Lossy Compression and Mixed Precision Strategies for Memory-Bound Linear Algebra

PPAM 2022
Gdansk, Poland

Hartwig Anzt
Innovative Computing Lab, University of Tennessee

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, the Horizon2020 Program of the European Commission, and the Helmholtz Impuls und VernetzungsfondVH-NG-1241.

Running iterative methods in different precision formats

Linear System Ax=b with cond(A) ≈ 10^7

(apache2 from SuiteSparse) **NVIDIA V100 GPU**

```
Double precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 9.6639e-09
GMRES iteration count: 23271
GMRES execution time: 43801 ms
```

Relative residual ~10^{-12}

Running iterative methods in different precision formats

Linear System Ax=b with cond(A) ≈ 10^7

(apache2 from SuiteSparse) **NVIDIA V100 GPU**

```
Double precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 9.6639e-09
GMRES iteration count: 23271
GMRES execution time: 43801 ms
```

Relative residual ~10^{-12}

```
Single precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 0.00175464
GMRES iteration count: 25000
GMRES execution time: 27376 ms
```

Relative residual ~10^{-7}

Running iterative methods in different precision formats

Linear System Ax=b with cond(A) ≈ 10^7

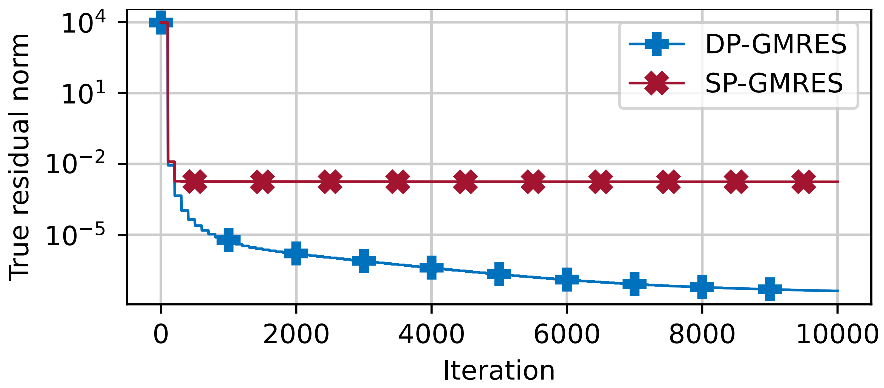
(apache2 from SuiteSparse) **NVIDIA V100 GPU**

```
Double precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 9.6639e-09
GMRES iteration count: 23271
GMRES execution time: 43801 ms
```

Relative residual ~10^{-12}

```
Single precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 0.00175464
GMRES iteration count: 25000
GMRES execution time: 27376 ms
```

Relative residual ~10^{-7}



Running iterative methods in different precision formats

Linear System Ax=b with cond(A) ≈ 10^7

(apache2 from SuiteSparse) **NVIDIA V100 GPU**

```
Double precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 9.6639e-09
GMRES iteration count: 23271
GMRES execution time: 43801 ms
```

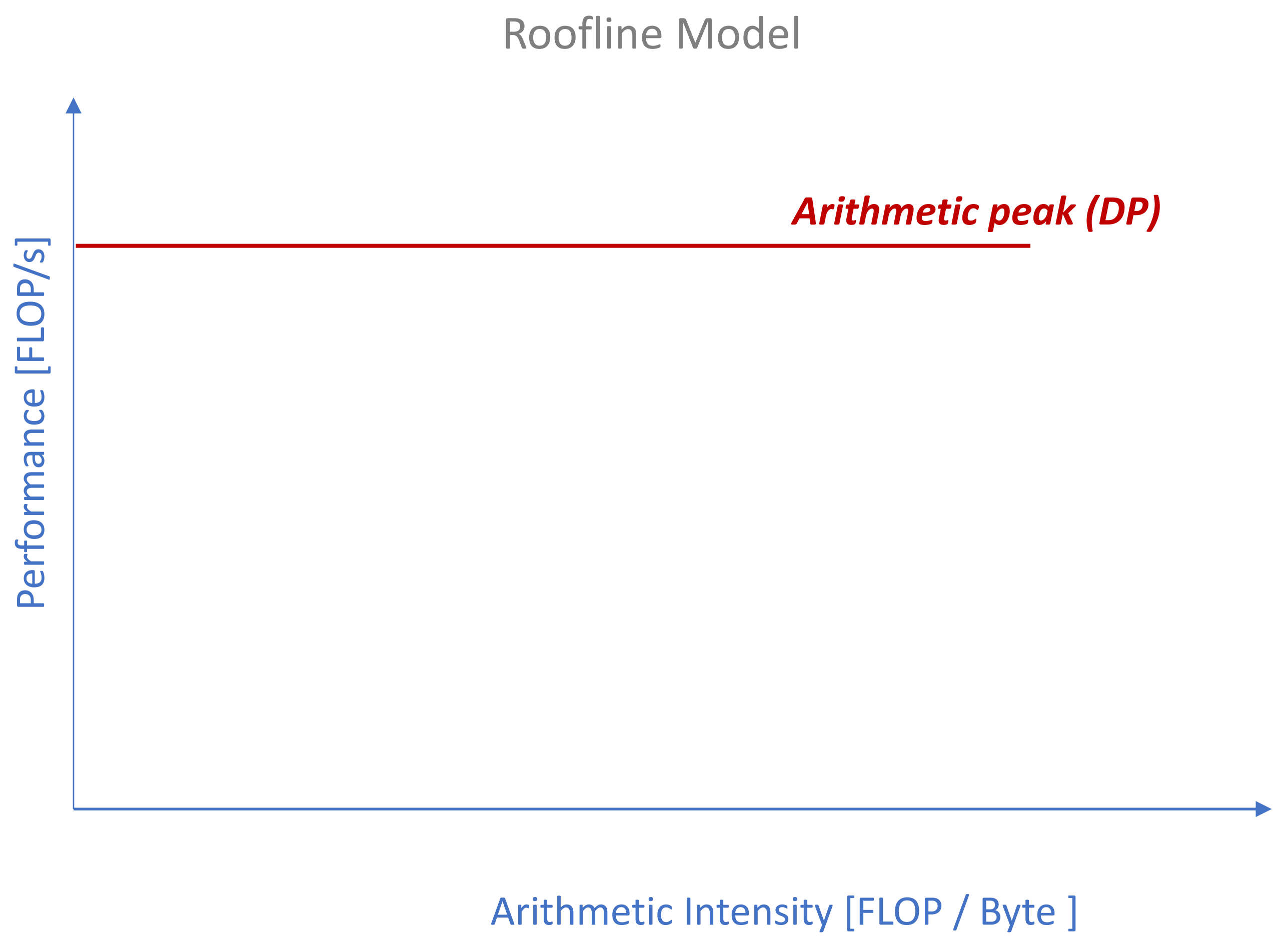
Relative residual ~10^{-12}

```
Single precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 0.00175464
GMRES iteration count: 25000
GMRES execution time: 27376 ms
```

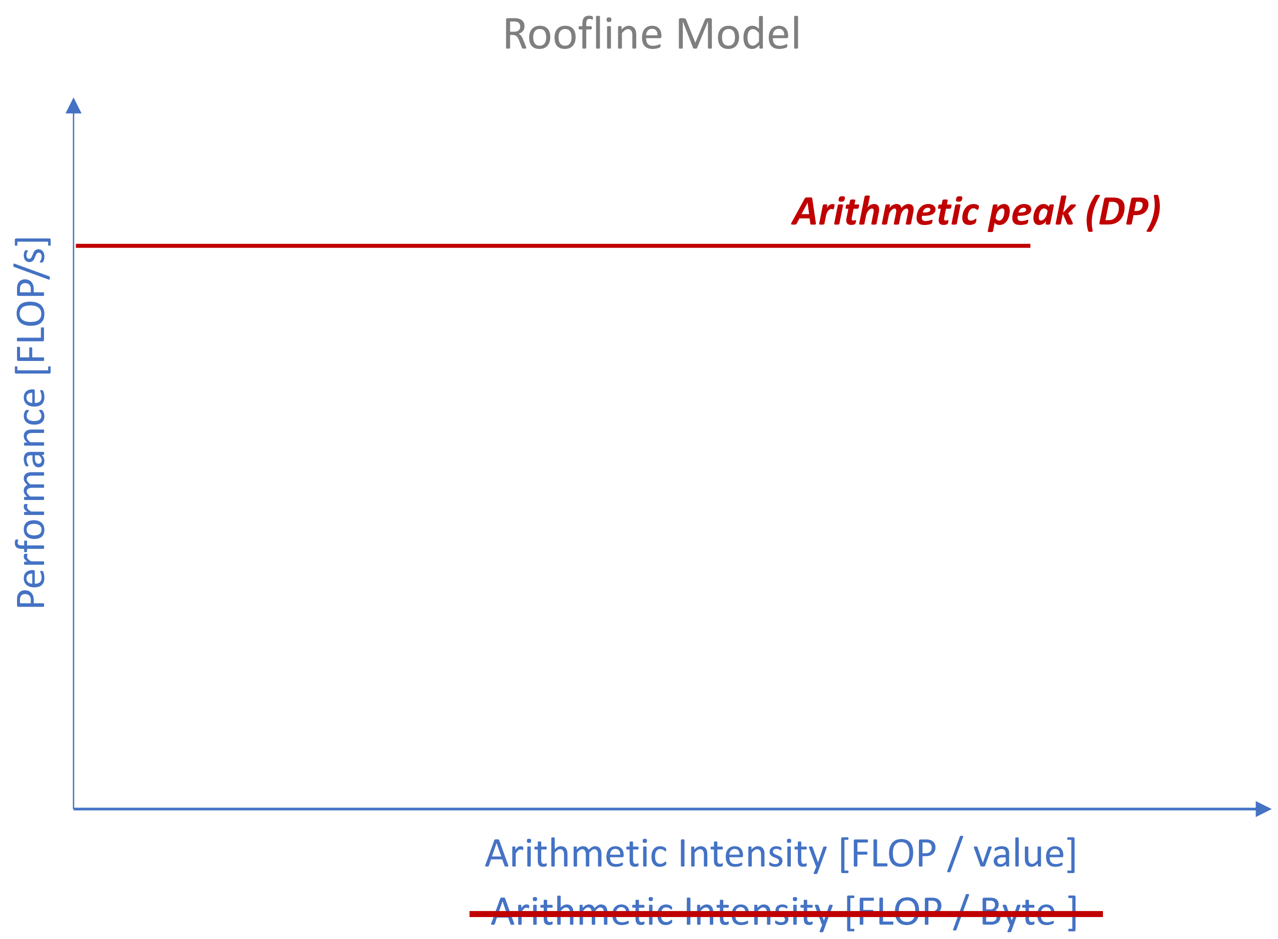
Relative residual ~10^{-7}

~2x faster!

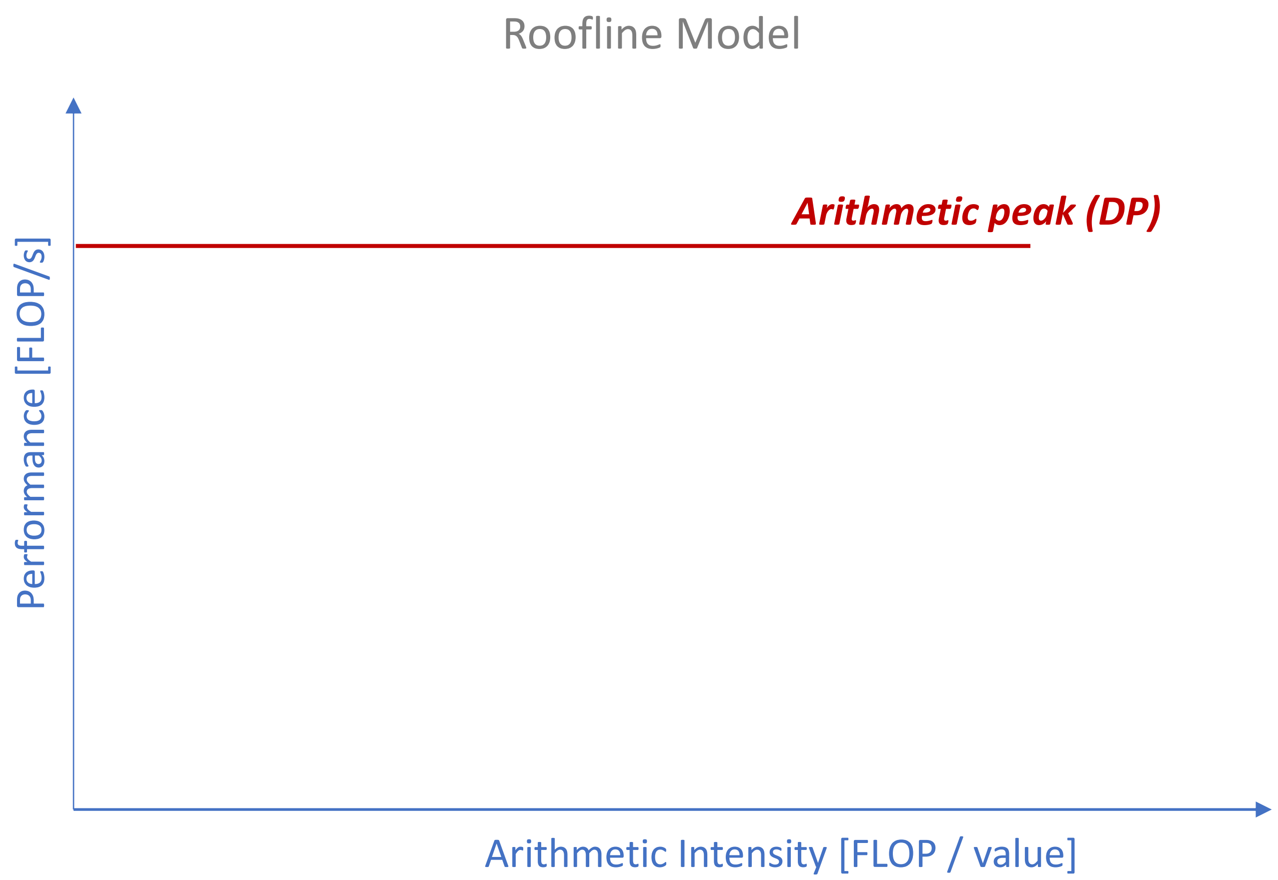
Why are we faster with a single precision algorithm?



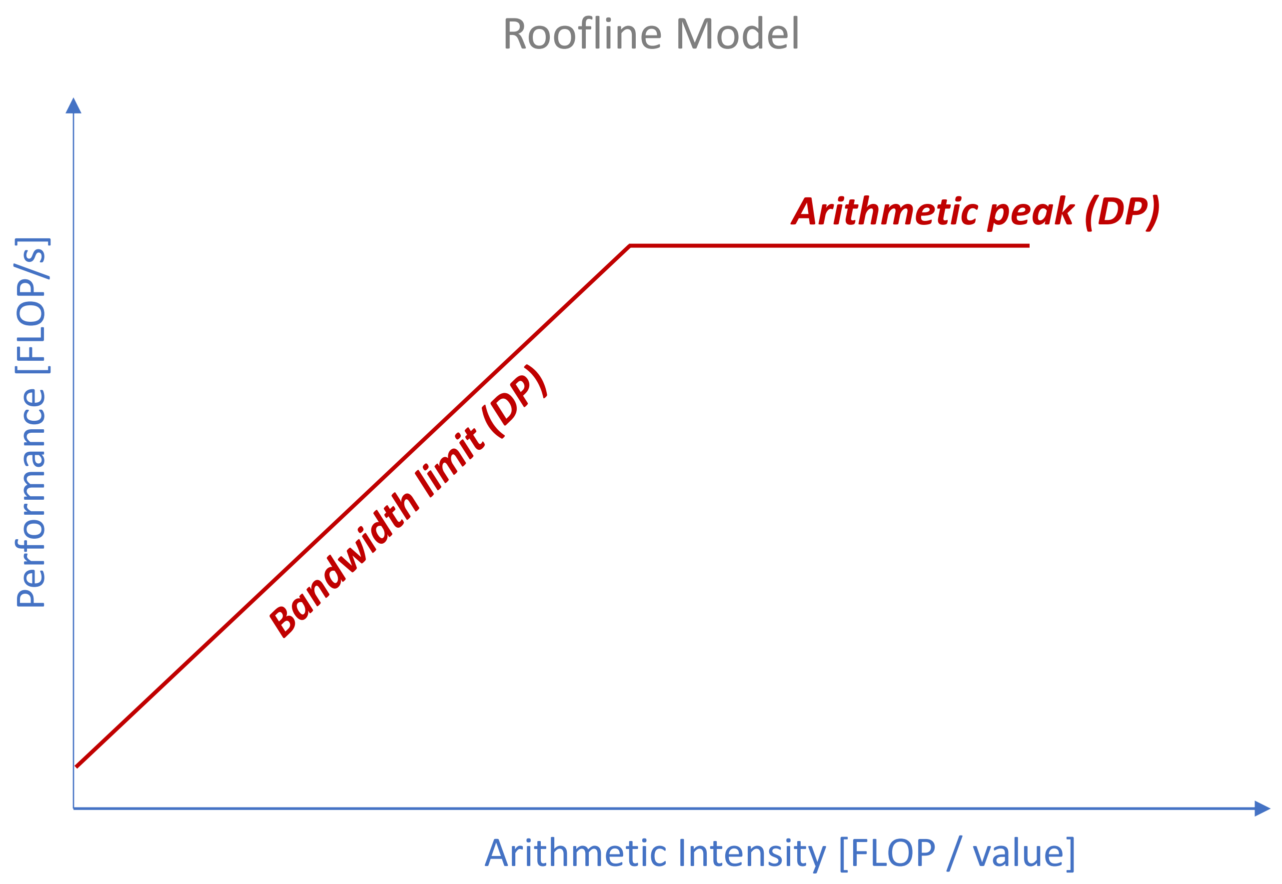
Why are we faster with a single precision algorithm?



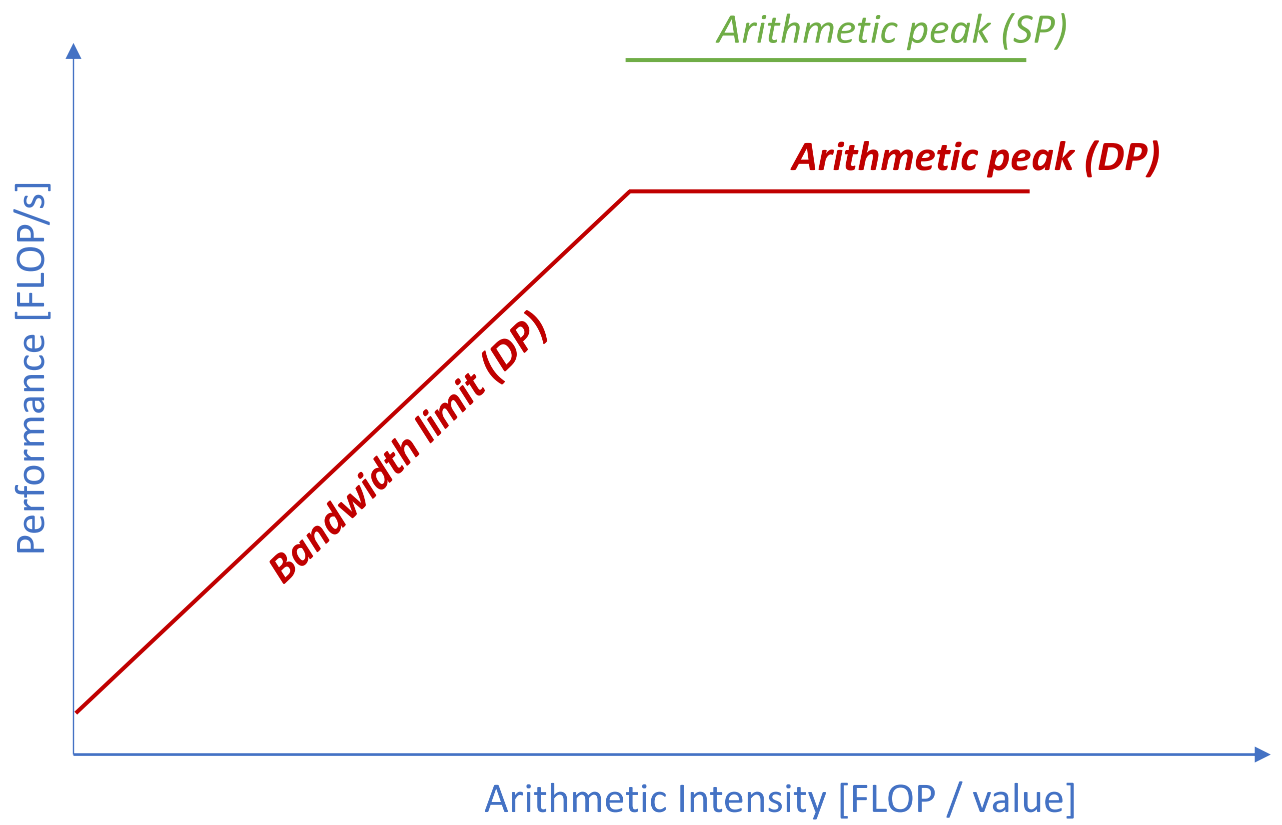
Why are we faster with a single precision algorithm?



