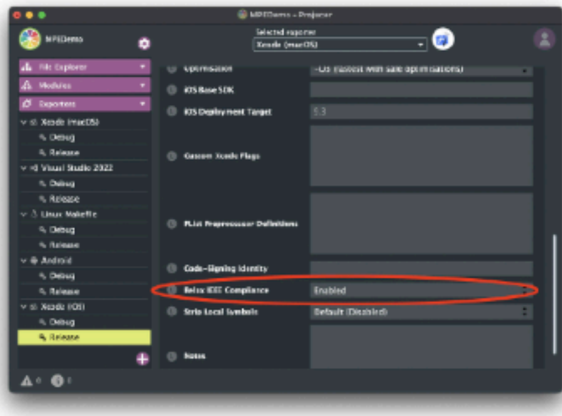
fr810 1 7d

Author of `SIMDRegister` here: I'd just like to add that before embarking on `SIMDRegister`, check that your compiler isn't already auto-vectorizing the tight loops in your code. In my experience, the compiler does a good job auto-vectorizing even moderately complex loops (but see my note at the end of this post).

To ensure that the compiler is allowed to even auto-vectorize your code, be sure that:

1. You are building in release mode (i.e. at least optimisation level `-O3`)
2. You have "Relax IEEE compliance" enabled in the Projucer. This is absolutely crucial, especially on arm/arm64, as SIMD instructions do not have the same denormal/round-to-zero (arm) and/or multiply-accumulate rounding (x86/arm) behaviour as normal IEEE compliant floating point instructions. Hence, the compiler is not allowed to replace your loops with SIMD instructions if it needs to ensure IEEE compliance.

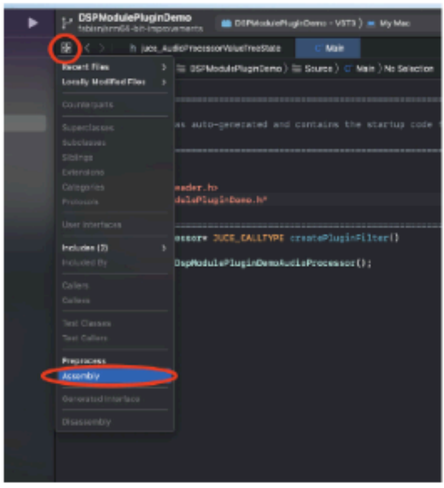


Note if you are compiling with `-Ofast` then "Relax IEEE compliance" will be enabled regardless of the Projucer setting.

To check if your code is being auto-vectorized, you need to look at the assembly. I recommend doing this, even when using `SIMDRegister`, as you will often have unexpected results.

I highly recommend `godbolt.org` for this: `godbolt.org` has the advantage that you can change compiler, compiler version and compiler options on the fly and it neatly colour highlights which parts of your source code correspond to which assembly.

If your project is too big to fit into godbolt then you can also use Xcode's Assembly view:



Unfortunately, it doesn't do any colour highliting. **Update: Also, see post below on strange behaviour when trying to view assembly in recent Xcode versions.**

What you will often see, is that the compiler will generate two versions of your code: one uses SIMD and the other doesn't. This is because the compiler does not know if your audio buffer pointers are SIMD-aligned. Hence, the compiler creates a non-SIMD "pre-amble" until the buffer pointers are SIMD-aligned, the core of the algorithm (which uses SIMD) and then a non-SIMD epilogue to finish up any remaining elements that didn't fit into a full SIMD register.

If your loop involves multiple buffer pointers then, depending on your exact algorithm, the compiler may need to create multiple versions of pre-ambles/epilogues (i.e. only the first buffer is SIMD aligned, but the second isn't etc.). If this gets too complicated, the compiler will give up and just do two completely separate versions of your code (one with SIMD and the other without).

To avoid this, you can use C++20's `std::assumed_aligned` (earlier compiler versions may have `__builtin_assume_aligned`) to tell the compiler that you know that the buffer pointers will be SIMD aligned. This is recommended even for simple loops as it will avoid at least one conditional (to check for alignment) and it has the extra benefit of reducing your code-size by getting rid of the pre-ambles/epilogues.

As mentioned, the compiler is pretty good at auto-vectorizing even moderately complex loops. However, here are a few examples, where I have seen the compiler fail at auto-vectorizing:

1. The compiler needs to follow the "as-if" rule when optimizing, i.e. it may transform your code into something completely different (i.e. by re-ordering loops, for example) but the outcome of your program must be the same "as-if" the compiler did no optimizations (unless you've written undefined behaviour).

