The Future is Sparse Workshop
Friday, Nov 17, 2023

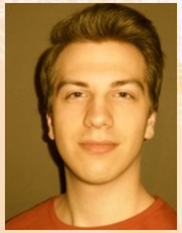
Tensor Cores for Matrix Multiplication Are on the Rise – Is There Any Hope for Sparse Operations?

Hartwig Anzt
University of Tennessee, Knoxville


Terry Cojean

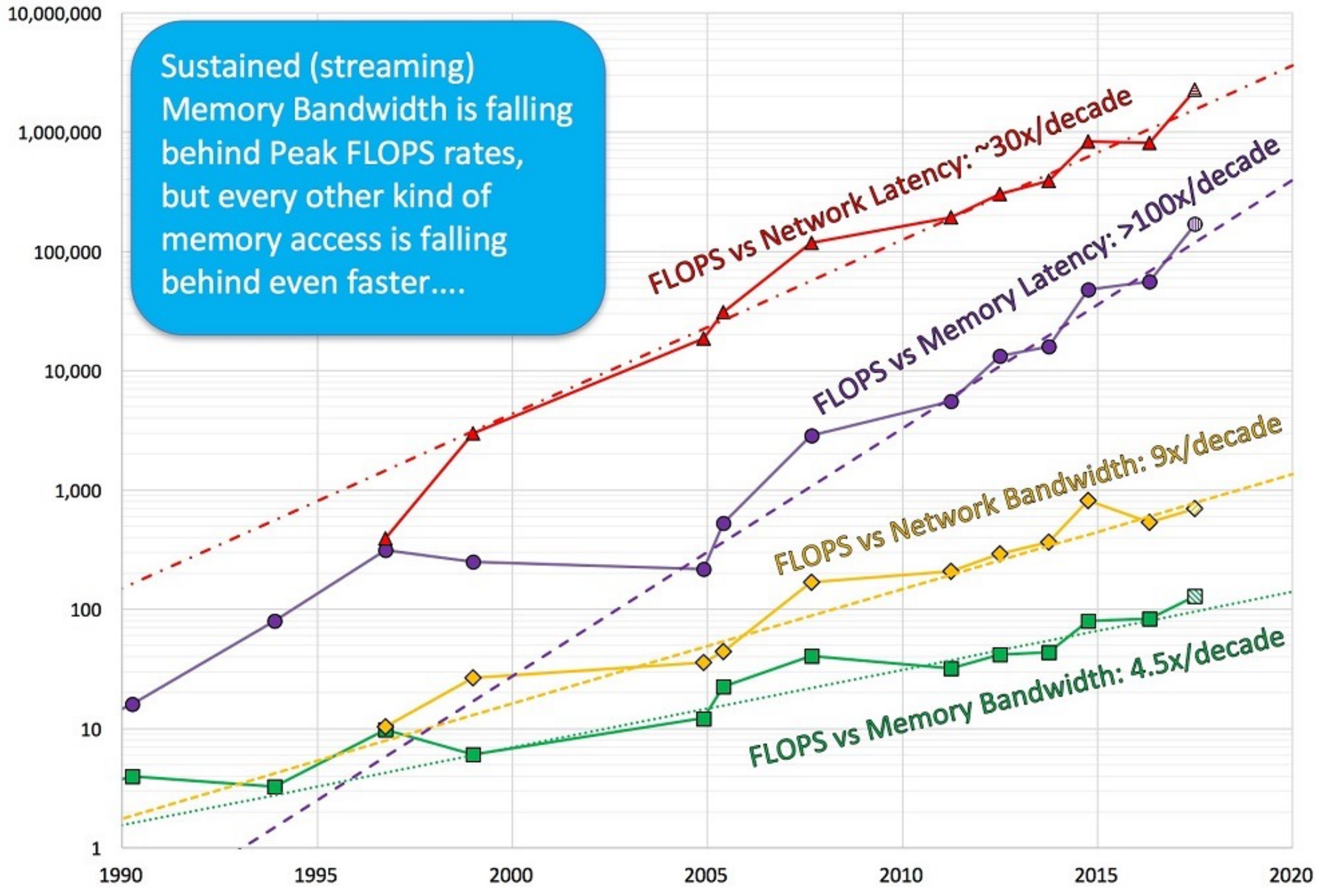

Goran Flegar


Fritz Göbel


T. Grützmacher


E. Quintana-Orti

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.



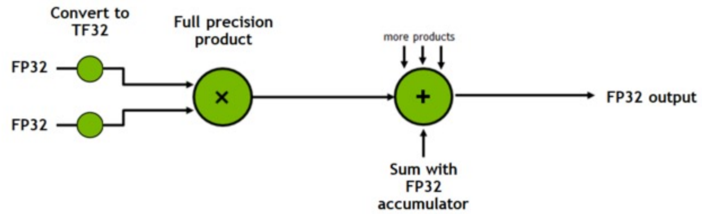
Trends in the relative performance of floating-point arithmetic and several classes of data access for select HPC servers over the past 25 years. Source: John McCalpin

NVIDIA

Form Factor	H100 SXM
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core	989 teraFLOPS[2]
BFLOAT16 Tensor Core	1,979 teraFLOPS[2]
FP16 Tensor Core	1,979 teraFLOPS[2]
FP8 Tensor Core	3,958 teraFLOPS[2]
INT8 Tensor Core	3,958 TOPS[2]
GPU memory	80GB
GPU memory bandwidth	3.35TB/s

2x

14x



AMD

PERFORMANCE	MI250
Compute Units	208CU
Stream Processors	13,312
Peak FP64/FP32 Vector	45.3 TFLOPS
Peak FP64/FP32 Matrix	90.5 TFLOPS
Peak FP16/BF16	362.1 TFLOPS
Peak INT4/INT8	362.1 TOPS

2x

MEMORY	
Memory Size	128GB HBM2e
Memory Interface	8,192 bits
Memory Clock	1.6GHz
Memory Bandwidth	up to 3.2TB/sec[2]

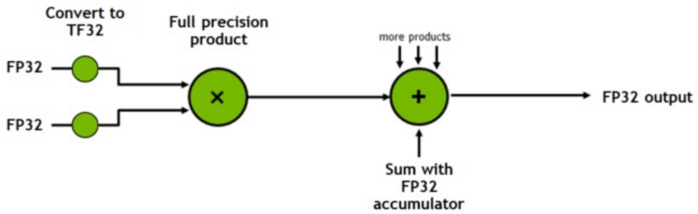
NVIDIA

Form Factor	H100 SXM
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core	989 teraFLOPS[2]
BFLOAT16 Tensor Core	1,979 teraFLOPS[2]
FP16 Tensor Core	1,979 teraFLOPS[2]
FP8 Tensor Core	3,958 teraFLOPS[2]
INT8 Tensor Core	3,958 TOPS[2]
GPU memory	80GB
GPU memory bandwidth	3.35TB/s

2x

14x

2x



AMD

PERFORMANCE	MI250
Compute Units	208CU
Stream Processors	13,312
Peak FP64/FP32 Vector	45.3 TFLOPS
Peak FP64/FP32 Matrix	90.5 TFLOPS
Peak FP16/BF16	362.1 TFLOPS
Peak INT4/INT8	362.1 TOPS

2x

4x

MEMORY	
Memory Size	128GB HBM2e
Memory Interface	8,192 bits
Memory Clock	1.6GHz
Memory Bandwidth	up to 3.2TB/sec[2]

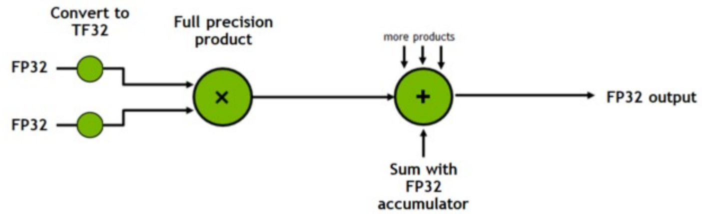
NVIDIA

Form Factor	H100 SXM
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core	989 teraFLOPS[2]
BFLOAT16 Tensor Core	1,979 teraFLOPS[2]
FP16 Tensor Core	1,979 teraFLOPS[2]
FP8 Tensor Core	3,958 teraFLOPS[2]
INT8 Tensor Core	3,958 TOPS[2]
GPU memory	80GB
GPU memory bandwidth	3.35TB/s

2x

14x

2x



AMD

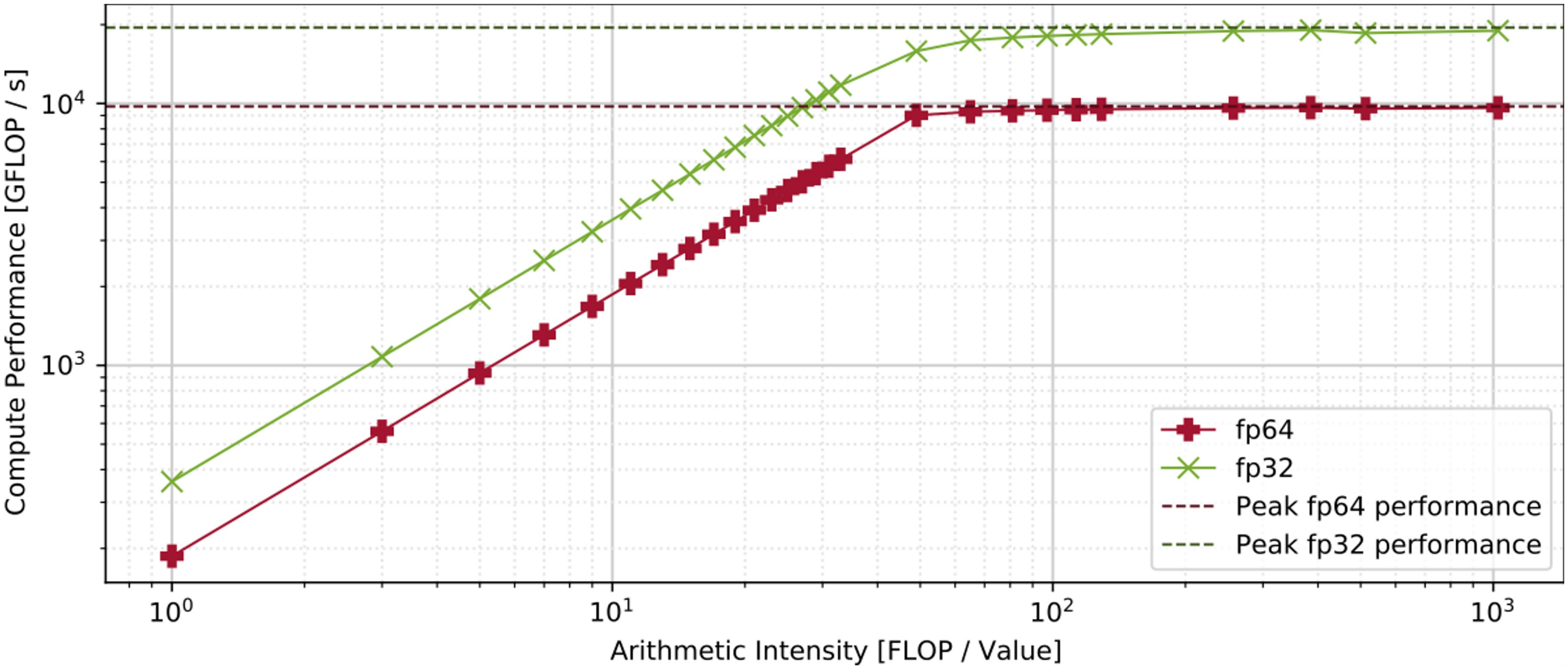
PERFORMANCE	MI250
Compute Units	208CU
Stream Processors	13,312
Peak FP64/FP32 Vector	45.3 TFLOPS
Peak FP64/FP32 Matrix	90.5 TFLOPS
Peak FP16/BF16	362.1 TFLOPS
Peak INT4/INT8	362.1 TOPS

MEMORY	
Memory Size	128GB HBM2e
Memory Interface	8,192 bits
Memory Clock	1.6GHz
Memory Bandwidth	up to 3.2TB/sec[2]

