Monthly Summary: September, 2020

The primary goal of the work this month was the development of an interface between existing PSCAD code and `QRFactor`. This process required a fundamental shift in the implementation of `QRFactor` from the standard, single-file CUDA project to a more object-oriented approach. This way, the system matrix could be factored and held on the device as its own object while repeated calls with some right hand side vector $\mathbf{b}$ as an input would return the desired output $\mathbf{x}$ from the equation $A\mathbf{x} = \mathbf{b}$. This would mean that the class QRFactor would need to support device-side functions as well as host-side functions. In this summary, we discuss the structure of our object-oriented approach as well as the technical details that need to be addressed to include GPU code within a standard C++ program.

The QRFactor Class

The QRFactor class provides the required interface between the existing PSCAD program and the GPU-based methods incorporated into `QRFactor`. In figure 1, we present a schematic of the structure of this class.

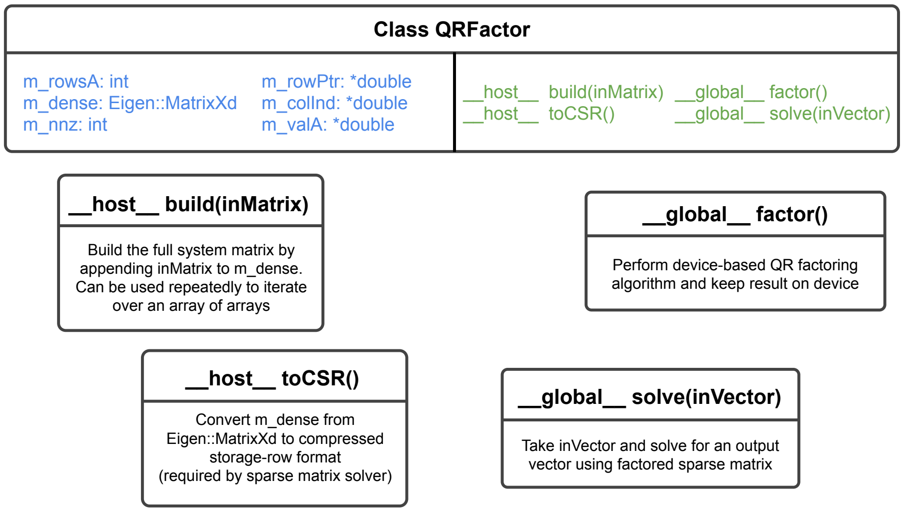
Class QRFactor
m_rowsA: int
m_dense: Eigen::MatrixXd
m_nnz: int
m_rowPtr: *double
m_colInd: *double
m_valA: *double
__host__ build(inMatrix)
__host__ toCSR()
__global__ factor()
__global__ solve(inVector)
__host__ build(inMatrix)
Build the full system matrix by appending inMatrix to m_dense. Can be used repeatedly to iterate over an array of arrays
__global__ factor()
Perform device-based QR factoring algorithm and keep result on device
__host__ toCSR()
Convert m_dense from Eigen::MatrixXd to compressed storage-row format (required by sparse matrix solver)
__global__ solve(inVector)
Take inVector and solve for an output vector using factored sparse matrix

Figure 1: *Schematic of the QRFactor class. Class variables are shown in blue while class functions are shown in green.*

In the event that GPU methods are required for a particular problem, the existing PSCAD code will pass an array of dense matrices to `QRFactor`. These dense matrices must be assembled into a single, large (sparse) system matrix before being passed to the GPU for factoring. To achieve this, the QRFactor class has a private dense matrix `m_dense` that can be iteratively extended to include the smaller submatrices. This occurs on the CPU with the open source Eigen package, an optimized C++ library for manipulating and solving linear systems. Once all the subsystems have been combined to give the full system matrix, another CPU function will convert the matrix into the compressed storage format that is required for factoring and solving on the GPU.

