Introduction

The PyQuasar library functions as a bidirectional bridge between Python and Quasar: it allows Quasar functions to be used from Python (by serving as a Python class extension) and simultaneously allows Python functions to be used from Quasar (by serving as a Quasar external library).

The integration brings several advantages:

1. Quasar becomes an alternative for NumPy, but with GPU/multi-core acceleration and automatic runtime scheduling
2. the Quasar libraries are extended by many available Python libraries, increasing the developer productivity. This is possible due to the dynamic character of the two languages.

Note that there is a small (but generally, negligible) overhead involved by crossing the Python-Quasar bridge, but this overhead relatively becomes zero if the task is heavy enough. Therefore, for most users this should not form any problem (computation intensive tasks are better written in the form of compute kernels anyway).

PyQuasar also provide a means to access compiled Quasar libraries (either compiled as .Net libraries or as native libraries) to be used from Python. In existing Python code bases, this allows Quasar to be used as an accelerator of certain critical parts of the code.

PyQuasar is designed with easiness of use in mind, while avoiding unnecessary conversions between Python objects and Quasar objects. By default PyQuasar will wrap Quasar objects into wrapper objects that are then passed to Python, and vice versa. When a conversion is required, an explicit method needs to be called (e.g., to_py() to convert to Python, from_py() to convert from Python, see below.)

Therefore as a library on its own PyQuasar adds significant functionality to both languages, simplifying development and making it easy to integrate Python code with Quasar code.

Installation

PyQuasar is part of the Quasar installation. You can find the following libraries in the OS_Runtimes directory:

OS	File	Purpose
Windows	OS_Runtimes\Windows_x64\pyquasar.pyd	Python -> Quasar bridge
Windows	OS_Runtimes\Windows_x64\pyquasar.64.dll	Quasar -> Python bridge
Ubuntu	OS_Runtimes/Ubuntu_x64/pyquasar.64.so	Python -> Quasar bridge
Ubuntu	OS_Runtimes/Ubuntu_x64/pyquasar.so	Quasar -> Python bridge

Note that shared libraries are provided for both directions. However, the files are the same! This is because Python and Quasar have different conventions for the file names of the libraries.

Important the version of Python to be used is fixed and depends on the OS

OS	Python version	Recommended installation
Windows	3.7	Anaconda with Python 3.7
Ubuntu	3.6	Ubuntu package manager (apt-get)

In case multiple Python versions are installed, Python virtual environments need to be used. In Windows, the Python environment can be selected in the Preferences dialog box. In Ubuntu, set your environment variables PYTHONPATH, PATH and LD_LIBRARY_PATH before starting Redshift (see, e.g., the virtualenvwrapper utility).

Part 1: Accessing Quasar functions from Python

Functions in Quasar programs can be called from Python. There are in fact two ways of loading Quasar programs: 1) either by compiling from source code or 2) by loading a pre-compiled library (.dll or .so). During initial development it may be useful to directly compile the Quasar source code from Python; when the code is more stable and less changes are being performed, the user may compile the code resulting in a dynamic link library, which results in faster loading and execution times.

PyQuasar can be loaded by simply by the following code

```
import pyquasar as q

q.init("cuda") # required (specifying the runtime engine and compute device)
```

The above code will initialize the Quasar runtime system using the default CUDA device. In case Quasar code needs to be compiled on the fly (either by loading a Quasar source code module via compile_file, or by compiling a Python string via compile_string), it is also necessary to specify that the compiler needs to be loaded:

```
q.init("cuda",loadCompiler=True)
```

The following runtime engines are currently available:

Init parameter	Device	Quasar runtime
cpu	Default CPU device	MONO/.Net Quasar runtime
cuda	Default CUDA device	MONO/.Net Quasar runtime
some_file.xml	Hyperion device configuration	MONO/.Net Quasar runtime
helios:cpu	Default CPU device	helios (µQuasar) runtime
helios:cuda	Default CUDA device	helios (µQuasar) runtime

It is mandatory to call the init function. If not called, any other operation will result in an error.

