What are you favorite low level code optimization tricks? [closed]

Asked 15 years, 10 months ago Modified 12 years, 11 months ago

Viewed 9k times

13

I know that you should only optimize things when it is deemed necessary. But, if it is deemed necessary, what are your favorite low level (as opposed to algorithmic level) optimization tricks.

For example: loop unrolling.

optimization

Share

Improve this question

Follow

asked Feb 27, 2009 at 2:56

Himadri Choudhury

10.3k ● 6 ● 41 ● 47

24 Answers

Sorted by: Highest score (default)

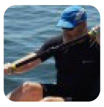
25

```
gcc -O2
```

Compilers do a lot better job of it than you can.

Share Improve this answer Follow

answered Feb 27, 2009 at 2:58



Paul Tomblin

183k ● 59 ● 323 ● 410

4 "...than you can." <-- general case. In some specific instances (such as algorithms, DSP, etc) a human can code a C routine that appears to be rather odd, but once compiled generates better assembly for the specific purpose than the compiler can. – Adam Davis Feb 27, 2009 at 3:21

Mainly due, though, to the fact that even great compiler optimizations only look at certain types of optimization and smaller sections of optimizable code. Once you understand the compiler and the assembly, then you can hand optimize much larger pieces of code that the compiler couldn't make better. – Adam Davis Feb 27, 2009 at 3:23

... but I'm splitting hairs - few people would ever need to do this. It's fun looking at how a compiler has turned a section of code into assembly - some of the compiler optimizations are actually quite intricate and odd until you really study it out. – Adam Davis Feb 27, 2009 at 3:27

5 "Compilers do a lot better job of it than you can" -- not if you're the compiler writer, and I thought that's who the question was addressing. "The last time I looked at the

assembler and used that to write better C" -- ur doin it rong. Look at the assembler and use it to write better assembler. – Windows programmer Feb 27, 2009 at 8:00

1 "Compilers do a lot better job of it than you can." I accept your challenge! – I. J. Kennedy Dec 18, 2010 at 17:06

20

Picking a power of two for filters, circular buffers, etc.

So very, very convenient.

-Adam

Share Improve this answer Follow

answered Feb 27, 2009 at 3:01

Adam Davis
93.5k ● 60 ● 271 ● 333

Can someone in short words explain what's the trick behind? You encounter it all the time, but I never figured that out... – daspostloch Jan 21, 2011 at 16:55

4 @daspostloch - The idea is that you often have to perform boundary checking and truncation on things that access the data. This means doing `if(input > MAX_SIZE) input = input - MAX_SIZE;` for instance. However, if the structure is a power of two in size, then both the check and the math can be done with one `AND` operation. For instance, if the data size is 128, then `input = input & 0x7F;` will truncate everything above 128, and otherwise leave `input` alone, meaning that the `if` statement earlier can be removed. – Adam Davis Jan 21, 2011 at 17:03

Why, bit twiddling hacks, of course!

12

Share Improve this answer

Follow

answered Feb 27, 2009 at 3:08

strager

89.9k ● 27 ● 138 ● 179

12

One of the most useful in scientific code is to replace `pow(x,4)` with `x*x*x*x`. Pow is almost always more expensive than multiplication. This is followed by

```
for(int i = 0; i < N; i++)
{
  z += x/y;
}
```

to

```
double denom = 1/y;
for(int i = 0; i < N; i++)
{
  z += x*denom;
}
```

But my favorite low level optimization is to figure out which calculations can be removed from a loop. Its always faster to do the calculation once rather than N times. Depending on your compiler, some of these may be automatically done for you.

Share Improve this answer

Follow

edited Feb 27, 2009 at 13:26

stroger

89.9k ● 27 ● 138 ● 179

answered Feb 27, 2009 at 3:25

Steve

12k ● 17 ● 71 ● 105

all slightly optimizing compilers perform at least dead code elimination and loop-invariant code motion (the one you describe here). still, i too tend to do it manually. especially if it makes the algorithm clearer. – Javier Feb 27, 2009 at 3:54

Believe it or not, I've seen actual performance gains when the denominator is slightly more complicated. Even with the Intel compiler. Also, I'm not talking about dead code, but code which doesn't need to be executed within the loop. – Steve Feb 27, 2009 at 4:11

I think modern compilers can easily do this sort of optimizations – user Aug 26, 2012 at 4:52

9

Inspect the compiler's output, then try to coerce it to do something faster.

