Semestrální práce z předmětu KIV/PPR

Standardní zadání

Hledání percentilu v souboru

Stanislav Král

A20N0091P

Západočeská univerzita v Plzni
Fakulta aplikovaných věd

Obsah

1 Zadání **2**

2 Analýza **3**
2.1 Percentil 3
2.1.1 Vzorec 3
2.1.2 Vyhledání v souboru s omezenou operační pamětí . . . 3
2.2 Paralelizace vyhledání percentilu 4

3 Popis implementace **8**
3.1 Sériová implementace 8
3.2 Vícevláknová implementace 9
3.3 Implementace výpočtu s využitím OpenCL 10
3.4 Naivní implementace 11

4 Měření výkonnosti vytvořeného programu **12**
4.1 Konfigurace testovacího prostředí 12
4.2 Výsledky měření 12

5 Uživatelská příručka **15**
5.1 Sestavení 15
5.2 Použití 15

6 Závěr **16**

1 Zadání

Program semestrální práce dostane, jako jeden z parametrů, zadaný souboru, přístupný pouze pro čtení. Bude ho interpretovat jako čísla v plovoucí čárce - 64-bitový double. Program najde číslo na arbitrárně zadaném percentilu, další z parametrů, a vypíše první a poslední pozici v souboru, na které se toto číslo nachází.

Program se bude spouštět následovně: `pprsolver.exe soubor percentil procesor`

- soubor – cesta k souboru, může být relativní k program.exe, ale i absolutní
- percentil – číslo 1 - 100
- procesor – řetězec určujíící, na jakém procesoru a jak výpočet proběhne
  - single – jednovláknový výpočet na CPU
  - SMP – vícevláknový výpočet na CPU
  - anebo název OpenCL zařízení – pozor, v systému může být několik OpenCL platforem

Součástí programu bude watchdog vlákno, které bude hlídat správnou funkci programu.

Testovaný soubor bude velký několik GB, ale paměť bude omezená na 250 MB. Zařídí validátor. Program musí skončit do 15 minut na iCore7 Skylake.

2 Analýza

2.1 Percentil

2.1.1 Vzorec

Definic percentilu existuje více, avšak z informací dostupných ze zadání a z přednášek předmětu KIV/PPR lze pozici percentilu definovat pomocí následujícího vzorečku:

$$position = floor((numbers - 1) \cdot (percentile/100)), \tag{1}$$

kde *numbers* je počet 64-bitových čísel v plovoucí čárce ve vstupním souboru, *percentile* hledaný percentil, $floor$ funkce, která převede dané číslo na nejbližší celé menší číslo a *position* pozice hledaného percentilu, uvažujeme li, že je soubor seřazený.

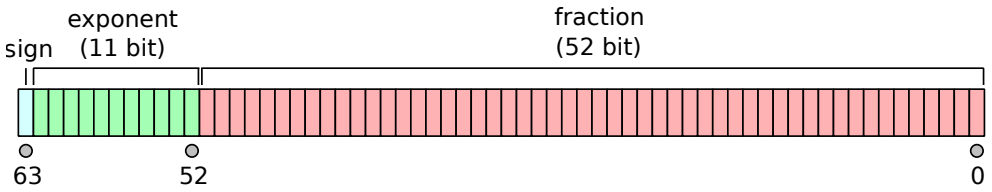
2.1.2 Vyhledání v souboru s omezenou operační pamětí

Ze zadání plyne, že paměť, kterou bude mít vytvořený program dostupný, bude omezena na 250MB, a tak je třeba algoritmus pro vyhledání percentilu tomuto omezení přizpůsobit. Naivní algoritmus pro vyhledání percentilu totiž předpokládá, že se celý soubor se vejde do paměti, kde bude seřazen, aby bylo možné zjistit hodnotu hledaného percentilu.

Jako nejjednodušší řešení toho problému se jeví rozdělení intervalu čísel, které se mohou v souboru nacházet, na podintervaly, a následně vytvořit histogram četností čísel v jednotlivých podintervalech. Počet položek v podintervalu, ve kterém se percentil nachází, by pak měl být dostatečně malý na to, aby se tyto položky vešly do operační paměti, a byly seřazeny podle hodnoty. Vyhledání percentilu v seřazené posloupnosti čísel je pak jednoduché, jelikož známe pozici percentilu. Celý soubor je nutné přečíst alespoň dvakrát.

