Semestrální práce z KIV/OS

Vyhledávání percentilu v souboru

Přemysl Kouba
A21N0056P
kouba.prem@gmail.com

6. 12. 2021

Obsah

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 určující, 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 Úvod

Cílem tohoto dokumentu je seznámit s čtenářem s řešením problému hledání percentilu v souboru, které je zadání semestrální práce.

Program pro výpočet percentilu v souboru má omezené zdroje, tj. maximální velikost dostupné paměti a maximální dobu běhu výpočtu.

Požadavkem na program je, aby byl program schopen operovat třemi různými způsoby, tj. sériově, paralelně s použitím SMP a paralelně použitím knihovny OpenCL.

Je tedy nutné navrhnout a implementovat takový efektivní algoritmus, který při daném omezení vypočte percentil za dobu maximálně 15 minut na referenčním stroji.

3 Algoritmus

V algoritmu jsem použil následující definici percentilu:

$$i = \left\lceil \frac{P}{100} \cdot N \right\rceil \tag{3.1}$$

Kde N je seřazená množina hodnot. Zároveň platí, že percentil nabývá hodnot $(1 \leq P \leq 100)$

Pro nalezení percentilu je nutné mít seřazenou posloupnost. V případě neomezené kapacity paměti, lze všechny čísla načíst do paměti a následně je seřadit. Při vhodném algoritmu řazení a zároveň paralelizaci je rychlost výpočtu dostačující. To však nelze tvrdit o nárocích na paměť. Součástí práce je i tento zcela naivní algoritmus, který posloužil pro otestování přesnosti (a rychlosti) dalších algoritmů.

Pro ostatní povinné části práce byl použit následující algoritmus:

1. **vytvoř histogramu dle četností čísel v bucketech s využitím maskování čísel pomocí bitových operací**
2. najdi bucket ve kterém leží percentil
3. zkontroluj:
   - pokud již nejlze dále maskovat čísla => krok 5
   - pokud se počet čísel v bucketu nevejde do paměti pro řazení => krok 1
   - počet čísel v bucketu se vejde do paměti => krok 4
4. **seřaď čísla v bucketu a najdi jejich pozice (konec)**
5. **číslo již bylo nalezeno, najdi pozice (konec)**

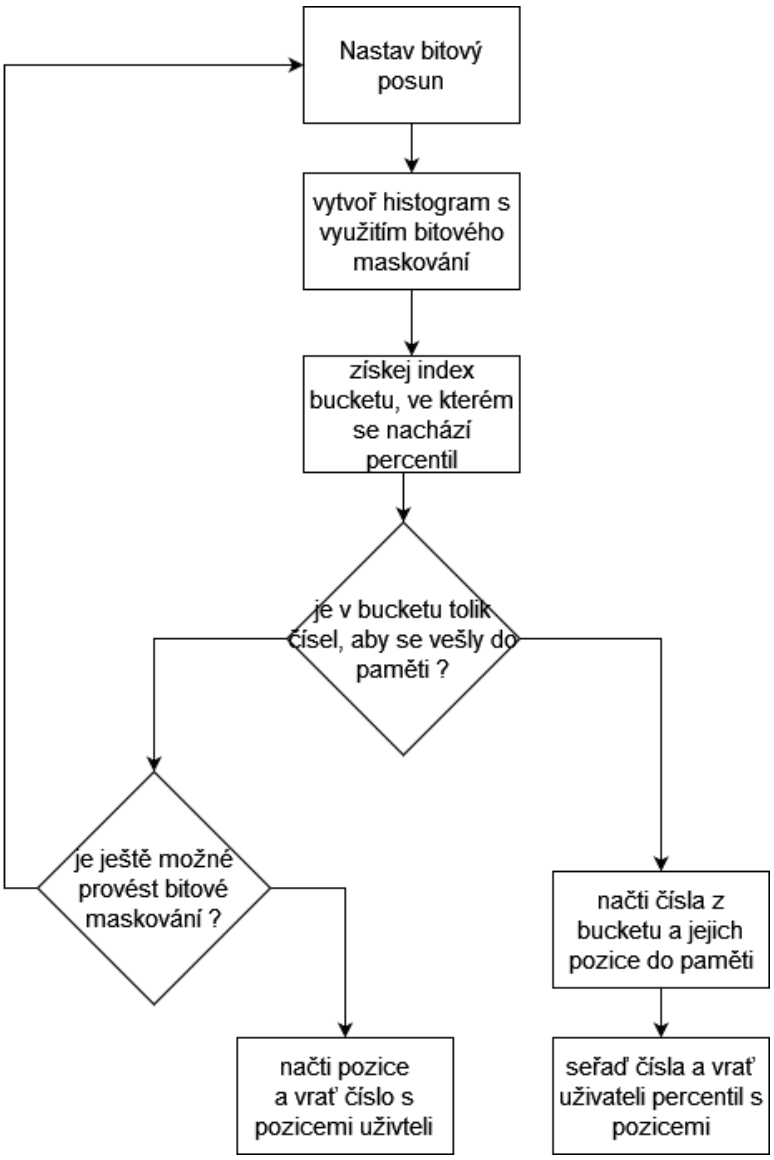
poznámka: tučně vyznačené kroky provádějí načítání souboru