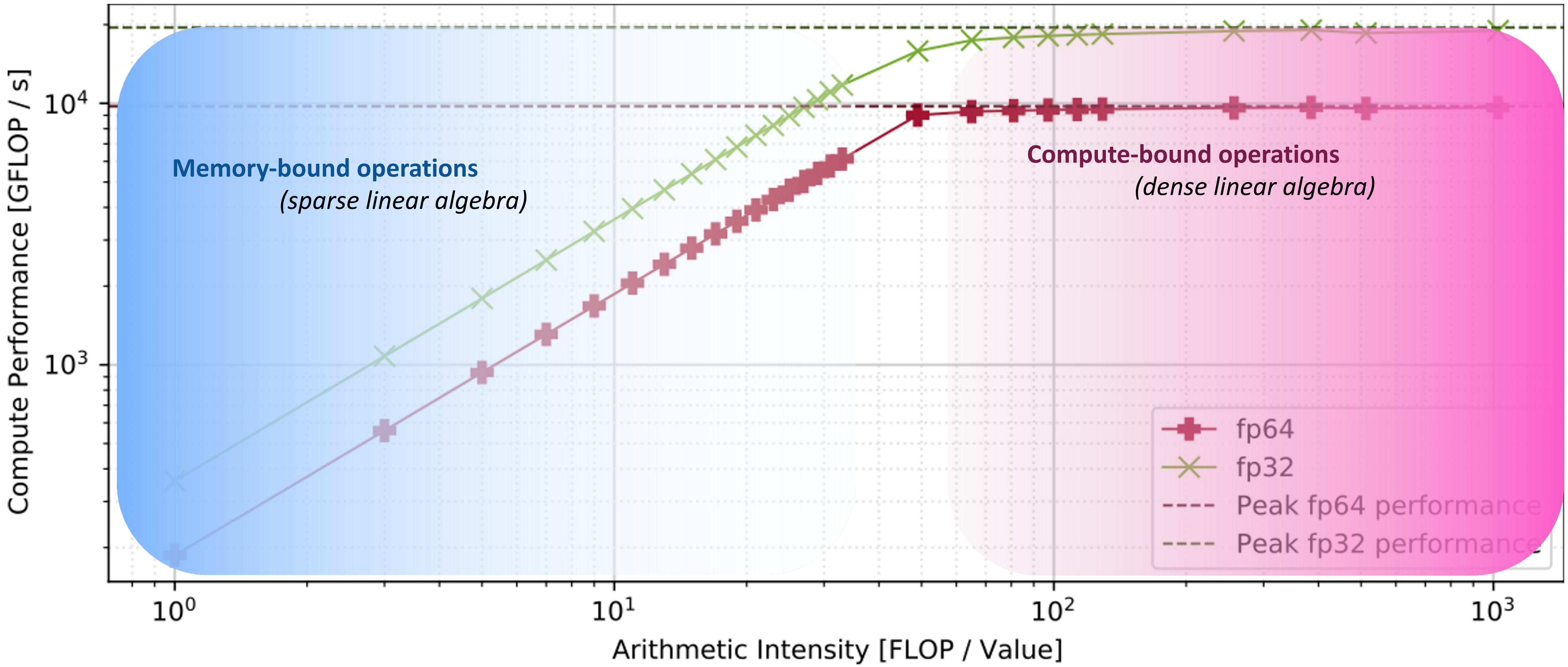
2x

4x

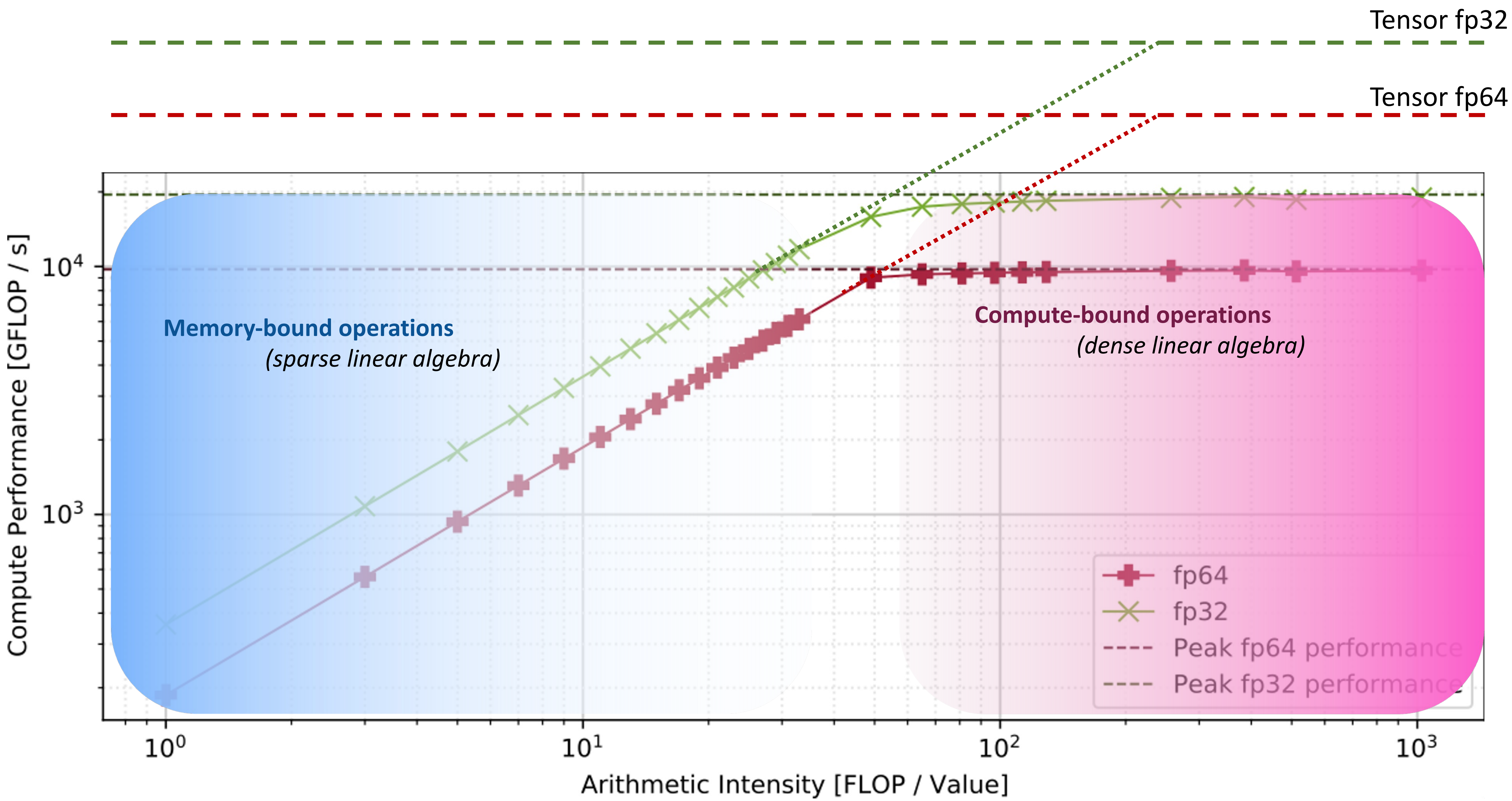
- (Dense) Matrix Performance >> Vector Operation Performance
- Low Precision Performance >> High Precision Performance



Compute Performance [GFLOP / s]
10^4
10^3
10^0
10^1
10^2
10^3
Arithmetic Intensity [FLOP / Value]
fp64
fp32
Peak fp64 performance
Peak fp32 performance



Memory-bound operations
(sparse linear algebra)
Compute-bound operations
(dense linear algebra)
fp64
fp32
Peak fp64 performance
Peak fp32 performance
Compute Performance [GFLOP / s]
10^4
10^3
10^0
10^1
10^2
10^3
Arithmetic Intensity [FLOP / Value]

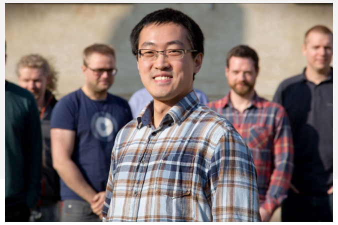


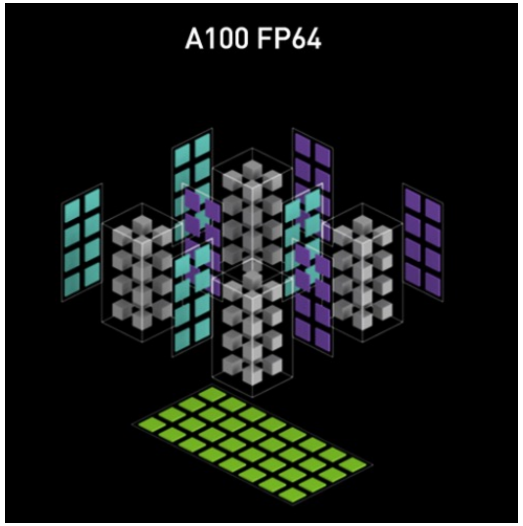
Tensor fp32
Tensor fp64
Compute Performance [GFLOP / s]
10^4
10^3
Memory-bound operations
(sparse linear algebra)
Compute-bound operations
(dense linear algebra)
fp64
fp32
Peak fp64 performance
Peak fp32 performance
10^0
10^1
10^2
10^3
Arithmetic Intensity [FLOP / Value]

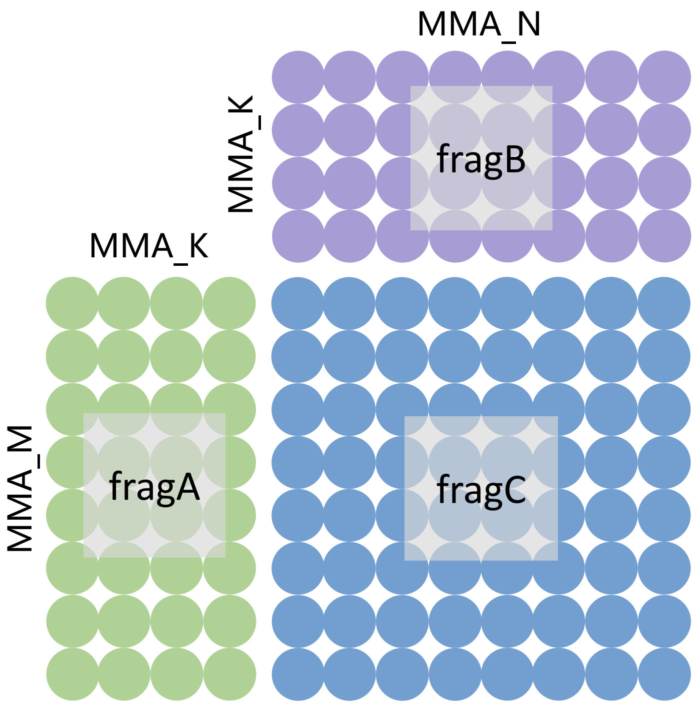
Presentations: https://dl.acm.org/doi/pdf/10.1145/3581784.3607051
10:30am - 11am MST

DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication

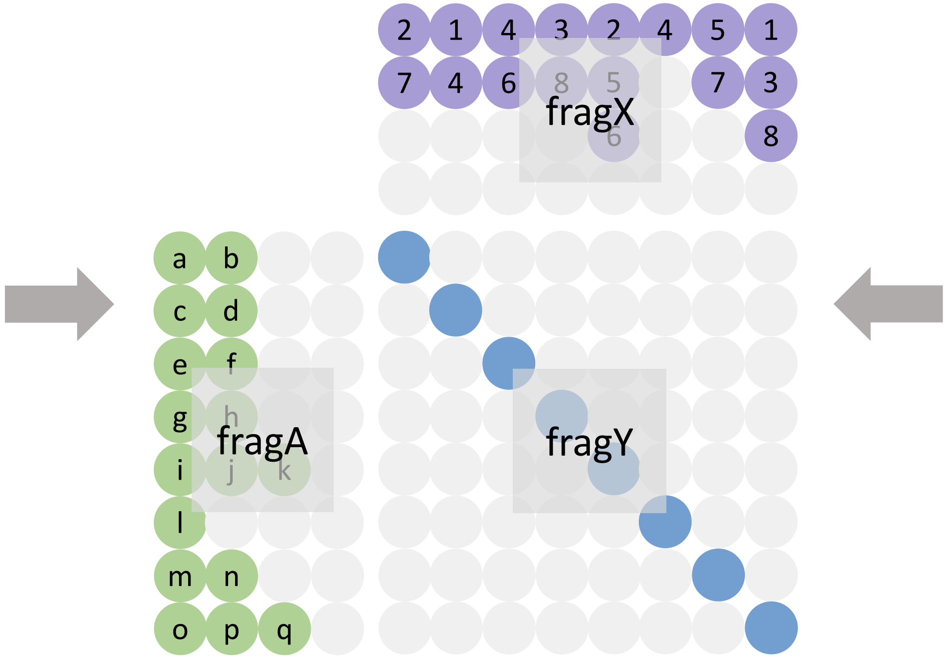
Authors: Yuechen Lu, Weifeng Liu



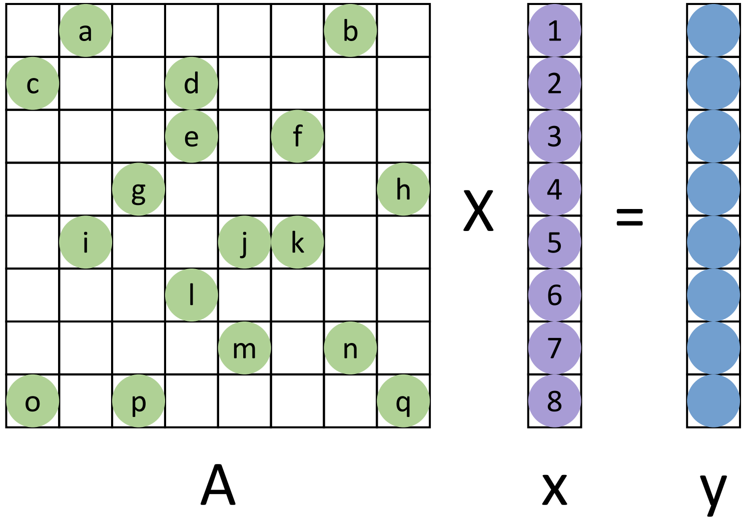




Compute layout of the double precision mma_m8n8k4 instruction

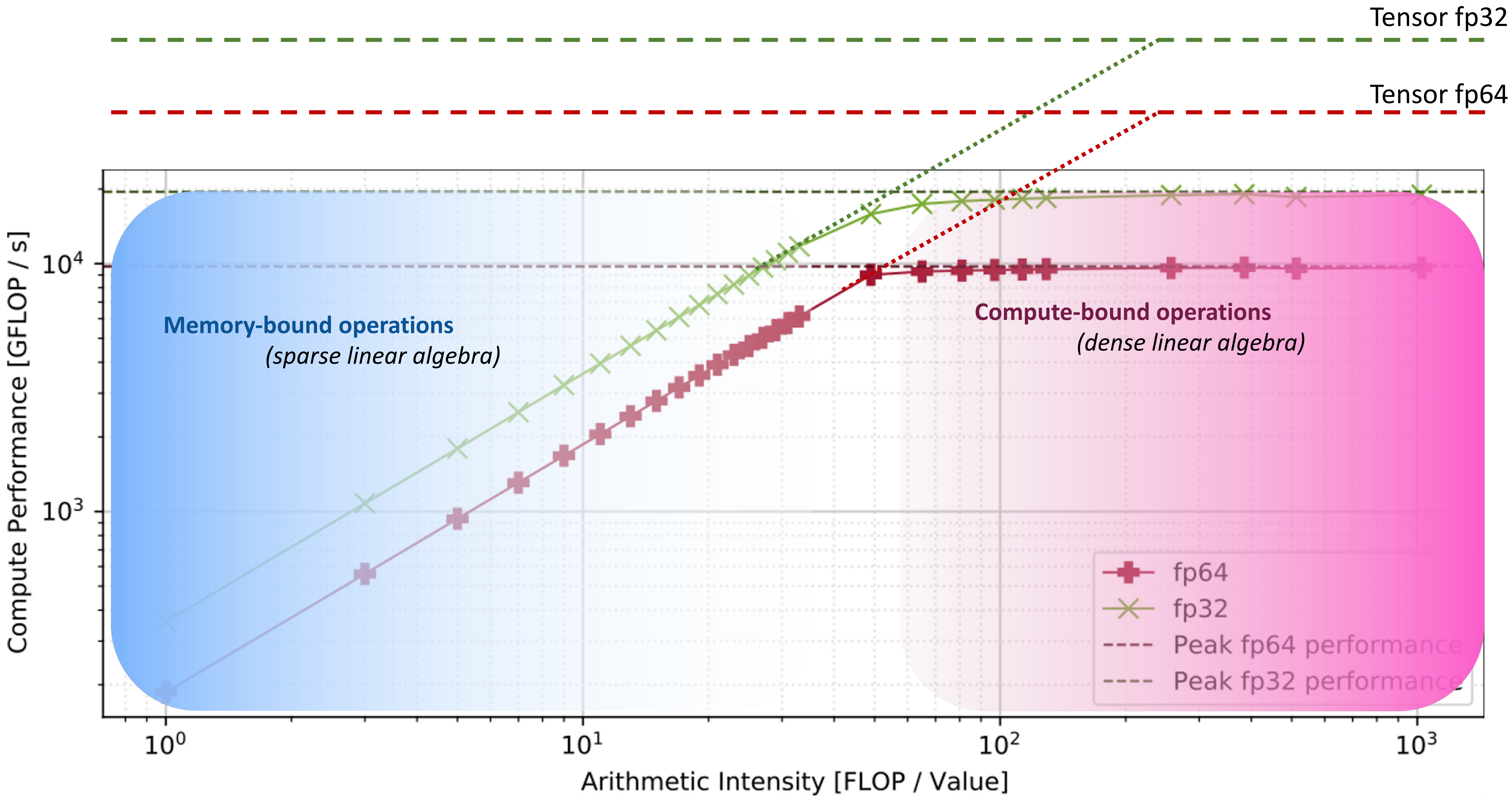


Compute 8x8 SpMV using mma instruction

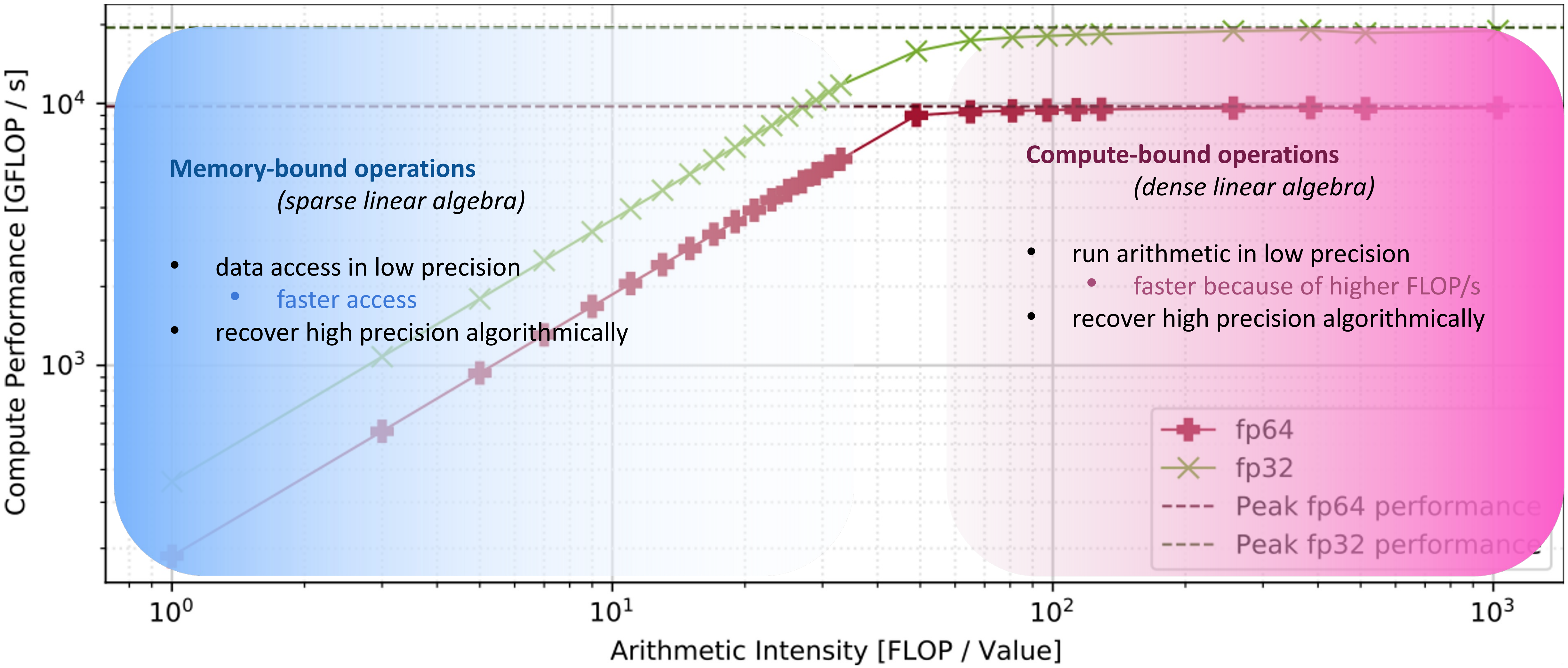


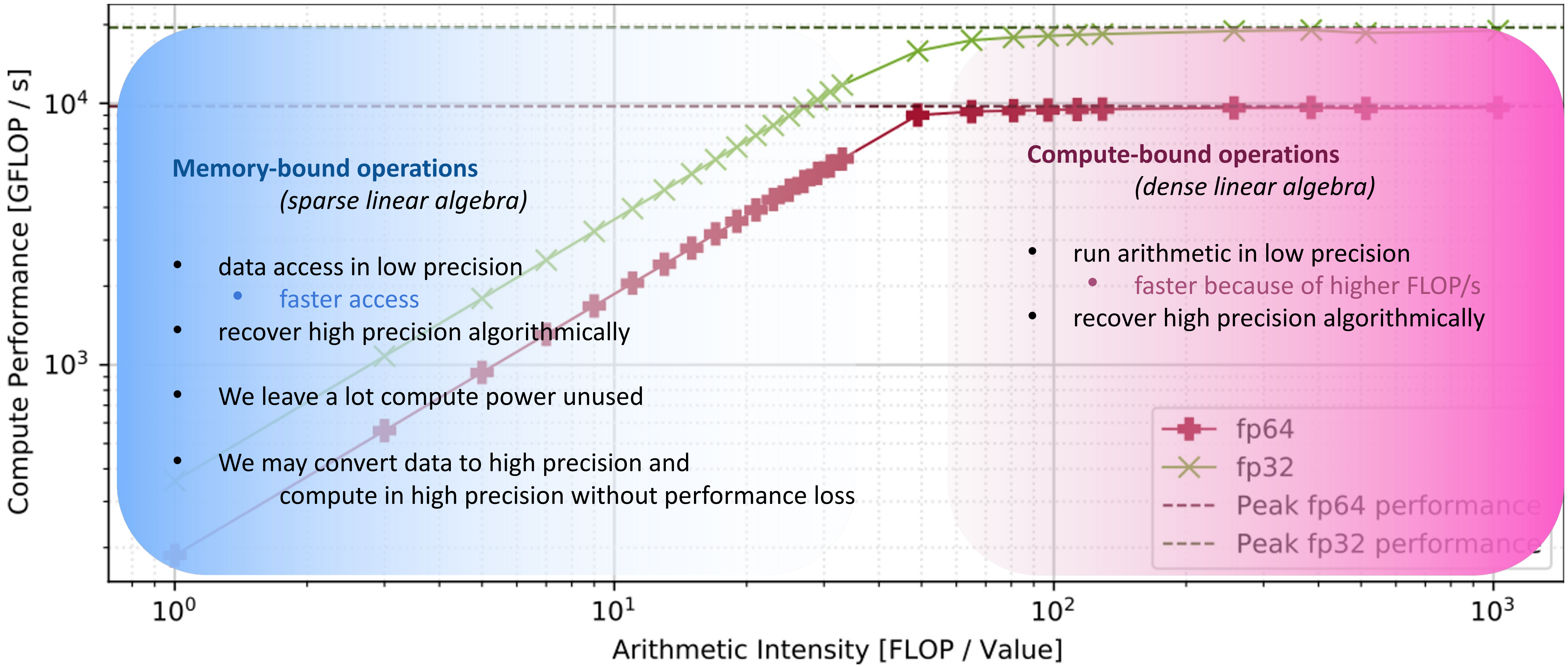
Example of SpMV for a matrix A of size 8x8

Yuechen Lu



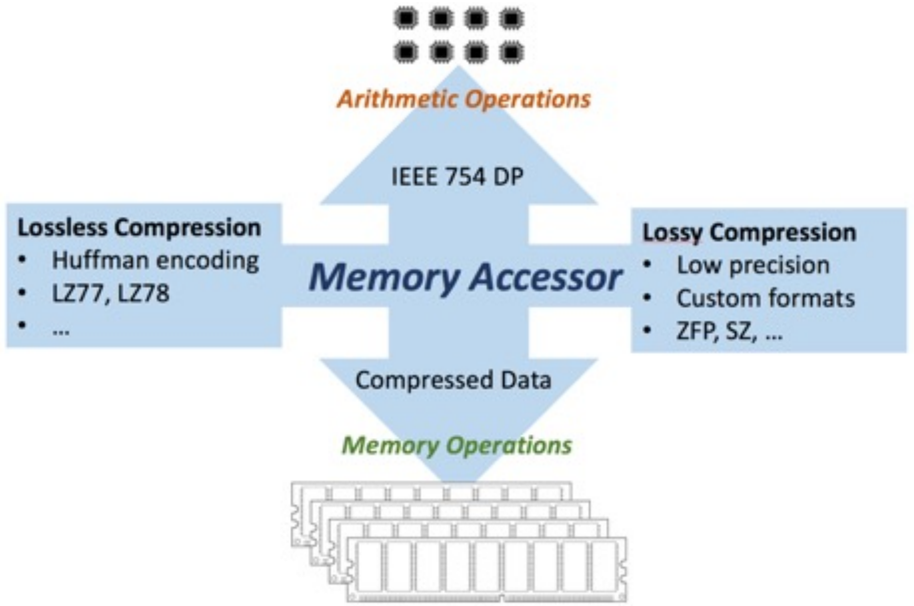
Tensor fp32
Tensor fp64
Memory-bound operations
(sparse linear algebra)
Compute-bound operations
(dense linear algebra)
Compute Performance [GFLOP / s]
10^4
10^3
10^0
10^1
10^2
10^3
Arithmetic Intensity [FLOP / Value]
fp64
fp32
Peak fp64 performance
Peak fp32 performance


Memory-bound operations
(sparse linear algebra)
data access in low precision
faster access
recover high precision algorithmically
Compute-bound operations
(dense linear algebra)
run arithmetic in low precision
faster because of higher FLOP/s
recover high precision algorithmically
fp64
fp32
Peak fp64 performance
Peak fp32 performance
Compute Performance [GFLOP / s]
Arithmetic Intensity [FLOP / Value]
10^4
10^3
10^0
10^1
10^2
10^3



Memory-bound operations
(sparse linear algebra)
• data access in low precision
• faster access
• recover high precision algorithmically
• We leave a lot compute power unused
• We may convert data to high precision and compute in high precision without performance loss
Compute-bound operations
(dense linear algebra)
• run arithmetic in low precision
• faster because of higher FLOP/s
• recover high precision algorithmically
fp64
fp32
Peak fp64 performance
Peak fp32 performance
Compute Performance [GFLOP / s]
Arithmetic Intensity [FLOP / Value]
10^4
10^3
10^0
10^1
10^2
10^3

- Traditionally, we use a strong coupling between the precision formats used for arithmetic operations and storing data.
- *Maybe this is not the right thing?*
- *We should compute in fp64*
- *Maybe we should use the free compute cycles (vector/tensor cores) to compress the data*



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

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

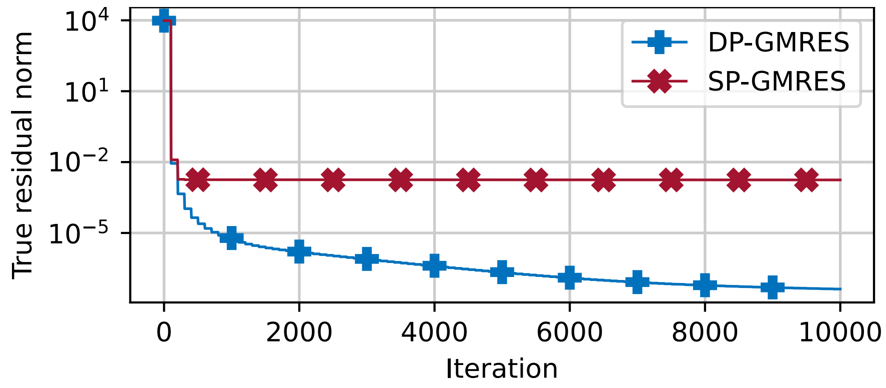
```
Double precision GMRES
Initial residual norm
sqrt(r^t r): 9670.36
Final residual norm
sqrt(r^T r): 9.6639e-09
GMRES iteration count: 23271
GMRES execution time: 43801 ms
```

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!

