

Reducing Numerical Precision Requirements in HPC Applications

William Dawson*
Katsuhisa Ozaki**
Prajval Kumar***
Jens Domke*
Takahito Nakajima*

* RIKEN R-CCS, **Shibaura Institute of Technology, ***IISER Thiruvananthapuram

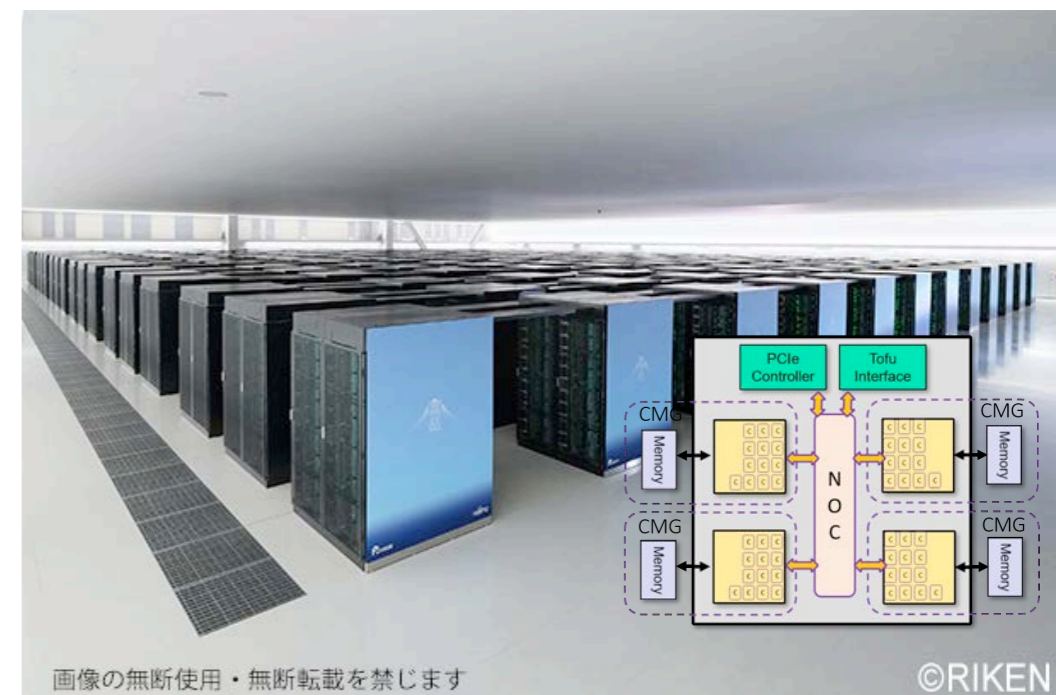
RIKEN R-CCS and supercomputer Fugaku

- Our team is located at the RIKEN Center for Computational Science in Kobe, Japan.
- Home to the supercomputer Fugaku.
 - #7 fastest supercomputer in the world (according to Top500 public data) as of June 2025.
- A64FX processor based on an ARM architecture.
 - CPU only nodes with 48 cores, 32 GB of memory, 512 bit vector length (SVE).
 - Each node has a theoretical peak of 3.4TFLOPs.
- A total of 158,976 nodes / 7,630,848 cores connected by Tofu interconnect D.
 - Compared to the fat nodes of current GPU machines, for Fugaku when you need more performance or memory, eagerly use more nodes.

View of Kobe Skyline



Supercomputer Fugaku at RIKEN R-CCS



画像の無断使用・無断転載を禁じます

©RIKEN

Future Computing Platforms at R-CCS

- Quantum Computing
 - In June of this year, an IBM Heron R2 quantum computer was installed at R-CCS.
 - Superconducting quantum computer.
 - 156 qubit machine, with a focus on hybrid quantum / classical supercomputer workflows.
- Development of FugakuNEXT
 - A successor to Fugaku slated for 2030.
 - An NVIDIA GPU machine with a CPU developed by Fujitsu (Monaka-X)
 - With my role as assistant working group leader of the applications areas, I am working on gathering apps from the community.
 - Our goal is to continuously benchmark and develop codes to get them ready for the next supercomputer - tell us what codes are important to your science!

IBM Heron R2



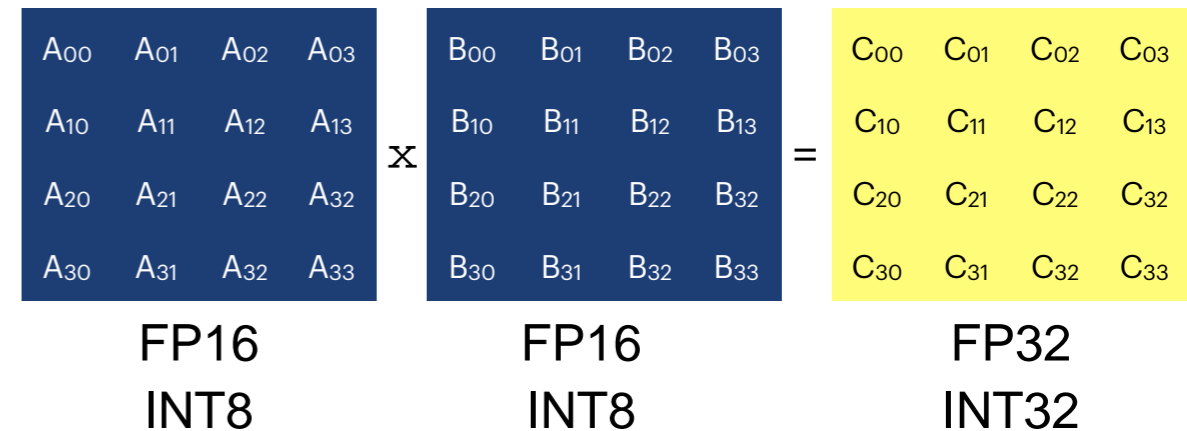
Application Areas for FugakuNEXT

1. Life Science
2. Materials / Energy
3. Climate Science / Weather
4. Earthquake / Tsunami Disaster Prevention
5. Manufacturing
6. Fundamental Science
7. Social Science, Digital Twins (Society 5.0)
8. Computational Science, Machine Learning, and Algorithms

Recent Trends on GPU Hardware

- High demands for deep learning compute resources will have a significant impact on the design of future supercomputers.
- Deep learning has very different requirements than traditional simulations.
 - Matrix multiplication is the key kernel.
 - Matrix multiplication in low precision is acceptable (or even preferable).
- Many modern GPUs contain “Tensor Processing Units”, which can multiply a 4x4 matrix in low precision, while accumulating in a higher precision.
- It is vital to exploit this capability.
 - The ratio of low precision / high precision performance is growing quickly.
 - Or we might say, the performance of high precision is stagnating...
- How can our HPC applications thrive under these conditions?

Tensor Core Multiplication



GPU Performance Trends

GPU Model	FP64	FP64 TC	FP16 (FP32)	INT8 (INT32)
A100 SXM	9.7 TFLOPS	19.5 TFLOPS	312 TFLOPS	624 TOPS
H100 SXM	34 TFLOPS	67 TFLOPS	990 TFLOPS	1979 TOPS
B200 HGX	37 TFLOPS	37 TFLOPS	2250 TFLOPS	4500 TOPS

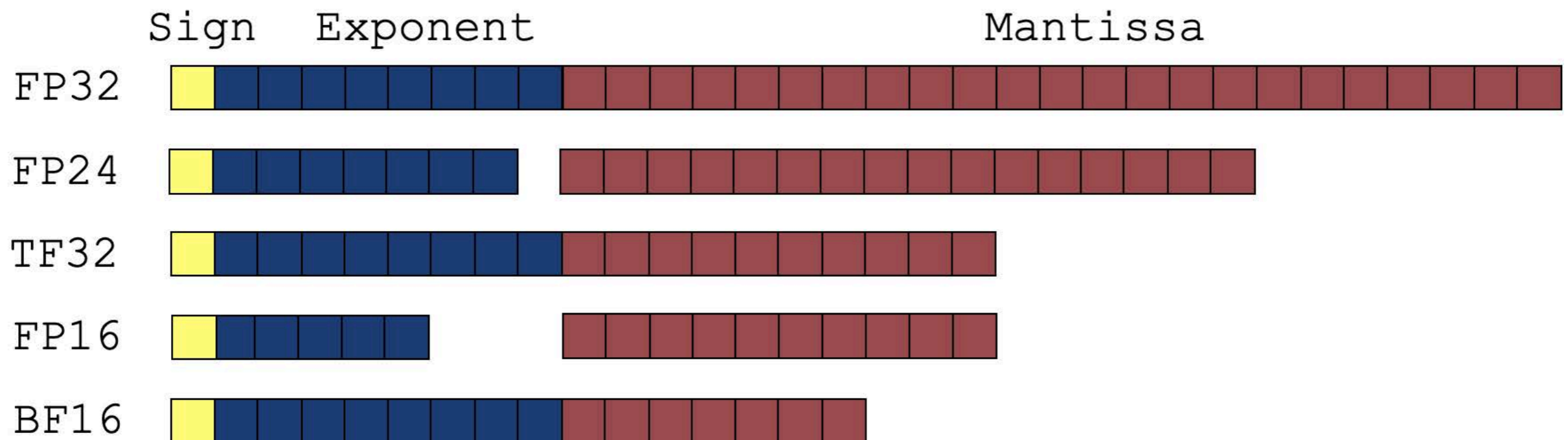
Low Precision Towards FugakuNEXT

- We can see similar specs in terms of ratio of low-precision to high-precision in the FugakuNEXT wishlist.
 - Depending on real hardware roadmaps, these ratios may change.
- We also see that while FP64 performance may increase by about a factor of five, the number is similar for the memory bandwidth.
- If your memory bandwidth bound, your speedup is going to be below 5x all the way in 2030?
 - But if you used low precision for your data, this could go to 10x, 20x, ...
 - The programming language and library support will be there for FugakuNEXT.
- In the past, pursuing low precision may have not seemed worth it - risk of a wrong solution for a mere 2x speedup. But now the timing is right.

