Accelerated Mesh Computations for ALE-FEM based 3D free Surface flows in GPU

Thivin Anandh[1]

Abstract— **This project focuses on using GPU to accelerate the mesh movement modules that comes in ALE based Finite Element method. In this paper, we propose strategies to calculate the mesh normals asynchronously in GPU by transferring the Finite element Data from CPU to GPU and to factorise the Mesh matrix in GPU asynchronously by overlapping that computation with the Navier Stokes computation in GPU. We have also discussed the timing difference between the direct solver and the sparse GPU solvers and the existing limitations in the lower level sparse Factorisation routines with less host interventions.**

I. INTRODUCTION

Arbitrary Lagrangian Based Finite Element method (ALE -FEM) is one of the reliable methods for Solving incompresible Navier Stokes equation while trying to accurately capture the interface. It captures the interface by moving the free surface layer with the velocity of the fluid at free Surface. In order to preserve the mesh quality , we need to move the interior points of the mesh accordingly by either solving the Poisson or linear elastic equation. However , this process is an additional step in CPU in solving the incompressible Navier stokes equation. Additionally , we need to impose constraints on the surface based on the nature of the problem , which further needs the computation of normals on the mesh surfaces. This process can be computed in the GPU given the finite element data structures are being transferred to the GPU . Further The Mesh Matrix can be solved in GPU itself using any iterative Solver routines or Factorisation Routines

II. MOTIVATION FOR PARALLELIZATION

The Motivation for parallelisation of these routines are listed below

- The mesh movement routines except for the solver routines is independent of the main NSE solver routine
- These routines , if properly channelised, can be transferred to the GPU and can be computed independent of CPU.
- The Normal calculation computations can be done in GPU such that they overlap the computation with CPU.

III. IMPLEMENTATION PREREQUISITES

To Aspects of parallelization of this process involves the following criterion

- Select a best Sparse Solver routine for Solving the system inside GPU.
- Figure out a way to transfer the complex FE data-structures from CPU to GPU using arrays to calculate the mesh normals.
- Modify existing algorithms to adapt the asynchronous mode of proposed CPU-GPU asynchronous algorithm.

IV. PROPOSED MODEL - I

In this model , We propose to use GPU as an accelerator to speed up the existing code by calculating the Mesh parameters in GPU and provide that Solution to the CPU to complete solution process (figure 1)

For this process, we need to figure out a way to transfer the FE data structure to GPU and choose an iterative or direct solver to compute the solution. Upon analysis of the time taken for all the process in the mesh modules 5 the solver process is taking most of the time. So it is necessary to find an efficient solver either iterative or direct solver on the GPU which could achieve an execution time lower than the CPU sparse algorithms.

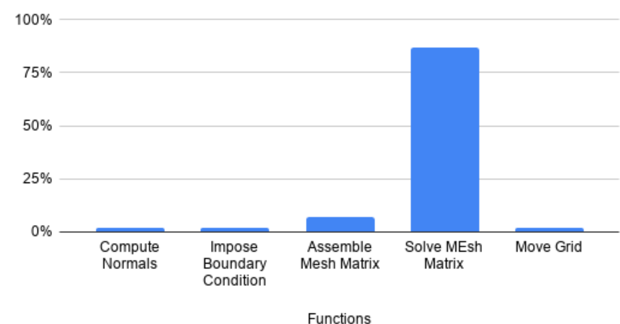

Fig. 2. Time Splitup - Mesh Modules

A. *Performance comparison of Sparse CPU solvers vs Sparse GPU Solvers*

CuSparse and CuSolver routines are used for calculation in these experiments. In case of sequential sparse solvers, we have used UMFPACK and Intel MKL Pardiso's Multifrontal Sparse direct solver libraries. For GPU, we choose Sparse QR for Direct Sparse Solvers and BiCGStab for iterative Solver Routines. For CuSparse QR, METIS reordering is performed in order to reduce the fill in's and increase speed up. The experiment is performed with the finite element matrix obtained from the meshing module which is a sparse matrix of size 14K $\times$ 14K with 997K non-zero elements.

