Ginkgo

Generated from pipelines/88870630 branch based on develop. Ginkgo version 1.0.0

Generated by Doxygen 1.8.16

Chapter 1

Main Page

This is the main page for the Ginkgo library pdf documentation. The repository is hosted on `github`. Documentation on aspects such as the build system, can be found at the Installation Instructions page. The Example programs can help you get started with using Ginkgo.

1.0.0.1 Modules

The Ginkgo library can be grouped into `modules` and these modules form the basic building blocks of Ginkgo. The modules can be summarized as follows:

- Executors : Where do you want your code to be executed ?
- Linear Operators : What kind of operation do you want Ginkgo to perform ?
  - Solvers : Solve a linear system for a given matrix.
  - Preconditioners : Precondition a system for a solve.
  - SpMV employing different Matrix formats : Perform a sparse matrix vector multiplication with a particular matrix format.
- Logging : Monitor your code execution.
- Stopping criteria : Manage your iteration stopping criteria.

Chapter 2

Installation Instructions

2.0.1 Building

Use the standard cmake build procedure:

```
mkdir build; cd build
cmake -G "Unix Makefiles" [OPTIONS] .. && make
```

Replace `[OPTIONS]` with desired cmake options for your build. Ginkgo adds the following additional switches to control what is being built:

- `-DGINKGO_DEVEL_TOOLS={ON, OFF}` sets up the build system for development (requires clang-format, will also download git-cmake-format), default is `ON`
- `-DGINKGO_BUILD_TESTS={ON, OFF}` builds Ginkgo's tests (will download googletest), default is `ON`
- `-DGINKGO_BUILD_BENCHMARKS={ON, OFF}` builds Ginkgo's benchmarks (will download gflags and rapidjson), default is `ON`
- `-DGINKGO_BUILD_EXAMPLES={ON, OFF}` builds Ginkgo's examples, default is `ON`
- `-DGINKGO_BUILD_EXTLIB_EXAMPLE={ON, OFF}` builds the interfacing example with deal.II, default is `OFF`
- `-DGINKGO_BUILD_REFERENCE={ON, OFF}` build reference implementations of the kernels, useful for testing, default is `ON`
- `-DGINKGO_BUILD_OMP={ON, OFF}` builds optimized OpenMP versions of the kernels, default is `OFF`
- `-DGINKGO_BUILD_CUDA={ON, OFF}` builds optimized cuda versions of the kernels (requires CUDA), default is `OFF`
- `-DGINKGO_BUILD_DOC={ON, OFF}` creates an HTML version of Ginkgo's documentation from inline comments in the code. The default is `OFF`.
- `-DGINKGO_DOC_GENERATE_EXAMPLES={ON, OFF}` generates the documentation of examples in Ginkgo. The default is `ON`.
- `-DGINKGO_DOC_GENERATE_PDF={ON, OFF}` generates a PDF version of Ginkgo's documentation from inline comments in the code. The default is `OFF`.
- `-DGINKGO_DOC_GENERATE_DEV={ON, OFF}` generates the developer version of Ginkgo's documentation. The default is `OFF`.
- `-DGINKGO_EXPORT_BUILD_DIR={ON, OFF}` adds the Ginkgo build directory to the CMake package registry. The default is `OFF`.

- -DGINKGO_WITH_CLANG_TIDY={ON, OFF} makes Ginkgo call clang-tidy to find programming issues. The path can be manually controlled with the CMake variable -DGINKGO_CLANG_TIDY_PA↩TH=<path>.
- -DGINKGO_WITH_IWYU={ON, OFF} makes Ginkgo call iwyu to find include issues. The path can be manually controlled with the CMake variable -DGINKGO_IWYU_PATH=<path>.
- -DGINKGO_VERBOSE_LEVEL=integer sets the verbosity of Ginkgo.
  - 0 disables all output in the main libraries,
  - 1 enables a few important messages related to unexpected behavior (default).
- -DCMAKE_INSTALL_PREFIX=path sets the installation path for make install. The default value is usually something like /usr/local
- -DCMAKE_BUILD_TYPE=type specifies which configuration will be used for this build of Ginkgo. The default is RELEASE. Supported values are CMake's standard build types such as DEBUG and RELEASE and the Ginkgo specific COVERAGE, ASAN (AddressSanitizer) and TSAN (ThreadSanitizer) types.
- -DBUILD_SHARED_LIBS={ON, OFF} builds ginkgo as shared libraries (OFF) or as dynamic libraries (ON), default is ON
- -DGINKGO_JACOBI_FULL_OPTIMIZATIONS={ON, OFF} use all the optimizations for the CUDA Jacobi algorithm. OFF by default. Setting this option to ON may lead to very slow compile time (>20 minutes) for the jacobi_generate_kernels.cu file and high memory usage.
- -DCMAKE_CUDA_HOST_COMPILER=path instructs the build system to explicitly set CUDA's host compiler to the path given as argument. By default, CUDA uses its toolchain's host compiler. Setting this option may help if you're experiencing linking errors due to ABI incompatibilities. This option is supported since CMake 3.8 but documented starting from 3.10.
- -DGINKGO_CUDA_ARCHITECTURES=<list> where <list> is a semicolon (;) separated list of architectures. Supported values are:
  - Auto
  - Kepler, Maxwell, Pascal, Volta
  - CODE, CODE(COMPUTE), (COMPUTE)

  Auto will automatically detect the present CUDA-enabled GPU architectures in the system. Kepler, Maxwell, Pascal and Volta will add flags for all architectures of that particular NVIDIA GPU generation. COMPUTE and CODE are placeholders that should be replaced with compute and code numbers (e.g. for compute_70 and sm_70 COMPUTE and CODE should be replaced with 70. Default is Auto. For a more detailed explanation of this option see the ARCHITECTURES specification list section in the documentation of the CudaArchitectureSelector CMake module.

For example, to build everything (in debug mode), use:

```
cmake  -G "Unix Makefiles" -H. -BDebug -DCMAKE_BUILD_TYPE=Debug -DGINKGO_DEVEL_TOOLS=ON \
               -DGINKGO_BUILD_TESTS=ON -DGINKGO_BUILD_REFERENCE=ON -DGINKGO_BUILD_OMP=ON \
               -DGINKGO_BUILD_CUDA=ON
cmake --build Debug
```

NOTE: Ginkgo is known to work with the Unix Makefiles and Ninja based generators. Other CMake generators are untested.

2.0.2 Third party libraries and packages

Ginkgo relies on third party packages in different cases. These third party packages can be turned off by disabling the relevant options.

- GINKGO_BUILD_CUDA=ON: CudaArchitectureSelector (CAS) is a CMake helper to manage CUDA architecture settings;
- GINKGO_BUILD_TESTS=ON: Our tests are implemented with Google Test;
- GINKGO_BUILD_BENCHMARKS=ON: For argument management we use gflags and for JSON parsing we use RapidJSON;
- GINKGO_DEVEL_TOOLS=ON: git-cmake-format is our CMake helper for code formatting.

By default, Ginkgo uses the internal version of each package. For each of the packages GTEST, GFLAGS, RAPIDJSON and CAS, it is possible to force Ginkgo to try to use an external version of a package. For this, Ginkgo provides two ways to find packages. To rely on the CMake find_package command, use the CMake option -DGINKGO_USE_EXTERNAL_<package>=ON. Note that, if the external packages were not installed to the default location, the CMake option -DCMAKE_PREFIX_PATH=<path-list> needs to be set to the semicolon (;) separated list of install paths of these external packages. For more Information, see the CMake documentation for CMAKE_PREFIX_PATH for details.

To manually configure the paths Ginkgo relies on the standard xSDK Installation policies for all packages except CAS (as it is neither a library nor a header, it cannot be expressed through the TPL format):

- -DTPL_ENABLE_<package>=ON
- -DTPL_<package>_LIBRARIES=/path/to/libraries.{so|a}
- -DTPL_<package>_INCLUDE_DIRS=/path/to/header/directory

When applicable (e.g. for GTest libraries), a ; separated list can be given to the TPL_<package>_{LIBRARIES|INCLUDE_DIRS} variables.

2.0.3 Installing Ginkgo

To install Ginkgo into the specified folder, execute the following command in the build folder
```
make install
```

If the installation prefix (see CMAKE_INSTALL_PREFIX) is not writable for your user, e.g. when installing Ginkgo system-wide, it might be necessary to prefix the call with sudo.

After the installation, CMake can find ginkgo with find_package(Ginkgo). An example can be found in the test_install.

Chapter 3

Testing Instructions

3.0.1 Running the unit tests

You need to compile ginkgo with `-DGINKGO_BUILD_TESTS=ON` option to be able to run the tests.

3.0.1.1 Using make test

After configuring Ginkgo, use the following command inside the build folder to run all tests:
```
make test
```

The output should contain several lines of the form:
```
    Start  1: path/to/test
1/13 Test  #1: path/to/test ............................   Passed    0.01 sec
```

To run only a specific test and see more details results (e.g. if a test failed) run the following from the build folder:
```
./path/to/test
```

where `path/to/test` is the path returned by `make test`.

3.0.1.2 Using CTest

The tests can also be ran through CTest from the command line, for example when in a configured build directory:
```
ctest -T start -T build -T test -T submit
```

Will start a new test campaign (usually in `Experimental` mode), build Ginkgo with the set configuration, run the tests and submit the results to our CDash dashboard.

Another option is to use Ginkgo's CTest script which is configured to build Ginkgo with default settings, runs the tests and submits the test to our CDash dashboard automatically.

To run the script, use the following command:
```
ctest -S cmake/CTestScript.cmake
```

The default settings are for our own CI system. Feel free to configure the script before launching it through variables or by directly changing its values. A documentation can be found in the script itself.

Chapter 4

Running the benchmarks

In addition to the unit tests designed to verify correctness, Ginkgo also includes a benchmark suite for checking its performance on the system. To compile the benchmarks, the flag `-DGINKGO_BUILD_BENCHMARKS=ON` has to be set during the `cmake` step. In addition, the `ssget command-line utility` has to be installed on the system.

The benchmark suite tests Ginkgo's performance using the `SuiteSparse matrix collection` and artificially generated matrices. The suite sparse collection will be downloaded automatically when the benchmarks are run. Please note that the entire collection requires roughly 100GB of disk storage in its compressed format, and roughly 25GB of additional disk space for intermediate data (such us uncompressing the archive). Additionally, the benchmark runs usually take a long time (SpMV benchmarks on the complete collection take roughly 24h using the K20 GPU), and will stress the system.

The benchmark suite is invoked using the `make benchmark` command in the build directory. The behavior of the suite can be modified using environment variables. Assuming the `bash` shell is used, these can either be specified via the `export` command to persist between multiple runs:

```
export VARIABLE="value"
...
make benchmark
```

or specified on the fly, on the same line as the `make benchmark` command:

```
env VARIABLE="value" ... make benchmark
```

Since `make` sets any variables passed to it as temporary environment variables, the following shorthand can also be used:

```
make benchmark VARIABLE="value" ...
```

A combination of the above approaches is also possible (e.g. it may be useful to `export` the `SYSTEM_NAME` variable, and specify the others at every benchmark run).

Supported environment variables are described in the following list:

- `BENCHMARK={spmv, solver, preconditioner}` - The benchmark set to run. Default is `spmv`.
  - `spmv` - Runs the sparse matrix-vector product benchmarks on the SuiteSparse collection.
  - `solver` - Runs the solver benchmarks on the SuiteSparse collection. The matrix format is determined by running the `spmv` benchmarks first, and using the fastest format determined by that benchmark. The maximum number of iterations for the iterative solvers is set to 10,000 and the requested residual reduction factor to 1e-6.
  - `preconditioner` - Runs the preconditioner benchmarks on artificially generated block-diagonal matrices.

- `DRY_RUN={true, false}` - If set to `true`, prepares the system for the benchmark runs (downloads the collections, creates the result structure, etc.) and outputs the list of commands that would normally be run, but does not run the benchmarks themselves. Default is `false`.
- `EXECUTOR={reference,cuda,omp}` - The executor used for running the benchmarks. Default is `cuda`.
- `SEGMENTS=<N>` - Splits the benchmark suite into `<N>` segments. This option is useful for running the benchmarks on an HPC system with a batch scheduler, as it enables partitioning of the benchmark suite and running it concurrently on multiple nodes of the system. If specified, `SEGMENT_ID` also has to be set. Default is `1`.
- `SEGMENT_ID=<I>` - used in combination with the `SEGMENTS` variable. `<I>` should be an integer between 1 and `<N>`. If specified, only the `<I>`-th segment of the benchmark suite will be run. Default is `1`.
- `SYSTEM_NAME=<name>` - the name of the system where the benchmarks are being run. This option only changes the directory where the benchmark results are stored. It can be used to avoid overwriting the benchmarks if multiple systems share the same filesystem, or when copying the results between systems. Default is `unknown`.

Once `make benchmark` completes, the results can be found in `<Ginkgo build directory>/benchmark/results/<YSTEM_NAME>/`. The files are written in the JSON format, and can be analyzed using any of the data analysis tools that support JSON. Alternatively, they can be uploaded to an online repository, and analyzed using Ginkgo's free web tool `Ginkgo Performance Explorer (GPE)`. (Make sure to change the "Performance data URL" to your repository if using GPE.)

Chapter 5

Example programs

Here you can find example programs that demonstrate the usage of Ginkgo. Some examples are built on one another and some are stand-alone and demonstrate a concept of Ginkgo, which can be used in your own code.

You can browse the available example programs

1. as **a graph** that shows how example programs build upon each other.
2. as **a list** that provides a short synopsis of each program.
3. or **grouped by topic**.

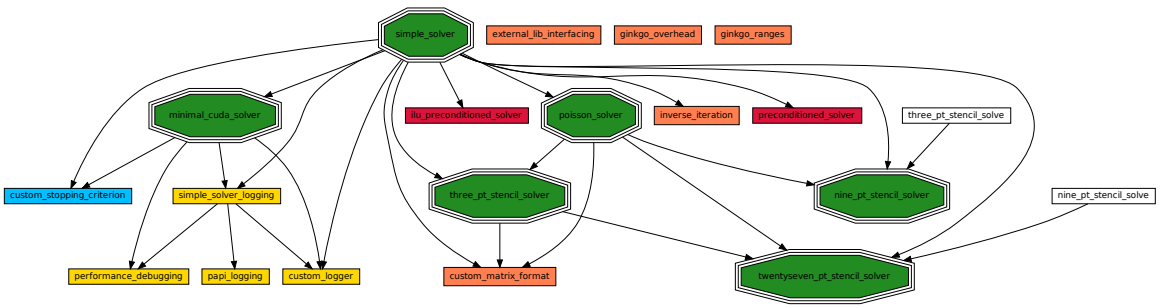
By default, all Ginkgo examples are built using CMake.

An example for building the examples and using Ginkgo as an external library without CMake can be found in the script provided for each example, which should be called with the form: `./build.sh PATH_TO_GINKGO_BUILD_DIR`

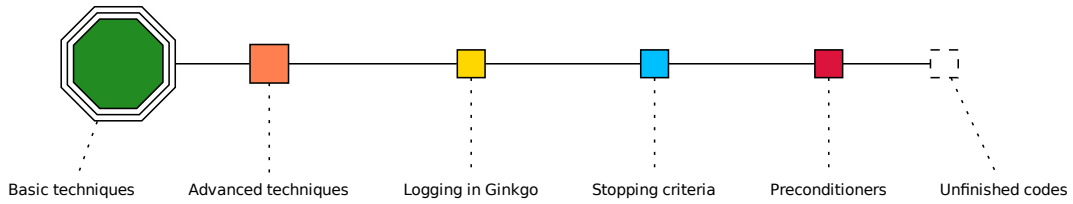
By default, Ginkgo is compiled with at least `-DGINKGO_BUILD_REFERENCE=ON`. To execute on a GPU, you need to have a GPU on the system and must have compiled Ginkgo with the `-DGINKGO_BUILD_CUDA=ON` option.

Connections between example programs

The following graph shows the connections between example programs and how they build on each other. Click on any of the boxes to go to one of the programs. If you hover your mouse pointer over a box, a brief description of the program should appear.



Legend:



Example programs

simple_solver	A minimal CG solver in Ginkgo, which reads a matrix from a file.
minimal_cuda_solver	A minimal solver on the CUDA executor than can be run on NVIDIA GPU's.
poisson_solver	Solve an actual physically relevant problem, the poisson problem. The matrix is generated within Ginkgo.
preconditioned_solver	Using a Jacobi preconditioner to solve a linear system.
three_pt_stencil_solver	Using a three point stencil to solve the poisson equation with array views.
nine_pt_stencil_solver	Using a nine point 2D stencil to solve the poisson equation with array views.
twentyseven_pt_stencil_solver	Using a twentyseven point 3D stencil to solve the poisson equation with array views.
external_lib_interfacing	Using Ginkgo's solver with the external library deal.II.
custom_logger	Creating a custom logger specifically for comparing the recurrent and the real residual norms.
custom_matrix_format	Creating a matrix-free stencil solver by using Ginkgo's advanced methods to build your own custom matrix format.
inverse_iteration	Using Ginkgo to compute eigenvalues of a matrix with the inverse iteration method.
simple_solver_logging	Using the logging functionality in Ginkgo to get solver and other information to diagnose and debug your code.
papi_logging	Using the PAPI logging library in Ginkgo to get advanced information about your code and its behaviour.
ginkgo_overhead	Measuring the overhead of the Ginkgo library.
custom_stopping_criterion	Creating a custom stopping criterion for the iterative solution process.
ginkgo_ranges	Using the ranges concept to factorize a matrix with the LU factorization.

Example programs grouped by topics

Solving a simple linear system with choice of executors.	simple_solver
Using the CUDA executor	minimal_cuda_solver
Using preconditioners	preconditioned_solver
Solving a physically relevant problem	poisson_solver, three_pt_stencil_solver, nine_↩ pt_stencil_solver, twentyseven_pt_stencil_solver, custom_matrix_format
Reading in a matrix and right hand side from a file.	simple_solver, minimal_cuda_solver, preconditioned_solver, inverse_iteration, simple↩ _solver_logging, papi_logging, custom_stopping_↩ criterion, custom_logger

Basic techniques

Using Ginkgo with external libraries.	external_lib_interfacing
Customizing Ginkgo	custom_logger, custom_stopping_criterion, custom↩ _matrix_format
Writing your own matrix format	custom_matrix_format
Using Ginkgo to construct more complex linear algebra routines.	inverse_iteration
Logging within Ginkgo.	simple_solver_logging, papi_logging, custom_logger
Constructing your own stopping criterion.	custom_stopping_criterion
Using ranges in Ginkgo.	ginkgo_ranges

Advanced techniques

Chapter 6

Module Documentation

6.1 CUDA Executor

A module dedicated to the implementation and usage of the CUDA executor in Ginkgo.

Classes

- class gko::CudaExecutor

  This is the Executor subclass which represents the CUDA device.

6.1.1 Detailed Description

A module dedicated to the implementation and usage of the CUDA executor in Ginkgo.

6.2 Executors

A module dedicated to the implementation and usage of the executors in Ginkgo.

Modules

- CUDA Executor

  A module dedicated to the implementation and usage of the CUDA executor in Ginkgo.
- OpenMP Executor

  A module dedicated to the implementation and usage of the OpenMP executor in Ginkgo.
- Reference Executor

  A module dedicated to the implementation and usage of the Reference executor in Ginkgo.

Classes

- class gko::Operation

  Operations can be used to define functionalities whose implementations differ among devices.
- class gko::Executor

  The first step in using the Ginkgo library consists of creating an executor.
- class gko::executor_deleter< T >

  This is a deleter that uses an executor's `free` method to deallocate the data.
- class gko::OmpExecutor

  This is the Executor subclass which represents the OpenMP device (typically CPU).
- class gko::ReferenceExecutor

  This is a specialization of the OmpExecutor, which runs the reference implementations of the kernels used for debugging purposes.
- class gko::CudaExecutor

  This is the Executor subclass which represents the CUDA device.

Macros

- #define GKO_REGISTER_OPERATION(_name, _kernel)

  Binds a set of device-specific kernels to an Operation.

6.2.1 Detailed Description

A module dedicated to the implementation and usage of the executors in Ginkgo.

Below, we provide a brief introduction to executors in Ginkgo, how they have been implemented, how to best make use of them and how to add new executors.

6.2.2 Executors in Ginkgo.

The first step in using the Ginkgo library consists of creating an executor. Executors are used to specify the location for the data of linear algebra objects, and to determine where the operations will be executed. Ginkgo currently supports three different executor types:

- OpenMP Executor specifies that the data should be stored and the associated operations executed on an OpenMP-supporting device (e.g. host CPU);
- CUDA Executor specifies that the data should be stored and the operations executed on the NVIDIA GPU accelerator;
- Reference Executor executes a non-optimized reference implementation, which can be used to debug the library.

6.2.3 Macro Definition Documentation

6.2.3.1 GKO_REGISTER_OPERATION

```
#define GKO_REGISTER_OPERATION(
              _name,
              _kernel )
```

Binds a set of device-specific kernels to an Operation.

It also defines a helper function which creates the associated operation. Any input arguments passed to the helper function are forwarded to the kernel when the operation is executed.

The kernels used to bind the operation are searched in `kernels::DEV_TYPE` namespace, where `DEV_TYPE` is replaced by `omp`, `cuda` and `reference`.

Parameters

_name	operation name
_kernel	kernel which will be bound to the operation

6.2.3.2 Example

```
 {c++}
// define the omp, cuda and reference kernels which will be bound to the
// operation
namespace kernels {
namespace omp {
void my_kernel(int x) {
     // omp code
}
}
namespace cuda {
void my_kernel(int x) {
     // cuda code
}
}
namespace reference {
void my_kernel(int x) {
```

```
    // reference code
}
}
// Bind the kernels to the operation
GKO_REGISTER_OPERATION(my_op, my_kernel);
int main() {
    // create executors
    auto omp = OmpExecutor::create();
    auto cuda = CudaExecutor::create(omp, 0);
    auto ref = ReferenceExecutor::create();
    // create the operation
    auto op = make_my_op(5); // x = 5
    omp->run(op);  // run omp kernel
    cuda->run(op);  // run cuda kernel
    ref->run(op);  // run reference kernel
}
```

6.3 Factorizations

A module dedicated to the implementation and usage of the Factorizations in Ginkgo.

Namespaces

- gko::factorization

  The Factorization namespace.

Classes

- class gko::factorization::ParIlu< ValueType, IndexType >

  ParILU is an incomplete LU factorization which is computed in parallel.

6.3.1 Detailed Description

A module dedicated to the implementation and usage of the Factorizations in Ginkgo.

6.4 Linear Operators

A module dedicated to the implementation and usage of the Linear operators in Ginkgo.

Modules

- Factorizations

  A module dedicated to the implementation and usage of the Factorizations in Ginkgo.
- SpMV employing different Matrix formats

  A module dedicated to the implementation and usage of the various Matrix Formats in Ginkgo.
- Preconditioners

  A module dedicated to the implementation and usage of the Preconditioners in Ginkgo.
- Solvers

  A module dedicated to the implementation and usage of the Solvers in Ginkgo.

Classes

- class gko::Combination< ValueType >

  The Combination class can be used to construct a linear combination of multiple linear operators `c1 * op1 + c2 * op2 + ...`
- class gko::Composition< ValueType >

  The Composition class can be used to compose linear operators `op1, op2, ..., opn` and obtain the operator `op1 * op2 * ...`
- class gko::LinOpFactory

  A LinOpFactory represents a higher order mapping which transforms one linear operator into another.
- class gko::ReadableFromMatrixData< ValueType, IndexType >

  A LinOp implementing this interface can read its data from a matrix_data structure.
- class gko::WritableToMatrixData< ValueType, IndexType >

  A LinOp implementing this interface can write its data to a matrix_data structure.
- class gko::Preconditionable

  A LinOp implementing this interface can be preconditioned.
- class gko::EnableLinOp< ConcreteLinOp, PolymorphicBase >

  The EnableLinOp mixin can be used to provide sensible default implementations of the majority of the LinOp and PolymorphicObject interface.
- class gko::Perturbation< ValueType >

  The Perturbation class can be used to construct a LinOp to represent the operation `(identity + scalar * basis * projector)`.
- class gko::matrix::Coo< ValueType, IndexType >

  COO stores a matrix in the coordinate matrix format.
- class gko::matrix::Csr< ValueType, IndexType >

  CSR is a matrix format which stores only the nonzero coefficients by compressing each row of the matrix (compressed sparse row format).
- class gko::matrix::Dense< ValueType >

  Dense is a matrix format which explicitly stores all values of the matrix.
- class gko::matrix::Ell< ValueType, IndexType >

  ELL is a matrix format where stride with explicit zeros is used such that all rows have the same number of stored elements.
- class gko::matrix::Hybrid< ValueType, IndexType >

  HYBRID is a matrix format which splits the matrix into ELLPACK and COO format.
- class gko::matrix::Identity< ValueType >

This class is a utility which efficiently implements the identity matrix (a linear operator which maps each vector to itself).

- class gko::matrix::IdentityFactory< ValueType >

  This factory is a utility which can be used to generate Identity operators.
- class gko::matrix::Sellp< ValueType, IndexType >

  SELL-P is a matrix format similar to ELL format.
- class gko::matrix::SparsityCsr< ValueType, IndexType >

  SparsityCsr is a matrix format which stores only the sparsity pattern of a sparse matrix by compressing each row of the matrix (compressed sparse row format).
- class gko::preconditioner::Ilu< LSolverType, USolverType, ReverseApply, IndexTypeParIlu >

  The Incomplete LU (ILU) preconditioner solves the equation $LUx = b$ for a given lower triangular matrix L, an upper triangular matrix U and the right hand side b (can contain multiple right hand sides).
- struct gko::preconditioner::block_interleaved_storage_scheme< IndexType >

  Defines the parameters of the interleaved block storage scheme used by block-Jacobi blocks.
- class gko::preconditioner::Jacobi< ValueType, IndexType >

  A block-Jacobi preconditioner is a block-diagonal linear operator, obtained by inverting the diagonal blocks of the source operator.
- class gko::solver::Bicgstab< ValueType >

  BiCGSTAB or the Bi-Conjugate Gradient-Stabilized is a Krylov subspace solver.
- class gko::solver::Cg< ValueType >

  CG or the conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.
- class gko::solver::Cgs< ValueType >

  CGS or the conjugate gradient square method is an iterative type Krylov subspace method which is suitable for general systems.
- class gko::solver::Fcg< ValueType >

  FCG or the flexible conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.
- class gko::solver::Gmres< ValueType >

  GMRES or the generalized minimal residual method is an iterative type Krylov subspace method which is suitable for nonsymmetric linear systems.
- class gko::solver::LowerTrs< ValueType, IndexType >

  LowerTrs is the triangular solver which solves the system L x = b, when L is a lower triangular matrix.
- class gko::solver::UpperTrs< ValueType, IndexType >

  UpperTrs is the triangular solver which solves the system U x = b, when U is an upper triangular matrix.

Macros

- #define GKO_CREATE_FACTORY_PARAMETERS(_parameters_name, _factory_name)

  This Macro will generate a new type containing the parameters for the factory `_factory_name`.
- #define GKO_ENABLE_LIN_OP_FACTORY(_lin_op, _parameters_name, _factory_name)

  This macro will generate a default implementation of a LinOpFactory for the LinOp subclass it is defined in.
- #define GKO_ENABLE_BUILD_METHOD(_factory_name)

  Defines a build method for the factory, simplifying its construction by removing the repetitive typing of factory's name.
- #define GKO_FACTORY_PARAMETER(_name, ...)

  Creates a factory parameter in the factory parameters structure.

Typedefs

- template<typename ConcreteFactory , typename ConcreteLinOp , typename ParametersType , typename PolymorphicBase = LinOpFactory>
  using gko::EnableDefaultLinOpFactory = EnableDefaultFactory< ConcreteFactory, ConcreteLinOp, ParametersType, PolymorphicBase >

  This is an alias for the EnableDefaultFactory mixin, which correctly sets the template parameters to enable a subclass of LinOpFactory.

6.4.1 Detailed Description

A module dedicated to the implementation and usage of the Linear operators in Ginkgo.

Below we elaborate on one of the most important concepts of Ginkgo, the linear operator. The linear operator (LinOp) is a base class for all linear algebra objects in Ginkgo. The main benefit of having a single base class for the entire collection of linear algebra objects (as opposed to having separate hierarchies for matrices, solvers and preconditioners) is the generality it provides.

6.4.2 Advantages of this approach and usage

A common interface often allows for writing more generic code. If a user's routine requires only operations provided by the LinOp interface, the same code can be used for any kind of linear operators, independent of whether these are matrices, solvers or preconditioners. This feature is also extensively used in Ginkgo itself. For example, a preconditioner used inside a Krylov solver is a LinOp. This allows the user to supply a wide variety of preconditioners: either the ones which were designed to be used in this scenario (like ILU or block-Jacobi), a user-supplied matrix which is known to be a good preconditioner for the specific problem, or even another solver (e.g., if constructing a flexible GMRES solver).

For example, a matrix free implementation would require the user to provide an apply implementation and instead of passing the generated matrix to the solver, they would have to provide their apply implementation for all the executors needed and no other code needs to be changed. See custom_matrix_format example for more details.

6.4.3 Linear operator as a concept

The linear operator (LinOp) is a base class for all linear algebra objects in Ginkgo. The main benefit of having a single base class for the entire collection of linear algebra objects (as opposed to having separate hierarchies for matrices, solvers and preconditioners) is the generality it provides.

First, since all subclasses provide a common interface, the library users are exposed to a smaller set of routines. For example, a matrix-vector product, a preconditioner application, or even a system solve are just different terms given to the operation of applying a certain linear operator to a vector. As such, Ginkgo uses the same routine name, LinOp::apply() for each of these operations, where the actual operation performed depends on the type of linear operator involved in the operation.

Second, a common interface often allows for writing more generic code. If a user's routine requires only operations provided by the LinOp interface, the same code can be used for any kind of linear operators, independent of whether these are matrices, solvers or preconditioners. This feature is also extensively used in Ginkgo itself. For example, a preconditioner used inside a Krylov solver is a LinOp. This allows the user to supply a wide variety of preconditioners: either the ones which were designed to be used in this scenario (like ILU or block-Jacobi), a user-supplied matrix which is known to be a good preconditioner for the specific problem, or even another solver (e.g., if constructing a flexible GMRES solver).

A key observation for providing a unified interface for matrices, solvers, and preconditioners is that the most common operation performed on all of them can be expressed as an application of a linear operator to a vector:

- the sparse matrix-vector product with a matrix A is a linear operator application $y = Ax$;
- the application of a preconditioner is a linear operator application $y = M^{-1}x$, where M is an approximation of the original system matrix A (thus a preconditioner represents an "approximate inverse" operator M^{-1}).
- the system solve $Ax = b$ can be viewed as linear operator application $x = A^{-1}b$ (it goes without saying that the implementation of linear system solves does not follow this conceptual idea), so a linear system solver can be viewed as a representation of the operator A^{-1}.

Finally, direct manipulation of LinOp objects is rarely required in simple scenarios. As an illustrative example, one could construct a fixed-point iteration routine $x_{k+1} = Lx_k + b$ as follows:

```
std::unique_ptr<matrix::Dense<» calculate_fixed_point(
        int iters, const LinOp *L, const matrix::Dense<> *x0
        const matrix::Dense<> *b)
{
    auto x = gko::clone(x0);
    auto tmp = gko::clone(x0);
    auto one = Dense<>::create(L->get_executor(), {1.0,});
    for (int i = 0; i < iters; ++i) {
        L->apply(gko::lend(tmp), gko::lend(x));
        x->add_scaled(gko::lend(one), gko::lend(b));
        tmp->copy_from(gko::lend(x));
    }
    return x;
}
```

Here, if L is a matrix, LinOp::apply() refers to the matrix vector product, and `L->apply(a, b)` computes $b = L \cdot a$. `x->add_scaled(one.get(), b.get())` is the `axpy` vector update $x := x + b$.

The interesting part of this example is the apply() routine at line 4 of the function body. Since this routine is part of the LinOp base class, the fixed-point iteration routine can calculate a fixed point not only for matrices, but for any type of linear operator.

Linear Operators

6.4.4 Macro Definition Documentation

6.4.4.1 GKO_CREATE_FACTORY_PARAMETERS

```
#define GKO_CREATE_FACTORY_PARAMETERS(
              _parameters_name,
              _factory_name )
```

Value:

```
public:                                                                 \
    class _factory_name;                                                \
    struct _parameters_name##_type                                      \
        : ::gko::enable_parameters_type<_parameters_name##_type,        \
                                        _factory_name>
```

This Macro will generate a new type containing the parameters for the factory `_factory_name`.

For more details, see GKO_ENABLE_LIN_OP_FACTORY(). It is required to use this macro **before** calling the macro GKO_ENABLE_LIN_OP_FACTORY(). It is also required to use the same names for all parameters between both macros.

Parameters

_parameters_name	name of the parameters member in the class
_factory_name	name of the generated factory type

6.4.4.2 GKO_ENABLE_BUILD_METHOD

```
#define GKO_ENABLE_BUILD_METHOD(
```

```
                _factory_name )
```

Value:

```
static auto build()->decltype(_factory_name::create())                  \
    {                                                                       \
        return _factory_name::create();                                     \
    }                                                                       \
    static_assert(true,                                                     \
                  "This assert is used to counter the false positive extra " \
                  "semi-colon warnings")
```

Defines a build method for the factory, simplifying its construction by removing the repetitive typing of factory's name.

Parameters

_factory_name	the factory for which to define the method

6.4.4.3 GKO_ENABLE_LIN_OP_FACTORY

```
#define GKO_ENABLE_LIN_OP_FACTORY(
                _lin_op,
                _parameters_name,
                _factory_name )
```

Value:

```
public:                                                                     \
    const _parameters_name##_type &get_##_parameters_name() const           \
    {                                                                       \
        return _parameters_name##_;                                         \
    }                                                                       \
                                                                            \
    class _factory_name                                                     \
        : public ::gko::EnableDefaultLinOpFactory<_factory_name, _lin_op,   \
                                                  _parameters_name##_type> { \
        friend class ::gko::EnablePolymorphicObject<_factory_name,          \
                                                    ::gko::LinOpFactory>;   \
        friend class ::gko::enable_parameters_type<_parameters_name##_type, \
                                                   _factory_name>;          \
        using ::gko::EnableDefaultLinOpFactory<                             \
            _factory_name, _lin_op,                                         \
            _parameters_name##_type>::EnableDefaultLinOpFactory;            \
    };                                                                      \
    friend ::gko::EnableDefaultLinOpFactory<_factory_name, _lin_op,         \
                                            _parameters_name##_type>;       \
                                                                            \
                                                                            \
private:                                                                    \
    _parameters_name##_type _parameters_name##_;                            \
                                                                            \
public:                                                                     \
    static_assert(true,                                                     \
                  "This assert is used to counter the false positive extra " \
                  "semi-colon warnings")
```

This macro will generate a default implementation of a LinOpFactory for the LinOp subclass it is defined in.

It is required to first call the macro GKO_CREATE_FACTORY_PARAMETERS() before this one in order to instantiate the parameters type first.

The list of parameters for the factory should be defined in a code block after the macro definition, and should contain a list of GKO_FACTORY_PARAMETER declarations. The class should provide a constructor with signature _lin↩ _op(const _factory_name *, std::shared_ptr<const LinOp>) which the factory will use a callback to construct the object.

A minimal example of a linear operator is the following:

```
 {c++}
struct MyLinOp : public EnableLinOp<MyLinOp> {
    GKO_ENABLE_LIN_OP_FACTORY(MyLinOp, my_parameters, Factory) {
        // a factory parameter named "my_value", of type int and default
        // value of 5
        int GKO_FACTORY_PARAMETER(my_value, 5);
    };
    // constructor needed by EnableLinOp
    explicit MyLinOp(std::shared_ptr<const Executor> exec) {
        : EnableLinOp<MyLinOp>(exec) {}
    // constructor needed by the factory
    explicit MyLinOp(const Factory *factory,
                     std::shared_ptr<const LinOp> matrix)
        : EnableLinOp<MyLinOp>(factory->get_executor()), matrix->get_size()),
          // store factory's parameters locally
          my_parameters_{factory->get_parameters()},
    {
         int value = my_parameters_.my_value;
         // do something with value
    }
```

MyLinOp can then be created as follows:

```
 {c++}
auto exec = gko::ReferenceExecutor::create();
// create a factory with default `my_value` parameter
auto fact = MyLinOp::build().on(exec);
// create a operator using the factory:
auto my_op = fact->generate(gko::matrix::Identity::create(exec, 2));
std::cout « my_op->get_my_parameters().my_value;  // prints 5
// create a factory with custom `my_value` parameter
auto fact = MyLinOp::build().with_my_value(0).on(exec);
// create a operator using the factory:
auto my_op = fact->generate(gko::matrix::Identity::create(exec, 2));
std::cout « my_op->get_my_parameters().my_value;  // prints 0
```

Note

It is possible to combine both the #GKO_CREATE_FACTORY_PARAMETER() macro with this one in a unique macro for class **templates** (not with regular classes). Splitting this into two distinct macros allows to use them in all contexts. See https://stackoverflow.com/q/50202718/9385966 for more details.

Parameters

_lin_op	concrete operator for which the factory is to be created [CRTP parameter]
_parameters_name	name of the parameters member in the class (its type is `<_parameters_name>_type`, the protected member's name is `<_parameters_name>_`, and the public getter's name is `get_<_parameters_name>()`)
_factory_name	name of the generated factory type

6.4.4.4 GKO_FACTORY_PARAMETER

```
#define GKO_FACTORY_PARAMETER(
                _name,
                ...  )
```

Value:

```
mutable _name{__VA_ARGS__};                                                \
                                                                             \
    template <typename... Args>                                              \
    auto with_##_name(Args &&... _value)                                     \
        const->const ::gko::xstd::decay_t<decltype(*this)> &                 \
    {                                                                        \
        using type = decltype(this->_name);                                  \
```

```
        this->_name = type{std::forward<Args>(_value)...};                  \
        return *this;                                                       \
    }                                                                       \
    static_assert(true,                                                     \
                  "This assert is used to counter the false positive extra " \
                  "semi-colon warnings")
```

Creates a factory parameter in the factory parameters structure.

Parameters

_name	name of the parameter
VA_ARGS	default value of the parameter

See also

GKO_ENABLE_LIN_OP_FACTORY for more details, and usage example

6.4.5 Typedef Documentation

6.4.5.1 EnableDefaultLinOpFactory

```
template<typename ConcreteFactory , typename ConcreteLinOp , typename ParametersType , typename
PolymorphicBase = LinOpFactory>
using gko::EnableDefaultLinOpFactory = typedef EnableDefaultFactory<ConcreteFactory, Concrete↩
LinOp, ParametersType, PolymorphicBase>
```

This is an alias for the EnableDefaultFactory mixin, which correctly sets the template parameters to enable a subclass of LinOpFactory.

Template Parameters

ConcreteFactory	the concrete factory which is being implemented [CRTP parmeter]
ConcreteLinOp	the concrete LinOp type which this factory produces, needs to have a constructor which takes a const ConcreteFactory $*$, and an std::shared_ptr<const LinOp> as parameters.
ParametersType	a subclass of enable_parameters_type template which defines all of the parameters of the factory
PolymorphicBase	parent of ConcreteFactory in the polymorphic hierarchy, has to be a subclass of LinOpFactory

6.5 Logging

A module dedicated to the implementation and usage of the Logging in Ginkgo.

Namespaces

- gko::log

  The logger namespace .

Classes

- class gko::log::Convergence< ValueType >

  Convergence is a Logger which logs data strictly from the `criterion_check_completed` *event.*
- class gko::log::Stream< ValueType >

  Stream is a Logger which logs every event to a stream.

6.5.1 Detailed Description

A module dedicated to the implementation and usage of the Logging in Ginkgo.

The Logger class represents a simple Logger object. It comprises all masks and events internally. Every new logging event addition should be done here. The Logger class also provides a default implementation for most events which do nothing, therefore it is not an obligation to change all classes which derive from Logger, although it is good practice. The logger class is built using event masks to control which events should be logged, and which should not.

6.6 SpMV employing different Matrix formats

A module dedicated to the implementation and usage of the various Matrix Formats in Ginkgo.

Classes

- class gko::matrix::Coo< ValueType, IndexType >

  COO stores a matrix in the coordinate matrix format.
- class gko::matrix::Csr< ValueType, IndexType >

  CSR is a matrix format which stores only the nonzero coefficients by compressing each row of the matrix (compressed sparse row format).
- class gko::matrix::Dense< ValueType >

  Dense is a matrix format which explicitly stores all values of the matrix.
- class gko::matrix::Ell< ValueType, IndexType >

  ELL is a matrix format where stride with explicit zeros is used such that all rows have the same number of stored elements.
- class gko::matrix::Hybrid< ValueType, IndexType >

  HYBRID is a matrix format which splits the matrix into ELLPACK and COO format.
- class gko::matrix::Identity< ValueType >

  This class is a utility which efficiently implements the identity matrix (a linear operator which maps each vector to itself).
- class gko::matrix::IdentityFactory< ValueType >

  This factory is a utility which can be used to generate Identity operators.
- class gko::matrix::Sellp< ValueType, IndexType >

  SELL-P is a matrix format similar to ELL format.
- class gko::matrix::SparsityCsr< ValueType, IndexType >

  SparsityCsr is a matrix format which stores only the sparsity pattern of a sparse matrix by compressing each row of the matrix (compressed sparse row format).

