ORE Compute Framework Interface

Acadia

30 October 2025

Document History

Date	Author	Comment
19 June 2023	Peter Caspers	initial release
15 April 2024	Peter Caspers	refactor context settings, add supportsDoublePrecision()
22 July 2024	Peter Caspers	add documentation of OpenCL framework, update doc of unit tests

Contents

1 Overview **4**

2 CMake setting to enable a compute framework **4**

3 The ComputeEnvironment singleton **5**

4 Implementation of the required interfaces **6**
- 4.1 Implementation of the ComputeFramework interface 6
- 4.2 Implementation of the ComputeContext interface 6
- 4.3 Random Variable Op Codes 8

5 Unit tests **9**

6 The typical usage of am external compute framework within ORE **10**
- 6.1 Context selection 10
- 6.2 Initiation of a calculation 10
- 6.3 Populating the input variables 10
- 6.4 Populating the input variates 11
- 6.5 Apply the operations (for new calculation only) 11
- 6.6 Declare the output (for new calculation only) 12
- 6.7 Finalize the calculation 13

7 OpenCL Framework **13**
- 7.1 Overview 13
- 7.2 Random Number Generation 13
- 7.3 Conditional Expectation 14

8 GPU Code Generator **14**
- 8.1 Variable management 14
- 8.2 Kernel function parts generation 15
- 8.3 Local variable replacement 16
- 8.4 Elimination of replaced local variables from the values buffer 16

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
- `testSimpleCalcWithDoublePrecision`: test a simple calculation using double precision
- `testLargeCalc`: test a larger scale calculation
- `testRngGeneration`: test random variate generation
- `testReplayFlowError`: test replay of calculation with invalid calls into ComputeContext interface
- `testRngGenerationMt19937`: test MersenneTwister random number generation
- `testConditionalExpectation`: test conditional expectation calculation

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`.

[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

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

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.

7 OpenCL Framework

This section documents the OpenCL framework. It relies on the GPU Code Generator described in 8.

7.1 Overview

The OpenCL framework provides contexts for all OpenCL devices.

This framework creates one context per device, i.e. within a single ORE process there will be only once context [2] per device that is shared between threads. The reason for this design choice is the observation that running several contexts in paralell can lead to high GPU memory usage and ultimately ressource allocation errors on the GPU.

7.2 Random Number Generation

We support MersenneTwister repcliating the code from QuantLib.

When new variates are required in `createInputVariates()` the function `updateVariatesPool()` will be called. We maintain a pool of random variates, the size of this pool, i.e. the number of standard normal variables available in the pool, is stored in `variatesPoolSize_`[3].

[2] Note that "context" refers to the OpenCL term here, not the implementation of of the ORE interface ComputeContext by OpenClContext. The latter will be instantiated per thread as per ORE design of the thread-local ComputeEnvironment singleton as usual.

[3] The underlying assumption is that the seed for the rng is the same across all computations that are run against the OpenClContext.

When `updateVariatesPool()` is called first we build the necessary programs on the GPU side:

- `oreInvCumN()`: function that replicates QuantLib's `InverseCumulativeNormal`, but with a 32bit unsigned integer input that is converted to a $(0, 1)$ double or float within the function.
- `ore_seedInitialization()`: kernel that repcliates the seed initilization of QuantLib's `MersenneTwisterUniformRng`. Input is a seed s. The kernel populates the mt state which has size 624. The kernel is not parallelized, i.e. its global work size is 1.
- `ore_twist(__global ulong* mt)`: kernel that recpliates the state twist of QuantLib's `MersenneTwisterUniformRng`. The kernel is not parallelized, i.e. its global work size is 1.
- `ore_generate()`: kernel that recpliates the random number generation of QuantLib's `MersenneTwisterUniformRng` combined with a call to `oreInvCumN()` to convert the result to a normal variate. The kernel has global work size 624 (size of mt state).

Then the new variates are generated following these steps:

- a new variates pool buffer is allocated on the device. The size is aligned with a multiple of 624.
- the old variates pool buffer is copied to the new buffer.
- the new variates are generated in the new variates buffer using the kernel `ore_generate()`.

