The Journal of Open Source Software

Universal Numbers Library: Multi-format Variable Precision Arithmetic Library

E. Theodore L. Omtzigt [1*] **and James Quinlan** [2*¶]

1 Stillwater Supercomputing, Inc, USA **2** School of Mathematical and Physical Sciences, University of New England, USA ¶ Corresponding author * These authors contributed equally.

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

Universal Numbers Library, or *Universal* for short, is a self-contained C++ header-only template library that contains implementations of many number representations and standard arithmetic on arbitrary configuration integer and real numbers (Omtzigt et al., 2020). In particular, the library includes integers, decimals, fixed-points, rationals, linear floats, tapered floats, logarithmic, SORNs, interval, level-index, and adaptive-precision binary and decimal integers and floats, each offering a verification suite.

The primary pattern using a posit number type as example, is:

```
#include <universal/number/posit/posit.hpp>

template<typename Real>
Real MyKernel(const Real& a, const Real& b) {
    return a * b;  // replace this with your kernel computation
}

constexpr double pi = 3.14159265358979323846;

int main() {
    using Real = sw::universal::posit<32,2>;

    Real a = sqrt(2);
    Real b = pi;
    std::cout « ”Result: ” « MyKernel(a, b) « std::endl;
}
```

Universal delivers software and hardware co-design capabilities to develop low and mixed-precision algorithms for reducing energy consumption in signal processing, Industry 4.0, machine learning, robotics, and high-performance computing applications (Omtzigt & Quinlan, 2022). The package includes command-line tools for visualizing and interrogating numeric encodings, an interface for setting and querying bits, and educational examples showcasing performance gain and numerical accuracy with the different number systems. In addition, a Docker container is available to experiment without cloning and building from the source code.

```
$ docker pull stillwater/universal
$ docker run -it --rm stillwater/universal bash
```

Universal started in 2017 as a bit-level arithmetic reference implementation of the evolving unum Type III (posit and valid) standard. However, the demands for supporting various number systems, such as adaptive-precision integers to solve large factorials, adaptive-precision floats to act as Oracles, or comparing linear and tapered floats provided the opportunity to create a

complete platform for numerical analysis and computational mathematics. As a result, several projects have leveraged *Universal*, including Matrix Template Library (MTL4), Geometry + Simulation Modules (G+SMO), Bembel, a fast IGA BEM solver, and the Odeint ODE solver.

The default build configuration will produce the command line tools, a playground, and educational and application examples. It is also possible to construct the full regression suite across all the number systems. For instance, the shortened output for the commands `single` and `single 1.23456789` are below.

```
$ single
min exponent                                         -125
max exponent                                          128
radix                                                   2
radix digits                                           24
min                                           1.17549e-38
max                                           3.40282e+38
lowest                                       -3.40282e+38
epsilon (1+1ULP-1)                            1.19209e-07
round_error                                           0.5
denorm_min                                     1.4013e-45
infinity                                              inf
quiet_NAN                                             nan
signaling_NAN                                         nan
...


$ single 1.23456789
scientific    : 1.2345679
triple form   : (+,0,0b00111100000011001010010)
binary form   : 0b0.0111'1111.001'1110'0000'0110'0101'0010
color coded   : 0b0.0111'1111.001'1110'0000'0110'0101'0010
```

Statement of need

High-performance computing (HPC), machine learning, and deep learning tasks (e.g., Carmichael et al., 2019; Cococcioni et al., 2022; Desrentes et al., 2022) have increased environmental impacts and financial costs due to massive energy consumption (Haidar, Abdelfattah, et al., 2018). These both result from growth requirements in processing and storage. In addition to redesigning algorithms to minimize data movement and processing, modern systems increasingly support multi-precision arithmetic in hardware (Haidar, Tomov, et al., 2018). Recently, NVIDIA added support for low-precision formats to its top-level GPUs to perform tensor operations (Choquette et al., 2021), including a 19-bit format having an exponent of 8 bits and a mantissa of 10 bits (see also [Intel Corporation (2018); kharya:2020]. In addition, the “Brain Floating Point Format,” commonly referred to as “bfloat16,” is a format developed by Google that enables the training and operation of deep neural networks using specialized processors called Tensor Processing Units, or TPUs, at higher performance and cheaper cost (Wang & Kanwar, 2019). As a result, we see a trend to redesign many standard algorithms. In particular, designing fast and energy-efficient linear solvers is an active area of research where low-precision numerics plays a fundamental role (Carson & Higham, 2018; Haidar et al., 2017; Haidar, Tomov, et al., 2018; Haidar, Abdelfattah, et al., 2018; Higham et al., 2019).

While the primary motivation for low-precision arithmetic is its high performance and energy efficiency, mixed-precision algorithm designs aim to identify and exploit opportunities to right-scale the number of systems used for critical computational paths representing the execution bottleneck. Furthermore, when these algorithms are incorporated into embedded devices and

The Journal of Open Source Software

custom hardware engines, we approach optimal performance and power efficiency. Therefore, investigations into computational mathematics and measuring mixed-precision algorithms' accuracy, efficiency, robustness, and stability are needed.

