Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Towards Stochastic Rounding for Scientific Applications

Master Thesis

Thomas Creavin

August 16, 2025

Advisors: Prof. Dr. T. Hoefler, Dr. A. Calotoiu
Department of Computer Science, ETH Zürich

Acknowledgments

I would like to profoundly thank my supervisor, Lex. Thank you for providing such deeply pragmatic advice, for taking meetings at all hours, and offering suggestions that significantly improved the quality of my thesis.

I want to thank my professor, Torsten. Thank you for your insightful questions and reviews. It's very clear to me why you're the professor!

I'm grateful to the whole Scalable Parallel Computing Lab whose work I build upon. I'm most grateful to Yakup who kindly assisted me with my ICON experiments.

I would be remiss not to acknowledge the role Oisín played in bringing about this thesis. Oisín so thoughtfully took it upon himself to reach out to John, and John in turn got in touch with Peter, who ultimately and generously gave me the idea to explore this topic. Thank you Peter, thank you John, and thank you so, so much Oisín.

On a more personal note, throughout the thesis, I was sustained by the companionship of my wonderful friends. In particular, I truly cherish the copious – verging on exorbitant – number of hours spent frivolously chatting with Timur, Steffen, and Filip. Though in some ways, it's an integral part of the process! In addition, I'm fortunate to have the support of my caring housemates, Jürg and Bettina.

Last but not least, this whole journey is made easier by the backing of my loving family: my dad Michael, my mom Nellie, my two brothers Adam and Paul, et al.

Thank you all!

"Everything old is new again."

— Proverb

Abstract

Modern hardware's increasing reliance on lower-precision floating-point arithmetic challenges traditional double-precision climate simulations. To address this, we integrated Stochastic Rounding (SR) into the DaCe framework, analyzed the performance properties of random number generators (RNGs) to maximize performance without compromising on accuracy. Our implementation enables automated transformation of existing applications to use SR with minimal developer effort. We successfully replicated existing literature results and expanded our analysis to various kernels and the ICON weather and climate model. We demonstrated that SR provides substantial benefits, up to three orders of magnitude error reduction, for computations involving long accumulation chains and simulations prone to stagnation in fixed points. We found that it offered marginal benefits to ICON and other kernels. This highlights the need for further advancements to make SR consistently successful in real-world applications, an exploration made significantly easier by our developed framework.

Contents

Chapter 1

Introduction

The needs of AI applications are increasingly driving the development and production of new hardware [6]. For decades, scientific applications have dominated the high-performance computing space, with many requiring or relying on double-precision computations. Deep neural networks (DNNs) [14], in contrast, do not need such high-precision tools to achieve their goals [7]. In fact, there are many approaches that show AI models can be trained more efficiently and used more cheaply by reducing the precision of the floating-point operations used [11], with new, efficient floating-point number representations being developed specifically for this purpose such as BFloat16.

New accelerators, like NVIDIA's GB200 and GH200 superchips, focus on improving the throughput of lower precision floating-point computation. The GH200 offers twice as much float32 (FP32) computation and at least 7 times more float32 tensor core compute when compared to float64 (FP64). This shift towards lower-precision, high-throughput computation is mirrored in the architecture of modern supercomputers, where the proportion of GPU-based compute resources has grown significantly in recent years. Four of the top ten supercomputers use GPUs, with Alps (2024) and Eagle (2025) are boasting the new NVIDIA GH200 superchips.

While traditionally, scientific computing did rely on double-precision floating-point computations, there are great incentives to move to single-precision: half the memory requirements, double the throughput, and relief for memory-bandwidth-bound applications. These benefits increase further if applications are able to move to half-precision or lower. However, retaining the same accuracy and being able to provide the same quality of scientific results remains an unsolved challenge. There are many different techniques to boost the accuracy of lower-precision computations, from emulating higher-precision floating-point operations with lower-precision numbers or integers [22], to stochastic rounding [5], which uses the rounding process itself to try and

Specification	GB200	GH200
FP64	80 TFLOPS	34 TFLOPS
FP64 Tensor Core	80 TFLOPS	67 TFLOPS
FP32	160 TFLOPS	67 TFLOPS
TF32 Tensor Core	2.5 PFLOPS	494 TFLOPS
FP16/BF16 Tensor Core	5 PFLOPS	990 TFLOPS
FP8 Tensor Core	10 PFLOPS	990 TFLOPS
INT8 Tensor Core	10 POPS	1,979 TOPS
FP4 Tensor Core	20 PFLOPS	–

Table 1.1: NVIDIA GB200 and GH200 specifications showing the increased performance of low-precision floats and in particular that of the low precision tensor cores.

eliminate some of the numerical errors introduced.

Early exploration into stochastic rounding [7] [21] has shown it to be a promising avenue, with multiple companies even adding hardware support for it in specialized chips (e.g., Graphcore's IPU).

In this work, we propose to expand this exploration by considering both more and varied micro-benchmarks as well as testing its applicability on components from the ICOsahedral Non-hydrostatic weather and climate modeling framework (ICON) [10]. Towards this goal, we introduce an automated approach that allows entire benchmarks and applications to be transformed in order to support stochastic rounding rather than the more common round-to-nearest solution, as manual rewriting of real applications is prohibitively expensive, especially if the benefit is unproven. Furthermore, we extend the popular NumPy benchmarking suite NPBench [25] with the ability to compare the error of floating-point functions ran in different precisions.

Our experiments reveal both the promise and the cost of stochastic rounding (SR) on existing hardware. On micro-benchmarks, applying SR increases computation time by a factor of 8.1×, with roughly 50% of the overhead dominated by random number generation. Interestingly, SR performs nearly identically whether using low- or high-quality RNGs, suggesting that runtime can be reduced by choosing a lower-quality generator without sacrificing accuracy. In terms of error reduction, SR generally provides modest improvements for typical computations, but its effect becomes pronounced in workloads with long accumulation chains. We find that for large dot products, SR can reduce error by up to three orders of magnitude. The

accumulation error is highly data dependent: vectors or arrays whose values predominantly accumulate in one direction exhibit faster error growth, whereas symmetric or zero-centered distributions are less sensitive. These observations emphasize that SR is most effective when carefully applied to workloads prone to numerical error accumulation.

Our contributions can be summarized as follows:

- An in-depth evaluation of stochastic rounding on a wide range of kernels as well as on components of the ICON climate model.
- A method to change both the rounding scheme and the representation of floating-point data used by an application with minimal developer effort.
- An extension of the NPBench benchmark suite [25] with the ability to compare the errors of different floating-point computation solutions.

Chapter 2

Background

2.1 A Note on Floating-Point Formats

Before we proceed, we describe the floating-point number representation in a bit more detail, especially what is meant by *double* and *single* precision. All data is stored and processed in binary form at the hardware level. This can be applied to integers in a straightforward way: the representation contains one bit for each power of two, and the number's value is obtained by summing the products of each bit's value with the corresponding power of two:

$$13_{10} = 8 + 4 + 1 = 2^3 + 2^2 + 2^0 = 1101_2.$$

An additional bit can be used in some formats to represent whether a number is positive or negative. Representing decimal values is more involved. One scheme for example represents the integer part as the sum of positive powers of 2 and the fractional part as the sum of negative powers of 2:

$$3.125_{10} = 2 + 1 + \frac{1}{8} = 2^1 + 2^0 + 2^{-3} = 11.001_2.$$

Most scinetific computations are performed using a binary floating-point scheme, as it allows representation of a much wider range of values compared to fixed-point representations. In a floating-point scheme, values are represented using a significand scaled by a power of two:

$$3.125_{10} = 1.5625_{10} \times 2^1 = 1.1001_2 \times 2^1.$$

The IEEE 754 standard defines several possible floating-point number representations using different number of bits:

Precision	Format	Sign bit	Significand bits	Exponent bits
Double	Floating-point 64	1	52	11
Single	Floating-point 32	1	23	8
Half	Floating-point 16	1	10	5

Table 2.1: IEEE 754 Floating-Point Number Representations

Currently, many competing proposals exist for additional representations, some more compact, some using even fewer bits, and all tailored more closely to the needs of machine learning applications. Some popular alternatives include BFloat16 which offers a wider range than Floating-point 16 but with reduced precision; TensorFloat-32 which enables faster computations on NVIDIA hardware; and block floating-point [24], which conserves memory by grouping significands into blocks that share a common exponent. Still, most scientific applications use double-precision.

Precision	Format	Sign bit	Significand bits	Exponent bits
BFloat16	Brain Floating-point 16	1	7	8
TF32	TensorFloat-32	1	10	8
Mini Float	Mini Floating-point 8	1	4	3
BFP16	Block Floating Point 16	1	15	8 (shared)

Table 2.2: Alternative Floating-Point Number Representations

2.2 Rounding Modes and Round-off Error

Going from double-precision, which can represent ~ 16 decimal digits[1], to single-precision, which can represent ~ 7, it is evident we cannot represent the same range of real numbers. For long-running calculations, some values may need to be truncated or rounded so they can fit within the representable range of the floating-point numbers. This introduces round-off errors, and these errors can accumulate and have a significant impact on the results. In such cases, *how* we round or truncate values plays an important role in simulation accuracy.

We will first discuss the rounding methods offered by the *Standard for Floating-Point Arithmetic*(IEEE 754), by presenting all five rounding modes:

- **Round to nearest**
  - *Round to nearest, ties to even* – rounds to the nearest value; if the number falls midway, it is rounded to the nearest value with an

[1]Number of digits can be computed with $\log_{10}(2) \cdot (\text{significand length} + 1)$

even least significant digit. This is the default for binary floating-point and the recommended default for decimal.

- *Round to nearest, ties away from zero* – rounds to the nearest value; if the number falls midway, it is rounded to the nearest value above (for positive numbers) or below (for negative numbers).

- **Directed roundings**
  - *Round toward 0* – directed rounding towards zero (also known as truncation).
  - *Round toward* $+\infty$ – directed rounding towards positive infinity (also known as rounding up or ceiling).
  - *Round toward* $-\infty$ – directed rounding towards negative infinity (also known as rounding down or floor).

The main drawback of these modes is the introduction of a small bias. Even with round to nearest, if a value is incremented by less than half floating-point spacing, it will be rounded down. This means that a potentially important piece of information is lost.

2.3 Stochastic Rounding

To address this, SR was proposed as early as the 1950s by [2]. Unlike the previous modes, it behaves probabilistically:

$$\mathrm{SR}(x) = \begin{cases} \lceil x \rceil, & \text{with probability } p(x), \\ \lfloor x \rfloor, & \text{with probability } 1 - p(x). \end{cases} \tag{2.1}$$

Where $\lfloor x \rfloor$ represents the largest representable floating-point value number less than or equal to x; $\lceil x \rceil$ the smallest greater than or equal to x; and p(x) is given by,

$$p(x) = 1 - \frac{\lceil x \rceil - x}{\lceil x \rceil - \lfloor x \rfloor} \tag{2.2}$$

Or put simply, SR rounds a value to the next larger or smaller floating-point representation with probability 1 minus the relative distances to those representations.

Stochastic rounding offers notable advantages over round to nearest. [4] show that:

1. Rounding errors are mean-independent random variables with zero mean.

2. For the dot product, stochastic rounding yields the expected result and avoids stagnation. In particular, the dot product of two uniform $[0, 1]$ vectors of n elements stagnates under round-to-nearest rounding for $n \gtrsim 10^6$. While with stochastic rounding, the computation progresses with an error bounded by $\sqrt{n} \cdot u$, where u (unit roundoff) is the maximum relative error introduced by rounding[2]

However, to use the SR rounding mode, it must be emulated in software. Although major chipmakers like NVIDIA [1] and AMD [15] have patented SR technology, SR is not available in hardware except for a limited selection of specialized chips such as Intel Loihi, SpiNNaker2 [17], and GraphCore's IPU.

