INCOM18: Proceedings of the 1st International Conference on Mechanical Engineering
Jadavpur University Kolkata India January 4 – 6, 2018
Paper No. INCOM18-210

GPU Acceleration of Local Matrix Generation in FEA by Utilizing Sparsity Pattern

S. Sanfui[1] , D. Sharma[2]
[1,2]Department of Mechanical Engineering, IIT Guwahati, Guwahati, Assam

ABSTRACT

This paper presents a strategy for accelerating generation of local stiffness matrices in Finite Element Analysis (FEA) on Graphics Processing Units (GPU) for large unstructured meshes. The proposed strategy is based on *the key idea of removing redundant computations from the matrix multiplication operation and thereby* reducing the total number of floating point operation (FLOP) count, which in turn reduces the total execution time. This is achieved by using a small array on the GPU shared memory that stores the sparsity pattern of the matrix containing the derivatives of the shape functions. This same array is used in conjunction with other input data to generate the local matrices of mesh sizes of up to 2 *million* elements in three dimensions. Comparison of the proposed strategy with a standard implementation shows improvement both in terms of execution time and performance.

Keywords: *FEA, GPU, Local matrix generation*

1. INTRODUCTION

The Finite Element Method (FEM) is a numerical method for approximating solutions of boundary value problems for partial differential equations(PDEs). It has been used in fields such as mechanical engineering, civil engineering, electrical engineering, medical applications etc., where a solution to PDEs is sought. 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(Zienkiewicz and Taylor[1]).

Although finite element analysis is widely popular in several fields, it suffers from a high computational complexity, because of the generation and resolution of system of equations performed in the solver step. Several applications that make use of FEM, have reported it to be the most or computationally expensive part of the whole applications especially for larger and more complex geometries (Georgescu et al. [2]). One of the remedies to this is using some form of parallel implementation to exploit the data parallel and throughput intensive nature of the FEA computations on platforms such as GPUs. The primary challenge for efficiently accelerating a scientific application is that all the existing scientific algorithms have been developed over the years to execute efficiently on sequential hardwares such as traditional CPUs. Markall et al. [3] has demonstrated that in order to efficiently port FEA on the GPU, an implementation, radically different than on the CPU, is required. Among different stages of FEA, the solver stage is generally considered to be the most time-consuming(Georgescu et al.[2]). But it has been shown that the local matrix generation stage can take up to 80% of the total time. In the present work, we target GPU implementation of the local matrix generation stage of FEA for large mesh sizes.

1.1 GPU architecture and CUDA

Graphics Processing Units are specialized circuits which, although primarily designed for rendering graphics, have been largely implemented for accelerating computation arising in various research, scientific and analytical applications. A number of platforms such as OpenCL, OpenMP and CUDA are derived over the past years for parallelizing general purpose applications. For the present implementation, CUDA, a parallel computing platform and API developed by NVIDIA, has been used with NVIDIA GPUs. CUDA provides the architecture and programming model that accommodates both the host (CPU) and device (GPU) simultaneously. The device code is written using particular extensions to the C programming syntax, inside special functions termed as Kernels. These Kernels may generate grids of thousands or even millions of threads to parallelize given task instead of running in a sequential manner.

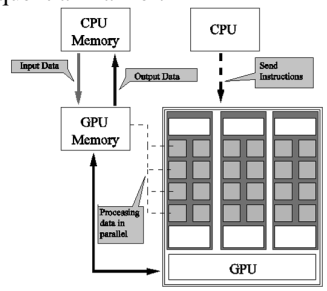

Fig. 1: CPU-GPU hybrid model

A schematic representation of the data flow in a CPU-GPU hybrid computing environment is presented in Fig. 1.

