Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Diplomová práce

Paralelní řadící algoritmy

Bc. Pavel Řehák

Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

8. ledna 2018

Poděkování

Děkuji své manželce a rodičům za velkou podporu při mém dlouhém magisterském studiu.

Prohlášení

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

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

V Praze dne 8. ledna 2018 .....

Odkaz na tuto práci

Řehák, Pavel. *Paralelní řadící algoritmy.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato práce pojednává o řadících algoritmech quicksort, mergesort a radixsort. V první části je popsána jejich sekvenční varianta. Následně je provedena paralelizace algoritmů pomocí technologie OpenMP. Implementace těchto algoritmů je poté ještě upravena i pro efektivní běh pod nVidia CUDA. V další kapitole popisuji svou implementaci těchto algoritmů. Na konci je provedeno měření a porovnání s jinou podobnou implementací od jiného autora.

Klíčová slova řadící algoritmy, paralelní, paralelizace, openmp, cuda, quicksort, mergesort, radixsort

Abstract

This thesis concerns about sorting algorithms quicksort, mergesort and radixsort. In first part is described their sequential variant. In next step is implementation paralelized by OpenMP technology. Implementation of this issue is in the next stage altered for effective run under nVidia CUDA. In next chapter I have described my implementation of these algorithms. In the end of this thesis measurement and comparison with other similar implementation is done.

Keywords sorting algorithms, parallel, parallelization, openmp, cuda, quicksort, mergesort, radixsort

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Cílem této diplomové práce je seznámit čtenáře s principy fungování vybraných řadících algoritmů a jejich efektivní implementace. Následuje paralelizace v OpenMP a následně pomocí technologie nVidia CUDA. Výsledné zrychlení bude porovnáno s výsledky jiného existujícího řešení.

První kapitola pojednává o řadících algoritmech, jejich vlastnostech obecně a následně detailně popisuje řadící algoritmy quicksort, mergesort a radixsort. Pomocí pseudokódu dopomohu k snažšímu pochopení problematiky jednotlivých algoritmů. V této kapitole mimo jiné také rozeberu různé metody výběru pivota. Také vysvětím rozdíl mezi in-place a out-place implementací řadícího algoritmu mergesort. V závěru popíši svou variantu implementace všech tří algoritmů nad datovými strukturami a celočíselnými hodnotami bez jakékoli paralelizace.

Ve druhé kapitole zmíním existující implementační nástroje pro paralelizaci, zejména technologie OpenMP a nVidia CUDA.

Ve třetí kapitole vysvětlím a odůvodním úpravy implementace sekvenčních algoritmů, tak aby běžely paralelně pomocí OpenMP.

Ve čtvrté kapitole se zaměřím na techniku paralelizace pomocí CUDA technologie. Dále vysvětlím úpravy zdrojového kódu, abychom dosáhli efektivní paralelizace pomocí této technologie.

V páté kapitole popíši svou implementaci programu, která obsahuje algoritmy quicksort, mergesort a radixsort ve variantě sekvenční, OpenMP i CUDA.

V poslední kapitole provedu testování mnou implementovaných algoritmů a výsledky zrychlení (či zpomalení) paralelizací vůči jiné a vhodně zvolené aplikaci se stejnou či podobnou funkčností nad identickým vzorkem dat.

Kapitola 1

Řadící algoritmy

Řadící algoritmus je správný termín pro častěji používané, alespoň mezi informatiky, slovní spojení *třídící algoritmus*. Obecně lze takový algoritmus definovat, tak že máme universum prvků, na kterém existuje úplné uspořádání. Cílem algoritmu je nalézt takovou permutaci prvků, pro kterou platí $a_1 < a_2 < \ldots < a_n$ pro všech n prvků z universa.

Práce s daty je pro každý program o hodně snažší, pokud se jedná o data seřazená. V seřazených datech lze vyhledávat velmi rychle, např. pomocí binárního vyhledávání s $\mathcal{O}(\log n)$. To bývá častým důvodem, proč data řadíme.

V této kapitole popíšu RAM model, porovnávací model využívaný u řazení, základní vlastnosti řadících algoritmů pro jejich možné popsání v dalších částech této práce. Následně věnuji pár odstavců jednodušším řadícím algoritmům (selectsort, bubblesort) a poté přejdu k detailnějšímu popisu zajímavějších řadících algoritmů (quicksort, mergesort, radixsort), což je hlavní náplní této diplomové práce.

1.1 Výpočetní model RAM

Abychom se nemuseli zabývat obtížnějšími otázkami, mezi které řadíme např. reprezentaci reálných čísel, tak si definujeme teoretický stroj RAM (Random Access Machine). Ten bude mít přesně definované chování, vlastnosti paměťových buněk a definovaný čas provádění instrukcí. V tomto výpočetním modelu dává smysl měřit časovou a paměťovou složitost níže popsaných algoritmů, jelikož se nemusíme zabývat nežádoucími vlivy, kam patří např. běh operačního systému.

Paměť tohoto modelu je tvořena neomezeně velkým blokem celočíselných buněk adresovatelnými celými čísly. Do každé buňky lze uložit jedno celé

číslo. Velikost každé buňky je buď neomezená nebo je definována její velikost. V tomto modelu je program definován jako konečná posloupnost aritmetických a řídících instrukcí. Instrukční sada se skládá z instrukcí zajišťující aritmetické operace, logické operace, větvení programu a přesuny dat mezi jednotlivými paměťovými buňkami.

Na začátku běhu programu, čímž je spuštění první instrukce, jsou v paměti vstupní data. Po sekvenčním provedení všech instrukcí, tedy ukončení programu, se nacházejí v paměti data výstupní. U takto definovaného programu můžeme měřit jeho časovou složitost jako počet provedených instrukcí a prostorovou složitost jako počet použitých paměťových buněk [19]. Do časové složitosti algoritmu nezapočítáváme načtení vstupních dat do paměti ani jakýkoliv výstup programu, ať již do konzole nebo do výstupního souboru.

1.2 Vlastnosti algoritmů

Každý řadící algoritmus má své vlastnosti stejně tak jako i jiný algoritmus. Díky tomu je lze kategorizovat do určitých skupin. Některé algoritmy jsou např. vhodnější pro menší množství dat, některé jsou naopak vhodné pro větší množství dat. V této sekci vysvětlím některé termíny a vlastnosti algoritmů, které používám v textu dále.

1.2.1 Časová složitost

Časová složitost nám pomáhá pochopit, jak se algoritmus chová při větších vstupech. Složitost u menších vstupů není nijak zajímavá, jelikož při rychlosti dnešních počítačů je doba běhu velmi krátká a jejich porovnávání nemá vypovídající hodnotu o rychlosti algoritmu. Pro určení se využívá asymptotická časová složitost, která je definována následovně.

Nechť f a g jsou dvě přirozené funkce z přirozených čísel do přirozených čísel. Definujeme:

1. $f(n) = \mathcal{O}(g(n))$, právě tehdy, když existuje konstanta $c > 0$ a n_0 takové, že pro každé $n \geq n_0$ platí $f(n) \leq c \cdot g(n)$,
2. $f(n) = o(g(n))$, právě tehdy, když existuje konstanta $c > 0$ a n_0 takové, že pro každé $n \geq n_0$ platí $f(n) < c \cdot g(n)$,
3. $f(n) = \Omega(g(n))$, právě tehdy, když existuje konstanta $c > 0$ a n_0 takové, že pro každé $n \geq n_0$ platí $f(n) \geq c \cdot g(n)$,
4. $f(n) = \omega(g(n))$, právě tehdy, když existuje konstanta $c > 0$ a n_0 takové, že pro každé $n \geq n_0$ platí $f(n) > c \cdot g(n)$,

5. $f(n) = \Theta(g(n))$, právě tehdy, když platí zároveň $f(n) = \mathcal{O}(g(n))$ i $f(n) = \Omega(g(n))$.

U algoritmů můžeme definovat tři typy časových složitostí, které nás zajímají. Prvním typem je tzv. *average-case*, který nám v asymptotické notaci říká, jaká je průměrná doba běhu algoritmu v závislosti na vstupních datech. Tento typ lze parametrizovat pomocí vstupních dat nebo náhodnými bity. Parametrizací pomocí náhodných bitů může být např. náhodný výběr pivota u quicksortu.

Druhým typem je *worst-case*, který nám ukazuje jak dlouho algoritmus poběží v nejhorším možném případě. Posledním typem je *best-case*, který nám naopak udává dobu běhu v nejlepším možném případě. Tento poslední typ se používá při analýze algoritmů méně a to z toho důvodu, že v realném využití algoritmů nás více zajímá worst-case. Ten když známe, tak můžeme vhodně vybrat hardware pro běh tohoto algoritmu, tak aby i v nejhorším případě měl systém dostatek zdrojů jak výpočetních tak i paměťových.

1.2.2 Paměťová složitost

K určení míry využití paměti algoritmem se používá také asymptotická notace. Některé řadící algoritmy umí provést své kroky bez potřeby zabírání další paměti navíc nebo si vystačí s konstantní částí paměti. Do paměťové složitosti se nezapočítává vstup - zajímá nás tedy pouze paměťová složitost operací algoritmu nad načtenými daty. Paměťovou složitost v této práci měřím jako počet použitých slov (buněk paměti). Formálně můžeme definovat dva typy algoritmů.

1. Pro *in-place* algoritmy platí, že využijí nanejvýš $o(n)$ paměti vedle paměti, kde jsou uložená vstupní data. Někteří autoři ale ve svých odborných publikacích používají striktnější definici a to konkrétně, že pro paměť použitou algoritmem navíc vedle vstupních uložených dat platí $\mathcal{O}(\log n)$ nebo i dokonce jen $\mathcal{O}(1)$ [20]. Příkladem in-place algoritmu je např. bubblesort, kterému stačí uložení do paměti pouze tří proměnných pro korektní fungování.

2. Všechny ostatní algoritmy nazýváme *out-place*. Jejich asymptotická paměťová složitost je závislá na n, tedy např. $\mathcal{O}(n)$ nebo $\mathcal{O}(n \log n)$ – jejich složitost je tedy alespoň lineární. Příkladem je klasický mergesort, který seřazená data ukládá do paměti vedle vstupních dat.

Z definice výše tedy platí, že in-place algoritmy jsou vždy paměťově úspornější než out-place. Tato paměťová úspora ale může být na úkor rychlosti algoritmu, jako je tomu například u algoritmu mergesort a jeho varianty in-place a out-place.

1.2.3 Stabilita

Stabilita řadícího algoritmu nám říká, zda-li je zachováno pořadí prvků se stejnou hodnotou na výstupu jako bylo pořadí těchto prvků na vstupu. Algoritmus je tedy stabilní, pokud relativní pořadí prvků se stejnými hodnotami je zachováno po seřazení, v opačném případě se jedná o algoritmus nestabilní. Mějme např. posloupnost $7, 5_{(1)}, 5_{(2)}, 3$ na vstupu. Pokud výsledkem vzestupného řazení bude posloupnost $3, 5_{(1)}, 5_{(2)}, 7$, tak můžeme prohlásit řazení za stabilní. Abychom mohli řadící algoritmus prohlásit za stabilní, pak musí při každém seřazení dojít ke stabilnímu seřazení. V opačném případě, pokud budou prohozeny prvky s hodnotou 5 a tedy výstupní posloupnost bude v pořadí $3, 5_{(2)}, 5_{(1)}, 7$, pak se jedná o nestabilní algoritmus.

1.3 Dolní odhad složitosti řazení

V této části se zaměříme na hledání odpovědi, proč nelze řadit rychleji nežli $\mathcal{O}(n \log n)$. Pro nalezení korektní odpovědi si musíme uvědomit dva předpoklady [20].

- Pracujeme v tzv. *porovnávacím modelu*, tedy mezi jediné povolené operace v algoritmu patří porovnávání prvků a jejich přesouvání v paměti.
- Algoritmus neobsahuje žádný prvek náhody a jeho kroky jsou určeny kroky předchozími. Takové algoritmu říkáme, že je deterministický.

Pokud jsou výše uvedené předpoklady splněné, pak řazení n-prvkové posloupnosti trvá v nejhorším možném případě $\Omega(n \log n)$ [20]. Pro dokázání výše uvedeného tvrzení si vybereme (či mírně upravíme) libovolný řadící algoritmus, který nejdříve všechny prvky porovná a následně až začne s přesouváním prvků v paměti. Následně vytvoříme rozhodovací strom, který nám popíše všechny možné stavy algoritmu. Vnitřní vrcholy reprezentují porovnání dvou prvků, v listech stromu se provede prohození prvků a algoritmus skončí.

Jelikož pro různě uspořádané vstupy musí algoritmus skončit v různých listech, tak musí existovat nejméně $n!$ listů a hloubka stromu musí být $\log n!$, což dle lemmatu $f! \geq n^{n/2}$ je $\Omega(n \log n)$ [20]. Z tohoto důvodu musí existovat takový vstup pro náš řadící algoritmus, který provede $\Omega(n \log n)$ porovnání.

1.4 Jednoduché řadící algoritmy

Popsané algoritmy níže jsou jednoduché a krátké na implementaci. Bonusem je i navíc, že se jedná o algoritmy typu in-place, tedy nejsou náročné na paměť. Jejich hlavní nevýhodou je časová složitost $\mathcal{O}(n^2)$, nehodí se tedy pro řazení většího souboru dat.

1.4.1 Bubblesort

Tento algoritmus pracuje na jednoduchém principu s kvadratickou složitostí. Algoritmus prochází opakovaně celé neseřazené pole a porovnává všechny sousedící prvky na pozicích `i` a `i+1`. Pokud je prvek na pozici `i+1` menší nežli prvek na pozici `i`, tak je prohodí. Poté se `i` inkrementuje o jedničku, opět se porovnají a případně i prohodí další sousedící prvky. Inkrementace probíhá tak dlouho, aby se porovnaly i dva poslední prvky v poli. Tento celý proces si lze představit jako probublávání větších prvků na konec pole a menších prvků na začátek, proto se tento algoritmus nazývá bublinkové řazení. Celý průchod polem se opakuje do té doby, dokud se prohodí alespoň jeden prvek. Pokud se v celém průchodu neprohodí ani jeden prvek, tak algoritmus končí a vstupní data jsou seřazená.

1.4.2 Selectsort

Na vstupu máme data o velikosti n. V prvním průchodu celého algoritmu se nalezne prvek s nejmenší hodnotou a prohodí se s prvkem na prvním místě. První prvek nyní tvoří seřazenou část a zbytek tvoří neseřazenou část. Algoritmus pokračuje tak, že prochází celou neseřazenou část, najde opět minimum a prohodí ho s prvním prvkem v neseřazené části. Velikost seřazené části se inkrementuje o hodnotu 1 a velikost neseřazené části se dekrementuje o hodnotu 1. Tento proces pokračuje do té doby, dokud v neseřazené části zbývá alespoň 1 prvek.

1.5 Quicksort

Tento algoritmus patří do třídy algoritmů typu *rozděluj a panuj* (angl. divide and conquer). Algoritmy tohoto typu nalézají řešení způsobem, že rozdělí problém na dílčí podproblémy stejného nebo podobného typu, které jsou snadněji řešitelné. Řešení těchto podproblémů je následně zkombinováno zpět a výsledkem je hledané řešení celého problému. Dělení na menší části lze provádět jak pomocí rekurzivního volání tak i pomocí iteračních cyklů.

Algoritmus je typu in-place a je datově citlivý. Pro náhodná data může být opravdu rychlý. Ale např. u vstupu s identickými hodnotami při klasické implementaci funkce `rozdělení` na dvě části ($<$ a $=>$) bude časová složitost $\mathcal{O}(n^2)$ [12]. I když nemusí dojít k žádnému prohazování prvků, tak bude provedeno n rekurzivních volání a v nich bude prováděno porovnávání. Tomuto lze předejít např. implementací funkce `rozdělení` na tři části ($<$, $=$ a $>$).

1.5.1 Princip algoritmu

V prvním kroku je vybrán jeden prvek, který se nazývá pivot. Existují různé možnosti, jak vybrat tento prvek a těm se věnuji v následující podkapitole 1.5.2. Druhým krokem je rozdělení pole na dvě části. V první části se nacházejí prvky s hodnotou menší než pivot, v druhé části se nacházejí prvky s hodnotou větší než pivot a mezi těmito dvěma částmi se nachází pivot. V tomto kroku tedy dochází ke správnému přeházení prvků, tak aby jejich hodnoty splňovali podmínku výše uvedenou. V posledním třetím kroku jsou rekurzivně opakovány kroky 1 a 2 na menších a menších množinách dat. Algoritmus tedy dělí problém a rekurzivně se v něm volá funkce `quicksort()` na seřazení daného podproblému. Celý tento cyklus se pokusím demonstrovat na pseudokódu společně s jednoduchým a vhodně zvoleným příkladem.

```
function quicksort (pole data, cislo levy, cislo pravy)
  if pravy - levy <= 1
    konec
  pozice_pivota = rozdeleni(data, levy, pravy)
  quicksort (data, levy, pozice_pivota - 1)
  quicksort (data, pozice_pivota + 1, pravy)

function rozdeleni (pole data, cislo levy, cislo pravy)
  pivot = vyber_pivota()
  j = levy
  for (i = levy; i < pravy; i++)
    if (data[i] < pivot)
      prohod A[j] a A[i]
      j++
  prohod A[j] a A[pravy]
  return j //pozice_pivota
```

Základem funkce `quicksort()` je algoritmus popsaný viz výše. Pokud je počet čísel k seřazení menší nebo rovno 1, pak již není co řadit a proto můžeme tuto větev rekurze ukončit. V ostatních případech se volá funkce `rozdeleni()`, která určí a vrátí pivota (různé způsoby výběru pivota jsou popsány v podkapitole 1.5.2) a zajistí, aby prvky v poli nalevo od pivota měly menší hodnotu nežli pivot a obdobně aby prvky napravo od pivota měly hodnotu větší nežli pivot. Následně se rekurzivně volá funkce `quicksort()` na dvě menší pole, která jsou rozdělena pivotem. Po ukončení poslední rekurze je pole seřazeno vzestupně. Pro správné rozdělení pole pomocí pivotu na dvě menší pole existuje následující algoritmus[1]:

Definujeme si dvě pomocné číselné proměnné `i` a `j`, první nastavíme hod-

notu 0 (začátek pole) a druhé nastavíme hodnotu indexu posledního prvku pole. Dokud je `i` < `j`, tak se opakují následující kroky:

1. Index `i` inkrementujeme (pokud je zapotřebí) do té doby, dokud hodnota v poli na místě indexu `i` není větší nežli pivot.

2. Index `j` dekrementujeme (pokud je zapotřebí) do té doby, dokud hodnota v poli na místě indexu `j` není menší nebo rovna nežli pivot.

3. Prohodíme prvky v poli s indexy `i` a `j`.

Posledním krokem je prohození pivota s prvkem z pole s indexem `i`.

Pro lepší pochopení problematiky uvažujme následující jednoduchý příklad. Mějme posloupnoust čísel $15, 3, 2, 1, 9, 5, 7, 8, 6$ reprezentující data, která chceme seřadit. Za pivota si určíme například prvek s číslem 7. Dalším krokem je projít celé neseřazené pole a porovnat ho s námi zvoleným pivotem. Pokud je prvek menší, pak ho umístíme do levé části pole. V případě, že je prvek větší než pivot, pak ho umístíme do pravé části pole. Nápomocné jsou nám dvě proměnné `i` a `j` obsahující indexy pole. Pole `data` na pozici `i` je první následující prvek z pole zleva, který je větší než pivot. Pole `data` na pozici `j` je první následující prvek z pole zprava, který je menší než pivot. Tyto pomocné proměnné `i` a `j` nám ukazují na prvky, které se mají v následujícím kroku mezi sebou prohodit. Tedy v praxi to vypadá následovně:

1. *15, 3, 2, 1, 9, 5, 7, 8, 6* - určení pivota (7) a pomocné proměnné i a j jakožto prvního a posledního prvku

2. *15, 3, 2, 1, 9, 5, 7, 8, 6* - určení proměnných i a j - $15 > 7$ a $6 < 7$

3. *6, 3, 2, 1, 9, 5, 7, 8, 15* - výměna proměnných i a j

4. *6, 3, 2, 1, 9, 5, 7, 8, 15* - určení proměnných i a j - $15 > 7$ a $6 < 7$

5. *6, 3, 2, 1, 5, 9, 7, 8, 15* - výměna proměnných i a j

6. Další určení proměnných *i, j* a jejich následná výměna se neprovádí, protože bychom již začali procházet pole podruhé.

7. *6, 3, 2, 1, 5, 7, 9, 8, 15* - výměna proměnné i a pivota (7)

Tímto první volání funkce `rozdeleni()` končí. Všechny prvky 6, 3, 2, 1, 5 nalevo od pivota mají menší hodnotu nežli pivot. Obdobně i všechny prvky 9, 8, 15 napravo od pivota mají vetší nebo rovnou hodnotu nežli pivot. Následuje rekurzivní volání funkce quicksort pro obě posloupnosti prvků. Po doběhnutí poslední rekurze je pole seřazené.

1.5.2 Výběr pivota

Cílem je nalézt takový prvek reprezentující pivota, který rozdělí pole na dvě stejně velké části. Je očividné, že hledaným prvkem je medián ze všech prvků, které chceme seřadit. Medián se hledá velmi lehce u seřazeného pole - jedná se o prostřední hodnotu. Jelikož předpokládáme, že na vstupu máme pole neseřazené, pak tuto metodu použít nemůžeme. Jelikož volba mediánu může značně ovlivnit časovou složitost (od $\mathcal{O}(n \log n)$ se můžeme dostat až k $\mathcal{O}(n^2)$), tak je vhodné při implementaci algoritmu dbát na vhodný výběr. Existují různé způsoby jak zvolit pivota, mezi nejčastější patří:

1. Jako pivot se určí první či poslední prvek ze vstupního pole dat. Tento způsob činí quicksort časově neoptimálním, pokud na vstupu je již seřazené pole.

2. Za pivota se určí prostřední nebo náhodně vybraný prvek ze vstupního pole dat. Tento způsob je již účinější a netrpí problémem popsaným v předešlém bodě ohledně seřazeného pole. Stále ale vybíráme z jednoho prvku náhodně vybraného z větší posloupnosti.

3. V praxi se ukazuje, že nejrychlejší metoda se jeví výběr mediánu ze tří prvků (náhodně vybraných nebo např. prvního, prostředního a posledního prvku [2]). Tuto metoda lze rozšířit i na výběr mediánu z většího počtu prvků, pokud to umožňuje velikost neseřazeného pole. Výhodou této metody je, že se zvyšuje pravděpodobnost, že vybraný prvek se blíží k reálnému mediánu z neseřazeného pole nejvíce ze všech třech popsaných metod. Avšak nevýhodou je režie, která je časově náročnější oproti předešlým metodám. Je zapotřebí minimálně 2 porovnání a v horším případě až 3 porovnání, abychom nalezli medián ze tří prvků. V porovnání s celkovou časovou náročností metody `rozdeleni()` u větších posloupností k seřazení se ale jedná o zanedbatelné zdržení.

4. Další možností je výběr pravého mediánu. Existuje algoritmus FIND na hledání mediánu se složitostí $\mathcal{O}(n^2)$ v nejhorším případě [22]. Tento algoritmus je založen na principu rozděluj a panuj. Druhým a efektivnějším algoritmem je SELECT se složitostí $\mathcal{O}(n)$, kde se hledá medián rekurzivně [23]. Výběr pravého mediánu je zaručeně nejpomalejší metodou ze všech výše popsaných, ale je nepřesnější při následném rozdělování pole na dvě stejně velké části.

Výše popsané metody hledání pivota lze rozdělit na deterministické a nedeterministické. Pokud je pro jeden vstup výběr pivota vždy stejný, pak se jedná o deterministický způsob výběru pivota. V případě, že při výběru pivota se používá prvek náhody, pak říkáme, že se jedná o nedeterministický způsob.

1.5.3 Časová a paměťová složitost

Nad množinou dat n je průměrná doba zpracování tohoto algoritmu rovna $\mathcal{O}(n \log n)$. Nevýhodou je, že v nejhorším případě se může doba zpracování zvýšit až na $\mathcal{O}(n^2)$. Jedná se o nestabilní algoritmus s paměťovou náročností $\mathcal{O}(\log n)$. Quicksort použije v nejhorším možném případě zhruba $n^2/2$ porovnání a v průměru použije zhruba $2n \ln n$ porovnání [1]. V realných testech nad pseudonáhodnými daty vychází tento algoritmus, co se týče rychlosti, lépe než heapsort [37] nebo mergesort viz kapitola 1.6. Ale v případě špatné volby pivota bude samozřejmě pomalejší, a proto ho nelze použít v některých kritických aplikacích, kde se s worst-case časovou složitostí $\mathcal{O}(n^2)$ nelze spokojit. Nevhodný výběr hypotetického mediánu (pivota) není jediným důvodem, kdy může tento algoritmus být po časové stránce doby běhu nevhodným.

