Contents

1 Preface **2**

2 The Quasar-C/C++ guide **4**

2.1 Writing kernel/device functions in C/C++ . . . 4
2.1.1 A simple tutorial . . . 4
2.1.2 Device functions . . . 8
2.1.3 Module initialization/termination . . . 10
2.1.4 Using the Host API from device functions . . . 10
2.1.5 Exception handling . . . 11
2.1.6 Manual vs. automatic compilation . . . 12
2.2 Writing C++ libraries that can be used from Quasar . . . 13
2.2.1 Using C++ classes from Quasar . . . 14
2.2.2 Typed versus untyped classes . . . 17
2.2.3 Functions . . . 17
2.2.4 Abstract vs. non-abstract classes . . . 18
2.2.5 Variadic functions . . . 18
2.2.6 Properties versus fields . . . 19
2.2.7 Example: passing pointer values to Quasar with transferred ownership . . . 19
2.3 Using the Quasar C++ host API . . . 21
2.3.1 Host API interfaces . . . 23
2.3.2 16-bit unicode strings . . . 25
2.3.3 Profiling of C++ programs . . . 27
2.4 Quasar High-Level Interface/Domain Specific Language (DSL) . . . 28
2.4.1 Calling Quasar functions . . . 29
2.4.2 Automatic reference counting: QValue . . . 29
2.4.3 Exception handling . . . 29
2.4.4 Runtime control . . . 30
2.5 Project configuration . . . 31
2.5.1 Visual C++ guidelines . . . 31
2.5.2 CMake (windows / linux) . . . 31
2.5.3 Distributing C++ host applications . . . 31
2.6 Examples . . . 32
2.6.1 Example 1: color temperature . . . 32
2.6.2 Example 2: integrating OpenCV . . . 33
2.6.3 Example 3: integrating CUDA C/C++ with Quasar . . . 37
2.6.4 Example 4: integrating OpenCL with Quasar . . . 39
2.6.5 Example 5: user-type matrices . . . 41
2.6.6 Example 6: multi-GPU programming . . . 42
2.6.7 Example device configuration files . . . 44

3 Example: developing a camera interface to Quasar **46**

4 Writing a managed Quasar library **55**

4.1 Memory management (important) . . . 56
4.1.1 Memory leak detection tools . . . 57

4.1.2 Manual memory management (mandatory!) 57
4.2 Quasar Values *(QValue)* 57
4.3 Computation Engine interface *(IComputationEngine)* 59
4.4 Evaluation Stack interface *(IEvaluationStack)* 60
4.5 Matrix factory *(IMatrixFactory)* 61
4.6 Matrix allocation flags *(AllocationFlags)* 63
4.7 Generic matrix construction 63
4.8 Boundary access modes *(BoundaryAccessMode)* 64
4.9 Operators *(OperatorTypes)* 64
4.10 Class interface 65
4.10.1 Exporting .Net classes to Quasar 65
4.10.2 Importing Quasar classes from .Net 66
4.10.3 Method call interface 67
4.10.4 Documentation attributes 68
4.11 Callback events 69
4.11.1 Example 70

5 Example F# program **71**

Title: Quasar - External Interface Reference

Preface

This document contains information on the *external interface* of Quasar. The external interface is useful for (see figure below):

1. Developing custom libraries in another programming language (for example a C# or C++ module that can be imported with the import statement). For example, for implementing *kernel* and *device* function modules outside of Quasar, e.g. in C/C++/CUDA… This is the ideal choice if you want to use existing C/C++ libraries from Quasar. In essence, you write a wrapper function to convert the input/output arguments to the Quasar data types.

2. Calling Quasar programs from other .Net languages, such as C#, F# or IronPython

3. Calling Quasar programs from C++, integrating Quasar kernels with C++ or CUDA kernels.

Because the Quasar runtime currently runs on top of the Common Language Infrastructure (CLI), it is straightforward to interface with Quasar using the following programming languages: C#, C++/CLI, Boo, F#, J# and Visual Basic.Net. For a full list of languages, see CLI languages.

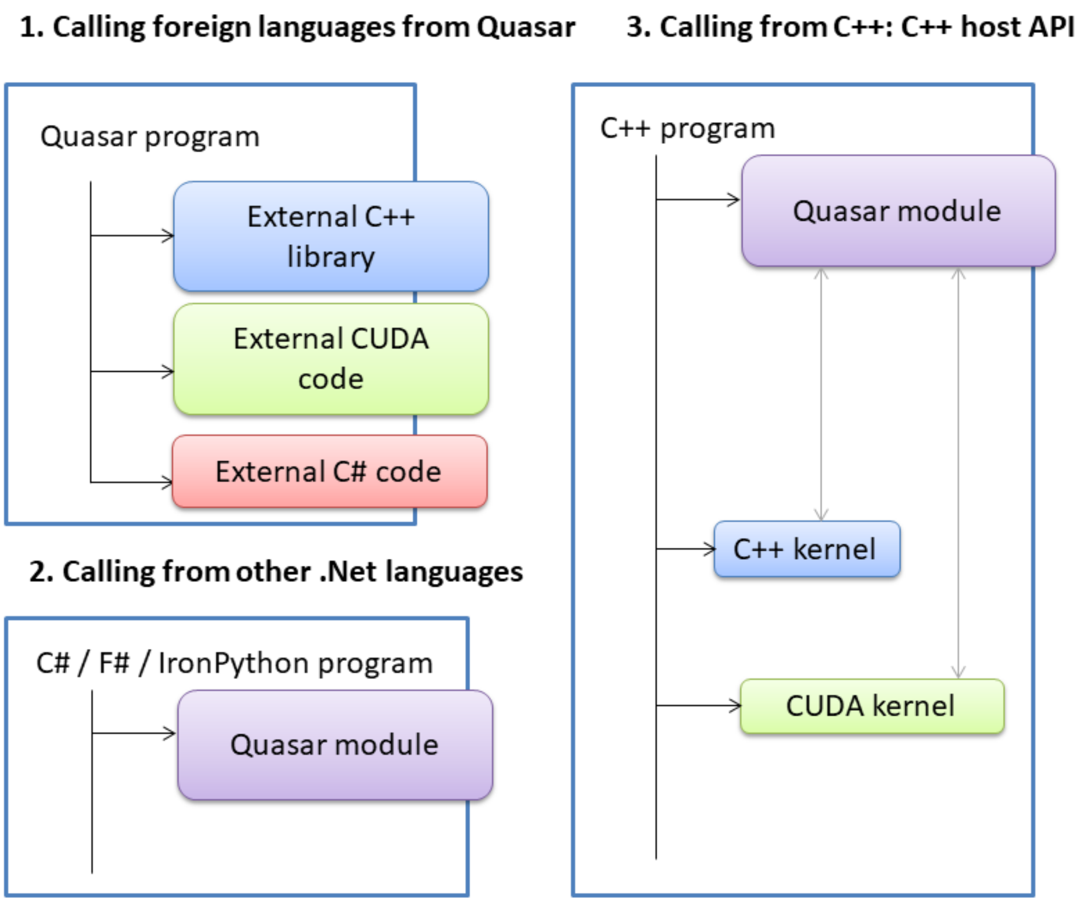
1. Calling foreign languages from Quasar
Quasar program
External C++ library
External CUDA code
External C# code
2. Calling from other .Net languages
C# / F# / IronPython program
Quasar module
3. Calling from C++: C++ host API
C++ program
Quasar module
C++ kernel
CUDA kernel

Figure 1: images

The Quasar-C/C++ guide

The Quasar C/C++ interface is used to integrate Quasar programs in existing applications, or to use existing C/C++ functions from Quasar. As such, the interface can be used in two ways.

- Running Quasar as a host and calling C/C++ functions from Quasar: see Writing kernel/device functions in C/C++.
- Running a C/C++ program as a host and accessing Quasar modules from C/C++: see Using the Quasar C++ host API.

The first approach allows existing C/C++ functions to be used from within Quasar, given that the function signatures are defined in a way that the Quasar compiler can recognize. In fact, the Quasar compiler can parse C/C++ modules and map functions that are named with the ENTRY-macro (such as ENTRY(some_function_name)) onto functions that can directly be called from Quasar. Both kernel and device functions can be defined from within C/C++. It is even possible to write CUDA kernels, e.g., in case the user wants to use advanced CUDA features that are currently not yet supported by Quasar. In all cases, the Quasar run-time generates the necessary stubs for translating between the (managed) Quasar level to the native C/C++ level.

Writing kernel/device functions in C/C++

In Quasar, it is relatively easy to operate with C/C++ functions. The mechanism to achieve this, just builds further upon the standard Quasar workflow:

- When writing .q modules, the kernel and device functions are extracted, and a C/C++ module is generated, which can subsequently be compiled using any C/C++ compiler (such as NVidia C/C++ compiler)
- Alternatively, it is possible (but not recommended when not necessary) to skip writing the Quasar module and to directly code the C/C++ module. This has the advantage that all possibilities of C/C++ become directly available to Quasar programs, while still enabling portable code (euhm… at the discretion of the programmer).

Summarizing, if you want to mess up things in an elegant way (C++ has a lot of caveats which are avoided/solved in Quasar), this is the ideal option for you. For example, you can obtain more control on the kernel function coding/optimization, or you can interact with existing C/C++ libraries without having to resort to managed C# programming.

A simple tutorial

Suppose we want to write a function fill_matrix to fill a matrix in C++. The Quasar main program can simply be as follows:

```
import "cpp_sample.cu"

function [] = main()

    x = uninit(10,10)
    parallel_do(1,x,fill_matrix)
    print x

endfunction
```

Note that we are directly importing a C++ (.cu) module. This code for this module is:

```
#include "quasar.h" // mandatory, to import Quasar data structures

using namespace Quasar;

// A CPU kernel function definition
extern "C" EXPORT void ENTRY(fill_matrix)(
    void* closure, // closure variables, reserved
    Matrix mtx, // the input argument being passed to the function
    int3 _gridDim, // the grid dimensions
    int3 _blockDim, // the block dimensions
    int _nThreads) // indication of the number of threads to use
{
    for (int m=0;m<mtx.dim1;m++)
        for (int n=0;n<mtx.dim2;n++)
        {
            mtx.data[m*mtx.dim2+n] = m+n;
        }
}
```

Each kernel function needs to be defined in a special way:

```
extern "C" EXPORT void ENTRY(function_name)(some_struct *data)
```

The Quasar import function performs some elementary parsing of the .cu file, which enables extracting kernel and device functions from the source file. The only argument to the function should always be a pointer to a struct that is declared *without* forward references. We pass the arguments in a struct so that the runtime can optionally cache the argument list, instead of regenerating it each time.

The argument list should always have the following order (if not, a compiler error is generated):

```
void* closure, // closure variables, reserved
... // user arguments
int3 _gridDim, // the grid dimensions
int3 _blockDim, // the block dimensions
int _nThreads // indication of the number of threads to use
```

The mapping from Quasar types to C/C++ types is as follows:

Quasar data type	C/C++ data type	Description
scalar	scalar	a scalar (floating-point number)
int	int	an integer (typically of 32-bit size)
cscalar	complex	a complex number
vec	Vector	a real-valued vector
mat	Matrix	a real-valued matrix
cube	Cube	a real-valued cube
cvec	CVector	a complex-valued vector
cmat	CMatrix	a complex-valued matrix
ccube	CCube	a complex-valued cube
ivecX	intX	a fixed-length (length X) integer vector

Quasar data type	C/C++ data type	Description
vecX	scalarX	a fixed-length (length X) real-valued vector
cvecX	complexX	a fixed-length (length X) complex-valued vector
vec[...]	VectorBase<...>	Composite vector types
mat[...]	MatrixBase<...>	Composite matrix types
cube[...]	CubeBase<...>	Composite cube types
string	char*	String data types

Some useful pre-processor defines:

Define	Meaning
TARGET_CUDA	Defined if are targetting CUDA (i.e. compiling with NVCC)
DBL_SCALAR	The type "scalar" is equivalent to "double" (otherwise "float")
__GNUG__	We are compiling with the GNU C++ compiler
_MSC_VER	We are compiling with the Microsoft (or Intel) C/C++ compiler
ENABLE_DYNAMIC _KERNEL_MEM	Dynamic kernel memory is enabled for this module (i.e. you can safely use the operators new and delete from CUDA kernel functions, making use of the built-in parallel memory allocation algorithm)

The Quasar compiler will automatically invoke the C/C++ compiler, in the same way as it does in case of a Quasar module. In this way, the .cu modules will run both under linux, windows and mac, and can considered to be portable (at least when no other exotic stuff is used).

Various functions have been defined for vector/matrix/cube operations. You can find them in the header file `quasar.h`, but below is a summary of some of the important functions:

Type	Function	Description
VectorBase<T>	vector_get_at<Mode>(A, pos)	Returns the value of a vector at the specified position
VectorBase<T>	vector_set_at<Mode>(A, pos, val)	Sets the value of a vector at the specified position
MatrixBase<T>	matrix_get_at<Mode>(A, pos)	Returns the value of a matrix at the specified position
MatrixBase<T>	matrix_set_at<Mode>(A, pos, val)	Sets the value of a matrix at the specified position
CubeBase<T>	cube_get_at<Mode>(A, pos)	Returns the value of a cube at the specified position
CubeBase<T>	cube_set_at<Mode>(A, pos, val)	Sets the value of a cube at the specified position
NCubeBase<T,N>	ncube_get_at<Mode>(A, pos)	Returns the value of a hypercube at the specified position
NCubeBase<T,N>	ncube_set_at<Mode>(A, pos, val)	Sets the value of a hypercube at the specified position

While setting/getting values from a vector/matrix/cube type, a boundary access mode can be specified. The following values are available:

Mode	Description
Default	Performs boundary checking; raises an exception when out-of-bounds
Unchecked	Performs no checking at all
Zero	Performs boundary extension with zeros
Const	Uses the read-only data cache load function (__ldg() in CUDA)

Note: since Quasar versions from 2018, circular (Circular), clamped (Clamp) and mirrored (Mirror) boundary extension are not available anymore in the external C++ interface. They can be enabled however by applying the functions periodize, clamp and mirror_ext to the index of the vector/matrix. This decision was made so that the Quasar compiler can apply boundary extension functions on an index basis, rather than applying them to all the indices (which may lead to an unnecessary performance cost).

Example: a 3x3 filter in CUDA Similarly, it is possible to provide an implementation for a CUDA kernel as a CPU kernel at the same time. We will illustrate this for a separable 3x3 averaging filter, which takes advantage of shared memory in CUDA and of OpenMP in CPU mode. We use the preprocessor definition TARGET_CUDA for finding out if we are running in CUDA mode or simple in CPU mode.

Main function:

```
import "cpp_filter3x3.cu"

function [] = main()

    x = imread("lena_big.tif")
    y = uninit(size(x))
    parallel_do(size(x),x,y,filter3x3)
    imshow(y,[])

endfunction
```

cpp_filter3x3.cu:

```
    #include "quasar.h"
    using namespace quasar;

    #ifdef TARGET_CUDA

        /* A 3x3 separable filter, using shared memory*/
        extern "C" __global__ void filter3x3(void* closure, Cube x, Cube y)
        {
            int3 pos = make_int3(blockIdx.y * blockDim.y + threadIdx.y,
                blockIdx.x * blockDim.x + threadIdx.x, blockIdx.z * blockDim.z + threadIdx.z);
            int3 blkpos = make_int3(threadIdx.y, threadIdx.x, threadIdx.z);
            int3 blkdim = make_int3(blockDim.y, blockDim.x, blockDim.z);

            /* Allocate temporary shared memory */
            shmem _shmem;
            shmem_init(&_shmem);
            Cube tmp = shmem_alloc<scalar>(&_shmem,blkdim.x+2,blkdim.y+2,blkdim.z);

            if (blkpos.x < 2)
```

```
                cube_set_at<Unchecked>(tmp, blkpos + make_int3(blkdim.x, 0, 0),
                    cube_get_at<Circular>(x, pos + make_int3(blkdim.x, 0, 0)));

            if (blkpos.y < 2)
                cube_set_at<Unchecked>(tmp, blkpos + make_int3(0, blkdim.y, 0),
                    cube_get_at<Circular>(x, pos + make_int3(0, blkdim.y, 0)));

            if (blkpos.x < 2 && blkpos.y < 2)
                cube_set_at<Unchecked>(tmp, blkpos + make_int3(blkdim.x, blkdim.y, 0),
                    cube_get_at<Circular>(x, pos + make_int3(blkdim.x, blkdim.y, 0)));

            cube_set_at<Unchecked>(tmp, blkpos, cube_get_at<Unchecked>(x, pos));

            cube_set_at<Unchecked>(y, pos, scalar(1.0/9) * (
                tmp.data[0] + tmp.data[1] + tmp.data[2] +
                tmp.data[3] + tmp.data[4] + tmp.data[5] +
                tmp.data[6] + tmp.data[7] + tmp.data[8]));
        }
#else

    /* A 3x3 separable filter - OpenMP-based implementation on the CPU */
    extern "C" EXPORT void ENTRY(filter3x3)(void* closure, Matrix x,
        Matrix y, int3 _gridDim, int3 _blockDim, int _nThreads)
    {
        omp_set_num_threads(_nThreads);
        #pragma omp parallel for
        for (int m=0; m<_gridDim.x * _blockDim.x; m++)
            for (int n=0; n<_gridDim.y * _blockDim.y; n++)
                for (int k=0; k<_gridDim.z * _blockDim.z; k++)
        {
            int3 pos = make_int3(m,n,k);
            cube_set_at<Unchecked>(y, pos, scalar(1.0/9) * (cube_get_at<Unchecked>(x, pos) +
                cube_get_at<Circular>(x, pos + [-1,-1,0]) +
                cube_get_at<Circular>(x, pos + [-1, 0,0]) +
                cube_get_at<Circular>(x, pos + [-1, 1,0]) +
                cube_get_at<Circular>(x, pos + [ 0,-1,0]) +
                cube_get_at<Circular>(x, pos + [ 0, 1,0]) +
                cube_get_at<Circular>(x, pos + [ 1,-1,0]) +
                cube_get_at<Circular>(x, pos + [ 1, 1,0]) +
                cube_get_at<Circular>(x, pos + [ 1, 1,0])));
        }
    }

#endif
```

Device functions

Device functions can be defined similarly, such as in the following example:

```
#include "quasar.h"
using namespace quasar;

extern "C" EXPORT __device__ void ENTRY(generate_matrix)(void *closure, int M, int N, Matrix Adata
    )
{
    for (int m=0;m<Adata.dim1;m++)
        for (int n=0;n<Adata.dim2;n++)
    {
        mtx.data[m*Adata.dim2+n] = m - n;
    }
}
```

It sufficies to use the modifiers extern "C" EXPORT __device__, then Quasar will recognize the function as being a device function. Note that the other of the keywords is of importance. This means that you can call the function directly from Quasar, without any extra efforts:

```
import "cpp_sample2.cpp"

function [] = main()
    A = generate_matrix(4, 4)
    print A
endfunction
```

This approach gives a lot of flexibility for interfacing C++ with Quasar.