Rozdělení intervalu na podintervaly obnáší definici postupu, jak jednotlivá čísla řadit do podintervalů a jak vůbec zvolit počet podintervalů. Počet podintervalů lze zvolit tak, aby se histogram vešel do paměti a počet položek v podintervalech nebyl příliš vysoký. Jednotlivá čísla pak lze vydělit počtem podintervalů, čímž získáme podinterval, do kterého dané číslo patří.

Předchozí postup se jeví jako validní, avšak lze provést optimalizaci výpočtu podintervalu tak, aby se namísto dělení používal bitový posun. Z definice 64-bitových čísel v plovoucí čárce (IEEE 754) lze totiž odvodit, že pro porovnávání dvou takových čísel stačí pouze jejich prvních 12 nejvýznamnějších bitů (znaménko a exponent).



Obrázek 1: Bitová reprezentace čísla v plovoucí řádce s dvojitou přesností dle IEE 754

Při ponechání prvních 12 bitů rozdělujeme interval čísel v plovoucí čárce na 2^{12} podintervalů. Samotný výpočet podintervalu, do kterého zpracovávané číslo patří, je tedy realizovaný pomocí bitového posunu o 52 bitů doprava.

Dalším důsledkem bitové reprezentace čísel typu `double` je to, že pokud budeme ponechávat více než prvních 12 bitů (tedy když budeme ponechávat i bity mantisy), porovnávání dvou takových čísel zůstane stále stejné. Pomocí počtu bitů, které budeme posouvat, lze tedy jednoduše tedy konfigurovat počet podintervalů.

V případě, že by se v nějakém podintervalu i tak vyskytovalo příliš mnoho unikátních čísel, a nebylo by možné ho dále zpracovávat, je možné ponechávat dalších n bitů následujících po bitech, které jsme ponechali při původním sestrojování histogramu, čímž podinterval rozdělíme na další podintervaly.

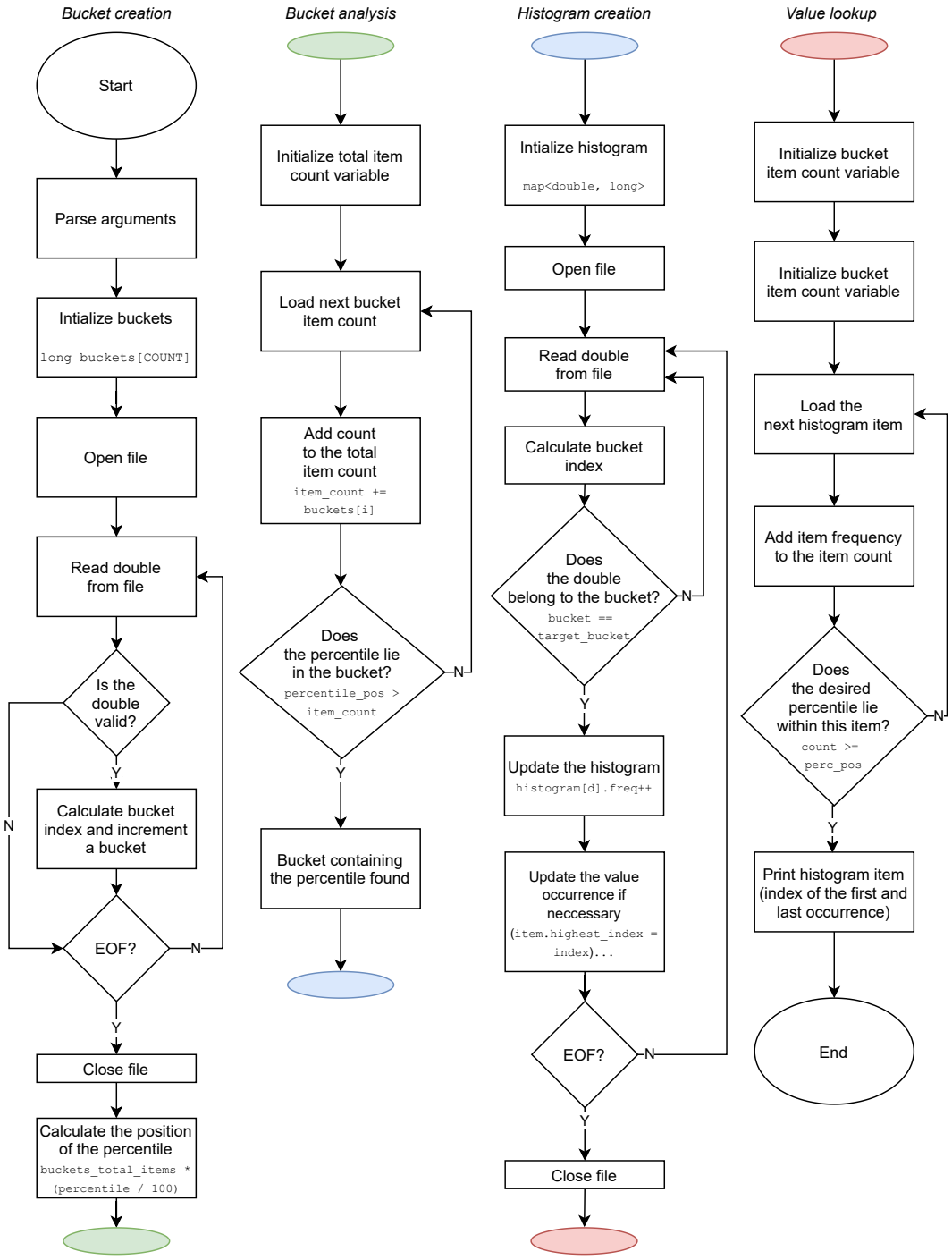
Rozdělování čísel do podintervalů pomocí bitového posunu poskytuje rychlý a jednoduchý způsob, jak rozdělovat čísla typu `double` do podintervalů, a tak efektivně sestavovat histogram i pro soubory, které se nevejdou do operační paměti.