   This means, however, that certain transformations are off-limits to the compiler - for example, heap memory allocations - as requiring to allocate memory is a different outcome of your program than not allocating memory.

   For example, a compiler is not allowed to *auto-vectorize* a direct convolution. This is because, as you move through your array, most of the time, the kernel and the array will not be SIMD aligned with respect to each other. However, by storing N copies of your kernel (where N is the number of elements in a SIMD vector) with each copy being shifted by one element, you can now always find a kernel that is SIMD aligned with your input vector. However, even a hypothetical god compiler would not be allowed to do this transformation as the compiler would need to allocate memory for the extra copies of your kernel. Here, you would need to program the convolution by hand using `SIMDRegister`.
2. Conditionals in your loop: as long as you only have simple ternary like statements, the compiler will replace those statements with SIMD masking tricks (i.e. no branching). Funnily, I've often seen MSVC use SIMD masking code with a ternary statement (i.e. using the `?` character), and not use it when writing the same code with an `if` statement. If the conditional gets even slightly too complex, compilers will usually use traditional branching (and thus sometimes inhibiting the use of auto-vectorization) as it's hard for a compiler to reason about the trade-off of having the need to compute both result values when selecting via bit-mask vs. computing only the required result via branching as the performance of latter highly depends on how often the condition is true or not (becuase of branch-predictors). Hence, if you know of a way to convert your conditional into a statement with bitmasks and no branch, and the compiler isn't doing it for you, then you will likely need to write it manually via `SIMDRegister`.
3. Using complex templating and template meta programming (like expression templates) somehow confuses the compiler. I'm not entirely sure why, but I've seen this a lot.

2 Replies 12 Reply

Credit to Fabian:

https://forum.juce.com/t/simdregister-is-it-worth-it/53362/4

- Build with at least –03

- Relax IEEE compliance (`-ffast-math`)

- -Ofast

- Optionally help compiler with

- std::assumed_aligned

46

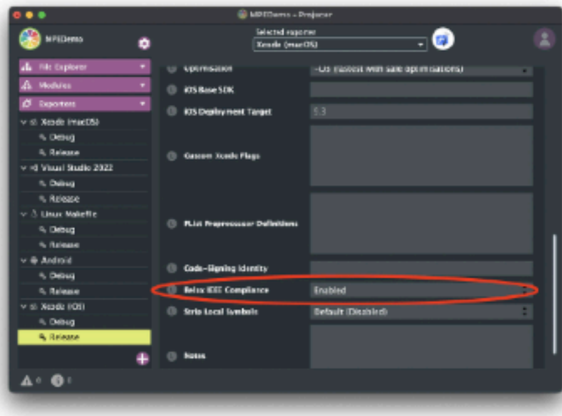
fr810

1 7d

Author of `SIMDRegister` here: I'd just like to add that before embarking on `SIMDRegister`, check that your compiler isn't already auto-vectorizing the tight loops in your code. In my experience, the compiler does a good job auto-vectorizing even moderately complex loops (but see my note at the end of this post).

To ensure that the compiler is allowed to even auto-vectorize your code, be sure that:

1. You are building in release mode (i.e. at least optimisation level `-O3`)
2. You have "Relax IEEE compliance" enabled in the Projucer. This is absolutely crucial, especially on arm/arm64, as SIMD instructions do not have the same denormal/round-to-zero (arm) and/or multiply-accumulate rounding (x86/arm) behaviour as normal IEEE compliant floating point instructions. Hence, the compiler is not allowed to replace your loops with SIMD instructions if it needs to ensure IEEE compliance.

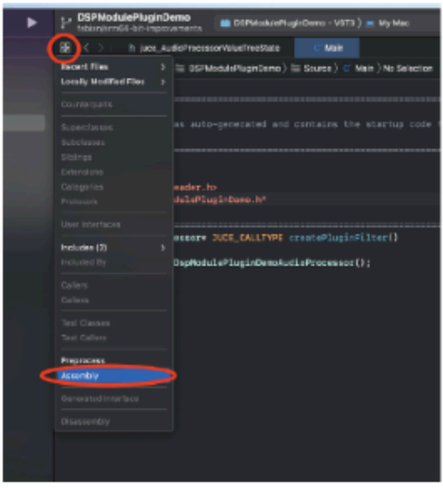


Note if you are compiling with `-Ofast` then "Relax IEEE compliance" will be enabled regardless of the Projucer setting.

