

When is assembly faster than C?

[closed]

Asked 15 years, 10 months ago Modified 3 years, 6 months ago

Viewed 152k times

502

votes



Closed. This question needs to be more [focused](#). It is not currently accepting answers.

Closed 3 years ago.



Locked. This question and its answers are [locked](#) because the question is off-topic but has historical significance. It is not currently accepting new answers or interactions.

One of the stated reasons for knowing assembler is that, on occasion, it can be employed to write code that will be more performant than writing that code in a higher-level language, C in particular. However, I've also heard it stated many times that although that's not entirely false, the cases where assembler can **actually** be used to generate more performant code are both extremely rare and require expert knowledge of and experience with assembly.

This question doesn't even get into the fact that assembler instructions will be machine-specific and non-portable, or

any of the other aspects of assembler. There are plenty of good reasons for knowing assembly besides this one, of course, but this is meant to be a specific question soliciting examples and data, not an extended discourse on assembler versus higher-level languages.

Can anyone provide some **specific examples** of cases where assembly will be faster than well-written C code using a modern compiler, and can you support that claim with profiling evidence? I am pretty confident these cases exist, but I really want to know exactly how esoteric these cases are, since it seems to be a point of some contention.

c

performance

assembly

Share

edited Jan 3, 2018 at 15:58

community wiki

6 revs, 5 users 62%

Adam Bellaire

-
- 20 actually it is quite trivial to improve upon compiled code. Anyone with a solid knowledge of assembly language and C can see this by examining the code generated. Any easy one is the first performance cliff you fall off of when you run out of disposable registers in the compiled version. On average the compiler will do far better than a human for a large project, but it is not hard in a decent sized project to find performance

issues in the compiled code. – [old_timer](#) Aug 19, 2014 at 15:14

19 Actually, the short answer is: Assembler is **always** faster or equal to the speed of C. The reason is that you can have assembly without C, but you can't have C without assembly (in the binary form, which we in the old days called "machine code"). That said, the long answer is: C Compilers are pretty good at optimizing and "thinking" about things you don't usually think of, so it really depends on your skills, but normally you can always beat the C compiler; it's still only a software that can't think and get ideas. You can also write portable assembler if you use macros and you're patient.
– user1985657 Nov 9, 2014 at 21:30

13 I strongly disagree that answers to this question need to be "opinion based" - they can quite objective - it is not something like trying to compare performance of favorite pet languages, for which each will have strong points and draw backs. This is a matter of understanding how far compilers can take us, and from which point it is better to take over. – [jsbueno](#) May 15, 2015 at 15:29

30 Earlier in my career, I was writing a lot of C and mainframe assembler at a software company. One of my peers was what I'd call an "assembler purist" (everything had to be assembler), so I bet him I could write a given routine that ran faster in C than what he could write in assembler. I won. But to top it off, after I won, I told him I wanted a second bet - that I could write something faster in assembler than the C program that beat him on the prior wager. I won that too, proving that most of it comes down to the skill and ability of the programmer more than anythings else. – [Valerie R](#) May 22, 2017 at 23:56

1 @RobertF: We often leave off the "at what cost" part of these questions. I can write fast C or Assembler - sometimes the C is cheaper to write, and sometimes the assembler is cheaper to write. Speed often comes two ways: better algorithms or

low-level infrastructure exploitation -quicksort in C will typically be faster than bubble sort in assembler. But if you implement identical logic in both, usually assembler gives you ways to exploit the machine architecture better than the compiler can - the compiler is general purpose, and you're creating a specific adaptation for a single use case.

– [Valerie R](#) Dec 13, 2019 at 17:38

Comments disabled on deleted / locked posts / reviews |

40 Answers

Sorted by:

Highest score (default)



1

2

Next

283

votes



Here is a real world example: Fixed point multiplies on old compilers.

These don't only come handy on devices without floating point, they shine when it comes to precision as they give you 32 bits of precision with a predictable error (float only has 23 bit and it's harder to predict precision loss). i.e. uniform *absolute* precision over the entire range, instead of close-to-uniform *relative* precision (`float`).

Modern compilers optimize this fixed-point example nicely, so for more modern examples that still need compiler-specific code, see

- [Getting the high part of 64 bit integer multiplication](#): A portable version using `uint64_t` for 32x32 => 64-bit multiplies fails to optimize on a 64-bit CPU, so you

need intrinsics or `__int128` for efficient code on 64-bit systems.

- [umul128 on Windows 32 bits](#): MSVC doesn't always do a good job when multiplying 32-bit integers cast to 64, so intrinsics helped a lot.
-

C doesn't have a full-multiplication operator (2N-bit result from N-bit inputs). The usual way to express it in C is to cast the inputs to the wider type and hope the compiler recognizes that the upper bits of the inputs aren't interesting:

```
// on a 32-bit machine, int can hold 32-bit fixed-point
int inline FixedPointMul (int a, int b)
{
    long long a_long = a; // cast to 64 bit.

    long long product = a_long * b; // perform multiplication

    return (int) (product >> 16); // shift by the fixed
}
```

The problem with this code is that we do something that can't be directly expressed in the C-language. We want to multiply two 32 bit numbers and get a 64 bit result of which we return the middle 32 bit. However, in C this multiply does not exist. All you can do is to promote the integers to 64 bit and do a $64 \times 64 = 64$ multiply.

x86 (and ARM, MIPS and others) can however do the multiply in a single instruction. Some compilers used to ignore this fact and generate code that calls a runtime

library function to do the multiply. The shift by 16 is also often done by a library routine (also the x86 can do such shifts).

So we're left with one or two library calls just for a multiply. This has serious consequences. Not only is the shift slower, registers must be preserved across the function calls and it does not help inlining and code-unrolling either.

If you rewrite the same code in (inline) assembler you can gain a significant speed boost.

In addition to this: using ASM is not the best way to solve the problem. Most compilers allow you to use some assembler instructions in intrinsic form if you can't express them in C. The VS.NET2008 compiler for example exposes the 32*32=64 bit mul as `__emul` and the 64 bit shift as `__ll_rshift`.

Using intrinsics you can rewrite the function in a way that the C-compiler has a chance to understand what's going on. This allows the code to be inlined, register allocated, common subexpression elimination and constant propagation can be done as well. You'll get a *huge* performance improvement over the hand-written assembler code that way.

For reference: The end-result for the fixed-point mul for the VS.NET compiler is:

```
int inline FixedPointMul (int a, int b)
{
```

```
return (int) __ll_rshift(__emul(a,b),16);  
}
```

The performance difference of fixed point divides is even bigger. I had improvements up to factor 10 for division heavy fixed point code by writing a couple of asm-lines.

Using Visual C++ 2013 gives the same assembly code for both ways.

gcc4.1 from 2007 also optimizes the pure C version nicely. (The Godbolt compiler explorer doesn't have any earlier versions of gcc installed, but presumably even older GCC versions could do this without intrinsics.)

See source + asm for x86 (32-bit) and ARM on [the Godbolt compiler explorer](#). (Unfortunately it doesn't have any compilers old enough to produce bad code from the simple pure C version.)