2. Local stiffness generation in FEA

This stage computes the local stiffness matrix based on the nodal data by performing numerical integration. It was demonstrated by Dziekonski et al. [4], 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. So inevitably, any improvement in this stage, either in terms of memory or in terms of execution time, can significantly reduce the total execution time of a standard FE application. The first work dedicated towards numerical integration for FEA on the GPU is by Maciol et al. [5]using the Gauss-Legendre Quadrature Method. Authors demonstrated the complete scalability of the numerical integration process on the GPU. 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. [4], 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. Banas et al. [6] addressed the problem of an implementation of numerical integration on the GPU that is portable across several GPU architectures and different orders of approximation. Apart from these, many works concentrating on applications of FEA have also implemented local matrix generation on the GPU (Reguly and Giles [7], Komatitsch et al. [8], Schmidt and Schulz [9]). In these cases, however, no implementational details are provided. In the present analysis we have accelerated the numerical integration using 8 noded hexahedron elements. The method used is the standard Gauss Quadrature method. To illustrate the method briefly, the eight shape functions for a Cubic Hexahedron element are given by

$$\boldsymbol{N_i} = {}^{1}/_{8}\,(1+\xi\boldsymbol{\xi_i})(1+\eta\boldsymbol{\eta_i})(1+\zeta\boldsymbol{\zeta_i}), \qquad (1)$$

where, ξ, η and ζ are the reference coordinates and ξ_i, η_i and ζ_i denote coordinates of the ith node. The elemental stiffness matrix is given by numerical integration as,

$$K = \iiint B^T C B \, dxdydz \qquad (2)$$
$$= \sum_{i,j,k} w_{\xi_i} w_{\eta_j} w_{\zeta_k} B^T C B |J(\xi_i, \eta_i, \zeta_i)|$$

Here, B is the matrix that contains partial derivatives of the shape functions with respect to x, y and z. J and C are the Jacobian and constitutive matrix respectively. w_{ξ_i}, w_{η_j} and w_{ζ_k} are the weights for carrying out numerical integration using Gauss Quadrature method. There exist different ways to distribute the workload among the parallel processors. Workload distribution in this context means the assignment and distribution of computation among different threads and thread-blocks. In the present implementation, each block is assigned to one finite element and each thread is responsible for computing one entry of the local stiffness matrix.

3. Proposed Acceleration Strategy

There are several ways a GPU application can be accelerated in order to achieve reduced runtime. Two of the direct ways to tackle this is by reducing the FLOP count or by reducing memory accesses to the DRAM. This, at the algorithm level, translates to restructuring the algorithm by reducing redundant accesses and redundant computation. However it should be also noted that just by merely reducing the FLOP count and memory accesses does not guarantee a reduced run-time. If, for example the reduced total memory accesses come at the price of an irregular memory access pattern, or if the reduced total FLOP count come at the price of increased branch divergence, the runtime may actually be increased instead of decreasing. In the present implementation of local stiffness matrix generation on GPU, we have applied a method with reduced memory accesses and FLOP count coupled with a parallelization strategy involving effective use of shared memory and registers. The key idea is to identify and remove the redundant computation and DRAM accesses, while at the same time maintaining data-coalescence and avoiding branch-divergence. As per the usual practice, the derivatives of the shape functions have been pre-computed on the CPU and sent to the GPU. Each thread block is assigned to one single element of the mesh. The element connectivity, constitutive matrix and coordinate information for a particular element are loaded into the shared memory of the corresponding thread-block from the DRAM at the beginning of the kernel. Furthermore, memory is allocated to each block for storing the jacobian matrices and their determinant values. Lastly a special map array called *shapeMap* is loaded into the shared memory that has the dimension $n_{DOF} \times n_{DOF}$. This array stores the *sparsity and repetition pattern* of the *B* matrix, that contain the derivatives of the shape functions. Thanks to this map array, the *B* matrix need not be constituted explicitly. Also it will be shown later how this array is used to reduce the total FLOP count compared to a standard implementation. Figure 2 explains how the *shapeMap* array is constructed. It can be seen from the figure that the location of the non-zero entries in the *B* matrix has a repeating pattern that repeats for each shape function marked in the image from N_1 to N_8. The *shapeMap* array

stores the location of the derivative of the particular shape function with respect to the three directions *x, y* and *z* for each column of the repeating pattern. For example, in the first column of *B*, the derivative of N_1 with respect to *x* is at the first row. Hence, *shapeMap[0][0] = 0*. The derivative of N_1 with respect to *y* is at row 5. Hence, *shapeMap[0][1] = 5*. The derivative of N_1 with respect to *z* is at row 4. Hence, *shapeMap[0][2] = 4* and so on.

