Check for updates

Optimizations in C++ Compilers

MATT GODBOLT

A PRACTICAL JOURNEY

Compilers are a necessary technology to turn high-level, easier-to-write code into efficient machine code for computers to execute. Their sophistication at doing this is often overlooked. You may spend a lot of time carefully considering algorithms and fighting error messages but perhaps not enough time looking at what compilers are capable of doing.

This article introduces some compiler and code generation concepts, and then shines a torch over a few of the very impressive feats of transformation your compilers are doing for you, with some practical demonstrations of my favorite optimizations. I hope you'll gain an appreciation for what kinds of optimizations you can expect your compiler to do for you, and how you might explore the subject further. Most of all, you may learn to love looking at the assembly output and may learn to respect the quality of the engineering in your compilers.

The examples shown here are in C or C++, which are the languages I've had the most experience with, but many of these optimizations are also available in other compiled languages. Indeed, the advent of front-end-agnostic compiler toolkits such as LLVM[3] means most of these optimizations work in the exact same way in languages such as Rust, Swift, and D.

ABOUT ME

I've always been fascinated by what compilers are capable of. I spent a decade making video games where every CPU cycle counted in the war to get more sprites, explosions, or complicated scenes on the screen than our competitors. Writing custom assembly, and reading the compiler output to see what it was capable of, was par for the course.

Fast-forward five years and I was at a trading company, having switched out sprites and polygons for fast processing of financial data. Just as before, knowing what the compiler was doing with code helped inform the way we wrote the code.

Obviously, nicely written, testable code is extremely important—especially if that code has the potential to make thousands of financial transactions per second. Being fastest is great, but not having bugs is even more important.

In 2012, we were debating which of the new C++11 features could be adopted as part of the canon of acceptable coding practices. When every nanosecond counts, you want to be able to give advice to programmers about how best to write their code without being antagonistic to performance. While experimenting with how code uses new features such as `auto`, lambdas, and range-based `for`, I wrote a shell script to run the compiler continuously and show its filtered output:

```
$ g++ /tmp/test.cc -O2 -c -S -o - -masm=intel \
    | c++filt \
    | grep -vE '\s+\.'
```

This proved so useful in answering all these "what if?" questions that I went home that evening and created Compiler Explorer.[1]

Over the years I've been constantly amazed by the lengths to which compilers go in order to take our code and turn it into a work of assembly code art. I encourage all compiled language programmers to learn a little assembly in order to appreciate what their compilers are doing for them. Even if you can't write it yourself, being able to read it is a useful skill.

All the assembly code shown here is for 64-bit x86 processors, as that's the CPU I'm most familiar with and is one of the most common server architectures. Some of the examples shown here are x86-specific, but in reality, many types of optimizations apply similarly on other architectures. Additionally, I cover only the GCC and Clang compilers, but equally clever optimizations show up on compilers from Microsoft Visual Studio and Intel.

OPTIMIZATION 101

This is far from a deep dive into compiler optimizations, but some concepts used by compilers are useful to know.

Many optimizations fall under the umbrella of *strength reduction:* taking expensive operations and transforming them to use less expensive ones. A very simple example of strength reduction would be taking a loop with a multiplication involving the loop counter:

```
for (int i = 0; i < 100; ++i)
{
    func(i * 1234);
}
```

Even on today's CPUs, multiplication is a little slower than simpler arithmetic, so the compiler will rewrite that loop to be something like:

```
for (int iTimes1234 = 0; iTimes1234 < 100 * 1234; i += 1234)
{
    func(iTimes1234);
}
```

Here, strength reduction took a loop involving multiplication and turned it into a sequence of equivalent operations using only addition.

There are many forms of strength reduction, more of which show up in the practical examples given later.

Another key optimization is *inlining*, in which the compiler replaces a call to a function with the body of that function. This removes the overhead of the call and often unlocks further optimizations, as the compiler can optimize the combined code as a single unit. You will see plenty of examples of this later.

Other optimization categories include:

➡ *Constant folding.* The compiler takes expressions whose values can be calculated at compile time and replaces them with the result of the calculation directly.

