

Modified Nodal Analysis Report

November 20, 2019

Author: Philippe Nadon

1 Project Description

1.1 Introduction

Oftentimes in electrical engineering, the voltage at each node in a circuit needs to be known to understand how the circuit works. As the complexity of the circuit grows, it becomes increasingly difficult or even impossible to use many of the common ways of solving the circuit, let alone doing it by hand.

1.2 Problem Definition

To solve this problem one would need to take a circuit and represent it in a textual manner (ie. a dataframe).

The dataframe would then need to be fed into an algorithm, which analyzes each component and outputs the value of every unknown in the circuit (eg. the voltage at each node) based on the relationship between each component, at discrete time intervals to simulate the current flow throughout the circuit.

The challenge of such a tool, is attempting to speed up the simulation of the circuit, since due to the data dependency between each time step, one cannot distribute the work per time step to multiple processes. The most parallelizable area is solving the system of linear equations.

1.3 Solution

Thus, a solution is to algorithmically implement Modified Nodal Analysis (MNA); a technique which is guaranteed to solve any circuit. To speed up computation time, the MNA implementation can be parallelized, which is greatly appreciated when working with very large and complex circuits.

The way the MNA algorithm works is that it takes the textual representation of the circuit, assembles a system of linear equations based off the known values of the components in the circuit, and solves for the unknowns in these equations. Due to the variation in the current sources, the voltage and current will vary in the circuit with respect to time, and thus the solver will repeatedly solve for these unknowns at each time step.

Thus, the computation resembles the form of $Ax = b$.

2 Introduction to Tool

This project explores methods in which one can implement a circuit analysis tool, using [Modified Nodal Analysis](#). A large portion of the work in this project is thanks to [this tutorial](#), which was immensely useful in developing this software.

The code was implemented in Python, using the typical scientific libraries (Numpy, Scipy, Pandas), as well as CuPy, which serves as a wrapper for Python code to interact with CUDA.

3 Techniques Used

3.1 Modified Nodal Analysis

The algorithm uses Modified Nodal Analysis to construct a system of linear equations, which is then solved on the CPU or GPU. As a result, both algorithm design and parallelism were part of the learning experience for making this tool.

3.2 Random Circuit Generator

To test for varying sizes and densities of circuits, a random circuit generator was developed. This circuit generator ensures that any two nodes on the circuit forms a cycle, and that the distribution of connections is well varied and distributed, using various algorithms as well as a [gamma distribution](#). Thus, this project also involves some graph theory.

3.3 CUDA Sparse Linear Solver

Both a [CPU-based solver](#) and a [GPU-based solver](#) were used, to compare the performance of both. Some knowledge on how CUDA works was required to use the GPU-based solver, which will be explained below.

4 About CUDA

The following information is from Cheng, Jie. “Programming Massively Parallel Processors. A Hands-on Approach.” *Scalable Computing: Practice and Experience 11* (2010): n. pag.

CUDA is a parallel computing platform released by NVIDIA in 2007, and was designed to tackle numerically-intensive tasks in parallel. CUDA offers developers an interface where they can directly execute code using the cores on a GPU. Because a GPU has many more cores than a CPU, its total throughput is orders of magnitude greater than that of a typical CPU. Since GPUs are typically used for graphics processing, they are especially adept at matrix computations, and thus performing linear solves on a GPU should have some real-world advantages over typical CPUs.

One significant bottleneck to CUDA computing, is the latency between beginning the transfer of data to the GPU through the PCIe lanes, and beginning the computation on the GPU. High levels of memory transfer between main memory and the dedicated GPU memory can slow down the computation, and so one must ensure that the amount of time spent transferring data is minimal.

5 Specifications of Hardware Used

Intel i5 4690, 4 cores at 3.5GHz, turbo boosting up to 3.9GHz 16GB DDR3 main memory NVIDIA GTX 970 with 4GB of GDDR5 memory, 1664 CUDA cores at 1050MHz

6 Results

The results are displayed below, first is the resulting CSV containing the times for each type of solver, per matrix size and node density:

```
[1]: import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

