Incrementing Vectors

Aug 26, 2019 • performance c++

What's faster, incrementing a `std::vector` of `uint8_t` or of `uint32_t`?

To make it concrete, let's consider these two implementations:

```
void vector8_inc(std::vector<uint8_t>& v) {
    for (size_t i = 0; i < v.size(); i++) {
        v[i]++;
    }
}

void vector32_inc(std::vector<uint32_t>& v) {
    for (size_t i = 0; i < v.size(); i++) {
        v[i]++;
    }
}
```

Let's Guess

It's easy to answer this question with a benchmark, and we will get to that soon, but for now let's take some guesses (also known as "reasoning from first principles" if you want to pretend there is a lot of science in it).

First, we might reasonably ask: *How big are these vectors*?

Let's pick a number then. I'm going with 20,000 elements in each.

Then knowing the hardware we're testing on, Intel Skylake, we can check the characteristics of 8-bit and 32-bit add immediate instructions. It turns out that their primary performance characteristics are identical: 1 per cycle throughput and 4 cycles latency through memory[1]. In this case, we don't expect latency to matter since each add is independent, so we might hope this runs at 1 cycle per element, if the rest of the loop overhead can run in parallel.

We might also note that 20,000 elements means a working set of 20 KB for the `uint8_t` vector, but 80 KB for the `uint32_t` vector. The former fits cleanly in the L1 cache of modern x86 machines, while the second doesn't. So maybe the 8-bit version will have a leg up due to caching effects?

Finally, we might note that this problem seems like a textbook case for auto-vectorization: a loop with a known number of iterations doing an arithmetic operation on adjacent elements in memory. In that case, we might expect the 8-bit version to have a huge advantage over the 32-bit version, since every vector operation will process 4 times as many elements, and in general, on Intel, the byte element vector operations have the same performance as their 32-bit element counterparts.

Enough prologue. Let's benchmark this.

The Benchmark

For 20,000 elements, I get the following timings in *cycles per element* with `gcc 8` and `clang 8` at optimization levels `-O1` through `-O3`:

Compiler	Element	O1	O2	O3
gcc 8	`uint8_t`	2.0	2.0	2.0
gcc 8	`uint32_t`	2.3	1.3	0.2
clang 8	`uint8_t`	9.2	2.0	2.0
clang 8	`uint32_t`	9.2	0.2	0.2

The conclusion is that except at `-O1`, `uint32_t` is faster than `uint8_t` and sometimes dramatically so: for gcc at `-O3` the speedup was 5.4x and for clang at either `-O2` and `-O3` the speedup was 8.0x. Yes, incrementing 32-bit integers in a `std::vector` is up to *eight times faster* than incrementing 8-bits on a popular compiler with common optimization settings.

As usual, we hope the assembly will tell us what is going on.

Here's gcc 8 at `-O2` where the 32-bit version was "only" 1.5x faster than the 8-bit one[2]:

8-bit:

```
.L3:
    inc     BYTE PTR [rdx+rax]
```

```
    mov     rdx, QWORD PTR [rdi]
    inc     rax
    mov     rcx, QWORD PTR [rdi+8]
    sub     rcx, rdx
    cmp     rax, rcx
    jb      .L3
```

32-bit:

```
.L9:
    inc     DWORD PTR [rax]
    add     rax, 4
    cmp     rax, rdx
    jne     .L9
```

The 32-bit code looks like we'd expect from a not-unrolled[3] loop: an increment[4] with a memory destination, and then three loop control instructions: `add rax, 4` to increment the induction variable[5], and a `cmp` `jne` pair to check and branch on the loop termination condition. It looks fine – an unroll would help to amortize the loop overhead, getting us closer to 1 cycle/element store limit[6], but good enough for open source work.

So What's going on with the 8-bit version? In addition to the memory-destination `inc`, you have two other loads from memory and some stuff like a `sub` that appeared out of nowhere.

