

# FloatX: Floating Point Type Emulation Library

Goran Flegar   Cristiano Malossi   Enrique S. Quintana-Ortí  
Florian Scheidegger

23 November 2017

# Why FloatX?

- ▶ Transprecision hardware under development
- ▶ Experiment with transprecision algorithms
  - ▶ Even if hardware **not** available
  - ▶ Need emulation library: **FloatX** (Float Extended)

# Design principles

- ▶ Efficiency
- ▶ Ease of use

## As efficient as possible

- ▶ enables larger simulations (e.g. DNNs)
- ▶ use floating point types supported by hardware as “backend”,  
reduce precision after every operation
- ▶ avoid using extra memory
  - ▶ size of emulated type  $\leq$  size of backend type
- ▶ no overhead if emulated type = hardware-supported type
  - ▶ e.g. floatx<11, 52> as efficient as double
- ▶ CUDA support

## As easy to use as native types

- ▶ Arithmetic and relational operations, assignments
  - ▶ a and b FloatX numbers
  - ▶  $a + b$ ,  $a / b$ , ...
  - ▶  $a < b$ ,  $a > b$ , ...
  - ▶  $a = b$ ,  $a += b$ ,  $a -= b$ , ...

# As easy to use as native types

- ▶ Arithmetic and relational operations, assignments
  - ▶ a and b FloatX numbers
  - ▶  $a + b$ ,  $a / b$ , ...
  - ▶  $a < b$ ,  $a > b$ , ...
  - ▶  $a = b$ ,  $a += b$ ,  $a -= b$ , ...
- ▶ Interoperability between different emulated types
  - ▶ a and b FloatX numbers of **different** precision
  - ▶ arithmetic, relational operations, assignments
  - ▶ generalization of standard “type propagation” rules (e.g.  $a = b + c$ )
    - ▶ if a and b are of the same type,  $a == \text{decltype}(a)(c)$
    - ▶ if a and c are of the same type,  $a == \text{decltype}(a)(b) + c$
  - ▶ *common type*: `common_type(S, T)`
    - ▶ the least precise (smallest) type which is at least as precise as both S and T

# As easy to use as native types

- ▶ Arithmetic and relational operations, assignments
  - ▶ a and b FloatX numbers
  - ▶  $a + b$ ,  $a / b$ , ...
  - ▶  $a < b$ ,  $a > b$ , ...
  - ▶  $a = b$ ,  $a += b$ ,  $a -= b$ , ...
- ▶ Interoperability between different emulated types
  - ▶ a and b FloatX numbers of **different** precision
  - ▶ arithmetic, relational operations, assignments
  - ▶ generalization of standard “type propagation” rules (e.g.  $a = b + c$ )
    - ▶ if a and b are of the same type,  $a == b + \text{decltype}(a)(c)$
    - ▶ if a and c are of the same type,  $a == \text{decltype}(a)(b) + c$
  - ▶ *common type*: `common_type(S, T)`
    - ▶ the least precise (smallest) type which is at least as precise as both S and T
- ▶ Interoperability between emulated and native types
  - ▶  $a = a + 3$ ,  $a -= 3.5$ , `auto x = a + 3.5(decltype(x)?)`

## Language of choice: C++

- ▶ 0-overhead abstractions
- ▶ high efficiency
- ▶ operator overloading
- ▶ powerful compile-time type manipulation via template metaprogramming



## Example

```
floatx<7, 12> a = 1.2, b = 3.2; // 7 exp. bits, 12 sig. b
floatx<10, 9> c;
float d = 3.2;
double e = 5.2;

std::cin >> c;

c = a + b;    // decltype(a + b) == floatx<7, 12>
bool t = a < b;

a += c;

d = a / c;    // decltype(a / c) == floatx<10, 12>
e = c - d;    // decltype(c - d) == floatx<10, 23>
c = a * e;    // decltype(a * e) == floatx<11, 52> ?

std::cout << c;
```

# Current status

## What works

- ▶ Arithmetic, relational ops, assignment
  - ▶ Same floatx type, different floatx types, floatx and native type
- ▶ Round to nearest, tie to even rounding policy

## In progress

- ▶ *Backend type* other than `double`
- ▶ Optimal performance for equivalents of native types
- ▶ CUDA support

## Bonus

- ▶ Precisions fixed at compile time *cripple* experimentation
- ▶ *runtime* version of `floatx<Exp, Sig>`

```
floatxr<> ra(5, 7, 3.2);  
std::cout << ra + 3.4 << std::endl;  
floatx<8, 9> b;  
std::cout << ra - b << std::endl;  
ra.set_precision(2, 8);
```

- ▶ trade-off: higher memory & instruction requirements

# FlexFloat vs FloatX

## FlexFloat

- ▶ C library (also callable from other languages)
- ▶ Function-based syntax
- ▶ Enables access to transprecision hardware if available

## FloatX

- ▶ C++ library (cannot call from other languages, but better performance thanks to heavy inlining)
- ▶ Operator-based syntax, behaviour of builtin types
- ▶ Cannot access transprecision hardware directly (but can use FlexFloat as backend type!)