2.2 Paralelizace vyhledání percentilu

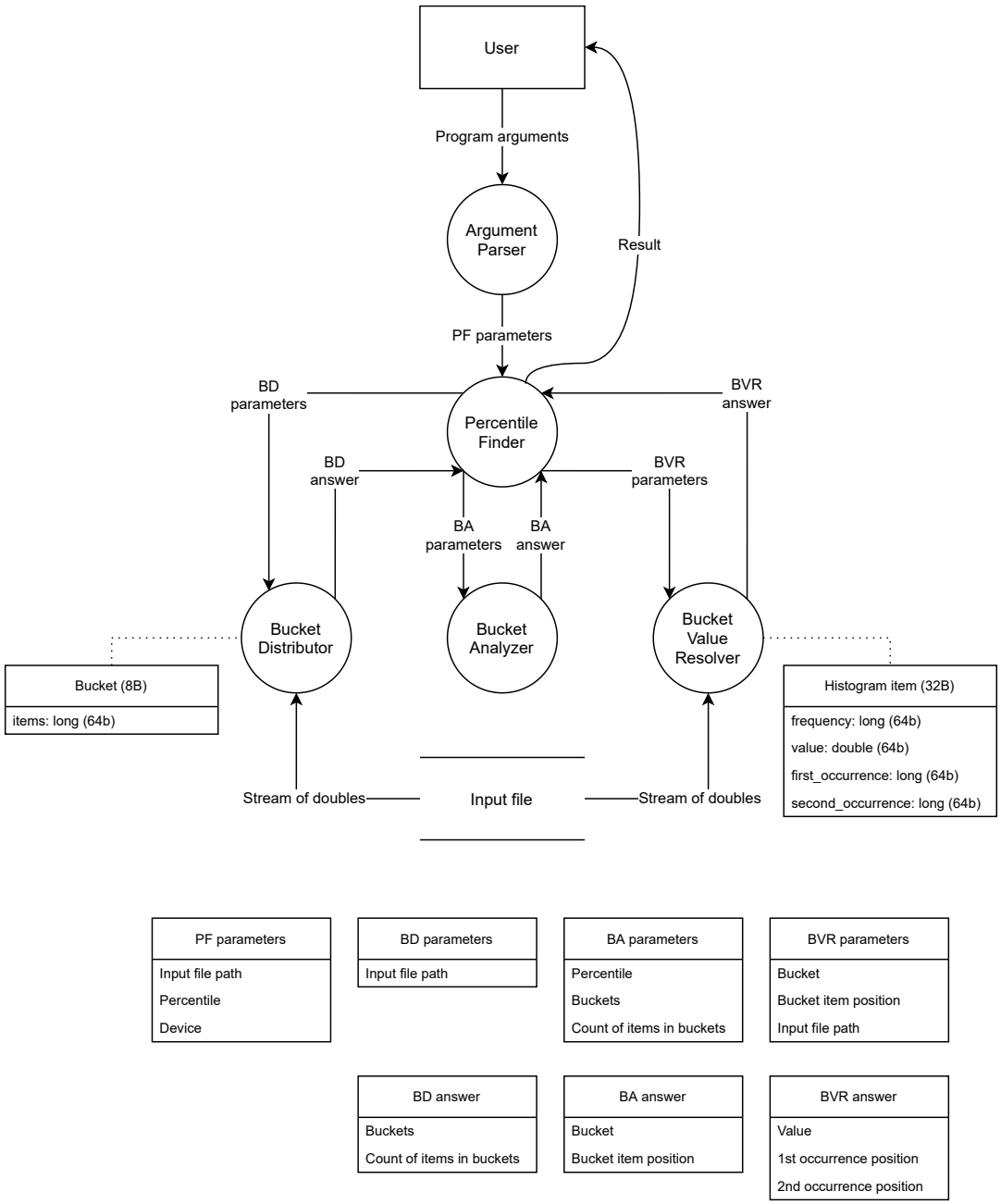
Jelikož postup pro vyhledání percentilu v souboru uvedený v sekci 2.1.2 je velmi jednoduchý a výpočetně nenáročný, tak se zdá, že hlavním faktorem, který bude ovlivňovat dobu vyhledávání, bude rychlost čtení ze souboru (tedy přístup k IO). Důsledkem toho je tedy efektivní paralelizace tohoto výpočtu složitá, jelikož čas strávený čekáním na IO je násobně větší než čas strávený sestavováním histogramu a jeho zpracováním za účelem vyhledání percentilu.

