Exploiting Symmetry in Elemental Computation and Assembly Stage of GPU-Accelerated FEA

*Subhajit Sanfui[1], †Deepak Sharma[1]

[1]Department of Mechanical Engineering, Indian Institute of Technology Guwahati, Assam, India

*Presenting author: s.sanfui@iitg.ac.in
†Corresponding author: dsharma@iitg.ac.in

Abstract

Finite element analysis (FEA) is one of the most popular numerical methods for solutions of boundary value problems in partial differential equations (PDEs). Owing to the computational load of handling fine meshes and complex geometries in many real-world problems coupled with the data-parallel and throughput-intensive nature of FEA computation, it is considered a highly suitable candidate for Graphics processing units (GPU) based acceleration. This work aims to accelerate the elemental stiffness calculation and assembly stages of FEA on a single GPU by exploiting the symmetric nature of the elemental and global stiffness matrix. The key idea of the implementation is to design GPU kernels that calculate and assemble only the lower triangular part of the local stiffness matrix into the global stiffness matrix. This leads to (a) reduced FLOP count and (b) reduced memory storage and access, which, in turn results in an overall reduced execution time. The elemental stiffness matrix is computed by assigning each GPU thread to one entry of the elemental stiffness matrix. The assembly stage is performed in two steps for computing the indices and values of the global stiffness matrix. For handling the race condition in assembly, mesh coloring and atomics-based approaches are implemented. The results from the proposed implementation is compared to a standard GPU-based version of the implementation that computes and assembles the complete elemental stiffness matrices. Hexahedral meshes with up to three million nodes are tested for the performance analysis. The proposed implementation is found to be approximately twice as fast as the standard implementation for both elemental calculation and assembly stages.

Keywords: Computation, Finite Element Method, Graphics Processing Units, Symmetry

Introduction

Finite Element Method (FEM) is a numerical method for approximating solutions of boundary value problems for partial differential equations (PDEs). It is extensively used in fields such as mechanical engineering, civil engineering, electrical engineering, medical applications. Due to its natural advantages such as flexibility, adaptability and ease of implementation even for complex geometries, it has become an integral part of a large variety of specializations. In industries like aviation, automotive and construction, FEM is usually an inherent part of the design process [1].

Although finite element analysis is widely popular in several fields, it often suffers from a high computational intensity, especially for real-world problems. This is primarily caused by the resolution of system of equations performed in the solver step [2]. Several applications [3][4] that make use of FEM, have reported it to be the most time consuming or computationally expensive part of the whole applications especially for larger and more complex geometries. Due to these reasons, many practical applications see the need for efficient parallelization of FEA. Among the several steps in a complete FEA, the numerical solver, global matrix assembly and elemental computation are the three most time-consuming and computationally expensive in nature. In the present work, the focus is kept on the elemental calculation and assembly into a global stiffness matrix by parallelization on a single GPU. For the numerical solver stage, multiple optimized libraries exist that can provide solution to the linear system of equation after application of proper boundary conditions.

Numerical Integration, as reported by many researchers, is a suitable candidate for GPU acceleration. For low order finite elements, the numerical integration becomes comparatively easy, whereas for high order elements, the computation turns equally tedious. This is because higher order elements necessitate higher number of Gauss points, which in turn increases the memory requirements while also increasing the amount of computation. It was demonstrated in [5], that using a 10th order Gaussian Quadrature, the numerical integration step requires 73% - 83% and 87% of the total time of matrix generation step on a CPU and GPU respectively. For low orders of approximations, on the other hand, values of shape functions and their derivatives may be pre-computed and stored for reuse since the necessary storage requirements would be sufficiently small. Such an implementation, sadly, for higher order elements is unrealizable with present hardware. The first work dedicated toward numerical integration on the GPU is by using the Gauss-Legendre Quadrature Method [6]. Authors demonstrated the complete scalability of the numerical integration process on the GPU. One limitation of this study was that it was conducted entirely in single-precision, which can pose serious problems for convergence in the solver stage. This is specifically true for iterative solvers such as CG solver, which are known to be notoriously sensitive to round-off errors. A maximum speedup of 20x was achieved for third order approximation on an NVIDIA GeForce GTX8800 compared to an AMD X2 at 2.4 GHz. Authors also concluded that the massive amount of parallelism was not fully realized due to the insufficient memory resources in individual threads. This finding was later supported by Dziekonski et al. [7], where several strategies on efficient generation and assembly of large finite-element matrices were presented, while maintaining the desired level of accuracy in numerical integration. Authors used higher-order curvilinear elements making the integration step more compute intensive. Again the method of Gauss Quadrature was used for analysis. A speedup of 2.5x was achieved on an NVIDIA Tesla C2075 over two 12-core AMD Opteron 6174 at 2.2 GHz. Among more recent works, Banas [8][9] addressed the problem of implementing numerical integration that is portable across several GPU architectures and different orders of approximation. This is in general difficult to achieve because of vastly varying memory size, memory hierarchy and computational resources available to the programmer across different GPU vendors such as NVIDIA, AMD and Intel. The OpenCL implementation in [8] achieved a maximum speedup of 4x tested on four GPUs with different architectures (NVIDIA GeForce GTX 580, Nvidia Tesla M2075, AMD HD5870 and AMD HD7950). The portable OpenCL implementation by Banas [9], targeting numerical integration for only first-order approximation, achieved a maximum speedup of approximately 9x when tested on a Tesla K20m, compared to an Intel Xeon E5 2620 CPU. Authors provided details on several optimization aspects for implementations on different target hardware, while also indicating the factors limiting the performance for different problem types on different architectures. Apart from these, many works concentrating on applications of FEA have also implemented numerical integration on the GPU [3][4][10]. However, in these cases, no relevant details on the implementation of numerical integration was provided.

