Semestrální práce z KIV/OS

Vyhledávání percentilu v souboru

Přemysl Kouba
A21N0056P
kouba.prem@gmail.com

28. 11. 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.

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.

Doplnit nějakou ilustraci algoritmu

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.

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. 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. srovnání

Rychlost program přímo závisí na rychlosti disku. Od toho se odvíjí také následná paralelizace a optimalizace. Mým cílem při paralelizaci je docílit pipeliningu – zpracování dat běží současně s jejich načítáním. Dalším kandidátem na paralelizaci je řazení prvků v bucketu při hledání hodnoty.

Optimalizace proběhla při volbě algoritmu. Bitové maskování je úloha vhodná pro procesory. Realizace bitového maskování využívá bitových posunů.

4.3.1 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. Načítání dat ze souboru probíhá sériově, pak následuje paralelní maskování. Na konci této pipeline je složen histogram frekvencí výskytu v bucketech.

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());
```

4.3.2 OpenCL

OPENCL ještě nemám

5 Uživatelská příručka

6 Závěr

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

Tady je taky todo

Program nabízí uživateli naivní (paměťově neomezené), sériové a SMP řešení tohoto problému. Zatím není hotová implementace OpenCL z důvodu nákazy COVID-19. Snad bude co nejdříve hotová.

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.

Z důvodu časové tísně jsem nepokračoval v dalších optimalizacích svého kódu, ale jen v zajištění hlavní funkčnosti.