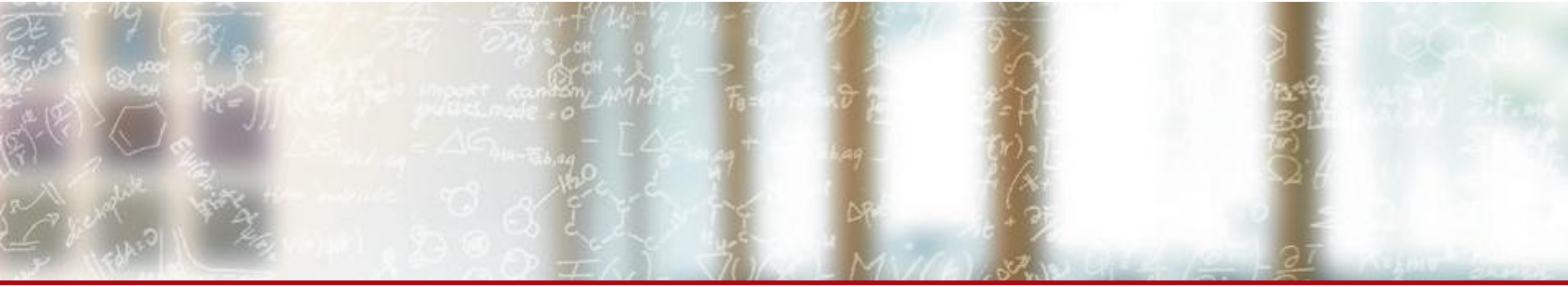
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Advanced C++ for HPC: Making a SIMD library

Sam Yates, CSCS (yates@cscs.ch)
14–16 October 2019

Outline

Why?

- Vectorization.
- SIMD intrinsics.
- Alternatives.

How?

- Design constraints.
- Representing ABIs: type maps.
- CRTP for fun and profit.
- Front-end interface.

Vectorization

CPU SIMD instructions: do more with less.

```
#define N 32768

// c = k*a + b
void sma(double* c, double k,
         const double* a, const double* b)
{
      for (unsigned i = 0; i<N; ++i) {
            c[i] = k*a[i] + b[i];
      }
}
```

g++-9.2 -fno-unroll-loops -O3 -mno-sse

```
      fldl 8(%rsp)
      xorl %eax, %eax
.L2:
      fldl (%rsi,%rax)
      fmul %st(1), %st
      faddl (%rdx,%rax)
      fstpl (%rdi,%rax)
      addq $8, %rax
      cmpq $262144, %rax
      jne .L2
      fstp %st(0)
      Ret
```

Vectorization

CPU SIMD instructions: do more with less.

```
#define N 32768

// c = k*a + b
void sma(double* __restrict c, double k,
         const double* a, const double* b)
{
      for (unsigned i = 0; i<N; ++i) {
            c[i] = k*a[i] + b[i];
      }
}
```

g++-9.2 -O3 -ffp-contract=off -march=skylake-avx512

```
      vbroadcastsd %xmm0, %ymm1
      xorl %eax, %eax
.L2:
      vmulpd (%rsi,%rax), %ymm1, %ymm0
      vaddpd (%rdx,%rax), %ymm0, %ymm0
      vmovupd %ymm0, (%rdi,%rax)
      addq $32, %rax
      cmpq $262144, %rax
      jne .L2
      vzeroupper
      ret
```

Vectorization

CPU SIMD instructions: do more with less.

`vbroadcastsd` — fill target register with single double-precision value.

`xorl %eax, %eax` — set eax/rax to zero.

`vmulpd` — multiply packed SIMD double-precision values.

`(%rsi, %rax)` — indirect/indexed addressing: contents of address rsi+rax

`xmm`*n*: first 128 bits of nth SIMD register
`ymm`*n*: first 256 bits of nth SIMD register [AVX]
`zmm`*n*: all 512 bits of nth SIMD register [AVX512]

128-bit register

```
    vbroadcastsd %xmm0, %ymm1
    xorl %eax, %eax
.L2:
    vmulpd (%rsi,%rax), %ymm1, %ymm0
    vaddpd (%rdx,%rax), %ymm0, %ymm0
    vmovupd %ymm0, (%rdi,%rax)
    addq $32, %rax
    cmpq $262144, %rax
    jne .L2
    vzeroupper
    ret
```

256-bit register

Vectorization

Wait, why is this only 4-wide?

```
#define N 32768

// c = k*a + b
void sma(double* __restrict c, double k,
         const double* a, const double* b)
{
      for (unsigned i = 0; i<N; ++i) {
            c[i] = k*a[i] + b[i];
      }
}
```

g++-9.2 -O3 -ffp-contract=off -march=skylake-avx512

```
      vbroadcastsd %xmm0, %ymm1
      xorl %eax, %eax
.L2:
      vmulpd (%rsi,%rax), %ymm1, %ymm0
      vaddpd (%rdx,%rax), %ymm0, %ymm0
      vmovupd %ymm0, (%rdi,%rax)
      addq $32, %rax
      cmpq $262144, %rax
      jne .L2
      vzeroupper
      ret
```