There has been only a handful of works that target the assembly stage of FEA on the GPU, despite of the inherent parallelism in this step. This is primarily because of this step being significantly less compute-intensive than the matrix-solver step. In other words, even a small to medium speedup in the linear solver stage would benefit the FEA process more than a decent to good speedup in the assembly step. The first work concerning assembly on GPU is by Filipovic et al. [11], where a speedup of 15x was achieved on an NVIDIA GeForce GTX280 compared to an Intel Core2Quad Q9550 CPU at 2.83 GHz. One important observation by the authors was that using one single kernel for the entire computation results in massive under-utilization of the GPU. Cecka et al. [2] has studied several aspects of assembly on an unstructured mesh using single precision arithmetic. Different algorithms for efficient use of global, shared and local memory available on the GPU along with methods to achieve memory coalescing are introduced by the authors. Four different implementations were analyzed for assembly. These are respectively, assembly by elements using graph-coloring, by NZ using shared memory, by NZ using local memory and by NZ using global memory. Among these, the first two were shown to be the most efficient. A speedup of about 35x was achieved for the best implementation on an NVIDIA GTX8800 compared to an Intel Core2 Quad Q9450 CPU at 2.66 GHz. Later, Dziekonski et al. [5] used an NVIDIA Tesla C2075 to accelerate both numerical integration and matrix assembly and achieved combined speedups of 81x and 19x over single and multi-threaded implementations respectively on a 12-core Opteron 6174. Markall et al. [12] have discussed several assembly strategies on multi-core and many-core architectures. Among more recent studies, Dinh and Marechal [13] studied a real-time FEM implementation on GPU, where a sorting-based implementation of parallel global assembly was performed. An implementation based on the principle of dividing the GPU assembly with standard sparse formats was presented by Sanfui and Sharma [14]. The authors used structured meshes with brick elements to demonstrate the advantage of workload division at the assembly stage. The implementation divided the assembly operation into a separate symbolic and a numeric kernel. Later, Zayer et al. [15] accelerated assembly of sparse matrices by modifying the assembly stage as a matrix-matrix multiplication with the aim to remove any CPU or GPU-based preprocessing. This approach enabled them to reduce storage and movement of data on the GPU. Among more recent works, Kiran et al. [16] presented a warp-based assembly approach for hexahedral elements in single precision where the numerical integration and assembly were performed in the same kernel. An implicit finite element model with cohesive zones and collision response was accelerated using CUDA by Gribanov et al. [17]. For handling the race condition in assembly, instead of coloring the elements, *atomicAdd* function was used to resolve it at the hardware level.

There has been several attempts to exploit properties of the stiffness matrix arising in the FEA to *further* accelerate it on the GPU. This essentially translates to two separate goals: Reducing the memory and/or reducing the total FLOP count of the application. For example, [18] implemented the SpMV stage of the FE solver on GPUs based on a FEA specific prefetching strategy to obtain a 3x speedup over the *traditional* SpMV implementation on the GPU. Another example is by [19], where the author exploited the typical sparsity pattern of global stiffness matrix depending on the number of degrees of freedom per node to reduce memory requirements and computation, thereby attaining a 18% to 51% performance improvements over the standard CuSPARSE library. The present work is in line of these work as we try to exploit the symmetry property of the local and global stiffness matrix to achieve those two goals. The local matrix generation stage, assembly stage and the SpMV stage are implemented on the GPU, each using two versions: One that exploits symmetry and one that does not. In the next two sections the methodology of implementations for the elemental calculations and matrix assembly are presented. This is followed by the results and discussion, after which the concluding remarks are presented.

