User Guide for GLU V3.0

Sheldon Tan
University of California, Riverside
contact: stan@ece.ucr.edu

December 9, 2019

Contents

1 Introduction

GPU accelerated LU factorization (GLU) method is a sparse LU solver on GPUs for circuit simulation and more general scientific computing. It is based on a hybrid right-looking LU factorization algorithm [1], which is highly efficient on the GPU platforms.

Compared to the GLU 1.0, there are several improvements: First, we fix a few bugs of the numerical factorization. One bug is related to the dependency detection issue, which can cause the inaccuracy in the factorized matrices. As a result, the dependency detection and level prediction codes have been rewritten, in which new dependencies can be easily added to generate level information.

Second, similar to many commercial LU factorization solver, we also added the HSL MC64 algorithm to the pre-process phase (which make the diagonal elements dominant) to improve the numerical stability of the LU factorization process.

Third, to further improve the numerical stability, numerically factorized results will be checked to avoid Inf (infinity) or NaN (not a number) in the diagonals. If the Inf and NaN happens, the forced perturbation is employed (adding a small diagonal value) during the numerical factorization is carried out [2] in the new GLU 3.0.

With the the HSL MC64 algorithm and zero-diagonal element mitigation techniques, GLU 3.0 does not need the dynamic pivoting to ensure the numerical stability of the LU factorization, which makes it more amenable for the GLU kernel computing.

2 GLU flowchart

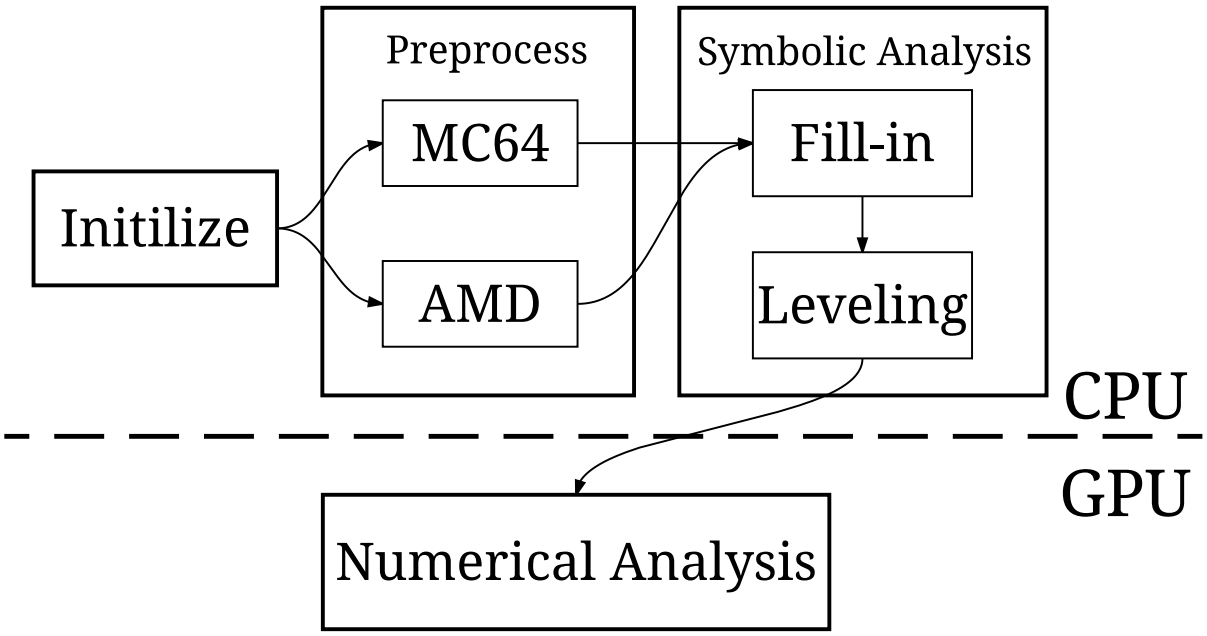

Figure 1: The flowchart of GLU V3.0

The flowchart of the GLU V3.0 is shown in Fig. 1. GLU compouting flow follows the similar flow of NICSLU [3, 4, 5]. The preprocess includes HSL MC64 algorithm [6] and AMD (Approximate Minimum Degree) algorithm [7] to ensure nonzero diagonal elements and to minimize the number of fill-ins. After that, the symbolic analysis is executed for the prediction of fill-in elements for better memory management

and independent level estimation. Finally, the numerical factorization can be deployed after the symbolic analysis.

3 GLU routines and their functions

In the following, we briefly explain the GLU basic routine functions in each sub-section.

3.1 Preprocess()

Preprocess function performs preprocess phase for the input matrices, including HSL MC64 algorithm and AMD algorithm to ensure dominant diagonal elements and to minimize the number of fill-ins. Please refer to NICSLU [4] for the details of the preprocess.

3.2 Symbolic_Matrix

Symbolic_Matrix initializes the class for the symbolic matrix, which contains new fill-ins after the symbolic prediction, level information for numerical factorization. The data are stored in CSC format. We also use CSR format to store the position information to accelerate the level prediction.