- *Constant propagation.* The compiler tracks the provenance of values and takes advantage of knowing that certain values are constant for all possible executions.
- *Common subexpression elimination.* Duplicated calculations are rewritten to calculate once and duplicate the result.
- *Dead code removal.* After many of the other optimizations, there may be areas of the code that have no effect on the output, and these can be removed. This includes loads and stores whose values are unused, as well as entire functions and expressions.
- *Loop invariant code movement.* The compiler recognizes that some expressions within a loop are constant for the duration of that loop and moves them outside of the loop. On top of this, the compiler is able to move a loop invariant conditional check out of a loop, and then duplicate the loop body twice: once if the condition is true, and once if it is false. This can lead to further optimizations.
- *Peephole optimizations.* The compiler takes short sequences of instructions and looks for local optimizations between those instructions.
- *Tail call removal.* A recursive function that ends in a call to itself can often be rewritten as a loop, reducing call overhead and reducing the chance of stack overflow.

The golden rule for helping the compiler optimize is to ensure it has as much information as possible to make the right optimization decisions. One source of information is your code: If the compiler can see more of your code, it's able to make better decisions. Another source of information is the compiler flags you use: telling your compiler the exact CPU architecture you're targeting can make a big difference. Of course, the more information a compiler has, the longer it could take to run, so there's a balance to be struck here.

Let's take a look at an example below, counting the number of elements of a vector that pass some test (compiled with GCC, optimization level 3, https://godbolt.org/z/acm19_count1):

```
int count(const vector<int> &vec)
{
    int numPassed = 0;
    for (size_t i = 0; i < vec.size(); ++i)
    {
        if (testFunc(vec[i]))
            numPassed++;
    }
    return numPassed;
}
```

If the compiler has no information about `testFunc`, it will generate an inner loop like the one on the next page:

```
.L4:
  mov edi, DWORD PTR [rdx+rbx*4]  ; read rbx'th element of vec
                                  ; (inlined vector::operator [])
  call testFunc(int)              ; call test function
  mov rdx, QWORD PTR [rbp+0]      ; reread vector base pointer
  cmp al, 1                       ; was the result of test true?
  mov rax, QWORD PTR [rbp+8]      ; reread the vector end pointer
  sbb r12d, -1                    ; add 1 if true, 0 if false
  inc rbx                         ; increment loop counter
  sub rax, rdx                    ; subtract end from begin...
  sar rax, 2                      ; and divide by 4 to get size()
                                  ; (inlined vector::size())
  cmp rbx, rax                    ; does loop counter equal size()?
  jb .L4                          ; loop if not
```

To understand this code, it's useful to know that a std::vector<> contains some pointers: one to the beginning of the data; one to the end of the data; and one to the end of the storage currently allocated below. The size of the vector is not directly stored, it's implied in the difference between the begin() and end() pointers. Note that the calls to vector<>::size() and vector<>::operator[] have been inlined completely.

```
template<typename T> struct _Vector_impl {
  T *_M_start;
  T *_M_finish;
  T *_M_end_of_storage;
};
```

In the assembly code, ebp points to the vector object, and the `begin()` and `end()` pointers are therefore `QWORD PTR [rbp+0]` and `QWORD PTR [rbp+8]`, respectively.

Another neat trick the compiler has done is to remove any branching: you might reasonably expect `if (testFunc(...))` would turn into a comparison and branch. Here the compiler does the comparison `cmp al, 1`, which sets the processor carry flag if `testFunc()` returned `false`, otherwise it clears it. The `sbb r12d, -1` instruction then subtracts `-1` with borrow, the subtract equivalent of carrying, which also uses the carry flag. This has the desired side effect: If the carry is clear (`testFunc()` returned `true`), it subtracts `-1`, which is the same as adding `1`. If the carry is set, it subtracts `-1 + 1`, which has no effect on the value. Avoiding branches can be advantageous in some cases if the branch isn't easily predictable by the processor.

It may seem surprising that the compiler reloads the `begin()` and `end()` pointers each loop iteration, and indeed it rederives `size()` each time too. However, the compiler is forced to do so: it has no idea what `testFunc()`does and must assume the worst. That is, it must assume that calls to `testFunc()` may cause the `vec` to be modified. The `const` reference here doesn't allow any additional optimizations for a couple of reasons: `testFunc()` may have a non-`const` reference to `vec` (perhaps through a global variable), or `testFunc()` might cast away `const`.

