Mixed Precision Strategies for Memory-Bound Linear Algebra

LANS Seminar
September 8th 2021

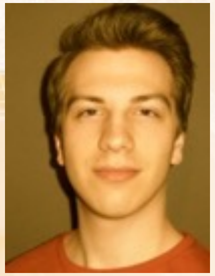
Hartwig Anzt
Steinbuch Centre for Computing (SCC)


Hartwig Anzt


Isha Aggarwal


Terry Cojean


Fritz Göbel


T. Grützmacher


Aditya Kashi


Marcel Koch


Pratik Nayak


Gregor Olinek


Tobias Ribizel

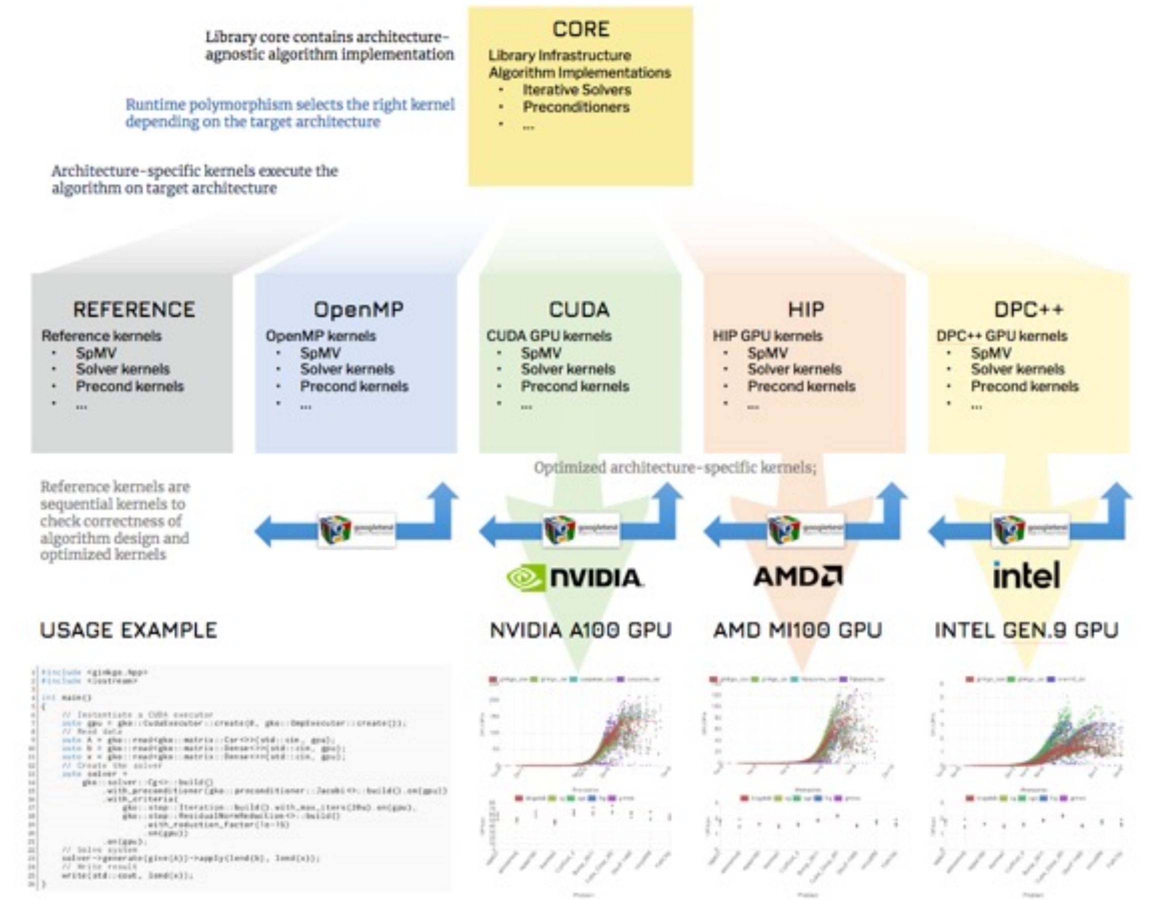

Mike Tsai

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 VernetzungsfondVH-NG-1241.

Ginkgo

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.

Ginkgo

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

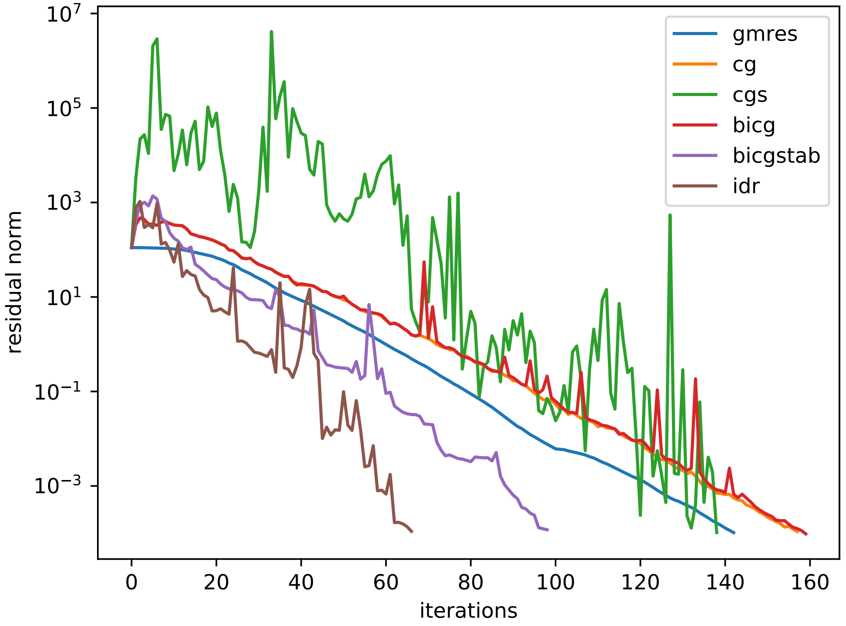
Open source, community-driven

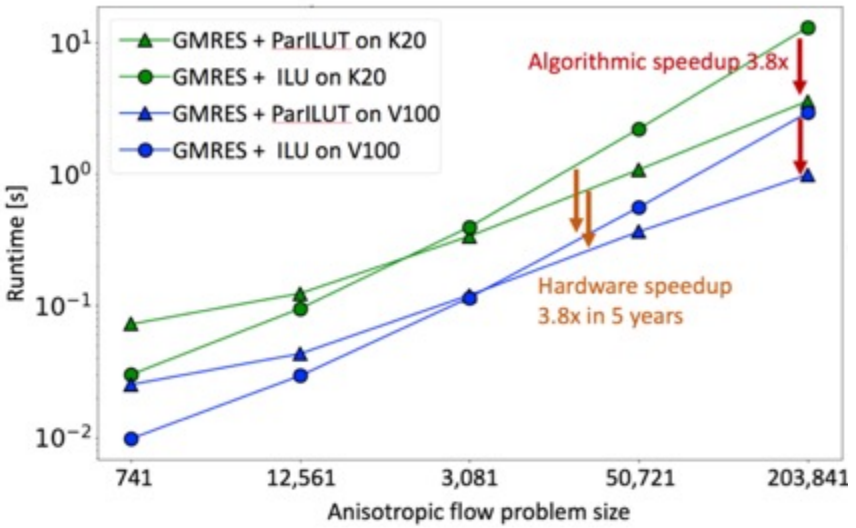
- ***Ginkgo is a child of the ECP effort***:
  - Lessons learnt in other software projects (e.g. MAGMA) are incorporated;
  - Focus on sparse linear algebra;
- **Better Scientific Software** (BSSW.io) proved to be a valuable resource;
- Freely available (BSD License), **GitHub**, and **Spack**;
- Modern 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;

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;
- 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
- ...

Ginkgo

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
- ...

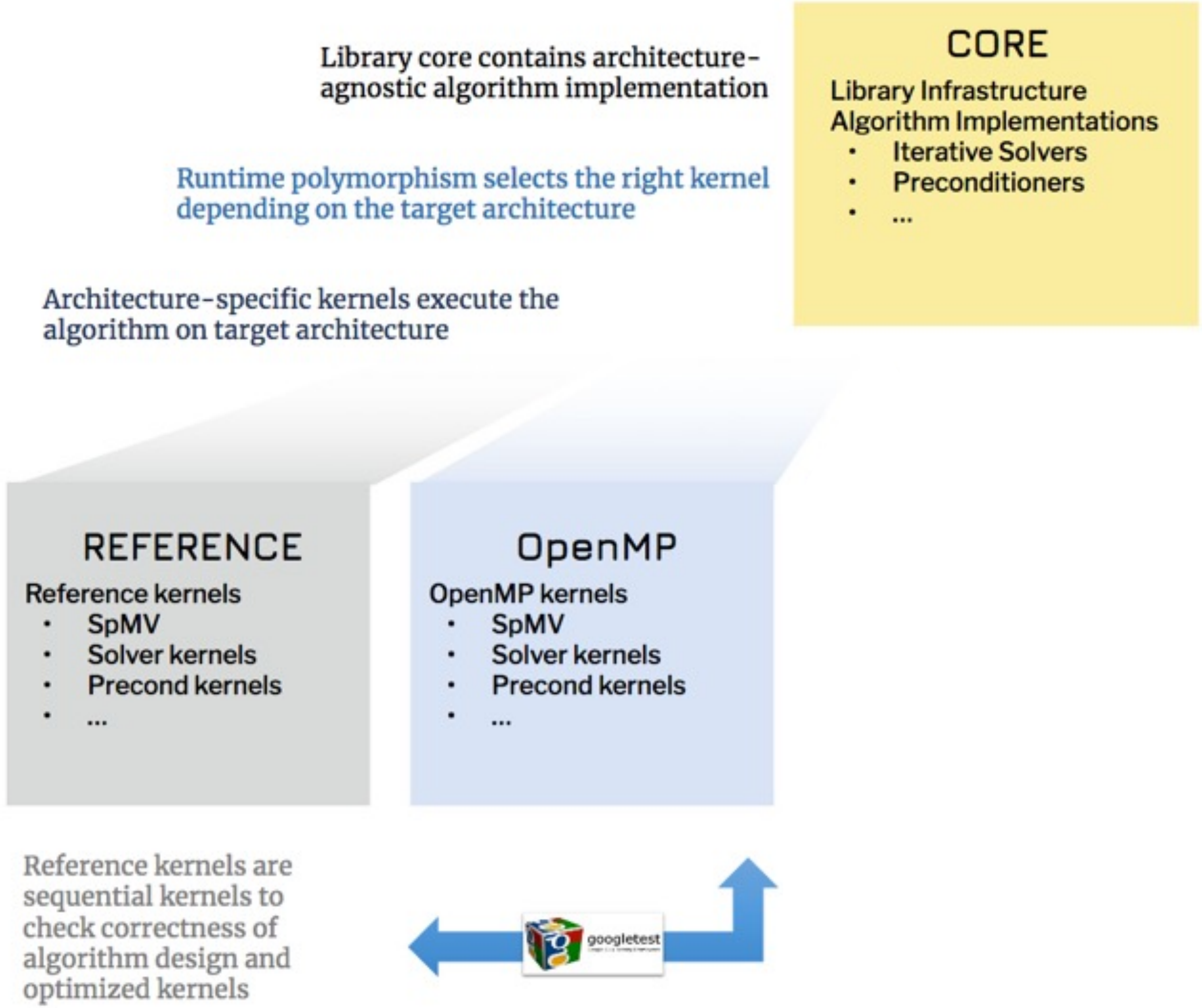
REFERENCE

Reference kernels

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

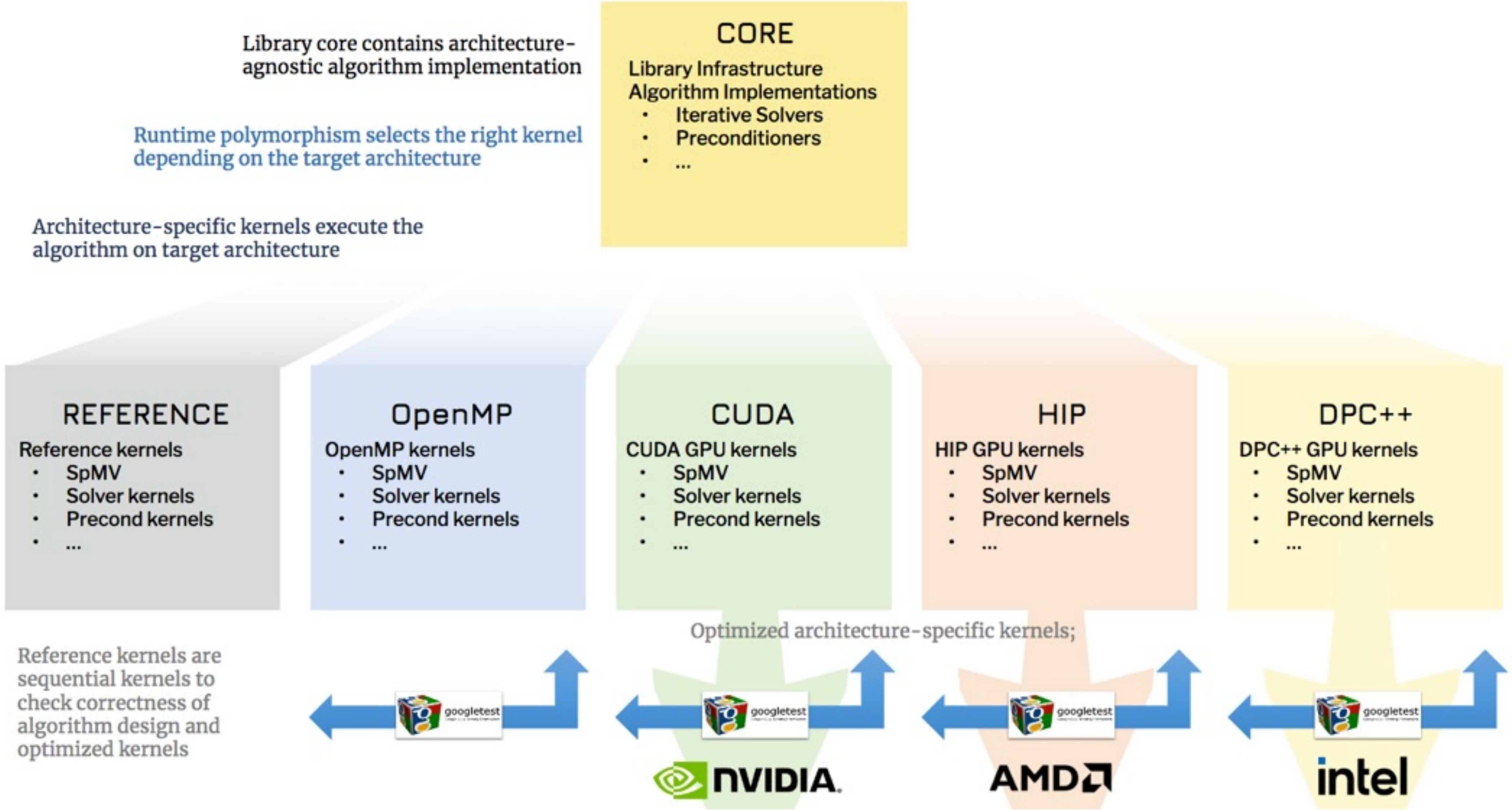
Reference kernels are sequential kernels to check correctness of algorithm design and optimized kernels

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



Ginkgo

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

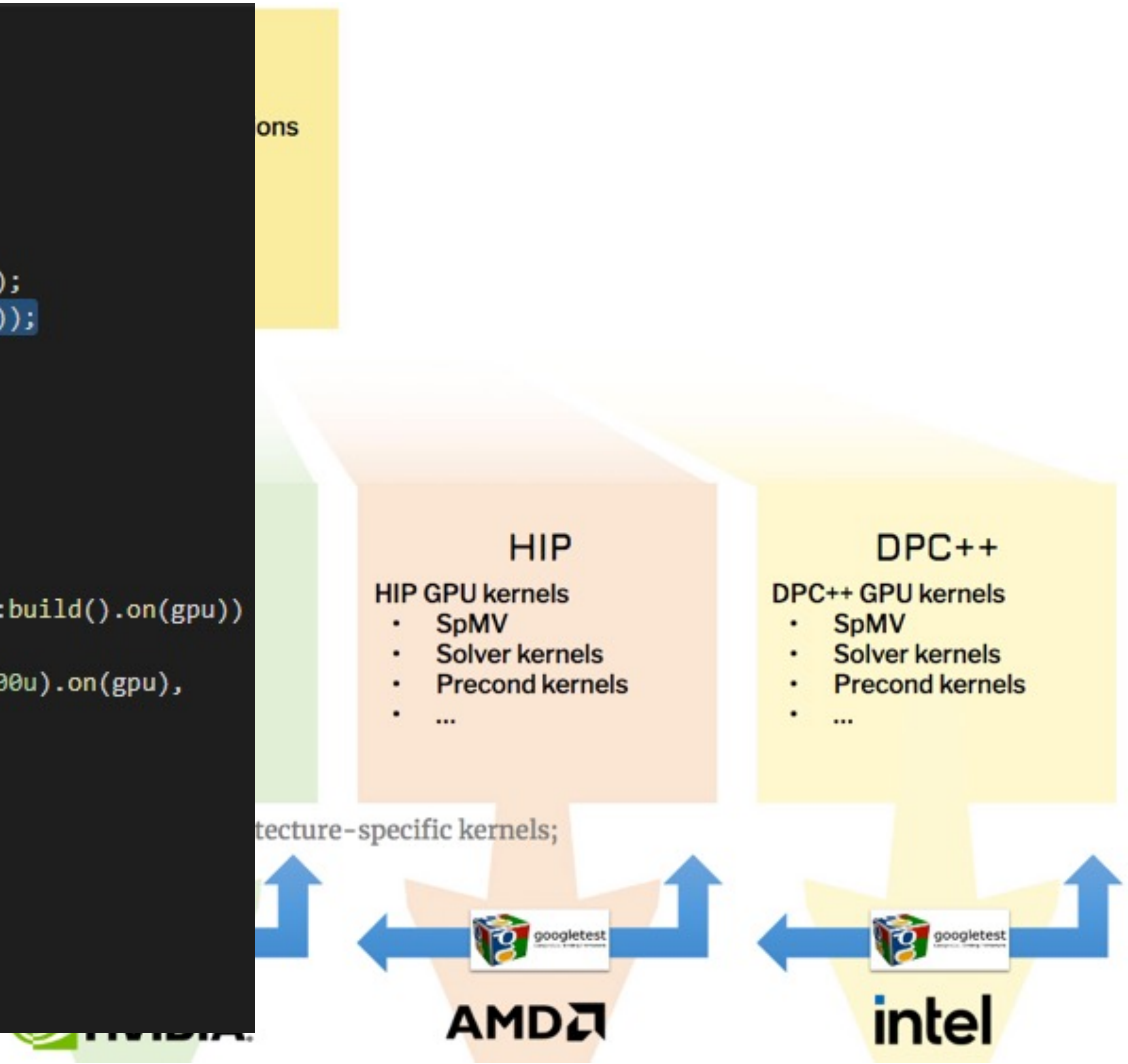


Ginkgo

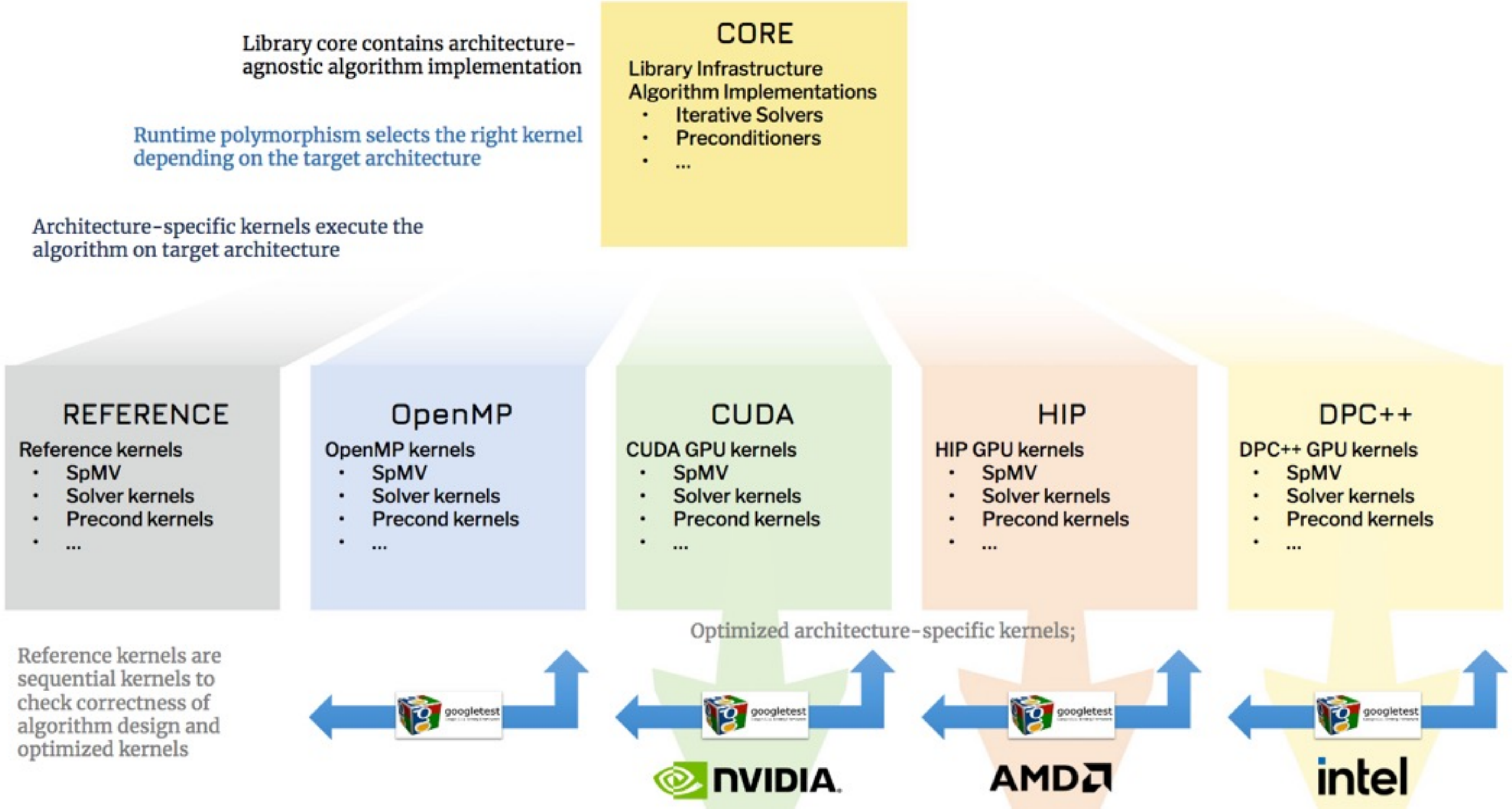
GPU-centric high performance sparse linear algebra. Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.

```cpp
#include <ginkgo/ginkgo.hpp>
#include <iostream>

int main()
{
    // Instantiate a GPU executor
    auto gpu =
-       gko::CudaExecutor::create(0, gko::OmpExecutor::create());
+       gko::DpcppExecutor::create(0, gko::OmpExecutor::create());
    // Read data
    auto A = gko::read<gko::matrix::Csr<>>(std::cin, gpu);
    auto b = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
    auto x = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
    // Create the solver
    auto solver =
        gko::solver::Cg<>::build()
            .with_preconditioner(gko::preconditioner::Jacobi<>::build().on(gpu))
            .with_criteria(
                gko::stop::Iteration::build().with_max_iters(1000u).on(gpu),
                gko::stop::ResidualNormReduction<>::build()
                    .with_reduction_factor(1e-15)
                    .on(gpu))
            .on(gpu);
    // Solve system
    solver->generate(give(A))->apply(lend(b), lend(x));
    // Write result
    write(std::cout, lend(x));
}
```

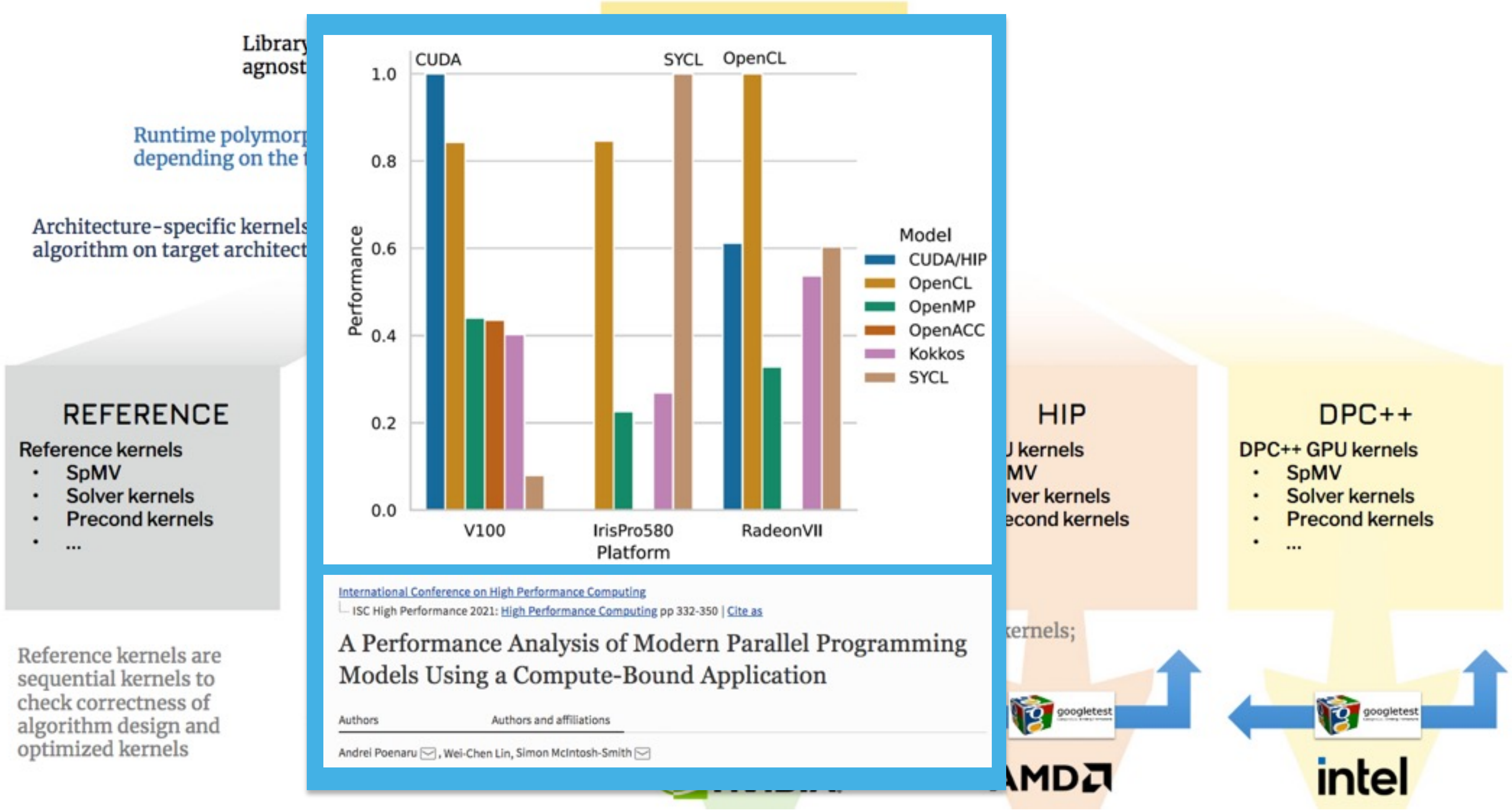


GPU-centric high performance sparse linear algebra. Sustainable and extensible C++ ecosystem with full support for AMD, NVIDIA, Intel GPUs.



