GPGPU-based parallel computing applied in the FEM using the conjugate gradient algorithm: a review

NILESHCHANDRA K PIKLE^{1,*}, SHAILESH R SATHE¹ and ARVIND Y VYAVHARE²

¹Department of Computer Science and Engineering, Visvesvaraya National Institute of Technology, Nagpur, India

²Department of Applied Mechanics, Visvesvaraya National Institute of Technology, Nagpur, India
e-mail: nilesh.pikle@gmail.com

MS received 7 August 2017; revised 1 December 2017; accepted 8 December 2017; published online 22 June 2018

Abstract. Parallelization of the finite-element method (FEM) has been contemplated by the scientific and high-performance computing community for over a decade. Most of the computations in the FEM are related to linear algebra that includes matrix and vector computations. These operations have the single-instruction multiple-data (SIMD) computation pattern, which is beneficial for shared-memory parallel architectures. General-purpose graphics processing units (GPGPUs) have been effectively utilized for the parallelization of FEM computations ever since 2007. The solver step of the FEM is often carried out using conjugate gradient (CG)-type iterative methods because of their larger convergence rates and greater opportunities for parallelization. Although the SIMD computation patterns in the FEM are intrinsic for GPU computing, there are some pitfalls, such as the underutilization of threads, uncoalesced memory access, lower arithmetic intensity, limited faster memories on GPUs and synchronizations. Nevertheless, FEM applications have been successfully deployed on GPUs over the last 10 years to achieve a significant performance improvement. This paper presents a comprehensive review of the parallel optimization strategies applied in each step of the FEM. The pitfalls and trade-offs linked to each step in the FEM are also discussed in this paper. Furthermore, some extraordinary methods that exploit the tremendous amount of computing power of a GPU are also discussed. The proposed review is not limited to a single field of engineering. Rather, it is applicable to all fields of engineering and science in which FEM-based simulations are necessary.

Keywords. Finite-element method (FEM); conjugate gradient (CG); sparse matrix–vector multiplication (SpMV); assembly-free FEM (AF-FEM); graphics processing units (GPUs); compute unified device architecture (CUDA); parallel computing.

1. Introduction

The finite-element method (FEM) is a widely used approximation method [1] that is applied to solve partial differential equations (PDEs) engendered from various fields of science and engineering, such as solid mechanics [2], electromagnetics [3] and biomedicine [4]. For better generalization of the FEM concept, consider a Poisson equation (second-order PDE) that arises while solving physical problems in different fields of engineering:

$$\Delta\phi = f \quad (1)$$

where Δ is a Laplacian operator and ϕ and f are any real-valued functions. For a 3D space, the Laplacian can be formulated as

$$\Delta\phi = \nabla^2\phi = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right)\phi = f \quad (2)$$

The exact solution for complex real-life problems is not feasible to compute using the Laplacian. These equations can be solved via either the finite-difference method or the FEM. Generating exact or analytical solutions for complex real-life problems with complex geometries is nearly impossible. An approximate solution obtained using the FEM is well suited for these problems. The matrix formulation of the FEM was first introduced for structural applications by John Argyris [5] in 1954, while the term *finite element*, coined by Ray William Clough [6], came into existence in 1960. Since the 1970s, this method has been adopted in engineering applications for simulations using commercial software packages on PCs. Traditionally, the FEM pipeline is divided into the following steps:

1. *discretization* of the domain into a finite set of elements,

*For correspondence

2. *numerical integration* to produce the local element level matrices,
3. *assembly* to map the local element level matrices to a global matrix and form a system of linear equations and
4. *solving* the system of equations.

The assembly step generates a very large sparse system of linear equations of the following type:

$$[A]\{x\} = \{b\} \quad (3)$$

where A is an assembled sparse global matrix, x is a vector of unknowns and b is a right-hand-side assembled known vector. Usually, this generated system is symmetric, sparse and positive definite. The solver step often dominates the entire performance of the FEM pipeline; seldom do the numerical integration and assembly steps dominate by virtue of the higher-order polynomial approximation [7]. For large simulations, e.g., earthquake modelling [8], a myriad of elements are generated via the discretization process, inducing a colossal sparse system of linear equations. This system requires a very large memory storage and many computations in the solver phase. Hence, it is rather implausible to process this system on single-core machines. To circumvent the storage issue, the practical approach is to store the global matrix directly in the sparse matrix storage format [9, 10]. This approach alleviates memory issues. However, for large simulations, this problem is inevitable.

To solve the system of linear equations in Eq. (3), direct or iterative solvers can be used. Direct or factorized solvers, such as Cholesky factorization and LU decomposition, have notoriously high complexity, have a recursive nature and are difficult to parallelize on a graphics processing unit (GPU) due to output dependencies arising in forward or backward substitution. In addition, direct solvers require a large memory space, as the global matrix is stored explicitly without being compressed. Iterative solvers [11] are preferred over direct solvers, as they curtail the memory requirement and computations. These solvers take advantage of the sparsity in the global coefficient matrix by storing it in compressed sparse matrix storage formats. Iterative solvers start with a guessed solution and gradually move towards the actual solution iteratively. The convergence rate of stationary iterative methods [11] such as the Gauss–Seidel, Jacobi, and successive over-relaxation (SOR) iterative methods is slower compared with that of non-stationary methods such as the conjugate gradient (CG), generalized minimal residual (GMRES) and quasi-minimal residual (QMS) methods. The primary focus of this paper is the parallelization of the FEM fused with the CG solver. The CG solver is encapsulated with a very computation-intensive sparse matrix–vector multiplication (SpMV). Sparse matrix storage formats avoid redundant computations (i.e., multiplying by zero) and reduce the memory space requirement. Hence, they are prominently used when optimizing the CG solver. To

improve the convergence rate of the CG solver, preconditioning techniques are used. Assembly-free (AF) strategies [12, 13] obviate the assembly of the global coefficient matrix and solve the system of equations at the element level. These methods require less memory and introduce fine-grained parallelism with a small increase in computational cost.

Advancements in computer architectures and the ingress of parallel computing in scientific applications have resulted in increased performance and throughput. A GPU [14] is an ensemble of hundreds and even thousands of cores on which millions of threads can be executed in parallel. A GPU is not a standalone device. Rather, it acts as a co-processor or an accelerator for a CPU. Initially, GPUs were designed for accelerating graphics-related applications. However, in 2007, NVIDIA introduced general purpose programming on a GPU (GPGPU) using the compute unified device architecture (CUDA) [15] on NVIDIA GPUs [16]. An ensemble of thousands of throughput-oriented cores and a hierarchical memory layout prevail over the FEM computations on a GPGPU. Most of the computations of an FEM solver are of single-instruction multiple-data (SIMD) type and are instinctively favourable on GPUs. GPU-based implementation of the FEM has been proposed in various fields of engineering, such as solid mechanics [17] and electromagnetics [18]. Over the last decade, the number of publications in the field of parallelization of the FEM on a GPU has increased by leaps and bounds. Several techniques are used to increase the performance on a GPU [19], such as reducing the memory transfer between a CPU (host) and GPU (device), properly utilizing threads, reducing the global memory latency using fast local memories, increasing the occupancy, avoiding thread divergence and increasing the coalesced memory access and global memory load/store efficiency.

In 2014, Hoole *et al* [20] presented a good review related to FEM computations on a GPU in the field of magnetics. Some topics, such as SpMV and preconditioning in the CG algorithm, were not covered in that paper. Moreover, there is currently no generalized review article available in the field of GPU-based parallelization of the FEM that is helpful for all newcomers from various fields of engineering. This situation prompted us to write this review article.

This paper is a comprehensive study of the parallelization of the FEM on GPUs and reviews the research that has been done over the last decade. This is a venture in which not much emphasis has been placed on pre-processing and post-processing as it varies with different applications. A discussion regarding the GPU is presented, but it refers to NVIDIA's GPU unless explicitly stated. This paper is not intended to elaborate on the GPU architecture and CUDA programming, which were discussed in detail in [14, 21]. Moreover, we eschew the optimizations regarding dense linear algebra, as the system of linear equations generated after the assembly process is sparse.

This paper elaborates on the observed progression in each step of the FEM, as depicted in figure 1. This paper is organized as follows. Section 2 focuses on discretization, numerical integration and assembly on GPUs. Section 3 is dedicated to parallelization of the CG and SpMV. Section 4 highlights some extraordinary strategies created for parallel optimization of the FEM. Section 5 notes the pitfalls and trade-offs in each step of the FEM while carrying out parallelization on a GPU. Finally, Section 6 concludes the review and presents the future scope and shortcomings found in the literature survey. The goal of this paper is to present a generalized review on the parallelization of the FEM on a GPU that is applicable in any engineering field that requires FEM-based simulations using a CG-type solver.

2. Global Matrix Generation on GPU

The global matrix generation process is further subdivided into three parts:

1. *discretization* of the entire domain by dividing it into finite elements,

2. application of *numerical integration* to each element to form local element matrices and
3. *assembly* of all element-level matrices to form a global coefficient matrix A from Eq. (3).

2.1 Discretization

Discretization (also known as meshing) [22] is the first step of FEM solvers, which divides the domain into a large number of finite discrete elements. As a matter of fact, the whole domain is subdivided into millions or even billions of elements for large simulations [8]. As discretization is the first step in FEM simulation, it affects the performance of subsequent steps and the memory storage. The available open-source mesh generators require a human-machine interface due to included parameters regarding the points, boundary conditions, etc. Sivasuthan *et al* [23] implemented a script-based mesh generation library that runs nonstop for optimization. Structured discretization results in a regular sparsity pattern in the global matrix for which sparse storage formats [10], such as JDS, ELLPACK and DIA, are considered suitable. In contrast, unstructured discretization results in a sporadic sparsity pattern for which CSR, COO and ELLPACK, among others, are well suited. These storage formats are discussed in detail in section 3.3. When a data request is made by the global memory of the GPU, a data segment that is generally 128 bytes in size is returned. This segment of data can be read by neighbouring threads of the half warp without another memory transaction being made. This reduces the global memory access latency for remaining threads of the half warp. This type of memory access is possible only when neighbouring threads request contiguous data, and this phenomenon is referred to as memory coalescing. Structured discretization yields an indisputable regular memory access pattern, whereas unstructured discretization requires special treatment to achieve coalesced memory access on a GPU. The number of elements and element type are important when considering the optimization of the FEM solver. The type of element, e.g., triangular, quad, prismatic or hexahedral brick, determines the optimization strategies and overall performance on a GPU. The selection criteria for the number of elements and type of element are based on the required accuracy, underlying hardware architecture, available memory and distinct applications. As the number of elements increases, the required accuracy also increases, at the cost of an increased memory space and number of computations. Similarly, for higher-order elements, the size of the local matrices generated during the numerical integration step also increases as a consequence of the increase in required memory and number of computations. Banaś *et al* [7] demonstrated this effect conclusively for a standard prismatic element by varying the degree of approximation. The sparsity pattern in the global matrix depends on the node numbering applied during the discretization

Figure 1. Steps in the finite element method pipeline.

process. For a coalesced memory access pattern on a GPU, the semi-bandwidth of the global matrix should be minimum [24].