Here's an annotated version:

8-bit:

```
.L3:
    inc     BYTE PTR [rdx+rax]     ; increment memory at v[i]
    mov     rdx, QWORD PTR [rdi]   ; load v.begin
    inc     rax                    ; i++
    mov     rcx, QWORD PTR [rdi+8] ; load v.end
    sub     rcx, rdx               ; end - start (i.e., vector.size())
    cmp     rax, rcx               ; i < size()
    jb      .L3                    ; next itr if i < size()
```

Here `vector::begin` and `vector::end` are the internal pointers that `std::vector` keeps to indicate the beginning and of the elements stored in its internal storage[7] - basically the same values used to implement `vector::begin()` and `vector::end()` (although the type differs). So all the extra instructions appear as a consequence of calculating `vector.size()`. Maybe doesn't seem too weird? Well maybe it does, because of course the 32-bit version uses `size()` also, but the `size()`

related instructions don't appear that loop. Instead, the `size()` calculation happens once, outside the loop.

What's the difference? The short answer is *pointer aliasing*. The long answer is up next.

The Long Answer

The vector `v` is passed to the function as a reference, which boils down to a pointer under the covers. The compiler needs to use the `v::begin` and `v::end` members of the vector to calculate `size()`, and in the source we've written `size()` is evaluated *every iteration*. The compiler isn't necessarily a slave to the source however: it can hoist the result of the `size()` function above the loop, but only if it can prove that this doesn't change the semantics of the program. In that regard, the only problematic part of the loop is the increment `v[i]++`. This writes to some unknown memory location. The question is: *can that write change the value of* `size()`?

In the case of writes to a `std::vector<uint32_t>` (which boil down to writes to `uint32_t *`) the answer is **no, it cannot change size()**. Writes to `uint32_t` objects can only change `uint32_t` objects, and the pointers involved in the calculation of `size()` are not `uint32_t` objects[8].

On the other hand, for `uint8_t`, at least for common compilers[9], the answer is **yes, it could in theory change the value of** `size()`, because `uint8_t` is a typedef for `unsigned char`, and `unsigned char` (and `char`) arrays are allowed to *alias any type*. That means that writes through `uint8_t` pointers are treated as potentially updating any memory of unknown provenance[10]. So the compiler assumes that every increment `v[i]++` potentially modifies `size()` and hence must recalculate it every iteration.

Now, you and I know writes to the memory pointed to by a `std::vector` are never going modify its own `size()` - that would mean that the vector object itself has somehow been allocated *inside its own heap storage*, an almost impossible chicken-and-egg scenario[11]. Unfortunately, the compiler doesn't know that!

What about the rest

So that explains the small performance difference between the `uint8_t` and `uint32_t` for `gcc -O2`. How about the huge, up to 8x difference, we see for clang or gcc at `-O3`?

That's easy - clang can autovectorize this loop in the `uint32_t` case:

```
.LBB1_6:
    vmovdqu ymm1, ymmword ptr [rax + 4*rdi]
    vmovdqu ymm2, ymmword ptr [rax + 4*rdi + 32]
    vmovdqu ymm3, ymmword ptr [rax + 4*rdi + 64]
    vmovdqu ymm4, ymmword ptr [rax + 4*rdi + 96]
    vpsubd  ymm1, ymm1, ymm0
    vpsubd  ymm2, ymm2, ymm0
    vpsubd  ymm3, ymm3, ymm0
    vpsubd  ymm4, ymm4, ymm0
    vmovdqu ymmword ptr [rax + 4*rdi], ymm1
    vmovdqu ymmword ptr [rax + 4*rdi + 32], ymm2
    vmovdqu ymmword ptr [rax + 4*rdi + 64], ymm3
    vmovdqu ymmword ptr [rax + 4*rdi + 96], ymm4
    vmovdqu ymm1, ymmword ptr [rax + 4*rdi + 128]
    vmovdqu ymm2, ymmword ptr [rax + 4*rdi + 160]
    vmovdqu ymm3, ymmword ptr [rax + 4*rdi + 192]
    vmovdqu ymm4, ymmword ptr [rax + 4*rdi + 224]
    vpsubd  ymm1, ymm1, ymm0
    vpsubd  ymm2, ymm2, ymm0
    vpsubd  ymm3, ymm3, ymm0
    vpsubd  ymm4, ymm4, ymm0
    vmovdqu ymmword ptr [rax + 4*rdi + 128], ymm1
    vmovdqu ymmword ptr [rax + 4*rdi + 160], ymm2
    vmovdqu ymmword ptr [rax + 4*rdi + 192], ymm3
    vmovdqu ymmword ptr [rax + 4*rdi + 224], ymm4
    add     rdi, 64
    add     rsi, 2
    jne     .LBB1_6
```

The loop as been unrolled 8x, and this is basically as fast as you'll get, approaching one vector (8 elements) per cycle in the L1 cache (limited by one store per cycle[12]).