PyQuasar exposes the following functions to Quasar:

Function	Purpose
init	Initializes the Quasar host
run_gui	Runs the Quasar GUI message loop and waits until all forms are closed
import_native	Imports a natively compiled Quasar library
import_dll	Imports a managed Quasar library (e.g., Quasar.UI.dll, Quasar.DNN.dll, ...)
compile_string	Compiles and imports a Quasar program from a source code string
compile_file	Compiles and imports a Quasar program from a file
typename	Parses a Quasar type name from a string

Both native and managed libraries are supported. Note however, that since µQuasar! is entirely native, onlysupported managed libraries can be imported. The following libraries are supported: Quasar.UI.dll, Quasar.DNN.dll, Quasar.Video.dll, Quasar

.CompMath.dll. For the MONO/.Net Quasar runtime this restriction does not exist. Custom Quasar libraries can always be compiled to native libraries. These libraries can be loaded with all runtimes.

Often, it is useful to parse Quasar types from Python. This is useful for example when constructing Quasar vectors or matrices:

```
w = q.typename("vec[mat]")(4) # in Python
```

is equivalent to

```
w = vec[mat](4) # In Quasar
```

Difference between native and managed Quasar libraries

The following table outlines some differences between native and managed Quasar libraries:

Type	Works across platforms	Works with Quasar	Works with μQuasar	C++ source code available
Managed library	Yes	Yes	No	No
Native Library	No (requires recompilation)	Yes	Yes	Yes

In general, there is no fixed or preferred choice here - depending on the toolchain (C++ based or .Net based) one may choose a managed library or native library.

pyquasar.qvalue Data type

Quasar values (vectors, matrices, cubes, strings, functions, ...) are all seen as instances of the class pyquasar.qvalue in Python. Functions are wrapped to that they can be called directly from Python. For objects, all methods and fields of the underlying Quasar type are exposed to Python. For vectors, matrices, ... binary operators are mapped automatically. For example:

Supported operation	Purpose
Unary operator (-), (!)	allows operations to be performed directly on Quasar values
Binary operators (+), (-)	allows operations to be performed directly on Quasar values
Subscription A [:,2,3]	allows matrix slicing or element selection directly on the Quasar value
to_py()	converts a vector/matrix/... to a NumPy array
lock ()	gets access to raw CPU or GPU memory pointer
unlock()	indicates that access is no longer needed

In particular, the to_py() method was added to easily convert Quasar vectors/matrices to NumPy arrays. By default, Quasar values are passed to Python without any conversion, which has the advantage that e.g., the array memory stays in the GPU memory. In some cases, it is desirable to pass Quasar values to NumPy functions, for this purpose to_py() is very useful.

A cube (3D array) can be constructed simply as follows:

```
A = q.zeros(20,20,3) # in Python
```

is equivalent to:

```
A = zeros(20,20,3) # in Quasar
```

The values can be then passed to any Quasar function that is exposed to Python. For example:

```
q.print(A) # calls the Quasar print function
q.parallel_do(..., A, ...) # invokes a parallel_do operation on ‘A‘
```

In addition, the value can be printed using the Python print function.

Locking and raw data pointer access

For interoperability with other matrix or tensor libraries (e.g., TensorFlow, PyTorch) it is often useful to have access to the raw data associated with Quasar objects. For PyTorch, this allows for example converting a Quasar matrix to a PyTorch Tensor object by directly copying the content of the data on the GPU. This approach avoids the time costly transfer between CPU and GPU. To obtain a raw pointer for a pyquasar.qvalue object, the methods lock() and unlock() can be used:

```
    def copy_data_to_dst_pointer(A, dst_ptr, num_bytes):
       ptr = A.lock(q.LOCK_READ, q.MEMRESOURCE_SINGLE_CUDA)
    cuda.memcpy_dtod(dst_ptr, ptr, num_bytes)
    A.unlock(q.LOCK_READ, q.MEMRESOURCE_SINGLE_CUDA)
```

lock() will lock the data for a specific device. Depending on the locking mode, other compute devices cannot simultaneously read/write from A within the locking region. It is important to always call unlock() when the operation is completed. The data pointer is only guaranteed to be valid within the locking region (outside the locking region, the runtime system preserves the right to move memory blocks, e.g., in other to reduce memory fragmentation, or to transfer memory blocks back to the CPU when insufficient GPU memory is available). The following locking modes are available:

