Fall 2022 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

GPU parallel implementation of GMRES for circuit simulation

Sijia Zhou

December 14, 2022

Abstract

In this project, I accomplished generating circuit matrices based on SPICE (circuit emulator) netlist files and solved circuit matrices using GMRES iterative method on GPU. I also compared performance of GMRES with dense format storage and sparse format storage.

Link to Final Project `git` repo: https://git.doit.wisc.edu/SZHOU359/finalproject759

Contents

1. General information

In this short section, please provide only the following information, in bulleted form (four bullets) and in this order:

1. Your home department: ECE
2. Current status: undergrad/**MS**/PhD student (choose one)
3. Choose one of the following two statements (there should be only one statement here):
    - **I release the ME759 Final Project code as open source and under a BSD3 license for unfettered use of it by any interested party.**
    - I am not interested in releasing my code as open-source code.

IMPORTANT NOTE: For bullet 3 above, your choice does not affect in any way the score for your Final Project. It will only tell me that sharing your code in the future is ok.

2. Problem statement

What I wanted to accomplish is to parallelize the analog electronic circuit simulation using the GMRES algorithm.

My ultimate goal is to accelerate the SPICE circuit simulation. This is related to my area of interest as I need to use SPICE to do simulation for power electronic circuits or analog circuits in general. When the scale of the circuit increases, the scale of the corresponding matrix grows in proportion. The situation becomes worse when doing transient analysis the non-linear circuits. SPICE needs to create equivalent linear models for the non-linear circuit devices (ex: MOSFET) by guessing its operating points. Nodal voltages at one instant are solved by guessing and iterating several cycles until the solution is converged. Each iteration will generate a matrix corresponding to a new circuit model. Imagine the number of large-scale matrices we need to solve to obtain some voltage waveforms.

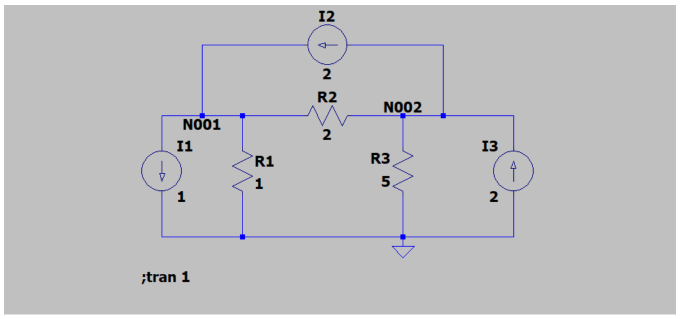
Accelerating matrix solving is obviously an effective way to accelerate the circuit simulation. SPICE takes LU factorization as its linear solver. Just as other direct solving methods, the LU factorization algorithm has time complexity of $O(n^3)$. Some research has been done for paralleling LU factorization while leveraging sparse and symmetric properties of circuit matrices [1] [2].

I am more interested in paralleling GMRES on GPU to solve circuit matrices in this project. I chose the iterative method as the circuit matrix is usually sparse and diagonal-dominant, which means that the iterative solving method should converge fast. And iterative methods generally have lower complexity and efficient for large scale matrix solving. GMRES has complexity of $O(n)$ for n-dimensional sparse matrices.

3. Solution description

3.1 Circuit matrix

SPICE can simulate the circuit described by the netlist file as shown in Figure 3.1. The netlist will record information including the name of the element, two nodes the element connects to and the value of the element.