Modern CPUs can do things C doesn't have operators for *at all*, like `popcnt` or bit-scan to find the first or last set bit. (POSIX has a `ffs()` function, but its semantics don't match x86 `bsf` / `bsr`. See https://en.wikipedia.org/wiki/Find_first_set).

Some compilers can sometimes recognize a loop that counts the number of set bits in an integer and compile it to a `popcnt` instruction (if enabled at compile time), but it's much more reliable to use `__builtin_popcnt` in GNU C,

or on x86 if you're only targeting hardware with SSE4.2:


```
_mm_popcnt_u32 from <immintrin.h> .
```

Or in C++, assign to a `std::bitset<32>` and use `.count()`. (This is a case where the language has found a way to portably expose an optimized implementation of popcount through the standard library, in a way that will always compile to something correct, and can take advantage of whatever the target supports.) See also https://en.wikipedia.org/wiki/Hamming_weight#Language_support.

Similarly, `ntohl` can compile to `bswap` (x86 32-bit byte swap for endian conversion) on some C implementations that have it.

Another major area for intrinsics or hand-written asm is manual vectorization with SIMD instructions. Compilers are not bad with simple loops like `dst[i] += src[i] * 10.0;`, but often do badly or don't auto-vectorize at all when things get more complicated. For example, you're unlikely to get anything like [How to implement atoi using SIMD?](#) generated automatically by the compiler from scalar code.

community wiki
6 revs, 6 users 61%
Nils Pipenbrinck

-
- 6 How about things like `{x=c%d; y=c/d;}`, are compilers clever enough to make that a single `div` or `idiv`? – [Jens Björnhager](#) May 30, 2010 at 2:06
-
- 6 Actually, a good compiler would produce the optimal code from the first function. Obscuring the source code with intrinsics or inline assembly *with absolutely no benefit* is not the best thing to do. – [slacker](#) Jul 23, 2010 at 0:20
-
- 67 Hi Slacker, I think you've never had to work on time-critical code before... inline assembly can make a *huge difference. Also for the compiler an intrinsic is the same as normal arithmetic in C. That's the point in intrinsics. They let you use a architecture feature without having to deal with the drawbacks. – [Nils Pipenbrinck](#) Jul 23, 2010 at 18:46
-
- 6 @slacker Actually, the code here is quite readable: the inline code does one unique operation, which is immediately understandable reading the method signature. The code lost only slowly in readability when an obscure instruction is used. What matters here is we have a method which does only one clearly identifiable operation, and that's really the best way to produce readable code these atomic functions. By the way, this is not so obscure a small comment like `/* (a * b) >> 16 */` can't immediately explain it. – [Dereckson](#) Dec 22, 2013 at 23:04 
-
- 6 To be fair, this example is a poor one, at least today. C compilers have long been able to do a 32x32 -> 64 multiply even if the language doesn't offer it directly: they recognize

that when you cast 32-bit arguments to 64-bit and then multiply them, it doesn't need to do a full 64-bit multiply, but that a 32x32 -> 64 will do just fine. I checked and all of [clang, gcc and MSVC in their current version get this right](#). This isn't new - I remember looking at compiler output and noticing this a decade ago. – [BeeOnRope](#) May 27, 2018 at 3:58

156

votes



Many years ago I was teaching someone to program in C. The exercise was to rotate a graphic through 90 degrees. He came back with a solution that took several minutes to complete, mainly because he was using multiplies and divides etc.

I showed him how to recast the problem using bit shifts, and the time to process came down to about 30 seconds on the non-optimizing compiler he had.

I had just got an optimizing compiler and the same code rotated the graphic in < 5 seconds. I looked at the assembly code that the compiler was generating, and from what I saw decided there and then that my days of writing assembler were over.

Share

[edited Jul 25, 2018 at 14:19](#)

community wiki

[4 revs, 4 users 56%](#)

[Peter Cordes](#)

-
- 3 Just wondering: Was the graphic in 1 bit per pixel format?
– [Nils Pipenbrinck](#) Feb 23, 2009 at 16:22
-
- 4 Yes it was a one bit monochrome system, specifically it was the monochrome image blocks on an Atari ST. – [lilburne](#) Feb 24, 2009 at 10:10
-
- 21 Did the optimizing compiler compile the original program or your version? – [Thorbjørn Ravn Andersen](#) Dec 21, 2014 at 10:55
-
- 1 On what processor? On 8086, I'd expect that optimal code for an 8x8 rotate would load DI with 16 bits of data using SI, repeat `add di,di / adc al,al / add di,di / adc ah,ah` etc. for all eight 8-bit registers, then do all 8 registers again, and then repeat the whole procedure three more times, and finally save four words in ax/bx/cx/dx. No way an assembler is going to come close to that. – [supercat](#) Jan 3, 2018 at 4:24
-
- 2 I really can't think of any platform where a compiler would be likely to get within a factor or two of optimal code for an 8x8 rotate. – [supercat](#) Jan 3, 2018 at 4:31
-

66
votes



Pretty much anytime the compiler sees floating point code, a hand written version will be quicker if you're using an old bad compiler. **(2019 update: This is not true in general for modern compilers.** Especially when compiling for anything other than x87; compilers have an easier time with SSE2 or AVX for scalar math, or any non-x86 with a flat FP register set, unlike x87's register stack.)

The primary reason is that the compiler can't perform any robust optimisations. [See this article from MSDN](#) for a

discussion on the subject. Here's an example where the assembly version is twice the speed as the C version (compiled with VS2K5):

```
#include "stdafx.h"
#include <windows.h>

float KahanSum(const float *data, int n)
{
    float sum = 0.0f, C = 0.0f, Y, T;

    for (int i = 0 ; i < n ; ++i) {
        Y = *data++ - C;
        T = sum + Y;
        C = T - sum - Y;
        sum = T;
    }

    return sum;
}

float AsmSum(const float *data, int n)
{
    float result = 0.0f;

    __asm
    {
        mov esi,data
        mov ecx,n
        fldz
        fldz
l1:
        fsubr [esi]
        add esi,4
        fld st(0)
        fadd st(0),st(2)
        fld st(0)
        fsub st(0),st(3)
        fsub st(0),st(2)
        fstp st(2)
        fstp st(2)
        loop l1
    }
    return result;
}
```

```

        fstp result
        fstp result
    }

    return result;
}

int main (int, char **)
{
    int count = 1000000;

    float *source = new float [count];

    for (int i = 0 ; i < count ; ++i) {
        source [i] = static_cast <float> (rand ()) / static_
(RAND_MAX);
    }

    LARGE_INTEGER start, mid, end;

    float sum1 = 0.0f, sum2 = 0.0f;

    QueryPerformanceCounter (&start);

    sum1 = KahanSum (source, count);

    QueryPerformanceCounter (&mid);

    sum2 = AsmSum (source, count);

    QueryPerformanceCounter (&end);

    cout << "  C code: " << sum1 << " in " << (mid.QuadPart
endl;
    cout << "asm code: " << sum2 << " in " << (end.QuadPart
endl;

    return 0;
}

```

And some numbers from my PC running a default release build*:

C code: [500137](#) in [103884668](#)
asm code: [500137](#) in [52129147](#)

Out of interest, I swapped the loop with a `dec/jnz` and it made no difference to the timings - sometimes quicker, sometimes slower. I guess the memory limited aspect dwarfs other optimisations. (Editor's note: more likely the FP latency bottleneck is enough to hide the extra cost of `loop`. Doing two Kahan summations in parallel for the odd/even elements, and adding those at the end, could maybe speed this up by a factor of 2.)