Despite the lack of hardware support, SR has recently garnered renewed interest, particularly in contexts where low-precision formats are already in use. This renewed attention was sparked by [7], which demonstrated that, for neural network training, a 16-bit fixed-point representation with stochastic rounding can be as effective as 32-bit floating-point numbers with round-to-nearest.

[2]Unit round-off is defined as $2^{-(p+1)}$ for a p-bit significand. For double, single, and half-precision, it corresponds to 1.1×10^{-16}, 6×10^{-8}, and 4.9×10^{-4}, respectively.

Chapter 3

State of the Art

3.1 Transitioning to Single-Precision

The European Centre for Medium-Range Weather Forecasts (ECMWF) offers a motivating example of successfully transitioning from high to lower precision while preserving the numerical accuracy of a real scientific application. Over four years, developers and scientists at ECMWF went from publishing an initial investigation into transitioning specific kernels to lower precision floating-point computation [23] to performing forecasts using the Integrated Forecasting System (IFS) in single-precision by default [9].

In the case of the IFS codebase, switching between single- and double-precision arithmetic (mostly) requires a simple update to the `FORTRAN KIND` parameter. This pattern can be found in the codebases of other weather and climate frameworks like ICON [10]. However, in these large codebases, changing the kind parameter is often only the first step of the process. Typically significantly more changes are required to move to lower precision successfully, summarized below in order of their complexity:

1. Some of these changes are simple, but tedious: they involve going through the code and ensuring that all data uses the kind parameter and there are no instances of variables with fixed precision, and replacing hard-coded thresholds and security constants with parameters defined using intrinsic functions of precision, e.g., replace code such as `x = 10.E+100` with the intrinsic FORTRAN function `x = huge(x)`.

2. Other changes require more extensive modifications, such as adapting the interfaces to linear algebra and Message Passing Interface (MPI) libraries, and ensuring the I/O routines still function correctly – input files storing data in binary form must be processed differently. Even some system functions can be dependent on the floating-point precision used.

3. Finally, the algorithms and model configuration need to be adapted to retain the accuracy needed for expressive weather forecasting: the time-stepping of the radiation scheme was adapted, Legendre transformations were left in double-precision because they are highly sensitive to round-off errors due to their recurrence relations and the vertical finite element scheme requires precomputing some integral operators in double-precision before finally truncating them to single-precision.

After all these challenges were overcome, the ECMWF team reduced the IFS model runtime by approximately 40% [9]. The example above demonstrates that while it is possible to transition even a full scientific application from double-precision to (mostly) single-precision computation, it is a very complex process requiring an in-depth understanding of both the software implementation and the physical processes being modeled. To reduce the effort required for transitioning to lower precision, various approaches aim to mitigate accuracy loss—ranging from modifying numerical rounding methods [21], to automatically reordering equations [20], to training neural networks for error correction [8].

3.2 Scientific Applications of Stochastic Rounding

Stochastic Rounding can be beneficial for training deep neural networks [7], but there is no guarantee it will generalize to scientific applications. Training machine learning applications is an inherently statistical process, and many architectures benefit from adding perturbations to their models [19].

In the following, we will introduce some existing applications of stochastic rounding for scientific problems found in literature, and reproduce their results using our approach. One such experiment from [17] demonstrates how applying SR can alleviate stagnation when computing the harmonic series ($\sum_{n=1}^{\infty} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots$) in low-precision. As shown in Table 3.1, low-precision fixed- and floating-point RTN formats stagnate early, with FP16 and BFloat16 incurring significant errors. Applying stochastic rounding to the 32-bit formats yields results comparable to double-precision while the 16-bit formats also see substantial improvement: BFloat16 approaches single-precision accuracy, and FP16, though less accurate, shows a marked improvement over its RTN counterpart. Notably, SR provides tighter bounds for floating-point than for fixed-point representations. These results quantitatively show the power of SR when applied to the right problems.

While the harmonic series is a highly idealized case, it serves to illustrate the core mechanism by which SR can reduce rounding error. To assess its impact in a more realistic setting, we next consider the work of [21], who investigated SR in the context of performing climate-related simulations at

Format	Sum at $i = 5 \times 10^6$	Error at $i = 5 \times 10^6$
FP64	16.002	0
FP32	15.404	0.598
FP16	7.086	8.916
BFloat16*	5.063	10.94
s16.15 RTN	11.938	4.064
s16.15 RD	10.553	5.449
s8.7 RTN	6.414	9.588
s8.7 RD	5.039	10.963
s16.15 SR	16.002 (s.d. 0.012)	-1.4×10^{-4}
FP32 SR*	16.002 (s.d. 8×10^{-4})	-3.5×10^{-5}
s8.7 SR	11.205 (s.d. 0.242)	4.797
FP16 SR*	11.638 (s.d. 0.012)	4.364
BFloat16 SR*	15.355 (s.d. 0.639)	0.647

Table 3.1: Summation of the harmonic series for different arithmetic formats from [17]. Our contributions are denoted by *. Sums and errors are reported relative to the double-precision result at the five-millionth iteration. The SR sums are obtained by running the experiment 50 times, each time using a different RNG seed. The formats sX.Y indicate a fixed precision type with a sign bit, X integer bits, and Y fractional bits. "RD" denotes the round-down mode and s.d. refers to the standard deviation.

single-precision. We attempt to both reproduce their results where the code is made available, and understand their generality.

The first experiment we consider is a simulation of a Lorenz system [16] across the main floating-point formats. This system exhibits features of nonlinear dynamics representative of the real atmosphere [18]. We present our recreation in Figure 3.1. When simulated correctly, the x and y coordinates should form a characteristic figure-eight orbit. As the system is highly chaotic, precise numerical reproduction of any specific orbit is not meaningful; following Paxton, we plot the orbits on a pixelated grid. From the figure, we observe that double and single-precision, in both rounding modes, produce visually similar distributions. In contrast, half-precision RTN suffers from the limited number of presentable values, forcing the simulation to become trapped in an early periodic orbit. Applying SR to half-precision introduces stochastic noise that prevents this stagnation, producing a trajectory that more closely resembles the higher-precision results.

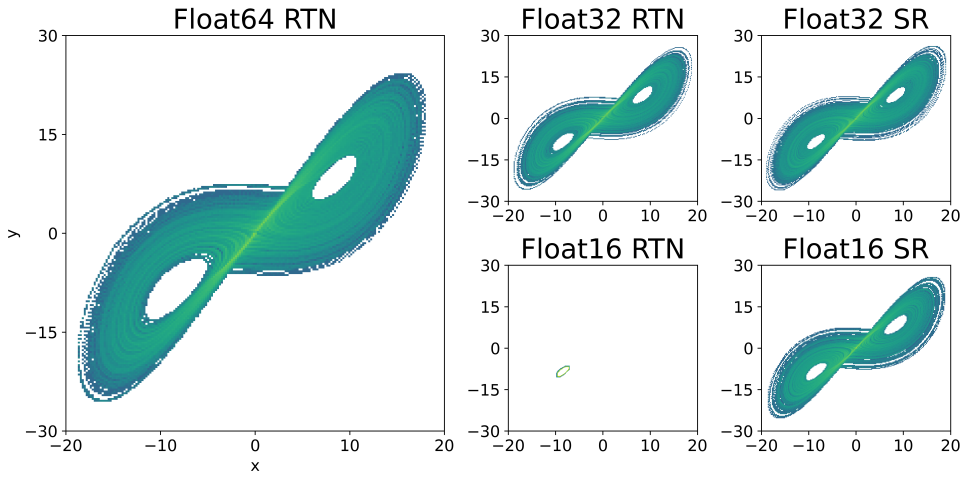


Figure 3.1: Our simulation of the Lorenz system at different floating-point precisions. Brighter colors indicate higher point density. Orbits are plotted on a 200×200 pixel grid, except for Float16 RTN, which is plotted on a 25×25 grid for legibility.

To quantify the differences between simulations, we compute the pixel-wise differences and plot the root mean squared error (RMSE) in Figure 3.2. As expected, the discrepancy for Float16 RTN is substantial. Interestingly, Float32 with stochastic rounding (SR) also exhibits a modest improvement over its round-to-nearest (RTN) counterpart, highlighting that SR can provide benefits even at higher precisions that don't suffer from stagnation.

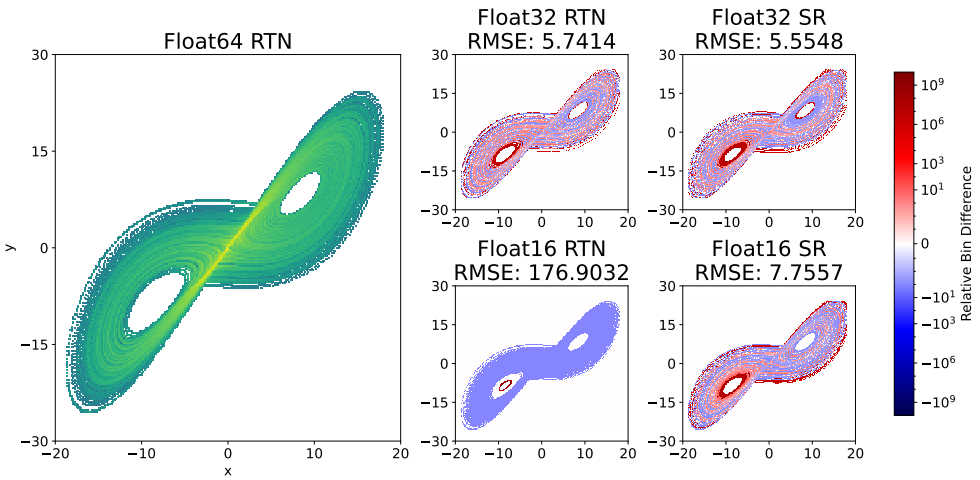


Figure 3.2: Root mean squared error (RMSE) of the pixel-wise differences between Lorenz system simulations across different floating-point formats.

The second experiment from Paxton investigates a nonlinear shallow-water model for turbulent flow over a ridge. They perform a twenty-year ensemble simulation of turbulent flow in a rectangular ocean basin at various precisions, computing an average snapshot for each. The difference between snapshots is quantified using the Wasserstein distance between their spatial distributions [12]. Results show that double- and single-precision RTN computations produce similar outcomes, whereas half-precision and BFloat16 perform notably worse. Applying stochastic rounding mitigates this degradation, reducing the errors of Float16 SR and BFloat16 SR to levels comparable with Float32 RTN.

This demonstrates that, even in a complex climate-modeling context, SR can effectively overcome the limitations of low-precision formats. We do not reproduce this experiment here, as it would require rewriting the Julia `ShallowWaters.jl`[1] package in Python, and instead re-use their original figure (Figure 3.3) to illustrate the visual difference between double- and half-precision after SR is applied.

