compression Conference, 1997. DCC '97. Proceedings*. 1997, s. 201–210. ISSN 1068-0314. Dostupné z DOI: 10.1109/DCC.1997.582019.

19. DEOROWICZ, Sebastian. Silesia compression corpus [online]. 2014 [cit. 2018-05-05]. Dostupné z: http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia.

20. HOLUB, J.; REZNICEK, J.; SIMEK, F. Lossless Data Compression Testbed: ExCom and Prague Corpus. In: *2011 Data Compression Conference*. 2011, s. 457–457. ISSN 1068-0314. Dostupné z DOI: 10.1109/DCC.2011.61.

21. WHEELER, Fred. Adaptive arithmetic coding source code [online]. 2012 [cit. 2018-05-05]. Dostupné z: http://www.fredwheeler.org/ac/.

Dodatek A

Míry běhu na korpusech Calgary, Canterbury a Prague Corpus

Tabulka A.1: Doba trvání [s] výpočtu komprimace souborů z korpusu Prague Corpus na GPU s použitím hešování pro rozdílné PVPP

Soubor	**Počet vytvářených pracovních položek**						
	1024	**1536**	**2048**	**2560**	**3072**	**4096**	**5120**
abbot	0,350	0,351	0,353	0,350	0,344	0,349	0,352
age	6,169	6,077	6,164	6,159	6,164	6,166	6,163
bovary	0,829	0,832	0,825	0,829	0,824	0,827	0,834
collapse	0,006	0,006	0,006	0,006	0,006	0,006	0,006
compress	0,179	0,181	0,184	0,180	0,176	0,181	0,181
corilis	4,256	4,250	4,200	4,201	4,253	4,197	4,252
cyprus	1,104	1,105	1,108	1,105	1,109	1,107	1,108
drkonqi	3,768	3,767	3,762	3,765	3,714	3,763	3,762
emission	6,307	6,387	6,389	6,305	6,390	6,388	6,304
firewrks	1,617	1,627	1,634	1,633	1,633	1,627	1,627
flower	0,427	0,328	0,295	0,192	0,188	0,190	0,188
gtkprint	3,486	3,483	3,436	3,436	3,484	3,436	3,485
handler	0,178	0,178	0,181	0,180	0,180	0,180	0,180
higrowth	1,009	1,006	1,003	1,005	0,994	1,003	1,003
hungary	1,210	1,206	1,208	1,210	1,195	1,207	1,207
libc06	1,170	1,170	1,171	1,170	1,169	1,169	1,169
lusiadas	7,099	7,073	7,177	7,211	7,180	7,175	7,085
lzlindmt	0,088	0,088	0,089	0,089	0,090	0,089	0,089
mailflder	1,263	1,261	1,262	1,261	1,245	1,262	1,262
mirror	4,263	4,261	4,258	4,209	4,259	4,262	4,205
modern	0,324	0,317	0,321	0,325	0,325	0,325	0,325
nightsht	1,065	0,909	0,661	0,729	0,739	0,483	0,480
render	0,088	0,087	0,087	0,087	0,088	0,086	0,088
thunder	3,534	3,514	3,522	3,530	3,492	3,521	3,514
ultima	0,443	0,434	0,438	0,445	0,440	0,437	0,438
usstate	0,080	0,080	0,079	0,079	0,080	0,080	0,079
venus	2,246	1,958	1,586	1,576	1,535	1,006	0,995
w01vett	4,243	4,245	4,292	4,298	4,293	4,290	4,236
wnvcrdt	4,078	4,083	4,035	4,085	4,077	4,034	4,087
xmlevent	0,084	0,084	0,085	0,085	0,085	0,085	0,084

Tabulka A.2: Doba trvání [s] výpočtu komprimace souborů z korpusu Prague Corpus na CPU s použitím hešování pro rozdílné PVPP