Currently, the fixed grid FEM (FG-FEM) is a trending topic in the field of FEM parallelization. The novel idea of the FG-FEM was first introduced by García-Ruítz and Steven [25] for 2D elasticity problems. The traditional FEM discretizes the domain by considering the physical boundaries. In the FG-FEM, a fixed grid of equally sized elements is superimposed onto the domain under consideration. The FG discretization is depicted in figure 2. The elements fall into 3 categories: inside the grid (*I*), outside the grid (*O*) and neither inside nor outside (*NIO*) the grid. The domain-specific properties are considered as constant for all *I* and *O* elements. For all *I* elements, the local matrix is computed in a manner similar to that of the traditional FEM, as given by Eq. (4):

$$A_I^e = \int_{\Omega^e} B^T D_I B d\Omega^e, \quad (4)$$

$$A_O^e = \Delta \int_{\Omega^e} B^T D_O B d\Omega^e = \Delta \cdot A_I^e. \quad (5)$$

For *O* elements, the elemental matrices are multiplied by a small constant Δ that is close to zero, as shown in Eq. (5), where A_O^e and A_I^e denote the elemental matrices for the outer and inner elements, respectively. The B matrix contains derivatives of the shape functions, and D is a domain-specific property matrix. For all inside elements *I*, the generated local matrices are the same because they share the same domain-specific properties and equal-sized elements. Thus, instead of generating and storing local matrices for all elements, a single elemental matrix is generated and stored. This dramatically reduces the memory usage on the GPU. As *NIO* elements lie in two different domains, the domain-specific properties are not continuous inside the element. For these elements, the local matrices can be computed either as a weighted average of *I* and *O* elements or some factor of

these elements. The elemental matrix for *NIO* elements can be computed as

$$\begin{aligned} A_{NIO}^e &= \ell^e \int_{\Omega^e} B^T D_I B d\Omega^e + (1 - \ell^e) \int_{\Omega^e} B^T D_O B d\Omega^e \\ &= \ell^e A_I^e + (1 - \ell^e) A_O^e. \end{aligned} \quad (6)$$

In Eq. (6), ℓ^e is a volume ratio V_I^e/V^e , where V_I^e is the volume of element *e* that lies inside the domain and V^e is the total volume of element. Elemental matrices in the outside domain are excluded from subsequent computations, as they increase the condition number of the global matrix [25].

2.2 Numerical integration

Numerical integration produces element-level equations or matrices using the shape functions and domain-specific properties, which will be assembled to form a global matrix. For example, if we consider the stress-strain analysis in a structural problem, the numerical integration process carried out is as follows:

$$A_e = \int_{\Omega^e} B^T D B d\Omega^e \quad (7)$$

where Ω^e defines the elemental local domain, D represents the material property matrix and the B matrix contains the derivatives of the shape functions. Parallelization of the numerical integration process is a topic seldom visited by researchers, as it has less effect on the FEM solver for lower-order polynomial approximations. Furthermore, there are no dependences during the generation of local matrices. Hence, parallelization is straightforward. Therefore, optimizations are applied only at the memory level to accelerate the numerical integration step. The computational complexity of numerical integration is application dependent. The number of computations associated with numerical integration increases with the increase in the order of elements, the order of approximation (p) and the number of shape functions. As stated by Kružer and Banaś [26], for higher-order approximations [27] of order greater than 5, the numerical integration process can dominate the overall performance of FEM simulations. To perform the numerical integration, three loops are necessary: one loop over elements and two on the shape functions, as shown in Eq. (7). The obvious way to parallelize the numerical integration is to parallelize the loop over elements. Parallelization of the loop over elements is suitable on multi-core CPUs because of the large cache memories. For a GPU, the memory resources available per thread are fewer. Therefore, the data required to perform numerical integration do not fit into the fast local memories. Macioł *et al* [28] proposed decomposition at the element level to tackle the limited memory issue. The computations of elemental matrices are divided into horizontal strips such that they

Figure 2. Fixed grid discretization in the FEM.

can fit into the shared memory. Therefore, multiple threads can operate on a single element. From Eq. (7), it is quite clear that though the computations of the matrix A_e are divided into strips, it requires the recalculation of the derivatives of all shape functions. This is the only penalty for this strategy. However, the authors achieved a significant speed-up for this method.

When the per-thread grain size of the computations is too coarse, it results in poor utilization of the computing power of the GPU. In contrast, a very fine-grained data decomposition yields underutilization of the threads. A mixed-grain approach of the numerical integration process on a GPU was suggested by Filipovič *et al* [29], who alluded to the underutilization of thread warps as well as available resources and achieved a significant speed-up of $8.3\times$. The accuracy of the solution, the number of computations and the number of entries in the elemental stiffness matrix increase with the degree of the polynomial (shape function). Banaś *et al* [7] elaborated on the computational cost and the memory cost for the degree of approximation from 2 to 7 and showed how they affect a GPU. They presented a portable implementation of this method using OpenCL on two GPUs from different vendors, one from NVIDIA and the other from AMD. To reduce the global memory latency, a tiling technique was suggested in that paper. The shared memory latency of a GPU is approximately 100 times faster than that of global memory. Tiling allows maximum reads from the shared memory of a GPU and reduces the global memory traffic. Data decomposition in the numerical integration process is feasibly done either by dividing the number of elements into disjoint subsets (coarse-grained) or by decomposing the local element matrix into further chunks (fine-grained). The first approach is suitable for conducting parallelization on a multi-core CPU, as less computing power is available, whereas the second approach is suitable for throughput-oriented many-core GPUs.

Selection of the element type, data structures in which to store auxiliary data (array indices in sparse matrix storage formats), and thread mapping also affects the overall performance on the GPU. Dziekonski *et al* [30] determined the selection criteria between curvilinear and rectilinear elements and the trade-off between accuracy and execution time based on the minimum quadrature points. In the following year, they proposed a multi-GPU approach for curvilinear tetrahedral elements [31] to parallelize the loop over elements as well as over the inner (element matrix) loop. They achieved a significant speed-up of $77.28\times$ compared with a single core and $15.56\times$ compared with a multi-threaded (24 CPU cores) implementation. In the numerical integration process, the inner loops carry out a smaller element-level dense matrix-matrix multiplication (on the shape functions). For this purpose, the *cublasDgemm()* routine from the CUBLAS [32] library is inconvenient, as the elemental matrix size is smaller. The dense matrix-vector product for a GPU based on the

cublasDgemm() routine can achieve significant speed-up if the matrix size is sufficiently large (at least 100×100). A CUDA *streaming* technique was suggested by Dziekonski *et al* [33] to alleviate the aforementioned problem by means of overlapping data transfer and the kernel execution by dividing independent tasks into two different CUDA streams. This strategy has achieved a significant speed-up of $81\times$ and $19\times$, respectively, compared with single-core and 24-core CPU machines.

Generally, the global memory of a GPU is smaller in size than CPU memory. Hence, the scalability of the numerical integration process is limited by the size of the global memory. An iterative approach was suggested by Dziekonski *et al* [31] to leverage the available GPU memory. The elements are divided into subsets such that the data of each subset can fit into the global memory. The partial computations are performed on a GPU iteratively for each subset. Finally, the results are gathered in the CPU main memory. This multi-GPU iterative scheme achieves a speed-up of a factor of $100\times$. Hardware architectures are being continuously upgraded in terms of the memory size, number of registers and bandwidth, among others. Another issue regarding the parallel implementation of the FEM is the portability on GPUs from different vendors. OpenCL [34] supports programming on GPUs from different vendors and effectively utilizes the total available computing power, including that of the CPU. Krużer and Banaś [26] proposed a numerical integration technique that uses OpenCL for higher-order finite prismatic elements and is highly portable.

As the order of approximation increases, the number of operations and the amount of memory required to store the local matrices increase linearly (see figure 3). The shared memory of a GPU is smaller (in kilobytes), but the access time is approximately hundred of times faster than that of global memory. For better results, maximum read/write should be performed from the shared memory. A new data decomposition technique was suggested by Banaś *et al* [7], which splits the local matrix into chunks and in which operations are performed on subsequent chunks in parallel. The local stiffness matrix is stored in the fast local memory, and the shape functions, with their derivatives, are stored in the register memory to avoid global memory access latency. Therefore, the performance of this operation can be limited due to inadequate memory resources on a GPU. To exploit the maximum available parallelism, Banaś *et al* [7] applied a two-way parallelism, with one being a loop over elements and the other being within a local element. To alleviate the global memory access penalties, the local matrix can be stored in a 1D array instead of a 2D array and decomposed to fit into the fast local memory. However, this approach suffers from repeated reads and computations over shape functions and their derivatives. The loop permuted implementations [7, 26, 33] (QSS, SQS, SSQ) were investigated by Banaś *et al* [35], who sought to choose the best approach based on the underlying hardware

Figure 3. Total number of operations for the numerical integration algorithm, and the sizes of data structures used for 3D prismatic elements as a function of the order of approximation p [26].

architecture. They concluded that no strategy can be selected as the best for all architectures; rather, it is application dependent. Woźniak [36] proposed a task-based parallelism that computes the task dependence graph during the numerical integration process for higher-order B-spline projections. Independent tasks (numerical integration of elements) at the same level are executed in parallel and achieve a speed-up factor of $2\times$.

The numerical integration process under multi-core and many-core architectures was analysed by Mamza *et al* [37]. In their paper, they compared both CUBLAS-based and CUDA-*stream*-based implementations on two GPUs. They observed that a kernel-based implementation is two times faster than a multi-core-CPU-based implementation. Numerical integration for lower-order elements was presented by Knepley and Terrel [38]. They proposed a batch-wise decomposition of elements per thread to perform numerical integration on a GPU. An interleaved strategy was proposed to avoid global memory latency while storing the numerical integration results. This strategy overlaps the storing and computing operations, thus minimizing the memory latency. Therefore, the optimal choice in regard to the parallelization strategy depends on the polynomial order, dimensions of the problem, equation being solved and required accuracy.

2.3 Assembly to form global matrix

During the assembly procedure [24], local element matrices are assembled to form a global matrix, and a right-hand-side element vector is assembled to form the global right-hand-side vector, as shown in Eqs. (8) and (9), respectively:

$$A = M(A_e) = \sum_{e=1}^{\varepsilon} M_e^T A_e M_e \quad (8)$$

$$b = M(b_e) = \sum_{e=1}^{\varepsilon} M_e b_e \quad (9)$$

where A_e is a local elemental matrix and ε is the total number of elements. Local elemental matrices are mapped to global matrix A using the local-to-global mapping function M . Similarly, a mapping function is also applied to the right-hand-side vector of known values in the system of linear equations in Eq. (3).

The global matrix generated after the assembly needs to be stored in a compact sparse matrix storage format. The GPU instance of the assembly process maps a single thread per elemental computation. Although this seems to be a very simple computation, it cannot achieve the best performance on a GPU, as it requires uncoalesced access to the global memory to search for the global matrix location. It also increases the global memory footprints when accessing auxiliary data such as the element-node connectivity information. The assembly process on a GPU was introduced by Cecka *et al* in [39]. Common nodes between elements are mapped at the same location in the global matrix. Thus, they share a common memory location in the global memory. This causes a severe problem, i.e., race conditions, when different threads try to access the same memory location in the global memory. For parallel updating in the global matrix, synchronization (atomic operation) is needed to avoid the race conditions, which entails a performance bottleneck. Atomic operations are more costly due to serialization of the global memory access. Furthermore, most of the current generation GPUs do not support atomic operations for double precision (DP). Different pragmatic approaches, such as colouring [8, 40, 41] and the task dependence graph [36], have been suggested to avoid these race conditions. These approaches alleviate synchronization overheads but do not remove them completely. The principal disadvantage of both of these synchronization methods is the need for pre-computing, which is hardly parallelizable. However, as it is performed only once as a part of preprocessing, the effect of the computational time can be neglected.

