FROM ACROBATICS TO ERGONOMICS

A Field Report on How to Make Libraries Helpful

February 1, 2026

Joel FALCOU

LABORATOIRE INTERDISCIPLINAIRE DES SCIENCES DU NUMÉRIQUE

CODE RECKONS
Science to the CORE

A Brief History of Tools

The Original Silicon Valley

The Madeleine prehistoric site (Tursac, Dordogne, France)

A Long Time Ago, In a Valley Far Far Away

Silicium Based Tools

- Flint and later bone based tools are a give-away hints of these periods.
- If they first ensured survival, they also were designed as early social status objects.
- Tools of various level of complexity can be found all over Africa, Europe and Asia.

Two-sided flint ~500'000 BCE

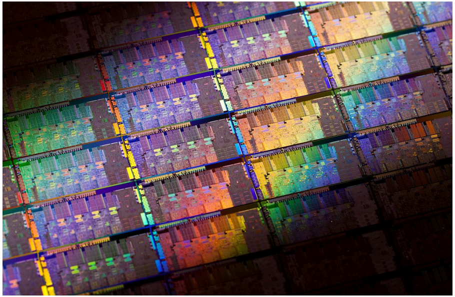
Stone Tools ~12'000 BCE

More Recently, In Another Valley Far Far Away

Silicium Based Tools

- Modern stone tools do computation with, obviously, a far better scale factor.
- CPUs are very flat flints that do computations when struck by lightning.
- *Insert joke about Moore's Law having been right for* 10^5 *years.*

Two-sided flint ~500'000 BCE



Flint-based calculator ~2000 CE

Why Making Libraries?

Libraries as know-how encapsulation

- We build libraries to avoid repeating complex, error-prone process.
- We build libraries so that others can avoid the *F*** Around and Find Out* stage.
- Libraries speed-up the pace at which new comers can become better at a given task.

Why Making Libraries?

Libraries as know-how encapsulation

- We build libraries to avoid repeating complex, error-prone process.
- We build libraries so that others can avoid the *F*** Around and Find Out* stage.
- Libraries speed-up the pace at which new comers can become better at a given task.

But what about software Libraries in C++?

- We want to achieve the elusive **Zero Cost Abstraction** state.
- So, at one point, we use *templates*.
- Then, all things fall apart.

This talk

We built libraries with *templates* ...

Libraries	Scope
`E.V.E`	Portable SIMD abstractions and algorithms
`Kumi`	*QoL* : Tuple and tuple algorithms
`Raberu`	*QoL* : Named parameters
Kiwaku (WIP)	Customisable data storage

... and we suffered so you don't

- What parts of “modern” C++ template worked.
- How do we craft our API so they are intuitive.
- Why did we chose to deviate from the standard sometimes.

API Design is Hard

Objectives

1. Be non-disruptive.
2. Domain driven optimizations.
3. Provide intuitive API for the user.
4. Support a wide architectural landscape.
5. Be efficient.

Designing Tools for Scientific Computing

Objectives

1. **Be non-disruptive.**
2. Domain driven optimizations.
3. Provide intuitive API for the user.
4. Support a wide architectural landscape.
5. **Be efficient.**

Our Approach

- **Design tools as C++ libraries.**

Designing Tools for Scientific Computing

Objectives

1. Be non-disruptive.
2. **Domain driven optimizations.**
3. **Provide intuitive API for the user.**
4. Support a wide architectural landscape.
5. Be efficient.

Our Approach

- Design tools as C++ libraries.
- **Design these libraries as Domain Specific Embedded Languages (DSEL)**

Designing Tools for Scientific Computing

Objectives

1. Be non-disruptive.
2. Domain driven optimizations.
3. **Provide intuitive API for the user.**
4. **Support a wide architectural landscape.**
5. Be efficient.

Our Approach

- Design tools as C++ libraries.
- Design these libraries as Domain Specific Embedded Languages (DSEL).
- **Use Parallel Programming Abstractions as parallel components.**

Designing Tools for Scientific Computing

Objectives

1. Be non-disruptive.
2. Domain driven optimizations.
3. Provide intuitive API for the user.
4. Support a wide architectural landscape.
5. **Be efficient.**

Our Approach

- Design tools as C++ libraries.
- Design these libraries as Domain Specific Embedded Languages (DSEL).
- Use Parallel Programming Abstractions as parallel components.
- **Use Generic/Generative Programming to deliver performance.**

Some Questions Worth Asking

What does "intuitive" mean?

