Monthly Summary: December, 2020 & January, 2021

During this period, the full functionality – both device-side and host-side functions – of `QRFactor` was incorporated into a DLL (Dynamic Link Library) version and documentation was created to reflect these changes. To produce a DLL that is compiler independent, wrapper functions were created that take and return QRFactor-type pointers. In § 1, we show how the QRFactor DLL is used by a client program.

The inclusion of GPU functions meant that the DLL template available in Visual Studio was not immediately applicable. Instead, customization of the template was required. See § 2 for details.

Following a December meeting with the UW and MHI teams, additional changes to QRFactor were discussed that would allow for better integration will future PSCAD EMT updates. These additional changes are outlined in § 4 and are currently being implemented.

1 Using QRFactor DLL Within a Client Program

As mentioned above, the QRFactor DLL was written to be compiler-independent and does this by taking and returning typed pointers, which are then converted into void types upon compilation. With this structure in mind, a client program uses the QRFactor DLL in the following way:

```
#include "qrgpu.h"  // Header file for QRFactor library

int main()
{
    double* inPtr;  // Pointer to column-major dense matrix
    int* rPtr;      // Pointer to number of rows
    int* cPtr;      // Pointer to number of columns
    int* nnz;       // Pointer to total number of non-zero values

    /*
    * Create an instance of QRFactor with the total number of
    * non-zero values as an input
    */
    QRFactor* qr = QR_Create(*nnz);

    /*
    * Build the non-zero entries with the input matrix. Repeat as
    * many times as required until all subsystems have been read in
    */
    qr = QR_buildTriplets(qr, inPtr, *rPtr, *cPtr);

    /*
    * Assuming all subsystems have been read in, now build the
    * full system sparse matrix and convert to CSR storage
    */
    qr = QR_buildSparseMatrix(qr);
```

```
    /*
    * Use GPU-based factoring method on the full system matrix
    */
    qr = QR_factor(qr);

    /*
    * The matrix has been factored and is held on the GPU. Solve
    * the linear system for each set of input data.
    */
    double* b1;      // Pointer to input vector
    double* x1;      // Pointer to variable vector
    qr = QR_solve(qr, const_cast<double*>(b1), x1);

    double* b2;      // Pointer to input vector
    double* x2;      // Pointer to variable vector
    qr = QR_solve(qr, const_cast<double*>(b2), x2);

    /*
    * Memory cleanup
    */
    QR_delete(qr);

    return 0;
}
```

2 Creating a DLL with Device-side Functions in Microsoft Visual Studio 2019

Beginning with a CUDA runtime template project in VS2019, several adjustments are required to create a DLL with CUDA functions. In particular, the project's General Properties page must be changed so that the Configuration Type is Dynamic Library (.dll). See figure 1 for a screenshot. Other library linking, such as to the CUDA Toolkit, should be included with the CUDA runtime template

By examining the contents of a C++ DLL template, we see that there are several files provided by VS2019 that are required to build a DLL: "framework.h," "pch.h," "dllmain.cpp," and "pch.cpp". These short files provide entry points for the DLL applications and must be in each new project. Therefore, they must be added to the CUDA DLL solution as well. See § 5 for these files.

The user then creates a new header file that contains the library contents. Within the header file, any function declarations that are accessible to the client program must be wrapped with import/export macros and the `EXTERN` keyword. As an example, the following code defines the import/export macro `QRGPU_API` and the `EXTERN` keyword before declaring the example function `function`:

