ORE Compute Framework Interface

Acadia

19 June 2023

Document History

Date	Author	Comment
19 June 2023	Acadia	initial release
15 April 2024	Acadia	refactor context settings, add supportsDoublePrecision()

Contents

1 Overview

This paper describes how to implement the ORE compute framework interface. An implementation of this interface can be used to do calculations on external devices like GPUs from ORE engines. For example, the scripted trade module supports this interface to speed up sensitivity or backtest calculations.

The interface is defined in `QuantExt/qle/math/computeenvironment.hpp`. A reference implementation is given in `QuantExt/qle/math/openclenvironment.hpp/cpp`. The latter can can also serve as a template for new framework implementations.

The file `QuantExt/qle/math/computeenvironment.hpp` contains three class declarations:

- `ComputeFramework`: This is one of two interfaces that needs to be implemented to add a new compute framework. See 4.1 for more details.
- `ComputeContext`: This is the second interface that needs to be implemented to add a new compute framework. See 4.2 for more details.
- `ComputeEnvironment`: This is a singleton exposing the compute frameworks to ORE . See 3 for more details.

The typical usage of these interfaces from ORE engines is described in 6.

2 CMake setting to enable a compute framework

We recommend to introduce a cmake setting that allows to enable / disable the build of each compute framework. The reason is that not every framework is supported on each machine.

For example, the build of the OpenCL framework requires the installation of an open cl driver and header files for development. The OpenCL framework is enabled with `-D ORE_ENABLE_OPENCL` in the cmake configure step.

The following section in `QuantExt/qle/CMakeLists.txt` takes care of the linking against the open cl driver on apple, linux and windows platforms:

```
if(ORE_ENABLE_OPENCL)
  if(APPLE)
    target_link_libraries(${QLE_LIB_NAME} "-framework OpenCL")
  else()
    find_package(OpenCL REQUIRED)
    target_link_libraries(${QLE_LIB_NAME} OpenCL::OpenCL)
  endif()
endif()
```

Furthermore we define a compiler macro in `cmake/commonSettings.cmake` whenever we enable OpenCL in the cmake build:

```
# set compiler macro if open cl is enabled
if (ORE_ENABLE_OPENCL)
```

```
  add_compile_definitions(ORE_ENABLE_OPENCL)
endif()
```

The latter flag is used in `QuantExt/qle/openclenvironment.cpp` to include the OpenCL headers

```
#ifdef ORE_ENABLE_OPENCL
#ifdef __APPLE__
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif
#endif
```

and also to distinguish a build with and without support for OpenCL. Notice that a minimal implementation of the compute framework interface is required even when the framework is disabled in the build. See 4.1 for more details on this.

3 The ComputeEnvironment singleton

The `ComputeEnvironment` is a thread local singleton that exposes external compute frameworks to ORE code. A new compute framework has to be registered to this singleton by instantiating a raw pointer to the framework and adding it to the frameworks container:

```
void ComputeEnvironment::reset() {
    // ...
    frameworks_.push_back(new OpenClFramework());
}
```

The `ComputeEnvironment` singleton provides the following methods:

- `getAvailableDevices()`: returns a set of descriptions of available devices for all registered frameworks. A description is a triplet like e.g. `OpenCL/Apple/AMD Radeon Pro 5500M Compute Engine` where the first component identifies the framework, and the second and third component the concrete compute device exposed by that framework. The second component is the name of the platform and the third the name of the device itself.
- `selectContext()`: selects a context to work with. A context corresponds one to one to a device within a framework.
- `context()`: returns a reference to the currently selected context
- `hasContext()`: returns true if a context was selected previously and can be accessed via `context()`

The fact that `ComputeEnvironment` is thread local means that each thread will access different instances of `ComputeFramework` and the `ComputeContext` instances within that framework. This means that the implementation of the methods in the latter two interfaces do not need to be thread-safe. However notice that calls into a lower level

library (like OpenCL) have to be made thread-safe dependent on the specification of that lower level library.