Locking mode	Purpose
LOCK_READ	The resource is only used for reading
LOCK_WRITE	The resource is only used for writing
LOCK_READWRITE	The resource is used for both reading and writing

It is important that the correct locking mode is specified: this avoids unnecessary memory transfers between CPU and GPU (or between GPUs pairwise in case of multiple GPUs). The following devices can be specified:

Device	Purpose
MEMRESOURCE_CPU	Memory resource for the main CPU device

Device	Purpose
MEMRESOURCE_SINGLE_CUDA	Memory resource for the first CUDA device
MEMRESOURCE_SINGLE_OPENCL	Memory resource for the current OpenCL device
MEMRESOURCE_DEVICE0	Configuration-dependent device #0
MEMRESOURCE_DEVICE1	Configuration-dependent device #1
MEMRESOURCE_DEVICE2	Configuration-dependent device #2
MEMRESOURCE_DEVICE3	Configuration-dependent device #3
MEMRESOURCE_DEVICE4	Configuration-dependent device #4

Devices MEMRESOURCE_DEVICEi are particularly useful when multiple GPUs are used; however, currently, PyQuasar does not provide a way yet to query the devices. So it is not (yet) possible to determine the device type for each device. However, for a multi-GPU configuration, the device numbering can be assumed to be as follows:

Device	Purpose
MEMRESOURCE_DEVICE1	CPU device
MEMRESOURCE_DEVICE2	CUDA device 0
MEMRESOURCE_DEVICE3	CUDA device 1
MEMRESOURCE_DEVICE4	CUDA device 2

Devices MEMRESOURCE_DEVICE0 can be OR'ed or added to obtain explicit numbers higher than 4. For example, device #4 would be MEMRESOURCE_DEVICE0+5.

Examples

The following examples show how Quasar functions and objects can be used from Python. You can find some more examples in the PyQuasar repository.

Example 1: from Python->Quasar

In the following example, two vectors are defined in Quasar and added. Finally, the result is converted to a NumPy array.

```
import numpy as np
import pyquasar as q

q.init("cpu") % Initializes Quasar using the CPU device
a = q.value([5,4,3,2,1]) + q.value(2.0)
print(a.to_py())
```

Example 2: creating a simple Quasar GUI from Python->Quasar

The Quasar user interface library can be imported via q. import_dll ("Quasar.UI. dll "). Essentially all classes and methods from this library are accessible from Python. The following example demonstrates how to create a simple window with a button that can be clicked.

```
import pyquasar as q

q.init("cpu")
q.import_dll("Quasar.UI.dll")

def on_click():
    print("Clicked")

frm = q.form("Quasar form")
frm.width = 800
frm.height = 600
button = frm.add_button("Click me")
button.onclick.add(on_click)

frm.wait() # Wait until the form is closed
```

Example 3: Launching a Quasar kernel from Python

The function compile_string can be used to directly compile Quasar source code from Python. The compiled code is then imported as a module (just like if import_native or import_dll was called). Then the Python program can access the variables and functions of the Quasar sub-program. The following example demonstrates how to launch a Quasar kernel on the GPU, from Python.