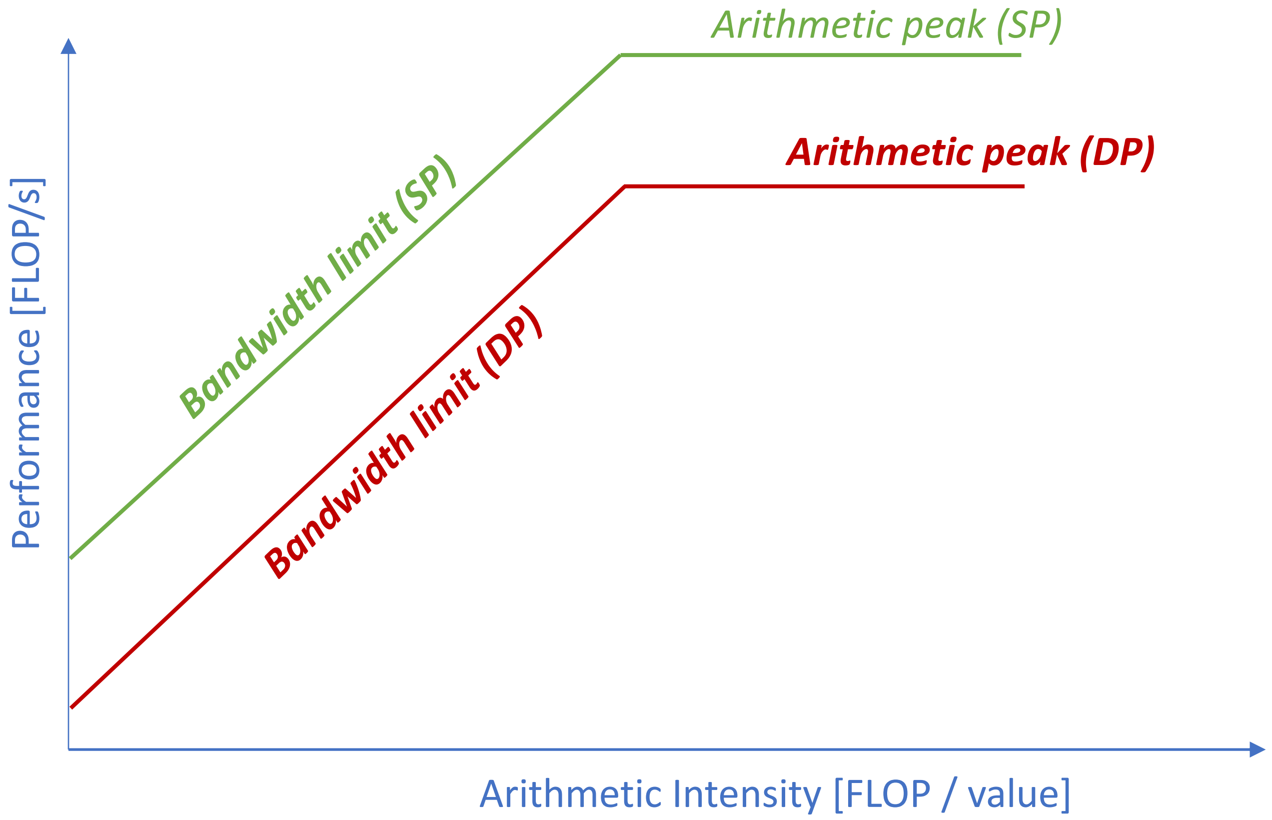
Why are we faster with a single precision algorithm?



Why are we faster with a single precision algorithm?



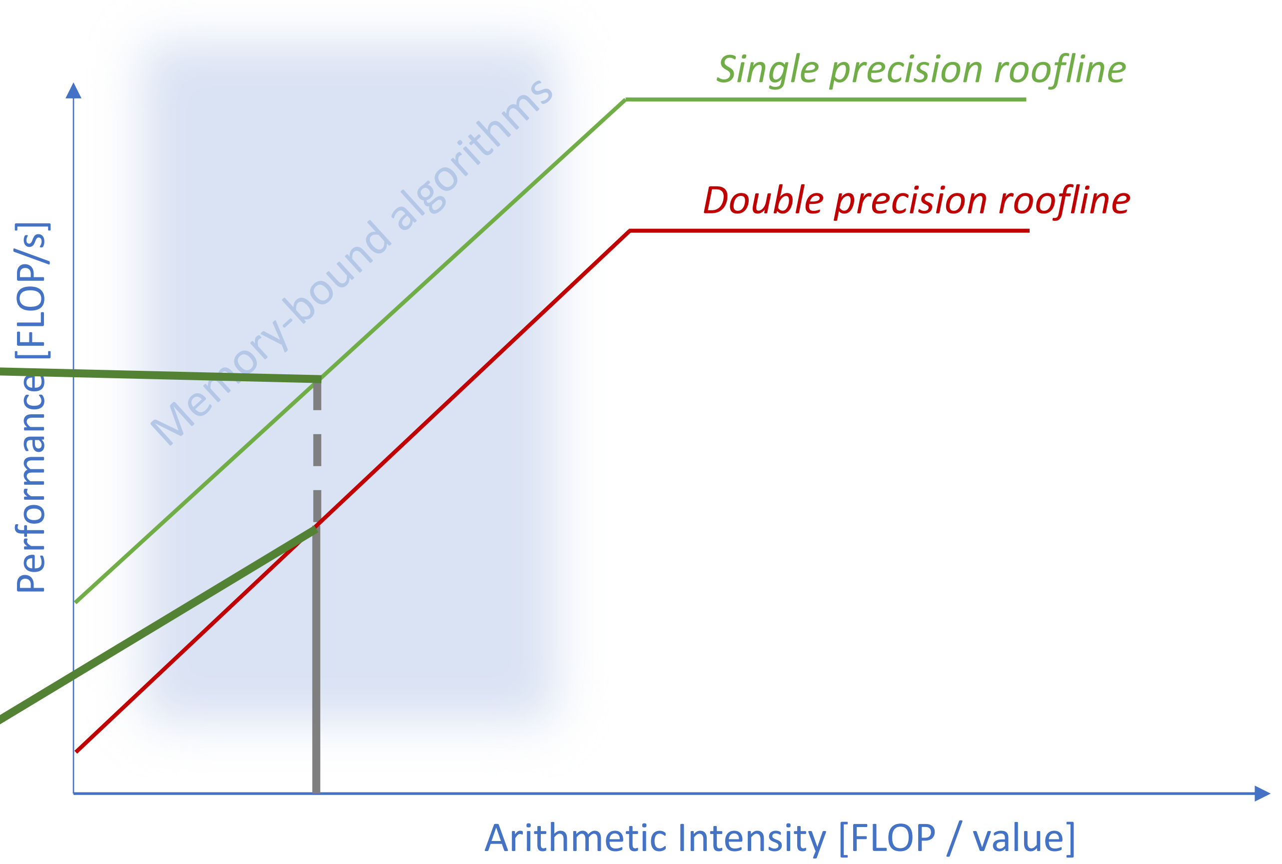
Why are we faster with a single precision algorithm?



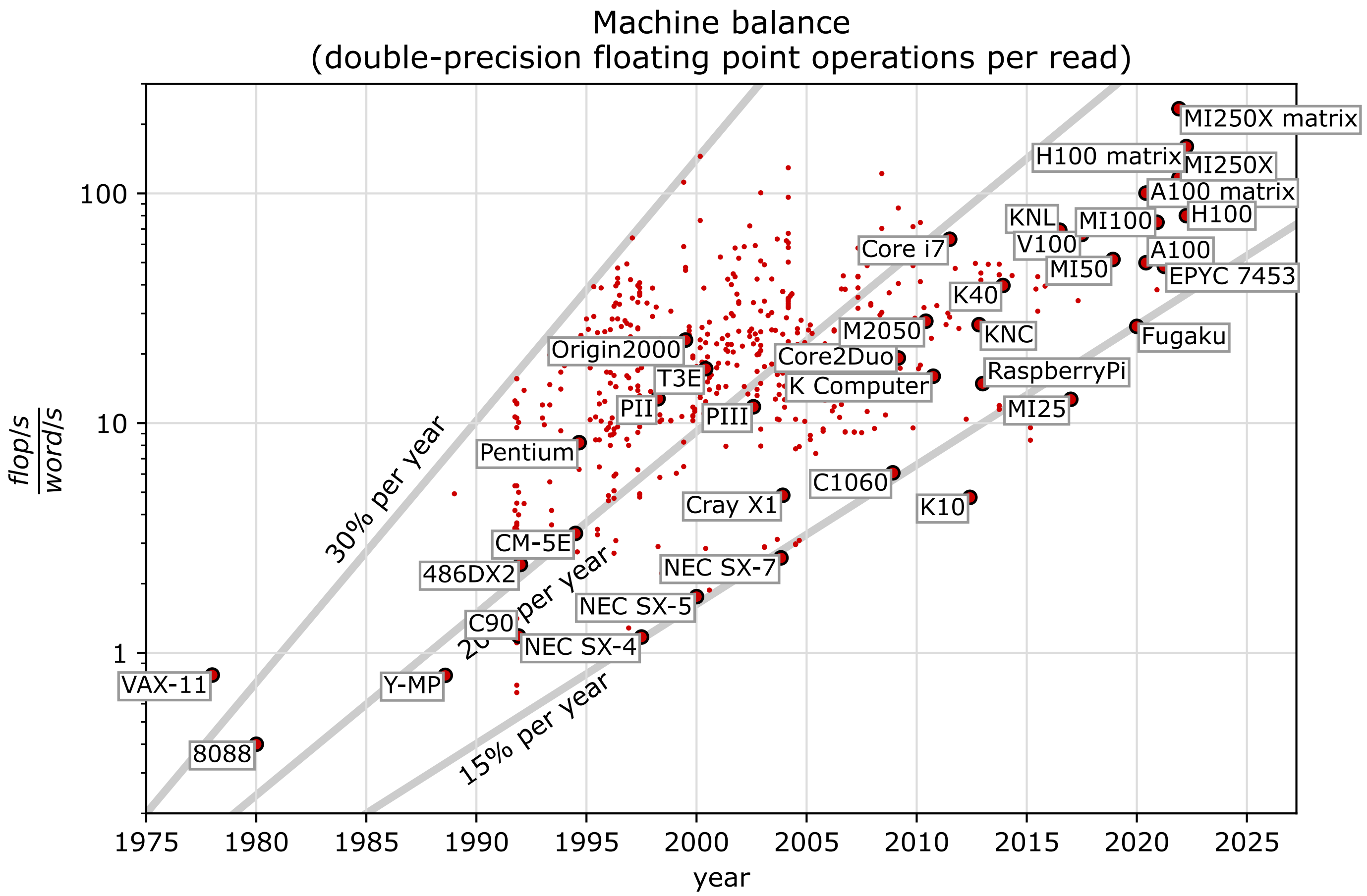
Why are we faster with a single precision algorithm?

```
Single precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 0.00175464
GMRES iteration count: 25000
GMRES execution time: 27376 ms
```

```
Double precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 9.6639e-09
GMRES iteration count: 23271
GMRES execution time: 43801 ms
```



The memory wall



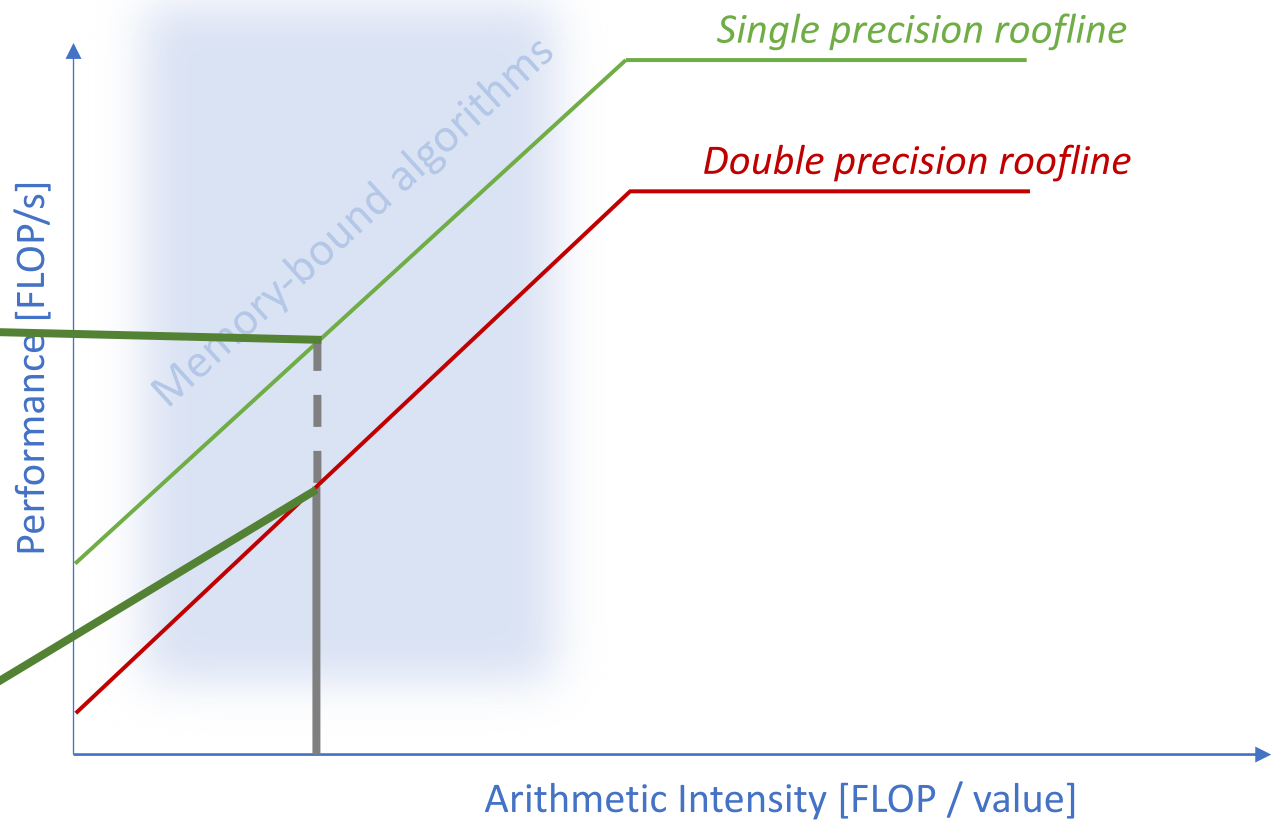
M. Gates

Compute performance grows faster than memory bandwidth.

Why are we faster with a single precision algorithm?

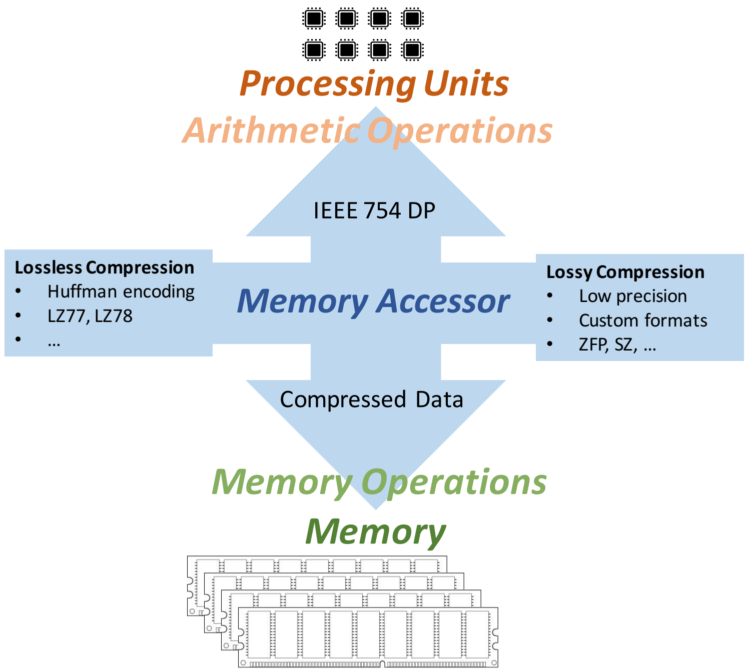
```
Single precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 0.00175464
GMRES iteration count: 25000
GMRES execution time: 27376 ms
```

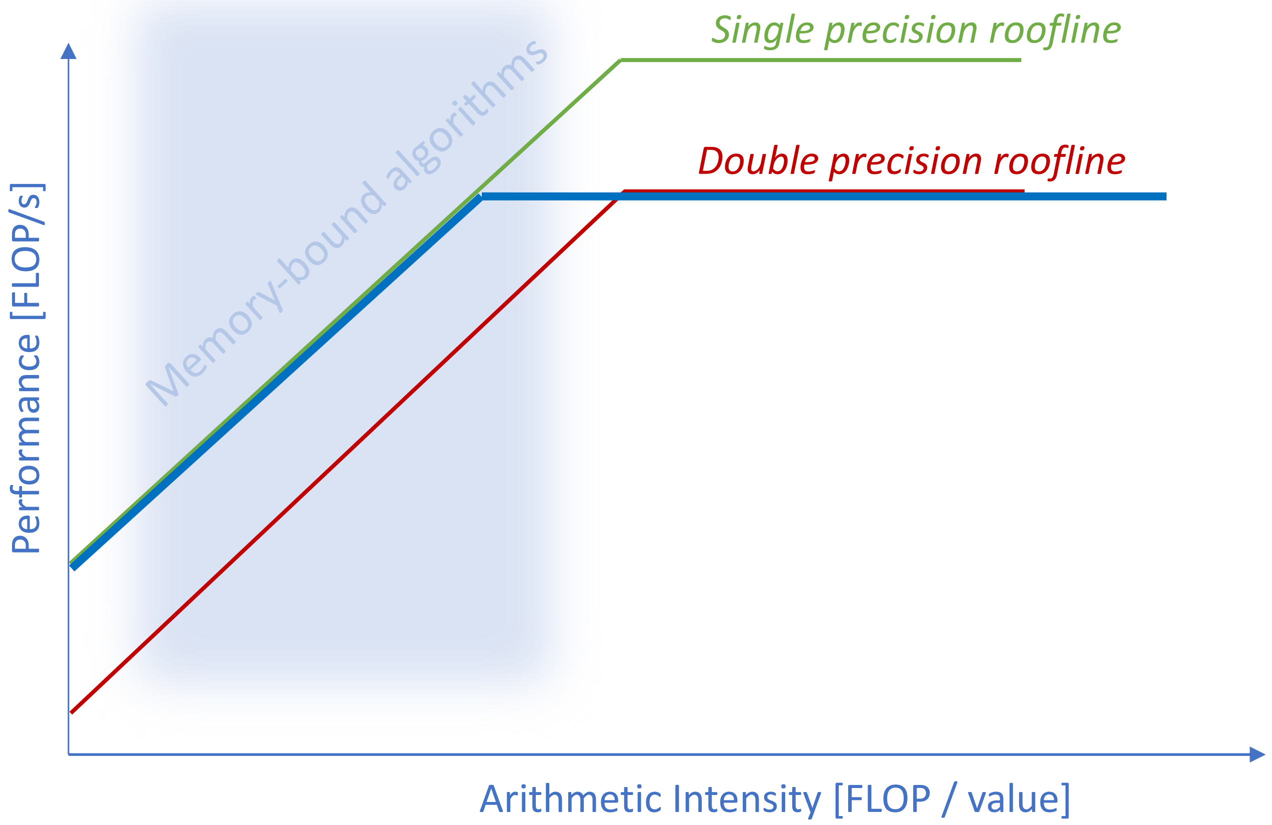
```
Double precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 9.6639e-09
GMRES iteration count: 23271
GMRES execution time: 43801 ms
```