```
[2]: df = pd.read_csv("stats.csv")
df = df.dropna()
df
```

```
[2]:
```

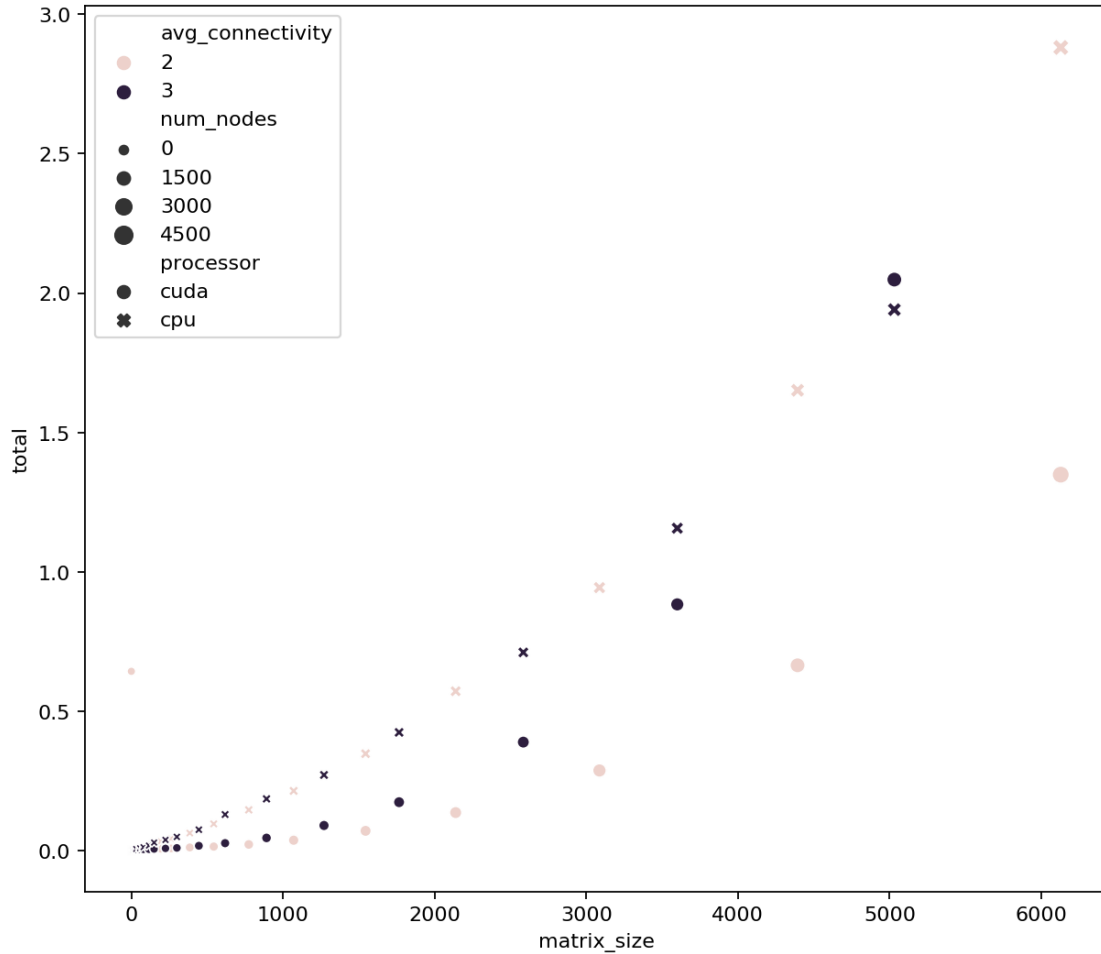
	fname	matrix_size	num_nodes	\
0	rand_data/rand_size-003_density-002.csv	1	2	
1	rand_data/rand_size-003_density-002.csv	1	2	
2	rand_data/rand_size-004_density-002.csv	4	4	
3	rand_data/rand_size-004_density-002.csv	4	4	
4	rand_data/rand_size-005_density-002.csv	7	5	
..	
81	rand_data/rand_size-021_density-003.csv	2585	1448	
82	rand_data/rand_size-022_density-003.csv	3600	2048	
83	rand_data/rand_size-022_density-003.csv	3600	2048	
84	rand_data/rand_size-023_density-003.csv	5031	2896	
85	rand_data/rand_size-023_density-003.csv	5031	2896	

	avg_connectivity	num_components	processor	total	transfer	solve
0	2	3	cuda	0.643223	0.239489	0.403734
1	2	3	cpu	0.000703	0.000249	0.000454
2	2	6	cuda	0.003000	0.000912	0.002088
3	2	6	cpu	0.001081	0.000262	0.000819
4	2	8	cuda	0.003379	0.000882	0.002497
..
81	3	4341	cpu	0.711151	0.002582	0.708568
82	3	6141	cuda	0.883403	0.003065	0.880338
83	3	6141	cpu	1.156598	0.003420	1.153178
84	3	8684	cuda	2.048458	0.003948	2.044510
85	3	8684	cpu	1.940628	0.005532	1.935096

[86 rows x 9 columns]

Plotting the time taken versus the matrix size gives us the following:

```
[8]: fig=plt.figure(figsize=(9, 8), dpi= 160, facecolor='w', edgecolor='k')
g = sns.scatterplot(x="matrix_size", y="total", hue="avg_connectivity",
style="processor", size="num_nodes", data=df)
```



- The avg_connectivity represents the average number of connections each node has, as the average connectivity increases, so does the complexity and density of the matrix.
- processor denotes which processor was used to compute the solution, either the GPU or the CPU
- The matrix size is simply the height or width of the A matrix
- total denotes the total computation time required to solve the linear system, including data transfer to each respective processor

7 Analysis and Conclusion

As we can see, oftentimes the GPU is far faster than the CPU. However, at very large matrix sizes this becomes problematic, and the data transfer overhead becomes far more dominant than the linear solver. As a result, the CPU becomes more efficient, as the GPU must pull memory from

main memory before performing computations, which is a costly task.

Another problem is due to the large size of some matrices, and the much smaller amount of memory on the GPU (4GB) versus main memory (16GB), the memory cost of parallelization may not be worthwhile. As a program is parallelized, the memory costs increase (since each independent procedure must obtain its own version of the code and initialize its own local variables), and so obtaining a GPU which contains more memory may be worthwhile.

The Python wrapper may also have some overhead, and could be more efficient. Using a lower-level package may be more complicated but result in faster computations. However, comparing Python GPU packages is outside the scope of this project.