VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE KOMPRESE DAT V PROSTŘEDÍ PARALELNÍCH ARCHITEKTUR

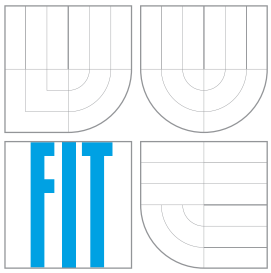
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUBOŠ JURÁNEK

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE KOMPRESE DAT V PROSTŘEDÍ PARALELNÍCH ARCHITEKTUR

ACCELERATION OF DATA COMPRESSION WITH PARALLEL ARCHITECTURES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUBOŠ JURÁNEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. VÁCLAV ŠIMEK
SUPERVISOR

BRNO 2014

Abstrakt

Tato bakalářská práce se zabývá využitím paralelních architektur, zejména GPU, pro akceleraci vybraných bezeztrátových komprimačních algoritmů, založených na statistické metodě, a transformací, měnící entropii vstupních dat pro dosažení lepšího kompresního poměru. V této bakalářské práci jsou také teoreticky shrnuty obecné informace o paralelních architekturách a možností programování na nich, hlavně pomocí technologií NVIDIA CUDA a OpenCL.

Abstract

This bachelor thesis deals with the use of parallel architectures, in particular the GPU, for acceleration of selected lossless compression algorithms, based on a statistical method, and transformations, which change the entropy of the input data to achieve better compression ratio. In this work there are in theory summarized general information about parallel architectures and programming options for them, mainly using NVIDIA CUDA and OpenCL.

Klíčová slova

GPGPU, CUDA, OpenCL, bezeztrátová komprese, Burrowsova-Wheelerova transformace.

Keywords

GPGPU, CUDA, OpenCL, lossless compression, Burrows–Wheeler transform.

Citace

Luboš Juránek: Akcelerace komprese dat v prostředí
paralelních architektur, bakalářská práce, Brno, FIT VUT v Brně, 2014

Akcelerace komprese dat v prostředí paralelních architektur

Prohlášení

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

.....
Luboš Juránek
18. května 2014

Poděkování

Zde bych rád poděkoval svému vedoucímu, Ing. Václavu Šimkovi za odbornou pomoc a vedení mé bakalářské práce.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Po více než dvacet let společnosti Intel a AMD zvyšovali výkon a snižovali cenu svých procesorů. Každá generace procesorů nabízela mnohem vyšší rychlost běhu softwaru, aniž by jej vývojáři museli měnit. Od roku 2003 se však situace změnila. Vlivem přílišných energetických nároků a problémům s odváděním tepla se ukázalo, že další zvyšování taktovací frekvence a počtu instrukcí zpracovaných v jednom taktu, nebude nadále možné.

Výrobci přešli na výrobu procesorů s více výpočetními jednotkami, jádry procesoru. Všechny dosavadní sekvenční programy tak přestaly využívat plnou výpočetní sílu nových procesorů. Pro softwarové vývojáře tak vznikla zajímavá možnost věnovat se paralelnímu programování, které do té doby nebylo příliš rozšířené. Paralelním architekturám, na kterých je toto programování možné, se věnuje kapitola 1.

Zajímavou možností, jak zvýšit výpočetní rychlost pomocí paralelního programování, bylo nejen využití více jader procesorů, ale také akcelerace za pomoci grafické karty. Běžná grafická karta ve stolním počítači se postupným vývojem, zejména díky hernímu průmyslu, dostala na výkon několikanásobně překračující běžné procesory. Jak využít grafickou kartu a další paralelní architektury k akceleraci algoritmů se věnuje kapitola 2.

Hlavním tématem této práce je akcelerace komprese dat v prostředí paralelních architektur. Metod komprese dat, tedy zmenšení jejich datového objemu, je v dnešní době nepřeberné množství určené pro nejrůznější typy dat od textu přes hudbu až například po video. Tato práce se věnuje zejména bezeztrátovým komprimačním metodám založeným na principu statistických kompresních metod. Více o kompresních algoritmech postavených na těchto metodách, ale i obecné informace o dalších, je obsaženo v kapitole 3.

Cílem, který si tato práce klade, je implementace paralelních variant vybraných komprimačních algoritmů a pomocných transformací, které se pro tento účel používají. Tyto paralelní varianty by měly dosahovat rychlejších výpočtů, než jejich sekvenční protějšky. Tato akcelerace konkrétních kompresních algoritmů na výpočetním procesoru grafické karty, využívající teoretické znalosti z předchozích kapitol, je popsána, analyzována a zhodnocena v poslední kapitole 4.

Kapitola 1

Paralelní architektury

1.1 Rozdělení

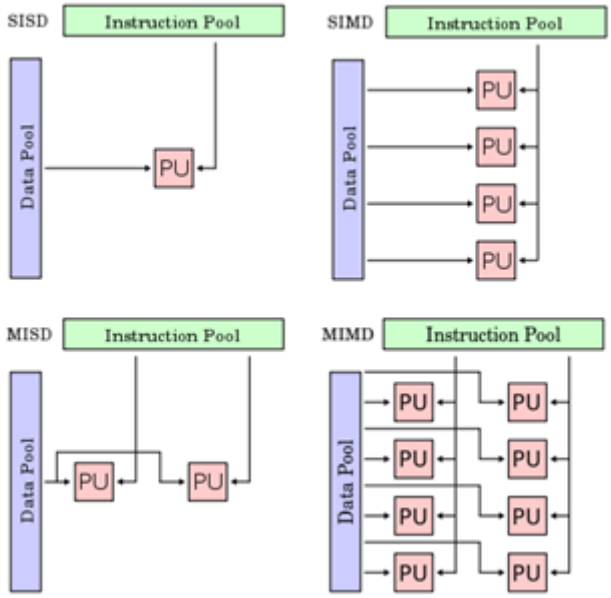
Rozdělit paralelní architektury můžeme hned několika způsoby. Pravděpodobně nejznámější je rozdělení dle Flynnovy klasifikace, která rozděluje jednotlivé architektury dle toku instrukcí a toku dat [4].

	Single instruction	**Multiple instructions**
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Tabulka 1.1: Flynnova klasifikace

- Single Instruction, Single Data (SISD) - jeden jednoduchý datový typ na jednu instrukci; klasický jednoprocesorový počítač s Von-Neumannovskou architekturou. I zde je možné zpracovávat více instrukcí a dat současně, například pomocí zřetězeného zpracování či superskalární architektury, ale jedná se pouze o paralelní zpracování sekvenčního kódu a nikoliv provedení paralelního kódu [14].
- Multiple Instruction, Single Data (MISD) - více instrukcemi prochází jedny data; málo komerčních paralelních systémů je založeno na principech MISD, ale příkladem můžou být systolické řady, což je síť jednoduchých výpočetních jednotek, skrz které prochází data [5].
- Single Instruction, Multiple Data (SIMD) - více dat na jednu instrukci; stejná instrukce je provedena více procesory, každý se svým vlastním datovým tokem. Každý procesor má tedy vlastní datovou paměť, ale je zde pouze jedna instrukční paměť a řídící procesor. Zde jsou příkladem zřetězené vektorové procesory, procesorové řady s jednou řídící a několika výpočetními jednotkami, a vzhledem k zadání této práce hlavně grafické procesory (GPU).
- Multiple Instruction, Multiple Data (MIMD) - různé instrukce nad různými daty; každý procesor nahrává vlastní instukce a operuje s vlastními daty. Dnes většina paralelních architektur včetně multiprocesorů, multipočítačů a vícejádrových procesorů

(symetrické multiprocesory (SMP), kde každé jádro je považováno za samostatný procesor [8]) spadá do této skupiny.



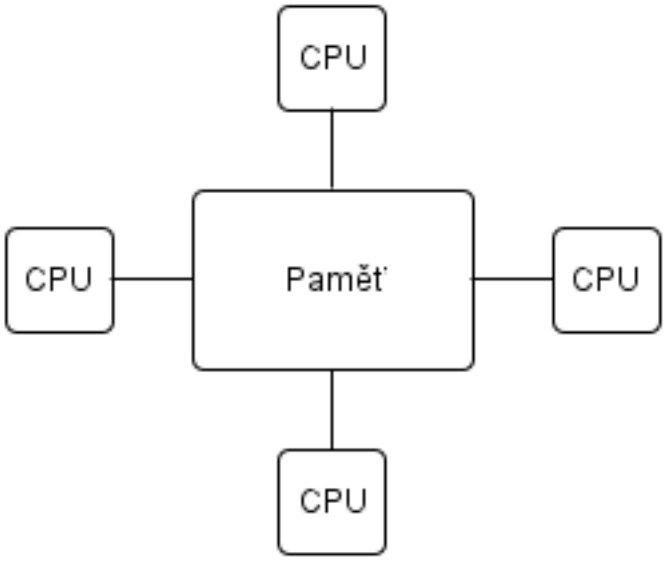
Obrázek 1.1: Flynnova klasifikace (Colin M.L. Burnett, 2007)

Přestože je Flynnova klasifikace poměrně zastaralá a nepřesná, díky své jednoduchosti se používá dodnes jako první náhled na problematiku a rozdělení paralelních architektur. Mnoho dnešních počítačů jsou hybridy těchto kategorií a někdy se přidávají ještě další kategorie jako:

- Multiple SIMD (MSIMD) - více nezávislých podsystémů SIMD v jednom systému.
- Single Program, Multiple Data (SPMD) - způsob paralelního zpracování, kdy běží stejný program na více procesorech s jinými vstupy, a tak jsou získávány různé výstupy rychleji [2, 3].
- Multiple Program, Multiple Data (MPMD) - více autonomních procesorů souběžně provádí více nezávislých programů. Příkladem je speciální vícejádrový procesor Cell využívaný v herní konzoli PlayStation 3 [7].

Paralelní architektury můžeme také rozdělit dle architektury paměti [1].

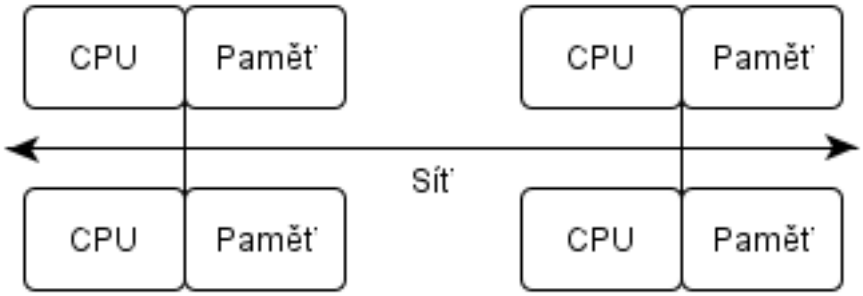
- Architektury se sdílenou pamětí - taková architektura, kde počítač má paměť přístupnou většímu množství procesorů. Dva základní typy architektur se sdílenou pamětí jsou Uniform Memory Access (UMA) a Non-Uniform Memory Access (NUMA) rozdělené podle stejné či rozdílné přístupové době k sdílené paměti jednotlivých procesorů. Nejběžnějším příkladem UMA jsou symetrické multiprocesory (SMP), mezi které se dají řadit také dnešní vícejádrové procesory. Propojením takových multiprocesorů (například vícejádrové procesory ve víceprocesorovém socketu) pak vznikne nejběžnější příklad NUMA.



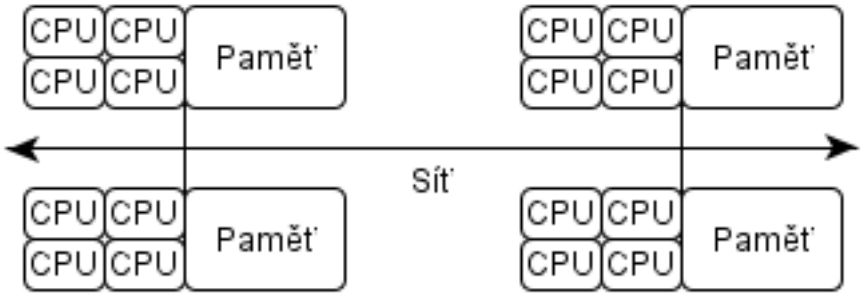
Obrázek 1.2: Sdílená paměť - Uniform Memory Access (UMA)

Obrázek 1.3: Sdílená paměť - Non-Uniform Memory Access (NUMA)

- Architektury s distribuovanou pamětí - taková paralelní architektura, kde počítač má větší množství procesorů, každý z nich má vlastní oddělený paměťový prostor, a všechny tyto paměťové prostory propojuje propojovací síť. Propojovací síť musí zajišťovat komunikaci pomocí zasílání zpráv. Příkladem takové architektury jsou počítačové clustery, kde komunikaci po síti může zajišťovat knihovna Message Passing Interface (MPI), jejíž nejznámější implementací je OpenMPI.
- Hybridní architektury s distribuovanou sdílenou pamětí - takové paralelní architektury, kde systém obsahuje více počítačů s distribuovanou pamětí propojené skrz síť, kde každý počítač má paralelní architekturu se sdílenou pamětí (jako například symetrický multiprocesor). Příkladem může být OpenMP (Open Multi-Processing), což je API pro multiprocesní programování se sdílenou pamětí, které propojíme do clusteru pomocí Message Passing Interface (MPI).