Fu *et al* [42] proposed a hierarchical assemblage strategy that avoids the synchronization overheads dramatically. They proposed a patch-based assembly scheme that ensures coalesced global memory access and effective synchronization. The contribution of each node or degree of freedom (DoF) is accumulated in the shared memory, and the results are stored in the global memory in chunks. The memory coalescing is achieved by storing nodal coordinates into different arrays instead of a single array. For example, in the case of a 2D domain, the nodal coordinates are stored in two arrays: one for x -coordinates and the other for y -coordinates. This storage pattern avoids the strided access pattern while accessing the nodal coordinates x and y . The optimization techniques, such as effective utilization of the shared memory and threads, reduction of the global

memory latency, data decomposition, and reduction of the data transfer from a host (CPU) to a device (GPU), and their trade-offs were excellently elaborated upon with experimentation by Cecka *et al* [43].

For coalesced memory access, the selection of data structures in which to store data related to element matrices is prominent. The optimal choice regarding the data structure used to store the auxiliary data (nodal coordinates, node numbering, element numbering, etc.) depends on the characteristics of the target architecture. The memory efficiency can be improved using a coalesced memory access pattern on a GPU and via proper cache memory utilization on a CPU. Markall *et al* [44] concluded that node-wise storage for a CPU and element-wise storage for GPUs yield the best performance. Figure 4 shows the discretization of a 2D domain using triangular elements. The CPU and GPU storage formats used to store the element-node connectivity information are also depicted in figure 4. The element-wise storage format on a GPU improves the coalesced memory access. Markall *et al* [45] suggested two strategies for global assembly: *Addto* and the *local matrix approach* (LMA). The *Addto* strategy directly maps the local element matrix entries directly to the global matrix using the local-to-global node/DoF mapping function. This process requires more costly atomic or colouring approaches to avoid the race conditions and thus ensure valid results. In contrast, the LMA approach obviates the generation of the global matrix and assembles the solution vector after calculating the elemental matrix–vector

product. This increases the coalesced memory access as well as the effective bandwidth utilization. As each node is associated with many neighbouring elements, the number of computations performed in the LMA approach is much larger than that in the *Addto* method. Based on this observation, the authors concluded that the LMA approach is suitable for a GPU and that the *Addto* approach is suitable for a CPU. Recently, Sanfui and Sharma [46] proposed a modified *Addto* approach for a GPU using two kernels. For 3D domain discretization, each node is associated with 3 DoFs. Therefore, each node has non-zero entries in the corresponding three rows of the global matrix. The number of non-zero entries in the global matrix depends on the number of neighbouring nodes. If the number of neighbouring nodes is n , then the total number of non-zero entries in a particular row is $3(n + 1)$. The first kernel determines the row indices and column indices associated with each node, which are stored in a 1D array. The second kernel accumulates the contribution based on these indices at a particular node. Finally, the local elemental entries are mapped to the global matrix. This strategy outperforms the *Addto* strategy proposed by Markall *et al* [45] on a GPU.

There are three ways to assemble the global matrix: (1) by the non-zero elements in a row of the global matrix (fine-grained); (2) by row of the global matrix (medium-grained) and (3) element-wise (coarse-grained). The choice between the aforementioned strategies still depends on factors such as the polynomial order, the size of the element matrices and the application, as addressed by Cecka *et al* [39, 47]. The authors investigated the performance for all levels of granularity on a GPU. They proposed mainly two assembly strategies, denoted as *GlobalNZ* and *SharedNZ*. In the case of *GlobalNZ*, to avoid the race conditions, two-phase kernels are proposed. In the first phase, computations are performed by a thread per element, and the results are stored in the global memory at distinct locations in a coalesced fashion. In the second phase, the kernel gathers the contributions of each element in the global matrix. The *SharedNZ* method utilizes the shared memory to store the elemental data to reduce the global memory footprints. This strategy is limited by the availability of the shared memory. Assembly by a single thread per element results in more *atomic* operations while updating the global matrix. DoFs associated with all elements are pre-computed on a CPU and sent to the GPU. Each thread retrieves the corresponding element DoFs and updates in parallel. One can observe that from the pattern of the final global matrix, the non-zero entries in each row correspond to the contributions of neighbouring elements by the edge. To avoid the CUDA *atomic* operations, assembly by edge was proposed by Meng *et al* [48]. Each thread assembles the row of the global matrix in the shared memory for faster memory access. As each thread computation is independent, the synchronization overheads due to the atomic operations are reduced. There is no guarantee that the global matrix contains a constant number of non-zero entries per row. If they

Figure 4. (a) A 2D domain decomposed into three elements. (b) Data storage pattern for CPU implementation. (c) Data storage pattern for GPU implementation. Threads accessing data in different elements (indicated by the arrows) achieve coalescence [44].

vary substantially, this thread mapping strategy leads to an unbalanced workload among the threads.

Assembly using direct sparse matrix formats causes uncoalesced access to the global memory. Moreover, the global matrix storage method determines the parallelization strategy for the solver step of the FEM. The LMA and matrix-free approach were elaborated by Reguly and Giles [49]. The former approach performs the direct assembly in the sparse matrix storage format, while the latter one does not perform the assembly at all. Rather, the matrix-free approach computes the local matrices in each iteration of the CG. This approach greatly reduces the amount of memory required to store local or global matrices by adding extra computations, which is not a problem on a GPU for lower-order elements. Dziekonski *et al* [31, 50] proposed direct sparse matrix storage for assembly on a multi-GPU platform. First, the assembly is performed directly in the COO storage format; then, it is converted into the CSR format on a GPU. For large matrices (higher-order elements), the scalability of the numerical integration and assembly process is restricted by the GPU memory size. To overcome this problem, Dziekonski *et al* adopted an iterative approach from [31], which decomposes the computations such that the GPU memory should not overflow. Each GPU receives the data in chunks and performs computations independently. After performing the computations, the results are gathered in the CPU memory. Recently, Dziekonski *et al* [51] proposed optimization strategies related to load balancing and communication overheads on a multi-GPU platform.

To circumvent these traditional parallel computing approaches, the novel idea of optimization of the numerical integration at the compiler level is prevailing for the automated FEM solver [52] under heterogeneous parallel computing architectures. Furthermore, a prototype compiler implementation for the unified form language (UFL), which allows generating a very efficient low-level code for the available hardware, was developed in the FEniCS project [53]. This compiler produces a directed acyclic graph (DAG) for conversion of the UFL to the CUDA, where every DAG node is a CUDA kernel. The performances of both the original CUDA-based code and the code derived using the UFL-based compiler were compared. It is observed that both codes have approximately identical execution times. Some compiler-level optimization strategies, such as pre-computation of the loop invariant code, expression splitting and data padding, are discussed in the COFFEE compiler [54].

3. GPU-based CG and PCG solver

Often, the system of equations resulting from the assembly phase of the FEM is sparse. Hence, iterative solvers are preferred over direct solvers, which are

suitable for dense systems. The CG [55] (see Algorithm 1) is a widely used iterative solver for the sparse positive definite matrix in scientific computing because of its higher convergence rate, lower time complexity, ease of parallelization and optimal storage requirement. Different versions of the CG [11], such as the bi-conjugate gradient (BiCG), BiCG-stabilized (BiCG-STAB) and CG squared (CGS), are used for distinct coefficient matrix types. The CG solver consists of five types of computations: SpMV, dot product, scalar–vector product, vector addition and subtraction. These operations embarrassingly follow the SIMD computing pattern, which is suitable on a GPU. The condition number of the global matrix in most real-life applications is very high. Convergence of the CG itself for such a system can be poor. To improve the convergence of the CG method, preconditioning is often conducted, which is referred to as the preconditioned conjugate gradient (PCG) (see Algorithm 2). Some of the important topics that are prominent during the parallelization of the CG solver on a GPU are 1. sparse storage formats; 2. SpMV; 3. parallel preconditioning; 4. convergence improvement; 5. reduction of the data transfer between a host (CPU) and a device (GPU); 6. accuracy; 7. mapping of the operations on a GPU (thread/warp/blocked/tiled); 8. effective utilization of the fast local memories of a GPU; 9. improvement of the global memory load/store efficiency and 10. precision (single/double/mixed) used for implementation, among others. The most computation-intensive part of the CG and PCG is SpMV (step 1 and step 3 of Algorithm 1 and Algorithm 2, respectively), which is a topic highly contemplated upon by researchers. Furthermore, preconditioning induces extra computations for the generation and application of a preconditioner. Past studies have shown that the performance of the CG can be increased at either the computation level or the convergence level. This research is being carried out in parallel, as there is a trade-off between computational cost of preconditioner and rate of convergence of PCG.

Algorithm 1: Conjugate Gradient

```

 $(res)_0 = b - Ax_0, \quad d_0 = (res)_0$ 
 $k = 0$ 
REPEAT
Step1:  $\alpha = \frac{(res)_k^T (res)_k}{d_k^T A d_k}$ 
Step2:  $x_{k+1} = x_k + \alpha_k d_k$ 
Step3:  $(res)_{k+1} = (res)_k - \alpha_k A d_k$ 
if converges STOP
Step4:  $\gamma = \frac{(res)_{k+1}^T (res)_{k+1}}{(res)_k^T (res)_k}$ 
Step5:  $d_{k+1} = (res)_{k+1} + \gamma_k d_k, \quad k = k + 1$ 
END REPEAT
Result is  $x_{k+1}$ 

```

Algorithm 2: Preconditioned Conjugate Gradient

```

(res)0 = b - Ax0, z0 = M-1(res)0, d0 = z0
k = 0
REPEAT
Step1: α =  $\frac{(res)_k^T z_k}{d_k^T A d_k}$ 
Step2: xk+1 = xk + αk dk
Step3: (res)k+1 = (res)k - αk A dk
if converges STOP
Step4: zk+1 = M-1(res)k+1
Step5: γ =  $\frac{z_{k+1}^T (res)_{k+1}}{z_k^T (res)_k}$ 
Step6: dk+1 = zk+1 + γk dk, k = k + 1
END REPEAT
Result is xk+1

```

3.1 Precision

The FEM is better known as an approximation method due to the discretization of a domain into finite elements and errors on the computer being rounded off, which worsen the accuracy if a higher-precision format is not adapted for large simulations. There is a precision trade-off in iterative solvers between single precision (SP) and double precision (DP). Initially, GPUs were not intended for DP computations. Itu *et al* [56] showed that the throughput of SP is approximately two times or more than that of DP. From table 1, one can notice that for matrix multiplication, the DP execution time is approximately 3 times the SP execution time on a GPU. Obviously, similar observations can be expected for the throughput as well. However, though DP achieves higher accuracy compared with SP, it doubles the memory requirement. A DP multiplication is approximately equal to four SP multiplications, and a DP addition is approximately equal to two SP additions. To balance the effects of SP and DP, a mixed precision refinement was proposed by Göddeke *et al* [57] for FEM computations. Baboulin *et al* [58] elaborated this refinement in iterative solvers to leverage both the SP and DP strategies. Operations with a time complexity of $O(n^3)$ are performed in SP, while operations with a complexity of $O(n^2)$ are conducted in DP [59]. In the case of the CG and PCG, SpMV (step 1 of Algorithm 1 and Algorithm 2) is computed in SP. The updates, such as the residual (step 3 of Algorithm 1 and Algorithm 2 and step 4 of algorithm 2), solution vector (step 2 of Algorithm 1 and Algorithm 2) and direction update vector (step 5 of Algorithm 1 and step 6 of Algorithm 2), are computed in DP. To solve the system of linear equations arising from the FEM, 99% of the computations

can be performed using SP to achieve the same accuracy as that of DP. To achieve better accuracy for FEM simulations, a mixed precision approach was presented by Göddeke *et al* [60] and Buttari *et al* [59]. The most computation-intensive but not so precise operations are computed on a GPU using a 32-bit float. More precise operations, such as residual computation and result updating, are computed on a CPU using 64-bit DP. The authors achieved nearly the same accuracy compared to the 64-bit execution on a CPU.