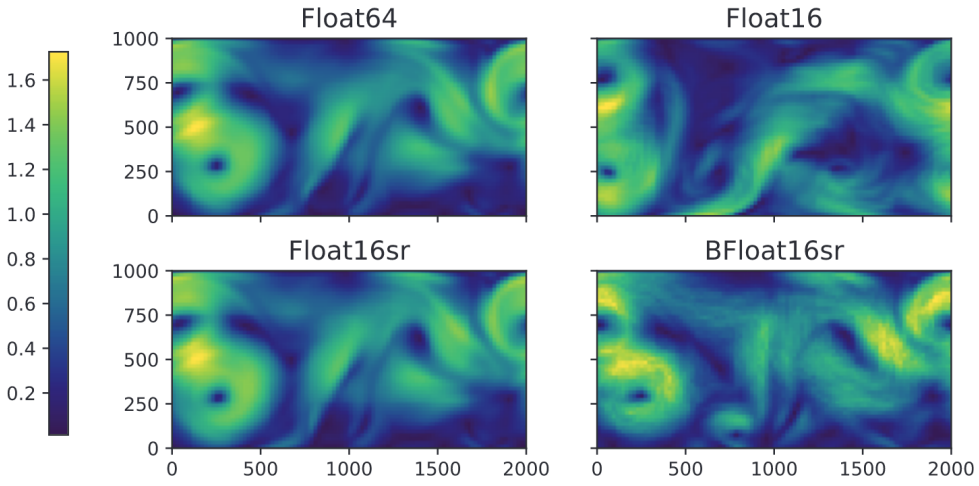


Figure 3.3: The shallow water model integrated at different precision levels by Paxton. A snapshot of the flow speed (m s^{-1}), initiated from the same initial condition, after 50 days.

Finally, Paxton simulate heat diffusion in a soil column warmed from the top and insulated at the bottom. The integration is carried out over 100 years, and since the source code was not available, we re-use their original figure (Figure 3.4). The results show that single- and half-precision RTN stagnate due to a small tendency term repeatedly rounding down to zero, preventing effective heat diffusion through the column. This issue can be mitigated by applying SR: Float32 SR produces results visually similar to the

[1] `https://github.com/milankl/ShallowWaters.jl`

double-precision simulation, while Float16 SR also improves over RTN, albeit with some visible artifacts.

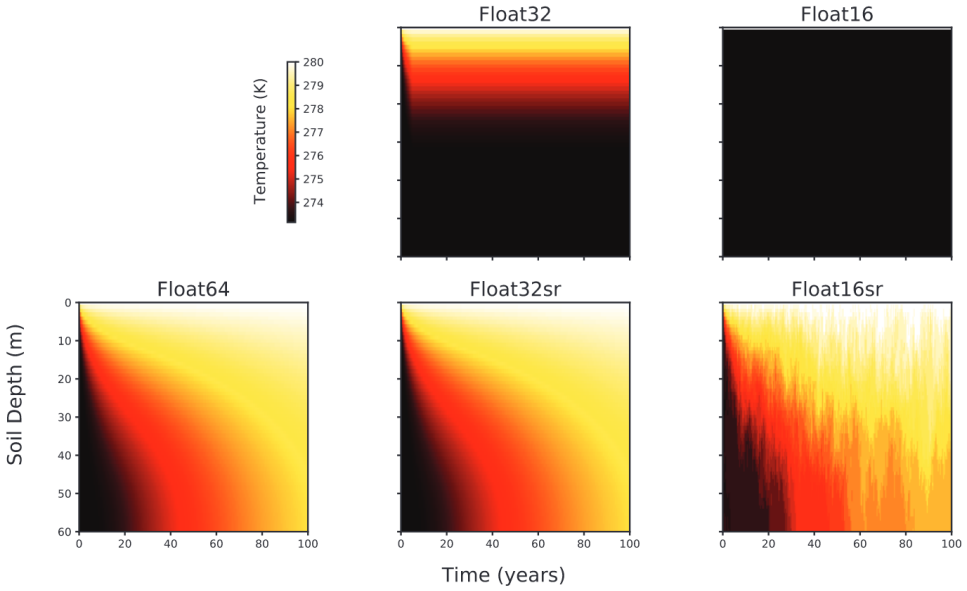


Figure 3.4: Heat diffusion in a soil column with different number formats and rounding modes from [21].

These findings are promising indicators that SR may enable production-level applications (e.g., ICON) to transition to lower-precision (e.g., single-precision) arithmetic with less additional work. However, providing a convenient way to explore SR while keeping the engineering effort bounded is an unsolved challenge.

Chapter 4

Testing Stochastic Rounding: Go Broad & Fail Fast

As seen from the example of IFS, moving to single-precision is no trivial feat. Some explorations, such as Paxton's work, suggest that it may be possible to achieve this move more easily with stochastic rounding. Given the crucial importance of climate modeling, we would like to explore how to add the option of using stochastic rounding to larger codes, such as components of ICON.

However, manually rewriting large scientific codes is a tremendous effort – as SR must currently be emulated – and whether SR will provide a benefit is not guaranteed.

To make it easier to quickly benchmark stochastic rounding and deploy it to larger applications, we use DaCe [3], a data-centric parallel programming framework. DaCe treats data movement as a first-class citizen in its intermediate representation, which leads to all data containers (arrays, individual variables) being easy to manipulate and change. This allows us to leverage the convenient overloading of operations for selected data-types and implement an emulation of floating-point computation using stochastic rounding with different precisions as a global program-level transformation.

Central to DaCe is its Stateful Dataflow multiGraphs (SDFG) data-centric intermediate representation: A transformable, interactive representation of code based on data movement. Since the input code and the SDFG are separate, it is possible to optimize a program without changing its source, so that it stays readable. This is of interest to us because it means we can leverage existing SDFGs of complex codes e.g., ICON to test SR on.

At the core of DaCe is its internal Stateful Dataflow multiGraph (SDFG) representation, a transformable, interactive model of code based on data movement. Because the SDFG is separate from the source code, programs

can be optimized without altering their original form. This separation is particularly useful for our purposes, as it enables us to apply SR to existing SDFGs from complex codes, like ICON, without almost any efforts.

Furthermore, we can use NPBench [25], a benchmark suite developed to compare the performance of different frameworks used to optimize NumPy-based scientific applications, as a starting point to create a framework to easily evaluate the performance and accuracy of stochastic rounding on a wide range of kernels representative of scientific computations.

In the following, we will discuss how we implemented stochastic rounding in DaCe, how we expanded NPBench, before discussing how we applied this method to a critical component of the dynamical core of ICON.

4.1 Stochastic Rounding in DaCe

DaCe has its own Python-defined type system that covers the full range of NumPy datatypes like bool, int, uint, float, complex, plus others like pointers.

Although DaCe is written in Python and accepts input programs written in NumPy, but also other languages such as Fortran or C. DaCe generates human-readable C++ code as output. Thus, during the compilation phase, DaCe maps custom high-level Python definitions to concrete C++ implementations. For instance, DaCe Float64 compiles down to C's double type; DaCe Float32 compiles to C's single-precision float type.

We implement stochastically rounded floats as distinct data types. We introduce two new types: DaCe `Float32sr` and DaCe `Float16sr` which can be used interchangeably with the existing round-to-nearest versions, requiring no additional implementation effort from developers and allow easy testing of their numerical impact. The implementation for `Float32sr` is given by Listing 4.1.

The addition of such new datatypes is quite lightweight, suggesting this is a viable path for further floating-point emulation schemes such as Ozaki [22]. Our new Python class definitions are concise – only 25 lines of code. The C++ definitions are more intricate, as the ultimate goal is to run real world applications on single-precision hardware accelerators like GPUs so we develop backend code for both CPU and GPU devices. We define and implement our own comparators, casting behavior, and our own arithmetic and I/O operations such as $\gg$. The `Float32sr` stochastic rounding function is given by Listing 4.2.

Rather than implement Equation 2.1 directly, we use efficient bit-wise operations to round with probabilities proportional to the nearest numerical representations. The pseudocode is provided in Algorithm 1. It follows

Listing 4.1 DaCe Type Definition

```
class Float32sr(typeclass):

def __init__(self):
    self.type = numpy.float32
    self.bytes = 4
    self.dtype = self
    self.typename = "float"
    self.stochastically_rounded = True

def to_json(self):
    return 'float32sr'

@staticmethod
def from_json(json_obj, context=None):
    return float32sr()

@property
def ctype(self):
    return "dace::float32sr"

@property
def ctype_unaligned(self):
    return self.ctype

def as_ctypes(self):
    return _FFI_CTYPES[self.type]

def as_numpy_dtype(self):
    return numpy.dtype(self.type)

@property
def base_type(self):
    return self
```

the SR implementation of the Julia StochasticRounding.jl[1] package closely [21], [13]. This method generates pseudo-random noise restricted to the significand bits that will be discarded during rounding. If the discarded bits high in value, and so close to being rounded up, then with a high probability the addition of random noise with cause the least significant kept bit to be incremented. If the discarded bits are low in value, then it is unlikely the random noise will cause the least significant kept bit to be incremented. In this way, the rounding occurs with an unbiased probability in expectation. The surplus bits are then truncated, and the result is cast to the target lower-

[1] https://github.com/milankl/StochasticRounding.jl

Listing 4.2 Single-precision Stochastic Rounding Function

```
DACE_HOST_DEVICE static float stochastic_round(double x) {
    uint64_t rbits = get_random_u64();

    if (std::isnan(x) || std::isinf(x)) {
    return static_cast<float>(x);
    }

    if (abs(x) > FLOATMAX_F32) {
        return std::signbit(x) ? -INFINITY : INFINITY;
    }

    // if x is subnormal, round randomly; N.B fast-math forces
    ↪  subnormal values to 0
    if (abs(x) < FLOATMIN_F32) {
        return static_cast<float>(x + rand_subnormal(rbits));
    }

    uint64_t bits = double_to_bits(x);
    const uint64_t mask = 0x000000001FFFFFFF;  // mask last 29
    ↪  surplus bits of the double prec mantissa

    bits += (rbits & mask);  // add stochastic perturbation
    bits &= ~mask;           // truncate lower bits
    double rounded = bits_to_double(bits);
    return static_cast<float>(rounded);
}
```

precision format. Further care is taken to correctly handle special cases, such as subnormal numbers, infinities, and NaNs.

Algorithm 1 Stochastic Rounding by Perturbation and Truncation

function StochasticRound(*double*)
    $rand \leftarrow \text{RNG32}()$
    $mask \leftarrow (1 \ll 29) - 1$ ▷ Mask bits lost to rounding
    $rand \leftarrow rand \;\&\; mask$
    $double \leftarrow double + rand$ ▷ Perturb bits lost to rounding
    $double \leftarrow double \;\&\; \sim mask$ ▷ Truncate surplus bits
    return FloatCast(*double*)
end function

In addition to ensuring correct SR implementation, performance is a key consideration. The Julia SR library reports that emulating SR can incur a

$5 - 10\times$ slowdown, with roughly half of the overhead coming from random number generation (RNG). Thus, the choice of generator is critical. With this in mind, we implement five variants to benchmark both runtime and error-reduction properties:

- *No RNG baseline* - uses a constant zero for the stochastic perturbation to measure SR overhead without RNG cost.
- *Default SR* - uses the Mersenne Twister, a high-quality RNG (Listing 4.3).
- *Low-quality RNGs* - two variants employing a Linear Congruential Generator (Listing **??**) and Xorshift (Listing 4.5).
- *Buffered SR* - uses a precomputed circular buffer of high-quality Mersenne Twister values to reduce RNG overhead (Listing 4.6).

