GPU Acceleration for Evolutionary Topology Optimization of Continuum Structures Using Isosurfaces

Jesús Martínez-Frutos*, David Herrero-Pérez

Department of Structures and Construction, Technical University of Cartagena, Campus Muralla del Mar, 30202 Cartagena, Murcia, Spain

Abstract

Evolutionary topology optimization of three-dimensional continuum structures is a computationally demanding task in terms of memory consumption and processing time. This work aims to alleviate these constraints proposing a well-suited strategy for Graphics Processing Unit (GPU) computing. Such a proposal adopts a fine-grained GPU instance of matrix-free iterative solver for structural analysis and an efficient GPU implementation for isosurface extraction and volume fraction calculation. The performance of the solving stage is evaluated using two preconditioning techniques, including the comparison with the sparse-matrix CPU implementation. The proposal is evaluated using topology optimization problems for real-world applications.

Keywords: Evolutionary topology optimization, Isosurfaces, GPU computing, Large scale, Multigrid preconditioning

1. Introduction

Topology optimization aims at finding the optimal layout of material within a design domain for a given set of boundary conditions such that the resulting material distribution meets a set of performance targets [1]. Such a material

*Corresponding author

Email addresses: jesus.martinez@upct.es (Jesús Martínez-Frutos), david.herrero@upct.es (David Herrero-Pérez)

distribution is obtained without assuming any prior structural configuration. This provides a powerful tool to find the best conceptual design that fulfills the design requirements at the early stages of the structural design. Not to mention the great impact of the optimization of geometry and topology of structures on its performance. Since the early work of Bendsøe and Kikuchi [2], topology optimization has been successfully applied to a wide range of problems, such as stiffness maximization of structures [1], design of compliant mechanisms [3, 4], maximization of temperature diffusivity [5], and minimization of acoustic response [6, 7], to name but a few [8].

According to [9], topology optimization methods can be broadly classified into density-based methods [10, 11], level set methods [12], phase field methods [13, 14], topological derivative methods [15] and evolutionary approaches [16]. The variants of Evolutionary Structural Optimization (ESO) method [17, 18] and Bi-directional Evolutionary Structural Optimization (BESO) method [19, 20] are some of the approaches included in the last category. These optimization methods are based on heuristic rules, which include from simple hard-kill strategies (elements with lowest strain energy density are removed) to bi-directional schemes [21] where elements can be reintroduced if considered rewarding. Apart from intuition, such methods use standard adjoint gradient analysis and filtering techniques to stabilize algorithms and results [22].

Despite the great advances both in theory and practical application of topology optimization achieved in the past decade, the computational requirements of large-scale 3D problems still remain as a primary challenge [8]. This is due to some demanding tasks involved in the topology optimization pipeline, such as the use of iterative methods to solve large systems of equations, the computation of sensitivities, the structural boundary extraction and the elemental density update. Such tasks may increase meaningfully the computation time of the topology optimization process. High Performance Computing (HPC) is then needed to address the topology optimization process, normally making use of task-level parallel computing to reduce the computational time of computationally intensive tasks [23, 24, 25].

The use of Graphics Processing Units (GPUs) for non-graphics applications is rapidly growing in popularity. This is due to the high computing capacity of these graphics cards for Massively Parallel Processing (MPP), also known as Data-Level Parallelism (DLP) [26], at reasonable cost. GPU computing consists of the use of a GPU together with a CPU to accelerate compute intensive applications. This is not a simple goal since the programming skills to be able to fully utilize GPU hardware can be considered as an art that can take years to master [27]. In fact, there exist numerous problems that prevent the use of GPU computing for certain scientific applications, such as memory related problems and lack of data-level parallelism. The memory related problems include excessive global memory transactions, non-coalesced global loads and stores that degrade global memory bandwidth, and shared memory accesses inducing bank conflicts, to name but a few. The lack of data-level parallelism prevents the exploitation of Single Instruction Multiple Data (SIMD) parallel computation for which GPU architectures are designed. Therefore, the proper implementation of topology optimization methods using GPUs requires a suitable formulation and selection of techniques allowing making use of the potential acceleration of massive parallel architectures and preventing memory related problems, which constraint severely the GPU performance. The reader is referred to [27, 28] for comprehensive reviews of this research field.

The use of GPU devices to speedup computationally demanding tasks in the topology optimization pipeline has sparked a broad interest last years, giving rise to several studies. The early work of Wadbro and Berggren [29] showed how GPUs can be used to efficiently solve large topology optimization problems using a gradient-based optimality criterion method. This early work implements a Preconditioned Conjugate Gradient (PCG) method on GPU to solve high resolution finite element models arising in heat conduction topology optimization problems. The grain size of this GPU instance is at the element level. The lack of native double-precision support for early GPUs limited the GPU instance to single-precision format, which not ensures the convergence of the solver due to round-off errors. A nodal-wise assembly-free GPU implementa-

tion for the solver of the SIMP method is proposed by Schmidt and Schulz [30]. Applied to the minimization of the structural compliance problem, this GPU instance achieves significant speedups. Such a strategy is followed by Suresh [31] achieving speedups of one order of magnitude for the solving of the system of equations of elasticity. Challis et al. [32] also used a matrix-free GPU instance of PCG solver with the aim of increasing the tractable computational resolution of topology optimization problems using a discrete level-set method.