Whoops, I was running a slightly different version of the code and it outputted the numbers the wrong way round (i.e. C was faster!). Fixed and updated the results.

Share

[edited Nov 11, 2019 at 14:14](#)

community wiki
[6 revs, 4 users](#) [84%](#)
[Skizz](#)

22 Or in GCC, you can untie the compiler's hands on floating point optimization (as long as you promise not to do anything with infinities or NaNs) by using the flag `-ffast-math`. They have an optimization level, `-Ofast` that is currently equivalent to `-O3 -ffast-math`, but in the future may include more optimizations that can lead to incorrect code generation in corner cases (such as code that relies on IEEE NaNs).

– [David Stone](#) Sep 9, 2012 at 19:04

2 Yeah, floats are not commutative, the compiler must do EXACTLY what you wrote, basically what @DavidStone said.
– [Alec Teal](#) Jan 2, 2014 at 7:53

2 Did you try SSE math? Performance was one of the reasons MS abandoned x87 completely in x86_64 and 80-bit long double in x86 – [phuciv](#) Mar 15, 2014 at 14:41

5 @Praxeolitic: FP add is commutative (`a+b == b+a`), but not associative (reordering of operations, so rounding of intermediates is different). re: this code: I don't think uncommented x87 and a `loop` instruction are a very awesome demonstration of fast asm. `loop` is apparently not actually a bottleneck because of FP latency. I'm not sure if he's pipelining FP operations or not; x87 is hard for humans to read. Two `fstp results` insns at the end is clearly not optimal. Popping the extra result from the stack would be better done with a non-store. Like `fstp st(0)` IIRC. – [Peter Cordes](#) Feb 8, 2016 at 9:19

2 @PeterCordes: An interesting consequence of making addition commutative is that while `0+x` and `x+0` are equivalent to each other, neither is always equivalent to `x`. – [supercat](#) Nov 14, 2019 at 21:26

65
votes



Without giving any specific example or profiler evidence, you can write better assembler than the compiler when you know more than the compiler.

In the general case, a modern C compiler knows much more about how to optimize the code in question: it knows how the processor pipeline works, it can try to reorder instructions quicker than a human can, and so on - it's basically the same as a computer being as good as or better than the best human player for boardgames, etc.

simply because it can make searches within the problem space faster than most humans. Although you theoretically can perform as well as the computer in a specific case, you certainly can't do it at the same speed, making it infeasible for more than a few cases (i.e. the compiler will most certainly outperform you if you try to write more than a few routines in assembler).

On the other hand, there are cases where the compiler does not have as much information - I'd say primarily when working with different forms of external hardware, of which the compiler has no knowledge. The primary example probably being device drivers, where assembler combined with a human's intimate knowledge of the hardware in question can yield better results than a C compiler could do.

Others have mentioned special purpose instructions, which is what I'm talking in the paragraph above - instructions of which the compiler might have limited or no knowledge at all, making it possible for a human to write faster code.

Share

answered [Feb 23, 2009 at 13:17](#)

community wiki
[Liedman](#)

Generally, this statement is true. The compiler does it's best to DWIW, but in some edge cases hand coding assembler gets the job done when realtime performance is a must. – [spoulson](#)
Feb 23, 2009 at 13:44

- 1 @Liedman: "it can try to reorder instructions quicker than a human can". OCaml is known for being fast and, surprisingly, its native-code compiler `ocamlc` skips instruction scheduling on x86 and, instead, leaves it up to the CPU because it can reorder more effectively at run-time. – [J D](#) Jan 28, 2012 at 10:48
-

- 1 Modern compilers do a lot, and it would take way too long to do by hand, but they're nowhere near perfect. Search gcc or llvm's bug trackers for "missed-optimization" bugs. There are many. Also, when writing in asm, you can more easily take advantage of preconditions like "this input can't be negative" that would be hard for a compiler to prove. – [Peter Cordes](#) Feb 8, 2016 at 9:23
-

54 In my job, there are three reasons for me to know and use assembly. In order of importance:
votes



1. Debugging - I often get library code that has bugs or incomplete documentation. I figure out what it's doing by stepping in at the assembly level. I have to do this about once a week. I also use it as a tool to debug problems in which my eyes don't spot the idiomatic error in C/C++/C#. Looking at the assembly gets past that.
2. Optimizing - the compiler does fairly well in optimizing, but I play in a different ballpark than most. I write

image processing code that usually starts with code that looks like this:

```
for (int y=0; y < imageHeight; y++) {  
    for (int x=0; x < imageWidth; x++) {  
        // do something  
    }  
}
```

the "do something part" typically happens on the order of several million times (ie, between 3 and 30). By scraping cycles in that "do something" phase, the performance gains are hugely magnified. I don't usually start there - I usually start by writing the code to work first, then do my best to refactor the C to be naturally better (better algorithm, less load in the loop etc). I usually need to read assembly to see what's going on and rarely need to write it. I do this maybe every two or three months.

3. doing something the language won't let me. These include - getting the processor architecture and specific processor features, accessing flags not in the CPU (man, I really wish C gave you access to the carry flag), etc. I do this maybe once a year or two years.

Share

edited Jul 21, 2013 at 0:51


community wiki

3 revs, 3 users 83%

plinth

1 @plinth: how do you mean "scraping cycles"? – [lang2](#) Apr 24, 2013 at 3:49

1 @lang2: it means getting rid of as many superfluous time spent in the inner loop as possible - anything that the compiler didn't manage to pull out, which may include using algebra to lift a multiply out of one loop to make it an add in the inner, etc.
– [plinth](#) Apr 25, 2013 at 14:31

1 [Loop tiling](#) appears to be unnecessary if you are only making one pass over the data. – [James M. Lay](#) Mar 21, 2015 at 16:30


@JamesM.Lay: If you only touch every element once, a better traversal order can give you spatial locality. (e.g. use all the bytes of a cache line that you touched, instead of looping down columns of a matrix using one element per cache line.)
– [Peter Cordes](#) Nov 14, 2019 at 21:30

41 Only when using some special purpose instruction sets the compiler doesn't support.

votes




To maximize the computing power of a modern CPU with multiple pipelines and predictive branching you need to structure the assembly program in a way that makes it a) almost impossible for a human to write b) even more impossible to maintain.