256 bits: 4 x double

32 bytes per iteration: 4 x double

Vectorization

How about clang?

```
#define N 32768

// c = k*a + b
void sma(double* __restrict c, double k,
         const double* a, const double* b)
{
      for (unsigned i = 0; i<N; ++i) {
            c[i] = k*a[i] + b[i];
      }
}
```

clang++-9 -O3 -fno-unroll-loops -march=skylake-avx512

```
      vbroadcastsd %xmm0, %zmm0
      xorl %eax, %eax
.LBB0_1:
      vmulpd (%rsi,%rax,8), %zmm0, %zmm1
      vaddpd (%rdx,%rax,8), %zmm1, %zmm1
      vmovupd %zmm1, (%rdi,%rax,8)
      addq $8, %rax
      cmpq $32768, %rax
      jne .LBB0_1
      vzeroupper
      retq
```

Auto-vectorization is tricksy

Compiler support in gcc and clang improving every release, but:

1. No guarantees on what you'll get.
2. Might perform unsafe transformations without being asked (gcc, icc 😠).

Auto-vectorization

Sometimes very clever…

```
#define N 32768

// return (a-b).c
double dotdiff(const double* a, const double* b,
               const double* c)
{
      double d = 0;
      for (unsigned i = 0; i<N; ++i) {
            d += (a[i]-b[i])*c[i];
      }
      return d;
}
```

clang++-9 -O3 -march=skylake-avx512

```
        vxorpd %xmm0, %xmm0, %xmm0
        xorl %eax, %eax
.LBB0_1:
        vmovsd (%rdi,%rax,8), %xmm1
        vmovsd 8(%rdi,%rax,8), %xmm2
        vsubsd (%rsi,%rax,8), %xmm1, %xmm1
        vmulsd (%rdx,%rax,8), %xmm1, %xmm1
        vsubsd 8(%rsi,%rax,8), %xmm2, %xmm2
        vmulsd 8(%rdx,%rax,8), %xmm2, %xmm2
        vaddsd %xmm1, %xmm0, %xmm0
        vaddsd %xmm2, %xmm0, %xmm0
        vmovsd 16(%rdi,%rax,8), %xmm1
        vsubsd 16(%rsi,%rax,8), %xmm1, %xmm1
        vmulsd 16(%rdx,%rax,8), %xmm1, %xmm1
        vmovsd 24(%rdi,%rax,8), %xmm2
        vsubsd 24(%rsi,%rax,8), %xmm2, %xmm2
        vaddsd %xmm1, %xmm0, %xmm0
        vmulsd 24(%rdx,%rax,8), %xmm2, %xmm1
        vaddsd %xmm1, %xmm0, %xmm0
        addq $4, %rax
        cmpq $32768, %rax
        jne .LBB0_1
        retq
```

Auto-vectorization

Sometimes very clever… if given enough leeway.

clang++-9 -O3 -ffast-math -march=skylake-avx512

```
        vxorpd %xmm0, %xmm0, %xmm0
        xorl %eax, %eax
        vxorpd %xmm1, %xmm1, %xmm1
        vxorpd %xmm2, %xmm2, %xmm2
        vxorpd %xmm3, %xmm3, %xmm3
.LBB0_1:
        vmovupd (%rdi,%rax,8), %zmm4
        vmovupd 64(%rdi,%rax,8), %zmm5
        vmovupd 128(%rdi,%rax,8), %zmm6
        vmovupd 192(%rdi,%rax,8), %zmm7
        vsubpd (%rsi,%rax,8), %zmm4, %zmm4
        vsubpd 64(%rsi,%rax,8), %zmm5, %zmm5
        vsubpd 128(%rsi,%rax,8), %zmm6, %zmm6
        vsubpd 192(%rsi,%rax,8), %zmm7, %zmm7
        vfmadd231pd (%rdx,%rax,8), %zmm4, %zmm0
        vfmadd231pd 64(%rdx,%rax,8), %zmm5, %zmm1
        vfmadd231pd 128(%rdx,%rax,8), %zmm6, %zmm2
        vfmadd231pd 192(%rdx,%rax,8), %zmm7, %zmm3
        addq $32, %rax
        cmpq $32768, %rax
        jne .LBB0_1
```

4x unroll
8-wide fma

```
        vaddpd %zmm0, %zmm1, %zmm0
        vaddpd %zmm0, %zmm2, %zmm0
        vaddpd %zmm0, %zmm3, %zmm0
        vextractf64x4 $1, %zmm0, %ymm1
        vaddpd %zmm1, %zmm0, %zmm0
        vextractf128 $1, %ymm0, %xmm1
        vaddpd %xmm1, %xmm0, %xmm0
        vpermilpd $1, %xmm0, %xmm1
        vaddsd %xmm1, %xmm0, %xmm0
        vzeroupper
        retq
```

4x8 reduction

Auto-vectorization

Transcendentals?

```cpp
#include <cmath>
#define N 32768

// c = exp(a)
void vexp(double* __restrict c, const double* a)
{
	for (unsigned i = 0; i<N; ++i) {
		c[i] = std::exp(a[i]);
	}
}
```

clang++-9 -O3 -fno-unroll-loops -ffast-math -march=skylake-avx512

```
# inside the loop:
vmovups (%r15,%rbx,8), %xmm0
vmovaps %xmm0, 48(%rsp)
vmovups 16(%r15,%rbx,8), %xmm0
vmovaps %xmm0, 16(%rsp)
vmovups 32(%r15,%rbx,8), %xmm0
vmovaps %xmm0, 64(%rsp)
vmovups 48(%r15,%rbx,8), %xmm0
vmovaps %xmm0, 32(%rsp)
vzeroupper
callq __exp_finite
vmovaps %xmm0, (%rsp)
vpermilpd $1, 32(%rsp), %xmm0
callq __exp_finite
# [ … and another 6 calls to __exp_finite ]
# [ … then pack everything into zmm0 and store ]
```

Auto-vectorization is tricksy

Compiler support in gcc and clang improving every release, but:

1. No guarantees on what you'll get.
2. Might perform unsafe transformations without being asked (gcc, icc 😠).
3. Often need to permit unsafe assumptions with `-ffast-math` anyway.
4. Even with `-ffast-math`, transcendentals may be serialized.