7.3 Conditional Expectation

Currently, conditional expectation is calculated on the host side, i.e. the necessary arguments are copied from device to host, the conditional expectation is calculated on the host and the result is copied back to the device. This is done using the `conditionalExpectation()` method for random variables, taking the regressand, regressors, filter, basis functions as inputs and using QR or SVD to estimate the linear regression.

Future dev will support calculation on the device.

8 GPU Code Generator

This is a generic code generator for GPU kernels and is e.g. used by the OpenCL framework.

8.1 Variable management

Input variables, input variates and (intermediate) result variables v_i are numbered as $i = 0, 1, 2, 3, \ldots$. They occur as contingent blocks in this sequence, first the input variables, then the variates and then the intermediate results. The numbering represented by i is translated to code generator internal variables with types input,

random variate, local and each with its own numbering starting at 0. The mapping between the external numbering and the latter internal numbering is done by `getVar()` (external to internal) resp. `getId()` (internal to external).

Output variables are declared by `declareOutputVariable()` and can be an input variable, input variate or intermediate result. They are not new variables, rather input variables, variates or intermediate results are marked as being an output to retrieve from the calculation. Output variables are ultimately ensured to be intermediate result variables v_i, and as such copied to the host and written to the output vector provided to the framework's `finalizeCalculation()` method.

Intermediate result variables are generated within `applyOperation()`. There is a distinction between pathwise operations and conditional expectation.

- for pathwise operations, a single intermediate result variable is generated
- for conditional expectation, no result is generated. The arguments to the conditional expectation and the result are ensured to be intermediate result variables and stored, so that they can be retrieved via conditionalExpectationVars(). It is then up to the framework to calculate the conditional expectation on the device or host.

The result variable ids (these are external ids) are generated by `generateResultId()`. This function will reuse variable ids that were freed by `freeVariable()`, provided that the freed variable is not an input variable or input variate or an argument or the result of an conditional expectation calculation.

The different variable types are represented as follows, `i` denoting the global work id.

- input variables: `x[offset + i]`, with x a buffer in the global address space
- input variates: `r[offset + i]`, with r a buffer in the global address space
- intermediate variables: `b[offset + i]`, with b a buffer in the global address space. It is refered to as the values buffer. Note that this buffer also contains conditional expectation arguments and results and all output variables.

8.2 Kernel function parts generation

A new kernel function is started if one of the following conditions are met:

- the max number of rhs-arguments (as a measure for code size) exceeds a given threshold `max_kernel_args`. This is done because we observe failing program builds when a kernel function gets too large.
- the result of a conditional expectation operation in the current kernel part is used as an argument to another operation. This is done because to calculate the conditional expectation result we need to stop the current kernel and invoke either another kernel on the device or do the claculation on the host (the latter requires copying the arguments of the conditional expectation to the host and the result back to the device).

8.3 Local variable replacement

The GPU code generator can replace intermediate variables by stackvariables $v0, v1, v2, \ldots$. If an intermediate variable is used frequently on the lhs or rhs of operations, replacing it by a stack variable can improve performance. This optimization follows the steps:

- For each kernel part that was determined in 8.2 we count the usage of intermediate result variables on the lhs or rhs of operations.
- We select the `max_local_vars` most frequently used variables that are also used at least `min_usage_threshold` times as variables to be replaced by stack variables.
- We determine the first usage on the lhs and rhs of each such variable within each kernel part. This information is used to determine whether the lhs replacement by a stack variable requires a declaration and whether rhs replacements by stack variables need an initialisation from the values buffer before their usage.
- We mark certain replaced variables as to be cached to the values buffer when they meet one of the following criteria. This is done per kernel part:
  - a variable is not used on the lhs strictly before it is used on the rhs in a kernel part strictly greater than the current part
  - a variable is an output variable and not use on the lhs in a strictly later part
  - a variable in the last kernel part is an output variable
  - a variable that is an argument or the result of conditional expectation operation

8.4 Elimination of replaced local variables from the values buffer

After the local variable replacement has been done (8.3), the following additional optimization is applied to the values buffer: If a variable is replaced by a stack variable in all kernel parts and is not marked as to be cached in any kernel part, it can be removed from the values buffer. The result of this step is expressed in bufferedLocalVarMap which maps an intermediate result id to the position in the values buffer. If no optimization is applied, this latter map is the identity.