Writing high performance code

CS448h
Nov. 3, 2015
Overview

Is it slow? deciding when to optimize

Where is it slow? identifying bottlenecks

How slow is it? estimating potential

Why is it slow? reasons for bottlenecks

How can I make it faster? levers you have
Is it slow?
deciding when to optimize
Yes.*

* but you should only care if:
  it’s a critical path
  you have real-time constraints
  you will make new things feasible
  someone will pay you $$$
Where is it slow?
identifying bottlenecks
Tools for analysis

Timers

Profilers

Reading generated code (assembly)
Tool #1: Timers

```c
#include <sys/time.h>
static double CurrentTimeInSeconds() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}

#include <Windows.h>
double current_time() {
    LARGE_INTEGER freq, t;
    QueryPerformanceCounter(&t);
    QueryPerformanceFrequency(&freq);
    return t.QuadPart / freq.QuadPart;
}
```
SCIENCE!
(demo time)
Tool #2: Profiling - Performance Counters

Each core has performance counters for key events:
e.g. elapsed cycles, instructions executed, L1-cache hit, L2-cache miss, branch mis-predict

Can be configured to interrupt the CPU when the counter \( \geq \) threshold.

Used to *sample* the execution of the program

Very low overhead (often < 5%)

*perf-tools* on Linux, *Instruments* on Mac
Perf-Tools

Install (ubuntu)

$ apt-get install linux-tools-common linux-base

$ perf

perf_<version> not found

You may need to install linux-tools-<version>

$ apt-get install linux-tools-<version>

Print common counters

$ perf stat <your cmd>

Record profiling info

$ perf record -g <your cmd>

Display profiling results

$ perf report -M intel
$ perf stat my/binary --arg 1
Performance counter stats for ‘my/binary --arg 1’:

  82627.485530 task-clock #  7.858 CPUs utilized
   9,158 context-switches #  0.000 M/sec
    15 CPU-migrations #  0.000 M/sec
   11,847 page-faults #  0.000 M/sec
  305,027,996,924 cycles #  3.692 GHz
  321,979,156,613 instructions #  1.06 insns per cycle
   29,568,248,179 branches # 357.850 M/sec
  379,261,417 branch-misses #  1.28% of all branches
10.515014818 seconds time elapsed

$ perf stat -e r20D1 my/binary --arg 1
  725,894 r20D1
10.607431962 seconds time elapsed

Code for specific counter
(man perf list for details)
EXAMPLE

$ perf record -g ..../src/bin/pbrt --ncores 1 killeroo-simple.pbrt

$ perf report -M intel
EXAMPLE

0.00 :  4344e8:   lea   rcx,[rcx+rcx*2]
0.00 :  4344ec:   shl   rcx,0x2
0.12 :  4344f0:   mov   QWORD PTR [rsp+0x28],rcx
0.00 :  4344f5:   nop   DWORD PTR [rax]
0.60 :  4344f8:   mov   rcx,QWORD PTR [rsp+0x20]
0.84 :  4344fd:   mov   ebx,eax
0.18 :  4344ff:   mov   rsi,QWORD PTR [rsp+0x8]
0.78 :  434504:   shl   rbx,0x5
0.36 :  434508:   add   rbx,rdx
0.78 :  43450b:   movss  xmm1,DWORD PTR [rbx+rcx*1]
12.21 :  434510:   mov   rcx,QWORD PTR [rsp+0x28]
0.18 :  434515:   subss  xmm1,xmm5
4.15 :  434519:   movss  xmm3,DWORD PTR [rbx+rcx*1]
0.00 :  43451e:   mov   rcx,QWORD PTR [rsp+0x8]
0.54 :  434523:   mulss  xmm1,DWORD PTR [rsp+0x18]
6.98 :  434529:   lea   rcx,[rsi+rcx*2]
0.42 :  43452d:   movss  xmm2,DWORD PTR [rbx+rcx*4+0x4]
0.12 :  434533:   mov   ecx,0x1
0.48 :  434538:   sub   ecx,DWORD PTR [rsp+0x1c]
0.72 :  43453c:   movsx   rcx,ecx
0.36 :  43453f:   lea   rcx,[rcx+rcx*2]
0.18 :  434543:   movss  xmm0,DWORD PTR [rbx+rcx*4+0x4]
1.02 :  434549:   subss  xmm0,xmm4
2.04 :  43454d:   mulss  xmm0,DWORD PTR [rsp+0x14]
7.04 :  434553:   ucomiss  xmm1,xmm0
4.45 :  434556:   ja   434690 <BVHAccel::Intersect(Ray const&, Intersection*) const+0x270>
3.77 :  43455e:   sub   rbx,rdx
Instruments
Part of Xcode
Very easy to use
Pick a template, attach to a process
Profilers are cool, but don’t be seduced by tools
For most analysis, timers + printf are all you need, and give you total control over your experiments.
Tool #3: reading the assembly