Tento algoritmus má výhodu v tom, že je konečný. Počet průchodů záleží na tom, jaké jsou zvoleny bitové posuny. V semestrální práci jsou bitové posuny o 20, 21 a 23 bitů. Při použití větších čísel pro maskování práce nesplní nároky na operační paměť, protože veškerá paměť je spotřebována na tvorbu histogramu.

Výhodou algoritmu je, že s takto zvolenými hodnotami bitových posunů je garantováno, že algoritmus doběhne nejpozději po 4 a nejméně po 2 průchodech souborem:

- 3 průchody jsou maskování vstupních dat
- 1 průchod je vždy pro určení pozic čísla v souboru

Situace se čtyřmi průchody souborem může nastat v případě, že soubor obsahuje velké množství podobných, nebo hůře stejných čísel.

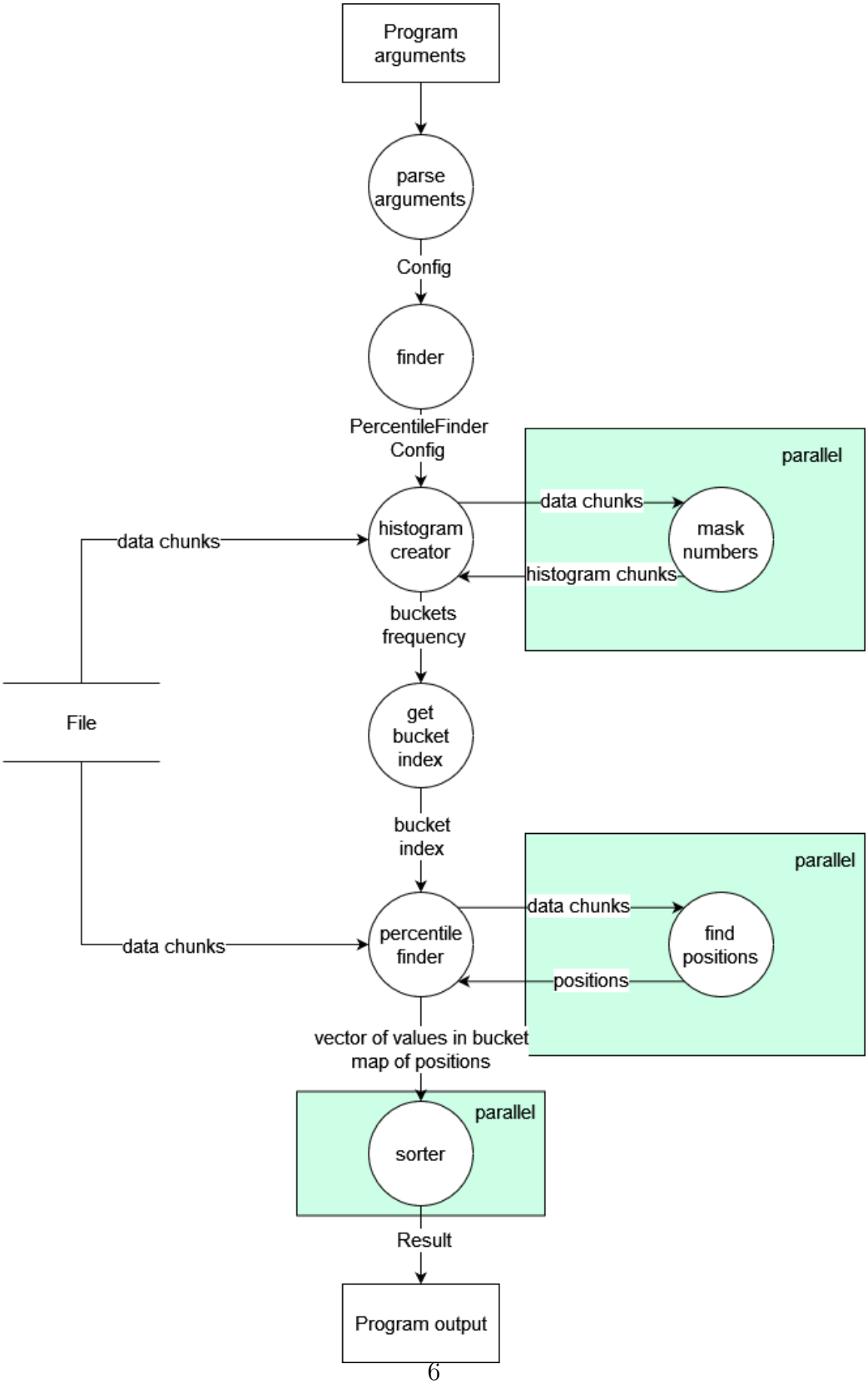


Obrázek 3.1: Algoritmus zjednodušeně

Načítání dat ze souboru je vždy v dávkách. Ty mají různou velikost v závislosti na zpracování (single/SMP/OpenCL).

Nevýhodou algoritmu dat je maskování dat. Pro záporná čísla je potřeba obrátit jejich řazení oproti jejich bitové reprezentaci tak, aby platilo, že v bucketu s indexem 0 budou zcela nejnižší čísla. Navíc je implementace vázaná na architekturu x86 (respektive x64) a umístění MSB. V případě opačné endianity nebude algoritmus fungovat. Maskování čísel je navíc výpočetně náročné. Pří dalších etapách je nutné nejen maskovat, ale také porovnávat s hranicemi intervalu.

Problém výpočetní náročnosti maskování lze odstranit vhodnou paralelizací úlohy.



Obrázek 3.2: Data flow programu

4 Implementace

Program je naimplementován v jazyce C++17. Pro paralelní verze jsou použity knihovny Intel TBB a OpenCL.