Functions

- template<typename Matrix , typename... TArgs>
  std::unique_ptr< Matrix > gko::initialize (size_type stride, std::initializer_list< typename Matrix::value_type > vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)

  Creates and initializes a column-vector.
- template<typename Matrix , typename... TArgs>
  std::unique_ptr< Matrix > gko::initialize (std::initializer_list< typename Matrix::value_type > vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)

  Creates and initializes a column-vector.
- template<typename Matrix , typename... TArgs>
  std::unique_ptr< Matrix > gko::initialize (size_type stride, std::initializer_list< std::initializer_list< typename Matrix::value_type >> vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)

  Creates and initializes a matrix.
- template<typename Matrix , typename... TArgs>
  std::unique_ptr< Matrix > gko::initialize (std::initializer_list< std::initializer_list< typename Matrix::value_type >> vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)

  Creates and initializes a matrix.

6.6.1 Detailed Description

A module dedicated to the implementation and usage of the various Matrix Formats in Ginkgo.

6.6.2 Function Documentation

6.6.2.1 initialize() [1/4]

```
template<typename Matrix , typename...  TArgs>
std::unique_ptr<Matrix> gko::initialize (
             size_type stride,
             std::initializer_list< std::initializer_list< typename Matrix::value_type >>
vals,
             std::shared_ptr< const Executor > exec,
             TArgs &&...  create_args )
```

Creates and initializes a matrix.

This function first creates a temporary Dense matrix, fills it with passed in values, and then converts the matrix to the requested type.

Template Parameters

Matrix	matrix type to initialize (Dense has to implement the ConvertibleTo<Matrix> interface)
TArgs	argument types for Matrix::create method (not including the implied Executor as the first argument)

Parameters

stride	row stride for the temporary Dense matrix
vals	values used to initialize the matrix
exec	Executor associated to the matrix
create_args	additional arguments passed to Matrix::create, not including the Executor, which is passed as the first argument

```
605 {
606     using dense = matrix::Dense<typename Matrix::value_type>;
607     size_type num_rows = vals.size();
608     size_type num_cols = num_rows > 0 ? begin(vals)->size() : 1;
609     auto tmp =
610         dense::create(exec->get_master(), dim<2>{num_rows, num_cols}, stride);
611     size_type ridx = 0;
612     for (const auto &row : vals) {
613         size_type cidx = 0;
614         for (const auto &elem : row) {
615             tmp->at(ridx, cidx) = elem;
616             ++cidx;
617         }
618         ++ridx;
619     }
620     auto mtx = Matrix::create(exec, std::forward<TArgs>(create_args)...);
621     tmp->move_to(mtx.get());
622     return mtx;
623 }
```

References gko::matrix::Dense< ValueType >::at().

6.6.2.2 initialize() [2/4]

```
template<typename Matrix , typename...  TArgs>
std::unique_ptr<Matrix> gko::initialize (
```

```
            size_type stride,
            std::initializer_list< typename Matrix::value_type > vals,
            std::shared_ptr< const Executor > exec,
            TArgs &&...  create_args )
```

Creates and initializes a column-vector.

This function first creates a temporary Dense matrix, fills it with passed in values, and then converts the matrix to the requested type.

Template Parameters

Matrix	matrix type to initialize (Dense has to implement the ConvertibleTo<Matrix> interface)
TArgs	argument types for Matrix::create method (not including the implied Executor as the first argument)

Parameters

stride	row stride for the temporary Dense matrix
vals	values used to initialize the vector
exec	Executor associated to the vector
create_args	additional arguments passed to Matrix::create, not including the Executor, which is passed as the first argument

References gko::matrix::Dense< ValueType >::at().

6.6.2.3 initialize() [3/4]

```
template<typename Matrix , typename...  TArgs>
std::unique_ptr<Matrix> gko::initialize (
            std::initializer_list< std::initializer_list< typename Matrix::value_type >>
vals,
            std::shared_ptr< const Executor > exec,
            TArgs &&...  create_args )
```

Creates and initializes a matrix.

This function first creates a temporary Dense matrix, fills it with passed in values, and then converts the matrix to the requested type. The stride of the intermediate Dense matrix is set to the number of columns of the initializer list.

Template Parameters

Matrix	matrix type to initialize (Dense has to implement the ConvertibleTo<Matrix> interface)
TArgs	argument types for Matrix::create method (not including the implied Executor as the first argument)

Parameters

vals	values used to initialize the matrix
exec	Executor associated to the matrix
create_args	additional arguments passed to Matrix::create, not including the Executor, which is passed as the first argument

6.6.2.4 initialize() [4/4]

```
template<typename Matrix , typename...  TArgs>
std::unique_ptr<Matrix> gko::initialize (
            std::initializer_list< typename Matrix::value_type > vals,
            std::shared_ptr< const Executor > exec,
            TArgs &&...  create_args )
```

Creates and initializes a column-vector.

This function first creates a temporary Dense matrix, fills it with passed in values, and then converts the matrix to the requested type. The stride of the intermediate Dense matrix is set to 1.

Template Parameters

Matrix	matrix type to initialize (Dense has to implement the ConvertibleTo<Matrix> interface)
TArgs	argument types for Matrix::create method (not including the implied Executor as the first argument)

Parameters

vals	values used to initialize the vector
exec	Executor associated to the vector
create_args	additional arguments passed to Matrix::create, not including the Executor, which is passed as the first argument

6.7 OpenMP Executor

A module dedicated to the implementation and usage of the OpenMP executor in Ginkgo.

Classes

- class gko::OmpExecutor

  This is the Executor subclass which represents the OpenMP device (typically CPU).

6.7.1 Detailed Description

A module dedicated to the implementation and usage of the OpenMP executor in Ginkgo.

6.8 Preconditioners

A module dedicated to the implementation and usage of the Preconditioners in Ginkgo.

Namespaces

- gko::preconditioner

  The Preconditioner namespace.

Classes

- class gko::Preconditionable

  A LinOp implementing this interface can be preconditioned.
- class gko::preconditioner::Ilu< LSolverType, USolverType, ReverseApply, IndexTypeParIlu >

  The Incomplete LU (ILU) preconditioner solves the equation $LUx = b$ for a given lower triangular matrix L, an upper triangular matrix U and the right hand side b (can contain multiple right hand sides).
- struct gko::preconditioner::block_interleaved_storage_scheme< IndexType >

  Defines the parameters of the interleaved block storage scheme used by block-Jacobi blocks.
- class gko::preconditioner::Jacobi< ValueType, IndexType >

  A block-Jacobi preconditioner is a block-diagonal linear operator, obtained by inverting the diagonal blocks of the source operator.

6.8.1 Detailed Description

A module dedicated to the implementation and usage of the Preconditioners in Ginkgo.

6.9 Reference Executor

A module dedicated to the implementation and usage of the Reference executor in Ginkgo.

Classes

- class gko::ReferenceExecutor

  This is a specialization of the OmpExecutor, which runs the reference implementations of the kernels used for debugging purposes.

6.9.1 Detailed Description

A module dedicated to the implementation and usage of the Reference executor in Ginkgo.

6.10 Solvers

A module dedicated to the implementation and usage of the Solvers in Ginkgo.

Namespaces

- gko::solver

  The ginkgo Solve namespace.

Classes

- class gko::solver::Bicgstab< ValueType >

  BiCGSTAB or the Bi-Conjugate Gradient-Stabilized is a Krylov subspace solver.
- class gko::solver::Cg< ValueType >

  CG or the conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.
- class gko::solver::Cgs< ValueType >

  CGS or the conjugate gradient square method is an iterative type Krylov subspace method which is suitable for general systems.
- class gko::solver::Fcg< ValueType >

  FCG or the flexible conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.
- class gko::solver::Gmres< ValueType >

  GMRES or the generalized minimal residual method is an iterative type Krylov subspace method which is suitable for nonsymmetric linear systems.
- class gko::solver::LowerTrs< ValueType, IndexType >

  LowerTrs is the triangular solver which solves the system L x = b, when L is a lower triangular matrix.
- class gko::solver::UpperTrs< ValueType, IndexType >

  UpperTrs is the triangular solver which solves the system U x = b, when U is an upper triangular matrix.

6.10.1 Detailed Description

A module dedicated to the implementation and usage of the Solvers in Ginkgo.

6.11 Stopping criteria

A module dedicated to the implementation and usage of the Stopping Criteria in Ginkgo.

Namespaces

- gko::stop

  The Stopping criterion namespace.

Classes

- class gko::stop::Combined

  The Combined class is used to combine multiple criterions together through an OR operation.
- class gko::stop::Iteration

  The Iteration class is a stopping criterion which stops the iteration process after a preset number of iterations.
- class gko::stop::ResidualNormReduction< ValueType >

  The ResidualNormReduction class is a stopping criterion which stops the iteration process when the relative residual norm is below a certain threshold.
- class gko::stopping_status

  This class is used to keep track of the stopping status of one vector.
- class gko::stop::Time

  The Time class is a stopping criterion which stops the iteration process after a certain amout of time has passed.

Macros

- #define GKO_ENABLE_CRITERION_FACTORY(_criterion, _parameters_name, _factory_name)

  This macro will generate a default implementation of a CriterionFactory for the Criterion subclass it is defined in.

Functions

- template<typename FactoryContainer >
  std::shared_ptr< const CriterionFactory > gko::stop::combine (FactoryContainer &&factories)

  Combines multiple criterion factories into a single combined criterion factory.

6.11.1 Detailed Description

A module dedicated to the implementation and usage of the Stopping Criteria in Ginkgo.

6.11.2 Macro Definition Documentation

6.11.2.1 GKO_ENABLE_CRITERION_FACTORY

```
#define GKO_ENABLE_CRITERION_FACTORY(
               _criterion,
               _parameters_name,
               _factory_name )
```

Value:

```
public:                                                                         \
    const _parameters_name##_type &get_##_parameters_name() const               \
    {                                                                           \
        return _parameters_name##_;                                             \
    }                                                                           \
                                                                                \
    class _factory_name                                                         \
        : public ::gko::stop::EnableDefaultCriterionFactory<                    \
              _factory_name, _criterion, _parameters_name##_type> {             \
        friend class ::gko::EnablePolymorphicObject<                            \
            _factory_name, ::gko::stop::CriterionFactory>;                      \
        friend class ::gko::enable_parameters_type<_parameters_name##_type,     \
                                                   _factory_name>;              \
        using ::gko::stop::EnableDefaultCriterionFactory<                       \
            _factory_name, _criterion,                                          \
            _parameters_name##_type>::EnableDefaultCriterionFactory;            \
    };                                                                          \
    friend ::gko::stop::EnableDefaultCriterionFactory<                          \
        _factory_name, _criterion, _parameters_name##_type>;                    \
                                                                                \
private:                                                                        \
    _parameters_name##_type _parameters_name##_;                                \
                                                                                \
public:                                                                         \
    static_assert(true,                                                         \
                  "This assert is used to counter the false positive extra " \
                  "semi-colon warnings")
```

This macro will generate a default implementation of a CriterionFactory for the Criterion subclass it is defined in.

This macro is very similar to the macro #ENABLE_LIN_OP_FACTORY(). A more detailed description of the use of these type of macros can be found there.

Parameters

_criterion	concrete operator for which the factory is to be created [CRTP parameter]
_parameters_name	name of the parameters member in the class (its type is `<_parameters_name>_type`, the protected member's name is `<_parameters_name>_`, and the public getter's name is `get_<_parameters_name>()`)
_factory_name	name of the generated factory type

6.11.3 Function Documentation

6.11.3.1 combine()

```
template<typename FactoryContainer >
std::shared_ptr<const CriterionFactory> gko::stop::combine (
             FactoryContainer && factories )
```

Combines multiple criterion factories into a single combined criterion factory.

This function treats a singleton container as a special case and avoids creating an additional object and just returns the input factory.

Template Parameters

FactoryContainer	a random access container type

Parameters

factories	a list of factories to combined

Returns

a combined criterion factory if the input contains multiple factories or the input factory if the input contains only one factory

```
117 {
118     switch (factories.size()) {
119     case 0:
120         GKO_NOT_SUPPORTED(nullptr);
121         return nullptr;
122     case 1:
123         return factories[0];
124     default:
125         auto exec = factories[0]->get_executor();
126         return Combined::build()
127             .with_criteria(std::forward<FactoryContainer>(factories))
128             .on(exec);
129     }
130 }
```

Chapter 7

Namespace Documentation

7.1 gko Namespace Reference

The Ginkgo namespace.

Namespaces

- accessor

  The accessor namespace.
- factorization

  The Factorization namespace.
- log

  The logger namespace .
- matrix

  The matrix namespace.
- name_demangling

  The name demangling namespace.
- preconditioner

  The Preconditioner namespace.
- solver

  The ginkgo Solve namespace.
- stop

  The Stopping criterion namespace.
- syn

  The Synthesizer namespace.
- xstd

  The namespace for functionalities after C++11 standard.

Classes

- class AbstractFactory

  The AbstractFactory is a generic interface template that enables easy implementation of the abstract factory design pattern.
- class AllocationError

  AllocationError is thrown if a memory allocation fails.
- class Array

  An Array is a container which encapsulates fixed-sized arrays, stored on the Executor tied to the Array.
- class BadDimension

  BadDimension is thrown if an operation is being applied to a LinOp with bad dimensions.
- class Combination

  The Combination class can be used to construct a linear combination of multiple linear operators `c1 * op1 + c2 * op2 + ...`
- class Composition

  The Composition class can be used to compose linear operators `op1, op2, ..., opn` *and obtain the operator* `op1 * op2 * ...`
- class ConvertibleTo

  ConvertibleTo interface is used to mark that the implementer can be converted to the object of ResultType.
- class copy_back_deleter

  A copy_back_deleter is a type of deleter that copies the data to an internally referenced object before performing the deletion.
- class CublasError

  CublasError is thrown when a cuBLAS routine throws a non-zero error code.
- class CudaError

  CudaError is thrown when a CUDA routine throws a non-zero error code.
- class CudaExecutor

  This is the Executor subclass which represents the CUDA device.
- class CusparseError

  CusparseError is thrown when a cuSPARSE routine throws a non-zero error code.
- struct default_converter

  Used to convert objects of type `S` *to objects of type* `R` *using static_cast.*
- struct dim

  A type representing the dimensions of a multidimensional object.
- class DimensionMismatch

  DimensionMismatch is thrown if an operation is being applied to LinOps of incompatible size.
- struct enable_parameters_type

  The enable_parameters_type mixin is used to create a base implementation of the factory parameters structure.
- class EnableAbstractPolymorphicObject

  This mixin inherits from (a subclass of) PolymorphicObject and provides a base implementation of a new abstract object.
- class EnableCreateMethod

  This mixin implements a static `create()` *method on* `ConcreteType` *that dynamically allocates the memory, uses the passed-in arguments to construct the object, and returns an std::unique_ptr to such an object.*
- class EnableDefaultFactory

  This mixin provides a default implementation of a concrete factory.
- class EnableLinOp

  The EnableLinOp mixin can be used to provide sensible default implementations of the majority of the LinOp and PolymorphicObject interface.
- class EnablePolymorphicAssignment

  This mixin is used to enable a default PolymorphicObject::copy_from() implementation for objects that have implemented conversions between them.
- class EnablePolymorphicObject

This mixin inherits from (a subclass of) PolymorphicObject and provides a base implementation of a new concrete polymorphic object.

- class Error

  The Error class is used to report exceptional behaviour in library functions.
- class Executor

  The first step in using the Ginkgo library consists of creating an executor.
- class executor_deleter

  This is a deleter that uses an executor's `free` method to deallocate the data.
- class KernelNotFound

  KernelNotFound is thrown if Ginkgo cannot find a kernel which satisfies the criteria imposed by the input arguments.
- class LinOpFactory

  A LinOpFactory represents a higher order mapping which transforms one linear operator into another.
- struct matrix_data

  This structure is used as an intermediate data type to store a sparse matrix.
- class NotCompiled

  NotCompiled is thrown when attempting to call an operation which is a part of a module that was not compiled on the system.
- class NotImplemented

  NotImplemented is thrown in case an operation has not yet been implemented (but will be implemented in the future).
- class NotSupported

  NotSupported is thrown in case it is not possible to perform the requested operation on the given object type.
- class null_deleter

  This is a deleter that does not delete the object.
- class OmpExecutor

  This is the Executor subclass which represents the OpenMP device (typically CPU).
- class Operation

  Operations can be used to define functionalities whose implementations differ among devices.
- class OutOfBoundsError

  OutOfBoundsError is thrown if a memory access is detected to be out-of-bounds.
- class Perturbation

  The Perturbation class can be used to construct a LinOp to represent the operation `(identity + scalar * basis * projector)`.
- class PolymorphicObject

  A PolymorphicObject is the abstract base for all "heavy" objects in Ginkgo that behave polymorphically.
- class precision_reduction

  This class is used to encode storage precisions of low precision algorithms.
- class Preconditionable

  A LinOp implementing this interface can be preconditioned.
- class range

  A range is a multidimensional view of the memory.
- class ReadableFromMatrixData

  A LinOp implementing this interface can read its data from a matrix_data structure.
- class ReferenceExecutor

  This is a specialization of the OmpExecutor, which runs the reference implementations of the kernels used for debugging purposes.
- struct span

  A span is a lightweight structure used to create sub-ranges from other ranges.
- class stopping_status

  This class is used to keep track of the stopping status of one vector.
- class StreamError

  StreamError is thrown if accessing a stream failed.

- class temporary_clone

  A temporary_clone is a special smart pointer-like object that is designed to hold an object temporarily copied to another executor.
- class Transposable

  Linear operators which support transposition should implement the Transposable interface.
- struct version

  This structure is used to represent versions of various Ginkgo modules.
- class version_info

  Ginkgo uses version numbers to label new features and to communicate backward compatibility guarantees:
- class WritableToMatrixData

  A LinOp implementing this interface can write its data to a matrix_data structure.

Typedefs

- template<typename ConcreteFactory , typename ConcreteLinOp , typename ParametersType , typename PolymorphicBase = LinOp↩ Factory>
  using EnableDefaultLinOpFactory = EnableDefaultFactory< ConcreteFactory, ConcreteLinOp, Parameters↩ Type, PolymorphicBase >

  This is an alias for the EnableDefaultFactory mixin, which correctly sets the template parameters to enable a subclass of LinOpFactory.
- template<typename T >
  using remove_complex = typename detail::remove_complex_impl< T >::type

  Obtains a real counterpart of a std::complex type, and leaves the type unchanged if it is not a complex type.
- template<typename T >
  using is_complex_s = detail::is_complex_impl< T >

  Allows to check if T is a complex value during compile time by accessing the `value` attribute of this struct.
- template<typename T >
  using reduce_precision = typename detail::reduce_precision_impl< T >::type

  Obtains the next type in the hierarchy with lower precision than T.
- template<typename T >
  using increase_precision = typename detail::increase_precision_impl< T >::type

  Obtains the next type in the hierarchy with higher precision than T.
- template<typename T , size_type Limit = sizeof(uint16) ∗ byte_size>
  using truncate_type = xstd::conditional_t< detail::type_size_impl< T >::value >=2 ∗Limit, typename detail↩ ::truncate_type_impl< T >::type, T >

  Truncates the type by half (by dropping bits), but ensures that it is at least `Limit` bits wide.
- using size_type = std::size_t

  Integral type used for allocation quantities.
- using int8 = std::int8_t

  8-bit signed integral type.
- using int16 = std::int16_t

  16-bit signed integral type.
- using int32 = std::int32_t

  32-bit signed integral type.
- using int64 = std::int64_t

  64-bit signed integral type.
- using uint8 = std::uint8_t

  8-bit unsigned integral type.
- using uint16 = std::uint16_t

  16-bit unsigned integral type.
- using uint32 = std::uint32_t

32-bit unsigned integral type.

- using uint64 = std::uint64_t

  64-bit unsigned integral type.

- using float16 = half

  Half precision floating point type.

- using float32 = float

  Single precision floating point type.

- using float64 = double

  Double precision floating point type.

- using full_precision = double

  The most precise floating-point type.

- using default_precision = double

  Precision used if no precision is explicitly specified.

Enumerations

- enum layout_type { layout_type::array, layout_type::coordinate }

  Specifies the layout type when writing data in matrix market format.

Functions

- template<size_type Dimensionality, typename DimensionType >
  constexpr bool operator!= (const dim< Dimensionality, DimensionType > &x, const dim< Dimensionality, DimensionType > &y)

  Checks if two dim objects are different.

- template<typename DimensionType >
  constexpr dim< 2, DimensionType > transpose (const dim< 2, DimensionType > &dimensions) noexcept

  Returns a dim<2> object with its dimensions swapped.

- template<typename T >
  constexpr bool is_complex ()

  Checks if T is a complex type.

- template<typename T >
  constexpr reduce_precision< T > round_down (T val)

  Reduces the precision of the input parameter.

- template<typename T >
  constexpr increase_precision< T > round_up (T val)

  Increases the precision of the input parameter.

- constexpr int64 ceildiv (int64 num, int64 den)

  Performs integer division with rounding up.

- template<typename T >
  constexpr T zero ()

  Returns the additive identity for T.

- template<typename T >
  constexpr T zero (const T &)

  Returns the additive identity for T.

- template<typename T >
  constexpr T one ()

  Returns the multiplicative identity for T.

- template<typename T >
  constexpr T one (const T &)

  Returns the multiplicative identity for T.

- template<typename T >
  constexpr T abs (const T &x)

  Returns the absolute value of the object.
- template<typename T >
  constexpr T max (const T &x, const T &y)

  Returns the larger of the arguments.
- template<typename T >
  constexpr T min (const T &x, const T &y)

  Returns the smaller of the arguments.
- template<typename T >
  constexpr T real (const T &x)

  Returns the real part of the object.
- template<typename T >
  constexpr T imag (const T &)

  Returns the imaginary part of the object.
- template<typename T >
  T conj (const T &x)

  Returns the conjugate of an object.
- template<typename T >
  constexpr auto squared_norm (const T &x) -> decltype(real(conj(x) ∗x))

  Returns the squared norm of the object.
- template<typename T >
  constexpr uint32 get_significant_bit (const T &n, uint32 hint=0u) noexcept

  Returns the position of the most significant bit of the number.
- template<typename T >
  constexpr T get_superior_power (const T &base, const T &limit, const T &hint=T{1}) noexcept

  Returns the smallest power of `base` *not smaller than* `limit`.
- template<typename T >
  xstd::enable_if_t< is_complex_s< T >::value, bool > isfinite (const T &value)

  Checks if all components of a complex value are finite, meaning they are neither +/- infinity nor NaN.
- template<typename ValueType = default_precision, typename IndexType = int32>
  matrix_data< ValueType, IndexType > read_raw (std::istream &is)

  Reads a matrix stored in matrix market format from an input stream.
- template<typename ValueType , typename IndexType >
  void write_raw (std::ostream &os, const matrix_data< ValueType, IndexType > &data, layout_type layout=layout_type::array)

  Writes a matrix_data structure to a stream in matrix market format.
- template<typename MatrixType , typename StreamType , typename... MatrixArgs>
  std::unique_ptr< MatrixType > read (StreamType &&is, MatrixArgs &&... args)

  Reads a matrix stored in matrix market format from an input stream.
- template<typename MatrixType , typename StreamType >
  void write (StreamType &&os, MatrixType ∗matrix, layout_type layout=layout_type::array)

  Reads a matrix stored in matrix market format from an input stream.
- template<typename R , typename T >
  std::unique_ptr< R, std::function< void(R ∗)> > copy_and_convert_to (std::shared_ptr< const Executor > exec, T ∗obj)

  Converts the object to R and places it on Executor exec.
- template<typename R , typename T >
  std::unique_ptr< const R, std::function< void(const R ∗)> > copy_and_convert_to (std::shared_ptr< const Executor > exec, const T ∗obj)

  Converts the object to R and places it on Executor exec.
- template<typename R , typename T >
  std::shared_ptr< R > copy_and_convert_to (std::shared_ptr< const Executor > exec, std::shared_ptr< T > obj)

Converts the object to R and places it on Executor exec.

- template<typename R , typename T >
  std::shared_ptr< const R > copy_and_convert_to (std::shared_ptr< const Executor > exec, std::shared_ptr< const T > obj)
- constexpr bool operator== (precision_reduction x, precision_reduction y) noexcept

  Checks if two precision_reduction encodings are equal.
- constexpr bool operator!= (precision_reduction x, precision_reduction y) noexcept

  Checks if two precision_reduction encodings are different.
- template<typename Pointer >
  detail::cloned_type< Pointer > clone (const Pointer &p)

  Creates a unique clone of the object pointed to by p.
- template<typename Pointer >
  detail::cloned_type< Pointer > clone (std::shared_ptr< const Executor > exec, const Pointer &p)

  Creates a unique clone of the object pointed to by p on Executor $exec$.
- template<typename OwningPointer >
  detail::shared_type< OwningPointer > share (OwningPointer &&p)

  Marks the object pointed to by p as shared.
- template<typename OwningPointer >
  std::remove_reference< OwningPointer >::type && give (OwningPointer &&p)

  Marks that the object pointed to by p can be given to the callee.
- template<typename Pointer >
  std::enable_if< detail::have_ownership< Pointer >), detail::pointee< Pointer > * >::type lend (const Pointer &p)

  Returns a non-owning (plain) pointer to the object pointed to by p.
- template<typename Pointer >
  std::enable_if<!detail::have_ownership< Pointer >), detail::pointee< Pointer > * >::type lend (const Pointer &p)

  Returns a non-owning (plain) pointer to the object pointed to by p.
- template<typename T , typename U >
  std::decay< T >::type * as (U *obj)

  Performs polymorphic type conversion.
- template<typename T , typename U >
  const std::decay< T >::type * as (const U *obj)

  Performs polymorphic type conversion.
- template<typename T >
  temporary_clone< T > make_temporary_clone (std::shared_ptr< const Executor > exec, T *ptr)

  Creates a temporary_clone.
- std::ostream & operator<< (std::ostream &os, const version &ver)

  Prints version information to a stream.
- std::ostream & operator<< (std::ostream &os, const version_info &ver_info)

  Prints library version information in human-readable format to a stream.
- template<typename Matrix , typename... TArgs>
  std::unique_ptr< Matrix > initialize (size_type stride, std::initializer_list< typename Matrix::value_type > vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)

  Creates and initializes a column-vector.
- template<typename Matrix , typename... TArgs>
  std::unique_ptr< Matrix > initialize (std::initializer_list< typename Matrix::value_type > vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)

  Creates and initializes a column-vector.
- template<typename Matrix , typename... TArgs>
  std::unique_ptr< Matrix > initialize (size_type stride, std::initializer_list< std::initializer_list< typename Matrix::value_type >> vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)

  Creates and initializes a matrix.

- template<typename Matrix , typename... TArgs>
  std::unique_ptr< Matrix > initialize (std::initializer_list< std::initializer_list< typename Matrix::value_type >> vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)

  Creates and initializes a matrix.
- bool operator== (const stopping_status &x, const stopping_status &y) noexcept

  Checks if two stopping statuses are equivalent.
- bool operator!= (const stopping_status &x, const stopping_status &y) noexcept

  Checks if two stopping statuses are different.

Variables

- constexpr size_type byte_size = CHAR_BIT

  Number of bits in a byte.

7.1.1 Detailed Description

The Ginkgo namespace.

7.1.2 Typedef Documentation

7.1.2.1 is_complex_s

```
template<typename T >
using gko::is_complex_s = typedef detail::is_complex_impl<T>
```

Allows to check if T is a complex value during compile time by accessing the `value` attribute of this struct.

If `value` is `true`, T is a complex type, if it is `false`, T is not a complex type.

Template Parameters

T	type to check

7.1.3 Enumeration Type Documentation

7.1.3.1 layout_type

```
enum gko::layout_type  [strong]
```

Specifies the layout type when writing data in matrix market format.

Enumerator

array	The matrix should be written as dense matrix in column-major order.
coordinate	The matrix should be written as a sparse matrix in coordinate format.

```
67                          {
71     array,
75     coordinate
76 };
```

7.1.4 Function Documentation

7.1.4.1 abs()

```
template<typename T >
constexpr T gko::abs (
             const T & x )  [inline], [constexpr]
```

Returns the absolute value of the object.

Template Parameters

T	the type of the object

Parameters

x	the object

Returns

x >= zero<T>() ? x : -x;

```
362 {
363     return x >= zero<T>() ? x : -x;
364 }
```

7.1.4.2 as() [1/2]

```
template<typename T , typename U >
const std::decay<T>::type* gko::as (
             const U * obj )  [inline]
```

Performs polymorphic type conversion.

This is the constant version of the function.

Template Parameters

T	requested result type
U	static type of the passed object

Parameters

obj	the object which should be converted

Returns

If successful, returns a pointer to the subtype, otherwise throws NotSupported.

```
309 {
310     if (auto p = dynamic_cast<const typename std::decay<T>::type *>(obj)) {
311         return p;
312     } else {
313         throw NotSupported(__FILE__, __LINE__, __func__, typeid(obj).name());
314     }
315 }
```

7.1.4.3 as() [2/2]

```
template<typename T , typename U >
std::decay<T>::type* gko::as (
            U * obj )  [inline]
```

Performs polymorphic type conversion.

Template Parameters

T	requested result type
U	static type of the passed object

Parameters

obj	the object which should be converted

Returns

If successful, returns a pointer to the subtype, otherwise throws NotSupported.

7.1.4.4 ceildiv()

```
constexpr int64 gko::ceildiv (
            int64 num,
            int64 den )  [inline], [constexpr]
```

Performs integer division with rounding up.

Parameters

num	numerator
den	denominator

Returns

returns the ceiled quotient.

Referenced by gko::preconditioner::block_interleaved_storage_scheme< index_type >::compute_storage_↩space().

7.1.4.5 clone() [1/2]

```
template<typename Pointer >
detail::cloned_type<Pointer> gko::clone (
            const Pointer & p )  [inline]
```

Creates a unique clone of the object pointed to by p.

The pointee (i.e. *p) needs to have a clone method that returns a std::unique_ptr in order for this method to work.

Template Parameters

Pointer	type of pointer to the object (plain or smart pointer)

Parameters

p	a pointer to the object

Note

The difference between this function and directly calling LinOp::clone() is that this one preserves the static type of the object.

Referenced by gko::temporary_clone< T >::temporary_clone().

7.1.4.6 clone() [2/2]

```
template<typename Pointer >
detail::cloned_type<Pointer> gko::clone (
            std::shared_ptr< const Executor > exec,
            const Pointer & p )  [inline]
```

Creates a unique clone of the object pointed to by p on Executor exec.

The pointee (i.e. *p) needs to have a clone method that takes an executor and returns a std::unique_ptr in order for this method to work.

Template Parameters

Pointer	type of pointer to the object (plain or smart pointer)

Parameters

exec	the executor where the cloned object should be stored
p	a pointer to the object

Note

The difference between this function and directly calling LinOp::clone() is that this one preserves the static type of the object.

7.1.4.7 conj()

```
template<typename T >
T gko::conj (
             const T & x )  [inline]
```

Returns the conjugate of an object.

Parameters

x	the number to conjugate

Returns

conjugate of the object (by default, the object itself)

Referenced by squared_norm().

7.1.4.8 copy_and_convert_to() [1/4]

```
template<typename R , typename T >
std::unique_ptr<const R, std::function<void(const R *)> > gko::copy_and_convert_to (
             std::shared_ptr< const Executor > exec,
             const T * obj )
```

Converts the object to R and places it on Executor exec.

If the object is already of the requested type and on the requested executor, the copy and conversion is avoided and a reference to the original object is returned instead.

Template Parameters

R	the type to which the object should be converted
T	the type of the input object

Parameters

exec	the executor where the result should be placed
obj	the object that should be converted

Returns

a unique pointer (with dynamically bound deleter) to the converted object

Note

This is a version of the function which adds the const qualifier to the result if the input had the same qualifier.

```
483 {
484     return detail::copy_and_convert_to_impl<const R>(std::move(exec), obj);
485 }
```

7.1.4.9 copy_and_convert_to() [2/4]

```
template<typename R , typename T >
std::shared_ptr<const R> gko::copy_and_convert_to (
            std::shared_ptr< const Executor > exec,
            std::shared_ptr< const T > obj )
```

This is the version that takes in the std::shared_ptr and returns a std::shared_ptr

If the object is already of the requested type and on the requested executor, the copy and conversion is avoided and a reference to the original object is returned instead.

Template Parameters

R	the type to which the object should be converted
T	the type of the input object

Parameters

exec	the executor where the result should be placed
obj	the object that should be converted

Returns

a shared pointer to the converted object

Note

This is a version of the function which adds the const qualifier to the result if the input had the same qualifier.

7.1.4.10 copy_and_convert_to() [3/4]

```
template<typename R , typename T >
std::shared_ptr<R> gko::copy_and_convert_to (
            std::shared_ptr< const Executor > exec,
            std::shared_ptr< T > obj )
```

Converts the object to R and places it on Executor exec.

This is the version that takes in the std::shared_ptr and returns a std::shared_ptr

If the object is already of the requested type and on the requested executor, the copy and conversion is avoided and a reference to the original object is returned instead.

Template Parameters

R	the type to which the object should be converted
T	the type of the input object

Parameters

exec	the executor where the result should be placed
obj	the object that should be converted

Returns

a shared pointer to the converted object

7.1.4.11 copy_and_convert_to() [4/4]

```
template<typename R , typename T >
std::unique_ptr<R, std::function<void(R *)> > gko::copy_and_convert_to (
            std::shared_ptr< const Executor > exec,
            T * obj )
```

Converts the object to R and places it on Executor exec.

If the object is already of the requested type and on the requested executor, the copy and conversion is avoided and a reference to the original object is returned instead.

Template Parameters

R	the type to which the object should be converted
T	the type of the input object

Parameters

exec	the executor where the result should be placed
obj	the object that should be converted

Returns

a unique pointer (with dynamically bound deleter) to the converted object

7.1.4.12 get_significant_bit()

```
template<typename T >
constexpr uint32 gko::get_significant_bit (
             const T & n,
             uint32 hint = 0u )  [inline], [constexpr], [noexcept]
```

Returns the position of the most significant bit of the number.

This is the same as the rounded down base-2 logarithm of the number.

Template Parameters

T	a numeric type supporting bit shift and comparison

Parameters

n	a number
hint	a lower bound for the position o the significant bit

Returns

maximum of `hint` and the significant bit position of `n`

7.1.4.13 get_superior_power()

```
template<typename T >
constexpr T gko::get_superior_power (
             const T & base,
             const T & limit,
             const T & hint = T{1} )  [inline], [constexpr], [noexcept]
```

Returns the smallest power of `base` not smaller than `limit`.

Template Parameters

T	a numeric type supporting multiplication and comparison

Parameters

base	the base of the power to be returned
limit	the lower limit on the size of the power returned
hint	a lower bound on the result, has to be a power of base

Returns

the smallest power of `base` not smaller than `limit`

7.1.4.14 give()

```
template<typename OwningPointer >
std::remove_reference<OwningPointer>::type&& gko::give (
            OwningPointer && p )  [inline]
```

Marks that the object pointed to by `p` can be given to the callee.

Effectively calls `std::move(p)`.

Template Parameters

OwningPointer	type of pointer with ownership to the object (has to be a smart pointer)

Parameters

p	a pointer to the object

Note

The original pointer `p` becomes invalid after this call.

7.1.4.15 imag()

```
template<typename T >
constexpr T gko::imag (
            const T &  )  [inline], [constexpr]
```

Returns the imaginary part of the object.

Template Parameters

T	type of the object

Parameters

x	the object

Returns

imaginary part of the object (by default, zero<T>())

7.1.4.16 is_complex()

```
template<typename T >
constexpr bool gko::is_complex ( )  [inline], [constexpr]
```

Checks if T is a complex type.

Template Parameters

T	type to check

Returns

`true` if T is a complex type, `false` otherwise

7.1.4.17 isfinite()

```
template<typename T >
xstd::enable_if_t<is_complex_s<T>::value, bool> gko::isfinite (
            const T & value )  [inline]
```

Checks if all components of a complex value are finite, meaning they are neither +/- infinity nor NaN.

Template Parameters

T	complex type of the value to check

Parameters

value	complex value to check

returns `true` if both components of the given value are finite, meaning they are neither +/- infinity nor NaN.

7.1.4.18 lend() [1/2]

```
template<typename Pointer >
std::enable_if<detail::have_ownership<Pointer>), detail::pointee<Pointer> *>::type gko::lend
(
            const Pointer & p )  [inline]
```

Returns a non-owning (plain) pointer to the object pointed to by `p`.

Template Parameters

Pointer	type of pointer to the object (plain or smart pointer)

Parameters

p	a pointer to the object

Note

This is the overload for owning (smart) pointers, that behaves the same as calling .get() on the smart pointer.

Referenced by gko::log::EnableLogging< Executor >::remove_logger().

7.1.4.19 lend() [2/2]

```
template<typename Pointer >
std::enable_if<!detail::have_ownership<Pointer>), detail::pointee<Pointer> *>::type gko↩
::lend (
            const Pointer & p )  [inline]
```

Returns a non-owning (plain) pointer to the object pointed to by `p`.

Template Parameters

Pointer	type of pointer to the object (plain or smart pointer)

Parameters

p	a pointer to the object

Note

This is the overload for non-owning (plain) pointers, that just returns `p`.

7.1.4.20 make_temporary_clone()

```
template<typename T >
temporary_clone<T> gko::make_temporary_clone (
            std::shared_ptr< const Executor > exec,
            T * ptr )
```

Creates a temporary_clone.

This is a helper function which avoids the need to explicitly specify the type of the object, as would be the case if using the constructor of temporary_clone.

Parameters

exec	the executor where the clone will be created
ptr	a pointer to the object of which the clone will be created

Referenced by gko::matrix::Dense< ValueType >::add_scaled(), gko::matrix::Coo< ValueType, IndexType >::apply2(), gko::matrix::Dense< ValueType >::compute_dot(), gko::matrix::Dense< ValueType >::compute_norm2(), and gko::matrix::Dense< ValueType >::scale().

7.1.4.21 max()

```
template<typename T >
constexpr T gko::max (
            const T & x,
            const T & y )  [inline], [constexpr]
```

Returns the larger of the arguments.

Template Parameters

T	type of the arguments

Parameters

x	first argument
y	second argument

Returns

x >= y ? x : y

Note

C++11 version of this function is not constexpr, thus we provide our own implementation.

7.1.4.22 min()

```
template<typename T >
constexpr T gko::min (
            const T & x,
            const T & y )  [inline], [constexpr]
```

Returns the smaller of the arguments.

Template Parameters

T	type of the arguments

Parameters

x	first argument
y	second argument

Returns

x <= y ? x : y

Note

C++11 version of this function is not `constexpr`, thus we provide our own implementation.

7.1.4.23 one() [1/2]

```
template<typename T >
constexpr T gko::one ( )  [inline], [constexpr]
```

Returns the multiplicative identity for T.

Returns

the multiplicative identity for T

7.1.4.24 one() [2/2]

```
template<typename T >
constexpr T gko::one (
            const T &  )  [inline], [constexpr]
```

Returns the multiplicative identity for T.

Returns

the multiplicative identity for T

Note

This version takes an unused reference argument to avoid complicated calls like `one<decltype(x)>()`. Instead, it allows `one(x)`.

7.1.4.25 operator"!=() [1/3]

```
template<size_type Dimensionality, typename DimensionType >
constexpr bool gko::operator!= (
            const dim< Dimensionality, DimensionType > & x,
            const dim< Dimensionality, DimensionType > & y )  [inline], [constexpr]
```

Checks if two dim objects are different.

Template Parameters

Dimensionality	number of dimensions of the dim objects
DimensionType	datatype used to represent each dimension

Parameters

x	first object
y	second object

Returns

`!(x == y)`

```
219 {
220     return !(x == y);
221 }
```

7.1.4.26 operator"!=() [2/3]

```
bool gko::operator!= (
            const stopping_status & x,
            const stopping_status & y )  [inline], [noexcept]
```

Checks if two stopping statuses are different.

Parameters

x	a stopping status
y	a stopping status

Returns

true if and only if `!(x == y)`

```
179 {
180     return x.data_ != y.data_;
181 }
```

7.1.4.27 operator"!=() [3/3]

```
constexpr bool gko::operator!= (
            precision_reduction x,
            precision_reduction y )  [constexpr], [noexcept]
```

Checks if two precision_reduction encodings are different.

Parameters

x	an encoding
y	an encoding

Returns

true if and only if x and y are different encodings.

```
368 {
369     using st = precision_reduction::storage_type;
370     return static_cast<st>(x) != static_cast<st>(y);
371 }
```

7.1.4.28 operator<<() [1/2]

```
std::ostream& gko::operator<< (
            std::ostream & os,
            const version & ver )  [inline]
```

Prints version information to a stream.

Parameters

os	output stream
ver	version structure

Returns

os

```
115 {
116     os « ver.major « "." « ver.minor « "." « ver.patch;
117     if (ver.tag) {
118         os « " (" « ver.tag « ")";
119     }
120     return os;
121 }
```

References gko::version::major, gko::version::minor, gko::version::patch, and gko::version::tag.

7.1.4.29 operator<<() [2/2]

```
std::ostream& gko::operator<< (
            std::ostream & os,
            const version_info & ver_info )
```

Prints library version information in human-readable format to a stream.

Parameters

os	output stream
ver_info	version information

Returns

os

7.1.4.30 operator==() [1/2]

```
bool gko::operator== (
            const stopping_status & x,
            const stopping_status & y )  [inline], [noexcept]
```

Checks if two stopping statuses are equivalent.

Parameters

x	a stopping status
y	a stopping status

Returns

true if and only if both x and y have the same mask and converged and finalized state

7.1.4.31 operator==() [2/2]

```
constexpr bool gko::operator== (
            precision_reduction x,
            precision_reduction y )  [constexpr], [noexcept]
```

Checks if two precision_reduction encodings are equal.

Parameters

x	an encoding
y	an encoding

Returns

true if and only if x and y are the same encodings

7.1.4.32 read()

```
template<typename MatrixType , typename StreamType , typename...  MatrixArgs>
std::unique_ptr<MatrixType> gko::read (
            StreamType && is,
            MatrixArgs &&...  args )  [inline]
```

Reads a matrix stored in matrix market format from an input stream.

