Performance Comparison of GPU-Based Jacobi Solvers Using CUDA Provided Synchronization Methods

MARIA ASLAM, OMER RIAZ, SHAHZAD MUMTAZ, AND ALI DANIYAL ASIF
Department of Computer Science and IT, The Islamia University of Bahawalpur, Bahawalpur 63100, Pakistan
Corresponding author: Omer Riaz (omer.riaz@iub.edu.pk)

ABSTRACT In this manuscript, variants of Jacobi solver implementation on general purpose graphical processing units (GPGPU) have been purposed and compared. During this work, parallel implementation of finite element method (FEM) using Poisson’s equation on shared memory architecture as well as on GPGPUs has been observed to identify computationally most expensive part of FEM software, which is linear algebra Jacobi solver. Sparse matrices were used for system of linear equations. Nine implementations of Jacobi solver have been developed and compared using various synchronization and computation methods like atomicAdd, atomicAdd_block, butterfly communication, grid synchronization, hybrid and whole GPU based computation methods, respectively. Experiments have showed that Jacobi implementations based on our implemented Butterfly communication method have outperformed CUDA 10.0 provided critical execution methods like atomicAdd, atomicAdd_block and grid methods. The GPU has achieved a max speedup of 46 times using GTX 1060 and 60 times using Quadro P4000 with double precision computations when compared with sequential implementation on Core-i7 8750H. All the developments were performed using C/C++ GNU compiler 7.3.0 on Ubuntu 18.04 and CUDA 10.0.

INDEX TERMS Jacobi solver, FEM, GPU, GTX 1060, Quadro P400, CUDA, butterfly communication, sparse matrix.

I. INTRODUCTION
High Performance Computing is referred as a branch of computer science used for solution of large and highly complex problems in domain of science, engineering and business.

In High Performance Computing, many-core processors have gained more popularity than multi-core CPUs. It consists of hundreds and even thousands of processing units on which thousands or millions of threads can be executed concurrently. Generally, GPUs cannot be stand-alone device that acts as coprocessor or an accelerator for CPU. In the beginning, GPUs were utilized to drive computations needed to show graphics on computers. The most important factor that drives rapid development of GPU hardware is need for realtime 3D graphics in games [1]. Until 2006, it was very challenging for programmers to write programs for early graphics chips in higher level programming interface, as underlying code must fit into APIs that intended to paint each pixel. For that purpose, one must has complete knowledge of Direct3D and OpenGL techniques to program these chips. However, it was not inspiring for researchers until GPUs were considered as special purpose processors. With the release of CUDA, everything changed dramatically. NVIDIA facilitated ease of parallel programming by devoting Silicon areas on chips. As GPUs have been evolved into massively parallel many-threaded multi-core units that supports highly efficient computation of large blocks of data in parallel and high memory bandwidth. The algorithms that include large blocks of data are done more efficiently in GPGPUs rather than on CPUs. The GPU programming strategies are not similar to traditional CPU programming because of dramatical differences in hardware which are discussed below. CUDA facilitated ease of parallel programming on NVIDIA GPUs by providing multi-threaded Single Instruction Multiple Data (SIMD) architecture. In programming context, the set of GPU instructions are referred as kernel, a function written in CUDA C. When host thread invokes kernel, the Compute Workload Distribution (CWD)
unit enumerates blocks of grids and maps them to unoccupied multiprocessors according to execution capacity of that GPU. On each SM block of threads execute concurrently. As soon as blocks finish, CWD unit initiates new blocks on available free multiprocessor. Each MP is designed in a way to execute hundreds of threads in parallel. For management of such a large amount of threads Single Instruction Multiple Thread (SIMT) unit is employed. Each MP’s SIMT unit schedule, manage and execute threads in group of 32 threads, known as Warp. Threads of warp run together at same program address but they have unique instruction address counter and register state. When one or more blocks are given to MP, it first partitioned them into warps that are scheduled and managed by warp scheduler. The SIMT unit selects a warp on time of instruction issue, that is ready to execute and issue next instruction to the active warp. At a time, one common instruction is executed by a warp. Therefore, maximum efficiency can be attained if all threads of warp agree on same execution path.

The GPU execution model is based on launching the kernel on grid of blocks. Each block comprises of group of threads and threads of same block can cooperate and synchronize with each other via fast shared memory. While threads of different blocks can’t synchronize with each other. The grid and block dimensions can be one, two or three and are accessible within kernel by using built in identifiers gridDim and blockDim, respectively and they determine total number of threads [2]. Each thread has unique local id within a block that can be one, two or three dimensional and each block has unique global id. These are combined to get global unique id per thread.

There are three memory spaces in GPUs as Figure 1 shows, discussed in descending order by speed: registers, shared and global memory.

- **The fastest available memory space on GPU is Registers.** Each thread of MP has Registers of private scope. If threads use more number of registers than are available physically, registers will also expand to L1 cache and global memory. Thus, when there are high number of threads, the number of registers available per thread is restricted, which is one of the major reason of why high occupancy may actually affect performance of GPU.

- **The shared memory is second fastest memory space on GPU.** This memory space can be as fast as registers if addressed properly. In GPU computing, shared memory is most powerful tool. The main difference between shared memory and registers is, its ability to share data among several threads. The shared memory is visible to all threads within one block, thus enables cooperation. The limited size of on-chip memory which is 49 KB per block in device of compute capability 6.x or more, is the main hurdle in utilizing registers or shared memory. This memory is organized into 32 banks, that serves 32 threads of one wrap concurrently.

- **The global memory is the third and slowest memory,** which is main memory for GPU. However, it has impressive bandwidth and high latency. Thread-level parallelism is used to hide these memory latencies. The GPUs transfers full cache line across bus for coalesced read, just as CPUs. The cost of transferring full cache line on bandwidth is similar as transferring a single element.

The execution of multiple threads requires some basic means of coordination. We must present mechanism for coordination of threads at block-level and at grid-level. The NVIDIA introduces a barrier synchronization method __syncthreads() for block-level coordination. When this method is called in kernel, all threads of a block have to wait at this point until all threads in the block reach at that point. This guarantees that all threads of block have completed a particular phase before any of them can start next phase of execution. Generally, a barrier synchronization is one of the most popular method of coordinating activities of parallel threads. However, there is also a need of certain mechanism for inter-block synchronization. The package of Cooperative groups describes both block-level and grid-level synchronization strategies. For grid-level synchronization, cudaLaunchCooperativeKernel has to be implemented instead of normal kernel launch<<<>>>>. The device must have support of cudaLaunchCooperativeKernel otherwise it fails to implement grid-level synchronization. When grid.sync() method is called in kernel all blocks of grid have to wait at barrier synchronization point until all of them reach at this point. The basic pattern of execution of barrier statement in grid-level synchronization is similar as in block-level, the only difference is instead of block of N threads the grid of N blocks is taken.

**What are atomic operations?** The operation in which thread can perform memory transaction without interference from other threads, is known as Atomic Operation. They are beneficial in multi-threaded application for synchronization of memory accesses to prevent race condition.
A. OVERVIEW OF FINITE ELEMENTS METHOD

Majority of engineers and scientists do simulation and modelling of physical phenomenon by complex differential equations. Nowadays, most of the problems are approximated by using numerical computation method. Typically, numerical computation method transforms complex partial differential equations into set of discrete equations that are to be approximate using computers.

The most suitable engineering analysis method to approximate solutions of partial differential equation is Finite Element Method (FEM). The partitioning of domain into mesh of discrete points (DOFs) is done to solve differential equations under some appropriate boundary conditions. The principle steps involved in FEM are five, as shown in Figure 4. The accuracy of numerical approximation depends on size and shape of mesh.

- **Laplace equation:**
  
  \[
  \text{Eq: } Au = b
  \]
  
  Poisson: \( -\nabla u = f \)

- **Read mesh file:** For this research work, 2D square domain \( \Omega_{2D} = (0, 1)^2 \) is used and discretized into triangle and line elements as can be seen in Figure 5. The line elements reside at the boundary of \( \Omega \) and will represent how a force will affect boundaries. While triangle elements will reside inside \( \Omega \) and will be used to calculate discrete problem functions. In this work, meshes are generated by a third party software called GMSH [3], which is used for discretization. The boundary elements are used to implement domain constraints like Dirichlet or Neumann etc. The mesh has to be read at initial stage in Finite Element Solver (FES) and is considered as critical part for parallel implementation.