(a)

```
I1 N001 0 1
R1 N001 0 1
R2 N002 N001 2
R3 0 N002 5
I2 N002 N001 2
I3 0 N002 2
;tran 1
.backanno
.end
```

(b)

Figure 3.1. (a) A SPICE circuit schematic from LTSPICE (b) The netlist equivalent

SPICE generates the circuit matrix based on the Kirchhoff's current laws. According to the Kirchhoff's current laws, the sum of currents flowing out of a node equal to currents flowing into the node. The circuit matrix can be generated based on the Kirchhoff's current laws.

$$\begin{bmatrix} \frac{1}{R_1}+\frac{1}{R_3} & -\frac{1}{R_1} & -\frac{1}{R_3} \\ -\frac{1}{R_1} & \frac{1}{R_1}+\frac{1}{R_2} & -\frac{1}{R_2} \\ -\frac{1}{R_3} & -\frac{1}{R_2} & \frac{1}{R_2}+\frac{1}{R_3} \end{bmatrix} \begin{bmatrix} V_0 \\ V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} I_1 - I_3 \\ -I_1 + I_2 \\ -I_2 + I_3 \end{bmatrix} \quad \text{(Eqn. 3.1)}$$

We can spot a pattern in the conductance matrix. The main diagonal is the sum of the conductance of elements which connect to the node. The off diagonal is the negative conductance of the element between two nodes. For the current matrix, when the current is flowing out of the node we add the negative current value to the entry with the row representing the node index. When the current is flowing in, we add the positive current value. Therefore, we can parse the netlist and generate a circuit matrix based on this pattern.

The augmented circuit matrix is 3 by 3 in our example but it only has the rank of 2. In fact, the n+1 by n+1 circuit matrix produced by the method described in the last paragraph always have rank n. The physic explanation of why the augmented circuit matrix has rank of n instead of n+1 is that a circuit model needs a reference voltage so the voltages of the rest of nodes can be computed. In our case, the reference voltage is ground at the node 0. Therefore, we need to remove the first row and the first column in the conductance matrix to produce an invertible matrix. The linear equation we actually solve is shown in Eqn. 3.2.

$$\begin{bmatrix} \frac{1}{R_1}+\frac{1}{R_2} & -\frac{1}{R_2} \\ -\frac{1}{R_2} & \frac{1}{R_2}+\frac{1}{R_3} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} -I_1 + I_2 \\ -I_2 + I_3 \end{bmatrix} \quad \text{(Eqn.3.2)}$$

The code structure to generate the final conductance matrix and the current matrix is described as follows. The parseNetlist function in parse.h parses the netlist file and generate an array of struct Element. Struct Element records the information of the two connected nodes, the element name and value for each circuit

element. Then we generate the augmented matrix using the elementList_to_augmented_Matrix function in the matrix_helper.h to generate the augmented matrix and decrease the matrix size by 1 in the augmented_Matrix_to_definite_matrix function.
In order to test the performance of the matrix solving method, we also need to generate random sets of elements and construct matrices that typically represents a circuit system. The random matrix is generated in the rand_resistor_circuit_model function in matrix_helper.h. The size of the matrix can be decided by the input of the number of nodes. Firstly, the function generates a random circuit element at every node and connects it the next node. It should be noted that the circuit loop should be formed otherwise it's not circuit system. Therefore, the first node is connected to the last node by a resistor so the head is connected to the tail. It should also be noted that the current on one branch should be the same, which means two current sources can't be on the same branch. Therefore, whenever a current source is randomly generated at the node, another branch with a resistor is added.
In this project, we will only consider circuit models containing only resistor and currents. As it does not contain non-linear devices, we don't need to iteratively guess operating points and solve multiple matrices to get nodal voltages. However, the randomly generated matrix is equivalent to typical circuit models and can indicate the simulation performance for non-linear circuits.