3.3 Symbolic_Matrix::fill_in()

This function symbolically factorizes the matrix after preprocessing, where all the positions of fill-ins and non-zero elements are assigned with non-zero initial values and their memories are allocated.

3.4 Symbolic_Matrix::csr()

Csr function generates the CSR information for the symbolic matrix, in which the position information can be used to accelerate the level prediction.

3.5 Symbolic_Matrix::predictLU()

PredictLU function generates and allocates all values for the fill-ins and non-zero elements. It will finish all information for the symbolic matrix with Symbolic_pivot function.

3.6 Symbolic_Matrix::leveling()

This function generates level information based on column dependencies.

3.7 LUonDevice()

LUonDevice runs on GPU to do numerical factorization of the given sparse matrix, which contains the main GLU kernal functions.

3.8 Symbolic_Matrix::solve()

This function solves the linear system $Ax = b$ given right-hand-side b. For the purpose of demonstration, the default behavior is that the solver uses an all-one b is to solve and write the results to disk in x.dat.

Table 1: GPU kernel runtimes of GLU3.0 against previous works

Matrix	Number of rows	nz	nnz	GPU time (ms)			
				GLU2.0	GLU3.0 (this work)	speed-up over GLU2.0	speed-up over [9]
rajat12	1879	12926	13948	2.44883	2.237	1.1	1.0
circuit_2	4510	21199	32671	8.36301	4.144	2.0	1.9
memplus	17758	126150	126152	6.90432	6.672	1.0	0.9
rajat27	20640	99777	143438	23.8673	10.539	2.3	2.0
onetone2	36057	227628	1306245	550.598	60.964	9.0	8.3
rajat15	37261	443573	1697198	458.611	71.135	6.4	6.1
rajat26	51032	249302	343497	104.12	32.366	3.2	4.2
circuit_4	80209	307604	438628	394.995	68.944	5.7	9.1
rajat20	86916	605045	2204552	2538.24	241.822	10.5	8.8
ASIC_100ks	99190	578890	3638758	2652.79	215.493	12.3	14.1
hcircuit	105676	513072	630666	243.846	46.996	5.2	9.5
Raj1	263743	1302464	7287722	7969.05	845.189	9.4	8.7
ASIC_320ks	321671	1827807	4838825	5632.8	216.517	26.0	21.3
ASIC_680ks	682712	2329176	4957172	11771.7	210.697	55.9	18.4
G3_circuit	1585478	4623152	36699336	38780.9	878.153	44.2	8.2
					Arithmetic mean	13.0	7.1
					Geometric mean	6.7	4.8

4 Code compilation and usage

GLU can be compiled in the Linux platforms and used with most recent Nvidia GPUs. To compile GLU on Linux, a C++ compiler with support of C++11 is required. Just type "make" in the "./src" directory. After compiling successfully, we can run GLU in the "src" folder or any other folder.

The basic usage is ./lu_cmd -i inputfile", where inputfile is a sparse matrix file with ".mtx" (there is one example matrix called "add32.mtx" in the "./src" folder).

5 Performance

Experiments are carried out on a Linux server with Intel(R) Xeon(R) CPU E5-2698 v3, 128GB RAM, and NVIDIA GTX TITAN X (3072 CUDA cores, 12GB GDDR5 memory, Maxwell architecture). The program is compiled with -O3. A set of typical circuit matrices are obtained from the UFL sparse matrix collection [8] as the benchmark matrices.

We test the execution time of GPU parts of our GLU, which is shown in Table 1. All matrices are sorted by the number of nonzero elements after symbolic factorization.

References

[1] K. He, S. X.-D. Tan, H. Wang, and G. Shi, "Gpu-accelerated parallel sparse lu factorization method for fast circuit analysis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 3, pp. 1140–1150, 2016.

[2] X. S. Li, J. W. Demmel, J. R. Gilbert, L. Grigori, M. Shao, and I. Yamazaki, "Superlu users guide," *Lawrence Berkeley National Laboratory Tech. Report, LBNL-44289*, 1999.

[3] X. Chen, Y. Wang, and H. Yang, "An adaptive lu factorization algorithm for parallel circuit simulation," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pp. 359–364, IEEE, 2012.

[4] X. Chen, Y. Wang, and H. Yang, "Nicslu: An adaptive sparse matrix solver for parallel circuit simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 2, pp. 261–274, 2013.

[5] X. Chen, W. Wu, Y. Wang, H. Yu, and H. Yang, "An escheduler-based data dependence analysis and task scheduling for parallel circuit simulation," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 58, no. 10, pp. 702–706, 2011.

[6] I. S. Duff and J. Koster, "The design and use of algorithms for permuting large entries to the diagonal of sparse matrices," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 4, pp. 889–901, 1999.

[7] P. R. Amestoy, T. A. Davis, and I. S. Duff, "Algorithm 837: Amd, an approximate minimum degree ordering algorithm," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 3, pp. 381–388, 2004.

[8] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[9] W.-K. Lee, R. Achar, and M. S. Nakhla, "Dynamic gpu parallel sparse lu factorization for fast circuit simulation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, no. 99, pp. 1–12, 2018.