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

5 September 2024

Summary

The IEEE 754 standard defines the representation and the properties of the floating-point numbers used as surrogates for reals in computer programs. Most programming languages only support the 32-bit (`Float32`) and 64-bit (`Float64`) formats implemented in hardware. Machine learning, Computer Graphics, and numerical algorithms analysis all have a need for smaller formats, 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 many 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 need 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, ...

A quick tour

To obtain a (pseudo-)random float in the domain $[0, 1)$ for a floating-point format with a p-bit significand, many libraries simply divide a pseudo-random integer taken from $[0, 2^p - 1]$ by 2^p (Goualard 2020). Does it ensure an even distribution of the bits in the fractional parts of the random floats, as required by applications such as *differential privacy* (Dwork 2006; Mironov 2012)? This can be systematically and quickly checked for a small floating-point format. We start by loading `MicroFloatingPoints` and `PyPlot` (alternatively, `PythonPlot` could also be used) for the graphics:

```
using MicroFloatingPoints, PyPlot
```

and we define a new IEEE 754-compliant floating-point type, say, with 7 bits for the exponent and 9 bits for the significand (i.e., 8 bits for the fractional part):

```
E = 7 # Size of the exponent part
f = 8 # Size of the fractional part
p = f+1 # Size of the significand
MuFP = Floatmu{E,f} # The new format
```

We now divide all integers in $[0, 2^p - 1]$ by 2^p to obtain a `MuFP` float, for which we record the value of each bit of its fractional part. An array `T` with `f` cells will accumulate the number of occurrences of a '`1`' over all floats produced (specifically, `T[i]` will contain the number of times the `(f-i)`-th bit of the fractional part was a `1` so far —with the 0-th bit being the rightmost one, as usual).

```
T = zeros(f)
for v in 0:(2^p-1)
    d = MuFP(v)/2^p
    fpart = bitstring(d)[2+E:end] # Isolating the fractional part
    for j in 1:f
        global T[j] += Int(fpart[j] == '1')
    end
end
```

We now normalize to $[0, 1]$ the number of occurrences, and display the results with a bar plot (Figure 1).

```
nT = map(x -> x/2^p,T)
plt.bar(1:f,nT)
plt.xticks(1:f,reverse(map((x)->string(Int(x-1)),1:f)))
plt.yticks(0:0.1:1)
plt.show()
```

We were expecting a probability of 0.5 for each bit of the fractional part to be `1`. The actual plot shows that it is not the case and that the probability decreases for the lowest bits. It is very easy to check that behavior for a larger type by, e.g., changing the value of `f` to `16` in our previous code:

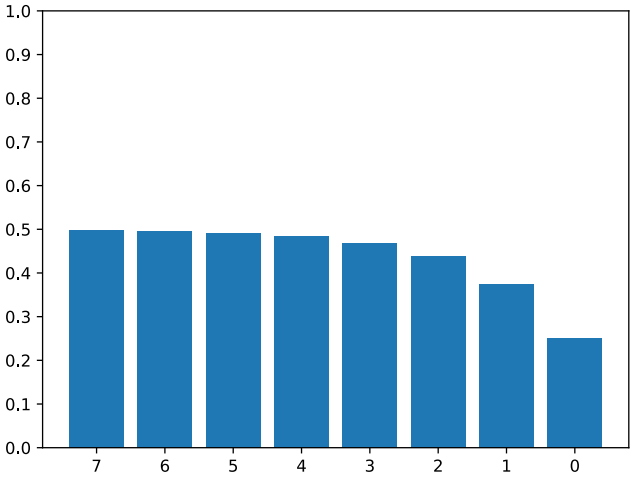

Figure 1: Probability of being 1 for each bit of the fractional part of a `Floatmu{7,8}` when dividing each integer in $[0, 2^9 - 1]$ by 2^9.

```
f = 16
```

The result in Figure 2 shows the same behavior for the larger type `Floatmu{7,16}`.

Limitations

At present, all computations are performed in double precision (`Float64` type), then correctly rounded to the `Floatmu{}` format chosen. As long as the precision of the `Floatmu{}` type is at most half the one of `Float64`, there is no *double rounding* issue (Martin-Dorel, Melquiond, and Muller 2013), and any final result obtained in that way is exactly the same as the one we would obtain by computing directly with the `Floatmu{}` precision (Rump 2016).

Small floating-point formats are 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 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). However, since it represents all floating-point formats by a pair of 32 bit integers, it cannot

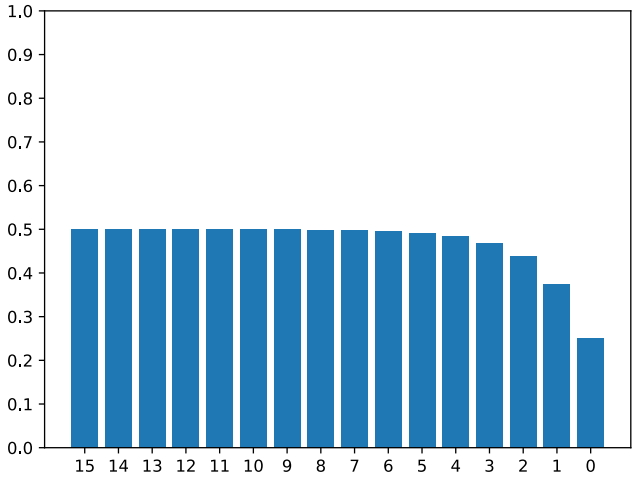

Figure 2: Probability of being 1 for each bit of the fractional part of a `Floatmu{7,16}` when dividing each integer in $[0, 2^{17} - 1]$ by 2^{17}.

compete with more specialized packages for applications that require storing and manipulating massive amounts of numbers. For such use cases, it should therefore be confined to preliminary investigations with more limited amounts of data.

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/10.114

5/3368086.

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.

Martin-Dorel, Érik, Guillaume Melquiond, and Jean-Michel Muller. 2013. “Some Issues Related to Double Rounding.” *BIT Numerical Mathematics* 53 (4): 897–924. https://doi.org/10.1007/s10543-013-0436-2.

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.

Rump, Siegfried M. 2016. “IEEE754 Precision-k Base-$\Beta$ Arithmetic Inherited by Precision-m Base-$\Beta$ Arithmetic for $k < m$.” *ACM Transactions on Mathematical Software* 43 (3): 20:1–15. https://doi.org/10.1145/2785965.