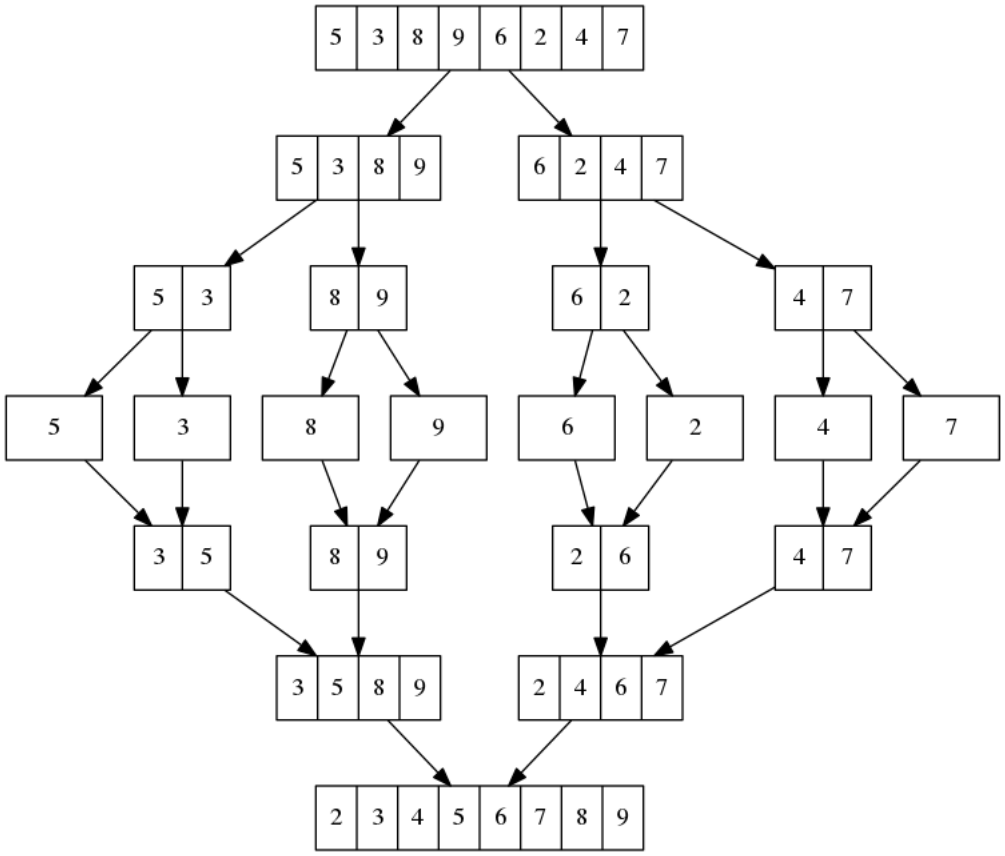
1.6 Mergesort

Tento řadící algoritmus patří stejně jako quicksort mezi algoritmy typu rozděluj a panuj. Jeho průměrná časová složitost je $\mathcal{O}(n \log n)$, což je stejná hodnota jako u quicksortu. Ale v případě nejhorší možné časové složitosti mergesort vyhrává - $\mathcal{O}(n \log n)$ oproti $\mathcal{O}(n^2)$ u quicksortu. Tento algoritmus řadíme mezi stabilní a standartně je implementovaný jako out-place, avšak existují i verze nestabilní a in-place. Průběh algoritmu není citlivý na počáteční uspořádání jeho vstupu[1] jako je tomu u quicksortu.

1.6.1 Princip algoritmu

Algoritmus lze rozdělit na dvě části – dělení a slučování pole. V první části, v které je na vstupu neseřazené pole, dochází k dělení na dvě menší pole. V případě sudé délky je pole rozděleno na dvě menší pole o identické velikosti. V případě že vstupní pole má lichý počet prvků, tedy $2k + 1$ pro $k \in N$, pak po rozdělení je velikost jednoho pole k a druhého $k+1$. Toto rozdělení probíhá rekurzivně, dokud velikost všech rozdělených polí není rovna 1. Poté můžeme říct, že je pole triviálně seřazeno, tzn. každé rekurzivně vzniklé pole o velikosti 1 prvku je seřazeno (a nemůže tomu být jinak, právě protože pole obsahuje jeden prvek). Nyní lze přistoupit k druhé části a to je slučování seřazených malých polí do jednoho velkého pole. Ta iterativně porovnává první prvky ze dvou menších seřazených polí. Menší prvek vždy umístí do nového pole na konec a přenastaví ukazatel na další prvek k porovnání. Opakováním tohoto kroku se ze dvou menších seřazených posloupností dat stane jedna seřazená posloupnost obsahující prvky z obou podposloupností. A takto je definován princip slévání pole zpět dohromady. Této metodě se také říká dvoucestné slévání, jelikož sléváme prvky ze dvou polí do jednoho výsledného.

Na obrázku 1.1 demonstruji princip fungování mergesortu. Nejdříve dojde k rozdělení vstupních dat na prvky a následně se provede dvoucestené slévání do seřazeného výstupního pole.



Obrázek 1.1: Postup mergesortu na příkladu

1.6.2 Varianta out-place

Výše uvedená varianta algoritmu je typu out-place, její asymptotická paměťová složitost je rovna $n + \mathcal{O}(\log n)$. Je zapotřebí alokovat pomocné pole o velikosti n a $\mathcal{O}(\log n)$ je pro zásobník a lokální proměnné. Tato varianta je tedy paměťově náročnější nežli varianta in-place.

1.6.3 Varianta in-place

Cílem této modifikované verze mergesortu je, aby asymptotická paměťová složitost algoritmu byla konstatní nebo logaritmická. V níže popsané verzi je paměťová složitost $\mathcal{O}(\log n)$. Způsob jak tohoto výsledku docílit je, že se

prvky na počátku algoritmu nevloží do nově vytvořeného pole s alokovanou pamětí o velikosti n, ale pouze dochází k jejich výměně v rámci pole, které je na vstupu. Tato změna oproti variantě out-place vede k značně vyšší režii, díky čemuž dochází k větší časové náročnosti. Z tohoto důvodu slouží tento algoritmus spíše pro edukativní účely (případně nalezne využití tam, kde volné paměti pro algoritmus je velmi málo) a nemá smysl ho upřednostňovat nad out-place variantou.

Pro změnu out-place varianty popsané v předchozí části na verzi alokující paměť navíc pouze o velikosti $\log n$ se princip dělení polí nemění. Naopak v případě slévání polí dochází k větším změnám. Bez újmy na obecnosti předpokládejme, že vstupní data mají velikost n a $\sqrt{n}$ je celočíselná hodnota [5]. Vstupní data si pomyslně rozdělíme na $\sqrt{n}$ bloků $(A_1, \ldots, A_k, B_1, \ldots, B_k)$, z toho poslední dva bloky B_{k-1}, B_k slouží jako pracovní prostor. Cyklus začíná postupným sléváním jednotlivých bloků A_i, B_j do pracovního prostoru. Zpočátku nastavíme $i = 1$ a $j = 1$. Jakmile jsou všechny prvky z některého z A_i, B_j bloku vyprázdněny do pracovního prostoru, pak tento blok vyměníme s pracovním blokem B_{k-1} a hodnotu i nebo j inkrementujeme (pokud byl vyprázdněn blok A resp. B, tak inkrementujeme i resp. j). Následně prohodíme bloky B_{k-1} s B_k (v kterém jsou již nějaká seřazená data). Ve chvíli, kdy tento cyklus skončí, tak posloupnost je tvořena bloky $A_1, \ldots, B_{k-2}$, které jsou uvnitř seřazené ale mezi sebou zatím nikoliv. Poté stačí libovolným a jednoduchým in-place algoritmem seřadit po blocích posloupnost $A_1, \ldots, B_{k-2}$ a také pracovní prostor B_{k-1}, B_k po prvcích. V posledním kroku se seřazená část $A_1, \ldots, B_{k-2}$ a seřazený pracovní prostor seřadí do jednoho výsledného pole.

1.7 Radixsort

Řadícímu algoritmu radixsort se také říká algoritmus číslicového řazení. Algoritmus pracuje s klíči jako s čísly v definované číselné soustavě o základu z. Samotné řazení pak probíhá po jednotlivých číslicích daného čísla, tedy radixsort cíleně dekomponuje klíče na menší části. Nejčastěji se používá tento algoritmus s klíči o číselném základu $z = 2^k$ pro $k \in N$ nebo případně $z = 10$. Klíčem ale nemusí být pouze celočíselná hodnota, ale např. i řetězec (vyhledávání v telefonním seznamu je typickým příkladem pro využití radixsortu nad posloupností písmen).

Rodina nízkourovňových programovacích jazyků (např. C, C++, ...) poskytuje výhodu díky přítomnosti technik pro manipulaci bit po bitu u čísel s $z = 2^k$. To se u tohoto algoritmu využije při dekomponování klíče [3], jelikož stačí dekomponovat klíč pouze na tolik prvních (či posledních) bitů, které jsou zapotřebí k porovnání [1].

Neseřazené hodnoty	Seřazené hodnoty	Postačující klíče
.396465048	.015583409	0
.353336658	.159072306	1590
.318693642	.159369371	1593
.015583409	.269971047	2
.159369371	.318693642	31
.691004885	.353336658	35
.899854354	.396465048	39
.159072306	.538069659	5
.604144269	.604144269	60
.269971047	.691004885	69
.538069659	.899854354	8

Tabulka 1.1: Dekomponování klíče při použití radixsortu.

Tabulka 1.1 ukazuje, že je postačující dekomponovat hodnotu pouze na potřebné číslice pro korektní porovnání celých klíčů. V prvním sloupci se nachází 11 hodnot k seřazení. V dalším sloupci jsou hodnoty již seřazené, povšimněte si ale třetího sloupce, který nám ukazuje, které číslice byly postačující ke správnému porovnání a tedy i k celkovému seřazení. Celkově stačilo porovnat pouze 22 jednoprvkových klíčů namísto všech 99 cifer (11 čísel, každá má 9 cifer) obsažených v číslech.

Algoritmus existuje ve variantách out-place i in-place. V případě implementace varianty out-place je paměťová složitost $n + \mathcal{O}(z)$ na jeden průchod cyklem, kde z je základ, dle kterého řadíme. Ve výrazu v předešlé větě je n kvůli potřebě alokovat pole o velikost n, kam se uloží výstup, a $\mathcal{O}(z)$ je místo pro přihrádky. Průchodů se provede tolik, kolik má nejvyšší číslo číslic dle základu podle kterého řadíme.

Radixsort pro klíče z rozsahu $1, \ldots, r$ není efektivní, pokud r je řádově větší než počet dat k seřazení [20]. V časové složitosti se totiž pak objeví čas potřebný pro inicializaci a procházení všech přihrádek. Pokud ale zapíšeme data k seřazení o vhodně zvoleném základu R, pak se z každého čísla stane k-tice cifer z rozsahu $0, \ldots, z-1$, kde $k = \lfloor \log_z r \rfloor + 1$. Tyto k-tice se pak lexikograficky seřadí, což lze zvládnout k průchody v celkovém čase $\Theta((\log_z r)(n + z))$.

Otázkou je, jak vhodně zvolit základ z. Při zvolení konstanty by časová složitost vyšla $\Theta(n \log r)$, což pro navzájem různé klíče $r \geq n$ není zajímavé, jelikož nedojde k překonání složitosti porovnávacích řadících algoritmů [20]. Naopak při zvolení $z = \Theta(n)$ lze dosáhnout složitosti $\Theta(n \frac{\log r}{\log n})$. Pokud by navíc vstup tvořila polynomiálně velká čísla vzhledem k n ($r \leq n^\alpha$ pro libovolně

zvolené pevné α), poté by platilo $\log r \leq \alpha \log n$ a časová složitost by byla tedy lineární.

1.7.1 In-place varianta

V případě in-place varianty je paměťová složitost pouze $\mathcal{O}(z + M)$, kde z je základ číselné soustavy a M je řád nejvyšší číslice. Tato varianta je mírně složitější na implementaci oproti klasické out-place verzi. Tento algoritmus je stabilní. Této variantě radixsortu se také říká binární radixsort a implementuje se ve variantě MSD. V odborné literatuře je ale také popsán i algoritmus typu in-place ve variantě LSD.

Její princip spočívá ve vytvoření dvou přihrádek. Do první přihrádky se přidávají data zespodu vstupního pole, do druhé přihrádky se data přidávají zeshora ze vstupního souboru. Do první přihrádky se umísťují hodnoty, které mají bit nulový a do druhé přihrádky se umísťují hodnoty s bitem jedničkovým, viz dále.

Průchodů celého algoritmu je tolik, kolik má nejvyšší číslo bitů. Začíná se u nejvíce důležitého bitu a pokračuje se po nejméně významný bit. V každém průchodu se vezme prvek po prvku a porovná se jeho bit na pozici aktuálního průchodu. Pokud je bit roven 1, tak se umístí na konec pole, což odpovídá druhé přihrádce, a počet prvků v druhé přihrádce se inkrementuje o jedničku. V případě, že bit je roven hodnotě 0, pak se prvek nepřemísťuje a počet prvků v první přihrádce se inkrementuje o jedničku.

Takto se porovnávají pouze prvky, které nejsou již umístěné v některé přihrádce. Tento jeden průchod skončí ve chvíli, kdy se ukazatele na obě přihrádky potkají. Pak následují průchody pro pozice ostatních bitů.

1.7.2 MSD typ

U tohoto typu začínáme řadit klíče od nejvíce důležité číslice (angl. most significant digit) - tedy číslice nejvíce vlevo v čísle [1]. Klíč je potřeba rozdělit na $z + 1$ částí. V tomto pomocném poli se na i-té pozici inkrementuje čítač jednotlivých cifer na i-té pozici čísla k seřazení. Po prvním průchodu dojde k naplnění pomocného pole. Následně se provede prefixový součet, což je proces při kterém se v pomocném poli na i-té pozici nastaví hodnota sumy přes všechny prvky j, pro které platí $j <= i$. Díky předešlému úkonu získáme pozice přihrádek, do kterých se následně vloží prvky se stejnou cifrou na dané pozici. Posledním a nejdůležitějším krokem je rekurzivní řazení jednotlivých přihrádek. MSD variantu lze implementovat ve stabilní i nestabilní verzi.

1.7.2.1 Pseudokód

```
function radixsortMSD (pole data, cislo levy, cislo pravy,
 cislo w)
  cislo i, j
  pole citac[R + 1]
  if w > 4 // 4 byty
    konec
  if r - l <= M
    vrat hodnotu insertSort(data, levy, pravy) a konec
  for (j = 0, j < R, j++)
    citac[j] = 0
  for (i = l, i <= r, i++)
    citac[cislice_na_pozici(data[i], w) + 1]++
  for (j = 1, j < R, j++)
    citac[j] += citac[j - 1]
  for (i = l, i <= r, i++)
    aux[citac[cislice_na_pozici(data[i], w)]++] = a[i]
  for (i = l, i <= r, i++)
    data[i] = aux[i - 1];
  radixsortMSD (data, l, citac[0] + l - 1, w + 1)
  for (j = 0, j < R - 1, j++)
    radixsortMSD (data, citac[j] + l,
     citac[j + 1] + l - 1, w + 1)
```

Proměnná `citac` je pole definované v textu jako přihrádka (použitá pro uložení počtů a dělících pozicí), proměnná `aux` je pomocné pole potřebné pro operace radixsortu (konkrétně přesuny klíčů). Proměnná `M` je předem definovaná jako mezní hodnota, kdy se má tento hybridní algoritmus přepnout z radixsortu na insertsort [3].

1.7.3 LSD typ

V případě LSD typu se začíná řadit od nejméně důležité číslice v klíči, tedy číslice nejvíce vpravo v čísle. Algoritmus je MSD algoritmu až na pár drobností podobný. Nedochází v něm k rekurzivnímu volání, nýbrž používá iterativních cyklů k seřazení posloupnosti. Samozřejmě je zapotřebí opět seřadit celé pole, ale při zařazování do přihrádek se postupuje odshora dolů. Je důležité zmínit, že LSD typ vyžaduje pro své korektní fungování, aby byl implementován jako stabilní algoritmus.

1.7.3.1 Pseudokód

```
function radixsortLSD (pole data, cislo levy, cislo pravy)
  cislo i, j, w
```

```
pole citac[R + 1]
for (w = 4 - 1, w >= 0, w--) // 4 byty
  for (j = 0, j < R, j++)
    citac[j] = 0
  for (i = l, i <= r, i++)
    citac[cislice_na_pozici(data[i], w) + 1]++
  for (j = 1, j < R, j++)
    citac[j] += citac[j - 1]
  for (i = l, i <= r, i++)
    aux[citac[cislice_na_pozici(data[i], w)]++] = a[i]
  for (i = l, i <= r, i++)
    data[i] = aux[i - 1];
```

Při tomto algoritmu dochází ke zpracování klíčů zprava doleva po bytech slov. Tento typ algoritmu je ve většině případů rychlejší než-li MSD typ, pokud předpokládáme, že hodnoty k seřazení mají stejný počet cifer. K algoritmu typu MSD musíme také připočíst větší nároky na menších datech díky většímu počtu rekurzivních volání, což zvyší režii.

Kapitola 2

Paralelizace algoritmů

Dosud jsme pojednávali o sekvenčních verzích algoritmů, tedy že kroky jsou vykonávány v sérii za sebou na jedné výpočetní jednotce. Tento algoritmus lze po sérii úprav (tomuto kroku se říká paralelizace algoritmu) upravit na paralelní verzi algoritmu, kterou lze efektivně spouštět na více jak jedné výpočetní jednotce. Výpočetní jednotkou mohou být jak jádra obsažená v procesoru, tak i jádra obsažená v grafických kartách. Důvodem paralelizace jakýchkoliv algoritmů je snaha o zrychlení doby běhu programu. Potenciální zrychlení z nejlepšího sekvenčního řadícího algoritmu s asymptotickou časovou složitostí je z $\mathcal{O}(n \log n)$ na $\mathcal{O}(\log n)$ při n výpočetních jádrech [4]. Těmto hodnotám se snažíme přiblížit při paralelizaci řadících algoritmů, ale je zapotřebí si uvědomit že těchto hodnot nelze v praxi dosáhnout, protože není reálné, abychom měli k dispozici tolik procesorů kolik je prvků k seřazení.

Prvním krokem k úspěšné paralelizaci sekvenčního kódu je nalezení té části kódu, která může být spuštěna souběžně na více procesorech. Někdy to může být relativně jednoduchý úkol, někdy naopak musí programátor svůj kód pozměnit, aby získal nezávislou sekvenci instrukcí, kterou lze paralelizovat. Může se jednat o část kódu nebo v krajním případě může být zapotřebí i zvolit jiný algoritmus řešící stejný problém, protože ten původní není vhodně paralelizovatelný. Naštěstí existuje plno různých strategií pro paralelizaci, které lze často použít u sekvenčních algoritmů. Tyto strategie jsou velmi dobře popsané v následujících publikacích [32] [33] [34] [35].

U paralelizace řadících algoritmů je možné si vybrat ze dvou způsobu, kterou oblast budeme paralelizovat. První alternativa patří do kategorie *task-parallelism* a jde o jakousi dělbu práce mezi jádra. Tedy každé vlákno dostane na starost nějakou úlohu a tu provádí nad celým vstupem. Druhou možnou kategorií je *data-parallelism*, v kterém je zásadní rozdělení vstupních dat ke zpracování mezi jádra.

Celý tento problém si lze dobře představit např. jako opravování písemné práce s pěti úlohami z matematiky pěti učiteli. V případě rozdělení práce (task-parallelism) každý n-tý učitel opravuje n-tý příklad u každé práce. Ve druhém případě (data-parallelism) si učitelé rozdělí všechny písemné práce rovným dílem a následně každý učitel opravuje všechny příklady ze své množiny přidělených písemných prací. Tyto dva typy paralelizace lze zkombinovat a v některých případech tak dosáhnout ještě lepší paralelizace.

Toho se využívá například při náročných matematických výpočtech sloužících k předpovědi počasí. Vstupními daty je rozsáhlá soustava rovnic a ta je rozdělena do mřížek obsahující naměřené atmosférické údaje. Pomocí datového paralelismu je rozdělena mezi určitou část procesorových jednotek. Pro zbytek výpočetních jednotek zbývají pomocí úkolového paralelismu různé simulace a modelování fyzikálních přírodních procesů.

V následujících podkapitolách definuji PRAM model a stručně shrnu implementační nástroje, kterými lze algoritmy (jak obecné tak i řadící) paralelizovat.

2.1 Výpočetní model PRAM

V kapitole 1.1 jsme si definovali sekvenční výpočetní model RAM, z kterého vychází výpočetní model PRAM (Parallel Random Access Machine). Narozdíl od RAM modelu, který se skládá z jedné výpočetní jednotky, obsahuje PRAM model neomezený počet procesorů RAM. Paměť je tvořena neomezeným počtem sdílených paměťových buněk. Navíc má každý procesor RAM svou lokální paměť o neomezené velikosti. Každý procesor může přistupovat do libovolné sdílené paměťové buňky v jednotkovém čase. Instrukce tohoto modelu lze rozdělit do tří typů:

- čtení dat
- provedení lokálního výpočtu
- zapsání dat

Tyto instrukce jsou prováděny synchronně. Konflikty při čtení nebo zápisu při souběžném přístupu jsou ošetřeny následujícími podmodely PRAM modelu [19]:

- Exclusive Read Exclusive Write (EREW) PRAM: Žádným dvěma procesorům není povoleno číst z nebo zapisovat do stejné buňky sdílené paměti najednou.

- Concurrent Read Exclusive Write (CREW) PRAM: Dvěma různým procesorům je dovoleno najednou číst z jedné paměťové buňky, ale do jedné paměťové buňky smí zapisovat pouze jeden procesor.
- Concurrent Read Concurrent Write (CRCW) PRAM: Je dovolené souběžné čtení z i zápis do jedné paměťové buňky pro dva různé procesory najednou.
  - Prioritní CRCW: Každému procesoru, který chce zapisovat, je přidělena priorita. Pouze procesoru s nejvyšší prioritou je povolen zápis.
  - Náhodný CRCW: Zápis je povolen náhodně vybranému procesoru. Algoritmus nesmí činit žádné předpoklady o tom, který procesor bude náhodně vybrán.
  - Shodný CRCW: Je povolen zápis více procesory, pokud chtějí zapsat stejnou hodnotu do paměťové buňky. Tuto podmínku musí zajistit algoritmus, v opačném případě není stav počítače definován a jedná se o chybu.

2.2 OpenMP

Knihovna OpenMP byla představena roku 1997 na IT konferenci *High Performance Computing, Networking, and Storage* v San Jose, Kalifornia, USA jako nový multiplatformní programovací nástroj pro efektivní využití sdílené paměti paralelních počítačů [6]. Do té doby se využívalo různých technik pro práci se sdílenou pamětí a představení OpenMP komunitě IT odborníků přišel ve správný čas, kdy se hledalo jednotné řešení pro problematiku programování nad sdílenou pamětí. Společnost OpenMP Architecture Review Board byla následně založena a měla na starost vývoj OpenMP. Počet společností přispívajících specifikacemi a svými znalostmni do této organizace rostl. V dnešní době lze využít výhod této techniky v jazycích Fortran, C a C++. Stolní počítace i dokonce notebooky jsou dnes navrhovány s vícejádrovými procesory schopnými zpracovávat více vláken v jednom čase. Úprava současných sekvenčních programů pro efektivní běh nad více jádry je právě možná díky např. této technologii OpenMP.

OpenMP je API pro práci se sdílenou pamětí, umožňující vývojářům pracovat snadněji a efektivněji vyvíjet aplikace paralelně běžící nad CPU. API je stále vyvíjenou platformou, tak aby byla aplikovatelná na široké škále různých SMP systémů. Toto rozhraní není novým programovacím jazykem, ale chová se jako nadstavba pro programovací jazyky Fortran, C a C++. Aplikuje se přidáním speciálních direktiv do zdrojového kódu sekvenčního algoritmu. Mnoho takových algoritmů lze alespoň bez větší námahy triviálně paralelizovat pro menší zrychlení.