Capabilities	CPU	Accelerator	Fugaku	Fugaku Comparison
FP64 Vector	> 48 PFLOPS	> 2.6 EFLOPS	537 PFLOPS	> 5.7 x
FP16 Matrix	> 1.5 EFLOPS	> 150 EFLOPS	2.15 EFLOPS	> 70.5 x
FP8 Matrix	> 3.0 EFLOPS	> 300 EFLOPS	—	
Memory Size	> 10 PiB	> 10 PiB	4.85 PiB	> 4.1 x
Memory Bandwidth	> 7 PB/s	> 800 PB/s	163 PB/s	> 4.9 x

Basics of Floating Point Numbers

- Floating point numbers are represented in terms of an sign bit, exponent, and mantissa.
 - “Floating point” so we use the exponent to shift the value (in binary) until 1 is the first digit, which is then stored implicitly.
- Several exotic floating point formats (TF32 and BF16) have been suggested for deep learning applications (less mantissa, more exponent).
- Warning: precision is limited by both the exponent and the mantissa.
- Many libraries exist for emulating arbitrary precisions, such as MPFR.



Test Algorithm from Electronic Structure Calculations

- One bottleneck in LCAO electronic structure codes is calculating the density matrix from a given Hamiltonian.
- Because the ratio of basis functions to occupied orbitals is not too large in LCAO codes, it's more efficient to just directly diagonalize the matrix using a dense solver.
 - Eigenvectors are then used to build the “density matrix” P .
 - P is a projection matrix on to the subspace spanned by the N lowest eigenvectors.
- We could explore the precision requirements of an eigenvalue solver, but there are many different algorithms and implementations are not trivial.

Dense Eigensolver Approach

INPUT: H, S the $M \times M$ Hamiltonian and Overlap.

$$HV = VSE$$

$$P = V[:, :N]V[:, :N]^T$$

OUTPUT: P is the $M \times M$ density matrix.

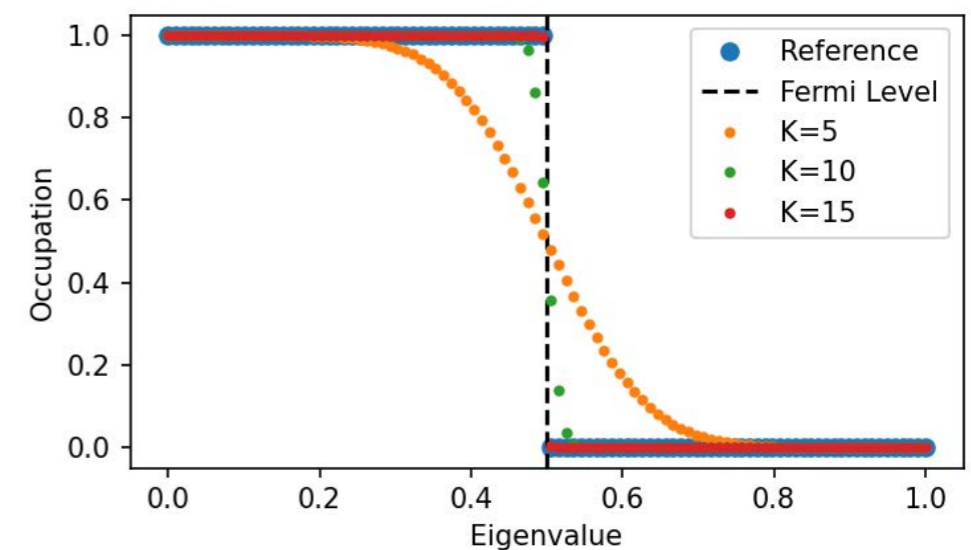
Test Algorithm from Electronic Structure Calculations

- There exists alternative algorithms for constructing P , without perform diagonalization.
- One such example is called purification.
 - Formulate the problem in terms of a recurrence relation in terms of matrix polynomials.
 - A large class of these and related methods exist, so I show the simplest one only.
 - Can beat dense eigenvalue solver even in the dense case on specific hardware.
- Since purification relies on matrix multiplication, it is easy to experiment on using different numerical precisions.

McWeeny Purification Algorithm

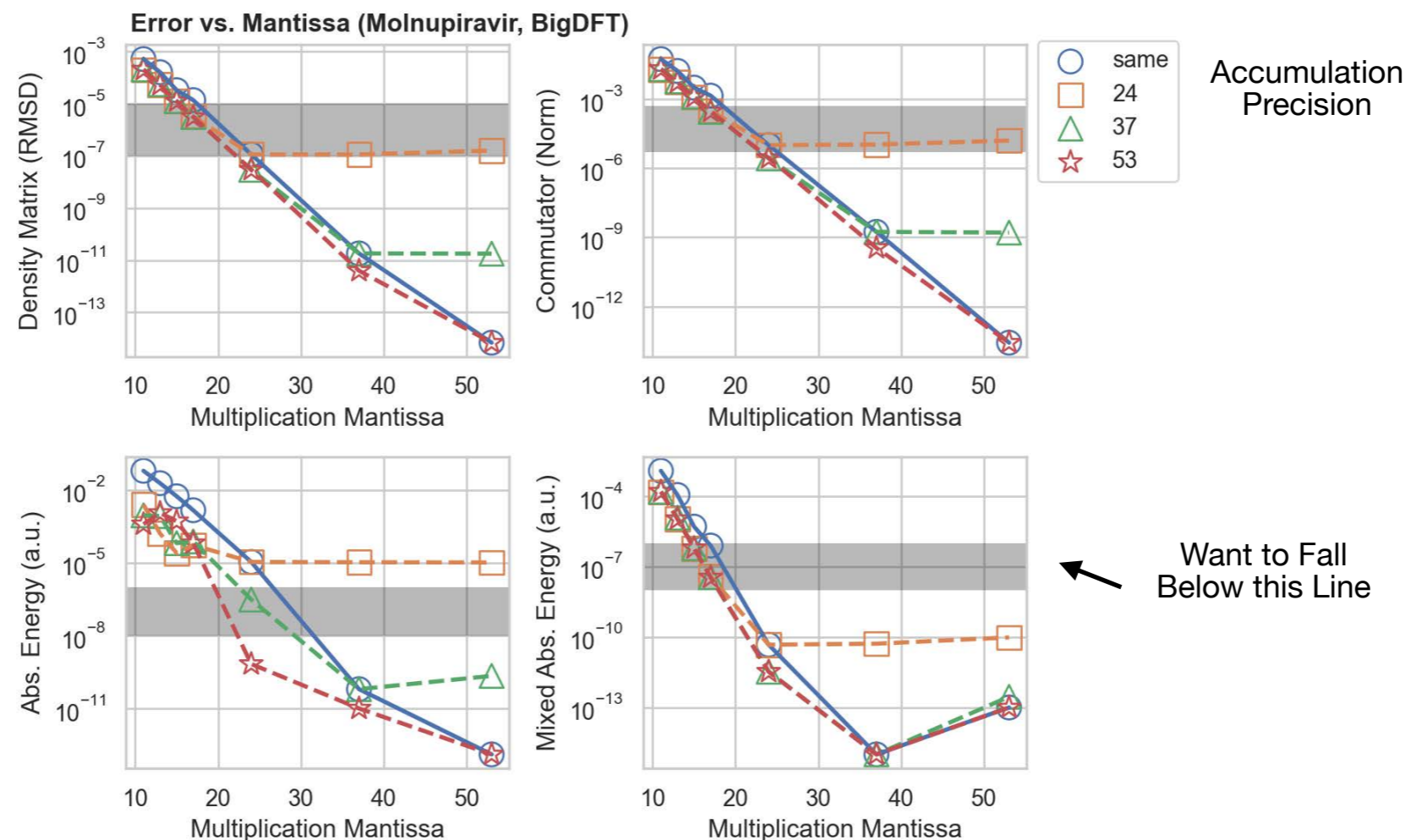
$$P_{k+1} = 3P_k^2 - 2P_k^3$$
$$P_0 = \frac{\lambda}{2}(\mu I - H) + \frac{1}{2}I$$

Visualization of Purification



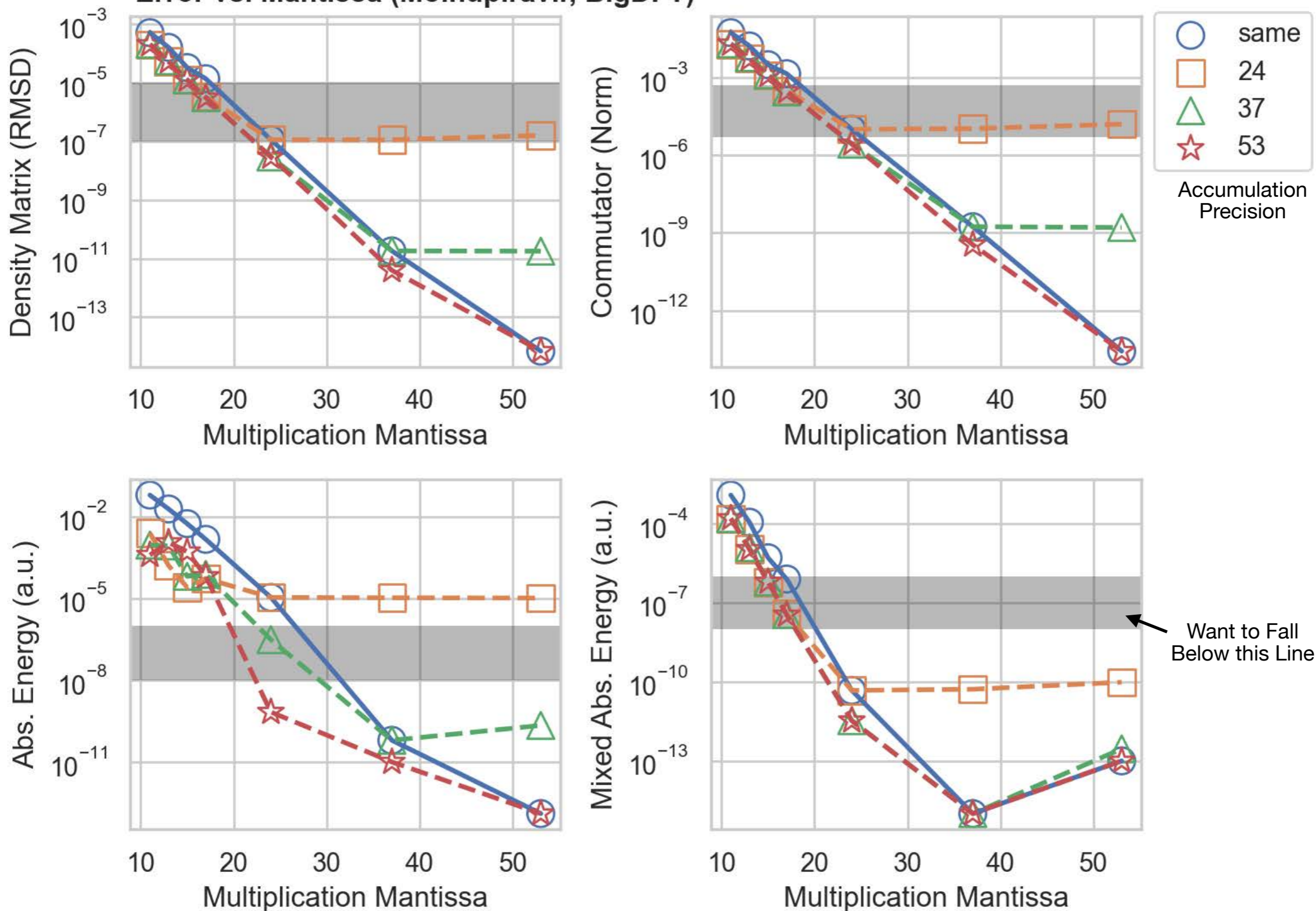
How Much Precision is Needed? (1)

- We implement the purification approach using MPFR, so that we can investigate how much precision we really need.
 - Multiplication Mantissa and Accumulation Mantissa set separately.
- We measure three error measures (RMSD of the error in P, $\|PH - HP\|$ (commutator), and the error in the energy ($\text{Tr}(HP)$)).
- Iterative refinement (mixed abs. energy plot) can help the energy value (take one step with full precision), but does not help the other measures.