Jedním ze způsobů, jak provést paralelní vyhledávání percentilu, je takový způsob, kdy jedno vlákno čte ze souboru, a pomocí kruhového bufferu předává načtená data (čísla) dalším vláknům, která je zpracovávají. Výsledky výpočtů pak ukládají pomocí řízeného přístupu do chráněné paměti histogramu tak, aby nedošlo ke konfliktům. Další variantou tohoto způsobu, když zvolíme správnou velikost podintervalů, je způsob, kdy si každé vlákno se-

stavuje svůj vlastní histogram, a po přečtení celého souboru jsou histogramy vláken zkombinovány do jednoho hlavním vláknem, přičemž tato varianta je náročnější na velikost používaného paměťového prostoru výměnou za teoreticky rychlejší dobu vyhledávání z důvodu nepřítomnosti synchronizace při přístupů do sdílené paměti histogramu.



Obrázek 2: Diagram typu control-flow znázorňující zjednodušený postup používaný při vyhledávání percentilu v souboru s omezenou operační pamětí.



Obrázek 3: Diagram typu data-flow znázorňující tok dat v algoritmu.

3 Popis implementace

K vyhledávání percentilů v souborech byl zvolen postup popsaný v sekci 2.1.2 nakonfigurovaný tak, aby bylo během sestavování histogramu ponecháváno prvních 21 bitů čísla. Pokud by se v podintervalu, ve kterém se nachází percentil, nacházelo příliš mnoho unikátních čísel, tak by se znovu spustilo sestavování histogramu, aby se počet unikátních čísel redukoval. Podinterval, ve kterém se nachází percentil, by se dělil na další podintervaly tak, že by se po původních 21 bitech ponechávalo i dalších 21 bitů. V každém podintervalu by tak mohlo být maximálně 2^{22} unikátních čísel.

3.1 Sériová implementace

Sériová implementace, při které probíhá výpočet jen na jednom vlákně, je velmi jednoduchá, a nachází se ve zdrojovém souboru `serial_bucketing.cpp`.

Funkce `create_buckets_serial` sestavuje histogram z čísel nacházejících se ve vstupním souboru. Velikost histogramu je daná počtem bitů, které jsou z každého načteného čísla ponechávány.

Po sestavení histogramu se vypočítá podinterval, ve kterém se percentil nachází, a přesná pozice relativní vůči seřazeným číslům nacházejících se v souboru.

Poté se ve funkci `find_percentile_value_serial` znovu čte celý vstupní soubor, ale zpracovávají se pouze čísla, která patří do výsledného podintervalu.

Výskyt každého zpracovávaného čísla je zaznamenáván do dalšího histogramu (již implementován pomocí datové struktury tabulka). Dále se také pro každé číslo zaznamenává jeho první a poslední výskyt v souboru.

Pokud by během druhého čtení souboru bylo načteno příliš mnoho unikátních čísel, a histogram s konkrétními čísly by se nevešel do omezené paměti, přistoupí se k rozdělení podintervalu na další podintervaly tak, jak je to popsáno v sekci 2.1.2.

Nakonec, po zpracování celého souboru, je z histogramu, dle předtím vypočítané pozice, vybráno konkrétní číslo (včetně jeho výskytů), které je považováno za hledaný percentil.