Funkce main nejprve provede zpracování vstupních argumentů a zkontroluje jejich správnost. Na základě daných argumentů vytvoří konkrétní řešení.

Tímto řešením je některý z PercentileFinderů, který má vždy funkci

```
find_percentile(ifstream* file, uint8_t percentile)
```

Vždy je vytvořena instance watchdoga a do příslušného finderu je předána jeho reference. Watchdog kontroluje zda-li program odpovídá (počítá) s periodou, která je definována v souboru *default_config.h*. K tomu, zda program odpovídá je použit čítač, který je inkrementován při činnosti. Při nečinnosti je spuštěna uživatelská funkce *timeout_callback*.
todoUdělám tam nějakou logiku ?

Bitové maskování čísel je obsaženo v souboru *number_masker*. Stejný kód využívá jak single řešení, tak řešení SMP. To platí i pro kód samotného algoritmu. Sdílený kód těchto dvou řešení je obsažen v souboru *resolver.cpp*

4.1 Naivní řešení

Naivní řešení je v souboru *resolver.cpp*. Toto řešení načte po částech soubor, vyčistí čísla a vše uloží do paměti. Následně seřadí vektor s čísly pomocí knihovní funkce *std::sort* s paralelizací.

4.2 Řešení single

je obsaženo v souboru *resolver_serial.cpp*. Toto řešení implementuje algoritmus z kapitoly **??**. Řešení lze rozdělit na dvě části - hledání bucketu a následné vyhodnocení.

Řešení pro single procesor je obsaženo v souborech:

- *resolver_single.h*
- *resolver_single.cpp*

Využívá také společné funkce *find_positions(), get_index_from_sorted_vector(), get_bucket_index() a find_result_last_stage()*, které jsou v souboru *resolver.cpp*.