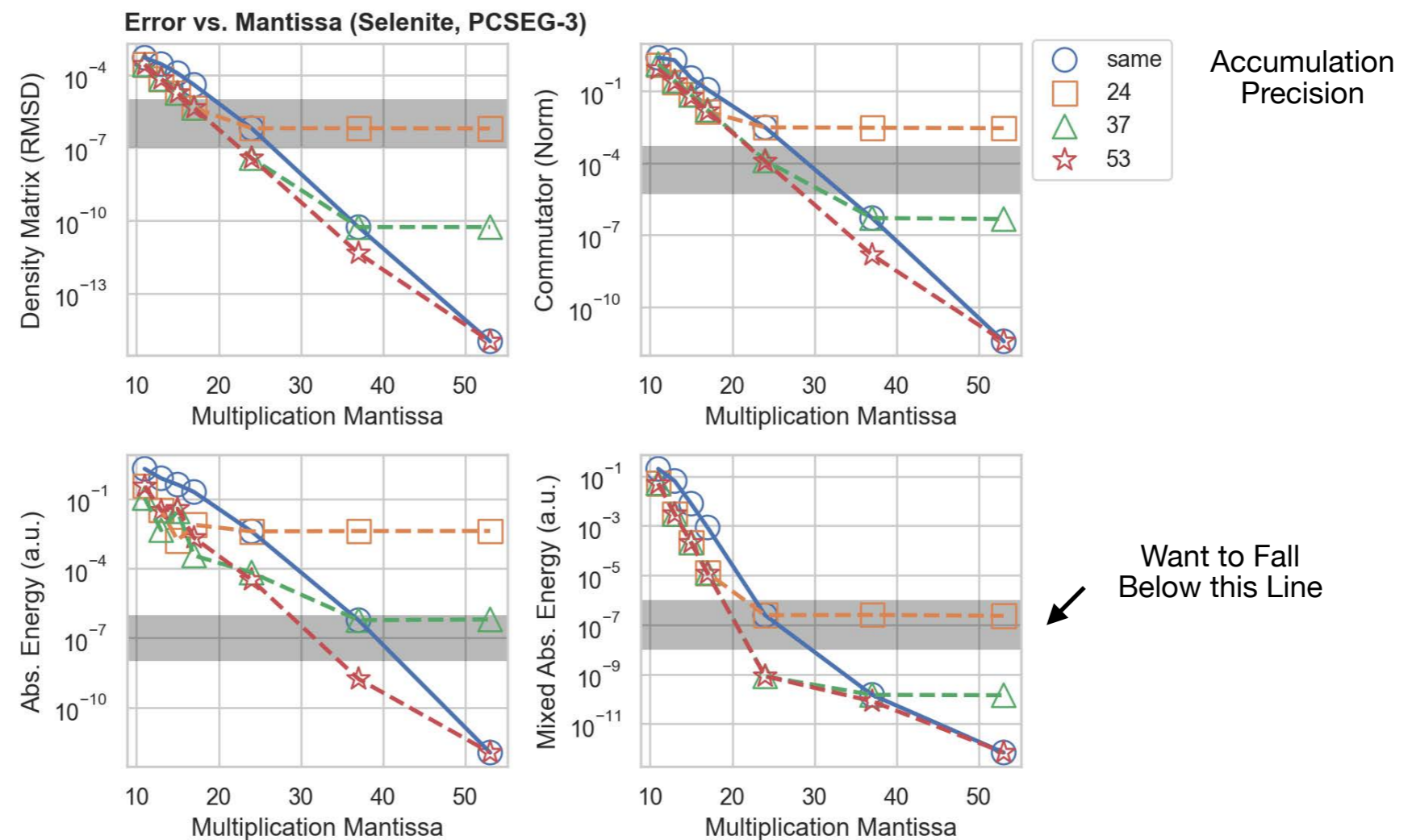
How Much Precision is Needed? (1)

Error vs. Mantissa (Molnupiravir, BigDFT)



How Much Precision is Needed? (2)

- Precision requirement depends on the type of LCAO basis.
 - Poorly conditioned basis sets (used for higher precision calculations) need higher precision calculations.
- Take away: switching to single precision isn't safe, yet double precision gives us far more precision than we really need.



Approximation Methods

- There exist various methods to try and emulate high precision multiplications as a sequence of low precision multiplies.

- We don't need double precision, we need something between single and double, so there should be a cheaper emulation strategy.

- Markidis method [1] - attempt to use FP16 tensor cores to recover full single precision.

- $A \approx A^0 + A^1$
- $A^0 = \text{FP16}[A]$, $A^1 = \text{FP16}[A - A^0]$.
- $AB \approx \text{FP32}[A^0 B^0 + A^0 B^1 + A^1 B^0 + A^1 B^1]$.
- Usually truncate the low order $A^1 B^1$ term. If $A = B$, we don't need to compute $A^1 B^0$.

- Ozaki-scheme [2]

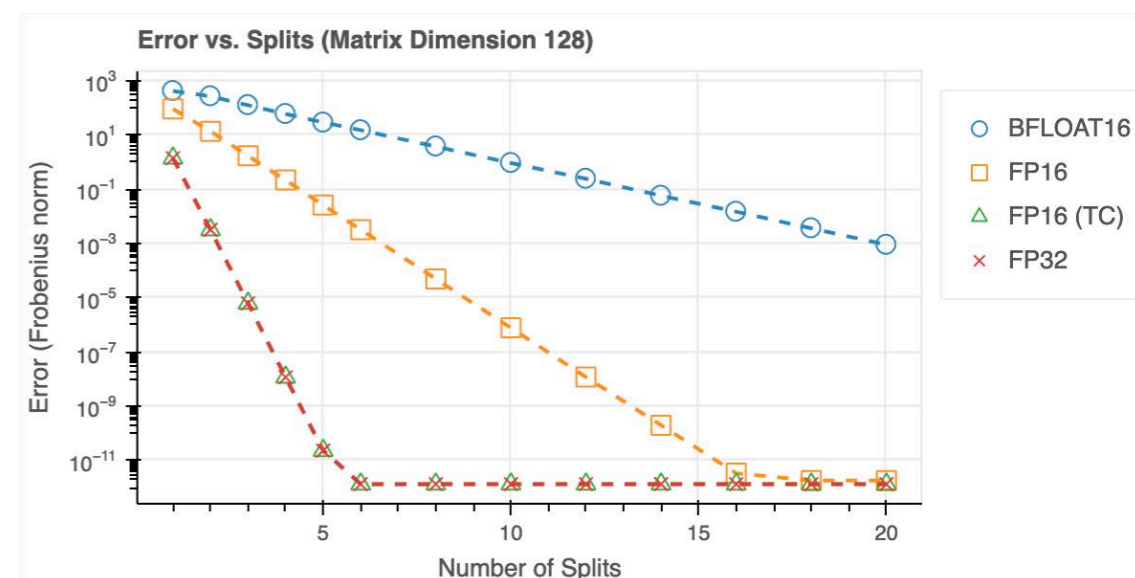
- Can be applied to any precision.
- Error is controlled by the number of splits (S).
- Cost grows $(S + 1)S / 2$.
- Error depends on the accumulation precision.

- For both: You should apply a scaling operation to prevent the exponent from going out of range.

Pictorial Representation of the Ozaki Scheme with Four Splits

$$\begin{array}{l}
 \mathbf{A} = \mathbf{A}_1 + \mathbf{A}_2 + \mathbf{A}_3 + \mathbf{A}_4 \\
 \mathbf{B} = \mathbf{B}_1 + \mathbf{B}_2 + \mathbf{B}_3 + \mathbf{B}_4
 \end{array}$$

$$\sum \begin{array}{cccc}
 \mathbf{A}_1\mathbf{B}_1 & \mathbf{A}_1\mathbf{B}_2 & \mathbf{A}_1\mathbf{B}_3 & \mathbf{A}_1\mathbf{B}_4 \\
 \mathbf{A}_2\mathbf{B}_1 & \mathbf{A}_2\mathbf{B}_2 & \mathbf{A}_2(\mathbf{B}_3 + \mathbf{B}_4) & \\
 \mathbf{A}_3\mathbf{B}_1 & \mathbf{A}_3(\mathbf{B}_2 + \mathbf{B}_3 + \mathbf{B}_4) & & \\
 \mathbf{A}_4\mathbf{B}_1 & & &
 \end{array} = \mathbf{AB}$$



[1] S. Markidis, et al, Nvidia tensor core programmability, performance & precision, in *2018 IEEE international parallel and distributed processing symposium workshops* pp. 522–531 (2018).

[2] K. Ozaki, et al, Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications, *Numerical Algorithms* 59, 95 (2012).

Approximation Methods

- There exist various methods to try and emulate high precision multiplications as a sequence of low precision multiplies.

- We don't need double precision, we need something between single and double, so there should be a cheaper emulation strategy.

- Markidis method [1] - attempt to use FP16 tensor cores to recover full single precision.