4 Implementation of the required interfaces

4.1 Implementation of the ComputeFramework interface

The `ComputeFramework` interface requires the implementation of the following methods:

- `getAvailableDevices()` should return the set of names of supported devices in that framework. By convention a device name is a triplet of the form `OpenCL/Apple/AMD Radeon Pro 5500M Compute Engine` where each component has the following meaning:
  - `OpenCL` the name of the framework. This name should be unique across all implementations of the `ComputeFramework` interface within ORE. If there are several implementations for the same framework, they should be distinguished by an additional description separated with an underscore, e.g. `OpenCL_OptimizedKernels`.
  - `Apple` the name of the platform.
  - `AMD Radeon Pro 5500M Compute Engine` the name of the actual device.

  Notice that the same device can appear under several frameworks, e.g. a GPU could be accessed both via an OpenCL driver or via a CUDA driver.
- `getContext()` should return a raw pointer to a `ComputeContext` implementation for a given device label or throw an error if the device label is not valid. See 4.2 for more details.
- `Destructor` The destructor of `ComputeFramework` is virtual and has an empty default implementation. It can be override in implementations to free ressources. For example, the `OpenClFramework` implementation stores raw pointers for each compute contexts that need to be deleted on destruction of the `OpenClFramework` instance.

If the build of a framework is *disabled* as described in 2, `getAvailableDevices()` should return an empty set. A call to `getContext()` should always throw a runtime exception in this case.

4.2 Implementation of the ComputeContext interface

The `ComputeContext` interface represents a context in which calculations can be performed. The relation to `ComputeFramework` is that `ComputeFramework` provides a pointer to `ComputeContext` for each device within the framework. There is a one to one relationship between contexts and devices. The `ComputeContext` interface requires the implementation of the following methods:

- `init()` initialize the context. This method is called every time `selectContext()` is called in `ComputeEnvironment` and should be used to do one-off set up tasks. For example `OpenClContext` will create an OpenCL context and command queue

for a device if this was not done before. If `init()` was already called, subsequent calls do nothing in that implementation.

- `initiateCalculation()`: start a new calculation. The input parameters are:
  - `n`: The size of the vectors on which the calculations will be performed.
  - `id`: The id of the calculation. If `id` is not `0` the steps from a previous calculation identified by `id` will be replayed in the current calculation if the version of the calculation matches the previous calculation. In this case only the input variables and random variates (see below) need to be set to retrieve the results of the calculation calling `finalizeCalculation()`. If `id` is `0` on the other hand, a new calculation `id` will be generated and returned.
  - `version`: The version of a calculation. This is a freely choosable integer, which is only used to identify different versions. Usually the first version of a calculation will be 0, then next 1, etc.
  - `Settings`: A struct summarizing settings for the compute environment:
    - debug: a flag indicating whether debug information on the number of performed operations and timings for data copying, kernel building and calculations should be collected. Defaults to false.
    - useDoublePrecision: a flag indicating whether double precision should be used for calculations. Defaults to false.
    - rngSequenceType: the sequence type for random number generation. One of MersenneTwister, MersenneTwisterAntithetic, Sobol, Burley2020Sobol, SobolBrownianBridge, Burley2020SobolBrownianBridge
    - seed: the seed for the random number generator
    - regressionOrder: the regression order to be used within regression models

  The output parameter is a pair consisting of
  - `id`: the calculation id of the new calculation.
  - `newCalc`: a bool indicating whether the calculation matches a previously recorded calculation and will be replayed as described above.

  See 6 for more context on ids and versions of calculations.
- `createInputVariable()`: create and input variable (scalar or vector, the latter given as a raw pointer) and return the identifier of the variable.
- `createInputVariates()`: create input variates (normally distributed random variates), the parameters are
  - `dim` the dimension (typically the number of assets in a MC simulation)
  - `steps` the number of steps (typically the number of time steps in a MC simulation)

The output parameter is a vector of a vector of ids for the generated variates. The inner vector loops of the steps, the outer vector over the dimensions.