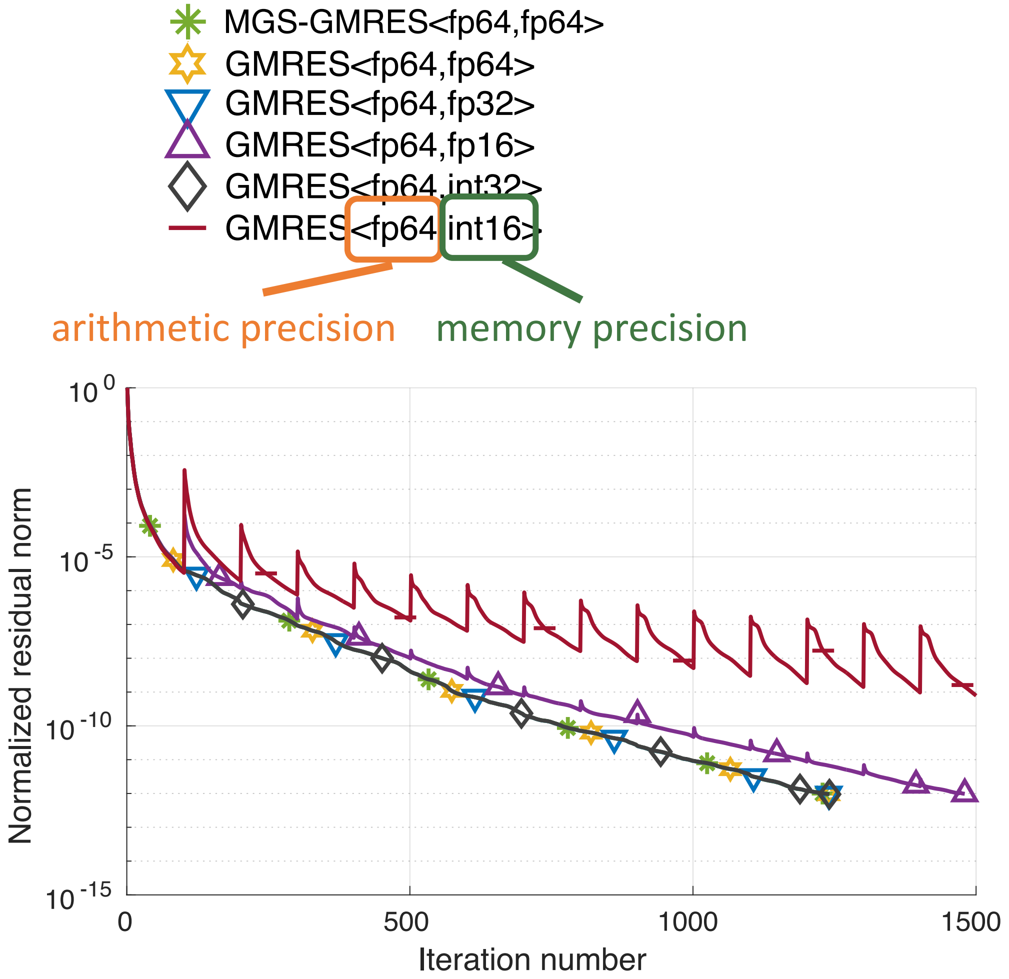


forward error ≈ (unit round-off) * (linear system's condition number)

N. Higham: Accuracy and stability of numerical algorithms. SIAM, 2002.

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;



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
```

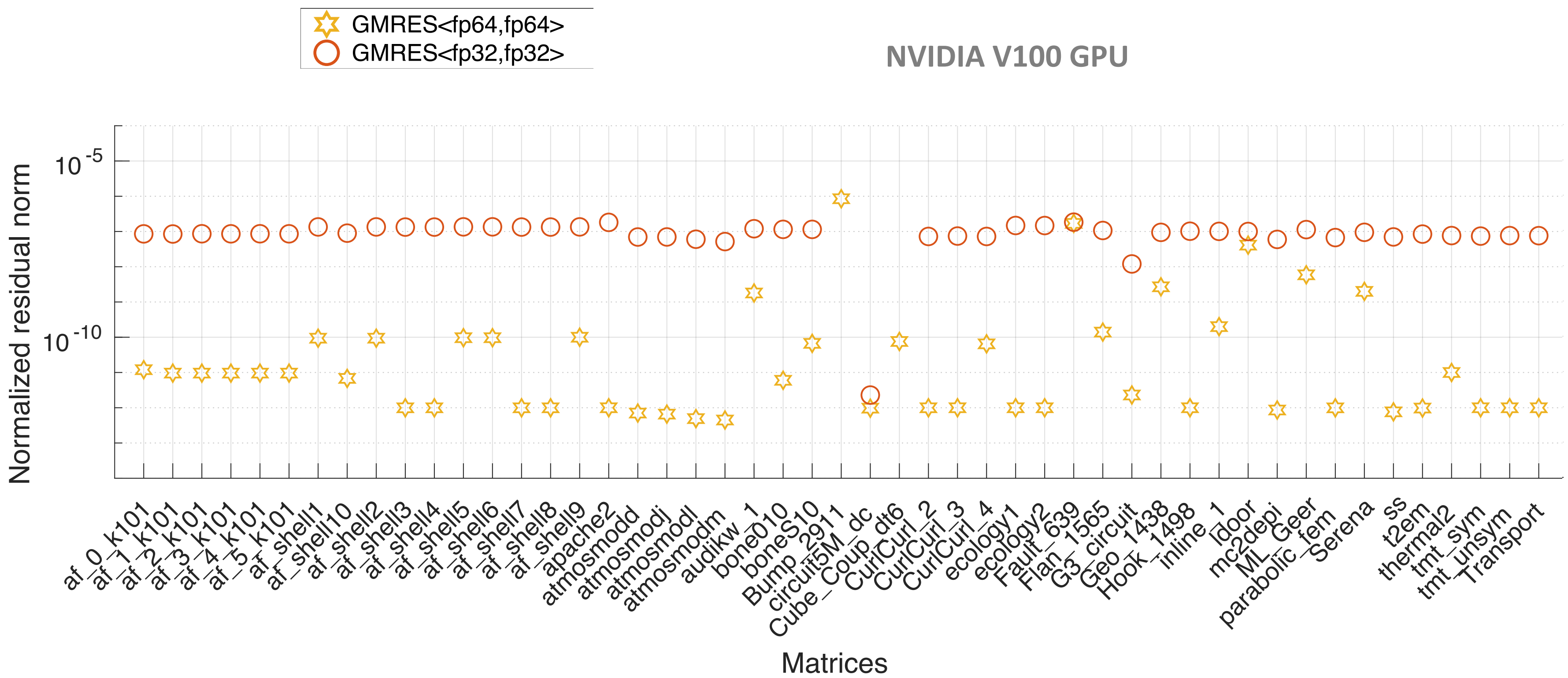
Relative residual ~10^{-7}

