GPU-Warp based Finite Element Matrices Generation and Assembly using Coloring Method

Utpal Kiran, Deepak Sharma* and Sachin Singh Gautam

Department of Mechanical Engineering, Indian Institute of Technology Guwahati, Assam-781039, India

Abstract

Finite element method has been successfully implemented on the graphics processing units to achieve a significant reduction in simulation time. In this paper, new strategies for the finite element matrix generation including numerical integration and assembly are proposed by using a warp per element for a given mesh. These strategies are developed using the well-known coloring method. The proposed strategies use a specialized algorithm to realize fine-grain parallelism and efficient use of on-chip memory resources. The warp shuffle feature of Compute Unified Device Architecture (CUDA) is used to accelerate numerical integration. The evaluation of elemental stiffness matrix is further optimized by adopting a partial parallel implementation of numerical integration. Performance evaluations of the proposed strategies are done for three-dimensional elasticity problem using the 8-noded hexahedral elements with three degrees of freedom per node. We obtain a speedup of up to 8.2× over the coloring based assembly by element strategy (using a single thread per element) on NVIDIA Tesla K40 GPU. Also, the proposed strategies achieve better arithmetic throughput and bandwidth.

Key words: Finite Element Method, Numerical Integration, Assembly, GPU, CUDA, Coloring Method

☆GPU-Warp based FE Matrices Generation and Assembly

*Corresponding author

Email addresses: ukiran@iitg.ernet.in, dsharma@iitg.ernet.in, ssg@iitg.ernet.in (Utpal Kiran, Deepak Sharma* and Sachin Singh Gautam)

1. Introduction

Computing on the general purpose graphics processing unit (GPGPU) has proved to be very efficient in accelerating compute-intensive as well as memory-intensive scientific codes. A GPU works together with a CPU (central processing unit) to accelerate scientific, analytic, engineering and enterprise applications around the world. There is a fundamental difference in the architectural design of a GPU and a CPU. GPUs are basically a many-core processors having thousands of cores designed to deliver high computational throughput, whereas CPUs sacrifice the computational throughput of the processor to increase the performance of a single core. This different design philosophy necessitates the development of specialized algorithms to exploit the potential performance of the GPU hardware [1]. GPUs are most suitable for throughput oriented numerical applications which operate over a huge amount of independent data set.

Finite Element Method (FEM) is one of the most popular numerical methods for obtaining the solution of partial differential equations. FEM simulation is highly compute and memory-intensive for complex problems having thousands or millions of mesh elements [2, 3]. The computational load further increases for problems in 3D (three dimensions). FEM primarily involves computation of elemental matrices and force vectors for all elements of the mesh and their assembly to a linear system of equations. For the 3D problem having a complex weak form or higher order elements, numerical integration and assembly time can be considerable. The performance can degrade more in problems where the assembly is required iteratively such as nonlinear elasticity. GPUs have been found to be very effective in accelerating almost every step of FEM based simulation [4]. The speedup of several orders of magnitude is reported in the assembly of elemental matrices [5, 6, 7] for different applications. Although the numerical integration is not so easily parallelizable, the notable speedup is observed in [8, 9, 10].

This paper aims to explore a more efficient strategy for GPU implementation of numerical integration and assembly for elasticity problems in 3D. The computation of the elemental matrix on GPU is problematic as it consists of a number of sequential steps. Our proposed assembly strategies make careful use of a specialized algorithm, suitable data structures, and recently added features in a programming language to obtain better performance. Three different strategies for assembly based on the coloring method for an 8-noded hexahedral element are presented. All strategies assign a warp to each element of the mesh for numerical integration and assembly either completely or partially. The warp shuffle feature of CUDA has been used in all the strategies to calculate Jacobian in a very efficient manner. The assembly to the global matrix is done in the CSR (Compressed sparse row) format by precomputing the indices. The profiling of the proposed strategies is done and performance is compared with established assembly by element strategy (single thread per element) [5, 7, 11].

This paper is organized as follows. Section 2 deals with details of previous works that have addressed the problem of implementing FEM on GPU. In section 3, the key components of the CUDA programming model and memory hierarchy available to each thread are presented. Finite element formulation of 3D elasticity problem is discussed along with the sequential and parallel implementation of FEM assembly. The parallel implementation is based on the coloring method and uses assembly by element strategy (single thread per element) for computation. The new strategies are proposed in section 4. In section 5, detailed analysis of the proposed strategies is presented and performance is compared to each other as well as with assembly by element strategy. The paper is concluded in section 6.

2. Previous works

Earlier studies on implementing FEM simulation on GPU have focused primarily on a solution for a sparse linear system of equations. It is because the solver is the most time-consuming step of FEM and that can be done in parallel.

There is a rich amount of literature discussing many efficient implementations of sparse linear solvers on GPU [12, 11, 7, 13]. Several efficient libraries, such as CUSP [14], MAGMA [15], AmgX [16], etc. also exist. New capabilities in hardware and more flexible programming environment have helped researchers to focus on other steps of FEM like the calculation of elemental matrices and assembly. These steps of FEM can consume a considerable amount of time for complex problems involving finer mesh [17], higher order finite elements [6], non-linear elasticity [18], and topology optimization [19] to name a few.

In [8], GPU implementation of numerical integration for electromagnetic application is discussed. Here, the computation for a single finite element is assigned to a single thread-block. Individual threads of the thread-block calculate a set of entries of the elemental stiffness matrix using one outer loop over integration points and two inner loops over shape functions. A grid of thread-blocks is launched for the whole mesh or a sub-mesh if the problem is large. The results show a speedup of 3× to 19× for a higher order prismatic element on NVIDIA GeForce 8800GTX. Another study by the same authors investigates numerical integration for quadrilateral elements with curved geometry using OpenCL in [20]. Numerical integration for the higher order tetrahedral element is implemented in [9] in which elemental matrix is computed for 32 elements in parallel. Each element uses 81 thread-blocks to perform the calculation for 81 Gauss points. Thereafter, the assembly of the elemental stiffness matrices is performed using a different kernel. There are some implementations in which integration and assembly are performed by single GPU kernel [10]. The well-known coloring method is used which assigns colors to the mesh such that no two elements of the same color share any common node. The GPU kernel then performs elemental stiffness calculations for the elements of the same color and then assemble them into the global stiffness matrix. The study shows coalesced access of node coordinates, but the detailed implementation of numerical integration is not discussed. The study demonstrates speedup of 7× and 10× for the quadrilateral and hexahedral (8-noded) elements respectively on NVIDIA GeForce GT430. Similar kind of assembly strategy using the coloring method

is found in [21]. However, authors suggest two types of implementations for elemental matrix computation for the 8-noded 3D elements. One using a single kernel for all the computations, while the second using three kernels. The latter implementation is found to be better that reduced the assembly time from 2.44 seconds to 0.65 seconds for 512,000 elements. However, the paper does not discuss the work distribution among the threads as well as the memory types used for storing intermediate values in numerical integration computation. In [22], numerical integration strategies which can scale to higher order elements are investigated. A group of threads called as work-group is used to do the computation for a block of the element stiffness matrix by looping over integration points. The GPU strategies are found to be performing better than CPU in all the test cases.

In [6], GPU-based simulation of seismic wave propagation is done using a high-order spectral-element. A thread block is assigned to each element consisting of 125 nodes to calculate the elemental contribution. The problem of race condition in global assembly is handled by using the coloring method. Another detailed analysis of global assembly for more general FEM implementation is presented in [5]. The authors investigate many approaches using different GPU memory types for storing elemental data and different kernel design for assembly. General conclusion recommends using two different approaches for low and high order elements. For lower order elements, the elemental data is stored in the shared memory of the GPU, while the assembly is done by associating the threads with non-zero (NZ) entries of the global matrix. For higher order elements, the best performing method uses a single thread per element to calculate the elemental data and stores it into the global memory of GPU. The assembly to the system of equations is done by parallel reduction. The elemental subroutine is treated as a black box which might hamper the performance when using the higher-order elements. While the most of the literature focuses on very specific implementation, in [11] a FEM implementation scalable to higher order finite elements is proposed. The elemental matrix calculation is done by allocating a single thread per element. A numerical integration scheme which

uses outer loops over shape functions and an inner loop over Gauss points is used. The numerical integration does not use any memory to store intermediate data and calculates them every time by doing the redundant computation. The other approaches which use local memory or global memory are found to be memory bandwidth bound. It also presents an insight into the effect of the data structure on the performance. In addition to the global matrix assembly approach, local matrix approach (LMA) and matrix-free approach are also discussed in the literature. Markall et al. [23] have compared several different assembly strategies on different many-core architectures. Authors proposed LMA approach for the assembly and demonstrated the suitability of this approach to many-core architecture for 2D meshes. In [7], a different approach to mesh partitioning is found. Disjoint sets of nodes called as patches are made out of the mesh. Element assignment is done in such a way that each element belongs to one patch only. The element stiffness matrix (4×4) is calculated by reading input data in the coalesced way and using a single thread per element. The CSR storage format for the global matrix is used which has precomputed column indices and row offsets. The assembly is done for all the elements belonging to a patch in the shared memory and final data is written in a coalesced manner to the global matrix in global memory. A new finite element assembly strategy based on sparse matrix multiplication is found in [24]. The strategy captures connectivity information of the mesh through sparse matrix representation and uses them to efficiently assemble elemental matrices by avoiding any type of preprocessing. Another recent finite element assembly method is found in [25]. In this work, the assembly process for 3D finite elements is dividing into a node-by-node symbolic part and an element-by-element numeric part.