OpenMP direktivy přidané do zdrojového kódu říkají kompilátoru, která část instrukcí má běžet paralelně a jak budou distribuovány mezi jednotlivá vlákna, které budou zpracovávat binární kód. Tyto direktivy jsou zapisovány ve speciální formě začínající textem `#pragma omp`. Při kompilování kódu pomocí gcc nebo g++ musíme kompilátoru dát na vědomí, že zdrojový kód obsahuje OpenMP direktivy pomocí parametru *-fopenmp*. To má za pozitivní následek, že pokud kompilátor spustíme bez tohoto přepínače, tak by se měl program úspěšně přeložit bez paralelizace a algoritmus by měl korektně fungovat v zkompilované sekvenční verzi.

Velkou výhodou OpenMP je možnost postupné paralelizace sekvenčního kódu. Předpokládejme, že máme sekvenční verzi. Provedeme nějakou triviální paralelizaci nad kódem. Změříme dobu výpočtu oproti sekvenční verzi a otestujeme funkčnost algoritmu (měl by dávat stejný výstup jako čistě sekvenční verze). Pokud je doba výpočtu kratší a algoritmus korektní, tak můžeme označit tuto verzi aplikace za dobrou a můžeme pokračovat v *netriviální* paralelizaci některých dalších částí algoritmu (např. partitiong u quicksortu nebo multiway merge u mergesortu).

2.3 Paralelizace na GPU

Není tomu tak dlouho, co výkon grafické karty byl využíván převážně k vykreslení počítačové grafiky nebo u počítačových her. Nyní, kdy je výkon grafických karet opravdu veliký, je možné využít GPU pro práci jako je například řazení dat. Toto si uvědomila společnost nVidia a v roce 2006 uvedla na trh novou architekturu grafických karet a k nim prostředí CUDA, které umožňuje programovat aplikace v jazyce C a C++. V roce 2008 byl uvolněn implementační nástroj OpenCL, který není závislý na výrobci resp. architektuře grafické karty. Jedná se o heterogenní nástroj, což v tomto konkrétním příkladě znamená, že výsledný program lze spustit na CPU, GPU, APU nebo DSP. OpenACC je nástroj, který umožňuje také heterogenní paralelní programování, konkrétně pro běh nad CPU a GPU.

Grafické jednotky se skládají z bloků jader, na kterých běží paralelně vlákna. V každém tomto bloku běží přes všechna jádra totožná sekvence po sobě jdoucích instrukcí ve stejnou chvíli. Díky této vlastnosti mají algoritmy běžící nad GPU potenciál mít kratší dobu běhu nežli stejné algoritmy implementované s během nad CPU. Zároveň je ale potřeba počítat s přičtením doby kopírování dat dovnitř a ven z paměti grafické karty, což běh algoritmu pod grafickou kartou na druhou stranu mírně zpomalí oproti CPU verzi.

2.3.1 OpenCL

Standard OpenCL umožňuje běh na různých druzích procesorů, počínaje od klasických CPU a GPU, tak i např. na Cell procesorech. Tato platforma definuje abstraktní hardwarový model společně se softwarovým rozhraním. Pomocí tohoto rozhraní může program přistupovat ke konkrétním výpočetním možnostem z různých hardwarových platforem, díky čemuž vzniká velká variabilita využití. Podpora programování v jazyce *OpenCL C* je pro všechny majoritní platformy – Windows, Linux, macOS, dokonce i pro iOS.

Jsou definovány 4 různé modely, které jsou abstraktním vyjádřením vlastností a chování platforem. Prvním modelem je platformový, který popisuje paralelní stroj jako systém, který obsahuje hostitelský systém a OpenCL zařízení. Vykonávací (někdy také nazývaný jako exekuční) model se skládá z hostitelského programu a kernelů, což jsou funkce které jsou spouštěny paralelně pomocí technologie OpenCL na kompatibilních zařízeních. Třetím modelem je paměťový, který popisuje paměťový prostor na hostu a na OpenCL zařízení. Posledním čtvrtým modelem je programovací, který popisuje data-parallelism a task-parallelism obdobně jako jsou popsány na začátku této kapitoly.

2.3.2 OpenACC

Tento programovací standard pro paralelní programování byl vyvinut společnostmi Cray, CAPS, nVidia a PGI. Jak již bylo řečeno v úvodu této podkapitoly, tak jeho primárním účelem je ulehčení programování v heterogením prostředí. Podobně jako u nástroje OpenMP i zde dochází k anotaci zdrojové kódu v jazycích C, C++ a Fortran pomocí direktiv, čímž se označí, která část kódu má běžet paralelně. Paralelizace je možná na všech třech majoritně rozšířených grafických kartách - nVidia, AMD i integrované GPU od Intelu v CPU.

Podpora pro OpenACC byla přidána do kompilátoru *gcc* ve verzi 5.1, jednalo se však o experimentální podporu. OpenACC ve verzi 2.0 bylo přidáno do *gcc* verze 6 a 7, ta má již značně lepší implementaci.

2.4 CUDA

Compute Unified Device Architecture neboli CUDA je technologie vyvinutá společností nVidia sloužící k využití výpočetních jader na grafických kartách nVidia pro programátory. Tyto jádra jsou o hodně jednodušší nežli jádra CPU, co se týče výpočetních schopností, ale jejich síla tkví ve vysokém počtu. Pro lepší představu uvádím v tabulce 2.1 počet jader na high-endových grafických kartách od společnosti nVidia. Pro využití tohoto systému nestačí pouze

Tabulka 2.1: High-end grafické karty od společnosti nVidia

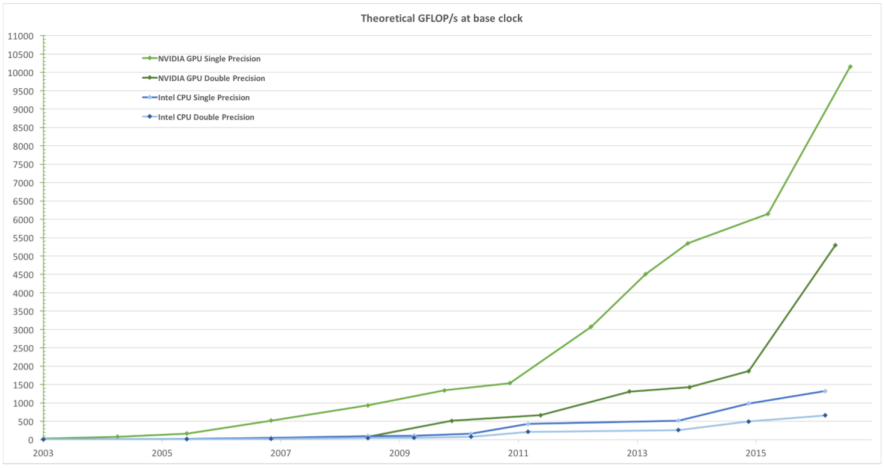
Označení grafické karty	Počet CUDA jader
nVidia 9800 GX2	256
nVidia GTX 295	480
nVidia GTX 480	480
nVidia GTX 590	1024
nVidia GTX 690	3072
nVidia GTX 780 Ti	2880
nVidia GTX 980 Ti	2816
nVidia GTX-1080 Ti	3584
nVidia GTX Titan Z	5760

hardwarové zázemí (grafická karta obsahující CUDA jádra), ale i nainstalovaný software CUDA Toolkit, který zpřístupňuje programátorovi potřebnou knihovnu v jazycích C a C++. Tato technologie byla představena v roce 2006 a je stále aktivně vyvíjena a současná verze je 9.0 (ke dni 26.10.2017).

Obrázek 2.1: Grafické znázornění rozdílů mezi CPU a GPU [9]

Technologie CUDA se používá v heterogenním prostředí. Bez CPU nelze zinicializovat výpočetní jádra GPU, nahrát do paměti grafické karty data a provádět výpočty. Proto se jedná o heterogenní prostředí, v kterém CPU nazýváme *host* a GPU nazýváme *device*. Při vývoji aplikace se snažíme o to, aby sekvenční část běžela na CPU a paralelní část na GPU. Na dnešních čtyřjádrových procesorech může běžet zároveň až 16 vláken a v případě aktivované technologie Hyper-threading až 32 vláken [8]. Moderní GPU zvládnou až 1536 aktivních vláken na jeden multiprocesor. Na grafických kartách o 16 multiprocesorech SM tedy může běžet až 24000 souběžně běžících vláken. Obrázek 2.1 znázorňuje rozdíl mezi CPU a GPU v oblasti hardwarové architektury. Grafické karty obsahují více aritmetických jednotek (ALU) sloužících k výpočtu a proto je nazýváme mnohojádrovou architekturou s větší propustností a

vyšším výkonem v oblasti výpočtu s plovoucí desetinnou čárkou oproti CPU, což velmi pěkně znázorňuje obrázek 2.2.

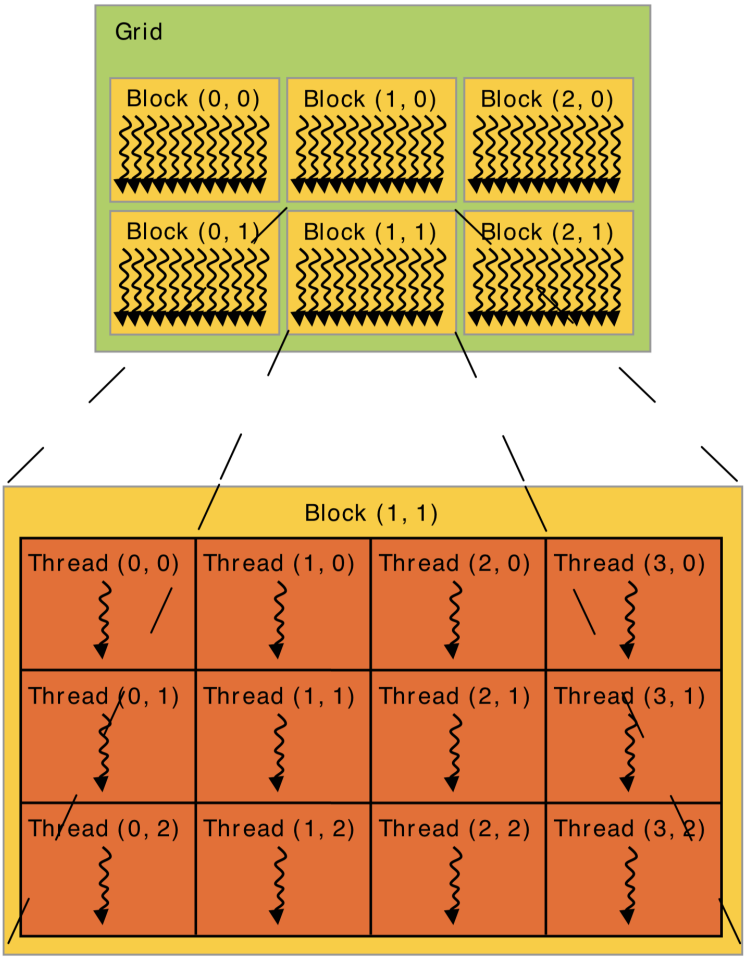


Obrázek 2.2: Počet operací s plovoucí desetinnou čárkou za sekundu u CPU a GPU [9]

2.4.1 Vývoj aplikace

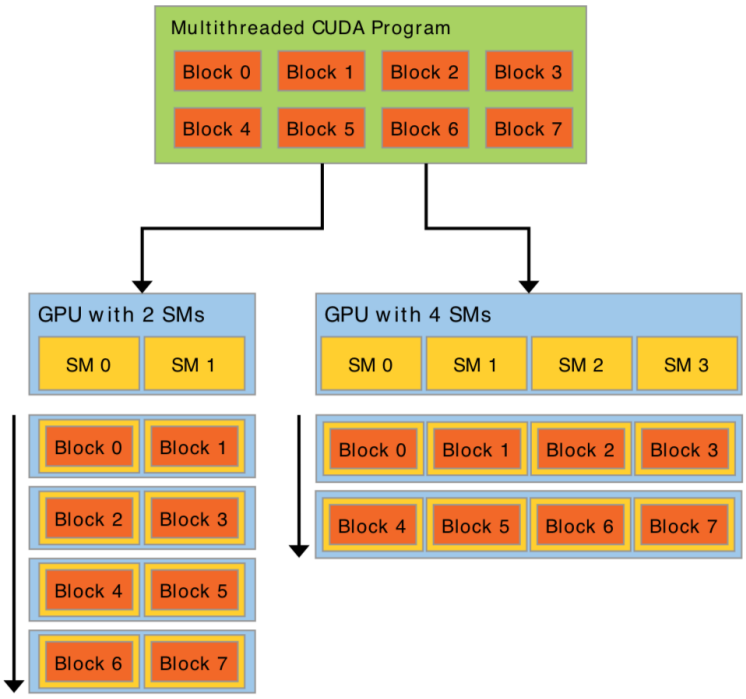
CUDA SDK rozšiřuje jazyk C/C++ o možnost definovat speciální funkci, které se říká kernel. To je ta část, která je spouštěna paralelně na GPU a volána z CPU (od CUDA v5.0 může být volána tato funkce i z GPU). Tuto funkci poznáme jednoduše tím, že vždy začíná klíčovým slovem `__global__` a v závorkách, které následují, $<<<\cdots>>>$ se označí pomocí proměnné typu *dim3* velikost mřížky a bloku. Druhým speciální typem je funkce označená klíčovým slovem `__device__`. Ta smí také běžet pouze na GPU a může být volána pouze z GPU. Třetím typem zajímavé funkce je taková, která může běžet pouze na CPU a může být volána také pouze z CPU a ta je označena klíčovým slovem `__host__`. Pokud není funkce označena ani jedním z těchto klíčových slov, pak je CUDA překladačem *nvcc* považována za hostitelskou (`__host__`) a nelze ji tedy spustit pod GPU.

Spuštěná vlákna pod GPU jsou organizována do 1D, 2D nebo 3D bloků. Každé z nich lze jednoznačně identifikovat podle `threadIdx` proměnné. V rámci bloku lze vlákna synchronizovat pomocí funkce `__syncthreads()`. Maximální počet spuštěných vláken v rámci jednoho bloku je 1024, což je hodnota dnešní hardwarové limitace. Bloky se podobně jako vlákna uspořádávají do 1D, 2D nebo 3D objektů, kterým říkáme mřížky (angl. grid). Každý blok je v rámci své mřížky jednoznačně identifikován pomocí proměnné `blockIdx`. Obrázek 2.3 ukazuje možné rozložení mřížky na bloky a rozložení bloků na vlákna.



Obrázek 2.3: Příklad rozložení bloků v mřížce a vláken v bloku [9]

Dnešní grafické karty od nVidia obsahují více tzv. multiprocesorů, které se také označují jako SMs. Když je CUDA program spuštěn na CPU a je zavolán kernel, pak jsou bloky z mřížky rozdistribuovány mezi takové multiprocesory, které zrovna neprovádějí žádnou práci a jsou volné. Vlákna v rámci jednoho bloku jsou spouštěna paralelně na jednom konkrétním multiprocesoru. Stejně tak i více bloků (a k nim přiřazená vlákna) mohu být spouštěna paralelně na jednom konkrétním multiprocesoru. Jakmile blok vláken dokončí svou práci, pak na tomto multiprocesoru mohu začít běžet nové bloky s další výpočetní úlohou. S větším počtem multiprocesorů se zkracuje doba běhu CUDA aplikace, jak ukazuje obrázek 2.4.



Obrázek 2.4: Příklad běhu programu pod GPU s různým počtem multiprocesorů [9]

2.4.2 Typy pamětí

Grafická karta nVidia s CUDA funkcí obsahuje celkem 6 různých pamětí. Tyto typy se liší druhem paměti, typem přístupu, umístěním, výčtem možných operací nad danou pamětí i tím zdali obsahují cache. Tyto typy pamětí bych popsal následovně dle specifikace *nVidia CUDA Compute Capability 2.x* [9]:

- globální – pomalejší paměť včetně cache, podporuje operace čtení i zápis, mohou k ní přistupovat všechna vlákna z GPU i host a je umístěná na DRAM,
- sdílená – rychlejší paměť bez cache, náchylná na konflikty v paměťovém prostoru, podporuje operace čtení i zápis, mohou k ní přistupovat všechna vlákna v rámci jednoho bloku z GPU a je umístěná na čipu,
- lokální – používaná pro data, která se nevejdou do registrů, je součástí globální paměti, pomalejší paměť včetně cache, podporuje operace čtení i zápis, může k ní přistupovat pouze jedno vlákno GPU a je umístěná na DRAM,
- registry – velmi rychlá paměť bez cache, podporuje operace čtení i zápis, může k ní přistupovat pouze jedno vlákno GPU a je umístěná na čipu,

- texturovací – paměť s cache optimalizovanou pro 2D přístupy, podporuje pouze operaci čtení, mohou k ní přistupovat všechna vlákna z GPU i host a je umístěna na DRAM,
- pro konstanty – pomalejší paměť s cache, místo kam se ukládají konstanty a argumenty kernelu, podporuje pouze operaci čtení, mohou k ní přistupovat všechna vlákna z GPU i host a je umístěna na DRAM.

Kapitola 3

Paralelizace řadících algoritmů pomocí OpenMP

V této kapitole budou vysvětleny a popsány techniky pro efektivní paralelizaci sekvenčních řadících algoritmů za pomocí technologie OpenMP. Zároveň budou zahrnuty okomentované grafy znázorňující výsledné zrychlení.

3.1 Quicksort

Sekvenční verze tohoto řadícího algoritmu je řešena pomocí rekurzivního volání. Pro paralelizaci je tedy vhodné použít direktivu `omp task`.

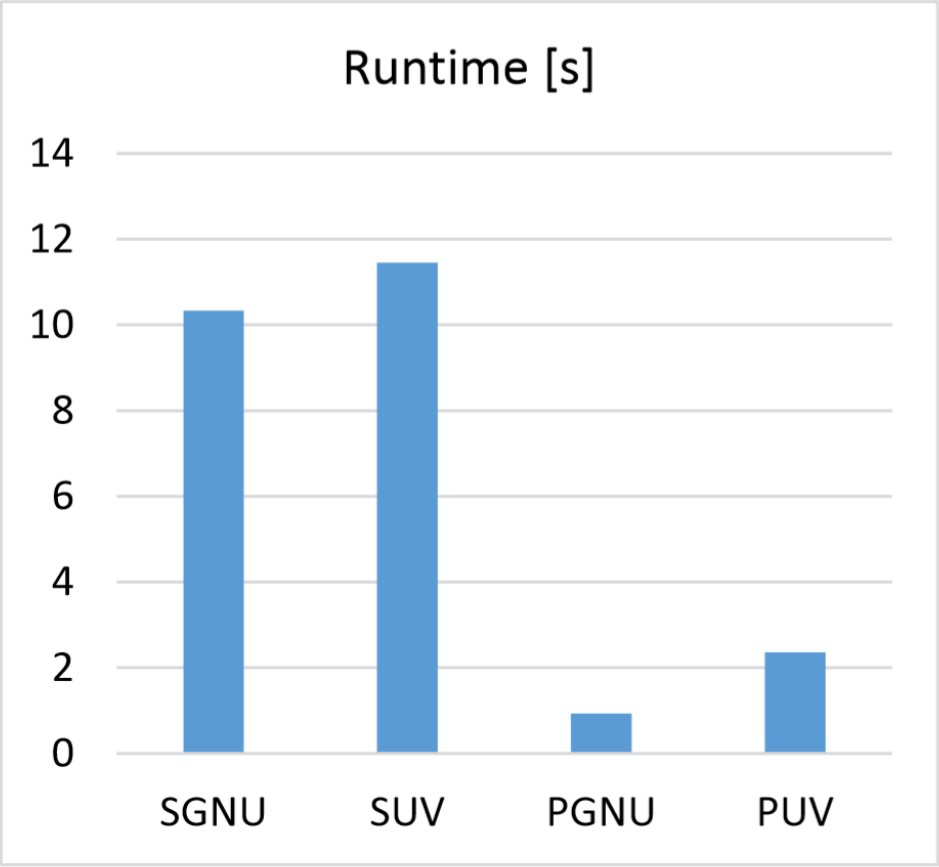
Prvním krokem paralelizace quicksortu je přidání funkce, pojmenujme ji např. `quicksortOMP()`, kde celý její vnitřek obalíme pomocí direktivy `omp parallel`, tak aby se vytvořila vlákna. Uvnitř této paralelní části zavoláme jednovláknově (pomocí direktivy `omp single` první úroveň (neboli kořen rekurze) naší hlavní funkce, která vykonává samotné řadící operace, kterou jsem pojmenoval ve své implementaci `quicksortImplOMP()`. Výsledná funkce by tedy vypadala následovně.

```
void quicksortOMP(T * a, T * b) {
#pragma omp parallel
  {
#pragma omp single
    quicksortImplOMP(a, b);
  }
}
```

V hlavní funkci `quicksortImplOMP()` se nejdříve provede sekvenční funkce `rozdělení` a uložení pozice pivota a následně se funkce zavolá rekurzivně na levou část a hned poté na pravou část pole. K paralelizaci přímo vybízí obě

rekurzivní volání. Tedy pro aplikaci funkčního paralelismu stačí přidat direktivy `omp task` před dvojím rekurzivním voláním této funkce jelikož se jedná o datově nezávislé úlohy. Tímto jsme dospěli do stavu, kdy algoritmus běží paralelně. Jeho nízkou efektivitu lze vidět na obrázku 3.1, kde jsou porovnány časové náročnosti quicksort algoritmu pomocí čtyř různých implementací nad daty o velikosti 10^9 celočíselných náhodných hodnot:

- sekvenční verze (SGNU) z libstdc++,
- paralelní verze (PGNU) z libstdc++,
- sekvenční učebnicová (z Wikipedie) verze (SUV) a
- paralelní verze (úpravy výše uvedené nad SUV verzí).



Obrázek 3.1: Porovnání přímočaré paralelizace quicksortu s jinými implementacemi [14]

Také je vhodné zdůraznit, že v knihovně libstdc++ se nenachází čistá implementace algoritmu quicksort (respektive mergesort v další sekci této práce), ale algoritmy jsou implementované jako hybridní – ve správnou chvíli se přepnou, aby nedocházelo k neefektivnímu častému volání rekurze nad malým množstvím dat, a místo toho pokračují pomocí jiného algoritmu s menšími konstantami. Takové algoritmy se chovají lépe na menších datech.

Toto a některá dále popsána experimentální měření byla provedena na procesoru Intel Xeon E5-2680 v3 (Haswell) s 12 jádry s GNU GCC verze 5.4 [14] a byla předvedena p. Ing. Danielem Langrem, Ph.D. na přednáškách z předmětu *Paralelní a distribuované programování* z magisterského oboru na Fakultě informačních technologií na ČVUT V Praze. Implementace PUV je 2.27-krát pomalejší nežli PGNU verze.

Existují celkem tři možné vylepšení, jak tuto paralelní verzi zdokonalit, aby dosahovala téměř podobných výsledků jako PGNU verze [14]:

- odstraněním nadbytečného vytváření úloh a volání rekurze,
- zavedením prahu (hybridní algoritmus) a
- paralelizací části, která má na starost rozdělení pole na dvě menší pole dle hodnoty pivota (partitioning).

Jednotlivé možnosti optimalizace popíši v následujících částech.

3.1.1 Odstranění nadbytečného vytváření úloh a volání rekurze

Druhé rekurzivní volání můžeme upravit tak, aby místo vytvoření nového *tasku* se provedla práce v soucašném vlákně. Odstraněním rekurzivního volání funkce se částečně zbavíme určité režie (předávání parametrů, vytváření místa na zásobníku, volání call instrukce, ...), která nám navyšovala dobu běhu algoritmu.

Proto se ponechá direktiva `omp task` pouze před prvním rekurzivním voláním. Druhé rekurzivní volání funkce nahradíme vhodně iterací a celý algoritmus vložíme do cyklu, jak demonstruje následující pseudokód:

```
quicksortImplOMP(a, b) {
  while (a < b) {
    pivot = partitioning(a, b);
#pragma omp task
    quicksortImplOMP(a, pivot - 1);
    a = pivot + 1;
```

```
  }
}
```

Díky těmto dvěma optimalizacím (odstranění druhého volání rekurze a vložení algoritmu do `while` cyklu s patřičnými úpravami) došlo ke snížení počtu vytváření OpenMP úloh z $\mathcal{O}(n)$ na $\mathcal{O}(\log^2 n)$, ke stejnému snížení došlo i u počtu volání rekurzivních funkcí [14].