```
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
```

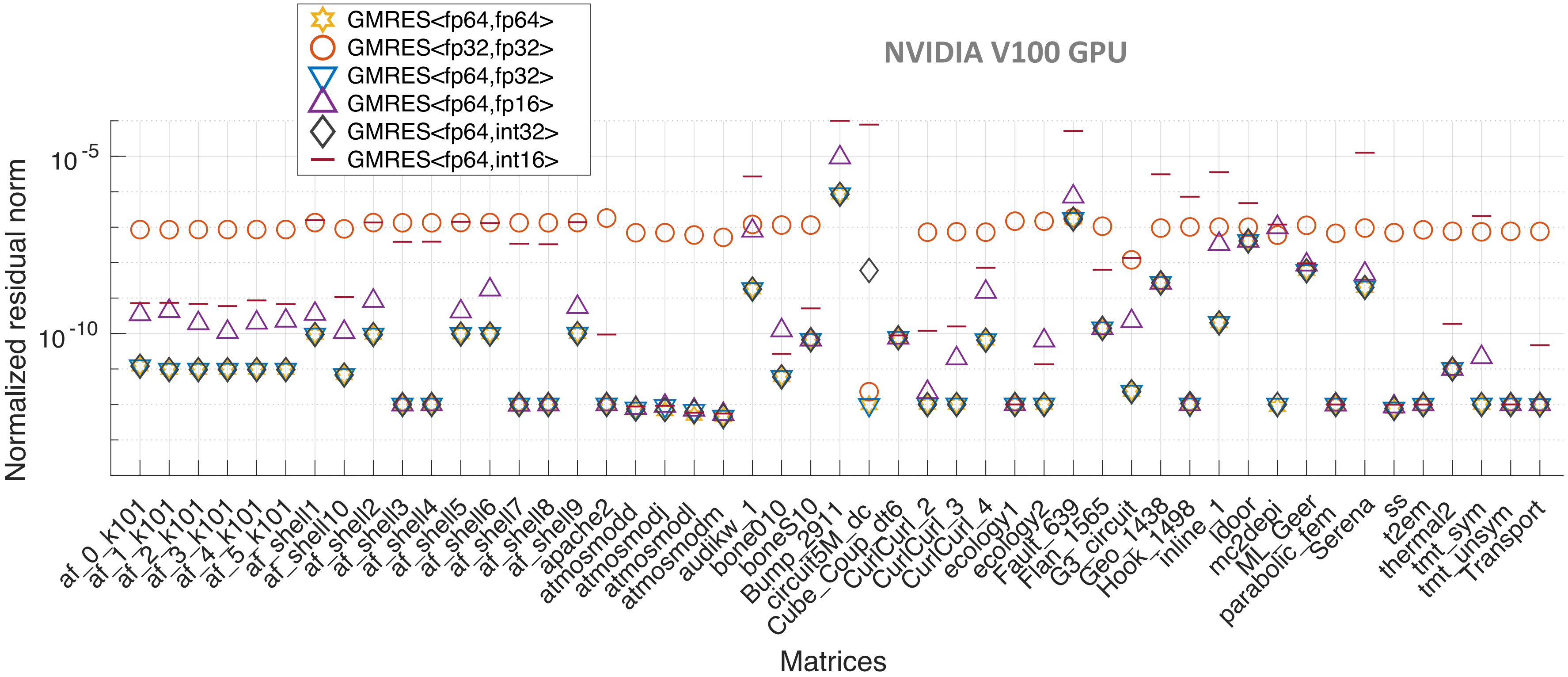
Relative residual ~10^{-12}

Accuracy of DP GMRES

Performance similar to SP GMRES


GMRES<fp64,fp64>
GMRES<fp32,fp32>
NVIDIA V100 GPU
Normalized residual norm
10^{-5}
10^{-10}
af_0_k101
af_1_k101
af_2_k101
af_3_k101
af_4_k101
af_5_k101
af_shell1
af_shell10
af_shell2
af_shell3
af_shell4
af_shell5
af_shell6
af_shell7
af_shell8
af_shell9
apache2
atmosmodd
atmosmodj
atmosmodl
atmosmodm
audikw_1
bone010
boneS10
Bump_2911
circuit5M_dc
Cube_Coup_dt6
CurlCurl_2
CurlCurl_3
CurlCurl_4
ecology1
ecology2
Fault_639
Flan_1565
G3_circuit
Geo_1438
Hook_1498
inline_1
ldoor
mc2depi
ML_Geer
parabolic_fem
Serena
ss
t2em
thermal2
tmt_sym
tmt_unsym
Transport
Matrices

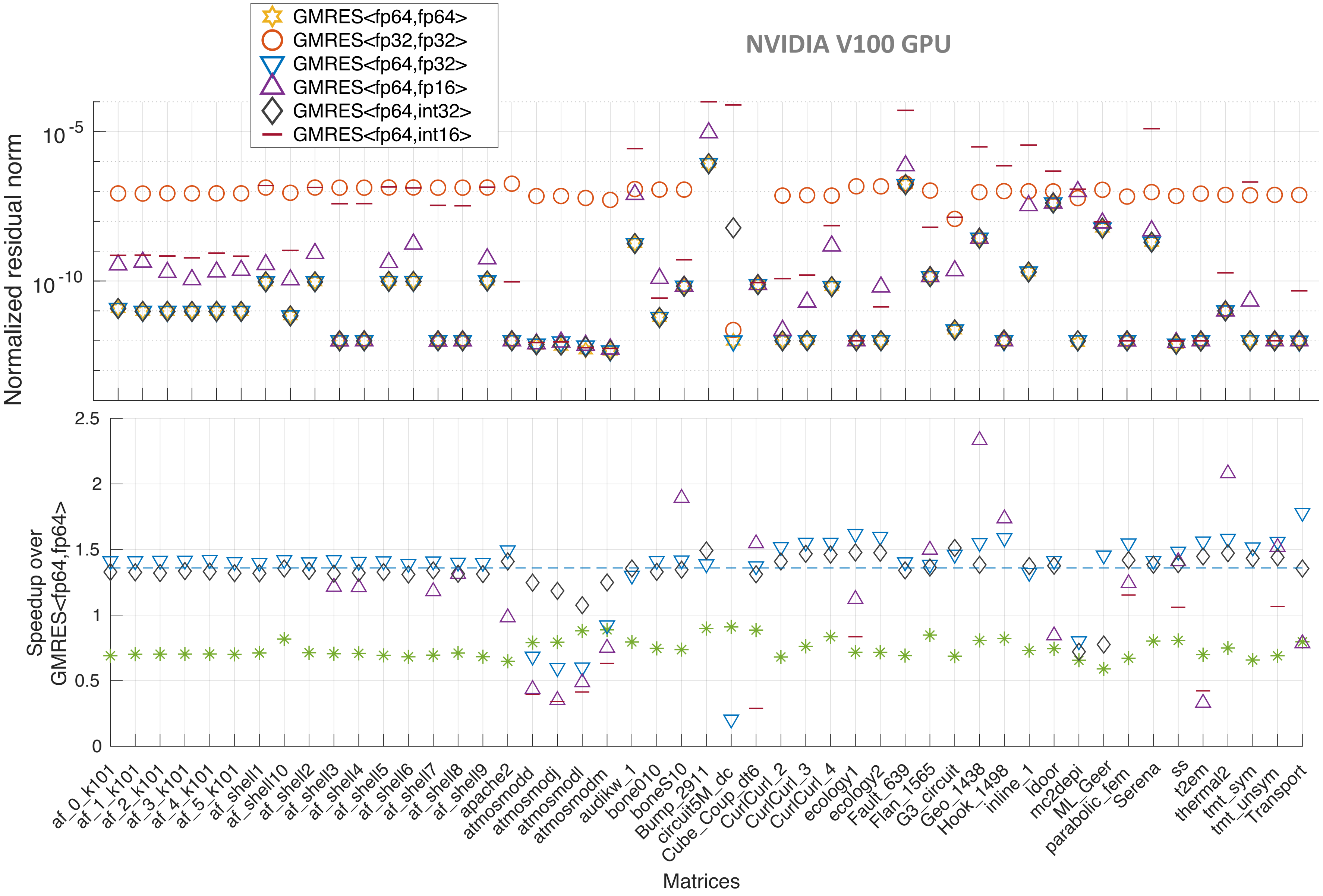
- CB-GMRES using 32-bit storage preserves DP accuracy (SP-GMRES does not)



- 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

Aliaga JI, Anzt H, Grützmacher T, Quintana-Ortí ES, Tomás AE. Compressed basis GMRES on high-performance graphics processing units. *The International Journal of High Performance Computing Applications*. 2022;0(0). doi:[10.1177/10943420221115140](https://doi.org/10.1177/10943420221115140)



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:

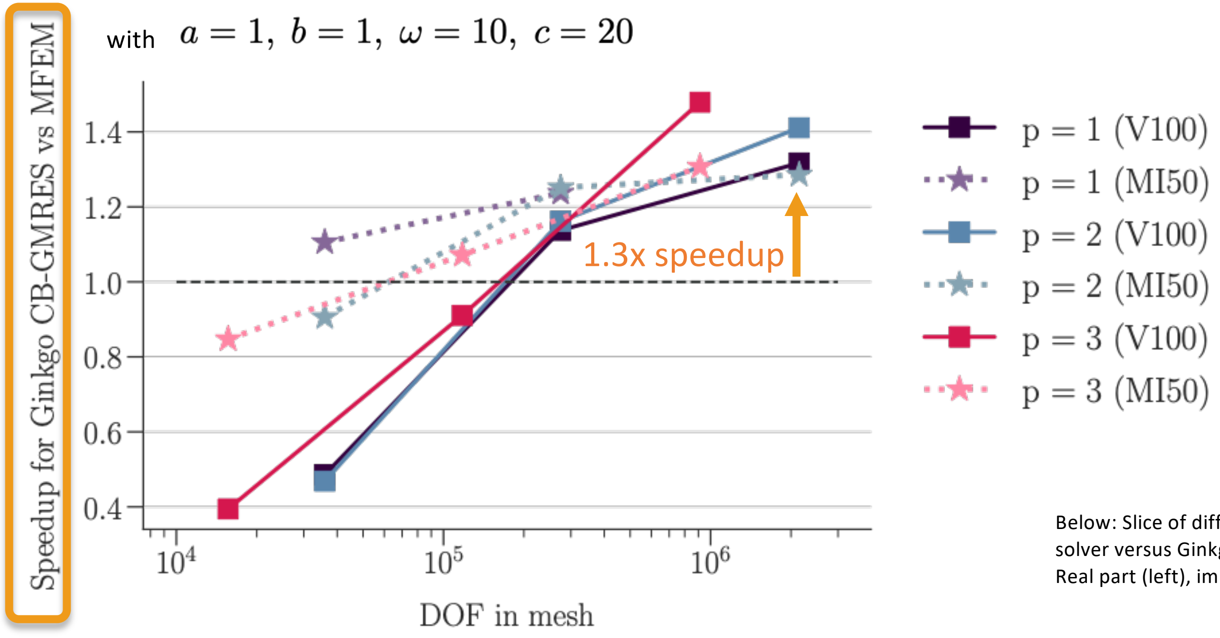
- ✓ Integration for Ginkgo's Compressed Basis GMRES solver, which uses mixed precision techniques for speedup (see example to right)

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:

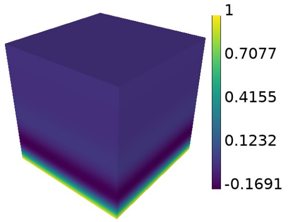
$$-\nabla \cdot (a \nabla u) - \omega^2 bu + i\omega cu = 0$$

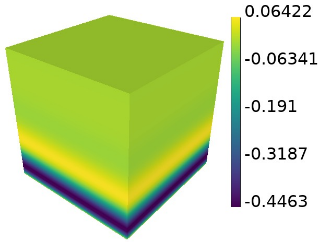
with $a = 1,\ b = 1,\ \omega = 10,\ c = 20$



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.

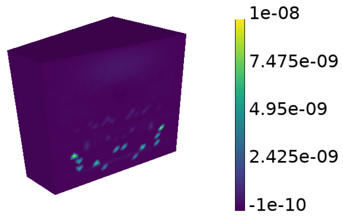
CUDA 10.1/NVIDIA V100 and ROCm 4.0/AMD MI50.
GMRES(50) used for both solvers. CB-GMRES used float/double.

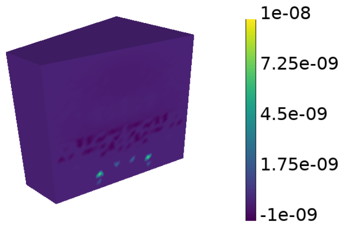




From top: Real part of solution, imaginary part of solution.

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

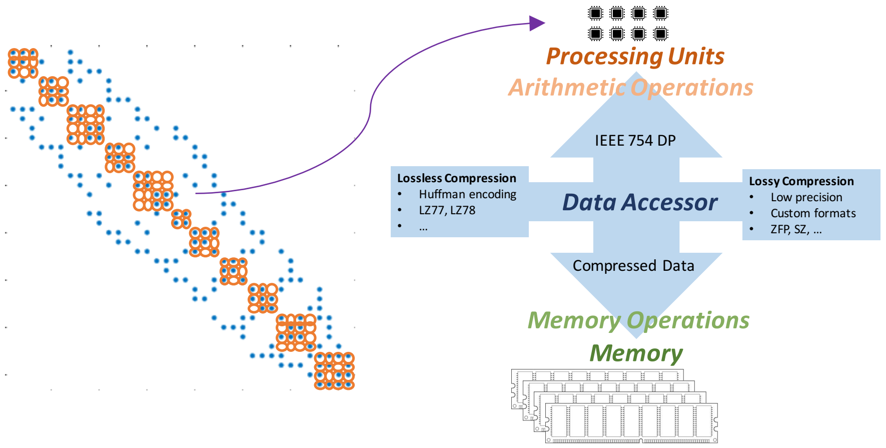




- **Preconditioning iterative solvers.**
  - Idea: Approximate inverse of system matrix to make the system "easier to solve": $P^{-1} \approx A^{-1}$

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

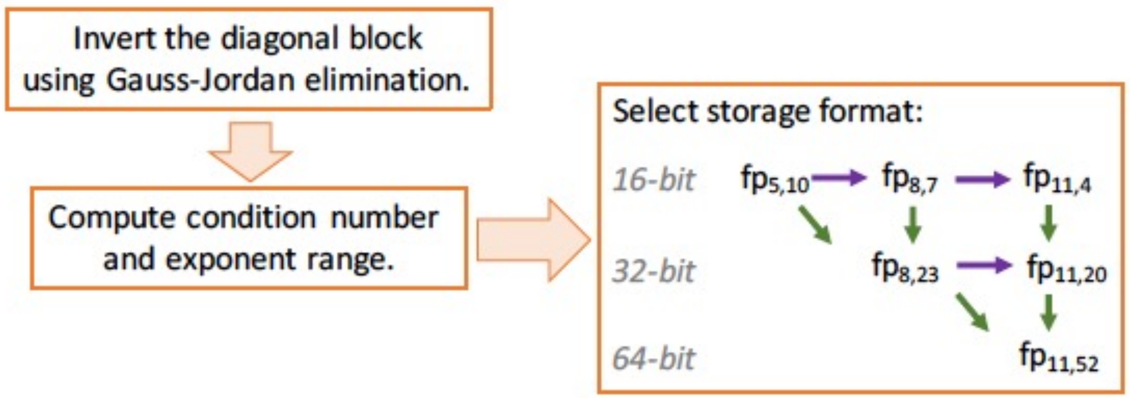
- **Block-Jacobi preconditioner** is based on **block-diagonal scaling**: $P = diag_B(A)$
  - Each block corresponds to one (small) linear system.
    - *Larger* blocks typically improve convergence.
    - *Larger* blocks make block-Jacobi more expensive.



- *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!*

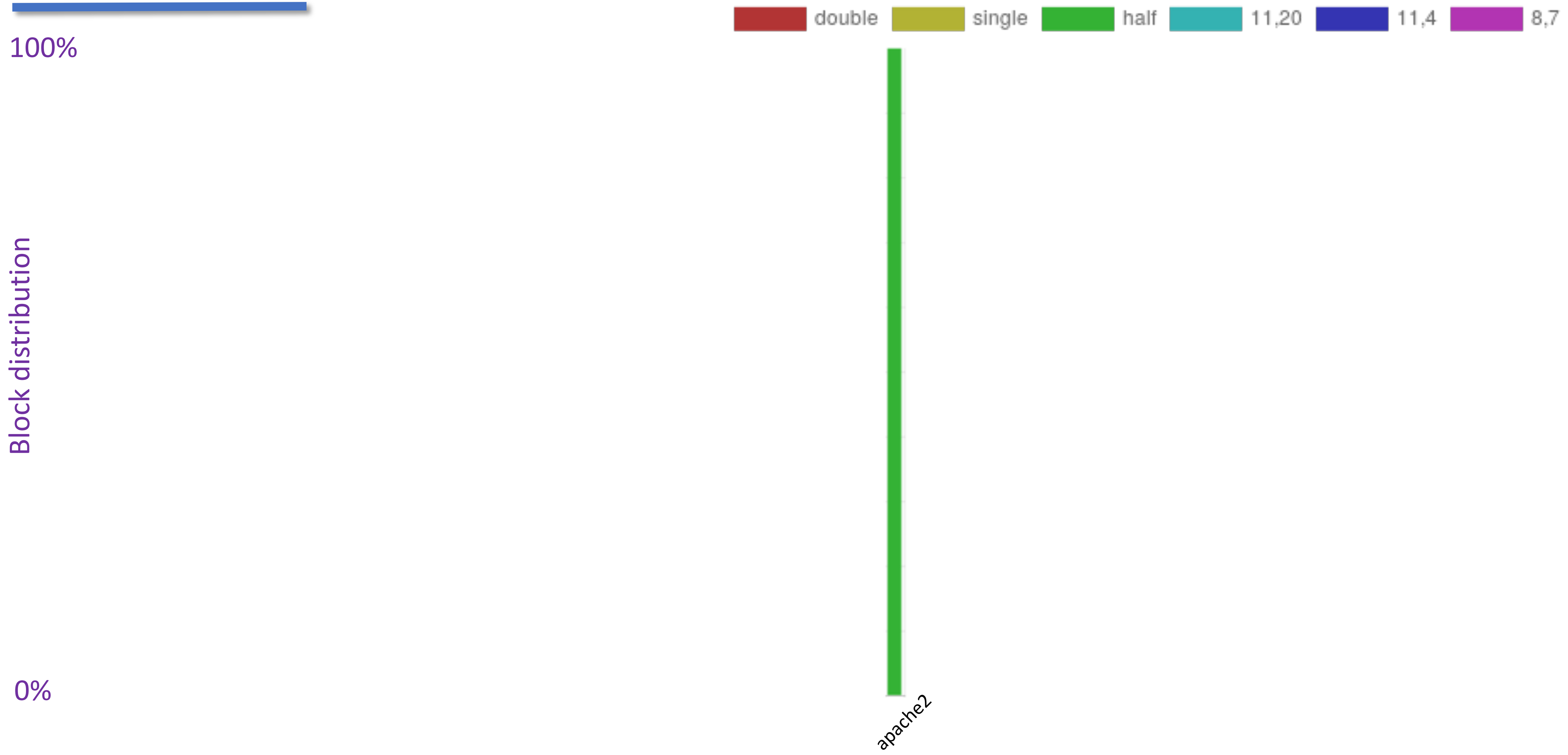
- Choose how much accuracy of the preconditioner should be preserved in the selection of the storage format.
- All computations use double precision, but store blocks in lower precision.



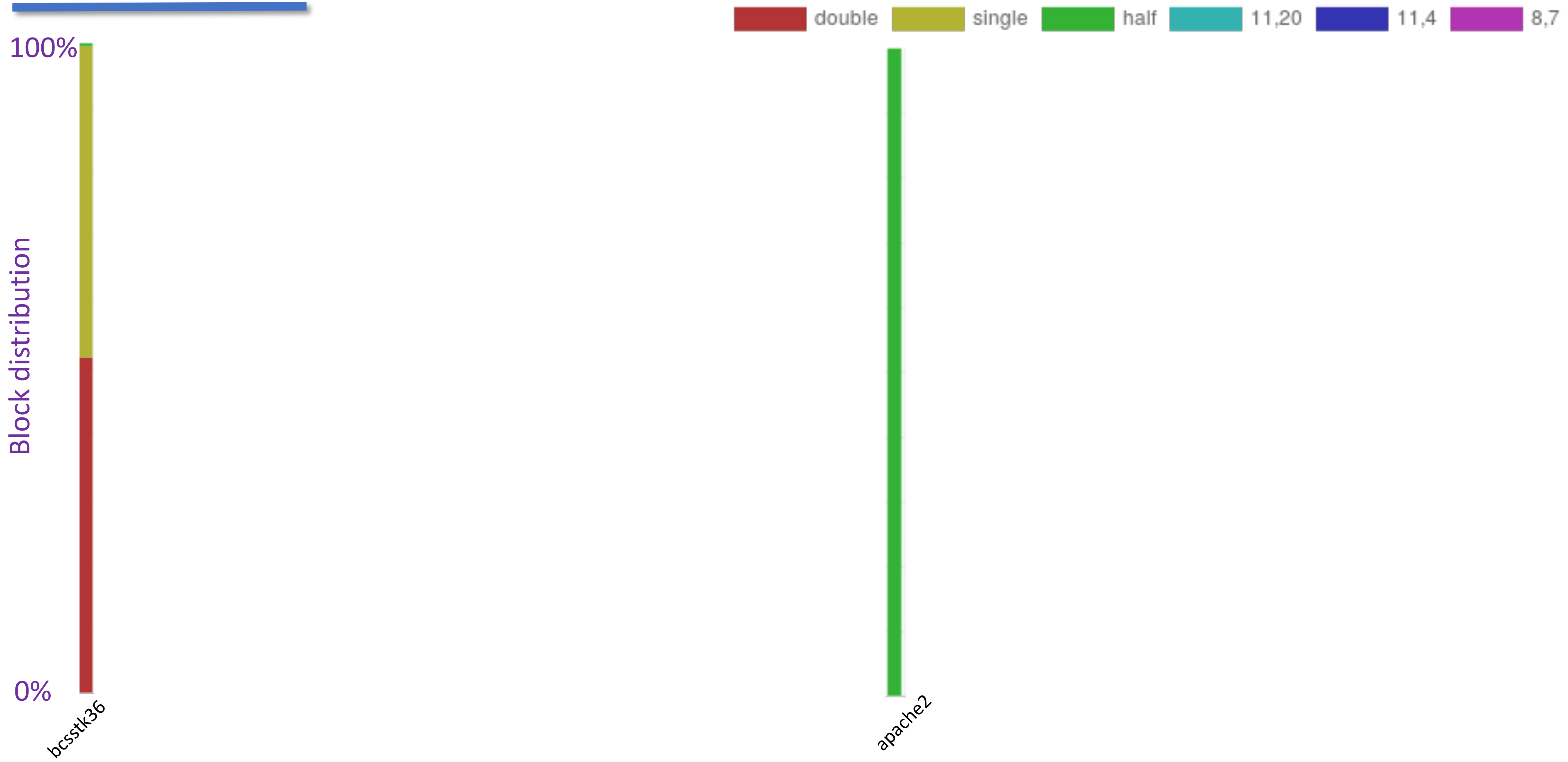
+ **Regularity preserved;**
+ Flexibility in the accuracy;
+ ”Not a low precision preconditioner”
  + Preconditioner is a constant operator;
  + No flexible Krylov solver needed ;

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

Mixed Precision Preconditioning



Mixed Precision Preconditioning



Mixed Precision Preconditioning

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

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
```