No vectorization happens for the `uint8_t` case, because need to recalculate `size()` check for loop termination in between every element completely inhibits it. So the ultimate cause is the same as the first case we looked at, but the impact is much bigger.

Auto-vectorization explains all the very fast results: gcc only autovectorizes at `-O3` but clang does it at `-O2` and `-O3` by default. The `-O3` result for gcc is somewhat slower than clang because it does not unroll the autovectorized loop.

Fixing it

So now that we know what the problem is, how do we fix it?

First, we'll try one that doesn't work – using a more idiomatic iterator-based loop, like this:

```
    for (auto i = v.begin(); i != v.end(); ++i) {
        (*i)++;
    }
```

It generates somewhat better code, compared to the `size()` version, at `gcc -O2`:

```
.L17:
    add     BYTE PTR [rax], 1
    add     rax, 1
    cmp     QWORD PTR [rdi+8], rax
    jne     .L17
```

Two extra reads have turned into one, because you just compare to the `end` pointer in the vector, rather than recalculate `size()` from both the begin and end pointers. It actually ties the `uint32_t` version in instruction count since the extra load was folded into the comparison. However, the underlying problem is still there and auto-vectorization is inhibited, so there is still a very large gap between `uint8_t` and `uint32_t` - more than 5x for both gcc and clang for build where auto-vectorization is allowed.

Let's try again. Again, we will fail or rather perhaps *succeed* in finding *another* way that doesn't work.

This is the one where we calculate `size()` only once, before the loop, and stash it in a local variable:

```
    for (size_t i = 0, s = v.size(); i < s; i++) {
        v[i]++;
    }
```

It seems like this should work, right? The problem was `size()` and now we are saying "freeze `size()` into a local `s` at the start of the loop, and the compiler knows locals cannot overlap with anything". Basically we manually hoist the thing the compiler couldn't. It actually does generate better code (compared to the original version):

```
.L9:
    mov     rdx, QWORD PTR [rdi]
    add     BYTE PTR [rdx+rax], 1
    add     rax, 1
```

```
    cmp     rax, rcx
    jne     .L9
```

There is only one extra read and no `sub`. So what is that extra read (`rdx, QWORD PTR [rdi]`) doing if it isn't part of calculating the size? It's loading the `data()` pointer from `v`!

Effectively, the implementation of `v[i]` is `*(v.data() + i)` and the member underlying `data()` (which is fact just the `begin` pointer) has the same potential problem as `size()`. I just didn't really notice it the original example because we got that read "for free" since we were loading it anyways to get the size.

We're almost there, I promise. We just need to remove the dependence on *anything* stored inside the `std::vector` in the loop. This is easiest if we modify the idiomatic iterator based approach like this:

```
for (auto i = v.begin(), e = v.end(); i != e; ++i) {
    (*i)++;
}
```

The results are dramatic (here we compare only `uint8_t` loops, with and without saving the end iterator before the loop):

Compiler	Loop Check	-O1	-O2	-O3
gcc 8	`i != v.end()`	1.3	1.3	1.3
gcc 8	`i != e`	1.3	1.3	0.06
gcc speedup		1x	1x	**20x**
clang 8	`i != v.end()`	29.9	1.3	1.3
clang 8	`i != e`	20.3	0.06	0.06
clang speedup		1.5x	**20x**	**20x**

After this small change, the speedup is a factor of *twenty* for the cases where vectorization is able to kick in. We didn't just tie the `uint32_t` case either: for gcc `-O3` and clang `-O2` and `-O3` the speedup of the `uint8_t` code over `uint32_t` is almost exactly 4x, as we originally expected after vectorization: after all, 4 times as many elements are handled per vector, and we need 4 times less bandwidth to whatever level of cache is involved[13].

Some of you who've gotten this far have probably screaming, nearly since the start:

What about the range-based for loop introduced in C++11?

Good news! It works fine. It is syntactic sugar for almost exactly the iterator-based version above, with the end pointer captured once before the loop, and so it performs equivalently.