Template Parameters

MatrixType	a ReadableFromMatrixData LinOp type used to store the matrix once it's been read from disk.
StreamType	type of stream used to write the data to
MatrixArgs	additional argument types passed to MatrixType constructor

Parameters

is	input stream from which to read the data
args	additional arguments passed to MatrixType constructor

Returns

A MatrixType LinOp filled with data from filename

References read_raw().

7.1.4.33 read_raw()

```
template<typename ValueType = default_precision, typename IndexType = int32>
matrix_data<ValueType, IndexType> gko::read_raw (
            std::istream & is )
```

Reads a matrix stored in matrix market format from an input stream.

Template Parameters

ValueType	type of matrix values
IndexType	type of matrix indexes

Parameters

is	input stream from which to read the data

Returns

A matrix_data structure containing the matrix. The nonzero elements are sorted in lexicographic order of their (row, colum) indexes.

Note

This is an advanced routine that will return the raw matrix data structure. Consider using gko::read instead.

Referenced by read().

7.1.4.34 real()

```
template<typename T >
constexpr T gko::real (
            const T & x )  [inline], [constexpr]
```

Returns the real part of the object.

Template Parameters

T	type of the object

Parameters

x	the object

Returns

real part of the object (by default, the object itself)

Referenced by squared_norm().

7.1.4.35 round_down()

```
template<typename T >
constexpr reduce_precision<T> gko::round_down (
            T val )  [inline], [constexpr]
```

Reduces the precision of the input parameter.

Template Parameters

T	the original precision

Parameters

val	the value to round down

Returns

the rounded down value

7.1.4.36 round_up()

```
template<typename T >
constexpr increase_precision<T> gko::round_up (
            T val )  [inline], [constexpr]
```

Increases the precision of the input parameter.

Template Parameters

T	the original precision

Parameters

val	the value to round up

Returns

the rounded up value

7.1.4.37 share()

```
template<typename OwningPointer >
detail::shared_type<OwningPointer> gko::share (
            OwningPointer && p )  [inline]
```

Marks the object pointed to by `p` as shared.

Effectively converts a pointer with ownership to std::shared_ptr.

Template Parameters

OwningPointer	type of pointer with ownership to the object (has to be a smart pointer)

Parameters

p	a pointer to the object

Note

The original pointer `p` becomes invalid after this call.

7.1.4.38 squared_norm()

```
template<typename T >
constexpr auto gko::squared_norm (
            const T & x ) -> decltype(real(conj(x) * x))   [inline], [constexpr]
```

Returns the squared norm of the object.

Template Parameters

T	type of the object.

Returns

The squared norm of the object.

References conj(), and real().

7.1.4.39 transpose()

```
template<typename DimensionType >
constexpr dim<2, DimensionType> gko::transpose (
            const dim< 2, DimensionType > & dimensions )  [inline], [constexpr], [noexcept]
```

Returns a dim<2> object with its dimensions swapped.

Template Parameters

DimensionType	datatype used to represent each dimension

Parameters

dimensions	original object

Returns

a dim<2> object with its dimensions swapped

7.1.4.40 write()

```
template<typename MatrixType , typename StreamType >
void gko::write (
             StreamType && os,
             MatrixType * matrix,
             layout_type layout = layout_type::array )  [inline]
```

Reads a matrix stored in matrix market format from an input stream.

Template Parameters

MatrixType	a ReadableFromMatrixData LinOp type used to store the matrix once it's been read from disk.
StreamType	type of stream used to write the data to

Parameters

os	output stream where the data is to be written
matrix	the matrix to write
layout	the layout used in the output

References write_raw().

7.1.4.41 write_raw()

```
template<typename ValueType , typename IndexType >
void gko::write_raw (
             std::ostream & os,
             const matrix_data< ValueType, IndexType > & data,
             layout_type layout = layout_type::array )
```

Writes a matrix_data structure to a stream in matrix market format.

Template Parameters

ValueType	type of matrix values
IndexType	type of matrix indexes

Parameters

os	output stream where the data is to be written
data	the matrix data to write
layout	the layout used in the output

Note

This is an advanced routine that writes the raw matrix data structure. If you are trying to write an existing matrix, consider using gko::write instead.

Referenced by write().

7.1.4.42 zero() [1/2]

```
template<typename T >
constexpr T gko::zero ( )  [inline], [constexpr]
```

Returns the additive identity for T.

Returns

additive identity for T

7.1.4.43 zero() [2/2]

```
template<typename T >
constexpr T gko::zero (
            const T &  )  [inline], [constexpr]
```

Returns the additive identity for T.

Returns

additive identity for T

Note

This version takes an unused reference argument to avoid complicated calls like `zero<decltype(x)>()`. Instead, it allows `zero(x)`.

7.2 gko::accessor Namespace Reference

The accessor namespace.

Classes

- class row_major

  A row_major accessor is a bridge between a range and the row-major memory layout.

7.2.1 Detailed Description

The accessor namespace.

7.3 gko::factorization Namespace Reference

The Factorization namespace.

Classes

- class Parllu

  ParILU is an incomplete LU factorization which is computed in parallel.

7.3.1 Detailed Description

The Factorization namespace.

7.4 gko::log Namespace Reference

The logger namespace .

Classes

- class Convergence

  Convergence is a Logger which logs data strictly from the `criterion_check_completed` *event.*
- struct criterion_data

  Struct representing Criterion related data.
- class EnableLogging

  EnableLogging is a mixin which should be inherited by any class which wants to enable logging.
- struct executor_data

  Struct representing Executor related data.
- struct iteration_complete_data

  Struct representing iteration complete related data.
- struct linop_data

  Struct representing LinOp related data.
- struct linop_factory_data

  Struct representing LinOp factory related data.
- class Loggable

  Loggable class is an interface which should be implemented by classes wanting to support logging.
- struct operation_data

  Struct representing Operator related data.
- struct polymorphic_object_data

  Struct representing PolymorphicObject related data.
- class Record

  Record is a Logger which logs every event to an object.
- class Stream

  Stream is a Logger which logs every event to a stream.

7.4.1 Detailed Description

The logger namespace .

The Logging namespace.

Logging

7.5 gko::matrix Namespace Reference

The matrix namespace.

Classes

- class Coo

  COO stores a matrix in the coordinate matrix format.
- class Csr

  CSR is a matrix format which stores only the nonzero coefficients by compressing each row of the matrix (compressed sparse row format).
- class Dense

  Dense is a matrix format which explicitly stores all values of the matrix.
- class Ell

  ELL is a matrix format where stride with explicit zeros is used such that all rows have the same number of stored elements.
- class Hybrid

  HYBRID is a matrix format which splits the matrix into ELLPACK and COO format.
- class Identity

  This class is a utility which efficiently implements the identity matrix (a linear operator which maps each vector to itself).
- class IdentityFactory

  This factory is a utility which can be used to generate Identity operators.
- class Sellp

  SELL-P is a matrix format similar to ELL format.
- class SparsityCsr

  SparsityCsr is a matrix format which stores only the sparsity pattern of a sparse matrix by compressing each row of the matrix (compressed sparse row format).

7.5.1 Detailed Description

The matrix namespace.

7.6 gko::name_demangling Namespace Reference

The name demangling namespace.

Functions

- template<typename T >
  std::string get_static_type (const T &)

  This function uses name demangling facilities to get the name of the static type (`T`) of the object passed in arguments.
- template<typename T >
  std::string get_dynamic_type (const T &t)

  This function uses name demangling facilities to get the name of the dynamic type of the object passed in arguments.

7.6.1 Detailed Description

The name demangling namespace.

7.6.2 Function Documentation

7.6.2.1 get_dynamic_type()

```
template<typename T >
std::string gko::name_demangling::get_dynamic_type (
            const T & t )
```

This function uses name demangling facilities to get the name of the dynamic type of the object passed in arguments.

Template Parameters

T	the type of the object to demangle

Parameters

t	the object we get the dynamic type of

```
100 {
101     return get_type_name(typeid(t));
102 }
```

7.6.2.2 get_static_type()

```
template<typename T >
std::string gko::name_demangling::get_static_type (
            const T &  )
```

This function uses name demangling facilities to get the name of the static type (`T`) of the object passed in arguments.

Template Parameters

T	the type of the object to demangle

Parameters

unused	

7.7 gko::preconditioner Namespace Reference

The Preconditioner namespace.

Classes

- struct block_interleaved_storage_scheme

  Defines the parameters of the interleaved block storage scheme used by block-Jacobi blocks.
- class Ilu

  The Incomplete LU (ILU) preconditioner solves the equation $LUx = b$ for a given lower triangular matrix L, an upper triangular matrix U and the right hand side b (can contain multiple right hand sides).
- class Jacobi

  A block-Jacobi preconditioner is a block-diagonal linear operator, obtained by inverting the diagonal blocks of the source operator.

7.7.1 Detailed Description

The Preconditioner namespace.

7.8 gko::solver Namespace Reference

The ginkgo Solve namespace.

Classes

- class Bicgstab

  BiCGSTAB or the Bi-Conjugate Gradient-Stabilized is a Krylov subspace solver.
- class Cg

  CG or the conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.
- class Cgs

  CGS or the conjugate gradient square method is an iterative type Krylov subspace method which is suitable for general systems.
- class Fcg

  FCG or the flexible conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.

- class Gmres

  GMRES or the generalized minimal residual method is an iterative type Krylov subspace method which is suitable for nonsymmetric linear systems.
- class Ir

  Iterative refinement (IR) is an iterative method that uses another coarse method to approximate the error of the current solution via the current residual.
- class LowerTrs

  LowerTrs is the triangular solver which solves the system L x = b, when L is a lower triangular matrix.
- class UpperTrs

  UpperTrs is the triangular solver which solves the system U x = b, when U is an upper triangular matrix.

7.8.1 Detailed Description

The ginkgo Solve namespace.

7.9 gko::stop Namespace Reference

The Stopping criterion namespace.

Classes

- class Combined

  The Combined class is used to combine multiple criterions together through an OR operation.
- class Criterion

  The Criterion class is a base class for all stopping criteria.
- struct CriterionArgs

  This struct is used to pass parameters to the EnableDefaultCriterionFactoryCriterionFactory::generate() method.
- class Iteration

  The Iteration class is a stopping criterion which stops the iteration process after a preset number of iterations.
- class ResidualNormReduction

  The ResidualNormReduction class is a stopping criterion which stops the iteration process when the relative residual norm is below a certain threshold.
- class Time

  The Time class is a stopping criterion which stops the iteration process after a certain amout of time has passed.

Typedefs

- using CriterionFactory = AbstractFactory< Criterion, CriterionArgs >

  Declares an Abstract Factory specialized for Criterions.
- template<typename ConcreteFactory , typename ConcreteCriterion , typename ParametersType , typename PolymorphicBase = CriterionFactory>
  using EnableDefaultCriterionFactory = EnableDefaultFactory< ConcreteFactory, ConcreteCriterion, ParametersType, PolymorphicBase >

  This is an alias for the EnableDefaultFactory mixin, which correctly sets the template parameters to enable a subclass of CriterionFactory.

Functions

- template<typename FactoryContainer >
  std::shared_ptr< const CriterionFactory > combine (FactoryContainer &&factories)

  Combines multiple criterion factories into a single combined criterion factory.

7.9.1 Detailed Description

The Stopping criterion namespace.

Stopping criteria

7.9.2 Typedef Documentation

7.9.2.1 EnableDefaultCriterionFactory

```
template<typename ConcreteFactory , typename ConcreteCriterion , typename ParametersType ,
typename PolymorphicBase = CriterionFactory>
using gko::stop::EnableDefaultCriterionFactory = typedef EnableDefaultFactory<ConcreteFactory,
ConcreteCriterion, ParametersType, PolymorphicBase>
```

This is an alias for the EnableDefaultFactory mixin, which correctly sets the template parameters to enable a subclass of CriterionFactory.

Template Parameters

ConcreteFactory	the concrete factory which is being implemented [CRTP parmeter]
ConcreteCriterion	the concrete Criterion type which this factory produces, needs to have a constructor which takes a const ConcreteFactory *, and a const CriterionArgs * as parameters.
ParametersType	a subclass of enable_parameters_type template which defines all of the parameters of the factory
PolymorphicBase	parent of ConcreteFactory in the polymorphic hierarchy, has to be a subclass of CriterionFactory

7.10 gko::syn Namespace Reference

The Synthesizer namespace.

7.10.1 Detailed Description

The Synthesizer namespace.

7.11 gko::xstd Namespace Reference

The namespace for functionalities after C++11 standard.

7.11.1 Detailed Description

The namespace for functionalities after C++11 standard.

Chapter 8

Class Documentation

8.1 gko::AbstractFactory< AbstractProductType, ComponentsType > Class Template Reference

The AbstractFactory is a generic interface template that enables easy implementation of the abstract factory design pattern.

`#include <ginkgo/core/base/abstract_factory.hpp>`

Public Member Functions

- template<typename... Args>
  std::unique_ptr< AbstractProductType > generate (Args &&... args) const

  Creates a new product from the given components.

8.1.1 Detailed Description

template<typename AbstractProductType, typename ComponentsType>
class gko::AbstractFactory< AbstractProductType, ComponentsType >

The AbstractFactory is a generic interface template that enables easy implementation of the abstract factory design pattern.

The interface provides the AbstractFactory::generate() method that can produce products of type `AbstractProductType` using an object of `ComponentsType` (which can be constructed on the fly from parameters to its constructors). The generate() method is not declared as virtual, as this allows subclasses to hide the method with a variant that preserves the compile-time type of the objects. Instead, implementers should override the generate_impl() method, which is declared virtual.

Implementers of concrete factories should consider using the EnableDefaultFactory mixin to obtain default implementations of utility methods of PolymorphicObject and AbstractFactory.

Template Parameters

AbstractProductType	the type of products the factory produces
ComponentsType	the type of components the factory needs to produce the product

8.1.2 Member Function Documentation

8.1.2.1 generate()

```
template<typename AbstractProductType, typename ComponentsType>
template<typename... Args>
std::unique_ptr<AbstractProductType> gko::AbstractFactory< AbstractProductType, Components↩
Type >::generate (
            Args &&... args ) const [inline]
```

Creates a new product from the given components.

The method will create an ComponentsType object from the arguments of this method, and pass it to the generate↩ _impl() function which will create a new AbstractProductType.

Template Parameters

Args	types of arguments passed to the constructor of ComponentsType

Parameters

args	arguments passed to the constructor of ComponentsType

Returns

an instance of AbstractProductType

```
93     {
94         auto product = this->generate_impl({std::forward<Args>(args)...});
95         for (auto logger : this->loggers_) {
96             product->add_logger(logger);
97         }
98         return product;
99     }
```

The documentation for this class was generated from the following file:

- ginkgo/core/base/abstract_factory.hpp (8045ac75)

8.2 gko::AllocationError Class Reference

AllocationError is thrown if a memory allocation fails.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- AllocationError (const std::string &file, int line, const std::string &device, size_type bytes)

  Initializes an allocation error.

8.2.1 Detailed Description

AllocationError is thrown if a memory allocation fails.

8.2.2 Constructor & Destructor Documentation

8.2.2.1 AllocationError()

```
gko::AllocationError::AllocationError (
            const std::string & file,
            int line,
            const std::string & device,
            size_type bytes )  [inline]
```

Initializes an allocation error.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
device	The device on which the error occurred
bytes	The size of the memory block whose allocation failed.

```
323         : Error(file, line,
324                 device + ": failed to allocate memory block of " +
325                     std::to_string(bytes) + "B")
326     {}
```

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.3 gko::Array< ValueType > Class Template Reference

An Array is a container which encapsulates fixed-sized arrays, stored on the Executor tied to the Array.

```
#include <ginkgo/core/base/array.hpp>
```

Public Types

- using value_type = ValueType

  The type of elements stored in the array.
- using default_deleter = executor_deleter< value_type[ ]>

  The default deleter type used by Array.
- using view_deleter = null_deleter< value_type[ ]>

  The deleter type used for views.

Public Member Functions

- Array () noexcept

  Creates an empty Array not tied to any executor.
- Array (std::shared_ptr< const Executor > exec) noexcept

  Creates an empty Array tied to the specified Executor.
- Array (std::shared_ptr< const Executor > exec, size_type num_elems)

  Creates an Array on the specified Executor.
- template<typename DeleterType >
  Array (std::shared_ptr< const Executor > exec, size_type num_elems, value_type *data, DeleterType deleter)

  Creates an Array from existing memory.
- Array (std::shared_ptr< const Executor > exec, size_type num_elems, value_type *data)

  Creates an Array from existing memory.
- template<typename RandomAccessIterator >
  Array (std::shared_ptr< const Executor > exec, RandomAccessIterator begin, RandomAccessIterator end)

  Creates an array on the specified Executor and initializes it with values.
- template<typename T >
  Array (std::shared_ptr< const Executor > exec, std::initializer_list< T > init_list)

  Creates an array on the specified Executor and initializes it with values.
- Array (std::shared_ptr< const Executor > exec, const Array &other)

  Creates a copy of another array on a different executor.
- Array (const Array &other)

  Creates a copy of another array.
- Array (std::shared_ptr< const Executor > exec, Array &&other)

  Moves another array to a different executor.
- Array (Array &&other)

  Moves another array.
- Array & operator= (const Array &other)

  Copies data from another array.
- Array & operator= (Array &&other)

  Moves data from another array.
- void clear () noexcept

  Deallocates all data used by the Array.
- void resize_and_reset (size_type num_elems)

  Resizes the array so it is able to hold the specified number of elements.
- size_type get_num_elems () const noexcept

  Returns the number of elements in the Array.
- value_type * get_data () noexcept

  Returns a pointer to the block of memory used to store the elements of the Array.
- const value_type * get_const_data () const noexcept

  Returns a constant pointer to the block of memory used to store the elements of the Array.
- std::shared_ptr< const Executor > get_executor () const noexcept

  Returns the Executor associated with the array.
- void set_executor (std::shared_ptr< const Executor > exec)

  Changes the Executor of the Array, moving the allocated data to the new Executor.

Static Public Member Functions

- static Array view (std::shared_ptr< const Executor > exec, size_type num_elems, value_type *data)

  Creates an Array from existing memory.

8.3.1 Detailed Description

template<typename ValueType>
class gko::Array< ValueType >

An Array is a container which encapsulates fixed-sized arrays, stored on the Executor tied to the Array.

The array stores and transfers its data as **raw** memory, which means that the constructors of its elements are not called when constructing, copying or moving the Array. Thus, the Array class is most suitable for storing POD types.

Template Parameters

ValueType	the type of elements stored in the array

8.3.2 Constructor & Destructor Documentation

8.3.2.1 Array() [1/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array ( )  [inline], [noexcept]
```

Creates an empty Array not tied to any executor.

An array without an assigned executor can only be empty. Attempts to change its size (e.g. via the resize_and_reset method) will result in an exception. If such an array is used as the right hand side of an assignment or move assignment expression, the data of the target array will be cleared, but its executor will not be modified.

The executor can later be set by using the set_executor method. If an Array with no assigned executor is assigned or moved to, it will inherit the executor of the source Array.

```
94         : num_elems_(0),
95           data_(nullptr, default_deleter{nullptr}),
96           exec_(nullptr)
97     {}
```

8.3.2.2 Array() [2/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array (
            std::shared_ptr< const Executor > exec )  [inline], [noexcept]
```

Creates an empty Array tied to the specified Executor.

Parameters

exec	the Executor where the array data is allocated

8.3.2.3 Array() [3/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array (
            std::shared_ptr< const Executor > exec,
            size_type num_elems )  [inline]
```

Creates an Array on the specified Executor.

Parameters

exec	the Executor where the array data will be allocated
num_elems	the amount of memory (expressed as the number of `value_type` elements) allocated on the Executor

8.3.2.4 Array() [4/11]

```
template<typename ValueType>
template<typename DeleterType >
gko::Array< ValueType >::Array (
            std::shared_ptr< const Executor > exec,
            size_type num_elems,
            value_type * data,
            DeleterType deleter )  [inline]
```

Creates an Array from existing memory.

The memory will be managed by the array, and deallocated using the specified deleter (e.g. use std::default_delete for data allocated with new).

Template Parameters

DeleterType	type of the deleter

Parameters

exec	executor where `data` is located
num_elems	number of elements in `data`
data	chunk of memory used to create the array
deleter	the deleter used to free the memory

See also

Array::view() to create an array that does not deallocate memory

Array(std::shared_ptr<cont Executor>, size_type, value_type*) to deallocate the memory using Executor::free() method

8.3.2.5 Array() [5/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array (
            std::shared_ptr< const Executor > exec,
            size_type num_elems,
            value_type * data )  [inline]
```

Creates an Array from existing memory.

The memory will be managed by the array, and deallocated using the Executor::free method.

Parameters

exec	executor where `data` is located
num_elems	number of elements in `data`
data	chunk of memory used to create the array

8.3.2.6 Array() [6/11]

```
template<typename ValueType>
template<typename RandomAccessIterator >
gko::Array< ValueType >::Array (
            std::shared_ptr< const Executor > exec,
            RandomAccessIterator begin,
            RandomAccessIterator end )  [inline]
```

Creates an array on the specified Executor and initializes it with values.

Template Parameters

RandomAccessIterator	type of the iterators

Parameters

exec	the Executor where the array data will be allocated
begin	start of range of values
end	end of range of values

8.3.2.7 Array() [7/11]

```
template<typename ValueType>
template<typename T >
```

```
gko::Array< ValueType >::Array (
            std::shared_ptr< const Executor > exec,
            std::initializer_list< T > init_list )  [inline]
```

Creates an array on the specified Executor and initializes it with values.

Template Parameters

T	type of values used to initialize the array (T has to be implicitly convertible to value_type)

Parameters

exec	the Executor where the array data will be allocated
init_list	list of values used to initialize the Array

8.3.2.8 Array() [8/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array (
            std::shared_ptr< const Executor > exec,
            const Array< ValueType > & other )  [inline]
```

Creates a copy of another array on a different executor.

This does not invoke the constructors of the elements, instead they are copied as POD types.

Parameters

exec	the executor where the new array will be created
other	the Array to copy from

8.3.2.9 Array() [9/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array (
            const Array< ValueType > & other )  [inline]
```

Creates a copy of another array.

This does not invoke the constructors of the elements, instead they are copied as POD types.

Parameters

other	the Array to copy from

8.3.2.10 Array() [10/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array (
            std::shared_ptr< const Executor > exec,
            Array< ValueType > && other )  [inline]
```

Moves another array to a different executor.

This does not invoke the constructors of the elements, instead they are copied as POD types.

Parameters

exec	the executor where the new array will be moved
other	the Array to move

8.3.2.11 Array() [11/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array (
            Array< ValueType > && other )  [inline]
```

Moves another array.

This does not invoke the constructors of the elements, instead they are copied as POD types.

Parameters

other	the Array to move

8.3.3 Member Function Documentation

8.3.3.1 clear()

```
template<typename ValueType>
void gko::Array< ValueType >::clear ( )  [inline], [noexcept]
```

Deallocates all data used by the Array.

The array is left in a valid, but empty state, so the same array can be used to allocate new memory. Calls to Array::get_data() will return a `nullptr`.

8.3.3.2 get_const_data()

```
template<typename ValueType>
const value_type* gko::Array< ValueType >::get_const_data ( ) const  [inline], [noexcept]
```

Returns a constant pointer to the block of memory used to store the elements of the Array.

Returns

a constant pointer to the block of memory used to store the elements of the Array

Referenced by gko::matrix::Dense< ValueType >::at(), gko::preconditioner::Jacobi< ValueType, IndexType >::get_blocks(), gko::preconditioner::Jacobi< ValueType, IndexType >::get_conditioning(), gko::matrix::Sellp< ValueType, IndexType >::get_const_col_idxs(), gko::matrix::SparsityCsr< ValueType, IndexType >::get_const_col_idxs(), gko::matrix::Ell< ValueType, IndexType >::get_const_col_idxs(), gko::matrix::Coo< ValueType, IndexType >::get_const_col_idxs(), gko::matrix::Csr< ValueType, IndexType >::get_const_col_idxs(), gko::matrix::Coo< ValueType, IndexType >::get_const_row_idxs(), gko::matrix::SparsityCsr< ValueType, IndexType >::get_const_row_ptrs(), gko::matrix::Csr< ValueType, IndexType >::get_const_row_ptrs(), gko::matrix::Sellp< ValueType, IndexType >::get_const_slice_lengths(), gko::matrix::Sellp< ValueType, IndexType >::get_const_slice_sets(), gko::matrix::Csr< ValueType, IndexType >::get_const_srow(), gko::matrix::SparsityCsr< ValueType, IndexType >::get_const_value(), gko::matrix::Sellp< ValueType, IndexType >::get_const_values(), gko::matrix::Ell< ValueType, IndexType >::get_const_values(), gko::matrix::Coo< ValueType, IndexType >::get_const_values(), gko::matrix::Dense< ValueType >::get_const_values(), gko::matrix::Csr< ValueType, IndexType >::get_const_values(), gko::Array< index_type >::operator=(), gko::matrix::Ell< ValueType, IndexType >::val_at(), and gko::matrix::Sellp< ValueType, IndexType >::val_at().

8.3.3.3 get_data()

```
template<typename ValueType>
value_type* gko::Array< ValueType >::get_data ( )  [inline], [noexcept]
```

Returns a pointer to the block of memory used to store the elements of the Array.

Returns

a pointer to the block of memory used to store the elements of the Array

Referenced by gko::matrix::Dense< ValueType >::at(), gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit::compute_ell_num_stored_elements_per_row(), gko::matrix::Sellp< ValueType, IndexType >::get_col_idxs(), gko::matrix::SparsityCsr< ValueType, IndexType >::get_col_idxs(), gko::matrix::Ell< ValueType, IndexType >::get_col_idxs(), gko::matrix::Coo< ValueType, IndexType >::get_col_idxs(), gko::matrix::Csr< ValueType, IndexType >::get_col_idxs(), gko::matrix::Coo< ValueType, IndexType >::get_row_idxs(), gko::matrix::SparsityCsr< ValueType, IndexType >::get_row_ptrs(), gko::matrix::Csr< ValueType, IndexType >::get_row_ptrs(), gko::matrix::Sellp< ValueType, IndexType >::get_slice_lengths(), gko::matrix::Sellp< ValueType, IndexType >::get_slice_sets(), gko::matrix::Csr< ValueType, IndexType >::get_srow(), gko::matrix::SparsityCsr< ValueType, IndexType >::get_value(), gko::matrix::Sellp< ValueType, IndexType >::get_values(), gko::matrix::Ell< ValueType, IndexType >::get_values(), gko::matrix::Coo< ValueType, IndexType >::get_values(), gko::matrix::Dense< ValueType >::get_values(), gko::matrix::Csr< ValueType, IndexType >::get_values(), gko::Array< index_type >::operator=(), gko::matrix::Ell< ValueType, IndexType >::val_at(), and gko::matrix::Sellp< ValueType, IndexType >::val_at().

8.3.3.4 get_executor()

```
template<typename ValueType>
std::shared_ptr<const Executor> gko::Array< ValueType >::get_executor ( ) const  [inline],
[noexcept]
```

Returns the Executor associated with the array.

Returns

the Executor associated with the array

Referenced by gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_hybrid_config(), and gko::Array< index_type >::operator=().

8.3.3.5 get_num_elems()

```
template<typename ValueType>
size_type gko::Array< ValueType >::get_num_elems ( ) const  [inline], [noexcept]
```

Returns the number of elements in the Array.

Returns

the number of elements in the Array

Referenced by gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit::compute_ell_num_stored_elements_per_row(), gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit::compute_ell_num_stored_elements_per_row(), gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_hybrid_config(), gko::matrix::SparsityCsr< ValueType, IndexType >::get_num_nonzeros(), gko::matrix::Csr< ValueType, IndexType >::get_num_srow_elements(), gko::matrix::Ell< ValueType, IndexType >::get_num_stored_elements(), gko::matrix::Coo< ValueType, IndexType >::get_num_stored_elements(), gko::matrix::Sellp< ValueType, IndexType >::get_num_stored_elements(), gko::matrix::Dense< ValueType >::get_num_stored_elements(), gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_elements(), gko::matrix::Csr< ValueType, IndexType >::get_num_stored_elements(), and gko::Array< index_type >::operator=().

8.3.3.6 operator=() [1/2]

```
template<typename ValueType>
Array& gko::Array< ValueType >::operator= (
            Array< ValueType > && other )  [inline]
```

Moves data from another array.

This does not invoke the constructors of the elements, instead they are copied as POD types.

The executor of this is preserved. In case this does not have an assigned executor, it will inherit the executor of other.

Parameters

other	the Array to move data from

Returns

this

8.3.3.7 operator=() [2/2]

```
template<typename ValueType>
Array& gko::Array< ValueType >::operator= (
            const Array< ValueType > & other )  [inline]
```

Copies data from another array.

This does not invoke the constructors of the elements, instead they are copied as POD types.

The executor of this is preserved. In case this does not have an assigned executor, it will inherit the executor of other.

Parameters

other	the Array to copy from

Returns

this

8.3.3.8 resize_and_reset()

```
template<typename ValueType>
void gko::Array< ValueType >::resize_and_reset (
            size_type num_elems )  [inline]
```

Resizes the array so it is able to hold the specified number of elements.

All data stored in the array will be lost.

If the Array is not assigned an executor, an exception will be thrown.

Parameters

num_elems	the amount of memory (expressed as the number of `value_type` elements) allocated on the Executor

Referenced by gko::Array< index_type >::operator=().

8.3.3.9 set_executor()

```
template<typename ValueType>
void gko::Array< ValueType >::set_executor (
            std::shared_ptr< const Executor > exec )  [inline]
```

Changes the Executor of the Array, moving the allocated data to the new Executor.

Parameters

exec	the Executor where the data will be moved to

8.3.3.10 view()

```
template<typename ValueType>
static Array gko::Array< ValueType >::view (
            std::shared_ptr< const Executor > exec,
            size_type num_elems,
            value_type * data )  [inline], [static]
```

Creates an Array from existing memory.

The Array does not take ownership of the memory, and will not deallocate it once it goes out of scope.

Parameters

exec	executor where `data` is located
num_elems	number of elements in `data`
data	chunk of memory used to create the array

Returns

an Array constructed from `data`

Referenced by gko::matrix::Dense< ValueType >::create_submatrix().

The documentation for this class was generated from the following file:

- ginkgo/core/base/array.hpp (8045ac75)

8.4 gko::matrix::Hybrid< ValueType, IndexType >::automatic Class Reference

automatic is a stratgy_type which decides the number of stored elements per row of the ell part automatically.

`#include <ginkgo/core/matrix/hybrid.hpp>`

Public Member Functions

- automatic ()

  Creates an automatic strategy.
- size_type compute_ell_num_stored_elements_per_row (Array< size_type > *row_nnz) const override

  Computes the number of stored elements per row of the ell part.

8.4.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >::automatic

automatic is a stratgy_type which decides the number of stored elements per row of the ell part automatically.

8.4.2 Member Function Documentation

8.4.2.1 compute_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::automatic::compute_ell_num_stored_↩
elements_per_row (
            Array< size_type > * row_nnz ) const  [inline], [override], [virtual]
```

Computes the number of stored elements per row of the ell part.

Parameters

row_nnz	the number of nonzeros of each row

Returns

the number of stored elements per row of the ell part

Implements gko::matrix::Hybrid< ValueType, IndexType >::strategy_type.

References gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit::compute_ell_num_stored↩
_elements_per_row().

The documentation for this class was generated from the following file:

- ginkgo/core/matrix/hybrid.hpp (3e51a52b)

8.5 gko::BadDimension Class Reference

BadDimension is thrown if an operation is being applied to a LinOp with bad dimensions.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- BadDimension (const std::string &file, int line, const std::string &func, const std::string &op_name, size_type op_num_rows, size_type op_num_cols, const std::string &clarification)

  Initializes a bad dimension error.

8.5.1 Detailed Description

BadDimension is thrown if an operation is being applied to a LinOp with bad dimensions.

8.5.2 Constructor & Destructor Documentation

8.5.2.1 BadDimension()

```
gko::BadDimension::BadDimension (
            const std::string & file,
            int line,
            const std::string & func,
            const std::string & op_name,
            size_type op_num_rows,
            size_type op_num_cols,
            const std::string & clarification )  [inline]
```

Initializes a bad dimension error.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
func	The function name where the error occurred
op_name	The name of the operator
op_num_rows	The row dimension of the operator
op_num_cols	The column dimension of the operator
clarification	An additional message further describing the error

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.6 gko::solver::Bicgstab< ValueType > Class Template Reference

BiCGSTAB or the Bi-Conjugate Gradient-Stabilized is a Krylov subspace solver.

```
#include <ginkgo/core/solver/bicgstab.hpp>
```

Public Member Functions

- std::shared_ptr< const LinOp > get_system_matrix () const

  Gets the system operator (matrix) of the linear system.

8.6.1 Detailed Description

template<typename ValueType = default_precision>
class gko::solver::Bicgstab< ValueType >

BiCGSTAB or the Bi-Conjugate Gradient-Stabilized is a Krylov subspace solver.

Being a generic solver, it is capable of solving general matrices, including non-s.p.d matrices. Though, the memory and the computational requirement of the BiCGSTAB solver are higher than of its s.p.d solver counterpart, it has the capability to solve generic systems. It was developed by stabilizing the BiCG method.

Template Parameters

ValueType	precision of the elements of the system matrix.

8.6.2 Member Function Documentation

8.6.2.1 get_system_matrix()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Bicgstab< ValueType >::get_system_matrix ( ) const
[inline]
```

Gets the system operator (matrix) of the linear system.

Returns

the system operator (matrix)

```
90     {
91         return system_matrix_;
92     }
```

The documentation for this class was generated from the following file:

- ginkgo/core/solver/bicgstab.hpp (c380ba80)

8.7 gko::preconditioner::block_interleaved_storage_scheme< IndexType > Struct Template Reference

Defines the parameters of the interleaved block storage scheme used by block-Jacobi blocks.

```
#include <ginkgo/core/preconditioner/jacobi.hpp>
```

Public Member Functions

- IndexType get_group_size () const noexcept

  Returns the number of elements in the group.
- size_type compute_storage_space (size_type num_blocks) const noexcept

  Computes the storage space required for the requested number of blocks.
- IndexType get_group_offset (IndexType block_id) const noexcept

  Returns the offset of the group belonging to the block with the given ID.
- IndexType get_block_offset (IndexType block_id) const noexcept

  Returns the offset of the block with the given ID within its group.
- IndexType get_global_block_offset (IndexType block_id) const noexcept

  Returns the offset of the block with the given ID.
- IndexType get_stride () const noexcept

  Returns the stride between columns of the block.

Public Attributes

- IndexType block_offset

  The offset between consecutive blocks within the group.
- IndexType group_offset

  The offset between two block groups.
- uint32 group_power

  Then base 2 power of the group.

8.7.1 Detailed Description

template<typename IndexType>
struct gko::preconditioner::block_interleaved_storage_scheme< IndexType >

Defines the parameters of the interleaved block storage scheme used by block-Jacobi blocks.

Template Parameters

IndexType	type used for storing indices of the matrix

8.7.2 Member Function Documentation

8.7.2.1 compute_storage_space()

```
template<typename IndexType>
size_type gko::preconditioner::block_interleaved_storage_scheme< IndexType >::compute_storage↩
_space (
            size_type num_blocks ) const  [inline], [noexcept]
```

Computes the storage space required for the requested number of blocks.

Parameters

num_blocks	the total number of blocks that needs to be stored

Returns

the total memory (as the number of elements) that need to be allocated for the scheme

Note

To simplify using the method in situations where the number of blocks is not known, for a special input `size_type{} - 1` the method returns `0` to avoid overallocation of memory.

```
114     {
115         return (num_blocks + 1 == size_type{0})
116                    ? size_type{0}
117                    : ceildiv(num_blocks, this->get_group_size()) * group_offset;
118     }
```

8.7.2.2 get_block_offset()

```
template<typename IndexType>
IndexType gko::preconditioner::block_interleaved_storage_scheme< IndexType >::get_block_offset
(
              IndexType block_id ) const  [inline], [noexcept]
```

Returns the offset of the block with the given ID within its group.

Parameters

block_id	the ID of the block

Returns

the offset of the block with ID `block_id` within its group

Referenced by gko::preconditioner::block_interleaved_storage_scheme< index_type >::get_global_block_offset().

8.7.2.3 get_global_block_offset()

```
template<typename IndexType>
IndexType gko::preconditioner::block_interleaved_storage_scheme< IndexType >::get_global_
block_offset (
              IndexType block_id ) const  [inline], [noexcept]
```

Returns the offset of the block with the given ID.

Parameters

block_id	the ID of the block

Returns

the offset of the block with ID `block_id`

8.7.2.4 get_group_offset()

```
template<typename IndexType>
IndexType gko::preconditioner::block_interleaved_storage_scheme< IndexType >::get_group_offset
(
            IndexType block_id ) const  [inline], [noexcept]
```

Returns the offset of the group belonging to the block with the given ID.

Parameters

block_id	the ID of the block

Returns

the offset of the group belonging to block with ID `block_id`

Referenced by gko::preconditioner::block_interleaved_storage_scheme< index_type >::get_global_block_offset().

8.7.2.5 get_group_size()

```
template<typename IndexType>
IndexType gko::preconditioner::block_interleaved_storage_scheme< IndexType >::get_group_size (
) const  [inline], [noexcept]
```

Returns the number of elements in the group.

Returns

the number of elements in the group

Referenced by gko::preconditioner::block_interleaved_storage_scheme< index_type >::compute_storage_space(), and gko::preconditioner::block_interleaved_storage_scheme< index_type >::get_block_offset().

8.7.2.6 get_stride()

```
template<typename IndexType>
IndexType gko::preconditioner::block_interleaved_storage_scheme< IndexType >::get_stride ( )
const  [inline], [noexcept]
```

Returns the stride between columns of the block.

Returns

stride between columns of the block

8.7.3 Member Data Documentation

8.7.3.1 group_power

```
template<typename IndexType>
uint32 gko::preconditioner::block_interleaved_storage_scheme< IndexType >::group_power
```

Then base 2 power of the group.

I.e. the group contains `1 << group_power` elements.

Referenced by gko::preconditioner::block_interleaved_storage_scheme< index_type >::get_group_offset(), gko::preconditioner::block_interleaved_storage_scheme< index_type >::get_group_size(), and gko::preconditioner::block_interleaved_storage_scheme< index_type >::get_stride().

The documentation for this struct was generated from the following file:

- ginkgo/core/preconditioner/jacobi.hpp (9c2e5ae6)

8.8 gko::solver::Cg< ValueType > Class Template Reference

CG or the conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.

`#include <ginkgo/core/solver/cg.hpp>`

Public Member Functions

- std::shared_ptr< const LinOp > get_system_matrix () const

  Gets the system operator (matrix) of the linear system.

8.8.1 Detailed Description

template<typename ValueType = default_precision>
class gko::solver::Cg< ValueType >

CG or the conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.

Though this method performs very well for symmetric positive definite matrices, it is in general not suitable for general matrices.

The implementation in Ginkgo makes use of the merged kernel to make the best use of data locality. The inner operations in one iteration of CG are merged into 2 separate steps.

Template Parameters

ValueType	precision of matrix elements

8.8.2 Member Function Documentation

8.8.2.1 get_system_matrix()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Cg< ValueType >::get_system_matrix ( ) const  [inline]
```

Gets the system operator (matrix) of the linear system.

Returns

the system operator (matrix)

```
85     {
86         return system_matrix_;
87     }
```

The documentation for this class was generated from the following file:

- ginkgo/core/solver/cg.hpp (c380ba80)

8.9 gko::solver::Cgs< ValueType > Class Template Reference

CGS or the conjugate gradient square method is an iterative type Krylov subspace method which is suitable for general systems.

`#include <ginkgo/core/solver/cgs.hpp>`

Public Member Functions

- std::shared_ptr< const LinOp > get_system_matrix () const
  Gets the system operator (matrix) of the linear system.

8.9.1 Detailed Description

template<typename ValueType = default_precision>
class gko::solver::Cgs< ValueType >

CGS or the conjugate gradient square method is an iterative type Krylov subspace method which is suitable for general systems.

The implementation in Ginkgo makes use of the merged kernel to make the best use of data locality. The inner operations in one iteration of CGS are merged into 3 separate steps.

Template Parameters

ValueType	precision of matrix elements

8.9.2 Member Function Documentation

8.9.2.1 get_system_matrix()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Cgs< ValueType >::get_system_matrix ( ) const  [inline]
```

Gets the system operator (matrix) of the linear system.

Returns

the system operator (matrix)

```
82     {
83         return system_matrix_;
84     }
```

The documentation for this class was generated from the following file:

- ginkgo/core/solver/cgs.hpp (c380ba80)

8.10 gko::matrix::Hybrid< ValueType, IndexType >::column_limit Class Reference

column_limit is a strategy_type which decides the number of stored elements per row of the ell part by specifying the number of columns.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Public Member Functions

- column_limit (size_type num_column=0)

  Creates a column_limit strategy.
- size_type compute_ell_num_stored_elements_per_row (Array< size_type > *row_nnz) const override

  Computes the number of stored elements per row of the ell part.

8.10.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >::column_limit

column_limit is a strategy_type which decides the number of stored elements per row of the ell part by specifying the number of columns.

8.10.2 Constructor & Destructor Documentation

8.10.2.1 column_limit()

```
template<typename ValueType = default_precision, typename IndexType = int32>
gko::matrix::Hybrid< ValueType, IndexType >::column_limit::column_limit (
            size_type num_column = 0 )  [inline], [explicit]
```

Creates a column_limit strategy.