Ginkgo

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



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

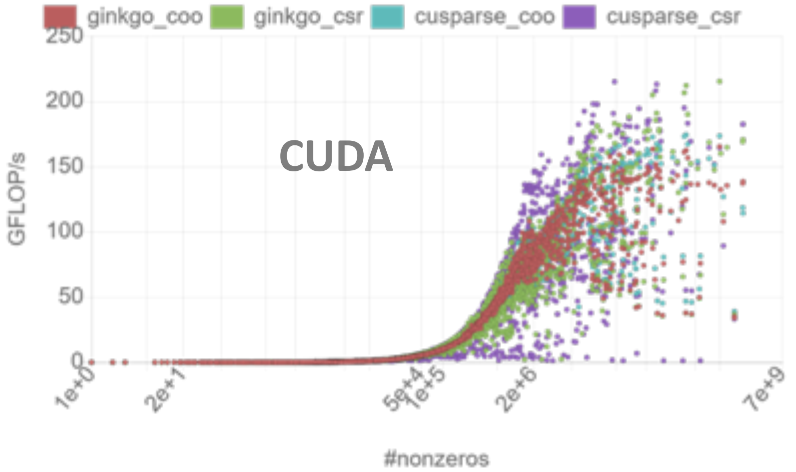
Authors | Authors and affiliations

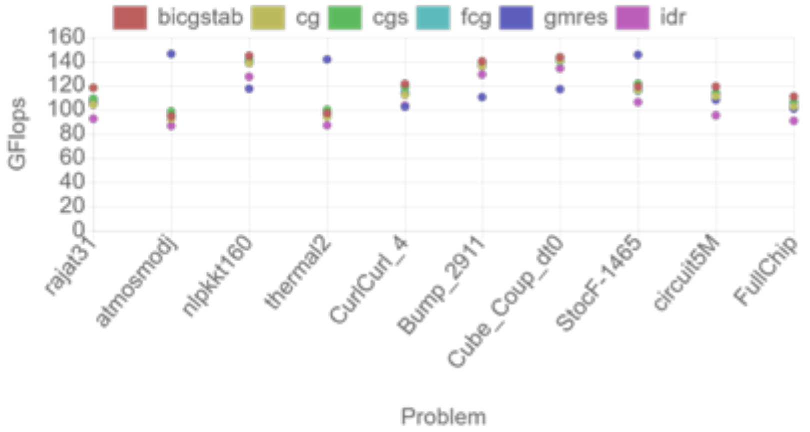
Andrei Poenaru, Wei-Chen Lin, Simon McIntosh-Smith

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

NVIDIA A100 GPU

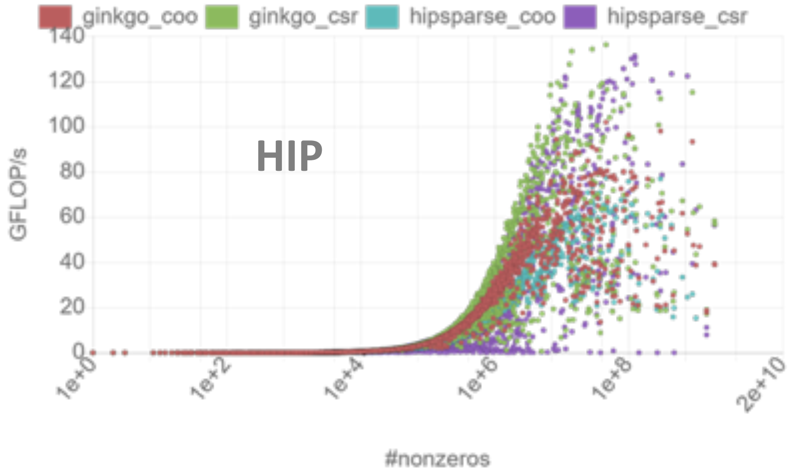
Perlmutter-type, external

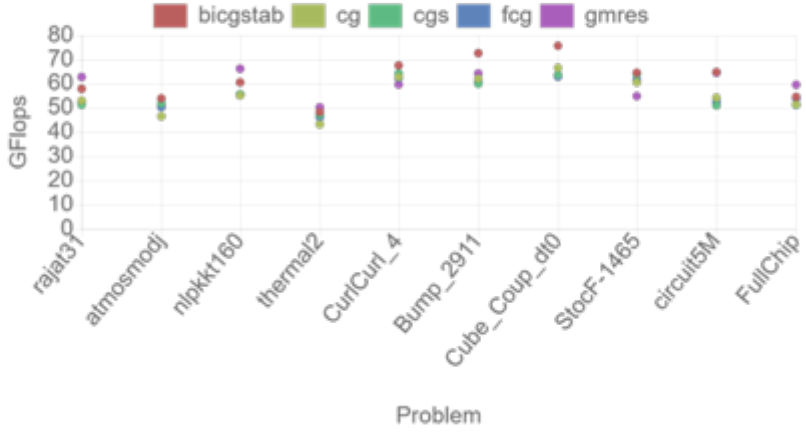




AMD MI100 GPU

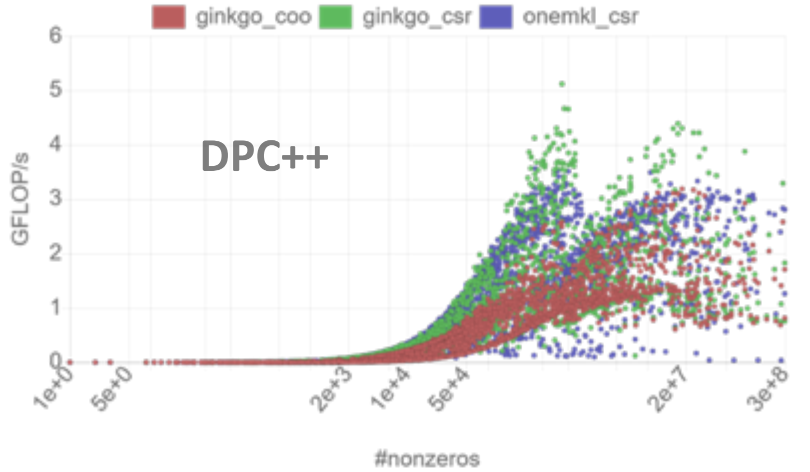
Pre-Frontier system Tulip (approved)

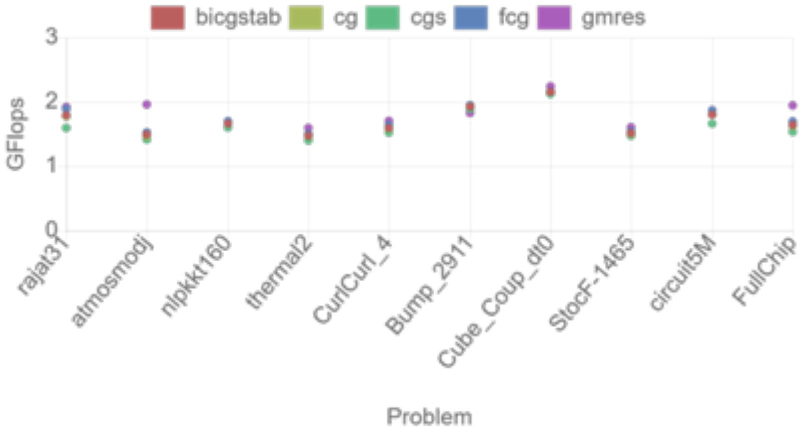




intel Gen9 GPU

integrated GPU





Test problem from Suite Sparse: https://people.engr.tamu.edu/davis/suitesparse.html

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 $Ax = b$)
$x := A \cdot b$	$x := M^{-1} \cdot b$	$x := S \cdot b$
	$M^{-1} \approx A^{-1}$	$S \approx A^{-1}$
	$M^{-1} = \Pi(A)$	$S = \Sigma(A)$

All of them can be expressed as

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

Test problem from Suite Sparse: https://people.engr.tamu.edu/davis/suitesparse.html

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

Linear Operator Design

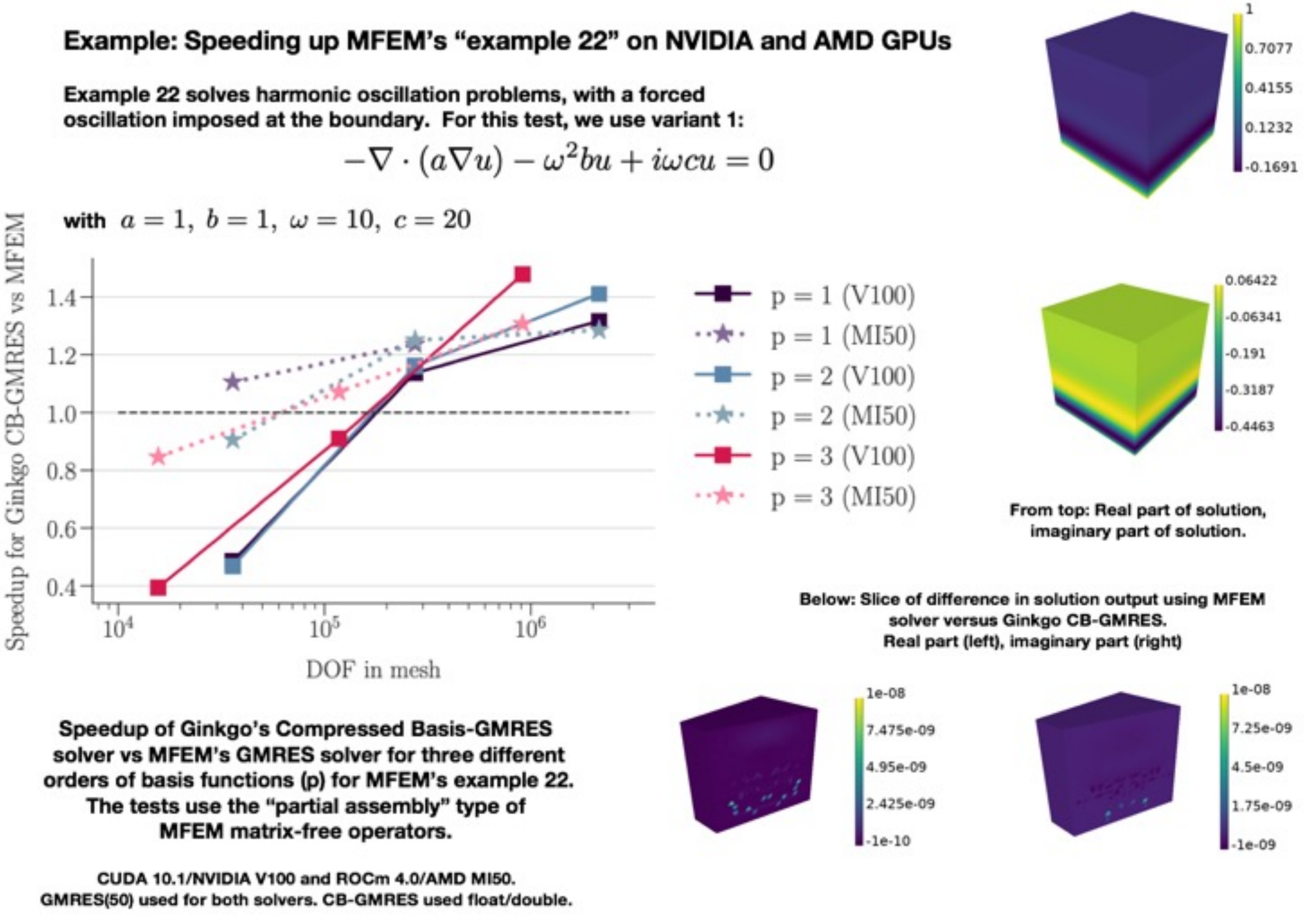
We express everything as Linear Operator and leverage

Matrix-Vector Product

$x := A \cdot b$

All of them can be expressed as

Application of a linear operator* (LinOp

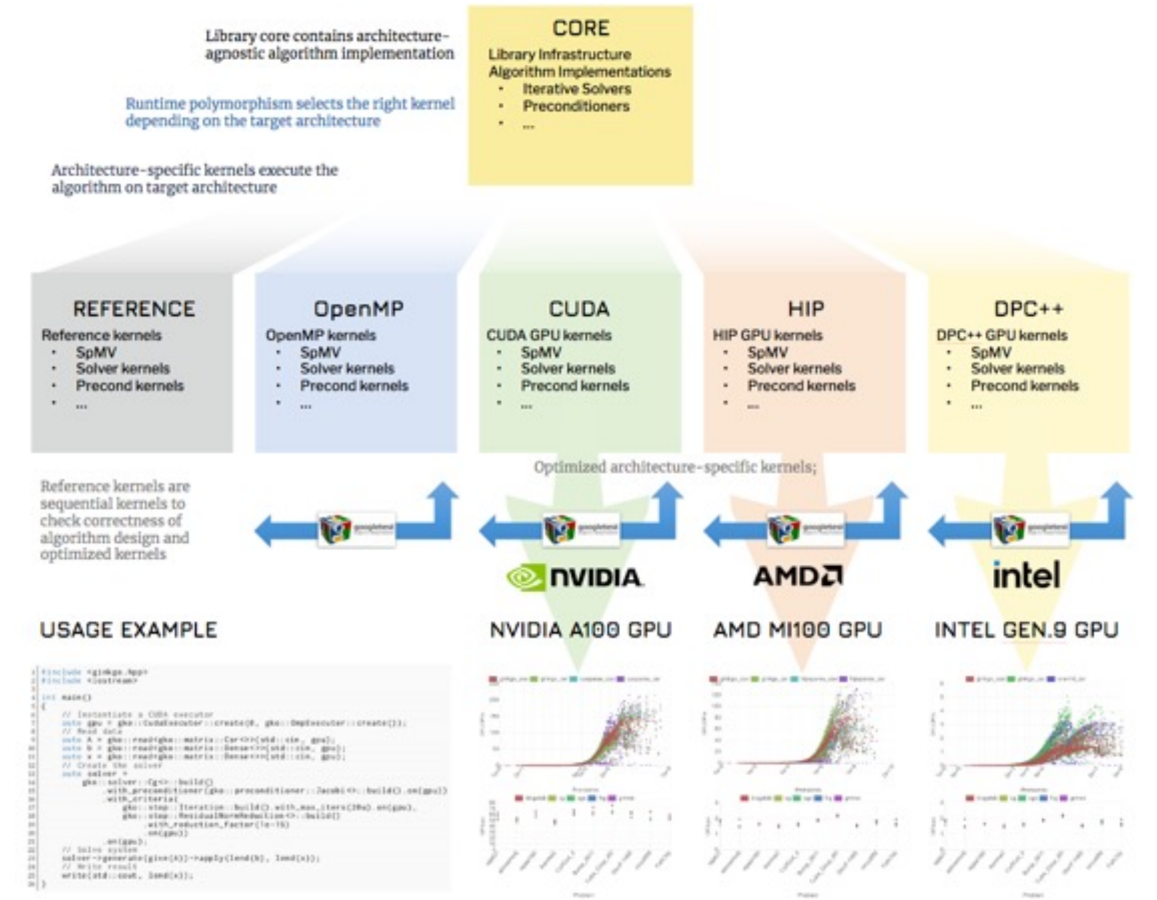


Test problem from Suite Sparse: https://people.engr.tamu.edu/davis/suitesparse.html

Ginkgo

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;
  - Mixed precision algorithms: adaptive precision Jacobi, FSPAI;
  - Decoupling of arithmetic precision and memory precision;
  - Batched iterative solvers;
  - Extensible, sustainable, production-ready;
- **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**;
  - Working on **SUNDIALS** integration;



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

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

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;

Ginkgo

"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

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

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 * Lp -> Hp
auto lplp_sec = timing(exec, lp_A, lp_b, lplp_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

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 * Lp -> Hp
auto lplp_sec = timing(exec, lp_A, lp_b, lplp_x);
// Lp * Hp -> Hp
auto lphp_sec = timing(exec, lp_A, hp_b, lphp_x);
```

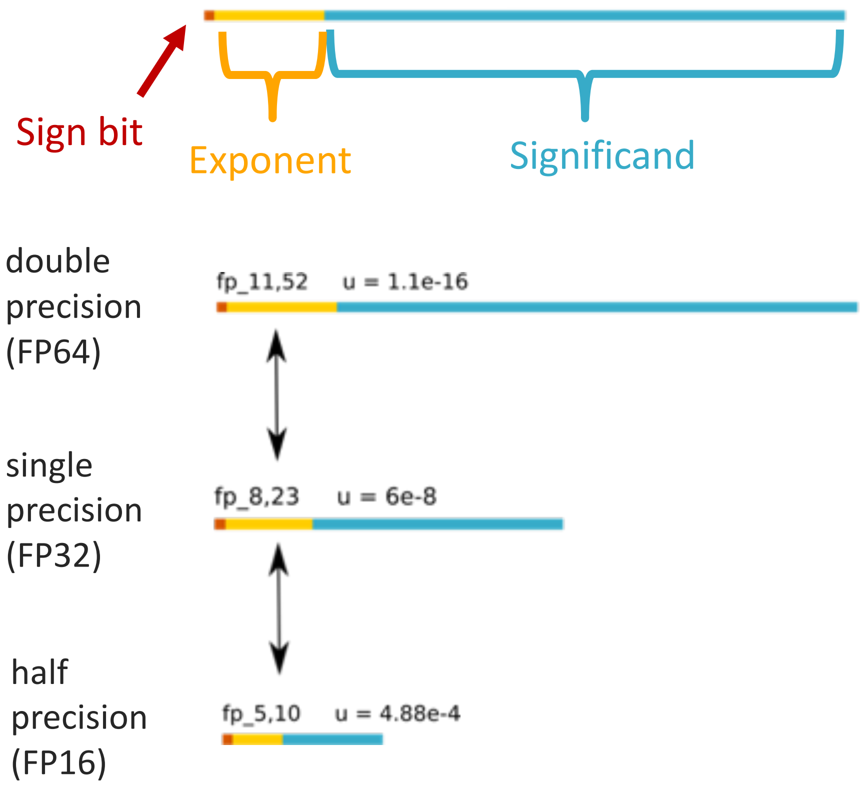
3. Investigate rounding effects

```
High Precision time(s): 9.8980000000e-07
High Precision result norm: 2.5547848401e+05
Low Precision time(s): 9.8890000000e-07
Low Precision relative error: 5.5253439244e-08
Hp * Lp -> Hp time(s): 1.3829000000e-06
Hp * Lp -> Hp relative error: 1.6328092846e-08
Lp * Lp -> Hp time(s): 1.3761000000e-06
Lp * Lp -> Hp relative error: 2.5540873856e-08
Lp * Hp -> Hp time(s): 1.3761000000e-06
Lp * Hp -> Hp relative error: 3.7166469483e-08
```

ginkgo -> examples -> mixed-spmv

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

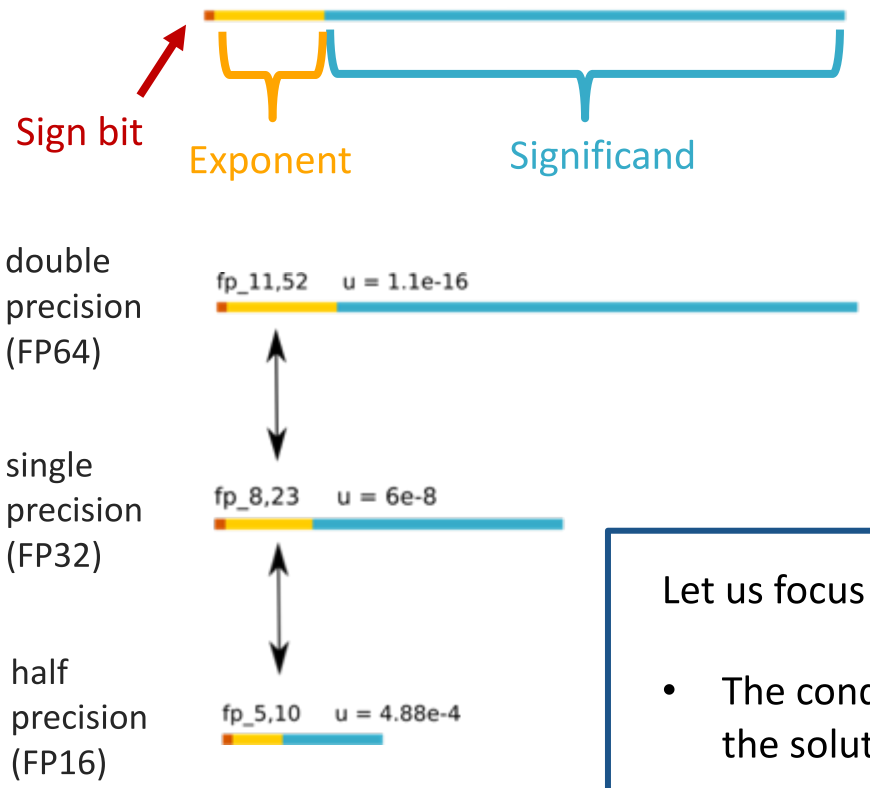
Background: Floating Point Formats, Accuracy, and Performance



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:

relative residual accuracy = **(unit round-off)** * **(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 cond(A) ≈ 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

Ginkgo

Linear System Ax=b with cond(A) ≈ 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
```

Accuracy improvement ~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

Running iterative methods in different precision formats

Linear System Ax=b with cond(A) ≈ 10^4

Double Precision

```
Initial residual norm sqrt(r^T r):
%%MatrixMarket matrix array real g
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
```

Accuracy improvement ~10^{12}

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

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

Running iterative methods in different precision formats

Ginkgo

Linear System Ax=b with 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
```

Accuracy improvement ~10^{12}

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
```

