SPECTRAL:
TECHNOLOGIES

WEEK 9

Optimizing C++ programs

Plan

- Environment setup (Linux)
- C++ features and corners
- **Utilizing CPU and RAM properly**

Cache associativity

– Consider 2 following matrix multiplication functions:

```
template<size_t n>
void mul1(int a[n][n], int b[n][n], int c[n][n]) {
    for (size_t i = 0; i < n; ++i)
        for (size_t j = 0; j < n; ++j) {
            int sum = 0;
            for (size_t k = 0; k < n; ++k)
                sum += a[i][k] * b[k][j];
            c[i][j] = sum;
        }
}

template<size_t n>
void mul2(int a[n][n], int b[n][n], int c[n][n]) {
    Transpose(b);
    for (size_t i = 0; i < n; ++i)
        for (size_t j = 0; j < n; ++j) {
            int sum = 0;
            for (size_t k = 0; k < n; ++k)
                sum += a[i][k] * b[j][k];
            c[i][j] = sum;
        }
}
```

Cache associativity

- Lets run it (16MB 16-way associative L3 cache) and measure the time for each func/n

```
mul1/2048       60032 ms
mul1/2049       34925 ms
mul2/2048       2345 ms
mul2/2049       2501 ms
```

- mul2 is much faster, than mul1 due to sequential access to memory in the inner loop
- But why is mul1/2049 much faster, than mul1/2048?

Cache associativity

- Recall how address into cache translation works. When $n = 2048 = 2^{11}$, every address inside `b` array in the inner loop has the same lower 13 bits (since `k` jumps over $2^{11} \cdot 4$ bytes at each step)
- There are $\frac{2^{24}}{2^6 \cdot 16} = 2^{14}$ sets, so lower 6 bits of address are used for offset and next 14 bits are used for set index
- However we only use $2^{14-(13-6)} = 2^7$ sets instead of 2^{14}, effectively shrinking L3 cache size by the factor of 2^7
- Thats why powers of 2 aren't good for sizes and shifts. The same effect can be observed for binary search

Prefetching and speculative execution

- Recall binary search example from perf lecture

```
int Query(int x) const {
    auto it = std::lower_bound(data_.begin(), data_.end(), x);
    return it == data_.end() ? 0 : *it;
}
```

- We measured it on an array of size 10^8

Prefetching and speculative execution

– Lets implement it manually

```
int Query(int x) const {
   size_t l = 0, r = data_.size();
   while (l + 1 < r) {
       size_t m = (l + r) / 2;
       if (data_[m] < x) {
           l = m;
       } else {
           r = m;
       }
   }
   if (data_[0] >= x) { // its here to exclude premature exit
       return data_[0]; // for fair comparison with std::lower_bound
   }
   return r == data_.size() ? 0 : data_[r];
}
```

– This version runs **2x** times faster on clang-13

Prefetching and speculative execution

– Lets examine `std::lower_bound` (rewritten from iterators version for clarity)

```
int Query(int x) const {
    size_t l = 0, r = data_.size();
    size_t len = data_.size();
    while (len > 0) {
        auto half = len >> 1;
        auto middle = l + half;
        if (data_[middle] < x) {
            l = middle;
            ++l;
            len = len - half - 1;
        } else {
            len = half;
        }
    }
    return l == data_.size() ? 0 : data_[l];
}
```

Prefetching and speculative execution

– `clang-13` generates following assembly for this search (only for the actual version with iterators, not for the code from the previous slide)

```
.LBB0_9:
        mov     rsi, rcx
        mov     rdx, rbp
        shr     rdx
        mov     rdi, rdx
        not     rdi
        add     rdi, rbp
        cmp     dword ptr [rcx + 4*rdx], eax
        Lea     rcx, [rcx + 4*rdx + 4]
        cmovge  rcx, rsi
        cmovge  rdi, rdx
        mov     rbp, rdi
        test    rdi, rdi
        jg      .LBB0_9
```

– Lets compare it with the assembly from our version

Prefetching and speculative execution

```
.LBB0_8:
        mov     rcx, rsi
        lea     rsi, [rdx + 1]
        cmp     rsi, rcx
        jae     .LBB0_11
.LBB0_9:
        lea     rsi, [rdx + rcx]
        shr     rsi
        cmp     dword ptr [rbx + 4*rsi], eax
        jge     .LBB0_8
        mov     rdx, rsi
        lea     rsi, [rdx + 1]
        cmp     rsi, rcx
        jb      .LBB0_9
```

– `std::lower_bound` got compiled into branchless version with `cmov` instruction, our version got compiled into code with branching