4.3 Paralelizace a optimalizace

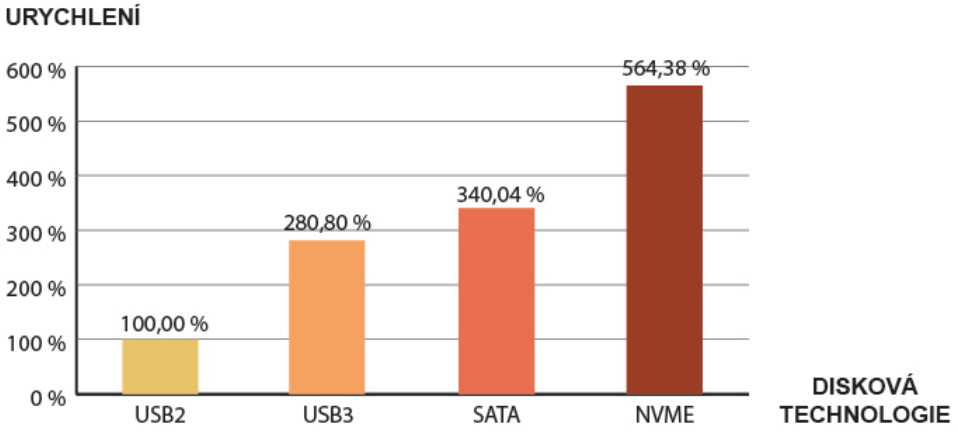
Problém hledání percentilu při omezení paměti vede k vícenásobnému čtení dat ze souboru. Hlavním urychlovačem programu je využití rychlejšího disku či omezení počtu průchodů.

4.3.1 Závislost na rychlosti disku

Vytvořil jsem srovnání časů běhu v závislosti na rychlosti disku na testovacím souboru. Testoval jsem řešení s použitím různých disků. Od flash disku (USB2 a USB3), přes klasický plotnový disk až k NVME disku.

Výsledky měření ukazují dvě věci:

- rychlost programu závisí převážně na rychlosti čtení z disku - ilustruje graf 4.1
- velikost urychlení závisí na rychlosti disku[1]



Obrázek 4.1: Graf urychlení běhu programu v závislosti na typu diskové technologie. Graf ilustruje urychlení sériové verze programu v závislosti na rychlosti disku.

[1]Ve chvíli, kdy je úzkým hrdlem programu čtení dat z disku, není možné dosáhnout velkého urychlení, protože výpočet je blokován.

Závislost velikosti urychlení na rychlosti disku

Z naměřených dat je také možné vypozorovat, že velikost urychlení výpočtu u paralelních verzi závisí na rychlosti dat. Čím větší je rychlost disku, tím větší je urychlení výpočtu. Výsledky měření jsou ilustrovány v tabulce na obrázku 4.2

Disková technologie	Typ zpracování					
	single		SMP		OpenCL	
	čas (ms)	zrychlení	čas (ms)	urychlení	čas (ms)	urychlení
USB2	124215	100%	110998	111,91%	117008	106,16%
USB3	44235	100%	33033	133,91%	40580	109,01%
SATA	36529	100%	25193	145,00%	31881	114,58%
NVME	22009	100%	14915	147,56%	17745	124,03%

Obrázek 4.2: Graf urychlení běhu programu v závislosti na typu diskové technologie. Graf ilustruje dosaženého urychlení různých verzí programu v závislosti na rychlosti disku.

Výsledky měření jsou dostupné v tabulce na adrese:

Naměřené hodnoty jsou uloženy v tabulce na následující adrese: `https://docs.google.com/spreadsheets/d/1n3qIzikCOOkA63JM7fygKWZ6Rta1cGRCIDmbWJhMUVw?usp=sharing`

4.3.2 Optimalizace

Kromě paralelizace, byly podniknuty i další kroky pro optimalizaci běhu programu:

- volba vhodného algoritmu
- alokace bufferu pro načítaná data v konstruktoru
- inicializace proměnných mimo cykly
- sloučení společné logiky do funkcí
  - zde není přínos výkonu, ale přínos snadné upravitelnosti a snazší čitelnost kódu