*DS 295 - Parallel Programming - Course Project

[1]Thivin Anandh D - PhD Student of Department of Computational and Data Sciences, Indian Institute of Science, India

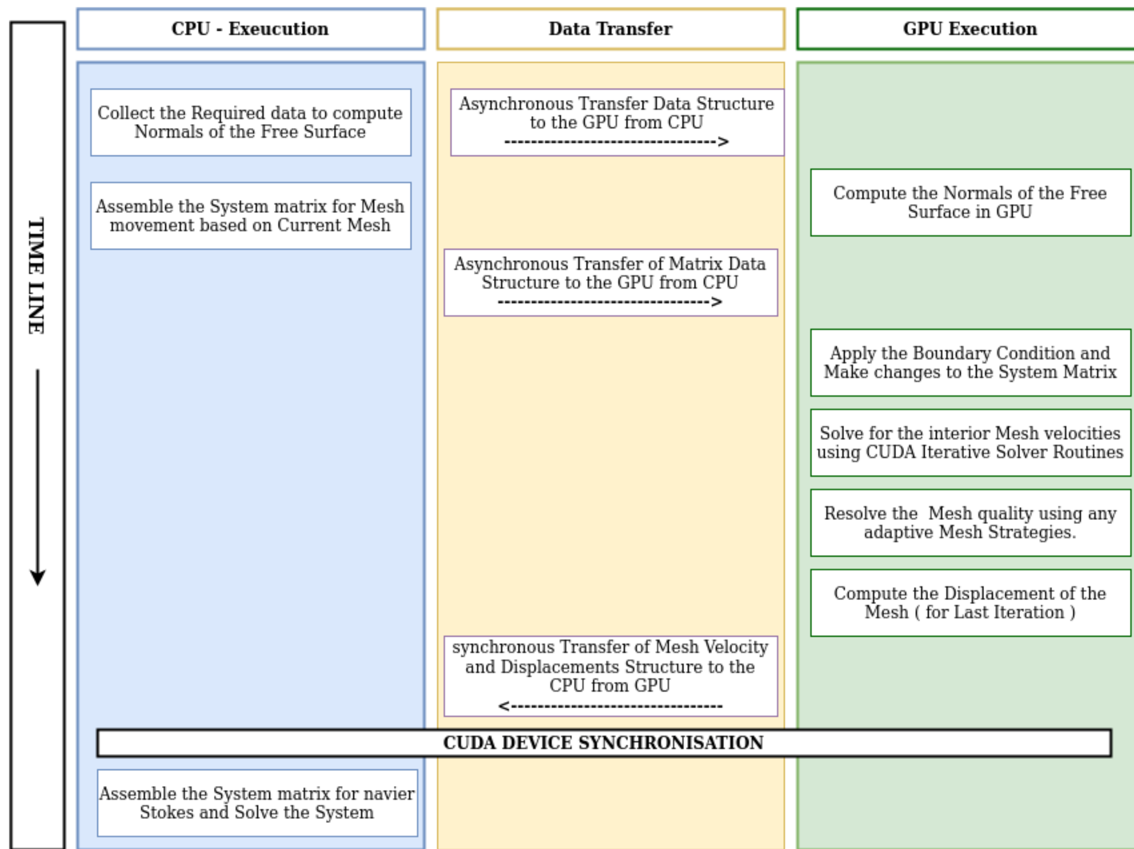

Fig. 1. Proposed Model I - Timeline

The experiments are performed on the system with following specifications mentioned in Table I.

TABLE I
CPU GPU SPECIFICATIONS

S.No	Item	Description
1	Processor	Intel(R) Xeon(R) Gold 6150
2	RAM	160 GB
3	Number of Cores	18
4	NUmber of Threads	36
5	Base Frequency	3.70 GHz
6	L3 Cache	25 MB

S.No	Item	Description
1	Graphic Processor	NVIDIA V100
2	Architecture	Volta
3	CUDA Cores	5376
4	Graphics DRAM	16GB
5	CUDA Cores/SM	64