3.2 GMRES

I did two implementations of GMRES on GPU. One is with dense storage based on the cublas library. The other is with CSR and BSR storage based on the cusparse library and cublas library.
There are two main steps in each iteration for the GMRES algorithm. The first step is finding orthonormal vectors for the nth Krylov subspace.
The linear equation system is denoted as $Ax = b$. The nth Krylov subspace for the system is

$$K_n = K_n(A, r_0) = span\{r_0, Ar_0, A^2r_0, \dots, A^{n-1}r_0\} \qquad \text{(Eqn. 3.3)}$$

Where $r_0 = b - Ax_0$ and x_0 is guessed at the start of iterations.
The Arnoldi iteration is used to find orthonormal vectors and its MATLAB code [3] is shown below.

```
% h, q are vectors, A and Q are matrices, k is an integer
1 function [h, q] = arnoldi(A, Q, k)
2   q = A*Q(:,k);    % Krylov Vector
3   for i = 1:k      % Modified Gram-Schmidt, keeping the Hessenberg matrix
4     h(i) = q' * Q(:, i);
5     q = q - h(i) * Q(:, i);
6   end
7
8   h(k + 1) = norm(q);
9   q = q / h(k + 1);
10 end
```

Figure 3.2 The code snippet of Arnoldi iteration [3]

Line 2 in Figure 3.2 is a matrix vector multiplication which can be parallelized by the cublasDgemv function in the cublas library or the cusparseDbsrmv function in the cusparse library. Line 3 to Line 6 is a loop however the loop can't be parallelized as h(i) is dependent on the vector q in the previous iteration. With in the loop, line 4 can be parallelized with cublasDdot and cublasDaxpy in the cublas library. Line 8 and 9 can be implemented by cublasDnrm2 and cublasDscal in the cublas library. However, the awakard part is that h(k+1) needs to be inversed first before putting in the cublasDscal function. The inversion can be done either on CPU or on GPU with a custom kernel. By trial and error, the overall running time of a custom kernel are longer than doing inversion on CPU with the usage of unified memory and I chose to do inversion on CPU in my final implementation.

Apart from orthonormal vectors in the matrix Q_n, the Arnoldi process also gets the upper Hessenberg matrix $\widetilde{H}_n$. The result of first step is needed for the second step, solving the least square problem. To get the minimum residual, y_n in Eqn.3.4 needs to be computed.

$$\|r_n\| = \|b - Ax_n\| = \left\| \|r_0\| e_1 - \widetilde{H}_n y_n \right\| \quad \text{(Eqn.3.4)}$$

Where $e_1 = (1,0,\ldots,0)^T$.

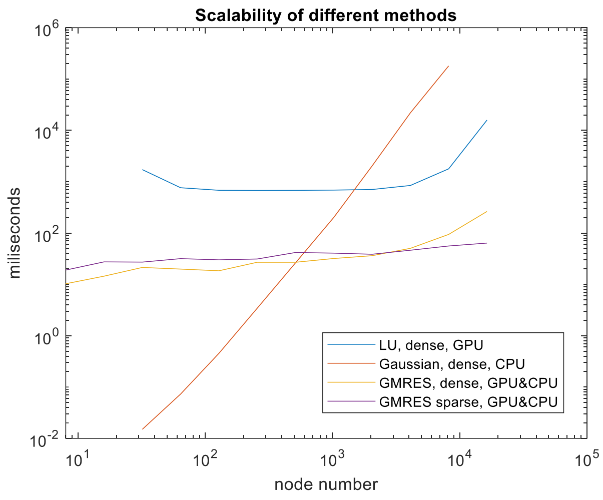
y_n is solved by rotating the upper Hessenberg matrix $\widetilde{H}_n$ to a triangular matrix. As the Hessenberg matrix differs only by the last column in every iteration, the rotation is applied on the last column and is done sequentially in the loop from Line 3 to 7 in Figure. Therefore, the code in Figure 3.3 can't be parallelized and is done on CPU.

```
1  function [h, cs_k, sn_k] = apply_givens_rotation(h, cs, sn, k)
2    % apply for ith column
3    for i = 1:k-1
4      temp   =  cs(i) * h(i) + sn(i) * h(i + 1);
5      h(i+1) = -sn(i) * h(i) + cs(i) * h(i + 1);
6      h(i)   = temp;
7    end
8    % update the next sin cos values for rotation
9    [cs_k sn_k] = givens_rotation(h(k), h(k + 1));
10   % eliminate H(i + 1, i)
11   h(k) = cs_k * h(k) + sn_k * h(k + 1);
12   h(k + 1) = 0.0;
13   end
```

