Algoritmy a jejich implementace

Mgr. Martin Mareš Ph.D.

12. března 2021

Obsah

1 Uvod, Cko

1.1 Uvod

- V tomto předmětu se zaměříme na PC hw architekturu.
- Cílovým OS je Linux, jelikož běží na většině super počítačů, klastrů apod. Ostatní OS stavěné na výkon se chovají podobně.
- Cko jako cílový jazyk (bohužel občas místo kompilátoru je komplikator). GCC jako cílový kompilátor.

Příklad 1.1. Motivační příklad: Transpozice matic. //TODO obrázek
a) Křivka naivního algoritmu roste (proč? na RAMu by neměla) + spiky.

- Na začátku skoky lze vysvětlit velkosti $L3$ cache, ale ne všude. Hlavně spacial lokality v hlavní paměti.
- skoku u 2^i protože cache není plně asociativní. Z množiny čísel se stejnými spodní bity, jenom málo se může dostat to cache na stejné místo. Podobně u dělitelů mocnin 2ky.

b) Je to naivní algoritmus + snažíme se vyhnout mocninám 2ky tak, že přidáváme konstantní mezery mezi řádky. Znázorňuje zelená křivka. Proč roste jako exp na nejmenších velikostech? Vysvětlíme později.
c) Tiling algoritmus
d) Transpose and swap (cache oblivious algoritmus). Chová se o něco lip než tiling.

Poznámka 1.2. Dotaz: má smysl při inicializaci algoritmu zjistit velikost cache? Obecná odpověď je spíše NE, ze 2 důvodů:

1. Reálná hierarchie může být extra složitá. Jenom velikost nepomůže.
2. U Transpose and Swap větší role hraje režie rekurze. Protože rekurze stoji víc než přistup do paměti. Takže se musíme zastavit na takové největší velikosti podproblému, který se vejde do cachi. Pokud víme, že $L_1 > 16kb$, tak funguje matice $64x64$.
3. Nastavení parametrů může škodit kompilaci. Jde ale obejit tak, že zkompilujeme několik verzi algoritmu s různé parametry, pak rychlé zjistit který je rychlejší a nadále používat.

   Např Linux kernel má k dispozici 4 algoritmy pro sw REID. Při bootu zkontroluje který je rychlejší a bude používat po celou dobu.

1.2 Cko

Tady budou pouze komentáře k slajdům Uvod do Cka.

Vývoj dialektů Cka

1. K&R Cko, původní návrh popsaný v knize.
2. První standardizace, ANSI a ISO. Je známá jako C89. Z ní se vytvořili dialekty dle překladače a taky C++.

3. C99 nejlepší featury z dialektů. Pak vznikla ještě C11 verze a C17 (druhá jen opravovala chyby první).
4. Občas Cko přebralo featury zpět od C++, např v C11 memory management pro paralelní programovaní.

Číselné typy

1. Není specifikováno, jestli **char** je *signed* nebo *unsigned*.
2. Typ **bool** máme protože kompilátor lepe optimalizuje. Taky lepší semantický význam. Jmenuje se ale **_Bool** aby kompilace nerozbila stávající programy, které již měli bool přes *typedef*. Pokud ale chci pravě **bool**, stačí použit $include < stdbool.h >$.
3. stejně jako v C++, existuji typy pevné délky, např u_int32. Hlavně se používají u síťových protokolu, taky inline assembler.

Union má pouze jednu ze dvou položek v pamětí, sám neví který to je. Je potřeba odvodit z kontextu.

Částečné typy

1. struct who_knowns_what *
2. int f() - nevíme parametry
3. int f(void) - nemá žádný parametr
4. int [] - pole neznámé velikosti

Reprezentace v pamětí

1. pozor na bitová pole při paralelním programovaní. (necelé bytů se hlavně používají hlavně pro síťové pakety)
2. pozor 2kový doplněk není zaručený.
3. C11 má operátor _AlignOf(type), nejčastěji velikost typu.
4. překladač nesmí prohodit pořadí položek uvnitř. Protože se zaručuje, že 2 struktury zděděné ze stejného typů mají stejný prefix v pamětí.

Literály

1. Pozor, přidaní nuly na začátek změní literál na 8 soustavu. Změní se i hodnota.
2. Veškeré celočíselné počítaní se provádí v typu int.
3. Floating-point literály bez ztráty přesností se zapisuji s exponentem, například 0x1.ffep10, kde p je oddělovač pro exponent.

Operátory

- používáme pointer->item, protože při zápisu *pointer.item tečka má větší prioritu, museli bychom psát (*pointer).item
- $a = x++$ pak a má původní hodnotu. $a = ++x$ pak a má hodnotu po inkrementu. $x = x++$ je hloupost, není definované.
- Hádanka $a+++++a =$??
- nepoužívat bitové operace u znaménkových typů
- !!x je přetypovaní na bool
- nemodulit záporným číslem
- float není lineárně uspořádané. $NaN \neq NaN$.
- počítaní s unsigned a signed dává unsigned
- nepsat $printf(”\%d\%d\%d”, a++, a++, a++$. Dobrým zvykem je měnit hodnotu proměnné pouze jednou v příkazu.

Nedefinované chovaní - cokoliv se může stát.

Pointry

- pointer na pole je něco jiného než pointer na první prvek. S pointrem na pole se ale běžně nepracuje.
- pointer na fce se zapisuje $int(*f)(inta)$;
- nepoužívat složitější konstrukce jako pointer na pointer poli pointrů.

Modifikátory

- u *const* záleží kde je. Protože $constint * x$ je konstantní integer, když $int * constx$ je const pointer.
- *register* int x - je historický, hint překladači, že může uložit do registru místo pamětí. Pak ale nejde použit pointer. Dneska se nepoužívá, překladač optimalizuje lepe.
- *volatile* (hist že se může nečekaně změnit). Třeba namapovaná periferie do pamětí. Třeba nějaký port cpe data, pokud nenapíšeme volatile, tak překladač může vyoptimalizovat 2 čtení na 1. Lze taky použit pro zrušení optimalizace pro měření.
- *restrict* - na adresu lze přistupovat pouze tímto pointrem nebo odvozeným. Pak překladač lepe optimalizuje. Např lze použit při kopirovaní pamětí, zaručí že 2 useky se nepřekrývají. Používá to funkce memcp nebo memmove.
- static int x - u globální jako private. Je viditelná uvnitř modulu.
- extern int x - společná proměnná pro několik modulu, stejný název.
- inline - místo volaní se vloží. Dneska překladač um sám.