- **Construction of mesh elements:** FEM is a technique which can be implemented to solve any given problem. During this work a third party mesh generator GMSH is used to generate 2D and 3D discrete domains called meshes. These meshes are refined into smaller elements to get better approximation. Meshes will be read by software at its initialization stage. A separate class Gmesh is implemented and its object is responsible for creating element objects like node and elements based on information from mesh file. The node object stores coordinates while element stores node ids, equation constants, force applied and formulation for stiffness matrix and load vector.

- **Construction of linear system of equations:** There are two type of elements irrespective to their shape which are domain element and boundary element. Domain elements contribute in development of stiffness matrix \( A \) and load vector \( \tilde{b} \) as can be seen in Figure 2. While boundary elements only contribute in load vector. For a 2D square domain all the boundary elements which...
have either y-coordinate equal to zero or x-coordinate equal to zero are dirichlet boundary $\Gamma_D$ elements with a constant force zero. The system of linear equation requires a sparse matrix and load vector which are implemented as nested binary trees and array, respectively. There are many compact sparse matrix formats available like CSR (compressed sparse rows), CSC (compressed sparse columns) and COO (Coordinate list) etc, but these formats required structure of sparse matrix be known at construction time. Using object for each element in mesh and allowing each element to populate global stiffness matrix and load vector only allows sparse matrix to populate without knowing possible structure. Secondly, the matrix structure has to be modified while applying Dirichlet constraints. It was better to use dynamic matrix data structures like two level nested binary trees where there will be approximately constant time of data insertion, deletion and searching as well.

**FIGURE 4.** Flow chart of finite element solver.
A collection of studies have been performed in last decade, with definite aim of offloading part or all of computationally expensive parts of finite element method on massively parallel architecture. There were two components in finite element pipeline, that were computationally expensive and challenging: global assembly phase and solution phase of linear system.

Early works on assembly phase like Bolz et al. [8] and Rodríguez-Navarro and Susín Sánchez [9] presented simplest assembly approaches that were designed on the basis of specific application. They solved each non zero entry in parallel on global linear matrix independently, which was best fit for many core architecture. However, these approaches were developed for specific applications which enabled them to solve only simple expressions, and not worked for general finite element applications. Few more complicated, but more general many-core assembly approaches have recently been introduced.

First on device assembly process was developed by Cecka et al. [10]. In global matrix, common DOFs between elements were map at same location. Therefore, these common DOFs share same memory location in global memory. The race condition can occur, when multiple threads access same share memory location concurrently. For global matrix parallel updation, atomic operation was required to avoid the race condition, which resulted in performance bottleneck. Atomic operations were more expensive as they made global memory accesses serialized. Moreover, most of the GPUs do not have support of atomic operations of double precision.

As an alternative of atomic operation, different techniques like, coloring [11], [12] and task dependent graph [13] have been published to avoid race condition. These techniques optimized overheads of synchronization but couldn’t alleviate them completely. The main drawback of both of these approaches was pre-processing which was difficult to parallelize. However, it was done only once, so its computational time effect can be neglected.

Fu et al. [14] introduced a hierarchical assembling approach that overcome the overhead of synchronization efficiently. They developed batch-based strategy for assembling, that guaranteed the coalesced access to global memory and synchronization. The contribution of every DOF was stored in shared memory and final results were accumulated in global memory in chunks. The coalescing of memory access was attained by storing coordinates of node in different arrays rather than single.

Cecka et al. [2] suggested variety of optimization strategies like, optimization of latency of global memory, effective workload distribution, proper shared memory usage and computational power and optimization of transfer of data between host and device. The data containers selection for storage of elemental data was effective for coalesced memory access. The best selection of data container that was utilized for storage of elemental data, coordinates of nodes etc based upon target architecture specifications. The coalesced
memory access can be used to improve efficiency of global memory.

Markall et al. [15] suggested node-wise storage pattern for CPU and element-wise storage for GPU to achieve good performance. This on-device storage pattern increases coalesced memory access of global memory. They proposed two approaches for assembly of global matrix: local matrix and addTo approach.

The addTo approach maps entries of local matrix into global matrix directly using mapping function, it required expensive atomic and coloring operations to avoid conflicts and to guarantee validity of results. While, in second approach exclude global matrix generation, solution vector was assembled after computation of elemental matrix by vector multiplication. Thus, this approach resulted in efficient memory bandwidth usage and improved coalesced memory access patterns. They concluded that second approach was well suited for many core GPUs while fist approach was well suited for multi-core CPU.

Sanfui and Sharma [16] developed an advance version ofaddTo GPU based two kernel approach. In descertization of 3D domain, each DOF was further connected with three DOFs. Therefore, non-zero entries of each DOF will be in three rows of global stiffness matrix. The number of neighboring DOFs determined number of non-zero entries of global stiffness matrix. Determination of column and row indexes associative with each DOF was done by first kernel, and stored in 1D array. Second kernel assembled contribution on the basis of indexes at their particular DOFs. Then, entries of local matrix were mapped to entries of global matrix. The aforementioned approach was modification of addTo approach presented by Markall et al. [17].

Cecka et al. [10], [18] published that there were three available approaches for assembly of global matrix on GPU.

- Assembly by non-zero entry in global matrix row
- Assembly by row of matrix
- Assembly by element

The choice among the above mentioned approaches was dependent on multiple factors like application, order of shape functions and elemental matrix size. They tested the performance of these approaches on many-core GPUs. The authors finally developed two ways for assembly: Assembly by global non-zero entries and assembly by shared non-zero entries. For first approach kernel comprises of two phases. Each thread performed computation for each element and accumulate results in global memory at non-conflict spaces in first part of kernel. In second part, accumulation of each element’s contribution was done. For 2nd approach, shared memory was used for storage of data to avoid global memory access latency. This approach was limited due to shared memory limited size. The approach which include element-wise assembling, each thread performed computations of an element required large number of atomic operation to avoid race condition.

Meng et al. [19] introduced Edge-wise assembly approach to optimize atomic operations. For efficient and faster memory access, each thread assembled row in shared memory. Computation of each thread was independent, thus it resulted in minimization of overheads due to serialization of global access. The number of non-zero entries of global matrix in each row were not guaranteed to be constant. Therefore, if there was a huge difference then it may lead to unbalanced workload distribution among threads. Furthermore, the parallelization pattern for solver phase of FEM was determined by storage pattern of global matrix.

Few more approaches for assembly of matrix were investigated by Reguly and Giles [20], which were assembly by local matrix and matrix free. The first approach directly implemented assembly in sparse matrix storage, while second approach did not include any assembling process. Matrix free approach performed computation in each iteration of solver to compute local matrix. This strategy resulted in optimization of memory requirements for local or global matrix storage, by including some extra calculations in each iteration which was not issue for smaller matrices in many core GPUs. Another author Dziekonski et al. [21]–[23] introduced multiple GPU based assembly for direct sparse matrix. In this approach, first assembling was done in coordinate (COO) storage pattern and finally conversion of COO into CSR was done on many-core GPUs. The limited GPU memory size resulted in limitation of scalability of assembling and numerical integration process for larger matrices. To mitigate this issue, the author in [21] has developed an iterative strategy, which included partitioning of memory in a way to avoid overflow of GPU memory. Chunk of data was provided to each GPU and they did calculation independently. After completing their calculations, final results were accumulated on host memory.

In literature, lot of work has been done by focusing sparse matrix by vector multiplication like, Bolz et al. [8] published that matrix by vector product is most important part of FEM codes. Bell and Garland in [24], [25] also performed parallel sparse matrix by vector multiplication in parallel on many-core GPUs.

The studies on optimization of sparse matrix by vector multiplication on GPU first performed in 2003 by Bolz et al. [8]. Bell and Garland [24] introduced GPU based parallel implementations for various storage formats like CSR, CCS, COO, HYBrid and ELL etc. Baskaran and Bordawekar [26] proposed four various approaches for optimization of Sparse matrix vector multiplications for different GPUs.

- without synchronization parallelism
- optimization of mapping of thread on basis of connection with optimized memory access
- optimize global memory access
- data reuse

The algorithms for SpMV using the above mentioned sparse matrix storage formats were already implemented in NVIDIA’s library, Cusparse [27]. All optimization
approaches that were proposed recently were compared with implementations of Cusparse based SpMV. It was important to have basic knowledge of these simple storage approaches for understanding of recently proposed SpMV optimization approaches. Li and Saad [28] reviewed implementations of GPU based SpMV multiplication associated with iterative solvers in an efficient way and also discussed preprocessing approaches.