Warning: due to compiler ABI differences (e.g., Visual C++ vs. G++), it is generally not safe to declare parameters of the QValue type. Using G++, we have found that the QValue parameter is passed as a pointer (QValue*) rather than by value, due to the presence of a destructor. In this case, we it is recommended either to pass a qvalue_t value (which does not have this problem), as follows:

```
extern "C" EXPORT __device__ void ENTRY(string_len)(void *closure, qvalue_t qstr, QValue* x)
{
    string_t wstr;
    host->GetString(qstr, wstr);
    int len = wstr.get_length();
    tprintf(_T("The length of the string is = %d"), len);
    *x = len;
}
```

It is also possible to take a qvalue_t value and to cast it to QValue, in this case, it is necessary to call QValue::Retain to avoid the object from being destroyed when the QValue goes out of scope:

```
extern "C" EXPORT __device__ void ENTRY(string_len)(void *closure, qvalue_t qstr, QValue* x)
{
    QValue qv = qstr; qv.Retain();
    string_t wstr;
    host->GetString(qt, wstr);
    *x = wstr.get_length();
}
```

Alternatively, ABI problems can be avoided completely by using the ADD_FUNCTION macro in combination with a native shared library (see lower).

Module initialization/termination

C++ modules can have entry/exit points. When a C++ library is loaded, Quasar calls the function module_init. Similarly, when a library is unloaded, the function module_term is called. In module_init, initialization operations can be perfomed. For example, it is useful to create a IQuasarHost instance, which gives access to all of the Quasar functionality (like creating vectors, matrices). In contrast to standalone binaries, in a C++ library, a Quasar host instance can be created using IQuasarHost ::Connect. This way, an IQuasarHost instance can be returned that is shared between several libraries.

```
static IQuasarHost *host; // Store host instance in static variable

extern "C" EXPORT __device__ void module_init(
    LPCVOID hostprivParams,
    LPCVOID moduleDetails,
    int flags)
{
    host = IQuasarHost::Connect(hostprivParams, moduleDetails, flags);
}

extern "C" EXPORT __device__ void module_term()
{
    if (host)
    {
        host->Release();
        host = NULL;
    }
}
```

The Quasar host instance is stored in a static variable. If the library is intended to be used in a multithreaded context, it may be declared as thread_local . The arguments provided to module_init should not be used by the library, instead they should be passed directly to IQuasarHost::Connect. The data stored at the pointer locations is implementation-specific and may change in a future version of Quasar.

In module_term, the host instance can then be released. In addition, cleanup operations can be performed.

Using the Host API from device functions

Device functions can make use of the Quasar Host API. This is mainly useful for allocating memory from within a kernel or device function that runs on the CPU. Also, it allows arbitrary Quasar objects to be passed to C++. To use the Quasar Host API, you need to include "quasar_dsl.h".

Memory allocation:

Quasar vectors and matrices can be allocated from device functions using the functions QValue::CreateVector<T>(), QValue::CreateMatrix<T>(), QValue::CreateCube<T>() and QValue::CreateNCube<T>(). The QValue data type supports automatic reference counting, therefore the allocated data will be destroyed automatically.

Remarks:

- When writing device functions that *return* vectors or matrices that are allocated inside the C++ function, you can not simply use malloc or the C++ operator new. This is because Quasar then would not know how to dispose the objects:

every library can have its own allocators and the Quasar runtime can impossibly know how to free the objects. Additionally, when returning output vectors or matrices, they need to be passed by a pointer to the parameter list (not as an explicit return value). This is because of application binary interface (ABI) reasons and portability across platforms.

- The garbage collector will not have any effect when the C++ kernel/device function is being called (the garbage collector is assuming that the allocated memory is still in use). When performing many allocations, it is adviced to reuse the allocated memory blocks (e.g. by putting them in a temporary pool).

Example:

The following example illustrates a device function that allocates a matrix, sets some values of this matrix and returns the resulting matrix:

```
#include "quasar.h"
#include "quasar_dsl.h"

// A function that uses a Quasar callback in order to allocate a matrix
extern "C" EXPORT __device__ void ENTRY(generate_matrix)(void *closure, int M, int N, QValue *
    retVal)
{
    QValue A = QValue::CreateMatrix<scalar>(M, N);
    Matrix Adata = host->LockMatrix<scalar>(A);
    for (int m=0;m<Adata.dim1;m++)
        for (int n=0;n<Adata.dim2;n++)
    {
        mtx.data[m*Adata.dim2+n] = m - n;
    }
    host->UnlockMatrix(A);

    *retVal = A;
}
```

Exception handling

When writing a device function in C++, exceptions of class quasar :: exception_t are thrown by the host API, allowing the handling of errors resulting from calling the host API. An example is the calling of the Quasar error function from C++, which will result in a managed exception to be translated to a C++ exception quasar :: exception_t.

A problem arises when these exceptions are not caught, and propagated back to the host API. This situation is handled differently in .Net and Mono:

- In .Net, the exceptions are caught as System.Runtime. InteropServices .SEHException. These exceptions are handled correctly by the Quasar runtime and the error handling works as exoected.

- In Mono, the exceptions are not caught and this results in a crash of the entire process. The solution is to implement exception handling in the device function:

```
#include "quasar.h"
#include "quasar_dsl.h"
```

```
extern "C" EXPORT __device__ void ENTRY(generate_matrix)(void *closure, int M, int N, QValue
    *retVal)
{
    try
    {
        QValue A = QValue::CreateMatrix<scalar>(M, N);
        Matrix Adata = host->LockMatrix<scalar>(A);
        for (int m=0;m<Adata.dim1;m++)
            for (int n=0;n<Adata.dim2;n++)
        {
            mtx.data[m*Adata.dim2+n] = m - n;
        }
        host->UnlockMatrix(A);

        *retVal = A;
    }
    catch (const quasar::runtime_t &ex)
    {
        IQuasarHost::GetInstance()->OnException(ex);
    }
}
```

By calling IQuasarHost::OnException(), the C++ exception will be passed to the Quasar runtime system.

Manual vs. automatic compilation

Automatic compilation By default, the Quasar compiler will attempt to compile the C++ script, using its standard script:

- cc_script.bat (Windows)
- cc_script.sh (Linux)
- cc_script.osx.sh (MAC OSX)

For module.cpp this will generate a shared library, with name module_HASH.BITNESS.so, module_HASH.BITNESS.dylib or module_HASH.BITNESS.dll depending on which operating system you use. Here:

- HASH is a hash-code that is used internally (mainly to avoid name collision when two modules with the same name reside in different directories)
- BITNESS is the number of bits of the architecture (32 for 32-bit, 64 for 64-bit).

For module.cu this will generate both a shared library, as a CUDA compiled library. This gives you the flexibility to write CUDA files manually. **Note**: it is strongly recommended not to do this except when, e.g., integrating existing C++/CUDA libraries or for making external functionality (e.g. camera interfaces) available to Quasar.

The Quasar compiler will make sure that the output libraries are placed in the Intermediate directory. This is to separate binary files from the source code files, so that users can easily clean up the binary files in one sweep.

Now, when the Quasar compiler invokes the C++ compiler, it will only pass the name of the source file it self, e.g.

```
g++ -x c++ module.cpp -fno-operator-names -O2 -shared -o module_ZZZ1.32.so
```

It may be desired to include other C++ source files in the build process or to link with other libraries. In this case, you will need to provide your own build script (for example, using makefiles). This can be achieved with manual compilation.

Manual compilation Here, you are supposed to write your own build scripts, either shell scripts or makefiles. Makefiles are preferred, but please make sure that the makefiles can be used on other operating systems as well (e.g. MSys in Windows). When you link with shared binaries, provide binaries for all operating systems when distributing the code.

Then let Quasar know that you choose to compile the C++ files manually, it suffices to select an appropriate output filename for the shared library. The output filename should have the following form:

```
module.${BITNESS}.${NATIVEEXT}
```

where ${BITNESS}=32 or 64 and ${NATIVEEXT}=.so, .dll, .dylib. This file must be placed in directory of the C++ source file. When Quasar detects that this file exists, it will copy the file to the intermediate directory Intermediate. Furthermore, Quasar will skip its automatic compilation step for this module.

Library loading and execution When the Quasar program starts, the compiled library will automatically be loaded into the memory of the computer. This is achieved using the standard OS functions (e.g., dlopen, LoadLibrary).

Writing C++ libraries that can be used from Quasar

As discussed in the previous sections, kernel and device functions can be implemented directly in C++. The source code module is then imported (import "mymodule.cpp") and is automatically compiled by the Quasar compiler. This mechanism follows essentially the same technique the Quasar compiler uses to generate C++ code for kernels defined in Quasar, with the difference that provided C++ code is manually written instead of autogenerated.

In some cases, it is not desirable that the C++ source code is provided together with the compiled library (which is required in the above approach). An alternative is to include a C++ file with only the definitions and an empty implementation (this is similar to how header files in C or C++ work). However, this process can be cumbersome, especially because multiple files need to be maintained. Also, often, the C++ library is part of the compilation toolchain (e.g., integrated with an existing C++ project).

To deal with this cases, Quasar offers the possibility to directly import *native* shared libraries. These libraries are imported as follows:

```
import "mylibrary.${BITNESS}.dll" % For a windows library
import "mylibrary.${BITNESS}.so" % For a linux library
import "mylibrary.${BITNESS}.dylib" % For a Mac OS/X library
import "mylibrary.${BITNESS}.${NATIVEEXT} % OS-independent
```

The BITNESS and NATIVEEXT parameters are filled in by the Quasar compiler. This way, Quasar code and native libraries can be written that works cross-platform. Depending on the target platform, the runtime will pick a suitable native library.

The compilation of the native library follows essentially the same techniques as in "Manual compilation" above. The only difference is that the Quasar compiler has no access to a C++ file containing the function definitions. Instead, it is requires that the library exports module_init and module_term functions which are called respectively when the library is loaded and unloaded.

In the module_init, functions can be registered by defining Quasar reductions from the C++ code. To simplify this process, the macro ADD_FUNCTION can be used.

Similarly, it is possible to define C++ classes that can be used from Quasar.

> As an example of a custom C++ build chain, see the CMake example Interop_Samples\Cpp_Quasar\lib_read_dataset. The CMake file can be used in different OS'es (e.g., linux, windows).

Using C++ classes from Quasar

Because C++17 currently lacks an reflection mechanism, it is necessary to register the class and its fields/methods/properties so that they can be used from Quasar. The approach we use is very similar to how C++ functions are exported in the scripting language Lua. We rely on the variadic template feature that is present in C++14. To take advantage of this feature, it is required to include the Quasar DSL class extensions, quasar_dsl_class_ext .h.

Below, an example is given of a natively defined class that can be used from Quasar.

```
#include "quasar_dsl.h"
#include "quasar_dsl_class_ext.h"

using namespace quasar;

static Type nativeClassType;

// Note: needs to implement IUnknown, so that is why we derive from IUnknown
class NativeClass : public ObjBase
{
public:
    int intField;
    QValue stringField;

public:
    NativeClass() : ObjBase((qvalue_t &)nativeClassType)
    {
        intField = 1;
        stringField = _T("Sample string field");
    }

public:
    void do_something()
    {
        tprintf(_T("The value of intField is %d!\n"), intField);
        tprintf(_T("The value of stringField is %ls!\n"), (LPCTSTR)stringField);
    }
    static NativeClass *from_qvalue(const qvalue_t &qv)
    {
        return static_cast<NativeClass *>(qv.private_obj);
    }

    static void register_type()
    {
        // Define the type so that it can be used from Quasar. Note that using this
        // mechanism, objects of the type are subject to automatic memory management
        // e.g. transfer between CPU and GPU etc. The objects can also be inspected
        // in the debugger, serialized to file, ...
        BEGIN_ABSTRACT_TYPE(builder, NativeClass, _T("native_class"), TYPEFLAGS_DYNAMIC_CLASS);
        ADD_FIELD(builder, intField);
        ADD_FIELD(builder, stringField);
        ADD_METHOD(builder, do_something);
        nativeClassType = builder.CreateType();
    }

    static void unregister_type()
    {
        nativeClassType = Type();
    }

    virtual ~NativeClass()
    {
        tprintf(_T("NativeClass destructor called\n"));
    }
};

// Needs to be defined within the Quasar namespace
namespace quasar
{
    DECLARE_STRUCT_TYPE(NativeClass); // Generates type information for our struct
    IMPLEMENT_TYPE(TYPE_UNTYPEDOBJECT, NativeClass, _T("native_class"))
}
```

Every class needs to implement the IUnknown interface (from Microsoft's Component Object Model); this can be achieved by inheriting simply from quasar :: ObjBase. Each class must then have a type instance (Type nativeClassType) associated to it. This type instance needs to be passed to the ObjBase constructor, so that Quasar functions can correctly determine the type of instances of NativeClass.

Important do not forget to set the type instance through the constructor: if not, Quasar cannot correctly determine the type of the object!

ObjBase provides several convenience functions for working with C++ classes. For example ObjBase :: wrap converts an allocated object (assuming dynamic allocation) to a QValue. Correspondingly, instances of NativeClass can be constructed as follows:

```
obj = new NativeClass;
QValue qvObj = obj->wrap();
```

The resulting QValue objects can then be passed as return values of functions, assigned to arrays etc. Functions may also accept QValue, then we need to determine the NativeClass object:

```
void do_something(QValue qvObj)
{
    NativeClass *obj = NativeClass::from_qvalue(qvObj);
    //obj->...
}
```

Finally, classes needs to be registered in order to be known to Quasar. This is achieved by using Quasar's reduction system. In quasar_dsl_class_ext .h several convenience macros have been defined for this purpose.

Macro	Purpose
BEGIN_TYPE	Starts the definition of a new class, with default constructor
BEGIN_ABSTRACT_TYPE	Starts the definition of a new class that cannot be instantiated from Quasar (e.g., with a private or protected constructor)
ADD_FIELD	Adds a field to the class
ADD_FIELD_RENAME	Adds a field to the class, but renames the field to the specific name. From Quasar, the renamed field will be visible
ADD_PROPERTY	Adds a property to a class, consisting of a get and a set function
ADD_PROPERTY_RENAME	Adds a property to the class and renames the property
ADD_OVERLOADED_METHOD	Adds an overloaded method to the class by explicitly identifying the signature of the overloaded method
ADD_VARIADIC_METHOD	Adds a variadic method to the class
ADD_VARIADIC_METHOD_RENAME	Adds a variadic method to the class, but renames the method
ADD_FUNCTION	Adds a global function, so that it can be called from Quasar
ADD_FUNCTION_RENAME	Adds a global function, but renames the function

The macros use template programming to determine Quasar mappings of the C++ types. As such, every class type involved must have a C++ mapping. C++ mappings can be defined using DECLARE_STRUCT_TYPE (to be placed in a header file) and IMPLEMENT_TYPE (to be placed in an implementation file). It is important that DECLARE_STRUCT_TYPE and IMPLEMENT_TYPE are placed in the global quasar namespace.

Correspondingly, type restrictions apply to the arguments of methods that can be called from Quasar. Below is a table with C++ types that have Quasar mappings defined.

C++ type	Quasar type
scalar	scalar
cscalar	cscalar
int	int
Vector	vec
Matrix	mat
Cube	cube
NCube<N>	cube{N}
CVector	cvec
CMatrix	cmat
CCube	ccube
NCCube<N>	ccube{N}
string_t	string
int8_t	int8
int16_t	int16

C++ type	Quasar type
int32_t	int32
int64_t	int64
uint8_t	uint8
uint16_t	uint16
uint32_t	uint32
uint64_t	uint64
qvalue_t	??
QValue	??
scalar1 , scalar2 ,...	vec(N)
cscalar1 , cscalar2 ,...	cvec(N)
int1 ,int2,...	ivec (N)

Returning multiple values can be achieved by using std :: tuple <... T> as return type of a function. Functions with pointer types and reference types as argument types can currently not be called from Quasar.

Typed versus untyped classes

When declaring and implementing types, it is necessary to inform Quasar which class type is being defined. In general, this will be dynamic class, however it is also possible to define nonwritable classes or mutable classes. For a discussion about the differences between these class types, see the Quick Reference Manual.

Quasar class type	Data type	Type flags
class	TYPE_TYPEDOBJECT	TYPEFLAGS_IMMUTABLE_CLASS
mutable class	TYPE_TYPEDOBJECT	TYPEFLAGS_MUTABLE_CLASS
dynamic class	TYPE_UNTYPEDOBJECT	TYPEFLAGS_DYNAMIC_CLASS

Currently it is not possible to define enumerations from C++. This functionality may be added in the future.