Also, better algorithms, data structures and memory management will give you at least an order of magnitude more performance than the micro-optimizations you can do in assembly.

community wiki



4 revs, 3 users 75%

Nir

-
- 4 +1, even though the last sentence doesn't really belong into this discussion - one would assume that assembler comes into play only after all possible improvements of algorithm etc have been realized. – [mghie](#) Feb 23, 2009 at 14:07
-
- 18 @Matt: Hand written ASM is often a *lot* better on some of the tiny CPUs EE's work with that have crappy vendor compiler support. – [Zan Lynx](#) Feb 28, 2009 at 2:44
-
- 5 "Only when using some special purpose instruction sets"?? You probably have never written a piece of hand-optimized asm code before. A moderately intimate knowledge of the architecture you are working on gives a good chance for you to generate a better code (size and speed) than your compiler. Obviously, as @mghie commented, you always start coding the best algos you can come with for you problem. Even for very good compilers, you really have to write your C code in a way that leads the compiler to the best compiled code. Otherwise, the generated code will be sub-optimal. – [ysap](#) Apr 5, 2011 at 20:33 
-
- 2 @ysap - on actual computers (not tiny underpowered embedded chips) in real world usage, the "optimal" code isn't going to be faster because for any large data set you performance is going to be limited by memory access and page faults (and if you don't have a large data set this is going to be fast either way and there's no point optimizing it) - those days I work mostly in C# (not even c) and the performance gains from the compacting memory manager out-weights the

overhead of the garbage collection, compacting and and JIT compilation. – [Nir](#) Apr 6, 2011 at 8:40

- 4 +1 for stating that compilers (esp. JIT) can do a *better* job than humans, *if* they are optimized for the hardware they are run on.
– [Sebastian](#) Nov 2, 2013 at 9:17
-

39
votes



Although C is "close" to the low-level manipulation of 8-bit, 16-bit, 32-bit, 64-bit data, there are a few mathematical operations not supported by C which can often be performed elegantly in certain assembly instruction sets:

1. Fixed-point multiplication: The product of two 16-bit numbers is a 32-bit number. But the rules in C says that the product of two 16-bit numbers is a 16-bit number, and the product of two 32-bit numbers is a 32-bit number -- the bottom half in both cases. If you want the *top* half of a 16x16 multiply or a 32x32 multiply, you have to play games with the compiler. The general method is to cast to a larger-than-necessary bit width, multiply, shift down, and cast back:

```
int16_t x, y;  
// int16_t is a typedef for "short"  
// set x and y to something  
int16_t prod = (int16_t)(((int32_t)x*y)>>16);`
```

In this case the compiler may be smart enough to know that you're really just trying to get the top half of a 16x16 multiply and do the right thing with the machine's native 16x16multiply. Or it may be stupid and require a library call to do the 32x32 multiply that's

way overkill because you only need 16 bits of the product -- but the C standard doesn't give you any way to express yourself.

2. Certain bitshifting operations (rotation/carries):

```
// 256-bit array shifted right in its entirety:
uint8_t x[32];
for (int i = 32; --i > 0; )
{
    x[i] = (x[i] >> 1) | (x[i-1] << 7);
}
x[0] >>= 1;
```

This is not too inelegant in C, but again, unless the compiler is smart enough to realize what you are doing, it's going to do a lot of "unnecessary" work.

Many assembly instruction sets allow you to rotate or shift left/right with the result in the carry register, so you could accomplish the above in 34 instructions: load a pointer to the beginning of the array, clear the carry, and perform 32 8-bit right-shifts, using auto-increment on the pointer.

For another example, there are [linear feedback shift registers](#) (LFSR) that are elegantly performed in assembly: Take a chunk of N bits (8, 16, 32, 64, 128, etc), shift the whole thing right by 1 (see above algorithm), then if the resulting carry is 1 then you XOR in a bit pattern that represents the polynomial.

Having said that, I wouldn't resort to these techniques unless I had serious performance constraints. As others have said, assembly is much harder to

document/debug/test/maintain than C code: the performance gain comes with some serious costs.

edit: 3. Overflow detection is possible in assembly (can't really do it in C), this makes some algorithms much easier.

Share

edited Mar 24, 2009 at 13:48

community wiki

[2 revs](#)

[Jason S](#)

23 Short answer? Sometimes.

votes



Technically every abstraction has a cost and a programming language is an abstraction for how the CPU works. C however is very close. Years ago I remember laughing out loud when I logged onto my UNIX account and got the following fortune message (when such things were popular):

The C Programming Language -- A language which combines the flexibility of assembly language with the power of assembly language.

It's funny because it's true: C is like portable assembly language.

It's worth noting that assembly language just runs however you write it. There is however a compiler in between C and the assembly language it generates and that is extremely important because **how fast your C code is has an awful lot to do with how good your compiler is.**

When gcc came on the scene one of the things that made it so popular was that it was often so much better than the C compilers that shipped with many commercial UNIX flavours. Not only was it ANSI C (none of this K&R C rubbish), was more robust and typically produced better (faster) code. Not always but often.

I tell you all this because there is no blanket rule about the speed of C and assembler because there is no objective standard for C.

Likewise, assembler varies a lot depending on what processor you're running, your system spec, what instruction set you're using and so on. Historically there have been two CPU architecture families: CISC and RISC. The biggest player in CISC was and still is the Intel x86 architecture (and instruction set). RISC dominated the UNIX world (MIPS6000, Alpha, Sparc and so on). CISC won the battle for the hearts and minds.

Anyway, the popular wisdom when I was a younger developer was that hand-written x86 could often be much faster than C because the way the architecture worked, it had a complexity that benefitted from a human doing it. RISC on the other hand seemed designed for compilers so noone (I knew) wrote say Sparc assembler. I'm sure such

people existed but no doubt they've both gone insane and been institutionalized by now.

Instruction sets are an important point even in the same family of processors. Certain Intel processors have extensions like SSE through SSE4. AMD had their own SIMD instructions. The benefit of a programming language like C was someone could write their library so it was optimized for whichever processor you were running on. That was hard work in assembler.

There are still optimizations you can make in assembler that no compiler could make and a well written assembler algoirthm will be as fast or faster than it's C equivalent. The bigger question is: is it worth it?

Ultimately though assembler was a product of its time and was more popular at a time when CPU cycles were expensive. Nowadays a CPU that costs \$5-10 to manufacture (Intel Atom) can do pretty much anything anyone could want. The only real reason to write assembler these days is for low level things like some parts of an operating system (even so the vast majority of the Linux kernel is written in C), device drivers, possibly embedded devices (although C tends to dominate there too) and so on. Or just for kicks (which is somewhat masochistic).

Share

[edited Apr 3, 2011 at 20:53](#)

There were a many people who used ARM assembler as the language of choice on Acorn machines (early 90's). IIRC they said that the small risc instruction set made it easier and more fun. But I suspect it's because the C compiler was a late arrival for Acorn, and the C++ compiler was never finished.

