The assume aligned attribute

Timur Doumler (papers@timur.audio)

Document #:	P0886R0
Date:	2018-02-12
Project:	Programming Language C++
Audience:	Evolution Working Group, Core Working Group

Abstract

This paper proposes to add an attribute `[[assume_aligned(alignment)]]` that can be applied to entities of pointer type. The attribute hints to the compiler that the value of the pointer is a memory address aligned to at least `alignment` bytes. This will allow the compiler to generate better-optimised code. Currently, several popular optimising compilers offer built-in `assume_aligned` intrinsics to achieve this, but each one requires a somewhat different syntax. The goal of this proposal is to standardise existing practice with a new standard attribute that provides the same functionality as the vendor-specific ones, but with a unified syntax that is easier to use and maximally consistent with the existing C++ language.

1 Motivation

1.1 Vectorisation

Many high-performance C++ programs today perform mathematical computations on contiguous chunks of numeric data. Often, this will be the most performance-critical part of an application. SIMD (single-instruction multiple-data) is a powerful technique to optimise such code, with new hardware architectures offering ever-increasing SIMD register sizes and instruction sets. Another piece of the puzzle are state-of-the-art optimising compilers which can auto-vectorise code and generate those instructions.

In such programs, the memory alignment of the data often plays a significant role. Consider:

```
void mult(float* x, int size, float factor)
{
    for (int i = 0; i < size; ++i)
        x[i] *= factor;
}
```

An optimising compiler on a platform supporting SIMD will auto-vectorise such a loop. However, the compiler cannot assume that `float* x` will be over-aligned to SIMD register size, even if we know that this is a static invariant in our program. The compiler will therefore either insert an extra branch with a loop prologue (to process the possibly present non-overaligned first elements of the array with single-data instructions), or load the data into SIMD registers using unaligned packed move instructions instead of aligned ones. If we had a way to tell the compiler that `float*`

`x` points to appropriately over-aligned memory, the compiler would be able to skip the prologue and use aligned packed move instructions, often resulting in a measurable performance improvement[1].

1.2 Low-latency file I/O

The possibility to hint to the compiler that a pointer points to over-aligned memory has other useful applications beyond vectorisation. For example, it is useful for low-latency file I/O to have pointers to data known to be aligned to the cache line size, the page size, the huge page size and so on. This way the algorithm can make the correct assumptions about data that can be accessed via DMA (direct memory access).

2 The problem

Consider a contiguous chunk of data over-aligned to 64 bytes:

```
float alignas(64) data[1024];
```

The alignment specification is not part of the type. Passing a pointer to this data to a function will prevent any optimisations based on the specified over-alignment.

In case the size of the data buffer is known at compile time, this is easy to fix by defining a wrapper type with an alignment specification in its class head:

```
template <typename T, std::size_t size, std::size_t alignment>
struct alignas(alignment) aligned_pack
{
   T data[size];
};
```

If we now use this wrapper type (instead of the raw pointer) to pass the data around, the compiler will know that the data is over-aligned as specified.

But what if the size of the data buffer is not known at compile time[2]? Consider:

```
float* data = get_next_overaligned_buffer();
```

For such a pointer to a dynamically-sized array, it is not possible in standard C++ to express the alignment specification of the data it points to. The compiler will always assume the natural alignment requirement of that type — in the case above, `alignof(float)` — because any such pointer value is valid.

We could try wrapping the data with some kind of templated, overalignment-aware "fancy pointer", container, array view, or span class. This is certainly a much better idea than passing around ranges via a raw pointer and a size (at least if you do not have to interface with C libraries). However, it does not get rid of the problem, but only moves it to another location in the code: inside any such class, you would inevitably have to hold a pointer to the data. Again, there is no way in standard C++ to turn the invariant of that pointer (namely, that its value is an over-aligned memory address) into a property that would be transparent to the compiler. Any contortions one might undertake to use the `alignas` specifier in this context are doomed to failure because in the case of a pointer to a dynamically-sized array, the alignment specification is not a property of its *type* (`float*`), but of its *value* (an over-aligned memory address), and may cease to be true after variable assignment (for

[1] The most recent Intel x86-64 chips do not have a performance penalty for unaligned vs. aligned memory access, but other architectures typically do. Furthermore, being able to skip the prologue/epilogue of a vectorised loop avoids unnecessary branches and reduces binary file size on all platforms including x86-64.