Obrázek 1.4: Distribuovaná paměť



Obrázek 1.5: Hybridní architektura

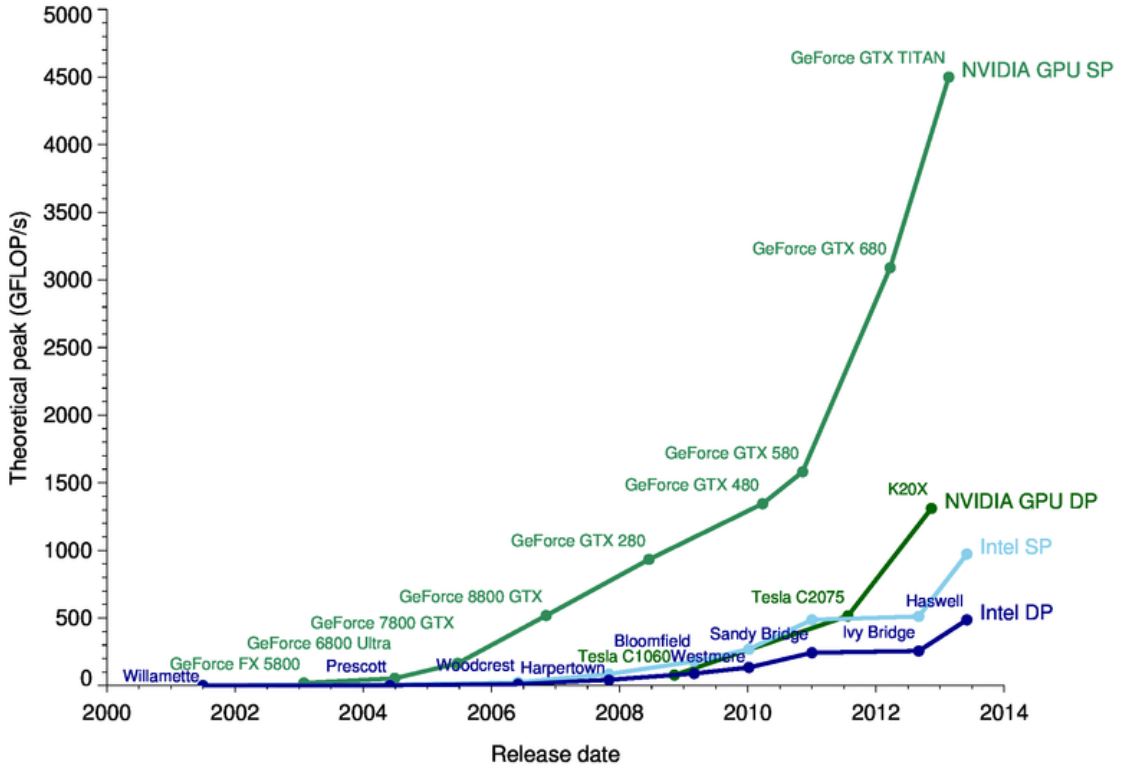
1.2 Srovnání GPU a CPU

Grafický procesor (GPU) je specializovaný řídící procesor umístěný na grafické kartě počítače. Jeho účelem je zajišťovat vykreslování dat na zobrazovacím zařízení. Protože se GPU chová při výpočtech jako paralelní procesor, je ho možné využít i k obecným výpočtům, o které se běžně stará procesor CPU.

Tomuto využití se říká General-Purpose Computing on Graphics Processing Units, zkráceně GPGPU [10]. Takové využití může být velmi výhodné z hlediska výkonu, protože GPU mají dnes obecně mnohem vyšší výkon v počtu operací v plovoucí řádové čárce za sekundu oproti běžným CPU.

Tento výkon je zajištěn mnohem vyšším počtem výpočetních jednotek, neboli jader, a celkovém součtu tranzistorů než má CPU, přestože jednotlivá jádra běží na nižších frekvencích. Můžeme tedy tvrdit, že zatímco klasický procesor CPU je optimalizován na co nejvyšší rychlost operace, tak pokud plně využijeme potenciál paralelního zpracování na GPU, bude tento teoretický výkon GPU několikanásobně vyšší než výkon CPU, a tento rozdíl stále roste, jak znázorňuje graf.

Tento graf srovnání pochází od společnosti NVIDIA, a proto srovnává pouze vlastní GPU s procesory od společnosti Intel. Dá se však předpokládat, že srovnání výrobků AMD by dopadlo velice podobně. Graf je rozdělen na SP (Single-precision floating-point format) a DP (Double-precision floating-point format) podle přesnosti čísel v plovoucí řádové čárce,



Obrázek 1.6: Srovnání teoretického výkonu CPU a GPU (převzato z [13])

s kterými je schopen procesor pracovat.

Nebylo by však efektivní CPU zcela nahradit. Přestože CPU mají mnohem menší počet jader, dnešními technologickými postupy již není dále možné zvyšovat jejich taktovací frekvence a mají menší paměťovou propustnost k RAM než GPU ke své DRAM, tak jsou CPU mnohem univerzálnější a jsou optimalizovány pro vykonávání sekvenčního kódu a velmi rychlému běhu jednoho vlákna (superskalarita, predikce skoků, využívání velké cache paměti, atd.)

GPU se tedy využívá tam, kde se dá výpočet paralelizovat a zatímco CPU provádí sekvenční části kódu, GPU pomocí svých SIMD jednotek provádí instrukce nad velkým množství stejných dat zároveň. Tomuto zrychlení a rozdělení výpočtu se říká akcelerace výpočtu pomocí GPU.

Pro srovnání a představu o rozdílech mezi GPU a CPU může sloužit i následující tabulka, která ukazuje údaje, uváděné výrobci, dvou moderních výpočetních jednotek. Pro ukázku byly vybrány současné nejnovější a nejvýkonnější modely určené pro běžné stolní počítače, šestijádrový procesor Intel Core i7 4960X a herní grafická karta GTX TITAN Black.

Přestože o masivní nárůst výkonu grafických akcelerátorů se minulosti postaral zejména herní průmysl, díky GPGPU dnes vznikají grafické karty, které jsou přímo určeny pro obecné výpočty. Takovými grafickými kartami je například řada NVIDIA Tesla, která již není optimalizována pro zpracování grafiky, ale díky své tzv. unifikované architektuře nabízí několik set programovatelných výpočetních jednotek s možností výpočtů ve dvojité přesnosti (Double-precision floating-point format). Grafické karty NVIDIA Tesla jsou dnes

Procesor	Intel Core i7 4960X	NVIDIA GTX TITAN Black
Počet tranzistorů	1860 milionů	7080 milionů
Taktovací frekvence	3600 MHz	980 MHz
Počet jader	6	2880
Výkonnost	156,5 GFLOPS	5000 GFLOPS
Datová propustnost	59,7 GBps	336,4 GBps

Tabulka 1.2: Srovnání CPU a GPU

v některých z nejrychlejších superpočítačů světa[1].

1.3 Alternativy

Podle vývoje dnešních superpočítačů si můžeme udělat obrázek o současných možnostech ve využití paralelních architektur. Výpočetní akcelerátory se stále více stávají součástí nejvýkonnějších světových superpočítačů. Procesory Cell, poprvé představeny v herní konzoli PlayStation 3 jsou vytlačovány GPU akcelerátory NVIDIA Tesla.

I zde však mají GPU akcelerátory velkou konkurenci v podobě koprocesorů Xeon Phi od Intelu, které jsou založeny na multiprocesorové architektuře Many Integrated Core (MIC). Tyto koprocesory umožňují využívat kolem šedesáti velmi výkonných malých jader založených na běžné procesorové instrukční sadě x86. Nabízí tak podobné využití jako GPU akcelerátory v GPGPU, se srovnatelným výkonem, ale se snadnějším programováním bez nutnosti přizpůsobovat se architektuře GPU. Prozatím jsou tyto koprocesory v podobě PCIe karet stejně jako grafické karty, ale Intel uvádí, že chystá i mnohojádrové Xeon Phi procesory do klasických procesorových socketů[2].

Další zajímavou inovací na poli paralelních architektur je AMD Accelerating Processing Unit (APU), dříve pojmenovaná jako AMD Fusion. Jde o heterogenní systémovou architekturu (HSA) označovanou jako heterogenní UMA (hUMA), dle rozdělení paralelních architektur podle architektury paměti, která spojuje CPU a GPU na stejném čipu, a tedy obě výpočetní jednotky sdílí stejnou paměť. To by mělo usnadnit vývoj softwaru, který využívá výhody CPU i GPU pro výpočty. Tato architektura je zatím určena do méně výkonných počítačů a nově také do nové generace herních konzolí, jako je například PlayStation 4 či Xbox One, kde spoléhá na efektivní spolupráci CPU a GPU[3].

[1] NVIDIA: High Performance Computing [online]. 2014 [cit. 2014-03-23]. URL http://www.nvidia.com/object/tesla-supercomputing-solutions.html

[2] Intel: Xeon Phi Coprocessor [online]. 2014 [cit. 2014-02-11]. URL https://software.intel.com/en-us/mic-developer

[3] AMD: Heterogeneous Computing [online]. 2014 [cit. 2014-02-12]. URL http://developer.amd.com/resources/heterogeneous-computing

Kapitola 2

Paralelní programování

2.1 GPGPU

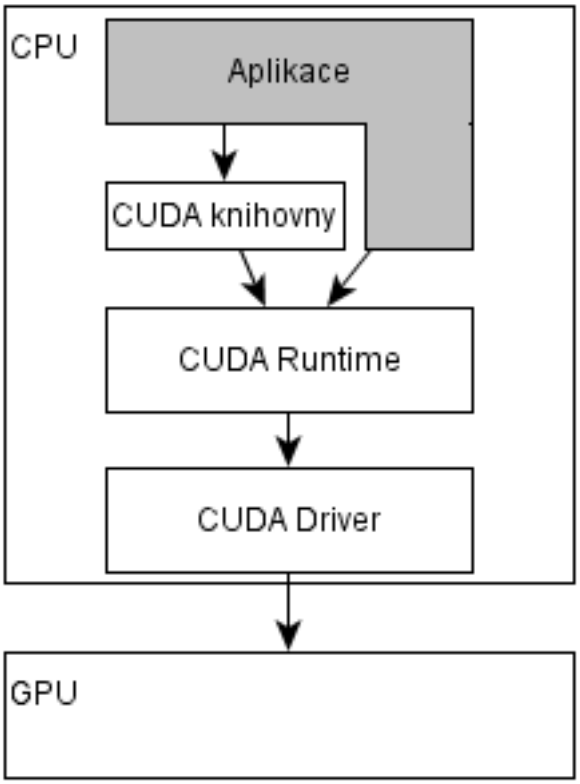
Využití výkonu GPU na jiné účely než je zpracovávání 2D či 3D grafiky se stalo velmi žádané mezi vývojáři softwaru, není tak divu, že vzniklo mnoho knihoven, API a frameworků, které GPGPU umožňují. Mezi nimi například DirectCompute a C++ AMP(Accelerated Massive Parallelism) od Microsoftu, ATI Stream od AMD/ATI, OpenCL (Open Computing Language) od Khronos Group konsorcia a CUDA (Compute Unified Device Architecture), kterou vyvíjí společnost NVIDIA. Poslední dvě zmiňovaná řešení GPGPU se v současnosti využívají nejvíce, a proto si podrobněji popíšeme jejich principy.