– [Andrew M](#) Feb 23, 2009 at 16:25

@AndrewM: Yeah, I wrote mixed-language applications in BASIC and ARM assembler for about 10 years. I learned C during that time but it wasn't very useful because it is as cumbersome as assembler and slower. Norcroft did some awesome optimizations but I think the conditional instruction set was a problem for the compilers of the day. – [J D](#) Jan 28, 2012 at 10:45

- 1 @AndrewM: well, actually ARM is kind of RISC done backwards. Other RISC ISAs were designed starting with what a compiler would use. The ARM ISA seems to have been designed starting with what the CPU provides (barrel shifter, condition flags → let's expose them in every instruction). – [ninjalj](#) Aug 23, 2013 at 12:37
-

17
votes



I'm surprised no one said this. The `strlen()` function is much faster if written in assembly! In C, the best thing you can do is

```
int c;  
for(c = 0; str[c] != '\0'; c++) {}
```

while in assembly you can speed it up considerably:

```

mov esi, offset string
mov edi, esi
xor ecx, ecx

lp:
mov ax, byte ptr [esi]
cmp al, cl
je end_1
cmp ah, cl
je end_2
mov bx, byte ptr [esi + 2]
cmp bl, cl
je end_3
cmp bh, cl
je end_4
add esi, 4
jmp lp

end_4:
inc esi

end_3:
inc esi

end_2:
inc esi

end_1:
inc esi

mov ecx, esi
sub ecx, edi

```


the length is in ecx. This compares 4 characters at time, so it's 4 times faster. And think using the high order word of eax and ebx, it will become *8 times faster* that the previous C routine!


community wiki

2 revs

BlackBear

-
- 3 How does this compare with the ones in strchr.nfshost.com/optimized_strlen_function ? – [ninjalj](#) Apr 5, 2011 at 21:19
-

@ninjalj: they are the same thing :) i didn't thought it can be done this way in C. It can be slightly improved I think
– [BlackBear](#) Apr 6, 2011 at 11:30 

There's still a bitwise AND operation before each comparison in the C code. It's possible that the compiler would be smart enough to reduce that to high and low byte comparisons, but I wouldn't bet money on it. There's actually a faster loop algorithm that's based on the property that `(word & 0xFEFEFEFF) & (~word + 0x80808080)` is zero iff all bytes in word are non-zero. – [user2310967](#) Feb 2, 2014 at 20:13 

16

votes



Point one which is not the answer.

Even if you never program in it, I find it useful to know at least one assembler instruction set. This is part of the programmers never-ending quest to know more and therefore be better. Also useful when stepping into frameworks you don't have the source code to and having at least a rough idea what is going on. It also helps you to understand JavaByteCode and .Net IL as they are both similar to assembler.

To answer the question when you have a small amount of code or a large amount of time. Most useful for use in

embedded chips, where low chip complexity and poor competition in compilers targeting these chips can tip the balance in favour of humans. Also for restricted devices you are often trading off code size/memory size/performance in a way that would be hard to instruct a compiler to do. e.g. I know this user action is not called often so I will have small code size and poor performance, but this other function that look similar is used every second so I will have a larger code size and faster performance. That is the sort of trade off a skilled assembly programmer can use.

I would also like to add there is a lot of middle ground where you can code in C compile and examine the Assembly produced, then either change you C code or tweak and maintain as assembly.

My friend works on micro controllers, currently chips for controlling small electric motors. He works in a combination of low level c and Assembly. He once told me of a good day at work where he reduced the main loop from 48 instructions to 43. He is also faced with choices like the code has grown to fill the 256k chip and the business is wanting a new feature, do you

1. Remove an existing feature
2. Reduce the size of some or all of the existing features maybe at the cost of performance.
3. Advocate moving to a larger chip with a higher cost, higher power consumption and larger form factor.

I would like to add as a commercial developer with quite a portfolio or languages, platforms, types of applications I have never once felt the need to dive into writing assembly. I have how ever always appreciated the knowledge I gained about it. And sometimes debugged into it.

I know I have far more answered the question "why should I learn assembler" but I feel it is a more important question then when is it faster.

so lets try once more You should be thinking about assembly

- working on low level operating system function
- Working on a compiler.
- Working on an extremely limited chip, embedded system etc

Remember to compare your assembly to compiler generated to see which is faster/smaller/better.

David.

Share

[edited Apr 5, 2011 at 20:25](#)

community wiki
[3 revs, 2 users 97%](#)
[David Waters](#)

or think that means a smart phone (32 bit, MB RAM, MB flash).

– [Martin](#) Jan 21, 2010 at 17:30

- 1 Time embedded applications are a great example! There are often weird instructions (even really simple ones like avr's `sbi` and `cbi`) that compilers used to (and sometimes still do) not take full advantage of, due to their limited knowledge of the hardware. – [felixpew](#) Jan 11, 2018 at 9:56
-

You write "This is part of the programmers never-ending quest to know more and therefore be better" but i beg to differ. I would express it as "This is part of some programmers' never-ending quest to know more and therefore be better". Most couldn't care less. – [Olof Forshell](#) Oct 27, 2020 at 11:37

15
votes

Matrix operations using SIMD instructions is probably faster than compiler generated code.



Share

answered [Feb 23, 2009 at 13:06](#)



community wiki
[Mehrdad Afshari](#)

Some compilers (the VectorC, if I remember correctly) generate SIMD code, so even that is probably no longer an argument for using assembly code. – [OregonGhost](#) Feb 23, 2009 at 13:08

- 5 For many of those situations you can use SSE intrinsics instead of assembly. This will make your code more portable (gcc visual c++, 64bit, 32bit etc) and you don't have to do register allocation. – [Laserallan](#) Feb 23, 2009 at 15:49
-

- 1 Sure you would, but the question did not ask where should I use assembly instead of C. It said when C compiler doesn't

generate a better code. I assumed a C source that is not using direct SSE calls or inline assembly. – [Mehrdad Afshari](#) Feb 23, 2009 at 16:12

9 Mehrdad is right, though. Getting SSE right is quite hard for the compiler and even in obvious (for humans, that is) situations most compilers don't employ it. – [Konrad Rudolph](#) Feb 23, 2009 at 16:30

1 You should use intrinsics for that, so it's not really assembler.. – [Nils](#) Jan 18, 2011 at 12:43

15
votes



A use case which might not apply anymore but for your nerd pleasure: On the Amiga, the CPU and the graphics/audio chips would fight for accessing a certain area of RAM (the first 2MB of RAM to be specific). So when you had only 2MB RAM (or less), displaying complex graphics plus playing sound would kill the performance of the CPU.

In assembler, you could interleave your code in such a clever way that the CPU would only try to access the RAM when the graphics/audio chips were busy internally (i.e. when the bus was free). So by reordering your instructions, clever use of the CPU cache, the bus timing, you could achieve some effects which were simply not possible using any higher level language because you had to time every command, even insert NOPs here and there to keep the various chips out of each others radar.

Which is another reason why the NOP (No Operation - do nothing) instruction of the CPU can actually make your

whole application run faster.

[EDIT] Of course, the technique depends on a specific hardware setup. Which was the main reason why many Amiga games couldn't cope with faster CPUs: The timing of the instructions was off.

Share

edited Feb 23, 2009 at 16:34

community wiki

2 revs

Aaron Digulla

The Amiga didn't have 16 MB of chip RAM, more like 512 kB to 2 MB depending on chipset. Also, a lot of Amiga games didn't work with faster CPUs due to techniques like you describe.

