

## ***Addressing the Memory Wall: Designing a Multiprecision Ecosystem***

PASC 2021: Multiprecision Numerics in Scientific High Performance Computing  
July 7<sup>th</sup>, 2021

Hartwig Anzt

Steinbuch Centre for Computing (SCC)



Hartwig Anzt



Isha Aggarwal



Terry Cojean



Fritz Göbel



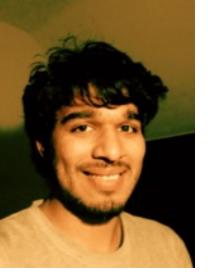
T. Grützmacher



Aditya Kashi



Marcel Koch



Pratik Nayak



Gregor Olenik



Tobias Ribizel



Mike Tsai

*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 and the Helmholtz Impuls und Vernetzungsfond VH-NG-1241.*



**GPU-centric high performance sparse linear algebra.** Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.

#### ▪ High performance sparse linear algebra

- Linear algebra building blocks: SpMV, SpMM, SpGEAM,...;
- Linear solvers: BiCG, BiCGSTAB, CG, CGS, FCG, GMRES, IDR;
- Advanced preconditioning techniques: ParILU, ParILUT, SAI;
- Batched iterative solvers;

#### ▪ Exascale early systems GPU-readiness

- Available: Nvidia GPU (CUDA), AMD GPU (HIP), Intel GPU (DPC++), CPU Multithreading (OpenMP);
- C++, CMake build;

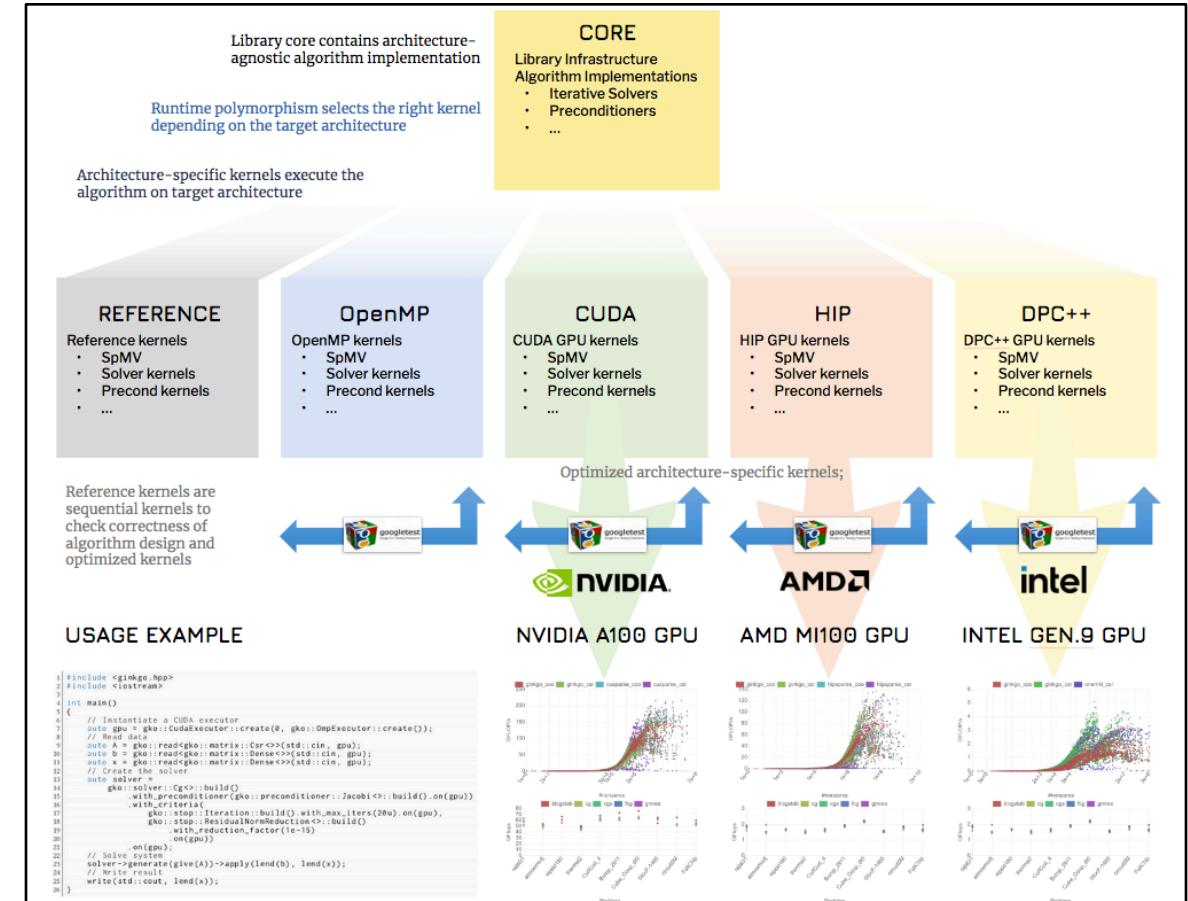
#### ▪ Open source, community-driven

- Freely available (BSD License), GitHub, and Spack;
- Part of the **xSDK** and **E4S** software stack;
- Can be used from **deal.II** and **MFEM**;



#### ▪ Modular precision ecosystem

- Decoupling of arithmetic precision and memory precision;
- Compressed Basis (CB) Krylov methods;
- Mixed precision algorithms: adaptive precision Jacobi, FSPAI;



<https://ginkgo-project.github.io/>



**GPU-centric high performance sparse linear algebra.** Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.

Open source, community-driven

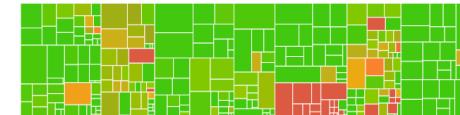


- **Ginkgo is a child of the ECP effort:**
  - Lessons learnt in other software projects (e.g. MAGMA) are incorporated;
  - Focus on sparse linear algebra;
- **Better Scientific Software** (BSSW.io) proved to be a valuable resource;
- Freely available (BSD License), **GitHub**, and **Spack**;
- Modern programming language (C++14) and build system (**CMake**);
- Continuous Integration and comprehensive unit testing;
- From the beginning on, part of the **xSDK** and **E4S** software stacks;
- **Portability as central design principle**;
- Using the **architecture-native languages** to push the performance to the limit;



Codecov Report

! No coverage uploaded for pull request base (develop@e2748a8). Click here to learn what that means.  
The diff coverage is 98.42%.

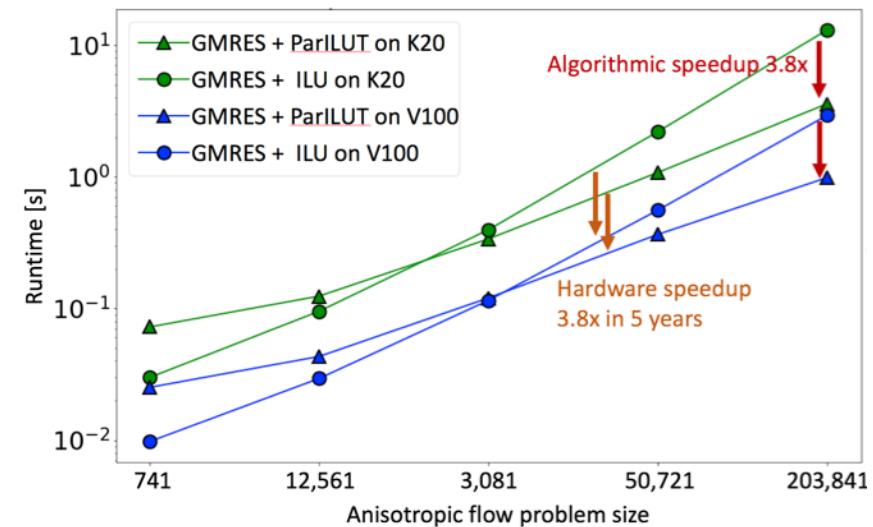
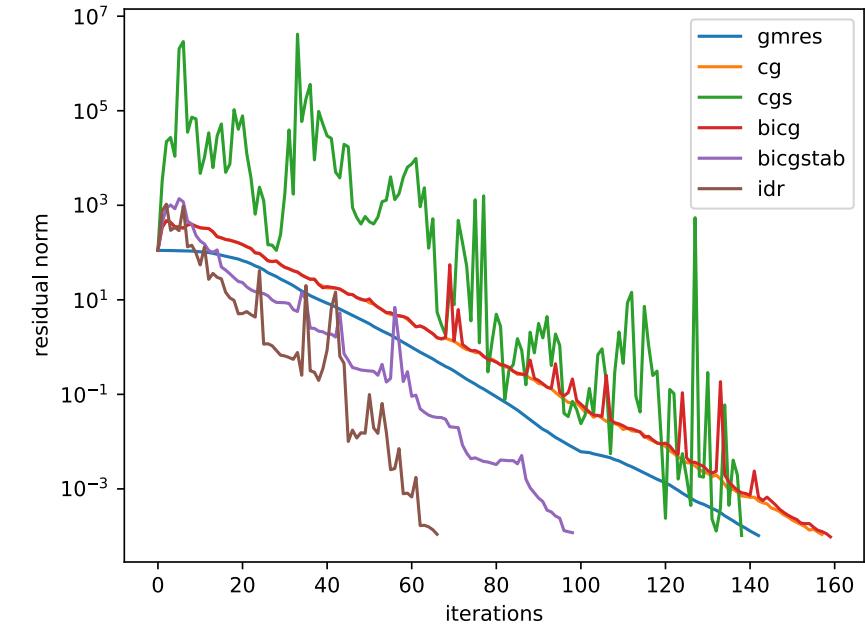


##	Coverage Diff	##
=====		
Coverage	?	91.76%
Files	?	285
Lines	?	18575
Branches	?	0
=====		
Hits	?	17846
Misses	?	1529
Partials	?	0

Impacted Files	Coverage Δ
include/ginkgo/core/preconditioner/lu.hpp	68.56% <80% (s)
reference/test/log/convergence.cpp	91.66% <100% (s)
reference/test/solver/bicgstab_kernels.cpp	100% <100% (s)

## High performance sparse linear algebra

- Linear algebra building blocks: SpMV, SpMM, SpGEAM,...;
- Linear solvers: BiCG, BiCGSTAB, CG, CGS, FCG, GMRES, IDR;
- Mixed precision algorithms:  
CB-GMRES, adaptive precision Jacobi, FSPAI;
- Advanced preconditioning techniques: ParILU, ParILUT, SAI;
- Batched solver technology;
- *Extensible, sustainable, production-ready;*





**GPU-centric high performance sparse linear algebra.** Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.

Library core contains architecture-agnostic algorithm implementation

## CORE

- Library Infrastructure
- Algorithm Implementations
  - Iterative Solvers
  - Preconditioners
  - ...



**GPU-centric high performance sparse linear algebra.** Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.

Library core contains architecture-agnostic algorithm implementation

## CORE

- Library Infrastructure
- Algorithm Implementations
  - Iterative Solvers
  - Preconditioners
  - ...

## REFERENCE

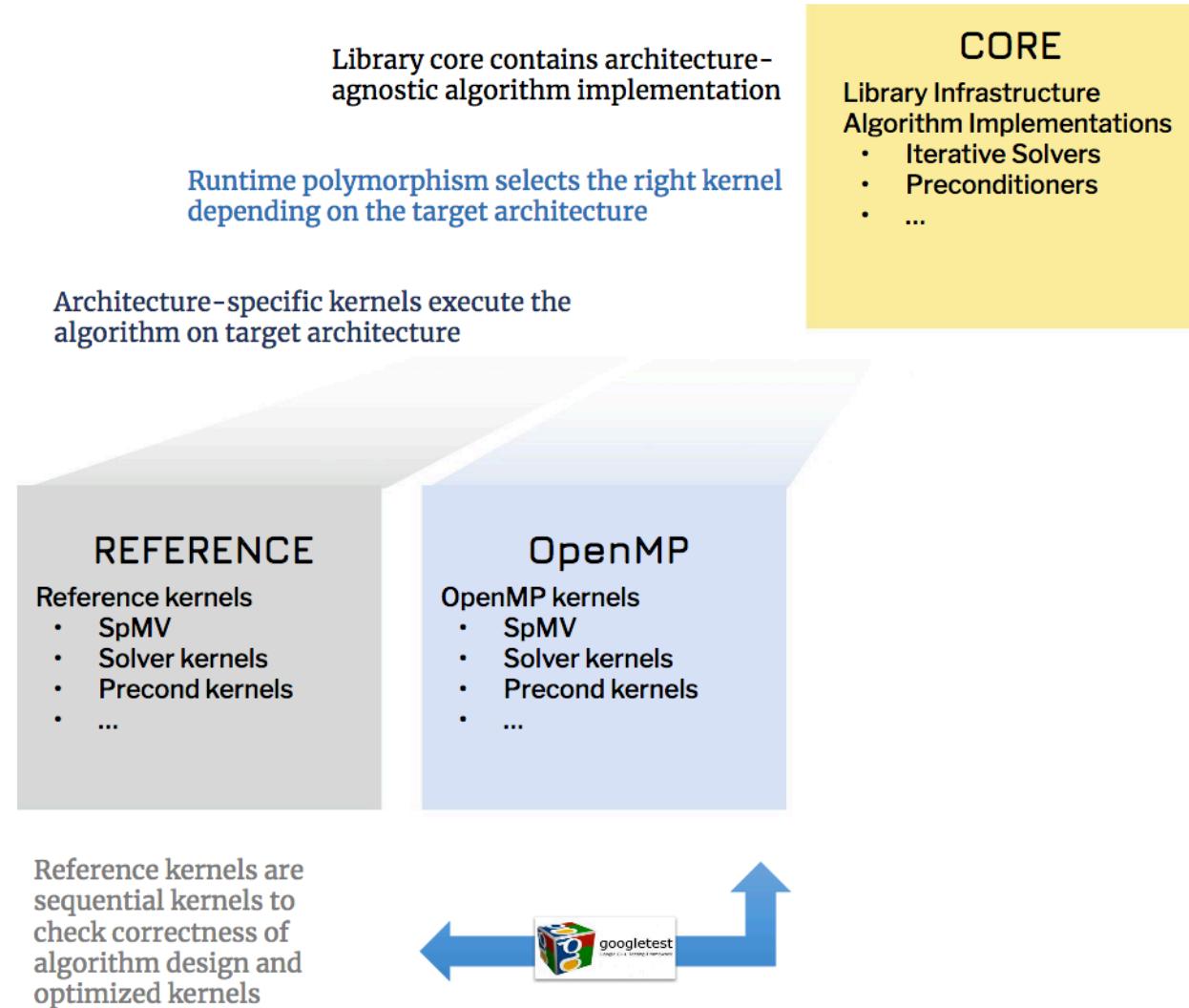
Reference kernels

- SpMV
- Solver kernels
- Precond kernels
- ...

Reference kernels are sequential kernels to check correctness of algorithm design and optimized kernels

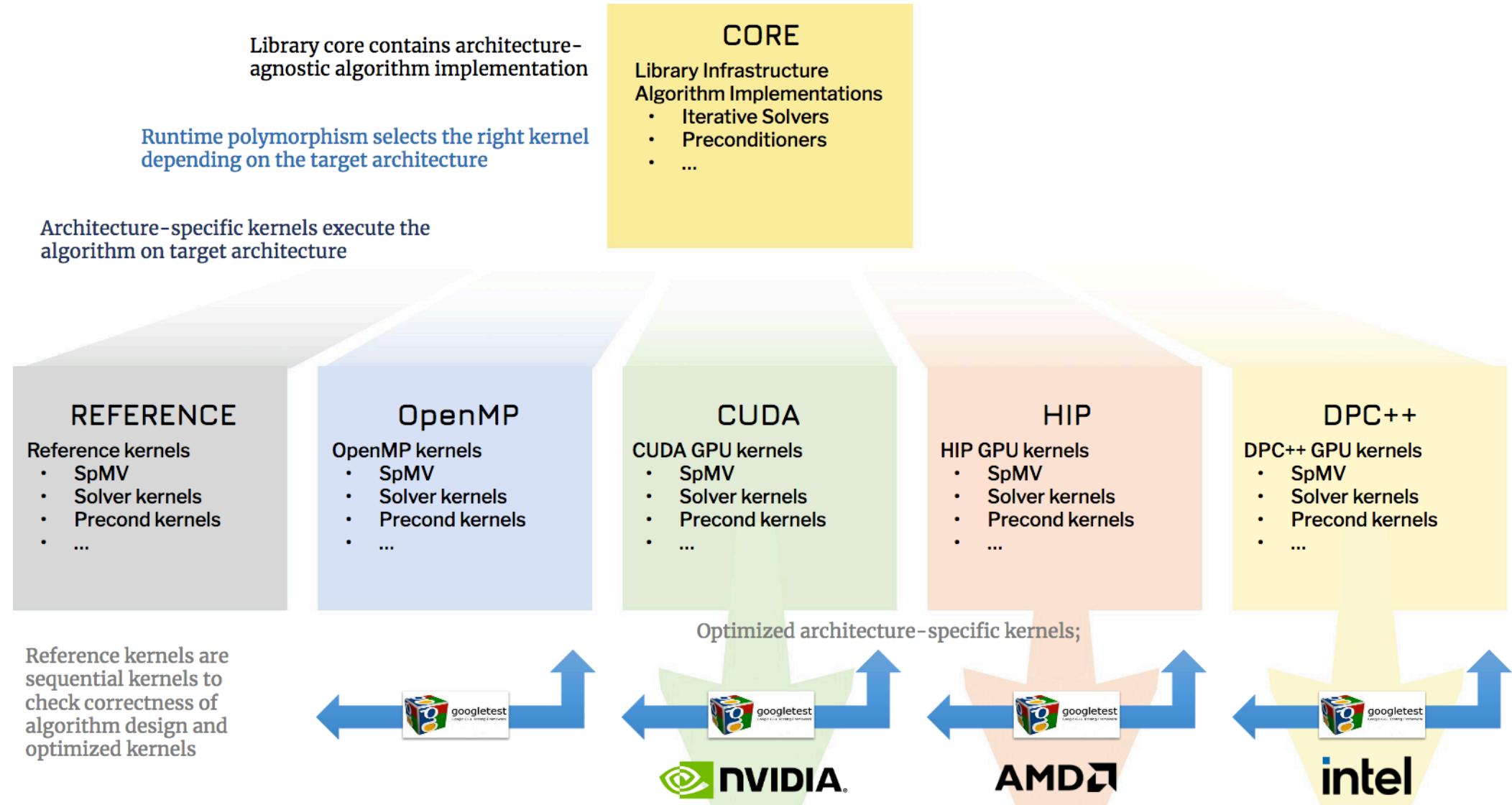


**GPU-centric high performance sparse linear algebra.** Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.





**GPU-centric high performance sparse linear algebra.** Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.

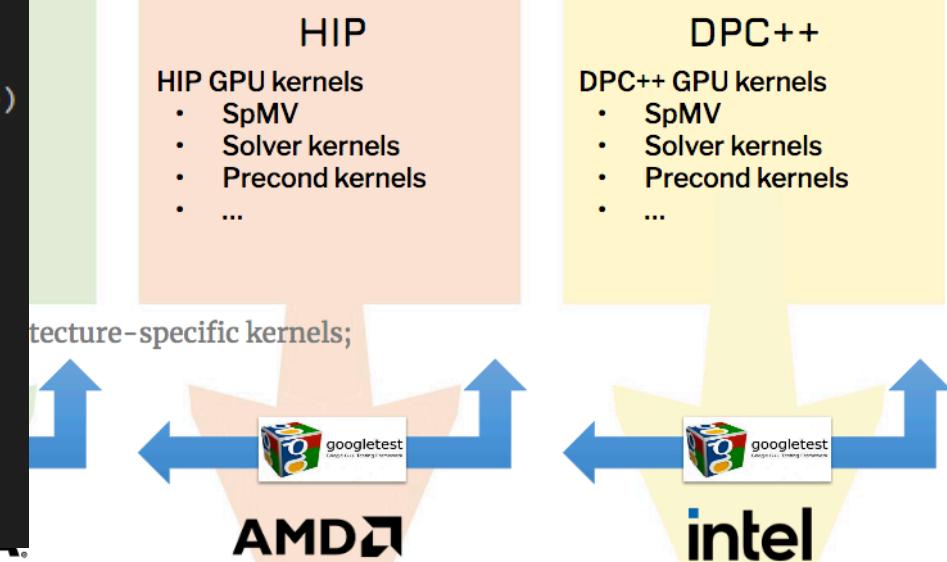




**GPU-centric high performance sparse linear algebra.** Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.

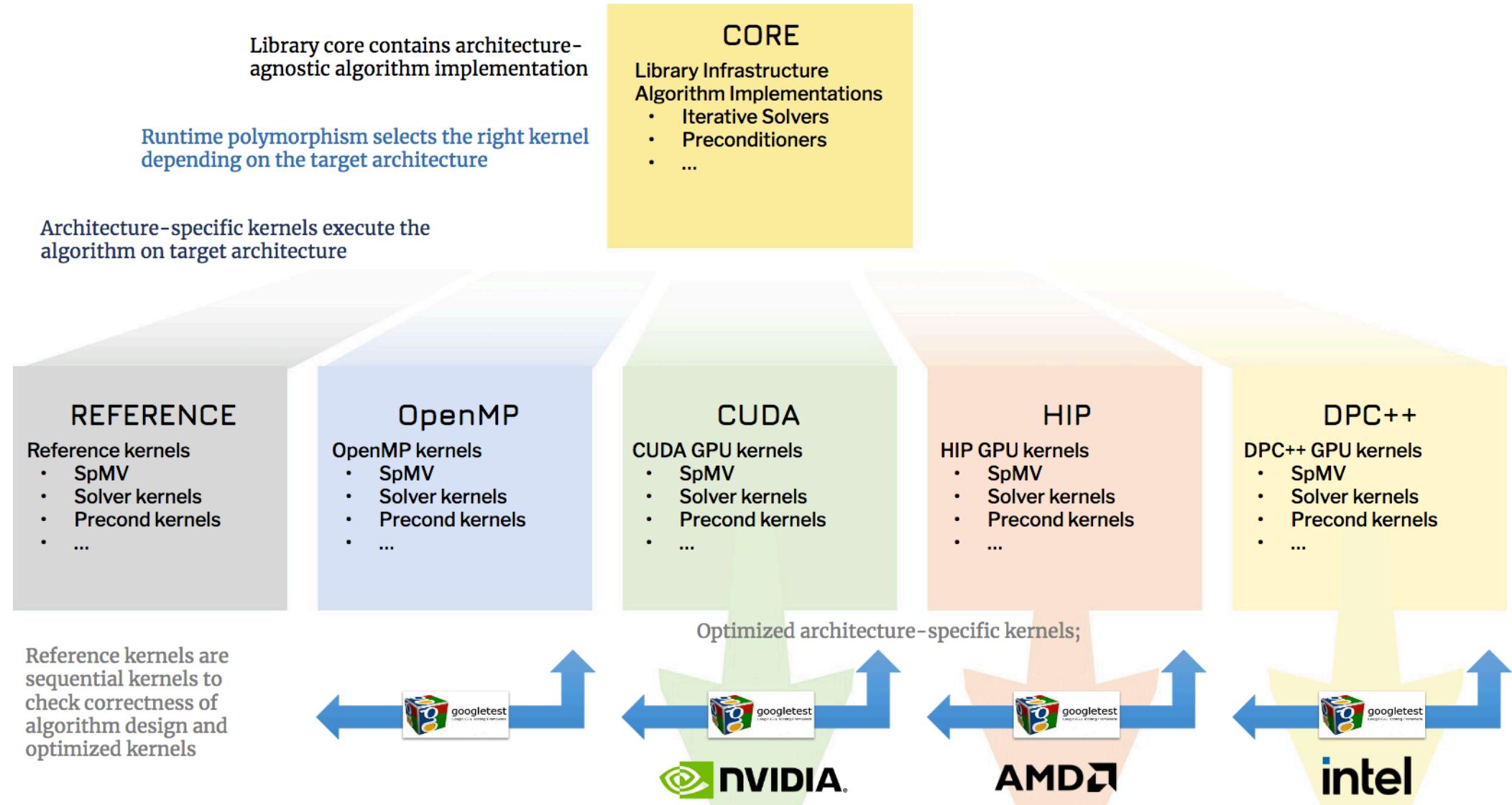
```
1 #include <ginkgo/ginkgo.hpp>
2 #include <iostream>
3
4 int main()
5 {
6     // Instantiate a GPU executor
7     auto gpu =
8         gko::CudaExecutor::create(0, gko::OmpExecutor::create());
9     +----- gko::DpcppExecutor::create(0, gko::OmpExecutor::create());
10
11    // Read data
12    auto A = gko::read<gko::matrix::Csr<>>(std::cin, gpu);
13    auto b = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
14    auto x = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
15
16    // Create the solver
17    auto solver =
18        gko::solver::Cg<>::build()
19            .with_preconditioner(gko::preconditioner::Jacobi<>::build().on(gpu))
20            .with_criteria(
21                gko::stop::Iteration::build().with_max_iters(1000u).on(gpu),
22                gko::stop::ResidualNormReduction<>::build()
23                    .with_reduction_factor(1e-15)
24                    .on(gpu))
25            .on(gpu);
26
27    // Solve system
28    solver->generate(give(A))->apply(lend(b), lend(x));
29
30    // Write result
31    write(std::cout, lend(x));
32 }
```

ons





**GPU-centric high performance sparse linear algebra.** Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.





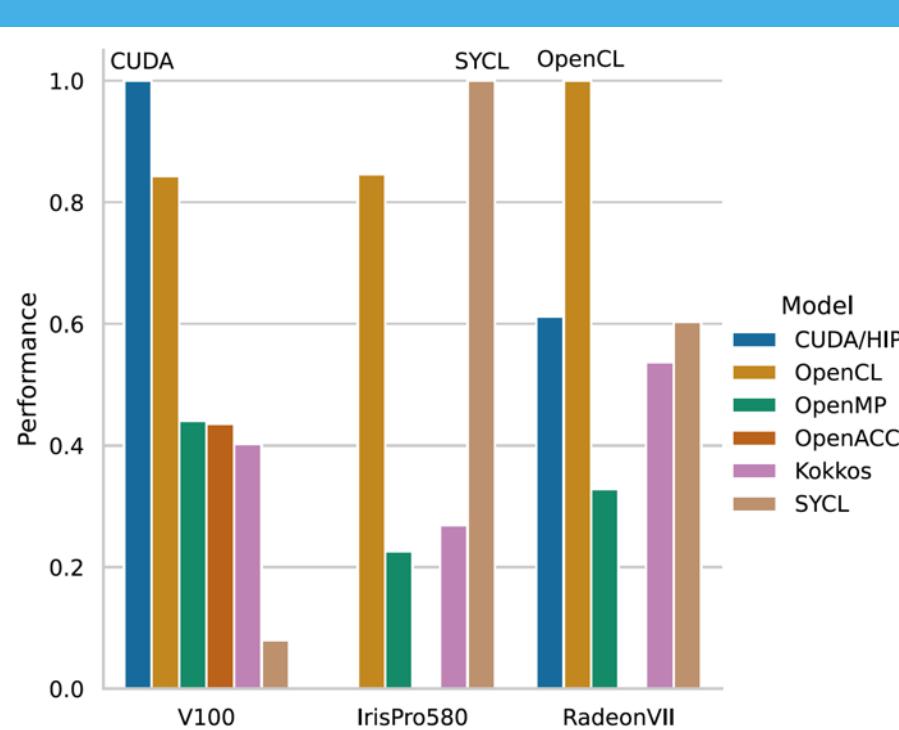
**GPU-centric high performance sparse linear algebra.** Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.

Library agnostic  
Runtime polymorphic depending on the target architecture  
Architecture-specific kernels for each algorithm on target architecture

## REFERENCE

- Reference kernels
- SpMV
  - Solver kernels
  - Precond kernels
  - ...