- Users should see terms and operations they're accustomed to.
- Incorrect usage should be reported **early** and **clearly**
- Components should have a **Single Responsibility**.
- **Your best friends are compilation errors**.

What does "extensible" mean?

- **Users** should be able to combine pieces of the API and obtain sensible results.
- **Power Users** should be able to customize their API usage to their own corner cases.
- **Developers/Contributors** should have a clear path to add to the API.

EVE API

EVE in a Nutshell

- C++ 20 wrapper around SIMD intrinsics.
  - Library of core types.
  - 250+ numerical functions.
  - Algorithms.
- Supports all x86, ARM, AARCH64 and PPC flavors; WIP for RISC-V & WASM.
- Boost license.
- **Team Work**: Dennis Yaroshevskiy, Jean-Thierry Lapresté, Alexis Aune, myself.

Info Dump

- Find it on `Github`
- Play with it on `Compiler Explorer`
- Have look at the `documentation`

EVE Core Principles

- **EVE** relies on few types representing SIMD registers.
- Majority of operations on those registers are done through **functions**.
- Those functions should accommodate the design space of existing hardware.

```
wide<float> a,b,c,x,y;

c = c + 4;                        // c = c+4
c = add[a<b](c,4);                // c = c+4 when a<b else c
c = mul[ignore_first(2)](c,b);    // c = c*b except for first 2 values
a = add[saturated](b,c);          // Addition with saturation
x = exp[pedantic](y,z);           // exp with special cases for denormals/infinites
x = min[numeric](x,y);            // minimum without taking NaNs into account
x = sqrt[raw](x);                 // sqrt with fast implementation, no error checking

y = sqrt[ignore_first(1)][raw](x);  // combines with masks
```

EVE API

The Issues

- A typical E.V.E functions may have had 2-10 overloads.
- Some SIMD instructions + types combo need to be emulated.
- Some functions required very specific optimizations.

Design decision

- Provide EVE API based on **Callable Objects**.
- Use reusable skeletons to encapsulate common behaviors.
- Use Concepts ensure function calls validity.
- Turn architecture detection into `constexpr` information.
- Use `if constexpr` to write code based on available SIMD instructions sets.

EVE API

Functions Design

- Building new functions comes with a lot of scaffolding.
- A lot of functions share similar behaviors.
- Use pre-made callable object types to simplify definitions of functions.

What did we do?

- Took inspiration from Tag Dispatching by implementing functions as objects.
- Made an *ad hoc* definition/implementation system.
- Make it usable externally.

The `eve::abs` function

```
template<typename Options>
struct  abs_t
      : elementwise_callable<abs_t, Options, saturated_option>
{
  template<eve::value T>
  constexpr T operator()(T v) const { return EVE_DISPATCH_CALL(v); }

  EVE_CALLABLE_OBJECT(abs_t, abs_);
};

inline constexpr auto abs = functor<abs_t>;
```

The eve::abs function

```cpp
// An EVE Callable is a template type
template<typename Options>
struct  abs_t
      : elementwise_callable<abs_t, Options, saturated_option>
{
  template<eve::value T>
  constexpr T operator()(T v) const { return EVE_DISPATCH_CALL(v); }

  EVE_CALLABLE_OBJECT(abs_t, abs_);
};

inline constexpr auto abs = functor<abs_t>;
```

Functions as Callable Objects

The `eve::abs` function

```
// An EVE Callable is a template type
template<typename Options>  //  This contains the semantic options
struct  abs_t
      : elementwise_callable<abs_t, Options, saturated_option>
{
  template<eve::value T>
  constexpr T operator()(T v) const { return EVE_DISPATCH_CALL(v); }

  EVE_CALLABLE_OBJECT(abs_t, abs_);
};

inline constexpr auto abs = functor<abs_t>;
```