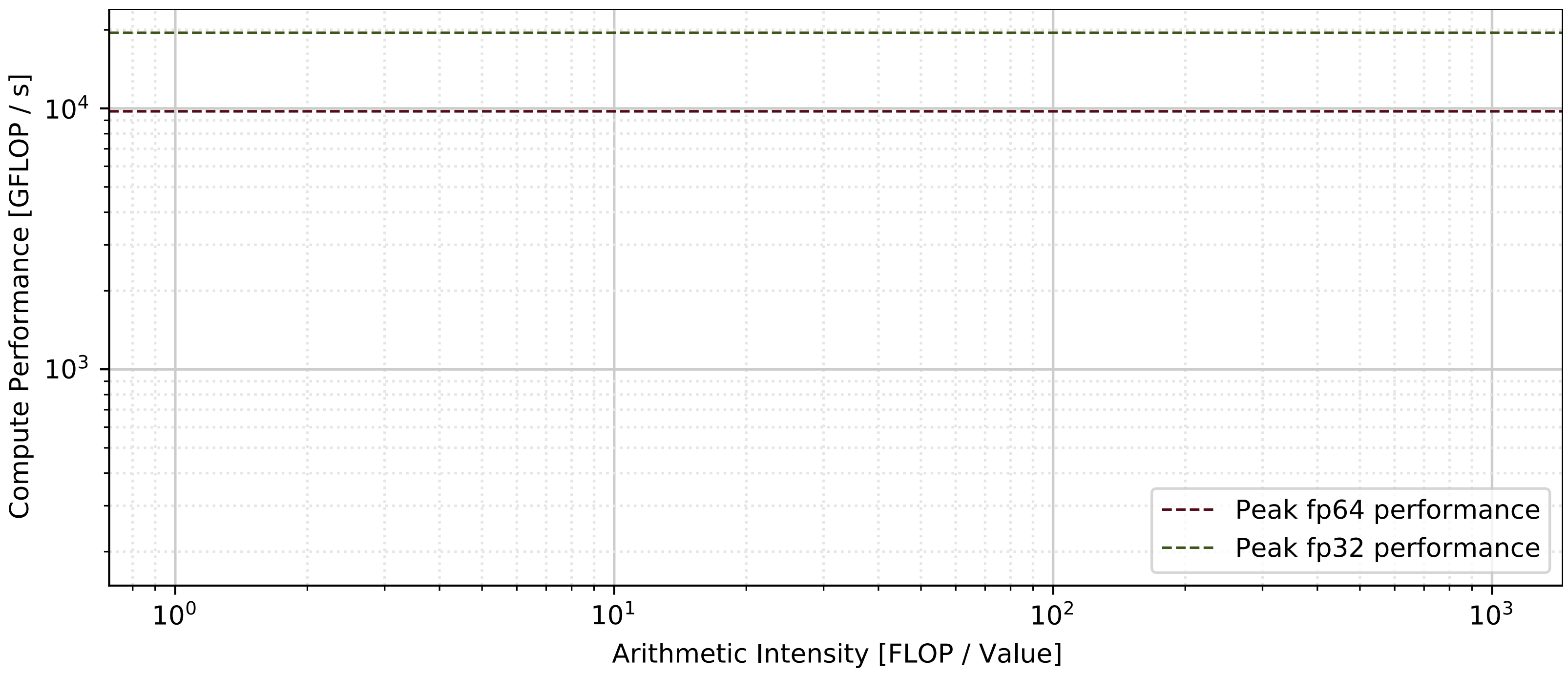
Can we get the best of both worlds?

- For memory-bound algorithms, the arithmetic operations are free, can use high precision formats.
- Data access should be as cheap as possible, use reduced precision.
- In-Register Compression

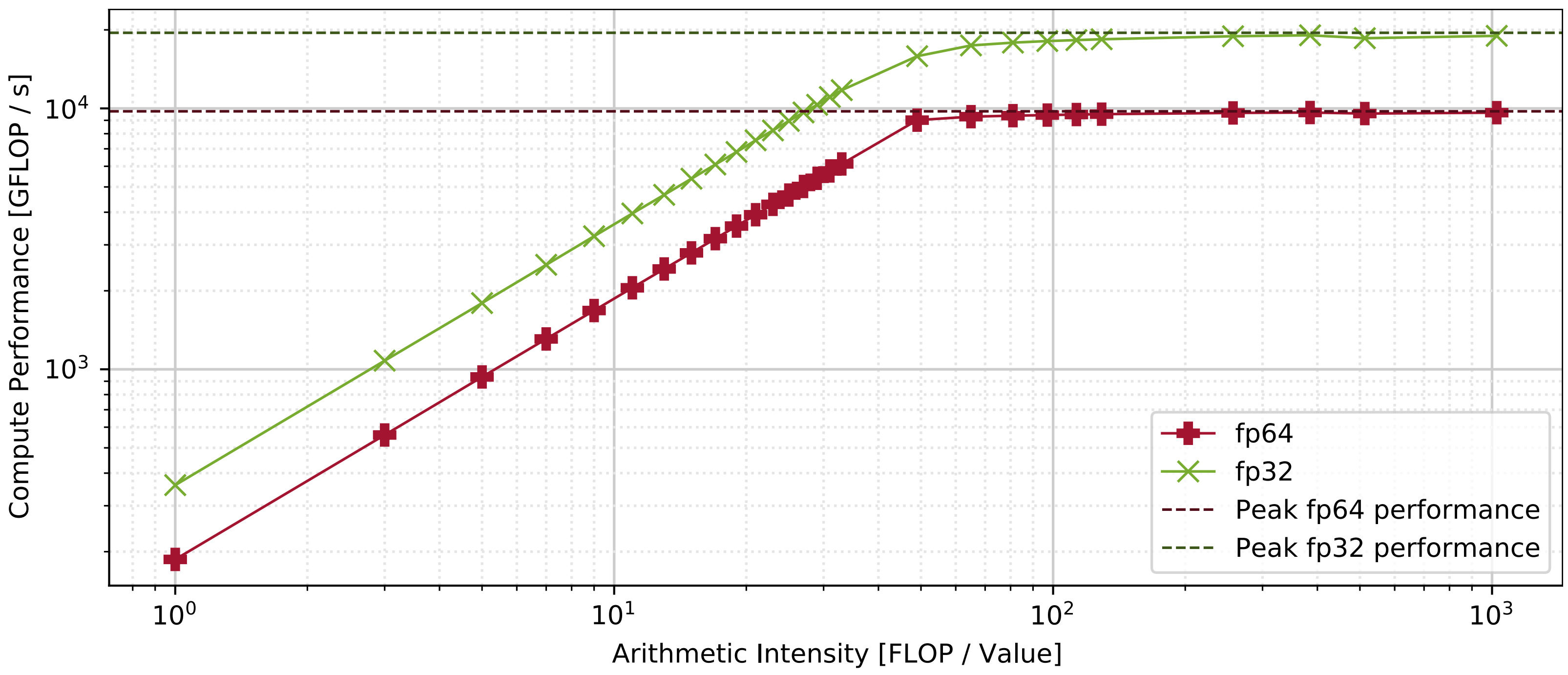




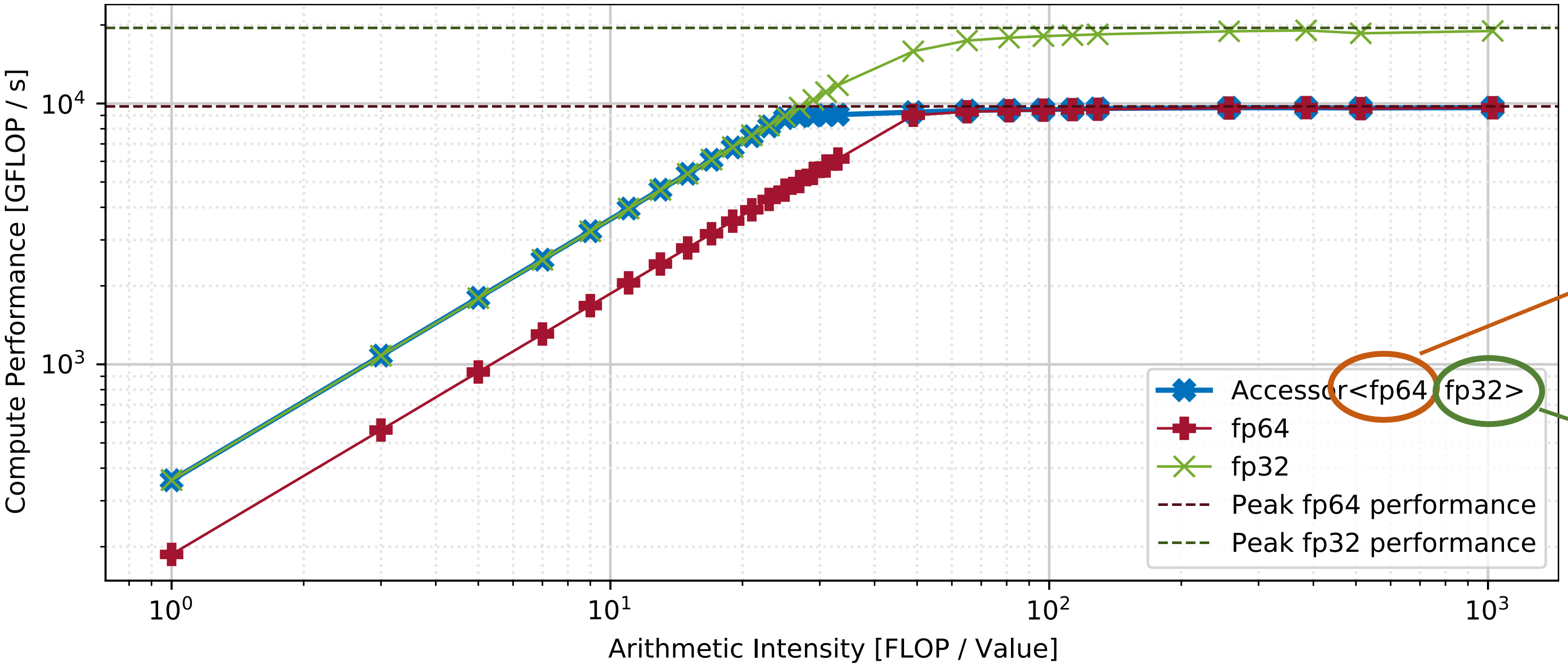
Memory Accessor for NVIDIA A100 GPU

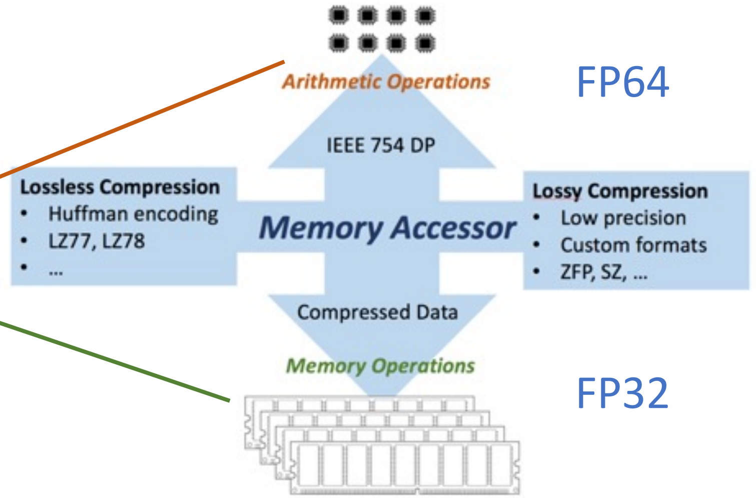


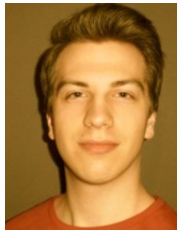
Memory Accessor for NVIDIA A100 GPU



Memory Accessor for NVIDIA A100 GPU







T. Grützmacher

Use the memory accessor to boost performance

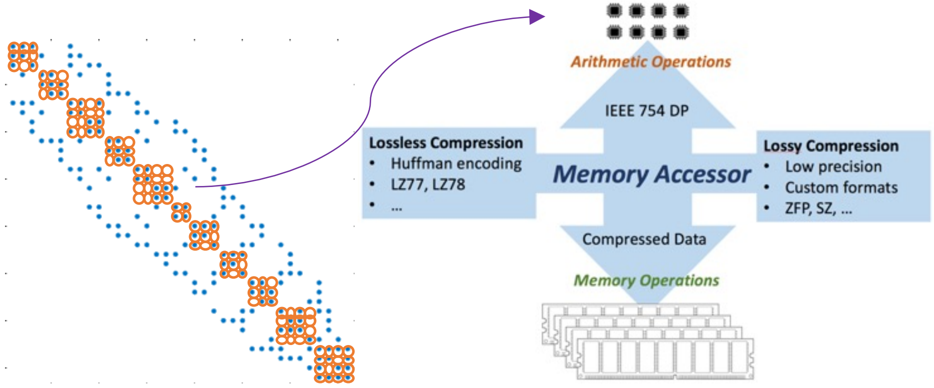
- *Start from double precision algorithm*
- *Use memory accessor to store intermediate data in compressed form*
- *Require double precision output accuracy*

Use the memory accessor to boost performance

Goran Flegar

E.S. Quintana-Orti

- *Start from double precision algorithm*
- *Use memory accessor to store intermediate data in compressed form*
- *Require double precision output accuracy*



- *Preconditioning*

[1] G Flegar, H Anzt, T Cojean, ES Quintana-Ortí, "Adaptive precision block-Jacobi for high performance preconditioning in the Ginkgo linear algebra software," ACM Transactions on Mathematical Software (TOMS) 47 (2), 1-28.

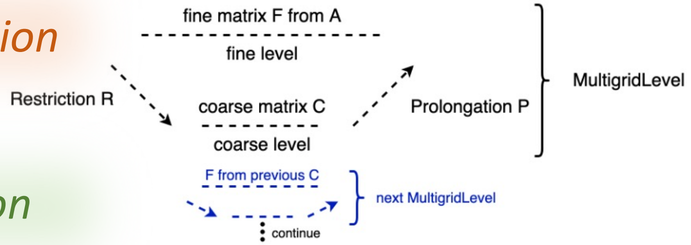
[2] F Göbel, T Grützmacher, T Ribizel, H Anzt, "Mixed precision incomplete and factorized sparse approximate inverse preconditioning on GPUs," European Conference on Parallel Processing, 550-564.

- *Multigrid*

[3] Stephen F. McCormick, Joseph Benzaken, Rasmus Tamstorf "Algebraic error analysis for mixed-precision multigrid solvers", https://arxiv.org/abs/2007.06614
[4] M Tsai, N Beams, H Anzt, "Mixed precision algebraic multigrid on GPUs," PPAM 2022.

high precision

low precision



- *Krylov solvers*

[5] J Aliaga, H Anzt, T Grützmacher, E S Quintana-Ortí, A Tomás, "Compressed basis GMRES on high-performance graphics processing units", IJHPCA 2022

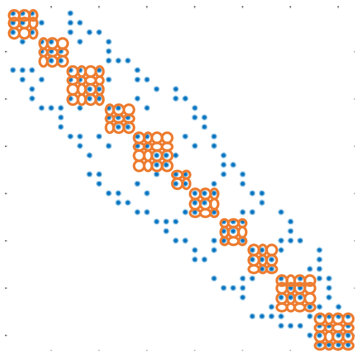
Mike Tsai

Using the memory accessor for mixed precision preconditioning

- **Preconditioning iterative solvers.**
  - Idea: Approximate inverse of system matrix to make the system “easier to solve”: $P^{-1} \approx A^{-1}$

    and solve $Ax = b \quad \Leftrightarrow \quad P^{-1}Ax = P^{-1}b \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}\,.$

- **Block-Jacobi preconditioner** is based on **block-diagonal scaling**: $P = diag_B(A)$
  - Each block corresponds to one (small) linear system.
    - *Larger* blocks typically improve convergence.
    - *Larger* blocks make block-Jacobi more expensive.



Using the memory accessor for mixed precision preconditioning

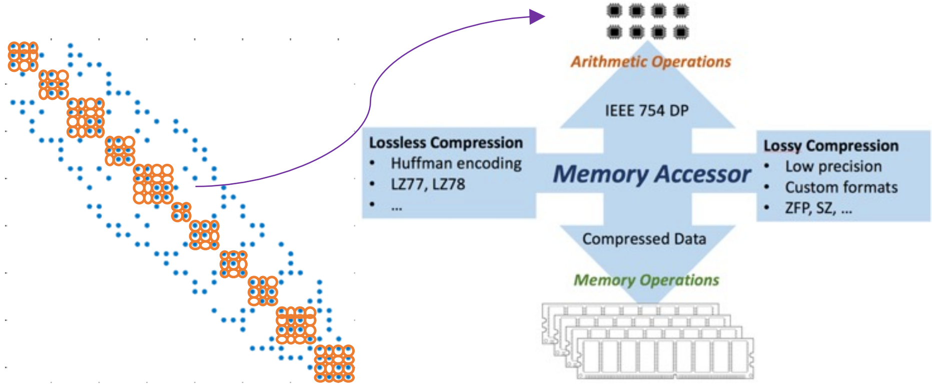
- **Preconditioning iterative solvers.**
  - Idea: Approximate inverse of system matrix to make the system "easier to solve": $P^{-1} \approx A^{-1}$

    and solve $Ax = b \quad \Leftrightarrow \quad P^{-1}Ax = P^{-1}b \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}\,.$

Goran Flegar

E.S. Quintana-Orti

- **Block-Jacobi preconditioner** is based on **block-diagonal scaling**: $P = diag_B(A)$
  - Each block corresponds to one (small) linear system.
    - *Larger* blocks typically improve convergence.
    - *Larger* blocks make block-Jacobi more expensive.