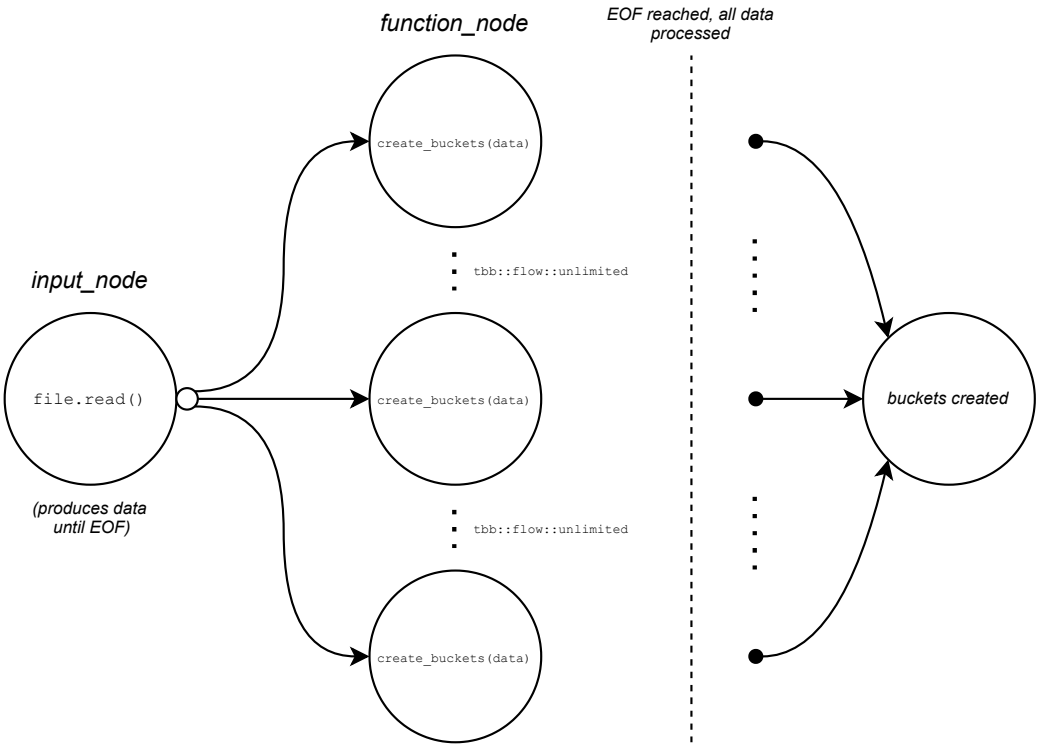
Díky výpočtu percentilu pomocí bitového posunu a následné aktualizaci histogramu pomocí $O(1)$ operace (zápis dle indexu do `std::vector`) je sériová implementace velmi rychlá.

3.2 Vícevláknová implementace

Tato implementace využívá konstrukce `flow_graph` z knihovny Intel Threading Building Blocks[1] k paralelizaci výpočtu histogramu vycházejícího z algoritmu popsaném v sekci 2.1.2.

První uzel grafu je typu `tbb::input_node` a čte data ze vstupního souboru, která rovnou předává následujícím uzlům typu `tbb::function_node` ke zpracování. Každý tento uzel, jakmile obdrží data, je postupně zpracovává, ignoruje čísla, která dle zadání nejsou považována za validní, vypočítává podinterval pro vyfiltrovaná čísla a aktualizuje histogram.

Každé vlákno, které zpracovává vstupní data, si vede vlastní histogram, aby nebylo třeba chránit přístup ke sdílené paměti. Po zpracování celého souboru hlavní vlákno sloučí histogramy jednotlivých vláken do jednoho výsledného.



Obrázek 4: Diagram popisující definici precedenčního grafu sestavení histogramu pomocí knihovny TBB

Po sestavení úvodního histogramu následuje sériový výpočet podinter-

[1] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html`

valu, ve kterém se nachází hledaný percentil, a přesná pozice relativní vůči seřazeným číslům nacházejících se v souboru.

Nakonec se znovu zahájí paralelní výpočet, velmi podobný výpočtu popsanému v předchozích odstavcích, avšak s tím rozdílem, že se zpracovávají pouze ta čísla ze vstupního souboru, která patří do podintervalu ve kterém se nachází hledaný percentil. Ke každému zpracovávanému číslu se vede záznam typu `bucket_item`, který obsahuje informaci o hodnotě čísla, počtu výskytů čísla, jeho prvním a posledním výskytu v souboru.