Listing 4.3 Random 64-bit unsigned integer generator using the Mersenne Twister RNG

```
static uint64_t get_random_u64() {
    return get_rng()();
}

static std::mt19937_64& get_rng() {
    static std::mt19937_64 rng(std::random_device{}());
    return rng;
}
```

Listing 4.4 Linear Congruential Generator 64-bit

```
DACE_HOST_DEVICE static inline uint64_t lcg64() {
    rng_state_64 = rng_state_64 * 6364136223846793005ULL +
    ↪ 1442695040888963407ULL; // 64-bit LCG constants
    return rng_state_64;
}
```

In addition, we need a method to easily transform existing DaCe SDFGs to use stochastic rounding without major engineering effort. Thankfully, DaCe has the concept of a pass – a means of iterating over the components of an SDFG detecting clearly described patterns, and changing the program according to predefined rules, similar to AST transformations in classical compilers. We have added a new pass that changes variable types and can convert a simple data types to another. This pass allows us to swap from double-precision to single-precision with stochastic rounding. The implementation is given by Listing 4.7 and an example of it in operation is

Listing 4.5 Xorshift 64-bit RNG

```
DACE_HOST_DEVICE static inline uint64_t xorshift64() {
    rng_state_64 ^= rng_state_64 << 13;
    rng_state_64 ^= rng_state_64 >> 7;
    rng_state_64 ^= rng_state_64 << 17;
    return rng_state_64;
}
```

Listing 4.6 Preloaded random number generator using a shared queue

```
DACE_HOST_DEVICE static uint64_t get_preloaded_random_u64() {
  struct SharedRandomQueue {
      std::array<uint64_t, 10000> data;
      std::atomic<bool> initialized{false};

      SharedRandomQueue() { initialize(); }

      void initialize() {
          for (auto& x : data) { x = float32sr::get_random_u64(); }
          initialized.store(true, std::memory_order_release);
      }

      uint64_t get(size_t idx) const {
          return data[idx % data.size()];
      }
  };

  static SharedRandomQueue shared_queue;
  thread_local static size_t thread_index = 0;

  size_t current_index = thread_index;
  thread_index = (thread_index + 1) % shared_queue.data.size();

  return shared_queue.get(current_index);
}
```

given by Listing 4.8.

4.1.1 Implementation Validation

To verify the correctness of our stochastic rounding (SR) implementations, we developed a comprehensive PyTest suite. We test to ensure that SR initializations are constant for assignments of the same type; test that high-precision inputs are stochastically rounded according to the expected probabilities;

Listing 4.7 DaCe Pass for changing data types within an SDFG

```
class TypeChange(ppl.Pass):
    CATEGORY: str = 'Simplification'

    def __init__(self, from_type: dace.typeclass = None, to_type:
    ↪ dace.typeclass = None):
        self._from_type = from_type
        self._to_type = to_type
        self._swaps_count = 0

    def modifies(self) -> ppl.Modifies:
        return ppl.Modifies.Nothing

    def should_reapply(self, modified: ppl.Modifies) -> bool:
        return False

    def apply_pass(self, sdfg: SDFG, _) -> Optional[int]:
        if hasattr(sdfg, "orig_sdfg") and sdfg.orig_sdfg: # apply the
        ↪ pass to orig cpu sdfg
            if hasattr(sdfg.orig_sdfg, "all_sdfgs_recursive"):
                for nested_sdfg in
                ↪ sdfg.orig_sdfg.all_sdfgs_recursive():
                    self._change_sdfg_type(nested_sdfg)

        for nested_sdfg in sdfg.all_sdfgs_recursive():
            self._change_sdfg_type(nested_sdfg)

        return self._swaps_count


    def report(self, pass_retval: int) -> str:
        if pass_retval is None:
            return "No arrays found to analyze."
        return
        ↪ f"Analyzed {pass_retval} arrays and printed their types."

    # Private methods that iterate over the SDFG omitted for brevity
```

that both the CPU and GPU implementations exhibit consistent behavior across; and all supported arithmetic and comparison operators are verified to confirm that SR is correctly applied throughout.

We also evaluate mixed rounding modes to ensure that stochastic rounding can coexist with standard rounding methods without introducing errors. Special attention is given to special values like subnormal numbers, which constitute a range of very small floating-point values requiring distinct handling. In

Listing 4.8 Applying the `TypeChange` pass through a DaCe pipeline

```
from dace.transformation import pass_pipeline as ppl
from dace.transformation.passes.type_change import TypeChange

tc = TypeChange(dace.float64, dace.float32sr)
type_change_pipeline = ppl.Pipeline([tc])
results_of_pass = type_change_pipeline.apply_pass(sdfg, {})
output_of_SDFG_call = sdfg(single_precision_input)
```

single-precision, this range spans approximately $[1.45 \times 10^{-45}, 1.18 \times 10^{-38}]$ values outside this interval are rounded to zero. Interestingly, when the fast-math compiler optimization is enabled, as it the default for DaCe, it automatically forces subnormal values to zeros which required extra handling to test these values.

4.2 Easier Evaluation of Schemes: SR in NPBench

With validation complete, our next goal is to quantify the performance impact and accuracy trade-offs of stochastic rounding in a broader set of workloads.

NPBench offers a diverse collection of kernels, spanning matrix operations, stencils, statistical functions, and physical simulations. It also provides infrastructure to generate varied input sizes and perform automated benchmarking. Crucially, our DaCe type-change pass allows us to apply SR to all existing DaCe kernels without modification, enabling consistent and large-scale evaluation.

However, NPBench is designed primarily to measure run-time performance. To evaluate rounding errors, we extend the framework with a dedicated error measurement suite. In addition, we modify the input generation procedure so that each run produces randomized data while using fixed seeds, ensuring that inputs are identical across the different implementations being compared.

4.2.1 Performance

We first measure performance on medium-sized NPBench inputs. This size regime is chosen to minimise variability between runs while still allowing many iterations across multiple kernels, including repeated sub-runs for SR variants.

Our initial comparison examines the default SR implementation that uses the high-quality Mersenne Twister RNG and an SR variant without RNG, against a DaCe Float32 RTN baseline (Figure 4.1). The results show our unoptimized implementation introduces a 54× slowdown, far exceeding the anticipated

5–10× emulation cost. Even without RNG, the SR infrastructure alone incurs an unavoidable 4× slowdown per operation.

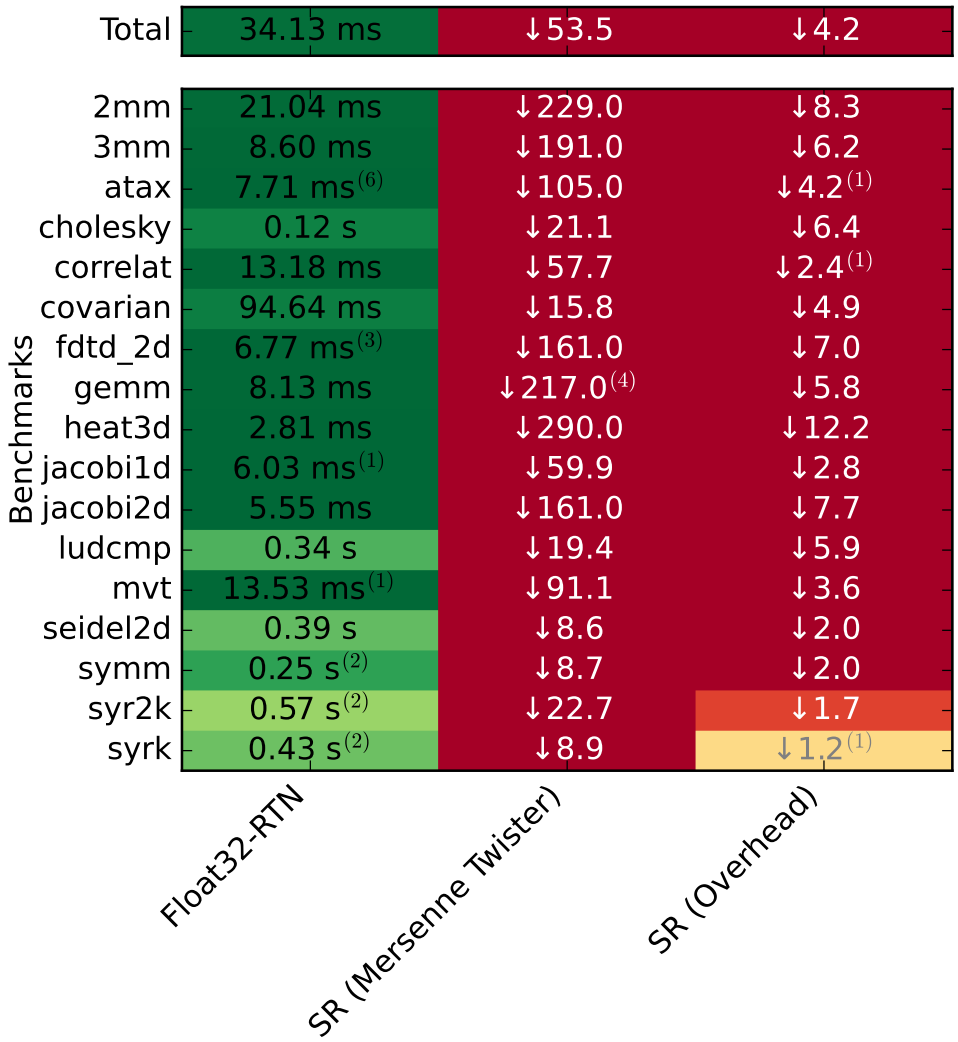


Figure 4.1: The top panel reports the average slowdown (relative to DaCe Float32 RTN) for our unoptimized implementation, and the average slowdown introduced by the SR infrastructure without RNG. Averages are computed as geometric means across all benchmarks. The lower panel presents per-benchmark slowdowns.

We then evaluate alternative RNG schemes to identify a more practical choice (Figure 4.2). All alternatives significantly reduce the overhead. The lightweight Linear Congruential Generator (LCG) is particularly notable, offering an order-of-magnitude speed-up over the Mersenne Twister and bringing the total slowdown to just 8× relative to Float32 RTN. This level of

overhead is acceptable if the RNG maintains strong error-reduction properties.

It is important to emphasize the broader performance context: if SR enables a double-precision application to run stably in single precision, then the move to modern GPUs or other accelerators can deliver large net speed-ups despite the emulation cost.

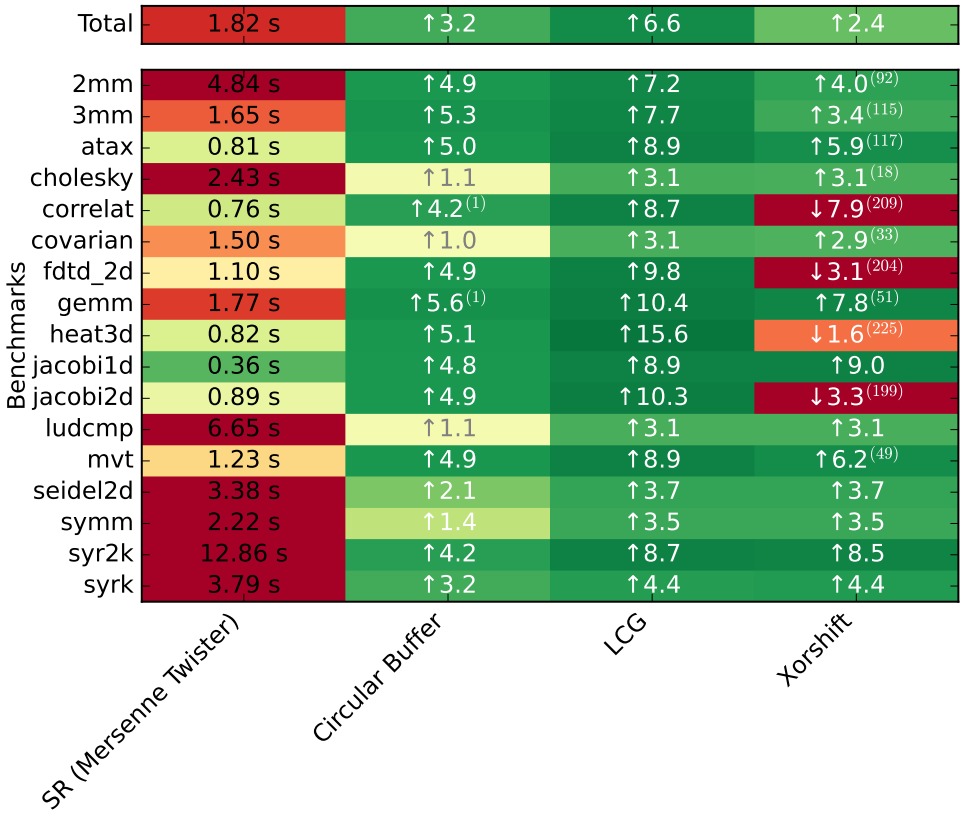


Figure 4.2: Runtime comparison of alternative RNG schemes across medium NPBench benchmarks. Mersenne Twister is used as the baseline.