3.1.2 Zavedení prahu

Dalším krokem paralelizace je vytvoření prahu v rekurzivním volání na konkrétní úrovni rekurzivního stromu, od kterého se bude již volat pouze sekvenční verze quicksortu v současném vlákně. Otázkou je, jak správně nastavit hodnotu prahu, aby zátěž vláken byla vyvážená. Jednou z možností, na jakou hodnotu nastavit mez, je $\frac{n}{p}$, kde n je délka vstupních dat (v našem případě $(b-a)$) a p je počet procesorů.

Z důvodu, že může i přesto přetrvávat problém s vyvážením vláken, pak je vhodnější hodnotu prahu vydělit konstantou k. Konstantu k zvolíme vhodně na základě empirického měření. Výsledný práh má tedy tvar $\frac{n}{kp}$. Výpočet tohoto výrazu je vložen do funkce `quicksortOMP()`. Z důvodu závislosti výrazu na počtu procesorů je zapotření vložit výpočet do paralelní části, ale samozřejmě před direktivu `omp single`, jinak by $p = 1$, což je nežádoucí.

Poté již stačí předefinovat funkci `quicksortImplOMP()`, tak aby akceptovala i třetí parametr a to tento vypočtený práh. V případě, že nechceme modifikovat funkci, tak můžeme z proměnné obsahující hodnotu prahu vytvořit globální proměnnou. Tato proměnná je totiž konstantou, její hodnota se vypočte jednou na začátku a poté se již nemění. Výpočet prahu může tedy vypadat např. takto:

```
const long seq_thr = (b - a + 1) / omp_get_num_threads() / k;
```

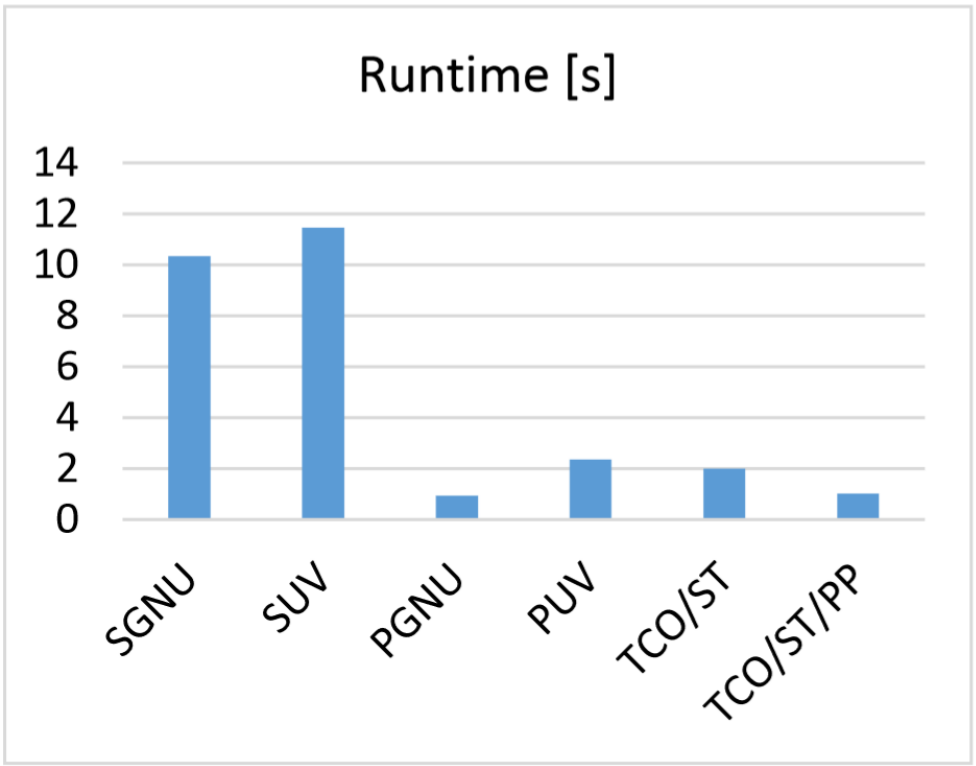
Poté již stačí na začátku smyčky porovnávat, zdali velikost dat (v našem případě $(n = b-a)$) je menší nežli hodnota námi vypočteného prahu. A pokud je opravdu menší, tak tyto data seřadíme sekvenčním algoritmem a následně současné vlákno ukončíme, aby nepokračovalo v paralelním řešení.

3.1.3 Paralelizace rozdělení pole na dvě části

V sekvenční verzi ve funkci `rozdělení` si nastavíme proměnou `i` na pozici prvního prvku a proměnnou `j` na pozici posledního prvku v poli. Následně inkrementujeme `i` a dekrementujeme `j` v cyklu, tak že se blížíme ke středu pole, dokud se prvky nestřetnou. V každém kroku cyklu porovnáváme prvky

na pozicích `i` a `j` s pivotem a dle algoritmu popsaném na začátku této práce tyto prvky případně prohodíme, tak aby výsledné pole obsahovalo v levé části prvky menší nežli pivot a v pravé části prvky větší nebo rovno pivotu.

Tento sekvenční algoritmus lze paralelizovat tak, že indexy `i` a `j` se stanou sdílenými proměnnými pro jednotlivá vlákna. Každé vlákno si bude nárokovat vlastní indexy `my_i` a `my_j` a na těchto indexech bude porovnávat hodnoty vůči pivotu. Aby vlákna řadila na disjunktních množinách dat, tak postačí, aby k nárokování nově inkrementované hodnoty `i` nebo `j` docházelo pomocí atomické operace, konkrétně pomocí direktivy `#omp atomic capture`, která získá hodnotu a inkrementuje ji. Při konci práce každého vlákna může dojít ke stavu, kdy nejvýše jeden prvek bude špatně zařazen. Počet nezařazených prvků tedy bude nejvýše p a to lze již triválně dořešit pomocí sekvenční verze algoritmu quicksort.



Obrázek 3.2: Porovnání vylepšené paralelizace quicksortu s jinými implementacemi [14]

Kvůli tzv. false-sharing [38] bude docházet ve výše popsaném modelu paralelizace k častému přepisování dat, které se nachází v cache paměti jednotlivých jader. Z důvodu zachování konzistence paměti zakročí některý z koherenčních protokolů [11] a při zápisu jedním z procesorů do bloku paměti, který mají načtený v cache i ostatní procesory, zneplatní tento blok všem ostatním procesorům a ty si budou muset tento blok načíst znovu s nejnovějšími daty.

Z tohoto důvodu je inkrementace `i` resp. dekrementace `j` čítačů po jednom neefektivní a je lepší ji provádět po blocích o velikosti K, tak aby byla zachována disjunktnost množin nad kterými budou jednotlivá vlákna pracovat. Velikost bloku K je vhodné zvolit na hodnotu velikosti tzv. cache-line [39] na daném procesoru (např. 64 bajtů). Po doseřazení pomocí blokové verze může, obdobně jako u verze po prvcích, zůstat nanejvýš p bloků neseřazených. U nich opět zjistíme, zdali jsou neseřazené a pokud jsou, tak finální řazení nad nimi můžeme opět provést např. pomocí sekvenční verze algoritmu quicksort.

Zároveň může nastat situace, kdy velikost dat k seřazení n nemusí být dělitelná velikostí bloku K a proto nám nakonci množiny dat může zbývat část, která bude naprosto netknutá a neseřazená tímto algoritmem. Pokud takovýto konec existuje, tak ho můžeme opět seřadit např. sekvenčním quicksortem. Důležité je také zmínit, že tento algoritmus vyžaduje aktivovaný vnořený paralelismus pomocí funkce `omp_set_nested()`.

Výsledné zrychlení se všemi těmito úpravami lze shlédnout na obrázku 3.2. V tomto grafu nás zajímá hodnota sloupečku s názvem PGNU (paralelní quicksort z libstdc++) a TCO/ST/PP, který odpovídá všem popsaným úpravám popsaných v předcházejících odstavcích. Jak je vidno z grafu, takto paralelizovaná verze má již lepší hodnoty v oblasti doby běhu algoritmu a rychlostí se přibližuje ke knihovní verzi.

3.2 Mergesort

Podobně jako u quicksortu je i tento algoritmus řešen pomocí rekurzivního volání sebe sama. Z tohoto důvodu se využije opět direktiva `omp task` pro efektivní paralelizaci a tímto způsobem se rozdělí práce mezi jednotlivá vlákna.

Pro lepší orientaci ve funkcích pokračuji v názvosloví použitém v minulé podkapitole, tedy k názvům funkcí přidám *OMP*. Dojde tedy k pojmenování funkce obalovací `mergesort0()` (wrapper function) a `mergesort0impl()` na `mergesort00MP()` a `mergesort0implOMP()`, aby bylo zřejmé, že se jedná o paralelní verze algoritmu a navíc, aby bylo zřetelné, který algoritmus je rekurzivně volán, když se v určeném prahu bude volat sekvenční nebo paralelní varianta.

U funkce `mergesortOOMP()` budeme postupovat podobně jako u quicksortu. A to tedy tak, že funkci obalíme do paralelní části a implementační funkci (což je vlastně první úroveň rekurze) zavoláme jednovláknově. Proměnná `tmp` nám předesílá, že se jedná o out-place variantu algoritmu.

```
template <typename T>
void mergesortOOMP(T * a, T * b) {
  T * tmp = new T[b-a];
#pragma omp parallel
  {
#pragma omp single
    mergesortOimplOMP(tmp, a, b-a);
  }
  delete[] tmp;
}
```

Výše popsané změny se týkají obalovací funkce. Nyní popíšu jednotlivé metody paralelizace uvnitř implementační funkce. Přímočará a triviální paralelizace by vypadala následovně.

```
#pragma omp task
  mergesortOimplOMP(tmp, a, mid);
#pragma omp task
  mergesortOimplOMP(tmp + mid, a + mid, length - mid);
```

Za touto částí kódu následuje volání funkce pro slévání dvou polí dohromady. V době, kdy tuto funkci zavoláme, potřebujeme, aby obě pole, které budeme slévat dohromady, byla již seřazená. To je důvod, proč v případě paralelní verze je zapotřebí počkat na dodělání obou úloh pomocí direktivy `omp taskwait` před zavoláním funkce, která má na starost slévání. Tedy kód uvedený výše by nevedl ke korektním výsledkům a můžeme ho označit za chybný. Pokud tuto chybu opravíme přidáním direktivy `omp taskwait` mezi druhou volanou rekurzi a voláním funkce pro slévání, pak dostaneme korektní a předpokládané chování algoritmu mergesort.

Pokud takto upravený kód i změříme nad stejnými daty jako u quicksortu, tak dostaneme zajímavé výsledky, viz tabulka 3.1 [14]. Důvodem značného zpomalení námi implementované přímočaré implementace vůči čistě sekvenční verzi tkví ve spouštění velkého množství OpenMP úloh, které slučují malé posloupnosti dat, což je zbytečné. Navíc tyto úlohy pracují nad daty, které jsou zarovnány v paměti vedle sebe, díky čemuž dochází k ukázkovému *false-sharing*, které není žádoucí. Existují celkem tři způsoby jak výše uvedenou přímočarou paralelizaci opravit, tak aby byla efektivnější [14]:

- zavedením prahu pro velikost dat k seřazení,

Tabulka 3.1: Porovnání přímočaré paralelizace quicksortu a mergesortu s knihovní verzí [14]

	QuickSort	**MergeSort**
SGNU	10,33 [s]	10,77 [s]
PGNU	0,94 [s]	0,98 [s]
SUV	11,46 [s]	13,86 [s]
PUV	2,36 [s]	198,46 [s]

- vytvářením polovičního počtu nových OpenMP úloh a
- paralelizací funkce pro slévání.

3.2.1 Zavedení prahu

Prvním zrychlením přímočaré paralelizace je nastavení prahu pomocí mezní hodnoty vůči velikosti dat, které zbývají k seřazení. Tuto mezní hodnotu jsem nastavil empirickým měřením na hodnotu 10^6 pro mou množinu testovaných dat. Pokud v aktuální úrovni rekurze je velikost pole k seřazení větší nežli tato mezní hodnota, pak se pokračuje ve volání paralelní funkce. V opačném případě se algoritmus přepne na sekvenční variantu mergesortu. Důvodem zavedení prahu je, že při menším množství dat by se zbytečně vytvářelo mnoho OpenMP úloh, které by měly na starost seřadit malé množsví dat. Režie pro vytvoření a správu těchto úloh je časově náročnější, než když je daná množina dat seřazena sekvenčně.

```
if (length >= 1000000) {
#pragma omp task
  mergesortOimplOMP(tmp, a, mid);
#pragma omp task
  mergesortOimplOMP(tmp + mid, a + mid, length - mid);
#pragma omp taskwait
} else {
  mergesortOimpl(tmp, a, mid);
  mergesortOimpl(tmp + mid, a + mid, length - mid);
}
```

3.2.2 Redundantní volání úloh

Obdobně jako u mnou popsané implementace paralelní verze quicksortu lze i zde nechat zpracovávat polovinu práce současným běžícím vláknem, je tedy zbytečné pro druhou polovinu vytvářet nové OpenMP úlohy. Tím se ušetří nemalé množství času na režii. Výsledný kód by vypadal následovně.

```
if (length >= 1000000) {
#pragma omp task
  mergesortOimplOMP(tmp, a, mid);
  mergesortOimplOMP(tmp + mid, a + mid, length - mid);
#pragma omp taskwait
} else {
  mergesortOimpl(tmp, a, mid);
  mergesortOimpl(tmp + mid, a + mid, length - mid);
}
```

Po provedení modifikací výše uvedených dojde k 63.2-násobnému zrychlení oproti verzi s přímočarou paralelizací (uvažujeme samozřejmě pouze korektní verzi, tedy s operací `omp taskwait`). Tato varianta je již alespoň cca 4x rychlejší nežli sekvenční, ale stále zdaleka nedosahuje rychlosti PGNU verze.

3.2.3 Paralelizace slévání pole

Sekvenční metoda slévání se nazývá dvoucestnou. To znamená, že máme dvě seřazené pole, které slučujeme dohromady do jednoho seřazeného pole. Posledním krokem vedoucí k lepším výsledkům je efektivní paralelizování této metody. Tato vylepšená metoda se nazývá *vícecestné slévání* (angl. multiway merge nebo k-way merge). Přímočará implementace, která by z každé k-té posloupnosti vybrala nejmenší prvek, ten vložila do výsledného pole na konec a celý tento cyklus by pokračoval, dokud by zbývala data ke slití v některé z k-té posloupností by trval čas $\Theta(nk)$.

Její podstatou je slévání p posloupností s $p-1$ hraničními hodnotami mezi jednotlivými posloupnostmi do jedné posloupnosti, kde p (v našem případě $p = k$) je počet vláken, které zatížíme prací. Pokud zvolíme vhodně hraniční hodnoty, tak dojde k vyváženému rozdělení posloupností a následnému efektivnímu zatížení vláken při práci. Výběr hraničních hodnot lze provést např. metodou vzorkování nebo výběrem náhodného prvku. Algoritmus lze také zrychlit uložením ukazatelů na každý nejmenší prvek z k-té posloupnosti do tzv. stromu nebo minimální haldy. Pak lze celý tento problém vyřešit v čase $\mathcal{O}(n \log k)$, kde proměnná k značí o kolika cestné slévání se jedná a proměnná n značí velikost množiny dat, kterou bude tato metoda slévat dohromady. U varianty in-place nelze rozumně implementovat vícecestné slévání. Nyní ještě rozeberu možné alternativy zrychlení pomocí uložení všech nejmenších prvků [36]:

- metoda **haldy** – Tento algoritmus alokuje nejmenší možnou haldu, do které se uloží ukazatele na vstupní pole. Ze začátku tyto ukazatele ukazují na nejmenší prvky vstupních polí. Následně jsou tyto ukazatele seřazeny na základě hodnoty na kterou ukazují. V $\mathcal{O}(k)$ krocích je halda

vytvořena za použití klasické procedury pro vytvoření haldy. Poté algoritmus ve smyčce přenáší elementy, na které ukazují kořenové ukazatele. Následně tyto ukazatele inkrementuje, tak aby ukazovaly na následující prvky. Poté se dekrementuje ukazatel na kořenový prvek. Procedura pro inkrementaci ukazatelů je ohraničena časem $\mathcal{O}(\log k)$. Jelikož se pracuje nad n prvky, pak celkový čas běhu je ohraničen časem $\mathcal{O}(n \log k)$.

- metoda **stromu** – Tato metoda vytváří vyvážený binární strom s k listy. Každý list odkazuje na jeden ze vstupních polí. Každý uzel obsahuje hodnotu a index. Pro list na i-té pozici platí, že je jeho hodnota nastavena na hodnotu na jakou ukazuje ukazatel. Pro každý vnitřní list je nastavena jeho hodnota na nejmenší hodnotu z celého jeho podstromu. Index u takového listu nám ukazuje, z kterého uzlu byla tato nejmenší hodnota přebrána. Díky tomu se v kořenovém uzlu objeví nejmenší hodnota z celého stromu. Iterativně se přenáší kořenový uzel s hodnotou a ukazatelem (nejmenší prvek) a následně se nastaví kořen ve zbylém stromu na jeden ze dvou uzlů (na ten který obsahuje menší hodnotu). Jelikož je tento strom o k listech vyvážený a obsahuje n elementů, pak celkový čas je ohraničen $\Theta(n \log k)$.

Tato druhá metoda může v běhu vypadat např. následovně. Mějmě 4 seřazené pole $\{\{6, 11, 16, 21\}, \{11, 14, 17, 20\}, \{3, 20, 27, 41\}, \{19, 23, 24, 25\}\}$, které potřebujeme slít dohromady. Vytvoříme binární strom o čtyřech listech, které odpovídají prvním prvkům ze vstupních polí (tedy listy s ukazately na 6, 11, 3 a 19). Uzly z každého pole jsou mezi sebou porovnány, směrem nahoru vždy probublá ta vyšší hodnota, čímž dostaneme do kořenového uzlu hodnotu 3. Tato hodnota je nahrazena ze stromu (a přidána do výstupního pole, kam sléváme data) následujícím prvkem z daného vstupního pole - tedy hodnotou 20. Tento cyklus se pak následně iterativně opakuje do naplnění výstupního pole všemi prvky.

3.3 Radixsort

Paralelizace řadícího algoritmu radixsort se velmi podobá paralelizaci countsortu [24]. V prvním kroku se každému z p procesorů nezávisle přiřadí $\frac{n}{p}$ elementů z vstupní množiny dat n. Všechna vlákna vypočítají kolektivně prefixové součty svých částí a následně vypočítají celkový prefixový součet. Každé vlákno si následně vypočte offset (z celkového prefixového součtu), kam bude zapisovat do výstupního pole a uloží do něj na správné pozice prvky. Důležitým krokem je optimální implementace prefixového paralelního součtu, na základě kterého se vypočítává offset [13].

3.3.1 Paralelní prefixový součet

Sekvenční výpočet prefixového součtu lze zadefinovat následující definicí. Mějme n-prvkovou posloupnost čísel

$$x_0, x_1, ..., x_{n-1}, x_n.$$

Prefixový součet prochází tuto posloupnost od začátku tohoto pole a přepíše (jedná se tedy implicitně o in-place variantu, out-place varianta lze vytvořit triviálně) každý prvek součtem ze všech předcházejících a současného prvku. Tedy pomocí pseudokódu by výpočet prefixového součtu vypadal následovně:

```
int pole[]; // předpokládáme, že v poli jsou již data
int tmp = pole[0];
for (i = 1; i < pole.length; i++) {
  pole[i] += tmp;
  tmp += (pole[i] - tmp);
}
```

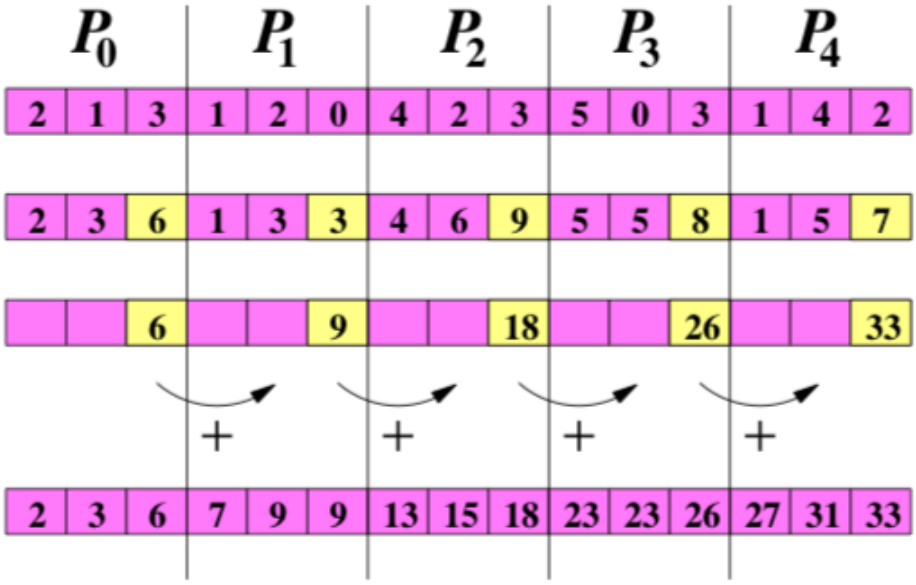
Paralelizace tohoto algoritmu se dá rozepsat do čtyř kroků [40].

1. Rozdělení posloupností na p částí, kde p je počet procesorů.
2. Každý procesor provede lokální výpočet nad svou částí.
3. Počínaje prvním procesorem až po předposlední procesor dojde k přeposlání vypočteného prefixového součtu následujícímu procesoru, který tuto hodnotu přičte ke svému vypočítanému prefixovému součtu, který se nachází v dané části jako poslední prvek.
4. Každý procesor si uchoval přijmutou hodnotu z minulého kroku a tu přičte ke všem prvkům mimo poslední prvek ve své části.

Celý tento proces přehledně demonstruje obrázek 3.3.

3.3.2 Paralelní redukce

V případě MSD i LSD algoritmu v sekvenční verzi lze urychlit celý algoritmus tím, že se vypočte u nejdelšího čísla pozice nejvyšší číslice. A to z toho důvodu, že stačí řadit číslice od této pozice, není zapotřebí řadit nulové hodnoty před ní. Výpočet tohoto indexu je použit i u mnou prezentované a implementované sekvenční verze. Tento proces lze zrychlit za použití paralelní redukce, která je velmi podobná paralelnímu prefixovému součtu s tím rozdílem, že výstup není zpět do pole (in-place) nýbrž do jedné celočíselné hodnoty. Důležité je zmínit, že použitá binární operace mezi jednotlivými prvky musí být asociativní. V případě binární operace sčítání se jedná o sumu nad prvky,



Obrázek 3.3: Postup při výpočtu paralelního prefixového součtu [10]

v případě binární operace násobení hovoříme o produktu. V sekvenční verzi by pseudokód vypadal následovně:

```
int pole[]; // předpokládáme, že v poli jsou již data
int suma = 0;
for (i = 0; i < pole.length; i++) {
  suma += pole[i];
}
```

Pro efektivní paralelizaci této procedury v prostředí OpenMP existuje direktiva:

```
#pragma omp parallel for reduction(+:suma)
```

Ta se vloží před `for` cyklus. Funkce `reduction` v ní volaná má dva parametry:

1. prvním parametrem je binární asociativní operace, která se má použít
2. druhým parametrem je jméno proměnné, do které se redukce provede (uloží)

Kapitola 4

Paralelizace řadících algoritmů pomocí CUDA

V případě implementace pomocí technologie CUDA se dá očekávat většího zrychlení nežli v případě OpenMP. Po nastudování materiálů, jak implementovat řadící algoritmy běžící na grafických kartách, jsem se rozhodl že tato práce nezůstane pouze u návrhu, jak upravit algoritmy výše popsané, ale implementuji je a výsledky porovnám se sekvenční i OpenMP verzí.

Do CUDA knihovny byla zahrnuta knihovna *thrust*, která obsahuje funkci *thrust::sort*. Tuto funkci také zahrnu do výsledného testování, abych mohl porovnat mnou implementované CUDA algoritmy. Tato funkce byla dříve implementována jako quicksort s výběrem pivota jako mediánu ze tří, poté byla ale upravena na introsort algoritmus [15]. Introsort algoritmus je velmi podobný quicksortu, ale má výhodu, že jeho časová složitost v nejhorším případě je rovna $\mathcal{O}(n \cdot \log n)$. Introsort je vlastně hybridní quicksort, který se přepne na heapsort ve chvíli, kdy se hloubka rekurze stane příliš velkou.