Note that compiling Quasar source code, requires the compiler to be loaded. This can be achieved by specifying the flag load_compiler=1 in init (). The Quasar compiler is only available for the MONO/.Net Quasar runtime (not for Helios).

```
import pyquasar as q

kernel_src = """
lerp = __device__ (a,b,c) -> a*c+b*(1-c)

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
endfunction
""";

q.init("cuda", load_compiler=1) # 1 to load the compiler (note: libhelios does not provide a compiler)
q.compile_string("color_temperature", kernel_src)

r = range(2)
print(dir(r))

def apply(img_in, temp):
    hot = q.value([1,0.2,0]) * 255
    cold = q.value([0.3,0.4,1]) * 255
    img_out = q.zeros(q.size(img_in))
    q.parallel_do(q.size(img_out,[0,1]),img_in, img_out, temp, cold, hot, q.color_temperature)
    return img_out

img_in = q.imread("lena_big.tif")
temp = 1
img_out = apply(img_in, temp)
q.imshow(img_out)
q.run_gui()
```

Dynamically compiling code on the fly is useful for rapid prototyping. For production-level code, it is best that the Quasar code is compiled to a native binary, which can then be loaded using import_native.

Part 2: Accessing Python functions from Quasar

This part describes how Python functions can be called from Quasar. This allows the Quasar programmer to access the entire Python eco-system and libraries. The integration of Python in Quasar is therefore complete, in the sense that:

- Quasar modules that use Python functionality can be compiled to a native library
- The native library can be integrated in a hosting executable, or even loaded from another Python script.
- It is possible to write a Python script that calls a Quasar function that on its turn uses Python functions.

The unnecessary communication/translation (called interoperability) is hence two-directional and happens behind the scenes.

The following example shows how to load the PyQuasar library from Quasar.

```
import "pyquasar.${BITNESS}.${NATIVEEXT}"

py = pyimport("builtins")
np = pyimport("numpy")

reduction x-> x.T = np.transpose(x)

v = np.array([1,2,3]) % v has shape (3,)
w = np.array([4,5]) % w has shape (2,)

py.print("v",v,sep:="=")
```

The use of parameters in pyquasar.${BITNESS}.${NATIVEEXT} is required to write architecture and platform-independent code. For example, in linux the string will expand to pyquasar.64.so while in windows it becomes pyquasar.64.dll. It is possible to import directly pyquasar.64.so but then the code is limited to work on one operating system.

Also note that Quasar does not have the .T syntax, but it can be defined in one line using a reduction.

pyvalue Data type

All Python objects are seen as instances of the class pyvalue in Quasar. Through this class, Python functions can be called and methods of Python objects are exposed. In addition, unary and binary operators are mapped onto Python, so that NumPy arrays can be added, subtracted from Quasar.

Supported operation	Purpose
Unary operator (-), (!)	allows operations to be performed directly on NumPy arrays
Binary operators (+), (-)	allows operations to be performed directly on NumPy arrays
from_py()	converts a vector/matrix/… to a Quasar vector/matrix

Warning: np.reshape requires an integer vector as second argument. This can be achieved in Quasar by either constructing the vector using rounded brackets (e.g., {1,2}), or by converting the vector using the function int().

Auxiliary functions

Function	Purpose
obj . to_py ()	Converts an object to a Python object
obj . enter ()	Used as alternative for the with statement in Python
obj . leave ()	Used as alternative for the with statement in Python
pyint (val)	Converts a scalar value to a Python integer
pybool(val)	Converts a scalar value to a Python boolean
pyfunc(val)	Converts the Python object to a function that can be called from Quasar

to_py() method, pyint(), pybool() functions

Sometimes it is useful to invoke a Python method directly on a Quasar value. Using the to_py() method, in principle any Quasar object can be converted to a corresponding Python object onto which Python methods can be executed.

For example:

```
[1.0,2.0,3.0,4.0].to_py().astype(py.int)
```

Allows a scalar vector to be converted to an integer vector.

Alternatively, the pyint () function can be used for converting objects to Python integers. Since the Quasar interpreter only supports floating point numbers, the pyint () function can be used to convert scalars to a Python integer representation. The advantage is that very large numbers can be represented in Python's arbitrary precision integers. Idem for pybool() which converts scalar values to a Boolean representation.

pyfunc() function

A Python object that is passed to Quasar, is not automatically *callable* - Quasar objects currently don't support an operator (). In the future, this may change. Currently the function pyfunc can be used to indicate that the Python object will be used as a function.