At the opposite end of the scale from modern C++, if I had just summoned my inner caveman and written a C-like function, we would have also been fine:

```
void array_inc(uint8_t* a, size_t size) {
    for (size_t i = 0; i < size; i++) {
        a[i]++;
    }
}
```

Here, the array pointer `a` and the `size` are by-value function parameters, so they cannot be be modified by any writes through `a`[14], just like local variables. It performs equivalently to the other successful options.

Finally, for compilers that offer it, you can use `__restrict` on the vector declaration, like this[15]:

```
void vector8_inc_restrict(std::vector<uint8_t>& __restrict v) {
    for (size_t i = 0; i < v.size(); i++) {
        v[i]++;
    }
}
```

The `__restrict` keyword is not standard C++, but it is in C since C99 (as `restrict`). For those compilers that implement it as an extension in C++, you can reasonably expect that it follows the C semantics. Of course, C doesn't have references at all - so you can mentally substitute the vector reference with a pointer to a vector.

Note that restrict is not *transitive:* the `__restrict` qualifier on the `std::vector` reference doesn't apply restrict to the heap storage backing `v.data()`, it only applies to the members of vector itself. In this case, that is enough because (as we saw above with local variables), proving to the compiler that the vector start/finish members themselves don't overlap anything is enough. The distinction is still relevant: it means that writes to `v.data()` may still cause aliasing problems for *other* objects in your function.

Disappointment

That brings us to our last, and perhaps most disappointing note: all of the ways we've fixed the function above really only apply directly to this specific case of a vector interfering with itself. We've fixed it by hoisting or isolating the state backing the `size()` or `end()` calls in the vector, and *not* by teaching the compiler that writes to the vector data don't modify anything. That doesn't scale well as our function grows.

The aliasing loophole is still there, and those writes still alias "anything" - it's just that there isn't anything else interesting to alias in this function ... yet. As soon as you add more to the function, it *bytes* us again. A random example. You'll be fighting the compiler to the end as long as you have writes to `uint8_t` arrays in your core loops[16].

Comments

Feedback of any type is welcome. I don't have a comments system[17] yet, so as usual I'll outsource discussion to this HackerNews thread.

> If you liked this post, check out the homepage for others you might enjoy.

---

1. *Through memory* here means that the dependency chain is through memory: writes to the same location need to read the last value written and hence are dependent (in practice, store-to-load forwarding will be involved if the writes are close in time). There are other ways that `add` to memory might be dependent, such as through the addressing calculation, but these don't apply here. ↩

2. Only the core loop is shown, the setup code is simple and fast. Plug it into godbolt if you want to see everything. ↩

3. Is *not-unrolled* a double negative? Should it simply be a *rolled* loop? Anyway, gcc generally doesn't unroll loops even at `-O2` or `-O3` except in some special cases like small *compile-time known* iteration counts. This often hurts it benchmarks versus clang, but helps a lot for code size. gcc will unroll loops if you use profile guided optimization, or you can request it on the command using `-funroll-loops`. ↩

4. Actually gcc using `inc DWORD PTR [rax]` is a missed optimization: `add [rax], 1` is almost always better since it is only 2 uops in the fused domain, compared to 3 for `inc`. In this case it makes only a difference of about 6%, but with a slightly unrolled loop where the store was the only repeated element it could make a bigger difference (for a loop unrolled even more it would probably stop mattering as you hit the 1 store/cycle limit, not any total uop limit). ↩

5. I call it the *induction* variable rather than simply `i` as in the source, because the compiler has transformed single increments of `i` in the original code into 4-byte increments of a pointer stored in `rax` and adjusted the loop condition appropriately. Basically our vector-indexing loop has been re-written as a pointer/iterator incrementing one, a form of *strength reduction*. ↩

6. In fact, if you add `-funroll-loops` you get to 1.08 cycles per element with gcc's 8x unroll. Even with this flag, gcc *doesn't* unroll the 8-bit version, so the gap between the two grows a lot! ↩