If, however, the compiler can see the body of `testFunc()`, and from this know that it does not in fact modify `vec`, the story is very different (https://godbolt.org/z/acm19_count2):

```
.L6:
  mov edi, DWORD PTR [rdx]   ; read next value
  call testFunc(int)         ; call testFunc with it
  cmp al, 1                  ; check return code
  sbb r8d, -1                ; add 1 if true, 0 otherwise
  add rdx, 4                 ; move to next element
  cmp rcx, rdx               ; have we hit the end?
  jne .L6                    ; loop if not
```

In this case the compiler has realized that the `vector`'s `begin()` and `end()` are constant during the operation of the loop. As such it has been able to realize that the call to `size()` is also a constant. Armed with this knowledge, it hoists these constants out of the loop, and then rewrites the index operation (`vec[i]`) to be a pointer walk, starting at `begin()` and walking up one `int` at a time to `end()`. This vastly simplifies the generated assembly.

In this example I gave the compiler a body to `testFunc()` but marked it as non-inlineable (a GNU extension) to demonstrate this optimization in isolation. In a more realistic codebase, the compiler could inline `testFunc()` if it believed it beneficial.

Another way to enable this optimization without exposing the body of the function to the compiler is to mark it as `[[gnu::pure]]` (another language extension). This promises the compiler that the function is pure—entirely a function of its arguments with no side effects.

Interestingly, using `range-for` in the initial example yields optimal assembly, even without knowing that testFunc() doesn't modify vec (https://godbolt.org/z/acm19_count3). This is because range-for is defined as a source code transformation that puts begin() and end() into local variables:

```
for (auto val : vec)
{
    if (testFunc(val))
        numPassed++;
}
```

is interpreted as:

```
{
    auto __begin = begin(vec);
    auto __end == end(vec);
    for (auto __it = __begin; __it != __end; ++__it)
    {
        if (testFunc(*__it))
            numPassed++;
    }
}
```

All things considered, if you need to use a "raw" loop, the modern range-for style is preferred: it's optimal even if the compiler can't see the body of called functions, and it is clearer to the reader. Arguably better still is to use the STL's count_if function to do all the work for you: the compiler still generates optimal code (https://godbolt.org/z/acm19_count4).

In the traditional single-translation-unit-at-a-time compilation model, function bodies are often hidden from call sites, as the compiler has seen only their declaration. LTO (link time optimization; also known as LTCG, for link time code generation) can be used to allow the compiler to see across translation unit boundaries. In LTO, individual translation units are compiled to an intermediate form instead of machine code. During the link process—when the entire program (or dynamic linked library) is visible—machine code is generated. The compiler can take advantage of this to inline across translation units, or at least use information about the side effects of called functions to optimize.

Enabling LTO for optimized builds can be a good win in general, as the compiler can see your whole program. I now rely on LTO to let me move more function bodies out of headers to reduce coupling, compile time, and build dependencies for debug builds and tests, while still giving me the performance I need in final builds.

Despite being a relatively established technology (I used LTCG in the early 2000s on the original Xbox), I've been surprised how few projects use LTO. In part this may be because programmers unintentionally rely on undefined behavior that becomes apparent only when the compiler gets more visibility: I know I've been guilty of this.

FAVORITE OPTIMIZATION EXAMPLES

Over the years I've collected a number of interesting real-world optimizations, both from first-hand experience optimizing my own code and from helping others understand their code on Compiler Explorer. Here are some of my favorite examples of how clever the compiler can be.

Integer division by a constant

It may be surprising to learn that—until very recently—about the most expensive thing you could do on a modern CPU is an integer divide. Division is more than 50 times more expensive than addition and more than 10 times more expensive than multiplication. (This was true until Intel's release of the Cannon Lake microarchitecture, where the maximum latency of a 64-bit divide was reduced from 96 cycles to 18.[6] This is only around 20 times slower than an addition, and 5 times more expensive than multiplication.)

Thankfully, compiler authors have some strength reduction tricks up their sleeves when it comes to division by a constant. I'm sure we've all realized that division by a power of two can often be replaced by a logical shift right—rest assured the compiler will do this for you. I would advise not writing a >> in your code to do division; let the compiler work it out for you. It's clearer, and the compiler also knows how to account properly for signed values: integer division truncates toward zero, and shifting down by itself truncates toward negative infinity.