2.1.1 NVIDIA CUDA

Byla představena v roce 2006 jako jedna z prvních technologií určených k obecným paralelním výpočtům na GPU. CUDA je hardwarovou i softwarovou architekturou, která je dostupná na operačních systémech Windows, Linux a Mac OS X, a softwaroví vývojáři můžou aplikace psát v jazycích C/C++ nebo Fortran. CUDA ale podporuje také využití dalších GPGPU výpočetních technologií včetně OpenCL, DirectCompute a C++ AMP. Pomocí knihoven je však možné technologii CUDA využívat i v dalších programovacích jazycích jako je například Python (PyCUDA[1]), Perl (KappaCUDA[2]) a Java (jCUDA[3]). Aplikace napsané pomocí technologie CUDA můžou fungovat pouze na NVIDIA grafických akcelerátorech, a to novějších než je GeForce řady 8 včetně, což je největší nevýhoda této architektury. CUDA SDK do verze 3 umožňovala emulaci GPU kódu na CPU, ale v současné době je jedinou možností využít neoficiální emulátor CUDA Waste[4] pro Windows nebo dynamický kompilátor Ocelot[5] pro Linux, umožňující spouštět CUDA programy na NVIDIA GPU, AMD GPU i x86-CPU bez rekompilace.

CUDA umožňuje programovat aplikace pomocí dvou API, vysokoúrovňové CUDA Runtime API a nízkoúrovňové CUDA Driver API, které nemohou být kombinovány. Program musí využívat jednu či druhou API. CUDA Runtime API představuje celý programovací model jako malou sadu rozšíření použitého programovacího jazyka, která zajišťuje veškeré operace na GPU, a runtime knihovnu. Od běžného programování na CPU se tak liší jen

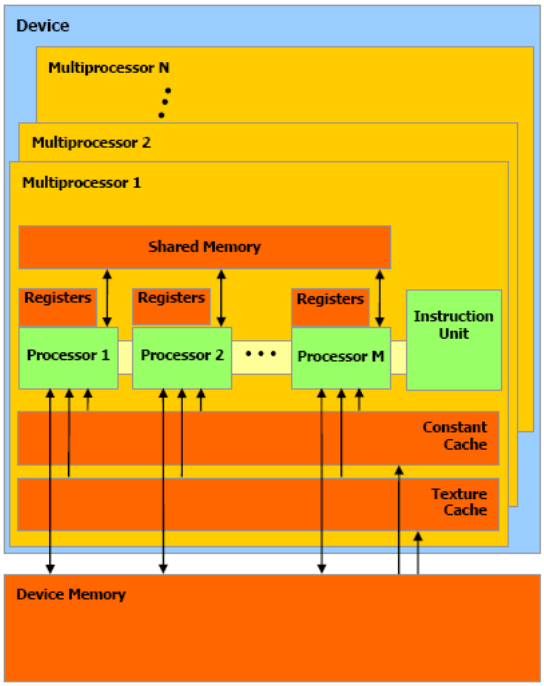
[1] http://mathema.tician.de/software/pycuda
[2] http://psilambda.com/download/kappa-for-perl
[3] http://www.jcuda.org
[4] https://code.google.com/p/cuda-waste
[5] https://code.google.com/p/gpuocelot



Obrázek 2.1: Úrovně CUDA aplikace

trochou nové syntaxe jazyka. Tato vysokoúrovňová Runtime API pracuje nad nízkoúrovňovou, kód je přeložen do jednoduchých instrukcí, které zpracovává Driver API. Pokud se programátor rozhodne pracovat s CUDA Driver API, má přístup přímo k těmto jednoduchým instrukcím, CUDA binárnímu či asemblerovému kódu, který může modifikovat. Takové programování a debugování je náročnější, ale nabízí vyšší úroveň kontroly a je nezávislé na programovacím jazyku. Na obrázku je ukázána ještě další, vyšší úroveň, CUDA knihovny, které obsahují již vytvořené funkce, optimalizované pro GPGPU. Jako příklad můžeme uvést cuBLAS (Basic Linear Algebra Subprograms) pro úlohy lineární algebry, například maticové násobení, a cuFFT (Fast Fourier Transform), která se využívá zejména při zpracování signálů. Přestože CUDA nabízí dvě úrovně API, postrádá vysokou úroveň abstrakce, a tak i při využití Driver API jsou vyžadovány znalosti o NVIDIA GPU.

NVIDIA GPU jsou tvořeny zejména velkým množstvím (stovky až tisíce) skalárních procesorů, které jsou propojeny do škálovatelných řad streaming multiprocesorů (SM, nová generace používá zkratku SMX), což je velký rozdíl oproti konkurenčním AMD grafickým kartám, které mají multiprocesory postavené z výpočetních jednotek typu VLIW (very long instruction word). Různé typy grafických karet mají různý počet SM, ale neplatí, že čím výkonnější grafická karta, tím je tento počet vyšší. Multiprocesory jsou však stále výkonnější a každý obsahuje stále více jader (skalárních procesorů). Například současná herní grafická karta GTX Titan Black obsahuje 15 SMX, kde každý má 192 stream procesorů. Každý takový multiprocesor provádí instrukce na principu nové architektury SIMT (sin-



Obrázek 2.2: Architektura NVIDIA GPU (převzato z [13])

gle instruction, multiple thread), která vychází z principu SIMD. Multiprocesor vytvoří skupinu 32 vláken nazvanou warp, vlákna přiřadí ke svým jednotlivým jádrům a každou instrukci provede paralelně na každém vláknu ve warpu. Jak je vidět na obrázku, každý multiprocesor má přímo na čipu svoji vlastní paměť, která se dělí na čtyři následující typy:

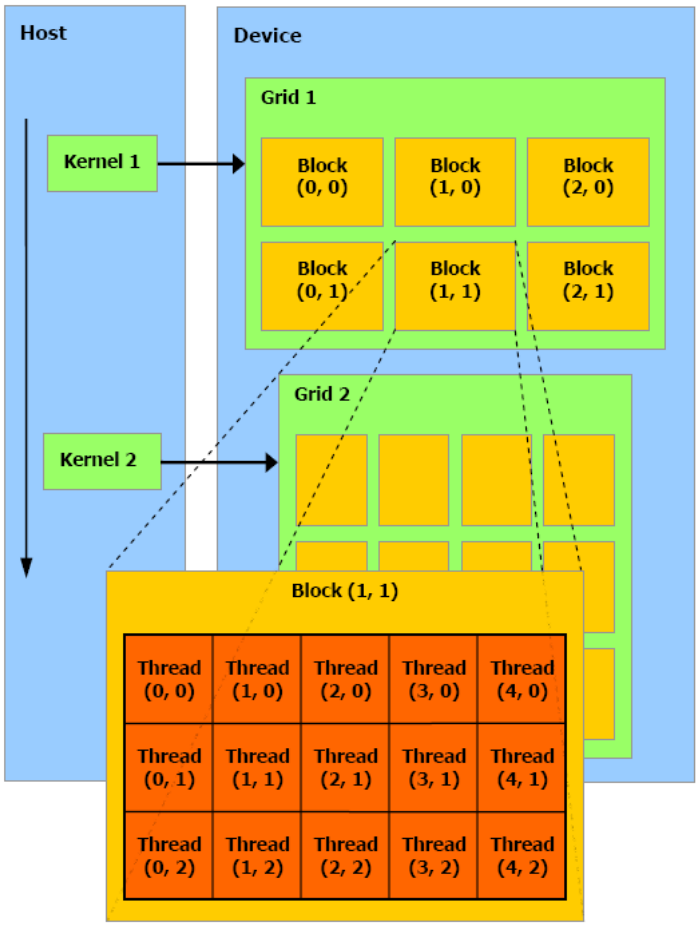
- Registry - každý skalární procesor v multiprocesoru má vlastní sadu lokálních 32-bitových paměťových registrů. Tato paměť je nejrychlejší na čipu a mezi jednotlivé procesory je rozděluje překladač. Každé vlákno přidělené jádru může přistupovat pouze k vlastním registrům a v případě, že tyto registry jsou zaplněny, dá se využít lokální paměť. Ta se může používat stejně jako pole registrů, je však fyzicky umístěna v globální paměti grafického akcelerátoru, tedy mimo čip, a tak je mnohem pomalejší.

- Sdílená paměť - všechny jednotlivé skalární procesorové jádra mají přístup do velmi rychlé sdílené paměti (shared memory), kterou má každý multiprocesor na svém čipu. Tato paměť umožňuje vyměňovat informace mezi jednotlivými vlákny na jednom multiprocesoru. Není však možné vyměňovat data mezi vlákny běžícími na jiných multiprocesorech.

- Kešovaná paměť konstant - paměť konstant (constant memory) je sice fyzicky mimo čip, je sdílená mezi všemi multiprocesory a pomalá. Je pro ni však vyhrazeno na čipu multiprocesoru L1 cache. Každý multiprocesor má tak kešovanou paměť pro konstanty (constant cache), která je sdílená mezi všemi jádry multiprocesoru a je pouze pro čtení. Protože je přímo na čipu, je rychlejší než globální paměť a dá se tak použít ke zrychlení načítání konstant, které se nemění po celou dobu výpočtu.

- Kešovaná paměť textur - paměť textur (texture memory) je také fyzicky mimo čip a funguje na stejném principu jako kešovaná paměť konstant. Stejně tak kešovaná paměť textur (texture cache) je pouze pro čtení, je sdílena mezi všemi skalárními procesory a zrychluje čtení z paměti textur, která je součástí paměti grafického akcelerátoru. Každý multiprocesor přistupuje ke kešované paměti konstant přes texturovou jednotku, která umožňuje specifickou práci s daty výhodnou při zpracovávání grafiky. Pro GPGPU je využívání paměti textur výhodné jen při specifických výpočtech.

V paměti na grafickém akcelerátoru, mimo čip, je navíc i globální paměť, která je sice pomalá, ale je největší ze všech a je přístupná všem multiprocesorům. Další její výhodou je největší paměťová propustnost. Také do ní má přístup CPU stejně jako do paměti konstant a paměti textur.



Obrázek 2.3: Organizace vláken, bloků a mřížek (převzato z [13])

Samotné programování CUDA aplikace probíhá tak, že se celá aplikace rozdělí na části, které běží na CPU (hostitel) a na ty, které běží na GPU (CUDA zařízení). CPU pak zavolá speciální funkci kernel, kterou provede každé vlákno přiřazené k GPU. Tato vlákna jsou seskupena do 1D, 2D nebo 3D bloků vláken, které jsou přiřazeny jednotlivým streaming multiprocesorům. Multiprocesor pak rozdělí jednotlivá vlákna bloku svým stream

procesorům, a tak mohou všechny vlákna jednoho bloku sdílet data přes sdílenou paměť. Všechny bloky jsou dále seskupeny do 1D, 2D nebo 3D mřížky, která provádí daný kernel. Ten určuje počet a organizaci bloků v mřížce, které jsou paralelně zpracovávány nezávisle na ostatních, a také určuje počet a organizaci vláken v jednom bloku, které jsou spouštěny paralelně po warpech na jednom multiprocesoru a je možné jejich běh synchronizovat. Ve spuštěném kernelu je také možné jednoznačně identifikovat každý blok v mřížce i každé vlákno v bloku pomocí unikátních indexů. Samotný výpočet na GPU probíhá tak, že grafická karta funguje na principu koprocesoru, který akceleruje přesně určené paralelní části aplikace, zatímco CPU provádí zbytek sekvenčně. Před samotným výpočtem je nutno vyhradit potřebné místo v paměti grafického akcelerátoru a přesunout tam potřebná data z paměti RAM. Poté se provede výpočet na GPU a výsledky se přesunou zpátky z paměti grafické karty do paměti RAM, odkud s nimi dále může pracovat CPU.