Share Improve this answer

Follow

answered Feb 27, 2009 at 3:08

MSN

54.5k ● 7 ● 78 ● 107

yeah, there's nothing better to kill some slack time. just be careful to make source code more readable, not less. (adds some to the challenge :-) – Javier Feb 27, 2009 at 3:49

With today's processors, you can't tell what's faster just by looking at the compiler output. If you profile it different ways, you might be able to tell why one method is faster than another, but it might not apply to the next piece of code. – Mark Ransom Mar 4, 2009 at 20:50

You can if the compiler emits opcode patterns that are known to be slow. E.g., variable shifts on a PPC chip or any number of load-hit-stores. It's less helpful in the general case, but for hotspots it is definitely useful. – MSN Mar 4, 2009 at 21:45

9

I wouldn't necessarily call it a low level optimization, but I have saved orders of magnitude more cycles through judicious application of caching than I have through all my applications of low level tricks combined. Many of these methods are applications specific.

- Having an LRU cache of database queries (or any other IPC based request).
- Remembering the last failed database query and returning a failure if re-requested within a certain time frame.
- Remembering your location in a large data structure to ensure that if the next request is for the same node, the search is free.
- Caching calculation results to prevent duplicate work. In addition to more complex scenarios, this is often found in `if` or `for` statements.

CPUs and compilers are constantly changing. Whatever low level code trick that made sense 3 CPU chips ago with a different compiler may actually be slower on the current architecture and there may be a good chance that this trick may confuse whoever is maintaining this code in the future.

Share Improve this answer Follow

answered Feb 27, 2009 at 4:41
Jeffrey Cohen
341 ● 1 ● 3

In any non-trivial cache, you have to worry about cache management: size, staleness, invalidation, correctness. Which adds overhead, complexity, and a new source of bugs. Still, judicious caching is enormously effective.
– George V. Reilly Feb 27, 2009 at 8:00

7

`++i` can be faster than `i++`, because it avoids creating a temporary.

Whether this still holds for modern C/C++/Java/C# compilers, I don't know. It might well be different for user-defined types with overloaded operators, whereas in the case of simple integers it probably doesn't matter.

But I've come to like the syntax... it reads like "increment i" which is a sensible order.

Share Improve this answer Follow

edited Feb 27, 2009 at 3:04

answered Feb 27, 2009 at 2:59

Thomas
181k ● 55 ● 376 ● 501

4 most modern compilers won't create the temporary if it's only used as a statement, and not as an expression. – Javier Feb 27, 2009 at 3:40

7

Using template metaprogramming to calculate things at compile time instead of at run-time.

Share Improve this answer Follow

answered Feb 27, 2009 at 3:23

Brian R. Bondy
347k ● 126 ● 602 ● 640

that's what i like of scripting languages, you can do a lot of calculations at load time to make later runtime faster. granted, 'load time' is really just part of runtime, but still let's separate performance concerns. – Javier Feb 27, 2009 at 3:44

Never thought of this. Do you have any examples? – Himadri Choudhury Feb 28, 2009 at 1:16

5

Years ago with a not-so-smart compilier, I got great mileage from function inlining, walking pointers instead of indexing arrays, and iterating down to zero instead of up to a maximum.

When in doubt, a little knowledge of assembly will let you look at what the compiler is producing and attack the inefficient parts (in your source language, using structures friendlier to your compiler.)

Share Improve this answer Follow

answered Feb 27, 2009 at 3:52

Justin Love
4,447 ● 28 ● 37

With today's processors, looking at the assembly will only get you so far. You really need to time it. – Mark Ransom Feb 27, 2009 at 3:54

Good point. I was thinking of a case where the compiler used RAM when there were plenty of registers. I ended up rewriting that program in assembly, but embedded systems are kind of a different world. – Justin Love Feb 27, 2009 at 4:36

I agree. And I don't like a compiler that's too smart. I just want it to make pretty good ASM for me. – Mike Dunlavey Mar 10, 2009 at 12:26

5

precalculating values.