Soubor	**Počet vytvářených pracovních položek**						
	1024	**1536**	**2048**	**2560**	**3072**	**4096**	**5120**
abbot	0,014	0,012	0,015	0,012	0,012	0,015	0,012
age	0,047	0,064	0,047	0,063	0,048	0,047	0,045
bovary	0,048	0,043	0,026	0,026	0,025	0,020	0,036
collapse	0,001	0,001	0,001	0,001	0,001	0,001	0,001
compress	0,003	0,003	0,003	0,004	0,003	0,003	0,003
corilis	0,113	0,107	0,108	0,111	0,097	0,097	0,114
cyprus	0,025	0,028	0,028	0,021	0,019	0,019	0,019
drkonqi	0,025	0,025	0,025	0,025	0,025	0,025	0,025
emission	0,281	0,258	0,276	0,255	0,253	0,270	0,272
firewrks	0,052	0,034	0,033	0,078	0,051	0,034	0,032
flower	0,059	0,040	0,034	0,030	0,024	0,024	0,063
gtkprint	0,017	0,017	0,017	0,019	0,017	0,017	0,017
handler	0,001	0,001	0,001	0,001	0,001	0,001	0,001
higrowth	0,009	0,009	0,009	0,009	0,015	0,010	0,010
hungary	0,076	0,063	0,052	0,051	0,036	0,028	0,028
libc06	0,007	0,007	0,007	0,007	0,006	0,007	0,008
lusiadas	0,161	0,159	0,157	0,162	0,195	0,193	0,190
lzlindmt	0,001	0,001	0,001	0,001	0,001	0,001	0,001
mailflder	0,007	0,006	0,007	0,006	0,007	0,006	0,007
mirror	0,024	0,024	0,024	0,025	0,024	0,025	0,027
modern	0,014	0,015	0,017	0,017	0,017	0,017	0,015
nightsht	0,134	0,091	0,067	0,063	0,104	0,132	0,155
render	0,001	0,001	0,001	0,001	0,001	0,001	0,001
thunder	0,204	0,182	0,172	0,321	0,178	0,264	0,250
ultima	0,018	0,018	0,018	0,019	0,019	0,017	0,020
usstate	0,001	0,001	0,001	0,001	0,001	0,001	0,001
venus	0,168	0,118	0,091	0,082	0,086	0,245	0,175
w01vett	0,204	0,239	0,290	0,187	0,266	0,218	0,216
wnvcrdt	0,107	0,119	0,108	0,113	0,106	0,119	0,120
xmlevent	0,001	0,001	0,001	0,001	0,001	0,001	0,001

Tabulka A.3: Doba trvání [s] výpočtu komprimace souborů z korpusu Prague Corpus na GPU. V značí dobu výpočtu, PP dobu trvání paměťových přesunů mezi hostem a OCL zařízením

Soubor	**Hešování**		**Hrubou silou**		**Z-algoritmus**	
	V	**PP**	**V**	**PP**	**V**	**PP**
abbot	0,349	0,015	6,587	0,004	7,817	0,004
age	6,166	0,015	6,706	0,004	4,363	0,005
bovary	0,827	0,015	6,334	0,004	7,795	0,005
collapse	0,006	0,014	0,218	0,004	0,276	0,004
compress	0,181	0,014	1,746	0,004	2,163	0,004
corilis	4,197	0,014	8,158	0,004	7,131	0,005
cyprus	1,107	0,015	2,713	0,004	3,284	0,006
drkonqi	3,763	0,015	3,810	0,004	4,204	0,005
emission	6,388	0,015	6,354	0,004	3,334	0,004
firewrks	1,627	0,014	8,464	0,004	9,889	0,004
flower	0,190	0,015	6,443	0,005	7,427	0,005
gtkprint	3,436	0,015	3,673	0,004	3,112	0,004
handler	0,180	0,014	1,160	0,004	1,330	0,004
higrowth	1,003	0,015	3,411	0,004	4,408	0,004
hungary	1,207	0,014	2,754	0,004	3,388	0,004
libc06	1,169	0,014	3,093	0,004	3,381	0,004
lusiadas	7,175	0,015	7,376	0,004	6,194	0,004
lzlindmt	0,089	0,015	1,029	0,004	1,277	0,004
mailfider	1,262	0,016	2,340	0,004	2,242	0,004
mirror	4,262	0,014	5,151	0,004	4,843	0,004
modern	0,325	0,015	3,272	0,004	3,970	0,004
nightsht	0,483	0,015	9,387	0,005	10,417	0,005
render	0,086	0,015	1,177	0,004	1,473	0,004
thunder	3,521	0,014	8,405	0,004	9,631	0,004
ultima	0,437	0,014	7,063	0,004	8,330	0,004
usstate	0,080	0,014	0,968	0,004	1,184	0,004
venus	1,006	0,015	8,735	0,005	9,855	0,005
w01vett	4,290	0,014	4,634	0,004	3,329	0,004
wnvcrdt	4,034	0,015	3,189	0,004	3,773	0,004
xmlevent	0,085	0,014	1,013	0,005	1,208	0,004