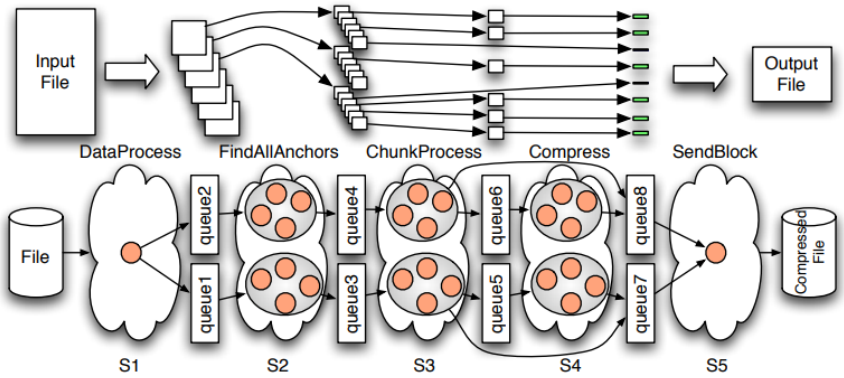
4.3.3 Paralelizace

Mým hlavním cílem pro paralelizaci je docílit pipeliningu – zpracování dat běží současně s jejich načítáním. Dalším kandidátem na paralelizaci je řazení prvků v bucketu při hledání hodnoty.

SMP

Pro paralelní verzi programu využívající SMP jsem nejprve chtěl použít bariéru z C++20, ale pak jsem zjistil, že cílový jazyk je C++17 a navíc jsem měl problém s použitím bariéry, který jsem nedovedl odstranit.

Následně jsem zvolil knihovnu Intel TBB konkrétně parallel pipeline. Funkčnost parallel pipeline ilustruje obrázek 4.3. Načítání dat ze souboru probíhá sériově, pak následuje paralelní maskování čísel ze souboru. Na konci této pipeline je složen histogram frekvencí výskytu v bucketech.



Obrázek 4.3: Graf urychlení běhu programu v závislosti na typu diskové technologie. Graf ilustruje urychlení sériové verze programu v závislosti na rychlosti disku.

Při hledání pozice prvku v souboru je opět využito TBB a parallel pipeline. Opět je zde sériové čtení z disku, následuje zpracování (hledání pozice). Na konci pipeline je složení mezivýsledků do jednoho vektoru, který je následně seřazen knihovní funkcí sort.

```
std::sort(std::execution::par_unseq,
          final_result.begin(), final_result.end());
```

Řešení pro SMP je obsaženo v souborech:

- *resolver_parallel.h*
- *resolver_parallel.cpp*

Využívá také společné funkce *find_positions()*, *get_index_from_sorted_vector()*, *find_result_last_stage()*, které jsou v souboru *resolver.cpp*.

OpenCL

Řešení využívá OpenCL jen poměrně málo. Jako nejvhodnější úlohu pro paralelizaci na akcelerátoru jsem vybral bitové maskování vstupu na indexy bucketů. Každé načtené číslo z paměti lze totiž maskovat zvlášť. To je ideální pro GPGPU. Řešení pro OpenCL je v souborech:

- *open_cl_default_header.h*
- *resolver_opencl.h*
- *resolver_opencl.cpp*
- *utils.cpp*

Funkce pro maskování čísel pro OpenCL je v souboru *open_cl_default_header.h*. Je napsána v jazyce OpenCL C. Skládá se z pár malých funkcí pro maskování čísel.

Přemýšlel jsem, jestli není vhodné také provést agregaci indexů bucketů v OpenCL. Nakonec jsem usoudil, že přínos není tak velký.

Řešení pro OpenCL dále využívá paralelizmu při řazení čísel

```
std::sort(std::execution::par_unseq,
          final_result.begin(), final_result.end());
```

Využívá také společné funkce *find_positions(), get_index_from_sorted_vector(), get_bucket_index(), find_result_last_stage()*, které jsou v souboru *resolver.cpp*.

5 Naměřené výsledky a testování

K měření hodnot a testování jsem využíval obraz distribuce debianu, který byl specifikován na portále. Soubor má velikost 3.67 GB.