Parameters

num_column	the specified number of columns of the ell part

8.10.3 Member Function Documentation

8.10.3.1 compute_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::column_limit::compute_ell_num_stored_↩
elements_per_row (
            Array< size_type > * row_nnz ) const  [inline], [override], [virtual]
```

Computes the number of stored elements per row of the ell part.

Parameters

row_nnz	the number of nonzeros of each row

Returns

the number of stored elements per row of the ell part

Implements gko::matrix::Hybrid< ValueType, IndexType >::strategy_type.

The documentation for this class was generated from the following file:

- ginkgo/core/matrix/hybrid.hpp (3e51a52b)

8.11 gko::Combination< ValueType > Class Template Reference

The Combination class can be used to construct a linear combination of multiple linear operators `c1 * op1 + c2 * op2 + ...`

```
#include <ginkgo/core/base/combination.hpp>
```

Public Member Functions

- const std::vector< std::shared_ptr< const LinOp > > & get_coefficients () const noexcept

  Returns a list of coefficients of the combination.
- const std::vector< std::shared_ptr< const LinOp > > & get_operators () const noexcept

  Returns a list of operators of the combination.

8.11.1 Detailed Description

template<typename ValueType = default_precision>
class gko::Combination< ValueType >

The Combination class can be used to construct a linear combination of multiple linear operators `c1 * op1 + c2 * op2 + ...`

- ck * opk.

Template Parameters

ValueType	precision of input and result vectors

8.11.2 Member Function Documentation

8.11.2.1 get_coefficients()

```
template<typename ValueType = default_precision>
const std::vector<std::shared_ptr<const LinOp> >& gko::Combination< ValueType >::get_↩
coefficients ( ) const  [inline], [noexcept]
```

Returns a list of coefficients of the combination.

Returns

a list of coefficients

```
70     {
71         return coefficients_;
72     }
```

8.11.2.2 get_operators()

```
template<typename ValueType = default_precision>
const std::vector<std::shared_ptr<const LinOp> >& gko::Combination< ValueType >::get_↩
operators ( ) const  [inline], [noexcept]
```

Returns a list of operators of the combination.

Returns

a list of operators

The documentation for this class was generated from the following file:

- ginkgo/core/base/combination.hpp (f9f0549a)

8.12 gko::stop::Combined Class Reference

The Combined class is used to combine multiple criterions together through an OR operation.

`#include <ginkgo/core/stop/combined.hpp>`

8.12.1 Detailed Description

The Combined class is used to combine multiple criterions together through an OR operation.

The typical use case is to stop the iteration process if any of the criteria is fulfilled, e.g. a number of iterations, the relative residual norm has reached a threshold, etc.

The documentation for this class was generated from the following file:

- ginkgo/core/stop/combined.hpp (f1a4eb68)

8.13 gko::Composition< ValueType > Class Template Reference

The Composition class can be used to compose linear operators `op1, op2, ..., opn` and obtain the operator `op1 * op2 * ...`

`#include <ginkgo/core/base/composition.hpp>`

Public Member Functions

- const std::vector< std::shared_ptr< const LinOp > > & get_operators () const noexcept
  Returns a list of operators of the composition.

8.13.1 Detailed Description

template<typename ValueType = default_precision>
class gko::Composition< ValueType >

The Composition class can be used to compose linear operators `op1, op2, ..., opn` and obtain the operator `op1 * op2 * ...`

- opn.

Template Parameters

ValueType	precision of input and result vectors

8.13.2 Member Function Documentation

8.13.2.1 get_operators()

```
template<typename ValueType = default_precision>
const std::vector<std::shared_ptr<const LinOp> >& gko::Composition< ValueType >::get_↩
operators ( ) const  [inline], [noexcept]
```

Returns a list of operators of the composition.

Returns

a list of operators

```
71     {
72         return operators_;
73     }
```

The documentation for this class was generated from the following file:

- ginkgo/core/base/composition.hpp (f9f0549a)

8.14 gko::log::Convergence< ValueType > Class Template Reference

Convergence is a Logger which logs data strictly from the `criterion_check_completed` event.

`#include <ginkgo/core/log/convergence.hpp>`

Public Member Functions

- const size_type & get_num_iterations () const noexcept
  Returns the number of iterations.
- const LinOp * get_residual () const noexcept
  Returns the residual.
- const LinOp * get_residual_norm () const noexcept
  Returns the residual norm.

Static Public Member Functions

- static std::unique_ptr< Convergence > create (std::shared_ptr< const Executor > exec, const mask_type &enabled_events=Logger::all_events_mask)
  Creates a convergence logger.

8.14.1 Detailed Description

template<typename ValueType = default_precision>
class gko::log::Convergence< ValueType >

Convergence is a Logger which logs data strictly from the `criterion_check_completed` event.

The purpose of this logger is to give a simple access to standard data generated by the solver once it has converged with minimal overhead.

This logger also computes the residual norm from the residual when the residual norm was not available. This can add some slight overhead.

8.14.2 Member Function Documentation

8.14.2.1 create()

```
template<typename ValueType = default_precision>
static std::unique_ptr<Convergence> gko::log::Convergence< ValueType >::create (
            std::shared_ptr< const Executor > exec,
            const mask_type & enabled_events = Logger::all_events_mask )  [inline], [static]
```

Creates a convergence logger.

This dynamically allocates the memory, constructs the object and returns an std::unique_ptr to this object.

Parameters

exec	the executor
enabled_events	the events enabled for this logger. By default all events.

Returns

an std::unique_ptr to the the constructed object

```
94      {
95          return std::unique_ptr<Convergence>(
96              new Convergence(exec, enabled_events));
97      }
```

8.14.2.2 get_num_iterations()

```
template<typename ValueType = default_precision>
const size_type& gko::log::Convergence< ValueType >::get_num_iterations ( ) const  [inline],
[noexcept]
```

Returns the number of iterations.

Returns

the number of iterations

8.14.2.3 get_residual()

```
template<typename ValueType = default_precision>
const LinOp* gko::log::Convergence< ValueType >::get_residual ( ) const  [inline], [noexcept]
```

Returns the residual.

Returns

the residual

8.14.2.4 get_residual_norm()

```
template<typename ValueType = default_precision>
const LinOp* gko::log::Convergence< ValueType >::get_residual_norm ( ) const  [inline], [noexcept]
```

Returns the residual norm.

Returns

the residual norm

The documentation for this class was generated from the following file:

- ginkgo/core/log/convergence.hpp (f1a4eb68)

8.15 gko::ConvertibleTo< ResultType > Class Template Reference

ConvertibleTo interface is used to mark that the implementer can be converted to the object of ResultType.

```
#include <ginkgo/core/base/polymorphic_object.hpp>
```

Public Member Functions

- virtual void convert_to (result_type *result) const =0

  Converts the implementer to an object of type result_type.
- virtual void move_to (result_type *result)=0

  Converts the implementer to an object of type result_type by moving data from this object.

8.15.1 Detailed Description

template<typename ResultType>
class gko::ConvertibleTo< ResultType >

ConvertibleTo interface is used to mark that the implementer can be converted to the object of ResultType.

This interface is used to enable conversions between polymorphic objects. To mark that an object of type `U` can be converted to an object of type `V`, `U` should implement ConvertibleTo<V>. Then, the implementation of `PolymorphicObject::copy_from` automatically generated by `EnablePolymorphicObject` mixin will use RTTI to figure out that `U` implements the interface and convert it using the convert_to / move_to methods of the interface.

As an example, the following function:

```
 {c++}
void my_function(const U *u, V *v) {
    v->copy_from(u);
}
```

will convert object `u` to object `v` by checking that `u` can be dynamically casted to `ConvertibleTo\<V\>`, and calling ConvertibleTo<V>::convert_to(V*)` to do the actual conversion.

In case `u` is passed as a unique_ptr, call to `convert_to` will be replaced by a call to `move_to` and trigger move semantics.

Template Parameters

ResultType	the type to which the implementer can be converted to, has to be a subclass of PolymorphicObject

8.15.2 Member Function Documentation

8.15.2.1 convert_to()

```
template<typename ResultType>
virtual void gko::ConvertibleTo< ResultType >::convert_to (
            result_type * result ) const  [pure virtual]
```

Converts the implementer to an object of type result_type.

Parameters

result	the object used to store the result of the conversion

Implemented in gko::EnablePolymorphicAssignment< ConcreteType, ResultType >, gko::EnablePolymorphicAssignment< SparsityC
gko::EnablePolymorphicAssignment< Dense< ValueType > >, gko::EnablePolymorphicAssignment< UpperTrs< ValueType, Index
gko::EnablePolymorphicAssignment< Hybrid< ValueType, IndexType > >, gko::EnablePolymorphicAssignment< Identity< ValueTy
gko::EnablePolymorphicAssignment< ConcreteLinOp >, gko::EnablePolymorphicAssignment< Composition< ValueType > >,
gko::EnablePolymorphicAssignment< Bicgstab< ValueType > >, gko::EnablePolymorphicAssignment< LowerTrs< ValueType, Inde
gko::EnablePolymorphicAssignment< Combination< ValueType > >, gko::EnablePolymorphicAssignment< Gmres< ValueType >

gko::EnablePolymorphicAssignment< Csr< ValueType, IndexType > >, gko::EnablePolymorphicAssignment< Ir< ValueType > >, gko::EnablePolymorphicAssignment< Coo< ValueType, IndexType > >, gko::EnablePolymorphicAssignment< Fcg< ValueType > gko::EnablePolymorphicAssignment< Ilu< LSolverType, USolverType, ReverseApply > >, gko::EnablePolymorphicAssignment< Co gko::EnablePolymorphicAssignment< Cgs< ValueType > >, gko::EnablePolymorphicAssignment< Ell< ValueType, IndexType > > gko::EnablePolymorphicAssignment< Jacobi< ValueType, IndexType > >, gko::EnablePolymorphicAssignment< Cg< ValueType > gko::EnablePolymorphicAssignment< Sellp< ValueType, IndexType > >, gko::EnablePolymorphicAssignment< Perturbation< Valu and gko::preconditioner::Jacobi< ValueType, IndexType >.

8.15.2.2 move_to()

```
template<typename ResultType>
virtual void gko::ConvertibleTo< ResultType >::move_to (
            result_type * result )  [pure virtual]
```

Converts the implementer to an object of type result_type by moving data from this object.

This method is used when the implementer is a temporary object, and move semantics can be used.

Parameters

result	the object used to emplace the result of the conversion

Note

ConvertibleTo::move_to can be implemented by simply calling ConvertibleTo::convert_to. However, this operation can often be optimized by exploiting the fact that implementer's data can be moved to the result.

Implemented in gko::EnablePolymorphicAssignment< ConcreteType, ResultType >, gko::EnablePolymorphicAssignment< SparsityC gko::EnablePolymorphicAssignment< Dense< ValueType > >, gko::EnablePolymorphicAssignment< UpperTrs< ValueType, Index gko::EnablePolymorphicAssignment< Hybrid< ValueType, IndexType > >, gko::EnablePolymorphicAssignment< Identity< ValueTy gko::EnablePolymorphicAssignment< ConcreteLinOp >, gko::EnablePolymorphicAssignment< Composition< ValueType > >, gko::EnablePolymorphicAssignment< Bicgstab< ValueType > >, gko::EnablePolymorphicAssignment< LowerTrs< ValueType, Inde gko::EnablePolymorphicAssignment< Combination< ValueType > >, gko::EnablePolymorphicAssignment< Gmres< ValueType > gko::EnablePolymorphicAssignment< Csr< ValueType, IndexType > >, gko::EnablePolymorphicAssignment< Ir< ValueType > >, gko::EnablePolymorphicAssignment< Coo< ValueType, IndexType > >, gko::EnablePolymorphicAssignment< Fcg< ValueType > gko::EnablePolymorphicAssignment< Ilu< LSolverType, USolverType, ReverseApply > >, gko::EnablePolymorphicAssignment< Co gko::EnablePolymorphicAssignment< Cgs< ValueType > >, gko::EnablePolymorphicAssignment< Ell< ValueType, IndexType > > gko::EnablePolymorphicAssignment< Jacobi< ValueType, IndexType > >, gko::EnablePolymorphicAssignment< Cg< ValueType > gko::EnablePolymorphicAssignment< Sellp< ValueType, IndexType > >, gko::EnablePolymorphicAssignment< Perturbation< Valu and gko::preconditioner::Jacobi< ValueType, IndexType >.

The documentation for this class was generated from the following file:

- ginkgo/core/base/polymorphic_object.hpp (3f08cf0a)

8.16 gko::matrix::Coo< ValueType, IndexType > Class Template Reference

COO stores a matrix in the coordinate matrix format.

```
#include <ginkgo/core/matrix/coo.hpp>
```

Public Member Functions

- void read (const mat_data &data) override

  Reads a matrix from a matrix_data structure.
- void write (mat_data &data) const override

  Writes a matrix to a matrix_data structure.
- value_type * get_values () noexcept

  Returns the values of the matrix.
- const value_type * get_const_values () const noexcept

  Returns the values of the matrix.
- index_type * get_col_idxs () noexcept

  Returns the column indexes of the matrix.
- const index_type * get_const_col_idxs () const noexcept

  Returns the column indexes of the matrix.
- index_type * get_row_idxs () noexcept

  Returns the row indexes of the matrix.
- const index_type * get_const_row_idxs () const noexcept
- size_type get_num_stored_elements () const noexcept

  Returns the number of elements explicitly stored in the matrix.
- LinOp * apply2 (const LinOp *b, LinOp *x)

  Applies Coo matrix axpy to a vector (or a sequence of vectors).
- const LinOp * apply2 (const LinOp *b, LinOp *x) const
- LinOp * apply2 (const LinOp *alpha, const LinOp *b, LinOp *x)

  *Performs the operation x = alpha * Coo * b + x.*
- const LinOp * apply2 (const LinOp *alpha, const LinOp *b, LinOp *x) const

  *Performs the operation x = alpha * Coo * b + x.*

8.16.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Coo< ValueType, IndexType >

COO stores a matrix in the coordinate matrix format.

The nonzero elements are stored in an array row-wise (but not neccessarily sorted by column index within a row). Two extra arrays contain the row and column indexes of each nonzero element of the matrix.

Template Parameters

ValueType	precision of matrix elements
IndexType	precision of matrix indexes

8.16.2 Member Function Documentation

8.16.2.1 apply2() [1/4]

```
template<typename ValueType = default_precision, typename IndexType = int32>
LinOp* gko::matrix::Coo< ValueType, IndexType >::apply2 (
            const LinOp * alpha,
            const LinOp * b,
            LinOp * x )  [inline]
```

Performs the operation x = alpha ∗ Coo ∗ b + x.

Parameters

alpha	scaling of the result of Coo ∗ b
b	vector(s) on which the operator is applied
x	output vector(s)

Returns

this

```
212     {
213         this->validate_application_parameters(b, x);
214         GKO_ASSERT_EQUAL_DIMENSIONS(alpha, dim<2>(1, 1));
215         auto exec = this->get_executor();
216         this->apply2_impl(make_temporary_clone(exec, alpha).get(),
217                           make_temporary_clone(exec, b).get(),
218                           make_temporary_clone(exec, x).get());
219         return this;
220     }
```

References gko::PolymorphicObject::get_executor(), and gko::make_temporary_clone().

8.16.2.2 apply2() [2/4]

```
template<typename ValueType = default_precision, typename IndexType = int32>
const LinOp* gko::matrix::Coo< ValueType, IndexType >::apply2 (
            const LinOp * alpha,
            const LinOp * b,
            LinOp * x ) const  [inline]
```

Performs the operation x = alpha ∗ Coo ∗ b + x.

Parameters

alpha	scaling of the result of Coo ∗ b
b	vector(s) on which the operator is applied
x	output vector(s)

Returns

this

References gko::PolymorphicObject::get_executor(), and gko::make_temporary_clone().

8.16.2.3 apply2() [3/4]

```
template<typename ValueType = default_precision, typename IndexType = int32>
LinOp* gko::matrix::Coo< ValueType, IndexType >::apply2 (
            const LinOp * b,
            LinOp * x )  [inline]
```

Applies Coo matrix axpy to a vector (or a sequence of vectors).

Performs the operation x = Coo * b + x

Parameters

b	the input vector(s) on which the operator is applied
x	the output vector(s) where the result is stored

Returns

this

References gko::PolymorphicObject::get_executor(), and gko::make_temporary_clone().

8.16.2.4 apply2() [4/4]

```
template<typename ValueType = default_precision, typename IndexType = int32>
const LinOp* gko::matrix::Coo< ValueType, IndexType >::apply2 (
            const LinOp * b,
            LinOp * x ) const  [inline]
```

References gko::PolymorphicObject::get_executor(), and gko::make_temporary_clone().

8.16.2.5 get_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Coo< ValueType, IndexType >::get_col_idxs ( )  [inline], [noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

References gko::Array< ValueType >::get_data().

8.16.2.6 get_const_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Coo< ValueType, IndexType >::get_const_col_idxs ( ) const  [inline],
[noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.16.2.7 get_const_row_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Coo< ValueType, IndexType >::get_const_row_idxs ( ) const  [inline],
[noexcept]
```

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.16.2.8 get_const_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::Coo< ValueType, IndexType >::get_const_values ( ) const  [inline],
[noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.16.2.9 get_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Coo< ValueType, IndexType >::get_num_stored_elements ( ) const  [inline],
[noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

References gko::Array< ValueType >::get_num_elems().

8.16.2.10 get_row_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Coo< ValueType, IndexType >::get_row_idxs ( )  [inline], [noexcept]
```

Returns the row indexes of the matrix.

Returns

the row indexes of the matrix.

References gko::Array< ValueType >::get_data().

8.16.2.11 get_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::Coo< ValueType, IndexType >::get_values ( )  [inline], [noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

References gko::Array< ValueType >::get_data().

8.16.2.12 read()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Coo< ValueType, IndexType >::read (
            const mat_data & data )  [override], [virtual]
```

Reads a matrix from a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::ReadableFromMatrixData< ValueType, IndexType >.

8.16.2.13 write()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Coo< ValueType, IndexType >::write (
            mat_data & data ) const  [override], [virtual]
```

Writes a matrix to a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::WritableToMatrixData< ValueType, IndexType >.

The documentation for this class was generated from the following file:

- ginkgo/core/matrix/coo.hpp (5545d209)

8.17 gko::copy_back_deleter< T > Class Template Reference

A copy_back_deleter is a type of deleter that copies the data to an internally referenced object before performing the deletion.

`#include <ginkgo/core/base/utils.hpp>`

Public Member Functions

- copy_back_deleter (pointer original)
  Creates a new deleter object.
- void operator() (pointer ptr) const
  Deletes the object.

8.17.1 Detailed Description

template<typename T>
class gko::copy_back_deleter< T >

A copy_back_deleter is a type of deleter that copies the data to an internally referenced object before performing the deletion.

The deleter will use the `copy_from` method to perform the copy, and then delete the passed object using the `delete` keyword. This kind of deleter is useful when temporarily copying an object with the intent of copying it back once it goes out of scope.

There is also a specialization for constant objects that does not perform the copy, since a constant object couldn't have been changed.

Template Parameters

T	the type of object being deleted

8.17.2 Constructor & Destructor Documentation

8.17.2.1 copy_back_deleter()

```
template<typename T >
gko::copy_back_deleter< T >::copy_back_deleter (
            pointer original )  [inline]
```

Creates a new deleter object.

Parameters

original	the origin object where the data will be copied before deletion

8.17.3 Member Function Documentation

8.17.3.1 operator()()

```
template<typename T >
void gko::copy_back_deleter< T >::operator() (
            pointer ptr ) const  [inline]
```

Deletes the object.

Parameters

ptr	pointer to the object being deleted

The documentation for this class was generated from the following file:

- ginkgo/core/base/utils.hpp (4bde4271)

8.18 gko::stop::Criterion Class Reference

The Criterion class is a base class for all stopping criteria.

```
#include <ginkgo/core/stop/criterion.hpp>
```

Classes

- class Updater

  The Updater class serves for convenient argument passing to the Criterion's check function.

Public Member Functions

- Updater update ()

  Returns the updater object.
- bool check (uint8 stoppingId, bool setFinalized, Array< stopping_status > *stop_status, bool *one_changed, const Updater &updater)

  This checks whether convergence was reached for a certain criterion.

8.18.1 Detailed Description

The Criterion class is a base class for all stopping criteria.

It contains a factory to instantiate criteria. It is up to each specific stopping criterion to decide what to do with the data that is passed to it.

Note that depending on the criterion, convergence may not have happened after stopping.

8.18.2 Member Function Documentation

8.18.2.1 check()

```
bool gko::stop::Criterion::check (
            uint8 stoppingId,
            bool setFinalized,
            Array< stopping_status > * stop_status,
            bool * one_changed,
            const Updater & updater )  [inline]
```

This checks whether convergence was reached for a certain criterion.

The actual implantation of the criterion goes here.

Parameters

stoppingId	id of the stopping criterion
setFinalized	Controls if the current version should count as finalized or not
stop_status	status of the stopping criterion
one_changed	indicates if one vector's status changed
updater	the Updater object containing all the information

Returns

whether convergence was completely reached

```
152     {
153         this->template log<log::Logger::criterion_check_started>(
154             this, updater.num_iterations_, updater.residual_,
155             updater.residual_norm_, updater.solution_, stoppingId,
156             setFinalized);
157         auto all_converged = this->check_impl(
158             stoppingId, setFinalized, stop_status, one_changed, updater);
159         this->template log<log::Logger::criterion_check_completed>(
160             this, updater.num_iterations_, updater.residual_,
161             updater.residual_norm_, updater.solution_, stoppingId, setFinalized,
162             stop_status, *one_changed, all_converged);
163         return all_converged;
164     }
```

Referenced by gko::stop::Criterion::Updater::check().

8.18.2.2 update()

```
Updater gko::stop::Criterion::update ( )  [inline]
```

Returns the updater object.

Returns

the updater object

The documentation for this class was generated from the following file:

- ginkgo/core/stop/criterion.hpp (f0a50f96)

8.19 gko::log::criterion_data Struct Reference

Struct representing Criterion related data.

`#include <ginkgo/core/log/record.hpp>`

8.19.1 Detailed Description

Struct representing Criterion related data.

The documentation for this struct was generated from the following file:

- ginkgo/core/log/record.hpp (f0a50f96)

8.20 gko::stop::CriterionArgs Struct Reference

This struct is used to pass parameters to the EnableDefaultCriterionFactoryCriterionFactory::generate() method.

`#include <ginkgo/core/stop/criterion.hpp>`

8.20.1 Detailed Description

This struct is used to pass parameters to the EnableDefaultCriterionFactoryCriterionFactory::generate() method.

It is the ComponentsType of CriterionFactory.

Note

Dependly on the use case, some of these parameters can be `nullptr` as only some stopping criterion require them to be set. An example is the `ResidualNormReduction` which really requires the `initial_residual` to be set.

The documentation for this struct was generated from the following file:

- ginkgo/core/stop/criterion.hpp (f0a50f96)

8.21 gko::matrix::Csr< ValueType, IndexType > Class Template Reference

CSR is a matrix format which stores only the nonzero coefficients by compressing each row of the matrix (compressed sparse row format).

`#include <ginkgo/core/matrix/csr.hpp>`

Public Member Functions

- void read (const mat_data &data) override
  Reads a matrix from a matrix_data structure.
- void write (mat_data &data) const override
  Writes a matrix to a matrix_data structure.
- std::unique_ptr< LinOp > transpose () const override
  Returns a LinOp representing the transpose of the Transposable object.
- std::unique_ptr< LinOp > conj_transpose () const override
  Returns a LinOp representing the conjugate transpose of the Transposable object.
- void sort_by_column_index ()
  Sorts all (value, col_idx) pairs in each row by column index.
- value_type * get_values () noexcept
  Returns the values of the matrix.
- const value_type * get_const_values () const noexcept
  Returns the values of the matrix.
- index_type * get_col_idxs () noexcept
  Returns the column indexes of the matrix.
- const index_type * get_const_col_idxs () const noexcept
  Returns the column indexes of the matrix.
- index_type * get_row_ptrs () noexcept
  Returns the row pointers of the matrix.
- const index_type * get_const_row_ptrs () const noexcept
  Returns the row pointers of the matrix.
- index_type * get_srow () noexcept

Returns the starting rows.

- const index_type * get_const_srow () const noexcept

  Returns the starting rows.

- size_type get_num_srow_elements () const noexcept

  Returns the number of the srow stored elements (involved warps)

- size_type get_num_stored_elements () const noexcept

  Returns the number of elements explicitly stored in the matrix.

- std::shared_ptr< strategy_type > get_strategy () const noexcept

  Returns the strategy.

8.21.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Csr< ValueType, IndexType >

CSR is a matrix format which stores only the nonzero coefficients by compressing each row of the matrix (compressed sparse row format).

The nonzero elements are stored in a 1D array row-wise, and accompanied with a row pointer array which stores the starting index of each row. An additional column index array is used to identify the column of each nonzero element.

Template Parameters

ValueType	precision of matrix elements
IndexType	precision of matrix indexes

8.21.2 Member Function Documentation

8.21.2.1 conj_transpose()

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::unique_ptr<LinOp> gko::matrix::Csr< ValueType, IndexType >::conj_transpose ( ) const
[override], [virtual]
```

Returns a LinOp representing the conjugate transpose of the Transposable object.

Returns

a pointer to the new conjugate transposed object

Implements gko::Transposable.

8.21.2.2 get_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Csr< ValueType, IndexType >::get_col_idxs ( )  [inline], [noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

```
356 { return col_idxs_.get_data(); }
```

References gko::Array< ValueType >::get_data().

8.21.2.3 get_const_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Csr< ValueType, IndexType >::get_const_col_idxs ( ) const  [inline],
[noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.21.2.4 get_const_row_ptrs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Csr< ValueType, IndexType >::get_const_row_ptrs ( ) const  [inline],
[noexcept]
```

Returns the row pointers of the matrix.

Returns

the row pointers of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.21.2.5 get_const_srow()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Csr< ValueType, IndexType >::get_const_srow ( ) const  [inline],
[noexcept]
```

Returns the starting rows.

Returns

the starting rows.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.21.2.6 get_const_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::Csr< ValueType, IndexType >::get_const_values ( ) const  [inline],
[noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.21.2.7 get_num_srow_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Csr< ValueType, IndexType >::get_num_srow_elements ( ) const  [inline],
[noexcept]
```

Returns the number of the srow stored elements (involved warps)

Returns

the number of the srow stored elements (involved warps)

References gko::Array< ValueType >::get_num_elems().

8.21.2.8 get_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Csr< ValueType, IndexType >::get_num_stored_elements ( ) const  [inline],
[noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

References gko::Array< ValueType >::get_num_elems().

8.21.2.9 get_row_ptrs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Csr< ValueType, IndexType >::get_row_ptrs ( )  [inline], [noexcept]
```

Returns the row pointers of the matrix.

Returns

the row pointers of the matrix.

References gko::Array< ValueType >::get_data().

8.21.2.10 get_srow()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Csr< ValueType, IndexType >::get_srow ( )  [inline], [noexcept]
```

Returns the starting rows.

Returns

the starting rows.

References gko::Array< ValueType >::get_data().

8.21.2.11 get_strategy()

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::shared_ptr<strategy_type> gko::matrix::Csr< ValueType, IndexType >::get_strategy ( )
const  [inline], [noexcept]
```

Returns the strategy.

Returns

the strategy

8.21.2.12 get_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::Csr< ValueType, IndexType >::get_values ( )  [inline], [noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

References gko::Array< ValueType >::get_data().

8.21.2.13 read()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Csr< ValueType, IndexType >::read (
            const mat_data & data )  [override], [virtual]
```

Reads a matrix from a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::ReadableFromMatrixData< ValueType, IndexType >.

8.21.2.14 transpose()

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::unique_ptr<LinOp> gko::matrix::Csr< ValueType, IndexType >::transpose ( ) const  [override],
[virtual]
```

Returns a LinOp representing the transpose of the Transposable object.

Returns

a pointer to the new transposed object

Implements gko::Transposable.

8.21.2.15 write()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Csr< ValueType, IndexType >::write (
            mat_data & data ) const  [override], [virtual]
```

Writes a matrix to a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::WritableToMatrixData< ValueType, IndexType >.

The documentation for this class was generated from the following files:

- ginkgo/core/matrix/coo.hpp (5545d209)
- ginkgo/core/matrix/csr.hpp (b8bd4773)

8.22 gko::CublasError Class Reference

CublasError is thrown when a cuBLAS routine throws a non-zero error code.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- CublasError (const std::string &file, int line, const std::string &func, int64 error_code)
  Initializes a cuBLAS error.

8.22.1 Detailed Description

CublasError is thrown when a cuBLAS routine throws a non-zero error code.

8.22.2 Constructor & Destructor Documentation

8.22.2.1 CublasError()

```
gko::CublasError::CublasError (
            const std::string & file,
            int line,
            const std::string & func,
            int64 error_code )  [inline]
```

Initializes a cuBLAS error.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
func	The name of the cuBLAS routine that failed
error_code	The resulting cuBLAS error code

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.23 gko::CudaError Class Reference

CudaError is thrown when a CUDA routine throws a non-zero error code.

`#include <ginkgo/core/base/exception.hpp>`

Public Member Functions

- CudaError (const std::string &file, int line, const std::string &func, int64 error_code)
  Initializes a CUDA error.

8.23.1 Detailed Description

CudaError is thrown when a CUDA routine throws a non-zero error code.

8.23.2 Constructor & Destructor Documentation

8.23.2.1 CudaError()

```
gko::CudaError::CudaError (
            const std::string & file,
            int line,
            const std::string & func,
            int64 error_code )  [inline]
```

Initializes a CUDA error.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
func	The name of the CUDA routine that failed
error_code	The resulting CUDA error code

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.24 gko::CudaExecutor Class Reference

This is the Executor subclass which represents the CUDA device.

`#include <ginkgo/core/base/executor.hpp>`

Public Member Functions

- std::shared_ptr< Executor > get_master () noexcept override

  Returns the master OmpExecutor of this Executor.
- std::shared_ptr< const Executor > get_master () const noexcept override

  Returns the master OmpExecutor of this Executor.
- void synchronize () const override

  Synchronize the operations launched on the executor with its master.
- void run (const Operation &op) const override

  Runs the specified Operation using this Executor.
- int get_device_id () const noexcept

  Get the CUDA device id of the device associated to this executor.
- int get_num_cores_per_sm () const noexcept

  Get the number of cores per SM of this executor.
- int get_num_multiprocessor () const noexcept

  Get the number of multiprocessor of this executor.
- int get_num_warps () const noexcept

  Get the number of warps of this executor.
- int get_major_version () const noexcept

  Get the major verion of compute capability.
- int get_minor_version () const noexcept

  Get the minor verion of compute capability.
- cublasContext * get_cublas_handle () const

  Get the cublas handle for this executor.
- cusparseContext * get_cusparse_handle () const

  Get the cusparse handle for this executor.

Static Public Member Functions

- static std::shared_ptr< CudaExecutor > create (int device_id, std::shared_ptr< Executor > master)

  Creates a new CudaExecutor.
- static int get_num_devices ()

  Get the number of devices present on the system.

8.24.1 Detailed Description

This is the Executor subclass which represents the CUDA device.

8.24.2 Member Function Documentation

8.24.2.1 create()

```
static std::shared_ptr<CudaExecutor> gko::CudaExecutor::create (
            int device_id,
            std::shared_ptr< Executor > master )  [static]
```

Creates a new CudaExecutor.

Parameters

device_id	the CUDA device id of this device
master	an executor on the host that is used to invoke the device kernels

8.24.2.2 get_cublas_handle()

```
cublasContext* gko::CudaExecutor::get_cublas_handle ( ) const  [inline]
```

Get the cublas handle for this executor.

Returns

the cublas handle (cublasContext*) for this executor

```
874 { return cublas_handle_.get(); }
```

8.24.2.3 get_cusparse_handle()

```
cusparseContext* gko::CudaExecutor::get_cusparse_handle ( ) const  [inline]
```

Get the cusparse handle for this executor.

Returns

the cusparse handle (cusparseContext*) for this executor

8.24.2.4 get_master() [1/2]

```
std::shared_ptr<const Executor> gko::CudaExecutor::get_master ( ) const  [override], [virtual],
[noexcept]
```

Returns the master OmpExecutor of this Executor.

Returns

the master OmpExecutor of this Executor.

Implements gko::Executor.

8.24.2.5 get_master() [2/2]

```
std::shared_ptr<Executor> gko::CudaExecutor::get_master ( )  [override], [virtual], [noexcept]
```

Returns the master OmpExecutor of this Executor.

Returns

the master OmpExecutor of this Executor.

Implements gko::Executor.

8.24.2.6 run()

```
void gko::CudaExecutor::run (
            const Operation & op ) const  [override], [virtual]
```

Runs the specified Operation using this Executor.

Parameters

op	the operation to run

Implements gko::Executor.

The documentation for this class was generated from the following file:

- ginkgo/core/base/executor.hpp (f1a4eb68)

8.25 gko::CusparseError Class Reference

CusparseError is thrown when a cuSPARSE routine throws a non-zero error code.

`#include <ginkgo/core/base/exception.hpp>`

Public Member Functions

- CusparseError (const std::string &file, int line, const std::string &func, int64 error_code)
  Initializes a cuSPARSE error.

8.25.1 Detailed Description

CusparseError is thrown when a cuSPARSE routine throws a non-zero error code.

8.25.2 Constructor & Destructor Documentation

8.25.2.1 CusparseError()

```
gko::CusparseError::CusparseError (
            const std::string & file,
            int line,
            const std::string & func,
            int64 error_code )  [inline]
```

Initializes a cuSPARSE error.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
func	The name of the cuSPARSE routine that failed
error_code	The resulting cuSPARSE error code

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.26 gko::default_converter< S, R > Struct Template Reference

Used to convert objects of type S to objects of type R using static_cast.

`#include <ginkgo/core/base/math.hpp>`

Public Member Functions

- R operator() (S val)

  Converts the object to result type.

8.26.1 Detailed Description

template<typename S, typename R>
struct gko::default_converter< S, R >

Used to convert objects of type S to objects of type R using static_cast.

Template Parameters

S	source type
R	result type

8.26.2 Member Function Documentation

8.26.2.1 operator()()

```
template<typename S , typename R >
R gko::default_converter< S, R >::operator() (
            S val )  [inline]
```

Converts the object to result type.

Parameters

val	the object to convert

Returns

the converted object

The documentation for this struct was generated from the following file:

- ginkgo/core/base/math.hpp (1c8cd641)

8.27 gko::matrix::Dense< ValueType > Class Template Reference

Dense is a matrix format which explicitly stores all values of the matrix.

`#include <ginkgo/core/matrix/dense.hpp>`

Public Member Functions

- std::unique_ptr< LinOp > transpose () const override

  Returns a LinOp representing the transpose of the Transposable object.
- std::unique_ptr< LinOp > conj_transpose () const override

  Returns a LinOp representing the conjugate transpose of the Transposable object.
- value_type * get_values () noexcept

  Returns a pointer to the array of values of the matrix.
- const value_type * get_const_values () const noexcept

  Returns a pointer to the array of values of the matrix.
- size_type get_stride () const noexcept

  Returns the stride of the matrix.
- size_type get_num_stored_elements () const noexcept

  Returns the number of elements explicitly stored in the matrix.
- value_type & at (size_type row, size_type col) noexcept

  Returns a single element of the matrix.
- value_type at (size_type row, size_type col) const noexcept

  Returns a single element of the matrix.
- ValueType & at (size_type idx) noexcept

  Returns a single element of the matrix.
- ValueType at (size_type idx) const noexcept

  Returns a single element of the matrix.
- void scale (const LinOp *alpha)

  Scales the matrix with a scalar (aka: BLAS scal).
- void add_scaled (const LinOp *alpha, const LinOp *b)

  Adds `b` scaled by `alpha` to the matrix (aka: BLAS axpy).
- void compute_dot (const LinOp *b, LinOp *result) const

  Computes the column-wise dot product of this matrix and `b`.
- void compute_norm2 (LinOp *result) const

  Computes the Euclidian (L^2) norm of this matrix.
- std::unique_ptr< Dense > create_submatrix (const span &rows, const span &columns, const size_type stride)

  Create a submatrix from the original matrix.

Static Public Member Functions

- static std::unique_ptr< Dense > create_with_config_of (const Dense *other)

  Creates a Dense matrix with the configuration of another Dense matrix.

8.27.1 Detailed Description

template<typename ValueType = default_precision>
class gko::matrix::Dense< ValueType >

Dense is a matrix format which explicitly stores all values of the matrix.

The values are stored in row-major format (values belonging to the same row appear consecutive in the memory). Optionally, rows can be padded for better memory access.

Template Parameters

ValueType	precision of matrix elements

Note

While this format is not very useful for storing sparse matrices, it is often suitable to store vectors, and sets of vectors.

8.27.2 Member Function Documentation

8.27.2.1 add_scaled()

```
template<typename ValueType = default_precision>
void gko::matrix::Dense< ValueType >::add_scaled (
            const LinOp * alpha,
            const LinOp * b )  [inline]
```

Adds `b` scaled by `alpha` to the matrix (aka: BLAS axpy).

Parameters

alpha	If alpha is 1x1 Dense matrix, the entire matrix is scaled by alpha. If it is a Dense row vector of values, then i-th column of the matrix is scaled with the i-th element of alpha (the number of columns of alpha has to match the number of columns of the matrix).
b	a matrix of the same dimension as this

References gko::PolymorphicObject::get_executor(), and gko::make_temporary_clone().

8.27.2.2 at() [1/4]

```
template<typename ValueType = default_precision>
ValueType gko::matrix::Dense< ValueType >::at (
            size_type idx ) const  [inline], [noexcept]
```

Returns a single element of the matrix.

Useful for iterating across all elements of the matrix. However, it is less efficient than the two-parameter variant of this method.

Parameters

idx	a linear index of the requested element (ignoring the stride)

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References gko::Array< ValueType >::get_const_data().

8.27.2.3 at() [2/4]

```
template<typename ValueType = default_precision>
ValueType& gko::matrix::Dense< ValueType >::at (
            size_type idx )  [inline], [noexcept]
```

Returns a single element of the matrix.

Useful for iterating across all elements of the matrix. However, it is less efficient than the two-parameter variant of this method.

Parameters

idx	a linear index of the requested element (ignoring the stride)

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References gko::Array< ValueType >::get_data().

8.27.2.4 at() [3/4]

```
template<typename ValueType = default_precision>
value_type gko::matrix::Dense< ValueType >::at (
              size_type row,
              size_type col ) const  [inline], [noexcept]
```

Returns a single element of the matrix.

Parameters

row	the row of the requested element
col	the column of the requested element

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References gko::Array< ValueType >::get_const_data().

8.27.2.5 at() [4/4]

```
template<typename ValueType = default_precision>
value_type& gko::matrix::Dense< ValueType >::at (
              size_type row,
              size_type col )  [inline], [noexcept]
```

Returns a single element of the matrix.

Parameters

row	the row of the requested element
col	the column of the requested element

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References gko::Array< ValueType >::get_data().

Referenced by gko::initialize().

8.27.2.6 compute_dot()

```
template<typename ValueType = default_precision>
void gko::matrix::Dense< ValueType >::compute_dot (
            const LinOp * b,
            LinOp * result ) const  [inline]
```

Computes the column-wise dot product of this matrix and b.

The conjugate of this is taken.

Parameters

b	a Dense matrix of same dimension as this
result	a Dense row vector, used to store the dot product (the number of column in the vector must match the number of columns of this)

References gko::PolymorphicObject::get_executor(), and gko::make_temporary_clone().

8.27.2.7 compute_norm2()

```
template<typename ValueType = default_precision>
void gko::matrix::Dense< ValueType >::compute_norm2 (
            LinOp * result ) const  [inline]
```

Computes the Euclidian (L^2) norm of this matrix.

Parameters

result	a Dense row vector, used to store the norm (the number of columns in the vector must match the number of columns of this)

References gko::PolymorphicObject::get_executor(), and gko::make_temporary_clone().

8.27.2.8 conj_transpose()

```
template<typename ValueType = default_precision>
std::unique_ptr<LinOp> gko::matrix::Dense< ValueType >::conj_transpose ( ) const  [override],
[virtual]
```

Returns a LinOp representing the conjugate transpose of the Transposable object.

Returns

a pointer to the new conjugate transposed object

Implements gko::Transposable.

8.27.2.9 create_submatrix()

```
template<typename ValueType = default_precision>
std::unique_ptr<Dense> gko::matrix::Dense< ValueType >::create_submatrix (
            const span & rows,
            const span & columns,
            const size_type stride )  [inline]
```

Create a submatrix from the original matrix.

Warning: defining stride for this create_submatrix method might cause wrong memory access. Better use the create_submatrix(rows, columns) method instead.

Parameters

rows	row span
columns	column span
stride	stride of the new submatrix.

References gko::span::begin, gko::PolymorphicObject::get_executor(), gko::matrix::Dense< ValueType >::get_↩ stride(), gko::matrix::Dense< ValueType >::get_values(), and gko::Array< ValueType >::view().

8.27.2.10 create_with_config_of()

```
template<typename ValueType = default_precision>
static std::unique_ptr<Dense> gko::matrix::Dense< ValueType >::create_with_config_of (
            const Dense< ValueType > * other )  [inline], [static]
```

Creates a Dense matrix with the configuration of another Dense matrix.

Parameters

other	The other matrix whose configuration needs to copied.

8.27.2.11 get_const_values()

```
template<typename ValueType = default_precision>
const value_type* gko::matrix::Dense< ValueType >::get_const_values ( ) const  [inline], [noexcept]
```

Returns a pointer to the array of values of the matrix.

Returns

the pointer to the array of values

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.27.2.12 get_num_stored_elements()

```
template<typename ValueType = default_precision>
size_type gko::matrix::Dense< ValueType >::get_num_stored_elements ( ) const  [inline], [noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

References gko::Array< ValueType >::get_num_elems().