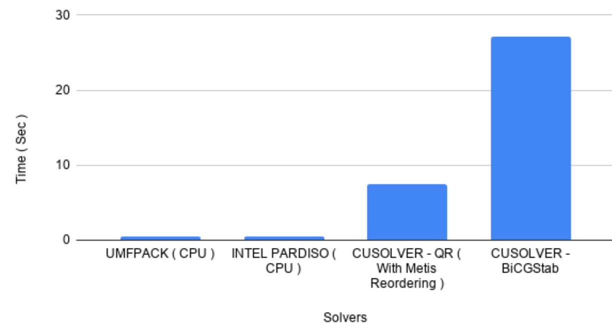

Fig. 3. Sparse CPU vs GPU Solvers

B. Inference

- As seen from the previous sections, solving the mesh matrix inside GPU to accelerate the computations is not a feasible option since the GPU execution times are way higher than the execution time of standard CPU libraries.
- We need to figure out other way of establishing parallelism apart from trying to solve the system in GPU which is discussed in the next section.

V. Proposed Model - II

Upon analysis of the mesh module, we could observe that, only the RHS of the mesh module (b) is dependent on the solution of the NSE equation whereas the matrix (A) is dependent only on the current state of the mesh . Using this, we propose the following model as shown in Figure 4.

VI. Parallel Implementation

A. *Implementation of FE Data Structures*

- Since the FE data structure is complex, we begin by filtering out only the required data necessary for our mesh computation in CPU and then saving them in relevant 1D arrays which can be sent to the GPU.
- In order for GPU to recognise those format's easily, we also create an mapping array for each element with which the GPU threads can identify on which location they need to pick their data from in the main FE arrays.
- Since the normal calculation for each type of boundary surface is different, different CUDA streams are created for each of them.
- All the data transfer related to those two process are sent asynchronously in their respective streams.
- The streams are synchronised, before the point where normal values are needed to complete the assembly of mesh system matrix.
- Multiple thread blocks are used to split the task of computing the normals.

B. *Implementation of Factorisation Routines*

There are no direct high level routines in CuSparse or CuSolver or Magma libraries that allows us to store the factors of the matrix so that the system can be solved in later course of time. However in CuSolver, there are some low level routines which can be used to obtain Q and R factors (LU low level routines are not available) and save it in the device, which can be used to solve the system later. There were two such factorisation routines available in CuSolver as below,

- Low-Level QR Factorisation
- LU Re-factorisation routines

C. *Performance of the Factorisation Routines*

The below plot shows the performance of the factorisation routines compared with the existing routines. It seems that the routines has more host intervention (host blocking calls) in order to complete the factorisation i.e. most of the prerequisite routines like pivoting, reordering, calculating the storage Space and checking singularity of matrix are being done in the host as blocking calls, which thwarts the speedup of the application.

D. *Performance Optimisation performed for QR Low Level Libraries*

In order to reduce the execution time, we tried to condense and eliminate most of the host dependent process. As part of the QR setup process, the following items are performed on the host for setup of QR (These are low level CuSolver Routines).

1) Obtain the matrix and perform reordering using METIS routines and obtain the reordering
2) Using the reordering compute the Permutation matrix of the given input Matrix
3) Create and allocate buffer(opaque data structure) for QR routines to operate on GPU and initialise the CUDA QR handle
4) Setup CUDA routine and compute the factorisation

However, the following process can be simplified by making the following logical assumptions. Since the FE mesh Matrix does not change its structure (Sparsity pattern) and the non-zero elements locations remains same, the Reordering will be same for all matrices and the same will be for permutation matrix. In that case, the memory (opaque data structure) needed by GPU to factorise will be same for every Iteration. These statements can lead us to the conclusion that we need to perform steps 1-4 (Completely executed on host) for the first iteration only , and after that we can save the data in such a way that we only need to perform step 4 for the upcoming matrices.

Note : This optimisation was not included in any of the example routines or any high/low level QR routines of CuSolver library. This was added as part of this project to reduce host intervention for our problem to increase parallelism.