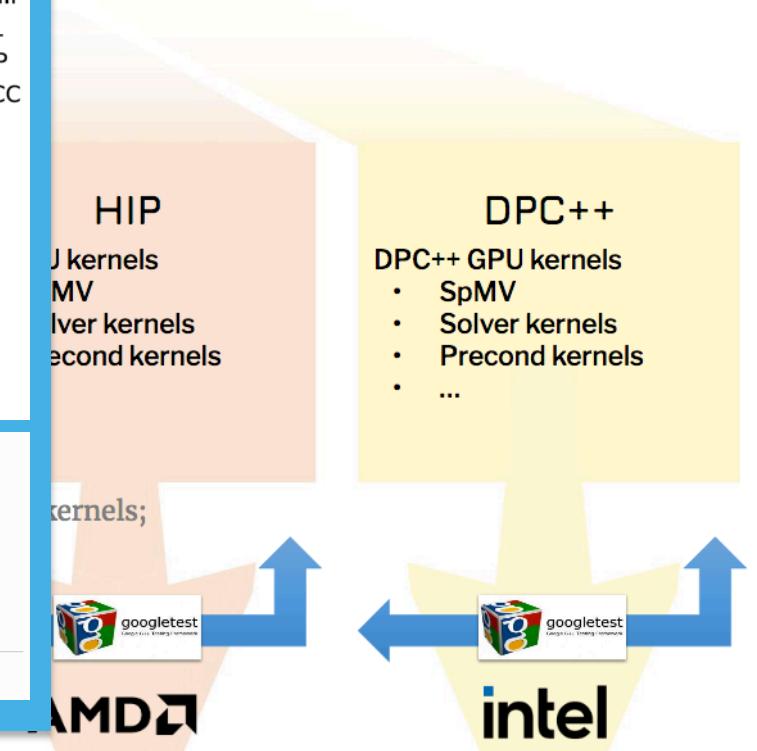
Reference kernels are sequential kernels to check correctness of algorithm design and optimized kernels



[International Conference on High Performance Computing](#)  
ISC High Performance 2021: High Performance Computing pp 332-350 | [Cite as](#)

## A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application

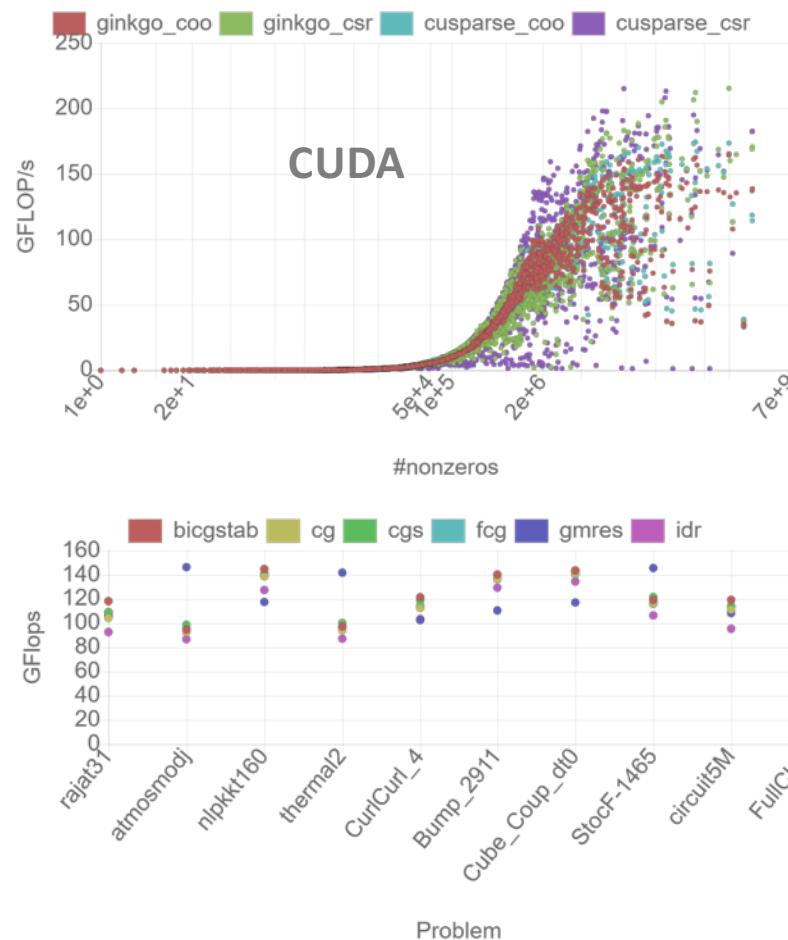
Authors  
Andrei Poenaru [✉](#), Wei-Chen Lin, Simon McIntosh-Smith [✉](#)  
Authors and affiliations



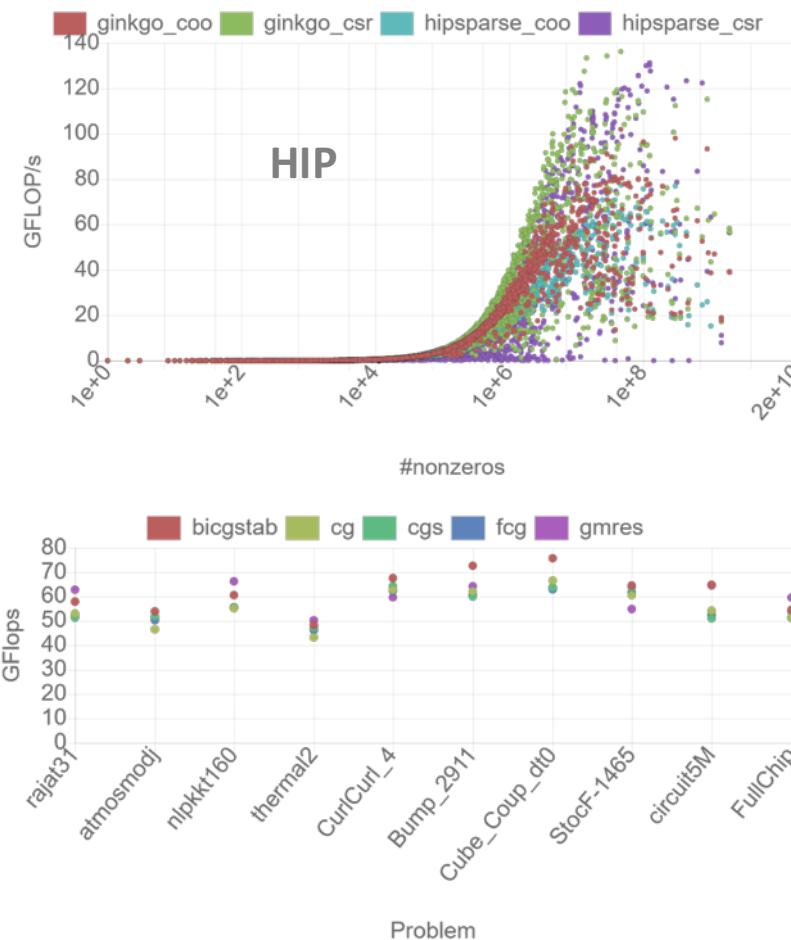


**GPU-centric high performance sparse linear algebra.** Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.

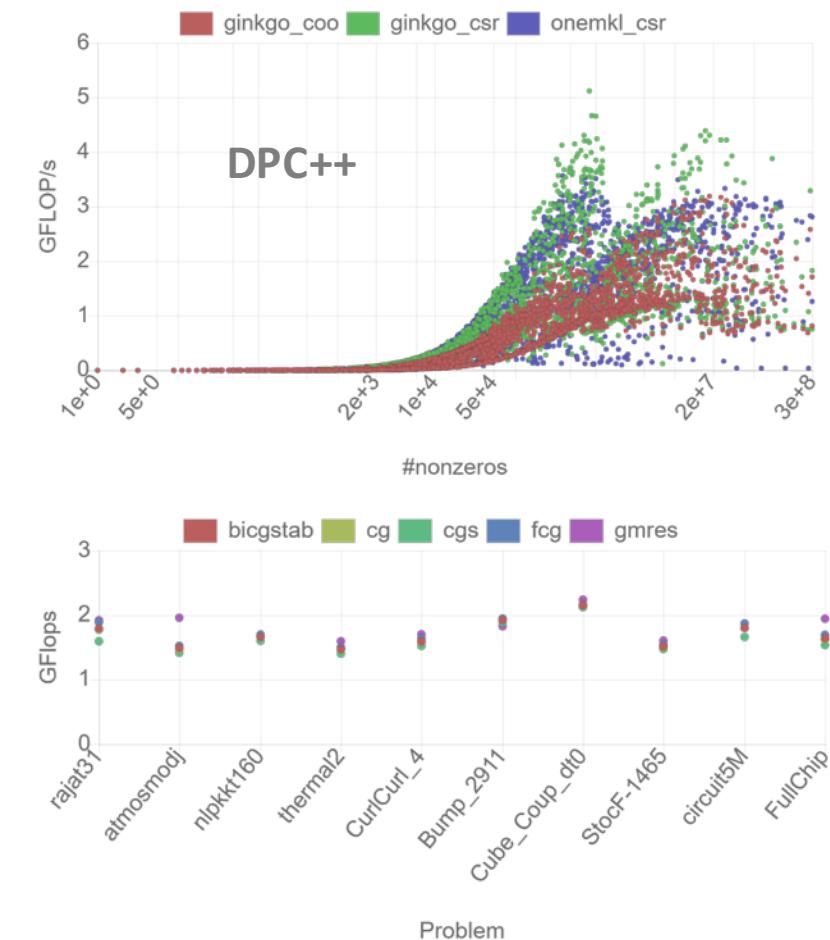
### NVIDIA A100 GPU



### AMD MI100 GPU



### intel Gen9 GPU





**GPU-centric high performance sparse linear algebra.** Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.

## Linear Operator Design

We express everything as *Linear Operator* and leverage C++ class inheritance.

Matrix-Vector Product

Preconditioner (for matrix  $A$ )

Solver (for system  $Ax = b$ )

$$x := A \cdot b$$

$$x := M^{-1} \cdot b$$

$$x := S \cdot b$$

$$M^{-1} \approx A^{-1}$$

$$M^{-1} = \Pi(A)$$

$$S \approx A^{-1}$$

$$S = \Sigma(A)$$

All of them can be expressed as

Application of a linear operator\* (LinOp)  $L : \mathbb{F}^m \rightarrow \mathbb{F}^m$

## Linear Operator Design

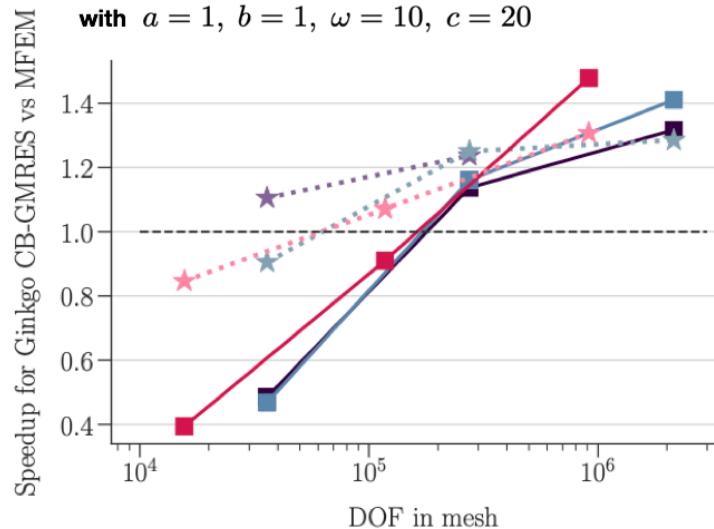
We express everything as *Linear Operator* and leverage it.

### Matrix-Vector Product

$$x := A \cdot b$$

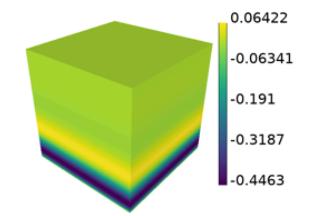
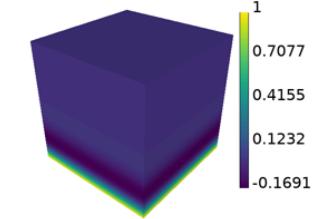
All of them can be expressed as

### Application of a linear operator\* (LinOp)



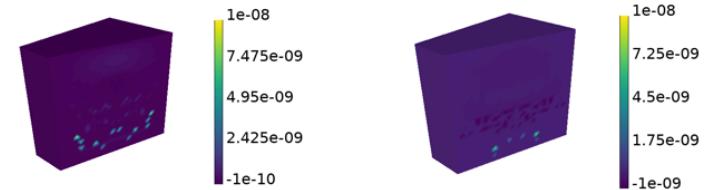
**Speedup of Ginkgo's Compressed Basis-GMRES solver vs MFEM's GMRES solver for three different orders of basis functions (p) for MFEM's example 22.**  
The tests use the "partial assembly" type of MFEM matrix-free operators.

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)





**GPU-centric high performance sparse linear algebra.** Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.

#### ▪ High performance sparse linear algebra

- Linear algebra building blocks: SpMV, SpMM, SpGEAM,...;
- Linear solvers: BiCG, BiCGSTAB, CG, CGS, FCG, GMRES, IDR;
- Advanced preconditioning techniques: ParILU, ParILUT, SAI;
- Batched iterative solvers;

#### ▪ Exascale early systems GPU-readiness

- Available: Nvidia GPU (CUDA), AMD GPU (HIP), Intel GPU (DPC++), CPU Multithreading (OpenMP);
- C++, CMake build;

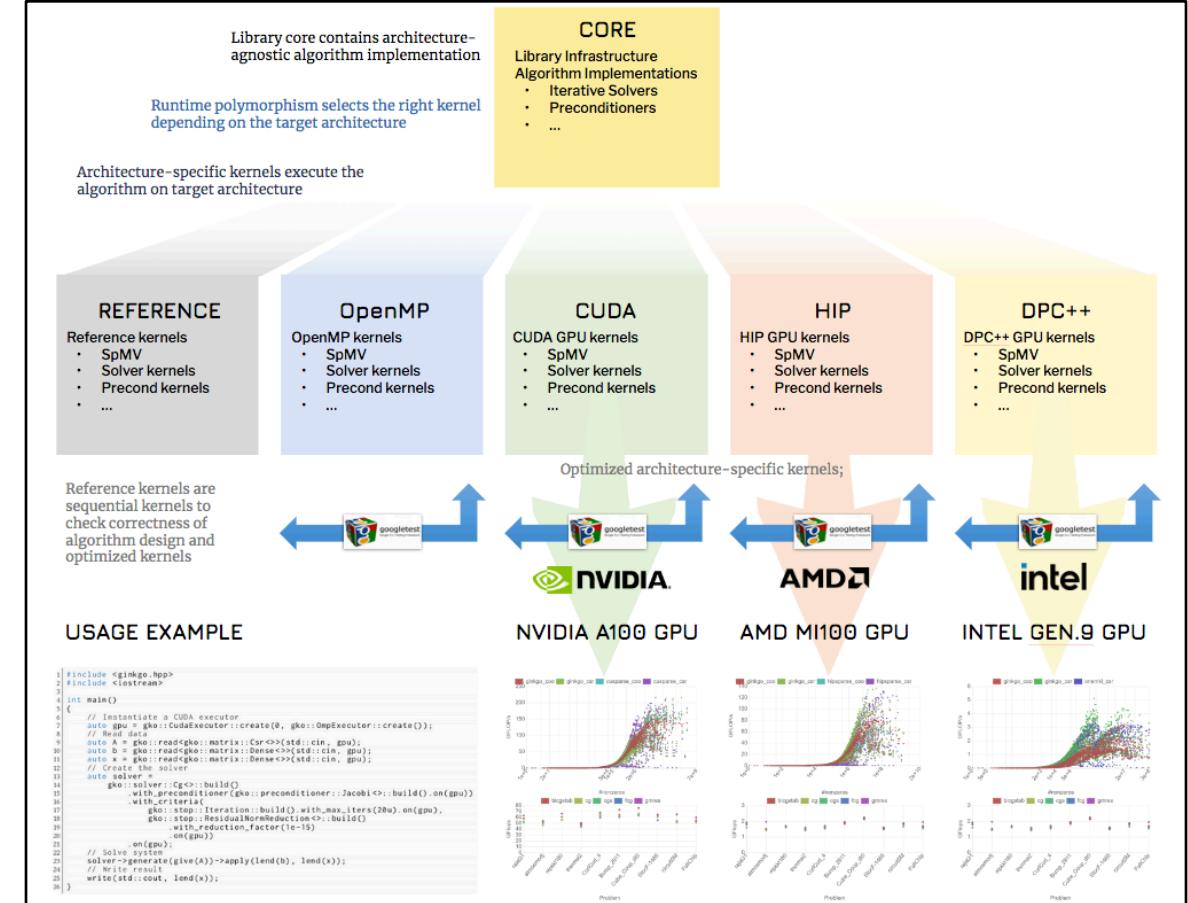
#### ▪ Open source, community-driven

- Freely available (BSD License), GitHub, and Spack;
- Part of the **xSDK** and **E4S** software stack;
- Can be used from **deal.II** and **MFEM**;



#### ▪ Modular precision ecosystem

- Decoupling of arithmetic precision and memory precision;
- Compressed Basis (CB) Krylov methods;
- Mixed precision algorithms: adaptive precision Jacobi, FSPAI;



<https://ginkgo-project.github.io/>

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 and the Helmholtz Impuls und Vernetzungsfond VH-NG-1241.

# Simplifying research on mixed precision algorithms

- Generally, designing mixed precision algorithms in C/C++ is tedious and messy;
  - Requires writing code with many explicit format conversions;
  - Significant level of data duplication (in different precisions);

```
float A;
double x, y;
y = A*x;
```

# Simplifying research on mixed precision algorithms

- Generally, designing mixed precision algorithms in C/C++ is tedious and messy;
  - Requires writing code with many explicit format conversions;
  - Significant level of data duplication (in different precisions);

```
float A;
double x, y;
y = A*x;
```

- **Idea: provide a framework, where the compiler handles all format conversions:**

- Create objects in different precisions and combine them in the code;
- Automatically-generated temporary copies are used to match the precisions;
- After completion of the arithmetic, the output is converted back to the requested format;
- No need to worry about explicit conversions;
- The framework may not provide good performance;
- It still allows for easy and quick investigation of numerical effects;
- If an idea has proven successful, one can replace the on-the-fly conversion with hardware-optimized implementations;



*"Mix & Match Precisions in Ginkgo"*

```
using mtx =
    gko::matrix::Csr<float, int>;
using vec =
    gko::matrix::Dense<double>;
...
auto solver = solver_gen->generate(A);
solver->apply(lend(b), lend(x));
```

# Mixing precisions in Ginkgo



## 1. Create Objects in different precision formats

```
using HighPrecision = double;
using LowPrecision = float;
using hp_mtx = gko::matrix::Ell<HighPrecision, IndexType>;
using lp_mtx = gko::matrix::Ell<LowPrecision, IndexType>;

// read the matrix into HighPrecision and LowPrecision.
auto hp_A = share(gko::read<hp_mtx>(std::ifstream("data/A mtx"), exec));
auto lp_A = share(gko::read<lp_mtx>(std::ifstream("data/A mtx"), exec));
// Set the shortcut for each dimension
auto A_dim = hp_A->get_size();
auto b_dim = gko::dim<2>{A_dim[1], 1};
auto x_dim = gko::dim<2>{A_dim[0], b_dim[1]};
auto host_b = hp_vec::create(exec->get_master(), b_dim);
// fill the b vector with some random data
std::ranlux48 rand_engine(32);
auto dist = std::uniform_real_distribution<RealValueType>(0.0, 1.0);
for (int i = 0; i < host_b->get_size()[0]; i++) {
    host_b->at(i, 0) = get_rand_value<HighPrecision>(dist, rand_engine);
}
```

ginkgo -> examples -> mixed-spmv

<https://github.com/ginkgo-project/ginkgo/tree/develop/examples/mixed-spmv>

# Mixing precisions in Ginkgo



## 1. Create Objects in different precision formats

```
using HighPrecision = double;
using LowPrecision = float;
using hp_mtx = gko::matrix::Ell<HighPrecision, IndexType>;
using lp_mtx = gko::matrix::Ell<LowPrecision, IndexType>;
// read the matrix into HighPrecision and LowPrecision.
auto hp_A = share(gko::read<hp_mtx>(std::ifstream("data/A mtx"), exec));
auto lp_A = share(gko::read<lp_mtx>(std::ifstream("data/A mtx"), exec));
// Set the shortcut for each dimension
auto A_dim = hp_A->get_size();
auto b_dim = gko::dim<2>{A_dim[1], 1};
auto x_dim = gko::dim<2>{A_dim[0], b_dim[1]};
auto host_b = hp_vec::create(exec->get_master(), b_dim);
// fill the b vector with some random data
std::ranlux48 rand_engine(32);
auto dist = std::uniform_real_distribution<RealValueType>(0.0, 1.0);
for (int i = 0; i < host_b->get_size()[0]; i++) {
    host_b->at(i, 0) = get_rand_value<HighPrecision>(dist, rand_engine);
}
```

## 2. Combine different precisions in computations

```
// Hp * Hp -> Hp
auto hp_sec = timing(exec, hp_A, hp_b, hp_x);
// Lp * Lp -> Lp
auto lp_sec = timing(exec, lp_A, lp_b, lp_x);
// Hp * Lp -> Hp
auto hplp_sec = timing(exec, hp_A, lp_b, hplp_x);
// Lp * Hp -> Hp
auto lpplp_sec = timing(exec, lp_A, hp_b, lpplp_x);
// Lp * Hp -> Hp
auto lphp_sec = timing(exec, lp_A, hp_b, lphp_x);
```

ginkgo -> examples -> mixed-spmv

<https://github.com/ginkgo-project/ginkgo/tree/develop/examples/mixed-spmv>

# Mixing precisions in Ginkgo



## 1. Create Objects in different precision formats

```
using HighPrecision = double;
using LowPrecision = float;
using hp_mtx = gko::matrix::Ell<HighPrecision, IndexType>;
using lp_mtx = gko::matrix::Ell<LowPrecision, IndexType>;
// read the matrix into HighPrecision and LowPrecision.
auto hp_A = share(gko::read<hp_mtx>(std::ifstream("data/A mtx"), exec));
auto lp_A = share(gko::read<lp_mtx>(std::ifstream("data/A mtx"), exec));
// Set the shortcut for each dimension
auto A_dim = hp_A->get_size();
auto b_dim = gko::dim<2>{A_dim[1], 1};
auto x_dim = gko::dim<2>{A_dim[0], b_dim[1]};
auto host_b = hp_vec::create(exec->get_master(), b_dim);
// fill the b vector with some random data
std::ranlux48 rand_engine(32);
auto dist = std::uniform_real_distribution<RealValueType>(0.0, 1.0);
for (int i = 0; i < host_b->get_size()[0]; i++) {
    host_b->at(i, 0) = get_rand_value<HighPrecision>(dist, rand_engine);
}
```

ginkgo -> examples -> mixed-spmv

<https://github.com/ginkgo-project/ginkgo/tree/develop/examples/mixed-spmv>

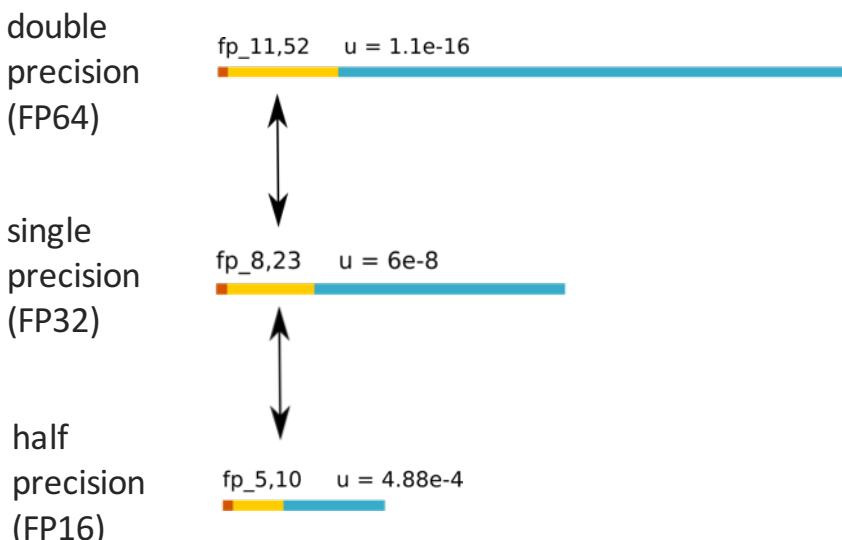
## 2. Combine different precisions in computations

```
// Hp * Hp -> Hp
auto hp_sec = timing(exec, hp_A, hp_b, hp_x);
// Lp * Lp -> Lp
auto lp_sec = timing(exec, lp_A, lp_b, lp_x);
// Hp * Lp -> Hp
auto hplp_sec = timing(exec, hp_A, lp_b, hplp_x);
// Lp * Hp -> Hp
auto lpplp_sec = timing(exec, lp_A, hp_b, lpplp_x);
// Lp * Hp -> Hp
auto lphp_sec = timing(exec, lp_A, hp_b, lphp_x);
```

## 3. Investigate rounding effects

High Precision time(s): 2.0568800000e-05  
High Precision result norm: 1.7725534898e+05  
Low Precision time(s): 2.0955600000e-05  
Low Precision relative error: 9.1052887738e-08  
Hp \* Lp -> Hp time(s): 2.1186100000e-05  
Hp \* Lp -> Hp relative error: 3.7799774251e-08  
Lp \* Lp -> Hp time(s): 2.0312300000e-05  
Lp \* Lp -> Hp relative error: 5.7910008031e-08  
Lp \* Hp -> Hp time(s): 2.0312300000e-05  
Lp \* Hp -> Hp relative error: 3.7173133506e-08

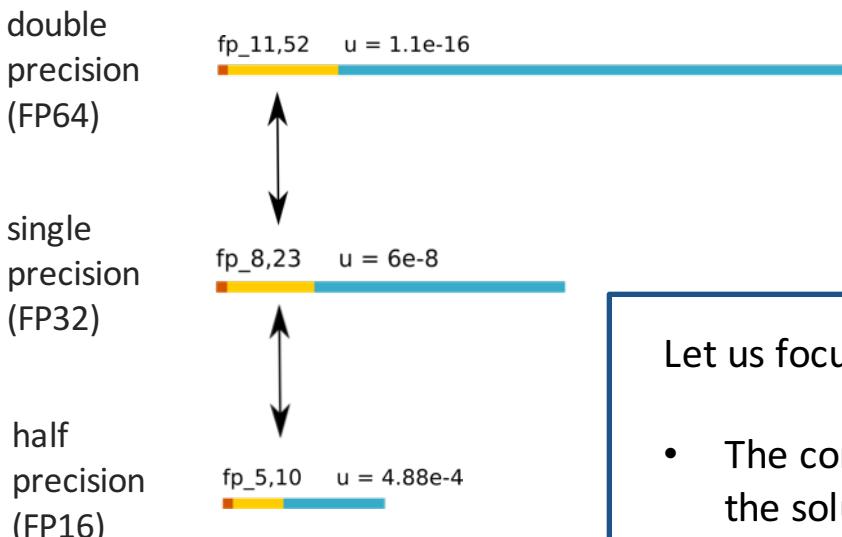
# Background: Floating Point Formats, Accuracy, and Performance



*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;
- **The data access cost linearly depends on the memory volume;**
- **Rounding effects accumulate over a sequence of computations;**

# Background: Floating Point Formats, Accuracy, and Performance



*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;
- **The data access cost linearly depends on the memory volume;**
- **Rounding effects accumulate over a sequence of computations;**

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:

$$\text{relative residual accuracy} = (\text{unit round-off}) * (\text{linear system's condition number})$$

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

# Running iterative methods in different precision formats



Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^4$

## Double Precision

```
Initial residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
111.127
Final residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
5.0775e-10
CG iteration count:    1231
CG execution time [ms]: 140.038
```

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

# Running iterative methods in different precision formats



Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^4$

## Double Precision

```
Initial residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
111.127
Final residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
5.0775e-10
Accuracy improvement ~1012
CG iteration count: 1231
CG execution time [ms]: 140.038
```

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

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



Experiments based on the Ginkgo library <https://ginkgo-project.github.io/>

[ginkgo/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp](https://github.com/GinkgoProject/ginkgo/blob/main/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp)

# Running iterative methods in different precision formats



Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^4$

Double Precision

```
Initial residual norm sqrt(r^T r):  
%%MatrixMarket matrix array real general  
1 1  
111.127  
Final residual norm sqrt(r^T r):  
%%MatrixMarket matrix array real general  
1 1  
5.0775e-10  
CG iteration count: 1231  
CG execution time [ms]: 140.038
```

...

- `ValueType = double; //u=1e-16`
- + `ValueType = float; //u=1e-7`

...

Accuracy improvement  $\sim 10^{12}$

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

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



Experiments based on the Ginkgo library <https://ginkgo-project.github.io/>

[ginkgo/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp](https://github.com/GinkgoProject/ginkgo/blob/main/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp)

# Running iterative methods in different precision formats



Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^4$

## Double Precision

```
Initial residual norm sqrt(r^T r):  
%%MatrixMarket matrix array real general  
1 1  
111.127  
Final residual norm sqrt(r^T r):  
%%MatrixMarket matrix array real general  
1 1  
5.0775e-10 Accuracy improvement ~1012  
CG iteration count: 1231  
CG execution time [ms]: 140.038
```

## Single Precision

```
Initial residual norm sqrt(r^T r):  
%%MatrixMarket matrix array real general  
1 1  
111.127  
Final residual norm sqrt(r^T r):  
%%MatrixMarket matrix array real general  
1 1  
0.179829 Accuracy improvement ~103  
CG iteration count: 1234  
CG execution time [ms]: 127.152
```

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

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



Experiments based on the Ginkgo library <https://ginkgo-project.github.io/>

[ginkgo/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp](https://github.com/GinkgoProject/ginkgo/blob/main/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp)

# Running iterative methods in different precision formats

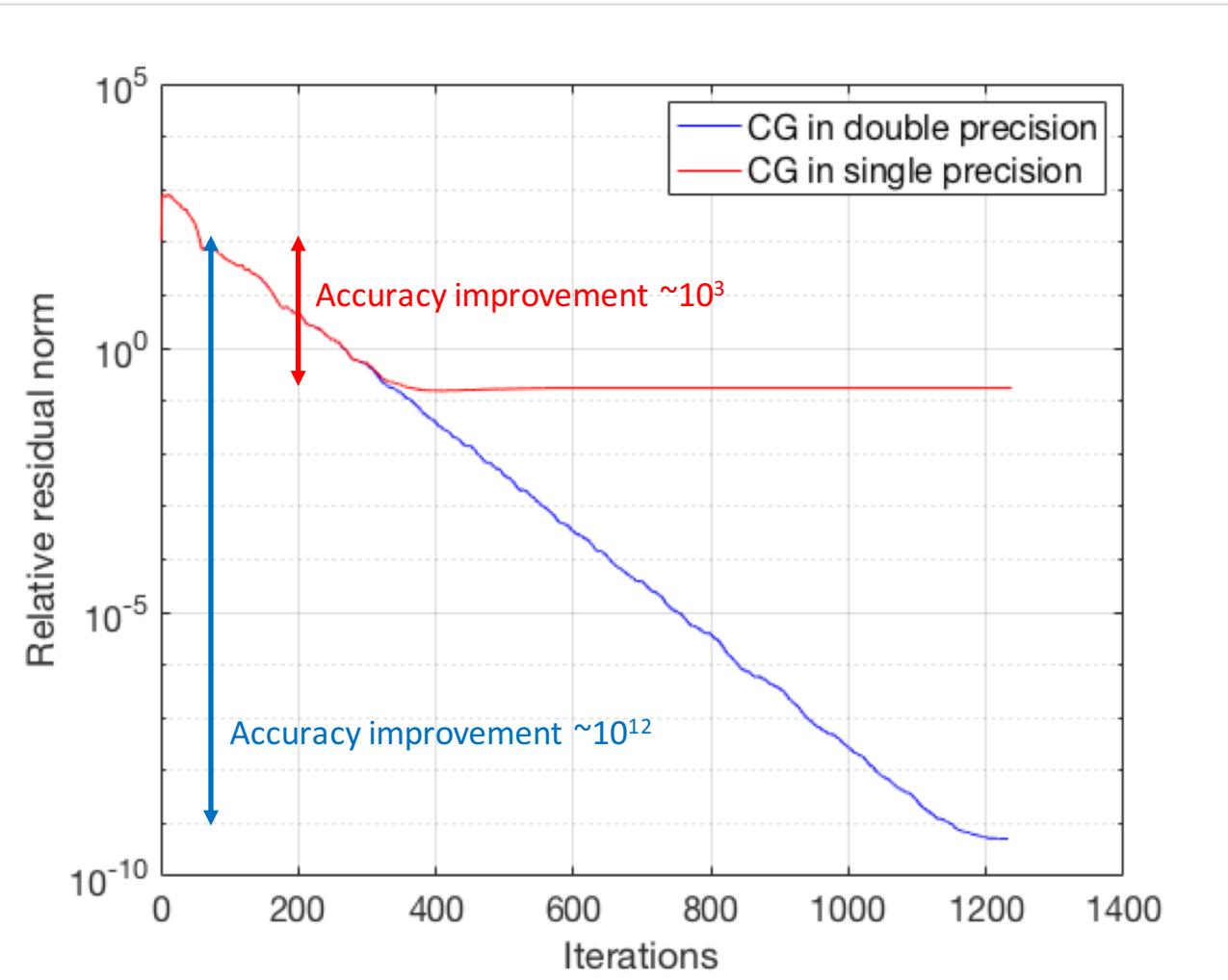


Linear System  $Ax=b$  with

Double Precision

```
Initial residual norm: 111.127
%%MatrixMarket mat
1 1
Final residual norm: 5.0775e-10
%%MatrixMarket mat
1 1
CG iteration count: 1234
CG execution time: 127.152
```