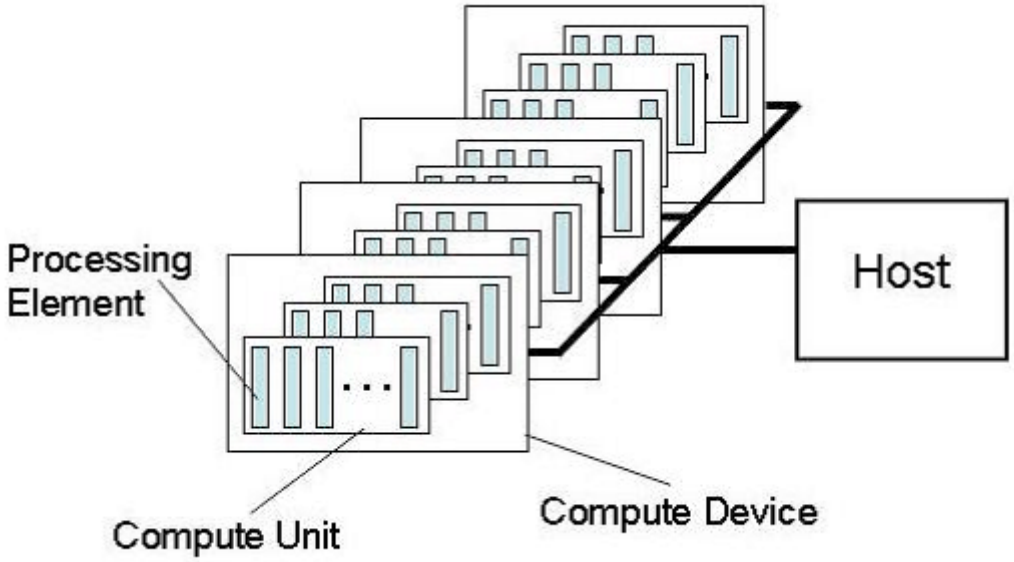
V současné době už CUDA umožňuje tzv. dynamický paralelismus, který dovoluje volat nové funkce kernel přímo z GPU. CPU tak může zavolat pouze první kernel a během složitého výpočtu nemusí GPU vracet mezivýsledky CPU, aby mohl zavolat další kernel. Místo toho volá další kernely GPU a vrátí až konečný výsledek. CUDA také nově virtuálně spojuje paměť grafického akcelerátoru a RAM z pohledu programátora a veškeré přesuny dat před a po výpočtu na GPU se provádí automaticky na pozadí.

2.1.2 OpenCL

Open Computing Language je framework vytvořen v roce 2008 pro paralelní programování na heterogenních systémech, jako je například počítač s CPU a GPU. S OpenCL je však možné programovat paralelně i na homogenních systémech, například vícejádrových procesorech, kde jedno z jader je využito jako řídící a další jádra jako výpočetní. OpenCL je vyvíjeno konsorciem Khronos, což je skupina zahrnující mnoho firem, z nichž mezi nejznámější patří AMD, NVIDIA, Intel, Apple a IBM. V současné době je OpenCL průmyslovým standardem pro paralelní programování na heterogenních počítačových systémech, je multiplatformní a umožňuje programovat aplikace pro vícejádrové CPU (x86 s podporou instrukcí SSE3), GPU (NVIDIA GeForce řady 8 a vyšší, ATI Radeon HD řady 4000 a vyšší), FPGA, DSP (digital signal processor), Cell procesory a dalších paralelních architekturách. Framework obsahuje jazyk OpenCL C, pomocí kterého se paralelní aplikace programují, jeho kompilátor a runtime knihovnu potřebnou k běhu aplikace napsané v tomto jazyce. OpenCL C je založen na základu jazyka C, přesněji jeho specifikace C99, a obsahuje navíc další rozšíření pro paralelní programování. Paralelní aplikace naprogramovaná za pomoci OpenCL se dělí na dvě části, hostitelská část, kterou je sekvenční kód v programovacím jazyce C/C++ běžící většinou na CPU, a část pro OpenCL zařízení (GPU pro GPGPU), která se nazývá kernel a je programována právě pomocí jazyka OpenCL C. V dnešní době již existuje možnost programovat s OpenCL prostřednictvím knihoven v dalších programovacích jazycích jako jsou Python(PyOpenCL[6]), Java(JOCL[7]) nebo .NET(OpenCL.NET[8]). Princip fungování OpenCL se dá zjednodušeně shrnout do čtyř modelů, platformního, exekučního, paměťového a programovacího modelu [6].

Platformní model specifikuje, že v systému musí být jeden hostitel, což je řídící procesor (většinou CPU) koordinující celý výpočet, a jedno nebo více OpenCL výpočetních zařízení (compute device), neboli procesorů, které vykonávají výpočet naprogramovaný v OpenCL

[6] http://mathema.tician.de/software/pyopencl
[7] http://www.jocl.org
[8] http://openclnet.codeplex.com



Obrázek 2.4: (Platformní model OpenCL (převzato z [9])

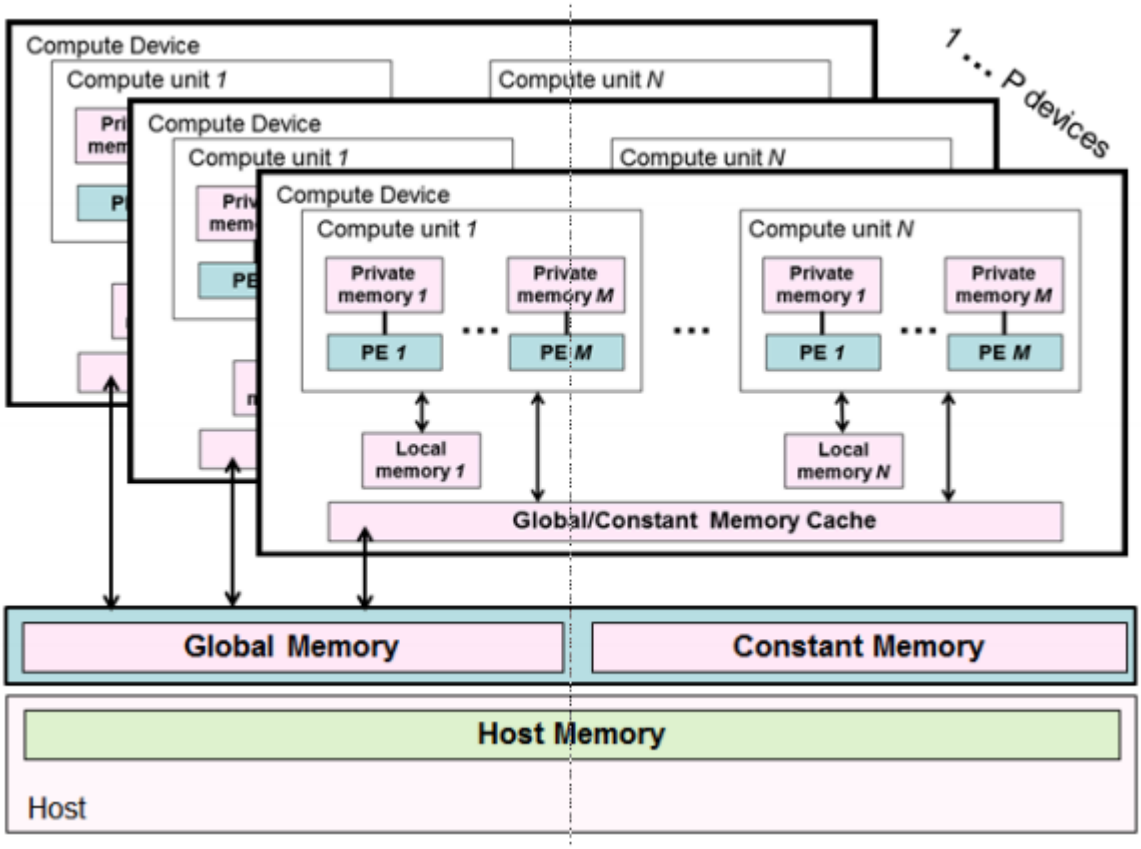
C kódu (funkce zvané kernely). Hostitel je připojen k těmto výpočetním zařízením, které jsou složeny z jednoho či více výpočetních jednotek (compute unit), což mohou být například streaming multiprocesory na NVIDIA grafických kartách. Tyto výpočetní jednotky se dále skládají z jednoho či více procesních elementů (processing element). Tyto procesní elementy musí vykonávat kód na principu paralelní architektury SIMD nebo SPMD. Procesním elementem by v případě NVIDIA grafické karty byl stream procesor.

Exekuční model definuje, jak se zpracovává celá aplikace a jak probíhá komunikace mezi hostitelem a výpočetními zařízeními. Celý program se skládá z hostitelské části a kernelů. Hostitelská část definuje použitá výpočetní zařízení, zajišťuje přesun dat mezi RAM a pamětí výpočetních zařízení a volá kernelové funkce. Kernelová funkce je pak paralelně vykonávána jednotlivými OpenCL výpočetními zařízeními. Každá taková instance kernelu se nazývá pracovní jednotka (work-item) a je vykonávána procesním elementem. Tyto pracovní jednotky jsou organizovány do pracovní skupiny (work-group), kterou zpracovává výpočetní jednotka. V rámci jedné pracovní skupiny mohou pracovní jednotky sdílet lokální paměť a být synchronizovány. Pracovní skupiny jsou společně a bez možnosti vzájemné synchronizace vykonávány na jednom výpočetním zařízení. Kernel se stará o vytvoření 1D, 2D nebo 3D indexového prostoru (index space), v kterém jsou svými unikátními indexy jednoznačně identifikovatelné pracovní skupiny a pracovní jednotky. Tento indexový prostor, v kterém jsou seskupeny jednotlivé pracovní skupiny, se nazývá NDRange. Celý princip je velmi podobný technologii CUDA, kde pracovní jednotky jsou vlákna, pracovní skupiny bloky a indexový prostor je dle CUDA terminologie mřížka. Předtím, než může být kernel předán k výpočtu, musí být vytvořen v hostitelské části aplikace kontext, který obsahuje všechny informace potřebné ke komunikaci mezi hostitelem a výpočetními zařízeními a také informace nutné k samotnému výpočtu. Kontext musí obsahovat:

- Zařízení - informace o OpenCL výpočetních zařízeních, které budou využívány.
- Kernely - množiny kernelů v podobě OpenCL C funkcí, které budou spuštěny na

OpenCL výpočetních zařízeních.

- Programové objekty - OpenCL zdrojové a binární kódy programů, které implementují potřebné množiny kernelů.
- Paměťové objekty - data, s kterými bude operováno na OpenCL výpočetních zařízeních, a které budou množiny kernelů zpracovávat.
- Příkazové fronty - příkazy potřebné k řízení výpočtů množin kernelů na OpenCL výpočetních zařízeních. Příkazy v příkazových frontách se dělí do dvou skupin. Ty, které budou vykonávány v pořadí přesně jak jdou za sebou (in order), a ty, které jsou na pořadí nezávislé (out of order). Mezi příkazy v příkazových frontách patří:
  - Příkazy zařízení - příkazy, které zajišťují ovládání OpenCL výpočetních zařízení a jejich vzájemnou synchronizaci.
  - Příkazy kernelů - příkazy, které se starají o spouštění kernelů a jejich vzájemnou synchronizaci.
  - Paměťové příkazy - příkazy, které se starají o přesunování dat.



Obrázek 2.5: (Paměťový model OpenCL (převzato z [9])

Paměťový model definuje abstraktní paměťovou hierarchii, kterou kernely využívají, bez ohledu na paměťovou architekturu použitých OpenCL výpočetních zařízení. Tento paměťový model velmi připomíná paměťovou hierarchii současných GPU, ale nijak nelimituje

jiné akcelerátory. Kernel má přístup ke čtyřem typům paměti. Následující popis jednotlivých typů má fyzické umístění pamětí popsáno z pohledu GPU, protože OpenCL vyžaduje tuto paměťovou specifikaci pro všechna výpočetní zařízení, i když mají jinou paměťovou architekturu (například v případě použití vícejádrového CPU jsou všechny typy paměti fyzicky v paměti hostitele) [17]. Tyto typy pamětí jsou:

- Globální paměť - tato paměť se fyzicky nachází v hlavní paměti výpočetního zařízení. Z globální paměti (global memory) můžou číst i do ní zapisovat všechny instance kernelu a hostitel. Pomocí SVM (shared virtual memory) se může globální paměť rozšířit o část paměti hostitele tím, že instance kernelu dostanou přístup do adresového prostoru paměti hostitele.
- Konstantní paměť - tato paměť se také fyzicky nachází v hlavní paměti výpočetního zařízení, ale v případě zařízení, které podporuje kešování konstantní paměti (constant memory), může být kešována přímo na čip a díky tomu zrychlena. Hostitel může konstantní paměť využívat pro čtení i zápis, ale instance kernelu z ní mohou pouze číst.
- Lokální paměť - tato paměť je fyzicky ve sdílené paměti na každé výpočetní jednotce výpočetního zařízení. Díky tomu z ní může číst i zapisovat každá instance kernelu v dané výpočetní skupině. Hostitel však musí přistupovat k lokální paměti (local memory) přes globální paměť.
- Privátní paměť - tuto paměť má svoji vlastní každá instance kernelu, která ji může využívat pro čtení i zápis. Nikdo jiný do privátní paměti (private memory) nemůže jakkoliv přistupovat a fyzicky se jedná o registry každého procesního elementu.