– [bk1e](#) Feb 23, 2009 at 15:07

-
- 1 @bk1e - Amiga produced a large range of different models of computers, the Amiga 500 shipped with 512K ram extended to 1Meg in my case. amigahistory.co.uk/amiedevsys.html is an amiga with 128Meg Ram – [David Waters](#) Feb 23, 2009 at 16:00

@bk1e: I stand corrected. My memory may fail me but wasn't chip RAM restricted to the first 24bit address space (i.e.

16MB)? And Fast was mapped above that? – [Aaron Digulla](#)

Feb 23, 2009 at 16:34

@Aaron Digulla: Wikipedia has more info about the distinctions between chip/fast/slow RAM:

en.wikipedia.org/wiki/Amiga_Chip_RAM – [bk1e](#) Feb 24, 2009 at 0:01

@bk1e: My mistake. The 68k CPU had only 24 address lanes, that's why I had the 16MB in my head. – [Aaron Digulla](#) Feb 24, 2009 at 9:08

14 A few examples from my experience:

votes



- Access to instructions that are not accessible from C. For instance, many architectures (like x86-64, IA-64, DEC Alpha, and 64-bit MIPS or PowerPC) support a 64 bit by 64 bit multiplication producing a 128 bit result. GCC recently added an extension providing access to such instructions, but before that assembly was required. And access to this instruction can make a huge difference on 64-bit CPUs when implementing something like RSA - sometimes as much as a factor of 4 improvement in performance.
- Access to CPU-specific flags. The one that has bitten me a lot is the carry flag; when doing a multiple-precision addition, if you don't have access to the CPU carry bit one must instead compare the result to see if it overflowed, which takes 3-5 more instructions per limb; and worse, which are quite serial in terms of data accesses, which kills performance on modern superscalar processors. When processing thousands of such integers in a row, being able to use `addc` is a huge win (there are superscalar issues with contention on the carry bit as well, but modern CPUs deal pretty well with it).

- SIMD. Even autovectorizing compilers can only do relatively simple cases, so if you want good SIMD performance it's unfortunately often necessary to write the code directly. Of course you can use intrinsics instead of assembly but once you're at the intrinsics level you're basically writing assembly anyway, just using the compiler as a register allocator and (nominally) instruction scheduler. (I tend to use intrinsics for SIMD simply because the compiler can generate the function prologues and whatnot for me so I can use the same code on Linux, OS X, and Windows without having to deal with ABI issues like function calling conventions, but other than that the SSE intrinsics really aren't very nice - the AltiVec ones seem better though I don't have much experience with them). As examples of things a (current day) vectorizing compiler can't figure out, read about [bitslicing AES](#) or [SIMD error correction](#) - one could imagine a compiler that could analyze algorithms and generate such code, but it feels to me like such a smart compiler is at least 30 years away from existing (at best).

On the other hand, multicore machines and distributed systems have shifted many of the biggest performance wins in the other direction - get an extra 20% speedup writing your inner loops in assembly, or 300% by running them across multiple cores, or 10000% by running them across a cluster of machines. And of course high level optimizations (things like futures, memoization, etc) are often much easier to do in a higher level language like ML

or Scala than C or asm, and often can provide a much bigger performance win. So, as always, there are tradeoffs to be made.

Share

answered [Oct 15, 2009 at 17:07](#)

community wiki

[Jack Lloyd](#)

Also, intrinsic based SIMD code tends to be *less* readable than the same code written in assembler: Much SIMD code relies on implicit reinterpretations of the data in the vectors, which is a PITA to do with the data types compiler intrinsics provide.

– [cmaster - reinstate monica](#) [Nov 20, 2017 at 10:06](#)

13

votes



I can't give the specific examples because it was too many years ago, but there were plenty of cases where hand-written assembler could out-perform any compiler. Reasons why:

- You could deviate from calling conventions, passing arguments in registers.
- You could carefully consider how to use registers, and avoid storing variables in memory.
- For things like jump tables, you could avoid having to bounds-check the index.

Basically, compilers do a pretty good job of optimizing, and that is nearly always "good enough", but in some situations

(like graphics rendering) where you're paying dearly for every single cycle, you can take shortcuts because you know the code, where a compiler could not because it has to be on the safe side.

In fact, I have heard of some graphics rendering code where a routine, like a line-draw or polygon-fill routine, actually generated a small block of machine code on the stack and executed it there, so as to avoid continual decision-making about line style, width, pattern, etc.

That said, what I want a compiler to do is generate good assembly code for me but not be too clever, and they mostly do that. In fact, one of the things I hate about Fortran is its scrambling the code in an attempt to "optimize" it, usually to no significant purpose.

Usually, when apps have performance problems, it is due to wasteful design. These days, I would never recommend assembler for performance unless the overall app had already been tuned within an inch of its life, still was not fast enough, and was spending all its time in tight inner loops.

Added: I've seen plenty of apps written in assembly language, and the main speed advantage over a language like C, Pascal, Fortran, etc. was because the programmer was far more careful when coding in assembler. He or she is going to write roughly 100 lines of code a day, regardless of language, and in a compiler language that's going to equal 3 or 400 instructions.

community wiki

3 revs

Mike Dunlavey

8 +1: "You could deviate from calling conventions". C/C++ compilers tend to suck at returning multiple values. They often use sret form where the caller stack allocates a contiguous block for a struct and passed a reference to it for the callee to fill it in. Returning multiple values in registers is several times faster. – J D Jan 28, 2012 at 10:52

1 @Jon: C/C++ compilers do that just fine when the function gets inlined (non-inlined functions have to conform to the ABI, this isn't a limitation of C and C++ but the linking model)
– Ben Voigt Feb 8, 2014 at 7:04

@BenVoigt: Here's a counter example

flyingfrogblog.blogspot.co.uk/2012/04/... – J D Feb 8, 2014 at 21:09

2 I don't see any function call getting inlined there. – Ben Voigt Feb 8, 2014 at 21:25

12

votes



More often than you think, C needs to do things that seem to be unnecessary from an Assembly coder's point of view just because the C standards say so.



Integer promotion, for example. If you want to shift a char variable in C, one would usually expect that the code would do in fact just that, a single bit shift.

The standards, however, enforce the compiler to do a sign extend to int before the shift and truncate the result to char afterwards which might complicate code depending on the target processor's architecture.

Share

answered [Mar 15, 2014 at 13:41](#)

community wiki
[mfro](#)

Quality compilers for small micros have for years been able to avoid processing the upper portions of values in cases where doing so could never meaningfully affect results. Promotion rules do cause problems, but most often in cases where a compiler has no way of knowing which corner cases are and are not relevant. – [supercat](#) Nov 14, 2019 at 21:29

10
votes



You don't actually know whether your well-written C code is really fast if you haven't looked at the disassembly of what compiler produces. Many times you look at it and see that "well-written" was subjective.

So it's not necessary to write in assembler to get fastest code ever, but it's certainly worth to know assembler for the very same reason.