Reguly and Giles [20] elaborated COO, CSR and ELLpack etc. The Coordinate (COO) storage pattern was simplest one for construction point of view, as it kept record of row and column index of each non-zero entry of sparse matrix. For parallel multiplication on many core GPU, each thread performed multiplication of one non-zero entry of matrix by a vector entry. This format resulted in excessive global memory accesses by each thread for 3 different arrays in arbitrary fashion. The GPU based SpMV implementations using COO required a barrier to avoid conflicts while updating resultant vector in parallel. That’s why, COO was not used in practice, sometimes it was implemented as moderator between FEM assembly phase and construction of other storage formats that were required for iterative solvers [29]. The CSR storage pattern optimized overheads of global memory access by minimizing row vector access. Bell and Garland [24] proposed two CSR versions: scalar and vector CSR. First version utilized one thread for multiplication of each row of matrix by vector. While second version used wrap of threads to perform this operation. Scalar CSR results in non-coalesced memory access of vector, whereas vector CSR overcome this issue.

Margaris et al. [30] considered two approaches: row-wise and column-wise for distribution of workload among threads in parallel jacobi solver implementation. They concluded that row-wise distribution performed better than column-wise distribution. Majority of previous work have been focused on parallelizing Jacobi iterative solver on GPU for solution of dense linear system of equations. Zhang et al. [31] proposed single on-device algorithm. They launched 256 threads per block as shared memory is limited in size, so total 30 \times 256 = 7,680 threads can be launched at a time and made X shared in block. They considered 2 CPU intel Xeon X5482 that consists of 4 cores and 3.2 GHz speed, and GPU NVIDIA Tesla C1060 that contains 30 SMs for experimentation. They concluded that with this algorithm they achieved 59 times speedup for floating point and 19 times speed up for double precision computations than single threaded CPU.

Lin and Chen [34] proposed two hybrid algorithms and named them as primary and improved algorithm. The first algorithm was comprised of typical hybrid implementation of jacobi while second algorithm was proposed to optimize data movement from GPU to CPU in each iteration and achieved maximum speedup of factor 10.2 in later one. They concluded that when matrix size increases speedup also increases which shows that this particular algorithm has best scalability.

Torun et al. [35] proposed GPU based jacobi algorithm for eigen analysis problem. They implemented matrix by matrix multiplication instead of matrix by vector in jacobi solver and considered coalesced memory access for global memory transaction. They proposed four techniques: Traditional Access Method, Symmetric Access Method, Maximum Coalesced Access Method and One Step Parallel Jacobi (OSPJ) method on the basis of memory access patterns and concluded that OSPJ is best than other three and achieved good speedups. For calculation of reachability probability, in [36] sequential and parallel GPU based jacobi solver were compared. By this comparison it is found that Jacobi solver is best for sparse matrices and BiCGStab is best for denser ones.

Ahamed and Magoulès [37] proposed first GPU based algorithm of jacobi solver for solution of sparse linear system of equations. They performed experimentations on 3D problems like 3D laplace equation and 3D heat equation etc. and achieved 23 times speedups with NVIDIA GPU Tesla K20c.

III. GPU FINITE ELEMENT SOLVER

In most of the documented previous work, it is observed that they had focused:

- on solving dense matrix.
- with a linear consecutive data structures.
- only on mathematics of jacobi solver as they do not add algorithms that consider on-device synchronization and communication methods.

In this work, sparse linear system is considered while the current data structure is not compatible. A is converted into CSR where three vectors one for row pointer, column id and values, as shown in Figure 6. We have proposed multiple implementations of GPU based jacobi solver by considering on-device communication and synchronization methods and compared their results.

IV. GPGPU BASED PARALLEL JACOBI SOLVER FORMULATION

In Jacobi Iterative Solver, the computation of solution for linear system of equation is an embarrassingly parallel part. Therefore, GPU is used to offload this stage of FES, while remaining stages are executed on shared memory architecture in parallel. The main objective is to compute solution in minimum amount of time that reduces overall application time.
Figure 7 shows the basic flow of program in CUDA programming model. The key points of mapping sequential algorithm to an algorithm suitable to run on GPU are:

- Allocation of memory on host.
- Allocation of same amount of memory on GPU by built-in method CudaMalloc.
- Initialization of data in memory allocated in CPU.
- Data has to be copied from host to device using built-in function CudaMemCpy() with parameter CudaMemCpyHostToDevice to specify direction for copy.
- Host thread invokes kernel, which transfer control from CPU to GPU and also specify grid size and block size required for program.
- Data has to be copied back from device to host using built-in method CudaMemCpy() with parameter to specify direction of copy CudaMemCpyDeviceToHost.
- The device memory has to be released using built-in method CudaFree().

Writing programs in CUDA and build up environment is relatively simple task. But it requires deep knowledge of architecture details and knowledge of writing codes in parallel. The key part of CUDA programming is kernel calls, where programmers must determine data parallelism required by code.

First, we must decide how workload has to be distributed among blocks of threads. In the beginning, we opted block-partition for distribution of tasks among threads but it results in performance loss. As the memory accesses to global memory must be coalesced and to shared memory must be conflict free. Otherwise, the performance of memory will bounds kernel. Therefore, cyclic partition is preferred over block in GPU as it gives coalesced memory access. Threads with consecutive thread ids access contiguous memory locations as can be seen in Figure 8. The partitioning of given workload into an appropriate number of threads is the point which defines successful parallel program.

Second, we must decide how to use hardware resources available to us which gives better performance. There are three levels of memory in latest GPUs: local, shared and global detail of which is discussed above. We optimized shared memory accesses due to its limited size and bank conflicts. We preferred to store and access data from global memory as it is accessible to all threads and has large size like 6 GB in GTX 1060 and 8 GB in Quadro P4000.

Example of basic CUDA program is shown in Figure 8. The HybridThree() is the entry point of kernel, a function written in CUDA C represented by __global__ identifier. The HybridThree <<< BPG, TPB >>> () is kernel invocation by host, which will enumerates BPG number of blocks with TPB threads each. These values can be set for each kernel call. An explicit barrier is required between kernel launches, to guarantee that all threads of first kernel have finish their work before second kernel launch. When kernel is
### Table 1. Comparison of Hybrid Solvers on the basis of On-device communication and Synchronization method.

|             | Block-level | Grid-level |
|-------------|-------------|------------|
| **GPU Solvers** | **Communication method** | **Synchronization method** | **Communication method** | **Synchronization method** |
| Solver one  | Butterfly   | synthreads() | AtomicAdd() | Grid.sync() |
| Solver two  | √           | √           | √           | √           |
| Solver three| √           | √           | √           | √           |
| Solver four | √           | √           | √           | √           |
| Solver five | √           | √           | √           | √           |
| Solver six  | √           | √           | √           | √           |

#### Algorithm 1 Sparse Matrix Vector Multiplication

```
1: procedure SpMV(rows, col, val, x) ↷ CSR format
2:    id ← threadIdx.x + blockIdx.x * blockDim.x
3:    for i ← id, DOFs do
4:        for j ← row[i], row[i + 1] do
5:            c[i] ← val[j] * x[col[j]] + c[i]
6:        end for
7:    i ← blockDim.x * gridDim.x + i
8:    return c
9: end procedure
```

#### Algorithm 2 Hybrid Solver One

```
1: /* — Kernel Definition for Device — */
2: procedure HybridOne(A, b, c)
3:    tid ← threadIdx.x ↷ local thread id
4:    id ← threadIdx.x + blockIdx.x * blockDim.x
5:    for i < id, DOFs do
6:        r ← b[i] - c[i]
7:        x[i] ← A[i][i]⁻¹ * r + x[i]
8:        norm_R[i] ← r * r + norm_R[i]
9:        norm_X[i] ← x[i] * x[i] + norm_X[i]
10:       i ← blockDim.x * gridDim.x + i
11: end for

12: /* — Second kernel invocation — */
13: procedure DeviceBarrier()
14: CudaMemCpy(A_nrmRes[], gNrmDist[]) ↷ device to host
15:   for i ← 0, DOFs; i + + do
16:       norm_res ← gNrmRes[i] + norm_res
17:       norm_dest ← gNrmDest[i] + norm_dest
18:   end for
19:   error ← √ norm_res / norm_dest
20: end while
```

launched on GPU, the hardware scheduler enumerates blocks of threads and maps them to available multiprocessors, and continues to do it until required resources are available.

While implementing Jacobi solver in parallel on CPU there were two problems needs to be addressed. The data validity issue of solution vector and the race condition may occur while calculating normals of residual and solution vector. The major bottleneck in GPU based implementation is reduction of norms of residual and solution vector. As on GPU block-level as well as grid level communication methods are needed.