Záznamy typu `bucket_item` jsou jednotlivými uzly vkládány do sdílené tabulky, která je implementována pomocí typu `concurrent_hash_map`[2] z jmenného prostoru knihovny TBB. Tato tabulka, speciálně navržená pro využití v paralelních výpočtech, umožňuje bezpečný přístup k jejím hodnotám tak, aby nedošlo k nebezpečnému souběhu, kdy by k jedné hodnotě přistupovalo více vláken najednou. Každý dotaz do tabulky dle klíče vrací hodnotu typu `accessor`, která, dokud není smazána, zabraňuje ostatním vláknům k chráněné hodnotě přistupovat.

Po druhém zpracování souboru, kdy je již sestavený histogram s konkrétními čísly nacházejícími se v souboru, se obdobně, jako v sériovém výpočtu, vyhledá percentil včetně jeho prvního a posledního výskytu, který je považován za výsledek.

3.3 Implementace výpočtu s využitím OpenCL

Jak již bylo zmíněno v sekci 2.2, dosáhnout efektivní paralelizace navrženého výpočtu, která by byla výrazně rychlejší než sériové řešení, není jednoduché.

V rámci této semestrální práce však byl implementován postup, který využívá OpenCL k paralelizaci sestavování úvodního histogramu. Hlavní vlákno načítá bloky čísel ze souboru (větší bloky než u ostatních předtím popsaných postupů za účelem redukce počtu nutných volání OpenCL zařízení), které následně předává OpenCL zařízení, jež pro každé číslo zjistí, jestli se jedná o validní číslo dle zadání, pomocí bitového posunu spočítá do jakého podintervalu patří a aktualizuje histogram.

Hlavní vlákno nakonec načte z OpenCL zařízení sestavený histogram, zpracuje ho, a vypočte v jakém podintervalu se nachází hledaný percentil a jaká je jeho přesná pozice relativní vůči seřazeným číslům nacházejících se v souboru.

Poté hlavní vlákno začne vytvářet histogram již s konkrétními čísly. OpenCL zařízení se již nepoužívá, jelikož se jeho použití zde jeví jako neefektivní.

[2] `https://docs.oneapi.io/versions/latest/onetbb/tbb_userguide/concurrent_hash_map.html`

vstupní čísla	`double`	`double`	...	`double`
po zpracování	14573	14958	...	1209

Tabulka 1: Znázornění hromadného výpočtu podintervalů, do kterých zpracovávaná čísla patří.

Během vývoje byl implementován výpočet, který v této fázi využíval OpenCL zařízení pouze k výpočtu podintervalů, do kterých zpracovávaná čísla patří, a následně byly podintervaly zpracovány hlavním vláknem, jež sestavovalo histogram s konkrétními čísly. Tento výpočet byl však v porovnání se sériovým zpracování velmi pomalý, a tak nakonec nebyl použit.

Nakonec se po sestavení histogramu hlavním vláknem vyhledá v histogramu daný percentil dle pozice, která byla předtím vypočítána. Tento percentil je pak považován za výsledek.

3.4 Naivní implementace

Za zmínku také stojí naivní implementace, která implementuje triviální algoritmus bez jakýchkoliv omezení na operační paměť, aby bylo možné porovnat, zdali ostatní implementace poskytují správné výsledky.

Tato implementace se nachází v souboru `naive.cpp`.

4 Měření výkonnosti vytvořeného programu

4.1 Konfigurace testovacího prostředí

Všechny testy byly prováděny na počítači s následujícím hardware:

- **CPU** – AMD Ryzen 5 1600 @ 3.2GHz, 6C / 12T, 16MB L3 cache, 14nm
- **RAM** – Crucial Ballistix Sport LT, 16GB (2x8GB) DDR4 2400 CL16
- **GPU** – AMD RX480 @ 1208MHz, 4GB GDDR5 VRAM @ 7000 MHz
- **úložiště** – Samsung 970 EVO 1TB, PCIe 3.0 4x NVMe, TLC, 3400 MB/s read, write 2500 MB/s

Výše zmíněná grafická karta vystupuje je dostupná pro výpočet v OpenCL pod názvem *Ellesmere*.

Pro překlad zdrojových kódů byly použity překladače `g++` a `gcc` ve verzi 9.3.0 na operačním systému Ubuntu 20.04.3 LTS (Linux 5.4.0).

Překlad byl prováděn v režimu `release`.