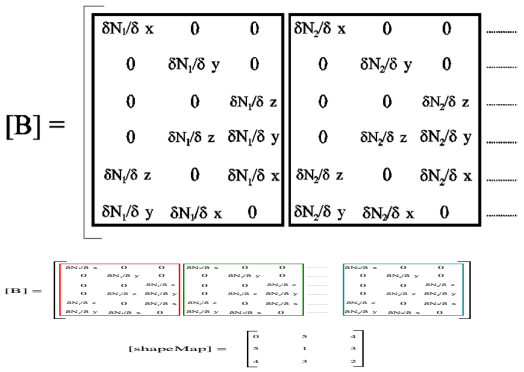

Fig. 2: Construction of *shapeMap* from *[B]*

All in all, for this stage the total amount of shared memory used in the kernel for each thread block in double precision is 1.16 kilobytes. The GPU used is an NVIDIA Tesla K40c, with 48 kilobytes of shared memory per streaming multiprocessor(SM). This allows for the kernel to have $(48 / 1.16) \sim 41$ thread blocks to run concurrently on each SM, which is higher than the maximum permissible thread blocks/SM (16 for compute capability 3.5). However, profiling the application reveals registers to be the limiter for the application. Hence further increasing the used shared memory would have no consequences on the final runtime.

As mentioned before, the implementation is based on the key idea of recognition and removal of redundant computation and data access coupled with an efficient parallelization strategy that conforms to the standard CUDA optimization practices. For computing the local stiffness matrix, three matrices need to be multiplied (B^T, *C* and *B* respectively). This multiplication summed over all the gauss points gives the 24×24 local stiffness matrix for a hexahedron element with 3 DOF per node. We adopted a strategy where each of the entry in the local stiffness matrix is allotted to one thread of the thread block. As can be seen from Fig. 3, for such a scheme, each of the thread has to compute exactly seven dot products. In Fig. 3. The darker boxes correspond to the entry *K[0][0]* computed by thread 0 and the lighter boxes correspond to *K[10][10],* handled by thread 76. Take the example of the dark box in the elemental stiffness matrix at *K[0][0]* shown in Fig. 3. To Compute this entry , the first row of B^T need to be multiplied with each column of *[C]*. And finally the resulting vector needs to be multiplied to the first column of *[B]*, resulting in a total of 7 inner products. Each of these dot products are of size six, irrespective of the number of nodes in the element for three dimensional elastic bodies. An inner product of size N always requires N multiplications and N - 1 additions, totaling to 2N - 1 FLOPs. This 11 FLOPs/inner product for the dimension of 6. So, in a traditional implementation, $7 \times 11 = 77$ FLOPs are required for each thread. Using the shapeMap array, the number of multiplications and additions for each inner product has been reduced to three and two respectively. The number of inner products have also been reduced from 7 to 4. In Fig. 3, for computing the dark entry of the elemental stiffness matrix, instead of multiplying the whole row of $[B]^T$ to *[C]*, only the non-zeros, marked in dark boxes are multiplied. Again instead of multiplying the vector with each column of the *[C]* matrix, only the columns at indices which are non-zero in the first column of are multiplied. Thus each thread will now perform $4 \times 5 = 20$ FLOPs. Furthermore, not having to store the *B* matrix gives the advantage of lower shared memory and registers usage by the thread block and threads respectively. So the total number of FLOPs saved for computing one element of the mesh is $(77 - 20) \times 576 = 32832$ FLOPs. For a high number of elements this saves a significant amount of the computation.

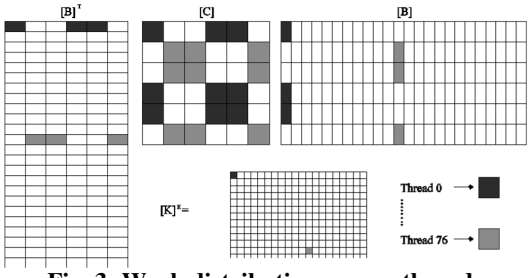

Fig. 3: Work distribution among threads

4. Results

The implementations were tested on a workstation with the configuration given in Table 1**.** For comparison, a standard implementation is used, where the exact same computations are carried out using the same methodology, but the *shapeMap* array is not used to remove redundant calculations. Figure 4 shows the variation of wall clock time of the application with the number of nodes in the mesh for the proposed implementation and the standard implementation.