Accuracy improvement ~10^3

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

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

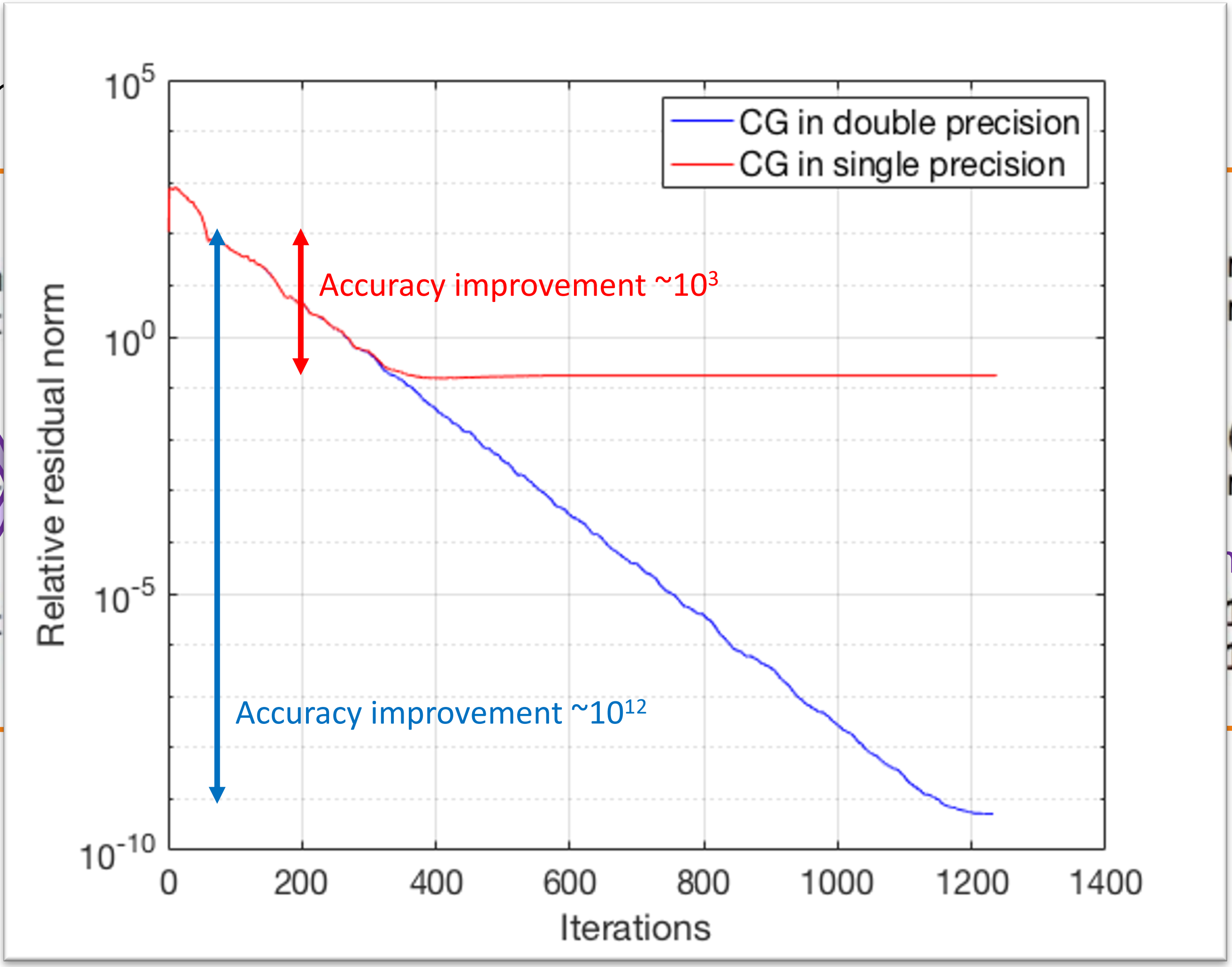
Experiments based on the Ginkgo library https://ginkgo-project.github.io/
ginkgo/examples/adaptiveprecision-blockjacobi/adaptiveprecision-blockjacobi.cpp

Running iterative methods in different precision formats

Linear System Ax=b with

Double Precision

```
Initial residual n
%%MatrixMarket mat
1 1
111.127
Final residual no
%%MatrixMarket mat
1 1
5.0775e-10
CG iteration count
CG execution time
```




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

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

mprovement ~10^3

```
1234
127.152
```

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

Ginkgo

Linear System Ax=b with cond(A) ≈ 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

Running iterative methods in different precision formats

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

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

Ginkgo

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

Double Precision CG + Double Precision I

```
Initial residual norm sqrt(r^T r)
%%MatrixMarket matrix array real
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

```
…
   - 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

Ginkgo

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

Double Precision CG + Double Precision Preconditioner

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

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

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

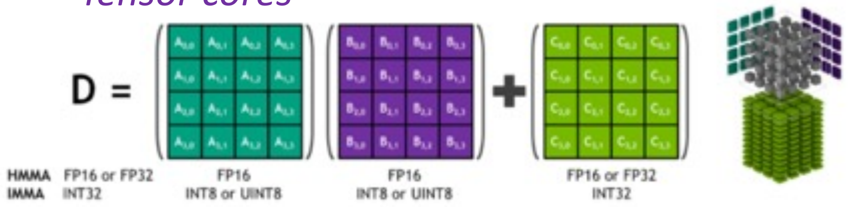
Why are we faster with a single precision CG solver?

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

**Tensor cores*



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.

Why are we faster with a single precision CG solver?

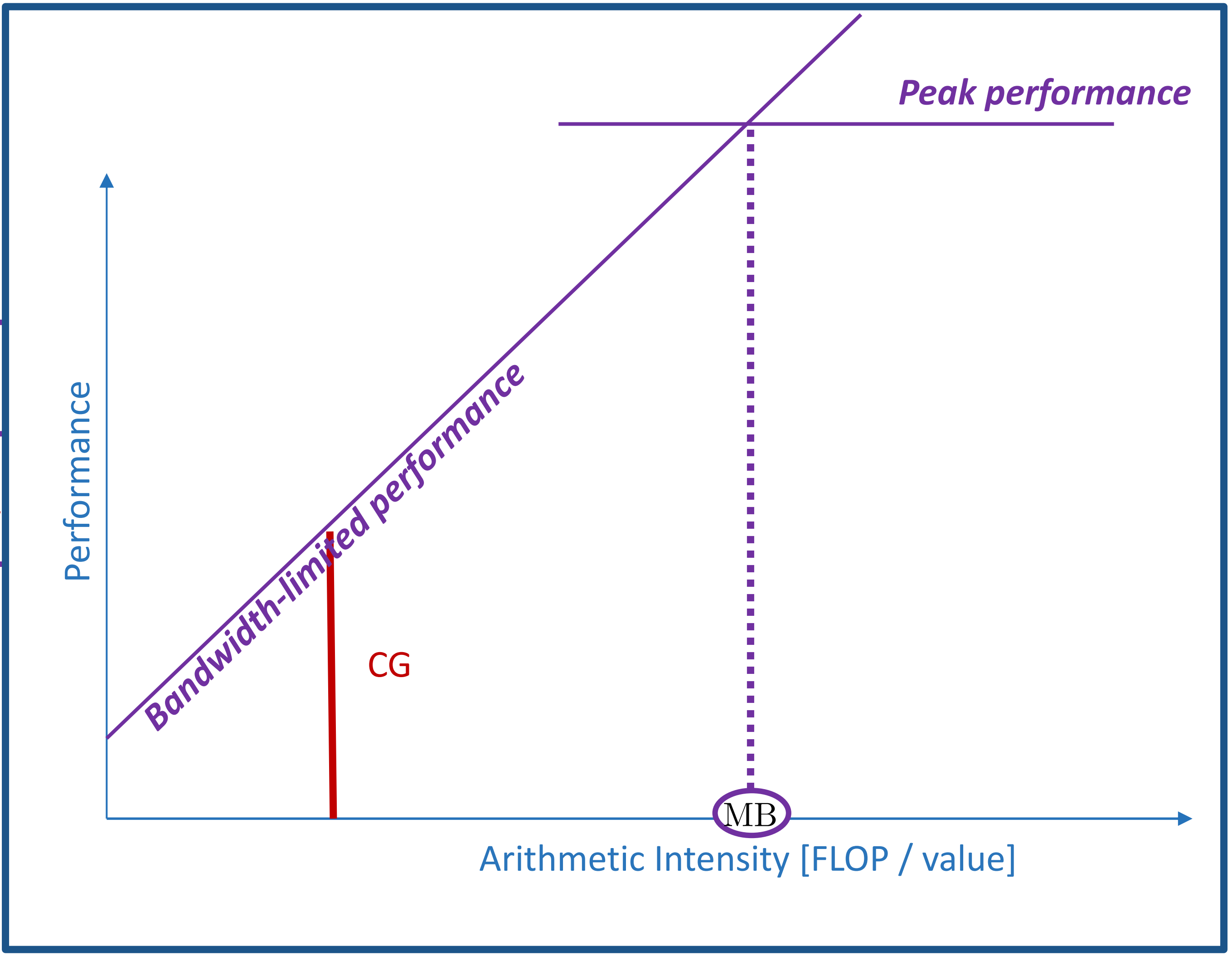
	2006	2007	2008	2009	2010	2011	2012
NVIDIA GPU generation			Tesla	Fermi			Keple
Rel. compute performance			1 : 2	1 : 2			1 : 2
Rel. memory performance			1 : 2	1 : 2			1 : 2

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

Differences between Volta, Maxwell, Tesla...

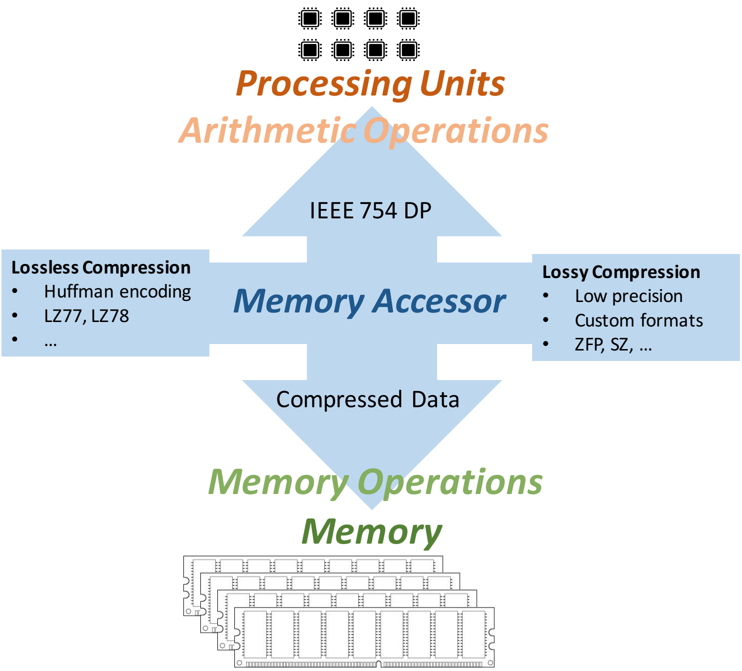
For **memory-bound applications**, the performance gains from using lower precision are **architecture-independent** and correspond to the floating point format complexity (#bits).

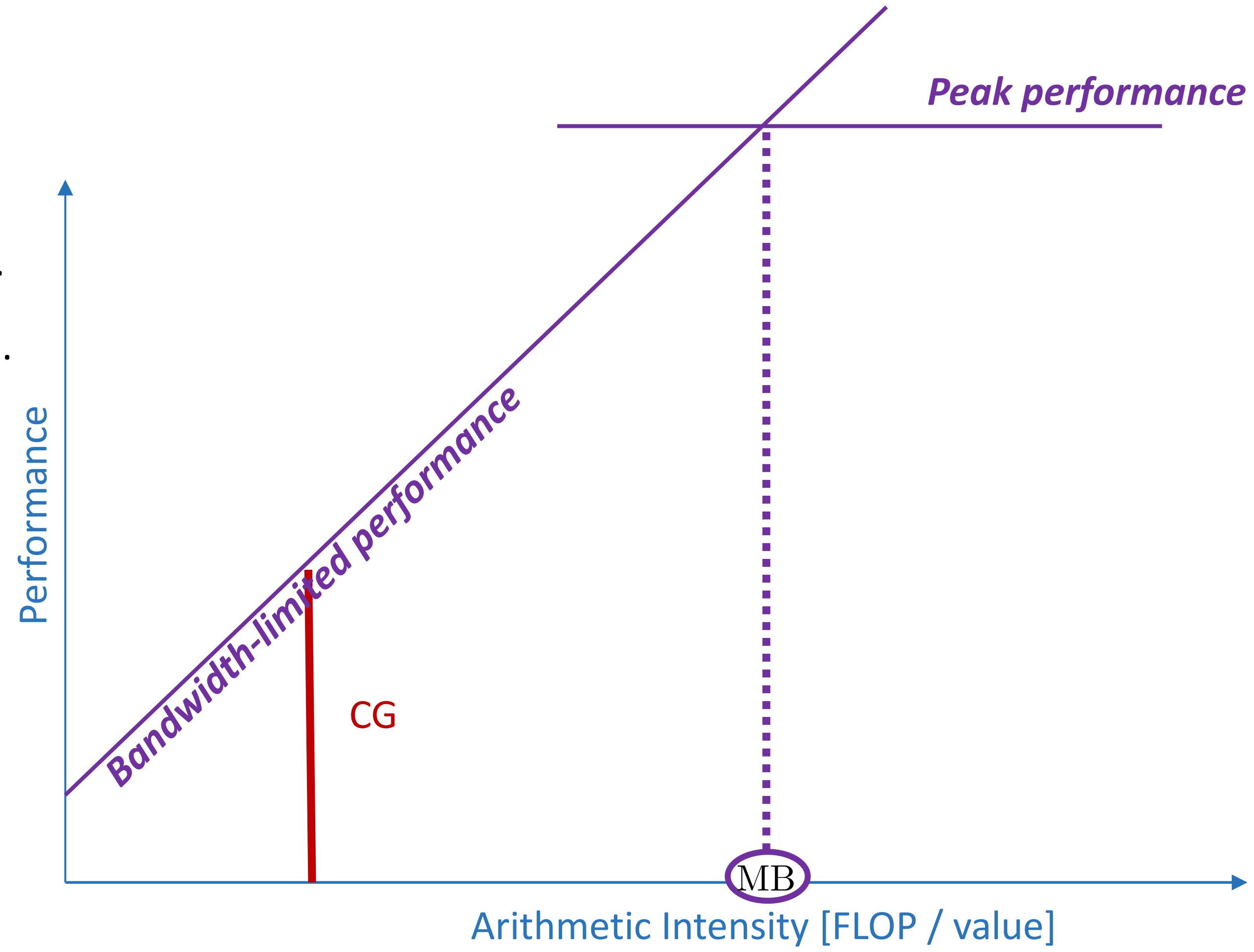
Generally, 2x for FP32, 4x for FP16.



Decoupling the memory precision from the arithmetic precision

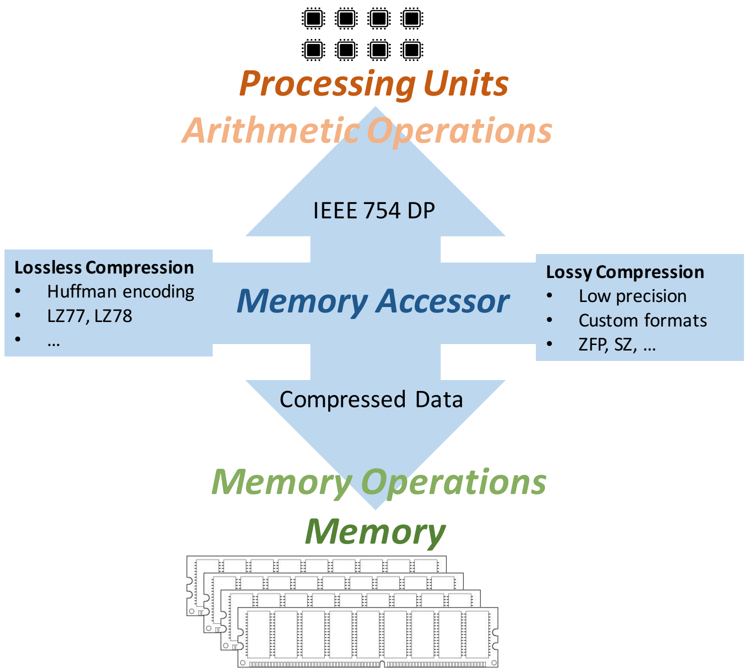
- Traditionally, we use a strong coupling between the precision formats used for arithmetic operations and storing data.
- The arithmetic operations are free, use high precision formats.
- **Data access** should be as cheap as possible, **reduced precision**.

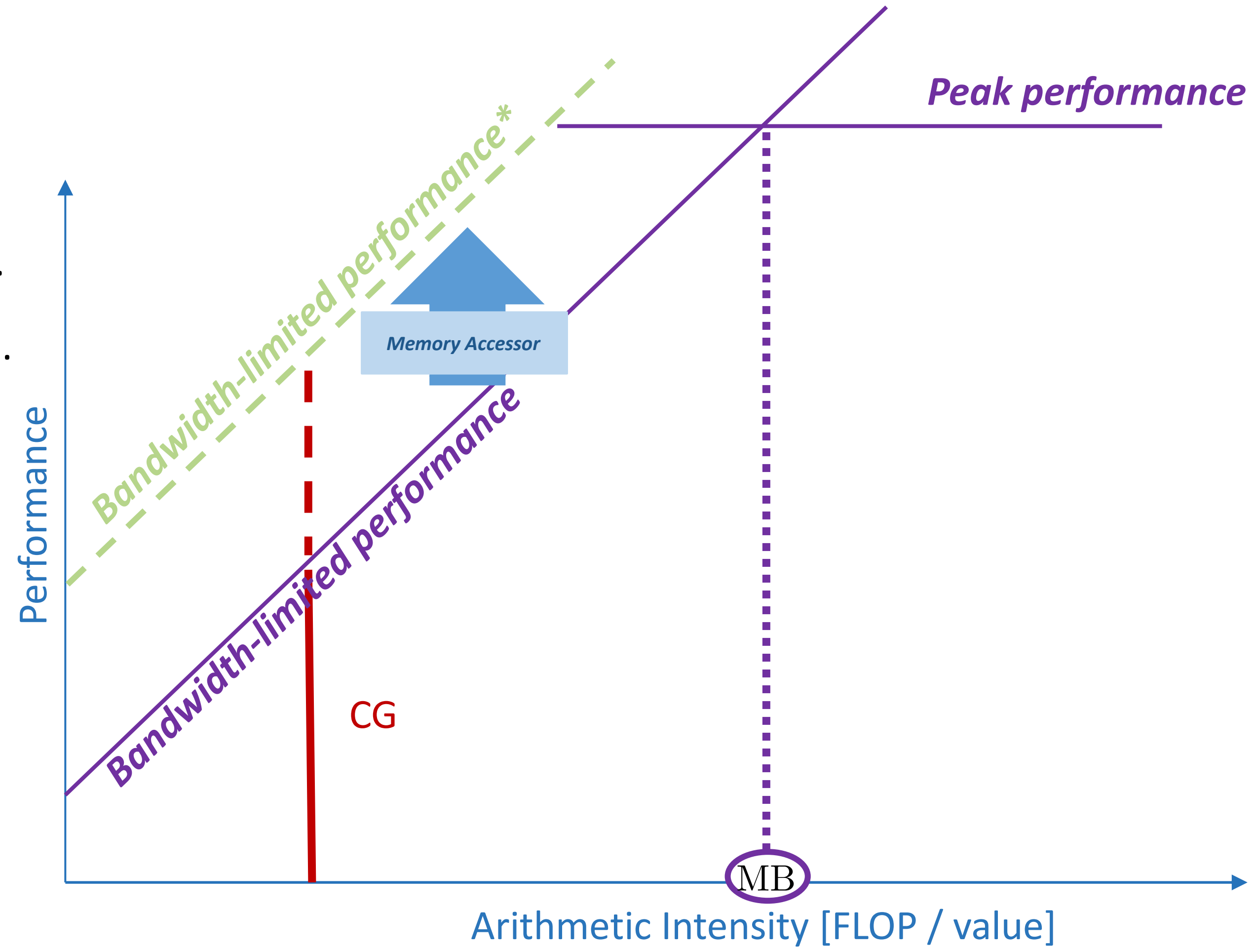




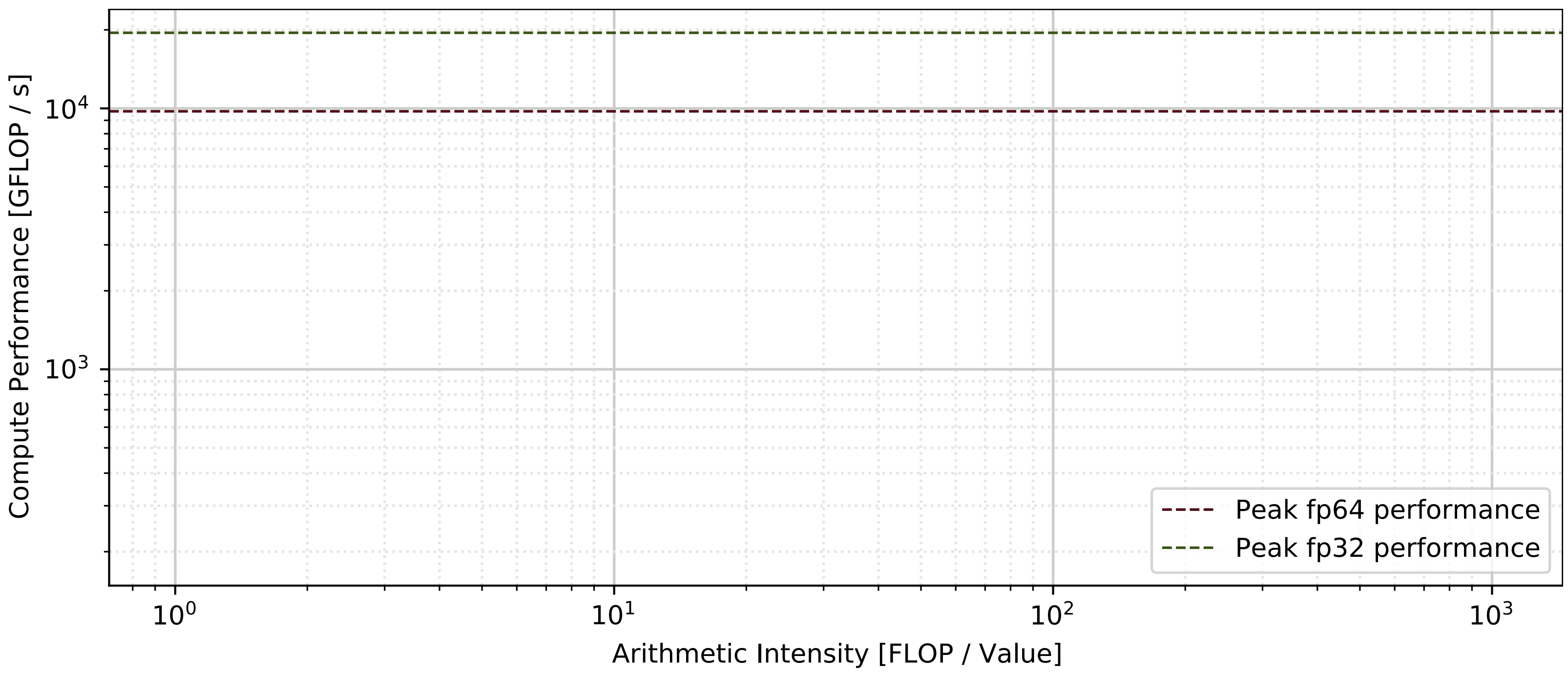
Decoupling the memory precision from the arithmetic precision

- 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**.