Některá měření byla prováděna na souboru `ubuntu-21.04-desktop-amd64.iso` (sha1: `34409f0432bd67fc12ad7b4aed6348da77b292fb`) o velikosti 2.7GiB. Tento soubor je dostupný na oficiálních stránkách operačního systému Ubuntu.

Jako nástroj pro měření byl zvolen program `time` běžně dostupný v Ubuntu OS.

4.2 Výsledky měření

V rámci měření byl program spuštěn čtyřikrát pro každé dostupné zařízení na soubor obsahující obraz Ubuntu OS. Hledaný byl percentil 50. Časy běhů jsou zaznamenány v následující tabulce:

Zařízení	1.	2.	3.	4.	Průměr[s]
SINGLE	4.197	4.308	4.195	4.257	4.239
SMP	5.486	5.466	5.496	5.535	5.495
Ellesmere	3.294	3.069	3.374	3.105	3.211

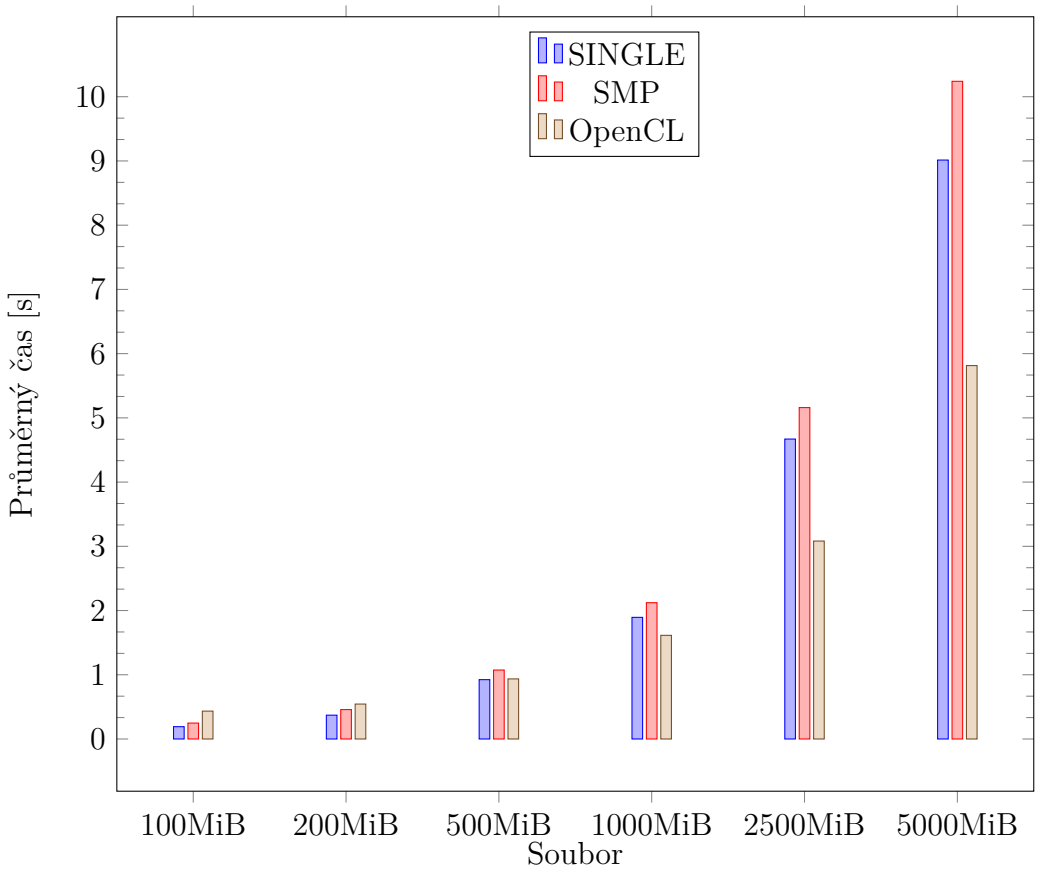
Tabulka 2: Časy běhů programu při zpracování souboru s obrazem Ubuntu OS.

Z naměřených dat lze spočítat pro oba paralelní výpočty urychlení oproti sériové verzi dle následujícího vzorce:

$$S = \frac{T_s}{T_p} \tag{2}$$

přičemž T_s je čas běhu sériového výpočtu a T_p čas běhu paralelního. Po dosazení naměřených hodnot do vzorce získáváme urychlení 0.771 pro paralelní výpočet pomocí zařízení SMP, zatímco pro paralelní výpočet pomocí zařízení OpenCL (*Ellesmere*) získáváme urychlení 1.32.

