JANUARY 2, 2018JANUARY 3, 2018 ⁄ JEFFAMSTUTZ

Building a C++ SIMD Abstraction (3/N) – A Tour of 'tsimd'

In this post I want to introduce the library I've been working on for a couple of months now: **tsimd (https://github.com/ospray/tsimd)**. My last post looked at what is out there for C++ developers to vectorize code, motivating the need for tsimd. You don't have to agree with everything I say (it's just an opinion!), but have a look at it anyway if you disagree and haven't read it yet (found **here (https://jeffamstutz.io/2017/12/14/building-a-c-simd-abstraction-2-n-status-quo-my-perspective/)**).

Without further ado, let's do a high-level tour of tsimd. By simply including "tsimd/tsimd.h", you get the following as a header-only library:

The main SIMD register wrapper, tsimd::pack<T, W>

Everything in tsimd centers around a single template class which represents a collection of *T* values of width *W*. Because this is intended to abstract only SIMD instructions, valid types for *T* are only as follows:

- integral types: *int* and *long long*
- floating point types: *float* and *double*
- boolean types: *tsimd::bool32_t* and *tsimd::bool64_t*

Using those element types, you can instantiate pack<> using widths of 1, 4, 8, and 16. This is biased towards x86 instructions, but could certainly be expanded to other CPU ISAs very easily. So, for instance, a 'pack<float, 8>' would map logically to an AVX '__m256' (more on this later).

Because the combination of <T, W> pairs for pack<> are well defined, tsimd provides a complete set of type aliases to remove the visual noise generated by lots of angle brackets. These shortcuts are defined in "**pack.h (https://github.com/ospray/tsimd/blob/master/tsimd/detail/pack.h#L165)**", where they define things like:

```
using namespace tsimd;
// specify a particular width
using vfloat8 = pack<float, 8>;

// create a 'varying' declaration where width is set by ISA compile flags
using vfloat = pack<float, TSIMD_DEFAULT_WIDTH>; // 'vfloat8' with AVX/AVX2
```

This gives you the ability to code to a particular width, or define some SIMD kernel that adjusts to a particular native width that changes with ISA compiler flags. Note that you can always query the compile-time width of a pack with the 'pack<>::static_size' member.

If you haven't noticed, the list of above types fall into 2 categories: 32-bit and 64-bit elements (this may be expanded later to support 8-bit and 16-bit types). This then differentiates what vbool types are used by various operators and functions to ensure that masks match the size of the pack<> used. An example:

```
using namespace tsimd;
vfloat8 a = 1.f, b = 3.f;

// All elements in 'comparison' would be 'true'
vboolf8 comparison = a < b;
```

The boolean types 'tsimd::bool32_t' and 'tsimd::bool64_t' exist to ensure that the underlying bit representation of a vbool element are both correctly sized and have the correct value for 'true' and 'false'. For example, elements in an AVX 'vboolf8' need to be either 0x0 (false) or 0xFFFFFFFF (true). As a user, simply treat 'bool32_t' and 'bool64_t' as a normal 'bool' and you'll be just fine!

Basic operators

There are 3 categories of operators that tsimd provides for pack<>: arithmetic, logical, and bitwise. The idea is to provide all operators that are defined on valid elements of pack<>. Thus if an operator is defined for 'float' or 'int', then it will defined for 'vfloat' or 'vint'. Here are some brief examples:

```
using namespace tsimd;
vint a = //...
vint b = //...

// arithmetic
vint c = a + b + 2;
a += b;
b += 1;

// logical
vboolf l = (a == b);
l = !l;

// bitwise
vint ls = (a << 4);
ls = a ^ b;
```

Something to keep in mind is that, where it makes sense, all operators are defined to also work with pack<> and scalar types (or "uniform" values). This means that scalar values will be promoted to pack<> when a pack<> is present in an expression without the need to explicitly convert scalar values to pack<> (though you certainly can, if you want to).

Per-element access

Another key feature of pack<> is random access to pack<> elements via operator (). This makes it easy to implement intra-register functions. While there are a number of intra-register functions already provided in the tsimd library, this makes it simple to implement your own. Here's a simple (silly) example:

```
using namespace tsimd;
void print_values(const vfloat &v)
{
  for (int i = 0; i < vfloat::static_size; ++i)
    std::cout << v[i] << ' ';
  std::cout << std::endl;
}
```

Another feature of pack<> is iteration via begin() and end(). This enables STL-style iteration and composibility with existing algorithms that operates over containers. While it should be noted that not all STL algorithms will peform well, it is at least supported and the style is preserved so SIMD specific algorithms can be written in the same way. Also remember that iterators enable the use of range-based for-loops. Thus you can also write:

```
using namespace tsimd;
void print_values(const vfloat &v)
{
  for (float value : v)
    std::cout << value << ' ';
  std::cout << std::endl;
}
```

Register size emulation

If you specify a pack<> type that isn't native to the ISA being compiled, then tsimd will automatically try to select the best alternative implementation available. For example, if I am compiling for AVX and I have some 16-wide specific kernel (i.e. using vfloat16 or vint16), then the implementation will fall back to 2x 8-wide operations on the 16 values in the pack<>. This keeps portability reasonable for algorithms that may be tightly coupled to the SIMD register size. Furthermore, depending on the algorithm, it may be more efficient to do double pumping of a larger ISA…tsimd leaves it up to you to choose what's best!

I will note, however, that compiler optimizers seem to be less happy when emulation kicks in, so I wouldn't rely on it *too* much. For example, the mandelbrot example found in the repo only emulates well with clang: gcc, icc, and MSVC don't seem to optimize emulated operations quite as well. I hope this improves over time, though.