The strategies presented in the above-mentioned studies achieve a significant speedup on GPUs which, however, are outdated. But the methods are still viable. Based on the survey of previous work, we found that numerical integration which does redundant computation and uses no off-chip memory is more suitable for GPU implementation. The other approaches which use local memory or global memory perform better for lower order elements becomes

memory bandwidth bound for elements having larger stiffness matrix size. This is particularly true in the case when a single thread per element strategy is used. In this work, we present a strategy which uses multiple threads to do the computation for one element. This provides us sufficient on-chip memory space for numerical integration (without redundant computation) as well as an elemental matrix. The existing coloring method is used to avoid the race condition during assembly. The mesh is divided into disjoint sets of elements denoted by different colors in the same way as in earlier works. Computation corresponding to each of the colors is done sequentially by the GPU kernel presented. The proposed strategies implement element matrix computation and assembly using the same kernel.

3. Preliminaries

3.1. GPU Architecture and CUDA

GPUs are throughput oriented devices designed to process the data parallel and data throughput tasks. The data parallel applications are those which operate over the independent data sets, whereas data throughput tasks need to process a huge amount of data. The hardware design of GPU is based on many-core processing units and differs considerably from multi-core processing units like CPU. The many-core design consists of a large number of simple processing units, very small cache, and a high memory bandwidth. On the other hand, the multi-core processor consists of few highly complex and efficient cores along with a large cache memory and low memory bandwidth. In this paper, GPU developed by NVIDIA is used, which consists of a number of streaming multiprocessor (SMs). Each SM contains a number of streaming-processor (SPs) and on-chip memory. It also has an off-chip memory (DRAM) known as global memory, which has the highest latency on GPU. The off-chip memory is also used for data transfer between CPU and GPU. The on-chip memory available to each SM is further divided into registers, configurable shared memory, and read-only data cache. Registers are allotted to one thread and cannot be accessed by

any other thread. The shared memory is common to a block of threads where it can be accessed by all threads of the block. The on-chip memory is much faster than the off-chip memory. Local memory is the private memory space of each thread. This memory often occupies space in registers, but can spill over to off-chip memory. Constant memory is read-only for GPU threads. Figure 1 shows NVIDIA Tesla K40 streaming multiprocessor along with memory hierarchy.

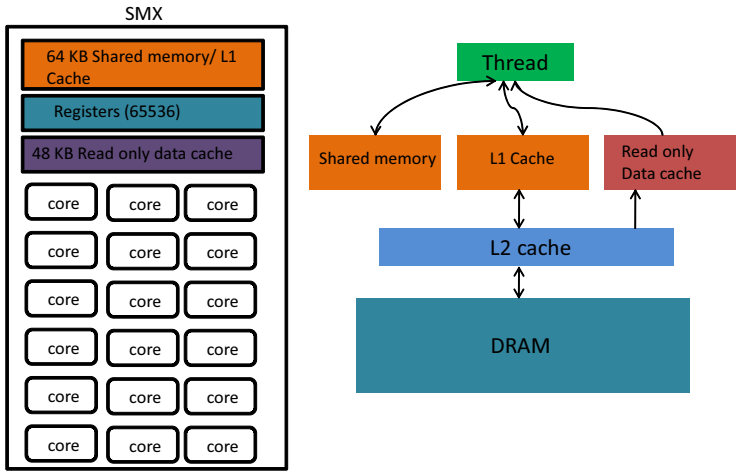

Figure 1: A SMX of NVIDIA Tesla K40 based on Kepler architecture. It has 15 SMX and 192 CUDA cores. The figure also shows memory hierarchy for a thread.

Compute Unified Device Architecture (CUDA) [26] is a parallel programming platform created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose computation. CUDA provides flexibility of using many languages like C, C++, FORTRAN, etc. to program the hardware (GPU). Interested readers can refer to [27] for more details.

3.2. Linear Elastic Finite Element Formulation

Finite element method is a numerical technique for solving partial differential equations (PDEs). The governing PDEs, that is, the strong form for linear elastic material is given by

$$\boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \boldsymbol{b} = \mathbf{0}, \tag{1}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress tensor, and $\boldsymbol{b}$ is the body force per unit volume. The strong form is supplemented by the displacement $\boldsymbol{u}$ and traction $\boldsymbol{t}$ boundary conditions given by

$$\begin{aligned} \boldsymbol{u} &= \boldsymbol{u}_o \quad on\ \Gamma_u, \\ \boldsymbol{t} &= \bar{\boldsymbol{t}} \quad on\ \Gamma_t, \end{aligned} \tag{2}$$

where $\boldsymbol{u}_o$ and $\bar{\boldsymbol{t}}$ are the specified displacement and traction on the displacement boundary Γ_u and the traction boundary Γ_t, respectively. The constitutive equation for the linear elastic material is given by Hooke's law as

$$\boldsymbol{\sigma} = \mathbf{D}\boldsymbol{\epsilon}. \tag{3}$$

Here, $\mathbf{D}$ is the material constitutive tensor and $\boldsymbol{\epsilon}$ is the small strain tensor given by

$$\boldsymbol{\epsilon} = \frac{1}{2}\left[\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T\right]. \tag{4}$$

The solution for the strong form given in (1) is difficult to obtain for any arbitrary geometry and boundary conditions. Hence, the strong form is converted into the weak form using the Galerkin weighted residual approach [28]. The domain on which the solution is sought is discretized into a number of polygons or polyhedra, called as the elements. The displacement at a point $\boldsymbol{x}$ over a typical element domain Ω^e is approximated as

$$\boldsymbol{u}(\boldsymbol{x}) = \sum_{i=1}^{n} \mathbf{u}_i \phi_i(\boldsymbol{x}), \tag{5}$$

where n is the total number of nodes per element, $\mathbf{u}_i$ is the nodal displacement vector of node i, and $\phi_i(\boldsymbol{x})$ is the shape function of node i. This can be rearranged in the following matrix form given by

$$\boldsymbol{u} = \left\{ \begin{array}{c} u_x \\ u_y \\ u_z \end{array} \right\} = \boldsymbol{\phi}\boldsymbol{\Delta}^e, \tag{6}$$

where, $\boldsymbol{\phi}$ is the shape function matrix given by

$$\boldsymbol{\phi} = \begin{bmatrix} \phi_1 & 0 & 0 & \phi_2 & 0 & 0 & ... & \phi_n & 0 & 0 \\ 0 & \phi_1 & 0 & 0 & \phi_2 & 0 & 0 & ... & \phi_n & 0 \\ 0 & 0 & \phi_1 & 0 & 0 & \phi_2 & 0 & 0 & ... & \phi_n \end{bmatrix}, \tag{7}$$

and $\boldsymbol{\Delta}^e$ is the elemental displacement vector given by

$$\boldsymbol{\Delta}^e = \begin{Bmatrix} u_x^1 & u_y^1 & u_z^1 & u_x^2 & u_y^2 & u_z^2 & ... & u_x^n & u_y^n & u_z^n \end{Bmatrix}^T. \tag{8}$$

Here, u_x^i, u_y^i, u_z^i are the x, y and z components of the displacement of the i^{th} node, respectively. The substitution of (6) into the weak form and assembly over all the elements gives the global form given by

$$\mathbf{K}\boldsymbol{\Delta} = \mathbf{F} + \mathbf{Q}, \tag{9}$$

where $\mathbf{K} = \sum_e \mathbf{K}^e$, $\mathbf{F} = \sum_e \mathbf{F}^e$, and $\mathbf{Q} = \sum_e \mathbf{Q}^e$. The expressions for elemental matrices are given by

$$\begin{aligned} \mathbf{K}^e &= \int_{\Omega_e} \mathbf{B}^T \mathbf{D} \mathbf{B} d\mathbf{x}, \\ \mathbf{F}^e &= \int_{\Omega_e} \boldsymbol{\phi}^T \boldsymbol{b} d\mathbf{x}, \\ \mathbf{Q}^e &= \oint_{\Gamma_e} \boldsymbol{\phi}^T \bar{\boldsymbol{t}} d\mathrm{s}. \end{aligned} \tag{10}$$

The elemental stiffness matrix $\mathbf{K}^e$ is of the order $3n \times 3n$, the elemental body force vector $\mathbf{F}^e$ and the elemental external force vector $\mathbf{Q}^e$ are of the order $3n \times 1$, where $3n$ is the total degree of freedom (DOF) per element. The matrix $\mathbf{B}$ is also known as strain-displacement matrix and is defined as

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 & \mathbf{B}_3 & .... & \mathbf{B}_n \end{bmatrix}, \tag{11}$$

$$\mathbf{B}_i = \begin{bmatrix} \partial\phi_i/\partial x & 0 & 0 \\ 0 & \partial\phi_i/\partial y & 0 \\ 0 & 0 & \partial\phi_i/\partial z \\ 0 & \partial\phi_i/\partial z & \partial\phi_i/\partial y \\ \partial\phi_i/\partial z & 0 & \partial\phi_i/\partial x \\ \partial\phi_i/\partial y & \partial\phi_i/\partial x & 0 \end{bmatrix} \tag{12}$$

The constitutive matrix $\mathbf{D}$ contains the elastic properties of the material and has the following form for isotropic materials:

$$\mathbf{D} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-2\nu & 0 & 0 \\ 0 & 0 & 0 & 0 & 1-2\nu & 0 \\ 0 & 0 & 0 & 0 & 0 & 1-2\nu \end{bmatrix} \tag{13}$$

where E is the Young's modulus and ν is the Poisson's ratio. A numerical integration scheme based on the Gauss quadrature rule is employed to integrate the expression given by (10). Algorithm 1 presents the basic steps in the numerical integration of the elemental matrices. At each Gauss point, the shape function matrix ($\boldsymbol{\phi}$), and the strain-displacement matrix ($\mathbf{B}$) are computed together with the Jacobian matrix ($\mathbf{J}$). The determinant ($|\mathbf{J}|$) and inverse of Jacobian ($\mathbf{J}^{-1}$), required to transform the shape function values from natural (reference) coordinates to physical coordinates, are also computed. Once the strain-displacement matrix $\mathbf{B}$ is found, the elemental stiffness matrix can be generated by carrying-out the required matrix multiplication.