GPU computing using the sparse-matrix representation permits the efficient assembly and solving of the system of equations of elasticity [33]. However, a higher performance can be achieved exploiting the grid regularity and performing the operations "on-the-fly". The former permits to exploit data locality providing reduced memory access and making use of on-chip memory, which is much more efficient than global device memory. The latter avoids storing the matrix of coefficients explicitly in global device memory, which affects seriously the GPU performance. For these reasons, the matrix-free GPU implementations of topology optimization using regular grids show good performance results for the Finite Element Analysis (FEA). The GPU instance of PCG solver using geometric multigrid preconditioning in topology optimization configured to perform a reduced number of FEAs, and iterations per FEA, permitted Wu et al. [34] to solve large-scale problems in a short time. This is done configuring the iterative method with low tolerance level along with SIMP method using standard Optimality Criteria (OC) method [1]. By using this configuration the solution is likely to converge to a local minimum and a mechanical more sound solution might exist [34], however this loss in accuracy is assumed by the authors for the sake of efficiency. The GPU instance is based on the node-wise GPU parallelization proposed by Dick et al. [35], where the grid regularity is exploited to perform coarsening and matrix-vector multiplication operations efficiently. Besides, the operations at the finest level are performed "on-the-fly" to increase the GPU performance.

This paper proposes a multi-granular GPU implementation of the different stages involved in the evolutionary topology optimization method driven by

stress isosurfaces [36] and its variants [37]. On the one hand, a fine-grained GPU implementation of matrix-free PCG solver for structural analysis using the Fixed-Grid FEA (FGFEA) method is proposed. This technique permits to exploit data locality maximizing the GPU performance for FEA [38]. The matrix-vector multiplication operations of PCG GPU instance are performed at the Degree of Freedom (DoF) level, which allows reducing and balancing the workload for all the threads of the MPP architecture [26, 39]. It also permits to compute the matrix-vector operations only for the DoFs within the model (those DOFs attached to inside and boundary elements of the fixed grid), which permits to reduce the computational cost significantly during the optimization process using the iterative solver with Jacobi preconditioner. This improvement is evaluated by the inclusion/exclusion of outside elements in the global system matrix. The performance of structural analysis in evolutionary topology optimization is evaluated in terms of speedup and wall-clock time analyzing two preconditioning techniques; in particular, Jacobi preconditioner and geometric multigrid preconditioner. On the other hand, the other tasks of ESO method driven by stress isosurfaces are implemented using GPU computing in order not to limit the theoretical speedup according to Amdahl's law [40]. In particular, the stress field calculation and the boundary extraction and volume calculation are implemented on GPU at the node level and element level respectively. The proposed GPU instance for the former task balances the workload of CUDA threads increasing the GPU performance. The latter can take about the 10% of the FEA according to [41]. This work proposes a method for the efficient calculation of the volume enclosed by the isosurface generated by the Marching Cubes (MC) algorithm. Such a proposal keeps the cell wise strategy of MC algorithm using a lookup table driven approach. This permits to reduce the computational cost using CPU and also facilitates the data parallelism exploitation at the element level leveraging the power of many-core GPUs.

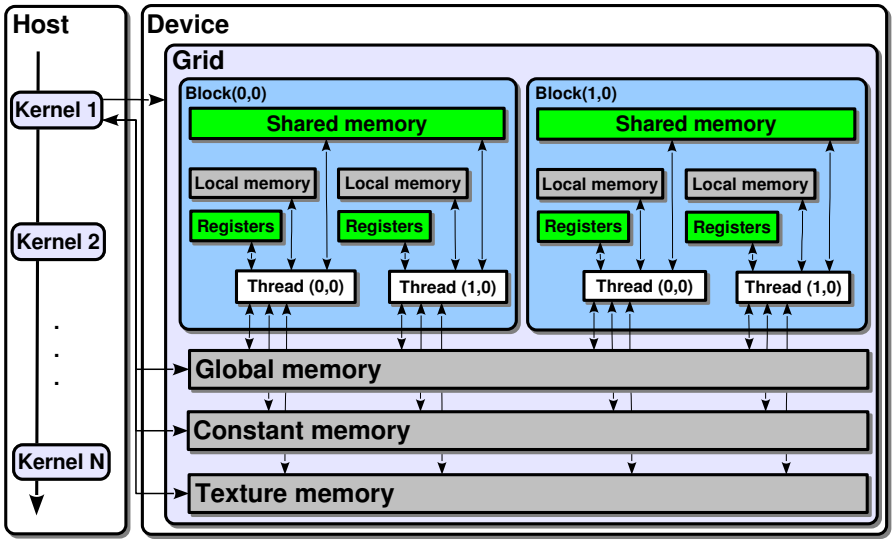
The study of the performance of evolutionary topology optimization driven by stress isosurfaces, with severe computational shortcomings, allows us to explore the challenges in the parallel implementation for every stage of the topol-