[2] This situation is typical, for example, for audio processing, where the data buffer size is determined at runtime. A stream of such dynamically sized data buffers is received on a high-priority thread callback, and the program has to perform computations on that data under real-time constraints. Similar scenarios occur in other domains of low-latency computing.

a more detailed explanation, see section 6.1). There is currently no way to express this property in C++, and this problem is not solvable via the type system. A new language feature is required.

3 Current practice: vendor-specific attributes

Several popular compiler vendors already offer an "assume aligned" attribute that exists specifically to solve the problem described above. Unfortunately, the syntax required for the attribute, and the places in the code where its usage is allowed, vary from vendor to vendor. An overview is given below.

3.1 GCC

[GCC] offers a built-in function

```
void* __builtin_assume_aligned(const void* ptr, size_t alignment);
```

This function returns its first argument, and allows the compiler to assume that the returned pointer is aligned with least `alignment` bytes. With this built-in function, the function `mult` from above can be optimised as follows:

```
void mult(float* x, int size, float factor)
{
    float* ax = (float*)__builtin_assume_aligned(x, 64);

    for (int i = 0; i < size; ++i)
        ax[i] *= factor;
}
```

where we told the compiler that `x` points to a range aligned to 64 bytes.

3.2 Clang

[Clang] offers the built-in function `__builtin_assume_aligned` from GCC as well as an attribute

```
__attribute__((assume_aligned(alignment)))
```

This attribute can only be applied to a function declaration and specifies that the return value of the function (which must be a pointer type) is an address aligned with at least `alignment` bytes.

3.3 Intel C++ Compiler

The Intel C++ compiler [ICC] offers a built-in compiler intrinsic

```
__assume_aligned(ptr, alignment);
```

which can be put anywhere inside a code block. It specifies that at this point in the code, the compiler can assume that the address given by the value of `ptr` is aligned with at least `alignment` bytes. With this intrinsic, the equivalent code would look like this:

```
void mult(float* x, int size, float factor)
{
    __assume_aligned(x, 64);

    for (int i = 0; i < size; ++i)
        x[i] *= factor;
}
```

3.4 Other compilers and languages

The MSVC compiler does not currently offer an assume aligned attribute. [OpenMP] offers `#pragma omp simd aligned`. [Fortran] offers a compiler directive `ASSUME_ALIGNED`.

4 Design goals

The main goal of this proposal is to standardise existing practice. We would like to offer the functionality of the vendor-specific built-ins described above through a standard `assume_aligned` attribute. This should have a familiar syntax that is easy to use and maximally consistent with the existing C++ language. It should further be a pure addition, not interfering with any parts of the existing language and not requiring modifications of the standard library.

The new attribute should adhere to the established principle for standard attributes: given a valid program, removal of the attribute from the code should not change its semantic meaning.

Another design goal is to make `assume_aligned` easily implementable. Compiler vendors have vast experience with attributes that arise through a static invariant and then propagate through function calls. It would be desirable to re-use as much as possible of that existing machinery.

`assume_aligned` is a hint to the compiler that it is free to generate more efficient code — it is not required to do so. On the other hand, `assume_aligned` should not cause any runtime overhead under normal circumstances. It only declares that the return value of a function or the argument value of a parameter is guaranteed to be a memory address over-aligned as specified. It does not cause the compiler to check[3] or enforce this guarantee — instead, it needs to be ensured by the program itself. If the compiler has generated SIMD instructions assuming over-alignment, but the actual address returned or passed in at runtime will not, in fact, be suitably over-aligned, most likely those instructions will crash the program — the behaviour is undefined, just like when dereferencing an invalid pointer.

The broader goal of this proposal is to allow for smart library classes that wrap such over-aligned memory buffers. This would make it possible to use the extra optimisations safely, enforce the over-alignment property at compile time, as well as overload or specialise on it (e.g. to select the optimal algorithm for the specified over-alignment at compile time). This can be accomplished through an alignment-aware dynamic container class, a span, or perhaps an array view. It should be clear from the previous discussion that such a class cannot be written without additional language support. This proposal aims to provide that language support.