- $A \approx A^0 + A^1$
- $A^0 = \text{FP16}[A]$, $A^1 = \text{FP16}[A - A^0]$.
- $AB \approx \text{FP32}[A^0 B^0 + A^0 B^1 + A^1 B^0 + A^1 B^1]$.
- Usually truncate the low order $A^1 B^1$ term. If $A = B$, we don't need to compute $A^1 B^0$.

- Ozaki-scheme [2]

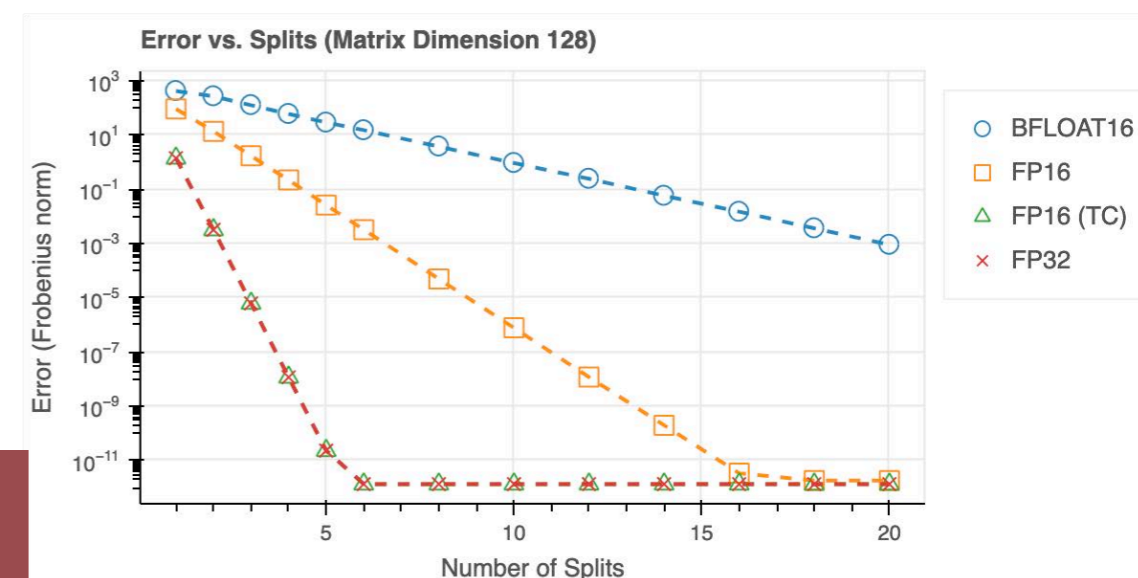
- Can be applied to any precision.
- Error is controlled by the number of splits (S).
- Cost grows $(S + 1)S / 2$.
- Error depends on the accumulation precision.

- For both: You should apply a scaling operation to prevent the exponent from going out of range.

Pictorial Representation of the Ozaki Scheme with Four Splits

$$\begin{array}{l}
 \mathbf{A} = \mathbf{A}_1 + \mathbf{A}_2 + \mathbf{A}_3 + \mathbf{A}_4 \\
 \mathbf{B} = \mathbf{B}_1 + \mathbf{B}_2 + \mathbf{B}_3 + \mathbf{B}_4
 \end{array}$$

$$\sum \begin{array}{cccc}
 \mathbf{A}_1\mathbf{B}_1 & \mathbf{A}_1\mathbf{B}_2 & \mathbf{A}_1\mathbf{B}_3 & \mathbf{A}_1\mathbf{B}_4 \\
 \mathbf{A}_2\mathbf{B}_1 & \mathbf{A}_2\mathbf{B}_2 & \mathbf{A}_2(\mathbf{B}_3 + \mathbf{B}_4) & \\
 \mathbf{A}_3\mathbf{B}_1 & \mathbf{A}_3(\mathbf{B}_2 + \mathbf{B}_3 + \mathbf{B}_4) & & \\
 \mathbf{A}_4\mathbf{B}_1 & & &
 \end{array} = \mathbf{AB}$$

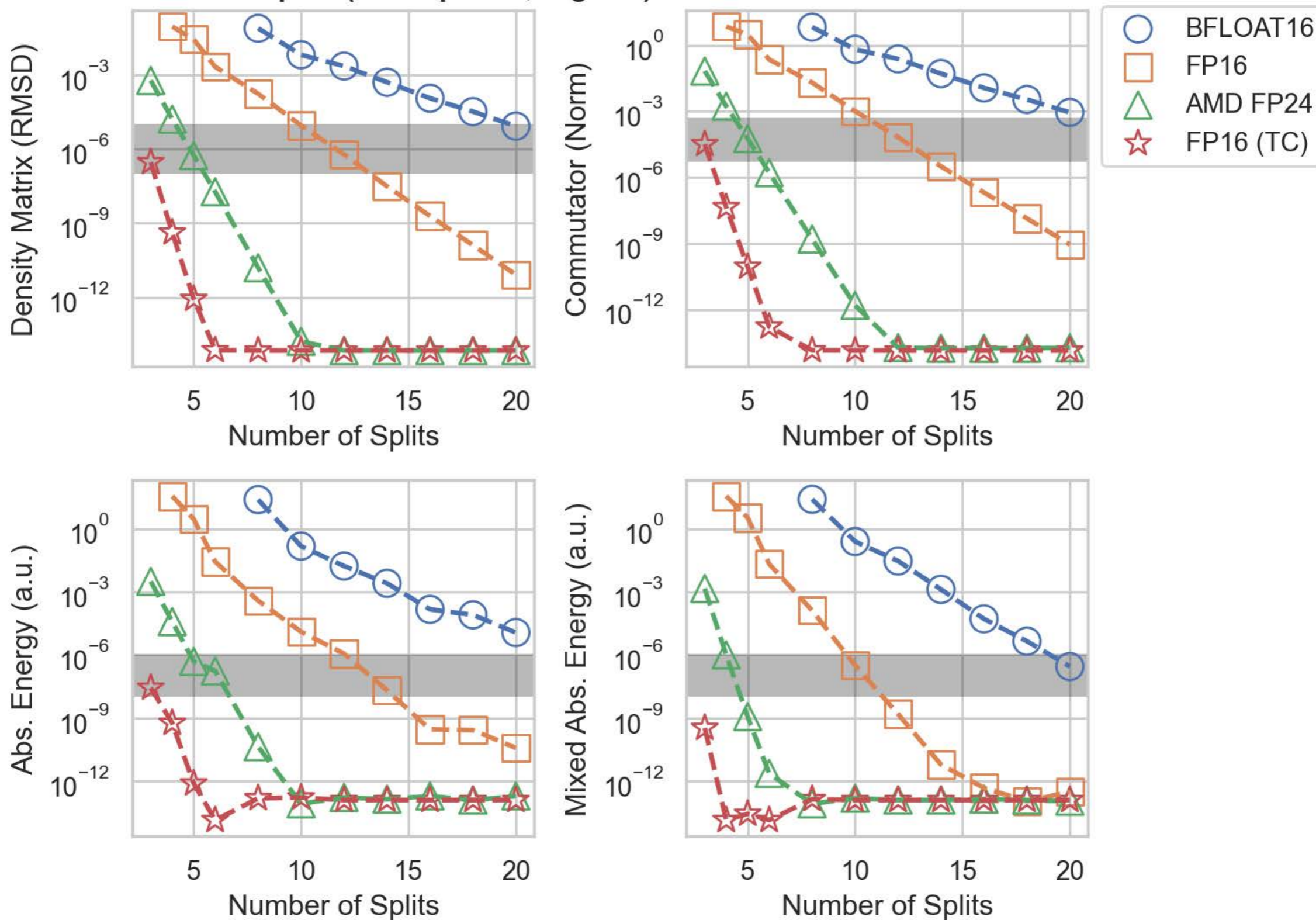


[1] S. Markidis, et al, Nvidia tensor core programmability, performance & precision, in *2018 IEEE international parallel and distributed processing symposium workshops* pp. 522–531 (2018).

[2] K. Ozaki, et al, Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications, *Numerical Algorithms* 59, 95 (2012).

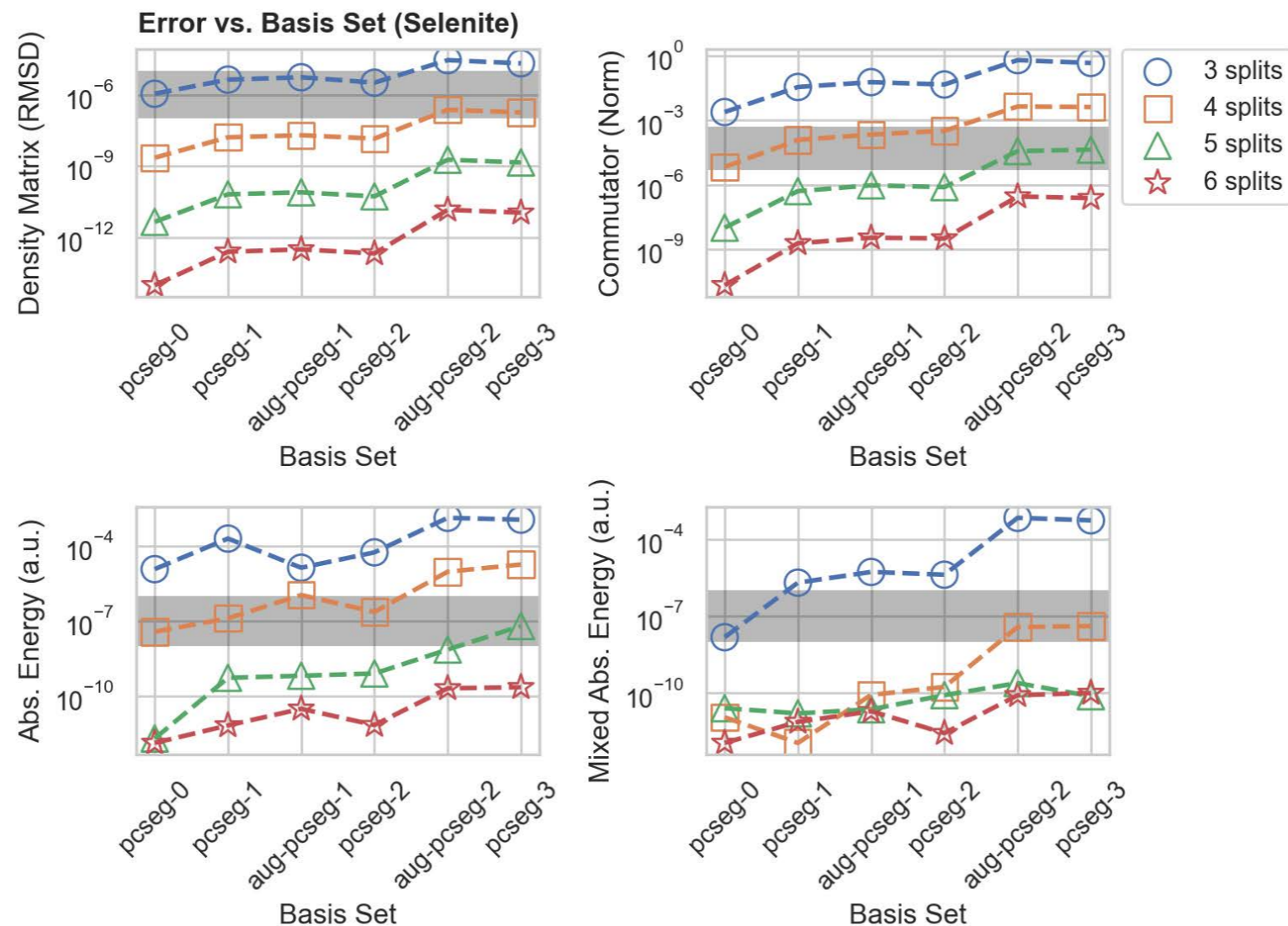
Using the Ozaki Scheme (1)

