afft: a C++17 Wrapper Library for FFT-like Computations on Various Targets

David Bayer[1] and Jiri Jaros[1]
[1]Faculty of Information Technology, Brno University of Technology,
Centre of Excellence IT4Innovations, CZ

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

1 Introduction and Motivation

Fast Fourier Transformation (FFT) and other related transformations are very demanding and time-consuming computations. There are many C/C++ libraries focusing on providing an efficient FFT implementation on a specific hardware such as CPUs, GPUs and others. However, their tight specialization implies low portability. If an application is supposed to be multiplatform, either a several versions of the program or a wrapper around the FFT must be written. The *afft* library is a modern C++17 wrapper library addressing this problem, allowing to use most of features offered by the backend libraries while providing extra layer of safety checks and other features.

2 Five Steps to Execution

A lot of libraries split the computation into two separate phases. During the first one a plan object describing the transformation is created. This operation can be very expensive because the library often chooses the best of the implemented algorithms via measurements and allocates temporary workspace. The second phase consists of repetitively executing the plan with no more initialization penalties. The *afft* library copies this concept and splits the first phase into four.

2a Defining the Transformation

This phase consists of defining the mathematical properties of the transformation, independently on the target architecture. The user is required to specify:

- the direction of the transformation,
- the precision of the memory and execution datatypes,
- the shape of the data and
- the placement of the transformation.

The user may also select the axes along which the transformation shall be computed, and the normalization type applied to the results. Each transformation type may have its own parameters, e. g. DCT and DST types for DTT.

```
afft::dft::Parameters dftParameters
{
  .direction     = afft::Direction::forward,
  .precision     = afft::makePrecision<PrecT>(),
  .shape         = {{500, 250, 1020}},
  .axes          = {{1, 2}},
  .normalization = afft::Normalization::orthogonal,
  .placement     = afft::Placement::outOfPlace,
  .type          = afft::dft::Type::complexToComplex,
};
```

```
afft::cpu::Parameters cpuParameters
{
  .complexFormat   = afft::ComplexFormat::interleaved,
  .preserveSource  = true;
  .workspacePolicy = afft::WorkspacePolicy::performance,
  .memoryLayout    = {.srcStrides = afft::makeStrides(
                        {{500, 250, 1024}}),
                      .dstStrides = afft::makeTransposedStrides(
                        {{500, 1020, 256}}, {{0, 2, 1}})},
  .alignment       = afft::getAlignment(src, dst),
  .threadLimit     = 8;
};
```

2b Specifying the Target

The library distinguishes two types of targets - CPU and GPU. The computation can be performed on one or more targets on a single or multiple nodes (via MPI). The target parameters consist of:

- complex numbers format,
- prevent source buffer destruction flag,
- workspace policy parameter and
- memory layout according to a specified distribution.

Furthermore, each target has its own parameters such as memory alignment and thread limit for CPUs and device specification for GPUs. For MPI applications also the communicator must set.

2c Configuring the Backend Belection

The last part before baking the plan is specifying the plan initialization parameters which allow to influence the transformation backend selection. It allows the user to:

- constraint the considered backends,
- set the order in which the backends are evaluated,
- specify the backend selection strategy and
- plan initialization effort.

```
afft::InitParameters initParameters
{
  .backendMask      = (afft::Backend::fftw3 | afft::Backend::mkl),
  .backendInitOrder = {{afft::Backend::mkl, afft::Backend::fftw3}},
  .selectStrategy   = afft::SelectStrategy::best,
  .initEffort       = afft::InitEffort::high,
};
```

2d Baking the Plan

When all the information are gathered, the transformation plan may be created. It is done using the `makePlan` factory function. First, an internal description of the transformation is created and validated. Then a transformation backend is selected according to the passed strategy:

- first - selects the first backend which supports the given configuration or
- best - creates plans of all chosen backends and selects the fastest.

```
afft::Plan plan = afft::makePlan(dftParameters,
                                 cpuParameters,
                                 initParameters);
```

2e The Execution

The transformation can be executed using the plan's `execute` or `executeUnsafe` methods. Before the execution begins, several safety checks are performed to prevent an unwanted behaviour such as buffer validity and placement. On top of that the safe execution variant also checks the datatype of the buffers to match the expected values. Additional target parameters may be passed to the function. This way can be specified e. g. GPU's stream or workspace.

```
std::complex<PrecT>* src = ...;
std::complex<PrecT>* dst = ...;

plan.execute(src, dst, afft::cpu::ExecutionParameters{});
```

3 Conclusions

The *afft* library is an easy-to-use C++17 library for computing FFT-like transformations on CPUs and GPUs. It unifies the interface to already existing libraries, simplifies the portability of applications to various platforms and introduces new features. It is a versatile platform designed to be easily extended by other mathematical transformations, new targets and backend libraries.

4 Current and Future Work

Next steps in the development are to

- finalize the implementation including unit and module testing,
- deploy the library to real world applications, e. g. *k-Wave* project,
- create bindings for Python and MATLAB.

VSB TUO | IT4I

k-Wave
A MATLAB toolbox for the time-domain simulation of acoustic wave fields

European Innovation Council

citrus

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID:90254). This project has received funding from the European Unions Horizon Europe research and innovation programme under grant agreement No 101071008.}