To check if your code is being auto-vectorized, you need to look at the assembly. I recommend doing this, even when using `SIMDRegister`, as you will often have unexpected results.

I highly recommend `godbolt.org` for this: `godbolt.org` has the advantage that you can change compiler, compiler version and compiler options on the fly and it neatly colour highlights which parts of your source code correspond to which assembly.

If your project is too big to fit into godbolt then you can also use Xcode's Assembly view:



Unfortunately, it doesn't do any colour highliting. **Update: Also, see post below on strange behaviour when trying to view assembly in recent Xcode versions.**

What you will often see, is that the compiler will generate two versions of your code: one uses SIMD and the other doesn't. This is because the compiler does not know if your audio buffer pointers are SIMD-aligned. Hence, the compiler creates a non-SIMD "pre-amble" until the buffer pointers are SIMD-aligned, the core of the algorithm (which uses SIMD) and then a non-SIMD epilogue to finish up any remaining elements that didn't fit into a full SIMD register.

If your loop involves multiple buffer pointers then, depending on your exact algorithm, the compiler may need to create multiple versions of pre-ambles/epilogues (i.e. only the first buffer is SIMD aligned, but the second isn't etc.). If this gets too complicated, the compiler will give up and just do two completely separate versions of your code (one with SIMD and the other without).

To avoid this, you can use C++20's `std::assumed_aligned` (earlier compiler versions may have `__builtin_assume_aligned`) to tell the compiler that you know that the buffer pointers will be SIMD aligned. This is recommended even for simple loops as it will avoid at least one conditional (to check for alignment) and it has the extra benefit of reducing your code-size by getting rid of the pre-ambles/epilogues.

As mentioned, the compiler is pretty good at auto-vectorizing even moderately complex loops. However, here are a few examples, where I have seen the compiler fail at auto-vectorizing:

1. The compiler needs to follow the "as-if" rule when optimizing, i.e. it may transform your code into something completely different (i.e. by re-ordering loops, for example) but the outcome of your program must be the same "as-if" the compiler did no optimizations (unless you've written undefined behaviour).

   This means, however, that certain transformations are off-limits to the compiler - for example, heap memory allocations - as requiring to allocate memory is a different outcome of your program than not allocating memory.

   For example, a compiler is not allowed to *auto-vectorize* a direct convolution. This is because, as you move through your array, most of the time, the kernel and the array will not be SIMD aligned with respect to each other. However, by storing N copies of your kernel (where N is the number of elements in a SIMD vector) with each copy being shifted by one element, you can now always find a kernel that is SIMD aligned with your input vector. However, even a hypothetical god compiler would not be allowed to do this transformation as the compiler would need to allocate memory for the extra copies of your kernel. Here, you would need to program the convolution by hand using `SIMDRegister`.
2. Conditionals in your loop: as long as you only have simple ternary like statements, the compiler will replace those statements with SIMD masking tricks (i.e. no branching). Funnily, I've often seen MSVC use SIMD masking code with a ternary statement (i.e. using the `?` character), and not use it when writing the same code with an `if` statement. If the conditional gets even slightly too complex, compilers will usually use traditional branching (and thus sometimes inhibiting the use of auto-vectorization) as it's hard for a compiler to reason about the trade-off of having the need to compute both result values when selecting via bit-mask vs. computing only the required result via branching as the performance of latter highly depends on how often the condition is true or not (becuase of branch-predictors). Hence, if you know of a way to convert your conditional into a statement with bitmasks and no branch, and the compiler isn't doing it for you, then you will likely need to write it manually via `SIMDRegister`.
3. Using complex templating and template meta programming (like expression templates) somehow confuses the compiler. I'm not entirely sure why, but I've seen this a lot.

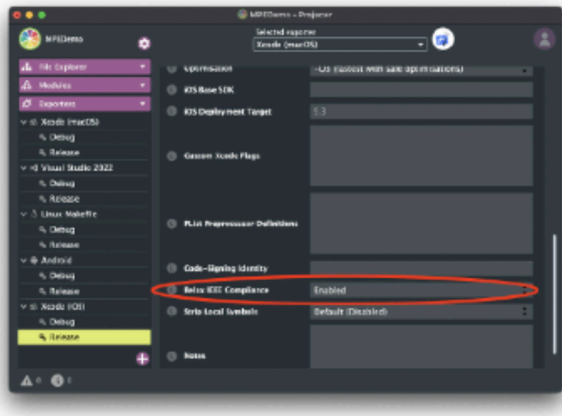
fr810

1 7d

Author of `SIMDRegister` here: I'd just like to add that before embarking on `SIMDRegister`, check that your compiler isn't already auto-vectorizing the tight loops in your code. In my experience, the compiler does a good job auto-vectorizing even moderately complex loops (but see my note at the end of this post).