A purple curved arrow points from the text "CG execution time" to the value "127.152".



```
rt(r^T r):
ray real general

(r^T r):
ray real general

Improvement ~ $10^3$ 
1234
127.152
```

Experiments based on the Ginkgo library <https://ginkgo-project.github.io/>

[ginkgo/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp](https://github.com/ginkgo-project/ginkgo/blob/main/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp)

# Running iterative methods in different precision formats



Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^4$

## Double Precision

```
Initial residual norm sqrt(r^T r):  
%%MatrixMarket matrix array real general  
1 1  
111.127  
Final residual norm sqrt(r^T r):  
%%MatrixMarket matrix array real general  
1 1  
5.0775e-10  
CG iteration count: 1231  
CG execution time [ms]: 140.038
```

## Single Precision

```
Initial residual norm sqrt(r^T r):  
%%MatrixMarket matrix array real general  
1 1  
111.127  
Final residual norm sqrt(r^T r):  
%%MatrixMarket matrix array real general  
1 1  
0.179829  
CG iteration count: 1234  
CG execution time [ms]: 127.152
```

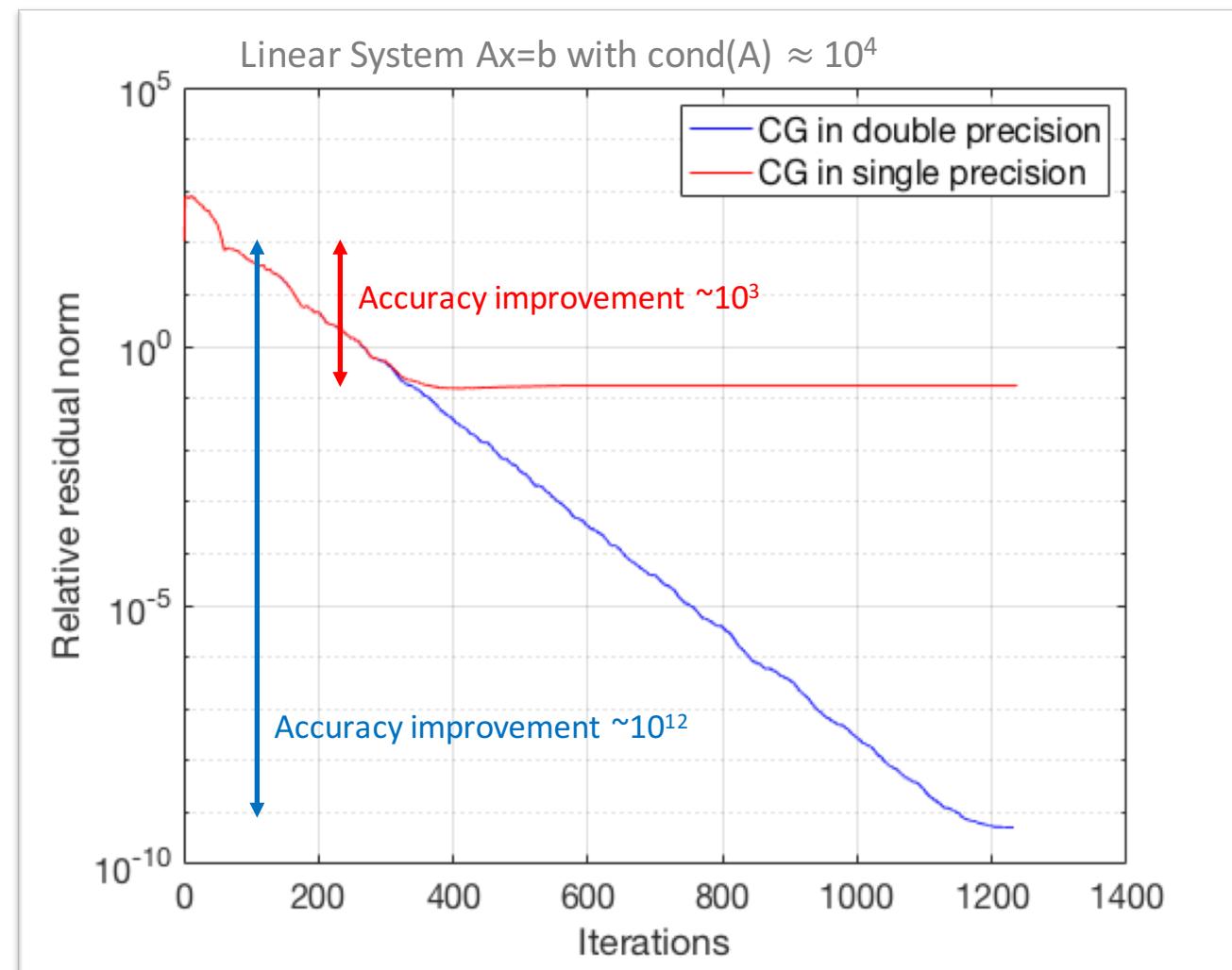
Single Precision is 10% faster!

Experiments based on the Ginkgo library <https://ginkgo-project.github.io/>

[ginkgo/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp](https://github.com/GinkgoProject/ginkgo/blob/main/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp)

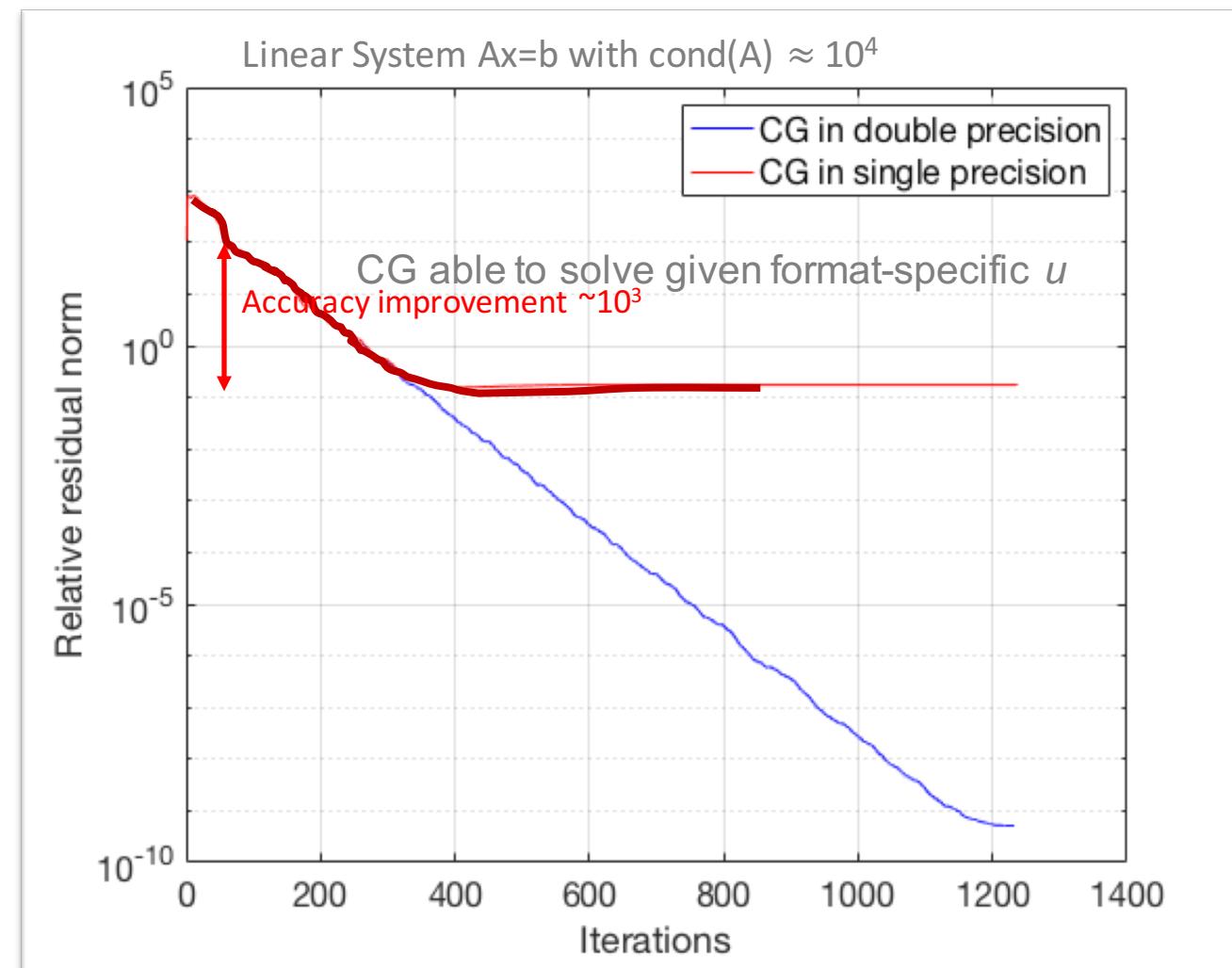
## **Combining precision formats: Mixed Precision Iterative Refinement**

What if we could extend the “nice” convergence of the single precision solver to the complete iteration process?



## Combining precision formats: Mixed Precision Iterative Refinement

What if we could extend the “nice” convergence of the single precision solver to the complete iteration process?



# Combining precision formats: Mixed Precision Iterative Refinement

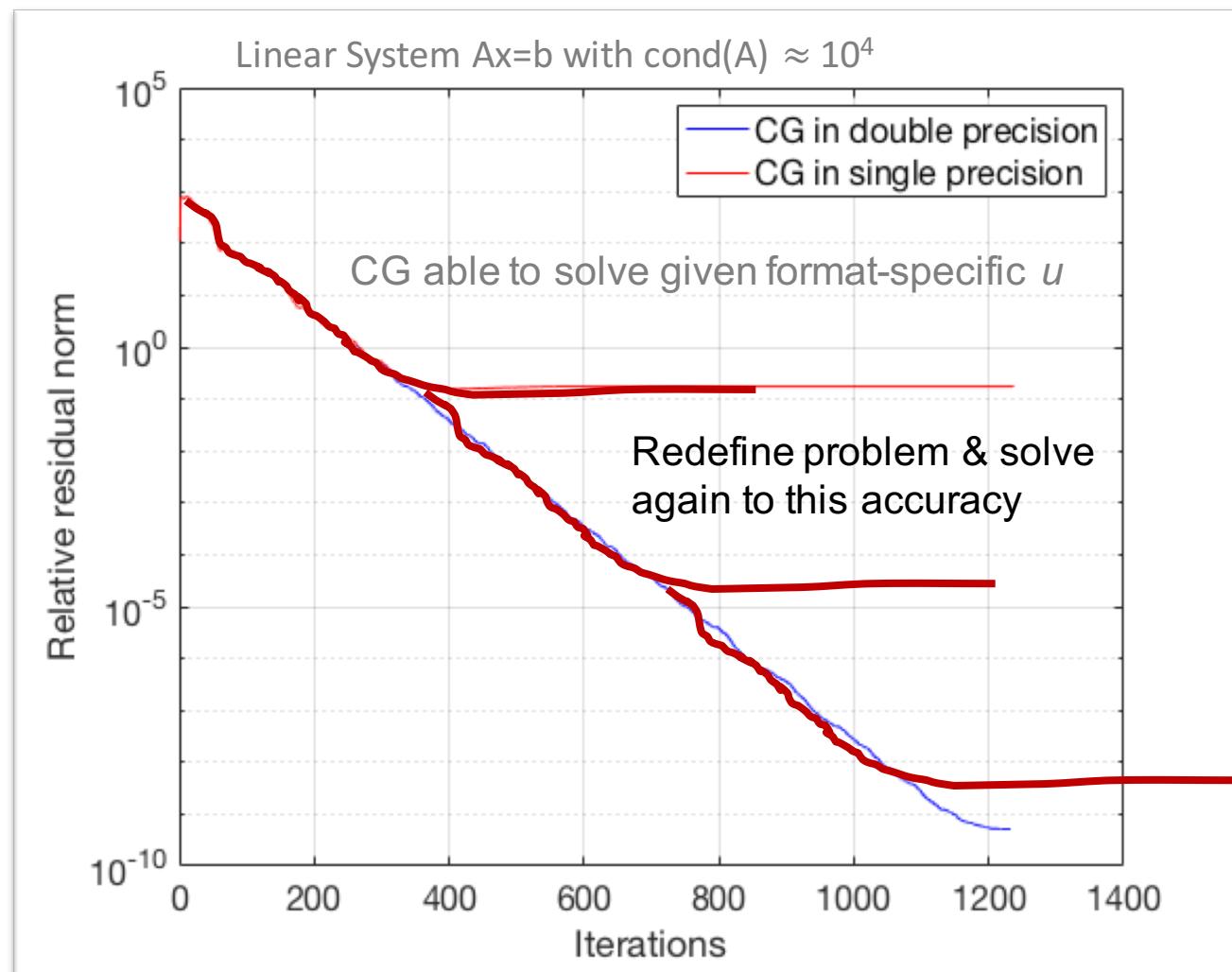
What if we could extend the “nice” convergence of the single precision solver to the complete iteration process?

## Mixed Precision Iterative Refinement

For an approximate solution  $x^{(k)}$ , the residual computes as  $r = b - Ax^{(k)}$ .

The exact solution for  $Ax = b$  is  $x = x^{(k)} + c$  where  $c$  is the solution of  $Ac = r$ .

Take  $Ac = r$  be the redefined problem for the next step.



# Combining precision formats: Mixed Precision Iterative Refinement

What if we could extend the “nice” convergence of the single precision solver to the complete iteration process?

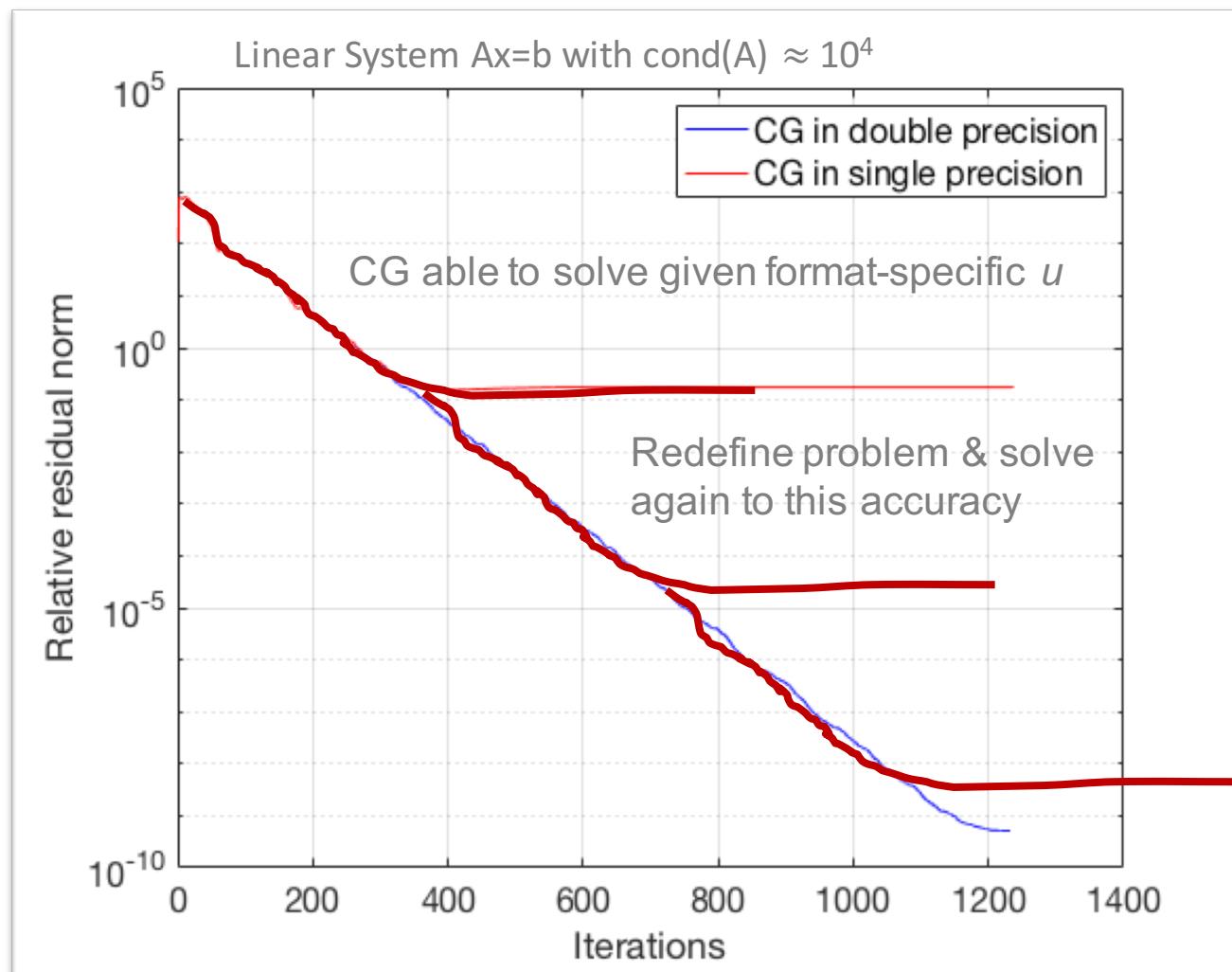
## Mixed Precision Iterative Refinement

For an approximate solution  $x^{(k)}$ , the residual computes as  $r = b - Ax^{(k)}$ .

The exact solution for  $Ax = b$  is  $x = x^{(k)} + c$  where  $c$  is the solution of  $Ac = r$ .

Take  $Ac = r$  be the redefined problem for the next step.

```
Choose initial guess x      high precision
do {
    Compute r = b - Ax  high precision
    Solve A * c = r    low precision
    Update x = x + c  high precision
} while ( ||r|| > tol )    high precision
```



# Combining precision formats: Mixed Precision Iterative Refinement

Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^4$

## Double Precision Iterative Refinement

```
Initial residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
111.127
Final residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
7.16102e-11 Rel. Residual ~10^-14
MPIR iteration count: 18
MPIR execution time [ms]: 213.491
```

Experiments based on the Ginkgo library <https://ginkgo-project.github.io/ginkgo/examples/mixed-precision-ir/mixed-precision-ir.cpp>

# Combining precision formats: Mixed Precision Iterative Refinement

Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^4$

## Double Precision Iterative Refinement

```
Initial residual norm sqrt(r^T r):  
%MatrixMarket matrix array real general  
1 1  
111.127  
Final residual norm sqrt(r^T r):  
%MatrixMarket matrix array real general  
1 1  
7.16102e-11 Rel. Residual ~10-14  
MPIR iteration count: 18  
MPIR execution time [ms]: 213.491
```

## Mixed Precision Iterative Refinement

```
Initial residual norm sqrt(r^T r):  
%MatrixMarket matrix array real general  
1 1  
111.127  
Final residual norm sqrt(r^T r):  
%MatrixMarket matrix array real general  
1 1  
7.41333e-11 Rel. Residual ~10-14  
MPIR iteration count: 18  
MPIR execution time [ms]: 183.296
```

Single precision inner CG

Experiments based on the Ginkgo library <https://ginkgo-project.github.io/ginkgo/examples/mixed-precision-ir/mixed-precision-ir.cpp>

# Combining precision formats: Mixed Precision Iterative Refinement

Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^4$

## Double Precision Iterative Refinement

```
Initial residual norm sqrt(r^T r):  
%>MatrixMarket matrix array real general  
1 1  
111.127  
Final residual norm sqrt(r^T r):  
%>MatrixMarket matrix array real general  
1 1  
7.16102e-11 Rel. Residual ~10^-14  
MPIR iteration count: 18  
MPIR execution time [ms]: 213.491
```

## Mixed Precision Iterative Refinement

```
Initial residual norm sqrt(r^T r):  
%>MatrixMarket matrix array real general  
1 1  
111.127  
Final residual norm sqrt(r^T r):  
%>MatrixMarket matrix array real general  
1 1  
7.41333e-11 Rel. Residual ~10^-14  
MPIR iteration count: 18  
MPIR execution time [ms]: 183.296
```

Single precision inner CG

16% runtime improvement

Experiments based on the Ginkgo library <https://ginkgo-project.github.io/ginkgo/examples/mixed-precision-ir/mixed-precision-ir.cpp>

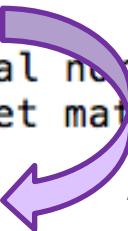
# Running iterative methods in different precision formats



Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^7$  ( 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  $\sim 10^9$

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

# Running iterative methods in different precision formats



Linear System Ax=b with  $\text{cond}(A) \approx 10^7$  (double)

Double Precision CG + Double Precision BLAS

```
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
```

Accuracy improvement  $\sim 10^9$

```
CG iteration count: 4797
```

```
CG execution time [ms]: 2971.18
```

...

- `ValueType = double; //u=1e-16`
- + `ValueType = float; //u=1e-7`

...

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

# Running iterative methods in different precision formats



Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^7$  ( 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 Accuracy improvement ~109  
CG iteration count: 4797  
CG execution time [ms]: 2971.18
```

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 No improvement  
CG iteration count: 8887  
CG execution time [ms]: 2972.46
```

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

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



Experiments based on the Ginkgo library <https://ginkgo-project.github.io/>

[ginkgo/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp](https://github.com/GinkgoProject/ginkgo/blob/main/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp)

# Why are we faster with a single precision CG solver?

														double : single : half		
2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	
NVIDIA GPU generation				Tesla	Fermi	Kepler			Maxwell		Pascal			Volta		Ampere
Rel. compute performance				1 : 2	1 : 2		1 : 2		1 : 32		1 : 2 : 4		1 : 2 : 16*		1 : 2 : 32 1* : 8* : 32*	
Rel. memory performance				1 : 2	1 : 2		1 : 2		1 : 2		1 : 2 : 4		1 : 2 : 4		1 : 2 : 4	

For **compute-bound applications**, the performance gains from using lower precision **depend on the architecture**.

*Differences between Volta, Maxwell, Tesla...*

For **memory-bound applications**, the performance gains from using lower precision are **architecture-independent** and correspond to the floating point format complexity (#bits).

*Generally, 2x for FP32, 4x for FP16.*

\*Tensor cores

$$\mathbf{D} = \begin{pmatrix} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} & \mathbf{A}_{0,2} & \mathbf{A}_{0,3} \\ \mathbf{A}_{1,0} & \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \mathbf{A}_{1,3} \\ \mathbf{A}_{2,0} & \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \mathbf{A}_{2,3} \\ \mathbf{A}_{3,0} & \mathbf{A}_{3,1} & \mathbf{A}_{3,2} & \mathbf{A}_{3,3} \end{pmatrix} \begin{pmatrix} \mathbf{B}_{0,0} & \mathbf{B}_{0,1} & \mathbf{B}_{0,2} & \mathbf{B}_{0,3} \\ \mathbf{B}_{1,0} & \mathbf{B}_{1,1} & \mathbf{B}_{1,2} & \mathbf{B}_{1,3} \\ \mathbf{B}_{2,0} & \mathbf{B}_{2,1} & \mathbf{B}_{2,2} & \mathbf{B}_{2,3} \\ \mathbf{B}_{3,0} & \mathbf{B}_{3,1} & \mathbf{B}_{3,2} & \mathbf{B}_{3,3} \end{pmatrix} + \begin{pmatrix} \mathbf{C}_{0,0} & \mathbf{C}_{0,1} & \mathbf{C}_{0,2} & \mathbf{C}_{0,3} \\ \mathbf{C}_{1,0} & \mathbf{C}_{1,1} & \mathbf{C}_{1,2} & \mathbf{C}_{1,3} \\ \mathbf{C}_{2,0} & \mathbf{C}_{2,1} & \mathbf{C}_{2,2} & \mathbf{C}_{2,3} \\ \mathbf{C}_{3,0} & \mathbf{C}_{3,1} & \mathbf{C}_{3,2} & \mathbf{C}_{3,3} \end{pmatrix}$$

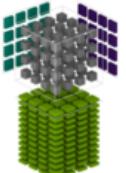
FP16 or FP32  
INT32

HMMA    FP16 or FP32  
IMMA    INT32

FP16  
INT8 or UINT8

FP16  
INT8 or UINT8

FP16 or FP32  
INT32



# Why are we faster with a single precision CG solver?

2006 2007 2008 2009 2010 2011 2012

NVIDIA GPU generation Tesla Fermi Kepler

Rel. compute performance 1 : 2 1 : 2 1 : 2

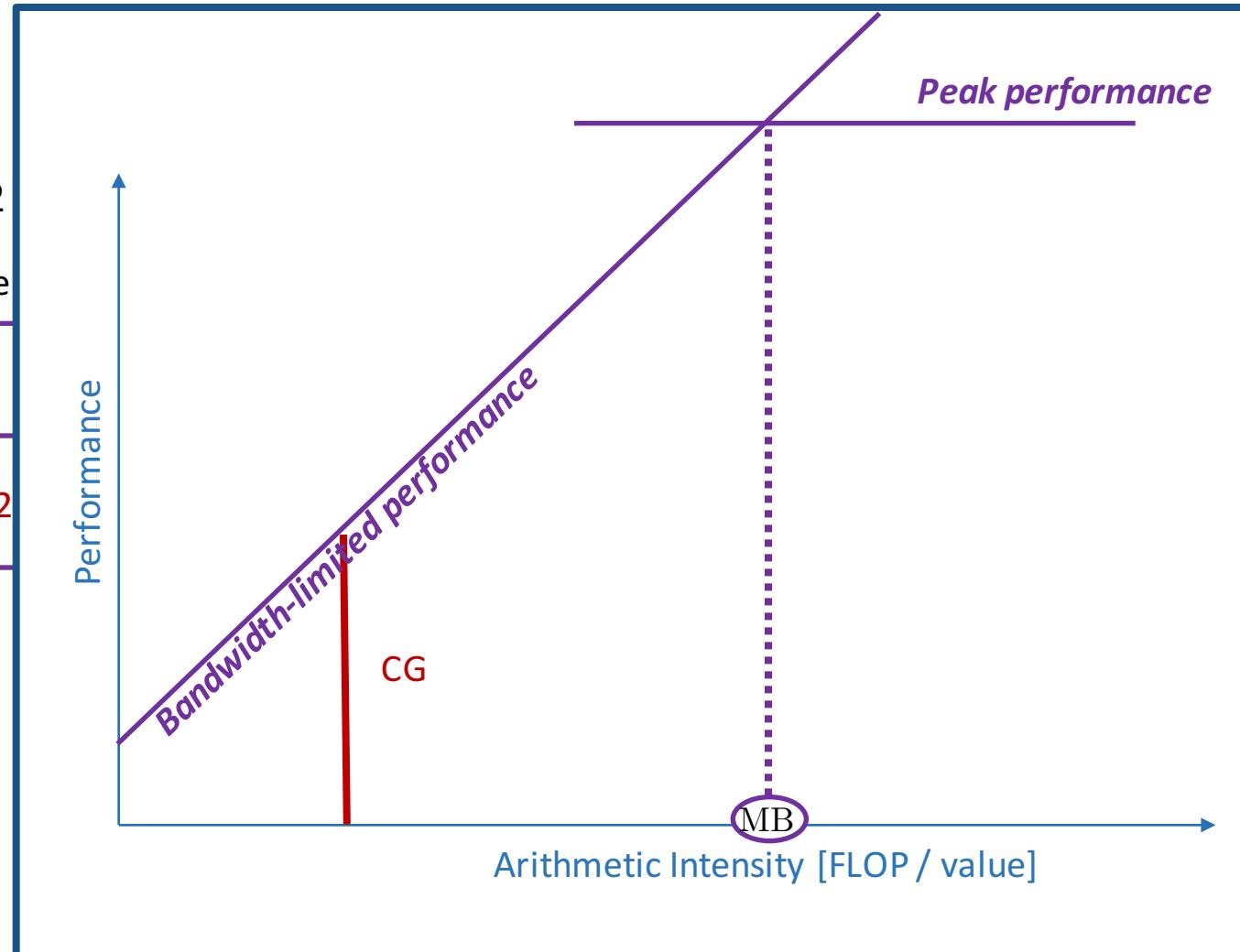
Rel. memory performance 1 : 2 1 : 2 1 : 2

For **compute-bound applications**, the performance gains from using lower precision **depend on the architecture**.