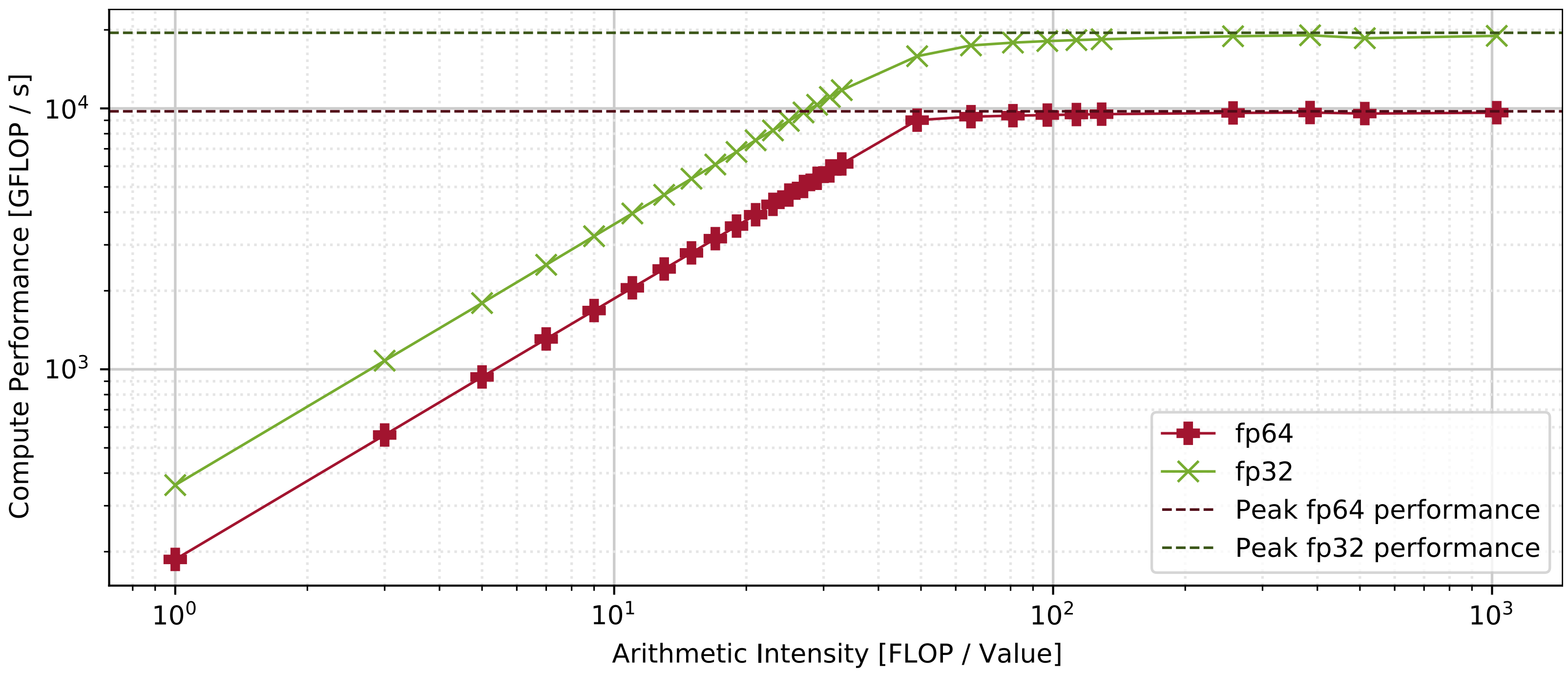




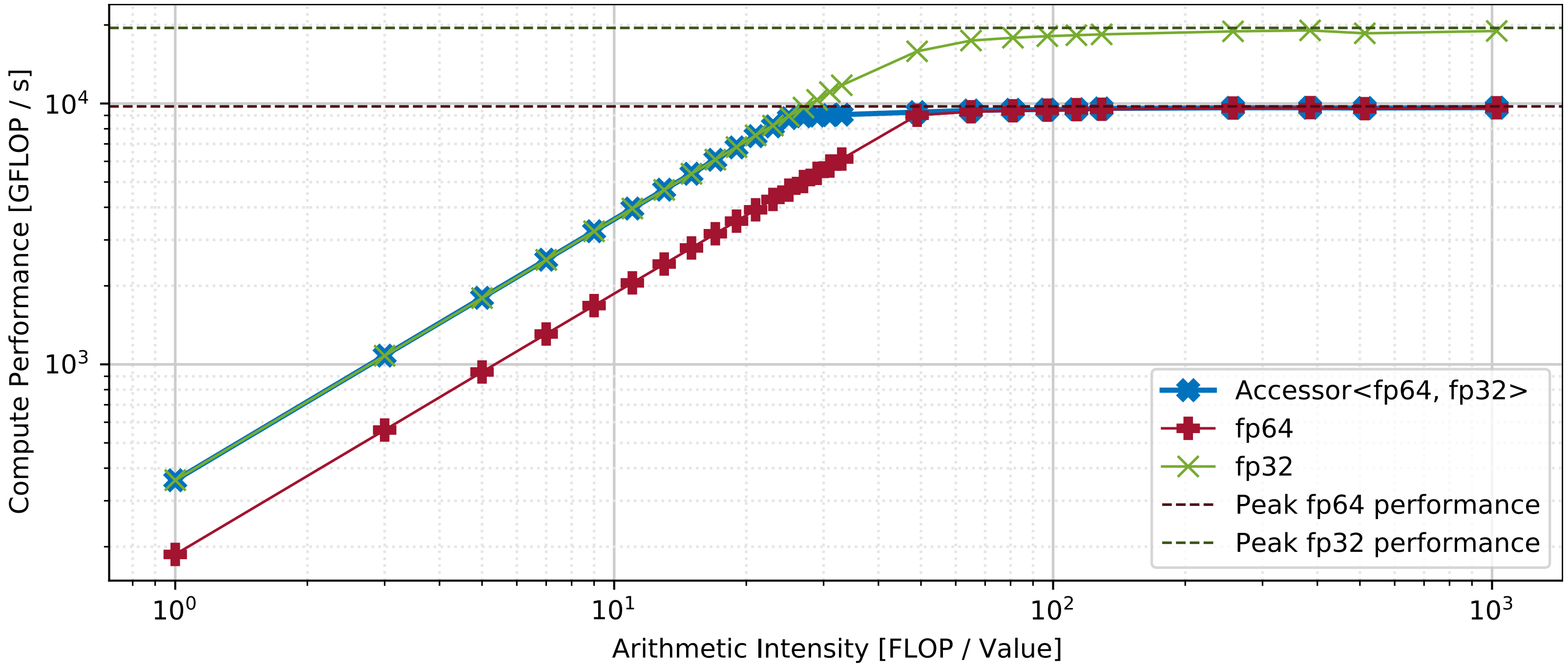
Memory Accessor for NVIDIA A100 GPU



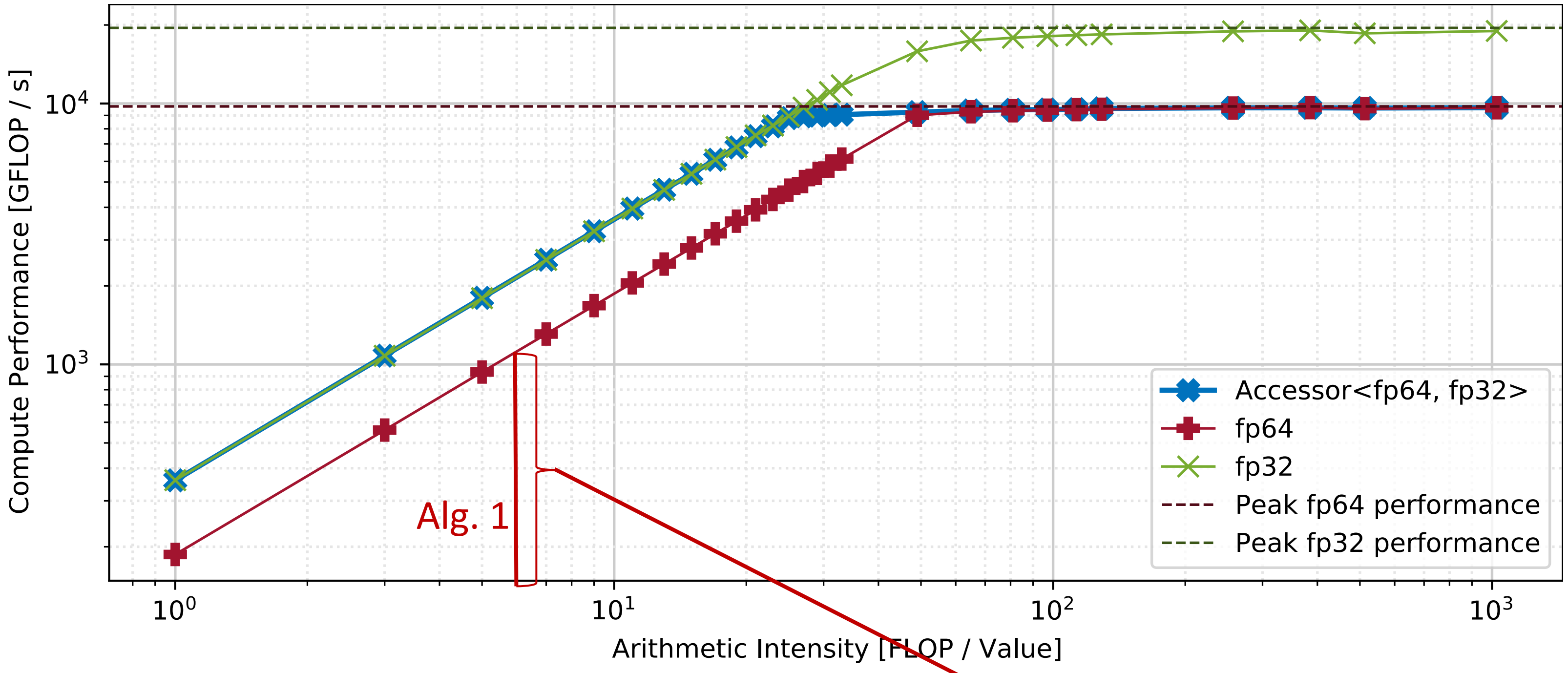
Memory Accessor for NVIDIA A100 GPU



Memory Accessor for NVIDIA A100 GPU

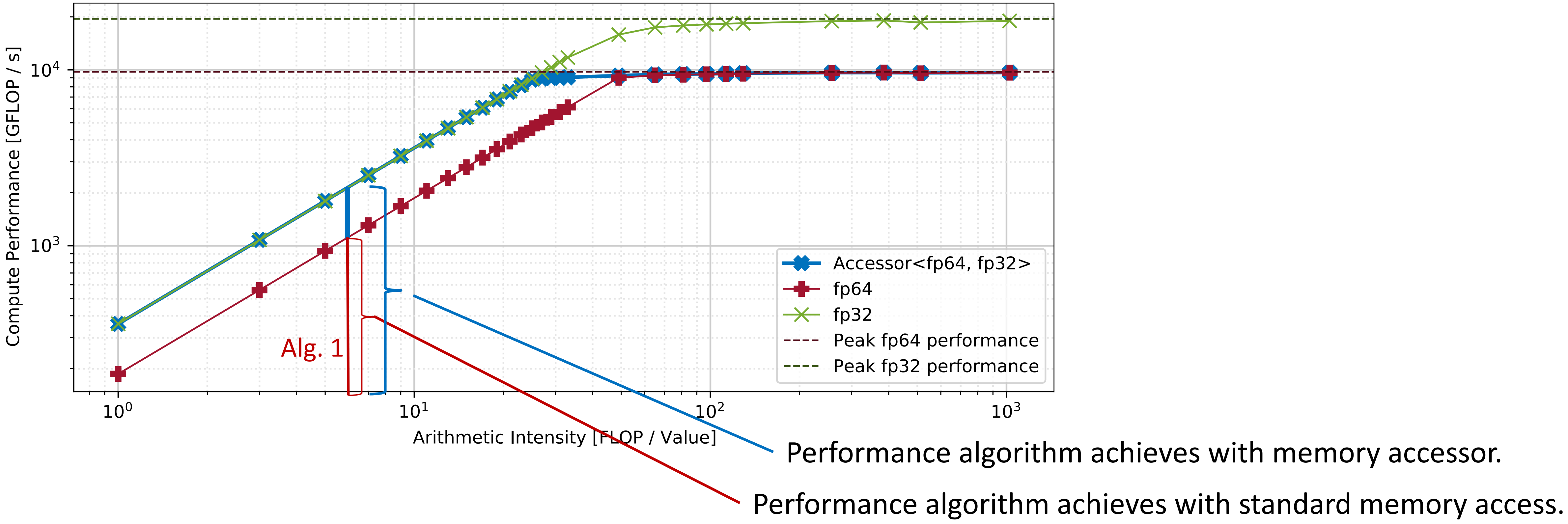


Memory Accessor for NVIDIA A100 GPU



Performance algorithm achieves with standard memory access.

Memory Accessor for NVIDIA A100 GPU



Rethinking Algorithms: Use the memory accessor to increase accuracy

Can we use the memory accessor to reduce the rounding effects?

Design

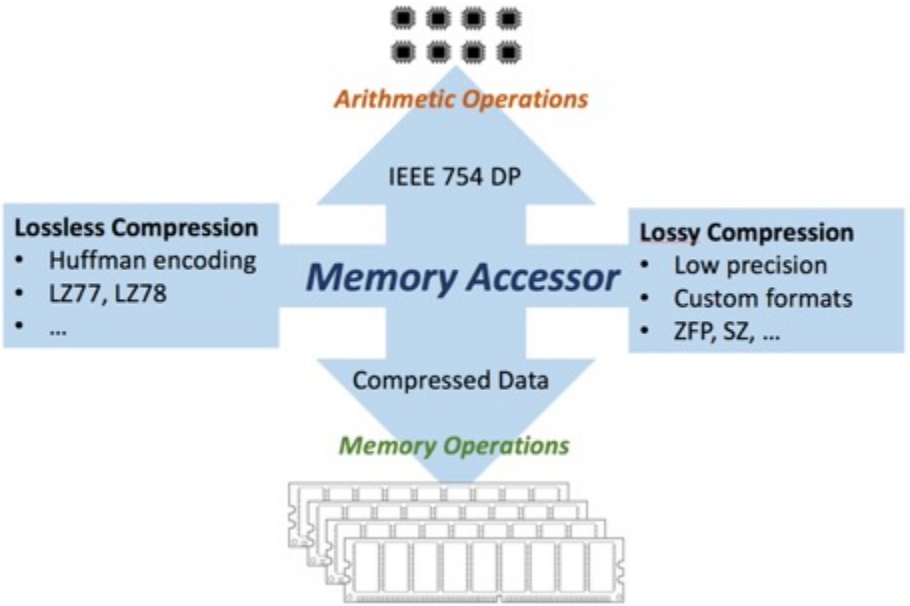
- Memory access in low precision (e.g. fp32);
- Computations in high precision (e.g. fp64);

Characteristics

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

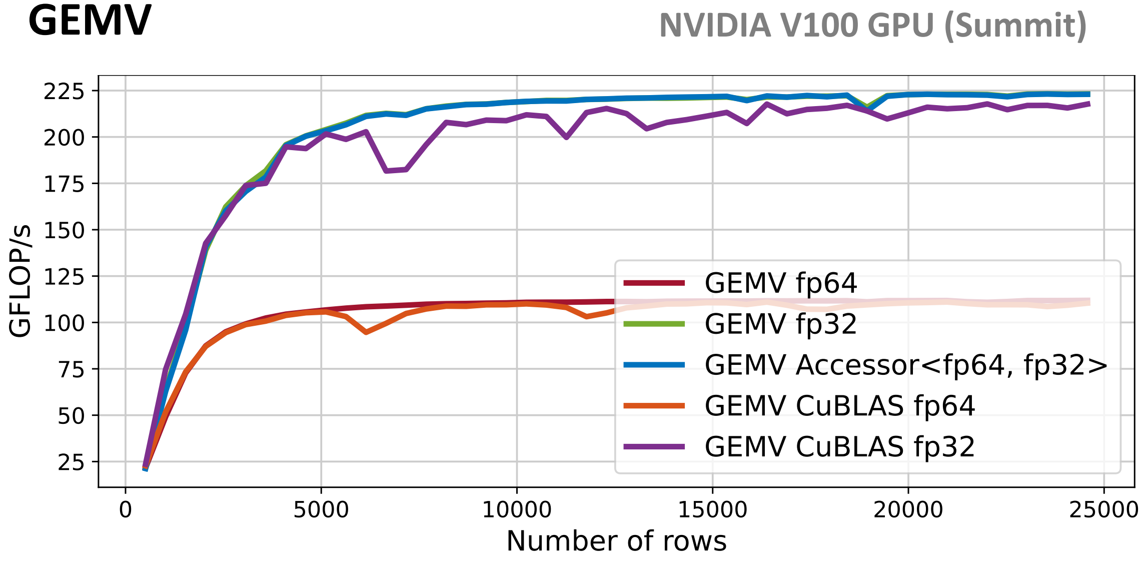
Usage

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

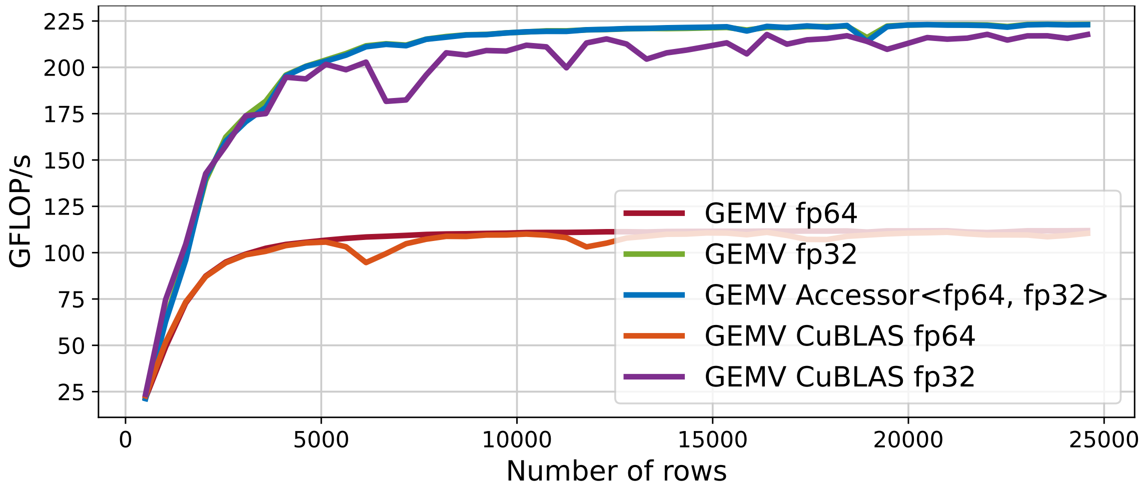
GEMV

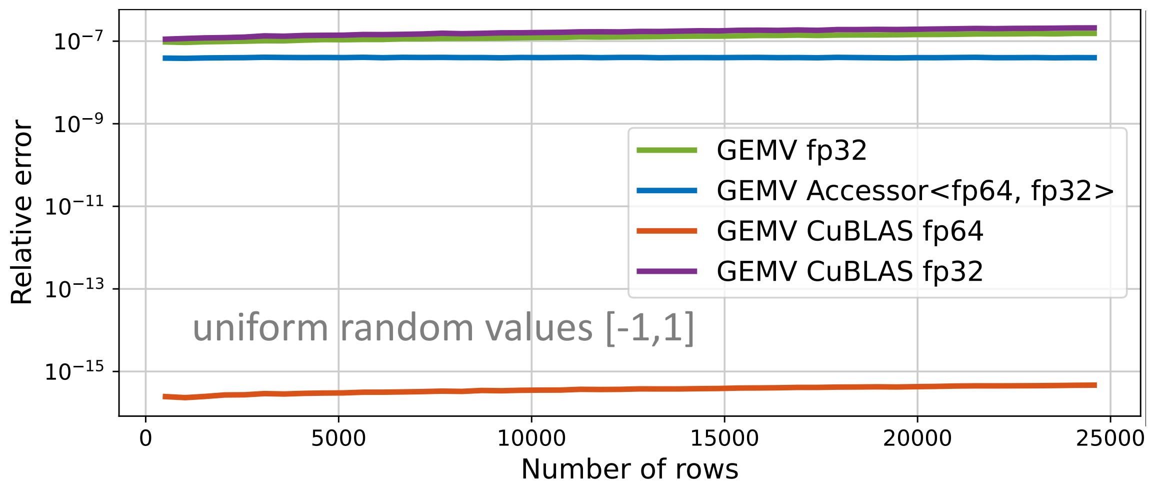


Accessor-BLAS: Replacing LP BLAS to improve accuracy

GEMV

NVIDIA V100 GPU (Summit)

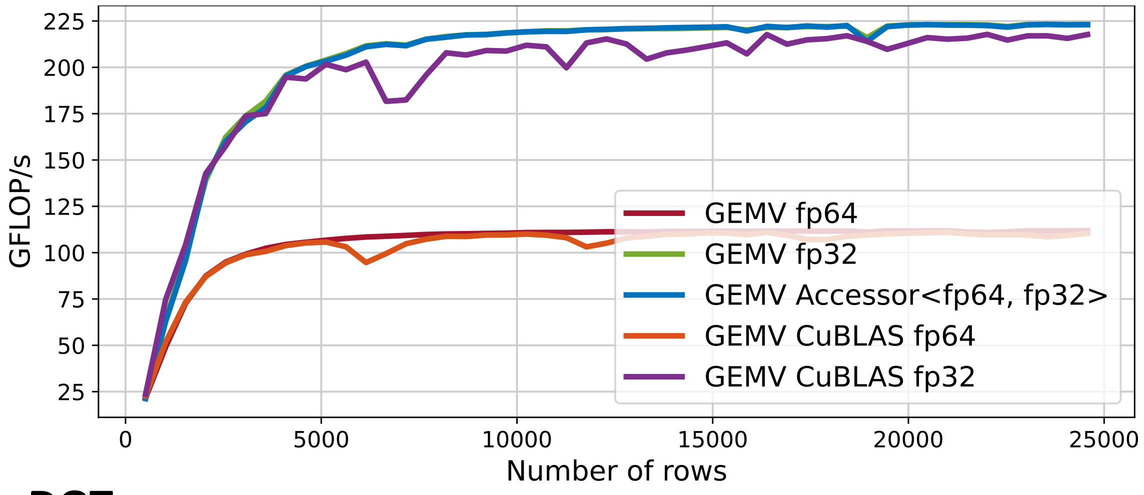


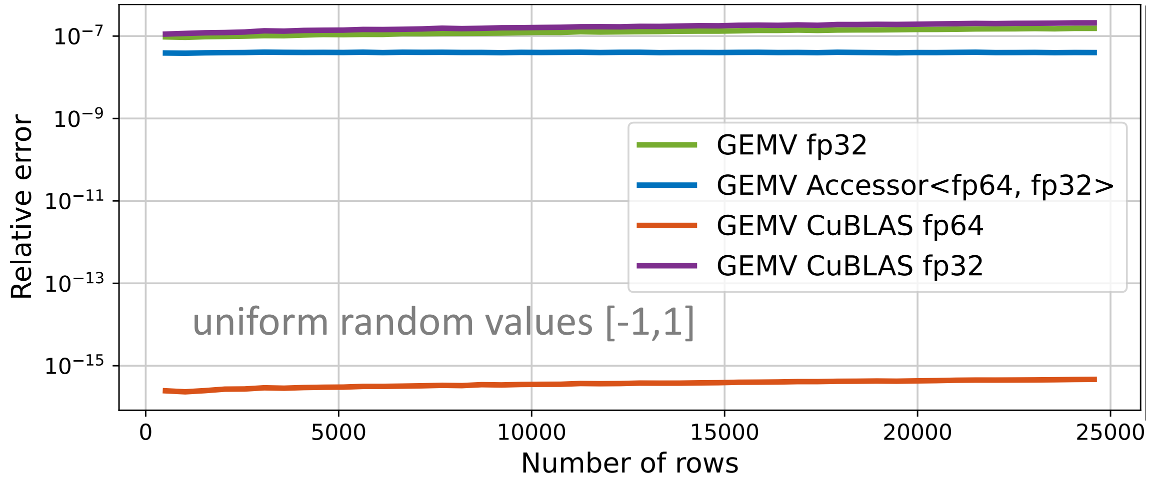


Accessor-BLAS: Replacing LP BLAS to improve accuracy

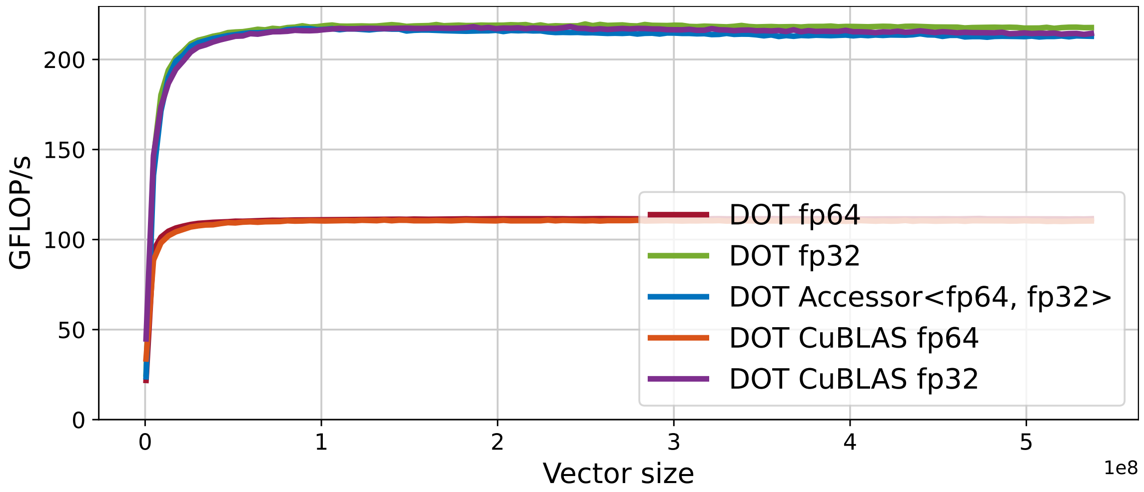
GEMV

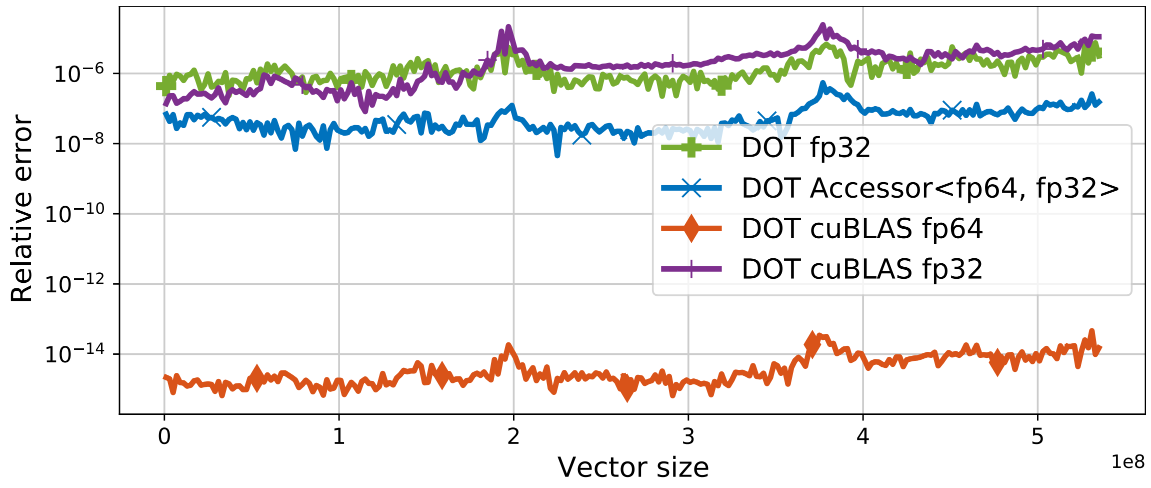
NVIDIA V100 GPU (Summit)





DOT





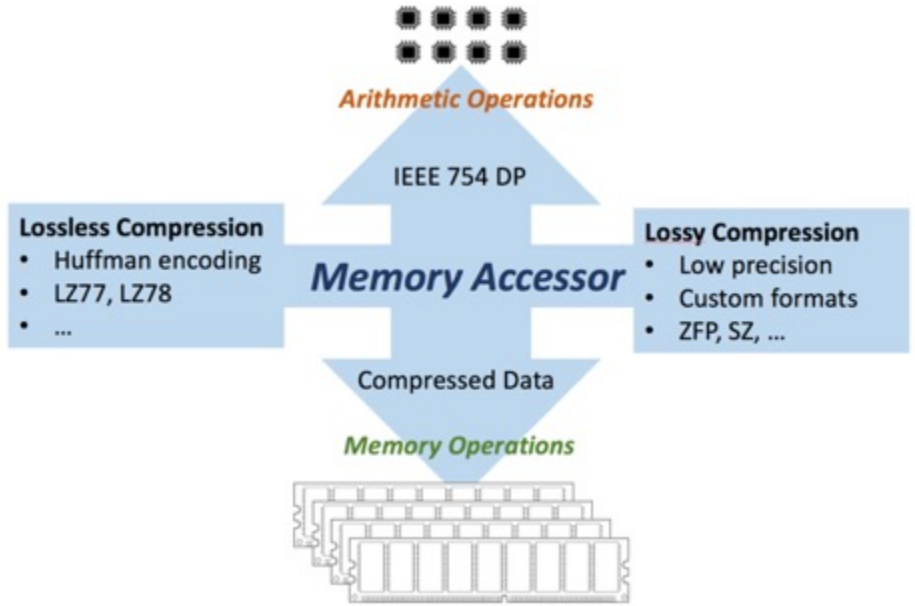
Rethinking Algorithms: Use the memory accessor to boost performance

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;*

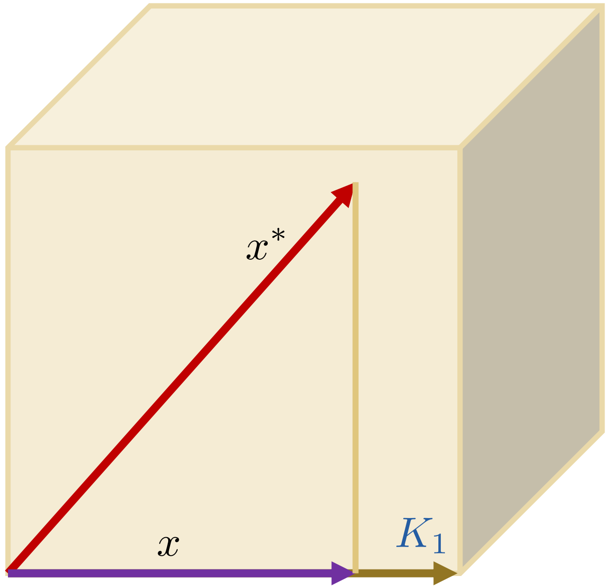


Rethinking Algorithms I: Self-Healing Iterative Methods

- **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…).