Functions

When directly importing a native shared library (e.g., using import "mylibrary.${BITNESS}.${NATIVEEXT}), functions need to be registered in order to be called from Quasar. For this purpose, the macros ADD_FUNCTION and ADD_FUNCTION_RENAME can be used. Both macros register the function in the reduction system of Quasar (i.e., compiler, runtime or both) so that the functions can be called from Quasar and so that their type can be inferred. In contrast to ADD_FUNCTION, ADD_FUNCTION_RENAME registers the function under a different name. This is useful, e.g., when different capitalization rules are used for the function names in C++ compared to Quasar.

Functions are declared slightly differently, compared to the static approach: * When the ADD_FUNCTION or ADD_FUNCTION_RENAME macros are used, it is no longer necessary to declare the functions as extern "C" EXPORT __device. * The closure parameter void* closure is no longer necessary and should be removed * Return values are used instead of pointer out parameters. In case of multiple return values, use std :: tuple <... T> as return type.

Example: dynamically registering functions The following example exports a string_len function to Quasar. The macro ADD_FUNCTION registers the function, so that it can be called from Quasar code.

```
QValue string_len(QValue qstr)
{
    string_t wstr;
    host->GetString(qstr, wstr);
    return wstr.get_length();
}

  static IQuasarHost *host; // Store host instance in static variable

  extern "C" EXPORT __device__ void module_init(
    LPCVOID hostprivParams,
    LPCVOID moduleDetails,
    int flags)
  {
    host = IQuasarHost::Connect(hostprivParams, moduleDetails, flags);

    ADD_FUNCTION(string_len);
  }

  extern "C" EXPORT __device__ void module_term()
  {
    if (host) { host->Release(); host = NULL; }
  }
```

Abstract vs. non-abstract classes

BEGIN_ABSTRACT_TYPE starts the definition of a class or type with no constructor defined. Such type can hence not be constructed from Quasar. This is useful when the object instances are all constructed from C++ and when it is not desired that the Quasar users create their own instances.

On the other hand, BEGIN_TYPE automatically registers the default constructor of the class. Object instances can then be created from Quasar using the function new.

Variadic functions

Variadic functions have a variable number of input arguments. When using the macro ADD_VARIADIC_METHOD, the C++ method/function should have the following signature:

```
    void my_function(const qvalue_t *args, int num_args);
```

At runtime, the function should check the number of arguments and the type of the arguments. For variadic functions with partially defined argument types, such as the function type [(cube, mat, ... vec[string]) −>()], there is currently no way to encode the type of the first arguments with known type into the C++ signature. Therefore, the types of these arguments must be checked at runtime.

Properties versus fields

When registering a field of a class, the offset of the field is passed to Quasar, which causes Quasar to write directly to the object memory when the value of a field is changed.

When it is desired to check to the value that is written, or when the value of the field is the result of a (light) computation, properties can be used. When declaring the property:

```
    ADD_PROPERTY(tb, zoom);
```

both the get and set method must exist:

```
    void set_zoom(scalar zoom);
    scalar get_zoom() const;
```

get-only properties are not supported. It is possible however to throw an exception from the set method (e.g., throw exception (_T("The field 'zoom' cannot be assigned to"));).

Example: passing pointer values to Quasar with transferred ownership

In some cases, it is useful to pass pointer values to Quasar. To achieve this in a safe way, we will define a NativePointer class, that is registered in Quasar. Wrapped in a qvalue_t instance, the NativePointer instances can be passed from C++ to Quasar (and vice versa). This also gives the advantage that the ownership of the native pointer can be controlled, even when only a Quasar function is holding a single reference to it. Example implementation code is given below.

```
#include "quasar_dsl.h"
#include "quasar_dsl_class_ext.h"
#include <Unknwn.h>

using namespace quasar;
static Type nativePointerType;

// Note: needs to implement IUnknown, so that is why we derive from IUnknown
class NativePointer : public ObjBase
{
    void *ptr;

public:
    NativePointer(void *ptr) :ObjBase((qvalue_t &)nativePointerType), ptr(ptr)
    {
    }
    virtual ~NativePointer() // Called when no references exist to the object
    {
    }

public:
    // Allow casting to a pointer value
    void * ptr_val() const { return ptr; }

    static NativePointer *from_qvalue(const qvalue_t &qv)
    {
        return static_cast<NativePointer *>(qv.private_obj);
    }

    static void register_type()
    {
        // Define the type so that it can be used from Quasar. Note that using this
        // mechanism, objects of the type are subject to automatic memory management
        // e.g. transfer between CPU and GPU etc. The objects can also be inspected
        // in the debugger, serialized to file, ...
        BEGIN_ABSTRACT_TYPE(builder, NativePointer, _T("native_pointer"), TYPEFLAGS_DYNAMIC_CLASS);
        nativePointerType = builder.CreateType();
    }
    static void unregister_type()
    {
        nativePointerType = Type();
    }
};

// Needs to be defined within the Quasar namespace - used for template-based automatic
// type translation from C++ to Quasar.
namespace quasar
{
    // DECLARE_STRUCT_TYPE belongs in the header (.h) file
    DECLARE_STRUCT_TYPE(NativePointer); // Generates type information for our struct

    // IMPLEMENT_TYPE needs to be put in the source (.cpp) file
    IMPLEMENT_TYPE(TYPE_UNTYPEDOBJECT, NativePointer, _T("native_pointer"))
}

int main(void)
{
    using quasar::ref;
    try
    {
        ref<IQuasarHost> host = IQuasarHost::Create(_T("cpu"));

        // We need to register the type before we can use it
        NativePointer::register_type();

        // Create a typed object.
        QValue obj = (new NativePointer((void*)0x10203040))->wrap();

        Function print(_T("print(...)"));
        Function type_(_T("type(...)"));

        // Check the type
        Type obj_type = type_(obj);

        if (host->TypeCompareToSpecification(obj_type, nativePointerType))
        {
            // Now we are sure about the type, we extract the pointer
            tprintf(_T("Pointer: %p\n"), NativePointer::from_qvalue(obj)->ptr_val());
        }
        else
        {
            tprintf(_T("Something went wrong and our object has the wrong type..."));
        }

        // Release the type before the "host" object goes out of scope
        NativePointer::unregister_type();
        return 0;
    }
    catch (exception_t ex)
    {
        tprintf(_T("An error occurred:\nSource: %s\nMessage: %s\nStack trace: %s"),
            ex.source.get_buf(),
            ex.message.get_buf(),
            ex.stack_trace.get_buf());
        return 1;
    }
}
```

Although this functionality is quite specialized, it allows C++ libraries to be defined with classes that can be used from Quasar.

Using the Quasar C++ host API

> Prerequisites: - when using the Quasar C++ host API from linux, the following development packages need to be installed: libmono−2.0−dev and libglib2 .0−dev. See the folder Interop_Samples/CppAPI for a CMake file. - for Visual Studio in Windows, please make sure that the .Net development module is installed with Visual Studio. In particular, header file metahost.h is required.

In this section, we explain how a Quasar program can be accessed from within C++/CUDA code. This allows you to develop a native C++ console/GUI application, that loads and interacts with Quasar dynamically, at run-time. This approach is an easy way to integrate GPU processing in your C++/CUDA application. The resulting applications can work in CPU mode (OpenMP), with CUDA and even with OpenCL. This way, it is also not necessary to provide different code paths for your algorithms.

Two programming interfaces exist for accessing Quasar:

1. C-like low-level interface via quasar_host.h. This interface leaves memory management (adding and releasing references) to the programmer.
2. C++-style high-level interface via quasar_dsl.h. This interface provides automatic memory management through smart pointers. As in many libraries, the C++ high level interface is a wrapper built around the C interface.

In the following, we will first discuss the low-level interface (quasar_host.h). The communication with Quasar is done using the IQuasarHost interface, as illustrated in the following example:

```
#include "quasar_host.h"
#include "quasar_dsl.h"

int main(void)
{
    // Creates the quasar host
    LPCTSTR deviceName = _T("cuda");
    ref<IQuasarHost> host = IQuasarHost::Create(deviceName, false);

    // Problem loading Quasar
    if (host == NULL)
    {
        tprintf(_T("Could not create a Quasar host instance!"));
        return -1;
    }

    return 0;
}
```

An instance of the Quasar host object is created using IQuasarHost::Create, which takes a device name (note that we use generic string types for cross-platform compatibility). The device name can be cpu (to specify the CPU computation device), cuda (to specify an arbitray CUDA device) and auto (to automatically select a device). In the automatic mode, preference is first given to CUDA, then OpenCL and if both are not supported, a CPU computation device is used. Alternatively, it is possible to pass the file name of a Quasar Hyperion device configuration XML file (which permits multi-device configurations). See example device XML configurations for more info on these device configurations. For example:

```
LPCTSTR deviceName = _T("cuda_dualgpu.device.xml");
bool loadCompiler = false;
ref<IQuasarHost> host = IQuasarHost::Create(deviceName, loadCompiler);
```

Also important is the second parameter passed to IQuasarHost::Create, which indicates whether the Quasar compiler needs to be loaded. Note that including the Quasar compiler requires a Quasar license! When possible, it is preferred to only load the Quasar runtime (by specifying false). The benefits of the two approaches are listed below:

loadCompiler = false:

- .q files cannot be loaded at run-time. No run-time compilation possible. Instead, .q files need to be compiled to .qlib files (either using Quasar Redshift, or by invoking the Quasar compiler by the command-line).
- .qlib files are often significantly faster in execution, and are completely pre-compiled.
- In this mode, the run-time environment starts up slightly faster.

loadCompiler = true: * Both .q and .qlib-files can be loaded. * Run-time compilation is possible (see function LoadModuleFromSource). * Most flexibility * Slightly slower loading of the run-time environment * Quasar license required

Quasar modules can be loaded using the functions LoadSourceModule (.q files) and LoadBinaryModule (.qlib files).

```
// Loading a .q source module
LPCTSTR errorMsg;
bool success = host->LoadSourceModule(_T("color_temperature.q"), &errorMsg);

// Loading a .qlib quasar library
LPCTSTR errorMsg;
bool success = LoadBinaryModule(_T("color_temperature.qlib"), &errorMsg);
```

The compilation of a .q file to a .qlib can easily be done as follows:

```
mono Quasar.exe -make_lib color_temperature.q
```

During the compilation process, the compiler will collect all the modules that depend on color_temperature.q and bundle all resulting binary modules in the .qlib file. Note that, in case .dll-files or other .qlib-files are used from within the Quasar program, these files also need to be distributed when deploying the application to the end user. It often suffices to put all modules within the same folder (e.g. a bin-dir).

Host API interfaces

An overview of the main interfaces in the C++ host API is given below:

Quasar interfaces

Description

IQuasarHost

Handles the main communication with the Quasar host

IEvaluationStack

Represents an abstracted evaluation stack for performing arithmetic operations.

The evaluation stack is implemented by the underlying computation engine

IComputationEngine

Gives access to functionality of a Quasar computation engine

ITypeEnvironment

Enables retrieving platform-specific type information

IMatrixFactory

Helper interface for constructing vectors/matrices/cubes of various types

IRuntimeReductionEngine

Allows to dynamically define/undefine reductions at run-time

IQuasarHost interface The IQuasarHost interface handles the main communication with the Quasar host. There can only exist one IQuasarHost object at the time. Constructing IQuasarHost instances can be computationally costly, because it involves loading a lot of library depencies (e.g., CUDA, OpenCL). Therefore the IQuasarHost instance should be kept alive as long as needed. The following functions are exposed via the IQuasarHost interface:

IQuasarHost interface	Description
LoadSourceModule	Loads a Quasar source module (.q file)
LoadBinaryModule	Loads a Quasar binary module (.qlib file)
LoadModuleFromSource	Loads a Quasar module from a source string
FunctionExists	Checks whether a function with the specified name exists within the Quasar host environment.
FunctionCall	Calls the specified Quasar function
CreateVector	Creates a vector with the specified dimensions
CreateMatrix	Creates a matrix with the specified dimensions

IQuasarHost interface	Description
CreateCube	Creates a cube with the specified dimensions
CreateTypedObject	Creates a used-defined typed object instance
CreateUntypedObject	Creates a used-defined untyped object instance
CreateString	Creates a Quasar string instance
CreateLambda	Creates a Quasar lambda expression (function) for the specified callback function
AddRef	Increases the reference count for a given Quasar value (qvalue_t)
ReleaseRef	Decreases the reference count for a given Quasar value (qvalue_t). When reached zero, the object is deleted.
DeleteValue	Deletes the specified Quasar value. For refcountable objects, this function calls ReleaseRef. For other objects (e.g. strings), the value is directly deleted.
Lock	Maps the specified Quasar object (e.g., a matrix) onto system memory so that it can be directly written. Locks the object such that no other (asynchronous) functions can modify the data.
Unlock	Unmaps the specified Quasar object. This function needs always to be called in combination with Lock
GetPrimitiveTypeHandle	Obtains a Quasar type handle for the specified primitive data type
RunApp	Runs the loaded Quasar application and waits until all windows (e.g. imshow(), plot()) are closed. May return immediately in case no windows are created.
DoEvents	Handles all queued windowing events (such as redrawing events, mouse click events). Calling this function now and then may make the application more responsive.
ReadVariable	Reads the value of a Quasar host variable
WriteVariable	Replaces or sets the value of a Quasar host variable
GetField	For object instances, gets the value of the field with the specified name
SetField	For object instances, sets the value of the field with the specified name
LookupFunction	Looks up a function based on the specified function signature and returns a Quasar value (qvalue_t) for this function
LookupType	Looks up a type by name and returns a Quasar value (qvalue_t) for this type
LookupMethod	Looks up a method based on the specified method signature and returns a Quasar value (qvalue_t) for this method
MethodCall	Calls a method on an object with the specified name and set of arguments
GetType	Returns the type of the specified Quasar value (qvalue_t)
CreateType	Creates a new user-defined object type (either a typed or an untyped object type).
AddField	Adds a new field to an object type created using CreateType.
AddParameter	Adds a new generic parameter to an object type created using CreateType.
FinalizeType	Finalizes the type definition, performing data layout. Function to be called in combination with CreateType. After calling, no modifications are allowed anymore to the type definition.
EnableProfiling	Activates the Quasar profiler, writing profile data to the specified output file. Useful for performance analysis.
CreateStack	Creates an evaluation stack object.
GetComputationEngine	Gets a computation engine object, for direct access to the computation engine.

When a Quasar module is loaded (LoadSourceModule or LoadBinaryModule), the global definitions of the module are stored in the Quasar host environment. These definitions can then be accessed via the methods ReadVariable and WriteVariable. It is possible to call Quasar functions in a module, using FunctionCall.

IEvaluationStack interface Represents an abstracted evaluation stack for performing arithmetic operations. The evaluation stack is implemented by the underlying computation engine.

IMatrixFactory interface Helper interface for constructing vectors/matrices/cubes of various types.

IMatrixFactory interface

Description

New

Constructs a new matrix initialized with the specified array in system memory

IRuntimeReductionEngine interface Allows to dynamically define/undefine reductions at run-time.

IRuntimeReductionEngine interface

Description

Add

Adds an expression and corresponding handler to the runtime reduction engine

Remove

Removes an expression from the runtime reduction engine

ITypeEnvironment interface Enables retrieving platform-specific type information.

ITypeEnvironment interface

Description

GetScalarType

Retrieves the current scalar type (single precision float/double precision float)

IComputationEngine interface Gives access to functionality of a Quasar computation engine.

IComputationEngine interface	Description
GetName	Returns the name of the current computation engine
GetEvaluationStack	Gets the evaluation stack instance associated to this computation engine
GetMatrixFactory	Gets the matrix factory instance associated to this computation engine
GetTypeEnvironment	Gets the type environment instance associated to this computation engine
GetRuntimeReductionEngine	Gets the runtime reduction instance associated to this computation engine
Process	Performs the specified arithmetic computation to one or two elements at the top of the evaluation stack
ConstructMatrix	Constructs a vector or matrix based on the values pushed to the evaluation stack
ConstructCellMatrix	Constructs a cell vector or cell matrix based on the values pushed to the evaluation stack
FunctionCall	Calls a function with the specified name on arguments pushed to the evaluation stack
ArrayGetAt	Reads a value from a vector/matrix/cube based on indices pushed to the evaluation stack
ArraySetAt	Writes a value to a vector/matrix/cube based on indices pushed to the evaluation stack
Synchronize	Synchronizes all computation devices with the host thread

16-bit unicode strings

For interoperability with .Net, the Quasar C++ host API uses `wchar_t` wide characters. This is to allow easy representation of, e.g., Chinese characters, something that is not possible with the ASCII/ANSI character sets alone. For platform-dependent reasons (see below), the host API relies on the basic type TCHAR which represents a wchar_t unicode character.

Platform-specific handling of unicode strings In Windows, the size of the wchar_t type is 2 bytes (representing UCS-2). In Linux, wchar_t is by default 4 bytes. The compiler flag −fshort−wchar (which causes GCC to treat wchar_t as a 16-bit character string), does not help, because no C standard library functions are available for 16-bit wchar_ts (called char16_t). To simplify the programming, both in Windows and Linux the serialize defaultwchar_t data type is used. Mono in linux uses again 16-bit Unicode character strings. For this reason, in Linux, the Quasar host implementation will transparently convert 16-bit Unicode strings to 32-bit wide character strings and back. In Windows, there is no conversion required.

In char16_string .h, the TCHAR and corresponding pointer LPTSTR and const pointer version LPCTSTR are defined in C++ as follows:

```
    #include <wchar.h>
    typedef const wchar_t *LPCTSTR;
    typedef wchar_t *LPTSTR;
    typedef wchar_t TCHAR;
```

These conventions follow the Microsoft generic text mappings, a technique that allows applications to easily be retargetted for ANSI, UNICODE and multi-byte character representations. For this purpose, it is necessary to define all constant string symbols using the _T() function, such as in the following example:

```
LPCTSTR my_text = _T("This is a sample text");
```

This way, the C/C++ compiler will correctly store the string in the correct character format.