4.2.2 Rounding Error

Given the substantial performance overhead of SR, it must deliver a proportional reduction in numerical error. Using our extended NPBench error-measurement suite, we evaluated four RNG variants (Figure 4.3). Across benchmarks, high- and low-quality RNGs produce nearly identical accuracy gains. Although average relative errors are already low in the Float32 RTN baseline, SR notably reduces the maximum observed error—particularly for the `atax`, `2mm`, and `3mm` kernels.

An unexpected result is that the circular-buffer approach underperforms relative to all other RNG implementations, despite using a large precomputed buffer of 10,000 high-quality Mersenne Twister values. The LCG emerges as the best overall choice, offering both strong accuracy and minimal runtime overhead. Future exploration could include related generators such as the Permuted Congruential Generator[2], which augments an LCG with a permutation function to improve statistical properties.

We extended the analysis by evaluating NPBench kernels under different data initialization schemes to assess error improvement across various distributions (Figure 4.4, Figure 4.5). While mean relative errors show good improvements, they remain close to the theoretical truncation cost when converting double-precision values to single-precisions and so is not so valuable.

The reduction in maximum errors is particularly noteworthy. Even with relatively small input sizes, many benchmarks exhibit instances of extremely large errors that SR effectively mitigates. For example, under the Uniform[0,1000] distribution, the average maximum relative error of 11.2$\times$ is reduced by approximately half to 5.0$\times$ when using SR. This substantial reduction in error outliers suggests that SR's primary benefit lies in preventing catastrophic accumulation errors rather than improving typical-case precision. We anticipate that larger input sizes would amplify these benefits, with SR providing more pronounced reductions in both mean and maximum relative errors as accumulation chains lengthen.

[2] `https://www.pcg-random.org/`

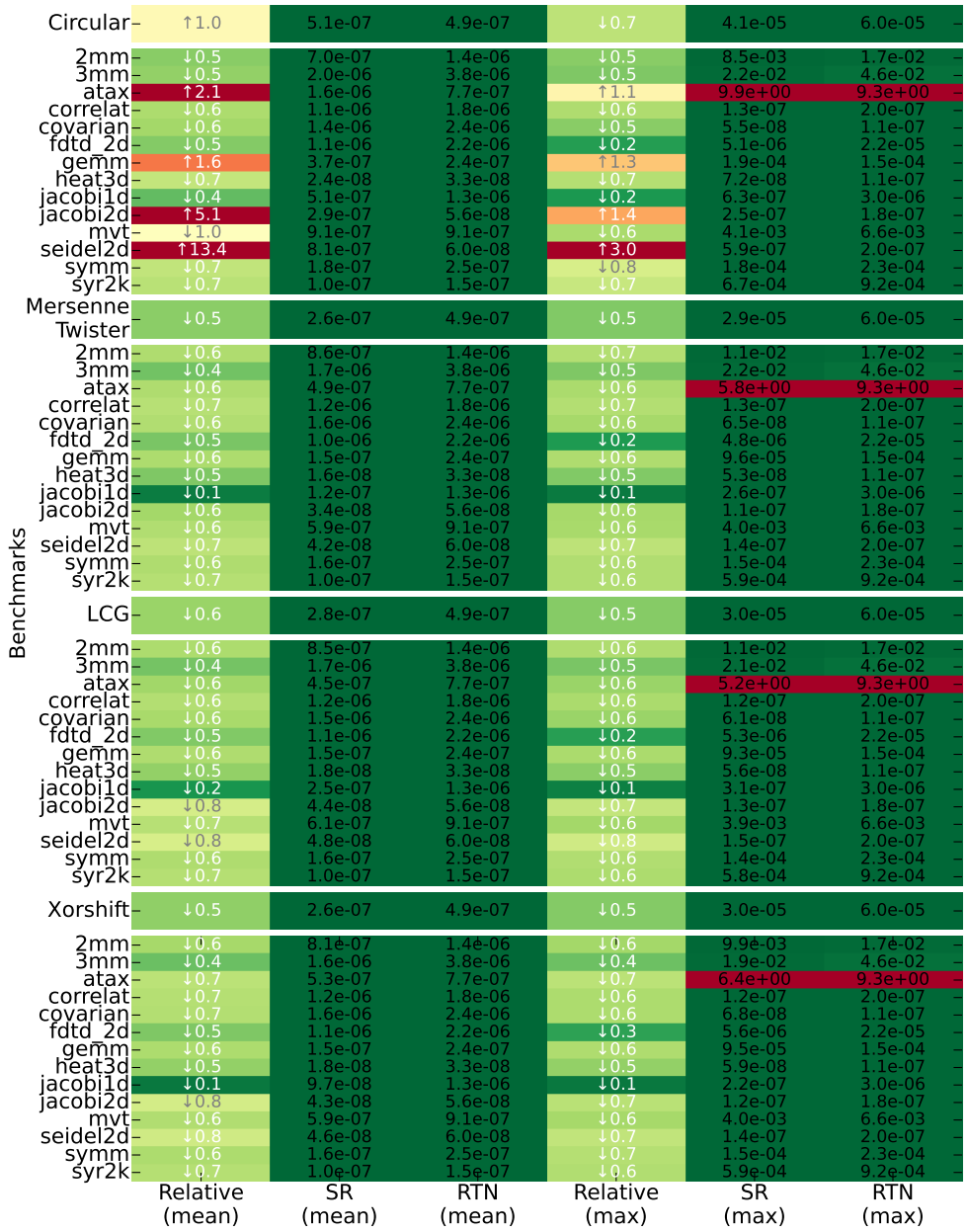


Figure 4.3: A comparison of RNGs and their effect of SR quality. The two "Relative" columns show the change in mean error and max error when going from RTN to SR. Performed on subset of medium NPBench benchmarks over 10 runs with each SR sum instance averaged over 5 samples. The input data is generated from a uniform random distribution with a minimum of zero and a maximum of one.

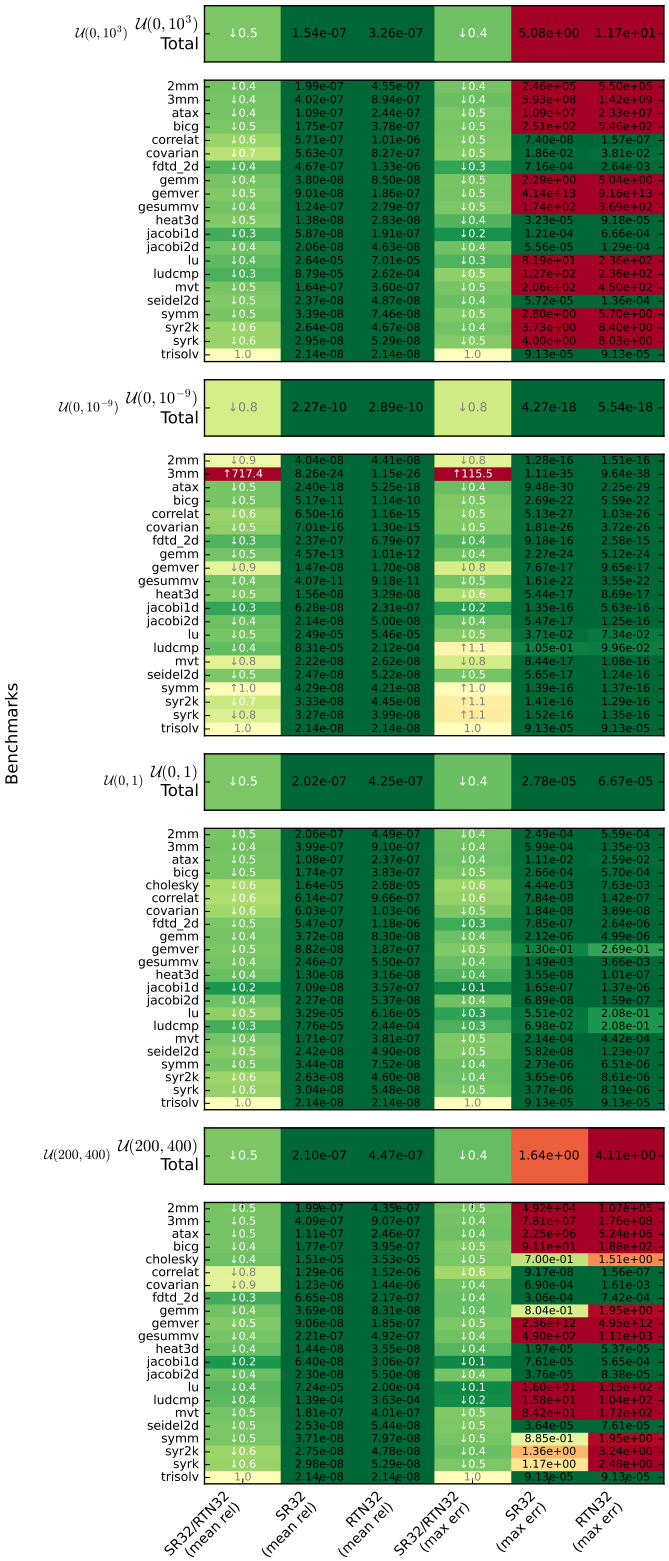


Figure 4.4: Error comparison across data initialization types on small NPBench kernels using the Mersenne Twister. Each result is averaged over 10 runs with each SR sum instance averaged over 5 samples.