Algorithm 1 Numerical integration

1: Initialize $\mathbf{K}_e$, $\mathbf{F}_e$ and $\mathbf{Q}_e$ to zero
2: **for** q=1 to Number of Gauss points **do**
3: Read $\boldsymbol{\phi}(q)$ and d$\boldsymbol{\phi}(q)$
4: Read node coordinates
5: Compute $\mathbf{J}$, $|\mathbf{J}|$, and $\mathbf{J}^{-1}$
6: Compute $\mathbf{B}$
7: $\mathbf{K}_e += \mathbf{B}^T\mathbf{D}\mathbf{B}$
8: $\mathbf{F}_e += \boldsymbol{\phi}^T\boldsymbol{b}$
9: $\mathbf{Q}_e += \boldsymbol{\phi}^T\bar{\boldsymbol{t}}$
10: **end for**

Two kinds of data sets are required for FEM computation, one is the nodal data matrix $C(q)$ and another is the connectivity matrix $P(e, i)$. The nodal data matrix contains coordinate values of all the nodes having the global node number q and any other field value associated with the node. The connectivity matrix $P(e, i)$ contains the global node number of the i^{th} node of each element e, i.e., it presents mapping from local node number of an element to global node number. Global matrix is obtained by assembling the elemental matrices of all the elements of a mesh into a system of equations.

3.3. Sequential FEM assembly

The elemental stiffness matrix and the elemental load vectors of all the elements are accumulated to form the global stiffness matrix and the global force vector. The contribution of each element to a non-zero of the global matrix depends on the connectivity pattern of the mesh. Mesh connectivity also influences the sparsity of global matrix. The most common way of sequential global assembly is known as Addto algorithm [29] which is presented in Algorithm 2. The outer loop over all the elements uses an elemental subroutine to calculate the elemental stiffness matrix and load vector. Once these values are calculated, the assembly is done based on a mapping from local DOF to global DOF given by the connectivity matrix $P(e, i)$.

3.4. Parallel FEM assembly using coloring method

The coloring is one of the simplest and widely used methods for avoiding race condition in FEM assembly. It is robust and works well for a wide range of problems and types of elements. As shown by Cecka et al. [5] and Komatitsch et al. [6], the coloring method is particularly suitable for higher order elements. The main idea behind the coloring method is to partition the mesh into sets of elements such that no two elements belonging to the same set have any node in common. Here, each set is identified with a unique color. Assembly can now be done for elements corresponding to one color simultaneously because threads working over different elements will not have any access to the same piece of

Algorithm 2 Sequential FEM assembly or Addto Algorithm

```
1: Initialize K and F to zero
2: for all elements e do
3:    (K^e, F^e) ← elementalSubroutine(e)
4:    for all local node n_1 do
5:       F(P(e, n_1))+ = F^e(n_1)        // Assembly of local force vectors
6:       for all local node n_2 do
7:          K(P(e, n_1), P(e, n_2))+ = K^e(n_1, n_2))
          // Assembly of local matrices
8:       end for
9:    end for
10: end for
```

memory. Assembly for different colors is done in a sequential manner, thus preventing any possibility of race condition. The parallel FEM assembly based on the coloring method is shown in Algorithm 3. CUDA kernel is launched for each color in sequence, whereas in each kernel one thread is assigned to one element. E_k is the mapping from the local element number in the k^{th} color set to the global element number. Input data are accessed by each thread for its global element number. Once the computation of elemental stiffness matrix and load vector is over, assembly into the global matrix is done by the same kernel.

Algorithm 3 Parallel FEM assembly using the coloring method on GPU

```
1: for all colors k do
2:    threadId ← blockDim.x * threadIdx.y + threadIdx.x
3:    if threadId < N_k then        // N_k: Number of elements in k^th color
4:       e = E_k(threadId)          // E_k: Local to global element mapping
5:       compute K^e and F^e
6:       K ← Accumulate(K^e)
7:       F ← Accumulate(F^e)
8:    end if
9: end for
```

While using assembly by element strategy, one must decide the memory type for storing input and output data like node coordinates, elemental stiffness matrix, and connectivity matrix. The selection is to be made between on-chip and off-chip memories provided by GPU. The off-chip global memory is slow and bigger in size, whereas the on-chip memory is fast but smaller in size. Along with input and output data, geometrical parameters (Jacobian, determinant, and inverse of Jacobian) required in numerical integration are also stored in between the computations. Thus, the selection of memory types for all these data must be done judiciously accounting its effect on performance. Also, it shows the number of different implementations that can be produced based upon the selection of memory type.

In our experiments, NVIDIA card based on Kepler architecture is used and we decide the memory type for assembly accordingly. The on-chip memory resources available on NVIDIA Tesla K40 GPU are 48 KB shared memory per SM and 65536 registers per SM. For full occupancy, shared memory is shared by 2048 threads (Kepler architecture). Hence for each thread, allowed number of values to store on shared memory is given by

$$\frac{48 \times 1024}{2048 \times 4} = 6, \tag{14}$$

considering the size of each value equal to four bytes. The number of registers available per thread shall be 32. This is in contrast to the requirement posed by our test problem using an 8-noded hexahedral element. Each element requires space to keep 24 values for coordinates of eight nodes and eight values for global connectivity of each node. Since, there are eight nodes with three DOF per node, the total degrees of freedom per element is 24. Consequently, the size of the element stiffness matrix becomes 576. In addition to this, geometry parameters in numerical integration also need to be stored. We can easily observe the difference between the amount of fast memory resource available and the size of data that an element is required to work with. This indicates that the assembly strategy using a single thread per element put a constraint on the usage of fast on-chip memory for data that are frequently needed. We address this issue in

our proposed assembly strategies by assigning a warp to an element.

We make an attempt to optimize assembly by element strategy (using a single thread per element) for our test problem. The implementation, which we later use to compare with our proposed assembly strategies, uses local memory to store the element stiffness matrix. The numerical integration uses an outer loop over Gauss points while two inner loops over shape functions. Geometrical parameters are calculated once for each Gauss point and are used within inner loops to calculate all stiffness matrix entries. The derivatives of shape functions in natural coordinates are pre-computed on the CPU. It is stored in the shared memory. Nodal coordinates are read in a coalesced manner every time when required. Once computation of the element stiffness matrix is over, connectivity matrix is loaded in a coalesced way into the local memory. The assembly to the global matrix is done in CSR format with the help of the connectivity matrix and the precomputed index into CSR value array.

4. Proposed Assembly Strategies for GPU

Our proposed assembly strategies use 32 threads (a warp) per element for computation of elemental stiffness matrix and assembly into global matrix. As discussed in section 3.4, for a strategy using a single thread per element, a thread can have storage space of six values in a shared memory having a size of four bytes each. By the same logic, assigning a warp to an element allows us to keep $32 \times 6 = 192$ values per element in shared memory. This is more space than required to keep nodal data and connectivity matrix. Our proposed strategies also use shared memory to keep intermediate parameters in numerical integration. More space in shared memory implies low register pressure, higher data reuse and very few or no local memory access. The use of 32 threads to compute the elemental stiffness matrix also entails reduced arithmetic intensity on each thread. One warp is assigned to one element for computation of elemental matrix as well as assembly. We divide the computation equally among all the threads of a warp. Thus for an element having 576 entries in elemental

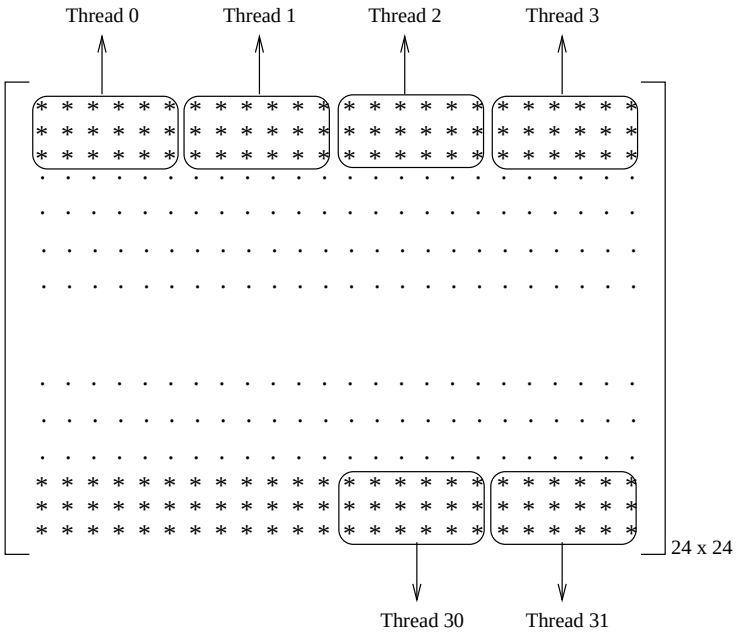

Figure 2: Distribution of elemental matrix entries for 4 threads per row for an 8-noded hexahedral element with 3-DOF per node.

stiffness matrix, one thread is required to compute 18 entries. We can choose to assign any 18 entries to a thread out of total 576. But considering its effect in implementation and also in the assembly, we investigate only two types of choices. We call our strategies: "Assembly by warp", "Assembly by warp using PPNI", and "Assembly by two kernels" respectively. In the following sections, different variants of the strategies are presented. The assembly strategies proposed in this article provide an efficient GPU kernel to use with the popular coloring method.