To ensure that the compiler is allowed to even auto-vectorize your code, be sure that:

1. You are building in release mode (i.e. at least optimisation level `-O3`)
2. You have "Relax IEEE compliance" enabled in the Projucer. This is absolutely crucial, especially on arm/arm64, as SIMD instructions do not have the same denormal/round-to-zero (arm) and/or multiply-accumulate rounding (x86/arm) behaviour as normal IEEE compliant floating point instructions. Hence, the compiler is not allowed to replace your loops with SIMD instructions if it needs to ensure IEEE compliance.

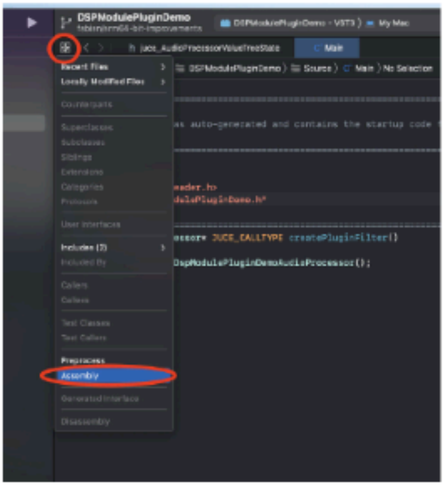


Note if you are compiling with `-Ofast` then "Relax IEEE compliance" will be enabled regardless of the Projucer setting.

To check if your code is being auto-vectorized, you need to look at the assembly. I recommend doing this, even when using `SIMDRegister`, as you will often have unexpected results.

I highly recommend `godbolt.org` for this: `godbolt.org` has the advantage that you can change compiler, compiler version and compiler options on the fly and it neatly colour highlights which parts of your source code correspond to which assembly.

If your project is too big to fit into godbolt then you can also use Xcode's Assembly view:



Unfortunately, it doesn't do any colour highliting. **Update: Also, see post below on strange behaviour when trying to view assembly in recent Xcode versions.**

What you will often see, is that the compiler will generate two versions of your code: one uses SIMD and the other doesn't. This is because the compiler does not know if your audio buffer pointers are SIMD-aligned. Hence, the compiler creates a non-SIMD "pre-amble" until the buffer pointers are SIMD-aligned, the core of the algorithm (which uses SIMD) and then a non-SIMD epilogue to finish up any remaining elements that didn't fit into a full SIMD register.

If your loop involves multiple buffer pointers then, depending on your exact algorithm, the compiler may need to create multiple versions of pre-ambles/epilogues (i.e. only the first buffer is SIMD aligned, but the second isn't etc.). If this gets too complicated, the compiler will give up and just do two completely separate versions of your code (one with SIMD and the other without).

To avoid this, you can use C++20's `std::assumed_aligned` (earlier compiler versions may have `__builtin_assume_aligned`) to tell the compiler that you know that the buffer pointers will be SIMD aligned. This is recommended even for simple loops as it will avoid at least one conditional (to check for alignment) and it has the extra benefit of reducing your code-size by getting rid of the pre-ambles/epilogues.

As mentioned, the compiler is pretty good at auto-vectorizing even moderately complex loops. However, here are a few examples, where I have seen the compiler fail at auto-vectorizing:

1. The compiler needs to follow the "as-if" rule when optimizing, i.e. it may transform your code into something completely different (i.e. by re-ordering loops, for example) but the outcome of your program must be the same "as-if" the compiler did no optimizations (unless you've written undefined behaviour).

   This means, however, that certain transformations are off-limits to the compiler - for example, heap memory allocations - as requiring to allocate memory is a different outcome of your program than not allocating memory.