The `eve::abs` function

```cpp
// An EVE Callable is a template type
template<typename Options>  //  This contains the semantic options
struct  abs_t
      : elementwise_callable<abs_t, Options, saturated_option>
//      ^ This base class provides the common implementation logic
{
  template<eve::value T>
  constexpr T operator()(T v) const { return EVE_DISPATCH_CALL(v); }

  EVE_CALLABLE_OBJECT(abs_t, abs_);
};

inline constexpr auto abs = functor<abs_t>;
```

Functions as Callable Objects

The `eve::abs` function

```cpp
// An EVE Callable is a template type
template<typename Options>  //  This contains the semantic options
struct  abs_t
      : elementwise_callable<abs_t, Options, saturated_option>
//       ^ This base class provides the common implementation logic
{
  // Notice the high level user facing concept
  template<eve::value T>
  constexpr T operator()(T v) const { return EVE_DISPATCH_CALL(v); }

  EVE_CALLABLE_OBJECT(abs_t, abs_);
};

inline constexpr auto abs = functor<abs_t>;
```

The `eve::abs` function

```cpp
// An EVE Callable is a template type
template<typename Options>  //  This may contains some semantic options
struct  abs_t
      : elementwise_callable<abs_t, Options, saturated_option>
//      ^ This base class provides the common implementation
{
  // Notice the high level user facing concept
  template<eve::value T>
    // The implementation is always the same macro call
  constexpr T operator()(T v) const { return EVE_DISPATCH_CALL(v); }

  EVE_CALLABLE_OBJECT(abs_t, abs_);
};

inline constexpr auto abs = functor<abs_t>;
```

The eve::abs function

```cpp
// An EVE Callable is a template type
template<typename Options>  //  This may contains some semantic options
struct  abs_t
      : elementwise_callable<abs_t, Options, saturated_option>
//       ^ This base class provides the common implementation
{
  // Notice the high level user facing concept
  template<eve::value T>
    // The implementation is always the same macro call
  constexpr T operator()(T v) const { return EVE_DISPATCH_CALL(v); }

  // This bind the type to the external functions to overload
  EVE_CALLABLE_OBJECT(abs_t, abs_);
};

inline constexpr auto abs = functor<abs_t>;
```

The `eve::abs` function

```
// An EVE Callable is a template type
template<typename Options>  //  This may contains some semantic options
struct  abs_t
      : elementwise_callable<abs_t, Options, saturated_option>
//      ^ This base class provides the common implementation
{
  // Notice the high level user facing concept
  template<eve::value T>
    // The implementation is always the same macro call
  constexpr T operator()(T v) const { return EVE_DISPATCH_CALL(v); }

  // This bind the type to the external functions to overload
  EVE_CALLABLE_OBJECT(abs_t, abs_);
};

// The type is turned into an EVE callable here
inline constexpr auto abs = functor<abs_t>;
```

The Callable Skeletons

Generic Programming is all about automation

- Most EVE functions behave in a similar patterns.
  - Is there any conversion to do?
  - Are we operating over or across SIMD lanes?
- Can we simplify the implementation for contributors?

The EVE Callables

- Encapsulate various level of recurring code patterns.
- Streamline the error-checking process.
- Can be extended in an ***Open/Close*** way.

eve::callable

```cpp
template<template<typename> class Func, typename OptionsValues, typename... Options>
struct callable : decorated_with<OptionsValues, Options...>
{
  template<callable_options O>
  EVE_FORCEINLINE constexpr auto operator[](O const& opts) const;

  template<typename... Args>
  EVE_FORCEINLINE constexpr auto behavior(auto arch, Args&&... args) const
  {
    return Func<OptionsValues>::deferred_call(arch, EVE_FWD(args)...);
  }

  template<typename... Args>
  EVE_FORCEINLINE constexpr auto retarget(auto arch, Args&&... args) const
  {
    return Func<OptionsValues>::deferred_call(arch, this->options(), EVE_FWD(args)...);
  }
};
```