8.27.2.13 get_stride()

```
template<typename ValueType = default_precision>
size_type gko::matrix::Dense< ValueType >::get_stride ( ) const  [inline], [noexcept]
```

Returns the stride of the matrix.

Returns

the stride of the matrix.

Referenced by gko::matrix::Dense< ValueType >::create_submatrix().

8.27.2.14 get_values()

```
template<typename ValueType = default_precision>
value_type* gko::matrix::Dense< ValueType >::get_values ( )  [inline], [noexcept]
```

Returns a pointer to the array of values of the matrix.

Returns

the pointer to the array of values

References gko::Array< ValueType >::get_data().

Referenced by gko::matrix::Dense< ValueType >::create_submatrix().

8.27.2.15 scale()

```
template<typename ValueType = default_precision>
void gko::matrix::Dense< ValueType >::scale (
            const LinOp * alpha )  [inline]
```

Scales the matrix with a scalar (aka: BLAS scal).

Parameters

alpha	If alpha is 1x1 Dense matrix, the entire matrix is scaled by alpha. If it is a Dense row vector of values, then i-th column of the matrix is scaled with the i-th element of alpha (the number of columns of alpha has to match the number of columns of the matrix).

References gko::PolymorphicObject::get_executor(), and gko::make_temporary_clone().

8.27.2.16 transpose()

```
template<typename ValueType = default_precision>
std::unique_ptr<LinOp> gko::matrix::Dense< ValueType >::transpose ( ) const  [override],
[virtual]
```

Returns a LinOp representing the transpose of the Transposable object.

Returns

a pointer to the new transposed object

Implements gko::Transposable.

The documentation for this class was generated from the following files:

- ginkgo/core/matrix/coo.hpp (5545d209)
- ginkgo/core/matrix/dense.hpp (b8bd4773)

8.28 gko::dim< Dimensionality, DimensionType > Struct Template Reference

A type representing the dimensions of a multidimensional object.

```
#include <ginkgo/core/base/dim.hpp>
```

Public Member Functions

- constexpr dim (const dimension_type &size=dimension_type{})

  Creates a dimension object with all dimensions set to the same value.
- template<typename... Rest>
  constexpr dim (const dimension_type &first, const Rest &... rest)

  Creates a dimension object with the specified dimensions.
- constexpr const dimension_type & operator[ ] (const size_type &dimension) const noexcept

  Returns the requested dimension.
- dimension_type & operator[ ] (const size_type &dimension) noexcept
- constexpr operator bool () const

  Checks if all dimensions evaluate to true.

Friends

- constexpr friend bool operator== (const dim &x, const dim &y)

  Checks if two dim objects are equal.
- constexpr friend dim operator* (const dim &x, const dim &y)

  Multiplies two dim objects.

8.28.1 Detailed Description

template<size_type Dimensionality, typename DimensionType = size_type>
struct gko::dim< Dimensionality, DimensionType >

A type representing the dimensions of a multidimensional object.

Template Parameters

Dimensionality	number of dimensions of the object
DimensionType	datatype used to represent each dimension

8.28.2 Constructor & Destructor Documentation

8.28.2.1 dim() [1/2]

```
template<size_type Dimensionality, typename DimensionType = size_type>
constexpr gko::dim< Dimensionality, DimensionType >::dim (
            const dimension_type & size = dimension_type{} )  [inline], [constexpr]
```

Creates a dimension object with all dimensions set to the same value.

Parameters

size	the size of each dimension

8.28.2.2 dim() [2/2]

```
template<size_type Dimensionality, typename DimensionType = size_type>
template<typename...  Rest>
constexpr gko::dim< Dimensionality, DimensionType >::dim (
            const dimension_type & first,
            const Rest &...  rest )  [inline], [constexpr]
```

Creates a dimension object with the specified dimensions.

If the number of dimensions given is less than the dimensionality of the object, the remaining dimensions are set to the same value as the last value given.

For example, in the context of matrices `dim<2>{2, 3}` creates the dimensions for a 2-by-3 matrix.

Parameters

first	first dimension
rest	other dimensions

8.28.3 Member Function Documentation

8.28.3.1 operator bool()

```
template<size_type Dimensionality, typename DimensionType = size_type>
constexpr gko::dim< Dimensionality, DimensionType >::operator bool ( ) const  [inline], [constexpr]
```

Checks if all dimensions evaluate to true.

For standard arithmetic types, this is equivalent to all dimensions being different than zero.

Returns

true if and only if all dimensions evaluate to true

8.28.3.2 operator[]() [1/2]

```
template<size_type Dimensionality, typename DimensionType = size_type>
constexpr const dimension_type& gko::dim< Dimensionality, DimensionType >::operator[] (
            const size_type & dimension ) const  [inline], [constexpr], [noexcept]
```

Returns the requested dimension.

For example, if `d` is a dim<2> object representing matrix dimensions, `d[0]` returns the number of rows, and `d[1]` returns the number of columns.

Parameters

dimension	the requested dimension

Returns

the `dimension`-th dimension

8.28.3.3 operator[]() [2/2]

```
template<size_type Dimensionality, typename DimensionType = size_type>
dimension_type& gko::dim< Dimensionality, DimensionType >::operator[] (
            const size_type & dimension )  [inline], [noexcept]
```

8.28.4 Friends And Related Function Documentation

8.28.4.1 operator*

```
template<size_type Dimensionality, typename DimensionType = size_type>
constexpr friend dim operator* (
            const dim< Dimensionality, DimensionType > & x,
            const dim< Dimensionality, DimensionType > & y )  [friend]
```

Multiplies two dim objects.

Parameters

x	first object
y	second object

Returns

a dim object representing the size of the tensor product `x * y`

8.28.4.2 operator==

```
template<size_type Dimensionality, typename DimensionType = size_type>
constexpr friend bool operator== (
            const dim< Dimensionality, DimensionType > & x,
            const dim< Dimensionality, DimensionType > & y )  [friend]
```

Checks if two dim objects are equal.

Parameters

x	first object
y	second object

Returns

true if and only if all dimensions of both objects are equal.

The documentation for this struct was generated from the following file:

- ginkgo/core/base/dim.hpp (f1a4eb68)

8.29 gko::DimensionMismatch Class Reference

DimensionMismatch is thrown if an operation is being applied to LinOps of incompatible size.

`#include <ginkgo/core/base/exception.hpp>`

Public Member Functions

- DimensionMismatch (const std::string &file, int line, const std::string &func, const std::string &first_name, size_type first_rows, size_type first_cols, const std::string &second_name, size_type second_rows, size_type second_cols, const std::string &clarification)

  Initializes a dimension mismatch error.

8.29.1 Detailed Description

DimensionMismatch is thrown if an operation is being applied to LinOps of incompatible size.

8.29.2 Constructor & Destructor Documentation

8.29.2.1 DimensionMismatch()

```
gko::DimensionMismatch::DimensionMismatch (
            const std::string & file,
            int line,
            const std::string & func,
            const std::string & first_name,
            size_type first_rows,
            size_type first_cols,
            const std::string & second_name,
            size_type second_rows,
            size_type second_cols,
            const std::string & clarification )  [inline]
```

Initializes a dimension mismatch error.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
func	The function name where the error occurred
first_name	The name of the first operator
first_rows	The output dimension of the first operator
first_cols	The input dimension of the first operator
second_name	The name of the second operator
second_rows	The output dimension of the second operator
second_cols	The input dimension of the second operator

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.30 gko::matrix::Ell< ValueType, IndexType > Class Template Reference

ELL is a matrix format where stride with explicit zeros is used such that all rows have the same number of stored elements.

`#include <ginkgo/core/matrix/ell.hpp>`

Public Member Functions

- void read (const mat_data &data) override
  Reads a matrix from a matrix_data structure.
- void write (mat_data &data) const override
  Writes a matrix to a matrix_data structure.
- value_type * get_values () noexcept
  Returns the values of the matrix.
- const value_type * get_const_values () const noexcept
  Returns the values of the matrix.
- index_type * get_col_idxs () noexcept
  Returns the column indexes of the matrix.
- const index_type * get_const_col_idxs () const noexcept
  Returns the column indexes of the matrix.
- size_type get_num_stored_elements_per_row () const noexcept
  Returns the number of stored elements per row.
- size_type get_stride () const noexcept
  Returns the stride of the matrix.
- size_type get_num_stored_elements () const noexcept
  Returns the number of elements explicitly stored in the matrix.
- value_type & val_at (size_type row, size_type idx) noexcept
  Returns the `idx`-th non-zero element of the `row`-th row .
- value_type val_at (size_type row, size_type idx) const noexcept
  Returns the `idx`-th non-zero element of the `row`-th row .
- index_type & col_at (size_type row, size_type idx) noexcept
  Returns the `idx`-th column index of the `row`-th row .
- index_type col_at (size_type row, size_type idx) const noexcept
  Returns the `idx`-th column index of the `row`-th row .

8.30.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Ell< ValueType, IndexType >

ELL is a matrix format where stride with explicit zeros is used such that all rows have the same number of stored elements.

The number of elements stored in each row is the largest number of nonzero elements in any of the rows (obtainable through get_num_stored_elements_per_row() method). This removes the need of a row pointer like in the CSR format, and allows for SIMD processing of the distinct rows. For efficient processing, the nonzero elements and the corresponding column indices are stored in column-major fashion. The columns are padded to the length by user-defined stride parameter whose default value is the number of rows of the matrix.

Template Parameters

ValueType	precision of matrix elements
IndexType	precision of matrix indexes

8.30.2 Member Function Documentation

8.30.2.1 col_at() [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type gko::matrix::Ell< ValueType, IndexType >::col_at (
            size_type row,
            size_type idx ) const  [inline], [noexcept]
```

Returns the `idx`-th column index of the `row`-th row .

Parameters

row	the row of the requested element
idx	the idx-th stored element of the row

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

```
209     {
210         return this->get_const_col_idxs()[this->linearize_index(row, idx)];
211     }
```

References gko::matrix::Ell< ValueType, IndexType >::get_const_col_idxs().

8.30.2.2 col_at() [2/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type& gko::matrix::Ell< ValueType, IndexType >::col_at (
            size_type row,
            size_type idx )  [inline], [noexcept]
```

Returns the `idx`-th column index of the `row`-th row .

Parameters

row	the row of the requested element
idx	the idx-th stored element of the row

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References gko::matrix::Ell< ValueType, IndexType >::get_col_idxs().

8.30.2.3 get_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Ell< ValueType, IndexType >::get_col_idxs ( )  [inline], [noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

References gko::Array< ValueType >::get_data().

Referenced by gko::matrix::Ell< ValueType, IndexType >::col_at().

8.30.2.4 get_const_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Ell< ValueType, IndexType >::get_const_col_idxs ( ) const  [inline],
[noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

Referenced by gko::matrix::Ell< ValueType, IndexType >::col_at().

8.30.2.5 get_const_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::Ell< ValueType, IndexType >::get_const_values ( ) const  [inline],
[noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.30.2.6 get_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Ell< ValueType, IndexType >::get_num_stored_elements ( ) const  [inline],
[noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

References gko::Array< ValueType >::get_num_elems().

8.30.2.7 get_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Ell< ValueType, IndexType >::get_num_stored_elements_per_row ( ) const
[inline], [noexcept]
```

Returns the number of stored elements per row.

Returns

the number of stored elements per row.

8.30.2.8 get_stride()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Ell< ValueType, IndexType >::get_stride ( ) const  [inline], [noexcept]
```

Returns the stride of the matrix.

Returns

the stride of the matrix.

8.30.2.9 get_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::Ell< ValueType, IndexType >::get_values ( )  [inline], [noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

References gko::Array< ValueType >::get_data().

8.30.2.10 read()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Ell< ValueType, IndexType >::read (
            const mat_data & data )  [override], [virtual]
```

Reads a matrix from a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::ReadableFromMatrixData< ValueType, IndexType >.

8.30.2.11 val_at() [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type gko::matrix::Ell< ValueType, IndexType >::val_at (
```

```
            size_type row,
            size_type idx ) const  [inline], [noexcept]
```

Returns the `idx`-th non-zero element of the `row`-th row .

Parameters

row	the row of the requested element
idx	the idx-th stored element of the row

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References gko::Array< ValueType >::get_const_data().

8.30.2.12 val_at() [2/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type& gko::matrix::Ell< ValueType, IndexType >::val_at (
            size_type row,
            size_type idx ) [inline], [noexcept]
```

Returns the `idx`-th non-zero element of the `row`-th row .

Parameters

row	the row of the requested element
idx	the idx-th stored element of the row

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References gko::Array< ValueType >::get_data().

8.30.2.13 write()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Ell< ValueType, IndexType >::write (
            mat_data & data ) const [override], [virtual]
```

Writes a matrix to a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::WritableToMatrixData< ValueType, IndexType >.

The documentation for this class was generated from the following files:

- ginkgo/core/matrix/csr.hpp (b8bd4773)
- ginkgo/core/matrix/ell.hpp (8045ac75)

8.31 gko::enable_parameters_type< ConcreteParametersType, Factory > Struct Template Reference

The enable_parameters_type mixin is used to create a base implementation of the factory parameters structure.

`#include <ginkgo/core/base/abstract_factory.hpp>`

Public Member Functions

- std::unique_ptr< Factory > on (std::shared_ptr< const Executor > exec) const

  Creates a new factory on the specified executor.

8.31.1 Detailed Description

template<typename ConcreteParametersType, typename Factory>
struct gko::enable_parameters_type< ConcreteParametersType, Factory >

The enable_parameters_type mixin is used to create a base implementation of the factory parameters structure.

It provides only the on() method which can be used to instantiate the factory give the parameters stored in the structure.

Template Parameters

ConcreteParametersType	the concrete parameters type which is being implemented [CRTP parameter]
Factory	the concrete factory for which these parameters are being used

8.31.2 Member Function Documentation

8.31.2.1 on()

```
template<typename ConcreteParametersType, typename Factory>
std::unique_ptr<Factory> gko::enable_parameters_type< ConcreteParametersType, Factory >::on (
            std::shared_ptr< const Executor > exec ) const  [inline]
```

Creates a new factory on the specified executor.

Parameters

exec	the executor where the factory will be created

Returns

a new factory instance

The documentation for this struct was generated from the following file:

- ginkgo/core/base/abstract_factory.hpp (8045ac75)

8.32 gko::EnableAbstractPolymorphicObject< AbstactObject, PolymorphicBase > Class Template Reference

This mixin inherits from (a subclass of) PolymorphicObject and provides a base implementation of a new abstract object.

`#include <ginkgo/core/base/polymorphic_object.hpp>`

8.32.1 Detailed Description

template<typename AbstactObject, typename PolymorphicBase = PolymorphicObject>
class gko::EnableAbstractPolymorphicObject< AbstactObject, PolymorphicBase >

This mixin inherits from (a subclass of) PolymorphicObject and provides a base implementation of a new abstract object.

It uses method hiding to update the parameter and return types from `PolymorphicObject to`AbstractObject` wherever it makes sense. As opposed to EnablePolymorphicObject, it does not implement PolymorphicObject's virtual methods.

Template Parameters

AbstractObject	the abstract class which is being implemented [CRTP parameter]
PolymorphicBase	parent of AbstractObject in the polymorphic hierarchy, has to be a subclass of polymorphic object

See also

EnablePolymorphicObject for creating a concrete subclass of PolymorphicObject.

The documentation for this class was generated from the following file:

- ginkgo/core/base/polymorphic_object.hpp (3f08cf0a)

8.33 gko::EnableCreateMethod< ConcreteType > Class Template Reference

This mixin implements a static `create()` method on `ConcreteType` that dynamically allocates the memory, uses the passed-in arguments to construct the object, and returns an std::unique_ptr to such an object.

`#include <ginkgo/core/base/polymorphic_object.hpp>`

8.33.1 Detailed Description

template<typename ConcreteType>
class gko::EnableCreateMethod< ConcreteType >

This mixin implements a static `create()` method on `ConcreteType` that dynamically allocates the memory, uses the passed-in arguments to construct the object, and returns an std::unique_ptr to such an object.

Template Parameters

ConcreteObject	the concrete type for which `create()` is being implemented [CRTP parameter]

The documentation for this class was generated from the following file:

- ginkgo/core/base/polymorphic_object.hpp (3f08cf0a)

8.34 gko::EnableDefaultFactory< ConcreteFactory, ProductType, ParametersType, PolymorphicBase > Class Template Reference

This mixin provides a default implementation of a concrete factory.

`#include <ginkgo/core/base/abstract_factory.hpp>`

Public Member Functions

- const parameters_type & get_parameters () const noexcept

  Returns the parameters of the factory.

Static Public Member Functions

- static parameters_type create ()

  Creates a new ParametersType object which can be used to instantiate a new ConcreteFactory.

8.34.1 Detailed Description

template<typename ConcreteFactory, typename ProductType, typename ParametersType, typename PolymorphicBase>
class gko::EnableDefaultFactory< ConcreteFactory, ProductType, ParametersType, PolymorphicBase >

This mixin provides a default implementation of a concrete factory.

It implements all the methods of AbstractFactory and PolymorphicObject. Its implementation of the generate_impl() method delegates the creation of the product by calling the `ProductType::ProductType(const ConcreteFactory *, const components_type &)` constructor. The factory also supports parameters by using the `ParametersType` structure, which is defined by the user.

For a simple example, see IntFactory in `core/test/base/abstract_factory.cpp`.

Template Parameters

ConcreteFactory	the concrete factory which is being implemented [CRTP parameter]
ProductType	the concrete type of products which this factory produces, has to be a subclass of PolymorphicBase::abstract_product_type
ParametersType	a type representing the parameters of the factory, has to inherit from the enable_parameters_type mixin
PolymorphicBase	parent of ConcreteFactory in the polymorphic hierarchy, has to be a subclass of AbstractFactory

8.34.2 Member Function Documentation

8.34.2.1 create()

```
template<typename ConcreteFactory , typename ProductType , typename ParametersType , typename
PolymorphicBase >
static parameters_type gko::EnableDefaultFactory< ConcreteFactory, ProductType, Parameters
Type, PolymorphicBase >::create ( )  [inline], [static]
```

Creates a new ParametersType object which can be used to instantiate a new ConcreteFactory.

This method does not construct the factory directly, but returns a new parameters_type object, which can be used to set the parameters of the factory. Once the parameters have been set, the parameters_type::on() method can be used to obtain an instance of the factory with those parameters.

Returns

a default parameters_type object

8.34.2.2 get_parameters()

```
template<typename ConcreteFactory , typename ProductType , typename ParametersType , typename
PolymorphicBase >
const parameters_type& gko::EnableDefaultFactory< ConcreteFactory, ProductType, Parameters↩
Type, PolymorphicBase >::get_parameters ( ) const  [inline], [noexcept]
```

Returns the parameters of the factory.

Returns

the parameters of the factory

The documentation for this class was generated from the following file:

- ginkgo/core/base/abstract_factory.hpp (8045ac75)

8.35 gko::EnableLinOp< ConcreteLinOp, PolymorphicBase > Class Template Reference

The EnableLinOp mixin can be used to provide sensible default implementations of the majority of the LinOp and PolymorphicObject interface.

```
#include <ginkgo/core/base/lin_op.hpp>
```

Additional Inherited Members

8.35.1 Detailed Description

template<typename ConcreteLinOp, typename PolymorphicBase = LinOp>
class gko::EnableLinOp< ConcreteLinOp, PolymorphicBase >

The EnableLinOp mixin can be used to provide sensible default implementations of the majority of the LinOp and PolymorphicObject interface.

The goal of the mixin is to facilitate the development of new LinOp, by enabling the implementers to focus on the important parts of their operator, while the library takes care of generating the trivial utility functions. The mixin will provide default implementations for the entire PolymorphicObject interface, including a default implementation of `copy_from` between objects of the new LinOp type. It will also hide the default LinOp::apply() methods with versions that preserve the static type of the object.

Implementers of new LinOps are required to specify only the following aspects:

1. Creation of the LinOp: This can be facilitated via either EnableCreateMethod mixin (used mostly for matrix formats), or GKO_ENABLE_LIN_OP_FACTORY macro (used for operators created from other operators, like preconditioners and solvers).
2. Application of the LinOp: Implementers have to override the two overloads of the LinOp::apply_impl() virtual methods.

Template Parameters

ConcreteLinOp	the concrete LinOp which is being implemented [CRTP parameter]
PolymorphicBase	parent of ConcreteLinOp in the polymorphic hierarchy, has to be a subclass of LinOp

The documentation for this class was generated from the following file:

- ginkgo/core/base/lin_op.hpp (4c758394)

8.36 gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase > Class Template Reference

EnableLogging is a mixin which should be inherited by any class which wants to enable logging.

`#include <ginkgo/core/log/logger.hpp>`

Public Member Functions

- void add_logger (std::shared_ptr< const Logger > logger) override

  Adds a new logger to the list of subscribed loggers.
- void remove_logger (const Logger *logger) override

  Removes a logger from the list of subscribed loggers.

8.36.1 Detailed Description

template<typename ConcreteLoggable, typename PolymorphicBase = Loggable>
class gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase >

EnableLogging is a mixin which should be inherited by any class which wants to enable logging.

All the received events are passed to the loggers this class contains.

Template Parameters

ConcreteLoggable	the object being logged [CRTP parameter]
PolymorphicBase	the polymorphic base of this class. By default it is Loggable. Change it if you want to use a new superclass of `Loggable` as polymorphic base of this class.

8.36.2 Member Function Documentation

8.36.2.1 add_logger()

```
template<typename ConcreteLoggable, typename PolymorphicBase = Loggable>
void gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase >::add_logger (
            std::shared_ptr< const Logger > logger )  [inline], [override], [virtual]
```

Adds a new logger to the list of subscribed loggers.

Parameters

logger	the logger to add

Implements gko::log::Loggable.

```
524     {
525         loggers_.push_back(logger);
526     }
```

8.36.2.2 remove_logger()

```
template<typename ConcreteLoggable, typename PolymorphicBase = Loggable>
void gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase >::remove_logger (
             const Logger * logger )  [inline], [override], [virtual]
```

Removes a logger from the list of subscribed loggers.

Parameters

logger	the logger to remove

Note

The comparison is done using the logger's object unique identity. Thus, two loggers constructed in the same way are not considered equal.

Implements gko::log::Loggable.

The documentation for this class was generated from the following file:

- ginkgo/core/log/logger.hpp (0d7578c9)

8.37 gko::EnablePolymorphicAssignment< ConcreteType, ResultType > Class Template Reference

This mixin is used to enable a default PolymorphicObject::copy_from() implementation for objects that have implemented conversions between them.

```
#include <ginkgo/core/base/polymorphic_object.hpp>
```

Public Member Functions

- void convert_to (result_type *result) const override

  Converts the implementer to an object of type result_type.
- void move_to (result_type *result) override

  Converts the implementer to an object of type result_type by moving data from this object.

8.37.1 Detailed Description

template<typename ConcreteType, typename ResultType = ConcreteType>
class gko::EnablePolymorphicAssignment< ConcreteType, ResultType >

This mixin is used to enable a default PolymorphicObject::copy_from() implementation for objects that have implemented conversions between them.

The requirement is that there is either a conversion constructor from `ConcreteType` in `ResultType`, or a conversion operator to `ResultType` in `ConcreteType`.

Template Parameters

ConcreteType	the concrete type from which the copy_from is being enabled [CRTP parameter]
ResultType	the type to which copy_from is being enabled

8.37.2 Member Function Documentation

8.37.2.1 convert_to()

```
template<typename ConcreteType, typename ResultType = ConcreteType>
void gko::EnablePolymorphicAssignment< ConcreteType, ResultType >::convert_to (
            result_type * result ) const  [inline], [override], [virtual]
```

Converts the implementer to an object of type result_type.

Parameters

result	the object used to store the result of the conversion

Implements gko::ConvertibleTo< ResultType >.

8.37.2.2 move_to()

```
template<typename ConcreteType, typename ResultType = ConcreteType>
void gko::EnablePolymorphicAssignment< ConcreteType, ResultType >::move_to (
            result_type * result )  [inline], [override], [virtual]
```

Converts the implementer to an object of type result_type by moving data from this object.

This method is used when the implementer is a temporary object, and move semantics can be used.

Parameters

result	the object used to emplace the result of the conversion

Note

ConvertibleTo::move_to can be implemented by simply calling ConvertibleTo::convert_to. However, this operation can often be optimized by exploiting the fact that implementer's data can be moved to the result.

Implements gko::ConvertibleTo< ResultType >.

The documentation for this class was generated from the following file:

- ginkgo/core/base/polymorphic_object.hpp (3f08cf0a)

8.38 gko::EnablePolymorphicObject< ConcreteObject, PolymorphicBase > Class Template Reference

This mixin inherits from (a subclass of) PolymorphicObject and provides a base implementation of a new concrete polymorphic object.

`#include <ginkgo/core/base/polymorphic_object.hpp>`

8.38.1 Detailed Description

template<typename ConcreteObject, typename PolymorphicBase = PolymorphicObject>
class gko::EnablePolymorphicObject< ConcreteObject, PolymorphicBase >

This mixin inherits from (a subclass of) PolymorphicObject and provides a base implementation of a new concrete polymorphic object.

The mixin changes parameter and return types of appropriate public methods of PolymorphicObject in the same way EnableAbstractPolymorphicObject does. In addition, it also provides default implementations of PolymorphicObject's vritual methods by using the *executor default constructor* and the assignment operator of ConcreteObject. Consequently, the following is a minimal example of PolymorphicObject:

```
 {c++}
struct MyObject : EnablePolymorphicObject<MyObject> {
    MyObject(std::shared_ptr<const Executor> exec)
        : EnablePolymorphicObject<MyObject>(std::move(exec))
    {}
};
```

In a way, this mixin can be viewed as an extension of default constructor/destructor/assignment operators.

Note

This mixin does not enable copying the polymorphic object to the object of the same type (i.e. it does not implement the ConvertibleTo<ConcreteObject> interface). To enable a default implementation of this interface see the EnablePolymorphicAssignment mixin.

Template Parameters

ConcreteObject	the concrete type which is being implemented [CRTP parameter]
PolymorphicBase	parent of ConcreteObject in the polymorphic hierarchy, has to be a subclass of polymorphic object

The documentation for this class was generated from the following file:

- ginkgo/core/base/polymorphic_object.hpp (3f08cf0a)

8.39 gko::Error Class Reference

The Error class is used to report exceptional behaviour in library functions.

`#include <ginkgo/core/base/exception.hpp>`

Public Member Functions

- Error (const std::string &file, int line, const std::string &what)

  Initializes an error.
- virtual const char * what () const noexcept override

  Returns a human-readable string with a more detailed description of the error.

8.39.1 Detailed Description

The Error class is used to report exceptional behaviour in library functions.

Ginkgo uses C++ exception mechanism to this end, and the Error class represents a base class for all types of errors. The exact list of errors which could occur during the execution of a certain library routine is provided in the documentation of that routine, along with a short description of the situation when that error can occur. During runtime, these errors can be detected by using standard C++ try-catch blocks, and a human-readable error description can be obtained by calling the Error::what() method.

As an example, trying to compute a matrix-vector product with arguments of incompatible size will result in a DimensionMismatch error, which is demonstrated in the following program.

```
#include <ginkgo.h>
#include <iostream>
using namespace gko;
int main()
{
    auto omp = create<OmpExecutor>();
    auto A = randn_fill<matrix::Csr<float»(5, 5, 0f, 1f, omp);
    auto x = fill<matrix::Dense<float»(6, 1, 1f, omp);
    try {
        auto y = apply(A.get(), x.get());
    } catch(Error e) {
        // an error occured, write the message to screen and exit
        std::cout « e.what() « std::endl;
        return -1;
    }
    return 0;
}
```

8.39.2 Constructor & Destructor Documentation

8.39.2.1 Error()

```
gko::Error::Error (
            const std::string & file,
            int line,
            const std::string & what )  [inline]
```

Initializes an error.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
what	The error message

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.40 gko::Executor Class Reference

The first step in using the Ginkgo library consists of creating an executor.

```
#include <ginkgo/core/base/executor.hpp>
```

Public Member Functions

- virtual void run (const Operation &op) const =0
  Runs the specified Operation using this Executor.
- template<typename ClosureOmp , typename ClosureCuda >
  void run (const ClosureOmp &op_omp, const ClosureCuda &op_cuda) const
  Runs one of the passed in functors, depending on the Executor type.
- template<typename T >
  T * alloc (size_type num_elems) const
  Allocates memory in this Executor.
- void free (void *ptr) const noexcept
  Frees memory previously allocated with Executor::alloc().
- template<typename T >
  void copy_from (const Executor *src_exec, size_type num_elems, const T *src_ptr, T *dest_ptr) const
  Copies data from another Executor.
- virtual std::shared_ptr< Executor > get_master () noexcept=0
  Returns the master OmpExecutor of this Executor.
- virtual std::shared_ptr< const Executor > get_master () const noexcept=0
  Returns the master OmpExecutor of this Executor.
- virtual void synchronize () const =0
  Synchronize the operations launched on the executor with its master.

8.40.1 Detailed Description

The first step in using the Ginkgo library consists of creating an executor.

Executors are used to specify the location for the data of linear algebra objects, and to determine where the operations will be executed. Ginkgo currently supports three different executor types:

- OmpExecutor specifies that the data should be stored and the associated operations executed on an OpenMP-supporting device (e.g. host CPU);
- CudaExecutor specifies that the data should be stored and the operations executed on the NVIDIA GPU accelerator;
- ReferenceExecutor executes a non-optimized reference implementation, which can be used to debug the library.

The following code snippet demonstrates the simplest possible use of the Ginkgo library:

```
auto omp = gko::create<gko::OmpExecutor>();
auto A = gko::read_from_mtx<gko::matrix::Csr<float»("A.mtx", omp);
```

First, we create a OMP executor, which will be used in the next line to specify where we want the data for the matrix A to be stored. The second line will read a matrix from the matrix market file 'A.mtx', and store the data on the CPU in CSR format (gko::matrix::Csr is a Ginkgo matrix class which stores its data in CSR format). At this point, matrix A is bound to the CPU, and any routines called on it will be performed on the CPU. This approach is usually desired in sparse linear algebra, as the cost of individual operations is several orders of magnitude lower than the cost of copying the matrix to the GPU.

If matrix A is going to be reused multiple times, it could be beneficial to copy it over to the accelerator, and perform the operations there, as demonstrated by the next code snippet:

```
auto cuda = gko::create<gko::CudaExecutor>(0, omp);
auto dA = gko::copy_to<gko::matrix::Csr<float»(A.get(), cuda);
```

The first line of the snippet creates a new CUDA executor. Since there may be multiple NVIDIA GPUs present on the system, the first parameter instructs the library to use the first device (i.e. the one with device ID zero, as in cudaSetDevice() routine from the CUDA runtime API). In addition, since GPUs are not stand-alone processors, it is required to pass a "master" OmpExecutor which will be used to schedule the requested CUDA kernels on the accelerator.

The second command creates a copy of the matrix A on the GPU. Notice the use of the get() method. As Ginkgo aims to provide automatic memory management of its objects, the result of calling gko::read_from_mtx() is a smart pointer (std::unique_ptr) to the created object. On the other hand, as the library will not hold a reference to A once the copy is completed, the input parameter for gko::copy_to() is a plain pointer. Thus, the get() method is used to convert from a std::unique_ptr to a plain pointer, as expected by gko::copy_to().

As a side note, the gko::copy_to routine is far more powerful than just copying data between different devices. It can also be used to convert data between different formats. For example, if the above code used gko::matrix::Ell as the template parameter, dA would be stored on the GPU, in ELLPACK format.

Finally, if all the processing of the matrix is supposed to be done on the GPU, and a CPU copy of the matrix is not required, we could have read the matrix to the GPU directly:

```
auto omp = gko::create<gko::OmpExecutor>();
auto cuda = gko::create<gko::CudaExecutor>(0, omp);
auto dA = gko::read_from_mtx<gko::matrix::Csr<float»("A.mtx", cuda);
```

Notice that even though reading the matrix directly from a file to the accelerator is not supported, the library is designed to abstract away the intermediate step of reading the matrix to the CPU memory. This is a general design approach taken by the library: in case an operation is not supported by the device, the data will be copied to the CPU, the operation performed there, and finally the results copied back to the device. This approach makes using the library more concise, as explicit copies are not required by the user. Nevertheless, this feature should be taken into account when considering performance implications of using such operations.

8.40.2 Member Function Documentation

8.40.2.1 alloc()

```
template<typename T >
T* gko::Executor::alloc (
            size_type num_elems ) const  [inline]
```

Allocates memory in this Executor.

Template Parameters

T	datatype to allocate

Parameters

num_elems	number of elements of type T to allocate

Exceptions

AllocationError	if the allocation failed

Returns

pointer to allocated memory

8.40.2.2 copy_from()

```
template<typename T >
void gko::Executor::copy_from (
            const Executor * src_exec,
            size_type num_elems,
            const T * src_ptr,
            T * dest_ptr ) const  [inline]
```

Copies data from another Executor.

Template Parameters

T	datatype to copy

Parameters

src_exec	Executor from which the memory will be copied

Parameters

num_elems	number of elements of type T to copy
src_ptr	pointer to a block of memory containing the data to be copied
dest_ptr	pointer to an allocated block of memory where the data will be copied to

8.40.2.3 free()

```
void gko::Executor::free (
            void * ptr ) const  [inline], [noexcept]
```

Frees memory previously allocated with Executor::alloc().

If `ptr` is a `nullptr`, the function has no effect.

Parameters

ptr	pointer to the allocated memory block

8.40.2.4 get_master() [1/2]

```
virtual std::shared_ptr<const Executor> gko::Executor::get_master ( ) const  [pure virtual],
[noexcept]
```

Returns the master OmpExecutor of this Executor.

Returns

the master OmpExecutor of this Executor.

Implemented in gko::CudaExecutor, and gko::OmpExecutor.

8.40.2.5 get_master() [2/2]

```
virtual std::shared_ptr<Executor> gko::Executor::get_master ( )  [pure virtual], [noexcept]
```

Returns the master OmpExecutor of this Executor.

Returns

the master OmpExecutor of this Executor.

Implemented in gko::CudaExecutor, and gko::OmpExecutor.

8.40.2.6 run() [1/2]

```
template<typename ClosureOmp , typename ClosureCuda >
void gko::Executor::run (
            const ClosureOmp & op_omp,
            const ClosureCuda & op_cuda ) const  [inline]
```

Runs one of the passed in functors, depending on the Executor type.

Template Parameters

ClosureOmp	type of op_omp
ClosureCuda	type of op_cuda

Parameters

op_omp	functor to run in case of a OmpExecutor or ReferenceExecutor
op_cuda	functor to run in case of a CudaExecutor

References run().

8.40.2.7 run() [2/2]

```
virtual void gko::Executor::run (
            const Operation & op ) const  [pure virtual]
```

Runs the specified Operation using this Executor.

Parameters

op	the operation to run

Implemented in gko::CudaExecutor, and gko::ReferenceExecutor.

Referenced by run().

The documentation for this class was generated from the following file:

- ginkgo/core/base/executor.hpp (f1a4eb68)

8.41 gko::log::executor_data Struct Reference

Struct representing Executor related data.

```
#include <ginkgo/core/log/record.hpp>
```

8.41.1 Detailed Description

Struct representing Executor related data.

The documentation for this struct was generated from the following file:

- ginkgo/core/log/record.hpp (f0a50f96)

8.42 gko::executor_deleter< T > Class Template Reference

This is a deleter that uses an executor's `free` method to deallocate the data.

`#include <ginkgo/core/base/executor.hpp>`

Public Member Functions

- executor_deleter (std::shared_ptr< const Executor > exec)
  Creates a new deleter.
- void operator() (pointer ptr) const
  Deletes the object.

8.42.1 Detailed Description

template<typename T>
class gko::executor_deleter< T >

This is a deleter that uses an executor's `free` method to deallocate the data.

Template Parameters

T	the type of object being deleted

8.42.2 Constructor & Destructor Documentation

8.42.2.1 executor_deleter()

```
template<typename T >
gko::executor_deleter< T >::executor_deleter (
            std::shared_ptr< const Executor > exec )  [inline], [explicit]
```

Creates a new deleter.

Parameters

exec	the executor used to free the data

8.42.3 Member Function Documentation

8.42.3.1 operator()()

```
template<typename T >
void gko::executor_deleter< T >::operator() (
            pointer ptr ) const  [inline]
```

Deletes the object.

Parameters

ptr	pointer to the object being deleted

The documentation for this class was generated from the following file:

- ginkgo/core/base/executor.hpp (f1a4eb68)

8.43 gko::solver::Fcg< ValueType > Class Template Reference

FCG or the flexible conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.

```
#include <ginkgo/core/solver/fcg.hpp>
```

Public Member Functions

- std::shared_ptr< const LinOp > get_system_matrix () const

  Gets the system operator (matrix) of the linear system.

8.43.1 Detailed Description

template<typename ValueType = default_precision>
class gko::solver::Fcg< ValueType >

FCG or the flexible conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.

Though this method performs very well for symmetric positive definite matrices, it is in general not suitable for general matrices.

In contrast to the standard CG based on the Polack-Ribiere formula, the flexible CG uses the Fletcher-Reeves formula for creating the orthonormal vectors spanning the Krylov subspace. This increases the computational cost of every Krylov solver iteration but allows for non-constant preconditioners.

The implementation in Ginkgo makes use of the merged kernel to make the best use of data locality. The inner operations in one iteration of FCG are merged into 2 separate steps.

Template Parameters

ValueType	precision of matrix elements

8.43.2 Member Function Documentation

8.43.2.1 get_system_matrix()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Fcg< ValueType >::get_system_matrix ( ) const  [inline]
```

Gets the system operator (matrix) of the linear system.

Returns

the system operator (matrix)

```
90     {
91         return system_matrix_;
92     }
```

The documentation for this class was generated from the following file:

- ginkgo/core/solver/fcg.hpp (c380ba80)

8.44 gko::solver::Gmres< ValueType > Class Template Reference

GMRES or the generalized minimal residual method is an iterative type Krylov subspace method which is suitable for nonsymmetric linear systems.

```
#include <ginkgo/core/solver/gmres.hpp>
```

Public Member Functions

- std::shared_ptr< const LinOp > get_system_matrix () const
  Gets the system operator (matrix) of the linear system.
- size_type get_krylov_dim () const
  Returns the krylov dimension.

8.44.1 Detailed Description

template<typename ValueType = default_precision>
class gko::solver::Gmres< ValueType >

GMRES or the generalized minimal residual method is an iterative type Krylov subspace method which is suitable for nonsymmetric linear systems.

The implementation in Ginkgo makes use of the merged kernel to make the best use of data locality. The inner operations in one iteration of GMRES are merged into 2 separate steps.

Template Parameters

ValueType	precision of matrix elements

8.44.2 Member Function Documentation

8.44.2.1 get_krylov_dim()

```
template<typename ValueType = default_precision>
size_type gko::solver::Gmres< ValueType >::get_krylov_dim ( ) const  [inline]
```

Returns the krylov dimension.

Returns

the krylov dimension

```
94 { return krylov_dim_; }
```

8.44.2.2 get_system_matrix()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Gmres< ValueType >::get_system_matrix ( ) const
[inline]
```

Gets the system operator (matrix) of the linear system.

Returns

the system operator (matrix)

The documentation for this class was generated from the following file:

- ginkgo/core/solver/gmres.hpp (c380ba80)

8.45 gko::matrix::Hybrid< ValueType, IndexType > Class Template Reference

HYBRID is a matrix format which splits the matrix into ELLPACK and COO format.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Classes

- class automatic

  automatic is a stratgy_type which decides the number of stored elements per row of the ell part automatically.
- class column_limit

  column_limit is a strategy_type which decides the number of stored elements per row of the ell part by specifying the number of columns.
- class imbalance_bounded_limit

  imbalance_bounded_limit is a stratgy_type which decides the number of stored elements per row of the ell part.
- class imbalance_limit

  imbalance_limit is a strategy_type which decides the number of stored elements per row of the ell part according to the percent.
- class minimal_storage_limit

  minimal_storage_limit is a stratgy_type which decides the number of stored elements per row of the ell part.
- class strategy_type

  strategy_type is to decide how to set the hybrid config.

Public Member Functions

- void read (const mat_data &data) override

  Reads a matrix from a matrix_data structure.
- void write (mat_data &data) const override

  Writes a matrix to a matrix_data structure.
- value_type * get_ell_values () noexcept

  Returns the values of the ell part.
- const value_type * get_const_ell_values () const noexcept

  Returns the values of the ell part.
- index_type * get_ell_col_idxs () noexcept

  Returns the column indexes of the ell part.
- const index_type * get_const_ell_col_idxs () const noexcept

  Returns the column indexes of the ell part.
- size_type get_ell_num_stored_elements_per_row () const noexcept

  Returns the number of stored elements per row of ell part.
- size_type get_ell_stride () const noexcept

  Returns the stride of the ell part.
- size_type get_ell_num_stored_elements () const noexcept

  Returns the number of elements explicitly stored in the ell part.
- value_type & ell_val_at (size_type row, size_type idx) noexcept

  Returns the `idx`-th non-zero element of the `row`-th row in the ell part.
- value_type ell_val_at (size_type row, size_type idx) const noexcept

  Returns the `idx`-th non-zero element of the `row`-th row in the ell part.
- index_type & ell_col_at (size_type row, size_type idx) noexcept

  Returns the `idx`-th column index of the `row`-th row in the ell part.
- index_type ell_col_at (size_type row, size_type idx) const noexcept

  Returns the `idx`-th column index of the `row`-th row in the ell part.
- const ell_type * get_ell () const noexcept

  Returns the matrix of the ell part.
- value_type * get_coo_values () noexcept

  Returns the values of the coo part.
- const value_type * get_const_coo_values () const noexcept

  Returns the values of the coo part.