In Visual C++, it is necessary to set the default character set to UNICODE, in order for the preprocessor symbols _UNICODE and _MBCS to be defined.

For storing wide character strings, the class quasar :: string_t can be used.

The following functions are available for string manipulation (use functions from the second column):

C library function	Generic version	Purpose
strncpy	tcsncpy	Copies n characters from one buffer to another buffer
strchr	tcschr	Looks for the occurrence of the specified character in the string
strnlen	tcsnlen	Returns the length of a string with maximum size n
strlen	tcslen	Returns the length of a string
vsprintf	vstprintf	Formats a string using a C-style variadic argument list, writing the result to a buffer
sprintf	stprintf	Formats a string and write the result to a buffer
printf	tprintf	Formats a string and prints it to the standard output stdout

Many of these functions are also conveniently wrapped using the quasar :: string_t class, which has the following definition:

```
class string_t
{
private:
    int length;
    const TCHAR *chars;
public:
    string_t():length(0),chars(NULL) {}
    string_t(const string_t &src) { set_string(src.chars, src.length); }
    string_t(const TCHAR *chars) { set_string(chars, tcslen(chars)); }
    string_t(const TCHAR *chars, int length) { set_string(chars, length); }
    string_t &operator=(const string_t &src);
    static string_t unmarshal(const TCHAR *chars, int length);
    static string_t format(const TCHAR *fmt, ...);
    virtual ~string_t();
    operator const TCHAR *() const { return chars; }
    LPCTSTR get_buf() const { return chars; }
    int get_length() const { return length; }
    LPTSTR find_char(TCHAR chr) const { return tcschr(chars, chr); }
private:
    void set_string(const TCHAR *chars, int length);
};
```

Currently, there are no functions provided for converting between TCHAR and 8-bit char. In the future, such functions may be added. Instead, the C library functions mbtowc and wctomb can be used for this purpose.

Profiling of C++ programs

The profiling host API has the purpose of finding bottlenecks in C++ host code that dynamically calls Quasar functions. For example, the profiler can track the exeuction time of kernel functions, detect memory leaks also resulting from the reference counting scheme in C++, etc.

Profiling information of C++ programs using the Quasar host interface can be recorded using the EnableProfiling host API. It is sufficient to call this function after the IQuasarHost object has been created. The EnableProfiling function accepts two parameters: the profiling mode and an output filename (which is optional depending on the profiling mode, see notes below).

```
bool IQuasarHost::EnableProfiling(ProfilingModes profilingMode,
                                  LPCTSTR outputFileName = NULL);
```

The function returns TRUE when a profiler was attached and FALSE otherwise.

The following profiling modes are defined:

```
enum ProfilingModes
{
   PROFILE_EXECUTIONTIME = 2, // Profiles the execution time of functions (both CPU/GPU)
   PROFILE_MEMLEAKS = 3, // Detects memory leaks and gives more information on these leaks
   PROFILE_ACCURACY = 4, // Profiles the accuracy of the operations
};
```

The parameter outputFileName specifies an optional output file name parameter (.qprof extension for PROFILE_EXECUTIONTIME and .qacc for PROFILE_ACCURACY). These files can be opened in Quasar Redshift for inspection.

Notes: * The profiler is only available when the loadCompiler parameter of IQuasarHost::Create is set to true. The reason is that the profiler is intended to be used as a developer tool (e.g., to identify bottlenecks or memory leaks in the code). If loadCompiler=false, the function consistently returns false . * For PROFILE_ACCURACY, the parameter outputFileName is mandatory. If a NULL value is passed, an exception will be thrown. * For PROFILE_EXECUTIONTIME, the parameter outputFileName is optional. If a NULL value is passed, profiling information is printed to the console. * For PROFILE_MEMLEAKS , the parameter outputFileName is ignored. All memory leak information is printed to the console.

Quasar High-Level Interface/Domain Specific Language (DSL)

The Quasar DSL provides various high-level functions and classes that simplify the interaction with the Quasar interfaces. Because of the automatic reference counting it is recommended to use the DSL as much as possible from user code.

In the DSL, the central data type is QValue. An example is given below:

```
scalar vals[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
QValue B = QValue::CreateMatrix<scalar>(4, 4, vals);
print(B(0,0));
```

The above code fragment first constructs a 4x4 matrix based on the specified array of scalar values stored in system memory. Then, the matrix element at position (0,0) is printed. For each QValue, Quasar keeps internal data structures holding CPU and GPU pointers. Quasar automatically transfers the data in between CPU and GPU whenever necessary. Using the function host−>Lock() it is possible to get access to the raw data pointer for a specific device. For example, to get a CUDA device pointer (CUdeviceptr), you can call:

```
CUdeviceptr devicePtr;
LockResults result = host->Lock(im, TYPE_SCALAR, LOCK_READ, MEMRESOURCE_SINGLE_CUDA,
    (void**) &devicePtr);
...
host->Unlock(im, LOCK_READ, MEMRESOURCE_SINGLE_CUDA);
```

Then you can use the obtained device pointer from CUDA code (CUDA kernels, run-time and driver API functions). It is required to call the function Unlock when the data is no longer be needed. This way, Quasar knows when a specified object is in use by user C++ code. The following memory resources can currently be locked:

- MEMRESOURCE_MANAGED: will return a managed memory object handle
- MEMRESOURCE_CPU: will return a pointer (void*) that can be used on the CPU
- MEMRESOURCE_SINGLE_CUDA: will return a device pointer that can be used from CUDA. Can only be used for single device configurations.
- MEMRESOURCE_SINGLE_OPENCL: will return a device pointer that can be used from OpenCL. Can only be used for single device configurations.

Note that in order to lock these memory resources, it is required that the host is opened with the corresponding device enabled. For example, to use MEMRESOURCE_SINGLE_OPENCL, a Hyperion device configuration file for OpenCL needs to be used. See example device XML configurations for more info.

Calling Quasar functions

Quasar functions can be called directly using the Function class. Therefore, the functions need to be declared *after* the IQuasarHost object has been constructed, for example as follows:

```
Function ln(_T("log(??)"));
Function print(_T("print(...)"));
```

Note that the function signature needs to be specified. This is for correct function binding in case the specified function is overloaded. Functions with variadic arguments can specify ... . In between the parentheses, a comma-separated list of the argument types need to be specified. In case any type is allowed, the double question mark ?? should be used (in C++ this needs to be escaped as ?\?):

```
Function linspace(_T("linspace(?\?,?\?,?\?)"))
```

The CoreLib class contains some predefined Quasar functions. The functions can then be called using the normal parentheses. The return value is always QValue:

```
CoreLib coreLib;
QValue A = coreLib.linspace(0, 10, 100);
```

Automatic reference counting: QValue

Automatic reference counting for Quasar values (qvalue_t) can be obtained by using the QValue class defined in quasar_dsl.h. The class hence acts as a smart pointer. Various functionality is supported for the QValue class. For example:

- Assignment of scalar values QValue a = 0.5f;
- Assignment of string values QValue b = _T("Hello");
- Assignment of matrices QValue c = QValue::CreateMatrix<scalar>(4, 4);
- Assignment of fixed length arrays scalar array[3]; QValue d = array;
- Assignment of lambda expressions QValue e = QLambda::CreateLambda(QValue::LookupType(_T("[cube[uint8]−>mat]")), lambda);

It is important to realize that in many cases, QValue keeps a pointer to an object in managed memory. This gives some flexibility in the implementation of the Quasar run-time system, for example, the ability to support multiple computation engines, multiple devices etc., without requirign that the C++ interface needs to be adjusted.

Exception handling

Almost all functions of the Quasar DSL automatically generate C++ exceptions when needed. The exception is of the type exception_t, which is defined as follows:

```
class exception_t
{
public:
    string_t source; // The source module that generated the exception
    string_t message; // The error message
    string_t stack_trace; // The internal stack trace of the error
};
```

These exceptions can then be caught with the usual try {} catch {} pattern. Alternatively, the user can also specify a custom exception handler, as follows:

```
bool custom_exception_handler(const quasar::exception_t & ex)
{
    tprintf(_T("Custom exception handler:\nsource: %s\nmessage: %s\nstack trace: %s\n"),
        (LPCTSTR) ex.source,
        (LPCTSTR) ex.message,
        (LPCTSTR) ex.stack_trace);
    return true; // If true: continue execution, if false: throw ‘ex‘.
}

host->SetCustomExceptionHandler(custom_exception_handler);
```

The function custom_exception_handler is then called before the exception is thrown. If the function returns true, the program will continue executing. The custom exception handler can be used for logging exceptions, displaying message boxes to the user, crash dialog boxes etc. It is also possible to unset the custom exception handler, by calling

```
host->SetCustomExceptionHandler(NULL);
```

This way, exception handlers can be changed throughout the code.

Runtime control

Using the quasar :: RuntimeControl class, the runtime system can be tuned manually, allowing the device for memory allocations to be specified for each allocation, allowing manual memory transfer between devices and overriding the automatic scheduler. The following functions are available:

RuntimeControl class	Description
Alloc	The next memory allocation will be performed on the specified device (e.g. "cpu", "gpu", "auto").
Transfer	Transfers a variable to the specified device.
Schedule	Sets the scheduling mode for the next kernel function call (e.g. "cpu", "gpu", "auto").
ScheduleGPU	The next kernel function launch will be scheduled on the specified GPU
ScheduleCPU	The next kernel function launch will be scheduled on the CPU
ScheduleAuto	Switch back to the auto scheduling mode (default).

Project configuration

Quasar .qlib libraries can be built in 32-bit floating point precision mode (float) or in 64-bit floating point precision mode (double. However, the bitness of the C++ application needs to match the .qlib libraries. In case you intend to use double, you need to compile the module using the −DDBL_SCALAR flag.

Visual C++ guidelines

Supported versions of the Visual C++ compiler are: Visual C++ 2010, Visual C++ 2012, Visual C++ 2013, Visual C++ 2015 and Visual C++ 2017. Make sure that the installed Visual C++ version also has the right Windows SDK version installed (for example Windows SDK 7.1 for Visual C++ 2010, "Universal CRT" for Visual C++ 2017).

Because former Express Editions (e.g., Visual C++ 2010) do not come with OpenMP support, it is neccessary to disable OpenMP in the Project Settings / Compilation Settings of Visual C++. For Visual Studio community editions, OpenMP support is included.

Important: the default character set needs to be set to Unicode (see Unicode Strings in Quasar).

Because older versions of Visual C++ do not support C++11 and C++14, these features have been disabled when you compile with these older versions. This is mostly relevant when including quasar_dsl.h, which provides lambda expression and variadic template pack wrapping functions for Quasar.

You can directly add the Quasar source and include files (quasar.h, quasar_host.h, quasar_host.cpp, quasar_dsl.h, quasar_dsl.cpp, char16_string.h, char16_string.c etc.) to your project, depending on which files you intend to use. You can find these files in de include folder of the Quasar installation.

Also, you need to make sure that the bitness of your application matches the bitness of the Quasar installation. If you installed the 64-bit version of Quasar, you need to compile Quasar for the x64-platform. On the other hand, you installed the 32-bit version of Quasar, you need to compile Quasar for x86.

Finally, note that Quasar is located using the QUASAR_PATH environment variable. In case you installed both the 32-bit and 64-bit version of Quasar, you need to make sure that QUASAR_PATH points to the right version that matches the bitness of your application.

CMake (windows / linux)

Various C++ host API samples are available in the Interop_Samples/Cpp_API folder of the Quasar installation. The included CMake file can be used for compiling the samples.

Distributing C++ host applications

Distribution of C++ Quasar host applications also requires distributing the Quasar runtime. We are currently preparing an installer for the Quasar runtime system, which comes without the Quasar Compiler and Redshift. Contact us (`info@gepura.io`) to obtain more information about when the Quasar runtime installer will be released.

Examples

In the following, we give a few examples of the Quasar C++ host API. The original source files can be found in the Interop_Samples/Cpp_API folder of the Quasar installation.

Example 1: color temperature

Some samples can be found in the Interop_Samples/Cpp_API folder of the Quasar installation. Here we give a simple example, to show how a Quasar function can be called from Quasar.

First, the file color_temperature.q implements a simple color temperature filter on an image, making the image appear colder or warmer, depending on the temperature parameter.

color_temperature.q

```
function [] = __kernel__ color_temperature(x : cube, y : cube, temp : scalar,
    cold : vec3, hot : vec3, pos : vec2)
    input = x[pos[0],pos[1],0..2]
    if temp<0
        output = lerp(input,cold,(-0.25)*temp)
    else
        output = lerp(input,hot,0.25*temp)
    endif
    y[pos[0],pos[1],0..2] = output
end

function [img_out] = apply(img_in, temp)
    hot = [1,0.2,0]*255
    cold = [0.3,0.4,1]*255
    img_out = zeros(size(img_in))
    parallel_do(size(img_out,0..1),img_in,img_out,temp,cold,hot,color_temperature);
end
```

Our goal is to call the function color_temperature from C++. This can either be achieved by invoking the function apply, or by calling parallel_do and passing color_temperature as a parameter. For demonstrational purposes, we use the second approach.

sample.cpp

```
#include "quasar_dsl.h"

void sample(void)
{
    using namespace quasar;

    // Creates the quasar host and loads the specified module
    ref<IQuasarHost> host = IQuasarHost::Create(_T("cuda"));

    // Problem loading Quasar
    if (host == NULL)
        return;

    LPCTSTR errorMsg;
    if (!host->LoadSourceModule(_T("color_temperature.q"), &errorMsg))
    {
        tprintf(_T("%s\n"), errorMsg);
         return;
    }

    Function imread(_T("imread(string)"));
    Function parallel_do(_T("parallel_do()"));
    Function imshow(_T("imshow(cube)"));

    // Checks whether the function 'apply' is available
    tprintf(_T("The function 'apply' exists: %d\n", host->FunctionExists(_T("apply"))));

    // Load an image and show it
    QValue img = imread(_T("lena_big.tif"));
    tprintf(_T("The image has dimensions %dx%dx%d\n"),
        size(img,0), size(img,1), size(img,2));

    // Call a kernel function
    QValue kernelFunc = QValue::ReadHostVariable(_T("color_temperature"));

    // x : cube, y : cube, temp : scalar, cold : vec3, hot : vec3, pos : vec2
    // hot = [1,0.2,0]*255;
    // cold = [0.3,0.4,1]*255
    scalar sz[2] = { size(img,0), size(img,1) };
    scalar hot[3] = {1 * 255, 0.2 * 255, 0 };
    scalar cold[3] = {0.3 * 255, 0.4 * 255, 255 };
    parallel_do(sz, img, img, 0.5, cold, hot, kernelFunc);

    imshow(img);

    // Wait until all windows are closed
    host->RunApp();
}
```

First, an IQuasarHost object is created. Next, this host object is used to load the source module color_temperature.q. Through the Function class, Quasar functions can easily be accessed. For function binding, it is required to pass the parameter types to the constructor of the Function class. Then, an input image is loaded and the kernel function is accessed through QValue ::ReadHostVariable. With a simple parallel_do call, the kernel function is launched. The sample program ends with calling host−>RunApp(), which is required to ensure that the program terminates as soon all display windows (see imshow(img)) have been closed.

Example 2: integrating OpenCV

This example demonstrates how OpenCV can be integrated in Quasar applications, so that OpenCV functions can be called from Quasar. We consider the face detection problem, for which the necessary functions exist in OpenCV.

For our purpose, we define a function facedetection : [cube[uint8] −> cube] in Quasar. This function will later be implemented in C++. The Quasar program opens a webcam and shows the webcam input stream in a display window. At the same time, the frame buffer is passed to the C++ program via the function facedetection (callback mechanism). The face detection returns rectangles corresponding to the detected faces. These rectangles are then drawn by the Quasar program.