The Callable Skeletons

The Secret Macros

```
#define EVE_CALLABLE_OBJECT(TYPE,NAME)                                          \
template<typename... Args>                                                      \
static EVE_FORCEINLINE constexpr auto deferred_call(auto arch, Args&&...args)  \
-> decltype(NAME(eve::detail::adl_helper, arch, EVE_FWD(args)...))              \
{                                                                               \
  return NAME(eve::detail::adl_helper, arch, EVE_FWD(args)...);                 \
}                                                                               \
/**/

// Somewhere else
namespace eve::detail
{
  template<typename T, typename N, callable_options O>
  wide<T,N> abs_(EVE_REQUIRES(sse2_), O const& o, wide<T,N> const& v) requires x86_abi<abi_t<T, N>>
  {
    // Actual X86 SSE2 and above implementations
  }
}
```

Implementation of Architecture-Optimized Callables

The true power of `if constexpr`

```cpp
template<typename T, typename N, callable_options O>
wide<T,N> abs_(EVE_REQUIRES(sse2_), O const& opts, wide<T, N> const& v)
requires x86_abi<abi_t<T, N>>
{
  constexpr auto c = categorize<wide<T, N>>();

  if        constexpr( match(c, category::unsigned_)  ) return v;
  else if constexpr( match(c, category::float_)      ) return bit_andnot(v, mzero(as(v)));
  else if constexpr( c == category::int64x8          ) return _mm512_abs_epi64(v);
  else if constexpr( match(c, category::size64_)     ) return map(eve::abs, v);
  else if constexpr( c == category::int32x16         ) return _mm512_abs_epi32(v);
  // ... etc

  else if constexpr( c == category::int8x16 )
  {
    if constexpr( current_api >= ssse3 ) return _mm_abs_epi8(v); else return _mm_min_epu8(v, -v);
  }
}
```

Compile-Time Error Management

Reporting Usage Errors

- Concept placement is key.
  - `Check for concepts at the highest API point possible`.
  - If your error is coming from **inside your library**, you failed.
- Preventing cascading errors.
  - `auto` return type can be a false friend.
  - The `ignore trick`.

Reporting Logic Errors

- `static_assert` has to be used `strategically`.
- It breaks SFINAE-friendliness but clearly indicates something's deeply wrong.

What Did We Learn?

Concepts or Not ?

- Use Concepts to dispatch over **type's capabilities**.
- Use `if constexpr` to select implementation details.
- **Filter out bad calls as high in the call tree as possible**.

Errors Reporting

- Concepts failures report API misuses.
- **Static Asserts report logical issues**.
- Try to be as SFINAE-friendly as possible:
  - Other components can inquires about functions' availability.
  - Improve **Dependency** management across libraries.

Going Off-Road

nD Array Design Space

A small sampling

- Owning or non-owning array ?
- Are my dimensions runtime or compile-time? or both?
- Storage order : C, FORTRAN, arbitrary, ...?
- Indexes start at 0? 1? -3? any user value?
- Memory allocation : Allocator-based? Which allocator model? Stack or Heap?

One implementation to rule them all ?

- Monolithic implementation leads to unmaintainable code.
- Arbitrary restrictions on API is not a solution.
- Can we get creative here and get some results?

KIWAKU in a nutshell

- C++20 and onward multi-dimensional containers library.
- Focus on ergonomics and extensibility.
- Start from scratch with a new library to benefit from C++20 novelties.
- **Team Work**: Sylvain Joube, Adrien Henrot, David Chamont, Hadrien Grasland.

API Basics

- Provides containers and traversal algorithms.
- Use Concepts to define user's interface.
- Extensive use compile-time computations and NTTP.

`kwk::view`

- Never owns any memory
- Wraps existing memory in a nD-array like interface.
- Is designed to be as small as possible in 'resting position'

`kwk::table`

- Owns memory allocated via an allocator or on the stack.
- Wraps said memory in a nD-array like interface.
- Is designed to be as small as possible in 'resting position'.

Two sides of the same coin

- A `kwk::table` is a `kwk::view` over the memory it owns.
- Efficient `kwk::view` leads to efficient `kwk::table`.

Constructing KIWAKU containers

- Use user-defined deduction guides as pseudo-function.
- We use **RABERU** named parameters as information carriers.
- At compile-time:
  - A `constexpr` function aggregates those information.
  - Options validity are checked.
  - A NTTP containing the elements to build the type is produced.
  - This NTTP is passed to the container that exploits it to built itself.