Table 1: Test setup configuration

CPU	Xeon ES1650 (6 core, 3.2 GHz)
RAM	16 GB
GPU	Nvidia Tesla K40c
GPU DRAM	12 GB

GPU cores	2880
GPU Bandwidth	288 GB/s
Peak FLOP/s	1.43TFOP/s (FP64)

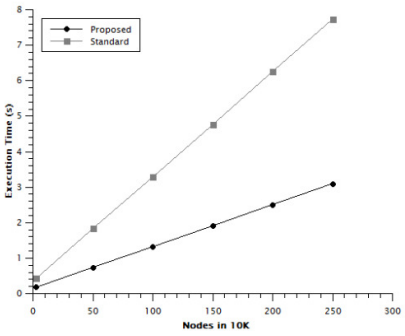

Fig. 4: Variation of execution time with nodes

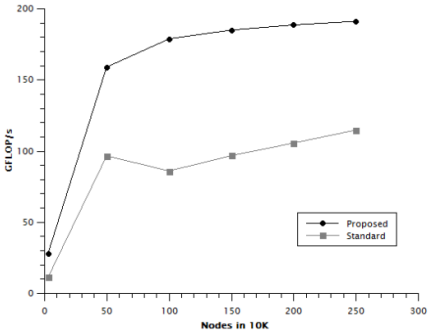

Fig. 5: Variation of GFLOP/s with nodes

There can be seen a linear increase of the execution time with increase in number of nodes for both the implementations. This implies that the implementation is scalable with the problem size. However, after a certain limit, the slope of the execution time is expected to increase. The execution time of the standard implementation is approximately 2.5 times the execution time of the proposed implementation. In Fig. 5, the GFLOP/s count of the application is plotted for *double precision*. The GFLOP/s count is considered to be an important performance metric for parallel implementations. It can be observed that the count keeps increasing up to a node number of approximately 1,000,000 at a rapid rate for both the proposed and standard implementations. After that mark, the count is increased slightly with increase in node numbers. After 2,000,000 there is almost no change in the GFLOP/s count. This is again expected because, for lower mesh sizes, the GPU is not completely occupied. The GFLOP/s count of the proposed method is seen to be approximately 60% – 70% higher for mesh sizes above 1,000,000. After the GPU is completely occupied and the rate of increase of GFLOP/s decreases and a somewhat stable value is reached around 190 GFLOP/s for the proposed implementation.

5. Conclusion and Future Work

A strategy for computing elemental stiffness matrices for large meshes on the graphics processing units is presented. Results are obtained for mesh sizes of over 2 million. Using the shapeMap array, the number of FLOP count is decreased significantly. The execution time and GFLOP/s comparison with increasing number of node reveals that the savings in FLOP count is more pronounced for higher mesh sizes. The execution time is seen to increase linearly with the number of nodes. Comparison with a standard implementation reveals both decrease of execution time and performance improvements in terms of GFLOP/s. For future work, this implementation may be extended to different element types and different workload distribution strategies.

6. REFERENCES

[1] Zienkiewicz, O. C. and Taylor, R. L., 1977, The finite element method, Volume 3. McGraw hill London.

[2] Georgescu, S., Chow, P., and Okuda, H., 2013, Archives of Computational Methods in Engineering 20(2), 111-121.

[3] Markall, G., Slemmer, A., Ham, D., Kelly, P., Cantwell, C., and Sherwin, S., 2013, International Journal for Numerical Methods in Fluids 71(1), 80-97.

[4] Dziekonski, A., Sypek, P., Lamecki, A., and Mrozowski, M., 2012, Progress In Electromagnetics Research 128, 249-265.

[5] Maciol, P., Plaszewski, P., and Bana´s, K., 2010, Procedia Computer Science 1 (1), ICCS 2010, Amsterdam, Netherlands, 1093 - 1100.

[6] Banas, K., Paszewski, P., and Maciol, P., 2014, Computers & Mathematics with Applications 67(6), 1319 - 1344.

[7] Reguly, I. Z. and Giles, M.B., 2015, International Journal of Parallel Programming 43(2), 203-239.

[8] Komatitsch, D., Micha, D., and Erlebacher, G., 2009, Journal of Parallel and Distributed Computing 69(5), 451 - 460.

[9] Schmidt, S. and Schulz, V., 2011, Computing and Visualization in Science 14(6), 249-256.