   For example, a compiler is not allowed to *auto-vectorize* a direct convolution. This is because, as you move through your array, most of the time, the kernel and the array will not be SIMD aligned with respect to each other. However, by storing N copies of your kernel (where N is the number of elements in a SIMD vector) with each copy being shifted by one element, you can now always find a kernel that is SIMD aligned with your input vector. However, even a hypothetical god compiler would not be allowed to do this transformation as the compiler would need to allocate memory for the extra copies of your kernel. Here, you would need to program the convolution by hand using `SIMDRegister`.
2. Conditionals in your loop: as long as you only have simple ternary like statements, the compiler will replace those statements with SIMD masking tricks (i.e. no branching). Funnily, I've often seen MSVC use SIMD masking code with a ternary statement (i.e. using the `?` character), and not use it when writing the same code with an `if` statement. If the conditional gets even slightly too complex, compilers will usually use traditional branching (and thus sometimes inhibiting the use of auto-vectorization) as it's hard for a compiler to reason about the trade-off of having the need to compute both result values when selecting via bit-mask vs. computing only the required result via branching as the performance of latter highly depends on how often the condition is true or not (becuase of branch-predictors). Hence, if you know of a way to convert your conditional into a statement with bitmasks and no branch, and the compiler isn't doing it for you, then you will likely need to write it manually via `SIMDRegister`.
3. Using complex templating and template meta programming (like expression templates) somehow confuses the compiler. I'm not entirely sure why, but I've seen this a lot.

Credit to Fabian:

https://forum.juce.com/t/simdregister-is-it-worth-it/53362/4

- Build with at least –03
- Relax IEEE compliance (`–ffast–math`)
  - `–Ofast`
- Optionally help compiler with
  - `std::assumed_aligned`

but the second isn't etc.). If this gets too complicated, the compiler will give up and just do two completely separate versions of your code (one with SIMD and the other without).

To avoid this, you can use C++20's `std::assumed_aligned` (earlier compiler versions may have `__builtin_assume_aligned`) to tell the compiler that you know that the buffer pointers will be SIMD aligned. This is recommended even for simple loops as it will avoid at least one conditional (to check for alignment) and it has the extra benefit of reducing your code-size by getting rid of the pre-ambles/epilogues.

As mentioned, the compiler is pretty good at auto-vectorizing even moderately complex loops. However, here are a few examples, where I have seen the compiler fail at auto-vectorizing:

1. The compiler needs to follow the "as-if" rule when optimizing, i.e. it may transform your code into something completely different (i.e. by re-ordering loops, for example) but the outcome of your program must be the same "as-if" the compiler did no optimizations (unless you've written undefined behaviour).

   This means, however, that certain transformations are off-limits to the compiler - for example, heap memory allocations - as requiring to allocate memory is a different outcome of your program than not allocating memory.

   For example, a compiler is not allowed to *auto-vectorize* a direct convolution. This is because, as you move through your array, most of the time, the kernel and the array will not be SIMD aligned with respect to each other. However, by storing N copies of your kernel (where N is the number of elements in a SIMD vector) with each copy being shifted by one element, you can now always find a kernel that is SIMD aligned with your input vector. However, even a hypothetical god compiler would not be allowed to do this transformation as the compiler would need to allocate memory for the extra copies of your kernel. Here, you would need to program the convolution by hand using `SIMDRegister`.
2. Conditionals in your loop: as long as you only have simple ternary like statements, the compiler will replace those statements with SIMD masking tricks (i.e. no branching). Funnily, I've often seen MSVC use SIMD masking code with a ternary statement (i.e. using the `?` character), and not use it when writing the same code with an `if` statement. If the conditional gets even slightly too complex, compilers will usually use traditional branching (and thus sometimes inhibiting the use of auto-vectorization) as it's hard for a compiler to reason about the trade-off of having the need to compute both result values when selecting via bit-mask vs. computing only the required result via branching as the performance of latter highly depends on how often the condition is true or not (becuase of branch-predictors). Hence, if you know of a way to convert your conditional into a statement with bitmasks and no branch, and the compiler isn't doing it for you, then you will likely need to write it manually via `SIMDRegister`.
3. Using complex templating and template meta programming (like expression templates) somehow confuses the compiler. I'm not entirely sure why, but I've seen this a lot.

2 Replies 12 Reply

```cpp
#include <iostream>
#include <numeric>
#include <cmath>

int main()
{
  double a = std::numeric_limits<double>::infinity();

  if (std::isinf(a))
    std::cout << "Inf detected.\n";
  else
    std::cout << "Inf NOT detected.\n";

  if (std::isfinite(a))
    std::cout << "Is finite.\n";
  else
    std::cout << "Is NOT finite.\n";

  return 0;
}
```

https://godbolt.org/z/Y56o6ffE4