4.1 Quicksort

Identicky jako u sekvenčního algoritmu i zde dochází k rekurzivnímu rozdělení pole na dvě poloviny. V každé této iteraci je zvolen nový pivot, tím jsou vytvořeny dvě podposloupnosti, které se mohou seřadit nezávisle na sobě. Volba pivota může být determenistická nebo i náhodná. Po určité době běhu rekurze bude již vytvořeno dostatečné množsví podposloupností, tak aby každému vláknu mohla být jedna přiřazena. Ale předtím než se algoritmus dostane do tohoto stádia, tak musí vlákna pracovat na stejných posloupnostech. Celý proces se dá rozdělit do dvou fází [16].

V první fázi některá vlákna v bloku mohou pracovat na různých částech jedné sekvence elementů, které se mají seřadit. Tato procedura vyžaduje řádnou a pečlivou synchronizaci mezi jednotlivými vlákny uvnitř bloku. A to z důvodu, že výsledky z různých bloků musí být spojeny do dvou podposloupností (prvky které jsou menší nežli pivot a prvky které jsou větší nežli pivot). Důvodem je, že některé bloky mohly být spuštěny sekvenčně a není známo pořadí v kterém byly spuštěny. Korektní synchronizací v tomto bodě je tedy vyčkat až všechny bloky dokončí svou práci. V tomto případě se tedy nabízí použití jakési bariéry v kódu.

Po překročení bariéry dojde k přidělení nových sekvencí dat nad kterými budou data pracovat. Za tímto úkonem se skýtá operace překopírování dat z CPU do paměti GPU, která zabere relativně krátký čas, ale i přesto je cílem optimalizace tuto procedoru opakovat minimálně. Ve chvíli kdy existuje již dostatečné množství vytvořených podposloupností, tak že každému vláknu může být přidělena jedna tato podposloupnost, může započíst druhá fáze.

V druhé fázi má každé vlákno uvnitř bloku přiřazenou vlastní část dat nad kterou pracuje. Z tohoto důvodu není v této fázi zapotřebí provádět synchronizaci mezi vlákny. Tato část oproti 1. fázi běží pouze na grafickém procesoru. Menší náročnosti na sdílenou paměť lze dosáhnout použitím explicitního zásobníku a rekurzení se vždy na nejmenší podposloupnosti. Je efektivnější použít některý jiný řadící algoritmus ve chvílí, kdy podposloupnosti mají již relativně malou velikost [17]. A to z důvodu, že režie funkce `rozdělování` do polí je relativně značná, když se potýkáme s kratkými posloupnostmi.

4.1.1 Out-place a in-place varinta

V případě sekvenční verze tohoto algoritmu je paměťová náročnost redukována, jelikož se jedná o implementaci typu in-place. Ale v případě návrhu algoritmu na grafickém procesoru je zapotřebí myslet na omezenou velikost paměti cache a relativně velkou časovou náročnost synchronizace vláken. A jelikož je zapotřebí synchronizace mezi stovkami vláken, tak výhody in-place varianty se vytrácejí a nastává otázka, zdali out-place varianta by nebyla efektivnější. Samozřejmě nikoliv po stránce paměťové náročnosti, ale po stránce doby běhu algoritmu. Tato varianta se v tomto případě případě implementuje tak, že v každé iteraci funkce pro rozdělení pole se načtou data do hlavní vyrovnávací paměti a výsledek procesu se uloží do dočasného pole. Následně se ukazatele na tyto dvě pole prohazují, dokud zbývají nějaká data k zprocesování pomocí funkce `rozdělení`.

4.1.2 Funkce pro rozdělení pole

Nejdříve se vybere sekvence dat pro rozdělení. Tato část se následně rovnoměrně rozdělí na b částí, kde b je počet volných bloků, které mohou začít pracovat. Každému takovému bloku je následně přiřazena adekvátní část k rozdělení. Blok následně projde data, která jsou mu přiřazena, tak že všechna vlákna přístupují k souvislé části paměti. To má za následek plné využití potenciálu programování pod systémem CUDA, jelikož takto implementovaná operace čtení má značně nižší paměťové nároky na čtení než kdyby se přistupovalo k souvislé paměti jakýmkoliv jiným způsobem.

4.1.3 Paralelní prefixový součet

Hlavním úkolem funkce `rozdělení` zůstává stejný úkol jako tomu je u sekvenční nebo OpenMP verze. Jediným rozdílem je, že jelikož v tomto případě se nejedná o in-place variantu, tak prvky menší resp. větší než pivot se přesouvají do dočasného pole. Otázka je, jak každé vlákno bude vědět do které části dočasného pole má zapsat svůj výsledek, tak aby zůstala efektivita CUDA a nebyla zbytečně často prováděna komunikační synchronizace mezi jednotlivými vlákny?

Jedním z možných přístupů je výpočet paralelního prefixového součtu [16]. Tedy každé vlákno si načte prvek, poté se provede synchronizace pomocí bariéry - tedy počkáme, dokud si všechna vlákna nenačtou jeden prvek. Poté se vypočte prefixový součet nad čísly vláken, které chtějí zapisovat do levé části výstupního pole (hodnoty menší nežli pivot) a také druhý prefixový součet nad čísly vláken, které chtějí zapisovat do pravé části výstupního pole (hodnoty větší nežli pivot).

Od této chvíle každé vlákno ví na jakou konkrétní pozici má prvek zapsat. A to díky tomu, že ví, že x vláken s menší hodnotou *id* vlákna než je jeho vlastní bude zapisovat do levé části a y vláken s větší hodnotou *id* vlákna budou zapisovat do pravé části. Pokud bude na základě porovnání zapisovat do levé části pole, tak bude zapisovat do dočasného pole na pozici $x + 1$ a v případě, že bude vlákno zapisovat do pravé části pole, tak správná pozice k provedení zápisu je $n - (y + 1)$, kde n je velikost celého dočasného pole.

4.1.4 Metoda dvou průchodů

Výpočet prefixového součtu trvá relativně nezanedbatelnou dobu. Tomu se lze vyhnout pomocí použití tzv. metody dvou průchodů přes všechna vstupní data [16].

V prvním průchodu si každé vlákno napočítá přes všechny elementy, které jsou mu přiděleny, dvě sumy. První sumou je počet prvků menších nežli pivot a druhou sumou je počet prvků větších nežli pivot. Jakmile všechna vlákna v rámci bloku dodělají tuto práci, tak mohou vypočítané sumy využít místo výsledku z prefixového součtu. V tuto chvíli každé vlákno ví, kolik paměti zabírají vlákna s nižším *id* než je jeho vlastní. Tím dostaneme velikost pole, které je potřeba alokovat v rámci bloku a to pouze jednou při každé iteraci. V této první fázi je ale i přesto zapotřebí jednou vypočítat prefixový součet pro výpočet celkové velikosti potřebné pro každý jednotlivý blok.

Nyní každé vlákno ví, kam má data přesně zapisovat. V tuto chvíli začnou vlákna opět procházet všechna jim přiřazená data a začnou je ukládat do dočasného pole na správné místo (to znají z prvního průchodu). Posledním krokem je uložení pivota do paměti mezi levou a pravou část.

4.2 Mergesort

V této sekci popíšu možné úpravy nad sekvenční verzí mergesortu, tak aby byla efektivně zparalelizována nad GPU. Jednou z možností je využít algoritmus *Merge Path* popsaný zaměstnanci z *Israel Institute of Technology* [18]. Jimi popsaná metoda rozděluje dvě seřazené pole do párů sekvencí prvků po sobě jdoucích v paměti. Z tohoto páru je vždy sekvence prvků z jednoho pole a druhá sekvence prvků z druhého pole. Každý pár může obsahovat libovolný počet prvků a prvky z každého páru tvoří po sobě jdoucí sekvenci prvků umístěnou ve výstupním (seřazeném) poli. Tento algoritmus nevyžaduje synchronizaci na úrovni grafické karty a jedná se o velmi efektivní řadící (slévací) algoritmus, co se týče čtení z a zápisu do cache paměti. Druhou variantou je paralelizace slévání, tak jak je popsána v OpenMP sekci, tedy pomocí metody multi-way merge.

4.2.1 Paralelní slévání

Za předpokladu, že máme k dispozici p jader, pak můžeme říct, že každé vlákno je zodpovědné za slévání své části do výsledného pole. Víme, že tato část je veliká $\frac{n}{p}$. Jelikož každé vlákno obdrží stejně velkou podposloupnost k zpracování, tak víme, že všechna vlákna skončí v přibližně stejnou chvíli. Metoda *Merge Path* se skládá ze dvou částí:

1. Část rozdělení: Každé vlákno rozdělí vstupní pole na části. Každé vlákno si najde nepřekrývající se část z každého vstupního pole. Tyto dvě části sice nemusí mít stejnou velikost mezi sebou, ale součet těchto dvou částí v rámci rozdělení bude vždy stejný napříč celým během algoritmu.
2. Část slévání: Každému vláknu jsou přiděleny dvě podposloupnosti, které má za úkol slít. Tato procedura je stejná jako u sekvenčního algoritmu.

Vlákna musí výsledky zpracovávat do disjunktních částí výstupního pole, díky čemuž tento proces může běžet paralelně napříč jádry a bez synchronizačních bariér typu zámek, které by celkový čas doby běhu algoritmu prodlužovaly.

Obě tyto části běží paralelně na systému, nejprve první a po jejím skončení pokračuje druhá část.

4.2.2 Merge Path

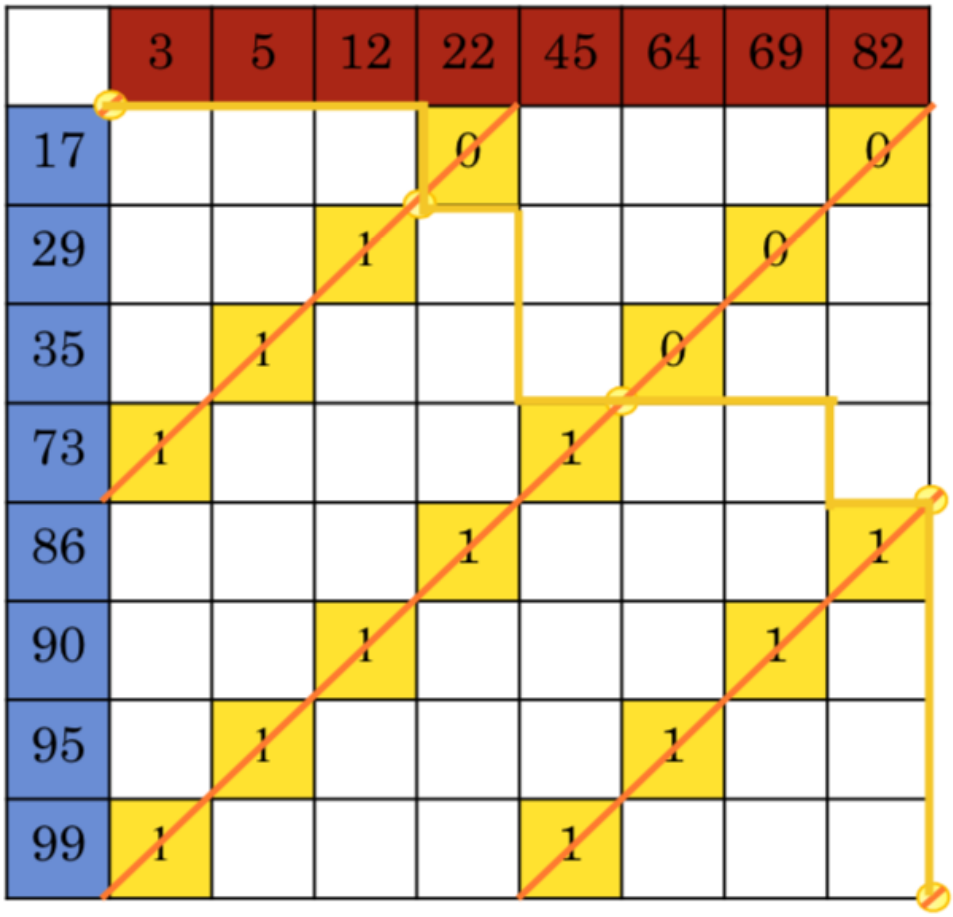
Tento paralelní algoritmus [27] slévá dvě pole do jednoho výsledného, které je seřazené. Níže popsaný algoritmus je definován pro CREW PRAM, ale dá se adaptovat pomocí změn na jiné varianty PRAM modelu. Algoritmus je dobře vybalancovaný, co se týče rovnoměrně rozložené zátěže mezi jednotlivá vlákna. Zároveň u něj nedochází k zbytečným a časově náročným zámkům, kvůli čekání na přístup k paměťovým sekcím více než jedním vláknem. Tento algoritmus nepotřebuje ke svému korektnímu fungování zámky.

Mějme dvě seřazená pole A a B. Bez újmy na obecnosti předpokládejme, že prvky jsou v polích A a B seřazeny vzestupně. Vytvoříme si matici M o $|A|+1$ řádcích a $|B|+1$ sloupcích. Do prvního sloupce vložíme popořadě všechny prvky z pole A, do prvního řádku vložíme popořadě všechny prvky z pole B. Následně vytvoříme pomocí níže popsaného algoritmu cestu, ta je v obrázku 4.1 značena pomocí žluté křivky s puntíky. Do matice zapíšeme na pozici $M[i,j]$ hodnotu 1, pokud platí $A[i] > A[j]$, jinak zapíšeme hodnotu 0. Jak znázorňuje obrázek 4.2, tak křivku (Merge Path) vytvoříme na hranici hodnot 0 a 1. Na obrázku lze vidět, že konstrukci Merge Path cesty lze také pojmout tak, že začneme v horním levém rohu ($i = 0, j = 0$) matice a postupujeme následovně po matici:

- Pokud $A[i] \geq B[j]$, pak se posuneme směrem dolů o jednu pozici.
- Pokud $A[i] < B[j]$, pak se posuneme směrem doprava o jednu pozici.

Tento algoritmus pokračuje do doby, dokud se křivka nedotvoří až do pravého dolního rohu matice.

Ve chvíli kdy je vytvořena a definována konkrétní a jednoznačná cesta Merge Path, pak je i známo přesné pořadí slévání jednotlivých částí. A díky znalosti cesty lze slévání rozdělit na jednotlivé části a ty slévat souběžně. Nevýhodou této varianty je, že spočítání celé cesty pomoci diagonálního binárního hledání pro nalezení všech změn (dolů, doprava) trasy zabere $\mathcal{O}(n \log n)$ času. Řešením tohoto nedostatku je spočítáním pouze p bodů na této cestě, kde p je počet procesorů. Pro nalezení těchto p bodů se využije p úseček, které jsou

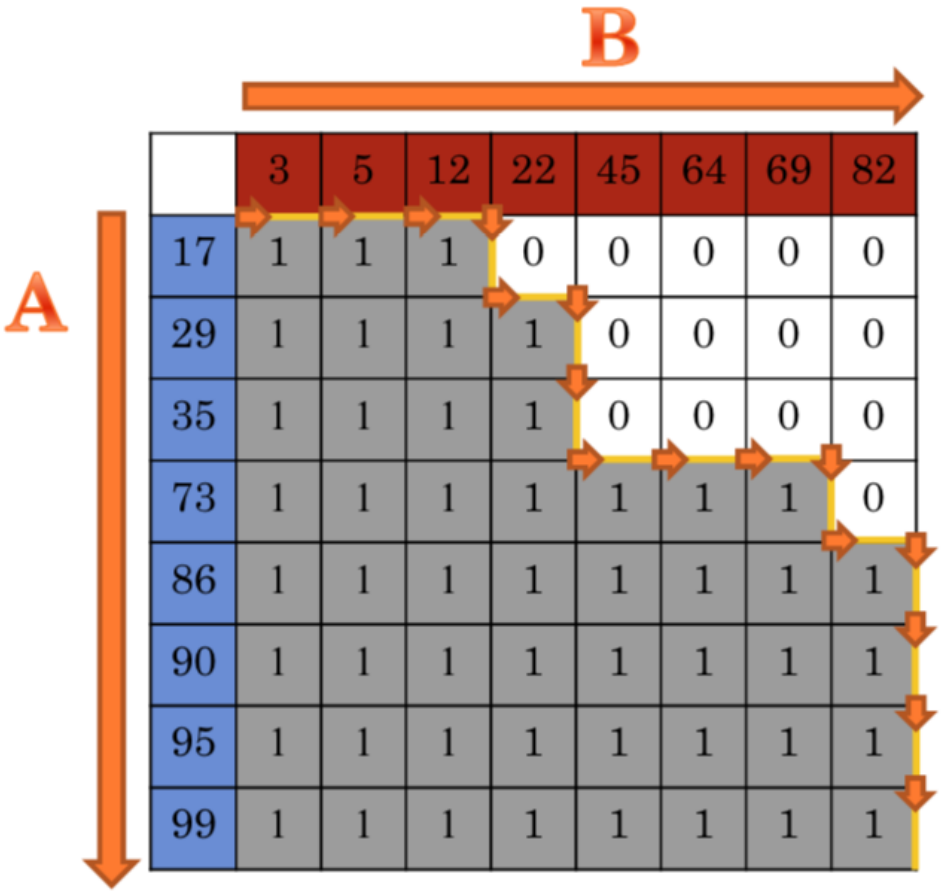


Obrázek 4.1: Konstrukce cesty Merge Path [18]

rovnoběžné s vedlejší diagonálou v matici a jsou rovnoměrně rozloženy (mají rovnou vzdálenost mezi sebou) v matici, což nám zajistí rovnoměrnou distribuci práce mezi jednotlivá vlákna. Křížení těchto úseček s námi vytvořenou cestou Merge Path vytvoří p hledaných bodů.

4.2.3 Rozdělení pole na GPU

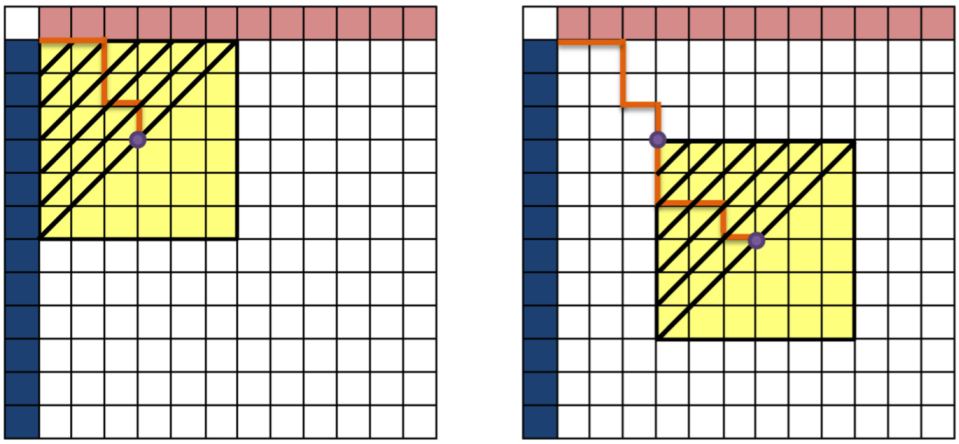
Výše popsaný algoritmus pro rozdělení pole pomocí vedlejších diagonál není složité implementovat pro běh pod grafickou kartou. Na obrázku 4.3 je znázorněno rozdělení hledání cesty Merge Path mezi jednotlivá vlákna GPU - každé vlákno prohledává určitý blok a v něm hledá křížové body mezi diagonálou a cestou Merge Path. Veškeré nalezené křížové body po spojení vytváří cestu Merge Path.



Obrázek 4.2: Zapisování hodnot 0 a 1 do matice použité u algoritmu Merge Path [18]

V rámci jednoho vlákna na GPU lze provést hledání křížových bodů několika způsoby, mezi ně patří níže 3 popsané metody pomocí binárního půlení [27], ale zajisté jich existuje více. V těchto metodách w značí velikost warpu na GPU.

1. Metoda půlení do w-šířky: Tato metoda načte w po sobě jdoucích elememtů z pole A a B. Uvnitř bloku o velikosti w si každé vlákno uloží prvek z polí A a B z globální paměti do své lokální paměti. Tento přístup efektivně využívá principu prostorové lokality. Jelikož křížový bod je unikátní, tak pouze jedno vlákno nalezne tento bod a jeho adresa se uloží do globální paměti, čímž se odstraní potřeba pro použití synchronizačních metod. Tímto procesem dochází k paralelnímu vyhledávání w bodů. Celková časová složitost nalezení cesty Merge Path v tomto



Obrázek 4.3: Rozdělení hledání cesty Merge Path mezi vlákna GPU [18]

případě je rovna $\mathcal{O}(w \cdot \log n - \log w)$.

2. Klasická metoda půlení: Tento jednodušší přístup se implementuje tak, že každé vlákno provádí binární půlení na jemu přidělené diagonále. Celková časová náročnost v tomto případě je $\mathcal{O}(\log n)$.

3. Hledání rozdělením na w částí: Vedlejší diagonála se rozdělí na 32 disjunktních a stejně velkých částí. Každé vlákno ve warpu má na starost právě jednu operaci porovnání. Každé vlákno si zkontroluje zdali se hledaný bod nachází v jeho části dat. Stejně jako v první variantě, tak i v této pouze jedno vlákno ve warpu nalezne hledaný bod a z toho důvodu není zapotřebí synchronizačních metod. Výhodou této metody je, že každým průchodem se hledaný prostor zmenší w-krát. Celková časová náročnost v tomto případě je $\mathcal{O}(w \log_w n - \log n)$.

První dvě výše popsané varianty jsou časově rychlejší a to z důvodu, že poslední varianta je neefektivní, co se týče přístupu do globální paměti. V každé iteraci dochází k celkem w přístupům do globální paměti, čímž dochází ke zpomalení z důvodu čekání na načtení dat.

4.2.4 Slévání na GPU

Fáze slévání z Merge Path algoritmu, který byl popsán zaměstnanci z *Israel Institute of Technology*, není přizpůsobena pro běh nad GPU paralelně, jelikož tato fáze je čistě sekvenční pro každé jádro. Je tedy zapotřebí upravit tuto fázi tak, aby běžela paralelně na všech SPs na každém SM [27].

Pro zjednodušení řekněme, že části vytvořené z předešlého algoritmu jsou o hodně větší nežli velikost warpu w a že pro velikost bloku Z platí $Z \geq w$. Překopírujeme blok po sobě jdoucích prvků o velikosti Z z každého rozděleného pole a uložíme jej do sdílené lokální paměti. Jelikož máme blok o velikosti Z, tak je možné nalézt cestu Merge Path nad Z elementy pomocí křížového diagonálního binárního vyhledávání (algoritmus popsaný výše, kdy hledáme křížové body). Pro nalezení této cesty se spotřebuje $\mathcal{O}(\log Z)$ iterací algoritmu.

Ve chvíli, kdy známe celou cestu Merge Path, pak je možné všech Z prvků slít souběžně, jelikož každý prvek bude zapsán na pozici konkrétního indexu. Jakmile dojde ke slití, tak je možné celý proces opakovat pro další blok Z elementů. Jak ukazuje obrázek 4.3, tak po skončení prvého hledání cesty Merge Path v levém horním rohu se přejde v každém dalším kroku na blok prvků, tak aby začínal tam, kde skončil předchozí v hledání cesty.

4.3 Radixsort

Varianta LSD řadícího algoritmu radixsort pracuje na principu, že se řadí po jednotlivých číslicích. V `for` cyklu se začne od nejméně důležité číslice až po nejvíce důležitou číslici, řekněme že těchto průchodů je k. I v CUDA variantě lze implementovat výpočet maximální číslice u největšího čísla, tak jak je tato metoda popsána v OpenMP verzi této práce. Kopírování dat do paměti karty trvá nezanedbatelnou dobu. A jelikož pro např. 32-bitové klíče se jedná o $k = 32$ operací rozházení do přihrádek [25] a při každé této operaci se v paměti GPU načítá znovu seřazené pole, tak nastává otázka, jak toto optimalizovat. Jednoduchá odpověď zní – zvětšit základ z, čímž se sníží počet přihrádek a tím pádem i počet kopírování dat v rámci grafické karty.