7. These are private members and their names will be implementation specific, but they aren't actually called `start` and `finish` in stdlibc++ as used by gcc, rather they are `_Vector_base::_Vector_impl::_M_start` and `_Vector_base::_Vector_impl::_M_finish`, i.e., they live in a structure `_Vector_impl` which is a member `_M_impl` (in fact the only member) of `_Vector_base`, which is the base class of `std::vector`. Whew! Luckily the compiler removes all this abstraction just fine. ↩

8. The standard makes no guarantees about what the internal types of `std::vector` members will be, but in libstdc++'s implementation they are simply `Alloc::pointer` where `Alloc` is the allocator for the vector, which for the default `std::allocated` is simply `T*`, i.e., a raw pointer to object, in this case `uint32_t *`. ↩

9. The caveat *at least for common compilers* appears because it seems to be the case that `uint8_t` could be treated as a type distinct from `char`, `signed char` and `unsigned char`. Because the aliasing loophole only applies to *character types*, this would presumably mean that the rule wouldn't apply to `uint8_t` and so they would behave like any other non-char type. No compiler I'm aware of actually implement this: they all `typedef` `uint8_t` to `unsigned char`, so the distinction is invisible to the compiler even if it wanted to act on it. ↩

10. *Unknown provenance* here just means that the compiler doesn't know where the memory points to or how it was created. Arbitrary pointers passed in to a function qualify, global and static member variables qualify. Local variables, on the other hand, have a known storage location that doesn't overlap with anything else, so those are usually safe from aliasing issues (assuming you don't allow the local to escape somehow). Pointers created from `malloc` or `new` calls that the compiler can *see* at the point of use behave a lot like local variables, at least for some compilers: the compiler knows these calls return regions of memory that don't overlap with anything else. Usually the compiler cannot see the originating `malloc` or `new` call, however. ↩

11. Maybe it is *possible* for `std::vector`? E.g., create a `std::vector<uint8_t> a` of a suitable size, then placement-new a new vector `b` *inside* the `a.data()` array, at a suitably aligned location. Then `std::swap(a, b)` which will just swap the underlying storage, so now `b` lives inside its own storage? Or skip the swap and use the move constructor directly when placement-new constructing `b`. Such a vector has no practical purpose and will break when you do pretty much anything (e.g., adding an element may expand the vector), destroying the original backing storage and hence the vector itself. ↩

12. It only gets 8 element per cycle suitably aligned: i.e., aligned by 32. In this test my `std::vector` instances were lucky enough to get that alignment by default. ↩

13. In fact, you could get a better than 4x speedup in the case that the smaller element case fit in a inner cache level but the larger one didn't. That is actually the case here, with the 8-bit case fitting in the L1 cache, but the 32-bit case being more than 2x as large as the L1 - but there is no additional speedup because the L2 cache is apparently able to sustain the required bandwidth. ↩

14. Although a `vector` can conceptually live inside its own heap storage, the same is not true of a pointer: a write through a pointer cannot modify the pointer. `vector` has an extra level of indirection which makes this inception-style stuff possible. ↩

15. It is show here with the indexing based approach `v[i]` but it works with the iterator-based approach as well. ↩

16. There are more extreme solutions out there, e.g., you can create an *opaque typedef*, which is basically just a `struct` wrapping a `uint8_t` value, combined with operator overloading and perhaps implicit conversions to make it act like a `uint8_t` while being a totally different type as far as the compiler is concerned. Surprisingly, while this works great in clang, it doesn't work at all gcc, producing the same code as `uint8_t`. Somewhere in the guts of its optimizer, `gcc` ends up treating this type as a character type along with all the aliasing implications. Another extreme solution would be to create your own `vector`-alike type, but whose pointer member(s) are declared `__restrict`. ↩

17. If anyone has a recommendation or a if anyone knows of a comments system that works with static sites, and which is not Disqus, has no ads, is free and fast, and lets me own the comment data (or at least export it in a reasonable format), I am all ears. ↩

Comments

Noam • March 5th, 2023 06:49

Thanks, learned a lot, how do you benchmark and how do you got cycles count ?

↪ Reply to Noam

Bajwa • August 3th, 2022 21:02

Really nice post! I've learned a lot from this post.

↪ Reply to Bajwa

---

victor yodaiken • August 22th, 2021 15:20

Nice explanation, but with needs a couple of caveats.

1. Explanations of the positive effect of TBAA always use the same loop example - which leads one to imagine there are no other clear cases.