Programovací model definuje, jakým způsobem probíhá samotný výpočet množinou kernelů a jak je tento výpočetní model namapován na fyzický hardware. OpenCL umožňuje využívat dva programovací modely či jejich kombinaci. Jedním je datově paralelní programovací model a druhým úlohově paralelní programovací model. Datově paralelní programovací model je ve standardu OpenCL preferován a jedná se o spouštění množiny instancí stejných kernelů, které provádějí jednu úlohu nad různými daty paralelně. Můžeme tedy říci, že je vždy paralelně provedena jedna instrukce nad různými daty, což odpovídá paralelní architektuře SIMD. Úlohově paralelní programovací model (multitasking) umožňuje spouštět množinu instancí různých kernelů, kdy každá instance kernelu řeší paralelně jinou úlohu. Zde každý procesní element zpracovává různé instrukce nad různými daty, jak je definováno v paralelní architektuře MIMD. Tento programovací model má však nevýhodu nemožnosti komunikace jednotlivých instancí kernelů mezi sebou. Typicky se akcelerovaná část OpenCL programu skládá z následujících kroků: Vyberou se potřebná výpočetní zařízení (například GPU), vytvoří se kontext, vytvoří se příkazové fronty, zkompiluje se program, vytvoří se množina kernelů, alokuje se paměť na výpočetních zařízeních, provede se samotný výpočet, vrátí se výsledek hostiteli a nakonec se uvolní paměť na výpočetních zařízeních.

2.2 Počítačový cluster

Další možností jak dosáhnout výkonného paralelního výpočtu je vytvoření počítačového clusteru. Počítačový cluster je seskupení spolupracujících počítačů, které jsou propojeny počítačovou sítí, na rozdíl od GPGPU či vícejádrového CPU, kde jsou jednotlivé výpočetní

jednotky propojeny sběrnicí. Nejedná se tak o paralelní architekturu se sdílenou pamětí, ale o architekturu s distribuovanou pamětí. Pomyslným vrcholem paralelních architektur je využití obou paralelních architektur. Můžeme tak paralelně pracující počítače, běžící například na principu GPGPU, propojit do paralelního clusteru. Takové architektuře se dle paměti říká distribuovaně sdílená architektura paměti. Dle Flynnovy klasifikace by pak takový superpočítač byl MIMD clusterem multiprocesorových počítačů, kde každý procesor je architektury SIMD. Na tomto principu dnes běží většina superpočítačů.

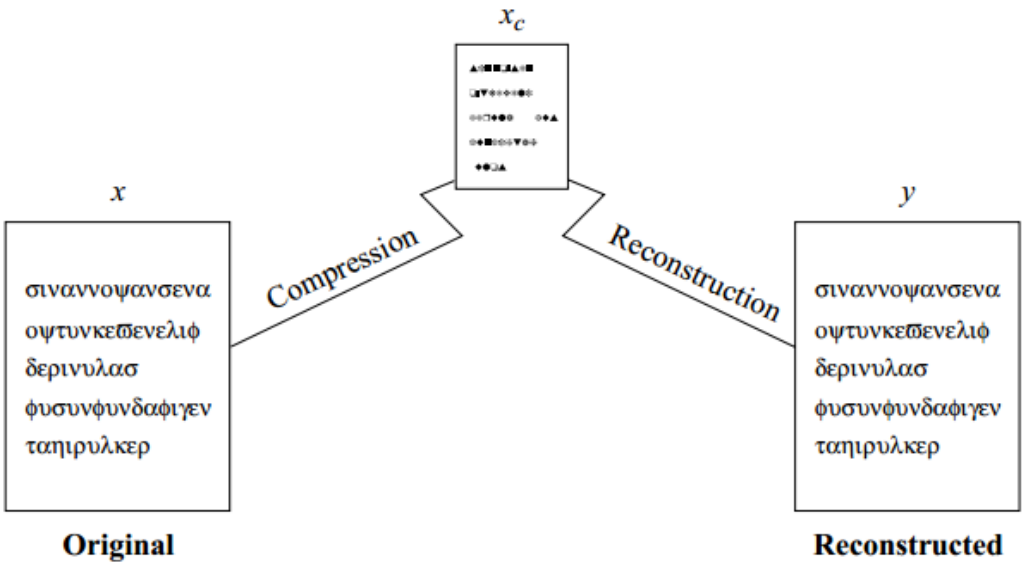
Komunikace po síti mezi jednotlivými počítači v clusteru je většinou zajištěna pomocí MPI (Message Passing Interface), což je komunikační protokol, který řeší komunikaci pomocí zasílání zpráv a je nezávislý na programovacím jazyku. Jednou ze známých implementací MPI je Open MPI[9].

[9] http://www.open-mpi.org

Kapitola 3

Kompresní algoritmy

Pokud bychom si měli popsat kompletní kompresní techniku, zjistíme, že je to vlastně dvojice algoritmů, kompresní a dekompresní (rekonstrukční). Kompresní algoritmus vytváří ze vstupu jeho komprimovanou reprezentaci, jejíž objem dat je, v případě efektivní komprese, menší než původní vstup. Dekompresní algoritmus poté z komprimované reprezentace vytvoří rekonstrukci původního vstupu. Kompresní techniky se dělí na dvě velké skupiny, ztrátové komprese a bezeztrátové komprese. V případě že vstup a rekonstrukce původního vstupu jsou totožné, jedná se o bezeztrátovou kompresi, v opačném případě jde o ztrátovou kompresi.



Obrázek 3.1: Znázornění komprese a rekonstrukce (převzato z [16])

Techniky ztrátové komprese většinou způsobí ztrátu některých informací z původního vstupu, rekonstrukce dat neproběhne zcela přesně. Na druhou stranu většinou poskytují mnohem vyšší kompresní poměr v objemu dat mezi původním vstupem a komprimovanou reprezentací. To je způsobeno dvojicí důležitých částí kompresního algoritmu ztrátové komprese, transformací původních dat a potlačením některých dat. Transformace většinou

převádí původní data v podobě signálu do frekvenční domény, která je vhodná pro zjištění, která data bude vhodné potlačit. Používané transformace jsou například DCT (discrete cosine transform), FFT (fast fourier transform) nebo DWT (discrete wavelet transform). Poté dochází k potlačování dat, kdy se většinou rozhoduje podle schopností člověka konkrétní data vnímat (například frekvence zvuku nebo kvalita obrazu). Nakonec může být na data provedena i některá z bezeztrátových komprimačních technik. Techniky ztrátové komprese se využívají tam, kde ztráta kvality komprimovaných dat není problém, například u zvuku či obrazu, kde nedokonalost lidského sluchu či zraku dává velký prostor pro možnost potlačení dat.

Techniky bezeztrátové komprese neumožňují ztrátu žádné informace z původního vstupu dat. Rekonstrukce komprimovaných dat musí proběhnout naprosto přesně do podoby původního vstupu. Bezeztrátová komprese se využívá například při kompresi textových dat nebo dat určených pro pozdější úpravy. Mnoho bezeztrátových kompresních programů používá kombinace kompresních algoritmů a často také transformaci dat, která má upravit vstupní data pro dosažení lepšího kompresního poměru. Tato práce se nadále bude zabývat pouze technikami bezeztrátové komprese a transformacemi, které se v kombinaci s algoritmy bezeztrátové komprese používají. Většina technik bezeztrátové komprese se dá rozdělit dle dvou kompresních metod na slovníkové metody komprese a statistické metody komprese.

Slovníkové kompresní metody fungují na principu nacházení opakujících se částí textu, které jsou poté redukovány na jediný výskyt a místo zbývajících výskytů je vložen odkaz. Tento vložený odkaz může odkazovat na předchozí výskyt opakující se části textu nebo na speciální slovník v paměti, který si algoritmus vytváří z opakujících se textů během průchodu vstupních dat. Samotná komprimace je pak zajištěna tím, že odkaz na opakující se část textu je kratší než nahrazovaný text. Pokud je toto splněno, objem dat se zmenšuje. Algoritmy založené na tomto principu publikovali v letech 1977 a 1978 Abraham Lempel a Jasob Ziv, po nichž se oba algoritmy nazývají LZ77 a LZ78 [12]. První zmiňovaný při průchodu vstupními daty nahrazuje opakující se části textu za odkazy na předchozí výskyt, zatímco druhý odkazuje na slovník, který si ukládá v paměti. V dnešní době existuje kompresních algoritmů, založených na tomto principu, více, ale většinou se jedná o vylepšení původních dvou.

Statistické kompresní metody pracují se statistikou počtu výskytů jednotlivých znaků ve vstupních datech. Znaky s vysokou pravděpodobností výskytu jsou nahrazeny kódovým slovem s co nejkratší délkou, zatímco pro znaky s nízkou pravděpodobností výskytu jsou určeny zbylé delší kódová slova. Díky tomuto postupu je většina znaků vstupních dat nahrazena bitovými řetězci kratšími než je 8 bitů (standardní bitová délka jednoho znaku). Důležitou součástí kompresní metody však není pouze způsob kódování vstupních dat, ale také model, dle kterého se rozdělí znaky podle pravděpodobnosti výskytu. V tomto případě rozdělujeme dva základní modely, statický a adaptivní. Statický model, který je většinou dvouprůchodový, je vždy kompletně vytvořen před samotným začátkem komprimace a je většinou určen podle typu komprimovaných dat (například četnost jednotlivých znaků v průměrném českém textu). Adaptivní model, který je většinou jednoprůchodový, pravděpodobnost výskytu znaku neustále přepočítává podle již načtených znaků. Tento model je pomalejší a paměťově náročnější, ale obecně dosahuje lepších výsledků komprese. Nejznámějšími zástupci statistických kompresních metod jsou Huffmanovo kódování a aritmetické kódování. Těmito kompresními metodami se budeme dále zabývat.

3.1 Pomocné transformace

3.1.1 Burrowsova-Wheelerova transformace

Burrows-Wheeler transform (BWT) je pomocným algoritmem, který se používá při bezeztrátové kompresi dat, přestože sám data nekomprimuje. Tento algoritmus vytvořili Michael Burrows a David Wheeler v roce 1994 [16], a jeho funkcí je vytvořit takovou permutaci vstupních dat, která obsahuje stejné symboly několikrát za sebou, a zároveň je možné snadno znovu rekonstruovat původní vstupní data. Takový výstup je pak velmi vhodný pro další transformace a komprese jakou jsou například Move-to-front transformace či Run-length encoding, které jsou popsány níže.

Celé hledaní vhodné permutace funguje tak, že se vytvoří všechny možné rotace vstupních dat jako řádky tabulky, které se následně lexikograficky uspořádají. Výslednou permutací je pak poslední sloupec symbolů uspořádaných rotací, jak přehledně ukazuje tabulka BWT transformace textu. Pro jednoduchost byl zvolen krátký řetězec, ale BWT dosahuje tím lepších výsledků, čím delší jsou vstupní data. Pro ukázku byl do řetězce přidán symbol EOF (end-of-file), pomocí kterého je možné provést rekonstrukci vstupních dat z dat výstupních bez jakékoliv další pomocné informace. Nevýhodou tohoto způsobu je nutnost pracovat s jedním symbolem navíc, jinou možností by bylo uložit informaci o řádku v tabulce seřazených rotací, na kterém je původní vstupní řetězec (v tomto případě třetí řádek, tedy index $I = 2$).

Vstup	**Rotace**	**Seřazení**	**Výstup**
T	TEST@	EST@T	T
E	EST@T	ST@TE	E
S	ST@TE	TEST@	@
T	T@TES	T@TES	S
@	@TEST	@TEST	T

Tabulka 3.1: BWT transformace textu ('@' značí EOF)

Pro zpětnou transformaci na rekonstrukci původních dat stačí vytvořit prázdnou tabulku, kam vložíme výstupní permutaci jako první sloupec. Ten následně lexikograficky uspořádáme a znovu vložíme výstupní permutaci jako první sloupec. Tento postup stačí opakovat tolikrát, kolik symbolů permutace má. Pokud výstupní permutace BWT obsahovala symbol EOF, pak v posledním kroku najdeme v tabulce takový řetězec, který končí symbolem EOF. V opačném případě je nutné znát index řádku, na kterém se rekonstruovaný původní řetězec nachází, již z původní aplikace BWT.