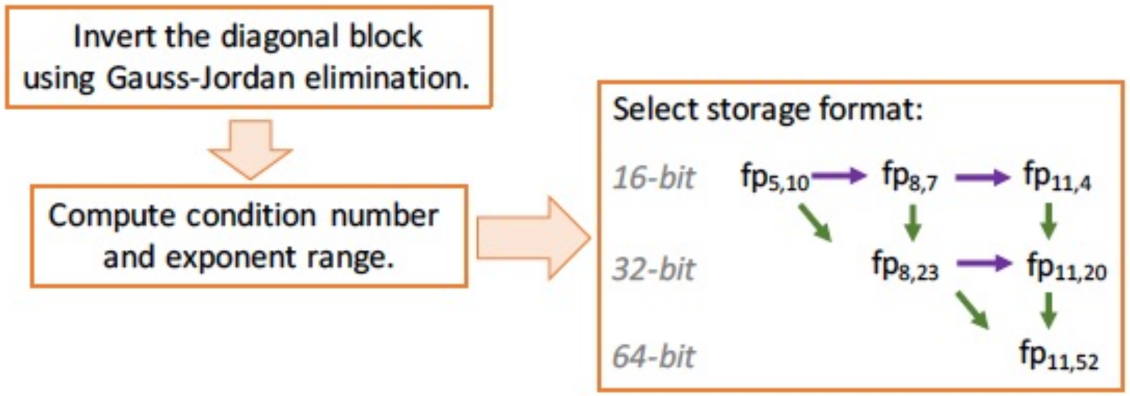


- *Why should we store the preconditioner matrix P^{-1} in full (high) precision?*

- Use the accessor to store the inverted diagonal blocks in lower precision.
  - *Be careful to preserve the regularity of each inverted diagonal block!*

Using the memory accessor for mixed precision preconditioning

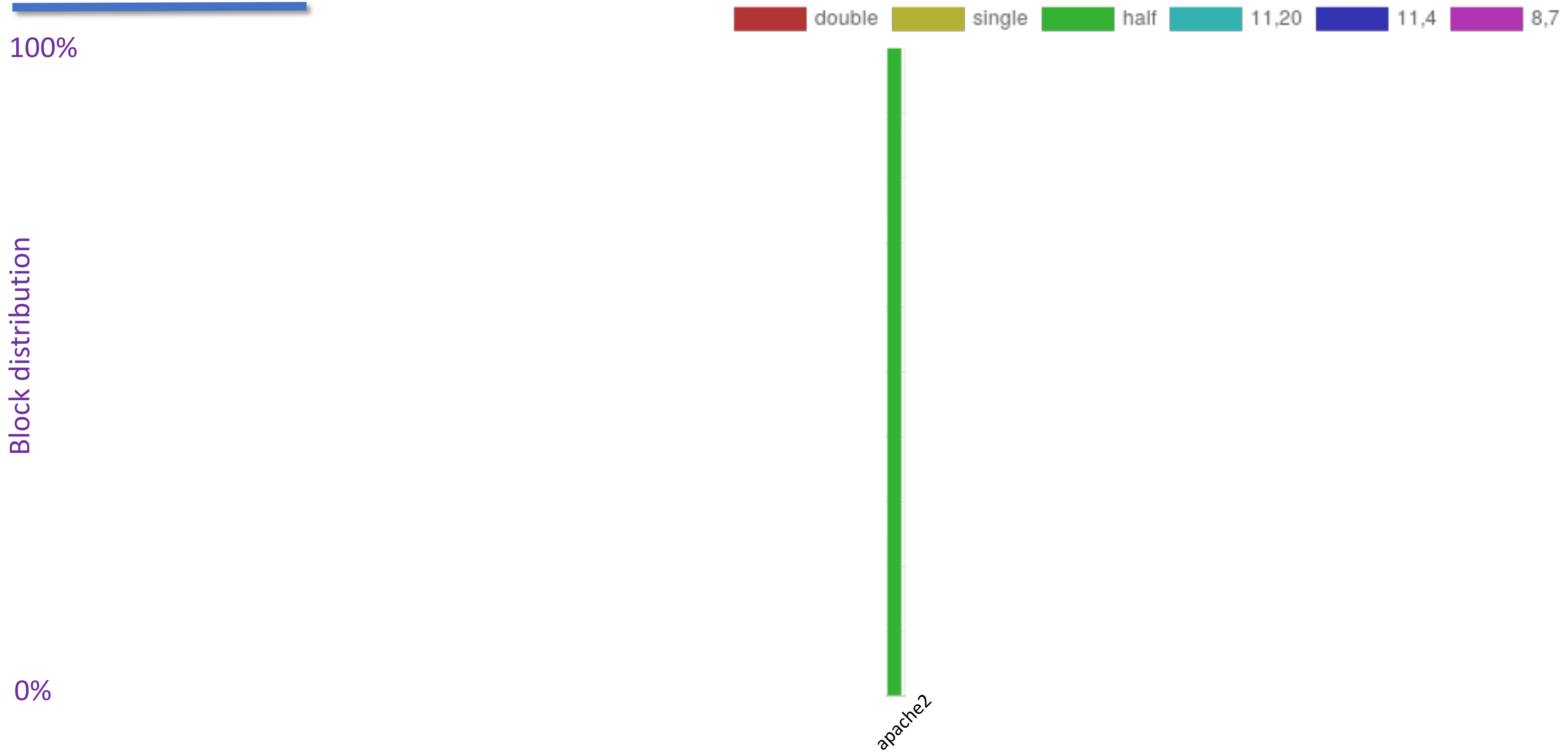
- Choose how much accuracy of the preconditioner should be preserved in the selection of the storage format.
- All computations use double precision, but store blocks in lower precision.



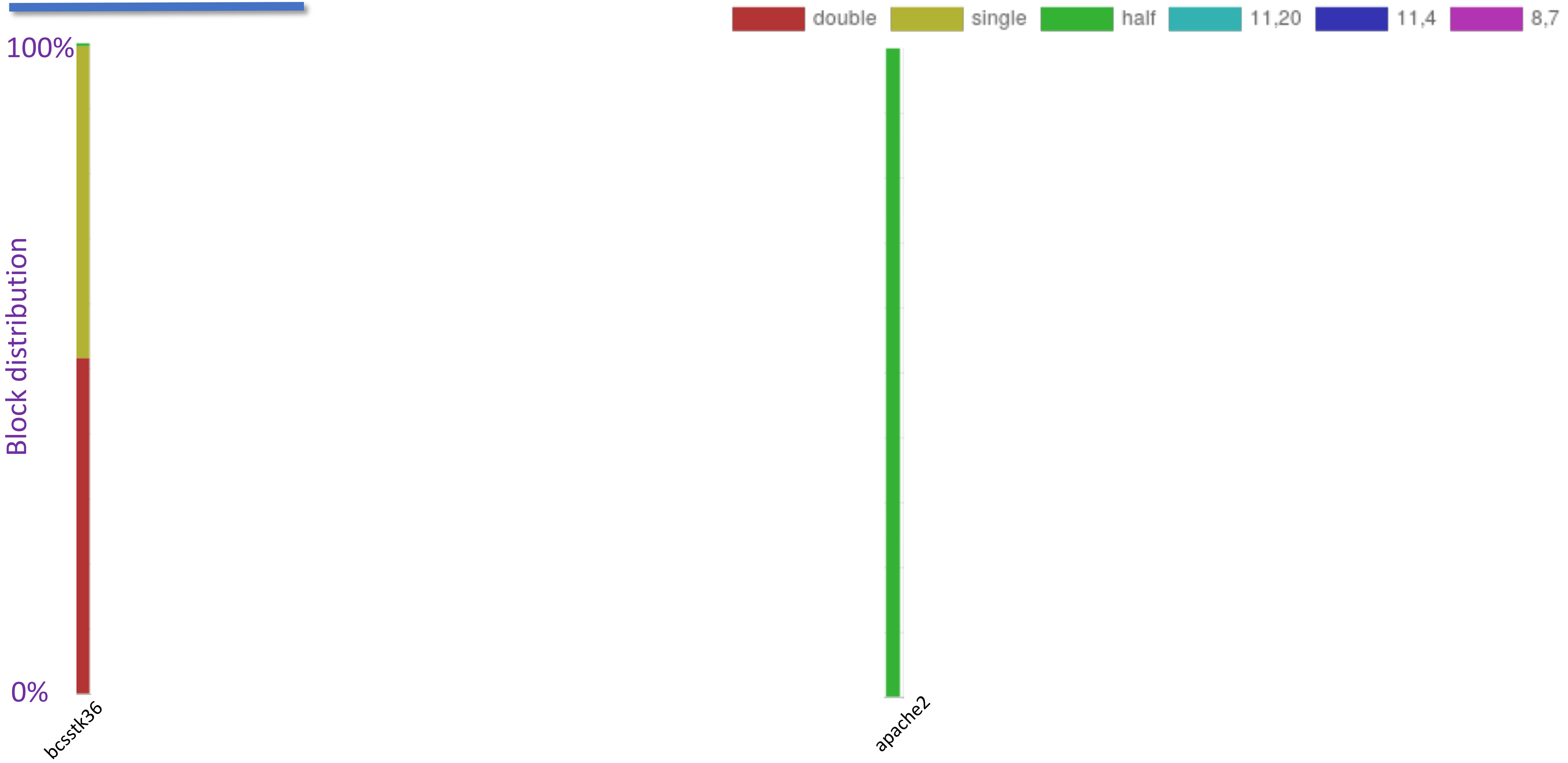
+ **Regularity preserved;**
+ Flexibility in the accuracy;
+ ”Not a low precision preconditioner”
    + Preconditioner is a constant operator;
    + No flexible Krylov solver needed ;

- **Overhead** of the **precision detection**
  (condition number calculation);
- **Overhead** from storing **precision information**
  (need to additionally store/retrieve flag);
- Speedups / preconditioner quality **problem-dependent**;

Using the memory accessor for mixed precision preconditioning



Using the memory accessor for mixed precision preconditioning



Using the memory accessor for mixed precision preconditioning

Using the memory accessor for mixed precision preconditioning

Linear System Ax=b with cond(A) ≈ 10^7 *(apache2 from SuiteSparse)* **NVIDIA A100 GPU**

Double Precision CG + Double Precision Preconditioner

```
Initial residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
1390.67
Final residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
3.97985e-06
CG iteration count:     4797
CG execution time [ms]: 2971.18
```

Accuracy improvement ~10^9

Single Precision CG + Single Precision Preconditioner

```
Initial residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
1390.67
Final residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
1588.77
CG iteration count:     8887
CG execution time [ms]: 2972.46
```

No improvement

Experiments based on the Ginkgo library https://ginkgo-project.github.io/
ginkgo/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp

Using the memory accessor for mixed precision preconditioning

Ginkgo

Linear System Ax=b with cond(A) ≈ 10^7 *(apache2 from SuiteSparse)* **NVIDIA A100 GPU**

Double Precision CG + Double Precision Preconditioner

```
Initial residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
1390.67
Final residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
3.97985e-06
CG iteration count:     4797
CG execution time [ms]: 2971.18
```

Double Precision CG + **Mixed Precision Preconditioner**

- *Preconditioner remains a constant operator*
- *Attainable accuracy of CG unaffected*
- *Faster because of less data movement*

Experiments based on the Ginkgo library https://ginkgo-project.github.io/
ginkgo/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp

Using the memory accessor for mixed precision preconditioning

Ginkgo

Linear System Ax=b with cond(A) ≈ 10^7 *(apache2 from SuiteSparse)* **NVIDIA A100 GPU**

Double Precision CG + Double Precision Preconditioner

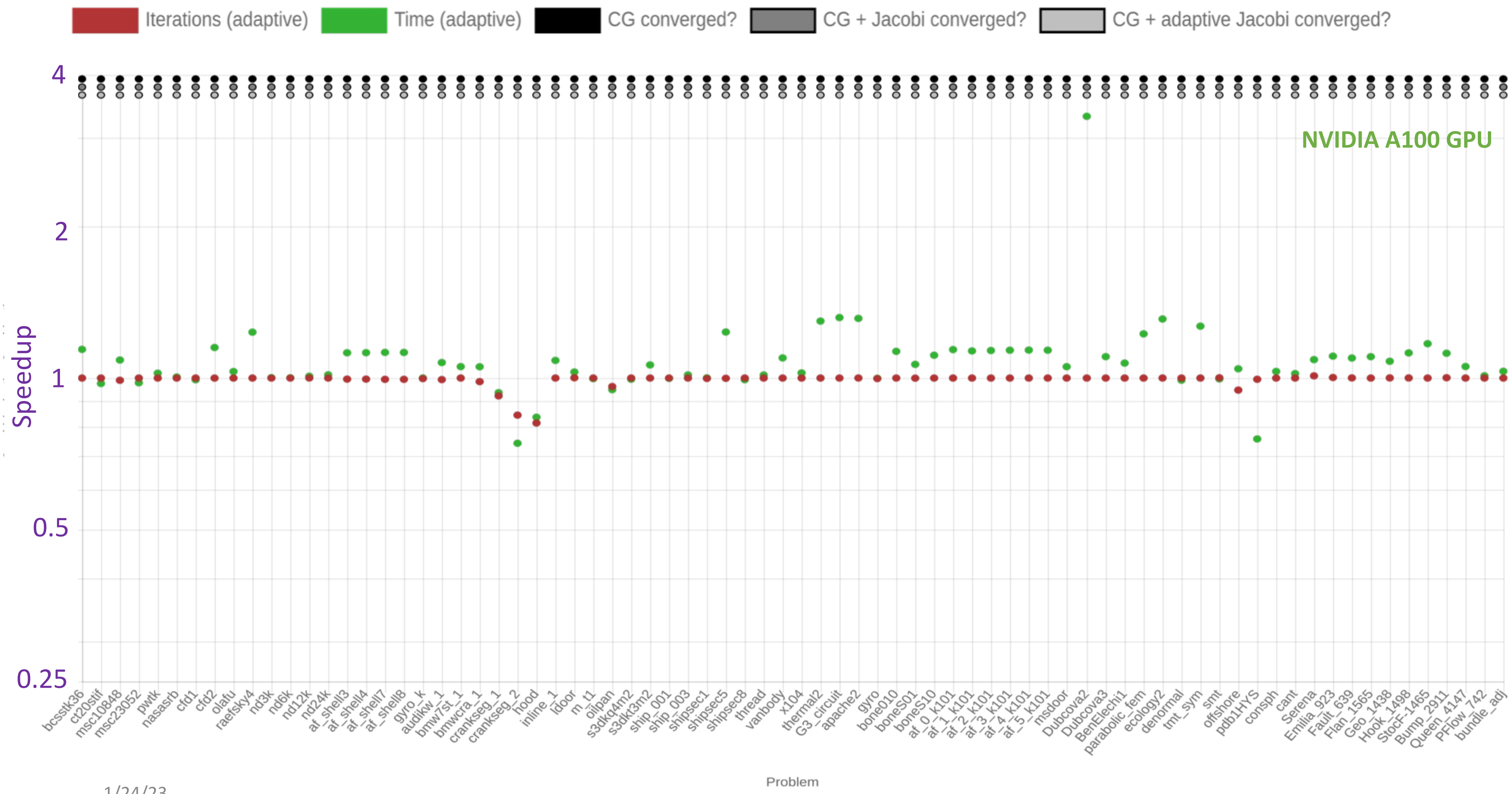
```
Initial residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
1390.67
Final residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
3.97985e-06
CG iteration count:     4797
CG execution time [ms]: 2971.18
```

Double Precision CG + **Mixed Precision Preconditioner**

```
Initial residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
1390.67
Final residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
3.98574e-06
CG iteration count:     4794
CG execution time [ms]: 2568.1
```

16% runtime improvement

Experiments based on the Ginkgo library https://ginkgo-project.github.io/
ginkgo/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp

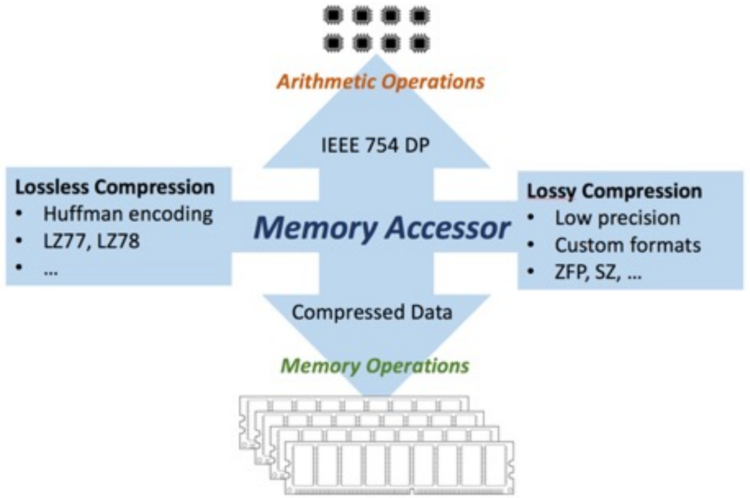


Iterations (adaptive)
Time (adaptive)
CG converged?
CG + Jacobi converged?
CG + adaptive Jacobi converged?
NVIDIA A100 GPU
Speedup
4
2
1
0.5
0.25
bcsstk36 ct20stif msc10848 msc23052 pwtk nasasrb cfd1 cfd2 olafu raefsky4 nd3k nd6k nd12k nd24k af_shell3 af_shell4 af_shell7 af_shell8 gyro_k audikw_1 bmw7st_1 bmwcra_1 crankseg_1 crankseg_2 hood inline_1 ldoor m_t1 oilpan s3dkq4m2 s3dkt3m2 ship_001 ship_003 shipsec1 shipsec5 shipsec8 thread vanbody x104 thermal2 G3_circuit apache2 gyro bone010 boneS01 boneS10 af_0_k101 af_1_k101 af_2_k101 af_3_k101 af_4_k101 af_5_k101 msdoor Dubcova2 Dubcova3 BenElechi1 parabolic_fem ecology2 denormal tmt_sym smt offshore pdb1HYS consph cant Serena Emilia_923 Fault_639 Flan_1565 Geo_1438 Hook_1498 StocF-1465 Bump_2911 Queen_4147 PFlow_742 bundle_adj
Problem

Use the memory accessor to boost performance

Can we use the memory accessor to accelerate the algorithm without changing the final result?

- Yes, if we can do all operations in registers and write the final result in high precision.
- Not in general, if we read/write intermediate date in low precision.
- We need to analyze the error propagation and adapt the algorithms to the application & data.