Error vs. Splits (Molnupiravir, BigDFT)



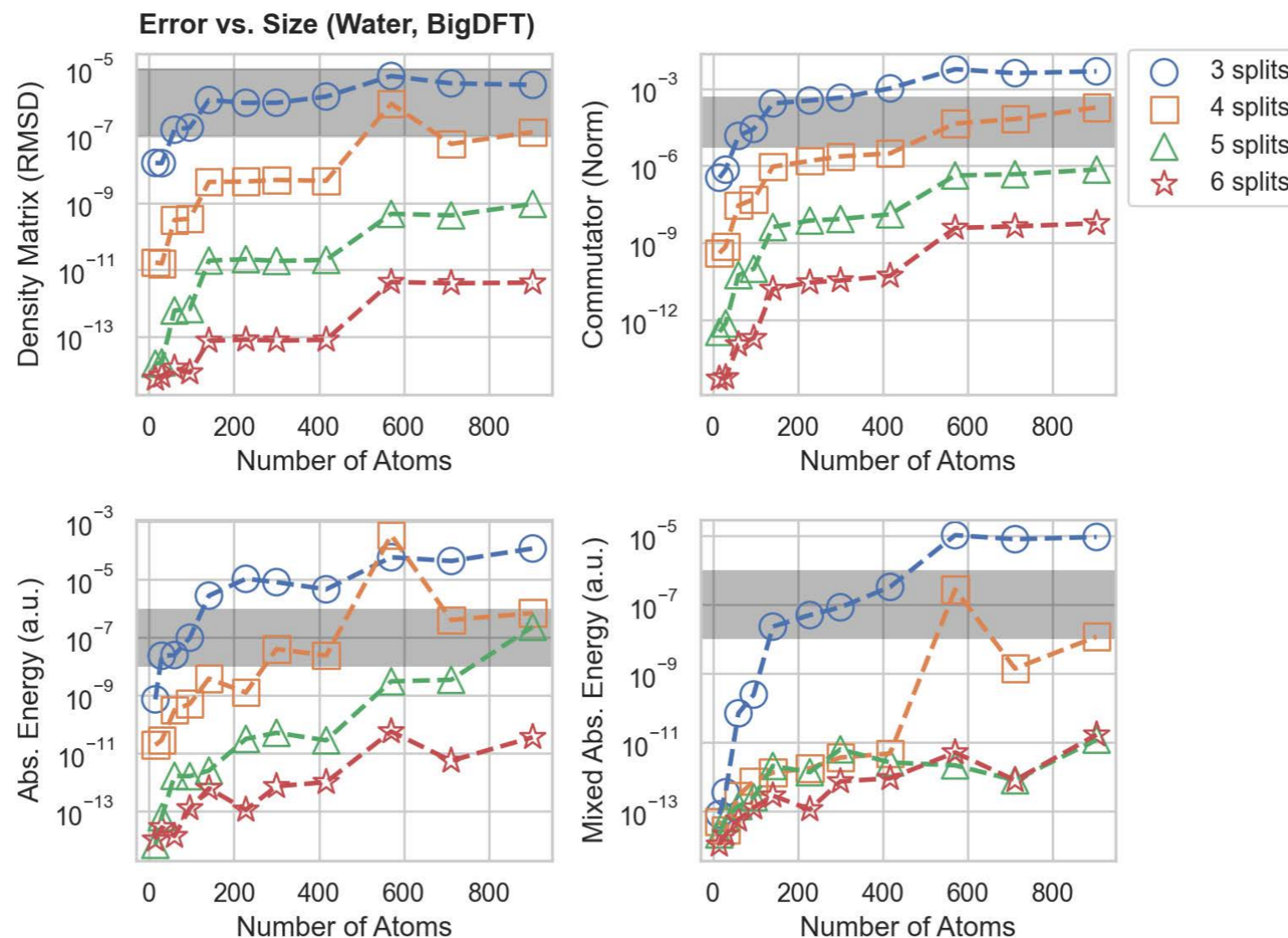
Using the Ozaki Scheme (2)

- Precision requirements depend on the basis set employed.
- A larger basis set leads to both larger matrices, and a larger range of values in the matrix.
- We can see the effect by using Gaussian basis sets of progressively higher quality, or including diffuse functions.



Using the Ozaki Scheme (3)

- Error will accumulate with an approximate scheme. Thus, the larger the matrix, the more error you will get.
- However, experimenting on different sizes of systems showed a slow growth in the error with the system size.



Comparison to the Markidis Method

- The Markidis method provides a hardware efficient way to emulate higher precisions with very low overhead (1-2 extra multiplications) - already tested by a different group [1].
 - $A \approx A^0 + A^1$
 - $A^0 = \text{FP16}[A]$, $A^1 = \text{FP16}[A - A^0]$.
 - $AB \approx \text{FP32}[A^0 B^0 + A^0 B^1 + A^1 B^0 + A^1 B^1]$.
 - We also apply scaling to keep the exponent in range.
- Downside: it can at best try to get single precision accuracy, a struggles for bigger basis sets.
- (A) PCSEG-1; (B) AUG-PCSEG-1; (C) PCSEG-3.

system	1 Term	2 Terms	3 Terms	4 Terms	4-MPFR	Ozaki (5)	FP64
RMSD errors							
Molnupiravir	2.2×10^{-4}	1.4×10^{-4}	5.1×10^{-7}	4.4×10^{-7}	2.1×10^{-7}	8.5×10^{-13}	7.0×10^{-15}
Water 237	6.1×10^{-5}	3.7×10^{-5}	2.0×10^{-7}	1.8×10^{-7}	1.1×10^{-7}	4.2×10^{-10}	8.1×10^{-15}
Selenite/A	5.6×10^{-4}	3.2×10^{-4}	1.1×10^{-6}	1.1×10^{-6}	5.8×10^{-7}	6.5×10^{-11}	1.1×10^{-15}
Selenite/B	3.8×10^{-4}	2.3×10^{-4}	1.2×10^{-6}	1.3×10^{-6}	8.6×10^{-7}	7.8×10^{-11}	1.3×10^{-15}
Selenite/C	2.7×10^{-4}	1.6×10^{-4}	2.8×10^{-6}	2.9×10^{-6}	8.0×10^{-7}	1.4×10^{-9}	1.0×10^{-15}
Commutator errors							
Molnupiravir	2.2×10^{-2}	1.3×10^{-2}	5.5×10^{-5}	4.6×10^{-5}	2.3×10^{-5}	8.6×10^{-11}	2.8×10^{-14}
Water 237	6.5×10^{-2}	4.1×10^{-2}	2.2×10^{-4}	1.9×10^{-4}	1.1×10^{-4}	4.7×10^{-7}	2.9×10^{-13}
Selenite/A	4.0×10^{-1}	1.5×10^{-1}	4.2×10^{-3}	4.5×10^{-3}	4.3×10^{-3}	5.1×10^{-7}	4.1×10^{-13}
Selenite/B	4.1×10^{-1}	1.6×10^{-1}	5.2×10^{-3}	5.7×10^{-3}	4.6×10^{-3}	9.5×10^{-7}	1.4×10^{-12}
Selenite/C	1.3×10^0	6.6×10^{-1}	1.4×10^{-2}	1.6×10^{-2}	1.2×10^{-2}	4.3×10^{-5}	3.8×10^{-12}
Abs. energy errors							
Molnupiravir	2.6×10^{-3}	1.9×10^{-3}	8.7×10^{-5}	8.5×10^{-5}	1.2×10^{-5}	4.8×10^{-13}	1.1×10^{-13}
Water 237	1.9×10^{-2}	2.2×10^{-5}	1.2×10^{-3}	1.2×10^{-3}	3.6×10^{-4}	3.5×10^{-9}	1.7×10^{-12}
Selenite/A	1.3×10^{-1}	1.4×10^{-3}	4.3×10^{-3}	4.3×10^{-3}	1.0×10^{-3}	5.4×10^{-10}	0
Selenite/B	5.8×10^{-2}	8.7×10^{-2}	8.5×10^{-3}	8.4×10^{-3}	2.2×10^{-3}	6.5×10^{-10}	4.5×10^{-13}
Selenite/C	3.5×10^{-1}	2.1×10^{-1}	1.3×10^{-2}	1.3×10^{-2}	4.0×10^{-3}	6.3×10^{-8}	9.1×10^{-13}
Mixed abs. energy errors							
Molnupiravir	1.9×10^{-4}	6.2×10^{-5}	1.2×10^{-9}	9.8×10^{-10}	2.0×10^{-10}	1.4×10^{-13}	1.1×10^{-13}
Water 237	1.6×10^{-3}	5.3×10^{-4}	2.3×10^{-8}	1.9×10^{-8}	6.9×10^{-9}	8.0×10^{-13}	1.7×10^{-12}
Selenite/A	8.7×10^{-3}	2.4×10^{-3}	8.4×10^{-8}	8.3×10^{-8}	2.9×10^{-8}	1.6×10^{-11}	0
Selenite/B	8.8×10^{-3}	3.0×10^{-3}	2.4×10^{-7}	2.8×10^{-7}	6.9×10^{-8}	2.3×10^{-11}	4.5×10^{-13}
Selenite/C	5.4×10^{-2}	1.6×10^{-2}	4.2×10^{-6}	4.6×10^{-6}	5.3×10^{-7}	7.5×10^{-11}	9.1×10^{-13}