Double Precision CG + **Mixed Precision Preconditioner**

We hope that:

- *Attainable accuracy of CG unaffected*
- *Preconditioner remains a constant operator*

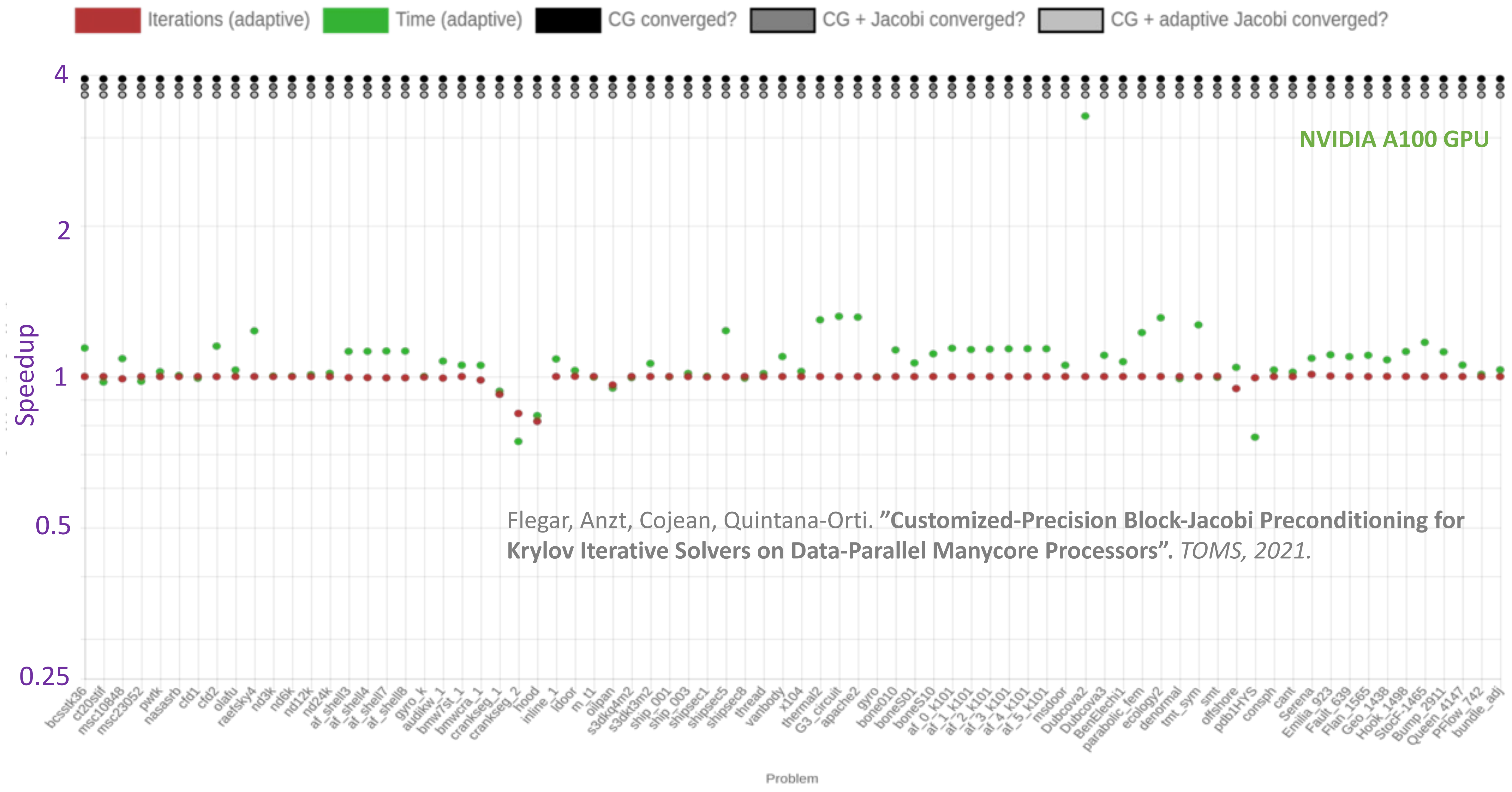
Mixed Precision Preconditioning

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

Double Precision CG + Double Precision Preconditioner	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 3.97985e-06 CG iteration count: 4797 CG execution time [ms]: 2971.18	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



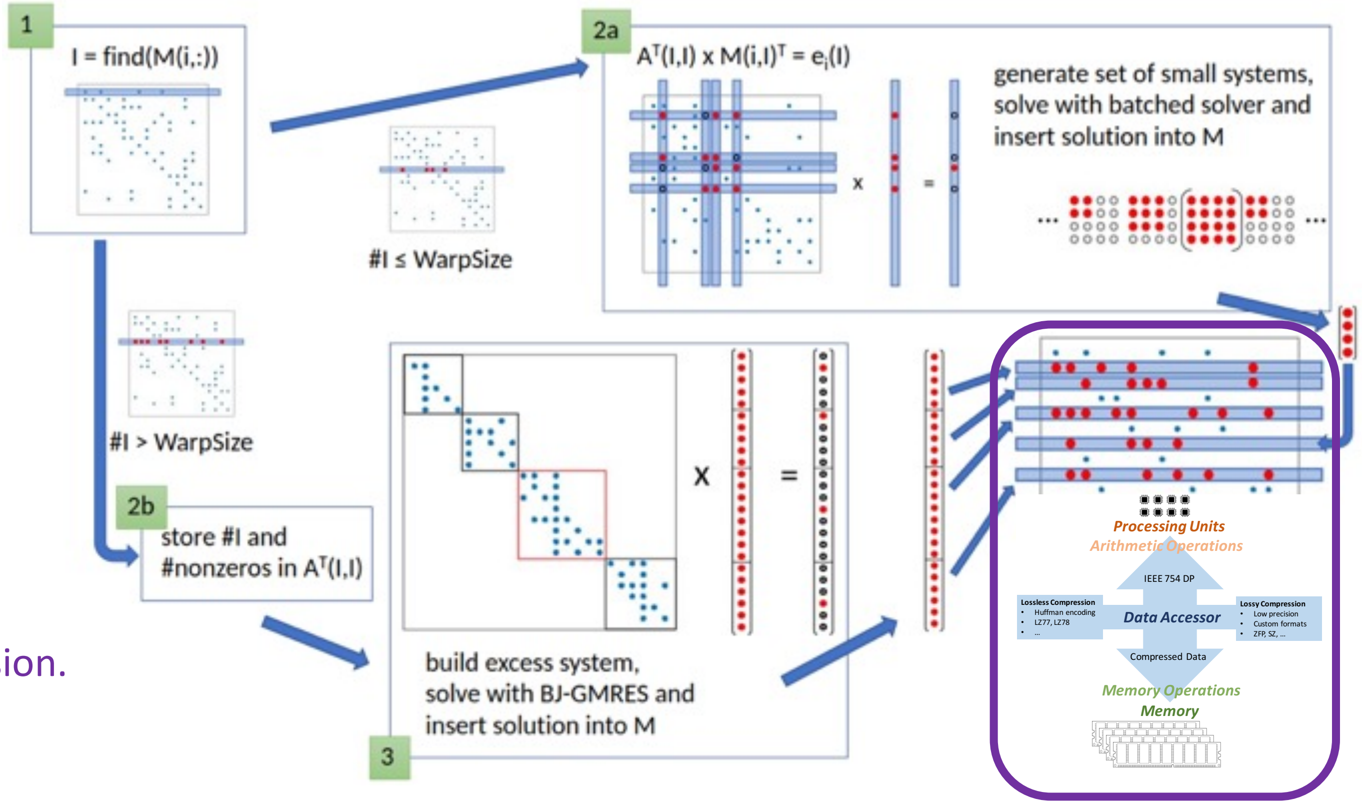
Flegar, Anzt, Cojean, Quintana-Orti. **"Customized-Precision Block-Jacobi Preconditioning for Krylov Iterative Solvers on Data-Parallel Manycore Processors".** *TOMS, 2021.*

- **Sparse Approximate Inverse Preconditioner**
  - $M \approx A^{-1}$ and sparse
  - Incomplete Sparse Approximate Inverse (ISAI)

    uses sparsity pattern of A;
  - Factorized Sparse Approximate Inverse (FSPAI)

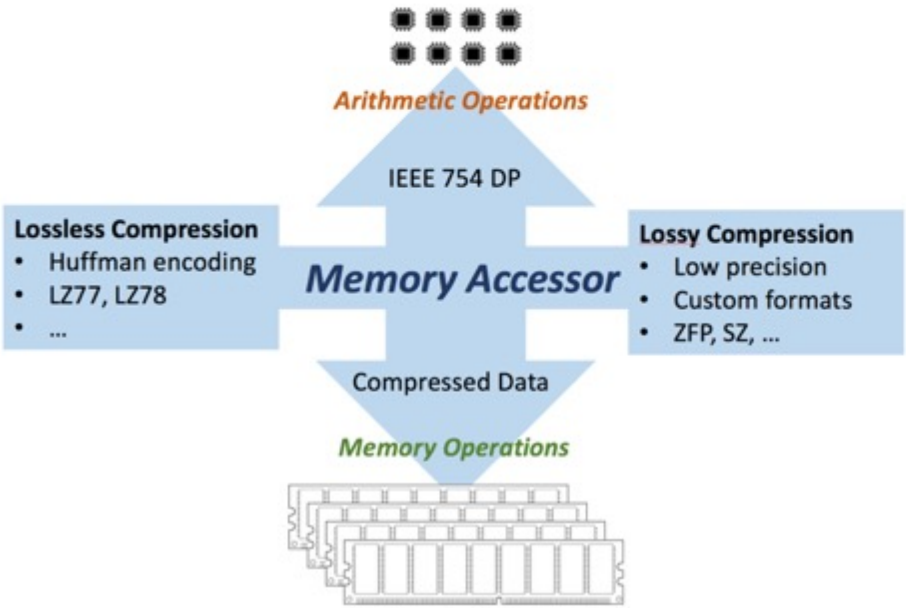
    stores inverse approximation in factorized form;

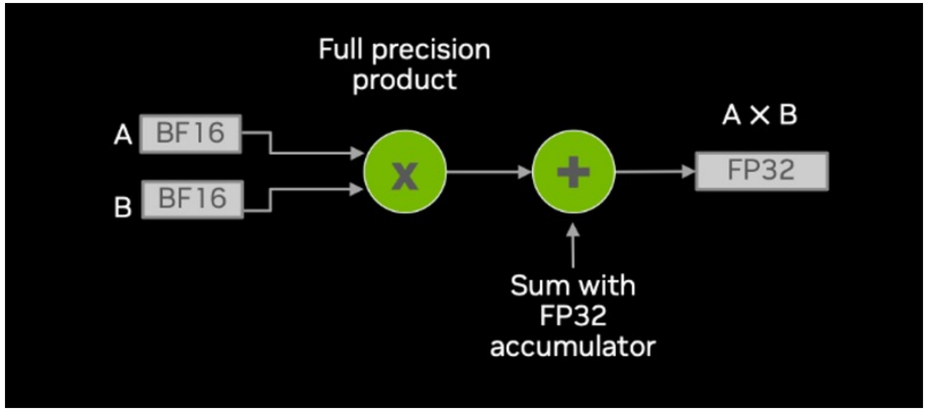
- Use the accessor to store the preconditioner in lower precision.