### A. HYBRID SOLVERS

Table 1 shows synchronization and communication characteristics of hybrid implementations of Jacobi solver. The details of these implementations are provided in later sections. There are two algorithms used to distribute number of rows among the blocks and among the grid.

In all algorithms device code, host code, CUDA provided methods and synchronization points are written in blue, black, green and red color respectively.

1) SOLVER ONE

The Hybrid Solver one is proposed for device of compute capability 2.1 or more which has compatibility with CUDA v 6.5. This CUDA version was lacked of any on-device global synchronization method. This solver is proposed without any synchronization and communication method, as can be seen in first row of Table 1. In each iteration, where a synchronization point was required, the computations were switched from device to host.

- Figure 9 demonstrates execution flow of solver on CPU as well as on GPU, detail of which is given below.
- **Data validity issue:** We proposed two separate kernels, first for matrix by vector multiplication as can be seen in Algorithm 1 and second for computation of residual vector, solution vector and two vectors of size DOFs for norms of residual and solution vector as shown in Algorithm 2.
- **Race condition issue:** To avoid race condition which can occur while computing scalar values of residual and
solution vector, a global communication method was needed. As this solver has no support of global communication method. Thus, computations were switched from device to host with vectors of size total number of unknowns, which are named as DOFs.

- **Device synchronization issue**: The device synchronization points are needed after each kernel invocation to guaranteed that all threads have finished their work before another kernel launch. These synchronization points are represented in brick red color in Figure 9 as well as on line 17 and 20 of Algorithm 2.

- The main issue in this particular solver is large number of memory copying from device to host in each iteration.

2) SOLVER TWO

The Hybrid Solver 2 has optimized memory copying issue which is discussed above. The approach opted in this solver has optimized number of memory copying from DOFs to Total blocks count as can be seen in Algorithm 3 and 4.

- Second row of Table 1 shows that block-level communication and synchronization methods are opted in this particular solver.

- Figure 10 demonstrates execution flow of solver on CPU as well as on GPU, detail of which is given below.

- **Data validity issue**: is resolved as discussed in Section IV-A1.

- **Race condition issue**:
  - Block-level
  - *Reduction*: As shown in second row of Table 1, butterfly communication method is opted for block-level thread safe reduction.

**Algorithm 3 Hybrid Solver 2**

1: /* — Kernel Definition for Device — */
2: procedure HybridTwo(A, b, c)
3:   tid ← threadIdx.x
4:   id ← threadIdx.x + blockIdx.x * blockDim.x
5:   Shared sNR[blockDim], sND[blockDim]
6:   while i < id, DOFs do
7:     r ← b[i] − c[i]
8:     x[i] ← A[i][i]−1 * r + x[i]
9:     sNR[tid] ← r * r + sNR[tid]
10:    sND[tid] ← x[i] * x[i] + sND[tid]
11:    i ← blockDim.x * gridDim.x + i
12:  end while
13:  syncThreads() ➤ Block Level Synchronization
14:  for s ← 1, blockDim.x do ➤ Butterfly Comm
15:    if ((tid%(s*2)==0) && (tid+s<blockDim.x)) then
16:      sNR[tid] ← sNR[tid + s] + sNR[tid]
17:      sND[tid] ← sND[tid + s] + sND[tid]
18:    end if
19:  syncThreads()
20:  s ← s * 2
21: end for
22:  if tid == 0 then
23:    gNrmRes[blockIdx.x] ← sNR[tid]
24:    gNrmDist[blockIdx.x] ← sND[tid]
25:  end if
26: end procedure
27: /* — Jacobi Solver for Host — */
28: CudaMemCpy(A, b, c)
29: while (k < total iteration) & (error > 10E−6) do
30:  SpMV <<< BPG, TPB >>> (js)
31:  DeviceBarrier()
32:  /* — Second kernel invocation — */
33:  HybridTwo <<< BPG, TPB >>> (js)
34:  DeviceBarrier()
35:  CudaMemCpy(gNrmRes, gNrmDist)
36:  for i ← 0, TotBlks do
37:    normRes ← gNrmRes[i] + normRes
38:    normDist ← gNrmDist[i] + normDist
39:  end for
40:  error ← \sqrt{\frac{normRes}{normDist}}
41: end while

The implementation detail of butterfly communication method can be seen on page number 107 of [38].

* **Synchronization**: After block-level reduction, syncthread() method is used for block-level synchronization point, where all threads of each block have to wait, to guaranteed that all blocks have updated shared value. These synchronization points can be seen in Figure 10 and on line 13 and 19 of Algorithm 3 in brick red color.
Now, thread 0 of each block has to populate their reduced value at particular block ids in global vector.

- **Grid-level**
  - **Reduction**: To avoid race condition which can occur while computing scalar values of residual and solution vector, a global communication method was needed.
    As this solver has no support of global communication method, thus, computations were switched from device to host with vectors of size total number of blocks.

  - Finally, error is calculated for decision of next iteration.
  - **Device synchronization**: is done as discussed in Section IV-A1.

3) **SOLVER THREE**
The Hybrid Solver 3 is proposed for device of compute capability 6.1 or more as well as for CUDA v9.1 on windows. As NVIDIA introduced block-level communication method in this CUDA version. The basic implementation pattern of Hybrid Solver 2 and 3 is similar, the only difference is in calculation of reduced values of normals of residual and solution vector.

  - Third row of Table 1 comprises implementation details of solver three, which shows that both block-level communication and synchronization methods are opted in this particular solver.
  - Figure 11 demonstrates execution flow of solver on CPU as well as on GPU, detail of which is given below.

4) **SOLVER FOUR**
The basic implementation pattern of Hybrid Solver 2 and 4 is similar, as in both Jacobi solver is implemented by invoking...
two separate kernels as discussed above and shown in Algorithm 5. The only difference is in computation of scalar values of norms of residual and solution vector.

- Fourth row of Table 1 comprises implementation details of solver four, which shows that grid-level communication method as well as block-level communication and synchronization methods are opted in this particular solver.
- Figure 12 demonstrates execution flow of solver on CPU as well as on GPU.
- Data validity issue: is resolved as discussed in Section IV-A1.
- Race condition issue:
  - Block-level
    * Reduction: Butterfly communication method

\[ \text{Algorithm 4 Hybrid Solver 3} \]

1: /* — Kernel Definition for Device — */
2: procedure HybridThree(A, b, c)
3:   tid ← threadIdx.x >> local thread id
4:   id ← threadIdx.x + blockDim.x * blockIdx.x
5:   Shared blkRes ← blkDist ← 0
6:   res ← dist ← 0
7:   while i < id, DOFs do
8:     r ← blockIdx.x - c[i]
9:     x[i] ← A[i][i]^{-1} * r + x[i]
10:    res ← r * r + res
11:   end do
12: end while
13: atomicAdd(blkRes, res)
14: atomicAdd(blkDist, dist)
15: syncThreads() >> Block Level Synchronization
16: while k < total_iteration & (error > 10\(E\) - 6) do
17:   SpMV <<< BPG, TPB >>> (js)
18:   DeviceBarrier()
19:   /* — Second kernel invocation — */
20:   HybridThree <<< BPG, TPB >>> (js)
21:   DeviceBarrier()
22:   CudaMemCpy(gNrmRes, gNrmDist)
23:   for i ← 0, TotalBlks do
24:     normRes ← gNrmRes[i] + normRes
25:     normDist ← gNrmDist[i] + normDist
26:   end for
27:   error ← \sqrt{\frac{\text{normRes}}{\text{normDist}}}
28: end while

\[ \text{Algorithm 5 Hybrid Solver 4} \]