Auto-vectorization

Scatter conflicts

```
#define N 32768

void indirect_product(
      double* __restrict c, const unsigned* index,
      const double* a, const double* b)
{
      for (unsigned i = 0; i<N; ++i) {
            // This is fine:
            c[index[i]] = a[i]*b[i];
      }
}
```

clang++-9 -O3 -fno-unroll-loops -ffast-math -march=skylake-avx512

```
      xorl %eax, %eax
.LBB0_1:
      vmovupd (%rcx,%rax,8), %zmm0
      vmulpd (%rdx,%rax,8), %zmm0, %zmm0
      vpmovzxdq (%rsi,%rax,4), %zmm1
      kxnorw %k0, %k0, %k1
      vscatterqpd %zmm0, (%rdi,%zmm1,8) {%k1}
      addq $8, %rax
      cmpq $32768, %rax
      jne .LBB0_1
      vzeroupper
      retq
```

Auto-vectorization

Scatter conflicts

```
#define N 32768

void indirect_compound_product(
      double* __restrict c, const unsigned* index,
      const double* a, const double* b)
{
      for (unsigned i = 0; i<N; ++i) {
            // This is NOT fine:
            c[index[i]] += a[i]*b[i];
      }
}
```

Potential conflict: e.g. if `index[2]==index[4]`

clang++-9 -O3 -fno-unroll-loops -ffast-math -march=skylake-avx512

```
      xorl %eax, %eax
.LBB0_1:
      vmovsd (%rdx,%rax,8), %xmm0
      vmovsd (%rcx,%rax,8), %xmm1
      movl (%rsi,%rax,4), %r8d
      vfmadd213sd (%rdi,%r8,8), %xmm0, %xmm1
      vmovsd %xmm1, (%rdi,%r8,8)
      incq %rax
      cmpq $32768, %rax
      jne .LBB0_1
      retq
```

With *-Rpass-analysis=loop-vectorize*

```
<source>:7:5: remark: loop not vectorized: unsafe
dependent memory operations in loop. [...]
```

Auto-vectorization is tricksy

Compiler support in gcc and clang improving every release, but:

1. No guarantees on what you'll get.
2. Might perform unsafe transformations without being asked (gcc, icc 😠).

3. Often need to permit unsafe assumptions with `-ffast-math` anyway.
4. Even with `-ffast-math`, transcendentals may be serialized.

5. Can't express knowledge of potential conflicts or lack of conflicts in arithmetic syntax.

SIMD Intrinsics

Compilers generally support a number of *intrinsic* built-in functions.

- Provide an interface to non-standard functionality.
- Compiler specific.

SIMD operations for a particular CPU architecture are provided by intrinsics in gcc, clang, and icc, and are (mostly) the same for each compiler.

- Most correspond to a single CPU SIMD operation.
- They aren’t necessarily *compiled exactly* to that CPU SIMD operation.

SIMD Intrinsics

Auto-vectorized

```
#include <cmath>
#include <immintrin.h>
#define N 32768

void sfma(
        double* __restrict c, double k,
        const double* a, const double* b)
{
        for (unsigned i = 0; i<N; ++i) {
                c[i] = std::fma(k, a[i], b[i]);
        }
}
```

clang++-9 -O3 -fno-unroll-loops -march=skylake-avx512

```
        vbroadcastsd %xmm0, %zmm1
        xorl %eax, %eax
.L2:
        vmovupd (%rsi,%rax), %zmm0
        vfmadd213pd (%rdx,%rax), %zmm1, %zmm0
        vmovupd %zmm0, (%rdi,%rax)
        addq $64, %rax
        cmpq $262144, %rax
        jne .L2
        vzeroupper
        ret
```

SIMD Intrinsics

Explicit vectorization with intrinsics

```
#include <immintrin.h>
#define N 32768

void sfma(
        double* __restrict c, double k,
        const double* a, const double* b)
{
        __m512d vk = _mm512_set1_pd(k);
        for (unsigned i = 0; i<N; i += 8) {
                __m512d va = _mm512_loadu_pd(a+i);
                __m512d vb = _mm512_loadu_pd(b+i);
                __m512d vc = _mm512_fmadd_pd(vk, va, vb);
                _mm512_storeu_pd(c+i, vc);
        }
}
```

clang++-9 -O3 -fno-unroll-loops -march=skylake-avx512

```
        vbroadcastsd %xmm0, %zmm1
        xorl %eax, %eax
.L2:
        vmovupd (%rsi,%rax), %zmm0
        vfmadd213pd (%rdx,%rax), %zmm1, %zmm0
        vmovupd %zmm0, (%rdi,%rax)
        addq $64, %rax
        cmpq $262144, %rax
        jne .L2
        vzeroupper
        retq
```

Exactly the same generated code.

