Parallelizing Calculation of Power Distribution Factors PTDF and LODF

Shreeganesh Bhat, John Sandell
University of Minnesota, Twin Cities

ABSTRACT

Distribution factors are used mainly in security and contingency analysis. They are used to approximately determine the impact of generation and load on transmission flows. Power Transfer Distribution Factor (PTDF) and Load Outage Distribution factor (LODF) as two such factors which will give an insight on effects of power generation and load. PTDF calculates a relative change in power flow on a particular line due to a change in injection and corresponding withdrawal at a pair of busses and LODF calculates a redistribution of power in the system in case of an outage. Goal of this project is to parallelize the calculations using CUDA C on the GPU GTX 480.

Index Terms

Power Flow; Power System Security; Power Transfer Distribution Factor; Load Transfer Distribution factor

1. INTRODUCTION

Long distance bulk power transfers are essential for an economic and secure supply of electric power. To operate system safely, the transfer capabilities must be calculated so that the transfers do not exceed the capability. For analysis, power system operators rely heavily on simulation of a model of the system obtained offline. Calculating Distribution factors are one such method which is used for online contingency analysis, generation re-dispatch, and congestion relief etc. They show how a power flow variable such as flow, voltage, phase angle, etc. change with the change of another value such as injection, flow etc. For this calculation, sequential programming methods are not suitable for power flow analysis of the system with more than thousand nodes as they are time consuming. So this project aims to develop an application using parallel computation techniques to perform these computations faster.

2. DESIGN OVERVIEW

PTDF Calculation : PTDF is the relative change in power flow on a particular line due to power injection and corresponding withdrawal at a pair of busses. For example, consider a pair of buses, '*m*' and '*n*' and a transmission line '*l*' as shown below,

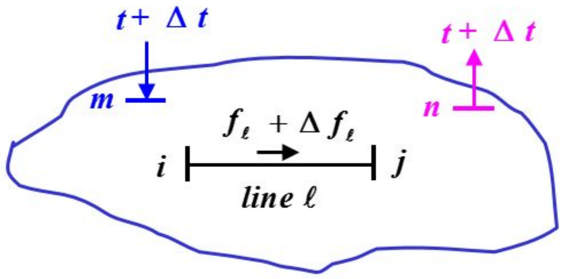


Figure 1: Change in power flow in a line

Which is connected between the pair of buses '*i*' and '*j*'. Assume initially, flow through the line '*l*' is '*fl*'. Now imagine, '*Δt*' amount of power has been injected at the bus '*m*' and the same amount is withdrawn at the bus n. Now, this transaction causes additional flow '*fl + Δfl*' on the line '*l*'. The additional amount of power that flows over the line due to the injection and withdrawal of power '*Δt*' is given by PTDF.

It is represented as shown below $\varphi_l^{(w)}$

$$PTDF\left(\varphi_l^{(w)}\right)=\frac{\Delta f_l}{\Delta t} \quad \text{..................................(1)}$$

In order to reduce the calculation time, DC Power flow method has been used to solve power flow problem. Problem is simplified by making the system linear by making following three assumptions:

1. Line resistances (active power losses) are negligible i.e. R << X
2. Voltage angle differences are assumed to be small i.e. sin (Θ) = Θ and cos (Θ) = 1.
3. Magnitudes of bus voltages are set to 1.0 per unit (flat voltage profile).

Based on these assumptions, voltage angles and active power injections are the variables of DCPF. Active power injections are known in advance. Therefore,

$$P_i = \sum_{j=1}^{N} B_{ij}(\theta_i - \theta_j) \quad \text{........................(2)}$$

In which, '*Bij*' is the reciprocal of the reactance between bus *'i'* and bus *'j'* and '*Bij*' is the imaginary part of *'Yij'*. As a result, active power flow through transmission line *'i'*, between buses '*s*' and *'r'* , with reactance between the line '*XLi*' of a line *'l'* can be calculated as

$$P_{Li} = \frac{1}{X_{Li}}(\theta_s - \theta_r) \quad \text{.................................(3)}$$

So, the DC power flow equations in the matrix form and the corresponding matrix relation for flows through branches are represented as

$$\theta = [\mathbf{B}]^{-1}\mathbf{P}$$

$$\mathbf{P_L} = (\mathbf{b} \times \mathbf{A})\theta \quad \text{...............................(4)}$$