Composing with native SIMD intrinsics

The last item I want to highlight in this section is how pack<> composes well with existing hand-coded kernels written with intrinsics. This is enabled through construction and conversion operators to/from intrinsic data types. The decision to enable this was informed by feedback that I received early on that many users wanted the option to progressively convert their kernels to tsimd without it being "all-in or nothing".

Because pack<> will implicitly convert to/from intrinsic types (if the right compiler flags are specified), existing hand-coded kernels can be integrated with tsimd types on a line-by-line basis. Here's a contrived example:

```
// assume this is in an existing hand-coded AVX kernel
__m256 a = //...

// ...some hand-coded kernel, blah, blah...

tsimd::vfloat8 other_value(a);

// ...some section of tsimd code in this kernel...

__m256 b = _mm256_sqrt_ps(other_value);// back to intrinsics

// ...
```

Library functions

Now that I hope you have a rough understanding of pack<>, tsimd also comes with a number of functions which operate with pack<> types. These functions currently come in 4 categores: algorithm, math, memory, and random. The collection of provided library functions is likely to grow over time, with the possibility of adding additional categories later.

Algorithms

Functions in the ‘algorithm’ category are generic functions that are specific to operation on pack<> which do not fit the other 3 categories. There are 2 that I want to specifically address: select() and mask reductions.

The select() function takes two pack<> values and returns a per-lane selection of each input value based on a provided vbool mask (i.e. a ‘blend’ operation). Here’s an example:

```cpp
using namespace tsimd;
vint4 a = -1;
vint4 b;
std::iota(b.begin(), b.end(), 0);

// a == { -1, -1, -1, -1 };
// b == { 0, 1, 2, 3 };

vint4 result = select(b >= 2, b, a);

// result == { -1, -1, 2, 3 };
```

This is the core of how control flow works with SIMD: typically you would execute any or all of the branches in an if/else chain and use select() calls to piece together the final value. Of course you *can* optimize whether you skip entire branches based on detecting special input conditions, but that is up to you!

The other functions I want to highlight are mask reductions: all(), any(), and none(). These are useful to test whether a branch needs to be considered at all. Here are some simple examples:

```cpp
using namespace tsimd;
vint4 a = -1;
vint4 b;
std::iota(b.begin(), b.end(), 0);

// a == { -1, -1, -1, -1 };
// b == { 0, 1, 2, 3 };

assert(all(a == -1));
assert(any(b > 2));
assert(none(a > 0));
```

Have a look at the mandelbrot example (**here (https://github.com/ospray/tsimd/blob/master/examples/mandelbrot/mandelbrot.cpp#L82)**) to see both none() and select() in action.

Math

Math functions are as straightforward as the ones found in the system <cmath> header, so I won't go into any detail here. Currently the following functions are implemented (with *many many* more to come):

- abs()
- ceil()
- cos()
- exp()
- floor()
- log()
- max()
- min()
- pow()
- sin()
- sqrt()
- tan()

Memory

Memory functions are about loading and storing values to/from arrays. There are 2 types of this: coherent (load/store) and incoherent (gather/scatter). These functions are mostly useful when you want to implement a "load-compute-store" pattern. Here's an example of load():

```cpp
std::vector<float> myVector = //...assume 'myVector' contains lots of values

vfloat8 v = load<vfloat8>(myVector.data());
// 'v' contains the first 8 values from 'myVector'
```

…and store() looks like:

```cpp
std::vector<float> myVector = //...same as before

vfloat8 v = 1.f;
store(v, myVector.data());
// The first 8 values of 'myVector' are now 1.f
```

gather()/scatter() operate like load()/store(), but you provide a vint of offsets where each lane is written into the destination using the corresponding offset value. These are less efficient, but are often necessary when implementing various vectorized algorithms.

Finally, all memory functions also have a masked version. This allows users to specify a mask which will do load()/store()/gather()/scatter() on a subset of values in a pack. Thus a masked store() would only write values for the lanes which are 'true' in the input mask.

Random

While it is *very* young right now, tsimd also can/will provide random number generators in the same style as <random> in the STL. Currently only 'uniform_real_distribution' and a precomputed Halton sequence generator are implemented, but I see no reason that the entire set of features found in the STL <random> header couldn't be implemented with tsimd types. Here's a simple example of what I've got thus far:

```
using tsimd::vfloat8;

tsimd::default_halton_engine2<8> vrng;
tsimd::uniform_real_distribution<vfloat8> vdist(0.f, 1.f);

vfloat8 random_numbers = vdist(vrng);
// each value in 'random_numbers' is something between 0.f and 1.f
```

Conclusion

I hope tsimd looks appealing to those of you who want a solution that lets you have tight control over your SIMD code while allowing you to write it independant of specifc ISA register sizes. Furthermore, operator overloading helps keep syntax looking as natural as possible. There's plenty that still needs to be worked on, but I think there's enough implemented now to do useful work.

The next couple of posts I will look at some implementation details of tsimd that make it easier to keep some of the complexity tamed. Specifically, the use of some custom type traits and simple use of SFINAE reduce the amount of code duplication required. I think type traits and SFINAE can be a bit scary for some C++ developers, so I aim to provide both a real-world example of why you should consider them (when appropriate), and how they can be used in a way that reduces (not increases!) complexity in your template code.

I hope you try out tsimd sometime…please let me know how it goes and how the library can be improved! Unfortunately there isn't any documentation yet, but I would like that to change sometime soon. I've tried to organize things according to how I described them in this article, so don't be scared to have a look around in the tsimd headers.

Until next time…happy coding!
Posted in **C++**, **Performance**/Tagged **C++**, **Vectorization**/**Leave a comment**