However, what if you're dividing by a non-power-of-two value? Are you out of luck?

```
unsigned divideByThree(unsigned x)
{
    return x / 3;
}
```

Luckily the compiler has your back again. This code gets compiled to (https://godbolt.org/z/acm19_div3):

```
divideByThree(unsigned int):
  mov eax, edi          ; eax = edi
  mov edi, 2863311531   ; edi = 0xaaaaaaab
  imul rax, rdi         ; rax = rax * 0xaaaaaaab
  shr rax, 33           ; rax >>= 33
  ret
```

Not a divide instruction in sight. Just a shift, and a multiply by a strange large constant: the 32-bit unsigned input value is multiplied by `0xaaaaaaab`, and the resulting 64-bit value is shifted down by 33 bits. The compiler has replaced division with a cheaper multiplication by the reciprocal, in fixed point. The fixed point in this case is at bit 33, and the constant is one-third expressed in these terms (it's actually 0.33333333337213844). The compiler has an algorithm for determining appropriate fixed points and constants to achieve the division while preserving the same rounding as an actual division operation with the same precision over the range of the inputs. Sometimes this requires a number of extra operations—for example, in dividing by 1023 (https://godbolt.org/z/acm19_div1023):

```
divideBy1023(unsigned int):
  mov eax, edi
  imul rax, rax, 4198405
  shr rax, 32
  sub edi, eax
  shr edi
  add eax, edi
  shr eax, 9
  ret
```

The algorithm is well known and documented extensively in the excellent book, "*Hacker's Delight*".[8]

In short, you can rely on the compiler to do a great job of optimizing division by a compile-time-known constant.

You might be thinking: why is this such an important optimization? How often does one actually perform integer division, anyway? The issue is not so much with division itself as with the related modulus operation, which is often used in hash-map implementations as the operation to bring a hash value into the range of the number of hash buckets.

Knowing what the compiler can do here can lead to interesting hash-map implementations. One approach is to use a fixed number of buckets to allow the compiler to generate the perfect modulus operation without using the expensive divide instruction.

Most hash maps support rehashing to a different number of buckets. Naively this would lead to a modulus with a number known only at runtime, forcing the compiler to emit a slow divide instruction. Indeed, this is what the GCC libstdc++ library implementation of `std::unordered_map` does.

Clang's libc++ goes a little further: it checks if the number of buckets is a power of two, and if so skips the divide instruction in favor of a logical `AND`. Having a power-of-two bucket count is alluring as it makes the modulus operation fast, but in order to avoid excessive collisions it relies on having a good hash function. A prime-number bucket count gives decent collision resistance even for simplistic hash functions.

Some libraries such as `boost::multi_index` go a step further: instead of storing the actual number of buckets, they use a fixed number of prime-sized bucket counts.

```
size_t reduce(size_t hash, int bucketCountIndex) {
    switch (tableSizeIndex)
    {
        case 0: return hash % 7;
        case 1: return hash % 17;
        case 2: return hash % 37;
        // and so on...
    }
}
```

That way, for all possible hash-table sizes the compiler generates the perfect modulus code, and the only extra cost is to dispatch to the correct piece of code in the `switch` statement.

GCC 9 has a neat trick for checking for divisibility by a non-power-of-two (https://godbolt.org/z/acm19_multof3):

```
bool divisibleBy3(unsigned x)
{
    return x % 3 == 0;
}
```

This compiles to:

```
divisibleBy3(unsigned int):
  imul edi, edi, -1431655765    ; edi = edi * 0xaaaaaaab
  cmp edi, 1431655765           ; compare with 0x55555555
  setbe al                      ; return 1 if edi <= 0x55555555
  ret
```

This apparent witchcraft is explained very well by Daniel Lemire in his blog.[2] As an aside, it's possible to do these kinds of integer division tricks at runtime too. If you need to divide many numbers by the same value, you can use a library such as `libdivide`.[5]

Counting set bits

How often have you wondered, How many set bits are in this integer? Probably not all that often. But it turns out this simple operation is surprisingly useful in a number of cases. For example, calculating the Hamming distance between two bitsets, dealing with packed representations of sparse matrices, or handling the results of vector operations.

You might write a function to count the bits as follows:

```
int countSetBits(unsigned a)
{
    int count = 0;
    while (a != 0)
    {
        count++;
        a &= (a - 1); // clears the bottom set bit
    }
    return count;
}
```

Of note is the bit manipulation "trick" `a &= (a - 1);`, which clears the bottom-most set bit. It's a fun one to prove to yourself how it works on paper. Give it a go.