Good idea: look at the assembly of whatever you’re timing to sanity check what it’s doing, spot anything suspicious.

x86 isn’t MIPS, but it’s *not that hard* to learn to skim.
READING X86 ASSEMBLY

Two syntaxes exist:

- **intel**: `movss xmm1,DWORD PTR [rbx+rcx*1]`
- **at&t**: `movss (%rbx,%rcx,1),%xmm1`

Intel’s manual uses Intel syntax (surprise!), Linux by default uses AT&T, but tools have options.

**Recommendation:** use **Intel syntax** documented by the manuals may be easier to read.
X86 REGISTERS

16 64-bit general purpose reg (for integers and pointers):

RAX  RBX  RCX  RDX  RSI  RDI  RBP  RSP <- stack pointer
R8   R9   R10  R11  R12  R13  R14  R15

16 General Purpose 128/256-bit SSE/AVX registers (for floating point and vectors):

XMM0/YMM0 through XMM15/YMM15

RIP <- instruction pointer, used to address constants

Smaller registers have different names (e.g. lower 32-bits of RAX is EAX)

There are other registers, we don’t need to talk about them
AN X86 INSTRUCTION

dest, src; 2-Op rather than 3-Op:

```
add rbx, rdx  \(rbx = rbx + rdx\)
```

most instructions can take a memory location as the source (but not the dest):

```
add rbx, QWORD PTR [rbx+rcx*4+0x4]
```

\(rbx = rbx + \) MEMORY\([rbx+rcx*4 + 0x4]\)

- size of load
- can omit any component
- 1, 2, or 4, 32-bit constant
STORES AND LOADS

Load Constant:
\[
\text{mov} \quad \text{ecx}, 0x1 \quad \text{ecx} = 0x1
\]

Store:
\[
\text{mov} \quad \text{QWORD PTR} \ [\text{rsp} + 0x20], \text{rsi} \\
\text{MEMORY} [\text{rsp} + 0x20] = \text{rsi}
\]

Load:
\[
\text{mov} \quad \text{rsi}, \text{QWORD PTR} \ [\text{rsp} + 0x20] \\
\text{rsi} = \text{MEMORY} [\text{rsp} + 0x20]
\]
BRANCHING

cmp rax, rbp //sets comparison flags
je 434610 //examines flags to decide

if(rax == rbp) goto address 434610

Knowing how instructions sets the flags is basically black magic (i.e. you look it up in the manual if you need to know)
How slow is it?
estimating peak performance potential
Before optimizing, we want to get an idea of how much faster we can make something. Is it even worth pursuing?

We have a lower bound: our current code

We want to estimate an optimistic upper bound.*

*many people don’t do this (even in academic papers!) and don’t know when to stop trying
Key concept: *bottlenecks*

Compute

FLOPS, IPC

Bandwidth

to memory, caches, of ports, …
UPPER BOUND ON PERFORMANCE

Based on fundamental limits in the hardware

Throughput of main memory

~25GB/s peak*

Throughput of instructions (instructions-per-clock)

~1-5 IPC / core, dependent on instruction mix*

*for an i7-3770K with a dual channel memory controller
## MEMORY LIMITS

<table>
<thead>
<tr>
<th>cache</th>
<th>layout</th>
<th>latency</th>
<th>measured bw</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 code</td>
<td>32 kB, 8 way, 64 B line size, per core</td>
<td>4 cycles</td>
<td>~100 GB/s</td>
</tr>
<tr>
<td>L1 data</td>
<td>32 kB, 8 way, 64 B line size, per core</td>
<td>4 cycles</td>
<td>~100 GB/s</td>
</tr>
<tr>
<td>L2</td>
<td>256 kB, 8 way, 64 B line size, per core</td>
<td>~12 cycles</td>
<td>~50 GB/s</td>
</tr>
<tr>
<td>L3</td>
<td>up to 16 MB, 12 way, 64 B line size, shared</td>
<td>~20 cycles</td>
<td>~33 GB/s</td>
</tr>
<tr>
<td>Main</td>
<td>hundreds of cycles</td>
<td></td>
<td>~20 GB/s</td>
</tr>
</tbody>
</table>