For instance, instead of sin(a) or cos(a), if your application doesn't necessarily need angles to be very precise, maybe you represent angles in 1/256 of a circle, and create arrays of floats sine[] and cosine[] precalculating the sin and cos of those angles.

And, if you need a vector at some angle of a given length frequently, you might precalculate all those sines and cosines already multiplied by that length.

Or, to put it more generally, trade memory for speed.

Or, even more generally, "All programming is an exercise in caching" -- Terje Mathisen

Some things are less obvious. For instance traversing a two dimensional array, you might do something like

```
    for (x=0;x<maxx;x++)
       for (y=0;y<maxy;y++)
          do_something(a[x,y]);
```

You might find the processor cache likes it better if you do:

```
   for (y=0;y<maxy;y++)
       for (x=0;x<maxx;x++)
           do_something(a[x,y]);
```

or vice versa.

Share Improve this answer Follow

answered Feb 27, 2009 at 5:05

smcameron

786 ● 4 ● 7 ● 8

5

Don't do loop unrolling. Don't do Duff's device. Make your loops as small as possible, anything else inhibits x86 performance and gcc optimizer performance.

Getting rid of branches can be useful, though - so getting rid of loops completely is good, and those branchless math tricks really do work. Beyond that, try never to go out of the L2 cache - this means a lot of precalculation/caching should also be avoided if it wastes cache space.

And, especially for x86, try to keep the number of variables in use at any one time down. It's hard to tell what compilers will do with that kind of thing, but usually having less loop iteration variables/array indexes will end up with better asm output.

Of course, this is for desktop CPUs; a slow CPU with fast memory access can precalculate a lot more, but in these days that might be an embedded system with little total memory anyway…

Share Improve this answer Follow

edited Feb 27, 2009 at 7:39

answered Feb 27, 2009 at 6:13

alex strange
1,249 ● 9 ● 9

4

I've found that changing from a pointer to indexed access may make a difference; the compiler has different instruction forms and register usages to choose from. Vice versa, too. This is extremely low-level and compiler dependent, though, and only good when you need that last few percent.

E.g.

```
for (i = 0;  i < n;  ++i)
    *p++ = ...; // some complicated expression
```

vs.

```
for (i = 0;  i < n;  ++i)
    p[i] = ...; // some complicated expression
```

Share Improve this answer Follow

answered Feb 27, 2009 at 3:40

Mark Ransom
308k ● 44 ● 416 ● 647

That one seems fairly obvious to me, two increments versus one. What about putting `p++` inside the `for()` and ditching `i` altogether? – user42092 Feb 27, 2009 at 3:44

You mean for (end = p+n; p != end; ++p) ? That might work. Might not though, because the compiler might optimize the 'i' loop. You really have to try these things and see which is fastest, because there are too many variables. – Mark Ransom Feb 27, 2009 at 4:02

4

Optimizing cache locality - for example when multiplying two matrices that don't fit into cache.

Share Improve this answer

answered Feb 27, 2009 at 13:42

Follow

sharptooth
170k ● 108 ● 535 ● 1k

3

Counting down a loop. It's cheaper to compare against 0 than N:

```
for (i = N; --i >= 0; ) ...
```

Shifting and masking by powers of two is cheaper than division and remainder, / and %

```
#define WORD_LOG 5
#define SIZE (1 << WORD_LOG)
#define MASK (SIZE - 1)

uint32_t bits[K]

void set_bit(unsigned i)
{
    bits[i >> WORD_LOG] |= (1 << (i & MASK))
}
```

Edit

```
(i >> WORD_LOG) == (i / SIZE) and
(i & MASK) == (i % SIZE)
```

because SIZE is 32 or 2^5.