Správným krokem úspěšné a efektivní paralelizace je rozdělení vstupního pole na bloky [26]. Každý blok dat bude přiřazen jinému vláknu a bude mít vlastní přihrádky. Za pomoci paralelního prefixového součtu se následně vypočítají offsety, kam jednotlivé procesory budou mět zapsat do globální proměnné obsahující přihrádky pro všechny bloky. Tento model je ale stále relativně neefektivním, co se týče přístupu do paměti GPU. Přístupy do paměti grafické karty jsou neuspořádané a nenachází se *po sobě*, což by určitě vedlo k lepším výsledků, co se týče doby běhu algoritmu.

4.3.1 Optimalizace

První optimalizací pro optimalizaci práce s pamětí je použití $z > 2$ [25], kde z značí číselný základ, podle kterého řadíme data. Pro využití maximální koherence při rozhazování dat do přihrádek se jeví jako nejlepší volba použití

sdílené paměti na GPU na které se pak lokálně řadí bloky prvků. Toto řazení probíhá klasicky v `for` cyklu po jednotlivých číslících dané soustavy z.

Díky řazení prvků lokálně předtím nežli se vypočítá pozice, kam se mají prvky uložit, a rozházení těchto prvků do přihrádek zaručíme, že prvky se stejným klíčem v rámci bloku jsou umístěny za sebou v paměti na GPU. Tento algoritmus je stabilní a implementuje se ve variantě LSD. Jeho implementace je rozdělena do čtyř různých kernelů [25], které jsou postupně volány pro každou číslici ze základu z.

1. Seřazení každého bloku dle i-té číslice ve sdílené paměti bloku na GPU.
2. Spočítání offsetů pro všechny přihrádky a uložení těchto informací do globální proměnné.
3. Spočítání paralelního prefixového součtu nad globální proměnnou z předešlého bodu.
4. Vypočtení přesné pozice pro každý prvek díky vypočtenému paralelnímu prefixovému součtu a jeho uložení do paměti na správné místo.

V následujících odstavcích se rozepíši o jednotlivých kernelech.

Při řazení bloků v lokální paměti GPU za předpokladu, že číslo $z = 2^b$ – tedy že základ je mocninou dvojky, může proběhnout v s/b průchodech, kde s značí počet bitů čísla, které řadíme. Výběr hodnoty b je klíčový pro dobu běhu algoritmu. Pokud je vybrána hodnota moc velká, pak se tvoří mnoho přihrádek v každém bloku a díky tomu se následně vypočte mnoho offsetů, čímž se snižuje koherence rozdělení do přihrádek. Pokud ale zvolíme hodnotu b moc malou na druhou stranu, tak to pak vede k zvýšení počtu průchodů pro každou číslici. Hodnota $b = 4$ se na základě měření jeví jako optimální [25].

Druhý kernel, který spočítá sumu prvků pro každou přihrádku v rámci bloku, tak zároveň vypočítá i lokální offsety – tyto offsety jsou tedy vypočteny pouze v rámci současného bloku. Tyto hodnoty se následně zapíší do jedné globální proměnné a to z důvodu, aby nad ní mohl být spočítán prefixový součet.

Ve třetím kernelu se provede prefixový součet, nejlépe ve verzi paralelního zpracování. Toho lze docílit několika způsoby. První možností je implementace vlastního výpočtu paralelního prefixového součtu pod CUDA, což není úplně triviální záležitost. Druhou možností je využití knihovní verze z externí CUDPP. Třetí možností je použití knihovny *thrust*, která byla integrována do *CUDA SDK* a obsahuje varianty exkluzivního i inkluzivního paralelního prefixového součtu.

Poslední kernel má již na starost pouze uložení seřazeného prvku na správnou pozici do výstupního pole v GPU paměti. Tím algoritmus končí, v paměti grafické karty se nachází již seřazená data.

Kapitola 5

Implementace

V této kapitole stručně popíši jednotlivé implementace řadících algoritmů, tak jak byly mnou naprogramovány. Na kompaktním disku, které je přílohou této práce, se nachází veškeré zdrojové kódy těchto implementací.

5.1 Sekvenční verze algoritmů

5.1.1 Quicksort

Při implementaci tohoto řadícího algoritmu jsem zvolil výběr mediánu jako prostředního prvku ze vstupních dat. Důvodem je, že vstupní data budou generována náhodně mnou implementovaným generátorem vstupních dat a rozdělení hodnot bude téměř uniformě náhodné. Výběr mediánu ze tří náhodných prvků jsem nezvolil z důvodu, aby se do měření a porovnávání s jinými již existujícími implementacemi nevnášel prvek náhody. Ten by totiž ztěžoval testování, protože výsledné časy pro dva různé běhy by byly různé. To by šlo vyřešit velký počtem běhů algoritmu nad stejnými daty a následným průměrováním času, přesto jsem ale zvolil deterministický způsob výběru pivota.

V prvních implementacích jsem quicksort verzi implementoval s výběrem pivota jako prvního prvku v neseřazeném vstupním souboru. Tato varianta byla relativně rychlá se vstupním souborem obsahující náhodná data. Bohužel u dat se sestupně seřazenými daty nebo vzestupně seřazenými daty se algoritmus velmi zpomalil. Po změně pivota na prostřední prvek byl tento problém eliminován a výsledné časy byly již relativně konzistentní.

Tato implementace sekvenční verze se od pseudokódu popsaného v první kapitole liší pouze minimálně. Pivot je udržován na začátku pole. Následně jsou v cyklu prohozeny prvky pomocí pomocných indexů *i, j*. Na konci je vrácen pivot na své správné místo. A na takto upraveném poli je rekurzivně zavolána funkce `quicksort()` na levou a pravou část.

5.1.2 Mergesort out-place

Na začátku zavolání funkce `mergesortO()` si vytvoříme pomocné pole `tmp` o velikosti n. Díky této proměnné `tmp` se jedná o algoritmus typu out-place. Ve funkci `mergesortOimpl()`, která je již volána rekurzivně, se v každé úrovni rekurze střídají parametry pomocného a vstupního pole a dochází k postupnému půlení jejich velikostí. V každé úrovni je pak provedeno slití obou polí zpět dohromady.

5.1.3 Mergesort in-place

Konkrétně v mé implementaci, v případě kdy nám v rekurzi zbývá seřadit méně než 16 prvků ($\sqrt{16} = 4$, což tedy odpovídá 4 blokům), se algoritmus přepne na selectsort a pole se dořadí dle tohoto algoritmu, jedná se tedy o hybridní verzi řadícího algoritmu. Selectsort jsem implementoval tak, že umí řadit po jednotlivých prvcích a i po blocích. Obdobně je využit selectsort i na konci funkce k seřazení jednotlivých již seřazených bloků do jednoho celku (bloky jsou uvnitř seřazené, ale je zapotřebí je přeházet mezi sebou tak, aby tvořily seřazenou posloupnost jako celek). Zároveň je použit i k doseřazení pracovního prostoru včetně posloupnosti prvků, které se nachází za pracovním prostorem (vstup n byl rozdělen na $\sqrt{n}$ bloků, ale jelikož n nemusí být mocninou čísla 2, pak za posledním blokem pracovního prostoru mohou zbývat prvky k seřazení, které byly doteď opomíjeny). A v posledním kroku je opět selectsort zavolán nad již seřazenou částí bloků a seřazenou částí pracovního prostoru včetně prvků za ním, kde je využit cyklický buffer [41]. Po tomto posledním kroku máme data seřazená, ale už z popisu algoritmu tušíme, že jeho časová náročnost bude vyšší nežli u out-place varianty.

5.1.4 Radixsort MSD

Pro implementaci čistě sekvenční verze jsem použil MSD typ algoritmu. V prvním kroku jsem implementoval verzi, ve které se data uchovávala pomocí spojového seznamu. Po otestování doby trvání běhu algoritmu vůči ostatním již naimplementovaným jsem se rozhodl od této verze upustit z důvodu časové neoptimality a implementoval jsem verzi, která používá pole místo spojového seznamu. Proměnná `w` definuje počet bitů jedné číslice, podle které řadíme.

Nejdříve se projde celé pole a uloží se pořadí nejvyšší číslice v číselné soustavě, v které řadíme, u nejvyššího čísla do proměnné `max`, jelikož není třeba řadit čísla obsahující na pozicích před touto maximální pozicí `max` samé nuly. Tato heuristika mírně zlepšila výkon celého algoritmu. Následně se provádí radixsort klasickým způsobem, tak jak ho popisuji v teoretické části této práce.

V závěrečné kapitole s naměřenými výsledky doby běhů jednotlivých algoritmů jsem do sekvenční části zařadil i algoritmus typu LSD. Jedná se o

OpenMP verzi LSD algoritmu při běhu nad jedním výpočetním jádrem o jednom vláknu, tedy defakto jsem při měření zeserializoval paralelní algoritmus. To jsem učinil pro lepší porovnání LSD a MSD časové náročnosti v sekvenční variantě.

5.2 OpenMP verze algoritmů

5.2.1 Quicksort

Změny v mé implementaci paralelní verze tohoto algoritmu jsem provedl téměř dle popsaných úpráv výše uvedených. Zavedení prahu, kdy se algoritmus přepne na sekvenční verzi, bylo jednoduchou změnou. Hodnotu, kdy se algoritmus přepne na sekvenční verzi jsem nastavil na hodnotu

```
seq_thr = (b - a + 1) / omp_get_num_threads() / 1
```

Tedy parametr k jsem empirickým měřením nastavil na hodnotu 1.

Přebytečnou režii způsobenou zbytečně nadbytečným voláním rekurzivních úloh jsem řešil odstraněním `omp task` nad voláním funkce pro pravou polovinu pole. Algoritmus jsem ale již neupravoval pro běh ve `while` cyklu, jak je popsáno výše.

Největší a nejdůležitější úpravou byla paralelizace rozdělování pole. Tu jsem implementovat tak, že vstupní množina dat se rozdělí na p částí, kde p je počet vláken, které mohou vykonávat práci algoritmu. Každé vlákno bude pracovat pouze nad svou p-tou částí a to tak, že si do svých privátních čítačů uloží, kolik prvků je menších a kolik větších nežli pivot. Následně si všechna vlákna tyto informace nasdílí pomocí globální proměnné pole čítačů. Poté nad touto globální proměnnou se spočítá prefixový součet. Následně může každé vlákno uložit do pomocného pole do správné části svou levou a svou pravou část svých dat, jelikož má nasdílené čítače od ostatních vláken a ví přesně, kam tyto data může uložit, tak aby se nepřepisovala.

5.2.2 Mergesort out-place

Obalovací funkci `mergesortOOMP()` jsem implementoval téměř stejně jako je popsána na začátku kapitoly 3.2. U rekurzivní funkce `mergesortOimplOMP()` jsem empirickým měřením nastavil hodnotu prahu, kdy se paralelní verze přepne na sekvenční, na hodnotu 10^6. Tato část kódu, kde se buď volá rekurzivní parelelní nebo rekurzivní sekvenční funkce, se také neliší od té referenční popsané výše. Dále následuje kontrola nad v jakém počtu vláken kód běží. Pokud se jedná o jedno vlákno, tak se přepneme do sekvenční verze slévání, tak jak je popsána v kapitole 1.6.1.

V případě, že paralelní blok běží nad více než jedním vláknem, tak se data slijí pomocí paralelního 2-cestného slévání (tzn. slévají se dohromady *pouze* dvě pole A a B do C, ale proces slévání dělá p vláken – v tom se proces liší od sekvenční verze). Algoritmus najde pro každé vlákno začátek a konec v posloupnosti $(A_1 \dots A_k)$ nad kterou bude pracovat. Začátek a konec v A_i posloupnosti se najde lehce, tak že celou posloupnost rozdělím podle počtu vláken a každé vlákno pracuje nad svým blokem. Začátek resp. konec v konkrétním bloku B_i posloupnosti hledám pomocí binárního vyhledávání, tak aby měl nejbližší nižší hodnotu počátenímu resp. konečnému prvku daného bloku v A_i posloupnosti. Následně si vypočítám v paralelním bloku pro každé vlákno offset, kam přesně má zapisovat do výstupního pole C a tam pak již slévá.

5.2.3 Mergesort in-place

Tato varianta algoritmu se ukázala jako nejpomalejší ze všech pěti sekvenčních algoritmů (quicksort, mergesort out-place, mergesort in-place, radixsort MSD, radixsort LSD), které byly v rámci této práce implementovány. Dosahovala nejhorších výsledků a byla řádové pomalejší než ostatní algoritmy, které měly relativně podobné výsledky, co se týče doby běhu programu. Z tohoto důvodu jsem tento algoritmus paralelizoval pouze triviálně a nikoliv důkladněji. Tedy neparalelizoval jsem funkci slévání, která sama o sobě v sekvenční verzi byla již obtížnější k implementaci a ponechal jsem v tomto případě tedy verzi se sekvenčním dvoucestným slévaním. V následujících bodech popíši, jak jsem algoritmus paralelizoval.

1. Obalovací funkci jsem upravil tak, aby spouštení celého kódu probíhalo v paralelní sekci, s tím, že první úroveň rekurze proběhne v jednom vlákně.

```
template <typename T>
void mergesortIOMP(T * a, T * b, int pocet_vlaken) {
#pragma omp parallel num_threads(pocet_vlaken)
  {
#pragma omp single
    mergesortIimplOMP(a, b, 2*pocet_vlaken);
  }
}
```

2. Obdobně jako u sekvenční verze je i u této zaveden práh, který sepne, když již zbývá málo prvků k seřazení. Tento práh jsem empirickým měřením nastavil na hodnotu 16, což odpovídá 4 blokům k seřazení (z toho poslední dva jsou pracovním prostorem), jelikož $\sqrt{16} = 4$. Při překročení prahu se algoritmus přepne na selectsort. Jedná se tedy o hybridní algoritmus.

```
if (length < 16) {
  selectSort(a, b, 1);
  return;
}
```

3. Obdobně jako u OpenMP verze out-place varianty i zde probíhá první úroveň rekurzivního stromu volání pro určitý počet vláken a každá další úroveň rekurze je volána na přesně polovičním počtu vláken. Pokud nejsme v rekurzi již moc hluboko (`pocet_vlaken <= 1`), tak zavoláme paralelní verzi. V opačném případě se zavolá sekvenční verze algoritmu. V případě paralelní verze jsem provedl jistou optimalizaci tím, že se zbytečně nevolá OpenMP úloha pro druhou polovinu posloupnosti. Tuto práci může udělat současné vlákno, jedná se vlastně o stejnou optimalizaci jako u out-place verze popsané v kapitole 3.2.2.

```
if (pocet_vlaken <= 1) {
  mergesortI(a, a + mid);
  mergesortI(a + mid, a + new_n);
} else {
#pragma omp task
  mergesortIimplOMP(a, a + mid, pocet_vlaken);
  mergesortIimplOMP(a + mid, a + new_n, pocet_vlaken);
#pragma omp taskwait
}
```

5.2.4 Radixsort MSD

Mnou implementovaná verze je in-place variantou. Tento fakt zmiňuji proto, protože v odborné literatuře existují i popsané implementace out-place varianty nad technologií OpenMP.

Nad sekvenční verzí algoritmu jsem v prvém kroku provedl tu změnu, že se pomocí paralelní redukce za použití `omp parallel for reduction()` vypočítá na začátku algoritmu maximální pozice číslice největšího čísla. Tento krok se prováděl i v sekvenční verzi, ale pouze sekvenčně. Jelikož se ve verzi MSD volá implementační funkce rekurzivně, tak jsem zavedl práh. Prahovou hodnotu jsem zvolil na základě empirického měření, dochází tedy k přepnutí z paralelní verze na sekvenční. V případě nehybridní verze docházelo ke zbytečnému řazení malých posloupností pomocí OpenMP technologie, což kvůli její režii celkově čas běhu algoritmu zvyšovalo. Prahovou hodnotu jsem nastavil pomocí následujícího vzorce:

```
SWITCH_TO_RADIX_SERIAL = (b-a) / ((1 << w)*omp_get_num_procs());
```

Hodnota $(b-a)$ odpovídá počtu prvků, $(1 << w) = 2^w$ je maximální počet číslic, podle kterých řadíme a funkce `omp_get_num_procs()` nám vrátí počet procesorů, které lze využít k paralelní práci.

V závěrečné fázi implementační funkce lze provést paralelizaci nad `for` cyklem mající na starost rekurzivní seřazení obsahu jednotlivých přihrádek.

```
#pragma omp parallel for
  for (int j = 0; j < (1 << w); j++) {
    radixsortImplOMP<T, w>(prihradky[j], prihradky_pointery[j],
     i-1);
  }
```

5.2.5 Radixsort LSD

Verze LSD se ve své podstatě liší od MSD verze hlavně tím, že sama sebe nevolá pomocí rekurzivní funkce. Je tedy pouze iterativní. Sekvenční verze obsahuje pouze čtyři po sobě jdoucí (nikoliv vnořené) `for` cykly. V případě paralelizace této verze jsem se rozhodl i k paralelizaci procesu rozřazování do přihrádek. Mimo počáteční definice, deklarace a uvolnění paměti na konci algoritmu je celý kód funkce obalen pomocí direktivy:

```
#pragma omp parallel num_threads(pocet_vlaken)
```

V rámci tohoto bloku si každé vlákno lokálně vypočítá nad svou částí posloupnosti pozici nejvyšší číslice u největšího čísla. Následně se v kritické sekci uloží do globální proměnné nejvyšší napočítaná pozice. Poté se pro každou číslici provedou následující čtyři (resp. pět) `for` cykly [1]:

1. spočte se, kolik do každé přihrádky patří prvků
2. získané informace z předešlého `for` cyklu se uloží do globální proměnné
3. pouze hlavní vlákno provede prefixový součet nad globální proměnnou (PPS jsem neprováděl, jelikož pro $w = 4$ se vytvoří 16 přihrádek a na testovaném počítači s čtyřjádrovým procesorem s HT technologií by se jednalo o pole o velikosti nanejvýš 128 prvků a to je přílíš malé pro paralelizaci)
4. nastavíme si ukazatele, kam bude každé vlákno zapisovat do výstupního pole
5. samotné zapsání dat do výstupního pole na správné pozice

Tato implementace je in-place.

5.3 CUDA verze algoritmů

5.3.1 Quicksort

Nejdříve dojde k alokaci paměti a následnému překopírování veškerých dat k seřazení do paměti grafické karty. Práh pro přepnutí z quicksortu na jiný algoritmus, kdy zbývá již relativně malá sekvence k seřazení, jsem nastavil empirickým měřením na hodnotu 1024^2. Algoritmus na který se přepne je bitonicsort - na GPU se implementuje relativně jednoduše a je velmi rychlý pro krátké posloupnosti. Na funkci `rozdělení` (partitioning) jsem využil funkci z knihovny thrust, která se jmenuje *thrust::partition()* - je rychlá a nepřišlo mi užitečné implementovat svou vlastní funkci na tuto operaci.

5.3.2 Mergesort out-place

Jelikož implementace algoritmů pod GPU nemá být náplní této práce, tak jsem se rozhodl neimplementovat variantu Merge Path. Místo toho jsem usoudil, že přeměna OpenMP verze s paralelním 2-cestným sléváním bude dostatečná pro demonstraci.

Mimo nakopírování dat na GPU na začátku a uvolnění paměti na GPU na konci algoritmu je volán cyklicky kernel. Ten na základě rozhodnutí, zdali zbývá ke slití více než jeden pár prvků na vlákno, přepne na sekvenční 2-cestné slévání. V opačném případě provádí paralelní 2-cestné slévání. Obě tyto slévácí funkce nejsou optimalizované tak, aby měly přístupy do paměti blízko sebe.

Identicky jako u OpenMP verze se i zde používá binární půlení na vyhledávání vhodného začátku a konce v druhém poli.

5.3.3 Radixsort LSD

Mnou implementovaná verze prvně provede alokaci a následné překopírovaní celé neseřazené posloupnosti dat do paměti GPU. Následně se zavolá kernel, který vypočte pozici maximální číslice u největšího čísla z dat v GPU paměti. Napočtené hodnoty z bloků se vrátí zpět do paměti CPU, kde se vypočte globální hodnota maximální číslice. Následně se volají v `for` cyklu pro každou číslici (LSD varianta) tyto tři funkce:

1. Kernel `radix1()` má na starost napočítání lokálních offsetů v rámci bloku a zároveň uložení těchto hodnot do globální proměnné.

2. Knihovní funkce `thrust::inclusive_scan()` vypočte prefixový součet, který nám poslouží k tomu, abychom věděli, kam přesně uložit seřazené prvky na správná místa.

3. Kernel `radix2()` provede zapsání prvků do výstupního pole na správné pozice na základě vypočtených globálních offsetů, které známe z předešlého kroku.

Kapitola 6

Výsledky a měření

Po nastudování jednotlivých algoritmů z odborné literatury jsem vždy provedl i jejich implementaci. V případě CUDA varianty algoritmů bylo náplní této práce pouze nastudovat a navrhnout změny, které by vedly k optimálnímu běhu nad GPU. Já jsem se rozhodl algoritmy i implementovat, abych mohl porovnat výsledné časy z měření. V následujících odstavcích demonstruji své testovací prostředí a zhodnotím mnou naměřené časy.

6.1 Generování dat

Pomocí mnou napsaného bashového skriptu *skript-generuj-data*, který je uložen na CD v příloze této práce, jsem generoval data téměr uniformně náhodná. Kratší popis tohoto skriptu se nachází v příloze B této práce.

Tímto skriptem jsem nechal vygenerovat data o velikosti 10 000 000 resp. 100 000 000, které v rámci testování používám a referuji na ně jako $n = 10^7$ resp. $n = 10^8$.

Po vygenerování souborů obsahující neseřazená data jsem ještě vygeneroval ještě jiné zajímavé soubory, které mohou prověřit výkon mnou implementovaných algoritmů. Jedná se o tyto soubory:

- Náhodná data: Tyto data jsou výstupem z výše popsaného skriptu a vygenerovaný soubor používám k modifikaci, abych vygeneroval následující soubory.
- Seřazená data: Pomocí GNU sort aplikace jsem data seřadil.
- Téměř seřazená data: V úplně seřazeném souboru se veme k řádků a ty se přemístí na jiné místo. Tuto hodnotu jsem nastavil na 1000. K vytvoření těchto souborů jsem využil znalosti jazyku python a naprogramoval skript *sorted-almost.py*, který se nachází na přilozeném CD.

- Reverzně seřazená data: Pomocí GNU sort aplikace jsem data seřadil sestupně.

Zadání této práce mluví pouze o řazení struktur. Implementoval jsem všechny algoritmy tak, že umí řadit jak struktury, tak i samotné celočíselné hodnoty. To jsem udělal proto, abych mohl porovnat rozdíly v naměřených časech mezi řazeními těchto dvou různých datových typů. Jelikož struktura je paměťově náročnější nežli celočíselné hodnoty, pak očekávám, že řazení struktur bude o něco pomalejší.

6.2 Měřící prostředí

Vývoj aplikace i měření probíhalo na sestaveném počítači s následujícími hardwarovými parametry:

- procesor Intel i7-7770K (čtyřjádrový s technologií HT)
- základní deska MSI Z270 GAMING PRO CARBON
- operační paměť Kingston 2x8GB DDR4 3000MHz CL15
- SSD disk Samsung 960 EVO 250GB M.2 NVMe
- grafická karta Gigabyte GeForce AORUS GTX 1080 Ti Xtreme Edition 11GB

Na tento počítač jsem nainstaloval linuxovou distribuci Linux Mint 18.2, protože jsem ji shledal přívětivou a příjemnou k implementaci celého tohoto projektu. V době testování byly nainstalovány následující verze důležitých programů:

- linuxové jádro verze 4.8.0-53 z distribuce Linux Mint
- gcc verze 6.3.0
- CUDA SDK verze 9.0

6.3 Implementace

Implementace celého programu se nachází na přiloženém CD k této práci. Program lze zkompilovat pomocí přiloženého souboru *Makefile*. Vstupními parametry programu jsou:

1. parametr je cesta k souboru s daty k seřazení
2. parametr je algoritmus, který se má použít pro seřazení. Zde je výpis možných variant:

- qs – quicksort (sekvenční verze), viz kapitola 1.5
- mso – mergesort out–place (sekvenční verze), viz kapitola 1.6
- msi – mergesort in–place (sekvenční verze), viz kapitola 1.6.3
- ss – selectsort (sekvenční verze), viz kapitola 1.4.2
- rs – radixsort MSD (sekvenční verze), viz kapitola 1.7
- qso – quicksort (OpenMP verze), viz kapitola 3.1
- msoo – mergesort out–place (OpenMP verze), viz kapitola 3.2
- msio – mergesort in–place (OpenMP verze), viz kapitola 5.2.3
- rso – radixsort MSD (OpenMP verze), viz kapitola 5.2.4
- lsd – radixsort LSD (OpenMP verze), viz kapitola 5.2.5
- qscuda – quicksort (CUDA verze), viz kapitola 4.1
- mscuda – mergesort (CUDA verze), viz kapitola 4.2
- rscuda – radixsort LSD (CUDA verze), viz kapitola 4.3

Následující algoritmy, které lze použít v mé implementaci, nebyly mnou implenetované a byly přidány kvůli porovnání výsledků.