Note that the callback mechanism is a more advanced way for building a bridge between Quasar and C++. A simpler way is by wrapping each of the individual OpenCV functions and importing the C++ module from Quasar (e.g. import " face_detection

.q"). But here, the callback approach has the advantage that the implementation of facedetection can completely be changed dynamically at run-time by the C++ program.

face_detection.q:

```
import "Quasar.Video.dll"
import "Quasar.UI.dll"
import "inttypes.q"

% Function will be externally defined in the C++ program. However
% we need to declare it here so that the Quasar compiler knows
% about it.
facedetection : [cube[uint8] -> cube]

function [] = run()
    cams = vidcamlist()
    if numel(cams)==0
        error "Unfortunately, we could not find a webcam in your system."
    endif
    % select the first webcam
    cam = cams[0]

    % Opens the specified video file for playing
    stream = vidopen(sprintf("cam:video=""%s"",video_size=640x480,frame_rate=30",cam))
    vidstate = object()
    [vidstate.is_playing, vidstate.allow_seeking, vidstate.show_next_frame] = [true, true, true]
    print "Video stream information: ",stream

    frm = form(sprintf("Real-time face detection - %s", cam))
    disp = frm.add_display()
    [disp.width, disp.height] = [1536, 1024]
    frm.center()
    frm.show()
    [frm.width, frm.height] = [stream.frame_width + 80, stream.frame_height + 120]

    layer = new(qvectorlayer3d)
    renderer = disp.create_opengl_renderer()
    renderer.background_color = "black"
    renderer.enable_zbuffer = true
    renderer.draw_backfaces = true
    renderer.show_coords = false
    renderer.pitch = 340

    draw_stilllife(renderer, layer)

    while !frm.closed()
        if vidstate.is_playing || vidstate.show_next_frame
            if vidstate.is_playing ? !vidreadframe(stream) : false
                vidseek(stream,0) % Play in a loop
            endif
            A = facedetection(stream.rgb_data)

            if size(A,0)>0
                renderer.roll = -90*asin(0.5*(A[0]+A[2])/size(stream.rgb_data,1))
                renderer.pitch = 280+0.25*(A[1]+A[3]-0.5*size(stream.rgb_data,0))
            endif

            layer2 = new(qvectorlayer)
            layer2.translatetransform([10, 10])
            layer2.scaletransform([0.5,0.5])
            layer2.drawimage([0,0], float(stream.rgb_data))
            layer2.setpencolor([1,0,0,0])
            layer2.drawrect([0,0],size(stream.rgb_data,[1,0]))
            layer2.setpencolor([0,1,0,0])
            for i=0..size(A,0)-1
                layer2.drawrect(A[i,0..1],A[i,2..3])
            end

            renderer.clear()
            renderer.add(layer, "layer")
            renderer.add(layer2, "layer")

            if frm.closed()
                break
            endif

            vidstate.allow_seeking = false
            vidstate.allow_seeking = true
            vidstate.show_next_frame = false
        endif
        pause(50)
    end
end
```

The C++ host program is then as follows:

face_detection.cpp:

```
// Note : quasar_dsl.h should be placed first!
#include "quasar_dsl.h"
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/legacy/legacy.hpp>
#include <opencv2/objdetect/objdetect.hpp>
#include <opencv2/highgui/highgui.hpp>

using namespace cv;
using namespace quasar;

static CascadeClassifier face_cascade;

Mat convert_image(qvalue_t &qv)
{
    IQuasarHost* host = IQuasarHost::GetInstance();
    int C = qv.dim3;

    Mat img(qv.dim1, qv.dim2, C == 3 ? CV_8UC3 : CV_8UC1);

    CubeBase<uint8_t> data = host->LockCube<uint8_t>(qv, LOCK_READ);

    // Need to convert from BGR to RGB
    for (int m = 0; m < img.rows; m ++)
    {
        uint8_t *dstPtr = &img.data[m * img.step.p[0]];
        const uint8_t *srcPtr = (const uint8_t *) &data.data[m * qv.dim2 * C];

        for (int n = 0; n < img.cols; n ++)
        {
            dstPtr[n*C+0] = srcPtr[n*C+2];
            dstPtr[n*C+1] = srcPtr[n*C+1];
            dstPtr[n*C+2] = srcPtr[n*C+0];
        }
    }

    host->UnlockCube(qv, LOCK_READ);
    return img;
}

void facedetection(qvalue_t *argsIn, int nArgsIn, qvalue_t *argsOut, int nArgsOut)
{
    IQuasarHost* host = IQuasarHost::GetInstance();
    Mat captureFrame = convert_image(argsIn[0]);
    Mat grayscaleFrame;

    //convert captured image to gray scale and equalize
    cvtColor(captureFrame, grayscaleFrame, CV_BGR2GRAY);
    equalizeHist(grayscaleFrame, grayscaleFrame);

    //create a vector array to store the face found
    std::vector<Rect> faces;

    //find faces and store them in the vector array
    face_cascade.detectMultiScale(grayscaleFrame, faces,
    1.1, 3, CV_HAAR_SCALE_IMAGE, Size(150,150));

//create a matrix containing the face coordinates
    qvalue_t qv = host->CreateMatrix<scalar>(faces.size(), 4);
    MatrixBase<scalar> mtx = host->LockMatrix<scalar>(qv, LOCK_WRITE);

    for (int i = 0; i < faces.size(); i ++)
    {
        mtx.data[4*i+0] = faces[i].x;
        mtx.data[4*i+1] = faces[i].y;
        mtx.data[4*i+2] = faces[i].x + faces[i].width;
        mtx.data[4*i+3] = faces[i].y + faces[i].height;
    }

    host->UnlockMatrix(qv, LOCK_WRITE);
    argsOut[0] = qv;
}

void sample(void)
{
    // Initialize OpenCV face detection
    char xmlFile[_MAX_PATH];
    size_t retSize;
    getenv_s(&retSize, xmlFile, "OPENCV_DIR");
    strcat_s(xmlFile, "/../../../sources/data/haarcascades"
    "/haarcascade_frontalface_alt2.xml");

    // Create the cascade classifier object used for the face detection
    face_cascade.load(xmlFile);

    // Initialize Quasar
    using quasar::ref;
    ref<IQuasarHost> host = IQuasarHost::Create(_T("cuda"));

    // Register our function
    QValue lambdaType = QValue::LookupType(_T("[cube[uint8]->mat]"));
    QValue lambda = QValue::CreateLambda(lambdaType, facedetection);
    host->WriteVariable(_T("facedetection"), lambda);

    // Load the Quasar program
    tprintf(_T("Compiling code...\n"));
    LPCTSTR errorMsg = NULL;
    if (!host->LoadSourceModule(_T("opencv_facedetection.q"), &errorMsg))
    {
        tprintf(_T("Error while compiling program:\n%s\n"), errorMsg);
        return;
    }

    // Call the "run" function
    Function run(_T("run()"));
    run();

    host->RunApp();
}
```

An important function is convert_image, which converts from a Quasar cube object to an OpenCV matrix. In the conversion process, the R and the B values need to be swapped. Note that the conversion process imposes a (small) additional overhead, which should be avoided when possible.

Example 3: integrating CUDA C/C++ with Quasar

This sample demonstrates the interoperability of Quasar with CUDA, so that existing CUDA applications can use routines programmed in CUDA.

The function host−>Lock is used with parameter MEMRESOURCE_SINGLE_CUDA to obtain a CUDA device pointer (CUdeviceptr) for a Quasar object. Then, various CUDA operations are performed using this pointer (such as copying the memory back to the CPU).

```
#include "quasar_dsl.h"
#include "cuda.h"

static void checkResult(CUresult result)
{
    if (result != CUDA_SUCCESS)
    {
        tprintf(_T("CUDA operation failed with error code %d\n"), result);
        exit(-1);
    }
}

void sample(void)
{
    using namespace quasar;

    ref<IQuasarHost> host = IQuasarHost::Create(_T("cuda0.device.xml"), false);
    if (host == NULL)
    return; // Problem loading Quasar

    Function linspace(_T("linspace(scalar,scalar,scalar)"));
    Function ones(_T("ones()"));
    Function transpose(_T("transpose()"));
    Function imshow(_T("imshow(cube)"));
    const int N = 512;

    QValue im = ones(N,1) * linspace(0,255,N);

    // Obtain the current CUDA context & device name
    CUcontext ctx;
    checkResult(cuCtxGetCurrent(&ctx));
    tprintf(_T("CUDA context: %x\n"), ctx);

    CUdevice device;
    checkResult(cuCtxGetDevice(&device));

    char deviceName[80];
    checkResult(cuDeviceGetName(deviceName, sizeof(deviceName), device));
    printf("CUDA device name: %s\n", deviceName);

    // Now we attempt to obtain the device pointer for im
    CUdeviceptr devicePtr;
    LockResults result = host->Lock(im, TYPE_SCALAR, LOCK_READ,
        MEMRESOURCE_SINGLE_CUDA, (void**) &devicePtr);

    switch (result)
    {
    // Called in case we want to obtain a CUDA pointer when running Quasar in
    // CPU mode.
    case LOCKRESULT_RES_NOT_AVAILABLE:
        tprintf(_T("Lock failed - the requested resource is not available\n"));
        break;
    case LOCKRESULT_OUT_OF_MEM:
        tprintf(_T("Lock failed - insufficient memory resources\n"));
        break;
    case LOCKRESULT_INVALID:
        tprintf(_T("Lock failed - invalid request\n"));
        break;
    }

    tprintf(_T("CUDA device pointer: %x\n"), devicePtr);

    //=====================================================================
    // 1. Accessing Quasar matrices from CUDA
    //=====================================================================
    // To check whether the device pointer is valid, we copy a number of values to
    // system memory
    scalar vals[8];
    checkResult(cuMemcpyDtoH(vals, devicePtr, sizeof(scalar)*8));
    for (int i = 0; i < 8; i ++)
        tprintf(_T("%g "), vals[i]);
    tprintf(_T("\n"));

    // Always need to unlock!!!
    host->Unlock(im, LOCK_READ, MEMRESOURCE_SINGLE_CUDA);

    //=====================================================================
    // 2. Accessing CUDA data from Quasar
    //=====================================================================
    const int P = 256;
    const int Q = 256;
```

```
    scalar data[P*Q];
    for (int m = 0; m < P; m++)
        for (int n = 0; n < Q; n++)
        {
            data[m*Q+n]=cos(4*m*n*M_PI/(P*Q));
        }

    CUdeviceptr devicePtr2;
    checkResult(cuMemAlloc(&devicePtr2, sizeof(scalar) * P*Q));
    checkResult(cuMemcpyHtoD(devicePtr2, data, sizeof(scalar) * P*Q));

    // Now, we want to construct a Quasar matrix with the specified device
    // pointer.
    QValue im2 = host->CreateMatrix<scalar>(P,Q);
    host->Lock(im2, TYPE_SCALAR, LOCK_WRITE,
        MEMRESOURCE_SINGLE_CUDA, (void**) &devicePtr);
    checkResult(cuMemcpyDtoD(devicePtr, devicePtr2, sizeof(scalar) * P*Q));
    host->Unlock(im2, LOCK_WRITE, MEMRESOURCE_SINGLE_CUDA);

    checkResult(cuMemFree(devicePtr2));

    imshow(im);
    imshow(im2);

    host->RunApp();
}
```

Example 4: integrating OpenCL with Quasar

The follwing sample is an OpenCL port of the previous sample. The sample demonstrates the interoperability of Quasar with OpenCL, so that existing OpenCL applications can use routines programmed in OpenCL.

In particular, the example shows the usage of host−>Lock with parameter MEMRESOURCE_SINGLE_OPENCL to obtain an OpenCL device pointer (cl_mem) for a Quasar object.

Note that this sample can only work in combination with an OpenCL device (therefore, opencl0.device.xml is specified as the device configuration file).

```
#include "quasar_dsl.h"
#include "CL/OpenCL.h"

static void checkResult(cl_int result)
{
    if (result != CL_SUCCESS)
    {
        tprintf(_T("OpenCL operation failed with error code %d\n"), result);
        exit(-1);
    }
}

void sample12(void)
{
    using namespace quasar;

    // Need a Hyperion device configuration file for OpenCL
```

```
ref<IQuasarHost> host = IQuasarHost::Create(_T("opencl0.device.xml"), false);
if (host == NULL)
return; // Problem loading Quasar

Function linspace(_T("linspace(scalar,scalar,scalar)"));
Function ones(_T("ones()"));
Function transpose(_T("transpose()"));
Function imshow(_T("imshow(cube)"));
const int N = 512;

QValue im = ones(N,1) * linspace(0,255,N);

// Obtain access to the OpenCL context & command queue
cl_context context;
cl_command_queue cmdQueue;
host->QueryProperty(OPENCL_CURRENT_CONTEXT, 0, (void**)&context, sizeof(context));
host->QueryProperty(OPENCL_COMMANDQUEUE, 0, (void**)&cmdQueue, sizeof(cmdQueue));

tprintf(_T("OpenCL context: %x\n"), context);
tprintf(_T("OpenCL command queue: %x\n"), cmdQueue);

// Now we attempt to obtain the device pointer for im
cl_mem devicePtr;
LockResults result = host->Lock(im, TYPE_SCALAR, LOCK_READ,
   MEMRESOURCE_SINGLE_OPENCL, (void**) &devicePtr);

switch (result)
{
// Called in case we want to obtain a OpenCL pointer when running Quasar in
// CPU mode.
case LOCKRESULT_RES_NOT_AVAILABLE:
   tprintf(_T("Lock failed - the requested resource is not available\n"));
   break;
case LOCKRESULT_OUT_OF_MEM:
   tprintf(_T("Lock failed - insufficient memory resources\n"));
   break;
case LOCKRESULT_INVALID:
   tprintf(_T("Lock failed - invalid request\n"));
   break;
}

tprintf(_T("OpenCL device pointer: %x\n"), devicePtr);

//==================================================================
// 1. Accessing Quasar matrices from OpenCL
//==================================================================
// To check whether the device pointer is valid, we copy a number of values to
// system memory
scalar vals[8];
checkResult(clEnqueueReadBuffer(cmdQueue, devicePtr, true, 0,
   sizeof(scalar)*8, vals, 0, NULL, NULL));

for (int i = 0; i < 8; i ++)
   tprintf(_T("%g "), vals[i]);
tprintf(_T("\n"));

// Always need to unlock!!!
host->Unlock(im, LOCK_READ, MEMRESOURCE_SINGLE_OPENCL);

//==================================================================
    // 2. Accessing OpenCL data from Quasar
    //========================================================================
    const int P = 256;
    const int Q = 256;
    scalar data[P*Q];
    for (int m = 0; m < P; m++)
        for (int n = 0; n < Q; n++)
        {
            data[m*Q+n]=cos(4*m*n*M_PI/(P*Q));
        }

    cl_int errCode;
    cl_mem devicePtr2 = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
        sizeof(scalar) * P * Q, data, &errCode);
    checkResult(errCode);

    // Now, we want to construct a Quasar matrix with the specified device
    // pointer.
    QValue im2 = host->CreateMatrix<scalar>(P,Q);
    host->Lock(im2, TYPE_SCALAR, LOCK_WRITE,
        MEMRESOURCE_SINGLE_OPENCL, (void**) &devicePtr);
    checkResult(clEnqueueCopyBuffer(cmdQueue, devicePtr2, devicePtr, 0, 0,
        sizeof(scalar) * P*Q, 0, NULL, NULL));
    host->Unlock(im2, LOCK_WRITE, MEMRESOURCE_SINGLE_OPENCL);

    checkResult(clReleaseMemObject(devicePtr2));

    imshow(im);
    imshow(im2);

    host->RunApp();
}
```

Example 5: user-type matrices

The following example illustrates how to construct user-type matrices, via the C++ external interface. Such objects can then be transparently used on the GPU, with automatic memory transfers performed by Quasar.

```
#include "quasar_dsl.h"

using namespace quasar;

struct vertex
{
    float x;
    float y;
    float z;
};

namespace quasar
{
    DECLARE_STRUCT_TYPE(vertex); // Generates type information for our struct
    IMPLEMENT_TYPE(TYPE_TYPEDOBJECT, vertex, _T("vertex"))
}

void sample16(void)
```

```
{
    using namespace quasar;
    using quasar::ref;

    // Creates the quasar host and loads the specified module
    ref<IQuasarHost> host = IQuasarHost::Create(_T("cuda"), true);

    // Problem loading Quasar
    if (host == NULL)
        return;

    Function print(_T("print()"));

    try
    {
        TypeBuilder builder(_T("sample16"), _T("vertex"));
        builder.AddField(_T("x"), Type(_T("scalar'single")));
        builder.AddField(_T("y"), Type(_T("scalar'single")));
        builder.AddField(_T("z"), Type(_T("scalar'single"))); // Single precision scalar number
        Type type = builder.CreateType();

        vertex vertices[3] = { {0,1,2}, {2,0,1}, {1,2,0} };
        QValue array = vertices;

        QValue elem = QValue::FromObj(vertices[2]);
        print(_T("The vertex is: "), elem);

        QValue str = QValue(_T("Test string"));
        string_t strVal = str.operator string_t();
        tprintf(_T("String value: %s\n"), (LPCTSTR)strVal);

        vertex v = (vertex) array(0);
        tprintf(_T("Vertex read from array: (%g %g %g)\n"), v.x, v.y, v.z);
        v.x = 5;
        array(1) = v;

        print(_T("The list is: "), array);
        print(_T("The element at position 0 is: "), array(0));
    }
    catch (exception_t ex)
    {
        print(_T("An error occurred: "), ex.message);
    }
}
```

Example 6: multi-GPU programming

The following example illustrates how to use the quasar :: RuntimeControl class in order to schedule Quasar kernels to multiple GPUs.

```
#include "quasar_dsl.h"
using namespace quasar;

void sample20(void)
{
    using namespace quasar;
```

```
LPCTSTR sourceProgram =
    _T("function[] = __kernel__ filter_kernel(x:cube, y : cube, offset : ivec3, pos : ivec3)\n")
    _T(" dpos = pos + offset;\n")
    _T(" M = 9\n")
    _T(" sum = 0.0\n")
    _T(" for m = -M..M\n")
    _T(" for n = -M..M\n")
    _T(" sum += x[dpos + [m, n, 0]]\n")
    _T(" end\n")
    _T(" end\n")
    _T(" y[pos] = sum ./ (2 * M + 1) ^ 2\n")
    _T("end\n");

LPCTSTR errorMsg = NULL;
using quasar::ref;

ref<IQuasarHost> host = IQuasarHost::Create(_T("hycuda2.device.xml"));
if (host == NULL)
    return; // Problem loading Quasar

// Import Quasar functions
Function tic(_T("tic()"));
Function toc(_T("toc(...??)"));
Function imread(_T("imread(string)"));
Function imshow(_T("imshow(cube)"));
Function parallel_do(_T("parallel_do(...??)"));
Function uninit(_T("uninit(??)"));

tprintf(_T("Compiling code...\n"));
tic();
if (!host->LoadModuleFromSource(_T("sample20"), sourceProgram, &errorMsg))
{
    tprintf(_T("Error while compiling program:\n%s\n"), errorMsg);
    return;
}
toc();

QValue kernelFunc;
if (!QValue::ReadHostVariable(_T("filter_kernel"), kernelFunc))
    tprintf(_T("Error reading variable 'filter_kernel'!\n"));

// Load an image
QValue img = imread(_T("lena_big.tif"));
QValue img_out = uninit(size(img));
RuntimeControl runtimeCtl;

tic();
for (int k = 0; k < 50; k++)
{
    // Select the GPU
    runtimeCtl.ScheduleGPU(k % 2);

    // Execute the kernel function
    parallel_do(size(img), img, img_out, make_scalar3(0, 0, 0), kernelFunc);
}
toc();

// Switch back to the automatic scheduling mode
runtimeCtl.ScheduleAuto();
```

```
    host->RunApp();
}
```

Example device configuration files

In this section, we give some example device XML configuration files. These configuration files are to be used with the Hyperion runtime engine; they allow defining a computation engine with multiple devices and while setting various parameters. The main XML tags are <cpu−device>, <cuda−device> and <opencl−device> (for respectively defining a CPU device, a CUDA device and an OpenCL device). It is therefore possible to combine CUDA devices with OpenCL devices (for example, an NVidia Geforce GPU with an Intel HD Graphics GPU).