Share Improve this answer

Follow

edited Feb 27, 2009 at 7:53

answered Feb 27, 2009 at 3:05

George V. Reilly
16.3k ● 7 ● 45 ● 39

1 Compilers are capable of converting a loop into the faster countdown form automatically, if the index variable isn't used in any expressions. – Mark Ransom Feb 27, 2009 at 3:32

i like downcounting; but mostly because it makes while(){..} loops nicer than for(;;){...} – Javier Feb 27, 2009 at 3:42

Compilers are quite smart nowadays about implementing divides by a constant using shifts & other tricks (see hexblog.com/2005/11/do_you_know_the_division_opera.html). But then, sometimes they aren't (hexblog.com/2005/12/the_longest_arithmetic_operati.html) – Michael Burr Feb 27, 2009 at 4:49

What's MASK for in set_bit()? bounds checking? – smcameron Feb 27, 2009 at 4:50

When the compiler can't prove that your divisor is a power of two, it has to fall back on the slower operations. You can help it along. Personally, I find the shifts and masks express my intent more clearly. – George V. Reilly Feb 27, 2009 at 7:56

3

Jon Bentley's Writing Efficient Programs is a great source of low- and high-level techniques -- if you can find a copy.

Share Improve this answer

Follow

answered Feb 27, 2009 at 8:05

George V. Reilly
16.3k ● 7 ● 45 ● 39

3

Eliminating branches (if/elses) by using boolean math:

```
if(x == 0)
    x = 5;

// becomes:

x += (x == 0) * 5;
// if '5' was a base 2 number, let's say 4:
x += (x == 0) << 2;

// divide by 2 if flag is set
sum >>= (blendMode == BLEND);
```

This REALLY speeds things out especially when those ifs are in a loop or somewhere that is being called a lot.

Share Improve this answer Follow

answered Feb 27, 2009 at 8:37

LiraNuna
67.1k ● 15 ● 121 ● 141

I doubt this would be an optimization at the assembly level. How would you describe the compare-and-multiply in x86 code? – strager Feb 27, 2009 at 13:30

Maybe this compiler is able to generate cmov only for the latter cases. – Joshua Feb 27, 2009 at 16:21

The one from Assembler:

3

```
xor ax, ax
```

instead of:

```
mov ax, 0
```

Classical optimization for program size and performance.

Share Improve this answer Follow

answered Feb 27, 2009 at 9:26

Anonymous
3,051 ● 4 ● 25 ● 25

What's the performance difference? – EmmanuelMess May 6, 2021 at 0:01

Allocating with new on a pre-allocated buffer using C++'s placement new.

2

Share Improve this answer Follow

answered Feb 27, 2009 at 3:15

Brian R. Bondy
347k ● 126 ● 602 ● 640

In SQL, if you only need to know whether any data exists or not, don't bother with `COUNT(*)`:

2

```
SELECT 1 FROM table WHERE some_primary_key =
some_value
```

If your `WHERE` clause is likely return multiple rows, add a `LIMIT 1` too.

(Remember that databases can't see what your code's doing with their results, so they can't optimise these things away on their own!)

Share Improve this answer Follow

answered Feb 27, 2009 at 3:39

user42092

Why not throw in the LIMIT 1 regardless? – strager Feb 27, 2009 at 13:28

Yes, always LIMIT 1 / TOP 1 here. Fast-first-row is what you want. – Joshua Feb 27, 2009 at 16:22

I took a look at Postgres' query plan with and without, and putting a limit on an already single-row select just seems to add overhead. Maybe it's different for other DB systems though. – user42092 Feb 27, 2009 at 17:48

If you're checking a primary key column, there's going to be an index on it which the optimizer will use, so adding the limit will not help you and may, as Ant P suggested, add overhead. If it's not a PK, then it should help.
– Graeme Perrow Mar 25, 2009 at 18:55

- Recycling the frame-pointer all of a sudden
- Pascal calling-convention

2

- Rewrite stack-frame tail call optimizarion (although it sometimes messes with the above)
- Using `vfork()` instead of `fork()` before `exec()`
- And one I am still looking for, an excuse to use: data driven code-generation at runtime