3.2 Parallel Preconditioning for CG

The convergence rate [11] of iterative solvers depends on the spectral properties of coefficient matrix A from Eq. (3). Preconditioners are often used in iterative solvers to increase the convergence rate by improving the spectral properties of the system of equations. A preconditioned matrix approximates either A in the case of incomplete factorization (IF) and direct preconditioners or A^{-1} in the case of sparse approximate inverse preconditioners (SAIPs). Instead of solving the system shown in Eq. (3), a transformed system (see Eq. (10)), which may have better spectral properties, is solved.

$$M^{-1}Ax = M^{-1}b \quad (10)$$

where M is the preconditioned matrix that approximates A . There is a trade-off between construction and application of the preconditioners on a GPU. Preconditioners must be easy to construct, apply and parallelize. Direct preconditioners, such as Jacobi and block Jacobi, are easy to construct and parallelize but have a poor convergence rate. These types of preconditioners are directly derived from the global matrix without processing. Generally, two types of preconditioners are used in the parallel implementation of iterative solvers: either IF-based or SAIP [61]. The trade-off between the selection of a preconditioner and application on a GPU was exemplified by Li and Saad [62].

3.2a Parallelization of incomplete factorized preconditioner: For IF preconditioners such as incomplete Cholesky (IC) and LU, the PCG requires one SpMV and two triangular solvers (forward/backward substitution), and the BiCG [11] requires two SpMV and four triangular solvers. The IC preconditioner is obtained by factorizing the coefficient matrix $A = MM^T$, where M is a lower triangular matrix. IF preconditioners are hostile for parallelization on

Table 1. Performance of matrix–matrix multiplication on NVIDIA’s GTX260 GPU for single and double precision [56].

Operation	Exec. time (μs)	Global memory throughput	Instruction throughput
Float kernel	10178.2	65.86	0.724
Double kernel	32968.2	41.83	0.403

a GPU due to output dependences in the solution phase. Using IF preconditioners, step 4 of Algorithm 2 is computed via forward substitution. A comparison of CUSPARSE-based [63] implementation of IF-based preconditioners such as IC and incomplete LU (ILU) with Intels Math Kernel Library (MKL) [64] was shown by Naumov [65]. This implementation achieves a speed-up of only $2\times$ due to the output dependences in the forward or backward substitution. Level scheduling [62] alleviates the dependence issue in the forward/backward substitution by considering the sparsity pattern of matrix M . Unknowns $u(i)$ are divided into independent levels $l(i)$ (no dependence in forward or backward substitution due to the sparse pattern). Therefore, parallelization can be applied at each level, and the level can be computed as

$$l(i) = 1 + \max_j \{l(j)\} \text{ for all } j \text{ } L_{ij} \neq 0 \quad (11)$$

where L_{ij} is a lower triangular matrix in CSR format. This strategy is exacerbated on a GPU with small levels (for a denser matrix), as it decreases the number of independent rows of the triangular matrix. To exploit the parallelism in ICCG, the algebraic colouring strategy was proposed by Iwashita and Shimasaki [40, 41] for multi-core systems. The proposed technique divides independent rows of a triangular matrix based on the output dependences. Each row is categorized into a colour set, and rows (DoFs) having the same colour are not dependent on each other. Therefore, the rows can be executed in parallel. It is notable that independent rows in a colour set are computed in parallel, whereas each colour set is launched in sequence. This strategy also has the same drawback as that of level scheduling for fewer colours on a GPU. An increase in the number of colours (number of independent unknowns) reduces the degree of parallelism and increases the synchronization overhead by launching more kernels. Fialko and Zeglen [66] observed that the performance of the CG operation depends on the density of the coefficient and preconditioner matrix. Selection of the target platform (CPU or GPU) for the best performance also depends on the characteristics of these matrices. GPUs are more suitable for the sparse matrices, as the level scheduling from Eq. (11) can be effectively applied, whereas for slightly denser matrices, CPUs can achieve the best performance. Gao *et al* [67] proposed a novel parallelization approach for the forward/backward substitution method on a GPU. This strategy is applicable and limited to structured sparse matrices with diagonally dominant non-zero entries. The parallelization is based on the ordering of the independent computations per thread block.

3.2b Sparse approximate inverse preconditioning on GPU: To circumvent the tediousness of IF-based preconditioners, sparse approximate inverse preconditioners have been proposed in various papers [68, 69]. A SAIP approximates

the inverse of the coefficient matrix A from Eq. (3). The approximate inverse is computed using the least squares method by minimizing the residual of the norm

$$\|(AM - I)\|_F^2 \quad (12)$$

where M is an approximate inverse of A and I is the identity matrix. Equation (12) can be further decomposed into n independent least squares problems to minimize:

$$\min_{m_j} \|(Am_j - i_j)\|_2^2, j = 1, 2, 3 \dots n \quad (13)$$

where i_j is the j^{th} column of the identity matrix and m_j is the j^{th} column of the preconditioned matrix M . Equation (13) clearly implies the parallelization introduced in the SAIP due to SpMV. Application of the preconditioner is replaced with SpMV, unlike the use of forward/backward substitution in IF-based preconditioning. The generated preconditioner matrix is also sparse, and the computations for column j of matrix M are independent. Therefore, effective utilization of the GPU computing power and memory can be achieved. The SAIPs are further divided into two subcategories: static and adaptive. In the case of static SAIPs, the preconditioner is computed during preprocessing, whereas in adaptive preconditioning, the preconditioner is initialized with some simple sparse pattern. Based on some threshold value of the residual, its augmentation will be carried out in each iteration.

A multi-GPU SAIP technique based on the heuristic incomplete Poisson's preconditioner was proposed by Ament *et al* [70]. The proposed preconditioning approach is matrix-free and highly parallelizable compared with factorized preconditioners. The convergence rate of this preconditioner resides in between that of the simple Jacobi and IF-based preconditioners. The proposed preconditioner is limited to structured discretization only. A more generalized SAIP based on Neuman series expansion was proposed by Helfenstein and Koko [71]; it is applicable to both structured and unstructured discretizations. The authors have achieved a speed-up of approximately $10\times$ on a single GPU by mapping a warp to a row in SpMV. A GPU-based explicit SAIP [71] using the retention parameter [72] (number of elements kept in the SAIP) was suggested by Gravvanis *et al* [73]. They concluded pragmatically that a significant improvement can be achieved if the retention parameter is a multiple of the semi-bandwidth of the coefficient matrix. Reordering using the reverse Cuthill–McKee [74] was carried out to increase the convergence rate of IFP-based solvers by Fujiwara *et al* [75]. This reordering approach was proposed by Camargos *et al* [76] for GPUs and achieves significant speed-up. Recently, Bernaschi *et al* [77] proposed GPU-based application of a SAIP using two matrix storage formats (CSR and ELLPACK) and achieved a significant performance enhancement.

3.3 SpMV on GPU

SpMV is the most dominant computation of CG and PCG solvers. The sparse matrix generated from the assembly process is dependent on application and can have different sparse patterns. Ample research on sparse matrix computations on a GPU has been conducted over the last decade, as it is a key computation in FEM simulations. Linear systems of equations engendered from the FEM of type $Ax = b$ are sparse in nature. As SpMV computations are memory bound, they require an excess of memory transactions, which is detrimental for the performance on a GPU. Furthermore, irregular memory access patterns and lower arithmetic intensity deteriorate the performance of SpMV. The number of computations in SpMV cannot be reduced. Therefore, optimizing the global memory access is the only way to improve the performance of SpMV. Sparse matrices are generally stored in compact storage formats to avoid wasting memory by storing zeros. The effective utilization of and access pattern to GPU memory are very important while improving the performance of SpMV. The sparsity pattern depends on various factors, such as the discretization type and node numbering, as explained in section 2.1.

To understand the different matrix storage formats, a model matrix is taken into consideration, as depicted in figure 5. Using this model, some basic matrix storage formats are also depicted in figure 5. Copacetic analysis of

different unblocked sparse matrix storage formats such as CSR, COO, ELLPACK, DIA and their respective SpMV GPU instances were presented by Bell and Garland [10]. The implementations of SpMV using these storage formats are available in the NVIDIA's CUSPARSE library [63]. All the recently proposed SpMV optimization techniques are compared to SpMV based on CUSPARSE library (codes are available at [78]). While explaining the recent SpMV strategies, it is necessary to understand these basic strategies. An excellent review on GPU-based iterative solvers related to SpMV and preconditioning approaches was presented by Li and Saad in 2010 [62].

For diagonally dominant matrices, the DIA format (see figure 5-1) is more suitable. These matrices are generated in the FEM using a tensor or stencil grid. The DIA storage format does not require extra memory space to store column and row indices. Rather, it can be computed implicitly using the *offset* array. This effectively reduces the memory footprints on a GPU. The DIA format is memory- and time-efficient but not applicable to matrices with a sporadic sparsity pattern. The ELLPACK storage format (see figure 5-4) is more generalized than the DIA format. The non-zero entries are stored in a matrix of size mk , where m is the number of rows of the matrix and k is the $\max(nnz_i)$, where nnz_i is the number of non-zeros in row i . For rows having fewer non-zeros than k , they are padded with zeros. If the number of non-zeros per row varies substantially, the

Figure 5. Sparse matrix storage formats.

ELLPACK format is not efficient, as zero padding causes the need for extra memory as well as redundant computations. The data stored in the VAL array in ELLPACK are in column-major-order, which upsurges the coalesced memory access. The ELLPACK storage format is not adapted directly in practice, as the sporadic pattern does not meet the criteria for the k value. The coordinate-wise (COO) storage format (see figure 5-2) is the simplest storage format for construction, as it stores the row and column indices for every non-zero entry. Each thread on a GPU computes a single multiplication of the corresponding non-zero entry and the input vector entry. The COO format increases the memory footprints excessively, as each thread accesses the global memory in an irregular fashion for 3 arrays: VAL, COL and ROW. The GPU instance of SpMV using the COO format needs synchronization to avoid race conditions while updating the result vector. For this reason, the COO storage format is not considered as itself in practice. It is sometimes used as an intermediary between the assembly process of the FEM and other optimized sparse matrix storage formats [33]. It is also used in hybrid models to support other optimized storage formats. In the compressed row storage (CSR) format (see figure 5-3), the non-zero entries per row are implicitly calculated using the RPTR vector as follows: $\text{RPTR}[i + 1] - \text{RPTR}[i]$. The CSR format alleviates the global memory transaction overheads by reducing the ROW vector access. Bell and Garland [10] proposed two versions of CSR-based SpMV: *scalar*-CSR and *vector*-CSR. The former version uses a single thread per dot product of a row of the matrix and input vector, whereas in the latter version, a warp of threads is used to carry out this operation. *Scalar*-CSR endures uncoalesced memory access to the VAL vector. *Vector*-CSR alleviates this issue but still suffers from partially uncoalesced access. Hybrid storage formats are used to combine the two storage formats to take advantage of both. Bell and Garland [10] proposed a hybrid (HYB) approach that combines COO and ELLPACK (see figure 5-5). This strategy avoids the overheads of zero padding of the ELLPACK format by restricting the columns of the VAL array up to k , where k is selected such that zero padding is not required to store the VAL array. Non-zero entries remain after the k columns are stored in the COO format. Two separate SpMV kernels are launched for the ELLPACK and COO formats. SpMV based on the hybrid strategy outperforms all the other strategies discussed earlier. Until now, we have discussed the basic sparse storage formats and their respective SpMV implementations on a GPU. Most of the optimized storage formats and SpMV strategies are derived from these basic storage formats.