The configuration files are generated automatically in the after a fresh Quasar installation, they are put in C:\Users\UserName \AppData\Local\Quasar (Windows) and ~/. config/Quasar (Linux). It is possible to let Quasar regenerate these XML files, using Quasar.exe −install :hyperion (Windows) or ./ quasar −install :hyperion (Linux).

CPU only device config file:

```
<quasar>
    <computation-engine name="v2 CPU engine" short-name="CPU - v2">
        <cpu-device num-threads="2" max-cmdqueue-size="32" />
    </computation-engine>
</quasar>
```

CUDA device config file (note that ordinal specifies the GPU index):

```
<quasar>
    <computation-engine name="v2 CUDA/CPU engine" short-name="CUDA - v2">
        <cpu-device num-threads="4" max-cmdqueue-size="16" cuda-hostpinnable-memory="true" />
        <cuda-device max-concurrency="4" max-cmdqueue-size="128" ordinal="0" />
    </computation-engine>
</quasar>
```

CUDA multi-GPU device config file:

```
<quasar>
    <computation-engine name="v2 dual CUDA/CPU engine" short-name="CUDA - v2">
        <cpu-device num-threads="4" max-cmdqueue-size="16" cuda-hostpinnable-memory="true" />
        <cuda-device max-concurrency="4" max-cmdqueue-size="128" ordinal="0" />
        <cuda-device max-concurrency="4" max-cmdqueue-size="128" ordinal="1" />
    </computation-engine>
</quasar>
```

OpenCL device config file:

```
<quasar>
    <computation-engine name="v2 GeForce GTX 780M engine" short-name="OpenCL - v2">
        <cpu-device num-threads="2" max-cmdqueue-size="32" />
        <opencl-device max-concurrency="4" max-cmdqueue-size="16" type="*" ordinal="1" />
    </computation-engine>
</quasar>
```

Example: developing a camera interface to Quasar

As an example of using the Quasar C++ API, we explain how a camera interface with Quasar can be created. We will do this for the Blackmagic Decklink HDMI capture card, although this example can easily be extended to other cards of cameras. The Decklink card can be interfaced from C++, by including the header files DeckLinkAPI_h.h and DeckLinkDevice.h. The source code for this example can be found in the Interop_Samples/DeckLinkVideoCapture folder.

For the interface with Quasar, we will define the following functions in C++:

Function name	Description
decklink_get_devices	Returns a vector with the names of the available devices
decklink_open	Opens the specified Decklink device
decklink_close	Closes the current Decklink device
decklink_startcapture	Starts capturing on the current Decklink device
decklink_stopcapture	Stops capturing on the current Decklink device
decklink_readframe	Waits and reads one frame from the current Decklink device
decklink_get_displaymodes	Returns the display modes for the current device
module_init	Function called by Quasar when the camera interface module is loaded
module_term	Function called by Quasar when the camera interface module is unloaded

In the following, we discuss the .cpp implementation file, as well as the individual functions that are required. First, we include the necessary header files:

```
// File: decklinkcapture.cpp
#include "quasar_dsl.h"
#include "DeckLinkAPI_h.h"
#include "DeckLinkDevice.h"
```

Here, quasar_dsl.h includes all definitions of the Quasar Domain Specific Language. DeckLinkAPI_h.h and DeckLinkDevice.h are header files for the camera API. Next, we define a number of static variables that will be used later on.

```
using namespace quasar;
class ScreenPreviewCallback;

// Static objects (static objects are bad, here they are used for
// simplicity)
static IQuasarHost* host;
static DeckLinkDeviceDiscovery * discovery;
static DeckLinkDevice * device;
static Function* error;
static ScreenPreviewCallback * screenPreviewCallback;
```

Note that static variables can be avoided by wrapping the native pointers. For simplicity, we don't take this approach here. The Decklink camera API further requires the implementation of the IDeckLinkScreenPreviewCallback interface, which inherits from COM's IUnknown. Therefore, we will be required to implement the reference counting functions AddRef and Release as well as the interfacing querying function QueryInterface.

```
class ScreenPreviewCallback : public IDeckLinkScreenPreviewCallback
{
private:
    ULONG m_cRef;
    HANDLE frameEvent;
    IDeckLinkVideoFrame * frame;
    QValue buffer;

public:
    ScreenPreviewCallback();
    virtual ~ScreenPreviewCallback();
    ULONG AddRef();
    ULONG Release();
    HRESULT QueryInterface(REFIID riid, LPVOID * ppvObj);
    HRESULT STDMETHODCALLTYPE ScreenPreviewCallback::DrawFrame(
        IDeckLinkVideoFrame * theFrame);
    void ReadFrame(qvalue_t &buffer);
};
```

We rely on a Win32 event to indicate that a frame is ready to be processed. This event is created inside the ScreenPreviewCallback constructor.

```
ScreenPreviewCallback::ScreenPreviewCallback() : m_cRef(1)
{
    frameEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
    frame = NULL;
}
```

The ScreenPreviewCallback destructor simply destroys the Win32 event and releases the frame buffer.

```
ScreenPreviewCallback::~ScreenPreviewCallback()
{
    CloseHandle(frameEvent);
    if (frame != NULL)
    {
        frame->Release();
        frame = NULL;
    }
}
```

ScreenPreviewCallback :: DrawFrame is called whenever a new frame is available. Here, we pulse frameEvent indicating that a new frame is ready to be processed.

```
HRESULT STDMETHODCALLTYPE ScreenPreviewCallback::DrawFrame(
    IDeckLinkVideoFrame * theFrame)
{
    if (frame != NULL)
    {
        frame->Release();
        frame = NULL;
    }
    frame = theFrame;
    theFrame->AddRef();
    PulseEvent(frameEvent);
    return S_OK;
}
```

We provide a generic implementation for QueryInterface. This provides no specific functionality.

```
HRESULT ScreenPreviewCallback::QueryInterface(REFIID riid, LPVOID * ppvObj)
{
    // Always set out parameter to NULL, validating it first.
    if (!ppvObj)
        return E_INVALIDARG;
    * ppvObj = NULL;
    if (riid == IID_IUnknown)
    {
        // Increment the reference count and return the pointer.
        * ppvObj = (LPVOID)this;
        AddRef();
        return NOERROR;
    }
    return E_NOINTERFACE;
}
```

The reference counting mechanism is implemented in the following functions. We use atomic integer operations to increase and decrease the reference counts. When the reference count becomes zero, the object is destroyed using delete this.

```
ULONG ScreenPreviewCallback::AddRef()
{
    InterlockedIncrement(&m_cRef);
    return m_cRef;
}
ULONG ScreenPreviewCallback::Release()
{
    // Decrement the object's internal counter.
    ULONG ulRefCount = InterlockedDecrement(&m_cRef);
    if (0 == m_cRef)
    {
        delete this;
    }
    return ulRefCount;
}
```

In the following method, a frame is read and converted to a Quasar value (qvalue_t). First, we wait for a frame to be ready. Then we read the frame row by row. Note that the input frames contain a pitch that may be different from the image width times the pixel data size (here 2 bytes, corresponding with the YUV422 format); Quasar however, currently does not support pitched image data. A straightforward solution would be to allocate an aligned frame buffer and then crop the image within Quasar. For simplicity, here we allocate an unaligned frame buffer and copy the rows of the image individually using memcpy.

```
void ScreenPreviewCallback::ReadFrame(qvalue_t &bufferOut)
{
    WaitForSingleObject(frameEvent, INFINITE);

    int width = (int)frame->GetWidth();
    int height = (int)frame->GetHeight();
```

```
  // Allocate frame buffer if not allocated yet.
   if (((qvalue_t) buffer).type == TYPE_VOID ||
       size(buffer, 0) != height ||
       size(buffer, 1) != width)
   {
       buffer = host->CreateMatrix<unsigned short>(height, width);
   }

   unsigned char * buf = NULL;
   if (frame->GetBytes((void ** )&buf) == S_OK)
   {
       auto cube = host->LockMatrix<unsigned char>(buffer, quasar::LOCK_WRITE, MEMRESOURCE_CPU);
       int pitch = (int)frame->GetRowBytes();
       int row_bytes = width * sizeof(unsigned short);

       for (int row = 0; row < height; row++)
       {
           memcpy(cube.data + row * row_bytes,
               buf + pitch * row,
               row_bytes);
       }

       host->UnlockMatrix(buffer, quasar::LOCK_WRITE, MEMRESOURCE_CPU);
   }

   bufferOut = (qvalue_t) this->buffer;
}
```

The following function returns a list of DeckLink devices, as a cell vector of type string vec[??]. We use the QValue constructor to convert from const char * to a Quasar string value. Note that the result is passed as a pointer output argument instead of as a return value. This is to uniformize the application binary interface (ABI, which may sometimes differ between compilers).

```
extern "C" EXPORT __device__ void ENTRY(decklink_get_devices)(void* closure, QValue* devices)
{
   QValue result = host->CreateVector<qvalue_t>(discovery->Count());

   for (int index = 0; index < discovery->Count(); index++)
   {
       DeckLinkDevice * dd = new DeckLinkDevice(discovery->GetDeckLink(index));
       dd->Init();
       result(index) = QValue(dd->GetDeviceName().c_str());
       dd->Release();
   }
   * devices = result;
}
```

The exported function is always declared using:

```
extern "C" EXPORT __device__ void ENTRY(XXXX)
```

in order to work consistently across platforms. The Quasar compiler also parses the .cpp implementation file to determine the functions that can be called from Quasar. Functions that do not have the above signature cannot be called from Quasar.

The following function simply checks if the camera is initialized, if not, the error function is called.

```
static void decklink_checkdevice()
{
    if (device == NULL)
    {
        (* error)(_T("No device opened: use decklink_open() first to open a device"));
        return;
    }
}
```

Next, decklink_get_displaymodes returns a vector with the display modes; every display mode is represented by a string. The implementation is fairly similar to decklink_get_devices .

```
extern "C" EXPORT __device__ void ENTRY(decklink_get_displaymodes)(void * closure, QValue *
     displayModes)
{
    decklink_checkdevice();

    std::vector<std::wstring> display_modes;
    device->GetDisplayModeNames(display_modes);
    QValue result = host->CreateVector<qvalue_t>(display_modes.size());

    for (int index = 0; index < display_modes.size(); index++)
    {
        result(index) = QValue(display_modes.at(index).c_str());
    }
    * displayModes = result;
}
```

decklink_close closes the current camera, by stopping the camera capturing.

```
extern "C" EXPORT __device__ void ENTRY(decklink_close)(void* closure)
{
    if (device != NULL)
    {
        if (device->IsCapturing())
            device->StopCapture();
        device->Release();
        device = NULL;
    }
}
```

decklink_open opens the specified camera index. We simply create a new instance of DeckLinkDevice and use it to initialize the static variable device.

```
extern "C" EXPORT __device__ void ENTRY(decklink_open)(void* closure, int device_index)
{
    ENTRY(decklink_close)(NULL);

    if (device_index < 0 || device_index >= discovery->Count())
    {
        (* error)(_T("decklink_open: invalid index!"));
        return;
    }
    device = new DeckLinkDevice(discovery->GetDeckLink(device_index));
    device->Init();
}
```

decklink_startcapture needs to be called to begin capturing using the specified display mode (the index corresponds to the list returned by decklink_get_displaymodes). We create a ScreenPreviewCallback instance and pass it to device−>StartCapture.

```
extern "C" EXPORT __device__ void ENTRY(decklink_startcapture)(void* closure, int display_mode_index)
{
    decklink_checkdevice();

    screenPreviewCallback = new ScreenPreviewCallback();
    device->StartCapture(display_mode_index, screenPreviewCallback, false);
    screenPreviewCallback->Release();
}
```

Stopping the capturing process is fairly straightforward:

```
extern "C" EXPORT __device__ void ENTRY(decklink_stopcapture)(void* closure)
{
    decklink_checkdevice();
    device->StopCapture();
    screenPreviewCallback = NULL;
}
```

Perhaps the most interesting function is decklink_readframe, which waits until a frame is ready to be processed and then returns a reference to this frame. The main work is actually done in ScreenPreviewCallback :: ReadFrame.

```
extern "C" EXPORT __device__ void ENTRY(decklink_readframe)(void* closure, QValue* pbuffer)
{
    decklink_checkdevice();
    if (!device->IsCapturing() || screenPreviewCallback == NULL)
    {
        (* error)(_T("decklink_readframe: device is currently not capturing!"));
        return;
    }

    qvalue_t buffer;
    screenPreviewCallback->ReadFrame(buffer);
    * pbuffer = buffer;
    pbuffer->Retain();
}
```

Finally, two functions module_init, module_term cannot be called from Quasar but are called implicitly by the runtime system, to intialize and deinitialize the camera interface. In module_init we connect to the Quasar host, we perform function binding (which allows Quasar functions to be called from C++) and we use the decklink discovery API to determine the available devices in the system.

```
extern "C" EXPORT __device__ void module_init(
    LPCVOID hostprivParams,
    LPCVOID moduleDetails,
    int flags)
{
    host = IQuasarHost::Connect(hostprivParams, moduleDetails, flags);

    error = new Function(L"error(...)"); // Function binding.

    CoInitialize(NULL); // Initializes COM

    discovery = new DeckLinkDeviceDiscovery();
    discovery->Enable();

    // Wait for the devices to be enumerated
    if (!discovery->WaitForDevices())
    {
        wprintf(L"Warning: no DeckLink devices discovered!\n");
    }
}
```

In module_term we can safely clean up all the native resources, by releasing the individual objects.

```
extern "C" EXPORT __device__ void module_term()
{
    if (device != NULL)
    {
        device->StopCapture();
        device->Release();
        device = NULL;
    }
    if (discovery != NULL)
    {
        discovery->Release();
        discovery = NULL;
    }
    if (error != NULL)
    {
        delete error;
        error = NULL;
    }
    // Release the host connection
    if (host)
    {
        delete host;
        host = NULL;
    }

    CoUninitialize(); // Uninitializes COM
}
```

The above .cpp file can then be compiled using G++ or MSVC and linked with the Decklink static library. It is then important to name the resulting binary decklinkcapturedemo.64. dll (or decklinkcapturedemo.64.so in Linux) and to place it in the same folder as decklinkcapturedemo.cpp, so that Quasar can easily find the library. Here, 64 indicate the bitness (32 for x86, 64 for x64).

Below, we give an example of how the above library can be used from Quasar. We can directly import the "decklinkcapture.cpp" source file. The kernel function convert_YUV422_to_RGB is used to convert the video data from YUV422 format to RGB. When a GPU is available, this function will be executed in parallel on the GPU.

```
% File: decklinkcapturedemo.q

import "decklinkcapture.cpp" % Import our C++ library
import "Quasar.UI.dll"

function index = indexof(list, text)
    index = -1
    for i=0..numel(list)-1
        if list[i] == text
            index = i
        endif
    endfor
endfunction

devices = decklink_get_devices()
print devices

device_name = "DeckLink Mini Recorder" % note: currently only one capture card at the time can be
     opened
display_mode = "1080i59.94" % note: needs to match the camera settings

device_index = indexof(devices, device_name)
if device_index<0
    error "Could not select device: ", device_name, ". Device unknown"
endif

decklink_open(device_index)
display_modes = decklink_get_displaymodes()
display_mode_index = indexof(display_modes, display_mode)
if display_mode_index<0
    error "Could not select display mode: ", display_mode
endif
decklink_startcapture(display_mode_index)

frm = form(sprintf("Camera: %s, display mode: %s", devices[device_index], display_mode))
disp = frm.add_display()
frm.show()

function [] = __kernel__ convert_YUV422_to_RGB(im_in : mat[uint16]'const, im_out : cube, pos : ivec2)
    val = [im_in[pos[0],pos[1] * 2],im_in[pos[0],pos[1] * 2+1]]

    Rc = [1.164, 1.596, 0.0]
    Gc = [1.164, -0.813, -0.391]
    Bc = [1.164, 0.0, 2.018]

    Y1 = shr(and(val[0],0ff00h),8)-16
    Y2 = shr(and(val[1],0ff00h),8)-16
    V = (and(val[0],000ffh)-128)
    U = (and(val[1],000ffh)-128)

    R1 = clamp(dotprod(Rc,[Y1,U,V]),255)
    G1 = clamp(dotprod(Gc,[Y1,U,V]),255)
    B1 = clamp(dotprod(Bc,[Y1,U,V]),255)

    R2 = clamp(dotprod(Rc,[Y2,U,V]),255)
    G2 = clamp(dotprod(Gc,[Y2,U,V]),255)
    B2 = clamp(dotprod(Bc,[Y2,U,V]),255)

    im_out[pos[0],pos[1] * 2+0,0..2] = [R1,G1,B1]
    im_out[pos[0],pos[1] * 2+1,0..2] = [R2,G2,B2]
```

```
endfunction

first = true
{!interpreted for}
while !frm.closed()
    im = decklink_readframe()
    if first
        imc = zeros(size(im,0),size(im,1),3)
        first = false
    endif
    parallel_do([size(im,0),size(im,1)/2],im,imc,convert_YUV422_to_RGB)

    disp.imshow(imc,[0,255])
    pause(0)
endwhile
decklink_close()
```

Current limitations: * When the processing rate is lower than the frame rate, the underlying buffer may overrun causing the software to hang in some camera systems. In practice, it may be necessary to implement a frame skip mechanism, which skips/disregards frames when the processing of the previous frames did not finish in a timely manner. * The above example only works with one camera at the time. Extensions are possible to support multiple cameras (for example, by passing the camera index to the function decklink_readframe).