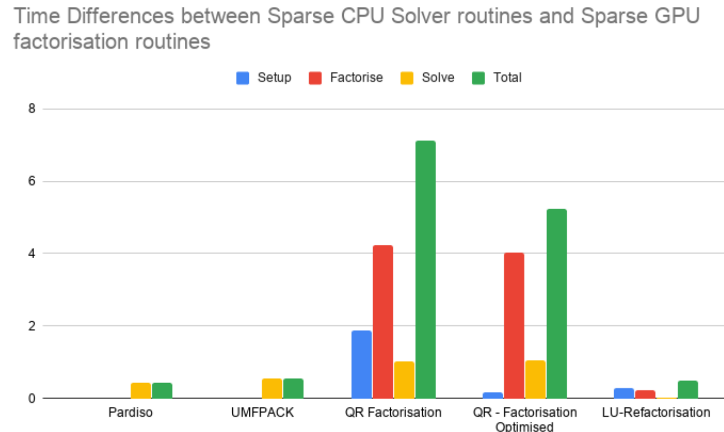

Fig. 5. Sparse CPU vs GPU Refactorisation Routines

E. *Inferences from analysis on GPU Refactorisation routines*

- The optimisation performed for QR routines showed a significant reduction in the setup phase of the routine
- The QR factorisation cannot be used for our setup. Though the factorisation time can be completely overshadowed by the CPU execution, the solving time of QR is alone higher than the CPU's total solver time. 9
- We can use LU-Refactorisation routines for our problem since that is the one that gives very close timing to our Sequential counterparts.

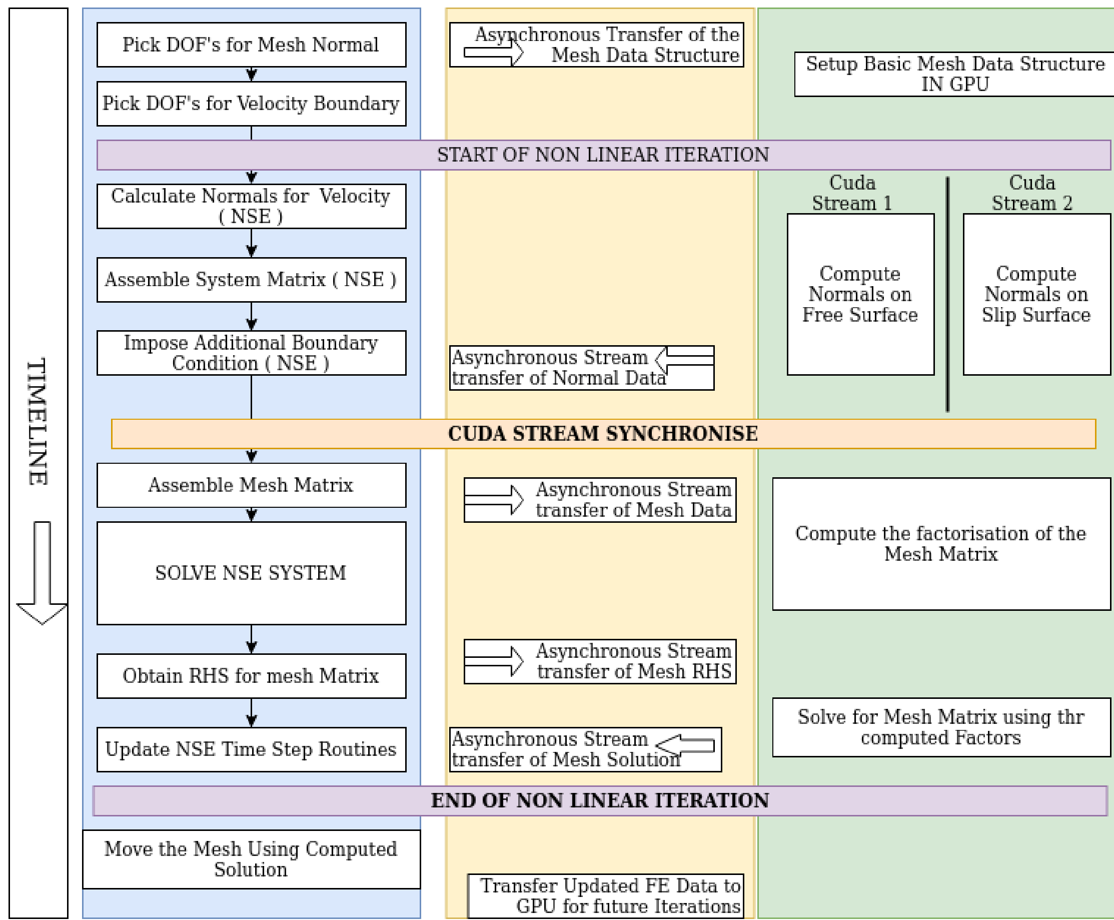

Fig. 4. Proposed Model II - Timeline

Note : The time taken for LU factorisation in CPU is not being included for calculation since it is done for one time at the start of the process