Share Improve this answer Follow

edited Jan 18, 2012 at 20:47
Niklas R
16.8k ● 30 ● 111 ● 211

answered Feb 27, 2009 at 3:07
Joshua
43.1k ● 9 ● 78 ● 148

modern fork() implementations use copy on write, which along with a few widely used hacks, make it just as fast as vfork(). in Linux, vfork() calls clone(), just like fork() – Javier Feb 27, 2009 at 3:48

vfork() will always be at least one pagefault faster, and see the flag on clone() that is CLONE_VFORK. The in parent process's memory space is still there. Try it with a volatile variable if you don't believe me. – Joshua Feb 27, 2009 at 4:00

+1 for code generation, if some of the data changes very seldom, it's a big win. Not only faster, but maybe surprisingly, simpler to write. – Mike Dunlavey Mar 10, 2009 at 12:30

Liberal use of __restrict to eliminate load-hit-store stalls.

1

Share Improve this answer

answered Feb 27, 2009 at 9:05

Follow

Crashworks
41.3k ● 14 ● 107 ● 171

1

Rolling up loops.

Seriously, the last time I needed to do anything like this was in a function that took 80% of the runtime, so it was worth trying to micro-optimize if I could get a noticeable performance increase.

The first thing I did was to roll up the loop. This gave me a very significant speed increase. I believe this was a matter of cache locality.

The next thing I did was add a layer of indirection, and put some more logic into the loop, which allowed me to only loop through the things I needed. This wasn't as much of a speed increase, but it was worth doing.

If you're going to micro-optimize, you need to have a reasonable idea of two things: the architecture you're actually using (which is vastly different from the systems I grew up with, at least for micro-optimization purposes), and what the compiler will do for you.

A lot of the traditional micro-optimizations trade space for time. Nowadays, using more space increases the chances of a cache miss, and there goes your performance. Moreover, a lot of them are now done by modern compilers, and typically better than you're likely to do them.

Currently, you should (a) profile to see if you need to micro-optimize, and then (b) try to trade computation for space, in the hope of keeping as much as possible in cache. Finally, run some tests, so you know if you've improved things or screwed them up. Modern compilers and chips are far too complex for you to keep a good mental model, and the only way you'll know if some optimization works or not is to test.

Share Improve this answer Follow

answered Feb 27, 2009 at 16:45

David Thornley
57k ● 9 ● 95 ● 158

1

In addition to Joshua's comment about code generation (a big win), and other good suggestions, ...

I'm not sure if you would call it "low-level", but (and this is downvote-bait) 1) stay away from using any more levels of abstraction than absolutely necessary, and 2) stay away from event-driven notification-style programming, if possible.

1. If a computer executing a program is like a car running a race, a method call is like a detour. That's not necessarily bad except there's a strong temptation to nest those things, because once you're written a method call, you tend to forget what that call could cost you.

2. If your're relying on events and notifications, it's because you have multiple data structures that need to be kept in agreement. This is costly, and should only be done if you can't avoid it.

In my experience, the biggest performance killers are too much data structure and too much abstraction.

Share Improve this answer Follow

answered Mar 10, 2009 at 12:42

Mike Dunlavey
40.6k ● 15 ● 94 ● 138

1

I was amazed at the speedup I got by replacing a for loop adding numbers together in structs:

```
const unsigned long SIZE = 100000000;

typedef struct {
    int a;
    int b;
    int result;
} addition;

addition *sum;

void start() {
    unsigned int byte_count = SIZE *
sizeof(addition);

    sum = malloc(byte_count);
    unsigned int i = 0;

    if (i < SIZE) {
        do {
            sum[i].a = i;
            sum[i].b = i;
            i++;
        } while (i < SIZE);
    }
}

void test_func() {
    unsigned int i = 0;

    if (i < SIZE) { // this is about 30% faster
than the more obvious for loop, even with O3
        do {
            addition *s1 = &sum[i];
            s1->result = s1->b + s1->a;
            i++;
        } while ( i<SIZE );
    }
```

Why doesn't gcc optimise for loops into this? Or is there something I missed? Some cache effect?

Share Improve this answer Follow

answered Mar 25, 2009 at 18:48

Mike A
1,303 ● 12 ● 14

1 What exactly do you mean by the "for loop" that you are comparing this to? – Himadri Choudhury Mar 25, 2009 at 19:16