4.1. Assembly by warp using 4 threads per row

The selection of 18 entries for a thread in a warp to work upon is shown in Figure 2. The entries of the elemental stiffness matrix (24×24) for an 8-noded hexahedral element with three DOF per node are divided among 32 threads. The entries assigned to each thread span over three rows and six columns. The assembly by warp using 4 threads per row is presented in Figure 3.

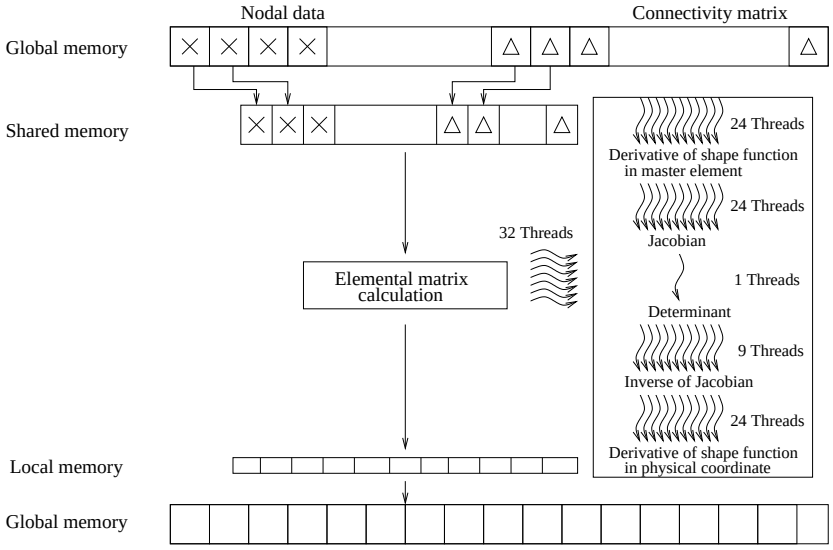

Figure 3: Assembly by warp using 4 threads. Each thread calculates its allotted entries independently and stores them into the local memory.

Nodal coordinates and connectivity matrix are laid together in global memory in an element-wise manner. Such reordering of input data is done for each color set. This enables a warp to read 24 values for coordinates and 8 connectivity values at once in the most efficient way (coalesced). The derivatives of the shape functions in natural coordinates are pre-computed on CPU for all the Gauss points. Once these data are loaded into shared memory, we proceed with numerical integration by involving many threads of the warp. The Jacobian is calculated by multiplying the coordinates of nodes with shape function derivative. The schematic for the Jacobian computation is shown in Figure 4. The 24 threads read 24 values of shape function derivative from shared memory and multiply with eight values of nodal coordinates. The warp shuffle feature of CUDA is used for the reduction of eight values spread over eight threads. Here at a time, three entries of Jacobian are calculated. It requires two more passes of the computation to find total nine entries of the Jacobian.

The determinant is then calculated by using one thread as shown in Figure 3. The determinant is found by a conventional method involving co-factors

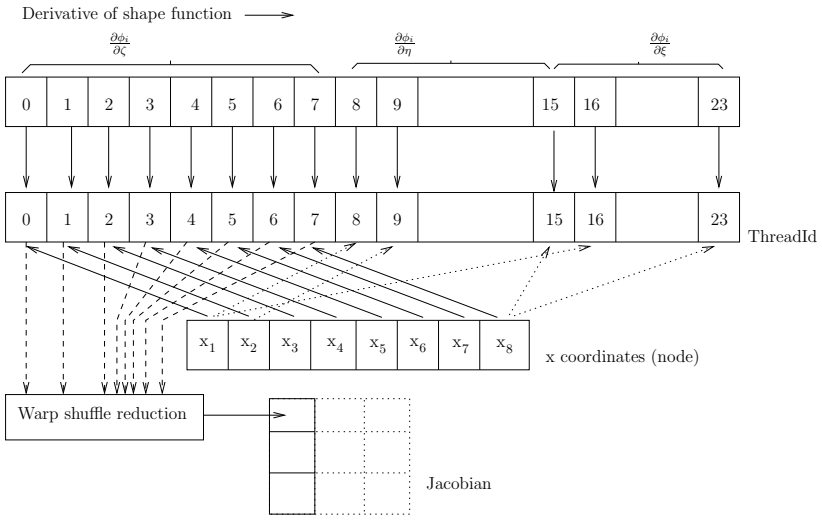

Figure 4: The access pattern for Jacobian calculation is shown. Threads store the multiplication value in their registers for utilization in warp shuffle reduction.

Table 1: Overall running time of assembly by warp (4 threads) strategy for three different approaches for calculating Jacobian. Problem solved for 6.5 million degrees of freedom.

Jacobian calculation approach	Time (sec)
Atomic functions	1.39
Shared memory	1.09
Warp shuffle	1.03

evaluation. A total of nine threads are used to find the inverse of the Jacobian. Each thread calculates co-factor of one entry of Jacobian and stores in their registers. These values in registers are divided by the value of determinant from shared memory. Finally, it is transposed to store in shared memory by overwriting the Jacobian. We again use 24 threads to find derivative of shape function in physical coordinates. Numerical integration steps are shown in Algorithm 4. The other approaches for calculating Jacobian are also investigated. Table 1 shows the running time of assembly by warp (4 threads) for approaches using atomic functions, shared memory and warp shuffle. The approach using atomic function takes approximately 34 % more time than that using warp shuffle. The shared memory based approach consumes 5.8 % more time than warp shuffle and requires additional space in shared memory. Since warp shuffle-based approach takes the least time and requires no additional space in shared memory, we opt it to find Jacobian in this paper.

Computation of entries of the elemental matrix is done by each of the thread independently. Each thread evaluates the multiplication $\mathbf{B}^{\mathrm{T}}\mathbf{DB}$ for their 18 entries and store in local memory. Threads pick their corresponding shape function derivative in physical coordinates from shared memory and perform multiplication with material property matrix $\mathbf{D}$. With proper indexing into shape function derivative array, we could do this computation without any branching in code.

Each thread is now responsible to assemble its own 18 entries into global matrix. During assembly, each thread accesses their calculated entries stored in a local array using the same index, which results in zero replay overhead. Therefore, values are accessed from local memory in a most efficient way. Assembly is done into the global matrix having CSR format. Entries of the element stiffness matrices are accumulated into the value array of CSR format. The location of entries into the value array is precomputed on CPU and sent to GPU. We call it CSR index. Since we are using an 8-noded element, a total of 64 entries in CSR index are required for each element. Each row of the element stiffness matrix requires eight CSR indices to accumulate into the global matrix. As discussed in the beginning of this section, four threads are responsible for assembly of all

Algorithm 4 Numerical integration by a warp

1: $threadId \leftarrow blockDim.x * threadIdx.y + threadIdx.x$ // Indicies of threads in a block.
2: $LaneId \leftarrow threadId \ \% \ 32$ // Indicies of threads in a warp.
3: $Derivative_shapefn_nat \leftarrow$Read shape function derivative in natural coordinate from global memory into shared memory.
4: **for** all the Gausspoints **do**
5: **if** $LaneId < 24$ **then**
6: $Jacobian \leftarrow jacobian()$
7: **end if**
8: **if** $LaneId < 1$ **then**
9: $Determinant \leftarrow Calculate\_determinant()$.
10: **end if**
11: **if** $LaneId < 9$ **then**
12: $Jacobian \leftarrow Calculate\_Inverse()$.
13: **end if**
14: **if** $LaneId < 24$ **then**
15: Evaluate derivative of shape function in physical coordinate.
16: **end if**
17: Calculate elemental stiffness matrix
18: **end for**

entries in a row. Therefore, they access CSR indices in two passes. In the first pass, indices located at even places are accessed, whereas in the second pass indices at odd places are accessed. The storage pattern for CSR index is shown in Figure 5. For each element, the CSR indices are stored in two separate groups based on their positions on even and odd places. As shown in the figure, the even place indices are represented by the cyan color. The data is stored in the column-major convention so that all the warps read contiguous memory locations. Each thread accumulates value into the global matrix (the value array of CSR format) by using connectivity matrix and CSR index. Access to the global matrix is not coalesced as different threads of a warp accumulates values to strided locations in global memory.

