FP

Configurable floating point conversions and operations for ML in Rust.

Nicholas Teague, January 2022

Background

- The IEEE Working Group P3109 (Arithmetic Formats for ML) has recently been working towards a new standard for an 8-bit floating point format designed around the needs of machine learning.
- The initiative helped the developer recognize the potential for a formatting convention agnostic library that could perform various operations using a configurable formatting basis.
    - For example the same library and conventions could be applied for an 8-bit floating point, 4-bit, 128-bit, etc.
- The Rust programming language was adopted to due to reputation as ~best in class performance with more robust security than C.
- Licensed using BSD-3-Clause and current draft shared on GitHub.

Developer

Nicholas Teague

- Founder and developer of Automunge, a dataframe preprocessing platform
- Author of “From the Diaries of John Henry” essay collection :)
- Workshop papers at top tier conferences like NeurIPS, ICLR, ICML
- Professional background in engineering, technical sales, etc
- University of Florida graduate (Masters in engineering, MBA)

Current Status

- “Lightly validated” demonstrations of basic mathematic operations in a Rust notebook
- Extensions to some common ML operations including dot product
- Root primitives of bitwise operations aggregated to higher level mathematics
    - I speculate this convention may eventually enable a reduced chip instruction set.
- Represents numbers as boolean vectors, e.g. `vec![true, true, false, true]`
- Special encodings are optional and custom definable for a given precision
- Haven’t yet done performance benchmarking, relying on reputation of Rust language
- A version of posits format conversions is also implemented

Potential Extensions

- Subnormal support not yet implemented
- Range of special encodings currently limited (+/-inf, nan, +/-0)
- Does not yet offer flagging, only propagation of edge cases by special encodings
- Special encodings not yet supported within mixed precision operations
- A foundation for stochastic rounding in place, some extensions likely possible
- Pending implementing wrappers for e.g. Cuda, Pytorch, etc.

Architecture

Boolean Vectors

- The vector is a fundamental Rust data type which allows variable number of entries of a common data type
- We represent numbers with the Vec!<bool> type for boolean entries as the boolean data type has a single bit representational footprint
- Different functions may interpret a boolean vector as integer or floating point
- Note that when appending an entry to end of a rust vector the other entries are static, however when appending to front (e.g. for a right shift) the whole vector is shifted in memory
    - There is a variation on rust vector called VecDeque which allows static entries when appending to front or back, needs benchmarking

Bitwise Operations

- bitwise_and
- bitwise_or
- bitwise_xor
- bitwise_not
- bitwise_leftshift
- bitwise_rightshift
- Some variations possible based on different rust conventions for mutable variables

Integer Operations

- integer_convert(.) - for use to validate the integer arithmetic operations
- integer_revert(.) - translates a u32 to Vec<bool>
- integer_add(.)
- integer_increment(.) - increments vector encoding by 1
- integer_decrement(.) - decrements vector encoding by 1
- integer_subtract(.)
- integer_multiply(.)
- integer_greaterthan(.)
- integer_divide(.)

- Similar operations also available for “signed integer” (assuming leading sign bit)
- **hat tip to a few tutorials from iq.opengenus.org, particularly for add and subtract

Fp basic operations

- fp_add(.) returns without rounding, same expwidth basis
- fp_add_variable_exp(.) - returns without rounding, variable expwidth basis
- fp_subtract(.) - (a wrapper for fp_add)
- mantissa_multiply(.) - support function for fp_multiply
- fp_multiply(.)
- fp_multiply_variable_exp(.)
- fp_divide(.)
- fp_abs(.)
- fp_greaterthan(.)
- fp_equals
- (And a few more related support functions)

Fp advanced operations

- fp_match_precision(.) - converts vector_one precision to a given expwidth and bitwidth
- fp_convert_exp_precision(.) - support function, expands or decreases exponent
- fp_convert_mantissa_precision(.) - support function, expands or decreases mantissa, rounding applied
- fp_increment_mantissa(.) - support function
- fp_compare(.)
- fp_fma(.)
- fp_set_scaler(.)
- fp_dot(.)

Floating Point Specification

- For variations on the conventional 754 floating point with varied width, typically specification resembles something like “1_4_3”, where 1 is the sign bit, 4 is the exponent, and 3 is the significand
- We can simplify to just specifying the exponent width, with the rest inferred by assuming first bit is always sign, trailing bits are significand
- We currently apply a minimal data type (u8) for exponent width specification, a tradeoff is limiting application to ultra high precision applications
- Rust has a variable bit width data integer type (usize), a future extension may replace u8 for this use case
- Some other cases may benefit from additional specification, for example 754 assumes a single convention for exponent bias, assumes a constant bit width, etc

Special Encodings

- Optional special encodings support for floating point relies on assignment of vector configuration towards (currently a limited) set of special encodings via a rust “struct” data type passed through operations
    - +/- inf, NaN, +/-0
- Note that zero is treated as a special encoding in current form (since don’t yet have subnormal support)
- A limitation of this approach is specification will generally be associated with a specific bit width, we have some ideas of how to extend to mixed precision support, pending implementation

Operations

- **Compare**
    - Compare two FP vectors, potentially of different exponent and significand widths, and determines if within a specified tolerance (e.g. to determine if a==b)
- **Match Precision**
    - Convert exponent and significand width one one vector to match that of another
    - In cases of reduced significand width, this is where rounding occurs
- **FMA**
    - “Fused multiply add”, accepts three input values a, b, c and returns c+=a*b
    - As implemented assumes a and b in common precision, c returned with infinite precision (meaning no rounding applied, exponent expanded as needed to retain information)
    - (Assumes any scaling is performed externally.)
- **Dot**
    - Resembles FMA, but instead of c+=a*b (where a and b are distinct values), accepts a and b as as equivalent length sets of distinct values, returns c+= [a1, a2, a3] dot [b1, b2, b3]

Rounding

- Currently rounding is implemented in the “match precision” operation
- The special encodings struct includes option to select between nearest versus stochastic rounding
- The stochastic rounding currently uses the “rand” crate for sampling
- The sampling is weighted by configuration of the dropped trailing bits

Validations

- Conversions:
  - We validate conversions to and from the bool vector form by generating the set of all configurations associated with a bit width and confirming value retention in forward and backward translation
- Operations:
  - Operations are validated by way of randomly sampling input vectors and comparing output to equivalent operations conducted in f32