Using SIMD intrinsics vs auto-vectorization

The Good

- Generated code will be close to what you write.
- Free to support full IEEE semantics or make numerical assumptions.
- Can supply own vectorized implementations of transcendental functions.
- Can be explicit about conflict assumptions in indirect memory operations.

The Bad

- Different code for every architecture: huge maintenance burden.
- Very, very non-standard C++.

The Ugly

⁉

```
a = _mm512_castsi512_pd(_mm512_and_epi64(_mm512_castpd_si512(x),
_mm512_set1_epi64(0x7fffffffffffffff)))
```

Alternatives

1. OpenMP SIMD extensions

2. The Vc library: `github.com/VcDevel/Vc`

3. `std::experimental::simd` — coming soon to a libstdc++ near you.

   Proposed SIMD extensions described in working draft `N4808` are based on a subset of the Vc library.

Or we can write our own.

Write your own SIMD library, what are you crazy?

Excuses:

- Vc didn't do everything we wanted in our project.
- We really like writing ratpoly transcendental function approximations.*
- It turned out to be useful as an example of C++ patterns.

* This might be a slight overstatement.

SIMD library design requirements and goals

1. Broadly follow the API given in the N4808 proposal, or a simpler version.
    - Lower burden for migration to a future standard library implementation.
    - Benefit from existing API design work.

SIMD library design requirements and goals

1. Broadly follow the API given in the N4808 proposal, or a simpler version.

```
#include <experimental/simd>
#define N 32768

using double4 = std::experimental::fixed_size_simd<double, 4>;
using std::experimental::element_aligned;

void sma(double* __restrict c, double k, const double* a, const double* b) {
      for (unsigned i = 0; i<N; i += 4) {
            double4 va(a+i, element_aligned);
            double4 vb(b+i, element_aligned);
            auto vc = k*va + vb;
            vc.copy_to(c+i, element_aligned);
      }
}
```

SIMD library design requirements and goals

You can run this code on Compiler Explorer right now.

`gcc-9.2 -O3 -march=skylake-avx512 -ffast-math -std=c++17`

```
#include <experimental/simd><https://raw.githubusercontent.com/VcDevel/std-simd/compiler_explorer/simd.h>
#define N 32768

using double4 = std::experimental::fixed_size_simd<double, 4>;
using std::experimental::element_aligned;

void sma(double* __restrict c, double k, const double* a, const double* b) {
      for (unsigned i = 0; i<N; i += 4) {
            double4 va(a+i, element_aligned);
            double4 vb(b+i, element_aligned);
            auto vc = k*va + vb;
            vc.copy_to(c+i, element_aligned);
      }
}
```

SIMD library design requirements and goals

1. Broadly follow the API given in the N4808 proposal, or a simpler version.

2. Separate user-visible API from architecture-specific implementations.

   - Many SIMD back-ends can share a common front-end that handles all the syntactic sugar.

   - Reduces the development cost of writing a new back-end.

SIMD library design requirements and goals

1. Broadly follow the API given in the N4808 proposal, or a simpler version.

2. Separate user-visible API from architecture-specific implementations.

```
// user code

double4 a = b + c;

double* x = ...;
int4 index = ...;
indirect(x, index) = a;
```

```
// equivalent back-end code

double4_impl::vector_type A =
double4_impl::add(B, C);

double4_impl::scatter(tag<int4_impl
>{}, A, x, I);
```

SIMD library design requirements and goals

1. Broadly follow the API given in the N4808 proposal, or a simpler version.

2. Separate user-visible API from architecture-specific implementations.

3. Decouple back-end functionality from any specific SIMD data representation.

    - Operations on the same data representation (e.g. `__m256`) may have different implementations for different SIMD ISAs that are supported on the same platform (e.g. AVX2, AVX512).

    - Back-ends then should provide functions operating on SIMD data, rather than wrapping SIMD data and providing operations as methods.

SIMD library design requirements and goals

1. Broadly follow the API given in the N4808 proposal, or a simpler version.

2. Separate user-visible API from architecture-specific implementations.

3. Decouple back-end functionality from any specific SIMD data representation.

4. Supply fallback implementations for operations that are unimplemented or unsupported by the architecture.

    - If we can always move to and from an array representation, we can write generic fallback code, relying on the optimizer to remove redundant copying.

`tinysimd`: a tiny SIMD library

Let’s make a prototype!

- Two back-ends: AVX2 and a generic std::array-based back-end.
- Arithmetic operations: + and * for SIMD double and int values.
- Store to/from memory.
- Element (lane) access.
- Indirect memory operations (gather/scatter).

`tinysimd`: a tiny SIMD library

Back-end design: ABI mapping

How will we represent a particular SIMD data-type and set of operations?

- N4808: an *ABI tag* describes a mapping from a data-parallel type to a particular width and binary representation.
- *Simplify for `tinysimd`: make sizes explicit, and use an ABI tag to select a SIMD ISA implementation, too.*
- Two explicit choices: `avx2` and `generic`.