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

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



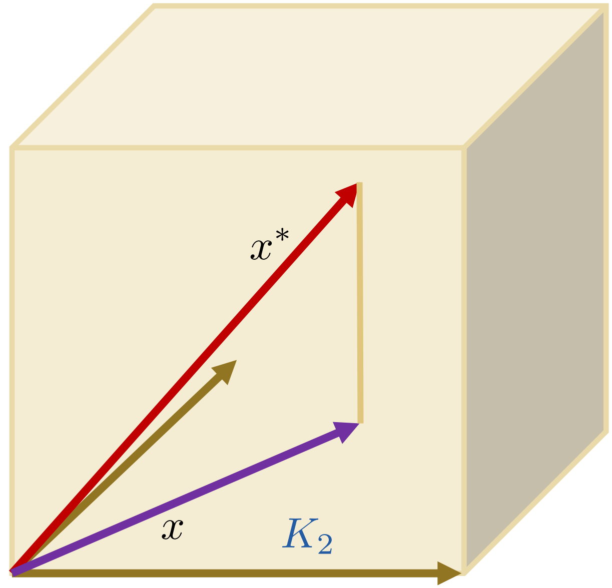
*GMRES in the modular precision ecosystem

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…).

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

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



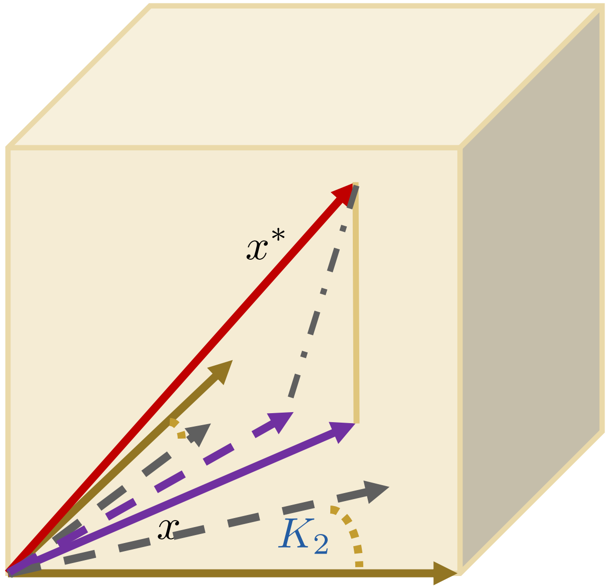
**GMRES in the modular precision ecosystem*

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…).

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

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



Compressed Basis (CB-) GMRES

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

Compressed Basis (CB-) GMRES

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

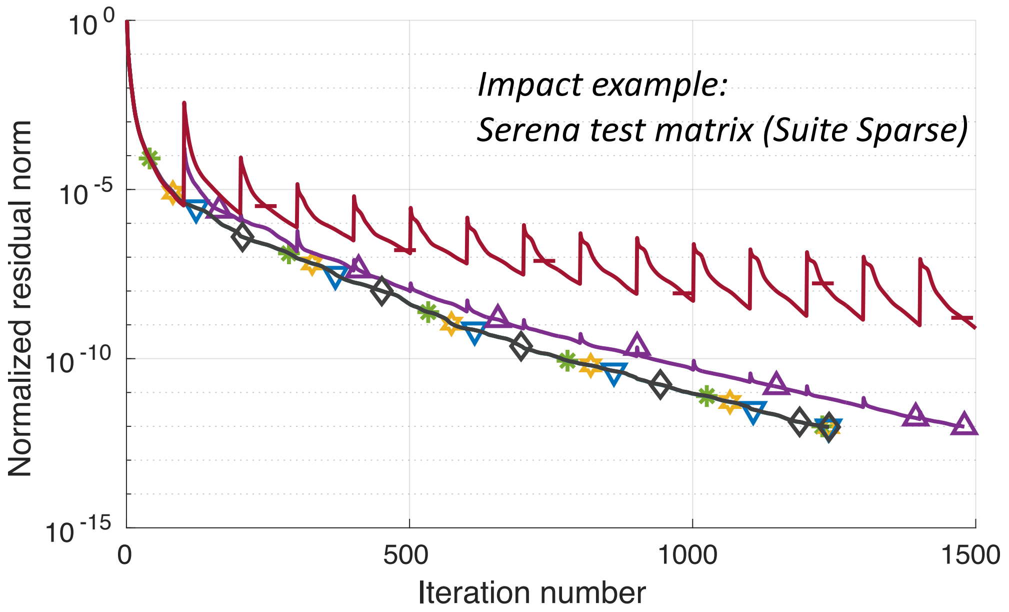
Compressed Basis (CB-) GMRES

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

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

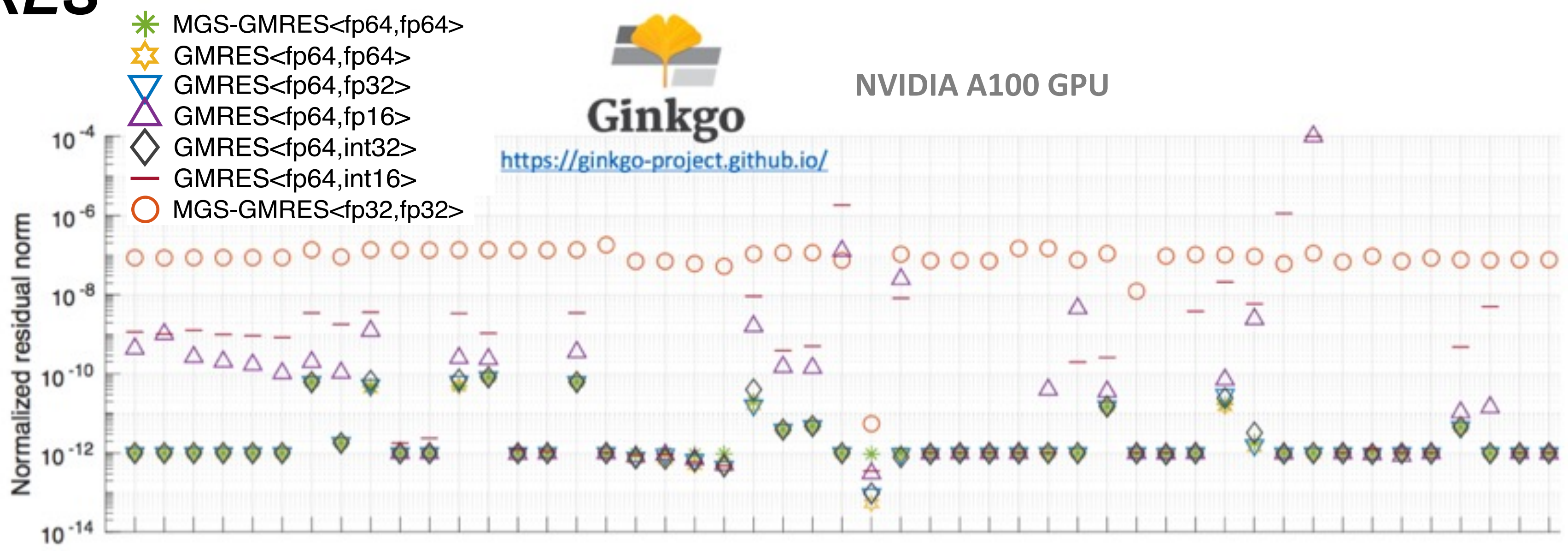
MGS-GMRES<fp64,fp64>
GMRES<fp64,fp64>
GMRES<fp64,fp32>
GMRES<fp64,fp16>
GMRES<fp64,int32>
GMRES<fp64,int16>

arithmetic precision memory precision



Compressed Basis GMRES

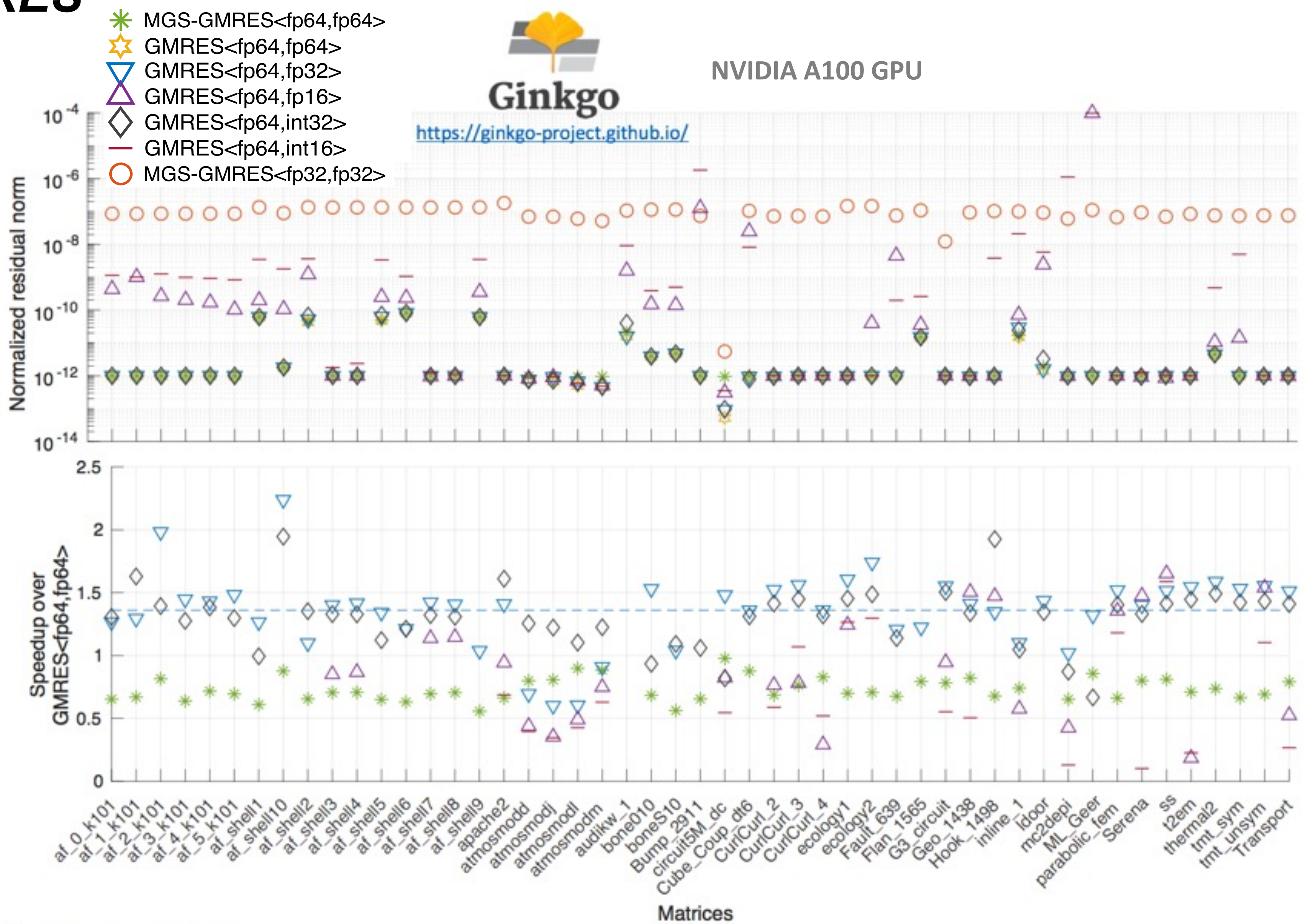
- CB-GMRES using 32-bit storage preserves DP accuracy (SP-GMRES does not)



Compressed Basis GMRES

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

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



Ginkgo
https://ginkgo-project.github.io/

Compressed Basis GMRES on High Performance GPUs

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

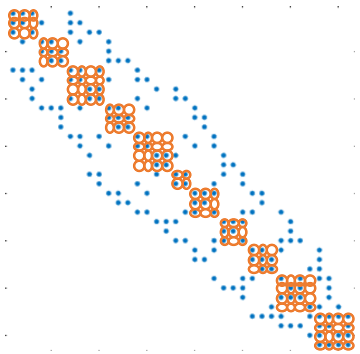
https://arxiv.org/abs/2009.12101

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

- **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.



Rethinking Algorithms II: Approximate Linear Operators

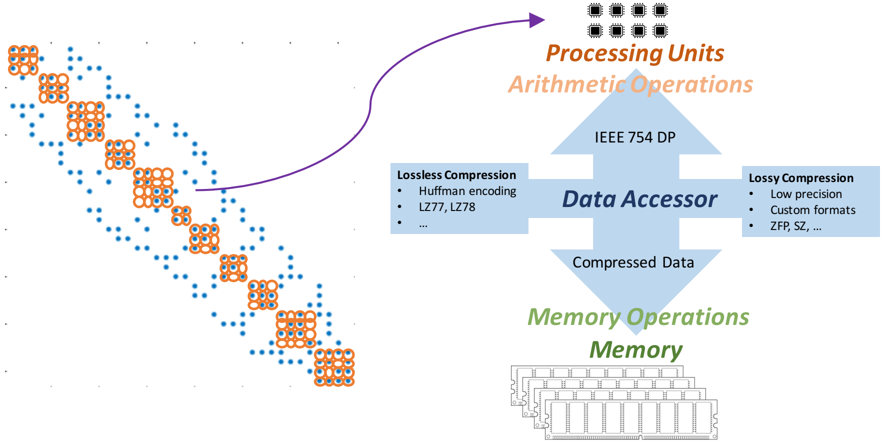
- **Preconditioning iterative solvers.**
  - Idea: Approximate inverse of system matrix to make the system "easier to solve": $P^{-1} \approx A^{-1}$

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

- **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.

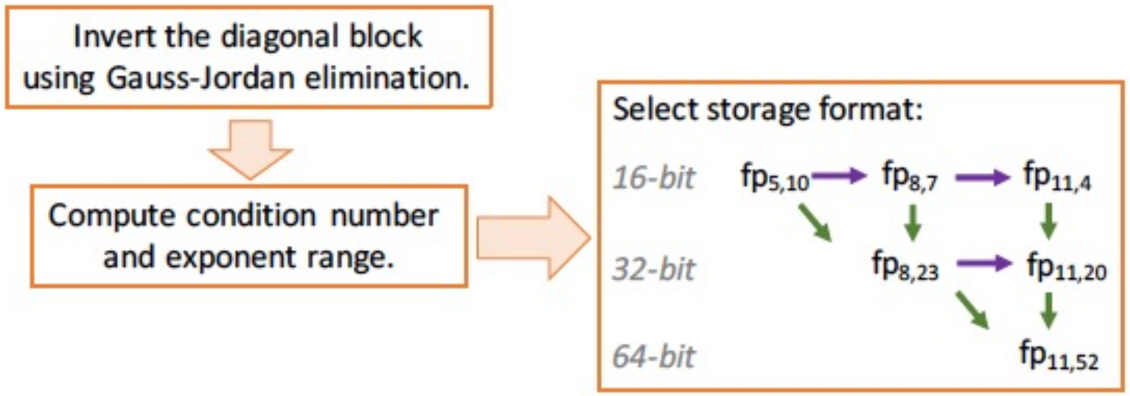
- *Why should we store the preconditioner matrix P^{-1} in full (high) precision?*

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



Mixed Precision Preconditioning

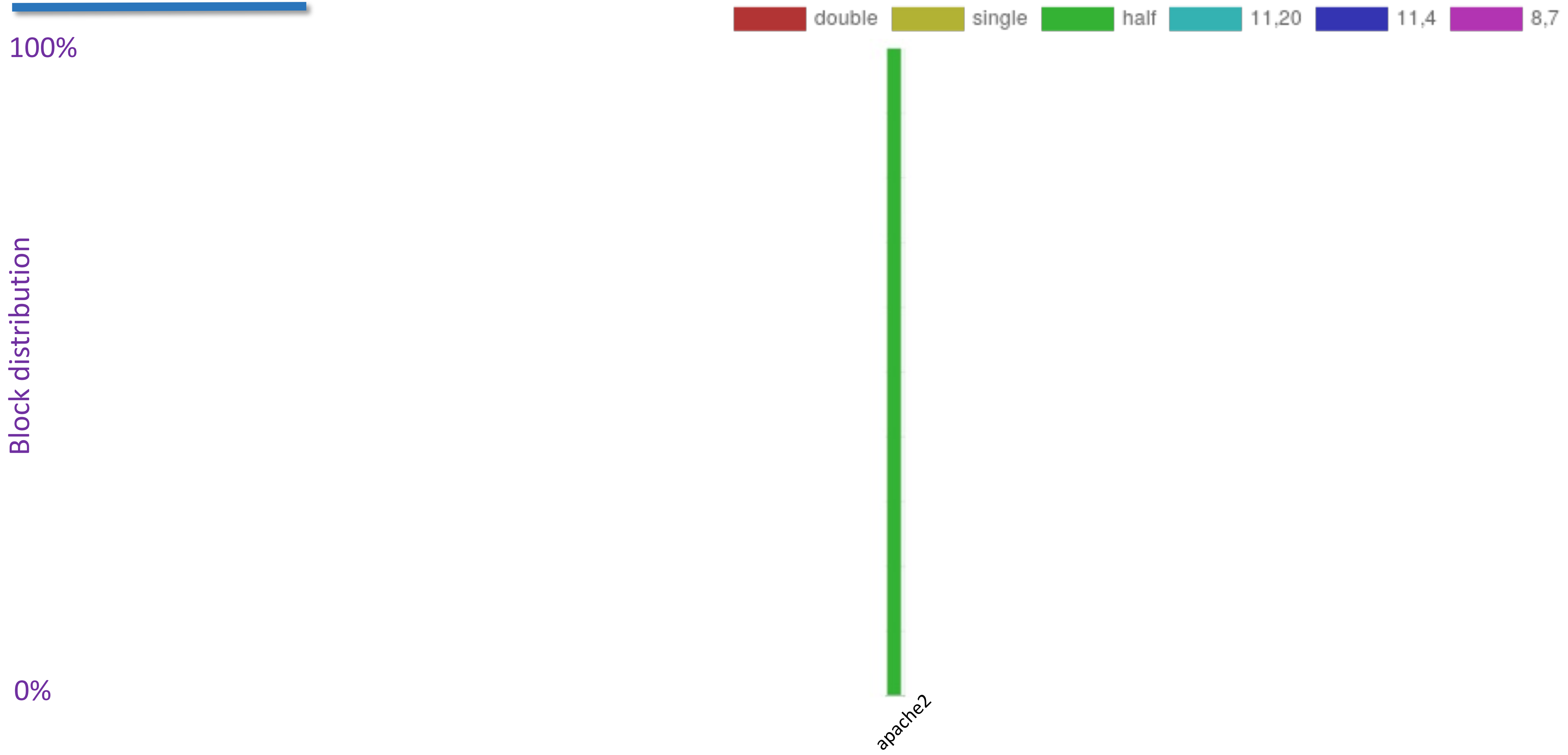
- Choose how much accuracy of the preconditioner should be preserved in the selection of the storage format.
- All computations use double precision, but store blocks in lower precision.