Possibilities in the context of solving linear systems:

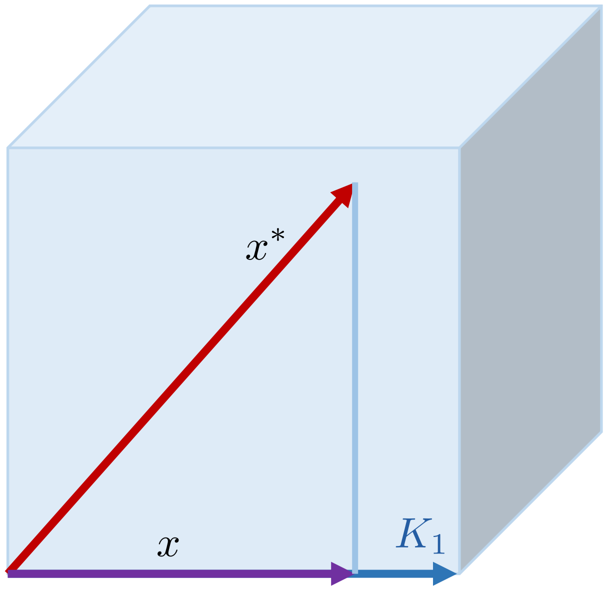
- *Approximate linear operators / Preconditioners / Inner solvers;*
- ***"Self-healing" iterative methods;***

Rethinking Algorithms: Self-Healing Iterative Methods

- **Krylov iterative solvers**
- Krylov methods aim at approximating the solution to a linear problem in a subspace.
- Over the iterations, a nested sequence of Krylov subspaces is generated, adding one basis vector in each iteration.
- Orthonormalization ensures a orthonormal basis is formed (Classical Gram-Schmidt, Modified Gram Schmidt…).

$$K_0 \subset K_1 \subset K_2 \subset \ldots$$

$$K_i(A, r) = span\{b, Ab, A^2b, \ldots A^{i-1}b\}$$

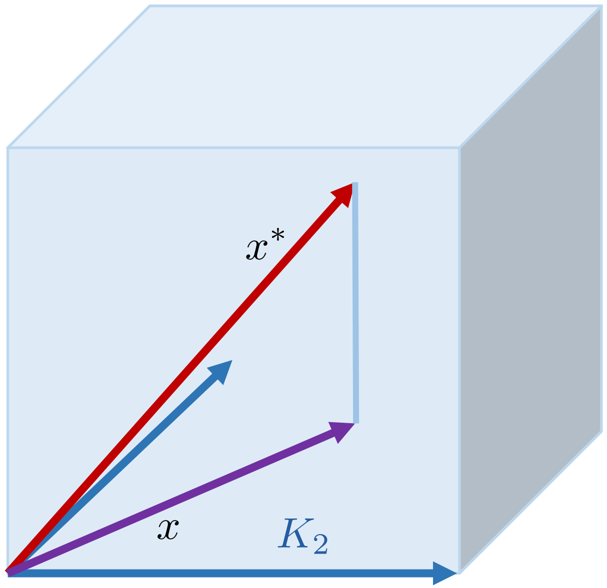


Rethinking Algorithms: Self-Healing Iterative Methods

- **Krylov iterative solvers**
- Krylov methods aim at approximating the solution to a linear problem in a subspace.
- Over the iterations, a nested sequence of Krylov subspaces is generated, adding one basis vector in each iteration.
- Orthonormalization ensures a orthonormal basis is formed (Classical Gram-Schmidt, Modified Gram Schmidt…).

$$K_0 \subset K_1 \subset K_2 \subset \dots$$

$$K_i(A, r) = span\{b, Ab, A^2b, \dots A^{i-1}b\}$$

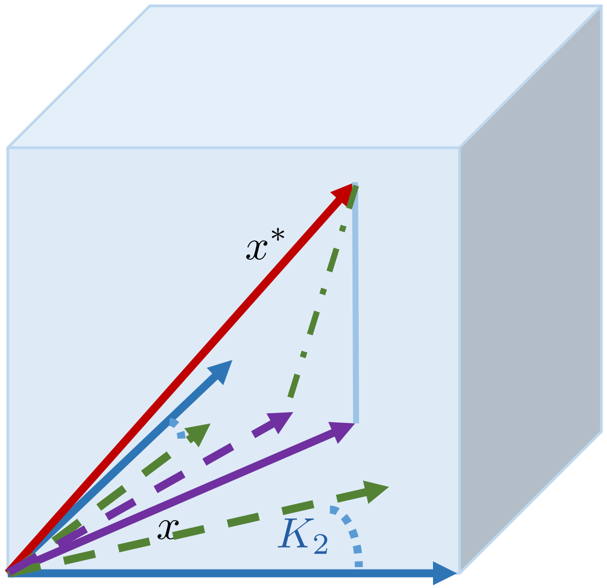


Rethinking Algorithms: Self-Healing Iterative Methods

- **Krylov iterative solvers**
- Krylov methods aim at approximating the solution to a linear problem in a subspace.
- Over the iterations, a nested sequence of Krylov subspaces is generated, adding one basis vector in each iteration.
- Orthonormalization ensures a orthonormal basis is formed (Classical Gram-Schmidt, Modified Gram Schmidt…).

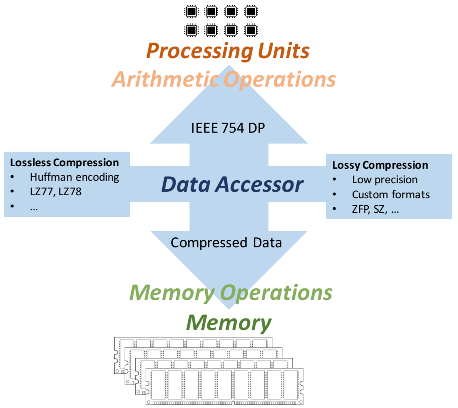
$$K_0 \subset K_1 \subset K_2 \subset \ldots$$

$$K_i(A, r) = span\{b, Ab, A^2b, \ldots A^{i-1}b\}$$



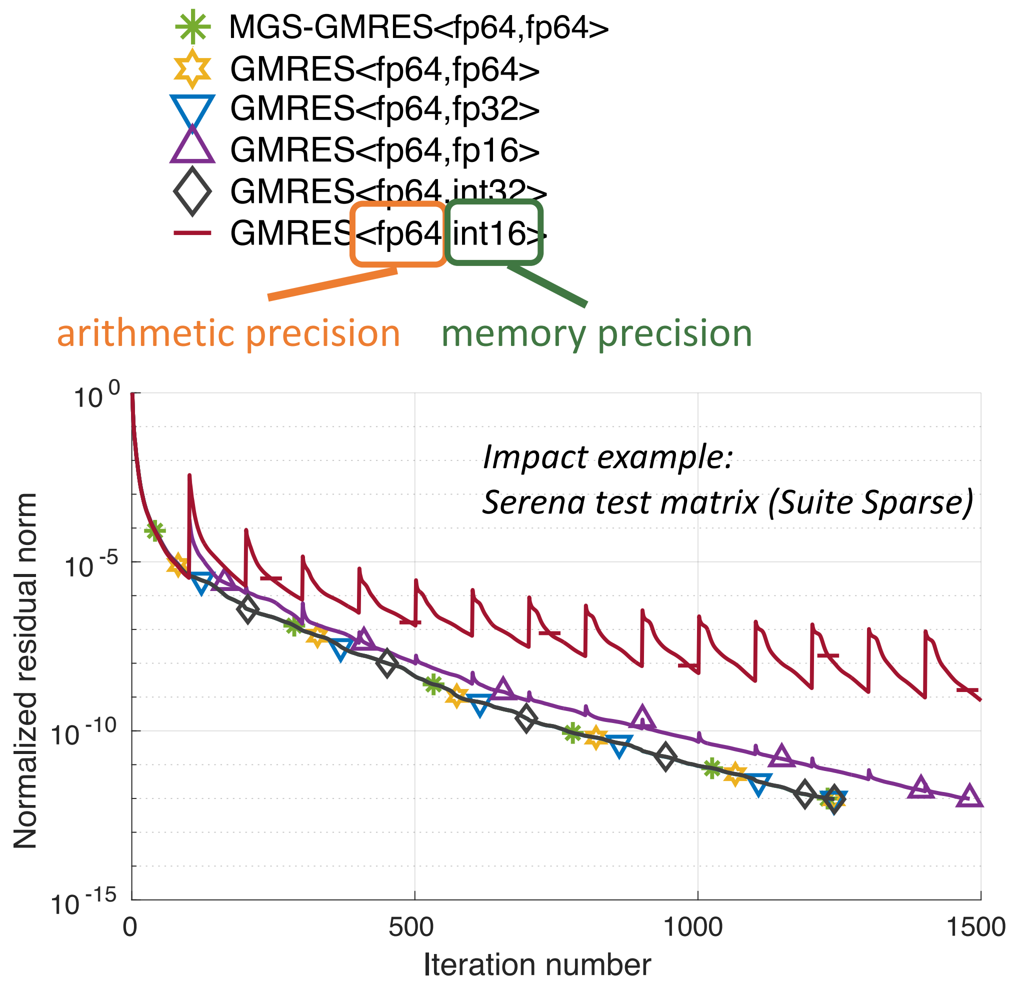
Compressed Basis (CB-) GMRES

- Use double precision in all arithmetic operations;
- Store Krylov basis vectors in lower precision;
  - Search directions are no longer DP-orthogonal;
  - Hessenberg system maps solution to “perturbed” Krylov subspace;
  - Additional iterations may be needed;
  - As long as the loss-of-orthogonality is moderate, we should see moderate convergence degradation;



Compressed Basis (CB-) GMRES

- **Krylov iterative solvers**
- Krylov methods aim at approximating the solution to a linear problem in a subspace.
- Over the iterations, a nested sequence of Krylov subspaces is generated, adding one basis vector in each iteration.
- Orthonormalization ensures a orthonormal basis is formed (Classical Gram-Schmidt, Modified Gram Schmidt…).



Compressed Basis (CB-) GMRES

- Use double precision in all arithmetic operations;
- Store Krylov basis vectors in lower precision;
  - Search directions are no longer DP-orthogonal;
  - Hessenberg system maps solution to "perturbed" Krylov subspace;
  - Additional iterations may be needed;
  - As long as the loss-of-orthogonality is moderate, we should see moderate convergence degradation;

Running iterative methods in different precision formats

Linear System Ax=b with cond(A) ≈ 10^7

(apache2 from SuiteSparse) **NVIDIA V100 GPU**

```
Double precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 9.6639e-09
GMRES iteration count: 23271
GMRES execution time: 43801 ms
```

Relative residual ~10^{-12}

```
Single precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 0.00175464
GMRES iteration count: 25000
GMRES execution time: 27376 ms
```

Relative residual ~10^{-7}

~2x faster!

Running iterative methods in different precision formats

Linear System Ax=b with cond(A) ≈ 10^7

(apache2 from SuiteSparse) **NVIDIA V100 GPU**

```
Double precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 9.6639e-09
GMRES iteration count: 23271
GMRES execution time: 43801 ms
```

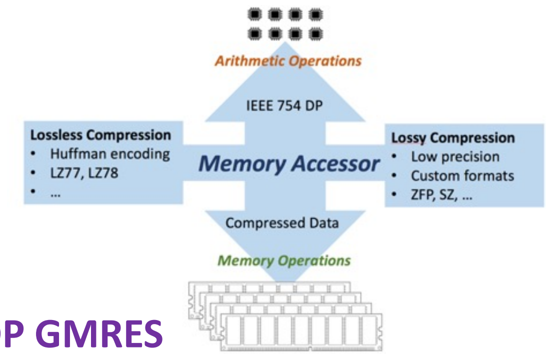
Relative residual ~10^{-12}

```
Single precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 0.00175464
GMRES iteration count: 25000
GMRES execution time: 27376 ms
```

Relative residual ~10^{-7}

```
Compressed Basis GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 9.6591e-09
GMRES iteration count: 23271
GMRES execution time: 29369 ms
```

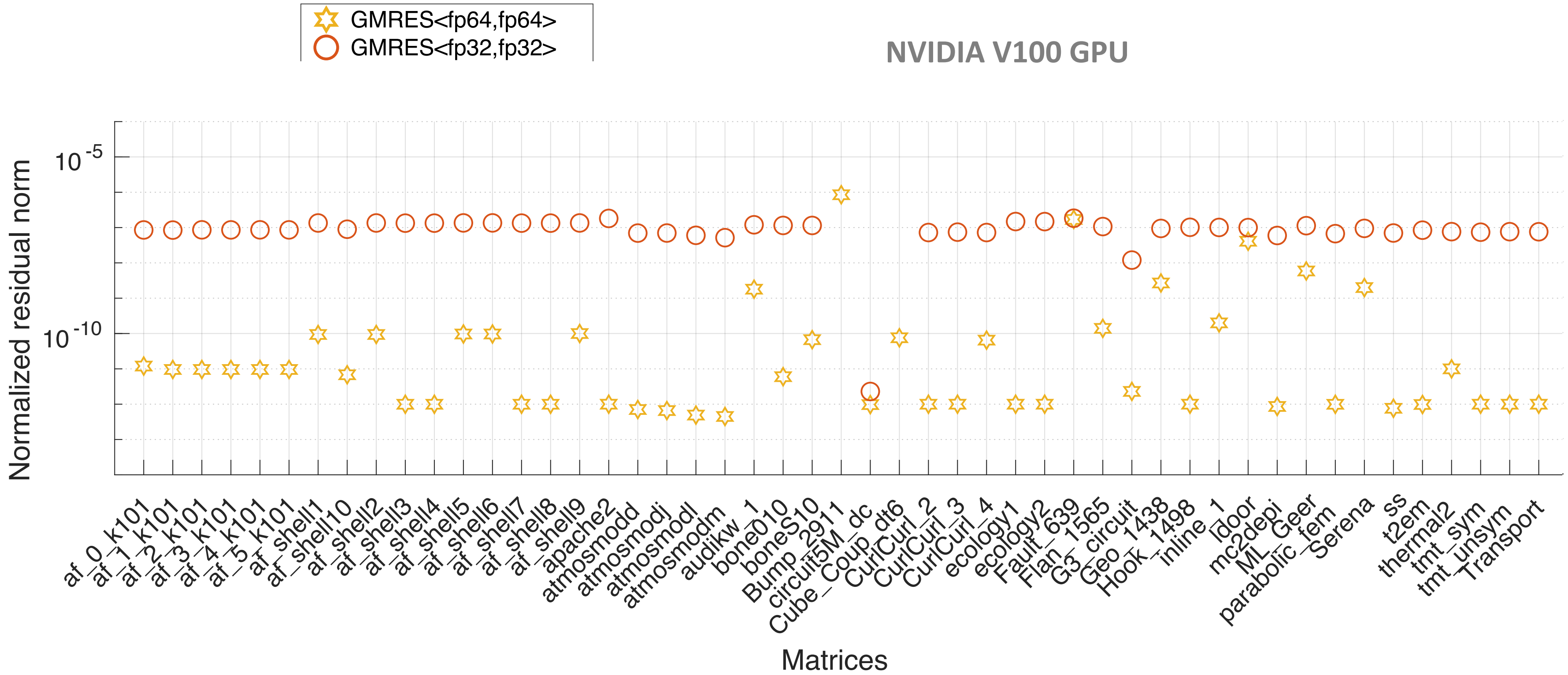
Relative residual ~10^{-12}



Accuracy of DP GMRES

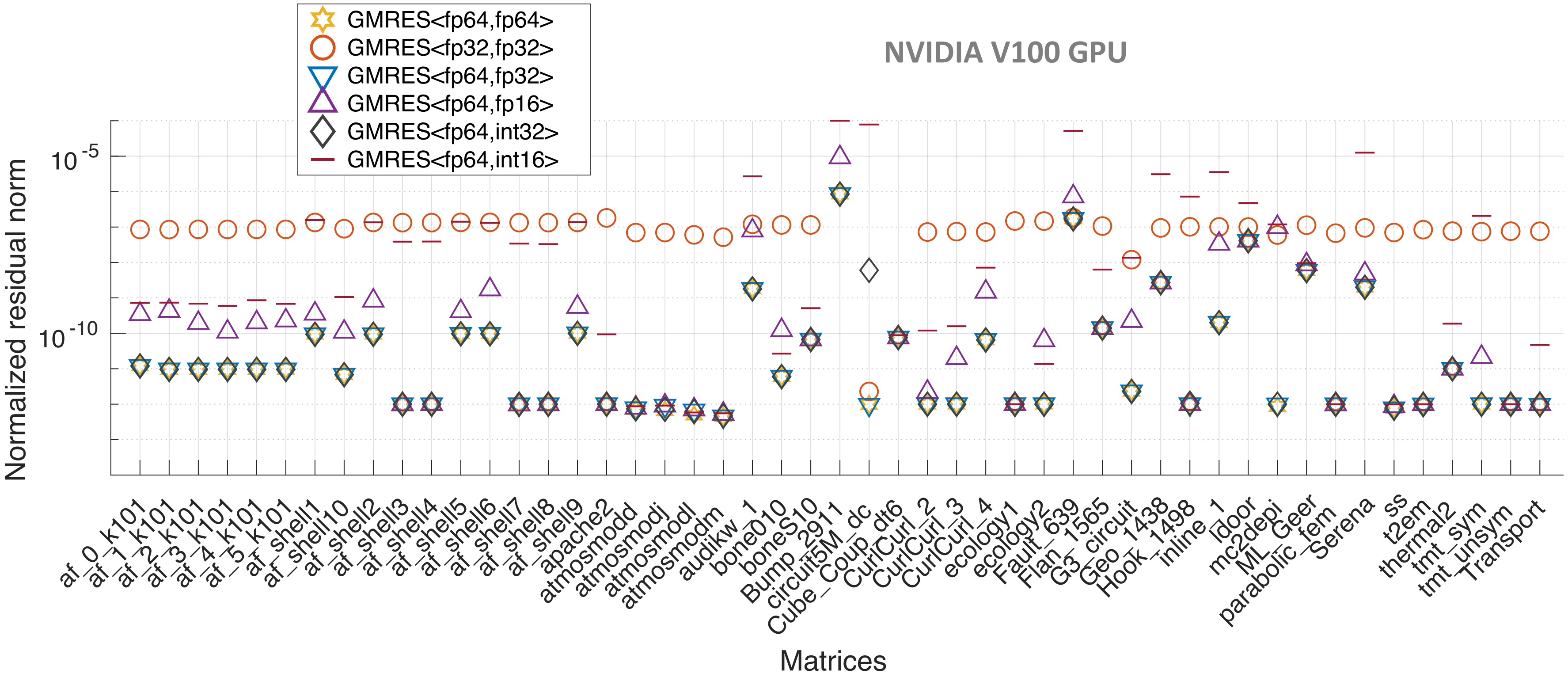
Performance similar to SP GMRES

Compressed Basis GMRES



Compressed Basis GMRES

- CB-GMRES using 32-bit storage preserves DP accuracy (SP-GMRES does not)

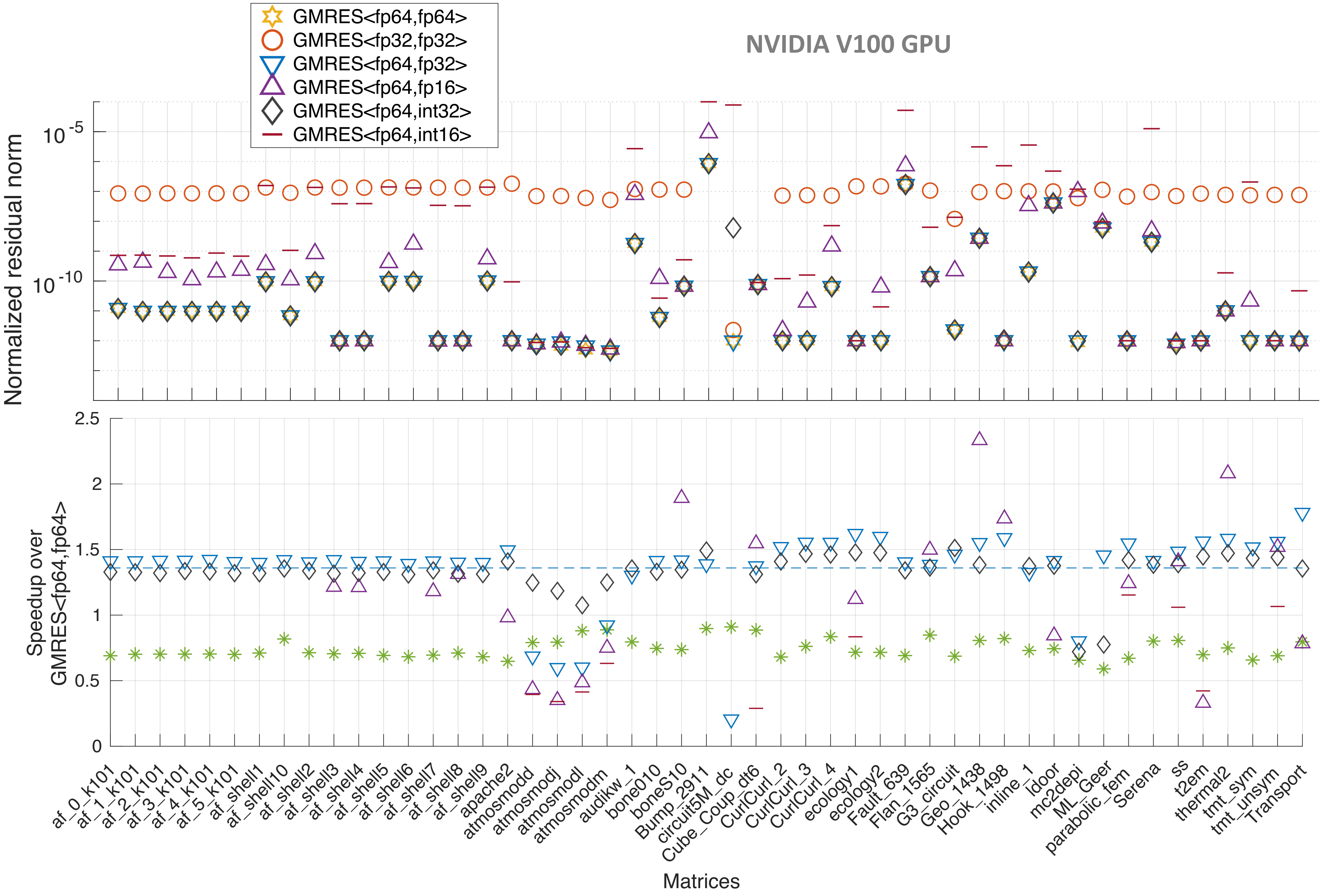


Compressed Basis GMRES

- CB-GMRES using 32-bit storage preserves DP accuracy (SP-GMRES does not)