Blocking techniques are often used to improve the performance by reducing the memory bandwidth requirement. The blocked sparse matrix storage formats also increase the coalesced memory access pattern on a GPU (see figure 5-7 for the blocked CSR format). Monakov and Avetisyan [79] proposed a hybrid blocking strategy using the blocked CSR

and blocked COO formats. Non-zero entries are grouped together in a block of size $B_r B_c$, and indices are stored on each block, unlike for each non-zero entry in the CSR and COO formats. The blocked storage formats introduce redundant computations if the block size does not match that of all non-zero entries for the irregular sparse pattern. Therefore, blocking strategies are intrinsic to add the overheads of zero entries in the blocks. The performance of ELLPACK is poor when there is substantial variation in the number of non-zero entries per row because of the redundant computations (i.e., multiplying by zero) due to the zero padding. The blocked-ELLPACK (BELLPACK) storage scheme proposed by Choi *et al* [80] avoids the aforementioned snags. For fewer non-zero entries and a larger block size, the BELLPACK format also introduces redundant computations by padding zeros to match the block size. However, the amount of zero padding is reduced in BELLPACK compared with the original ELLPACK format. The ELLPACK-R method, proposed by Vázquez *et al* [81], mitigates the aforementioned issue by keeping track of the row length in the data array. Reordering [74] is often applied to sparse matrices to cluster the non-zero entries together (to reduce the bandwidth of a sparse matrix). Moreover, reordering induces increased spatial locality, which results in increased coalesced memory access. However, reordering is tentative for adding preprocessing overheads. The reordered BELLPACK technique suggested by Pichel *et al* [82] outperforms BELLPACK [80], with a cogent speed-up of $2.6\times$.

To improve the coalesced memory access in the COO format, the sliced COO (SCOO) format was suggested by Dang and Schmidt [83]. The GPU instance of SpMV based on this format utilizes the shared memory to store the partial results. The SCOO format with respect to the reference model matrix is depicted in figure 5-6. The slice size is computed such that partial products can fit into the shared memory. This method outperforms the COO, CSR and HYB methods from the CUSPARSE library. The authors also scaled this method on a multi-GPU platform [84]. To reduce the zero padding overheads of the ELLPACK format, Monakov *et al* [85] proposed the sliced-ELLPACK storage format shown in figure 5-8. Rows of the input matrix are divided into slices of size s . If the number of matrix rows is not divisible by s , extra zero rows are added at the end. The size of s is smaller; hence, the overheads are negligible compared with the baseline ELLPACK format. When the slice size is 1, the sliced-ELLPACK format is similar to CSR format, and if the slice size is equal to the number of rows, the format functions in a manner identical to that of the ELLPACK method. Note that slicing reduces zero padding overheads but does not remove them completely. The reordering of rows can be performed to mitigate the zero padding overheads. However, re-ordering may lead to decreased data locality, which must be addressed via a trade-off. An upgraded version of sliced-ELLPACK with the sorting technique was proposed by Kreutzer *et al* [86],

referred to as SELL- $C - \sigma$, where the C parameter determines the slice size and σ is defined as the *sorting parameter*. The *sorting parameter* σ suggests the number of rows of the matrix that need to be sorted locally. The parameter σ is selected such that it maximizes the occupancy on a GPU. Multiple threads are assigned per row of the matrix to compute the dot product. This achieves coalesced memory access as well as fine-grained data decomposition. A slight modification to this method was proposed by Antz *et al* [87]. They suggested applying zero padding to rows to make the number of entries in the rows of a matrix as a multiple of the number of threads.

Until now, the basic and some derived sparse matrix storage formats and their respective SpMV strategies, with their advantages and disadvantages, have been discussed. Recently, Filippone *et al* [88] proposed a constructive survey on sparse matrix storage formats and their respective SpMV strategies on a GPU as of 2016. They surveyed nearly 71 matrix storage formats. They provided the progress of research in the direction of each matrix storage format, as discussed earlier, with their trade-offs. The strategies proposed over a decade, with their advantages, disadvantages, pitfalls and lists of solutions, were organized excellently in a hierarchical manner.

Gao *et al* [89] recently proposed an extensive profile-based optimization model for SpMV in the PCG algorithm for a multi-GPU platform. The matrix is divided among the multiple GPUs, and the computations are performed independently. They proposed a device- and matrix-type-independent algorithm that accurately determines the execution time for the different SpMV kernels. They also provided a framework that automatically determines the optimal storage format and SpMV strategy with respect to the particular GPU. This model is constructed only once as part of the preprocessing on every GPU. Gao *et al* [90] extended the aforementioned strategy further for every operation in the CG. Their proposed strategy is extensible and can be applied to any system of linear equations generated by the FEM on a multi-GPU platform.

One of the shortcomings of CSR-based SpMV is the imbalanced workload among the threads for an irregular sparsity pattern. This leads to a waiting overhead for other thread warps having less work; hence, the computing power of a GPU cannot be utilized effectively. Flegar and Quintana-Ortí [91] proposed a balanced CSR method for irregular sparse matrices referred to as CSR-I. The storage format is similar to that of CSR, with the addition of a dimension vector. Each thread has an equal work distribution, and a single thread warp can be applied to one or more rows of the matrix. Their proposed strategy is limited to the arrowhead matrix, where non-zero entries are accumulated near the diagonal and first row and column of the matrix. Recently, Merrill and Garland [92] proposed a merging technique to improve the performance of CSR-based SpMV. This strategy does not require any preprocessing and storage conversion overheads. The proposed strategy effectively balances the workload among the

threads and outperforms all current CSR-based implementations. For smaller matrices, the merging strategy fails to accelerate the performance as achieved using *vector*-CSR. Currently, CPUs with multiple cores on a single socket are being developed. When the SpMV kernel is launched, the CPU acts a controller, and the whole computing power of the CPU is not utilized. To utilize the GPU and CPU computing power, Yang *et al* [93] proposed a heterogeneous computing model for SpMV that uses a hybrid storage format (COO + ELL). The matrix is divided into two submatrices, one for GPU computations and the other for CPUs using a distribution function. The GPU submatrix is stored in ELL format, and the CPU submatrix is stored in COO format. The time required to compute the SpMV is the maximum of the CPU and GPU times.

Multi-GPU strategies for solving the CG method [48, 94, 95] divide the disjoint SpMV computations on multiple GPUs. An obvious data decomposition technique is to divide the global matrix rows into chunks and assign each chunk to separate GPUs. Each GPU will perform the partial multiplication independently. Finally, results will be gathered on a CPU. Though this strategy is simpler to implement, it comes with two severe bottlenecks: data transfer in each iteration and inter-GPU synchronization overheads. The results obtained by Meng *et al* [48] show that the performance of the PCG is good on a multi-GPU platform but does not scale linearly with respect to the number of GPUs because of the aforementioned bottlenecks. Recently, Lin and Xie [94] proposed a Jacobi preconditioned CG on a multi-GPU cluster, where each cluster node contains one or more GPUs. First, the global matrix A is divided among the cluster nodes equally, and decomposition is again performed for each GPU at each cluster node. Similarly, the multiplying input vector is divided among the GPUs. Lin and Xie adopted the ELLPACK-R [88] sparse matrix storage format, which is a tailored version of the ELLPACK format. Each GPU performs partial SpMV to generate the partial result vector. To compute the SpMV, each GPU needs its own partitioned input vector and that from other GPUs to complete the partial results; hence, inter-GPU communication is required. They proposed a reordering strategy to reduce the bandwidth of the global matrix so that communication between GPUs is minimized. The analysis is performed on a banded diagonal matrix. Their proposed strategy achieves a speed-up of approximately $100\times$.

4. Some extraordinary strategies favourable to parallelization of FEM on GPU

In contrast with the earlier traditional FEM method, which performs assembly, the AF technique bypasses the generation of a global coefficient matrix and solves the system of linear equations at the element level. The system of linear equations from Eq. (3) is replaced by Eq. (14):

$$Ax = \sum_{e=1}^N M_e^T A_{(e)} M_e x = \sum_{e=1}^N M_e^T A_e x_{(e)} = C(A_e x_{(e)}) \quad (14)$$

where $A_{(e)}$ is the local e^{th} element matrix and $x_{(e)}$ and $b_{(e)}$ are the corresponding unknown and known element level vectors, respectively. M_e denotes the local-to-global node number mapping matrix, and C is an assembly operator. Figure 6 shows the assembly and assembly FEM computations for a 1D domain, which is discretized into two elements. After the generation of local elemental matrices, there are two choices. SpMV is carried out in the CG if *assembly* is performed; otherwise, local dense matrix-vector products (MvPs) are calculated. As each dense MvP in the AF method is an independent task, a single thread can be applied per MvP on the GPU. As depicted in figure 6, node 2 is common between element 1 and element 2. The race conditions arise while updating location y_2 in the result vector. For such parallel implementation, synchronization is needed while updating the result vector y . The AF strategy reduces the memory requirement and introduces higher degrees of parallelism compared with the traditional FEM. Unlike SpMV, the smaller dense MvPs have regular-sized elemental matrices and hence a regular memory access pattern. On a GPU, the elemental dense MvPs are calculated either at the element level, referred to as element-by-element (EbE), the node level, referred to as node-by-node (NbN), or the DoF level, referred to as DoF-by-DoF (DbD). For the EbE method, the SpMV from step 3 of Algorithm 1 and Algorithm 2 is replaced with

$$\mathbf{A}(A_e d_e) = \mathbf{A}(y_e) \quad (15)$$

where d_e is an elemental input vector and y_e is an elemental result vector. Assembly operator $\mathbf{A}$ denotes the transition from local to global node/DoFs numbers, i.e., $\mathbf{A}$ maps local elemental matrices to global matrix. It is important to note that the result vector y of SpMV is equivalent to the assembled vector of the elementary MvPs given by Eq. (15). Equation (15) can be further decomposed at the DoF/node level [96] as follows:

$$y^{(u)} = \sum_{e \in \varepsilon^{(u)}} A_e (I_e^{-1}, :) d_e \quad (16)$$

where I_e^{-1} is a local-to-global node number mapping operator and $\varepsilon^{(u)}$ denotes the set of elements attached to node/DoF u . As most of the nodes are common for two or more elements, parallelization of Eq. (15) results in race conditions while updating the resulting y vector. Meanwhile, in Eq. (16), race conditions are not encountered, as each thread is assigned to compute all elemental contributions at the respective node/DoF. The EbE-FEM [12, 13] is a very old technique suggested for computers with low memory. The first parallel implementation of the AF method at the element level (using Eq. (15)) on a GPU

was proposed by Kiss *et al* [97]. They also scaled this method on two GPUs to a single cluster node. A single thread is responsible for computing one elemental MvP, as given in Eq. (15). Graph colouring can be performed by considering nodes or elements as the vertices of a graph. Kiss *et al* [97] adopted an element-level graph colouring technique to avoid the race conditions. Elements that do not share common nodes are assigned the same colour, whereas neighbouring elements are assigned different colours. Thus, all elements are divided into distinct sets of colours that do not carry any dependence. A separate kernel is launched for each coloured set c in sequence, as shown in Eq. (17):