5 Proposed syntax and semantics

5.1 Argument

The attribute `[[assume_aligned(...)]]` has one required argument which specifies the alignment requirement. Possible arguments are the same as for `alignas(...)`, either an integral constant expression that evaluates to an alignment value, or a type:

```
[[assume_aligned(64)]]                          // OK: integer literal
[[assume_aligned(4 * alignof(double))]]         // OK: integral constant expression
[[assume_aligned(boost::simd::pack<float>)]]    // OK: type

[[assume_aligned]]      // error: alignment missing
[[assume_aligned(0)]]   // error: invalid alignment value
```

5.2 Applying the attribute

The proposed syntax for applying the attribute `[[assume_aligned(...)]]` is the same as that of the existing attribute `[[carries_dependency]]`: it can apply to either a function *parameter* or a function *declaration*. In the first case, the alignment property is declared to hold for the argument value, and in the second case, for the return value of the function.

[3] An obvious exception are tools like undefined behaviour sanitisers, which should be allowed to assert on the alignment at runtime.

When applied to a function parameter, then, for example, the function `mult` from above could be written as:

```
void mult([[assume_aligned(64)]] float* x, int size, float factor)
{
    for (int i = 0; i < size; ++i)
        x[i] *= factor;
}
```

When applied to a function declaration, then, for example, a function that always returns a pointer aligned to L1 cacheline size might be declared as follows:

```
[[assume_aligned(std::hardware_destructive_interference_size)]] char* get_cacheline();
```

These semantics are identical to Clang's `__attribute__((assume_aligned(...)))`.

The attribute can only apply to entities of pointer type. Anything else will result in a compiler error:

```
[[assume_aligned(64)]] float foo();  // error: float is not a pointer type
```

5.3 Effect

The compiler is free to assume that when the attribute is applied to a function parameter or return value, the value passed in or returned will be an address aligned at least as strictly as specified by the attribute. The compiler can therefore, for example, skip runtime checks of the alignment, and directly insert instructions that give the optimal performance for the specified alignment. If this assumption does not hold at runtime, calling the function will result in undefined behaviour.

If `[[assume_aligned(...)]]` is used on a pointer of type `T*` specifying an extended alignment, and the current platform does not support the specified alignment for type `T`, this will also result in a compiler error, exactly like `alignas(...)` would.

On the other hand, if `assume_aligned` is used on a pointer of type `T*`, and `alignof(T)` is at least as strict as the alignment specified by `assume_aligned`, then the attribute is simply ignored, because the compiler already assumes an alignment at least as strict as the one specified by the attribute:

```
[[assume_aligned(char)]] double* foo();      // OK; attribute ignored
[[assume_aligned(double)]] double* bar();    // OK; attribute ignored
```

This is different from `alignas(char) double`, which would result in a compiler error. However it is consistent with the semantics of the existing vendor-specific built-ins.

6 Additional considerations

6.1 No application to variables and data members

It is not allowed to apply the attribute to a variable or data member declaration:

```
[[assume_aligned(64)]] float* x;   // error: attribute only applies to functions,
                                   // methods, and parameters
```

There are several reasons why we should not allow this syntax. Most importantly, it suggests that the over-alignment assumption would somehow be a property of the variable `x`. This is misleading, because it is not at all what is going on. If we assign a value to `x` in the following line of code, there is no alignment requirement associated with that new value. The same is true for a pointer that holds an over-aligned value at some point in time, but is incremented later. The value is then no longer over-aligned. The over-alignment assumption is not a property of objects or types — it is a property of a pointer value at some point in time. By allowing the attribute on functions and

parameters only, we make it clear that the over-alignment assumption holds only at the point in time when a function returns, or when a parameter is passed in.

The other reason to disallow applying `assume_aligned` to variable declarations is that the existing vendor-specific built-ins do not allow this either. Adding it would require more complex wording, increase the effort for implementers, and introduce semantics to the language for which there is currently no implementation experience.