When targeting the Haswell microarchitecture, GCC 8.2

compiles this code to the assembly in (https://godbolt.org/z/acm19_bits):

```
countSetBits(unsigned int):
  xor eax, eax        ; count = 0
  test edi, edi       ; is a == 0?
  je .L4              ; if so, return
.L3:
  inc eax             ; count ++
  blsr edi, edi       ; a &= (a - 1);
  jne .L3             ; jump back to L3 if a != 0
  ret
.L4:
  Ret
```

Note how GCC has cleverly found the BLSR bit-manipulation instruction to pick off the bottom set bit. Neat, right? But not as clever as Clang 7.0:

```
countSetBits(unsigned int):
  popcnt eax, edi      ; count = number of set bits in a
  ret
```

This operation is common enough that there's an instruction on most CPU architectures to do it in one go: POPCNT (population count). Clang is clever enough to take a whole loop in C++ and reduce it to a single instruction. This is a great example of good instruction selection: Clang's code generator recognizes this pattern and is able to choose the perfect instruction.

I was actually being a little unfair here: GCC 9 also implements this, and in fact shows a slight difference:

```
countSetBits(unsigned int):
  xor eax, eax          ; count = 0
  popcnt eax, edi       ; count = number of set bits in a
  ret
```

At first glance this appears to be suboptimal: why on earth would you write a zero value, only to overwrite it immediately with the result of the "population count" instruction `popcnt`?

A little research brings up Intel CPU erratum SKL029: "`POPCNT` Instruction May Take Longer to Execute Than Expected"—there's a CPU bug! Although the `popcnt` instruction completely overwrites the output register `eax`, it is incorrectly tagged as depending on the prior value of `eax`. This limits the CPU's ability to schedule the instruction until any prior instructions writing to `eax` have completed—even though they have no impact.

GCC's approach here is to break the dependency on `eax`: the CPU recognizes `xor eax, eax` as a dependency-breaking idiom. No prior instruction can influence `eax` after `xor eax, eax`, and the `popcnt` can run as soon as its input operand `edi` is available.

This affects only Intel CPUs and seems to be fixed in the Cannon Lake microarchitecture, although GCC still emits `XOR` when targeting it.

Chained conditionals

Maybe you've never needed to count the number of set bits in an integer, but you've probably written code like this before:

```
bool isWhitespace(char c)
{
    return c == ' '
      || c == '\r'
      || c == '\n'
      || c == '\t';
}
```

Instinctively, I thought the code generation would be full of compares and branches, but both Clang and GCC use a clever trick to make this code pretty efficient. Below is GCC 9.1's output (https://godbolt.org/z/acm19_conds):

```
isWhitespace(char):
  xor eax, eax                 ; result = false
  cmp dil, 32                  ; is c > 32
  ja .L4                       ; if so, exit with false
  movabs rax, 4294977024       ; rax = 0x100002600
  shrx rax, rax, rdi           ; rax >>= c
  and eax, 1                   ; result = rax & 1
.L4:
  ret
```

The compilers turn this sequence of comparisons into a lookup table. The magic value loaded into rax is a 33-bit lookup table, with a one-bit in the locations where you would return true (indices 32, 13, 10, and 9 for ' ', \r, \n, and \t, respectively). The shift and & then pick out the cth bit and return it. Clang generates slightly different but broadly equivalent code. This is another example of strength reduction.

I was pleasantly surprised to see this kind of optimization. This is definitely the kind of thing that—prior to investigating in Compiler Explorer—I would have written manually assuming I knew better than the compiler.

One unfortunate thing I did notice while experimenting is—for GCC, at least—the order of the comparisons can affect the compiler's ability to make this optimization. If you switch the order of the comparison of the \r and \n, GCC generates the code below.

```
isWhitespace(char):
  cmp dil, 32   ; is c == 32?
  sete al       ; al = 1 if so, else 0
  cmp dil, 10   ; is c == 10?
  sete dl       ; dl = 1 if so, else 0
  or al, dl     ; al |= dl
  jne .L3       ; if al is non-zero return it (c was ` ` or `\n`)
  and edi, -5   ; clear bit 2 (the only bit that differs between
                ;               `\r` and `\t`)
  cmp dil, 9    ; compare with `\t`
  sete al       ; dl = 1 if so, else 0
.L3:
  ret
```

There's a pretty neat trick with the and to combine the comparison of \r and \t, but this seems worse than the code generated before. That said, a simplistic benchmark on

Quick Bench suggests the compare-based version might be a tiny bit faster in a predictable tight loop. Who ever said this was simple, eh?