Symmetry in FEA

Symmetry in Elemental Computation

In order to take advantage of the symmetry, a specialized kernel is written which is responsible for computing only the lower triangular part of the elemental stiffness matrix. In this kernel, each thread block is assigned to one element of the mesh and each thread of that thread block is assigned to one entry of the elemental stiffness matrix. A standard version of the kernel is also implemented where the entire elemental matrix is computed for comparison.

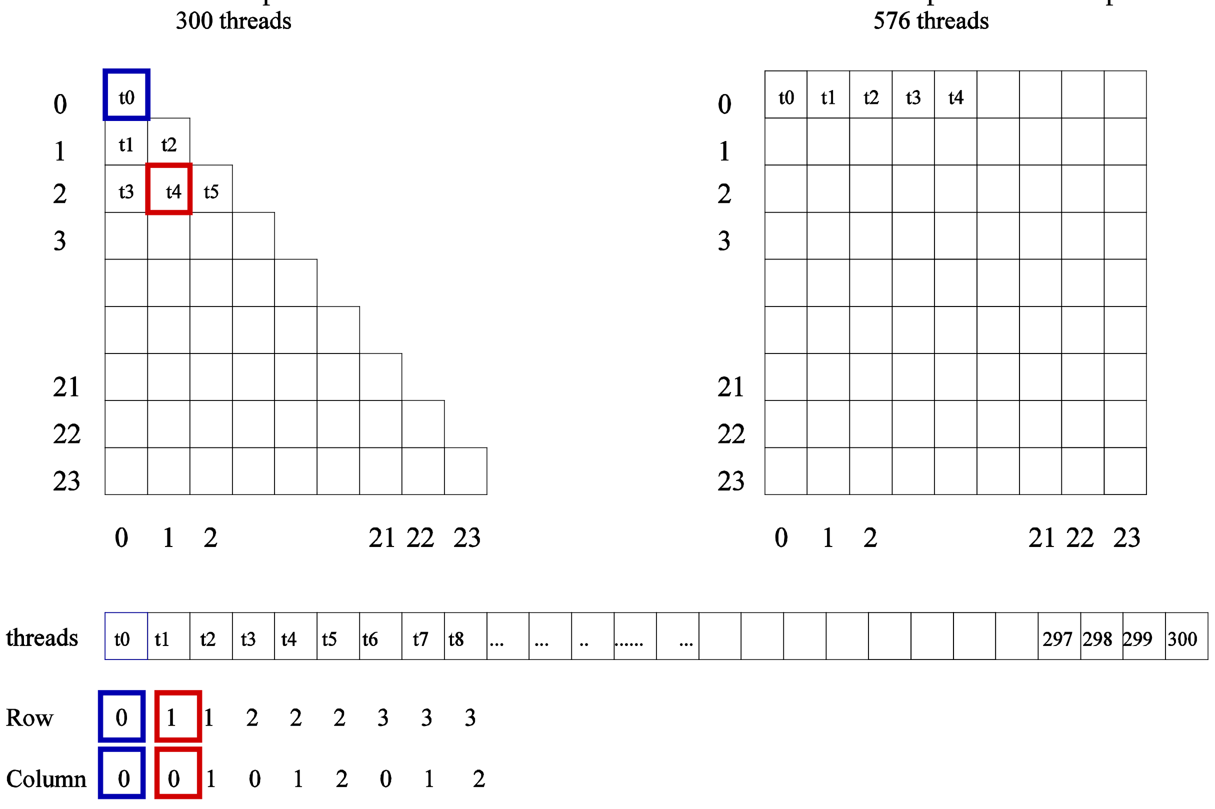


Figure 1. Distribution of threads in symmetric and non-symmetric kernel

As shown in figure 1, for a one entry per node GPU kernel, each block needs to have exactly 300 threads (shown in the left) as compared to 576 in the standard version (shown in the right). Since each warp has 32 threads, this corresponds to 10 and 18 warps respectively for the two versions. It should be noted here that the last warp in the symmetric version only has 12 threads. Although it is usual practice to take the number of threads as an exact multiple of the warp size, taking 320 threads per block and having 20 threads of the last warp as idle does not improve the results. So the block size is kept at 300.