Purification iterations

Comparison to the Markidis Method

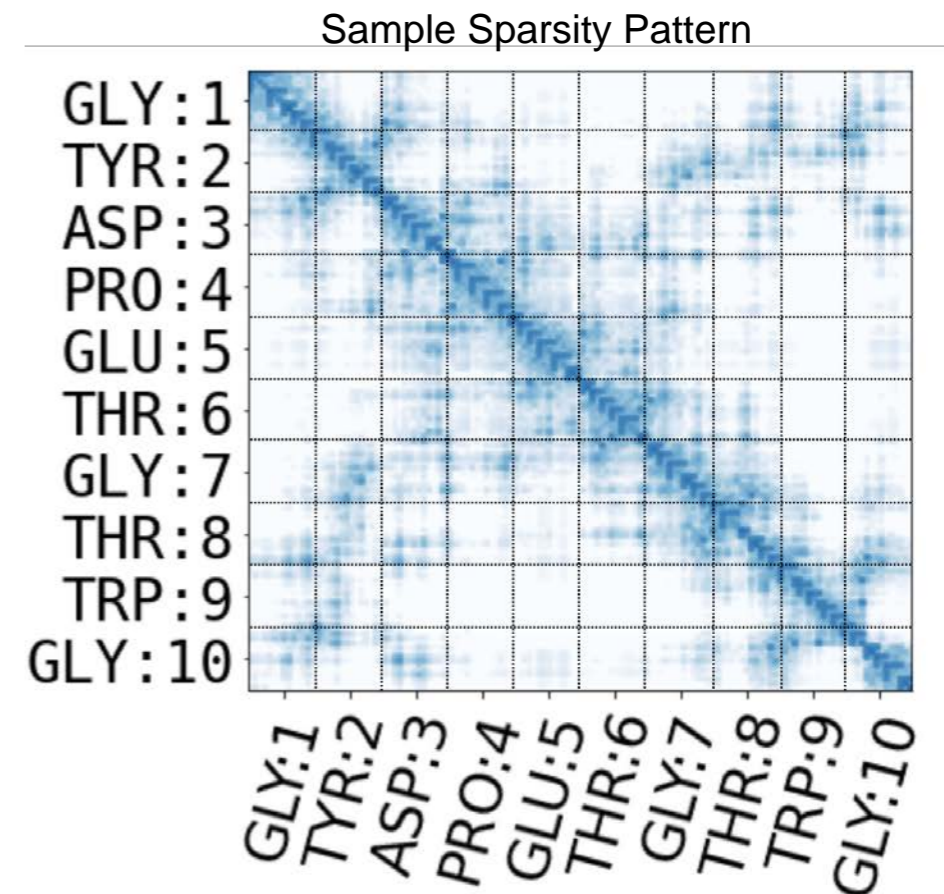
system	1 Term	2 Terms	3 Terms	4 Terms	4-MPFR	Ozaki (5)	FP64
RMSD errors							
Molnupiravir	2.2×10^{-4}	1.4×10^{-4}	5.1×10^{-7}	4.4×10^{-7}	2.1×10^{-7}	8.5×10^{-13}	7.0×10^{-15}
Water 237	6.1×10^{-5}	3.7×10^{-5}	2.0×10^{-7}	1.8×10^{-7}	1.1×10^{-7}	4.2×10^{-10}	8.1×10^{-15}
Selenite/A	5.6×10^{-4}	3.2×10^{-4}	1.1×10^{-6}	1.1×10^{-6}	5.8×10^{-7}	6.5×10^{-11}	1.1×10^{-15}
Selenite/B	3.8×10^{-4}	2.3×10^{-4}	1.2×10^{-6}	1.3×10^{-6}	8.6×10^{-7}	7.8×10^{-11}	1.3×10^{-15}
Selenite/C	2.7×10^{-4}	1.6×10^{-4}	2.8×10^{-6}	2.9×10^{-6}	8.0×10^{-7}	1.4×10^{-9}	1.0×10^{-15}
Commutator errors							
Molnupiravir	2.2×10^{-2}	1.3×10^{-2}	5.5×10^{-5}	4.6×10^{-5}	2.3×10^{-5}	8.6×10^{-11}	2.8×10^{-14}
Water 237	6.5×10^{-2}	4.1×10^{-2}	2.2×10^{-4}	1.9×10^{-4}	1.1×10^{-4}	4.7×10^{-7}	2.9×10^{-13}
Selenite/A	4.0×10^{-1}	1.5×10^{-1}	4.2×10^{-3}	4.5×10^{-3}	4.3×10^{-3}	5.1×10^{-7}	4.1×10^{-13}
Selenite/B	4.1×10^{-1}	1.6×10^{-1}	5.2×10^{-3}	5.7×10^{-3}	4.6×10^{-3}	9.5×10^{-7}	1.4×10^{-12}
Selenite/C	1.3×10^0	6.6×10^{-1}	1.4×10^{-2}	1.6×10^{-2}	1.2×10^{-2}	4.3×10^{-5}	3.8×10^{-12}
Abs. energy errors							
Molnupiravir	2.6×10^{-3}	1.9×10^{-3}	8.7×10^{-5}	8.5×10^{-5}	1.2×10^{-5}	4.8×10^{-13}	1.1×10^{-13}
Water 237	1.9×10^{-2}	2.2×10^{-5}	1.2×10^{-3}	1.2×10^{-3}	3.6×10^{-4}	3.5×10^{-9}	1.7×10^{-12}
Selenite/A	1.3×10^{-1}	1.4×10^{-3}	4.3×10^{-3}	4.3×10^{-3}	1.0×10^{-3}	5.4×10^{-10}	0
Selenite/B	5.8×10^{-2}	8.7×10^{-2}	8.5×10^{-3}	8.4×10^{-3}	2.2×10^{-3}	6.5×10^{-10}	4.5×10^{-13}
Selenite/C	3.5×10^{-1}	2.1×10^{-1}	1.3×10^{-2}	1.3×10^{-2}	4.0×10^{-3}	6.3×10^{-8}	9.1×10^{-13}
Mixed abs. energy errors							
Molnupiravir	1.9×10^{-4}	6.2×10^{-5}	1.2×10^{-9}	9.8×10^{-10}	2.0×10^{-10}	1.4×10^{-13}	1.1×10^{-13}
Water 237	1.6×10^{-3}	5.3×10^{-4}	2.3×10^{-8}	1.9×10^{-8}	6.9×10^{-9}	8.0×10^{-13}	1.7×10^{-12}
Selenite/A	8.7×10^{-3}	2.4×10^{-3}	8.4×10^{-8}	8.3×10^{-8}	2.9×10^{-8}	1.6×10^{-11}	0
Selenite/B	8.8×10^{-3}	3.0×10^{-3}	2.4×10^{-7}	2.8×10^{-7}	6.9×10^{-8}	2.3×10^{-11}	4.5×10^{-13}
Selenite/C	5.4×10^{-2}	1.6×10^{-2}	4.2×10^{-6}	4.6×10^{-6}	5.3×10^{-7}	7.5×10^{-11}	9.1×10^{-13}

Purification iterations

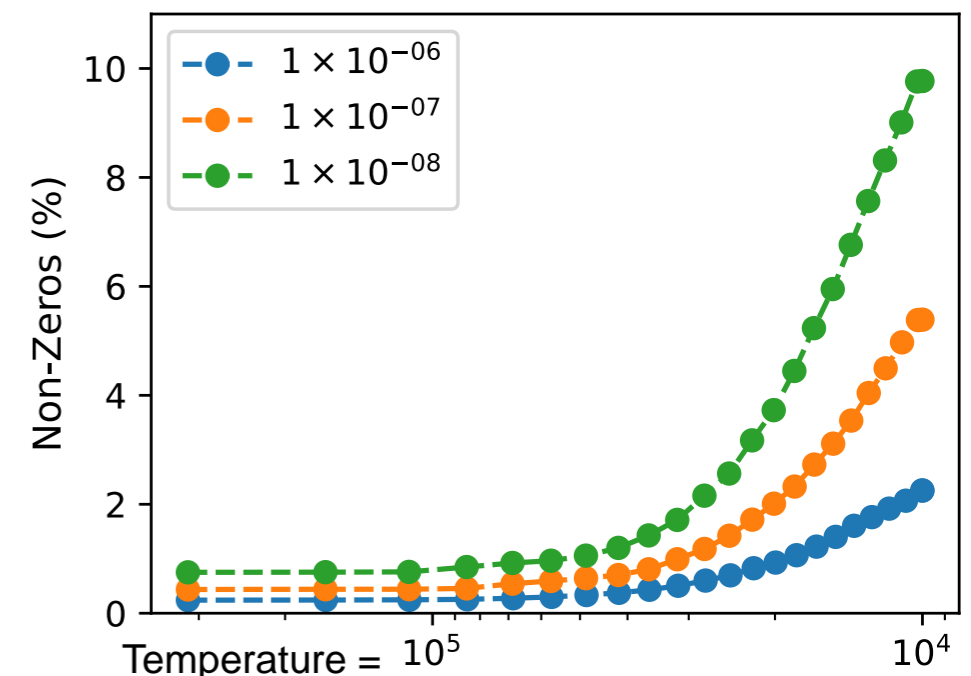
[1] Finkelstein, Joshua, Justin S. Smith, Susan M. Mniszewski, Kipton Barros, Christian FA Negre, Emanuel H. Rubensson, and Anders MN Niklasson. "Mixed precision Fermi-operator expansion on tensor cores from a machine learning perspective." *Journal of Chemical Theory and Computation* 17, no. 4 (2021): 2256-2265.

A Broader Lesson to Consider - 1

- For $O(N)$ methods of large system, we use sparse matrix methods.
 - Sparse Matrix - Sparse Matrix multiplies where we are filtering small matrix values.
- Thus, it is not so easy to exploit the dense tensor cores, and the operation is memory bandwidth bound.
- The design of tensor cores teaches us to think of precision in two stages: the precision of the matrix going in, and the precision of the accumulated result.
 - These do not have to be the same!
- In our MPFR experiments, we found that single precision was limited by the number of bits used for accumulation.
 - This suggests that mixed precision schemes could be useful.