- Speedups problem-dependent
- Speedup Ø1.4x (for restart 100)
- 16-bit storage mostly inefficient

Ginkgo



Integration into MFEM

Improve current Ginkgo-MFEM integration:

- √ MFEM and Ginkgo operate directly on same data without copies
- √ New `GinkgoExecutor` class automatically matches MFEM `Device` configuration - for CPU, CUDA, or HIP
- √ Ginkgo can use MFEM matrix-free operators in solvers

Add Ginkgo preconditioners to MFEM:

- √ Ginkgo preconditioners can be used with Ginkgo solvers, or used with MFEM solvers
- √ Includes Ginkgo's new ILU-ISAI/IC-ISAI preconditioners, which use the Incomplete Sparse Approximate Inverse to apply the ILU or IC factorization for improved GPU performance

Add new Ginkgo solver to MFEM:

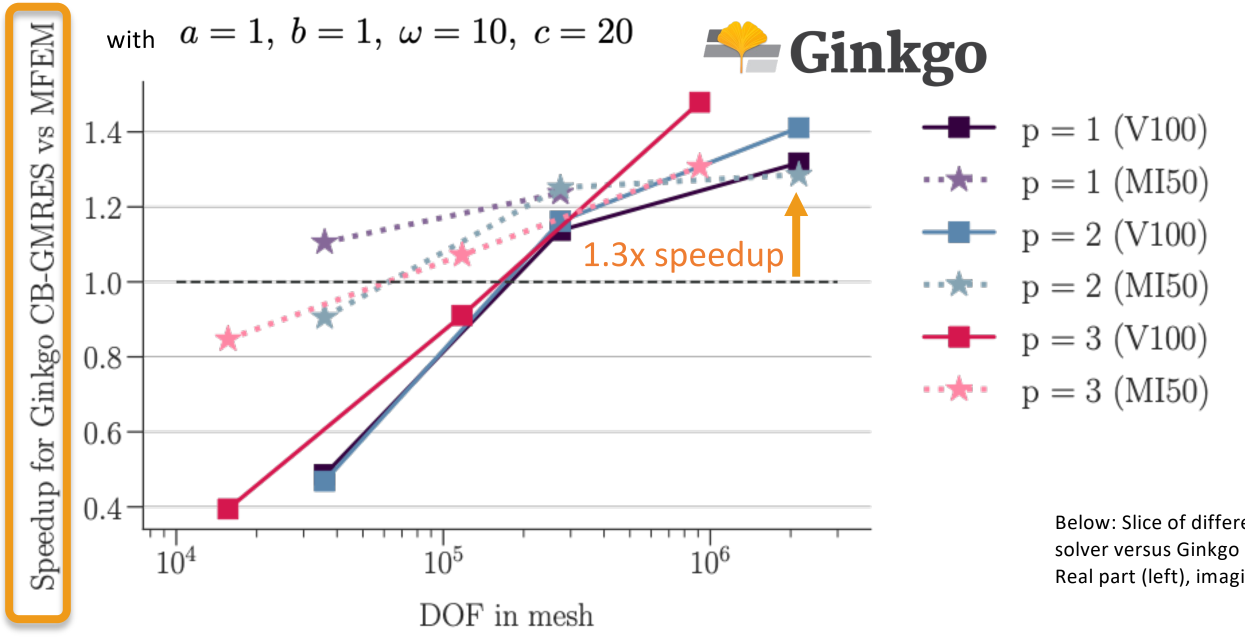
- √ Integration for Ginkgo's Compressed Basis GMRES solver, which uses mixed precision techniques for speedup (see example to right)

Example: Speeding up MFEM's "example 22" on NVIDIA and AMD GPUs

Example 22 solves harmonic oscillation problems, with a forced oscillation imposed at the boundary. For this test, we use variant 1:

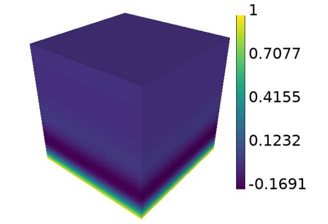
$$-\nabla \cdot (a\nabla u) - \omega^2 bu + i\omega cu = 0$$

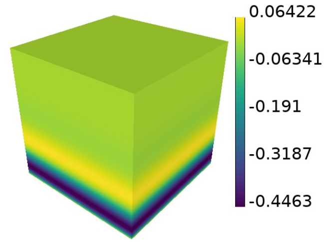
with $a = 1,\ b = 1,\ \omega = 10,\ c = 20$



Speedup of Ginkgo's Compressed Basis-GMRES solver vs MFEM's GMRES solver for three different orders of basis functions (p) for MFEM's example 22. The tests use the "partial assembly" type of MFEM matrix-free operators.

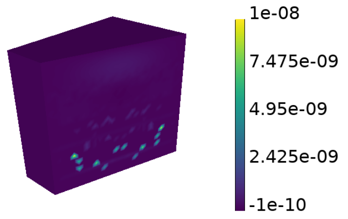
CUDA 10.1/NVIDIA V100 and ROCm 4.0/AMD MI50.
GMRES(50) used for both solvers. CB-GMRES used float/double.

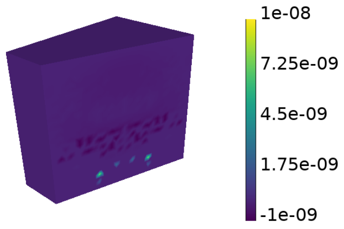




From top: Real part of solution, imaginary part of solution.

Below: Slice of difference in solution output using MFEM solver versus Ginkgo CB-GMRES.
Real part (left), imaginary part (right)





Natalie Beams (Univ. of Tennessee)

Using the memory accessor to boost accuracy

Instead of improving the performance of memory-bound high precision algorithms, the memory accessor can be used to increase the accuracy of memory-bound low precision algorithms – at no cost.

Design

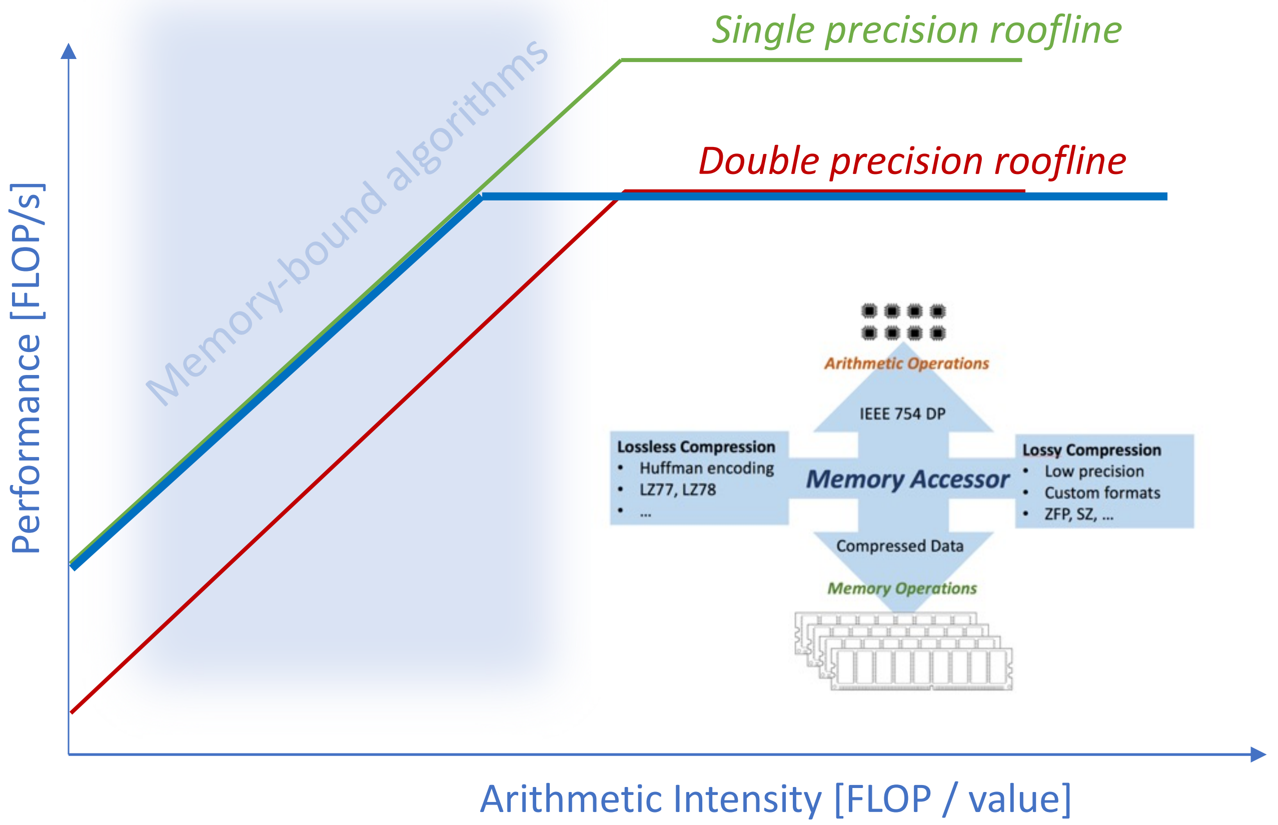
- Memory access in low precision (e.g. fp32);
- Computations in high precision (e.g. fp64);

Characteristics

- Performance of low precision BLAS;
- Higher accuracy than low precision BLAS;

Usage

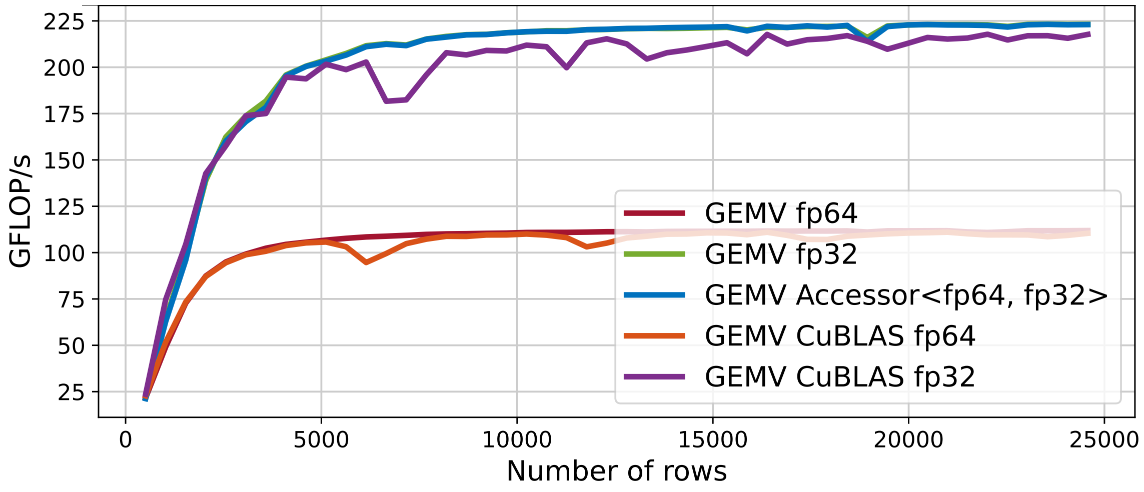
1. Can replace low precision BLAS to increase accuracy;
2. Can replace high precision BLAS if information loss is acceptable;
   (without having to deal with explicit mixed precision usage)



Accessor-BLAS: Replacing LP BLAS to improve accuracy

GEMV

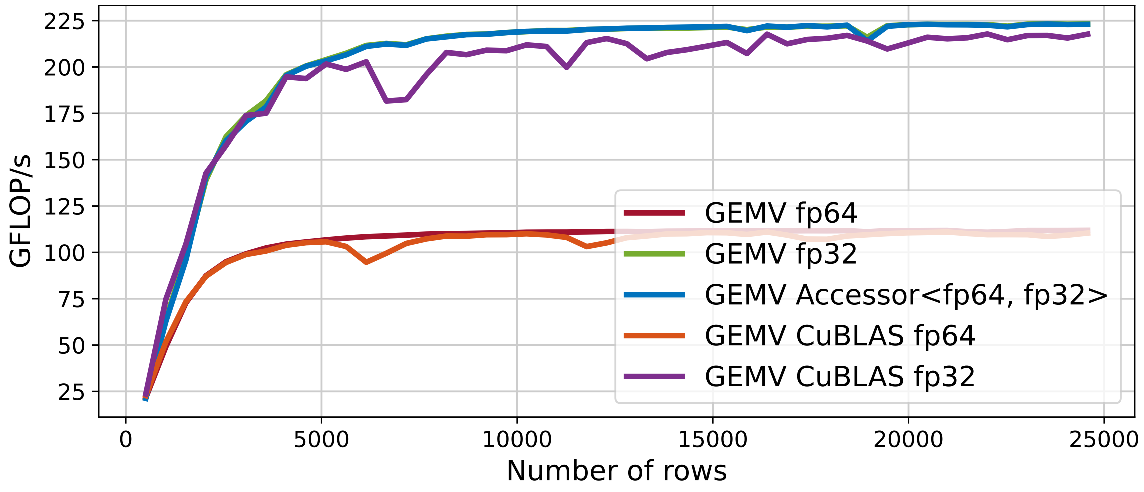
NVIDIA V100 GPU (Summit)

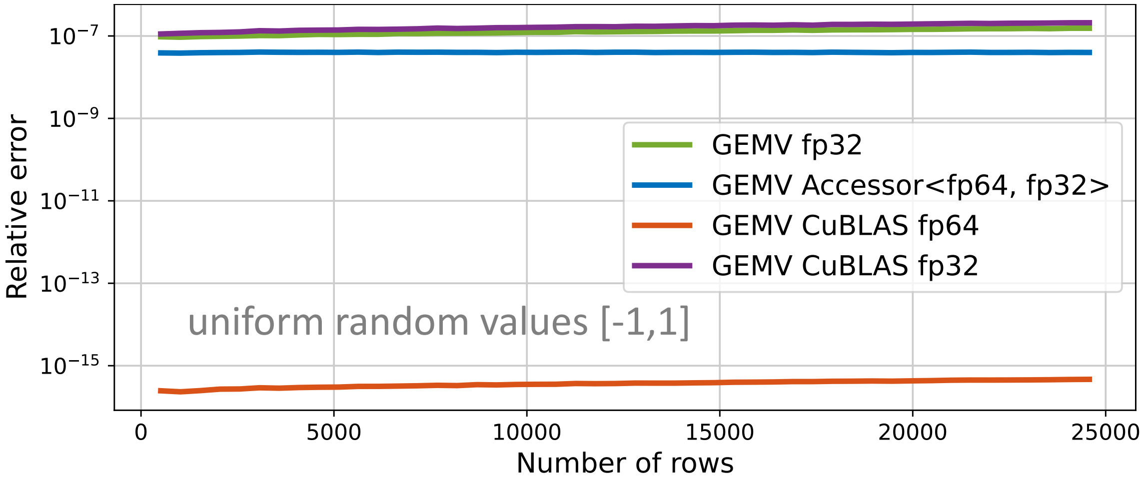


Accessor-BLAS: Replacing LP BLAS to improve accuracy

GEMV

NVIDIA V100 GPU (Summit)