Časy běhů programů pro percentil 50 a soubory různých velikostí obsahující náhodná data jsou zaznamenány v následujícím grafu:



Obrázek 5: Graf zobrazující data naměřená při spouštění programu pro soubory o různých velikostech.

Při získávání průměrných časů běhu programu pro náhodné soubory (generovány čtením z `/dev/urandom`) byl program spuštěn vždy pětkrát pomocí speciálního bash skriptu:

```
(
    for file in 100MiB 200MiB 500MiB 1000MiB 2500MiB 5000MiB
    do
        echo $file:
        echo "---------------"
        for device in SINGLE SMP Ellesmere
        do
            echo $device:
            (/usr/bin/time -v ./pprsolver ../data/$file.bin 50
                $device) 2>&1 | grep -E "Maximum resident set
                size|Elapsed|e) page faults"
        done
    done
) | tee benchmark_result
```

Ukázka kódu 1: Bash skript pro spuštění vytvořeného programu pro různé soubory a pro všechna dostupná zařízení.

5 Uživatelská příručka

5.1 Sestavení

Program lze sestavit buď pomocí Microsoft Visual Studio IDE po nahrání hlavičkových a zdrojových souborů do řešení nebo pomocí použití nástroje CMake:

```
cd src
cmake .
cmake --build . --target pprsolver
```

Po použití nástroje CMake vznikne v pracovním adresáři spustitelný program `pprsolver`.

5.2 Použití

Program vyžaduje při použití následující argumenty:

1. cesta k souboru ke zpracování,
2. hledaný percentil (číslo z rozsahu 1 až 100 včetně),
3. a zařízení, které se má použít pro vyhledání percentilu (`SINGLE` – jednovláknový výpočet, `SMP` – vícevláknový výpočet a nebo název dostupného OpenCL zařízení).

Ukázka spuštění programu pro obraz Ubuntu OS a hledaný percentil 50 při použití jednovláknového výpočtu:

```
./pprsolver ubuntu-21.04-desktop-amd64.iso 50 SINGLE
```

6 Závěr

V rámci této semestrální práce byl vytvořen program napsaný v jazyce C++ 17, který umožňuje vyhledat percentil v libovolně velkém souboru i v případech, kdy je zpracovávaný soubor násobně větší než operační paměť dostupná programu. Před spuštěním programu lze vybrat zdali má být výpočet sériový či paralelní.

Všechny tři implementované výpočty vycházejí z navrženého algoritmu, přičemž došlo jen k minimálním úpravám algoritmu oproti jeho původnímu návrhu.

Paralelní výpočet je implementován pomocí knihovny Intel Threading Building Blocks, a využívá konstrukce `tbb::flow_graph`.

Druhý paralelní výpočet je implementován pomocí OpenCL, kdy je na OpenCL zařízení sestavován úvodní histogram.

Jelikož je navržený algoritmus pro vyhledávání percentilu v souboru velmi efektivní, tak jednotlivé paralelizace algoritmu nedosahují žádného nebo jen malého urychlení. Omezujícím faktorem pro urychlení pomocí paralelizace je také to, že velkou část času výpočtu se čeká na IO, kde nám paralelizace příliš nepomůže.

V případě paralelizace pomocí knihovny TBB byl tento výpočet rychlejší než jednovláknový výpočet pouze v případě, když nebyl program sestavován v režimu `release`. Hlavním důvodem, proč je tato paralelizace pomalá, je to, že v jednotlivých vláknech nedochází k náročným výpočtům. Dodatečná synchronizace mezi vlákny pak způsobuje, že oproti sériovému výpočtu dojde ke zpomalení.

Z výsledků měření lze pozorovat, že sestavování histogramu na OpenCL zařízení je dostatečně efektivní a přináší tak zrychlení oproti sériovému výpočtu. Jedním z důvodů viditelného zrychlení je fakt, že data ze vstupního souboru jsou v případě výpočtu pomocí OpenCL zařízení čtena po větších blocích, než u ostatních výpočtů, a tak je dobře využita paralelizace výpočtu pro jednotlivá čísla a redukuje se i počet volání čtení ze souboru.

Pokud bychom chtěli dosáhnout lepších výsledků paralelizace, bylo by třeba se pokusit navhrnout jiný algoritmus pro vyhledání percentilu, který by šel lépe paralelizovat. Osobně si však myslím, že navržený algoritmus, který čerpá hlavně z využití bitového posunu, je dostatečně dobrý.