HW prostředky stroje:

- AMD Ryzen 1600 (6jader, 12 vláken)
- 16GB RAM DDR4 s časováním CL17
- AMD RX 590, 8GB GDDR5
- Seagate FireCuda, 3,5"- 1TB SATA3.0
- NVME disk Samsung evo 970
- USB SanDIsk 3.2

Operační systém Windows 10

Měření rychlosti – spuštění programu v módu benchmark a následné zpracování výsledných časů. Mód benchmark postupně nalezne všechny percentily v souboru a změří čas jednotlivých typů zpracování. Program byl zkompilován se všemi optimalizacemi viz soubor *checker.ini*. Výsledky měření ilustruje obrázek 5.1 a 5.2

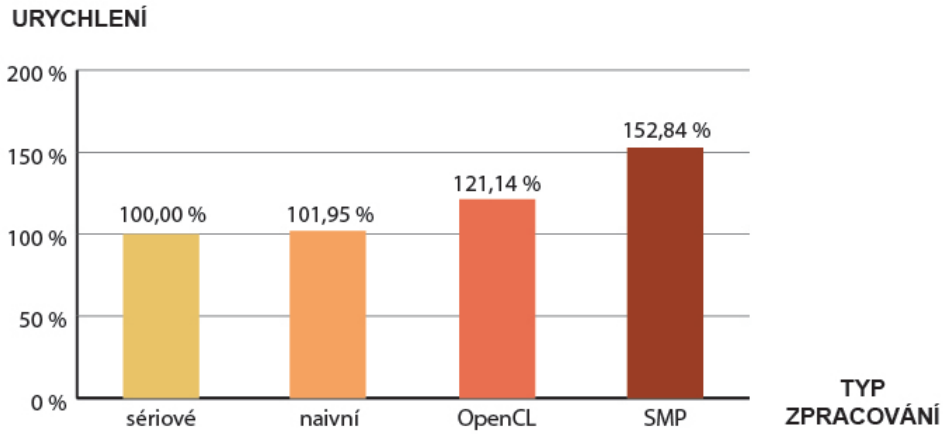
Typ zpracování	SUM (ms)	AVG (ms)	SPEEDUP%
naivní	2201016	22232,48485	101,95%
SMP	1468135	14829,64646	152,84%
sériové	2243848	22665,13131	100,00%
openCL	1852196	18709,05051	121,15%

Obrázek 5.1: Výsledky měření rychlosti programu v režimu benchmark[1]

Testování funkčnosti práce bylo ověřeno na třech testovacích souborech. První soubor je součástí repozitáře a obsahuje 16hodnot, které jsou známy a u kterých jsem si ručně vypočítal percentily tak, abych věděl, že alespoň základní funkčnost je správně.

Druhý testovací soubor byl použit při měření rychlosti jednotlivých implementací

Třetí testovací soubor má 655MB a obsahuje pouze dvě čísla (-10 a -9). Na tomto souboru jsem testoval scénář, kdy se musí namaskovat všechna



Obrázek 5.2: Graf urychlení výpočtu na základě typu zpracování

čísla (a program 4x prochází soubor), protože výsledek se nevejde do paměti. Cílem bylo otestovat, že aplikace nespadne a že se vejde do paměti. Zcela upřímně uznávám, že v algoritmu je chyba a nepodává pro tento soubor správné výsledky. Vycházel jsem z toho, že zadavatel předem slíbil, že tento krajní scénář nebude zkoušet.

V rámci práce jsem se snažil nejprve mít jednotkové testy, ale v C++ prostředí není jejich tvorba zcela intuitivní. Pár jich vzniklo, ale do konečné implementace se nedostaly - odstranil jsem je pro jejich neaktuálnost.

Naměřené hodnoty jsou uloženy v tabulce na následující adrese: `https://docs.google.com/spreadsheets/d/1n3qIzikCOOkA63JM7fygKWZ6Rta1cGRCIDmbWJhMUVw?usp=sharing`

Bylo dosaženo průměrného urychlení 152,84% pro paralelní zpracování. Při použití OpenCL bylo dosaženo průměrného urychlení 121,14% oproti sériovému zpracování. Výsledku bylo dosaženo podle testovacího scénáře definovaného v této kapitole. Proti zcela naivnímu algoritmu bylo dosažené urychlení ještě menší. Situaci ilustrují obrázky 5.1, 5.2.