- index_type * get_coo_col_idxs () noexcept
  Returns the column indexes of the coo part.
- const index_type * get_const_coo_col_idxs () const noexcept
  Returns the column indexes of the coo part.
- index_type * get_coo_row_idxs () noexcept
  Returns the row indexes of the coo part.
- const index_type * get_const_coo_row_idxs () const noexcept
  Returns the row indexes of the coo part.
- size_type get_coo_num_stored_elements () const noexcept
  Returns the number of elements explicitly stored in the coo part.
- const coo_type * get_coo () const noexcept
  Returns the matrix of the coo part.
- size_type get_num_stored_elements () const noexcept
  Returns the number of elements explicitly stored in the matrix.
- std::shared_ptr< strategy_type > get_strategy () const noexcept
  Returns the strategy.
- Hybrid & operator= (const Hybrid &other)
  Copies data from another Hybrid.

8.45.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >

HYBRID is a matrix format which splits the matrix into ELLPACK and COO format.

Achieve the excellent performance with a proper partition of ELLPACK and COO.

Template Parameters

ValueType	precision of matrix elements
IndexType	precision of matrix indexes

8.45.2 Member Function Documentation

8.45.2.1 ell_col_at() [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type gko::matrix::Hybrid< ValueType, IndexType >::ell_col_at (
            size_type row,
            size_type idx ) const  [inline], [noexcept]
```

Returns the `idx`-th column index of the `row`-th row in the ell part.

Parameters

row	the row of the requested element
idx	the idx-th stored element of the row

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

```
453     {
454         return ell_->col_at(row, idx);
455     }
```

8.45.2.2 ell_col_at() [2/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type& gko::matrix::Hybrid< ValueType, IndexType >::ell_col_at (
             size_type row,
             size_type idx )  [inline], [noexcept]
```

Returns the `idx`-th column index of the `row`-th row in the ell part.

Parameters

row	the row of the requested element
idx	the idx-th stored element of the row

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

8.45.2.3 ell_val_at() [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type gko::matrix::Hybrid< ValueType, IndexType >::ell_val_at (
             size_type row,
             size_type idx ) const  [inline], [noexcept]
```

Returns the `idx`-th non-zero element of the `row`-th row in the ell part.

Parameters

row	the row of the requested element
idx	the idx-th stored element of the row

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

8.45.2.4 ell_val_at() [2/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type& gko::matrix::Hybrid< ValueType, IndexType >::ell_val_at (
            size_type row,
            size_type idx ) [inline], [noexcept]
```

Returns the `idx`-th non-zero element of the `row`-th row in the ell part.

Parameters

row	the row of the requested element
idx	the idx-th stored element of the row

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

8.45.2.5 get_const_coo_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Hybrid< ValueType, IndexType >::get_const_coo_col_idxs ( )
const [inline], [noexcept]
```

Returns the column indexes of the coo part.

Returns

the column indexes of the coo part.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

8.45.2.6 get_const_coo_row_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Hybrid< ValueType, IndexType >::get_const_coo_row_idxs ( )
const  [inline], [noexcept]
```

Returns the row indexes of the coo part.

Returns

the row indexes of the coo part.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

8.45.2.7 get_const_coo_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::Hybrid< ValueType, IndexType >::get_const_coo_values ( ) const
[inline], [noexcept]
```

Returns the values of the coo part.

Returns

the values of the coo part.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

8.45.2.8 get_const_ell_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Hybrid< ValueType, IndexType >::get_const_ell_col_idxs ( )
const  [inline], [noexcept]
```

Returns the column indexes of the ell part.

Returns

the column indexes of the ell part

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

8.45.2.9 get_const_ell_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::Hybrid< ValueType, IndexType >::get_const_ell_values ( ) const
[inline], [noexcept]
```

Returns the values of the ell part.

Returns

the values of the ell part

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

8.45.2.10 get_coo()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const coo_type* gko::matrix::Hybrid< ValueType, IndexType >::get_coo ( ) const  [inline],
[noexcept]
```

Returns the matrix of the coo part.

Returns

the matrix of the coo part

8.45.2.11 get_coo_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Hybrid< ValueType, IndexType >::get_coo_col_idxs ( )  [inline], [noexcept]
```

Returns the column indexes of the coo part.

Returns

the column indexes of the coo part.

8.45.2.12 get_coo_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::get_coo_num_stored_elements ( ) const
[inline], [noexcept]
```

Returns the number of elements explicitly stored in the coo part.

Returns

the number of elements explicitly stored in the coo part

8.45.2.13 get_coo_row_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Hybrid< ValueType, IndexType >::get_coo_row_idxs ( )  [inline], [noexcept]
```

Returns the row indexes of the coo part.

Returns

the row indexes of the coo part.

8.45.2.14 get_coo_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::Hybrid< ValueType, IndexType >::get_coo_values ( )  [inline], [noexcept]
```

Returns the values of the coo part.

Returns

the values of the coo part.

8.45.2.15 get_ell()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const ell_type* gko::matrix::Hybrid< ValueType, IndexType >::get_ell ( ) const  [inline],
[noexcept]
```

Returns the matrix of the ell part.

Returns

the matrix of the ell part

8.45.2.16 get_ell_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Hybrid< ValueType, IndexType >::get_ell_col_idxs ( )  [inline], [noexcept]
```

Returns the column indexes of the ell part.

Returns

the column indexes of the ell part

8.45.2.17 get_ell_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::get_ell_num_stored_elements ( ) const
[inline], [noexcept]
```

Returns the number of elements explicitly stored in the ell part.

Returns

the number of elements explicitly stored in the ell part

8.45.2.18 get_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::get_ell_num_stored_elements_per_row ( )
const  [inline], [noexcept]
```

Returns the number of stored elements per row of ell part.

Returns

the number of stored elements per row of ell part

8.45.2.19 get_ell_stride()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::get_ell_stride ( ) const  [inline],
[noexcept]
```

Returns the stride of the ell part.

Returns

the stride of the ell part

8.45.2.20 get_ell_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::Hybrid< ValueType, IndexType >::get_ell_values ( )  [inline], [noexcept]
```

Returns the values of the ell part.

Returns

the values of the ell part

8.45.2.21 get_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::get_num_stored_elements ( ) const
[inline], [noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

8.45.2.22 get_strategy()

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::shared_ptr<strategy_type> gko::matrix::Hybrid< ValueType, IndexType >::get_strategy ( )
const  [inline], [noexcept]
```

Returns the strategy.

Returns

the strategy

8.45.2.23 operator=()

```
template<typename ValueType = default_precision, typename IndexType = int32>
Hybrid& gko::matrix::Hybrid< ValueType, IndexType >::operator= (
            const Hybrid< ValueType, IndexType > & other )  [inline]
```

Copies data from another Hybrid.

Parameters

other	the Hybrid to copy from

Returns

this

8.45.2.24 read()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Hybrid< ValueType, IndexType >::read (
            const mat_data & data )  [override], [virtual]
```

Reads a matrix from a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::ReadableFromMatrixData< ValueType, IndexType >.

8.45.2.25 write()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Hybrid< ValueType, IndexType >::write (
            mat_data & data ) const  [override], [virtual]
```

Writes a matrix to a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::WritableToMatrixData< ValueType, IndexType >.

The documentation for this class was generated from the following files:

- ginkgo/core/matrix/csr.hpp (b8bd4773)
- ginkgo/core/matrix/hybrid.hpp (3e51a52b)

8.46 gko::matrix::Identity< ValueType > Class Template Reference

This class is a utility which efficiently implements the identity matrix (a linear operator which maps each vector to itself).

`#include <ginkgo/core/matrix/identity.hpp>`

Additional Inherited Members

8.46.1 Detailed Description

template<typename ValueType = default_precision>
class gko::matrix::Identity< ValueType >

This class is a utility which efficiently implements the identity matrix (a linear operator which maps each vector to itself).

Thus, objects of the Identity class always represent a square matrix, and don't require any storage for their values. The apply method is implemented as a simple copy (or a linear combination).

Note

This class is useful when composing it with other operators. For example, it can be used instead of a preconditioner in Krylov solvers, if one wants to run a "plain" solver, without using a preconditioner.

Template Parameters

ValueType	precision of matrix elements

The documentation for this class was generated from the following file:

- ginkgo/core/matrix/identity.hpp (8045ac75)

8.47 gko::matrix::IdentityFactory< ValueType > Class Template Reference

This factory is a utility which can be used to generate Identity operators.

`#include <ginkgo/core/matrix/identity.hpp>`

Static Public Member Functions

- static std::unique_ptr< IdentityFactory > create (std::shared_ptr< const Executor > exec)
  Creates a new Identity factory.

Additional Inherited Members

8.47.1 Detailed Description

template<typename ValueType = default_precision>
class gko::matrix::IdentityFactory< ValueType >

This factory is a utility which can be used to generate Identity operators.

The factory will generate the Identity matrix with the same dimension as the passed in operator. It will throw an exception if the operator is not square.

Template Parameters

ValueType	precision of matrix elements

8.47.2 Member Function Documentation

8.47.2.1 create()

```
template<typename ValueType = default_precision>
static std::unique_ptr<IdentityFactory> gko::matrix::IdentityFactory< ValueType >::create (
            std::shared_ptr< const Executor > exec )  [inline], [static]
```

Creates a new Identity factory.

Parameters

exec	the executor where the Identity operator will be stored

Returns

a unique pointer to the newly created factory

```
129     {
130         return std::unique_ptr<IdentityFactory>(
131             new IdentityFactory(std::move(exec)));
132     }
```

The documentation for this class was generated from the following file:

- ginkgo/core/matrix/identity.hpp (8045ac75)

8.48 gko::preconditioner::Ilu< LSolverType, USolverType, ReverseApply, IndexTypeParIlu > Class Template Reference

The Incomplete LU (ILU) preconditioner solves the equation $LUx = b$ for a given lower triangular matrix L, an upper triangular matrix U and the right hand side b (can contain multiple right hand sides).

```
#include <ginkgo/core/preconditioner/ilu.hpp>
```

Public Member Functions

- std::shared_ptr< const l_solver_type > get_l_solver () const

  Returns the solver which is used for the provided L matrix.
- std::shared_ptr< const u_solver_type > get_u_solver () const

  Returns the solver which is used for the provided U matrix.

8.48.1 Detailed Description

template<typename LSolverType = solver::LowerTrs<>, typename USolverType = solver::UpperTrs<>, bool ReverseApply = false, typename IndexTypeParIlu = int32>
class gko::preconditioner::Ilu< LSolverType, USolverType, ReverseApply, IndexTypeParIlu >

The Incomplete LU (ILU) preconditioner solves the equation $LUx = b$ for a given lower triangular matrix L, an upper triangular matrix U and the right hand side b (can contain multiple right hand sides).

It allows to set both the solver for L and the solver for U independently, while providing the defaults solver::LowerTrs and solver::UpperTrs, which are direct triangular solvers. For these solvers, a factory can be provided (with `with_l_solver_factory` and `with_u_solver_factory`) to have more control over their behavior. In particular, it is possible to use an iterative method for solving the triangular systems. The default parameters for an iterative triangluar solver are:

- reduction factor = 1e-4
- max iteration = <number of="" rows="" of="" the="" matrix="" given="" to="" the="" solver>=""> Solvers without such criteria can also be used, in which case none are set.

An object of this class can be created with a matrix or a gko::Composition containing two matrices. If created with a matrix, it is factorized before creating the solver. If a gko::Composition (containing two matrices) is used, the first operand will be taken as the L matrix, the second will be considered the U matrix. ParIlu can be directly used, since it orders the factors in the correct way.

Note

When providing a gko::Composition, the first matrix must be the lower matrix (L), and the second matrix must be the upper matrix (U). If they are swapped, solving might crash or return the wrong result.

Do not use symmetric solvers (like CG) for L or U solvers since both matrices (L and U) are, by design, not symmetric.

This class is not thread safe (even a const object is not) because it uses an internal cache to accelerate multiple (sequential) applies. Using it in parallel can lead to segmentation faults, wrong results and other unwanted behavior.

Template Parameters

LSolverType	type of the solver used for the L matrix. Defaults to solver::LowerTrs
USolverType	type of the solver used for the U matrix Defaults to solver::UpperTrs
ReverseApply	default behavior (ReverseApply = false) is first to solve with L (Ly = b) and then with U (Ux = y). When set to true, it will solve first with U, and then with L.
IndexTypeParIlu	Type of the indices when ParIlu is used to generate both L and U factors. Irrelevant otherwise.

8.48.2 Member Function Documentation

8.48.2.1 get_l_solver()

```
template<typename LSolverType = solver::LowerTrs<>, typename USolverType = solver::Upper↩
Trs<>, bool ReverseApply = false, typename IndexTypeParIlu = int32>
std::shared_ptr<const l_solver_type> gko::preconditioner::Ilu< LSolverType, USolverType,
ReverseApply, IndexTypeParIlu >::get_l_solver ( ) const  [inline]
```

Returns the solver which is used for the provided L matrix.

Returns

the solver which is used for the provided L matrix

```
152     {
153         return l_solver_;
154     }
```

8.48.2.2 get_u_solver()

```
template<typename LSolverType = solver::LowerTrs<>, typename USolverType = solver::Upper↩
Trs<>, bool ReverseApply = false, typename IndexTypeParIlu = int32>
std::shared_ptr<const u_solver_type> gko::preconditioner::Ilu< LSolverType, USolverType,
ReverseApply, IndexTypeParIlu >::get_u_solver ( ) const  [inline]
```

Returns the solver which is used for the provided U matrix.

Returns

the solver which is used for the provided U matrix

The documentation for this class was generated from the following file:

- ginkgo/core/preconditioner/ilu.hpp (7160b772)

8.49 gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit Class Reference

imbalance_bounded_limit is a stratgy_type which decides the number of stored elements per row of the ell part.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Public Member Functions

- imbalance_bounded_limit (float percent=0.8, float ratio=0.0001)

  Creates a imbalance_bounded_limit strategy.
- size_type compute_ell_num_stored_elements_per_row (Array< size_type > *row_nnz) const override

  Computes the number of stored elements per row of the ell part.

8.49.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit

imbalance_bounded_limit is a stratgy_type which decides the number of stored elements per row of the ell part.

It uses the imbalance_limit and adds the upper bound of the number of ell's cols by the number of rows.

8.49.2 Member Function Documentation

8.49.2.1 compute_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit::compute_ell_↩
num_stored_elements_per_row (
            Array< size_type > * row_nnz ) const  [inline], [override], [virtual]
```

Computes the number of stored elements per row of the ell part.

Parameters

row_nnz	the number of nonzeros of each row

Returns

the number of stored elements per row of the ell part

Implements gko::matrix::Hybrid< ValueType, IndexType >::strategy_type.

References gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit::compute_ell_num_stored_elements↩
_per_row(), and gko::Array< ValueType >::get_num_elems().

Referenced by gko::matrix::Hybrid< ValueType, IndexType >::automatic::compute_ell_num_stored_elements_↩
per_row().

The documentation for this class was generated from the following file:

- ginkgo/core/matrix/hybrid.hpp (3e51a52b)

8.50 gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit Class Reference

imbalance_limit is a strategy_type which decides the number of stored elements per row of the ell part according to the percent.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Public Member Functions

- imbalance_limit (float percent=0.8)

  Creates a imbalance_limit strategy.
- size_type compute_ell_num_stored_elements_per_row (Array< size_type > *row_nnz) const override

  Computes the number of stored elements per row of the ell part.

8.50.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit

imbalance_limit is a strategy_type which decides the number of stored elements per row of the ell part according to the percent.

It sorts the number of nonzeros of each row and takes the value at the position `floor(percent * num_row)` as the number of stored elements per row of the ell part. Thus, at least `percent` rows of all are in the ell part.

8.50.2 Constructor & Destructor Documentation

8.50.2.1 imbalance_limit()

```
template<typename ValueType = default_precision, typename IndexType = int32>
gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit::imbalance_limit (
            float percent = 0.8 )  [inline], [explicit]
```

Creates a imbalance_limit strategy.

Parameters

percent	the row_nnz[floor(num_rows*percent)] is the number of stored elements per row of the ell part

8.50.3 Member Function Documentation

8.50.3.1 compute_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit::compute_ell_num_↩
stored_elements_per_row (
            Array< size_type > * row_nnz ) const  [inline], [override], [virtual]
```

Computes the number of stored elements per row of the ell part.

Parameters

row_nnz	the number of nonzeros of each row

Returns

the number of stored elements per row of the ell part

Implements gko::matrix::Hybrid< ValueType, IndexType >::strategy_type.

References gko::Array< ValueType >::get_data(), and gko::Array< ValueType >::get_num_elems().

Referenced by gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit::compute_ell_num_↩ stored_elements_per_row(), and gko::matrix::Hybrid< ValueType, IndexType >::minimal_storage_limit::compute↩ _ell_num_stored_elements_per_row().

The documentation for this class was generated from the following file:

- ginkgo/core/matrix/hybrid.hpp (3e51a52b)

8.51 gko::solver::Ir< ValueType > Class Template Reference

Iterative refinement (IR) is an iterative method that uses another coarse method to approximate the error of the current solution via the current residual.

`#include <ginkgo/core/solver/ir.hpp>`

Public Member Functions

- std::shared_ptr< const LinOp > get_system_matrix () const

  Returns the system operator (matrix) of the linear system.
- std::shared_ptr< const LinOp > get_solver () const

  Returns the solver operator used as the inner solver.
- void set_solver (std::shared_ptr< const LinOp > new_solver)

  Sets the solver operator used as the inner solver.

8.51.1 Detailed Description

template<typename ValueType = default_precision>
class gko::solver::Ir< ValueType >

Iterative refinement (IR) is an iterative method that uses another coarse method to approximate the error of the current solution via the current residual.

For any approximation of the solution `solution` to the system `Ax = b`, the residual is defined as: `residual = b - A solution`. The error in `solution`, `e = x - solution` (with `x` being the exact solution) can be obtained as the solution to the residual equation `Ae = residual`, since `A e = Ax - A solution = b - A solution = residual`. Then, the real solution is computed as `x = solution + e`. Instead of accurately solving the residual equation `Ae = residual`, the solution of the system `e` can be approximated to obtain the approximation `error` using a coarse method `solver`, which is used to update `solution`, and the entire process is repeated with the updated `solution`. This yields the iterative refinement method:

```
solution = initial_guess
while not converged:
    residual = b - A solution
    error = solver(A, residual)
    solution = solution + error
```

Assuming that `solver` has accuracy `c`, i.e., `| e - error | <= c | e |`, iterative refinement will converge with a convergence rate of `c`. Indeed, from `e - error = x - solution - error = x - solution*` (where `solution*` denotes the value stored in `solution` after the update) and `e = inv(A) residual = inv(A)b - inv(A) A solution = x - solution` it follows that | x - solution* | <= c | x - solution |.

Unless otherwise specified via the `solver` factory parameter, this implementation uses the identity operator (i.e. the solver that approximates the solution of a system Ax = b by setting x := b) as the default inner solver. Such a setting results in a relaxation method known as the Richardson iteration with parameter 1, which is guaranteed to converge for matrices whose spectrum is strictly contained within the unit disc around 1 (i.e., all its eigenvalues `lambda` have to satisfy the equation `|lambda - 1| < 1).

Template Parameters

ValueType	precision of matrix elements

8.51.2 Member Function Documentation

8.51.2.1 get_solver()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Ir< ValueType >::get_solver ( ) const  [inline]
```

Returns the solver operator used as the inner solver.

Returns

the solver operator used as the inner solver

```
117 { return solver_; }
```

8.51.2.2 get_system_matrix()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Ir< ValueType >::get_system_matrix ( ) const  [inline]
```

Returns the system operator (matrix) of the linear system.

Returns

the system operator (matrix)

8.51.2.3 set_solver()

```
template<typename ValueType = default_precision>
void gko::solver::Ir< ValueType >::set_solver (
            std::shared_ptr< const LinOp > new_solver )  [inline]
```

Sets the solver operator used as the inner solver.

Parameters

new_solver	the new inner solver

The documentation for this class was generated from the following file:

- ginkgo/core/solver/ir.hpp (4147c548)

8.52 gko::stop::Iteration Class Reference

The Iteration class is a stopping criterion which stops the iteration process after a preset number of iterations.

```
#include <ginkgo/core/stop/iteration.hpp>
```

8.52.1 Detailed Description

The Iteration class is a stopping criterion which stops the iteration process after a preset number of iterations.

Note

to use this stopping criterion, it is required to update the iteration count for the ::check() method.

The documentation for this class was generated from the following file:

- ginkgo/core/stop/iteration.hpp (f1a4eb68)

8.53 gko::log::iteration_complete_data Struct Reference

Struct representing iteration complete related data.

`#include <ginkgo/core/log/record.hpp>`

8.53.1 Detailed Description

Struct representing iteration complete related data.

The documentation for this struct was generated from the following file:

- ginkgo/core/log/record.hpp (f0a50f96)

8.54 gko::preconditioner::Jacobi< ValueType, IndexType > Class Template Reference

A block-Jacobi preconditioner is a block-diagonal linear operator, obtained by inverting the diagonal blocks of the source operator.

`#include <ginkgo/core/preconditioner/jacobi.hpp>`

Public Member Functions

- size_type get_num_blocks () const noexcept

  Returns the number of blocks of the operator.
- const block_interleaved_storage_scheme< index_type > & get_storage_scheme () const noexcept

  Returns the storage scheme used for storing Jacobi blocks.
- const value_type * get_blocks () const noexcept

  Returns the pointer to the memory used for storing the block data.
- const remove_complex< value_type > * get_conditioning () const noexcept

  Returns an array of 1-norm condition numbers of the blocks.
- size_type get_num_stored_elements () const noexcept

  Returns the number of elements explicitly stored in the matrix.
- void convert_to (matrix::Dense< value_type > *result) const override

  Converts the implementer to an object of type result_type.
- void move_to (matrix::Dense< value_type > *result) override

  Converts the implementer to an object of type result_type by moving data from this object.
- void write (mat_data &data) const override

  Writes a matrix to a matrix_data structure.

8.54.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::preconditioner::Jacobi< ValueType, IndexType >

A block-Jacobi preconditioner is a block-diagonal linear operator, obtained by inverting the diagonal blocks of the source operator.

The Jacobi class implements the inversion of the diagonal blocks using Gauss-Jordan elimination with column pivoting, and stores the inverse explicitly in a customized format.

If the diagonal blocks of the matrix are not explicitly set by the user, the implementation will try to automatically detect the blocks by first finding the natural blocks of the matrix, and then applying the supervariable agglomeration procedure on them. However, if problem-specific knowledge regarding the block diagonal structure is available, it is usually beneficial to explicitly pass the starting rows of the diagonal blocks, as the block detection is merely a heuristic and cannot perfectly detect the diagonal block structure. The current implementation supports blocks of up to 32 rows / columns.

The implementation also includes an improved, adaptive version of the block-Jacobi preconditioner, which can store some of the blocks in lower precision and thus improve the performance of preconditioner application by reducing the amount of memory transfers. This variant can be enabled by setting the Jacobi::Factory's `storage_optimization` parameter. Refer to the documentation of the parameter for more details.

Template Parameters

ValueType	precision of matrix elements
IndexType	integral type used to store pointers to the start of each block

Note

The current implementation supports blocks of up to 32 rows / columns.

When using the adaptive variant, there may be a trade-off in terms of slightly longer preconditioner generation due to extra work required to detect the optimal precision of the blocks.

8.54.2 Member Function Documentation

8.54.2.1 convert_to()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::preconditioner::Jacobi< ValueType, IndexType >::convert_to (
            matrix::Dense< value_type > * result ) const  [override], [virtual]
```

Converts the implementer to an object of type result_type.

Parameters

result	the object used to store the result of the conversion

Implements gko::ConvertibleTo< matrix::Dense< ValueType > >.

8.54.2.2 get_blocks()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::preconditioner::Jacobi< ValueType, IndexType >::get_blocks ( ) const
[inline], [noexcept]
```

Returns the pointer to the memory used for storing the block data.

Element (`i`, `j`) of block `b` is stored in position `(get_block_pointers()[b] + i) * stride + j` of the array.

Returns

the pointer to the memory used for storing the block data

References gko::Array< ValueType >::get_const_data().

8.54.2.3 get_conditioning()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const remove_complex<value_type>* gko::preconditioner::Jacobi< ValueType, IndexType >::get_↩
conditioning ( ) const  [inline], [noexcept]
```

Returns an array of 1-norm condition numbers of the blocks.

Returns

an array of 1-norm condition numbers of the blocks

Note

This value is valid only if adaptive precision variant is used, and implementations of the standard non-adaptive variant are allowed to omit the calculation of condition numbers.

References gko::Array< ValueType >::get_const_data().

8.54.2.4 get_num_blocks()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_blocks ( ) const  [inline],
[noexcept]
```

Returns the number of blocks of the operator.

Returns

the number of blocks of the operator

8.54.2.5 get_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_elements ( )
const  [inline], [noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

References gko::Array< ValueType >::get_num_elems().

8.54.2.6 get_storage_scheme()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const block_interleaved_storage_scheme<index_type>& gko::preconditioner::Jacobi< ValueType,
IndexType >::get_storage_scheme ( ) const  [inline], [noexcept]
```

Returns the storage scheme used for storing Jacobi blocks.

Returns

the storage scheme used for storing Jacobi blocks

8.54.2.7 move_to()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::preconditioner::Jacobi< ValueType, IndexType >::move_to (
            matrix::Dense< value_type > * result )  [override], [virtual]
```

Converts the implementer to an object of type result_type by moving data from this object.

This method is used when the implementer is a temporary object, and move semantics can be used.

Parameters

result	the object used to emplace the result of the conversion

Note

ConvertibleTo::move_to can be implemented by simply calling ConvertibleTo::convert_to. However, this operation can often be optimized by exploiting the fact that implementer's data can be moved to the result.

Implements gko::ConvertibleTo< matrix::Dense< ValueType > >.

8.54.2.8 write()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::preconditioner::Jacobi< ValueType, IndexType >::write (
            mat_data & data ) const  [override], [virtual]
```

Writes a matrix to a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::WritableToMatrixData< ValueType, IndexType >.

The documentation for this class was generated from the following file:

- ginkgo/core/preconditioner/jacobi.hpp (9c2e5ae6)

8.55 gko::KernelNotFound Class Reference

KernelNotFound is thrown if Ginkgo cannot find a kernel which satisfies the criteria imposed by the input arguments.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- KernelNotFound (const std::string &file, int line, const std::string &func)

  Initializes a KernelNotFound error.

8.55.1 Detailed Description

KernelNotFound is thrown if Ginkgo cannot find a kernel which satisfies the criteria imposed by the input arguments.

8.55.2 Constructor & Destructor Documentation

8.55.2.1 KernelNotFound()

```
gko::KernelNotFound::KernelNotFound (
            const std::string & file,
            int line,
            const std::string & func )  [inline]
```

Initializes a KernelNotFound error.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
func	The name of the function where the error occurred

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.56 gko::log::linop_data Struct Reference

Struct representing LinOp related data.

`#include <ginkgo/core/log/record.hpp>`

8.56.1 Detailed Description

Struct representing LinOp related data.

The documentation for this struct was generated from the following file:

- ginkgo/core/log/record.hpp (f0a50f96)

8.57 gko::log::linop_factory_data Struct Reference

Struct representing LinOp factory related data.

`#include <ginkgo/core/log/record.hpp>`

8.57.1 Detailed Description

Struct representing LinOp factory related data.

The documentation for this struct was generated from the following file:

- ginkgo/core/log/record.hpp (f0a50f96)

8.58 gko::LinOpFactory Class Reference

A LinOpFactory represents a higher order mapping which transforms one linear operator into another.

`#include <ginkgo/core/base/lin_op.hpp>`

Additional Inherited Members

8.58.1 Detailed Description

A LinOpFactory represents a higher order mapping which transforms one linear operator into another.

In Ginkgo, every linear solver is viewed as a mapping. For example, given an s.p.d linear system $Ax = b$, the solution $x = A^{-1}b$ can be computed using the CG method. This algorithm can be represented in terms of linear operators and mappings between them as follows:

- A Cg::Factory is a higher order mapping which, given an input operator A, returns a new linear operator A^{-1} stored in "CG format"
- Storing the operator A^{-1} in "CG format" means that the data structure used to store the operator is just a simple pointer to the original matrix A. The application $x = A^{-1}b$ of such an operator can then be implemented by solving the linear system $Ax = b$ using the CG method. This is achieved in code by having a special class for each of those "formats" (e.g. the "Cg" class defines such a format for the CG solver).

Another example of a LinOpFactory is a preconditioner. A preconditioner for a linear operator A is a linear operator M^{-1}, which approximates A^{-1}. In addition, it is stored in a way such that both the data of M^{-1} is cheap to compute from A, and the operation $x = M^{-1}b$ can be computed quickly. These operators are useful to accelerate the convergence of Krylov solvers. Thus, a preconditioner also fits into the LinOpFactory framework:

- The factory maps a linear operator A into a preconditioner M^{-1} which is stored in suitable format (e.g. as a product of two factors in case of ILU preconditioners).
- The resulting linear operator implements the application operation $x = M^{-1}b$ depending on the format the preconditioner is stored in (e.g. as two triangular solves in case of ILU)

8.58.1.1 Example: using CG in Ginkgo

```
 {c++}
// Suppose A is a matrix, b a rhs vector, and x an initial guess
// Create a CG which runs for at most 1000 iterations, and stops after
// reducing the residual norm by 6 orders of magnitude
auto cg_factory = solver::Cg<>::build()
    .with_max_iters(1000)
    .with_rel_residual_goal(1e-6)
    .on(cuda);
// create a linear operator which represents the solver
auto cg = cg_factory->generate(A);
// solve the system
cg->apply(gko::lend(b), gko::lend(x));
```

The documentation for this class was generated from the following file:

- ginkgo/core/base/lin_op.hpp (4c758394)

8.59 gko::log::Loggable Class Reference

Loggable class is an interface which should be implemented by classes wanting to support logging.

`#include <ginkgo/core/log/logger.hpp>`

Public Member Functions

- virtual void add_logger (std::shared_ptr< const Logger > logger)=0

  Adds a new logger to the list of subscribed loggers.
- virtual void remove_logger (const Logger *logger)=0

  Removes a logger from the list of subscribed loggers.

8.59.1 Detailed Description

Loggable class is an interface which should be implemented by classes wanting to support logging.

For most cases, one can rely on the EnableLogging mixin which provides a default implementation of this interface.

8.59.2 Member Function Documentation

8.59.2.1 add_logger()

```
virtual void gko::log::Loggable::add_logger (
            std::shared_ptr< const Logger > logger )  [pure virtual]
```

Adds a new logger to the list of subscribed loggers.

Parameters

logger	the logger to add

Implemented in gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase >, gko::log::EnableLogging< PolymorphicObject >, and gko::log::EnableLogging< Executor >.

8.59.2.2 remove_logger()

```
virtual void gko::log::Loggable::remove_logger (
            const Logger * logger )  [pure virtual]
```

Removes a logger from the list of subscribed loggers.

Parameters

logger	the logger to remove

Note

The comparison is done using the logger's object unique identity. Thus, two loggers constructed in the same way are not considered equal.

Implemented in gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase >, gko::log::EnableLogging< PolymorphicObject >, and gko::log::EnableLogging< Executor >.

The documentation for this class was generated from the following file:

- ginkgo/core/log/logger.hpp (0d7578c9)

8.60 gko::log::Record::logged_data Struct Reference

Struct storing the actually logged data.

`#include <ginkgo/core/log/record.hpp>`

8.60.1 Detailed Description

Struct storing the actually logged data.

The documentation for this struct was generated from the following file:

- ginkgo/core/log/record.hpp (f0a50f96)

8.61 gko::solver::LowerTrs< ValueType, IndexType > Class Template Reference

LowerTrs is the triangular solver which solves the system L x = b, when L is a lower triangular matrix.

`#include <ginkgo/core/solver/lower_trs.hpp>`

Public Member Functions

- std::shared_ptr< const matrix::Csr< ValueType, IndexType > > get_system_matrix () const

  Gets the system operator (CSR matrix) of the linear system.

8.61.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::solver::LowerTrs< ValueType, IndexType >

LowerTrs is the triangular solver which solves the system L x = b, when L is a lower triangular matrix.

It works best when passing in a matrix in CSR format. If the matrix is not in CSR, then the generate step converts it into a CSR matrix. The generation fails if the matrix is not convertible to CSR.

Note

As the constructor uses the copy and convert functionality, it is not possible to create a empty solver or a solver with a matrix in any other format other than CSR, if none of the executor modules are being compiled with.

Template Parameters

ValueType	precision of matrix elements
IndexType	precision of matrix indices

8.61.2 Member Function Documentation

8.61.2.1 get_system_matrix()

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::shared_ptr<const matrix::Csr<ValueType, IndexType> > gko::solver::LowerTrs< ValueType,
IndexType >::get_system_matrix ( ) const  [inline]
```

Gets the system operator (CSR matrix) of the linear system.

Returns

the system operator (CSR matrix)

```
94      {
95          return system_matrix_;
96      }
```

The documentation for this class was generated from the following file:

- ginkgo/core/solver/lower_trs.hpp (4c758394)

8.62 gko::matrix_data< ValueType, IndexType > Struct Template Reference

This structure is used as an intermediate data type to store a sparse matrix.

```
#include <ginkgo/core/base/matrix_data.hpp>
```

Classes

- struct nonzero_type

  Type used to store nonzeros.

Public Member Functions

- matrix_data (dim< 2 > size_=dim< 2 >{}, ValueType value=zero< ValueType >())

  Initializes a matrix filled with the specified value.
- template<typename RandomDistribution , typename RandomEngine >
  matrix_data (dim< 2 > size_, RandomDistribution &&dist, RandomEngine &&engine)

  Initializes a matrix with random values from the specified distribution.
- matrix_data (std::initializer_list< std::initializer_list< ValueType >> values)

  List-initializes the structure from a matrix of values.
- matrix_data (dim< 2 > size_, std::initializer_list< detail::input_triple< ValueType, IndexType >> nonzeros_)

  Initializes the structure from a list of nonzeros.
- matrix_data (dim< 2 > size_, const matrix_data &block)

  Initializes a matrix out of a matrix block via duplication.
- template<typename Accessor >
  matrix_data (const range< Accessor > &data)

  Initializes a matrix from a range.
- void ensure_row_major_order ()

  Sorts the nonzero vector so the values follow row-major order.

Static Public Member Functions

- static matrix_data diag (dim< 2 > size_, ValueType value)

  Initializes a diagonal matrix.
- static matrix_data diag (dim< 2 > size_, std::initializer_list< ValueType > nonzeros_)

  Initializes a diagonal matrix using a list of diagonal elements.
- static matrix_data diag (dim< 2 > size_, const matrix_data &block)

  Initializes a block-diagonal matrix.
- template<typename ForwardIterator >
  static matrix_data diag (ForwardIterator begin, ForwardIterator end)

  Initializes a block-diagonal matrix from a list of diagonal blocks.
- static matrix_data diag (std::initializer_list< matrix_data > blocks)

  Initializes a block-diagonal matrix from a list of diagonal blocks.
- template<typename RandomDistribution , typename RandomEngine >
  static matrix_data cond (size_type size, remove_complex< ValueType > condition_number, RandomDistribution &&dist, RandomEngine &&engine, size_type num_reflectors)

  Initializes a random dense matrix with a specific condition number.
- template<typename RandomDistribution , typename RandomEngine >
  static matrix_data cond (size_type size, remove_complex< ValueType > condition_number, RandomDistribution &&dist, RandomEngine &&engine)

  Initializes a random dense matrix with a specific condition number.

Public Attributes

- dim< 2 > size

  Size of the matrix.
- std::vector< nonzero_type > nonzeros

  A vector of tuples storing the non-zeros of the matrix.

8.62.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
struct gko::matrix_data< ValueType, IndexType >

This structure is used as an intermediate data type to store a sparse matrix.

The matrix is stored as a sequence of nonzero elements, where each element is a triple of the form (row_index, column_index, value).

Note

All Ginkgo functions returning such a structure will return the nonzeros sorted in row-major order.

All Ginkgo functions that take this structure as input expect that the nonzeros are sorted in row-major order.

This structure is not optimized for usual access patterns and it can only exist on the CPU. Thus, it should only be used for utility functions which do not have to be optimized for performance.

Template Parameters

ValueType	type of matrix values stored in the structure
IndexType	type of matrix indexes stored in the structure

8.62.2 Constructor & Destructor Documentation

8.62.2.1 matrix_data() [1/6]

```
template<typename ValueType = default_precision, typename IndexType = int32>
gko::matrix_data< ValueType, IndexType >::matrix_data (
            dim< 2 > size_ = dim<2>{},
            ValueType value = zero<ValueType>() )  [inline]
```

Initializes a matrix filled with the specified value.

Parameters

*size_*	dimensions of the matrix
value	value used to fill the elements of the matrix

```
143                                     {}, ValueType value = zero<ValueType>())
144         : size{size_}
145     {
146         if (value == zero<ValueType>()) {
147             return;
148         }
149         for (size_type row = 0; row < size[0]; ++row) {
150             for (size_type col = 0; col < size[1]; ++col) {
151                 nonzeros.emplace_back(row, col, value);
152             }
153         }
154     }
```

8.62.2.2 matrix_data() [2/6]

```
template<typename ValueType = default_precision, typename IndexType = int32>
template<typename RandomDistribution , typename RandomEngine >
gko::matrix_data< ValueType, IndexType >::matrix_data (
            dim< 2 > size_,
            RandomDistribution && dist,
            RandomEngine && engine )  [inline]
```

Initializes a matrix with random values from the specified distribution.

Template Parameters

RandomDistribution	random distribution type
RandomEngine	random engine type

Parameters

*size_*	dimensions of the matrix
dist	random distribution of the elements of the matrix
engine	random engine used to generate random values

8.62.2.3 matrix_data() [3/6]

```
template<typename ValueType = default_precision, typename IndexType = int32>
gko::matrix_data< ValueType, IndexType >::matrix_data (
            std::initializer_list< std::initializer_list< ValueType >> values )  [inline]
```

List-initializes the structure from a matrix of values.

Parameters

values	a 2D braced-init-list of matrix values.

8.62.2.4 matrix_data() [4/6]

```
template<typename ValueType = default_precision, typename IndexType = int32>
gko::matrix_data< ValueType, IndexType >::matrix_data (
            dim< 2 > size_,
            std::initializer_list< detail::input_triple< ValueType, IndexType >> nonzeros_ )
[inline]
```

Initializes the structure from a list of nonzeros.

Parameters

*size_*	dimensions of the matrix
nonzeros↩ _	list of nonzero elements

8.62.2.5 matrix_data() [5/6]

```
template<typename ValueType = default_precision, typename IndexType = int32>
gko::matrix_data< ValueType, IndexType >::matrix_data (
            dim< 2 > size_,
            const matrix_data< ValueType, IndexType > & block )  [inline]
```

Initializes a matrix out of a matrix block via duplication.

Parameters

size	size of the block-matrix (in blocks)
diag_block	matrix block used to fill the complete matrix

References gko::matrix_data< ValueType, IndexType >::size.

8.62.2.6 matrix_data() [6/6]

```
template<typename ValueType = default_precision, typename IndexType = int32>
template<typename Accessor >
gko::matrix_data< ValueType, IndexType >::matrix_data (
            const range< Accessor > & data )  [inline]
```

Initializes a matrix from a range.

Template Parameters

Accessor	accessor type of the input range

Parameters

data	range used to initialize the matrix

References gko::range< Accessor >::length().

8.62.3 Member Function Documentation

8.62.3.1 cond() [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
template<typename RandomDistribution , typename RandomEngine >
static matrix_data gko::matrix_data< ValueType, IndexType >::cond (
            size_type size,
            remove_complex< ValueType > condition_number,
            RandomDistribution && dist,
            RandomEngine && engine )  [inline], [static]
```

Initializes a random dense matrix with a specific condition number.

The matrix is generated by applying a series of random Hausholder reflectors to a diagonal matrix with diagonal entries uniformly distributed between `sqrt(condition_number)` and `1/sqrt(condition_number)`.

This version of the function applies `size - 1` reflectors to each side of the diagonal matrix.

Template Parameters

RandomDistribution	the type of the random distribution
RandomEngine	the type of the random engine

Parameters

size	number of rows and columns of the matrix
condition_number	condition number of the matrix
dist	random distribution used to generate reflectors
engine	random engine used to generate reflectors

Returns

the dense matrix with the specified condition number

References gko::matrix_data< ValueType, IndexType >::cond(), and gko::matrix_data< ValueType, IndexType >::size.