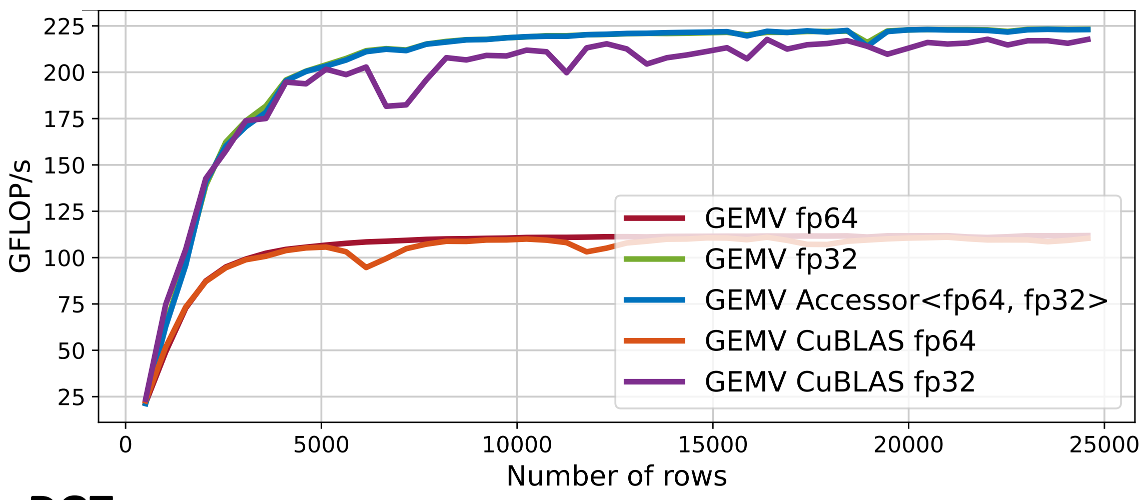


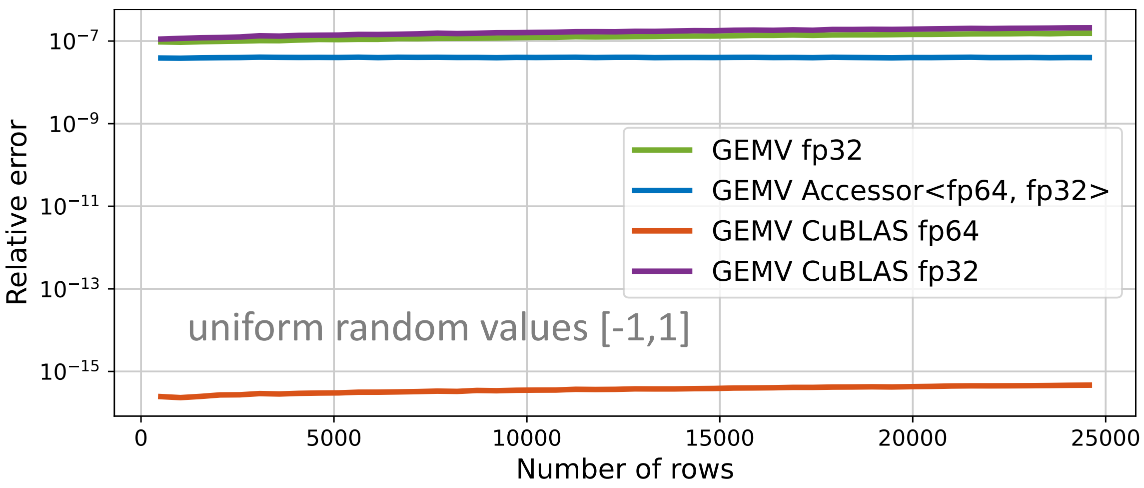


Accessor-BLAS: Replacing LP BLAS to improve accuracy

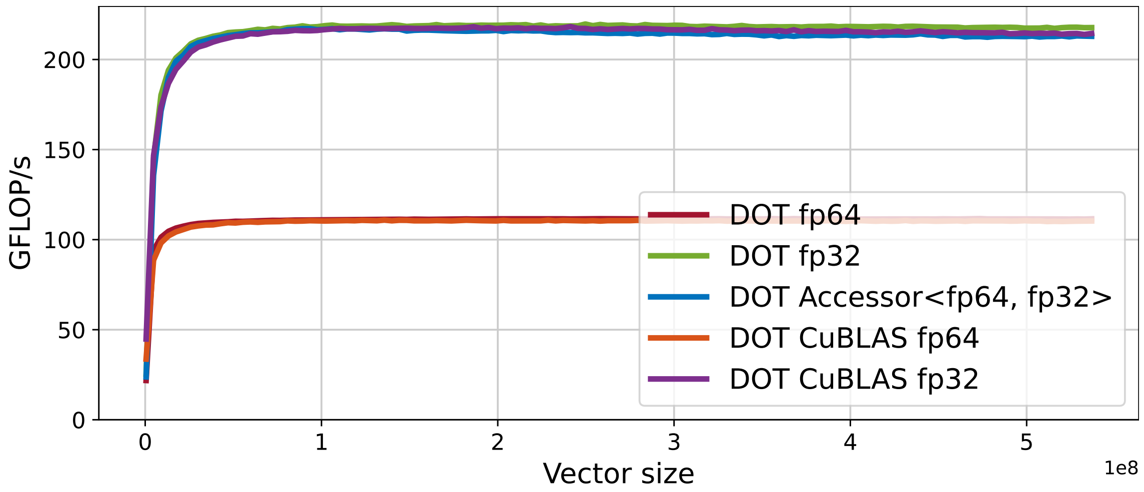
GEMV

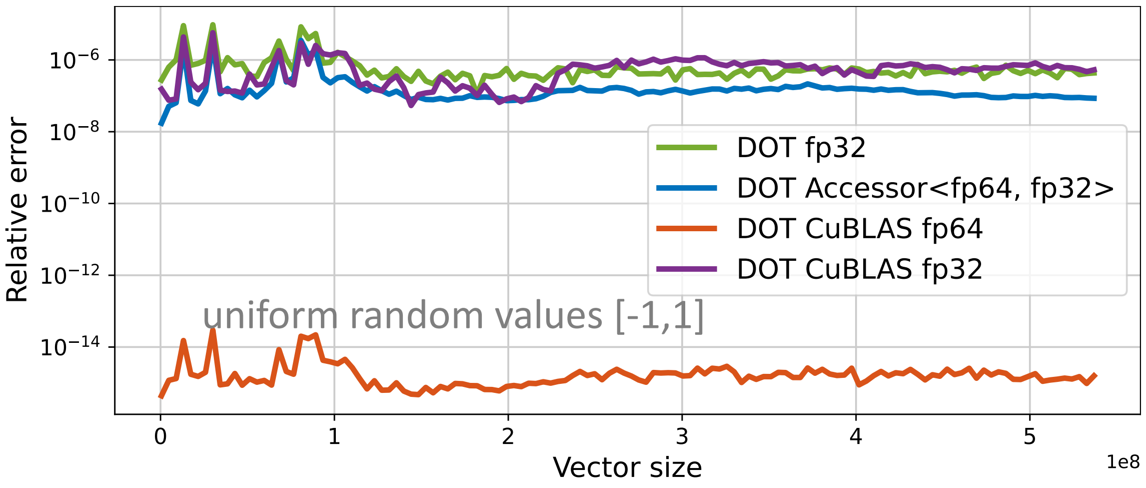
NVIDIA V100 GPU (Summit)





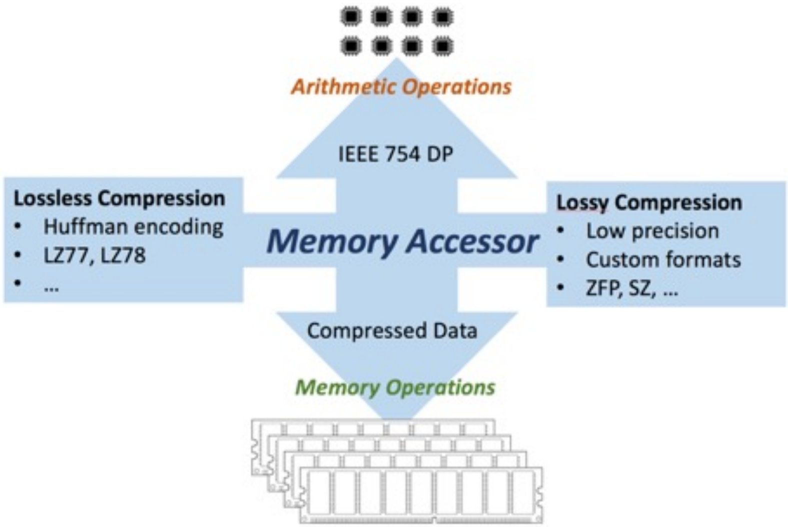
DOT





Summary and resources

- *For memory-bound algorithms, mixed precision can boost performance through reduced data movement.*
- **Memory accessor** allows to **compress data in main memory** but **do all arithmetic in high (double) precision**.
- **Approximate operators** (preconditioners, lower multigrid levels) and **self-healing iterative methods** can accept/compensate **information loss**.
- **Memory-bound low precision algorithms can increase accuracy at no cost.**



Ginkgo

Accessor-based GEMV and DOT available as open-source code:
https://github.com/ginkgo-project/accessor-BLAS

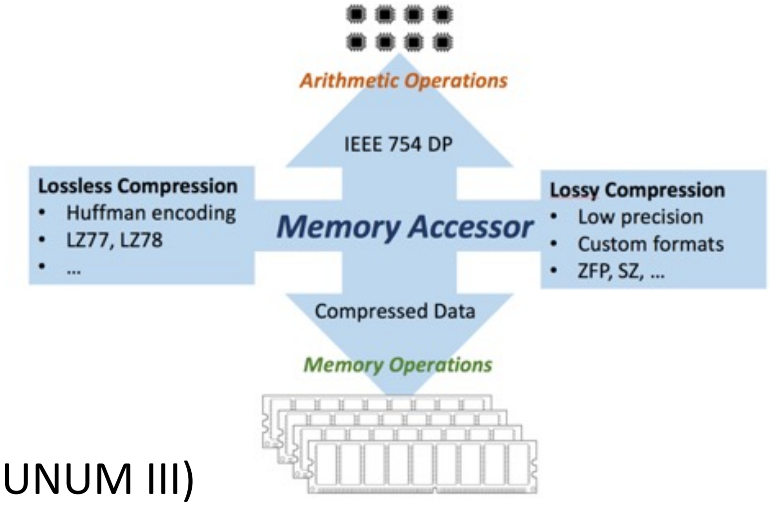
Mixed Precision block-Jacobi preconditioning:
https://github.com/ginkgo-project/ginkgo/tree/develop/examples/adaptiveprecision-blockjacobi

Mixed Precision Iterative Refinement:
https://github.com/ginkgo-project/ginkgo/tree/develop/examples/mixed-precision-ir

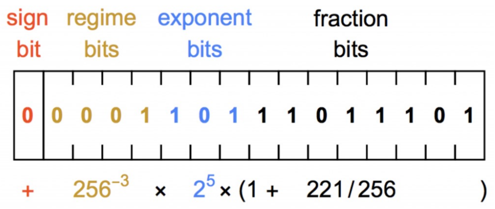
Compressed-Basis GMRES:
https://github.com/ginkgo-project/ginkgo/tree/develop/examples/cb-gmres

Let's try harder

- **IEEE 754 fp64 in arithmetic operations**
- **More sophisticated in-register compression**
  - Custom formats
  - Compression techniques (SZ, ZFP)
- **Store data in compressed format**



POSIT (UNUM III)



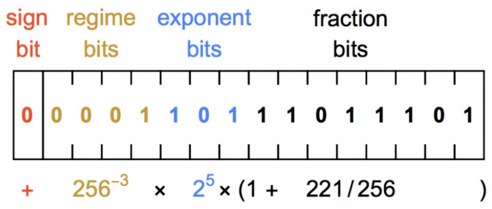
John L. Gustafson

Dynamic Range

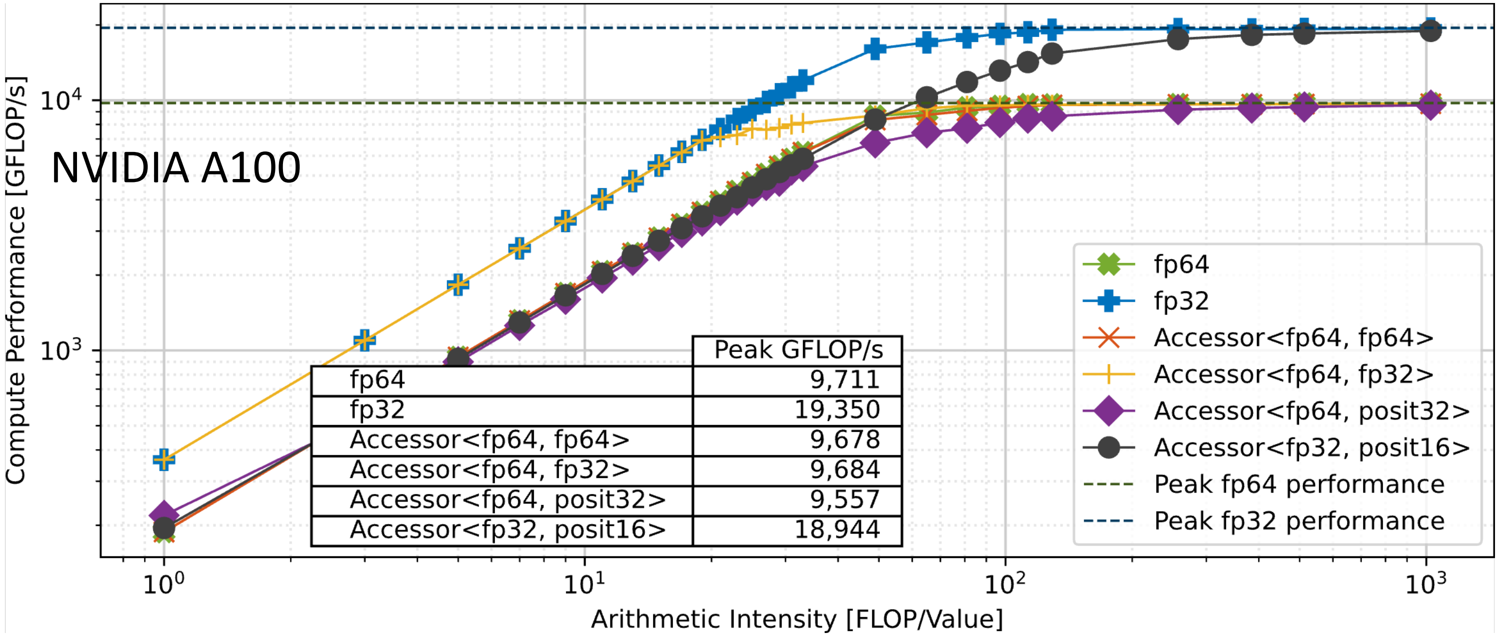
Size [bits]	IEEE exponent size [bits]	Approx. IEEE dynamic range	Approx. Posit dynamic range	Posit exp. bits
16	5	$[6 \cdot 10^{-8}, 7 \cdot 10^{4}]$	$[1 \cdot 10^{-17}, 7 \cdot 10^{16}]$	2
32	8	$[1 \cdot 10^{-45}, 3 \cdot 10^{38}]$	$[8 \cdot 10^{-37}, 1 \cdot 10^{36}]$	2
64	11	$[5 \cdot 10^{-324}, 2 \cdot 10^{308}]$	$[2 \cdot 10^{-75}, 5 \cdot 10^{75}]$	2

	IEEE	Posit
Special values	• IEEE defines ± 0, $\pm\infty$ and NaN (quiet and signaling) as special values • A lot of NaN representations (fp32 has $2^{24} - 1 \approx 10^{7}$ different NaNs)	• Posit only has 2: 0 and NaR • NaR (Not a Real) is used as an error-value (like NaN and $\pm\infty$)
Gradual over- and underflow	• IEEE supports gradual underflow with subnormal numbers (fraction has an implicit 0.) • No support for gradual overflow	• Posit supports both gradual over- and underflow through the regime • The farther away from 1.0, the fewer fraction bits

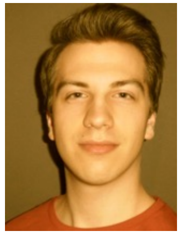
Using POSIT as memory format



Unum Type III
John L. Gustafson

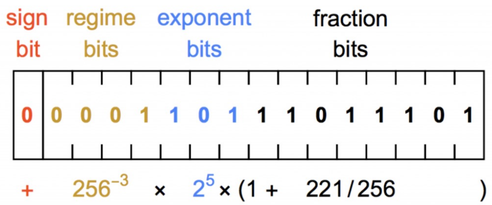


	Peak GFLOP/s
fp64	9,711
fp32	19,350
Accessor<fp64, fp64>	9,678
Accessor<fp64, fp32>	9,684
Accessor<fp64, posit32>	9,557
Accessor<fp32, posit16>	18,944

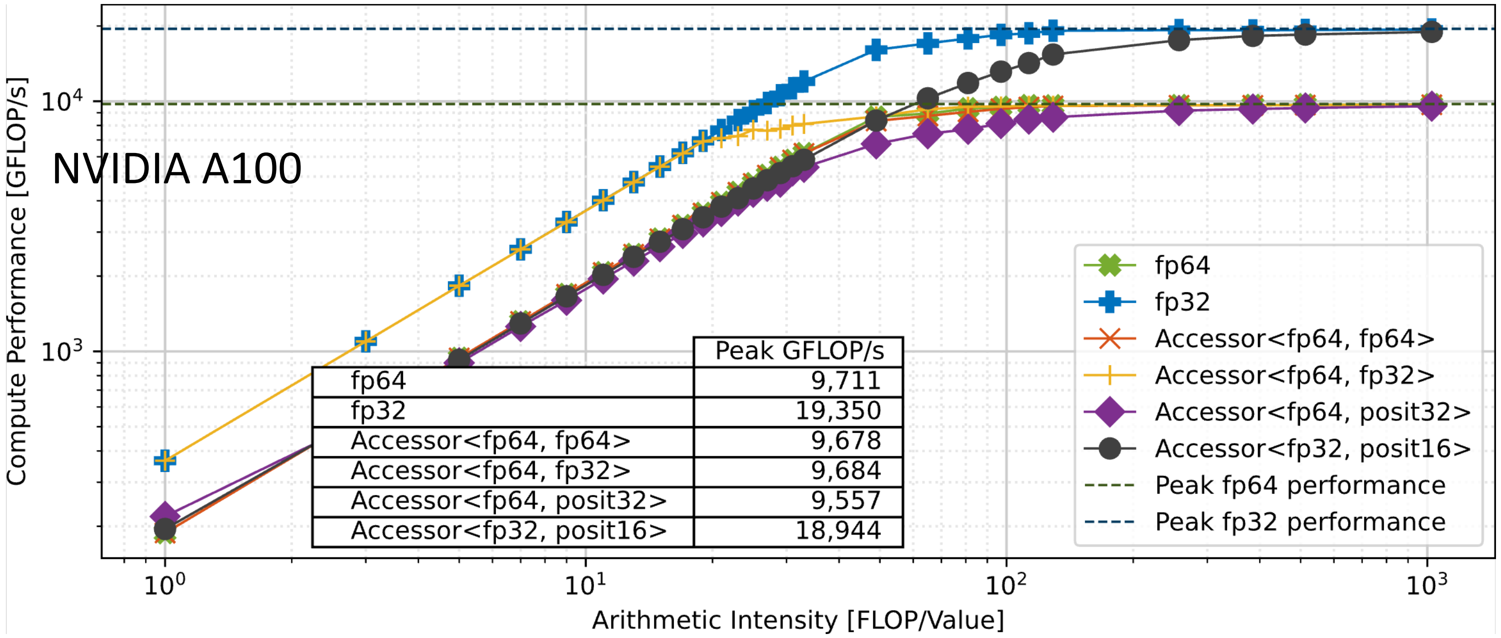


T. Grützmacher

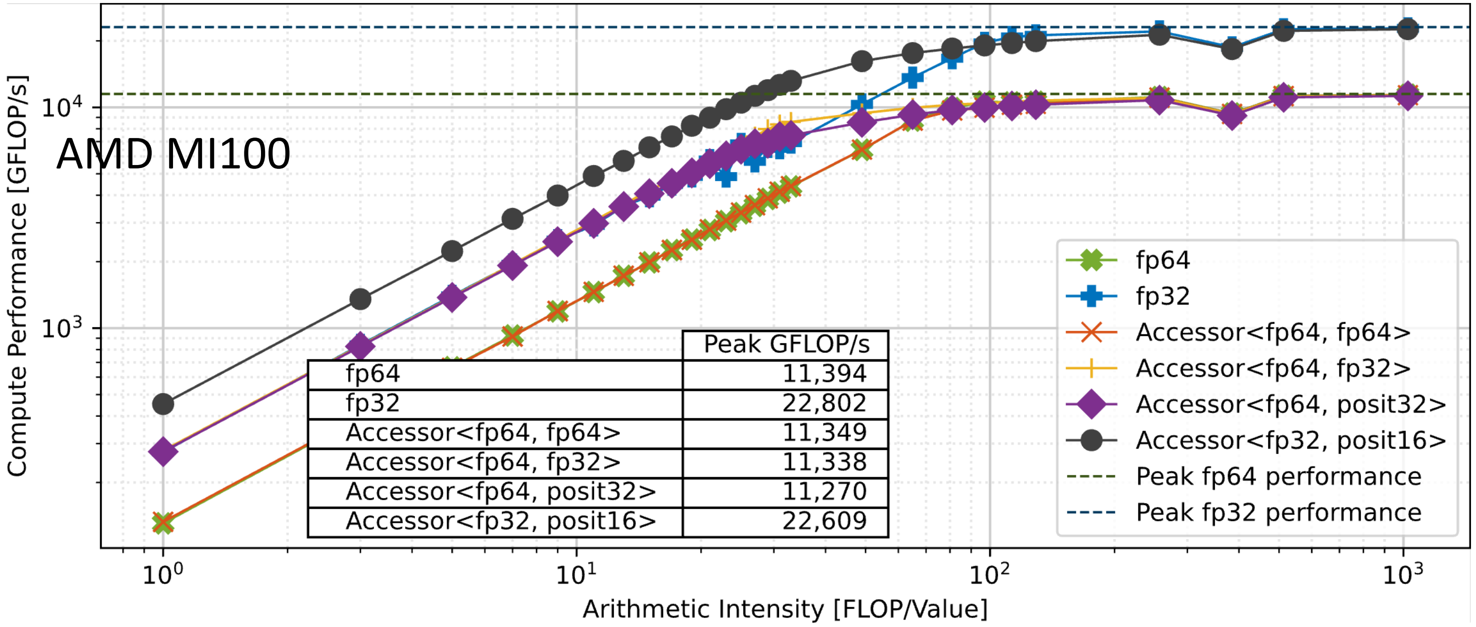
Using POSIT as memory format



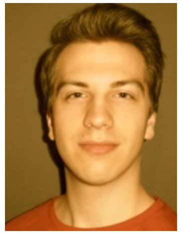
Unum Type III
John L. Gustafson



	Peak GFLOP/s
fp64	9,711
fp32	19,350
Accessor<fp64, fp64>	9,678
Accessor<fp64, fp32>	9,684
Accessor<fp64, posit32>	9,557
Accessor<fp32, posit16>	18,944