6 Uživatelská příručka

Program má 3 povinné parametry, které jsou dle zadání:

- cesta k souboru (relativní nebo absolutní)
- percentil - číslo od 1 do 100
- způsob výpočtu - řetězec
  - „single“ - výpočet pro jedno jádro
  - „SMP“ - výpočet s použitím paralelizmu
  - „<název OpenCL zařízení>“ - paralelní výpočet s užitím OpenCL zařízení

Program dále umožňuje uživateli použít volitelný 4. parametr „-b“, který začne vykonávat benchmark všech implementovaných způsobů zpracování (i naivního !) a vypíše časy jednotlivých běhů na STDOUT. POZOR, tato možnost spotřebuje velké množství operační paměti - při načítání souboru o velikosti 3,8GB můŽe dosáhnout velikost alokované paměti až **10GB**. To je způsobeno naivním řešením, které všechny čísla načte do paměti a seřadí je.

Program také obsahuje watchdog vlákno. To má timeout 15vteřin a pokud program do této doby nezačne odpovídat, objeví se uživateli výzva, zda-li chce program ukončit, nebo chce počkat na odpověď programu - čeká na odpověď (znak) ve tvaru „Y/y“ nebo „N/n“.

7 Závěr

V rámci předmětu kiv/ppr vznikla tato práce a program, pro přečtení souboru a nalezení zadaného percentilu včetně pozice prvního a posledního výskytu.

Program nabízí uživateli naivní (paměťově neomezené), sériové a SMP řešení tohoto problému. Po Covidové pauze bylo dopracováno řešení OpenCL, které ovšem využívá GPGPU jen částečně. To je dáno především nevhodností úloh pro tyto akcelerátory.

Bylo dosaženo průměrného urychlení 152,84% pro paralelní zpracování. Při použití OpenCL bylo dosaženo průměrného urychlení 121,14% oproti sériovému zpracování. Výsledku bylo dosaženo podle testovacího scénáře definovaného v této kapitole. Proti zcela naivnímu algoritmu bylo dosažené urychlení ještě menší. Situaci ilustrují obrázky 5.1, 5.2.

V této práci nic neskončilo tak, jak jsem původně zamýšlel. Původně jsem chtěl vytvářet práci v SYCLu, ale zjistil jsem, že můj HW není podporovaný a že rozchodit toto prostředí by bylo na velmi dlouho. Následně jsem prezentoval svůj algoritmus tak, že rozdělím čísla do intevalů. Kolegové mě inspirovali abych použil bitové maskování, které je rychlejší. Následně jsem neodhadl časovou náročnost řešení problému a navíc jsem onemocněl.

Program splňuje požadavky, které jsou na něj kladeny. Nemyslím si ale, že by převyšoval svojí kvalitou a rychlostí práce ostatních kolegů. Kód je místy nepřehledný a řešení některých částí algoritmu je stylem pokus omyl.

Vliv na výsledek má nejen můj velký odklon od akademické sféry (studuji magisterské studium již 4 roky s přestávkou), ale také velká neznalost jazyka C++. Je sice příjemné, že jsem si v rámci této práce opět osvěžil jak vůbec tento jazyk vypadá, ale musím konstatovat, že jej vůbec neovládám a navíc shánění materiálů a knihoven mě velice iritovalo. Visual Studio naprosto tvrdošijně odmítalo sdělit proč nechce kód zbuildit. Následovala instalace programu CLion. Ten je zase založený na Javě a má velké nároky na paměť. Což je při omezené kapacitě mého notebooku problém.

Kladně naopak musím ohodnotit svojí snahu využít a oddělit společný kód do několika málo funkcí, které jsou volány napříč typy zpracování. Výsledkem je nejen lepší čitelnost kódu, ale také snazší údržba a případné hledání a opravování chyb.

8 Reference

V úplném závěru bych rád přiznal určitou mírou inspirace pracemi některých kolegů. Jejich práce jsou volně k nahlédnutí na githubu a není vždy nutné vymýšlet znovu kolo. Hlavním motivem byla občasná bezradnost ve volbě technologie - například při snaze o použití bariéry. Inspirace byla volná - zvážil jsem vždy kroky svých kolegů a zhodnotil, zda-li mají přínos pro moje řešení.

- https://gitlab.com/mkrysl/kiv-ppr-percentile.git
- https://github.com/MFori/PPR.git