Tabulka A.4: Doba trvání [s] výpočtu komprimace souborů z korpusu Prague Corpus na CPU. V značí dobu výpočtu, PP dobu trvání paměťových přesunů mezi hostem a OCL zařízením

Soubor	**Hešování**		**Hrubou silou**		**Z-algoritmus**	
	V	**PP**	**V**	**PP**	**V**	**PP**
abbot	0,015	0,065	1,436	0,026	1,956	0,028
age	0,047	0,066	0,248	0,027	0,263	0,027
bovary	0,020	0,065	0,827	0,027	1,106	0,027
collapse	0,001	0,065	0,001	0,027	0,002	0,028
compress	0,003	0,065	0,117	0,027	0,150	0,027
corilis	0,097	0,066	1,405	0,026	1,874	0,026
cyprus	0,019	0,066	0,197	0,027	0,259	0,027
drkonqi	0,025	0,065	0,162	0,027	0,202	0,027
emission	0,270	0,064	0,547	0,027	0,454	0,027
firewrks	0,034	0,062	2,107	0,027	2,950	0,026
flower	0,024	0,057	4,267	0,020	6,103	0,020
gtkprint	0,017	0,066	0,058	0,027	0,057	0,028
handler	0,001	0,069	0,010	0,027	0,014	0,027
higrowth	0,010	0,064	0,221	0,027	0,277	0,027
hungary	0,028	0,064	0,286	0,026	0,374	0,026
libc06	0,007	0,067	0,070	0,029	0,088	0,027
lusiadas	0,193	0,068	0,536	0,027	0,670	0,027
lzlindmt	0,001	0,064	0,020	0,027	0,026	0,027
mailflder	0,006	0,064	0,047	0,027	0,057	0,027
mirror	0,025	0,065	0,148	0,028	0,180	0,027
modern	0,017	0,064	0,538	0,028	0,641	0,028
nightsht	0,132	0,048	8,940	0,011	12,947	0,011
render	0,001	0,064	0,013	0,027	0,018	0,027
thunder	0,264	0,070	2,199	0,025	3,025	0,025
ultima	0,017	0,063	1,391	0,026	1,999	0,026
usstate	0,001	0,064	0,006	0,027	0,008	0,028
venus	0,245	0,053	6,375	0,015	9,132	0,015
w01vett	0,218	0,068	0,403	0,027	0,341	0,027
wnvcrdt	0,119	0,067	0,285	0,027	0,236	0,027
xmlevent	0,001	0,064	0,006	0,027	0,008	0,027

Tabulka A.5: Doba trvání [s] výpočtu komprimace souborů z korpusu Prague Corpus jednotlivými implementovanými metodami na GPU a CPU + programem Gzip na CPU

Soubor	**Hešování**		**Hrubou silou**		**Z-algoritmus**		**Gzip**
	GPU	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	
abbot	0,364	0,080	6,592	1,463	7,821	1,983	0,043
age	6,181	0,113	6,710	0,275	4,368	0,290	0,038
bovary	0,841	0,085	6,338	0,854	7,800	1,133	0,179
collapse	0,020	0,066	0,222	0,028	0,280	0,029	0,024
compress	0,195	0,069	1,750	0,144	2,167	0,176	0,024
corilis	4,211	0,163	8,162	1,431	7,135	1,900	0,070
cyprus	1,122	0,085	2,716	0,224	3,289	0,286	0,026
drkonqi	3,777	0,090	3,814	0,189	4,209	0,228	0,027
emission	6,403	0,334	6,359	0,574	3,339	0,480	0,075
firewrks	1,641	0,096	8,468	2,134	9,893	2,976	0,081
flower	0,205	0,081	6,448	4,287	7,432	6,122	0,474
gtkprint	3,451	0,083	3,677	0,085	3,116	0,084	0,022
handler	0,195	0,070	1,164	0,037	1,334	0,041	0,024
higrowth	1,017	0,075	3,415	0,248	4,412	0,304	0,031
hungary	1,222	0,092	2,758	0,313	3,392	0,400	0,073
libc06	1,183	0,074	3,096	0,098	3,385	0,114	0,028
lusiadas	7,189	0,260	7,380	0,563	6,198	0,697	0,057
lzlindmt	0,104	0,065	1,033	0,048	1,281	0,053	0,025
mailflder	1,278	0,071	2,344	0,074	2,246	0,085	0,021
mirror	4,277	0,090	5,155	0,176	4,847	0,207	0,033
modern	0,340	0,081	3,277	0,567	3,974	0,669	0,063
nightsht	0,498	0,180	9,393	8,951	10,422	12,958	0,772
render	0,101	0,065	1,180	0,040	1,477	0,046	0,025
thunder	3,536	0,333	8,410	2,224	9,635	3,050	0,223
ultima	0,452	0,080	7,068	1,417	8,334	2,025	0,066
usstate	0,094	0,065	0,972	0,033	1,188	0,036	0,023
venus	1,021	0,298	8,740	6,390	9,861	9,148	1,045
w01vett	4,305	0,286	4,638	0,430	3,333	0,368	0,043
wnvcrdt	4,048	0,186	3,194	0,312	3,777	0,263	0,025
xmlevent	0,100	0,064	1,018	0,033	1,212	0,035	0,020

