std::simd

Dr. Matthias Kretz

GSI Helmholtz Centre for Heavy Ion Research

WG21 LEWG review | 2023-02-08
Outline

Introduction

A Data-Parallel Type

std::simd
Introduction

A Data-Parallel Type

std::simd

multiple operations in one instruction

operation often a C++ operator, e.g. +, −, *

instruction one step of machine code

(basic idea: a CPU core executes instructions serially in the specified order)

SIMD – Single Instruction Multiple Data

x₀ + y₀
x₁ + y₁
x₂ + y₂
x₃ + y₃

↷

x₀ x₁ x₂ x₃
+y₀ y₁ y₂ y₃
Introduction

A Data-Parallel Type

std::simd

Data-Parallelism

Data Parallel

- same code
- different data
- may execute in parallel

Example

```cpp
for (auto &x : data) {
    x = transform(x); // transform is a pure function
}
```
We have the `unseq` and `par_unseq` execution policies.

Program-defined code executed from parallel algorithms exposes “vector semantics”:
- different from C++’s sequenced-before semantics
- access to globals may have surprising results
- thread synchronization has undefined behavior
- exceptions have undefined behavior
- no I/O (e.g. “printf debugging”)

Implementations might need all called functions to be inline to actually perform vectorization

Control-flow (`break`, `return`, ...) often inhibits vectorization

Loop based vectorization provides no intuition or support with data structures.
Data-Parallel Types

One variable stores $\mathcal{W}_T$ values. ($\mathcal{W}$ for “width”)
One operator signifies $\mathcal{W}_T$ operations (element-wise).

```cpp
int x = 0;
x += 1;
```

vs.

```cpp
std::simd<int> x = 0;
x += 1;
```
Introduction

A Data-Parallel Type

std::simd

Fundamental Type

abstracts

scalar

Registers & Instructions

Programming Language

abstracts

Computer

SIMD

Registers & Instructions

Matthias Kretz  WG21 LEWG review | 2023-02-08

GSI Helmholtz Centre for Heavy Ion Research
Introduction

A Data-Parallel Type

std::simd

Fundamental Type
- abstracts
  - scalar Registers & Instructions

Programming Language
- abstracts

Computer

simd<T>
- abstracts

SIMD Registers & Instructions

Matthias Kretz  WG21 LEWG review | 2023-02-08

GSI  Helmholtz Centre for Heavy Ion Research
• In contrast to SIMT and vector loops, `std::simd` makes the chunk size a constant expression.
• Operation on a larger index space than $\mathcal{W}_T$ requires a loop and/or multiple threads.
• Clear separation of serial, SIMD-parallel, and thread parallel execution.
• No restriction on I/O, exceptions, function calls, and synchronization.
• API & ABI for vectorization across multiple translation units (and library boundaries).
  • The `std::simd` ABI could be the ABI for function calls from `unseq` loops.
Example

One multiplication:

```cpp
float f(float x) {
    return x * 2.f;
}
```

https://godbolt.org/z/1TY9jbqqj

Wₜ multiplications in parallel:

```cpp
std::simd<float> f(std::simd<float> x) {
    return x * 2.f;
}
```
Data-Parallel Conditionals

Example

One compare and 0 or 1 assignment:

```cpp
float f(float x) {
  if (x > 0.f) { x *= 2.f; }
  return x;
}
```

\(\mathcal{N}_T\) compares and 0–\(\mathcal{N}_T\) assignments in parallel:

```cpp
std::simd<
float
>
f(std::simd<
float
>x) {
  x = std::conditional_operator(
    x > 0.f, x * 2.f, x);
  return x;
}
```

- Compares yield \(\mathcal{N}_T\) boolean answers
- Return type of compares: `std::simd_mask<T, Abi>`
- Reduction functions: `all_of`, `any_of`, `none_of`
- `simd` code typically uses no/few branches, relying on masked assignment instead
**Example**

**TS syntax**

```cpp
template <typename T> T f(T x) {
    stdx::where(x > 0.f, x) *= 2.f;
    return x;
}
```

f is “vectorizable” in the sense that it can be specialized for float and `stdx::simd<float>`.

**Preferred C++26 syntax**

```cpp
template <typename T> T f(T x) {
    return x > 0.f ? x * 2 : x;
}
```

So much simpler and clearer. Easy to write `simd`-compatible code before ever using `simd`.