*Differences between Volta, Maxwell, Tesla...*

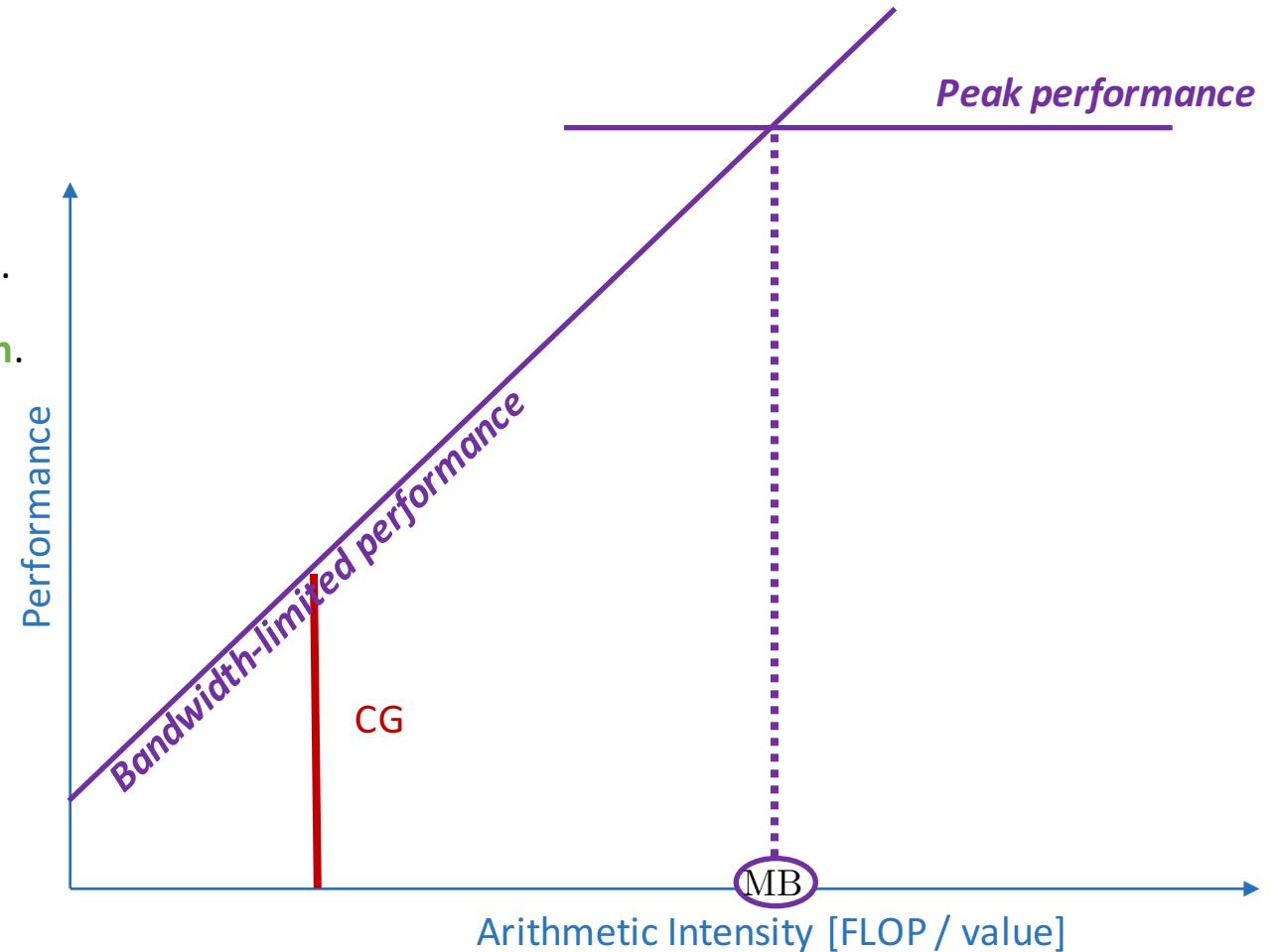
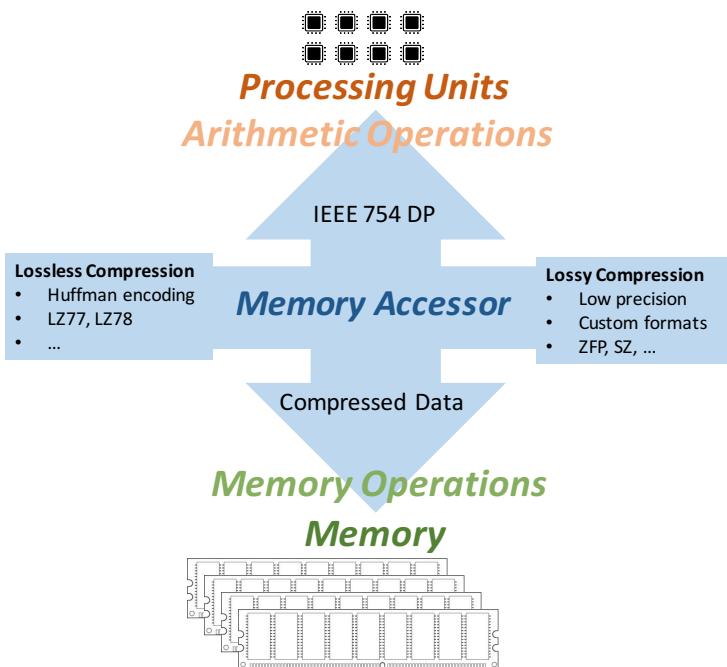
For **memory-bound applications**, the performance gains from using lower precision are **architecture-independent** and correspond to the floating point format complexity (#bits).

*Generally, 2x for FP32, 4x for FP16.*



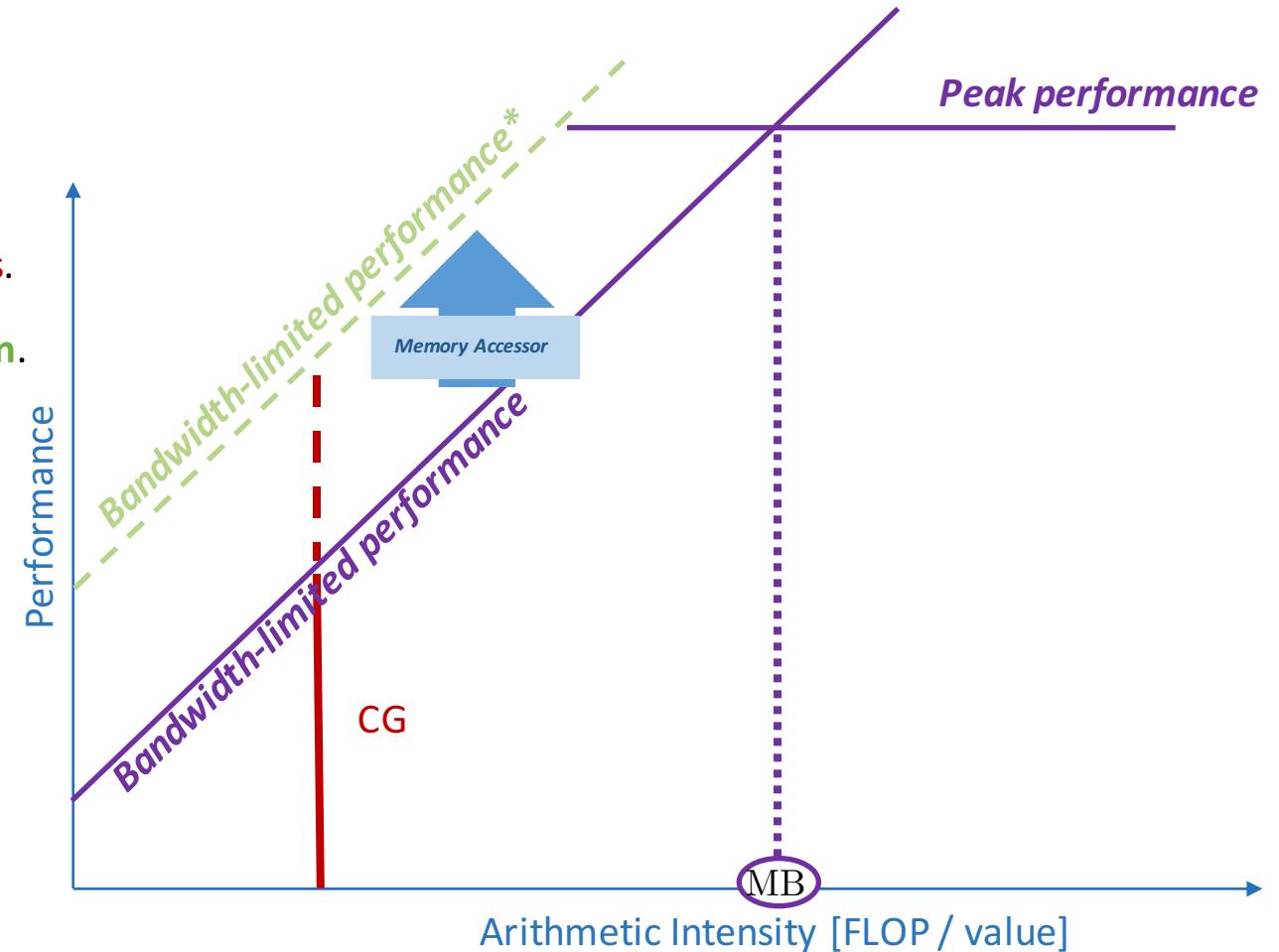
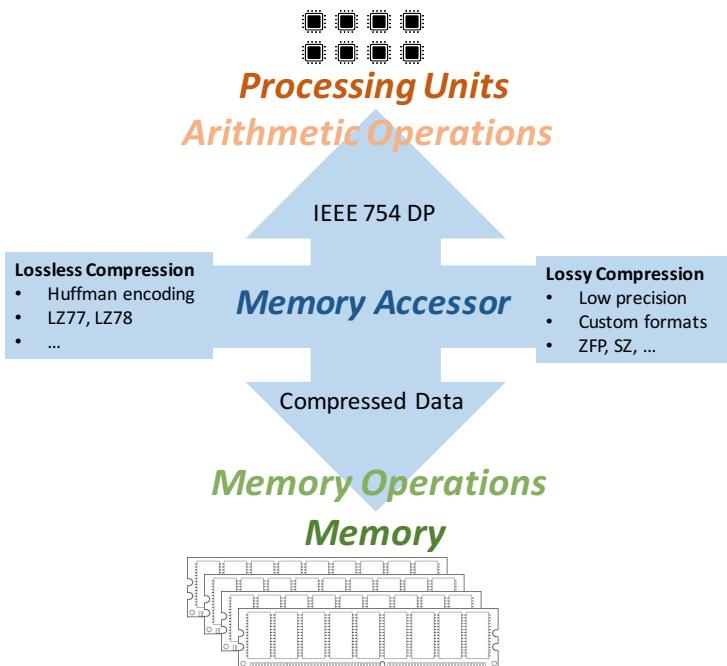
# Decoupling the memory precision from the arithmetic precision

- Traditionally, we use a strong coupling between the precision formats used for **arithmetic operations** and **storing data**.
- The **arithmetic operations** are free, use **high precision formats**.
- Data access** should be as cheap as possible, **reduced precision**.

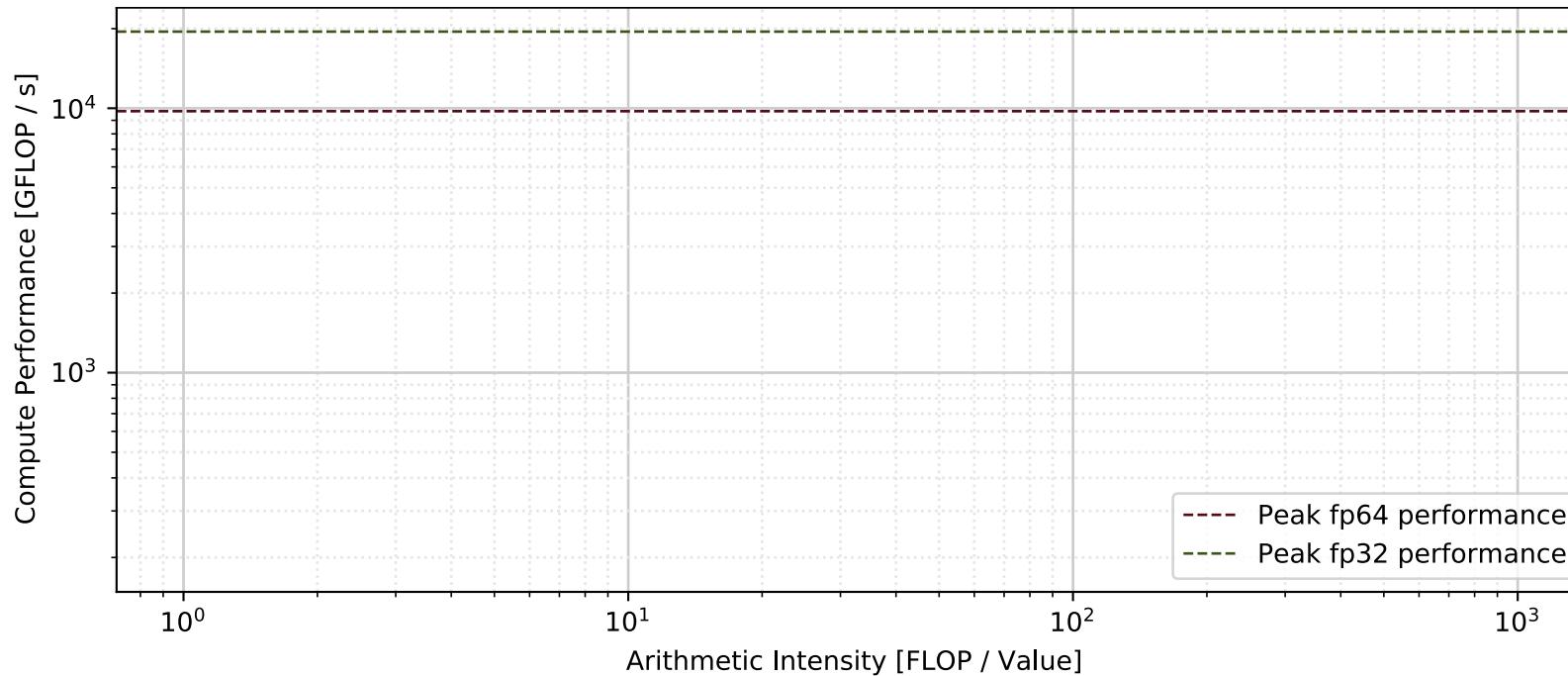


# Decoupling the memory precision from the arithmetic precision

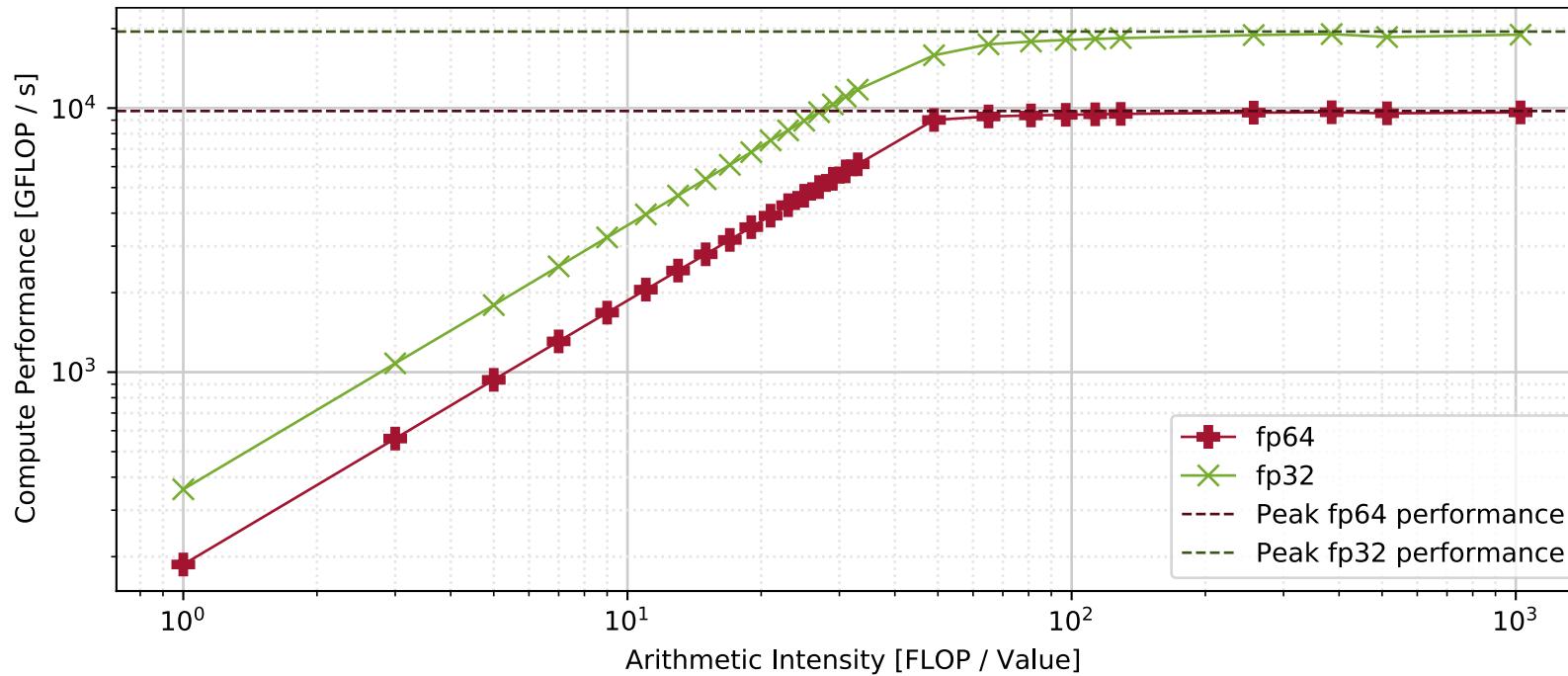
- Traditionally, we use a strong coupling between the precision formats used for **arithmetic operations** and **storing data**.
- The **arithmetic operations** are free, use **high precision formats**.
- Data access** should be as cheap as possible, **reduced precision**.



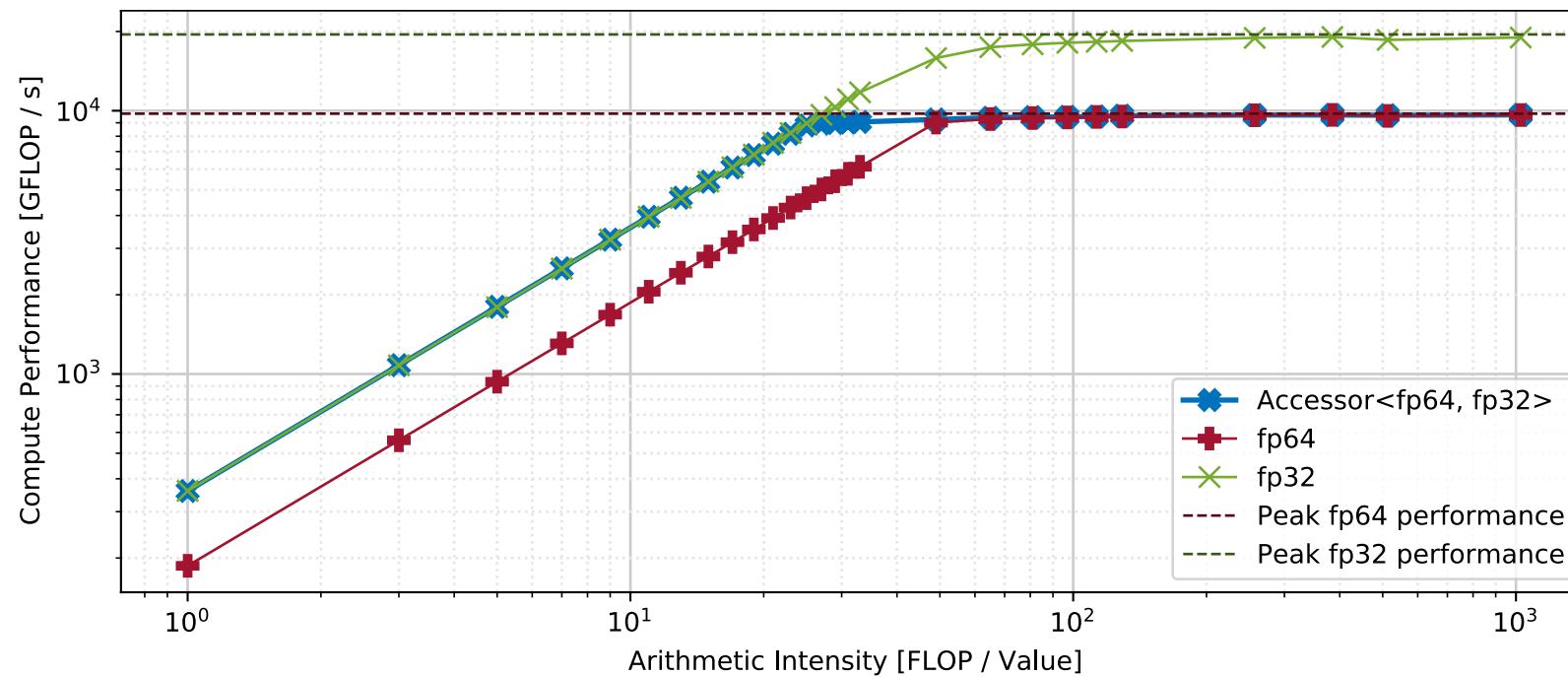
# *Memory Accessor for NVIDIA A100 GPU*



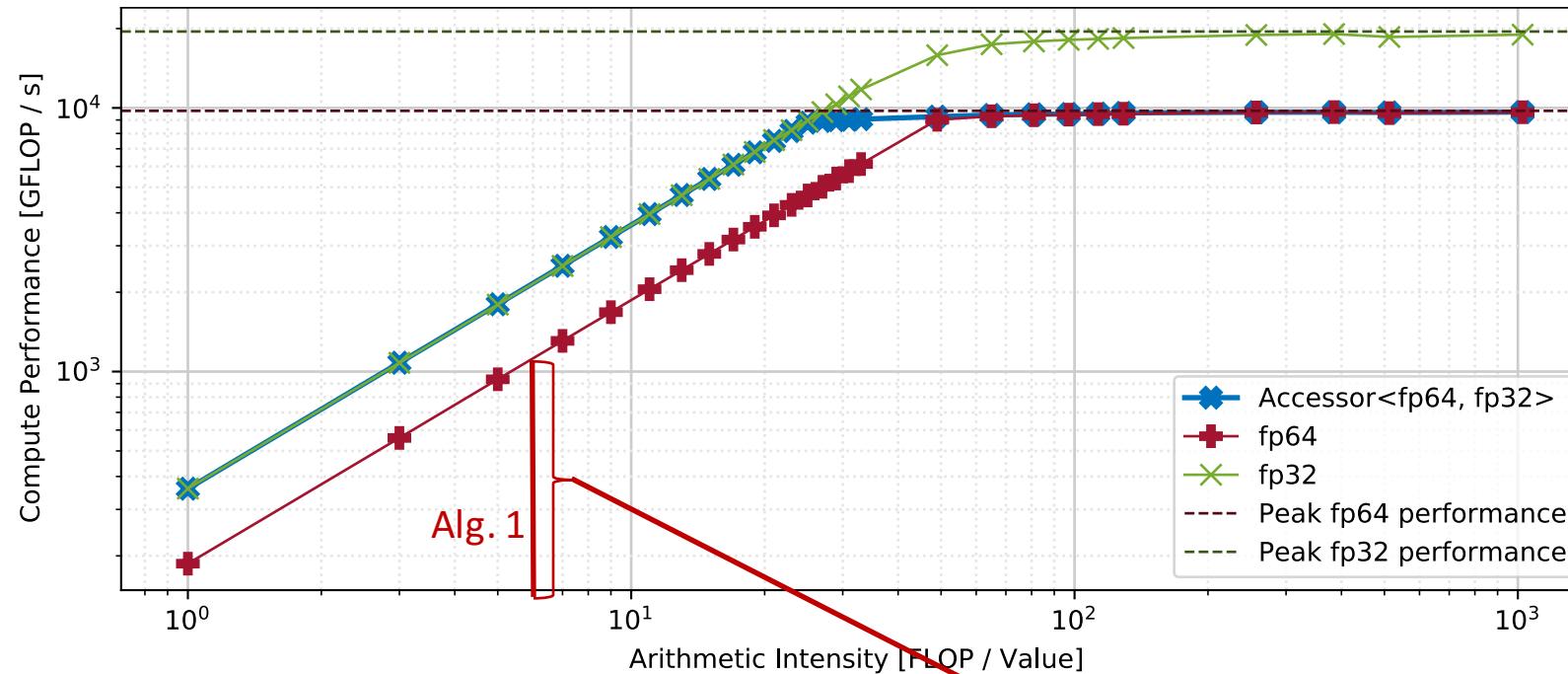
# **Memory Accessor for NVIDIA A100 GPU**



# *Memory Accessor for NVIDIA A100 GPU*



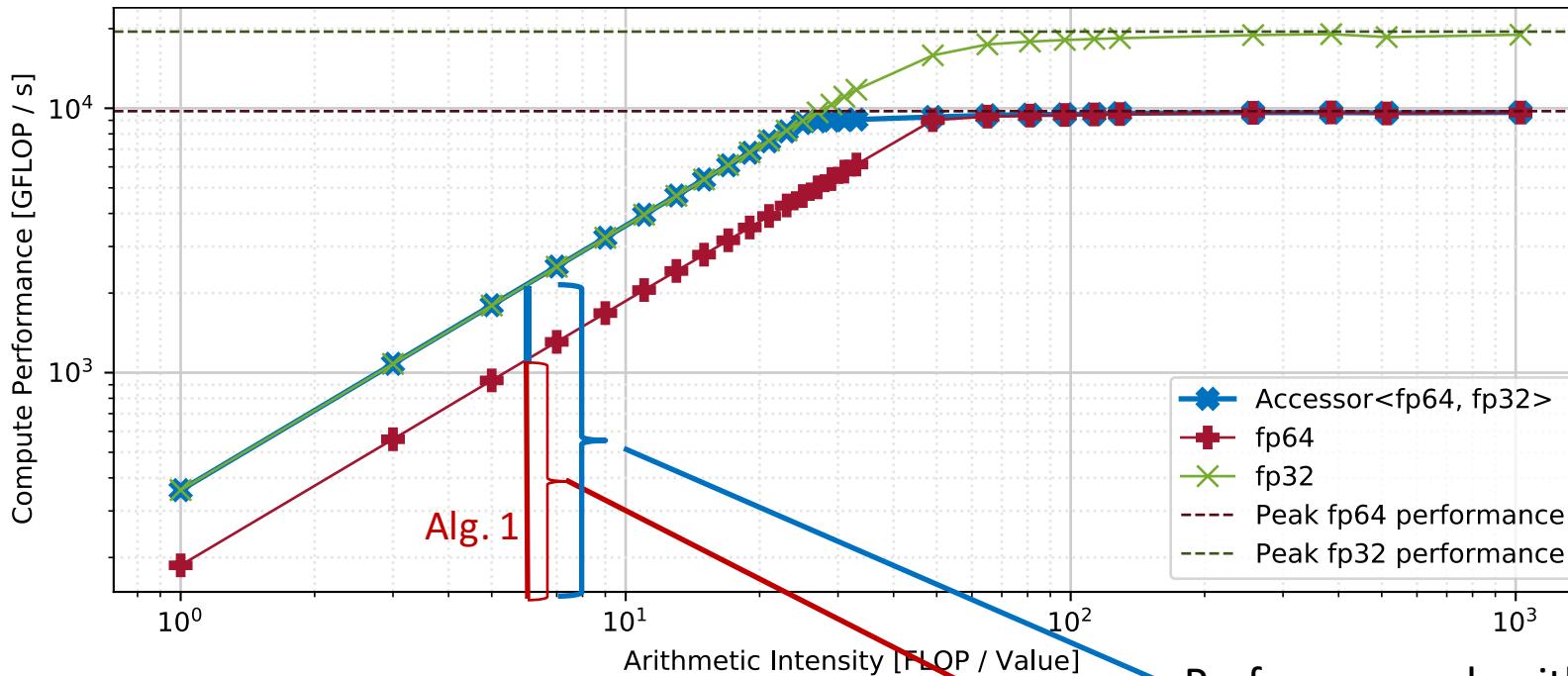
# Memory Accessor for NVIDIA A100 GPU



Alg. 1

Performance algorithm achieves with standard memory access.

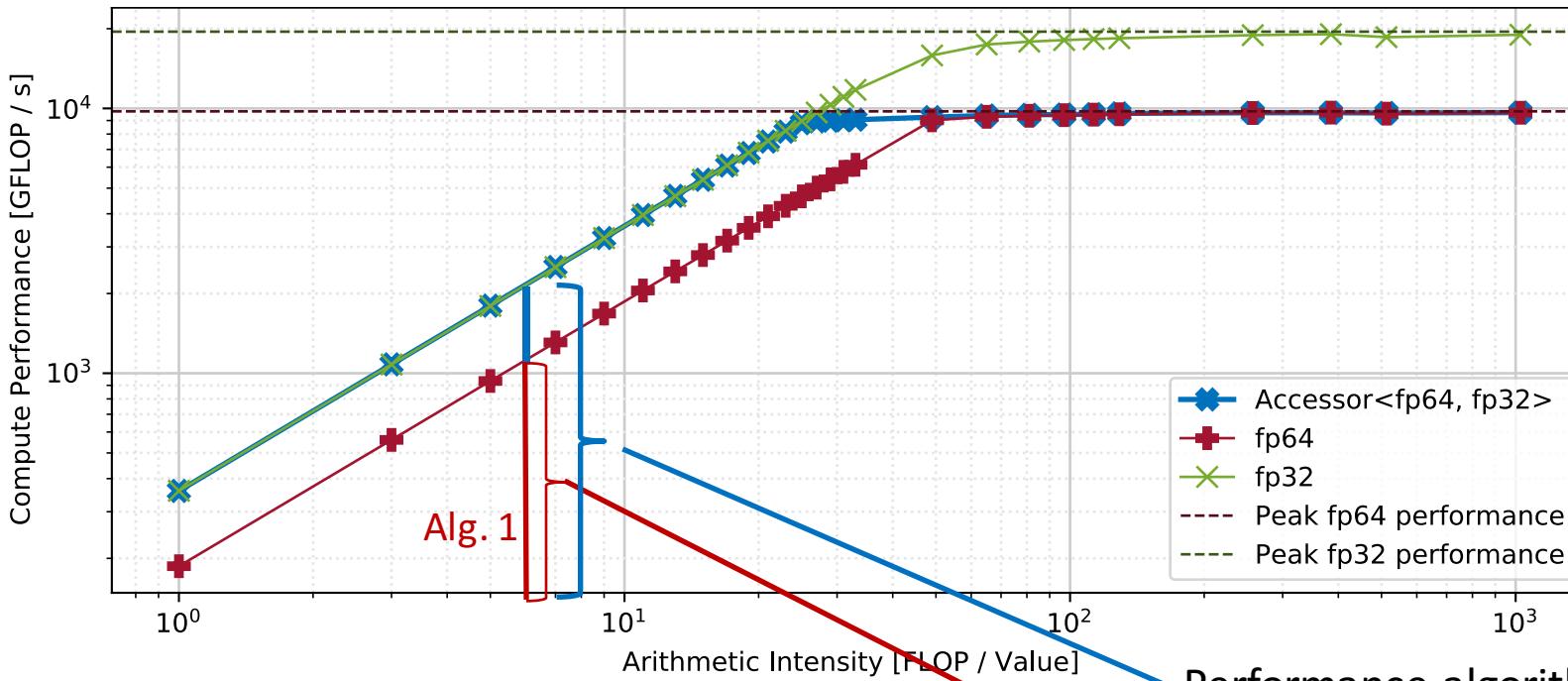
# Memory Accessor for NVIDIA A100 GPU



Performance algorithm achieves with memory accessor.

Performance algorithm achieves with standard memory access.

# Memory Accessor for NVIDIA A100 GPU



Performance algorithm achieves with memory accessor.

Performance algorithm achieves with standard memory access.