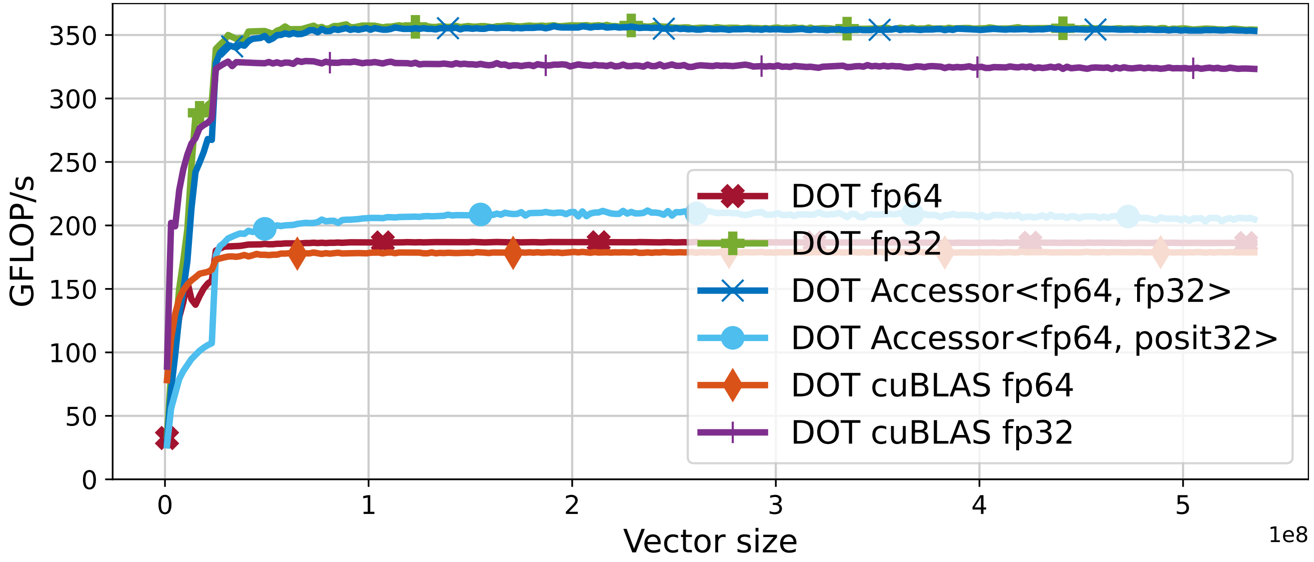
	Peak GFLOP/s
fp64	11,394
fp32	22,802
Accessor<fp64, fp64>	11,349
Accessor<fp64, fp32>	11,338
Accessor<fp64, posit32>	11,270
Accessor<fp32, posit16>	22,609

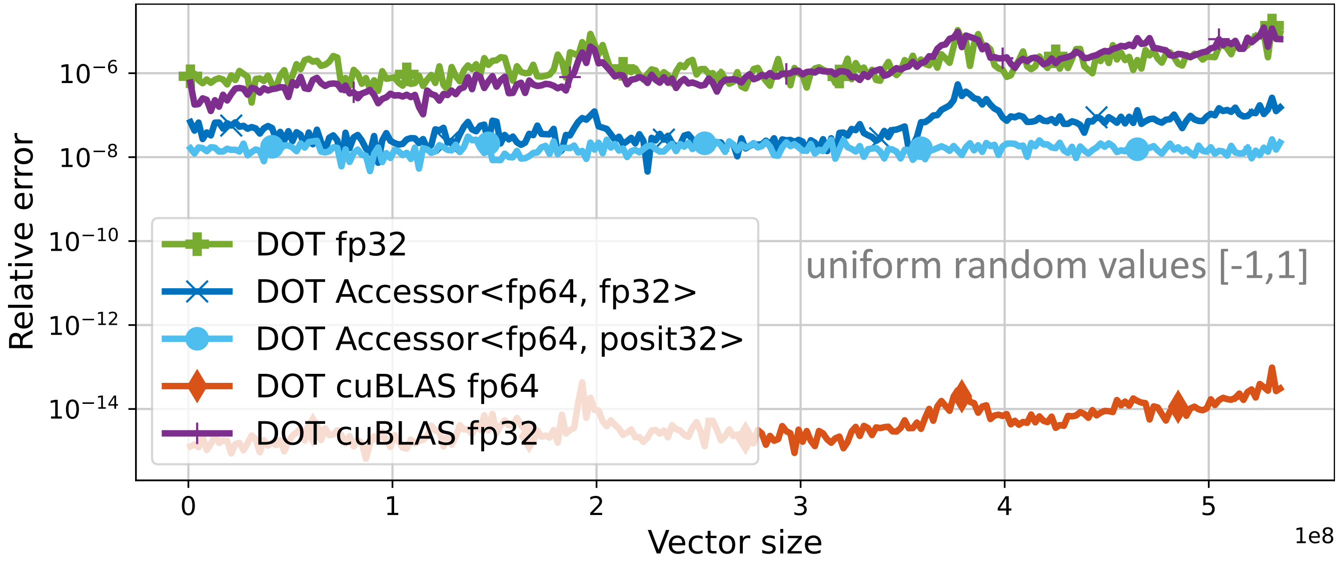


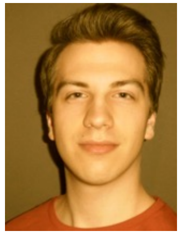
T. Grützmacher

Using POSIT as memory format

NVIDIA A100





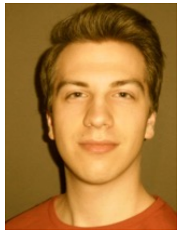


T. Grützmacher

Using ZFP / SZ compression as memory format


F. Capello

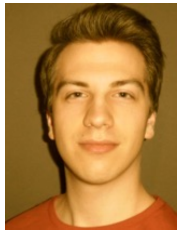

R. Underwood


T. Grützmacher

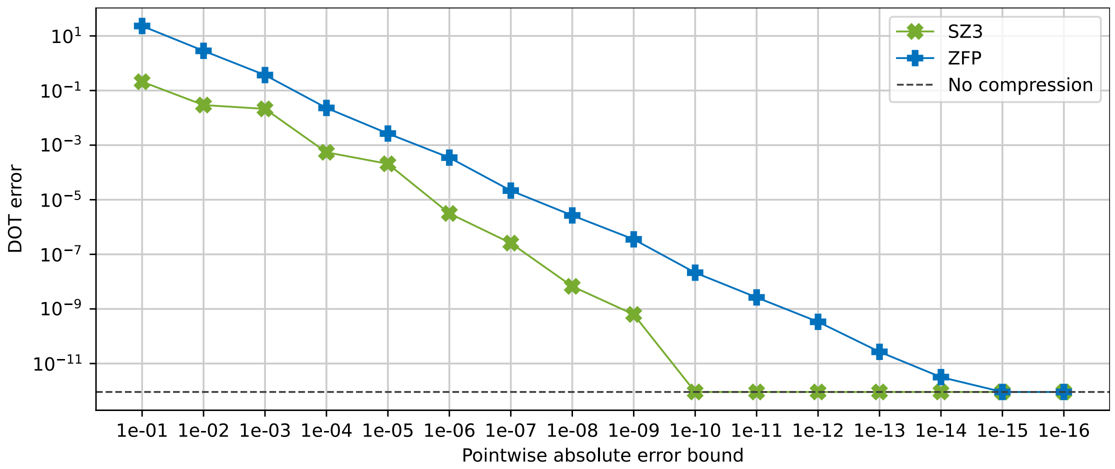
Using ZFP / SZ compression as memory format


F. Capello

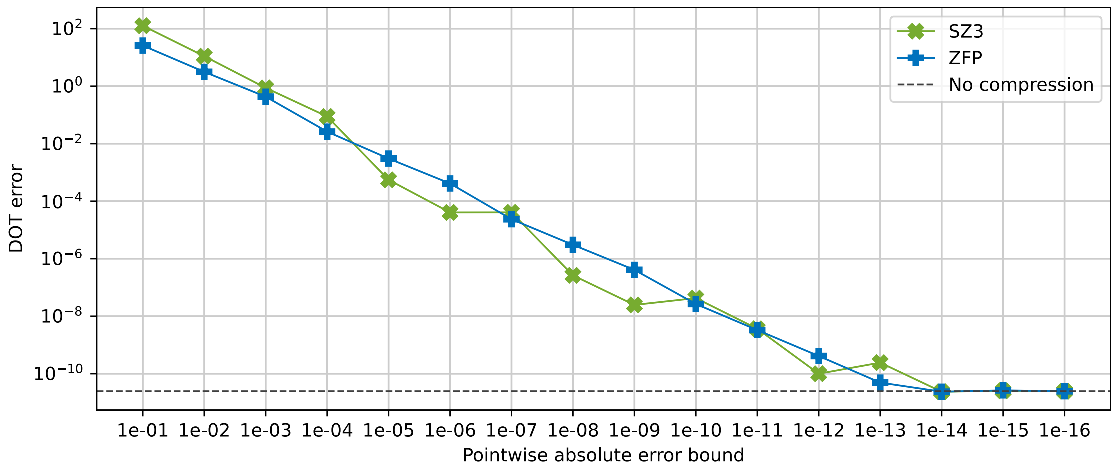

R. Underwood


T. Grützmacher

Random data



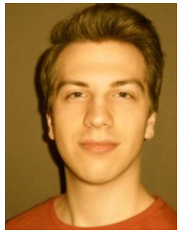
1D sine function



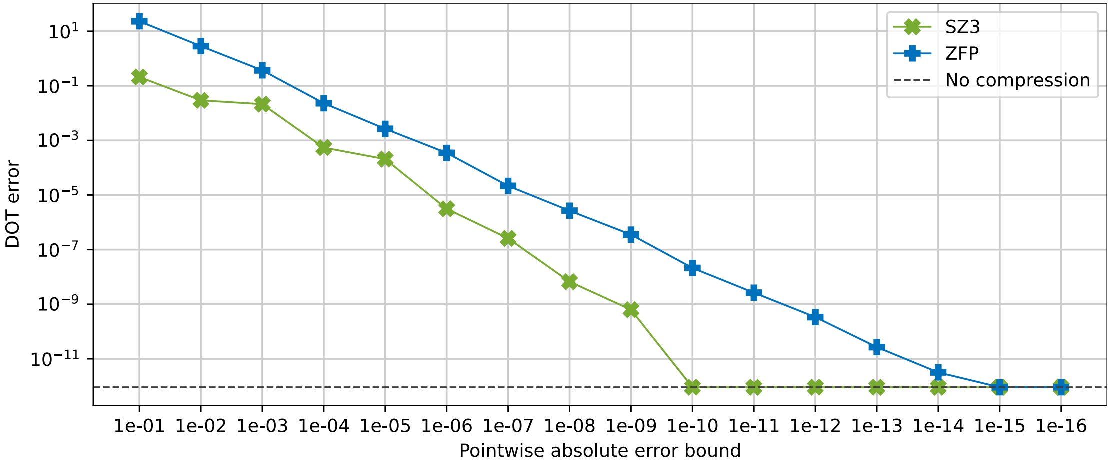
Using ZFP / SZ compression as memory format


F. Capello

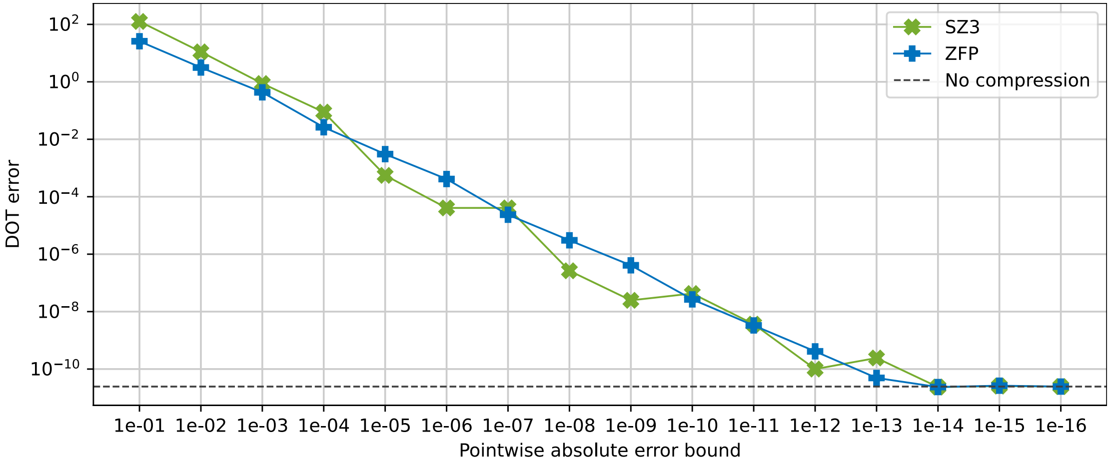

R. Underwood

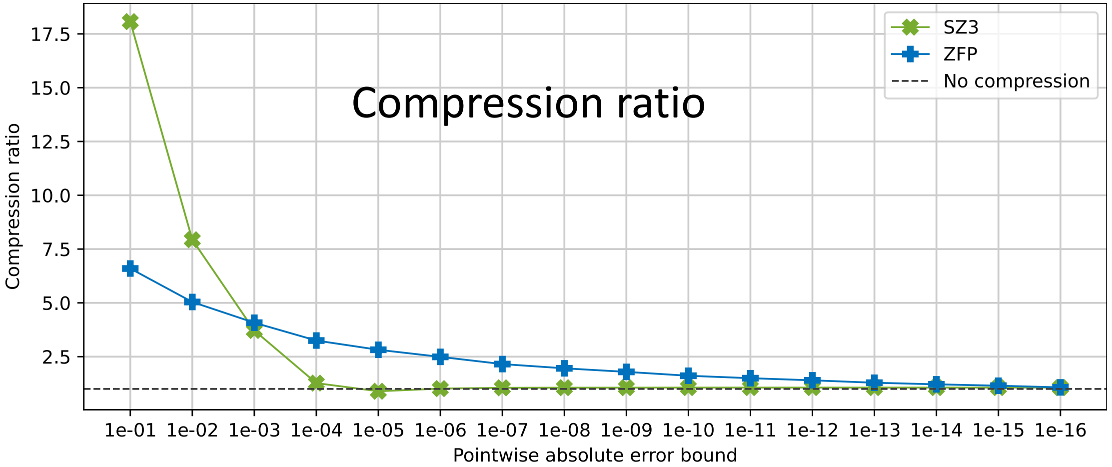

T. Grützmacher

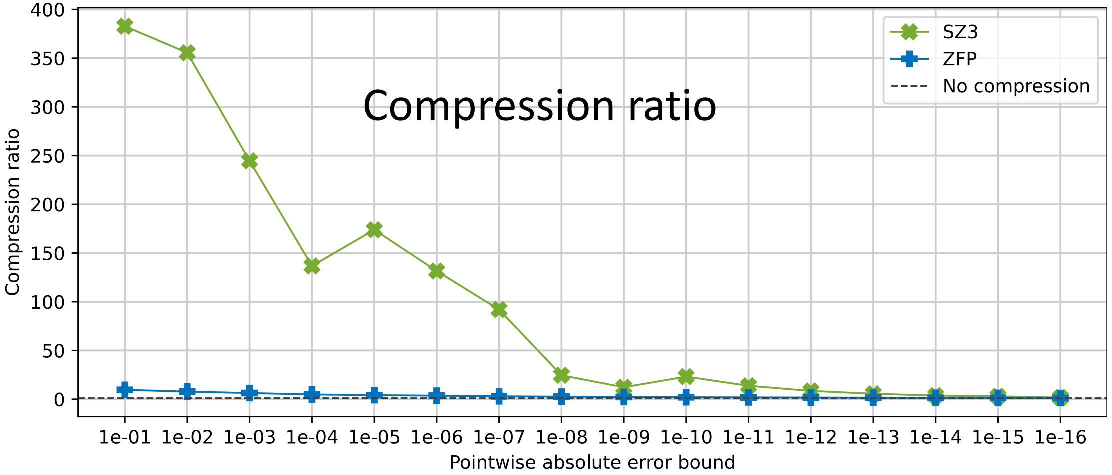
Random data



1D sine function

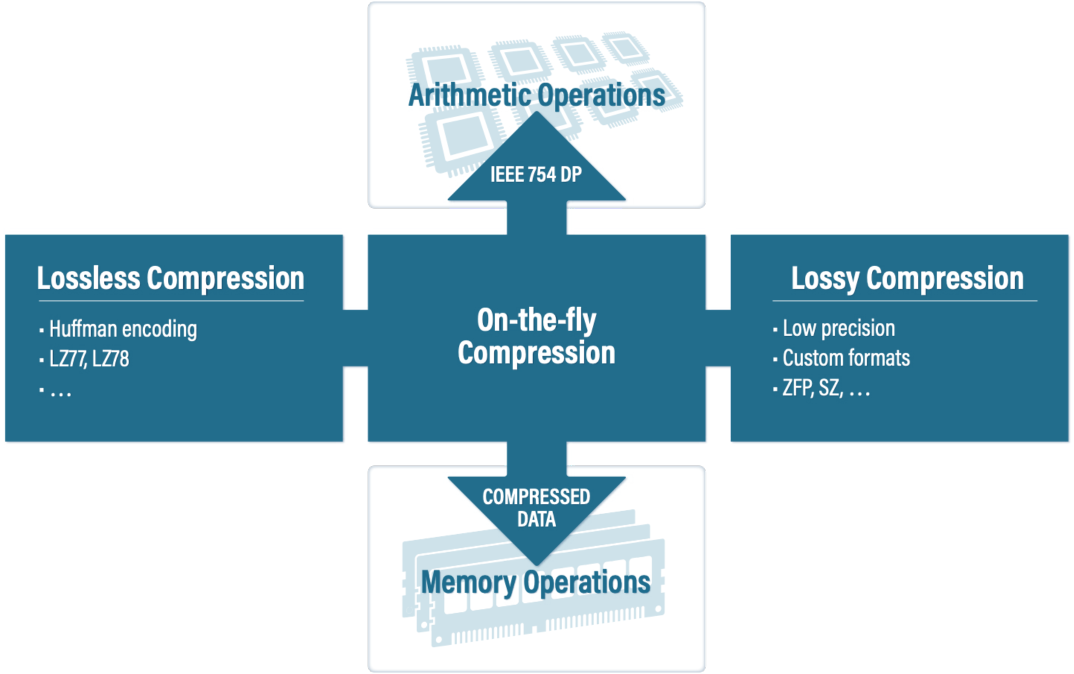






Let's try harder

- IEEE 754 fp64 in arithmetic operations
- More sophisticated in-register compression
    - Custom formats
    - Compression techniques (SZ, ZFP)
- Store data in compressed format



Trade-off:

- Aggressive compression comes with larger information loss
- Element-wise compression allows only for moderate compression ratios
- Block-wise compression makes random access difficult
- Register count limits the block size (hardware specific)
- Data-dependent compression efficiency