Where,
P is an N x 1 vector of bus active power injections for buses 1to N,
B is an N x N admittance matrix with R = 0
Θ is an N x 1 vector of bus voltage angles for buses 1 … N
PL is an M x 1 vector of branch flows (M is the number of branches),

b is M x M matrix (bkk is equal to the susceptance of line k and non-diagonal elements are zero)
A is M x N bus-branch incidence matrix

Above equation can be written in the matrix form as shown below

$$\begin{bmatrix} P_1 \\ P_2 \\ \\ \end{bmatrix} = [Bx] \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \end{bmatrix} \text{ where } Bx = \begin{bmatrix} \sum_{j=1}^{numbus} \frac{1}{x_{ij}} & \frac{-1}{x_{ij}} & \cdots \\ \frac{-1}{x_{ij}} & \sum_{j=1}^{numbus} \frac{1}{x_{ij}} & \cdots \\ \vdots & \vdots & \vdots \\ \end{bmatrix} \quad (5)$$

We make use of parallel programming technique to generate this matrix from the data provided in the input excel sheet, explained later in the implementation section.

Note that, the matrix ‘Bx’ is a singular matrix. To make this matrix non-singular, we zero out the row of ‘Bx’ corresponding to the reference bus and then set the diagonal term of the reference bus row to 1. Now this matrix ‘Bx_alt’ can be inverted and we have called this reduced matrix as ‘Bx_alt’ in our program

$$[X\_\text{alt}] = [Bx\_\text{alt}]^{-1} \quad ....(6)$$

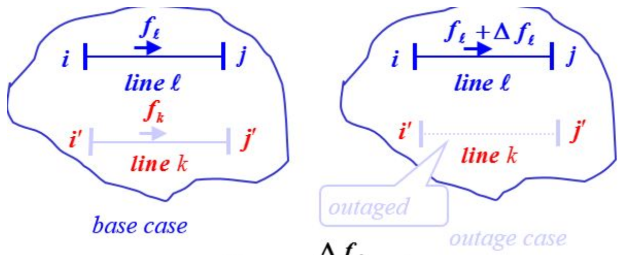
Now (3) can be modified to calculate PTDF as show below

$$PTDF_{r,s,i,j} = \frac{1}{x_{ij}}\left[\left(X_{is} - X_{ir}\right) - \left(X_{js} - X_{jr}\right)\right] \quad ....(7)$$

Before calculating LODF, PTDF value should be checked to see if a line outage will not cause islanding condition. Islanding refers to the condition in which a distributed generator continues to power a location even though electrical grid power from the electric utility is no longer present. Islanding can be dangerous to utility workers, who may not realize that a circuit is still powered, and it may prevent automatic re-connection of devices. For that reason, distributed generators must detect islanding and immediately stop producing power. This condition is detected when the diagonal of any line in PTDF is very close to 1.0. I,e all the power flowing through this line.

If such a lines are detected, those lines causing islanding condition are listed and for the calculation purpose, PTDF value corresponds to the line is set to zero to avoid divide by zero errors.

Calculating LODF: When an outage occurs, the power flowing over the outage line is redistributed onto the remaining lines in the system. Since each contingency requires separate LODFs, a quick calculation of LODFs, especially with multiple-line outages, could speed up contingency analysis and significantly improve the security analysis of the power system. This project provides a direct method for expressing LODFs in terms of power transfer distribution factors (PTDFs) of pre-contingency network.



$$d_\ell^k = \frac{\Delta f_\ell}{f_k} = d_{\ell,k}$$

Figure 2 : Change in flow due to an outage

In the figure shown above we want the flow on line *‘l’* from bus *‘i’*
to *‘j’*, with the outage of line ‘*k’* from bus ‘*n’* to ‘*m’*. We note that the original power flowing on line k from bus n to bus m is Pnm. When the injections *‘ΔPn’* and ‘*ΔPm’* are added to bus *‘n’* and bus *‘m’* respectively, the resulting flow on line *‘k’* is ‘ *nm’*. Using the same logic, opening of line k can then be simulated if,

$$\Delta Pn = \dot{P}nm \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(8)$$
$$\Delta Pm = -\dot{P}nm \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(9)$$