Summation

Sometimes you need to add a bunch of things up. Compilers are extremely good at taking advantage of the vectorized instructions available in most CPUs these days, so even a pretty straightforward piece of code such as

```
int sumSquared(const vector<int> &v)
{
    int res = 0;
    for (auto i : v)
    {
        res += i * i;
    }
    return res;
}
```

gets turned into code whose core loop looks like below (https://godbolt.org/z/acm19_sum):

```
.loop:
  vmovdqu ymm2, YMMWORD PTR [rax]    ; read 32 bytes into ymm2
  add rax, 32                        ; advance to the next element
  vpmulld ymm0, ymm2, ymm2           ; square ymm2, treating as
                                     ;   8 32-bit values
  vpaddd ymm1, ymm1, ymm0            ; add to sub-totals
  cmp rax, rdx                       ; have we reached the end?
  jne .loop                          ; if not, keep looping
```

The compiler has been able to process eight values per instruction, by separating the total into eight separate subtotals for each one. At the end it sums across those subtotals to make the final total. It's as if the code was rewritten for you to look more like:

```
int res_[] = {0,0,0,0,0,0,0,0};
for (; index < v.size(); index += 8)
{
    // This can be performed by parallel instructions without
    // an actual loop. The following boils down to a couple
    // of vector instructions:
    for (size_t j = 0; j < 8; ++j)
    {
        auto val = v[index + j];
        res_[j] += val * val;
    }
}
res = res_[0] + res_[1]
    + res_[2] + res_[3]
    + res_[4] + res_[5]
    + res_[6] + res_[7];
```

Simply place the compiler's optimization level at a high enough setting and pick an appropriate CPU architecture to target, and vectorization kicks in. Fantastic!

This does rely on the fact that separating the totals into individual subtotals and then summing at the end is equivalent to adding them in the order the program specified. For integers, this is trivially true; but for floating-point data types this is not the case. Floating point operations are not associative: (a+b)+c is not the same as a+(b+c), as—among other things—the precision of the result of an addition depends on the relative magnitude of the two inputs.

This means, unfortunately, that changing the vector<int> to be a vector<float> doesn't result in the code you would ideally want. The compiler could use some vector operations (it can square eight values at once), but it is forced to sum across those values serially below (https://godbolt.org/z/acm19_sumf):

```
.loop:
  vmovups ymm4, YMMWORD PTR [rax]   ; read 32 bytes into ymm4
  add rax, 32                       ; advance
  vmulps ymm1, ymm4, ymm4           ; square 8 floats
                                    ; (the one parallel operation)
  vaddss xmm0, xmm0, xmm1           ; accumulate the first value
  vshufps xmm3, xmm1, xmm1, 85      ; shuffle things around
                                    ; (permutes the 8 floats
                                    ;  within the register)
  vshufps xmm2, xmm1, xmm1, 255     ; ...
  vaddss xmm0, xmm0, xmm3           ; accumulate the second value
  vunpckhps xmm3, xmm1, xmm1        ; more shuffling
  vextractf128 xmm1, ymm1, 0x1      ; ...
  vaddss xmm0, xmm0, xmm3           ; accumulate third...
  vaddss xmm0, xmm0, xmm2           ; and fourth value
  vshufps xmm2, xmm1, xmm1, 85      ; shuffling
  vaddss xmm0, xmm0, xmm1           ; accumulate fifth
  vaddss xmm0, xmm0, xmm2           ; and sixth
  vunpckhps xmm2, xmm1, xmm1        ; shuffle some more...
  vshufps xmm1, xmm1, xmm1, 255     ; ...
  vaddss xmm0, xmm0, xmm2           ; accumulate the seventh
  vaddss xmm0, xmm0, xmm1           ; and final value
  cmp rax, rcx                      ; are we done?
  jne .loop                         ; if not, keep going
```

This is unfortunate, and there's not an easy way around it. If you're absolutely sure the order of addition is not important in your case, you can give GCC the dangerous (but amusingly named) `-funsafe-math-optimizations` flag. This lets it generate this beautiful inner loop below (https://godbolt.org/z/acm19_sumf_unsafe):

```
.loop:
  vmovups ymm2, YMMWORD PTR [rax]    ; read 8 floats
  add rax, 32                        ; advance
  vfmadd231ps ymm0, ymm2, ymm2       ; for the 8 floats:
                                     ;   ymm0 += ymm2 * ymm2
  cmp rax, rcx                       ; are we done?
  jne .loop                          ; if not, keep going
```

Amazing stuff: processing eight floats at a time, using a single instruction to accumulate and square. The drawback is potentially unbounded precision loss. Additionally, GCC doesn't allow you to turn this feature on for just the functions you need it for—it's a per-compilation unit flag. Clang at least lets you control it in the source code with `#pragma Clang fp contract`.