Two exploitation strategies:

- Increase accuracy for low precision algorithms by doing arithmetic in higher precision
- Improve performance for algorithms that can compensate information loss induced by data compression

# Accessor-BLAS (BLAS 1 / BLAS 2)

---

## Design

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

## Characteristics

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

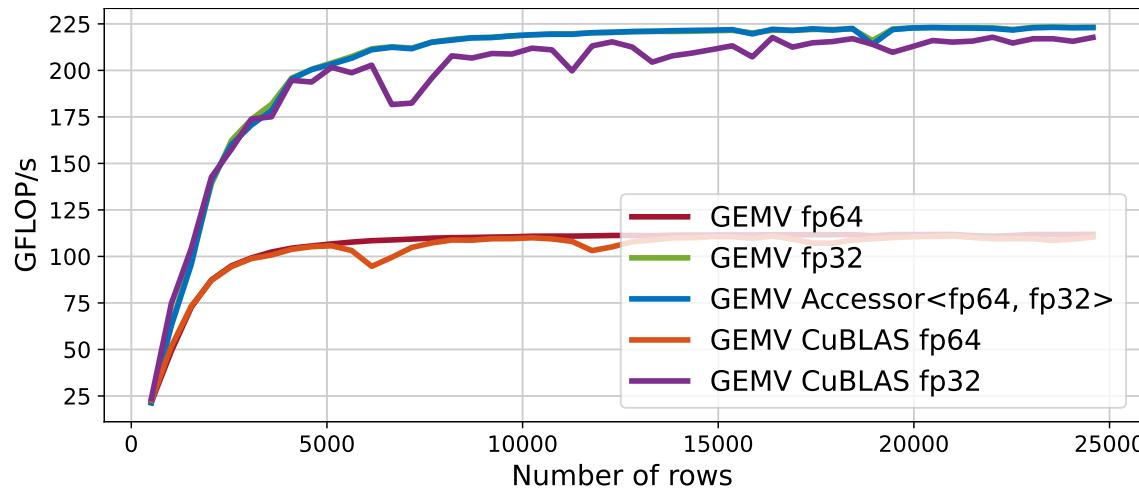
## Usage

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

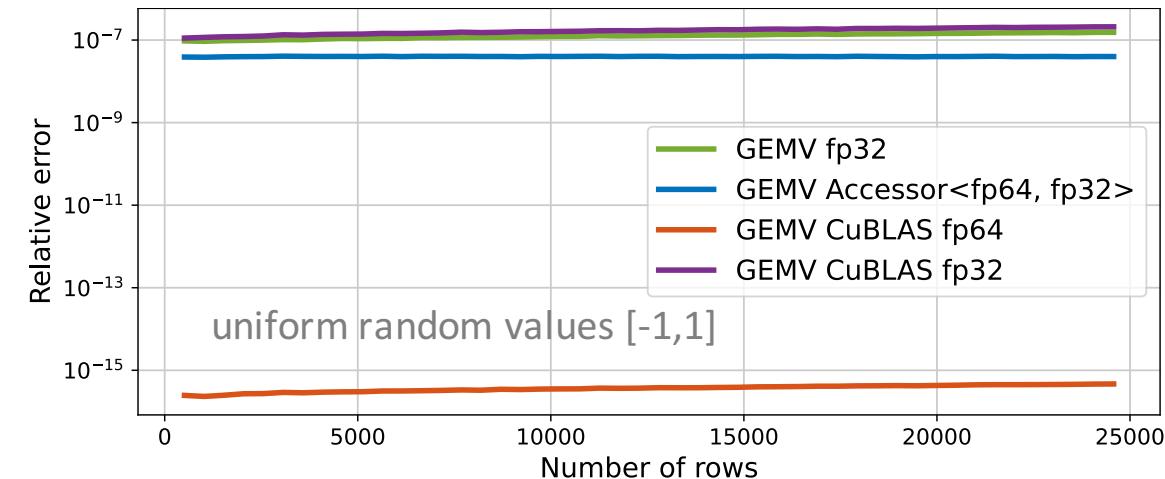
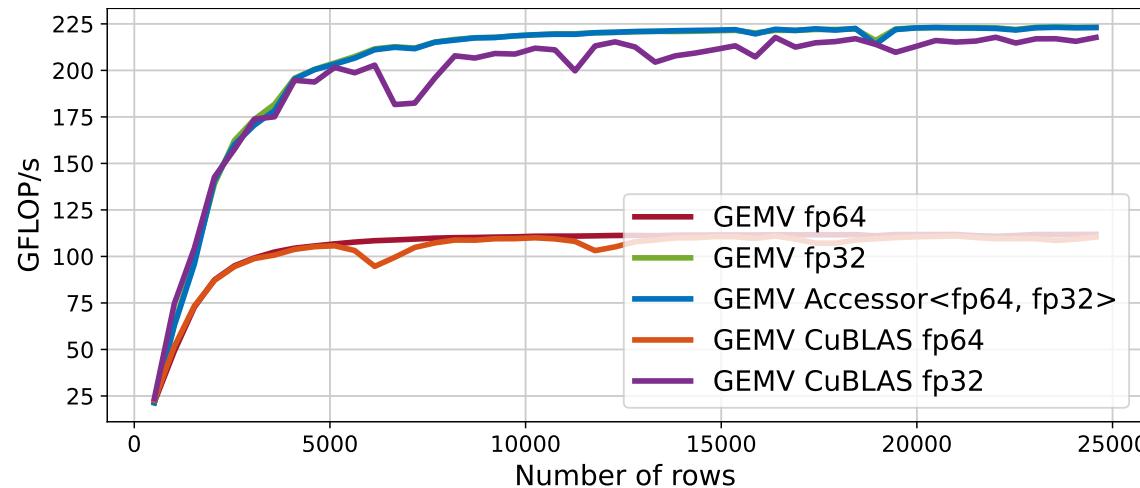
NVIDIA V100 GPU (Summit)



# Accessor-BLAS: Replacing LP BLAS to improve accuracy

## GEMV

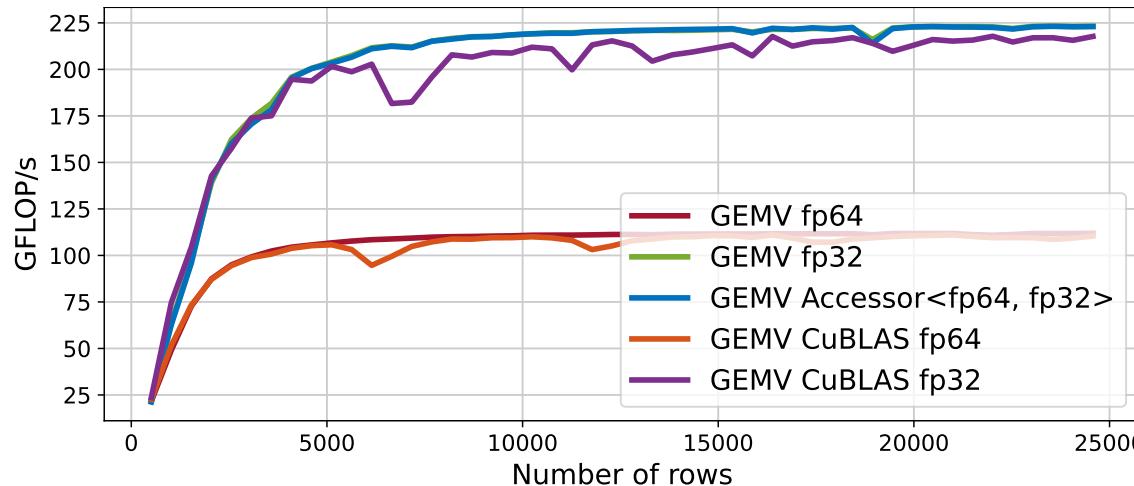
NVIDIA V100 GPU (Summit)



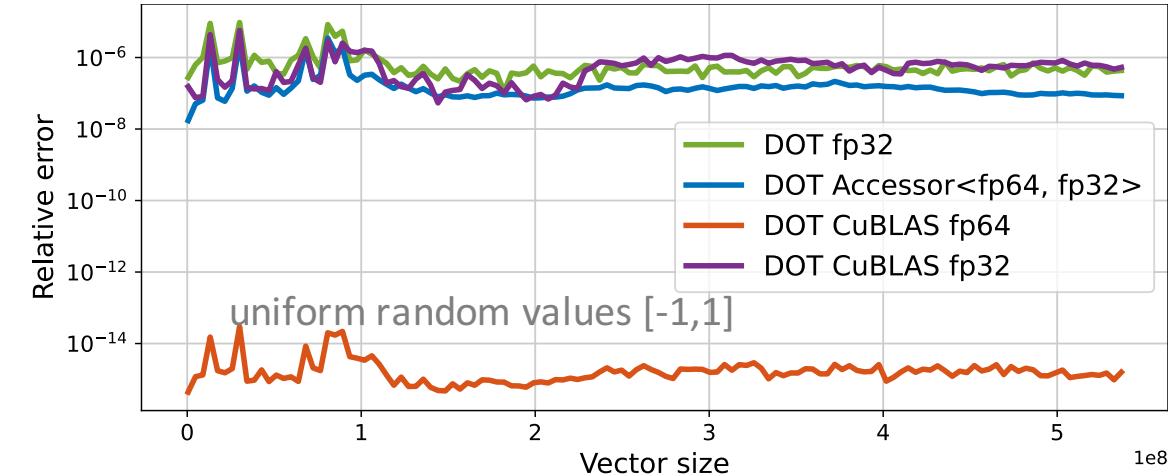
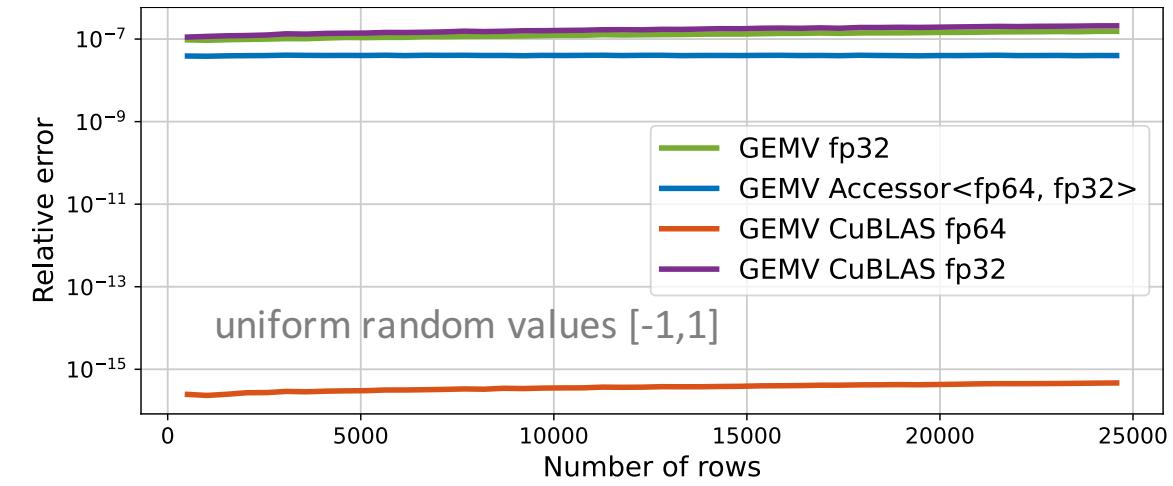
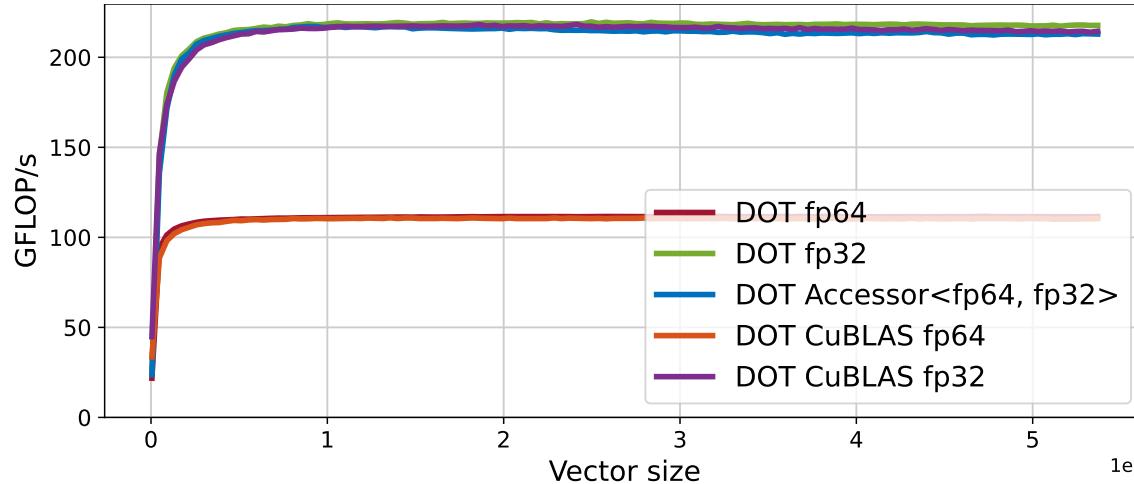
# Accessor-BLAS: Replacing LP BLAS to improve accuracy

## GEMV

NVIDIA V100 GPU (Summit)



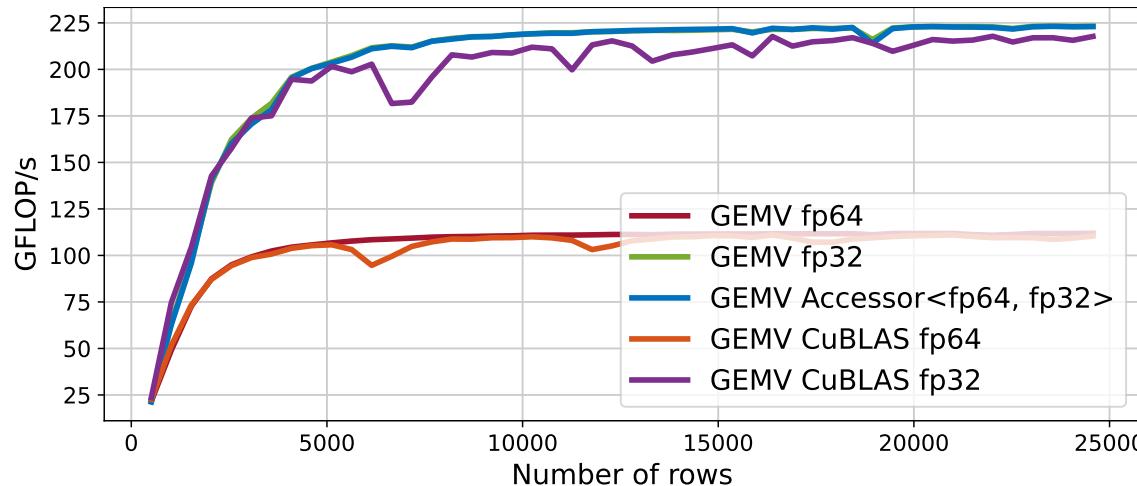
## DOT



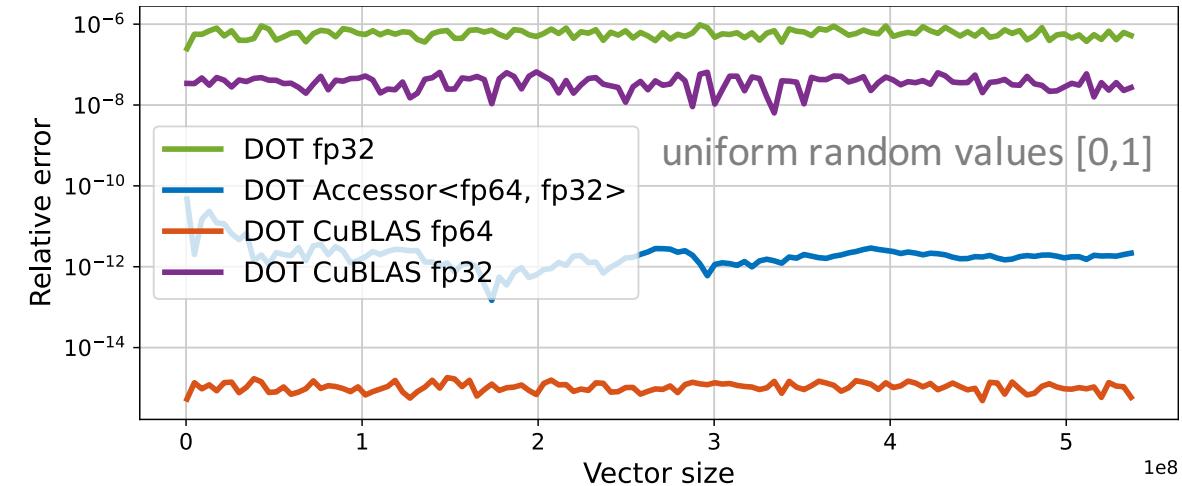
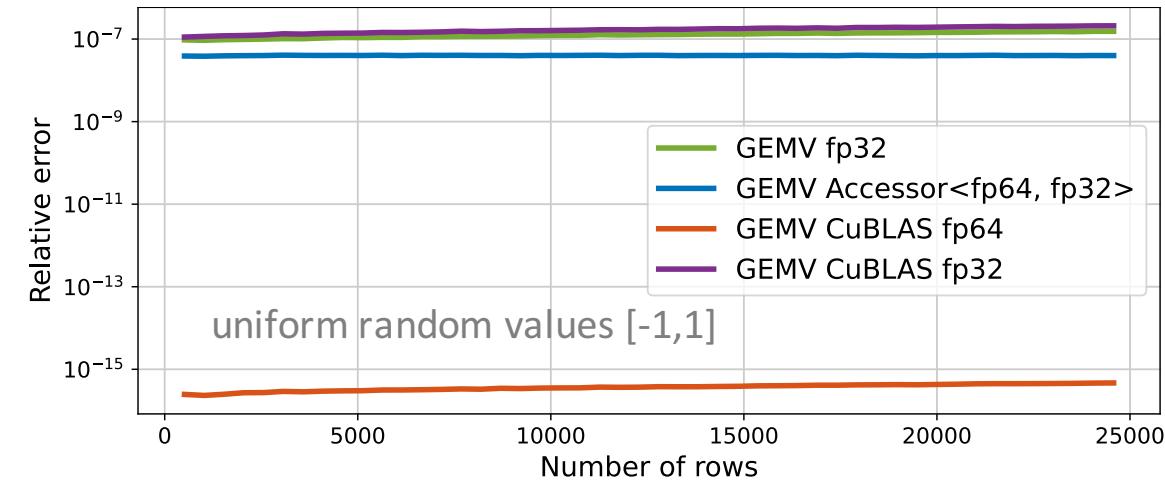
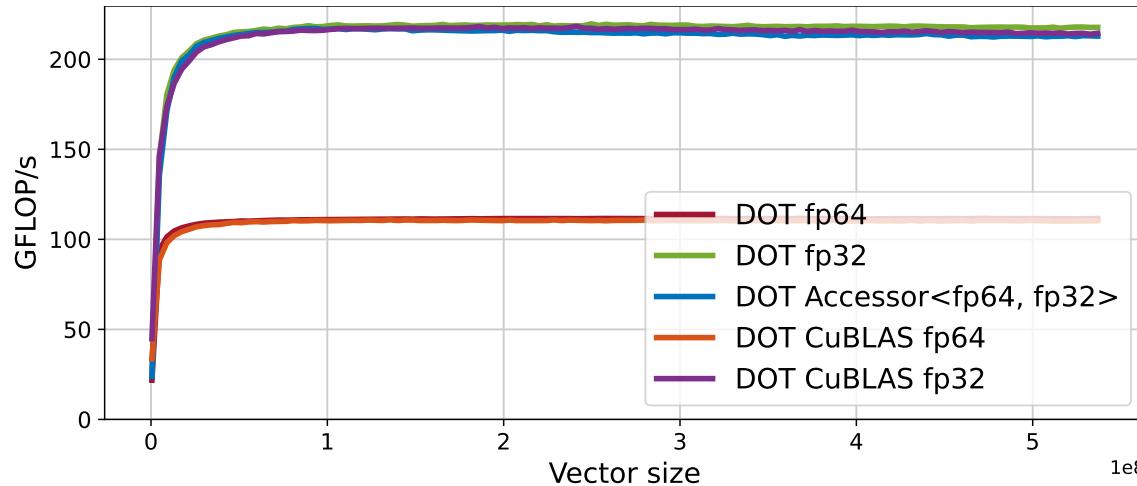
# Accessor-BLAS: Replacing LP BLAS to improve accuracy

## GEMV

NVIDIA V100 GPU (Summit)



## DOT



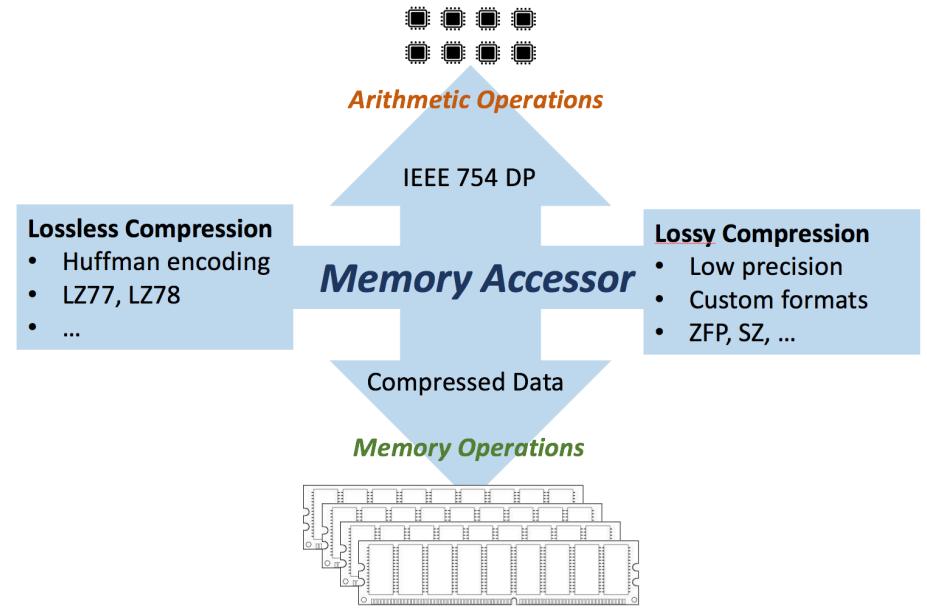
# Rethinking Algorithms: Use the accessor to boost performance

*Can we use the memory accessor to accelerate applications by compressing data without changing the application output?*

- No, not in general.
- Yes, in some cases.
- We need to adapt the approach to the application & data.

*Two possibilities in the context of solving linear systems:*

- “*Self-healing*” iterative methods;
- *Approximate linear operators*;



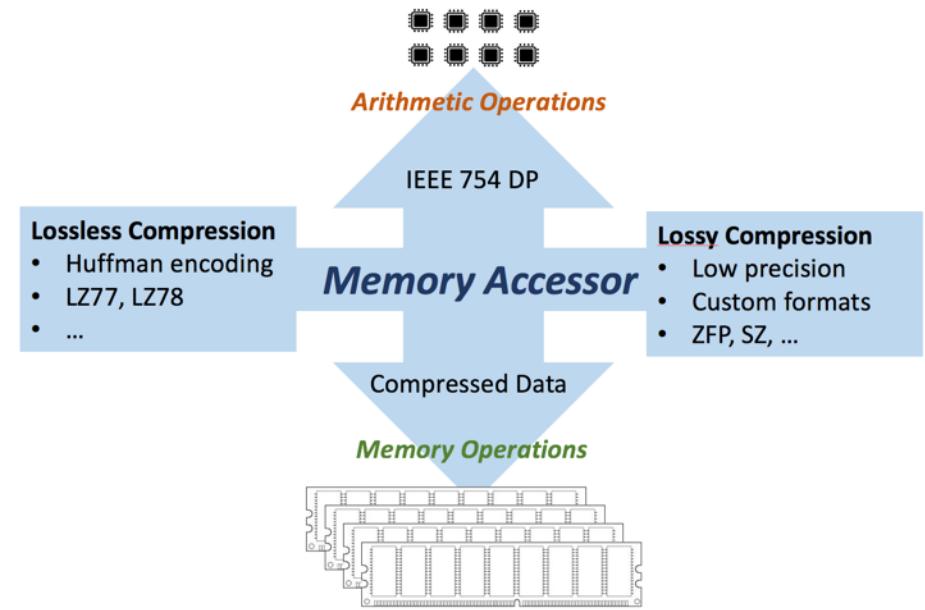
# Rethinking Algorithms: Use the memory accessor to boost performance

Can we use the memory accessor to accelerate applications by compressing data without changing the application output?

- No, not in general.
- Yes, in some cases.
- We need to adapt the approach to the application & data.

*Two possibilities in the context of solving linear systems:*

- “Self-healing” iterative methods;
- Approximate linear operators;

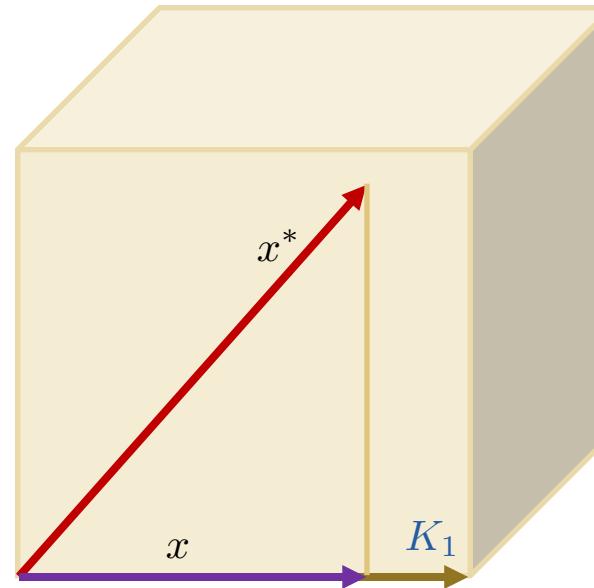


# Rethinking Algorithms I: Self-Healing Iterative Methods

- **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...).

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

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



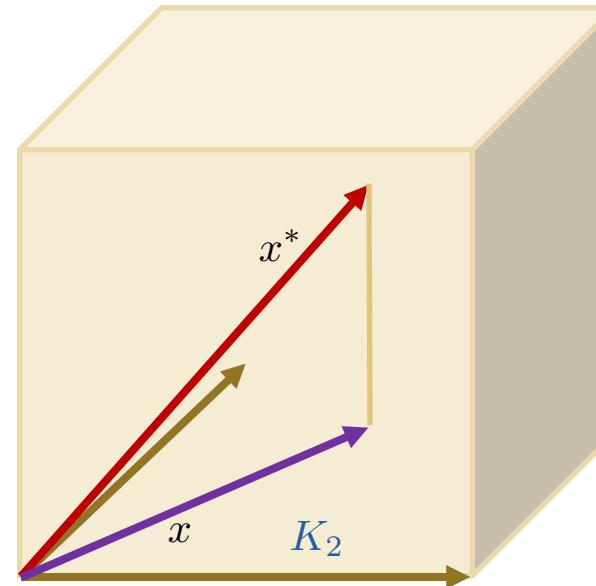
\*GMRES in the modular precision ecosystem

# Rethinking Algorithms I: 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...).

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

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



\*GMRES in the modular precision ecosystem

# Rethinking Algorithms I: Self-Healing Iterative Methods

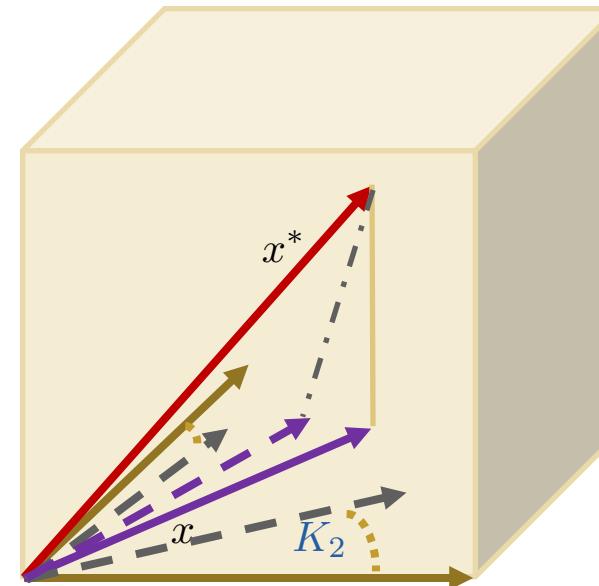
- **Krylov iterative solvers**
- **Krylov methods** aim at approximating the solution to a linear problem in a subspace.
- Over the iterations, a nested sequence of **Krylov subspaces** is generated, adding one basis vector in each iteration.
- **Orthonormalization** ensures a orthonormal basis is formed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

## 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;

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

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

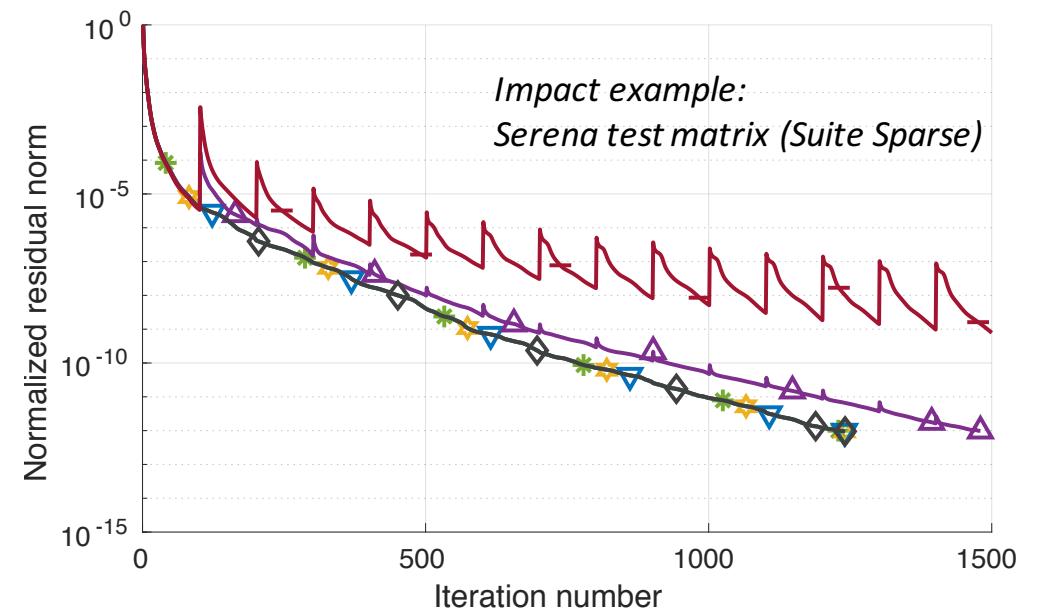
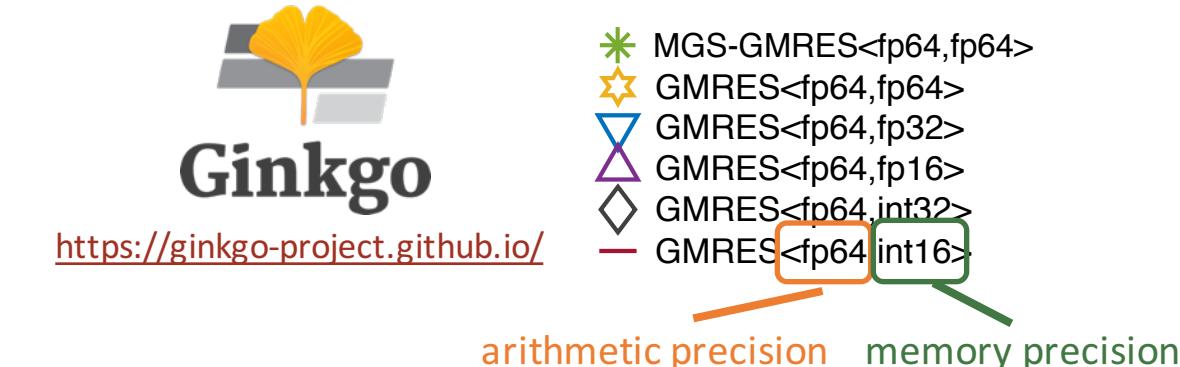


# Compressed Basis (CB-) GMRES

- **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...).