Share

edited [Feb 24, 2009 at 4:50](#)

community wiki

2 revs

sharptooth

-
- 3 "So it's not necessary to write in assembler to get fastest code ever" Well, I've not seen a compiler do the optimal thing in any case that was not trivial. An experienced human can do better than the compiler in virtually all cases. So, it's absolutely necessary to write in assembler to get "the fastest code ever".
– [cmaster - reinstate monica](#) Nov 20, 2017 at 10:01
-

@cmaster In my experience compiler output is well, random. Sometimes it's really good and optimal and sometimes is "how could this garbage have been emitted". – [sharptooth](#) Nov 21, 2017 at 7:35

10
votes



Tight loops, like when playing with images, since an image may consist of millions of pixels. Sitting down and figuring out how to make best use of the limited number of processor registers can make a difference. Here's a real life sample:

<http://danbystrom.se/2008/12/22/optimizing-away-ii/>

Then often processors have some esoteric instructions which are too specialized for a compiler to bother with, but on occasion an assembler programmer can make good use of them. Take the XLAT instruction for example. Really great if you need to do table look-ups in a loop *and* the table is limited to 256 bytes!

Updated: Oh, just come to think of what's most crucial when we speak of loops in general: the compiler has often

no clue on how many iterations that will be the common case! Only the programmer know that a loop will be iterated MANY times and that it therefore will be beneficial to prepare for the loop with some extra work, or if it will be iterated so few times that the set-up actually will take longer than the iterations expected.



Share

edited Sep 5, 2009 at 2:39

community wiki
3 revs, 2 users 94%
Dan Byström

-
- 3 Profile directed optimization gives the compiler information about how often a loop is used. – [Zan Lynx](#) Feb 28, 2009 at 2:50
-

9
votes

I have read all the answers (more than 30) and didn't find a simple reason: assembler is faster than C if you have read and practiced the [Intel® 64 and IA-32 Architectures Optimization Reference Manual](#), **so the reason why assembly may be slower is that people who write such slower assembly didn't read the Optimization Manual.**

In the good old days of Intel 80286, each instruction was executed at a fixed count of CPU cycles. Still, since Pentium Pro, released in 1995, Intel processors became superscalar, utilizing Complex Pipelining: Out-of-Order Execution & Register Renaming. Before that, on Pentium,

produced in 1993, there were U and V pipelines. Therefore, Pentium introduced dual pipelines that could execute two simple instructions at one clock cycle if they didn't depend on one another. However, this was nothing compared with the Out-of-Order Execution & Register Renaming that appeared in Pentium Pro. This approach introduced in Pentium Pro is practically the same nowadays on most recent Intel processors.

Let me explain the Out-of-Order Execution in a few words. The fastest code is where instructions do not depend on previous results, e.g., you should always clear whole registers (by `movzx`) to remove dependency from previous values of the registers you are working with, so they may be renamed internally by the CPU to allow instruction execute in parallel or in a different order. Or, on some processors, false dependency may exist that may also slow things down, like [false dependency on Pentium 4 for inc/dec](#), so you may wish to use `add eax, 1` instead or `inc eax` to remove dependency on the previous state of the flags.

You can read more on Out-of-Order Execution & Register Renaming if time permits. There is plenty of information available on the Internet.

There are also many other essential issues like branch prediction, number of load and store units, number of gates that execute micro-ops, memory cache coherence protocols, etc., but the crucial thing to consider is the Out-of-Order Execution. Most people are simply not aware of


the Out-of-Order Execution. Therefore, they write their assembly programs like for 80286, expecting their instructions will take a fixed time to execute regardless of the context. At the same time, C compilers are aware of the Out-of-Order Execution and generate the code correctly. That's why the code of such uninformed people is slower, but if you become knowledgeable, your code will be faster.



There are also lots of optimization tips and tricks besides the Out-of-Order Execution. Just read the Optimization Manual above mentioned :-)

However, assembly language has its own drawbacks when it comes to optimization. According to Peter Cordes (see the comment below), some of the optimizations compilers do would be unmaintainable for large code-bases in hand-written assembly. For example, suppose you write in assembly. In that case, you need to completely change an inline function (an assembly macro) when it inlines into a function that calls it with some arguments being constants. At the same time, a C compiler makes its job a lot simpler—and inlining the same code in different ways into different call sites. There is a limit to what you can do with assembly macros. So to get the same benefit, you'd have to manually optimize the same logic in each place to match the constants and available registers you have.

Share

[edited Jun 4, 2021 at 1:06](#)



-
- 1 It's also worth pointing out that some of the optimizations compilers do would be *unmaintainable* for large code-bases in hand-written asm. e.g. completely changing an inline function (asm macro) when it inlines into a function that calls it with some args being constants that makes its job a lot simpler. And inlining the same code different ways into different callsites. There's a limit to what you can do with asm macros. So to get the same benefit, you'd have to manually optimize the same logic in each place to match the constants and free registers you have. – [Peter Cordes](#) Mar 10, 2021 at 7:19 
-

-
- 8 votes   I think the general case when assembler is faster is when a smart assembly programmer looks at the compiler's output and says "this is a critical path for performance and I can write this to be more efficient" and then that person tweaks that assembler or rewrites it from scratch.

Share

answered [Feb 23, 2009 at 13:11](#)

community wiki
[Doug T.](#)

-
- 7 votes   It all depends on your workload.
For day-to-day operations, C and C++ are just fine, but there are certain workloads (any transforms involving video

(compression, decompression, image effects, etc)) that pretty much require assembly to be performant.

They also usually involve using CPU specific chipset extensions (MME/MMX/SSE/whatever) that are tuned for those kinds of operation.

Share

answered [Feb 24, 2009 at 4:58](#)

community wiki
[Larry Osterman](#)

7
votes



It might be worth looking at [Optimizing Immutable and Purity by Walter Bright](#) it's not a profiled test but shows you one good example of a difference between handwritten and compiler generated ASM. Walter Bright writes optimising compilers so it might be worth looking at his other blog posts.

Share

edited [Dec 1, 2018 at 22:51](#)

community wiki
[2 revs, 2 users 80%](#)
[James Brooks](#)

6
votes

[Linux assembly howto](#), asks this question and gives the pros and cons of using assembly.



Share

answered [Feb 23, 2009 at 15:50](#)



community wiki
[pseudosaint](#)

6

votes



I have an operation of transposition of bits that needs to be done, on 192 or 256 bits every interrupt, that happens every 50 microseconds.

It happens by a fixed map(hardware constraints). Using C, it took around 10 microseconds to make. When I translated this to Assembler, taking into account the specific features of this map, specific register caching, and using bit oriented operations; it took less than 3.5 microsecond to perform.

Share

answered [May 24, 2009 at 15:28](#)

community wiki
[SurDin](#)

6

votes



The simple answer... One who *knows* assembly *well* (aka has the reference beside him, and is taking advantage of every little processor cache and pipeline feature etc) is guaranteed to be capable of producing much faster code than *any* compiler.

However the difference these days just doesn't matter in the typical application.