Python tuples

In Quasar, tuples are represented by cell vectors/matrices (of type vec [??], vec[scalar], . . .). When passing cell vectors/matrices to Python, they will be passed as a NumPy array rather than a tuple (which a function might be expecting). To convert NumPy arrays to tuples, we can simply call the Python tuple function (py. tuple):

```
py.print(py.tuple({1,2,3,4}))
```

Python ‘with’ statement

In Python, the with statement allows resources to be disposed automatically, when they are no longer in use. For example, for a text file:

```
with open("x.txt") as f:
    data = f.read()
    #do something with data

    no_grad = torch.no_grad()
    no_grad.enter()
```

When the program exits the with block, the file handle will automatically be closed. Currently, Quasar does not have an equivalent statement (note however, that Quasar also automatically releases variables when they go out of scope, but it is not possible to create scopes similar to with in Python or {} in C or C++).

As an alternative, the methods enter and exit can be used. Both methods must always be used together.

```
f = py.open("x.txt")
f.enter()
data = f.read()
f.exit()
```

Example 1: from Quasar->Python

The following example shows how to use NumPy functions from Quasar.

```
import "pyquasar.${BITNESS}.${NATIVEEXT}"

py = pyimport("builtins")
np = pyimport("numpy")

reduction x-> x.T = np.transpose(x)

function [] = test1()
    v = np.array([1,2,3]) % v has shape (3,)
    w = np.array([4,5]) % w has shape (2,)

    % Keyword arguments are supported
    py.print("v",v,sep:="=")

    % To compute an outer product, we first reshape v to be a column
    % vector of shape (3, 1); we can then broadcast it against w to yield
    % an output of shape (3, 2), which is the outer product of v and w:
    % [[ 4 5]
    % [ 8 10]
    % [12 15]]
    py.print(np.reshape(v, pyint([3, 1])))
endfunction
```

Example 2: creating a Tkinder GUI from Quasar

The following example shows how to create a Tkinder GUI from within Quasar.

```
import "pyquasar.${BITNESS}.${NATIVEEXT}"

py = pyimport("builtins")
tk = pyimport("tkinter")
window = tk.Tk()
window.title("Quasar/Python TK test")
window.mainloop()
```

PyTorch integration

In this chapter, we explain how the popular deep learning library PyTorch can be integrated with Quasar. Note first of all that Quasar has its own deep learning libraries, like Quasar.DNN.dll for access to cuDNN functions and Quasar.CompMath.dll for differentiable programming. Nevertheless, it is useful to combine features from all mentioned libraries, to reduce the research or development time and to even increase the possibilities.

Since PyQuasar provides a general purpose Python<->Quasar bridge, actually no specific features are required to support PyTorch from Quasar. However, some utility functions - for example to copy GPU memory from Quasar directly to PyTorch - are useful.

The PyTorch integration works in two directions: PyTorch functions can be called from Quasar, Quasar functions can be called from PyTorch. These two options are discussed in the following subsections.

Accessing PyTorch functions from Quasar

The sample from “Learn PyTorch by example” (fitting of a two layer neural network to random data) uses the following Python code:

```
import torch

dtype = torch.float
device = torch.device("cuda")

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# Randomly initialize weights
w1 = torch.randn(D_in, H, device=device, dtype=dtype)
w2 = torch.randn(H, D_out, device=device, dtype=dtype)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum().item()
    if t % 100 == 99:
        print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # Update weights using gradient descent
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Below is an equivalent Quasar implementation. The mapping is relatively straightforward. Special attention needs to be paid to the following cases (also see previous sections): * Some python functions expect integer values or boolean values. By default, Quasar will passes these values as scalar values (64-bit floating point). To avoid this, use the pyint and pybool functions. * Quasar objects are currently non-callable. To convert a Python object to a function that can be called from Quasar, use pyfunc.