This means that all of the injected power into bus *‘n’* flows in line ‘*k’* and out of bus *‘m’* so that there is no flow through the breaker connecting bus *‘n’* to the remainder of the system, and no flow through the breaker *‘m’*. Now ‘ *nm’* can be calculated easily if we note that the flow on line *‘k’* for an injection into bus *‘n’* and out of bus *‘m’* is simply

$$\dot{P}nm = Pnm + PTDFn,m,k * \Delta Pn \ldots\ldots\ldots(10)$$

We are using the PTDF to calculate how much of the injection *‘ΔPn’* ends up flowing on line *‘k’*, but by definition

$$\Delta Pn = \dot{P}nm \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots..(11)$$

Then,

$$\dot{P}nm = Pnm + PTDFn,m,k * \dot{P}nm \ldots\ldots\ldots..(12)$$

Simplifying equation 11, we get

$$LODF_{\ell,k} = PTDF_{n,m,\ell}\left(\frac{1}{1 - PTDF_{n,m,k}}\right) \quad ....(13)$$

Equation (5), (6), (7) and (13) shown above involves multiplication, building a matrix and calculating an inversion performed on a GPU GTX 480 as explained below.

3. IMPLEMENTATION ON GPU

Calculating PTDF and LODF requires several calculations, most of which are matrix multiplications and inversions. MATLAB’s Parallel Computing Toolbox provides a convenient interface to accessing and using Cuda kernels in-line with other MATLAB code, which is similar to running kernels in C. The first step is compiling a kernel from a .cu file into a .ptx file via the command window. The command to do this is shown in Figure 3. This .ptx file is specific to MATLAB and it enables MATLAB to interface to the GPU. This is also comparable to compiling .cu files with "nvcc" to be used with .c and .cpp files. Then within a MATLAB script, a kernel object must be created and its block size and grid size parameters modified accordingly. Then GPU memory is allocated and filled with the 'gpuArray()' command and the kernel can be called and ran. Lastly, any values in the GPU memory can be gathered and cleared to make room for the next kernel. All of these required MATLAB commands are shown in Figure 3.

```
% Compiles .cu file into a .ptx file. Needs to ran from
% the MATLAB command window
system('nvcc -ptx example.cu')

% creates CUDAKernel object to be used in kernel calls
k = parallel.gpu.CUDAKernel('example.ptx', 'example.cu')

% sets grid dimensions and block dimensions
k.GridSize = [GRID_SIZE GRID_SIZE GRID_SIZE];
k.ThreadBlockSize = [BLOCK_SIZE BLOCK_SIZE BLOCK_SIZE];

% Copies array "X" to the GPU and returns a gpuArray
% object to G
G = gpuArray(X)

% Calls and evaluates kernel k with arguments A,B,C and
% returns A to Y
Y = feval(k, A, B, C)

% copies gpuArray object B into array Z
Z = gather(B)

% clears allocated memory associated with kernel k
clear k
```

Figure 3: Commands to compile and use Cuda kernels in MATLAB.

3.1 Matrix Multiplication

The implemented matrix multiplication kernel utilizes global constant memory and coalesced warps to achieve speedups. It begins by initializing two dimensional blocks and grids to cover the dimensions of the output matrix. Each thread then takes the dot product of the corresponding row of the multiplicand and the corresponding column of the multiplier. That value is initially stored in a register, but it gets copied to the output matrix in global memory after the dot product is complete. This implementation allows for each element of the output matrix to be computed in parallel instead of sequentially.

Because shared memory was not implemented, tiling was not necessary. This was because the size of the matrices was roughly limited to 10,000x10,000 due to the application. It was thought shared memory would not provide much extra benefit since the multiplication kernel was already running considerably faster than the MATLAB matrix multiplication method. Because this kernel was used a total of six times throughout the PTDF and LODF calculations so it was thought speeding this up with even just a basic kernel would yield large performance increases. This kernel for multiplying matrices is shown in Figure 4.

```
__global__ void MatrixMulKernel( double *P ,const double
    *M, const double *N , const int Mat1_Width, const int
    Mat1_Height, const int Mat2_Width, const int
    Mat2_Height , const int Mat3_Width)
{
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
    double Pvalue = 0.0;

    if((Row < Mat1_Height) && (Col < Mat2_Width))
    {
        for (int k = 0; k < Mat1_Width ; ++k)
        Pvalue += M[Row*Mat1_Width + k] *
            N[k*Mat2_Width + Col];

        P[Col*Mat3_Width + Row] = Pvalue;
    }
}
```

Figure 4: Cuda matrix multiplication kernel.

