Caches and You

Florian “Florob” Zeitz

2018-03-12

Two hard problems in computer science

1.
2.

Two hard problems in computer science

1. Naming things
2.

Two hard problems in computer science

1. Naming things
2. Cache invalidation

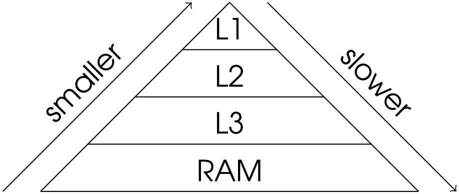
Two hard problems in computer science

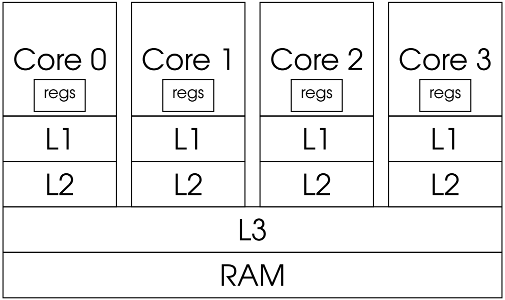
1. Naming things
2. Cache invalidation
3. Off by one errors

Introduction

- Problem:
  - RAM access is *very* slow
  - modern CPUs are *very* fast
  - a lot of time is spend waiting for memory
- Solution: cache memory in fast on-die memory
- usually two to three levels of cache

Cache levels





- L1 cache per core, separate caches for instructions and data
- L2 cache per core, shared for instructions and data
- L3 cache shared among cores, shared for instructions and data, doesn't always exist

Cacheline

- granularity of data transfered between memory and caches is fixed
- fetching data always fetches the whole cacheline
- invalidating data always invalidates the whole cacheline

Cache invalidation

- modern caches are usually coherent
- data is held consistent between per core caches
- writing a cacheline on one core invalidates it on all others
- inclusive caching: removing a cacheline from an outer cache level removes it from all inner caches

Intel Skylake characteristics

- L1 Data Cache: 32 KiB
- L1 Instruction Cache: 32 KiB
- L2 Cache: 256 KiB
- L3 Cache: 8 MiB
- Cacheline: 64 B

- L1 Data Cache Latency: ∼1 ns
- L2 Cache Latency: ∼3 ns
- L3 Cache Latency: ∼10 ns
- RAM Latency: ∼60 ns

Prefetching

- linear reads are detected
- forward reads fetch the following cacheline(s)
- backwards reads fetch the preceding cacheline(s)

Matrix multiplication


```
fn mul(a: &[[u32; DIM]], b: &[[u32; DIM]]) -> Vec<[u32; DIM]> {
    let mut result = vec![[0; DIM]; DIM];
    for (i, row) in result.iter_mut().enumerate() {
        for (j, cell) in row.iter_mut().enumerate() {
            for k in 0..DIM {
                *cell += a[i][k] * b[k][j];
            }
        }
    }
    result
}
```

- goes through `a` linearly in row major order
- goes through `b` with gaps in column major order

Matrix multiplication, B transposed

```rust
fn mul_t(a: &[[u32; DIM]], b: &[[u32; DIM]]) -> Vec<[u32; DIM]> {
    let mut result = vec![[0; DIM]; DIM];
    for (i, row) in result.iter_mut().enumerate() {
        for (j, cell) in row.iter_mut().enumerate() {
            for k in 0..DIM {
                *cell += a[i][k] * b[j][k];
            }
        }
    }
    result
}
```

- goes through `a` and `b` linearly in row major order
- approximately 9 times faster (`DIM = 1024` Intel Core i7-3720QM)

Matrix multiplication

- CPUs are very good at linear reads
- data storage is important
- avoid skipping large chunks of data
- note: `Vec<[u32; DIM]>` not `Vec<Vec<u32>>`

Working on live objects


```
struct Object {
    is_live: bool,
    id: u64,
    name: String,
    position: (f64, f64),
    velocity: (f64, f64),
}
```

```
fn main() {
    let objs = ...;

    for obj in &objects {
        if obj.is_live {
            do_work(obj)
        }
    }
}
```

- each objects is larger than a cacheline
- each check for liveness fetches a new cacheline
- linear traversal, but fetches a lot of unneeded data, if most objects are not live

Working on live objects

- data locality is important
- bad performance if loops act on one/few fields of an object
- alternative: use separate vectors for data often traversed
- alternative: convert Array of Structs (AoS) to Struct of Arrays (SoA) may be done as a compiler optimization (rarely)

Code size

- hot code should ideally fit into L1 cache
- calling a cached function *may* be better than running uncached straight-line code
- inlining blows up code size/duplicates code (`#[inline(never)]`)
- specialization creates multiple versions of code (consider passing `&Trait`)
- *always measure*

False sharing


```rust
fn increment(i: usize, j: usize) {
    let x: Arc<[AtomicUsize; 16]> = ...;

    let xp = x.clone();
    thread::spawn(move || {
        for _ in 0..100_000_000 {
            xp[i].fetch_add(1, Ordering::Relaxed);
        }
    });
    for _ in 0..100_000_000 {
        x[j].fetch_add(1, Ordering::Relaxed);
    }
}
```

- `increment(0, 1)` takes about 6 times longer than `increment(0, 7)`

False sharing

- Thread A: reads memory
- Thread B writes (near) it, causing cache misses in Thread A
- conditions:
  - values on same cacheline
  - access by different cores
  - frequent access
  - at least one writer

Vec is bae

- one contiguous (heap) allocation
- fast iteration
- often outperforms other data structures at their speciality, up to a certain size

LinkedList is meh

- one heap allocation per element
- expensive iteration (pointer chasing, many cache misses)
- insertion?
- `append()` (joining lists) is cheap
- `push_front()` and `push_back()` are cheap
- but if you need the latter two…

VecDeque

- growable ring buffer
- one contiguous (heap) allocation
- fast iteration
- fast `push_front()` and `push_back()`

HashMap

- linear probing and Robin Hood bucket stealing
- unlike in other languages fairly cache efficient
- values with same hash value are adjacent
- fast iteration

BTreeMap

- binary tree
- groups multiple items in a node
- better cache efficiency than a regular binary tree

Summary

- caches are good with temporal and spacial locality
- try to keep the working set small (instructions and memory)
- use `Vec`

Thank you for your attention. Any questions?

https://babelmonkeys.de/~florob/talks/RC-2018-03-12-caches-and-you.pdf