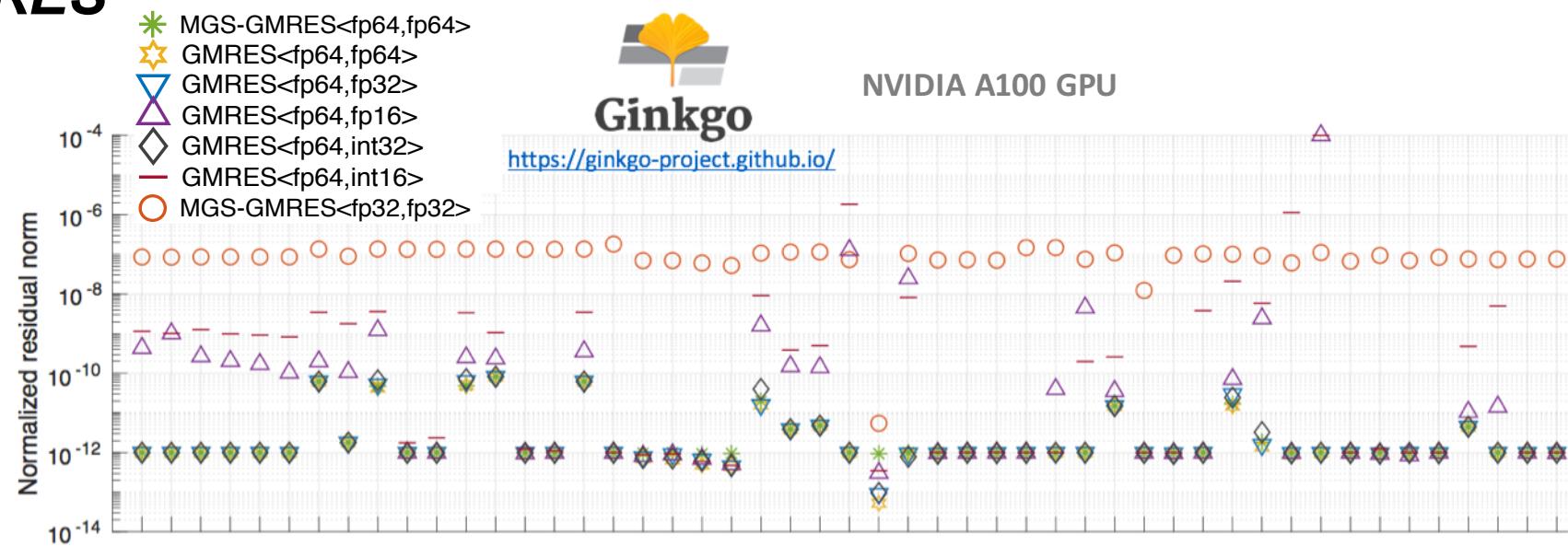
## Compressed Basis (CB-) GMRES

- 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;



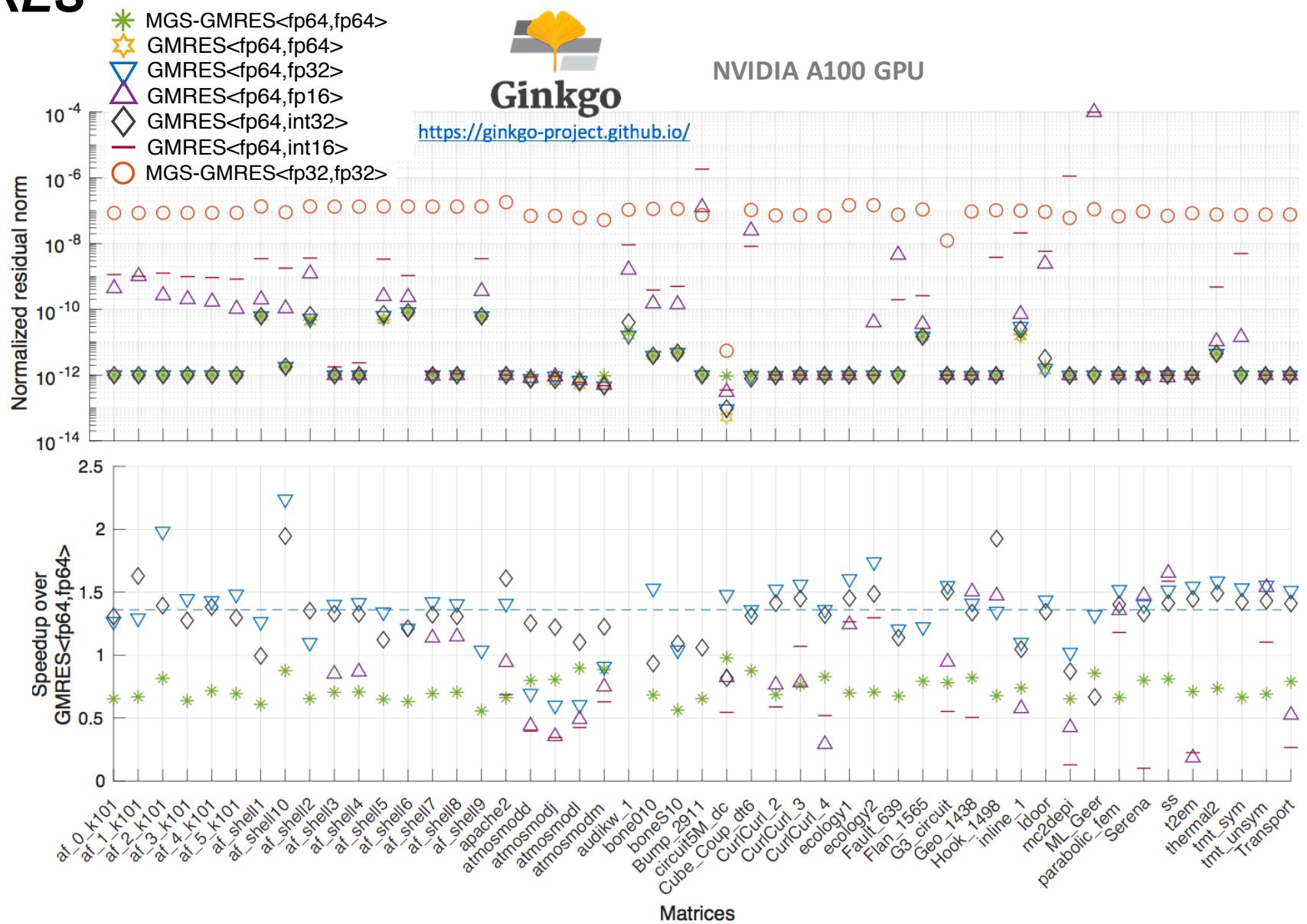
# Compressed Basis GMRES

- 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  $\approx 1.4x$  (for restart 100)
- 16-bit storage mostly inefficient



## Compressed Basis GMRES on High Performance GPUs

José I. Aliaga, Hartwig Anzt, Thomas Grützmacher, Enrique S. Quintana-Ortí, Andrés E. Tomás

<https://arxiv.org/abs/2009.12101>

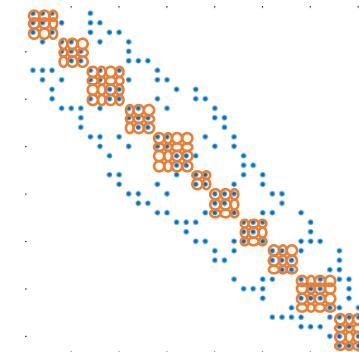
## *Rethinking Algorithms II: Approximate Linear Operators*

---

# Rethinking Algorithms II: Approximate Linear Operators

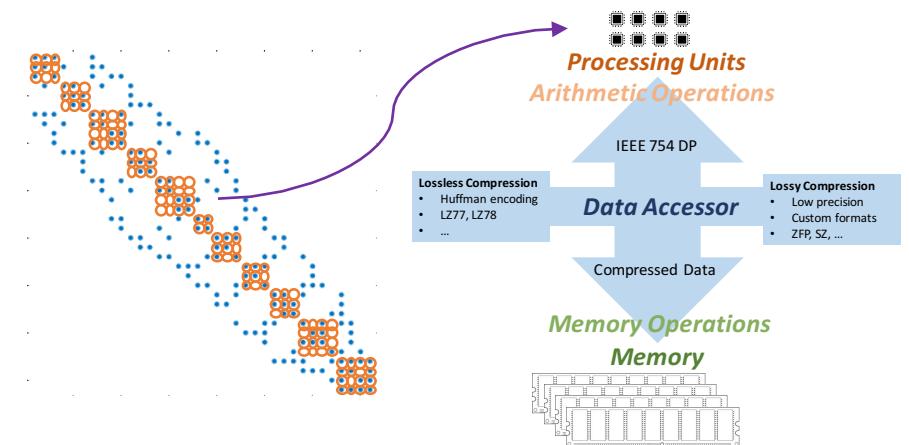
---

- **Preconditioning iterative solvers.**
  - Idea: Approximate inverse of system matrix to make the system “easier to solve”:  $P^{-1} \approx A^{-1}$  and solve  $Ax = b \Leftrightarrow P^{-1}Ax = P^{-1}b \Leftrightarrow \tilde{A}x = \tilde{b}$ .
- **Block-Jacobi preconditioner** is based on **block-diagonal scaling**:  $P = \text{diag}_B(A)$ 
  - Each block corresponds to one (small) linear system.
    - *Larger* blocks typically improve convergence.
    - *Larger* blocks make block-Jacobi more expensive.



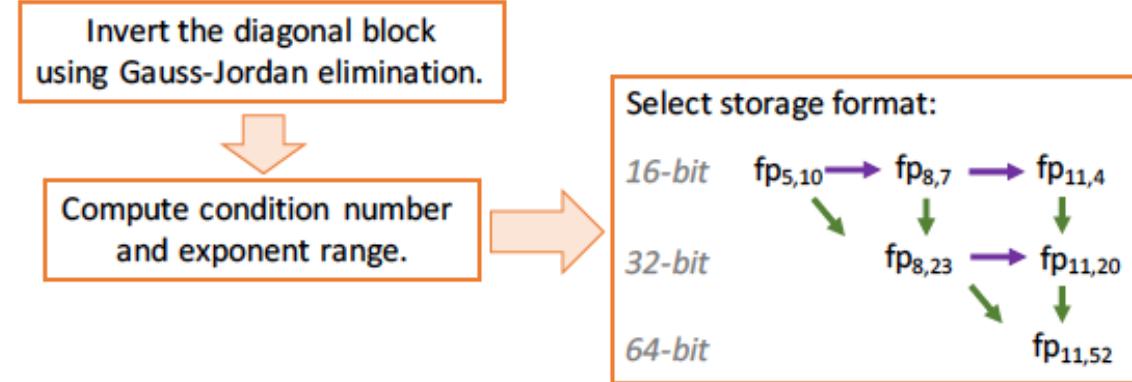
# Rethinking Algorithms II: Approximate Linear Operators

- Preconditioning iterative solvers.
  - Idea: Approximate inverse of system matrix to make the system “easier to solve”:  $P^{-1} \approx A^{-1}$  and solve  $Ax = b \Leftrightarrow P^{-1}Ax = P^{-1}b \Leftrightarrow \tilde{A}x = \tilde{b}$ .
- Block-Jacobi preconditioner is based on block-diagonal scaling:  $P = \text{diag}_B(A)$ 
  - 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!



# Mixed Precision Preconditioning

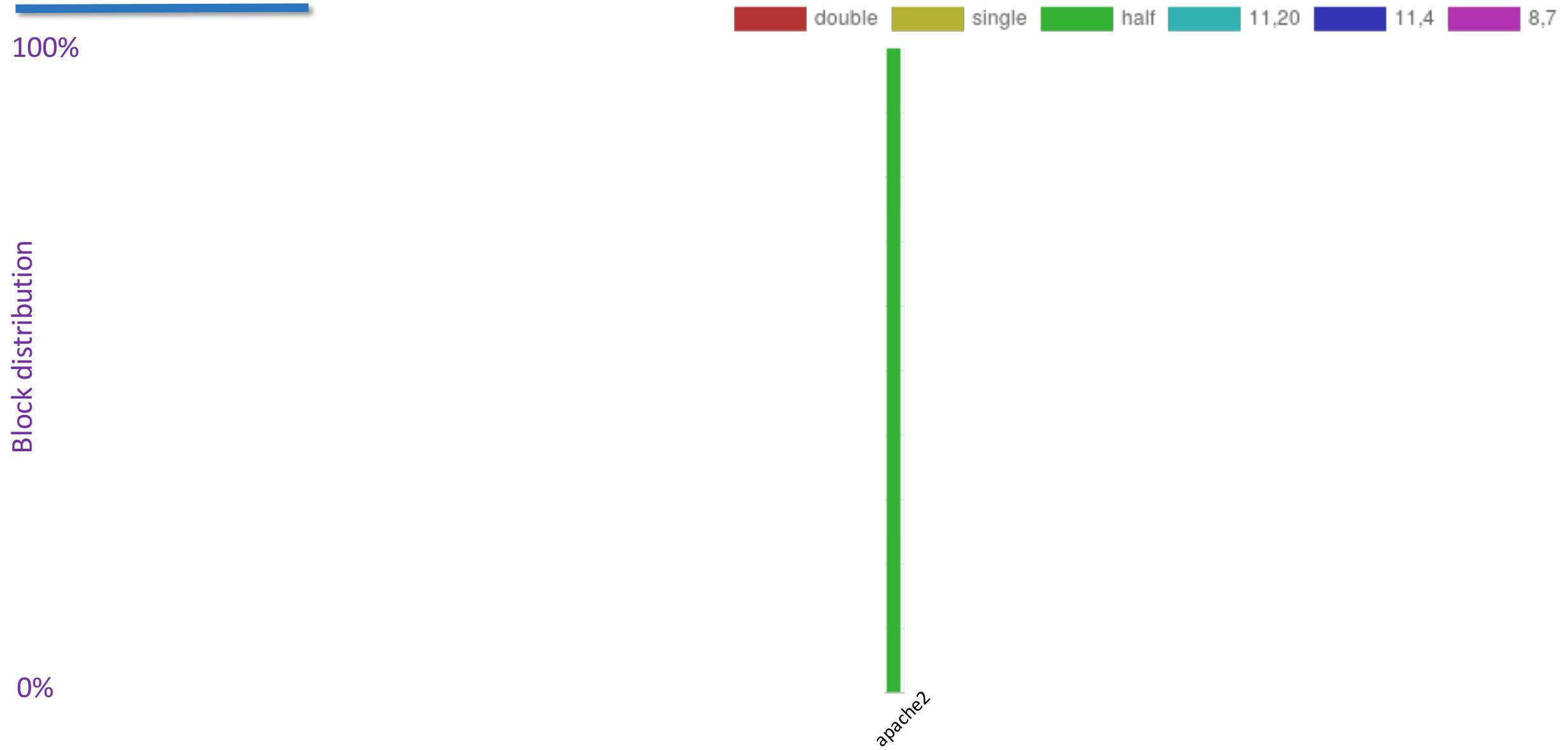
- Choose how much accuracy of the preconditioner should be preserved in the selection of the storage format.
- All computations use double precision, but store blocks in lower precision.



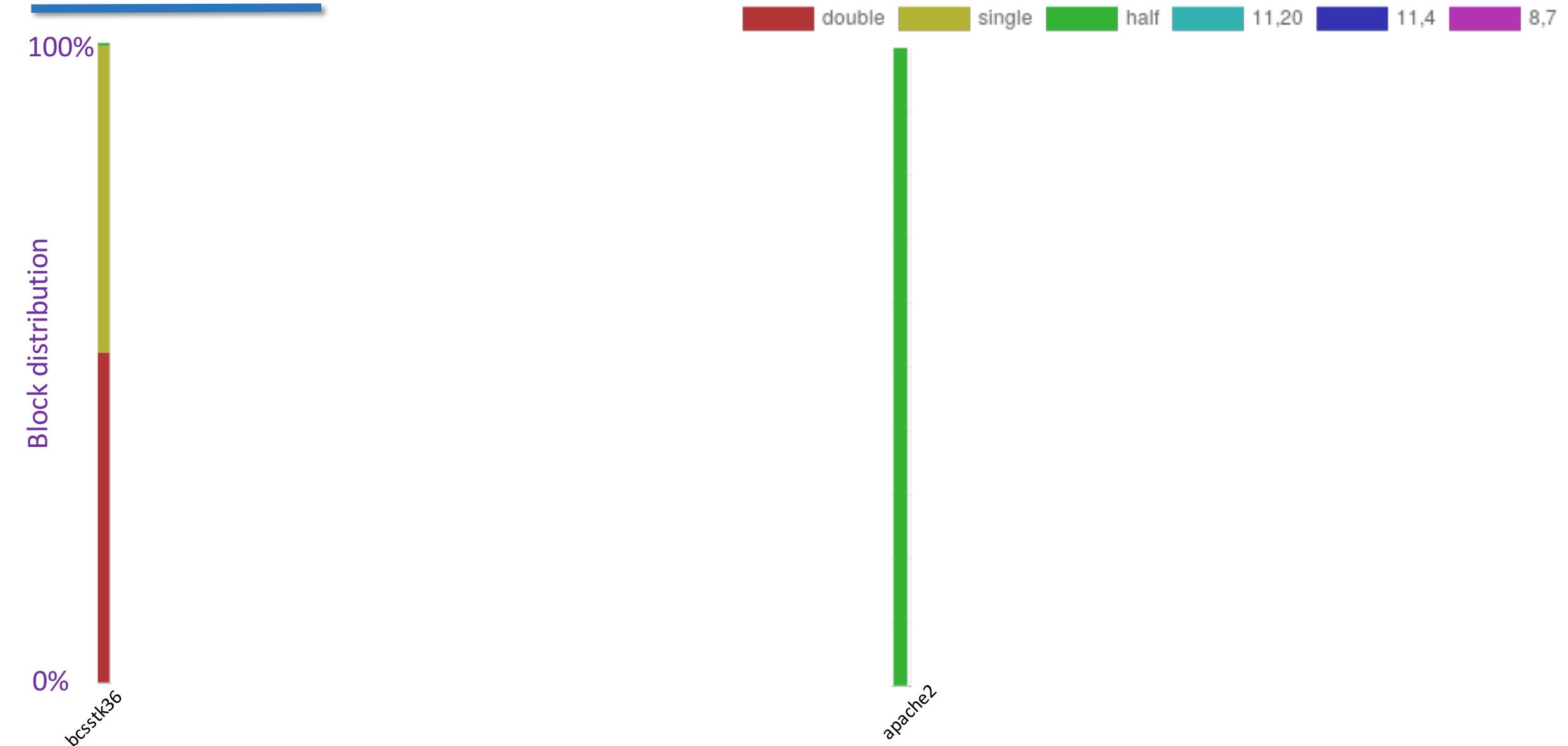
- + Regularity preserved;
- + Flexibility in the accuracy;
- + "Not a low precision preconditioner"
  - + Preconditioner is a constant operator;
  - + No flexible Krylov solver needed;

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

# *Mixed Precision Preconditioning*



# *Mixed Precision Preconditioning*



# Mixed Precision Preconditioning



# Mixed Precision Preconditioning



Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^7$  ( 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 Accuracy improvement ~109  
CG iteration count: 4797  
CG execution time [ms]: 2971.18
```

## 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 No improvement  
CG iteration count: 8887  
CG execution time [ms]: 2972.46
```

Experiments based on the Ginkgo library <https://ginkgo-project.github.io/>

[ginkgo/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp](https://github.com/GinkgoProject/ginkgo/blob/main/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp)

# Mixed Precision Preconditioning



Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^7$  ( 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

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

Experiments based on the Ginkgo library <https://ginkgo-project.github.io/>

[ginkgo/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp](#)

# Mixed Precision Preconditioning



Linear System  $Ax=b$  with  $\text{cond}(A) \approx 10^7$  ( 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?

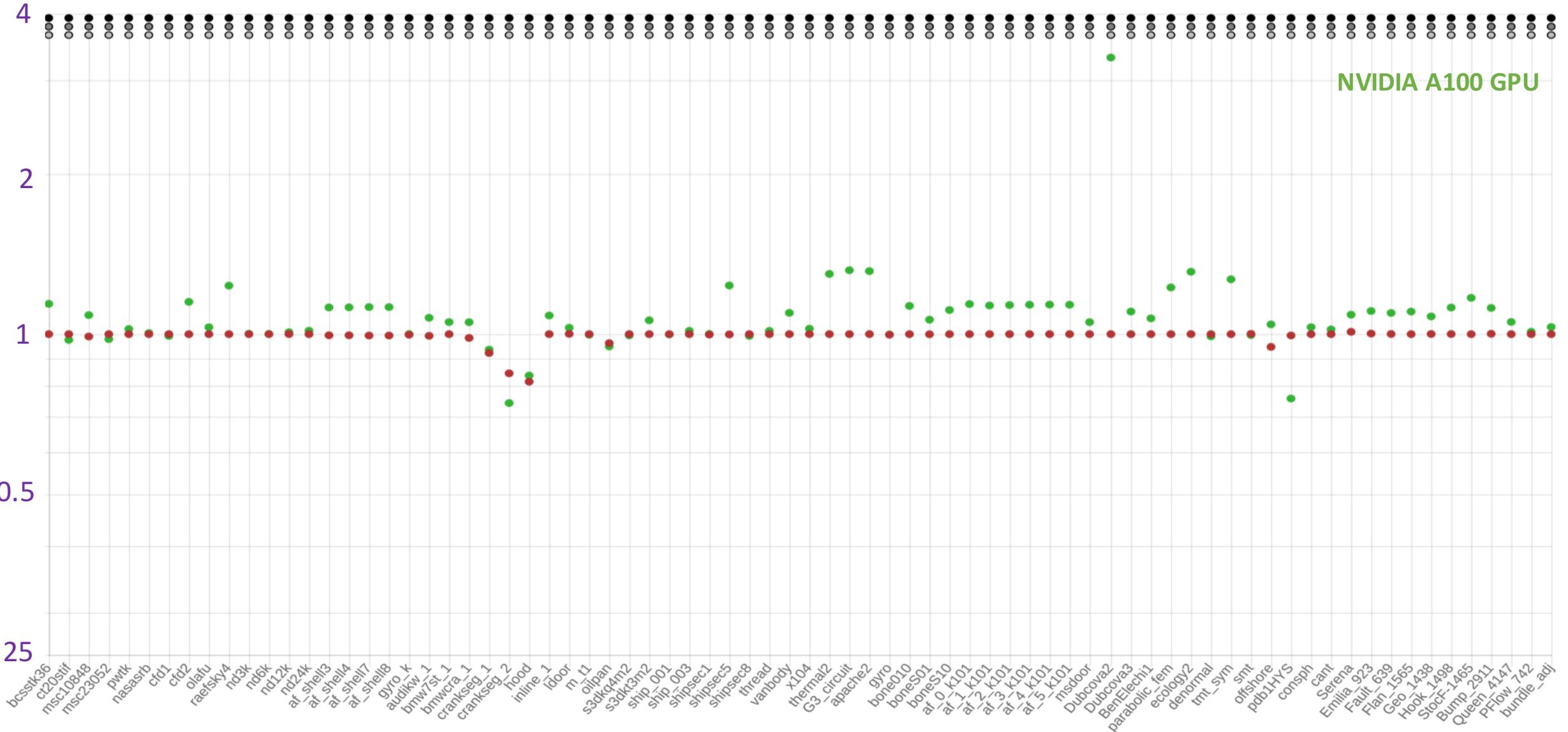
4

2

Speedup

0.5

0.25



Problem

 Iterations (adaptive) 
  Time (adaptive) 
  CG converged? 
  CG + Jacobi converged? 
  CG + adaptive Jacobi converged?

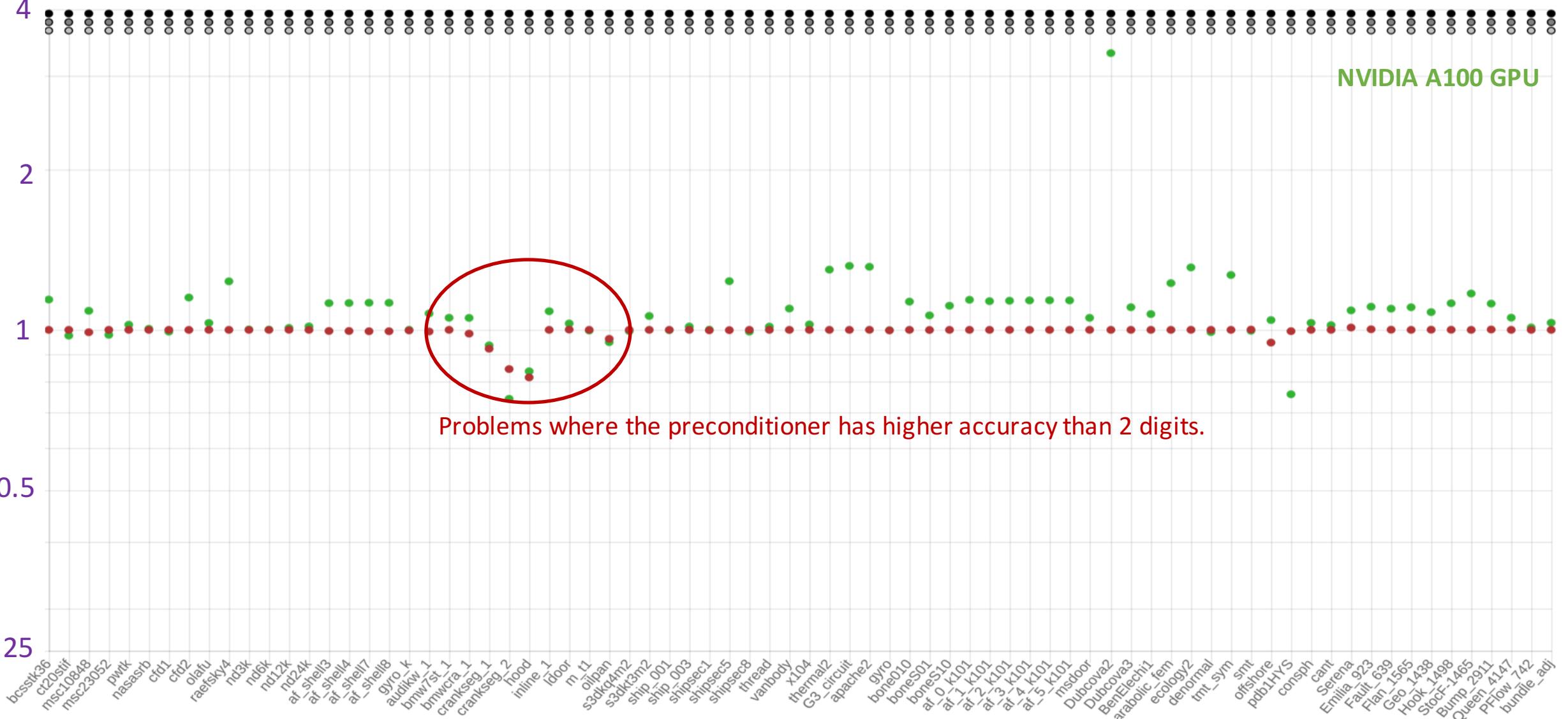
4

2

Speedup

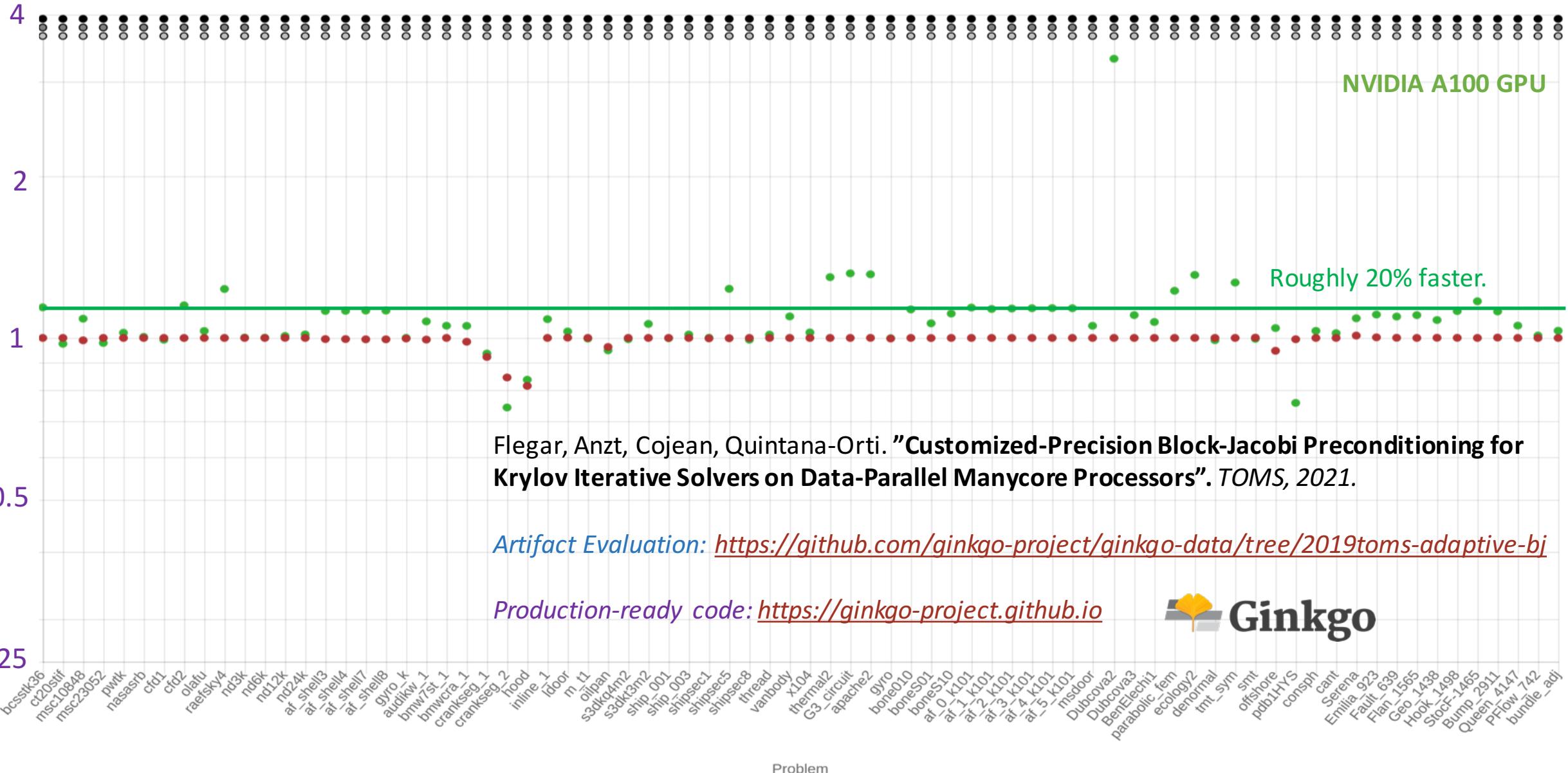
0.5

0.25



Problem

Iterations (adaptive) Time (adaptive) CG converged? CG + Jacobi converged? CG + adaptive Jacobi converged?

