MicroFloatingPoints.jl: providing very small IEEE 754-compliant floating-point types

14 August 2024

Summary

The IEEE 754 standard defines the representation and the properties of the floating-point numbers used as surrogates for reals in most applications. Usually, programming languages only support the two formats —corresponding to different ranges and precisions— implemented in most hardware: `Float32` on 32 bits, and `Float64` on 64 bits (aka, respectively, `float` and `double` in programming languages of the C family). Machine learning, Computer Graphics, and numerical algorithms analysis all have a need for smaller formats (minifloats), which are often neither supported in hardware, nor are they available as established types in programming languages. The `MicroFloatingPoints.jl` Julia library offers a parametric type that can be instantiated to compute with IEEE 754-compliant floating-point numbers with varying ranges and precisions (up to and including `Float32`). It also provides the programmer with various means to visualize what is computed.

Statement of need

Proving the properties of numerical algorithms involving floating-point numbers can be a very challenging task. Insight can often be gained by executing systematically the algorithm under study for all possible inputs. There are, however, too many values to consider with the classically available types `Float32` and `Float64`. Hence the need for libraries that offer smaller IEEE 754-compliant types to play with. SIPE (Lefevre 2013), FloatX (Flegar et al. 2019), and CPFloat (Fasi and Mikaitis 2023), to name a few, are such libraries. However, being written in languages such as C or C++, they lack the interactivity and tight integration with graphical facilities that can be obtained from using script languages such as Julia. `MicroFloatingPoints.jl` is a Julia library that fills this gap by offering a parametric type `Floatmu` that can be instantiated to simulate in software small floating-point types: `Floatmu{8,23}` is a type using 8 bits to represent the exponent and 23 bits for the fractional part, which is equivalent to

`Float32`; `Floatmu{8,7}` is equivalent to the Google Brain `bfloat16` format, … `MicroFloatingPoints.jl` was, for example, instrumental in our understanding of the flaws of the algorithms computing random floating-point numbers by dividing a random integer by a constant (Goualard 2020): being able to execute the algorithms for all possible inputs and to quickly display the results graphically in an interactive and integrated environment such as the one provided by Jupyter gave us the impetus to demonstrate rigorously that such procedures cannot ensure an even distribution of the bits in the fractional parts of the random floats, rendering them useless for applications such as *differential privacy* (Dwork 2006; Mironov 2012). Figure 1 exemplifies the kind of result easily obtainable with `MicroFloatingPoints.jl`: using the type `Floatmu{7,16}`, we systematically divide *all* integers in $[0, 2^{17} - 1]$ by 2^{17} to obtain floating-point numbers in $[0, 1)$. The picture shows the probability of being `1` for each bit of the fractional part of the result.

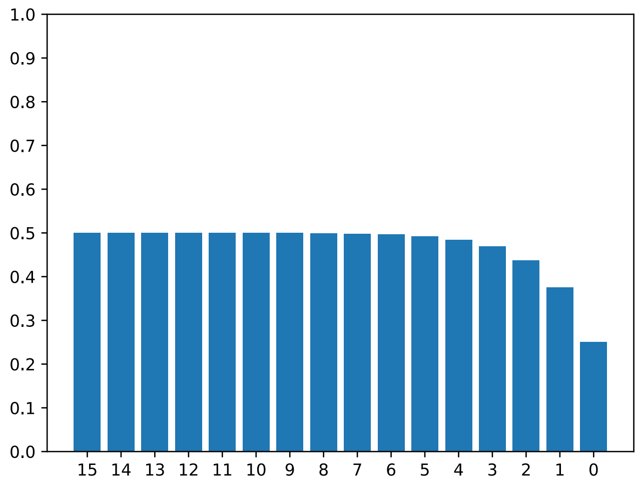

Figure 1: Probability of being 1 for each bit of the fractional part of a `Floatmu{7,16}` when dividing each integer in $[0, 2^{17} - 1]$ by 2^{17}.

Small floating-point formats are also increasingly used in machine learning algorithms, where the precision and range are less important than the capability to store and manipulate as many values as possible. There are already some established formats implemented in hardware (e.g., IEEE 754 `Float16` —available natively in Julia— and Google Brain `bfloat16` —provided by the Julia Package `BFloat16s.jl`). There is, however, still a need for more flexibility to test the behavior of well-known and new algorithms with varying precisions and ranges. The parametric type of `MicroFloatingPoints.jl` can be put to good use there too, and has already been for the study of training neural networks (Arthur et al. 2023).

References

Arthur, Benjamin J., Christopher M. Kim, Susu Chen, Stephan Preibisch, and Ran Darshan. 2023. “A Scalable Implementation of the Recursive Least-Squares Algorithm for Training Spiking Neural Networks.” *Frontiers in Neuroinformatics* 17 (June). https://doi.org/10.3389/fninf.2023.1099510.

Dwork, Cynthia. 2006. “Differential Privacy.” In *Automata, Languages and Programming*, edited by Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, 1–12. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. https://doi.org/10.1007/11787006_1.

Fasi, Massimiliano, and Mantas Mikaitis. 2023. “CPFloat: A C Library for Simulating Low-Precision Arithmetic.” *ACM Transactions on Mathematical Software* 49 (2): 18:1–32. https://doi.org/10.1145/3585515.

Flegar, Goran, Florian Scheidegger, Vedran Novaković, Giovani Mariani, Andrés E. Tom´s, A. Cristiano I. Malossi, and Enrique S. Quintana-Ortí. 2019. “FloatX: A C++ Library for Customized Floating-Point Arithmetic.” *ACM Transactions on Mathematical Software* 45 (4): 1–23. https://doi.org/10.1145/3368086.

Goualard, Frédéric. 2020. “Generating Random Floating-Point Numbers by Dividing Integers: A Case Study.” In *Proceedings of the International Conference on Computational Science*, edited by V. Krzhizhanovskaya, 12138:15–28. Lecture Notes in Computer Science. Amsterdam, The Netherlands: Springer. https://doi.org/10.1007/978-3-030-50417-5_2.

Lefevre, Vincent. 2013. “SIPE: Small Integer Plus Exponent.” In *Proceedings of the 2013 IEEE 21st Symposium on Computer Arithmetic*, 99–106. ARITH ’13. USA: IEEE Computer Society. https://doi.org/10.1109/ARITH.2013.22.

Mironov, Ilya. 2012. “On Significance of the Least Significant Bits for Differential Privacy.” In *Proceedings of the 2012 ACM Conference on Computer and Communications Security - CCS ’12*, 650–61. Raleigh, North Carolina, USA: ACM Press. https://doi.org/10.1145/2382196.2382264.