Vstup	**1.**		**2.**		**3.**		**4.**		**5.**
T	E	TE	ES	TES	EST	TEST	EST@	TEST@	EST@T
E	S	ES	ST	EST	ST@	EST@	ST@T	EST@T	ST@TE
@	T	@T	TE	@TE	TES	@TES	TEST	@TEST	**TEST@**
S	T	ST	T@	ST@	T@T	ST@T	T@TE	ST@TE	T@TES
T	@	T@	@T	T@T	@TE	T@TE	@TES	T@TES	@TEST

Tabulka 3.2: BWT zpětná transformace textu ('@' značí EOF)

Pro zpětnou rekonstrukci původních dat existuje ještě další, paměťově méně náročný způsob [15]. Pro tento postup je nutné již při první transformaci vstupních dat na permutaci pomocí BWT uložit do paměti index řádku tabulky seřazených rotací, na kterém se nachází původní vstupní řetězec (v ukázce je index $I = 2$). Poté lexikograficky uspořádáme výstupní řetězec L, čímž vznikne pomocný řetězec F (ten je stejný jako první sloupec seřazených rotací). Řetězec F potřebujeme pouze pro vytvoření pole T, které obsahuje indexy jednotlivých symbolů výstupního řetězce L po lexikografickém uspořádání. Z výstupního řetězce BWT L (s délkou n), pole indexů T a konstanty I získáme zpět rekonstruovaný vstupní řetězec následujícím způsobem:

$$S[n-1-i] \leftarrow L[T^i[I]], \text{ pro } i = 0, 1, ..., n-1, \text{ kde } T^0[j] = j, \text{ a } T^{i+1}[j] = T[T^i[j]].$$

Celá rekonstrukce vstupního řetězce je pak provedena takto:

$$S[4-1-0] = L[T^0[I]] = L[T^0[2]] = L[2] =' T',$$
$$S[4-1-1] = L[T^1[I]] = L[T[T^0[I]]] = L[T[2]] = L[3] =' S',$$
$$S[4-1-2] = L[T^2[I]] = L[T[T[T^0[I]]]] = L[T[T[2]]] = L[T[3]] = L[1] =' E',$$
$$S[4-1-3] = L[T^3[I]] = L[T[T[T[T^0[I]]]]] = L[T[T[T[2]]]] = L[T[T[3]]] = L[T[1]] = L[0] =' T'.$$

Rotace	**Seřazení**	**Index**	F	T	L
TEST	ESTT	0	E	2	T
ESTT	STTE	1	S	0	E
STTE	TEST	**2**	T	3	T
TTES	TTES	3	T	1	S

Tabulka 3.3: Pomocné řetězce pro zpětnou BWT

3.1.2 Move-to-front transformace

Move-to-front (MTF) transformace je pomocný algoritmus pro kompresi dat, který se většinou používá po provedení Burrowsovy-Wheelerovy transformace. Jeho smyslem je snížit entropii vstupních dat, a tím vylepšit kompresní poměr při následném použití entropického kódování. Dosahuje toho tím, že nahrazuje symboly vstupních dat za malá čísla, v případě opakujících se stejných znaků ve vstupních datech dokonce symboly nahrazuje za posloupnost nul. Proto je výhodné transformaci MTF používat na datový vstup, který takové opakující se posloupnosti stejných znaků obsahuje, což bývá zajištěno algoritmem BWT. Algoritmus MTF musí mít připravený seznam všech symbolů, které se vyskytují ve vstupních datech, poté MTF nahrazuje symboly na vstupu za indexy stejného symbolu v seznamu, který je následně posunut na začátek celého seznamu. Jak můžeme vidět, algoritmus MTF musí udržovat informace nejen o transformovaných vstupních datech, ale i o celém seznamu symbolů, proto se může stát, že v případě, kdy vstupní data neobsahují opakující se posloupnosti stejných symbolů, tak použití transformace MTF naopak entropii dat zvýší. Proto se nedoporučuje používat algoritmus MTF bez předchozího použití BWT, který zvyšuje efektivitu MTF transformace. Při zpětné rekonstrukci vstupních dat se ve stejném pořadí, jako probíhala předchozí transformace, nahrazují indexy za odpovídající symboly ze seznamu symbolů a symbol je pak stejným způsobem přesunut na začátek.

Vstup	Výstup	Seznam	Vstup	Výstup	Seznam
R	3	EJMRSTU	J	1	EJMRSTU
E	1	REJMSTU	M	2	JEMRSTU
S	4	ERJMSTU	R	3	MJERSTU
E	1	SERJMTU	S	4	RMJESTU
T	5	ESRJMTU	U	6	SRMJETU
U	6	TESRJMU	E	5	USRMJET
J	5	UTESRJM	E	0	EUSRMJT
E	3	JUTESRM	E	0	EUSRMJT
M	6	EJUTSRM	E	0	EUSRMJT
E	1	MEJUTSR	T	6	EUSRMJT

Tabulka 3.4: Ukázka MTF před a po BWT

3.1.3 Run-length encoding

Run-length encoding (RLE) je velmi jednoduchá technika bezeztrátové komprese, která komprimuje opakující se sekvence stejných symbolů. Přestože sama o sobě nedosahuje vysokého kompresního poměru, často se používá jako pomocný algoritmus před použitím kvalitnějšího entropického kódování. Komprese RLE dosahuje nejlepších výsledku na datovém vstupu, který obsahuje co největší množství opakujících se posloupností stejných symbolů, proto je vhodné jej používat na výstupy algoritmu BWT či transformace MTF.

Zcela nejzákladnější algoritmus RLE nahrazuje posloupnosti stejných symbolů za dvojici obsahující délku posloupnosti a symbol. Tento postup může být použit i pro nahrazení všech symbolů, včetně těch, které se neopakují. Výhodou tohoto postupu je, že vstupní data mohou být načítány po jednom bytu. Nevýhodou však je, že v případě, kdy datový vstup neobsahuje opakující se posloupnosti stejných symbolů, může namísto komprese mít datový výstup až dvojnásobný datový objem. Tento problém můžeme snadno odstranit nahrazováním nejméně dvou stejných symbolů:

$$AZAZAZAZZZAZZAA \rightarrow AZAZAZA3ZA2Z2A(\text{úspora 1 znak})$$

Tento algoritmus má však také problém, protože může být použit jen na datový vstup, který sám neobsahuje žádná čísla. V případě, že datový vstup obsahuje číselné symboly, musíme komprimovanou dvojici označit použitím dalšího escapovacího symbolu. Komprimovat však posloupnost dvou stejných symbolů trojicí symbolů (escapovací symbol + délka posloupnosti + symbol) nemá však komprimační efekt, naopak délka dat narůstá. Pro tento postup je tedy nutné nahrazovat pouze posloupnosti stejných symbolů delší než nebo rovno třem: (jako escapovací znak byl zvolen '@')

$$AZZAA22ZZZAAAA2222 \rightarrow AZZAA22@3Z@4A@42(\text{úspora 2 znaky})$$

Ani tento algoritmus RLE není použitelný v každém případě. Pro datový vstup, který může obsahovat jakýkoliv existující symbol, není možné zvolit bezpečný obecný escapovací symbol. Možnou variantou RLE pro tento případ může být použití samotného nahrazovaného symbolu jako symbolu escapovacího. V tomto postupu by byla nahrazena každá posloupnost alespoň dvou stejných symbolů trojicí symbolů, kde první dva by byla dvojice nahrazovaných symbolů a třetí délka posloupnosti:

$$AZZAA22ZZZAAAA2222 \rightarrow AZZ2AA2222ZZ3AA4224(\text{prodloužení o 2 znaky})$$

Jak můžeme vidět, nahrazování alespoň dvou stejných symbolů trojicí symbolů může vést ke zvětšení datového objemu kódovaných dat. Použití algoritmu RLE tedy je většinou výhodné pouze pro datové vstupy, kde je zajištěno velké množství dlouhých posloupností opakujících se stejných symbolů.

Rekonstrukce původních vstupních dat po provedení algoritmu RLE je také jednoduché. Jakmile dekodér narazí na escapovací znak (ať už je to samotná délka posloupnosti v prvním případě, znak '@' v druhém případě či posloupnost dvou stejných znaků v případě třetím), načte délku posloupnosti a vypíše daný symbol několikrát.

3.2 Entropické kódování

3.2.1 Huffmanovo kódování

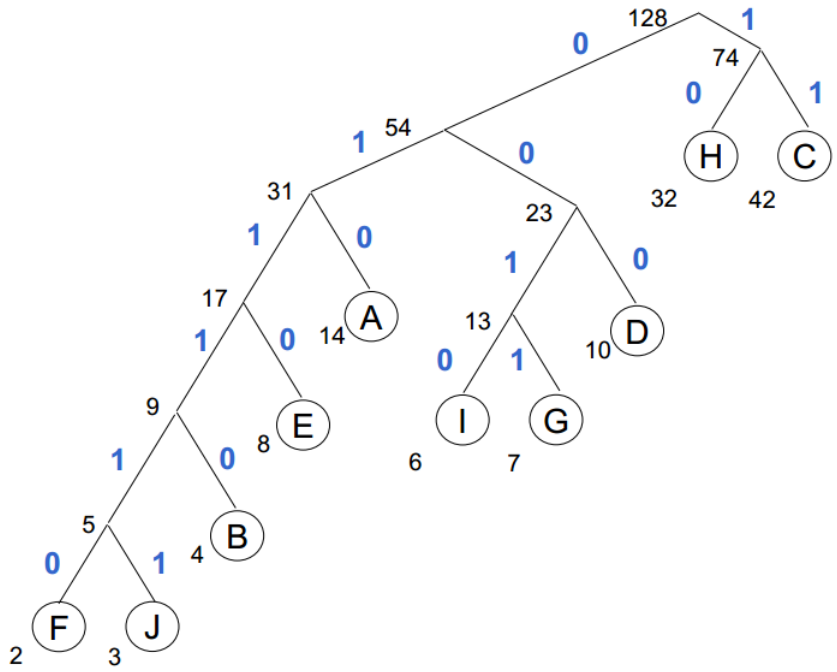
Huffmanovo kódování je jedním z nejstarších (bylo publikováno v roce 1952), ale i nejpoužívanějších a nejznámějších kódování pro bezeztrátovou kompresi dat. Samotnou komprimaci dat provádí na principu nahrazování velmi častých symbolů ve vstupních datech velmi krátkým bitovým kódem (nejčastější symbol může být nahrazen i jediným bitem), zatímco pro méně časté znaky zbude kód delší. Existují dvě hlavní varianty Huffmanova kódování, statická a adaptivní.

Statické Huffmanovo kódování je dvouprůchodové, kdy v prvním průchodu je potřeba zjistit četnost výskytu jednotlivých symbolů v datovém vstupu. Na základě této četnosti je vytvořen binární strom (lze použít i jinou datovou strukturu, ale binární strom se pro tento účel využívá nejčastěji), a z něj jsou poté následujícím postupem získány kódová slova, kterými nahradíme původní symboly[18]:

1. Jednotlivé symboly prohlásíme za uzly (listy) stromu. Postupně po dvojicích spojujeme uzly s nejmenším ohodnocením až zkonstruujeme Huffmanův strom.
2. Systematicky ohodnotíme hrany stromu (např. hrana vedoucí do uzlu s menším ohodnocením 0, jinak 1).
3. Cesta z vrcholu až k listu tvoří kódové slovo symbolu odpovídajícího danému listu.

Adaptivní varianta Huffmanova kódování je jednoprůchodová a nepotřebuje dopředu zjistit četnost výskytu jednotlivých symbolů v datovém vstupu. Místo toho sestavuje a neustále upravuje binární strom podle zpracovávaných znaků datového vstupu.