`tinysimd`: a tiny SIMD library

Back-end design: ABI mapping

Convention:

- `abi::`*`abitag`*`<T, N>::type` is the *`abitag`* implementation type for `N`-wide vector of `T`, or `void` if unavailable.
- `abi::generic<T, N>::type` will map to a valid implementation based on `std::array` for every `T` and `N`.
- `abi::default_abi<T, N>::type` will map to the ‘best’ available implementation for `T` and `N`.

```
#include <tinysimd/generic.h>
#include <tinysimd/avx2.h>

namespace tinysimd {

template <typename...>
struct first_not_void_of; // (details elided)

namespace abi {
    template <typename T, unsigned N>
    struct default_abi {
        using type = typename first_not_void_of<
            typename avx2<T, N>::type,
            typename generic<T, N>::type
        >::type;
    };
}
}
```

`tinysimd`: a tiny SIMD library

Back-end design: implementation API

What will an implementation class look like? Need a class which provides:

- A type for the SIMD representation — can use a traits class.
- Functions operating on this representation — these can be static member functions.

What functions do we need for `tinysimd`?

- Copy to/from memory
- Lane access
- Scalar to vector
- Arithmetic: add, multiply
- Indirect reads (gather)
- Indirect writes (scatter)

tinysimd: a tiny SIMD library

Back-end design: fallback implementations

If an implementation provides a traits class instance, and provides a `copy_to` and `copy_from` function for writing to and reading from memory, a fallback implementation class can perform every other operation with everyday C++.

```
// in tinysimd/fallback.h

template <typename I> struct simd_traits {
      static constexpr unsigned width = 0;
      using scalar_type = void;
      using vector_type = void;
};
```

```
// in tinysimd/avx2.h

struct avx2_double4;
template <> struct simd_traits<avx2_double4> {
      static constexpr unsigned width = 4;
      using scalar_type = std::int32_t;
      using vector_type = __m256d;
};
```

```
template <typename I> struct fallback {
    static constexpr unsigned width = simd_traits<I>::width;
    using scalar_type = typename simd_traits<I>::scalar_type;
    using vector_type = typename simd_traits<I>::vector_type;
    using store = scalar_type[width];

    static vector_type broadcast(scalar_type x) {
        store a;
        std::fill(std::begin(a), std::end(a), x);
        return I::copy_from(a);
    }
    static vector_type add(vector_type u, const vector_type v) {
        store a, b, r;
        I::copy_to(u, a);
        I::copy_to(v, b);
        for (unsigned i = 0; i<width; ++i) r[i] = a[i]+b[i];
        return I::copy_from(r);
    }
    // similarly: element(), set_element(), mul() …
};
```

`tinysimd`: a tiny SIMD library

Back-end design: fallback implementations

An actual SIMD implementation class can use intrinsics, or call the fallback implementation. But:

- Forwarding methods add a lot of boilerplate.
- Adding a new back-end method requires adding this method to every implementation class.

Solution: just inherit from the fallback class!

```
struct avx2_double4 {
    static void copy_to(__m256d v, double* p) {
        _mm256_storeu_pd(p, v);
    }
    static __m256d copy_from(const double* p) {
        return _mm256_loadu_pd(p);
    }
    static __m256d broadcast(double v) {
        return _mm256_set1_pd(v);
    }
    static __m256d add(__m256d a, __m256d b) {
        return fallback<avx2_double4>::add(a, b);
    }
    // And mul, etc. ...
};
```

`tinysimd`: a tiny SIMD library

Back-end design: CRTP

CRTP is the Curiously Recurring Template Pattern: a class X derives from a parameterized class B, with parameter X.

- Used for ‘static polymorphism’ (among other things).
- Often abused.

For us:

- Base class provides generic implementations; uses specialized methods from the derived class where it can.
- Missing methods in the derived class automatically are provided by the base.

```
struct avx2_double4: fallback<avx_double4> {
    static void copy_to(__m256d v, double* p) {
        _mm256_storeu_pd(p, v);
    }
    static __m256d copy_from(const double* p) {
        return _mm256_loadu_pd(p);
    }
    static __m256d broadcast(double v) {
        return _mm256_set1_pd(v);
    }
    // Fallback methods for everything else just inherited.
};
```

Aside: CRTP

Perils of CRTP implementation

Compiler won’t catch errors that it would with dynamic types and overloads:

- A static method `fallback<I>::foo()` might call `add()` (invoking the fallback version) instead of `I::add()` (the possibly specialized version from `I`).
- A static method `I::bar(...)` may have a mismatch in its signature preventing it from specializing `fallback<i>::bar(...)`.

Aside: CRTP

CRTP without static methods

Member functions in derived class can be invoked by base class by downcasting `this`.

- Base class can implement a common functionality based on operations implemented in the derived class — reduces code duplication.
- Same sorts of caveats apply as in static case.

```
template <typename D> struct base {
    D* derived() {
        return static_cast<D*>(this);
    }
    void foo() {
        derived()->op_bar();
        derived()->op_baz();
    }
};

struct derived: base<derived> {
    friend struct base<derived>;
private:
    void op_bar() { /* … */ }
    void op_baz() { /* … */ }
};
```

tinysimd: a tiny SIMD library

Back-end design: generic implementation

Generic no-intrinsics back-end: just use fallback routines.

Hopefully compiler will elide all those copies?