## OP LIMITS

<table>
<thead>
<tr>
<th>Port</th>
<th>type</th>
<th>op</th>
<th>max size</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>float</td>
<td>fp add</td>
<td>256</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>float</td>
<td>fp mul</td>
<td>256</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>float</td>
<td>fp div and sqrt</td>
<td>128</td>
<td>10-22</td>
</tr>
<tr>
<td>5</td>
<td>float</td>
<td>fp mov, shuffle</td>
<td>256</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>float</td>
<td>fp boolean</td>
<td>256</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>load</td>
<td>memory read</td>
<td>128</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>load</td>
<td>memory read</td>
<td>128</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td>memory write</td>
<td>128</td>
<td>*</td>
</tr>
</tbody>
</table>

**EXAMPLE: MEMCPY**

```c
void mymemcpy1(void *destv, void *srcv, int nbytes) {
    char *dest = (char*)destv;
    char *src = (char*)srcv;
    for(int i = 0; i < nbytes; i++)
        dest[i] = src[i];
}
```

Compute bound?

not clear how many instructions we need to execute...

Memory Bound?

the copy cannot be faster than the time it takes to read nbytes from memory and then write nbytes to memory

Memory bandwidth is:

- writing a byte requires reading the entire cache-line from main memory so it counts as 1 read + 1 write!
- $$\frac{2 \times \text{nbytes}}{\text{elapsed\_time}}$$
- $$\frac{3 \times \text{nbytes}}{\text{elapsed\_time}}$$
MEASURING OUR LOWER BOUND

__attribute__((__noinline__)) //force it not to inline so it can’t optimize away
void mymemcpy1(void * destv, void * srcv, int bytes) { ... }

int main()
{
    double * src = new double[N];
    double * dest = new double[N];
    for(int i = 0; i < N; i++) {
        src[i] = i;
        dest[i] = 0; //TOUCH ALL THE MEMORY BEFORE YOU START TIMING!!!!!
        // the OS will only actually give you the memory
        // when you write to it
    }

    double start = current_time();
    //sample multiple runs to cover timer inaccuracy/noise
    for (int i = 0; i < 10; i++) mymemcpy1(dest,src,N*sizeof(double));
    double end = current_time();
    printf("Throughput: %f GB/sec", (10*N/(1024*1024*1024))/(end-start));
}
MEMCPY 1 RESULT:

1.79 GB/s copied

5.37 GB/s effective memory bandwidth

Out of a possible 20GB/s!
Why is it slow?
reasons for bottlenecks
Is your code compute or memory bound?

**Experiment 1**
- Halve core compute (e.g., FLOPS), fix memory access
- Perf increases $\rightarrow$ probably compute bound

**Experiment 2**
- Halve memory access, keep compute the same
- Perf increases $\rightarrow$ probably memory bound
// Experiment 1: FLOPS

control1:
    dest[i] = sqrt(src[i]); // 0.22 GHz

experimental1:
    dest[i] = i % 2 == 0 ? src[i] // 0.43 GHz : sqrt(src[i])

// Experiment 2: Memory

control2:
    dest[i] = sqrt(src[i]) // 0.22 GHz

experimental2:
    dest[i/2] = sqrt(src[i/2]) // 0.22 GHz

// C code example:

void sqrtv(double *dest, double *src, int N) {
    for(int i = 0; i < N; i++) {
        dest[i] = sqrt(src[i]);
    }
}
TECHNIQUE 2: ESTIMATE

Recall mymemcpy1 results:

- 1.79 GB/s copied
- 5.37 GB/s effective memory bandwidth
- 20 GB/s possible bandwidth

It’s not memory bound, how many ops is it calculating? Let’s estimate it!
ESTIMATE OPS