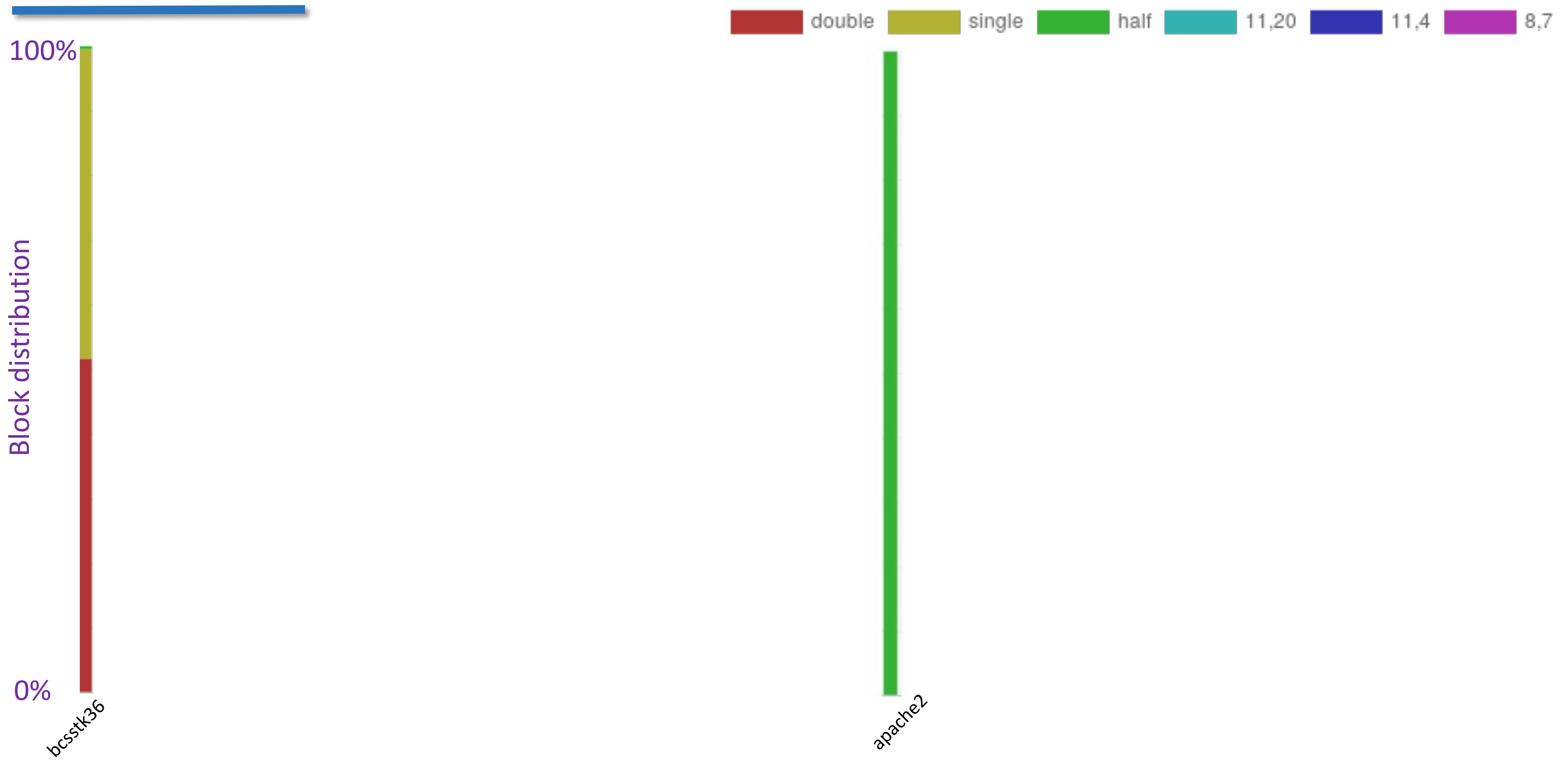
+ **Regularity preserved;**
+ Flexibility in the accuracy;
+ ”Not a low precision preconditioner”
    + Preconditioner is a constant operator;
    + No flexible Krylov solver needed ;

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

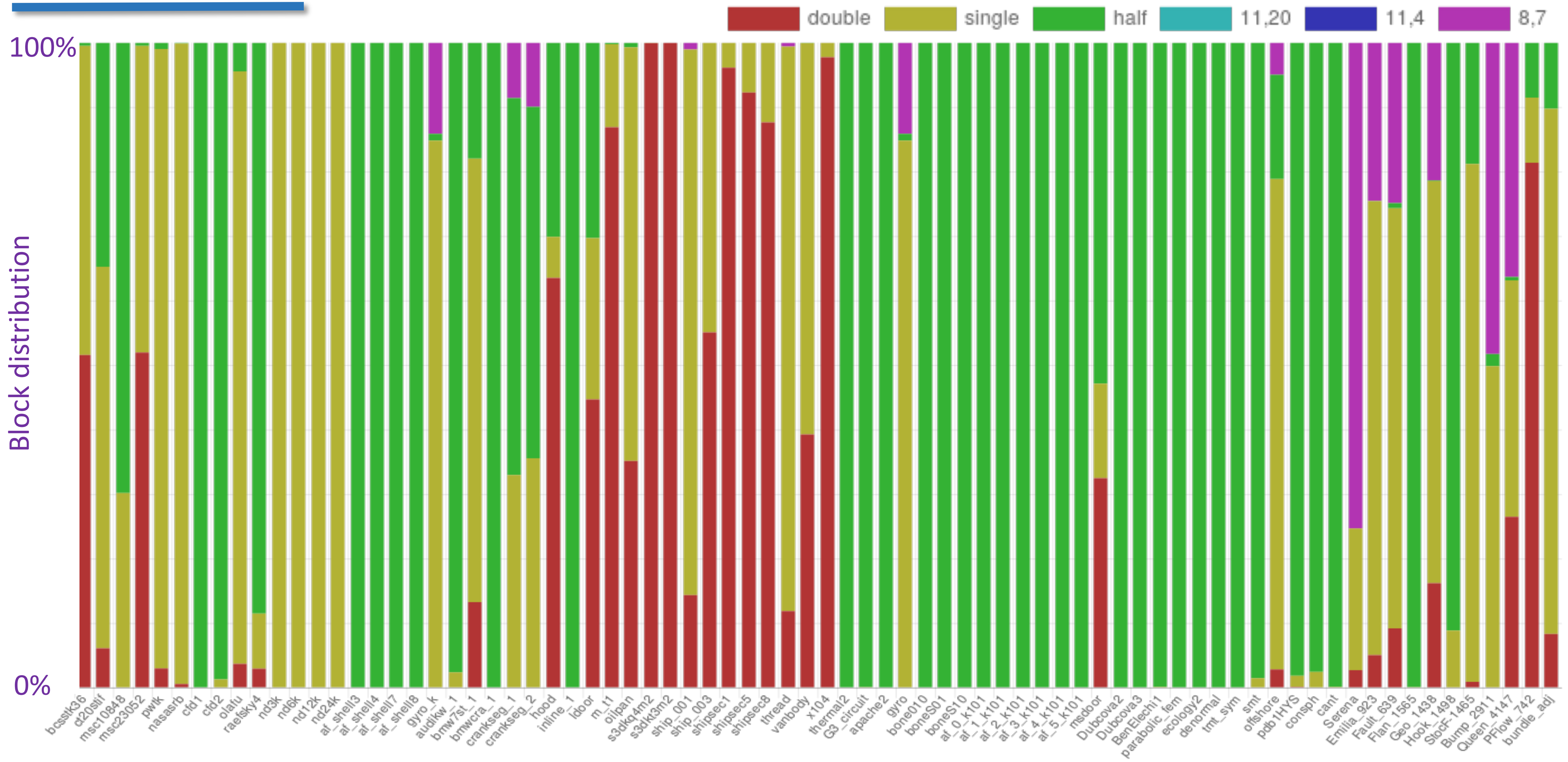
Mixed Precision Preconditioning



Mixed Precision Preconditioning



Mixed Precision Preconditioning



Mixed Precision Preconditioning

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

Double Precision CG + Double Precision Preconditioner

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

Accuracy improvement ~10^9

Single Precision CG + Single Precision Preconditioner

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

No improvement

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

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**

- *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 cond(A) ≈ 10^7 *(apache2 from SuiteSparse)* **NVIDIA A100 GPU**

Double Precision CG + Double Precision Preconditioner

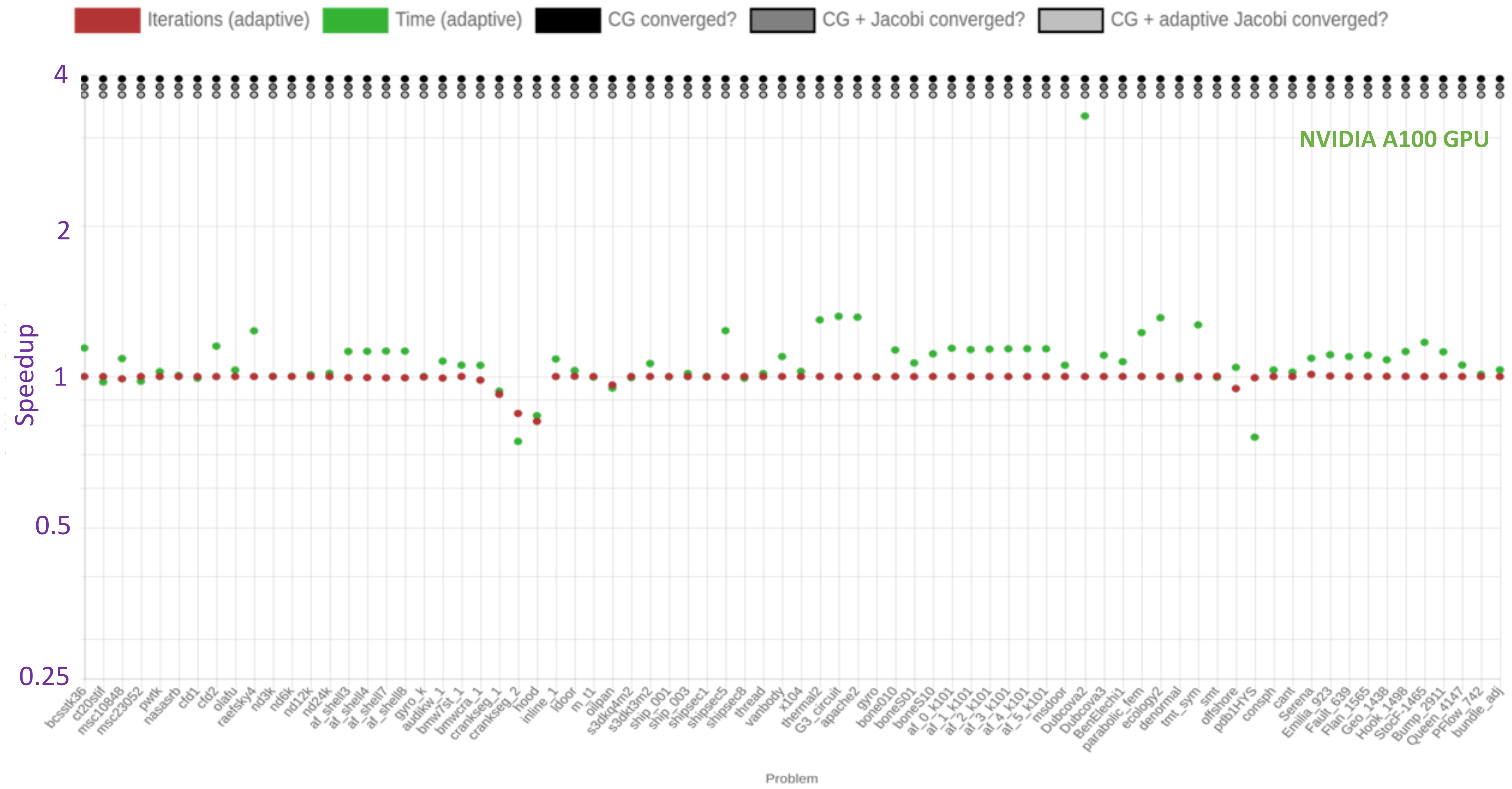
```
Initial residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
1390.67
Final residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
3.97985e-06
CG iteration count:     4797
CG execution time [ms]: 2971.18
```

Double Precision CG + **Mixed Precision Preconditioner**

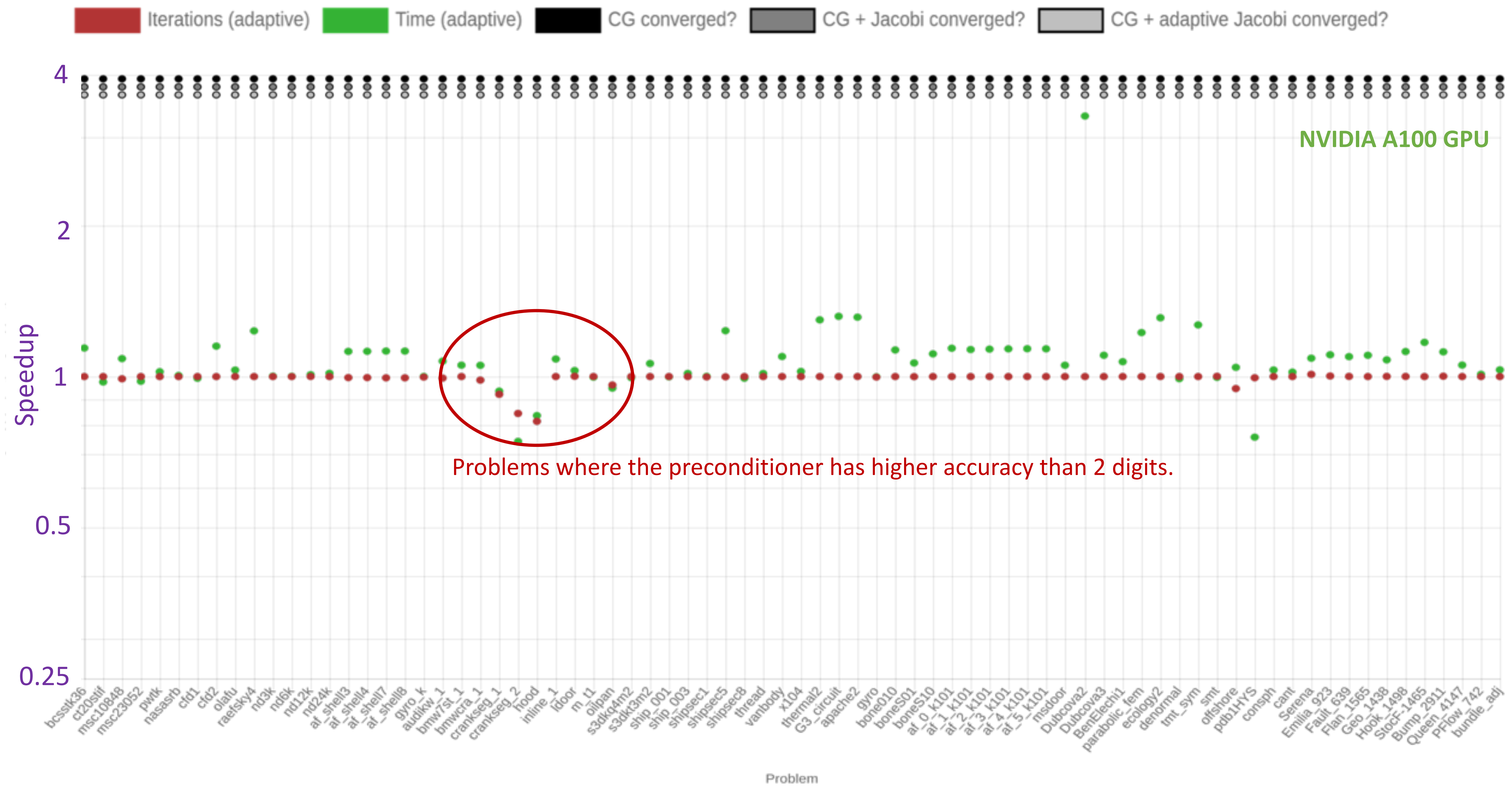
```
Initial residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
1390.67
Final residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
3.98574e-06
CG iteration count:     4794
CG execution time [ms]: 2568.1
```

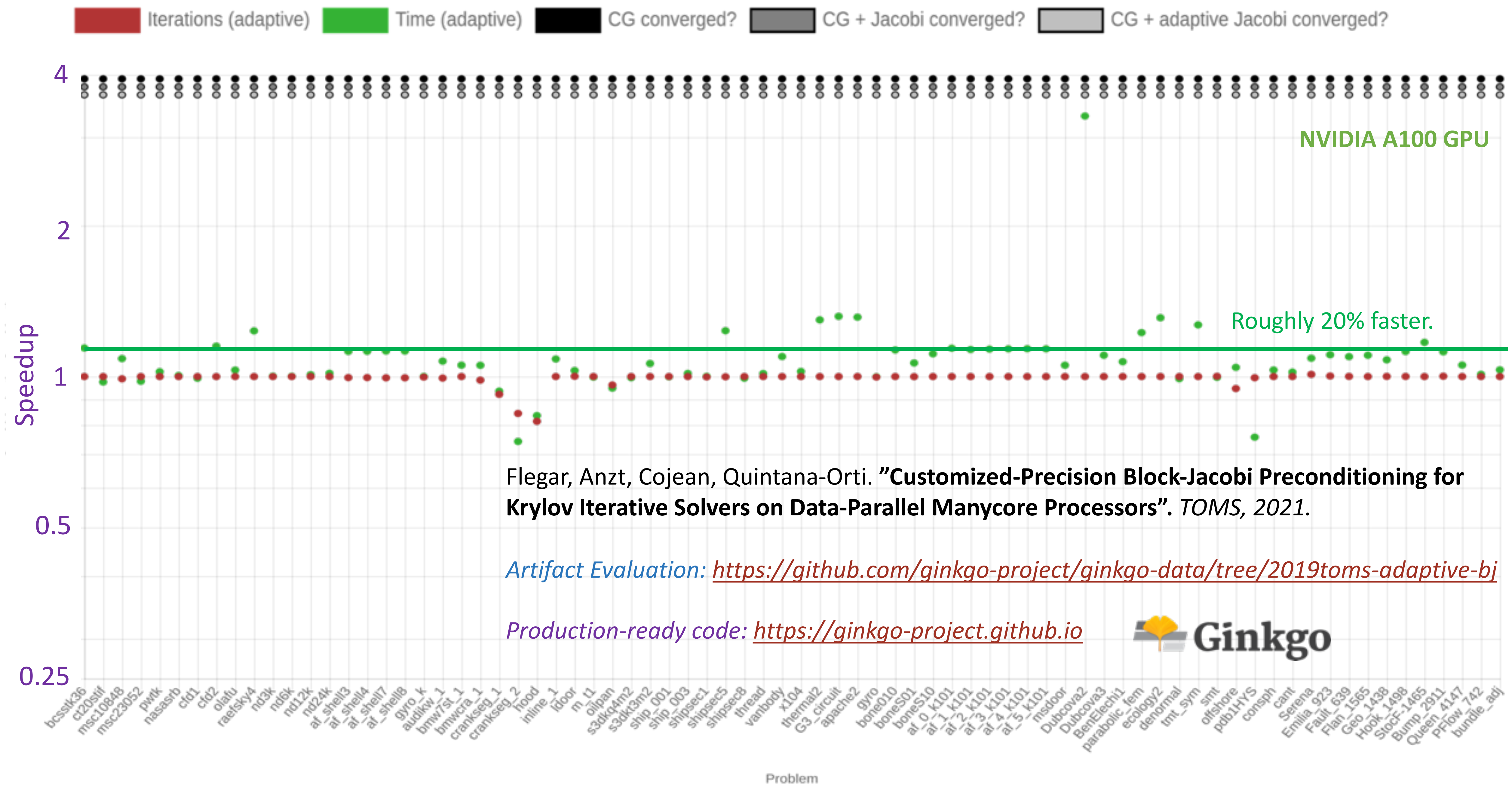
16% runtime improvement

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



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


Iterations (adaptive)
Time (adaptive)
CG converged?
CG + Jacobi converged?
CG + adaptive Jacobi converged?
NVIDIA A100 GPU
Speedup
4
2
1
0.5
0.25
Problems where the preconditioner has higher accuracy than 2 digits.
Problem



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

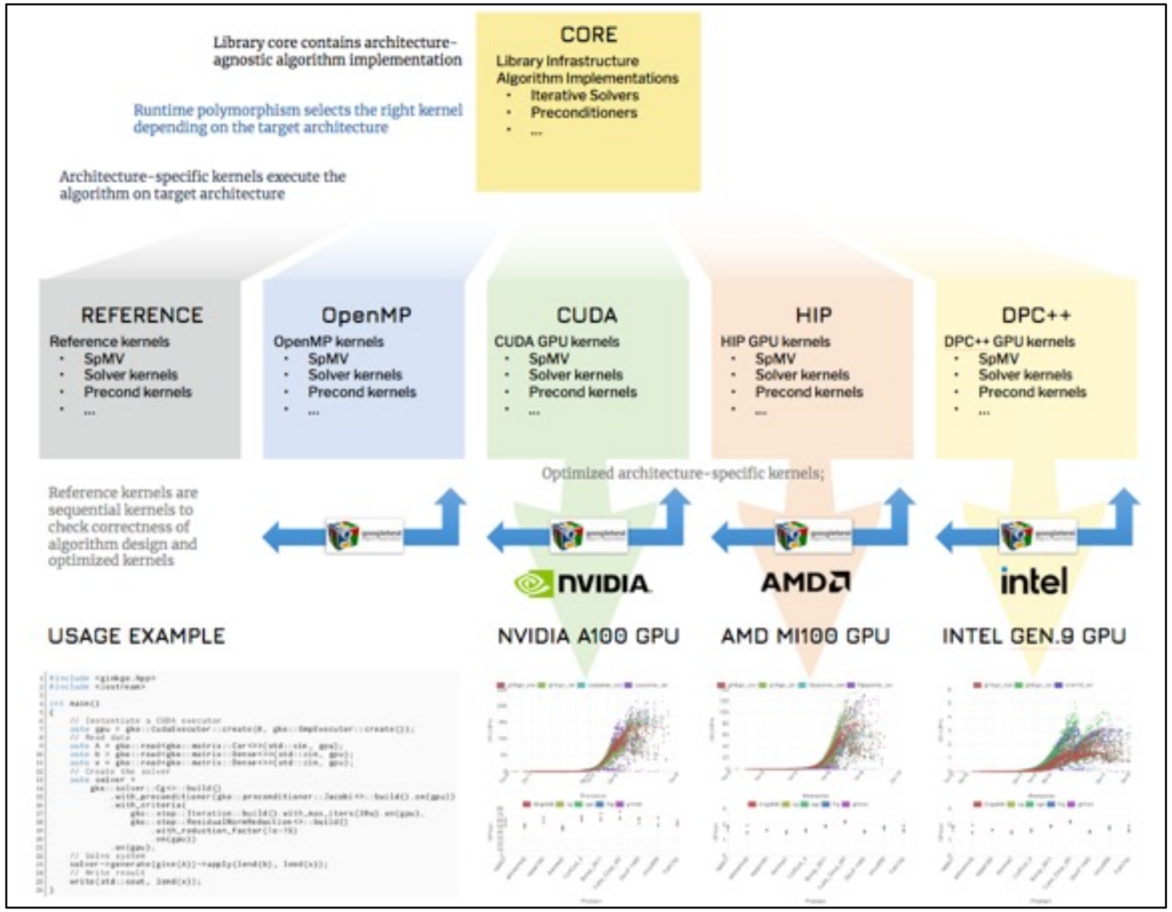
Artifact Evaluation: *https://github.com/ginkgo-project/ginkgo-data/tree/2019toms-adaptive-bj*

Production-ready code: *https://ginkgo-project.github.io*

Ginkgo

Ginkgo

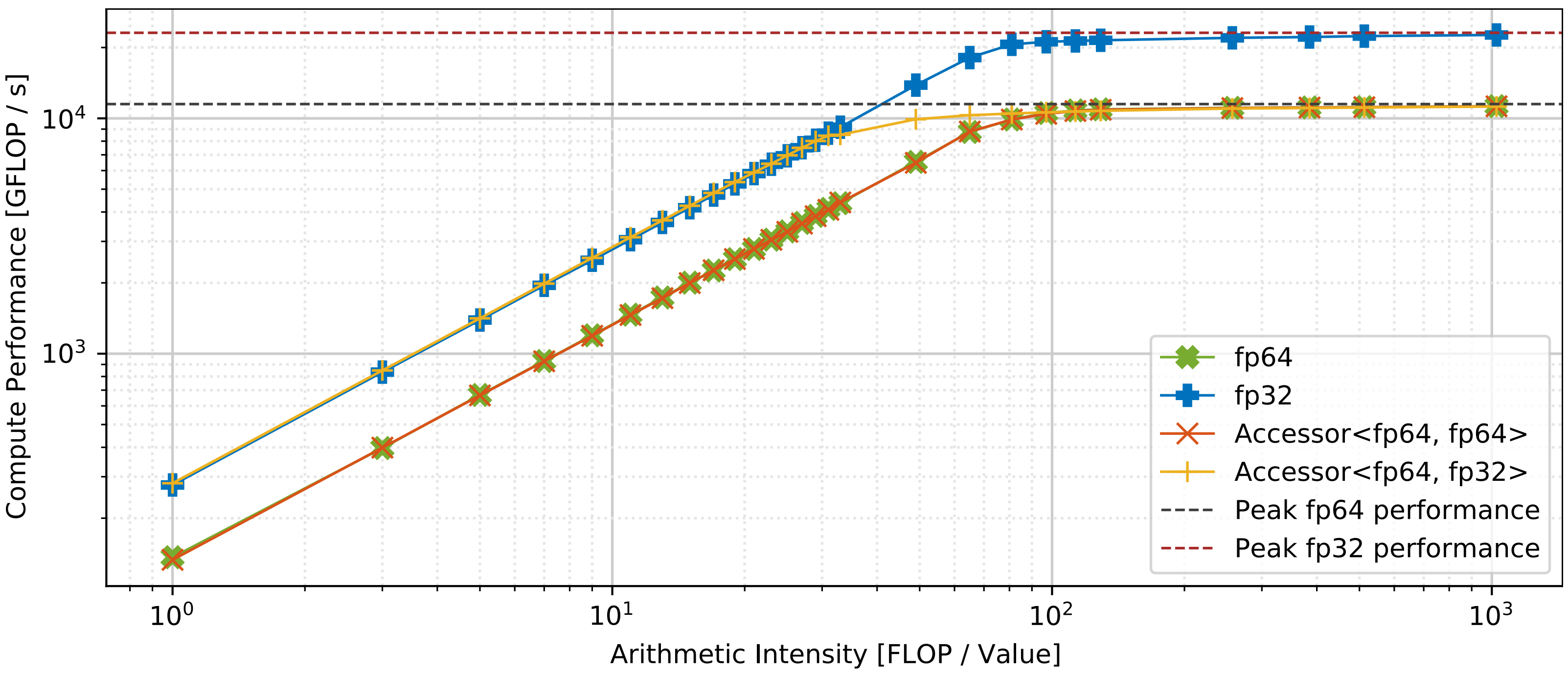
- **High performance sparse linear algebra**
  - Linear algebra building blocks: SpMV, SpMM, SpGEAM,...;
  - Linear solvers: BiCG, BiCGSTAB, CG, CGS, FCG, GMRES, IDR;
  - Advanced preconditioning techniques: ParILU, ParILUT, SAI;
  - Batched iterative solvers;
- **Exascale early systems GPU-readiness**
  - 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;



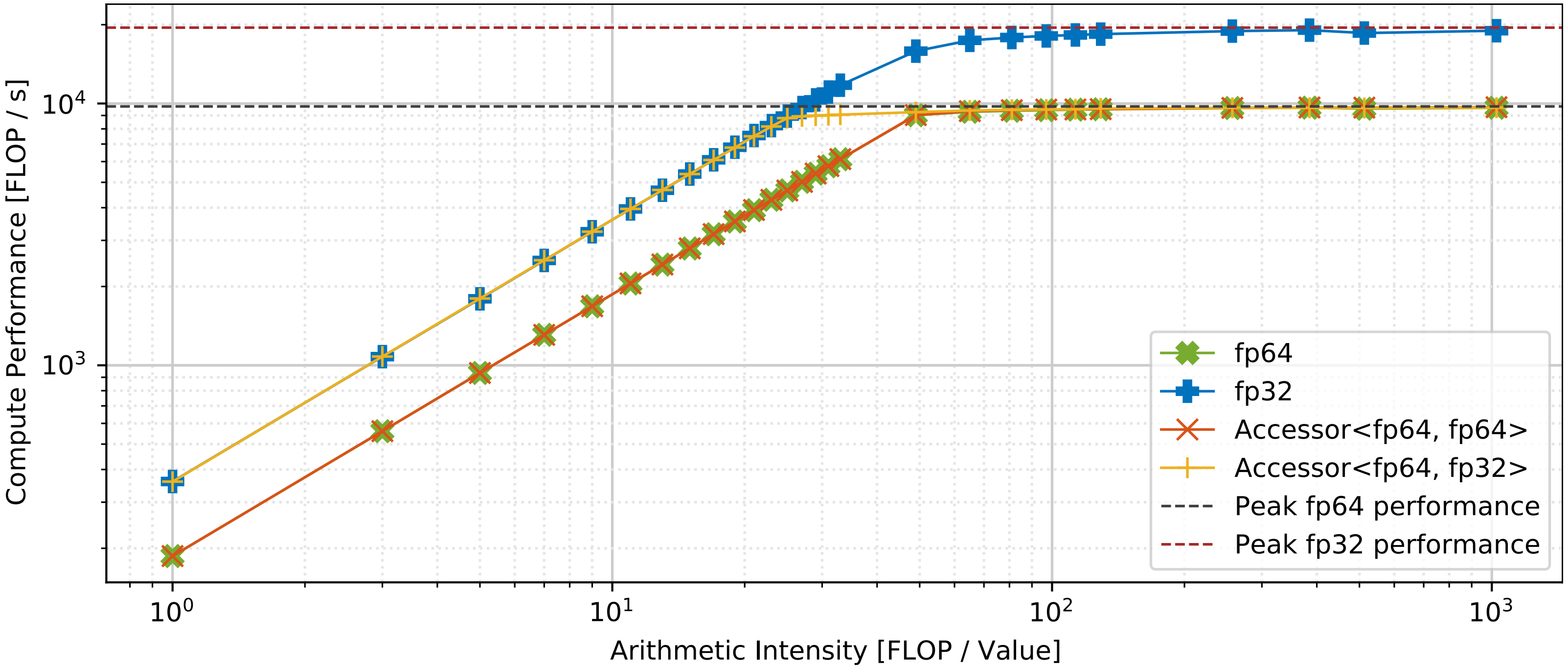
https://ginkgo-project.github.io/

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.