Göbel at al: Mixed Precision Incomplete and Factorized Sparse Approximate Inverse Preconditioning on GPUs, EuroPar 2021.

Is there any hope for sparse operations? – Yes, but we will have to work harder…





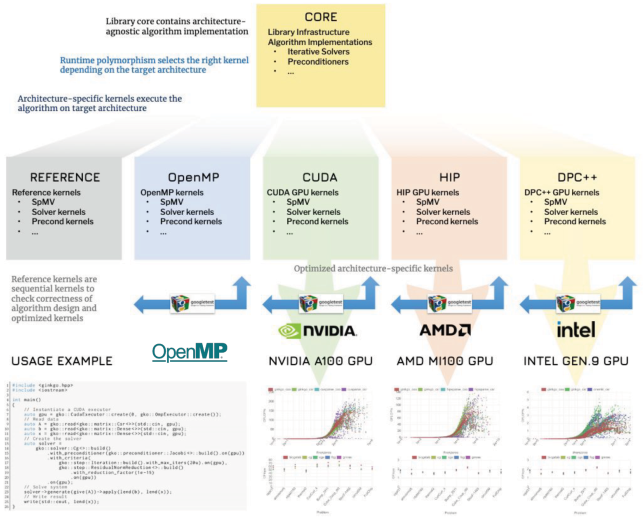
NVIDIA Tensor Core operation.

- The gap between compute performance and memory performance is widening
- Hardware increasingly features powerful matrix engines
- We should use these ”free” FLOP/s
- One strategy is to use lossy/lossless data compression for memory operations and communication
- Smart algorithms for sparse data (e.g. DASP)
- We need efficient need on-chip data compression

Ginkgo

DESIGN

Ginkgo is a C++ framework for sparse numerical linear algebra. Using a universal linear operator abstraction, Ginkgo provides basic building blocks such as the sparse matrix vector product for a variety of matrix formats, iterative solvers, and preconditioners. Ginkgo targets multi- and many-core systems, and currently features back-ends for AMD GPUs, Intel CPU/GPUs, NVIDIA GPUs, and OpenMP-supporting architectures. Core functionality is separated from hardware-specific kernels for easy extension to other architectures, with runtime polymorphism selecting the correct kernels.



SUSTAINABILITY

Ginkgo is part of the Extreme-scale Scientific Software Stack (E4S) and the extreme-scale Software Development Kit (xSDK), and adopts the xSDK community policies for sustainable software development and high software quality. The source code of the Ginkgo library can be accessed in a public git repository on GitHub. Code development in Ginkgo is realized in a Continuous Integration / Continuous Benchmarking framework. GitLab runners are used on private servers and HPC clusters where Docker/Enroot is used to provide different compilation and execution environments. To test the correct execution, each functionality is complemented by unit tests. The unit testing is realized using the Google Test framework.

Karlsruher Institut für Technologie

THE UNIVERSITY OF TENNESSEE KNOXVILLE

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 EuroHPC Project MICROCARD, and the Helmholtz Impuls und Vernetzungsfond VH-NG-1241.

SPONSORED BY

EXASCALE COMPUTING PROJECT

EuroHPC Joint Undertaking

HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

HARDWARE PARTNERS

AMD

intel

NVIDIA

INTEGRATION

deal.II

OpenFOAM

sundials

HYTEG

openCARP

Spack

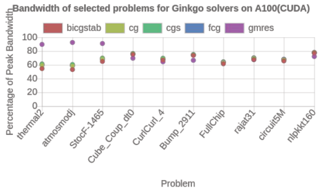
xSDK

E4S

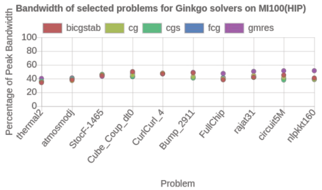
Ginkgo

PERFORMANCE

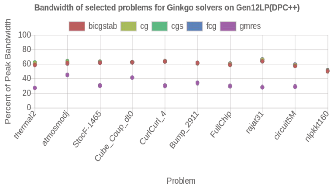
NVIDIA



AMD

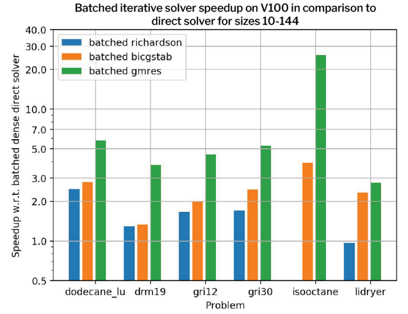


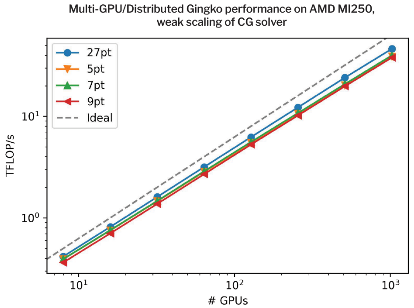
INTEL



FUNCTIONALITY

	Functionality	OMP	CUDA	HIP	DPC++
Basic	SpMV	✔	✔	✔	✔
	SpMM	✔	✔	✔	✔
	SpGeMM	✔	✔	✔	✔
Krylov solvers	BiCG	✔	✔	✔	✔
	BiCGSTAB	✔	✔	✔	✔
	CG	✔	✔	✔	✔
	CGS	✔	✔	✔	✔
	GMRES	✔	✔	✔	✔
	IDR	✔	✔	✔	✔
Preconditioners	(Block-)Jacobi	✔	✔	✔	✔
	ILU/IC		✔	✔	✔
	Parallel ILU/IC	✔	✔	✔	✔
	Parallel ILUT/ICT	✔	✔	✔	✔
	Sparse Approximate Inverse	✔	✔	✔	✔
Batched	Batched BiCGSTAB	✔	✔	✔	
	Batched CG	✔	✔	✔	
	Batched GMRES	✔	✔	✔	
	Batched ILU	✔	✔	✔	✔
	Batched ISAI	✔	✔	✔	✔
	Batched Jacobi	✔	✔	✔	✔
AMG	AMG preconditioner	✔	✔	✔	✔
	AMG solver	✔	✔	✔	✔
	Parallel Graph Match	✔	✔	✔	✔
Sparse direct	Symbolic Cholesky	✔	✔	✔	✔
	Numeric Cholesky	UNDER DEVELOPMENT			
	Symbolic LU	UNDER DEVELOPMENT			
	Numeric LU	✔	✔	✔	
	Sparse TRSV	✔	✔	✔	
Utilities	On-Device Matrix Assembly	✔	✔	✔	✔
	MC64/RCM reordering	✔			
	Wrapping user data	✔			
	Logging	✔			
	PAPI counters	✔			





We have no standard for sparse operations

cuSPARSE, rocSPARSE, oneAPI, PETSc, Trilinos…
all have different interfaces & functionality

We try to define an interface that allows for horizontal and vertical compatibility:

- Useful as building blocks for high-level algorithms
- Vendors can wrap their current interface

- Horizontal: bindings for Fortran, C, …

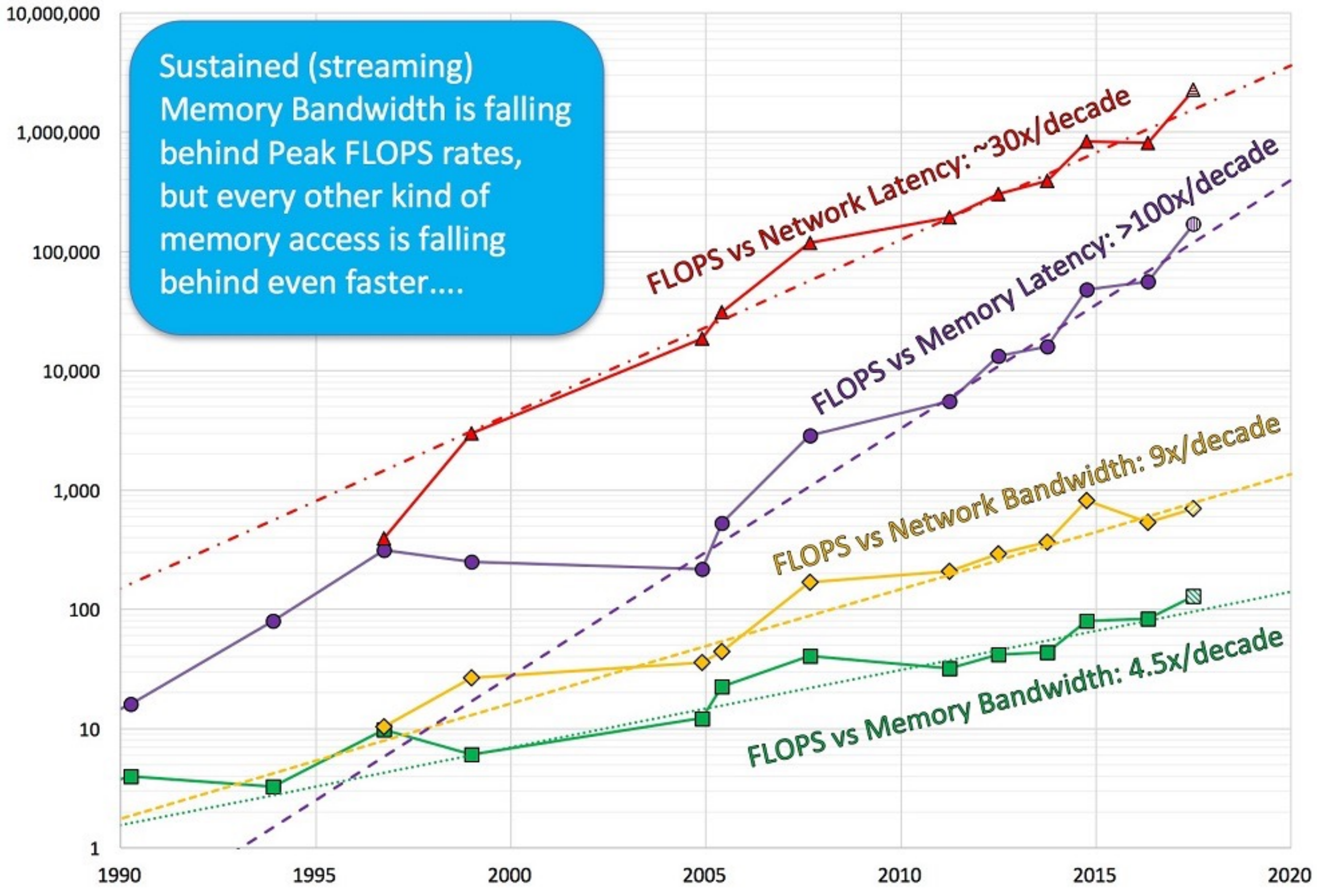
https://icl.utk.edu/workshops/sparseblas2023/index.html

Want to participate in the discussion
– reach out hanzt@icl.utk.edu

Hartwig Anzt • Sie
Director of the Innovative Computing Laboratory (ICL) an…
1 Woche •

Three days of good talks, discussions, and prototyping are over: Thank you for the sparseBLAS workshop held at the Innovative Computing Laboratory (ICL) of the Tickle College of Engineering at the University of Tennessee. Together with experts from Intel Corporation, NVIDIA, AMD, Arm, MathWorks, University of California, Berkeley, Intel Labs, Computing at ORNL, Karlsruher Institut für Technologie (KIT), RIKEN, Lawrence Livermore National Laboratory, Sandia National Laboratories, and Massachusetts Institute of Technology, we worked on defining a common understanding and interface design for sparse linear algebra functionality.

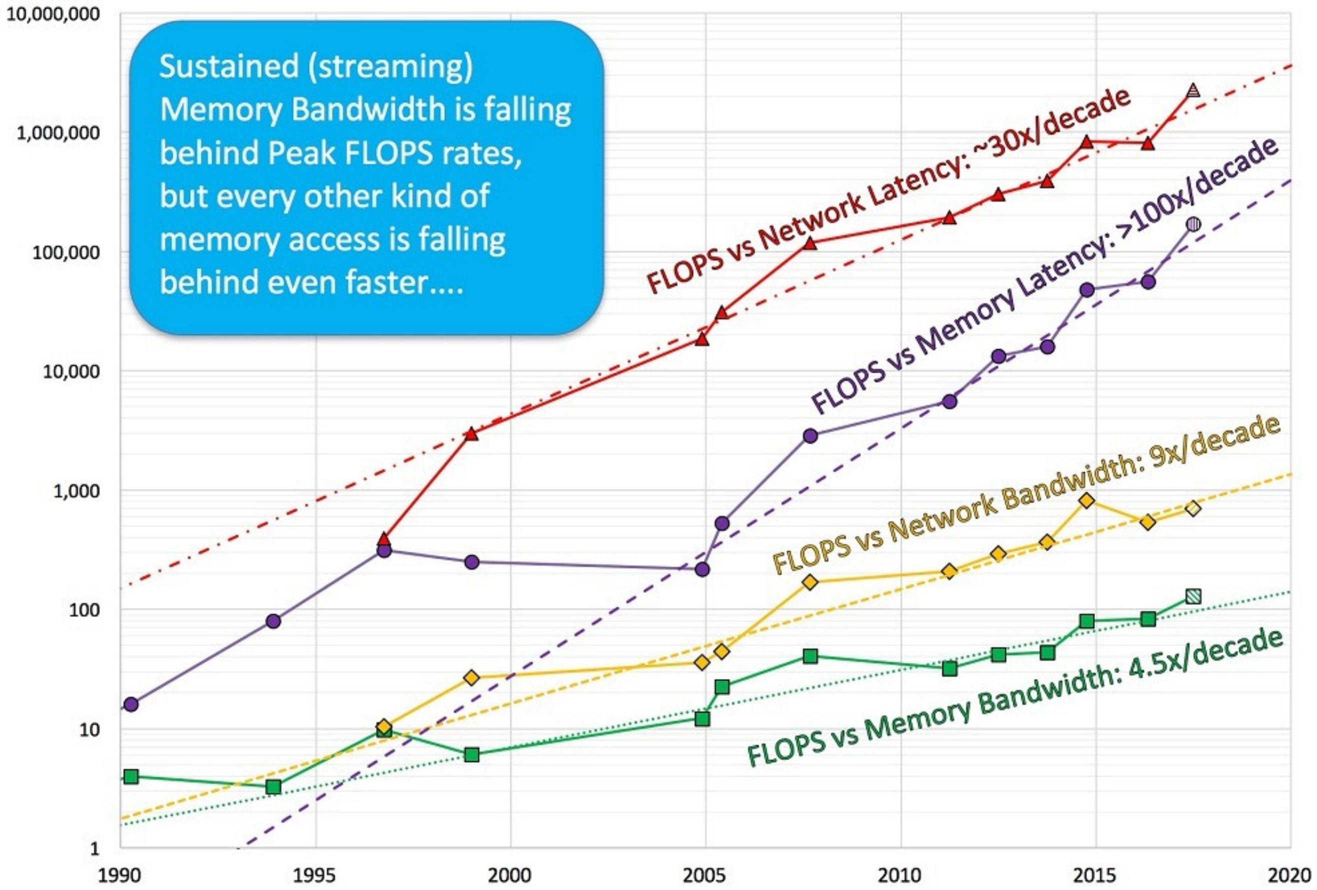
Any hope for sparse operations? – Yes, but we will have to work harder…