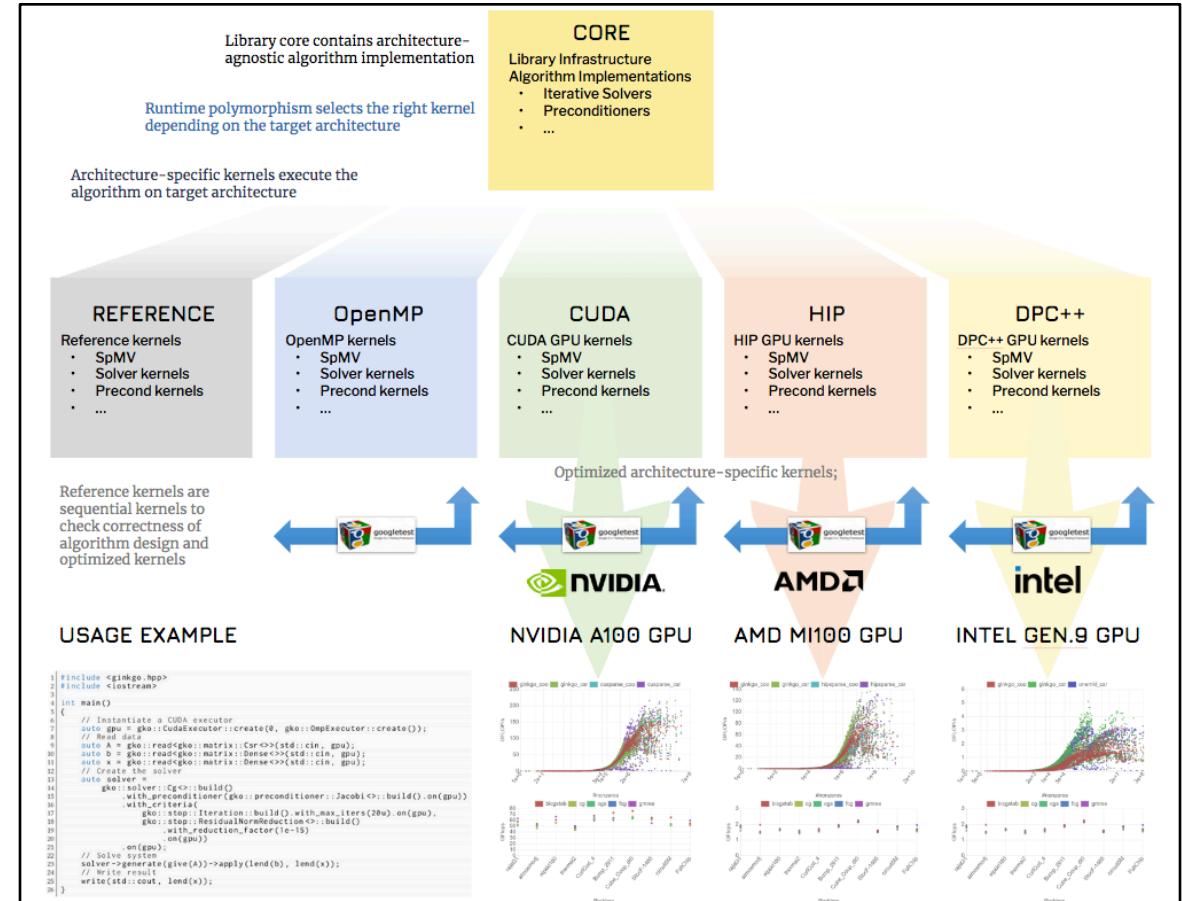




# Ginkgo



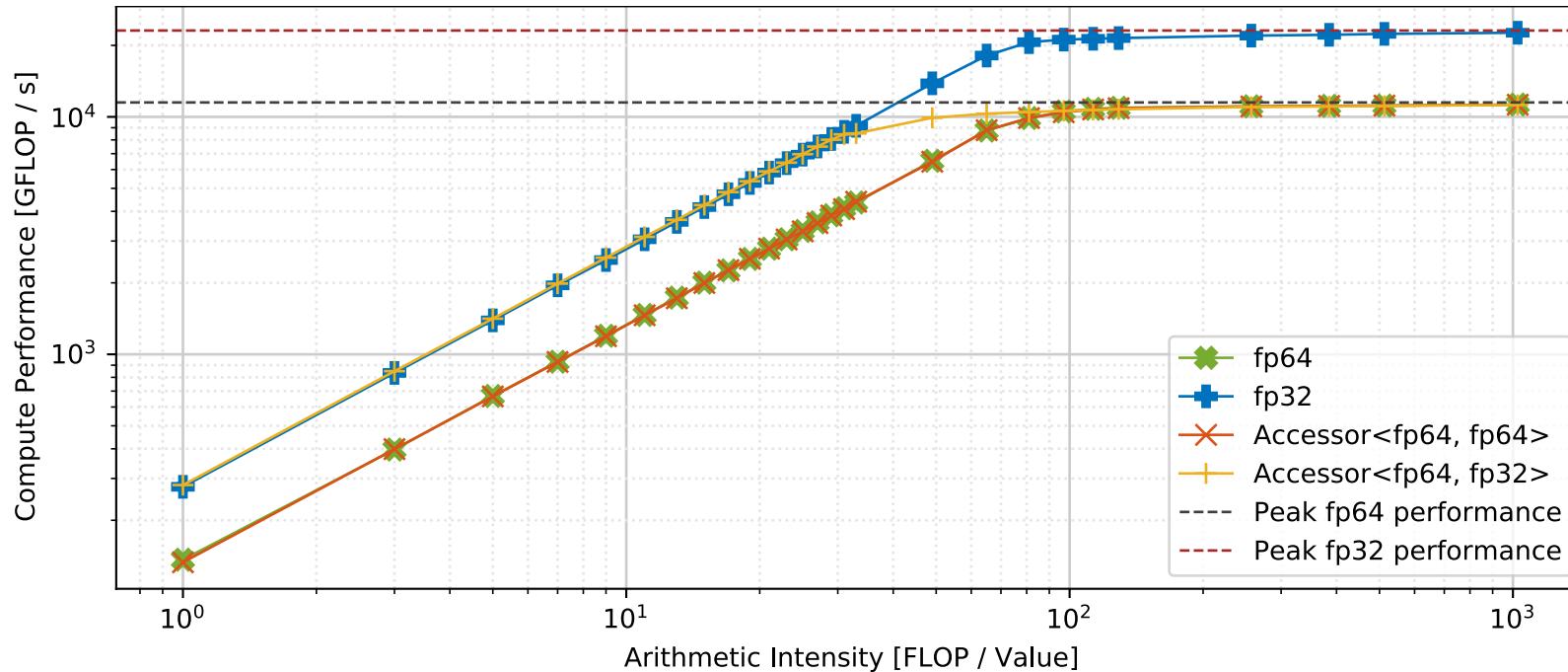
- **High performance sparse linear algebra**
    - Linear algebra building blocks: SpMV, SpMM, SpGEAM,...;
    - Linear solvers: BiCG, BiCGSTAB, CG, CGS, FCG, GMRES, IDR;
    - Advanced preconditioning techniques: ParILU, ParILUT, SAI;
    - Batched iterative solvers;
  - **Exascale early systems GPU-readiness**
    - Available: Nvidia GPU (CUDA), AMD GPU (HIP), Intel GPU (DPC++), CPU Multithreading (OpenMP);
    - C++, CMake build;
  - **Open source, community-driven**
    - Freely available (BSD License), GitHub, and Spack;
    - Part of the **xSDK and E4S software stack**;
    - Can be used from **deal.II** and **MFEM**;
  - **Modular precision ecosystem**
    - Decoupling of arithmetic precision and memory precision;
    - Compressed Basis (CB) Krylov methods;
    - Mixed precision algorithms: adaptive precision Jacobi, FSPAI



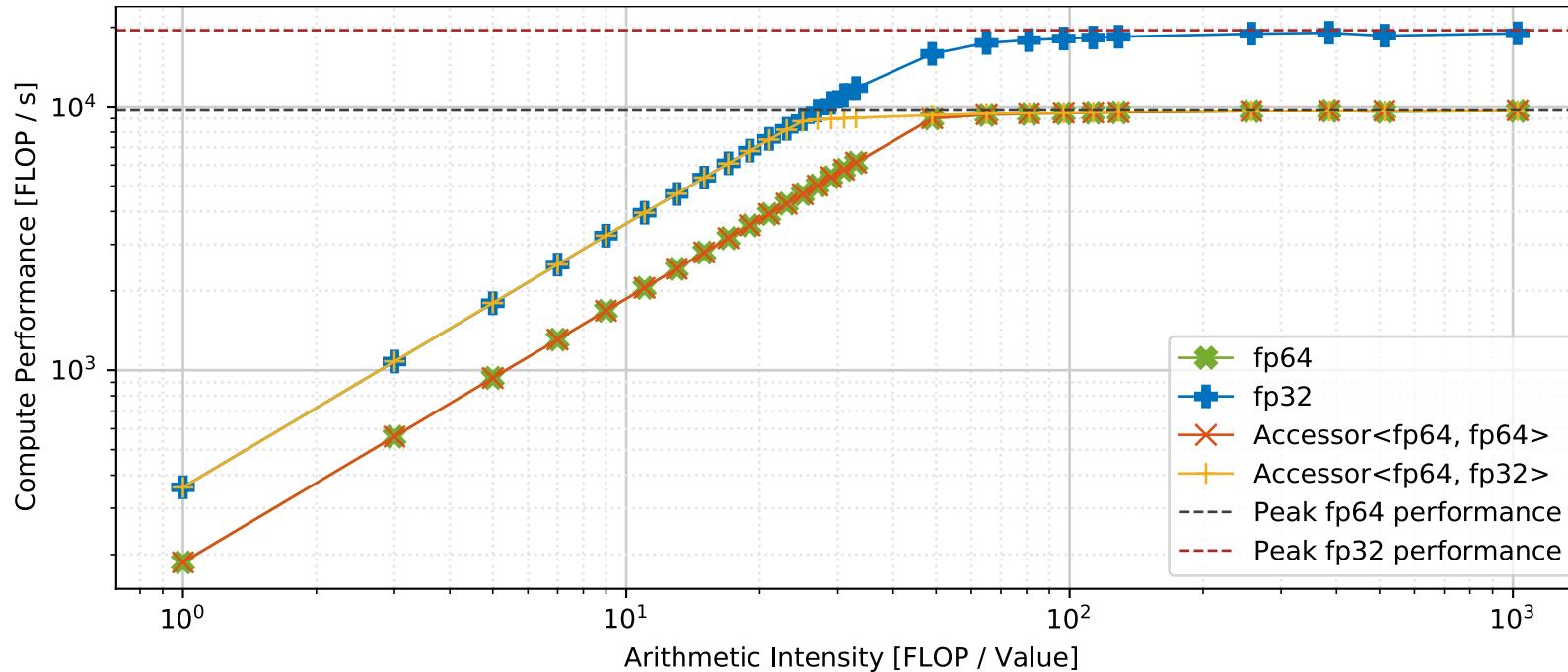
<https://ginkgo-project.github.io/>

*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 and the Helmholtz Impuls und Vernetzungsfond VH-NG-1241.*

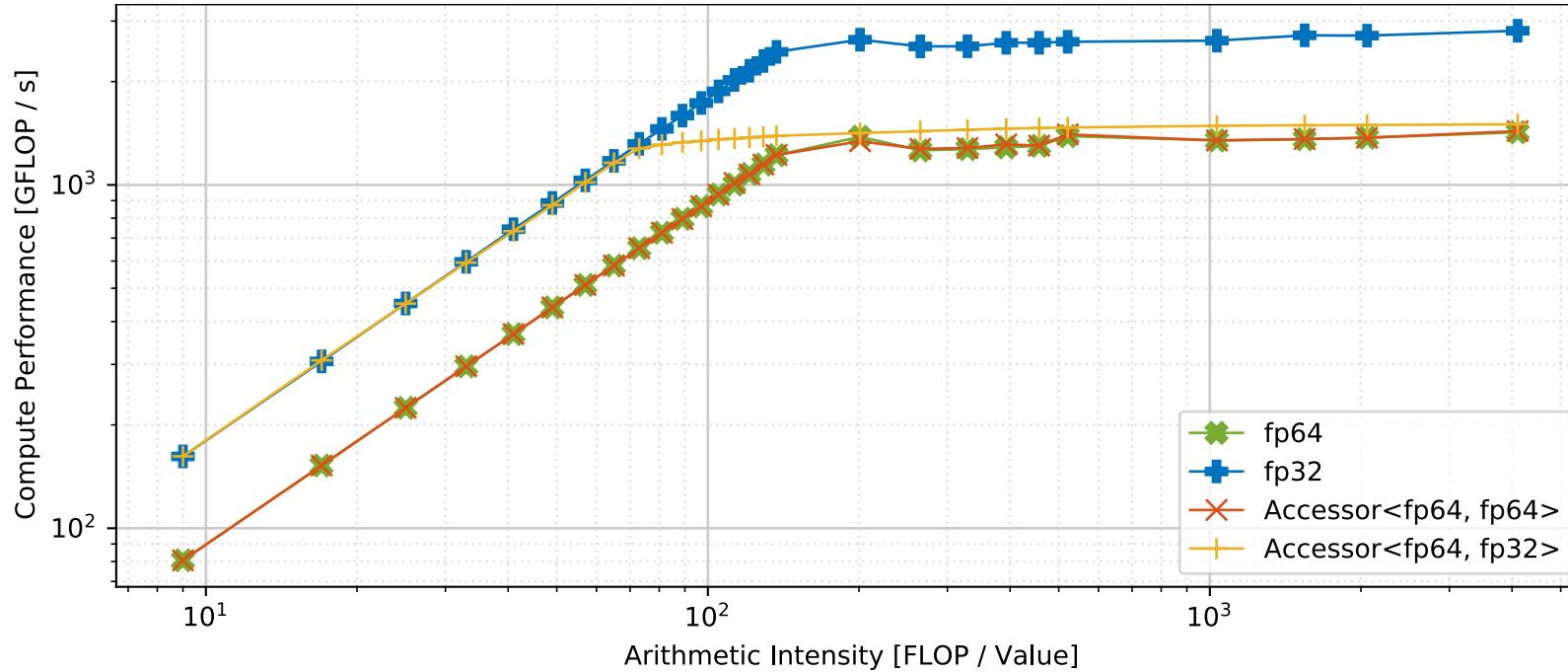
# Accessor for AMD MI100 GPU



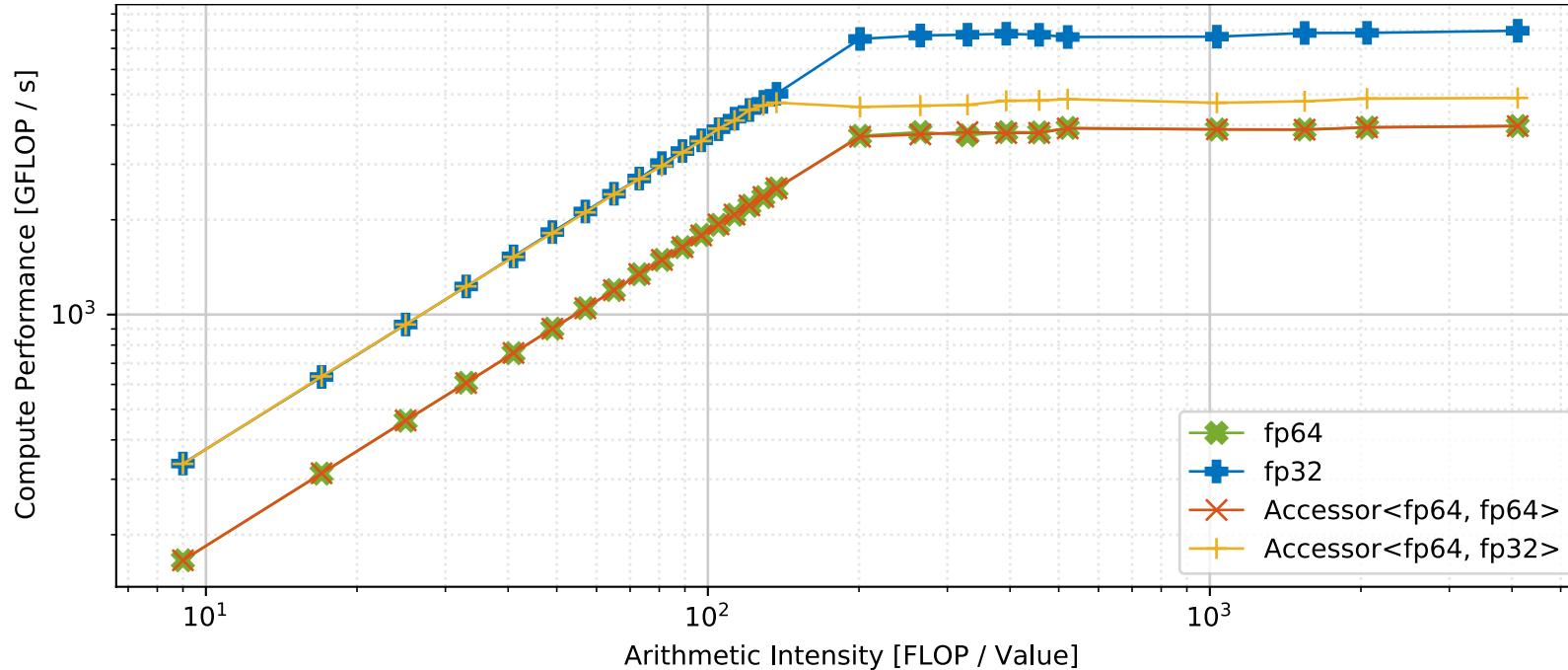
# Accessor for NVIDIA A100 GPU



# Accessor for Intel Skylake CPU



# Accessor for AMD EPYC CPU



# Memory accessor for memory-bound routines

---

## Design

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

## Characteristics

- Performance of low precision routine;
- Higher accuracy of low precision routine;

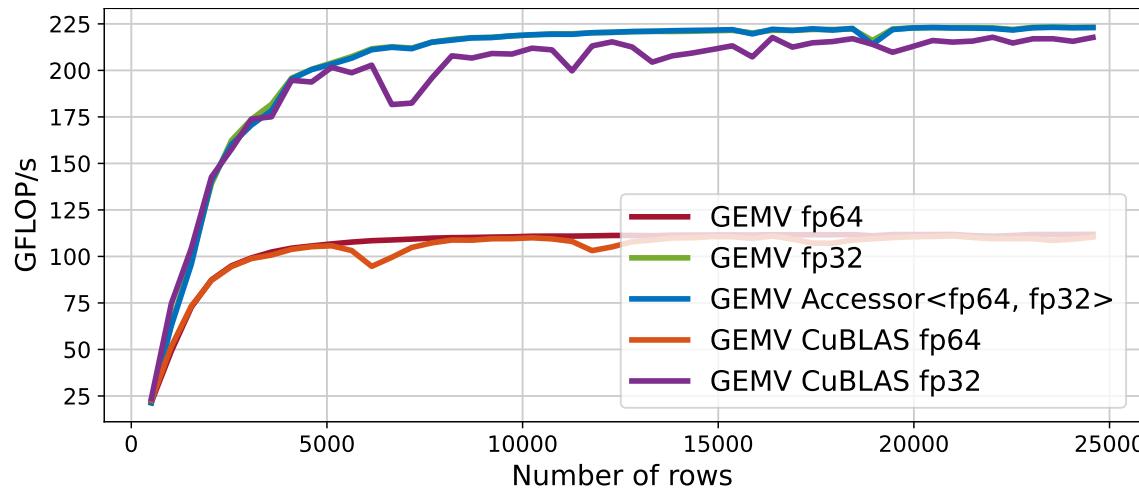
## Usage

1. Accessor-BLAS can replace low precision BLAS to increase accuracy;
2. Accessor-BLAS 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

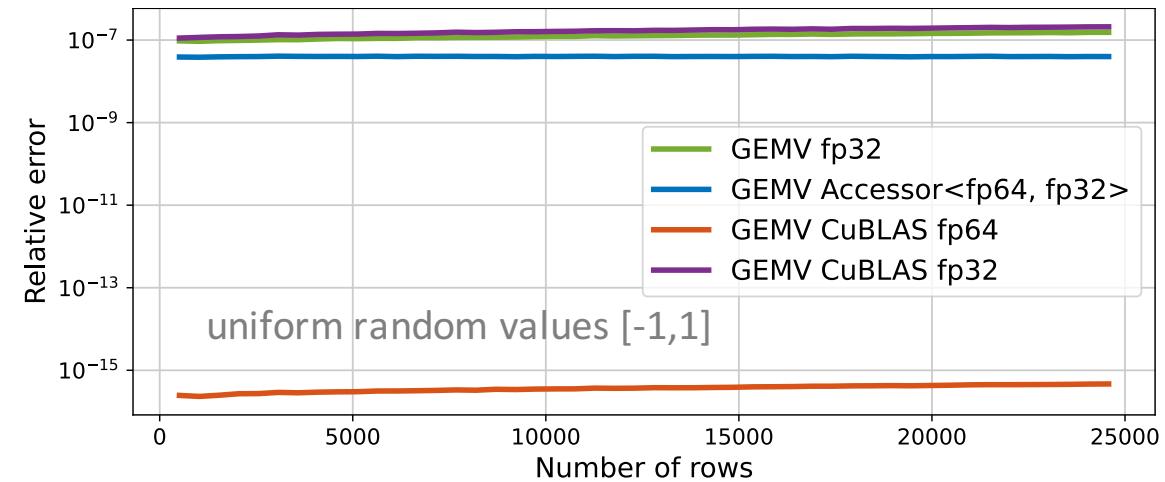
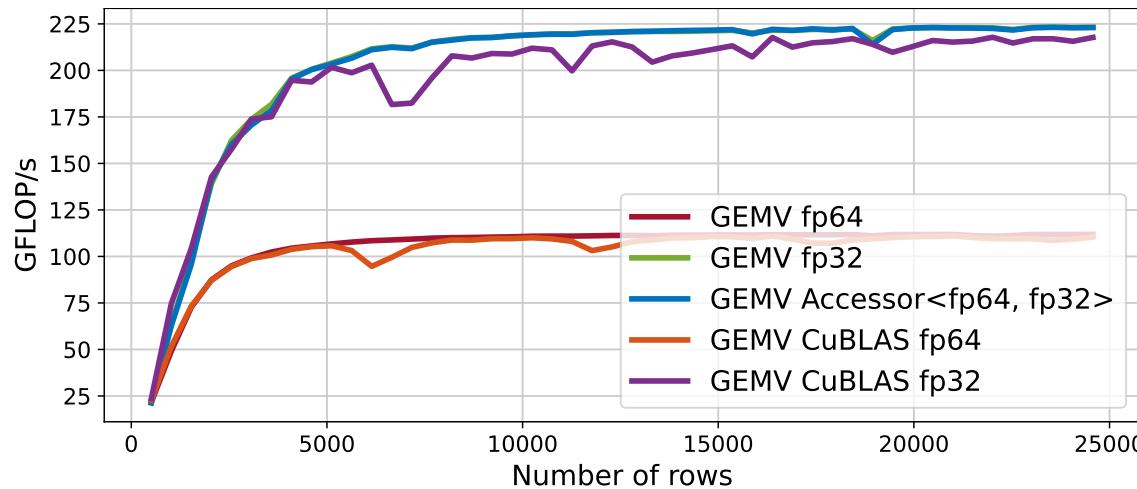
NVIDIA V100 GPU (Summit)



# Accessor-BLAS: Replacing LP BLAS to improve accuracy

## GEMV

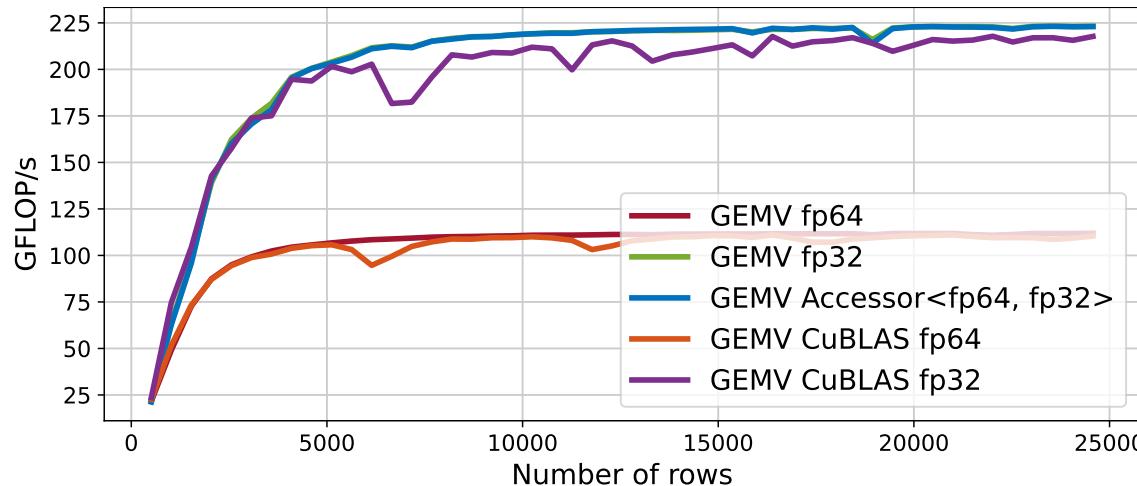
NVIDIA V100 GPU (Summit)



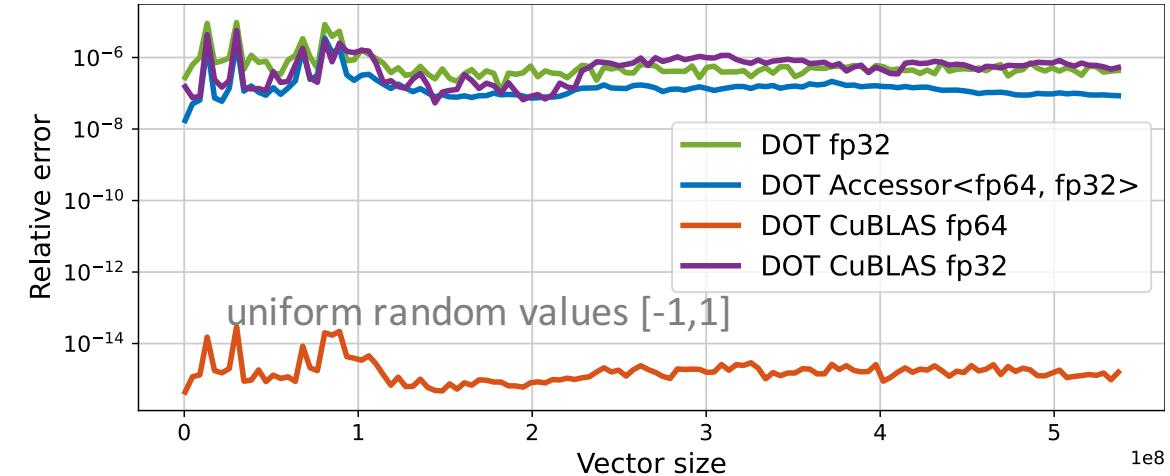
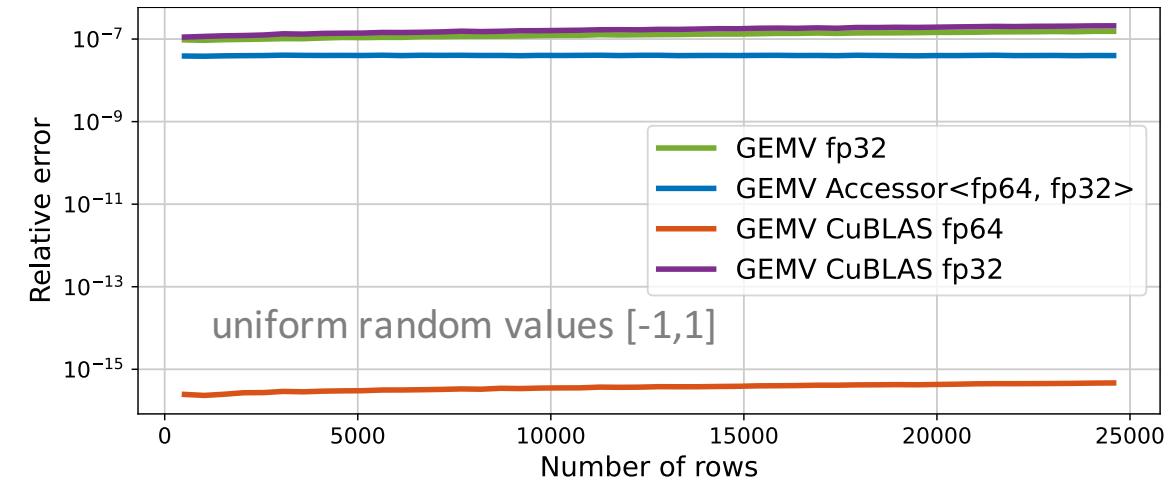
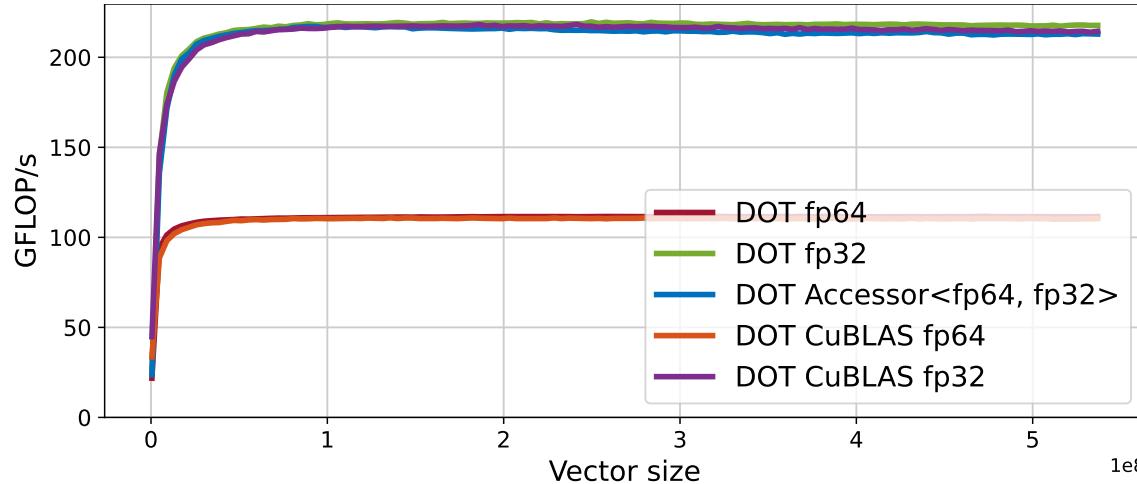
# Accessor-BLAS: Replacing LP BLAS to improve accuracy