Are CTADs Turing-Complete?

- Sort of: CTAD can be used to gather arbitrary information then selecting type.
- Pros: Constructions can be made as intuitive as possible.
- Cons: Unholy Symbol Names.

Constructing KIWAKU containers

Source Editor: C++ source #1

```cpp
#include <kwk/kwk.hpp>
#include <vector>
#include <iostream>

int main()
{
  using namespace kwk;

  std::vector<double> data(18);
```

Algorithms

- Extends classical standard algorithms to nD.
- Reorder parameters so `zip` is replaced by regular variadics.
- Hardware support is done via Execution Policy like objects.

Benefits

- No zip means simpler base code.
- Most core algorithms can be then build from a handful of algorithmic kernels.
- Most implementation turns into tuple manipulation before code generation.

Algorithms

Source Editor: C++ source #1

```cpp
#include <kwk/kwk.hpp>
#include <iostream>

using namespace kwk;

void f(kwk::concepts::container<kwk::_3D> auto& c)
{
  kwk::generate([&](auto i0, auto i1, auto i2)
                {
                  return i0 * 100 + i1 * 10 + i2;
```

Better Execution Policy

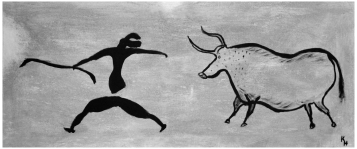
- KIWAKU context acts as execution policy but are extensible.
- Power Users/Developers can add their own.

```
kwk::sycl::gpu.with( [](auto& in, auto& in_out)
{
  // Data "in" is sent to the GPU memory once
  auto pos = kwk::find_if([](auto item) { return std::abs(item) > 10; }, in);

  // "in" still is in GPU memory
  kwk::transform([pos](auto a, auto b) { return a + pos*b; }, in_out, in, in_out);

  // "in_out" sill in GPU memory
  result = kwk::reduce([](auto e1, auto e2) { return e1 * e2 ; }, in_out);
}
, kwk::sycl::in{table1}, kwk::sycl::inout{view1}
);
```

Parting Words



What Did We Learn?

- Incorporate non-member functions into types's interfaces.
- Shift focus from Interface/Class/Object to Concept/Type/Value.
- Emphasis entities collaboration across libraries.
- Simple components are key to efficient design.

Building on SOLID Foundations

What Did We Learn?

- Incorporate non-member functions into types's interfaces.
- Shift focus from Interface/Class/Object to Concept/Type/Value.
- Emphasis entities collaboration across libraries.
- Simple components are key to efficient design.

The SOLID Principles for OOP

- **Single Responsibility**: *A class should have one and only one job.*
- **Open/Close**: *Software entities should be open for extension but closed for modification.*
- **Substitution Principle**: *Derived types should complement the base object behaviour.*
- **Interface Segregation**: *Many specific client interface are better than one general purpose interface.*
- **Dependency Inversion**: *Entities must depend on abstractions, not on concretions.*

Building on SOLID Foundations

What Did We learn?

- Incorporate non-member functions into types's interfaces.
- Shift focus from Interface/Class/Object to Concept/Type/Value.
- Emphasis entities collaboration across libraries.
- Simple components are key to efficient design.

The SOLID Principles for Generic Programming

- **Single Responsibility**: *A **type** or **function** should have one and only one job.*
- **Open/Close**: *Software entities extension is a **core design choice**.*
- **Substitution Principle**: ***Refined interface** should complement the **base concept behaviour**.*
- **Interface Segregation**: *Many specific purpose **libraries** are better than a general purpose **library**.*
- **Dependency Inversion**: *Entities must depend on **concepts**, not on **types**.*

Go Out and Build Libraries

Maximize Interoperability!

- Languages evolved to support more generic composition mechanisms.
- Don't make QoL libraries go dark.
- **C++: We need a standard package manager!**

Make It Known!

- The community needs more venue to **publish** and **promote** tools.
- Motivate your students and collaborators to communicate about their libraries.

Shoot Out

- Denis Yaroshesky, Jean Thierry Lapresté, Alexis Aune
- Sylvain Joube, Adrien Henrot, David Chamont, Hadrien Grasland

Thanks for your attention !