Trends in the relative performance of floating-point arithmetic and several classes of data access for select HPC servers over the past 25 years. Source: John McCalpin

- The gap between compute performance and memory performance is widening
- Hardware increasingly features powerful matrix engines
- We should use these ”free” FLOP/s
- One strategy is to compress data for memory operations
- We need efficient need on-chip data compression

Any hope for sparse operations? – Yes, but we will have to work harder…



Trends in the relative performance of floating-point arithmetic and several classes of data access for select HPC servers over the past 25 years. Source: John McCalpin

- The gap between compute performance and memory performance is widening
- Hardware increasingly features powerful matrix engines
- We should use these ”free” FLOP/s
- One strategy is to compress data for memory operations
- We need efficient need on-chip data compression

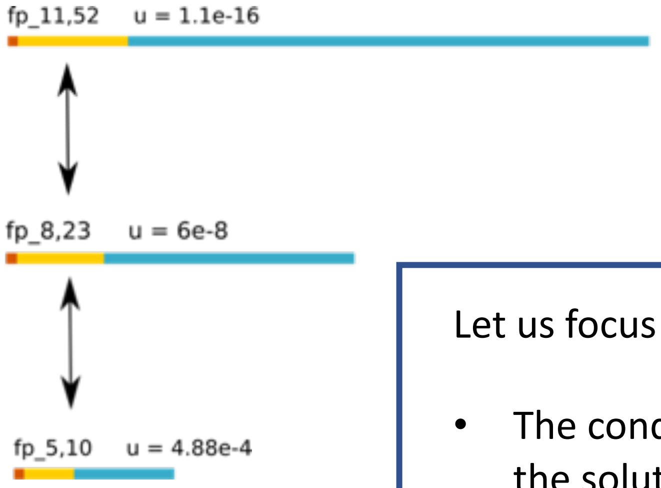
Background: Floating Point Formats, Accuracy, and Performance

$$u_d \approx 10^{-16}$$
$$u_s \approx 10^{-7}$$

double precision (FP64)

single precision (FP32)

half precision (FP16)



Broadly speaking….

- The length of the **exponent** determines the **range** of the values that can be represented;
- The length of the **significand** determines how **accurate** values can be represented;
- **Rounding effects accumulate over a sequence of computations;**
- **The data access cost linearly depends on the memory volume;**

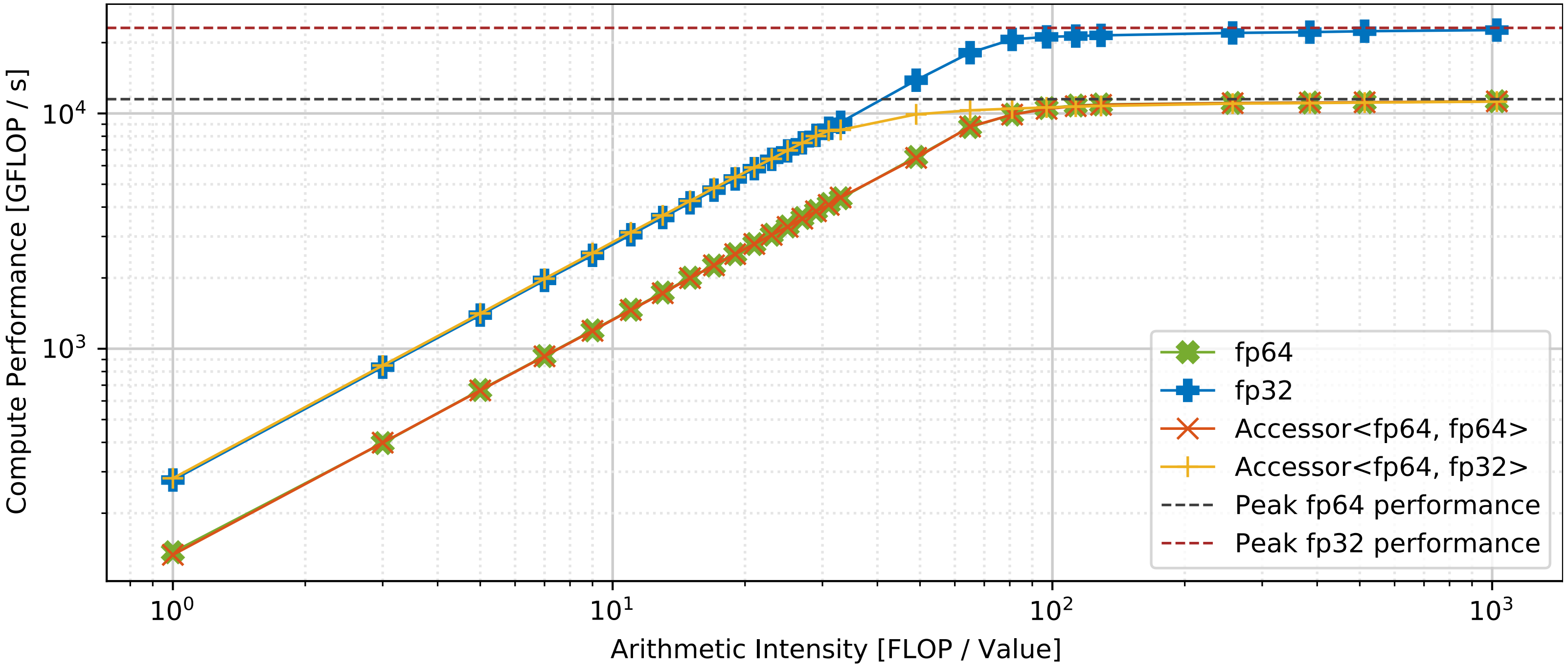
Let us focus on linear systems of the form Ax=b.

- The conditioning of a linear system reflects how sensitive the solution x is with regard to changes in the right-hand side b.
- Rule of thumb:

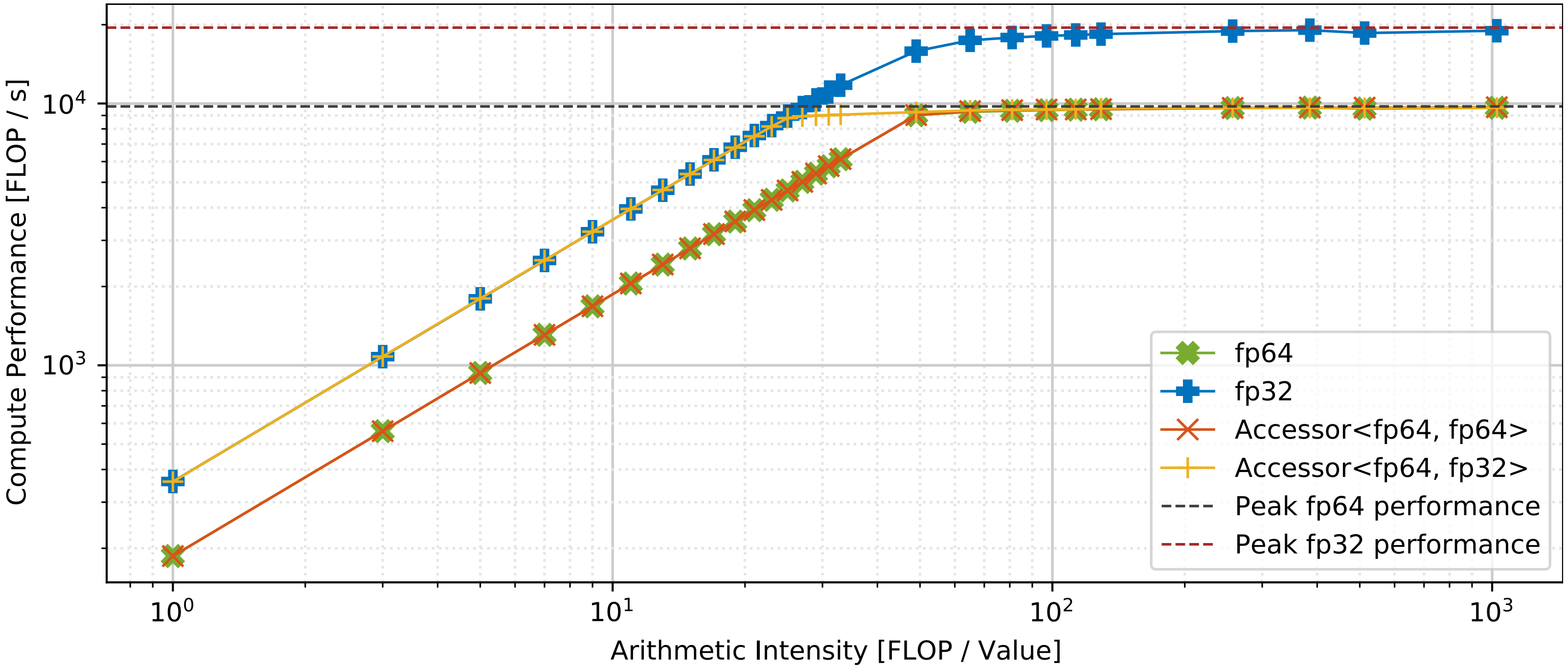
relative residual accuracy ≈ **(unit round-off)** * **(linear system's condition number)**

N. Higham: Accuracy and stability of numerical algorithms. SIAM, 2002.

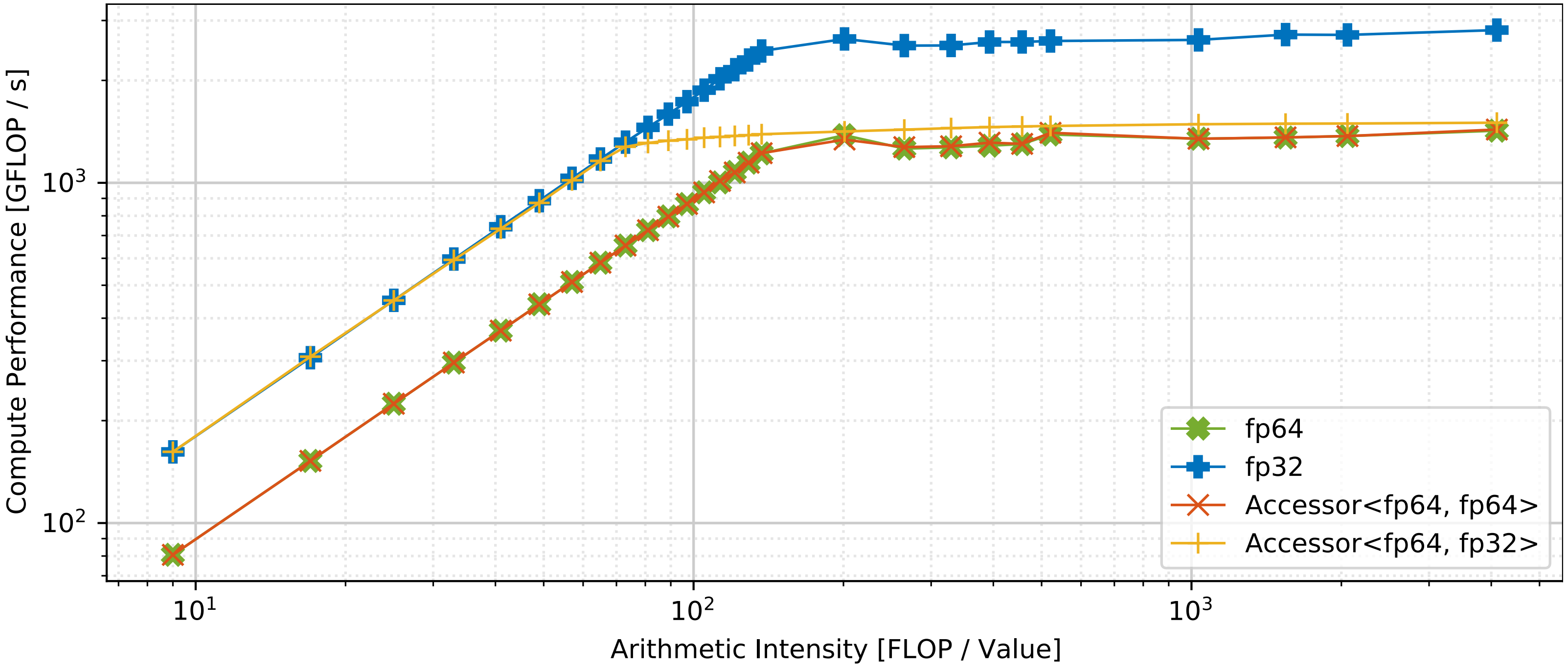
Accessor for AMD MI100 GPU



Accessor for NVIDIA A100 GPU



Accessor for Intel Skylake CPU



Accessor for AMD EPYC CPU