Writing a managed Quasar library

In this section, we will explain how to write a library that can be used from Quasar programs. In general, there is a lot of flexibility in the Quasar runtime system, which allows a lot of possibilities (so many that I will not likely be able to cover them all in this document).

The idea is that complicated work happening behind the scenes is hidden from the user. The user has access to a rather "simple" interface and does not have to worry about computation engine issues, memory allocation etc.

In the following, the name of the library that we will develop, will be MyLibrary (the compiled version is MyLibrary.dll). For illustrational purposes, the library will we implemented in C#.

In Quasar programs, you can then simply access the library by using the statement:

```
import "MyLibrary.dll"
```

The main mechanism that the Quasar compiler/runtime uses for loading custom modules is *Reflection*. To write a Quasar library, it is best to copy-and-paste the following code into an empty C# file.

```
using System;
using Quasar.Computations;
using Quasar.Computations.Math;

namespace MyLibrary
{
    public static class MyLibrary
    {
        /// <summary>
        /// The computation engine to be used.
        /// </summary>
        static IComputationEngine ce;
        /// <summary>
        /// The evaluation stack, which is used for passing arguments to
        /// internal Quasar functions and for intermediate results of computation
        /// </summary>
        static IEvaluationStack es;

        /// <summary>
        /// The @run function is called by Quasar at initialization of the library
        /// (i.e., when executing import "MyLibrary.dll"). Here, we just keep
        /// track of the computation engine and the evaluation stack.
        /// </summary>
        public static void @run(IComputationEngine ce, IEvaluationStack es)
        {
            MyLibrary.ce = ce;
            MyLibrary.es = es;

            // Initialization code can be put here
        }

        /// <summary>
        /// The @main function is called by Quasar when the library is executed
        /// in standalone mode. For example, through the commandline:
        ///
        /// Quasar.exe -debug -gpu:0 Quasar.Video.dll
        /// (although I have to check if this use case is currently implemented)
        /// </summary>
        public static void @main(QValue args)
        {
            // Main function - called when the DLL is executed directly.
        }

        /// <summary>
        /// The @cleanup function is called by Quasar usually when the library
        /// is released (for example when the program is terminated, or when the
        /// debug session in Redshift is finished).
        ///
        /// At this stage, unmanaged resources can be freed.
        /// </summary>
        public static void @cleanup()
        {
        }
    }
}
```

There are a number of fields and functions exposed:

- IComputationEngine ce: interface to the computation engine (e.g. the CPU engine, the CUDA engine, the OpenCL engine). All computation engines adhere to the same interface, which makes it easy to develop programs that target different platforms.
- IEvaluationStack es: in Quasar, expressions are evaluated by pushing/popping objects to or from the evaluation stack. The evaluation stack also enables some internal memory optimizations.
- @run(): is called by Quasar during initialization of the library
- @main(): is rarely called, only when instructing Quasar to "execute" the library directly (such as the main function in Quasar code). For example, through the following commandline:

  ```
  Quasar.exe MyLibrary.dll
  ```

  you can start the Quasar library and the main function gets called.
- @cleanup(): function called when the library is unloaded (for example the program terminates, or the Quasar Redshift debug session is terminated). Here the module can (actually **must**) dispose all its internal unmanaged resources.

To compile, simply add a reference to Quasar.Runtime.dll and you are ready to go.

Also note that the Quasar compiler is able to generate libraries itself from Quasar code; to this end, the compiler will expand code that conforms to what is explained here.

Memory management (important)

To enable efficient GPU memory management, there is no automatic memory management (read: garbage collection) at the level of Quasar libraries. This means that it *is* possible to write libraries that cause memory leaks. However, the memory profiler that comes with Quasar is able to detect the objects that are leaked.

Memory leak detection tools

To invoke the memory profiler from the commandline, you can use the −memleak option:

```
Quasar.exe -memleak MyLibrary.dll
```

Note that, in order to find out the locations of the memory leak (in terms of line numbers), it may be advisable to create a Quasar program that imports the library and run the program in memory leak detection mode:

```
MyProg.q:
import "MyLibrary.dll"
% Do some stuff...

Quasar.exe -memleak MyProg.q
```

This way, line numbers (of MyProg.q) will be obtained of locations that cause the memory leaks.

Manual memory management (mandatory!)

For manual memory management, reference counting is used, and there are a number of conventions:

1. When an object is stored somewhere for later reference (e.g. in a field, dictionary, hash table, . . .) its reference count must be increased (see IComputationEngine.AddRef)
2. When an object is not needed anymore, its reference count must be decreased (see IComputationEngine.Release)
3. Circular references need to be avoided at all cost. As an alternative, weak references can be used. The only exceptions are the *typed objects* (QTypedObject), the Quasar typed object system implements a scheme that is free from memory leaks by circular references.

Quasar Values *(QValue)*

The main data type in Quasar is QValue. Quasar values are a "general" approach to hold information. Basically, the QValue is a struct that encapsulates the actual value (object value) through a boxing operation. For efficient processing, there is also a type member, that holds information about the type.

```
public struct QValue
{
    public enum Types
    {
        // Main types
        QVT_VOID,
        QVT_SCALAR, // Double
        QVT_COMPLEXSCALAR, // Quasar.Computations.Math.ComplexScalar
        QVT_MATRIX, // Quasar.Computations.Math.Matrix
        QVT_COMPLEXMATRIX, // Quasar.Computations.Math.ComplexMatrix
        QVT_STRING, // System.String
        QVT_CELLMATRIX, // Quasar.Computations.Math.Matrix
        QVT_UNTYPEDOBJECT, // Quasar.Computations.QObject
        QVT_LAMBDAEXPRESSION, // Quasar.Computations.QLambdaExpression
        QVT_TYPEDOBJECT, // Quasar.Computations.QTypedObject
        QVT_TYPEINFO // Quasar.TypeSystem.QTypeInfo
    }
    public Types type;
    public object value;
}
```

Shown above are the main Quasar data types. There are more types, however these are reserved for internal operations (e.g. for the interpreter). On the right in comments, the corresponding value class (for the field value) is shown. These classes will be explained later in more detail.

In Quasar, there are both *typed* objects and *untyped* objects. An object is *typed* when all the types of the fields are known at compile-time. Untyped is the opposite of typed. A typed object is defined in Quasar using the type statement:

```
% typed object
type point : class
    x : scalar
    y : scalar
end
p = point(x:=1.0, y:=1.0)

% analogue untyped object
p = object()
p.x = 1.0
p.y = 1.0
```

Typed objects are suitable for being transferred to the computation device (GPU), whereas untyped objects are more useful for rapid prototyping. Typed objects without pointers can directly be copied internally to the GPU memory (using a memcpy -like function), for this the types of all the fields need to be known by the compiler/run-time.

QVT_LAMBDAEXPRESSION represents both lambda expressions and functions:

```
my_lambda = () -> ()
function [] = my_function()
end
```

The unification is important internally, because it allows a lambda expression to be passed to a function (or other lambda expression) that expects a function variable as input argument.

QVT_TYPEINFO denotes runtime information about a given type.

QVT_VOID is only used as return value of functions or lambda expressions that do not return values (such as void in C/C++).

Computation Engine interface *(IComputationEngine)*

To interact with computation devices (e.g. GPU, multi-core CPU), a uniform interface is used. This interface is the IComputationEngine , and it exposes a relatively large number of functions. By using the interface, programs can easily address different types of platforms. In the future, a possibility is to also build a RemoteComputationEngine, to enable distributed processing on remote GPUs (please note that this is already possible in a completely different way using the Quasar distributed processing library).

```
public interface IComputationEngine : IDisposable
{
    string Name { get; }
    IEvaluationStack EvaluationStack {get; } // Get the current evaluation stack
    IMatrixFactory MatrixFactory { get; }
    TypeEnvironment TypeEnvironment { get; }
    Type FltType { get; }
    Type ComplexFltType { get; }
    // Evaluate a given operator (using the stack)
    void Process(Compiler.OperatorTypes op);
    // Construct a vector of size (rows), recursively called for higher dimensional matrices
    void ConstructMatrix(int num_rows_cols);
    // Construct a cell matrix of size (length)
    void ConstructCellMatrix(int length);
    // Perform an internal function call (for example: fft1, fft2, ...)
    void FunctionCall(string function_name, int num_args);
    // Check whether a given function is implemented by the engine
    bool FunctionExists(string function_name, int num_args);
    // Array indexing (getter)
    QValue ArrayGetAt(QValue x, int[] indices, BoundaryAccessMode boundsMode);
    // Array indexing (setter)
    void ArraySetAt(QValue x, int[] indices, QValue val, BoundaryAccessMode boundsMode);
    // Array indexing (slice getter)
    QValue ArrayGetSliceAt(QValue x, QValue[] indices, BoundaryAccessMode boundsMode);
    // Array indexing (slice setter)
    void ArraySetSliceAt(QValue x, QValue[] indices, QValue val, BoundaryAccessMode boundsMode);
    // Array indexing (generic getter, using the stack)
    void ArrayGetAt(QValue x, int num_indices, BoundaryAccessMode boundsMode);
    // Array indexing (generic setter, using the stack)
    void ArraySetAt(QValue x, int num_indices, BoundaryAccessMode boundsMode);
    QValue Addref(QValue x); // Add one reference to an given value
    void Release(QValue x); // Release one reference from a given value
    // module name without file extension
    void LoadModule(System.Reflection.Assembly assembly, string fileName, string moduleName);
    void UnloadModule(string moduleName);
    void RegisterBuiltinFunction(QFunctionDescriptor descriptor);
    void RegisterBuiltinFunction(string name, Delegate functor, BuiltInFunctionTypes functionType);
    void MVAssign(int num_args); // Multiple variable assignment
    RuntimeReductionEngine RuntimeReductionEngine { get; }
    void AddReduction(string text, QLambdaExpression expr, QLambdaExpression whereClause);
    void FinalizeType(TypedObjectInfo classDef, Type rawType);
    void AttachProfiler(IProfiler profiler);
    QTypeInfo FindType(string typeName, bool throwExceptionIfNotExists);
}
```

Some of the members are discusses below in more detail:

- Name: returns the name of the computation engine (for example *“Geforce GT 435M CUDA”*).

- EvaluationStack: this is mainly be used for passing values to functions (e.g. through FunctionCall (.) or Process (.)) and reading the return values.
- MatrixFactory: gives access to an object that allows you to create matrices with different dimensions in several ways.
- TypeEnvironment: reserved for internal usage.
- FltType: the current float datatype (either System.Single or System.Double).
- ComplexFltType: the current complex number datatype (either Quasar.Computations.Math.FltComplex or Quasar.Computations .Math.Complex).
- Process (.) : evaluation of various binary operations (for example: addition, subtraction, multiplication (matrix multiplication), divide, right division, power, point-wise multiplication, point-wise division,point-wise power, less than, less than or equal, greater, greater or equal, equal, not equal, assignment, negation, logical inversion, logical AND, logical or, . . .)
- ConstructMatrix (.) : constructs a matrix based on data that is pushed onto the stack. This function is used by the interpreter, and is useful when building a matrix out of several row vectors. For practical purposes, the IMatrixFactory interface should be slightly more easy to use.
- ConstructCellMatrix (.) : idem but for cell matrices. When the types of all input elements are equal, a typed cell matrix is created (e.g. vec[cube], mat[mat], . . .). Otherwise creates an untyped cell matrix (cell (.)).
- FunctionCall (.) : calls a given built-in function with the specified set of arguments (that are pushed to the evaluation stack). Some of the functions are: numel, size, abs, asin, acos, atan, atan2, ceil, round, cos, sin, exp, exp2, floor, mod, frac, log, log2, log10, max, min, pow, saturate, sign, sqrt, tan, angle, conj, float, int, isinf, isfinite , isnan, periodize, mirror_ext, clamp, sinh, cosh, tanh, asinh, acosh, atanh. Note that these functions are registered first by the runtime system, before they can be called (RegisterBuiltinFunction). This allows the runtime to define new functions, without requiring changes to the computation engines. Note that it is not possible to call functions defined in .q modules using FunctionCall (.) (unless they are defined through reductions).
- FunctionExists (.) : checks if the given function is defined and/or implemented by the computation engine.
- ArrayGetAt(.): gets a matrix element at the specified position and with the given boundary access mode.
- ArraySetAt (.) : sets the matrix element at the specified position to the given value
- ArrayGetSliceAt: idem but for matrix slices. The indices are each vectors with 0-based coordinates.
- ArraySetSliceAt: for setting a slice of a matrix. The indices are each vectors with 0-based coordinates. val should have the correct dimensions, otherwise an error is generated.
- Addref: adds a reference to the given object
- Release: releases a reference of the object
- FinalizeType: finalization & registration of a user-defined type (or class).
- AttachProfiler : when you are writing your own profiler (by implementing IProfiler) for Quasar, you are likely going to need this function.
- FindType: looks up type information for the specified type name.

Other functions are not intended to be used directly from user-code.

Evaluation Stack interface *(IEvaluationStack)*

The evaluation stack acts like a standard stack onto which values can be pushed. Please make sure that, when pushing objects that require memory allocation/deallocation, the PushRef(.) functions are used.

```
public interface IEvaluationStack
{
    QValue PopValue();
    void PushValue(QValue value);
    QValue Pop();
    QValue[] PopValue(int count);
    void Push(QValue value);
    void Push(double value);
    void Push(string value);
    void Push(QValue[] value);
    void PushRef(QValue value);
    void PushRef(QValue[] value);
    void Clear();
    void PushContext();
    void PopContext();
    QValue Evaluate(QValue value);
    int Count { get; }
}
```

Matrix factory *(IMatrixFactory)*

The matrix factory is needed for constructing matrices on the fly. Whenever possible, the Quasar library should just use the new method, which accepts a .NET array, together with the dimensions of the matrix. The other functions are there for flexibility and efficiency reasons. One subtle issue, is that the element type of the Array needs to be equal to IComputationEngine .FltType, otherwise runtime errors will likely be generated.

```
    public interface IMatrixFactory
    {
        Matrix New(AllocationFlags flags, Array elems, params int[] dims);
        Matrix New(AllocationFlags flags, Array elems, ElemMaxDims elemMaxDims,
                  params int[] dims);
        Matrix Alloc(AllocationFlags flags, Type elemType, params int[] dims);
        Matrix Alloc(AllocationFlags flags, Type elemType, ElemMaxDims elemMaxDims,
                    params int[] dims); // Allocate, without zero initialization
        Matrix Linspace(AllocationFlags flags, Type elemType, double start, double end,
                       int count);
        Matrix Zeros(AllocationFlags flags, Type elemType, params int[] dims);
        Matrix Zeros(AllocationFlags flags, Type elemType, ElemMaxDims elemMaxDims,
                    params int[] dims);
        Matrix Ones(AllocationFlags flags, Type elemType, params int[] dims);
        Matrix ConstantMatrix(AllocationFlags flags, Type elemType, double value,
                             params int[] dims);
        Matrix IdentityMatrix(AllocationFlags flags, Type elemType, int rows);
        Matrix Rand(AllocationFlags flags, Type elemType, params int[] dims);
        Matrix Randn(AllocationFlags flags, Type elemType, params int[] dims);
        ComplexMatrix Complex(AllocationFlags flags, Array elems, params int[] dims);
        ComplexMatrix Complex(Matrix re);
        ComplexMatrix Complex(Matrix re, Matrix im);
        ComplexMatrix ConstantComplexMatrix(AllocationFlags flags, Type elemType, Complex value,
                                           params int[] dims);
        ComplexMatrix CAlloc(AllocationFlags flags, Type elemType,
                            params int[] dim);
    }
```

The description of the functions is as follows:

Function name	Description
New	Creates a matrix based on the specified .Net array.
Alloc	Allocates a matrix with the specified dimensions and element type, but does not initialize it (the values are undetermined).
Linspace	Creates a linearly spaced vector with minimum start and maximum and of length 'count'.
Zeros	Allocates a matrix with the specified dimensions and element type, initialized with zeros.
Ones	Allocates a matrix with the specified dimensions and element type, initialized with ones.
ConstantMatrix	Allocates a matrix with the specified dimensions and element type, initialized with the specified value.
IdentityMatrix	Creates an identity square matrix of dimensions rows x rows
Rand	Generates a uniform random noise matrix with specified dimensions (values are between 0 and 1).
Randn	Generates a Gaussian random noise matrix with specified dimensions (with mean 0 and standard deviation 1).
Complex	Generates a complex-valued matrix.
ConstantComplexMatrix	Generates a complex-valued matrix with a constant value.
CAlloc	Allocates a complex-valued matrix with the specified dimensions and element type, but does not initialize it (the values are undetermined).

The functions Alloc and CAlloc only allocate memory, without initializing it, like in C malloc. Only use this function when it is certain that all values of the matrix will be initialized in a next step. For example, when the goal is to calculate the sum of two matrices, it is not necessary to initialize the end result with zeros. Using the function Alloc (or CAlloc) can in this case bring a small performance improvement compared to Zeros (or ConstantComplexMatrix).

A standard programming pattern for generating a matrix is:

```
Matrix A;
if (ce.FltType == typeof(float))
{
    float[] flt_data = new float[numel];
    for (int i = 0; i < numel; i ++)
        flt_data[i] = func(i);
    A = ce.MatrixFactory.New(AllocationFlags.None, flt_data, 1, numel);
}
else if (ce.FltType == typeof(double))
{
    double[] dbl_data = new double[numel];
    for (int i = 0; i < numel; i ++)
        dbl_data[i] = func(i)
    A = ce.MatrixFactory.New(AllocationFlags.None, dbl_data, 1, numel);
}
else throw new NotSupportedException();
```

Here, ce.FltType is the currently selected floating point type (either typeof(float) or typeof(double)).

Note that the array passed to IMatrixFactory .New always needs to be a one-dimensional .Net array. In case higher dimensional arrays are passed, an exception is generated.

Another convention is that vectors should be row vectors by default (hence the dimensions are [1, numel], rather than [numel, 1], which would correspond to a column vector).