1: /* — Kernel Definition for Device — */
2: procedure HybridFour(A, b, c)
3:   tid ← threadIdx.x >> local thread id
4:   id ← threadIdx.x + blockDim.x * blockIdx.x
5:   Shared blkRes[blockDim.X], blkDist[blockDim.X]
6:   while i < id, DOFs do
7:     r ← blockIdx.x - c[i]
8:     x[i] ← A[i][i]^{-1} * r + x[i]
9:     blkRes[tid] ← r * r + blkRes[tid]
10:    blkDist[tid] ← x[i] * x[i] + blkDist[tid]
11:   i ← blockDim.x * gridDim.x + i
12: end while
13: syncThreads() >> Block Level Synchronization
14: for s ← 1, blockDim.x do >> Butterfly Comm
15:   if ((tid%\(s*2\))==0)&&(tid+s<blockDim.x) then
16:     blkRes[tid] ← blkRes[tid] + s + blkRes[tid]
17:     blkDist[tid] ← blkDist[tid] + s + blkDist[tid]
18:   end if
19:   s ← s + 2
20: syncThreads()
21: end for
22: if tid == 0 then
23:   atomicAdd(gNrmRes, blkRes[tid])
24:   atomicAdd(gNrmDist, blkDist[tid])
25: end if
26: end procedure
27: /* — Jacobi Solver for Host — */
28: CudaMemCpy(A, b, c)
29: while k < total_iteration & (error > 10\(E\) - 6) do
30:   SpMV <<< BPG, TPB >>> (js)
31:   DeviceBarrier()
32:   /* — Second kernel invocation — */
33:   HybridFour <<< BPG, TPB >>> (js)
34:   DeviceBarrier()
35:   CudaMemCpy(gNrmRes, gNrmDist)
36:   for i ← 0, TotalBlks do
37:     normRes ← gNrmRes[i] + normRes
38:     normDist ← gNrmDist[i] + normDist
39:   end for
40:   error ← \sqrt{\frac{\text{normRes}}{\text{normDist}}}
41: end while

* Synchronization: syncthreads() method is used for block-level synchronization, as discussed in IV-A2. These synchronization points can be seen in Figure 12 and on line 13 and 20 of Algorithm 5 in brick red color.

* Grid-level

* Reduction: CUDA provided method atomic_add() is used by thread 0 of each block, which can be seen on line 23 and 24 in green color.
In this way, scalar values of norms of residual and solution vector are computed.

* **Synchronization:** After grid level reduction, a global synchronization point is needed where all blocks of grid have to wait, to guarantee that all blocks have updated global value. As the current CUDA version has no such synchronization point, that’s why scalar values are copied back to host. Then, square root of scalar values of norms of residual and solution vector is taken by host thread for calculation of error and for decision of next iteration.

  - **Device synchronization:** is done as discussed in Section IV-A1.

### 5) SOLVER FIVE

The basic implementation pattern of Hybrid Solver 3 and 5 is similar, as in both Jacobi solver is implemented by invoking two separate kernels as discussed above and shown in Algorithm 6. The only difference is in computation of scalar values of norms of residual and solution vector.

  - Fifth row of Table 1 comprises implementation details of solver five, which shows that grid-level communication method as well as block-level communication and synchronization methods are opted in this particular solver.
  - Figure 13 demonstrates execution flow of solver on CPU as well as on GPU.
  - **Data validity issue:** is resolved as discussed in Section IV-A1.
  - **Race condition issue:**
    - **Block-level**
      * **Reduction:** CUDA provided `atomicAddBlock()` is used for block-level reduction of norms of residual and solution vector, which can be seen on line 14 and 15 of Algorithm 6 in green color.
      * **Synchronization:** `syncthread()` method is used for block-level synchronization, as discussed in IV-A2. These synchronization points can be seen in Figure 13 and on line 16 in brick red color.
    - **Grid-level**
      * **Reduction:** CUDA provided method `atomic_add()` is used by thread 0 of each block.
      In this way, scalar values of norms of residual and solution vector are computed.
      * **Synchronization:** as discussed in Section IV-A4 current CUDA version has no grid-level synchronization point, that’s why scalar values are copied back to host. Then, square root of scalar values of norms of residual and solution vector is taken by host thread for calculation of error and for decision of next iteration.

  - **Device synchronization:** is done as discussed in Section IV-A1.

### 6) SOLVER SIX

The Hybrid Solver 6 is also proposed for device of compute capability 6.1 or more and for CUDA v9.1 or more.

  - Sixth row of Table 1 comprises implementation details of solver six, which shows that only grid-level communication method is opted in this particular solver.
Algorithm 6 Hybrid Solver 5

```plaintext
1: /* — Kernel Definition for Device — */
2: procedure HybridFive(A, b, c)
3:    tid ← threadIdx.x  // local thread id
4:    id ← threadIdx.x + blockIdx.x * blockDim.x
5:    Shared blkRes ← blkDist ← 0
6:    res ← dist ← 0
7:    while i < id, DOFs do
8:       r ← b[i] − c[i]
9:       x[i] ← A[i][i]−1 * r + x[i]
10:      res ← r * r + res
11:      dist ← x[i] * x[i] + dist
12:      i ← blockDim.x * gridDim.x + i
13: end while
14: atomicAdd(blkRes, res)
15: atomicAdd(blkDist, dist)
16: syncThreads()  // Block Level Synchronization
17: if tid == 0 then
18:    atomicAdd(normRes, blkRes)
19:    atomicAdd(normDist, blkDist)
20: end if
end procedure
```

Figure 14 demonstrates execution flow of solver on CPU as well as on GPU.

- **Data validity issue:** is resolved as discussed in Section IV-A1.
- **Race condition issue:**
  - **Reduction:** CUDA built-in method atomicAdd() is used. Threads of all blocks add their local values directly into global scoped variable as shown on line 13 and 14 of Algorithm 7. Thus, there is no need of block-level thread safe reduction methods and synchronization points. In this way, scalar values of norms of residual and solution vector are computed.
  - **Synchronization:** current CUDA version has no grid-level synchronization point as discussed in Section IV-A4, that’s why scalar values are copied back to host.

Algorithm 7 Hybrid Solver 6

```plaintext
1: /* — Kernel Definition for Device — */
2: procedure HybridSix(A, b, c)
3:    tid ← threadIdx.x  // local thread id
4:    id ← threadIdx.x + blockIdx.x * blockDim.x
5:    res ← dist ← 0
6:    while i < id, DOFs do
7:       r ← b[i] − c[i]
8:       x[i] ← A[i][i]−1 * r + x[i]
9:       res ← r * r + res
10:      dist ← x[i] * x[i] + dist
11:      i ← blockDim.x * gridDim.x + i
12: end while
13: atomicAdd(normRes, res)
14: atomicAdd(normDist, dist)
15: end procedure
```

- **Race condition issue:**
  - **Reduction:** CUDA built-in method atomicAdd() is used. Threads of all blocks add their local values directly into global scoped variable as shown on line 13 and 14 of Algorithm 7. Thus, there is no need of block-level thread safe reduction methods and synchronization points. In this way, scalar values of norms of residual and solution vector are computed.
  - **Synchronization:** current CUDA version has no grid-level synchronization point as discussed in Section IV-A4, that’s why scalar values are copied back to host.

Then, square root of scalar values of norms of residual and solution vector is taken by host thread for calculation of error and for decision of next iteration.
• **Device synchronization**: is done as discussed in Section IV-A1.

**B. FULL GPU SOLVERS**

The Hybrid versions are proposed for devices which are lack of cudaLaunchCooperativeKernel support. As again and again switching from device to host and vice versa is not good practice. Thus, the implementation of single Jacobi Iterative solver method on device required grid-level synchronization points. We proposed global lock-based synchronization points as discussed in [39], [40]. In lock-based synchronization point global variable mutex is used to count number of blocks that reach the synchronization point, when block finish its computation one of its thread will add 1 to mutex through CUDA provided atomicADD(). Then all threads will check value of mutex with goal value, if mutex is equal to goal value then all threads can continue further computations. We found after performing several experiments with variety of test data, that it is not efficiently works for large meshes.

Therefore, we decided to opt CUDA provided global synchronization points which are available only when cudaLaunchCooperativeKernel support is enabled. In windows, the default working mode is WDDM, TCC mode is required for enabled cudaLaunchCooperativeKernel support. In ubuntu, the working mode of device is TCC, which enabled cudaLaunchCooperativeKernel support.

It is necessary for using runtime cudaLaunchCooperativeKernel API to ensure the co-residency of blocks of threads on GPU, grid-size needs to be consider carefully.

Multiple Full GPU versions are proposed for devices which has support of cudaLaunchCooperativeKernel. This support enables the implementation of Jacobi solver in a way that once CPU thread invokes kernel it will exits from device only when convergence criterion is satisfied.

Table 2 shows implementation detail of Full GPU Solvers by checking those columns that contain synchronization and communication methods, which are opted by particular solver.

1) **SOLVER SEVEN**

The Solver seven is modification of solver version 4. The only difference is instead of invoking two separate kernels, it invokes single kernel, which is cudaLaunchCooperativeKernel.