## GEMV

NVIDIA V100 GPU (Summit)



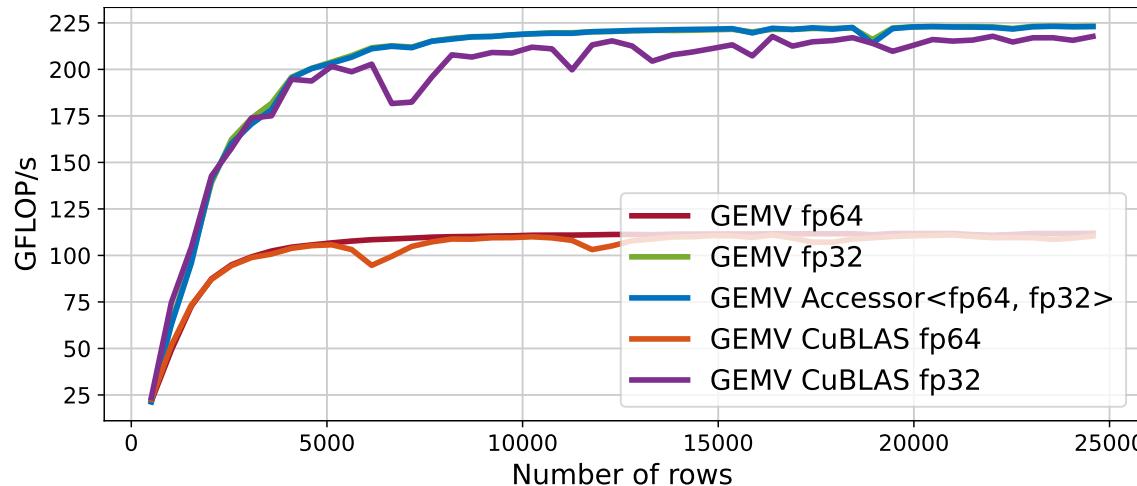
## DOT



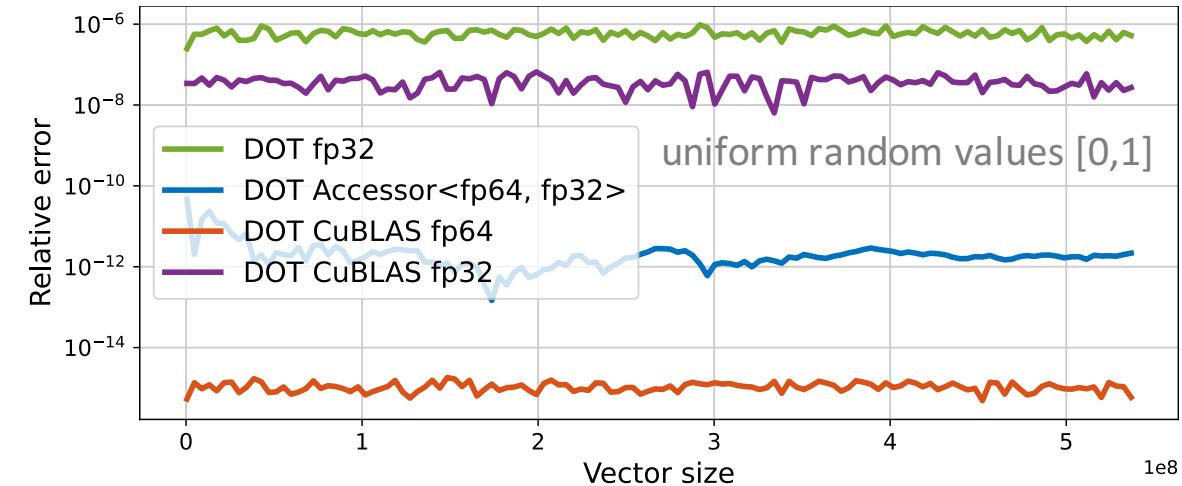
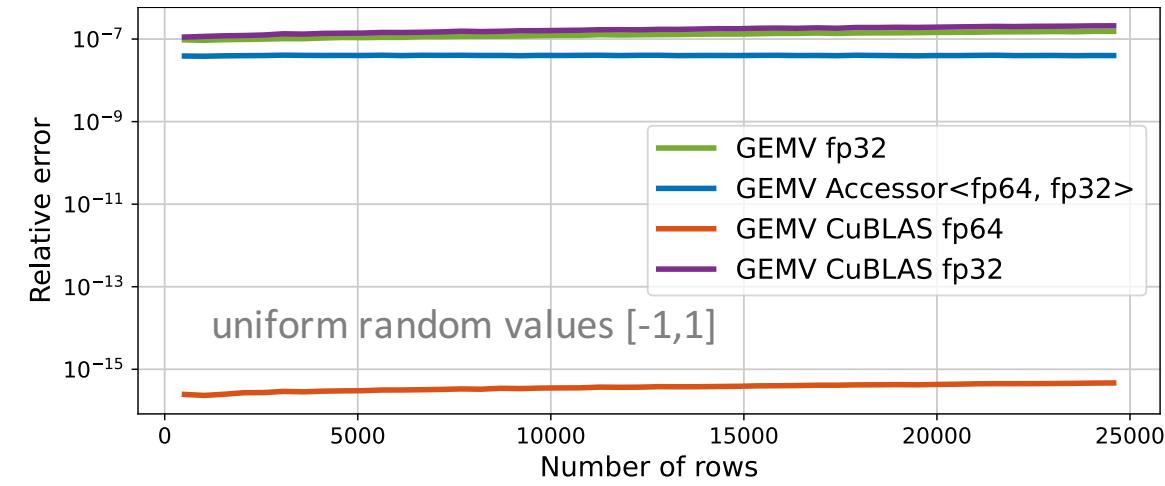
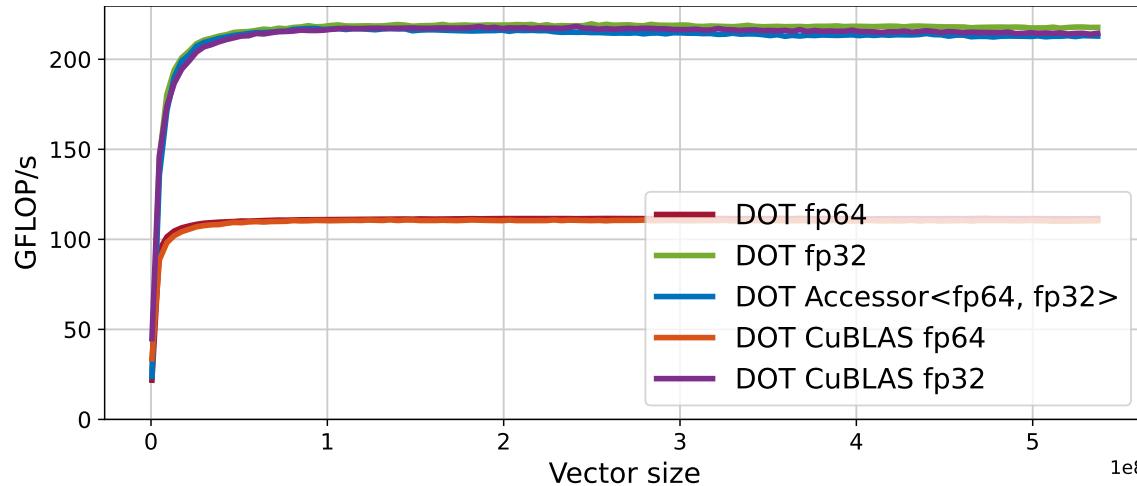
# Accessor-BLAS: Replacing LP BLAS to improve accuracy

## GEMV

NVIDIA V100 GPU (Summit)



## DOT



# Mixed Precision Sparse Approximate Inverse

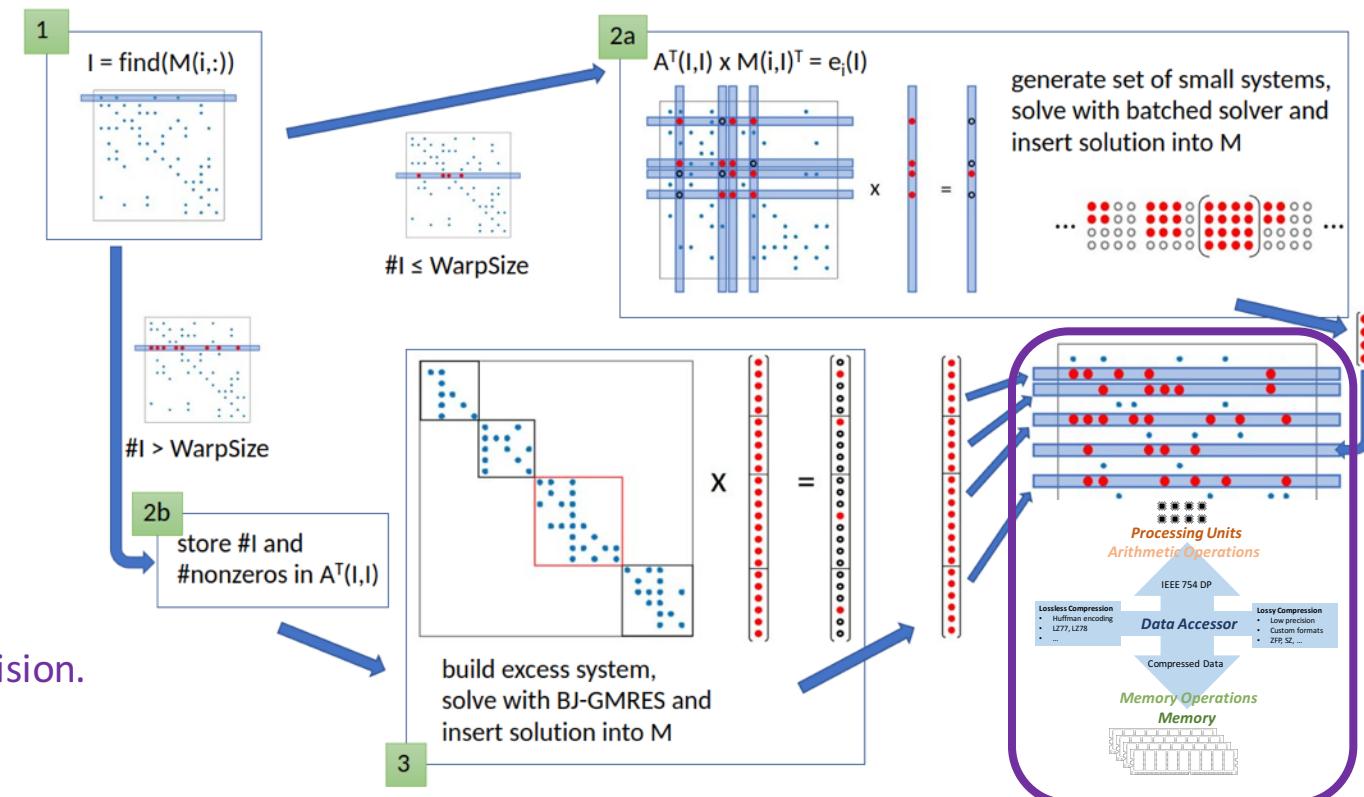
- Preconditioning iterative solvers.

- Idea: Approximate inverse of system matrix to make the system “easier to solve”:  $P^{-1} \approx A^{-1}$   
and solve  $Ax = b \Leftrightarrow P^{-1}Ax = P^{-1}b \Leftrightarrow \tilde{A}x = \tilde{b}$ .

- Sparse Approximate Inverse Preconditioner

- $M \approx A^{-1}$  and sparse
- Incomplete Sparse Approximate Inverse (ISAI)
  - uses sparsity pattern of  $A$ ;
- Factorized Sparse Approximate Inverse (FSPA)
  - stores inverse approximation in factorized form;

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



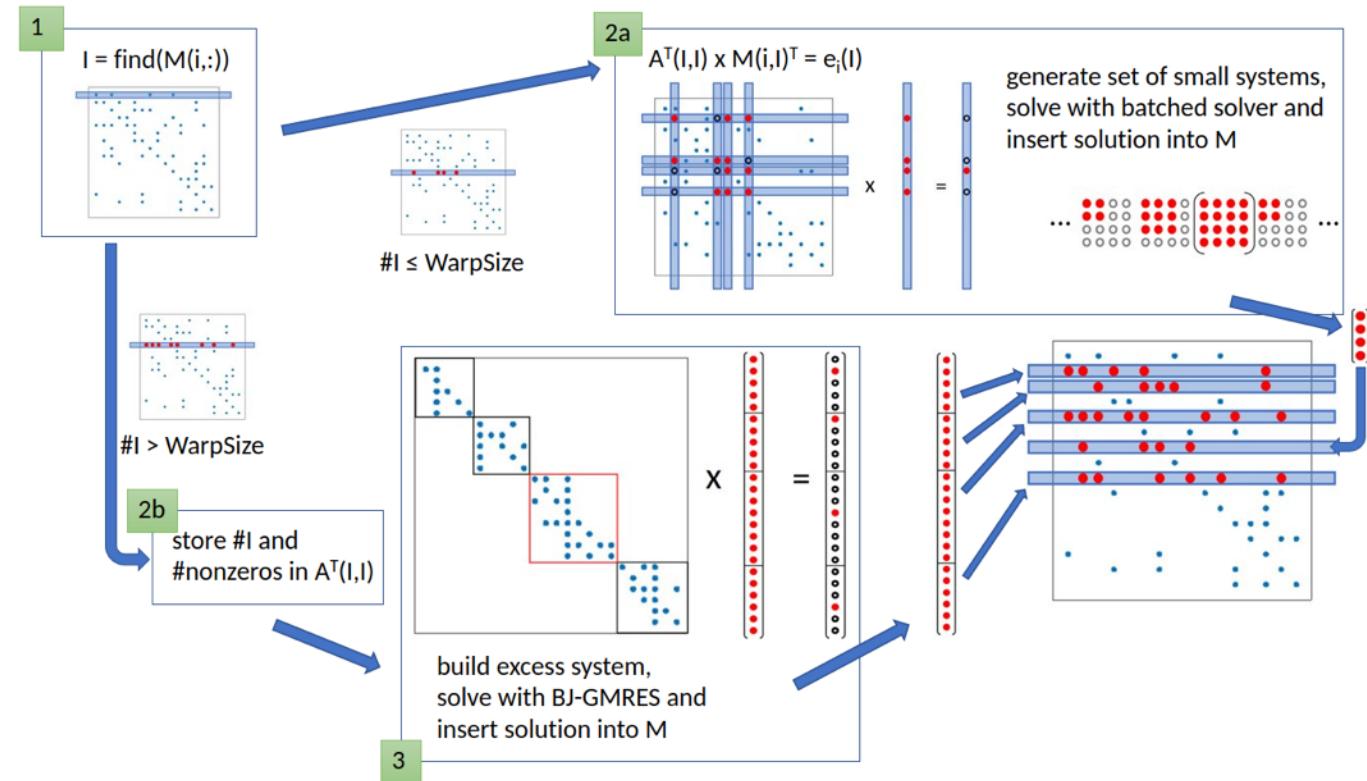
# Mixed Precision Sparse Approximate Inverse

- Preconditioning iterative solvers.

- Idea: Approximate inverse of system matrix to make the system “easier to solve”:  $P^{-1} \approx A^{-1}$   
and solve  $Ax = b \Leftrightarrow P^{-1}Ax = P^{-1}b \Leftrightarrow \tilde{A}x = \tilde{b}$ .

- 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;



# Mixed Precision Sparse Approximate Inverse

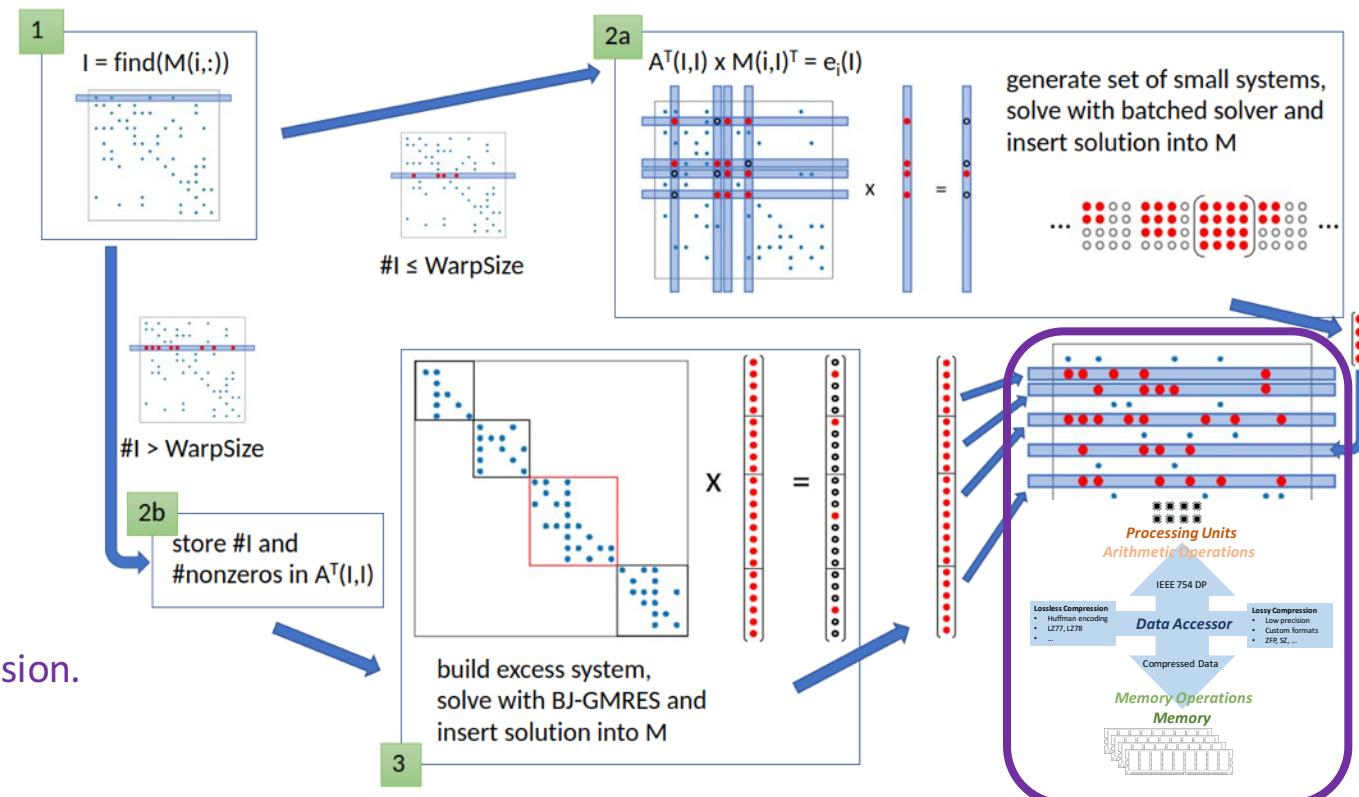
- Preconditioning iterative solvers.

- Idea: Approximate inverse of system matrix to make the system “easier to solve”:  $P^{-1} \approx A^{-1}$  and solve  $Ax = b \Leftrightarrow P^{-1}Ax = P^{-1}b \Leftrightarrow \tilde{A}x = \tilde{b}$ .

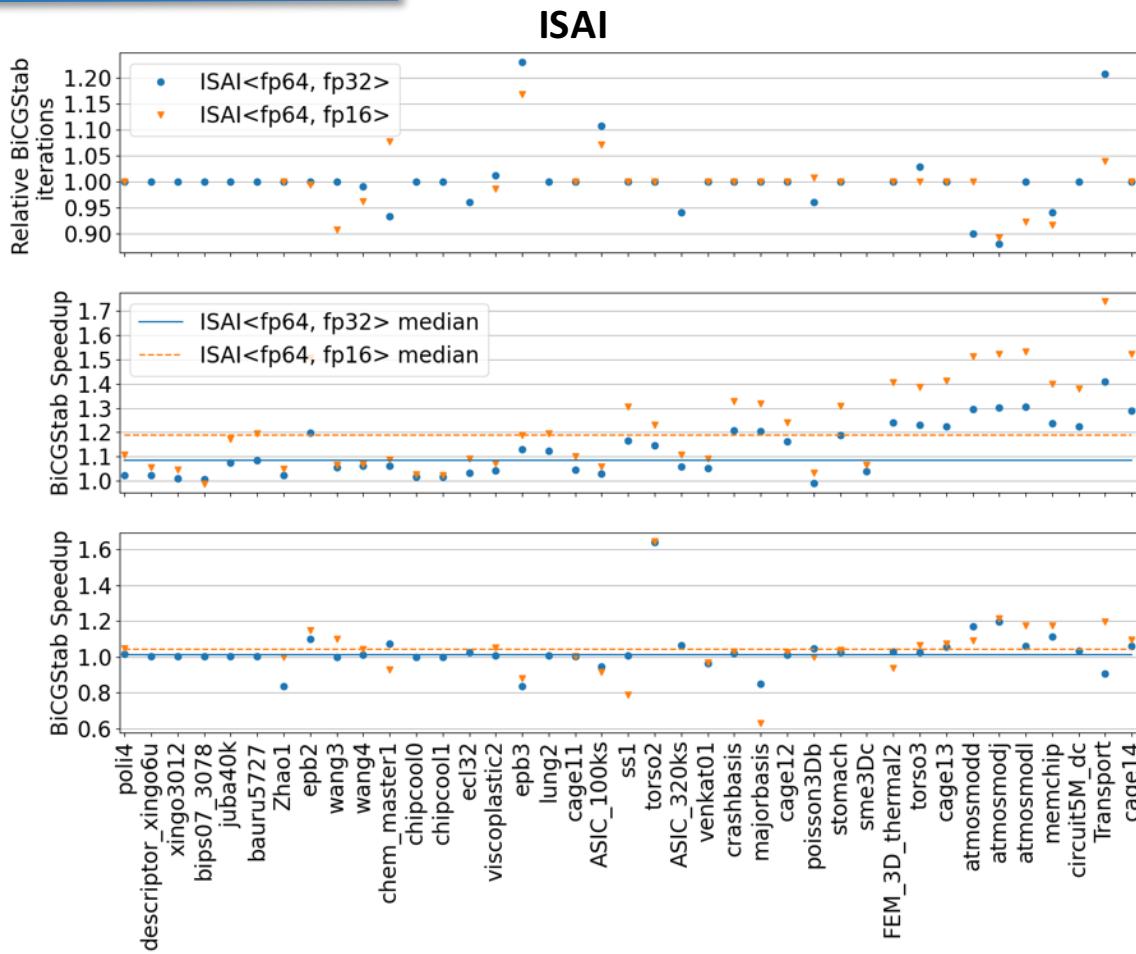
- Sparse Approximate Inverse Preconditioner

- $M \approx A^{-1}$  and sparse
- Incomplete Sparse Approximate Inverse (ISAI)
  - uses sparsity pattern of  $A$ ;
- Factorized Sparse Approximate Inverse (FSPA)
  - stores inverse approximation in factorized form;

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

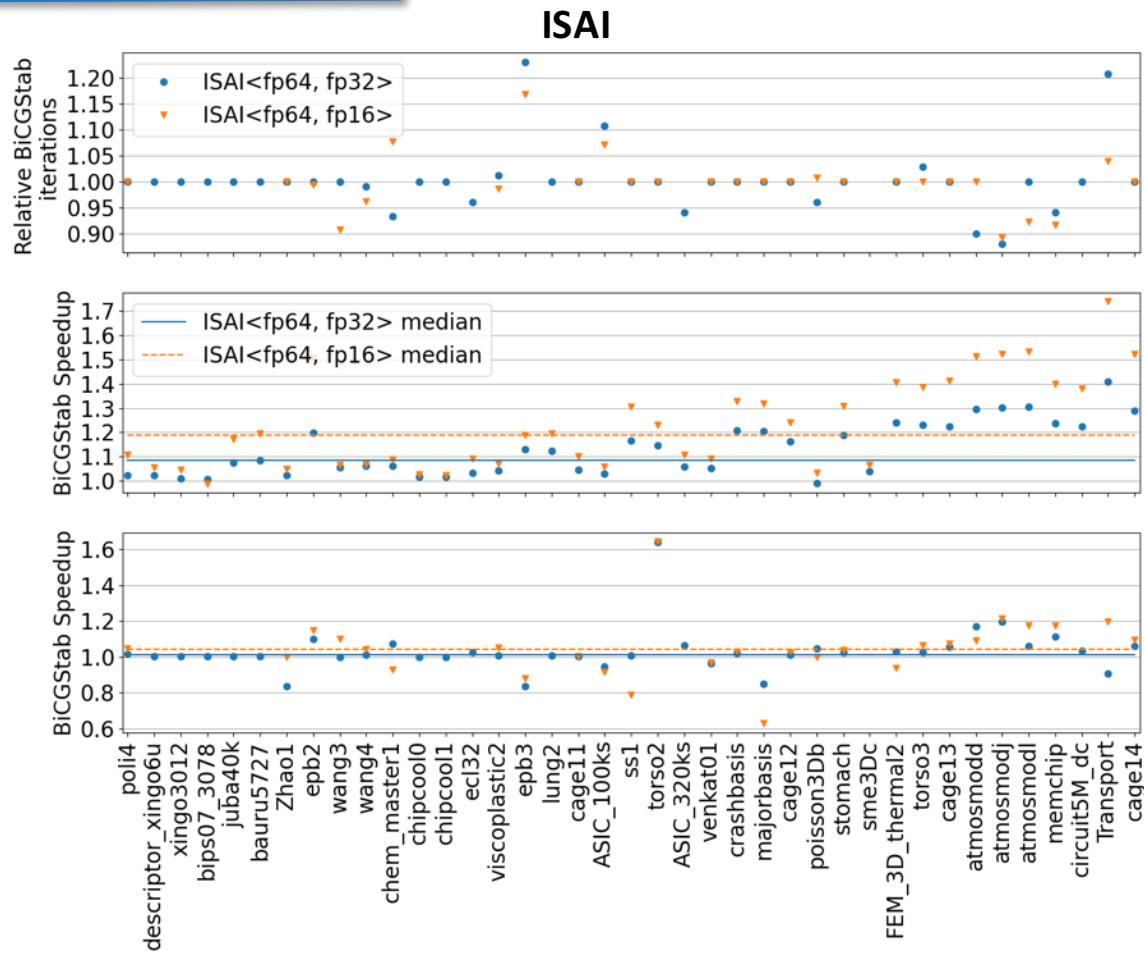


# Mixed Precision Sparse Approximate Inverse

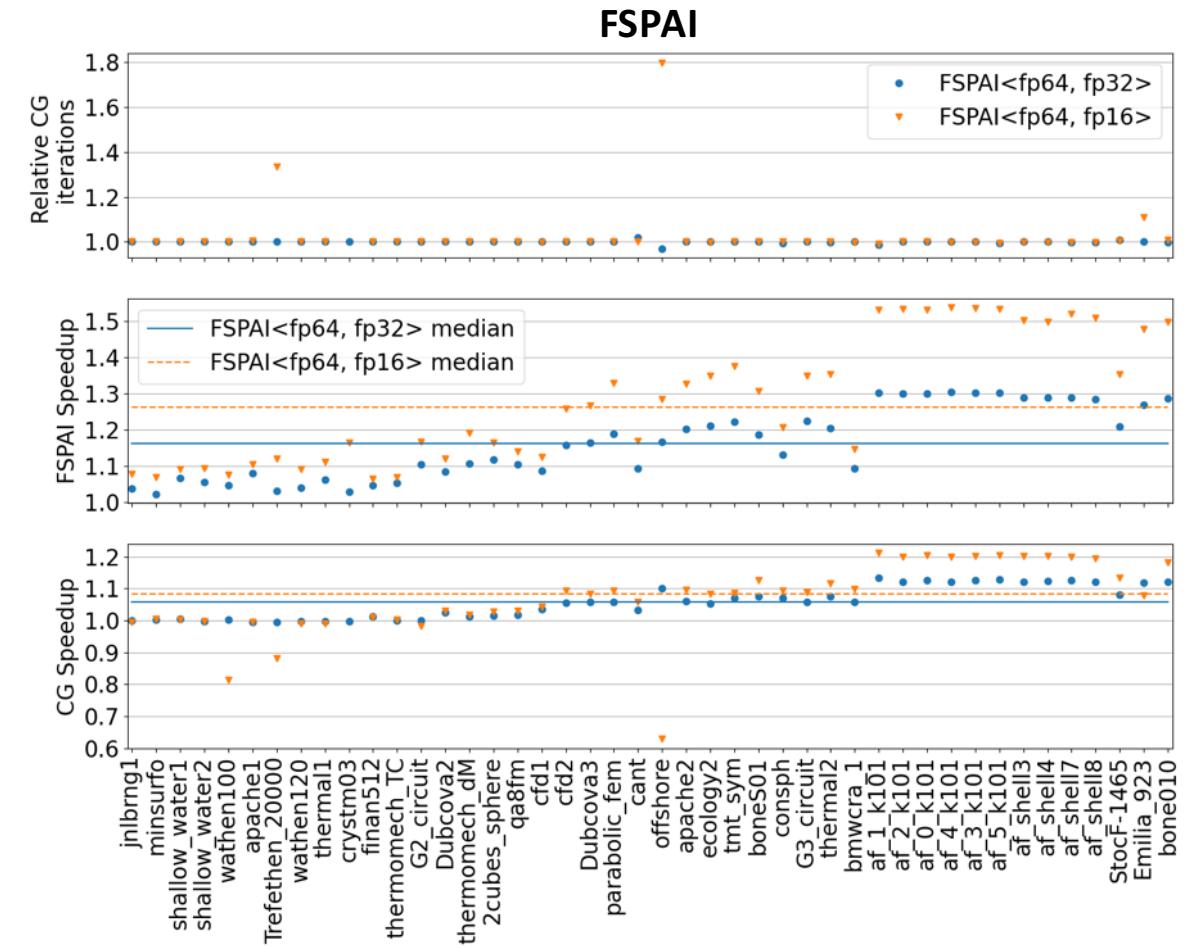


- ~10% speedup for ISAI<fp64,fp32>
- ~20% speedup for ISAI<fp64,fp16> - unstable!

# Mixed Precision Sparse Approximate Inverse



- ~10% speedup for ISAI<fp64,fp32>
- ~20% speedup for ISAI<fp64,fp16> - unstable!



- ~15% speedup for FSPAI<fp64,fp32>
- ~25% speedup for FSPAI<fp64,fp16>