(gdb) disas mymemcpy1(void*, void*, int)
Dump of assembler code for function _Z9mymemcpy1PvS_i:
 0x000000000000400720 <+0>:  test  edx,edx
 0x000000000000400722 <+2>:  jle  0x40073e <_Z9mymemcpy1PvS_i+30>
 0x000000000000400724 <+4>:  data32 data32 nop WORD PTR cs:[rax+rax*1+0x0]
 0x000000000000400730 <+16>:  mov  al,BYTE PTR [rsi]
 0x000000000000400732 <+18>:  mov  BYTE PTR [rdi],al
 0x000000000000400734 <+20>:  inc  rdi
 0x000000000000400737 <+23>:  inc  rsi
 0x00000000000040073a <+26>:  dec  edx
 0x00000000000040073c <+28>:  jne  0x400730 <_Z9mymemcpy1PvS_i+16>
 0x00000000000040073e <+30>:  ret
(gdb) disas mymemcpy1(void*, void*, int)

Dump of assembler code for function _Z9mymemcpy1PvS_i:

0x0000000000400720 <+0>: test edx,edx
0x0000000000400722 <+2>: jle 0x40073e <_Z9mymemcpy1PvS_i+30>
0x0000000000400724 <+4>: data32 data32 nop WORD PTR cs:[rax+rax*1+0x0]
0x0000000000400730 <+16>: mov al,BYTE PTR [rsi]
0x0000000000400732 <+18>: mov BYTE PTR [rdi],al
0x0000000000400734 <+20>: inc rdi
0x0000000000400737 <+23>: inc rsi
0x000000000040073a <+26>: dec edx
0x000000000040073c <+28>: jne 0x400730 <_Z9mymemcpy1PvS_i+16>
0x000000000040073e <+30>: ret

6 Ops/iteration * niterations / elapsed_time

10.77 effective GOPS

Note: it is much harder to estimate a realistic maximum IPC since it depends on how these instructions will get scheduled.
How can I make it faster?

levers you have for improving performance
Major levers for performance

Parallelism
  threads/cores, vectors/SIMD, ILP

Locality
  caches, registers, reuse

other*  
  * amount of work, “code quality”
TRICKS FOR COMPUTE-BOUND CODE

Vectorize! 4/8-wide vectors will give you 4/8x FLOPS

Parallelize! FLOPS scale linearly with \( n \) cores

- Threads don’t have to be scary!
- Use simple work queues, or just \#pragma omp parallel for

Intel and AMD ship faster versions of math functions (log, cos, exp)

- Up to 2x faster!
- Intel MKL (normally comes with ICC), AMD ACML
**COMPUTE BOUND: REDUCE THE OPS**

Process 2 bytes in parallel

```c
__attribute__((__noinline__))
void mymemcpy2(void * destv, void * srcv, int bytes) {
    short * dest = (short*) destv;
    short * src = (short*) srcv;
    bytes /= sizeof(short);
    for(int i = 0; i < bytes; i++)
        dest[i] = src[i];
}
```

Process 16 bytes in parallel

```c
__attribute__((__noinline__))
void mymemcpy5(void * destv, void * srcv, int bytes) {
    float * dest = (float*) destv;
    float * src = (float*) srcv;
    bytes /= sizeof(float);
    for(int i = 0; i < bytes; i += 4) {
        _mm_store_ps(&dest[i], _mm_load_ps(&src[i]));
    }
}
```
BANDWIDTH BOUND: REDUCE THE BANDWIDTH

```c
__attribute__((__noinline__))
void mymemcpy5(void * destv, void * srcv, int bytes) {
    float * dest = (float*) destv;
    float * src = (float*) srcv;
    bytes /= sizeof(float);
    for(int i = 0; i < bytes; i += 4) {
        // _mm_store_ps(&dest[i], _mm_load_ps(&src[i]));
        _mm_stream_ps(&dest[i], _mm_load_ps(&src[i]));
    }
}

_mm_stream_ps is a cache-bypassing write.
If you write the entire cache line together it can skip the read
making the total bandwidth 2.0*nbytes/elapsed_time
```
Standard cache-blocking techniques

Cache-bypassing writes (where appropriate)

Parallelize: each core has its own L1 and L2, which makes blocking techniques more effective. However, this will not scale main memory bandwidth (usually/much).
Putting it all together…

let’s optimize something real!
Summary

Is it slow? yes (but you might not care)
Where is it slow? timers + profiling (science!)
How slow is it? estimate peak potential
Why is it slow? experiment, estimate costs
How can I make it faster? parallelism, locality