DOES SOFTWARE MATURES LIKE CHEESE?

THE COMING OF AGE OF AN HPC LIBRARY

February 18, 2021

Joel FALCOU

Powered by Markdeep and Markdeep-Slides

The Genesis of E.V.E.

1001 Flavors of SIMD

- Available on all major CPU yet used sporadically
- SIMD instruction set provides large registers
- Operations are performed on multiple data at once
- Usually used by using intrinsics or praying to the AutoVectorizer Gods

E.V.E - A SIMD wrapper library

- Started as a OCAML wrapper for PPC Altivec (2005)
- Evolved into a Altivec/SSE2 wrapper (2006)
- Once hidden inside NT2 (2008-2014), then as Boost.SIMD (2014-2017)
- Now under reconstruction as a C++20 library

A Slice of Performances

Main E.V.E features

- Provides a type-based wrapper around most current SIMD instruction sets
- Strongly oriented toward numerical applications
- User-level trade-off management (fast? precise? you decide!)
- Designed as to take advantage of most of latest C++ standard

Want to know more ?

- Find it on `Github`
- Play with it on `Compiler Explorer`
- Have look at the `in-progress documentation`
- Bug me after this talk ;)

A Slice of Elegance

A Portable simd_strlen

```cpp
std::size_t simd_strlen(unsigned char const* s)
{
  eve::aligned_ptr aligned_s = eve::previous_aligned_address(s);

  eve::wide     cur     = eve::unsafe(eve::load)(aligned_s);
  auto          ignore  = eve::ignore_first(s - aligned_s.get());
  std::optional match   = eve::first_true[ignore](cur == 0);

  while (!match)
  {
    aligned_s += wide::static_size;
    cur = eve::unsafe(eve::load)(aligned_s);
    match = eve::first_true(cur == 0);
  }

  return static_cast<std::size_t>(aligned_s.get() + *match - s);
}
```

The Message of this Talk

Library design is two-sided

- User-facing API must be compelling to use
- Dev-facing API must enable fast development

How do three standards change the library?

- New language features
- New idioms

How the vision of a long term project changes?

- The importance of user API perception
- Design for usability

Type Interface

Type Interfaces as Public Relationships

The Context

- First version of E.V.E provided a `pack` abstraction for SIMD registers
- `pack` was complete with array and tuple-like interface
- One can even iterate over the scalar contents

```
// Definition
template<typename Type, std::size_t Cardinal, typename ABI = ...> struct pack;

// Usage
pack<int,8> x;

x[0] = 1;
for(std::size_t i=1;i<x.size();++i)
  x[i] = 2* x[i-1];

std::cout << x[x.size()-1] << "\n";
```

Type Interfaces as Public Relationships

The Issues

- People kept trying to use `pack`, an abstraction of a register, as a genuine array
- Most frequent question : “Why does `pack<float,735>` doesn't work?”
- People will often just write bad scalar code instead of using SIMD
- Implementation required heavy aliasing hand-waving

The Solution

- Renamed `pack` (smells like an array) to `wide` (descriptor of the register)
- Cut off the `Type x Integer` interface of `pack` in favor of an all-type one
- Remove the Iterator and Array interface in favor of explicit `get/set`
- **[C++11]** Provide a lambda based constructor to prevent people iterating at initialization
- **[C++11]** Statically assert sizes are actual SIMD compatible size

Type Interfaces as Public Relationships

The Solution

```cpp
// Definition
template<typename Type, typename Cardinal, typename ABI = ...> struct wide;

// Usage
wide<int,fixed<8>> x( [](auto i, auto c) { return 1+i; });

std::cout << x.get(x.size()-1) << "\n";
```

Specifics

- `fixed` is a **Cardinal type** and asserts size are SIMD compatible
- `fixed` provides internal types to generically compute an upcast or downcast Cardinal
- Other types are provided to discriminate between scalar and SIMD register of size 1
- The explicit nature of `get` makes you pause to think about it

Type Interfaces as Public Relationships

Assessing the Situation

- Users have been trained to recognize API by name
- If it looks like an array, why can't I use it as such?
- Our mistake was to fall into the Uncanny Valley of APIs

OUR FINDINGS

- API are not just about **adding** but also about **removing**.
- Don't over mimic existing type if there are “ifs” and “buts”
- Prefer semantic-rich types to simple integral constant if possible
- *“Make APIs that are easy to use correctly and hard to use incorrectly”*, Scott Meyers

Functions & Objects

Functions: The Powerhouse of Numerical Libraries

Objectives

- SIMD implementation in E.V.E need to be reusable
- Overload should be possible over architecture, instruction sets, types
- Adding special case for specific optimization must be allowed
- Easy-to-use, Easy-to-discover API

Initial Design

- E.V.E functions were 50% functions, 50% functions calling Callable Object internally
- Used Boost.Dispatch, a generic version of tag dispatching
- Some cases required resolving multiple overload resolutions
- Advantage: compile at 11, go get lunch, compilation is ready for coffee