VII. SPEEDUP OBTAINED

The following figure shows the speed up of using CUDA threads for Computation of Normals in Sequential vs Parallel (Figure 9)

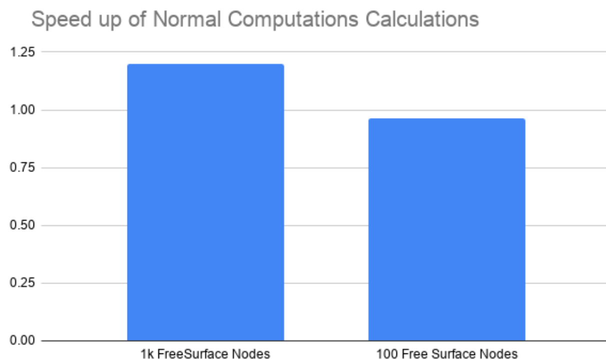

Fig. 7. Speedup - Normal Computation

The following figure shows the speedup of computations using CUDA Re-factorisation routines. (Figure 8)

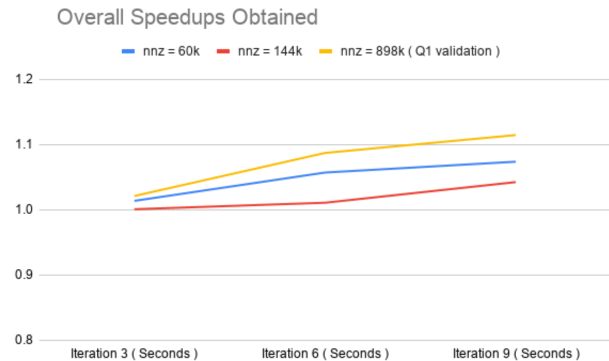

Fig. 8. Speedup - using CUDA Refactorisation

VIII. RESULT ANALYSIS

1) The lower speed up obtained at Computing mesh normals is due to the fact that , Finite element routines are mostly thread Divergent since that part involves more conditional statements like if and switch cases.
2) The achieved speedup was due to the fact that for non linear sub iterations only rhs changes at each sub step

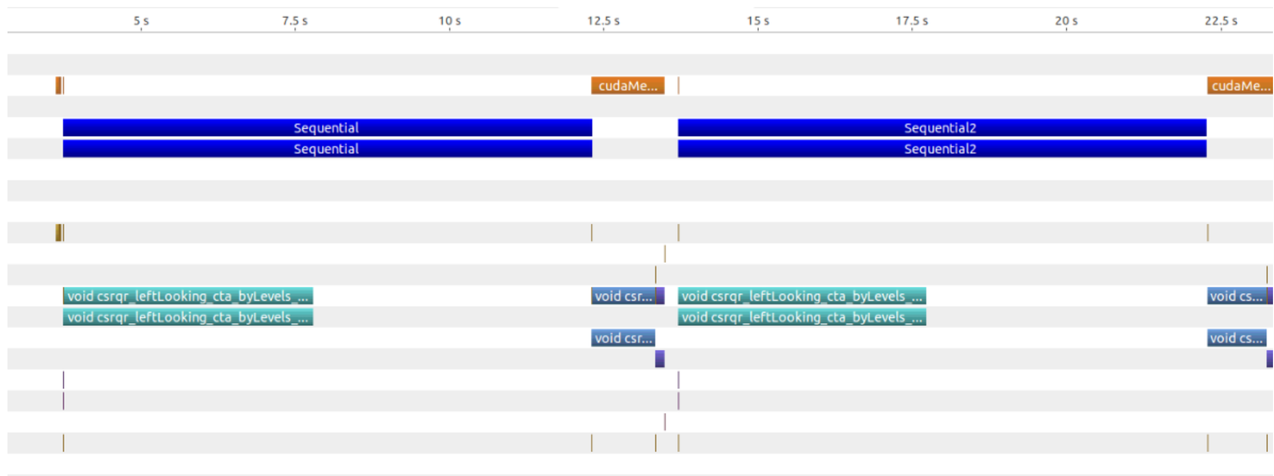

Fig. 6. NVProf - Timeline - QR Factorisation

there by eliminating the re factorisation routine and uses only solve routine at those steps.