Each of the threads in the thread block needs to know what row and column number it needs to calculate the elemental stiffness entry for. For example, in figure 1, the thread t4 needs to know that it is responsible for computing $K^e[2][1]$. Similarly, t10 needs to know that it is responsible for computing $K^e[4][0]$. It is important to note here that for a CPU-based sequential implementation of the same is not a major issue. A simple conditional statement with counters can take care of the issue. But, there is a different scenario for parallel versions on the GPU. This is because inside the kernel all the 300 threads need to know their target indices simultaneously. An approach could be where the same while loop is run for each thread, but with (i < threadIdx.x) as the termination condition. This will have the while loop running for 0, 1, 2, 3... iterations for t0, t1, t2, t3 and so on respectively. The first problem with this is that this will create an uneven load on threads. Second and most importantly, this will create massive branch divergence within each warp due to the conditional statements. To counter this we have implemented a numbering scheme that computes the values of the target index from the thread index alone. The row and column index for each thread is computed using equations (1) and (2) respectively.

$$m = (\sqrt{1 + 8 \times threadIdx.x} - 1)/2 \tag{1}$$

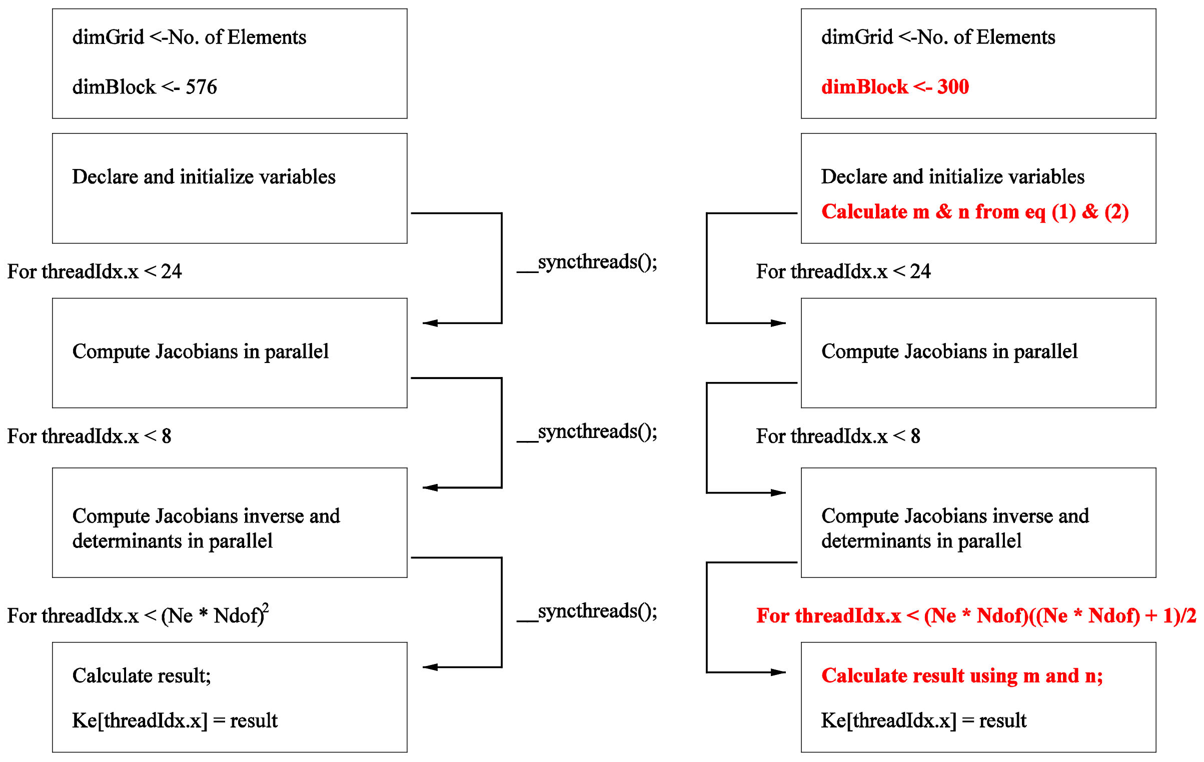
$$n = threadIdx.x + m - (m+1)(m+2)/2 + 1 \tag{2}$$

It is important to note that all the divisions and the square root is performed as integer calculations. This means that if a floating point value is obtained at some point, the fractional part is dropped. The values for m and n with corresponding threadIdx.x up to 39 is shown in table 1.

Table 1: Value of m and n with threadIdx.x

threadIdx.x	m	n	threadIdx.x	m	n	threadIdx.x	m	n	threadIdx.x	m	n
0	0	0	10	4	0	20	5	5	30	7	2
1	1	0	11	4	1	21	6	0	31	7	3
2	1	1	12	4	2	22	6	1	32	7	4
3	2	0	13	4	3	23	6	2	33	7	5
4	2	1	14	4	4	24	6	3	34	7	6
5	2	2	15	5	0	25	6	4	35	7	7
6	3	0	16	5	1	26	6	5	36	8	0
7	3	1	17	5	2	27	6	6	37	8	1
8	3	2	18	5	3	28	7	0	38	8	2
9	3	3	19	5	4	29	7	1	39	8	3