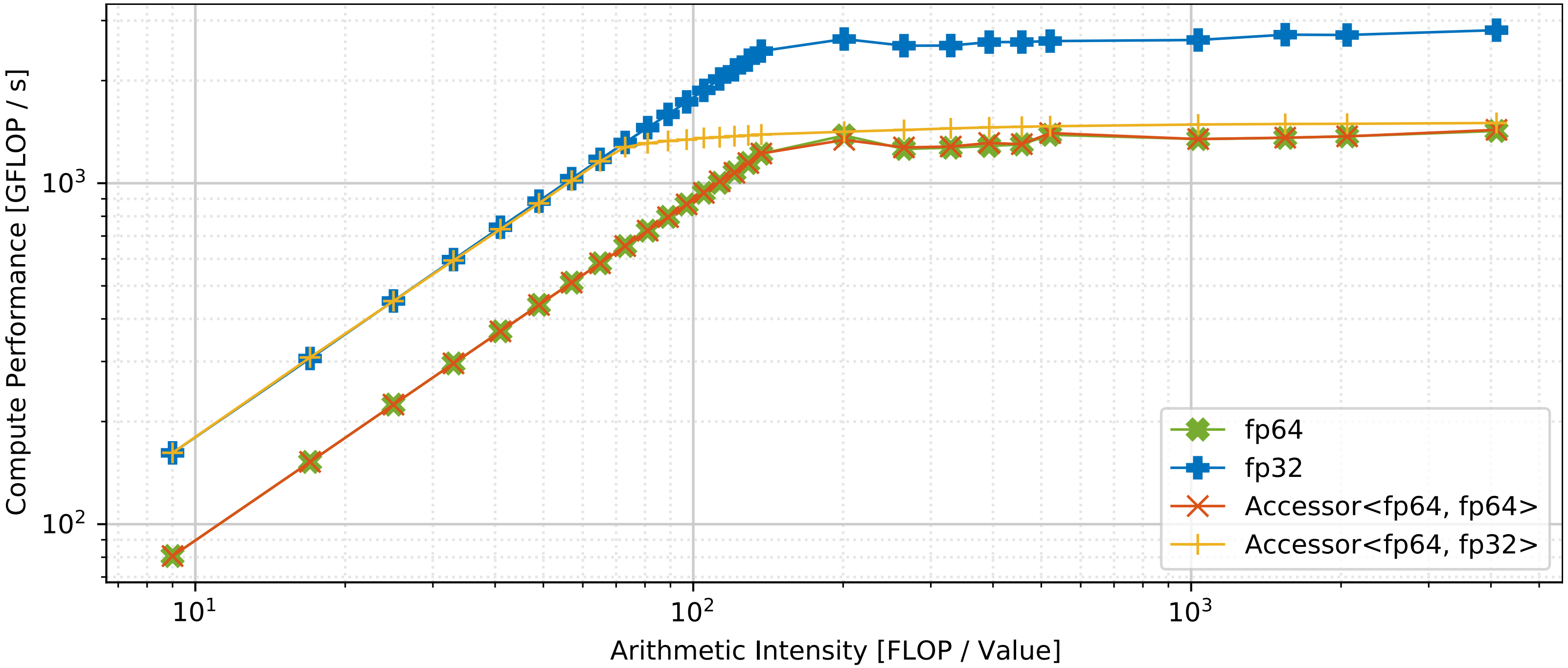
Accessor for AMD MI100 GPU



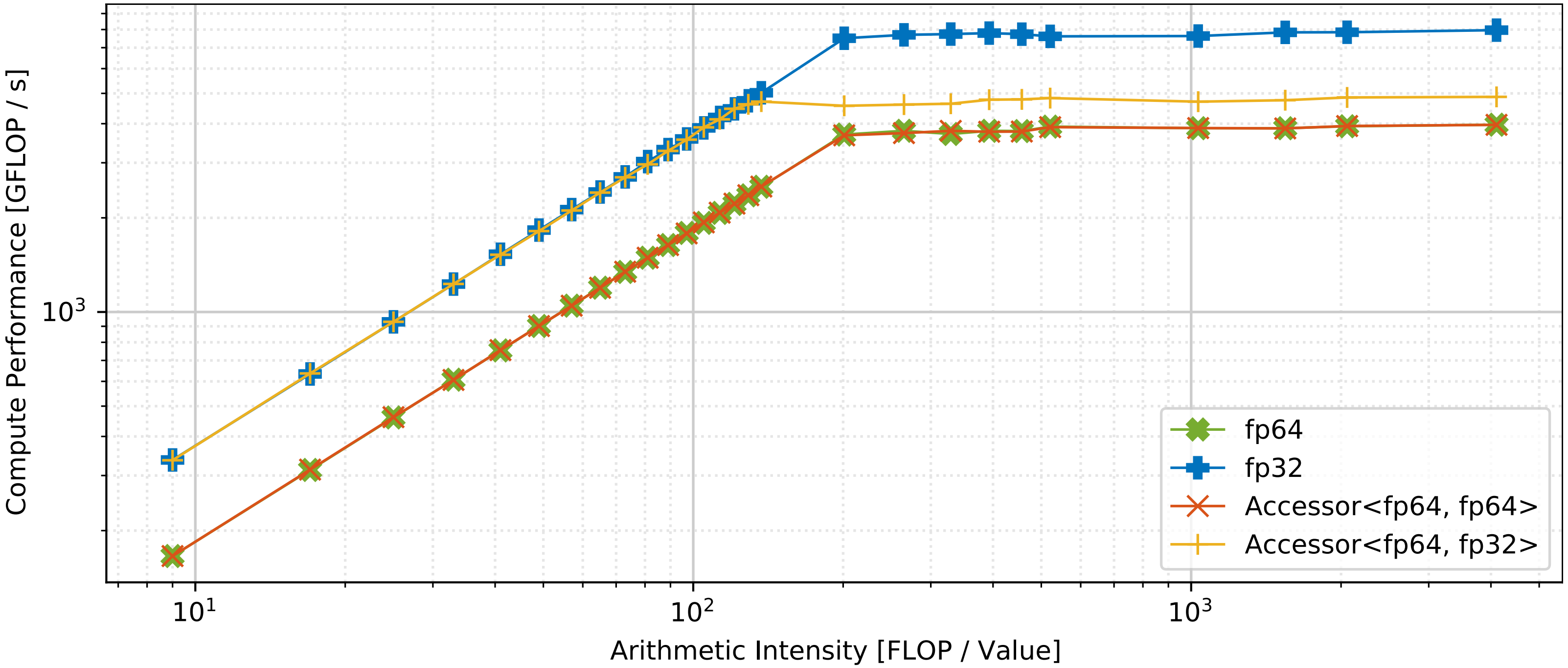
Accessor for NVIDIA A100 GPU



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;

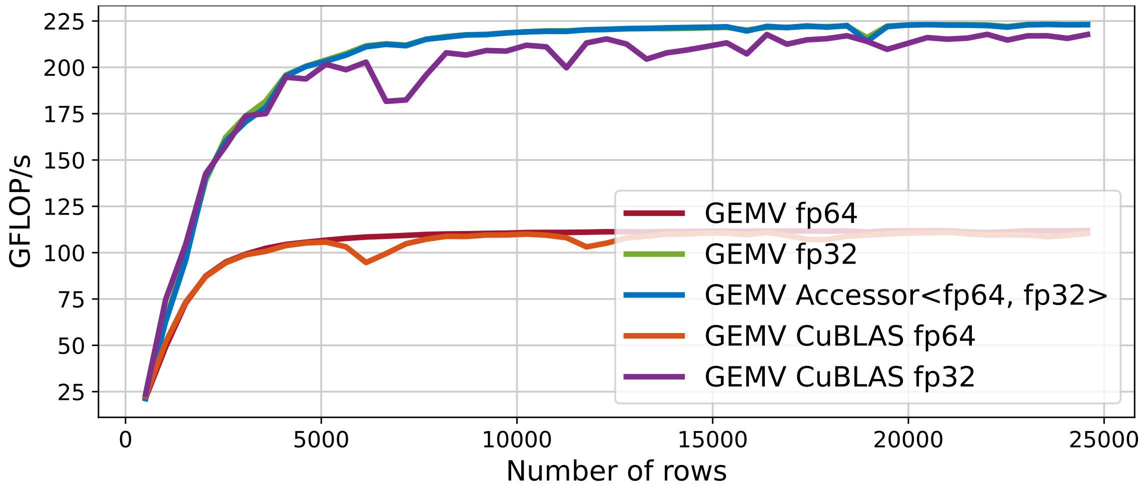
Usage

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

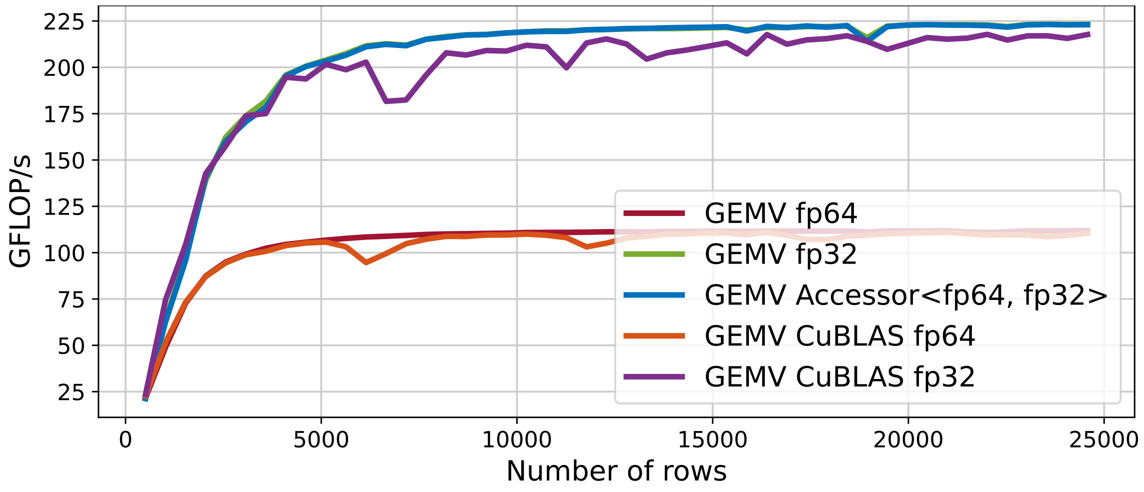
NVIDIA V100 GPU (Summit)

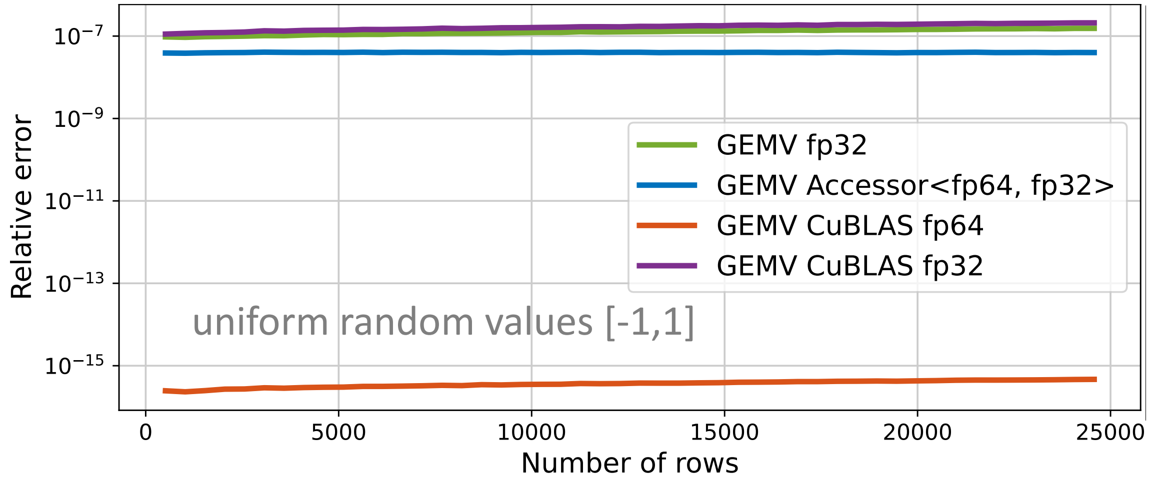


Accessor-BLAS: Replacing LP BLAS to improve accuracy

GEMV

NVIDIA V100 GPU (Summit)

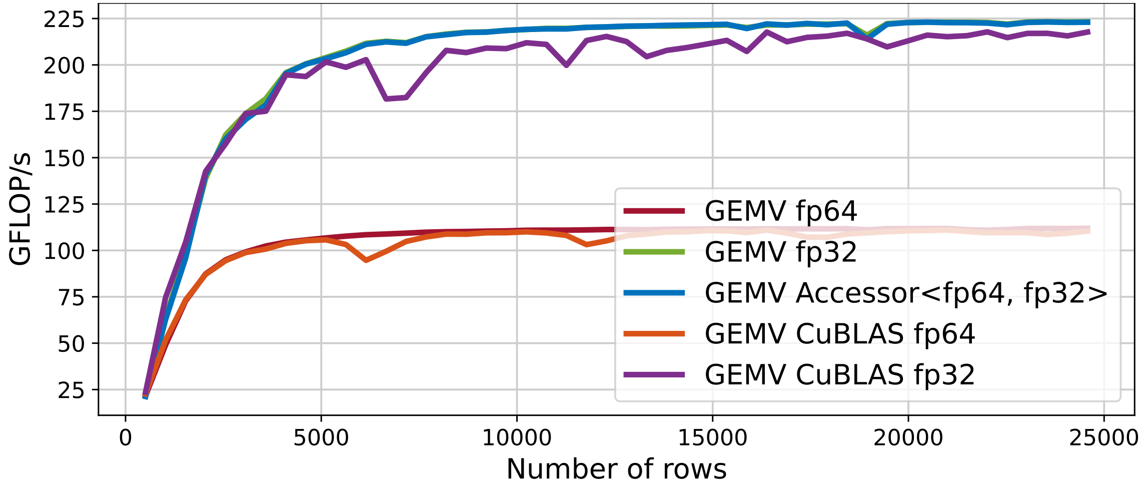


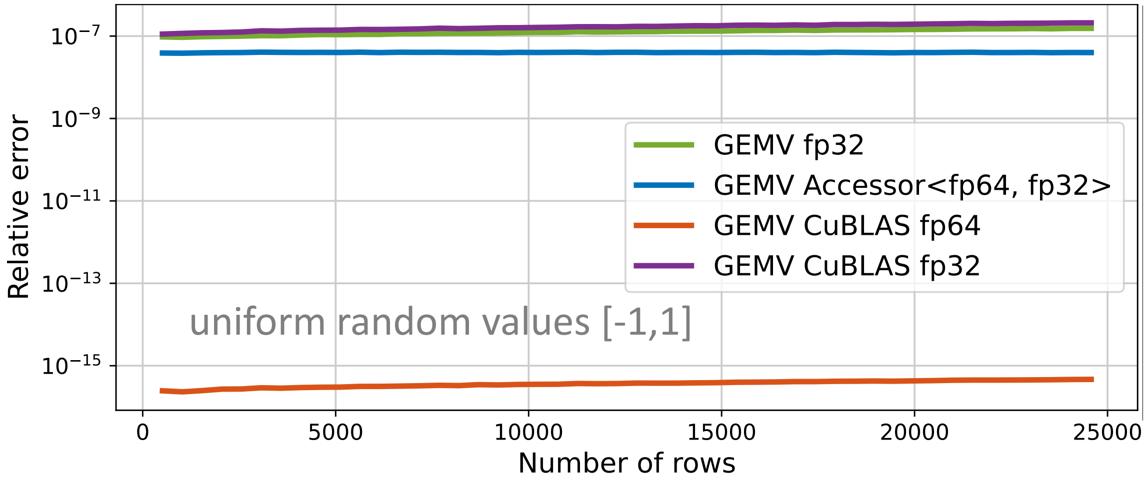


Accessor-BLAS: Replacing LP BLAS to improve accuracy

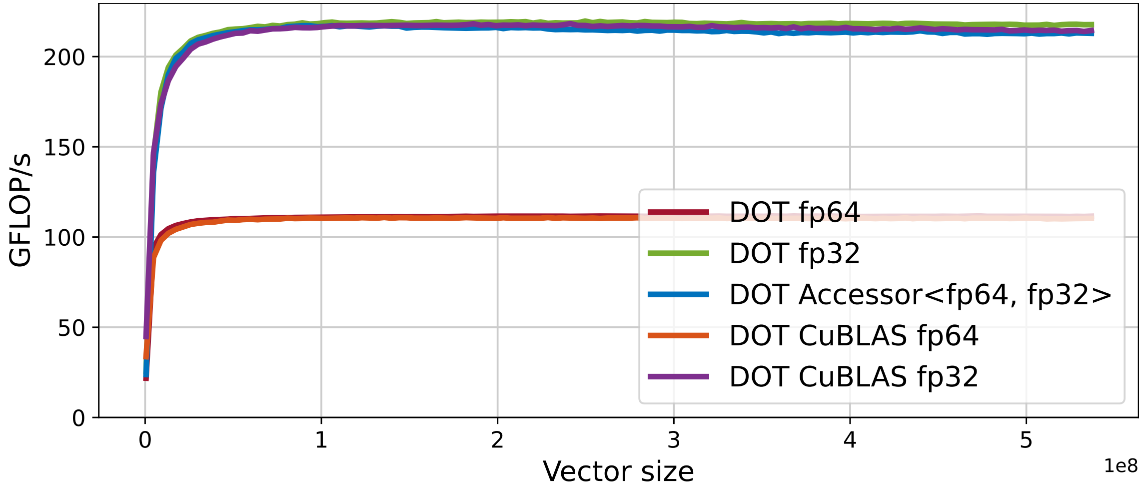
GEMV

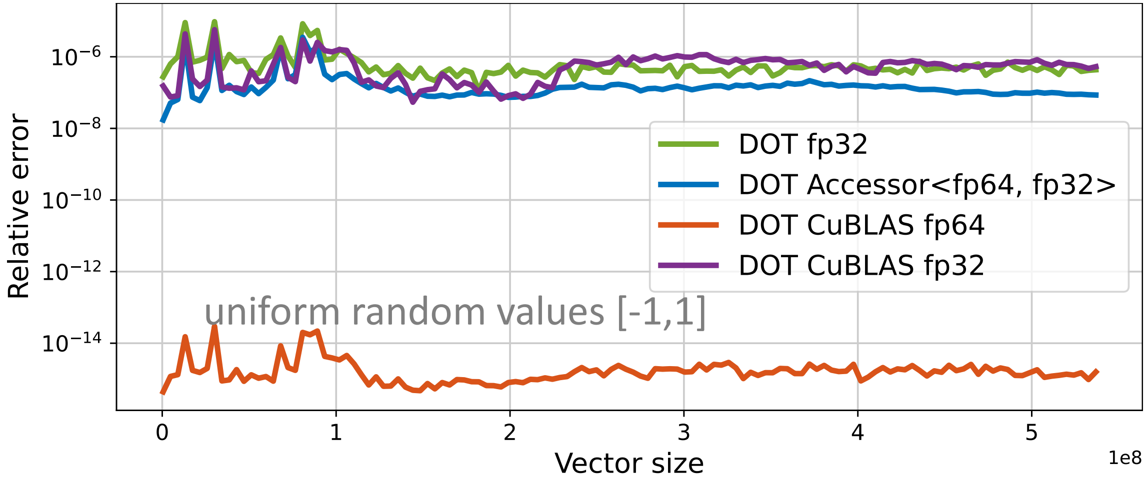
NVIDIA V100 GPU (Summit)