Functions: The Powerhouse of Numerical Libraries

Objectives

- SIMD implementation in E.V.E need to be reusable
- Overload should be possible over architecture, instruction sets, types
- Adding special case for specific optimization must be allowed
- Easy-to-use, Easy-to-discover API

New Design

- [C++11/17] E.V.E uses (inline) Callable Object that calls specific optimized functions
- [C++11] Use object as type parameters
- [C++11] New API due to adding members to said Callable
- [C++14] Higher-order functions as decorators

Functions: The Powerhouse of Numerical Libraries

Types as parameters

- Some functions require a type as parameter
- But some people are still scared of templates
- Use a throw-away object to pass type to the function

```
// Definition
template<typename T> struct as { using type = T; };

inline constexpr as<double> double_ = {};
inline constexpr as<float>  single_ = {}; // etc...

// Usage
wide<int>   w;
wide<float> x = bit_cast(w, as<wide<float>>()); // use explicit type
auto        y = bit_cast(w, as(x));             // use the same type as x
auto        z = convert(x, single_);            // use pre-made type
```

Functions: The Powerhouse of Numerical Libraries

Adding API on top of Callables

- SIMD has supports for conditionally masked operations
- Acts more as semantic modifications than parameters
- E.V.E functions can be passed conditionals via `operator[]`
- Masking capability is defined on a per function basis via traits

```
// Usage
wide<int> a,b,c;

c = c + 4;                     // c = c+4
c = add[a<b](c,4);             // c = c+4 when a<b

c = c * b;                     // c = c*b
c = mul[ignore_first(2)](c,b); // c = c*b except for first 2 values
```

Functions: The Powerhouse of Numerical Libraries

Higher-Order Functions as decorators

- SIMD implementation is full of trade-off: speed, precision, standard conformance
- As functions are Callables Objects, pass them to decorator Callables
- Returns a properly setup lambda selecting the correct implementation in a lazy way
- Decorators are combinable, saving names from design space

```cpp
struct pedantic_
{
  template<typename Callable> constexpr auto operator()(Callable&& f) noexcept
  {
    return  [func = std::forward<Callable>(f)]<typename... Args>(Args&&... args)
            {
              return func( pedantic_{}, std::forward<Args>(args)...);
            };
  }
};
```

Functions: The Powerhouse of Numerical Libraries

Higher-Order Functions as decorators

- SIMD implementation is full of trade-off: speed, precision, standard conformance
- As functions are Callables Objects, pass them to decorator Callables
- Returns a properly setup lambda selecting the correct implementation in a lazy way
- Decorators are combinable, saving names from design space

```
// Usage
wide<float> x,y;

x = exp(y);                      // Regular exp call
a = saturated(add)(b,c);         // Addition with saturation
x = pedantic(exp)(y,z);          // exp with special cases for denormals/infinites
x = numeric(min)(x,y);           // minimum without taking NaNs into account
x = raw(sqrt)(x);                // sqrt with fast implementation, no error checking

y = diff(pedantic)(exp)(x);     // differential, pedantic exponential
```

Implementation of Architecture-Optimized Callables

The Issues

- A typical E.V.E functions may have had 4-8 overloads
- Some SIMD instructions + types combo were to be emulated
- Some SIMD architecture just didn't support some types
- Some functions required very specific optimizations

Design decision

- Detect SIMD architectures and instructions sets via traits
- [C++20] Use Concepts to overload on SIMD architecture
- [C++17] Use `if constexpr` to write code based on available SIMD instructions sets
- [C++14/17] Use enum based categorization to simplify type recognition

Implementation of Architecture-Optimized Callables

The Issues

- Each SIMD architecture register has a given size in bits: 128, 256, etc...
- E.V.E. calls those family of registers a **SIMD ABI**: eg. `x86_256`
- All available ABI of a given architecture models this architecture Concept
- E.g: the `x86_abi` concept is modeled by the `x86_128`, `x86_256` and `x86_512` type

Using Concepts to discriminate architectures

- This ABI plays a part in the overload resolutions
- A trait computing the ABI associated to a Type/Cardinal pair is available
- A Concept for each of those ABI based on this trait is defined
- Overload based on ABI dramatically reduces the number of overloads to consider

Implementation of Architecture-Optimized Callables

Concepts for SIMD ABI

```cpp
template<typename Type, typename Cardinal> consteval auto abi_of()
{
  constexpr auto width  = sizeof(Type) * Cardinal;
  if constexpr( spy::simd_instruction_set == spy::x86_simd_ )
  {
           if constexpr( width <= 16) return x86_128_{};
      else if constexpr( width == 32) return x86_256_{};
      else if constexpr( width == 64) return x86_512_{};
  }
  else if constexpr( spy::simd_instruction_set == spy::arm_simd_ )
  {
           if constexpr( width <= 8 )  return arm_64_{};
      else if constexpr( width == 16)  return arm_128_{};
  }
// ... etc ...
}
```