- First row of Table 2 comprises implementation details of solver seven, which shows that grid-level and block-level communication as well as synchronization methods are opted in this particular solver.
- Figure 15 demonstrates execution flow of solver on CPU as well as on GPU.
- **Data validity issue**: To solve issue of data validity, instead of separate kernel invocation, this particular solver has used temporary global scoped vector to store updated values of solution vector. At the end of each iteration the solver has copied temporary vector into solution vector which will be used in matrix by vector multiplication of next iteration to ensure validity of solution vector in each iteration.

- **Race condition issue**:
  - **Block-level**
    * **Reduction**: *Butterfly communication method*, which can be seen on line 19 to 26 of Algorithm 8.
    * **Synchronization**: syncthread() method is used for block-level synchronization, as discussed in IV-A2. These synchronization points can be seen in Figure 14 and on line 18 and 24 in brick red color.
  - **Grid-level**
    * **Reduction**: CUDA provided method *atomicAdd*() is used by thread 0 of each block.
      In this way, scalar values of norms of residual and solution vector are computed.
    * **Synchronization**: After grid level reduction, a global synchronization point is needed, where all blocks of grid have to wait, to guarantee that all blocks have updated global value. As the current CUDA version has provided grid-level synchronization point in cudaLaunchCooperativeKernel API as on line 31 of Algorithm 8.

Then, square root of scalar values of norms of residual and solution vector is taken and error is calculated on device for decision of next iteration.
TABLE 2. Comparison of Full GPU Solvers on the basis of On-device communication and Synchronization method.

| Block-level         | Grid-level          |
|---------------------|---------------------|
| Communication method| Synchronization method | Communication method | Synchronization method |
| GPU Solvers         |                     |                      |
| Butterfly           | AtomicAdd()         | syncthreads()        | AtomicAdd()            | Grid sync() |
| Solver seven        | ✓                    | ✓                     | ✓                      | ✓          |
| Solver eight        | ✓                    | ✓                     | ✓                      | ✓          |
| Solver nine         | ✓                    | ✓                     | ✓                      | ✓          |

Algorithm 8 Solver 7

1: /* — Kernel Definition for Device — */
2: procedure FullGpuOne(A, b, c)
3: tid ← threadIdx.x ▷ local thread id
4: while (iteration < 10E6) && (error > 10E−6) do
5:  norm_res ← norm_dest ← 0
6:  Shared blkRes[blockDim.x], blkDist[blockDim.x]
7:  for i ← id.DOFs do
8:      for j ← row[i], row[i+1] do
9:         c[i] ← val[j] * x[col[j]] + c[i]
10:     j ← j + 1
11:     end for
12:     r ← b[i] − c[i]
13:     x[i] ← A[i][i]−1 * r + x[i]
14:     blkRes[tid] ← blkRes[tid] + r * r
15:     blkDist[tid] ← blkDist[tid] + x[i] * x[i]
16:     i ← blockDim.x * gridDim.x + i
17:     end for
18:     syncThreads()
19:     for s ← 1, blockDim.x do ▷ Butterfly comm
20:         if (tid%s*2==0) && (tid+s<blockDim.x)) then
21:             blkRes[tid]+ = blkRes[tid + s]
22:             blkDist[tid]+ = blkDist[tid + s]
23:         end if
24:     end for
25:     s ← s * 2
26:     syncThreads()
27:     if tid == 0 then
28:         atomicAdd(normRes, blkRes[tid])
29:         atomicAdd(normDes, blkDist[tid])
30:     end if
31:     syncBlocks()
32:     error ← √(normRes/normDist)
33: end while
34: end procedure

2) SOLVER EIGHT
The Solver eight is modification of Hybrid version five, the only difference is instead of invoking two separate kernels with continuous switching between host and device it invokes single kernel, which is cudaLaunchCooperativeKernel.

- Second row of Table 2 comprises implementation details of solver eight, which shows that grid-level and block-level communication as well as synchronization methods are opted in this particular solver.
  - Figure 16 demonstrates execution flow of solver on CPU as well as on GPU.
  - Data validity issue: is resolved as discussed in Section IV-B1.
  - Race condition issue:
    - Block-level
      - Reduction: CUDA provided AtomicAddBlock() is implemented for reduction of normals of residual and solution vector which can be seen on line 18 and 19 of Algorithm 9 in green colored text.
      - Synchronization: syncthread() method is used for block-level synchronization, as discussed in IV-A2. These synchronization points can be seen in Figure 16 and on line 20 in brick red color.
    - Grid-level
      - Reduction: CUDA provided method atomic__Add() is used by thread 0 of each block. In this way, scalar values of norms of residual and solution vector are computed.
The solver nine is a modification of Hybrid version 6, the only difference is instead of invoking two separate kernels it invokes single kernel, which is cudaLaunchCooperativeKernel.

- Third row of Table 2 comprises implementation details of solver nine, which shows that only grid-level communication as well as synchronization methods are opted in this particular solver.
- Figure 17 demonstrates execution flow of solver on CPU as well as on GPU.
- Data validity issue: is resolved as discussed in Section IV-B1.
- Race condition issue:
  - Grid-level
    * Reduction: CUDA built-in method atomicAdd() is used. Threads of all blocks add their local values directly into global scoped variable as shown on line 17 and 18 of Algorithm 10.

### TABLE 3. Comparison of Speedups of Solvers with block partition on the basis of On-device Communication and Synchronization method with sq6, sq7, sq8 and sq9.

| GPU Solvers | Third (sq6) | Fourth (sq7) | Fifth (sq8) | Sixth (sq9) | Third (sq6) | Fourth (sq7) | Fifth (sq8) | Sixth (sq9) |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| Solver 1    | 13.34       | 12.99       | 8.11        | 6.03        | 9.35        | 14.84       | 11.04       | 7.89        |
| Solver 2    | 25.82       | 14.30       | 8.33        | 6.04        | 22.99       | 19.96       | 11.66       | 7.98        |
| Solver 3    | 2.87        | 6.90        | 7.41        | 5.97        | 1.89        | 5.59        | 8.90        | 7.75        |
| Solver 4    | 25.80       | 14.35       | 8.23        | 6.04        | 22.58       | 19.99       | 11.59       | 7.94        |
| Solver 5    | 2.76        | 6.87        | 7.42        | 5.98        | 1.89        | 5.62        | 8.98        | 7.71        |
| Solver 6    | 19.27       | 13.98       | 8.27        | 6.04        | 12.74       | 17.54       | 11.54       | 7.96        |
| Solver 7    | 26.47       | 13.64       | 6.29        | 4.44        | 27.05       | 18.56       | 9.17        | 5.99        |
| Solver 8    | 3.08        | 7.86        | 6.01        | 4.37        | 1.99        | 6.35        | 7.88        | 5.88        |
| Solver 9    | 25.33       | 13.99       | 6.52        | 6.04        | 18.17       | 18.34       | 9.22        | 5.95        |

### Algorithm 9 Solver 8

```c
1: /* — Kernel Definition for Device — */
2: procedure FullGpuTwo(A, b, c)
3: while (iteration < 10E6) & &(error > 10E - 6) do
4:    global normRes ← norm_dest ← 0
5:    Shared blkRes ← blkDest ← 0
6:    local res ← dest ← 0
7:    for i ← id.DOFS do
8:        for j ← row[i], row[i + 1] do
9:            c[i] ← val[j] * x[col[j]] + c[i]
10:           j ← j + 1
11:       end for
12:       r ← b[i] - c[i]
13:       x[i] ← A[i][i]^{-1} * r + x[i]
14:       res ← res + r * r
15:       dest ← dest + x[i] * x[i]
16:       i ← blockDim.x * gridDim.x + i
17:     end for
18:    atomicAdd(blkRes, res)
19:    atomicAdd(blkDest, dest)
20:    syncThreads()
21: if tid == 0 then
22:        atomicAdd(norm_res, blkRes)
23:        atomicAdd(norm_dest, blkDest)
24: end if
25: syncBlocks()
26: error ← √norm
27: end while
end procedure
```

### Algorithm 10 Solver 9

```c
1: /* — Kernel Definition for Device — */
2: procedure FullGpuThree(A, b, c)
3: while (iteration < 10E6) & &(error > 10E - 6) do
4:    global normRes ← normDist ← 0
5:    res ← dest ← 0
6:    for i ← id.DOFs do
7:        for j ← row[i], row[i + 1] do
8:            c[i] ← val[j] * x[col[j]] + c[i]
9:            j ← j + 1
10:       end for
11:       r ← b[i] - c[i]
12:       x[i] ← A[i][i]^{-1} * r + x[i]
13:       res ← res + r * r
14:       dist ← dist + x[i] * x[i]
15:       i ← blockDim.x * gridDim.x + i
16:     end for
17:    atomicAdd(normRes, res)
18:    atomicAdd(normDest, dist)
19:    syncBlocks()
20: error ← √normRes / normDist
21: end while
end procedure
```

### Synchronization:
After grid level reduction, a global synchronization point is needed as discussed in Section IV-B1. This grid-level barrier can be seen on line 25 of Algorithm 9. Then, square root of scalar values of norms of residual and solution vector is taken and error is calculated on device for decision of next iteration.