Figure 2 demonstrates the flow of work in the standard and the symmetric kernel. On the left side of the figure, the changes from the standard kernel are highlighted in red. As discussed the first change is in the kernel launch parameter. Where the standard version require 576 threads per thread block for eight-noded hexahedral mesh, the symmetric version require only 300. In the next change the values of m and n are computed from equations (1) and (2). These values are stored in per thread registers for faster access. Following this a *__syncthreads()* is called for barrier synchronization. Following this in both approaches 24 threads collaborate to compute the Jacobian matrices. After another synchronization is performed, 8 threads collaborate to compute the Jacobian inverse and determinants for both the approaches. After this another barrier synchronization is performed. Finally in case of the symmetric kernel, the value of the entry is calculated using the values of m and n and the values are stored in the elemental matrix in parallel.



dimGrid <-No. of Elements
dimBlock <- 576
Declare and initialize variables
For threadIdx.x < 24
Compute Jacobians in parallel
For threadIdx.x < 8
Compute Jacobians inverse and determinants in parallel
For threadIdx.x < (Ne * Ndof)2
Calculate result;
Ke[threadIdx.x] = result
__syncthreads();
__syncthreads();
__syncthreads();
dimGrid <-No. of Elements
dimBlock <- 300
Declare and initialize variables
Calculate m & n from eq (1) & (2)
For threadIdx.x < 24
Compute Jacobians in parallel
For threadIdx.x < 8
Compute Jacobians inverse and determinants in parallel
For threadIdx.x < (Ne * Ndof)((Ne * Ndof) + 1)/2
Calculate result using m and n;
Ke[threadIdx.x] = result

Figure 2. Flow of work in standard (left) and symmetric (right) kernel

Symmetry in the Assembly Phase

Unlike in the CPU, assembly of the local matrices on the GPU is considered to be a tedious and complex task on the GPU. This is why often *Assembly-free* methods are preferred on the GPU. Markall et al. [12] has commented in his work that assembly-free methods are more suitable for GPUs, whereas, assembly is more suitable for the CPUs.

We have implemented assembly of the local stiffness matrices on the GPU using both standard and symmetric versions. The key idea behind exploiting symmetry in the assembly stage is that to obtain the lower (or upper) triangular part of the global stiffness matrix, one only needs to assemble the lower (or upper) triangular part of all the local stiffness matrix. The idea is shown in figure 3, where the example of a simply supported beam is discretized using 4 elements in 2D. The four elemental stiffness matrices are shown with their lower triangular part highlighted. It is shown that during assembly, if the lower triangular part of the local matrix is assembled, the resulting global stiffness matrix is also lower triangular in nature, thus obviating the need to compute the upper half of the matrix.

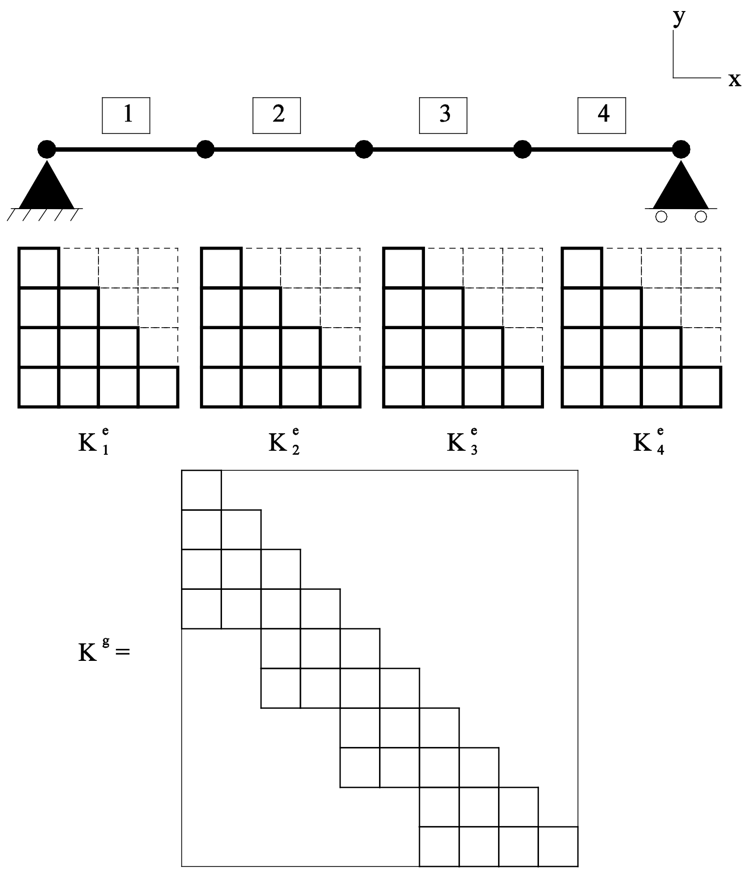


Figure 3. Assembly in the symmetric kernel for only the lower triangular part of the elemental stiffness matrix