While playing around with these kinds of optimizations, I discovered that compilers have even more tricks up their sleeves:

```
int sumToX(int x)
{
    int result = 0;
    for (int i = 0; i < x; ++i)
    {
        result += i;
    }
    return result;
}
```

GCC generates fairly straightforward code for this, and with appropriate compiler settings will use vector operations as above. Clang, however, generates this code (https://godbolt.org/z/acm19_sum_up):

```
sumToX(int): # @sumToX(int)
  test edi, edi               ; test x
  jle .zeroOrBelow            ; skip if x <= 0
  lea eax, [rdi - 1]          ; eax = x - 1
  lea ecx, [rdi - 2]          ; ecx = x - 2
  imul rcx, rax               ; rcx = ecx * eax
  shr rcx                     ; rcx >>= 1
  lea eax, [rcx + rdi]        ; eax = rcx + x
  add eax, -1                 ; return eax - 1
  ret
.zeroOrBelow:
  xor eax, eax                ; answer is zero
  ret
```

First, note there's no loop at all. Working through the generated code, you see that Clang returns:

$$\frac{(x-1)(x-2)}{2} + x - 1$$

It has replaced the iteration of a loop with a closed-form general solution of the sum. The solution differs from what I would naively write myself:

$$\frac{x(x-1)}{2}$$

This is presumably a result of the general algorithm Clang uses.

Further experimentation shows that Clang is clever enough to optimize many of these types of loops. Both Clang and GCC track loop variables in a way that allows this kind of optimization, but only Clang chooses to generate the closed-form version. It's not always less work: for small values of `x` the overhead of the closed-form solution might be more than just looping. Krister Walfridsson goes into great detail about how this is achieved in a blog post.[7]

It's also worth noting that in order to do this optimization, the compiler may rely on signed integer overflow being undefined behavior. As such, it can assume that your code cannot pass a value of `x` that would overflow the result (65536, in this case). If Clang can't make that assumption, it is sometimes unable to find a closed-form solution (https://godbolt.org/z/acm19_sum_fail).

Devirtualization

Although it seems to have fallen out of favor a little, traditional virtual-function-based polymorphism has its place. Whether it's to allow for genuine polymorphic behavior, or add a "seam" for testability, or allow for future extensibility, polymorphism through virtual functions can be a convenient choice.

As we know, though, virtual functions are slow. Or are they? Let's see how they affect the sum-of-squares example from earlier—something like below.

```
struct Transform
{
    int operator()(int x) const { return x * x; }
};

int sumTransformed(const vector<int> &v,
                   const Transform &transform)
{
    int res = 0;
    for (auto i : v)
    {
        res += transform(i);
    }
    return res;
}
```

Of course, this isn't polymorphic yet. A quick run through the compiler shows the same highly vectorized assembly (https://godbolt.org/z/acm19_poly1).

Now adding the `virtual` keyword in front of the `int operator()` should result in a much slower implementation, filled with indirect calls, right? Well, sort of (https://godbolt.org/z/acm19_poly2). There's a lot more going on than before, but at the core of the loop is something perhaps surprising.

```
; rdx points to the vtable
.L8:
  mov rax, QWORD PTR [rdx]   ; read the virtual function pointer
  mov esi, DWORD PTR [rbx]   ; read the next int element
  ; compare the function pointer with the address of the only
  ; known implementation...
  cmp rax, Transform::operator()(int) const
  jne .L5                    ; if it's not the only known impl,
                             ; then jump off to a more complex case
  imul esi, esi              ; square the number
  add rbx, 4                 ; move to next
  add r12d, esi              ; accumulate the square
  cmp rbp, rbx               ; finished?
  jne .L8                    ; if not, loop
```

What's happened here is GCC has made a bet. Given that it has seen only one implementation of the `Transform` class, it is likely going to be that one implementation that is used. Instead of blindly indirecting through the virtual function pointer, it has taken the slight hit of comparing the pointer against the only known implementation. If it matches, then the compiler knows what to do: it inlines the body of the `Transform::operator()` and squares it in place.