- `applyOperation()`: apply an operation to a list of arguments and return the id of the result. The list of operations is described in 4.3
- `freeVariable()`: indicate that the variable with the given id will no longer be referenced in subsequent calculations. The variable id can be reused as a new variable.
- `declareOutputVariable()`: declare a variable with given id as part of the output vector.
- `finalizeCalculation()`: execute the calculation and populate the given vector of vector of floats with the result. The inner vector is given as a raw pointer and must have the size of the calculation. The outer vector matches the output variables in the order they were declared before.
- `deviceInfo()`: provide key-value pairs that describes the device, only used for information purposes
- `supportsDoublePreicions()`: should return true if doule precision is supported, otherwise false
- `debugInfo()`: provide info on the number of elementary operations and timings on data copying, program build and calculations. This info is collected if a new calculation is started with flag debug set to true

The following order of calls to a `ComputeContext` from top to bottom is guaranteed:

1. `initiateCalculation()`: once
2. `createInputVariable()`: never, once, or several times, if a calc is replayed, the same number and type of input variables must be created (just with possibly different values)
3. `createInputVariates()`: called if the current calculation is not replayed from a previous calculation, then can be called never, once or several times
4. `applyOperation()`: only called if the current calculation is not replayed from a previous calculation, then can be called never, once, or several times
5. `freeVariable()`: only called if the current calculation is not replayed, then can be called never, once, or several times
6. `declareOutputVariable()`: only called if the current calculation is not replayed, then can be called never, once, or several times
7. `finalizeCalculation()`: once

4.3 Random Variable Op Codes

Table 1 summarizes the random variable operations that are defined in `QuantExt/qle/math/randomvariable_opcodes.hpp`. Each op code needs to be implemented in `ComputeContext::applyOperation()`.

OpCode	#args	description
Add	2	sum of two variables
Subtract	2	difference of two variables
Negative	1	negative of a variable
Mult	2	product of two variables
Div	2	quotient of two variables
ConditionalExpectation	n	conditional expectation of first argument given the rest of the arguments as regressors
IndicatorEq	2	indicator function for equality of two variables
IndicatorGt	2	indicator function for greater-than relation of two variables
IndicatorGeq	2	indicator function for greater-than-or-equal relation of two variables
Min	2	minimum of two variables
Max	2	maximum of two variables
Abs	1	absolute value of a variable
Exp	1	exponential of a variable
Sqrt	1	square root of a variable
Log	1	natural logarithm of a variable
Pow	2	power of basis, exponent
NormalCdf	1	normal cumulative distribuion
NormalPdf	1	normal distribution density

Table 1: Random Variable Op Codes

5 Unit tests

There are some unit tests testing all available external frameworks. The tests are located in `QuantExtTestSuite/ComputeEnvironmentTest`:

- `testEnvironmentInit`: test to initialize each framework
- `testSimpleCalc`: test a simple calculation
- `testLargeCalc`: test a larger scale calculation
- `testRngGeneration`: test random variate generation
- `test_ScriptedTrade_FX_TaRF_XAUUSD_sensi_ops`: test a (simulated) sensi calculation for an FX TaRF

The unit test also provide examples how an external calculation is orchestrated.

6 The typical usage of am external compute framework within ORE

This section describes how a compute framework is typically used from an ORE engine. We use `ScriptedInstrumentPricingEngineCG` as an example.

6.1 Context selection

If an external device is configured for a computation the first step is to make sure that there is a context for the device as in

```
if (!ComputeEnvironment::instance().hasContext()) {
    ComputeEnvironment::instance().selectContext(externalComputeDevice_);
}
```

where `externalComputeDevice_` is a string identifying the device.

6.2 Initiation of a calculation

Whenever a new computation i.e. the (re)pricing of a trade is requested a calculation is initiated

```
std::tie(externalCalculationId_, newExternalCalc) =
  ComputeEnvironment::instance().context()
      .initiateCalculation(model_->size(), externalCalculationId_, cgVersion_, settings);
```

providing

- `model_->size()`: the size of the vectors participating in the calculation
- `externalCalculationId_`: an external calculation id tied to the specific calculation, this is initially zero to get a new id and for subsequent calculations of the same trade this id is reused
- `cgVersion_`: a version number (initially zero). The version is incremented whenever a repricing requires a different sequence of operations to be performed, which is the case when the evaluation date changes
- `settings`: the settings to be used, see 4.1 for details. Notice that for a replayed calculation the settings must match the ones from the initial calculation.

and retrieving the external calc id and a flag indicating whether the calculation requires a resubmission of an operation sequence (`newExternalCalc` = `true`) or not (`newExternalCalc` = `false`)

6.3 Populating the input variables

The next step is to initialize the input variables. For scripted trades this is done using `ExternalRandomVariable` as a proxy class (replacing the usual `RandomVariable` class for "ordinary" calculations). The code snipped

```
valuesExternal[index] = ExternalRandomVariable((float)v);
```

creates an input variable with value `v` and stores the variable instance in a vector `values` at a specific position `index`. Notice that the latter `index` is independent of the

variable id within the compute context[1].