- Sometimes it will.
- Sometimes the optimizer gets horribly confused if width>1.

```
struct generic: fallback<generic<T, N>> {
    using vector_type = std::array<T, N>;
    static void copy_to(vector_type v, T* p) {
        std::copy(std::begin(v), std::end(v), p);
    }
    static vector_type copy_from(const T* p) {
        vector_type v;
        std::copy(p, p+N, std::begin(v));
        return v;
    }
};
template <typename T, unsigned N>
struct simd_traits<generic<T, N>> {
    static constexpr unsigned width = N;
    using scalar_type = T;
    using vector_type = std::array<T, N>;
};
```

`tinysimd`: a tiny SIMD library

Does it work?

With implementations filled out for avx2_double4, do we get vectorized code?

```
#include <tinysimd/avx2.h>
#define N 32768

using namespace tinysimd;
using S = avx2_double4;
using vec = simd_traits<S>::vector_type;
constexpr unsigned width = simd_traits<S>::width;

void sfma(double* __restrict c, double k,
          const double* a, const double* b)
{
    for (unsigned i = 0; i<N; i += width) {
        vec va = S::copy_from(a+i);
        vec vb = S::copy_from(b+i);
        vec vc = S::fma(S::broadcast(k), va, vb);
        S::copy_to(vc, c+i);
    }
}
```

clang++-9 -O3 -fno-unroll-loops -march=skylake-avx512

```
sfma(double*, double, double const*, double const*):
        vbroadcastsd %xmm0, %ymm0
        xorl %eax, %eax
.LBB0_1:
        vmovupd (%rsi,%rax,8), %ymm1
        vfmadd213pd (%rdx,%rax,8), %ymm0, %ymm1
        vmovupd %ymm1, (%rdi,%rax,8)
        addq $4, %rax
        cmpq $32768, %rax
        jb .LBB0_1
        vzeroupper
        retq
```

`tinysimd`: a tiny SIMD library

Back-end design: heterogeneous operations — gather and scatter

Scatter operation (indirect SIMD write): `p[index[i]] = v[i]`

Gather operation (indirect SIMD read): `v[i] = p[index[i]]`

Heterogeneous operation: SIMD value type and SIMD index type.

SIMD value implementation class knows how to interpret the value vector-type, but how will it know how to interpret the index vector-type?

`tinysimd`: a tiny SIMD library

Back-end design: heterogeneous operations — gather and scatter

A ‘tag’ type can represent the implementation class for the index vector, without requiring that the implementation class be complete.

```
template <typename X> struct tag {};
```

Back-end can then overload gather for different index implementations, that may share the same index *representation*.

- Example: suppose `avx512_int8` and `avx512_long8` both use a `__m512i` representation. Then different `gather` overloads for `tag<avx512_int8>` and `tag<avx512_long8>` can use the corresponding intrinsic.

tinysimd: a tiny SIMD library

Back-end design: heterogeneous operations — gather and scatter

In the fallback class, use the Index implementation provided by the tag to extract the indicies, and perform the loads one by one:

```
template <typename I> struct tag {};
template <typename I> struct fallback<I> {
    // ...
    template <typename IndexI>
    static vector_type gather(tag<IndexI>, const scalar_type* __restrict p,
                              typename simd_traits<IndexI>::vector_type index) {
        using index_store = typename simd_traits<IndexI>::scalar_type[width];
        index_store j;
        IndexI::copy_to(index, j);

        store a;
        for (unsigned i = 0; i<width; ++i) a[i] = p[j[i]];
        return I::copy_from(a);
    }
};
```

`tinysimd`: a tiny SIMD library

Back-end design: heterogeneous operations — gather and scatter

In the implementation class:

- Use a using declaration to bring in the templated overloads from fallback.
- Provide specific implementations as required.

```
struct avx2_int4;
struct avx2_double2: fallback<avx2_double4> {
    // ...
    using fallback<avx2_double4>::gather;
    static __m256d gather(tag<avx2_int4>, const double* p, __m128i index) {
        return _mm256_i32gather_pd(p, index, 8);
    }
};
```

Extend similarly for scatter operations (and other heterogeneous operations such as value casting).

tinysimd: a tiny SIMD library

Back-end design: gather demo

```
#include <tinysimd/avx2.h>
#define N 32768

using namespace tinysimd;
using d4impl = avx2_double4;
using d4 = simd_traits<d4impl>::vector_type;
using i4impl = avx2_int4;
using i4 = simd_traits<i4impl>::vector_type;
constexpr unsigned width = 4;

// c[i] = a[index[i]]*b[index[i]]
void mul_indexed(double* __restrict c, const int* index,
                 const double* a, const double* b)
{
    for (unsigned i = 0; i<N; i += width) {
        i4 vi = i4impl::copy_from(index+i);
        d4 va = d4impl::gather(tag<i4impl>{}, a, vi);
        d4 vb = d4impl::gather(tag<i4impl>{}, b, vi);
        d4impl::copy_to(d4impl::mul(va, vb), c+i);
    }
}
```

clang++-9 -O3 -fno-unroll-loops -march=skylake-avx512

```
mul_indexed(double*, int const*, double const*, double
const*):
        xorl %eax, %eax
.LBB0_1:
        vmovupd (%rsi,%rax,4), %xmm0
        vpcmpeqd %ymm1, %ymm1, %ymm1
        vxorpd %xmm2, %xmm2, %xmm2
        vgatherdpd %ymm1, (%rdx,%xmm0,8), %ymm2
        vpcmpeqd %ymm1, %ymm1, %ymm1
        vxorpd %xmm3, %xmm3, %xmm3
        vgatherdpd %ymm1, (%rcx,%xmm0,8), %ymm3
        vmulpd %ymm3, %ymm2, %ymm0
        vmovupd %ymm0, (%rdi,%rax,8)
        addq $4, %rax
        cmpq $32768, %rax
        jb .LBB0_1
        vzeroupper
        retq
```

`tinysimd`: a tiny SIMD library

Front-end

Public API components:

1. A wrapper `simd_wrap<I>` around the implementation class `I`, containing a member of type `simd_traits<I>::vector_type`. Provides operator overloads.