3) SOLVER NINE

The solver nine is modification of Hybrid version 6, the only difference is instead of invoking two separate kernels it...
TABLE 4. Overview of meshes and hardware setup used for performance evaluation.

| Mesh | Nodes | Elements | NNZ | Processor | Core i7 8750H | GTX 1060 | Quadro P4000 |
|------|-------|----------|-----|-----------|---------------|----------|--------------|
| sq6  | 33025 | 66052    | 196255 | -         | 14            | 10       | -            |
| sq7  | 131525| 263172   | 785887 | No. of cores | 6            | 1280     | 1792         |
| sq8  | 525313| 1050628  | 3144639 | Processor speed (GHz) | 4.10 | 1.7 | 1.5 |
| sq9  | 2099201| 4198404 | 12581503 | Memory (GB) | 16 | 6 | 8 |

TABLE 5. Comparison of Speedups of Solvers with cyclic partition on the basis of On-device Communication and Synchronization method with sq6, sq7, sq8 and sq9.

| GPU Solvers | Third (sq6) | Fourth (sq7) | Fifth (sq8) | Sixth (sq9) | Third (sq6) | Fourth (sq7) | Fifth (sq8) | Sixth (sq9) |
|-------------|-------------|--------------|-------------|-------------|-------------|--------------|-------------|-------------|
| Solver 1    | 18.20       | 33.99        | 42.33       | 44.79       | 10.00       | 28.45        | 46.44       | 56.23       |
| Solver 2    | 35.55       | 45.54        | 45.62       | 45.63       | 27.82       | 48.47        | 57.44       | 59.64       |
| Solver 3    | 2.81        | 9.17         | 22.87       | 37.00       | 1.78        | 6.20         | 18.99       | 39.83       |
| Solver 4    | 33.83       | 44.14        | 45.34       | 45.59       | 26.78       | 48.52        | 57.27       | 59.63       |
| Solver 5    | 2.78        | 9.02         | 22.08       | 36.09       | 1.78        | 6.33         | 19.25       | 39.88       |
| Solver 6    | 22.79       | 37.29        | 43.43       | 45.19       | 13.56       | 33.23        | 50.54       | 57.98       |
| Solver 7    | 43.87       | 41.49        | 38.71       | 36.79       | 40.67       | 48.44        | 48.65       | 47.56       |
| Solver 8    | 2.91        | 9.42         | 21.91       | 32.13       | 1.87        | 6.61         | 18.96       | 36.19       |
| Solver 9    | 33.98       | 44.11        | 46.44       | 45.38       | 18.13       | 40.38        | 34.09       | 36.29       |

FIGURE 17. Flow diagram of Solver nine.

Thus, there is no need of block-level thread safe reduction methods and synchronization points. In this way, scalar values of norms of residual and solution vector are computed.

* Synchronization: After grid level reduction, a global synchronization point is needed, as discussed in Section IV-B1. This grid-level barrier is on line 19 of Algorithm 10.

Then, square root of scalar values of norms of residual and solution vector is taken and error is calculated on device for decision of next iteration.

V. EXPERIMENTS AND RESULTS

Experiments were performed for evaluation of performance by using data set sq6 to sq9, detail of which can be seen on left side of Table 4. The sequential timing was taken on CPU Intel Core-i7 8750h and timing of parallel implementation was taken on two NVIDIA GPUs, hardware detail of which is given on right side of Table 4. Red color is used to highlight best speedup obtained for each mesh using different Jacobi implementations in case of GTX 1060 as well as in Quadro P4000 in Table 3 and 5. Table 3 demonstrates the speedup of on-device Jacobi solvers with multiple meshes on GPUs. The block workload distribution approach was used for larger meshes. These results showed that block partition was not suitable on GPU architecture, the main reason of poor speedups was non-coalesced memory accesses while fetching data from global memory space. For this reason we preferred to use cyclic partition over block in GPU implementation.

Table 5 demonstrates the speedup of on-device Jacobi solvers with multiple meshes on both GPUs. The cyclic workload distribution approach is used for larger meshes.

Figure 18 demonstrates the speedup of all solvers for meshes sq6, sq7, sq8 and sq9 on two GPUs named as GeForce GTX 1060 and Quadro P4000. The approach for workload distribution in all solvers is block partitioning. As we can see these solvers perform poor with larger meshes as compared to small. The overall speedup of solver 2 and 4 is equal and better as compared to others.

Figure 19 demonstrates the speedups of all solvers for meshes sq6, sq7 and sq8 on two GPUs named as GeForce GTX 1060 and Quadro P4000. The approach for workload distribution in all solvers is Thread-level parallelism i.e number of threads are equal to total DOFs. The overall speedups of solver 2 and 4 are better as compared to others. On the
contrary, solver 3 and 5 did not perform good as these implemented block-level CUDA provided communication method which results in performance loss.

A. CONCLUSION
The study of FEM for shared memory has identified that Jacobi iterative solver is most time consuming part of software. Thus, we offload this expansive part on GPGPU, as it is well suited for massively parallel codes. We implemented multiple designs for GPU based Jacobi iterative solver to meet synchronization and communication requirements. For this purpose, we proposed nine solvers that varies from each other on the basis of opted synchronization and communication approaches.
The synchronization and communication approaches were opted for block and grid-level thread-safe reduction. The comparison of grid and block-level synchronization and communication methods opted by solvers can be seen in Table 2. After comparing each solver on the basis of speedup we came up at the point that Solver 2 and 4 were best as these achieved speedup of 46 on GTX 1060 and 60 on Quadro P4000 as compared to others.

We have compared CUDA synchronization methods with our CUDA implementation of Butterfly synchronization method. Where solvers 2, 4 and 7 were implemented using Butterfly algorithm, while solvers 3, 5 and 8 were implemented using CUDA provided synchronization method like atomicAdd_block, atomicAdd and grid synchronization. All the experiments had shown that our Butterfly synchronization technique had outperformed CUDA provided synchronization method as can be seen in Table 5.

Even though, solver 2 and 4 were differ in terms of grid-level synchronization methods. As in solver 2, where grid-level synchronization was required we switched from device with vector of size blocks and did remaining computation on host to guaranteed thread-safe reduction. On the contrary, solver 4 had implemented on-device grid-level synchronization methods for thread safe reduction on device and switched from device with only two scalar values in each iteration. We observed that this implementation difference had no significant impact on performance of these solvers.

B. FUTURE WORK
The above mentioned work has more dimensions like study of dynamic thread safe sparse matrix data structures. During this work, due to lack of time we have not considered global memory access latencies and texture memories etc. The current work has to be studied for more dimensional meshes, as the dimension of mesh and the type of problem affects the structure of sparse matrix. Furthermore, a work need to be done for advance iterative solvers such as conjugate gradient. In this implementation, the solver part is implemented on GPUs but it will be better if whole application can be ported on many-core GPU.

GPUs are growing very rapidly, as new GPUs are being introduced every few months or so. The physical design of these GPUs are changing, so in future work more advanced GPUs will be considered.