8.62.3.2 cond() [2/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
template<typename RandomDistribution , typename RandomEngine >
static matrix_data gko::matrix_data< ValueType, IndexType >::cond (
            size_type size,
            remove_complex< ValueType > condition_number,
            RandomDistribution && dist,
            RandomEngine && engine,
            size_type num_reflectors )  [inline], [static]
```

Initializes a random dense matrix with a specific condition number.

The matrix is generated by applying a series of random Hausholder reflectors to a diagonal matrix with diagonal entries uniformly distributed between `sqrt(condition_number)` and `1/sqrt(condition_number)`.

Template Parameters

RandomDistribution	the type of the random distribution
RandomEngine	the type of the random engine

Parameters

size	number of rows and columns of the matrix
condition_number	condition number of the matrix
dist	random distribution used to generate reflectors
engine	random engine used to generate reflectors
num_reflectors	number of reflectors to apply from each side

Returns

the dense matrix with the specified condition number

References gko::matrix_data< ValueType, IndexType >::size.

Referenced by gko::matrix_data< ValueType, IndexType >::cond().

8.62.3.3 diag() [1/5]

```
template<typename ValueType = default_precision, typename IndexType = int32>
static matrix_data gko::matrix_data< ValueType, IndexType >::diag (
            dim< 2 > size_,
            const matrix_data< ValueType, IndexType > & block )  [inline], [static]
```

Initializes a block-diagonal matrix.

Parameters

*size_*	the size of the matrix
diag_block	matrix used to fill diagonal blocks

Returns

the block-diagonal matrix

References gko::matrix_data< ValueType, IndexType >::nonzeros, and gko::matrix_data< ValueType, IndexType >::size.

8.62.3.4 diag() [2/5]

```
template<typename ValueType = default_precision, typename IndexType = int32>
static matrix_data gko::matrix_data< ValueType, IndexType >::diag (
            dim< 2 > size_,
            std::initializer_list< ValueType > nonzeros_ )  [inline], [static]
```

Initializes a diagonal matrix using a list of diagonal elements.

Parameters

*size_*	dimensions of the matrix
*nonzeros_*	list of diagonal elements

Returns

the diagonal matrix

References gko::matrix_data< ValueType, IndexType >::nonzeros.

8.62.3.5 diag() [3/5]

```
template<typename ValueType = default_precision, typename IndexType = int32>
static matrix_data gko::matrix_data< ValueType, IndexType >::diag (
            dim< 2 > size_,
            ValueType value )  [inline], [static]
```

Initializes a diagonal matrix.

Parameters

*size_*	dimensions of the matrix
value	value used to fill the elements of the matrix

Returns

the diagonal matrix

References gko::matrix_data< ValueType, IndexType >::nonzeros.

Referenced by gko::matrix_data< ValueType, IndexType >::diag().

8.62.3.6 diag() [4/5]

```
template<typename ValueType = default_precision, typename IndexType = int32>
template<typename ForwardIterator >
static matrix_data gko::matrix_data< ValueType, IndexType >::diag (
            ForwardIterator begin,
            ForwardIterator end )  [inline], [static]
```

Initializes a block-diagonal matrix from a list of diagonal blocks.

Template Parameters

ForwardIterator	type of list iterator

Parameters

begin	the first iterator of the list
end	the last iterator of the list

Returns

the block-diagonal matrix with diagonal blocks set to the blocks between begin (inclusive) and end (exclusive)

References gko::matrix_data< ValueType, IndexType >::nonzeros.

8.62.3.7 diag() [5/5]

```
template<typename ValueType = default_precision, typename IndexType = int32>
static matrix_data gko::matrix_data< ValueType, IndexType >::diag (
            std::initializer_list< matrix_data< ValueType, IndexType > > blocks )  [inline],
[static]
```

Initializes a block-diagonal matrix from a list of diagonal blocks.

Parameters

blocks	a list of blocks to initialize from

Returns

the block-diagonal matrix with diagonal blocks set to the blocks passed in blocks

References gko::matrix_data< ValueType, IndexType >::diag().

8.62.4 Member Data Documentation

8.62.4.1 nonzeros

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::vector<nonzero_type> gko::matrix_data< ValueType, IndexType >::nonzeros
```

A vector of tuples storing the non-zeros of the matrix.

The first two elements of the tuple are the row index and the column index of a matrix element, and its third element is the value at that position.

Referenced by gko::matrix_data< ValueType, IndexType >::diag(), and gko::matrix_data< ValueType, IndexType >::ensure_row_major_order().

The documentation for this struct was generated from the following file:

- ginkgo/core/base/matrix_data.hpp (4bde4271)

8.63 gko::matrix::Hybrid< ValueType, IndexType >::minimal_storage_limit Class Reference

minimal_storage_limit is a stratgy_type which decides the number of stored elements per row of the ell part.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Public Member Functions

- minimal_storage_limit ()
  Creates a minimal_storage_limit strategy.
- size_type compute_ell_num_stored_elements_per_row (Array< size_type > *row_nnz) const override
  Computes the number of stored elements per row of the ell part.

8.63.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >::minimal_storage_limit

minimal_storage_limit is a stratgy_type which decides the number of stored elements per row of the ell part.

It is determined by the size of ValueType and IndexType, the storage is the minimum among all partition.

8.63.2 Member Function Documentation

8.63.2.1 compute_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::minimal_storage_limit::compute_ell_↩
num_stored_elements_per_row (
            Array< size_type > * row_nnz ) const  [inline], [override], [virtual]
```

Computes the number of stored elements per row of the ell part.

Parameters

row_nnz	the number of nonzeros of each row

Returns

the number of stored elements per row of the ell part

Implements gko::matrix::Hybrid< ValueType, IndexType >::strategy_type.

References gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit::compute_ell_num_stored_elements↩
_per_row().

The documentation for this class was generated from the following file:

- ginkgo/core/matrix/hybrid.hpp (3e51a52b)

8.64 gko::matrix_data< ValueType, IndexType >::nonzero_type Struct Reference

Type used to store nonzeros.

`#include <ginkgo/core/base/matrix_data.hpp>`

8.64.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
struct gko::matrix_data< ValueType, IndexType >::nonzero_type

Type used to store nonzeros.

The documentation for this struct was generated from the following file:

- ginkgo/core/base/matrix_data.hpp (4bde4271)

8.65 gko::NotCompiled Class Reference

NotCompiled is thrown when attempting to call an operation which is a part of a module that was not compiled on the system.

`#include <ginkgo/core/base/exception.hpp>`

Public Member Functions

- NotCompiled (const std::string &file, int line, const std::string &func, const std::string &module)

  Initializes a NotCompiled error.

8.65.1 Detailed Description

NotCompiled is thrown when attempting to call an operation which is a part of a module that was not compiled on the system.

8.65.2 Constructor & Destructor Documentation

8.65.2.1 NotCompiled()

```
gko::NotCompiled::NotCompiled (
            const std::string & file,
            int line,
            const std::string & func,
            const std::string & module )  [inline]
```

Initializes a NotCompiled error.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
func	The name of the function that has not been compiled
module	The name of the module which contains the function

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.66 gko::NotImplemented Class Reference

NotImplemented is thrown in case an operation has not yet been implemented (but will be implemented in the future).

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- NotImplemented (const std::string &file, int line, const std::string &func)

  Initializes a NotImplemented error.

8.66.1 Detailed Description

NotImplemented is thrown in case an operation has not yet been implemented (but will be implemented in the future).

8.66.2 Constructor & Destructor Documentation

8.66.2.1 NotImplemented()

```
gko::NotImplemented::NotImplemented (
            const std::string & file,
            int line,
            const std::string & func )  [inline]
```

Initializes a NotImplemented error.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
func	The name of the not-yet implemented function

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.67 gko::NotSupported Class Reference

NotSupported is thrown in case it is not possible to perform the requested operation on the given object type.

`#include <ginkgo/core/base/exception.hpp>`

Public Member Functions

- NotSupported (const std::string &file, int line, const std::string &func, const std::string &obj_type)
  Initializes a NotSupported error.

8.67.1 Detailed Description

NotSupported is thrown in case it is not possible to perform the requested operation on the given object type.

8.67.2 Constructor & Destructor Documentation

8.67.2.1 NotSupported()

```
gko::NotSupported::NotSupported (
            const std::string & file,
            int line,
            const std::string & func,
            const std::string & obj_type )  [inline]
```

Initializes a NotSupported error.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
func	The name of the function where the error occured
obj_type	The object type on which the requested operation cannot be performed.

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.68 gko::null_deleter< T > Class Template Reference

This is a deleter that does not delete the object.

`#include <ginkgo/core/base/utils.hpp>`

Public Member Functions

- void operator() (pointer) const noexcept
  Deletes the object.

8.68.1 Detailed Description

template<typename T>
class gko::null_deleter< T >

This is a deleter that does not delete the object.

It is useful where the object has been allocated elsewhere and will be deleted manually.

8.68.2 Member Function Documentation

8.68.2.1 operator()()

```
template<typename T >
void gko::null_deleter< T >::operator() (
            pointer  ) const  [inline], [noexcept]
```

Deletes the object.

Parameters

ptr	pointer to the object being deleted

The documentation for this class was generated from the following file:

- ginkgo/core/base/utils.hpp (4bde4271)

8.69 gko::OmpExecutor Class Reference

This is the Executor subclass which represents the OpenMP device (typically CPU).

`#include <ginkgo/core/base/executor.hpp>`

Public Member Functions

- std::shared_ptr< Executor > get_master () noexcept override
  Returns the master OmpExecutor of this Executor.
- std::shared_ptr< const Executor > get_master () const noexcept override
  Returns the master OmpExecutor of this Executor.
- void synchronize () const override
  Synchronize the operations launched on the executor with its master.

Static Public Member Functions

- static std::shared_ptr< OmpExecutor > create ()
  Creates a new OmpExecutor.

8.69.1 Detailed Description

This is the Executor subclass which represents the OpenMP device (typically CPU).

8.69.2 Member Function Documentation

8.69.2.1 get_master() [1/2]

`std::shared_ptr<const Executor> gko::OmpExecutor::get_master ( ) const [override], [virtual], [noexcept]`

Returns the master OmpExecutor of this Executor.

Returns

the master OmpExecutor of this Executor.

Implements gko::Executor.

8.69.2.2 get_master() [2/2]

```
std::shared_ptr<Executor> gko::OmpExecutor::get_master ( )  [override], [virtual], [noexcept]
```

Returns the master OmpExecutor of this Executor.

Returns

the master OmpExecutor of this Executor.

Implements gko::Executor.

The documentation for this class was generated from the following file:

- ginkgo/core/base/executor.hpp (f1a4eb68)

8.70 gko::Operation Class Reference

Operations can be used to define functionalities whose implementations differ among devices.

```
#include <ginkgo/core/base/executor.hpp>
```

Public Member Functions

- virtual const char * get_name () const noexcept

  Returns the operation's name.

8.70.1 Detailed Description

Operations can be used to define functionalities whose implementations differ among devices.

This is done by extending the Operation class and implementing the overloads of the Operation::run() method for all Executor types. When invoking the Executor::run() method with the Operation as input, the library will select the Operation::run() overload corresponding to the dynamic type of the Executor instance.

Consider an overload of `operator<<` for Executors, which prints some basic device information (e.g. device type and id) of the Executor to a C++ stream:

```
std::ostream& operator«(std::ostream &os, const gko::Executor &exec);
```

One possible implementation would be to use RTTI to find the dynamic type of the Executor, However, using the Operation feature of Ginkgo, there is a more elegant approach which utilizes polymorphism. The first step is to define an Operation that will print the desired information for each Executor type.

```
class DeviceInfoPrinter : public gko::Operation {
public:
    explicit DeviceInfoPrinter(std::ostream &os) : os_(os) {}
    void run(const gko::OmpExecutor *) const override { os_ « "OMP"; }
    void run(const gko::CudaExecutor *exec) const override
    { os_ « "CUDA(" « exec->get_device_id() « ")"; }
    // This is optional, if not overloaded, defaults to OmpExecutor overload
    void run(const gko::ReferenceExecutor *) const override
    { os_ « "Reference CPU"; }
private:
    std::ostream &os_;
};
```

Using DeviceInfoPrinter, the implementation of operator<< is as simple as calling the run() method of the executor.

```
std::ostream& operator«(std::ostream &os, const gko::Executor &exec)
{
    DeviceInfoPrinter printer(os);
    exec.run(printer);
    return os;
}
```

Now it is possible to write the following code:

```
auto omp = gko::OmpExecutor::create();
std::cout « *omp « std::endl
          « *gko::CudaExecutor::create(0, omp) « std::endl
          « *gko::ReferenceExecutor::create() « std::endl;
```

which produces the expected output:

```
OMP
CUDA(0)
Reference CPU
```

One might feel that this code is too complicated for such a simple task. Luckily, there is an overload of the Executor::run() method, which is designed to facilitate writing simple operations like this one. The method takes two closures as input: one which is run for OMP, and the other one for CUDA executors. Using this method, there is no need to implement an Operation subclass:

```
std::ostream& operator«(std::ostream &os, const gko::Executor &exec)
{
    exec.run(
        [&]() { os « "OMP"; },  // OMP closure
        [&]() { os « "CUDA("    // CUDA closure
                   « static_cast<gko::CudaExecutor&>(exec)
                       .get_device_id()
                   « ")"; });
    return os;
}
```

Using this approach, however, it is impossible to distinguish between a OmpExecutor and ReferenceExecutor, as both of them call the OMP closure.

8.70.2 Member Function Documentation

8.70.2.1 get_name()

```
virtual const char* gko::Operation::get_name ( ) const  [virtual], [noexcept]
```

Returns the operation's name.

Returns

the operation's name

The documentation for this class was generated from the following file:

- ginkgo/core/base/executor.hpp (f1a4eb68)

8.71 gko::log::operation_data Struct Reference

Struct representing Operator related data.

```
#include <ginkgo/core/log/record.hpp>
```

8.71.1 Detailed Description

Struct representing Operator related data.

The documentation for this struct was generated from the following file:

- ginkgo/core/log/record.hpp (f0a50f96)

8.72 gko::OutOfBoundsError Class Reference

OutOfBoundsError is thrown if a memory access is detected to be out-of-bounds.

`#include <ginkgo/core/base/exception.hpp>`

Public Member Functions

- OutOfBoundsError (const std::string &file, int line, size_type index, size_type bound)
  Initializes an OutOfBoundsError.

8.72.1 Detailed Description

OutOfBoundsError is thrown if a memory access is detected to be out-of-bounds.

8.72.2 Constructor & Destructor Documentation

8.72.2.1 OutOfBoundsError()

```
gko::OutOfBoundsError::OutOfBoundsError (
            const std::string & file,
            int line,
            size_type index,
            size_type bound )  [inline]
```

Initializes an OutOfBoundsError.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
index	The position that was accessed
bound	The first out-of-bound index

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.73 gko::factorization::ParIlu< ValueType, IndexType > Class Template Reference

ParILU is an incomplete LU factorization which is computed in parallel.

`#include <ginkgo/core/factorization/par_ilu.hpp>`

Additional Inherited Members

8.73.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::factorization::ParIlu< ValueType, IndexType >

ParILU is an incomplete LU factorization which is computed in parallel.

L is a lower unitriangular, while U is an upper triangular matrix, which approximate a given matrix A with $A \approx LU$. Here, L and U have the same sparsity pattern as A, which is also called ILU(0).

The ParILU algorithm generates the incomplete factors iteratively, using a fixed-point iteration of the form

$$F(L,U) = \begin{cases} \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right), & i > j \\ a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, & i \leq j \end{cases}$$

In general, the entries of L and U can be iterated in parallel and in asynchronous fashion, the algorithm asymptotically converges to the incomplete factors L and U fulfilling $(R = A - L \cdot U)\,|_{\mathcal{S}} = 0|_{\mathcal{S}}$ where $\mathcal{S}$ is the pre-defined sparsity pattern (in case of ILU(0) the sparsity pattern of the system matrix A). The number of ParILU sweeps needed for convergence depends on the parallelism level: For sequential execution, a single sweep is sufficient, for fine-grained parallelism, 3 sweeps are typically generating a good approximation.

The ParILU algorithm in Ginkgo follows the design of E. Chow and A. Patel, Fine-grained Parallel Incomplete LU Factorization, SIAM Journal on Scientific Computing, 37, C169-C193 (2015).

Template Parameters

ValueType	Type of the values of all matrices used in this class
IndexType	Type of the indices of all matrices used in this class

The documentation for this class was generated from the following file:

- ginkgo/core/factorization/par_ilu.hpp (50506722)

8.74 gko::Perturbation< ValueType > Class Template Reference

The Perturbation class can be used to construct a LinOp to represent the operation `(identity + scalar * basis * projector)`.

`#include <ginkgo/core/base/perturbation.hpp>`

Public Member Functions

- const std::shared_ptr< const LinOp > get_basis () const noexcept

  Returns the basis of the perturbation.
- const std::shared_ptr< const LinOp > get_projector () const noexcept

  Returns the projector of the perturbation.
- const std::shared_ptr< const LinOp > get_scalar () const noexcept

  Returns the scalar of the perturbation.

8.74.1 Detailed Description

template<typename ValueType = default_precision>
class gko::Perturbation< ValueType >

The Perturbation class can be used to construct a LinOp to represent the operation `(identity + scalar * basis * projector)`.

This operator adds a movement along a direction constructed by `basis` and `projector` on the LinOp. `projector` gives the coefficient of `basis` to decide the direction.

For example, the Householder matrix can be represented with the Perturbation operator as follows. If u is the Householder factor then we can generate the `Householder transformation`, H = (I - 2 u u∗). In this case, the parameters of Perturbation class are scalar = -2, basis = u, and projector = u∗.

Template Parameters

ValueType	precision of input and result vectors

Note

the apply operations of Perturbation class are not thread safe

8.74.2 Member Function Documentation

8.74.2.1 get_basis()

```
template<typename ValueType = default_precision>
const std::shared_ptr<const LinOp> gko::Perturbation< ValueType >::get_basis ( ) const  [inline],
[noexcept]
```

Returns the basis of the perturbation.

Returns

the basis of the perturbation

```
81     {
82         return basis_;
83     }
```

8.74.2.2 get_projector()

```
template<typename ValueType = default_precision>
const std::shared_ptr<const LinOp> gko::Perturbation< ValueType >::get_projector ( ) const
[inline], [noexcept]
```

Returns the projector of the perturbation.

Returns

the projector of the perturbation

8.74.2.3 get_scalar()

```
template<typename ValueType = default_precision>
const std::shared_ptr<const LinOp> gko::Perturbation< ValueType >::get_scalar ( ) const  [inline],
[noexcept]
```

Returns the scalar of the perturbation.

Returns

the scalar of the perturbation

The documentation for this class was generated from the following file:

- ginkgo/core/base/perturbation.hpp (c923f483)

8.75 gko::log::polymorphic_object_data Struct Reference

Struct representing PolymorphicObject related data.

```
#include <ginkgo/core/log/record.hpp>
```

8.75.1 Detailed Description

Struct representing PolymorphicObject related data.

The documentation for this struct was generated from the following file:

- ginkgo/core/log/record.hpp (f0a50f96)

8.76 gko::PolymorphicObject Class Reference

A PolymorphicObject is the abstract base for all "heavy" objects in Ginkgo that behave polymorphically.

`#include <ginkgo/core/base/polymorphic_object.hpp>`

Public Member Functions

- std::unique_ptr< PolymorphicObject > create_default (std::shared_ptr< const Executor > exec) const

  Creates a new "default" object of the same dynamic type as this object.
- std::unique_ptr< PolymorphicObject > create_default () const

  Creates a new "default" object of the same dynamic type as this object.
- std::unique_ptr< PolymorphicObject > clone (std::shared_ptr< const Executor > exec) const

  Creates a clone of the object.
- std::unique_ptr< PolymorphicObject > clone () const

  Creates a clone of the object.
- PolymorphicObject * copy_from (const PolymorphicObject *other)

  Copies another object into this object.
- PolymorphicObject * copy_from (std::unique_ptr< PolymorphicObject > other)

  Moves another object into this object.
- PolymorphicObject * clear ()

  Transforms the object into its default state.
- std::shared_ptr< const Executor > get_executor () const noexcept

  Returns the Executor of the object.

8.76.1 Detailed Description

A PolymorphicObject is the abstract base for all "heavy" objects in Ginkgo that behave polymorphically.

It defines the basic utilities (copying moving, cloning, clearing the objects) for all such objects. It takes into account that these objects are dynamically allocated, managed by smart pointers, and used polymorphically. Additionally, it assumes their data can be allocated on different executors, and that they can be copied between those executors.

Note

Most of the public methods of this class should not be overridden directly, and are thus not virtual. Instead, there are equivalent protected methods (ending in <method_name>_impl) that should be overriden instead. This allows polymorphic objects to implement default behavior around virtual methods (parameter checking, type casting).

See also

EnablePolymorphicObject if you wish to implement a concrete polymorphic object and have sensible defaults generated automatically. EnableAbstractPolymorphicObject if you wish to implement a new abstract polymorphic object, and have the return types of the methods updated to your type (instead of having them return PolymorphicObject).

8.76.2 Member Function Documentation

8.76.2.1 clear()

```
PolymorphicObject* gko::PolymorphicObject::clear ( )  [inline]
```

Transforms the object into its default state.

Equivalent to `this->copy_from(this->create_default())`.

See also

clear_impl() when implementing this method

Returns

this

8.76.2.2 clone() [1/2]

```
std::unique_ptr<PolymorphicObject> gko::PolymorphicObject::clone ( ) const  [inline]
```

Creates a clone of the object.

This is a shorthand for clone(std::shared_ptr<const Executor>) that uses the executor of this object to construct the new object.

Returns

A clone of the LinOp.

8.76.2.3 clone() [2/2]

```
std::unique_ptr<PolymorphicObject> gko::PolymorphicObject::clone (
            std::shared_ptr< const Executor > exec ) const  [inline]
```

Creates a clone of the object.

This is the polymorphic equivalent of the *executor copy constructor* `decltype(*this)(exec, this)`.

Parameters

exec	the executor where the clone will be created

Returns

A clone of the LinOp.

References create_default().

8.76.2.4 copy_from() [1/2]

```
PolymorphicObject* gko::PolymorphicObject::copy_from (
            const PolymorphicObject * other )  [inline]
```

Copies another object into this object.

This is the polymorphic equivalent of the copy assignment operator.

See also

copy_from_impl(const PolymorphicObject *)

Parameters

other	the object to copy

Returns

this

8.76.2.5 copy_from() [2/2]

```
PolymorphicObject* gko::PolymorphicObject::copy_from (
            std::unique_ptr< PolymorphicObject > other )  [inline]
```

Moves another object into this object.

This is the polymorphic equivalent of the move assignment operator.

See also

copy_from_impl(std::unique_ptr<PolymorphicObject>)

Parameters

other	the object to move from

Returns

this

8.76.2.6 create_default() [1/2]

```
std::unique_ptr<PolymorphicObject> gko::PolymorphicObject::create_default ( ) const  [inline]
```

Creates a new "default" object of the same dynamic type as this object.

This is a shorthand for create_default(std::shared_ptr<const Executor>) that uses the executor of this object to construct the new object.

Returns

a polymorphic object of the same type as this

Referenced by clone().

8.76.2.7 create_default() [2/2]

```
std::unique_ptr<PolymorphicObject> gko::PolymorphicObject::create_default (
            std::shared_ptr< const Executor > exec ) const  [inline]
```

Creates a new "default" object of the same dynamic type as this object.

This is the polymorphic equivalent of the *executor default constructor* `decltype(*this)(exec);`.

Parameters

exec	the executor where the object will be created

Returns

a polymorphic object of the same type as this

8.76.2.8 get_executor()

```
std::shared_ptr<const Executor> gko::PolymorphicObject::get_executor ( ) const  [inline],
[noexcept]
```

Returns the Executor of the object.

Returns

Executor of the object

Referenced by gko::matrix::Dense< ValueType >::add_scaled(), gko::matrix::Coo< ValueType, IndexType >::apply2(), gko::matrix::Dense< ValueType >::compute_dot(), gko::matrix::Dense< ValueType >::compute_norm2(), gko::matrix::Dense< ValueType >::create_submatrix(), and gko::matrix::Dense< ValueType >::scale().

The documentation for this class was generated from the following file:

- ginkgo/core/base/polymorphic_object.hpp (3f08cf0a)

8.77 gko::precision_reduction Class Reference

This class is used to encode storage precisions of low precision algorithms.

`#include <ginkgo/core/base/types.hpp>`

Public Types

- using storage_type = uint8

  The underlying datatype used to store the encoding.

Public Member Functions

- constexpr precision_reduction () noexcept

  Creates a default precision_reduction encoding.
- constexpr precision_reduction (storage_type preserving, storage_type nonpreserving) noexcept

  Creates a precision_reduction encoding with the specified number of conversions.
- constexpr operator storage_type () const noexcept

  Extracts the raw data of the encoding.
- constexpr storage_type get_preserving () const noexcept

  Returns the number of preserving conversions in the encoding.
- constexpr storage_type get_nonpreserving () const noexcept

  Returns the number of non-preserving conversions in the encoding.

Static Public Member Functions

- constexpr static precision_reduction autodetect () noexcept

  Returns a special encoding which instructs the algorithm to automatically detect the best precision.
- constexpr static precision_reduction common (precision_reduction x, precision_reduction y) noexcept

  Returns the common encoding of input encodings.

8.77.1 Detailed Description

This class is used to encode storage precisions of low precision algorithms.

Some algorithms in Ginkgo can improve their performance by storing parts of the data in lower precision, while doing computation in full precision. This class is used to encode the precisions used to store the data. From the user's perspective, some algorithms can provide a parameter for fine-tuning the storage precision. Commonly, the special value returned by precision_reduction::autodetect() should be used to allow the algorithm to automatically choose an appropriate value, though manually selected values can be used for fine-tuning.

In general, a lower precision floating point value can be obtained by either dropping some of the insignificant bits of the significand (keeping the same number of exponent bits, and thus preserving the range of representable values) or using one of the hardware or software supported conversions between IEEE formats, such as double to float or float to half (reducing both the number of exponent, as well as significand bits, and thus decreasing the range of representable values).

The precision_reduction class encodes the lower precision format relative to the base precision used and the algorithm in question. The encoding is done by specifying the amount of range non-preserving conversions and

the amount of range preserving conversions that should be done on the base precision to obtain the lower precision format. For example, starting with a double precision value (11 exp, 52 sig. bits), the encoding specifying 1 non-preserving conversion and 1 preserving conversion would first use a hardware-supported non-preserving conversion to obtain a single precision value (8 exp, 23 sig. bits), followed by a preserving bit truncation to obtain a value with 8 exponent and 7 significand bits. Note that non-preserving conversion are always done first, as preserving conversions usually result in datatypes that are not supported by builtin conversions (thus, it is generally not possible to apply a non-preserving conversion to the result of a preserving conversion).

If the specified conversion is not supported by the algorithm, it will most likely fall back to using full precision for storing the data. Refer to the documentation of specific algorithms using this class for details about such special cases.

8.77.2 Constructor & Destructor Documentation

8.77.2.1 precision_reduction() [1/2]

```
constexpr gko::precision_reduction::precision_reduction ( )  [inline], [constexpr], [noexcept]
```

Creates a default precision_reduction encoding.

This encoding represents the case where no conversions are performed.

Referenced by common().

8.77.2.2 precision_reduction() [2/2]

```
constexpr gko::precision_reduction::precision_reduction (
            storage_type preserving,
            storage_type nonpreserving )  [inline], [constexpr], [noexcept]
```

Creates a precision_reduction encoding with the specified number of conversions.

Parameters

preserving	the number of range preserving conversion
nonpreserving	the number of range non-preserving conversions

8.77.3 Member Function Documentation

8.77.3.1 autodetect()

```
constexpr static precision_reduction gko::precision_reduction::autodetect ( )  [inline], [static],
[constexpr], [noexcept]
```

Returns a special encoding which instructs the algorithm to automatically detect the best precision.

Returns

a special encoding instructing the algorithm to automatically detect the best precision.

8.77.3.2 common()

```
constexpr static precision_reduction gko::precision_reduction::common (
            precision_reduction x,
            precision_reduction y )  [inline], [static], [constexpr], [noexcept]
```

Returns the common encoding of input encodings.

The common encoding is defined as the encoding that does not have more preserving, nor non-preserving conversions than the input encodings.

Parameters

x	an encoding
y	an encoding

Returns

the common encoding of `x` and `y`

References precision_reduction().

8.77.3.3 get_nonpreserving()

```
constexpr storage_type gko::precision_reduction::get_nonpreserving ( ) const  [inline], [constexpr],
[noexcept]
```

Returns the number of non-preserving conversions in the encoding.

Returns

the number of non-preserving conversions in the encoding.

8.77.3.4 get_preserving()

```
constexpr storage_type gko::precision_reduction::get_preserving ( ) const  [inline], [constexpr],
[noexcept]
```

Returns the number of preserving conversions in the encoding.

Returns

the number of preserving conversions in the encoding.

8.77.3.5 operator storage_type()

```
constexpr gko::precision_reduction::operator storage_type ( ) const  [inline], [constexpr],
[noexcept]
```

Extracts the raw data of the encoding.

Returns

the raw data of the encoding

The documentation for this class was generated from the following file:

- ginkgo/core/base/types.hpp (4bde4271)

8.78 gko::Preconditionable Class Reference

A LinOp implementing this interface can be preconditioned.

```
#include <ginkgo/core/base/lin_op.hpp>
```

Public Member Functions

- virtual std::shared_ptr< const LinOp > get_preconditioner () const

  Returns the preconditioner operator used by the Preconditionable.
- virtual void set_preconditioner (std::shared_ptr< const LinOp > new_precond)

  Sets the preconditioner operator used by the Preconditionable.

8.78.1 Detailed Description

A LinOp implementing this interface can be preconditioned.

8.78.2 Member Function Documentation

8.78.2.1 get_preconditioner()

```
virtual std::shared_ptr<const LinOp> gko::Preconditionable::get_preconditioner ( ) const
[inline], [virtual]
```

Returns the preconditioner operator used by the Preconditionable.

Returns

the preconditioner operator used by the Preconditionable

```
475     {
476         return preconditioner_;
477     }
```

8.78.2.2 set_preconditioner()

```
virtual void gko::Preconditionable::set_preconditioner (
             std::shared_ptr< const LinOp > new_precond )  [inline], [virtual]
```

Sets the preconditioner operator used by the Preconditionable.

Parameters

new_precond	the new preconditioner operator used by the Preconditionable

The documentation for this class was generated from the following file:

- ginkgo/core/base/lin_op.hpp (4c758394)

8.79 gko::range< Accessor > Class Template Reference

A range is a multidimensional view of the memory.

```
#include <ginkgo/core/base/range.hpp>
```

Public Types

- using accessor = Accessor

  The type of the underlying accessor.

Public Member Functions

- template<typename... AccessorParams>
  constexpr range (AccessorParams &&... params)

  Creates a new range.
- template<typename... DimensionTypes>
  constexpr auto operator() (DimensionTypes &&... dimensions) const -> decltype(std::declval< accessor >()(std::forward< DimensionTypes >(dimensions)...))

  Returns a value (or a sub-range) with the specified indexes.
- template<typename OtherAccessor >
  const range & operator= (const range< OtherAccessor > &other) const
- const range & operator= (const range &other) const

  Assigns another range to this range.
- constexpr size_type length (size_type dimension) const

  Returns the length of the specified dimension of the range.
- constexpr const accessor * operator-> () const noexcept

  Returns a pointer to the accessor.
- constexpr const accessor & get_accessor () const noexcept

  `Returns a reference to the accessor.

Static Public Attributes

- static constexpr size_type dimensionality = accessor::dimensionality

  The number of dimensions of the range.

8.79.1 Detailed Description

template<typename Accessor>
class gko::range< Accessor >

A range is a multidimensional view of the memory.

The range does not store any of its values by itself. Instead, it obtains the values through an accessor (e.g. accessor::row_major) which describes how the indexes of the range map to physical locations in memory.

There are several advantages of using ranges instead of plain memory pointers:

1. Code using ranges is easier to read and write, as there is no need for index linearizations.
2. Code using ranges is safer, as it is impossible to accidentally miscalculate an index or step out of bounds, since range accessors perform bounds checking in debug builds. For performance, this can be disabled in release builds by defining the `NDEBUG` flag.
3. Ranges enable generalized code, as algorithms can be written independent of the memory layout. This does not impede various optimizations based on memory layout, as it is always possible to specialize algorithms for ranges with specific memory layouts.
4. Ranges have various pointwise operations predefined, which reduces the amount of loops that need to be written.

8.79.1.1 Range operations

Ranges define a complete set of pointwise unary and binary operators which extend the basic arithmetic operators in C++, as well as a few pointwise operations and mathematical functions useful in ginkgo, and a couple of non-pointwise operations. Compound assignment (+=, *=, etc.) is not yet supported at this moment. Here is a complete list of operations:

- standard unary operations: +, -, !, ~
- standard binary operations: +, * (this is pointwise, not matrix multiplication), /, %, <, >, <=, >=, ==, !=, ||, &&, |, &, ^, <<, >>
- useful unary functions: `zero`, `one`, `abs`, `real`, `imag`, `conj`, `squared_norm`
- useful binary functions: `min`, `max`

All binary pointwise operations also work as expected if one of the operands is a scalar and the other is a range. The scalar operand will have the effect as if it was a range of the same size as the other operand, filled with the specified scalar.

Two "global" functions `transpose` and `mmul` are also supported. `transpose` transposes the first two dimensions of the range (i.e. `transpose(r)(i, j, ...) == r(j, i, ...)`). `mmul` performs a (batched) matrix multiply of the ranges - the first two dimensions represent the matrices, while the rest represent the batch. For example, given the ranges `r1` and `r2` of dimensions `(3, 2, 3)` and `(2, 4, 3)`, respectively, `mmul(r1, r2)` will return a range of dimensions `(3, 4, 3)`, obtained by multiplying the 3 frontal slices of the range, and stacking the result back vertically.

8.79.1.2 Compound operations

Multiple range operations can be combined into a single expression. For example, an "axpy" operation can be obtained using `y = alpha * x + y`, where `x` an `y` are ranges, and `alpha` is a scalar. Range operations are optimized for memory access, and the above code does not allocate additional storage for intermediate ranges `alpha * x` or `aplha * x + y`. In fact, the entire computation is done during the assignment, and the results of operations + and * only register the data, and the types of operations that will be computed once the results are needed.

It is possible to store and reuse these intermediate expressions. The following example will overwrite the range `x` with it's 4th power:

```
 {c++}
auto square = x * x;  // this is range constructor, not range assignment!
x = square;  // overwrites x with x*x (this is range assignment)
x = square;  // overwrites new x (x*x) with (x*x)*(x*x) (as is this)
```

8.79.1.3 Caveats

__`mmul` is not a highly-optimized BLAS-3 version of the matrix multiplication.__ The current design of ranges and accessors prevents that, so if you need a high-perfromance matrix multiplication, you should use one of the libraries that provide that, or implement your own (you can use pointwise range operations to help simplify that). However, range design might get improved in the future to allow efficient implementations of BLAS-3 kernels.

Aliasing the result range in `mmul` and `transpose` is not allowed. Constructs like `A = transpose(A)`, `A = mmul(A, A)`, or `A = mmul(A, A) + C` lead to undefined behavior. However, aliasing input arguments is allowed: `C = mmul(A, A)`, and even `C = mmul(A, A) + C` is valid code (in the last example, only pointwise operations are aliased). `C = mmul(A, A + C)` is not valid though.

8.79.1.4 Examples

The range unit tests in core/test/base/range.cpp contain lots of simple 1-line examples of range operations. The accessor unit tests in core/test/base/range.cpp show how to use ranges with concrete accessors, and how to use range slices using `span`s as arguments to range function call operator. Finally, examples/range contains a complete example where ranges are used to implement a simple version of the right-looking LU factorization.

Template Parameters

Accessor	underlying accessor of the range

8.79.2 Constructor & Destructor Documentation

8.79.2.1 range()

```
template<typename Accessor>
template<typename...  AccessorParams>
constexpr gko::range< Accessor >::range (
            AccessorParams &&...  params )  [inline], [explicit], [constexpr]
```

Creates a new range.

Template Parameters

AccessorParam	types of parameters forwarded to the accessor constructor

Parameters

params	parameters forwarded to Accessor constructor.

```
318         : accessor_{std::forward<AccessorParams>(params)...}
319     {}
```

8.79.3 Member Function Documentation

8.79.3.1 get_accessor()

```
template<typename Accessor>
constexpr const accessor& gko::range< Accessor >::get_accessor ( ) const  [inline], [constexpr],
[noexcept]
```

`Returns a reference to the accessor.

Returns

reference to the accessor

Referenced by gko::range< Accessor >::operator=().

8.79.3.2 length()

```
template<typename Accessor>
constexpr size_type gko::range< Accessor >::length (
            size_type dimension ) const  [inline], [constexpr]
```

Returns the length of the specified dimension of the range.

Parameters

dimension	the dimensions whose length is returned

Returns

the length of the `dimension`-th dimension of the range

Referenced by gko::matrix_data< ValueType, IndexType >::matrix_data().

8.79.3.3 operator()()

```
template<typename Accessor>
template<typename...  DimensionTypes>
constexpr auto gko::range< Accessor >::operator() (
            DimensionTypes &&...  dimensions ) const -> decltype(std::declval<accessor>()(
std::forward<DimensionTypes>(dimensions)...))    [inline], [constexpr]
```

Returns a value (or a sub-range) with the specified indexes.

Template Parameters

DimensionTypes	The types of indexes. Supported types depend on the underlying accessor, but are usually either integer types or spans. If at least one index is a span, the returned value will be a sub-range.

Parameters

dimensions	the indexes of the values.

Returns

a value on position `(dimensions...)`.

References gko::range< Accessor >::dimensionality.

8.79.3.4 operator->()

```
template<typename Accessor>
constexpr const accessor* gko::range< Accessor >::operator-> ( ) const  [inline], [constexpr],
[noexcept]
```

Returns a pointer to the accessor.

Can be used to access data and functions of a specific accessor.

Returns

pointer to the accessor

8.79.3.5 operator=() [1/2]

```
template<typename Accessor>
const range& gko::range< Accessor >::operator= (
            const range< Accessor > & other ) const  [inline]
```

Assigns another range to this range.

The order of assignment is defined by the accessor of this range, thus the memory access will be optimized for the resulting range, and not for the other range. If the sizes of two ranges do not match, the result is undefined. Sizes of the ranges are checked at runtime in debug builds.

Note

Temporary accessors are allowed to define the implementation of the assignment as deleted, so do not expect `r1 * r2 = r2` to work.

Parameters

other	the range to copy the data from

References gko::range< Accessor >::get_accessor().

8.79.3.6 operator=() [2/2]

```
template<typename Accessor>
template<typename OtherAccessor >
const range& gko::range< Accessor >::operator= (
            const range< OtherAccessor > & other ) const  [inline]
```

This is a version of the function which allows to copy between ranges of different accessors.

Template Parameters

OtherAccessor	accessor of the other range

The documentation for this class was generated from the following file:

- ginkgo/core/base/range.hpp (f1a4eb68)

8.80 gko::ReadableFromMatrixData< ValueType, IndexType > Class Template Reference

A LinOp implementing this interface can read its data from a matrix_data structure.

`#include <ginkgo/core/base/lin_op.hpp>`

Public Member Functions

- virtual void read (const matrix_data< ValueType, IndexType > &data)=0
  Reads a matrix from a matrix_data structure.

8.80.1 Detailed Description

template<typename ValueType, typename IndexType>
class gko::ReadableFromMatrixData< ValueType, IndexType >

A LinOp implementing this interface can read its data from a matrix_data structure.

8.80.2 Member Function Documentation

8.80.2.1 read()

```
template<typename ValueType, typename IndexType>
virtual void gko::ReadableFromMatrixData< ValueType, IndexType >::read (
            const matrix_data< ValueType, IndexType > & data )  [pure virtual]
```

Reads a matrix from a matrix_data structure.

Parameters

data	the matrix_data structure

Implemented in gko::matrix::Hybrid< ValueType, IndexType >, gko::matrix::Csr< ValueType, IndexType >, gko::matrix::Coo< ValueType, IndexType >, gko::matrix::Ell< ValueType, IndexType >, gko::matrix::Sellp< ValueType, IndexType >, and gko::matrix::SparsityCsr< ValueType, IndexType >.

The documentation for this class was generated from the following file:

- ginkgo/core/base/lin_op.hpp (4c758394)

8.81 gko::log::Record Class Reference

Record is a Logger which logs every event to an object.

`#include <ginkgo/core/log/record.hpp>`

Classes

- struct logged_data

  Struct storing the actually logged data.

Public Member Functions

- const logged_data & get () const noexcept

  Returns the logged data.
- logged_data & get () noexcept

Static Public Member Functions

- static std::unique_ptr< Record > create (std::shared_ptr< const Executor > exec, const mask_type &enabled_events=Logger::all_events_mask, size_type max_storage=1)

  Creates a Record logger.

8.81.1 Detailed Description

Record is a Logger which logs every event to an object.

The object can then be accessed at any time by asking the logger to return it.

Note

Please note that this logger can have significant memory and performance overhead. In particular, when logging events such as the `check` events, all parameters are cloned. If it is sufficient to clone one parameter, consider implementing a specific logger for this. In addition, it is advised to tune the history size in order to control memory overhead.

8.81.2 Member Function Documentation

8.81.2.1 create()

```
static std::unique_ptr<Record> gko::log::Record::create (
            std::shared_ptr< const Executor > exec,
            const mask_type & enabled_events = Logger::all_events_mask,
            size_type max_storage = 1 )  [inline], [static]
```

Creates a Record logger.

This dynamically allocates the memory, constructs the object and returns an std::unique_ptr to this object.

Parameters

exec	the executor
enabled_events	the events enabled for this logger. By default all events.
max_storage	the size of storage (i.e. history) wanted by the user. By default 0 is used, which means unlimited storage. It is advised to control this to reduce memory overhead of this logger.

Returns

an std::unique_ptr to the the constructed object

```
397     {
398         return std::unique_ptr<Record>(
399             new Record(exec, enabled_events, max_storage));
400     }
```

8.81.2.2 get() [1/2]

```
const logged_data& gko::log::Record::get ( ) const  [inline], [noexcept]
```

Returns the logged data.

Returns

the logged data

8.81.2.3 get() [2/2]

```
logged_data& gko::log::Record::get ( )  [inline], [noexcept]
```

The documentation for this class was generated from the following file:

- ginkgo/core/log/record.hpp (f0a50f96)

8.82 gko::ReferenceExecutor Class Reference

This is a specialization of the OmpExecutor, which runs the reference implementations of the kernels used for debugging purposes.