$$y^{(c)} = \mathbf{A}_{e \in \varepsilon^{(c)}} A_e d_e \quad (17)$$

where $\varepsilon^{(c)}$ denotes a set of elements in the coloured set c . This is how synchronization is achieved by launching separate kernels for each coloured set. In the same year, Fernández *et al* [98] proposed a GPU-based implementation of the AF-EbE approach using a Jacobi solver. This approach also scales effectively on a GPU, but its performance is limited because of the poor convergence of the Jacobi solver. The thread per DoF/node given in Eq. (16) avoids the synchronization overheads but suffers from an unbalanced load per thread. The number of computations per thread depends on the degree of the corresponding node (number of elements connected to a node). For unstructured discretization, the degree of each node may vary substantially. This causes other thread warps to wait for heavily loaded warps. While updating the global result vector y , the synchronization can be achieved either by graph colouring or by using the CUDA *atomics*. The CUDA *atomic* operations guarantee the correctness of the solution by causing other threads accessing the same memory locations to wait until the read/write operation is fully completed. The CUDA *atomics* serializes the global memory access, and graph colouring introduces synchronization overheads of the multiple kernels launched for each coloured set. Martínez-Frutos *et al* [96] demonstrated the performance of node-based colouring as well as that of element-based colouring. They observed that node-based colouring balances the workload among the threads better than element-based colouring. They also concluded that for the current generation of GPUs, the colouring technique does not outperform the atomic-based implementations for the AF-FEM. To avoid these limitations of colouring, atomic operations and an unbalanced load distribution, Martínez-Frutos *et al* [96] proposed a two-phase kernel. In the first phase, partial products from Eq. (16) are stored in an intermediate array $\bar{y}$; in the second phase, the contributions of elements at each node are gathered, as shown in Eqs. (18) and (19). The number of threads launched in the EbE method from Eq. (16) is equal to the total number of elements. For the DbD method, from Eq. (17), the number of threads required to compute the MvPs is equal to the

Figure 6. Assembly and assembly-free finite element computations.

total number of DoFs (N_{DoF}). For the two-phase approach (see Eqs. (18) and (19)), two separate kernels are launched. The first kernel launches $n_e \times \varepsilon$ threads, where n_e denotes the DoFs per element and ε represents the total number of elements. This DoF-based strategy introduces a fine-grained data decomposition compared with Eq. (16). The second kernel launches N_{DoF} threads to accumulate the partial results at the corresponding memory location in the result vector.

$$\bar{y}^{(u,e)} = A_e(I_e^{-1}(u), :)d_e, \quad (18)$$

$$y^{(u)} = \sum_{e \in e^{(u)}} \bar{y}^{(u,e)}. \quad (19)$$

A major shortcoming of the AF approach is the application of a preconditioner at the element level. Most of the preconditioners are generated from the global matrix A , but assembly-free methods do not generate the global matrix at all. Therefore, the generation and application of a

preconditioner at the element level is a challenging task. The Jacobi preconditioner is easy to construct and apply even at the element level; thus, most of the literature has adopted this preconditioner. Hughes *et al* [99] suggested a CG-based EbE technique, which uses a simple Jacobi preconditioner. Martínez-Frutos *et al* [96] also adopted the Jacobi preconditioner to achieve application at the DoF level. The Jacobi preconditioner is easy to construct, but its convergence rate is poor compared with that of the IF preconditioners. IF preconditioners have a greater convergence rate but are difficult to apply at the element level because they are generated from the global matrix. Recently, an inter-element assembly process was proposed by Yan *et al* [100] to create a Gauss–Seidel preconditioner for the EbE method. This preconditioner is constructed without generating the global matrix, and its convergence rate is higher than that of the Jacobi preconditioner. Instead of adding the contributions of all elements in the global matrix, the contributions are added locally in between each elemental matrix.

The major drawback of SpMV is uncoalesced memory access, which does not occur in the case of the AF method, where dense MvPs are computed. SpMV in the iterative CG solver is replaced by element-level dense MvPs, as shown in Eq. (15). Thus, memory-bound SpMV is converted into computationally intensive MvPs for which GPUs are intended. Furthermore, each elemental matrix is of the same size and induces regular memory access, because of which coalesced memory access can be achieved. SpMV based on the blocked CSR format was compared to assembly-free MvPs for tetrahedral elements by Akbariyeh *et al* [101]. They observed that the performance of the AF approach is much lower due to the limited register availability. The performance depends on the available GPU fast local memory. For higher-order elements [7], the performance is poor compared with that of the lower-order FEM solver. This is because the data related to elements do not fit into the shared memory. As discussed in the discretization section (section 2.1), the FG method does not require the generation of all elemental matrices. Martínez-Frutos and Herrero-Pérez [102] proposed the parallelization of the FG method applied to the 3D domain. The sizes of all elements are the same; as a consequence, entries in the local matrices are also the same, provided that domain-specific properties are homogeneous throughout the domain. As the local matrices generated for inner elements are the same, a common elemental matrix is stored in the constant memory, and the multiplying input vector, into the shared memory. The elemental matrices are computed on-the-fly in each iteration of the CG algorithm. This strategy reduces not only the memory space but also the more costly global memory transactions dramatically. Furthermore, the multiplying vector and node number mapping arrays are stored in the shared memory, which also reduces the global memory footprints.

Recently, parallelization of topology optimization [103] in the FEM has received interest in the HPC community [104–107] for elasticity problems. Topology optimization is used to find an optimal design model for particular structures. We start with a rough model, without any concern for the internal structure. After this, the whole domain undergoes meshing, with domain-specific properties being defined and boundary conditions being applied as in the traditional FEM. Then, FEM analysis is initialized to obtain the first base line of the optimization. This base line gives the idea of the design structure. Following this, the optimization process is applied to define the design area and non-design area. To perform this analysis, the whole domain has to undergo all FEA steps, which is a very computation-intensive task. Over the last two years, Ram and Sharma [106] and Martínez-Frutos and Herrero-Pérez [107] have successfully exploited the computing power of GPUs to accelerate the topology optimization strategy. Martínez-Frutos *et al* [104] and Martínez-Frutos and Herrero-Pérez [107] adopted the FG-FEM to accelerate the topology optimization in structural analysis.

5. Pitfalls and trade-offs

Apparently, FEM processing favours GPU-based parallelization because of the intrinsic SIMD-type computations. Nevertheless, the following pitfalls and trade-offs are encountered while deploying FEM applications on a GPU.

1. **Discretization:** Discretization is the first step, and all subsequent parallel optimization choices are based on this step. There is a trade-off in selection between the number of elements and order of approximation. To increase accuracy, two strategies are used: *h*-adaptive and *p*-adaptive. The *h*-adaptive strategy improves accuracy by increasing the number of elements (by reducing the size of elements), while the *p*-adaptive strategy improves accuracy by increasing the polynomial order. Many researchers are skeptical about the *hp*-adaptive refinement, as it increases the number of computations and execution time. However, these strategies provide a higher convergence rate. The fixed-grid method dramatically reduces the memory consumption but it is applicable over a domain having homogeneous domain-specific properties. For an inhomogeneous domain, such as human bones [108], the domain-specific properties vary continuously across elements. Therefore, the elemental matrices generated may be different for every element. Furthermore, in some scenarios, we need the density of elements to be more concentrated to obtain a precise solution at that point. Therefore, the fixed-grid method is not applicable in these situations.
2. **Numerical integration:** Numerical integration is an embarrassingly parallel application, but the order of approximation affects the performance. Higher-order elements increase the accuracy and have a higher arithmetic intensity but cannot fit into the limited-size fast local memory of a GPU due to the increased memory requirement per element. In most scenarios, the numerical integration process is limited by the available hardware resources. Lower-order elements have very poor arithmetic intensity and may lead to underutilization of the GPU's computing power.
3. **Assembly:** The assembly procedure requires synchronization (atomic operations) to avoid the race conditions due to parallel updating at the same memory location between the common nodes of elements. To alleviate the drawback of CUDA *atomics*, strategies such as graph colouring and the use of a dependence graph can be carried out. An increase in the number of colours reduces the parallelism and increases the synchronization overheads. Fewer colours lead to underutilization of the GPU computing power. Determination of the optimal choice of colours is an NP-hard problem.
4. **CG solver:** Although operations in the CG solver are of SIMD type and favourable on a GPU, there are some problems related to SpMV and parallel preconditioning. Sparse matrix storage formats are application dependent,

and no storage format guarantees a good performance for all types of sparse matrices. Furthermore, the arithmetic intensity in SpMV is low; hence, the performance degrades due to the large number of memory transactions. The optimization strategy of SpMV on a GPU depends on the sparse pattern in the global matrix generated from the assembly process. IF preconditioners have a better convergence rate but are hostile for parallelization on a GPU due to output dependences in forward and backward substitution. IF preconditioners often reduce the throughput on a GPU. Direct preconditioners are easy to construct and parallelize but have a poor convergence rate. SAIPs introduce parallelism by converting the application of a preconditioner into SpMV, but they are not as robust as IF preconditioners. However, preconditioners are not generic; rather, they are application dependent.

5. **Precision:** Use of SP increases throughput and reduces the time for execution, but DP arithmetic leads to better accuracy in iterative solvers. Mixed precision alleviates these issues, but to preserve both types of precision, extra memory storage space is required.
6. **Underlying hardware architecture:** The selection of parallel optimization techniques depends on the underlying hardware parameters, such as the GPU series, available shared memory, maximum number of threads that can be launched, number of streaming multiprocessors, optimum thread block size and grid size. Although OpenCL provides a platform on which the heterogeneous computing power of GPUs from different vendors can be utilized, in many scenarios, the performance is slow compared with that of the CUDA. The hardware architectures are continuously upgraded in terms of available memory, computing power, FLOPs, number of registers, etc. Therefore, the optimization strategies may vary according to the new generation of hardware architectures.
7. **Multi-GPU speedup:** Multi-GPU strategies are adopted to scale and accelerate the different steps of the FEM, but the speed-up achieved is below linear with respect to the number of GPUs. Multi-GPU implementations introduce synchronization and data transfer overheads. Again, inter-GPU communication is much slower compared with the memory bandwidth within a GPU. This idle time can dominate the overall performance for smaller problems. Different decomposition techniques are used to convert a larger system into several smaller parts such that they can fit in the onboard memories of CPUs and GPUs. This introduces massive parallelization, but partial results may require more costly synchronization. Decomposed data segments are sent to the GPUs in chunks, which increases the data transfer time. This time is considerable compared to the overall execution time. The major overheads in multi-GPU approaches are inter-node CPU-to-CPU communications, intra-node CPU-to-GPU (within the same cluster

node) communications, intra-node GPU-to-GPU communications, inter-node GPU-to-GPU communications, application-dependent communications and system-dependent communications. Nevertheless, multi-GPU implementations achieve better performances compared with those of single GPU implementations provided that the applications are sufficiently large.

8. **Assembly-free solver:** Though AF solvers alleviate the storage issues, the accuracy of the solutions is prone to degrade while scaling. For fewer elements, the accuracy is good, but for a myriad of elements, the accuracy is not as good as that of the traditional assembly-based method. The convergence of the AF methods is slower than that of the traditional FEM [109]. The colouring scheme is used to avoid the race conditions but induces two drawbacks: having fewer colours reduces parallelism, while having numerous colours increases the synchronization overheads between kernels.