Share

edited Jan 21, 2010 at 16:38

community wiki

2 revs

Longpoke

-
- 1 You forgot to say "given a lot of time and effort", and "creating a maintenance nightmare". A colleague of mine was working on optimising a performance-critical section of OS code, and he worked in C much more than assembly, as it let him investigate the performance impact of high-level changes within a reasonable timeframe. – [Artelius](#) Jun 11, 2010 at 13:31 
-

I agree. Sometimes you use macros and scripts to generate assembly code in order to save time and develop rapidly. Most assemblers these days have macros; if not, you can make a (simple) macro pre-processor using a (fairly simple RegEx) Perl script. – [user1985657](#) Nov 9, 2014 at 21:44

This. Precisely. The compiler to beat the domain experts has not been invented yet. – [cmaster - reinstate monica](#) Nov 20, 2017 at 10:20

5 <http://cr.yp.to/qhasm.html> has many examples.

votes



Share

answered [Feb 23, 2009 at 16:27](#)



community wiki

4

votes



One of the possibilities to the CP/M-86 version of PolyPascal (sibling to Turbo Pascal) was to replace the "use-bios-to-output-characters-to-the-screen" facility with a machine language routine which in essence was given the x, and y, and the string to put there.

This allowed to update the screen much, much faster than before!

There was room in the binary to embed machine code (a few hundred bytes) and there was other stuff there too, so it was essential to squeeze as much as possible.

It turns out that since the screen was 80x25 both coordinates could fit in a byte each, so both could fit in a two-byte word. This allowed to do the calculations needed in fewer bytes since a single add could manipulate both values simultaneously.

To my knowledge there is no C compilers which can merge multiple values in a register, do SIMD instructions on them and split them out again later (and I don't think the machine instructions will be shorter anyway).

community wiki

Thorbjørn Ravn Andersen

4

votes



One of the more famous snippets of assembly is from Michael Abrash's texture mapping loop ([explained in detail here](#)):

```
add edx,[DeltaVFrac] ; add in dVFrac
sbb ebp,ebp ; store carry
mov [edi],al ; write pixel n
mov al,[esi] ; fetch pixel n+1
add ecx,ebx ; add in dUFrac
adc esi,[4*ebp + UVStepVCarry]; add in steps
```

Nowadays most compilers express advanced CPU specific instructions as intrinsics, i.e., functions that get compiled down to the actual instruction. MS Visual C++ supports intrinsics for MMX, SSE, SSE2, SSE3, and SSE4, so you have to worry less about dropping down to assembly to take advantage of platform specific instructions. Visual C++ can also take advantage of the actual architecture you are targetting with the appropriate /ARCH setting.



Share

answered [Feb 23, 2009 at 16:17](#)

community wiki
[MSN](#)

Even better, those SSE intrinsics are specified by Intel so they're actually fairly portable. – [James](#) Jan 20, 2010 at 19:37

4
votes

Given the right programmer, Assembler programs can always be made faster than their C counterparts (at least marginally). It would be difficult to create a C program where you couldn't take out at least one instruction of the Assembler.

Share

answered [Feb 23, 2009 at 16:24](#)

community wiki
[Beep beep](#)

This would be a bit more correct: "It would be difficult to create a **nontrivial** C program where ..." Alternatively, you could say: "It would be difficult to **find a real-world** C program where ..." Point is, there are trivial loops for which compilers do produce optimal output. Nevertheless, good answer.
– [cmaster - reinstate monica](#) Nov 20, 2017 at 10:25

4

votes



gcc has become a widely used compiler. Its optimizations in general are not that good. Far better than the average programmer writing assembler, but for real performance, not that good. There are compilers that are simply incredible in the code they produce. So as a general answer there are going to be many places where you can go into the output of the compiler and tweak the assembler for performance, and/or simply re-write the routine from scratch.

Share

answered [May 24, 2009 at 15:14](#)

community wiki

[old_timer](#)

-
- 9 GCC does extremely smart "platform-independent" optimisations. However, it is not so good at utilising particular instruction sets to their fullest. For such a portable compiler it does a very good job. – [Artelius](#) Jun 22, 2009 at 12:56
-
- 2 agreed. Its portability, languages coming in and targets going out are amazing. Being that portable can and does get in the way of being really good at one language or target. So the opportunities for a human to do better are there for a particular optimization on a specific target. – [old_timer](#) Jun 22, 2009 at 19:00
-
- +1: GCC certainly isn't competitive at generating fast code but I'm not sure that is because it is portable. LLVM is portable and I've seen it generate code 4x faster than GCCs. – [J D](#) Jan 28, 2012 at 12:35
-

I prefer GCC, since it's been rock solid for many years, plus it's available for almost every platform that can run a modern portable compiler. Unfortunately I have not been able to build LLVM (Mac OS X/PPC), so I will probably not be able to switch to it. One of the good things about GCC is that if you write code that builds in GCC, you're most likely keeping close to the standards, and you will be sure that it can be built for almost any platform. – user1985657 Nov 9, 2014 at 21:41

4

votes



Longpoke, there is just one limitation: time. When you don't have the resources to optimize every single change to code and spend your time allocating registers, optimize few spills away and what not, the compiler will win every single time. You do your modification to the code, recompile and measure. Repeat if necessary.

Also, you can do a lot in the high-level side. Also, inspecting the resulting assembly may give the IMPRESSION that the code is crap, but in practice it will run faster than what you think would be quicker. Example:

```
int y = data[i]; // do some stuff here.. call_function(y, ...);
```

The compiler will read the data, push it to stack (spill) and later read from stack and pass as argument. Sounds shite? It might actually be very effective latency compensation and result in faster runtime.

```
// optimized version call_function(data[i], ...); // not so optimized after all..
```

The idea with the optimized version was, that we have reduced register pressure and avoid spilling. But in truth, the "shitty" version was faster!

Looking at the assembly code, just looking at the instructions and concluding: more instructions, slower, would be a misjudgment.

The thing here to pay attention is: many assembly experts *think* they know a lot, but know very little. The rules change from architecture to next, too. There is no silver-bullet x86 code, for example, which is always the fastest. These days is better to go by rules-of-thumb:

- memory is slow
- cache is fast
- try to use cache better
- how often you going to miss? do you have latency compensation strategy?
- you can execute 10-100 ALU/FPU/SSE instructions for one single cache miss
- application architecture is important..
- .. but it doesn't help when the problem isn't in the architecture

Also, trusting too much into compiler magically transforming poorly-thought-out C/C++ code into "theoretically optimum" code is wishful thinking. You have to know the compiler and

tool chain you use if you care about "performance" at this low-level.

Compilers in C/C++ are generally not very good at re-ordering sub-expressions because the functions have side effects, for starters. Functional languages don't suffer from this caveat but don't fit the current ecosystem that well.

There are compiler options to allow relaxed precision rules which allow order of operations to be changed by the compiler/linker/code generator.

This topic is a bit of a dead-end; for most it's not relevant, and the rest, they know what they are doing already anyway.

It all boils down to this: "to understand what you are doing", it's a bit different from knowing what you are doing.

Share

answered [Sep 17, 2010 at 13:12](#)

community wiki
[tiredcoder](#)

1

2

Next