GPU implementations of global stiffness assembly has the inevitable problem of race condition due to the parallel nature of the algorithm. We have implemented two strategies to counter the data race. The first one using *atomic operations*, has the disadvantage that it cannot use double precision. This creates a major issue in the linear solver stage because the method of conjugate gradients is considered to be notoriously sensitive to round-off errors. However, it should be mentioned here that on the latest GPUs from NVIDIA such as the Pascal P100 with compute capability 6.X, a version of atomic operations with double precision is introduced. In the second approach, a coloring scheme is applied. The key idea is to partition the elements into several different colors such that no two elements of the same color share a common node. Then the algorithm is run for the different colors in a sequential manner. The assembly implementation with coloring is run entirely in double precision.

Results and Discussion

For performance analysis of the proposed algorithms based on symmetry exploitation, a workstation with Intel Xeon ES1650 (6 core, 3.2 GHz) processor, 12 GB RAM, and NVIDIA K40c GPU is used. The GPU has 12 GB of global memory with 15 multiprocessors and 192 cores per multiprocessor. A cantilever beam with an end load meshed with eight-noded hexahedral elements (HEX8) are considered for the analysis.

The symmetry exploiting version of the code is tested and compared with the standard implementation that computes the entire matrix. Figure 4(a) shows the variation of the wall clock time versus the number of nodes for both symmetric and standard version. It can be seen that both application show the same trend of linearly increasing execution time. It can also be observed the symmetric version takes considerably lower time than the standard version especially at higher node numbers. Figure 4(b) shows the variation of the GFLOP/s count with the number of nodes for both version. Both the curves exhibit the trend of increasing GFLOP/s with increasing node numbers at lower mesh size and a more or less stable state for higher mesh sizes. However, it can be seen that for a mesh size of more than 1,000,000, there is an approximately 25% increase in the GFLOP/s count in the symmetric version over the standard implementation. Figure 5 shows the variation of speedup of the symmetric version over the standard implementation with increasing number of nodes. It can be seen that the speedup increases rapidly up to a mesh size of 1,500,000 nodes. After this point the speedup increases very slowly with increasing node numbers.

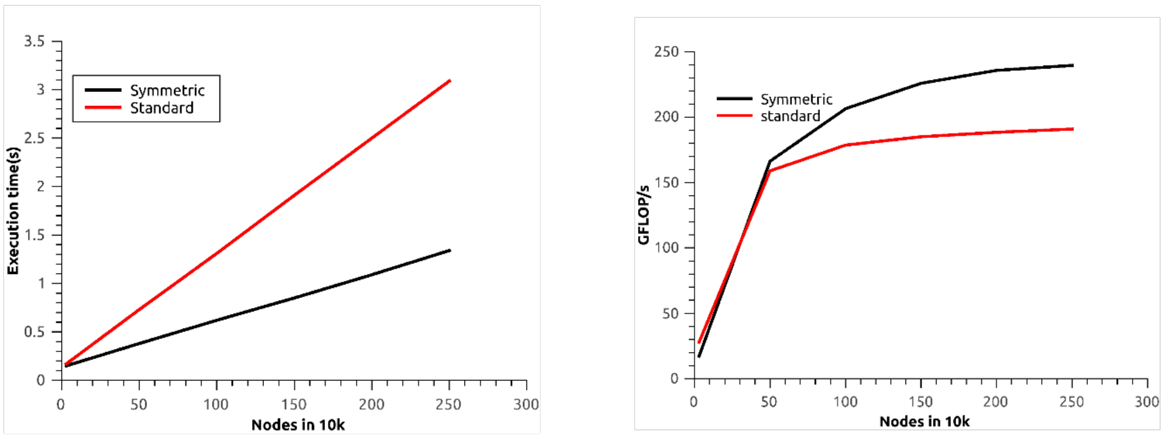


Figure 4. Wall clock time (a) and GFLOP/s count (b) for local matrix generation vs number of nodes in 10,000 in symmetric and standard versions.