2. A type map `simd<V, N, Abi>` to the wrapper `simd_wrap<I>` where `I` is `Abi<V, N>::type`.

3. A function `indirect` that wraps a pointer and a SIMD index into an indirect_expression object used to describe indexed operations.

`tinysimd`: a tiny SIMD library

Front-end: `simd_wrap`

Constructors

```
template <typename I> struct simd_wrap {
private:
    static_assert(!std::is_void<I>::value, "unsupported");
    using vector_type = typename simd_traits<I>::vector_type;
    vector_type value_;

public:
    using scalar_type = typename simd_traits<I>::scalar_type;
    static constexpr unsigned width = simd_traits<I>::width;

    simd_wrap() = default;
    simd_wrap(const simd_wrap& other) = default;

    simd_wrap(const scalar_type& x):
       value_(I::broadcast(x)) {}
    simd_wrap(const scalar_type (&a)[width]):
        value_(I::copy_from(a)) {}
    explicit simd_wrap(const scalar_type *p):
        value_(I::copy_from(p)) {}
```

Assignment and writes to memory

```
    simd_wrap& operator=(const simd_wrap& other) = default;
    simd_wrap& operator=(const scalar_type& x) {
        value_ = I::broadcast(x);
        return *this;
    }
    void copy_to(scalar_type* p) const {
        I::copy_to(value_, p);
    }

    // ...
};
```

tinysimd: a tiny SIMD library

Front-end: simd_wrap arithmetic operations

```
template <typename I> struct simd_wrap {
    // …
private:
    static simd_wrap wrap(const vector_type& v) {
        simd_wrap s;
        s.value_ = v
        return s;
    }
public:
    friend simd_wrap operator+(const simd_wrap& a, const simd_wrap& b) {
        return wrap(I::add(a.value_, b.value_));
    }
    friend simd_wrap operator*(const simd_wrap& a, const simd_wrap& b) {
        return wrap(I::mul(a.value_, b.value_));
    }
    friend simd_wrap fma(const simd_wrap& a, const simd_wrap& b, const simd_wrap& c) {
        return wrap(I::fma(a.value_, b.value_, c.value_));
    }

    simd_wrap& operator+=(const simd_wrap& a) { value_ = I::add(value_, a.value_); return *this; }
    simd_wrap& operator*=(const simd_wrap& a) { value_ = I::mul(value_, a.value_); return *this; }
    // …
};
```

`tinysimd`: a tiny SIMD library

Front-end: `simd_wrap` element access

If v is a SIMD vector value, how do we change the elements in this vector?

A proxy class can hold on to the SIMD value and the element index, and perform an update when it is assigned:

```
simd<double, 4> v = … ;

// v[3] is actually of type
// simd<double, 4>::element_proxy
v[3] = 2.3;
```

```
template <typename I> struct simd_wrap { // …
    struct element_proxy{
        vector_type* vptr;
        int i;

        element_proxy operator=(scalar_type x){
            I::set_element(*vptr, i, x);
            return *this;
        }
        operator scalar_type() const {
            return I::element(*vptr, i);
        }
    };

    element_proxy operator[](int i) {
        return element_proxy{&value_, i};
    }
    scalar_type operator[](int i) const {
        return I::element(value_, i);
    }
};
```

`tinysimd`: a tiny SIMD library

Front-end: adding indirect operations

How to represent indexed, indirect access to memory? Unify syntax for gather and scatter operations?

```
simd<int, 4> indices = … ;

// gather: v[i] = p[indices[i]]
simd<double, 4> v = indirect(p, indices);

// scatter: p[indices[i]] = v[i]
indirect(p, indices) = v;
```

Make an `indirect_expression` object that is returned by `indirect()`, and represents the pointer to memory and SIMD value of offsets.

```
template <typename I> struct simd_wrap;

template <typename I, typename T>
struct indirect_expression {
    using index_type = typename simd_traits<I>::vector_type;
    T* p;
    index_type index;

    indirect_expression(T* p, const index_type& index):
        p(p), index(index) {}

    template <typename J>
    indirect_expression& operator=(const simd_wrap<J>& a) {
        a.copy_to(*this);
        return *this;
    }
};
```

tinysimd: a tiny SIMD library

Front-end: adding indirect operations

We can add `indirect()` to `simd_wrap` as a friend function (for access to the private `value_`):

```
template <typename I> struct simd_wrap {
    // ...
    template <
        typename Ptr,
        typename =
            std::enable_if_t<std::is_pointer<Ptr>::value>
    >
    friend auto indirect(Ptr p, const simd_wrap& index)
    {
        using V = std::remove_reference_t<decltype(*p)>;
        return indirect_expression<I, V>(p, index.value_);
    }
};
```