3.2 Matrix Inversion

Our matrix inversion kernel is an implementation of Gauss-Jordan elimination. It takes an input matrix and an identity matrix and iteratively performs elementary row operations to both matrices with the goal of reducing the input matrix into an identity matrix. When these operations are performed on the original identity matrix, it will iteratively turn into the inverse of the original input matrix. To do this in Cuda, we have implemented three different versions that are all variations of a proven Cuda matrix inverse program.

The first version consists of iteratively running four steps to reduce the matrices. Pseudocode for the kernel is shown in Figure 4.

```
__global__ void invert(double *I, double *A, const int *n)
{
//iterate through rows
for (int i = 0; i<n[0]; i++){
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    //Non diagonal normalization
    if x and y are within the matrix dimensions
         if x and y are not equal
            divide A[y][x] and I[y][x] by A[x][x]

    //Diagonal normalization
    if x and y are within the matrix dimensions
        if x and y are equal
            divide A[x][x] and I[x][x] by A[x][x]

    //Gauss Jordan Elimination
    if x and y are within the matrix dimensions
        subtract one row from another

    //Set to zero
    if x and y are within the matrix dimensions
        Set leading coefficients of A to 0
}
```

Figure 4: Pseudocode for version one of matrix inversion.

After each step, threads are synchronized so avoid race conditions within blocks. Race conditions could arise when a part or all of the matrix is smaller than the thread-block and some threads are required to do less work than others so they execute the kernel faster which leads to modifying memory with undesired values. Since synchronizing across all blocks is not possible short of ending the kernel, this implementation does not work for matrices larger than the thread-block size.

The second version divided up the first kernel into four separate kernels and attempted to sequentially call them in a for-loop in MATLAB. Issues getting correct data transfer between MATLAB on the CPU and the GPU proved to be a difficult task and this version of the method did not perform for matrices and blocks larger than 8x8. The key difficulty that could have been solved given more time was ensuring all data was correct when transferring from one kernel to MATLAB and then from MATLAB to the next kernel. This was attempted via a single output matrix that was equivalent to the concatenation of the input matrix and the identity matrix. But again, getting the desired data in the correct spot takes time to debug and complete.

The third matrix inversion kernel was a combination of the first two. The four required steps were put into separate kernels but this time they were iteratively called from a for-loop in a main C function. The idea was to precompile a C and Cuda inverse program then call it from MATLAB and give it the required data via a comma-separated-variable file generated in MATLAB. A block diagram of this idea is outlined in Figure 5 from the MATLAB script point of view. Here all kernel initializations would be avoided in MATLAB and a known operational kernel could be used to invert a matrix of any size. The drawback to this method is increased latency due to reading and writing .csv files. This latency proved to be too large and actually degraded performance, on the order of tens of seconds for large matrices which is orders of magnitudes larger than the actual computation time for either system.

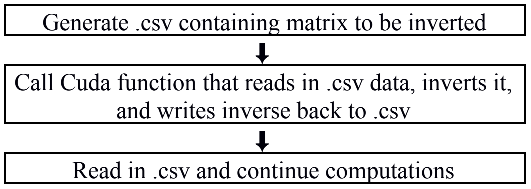

Generate .csv containing matrix to be inverted
Call Cuda function that reads in .csv data, inverts it, and writes inverse back to .csv
Read in .csv and continue computations

Figure 5: Flowchart for version three of matrix inversion.

3.2 Admittance Matrix Calculation

Parallelizing the admittance matrix calculation was an experiment to observe the effectiveness of converting basic for-loops into Cuda kernels. Figure 6 is the original MATLAB implementation for computing the admittance matrix previously detailed.

```
for iline = 1 : numline
if BranchStatus(iline) == 1
    Bx(frombus(iline),tobus(iline)) =
        Bx(frombus(iline),tobus(iline))-1/xline(iline);
    Bx(tobus(iline),frombus(iline)) =
        Bx(tobus(iline),frombus(iline))-1/xline(iline);
    Bx(frombus(iline), frombus(iline)) =
        Bx(frombus(iline),frombus(iline))+1/xline(iline);
    Bx(tobus(iline),tobus(iline)) =
        Bx( tobus(iline),tobus(iline))+ 1/xline(iline);
end
end
```

Figure 6: MATLAB implementation of computing admittance matrix.