Custom number systems that optimize the entire system's performance per watt (W) are crucial components with the rise of embedded devices demanding intelligent behavior. Likewise, energy efficiency is an essential differentiator for embedded intelligence applications. Using the distinct arithmetic requirements of the control and data flow can result in considerable performance and power efficiency gains when creating unique compute solutions. Even within the data flow, we observe many requirements for precision and the required dynamic range of the arithmetic operations.

Verification Suite

Each number system contained within *Universal* is supported by a comprehensive verification environment testing library class API consistency, logic and arithmetic operators, the standard math library, arithmetic exceptions, and language features such as compile-time constexpr. The verification suite is run as part of the `make test` command in the build directory.

Due to the size of the library, the build system for *Universal* allows for fine-grain control to subset the test environment for productive development and verification. There are twelve core build category flags defined:

- BUILD_APPLICATIONS
- BUILD_BENCHMARKS
- BUILD_CI
- BUILD_CMD_LINE_TOOLS
- BUILD_C_API
- BUILD_DEMONSTRATION
- BUILD_EDUCATION
- BUILD_LINEAR_ALGEBRA
- BUILD_MIXEDPRECISION_SDK
- BUILD_NUMBERS
- BUILD_NUMERICS
- BUILD_PLAYGROUND

The flags, when set during cmake configuration, i.e. `cmake -DBUILD_CI=ON ..`, enable build targets specialized to the category. For example, the `BUILD_CI` flag turns on the continuous integration regression test suites for all number systems, and the `BUILD_APPLICATIONS` flag will build all the example applications that provide demonstrations of mixed-precision, high-accuracy, reproducible and/or interval arithmetic.

Each build category contains individual targets that further refine the build targets. For example, `cmake -DBUILD_NUMBER_POSIT=ON -DBUILD_DEMONSTRATION=OFF ..` will build just the fixed-size, arbitrary configuration posit number system regression environment.

It is also possible to run specific test suite components, for example, to validate algorithmic changes to more complex arithmetic functions, such as square root, exponent, logarithm, and trigonometric functions. Here is an example, assuming that the logarithmic number system has been configured during the cmake build generation:

```
$ make lns_trigonometry
```

The Journal of Open Source Software

The repository's README file has all the details about the build and regression environment and how to streamline its operation.

Availability and Documentation

Universal Number Library is available under the MIT License. The package may be cloned or forked from the GitHub repository. Documentation is provided via `Docs`, including a tutorial introducing primary functionality and detailed reference and communication networks. The library employs extensive unit testing.

Acknowledgements

We want to acknowledge all code contributions, bug reports, and feedback from numerous other developers and users.

References

Carmichael, Z., Langroudi, H. F., Khazanov, C., Lillie, J., Gustafson, J. L., & Kudithipudi, D. (2019). Deep positron: A deep neural network using the posit number system. *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1421–1426.

Carson, E., & Higham, N. J. (2018). Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing*, *40*(2), A817–A847.

Choquette, J., Gandhi, W., Giroux, O., Stam, N., & Krashinsky, R. (2021). NVIDIA A100 tensor core GPU: Performance and innovation. *IEEE Micro*, *41*(2), 29–35.

Cococcioni, M., Rossi, F., Emanuele, R., & Saponara, S. (2022). Small reals representations for deep learning at the edge: A comparison. *Proc. Of the 2022 Conference on Next Generation Arithmetic (CoNGA'22)*.

Desrentes, O., Resmerita, D., & Dinechin, B. D. de. (2022). A Posit8 decompression operator for deep neural network inference. *Next Generation Arithmetic: Third International Conference, CoNGA 2022, Singapore, March 1–3, 2022, Revised Selected Papers*, *13253*, 14.

Haidar, A., Abdelfattah, A., Zounon, M., Wu, P., Pranesh, S., Tomov, S., & Dongarra, J. (2018). The design of fast and energy-efficient linear solvers: On the potential of half-precision arithmetic and iterative refinement techniques. *International Conference on Computational Science*, 586–600.

Haidar, A., Tomov, S., Dongarra, J., & Higham, N. J. (2018). Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 603–613.

Haidar, A., Wu, P., Tomov, S., & Dongarra, J. (2017). Investigating half precision arithmetic to accelerate dense linear system solvers. *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 1–8.

Higham, N. J., Pranesh, S., & Zounon, M. (2019). Squeezing a matrix into half precision, with an application to solving linear systems. *SIAM Journal on Scientific Computing*, *41*(4), A2536–A2551.

Intel Corporation. (2018). *BFLOAT16 - Hardware Numerics Definition*. https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-pap pdf

Omtzigt, E. T. L., Gottschling, P., Seligman, M., & Zorn, W. (2020). Universal Numbers Library: Design and implementation of a high-performance reproducible number systems library. *arXiv:2012.11011*.

Omtzigt, E. T. L., & Quinlan, J. (2022). Universal: Reliable, reproducible, and energy-efficient numerics. *Conference on Next Generation Arithmetic*, 100–116.

Wang, S., & Kanwar, P. (2019). BFloat16: The secret to high performance on cloud TPUs. *Google Cloud Blog*, *4*.