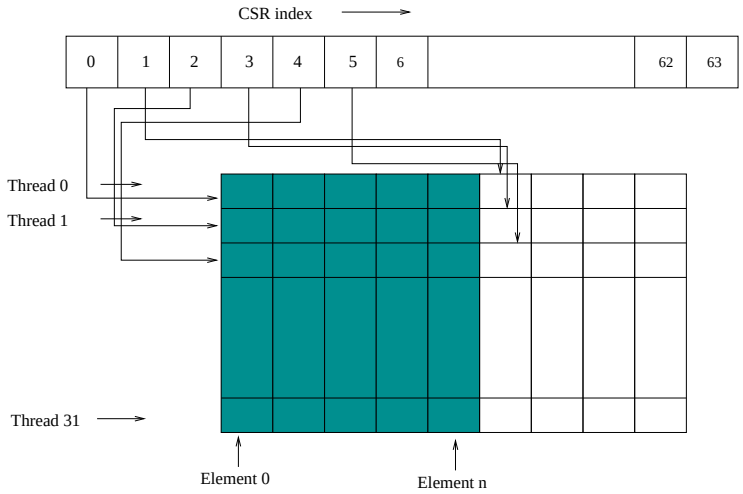

Figure 5: The CSR index data read pattern.

4.2. Assembly by warp using 8 threads per row

In this strategy, the threads now work over the different set of entries. Each thread is assigned entries spanning over six rows and three columns of the elemental matrix as shown in Figure 6. The figure represents the distribution of elemental matrix entries in a similar way as discussed in section 4.1.

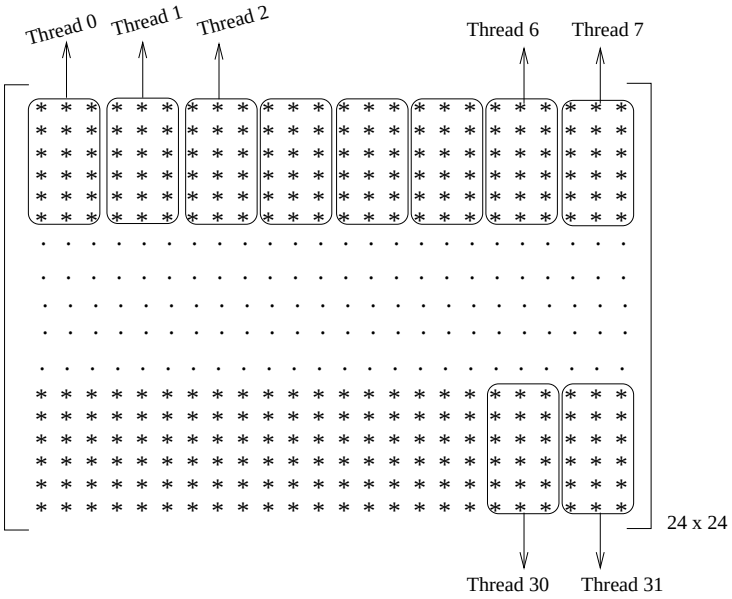

Figure 6: Distribution of elemental entries for 8 threads per row.

The input data is read as described using Figure 3. The numerical integration also follows exactly the same strategy as given in Algorithm 4. Since each thread now computes a different set of 18 entries, the approach to the computation of $\mathbf{B}^{\mathrm{T}}\mathbf{DB}$ changes accordingly. The CSR index is read in two passes as in the previous strategy. It now uses different criterion to partition CSR indices into two groups. The 18 entries allotted to a thread now span over six rows of element stiffness matrix. These correspond to six degrees of freedom of 2 nodes. The assembly now takes place for all degrees of freedom of 1^{st}, 3^{rd}, 5^{th} and 7^{th} nodes first. This requires corresponding CSR indices in the first pass. In the second pass, the CSR indices for 2^{nd}, 4^{th}, 6^{th} and 8^{th} nodes are read. The data is stored and read in a similar way as in the previous strategy. The access to the global memory for assembly is still not coalesced, but it is more localized compared to the previous strategy. In the strategy using 4 threads per row, a warp had to accumulate values in eight different rows of the global matrix at a time, whereas in this strategy values are accumulated in four different rows

only. This reduces the number of global memory transactions required in this strategy using 8 threads per row of elemental stiffness matrix.

4.3. Assembly by warp using PPNI

In this strategy of assembly, we develop a partial parallel implementation of numerical integration (PPNI) for assembly by a warp. Computation of element stiffness matrix requires an evaluation of weak form at each of the Gauss points followed by their summation. For each Gauss point, evaluation of numerical integration requires calculation of many geometrical parameters (Jacobian, determinant etc.). The previous strategies used a loop over the Gauss points for this computation. In PPNI, we calculate the geometrical parameters for all the Gauss points in parallel, whereas the weak form evaluation at each of the Gauss points is done inside a loop.

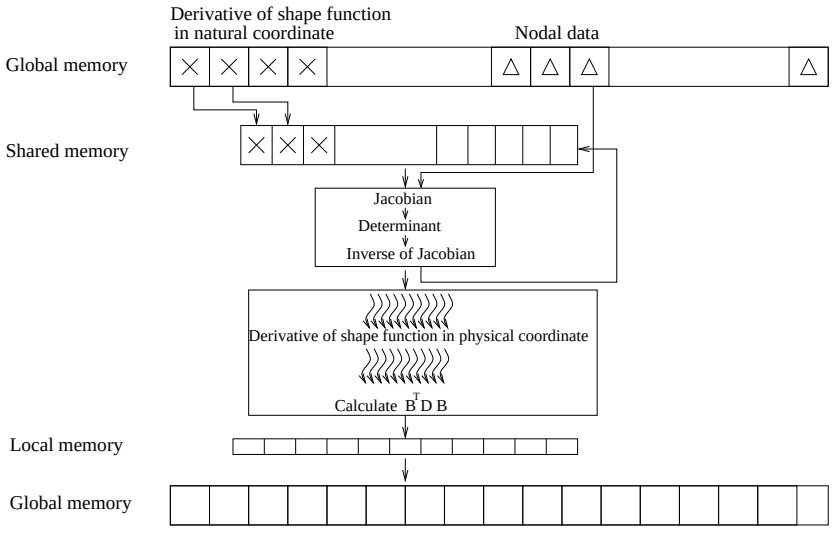

Figure 7: Assembly using partial parallel numerical integration (PPNI) strategy.

The schematic of the assembly using PPNI by a warp is shown in Figure 7. The nodal coordinates and derivatives of shape function in natural coordinates constitute the input data for computation of the element stiffness matrix. The derivatives of shape functions in natural coordinates are calculated for all the

Gauss points on CPU. It is stored in the shared memory of GPU so that all warps of a thread block can access. Like previous strategies, one warp is assigned to one element. Jacobians for all the Gauss points are calculated simultaneously and stored in shared memory. This requires space for 72 entries per element to store all Jacobians corresponding to eight Gauss points. The Jacobian calculation follows the same procedure as described in assembly by warp using 4 threads, except now it uses all the threads of a warp. In the previous strategy, three entries of the Jacobian were calculated using 24 threads of a warp at a time. Now it uses all 32 threads to calculate four entries of the Jacobian. This computation is repeated 18 times to calculate all 72 entries. The listing 1 shows the code for the Jacobian evaluation. The Jacobian calculation is followed by the calculation of determinant and inverse of Jacobian for all the Gauss points. Eight threads are used to calculate eight determinants. Each warp makes use of 27 threads to calculate the inverse of three Jacobians at a time. The threads are taken in multiple of nine such that one thread is assigned to find cofactor of one entry of a Jacobian. Each thread divides their cofactor by the corresponding determinant. The inverse of the Jacobian is stored in the shared memory by overwriting the Jacobian. Remaining part of the algorithm such as computation of shape function derivatives in physical coordinate and elemental matrix remain the same as previous strategies. Computation of $\mathbf{B}^{\mathrm{T}}\mathbf{DB}$ is done sequentially at each Gauss points by evaluating derivative of shape function in physical coordinates. Element stiffness matrix is stored in local memory of all threads of the warp. Once the elemental stiffness matrix is found, the connectivity matrix is loaded into the shared memory and assembly is done. The proposed PPNI assembly strategy is implemented using 4 threads per row and 8 threads per row assignment.

Listing 1: Jacobian evaluation using warp shuffle. The code shows computation of 24 entries of Jacobian. Corresponding to 8 Gauss points we have 8 Jacobians having total 72 entries.

```
#pragma unroll
  for(int i=0;i<6;i++)
  {
```

```
    float abc=dshapefn_natural[i*32+laneid]*
        nodal_data[connect_index*24+local_elementno*8+laneid%8];
    for (int j=4; j>=1; j/=2)
        abc += __shfl_xor(abc, j, 8);
    if((laneid %8)==0)
        {jacobian[wid*72+(laneid/8)*3+12*i]=abc;}
}
```

4.4. Assembly by two kernels

In the previous three strategies, a single kernel for each color is used. In this strategy, two kernels are used so that computational load at the single complex kernel is split into two kernels. These two kernels are optimized to give the best result for the task they are assigned to do. The first kernel calculates geometrical parameters required to transform shape function values and its derivatives from the natural coordinates to the physical coordinates. The second kernel calculates the element stiffness matrix and accumulates into the global matrix. This strategy is shown in Figure 8.