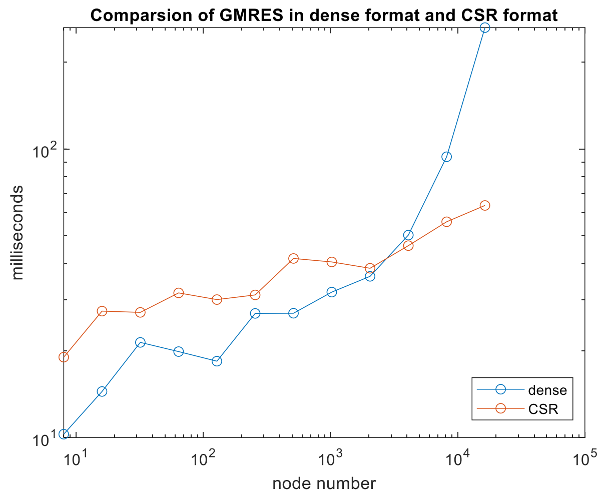
Figure 3.3 The code snippet of rotating Hessenberg matrix [4]

In this paragraph, I will discuss about the influence of the storage format on the implementation of GMRES. The profiling result of task_gmres.cu shows that the performance of GMRES implementation with a dense storage format is limited by host to device memory copy, the GPU page faults and cublas gemv functions (matrix-vector multiplication). The CSR and BSR storage format should alleviate this problem. In this project, the dense matrix is firstly generated from an array of struct elementList and then it's converted to the CSR form on CPU and is converted to the BSR form on GPU. The storage space for the CSR form has a magnitude of non-zero elements. In this project, if there are 10000 nodes, the corresponding matrix to be solved is almost 10000 by 10000 and the non-zero elements in the matrix is at most 1.8*10000*3 according to the implementation of the rand_resistor_circuit_model function in matrix.h. The memory size of CSR format copied from host to memory is 1.8*10000*3*3*8 bytes compared to 10000*10000*8 bytes. Therefore, CSR format should largely decrease memory operation time. Besides, the gemv function for the dense format is replaced by the bsrmv function in the cusparse library with the inputs in the BSR format. The time for matrix-vector multiplication is no longer a dominant reason limiting the speed of GMRES. Except the matrix-multiplication, all the other functions are still kept unchanged as the cublas function for the CSR format implementation.

4. Overview of results. Demonstration of your project



(a)



(b)

Figure 4.1 The time scalability of different methods

The results of Figure 4.1 and 4.2 are based on random generation of invertible circuit matrices with a fixed sparsity at a given node number. The sparsity of random matrices decrease with the number of nodes (i.e. the matrix size). The tests on GMRES has a error threshold of 0.001, which means the result is assumed to converge if the error is less than 0.001. However, if the error does not fall below the threshold after 20 iterations, the solving will stop in these tests. According to [4], GMRES will usually converge with tens of iterations. For the GMRES with the CSR format inputs, the conversion to the BSR format with a block dimension of 2 will be done on GPU before the actual GMRES algorithm.

It can be seen from Figure 4.1 (a) that GMRES is much faster than gaussian elimination on CPU and LU factorization on GPU on the matrix size becomes larger, which demonstrates the potential of circuit simulation acceleration.

From Figure 4.1 (b) it can be seen that sparse format storage outperforms the dense format in large scale matrices. It is as expected because the memory movement is decrased and also does unncessary floating points multiplications. The reason why the sparse format storage is slower is probably due to the overhead of converting CSR format to BSR format on the GPU.