Generic code! 🔥 (My GCC can do it 😊.)
The semantics of data-parallel types teach developers to design scalable and portable parallelization.

- Target-dependent $\mathcal{V}$
- Conversions between scalar and vector objects
- Conditional assignment instead of branching

- It becomes clear that data structures are the main challenge
- Translating an inherently data-parallel algorithm to data-parallel types is often trivial
- However, where do \texttt{simd} objects come from, and where can you put them?
- With vector loops and SIMT it is easy... to write inefficient memory access patterns.
- Using SIMD types makes the design challenges wrt. efficient vectorization obvious
- Subsequent designs can profit from this experience
The semantics of data-parallel types teach developers to design scalable and portable parallelization.

- It becomes clear that data structures are the main challenge
  - Translating an inherently data-parallel algorithm to data-parallel types is often trivial
  - However, where do \texttt{simd} objects come from, and where can you put them?
  - With vector loops and SIMT it is easy ...

- Using SIMD types makes the design challenges wrt. efficient vectorization obvious
- Subsequent designs can profit from this experience
The semantics of data-parallel types teach developers to design scalable and portable parallelization.

- It becomes clear that data structures are the main challenge
  - Translating an inherently data-parallel algorithm to data-parallel types is often trivial
  - However, where do `std::simd` objects come from, and where can you put them?
  - With vector loops and SIMT it is easy ... to write inefficient memory access patterns.
- Using SIMD types makes the design challenges wrt. efficient vectorization obvious
- Subsequent designs can profit from this experience
The semantics of data-parallel types teach developers to design scalable and portable parallelization.

- It becomes clear that data structures are the main challenge
  - Translating an inherently data-parallel algorithm to data-parallel types is often trivial
  - However, where do `std::simd` objects come from, and where can you put them?
  - With vector loops and SIMT it is easy ... to write inefficient memory access patterns.
- Using SIMD types makes the design challenges wrt. efficient vectorization obvious
- Subsequent designs can profit from this experience
Abstract

- Conceptually: SIMD types express data-parallelism.
- Wrong mindset: SIMD types are specific SIMD registers.

Which is why I like to call them “data-parallel types”.
There are implementations
...and lots of existing practice

- `std::experimental::simd` in libstdc++ since GCC 11
- `vir::stdx::simd` at https://github.com/mattkretz/vir-simd/
- `std::(experimental::)simd` implementation from Intel in progress
- `std::simd prototyping` https://github.com/mattkretz/simd-prototyping/

more existing practice
- Agner Fog's Vector Types
- E.V.E.
- xsimd
- Vc
- ...
A Data-Parallel Type

1 template <typename T, typename Abi = ...>
2 class simd;
3
4 template <typename T, typename Abi = ...>
5 class simd_mask;

- T must be a “vectorizable” type (arithmetic except bool)
  Note: Daniel Towne wants to add std::complex, I plan to add enums, and with
  reflection I’ll look into UDTs.

- simd<T> behaves just like T (as far as is possible)
- simd_mask<T> behaves like bool
  In contrast to bool, there are many different mask types:
  - storage: bit-masks vs. element-sized masks (and vir-simd uses array of bool),
  - SIMD width simd::size

- Abi determines width and ABI (i.e. how parameters are passed to functions)
- The TS uses the wrong default for the ABI tag (my strong opinion, to be fixed for C++26).
- The TS gives you the lowest common denominator for all possible implementations of the target architecture.
- So you want to **always use stdx::native_simd<T>** instead. (This will be std::simd<T>).
- **native_simd** sets the ABI tag to the widest efficient \( \mathcal{W}_T \) for your `-march=` setting. It also influences the representation of simd_mask (i.e. the `sizeof` may be very different).
- Note that therefore std::simd ABI depends on `-m` flags!
Constructors (simplified)

```cpp
template <typename T, typename Abi = ...>
class simd {
    simd() = default;
    simd(T);
    simd(contiguous_iterator auto const&, Flags);
    simd(Generator);
}
```

- The defaulted `default` constructor allows uninitialized and zero-initialized objects.
- The `broadcast` constructor initializes all elements with the given value.
  - Requires value-preserving conversion (P2509R0)
- The `load` constructor reads $N_T$ elements starting from the given address.
  - Flags provides a hint about alignment (and can be extended to do more: in Vc it controls streaming loads & stores, prefetching; P1928R3 suggests control over conversions)
- The `generator` constructor initializes each element via the generator function.
  - The generator function is called with `std::integral_constant<std::size_t, i>`, where $i$ is the index of the element to be initialized.
Constructor examples