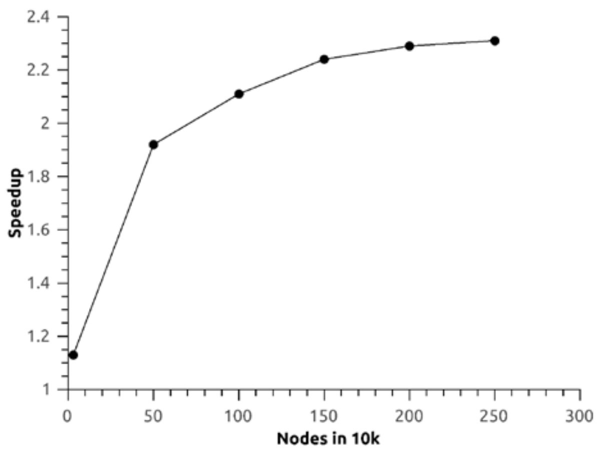
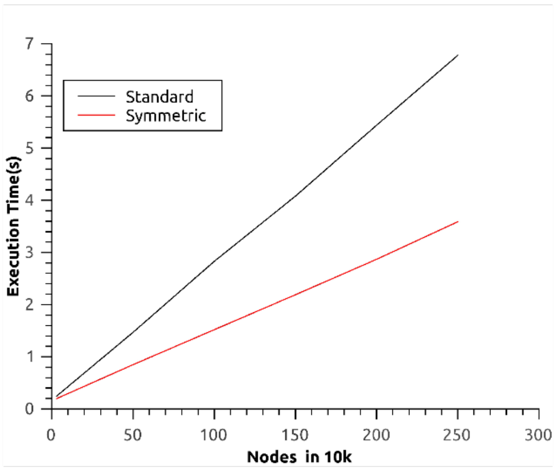


Figure 5. Comparison of speedup of symmetric version compared to atomics version with number of nodes

Two different versions, one symmetric and one standard, has been implemented for both coloring and atomics approach of assembly. It should be mentioned that due to lack of double precision support for atomics in the testing hardware only single precision has been implemented for atomics implementations. Figure 6(a) and figure 6(b) show the comparison of wall clock time with the number of nodes for Atomics and coloring respectively. Both symmetric and standard implementation is plotted in both of the figures. Although both figures 6(a) and figure 6(b) show similar time, it should be noted that figure (b) shows time using single precision arithmetic whereas, figure (a) uses double precision. Again a similar increasing trend is seen in the execution time for both the plots. The difference between the symmetric and standard implementation time increases as the mesh size is increased. The comparison of speedup in the entire proposed assembly operation using atomics and coloring over the standard implementation using coloring is plotted for different mesh sizes in figure 7. It can be observed that the speedup for the coloring approach is slightly higher than in case of the atomics approach. Also in both cases the speedup value increases with increasing mesh size and becomes stable at approximately 2x for coloring and 1.7x for atomics.



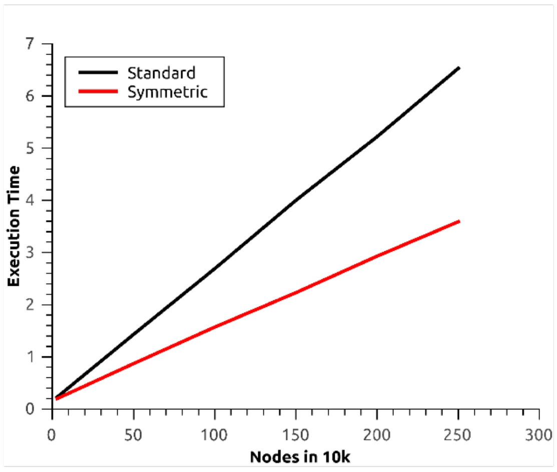


Figure 6. Wall clock time comparison for Coloring (A) and atomics (B) version with nodes in 10,000

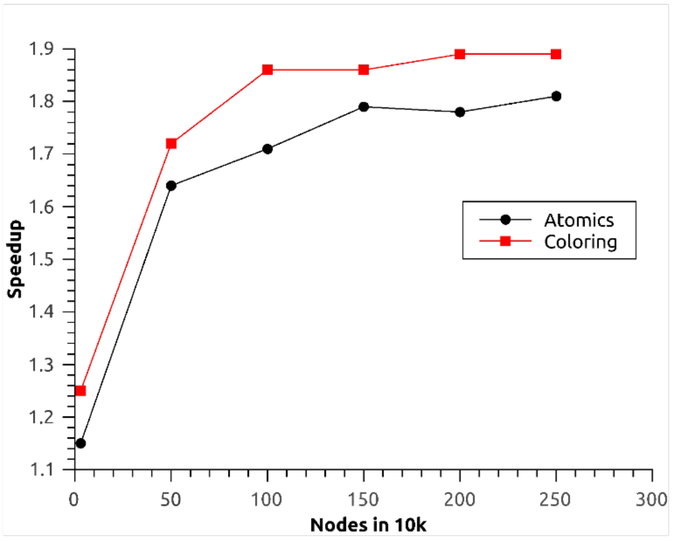


Figure 7. Comparison of speedup of symmetric version compared to atomics version with number of nodes