- aqsort – aqsort [28] (OpenMP verze)
- sort – sort z knihovny libstdc++ [29] (sekvenční verze)
- psort – paralelní sort z knihovny libstdc++ [30] (OpenMP verze)
- thrustsort – sort z knihovny thrust [31] (CUDA verze)

3. parametr určuje zdali na vstupu jsou celočíselné hodnoty (int) nebo struktury (struct)

4. parametr určuje počet vláken (jedná se o povinný parametr, využívá ho ale pouze OpenMP verze)

5. parametr definuje počet po sobě jdoucích spuštení algoritmu nad vstupními daty (nepočítají se do toho 3 běhy, které jsou v kódu implenentovány bez možnosti vypnutí) – výsledný čas na výstupu je již aritmeticky zprůměrován

6.4 Naměřené hodnoty

U každého algoritmu jsem měřil čas jeho doby běhu pomocí interní funkce `clock_gettime()` bez načítání vstupu a výpisu seřazených hodnot. V případě CUDA algoritmů jsem rozlišil dvě měření času a to včetně i bez doby strávené ve funkci `cudamemcpy()`. V následujících grafech není u CUDA variant algoritmů započteno do celkové času kopírování z CPU do GPU a to ani v opačném

směru. Tabulky v příloze A obsahují časy bez doby trvání algoritmu ve funkci `cudamemcpy()`. U všech algoritmů verze CUDA trvá funkce `cudamemcpy()` pro nakopírování dat z CPU do GPU a následně zpět u dat 10^7 cca 0.04 sekundy a u 10^8 cca 0.4 sekundy.

Každé měření pro jeden konkrétní algoritmus probíhalo tak, že nejprve proběhly 3 běhy a pak následně 20 běhů. První tři běhy, které se nezapočítavaly do výsledného času, jsem prováděl z důvodu zahřátí CPU resp. GPU, tak aby výsledky byly konzistentní a CPU resp. GPU se přeplo z úsporného režimu do normálního režimu. U zmíněných 20 běhů se z naměřených časů vypočítal aritmetický průměr a tato hodnota je ta finální, která se objevuje zde ve grafech resp. v tabulkách, která se nachází v příloze A této práce.

Data z grafů, které jsou prezentovány v této kapitole, jsou z měření řazení $n = 10^8$ struktur vstupního souboru. Kompletní tabulky s časy měření u počtu $n = 10^7$ a $n = 10^8$ struktur se nachází v příloze A.

Po doběhnutí posledního 20. měření jsem nechal výstup do souboru zkontrolovat programem GNU sort, abych si byl jistý, že algoritmus funguje korektně. Navíc jsem pomocí příkazu *wc -l* provedl kontrolu počtu řádků, že souhlasí se vstupem.

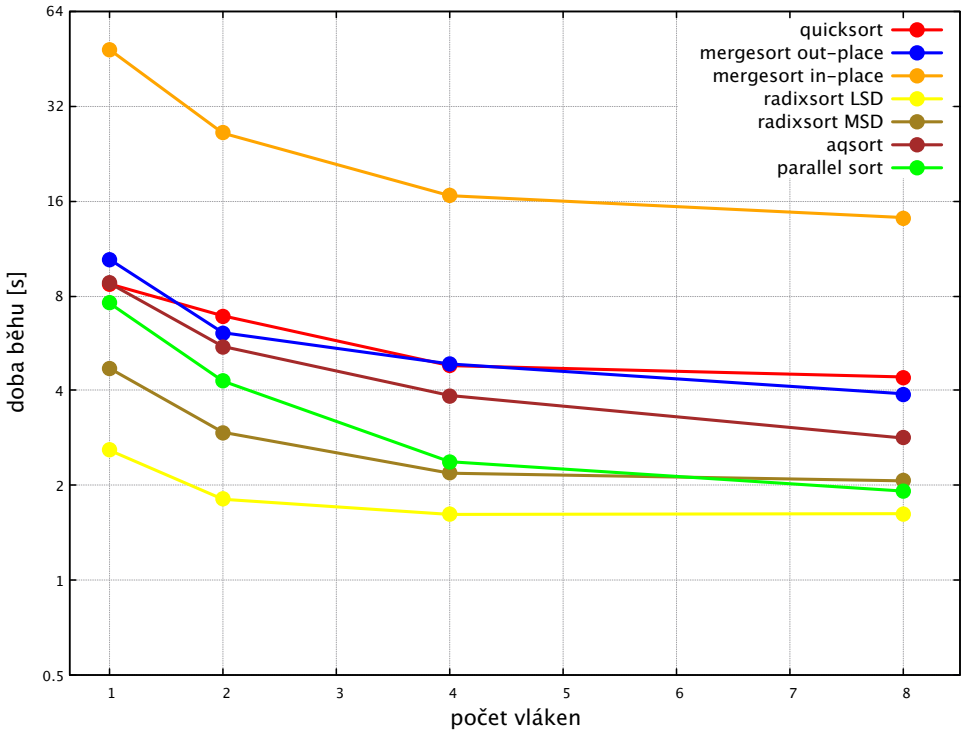
Očekávám, že výsledné časy v rámci jednotlivých algoritmů budou v tomto pořadí (od nejrychlejšího po nejpomalejší) u všech algoritmů mimo LSD variantu radixsortu:

1. seřazená data,
2. téměř seřazená data,
3. sestupně seřazená data a
4. náhodná data.

Dále očekávám, že řadící algoritmus radixsort bude nejrychlejší ve většině případů, protože nepatří do klasického porovnávacího modelu řadících algoritmů. Místo porovnání využívá struktury dat k řazení. Zároveň se domnívám, že algoritmus mergesort ve variantě in-place bude nejpomalejší, leč očekávám že by mohl mít rozumnou škálovatelnost v OpenMP verzi.

6.4.1 Porovnání OpenMP verzí

Ná následujících grafech 6.1, 6.2, 6.3, 6.4 je možné vidět zrychlení jednotlivých algoritmů v závislosti na počtu vláken u OpenMP variant algoritmů.

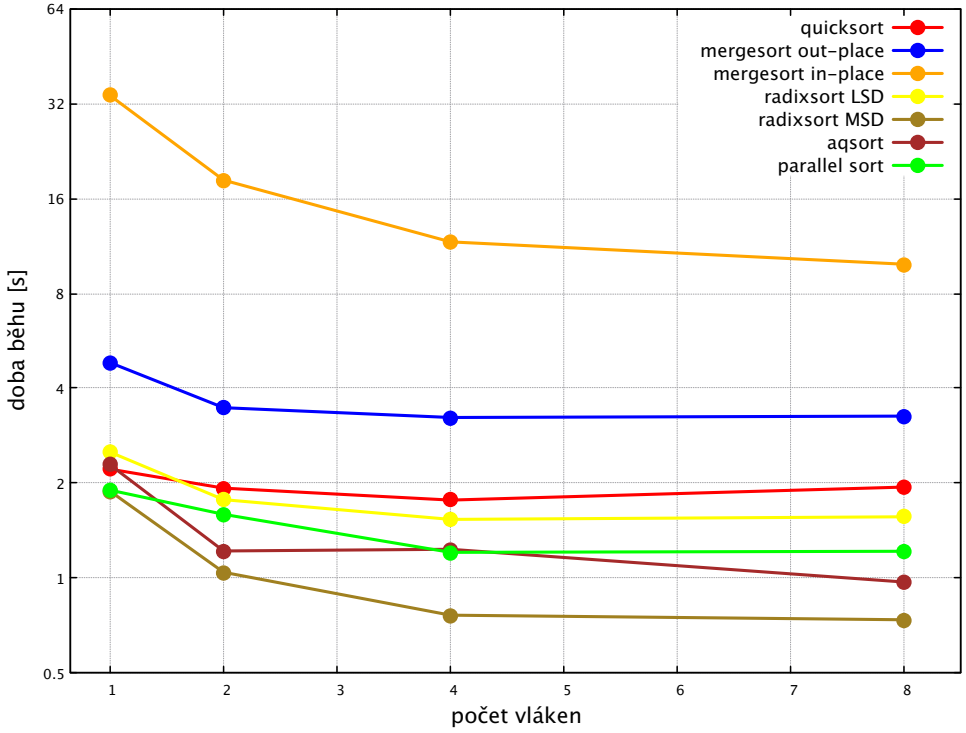


Obrázek 6.1: Naměřené doby běhu algoritmů u OpenMP verze nad náhodnými daty

Ze získaných časů doby běhu jednotlivých algoritmů u náhodných dat lze konstatovat, že mergesort in-place varianta je opravdu nejpomalejší, leč má dobrou asymptotickou složitost, tak její multiplikativní konstanty jsou vysoké. Radixsort ve variantě LSD má nejlepší čas, což je z důvodu, že využívá struktury dat místo porovnání. Nejrychlejším algoritmem ze skupiny algoritmů, které řadíme do porovnávacího modelu, je paralelní verze *sort* z libstdc++ knihovny kvůli optimalizacím, které se nachází v implementaci tohoto algoritmu. Hned za ním se nachází algoritmus aqsort, který předběhl jak mou implementaci quicksortu, tak i mergesortu. U tohoto vstupního souboru se často plete predikce procesorů, což je důvod vyšších časů doby běhů.

U varianty vstupu se seřazenými daty je vidět, že se všechny algoritmy zrychlily, což je opět očekávatelný výsledek. Jediným algoritmem, který nedospěl k výraznému zrychlení je LSD varianta radixsortu a to z důvodu, že tento algoritmus musí i na seřazených datech přehazovat prvky, protože začíná od nejméně významné číslice. Naopak MSD, který řadí od nejvíce významné číslice, nemusí přehazovat tak jako LSD a proto došlo v tomto případě ke

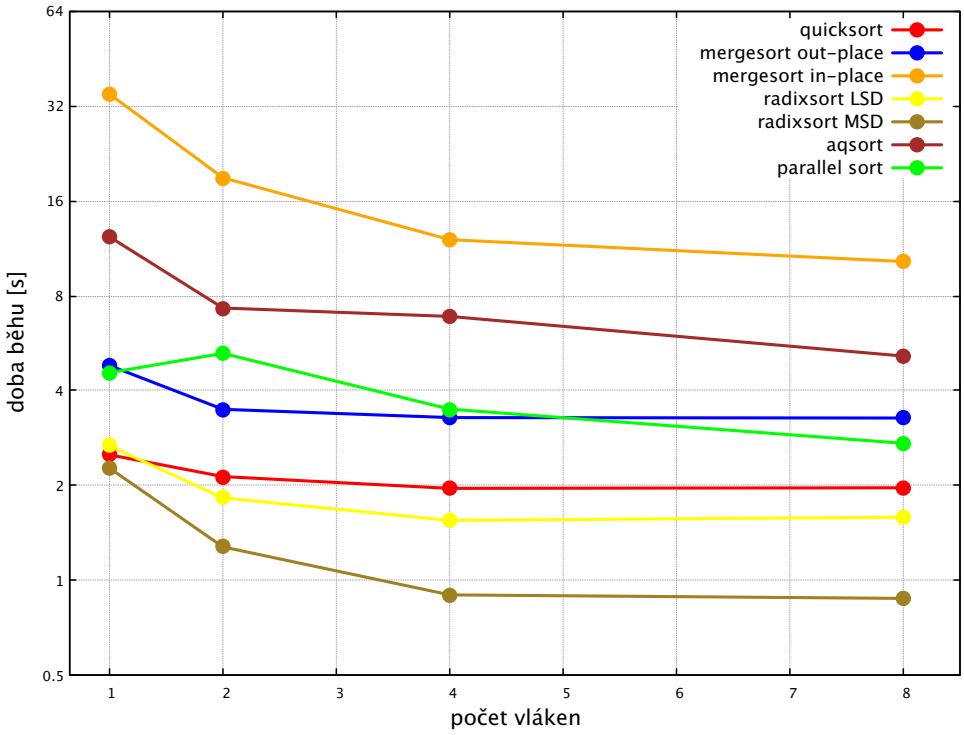
zrychlení. V tohoto souboru vstupních dat algoritmus aqsort zvítězil nad paralelním sortem z libstdc++. V tomto případě se mimo radixsort LSD procesor většinou neplete s predikcí, vše je pěkně za sebou zarovnané v paměti a proto algoritmy řadí rychle.



Obrázek 6.2: Naměřené doby běhu algoritmů u OpenMP verze nad již seřazenými daty

V případě vstupního souboru obsahující téměř seřazená data je očekávaný výsledek velmi podobný jako u plně seřazených dat. Algoritmy mají opravdu dle výsledných časů mimo aqsort a paralelní sort z libstdc++ podobné časy jako u plně seřazeného vstupního souboru. V případě aqsortu došlo k výrazném zhoršení času. U paralelní sortu z libstdc++ došlo sice k menšímu zhoršení, ale jak je vidno na grafu, tak v běhu s 2 vlákny došlo ke zhoršení oproti sekvenčnímu běhu. Z důvodu, že vstupní data jsou téměř seřazená, tak predikce procesoru bude opět dobrá s občasným chybováním.

U posledního měřeného typu vstupního souboru (sestupně seřazená data) je očekávatelné, že vyjma LSD algoritmus budou výsledné časy o hodně rychlejší nežli u náhodných dat a jen o trochu pomalejší nežli u seřazených dat. Na

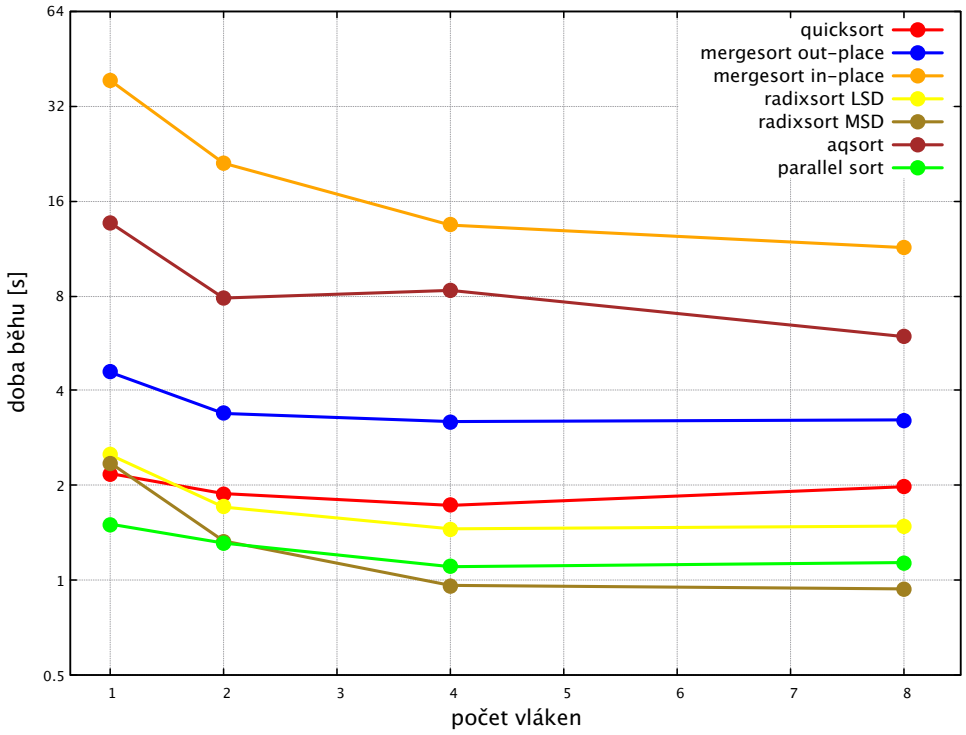


Obrázek 6.3: Naměřené doby běhu algoritmů u OpenMP verze nad téměř seřazenými daty

základě naměřených výsledků je vidět, že výsledky jsou velmi podobné jako u seřazeného souboru dat, u většiny případů došlo k mírnému očekávanému zhoršení a u některých k relativně zanedbatelnému zhoršení. Vyjímku tvoří algoritmus aqsort, který na tomto vstupu podává hned druhý nejhorší výsledek hned za algoritmem mergesort in-place. V tomto případě bude predikce procesoru opět vynikající.

Algoritmus aqsort je implementován jako hybridní verze quicksortu, který když rekurzí již příliš hluboku, tak se přepně na heapsort. Tento algoritmus má implementované paralelní rozdělení do pole. Aqsort porazil všechny mnou implementované OpenMP verze algoritmů patřících do porovnávacího modelu v náhodných a seřazených dat. Neočekávané chování tohoto algoritmu u téměř seřaných a sestupně seřazených dat si neumím vysvětlit a zajisté by bylo vhodné důkladnější prozkoumání příčiny tohoto problému.

Aqsort je velmi rychlý, viz konkrétní časy v tabulce v příloze A. Po prozkoumání kódu se lze dočíst, že se jedná o quicksort algoritmus se speciálním



Obrázek 6.4: Naměřené doby běhu algoritmů u OpenMP verze nad sestupně seřazenými daty

výběrem mediánu. Nejdříve se vstupní data rozdělí na tři části. Z každé této části se vybere první, prostřední a poslední prvek a následně se z těchto tří prvků vybere jeden medián. A takto se to provede u každé ze tří částí až nám zbydou tři mediány a z nich se vybere medián finální.

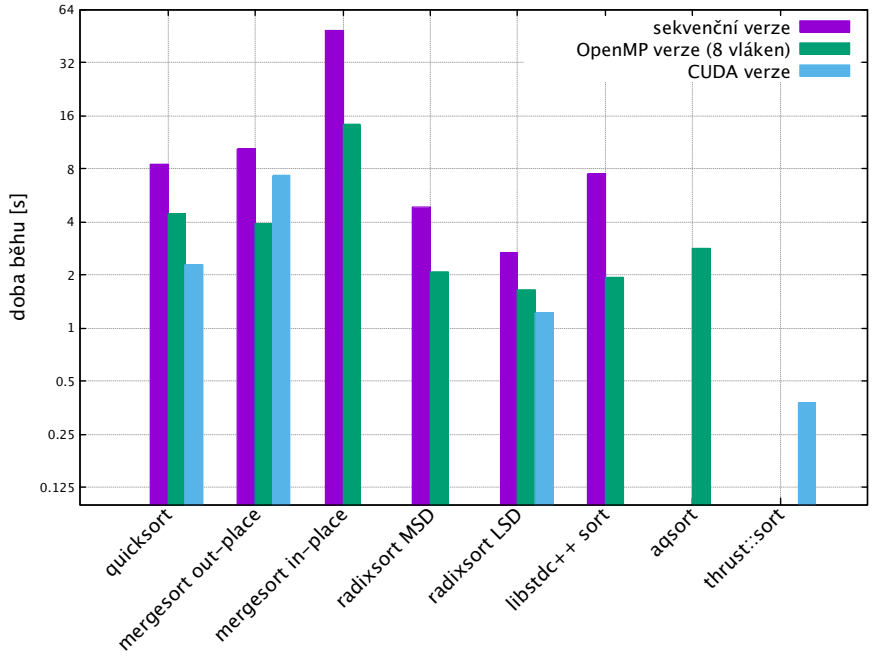
Algoritmus radixsort ve variantě LSD je relativně nezávislý na vstupním souboru dat a podává stále konzistentní výsledky. Důvodem je, že leč predikce procesoru bude často mylná, tak sám o sobě je velmi rychlý, protože využívá struktury řazených dat.

Algoritmus radixsort ve variantě MSD nemá implementovanou, narozdíl od LSD varianty, paralelizaci rozřazování do škatulek, ale i přesto je velmi rychlý. U tohoto algoritmu bude naopak predikce procesoru velmi dobrá.

6.4.2 Porovnání všech verzí

Následující dva grafy porovnávají algoritmy v různých verzích – sekvenční, OpenMP a CUDA. Zvolil jsem k demonstraci pomocí grafů pouze verze s ná-

hodnými a plně seřazenými daty.



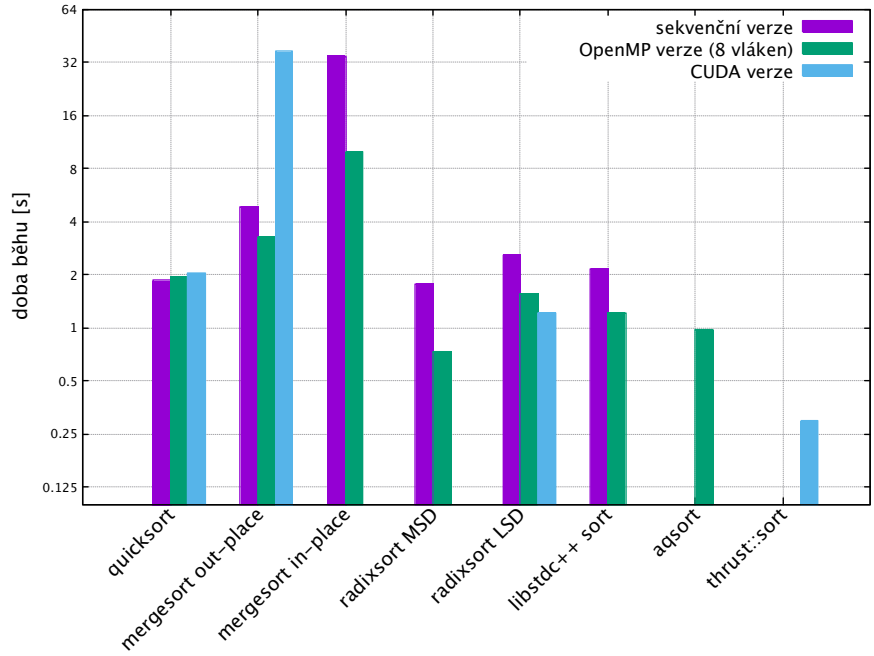
Obrázek 6.5: Naměřené doby běhu algoritmů u sekv., OpenMP a CUDA verze nad náhodnými daty

Verze použitých řadících algoritmů z knihoven libstdc++ a thrust je závislá na verzi nainstalované knihovny libstdc++ respektive CUDA SDK. V tomto i předchozím měření byly konkrétně použity algoritmy sekvenční sort a paralelní sort z knihovny libstdc++ verze 3.4.24 a metoda sort z knihovny thrust ve verzi 1.7.0, která je standartně přibalena do CUDA SDK verze 9.

Mergesort ve variantě CUDA u náhodných dat je relativně pomalý. Důvodem je, že implementovaná funkce slévání je neoptimální pro GPU (byl pouze převzat princip z implementované slévací funkce z OpenMP varianty a přepsán do CUDA varianty – paralelní dvoucestné slévání). Pro zvýšení rychlosti by bylo zapotřebí optimalizovat vlákna v bloku tak, aby v jednu chvíli prováděla stejné instrukce.

Dále je mergesort ve variantě CUDA o hodně pomalejší u seřazených dat. Důvodem zpomalení je, že námi implementované paralelní dvoucestné slévání rozdělí levou posloupnost na stejné části a k nim hledá odpovídající úseky v pravé části, které jsou v případě seřazených dat prázdné až na jeden úsek a tím je celá posloupnost.

Výše popsaný problém je u OpenMP verze také, ale je překrytý tím, že je vytvořen značně menší počet vláken nežli na GPU a proto se tento problém neprojeví ve větší míře. Výkon u OpenMP verze je dost závislý na úspěšnosti predikce procesoru a ta je v tomto případě dobrá (u seřazených dat).



Obrázek 6.6: Naměřené doby běhu algoritmů u sekv., OpenMP a CUDA verze nad seřazenými daty

Na základě tabulek, které jsou obsažené v příloze této práce, lze získat výsledné zrychlení konkrétního řadícího algoritmu v mé implementaci při běhu OpenMP (8 vláken) vůči sekvenčnímu a CUDA vůči sekvenčnímu viz tabulky 6.1 a 6.2. V případě, že algoritmus není implementován v sekvenční variantě, pak jsem do vyhodnocení zahrnul OpenMP běh s 1 vláknem – to se týká algoritmů aqsort a radixsort LSD. Na základě hodnot z těchto tabulek lze konstatovat, že tyto algoritmy lépe škálují u náhodných dat než u seřazených dat. Zároveň lze konstatovat, že mnou implementované algoritmy v CUDA variantě u seřazených dat podávají podprůměrné výsledky.