Some Algorithms have Fill In Over Iterations



A Broader Lesson to Consider - 2

- Low precision going in: reduce the cost of moving data from memory, sending messages to other processes.
- High precision going out: do actual ops on the data in double precision so we can accurately accumulate.
- As an experiment, we check the error in the energy if we perform purification on matrices where all intermediate matrices are converted to single precision (simulate single precision communication).
- Value of ε controls the sparsity of the matrix \rightarrow error in the final energy. For a larger values of ε , the error is unaffected by precision of communication.

Energy Errors in Hartree	Precision	$\varepsilon = 1 \times 10^{-5}$	$\varepsilon = 1 \times 10^{-6}$	$\varepsilon = 1 \times 10^{-8}$
			Bulk Silicon	BigDFT
	FP64	1.827×10^{-3}	5.904×10^{-5}	3.282×10^{-5}
	FP32	1.832×10^{-3}	6.016×10^{-5}	3.280×10^{-5}
		Water Bubble	NTChem (PCSEG-1)	
	FP64	1.030×10^{-3}	3.452×10^{-5}	2.440×10^{-7}
	FP32	1.038×10^{-3}	3.261×10^{-5}	5.289×10^{-5}

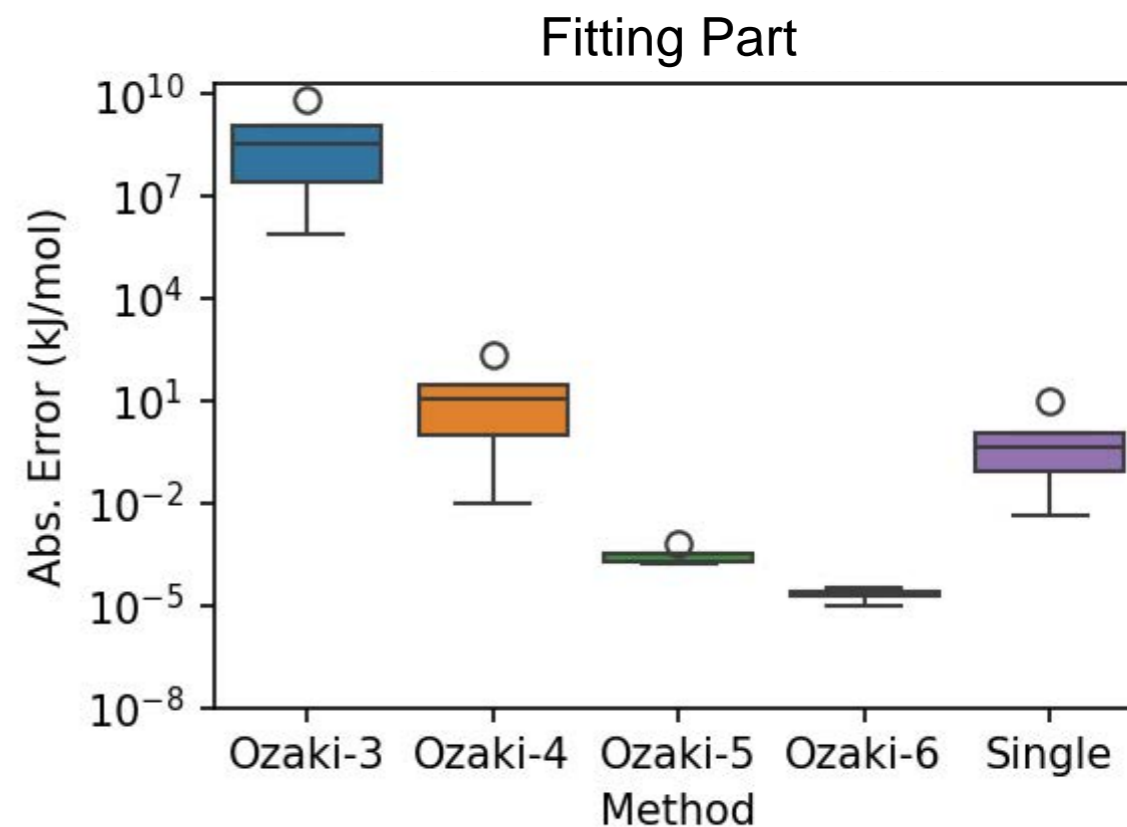
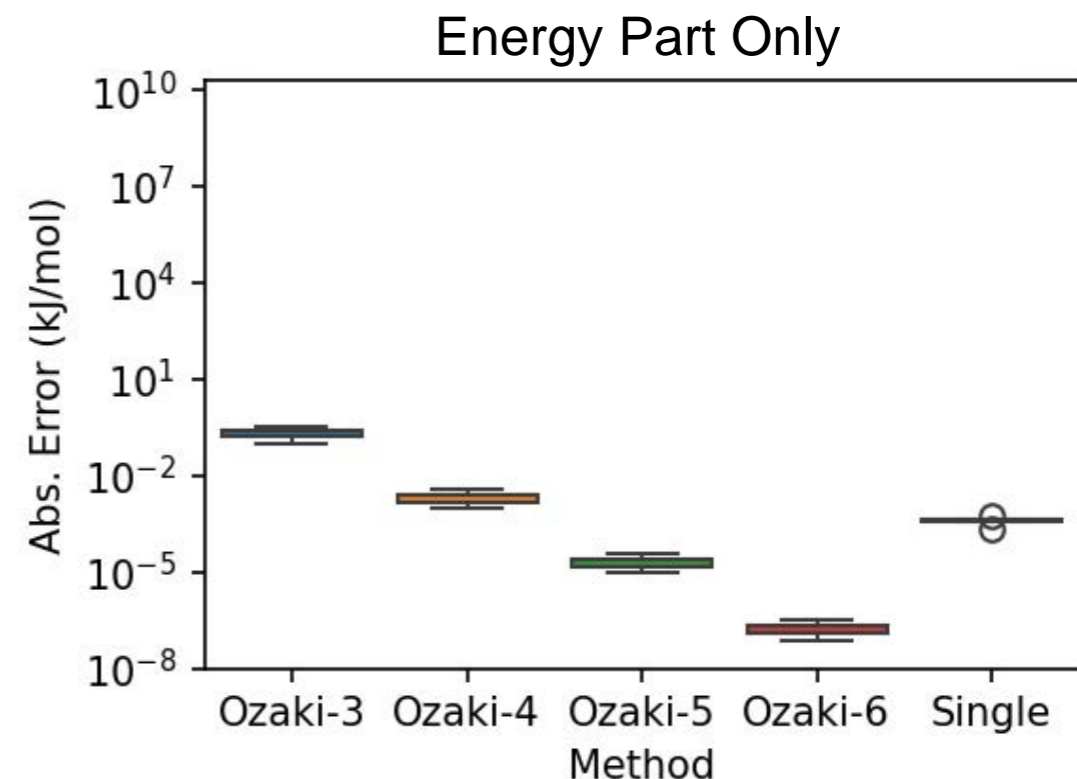
How Can I Use the Ozaki-scheme in my Code?

- If the performance of matrix-multiplication is important to you, and you have access to an NVIDIA GPU, you can try out the Ozaki-scheme very easily (and we can help).
 - <https://github.com/jdomke/ozIMMU>
 - This is for Ozaki scheme v1 targeting integer cores.
 - A similar repository has been made for Ozaki scheme v2.
- This code takes all CPU DGEMM calls and replaces them with the Ozaki-scheme.
 - So far no trouble running it with Fortran, C++, Python, Julia, etc.
- Compile the code with an NVIDIA compiler.
- Run your code intercepting calls to BLAS.

```
OZIEXTRA=( "-x" "LD_PRELOAD=${OZIMMU_LIB}" ) ← .so File that was compiled
OZIEXTRA+=( "-x" "OZIMMU_COMPUTE_MODE=fp64_int8_4" ) ← Number of Splits
OZIEXTRA+=( "-x" "OZIMMU_INFO=1" )
OZIEXTRA+=( "-x" "OZIMMU_ERROR=1" )
OZIEXTRA+=( "-x" "OZIMMU_ENABLE_CULIP_PROFILING=1" )
OZIEXTRA+=( "-x" "OZIMMU_INTERCEPT_THRESHOLD_M=0" ) ← Minimum Matrix Size
OZIEXTRA+=( "-x" "OZIMMU_INTERCEPT_THRESHOLD_N=0" )
OZIEXTRA+=( "-x" "OZIMMU_INTERCEPT_THRESHOLD_K=0" )
mpirun ${MPIEXTRA[@]} ${OZIEXTRA[@]} -np 1 ./myapp.exe
```

Examples of Automatic Profiling - 1

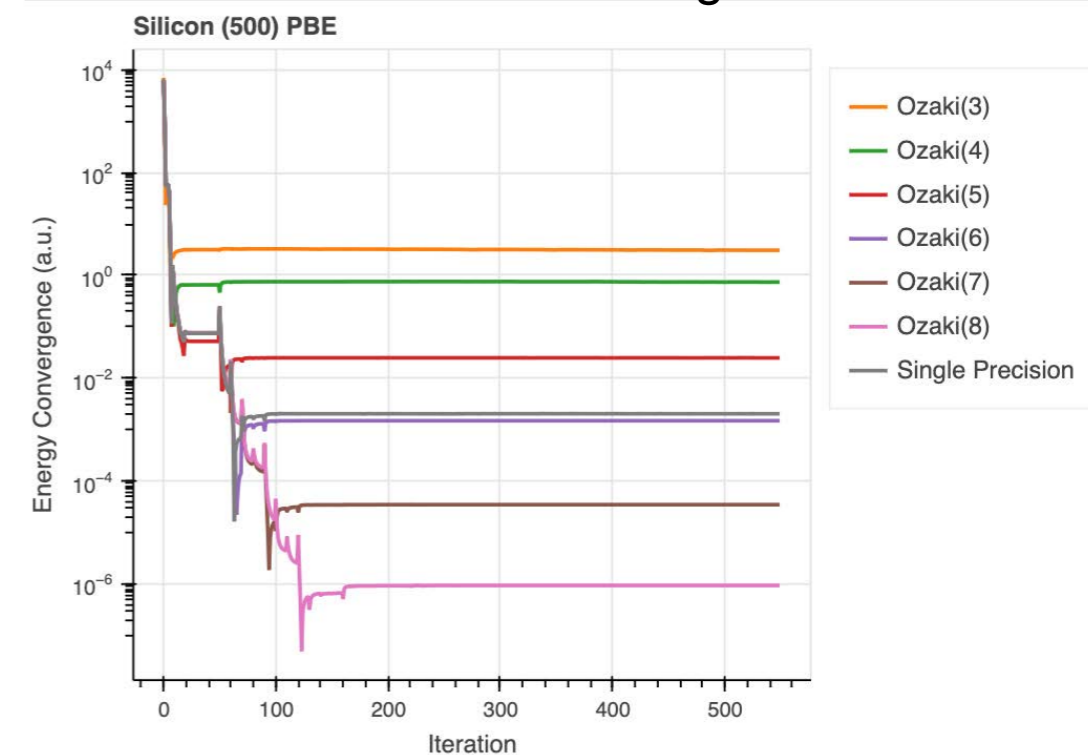
- RI-MP2 capability in the NTChem code (Fortran).
 - Relies heavily on local calls to DGEMM.
- Compute interaction energy between molecules in a dataset (L7 / cc-pVTZ).
- The algorithm can be split into two parts: fitting and computing the energy (both are very expensive).
 - The energy part is often said to be doable in single precision.
 - We find that 4 or 5 splits should be sufficient.
- For the remaining part, we hold the energy splits to 4 and compute the error.
 - 4 splits for the energy and 5 for the other parts is sufficient.