Nevýhodou Huffmanova kódování je, že pro rekonstrukci původních vstupních dat je nutné uchovat i jeho binární strom, bez kterého by rekonstrukční algoritmus nedokázal zjistit, jaká je délka kódových slov, a které symboly zastupují. Díky tomu je kompresní poměr horší a v extrémních případech může být výsledný datový objem vyšší než u datového vstupu. Použití pomocných transformací jako je BWT a MTF před použitím Huffmanova kódování upravuje data tak, aby byl kompresní poměr co nejlepší. V případě adaptivního Huffmanova kódování existují varianty, kdy není potřeba uchovávat pro rekonstrukci vstupních dat celý binární strom, ale vkládají do výstupního kódu escapovací znak před první výskyt každého symbolu, pomocí kterého je pak možné provést rekonstrukci vstupních dat.



Obrázek 3.2: Ukázka Huffmanova binárního stromu (převzato z [18])

Symbol	Počet výskytů	Kódové slovo
A	14	010
B	4	01110
C	42	11
D	10	000
E	8	0110
F	2	011110
G	7	0011
H	32	10
I	6	0010
J	3	011111

Tabulka 3.5: Kódová slova dle Huffmanova stromu

3.2.2 Aritmetické kódování

Alternativou k Huffmanově kódování může být například aritmetické kódování, které nenahrazuje jednotlivé symboly vstupních dat kratším binárním kódem, ale místo toho vytváří kód celého vstupu v podobě jediného desetinného čísla v intervalu $\langle 0,1)$. Tento interval je rozdělován na podintervaly podle načítaných symbolů vstupních dat, které jsou určeny z pravděpodobností výskytu daného symbolu, dokud není zpracován celý vstup. Ze vzniklého intervalu je možné poté vybrat libovolné číslo (nejlépe to, které je vyjádřeno nejkratším binárním kódem), a to definuje celý datový vstup a dá se zpětně rekonstruovat za předpokladu, že máme k dispozici seznam všech symbolů, jejich pravděpodobnosti a délku celého vstupu.

Sestavit zmíněný seznam všech vstupních symbolů a určit jejich pravděpodobnosti je nutné před provedením samotné komprese. Většinou je to provedeno staticky, průchodem celého datového vstupu předem, existuje však také algoritmus adaptivního aritmetického kódování, který průběžně upravuje pravděpodobnosti výskytu symbolů a jejich podintervaly během zpracovávání vstupních dat.

Práci algoritmu aritmetického kódování si můžeme názorně ukázat na příkladu, kde zakódujeme řetězec $ABABC$ se seznamem symbolů A (pravděpodobnost $0,4$), B (pravděpodobnost $0,4$) a C (pravděpodobnost $0,2$). Jak můžeme vidět v přehledné tabulce, z původního intervalu ⟨0,1) se vytváří podintervaly dle pravděpodobností jednotlivých symbolů a z nich se ve stejném poměru vytváří další podintervaly dle symbolu na vstupu, dokud není celý řetězec zpracován. Z výsledného intervalu ⟨0,20608;0,2112) si můžeme zvolit libovolné číslo, přičemž nejvýhodnější je $0,2109375$, protože má nejkratší binární zápis $(0011111001011)_2$. Toto binární číslo bude uchováno jako kód, ale není nutné jej uchovávat kompletní, protože celá část čísla bude s aritmetickým kódováním vždy 0. Výsledný kód řetězce bude tedy $(001011)_2$, což je oproti 40 bitům pěti znaků efektivní komprese. Prakticky je však aritmetické kódování nevhodné pro malé řetězce, protože kvůli rekonstrukci vstupního řetězce musíme ke komprimovaným datům připojit seznam všech symbolů, jejich pravděpodobnosti a délku celého vstupu.

Podinterval 'A'	**Podinterval 'B'**	**Podinterval 'C'**	**Vybraný podinterval**
⟨**0;0,4**)	⟨0,4;0,8)	⟨0,8;1)	A
⟨0;0,16)	⟨**0,16;0,32**)	⟨0,32;0,4)	B
⟨**0,16;0,224**)	⟨0,224;0,288)	⟨0,288;0,32)	A
⟨0,16;0,1856)	⟨**0,1856;0,2112**)	⟨0,2112;0,224)	B
⟨0,1856;0,19584)	⟨0,19584;0,20608)	⟨**0,20608;0,2112**)	C

Tabulka 3.6: Aritmetické zakódování řetězce $ABABC$

Kódové číslo	**Symbol**	**Nové kódové číslo**
$0,2109375$	A	$(0,2109375 - 0)/0,4 = 0,52734375$
$0,52734375$	B	$(0,52734375 - 0,4)/0,4 = 0,318359375$
$0,318359375$	A	$(0,318359375 - 0)/0,4 = 0,7958984375$
$0,7958984375$	B	$(0,7958984375 - 0,4)/0,4 = 0,98974609375$
$0,98974609375$	C	rekonstruován požadovaný počet symbolů

Tabulka 3.7: Rekonstrukce řetězce $ABABC$

Rekonstrukční algoritmus rozdělí symboly ze seznamu do stejných intervalů jako při kompresi, načte kódové číslo a zjistí, do kterého intervalu patří. Podle toho vypíše daný symbol, načtené kódové číslo upraví podle vzorce

kódové číslo = (kódové číslo−spodní hranice intervalu symbolu)/pravděpodobnost symbolu

a následně znovu porovnává s intervaly, což celé opakuje několikrát podle délky rekonstruovaného řetězce.

Kapitola 4

Implementace a analýza výkonnosti

V rámci této práce byla implementována bezeztrátová komprese a dekomprese dat, přesněji statistická kompresní metoda založena na Burrowsově-Wheelerově transformaci, v programovacím jazyce C/C++. Konkrétně jde o algoritmy Burrowsova-Wheelerova transformace, move-to-front transformace, run-length encoding a k entropickému zakódování byl využit algoritmus adaptivního aritmetického kódování. Kompresní celek (bez entropického zakódování) byl implementován také v paralelní variantě, která provádí výpočty pomocí paralelní architektury GPU za využití technologie NVIDIA CUDA.

Detailnější rozbory návrhu, sekvenční implementace, způsobu akcelerace pomocí paralelní architektury a výsledné analýze výkonnosti jednotlivých kompresních algoritmů a transformací jsou uvedeny v jednotlivých sekcích této kapitoly. Již zde je možno uvést, že všechny varianty jednotlivých algoritmů byly implementovány tak, aby měly stejný výstup, a byly tedy kompatibilní s jediným dekodérem. Rozdíl v kompresním poměru jednotlivých variant je tudíž nulový.

Překlad všech zdrojových kódů byl zajištěn pomocí NVIDIA GPU Computing Toolkit v6.0, a to konkrétně překladačem nvcc.exe pro zdrojové kódy CUDA, s využitím Microsoft Visual Studio 9.0 a jeho překladačem zdrojových kódů C/C++ cl.exe. Překlad proběhl na operačním systému Windows 7 Home Premium 32bit Service Pack 1 a kdykoliv není v analýze výkonnosti algoritmů řečeno jinak, pak byly binární soubory testovány na procesoru Intel Core2 Duo E8400 pro sekvenční část a grafické kartě NVIDIA GeForce GT610 pro paralelní část programu.

K testování algoritmů byly použity nenáhodná textová data o velikostech 1 kB, 10 kB, 100 kB a pro rychlejší algoritmy také 1 MB a 10 MB. Popis jednotlivých binárních a zdrojových souborů v příloze, stejně tak jako dávkových souborů pro testování a analýzu výkonnosti, je uveden v readme.txt u přílohy na CD.

4.1 Burrowsova-Wheelerova transformace

BWT algoritmus se skládá zejména ze dvou časově náročných částí, vytvoření všech rotací datového vstupu a jejich následnému lexikografickému seřazení. Zatímco vytvoření všech rotací je cyklus ($O(N)$) operací s lineární složitostí $O(N)$, což se dohromady rovná cyklu s kvadratickou složitostí $O(N^2)$, tak lexikografické seřazení má složitost dle zvoleného řadícího algoritmu. Pro demonstrační účely byl v této práci použit řadící algoritmus quicksort s průměrnou lineárnělogaritmickou složitostí $O(N * log(N))$. K porovnání byl implementován také odlišný algoritmus BWT, inspirován článkem Marka Nelsona [11], který nevytváří

všechny rotace vstupních dat předem, ale pouze dočasně během průběhu řadícího algoritmu, aby dle nich seřadil dále využité indexy symbolů vstupních dat. Tento algoritmus je tak rychlejší o jeden cyklus s kvadratickou složitostí, ale zpomaluje efektivitu quicksortu.

V případě implementace akcelerace BWT na GPU má smysl uvažovat akceleraci těchto dvou asymptoticky nejsložitějších částí. S řazením řetězců na GPU je však problém. V rámci této práce nebyla nalezena ani úspěšně vytvořena žádná efektivní implementace řazení řetězců podle všech symbolů, které obsahuje. Nejblíže tomuto řešení je nejspíše implementace string sort z CUDA knihovny CUDPP[1], která však má problém s kompatibilitou s 32bitovými operačními systémy a řadí řetězce pouze podle čtyř prvních symbolů, což by v případě BWT mohlo být nežádoucí. Řešením akcelerace BWT, které je v této práci prezentováno, tedy je převést na GPU vytvoření všech rotací vstupních dat ve snaze nahradit funkci s kvadratickou složitostí $O(N^2)$ paralelní funkcí, která bude mít ze sekvenčního pohledu složitost konstantní $O(1)$, tedy bude provedena jedinou jednoduchou operací, která souběžně poběží na mnoha procesorech dle délky vstupního řetězce.

Zdržením, které nutně musí nastat při jakékoliv snaze akcelerovat algoritmus na GPU, je nutnost přesunutí vstupního datového řetězce z paměti RAM na paměť grafického čipu, a po provedení rotací následně vrátit vektor rotací do operační paměti, odkud jej může procesor využít k seřazení. Díky tomuto postupu nahradíme funkci s kvadratickou složitostí $O(N^2)$ trojicí funkcí s asymptotickými složitostmi $O(N)+O(1)+O(N)$. Není snadné předem odhadnout praktickou rychlost zpracování těchto funkcí z důvodu zanedbávání multiplikativních i aditivních konstant při určování asymptotické složitosti, ale již z parabolického vyjádření kvadratické funkce můžeme očekávat, že pro vysoká čísla N, tedy velikost načteného bloku vstupních dat, bude rychlost zpracování na CPU nižší než součet lineárních funkcí na GPU. Toto chování si můžeme ověřit praktickým otestováním, které ukazuje tabulka.

Datový vstup	**Sekvenční varianta**	**Paralelní varianta**	**Paralelní varianta na GPU GTX 770**
1 kB dat	20 ms	220 ms	170 ms
10 kB dat	640 ms	900 ms	460 ms
100 kB dat (15 000 B / blok)	9 420 ms	9 140 ms	4 390 ms
100 kB dat (18 000 B / blok)	13 150 ms	12 190 ms	4 850 ms
100 kB dat (20 000 B / blok)	16 290 ms	14 160 ms	5 040 ms

Tabulka 4.1: Srovnání vytváření všech rotací

Na základě tohoto zjištění můžeme předpokládat, že pro sekvenční algoritmus je časově výhodnější zpracovávat vstup po malých blocích vstupních dat ($10^2 * 100 < 100^2 * 10$)[2], zatímco u paralelní verze se rychlost zpracování pro různě velké bloky vstupních dat příliš nemění ($(10+10)*100 = (100+100)*10$)[3]. Můžeme namítnout, že bychom mohli zrychlit sekvenční algoritmus zmenšením načítaných bloků vstupních dat tak, že by výsledná kva-

[1] http://cudpp.github.io

[2] příklad: kvadratická funkce $O(N^2)$, datový vstup 1000 symbolů, v prvním případě velikost bloku 10 symbolů (nutno 100 opakování), v druhém případě velikost bloku 100 symbolů (nutno 10 opakování)

[3] příklad: dvojice lineárních funkcí $O(N)+O(N)$, datový vstup 1000 symbolů, v prvním případě velikost bloku 10 symbolů (nutno 100 opakování), v druhém případě velikost bloku 100 symbolů (nutno 10 opakování)

dratická složitost byla menší nežli součet lineárních, a díky tomu by sekvenční algoritmus prováděl BWT rychleji než paralelní, protože by GPU nemohlo konkurovat využitím plného potenciálu paralelního zpracování a přesunování dat mezi operační pamětí a grafickou pamětí by tento algoritmus velmi zpomalovalo, ale optimalizovat sekvenční kód není cílem této práce. Tato práce si naopak klade za cíl ukázat využití paralelního algoritmu, a to je v tomto případě výhodné, pokud jsme nuceni načítat velké datové bloky pro BWT.

Analýzou výkonnosti celého BWT algoritmu zjistíme, že rozdíly v potřebném čase k provedení nejsou tak velké, jako v případě pouhého generování rotací vstupních dat. To je u jednotlivých variant BWT způsobeno řazením rotací vstupních dat, které je ve všech případech sekvenční. Pokud bychom nalezli vhodný způsob, jak toto řazení provádět efektivně paralelně na GPU, pak by akcelerace měla mnohem větší efekt. Přesto lze v tabulce srovnání výkonnosti jednotlivých variant vidět, že výkonná herní grafická karta dosahuje průměrně nejlepších výsledků. Výjimkou samozřejmě je optimalizovaná sekvenční BWT varianta dle Marka Nelsona, která funguje na zcela jiném principu.

Datový vstup	**BWT [11] Sekvenční**	**BWT Sekvenční**	**BWT Paralelní**	**BWT Paralelní na GPU GTX 770**
1 kB dat	20 ms	20 ms	240 ms	190 ms
10 kB dat	60 ms	780 ms	1 170 ms	1 050 ms
100 kB dat (15 000 B / blok)	780 ms	22 020 ms	25 950 ms	**20 800 ms**
100 kB dat (18 000 B / blok)	990 ms	65 140 ms	73 290 ms	65 550 ms
100 kB dat (20 000 B / blok)	1 200 ms	90 360 ms	97 450 ms	**89 030 ms**

Tabulka 4.2: Srovnání variant algoritmu BWT

4.2 Move-to-front transformace

Move-to-front transformace je rychlý sekvenční algoritmus, který zpracovává vstupní data po jednom symbolu. Tomuto přístupu je velmi těžké konkurovat paralelním přístupem. Můžeme však na tomto algoritmu názorně ukázat, že paralelizovat celý výpočet není nutně výhodnější než paralelizace dílčího úkonu.

MTF transformace využívá ke své funkci seznam symbolů. Veškerá manipulace s tímto seznamem má podobu cyklu s lineární časovou složitostí $O(N)$. Všechny tyto úkony se seznamem symbolů se dají paralelizovat a převést jejich provedení na výpočetní jednotku grafické karty. Tabulka srovnávající jednotlivé varianty MTF však ukazuje, že paralelizovat všechny tyto manipulace se seznamem symbolů není výhodné. Přenášení dat z operační paměti do paměti grafického akcelerátoru a zpět znatelně prodlužuje dobu nutnou pro výpočet. Pokud se však zaměříme pouze na paralelizaci vygenerování seznamu symbolů, není zpomalení příliš znatelné při velkém množství vstupních symbolů.

Nevýhodou pro paralelizaci MTF je, že tento seznam symbolů má vždy maximálně 256 prvků, protože obsahuje všechny možné jednobytové symboly. Tento počet prvků je příliš malý na to, aby akcelerace výpočtu na GPU ušetřila větší čas než čas nutný k přesunu dat do paměti GPU a zpátky do operační paměti. Pokud bychom však uvažovali o hypotetické

Datový vstup	MTF Sekvenční	MTF Paralelní Generování seznamu	MTF Paralelní
1 kB dat	<10 ms	60 ms	180 ms
10 kB dat	10 ms	80 ms	1 190 ms
100 kB dat	80 ms	120 ms	11 250 ms
1 MB dat	680 ms	750 ms	112 300 ms
10 MB dat	6 800 ms	6 880 ms	1 134 240 ms

Tabulka 4.3: Srovnání variant algoritmu MTF

situaci, kdy by seznam měl 1024 prvků, pak by se již paralelizace vyplatila, jak vidíme ve srovnávací tabulce.

Datový vstup	MTF Sekvenční	MTF Paralelní
1 kB dat	10 ms	80 ms
10 kB dat	20 ms	110 ms
100 kB dat	90 ms	140 ms
1 MB dat	710 ms	770 ms
10 MB dat	6 970 ms	**6 920 ms**

Tabulka 4.4: Akcelerace generování seznamu o 1024 prvcích

4.3 Run-length encoding

Pro implementaci run-length encoding byla zvolena poslední z možností tohoto algoritmu popsaných v předchozí kapitole této práce. Tedy varianta, která nahrazuje opakující se sekvence symbolů vstupních dat za dvojici těchto symbolů následované bytem, který určuje počet opakování.

RLE je algoritmus, který velmi rychle zpracovává datový vstup po jednom symbolu bez jakýchkoliv náročných výpočtů. Časově nejnáročnější částí algoritmu je výpis výstupních symbolů s lineární časovou složitostí $O(N)$. Pokud bychom chtěli tento algoritmus paralelizovat na GPU, je nutné předělat sekvenční implementaci z načítání po symbolu na načítání bloků vstupních dat. Tímto způsobem je možné provést algoritmus RLE na GPU stejně kvalitně jako na CPU, nikoliv však rychleji.

Naopak nutnost přesouvat velké množství dat do paměti GPU a zase zpátky do operační paměti algoritmus značně zpomaluje. To však neznamená, že neexistuje žádný případ, kdy by paralelní implementace RLE s načítáním vstupních dat po blocích mohl být výhodnější než klasická sekvenční varianta tohoto algoritmu s načítáním vstupních dat po jednom bytu.

Navíc pokud nebudeme uvažovat čas potřebný na přesun dat z paměti grafické karty do operační paměti, zjistíme, že při velkém množství dat GPU provádí veškeré výpočty nutné k provedení algoritmu RLE rychleji než CPU. Ke zpomalení oproti sekvenční variantě dochází až kopírováním vypočítaných dat. Ve srovnávací tabulce výkonnosti tedy vidíme, že paralelní RLE je sice pomalejší než sekvenční, ale proběhl také test RLE bez výpisu výstupních dat a tedy i nutnosti kopírovat data zpět, kde vidíme, že paralelní varianta na

GPU dosahuje rychlejších výpočtů na velkých vstupních datech nežli sekvenční varianta na CPU.

Datový vstup	**RLE Sekvenční**	**RLE Paralelní**	**RLE (bez výstupu) Sekvenční**	**RLE (bez výstupu) Paralelní**
1 kB dat	10 ms	110 ms	<10 ms	60 ms
10 kB dat	20 ms	120 ms	<10 ms	70 ms
100 kB dat	60 ms	680 ms	10 ms	80 ms
1 MB dat	640 ms	6 070 ms	80 ms	100 ms
10 MB dat	6 200 ms	60 096 ms	700 ms	**320 ms**

Tabulka 4.5: Srovnání variant algoritmu RLE

Závěr

Cílem této práce bylo prozkoumat možnosti paralelního programování s důrazem na GPGPU a následně využít získané znalosti k akceleraci vybraných algoritmů používaných k bezeztrátové kompresi dat, čemuž předcházela teoretická příprava v této oblasti. K akceleraci komprese dat založené na Burrowsově-Wheelerově transformaci k efektivní statistické kompresi byla, po důkladném prozkoumání dalších možností, použita technologie CUDA od společnosti NVIDIA.

Přestože velká konkurence použité proprietární paralelní architektury CUDA, OpenCL, si drží mnoho výhod, z nichž můžeme jmenovat otevřenost kódu, nezávislost na cílové paralelní architektuře a uznání OpenCL jako softwarového standardu pro paralelní programování heterogenních počítačových systémů, několik dalších faktorů mne vedlo ke konečné volbě řešení CUDA. Mezi tyto faktory patří dle mého názoru lepší dokumentace, dostupnost podpůrných materiálů, optimalizace pro konkrétní hardware, snadnější ladění aplikace a rychlejší učící křivka.

Z počátku jsem se domníval, že by bylo zajímavé, byť i mimo rozsah této práce, implementovat konkrétní kompresní algoritmus za použití CUDA i OpenCL pro GPGPU akceleraci a vzájemně je porovnat. Této problematice se však věnuje již mnoho veřejně dostupných studií. Myslím si, že mnohem zajímavější by bylo srovnat GPGPU a dnešní nejmodernější vícejádrové procesory. Mnoho algoritmů je výhodnější optimalizovat sekvenčně nežli je paralelizovat, přesun dat mezi operační pamětí a pamětí GPU přes sběrnici PCI-Express je velkým zdržením výpočtů a velké množství multiprocesorů GPU není zárukou velkého zrychlení pro všechny typy úloh, jak ukazuje Amdahlův zákon.

GPGPU je jednou z možností využití paralelního programování a tato práce názorně ukazuje, jak ji využít k akceleraci aplikací. GPGPU je v tomto směru úspěšná, a to zejména pro výpočty náročných úloh. Bylo by však chybou ji považovat za univerzální řešení. Proto je vhodné sledovat i další moderní alternativy a já jsem zvědav, jakým způsobem se bude dále vyvíjet, i v této práci zmíněná, vize společnosti Intel a jejich mnohojádrových procesorů Intel Xeon Phi.

Literatura

[1] Ali, A.; Syed, K. S.: An Outlook of High Performance Computing Infrastructures for Scientific Computing. *Advances in Computers*, ročník 91, 2013: s. 87–118.
URL http://dx.doi.org/10.1016/B978-0-12-408089-8.00003-3

[2] Darema, F.: The SPMD Model. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2001-9-11: str. 1.
URL http://link.springer.com/10.1007/3-540-45417-9_1

[3] Darema, F.; George, D.; Norton, V.; aj.: A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, ročník 7, č. 1, 1988: s. 11–24.
URL http://linkinghub.elsevier.com/retrieve/pii/0167819188900944

[4] Flynn, M. J.: Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, ročník 21, č. 9, 1972: s. 948–960.
URL
http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5009071

[5] Flynn, M. J.; Rudd, K. W.: Parallel architectures. *ACM Computing Surveys*, ročník 28, č. 1: s. 67–70.
URL http://portal.acm.org/citation.cfm?doid=234313.234345

[6] Gaster, B. R.; Howes, L.; Kaeli, D.; aj.: *Heterogeneous computing with OpenCL*. Amsterdam: Morgan Kaufmann, 2012.

[7] Gschwind, M.; Hofstee, H.; Flachs, B.; aj.: Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, ročník 26, č. 2, 2006: s. 10–24.
URL
http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1624323

[8] Karam, L.; Alkamal, I.; Gatherer, A.; aj.: Trends in multicore DSP platforms. *IEEE Signal Processing Magazine*, ročník 26, č. 6, 2009: s. 38–49.
URL
http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5230802

[9] Khronos: The OpenCL Specification [online]. 2014 [cit. 2014-04-16].
URL http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf

[10] Kirk, D.; mei Hwu, W.: *Programming massively parallel processors*. Boston: Morgan Kaufmann, druhé vydání, 2013.

[11] Nelson, M.: Data Compression with the Burrows-Wheeler Transform. *Dr. Dobb's Journal*, , č. 9, 1996.

[12] Nelson, M.; Gailly, J.-L.: *The data compression book*. New York: IDG Books Worldwide, Inc., druhé vydání, 1996.

[13] NVIDIA: CUDA Toolkit Documentation [online]. 2014 [cit. 2014-03-28].
URL http://docs.nvidia.com/cuda

[14] Quinn, M. J.: *Parallel programming in C with MPI and openMP*. Boston: McGraw-Hill, 2004.

[15] Salomon, D.: *Data Compression*. London: Springer, Čtvrté vydání, 2007.

[16] Sayood, K.: *Introduction to data compression*. San Francisco: Morgan Kaufmann Publishers, třetí vydání, 2006.

[17] Tsuchiyama, R.; Nakamura, T.; Iizuka, T.; aj.: *OpenCL programming book*. Sunnyvale: Fixstars, 2012.

[18] Vašíček, Z.: Kódy, 2013.
URL https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/INP-IT/excs/cv3-kody.pdf?cid=8005

Dodatek A

Obsah CD

- readme.txt - popis jednotlivých souborů v příloze na CD.
- src/bwt/ - složka obsahující soubory k Burrowsově-Wheelerově transformaci.
- src/mtf/ - složka obsahující soubory k move-to-front transformaci.
- src/rle/ - složka obsahující soubory k run-length encoding.
- src/ari/ - složka obsahující soubory k aritmetickému kódování.
- doc/bakalarska_prace.pdf - tato technická zpráva ve formátu pdf.
- doc/tex/ - složka obsahující zdrojové soubory technické zprávy v jazyce LaTeX.