3) Further analysis with Nvprof showed that the increased time is due to the fact that **the Re-factorization routine is executed in Blocking call rather than being executed as an Asynchronous kernel Call** (Figure 9)

Note : The Blocking behaviour of CudaRF routine is due to the fact that there is no provision given to incorporate a CUDA stream into an CUDA Refactor handle.[Source : CUDA Documentation and checked in cudaRefactor.h for that function]. This feature is available only for CuSparse (BLAS Routines) and Dense Matrix routines only. Upon Checking the NVIDIA Forum , this question regarding the usage of streams with CUDA RF routines is currently unanswered as of Jun-14-2020. URL : https://forums.developer.nvidia.com/t/why-cusolverrf-doesnt-support-stream-setting/66018

IX. CONCLUSIONS

- The Objective of this project was not only use the GPU as an accelerator to speed up the existing computation but also to asynchronously execute FE computations by overlapping them with the current CPU computations ,which was achieved
- The Partial import of FE Data Structures can be expanded further to transfer more FE data to GPU , which can be used for Solving Rigid Body equations in FSI Problems at GPU itself.
- If a non-blocking stream can be incorporated into CudaRF routines, then the Speedups can be massively increased.

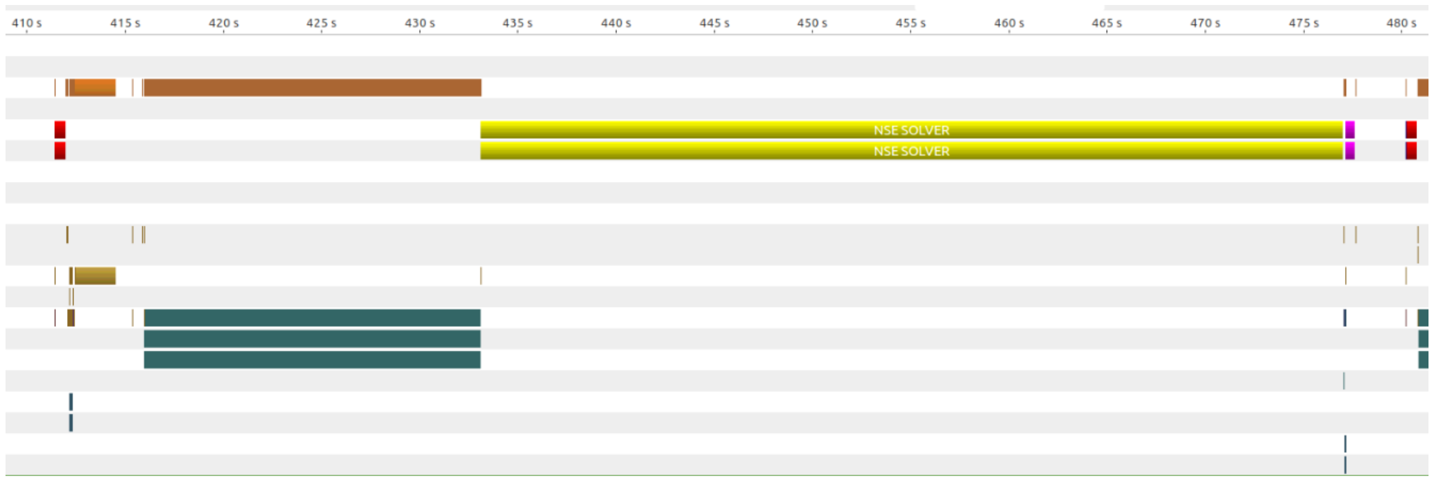

Fig. 9. NvProf - Timeline - CUDA Sparse LU Re-factorisation