User Guide for GLU V2.0

Lebo Wang and Sheldon Tan

University of California, Riverside

June 2017

Contents

Chapter 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 [7], which is highly efficient on the GPU platforms.

Compared to the GLU 1.0, there are several imppovement: 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 [8] in the new GLU 2.0.

With the the HSL MC64 algorithm and zero-diagonal element mitigation techniques, GLU 2.0 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.

Chapter 2

Using GLU within a C/C++ program

2.1 GLU flowchart

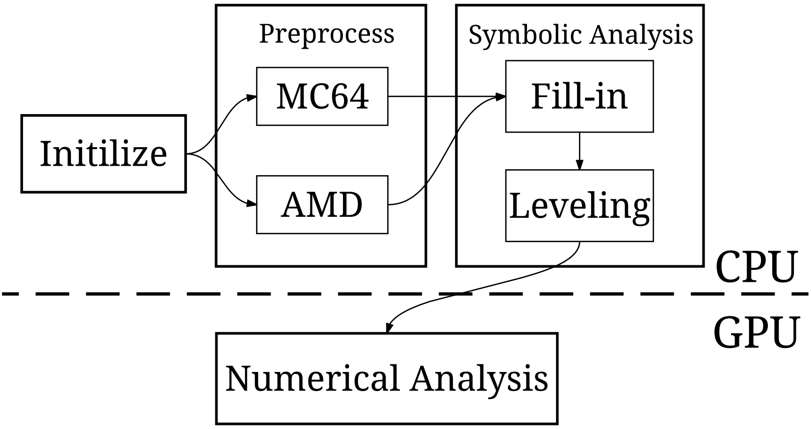

Figure 2.1: The flowchart of GLU V2.0

The flowchart of the GLU V2.0 is shown in Fig. 2.1. GLU compouting flow follows the similar flow of NICSLU [2, 3, 4]. The preprocess includes HSL MC64 algorithm [6] and AMD (Approximate Minimum Degree) algorithm [1] 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.

2.2 GLU routines and their functions

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

2.2.1 Preprocess

Preprocess function prefomrs 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 [3] for the details of the preprocess.

2.2.2 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.

2.2.3 Symbolic_Matrix.symbolic_pivot

Symbolic_pivot 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.

2.2.4 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.

2.2.5 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.

2.2.6 Symbolic_Matrix.U_W_leveling

U_W_leveling function generates level information including U-shape and W-shape dependencies.

2.2.7 Symbolic_Matrix.restore

Restore function resets the value of the symbolic matrix. It must be executed every time before doing the numerical factorization.

2.2.8 LUonDevice

LUonDevice runs on GPU to do numerical factorization.

Chapter 3

Compilation and test

GLU can be compiled in the Linux platforms and used with most recent Nvidia GPUs.

To compile GLU on Linux, gcc is required. Just type "make" in the "./src" directory. Make sure that the "preprocess" function has already been compiled. The "preprocess" function is in the "./ src/preprocess" folder. If you need to recompile the "preprocess" function, just type "make" in the "./src/preprocess" 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). If perturbation is needed, add "-p" like "./lu_cmd -i inputfile -p".

Chapter 4

Performance

Experiments are carried out on a Linux server with two 8-Core Xeon E5-2670 CPUs, DDR3-1600 64-GB memory. The server also consists of one K40c GPU, which serves as the GPU platforms for the GLU. A set of typical circuit matrices are obtained from the UFL sparse matrix collection [5] as the benchmark matrices.

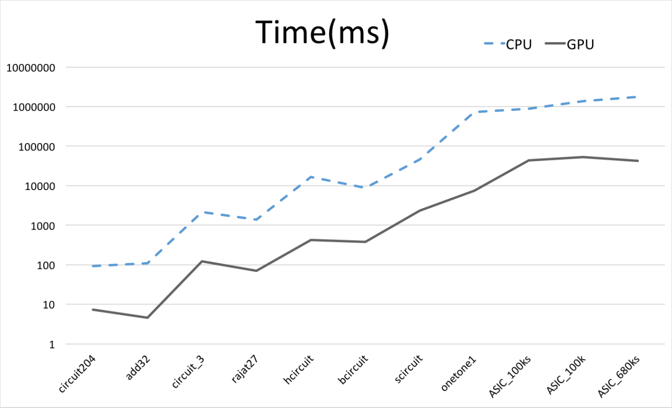

Figure 4.1: Performance of GLU

We test the execution time of both CPU and GPU parts of our GLU, which is shown in Fig.4.1. All matrices are sorted by the number of nonzero elements after symbolic factorization.

Bibliography

[1] Patrick R Amestoy, Timothy A Davis, and Iain S Duff. “Algorithm 837: AMD, an approximate minimum degree ordering algorithm”. In: *ACM Transactions on Mathematical Software (TOMS)* 30.3 (2004), pp. 381–388.

[2] Xiaoming Chen, Yu Wang, and Huazhong Yang. “An adaptive LU factorization algorithm for parallel circuit simulation”. In: *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*. IEEE. 2012, pp. 359–364.

[3] Xiaoming Chen, Yu Wang, and Huazhong Yang. “NICSLU: An adaptive sparse matrix solver for parallel circuit simulation”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.2 (2013), pp. 261–274.

[4] Xiaoming Chen et al. “An escheduler-based data dependence analysis and task scheduling for parallel circuit simulation”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 58.10 (2011), pp. 702–706.

[5] Timothy A Davis and Yifan Hu. “The University of Florida sparse matrix collection”. In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), p. 1.

[6] Iain S Duff and Jacko Koster. “The design and use of algorithms for permuting large entries to the diagonal of sparse matrices”. In: *SIAM Journal on Matrix Analysis and Applications* 20.4 (1999), pp. 889–901.

[7] Kai He et al. “GPU-accelerated parallel sparse LU factorization method for fast circuit analysis”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.3 (2016), pp. 1140–1150.

[8] Xiaoye S Li et al. “SuperLU users’ guide”. In: *Lawrence Berkeley National Laboratory Tech. Report, LBNL-44289* (1999).