Tabulka A.6: Doba trvání [s] výpočtu dekomprimace souborů z korpusu Prague Corpus klasickým sekvenčním přístupem, paralelním přístupem + doba dekomprimace programu Gzip

Soubor	**Sekv.**	**Paralelní s různým PVPP**				**Gzip**
		1024	**2048**	**3072**	**4096**	
abbot	0.016	0,106	0,106	0,104	0,105	0,029
age	0.016	0,038	0,038	0,038	0,038	0,020
bovary	0.047	0,596	0,589	0,600	0,593	0,063
collapse	0.016	0,001	0,001	0,001	0,001	0,019
compress	0.016	0,029	0,029	0,030	0,029	0,029
corilis	0.031	0,356	0,353	0,352	0,351	0,030
cyprus	0.016	0,139	0,141	0,139	0,139	0,023
drkonqi	0.016	0,030	0,030	0,030	0,030	0,022
emission	0.032	0,630	0,641	0,621	0,642	0,034
firewrks	0.047	0,440	0,446	0,440	0,436	0,035
flower	0.250	2,022	2,804	3,060	3,117	0,135
gtkprint	0.016	0,010	0,010	0,010	0,010	0,019
handler	0.016	0,003	0,003	0,003	0,003	0,020
higrowth	0.016	0,035	0,035	0,036	0,035	0,019
hungary	0.031	0,933	0,915	0,909	0,937	0,035
libc06	0.000	0,013	0,013	0,013	0,013	0,028
lusiadas	0.016	0,179	0,176	0,177	0,176	0,024
lzlindmt	0.016	0,006	0,006	0,006	0,006	0,025
mailflder	0.016	0,011	0,011	0,012	0,011	0,021
mirror	0.016	0,025	0,025	0,025	0,025	0,025
modern	0.031	0,108	0,109	0,109	0,109	0,032
nightsht	0.422	1,824	2,771	3,899	4,552	0,181
render	0.016	0,004	0,004	0,004	0,004	0,024
thunder	0.078	0,942	0,951	0,923	0,943	0,067
ultima	0.031	0,308	0,306	0,314	0,308	0,045
usstate	0.015	0,002	0,002	0,002	0,002	0,021
venus	0.468	2,230	3,032	3,846	3,937	0,173
w01vett	0.015	0,365	0,375	0,367	0,374	0,026
wnvcrdt	0.016	0,087	0,085	0,088	0,087	0,020
xmlevent	0.015	0,002	0,002	0,002	0,002	0,022

Tabulka A.7: Velikosti souborů [B] z korpusu Prague Corpus před a po zkomprimování programem gzip a implementovaným LZ77 s využitím hešování a následným zakódováním aritmetickým kodérem, kompresní poměry