6. Conclusions

A progressive review of each step in the FEM pipeline with the use of GPU-based parallel computing has been presented. The introduction of general purpose computing on GPUs improves the performances of FEM simulations. FEM computations are of SIMD type and are easy to parallelize, provided that a proper analysis of the dependences is carried out. Apparently, to achieve maximum speed-up and accuracy, selection of the hardware, programming language and data structure is prominent. SpMV in the solver step is the most dominating computation in the entire FEM pipeline; numerical integration and assembly also dominate for higher-order approximations. A significant speed-up is achieved in the numerical integration and assembly processes, but the solver process lags due to the memory-bound SpMV. The iterative solver CG has a lower time and space complexity compared with those of direct solvers and exhibits parallelism on a GPU. Selection of the optimal sparse matrix storage format is based on the GPU fast local memory and mainly the sparsity pattern in the global matrix. To increase the convergence rate of the solver, preconditioning techniques can be used, provided that the preconditioners are easy to construct, apply and parallelize. The application of IF preconditioners on a GPU is not favourable due to the forward and backward substitutions, which are recursive and highly dependent in nature. To implement an effective parallelization strategy, a trade-off between various preconditioning techniques, different sparse matrix storage formats, load mapping per thread, etc. needs to be considered. Acceleration of the FEM computations on a single GPU is now approaching saturation. Currently, multiple GPUs can be placed on a single board. Thus, the multi-GPU approach is an emerging trend in the parallelization of the FEM. Furthermore, heterogeneous

computing is a universal concern for accelerating FEM computations. To enhance the performance further, a hybrid computing method based on the CUDA and message passing interface (MPI) or the CUDA and open multiprocessing (OpenMP) can be used to accelerate the performances of FEM simulations via heterogeneous computing. To increase portability, OpenCL-based implementation may be the best choice.

References

- [1] Zienkiewicz O C, Taylor R L and Nithiarasu P 2000 *The finite element method: solid mechanics*, vol. 2. Oxford: Butterworth-heinemann
- [2] Singh I V, Mishra B K, Brahmanekar M, Bhasin V, Sharma K and Khan I A 2014 Numerical simulations of 3-d cracks using coupled EFGM and FEM. *Int. J. Comput. Methods Eng. Sci. Mech.* 15(3): 227–231
- [3] Jin J M 2015 *The finite element method in electromagnetics*, 3rd ed. New York: John Wiley & Sons
- [4] Moratal D 2012 *Finite element analysis-from biomedical applications to industrial development*. London: InTech
- [5] Argyris J 1954 and 1955 Energy theorems and structural analysis. *Aircraft Engineering* re-printed 1990 London: Butterworth's Scientific Publications
- [6] Clough W R 1960 The finite element method in plane stress analysis. In: *Proceedings of the 2nd Conference on Electronic Computation*, A.S.C.E. Structural Division, Pittsburgh, Pennsylvania
- [7] Banaś K, Płaszewski P and Macioł P 2014 Numerical integration on GPUs for higher order finite elements. *Comput. Math. Appl.* 67(6): 1319–1344
- [8] Komatitsch D, Michéa D and Erlebacher G 2009 Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *J. Parallel Distrib. Comput.* 69(5): 451–460
- [9] Dongarra J *Survey of sparse matrix storage formats*. www.netlib.org/utk/papers/templates/node90.html (visited 10th May 2017)
- [10] Bell N and Garland M 2008 *Efficient sparse matrix-vector multiplication on CUDA*. Nvidia Technical Report NVR-2008-004, Nvidia Corporation
- [11] Barrett R, Berry M, Chan T F, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C and Van der Vorst H 1994 *Templates for the solution of linear systems: building blocks for iterative methods*, 2nd ed. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania
- [12] Carey G F and Jiang B 1986 Element-by-element linear and nonlinear solution schemes. *Commun. Appl. Numer. Methods* 2(2): 145–153
- [13] Carey G F, Barragy E, Mclay R and Sharma M 1988 Element-by-element vector and parallel computations. *Commun. Appl. Numer. Methods* 4(3): 299–307
- [14] Nickolls J and Kirk D 2009 Graphics and computing GPUs. In: Patterson D A and Hennessy J L *Computer organization and design*, 4th ed. Appendix A: 1–77
- [15] NVIDIA CUDA 2007 *Compute unified device architecture programming guide*. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> (visited 23rd September 2017)
- [16] Owens J D, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn A E and Purcell T J 2007 A survey of general-purpose computation on graphics hardware. *Comput. Graph. Forum* 26(1): 80–113
- [17] Liu Y, Jiao S, Wu W and De S 2008 GPU accelerated fast FEM deformation simulation. In: *Proceedings of the Asia Pacific Conference on Circuits and Systems, APCCAS 2008, IEEE Macao*, pp. 606–609
- [18] Kákay A, Westphal E and Hertel R 2010 Speedup of FEM micromagnetic simulations with graphical processing units. *IEEE Trans. Magn.* 46(6): 2303–2306
- [19] Brodtkorb A R, Hagen T R and Sætra M L 2013 Graphics processing unit (GPU) programming strategies and trends in GPU computing. *J. Parallel Distrib. Comput.* 73(1): 4–13
- [20] Hoole S R H, Karthik V U, Sivasuthan S, Rahunathan A, Tyagarajan R S and Jayakumar P 2015 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.* 47(3): 607–627
- [21] Sanders J and Kandrot E 2010 *CUDA by example: an introduction to general-purpose GPU programming*. Massachusetts: Addison-Wesley Professional
- [22] Ho-Le K 1988 Finite element mesh generation methods: a review and classification. *Comput. Aided Des.* 20(1): 27–38
- [23] Sivasuthan S, Karthik V U, Jayakumar P, Thyagarajan R S, Udpa L and Hoole S R H 2015 A script-based, parameterized finite element mesh for design and NDE on a GPU. *IETE Tech. Rev.* 32(2): 94–103
- [24] Reddy J N 1993 *An introduction to the finite element method*, 2nd ed. New York: McGraw-Hill, vol. 2, no. 2.2
- [25] Garcia-Ruiz M J and Steven G P 1999 Fixed grid finite elements in elasticity problems. *Eng. Comput.* 16(2): 145–164
- [26] Kružel F and Banaś K 2013 Vectorized OpenCL implementation of numerical integration for higher order finite elements. *Comput. Math. Appl.* 66(10): 2030–2044
- [27] Solin P, Segeth K and Dolezel I 2003 *Higher-order finite element methods*. Boca Raton: Chapman & Hall, CRC Press
- [28] Macioł P, Płaszewski P and Banaś K 2010 3D finite element numerical integration on GPUs. *Procedia Comput. Sci.* 1(1): 1093–1100
- [29] Filipovič J, Peterlík I and Fousek J 2009 GPU acceleration of equations assembly in finite elements method—preliminary results. In: *Proceedings of the Symposium on Application Accelerators in HPC (SAAHPC)*
- [30] Dziekonski A, Sypek P, Lamecki A and Mrozowski M 2012 Accuracy, memory, and speed strategies in GPU-based finite-element matrix-generation. *IEEE Antennas Wirel. Propag. Lett.* 11: 1346–1349
- [31] Dziekonski A, Sypek P, Lamecki A and Mrozowski M 2013 Generation of large finite element matrices on multiple graphics processors. *Int. J. Numer. Methods Eng.* 94(2): 204–220
- [32] Nvidia Corporation 2008 Cublas library. Version 2.0, NVIDIA, Santa Clara, California
- [33] Dziekonski A, Sypek P, Lamecki A and Mrozowski M 2012 Finite element matrix generation on a GPU. *Prog. Electro-magn. Res.* 128: 249–265

- [34] Munshi A, Gaster B R, Mattson T G, Fung J and Ginsburg D 2011 OpenCL programming guide. London: Pearson Education
- [35] Banaś K, Kružel F and Bielański J 2016 Finite element numerical integration for first order approximations on multi- and many-core architectures. *Comput. Methods Appl. Mech. Eng.* 305: 827–848
- [36] Woźniak M 2015 Fast GPU integration algorithm for isogeometric finite element method solvers using task dependency graphs. *J. Comput. Sci.* 11: 145–152
- [37] Mamza J, Makyla P, Dziekonski A, Lamecki A and Mrozowski M 2012 Multi-core and multiprocessor implementation of numerical integration in Finite Element Method. In: *Proceedings of the 19th International Conference on Microwaves, Radar & Wireless Communications*, IEEE, Warsaw, vol. 2, pp. 457–461
- [38] Knepley M G and Terrel A R 2013 Finite element integration on GPUs. *ACM Trans. Math. Softw. (TOMS)* 39(2): 10:1–13
- [39] Cecka C, Lew A and Darve E 2010 Introduction to assembly of finite element methods on graphics processors. *IOP Conf. Ser. Mater. Sci. Eng.* 10(1): 012009
- [40] Iwashita T and Shimasaki M 2002 Algebraic multicolor ordering for parallelized ICCG solver in finite-element analyses. *IEEE Trans. Magn.* 38(2): 429–432
- [41] Iwashita T and Shimasaki M 2003 Algebraic block red-black ordering method for parallelized ICCG solver with fast convergence and low communication costs. *IEEE Trans. Magn.* 39(3): 1713–1716
- [42] Fu Z, Lewis T J, Kirby R M and Whitaker R T 2014 Architecting the finite element method pipeline for the GPU. *J. Comput. Appl. Math.* 257: 195–211
- [43] Cecka C, Lew A J and Darve E 2011 Assembly of finite element methods on graphics processors. *Int. J. Numer. Methods Eng.* 85(5): 640–669
- [44] Markall G R, Ham D A and Kelly Paul H J 2010 Towards generating optimized finite element solvers for GPUs from high-level specifications. *Procedia Comput. Sci.* 1(1): 1815–1823
- [45] Markall G R, Slemmer A, Ham D A, Kelly P H J, Cantwell C D and Sherwin S J 2013 Finite element assembly strategies on multicore and manycore architectures. *Int. J. Numer. Methods Fluids* 71(1): 80–97
- [46] Sanfui S and Sharma D 2017 A two-kernel based strategy for performing assembly in FEA on the graphic processing unit. In: *Proceedings of the IEEE International Conference on Advances in Mechanical, Industrial, Automation and Management Systems (AMIAMS)*, pp. 1–9
- [47] Cecka C, Lew A and Darve E 2011 Application of assembly of finite element methods on graphics processors for real-time elastodynamics. In: *GPU computing gems*, Jade ed. Massachusetts: Morgan Kaufmann, chapter 16, pp. 187–205
- [48] Meng H T, Nie B L, Wong S, Macon C and Jin J M 2014 GPU accelerated finite-element computation for electromagnetic analysis. *IEEE Antennas Propag. Mag.* 56(2): 39–62
- [49] Reguly I Z and Giles M B 2015 Finite element algorithms and data structures on graphical processing units. *Int. J. Parallel Program.* 43(2): 203–239
- [50] Dziekonski A, Sypek P, Lamecki A and Mrozowski M 2014 GPU-accelerated finite-element matrix generation for lossless, lossy, and tensor media. *IEEE Antennas Propag. Mag.* 56(5): 186–197
- [51] Dziekonski A, Sypek P, Lamecki A and Mrozowski M 2017 Communication and load balancing optimization for finite element electromagnetic simulations using multi-GPU workstation. *IEEE Trans. Microw. Theory Tech.* 65(8): 2661–2671
- [52] Logg A, Mardal M A and Wells G N 2012 *Automated solution of differential equations by the finite element method: the FEniCS book*, vol. 84. New York–Heidelberg–Dordrecht–London: Springer
- [53] Dupont T, Hoffman J, Jansson J, Johnson C, Kirby Robert C, Knepley M, Larson M, Logg A and Scott R 2003 *The fenics project*. Tech. Rep. 200321, Chalmers Finite Element Center Preprint Series
- [54] Luporini F, Varbanescu A L, Rathgeber F, Bercea G T, Ramanujam J, Ham D A and Kelly P H J 2014 COFFEE: an optimizing compiler for finite element local assembly. arXiv preprint [arXiv:1407.0904](https://arxiv.org/abs/1407.0904)
- [55] Shewchuk J R 1994 *An introduction to the conjugate gradient method without the agonizing pain*. Technical Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania
- [56] Itu L M, Suciu C, Moldoveanu F and Postelnicu A 2011 Comparison of single and double floating point precision performance for Tesla architecture GPUs. *Bull. Transilv. Univ. Brasov Ser. I Eng. Sci.* 4(53): 131–138
- [57] Göddeke D, Strzodka R and Turek R 2007 Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in FEM simulations. *Int. J. Parallel Emerg. Distrib. Syst.* 22(4): 221–256
- [58] Baboulin M, Buttari A, Dongarra J, Kurzak J, Langou J, Langou Julien, Luszczek P and Tomov S 2009 Accelerating scientific computations with mixed precision algorithms. *Comput. Phys. Commun.* 180(12): 2526–2533
- [59] Buttari A, Dongarra J, Kurzak J, Langou Julie, Langou Julien, Luszczek P and Tomov S 2006 Exploiting mixed precision floating point hardware in scientific computations. In: *Proceedings of the High Performance Computing Workshop*, pp. 19–36
- [60] Göddeke D, Strzodka R and Turek S 2005 Accelerating double precision FEM simulations with GPUs. In: *Proceedings of ASIM 18th Symposium on Simulation Technique*
- [61] Cosgrove J D F, Díaz J C and Griewank A 1992 Approximate inverse preconditionings for sparse linear systems. *Int. J. Comput. Math.* 44(1–4): 91–110
- [62] Li R and Saad Y 2013 GPU-accelerated preconditioned iterative linear solvers. *J. Supercomput.* 63(2): 443–466
- [63] Naumov M, Chien L S, Vandermersch P and Kapasi U 2010 Cuspars library. Presented at: *GPU Technology Conference San Jose*
- [64] Wang E, Zhang Q, Shen B, Zhang G, Lu X, Wu Q and Wang Y 2014 Intel math kernel library. In: *High-performance computing on the Intel®Xeon Phi™*. Springer International Publishing, pp. 167–188
- [65] Naumov M 2011 *Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS*. Nvidia Technical Report and White Paper
- [66] Fialko S Y and Zeglen F 2016 Preconditioned conjugate gradient method for solution of large finite element problems on CPU and GPU. *J. Telecommun. Inf. Technol.* nr 2: 26–33