This for-loop iterates for each bus, up to 1000 times in our test cases. The parallel version of this for-loop is shown in Figure 7 where it only iterates once. For the Cuda implementation multiple arrays had to be passed to this kernel in order to index the admittance matrix, thus increasing the kernel overhead. But reducing this loop to roughly one iteration compensates for any overheads of large matrices. To improve performance of the kernel, registers were used to store the indexing values leading to only three global memory reads.

```
__global__ void BuildBx( double * Bx, const int * frombus,
const int * tobus, const int * BranchStatus,
const double * xline, const int numline,
const int Mat4_Width)
{
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
int Col = blockIdx.x*blockDim.x+threadIdx.x;
    int fb = frombus[Row];
int tb = tobus[Row];

    if(Row == fb && Col == tb)
            Bx[fb*Mat4_Width + tb] /= xline[Row];
}
```

Figure 7: Cuda implementation of computing admittance matrix.

4. VERIFICATION

In order to verify the accuracy of the parallel implementations, the existing sequential version was also ran and both results were compared. The method of comparison consisted of using MATLAB's signal-to-noise ratio and mean-squared-error methods to determine these relationships between the sequential implementation and the parallel implementation. Both methods returned values corresponding to negligible errors.

Resulting PTDF and LODF matrices were also compared element wise by subtracting the Cuda matrices from the MATLAB matrices and observing error for each element. This method also checked out with our ideas of what acceptable error is in this application.

This application has different acceptable errors depending on the values in the PTDF and LODF matrices. As these values approach the critical value of 1, the importance of the error goes up to six or seven points of precision. If any values are within 10^{-6} of 1, the application automatically rounds the value up, labeling it as critical. But as the values move away from 1, error becomes less important. For example, acceptable error for values near 0.8 require five points of precision while values closer to 0.1 and 0 require four points of precision. Overall, any error obtained in the computed power factors was acceptable and did not have any noticeable effects.

5. PERFORMANCE

Overall, only a minor performance boost was seen. Figure 8 shows a plot of the performance for calculating PTDF and LODF on the CPU via MATLAB and using Cuda on the GPU in conjunction with MATLAB.

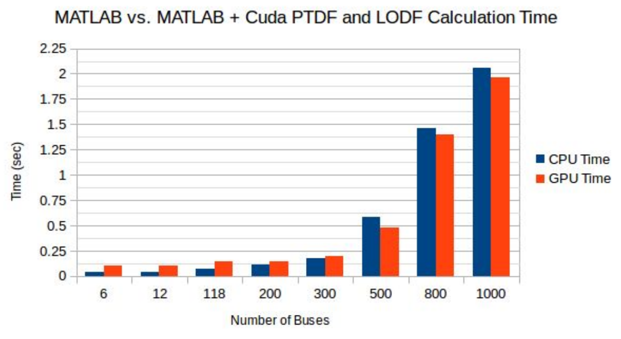


Figure 8: Time to calculate PTDF and LODF for serial and final parallel implementations.

Performance goals for the whole calculation were not achieved, but some individual kernel performance goals were. The matrix multiplication kernel was able to outperform MATLAB's matrix multiplication, easily multiplying matrices up to 8X faster in our test cases. With six matrix multiplications in the application, the parallel kernel improved overall calculation time by six times the improvement of one kernel. Although that is not including the overhead of setting up the thread dimensions, allocating memory, and copying data. We were able to call this kernel directly through MATLAB by compiling it into a .ptx file and then setting up block/grid dimensions and allocating memory similar to how it is done in Cuda. This yielded the shortest possible overhead while still allowing use of the GPU. The overhead of using the GPU was only noticeable for matrices smaller than 50x50 when the MATLAB multiplication would calculate faster. Using shared memory in the matrix multiplication kernel could have further improved performance. But seeing how little time is spent multiplying the matrices versus everything else in the MATLAB part of the application, we did not think shared memory would provide much of an overall improvement. Figure 9 compares the CPU and GPU matrix multiplication times.

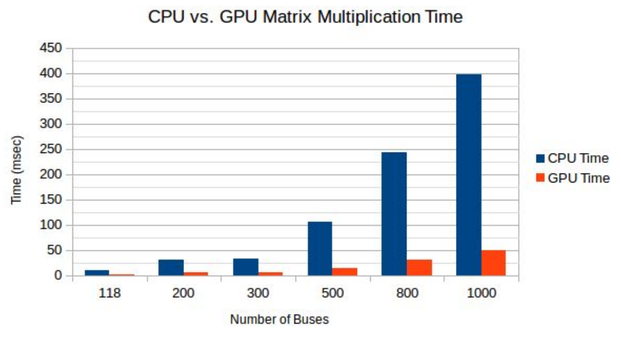


Figure 8: Comparison of CPU and GPU matrix multiplication times.