Po prostudování výsledných naměřených časů z tabulek v příloze A lze mimo jiné také konstatovat následující:

- Paralelní sort z knihovny libstdc++ a aqsort relativně lépe škálují vůči mým implementacím algoritmů.

	OpenMP	**CUDA**
quicksort	1.92	3.71
mergesort out-place	2.65	1.42
mergesort in-place	3.43	x
radixsort MSD	2.33	x
radixsort LSD	1.63	2.18
aqsort	3.10	x
libstdc++ (p)sort	3.91	x

Tabulka 6.1: Zrychlení OpenMP a CUDA verze vůči sekvenční verzi u náhodných dat s daty $n = 10^8$

	OpenMP	**CUDA**
quicksort	0.96	0.92
mergesort out-place	1.49	0.13
mergesort in-place	3.50	x
radixsort MSD	2.40	x
radixsort LSD	1.65	2.13
aqsort	2.35	x
libstdc++ (p)sort	1.78	x

Tabulka 6.2: Zrychlení OpenMP a CUDA verze vůči sekvenční verzi u seřazených dat s daty $n = 10^8$

- Obecně u seřazených (vzestupně, sestupně, téměř) dat dochází většinou k menšímu zrychlení u OpenMP variant, mimo LSD variantu radixsortu, která je přibližně stále stejně rychlá napříč všemi zde testovanými vstupními soubory.
- Algoritmus mergesort ve variantě in-place je sice rozhodně zde z prezentovaných algoritmů nejpomalejší, ale velmi dobře škáluje.
- U některých algoritmů je běh pro 4 vlákna mírně rychlejší nežli u běhu nad 8 vlákny. Myslím si, že toto je způsobeno režií vytváření a správy OpenMP vláken.

Závěr

Cílem této diplomové práce bylo, aby se student seznámil s různými řadícími algoritmy a poreferoval o možných způsobech jejich paralelizace v OpenMP a CUDA nástrojích. Rešerše byla provedena z mnoha zdrojů.

Paralelizace algoritmů v prostředí OpenMP mi přišla relativně intuitivní, na druhou stranu paralelizace algoritmů v prostředí CUDA mi přišla obtížnější a to z důvodu hledání chyb, které bylo obtížnější kvůli značně velkému počtu vytvořených vláken.

Doufám, že tato práce pomůže studentům či jiným osobám pochopit problematiku některých konkrétních řadících algoritmů a jejich možné paralelizace popsanými způsoby. Ty rozhodně nejsou všechny a zajisté se v odborné literatuře můžeme setkat i s dalšími variantami paralelizace. Byl bych rád, kdyby na tuto práci navázali studenti bakalářskou či diplomovou prací. Zde přikládám možné návrhy rozšíření této práce:

- změna funkce `pivot()` u quicksortu na několik různých způsobů výběru pivota a jejich následné porovnání,
- změna paralelního 2-cestného slévání na vícecestné slévání a to jak u OpenMP tak i CUDA verze,
- implementace Merge Path algoritmu do mergesortu u CUDA varianty,
- paralelizace mergesortu ve variantě in-place v prostředí CUDA,
- paralelizace funkce slévání u mergesortu v in-place variantě u OpenMP verze
- kvalitnější paralelizace GPU algoritmů, zejména tak, aby přístupy do paměti byly za sebou následující a vlákna v rámci bloku prováděla stejné instrukce a nečekala na sebe.

Tvorbou této diplomové práce jsem nastudoval velké množství materiálů o řadících algoritmech a lépe pochopil jejich konkrétní problematiku, převážně co se týče jejich paralelizace. Zároveň jsem rád, že tato práce měla i praktickou část a při implementaci jsem si mohl vyzkoušet paralelizaci pomocí technologie CUDA. Ta má podle mého názoru velkou budoucnost a velký potenciál. Důležité ale je, aby paralelní algoritmy byly implementovány efektivně, což není žádný triviální úkol, jak jsem si mohl vyzkoušet.

Literatura

[1] SEDGEWICK, Robert. *Algoritmy v C*. Praha: SoftPress, 2003. ISBN 80-864-9756-9.

[2] QuickSort. *Vishnu's front page* [online]. [cit. 2017-02-15]. Dostupné z: https://www.cp.eng.chula.ac.th/~vishnu/datastructure/QuickSort.pd

[3] SEDGEWICK, Robert. *Algorithms in C*. Reading, Mass.: Addison-Wesley Pub. Co., 1990. ISBN 0-201-51425-7.

[4] Parallel Algorithms - sorting. *Fernando Silva* [online]. [cit. 2017-02-27]. Dostupné z: http://www.dcc.fc.up.pt/~fds/aulas/PPD/1112/sorting.pdf

[5] Optimal Stable Merging. *Antonios Symvonis* [online]. [cit. 2017-03-10]. Dostupné z: https://ai2-s2-pdfs.s3.amazonaws.com/771e/a2251dde91415ea002b3697cdfde0117a81e.pdf

[6] CHAPMAN, Barbara, Gabriele. JOST a Ruud van der. PAS. *Using OpenMP: portable shared memory parallel programming.* Cambridge, Mass.: MIT Press, 2008. ISBN 9780262533027.

[7] Parallel Sorting. *Michael Hanke* [online]. [cit. 2017-10-25]. Dostupné z: https://www.math.kth.se/na/SF2568/parpro-17/F7.pdf

[8] CHENG, John. *Professional CUDA C programming.* Indianapolis: John Wiley, 2014. ISBN 9781118739327.

[9] NVIDIA CUDA C Programming Guide. *NVIDIA Corporation* [online]. [cit. 2017-10-31]. Dostupné z: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[10] prof. Pavel Tvrdík: *skripta z předmětu PI-PPA* [online]. [cit. 2017-11-29]. Dostupné z WWW: <https://edux.fit.cvut.cz/oppa/PI-PPA/prednasky/PI-PPA2011-1.pdf>

[11] Pavel Řehák.: *Emulátor činnosti skrytých pamětí pro sdílenou paměť*. Bakalářská práce, ČVUT FIT, 2013

[12] LO, Dan. *Finite element mesh generation*. Boca Raton, 2015. ISBN 978-041-5690-485.

[13] WEIRAN, Nie. *EECS221 Final Project-Parallel Radix Sort Using OpenMP* [online]. [cit. 2017-11-29]. Dostupné z: https://wenku.baidu.com/view/3ba40a0c6c85ec3a87c2c592.html

[14] Ing. Daniel Langr, Ph.D..: *skripta z předmětu MI-PDP* [online]. [cit. 2017-11-09]. Dostupné z WWW: <https://edux.fit.cvut.cz/archive/B162/MI-PDP.16/_media/lectures/mi-pdplecture06-openmpsorting.pdf>

[15] Sort. *Thrust::sort* [online]. [cit. 2017-11-30]. Dostupné z: http://www.sgi.com/tech/stl/sort.html

[16] Cederman, Daniel & Tsigas, Philippas. (2009). *GPUQuicksort: A practical Quicksort algorithm for graphics processors* [online]. ACM Journal of Experimental Algorithmics. Dostupné z http://www.cse.chalmers.se/research/group/dcs/TechReports/gpuqsort.pdf

[17] C. A. R. HOARE. Quicksort [online]. , 10-15 [cit. 2017-12-02]. Dostupné z WWW: <https://academic.oup.com/comjnl/article/5/1/10/395338>

[18] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, Yitzhak Birk. *Merge Path - Parallel Merging Made Simple* [online]. [cit. 2017-12-03]. Dostupné z WWW: <https://pdfs.semanticscholar.org/891f/e4ab8700b780184b6f5307a4cbe9cfcda8f4.pdf>

[19] TVRDÍK, Pavel. *Paralelní systémy a algoritmy*. Vyd. 2. V Praze: Nakladatelství ČVUT, 2006. ISBN 80-010-3565-4.

[20] MAREŠ, Martin a Tomáš VALLA. *Průvodce labyrintem algoritmů*. Praha: CZ.NIC, z.s.p.o., 2017. CZ.NIC. ISBN 978-80-88168-19-5.

[21] UNIVERSITY OF TENNESSEE, Knoxville, Tennessee. textitMPI: A Message-Passing Interface Standard: Version 3.1 [online]. [cit. 2017-12-12]. Dostupné z: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[22] *Státnice - Třídění* [online]. [cit. 2017-12-13]. Dostupné z: http://wiki.matfyz.cz/index.php?title=Státnice_-_Tř%C3%ADděn%C3%AD#Hled.C3.A1n.C3.AD_medi.C3.A1nu_technikou_rozd.C4.9B_a_panuj_.28algoritmus_FIND.29

[23] *Introduction & Median Finding* [online]. [cit. 2017-12-13]. Dostupné z: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec01.pdf

[24] SLUSALLEK, Philipp. *Sorting in Parallel* [online]. [cit. 2017-12-13]. Dostupné z: https://graphics.cg.uni-saarland.de/fileadmin/cguds/courses/ss14/pp_cuda/slides/06_-_Sorting_in_Parallel.pdf

[25] SATISH, Nadathur, Mark HARRIS a Michael GARLAND. *Designing Efficient Sorting Algorithms for Manycore GPUs* [online]. [cit. 2017-12-13]. Dostupné z: https://pdfs.semanticscholar.org/0e29/93ddba78626376651c3ab8d14f0d680f0595.pdf

[26] ZAGHA, Marco a Guy E. BLELLOCH. *Radix Sort For Vector Multiprocessors* [online]. [cit. 2017-12-13]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.51.7376&rep=rep1&type=pdf

[27] Oded Green, Robert McColl, David A. Bader. *GPU Merge Path - A GPU Merging Algorithm* [online]. [cit. 2017-12-15]. Dostupné z https://www.researchgate.net/profile/Oded_Green/publication/254462662_GPU_merge_path_a_GPU_merging_algorithm/links/543eeaa00cf2e76f02244884/GPU-merge-path-a-GPU-merging-algorithm.pdf

[28] *AQSORT: SCALABLE MULTI-ARRAY IN-PLACE SORTING WITH OPENMP* [online]. 2016, 2016(4) [cit. 2017-12-20]. ISSN 1895-1767. Dostupné z: http://www.scpe.org/index.php/scpe/article/download/1207/492

[29] *Sorting Algorithms* [online]. [cit. 2017-12-20]. Dostupné z: https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01027.html#g152148508b4a39e15ffbfbc987ab653a

[30] *Parallel Mode* [online]. [cit. 2017-12-20]. Dostupné z: https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode_using.html

[31] *Thrust: Sorting* [online]. [cit. 2017-12-20]. Dostupné z: https://thrust.github.io/doc/group___sorting.html

[32] CHANDY, K. Mani a Jayadev MISRA. *Parallel program design: a foundation*. Reading: Addison-Wesley Publishing Company, c1998. ISBN 02-010-5866-9.

[33] BRINCH HANSEN, Per. *Studies in computational science: parallel programming paradigms*. Englewood Cliffs, N.J.: Prenctice Hall, c1995. ISBN 978-013-4393-247.

[34] TIMOTHY G. MATTSON, BEVERLY A. SANDERS a BERNA L. MASSINGILL. *Patterns for parallel programming*. Reading: Addison-Wesley, 2013. ISBN 978-032-1940-780.

[35] QUINN, Michael J. *Parallel computing: theory and practice*. 2nd ed. New York: McGraw-Hill, c1994. ISBN 978-0070512948.

[36] BAMPIS, Evripidis, ed. *Experimental Algorithms*: 14th International Symposium, SEA 2015. ISBN 9783319200859.

[37] KALORKOTI, Kyriakos. *Inf2B Algorithms and Data Structures Note 8: Heapsort and Quicksort* [online]. [cit. 2018-01-02]. Dostupné z: https://www.inf.ed.ac.uk/teaching/courses/inf2b/algnotes/note08.pdf

[38] Bolosky, W. J. and Scott, M. L. 1993. *False sharing and its effect on shared memory performance* In 4th USENIX Symposium on Experiences with Distributed and Multiprocessor Systems, San Diego, California, September 22–23, 1993. USENIX Association, Berkeley, CA, 3-3.

[39] *The Basics of Caches* [online]. [cit. 2018-01-04]. Dostupné z: https://cseweb.ucsd.edu/classes/su07/cse141/cache-handout.pdf

[40] prof. Ing. Pavel Tvrdík CSc.: *skripta z předmětu MI-PPA* [online]. [cit. 2018-01-04]. Dostupné z WWW: <https://edux.fit.cvut.cz/oppa/PI-PPA/prednasky/PI-PPA2011-1.pdf>

[41] CHANDRASEKARAN, Siddharth. *Implementing Circular/Ring Buffer in Embedded C* [online]. [cit. 2018-01-02]. Dostupné z: http://embedjournal.com/implementing-circular-buffer-embedded-c/

Příloha A

Naměřené hodnoty z implementace

Veškeré zde přítomné tabulky a časy v nich uvedené reprezentují dobu běhu algoritmu ve vteřinách bez načítání vstupu ze souboru do RAM a výstupu seřazené posloupnosti. Implicitně do uvedených CUDA algoritmů nebyl započítán čas strávený ve funkci `cudamemcpy()` potřebný ke kopírování dat mezi CPU a GPU částí.

	náhodná	**seřazená**	**sestupně seř.**	**téměř seř.**
quicksort	0.725343	0.150664	0.160376	0.1987
mergesort out-place	0.880723	0.377942	0.362396	0.378528
mergesort in-place	4.08434	2.88221	3.23967	2.96086
radixsort MSD	1.11899	0.825843	0.930545	0.916331
radixsort LSD	0.236982	0.233834	0.236581	0.234476
libstdc++ sort	0.655587	0.176843	0.145561	0.443273

Tabulka A.1: Řazení 10^7 struktur pomocí sekvenčních algoritmů

	náhodná	**seřazená**	**sestupně seř.**	**téměř seř.**
quicksort	8.47417	1.86211	1.97541	2.2101
mergesort out-place	10.3614	4.86529	4.70607	4.86658
mergesort in-place	48.6087	34.6462	39.3544	34.8526
radixsort MSD	4.83099	1.76906	2.46858	2.31247
radixsort LSD	2.65807	2.58254	2.6255	2.56007
libstdc++ sort	7.51345	2.15978	1.76097	4.92309

Tabulka A.2: Řazení 10^8 struktur pomocí sekvenčních algoritmů

	1	**2**	**4**	**8**	**poměr**
quicksort	0.74720	0.52128	0.36375	0.28429	2.63
mergesort out-place	0.88352	0.51772	0.41570	0.33152	2.67
mergesort in-place	4.06388	2.24031	1.47499	1.27252	3.19
radixsort MSD	1.10993	0.53917	0.35541	0.35530	3.12
radixsort LSD	0.24108	0.17217	0.15460	0.15112	1.60
aqsort	0.7697	0.52913	0.34242	0.23219	3.31
libstdc++ psort	0.65179	0.37040	0.20066	0.16733	3.90

Tabulka A.3: Řazení 10^7 struktur pomocí OpenMP algoritmů u náhodných dat nad 1, 2, 4 a 8 vlákny s vypočteným poměrem zrychlení mezi 1 a 8 vláknovým během

	1	**2**	**4**	**8**	**poměr**
quicksort	8.72087	6.90974	4.79719	4.42492	1.97
mergesort out-place	10.4403	6.10821	4.86063	3.90945	2.67
mergesort in-place	48.2917	26.2033	16.5979	14.1541	3.41
radixsort MSD	4.71265	2.94567	2.18857	2.07369	2.27
radixsort LSD	2.5921	1.80684	1.62358	1.63442	1.59
aqsort	8.77419	5.51793	3.86157	2.83129	3.10
libstdc++ psort	7.57986	4.28637	2.37503	1.91969	3.95

Tabulka A.4: Řazení 10^8 struktur pomocí OpenMP algoritmů u náhodných dat nad 1, 2, 4 a 8 vlákny s vypočteným poměrem zrychlení mezi 1 a 8 vláknovým během

	1	**2**	**4**	**8**	**poměr**
quicksort	0.16834	0.14774	0.14846	0.17482	0.96
mergesort out-place	0.38987	0.26929	0.27088	0.25345	1.54
mergesort in-place	2.89804	1.57785	1.05722	0.91982	3.15
radixsort MSD	0.84883	0.36787	0.21318	0.22805	3.72
radixsort LSD	0.23774	0.16702	0.14951	0.15231	1.56
aqsort	0.18456	0.10406	0.12287	0.08446	2.18
libstdc++ psort	0.15577	0.12957	0.11073	0.09574	1.63

Tabulka A.5: Řazení 10^7 struktur pomocí OpenMP algoritmů u seřazených dat nad 1, 2, 4 a 8 vlákny s vypočteným poměrem zrychlení mezi 1 a 8 vláknovým během

	1	2	4	8	poměr
quicksort	2.21758	1.93004	1.76813	1.94385	1.14
mergesort out-place	4.82204	3.47298	3.23171	3.26516	1.48
mergesort in-place	33.8846	18.2819	11.6402	9.89599	3.42
radixsort MSD	1.88711	1.04448	0.76343	0.73819	2.56
radixsort LSD	2.51015	1.77438	1.53582	1.56601	1.60
aqsort	2.28619	1.21984	1.23527	0.97194	2.35
libstdc++ psort	1.90161	1.59301	1.20721	1.21617	1.56

Tabulka A.6: Řazení 10^8 struktur pomocí OpenMP algoritmů u seřazených dat nad 1, 2, 4 a 8 vlákny s vypočteným poměrem zrychlení mezi 1 a 8 vláknovým během

	1	2	4	8	poměr
quicksort	0.17488	0.15517	0.15185	0.17993	0.97
mergesort out-place	0.36825	0.26265	0.23952	0.24756	1.49
mergesort in-place	3.23769	1.79588	1.23418	1.03395	3.13
radixsort MSD	0.92902	0.40327	0.24183	0.25036	3.71
radixsort LSD	0.23803	0.17266	0.14716	0.15095	1.58
aqsort	1.26624	0.69312	0.73861	0.50714	2.50
libstdc++ psort	0.12406	0.11160	0.09985	0.09106	1.36

Tabulka A.7: Řazení 10^7 struktur pomocí OpenMP algoritmů u sestupně seřazených dat nad 1, 2, 4 a 8 vlákny s vypočteným poměrem zrychlení mezi 1 a 8 vláknovým během

	1	2	4	8	poměr
quicksort	2.18376	1.88572	1.73061	1.98402	1.10
mergesort out-place	4.57699	3.38716	3.19043	3.23129	1.42
mergesort in-place	38.572	21.0382	13.3994	11.3588	3.40
radixsort MSD	2.36119	1.33552	0.96540	0.94069	2.51
radixsort LSD	2.50445	1.70986	1.45824	1.48813	1.68
aqsort	13.6427	7.86442	8.31526	5.93054	2.30
libstdc++ psort	1.5094	1.31474	1.10697	1.14097	1.32

Tabulka A.8: Řazení 10^8 struktur pomocí OpenMP algoritmů u sestupně seřazených dat nad 1, 2, 4 a 8 vlákny s vypočteným poměrem zrychlení mezi 1 a 8 vláknovým během

	1	**2**	**4**	**8**	**poměr**
quicksort	0.21469	0.18294	0.17903	0.17958	1.20
mergesort out-place	0.39213	0.27244	0.25879	0.25087	1.56
mergesort in-place	2.99091	1.7023	1.10086	0.95425	3.13
radixsort MSD	0.90755	0.39729	0.23565	0.24852	3.65
radixsort LSD	0.23519	0.16821	0.14932	0.15165	1.55
aqsort	1.02956	0.55285	0.59192	0.35867	2.87
libstdc++ psort	0.41932	0.43293	0.27035	0.23280	1.80

Tabulka A.9: Řazení 10^7 struktur pomocí OpenMP algoritmů u téměř seřazených dat nad 1, 2, 4 a 8 vlákny s vypočteným poměrem zrychlení mezi 1 a 8 vláknovým během

	1	**2**	**4**	**8**	**poměr**
quicksort	2.50329	2.1369	1.96059	1.96936	1.27
mergesort out-place	4.80702	3.49041	3.28723	3.27587	1.47
mergesort in-place	34.9051	18.9211	12.0416	10.2687	3.40
radixsort MSD	2.27391	1.27762	0.90013	0.878123	2.59
radixsort LSD	2.67554	1.82891	1.55078	1.58739	1.69
aqsort	12.3043	7.30926	6.87705	5.14504	2.39
libstdc++ psort	4.55888	5.25144	3.4933	2.72266	1.67

Tabulka A.10: Řazení 10^8 struktur pomocí OpenMP algoritmů u téměř seřazených dat nad 1, 2, 4 a 8 vlákny s vypočteným poměrem zrychlení mezi 1 a 8 vláknovým během

	náhodná	**seřazená**	**sestupně seř.**	**téměř seř.**
quicksort	0.323608	0.342117	0.346795	0.331168
mergesort out-place	0.209175	3.50769	5.19065	1.8421
radixsort LSD	0.193867	0.191953	0.191675	0.192014
thrust::sort	0.0293144	0.0205739	0.0216428	0.0209466

Tabulka A.11: Řazení 10^7 struktur pomocí CUDA algoritmů

	náhodná	**seřazená**	**sestupně seř.**	**téměř seř.**
quicksort	2.28558	2.02854	2.01238	2.04661
mergesort out-place	7.2713	36.9856	53.4047	35.457
radixsort LSD	1.219886	1.21288	1.20534	1.21384
thrust::sort	0.375451	0.295782	0.303283	0.301704

Tabulka A.12: Řazení 10^8 struktur pomocí CUDA algoritmů

Příloha **B**

Skript pro generování dat

Generátor přijímá dva vstupy – počet hodnot, které má vygenerovat a druhým argumentem je jedna z hodnot *{int, struct, both}*. Výstup je automaticky do nově vytvořených souborů, každý záznam je uložen na nový řádek. První argument vstupu je celočíselná hodnota, která definuje kolik hodnot bude do výstupního souboru vygenerováno. Význam druhého argumentu popisují následující řádky:

- Parametr *int*: Generátor vygeneruje celošíselné hodnoty v uzavřeném intervalu $< 1; 3276832767 >$.
- Parametr *struct*: Generátor vygeneruje celošíselné hodnoty v intervalu $< 1; 3276832767 >$, které jsou následovany mezerou a poté náhodným osmiznakovým řetězcem, který se skládá z číslic i písmen. Tento náhodný řetězec slouží jako data, náhodné číslo pak slouží jako klíč, dle kterého se bude řadit.
- Parametr *both*: Tato metoda vygeneruje oba dva výstupní soubory, tak jak jsou popsány v bodech výše.

Příloha C

Seznam použitých zkratek

CUDA (Compute Unified Device Architecture) je nástroj od nVidia sloužící k paralelizaci algoritmů na grafických kartách nVidia

SDK (Software Development Kit) je soubor nástrojů pro vývoj software

C/C++ je programovací jazyk

GPU (Graphic Processing Unit) je grafický procesor, umožňující rychlé grafické výpočty

CPU (Central Processing Unit) je základní součástka počítače, která vykonává strojové instrukce a tím je tvořen počítačový program

OMP (Open Multi-Processing) je aplikační rozhraní sloužící k paralelizaci algoritmů nad CPU

SMP (Symmetric multiprocessing) je druh víceprocesorového systému ve kterém jsou si všechny procesory rovnocené

SMs (Multithreaded Streaming Multiprocessor) je označení pro pole procesorů na nVidia GPU

SP (Streaming Processor) odpovídá tzv. CUDA jádru

cache je skrytá paměť

DRAM (Dynamic Random Access Memory) je dynamická paměť

offset je celočíselná hodnota určující rozdíl v pozici mezi dvěma prvky v poli

PRAM (Parallel Random Access Machines) je model paralelního systému se sdílenou pamětí

CREW (Concurrent Read Exclusive Write) je jeden z možných přístupu u PRAM modelu, kdy jednu paměťovou buňku více uzlů může číst, ale pouze jeden uzel do ni smí zapisovat v jednu chvíli

BÚNO znamená bez újmy na obecnosti

APU (Accelerated Processing Unit) je čip do kterého je integrována jádra jak z CPU, tak i z GPU

DSP (Digital Signal Processor) je čip určený pro operace s digitálním signálem, např. audiem

PPS (Parallel Prefix Sum) je metoda výpočtu prefixového součtu nad polem v paralelní verzi

Příloha D

Obsah přiloženého CD

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