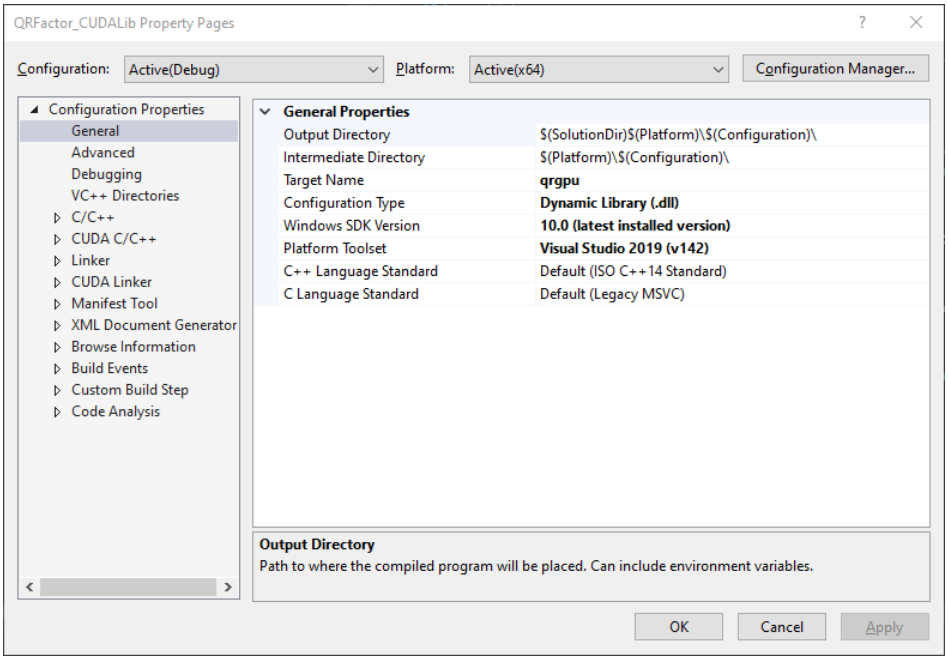


Figure 1: A snapshot of Configuration Properties → General tab after a CUDA runtime template is changed to a DLL project.

```
#ifdef QRGPU_EXPORTS
#define QRGPU_API __declspec(dllexport)
#else
#define QRGPU_API __declspec(dllimport)
#endif

#ifdef __cplusplus
#define EXTERN
#else
#define EXTERN extern "C"
#endif

EXTERN QRGPU_API function;      // Example function declaration
```

3 Using a Custom CUDA DLL in a Client Program in Microsoft Visual Studio 2019

Having created the custom DLL, a VS2019 client program can use the library once it has been properly linked. After creating a new Console Project in VS2019, we add the path to the CUDA DLL as well as any libraries that the DLL may rely on (in this example, the Eigen library and CUDA Toolkit libraries are also included) in the C++ Additional Include Directories. See figure 2.

Next, the CUDA DLL library must be pointed to directly by specifying the path of the `#include`

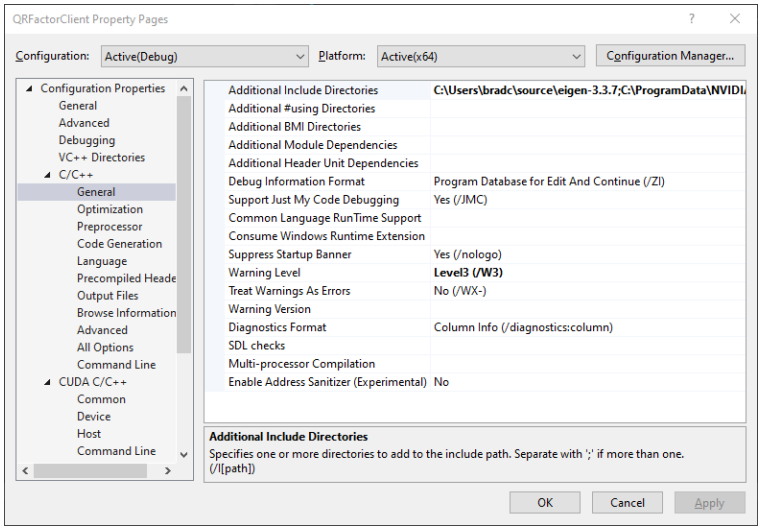


Figure 2: A snapshot of Configuration Properties → C/C++ → General tab in a client program. The CUDA DLL directory must be included in the Additional Include Directories, as well as any libraries that it may depend on.

declaration. This is set in the Linker General options, as seen in figure 3a. Note that the `$(IntDir)` macro will allow for the linking of either a *Debug* or *Release* version of the DLL to be linked. Also in the Linker options, the name of the CUDA DLL itself must be added on the Input tab under Additional Dependencies, as shown in figure 3b.

Finally, a Post-Build process must be added to copy the CUDA DLL into the client program's build output directory. This is done through the Build Events options by specifying a Post-Build Event. In the Command Line field, the CUDA DLL (named "example.dll") is copied directly into the build directory using

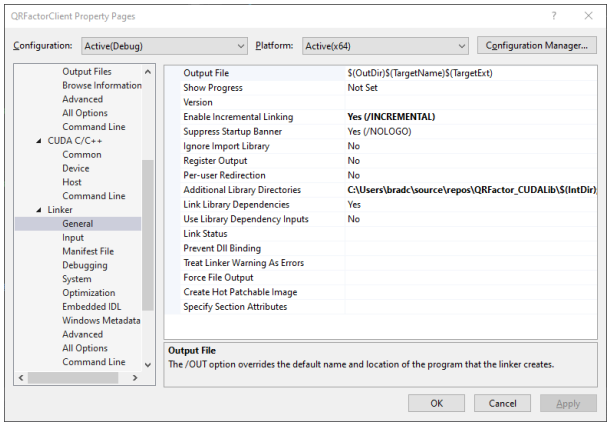
```
xcopy /y /d "Path\to\Library\$(IntDir)example.dll" "$(OutDir)"
```

See figure 4 for a snapshot. After these options have been set, the client program invokes the CUDA DLL using the standard `#include` macro.