The Admittance matrix calculation also benefited from the parallelization and ended up running up to 3.5X faster. The overhead for running the kernel is quite noticeable at low matrix sizes since there is really no increase in performance for any matrices up until about 500x500. But this also meannt turning the original for-loop into a two-line parallel implementation led it to run fast enough where the calculation time was insignificant compared to the overhead. This is also not including the lines of MATLAB code that were required to set up the thread dimensions, allocate memory, and copy data over. Figure 10 compares the CPU and GPU admittance matrix calculation times..

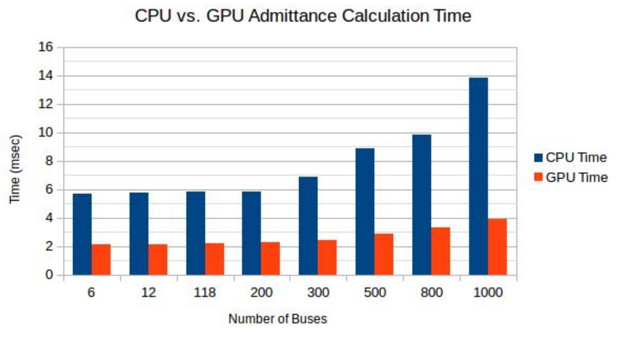


Figure 10: Comparison of CPU and GPU admittance matrix calculation times.

The goal that went unachieved was creating a beneficial matrix inversion kernel. The first version of this kernel performed on par with MATLAB's matrix inverse for matrices up to 32x32 but failed beyond that. The other two kernels either did not work for any reasonably sized matrices or performed too poorly. In the end, more time could have been spent on the inverse kernel and making it work either by calling it in MATLAB or implementing a more efficient C program to be called by MATLAB.

Overall, given more time and possibly implementing the whole application in C and Cuda could have yielded up to a 5x performance boost. The main reason for our lack of performance increase so far seems to be because so much of the application is written in MATLAB and creating all kinds of different kernels does not help much due to each kernels overhead. If we could condense more Cuda operations into fewer kernels, we think the overhead per kernel would drop down enough to increase performance significantly more than the current performance. MATLAB is also very optimized for the calculations it is performing. We think if the application was implemented serially in C, it would be 100x slower than the MATLAB implementation. That could be a future baseline to check Cuda performance against if future applications were to be entirely written in C and Cuda.

Another future optimization could be implementing file I/O kernels. Although it is not included in Figure 8, reading in the data from Microsoft Excel file and writing the data to files constitutes a significant amount of the time spent waiting for the application to complete. For example, for the 1000 bus case, it took two seconds to complete the calculation, but near two minutes to read and write all of the data. Since the data comes in several different data sets, it may also be possible to implement a stream. This would allow for calculations to take place on already parsed data while the next data sets are being read in. In the end, we think performance could be boosted if more of the C and Cuda capabilities were to be implemented.

6. CONCLUSION

In conclusion, there was not a performance for the overall power factor calculation given the two Cuda kernels that were implemented. As previously noted, MATLAB's matrix operations are extremely optimized and larger data sets would need to be analyzed in order to see clear performance increases on the GPU given current implementations. Performance could also be increased by writing more of the application in Cuda and C so that more of the application can be done on the GPU. If more is done on the GPU, then more operations could be condensed into fewer kernels, leading to less kernel overheads. Ideally, The entire application would be written in Cuda and C. And since most of it is manipulating matrices, most of it can be done in parallel. All in all, this was a good experience that introduced our team to GPU computing in real-world applications and enabled us to experiment with Cuda kernels on a pre-existing application.

7. REFERENCES

[1] Power Generation, Operation and Control, 3rd Edition by Allen J. Wood, Bruce F. Wollenberg, Gerald B. Sheble ISBN: 978-0-471-79055-6.

[2] Chandavarapu, S. and Bhat, S. 2015. Calculation of Distribution Factors PTDF and LODF. Technical Report. University of Minnesota Twin-Cities