That's right: the compiler has inlined a virtual call. This is amazing, and was a huge surprise when I first discovered this. This optimization is called *speculative devirtualization* and is the source of continued research and improvement by compiler writers. Compilers are capable of devirtualizing at LTO time too, allowing for whole-program determination of possible function implementations.

The compiler has missed a trick, however. Note that at the top of the loop it reloads the virtual function pointer from the vtable every time. If the compiler were able to notice that this value remains constant if the called function doesn't change the dynamic type of `Transform`, this check could be hoisted out of the loop, and then there would be no dynamic checks in the loop at all. The compiler could use loop-invariant code motion to hoist the vtable check out of the loop. At this point the other optimizations could kick in, and the whole code could be replaced with the vectorized loop from earlier in the case of the vtable check passing.

You would be forgiven for thinking that the dynamic type of the object couldn't possibly change, but it's actually allowed by the standard: an object can placement `new` over itself so long as it returns to its original type by the time it's destructed. I recommend that you never do this, though. Clang has an option to promise you never do such horrible things in your code: `-fstrict-vtable-pointers`.

Of the compilers I use, GCC is the only one that does this as a matter of course, but Clang is overhauling its type system to leverage this kind of optimization more.[4]

C++11 added the `final` specifier to allow classes and virtual methods to be marked as not being further overridden. This gives the compiler more information about which methods may profit from such optimizations, and in some cases may even allow the compiler to avoid a virtual call completely (https://godbolt.org/z/acm19_poly3). Even without the `final` keyword, sometimes the analysis phase is able to prove that a particular concrete class is being used (https://godbolt.org/z/acm19_poly4). Such static devirtualization can yield significant performance improvements.

Related articles

C Is Not a Low-level Language
Your computer is not a fast PDP-11.
David Chisnall
https://queue.acm.org/detail.cfm?id=3212479

Uninitialized Reads
Understanding the proposed revisions to the C language
Robert C. Seacord
https://queue.acm.org/detail.cfm?id=3041020

You Don't Know Jack about Shared Variables or Memory Models
Data races are evil.
Hans-J. Boehm, Sarita V. Adve
https://queue.acm.org/detail.cfm?id=2088916

CONCLUSION

Hopefully, after reading this article, you'll appreciate the lengths to which the compiler goes to ensure efficient code generation. I hope that some of these optimizations are a pleasant surprise and will factor in your decisions to write clear, intention-revealing code and leave it to the compiler to do the right thing. I've reinforced the idea that the more information the compiler has, the better job it can do. This includes allowing the compiler to see more of your code at once, as well as giving the compiler the right information about the CPU architecture you're targeting. There's a tradeoff to be made in giving the compiler more information: it can make compilation slower. Technologies such as link time optimization can give you the best of both worlds.

Optimizations in compilers continue to improve, and upcoming improvements in indirect calls and virtual function dispatch might soon lead to even faster polymorphism. I'm excited about the future of compiler optimizations. Go take a look at your compiler's output.

Thanks

The author would like to extend his thanks to Matt Hellige, Robert Douglas, and Samy Al Bahra, who gave feedback on drafts of this article.

References

1. Godbolt, M. 2012. Compiler explorer; https://godbolt.org/.
2. Lemire, D. 2019. Faster remainders when the divisor is a constant: beating compilers and libdivide. https://lemire.me/blog/2019/02/08/faster-remainders-when-the-divisor-is-a-constant-beating-compilers-and-libdivide/.
3. LLVM. 2003. The LLVM compiler infrastructure.; https://llvm.org.
4. Padlewski, P. 2018. RFC: Devirtualization v2. LLVM; http://lists.llvm.org/pipermail/llvm-dev/2018-March/121931.html.
5. ridiculous_fish. 2010. Libdivide; https://libdivide.com/.
6. Uops. Uops.info Instruction Latency Tables; https://uops.info/table.html.
7. Walfridsson, K. 2019. How LLVM optimizes power sums;

https://kristerw.blogspot.com/2019/04/how-llvm-optimizes-geometric-sums.html.
8. Warren, H. S. 2012. Hacker's Delight. 2nd edition. Addison-Wesley Professional.

Matt Godbolt *is the creator of the Compiler Explorer website. He is passionate about writing efficient code. He currently works at Aquatic Capital, and has worked on low-latency trading systems, on mobile apps at Google, run his own C++ tools company and spent more than a decade making console games. When he's not hacking on Compiler Explorer, Matt enjoys writing emulators for old 8-bit computer hardware.*