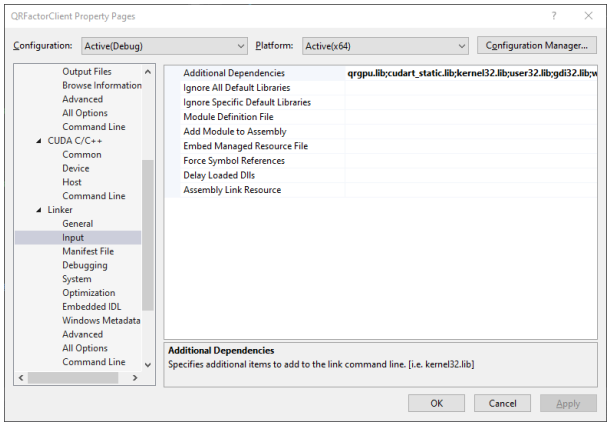
4 Pending Additions

The following additions to the QRFactor DLL were discussed and are currently being implemented:

- Check whether a GPU is connected and return an error if the library is not able to run on the existing architecture.
- Use a map container to trace the positions of the submatrices within the larger full system matrix. This can be used later to quickly identify subsystems with values that change within the EMT system being studied, allowing the system matrix to be rapidly rapidly updated without requiring reassembly.



(a) A snapshot of Linker → General tab in a client program. Under Additional Library Directories, the path to the CUDA DLL is added.



(b) A snapshot of Linker → Input tab in a client program. Under Additional Dependencies, the name of the library is included.

Figure 3: Linker options in the client program.

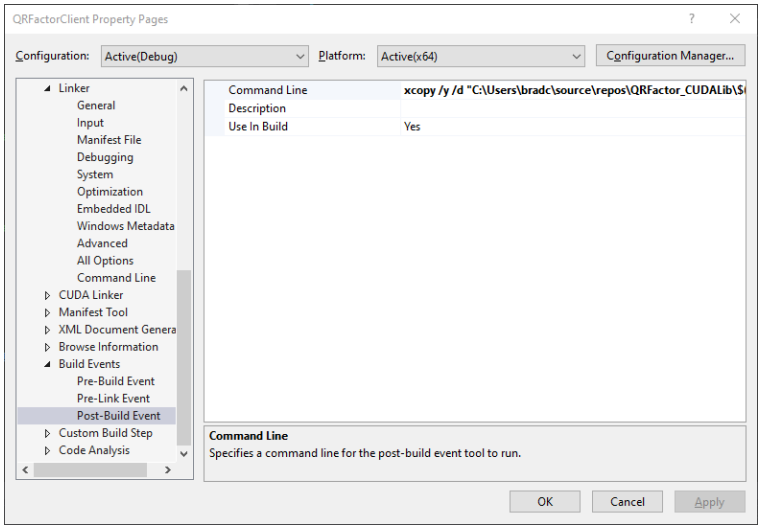


Figure 4: A snapshot of Build Events → Post-Build Event tab in a client program. The CUDA DLL is copied into the client program output directly as a post-build event.

5 DLL Template Files

These files are part of a C++ DLL template which, when constructing a CUDA DLL, must be added by hand. They are included here for reference.

pch.h

```
#ifndef PCH_H
#define PCH_H

// add headers that you want to pre-compile here
#include "framework.h"

#endif //PCH_H
```

pch.cpp

```
// pch.cpp: source file corresponding to the pre-compiled header
#include "pch.h"

// When you are using pre-compiled headers,
// this source file is necessary for compilation to succeed.
```

framework.h

```
#pragma once

#define WIN32_LEAN_AND_MEAN   // Exclude rarely-used stuff from Windows headers
// Windows Header Files
#include <windows.h>
```

dllmain.cpp

```
// dllmain.cpp : Defines the entry point for the DLL application.
#include "pch.h"

BOOL APIENTRY DllMain(HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
)
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
    case DLL_THREAD_ATTACH:
```

```
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```