Prefetching and speculative execution

- In the first version every iteration of binary search depends on the previous one
- In the second version cpu starts to execute next iteration speculatively based on the branch predictor's prediction
  - Namely, it starts to load data from memory (basically, prefetching) before next iteration starts
  - Even with poor branch predictor's performance the advantage of rare successful prefetch overweights the cost of branch misprediction
- `gcc` also generates `std::lower_bound` with branching, so both versions perform equally
- The first version might be faster on smaller arrays
- Sometimes profiling and asm inspection is needed to see the root cause of a slowdown

SPECTRAL:
TECHNOLOGIES

Perf annotate

– The same as `perf report`, but allows to profile on asm level:

```
> perf record ./bin
> perf annotate -Mintel
```

```
       120:┌─→mov      rcx,rsi
  4.69     │  lea      rsi,[rdx+0x1]
           │  cmp      rsi,rcx
  0.40     │↓ jae      144
       12c:│  lea      rsi,[rdx+rcx*1]
  0.40     │  shr      rsi,1
  0.90     │  cmp      DWORD PTR [rbx+rsi*4],eax
 86.30     └─ jge      120
  0.13       mov      rdx,rsi
  4.70       lea      rsi,[rdx+0x1]
             cmp      rsi,rcx
  0.13     ↑ jb       12c
       144:  mov      edx,DWORD PTR [rbx]
  0.92       cmp      edx,eax
  0.19     ↑ jge      e0
             xor      edx,edx
             cmp      rcx,0x5f5e100
           ↑ je       e0
             mov      edx,DWORD PTR [rbx+rcx*4]
           ↑ jmp      e0
       15a: → call     std::chrono::_V2::steady_clock::now@plt
```

Manual prefetching

- `__builtin_prefetch(addr)` can be used to prefetch data if you know, that `addr` is gonna be accessed soon
- Use it carefully, since it may actually make things worse

TLB misses

- In general using huge pages reduces the amount of TLB misses by a lot
- We can use *cache warmup* as it reduces the amount of misses for all caches
  - Simulate real workload when there is no such

SIMD

- x86 provides instructions to manipulate multiple values at once
  - vector instructions or simd (single instruction, multiple data)
- Starting with SSE (Streaming SIMD Extensions)
  - Expanded by SSE2, SSE3, SSE4
- Expanded later with AVX (Advanced Vector Extensions)
  - AVX2 and AVX-512

SSE

- SSE introduces instructions to manipulate 128-bit XMM registers
- Mostly fp calculations, but later versions added a lot of other instructions as well
  - e.g. SSE4.2 includes text processing instructions, CRC32 and etc
- GCC and Clang provide convinient data types and intrinsics to use
  - `#include <x86intrin.h>`
  - enable with `-march=...` or directly with `-msse4.2`
  - `__m128` for fp, `__m128i` for integers
  - List of instructions https://www.intel.com/content/www/us/en/docs/intrinsicsguide/index.html
- Data needs to be 16-byte aligned for faster execution

AVX

- Basically the same, but registers are 256 bit in AVX2
  - 512-bit in AVX-512
- `-mavx2`
- Note, that SIMD (especially AVX-512) may lead to lowering of cpu frequency due to higher power usage

Vectorization and loop unrolling

- Compilers try to vectorize code anyway
  - https://llvm.org/docs/Vectorizers.html
- To do this effectively they also do *loop unrolling*

```
int x;
for (x = 0; x < 100; x++) {
delete(x);
}
```

```
for (x = 0; x < 100; x += 5 ) {
    delete(x);
    delete(x + 1);
    delete(x + 2);
    delete(x + 3);
    delete(x + 4);
```

- Sometimes you need to do it by yourself to help the compiler
  - `#pragma unroll`
  - manually (handle corner cases, remove branching)

Vectorization

– -O3

```
int sum = 0;
for (int i = 0; i < size; ++i) {
    if (mask[i]) {
        sum += a[i];
    }
}
```

```
.LBB0_4:
        test    r8b, 1
        je      .LBB0_7
        cmp     byte ptr [rsi + rdx], 0
        je      .LBB0_7
        add     eax, dword ptr [rdi + 4*rdx]
.LBB0_7:
        ret
```

– vectorization happens with -mavx only

Vectorization

– vectorization with -O3

```
int sum = 0;
for (int i = 0; i < size; ++i) {
    sum += a[i] * mask[i];
}
```

Additional materials

- Array Layouts for Comparison-Based Searching https://arxiv.org/abs/1509.05053
- https://www.agner.org/optimize/
- https://en.algorithmica.org/hpc
- https://godbolt.org/