DOT

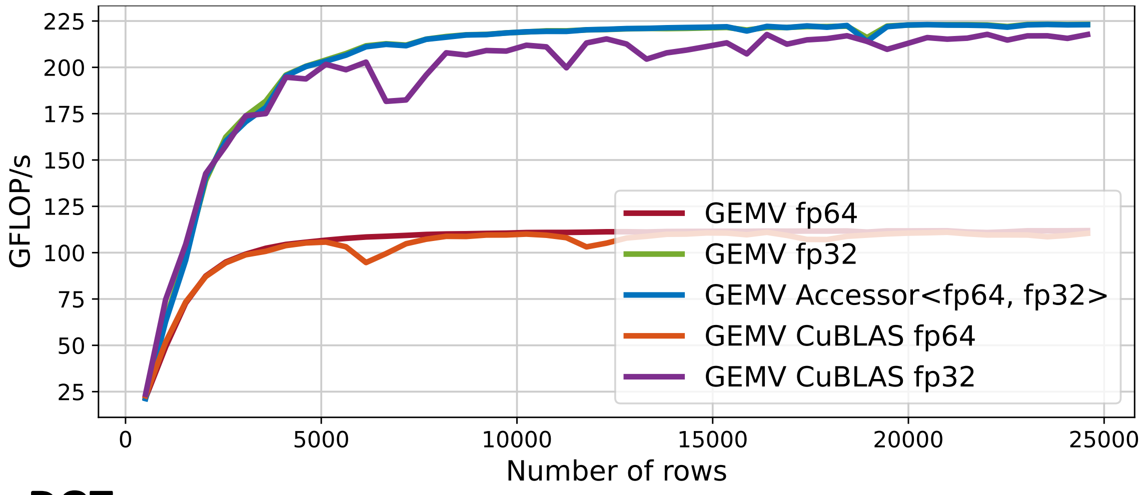


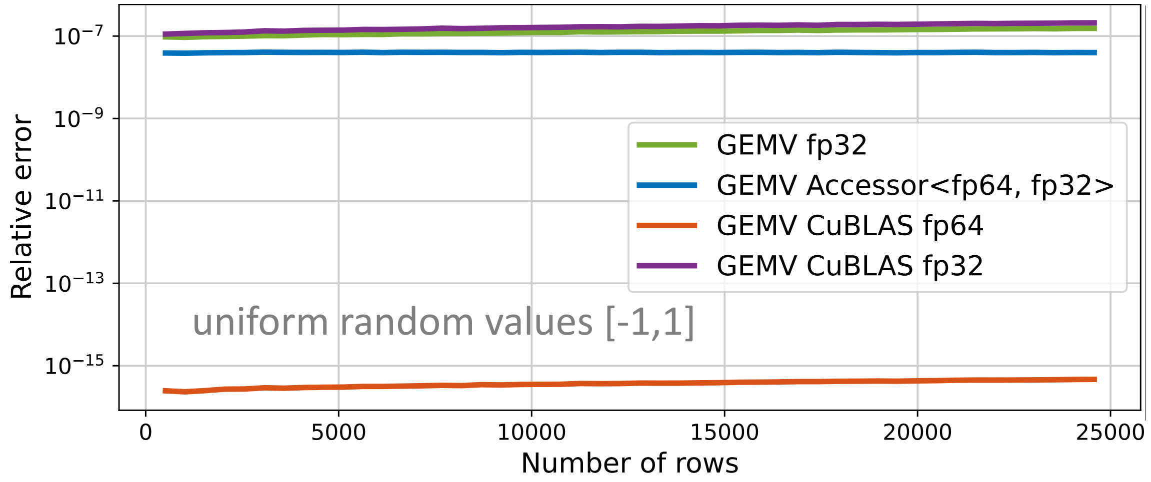


Accessor-BLAS: Replacing LP BLAS to improve accuracy

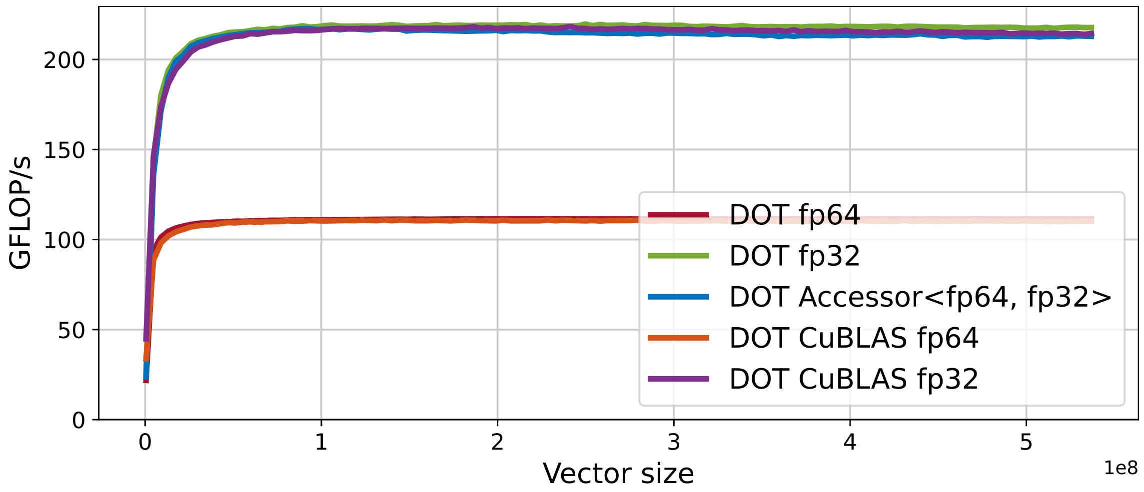
GEMV

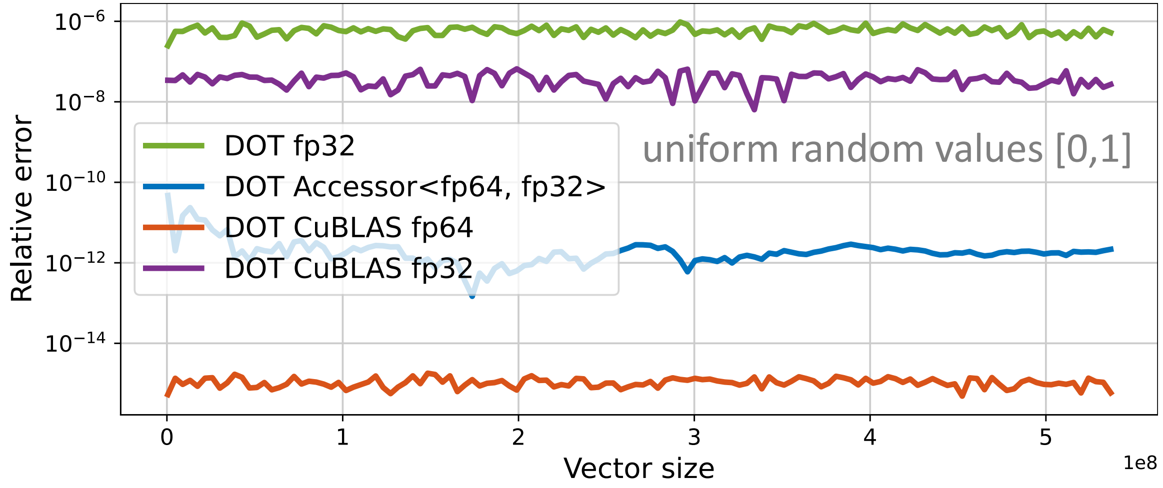
NVIDIA V100 GPU (Summit)





DOT





Mixed Precision Sparse Approximate Inverse

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

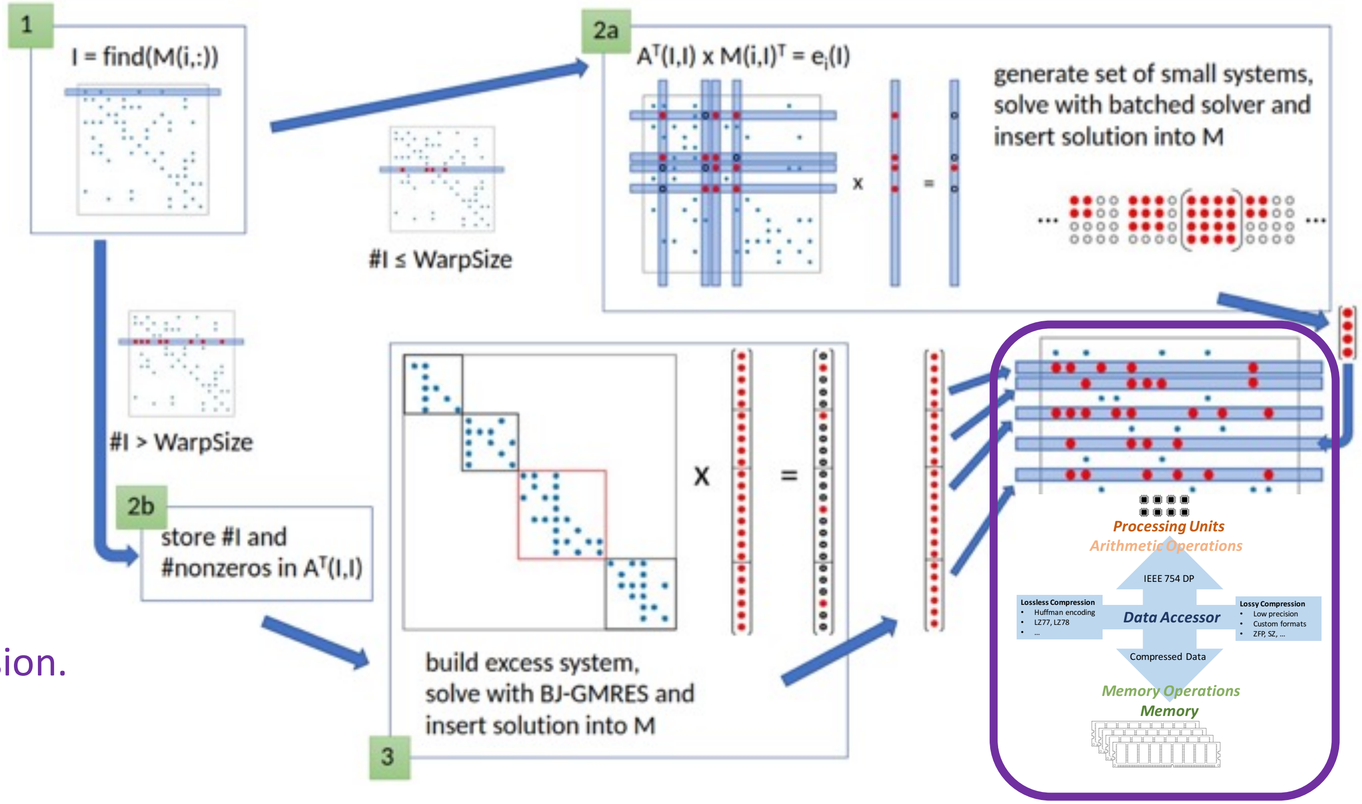
    and solve $Ax = b \quad \Leftrightarrow \quad P^{-1}Ax = P^{-1}b \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}\,.$

- **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;

- 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 submitted.

Mixed Precision Sparse Approximate Inverse

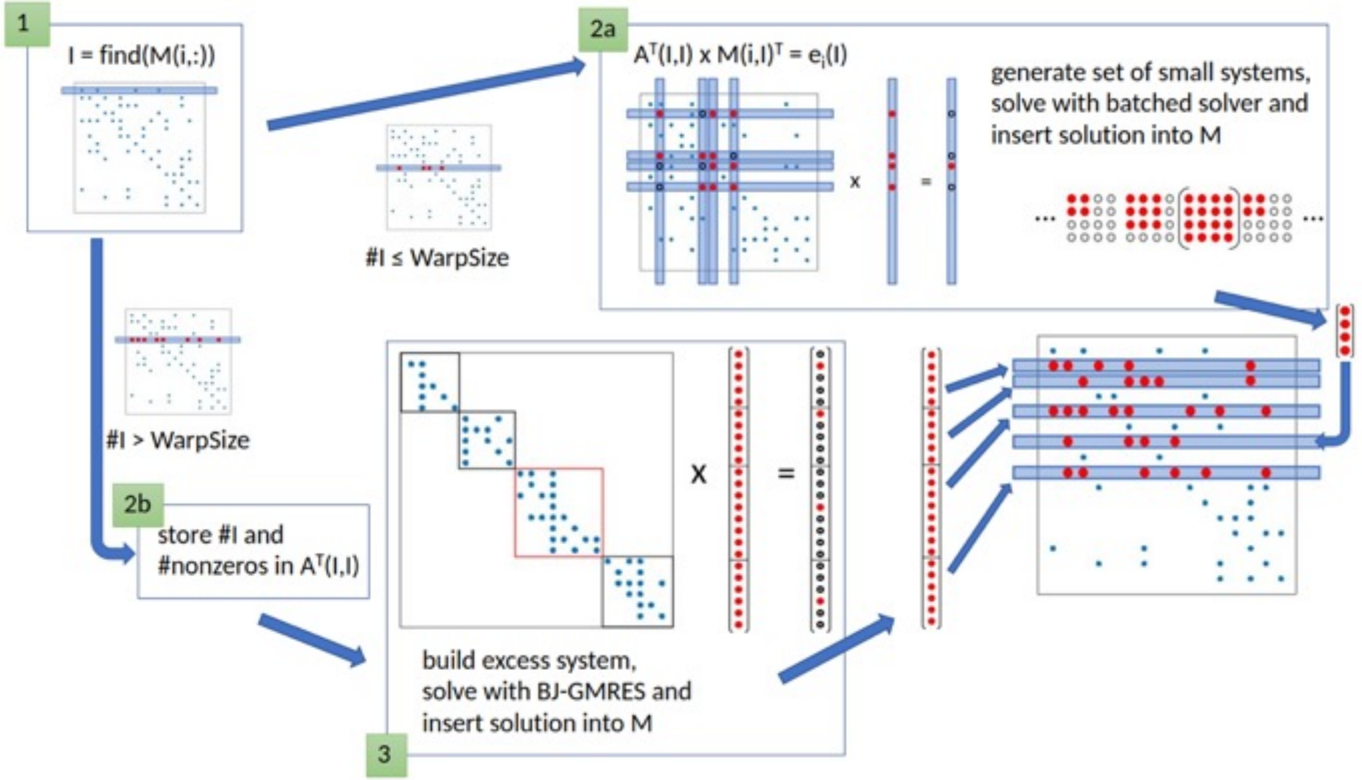
- **Preconditioning iterative solvers.**
  - Idea: Approximate inverse of system matrix to make the system "easier to solve": $P^{-1} \approx A^{-1}$

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

- **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": $P^{-1} \approx A^{-1}$

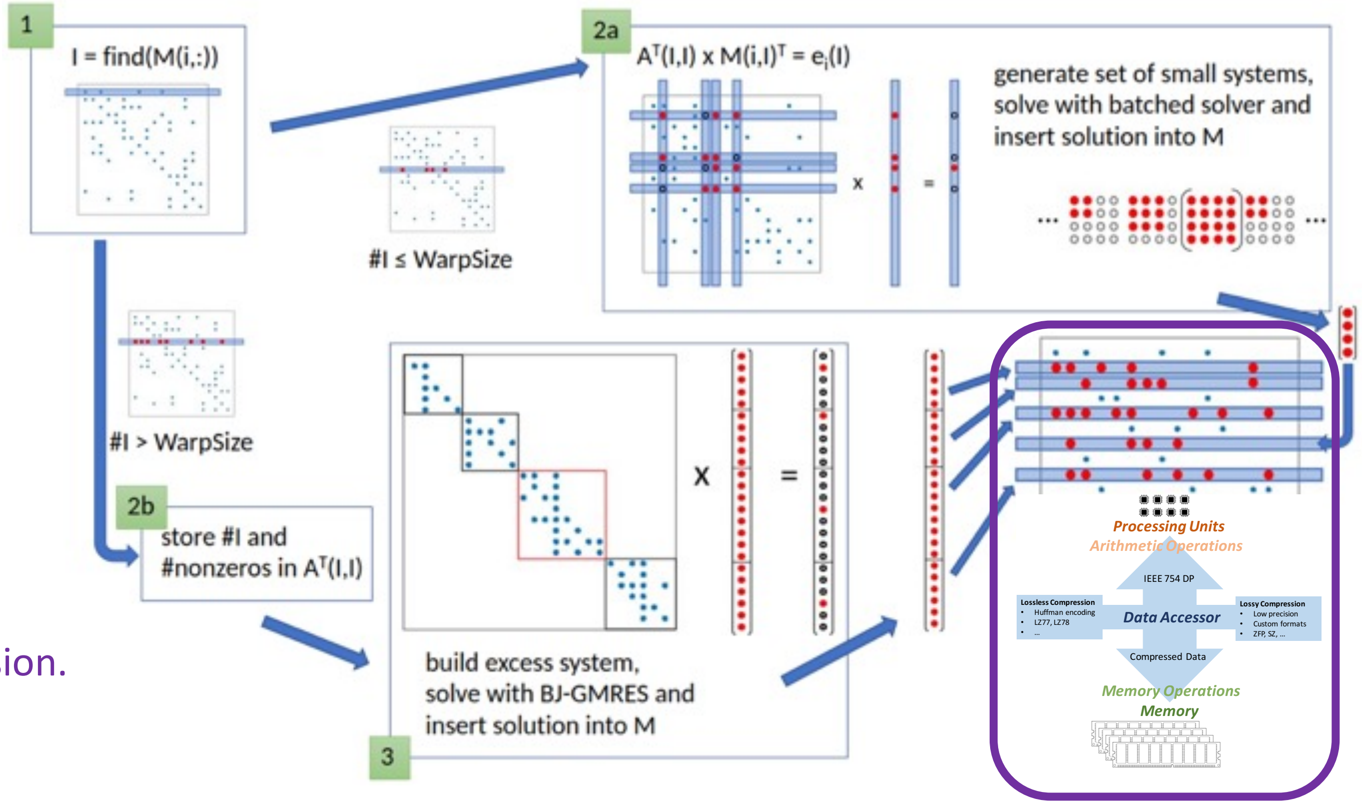
    and solve $Ax = b \quad \Leftrightarrow \quad P^{-1}Ax = P^{-1}b \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}\,.$

- **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)

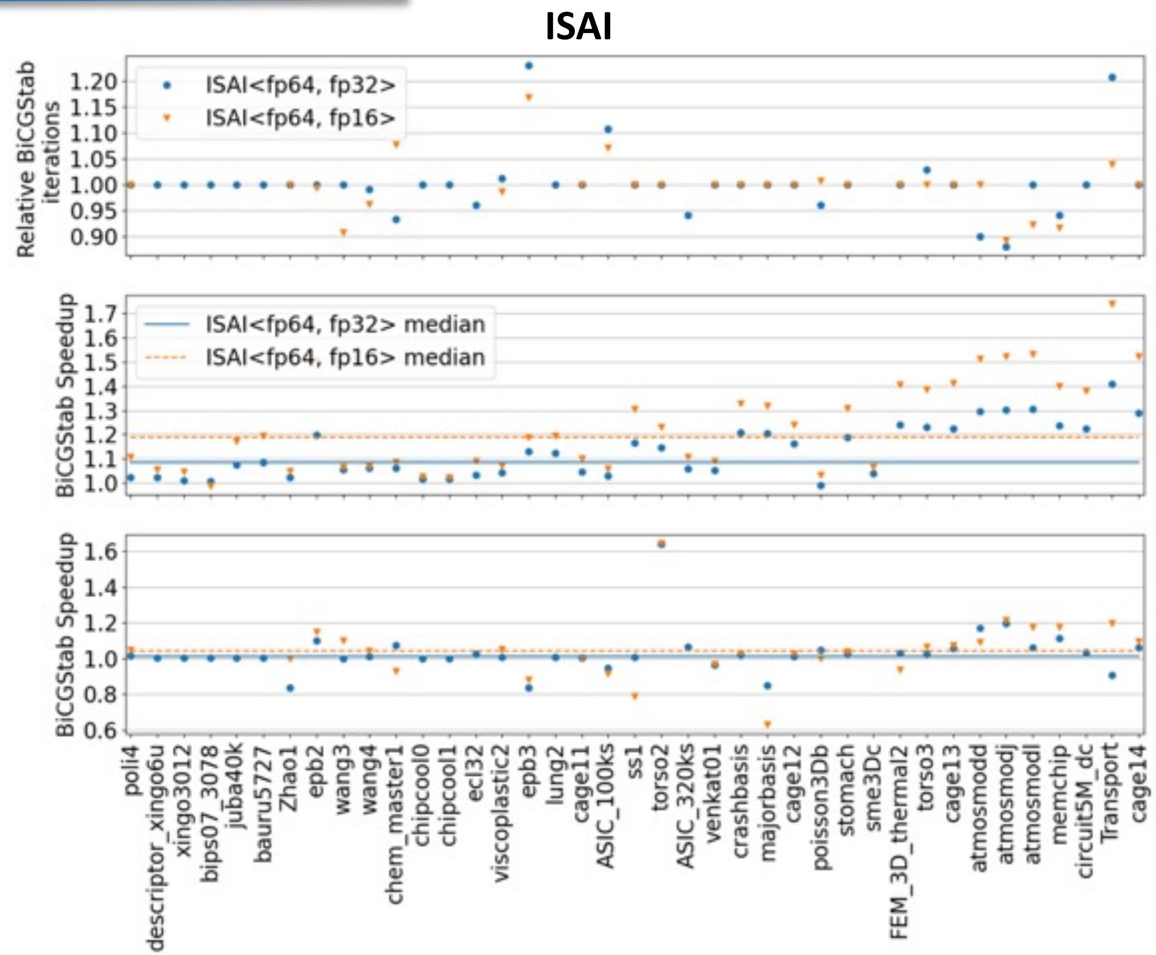
    stores inverse approximation in factorized form;

- 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 submitted.

Mixed Precision Sparse Approximate Inverse

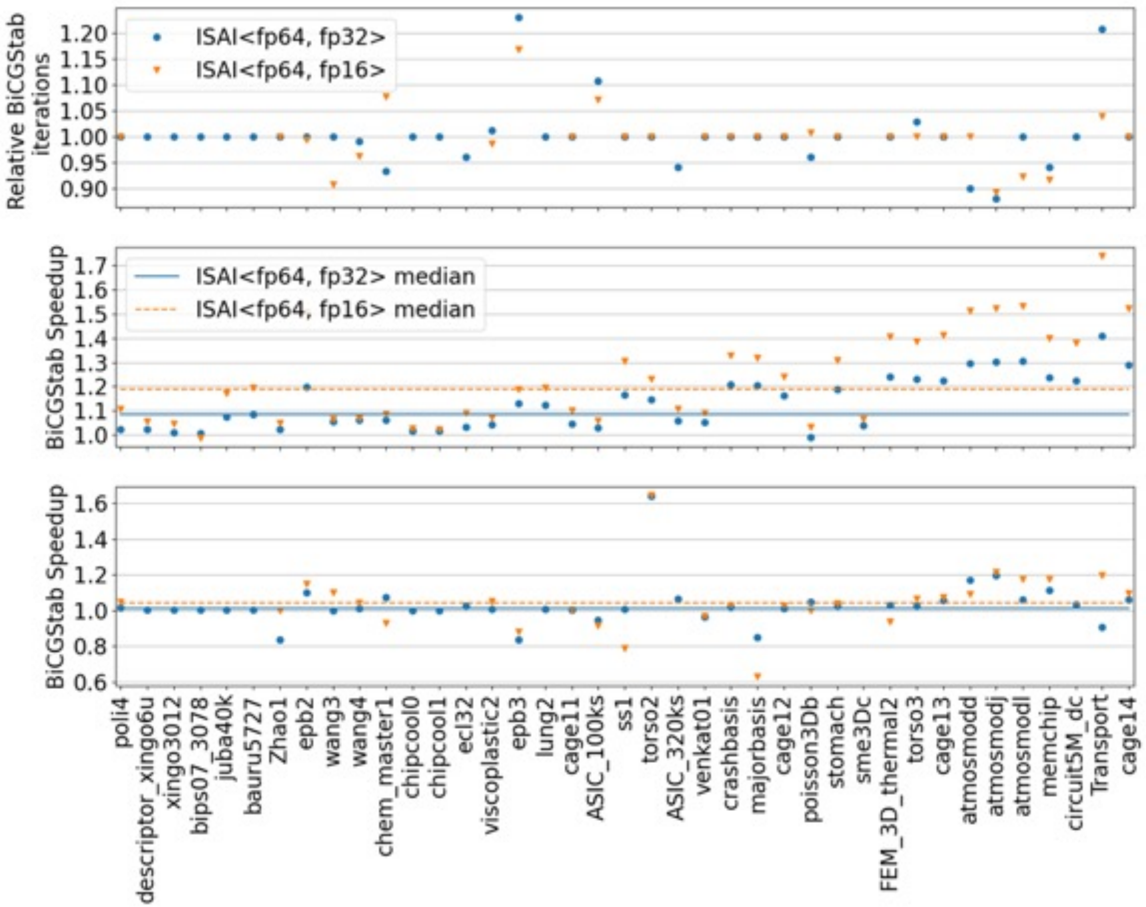


- ~10% speedup for ISAI<fp64,fp32>
- ~20% speedup for ISAI<fp64,fp16> - unstable!

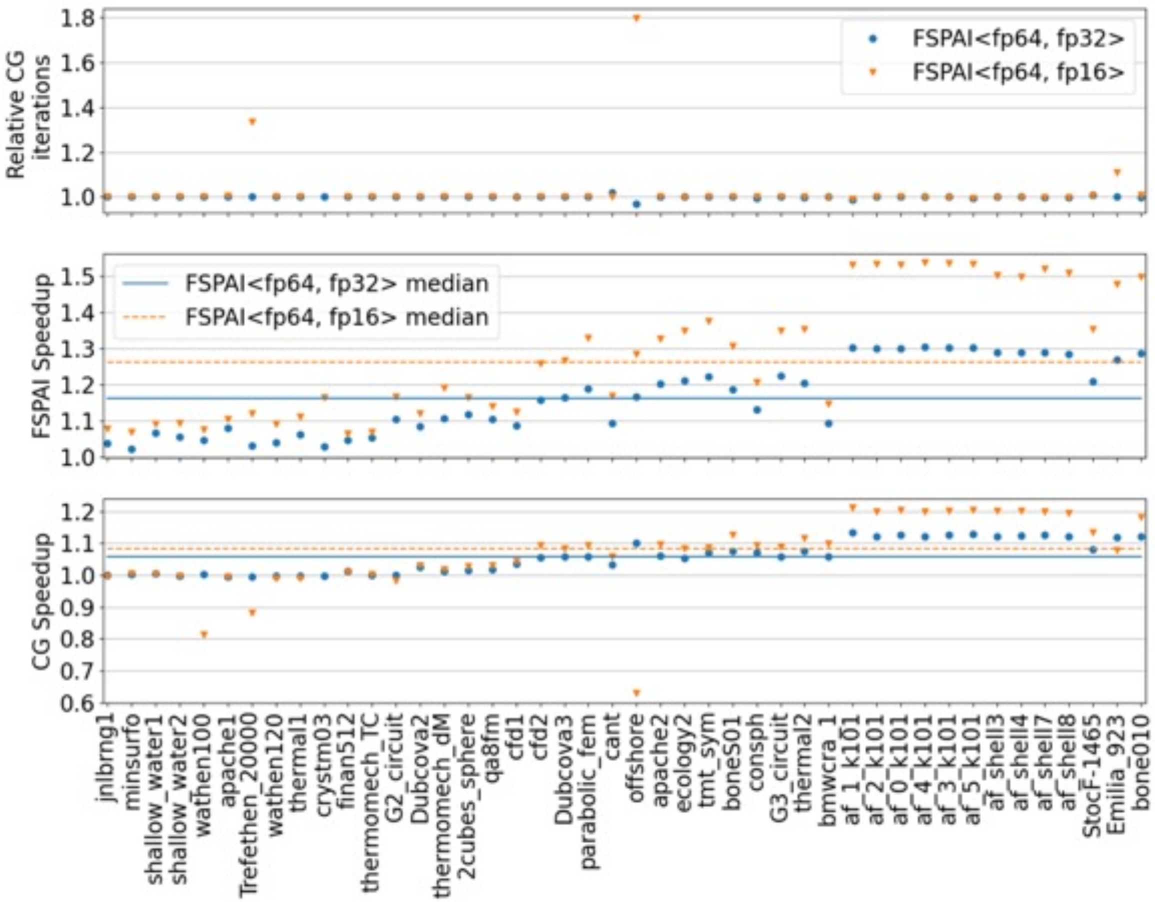
Göbel at al: Mixed Precision Incomplete and Factorized Sparse Approximate Inverse Preconditioning on GPUs, EuroPar submitted.

Mixed Precision Sparse Approximate Inverse

ISAI



FSPAI



- ~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>

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