- [67] Gao J, Liang R and Wang J 2014 Research on the conjugate gradient algorithm with a modified incomplete Cholesky preconditioner on GPU. *J. Parallel Distrib. Comput.* 74(2): 2088–2098
- [68] Benzi M, Meyer C D and Tůma M 1996 A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Sci. Comput.* 17(5): 1135–1149
- [69] Grote M J and Huckle T 1997 Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.* 18(3): 838–853
- [70] Ament M, Knittel G, Weiskopf D and Straßer W 2010 A parallel preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform. In: *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, IEEE, pp. 583–592
- [71] Helfenstein R and Koko J 2012 Parallel preconditioned conjugate gradient algorithm on GPU. *J. Comput. Appl. Math.* 236(15): 3584–3590
- [72] Gravvanis G A 2002 Explicit approximate inverse preconditioning techniques. *Arch. Comput. Methods Eng.* 9(4): 371–402
- [73] Gravvanis G A, Filelis-Papadopoulos C K and Giannoutakis K M 2012 Solving finite difference linear systems on GPUs: CUDA based parallel explicit preconditioned biconjugate conjugate gradient type methods. *J. Supercomput.* 61(3): 590–604
- [74] Cuthill E and McKee J 1972 Several strategies for reducing the bandwidth of matrices. In: Rose D J and Willoughby R A (Eds.) *Sparse matrices and their applications*. New York: Springer, pp. 157–166
- [75] Fujiwara K, Nakata T and Fusayasu H 1993 Acceleration of convergence characteristic of the ICCG method. *IEEE Trans. Magn.* 29(2): 1958–1961
- [76] Camargos A F P De, Silva V C, Guichon J M and Munier G 2014 Efficient parallel preconditioned conjugate gradient solver on GPU for FE modeling of electromagnetic fields in highly dissipative media. *IEEE Trans. Magn.* 50(2): 569–572
- [77] Bernaschi M, Bisson M, Fantozzi C and Janna C 2016 A factored sparse approximate inverse preconditioned conjugate gradient solver on graphics processing units. *SIAM J. Sci. Comput.* 38(1): C53–C72
- [78] Bell N and Garland M 2017 <https://code.google.com/archive/p/cusp-library/downloads> (visited 23rd June)
- [79] Monakov A and Avetisyan A 2009 Implementing blocked sparse matrix–vector multiplication on NVIDIA GPUs. *Embedded computer systems: architectures, modeling, and simulation*, pp. 289–297
- [80] Choi J W, Singh A and Vuduc R W 2010 Model-driven autotuning of sparse matrix–vector multiply on GPUs. *ACM Sigplan Not.* 45(5): 115–126
- [81] Vázquez F, Fernández J J and Garzón E M 2011 A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurr. Comput. Pract. Exp.* 23(8): 815–826
- [82] Pichel J C, Rivera F F, Fernández M and Rodríguez A 2012 Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs. *Microprocess. Microsyst.* 36(2): 65–77
- [83] Dang H V and Schmidt B 2012 The sliced COO format for sparse matrix–vector multiplication on CUDA-enabled GPUs. *Procedia Comput. Sci.* 9: 57–66
- [84] Dang H V and Schmidt B 2013 CUDA-enabled sparse matrix–vector multiplication on GPUs using atomic operations. *Parallel Comput.* 39(11): 737–750
- [85] Monakov A, Lokhmotov A and Avetisyan A 2010 Automatically tuning sparse matrix–vector multiplication for GPU architectures. *HiPEAC Proceedings, Lecture Notes in Computer Science* 5952, pp. 111–125
- [86] Kreutzer M, Hager G, Wellein G, Fehske H and Bishop A R 2014 A unified sparse matrix data format for efficient general sparse matrix–vector multiplication on modern processors with wide SIMD units. *SIAM J. Sci. Comput.* 36(5): C401–C423
- [87] Anzt H, Tomov S and Dongarra J *Implementing a sparse matrix vector product for the SELL-C/SELL-C- formats on NVIDIA GPUs*. University of Tennessee, Tech. Rep., UT-EECS-14-727
- [88] Filippone S, Cardellini V, Barbieri D and Fanfarillo A 2017 Sparse matrix–vector multiplication on GPGPUs. *ACM Trans. Math. Softw. (TOMS)* 43(4): 30
- [89] Gao J, Wang Y and Wang J 2017 A novel multigraphics processing unit parallel optimization framework for the sparse matrixvector multiplication. *Concurr. Comput. Pract. Exp.* 29(5): e3936
- [90] Gao J, Zhou Y, He G and Xia Y 2017 A multi-GPU parallel optimization model for the preconditioned conjugate gradient algorithm. *Parallel Comput.* 63: 1–16
- [91] Flegar G and Quintana-Ortí E S Balanced CSR sparse matrix–vector product on graphics processors. In: *Proceedings of the European Conference on Parallel Processing*. Cham: Springer, pp. 697–709
- [92] Merrill D and Garland M 2016 Merge-based parallel sparse matrix–vector multiplication. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, UT, Salt Lake City, pp. 678–689
- [93] Yang W, Li K and Li K 2017 A hybrid computing method of SpMV on CPUGPU heterogeneous computing systems. *J. Parallel Distrib. Comput.* 104: 49–60
- [94] Lin S and Xie Z 2017 A Jacobi PCG solver for sparse linear systems on multi-GPU cluster. *J. Supercomput.* 73(1): 433–454
- [95] Cevahir A, Nukada A and Matsuoka S 2009 Fast conjugate gradients with multiple GPUs. In: *Proceedings of the International Conference on Computational Science*, LNCS 5544. Berlin–Heidelberg: Springer, pp. 893–903
- [96] Martínez-Frutos J, Martínez-Castejón P J and Herrero-Pérez D 2015 Fine-grained GPU implementation of assembly-free iterative solver for finite element problems. *Comput. Struct.* 157: 9–18
- [97] Kiss I, Gyimothy S and Badics Z 2012 Parallel realization of the element-by-element FEM technique by CUDA. *IEEE Trans. Magn.* 48(2): 507–510
- [98] Fernández D M, Dehnavi M M, Gross W J and Giannopoulos D 2012 Alternate parallel processing approach for FEM. *IEEE Trans. Magn.* 48(2): 399–402
- [99] Hughes T J R, Levit I and Winget J 1983 An element-by-element solution algorithm for problems of structural and solid mechanics. *Comput. Methods Appl. Mech. Eng.* 36(2): 241–254
- [100] Yan X, Han X, Wu D, Xie D, Bai B and Ren Z 2017 Research on preconditioned conjugate gradient method based on EBE-FEM and the application in electromagnetic field analysis. *IEEE Trans. Magn.* 53(6): 1–4

- [101] Akbaryeh A, Dennis B H, Wang B P and Lawrence K L 2015 Comparison of GPU-based parallel assembly and assembly-free sparse matrix vector multiplication for finite element analysis of three-dimensional structures. In: *Proceedings of the Fifteenth International Conference on Civil, Structural and Environmental Engineering Computing, Civil-Comp Press, Stirlingshire, Scotland*
- [102] Martínez-Frutos J and Herrero-Pérez D 2015 Efficient matrix-free GPU implementation of fixed grid finite element analysis. *Finite Elem. Anal. Des.* 104: 61–71
- [103] Bendsoe M P and Sigmund O 2004 Topology optimization theory, methods, and applications. Berlin–Heidelberg: Springer
- [104] Martínez-Frutos J, Martínez-Castejón P J and Herrero-Pérez D 2017 Efficient topology optimization using GPU computing with multilevel granularity. *Adv. Eng. Softw.* 106: 47–62
- [105] Martínez-Frutos J and Herrero-Pérez D 2017 GPU acceleration for evolutionary topology optimization of continuum structures using isosurfaces. *Comput. Struct.* 182: 119–136
- [106] Ram L and Sharma D 2017 Evolutionary and GPU computing for topology optimization of structures. *Swarm Evol. Comput.* 35: 1–13
- [107] Martínez-Frutos J and Herrero-Pérez D 2016 Large-scale robust topology optimization using multi-GPU systems. *Comput. Methods Appl. Mech. Eng.* 311: 393–414
- [108] Baca V, Horak Z, Mikulénka P and Dzupa V 2008 Comparison of an inhomogeneous orthotropic and isotropic material models used for FE analyses. *Med. Eng. Phys.* 30(7): 924–930
- [109] Cai Y, Li G and Wang H 2013 A parallel node-based solution scheme for implicit finite element method using GPU. *Procedia Eng.* 61: 318–324