```
#include <ginkgo/core/base/executor.hpp>
```

Public Member Functions

- void run (const Operation &op) const override

  Runs the specified Operation using this Executor.

Additional Inherited Members

8.82.1 Detailed Description

This is a specialization of the OmpExecutor, which runs the reference implementations of the kernels used for debugging purposes.

8.82.2 Member Function Documentation

8.82.2.1 run()

```
void gko::ReferenceExecutor::run (
            const Operation & op ) const  [inline], [override], [virtual]
```

Runs the specified Operation using this Executor.

Parameters

op	the operation to run

Implements gko::Executor.

The documentation for this class was generated from the following file:

- ginkgo/core/base/executor.hpp (f1a4eb68)

8.83 gko::stop::ResidualNormReduction< ValueType > Class Template Reference

The ResidualNormReduction class is a stopping criterion which stops the iteration process when the relative residual norm is below a certain threshold.

```
#include <ginkgo/core/stop/residual_norm_reduction.hpp>
```

8.83.1 Detailed Description

template<typename ValueType = default_precision>
class gko::stop::ResidualNormReduction< ValueType >

The ResidualNormReduction class is a stopping criterion which stops the iteration process when the relative residual norm is below a certain threshold.

For better performance, the checks are run thanks to kernels on the executor where the algorithm is executed.

Note

To use this stopping criterion there are some dependencies. The constructor depends on `initial_residual` in order to compute the first relative residual norm. The check method depends on either the `residual_norm` or the `residual` being set. When any of those is not correctly provided, an exception ::gko::NotSupported() is thrown.

The documentation for this class was generated from the following file:

- ginkgo/core/stop/residual_norm_reduction.hpp (8045ac75)

8.84 gko::accessor::row_major< ValueType, Dimensionality > Class Template Reference

A row_major accessor is a bridge between a range and the row-major memory layout.

`#include <ginkgo/core/base/range_accessors.hpp>`

Public Types

- using value_type = ValueType

  Type of values returned by the accessor.
- using data_type = value_type *

  Type of underlying data storage.

Public Member Functions

- constexpr value_type & operator() (size_type row, size_type col) const

  Returns the data element at position (row, col)
- constexpr range< row_major > operator() (const span &rows, const span &cols) const

  Returns the sub-range spanning the range (rows, cols)
- constexpr size_type length (size_type dimension) const

  Returns the length in dimension `dimension`.
- template<typename OtherAccessor >
  void copy_from (const OtherAccessor &other) const

  Copies data from another accessor.

Public Attributes

- const data_type data

  Reference to the underlying data.
- const std::array< const size_type, dimensionality > lengths

  An array of dimension sizes.
- const size_type stride

  Distance between consecutive rows.

Static Public Attributes

- static constexpr size_type dimensionality = 2

  Number of dimensions of the accessor.

8.84.1 Detailed Description

template<typename ValueType, size_type Dimensionality>
class gko::accessor::row_major< ValueType, Dimensionality >

A row_major accessor is a bridge between a range and the row-major memory layout.

You should never try to explicitly create an instance of this accessor. Instead, supply it as a template parameter to a range, and pass the constructor parameters for this class to the range (it will forward it to this class).

Warning

The current implementation is incomplete, and only allows for 2-dimensional ranges.

Template Parameters

ValueType	type of values this accessor returns
Dimensionality	number of dimensions of this accessor (has to be 2)

8.84.2 Member Function Documentation

8.84.2.1 copy_from()

```
template<typename ValueType , size_type Dimensionality>
template<typename OtherAccessor >
void gko::accessor::row_major< ValueType, Dimensionality >::copy_from (
            const OtherAccessor & other ) const  [inline]
```

Copies data from another accessor.

Template Parameters

OtherAccessor	type of the other accessor

Parameters

other	other accessor

```
164     {
165         for (size_type i = 0; i < lengths[0]; ++i) {
166             for (size_type j = 0; j < lengths[1]; ++j) {
```

```
167                 (*this)(i, j) = other(i, j);
168             }
169         }
170     }
```

References gko::accessor::row_major< ValueType, Dimensionality >::lengths.

8.84.2.2 length()

```
template<typename ValueType , size_type Dimensionality>
constexpr size_type gko::accessor::row_major< ValueType, Dimensionality >::length (
            size_type dimension ) const  [inline], [constexpr]
```

Returns the length in dimension `dimension`.

Parameters

dimension	a dimension index

Returns

length in dimension `dimension`

References gko::accessor::row_major< ValueType, Dimensionality >::lengths.

8.84.2.3 operator()() [1/2]

```
template<typename ValueType , size_type Dimensionality>
constexpr range<row_major> gko::accessor::row_major< ValueType, Dimensionality >::operator()
(
            const span & rows,
            const span & cols ) const  [inline], [constexpr]
```

Returns the sub-range spanning the range (rows, cols)

Parameters

rows	row span
cols	column span

Returns

sub-range spanning the range (rows, cols)

References gko::span::begin, gko::accessor::row_major< ValueType, Dimensionality >::data, gko::span::end, gko::span::is_valid(), gko::accessor::row_major< ValueType, Dimensionality >::lengths, and gko::accessor::row↩_major< ValueType, Dimensionality >::stride.

8.84.2.4 operator()() [2/2]

```
template<typename ValueType , size_type Dimensionality>
constexpr value_type& gko::accessor::row_major< ValueType, Dimensionality >::operator() (
            size_type row,
            size_type col ) const  [inline], [constexpr]
```

Returns the data element at position (row, col)

Parameters

row	row index
col	column index

Returns

data element at (row, col)

References gko::accessor::row_major< ValueType, Dimensionality >::data, gko::accessor::row_major< ValueType, Dimensionality >::lengths, and gko::accessor::row_major< ValueType, Dimensionality >::stride.

The documentation for this class was generated from the following file:

- ginkgo/core/base/range_accessors.hpp (f1a4eb68)

8.85 gko::matrix::Sellp< ValueType, IndexType > Class Template Reference

SELL-P is a matrix format similar to ELL format.

```
#include <ginkgo/core/matrix/sellp.hpp>
```

Public Member Functions

- void read (const mat_data &data) override
  Reads a matrix from a matrix_data structure.
- void write (mat_data &data) const override
  Writes a matrix to a matrix_data structure.
- value_type * get_values () noexcept
  Returns the values of the matrix.
- const value_type * get_const_values () const noexcept
  Returns the values of the matrix.
- index_type * get_col_idxs () noexcept
  Returns the column indexes of the matrix.
- const index_type * get_const_col_idxs () const noexcept
  Returns the column indexes of the matrix.
- size_type * get_slice_lengths () noexcept
  Returns the lengths(columns) of slices.
- const size_type * get_const_slice_lengths () const noexcept

Returns the lengths(columns) of slices.

- size_type * get_slice_sets () noexcept

  Returns the offsets of slices.
- const size_type * get_const_slice_sets () const noexcept

  Returns the offsets of slices.
- size_type get_slice_size () const noexcept

  Returns the size of a slice.
- size_type get_stride_factor () const noexcept

  Returns the stride factor(t) of SELL-P.
- size_type get_total_cols () const noexcept

  Returns the total column number.
- size_type get_num_stored_elements () const noexcept

  Returns the number of elements explicitly stored in the matrix.
- value_type & val_at (size_type row, size_type slice_set, size_type idx) noexcept

  Returns the `idx`-th non-zero element of the `row`-th row with `slice_set` slice set.
- value_type val_at (size_type row, size_type slice_set, size_type idx) const noexcept

  Returns the `idx`-th non-zero element of the `row`-th row with `slice_set` slice set.
- index_type & col_at (size_type row, size_type slice_set, size_type idx) noexcept

  Returns the `idx`-th column index of the `row`-th row with `slice_set` slice set.
- index_type col_at (size_type row, size_type slice_set, size_type idx) const noexcept

  Returns the `idx`-th column index of the `row`-th row with `slice_set` slice set.

8.85.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Sellp< ValueType, IndexType >

SELL-P is a matrix format similar to ELL format.

The difference is that SELL-P format divides rows into smaller slices and store each slice with ELL format.

Template Parameters

ValueType	precision of matrix elements
IndexType	precision of matrix indexes

8.85.2 Member Function Documentation

8.85.2.1 col_at() [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type gko::matrix::Sellp< ValueType, IndexType >::col_at (
            size_type row,
            size_type slice_set,
            size_type idx ) const  [inline], [noexcept]
```

Returns the `idx`-th column index of the `row`-th row with `slice_set` slice set.

Parameters

row	the row of the requested element in the slice
slice_set	the slice set of the slice
idx	the idx-th stored element of the row in the slice

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the CPU results in a runtime error)

```
260     {
261         return this
262             ->get_const_col_idxs()[this->linearize_index(row, slice_set, idx)];
263     }
```

8.85.2.2 col_at() [2/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type& gko::matrix::Sellp< ValueType, IndexType >::col_at (
             size_type row,
             size_type slice_set,
             size_type idx )  [inline], [noexcept]
```

Returns the `idx`-th column index of the `row`-th row with `slice_set` slice set.

Parameters

row	the row of the requested element in the slice
slice_set	the slice set of the slice
idx	the idx-th stored element of the row in the slice

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the CPU results in a runtime error)

References gko::matrix::Sellp< ValueType, IndexType >::get_col_idxs().

8.85.2.3 get_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Sellp< ValueType, IndexType >::get_col_idxs ( )  [inline], [noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

References gko::Array< ValueType >::get_data().

Referenced by gko::matrix::Sellp< ValueType, IndexType >::col_at().

8.85.2.4 get_const_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Sellp< ValueType, IndexType >::get_const_col_idxs ( ) const
[inline], [noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.85.2.5 get_const_slice_lengths()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const size_type* gko::matrix::Sellp< ValueType, IndexType >::get_const_slice_lengths ( ) const
[inline], [noexcept]
```

Returns the lengths(columns) of slices.

Returns

the lengths(columns) of slices.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.85.2.6 get_const_slice_sets()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const size_type* gko::matrix::Sellp< ValueType, IndexType >::get_const_slice_sets ( ) const
[inline], [noexcept]
```

Returns the offsets of slices.

Returns

the offsets of slices.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.85.2.7 get_const_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::Sellp< ValueType, IndexType >::get_const_values ( ) const  [inline],
[noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.85.2.8 get_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Sellp< ValueType, IndexType >::get_num_stored_elements ( ) const  [inline],
[noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

References gko::Array< ValueType >::get_num_elems().

8.85.2.9 get_slice_lengths()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type* gko::matrix::Sellp< ValueType, IndexType >::get_slice_lengths ( )  [inline], [noexcept]
```

Returns the lengths(columns) of slices.

Returns

the lengths(columns) of slices.

References gko::Array< ValueType >::get_data().

8.85.2.10 get_slice_sets()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type* gko::matrix::Sellp< ValueType, IndexType >::get_slice_sets ( )  [inline], [noexcept]
```

Returns the offsets of slices.

Returns

the offsets of slices.

References gko::Array< ValueType >::get_data().

8.85.2.11 get_slice_size()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Sellp< ValueType, IndexType >::get_slice_size ( ) const  [inline],
[noexcept]
```

Returns the size of a slice.

Returns

the size of a slice.

8.85.2.12 get_stride_factor()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Sellp< ValueType, IndexType >::get_stride_factor ( ) const  [inline],
[noexcept]
```

Returns the stride factor(t) of SELL-P.

Returns

the stride factor(t) of SELL-P.

8.85.2.13 get_total_cols()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Sellp< ValueType, IndexType >::get_total_cols ( ) const  [inline],
[noexcept]
```

Returns the total column number.

Returns

the total column number.

8.85.2.14 get_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::Sellp< ValueType, IndexType >::get_values ( )  [inline], [noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

References gko::Array< ValueType >::get_data().

8.85.2.15 read()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Sellp< ValueType, IndexType >::read (
            const mat_data & data )  [override], [virtual]
```

Reads a matrix from a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::ReadableFromMatrixData< ValueType, IndexType >.

8.85.2.16 val_at() [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type gko::matrix::Sellp< ValueType, IndexType >::val_at (
            size_type row,
            size_type slice_set,
            size_type idx ) const  [inline], [noexcept]
```

Returns the `idx`-th non-zero element of the `row`-th row with `slice_set` slice set.

Parameters

row	the row of the requested element in the slice
slice_set	the slice set of the slice
idx	the idx-th stored element of the row in the slice

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the CPU results in a runtime error)

References gko::Array< ValueType >::get_const_data().

8.85.2.17 val_at() [2/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type& gko::matrix::Sellp< ValueType, IndexType >::val_at (
              size_type row,
              size_type slice_set,
              size_type idx )  [inline], [noexcept]
```

Returns the `idx`-th non-zero element of the `row`-th row with `slice_set` slice set.

Parameters

row	the row of the requested element in the slice
slice_set	the slice set of the slice
idx	the idx-th stored element of the row in the slice

Note

the method has to be called on the same Executor the matrix is stored at (e.g. trying to call this method on a GPU matrix from the CPU results in a runtime error)

References gko::Array< ValueType >::get_data().

8.85.2.18 write()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Sellp< ValueType, IndexType >::write (
              mat_data & data ) const  [override], [virtual]
```

Writes a matrix to a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::WritableToMatrixData< ValueType, IndexType >.

The documentation for this class was generated from the following files:

- ginkgo/core/matrix/csr.hpp (b8bd4773)
- ginkgo/core/matrix/sellp.hpp (8045ac75)

8.86 gko::span Struct Reference

A span is a lightweight structure used to create sub-ranges from other ranges.

`#include <ginkgo/core/base/range.hpp>`

Public Member Functions

- constexpr span (size_type point) noexcept

  Creates a span representing a point `point`.
- constexpr span (size_type begin, size_type end) noexcept

  Creates a span.
- constexpr bool is_valid () const

  Checks if a span is valid.

Public Attributes

- const size_type begin

  Beginning of the span.
- const size_type end

  End of the span.

8.86.1 Detailed Description

A span is a lightweight structure used to create sub-ranges from other ranges.

A `span s` represents a contiguous set of indexes in one dimension of the range, starting on index `s.begin` (inclusive) and ending at index `s.end` (exclusive). A span is only valid if its starting index is smaller than its ending index.

Spans can be compared using the `==` and `!=` operators. Two spans are identical if both their `begin` and `end` values are identical.

Spans also have two distinct partial orders defined on them:

1. `x < y` (`y > x`) if and only if `x.end < y.begin`
2. `x <= y` (`y >= x`) if and only if `x.end <= y.begin`

Note that the orders are in fact partial - there are spans `x` and `y` for which none of the following inequalities holds: `x < y`, `x > y`, `x == y`, `x <= y`, `x >= y`. An example are spans `span{0, 2}` and `span{1, 3}`.

In addition, `<=` is a distinct order from `<`, and not just an extension of the strict order to its weak equivalent. Thus, `x <= y` is not equivalent to `x < y || x == y`.

8.86.2 Constructor & Destructor Documentation

8.86.2.1 span() [1/2]

```
constexpr gko::span::span (
            size_type point )  [inline], [constexpr], [noexcept]
```

Creates a span representing a point `point`.

The `begin` of this span is set to `point`, and the `end` to `point + 1`.

Parameters

point	the point which the span represents

8.86.2.2 span() [2/2]

```
constexpr gko::span::span (
            size_type begin,
            size_type end )  [inline], [constexpr], [noexcept]
```

Creates a span.

Parameters

begin	the beginning of the span
end	the end of the span

References begin.

8.86.3 Member Function Documentation

8.86.3.1 is_valid()

```
constexpr bool gko::span::is_valid ( ) const  [inline], [constexpr]
```

Checks if a span is valid.

Returns

true if and only if `this->begin < this->end`

References begin, and end.

Referenced by gko::accessor::row_major< ValueType, Dimensionality >::operator()().

The documentation for this struct was generated from the following file:

- ginkgo/core/base/range.hpp (f1a4eb68)

8.87 gko::matrix::SparsityCsr< ValueType, IndexType > Class Template Reference

SparsityCsr is a matrix format which stores only the sparsity pattern of a sparse matrix by compressing each row of the matrix (compressed sparse row format).

`#include <ginkgo/core/matrix/sparsity_csr.hpp>`

Public Member Functions

- void read (const mat_data &data) override

  Reads a matrix from a matrix_data structure.
- void write (mat_data &data) const override

  Writes a matrix to a matrix_data structure.
- std::unique_ptr< LinOp > transpose () const override

  Returns a LinOp representing the transpose of the Transposable object.
- std::unique_ptr< LinOp > conj_transpose () const override

  Returns a LinOp representing the conjugate transpose of the Transposable object.
- std::unique_ptr< SparsityCsr > to_adjacency_matrix () const

  Transforms the sparsity matrix to an adjacency matrix.
- void sort_by_column_index ()

  Sorts each row by column index.
- index_type * get_col_idxs () noexcept

  Returns the column indices of the matrix.
- const index_type * get_const_col_idxs () const noexcept

  Returns the column indices of the matrix.
- index_type * get_row_ptrs () noexcept

  Returns the row pointers of the matrix.
- const index_type * get_const_row_ptrs () const noexcept

  Returns the row pointers of the matrix.
- value_type * get_value () noexcept

  Returns the value stored in the matrix.
- const value_type * get_const_value () const noexcept

  Returns the value stored in the matrix.
- size_type get_num_nonzeros () const noexcept

  Returns the number of elements explicitly stored in the matrix.

8.87.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::SparsityCsr< ValueType, IndexType >

SparsityCsr is a matrix format which stores only the sparsity pattern of a sparse matrix by compressing each row of the matrix (compressed sparse row format).

The values of the nonzero elements are stored as a value array of length 1. All the values in the matrix are equal to this value. By default, this value is set to 1.0. A row pointer array also stores the linearized starting index of each row. An additional column index array is used to identify the column where a nonzero is present.

Template Parameters

ValueType	precision of vectors in apply
IndexType	precision of matrix indexes

8.87.2 Member Function Documentation

8.87.2.1 conj_transpose()

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::unique_ptr<LinOp> gko::matrix::SparsityCsr< ValueType, IndexType >::conj_transpose ( )
const  [override], [virtual]
```

Returns a LinOp representing the conjugate transpose of the Transposable object.

Returns

a pointer to the new conjugate transposed object

Implements gko::Transposable.

8.87.2.2 get_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::SparsityCsr< ValueType, IndexType >::get_col_idxs ( )  [inline],
[noexcept]
```

Returns the column indices of the matrix.

Returns

the column indices of the matrix.

```
126 { return col_idxs_.get_data(); }
```

References gko::Array< ValueType >::get_data().

8.87.2.3 get_const_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::SparsityCsr< ValueType, IndexType >::get_const_col_idxs ( )
const  [inline], [noexcept]
```

Returns the column indices of the matrix.

Returns

the column indices of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.87.2.4 get_const_row_ptrs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::SparsityCsr< ValueType, IndexType >::get_const_row_ptrs ( )
const  [inline], [noexcept]
```

Returns the row pointers of the matrix.

Returns

the row pointers of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.87.2.5 get_const_value()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::SparsityCsr< ValueType, IndexType >::get_const_value ( ) const
[inline], [noexcept]
```

Returns the value stored in the matrix.

Returns

the value of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

8.87.2.6 get_num_nonzeros()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::SparsityCsr< ValueType, IndexType >::get_num_nonzeros ( ) const  [inline],
[noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

References gko::Array< ValueType >::get_num_elems().

8.87.2.7 get_row_ptrs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::SparsityCsr< ValueType, IndexType >::get_row_ptrs ( )  [inline],
[noexcept]
```

Returns the row pointers of the matrix.

Returns

the row pointers of the matrix.

References gko::Array< ValueType >::get_data().

8.87.2.8 get_value()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::SparsityCsr< ValueType, IndexType >::get_value ( )  [inline], [noexcept]
```

Returns the value stored in the matrix.

Returns

the value of the matrix.

References gko::Array< ValueType >::get_data().

8.87.2.9 read()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::SparsityCsr< ValueType, IndexType >::read (
            const mat_data & data )  [override], [virtual]
```

Reads a matrix from a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::ReadableFromMatrixData< ValueType, IndexType >.

8.87.2.10 to_adjacency_matrix()

```
template<typename ValueType = default_precision, typename IndexType = int32>
```

```
std::unique_ptr<SparsityCsr> gko::matrix::SparsityCsr< ValueType, IndexType >::to_adjacency↩
_matrix ( ) const
```

Transforms the sparsity matrix to an adjacency matrix.

As the adjacency matrix has to be square, the input SparsityCsr matrix for this function to work has to be square.

Note

The adjacency matrix in this case is the sparsity pattern but with the diagonal ones removed. This is mainly used for the reordering/partitioning as taken in by graph libraries such as METIS.

8.87.2.11 transpose()

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::unique_ptr<LinOp> gko::matrix::SparsityCsr< ValueType, IndexType >::transpose ( ) const
[override], [virtual]
```

Returns a LinOp representing the transpose of the Transposable object.

Returns

a pointer to the new transposed object

Implements gko::Transposable.

8.87.2.12 write()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::SparsityCsr< ValueType, IndexType >::write (
            mat_data & data ) const  [override], [virtual]
```

Writes a matrix to a matrix_data structure.

Parameters

data	the matrix_data structure

Implements gko::WritableToMatrixData< ValueType, IndexType >.

The documentation for this class was generated from the following files:

- ginkgo/core/matrix/csr.hpp (b8bd4773)
- ginkgo/core/matrix/sparsity_csr.hpp (b8bd4773)

8.88 gko::stopping_status Class Reference

This class is used to keep track of the stopping status of one vector.

`#include <ginkgo/core/stop/stopping_status.hpp>`

Public Member Functions

- bool has_stopped () const noexcept

  Check if any stopping criteria was fulfilled.
- bool has_converged () const noexcept

  Check if convergence was reached.
- bool is_finalized () const noexcept

  Check if the corresponding vector stores the finalized result.
- uint8 get_id () const noexcept

  Get the id of the stopping criterion which caused the stop.
- void reset () noexcept

  Clear all flags.
- void stop (uint8 id, bool set_finalized=true) noexcept

  Call if a stop occured due to a hard limit (and convergence was not reached).
- void converge (uint8 id, bool set_finalized=true) noexcept

  Call if convergence occured.
- void finalize () noexcept

  Set the result to be finalized (it needs to be stopped or converged first).

Friends

- bool operator== (const stopping_status &x, const stopping_status &y) noexcept

  Checks if two stopping statuses are equivalent.
- bool operator!= (const stopping_status &x, const stopping_status &y) noexcept

  Checks if two stopping statuses are different.

8.88.1 Detailed Description

This class is used to keep track of the stopping status of one vector.

8.88.2 Member Function Documentation

8.88.2.1 converge()

```
void gko::stopping_status::converge (
            uint8 id,
            bool set_finalized = true )  [inline], [noexcept]
```

Call if convergence occured.

Parameters

id	id of the stopping criteria.
set_finalized	Controls if the current version should count as finalized (set to true) or not (set to false).

References has_stopped().

8.88.2.2 get_id()

```
uint8 gko::stopping_status::get_id ( ) const  [inline], [noexcept]
```

Get the id of the stopping criterion which caused the stop.

Returns

Returns the id of the stopping criterion which caused the stop.

Referenced by has_stopped().

8.88.2.3 has_converged()

```
bool gko::stopping_status::has_converged ( ) const  [inline], [noexcept]
```

Check if convergence was reached.

Returns

Returns true if convergence was reached.

8.88.2.4 has_stopped()

```
bool gko::stopping_status::has_stopped ( ) const  [inline], [noexcept]
```

Check if any stopping criteria was fulfilled.

Returns

Returns true if any stopping criteria was fulfilled.

References get_id().

Referenced by converge(), finalize(), and stop().

8.88.2.5 is_finalized()

```
bool gko::stopping_status::is_finalized ( ) const  [inline], [noexcept]
```

Check if the corresponding vector stores the finalized result.

Returns

Returns true if the corresponding vector stores the finalized result.

8.88.2.6 stop()

```
void gko::stopping_status::stop (
            uint8 id,
            bool set_finalized = true )  [inline], [noexcept]
```

Call if a stop occured due to a hard limit (and convergence was not reached).

Parameters

id	id of the stopping criteria.
set_finalized	Controls if the current version should count as finalized (set to true) or not (set to false).

References has_stopped().

8.88.3 Friends And Related Function Documentation

8.88.3.1 operator"!=

```
bool operator!= (
            const stopping_status & x,
            const stopping_status & y )  [friend]
```

Checks if two stopping statuses are different.

Parameters

x	a stopping status
y	a stopping status

Returns

true if and only if `!(x == y)`

8.88.3.2 operator==

```
bool operator== (
            const stopping_status & x,
            const stopping_status & y )  [friend]
```

Checks if two stopping statuses are equivalent.

Parameters

x	a stopping status
y	a stopping status

Returns

true if and only if both x and y have the same mask and converged and finalized state

The documentation for this class was generated from the following file:

- ginkgo/core/stop/stopping_status.hpp (f1a4eb68)

8.89 gko::matrix::Hybrid< ValueType, IndexType >::strategy_type Class Reference

strategy_type is to decide how to set the hybrid config.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Public Member Functions

- strategy_type ()

  Creates a strategy_type.
- void compute_hybrid_config (const Array< size_type > &row_nnz, size_type *ell_num_stored_elements_per_row, size_type *coo_nnz)

  Computes the config of the Hybrid matrix (ell_num_stored_elements_per_row and coo_nnz).
- size_type get_ell_num_stored_elements_per_row () const noexcept

  Returns the number of stored elements per row of the ell part.
- size_type get_coo_nnz () const noexcept

  Returns the number of nonzeros of the coo part.
- virtual size_type compute_ell_num_stored_elements_per_row (Array< size_type > *row_nnz) const =0

  Computes the number of stored elements per row of the ell part.

8.89.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >::strategy_type

strategy_type is to decide how to set the hybrid config.

It computes the number of stored elements per row of the ell part and then set the number of residual nonzeros as the number of nonzeros of the coo part.

The practical strategy method should inherit strategy_type and implement its `compute_ell_num_stored_elements_per_row` function.

8.89.2 Member Function Documentation

8.89.2.1 compute_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
virtual size_type gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_ell_num_stored_elements_per_row (
            Array< size_type > * row_nnz ) const  [pure virtual]
```

Computes the number of stored elements per row of the ell part.

Parameters

row_nnz	the number of nonzeros of each row

Returns

the number of stored elements per row of the ell part

Implemented in gko::matrix::Hybrid< ValueType, IndexType >::automatic, gko::matrix::Hybrid< ValueType, IndexType >::minimal_sto gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit, gko::matrix::Hybrid< ValueType, IndexType >::imbalance_ and gko::matrix::Hybrid< ValueType, IndexType >::column_limit.

Referenced by gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_hybrid_config().

8.89.2.2 compute_hybrid_config()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_hybrid_config (
            const Array< size_type > & row_nnz,
            size_type * ell_num_stored_elements_per_row,
            size_type * coo_nnz )  [inline]
```

Computes the config of the Hybrid matrix (ell_num_stored_elements_per_row and coo_nnz).

For now, it copies row_nnz to the reference executor and performs all operations on the reference executor.

Parameters

row_nnz	the number of nonzeros of each row
ell_num_stored_elements_per_row	the output number of stored elements per row of the ell part
coo_nnz	the output number of nonzeros of the coo part

References gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_ell_num_stored_elements_↩ per_row(), gko::Array< ValueType >::get_executor(), and gko::Array< ValueType >::get_num_elems().

8.89.2.3 get_coo_nnz()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::get_coo_nnz ( ) const
[inline], [noexcept]
```

Returns the number of nonzeros of the coo part.

Returns

the number of nonzeros of the coo part

8.89.2.4 get_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::get_ell_num_stored_↩
elements_per_row ( ) const  [inline], [noexcept]
```

Returns the number of stored elements per row of the ell part.

Returns

the number of stored elements per row of the ell part

The documentation for this class was generated from the following file:

- ginkgo/core/matrix/hybrid.hpp (3e51a52b)

8.90 gko::log::Stream< ValueType > Class Template Reference

Stream is a Logger which logs every event to a stream.

```
#include <ginkgo/core/log/stream.hpp>
```

Static Public Member Functions

- static std::unique_ptr< Stream > create (std::shared_ptr< const Executor > exec, const Logger::mask_type &enabled_events=Logger::all_events_mask, std::ostream &os=std::cout, bool verbose=false)

  Creates a Stream logger.

8.90.1 Detailed Description

template<typename ValueType = default_precision>
class gko::log::Stream< ValueType >

Stream is a Logger which logs every event to a stream.

This can typically be used to log to a file or to the console.

Template Parameters

ValueType	the type of values stored in the class (i.e. ValueType template parameter of the concrete Loggable this class will log)

8.90.2 Member Function Documentation

8.90.2.1 create()

```
template<typename ValueType = default_precision>
static std::unique_ptr<Stream> gko::log::Stream< ValueType >::create (
            std::shared_ptr< const Executor > exec,
            const Logger::mask_type & enabled_events = Logger::all_events_mask,
            std::ostream & os = std::cout,
            bool verbose = false )  [inline], [static]
```

Creates a Stream logger.

This dynamically allocates the memory, constructs the object and returns an std::unique_ptr to this object.

Parameters

exec	the executor
enabled_events	the events enabled for this logger. By default all events.
os	the stream used for this logger
verbose	whether we want detailed information or not. This includes always printing residuals and other information which can give a large output.

Returns

an std::unique_ptr to the the constructed object

```
178     {
179         return std::unique_ptr<Stream>(
180             new Stream(exec, enabled_events, os, verbose));
181     }
```

The documentation for this class was generated from the following file:

- ginkgo/core/log/stream.hpp (f1a4eb68)

8.91 gko::StreamError Class Reference

StreamError is thrown if accessing a stream failed.

`#include <ginkgo/core/base/exception.hpp>`

Public Member Functions

- StreamError (const std::string &file, int line, const std::string &func, const std::string &message)

  Initializes a file access error.

8.91.1 Detailed Description

StreamError is thrown if accessing a stream failed.

8.91.2 Constructor & Destructor Documentation

8.91.2.1 StreamError()

```
gko::StreamError::StreamError (
            const std::string & file,
            int line,
            const std::string & func,
            const std::string & message )  [inline]
```

Initializes a file access error.

Parameters

file	The name of the offending source file
line	The source code line number where the error occurred
func	The name of the function that tried to access the file
message	The error message

The documentation for this class was generated from the following file:

- ginkgo/core/base/exception.hpp (8fbad33a)

8.92 gko::temporary_clone< T > Class Template Reference

A temporary_clone is a special smart pointer-like object that is designed to hold an object temporarily copied to another executor.

`#include <ginkgo/core/base/utils.hpp>`

Public Member Functions

- temporary_clone (std::shared_ptr< const Executor > exec, pointer ptr)

  Creates a temporary_clone.
- T * get () const

  Returns the object held by temporary_clone.
- T * operator-> () const

  Calls a method on the underlying object.

8.92.1 Detailed Description

template<typename T>
class gko::temporary_clone< T >

A temporary_clone is a special smart pointer-like object that is designed to hold an object temporarily copied to another executor.

After the temporary_clone goes out of scope, the stored object will be copied back to its original location. This class is optimized to avoid copies if the object is already on the correct executor, in which case it will just hold a reference to that object, without performing the copy.

Template Parameters

T	the type of object held in the temporary_clone

8.92.2 Constructor & Destructor Documentation

8.92.2.1 temporary_clone()

```
template<typename T >
gko::temporary_clone< T >::temporary_clone (
            std::shared_ptr< const Executor > exec,
            pointer ptr )  [inline], [explicit]
```

Creates a temporary_clone.

Parameters

exec	the executor where the clone will be created
ptr	a pointer to the object of which the clone will be created

References gko::clone().

8.92.3 Member Function Documentation

8.92.3.1 get()

```
template<typename T >
T* gko::temporary_clone< T >::get ( ) const  [inline]
```

Returns the object held by temporary_clone.

Returns

the object held by temporary_clone

8.92.3.2 operator->()

```
template<typename T >
T* gko::temporary_clone< T >::operator-> ( ) const  [inline]
```

Calls a method on the underlying object.

Returns

the underlying object

The documentation for this class was generated from the following file:

- ginkgo/core/base/utils.hpp (4bde4271)

8.93 gko::stop::Time Class Reference

The Time class is a stopping criterion which stops the iteration process after a certain amout of time has passed.

```
#include <ginkgo/core/stop/time.hpp>
```

8.93.1 Detailed Description

The Time class is a stopping criterion which stops the iteration process after a certain amout of time has passed.

The documentation for this class was generated from the following file:

- ginkgo/core/stop/time.hpp (ea195fb4)

8.94 gko::Transposable Class Reference

Linear operators which support transposition should implement the Transposable interface.

`#include <ginkgo/core/base/lin_op.hpp>`

Public Member Functions

- virtual std::unique_ptr< LinOp > transpose () const =0

  Returns a LinOp representing the transpose of the Transposable object.
- virtual std::unique_ptr< LinOp > conj_transpose () const =0

  Returns a LinOp representing the conjugate transpose of the Transposable object.

8.94.1 Detailed Description

Linear operators which support transposition should implement the Transposable interface.

It provides two functionalities, the normal transpose and the conjugate transpose.

The normal transpose returns the transpose of the linear operator without changing any of its elements representing the operation, $B = A^T$.

The conjugate transpose returns the conjugate of each of the elements and additionally transposes the linear operator representing the operation, $B = A^H$.

8.94.1.1 Example: Transposing a Csr matrix:

```
 {c++}
//Transposing an object of LinOp type.
//The object you want to transpose.
auto op = matrix::Csr::create(exec);
//Transpose the object by first converting it to a transposable type.
auto trans = op->transpose();
```

8.94.2 Member Function Documentation

8.94.2.1 conj_transpose()

```
virtual std::unique_ptr<LinOp> gko::Transposable::conj_transpose ( ) const  [pure virtual]
```

Returns a LinOp representing the conjugate transpose of the Transposable object.

Returns

a pointer to the new conjugate transposed object

Implemented in gko::matrix::Csr< ValueType, IndexType >, gko::matrix::Dense< ValueType >, and gko::matrix::SparsityCsr< ValueT

8.94.2.2 transpose()

```
virtual std::unique_ptr<LinOp> gko::Transposable::transpose ( ) const  [pure virtual]
```

Returns a LinOp representing the transpose of the Transposable object.

Returns

a pointer to the new transposed object

Implemented in gko::matrix::Csr< ValueType, IndexType >, gko::matrix::Dense< ValueType >, and gko::matrix::SparsityCsr< ValueT

The documentation for this class was generated from the following file:

- ginkgo/core/base/lin_op.hpp (4c758394)

8.95 gko::stop::Criterion::Updater Class Reference

The Updater class serves for convenient argument passing to the Criterion's check function.

```
#include <ginkgo/core/stop/criterion.hpp>
```

Public Member Functions

- Updater (const Updater &)=delete

  Prevent copying and moving the object This is to enforce the use of argument passing and calling check at the same time.
- bool check (uint8 stoppingId, bool setFinalized, Array< stopping_status > *stop_status, bool *one_changed) const

  Calls the parent Criterion object's check method.

8.95.1 Detailed Description

The Updater class serves for convenient argument passing to the Criterion's check function.

The pattern used is a Builder, except Updater builds a function's arguments before calling the function itself, and does not build an object. This allows calling a Criterion's check in the form of: stop_criterion->update() .num_↩ iterations(num_iterations) .residual_norm(residual_norm) .residual(residual) .solution(solution) .check(converged);

If there is a need for a new form of data to pass to the Criterion, it should be added here.

8.95.2 Member Function Documentation

8.95.2.1 check()

```
bool gko::stop::Criterion::Updater::check (
            uint8 stoppingId,
            bool setFinalized,
            Array< stopping_status > * stop_status,
            bool * one_changed ) const  [inline]
```

Calls the parent Criterion object's check method.

References gko::stop::Criterion::check().

The documentation for this class was generated from the following file:

- ginkgo/core/stop/criterion.hpp (f0a50f96)

8.96 gko::solver::UpperTrs< ValueType, IndexType > Class Template Reference

UpperTrs is the triangular solver which solves the system U x = b, when U is an upper triangular matrix.

`#include <ginkgo/core/solver/upper_trs.hpp>`

Public Member Functions

- std::shared_ptr< const matrix::Csr< ValueType, IndexType > > get_system_matrix () const
  Gets the system operator (CSR matrix) of the linear system.

8.96.1 Detailed Description

template<typename ValueType = default_precision, typename IndexType = int32>
class gko::solver::UpperTrs< ValueType, IndexType >

UpperTrs is the triangular solver which solves the system U x = b, when U is an upper triangular matrix.

It works best when passing in a matrix in CSR format. If the matrix is not in CSR, then the generate step converts it into a CSR matrix. The generation fails if the matrix is not convertible to CSR.

Note

As the constructor uses the copy and convert functionality, it is not possible to create a empty solver or a solver with a matrix in any other format other than CSR, if none of the executor modules are being compiled with.

Template Parameters

ValueType	precision of matrix elements
IndexType	precision of matrix indices

8.96.2 Member Function Documentation

8.96.2.1 get_system_matrix()

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::shared_ptr<const matrix::Csr<ValueType, IndexType> > gko::solver::UpperTrs< ValueType,
IndexType >::get_system_matrix ( ) const  [inline]
```

Gets the system operator (CSR matrix) of the linear system.

Returns

the system operator (CSR matrix)

```
94      {
95          return system_matrix_;
96      }
```

The documentation for this class was generated from the following file:

- ginkgo/core/solver/upper_trs.hpp (c380ba80)

8.97 gko::version Struct Reference

This structure is used to represent versions of various Ginkgo modules.

```
#include <ginkgo/core/base/version.hpp>
```

Public Attributes

- const uint64 major

  The major version number.
- const uint64 minor

  The minor version number.
- const uint64 patch

  The patch version number.
- const char *const tag

  Addition tag string that describes the version in more detail.

8.97.1 Detailed Description

This structure is used to represent versions of various Ginkgo modules.

Version structures can be compared using the usual relational operators.

8.97.2 Member Data Documentation

8.97.2.1 tag

```
const char* const gko::version::tag
```

Addition tag string that describes the version in more detail.

It does not participate in comparisons.

Referenced by gko::operator<<().

The documentation for this struct was generated from the following file:

- ginkgo/core/base/version.hpp (9c2e5ae6)

8.98 gko::version_info Class Reference

Ginkgo uses version numbers to label new features and to communicate backward compatibility guarantees:

```
#include <ginkgo/core/base/version.hpp>
```

Static Public Member Functions

- static const version_info & get ()

  Returns an instance of version_info.

Public Attributes

- version header_version

  Contains version information of the header files.
- version core_version

  Contains version information of the core library.
- version reference_version

  Contains version information of the reference module.
- version omp_version

  Contains version information of the OMP module.
- version cuda_version

  Contains version information of the CUDA module.

8.98.1 Detailed Description

Ginkgo uses version numbers to label new features and to communicate backward compatibility guarantees:

1. Versions with different major version number have incompatible interfaces (parts of the earlier interface may not be present anymore, and new interfaces can appear).
2. Versions with the same major number X, but different minor numbers Y1 and Y2 numbers keep the same interface as version X.0.0, but additions to the interface in X.0.0 present in X.Y1.0 may not be present in X.Y2.0 and vice versa.
3. Versions with the same major an minor version numbers, but different patch numbers have exactly the same interface, but the functionality may be different (something that is not implemented or has a bug in an earlier version may have this implemented or fixed in a later version).

This structure provides versions of different parts of Ginkgo: the headers, the core and the kernel modules (reference, OpenMP, CUDA). To obtain an instance of version_info filled with information about the current version of Ginkgo, call the version_info::get() static method.

8.98.2 Member Function Documentation

8.98.2.1 get()

```
static const version_info& gko::version_info::get ( )  [inline], [static]
```

Returns an instance of version_info.

Returns

an instance of version info

8.98.3 Member Data Documentation

8.98.3.1 core_version

```
version gko::version_info::core_version
```

Contains version information of the core library.

This is the version of the static/shared library called "ginkgo".

8.98.3.2 cuda_version

```
version gko::version_info::cuda_version
```

Contains version information of the CUDA module.

This is the version of the static/shared library called "ginkgo_cuda".

8.98.3.3 omp_version

```
version gko::version_info::omp_version
```

Contains version information of the OMP module.

This is the version of the static/shared library called "ginkgo_omp".

8.98.3.4 reference_version

```
version gko::version_info::reference_version
```

Contains version information of the reference module.

This is the version of the static/shared library called "ginkgo_reference".

The documentation for this class was generated from the following file:

- ginkgo/core/base/version.hpp (9c2e5ae6)

8.99 gko::WritableToMatrixData< ValueType, IndexType > Class Template Reference

A LinOp implementing this interface can write its data to a matrix_data structure.

```
#include <ginkgo/core/base/lin_op.hpp>
```

Public Member Functions

- virtual void write (matrix_data< ValueType, IndexType > &data) const =0

  Writes a matrix to a matrix_data structure.

8.99.1 Detailed Description

template<typename ValueType, typename IndexType>
class gko::WritableToMatrixData< ValueType, IndexType >

A LinOp implementing this interface can write its data to a matrix_data structure.

8.99.2 Member Function Documentation

8.99.2.1 write()

```
template<typename ValueType, typename IndexType>
virtual void gko::WritableToMatrixData< ValueType, IndexType >::write (
            matrix_data< ValueType, IndexType > & data ) const  [pure virtual]
```

Writes a matrix to a matrix_data structure.

Parameters

data	the matrix_data structure

Implemented in gko::matrix::Hybrid< ValueType, IndexType >, gko::matrix::Csr< ValueType, IndexType >, gko::preconditioner::Jacobi< ValueType, IndexType >, gko::matrix::Coo< ValueType, IndexType >, gko::matrix::Ell< ValueType, Inde gko::matrix::Sellp< ValueType, IndexType >, and gko::matrix::SparsityCsr< ValueType, IndexType >.

The documentation for this class was generated from the following file:

- ginkgo/core/base/lin_op.hpp (4c758394)

Index