# Monthly Summary: October, 2020

Development of the GPU/PSCAD interface continued this month with the variables and methods of the QRFactor class continuing to be refined. In particular, the method of creating the full system sparse matrix was improved by removing the need for an intermediate dense matrix during its construction. Instead, the non-zero entries are stored in an Eigen-specific object called a triplet that can be set dynamically as each subsystem is read in. Once all (column-major ordered) subsystems have been read in, existing Eigen methods are called to allocate the minimum amount of memory required for the sparse matrix for the full system. Once the sparse matrix has been created, Eigen generates the pointers required for the Compressed Storage Row-major (CSR) description that is then passed to the GPU where factoring and solving takes place. Forgoing the expensive and time-consuming step of an intermediate, dense-matrix representation of the full system will result in a significant speed up of data preprocessing, particularly for large systems. For the small test system of a $5 \times 4$ sparse matrix, the read-in and conversion to CSR format was timed at 135 $\mu$s. It remains to be seen how this will scale with matrix size.

Unlike previous versions of `QRFactor`, the QRFactor interface contains a mix of host (CPU) and device (GPU) functions that are compiled separately by `nvcc` and then linked. Furthermore, because `QRFactor` is now an interface to be called from another program as opposed to a stand-alone program, the implementation of the proven QR solving routines has changed. The main differences between the current interfacing version of `QRFactor` and earlier versions are:

- The building of the full system matrix, $A$, occurs incrementally as new dense matrices are read in, as opposed to reading the full system matrix from a file.
- The factoring of $A$ is performed by a *separate function* that is called once all the non-zero values of $A$ are known. The resulting factored version of $A$ must be held in class variables that will not go out of scope after the factoring occurs.
- The NVIDIA functions that perform the factoring and solving almost always take and return pointers. Storing the result of factoring in class pointers that can are referenced by a separate class function requires direct memory copying prior to the factoring pointers going out of scope.
- The results of the factoring must be called repeatedly for each solve step. Referencing the appropriate memory location is essential to accomplishing this.

During the development of the new interfacing version, care had to be taken to ensure that class variables were properly updated during each step of the factoring and solving. Passing these variables back to a class function that performs the solving is currently causing difficulties.

As a note for future refinement: it is unclear what the procedure `nvcc` uses when deciding which variables are able to be read by the host or device. Additional variable decorators such as `__device__` and/or `__managed__` could reduce possible duplications from not specifying the variable scope at compile time. The same may be said for class functions. Currently, no function decorators are required; instead, the compiler is able to determine which functions need to access the device. Again, it may be that `nvcc` is compiling two versions of each function – one for use on the device and one for use on the host – and using whichever version is required at runtime. Declaring which functions have access to the device at compile time may reduce memory overheads.