Maskinnær Optimering

Agenda

Nogle teknikker/ideer til at gøre kode hurtigere, som kan siges at ligge i forlængelse af dette kursus.

Vi forbigår relevans ved skrivning af compilere, sorry

Definer hurtigere

Implementationer med samme asymptotiske kompleksitet men konstant faktor til forskel.

Eller implementationer som ikke skal skalere ud over en bestemt grænse.

F.eks. er insertion ofte sort hurtigere end quicksort for små datasæt

Regel 1: Profiler dit program

Find ud af, hvor dit program bruger tiden

Lad være med at gætte. Brug en "profiler" (linux: brug "perf")

Pas på du ikke mister overblikket og kaster dig for tidligt ud i småting

Regel 2: Forstå caching

Hvor stor er en cache blok? Kan du bruge den viden til noget?

Kan maskinen håndtere flere cache miss i parallel?

Forstå hvordan dit program interagerer med caching

- Hvor er der spatial lokalitet?
- Hvor er der temporal lokalitet?

Optimer for cache blok størrelse

Ofte bestemmes performance af hvor mange cache blokke du skal flytte.

Så kan du lige så godt fylde en cache block ud. Måske giver det færre blokke at flytte.

Optimer for linear access / Undgå pointer chasing

Hvis du kan, så undgå datastrukturer, hvor du skal følge mange pointere.

Placer hellere de data der skal tilgås ved siden af hinanden. Også selvom det nogen gange bliver mere bekosteligt at lave ændringer.

indsættelse i midten af et array kan se mere bekosteligt ud end indsættelse i et træ, men det er forbløffende billigt.

Hashing - fra buckets til open addressing

Mange hash tabeller håndterer hash kollisioner ved at indsætte elementer i en "spand" (bucket). Typisk er "spanden" en liste eller et træ.

I de senere år ser man hyppigere at man søger i flere nabo-pladser i tabellen først, nabo-pladser i samme cache blok eller i de efterfølgende blokke. På en måde rykker mindre "buckets" ind i selve tabellen.

Hvis det er nødvendigt med større buckets bruges gerne arrays frem for lister.

Regel 3: Kend til vektor instruktioner

Vektor instruktioner arbejder på (typisk små) arrays.

Kan f.eks. søge meget hurtigt blandt nabo elementer

Kan give et boost til f.eks. hash-maps, hvis buckets organiseres som arrays eller (delvis) placeres i selve hash-tabellen.

Regel 4: Fjern hop hvis de begrænser ILP

Hvis et hop er svært at forudsige kan det nogle gange betale sig at skrive dem om til data-flow format. Det kaldes undertiden if-conversion. F.eks.

```
int result;
if (search_for < node->key)
    result = node->before;
else
    result = node->after;
```

Kan i stedet skrives

```
int result = (search_for < node->key) ? node_before : node_after;
```

Og hvis man er heldig vil det give branch-fri kode. Her kan man dog være nødt til at tjekke hvad compileren gør, for den kan godt bruge en branch alligevel.

Det er en tricky teknik. Hvis hoppet i praksis er nemt forudsigeligt kan man risikere at denne omskrivning gør programmet langsommere i stedet for hurtigere

Eksempel: Hurtigere bisektion (børn i samme blok)

Hvis man har et binært træ hvor knuderne er relativt små kan det måske betale sig at placere børn og forælder i samme cache blok.

Før:

```
struct Node {
    int key;
    struct Node* left;
    struct Node* right;
}
```

Efter:

```
struct Node {
    int key;
    int left_key;
    int right_key;
    struct Node* grand_children
}
```

Og så

```
int index = (search_for > node->left_key) + (search_for > node->key)
                + (search_for > node->right_key)
struct Node* next = node->grand_children[index];
```

Kend til særlige instruktioner der kan have stor betydning

Eksempel: Sparsely populated arrays og pop_count instruktionen

Ofte kan man have brug for mindre arrays (f.eks. i et radix træ) hvor en stor del af elementerne er nul-elementer. Tag for eksempel et array som vi gerne vil indexere med index 0..63

Men vi vil gerne kun bruge lager på de elementer der er ikke-nul.

En mulig repræsentation kunne være en bitmaske allerførst, med en bit sat for hvert element der var tilstede.

Opslag kan da formuleres som:

```
uint64_t lower_bits = (1 << (index & 0x3F) - 1);
uint64_t present_entries = lower_bits & ptr->present_mask;
int real_index = pop_count(present_entries)
ptr = ptr->entries[real_index];
```

Opsamling

Kend dit program! Brug en profiler til at forstå hvor det bruger tiden.

Der kan være gevinst ved at "matche" ens datastrukturer til lagerhierarkiet.

Der kan være lidt specielle instruktioner, som de fleste maskiner har, som kan gøre en stor forskel. F.eks. pop_count. Eller vektor-instruktioner.

Det er sjældent nødvendigt eller ønskeligt at skrive assembler. De særlige instrukioner kan ofte nås via særlige "intrinsics" i f.eks. C eller C++ eller ved at vride koden, så compileren kan vektorisere vigtige løkker.

Der kan være gevinst ved at omskrive kode, så der er færre betingede hop til at begrænse ILP - men forsigtig! et korrekt forudsagt hop er billigt. Ikke alle hop er en begrænsning.

Spørgsmål og Svar