Soubor	Velikost	Gzip	GZ KP	Hešování	Heš. KP
abbot	349 055	317 083	1,101	342 987	1,018
age	137 216	60 944	2,252	62 850	2,183
bovary	2 202 291	735 382	2,995	964 552	2,283
collapse	2 871	1 117	2,570	1 576	1,822
compress	111 646	20 792	5,370	33 505	3,332
corilis	1 262 483	650 530	1,941	766 341	1,647
cyprus	555 986	23 721	23,439	83 710	6,642
drkonqi	111 056	37 877	2,932	46 338	2,397
emission	2 498 560	295 700	8,450	495 604	5,041
firewrks	1 440 054	1 316 120	1,094	1 411 114	1,021
flower	10 287 665	5 554 661	1,852	7 398 695	1,390
gtkprint	37 560	11 379	3,301	13 745	2,733
handler	11 873	2 831	4,194	4 121	2,881
higrowth	129 536	49 330	2,626	60 241	2,150
hungary	3 705 107	152 096	24,360	593 238	6,246
libc06	48 120	16 200	2,970	19 958	2,411
lusiadas	625 664	185 297	3,377	237 127	2,639
lzlindmt	22 922	4 949	4,632	7 288	3,145
mailflder	43 732	9 640	4,537	13 862	3,155
mirror	90 968	35 331	2,575	43 010	2,115
modern	388 909	153 256	2,538	198 357	1,961
nightsht	14 751 763	13 439 836	1,098	14 447 255	1,021
render	15 984	3 840	4,163	5 586	2,861
thunder	3 172 048	2 430 408	1,305	2 488 066	1,275
ultima	1 073 079	701 789	1,529	815 444	1,316
usstate	8 251	2 048	4,029	2 878	2,867
venus	13 432	10 055 771	0,001	11 212 025	0,001
w01vett	1 381 141	73 032	18,911	116 423	11,863
wnvcrdt	328 550	17 816	18,441	27 758	11,836
xmlevent	7 542	2 117	3,563	3 049	2,474

Tabulka A.8: Velikosti souborů [B] z korpusu Calgary před a po zkomprimování programem gzip a implementovaným LZ77 s využitím hešování a následným zakódováním aritmetickým kodérem, kompresní poměry

Soubor	Velikost	Gzip	GZ KP	Hešování	Heš. KP
bib	111 261	35 063	3,173	51 739	2,150
book1	768 771	313 376	2,453	400 904	1,918
book2	610 856	206 687	2,955	279 878	2,183
geo	102 400	68 493	1,495	69 009	1,484
news	377 109	144 840	2,604	187 035	2,016
obj1	21 504	10 323	2,083	12 646	1,700
obj2	246 814	81 631	3,024	102 426	2,410
paper1	53 161	18 577	2,862	24 692	2,153
paper2	82 199	29 753	2,763	39 169	2,099
paper3	46 526	18 097	2,571	23 255	2,001
paper4	13 286	5 536	2,400	6 953	1,911
paper5	11 954	4 995	2,393	6 283	1,903
paper6	38 105	13 232	2,880	17 862	2,133
pic	513 216	56 442	9,093	66 358	7,734
progc	39 611	13 275	2,984	18 168	2,180
progl	71 646	16 273	4,403	23 178	3,091
progp	49 379	11 246	4,391	16 216	3,045
trans	93 695	18 985	4,935	34 128	2,745

Tabulka A.9: Velikosti souborů [B] z korpusu Canterbury před a po zkomprimování programem gzip a implementovaným LZ77 s využitím hešování a následným zakódováním aritmetickým kodérem, kompresní poměry

Soubor	Velikost	Gzip	GZ KP	Hešování	Heš. KP
alice29.txt	152 089	54 435	2,794	72 089	2,110
asyoulik.txt	125 179	48 951	2,557	62 887	1,991
cp.html	24 603	7 999	3,076	11 004	2,236
fields.c	11 150	3 143	3,548	4 286	2,601
grammar.lsp	3 721	1 246	2,986	1 789	2,080
kennedy.xls	1 029 744	206 779	4,980	138 225	7,450
lcet10.txt	426 754	144 885	2,945	196 970	2,167
plrabn12.txt	481 861	195 208	2,468	247 246	1,949
ptt5	513 216	56 443	9,093	66 358	7,734
sum	38 240	12 924	2,959	16 917	2,260
xargs.1	4 227	1 756	2,407	2 331	1,813

Dodatek B

Seznam použitých zkratek

ANSV All nearest smaller values – všechny nejbližší menší hodnoty

CPU Central processing unit – centrální procesorová jednotka

DSIZ Dictionary size – velikost doplňující hešovací tabulky

GPU Graphics processing unit – grafický procesor

HSIZ Hash table size – velikost hešovací tabulky

ISA Inverse suffix array – inverzní příponové pole

LPF Longest previous factor – nejdelší predchozí shoda

LCP Longest common prefix – nejdelší společná předpona

NSV Next smaller value – následující menší hodnota

PSV Previous smaller value – předchozí menší hodnota

PVPP Počet vytvářených pracovních položek

SA Suffix array – příponové pole

Dodatek C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF