VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VYUŽITÍ MODERNÍCH GPU PRO OBECNÉ VÝPOČTY

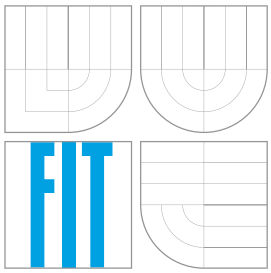
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JOSEF POTĚŠIL

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VYUŽITÍ MODERNÍCH GPU PRO OBECNÉ VÝPOČTY

GENERAL PURPOSE GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JOSEF POTĚŠIL

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. RNDr. PAVEL SMRŽ, Ph.D.

BRNO 2009

Abstrakt

Tato bakalářská práce pojednává o obecných výpočtech na grafických kartách. Konkrétním druhem zkoumaných výpočtů jsou řadící algoritmy. Začátek této práce se věnuje obecně tématice obecných výpočtů, dostupným nástrojům a technologiím. Dále pokračuje základními teoretickými informacemi o řazení. Nakonec popisuje implementaci několika řadících algoritmů v prostředí jazyka CUDA, testy výkonnosti a zhodnocení výsledků testů.

Abstract

This bachelor's thesis deals with the general-purpose computing on graphics processing units. The examined kind of algorithms are the sorting algorithms. The beginning of this thesis deals with the general-purpose thematic itself, available tools and technologies. It continues with the basic theoretical informations about sorting. At the end the implementation of some sorting algorithms in CUDA, performance tests and the evaulation of these tests are described.

Klíčová slova

Obecné výpočty, GPU, GPGPU, CUDA, Řazení.

Keywords

General purpose, GPU, CUDA, Sort.

Citace

Josef Potěšil: Využití moderních GPU pro obecné výpočty, bakalářská práce, Brno, FIT VUT v Brně, 2009

Využití moderních GPU pro obecné výpočty

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana docenta Pavla Smrže.

.....
Josef Potěšil
20. května 2009

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu panu docentu Smržovi za jeho vedení, vstřícnost a usměrňování mé práce.

Obsah

Kapitola 1

Úvod

Grafické karty a jejich jádra (GPU) slouží primárně k vykreslování počítačové grafiky. Moderní grafická jádra dosáhla v posledních několika letech velkého pokroku v navýšení výkonu a programovatelnosti, díky čemuž jsou nyní schopna provádět daleko širší spektrum algoritmů než dříve. Díky svojí masivně paralelní struktuře jsou pro některé úlohy vhodnější a mohou přinést výrazný výkonnostní nárůst. Současným trendem v oblasti vývoje procesorů a výpočetních jednotek je vzrůstající počet těchto jednotek na jeden čip. Důvodem k tomu je také fakt, že výpočetní jednotky nemohou neustále dosahovat vyšších frekvencí kvůli fyzikálním vlastnostem a barierám materiálů, z nichž jsou vyrobeny. Proto není příliš velký důvod předpokládat, že by se trend vyšší paralelizace mohl v příštích letech rychle změnit. Navíc dvou a vícejádrové procesory se stávají součástí běžně dostupných počítačů. Proto je důležité přizpůsobit tomuto trendu používané algoritmy. Řazení je jedním z nejvíce studovaných a nejčastěji se vyskytujících problémů a spousty jiných algoritmů vyžadují pro svou vlastní efektivitu buď seřazená vstupní data nebo kvalitní a rychlé implementace řazení. Proto je důležité mít na každé platformě paralelní řadící algoritmy, které maximálně a efektivně využijí dostupné prostředky.

Cílem této práce je zjistit, zda jsou grafické karty a patřičné řadící algoritmy dostatečné rychlé, aby jimi bylo možné nahradit současná řešení řazení pomocí běžných procesorů (CPU) a pro jaké druhy vstupních dat jsou vhodné.

Kapitola 2 slouží jako úvod do světa obecných výpočtů na grafických kartách, popisuje zkoumané oblasti, základní architekturu moderních grafických jader společnosti nVidia a dostupné programové prostředky, které umožňují psaní aplikací pro GPU.

Základní teorii obecně okolo řazení se věnuje kapitola 3. Zahrnuje klasifikaci řadících algoritmů, jejich vlastnosti a nakonec se věnuje popisu, vlastnostem a principům několika základních verzí řadících algoritmů, které byly použity pro běh na grafických kartách.

V kapitole 4 jsou blíže popsány implementace jednotlivých řadících algoritmů v jazyce CUDA.

V předposlední kapitole 5 jsou uvedeny testovací metodiky a samotné testy implementovaných algoritmů spolu s jejich hodnocením.

Závěrečná kapitola 6 shrnuje dosažené výsledky této práce.

Kapitola 2

GPGPU

Tato kapitola slouží jako úvod do světa GPGPU (General-purpose computing on Graphical processing units) neboli provádění obecných výpočtů pomocí výpočetních jednotech grafických jader, obeznámení se základními termíny, oblastmi využití a dostupnými nástroji.

2.1 ATI Stream, OpenCL

ATI Stream je programové rozhraní vyvíjené společností ATI, respektive AMD. Má za sebou poměrně zajímavý vývoj. Dříve stavěla hlavně na technologii „Close to Metal“, což je nízkoúrovňové programovací rozhraní. Dávalo přímý přístup k množině nativních instrukcí a paměti na kartách generace ATI Radeon X1950 (jádro R580) a novějších. Novější verze staví na OpenCL.

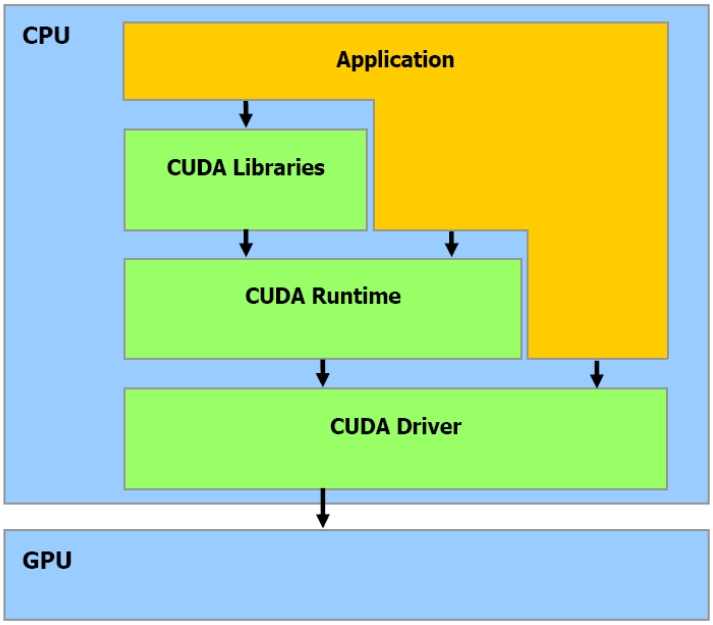
OpenCL je první otevřený standard pro paralelní programování a obecné výpočty nejen na grafickém hardware. Poskytuje jednotné programovací rozhraní pro vývojáře software běžících na klasických procesorech, grafických kartách, Cell procesoru a jiných paralelních architekturách. [12] OpenCL specifikace byla navržena skupinou The Khronos ^{TM}Group. Na této specifikaci se podílela většina významných hráčů na poli informačních technologií, např. Intel, AMD, nVidia, IBM, Sun, Samsung, Sony a spousta dalších. OpenCL jako takové je na jazyku C založené multiplatformní, nízkoúrovňové programové rozhraní zaměřené spíše pro více pokročilé a zkušené vývojáře. Jazyk je založen na standardu ISO C99 včetně standardních hlaviček a preprocesorových direktiv, obsahuje navíc vestavěné skalární i vektorové datové typy, množinu základních funkcí, kterou musí každá implementace mít a množinu speciálních funkcí, které nejsou nutné pro všechny platformy. Jazyk C99 je ovšem omezen o určité konstrukce (např. ukazatele na ukazatele nebo na funkce), ale na druhou stranu většina těchto omezení je vyvážena dalšími rozšířeními. OpenCL na rozdíl od CUDA, zmíněného níže, není omezen pouze na provoz na jednom zařízení, ale danou úlohu mohou počítat všechna zařízení, která jsou v počítači k dispozici. Jelikož hlavní hráči ohlásili v následujících měsících dostupné implementace včetně nVidie, domnívám se, že samotné rozhraní OpenCL by mohlo v budoucnu hrát větší roli než CUDA, přestože má CUDA momentálně lepší pozici a i programátoři si na dané rozhraní pomalu zvykli.

2.2 NVIDIA®CUDA™

CUDA je programový balík vyvinutý společností NVIDIA Corporation pro běh a vývoj aplikací a algoritmů počítaných grafickými jádry. Nepoužívá tedy jako dříve grafická rozhraní

DirectX ani OpenGL pro jeho přímé naprogramování, ale umí s těmito rozhraními spolupracovat. Byla vypuštěna v březnu roku 2007. CUDA je akronymem pro anglické „*Compute Unified Device Architecture*“. Jak název napovídá, není tato technologie určená pouze na omezené množství produktů, nýbrž pro všechny grafické karty této společnosti generace jádra G80 a novějších. Je tedy možné provozovat tyto výpočty i na méně výkonných a běžně dostupných kartách. Celý tento balík se skládá ze tří hlavních komponent:

- Grafická karta společnosti NVIDIA s patřičnou verzí ovladače. Jak již bylo uvedeno výše, pro skutečné využití je potřeba grafických jader G80 a novějších, tedy karet GeForce řady 8000, včetně mobilní řady GeForce Mobile a profesionální řady Quadro a Tesla. Nicméně není nezbytně nutné mít ve svém počítači tento hardware pro účely vývoje a testování. Balík umožňuje emulaci tohoto zařízení, takže prakticky každý si toto prostředí může vyzkoušet, včetně spuštění a ověření funkčnosti samotných výpočtů. Samozřejmě nemůžeme očekávat žádné urychlení, když procesor tyto výpočty emuluje.
- CUDA Toolkit obsahuje na jazyku C založené vývojové prostředí. Obsahuje překladač *nvcc*, hlavičkové soubory, knihovny pro výpočet rychlé Fourierovi transformace *CUFFT* a základní matematickou knihovnu lineární algebry *CUBLAS* („Basic Linear Algebra Subroutines“).
- CUDA SDK (software development kit) je poslední součástí, jedná se o názorné příklady různých algoritmů.



Obrázek 2.1: Jednodušší nákres architektury CUDA, převzato z [3]

2.2.1 Architektura

Jádro grafického procesoru je postaveno okolo pole multiprocesorů. Tyto multiprocesory se skládají z 8 skalárních procesorů, které jsou schopny vykonávat velké množství vláken

konkurentně. Vlákna jsou organizována do bloků maximálně po 512 (u nejnovější generace 1024). Tento blok vláken musí být zvolen tak, aby bylo možné jejich kód provádět samostatně, nesmí tedy záležet na pořadí vykonávání bloků. Tato podmínka zajistí vnitřnímu plánovači schopnost správně organizovat práci a zajistit maximální vytížení multiprocesorů, na kterých se blok spouští. Bloky vláken je ještě dále možno organizovat do *grid* jednotek, které mohou obsahovat více bloků vláken (pole bloků vláken). Pokud je tedy z programu zavoláno vykonávání nějaké úlohy o specifikovaném počtu grid jednotek, jsou dále očíslovány a je jich současně spuštěno tolik, kolik multiprocesorů přítomná grafická karta obsahuje. V rámci multiprocesoru vytvoření, správa a paralelní vykonání vláken nestojí žádnou režii. Poté, co jeden multiprocesor dokončí svou práci, vezme další grid a ten vykoná. Toto umožňuje dobrou škálovatelnost a paralelismus do budoucna, kdy zůstane tentýž kód, ale grid jednotky budou vykonávány na více multiprocesorech současně. [8]

2.2.2 Paměťový model

Vlákna mohou přistupovat k více druhům paměti během vykonávání svého kódu. Globální paměť není nijak optimalizovaná, nachází se v hlavní paměti karty. Přístup k ní trvá přibližně 400–600 cyklů. Podobně je na tom lokální paměť každého multiprocesoru. Ty se používají pro automatické proměnné takové velikosti, která se nevejde do registrů nebo by zabíraly příliš místa, jako pole struktur. Často jsou sem umístěna pole, u nichž překladač není schopen určit jejich velikost v době kompilace.

Naproti tomu paměť pro textury a konstanty je optimalizována, jde spíše o rychlou vyrovnávací paměť. Pokud čtená data nejsou v této paměti, bude tato operace stát stejně, jako jeden přístup ke globální paměti. Jinak je přístup sem stejně rychlý pro všechna vlákna jako do registru, pokud čtou všechna vlákna ze stejné adresy, v případě čtení více adres roste počet přístupů lineárně. Paměť textur se však pro GPGPU aplikace příliš nepoužívá. Velikost těchto pamětí je $8KB$ na jeden multiprocesor. Životnost globální paměti a paměti pro textury je po celou dobu běhu aplikace.

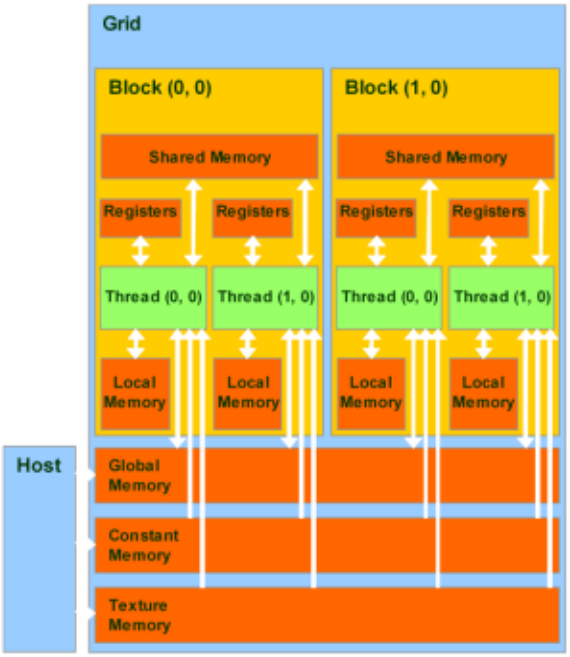
Za normálních okolností přístup k registrům nestojí žádné instrukce navíc. Může však docházet ke konfliktům v přístupu k paměťovým bankách registrů při jejich čtení následovaném ihned po zápisu hodnoty. K zakrytí tohoto problému pomáhá dostatečné množství aktivních vláken. Každý multiprocesor obsahuje 8192 32-bitových registrů, u nejnovějších generace jader se tento počet zdvojnásobil.

Sdílená paměť o velikosti $16KB$ je stejně jako registry přímo vestavěna do multiprocesoru a jedná se o obdobu L1 vyrovnávací paměti u procesorů. Čtení i zápis zabere pouze 4 strojové cykly. Aby byl tento prostor efektivně využíván, je rovnoměrně rozdělen do paměťových bank o velikosti 32 bitů. Pokud je tedy žádán přístup k různým datům uvnitř této paměti, jsou požadavky obslouženy současně. Optimální využití této paměti je nastíněno v sekci 2.2.4.

2.2.3 Toolkit a jazyk CUDA

Aplikace napsané v jazycích C/C++ mohou rozhraní CUDA využívat přímo. Aplikace zapsané v jiných vysokoúrovňových programovacích jazycích musí využívat jiné metody přístupu. Existuje několik více či méně oficiálně podporovaných projektů, které programátorům umožňují jednodušší přístup. Tyto projekty existují pro programovací jazyky *Fortran*, *Java* (JaCuda), *Python* (PyCUDA), *.NET* (CUDA.NET).

Základní schéma samotného překladu programu probíhá pomocí překladače *nvcc* tak,



Obrázek 2.2: Paměťová hierarchie, převzato z [1]

že CUDA kód je převeden do ANSI C zdrojových souborů, které jsou poté zpracovány hostitelských C/C++ překladačem. Fáze překladu vypadají zhruba takto [7]:

- Vstupní soubory jsou rozděleny pomocí CUDA front end podle jejich přípon.
- Zdrojové soubory určené k běhu na GPU jsou zpracovány pomocí *nvcc*. Podle přepínačemi zvoleného módu pro *nvcc* jsou soubory buď přeloženy do spustitelné formy pomocí assembleru a/nebo meziproduktu *ptx* kódu. Z něj je potom vyrobena jistá forma ukazatele na funkce, které se budou volat z *CUDA runtime* knihovny za běhu programu.
- Zbylé soubory přeloží C/C++ překladač na daném počítači.
- C/C++ překladač sloučí všechny svoje meziprodukty a výsledky činnosti *nvcc* do výsledného spustitelného souboru.

2.2.4 Zásady psaní CUDA aplikací

Jednou z důležitých zásad je optimální využití dostupných typů pamětí. Potenciálně velký výkon by mohl být snadno snížen například neustálým přístupem do hlavní paměti. Typickým vývojovým vzorem je použití následujícího postupu:

1. Nahrát data z globální paměti grafické karty do sdílené paměti jednotlivých multiprocesorů.
2. Synchronizovat všechna vlákna v rámci bloku, aby všechna vlákna měla bezpečný přístup ke čtení prvků zapsaných jinými vlákny.
3. Vykonat všechny operace ve sdílené paměti.

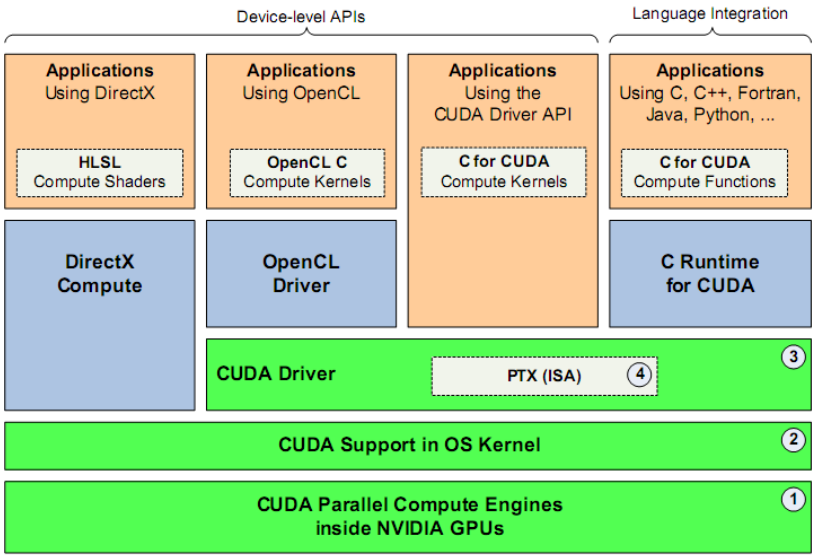
4. Opět synchronizovat všechna vlákna kvůli potřebě zajištění správného uložení výsledků.

5. Zapsat data zpět do globální paměti grafické karty, případně dále do paměti počítače RAM.

Přístup ke globální paměti není nijak optimalizován, respektive není zde žádná vyrovnávací paměť. Jedním ze dvou důležitých kroků je využívat datové struktury, které jsou správným způsobem zarovnané. Globální paměť je schopna jedinou instrukcí načtení 32-bitového, 64-bitového nebo 128-bitového slova do registrů. Proto by se měly pokud možno využívat struktury zarovnané na tuto velikost 4, 8 nebo 16 bytů, to platí například pro vestavěné datové typy jako **float2** nebo **float4**. Pro uživatelské datové typy je možné použít explicitně specifikaci kompilátoru **__align__(n)**. Pokud jsou datové struktury větší, měl by být opět použit tento specifikátor **__align__(16)** s tím, že kompilátor použije potřebný počet instrukcí, takže například 2 128-bitové instrukce.

V sekci 2.2.1 byla zmíněna organizace vláken. Ty je možno ještě dále dělit po 32 do *warp* jednotek. Jeden skalární procesor vykonává instrukce *warp* jednotky. Aby byla zajištěna maximální efektivita, měly by instrukce vláken být stejné. Poté procesor jednou instrukcí obslouží kód všech vláken. Pokud ve zpracování dojde ke změně instrukcí jednotlivých vláken, například vyskytne-li se podmínka v algoritmu, rozdělí vlákna podle jejich průběhu do skupin, provede je sériově a poté opět začne provádět kód vláken všech vláken jako dřív. Programátor může toto chování ignorovat, ale bylo by vhodné mít tuto věc na paměti.

Nyní zpět k paměti. Její propustnosti je nejoptimálnější, pokud je možné současný přístup vláken k datům zřetězit. Pro polovinu *warp* vláken generuje překladač čtecí instrukci potřebné velikosti v závislosti na velikosti čteného datového typu. Pro výkon je také důležité, aby data čtená jednou instrukcí ležela zarovnaně v jednom segmentu paměti.



Obrázek 2.3: Podrobnější nástin architektury CUDA, obrázek převzat z [9]

Vysvětlivky k obrázku 2.3:

1. Paralelní výpočetní stroj GPU

2. Podpora jádra operačního systému ke konfiguraci a inicializace
3. Ovladač grafické karty s programovým rozhraním
4. Soubor instrukcí pro funkce jádra

Posledním bodem je také skutečnost, že vlákno s určitým indexem v daném bloku vláken by mělo přistupovat ke stejnému indexu paměti, čili třetí vlákno k třetímu prvku načtených dat. Pokud nejsou tyto podmínky splněny, přistupuje každé vlákno k prvkům v globální paměti samostatně a tím rapidně klesá výkonnost.

2.3 Oblasti využití

- Zpracování obrazu je jednou z oblastí hlavního uplatnění programového prostředí CUDA, které se začalo v poslední době velmi rozšiřovat. Tato oblast je asi pro běžného uživatele nejznámější.
  - Programy pro editaci, kódování a přehrávání videa. To znamená akcelerování různých filtrů vylepšující výsledný obraz, samotný převod mezi formáty, nejčastěji do formátu H.264 s vysokým rozlišením. Zástupci jsou například PowerDirector7 Ultra, TMPGEnc 4.0 XPress nebo Badaboom od samotné NVIDIE a dosahují přibližně 3.5 až 4.5-násobného zrychlení.
  - Ray tracing, metoda pro vytváření obrazu pomocí sledování paprsku světla z každého obrazového bodu do scény, je též často zkoumaným algoritmem. Paralelizace takové úlohy je vlastně přirozená, jedno vlákno počítá trasu paprsku z jednoho obrazového bodu (pixelu). Provoz této aplikace na grafické kartě oproti běžnému procesoru může přinést až 16-násobné urychlení.
  - Další aplikací v oblasti zpracování obrazu je zpracování snímku černobílých mikrofilmů pořízených při skenování v reálném čase, typicky noviny, lékařské zprávy. V obraze se odstraní šum a možný výskyt artefaktů v pozadí, zvýší se kvalita písma beze ztráty detailů. Zrychlení přibližně dvacetinásobné.
  - GpuCV je otevřená knihovna pro zpracování obrazu. Jejím cílem je převést co největší množství algoritmů z knihovny OpenCV do svojí za účelemen zrychlení výpočtů při zachování stejného programového rozhraní.
- Oblast financí je jednou z disciplín spoléhající na obory jako počítačová inteligence, numerické metody, simulace pro zjišťování rizik spojenými s investičními a obchodními rozhodnutími finančních institutů. Počítají se lineární, diferenciální rovnice, pravděpodobnost a další. Grafické karty jsou pro tuto oblast výpočtů velmi vhodné, jelikož se vlastně jedná o matematický koprocesor a prováděné výpočty jsou většinou v plovoucí desetinné čárce, tudíž zde vynikne jejich daleko větší výkon. To umožňuje bankám nahradit stávající sestavy, složené například ze stovky procesorů, jedním počítačem o dvou grafických kartách. Tím si zachovají stejný výkon, ušetří místo a prostředky na energii a chlazení. Pro srovnání by bylo vhodné zmínit, že poslední generace grafických jader dosahují výkonnosti okolo 1000 GFLOPS (miliard operací v plovoucí desetinné čárce), kdežto běžné procesory kolem 40 GFLOPS.

- Bioinformatika
  - Folding@home je relativně velmi známý distribuovaný projekt Stanfordské univerzity, jehož cílem je výzkum a léčba Alzheimerovy, Parkinsonovy, Huntingtonovy choroby a velkého množství různých druhů rakoviny. Poté, co byli představeni klienti využívající výkon GPU, výrazně vzrostl jejich podíl na celkovém výkonu a dnes tvoří hlavní výpočetní sílu, viz obrázek 2.4.

OS Type	Native TFLOPS*	x86 TFLOPS*	Active CPUs	Total CPUs
Windows	304	304	319970	2640318
Mac OS X/PowerPC	5	5	6525	126747
Mac OS X/Intel	24	24	7608	85528
Linux	48	48	28040	378801
ATI GPU	995	1050	9759	56359
NVIDIA GPU	2294	4840	19279	100765
PLAYSTATION®3	1097	2315	38891	783587
Total	4767	8586	430072	4172105

Obrázek 2.4: Statistika podílu jednotlivých platforem na celkovém výkonu ke dni 2. května 2009

  - Z dalších projektů se například 200x urychluje sledování bílých krvinek pod mikroskopem nebo magnetická rezonance, vyhledávání stejných řetězců, neurální sítě a další.

- Simulace jsou další významnou oblastí, která se snaží využít vysokému numerického výkonu dnešních karet. I v samotném SDK jsou uvedeny příklady pro generování pseudonáhodných čísel, například pomocí algoritmu *Mersenne Twister*. Dále je zde ukázka simulační metody *Monte Carlo*. Trojrozměrné simulace pohybu částic nebo tekutin v reálném čase bývá využívána nejen v automobilovém průmyslu pro simulaci toku vzduchu kolem objektů nebo i z bioinformatiky pohyby částic na molekulární úrovni. V energetice se simulují rozsáhlé energetické systémy, dále zemětřesení, řeší se Boltzmannovy rovnice.
- Řešení dvou nebo trojrozměrných Eulerových rovnic, rozšíření pro Matlab
- Rozpoznávání řeči, zpracování signálů
- Grafové algoritmy
- Neuronové sítě
- Kryptografie
- Iterativní lineární systémy
- Geografické informační systémy, databáze
- Rozpoznávání objektů v obrazu
- Předpovědi počasí a další

Kapitola 3

Řazení

Řazení zajišťuje seřazení vstupních dat podle pořadí. Nejčastěji se řadí podle numerické hodnoty nebo abecedně. Řazení je jeden z nejstarších a nejčastěji řešených algoritmických problémů v počítačové oblasti. Velké množství algoritmů předpokládá pro svoji vlastní efektivitu rychlé a efektivní algoritmy pro řazení, případně seřazená vstupní data.

Nejprve se v této kapitole seznámíme s terminologií, připomeneme základní vlastnosti tohoto druhu algoritmů a jejich hlavní principy. Nakonec zde budou objasněny algoritmy, které byly použity k implementaci řazení na grafických kartách.

3.1 Definice problému

Na vstupu je posloupnost $S = (S_1, S_2, \ldots, S_n)$; cílem je najít takovou posloupnost $S' = (S'_1, S'_2, \ldots, S'_n)$, pro kterou platí dvě základní kritéria:

1. Tato posloupnost je seřazená:
   $(S'_1 \leq S'_2 \leq \ldots \leq S'_n)$.
2. Posloupnost S' je permutací původní posloupnosti S (obsahuje tedy stejná data, jen v jiném pořadí).

V definici relace uspořádání $\leq$ se přitom bere ohled pouze na klíče příslušných hodnot. [10] Klíčem se rozumí vlastnost prvku posloupnosti, která určuje pozici prvku v uspořádané množině.

3.2 Terminologie

V této části bych chtěl zmínit a objasnit základní pojmy jako řazení, třídění. Tyto pojmy jsou pro běžného člověka prakticky stejné, ale jsou zde určité rozdíly.

- **Třídění** (anglicky *sort, sorting)* je rozdělení položek do skupin se stejnými vlastnostmi podle klíče. Může být dané množina prvků různých typů a my je podle tohoto roztřídíme.
- **Uspořádání podle klíčů** (anglicky *collating)* je seřazení položek podle uspořádané množiny klíčů.
- **Řazení** (anglicky *sequencing)* je uspořádání položek podle relace lineárního uspořádání.

- **Slučování** (anglicky *coalescing)* je vytváření souboru položek sjednocením několika souborů stejného typu.
- **Setřiďování** (anglicky *merging)* je vytváření souboru položek sjednocením několika již seřazených souborů stejného typu.

V této práci se nadále bude pod pojmem řazení rozumět stejný význam jako anglicky sort.

3.3 Vlastnosti algoritmů

Zde budou uvedeny základní vlastnosti algoritmů, o kterých můžeme často slyšet.

- **Stabilita** označuje vlastnost algoritmu, která bere ohled na původní pořadí klíčů se stejnou hodnotou a toto pořadí ve výsledném poli zachová.
- **Přirozenost** určuje, že doba potřebná pro seřazení již seřazeného pole je menší než doba, která je potřebná k seřazení náhodně uspořádaného pole a ta je menší než doba seřazení obráceně seřazeného pole.
- **Dodatečná paměť** specifikuje, zda algoritmus ke své činnosti potřebuje nějakou dodatečnou pracovní paměť navíc. Pokud ji algoritmus nepotřebuje nebo velikost není závislá na množství vstupních dat, říkáme, že pracuje na místě (anglicky in situ) a označují se $O(1)$. Některé algoritmy naopak potřebují svůj výsledek uložit do druhého, stejně velkého pole ($O(N)$).
- **Přístup k paměti** označuje umístění dat. Metody *vnitřního řazení* se označují jako řazení polí a předpokládají data v operační paměti a přímý přístup k prvkům. Metody *vnějšího řazení*, označované jako řazení souborů nebo seznamů, předpokládájí sekvenční přístup k prvkům.
- **Typ procesoru** určuje, kolik operací může probíhat současně. U jednoho procesoru může probíhat jedna operace a následující začne až po skončení předchozí. Hovoříme o *sériovém* nebo-li *sekvenčním* řazení. *Paralelní* řazení využívá více procesorů a běží tedy více úloh současně.

3.4 Složitost

Složitost je jednou z důležitých vlastností, které slouží k porovnávání efektivity a výkonnosti dvou algoritmů. Jako kritéria se berou **čas** a **paměťový prostor**, tedy časová a paměťová složitost. Tato složitost je vždy závislá na vstupních datech, proto má většinou podobu funkce. Pro porovnání se nejčastěji používá **asymptotická časová složitost**. Ta má tři podoby:

- Horní hranice O (*Omicron*) vyjadřuje chování pro nejhorší možnou vstupní množinu dat. Toto hodnocení se používá nejčastěji. Určí největší počet potřebných kroků k vykonání algoritmu, nicméně většinou bude tento počet menší, nikdy však vyšší. Nejlepší řadící algoritmy dosahují horní hranice $\Omega = O(n \log n)$. Lze též dokázat, že tato složitost u řadících algoritmů, založených na principu porovnání dvou klíčů, je nejlepší možná.

- Dolní hranice Ω (*Omega*) vyjadřuje jakési nejlepší možné chování. Tohoto chování se nedosahuje příliš často, spíše jen ve výjimečných a ideálních případech.
- Θ (*Theta*) označuje třídu chování, což je skupina algoritmů se stejnou složitostí, například logaritmická, kvadratická.

3.5 Principy řazení

- **Princip výběru** (anglicky *selection*) je asi nejpřirozenější a nejjednodušší metoda. Postupně přesouvá nejmenší nebo největší prvek vstupní posloupnosti do výstupní.
- **Princip vkládání** (anglicky *insertion*) postupně zařazuje prvky vstupní posloupnosti na patřičné místo seřazené výstupní posloupnosti.
- **Princip rozdělování** (anglicky *partition*) rozdělí posloupnost na dvě části takovým způsobem, aby prvky první části posloupnosti byly menší než prvky druhé části.
- **Princip setřídění** (anglicky *merging*) nebo také slučování spojuje již seřazené posloupnosti do větších seřazených posloupností.

Toto byly principy, které se uplatňují nejčastěji. Existují další varianty, které většinou vhodně kombinují výše jmenované.

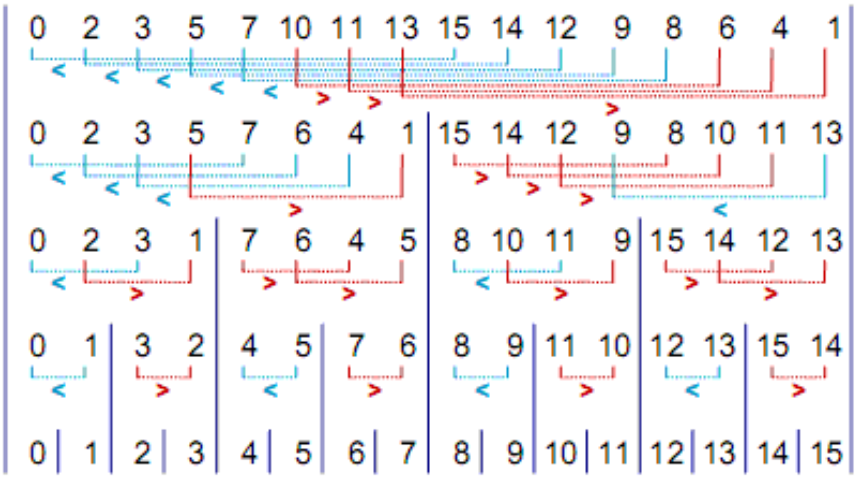
3.6 Bitonic sort

3.6.1 Vlastnosti

Tento řadící algoritmus byl speciálně navržen pro běh na paralelních systémech. Je nestabilní, nepřirozený, pracuje *in situ*, nepotřebuje tedy dodatečné pole a jeho časová složitost $O = (n \log^2 n)/p$, kde n představuje počet prvků řazeného pole a p počet procesorů k tomu použitých.

3.6.2 Popis algoritmu

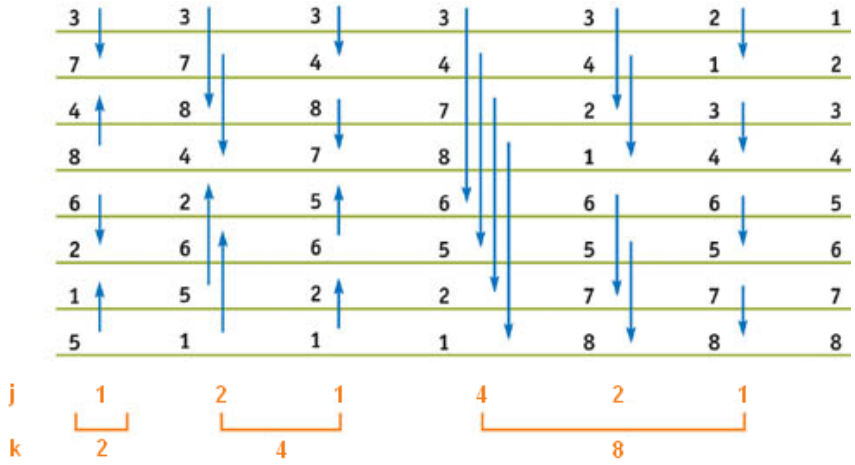
V první fázi algoritmus vytváří bitonické sekvence, které posléze setřídí. Bitonickou sekvencí se rozumí spojení dvou dalších sekvencí, jedné monotónně rostoucí (anglicky *asceding*) a druhé klesající (anglicky *descading*). Tyto sekvence musí být stejně dlouhé. Vždy se řadí vstupní posloupnosti o takové velikosti, která je mocninou čísla dvě, aby bylo možné tyto posloupnosti dále dělit na dvě poloviny. Algoritmus může vytvořit několik bitonických sekvencí. Na ně se posléze uplatňuje bitonické setřídění (anglicky *bitonic merge*). To v první fázi spojí posloupnosti tak, že v horní půlce jsou všechna čísla větší než ta ve spodní půlce. Dále se rekurzivně volá setřiďování na posloupnosti poloviční velikosti, dokud celá posloupnost není seřazená. Na obrázku 3.1 si ukážeme základní princip algoritmu. Řazena je bitonická sekvence o velikosti 16 prvků. Nejprve se porovnává první prvek první půlky sekvence s prvním prvkem druhé půlky, respektive prvky s odpovídajícími si indexy v jednotlivých částech, čili 1 a 9, 2 a 10, atd. Prakticky vzdálenost jednotlivých prvků je v prvním kroku rovna polovině celé délky, tedy 8. Pokud je prvek v první polovině větší než jeho odpovídající protějšek, jednoduše se tyto prvky prohodí. Tím se tedy přesunou všechna menší čísla do spodní poloviny a větší čísla do horní poloviny. Poté se snižuje tato vzdálenost na



Obrázek 3.1: Bitonické setřídění, převzato z [4]

polovinu, čímž nám vzniká více menších posloupností tak dlouho, dokud se neporovnávají sousední prvky a poté je již celá posloupnost seřazená.

Vytváření bitonických sekvencí vypadá podobně s tím rozdílem, že se nejprve porovnávají sousední prvky a střídavě se seřazují vzestupně a sestupně. Tím vzniknou bitonické posloupnosti o délce 4. Tyto posloupnosti dále spojujeme stejným způsobem, jak je vysvětleno výše, pouze pokud se nacházíme v první půlce, řadíme vzestupně a v druhé půlce sestupně. Určení operátoru porovnání se určuje operací logický součin (AND) velikosti kroku a indexu prvku.



Obrázek 3.2: Ukázka průběhu celého algoritmu bitonic sort, převzat a upraven z [6]

3.7 Quicksort

3.7.1 Vlastnosti

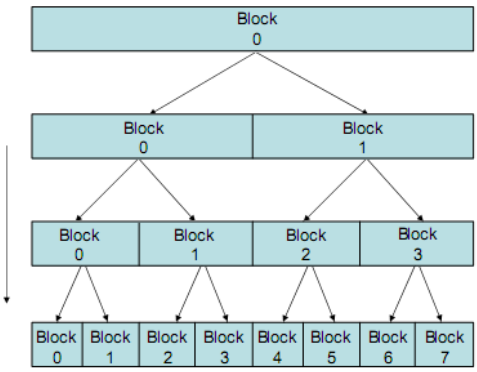
Quicksort je jeden z nejpoužívanějších a nejrychlejších řadících algoritmů. Pracuje na principu rozdělování. Algoritmus není stabilní, nechová se přirozeně a pracuje v „in situ“. Dolní

hranice a průměrná asymptotická časová složitost je $O(n \log n)$ a horní hranice $O(n^2)$. Této hranice se nicméně dá v praxi většinou vyhnout.

3.7.2 Popis algoritmu

Algoritmus nejprve vybere ze vstupní posloupnosti tzv. „pivota“. Ten by měl v ideálním případě představovat medián řazené části pole. Tyto posloupnosti se rozdělí tak, aby hodnoty prvků vlevo od pivota byly menší než hodnoty prvků vpravo. Tento princip se rekurzivně uplatňuje na nově vzniklá, ideálně poloviční, pole a volá se tak dlouho, dokud není velikost řazené posloupnosti rovna jedné. Poté již bude posloupnost seřazena. Hlavním problémem je tedy volba pivota. Nejlepší volbou by byl medián. Jeho nalezení je však poměrně drahé, musí se projít v daném kroku všechny prvky řazeného pole nebo se použije jiná metoda pro jeho přibližný výpočet. Nejčastěji používanou metodou je volba náhodného pivota.

Paralelizace tohoto algoritmu je relativně snadná, ale má také své komplikace. Na obrázku 3.3 je to vcelku patrné. Nejprve se musí vstupní posloupnost rozdělit do více částí, poté bude dostatečný počet bloků pro každý multiprocesor, jež se budou řadit nezávisle a budou ideálně takové velikosti, aby by se vešly do rychlé sdílené paměti. Problém je tedy na začátku algoritmu, postup je přesně opačný než u bitonic sortu. Možný způsob řešení je nastíněn v kapitole 4.2.



Obrázek 3.3: Rozdělování vstupu s ideální volbou pivota, převzat a upraven z [5]

3.8 Radix sort

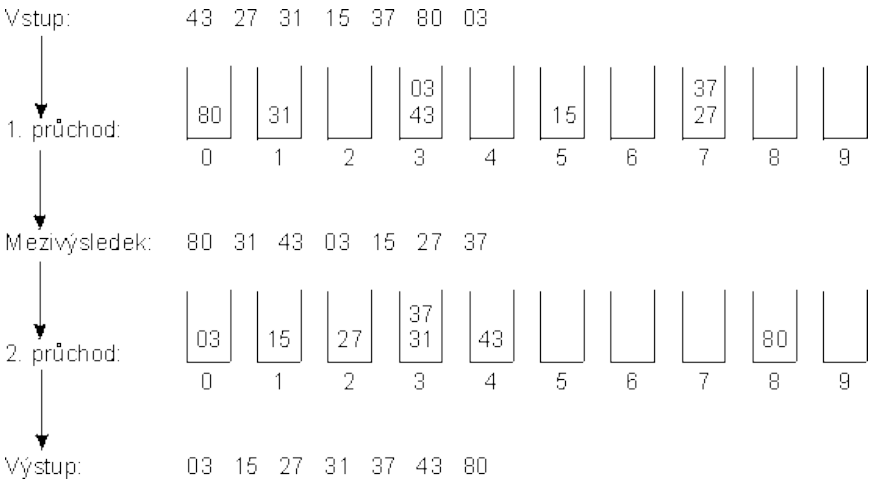
3.8.1 Vlastnosti

Radix sort je jeden z nejstarších a dobře známých algoritmů. Radix sort předpokládá, že klíčem je několika-bitové číslo. Protože čísla mohou představovat hodnoty znaků řetězců nebo vhodně formátované desetinné číslo, není algoritmus omezen pouze k řazení čísel. Je velmi efektivní pro řazení malých klíčů. Složitost je lineární $O(nk)$, závisí na velikosti vstupních dat n a délce klíče k, například počet bitů reprezentující číslo. Principiálně je stabilní, nepřirozený a nepracuje „in situ“. Jeho implementace však nemusí být stabilní a nemusí vyžadovat dodatečnou paměť. Existují dvě varianty algoritmu, které stabilitu ovlivňují. LSD (anglicky *Least significant digit*) bere v úvahu nejprve nejméně významné číslice nebo bity, což jej dělá stabilním a MSD (*Most significant digit*) třídí podle nejvíce významných číslic respektive bitů.

3.8.2 Popis algoritmu

Základní verze LSD varianty pracuje podle obrázku 3.4 následujícím způsobem. V prvním průchodu roztřídí posloupnost podle nejméně významného řádu nebo bitu do přehrádek. Ty následně spojí například do seznamu a ten opět roztřídí, tentokrát podle dalšího řádu čísla. Počet těchto opakování závisí na počtu řádů nebo délce bitové reprezentace čísla, proto lineární složitost.

MSD varianta v prvním průchodu roztřídí vstupní posloupnost podle nejvýznamnější číslice do přehrádek a tyto přehrádky rekurzivně stejným způsobem roztřiďuje s řádem o jeden nižší. Výsledné přehrádky pak stačí jen spojit ve správném pořadí.



Obrázek 3.4: Radixsort, ilustrace převzata z [11]

Kapitola 4

Implementace

V této kapitole budou popsány konkrétní implementace testovaných algoritmů, s hlavním ohledem na samotný algoritmus, začlenění výsledné implementace do cizích program a možné (nutné) úpravy.

4.1 Bitonic sort

Celý algoritmus bitonického řazení je implementován jako šablonové funkce, aby fungoval pokud možno s libovolnými datovými typy. Tyto typy jsou pouze omezeny na nutnost mít definovány operátory porovnání menší než $<$ a větší než $>$ a operátor přiřazení. Dále je zde určitý problém s velikostí datových typů, ale o tom až později. Celý algoritmus je uzavřen do jmenného prostoru *bitonic_sorter*.

Samotný řadící algoritmus se skládá ze tří fází. První rozdělí vstupní pole do bloků pevné velikosti, které jsou přiděleny jednotlivým multiprocesorům grafického čipu. Ty následně v těchto blocích vytvoří bitonické posloupnosti. Na začátku se podle zásad uvedených v sekci 2.2.4 překopírují data z globální paměti do sdílené paměti multiprocesoru. Vlákna dále pracují s touto rychlou pamětí. V současné implementaci zpracovává každé vlákno čtyři prvky vstupní posloupnosti. Po vytvoření bitonické posloupnosti jsou data překopírována zpět do globální paměti.

V další fázi algoritmu jsou již porovnávané prvky od sebe příliš vzdáleny, docházelo by při spojování bitonických sekvencí při přístupu ke globální paměti za účelem přenosu do sdílené paměti k nesekvenčnímu přístupu, tudíž je využití sdílené paměti vynecháno a s prvky se při porovnání a následném možném přehození pracuje přímo v globální paměti. Pokusy s využitím sdílené paměti v této fázi byly kontraproduktivní, přinesly zpomalení celého řazení. Jakmile klesne vzdálenost jednotlivých prvků na takovou úroveň, že je možno celý blok opět vměstnat do sdílené paměti, je toho využito, což se provádí v třetí fázi.

V úvodní části této podkapitoly byl zmíněn problém s velikostí datových typů. Jde o to, aby se prvky bloku, s nímž se pracuje, daly všechny přesunout do sdílené paměti a zachoval se dostatečný počet současně pracujících vláken. Pro datové typy do velikosti 8 bytů toto není problém. Pokud by se potřebovali řadit větší datové struktury, existovalo by několik možných řešení. Nejjednodušší by bylo snížit ve zdrojovém kódu počet současně probíhajících vláken v rámci bloku. Tím by však došlo k drastickému poklesu výkonu, proto tuto možnost nedoporučuji. Dalším řešením by bylo upravit nebo vytvořit novou datovou strukturu tak, aby obsahovala klíč například typu float, podle něhož by se řadilo a k němu ukazatel na zbytek struktury. To by za předpokladu 32 bitových ukazatelů mělo

fungovat. Posledním řešením by byla úprava samotného zdrojového kódu tak, aby každé vlákno nezpracovávalo čtyři prvky, ale menší počet i jeden prvek na vlákno.

4.2 Quicksort

Tato část je věnována popisu implementace algoritmu Quicksort převzatého z [2] za účelem porovnání výkonu. Algoritmus je rozdělen na dvě hlavní fáze.

V té první se řeší efektivní rozdělení celé posloupnosti mezi jednotlivé multiprocesory. Na vstupu je neseřazené pole, vybere se *pivot*, na levou stranu se přesunou všechny prvky, které jsou menší než pivot a vpravo budou umístěny všechny větší prvky. Toto musí rekurzivně probíhat tak dlouho, dokud není dostatečný počet takto vzniklých částí. Velikost těchto částí z výkonnostních důvodů neklesá pod určitou hranici. V takovém případě algoritmus přepne na řazení pomocí algoritmu *bitonic sort*, kvůli režii způsobenou rozdělováním. Tato hranice je stanovena tak, aby se prvky této délky vešly do rychlé lokální paměti. Toto chování se převážně vyskytuje v druhé části algoritmu. Po zvolení prvního pivota jsou přiděleny všem blokům vláken stejně velké části. Vlákna prochází prvky posloupnosti a počítají, kolik prvků je větších a kolik měnších než pivot. Následně je spočtena suma pro pole menších a větších čísel než pivot. Poté vlákna již ví, kam zapsat výsledná data do pomocné vyrovnávací paměti. Pozice pivota ve výsledné posloupnosti je v tuto dobu již známa a uložena. Pokud je některá ze dvou vzniklých posloupností větší než stanovená hranice, je dále rekurzivně dělena nebo pokud je již k dispozici dostatečný počet sekvencí, přechází se na druhou fázi.

Zde má každý multiprocesor dostatek dat, používá se lokální quicksort, je použit vlastní zásobník pro rekurzi, při níž se nejprve zpracovává nejmenší posloupnost.

Bohužel dostupná implementace, která je momentálně k dispozici (verze 1.1), podporuje pouze řazení celých nezáporných čísel. V jejím zdrojovém kódu se uvádí, že má potíže s překladačem verze 2.0, ale do budoucna by tento problém mohl být částečně odstraněn.

4.3 Radix sort

Implementace algoritmu radix sort byla také kvůli srovnání výkonnosti převzata z práce [5]. Algoritmus je rozdělen na čtyři hlavní fáze.

V první fázi se rozdělí vstupní posloupnost na části, za účelem práce se sdílenou pamětí. Velikost klíče (např. 32 bitů) rozhoduje o počtu průchodů funkce, která rozdělí přidělenou část do přehrádek. Každé vlákno v jednom průchodu zpracovává jeden bit jednoho prvku a má přehled o tom, kolik vláken zpracovávalo stejný bit. Tím se po všech průchodech určí jeho výsledná pozice v bloku.

V druhé fázi se pro všechny seřazené části z předchozího kroku spočítá, kolik prvků jednotlivé přehrádky obsahují spolu s relativní pozicí přehrádek od začátku části, v nichž se nacházely. Z těchto údajů se vytvoří tabulka histogramů.

V následujícím kroku se nad tabulkou spočte suma, čímž se určí začínající pozice přehrádek ve výsledném poli. Nakonec se pro každý prvek seřazených částí určí jeho výsledná pozice sečtením jeho pozice v seřazené části a pozice jeho přehrádky z tabulky. Výsledky jsou pak uloženy do globální paměti.

Implementace radix sortu dostupná v linuxové verzi SDK z verze 2.2 momentálně podporuje řazení desetinných čísel typu float nebo celých nezáporných čísel. Je zde také možnost předaní ukazatele na pole celých nezáporných čísel, což nejspíš vychází z předpokladu užití struktury typu klíč–hodnota.

Kapitola 5

Testování

V této kapitole je popsána testovací metodika, data, na nichž testy probíhaly a komentáře k naměřeným výsledkům.

5.1 Testovací sestava a metodika

Testy probíhaly na dvou počítačích, jejich parametry jsou omezeny pouze na významné komponenty. Například parametry pevných disků byly vynechány, protože všechna testovací data byla předem načtena do paměti RAM.

pcnlp3	Hardware
Procesor	Core 2 Duo 2,66 GHz, 3MB cache L2
Paměť	2 GB
Grafická karta	GeForce 8800 GT 512MB
	Software
Operační systém	Linux x86_64, jádro 2.6.24
Ovladač GK	180.44

pcjosth	Hardware
Procesor	Core 2 Duo 2,33 GHz, 3MB cache L2
Paměť	4 GB
Grafická karta	GeForce 280 GTX 512MB
	Software
Operační systém	Windows XP
Ovladač GK	185.85

Testovacími daty byla pole celých nezáporných čísel, desetinných čísel a pole datových struktur sestávající z klíče (desetinného čísla) a ukazatele na řetězec. Ta byla načtena do paměti, náhodně přeházena pomocí algoritmu *random_shuffle* ze standardní C++ knihovny. Tato data byla následně překopírována do paměti grafické karty a to dvakrát. Jedna kopie sloužila k uchování dat, aby všechny algoritmy měly stejná vstupní data. Druhá kopie byla pracovní, v ní probíhalo samotné řazení grafickým jádrem. Čas byl měřen v prostředí CUDA jejich vnitřním časovačem. V každém testu byl zvlášť měřen samotný čas nutný k seřazení a čas potřebný pro paměťové operace. Tím byla míněna alokace paměti na grafické kartě, zkopírování vstupních dat, po aplikaci algoritmu překopírování výsledků zpět do paměti

procesoru a nakonec uvolnění alokované paměti. Proto se v následujících tabulkách vyskytuje dvakrát položka zrychlení, jednou je započítán pouze čas potřebný k seřazení a podruhé jsou započítány i paměťové operace, což mělo simulovat možné nasazení v běžných podmínkách. Po každém testu byla ještě provedena kontrola, zda výsledná posloupnost je opravdu seřazena a zda nedošlo k chybě.

Jako řadící algoritmus na CPU byla zvolena funkce *std::sort* ze standardní šablonové knihovny C++. Jeho implementace je silně závislá na dodavateli překladače, ale většinou se jedná o implementaci Introsortu, což je varianta algoritmu Quicksort, která v některých situacích přepíná na řazení pomocí algoritmu *Heap sort*, například při špatné volbě pivota, kdy se posloupnost může rozdělit na dvě části velmi rozdílných velikostí. V některých překladačích se dodávají i paralelní verze, toto chování však nebylo bohužel pozorováno. Procesor využíval vždy jen jedno své jádro. Což je do jisté míry škoda, srovnání mohlo být o to zajímavější.

Na všech počítačích byl program kompilován pomocí překladače *nvcc* verze 2.2.

5.2 Výsledky

Kvůli algoritmu bitonic sort byly velikosti vstupních posloupností omezeny na velikosti mocniny čísla dvě. Toto omezení se dá obejít například přialokováním dodatečných položek, jež by byly po skončení řazení na konci posloupnosti, takže následné překopírování by nedělalo problém. Tím by však významně utrpěla efektiva, zvláště pro posloupnosti, jejichž velikost by byla jen o něco málo větší, než nejbližší mocnina.

Na druhém testovacím počítači nebyla nainstalována poslední verze ovladače, který by poskytoval plnou podporu pro CUDA verze 2.2. Nebylo tudíž možno otestovat na pcnlp3 algoritmus Quicksort. Kompilace proběhla bezproblému, za běhu však došlo k chybě s hlášením neimplementované funkcionality. To bylo poněkud zarážející, protože implementace byla vyvíjena se starší verzí překladače.

5.2.1 Celá čísla

Velikost	CPU (ms)	Bitonic (ms)	Radix (ms)	Quick (ms)	Datové operace (ms)
8192	0.66859	0.28003	0.39191	3.42816	0.26668
16384	1.46136	0.34816	0.41939	3.94738	0.33710
32768	3.09511	0.54597	0.59818	4.32770	0.41149
65536	6.57391	0.87768	0.93511	8.42227	0.59847
131072	14.10475	1.56647	1.27853	17.55379	0.96633
262144	29.22777	2.95541	2.02483	28.15770	1.65287
524288	58.64677	6.06318	3.54283	49.21295	3.70934
1048576	107.06160	12.62027	6.44352	92.41550	7.12262
2097152	198.86443	27.74814	11.59665	92.53993	14.14050
4194304	357.92874	58.31457	24.60136	126.86806	26.65422
8388608	636.01184	124.23375	49.34919	262.83919	50.50500

Tabulka 5.1: Řazení celých nezáporných čísel (*unsigned int*) na pcjosth

Velikost	CPU (ms)	Bitonic sort (ms)	Radix sort (ms)	Datové operace (ms)
8192	0.45300	0.24900	0.39700	0.24300
16384	0.96300	0.43100	0.54100	0.29900
32768	2.03200	0.70600	0.75500	0.41200
65536	4.29100	1.28800	1.34100	0.63100
131072	9.09700	2.58600	2.22400	1.05300
262144	18.99000	5.34400	3.95800	1.89700
524288	38.59200	11.37900	7.46800	3.05500
1048576	74.31300	24.67000	14.45600	5.30000
2097152	139.75000	53.39200	39.18000	9.60600
4194304	268.00900	117.75600	78.85500	18.18400
8388608	504.80399	255.53799	161.48700	35.59200

Tabulka 5.2: Řazení celých nezáporných čísel (*unsigned int*) na pcnlp3

Velikost	Zrychlení samotného řazení			Včetně paměťových operací		
	Bitonic sort	Radix sort	Quicksort	Bitonic sort	Radix sort	Quicksort
8192	2.38761	1.70598	0.19503	1.22296	1.01519	0.18095
16384	4.19738	3.48449	0.37021	2.13256	1.93176	0,34108
32768	5.66902	5.17424	0.71518	3.23265	3.06549	0.65309
65536	7.49010	7.03006	0.78054	4.45341	4.28663	0.72875
131072	9.00415	11.03201	0.80352	5.56883	6.28313	0.76159
262144	9.88957	14.43470	1.03800	6.34244	7.94729	0.98045
524288	9.67261	16.55367	1.19169	6.00119	8.08679	1.10817
1048576	8.48330	16.61539	1.15848	5.42279	7.89182	1.07558
2097152	7.16676	17.14843	2.14896	4.74745	7.72675	1.86411
4194304	6.13790	14.54915	2.82127	4.21247	6.98321	2.33144
8388608	5.11948	12.88799	2.41978	3.63979	6.36941	2.02975

Tabulka 5.3: Zrychlení řazení celých nezáporných čísel na pcjosth

Velikost	Zrychlení samotného řazení		Včetně paměťových operací	
	Bitonic sort	Radix sort	Bitonic sort	Radix sort
8192	1.81928	1.14106	0.92073	0.70781
16384	2.23434	1.78004	1.31918	1.14643
32768	2.87819	2.69139	1.81753	1.74122
65536	3.33152	3.19985	2.23606	2.17596
131072	3.51779	4.09038	2.49986	2.77601
262144	3.55352	4.79788	2.62257	3.24338
524288	3.39151	5.16765	2.67369	3.66740
1048576	3.01228	5.14063	2.47958	3.76154
2097152	2.61743	3.56687	2.21832	2.86455
4194304	2.27597	3.39876	1.97152	2.76187
8388608	1.97546	3.12597	1.73395	2.56143

Tabulka 5.4: Zrychlení řazení celých nezáporných čísel na pcnlp3

Výkony algoritmů prováděných na prvním počítači jsou podle předpokladů o poznání vyšší. Přítomná grafická karta je z teoretického hlediska přibližně 2,5krát rychlejší, což se potvrzuje i v testech. Zarážející jsou však výsledky algoritmu Quicksort, který měl být podle dostupných materiálů výkonnější. Jediný problém bych viděl v kompilaci, kdy překladač hlásí odstranění přebytečných synchronizační instrukcí, na něž by mohl algoritmus spoléhat. Tyto synchronizace však měly být nahrazeny atomickými funkcemi novějších karet, ale nejspíš se tak neděje. Proto předpokládám, že zdrojový kód si nerozumí s novějšími verzemi programového balíku od verze 2.0. Quicksort tedy nedosahuje výkonu CPU natož bitonic sortu. Procesor na druhém počítači je o něco výkonnější spolu se slabší grafickou kartou a výkonnostní nárůst není proto moc přesvědčivý.

5.2.2 Desetinná čísla

Velikost	CPU (ms)	Bitonic sort (ms)	Radix sort (ms)	Datové operace (ms)
8192	0.82617	0.27813	0.39124	0.28190
16384	1.79271	0.34804	0.41733	0.33523
32768	3.78560	0.54747	0.59996	0.43574
65536	8.01183	0.87380	0.93200	0.62768
131072	17.12434	1.56706	1.27336	0.99580
262144	36.19671	2.96056	2.03102	1.68234
524288	72.02647	6.05864	3.54569	3.22417
1048576	132.59962	12.61063	6.41217	6.97019
2097152	245.83641	26.78326	11.62865	14.02518
4194304	440.88678	57.27067	24.60843	26.80334
8388608	779.82715	123.07912	49.53699	49.89220

Tabulka 5.5: Řazení desetinných čísel (*float*) na pcjosth

Velikost	CPU (ms)	Bitonic sort (ms)	Radix sort (ms)	Datové operace (ms)
8192	0.60200	0.25000	0.39700	0.25300
16384	1.28400	0.43200	0.54200	0.29900
32768	2.72400	0.70900	0.76100	0.41300
65536	5.78600	1.29000	1.34800	0.62800
131072	12.25500	2.57800	2.24900	1.05100
262144	25.71500	5.36100	3.97800	1.88100
524288	52.32300	11.37800	7.46100	3.04900
1048576	101.75800	24.59400	14.52800	5.28200
2097152	193.99001	53.95100	39.68300	9.62200
4194304	374.36899	117.70200	79.48700	18.19500
8388608	712.99402	255.82899	160.90601	35.23500

Tabulka 5.6: Řazení desetinných čísel (*float*) na pcnlp3

Za povšimnutí stojí fakt, že grafický procesor vůbec nerozlišuje, zda porovnává celá nebo desetinná čísla, na rozdíl od klasického procesoru. Doby řazení pro stejnou délku vstupu se liší jen nepatrně, statistická chyba. Zajímavý je také lineární nárůst doby provádění v případě algoritmu radix sort, který opravdu potvrdil teoretické předpoklady a s dvojnásobnou

Velikost	Zrychlení samotného řazení		Včetně paměťových operací	
	Bitonic sort	Radix sort	Bitonic sort	Radix sort
8192	2.97047	2.11166	1.47524	1.22734
16384	5.15088	4.29561	2.62371	2.38212
32768	6.91473	6.30974	3.85025	3.65511
65536	9.16898	8.59637	5.33595	5.13682
131072	10.92766	13.44820	6.68171	7.54655
262144	12.22632	17.82194	7.79615	9.74770
524288	11.88823	20.31381	7.75912	10.63928
1048576	10.51491	20.67937	6.77191	9.90854
2097152	9.17873	21.14058	6.02416	9.58284
4194304	7.69830	17.91609	5.24403	8.57560
8388608	6.33598	15.74232	4.50842	7.84304

Tabulka 5.7: Zrychlení řazení desetinných čísel (*float*) na pcjosth

Velikost	Zrychlení samotného řazení		Včetně paměťových operací	
	Bitonic sort	Radix sort	Bitonic sort	Radix sort
8192	2.40800	1.51637	1.19682	0.92615
16384	2.97222	2.36900	1.75650	1.52675
32768	3.84203	3.57950	2.42781	2.32027
65536	4.48527	4.29228	3.01668	2.92814
131072	4.75368	5.44909	3.37696	3.71364
262144	4.79668	6.46430	3.55081	4.38897
524288	4.59861	7.01287	3.62674	4.97840
1048576	4.13751	7.00427	3.40601	5.13670
2097152	3.59567	4.88849	3.05145	3.93449
4194304	3.18065	4.70981	2.75480	3.83253
8388608	2.78699	4.43112	2.44961	3.63511

Tabulka 5.8: Zrychlení řazení desetinných čísel (*float*) na pcnlp3

délkou vstupu je doba také dvojnásobná. U větších posloupností je to zřetelnější.

Datové operace také trvají relativně dlouhou dobu v porovnání s dobou provádění samotného řazení. V případě algoritmu radix sort pro největší vstupní délky zabere prakticky víc času samotné kopírování dat nežli řazení, to prakticky navyšuje dobu trvání na dvojnásobek. Zajímavé také je pozorování, že na grafické kartě GeForce 8800GT trvá alokace a kopírování dat kratší dobu než novější a výkonnější kartě v prvním počítači, alespoň tedy pro vstupy největších délek, kde se už nedá hovořit o statistické chybě. Napadá mne snad jen jedno vysvětlení a to že proces přenosu dat je více závislý na rychlosti procesoru, který je v této situaci pomalejší komponentou.

5.2.3 Vlastní datová struktura

Velikost	CPU (ms)	Bitonic sort (ms)	Datové operace (ms)	Zrychlení	Zrychlení*
8192	0.84693	0.40900	0.33604	2.07075	1.13676
16384	1.86473	0.49335	0.43943	3.77975	1.99912
32768	3.97194	0.97954	0.66467	4.05491	2.41572
65536	8.36922	1.64281	1.04646	5.09447	3.11208
131072	17.91253	3.13722	1.71550	5.70969	3.69124
262144	37.65474	6.31849	3.23298	5.95946	3.94230
524288	76.53876	13.40108	7.17181	5.71139	3.72037
1048576	141.45062	28.81810	13.80554	4.90839	3.31860
2097152	268.11377	62.29179	26.76694	4.30416	3.01053
4194304	478.40256	133.89590	50.23302	3.57294	2.59819
8388608	843.32483	286.19550	98.13959	2.94667	2.19424

Tabulka 5.9: Řazení struktury typu klíč–hodnota na pcjosth

Velikost	CPU (ms)	Bitonic sort (ms)	Datové operace (ms)	Zrychlení	Zrychlení*
8192	0.60600	0.57100	0.29900	1.06130	0.69655
16384	1.29500	1.31600	0.41400	0.98404	0.74855
32768	2.72900	2.64100	0.63500	1.03332	0.83303
65536	5.78300	5.91100	1.06100	0.97835	0.82946
131072	12.27500	13.65000	1.90200	0.89927	0.78929
262144	25.48700	31.98700	3.06500	0.79679	0.72712
524288	51.62100	74.68500	5.29600	0.69118	0.64542
1048576	100.50100	171.94099	9.61000	0.58451	0.55357
2097152	193.54201	393.87900	18.24500	0.49137	0.46962
4194304	381.73001	885.73297	35.32900	0.43098	0.41445
8388608	730.86798	1981.66003	69.31000	0.36882	0.35635

Tabulka 5.10: Řazení struktury typu klíč–hodnota na pcnlp3

Dříve testované algoritmy jsou schopny řazení pouze číselných klíčů, ne uživatelem definované typy, což je minimálně pro radix sort škoda. Testy s posledními daty jsou také zajímavé. V případě GeForce 8800GT dochází zdvojnásobením velikosti datové struktury přidáním ukazatele k 8-násobnému nárůstu doby provádění, oproti 2-násobnému v případě GeForce 280 GTX. Netuším, čím by toto chování mohlo být způsobeno, snad jen vylepšenou architekturou novější generace grafických jader. Bohužel jsem neměl přístup k jiným kartám poslední generace pro ověření tohoto předpokladu. Tento jev naprosto degraduje výkon na druhém počítači a použití algoritmu je na starších kartách zbytečné. Naproti tomu novější karta se stále drží, ale dá se předpokládat, že se zvětšující se datovou strukturou by stejným tempem rostla i délka provádění, což nenastává u klasického procesoru.

Celkově lze konstatovat, že algoritmus bitonic sort dosahuje největšího zrychlení pro posloupnosti délky 256000, oproti tomu dva miliony pro radix sort. Oba se hodí spíše pro primitivní datové struktury.

Kapitola 6

Závěr

Cílem této bakalářské práce bylo zjistit, zda lze využít výkon moderních grafických karet pro jiné než grafické a vizualizační účely. Zkoumanou oblastí byly zvoleny řadící algoritmy, jakožto jeden z často se vyskytujících problémů. Tato oblast je v poslední době zájmem mnoha programátorských skupin.

Implementován byl řadící algoritmus bitonic sort, který byl původně navržen právě pro paralelní systémy. Ten byl napsán jako šablonová funkce, měl by být schopen práce s uživatelsky definovanými typy s případnými modifikacemi, jež byly zmíněny v části 4.1.

Významnou součástí byly také výkonnostní testy. Do srovnání byly zahrnuty algoritmy dostupné přímo v ukázkových příkladech k balíku CUDA a další dostupné implementace. Ty jsou momentálně omezeny na úzkou množinu vstupních dat, čemuž musela být testovací data přizpůsobena. Na základě testů a získaných zkušeností během práce v prostředí CUDA bych se odvážil konstatovat, že pro účely řazení jednoduchých datových struktur například číselného typu jsou více než vhodné, ale pro velké struktury či objekty již méně. S větší strukturou dochází k častějšímu přístupu do globální paměti grafické karty, což je velmi častá ale poměrně drahá operace.

V možnostech dalšího uplatnění jsem toho názoru, že grafické karty nejsou vhodné pro řazení řetězců. Hlavní důvod vidím v různé délce zpracovávaných řetězců. Každé vlákno by mohlo potřebovat řetězce jiných délek a to by nefungovalo korektně s mechanismy, jak jádro provádí a optimalizuje přístup do globální paměti, docházelo by tak během porovnávání a hlavně přesunů k přístupu do paměti, který by byl náhodný, nesouvislý a pro více vláken by se dal snad jen těžko optimalizovat a řetězit. Řešení tohoto problému bych viděl v použití „hashovací“ funkce, která by zohledňovala abecední pořadí a poté byl výsledek použit jako klíč k řazení. Nehledě na to, že CUDA momentálně nedisponuje žádnou podporou pro práci s řetězci, jež by byla obdobou standardní knihovny v jazyku C. Nevylučuji však tuto možnost definitivně, ale doménou grafických karet je spíše práce s plovoucí desetinnou čárkou.

Popravdě musím ještě konstatovat, že pokud by si dal někdo stejnou práci s návrhem, úpravou a optimalizací algoritmů pro vícejaderné procesory, mohl by se dopracovat k hodně podobným výsledkům a podle mého i pro širší množinu vstupních dat.

Velmi si vážím příležitosti vyzkoušet si paralelní programování, bylo to velmi zajímavé začít brát v úvahu provádění více činností najednou, jaké situace díky tomu mohou vyvstat a následně ladit nefunkční algoritmus. Jedná se o cennou zkušenost do budoucna. Prostředí CUDA mi tento přechod ulehčilo, ať už díky dobré dokumentaci, emulačnímu režimu nebo základu a napojení na jazyky C/C++.

Literatura

[1] Alexey Berillo: NVIDIA CUDA.
http://ixbtlabs.com/articles3/video/cuda-1-p5.html, 2008.

[2] Cederman„ Daniel and Tsigas„ Philippas: A Practical Quicksort Algorithm for Graphics Processors. In *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 978-3-540-87743-1, s. 246–258, doi:–http://dx.doi.org/10.1007/978-3-540-87744-8˙21″.

[3] Fedy Abi-Chahla: The CUDA APIs.
http://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954-6.html, 2008.

[4] Greß, Alexander and Zachmann, Gabriel: GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures (extended version). Technická Zpráva IfI-06-11, Clausthal University of Technology, sep 2006.

[5] Nadathur Satish and Mark Harris and Michael Garland: Designing Efficient Sorting Algorithms for Manycore GPUs. NVIDIA Technical Report NVR-2008-001, NVIDIA Corporation, sep 2008.

[6] NVIDIA Corporation: GPU Gems, chapter 37.
http://http.developer.nvidia.com/GPUGems/gpugems_ch37.html, 2008.

[7] NVIDIA Corporation: The CUDA Compiler Driver NVCC. 2008.

[8] NVIDIA Corporation: The CUDA Programming Guide. 2008.

[9] NVIDIA Corporation: NVIDIA CUDA Architecture Introduction & Overview.
http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf ,
2009.

[10] Wikipedie: Řadící algoritmus.
http://cs.wikipedia.org/wiki/Řadicí_algoritmus, 2009.

[11] www stránky: http://david.padrta.sweb.cz/obrazky/radixsort.gif.

[12] www stránky: OpenCL. http://www.khronos.org/opencl/.