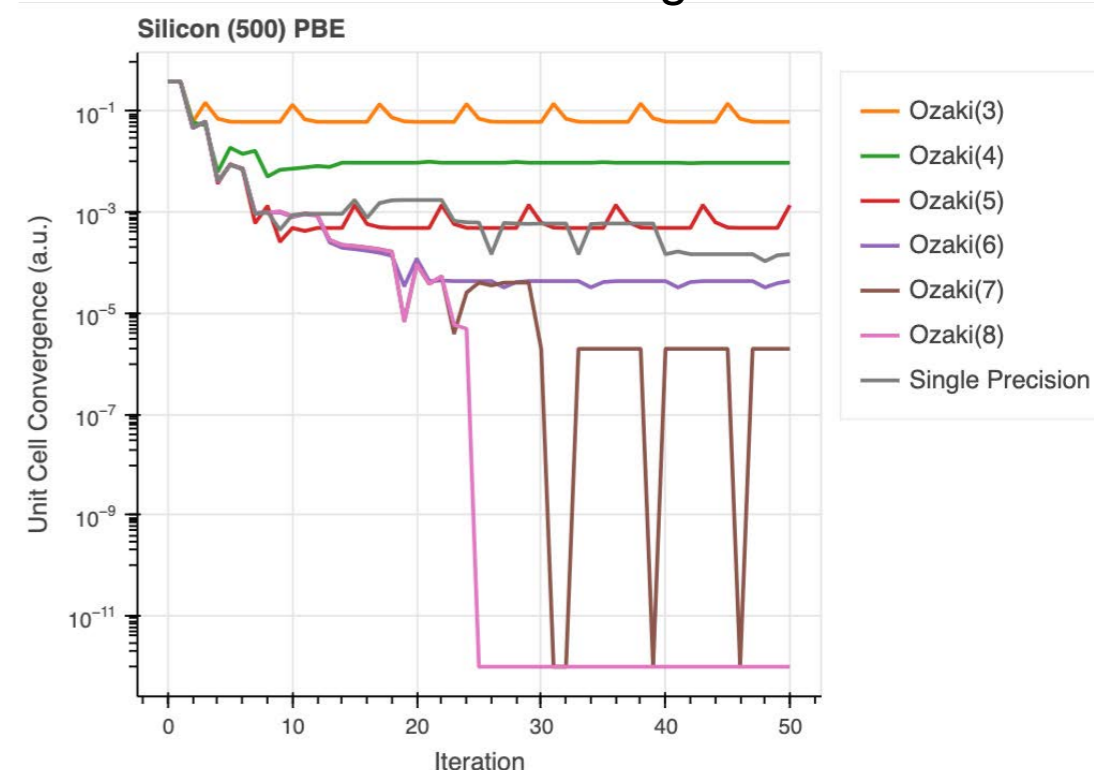
Examples of Automatic Profiling - 2

- Qbox Planewave Pseudopotential Density Functional Theory Code (C++).
- Scalapack PDGEMM call can be the bottleneck (orthogonalization step).
- Test optimizing the unit cell of bulk silicon.
 - A double loop: converge electronic structure -> compute stress tensor -> update cell size, repeat.
- Probably something like 6 splits should be sufficient for this case, with a final energy calculation with 7 or 8 splits.

Intermediate Energies

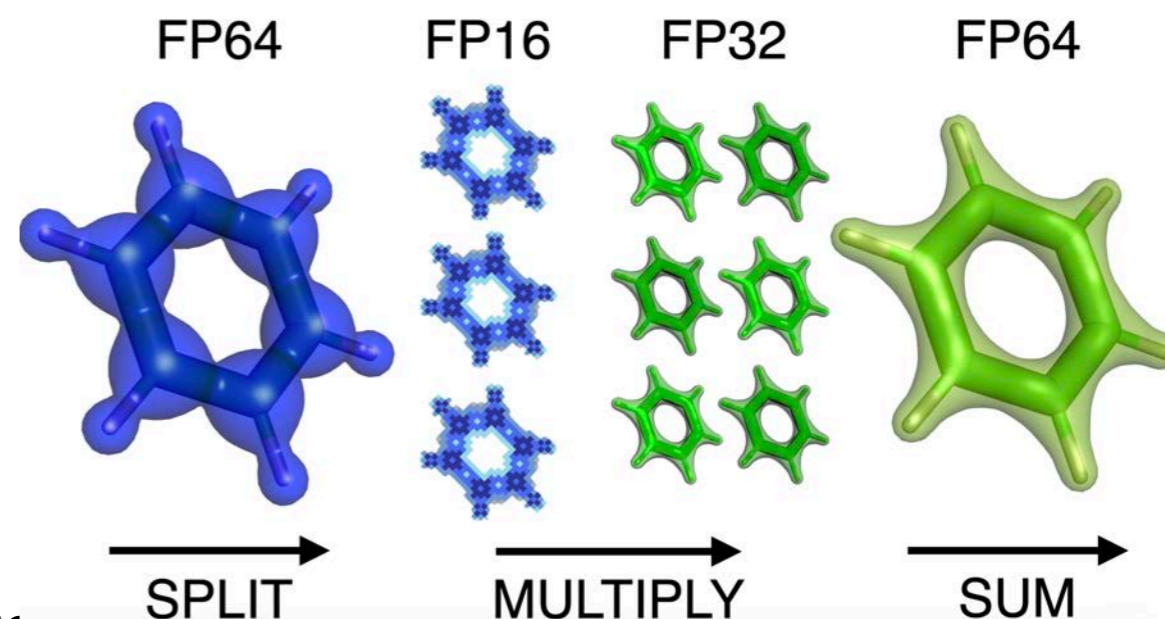


Unit Cell Convergence



Summary and Outlook

- We picked a representative computational bottleneck of materials / chemistry calculations, and solved it using a matrix multiplication dominated method.
 - Using MPFR to explore precision requirements, we find single precision is not enough, but double precision is overkill.
 - The Ozaki scheme can target this intermediate region while realizing computational savings.
- Using tensor cores teaches us to think in terms of precision requirements in terms of precision going in, and precision going out.
 - Going to low precision is a great way to reduce memory bandwidth.
- If you want your app to run on FugakuNEXT, get in touch with me (or Inoue-san).
 - It will require work on your part, but we can get your code into the benchmark set, get you access to testbeds, get your advice about design choices, etc.



WD, Katsuhisa Ozaki, Jens Domke, and Takahito Nakajima. "Reducing numerical precision requirements in quantum chemistry calculations." *Journal of Chemical Theory and Computation* (2024).

Acknowledgments: RIKEN-CEA Collaboration, TRIP-AGIS project, R-CCS Low Precision Working Group, Luigi Genovese (CEA Grenoble), Research Center for Computational Science Projects (23-IMS-C029, 24-IMS-C151)

<https://bigdft.org/>

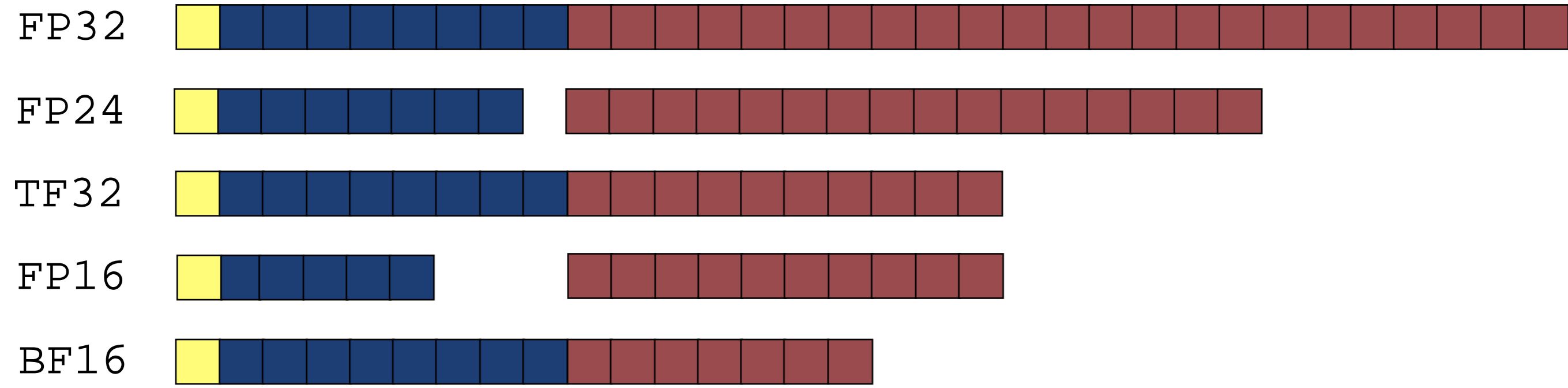
https://www.r-ccs.riken.jp/software_center/software/ntchem/overview/

<https://william-dawson.github.io/NTPoly/>

Sign

Exponent

Mantissa



$$A = A_1 + A_2 + A_3 + A_4$$

$$B = B_1 + B_2 + B_3 + B_4$$

$$\sum \begin{array}{|c|c|c|c|} \hline A_1B_1 & A_1B_2 & A_1B_3 & A_1B_4 \\ \hline A_2B_1 & A_2B_2 & A_2 \times (B_3 + B_4) & \\ \hline A_3B_1 & A_4 \times (B_2 + B_3 + B_4) & & \\ \hline A_4B_1 & & & \\ \hline \end{array} = AB$$