The former is done via the constructor

```
ExternalRandomVariable::ExternalRandomVariable(float v)
    : initialized_(true), v_(v) {
      id_ = ComputeEnvironment::instance().context().createInputVariable(v);
    }
```

calling the appropriate method in the current compute context and storing the provided `id` as a member of the external random variable.

6.4 Populating the input variates

The next step is the creating of random variates (if this is a new calculation and not replayed):

```
if (useExternalComputeFramework_ && newExternalCalc) {
    auto gen =
        ComputeEnvironment::instance().context().
                    createInputVariates(rv.size(), rv.front().size());
    for (Size k = 0; k < rv.size(); ++k) {
        for (Size j = 0; j < rv.front().size(); ++j)
            valuesExternal[rv[k][j]] = ExternalRandomVariable(gen[k][j]);
    }
}
```

where `k` loops over the dimensions (e.g. number of assets) and `j` loops over the time steps and `rv` is a vector of vectors storing the indices of the required random variates by dimensions and time steps. The external random variable contructor used in this case is

```
ExternalRandomVariable::ExternalRandomVariable(std::size_t id)
    : initialized_(true), id_(id) {}
```

simply storing the provided `id` that was previously generated by the call to `createInputVariates()` within the external random variable instance.

6.5 Apply the operations (for new calculation only)

The next step is to run the calculation if required, i.e. if the flag `newExternalCalc` is `true` (see above). This is done using the `forwardEvaluation()` algorithm on a computation graph `g`.

```
if (newExternalCalc) {
    forwardEvaluation(*g, valuesExternal, opsExternal_,
                      ExternalRandomVariable::deleter, useCachedSensis_,
                      opNodeRequirements_, keepNodes);
    ...
```

[1]having said this, the idea of using an integer id to identify variables is very similar in the scripted trade pricing engine and the compute context

The provided operations `opsExternal` are implemented via yet another constructor, e.g. the addition of two external random variates is represented by the lambda

```
ops.push_back([](const std::vector<const ExternalRandomVariable*>& args) {
    return ExternalRandomVariable(RandomVariableOpCode::Add, args);
});
```

using the external random variable constructor

```
ExternalRandomVariable::ExternalRandomVariable(
               const std::size_t randomVariableOpCode,
               const std::vector<const ExternalRandomVariable*>& args) {
    std::vector<std::size_t> argIds(args.size());
    std::transform(args.begin(), args.end(), argIds.begin(),
                   [](const ExternalRandomVariable* v) { return v->id(); });
    id_ = ComputeEnvironment::instance().context()
                   .applyOperation(randomVariableOpCode, argIds);
    initialized_ = true;
}
```

which calls into `applyOperation()` of the current compute context providing the appropriate op code and argument ids. The deleter of the ExternalRandomVariable is implemented as

```
std::function<void(ExternalRandomVariable&)> ExternalRandomVariable::deleter =
  std::function<void(ExternalRandomVariable&)>(
                                  [](ExternalRandomVariable& x) { x.clear(); });
```

which calls into `freeVariable()` in the compute context

```
void ExternalRandomVariable::clear() {
    ComputeEnvironment::instance().context().freeVariable(id_);
    initialized_ = false;
}
```

6.6 Declare the output (for new calculation only)

The next step is to declare the output by calling

```
valuesExternal[baseNpvNode].declareAsOutput();
```

which propagates the declaration to the compute context

```
void ExternalRandomVariable::declareAsOutput() const {
    ComputeEnvironment::instance().context().declareOutputVariable(id_);
}
```

and initializing a buffer storing the result of the computation

```
externalOutput_ = std::vector<std::vector<float>>(1,
                              std::vector<float>(model_->size()));
externalOutputPtr_ = std::vector<float*>(1, &externalOutput_.front()[0]);
```

6.7 Finalize the calculation

The final step which is the first one that is again independent of the `newExternalCalc` flag is to retrieve the result

```
ComputeEnvironment::instance().context().finalizeCalculation(externalOutputPtr_);
baseNpv_ = results_.value = externalAverage(externalOutput_[0]);
```

Here, `externalAverage()` computes the average of the retrieved output vector.