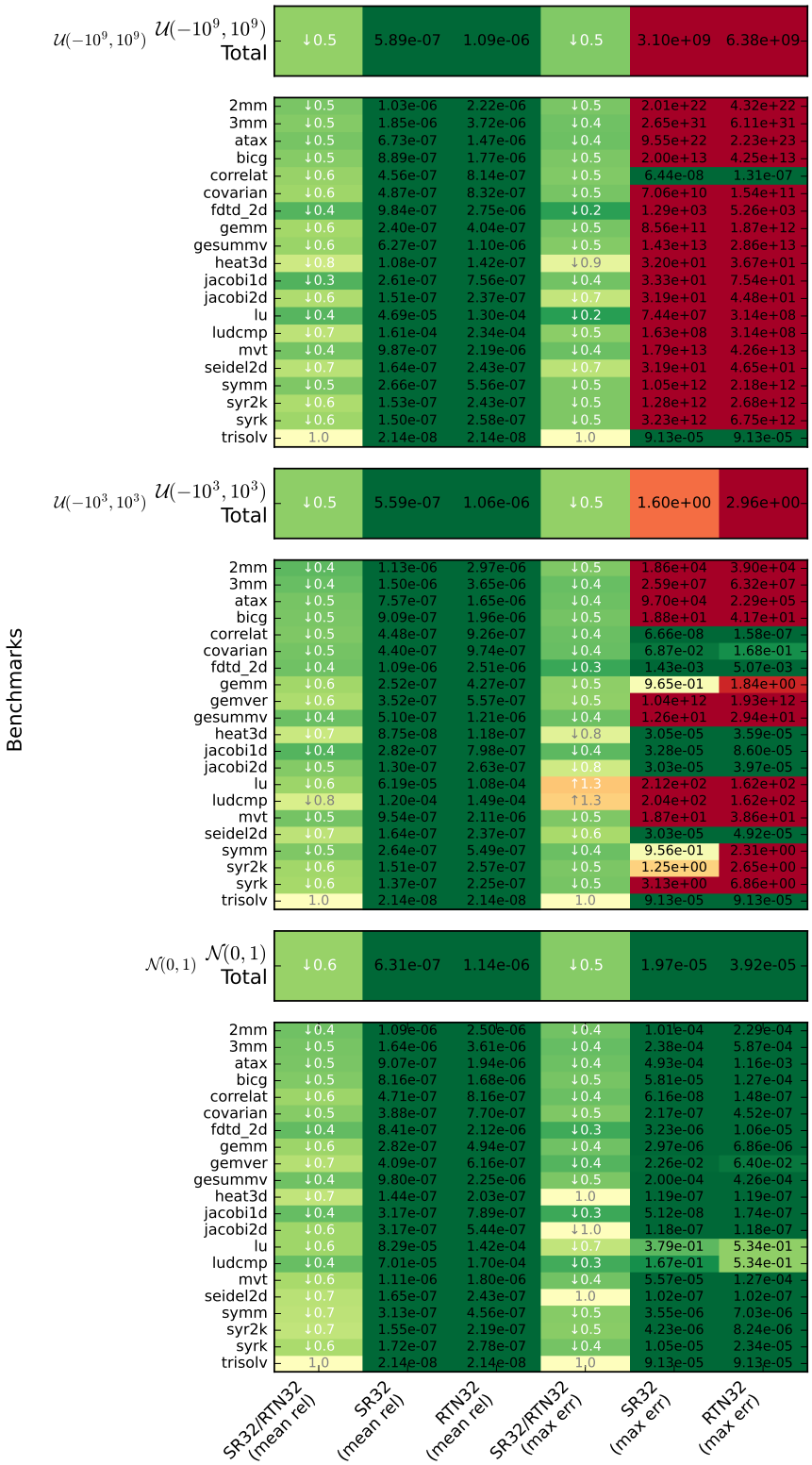


Figure 4.5: (Continued) Error comparison across data initialization types on small NPBench kernels using the Mersenne Twister. Each result is averaged over 10 runs with each SR scheme instance averaged over 5 samples.

Chapter 5

Results

With our stochastic rounding implementations validated and benchmarked, we now turn to reproducing established experiments from the literature. Previously, we extended the analysis of [17] on the convergence of the harmonic series (Table 3.1) and replicated the Lorenz system simulation of [21] (Figure 3.1). We now focus on two canonical numerical kernels examined by [4]: the dot product and the general matrix multiply (GEMM). Both are dominated by repeated accumulation operations, making them ideal for isolating and quantitatively assessing the influence of stochastic rounding on error growth. Their algorithmic simplicity minimizes confounding factors, ensuring that any differences arise primarily from rounding effects, while their ubiquity guarantees that the insights gained are directly applicable to a wide range of scientific computing workloads.

We evaluate both kernels under a variety of data initializations: purely random noise (masked to avoid extreme outliers); values drawn from a standard Gaussian distribution; strictly positive uniformly distributed values spanning small, medium, and large ranges; and medium-range uniformly distributed values centered at zero. These configurations are chosen to emulate the diversity of value distributions encountered in scientific computing. For example, temperature data in Kelvin typically lies within a narrow range of roughly 200–400.

5.1 Dot Product

We perform two sets of experiments: one at single-precision (Figure 5.2) and the other at half-precision (Figure **??**). Single-precision floating-point can represent values approximately in the range $\pm 10^{-38}$ to $\pm 10^{38}$, while half-precision is constrained to roughly $\pm 6 \times 10^{-8}$ to $\pm 65,504$. This significant difference in representable ranges necessitates using smaller data ranges for the half-precision experiments to avoid overflow conditions.

In both experimental settings, the choice of data initialization scheme profoundly influences the relative improvement of stochastic rounding over round-to-nearest. For distributions symmetric about zero, SR yields only modest error reductions, typically within the same order of magnitude as RTN. In contrast, for distributions where partial sums grow predominantly in a single direction, such as uniformly distributed positive values, the effect becomes far more pronounced, demonstrating SR's particular strength in mitigating systematic accumulation bias.

For single-precision experiments, once vector lengths exceed approximately one million elements, RTN errors increase rapidly while SR errors remain tightly bounded, achieving up to three orders of magnitude lower error than RTN. This behavior aligns closely with the theoretical predictions of [4]. The half-precision results show a similar pattern with RTN errors beginning to outpace SR around the two-thousand element threshold, reflecting the reduced precision's greater susceptibility to accumulation errors.

The experiments also illuminate the fundamental trade-off between Float16 and BFloat16 formats. While BFloat16 consistently exhibits higher absolute errors due to its reduced precision (7-bit vs. 10-bit significand), its wider exponent range (8 bits vs. 5 bits) enables completion of all experimental configurations. Float16 accumulations that exceed 65,536 overflow to infinity, forcing us to discard those runs. This limitation is particularly restrictive for distributions like Uniform[0,20], where Float16 could only reliably handle vectors of 512 elements before encountering overflow, whereas BFloat16 successfully processed the full range of vector sizes.

When accumulation is performed using higher-precision intermediate storage, the differences between RTN and SR become negligible. This occurs because the expanded significand eliminates the accumulation error that stochastic rounding is specifically designed to mitigate, confirming that SR's benefits are most pronounced when computational precision matches the precision limitations of the target format.

It is worth noting that we include the single-element vector case in our plots to establish the baseline truncation error when converting from higher to lower precision—this represents the unavoidable precision loss inherent to the format conversion itself. This baseline error corresponds to the unit roundoff of the target format. For single-precision floating-point, the unit roundoff is $\sim 6.0 \times 10^{-8}$; for Float16, it is $\sim 4.9 \times 10^{-4}$; and for BFloat16, $\sim 3.9 \times 10^{-3}$. These theoretical values are clearly reflected as the starting points in our error plots, providing a reference against which the accumulation-dependent error growth can be measured.

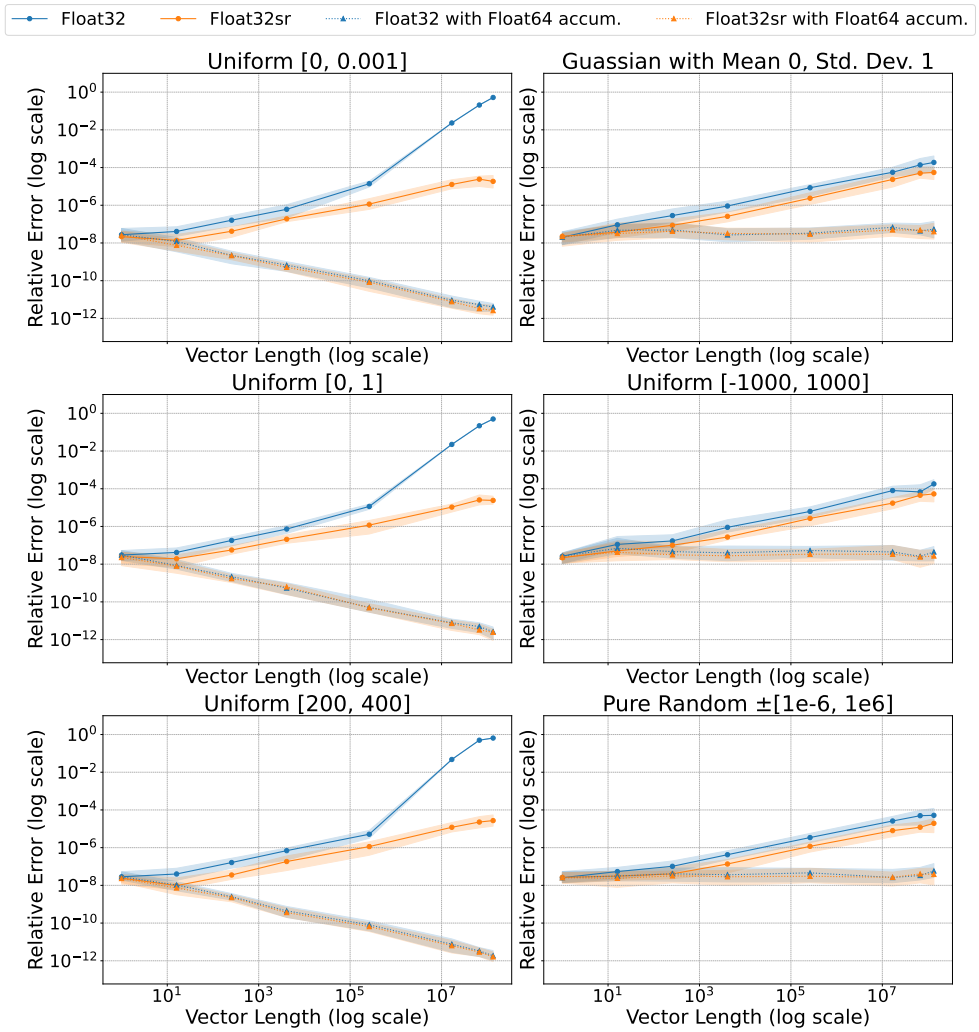


Figure 5.1: A comparison of relative errors when reducing precision from double to single precision using either RTN or SR for the dot product across different data distributions. We also consider a mixed-precision case in which the intermediate sum is stored in double precision. Each data point represents the median over 50 runs, with shaded bands indicating the inter-quartile range. For SR, each point is further averaged over 20 stochastic sub-runs.

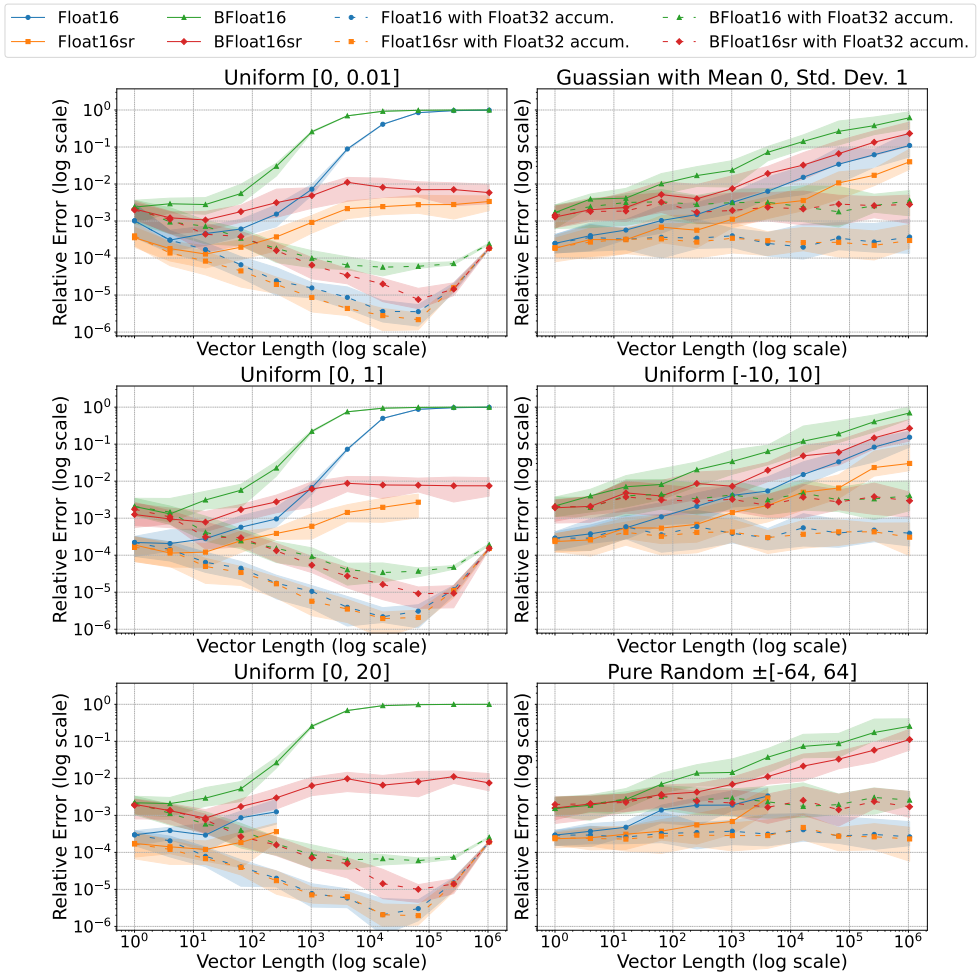


Figure 5.2: A comparison of relative errors when reducing precision from double to half-precision using either RTN or SR for the dot product across different data distributions. We also consider a mixed-precision case in which the intermediate sum is stored in single-precision. Each data point represents the median over 50 runs, with shaded bands indicating the inter-quartile range. For SR, each point is further averaged over 20 stochastic sub-runs.