Implementation of Architecture-Optimized Callables

Concepts for SIMD ABI

- Used in functions to discriminate optimization strategies
- Minimize the number of overloads of entry-points
- Reduced compile time **by a factor of 3**

```
template<typename T, typename N, x86_abi ABI>
auto add(wide<T,N,ABI> lhs, wide<T,N,ABI> rhs)
{
  // Do something with X86 SIMD instruction sets
}

template<typename T, typename N, non_native_abi ABI> // For all other cases
auto add(wide<T,N,ABI> lhs, wide<T,N,ABI> rhs)
{
       if constexpr( is_aggregated_v<ABI> ) return aggregate(add,lhs,rhs);
  else if constexpr( is_emulated_v<ABI>   ) return map(add,lhs,rhs);
}
```

Implementation of Architecture-Optimized Callables

The Issues

- SIMD instruction sets are widly divergent even for a given ABI
- Types, micro-architectures, etc all play a role
- How to be able to write the most efficient code with the least overloads?

`if constexpr` for intrinsics selection

- Use SPY to select instructon set at compile-time
- Provide a type → enumeration function to **categorize** types
- Categories are build as bitfield encoding base type, size and cardinal
- Nest `if constexpr` according to the optimisation we want to obtain

Implementation of Architecture-Optimized Callables

`if constexpr` for intrinsics selection

```
template<typename T, typename N, x86_abi ABI> auto add(wide<T,N,ABI> lhs, wide<T,N,ABI> rhs)
{
  constexpr auto c = categorize<wide<T,N,ABI>>();

        if constexpr  ( c == category::float64x8  ) return _mm512_add_pd(lhs,rhs);
  else  if constexpr  ( c == category::float64x4  ) return _mm256_add_pd(lhs,rhs);
  else  if constexpr  ( c == category::float64x2  ) return _mm_add_pd(lhs,rhs);
  // etc...
  else  if constexpr  ( c == category::uint8x16   ) return _mm_add_epi8(lhs,rhs);
  else  if constexpr  ( current_api >= avx2 )
  {
          if constexpr  ( c == category::int64x4  ) return _mm256_add_epi64(lhs,rhs);
    else  if constexpr  ( c == category::int32x8  ) return _mm256_add_epi32(lhs,rhs);
    else  if constexpr  ( c == category::int16x16 ) return _mm256_add_epi16(lhs,rhs);
    // etc...
    else  if constexpr  ( c == category::int8x32  ) return _mm256_add_epi8(lhs,rhs);
  }
}
```

Functions: The Powerhouse of Numerical Libraries

Assessing the Situation

- Functions as Objects is a very valuable API design tool
- Names is a very small design space. Protect it
- Our mistake was to be to clever in implementation, Keep It Stupid Simple

OUR FINDINGS

- Concept and `if constexpr` are great to structure large overload set
- HOF makes API design easier on name finding
- Don't be shy to try amping up the Object side of Function Objects
- Looking forward `std::tag_invoke`

Other API Decisions

Abstraction for Optimizations

The Issues

- Some SIMD idioms requires complicated knowledge or setup
- They are usually non-trivial for the users
- We could not wait for the users to discover them

Example: register swizzling

- SIMD registers can have their content moved around
- But each instructions sets has different rules for this
- How to have users not being left out by not using the correct swizzle ?
- *"Library design for compilation time"* as put by Victor Zverovich

Abstraction for Optimizations

Sample Swizzle

```
wide<float, fixed<4>> x;

// Direct index pattern -- not very portable
auto rx = x[ pattern<3,2,1,0> ];

// Parametric swizzle - use constexpr lambda
auto rx2 = x[ as_pattern{ [](auto i, auto c) { return c-i-1; }}];

// Parametric swizzle - using pre-defined pattern
auto rx2 = x[ reverse_n<4> ];
```

Benefits

- [C++20] Use a `consteval` mapping of patterns to implementation
- No need to remember which tricks work for which architecture
- Compile-time is mitigated by using `consteval` functions over template classes

Conclusion

Time to taste!

Impact on code - Before

- Peak Boost.SIMD was 650K LOC
- Average compile-time for unit tests: 10-12s
- API was heterogeneous and prone to errors

Impact on code - After

- EVE is 54K LOC for equivalent features
- Average compile-time for unit tests: 3-4s
- API streamlined and simplified

The Heavy Hitters

- [C++20] Concepts
- [C++17] `if constexpr`

A Long Journey

15 years of Design on moving stages

- Hardware and Software were moving targets: 10+ new SIMD IS appeared since
- The ever-evolving C++ standard helped leverage ideas we deemed impossible
- Encouraged us to play around API design for users and devs

API is everything

- Libraries are more than a collection functions and types
- Names have powers, Users have memories

A Huge Thanks to:

- **Jean-Thierry Lapresté**, Mathematician extraordinaire
- **Pierre Estérie**, my former PHD student
- **Denis Yaroshevskiy**, for being a great contributor ;)

Thanks for your attention !