```
import "pyquasar.${BITNESS}.${NATIVEEXT}"

function [] = pytorch_sample()

    py = pyimport("builtins")
    np = pyimport("numpy")
    torch = pyimport("torch")
    nn = pyimport("torch.nn")

    dtype = torch.float32
    device = torch.device("cuda")

    [N, D_in, H, D_out] = [64, 1000, 100, 10]

    % Create random Tensors to hold input and outputs.
    x = torch.randn(N, D_in, device:=device, dtype:=dtype)
    y = torch.randn(N, D_out, device:=device, dtype:=dtype)

    % Create random Tensors for weights.
    w1 = torch.randn(D_in, H, device:=device, dtype:=dtype, requires_grad:=pybool(true))
    w2 = torch.randn(H, D_out, device:=device, dtype:=dtype, requires_grad:=pybool(true))

    learning_rate = 1e-6
    relu = pyfunc(nn.ReLU(inplace:=false))

    for t=1..500 % Iteration
        % Forward pass: compute predicted y using operations
        y_pred = relu(x.mm(w1)).mm(w2)

        % Compute and print loss
        loss = (y_pred - y).pow(2).sum()
        if mod(t, 100) == 99
            print t, ",", loss.item()
        endif

        % Use autograd to compute the backward pass
        loss.backward()

        % Update weights using gradient descent
        no_grad = torch.no_grad()
        no_grad.enter()
        w1 -= w1.grad * learning_rate
        w2 -= w2.grad * learning_rate
        no_grad.exit()
    endfor

endfunction
```

Accessing Quasar functions from PyTorch

Conversion from tensor to Quasar matrices and back

The following functions are useful to convert Quasar matrices to PyTorch tensors and back:

```
import torch
import pycuda.autoinit
import pycuda.driver as cuda
import pyquasar as q
import numpy as np

def to_tensor(A, device):
    """
    Converts a Quasar matrix to PyTorch Tensor
    """
    assert q.type(A, "cube{:}"), "A : cube{:} expected"

    if device.type=="cpu":
        tensor = torch.tensor((), dtype=torch.float32, device="cpu").new_zeros(tuple(q.size(A).to_py().
            astype(int)))
        storage = tensor.storage().cpu()
        ptr = A.lock(q.LOCK_READ, q.MEMRESOURCE_CPU)
        q.memcpy(storage.data_ptr(), ptr, storage.size() * storage.element_size())
        A.unlock(q.LOCK_READ, q.MEMRESOURCE_CPU)
    else:
        tensor = torch.tensor((), dtype=torch.float32, device=device).new_zeros(tuple(q.size(A).to_py()
            .astype(int)))
        storage = tensor.storage().cuda()
        ptr = A.lock(q.LOCK_READ, q.MEMRESOURCE_SINGLE_CUDA)
        cuda.memcpy_dtod(storage.data_ptr(), ptr, storage.size() * storage.element_size())
        A.unlock(q.LOCK_READ, q.MEMRESOURCE_SINGLE_CUDA)

    return tensor

def from_tensor(A):
    """
    Converts a PyTorch tensor to a Quasar matrix
    """
    assert torch.is_tensor(A) and A.dtype == torch.float32, "torch.FloatTensor expected!"