Concluding Remarks

An implementation for accelerating FEA on the GPU based on the exploitation of the symmetry in the local and global matrix is presented. The implementation covers the elemental computation and assembly stage of a typical FEA. By exploiting symmetry in the local matrix generation stage, the execution time can be reduced by an amount of more than two. Furthermore there is also the benefit of lower storage space required, as only the symmetric part of the matrix is recorded. The symmetric version in the local matrix generation stage performs significantly better than the standard implementation. However, after the mesh size of approx. 1,500,000 nodes, the speedup is seen to be varying by little. The GFLOP/s count is 25% higher in the symmetric version than in the standard version of local matrix generation for higher node numbers. For handling race condition, the coloring method is the only viable option since atomics is incompatible with double precision, which is very important for the CG method. In the assembly stage as well approximately two speedup is obtained for the symmetric implementation over the standard one using both atomics and coloring. However, the speedup is slightly higher in case of coloring.

References

[1] Zienkiewicz, O. C., Taylor, R. L., Nithiarasu, P., & Zhu, J. Z. (1977). *The finite element method* (Vol. 3). London: McGraw-hill.

[2] Cecka, C., Lew, A. J., & Darve, E. (2011). Assembly of finite element methods on graphics processors. *International journal for numerical methods in engineering, 85*(5), 640-669.

[3] Schmidt, S., & Schulz, V. (2011). A 2589 line topology optimization code written for the graphics card. *Computing and Visualization in Science, 14*(6), 249-256.

[4] Komatitsch, D., Michéa, D., & Erlebacher, G. (2009). Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing, 69*(5), 451-460.

[5] Dziekonski, A., Sypek, P., Lamecki, A., & Mrozowski, M. (2012). Finite element matrix generation on a GPU. *Progress In Electromagnetics Research, 128*, 249-265.

[6] Macioł, P., Płaszewski, P., & Banaś, K. (2010). 3D finite element numerical integration on GPUs. *Procedia Computer Science, 1*(1), 1093-1100.

[7] Dziekonski, A., Sypek, P., Lamecki, A., & Mrozowski, M. (2012). Accuracy, memory, and speed strategies in GPU-based finite-element matrix-generation. *IEEE Antennas and Wireless Propagation Letters, 11*, 1346-1349.

[8] Banaś, K., Płaszewski, P., & Macioł, P. (2014). Numerical integration on GPUs for higher order finite elements. *Computers & Mathematics with Applications*, 67(6), 1319-1344.

[9] Banaś, K., Krużel, F., & Bielański, J. (2016). Finite element numerical integration for first order approximations on multi-and many-core architectures. *Computer Methods in Applied Mechanics and Engineering, 305*, 827-848.

[10] Reguly, I. Z., & Giles, M. B. (2015). Finite element algorithms and data structures on graphical processing units. *International Journal of Parallel Programming, 43*(2), 203-239.

[11] Filipovic, J., Peterlik, I., & Fousek, J. (2009, July). GPU Acceleration of equations assembly in finite elements method-preliminary results. *In SAAHPC: Symposium on Application Accelerators in HPC.*

[12] Markall, G. R., Slemmer, A., Ham, D. A., Kelly, P. H. J., Cantwell, C. D., & Sherwin, S. J. (2013). Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids, 71*(1), 80-97.

[13] Dinh, Q., & Marechal, Y. (2015). Toward real-time finite-element simulation on GPU. *IEEE Transactions on Magnetics*, *52*(3), 1-4.

[14] Sanfui, S., & Sharma, D. (2017, February). A two-kernel based strategy for performing assembly in FEA on the graphics processing unit. In *2017 International Conference on Advances in Mechanical, Industrial, Automation and Management Systems (AMIAMS)* (pp. 1-9). IEEE.

[15] Zayer, R., Steinberger, M., & Seidel, H. P. (2017, September). Sparse matrix assembly on the GPU through multiplication patterns. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)* (pp. 1-8). IEEE.

[16] Kiran, U., Sharma, D., & Gautam, S. S. (2018). GPU-Warp based Finite Element Matrices Generation and Assembly using Coloring Method. *Journal of Computational Design and Engineering.*

[17] Gribanov, I., Taylor, R., & Sarracino, R. (2018). Parallel implementation of implicit finite element model with cohesive zones and collision response using CUDA. *International Journal for Numerical Methods in Engineering*, *115*(7), 771-790.

[18] Dehnavi, M. M., Fernández, D. M., & Giannacopoulos, D. (2010). Finite-element sparse matrix vector multiplication on graphic processing units. *IEEE Transactions on Magnetics, 46*(8), 2982-2985.

[19] Altinkaynak, A. (2017). An efficient sparse matrix-vector multiplication on CUDA-enabled graphic processing units for finite element method simulations. *International Journal for Numerical Methods in Engineering, 110*(1), 57-78.