General Matrix Multiply (GEMM) The choice of data initialization similarly influences GEMM results (Figure 5.3, Figure 5.4). Unlike the dot product, however, we do not observe a dramatic growth of error. This is largely due to the relatively modest size of the constituent dot products. For instance, the largest matrix considered is 2048×2048, resulting in dot products of length 2048, which, as we saw previously, do not produce significant error accumulation. Likewise, when using higher-precision accumulators, the differences between RTN and SR are negligible, with both rounding modes producing nearly identical results. This finding suggests that stochastic rounding's advantages are most evident in computations with exceptionally

long accumulation sequences, rather than in the moderately-sized operations typical of standard matrix computations.

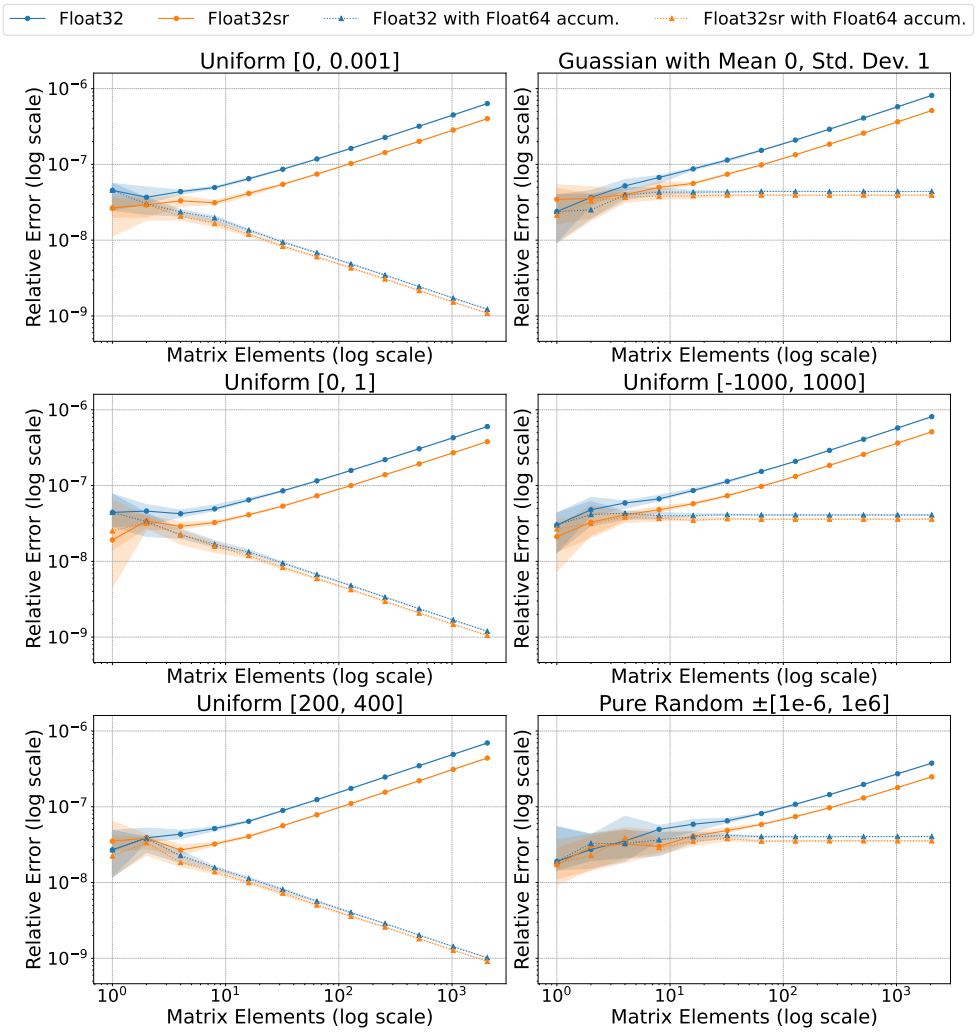


Figure 5.3: A comparison of relative errors when reducing precision from double- to single-precision using either RTN or SR for general matrix multiplication across different data distributions. We also consider a mixed-precision case in which intermediate sums are stored in double-precision. Each data point represents the median over twenty runs, with shaded bands indicating the inter-quartile range. For SR, each point is further averaged over five stochastic sub-runs.

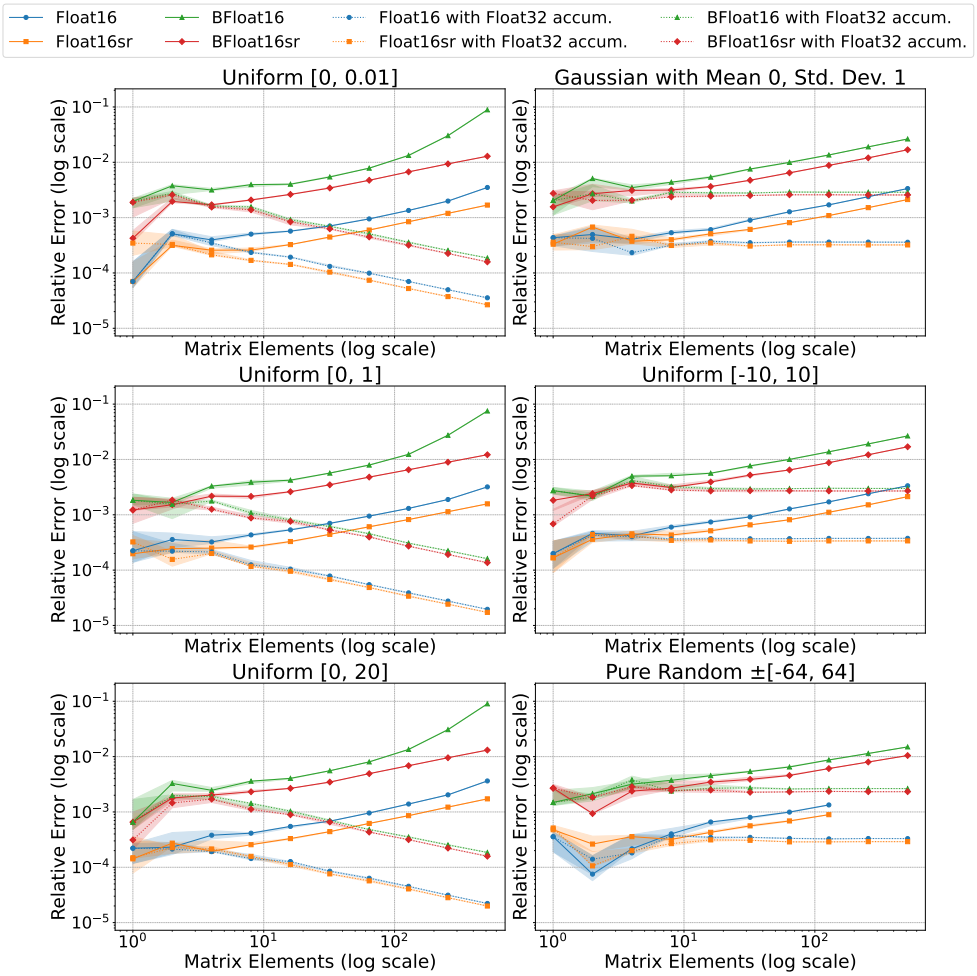


Figure 5.4: A comparison of relative errors when reducing precision from double- to half-precision using either RTN or SR for general matrix multiplication across different data distributions. We also consider a mixed-precision case in which intermediate sums are stored in single-precision. Each data point represents the median over 20 runs, with shaded bands indicating the inter-quartile range. For SR, each point is further averaged over 5 stochastic sub-runs.

5.2 Case Study: Stochastic Rounding in ICON

ICON is a next-generation modeling framework jointly developed by the Max Planck Institute for Meteorology (MPI-M) and the German Weather Service (DWD). Designed for both weather prediction and climate research, ICON employs a non-hydrostatic dynamical core on a quasi-uniform icosahedral grid, which avoids polar singularities and provides high scalability across modern high-performance computing systems. Its Earth System Model (ICON-ESM) couples atmospheric, oceanic, land-surface, and biogeochemical components to participate in CMIP6 experiments, offering robust climate

simulations with performance and bias metrics comparable to established models.

As a more challenging application, we applied stochastic rounding to ICON's velocity tendencies module which the SPCL team has carefully converted into SDFG form and validated numerically against the original FORTRAN implementation. This module is a particularly relevant test case because it sits at the heart of ICON's dynamical core, determining how momentum evolves on the icosahedral grid. It combines a mixture of advection, pressure-gradient, and corrective terms, each with distinct numerical sensitivities. From a rounding perspective, this mixture is instructive: while advection terms involve short-to-moderate length accumulations where SR might provide some stability benefits, corrective terms and coefficient-based routines can instead amplify noise if not carefully managed. As such, velocity tendencies has the potential to highlight both the promise and the limitations of SR. It would demonstrate that SR's effectiveness depends not only on the presence of long accumulation chains but also on the stability of the surrounding numerical formulations.

Our results, summarized in Table 5.1, show that despite the inherent error introduced when truncating double-precision values to single-precision, overall errors remain low across most variables, with the `z_kin_hor_e_1` array being the main outlier.

Further analysis suggests that SR's lack of improvement here stems from the characteristics of this component: it contains few long accumulation chains (where SR typically excels) and includes potentially unstable routines, such as the calculation of radial basis function coefficients. Similar to ECMWF's Legendre transformations, such numerically sensitive operations may be better preserved in double-precision.

While SR did not outperform RTN in this case, its seamless integration into a large, production-grade scientific application with minimal engineering effort demonstrates the flexibility of DaCe and the robustness of our SR implementation. Equally important, these results help refine our understanding of where SR offers the most benefit—pointing future efforts toward identifying and selectively targeting those high-impact computational patterns.