    mtx = q.uninit(list(A.size()))
    storage = A.storage()
    if storage.is_cuda:
        ptr = mtx.lock(q.LOCK_WRITE, q.MEMRESOURCE_SINGLE_CUDA)
        cuda.memcpy_dtod(ptr, storage.data_ptr(), storage.size() * storage.element_size())
        mtx.unlock(q.LOCK_WRITE, q.MEMRESOURCE_SINGLE_CUDA)
    else:
        storage = storage.cpu()
        ptr = mtx.lock(q.LOCK_WRITE, q.MEMRESOURCE_CPU)
        q.memcpy(ptr, storage.data_ptr(), storage.size() * storage.element_size())
        mtx.unlock(q.LOCK_WRITE, q.MEMRESOURCE_CPU)
    return mtx
```

The functions actually investigate if the data is stored in the CPU memory or GPU memory. In the former case the C function memcpy is used to copy the data on the CPU. In the latter case, the memory copy is performed on the GPU using pyCUDA.

PyTorch custom layer implementation using Quasar

The following example demonstrates how to implement a custom PyTorch layer using Quasar's differentiable programming concept. Practically, this means that the backward layer implementation is automatically derived from the forward layer implementation.

The Quasar code, which is very minimal in this example, is integrated directly into the Python script. As alternative (when compile_from_source==False), the Quasar code can be compiled to a .Net library mapping_layer.dll or even native library mapping_layer.64.so (see external interface reference).

The functions from_tensor and to_tensor are used to convert Quasar matrices to PyTorch tensors (see previous section).

```
import matplotlib.pyplot as plt
import torch
import pycuda.autoinit
import pycuda.driver as cuda
import pyquasar as q
import numpy as np
from pytorch_helpers import from_tensor, to_tensor
from torch.autograd import gradcheck

compile_from_source=True

if compile_from_source:
    quasar_src = """
    import "Quasar.CompMath.dll" % Differentiable programming support

    % Define the forward mapping function
    function y = mapping_forward(a : vec'clamped(256), x : cube)
        y = zeros(size(x))
        for [m,n,p]=0..size(x,0..2)-1
            y[m,n,p] = a[x[m,n,p]]
        endfor
    endfunction

    % The backward mapping is obtained by algorithmic differentiation
    mapping_backward = (a : vec'clamped(256), x : cube, da : cube) -> _
        $diffmul(mapping_forward(a, x), a, da)

    """
else:
    quasar_lib = "mapping_layer"

target_device = "cuda"

# Initialize Quasar
q.init(target_device,load_compiler=compile_from_source)
```

```
# Initialize PyTorch
dtype = torch.float32
device = torch.device(target_device) # Uncomment this to run on GPU


class Mapping(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weights):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input, weights)
        result = q.mapping_forward(from_tensor(weights), from_tensor(input))
        return to_tensor(result, device)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, weights, = ctx.saved_tensors
        assert ctx.needs_input_grad[1] and not ctx.needs_input_grad[0]
        dx = None

        da = q.mapping_backward(from_tensor(weights), from_tensor(input), from_tensor(grad_output))
        return dx, to_tensor(da, device)

if compile_from_source:
    # Compile the Quasar code
    q.compile_string("layer", quasar_src)
else:
    # Load the library
    q.import_native(quasar_lib)


# N is batch size; D_in is input dimension;
N, D_in = 1, 1000

# Create random Tensors to hold input and outputs.
x = torch.floor(torch.mul(torch.rand(N, D_in, device=device, dtype=dtype), 255))
y = torch.cos(x) + torch.mul(torch.randn(N, D_in, device=device, dtype=dtype), 0.1)

# Create Tensor for weights.
w = torch.linspace(0, 255, steps=256, device=device, dtype=dtype, requires_grad=True)

# To apply our Function, we use Function.apply method. We alias this as 'mapping'.
mapping = Mapping.apply

learning_rate = 1e-3
for t in range(500):

    # Forward pass: compute predicted y using operations; we compute
    # y_pred using our custom autograd operation.

    y_pred = mapping(x, w)
    # print(torch.max(torch.abs(y_pred)).item())

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    if t % 10 == 9:
        with torch.no_grad():
            print(t, loss.item(), ",", y_pred.abs().sum().item())

    # Use autograd to compute the backward pass.
    loss.backward()

    # Update weights using gradient descent
    with torch.no_grad():
        w -= learning_rate * w.grad

        # Manually zero the gradients after updating weights
        w.grad.zero_()

plt.plot(w.cpu().detach().numpy())
plt.show()

# Destroys the Quasar host
q.destroy()
```