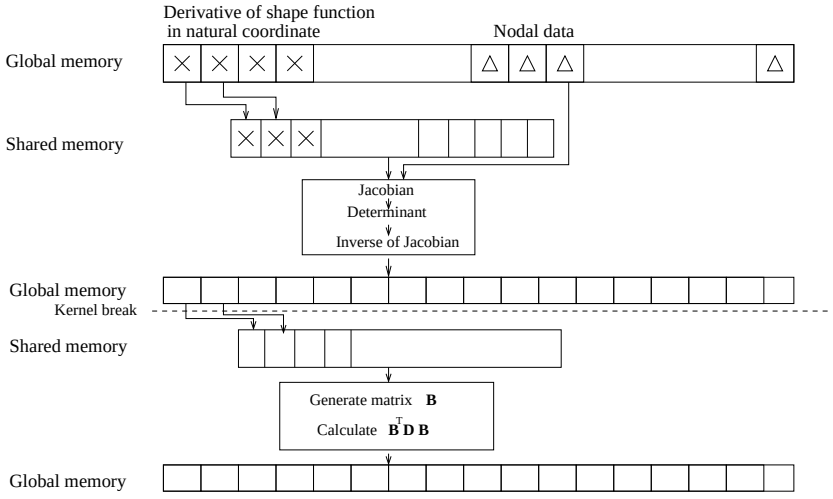

Figure 8: Assembly strategy using two kernels. The first kernel computes inverse of Jacobian corresponding to all the Gauss points, whereas in the second kernel evaluation and assembly of element stiffness matrices are done.

The first kernel reads input data (nodal coordinates and shape function derivatives in natural coordinates) from global memory and store into shared memory. The calculation of Jacobian, determinant, and the inverse of Jacobian is done in the same way as described in assembly by warp using PPNI. Once the inverse is calculated, it is stored in global memory along with determinant. The inverse is then read in the coalesced manner in the second kernel. The second kernel assigns a thread block of 576 threads to an element. Since there are 576 entries in the element stiffness matrix, one thread is responsible for calculation and assembly of one entry. The strain matrices ($\mathbf{B}$) are calculated for all the Gauss points simultaneously and stored in shared memory. Each thread calculates their corresponding entry of the element stiffness matrix for each Gauss point in sequence and adds them to obtain the final value. Like previous strategies, the assembly is done into the CSR format in an uncoalesced way. The entries of the element stiffness matrix are assigned to threads in a row-wise manner such that the first 24 threads are allotted entries in the first row and next 24 in the next row. This ensures that during assembly, for consecutive entries in the element stiffness matrix their corresponding threads access either consecutive locations or locations having smaller stride in the global matrix. This provides the higher level of localization in data access than strategies using 4 threads and 8 threads per row. Consequently, there are less global memory transactions and a higher amount of cache hit. This reduces the effect of uncoalesced access more effectively than previous strategies.

5. Results and Discussion

5.1. Problem statement

The proposed assembly strategies are compared with the most common strategy of assembly by element on a 3D elastic cantilever beam problem. Figure 9 shows a 3D cantilever beam with point loads at one end and the other end fixed. For simplicity, an 8-noded brick element of equal sides is considered for meshing. However, the proposed strategies do not use any geometric property

of the mesh and treat them as unstructured. The coloring of the mesh is done by the greedy algorithm as given in [30]. The coloring algorithm is implemented in CPU which takes about 1.4 seconds for the finest mesh of 2,097,152 elements. The problem is solved with geometric and material properties as given in the Table 2.

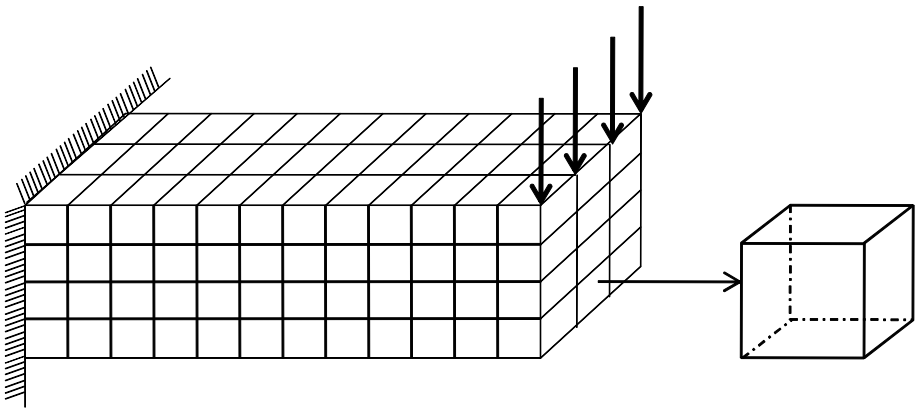

Figure 9: A 3D cantilever beam with boundary condition and mesh element.

Table 2: Geometric and material properties of a cantilever beam problem.

Length (l)	Breadth (b)	Height (h)	Young's modulus (E)	Poisson's ratio (ν)
16.0 m	2.0 m	2.0 m	200 GPa	0.333

In order to verify the efficiency of the proposed strategy for different problem sizes, four types of meshes are taken as given in Table 3. The boundary condition is implemented after assembling the system of equations in which the corresponding rows and columns of nodes on the boundary are made zero.

The strategies are tested on a hybrid CPU-GPU environment that consists of NVIDIA Tesla K40 GPU and Intel Xeon(R) E5-2650 CPU. The system uses GPU of compute capability 3.5 and CUDA runtime version 8.0. The GPU has 2880 CUDA cores clocked at 875 MHz with 384 bits bus width. The global memory is clocked at 3004 MHz. It also has configurable L1 cache and unified

Table 3: Finite element meshes.

Mesh	Elements	Nodes	Degrees of Freedom
Mesh 1	110 592	120 625	361875
Mesh 2	262 144	279 873	1055499
Mesh 3	1 000 000	1 043 001	3129003
Mesh 4	2 097 152	2 167 425	6 502 275

L2 cache. CUSP library [14] is used for the solution of the linear system of equations. CUDA Event API functions are used for reporting the execution time.

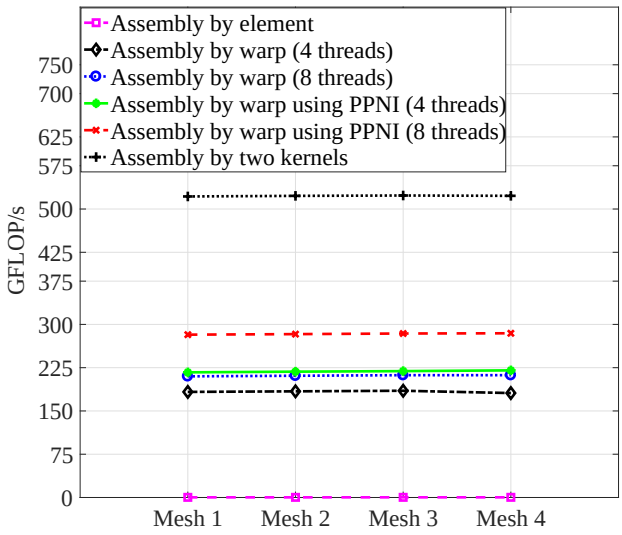

Figure 10: Effective arithmetic throughput obtained from assembly strategies.

5.2. Code Profiling and Performance Results

A comparison of the arithmetic throughput and memory bandwidth achieved by the proposed strategies with the theoretical peak values of NVIDIA Tesla K40 is done to show the level of utilization of the GPU. The Tesla K40 card has a peak memory bandwidth of 288 GB/s and a peak arithmetic throughput of 4096 GFLOP/s for a single precision floating point computation. NVIDIA's

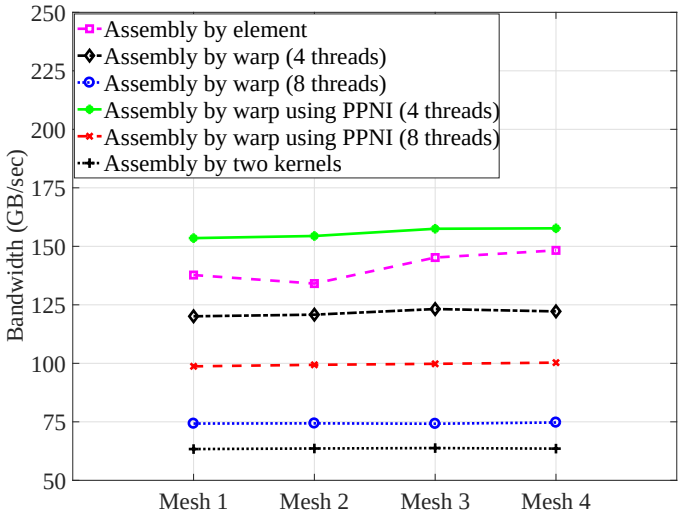

Figure 11: Effective bandwidth obtained from assembly strategies.