1. `stdx::native_simd<int> x0; // uninitialized`
2. `stdx::native_simd<int> x1 {}; // zero-initialized`
3. `stdx::native_simd<int> x2 = 1; // all elements are 1`
4. `stdx::native_simd<int> x3 (addr, stdx::vector_aligned); // load from aligned address`
5. `stdx::native_simd<int> iota([](int i) { return i; }); // [0, 1, 2, 3, 4, ...]`
You need to interact with the world somehow...

```cpp
void f(std::vector<float>& data) {
    using V = stdx::native_simd<float>;
    for (std::size_t i = 0; i < data.size(); i += V::size()) {
        V v(&data[i], stdx::element_aligned);
        v = sin(v);
        v.copy_to(&data[i], stdx::element_aligned);
    }
}
```

- The member functions `copy_from` and `copy_to` allow “conversion” from/to arrays of `T`.
- The above applies the sine to all values in `data`.
- Don’t be afraid that this copy costs performance.
  - Consider loading from memory into a register / storing from register into memory.
  - This is a necessary cost that always happens anyway.
- There’s a bug, though…
You need to interact with the world somehow...

```cpp
void f(std::vector<float>& data) {
  using V = stdx::nativesimd<float>;
  for (std::size_t i = 0; i < data.size(); i += V::size()) {
    V v(&data[i], stdx::element_aligned);
    v = sin(v);
    v.copy_to(&data[i], stdx::element_aligned);
  }
}
```

- The member functions `copy_from` and `copy_to` allow “conversion” from/to arrays of `T`.
- The above applies the sine to all values in `data`.
- Don’t be afraid that this copy costs performance.
  - Consider loading from memory into a register / storing from register into memory.
  - This is a necessary cost that always happens anyway.
- There’s a bug, though...
Introduction

A Data-Parallel Type

std::simd

Loads & stores (fixed)

You often need an “epilogue”:

```cpp
void f(std::vector<float>& data) {
  using V = stdx::native_simd<float>;
  std::size_t i = 0
  for (; i + V::size() <= data.size(); i += V::size()) {
    V v(&data[i], stdx::element_aligned);
    v = sin(v);
    v.copy_to(&data[i], stdx::element_aligned);
  }
  for (; i < data.size(); ++i) {
    data[i] = std::sin(data[i]);
  }
}
```

- Having to write the epilogue every time is error prone.
- The TS does not come with supporting code, but P0350 proposes useful higher-level API.
Subscripting

Loads & stores are great, but sometimes you just want to access it like an array.

```cpp
void f(stdx::native_simd<float> x) {
    for (std::size_t i = 0; i < x.size(); ++i) {
        x[i] = foo(x[i]);
        auto ref = x[i];
        ref = foo(x[i]); // ERROR doesn't compile
        x[i] = float(ref); // OK
    }
}
```

- **non-const subscripting returns a `simd::reference`**
- this type implements all non-const operators, i.e. (compound) assignment, increment and decrement, and also swap.
- all of the above functions are rvalue-ref qualified, i.e. are only allowed on temporaries
- What we all actually expect would be a *decay* of the reference proxy to the element type. Another paper I still have to write and defend in the committee.
- the conversion operator is not ref qualified
This is what you all came for, I guess

```cpp
void f(stdx::native_simd<float> x, stdx::native_simd<float> y) {
    x += y; // \( \mathcal{W}_\text{float} \) additions
    x = sqrt(x); // \( \mathcal{W}_\text{float} \) square roots
    ... // etc. all operators and <cmath>
}
```

- Operations act element-wise
- Speed-up is often a factor of \( \mathcal{W}_T \), but may be less, depending on hardware details.
Same for compares

```cpp
void f(stdx::native_simd<float> x, stdx::native_simd<float> y) {
  if (x < y) {} // nonono, you don’t write ’if (truefalsefalse)’ either
  where(x < y, x) = y; // x = y but only for the elements where x < y
  if (all_of(x < y)) {} // this makes sense, yes
}
```

- Comparisons return a `simd_mask`.
- `simd_mask` is not convertible to `bool`.
- `simd_mask` can be reduced to `bool` via `all_of`, `any_of`, or `none_of`.
- The SIMT model does not expose the nature of its `if` statements in code. It seems like branching but it isn’t really. With `std::simd` it is explicit.