It is important to note that there is no loss of functionality by disallowing the application of `assume_aligned` to variable declarations. Everything can be expressed by applying it to functions and parameters, and this will typically result in cleaner code. One of the intended usages of the attribute is to design library classes that manage over-aligned dynamic memory. The natural place to apply the attribute would be the public data access methods of such a class. As a motivational example to demonstrate how powerful this approach is, consider a class managing an over-aligned memory buffer, with the additional invariant that the size of that buffer is always a multiple of the alignment size. We would express this invariant in the class interface by applying the over-alignment assumption to both the `begin()` and the `end()` methods:

```
template <std::size_t alignment, typename T>
struct aligned_buffer
{
   [[assume_aligned(alignment)]] T* begin();
   [[assume_aligned(alignment)]] T* end();
private:
   T* data;    // no assume aligned attribute here!
};
```

Now, if we use `begin()` and `end()` in client code, for example in a range-based for loop, the compiler will see that both return over-aligned pointer values. When auto-vectorising the loop, it can then eliminate not only the prologue of the loop, but also the epilogue, and generate maximally efficient SIMD code.

There might still be cases where it is necessary to apply the over-alignment assumption to the value of a local variable at an arbitrary location in the code. Note that this can be easily achieved by defining the following helper function:

```
template <std::size_t alignment, typename T>
[[assume_aligned(alignment)]] constexpr T* assume_aligned(T* p)
{
   return p;
}
```

and using it, for example, like this:

```
float* alignedPtr = assume_aligned<64>(unalignedPtr);
```

This helper function has the same semantics as GCC's `__builtin_assume_aligned` (see 3.1). It is very useful in this context and could be considered for addition to the standard library in a future proposal.

6.2 No alignment offset (for now)

The `assume_aligned` built-ins offered by GCC and Clang can accept an extra argument of integer type, specifying an offset from the specified alignment that the compiler should assume. The offset is zero by default (if the extra argument is not provided). We decided to not include this additional feature in the initial version of this proposal, but it might be considered in a future version.

6.3 Not possible with contracts

It is worth asking whether the feature proposed here would alternatively be implementable with contracts, another feature currently proposed for the C++ language ([P0147R0], [P0542R2], and references therein). Conceptually, the over-alignment assumption expressed by `assume_aligned` can be seen as a contract. However, a problem immediately arises with implementing such a contract: how would we formulate the condition "the value of `x` is a memory address that meets the alignment requirement specified by `y`" in code? In practice, for *some* compilers on *some* platforms, something like

```
reinterpret_cast<std::uintptr_t>(x) % y == 0
```

could perhaps work. However, there are several problems with this. First, the above code is not portable, defined behaviour. The C++ standard does not guarantee in any way that the above expression (or in fact, *any* expression) is equivalent to the condition "the value of `x` is a memory address that meets the alignment requirement specified by `y`". Therefore, the feature is not implementable with contracts in their current form. But even if it were, a second problem arises: it is unclear whether compilers would be able to understand such expressions and correctly deduce the optimisation opportunities that follow from them — currently they cannot. This is made even worse by the fact that there are several different, equivalent ways to write such an expression. If we went down that route, we would then still need to add a new language feature to be able to correctly, portably write down the condition "the value of `x` is a memory address that meets the alignment requirement specified by `y`" — and then we would need contracts to work well with it. Rather than exploring this route and all its complications, it seems much more straightforward to add the feature in the way it is proposed here.

Again, it is worth pointing to existing practice. Both ICC and Clang have two separate "assume" built-ins: `__assume_aligned` for the over-alignment assumption, and `__assume` for assumptions that can be expressed through C++ expressions. The latter could be absorbed by contracts (in case they get standardised in the future), while the former could not.

7 Proposed wording

The proposed changes are relative to the C++ working paper [Smith2017].

Add to 10.6. Attributes [dcl.attr]:

10.6.9 Assume aligned attribute [dcl.attr.aligned]

1 The *attribute-token* `assume_aligned` can be used to mark an entity of pointer type to hint to the compiler that its value is a memory address aligned with a given extended alignment value. It shall appear at most once in each *attribute-list*. An *attribute-argument-clause* shall be present and have one of the following two forms:

`(` *constant-expression* `)`
`(` *type-id* `)`

2 When the *attribute-argument-clause* is of the form `(` *constant-expression* `)`, the *constant-expression* shall be an integral constant expression. The value of this expression shall specify an alignment value (6.6.5).

3 An *attribute-argument-clause* of the form `(` *type-id* `)` has the same effect as `(` `alignof(`*typeid* `))`.

4 The `assume_aligned` attribute may be applied to the *declarator-id* of a *parameter-declaration* in a function declaration or lambda, in which case it specifies that when the function is called, the argument value will be a memory address aligned at least as strictly as specified by the attribute. If the type of the parameter is not a pointer type, the program is ill-formed.