command line profiler `nvprof` is used for the measurement which gives effective arithmetic throughput and effective memory bandwidth observed by the hardware. The effective arithmetic throughput is calculated as

$$\text{Arithmetic throughput (GFLOP/s)} = \frac{\texttt{flop\_count\_sp}}{\text{kernel time (sec)} \times 10^9},$$

whereas the effective memory bandwidth is given by

$$\text{Bandwidth (GB/s)} = \frac{(\texttt{dram\_read\_transactions} + \texttt{dram\_write\_transactions}) \times 32}{\text{kernel time (sec)} \times 10^9},$$

where `flop_count_sp`, `dram_read_transactions`, `dram_write_transactions` are the metrics given by `nvprof`. Figures 10 and 11 show the effective arithmetic throughput and effective memory bandwidth respectively for the various strategies as a function of mesh size. The arithmetic throughput of assembly by element strategy is found to be the least at 0.213 GFLOP/s. The proposed strategies show significant improvement over it and achieve throughput in the range of 181 GFLOP/s to 523 GFLOP/s. The highest arithmetic throughput is achieved by the second kernel of assembly by two kernels strategy, reaching 12.7% of the peak value of Tesla K40. The device memory bandwidth utilization of two of our proposed strategies is found to be less than that of assembly

by element strategy. The maximum bandwidth achieved by assembly by element strategy is found to be 51.4% of the peak bandwidth, whereas assembly by warp using PPNI (4 threads) achieve the highest value reaching 54.5% of the peak bandwidth. The strategy using 8 threads uses less bandwidth than that of 4 threads. This behavior is expected since strategy using 8 threads requires less number of transactions to device memory due to the reason explained in section 4.2. The arithmetic throughput and memory bandwidth achieved by the proposed strategies are found to be much less than the peak values of the device. This indicates that the proposed kernels are not able to saturate the hardware resources. However, the results show a better utilization of GPU by our proposed strategies than assembly by element.

Table 4: Performance metrics for different strategies obtained for 2.01M elements and 6.5M degree of freedom.

Assembly strategies	ach_occ	l1_c_lhit	local_mem
Assembly by element	0.49	11.05%	55.38 %
Assembly by warp (4 threads)	0.49	91.80%	3.53%
Assembly by warp (8 threads)	0.49	99.87%	0.13%
Assembly by warp using PPNI (4 threads)	0.56	0%	0%
Assembly by warp using PPNI (8 threads)	0.56	0%	0%
Assembly by two kernel	0.57	0%	0%

ach_occ : `achieved_occupancy`, l1_c_lhit: `l1_cache_local_hit_rate`, local_mem: `local_memory_overhead`.

Further insights into the implementation of our proposed strategies can be obtained by the performance metrics given by nvprof. Table 4 lists some of the metrics (`achieved_occupancy` measures occupancy, `l1_cache_local_hit_rate` gives the hit rate in the L1 cache for local loads and stores, and `local_memory_overhead` gives the ratio of local memory traffic to total memory traffic between L1 and L2 caches) for the assembly strategies. Not much improvement in occupancy is observed for the proposed strategies against assembly by element. Occupancy

in proposed strategies is limited by registers use. An attempt to increase occupancy by controlling register use by `-maxrregcount` nvcc directive is found to have an adverse effect on execution time. There is a spillage of registers into local memory for some of the strategies. The local memory overhead for assembly by element strategy is found to be at 55.38%. This indicates a huge performance penalty. The request for local memory access is less expensive when it gets a cache hit in L1. The `l1_cache_local_hit_rate` for assembly by element is at 11.05%, whereas assembly by warp strategy shows a huge improvement achieving up to 99.87% cache hit. The higher hit rate for local memory access in L1 cache reduces the amount of memory transfer from device memory. This is reflected by the lesser value of `local_memory_overhead` in case of assembly by the warp. The zero values of `local_memory_overhead` and `l1_cache_local_hit_rate` for assembly by warp using PPNI and assembly by two kernels indicate that there is no register spillage for these two strategies.

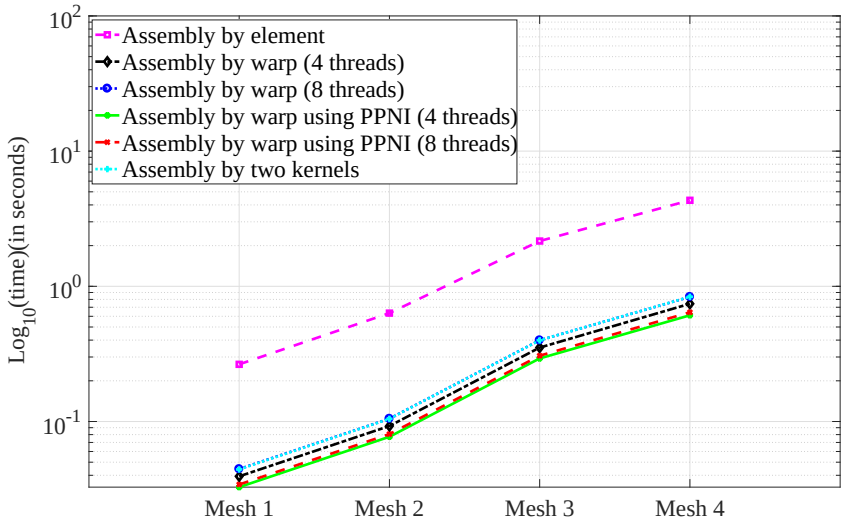

Figure 12: Assembly kernel running time for different mesh sizes.

The relative running time of the assembly kernel of our proposed strategies is presented in Figure 12. It shows the time spent in the computation of elemental stiffness matrix and assembly. Running time is measured using CUDA

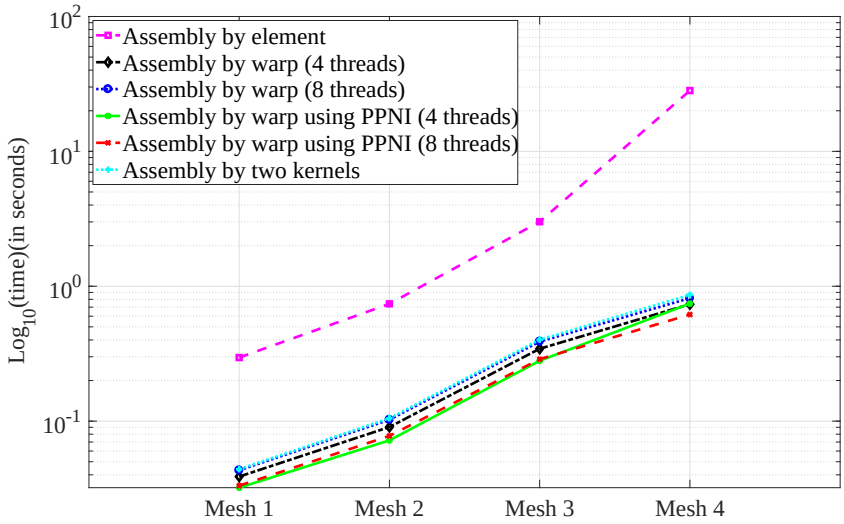

Figure 13: Assembly kernel running time for ANSYS mesh.

Event API functions. The results show a significant reduction in assembly time for our proposed strategies. The assembly by warp using PPNI achieves the least running time for all the mesh sizes. This is closely followed by assembly by warp strategy and assembly by two kernels. The comparison of kernel running time also suggests that the strategy taking lesser time has the bandwidth and GFLOP/s on the higher side. The assembly by warp using PPNI has better bandwidth and GFLOP/s than assembly by warp and assembly by element strategies. The assembly by two kernels strategy having best GFLOP/s, but lowest bandwidth takes more time than assembly by warp. The relatively inferior performance of assembly by two kernels can be attributed to more number of kernel launches as well as the global memory write and read in between the two kernels. The performance penalty due to local memory access in assembly by warp strategy seems to be significantly reduced due to a higher hit rate in the L1 cache. The running time of assembly by warp strategy is found to be much lesser than assembly by element, whereas both the strategies have the same level of occupancy. We also find that implementation of numerical integration has a significant effect on assembly kernel time. The assembly by warp using PPNI

takes much lesser time than the assembly by warp strategy, while both these strategies use the same thread assignment and have the same implementation to calculate $\mathbf{B}^{\mathrm{T}}\mathbf{DB}$. The difference lies in the partial parallel implementation of numerical integration, which not only improves occupancy and removes local memory access, but also enables better utilization of GPU by achieving better arithmetic throughput and bandwidth. Assembly strategies using 4 threads are found to be a bit better in terms of running time than those using 8 threads. The lesser requirement of bandwidth in strategies using 8 threads does not seem to be a dominating factor in kernel execution time. The performance of the proposed strategies is also evaluated on realistic meshes. The mapped mesh of the cantilever beam has been generated in ANSYS software package using the brick elements and with the mesh parameter as given in Table 3. To assess the impact of node connectivity on the performance, we follow the same data as given by ANSYS. Figure 13 shows the execution time of different strategies for the meshes generated by ANSYS. The proposed strategies achieve the similar timings as found in Figure 12. The assembly by element strategy seems to be highly dependent on the connectivity as its performance degrades significantly with the increasing mesh size. The initial results suggest that the warp based strategies are least affected by the node connectivity. However, more detailed study is required and can be taken as future work.