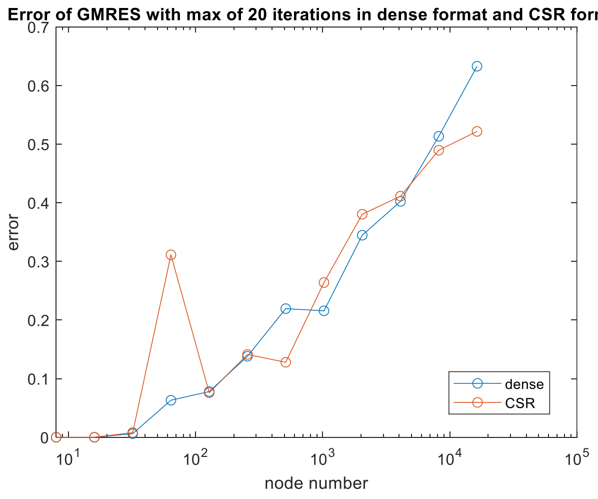


Figure 4.2 Residual errors of GMRES with max of 20 iterations in dense and CSR format

However, the problem of GMRES is that it can’t converge to the required accuracy within 20 iterations. It can be probably be improved with a proper preconditioner.

5. Deliverables:

Code structure:
Parse.h: the functions to parse the SPICE netlist files and create an array of struct elementlist
Matrix_helper.h: construct circuit matrices from an array of struct elementlist; randomly generate an array of elementlist; convert dense matrix format to CSR matrix format
gmres.h: all functions related to dense format and CSR format gmres implementation
task_gmres.cu: dense format GMRES with a random matrix
task_gmres_CSR.cu: CSR format GMRES with a random matrix
task_gmres_CSR-spice.cu: CSR format GMRES with the SPICE netlist file

Input files:
Draft1.txt Draft2.txt Draft3.txt (SPICE netlists) or randomly generated circuit matrices

Output files:
The nodal voltages, which are the solution of the circuit matrix, are generated in nodal_voltages.out files

How to run my code on euler:
sbatch task_gmres.sh, (./task_gmres [size of random matrix])
sbatch task_gmres_CSR.sh(./task_gmres [size of random matrix])
sbatch task_gmres_CSR-spice.sh(./task_gmres [Draft3.txt (spice netlist file)])

6. Conclusions and Future Work

Summarize the lessons learned and the highlights of your project work. Explain what remains to be done in the future, and how hard it would be to accomplish what is left at this point.
Finally, point out how the project has leveraged the ME759 material.
In this project, I implemented GMRES on GPU to solve circuit matrices in the CSR format. It demonstrates that GMRES can potentially accelerate circuit simulation and the CSR format GMRES outperforms the dense format for large-scale matrices. What remains to be done in the future is to find a proper preconditioner which can accelerate convergence and easy to compute in parallel on GPU.
I leveraged what I learnt from the ME759 material. I ran nvprof to test my code and found the memory bottleneck of the dense format implementation. I put the functions from two librarys, cublas and cusparse, in the same stream so that the kernels can be executed in sequence without calling cudaDeviceSynchronous.

References

[1] K. He, S. X. . -D. Tan, H. Wang and G. Shi, "GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 3, pp. 1140-1150, March 2016, doi: 10.1109/TVLSI.2015.2421287.
[2] N. Kapre and A. DeHon, "Parallelizing sparse Matrix Solve for SPICE circuit simulation using FPGAs," 2009 International Conference on Field-Programmable Technology, 2009, pp. 190-198, doi: 10.1109/FPT.2009.5377665.
[3] Wikipedia, “Generalized minimal residual method”, wikipedia.org,

https://en.wikipedia.org/wiki/Generalized_minimal_residual_method [12/14/2022]
[4] Raphaël Couturier, Stéphane Domas. Sparse Systems Solving on GPUs with GMRES. Journal of Supercomputing, 2012, 59 (3), pp.1504-1516. ffhal-00644456f https://hal.archives-ouvertes.fr/hal-00644456/document