Array	Method	Mean Abs Error	Median Abs Diff	Q75	Max Abs Error	Mean Rel Error
ddt_vn_apc_pc	SR	1.70×10^{-10}	1.16×10^{-10}	2.33×10^{-10}	2.79×10^{-9}	1.533×10^{-2}
	RTN	1.45×10^{-10}	8.73×10^{-11}	2.33×10^{-10}	2.10×10^{-9}	5.815×10^{-3}
vn_ie	SR	7.71×10^{-7}	4.77×10^{-7}	9.54×10^{-7}	7.63×10^{-6}	8.475×10^{-8}
	RTN	7.59×10^{-7}	4.77×10^{-7}	9.54×10^{-7}	3.81×10^{-6}	8.384×10^{-8}
vt	SR	8.08×10^{-7}	4.77×10^{-7}	9.54×10^{-7}	7.63×10^{-6}	2.038×10^{-2}
	RTN	7.56×10^{-7}	4.77×10^{-7}	9.54×10^{-7}	3.81×10^{-6}	2.368×10^{-2}
w_concorr_c	SR	2.73×10^{-15}	4.44×10^{-16}	1.78×10^{-15}	2.27×10^{-13}	2.572×10^{-3}
	RTN	2.39×10^{-15}	4.44×10^{-16}	1.78×10^{-15}	1.14×10^{-13}	1.952×10^{-3}
z_kin_hor_e_1	SR	1.81×10^{-5}	7.63×10^{-6}	3.05×10^{-5}	3.05×10^{-4}	1.056×10^{-7}
	RTN	1.65×10^{-5}	7.63×10^{-6}	3.05×10^{-5}	2.44×10^{-4}	9.640×10^{-8}
z_w_concorr_me_1	SR	3.86×10^{-15}	4.44×10^{-16}	3.55×10^{-15}	6.82×10^{-13}	6.396×10^{-3}
	RTN	3.51×10^{-15}	4.44×10^{-16}	1.78×10^{-15}	4.55×10^{-13}	1.458×10^{-4}

Table 5.1: Error statistic when running the ICON Velocity Tendencies module in single-precision RTN and SR. The statistics are calculated by comparing single-precision output to the expected double-precision output.

Chapter 6

Conclusion

As the push for low-precision GPU parallelism accelerates, it is crucial to continue exploring ways to safely lower the precision of scientific applications. Stochastic rounding is not a silver bullet but it is a valuable tool for improving numerical stability in targeted situations.

This thesis also highlights the power of the DaCe and NPBench frameworks. DaCe allowed us to apply SR to a large, complex Fortran codebase with minimal engineering effort, while NPBench provided a versatile platform for systematically evaluating both error and performance across a diverse set of kernels. Together, they form a strong foundation for exploring and validating future numerical techniques aimed at making low-precision computation both practical and reliable.

Summary of Contributions. This thesis makes several key technical and methodological contributions to evaluating and deploying stochastic rounding in scientific applications. We implement two new stochastic rounding single- and half-precision data types within the DaCe framework, that support both CPU and GPU execution. We add a new DaCe pass that enable the easy changing of SDFG types, unlocking the means for abstract tests involving arbitrary types.

Through systematic analysis of random number generators, we demonstrated that low-quality RNGs like Linear Congruential Generator achieve nearly identical accuracy to high-quality alternatives while reducing computational overhead from 54× to 8× compared to round-to-nearest arithmetic and how recycling a buffer of high quality randoms results in too poor RNG for SR.

We extended the NPBench benchmark suite with dedicated error measurement capabilities, enabling systematic comparison of floating-point computation schemes across diverse scientific kernels. Our experimental analysis revealed that stochastic rounding's effectiveness is highly data-dependent,

providing up to three orders of magnitude error reduction for workloads with long accumulation chains and unidirectional value growth, while offering minimal benefits for symmetric distributions.

Finally, we validated our approach on components of the ICON climate model, demonstrating that complex Fortran applications can be transformed to use stochastic rounding with minimal engineering effort, though results highlight that SR benefits are application-specific rather than universal. These contributions provide the scientific computing community with both the tools and insights necessary to evaluate whether stochastic rounding can facilitate the transition from double to single precision in their specific applications.

Limitations and Possible Extensions. Our research focused primarily on enabling double-precision applications to run at single-precision, but the benefits of stochastic rounding become more pronounced at lower precisions. Since SR is most beneficial for long accumulations, and half-precision formats require only short accumulations before error explodes, SR could offer immediate accuracy benefits for applications transitioning to half-precision or lower. Future work should systematically explore how SR can enhance already low-precision applications and quantify the precision-accuracy trade-offs across different formats.

Technical limitations in our experimental setup prevented comprehensive evaluation of all desired precision formats. While DaCe has been extended to support half-precision, the existing Float16 implementation on CPU is non-functional, forcing us to rely on the StochasticRounding.jl package for some experiments. This hybrid approach, while functional, is cumbersome and limited our ability to run some experiements. Extending DaCe to robustly support alternative low-precision formats such as BFloat16, MiniFloat, and TensorFloat-32 would enable more comprehensive NPBench experiments and better characterize SR's impact across the full spectrum of reduced-precision formats.

Our analysis of random number generators revealed unexpected results that warrant further investigation. The circular buffer approach, despite using high-quality Mersenne Twister values, showed acceptable performance but weaker error reduction compared to simpler generators. We hypothesize this may be due to pattern repetition in the 10,000-element buffer, but systematic exploration of different buffer sizes and cycling strategies is needed to confirm this theory. Additionally, while our chosen lightweight RNGs (LCG and Xorshift) proved effective, extending the analysis to more sophisticated generators like the Permuted Congruential Generator could provide insights into the relationship between RNG quality and SR effectiveness.

The ICON velocity tendencies results present an intriguing puzzle that merits deeper investigation. The marginally worse performance compared to round-

to-nearest was unexpected given the theoretical advantages of SR. A detailed analysis of the computational patterns within these modules could reveal why SR fails to provide benefits in this context. Furthermore, exploring iterative scenarios—where module outputs are fed back as inputs over multiple time steps—could illuminate whether SR's advantages emerge over longer computational chains, potentially making it valuable for extended climate simulations even when individual module executions show no improvement.

Bibliography

[1] Jeffrey M. Alben, Paulius Micikevicius, Hong Wu, and Michael Y. Siu. Stochastic rounding of numerical values. `https://patents.google.com/patent/US10684824B2/en`, 2019. US Patent US10684824B2, Status: Active.

[2] R. C. M. Barnes, E. H. Cooke-Yarborough, and D. G. A. Thomas. An electronic digital computor using cold cathode counting tubes for storage. *Electronic Engineering*, 23:286–291, 1951. Archived by the Computer Conservation Society.

[3] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, 2019.

[4] Michael P. Connolly, Nicholas J. Higham, and Theo Mary. Stochastic Rounding and Its Probabilistic Backward Error Analysis. *SIAM Journal on Scientific Computing*, 43(1):A566–A585, January 2021.

[5] Matteo Croci, Massimiliano Fasi, Nicholas J. Higham, Theo Mary, and Mantas Mikaitis. Stochastic rounding: implementation, error analysis and applications. *Royal Society Open Science*, 9(3):211631, March 2022.

[6] Jack Dongarra, John Gunnels, Harun Bayraktar, Azzam Haidar, and Dan Ernst. Hardware trends impacting floating-point computations in scientific applications, 2024.

[7] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. 2015.

[8] Akash Haridas, Nagabhushana Rao Vadlamani, and Yuki Minamoto. Deep neural networks to correct sub-precision errors in cfd. *Applications in Energy and Combustion Science*, 12:100081, 2022.

[9] Sam Hatfield, Kristian Mogensen, Peter Düben, Nils Wedi, and Michail Diamantakis. Operational Single-Precision Earth-System Modelling at ECMWF. Conference presentation at EGU General Assembly 2021, 2021. Accessed: 2025-08-03.

[10] J. H. Jungclaus, S. J. Lorenz, H. Schmidt, V. Brovkin, N. Brüggemann, F. Chegini, T. Crüger, P. De-Vrese, V. Gayler, M. A. Giorgetta, O. Gutjahr, H. Haak, S. Hagemann, M. Hanke, T. Ilyina, P. Korn, J. Kröger, L. Linardakis, C. Mehlmann, U. Mikolajewicz, W. A. Müller, J. E. M. S. Nabel, D. Notz, H. Pohlmann, D. A. Putrasahan, T. Raddatz, L. Ramme, R. Redler, C. H. Reick, T. Riddick, T. Sam, R. Schneck, R. Schnur, M. Schupfner, J.-S. von Storch, F. Wachsmann, K.-H. Wieners, F. Ziemen, B. Stevens, J. Marotzke, and M. Claussen. The icon earth system model version 1.0. *Journal of Advances in Modeling Earth Systems*, 14(4):e2021MS002813, 2022. e2021MS002813 2021MS002813.

[11] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A Study of BFLOAT16 for Deep Learning Training, June 2019. arXiv:1905.12322 [cs].

[12] L. V. Kantorovich. Mathematical methods of organizing and planning production. *Management Science*, 6(4):366–422, 1960.

[13] Milan Klöwer, Peter V. Coveney, E. Adam Paxton, and Tim N. Palmer. Periodic orbits in chaotic dynamical systems simulated at low precision. *Scientific Reports*, 13:11410, 2023.

[14] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[15] Gabriel H. Loh. Stochastic rounding logic. `https://patents.google.com/patent/US10628124B2/en`, 2019. US Patent US10628124B2, Status: Active.

[16] Edward N. Lorenz. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, 20(2):130–141, 1963.

[17] Mantas Mikaitis. Stochastic Rounding: Algorithms and Hardware Accelerator. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6, Shenzhen, China, July 2021. IEEE.

[18] Franco Molteni and Fred Kucharski. A heuristic dynamical model of the north atlantic oscillation with a lorenz-type chaotic attractor. *Climate Dynamics*, 52(9):6173–6193, May 2019.

[19] Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding Gradient Noise Improves Learning for Very Deep Networks, November 2015. arXiv:1511.06807 [stat].

[20] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.*, 50(6):1–11, June 2015.

[21] E. Adam Paxton, Matthew Chantry, Milan Klöwer, Leo Saffin, and Tim Palmer. Climate Modeling in Low Precision: Effects of Both Deterministic and Stochastic Rounding. *Journal of Climate*, 35(4):1215–1229, February 2022.

[22] Yuki Uchino, Katsuhisa Ozaki, and Toshiyuki Imamura. Performance Enhancement of the Ozaki Scheme on Integer Matrix Multiplication Unit. *The International Journal of High Performance Computing Applications*, 39(3):462–476, May 2025. arXiv:2409.13313 [cs].

[23] Filip Váňa, Peter Düben, Simon Lang, Tim Palmer, Martin Leutbecher, Deborah Salmond, and Glenn Carver. Single Precision in Weather Forecasting Models: An Evaluation with the IFS. *Monthly Weather Review*, 145(2):495–502, February 2017.

[24] J.H. Wilkinson. *Rounding Errors in Algebraic Processes*. Dover books on advanced mathematics. Dover, 1994.

[25] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. NPBench: a benchmarking suite for high-performance NumPy. In *Proceedings of the ACM International Conference on Supercomputing*, pages 63–74, Virtual Event USA, June 2021. ACM.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. **In consultation with the supervisor**, one of the following two options must be selected:

☐ I hereby declare that I authored the work in question independently, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☒ I hereby declare that I authored the work in question independently. In doing so I only used the authorised aids, which included suggestions from the supervisor regarding language and content and generative artificial intelligence technologies. The use of the latter and the respective source declarations proceeded in consultation with the supervisor.

Title of paper or thesis:

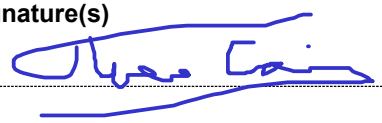
Towards Stochastic Rounding for Scientific Applications

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Creavin	Thomas

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guidelines.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zollikon, 15 August 2025	

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] For further information please consult the ETH Zurich websites, e.g. https://ethz.ch/en/the-eth-zurich/education/ai-in-education.html and https://library.ethz.ch/en/researching-and-publishing/scientific-writing-at-eth-zurich.html (subject to change).