The device-side functions `factor` and `solve` perform the factoring of the system matrix A into QR

and the solve for **x** given an input **b**. These function in the same way as the stand-alone `QRFactor` program. Along with the class functions shown in figure 1, there are also several small `get` and `set` functions for the private class variables. Finally, to ease in later debugging, the QRFactor class has its own set of error codes that describe where in the build, conversion, factoring, or solving processes an error occurs.

Compiling & Linking

Unlike the previous versions of `QRFactor`, the new interface requires the separate compilation of source files that contained a mix of CPU and GPU functions. In order to achieve this, the NVIDIA compiler `nvcc` must compile .cpp files as .cu files if CUDA code was detected, but pass .cpp files *without* CUDA code to the standard compiler. Furthermore, the NVIDIA linker needs to be able to link object files with device code to object files without device code to produce the final executable. To illustrate this process, we consider an example of separate compilation and linking with mixed host and device functions that is available on the NVIDIA blog. See figure 2 for an illustration of the build structure for this example.

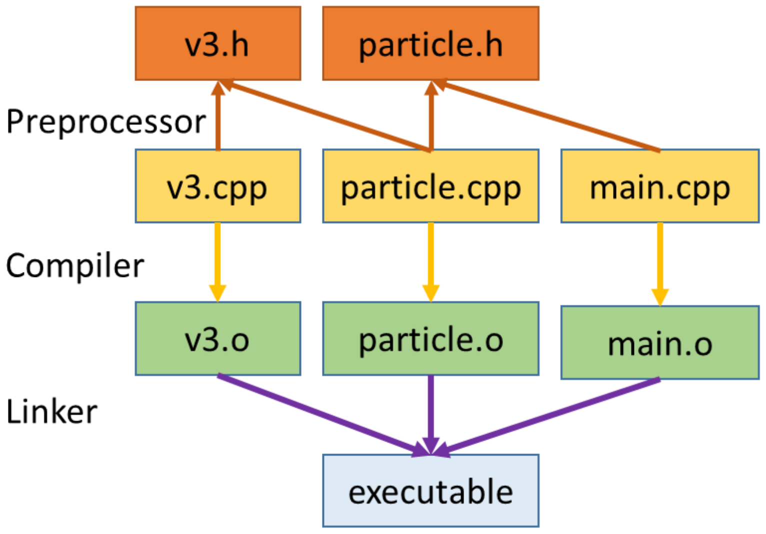

Figure 2: *C++ build structure for a sample program. Image credit NVIDIA blog.*

This example uses GPU-accelerated methods to describe the motion of a particle in three dimensions. The particle class contains functions that use both host-side functions and device-side functions, as well as relying on another class, v3, which itself contains mixed CPU and GPU functions. Like the functions that make up the QRFactor class, the `__host__`, `__device__`, and `__global__` decorators are used to indicate to the compiler whether to compile the functions as callable from the CPU only, the GPU only, or either. With these decorators written into the source files, calling `nvcc` with the `-x cu` option will tell the compiler to treat host code and device code differently, and the linker will be able to properly connect the resulting object files. In figure 3, we see the `-x` flag being used during compilation. While the `QRFactor` interface is currently being developed in Microsoft Visual Studio

2019, similar calls to the compiler are encapsulated in the project settings for a CUDA Runtime template.

```
objects = main.o particle.o v3.o

all: $(objects)
    nvcc -arch=sm_20 $(objects) -o app

%.o: %.cpp
    nvcc -x cu -arch=sm_20 -I. -dc $< -o $@

clean:
    rm -f *.o app
```

Figure 3: *The Makefile required to compile the sample program outlined in 2.*

For additional comments on programs with separate compilation – including possible options that can affect performance – see Chapter 6 in the CUDA NVCC Guide.

The development of the `QRFactor` interface is ongoing but is nearing completion. Actual integration with PSCAD software is forthcoming, and there will be several factors to consider, including the possibility of a flag to avoid initial subsystem splitting if it can be determined that GPU-based methods will be superior to multicore CPU methods. It is hoped that these and other possible considerations should not require any significant rewriting of the `QRFactor` interface due to its object-oriented approach.