Then overload `simd_wrap` constructor and `copy_to` for `indirect_expression`.

```
template <typename I> struct simd_wrap {
    // ...
    template <typename J>
    simd_wrap(indirect_expression<J, const scalar_type> pi):
        value_(I::gather(tag<J>{}, pi.p, pi.index)) {}

    template <typename J>
    simd_wrap(indirect_expression<J, scalar_type> pi):
        value_(I::gather(tag<J>{}, pi.p, pi.index)) {}

    template <typename J>
    void copy_to(indirect_expression<J, scalar_type> pi)
        const
    {
        I::scatter(tag<J>{}, value_, pi.p, pi.index);
    }
};
```

tinysimd: a tiny SIMD library

Examples

```
#include <tinysimd/simd.h>

using double4 = tinysimd::simd<double, 4>;

// c[i] = k*a[i] + b[i]
void sma(double* __restrict c, double k,
         const double* a, const double* b)
{
    for (unsigned i = 0; i<N; i += 4) {
        double4 va(a+i), vb(b+i);
        auto vc = k*va + vb;
        vc.copy_to(c+i);
    }
}
```

clang++-9 -O3 -fno-unroll-loops -march=skylake-avx512

```
sma(double*, double, double const*, double const*):
        vbroadcastsd %xmm0, %ymm0
        xorl %eax, %eax
.LBB0_1:
        vmulpd (%rsi,%rax,8), %ymm0, %ymm1
        vaddpd (%rdx,%rax,8), %ymm1, %ymm1
        vmovupd %ymm1, (%rdi,%rax,8)
        addq $4, %rax
        cmpq $32768, %rax
        jb .LBB0_1
        vzeroupper
        retq
```

tinysimd: a tiny SIMD library

Examples

```
#include <tinysimd/simd.h>

using int4 = tinysimd::simd<int, 4>;
using double4 = tinysimd::simd<double, 4>;

// c[i] = a[index[i]] + b[index[i]]
void mul_indexed(
    double* __restrict c, const int* index,
    const double* a, const double* b)
{
    for (unsigned i = 0; i<N; i+=4) {
        int4 vi(index+i);
        double4 vc = double4(indirect(a, vi))*
                     double4(indirect(b, vi));
        vc.copy_to(c+i);
    }
}
```

clang++-9 -O3 -fno-unroll-loops -march=skylake-avx512

```
permuted_mul(double*, int const*, double const*,
double const*):
      xorl %eax, %eax
.LBB0_1:
      vmovupd (%rsi,%rax,4), %xmm0
      vpcmpeqd %ymm1, %ymm1, %ymm1
      vxorpd %xmm2, %xmm2, %xmm2
      vgatherdpd %ymm1, (%rdx,%xmm0,8), %ymm2
      vpcmpeqd %ymm1, %ymm1, %ymm1
      vxorpd %xmm3, %xmm3, %xmm3
      vgatherdpd %ymm1, (%rcx,%xmm0,8), %ymm3
      vmulpd %ymm3, %ymm2, %ymm0
      vmovupd %ymm0, (%rdi,%rax,8)
      addq $4, %rax
      cmpq $32768, %rax
      jb .LBB0_1
      vzeroupper
      retq
```

`tinysimd`: a tiny SIMD library

Missing features — masks and conditionals

```
where(x>3, a) = b + c
```

Comparisons produce a `simd_mask_wrap`, that also wraps a SIMD implementation, but provides logical operations.

The `where` function produces a `where_expression`, representing a mask and a `simd_wrap` lvalue. Assignments are translated to masked SIMD expressions and blend operations provided by the implementation class.

`tinysimd`: a tiny SIMD library

Missing features — conversion/casting

```
simd<float, 4> x = /*…*/;
simd<double, 4> y = x + 2;
```

Add constructors to `simd_wrap<I>` that call another heterogeneous operation `I::cast_from` in the implementation class, with a tag for the source SIMD implementation.

A top-level templated function `simd_wrap<I> simd_cast(simd_wrap<J>)` asserts width equality and wraps the conversion functions.

Resources

Sample code

Git repo: https://github.com/eth-cscs/examples_cpp

`tinysimd` code and examples in directory `Code/tinysimd`

x86-64 instructions and intrinsics

Compiler intrinsics: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Félix Cloutier’s x86 instruction set reference: https://www.felixcloutier.com/x86/

Agner Fog’s instruction tables: https://www.agner.org/optimize/#manuals

Compiler explorer

https://godbolt.org

`tinysimd`: a tiny SIMD library

Addendum — constrained indices

General gather/scatter is typically slow, even if supported in the architecture.

But:

- If we knew the indices were all the same:
  - gather is a single scalar load.

- If we knew the indices were all distinct:
  - could do an indirect compound add without conflict.

`tinysimd`: a tiny SIMD library

Addendum — constrained indices

Allow user code to add a *constraint* to an indirect expression

Indices	Constraint
3, 3, 3, 3	`constant`
3, 4, 5, 6	`contiguous`
3, 5, 2, 7	`independent`
3, 5, 5, 8	`monotonic`
3, 5, 2, 5	`none`

`tinysimd`: a tiny SIMD library

Addendum — constrained indices

1. Add a `constraint` field to `indirect_expression`, defaulting to `constraint::none`.

2. Add fallback implementations for gather and scatter that take a `constraint`, and dispatch accordingly.

3. Add a fallback implementation for a scatter-reduce operation, that implements the indirect += operation, dispatching on `constraint`.