Matrix allocation flags *(AllocationFlags)*

When creating matrices, allocation flags need to be specified. These flags are as follows:

```
public enum AllocationFlags
{
    None = 0,
    ApplyUserAttributes = 1,
    ForceGPUTarget = 2,
    ForceCPUTarget = 4
}
```

With None, the storage is determined automatically by the run-time. ForceGPUTarget enforces the matrix allocation to take place on an available GPU. With ForceCPUTarget, CPU memory will be allocated. Finally, the option ApplyUserAttributes allows user code to override the behaviour. For example, it is possible that the user specifies the allocation mode in Quasar code, using the code attributes:

```
    {!alloc mode=cpu} % Allocation takes place on the CPU
    {!alloc mode=gpu} % Allocation takes place on the GPU
```

The main purpose is to gain a little bit of efficiency and to avoid zero memory to be copied unnecessarily between CPU and GPU.

Generic matrix construction

The following example illustrates how a generic matrix of a given type can be constructed, where the content of the matrix is initialized using a double array.

```
private Matrix GenerateGenericMatrix<T>(IMatrixFactory factory, double[] elems, params int[] dims)
{
    return factory.New(AllocationFlags.None, elems.Cast<T>(), dims);
}
```

The method Cast<T>() is an extension method that can be accessed by adding using Quasar.Computations.Math; to the top of the C# code file. This method will cast every element of the array elems to the specified type. For example:

```
Matrix intMatrix = GenerateGenericMatrix<int>(factory, new double[] { 1.0, 2.0, 3.0 }, 1, 3);
```

will create an integer matrix and initialize it with the values {1, 2, 3}.

Boundary access modes *(BoundaryAccessMode)*

Boundary access modes are used by the functions ArrayGetAt, ArrayGetSliceAt, ArraySetAt, ArraySetSliceAt of the IComputationEngine interface. The Quasar compiler sets the access modes automatically based on the corresponding variable modifiers. If you intend to access matrix elements from your own .NET code, you will have to specify the access mode manually.

```
public enum BoundaryAccessMode
{
    Default = 0, // Raise an exception (checked)
    Unchecked = 1, // Fastest option - no checking
    Zero = 2, // Boundary extension with zeros
    Circular = 3, // Circular extension
    Mirror = 4, // Mirrored extension
    Clamp = 5 // Clamps to the border value
}
```

Operators *(OperatorTypes)*

Below, we list the main operators in Quasar that are currently defined. Note that the computation engine may not necessarily implement *all* operators for *all* possible input data types. In case an operator is not implemented, an error (exception) will be generated.

Some operators, such as OP_PIPELINE_RIGHT and OP_PIPELINE_LEFT are only defined through reductions. Note that these reductions can be inserted at run-time using the function IComputationEngine.AddReduction(.).

```
public enum OperatorTypes : byte
{
    OP_ADD, // addition
    OP_SUB, // subtraction
    OP_MULTIPLY, // multiplication (matrix multiplication)
    OP_DIVIDE, // divide
    OP_RDIVIDE, // right division
    OP_POW, // power
    OP_PW_MULTIPLY, // point-wise multiplication
    OP_PW_DIVIDE, // point-wise division
    OP_PW_POW, // point-wise power
    OP_LESS, // less than
    OP_LESSOREQ, // less than or equal
    OP_GREATER, // greater
    OP_GREATEROREQ, // greater or equal
    OP_EQUAL, // equal
    OP_NOTEQUAL, // not equal
    OP_ASSIGN, // assignment
    OP_NEGATE, // negation (-)
    OP_INVERT, // logical inversion (!)
    OP_LOG_AND, // logical AND
    OP_LOG_OR, // logical or
    OP_DOTDOT, // sequence (a..b)
    OP_DOTDOTDOT, // sequence with step (a..b..c)
    OP_LAMBDADEFSTART, // start of a lambda expression ->
    OP_LAMBDADEFEND, // end of a lambda expression
    OP_ADD_ASSIGN, // inplace assignment +=
    OP_SUB_ASSIGN, // inplace subtraction -=
    OP_MULTIPLY_ASSIGN, // inplace multiplication *=
    OP_DIVIDE_ASSIGN, // inplace division /=
    OP_POW_ASSIGN, // inplace power ^=
    OP_COND_IF, // conditional IF ? :
    OP_SWITCH_CASE, // switch
    OP_DEF_ASSIGN, // reserved operator (:=)
    OP_PIPELINE_RIGHT, // reserved operator |>
    OP_PIPELINE_LEFT, // reserved operator <|
    OP_LAZYEXPR_START, // reserved
    OP_LAZYEXPR_END, // reserved
    OP_WHERE // reserved operator 'where'
}
```

Class interface

Exporting .Net classes to Quasar

.Net classes can be exported to Quasar, so that they can be used from within user programs. To this end, it is necessary to annotate the .Net class with the ClassVisible attribute (defined in Quasar.Computations.QObject). Furthermore, the class should inherit Quasar.Computations.QObject, as in the following C# example.

```
using Quasar.Computations;

[ClassVisible("quasarClassName")]
class MyClass : QObject
{
    [MemberVisible]
    public int value;

    [MemberVisible]
    public void function()
    {

    }
}
```

Next, class fields, properties and methods with the [MemberVisible] attribute set will be automatically accessible from Quasar functions (at least when the parameter type translation is possible, see the next Section).

Note that for properties, it is always required that both the *get* and the *set* properties are implemented, otherwise a run-time exception is generated. (this behavior may change in the future)

QObject inherits System.Dynamic.DynamicObject, which means that the classes can be used dynamically from C#:

```
dynamic obj = new MyClass();
obj.value = 4;
obj.function();
```

Importing Quasar classes from .Net

Similarly, classes defined in Quasar can be accessed from .Net. Use ce.FindType, as in the following example:

```
% Quasar code:
type myclass : dynamic class
    value : int
end
```

```
// C# code
QTypeInfo userTypeInfo = ce.FindType("myclass", true);
dynamic obj = userTypeInfo.GetTypedObjectInfo().CreateInstance<QObject>();
obj.value = 4;

// Equivalent C# code
QTypeInfo userTypeInfo = ce.FindType("myclass", true);
QObject obj = userTypeInfo.GetTypedObjectInfo().CreateInstance<QObject>();
obj["value"] = 4;
```

Important to know is that *dynamic classes* (i.e., Python-type classes in which fields can be added at run-time) inherit from QObject, while *static classes* inherit from QTypedObject.

It is also possible to define static classes from within C#. The following example illustrates how this can be achieved:

```
// (Optionally) creation of the computation engine
TypeEnvironment typeEnv = new TypeEnvironment();
RuntimeSettings runtimeSettings = new RuntimeSettings()
{
    TypeEnvironment = typeEnv
};
IComputationEngine ce = RuntimeSystem.CreateComputationEngine(
                         RuntimeSystem.ComputationPreference.GPU, runtimeSettings);

// Define the type 'point'
TypedObjectInfo point = new Quasar.TypeSystem.TypedObjectInfo("quasar", "point");
point.AddField(new TypedFieldInfo("x", QTypeInfo.Scalar));
point.AddField(new TypedFieldInfo("y", QTypeInfo.Scalar));
ce.FinalizeType(point, null);

// Define the type 'myType'
TypedObjectInfo myType = new Quasar.TypeSystem.TypedObjectInfo("quasar", "myType");
myType.AddField(new TypedFieldInfo("A", QTypeInfo.Cscalar));
myType.AddField(new TypedFieldInfo("B", QTypeInfo.Scalar));
myType.AddField(new TypedFieldInfo("P", point.GetInstanceType()));
myType.AddField(new TypedFieldInfo("Pstar",
                                point.GetInstanceType().GetPointerReferenceType()));
ce.FinalizeType(myType, null);

QTypedObject obj = myType.CreateInstance<QTypedObject>(true);
obj.SetValue(ce, "A", new Computations.Math.Complex(1, 2));

QTypedObject obj2 = myType.CreateInstance<QTypedObject>(true);
obj2.SetValue(ce, "A", new Computations.Math.Complex(2, 3));

QValue value = obj.GetValue(ce, "A");
```

This requires significantly more boilerplate code, as you can see. The reason is that each static type *needs to be mapped* onto device-specific containers (e.g. native C++ types, GPU types), for which each can have a different data layout. The responsibility of the mapping lies with the run-time. This way, the .Net library can be writting in a device-independent way without having to rely on certain structure sizing/packing requirements for a given target architecture.

Method call interface

There are two ways to define .Net methods that can be accessed from Quasar:

1. Methods with variadic arguments (i.e., variable number of arguments of arbitrary types). Use the following signature:

```
QValue MyFunction(params QValue[] args)
```

2. Methods with fixed argument

```
QValue MyFunction(int arg1, string arg2, QValue arg3)
```

In the second case, Quasar will perform automatic conversion (and type checking) to the destination type. If the conversion is not possible, an exception is generated. Not all .Net datatypes can be mapped onto Quasar types. The following table lists the mappings that have been defined

Type	.Net data type	Quasar data type
Any type	Quasar.Computations.QValue	??
Floating point type	float, double	scalar
Complex number	Quasar.Computations.Math.Complex	cscalar
String	System.String (string)	string
Boolean	System.Boolean (bool)	scalar
Integer	System.Int32 (int)	int
Vectors/matrices/cubes	Quasar.Computations.Math.Matrix	mat
Complex vectors/matrices/cubes	Quasar.Computations.Math.ComplexMatrix	cube
Functions/lambda expressions	Quasar.Computations.Math.QLambdaExpression	function

In some cases, the .Net data type cannot fully express the Quasar type information. In this case, it is possible to help the parameter type translator by specifying the Quasar type, using the ArgType attribute. This is done as follows:

```
[return: ArgType("vec")]
Matrix MyFunction([ArgType("cube[int]")] Matrix arg4)
```

In the above example, a Quasar function of type [cube[int] −> vec] is defined.

Note: multiple return values can also be returned from Quasar functions. To this end, create a cell vector (using the function ce.MatrixFactory.Create), in which each return value is stored in a different index. Example for a function returning MFNs, FXs and MYs:

```
QValue[] res = new QValue[]{
    MFNs,
    MXs,
    MYs
};
QValue argout = ce.MatrixFactory.New(AllocationFlags.None,res,1,res.Length);

return argout;
```

Documentation attributes

.Net classes can be annotated with a documentation category. The documentation category is used by the Redshift documentation browser to identify the location in which the documentation file for the class is placed:

```
[Category("category")]
```

Currently, the following categories have been defined:

```
* Algorithms
* General
* Image Processing
* Image Processing/Colors
* Image Processing/Demosaicing
* Image Processing/Edge detection
* Image Processing/Feature detection
* Image Processing/Features
* Image Processing/Filters
* Image Processing/HDR
* Image Processing/Histogram
* Image Processing/Interpolation
* Image Processing/Mathematical morphology
* Image Processing/Multiresolution
* Image Processing/Pattern matching
* Image Processing/Restoration
* Linear algebra
* Mathematics
* Medical image Processing
* Parallel Programming
* Statistics
* User interfaces
* Visualization
```

Additionally, a description attribute (System.ComponentModel.DescriptionAttribute) can be added to methods/functions, as follows:

```
[Description("description")]
```

An example description is given below:

```
[Description("Function: fopen\n\n" +
            "Opens the specified file for reading or writing. The file handle\n" +
            "must be closed using <fclose>\n\n" +
            ":function file_handle = fopen(filename : string, options)\n\n" +
            "Parameters:\n" +
            "filename - the path of the file to open\n" +
            "options - a string containing the options for loading:\n" +
            "o \"w\": open for writing\n" +
            "o \"r\": open for reading\n" +
            "o \"a\": open for appending\n" +
            "o \"b\": binary mode (for reading/writing binary data)\n" +
            "o \"t\": text mode (for reading/writing text)\n\n" +
            "See also:\n" +
            "<fclose>, <fread>, <fwrite>, <fprint>, <fprintf>\n\n" )]
public static QValue fopen(QValue fileName, QValue options)
```

It uses the Natural Docs (`http://www.naturaldocs.org/`) documentation. Note that in .Net, the line breaks need to be inserted with \n. Alternatively, C# multi-line strings can be used.

Callback events

In some cases (e.g., GUI interaction), it is necessary to have events as a callback mechanism. The idea is that Quasar user-code can be notified of certain events (such as a window being moved, maximized, etc.). This can be accomplished using the Quasar.Computations.Runtime.Event class:

```
using Quasar.Computations.Runtime;

[MemberVisible,
 Description("Event called when the form is moved by the user")]
public Event onmove = new Event();
```

The event class has the methods add, remove and fire :

```
[ClassVisible("qevent"),
 Category("User interfaces"),
 Description(@"A message sent by a GUI control to indicate a certain action
      (for example when the user clicks on a button")]
public class Event : QObject
{
    public void add(QLambdaExpression action);
    public void remove(QLambdaExpression action);
    public void fire(params QValue[] args);
}
```

The add method registers a specified lambda expression for callback. This function is typically called from Quasar code. The remove method unregisters a lambda expression (undoing the add operation). Next, events can be fired using the fire method. The fire method will call each of the registered event handlers using the specified parameters

```
if (onmove != null)
    onmove.fire(my_params);
```

Finally, it is important to realize that events, just like QObject-instances, are reference-counted. Therefore, it is necessary to release all events, in the Dispose function of the class (in case of implementing .Net's IDisposable interface). When the reference count of the event is zero, all reference handlers are released, so that, e.g., all implicit memory objects that referred to from the event handler (such as closure variables) are released as well.

```
void Dispose()
{
    if (onmove != null)
    {
        onmove.Release();
        onmove = null;
    }
}
```

Example

Below, a more extended example is given of the Quasar-to-.Net class interface:

```
using Quasar.Computations.Math;

[ClassVisible("qmyclass"),
 Category("Documentation category"),
 Description("This class can be used for any purpose.")]
public class MyClass : QObject
{
    [MemberVisible, Description("An integer value")]
    public int value;

    [MemberVisible, Description("An automatic property")]
    public string autoProperty { get; set; }

    [MemberVisible("vec[string]"), Description("A list of strings")]
    public Matrix stringList;

    [MemberVisible, Description("Processes the input values passed to this function")]
    public void Process([ArgType("cube[vec]")] Matrix)
    {
        ...
    }

    [MemberVisible]
    [return: ArgType("cube[uint8]")]
    public Matrix Rasterize()
    {
        ...
    }

    [MemberVisible,
     Description("Event called when the form is moved by the user")]
    public Event onmove = new Event();
}
```

Example F# program

Below I give an example of a standalone F# program (F# is a functional programming language that is derived from OCaml, but that is distributed with Microsoft Visual Studio 2012, so it has excellent IDE and debugging support).

F# - because Quasar seems to (more or less) naturally integrate in it. Some wrapper functions are required for dispatching the calls to the current computation engine.

The standalone F# program is a direct translation of the following Quasar program:

```
x = [1,2,3,4]
y = [5,6,7,8]
print x + y
print 4.0 * x
im = imread("lena_big.tif")
im_out = 255-im
imshow(im_out)
title("result")
```

This also gives a good idea of how the Quasar interpreter/EXE compiler work internally.

```
open System
open Quasar
open Quasar.TypeSystem
open Quasar.Computations
open Quasar.Computations.Runtime

let mutable engine : IComputationEngine = null
let mutable stack : IEvaluationStack = null

// Conversion to QValue
let inline (!>) (x:^a) : ^b = ((^a or ^b) : (static member op_Implicit : ^a -> ^b) x)

// Creating a vector
let vec ([<ParamArray>] arr : 'a array) =
    !> engine.MatrixFactory.New(arr, 1, arr.Length)

// Adding two values
let inline (+) (x : QValue) (y : QValue) =
    stack.PushRef(x)
    stack.PushRef(y)
    engine.Process(Compiler.OperatorTypes.OP_ADD)
    stack.PopValue()

// Subtracting two values
let inline (-) (x : QValue) (y : QValue) =
    stack.PushRef(x)
    stack.PushRef(y)
    engine.Process(Compiler.OperatorTypes.OP_SUB)
    stack.PopValue()

// Multiplying two values
let inline (*) (x : QValue) (y : QValue) =
    stack.PushRef(x)
    stack.PushRef(y)
    engine.Process(Compiler.OperatorTypes.OP_MULTIPLY)
    stack.PopValue()

let sin (x : QValue) =
    stack.PushRef(x)
    engine.FunctionCall("sin", 1)
    stack.PopValue()

let imread (x : string) =
    stack.Push(!> x)
    engine.FunctionCall("imread", 1)
    stack.PopValue()

let imshow (x : QValue) =
    stack.PushRef(x)
    engine.FunctionCall("imshow", 1)

let title (x : string) =
    stack.Push(!> x)
    engine.FunctionCall("title", 1)

//
// Quasar initialization
//
let qinit =
    RuntimeSystem.Initialize() // Initialize the runtime system

    let runtimeSettings = new RuntimeSettings()
    runtimeSettings.TypeEnvironment <- new TypeEnvironment()
    runtimeSettings.TypeEnvironment.scalarType <- ScalarTypes.Single

    // Create the computation engine
    engine <- RuntimeSystem.CreateComputationEngine(
        RuntimeSystem.ComputationPreference.GPU, runtimeSettings)
    stack <- engine.EvaluationStack

    // Initialize the core library
    CoreLibrary.run(engine, engine.EvaluationStack)

//
// Demo program
//
let demo =
    let x = vec([|1; 2; 3; 4|])
    let y = vec([|5; 6; 7; 8|])
    printf "%A\n" (x + y)
    printf "%A\n" (sin ((!> 4.0) * x))

    let im = imread("lena_big.tif")
    let im_out = !> 255.0 - im
    imshow im_out
    title "Result"
    RuntimeSystem.WaitTillFinished()

[<EntryPoint>]
let main argv =
    qinit
    demo
    printf "Success!"
    let key = Console.ReadKey()
    0 // return an integer exit code
```