[5] The `assume_aligned` attribute may also be applied to the *declarator-id* of a function declaration, in which case it specifies that the return value of the function will be a memory address aligned at least as strictly as specified by the attribute. If there is no such return value, or the return type is not a pointer type, the program is ill-formed.

[6] The first declaration of a function shall specify the `assume_aligned` attribute for its *declarator-id* if any declaration of the function specifies the `assume_aligned` attribute, and all declarations of the function that specify the `assume_aligned` attribute shall specify the same alignment value in its argument. Furthermore, the first declaration of a function shall specify the `assume_aligned` attribute for a parameter if any declaration of that function specifies the `assume_aligned` attribute for that parameter, and all declarations of the function that specify the `assume_aligned` attribute for that parameter shall specify the same alignment value in its argument. If a function or one of its parameters is declared with the `assume_aligned` attribute in its first declaration in one translation unit and the same function or one of its parameters is declared without the `assume_aligned` attribute or with an `assume_aligned` attribute whose argument specifies a different alignment value in its first declaration in another translation unit, the program is ill-formed, no diagnostic required.

[7] If the `assume_aligned` attribute is applied to an entity of pointer type `T` in one of the contexts described above, and the alignment requirement specified by the attribute evaluates to an extended alignment, and the implementation does not support that alignment for the type pointed to by `T`, the program is ill-formed. If the alignment requirement specified by the argument of the `assume_aligned` attribute evaluates to an alignment that is not stricter than the alignment requirement of the type pointed to by `T`, the *attribute-argument-clause* has no effect. Otherwise, if the declaration of a function specifies the `assume_aligned` attribute for its *declarator-id* and the value returned is not a memory address that meets the alignment requirement as specified by the argument of the attribute, or if the declaration of a function specifies the `assume_aligned` attribute for a parameter and the argument value passed for that parameter is not a memory address that meets the alignment requirement as specified by the argument of the attribute, the effect of calling the function is undefined.

[*Note:* The extended alignment assumption expressed by the `assume_aligned` attribute may result in generation of more efficient code. The attribute merely provides information to the compiler about this assumption. It is up to the program to ensure that the assumption actually holds. The attribute does not cause the compiler to verify or enforce this. — *end note*]

[*Example:*

```
// Function parameters with extended alignment assumption:

void add_64aligned([[assume_aligned(64)]] float* x,
                   [[assume_aligned(64)]] float* y, int size)
{
    for (int i = 0; i < size; ++i)
        x[i] += y[i];
}

// Function return value with extended alignment assumption:

[[assume_aligned(std::hardware_destructive_interference_size)]] char* get_cacheline();
```

— *end example*]

Acknowledgements

Huge thanks to Fabian Renn-Giles for providing the idea and motivation for this paper. Also, many thanks to Gašper Ažman, Arthur O'Dwyer, Ville Voutilainen, Richard Smith, Niall Douglas, and Mathias Gaunard for their very helpful comments during the writing of the paper.

References

[Clang] Attributes in clang – clang 6 documentation. https://clang.llvm.org/docs/AttributeReference.html, 2017 (accessed 2017-12-03).

[Fortran] Intel Fortran Compiler 17.0 Developer and Reference Guide: ASSUME_ALIGNED. https://software.intel.com/en-us/node/679017, 2018 (accessed 2018-01-11).

[GCC] Other built-in functions provided by gcc. https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html, 2017 (accessed 2017-12-03).

[ICC] Data alignment to assist vectorization. https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization, 2015 (accessed 2017-12-03).

[OpenMP] User and Reference Guide for the Intel C++ Compiler 15.0: omp simd. https://software.intel.com/en-us/node/524530, 2018 (accessed 2018-01-11).

[P0147R0] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. A Contract Design. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0380r1.pdf, 2016 (accessed 2018-01-11).

[P0542R2] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. Support for contract based programming in C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0542r0.html, 2017 (accessed 2018-01-11).

[Smith2017] Richard Smith. Working Draft, Standard for Programming Language C++. https://github.com/cplusplus/draft, 2017 (accessed 2017-12-07).