The speedup of our proposed strategies over assembly by element strategy on GPU is shown in Figure 14. The best speedup is achieved by assembly by warp using PPNI (4 threads) strategy for all the mesh sizes. It achieves the maximum speedup of 8.2× for 262,144 elements and 7.09× for the finest mesh consisting of 2.09 million elements. The assembly by warp strategy achieves speedups of 5.82× and 5.17× respectively for 4 threads and 8 threads variant for the mesh containing 2.09 million elements. The assembly by two kernel strategy achieves speedup in the range $5.17\times - 6.05\times$.

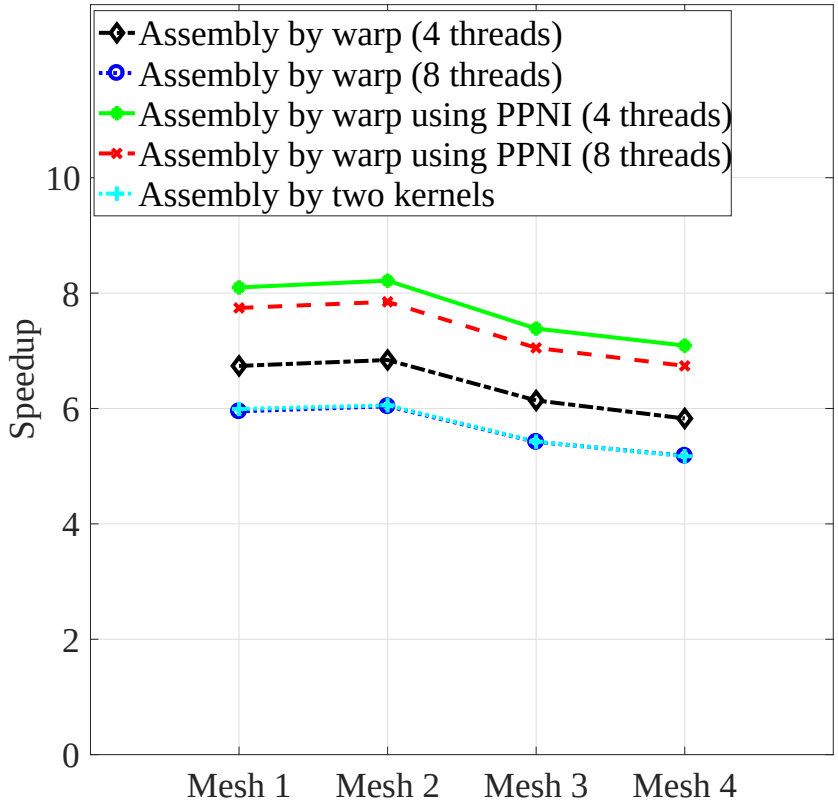

Figure 14: Speed up obtained for different strategies over assembly by element strategy on GPU.

6. Conclusion and future work

This paper attempts to find an effective strategy to implement the FE matrix generation and assembly on GPU for an elasticity problem. Three assembly strategies named as assembly by warp, assembly by warp using PPNI and assembly by two kernels, based on the popular coloring method have been presented for the linear hexahedral element. The focus of the proposed strategies was to optimize numerical integration in matrix generation and assembly step of FEM. All the proposed strategies made use of the warp shuffle feature of CUDA to calculate Jacobian efficiently. Assembly was done to the global matrix in the CSR format. The performance of the proposed strategies was compared with established assembly by element strategy. The test case was run on NVIDIA Tesla K40 card in single precision to obtain the running time of assembly kernels. Compared to the assembly by element strategy, the speedup in the range $6.73\times - 8.21\times$ has been obtained by assembly by warp using PPNI for different mesh sizes. The assembly by warp and assembly by two kernels achieved speedup in the range $5.17\times - 6.84\times$ and $5.17\times - 6.05\times$ respectively. Further, the results of code profiling showed better arithmetic throughput and bandwidth of our proposed strategies over the assembly by element. In the present work, the kernel has been designed for single-precision. This can limit its usage to certain cases for which a kernel with double-precision has to be designed. Also, the study can be extended to develop efficient GPU kernel for higher order finite elements. Moreover, a parallel implementation for precomputing the locations of entries in a value array of the CSR matrix can be done for reducing overhead.

References

[1] M. Garland, D. B. Kirk, Understanding throughput-oriented architectures, Communications of the ACM 53 (11) (2010) 58–66.

[2] S. Filippone, V. Cardellini, D. Barbieri, A. Fanfarillo, Sparse matrix-vector

multiplication on gpgpus, ACM Transactions on Mathematical Software (TOMS) 43 (4) (2017) 30.

[3] F. Mossaiby, R. Rossi, P. Dadvand, S. Idelsohn, Opencl-based implementation of an unstructured edge-based finite element convection-diffusion solver on graphics hardware, International Journal for Numerical Methods in Engineering 89 (13) (2012) 1635–1651.

[4] S. Georgescu, P. Chow, H. Okuda, Gpu acceleration for fem-based structural analysis, Archives of Computational Methods in Engineering 20 (2) (2013) 111–121.

[5] C. Cecka, A. J. Lew, E. Darve, Assembly of finite element methods on graphics processors, International Journal for Numerical Methods in Engineering 85 (5) (2011) 640–669.

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

[7] Z. Fu, T. J. Lewis, R. M. Kirby, R. T. Whitaker, Architecting the finite element method pipeline for the gpu, Journal of computational and applied mathematics 257 (2014) 195–211.

[8] P. Macioł, P. Płaszewski, K. Banaś, 3d finite element numerical integration on gpus, Procedia Computer Science 1 (1) (2010) 1093–1100.

[9] A. Dziekonski, P. Sypek, A. Lamecki, M. Mrozowski, Finite element matrix generation on a gpu, Progress In Electromagnetics Research 128 (2012) 249–265.

[10] J. Zhang, D. Shen, Gpu-based implementation of finite element method for elasticity using cuda, in: High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous

Computing (HPCC_EUC), 2013 IEEE 10th International Conference on, 2013, pp. 1003–1008. doi:10.1109/HPCC.and.EUC.2013.142.

[11] I. Reguly, M. Giles, Finite element algorithms and data structures on graphical processing units, International Journal of Parallel Programming 43 (2) (2013) 203–239.

[12] J. Bolz, I. Farmer, E. Grinspun, P. Schröoder, Sparse matrix solvers on the gpu: Conjugate gradients and multigrid, ACM Transactions on Graphics 22 (3) (2003) 917–924. doi:10.1145/882262.882364.
URL http://doi.acm.org/10.1145/882262.882364

[13] R. Li, Y. Saad, Gpu-accelerated preconditioned iterative linear solvers, The Journal of Supercomputing 63 (2) (2013) 443–466.

[14] S. Dalton, N. Bell, L. Olson, M. Garland, Cusp: Generic parallel algorithms for sparse matrix and graph computations, version 0.5.0 (2014).
URL http://cusplibrary.github.io/

[15] W. Bosma, J. Cannon, C. Playoust, The Magma algebra system. I. The user language, J. Symbolic Comput. 24 (3-4) (1997) 235–265, computational algebra and number theory (London, 1993). doi:10.1006/jsco.1996.0125.
URL http://dx.doi.org/10.1006/jsco.1996.0125

[16] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, et al., Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods, SIAM Journal on Scientific Computing 37 (5) (2015) S602–S626.

[17] R. Mafi, S. Sirouspour, Gpu-based acceleration of computations in nonlinear finite element deformation analysis, International journal for numerical methods in biomedical engineering 30 (3) (2014) 365–381.

[18] Y. Cai, G. Wang, G. Li, H. Wang, A high performance crashworthiness simulation system based on gpu, Advances in Engineering Software 86 (2015) 29–38.

[19] L. Ram, D. Sharma, Evolutionary and gpu computing for topology optimization of structures, Swarm and Evolutionary Computation 35 (2017) 1–13.

[20] P. Płaszewski, K. Banaś, P. Macioł, Higher order fem numerical integration on gpus with opencl, in: Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on, IEEE, 2010, pp. 337–342.

[21] S. Ohshima, M. Hayashi, T. Katagiri, K. Nakajima, Implementation and evaluation of 3d finite element method application for cuda, in: High Performance Computing for Computational Science-VECPAR 2012, Springer, 2012, pp. 140–148.

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

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

[24] R. Zayer, M. Steinberger, H.-P. Seidel, Sparse matrix assembly on the gpu through multiplication patterns, in: High Performance Extreme Computing Conference (HPEC), 2017 IEEE, IEEE, 2017, pp. 1–8.

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

[26] NVIDIA, Cuda toolkit documentation v8.0 (2016).

[27] NVIDIA Corporation, NVIDIA CUDA C programming guide, version 8.0 (2016).

[28] J. N. Reddy, An introduction to the finite element method, 3rd Edition, Vol. 2, McGraw-Hill New York, 1993.

[29] G. R. Markall, D. A. Ham, P. H. Kelly, Towards generating optimised finite element solvers for gpus from high-level specifications, Procedia Computer Science 1 (1) (2010) 1815–1823.

[30] J. Martínez-Frutos, P. J. Martínez-Castejón, D. Herrero-Pérez, Fine-grained gpu implementation of assembly-free iterative solver for finite element problems, Computers & Structures 157 (2015) 9 – 18. doi:https://doi.org/10.1016/j.compstruc.2015.05.010.
URL http://www.sciencedirect.com/science/article/pii/S0045794915001479