REFERENCES
[1] M. Czapinski, C. Thompson, and S. Barnes, “Reducing communication overhead in multi-GPU hybrid solver for 2D Laplace’s equation,” Int. J. Parallel Program., vol. 42, no. 6, pp. 1032–1047, Dec. 2014.
[2] C. Cecka, A. J. Lew, and E. Darve, “Assembly of finite element methods on graphics processors,” Int. J. Numer. Meth. Engng, vol. 85, no. 5, pp. 640–669, Feb. 2011.
[3] C. Geuzaine and J.-F. Remacle, “Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities,” Int. J. Numer. Methods Engng, vol. 79, no. 11, pp. 1309–1331, Sep. 2009.
[4] A. Kakay, E. Westphal, and R. Hertel, “Speedup of FEM micromagnetic simulations with graphical processing units,” IEEE Trans. Magn., vol. 46, no. 6, pp. 2303–2306, Jun. 2010.
[5] Y. Liu, S. Jiao, W. Wu, and S. De, “GPU accelerated fast FEM deformation simulation,” in Proc. IEEE Asia Pacific Conf. Circuits Syst. (APCCAS), Nov. 2008, pp. 606–609.
[6] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, “Graphics processing unit (GPU) programming strategies and trends in GPU computing,” J. Parallel Distrib. Comput., vol. 73, no. 1, pp. 4–13, Jan. 2013.
[7] S. R. H. Hoole, V. U. Karthik, S. Sivasathan, A. Rahumanathan, R. S. Thyagarajan, and P. Jayakumar, “Finite elements, design optimization, and nondestructive evaluation: A review in magnetics, and future directions in GPU-based, element-by-element coupled optimization and NDE,” Int. J. Appl. Electromagn. Mech., vol. 47, no. 3, pp. 607–627, Mar. 2015.
[8] J. Bolz, l. Farmer, E. Grinspun, and P. Schröder, “Sparse matrix solvers on the GPU: Conjugate gradients and multigrid,” ACM Trans. Graph., vol. 22, no. 3, pp. 917–924, 2003.
[9] J. Rodríguez-Navarro and A. Susín, “Non structured meshes for cloth GPU simulation using FEM,” in Proc. 3rd Workshop Virtual Reality Interaction. Phys. Simul., 2006, pp. 1–7.
[10] C. Cecka, A. Lew, and E. Darve, “Introduction to assembly of finite element methods on graphics processors,” in Proc. IOP Conf. Ser., Mater. Sci. Eng., vol. 10, Bristol, U.K.: IOP Publishing, Jun. 2010, Art. no. 012009.
[11] D. Konatitsch, D. Michéa, and G. Erlbacher, “Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA,” J. Parallel Distrib. Comput., vol. 69, no. 5, pp. 451–460, May 2009.
[12] T. Iwashita and M. Shimasaki, “Algebraic multicolor ordering for paralleled ICCG solver in finite-element analyses,” IEEE Trans. Magn., vol. 38, no. 2, pp. 429–432, Mar. 2002.
[13] M. Woźniak, “Fast GPU integration algorithm for isogeometric finite element method solvers using task dependency graphs,” J. Comput. Sci., vol. 11, pp. 145–152, Nov. 2015.
[14] Z. Fu, T. James Lewis, R. M. Kirby, and R. T. Whitaker, “Architecting the finite element method pipeline for the GPU,” J. Comput. Appl. Math., vol. 257, pp. 195–211, Feb. 2014.
[15] G. R. Markall, D. A. Ham, and P. H. J. Kelly, “Towards generating optimised finite element solvers for GPUs from high-level specifications,” Procedia Comput. Sci., vol. 1, no. 1, pp. 1815–1823, May 2010.
[16] S. Sanfui and D. Sharma, “A two-kernel based strategy for performing assembly in FEA on the graphics processing unit,” in Proc. Int. Conf. Adv. Mech., Ind., Autom. Manage. Syst. (AMIMAS), Feb. 2017, pp. 1–9.
[17] G. R. Markall, A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. D. Dhowin, “Finite element assembly strategies on multi-core and many-core architectures,” Int. J. Num. Meth. Fluids, vol. 71, no. 1, pp. 80–97, Jan. 2013.
[18] C. Cecka, A. Lew, and E. Darve, “Application of assembly of finite element methods on graphics processors for real-time elastodynamics,” in GPU Computing Gems Jade Edition. Amsterdam, The Netherlands: Elsevier, 2012, pp. 187–205.
[19] H.-T. Meng, B.-L. Nie, S. Wong, C. Macon, and J.-M. Jin, “GPU accelerated finite-element computation for electromagnetic analysis,” IEEE Antennas Propag. Mag., vol. 56, no. 2, pp. 39–62, Apr. 2014.
[20] I. Z. Reguly and M. B. Giles, “Finite element algorithms and data structures on graphical processing units,” IEEE Trans. Microw. Theory Tech., vol. 56, no. 8, pp. 2661–2671, Aug. 2008.
[21] A. Dziejonski, P. Sypek, A. Lamecki, and M. Mrozowski, “Generation of large finite-element matrices on multiple graphics processors,” Int. J. Num. Meth. Engng, vol. 94, no. 2, pp. 204–220, Apr. 2010.
[22] A. Dziejonski, P. Sypek, A. Lamecki, and M. Mrozowski, “GPU-accelerated finite-element matrix generation for lossless, lossy, and tensor media [EM programmer’s notebook],” IEEE Antennas Propag. Mag., vol. 56, no. 5, pp. 186–197, Oct. 2014.
[23] A. Dziejonski, P. Sypek, A. Lamecki, and M. Mrozowski, “Communication and load balancing optimization for finite element electromagnetic simulations using multi-GPU workstation,” IEEE Trans. Microw. Theory Tech., vol. 65, no. 8, pp. 2661–2671, Aug. 2017.
[24] N. Bell and M. Garland, “Efficient sparse matrix-vector multiplication on CUDA,” Nvidia, Santa Clara, CA, USA, Nvidia Tech. Rep. NVR-2008-004, 2008.
[25] N. Bell and M. Garland, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” in Proc. Conf. High Perform. Comput. Netw., Storage Anal. (SC). New York, NY, USA: ACM, 2009, p. 18.
[26] M. M. Baskaran and R. Bordawekar, “Optimizing sparse matrix-vector multiplication on GPUs,” IBM Res. Rep. RC24704 (W0812-047), 2008.

[27] M. Naumov, L. Chien, P. Vandermeersch, and U. Kapasi, “Cusparse library,” in Proc. GPU Technol. Conf., 2010, pp. 1–96.

[28] R. Li and Y. Saad, “GPU-accelerated preconditioned iterative linear solvers,” J. Supercomput., vol. 63, no. 2, pp. 443–466, Feb. 2013.

[29] A. Dziekonski, P. Sypek, A. Lamecki, and M. Mrozowski, “Finite element matrix generation on a GPU,” Prog. Electromagn. Res., vol. 128, pp. 249–265, 2012.

[30] A. Margaris, S. Souravlas, and M. Roumeliotis, “Parallel implementations of the Jacobi linear algebraic systems solve,” 2014, arXiv:1403.5805.

[31] Z. Zhang, Q. Miao, and Y. Wang, “CUDA-based Jacobi’s iterative method,” in Proc. Int. Forum Comput. Sci.-Technol. Appl., vol. 1, 2009, pp. 259–262.

[32] T. Wang, Y. Yao, L. Han, D. Zhang, and Y. Zhang, “Implementation of Jacobi iterative method on graphics processor unit,” in Proc. IEEE Int. Conf. Intell. Comput. Intell. Syst., vol. 3, Nov. 2009, pp. 324–327.

[33] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí, “Solving dense linear systems on graphics processors,” in EuroPar 2008—Parallel Processing (Lecture Notes in Computer Science), vol. 5168, E. Luque, T. Margalef, and D. Benitez, Eds. Berlin, Germany: Springer, 2008.

[34] J. Lin and Y. Chen, “Design and implementation of Jacobi algorithms on GPU,” in Proc. Int. Conf. Artif. Intell. Comput. Intell., vol. 1, Oct. 2010, pp. 448–450.

[35] M. U. Torun, O. Yilmaz, and A. N. Akansu, “FPGA, GPU, and CPU implementations of Jacobi algorithm for eigenanalysis,” J. Parallel Distrib. Comput., vol. 96, pp. 172–180, Oct. 2016.

[36] E. Cormie-Bowins, “A comparison of sequential and GPU implementations of iterative methods to compute reachability probabilities,” 2012, arXiv:1210.6412. [Online]. Available: http://arxiv.org/abs/1210.6412

[37] A.-K. C. Ahamed and F. Magoulès, “Efficient implementation of Jacobi iterative method for large sparse linear systems on graphic processing units,” J. Supercomput., vol. 73, no. 8, pp. 3411–3432, Aug. 2017.

[38] P. Pacheco, An Introduction to Parallel Programming, Amsterdam, The Netherlands: Elsevier, 2011.

[39] S. Xiao and W.-c. Feng, “Inter-block GPU communication via fast barrier synchronization,” in Proc. IEEE Int. Symp. Parallel Distrib. Pro- cess. (IPDPS), Apr. 2010, pp. 1–12.

[40] W.-c. Feng and S. Xiao, “To GPU synchronize or not GPU synchronize?” in Proc. IEEE Int. Symp. Circuits Syst., May 2010, pp. 3801–3804.