ogy optimization pipeline, and thus proposing a well-suited strategy of GPU computing for topology optimization. We have to remark that some of these tasks are common for different topology optimization methods, and thus the conclusions of this work can be applied to the GPU implementation of other shape/topology optimization approaches; in particular, Eulerian based methods in which the structural boundary can be modified by tracking the motion of a level-set function, as is done in the Level-Set Method (LSM).

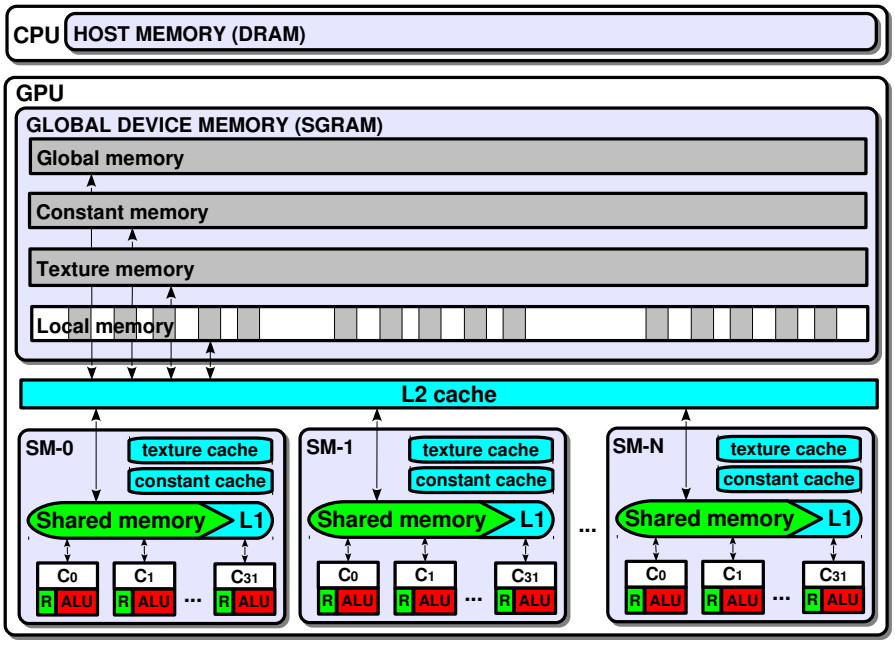
The paper is organized as follows. Section 2 provides an overview of the GPU architecture and the CUDA programming model. The bases of the techniques adopted in this work for evolutionary topology optimization driven by stress isosurfaces are described in section 3. The proposed GPU implementation of such techniques for evolutionary topology optimization is presented in section 4. Section 5 is devoted to the numerical experiments and the performance evaluation of the proposed GPU instance with respect to the classical CPU implementation. Finally, section 6 discusses about the conclusion of the proposal.

2. GPU and CUDA architecture

GPU devices were initially designed to satisfy the market demand of real-time and realistic 3D visualization. The use of these graphic cards with massively parallel architecture in non-graphics HPC applications is becoming very popular due to their high computing capacity. Currently, the use of Nvidia devices and its programming model, Compute Unified Device Architecture (CUDA), is the prevailing tendency, which is adopted in the developments presented in this work. Such a programming model allows us to view the GPU as a compute device able to perform data-parallel computation (data/SIMD parallelism) using multiple cores. The parallel code (single instruction) is defined as a C Language Extension function, called kernel, which is executed by a lot of CUDA threads using different data (multiple data). The kernel call, invoked from the host (CPU) to the device (GPU) as shown in Figure 1(a), should specify the



(a)



(b)

Figure 1: (a) Thread batching and memory model and (b) memory hierarchy of CUDA.

number of CUDA threads organized as a grid of thread blocks.

The CUDA threads have only access to the device SGRAM (Synchronous Graphic Random-Access Memory), a type of DRAM (Dynamic Random-Access Memory) with high bandwidth interface for graphics-intensive functions, and to the on-chip SRAM (Static Random-Access Memory) through the memory spaces depicted in Figure 1(a). The blocks are batch of threads able to cooperate by sharing data through shared memory and to synchronize their execution coordinating memory accesses. A key point is that CUDA architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). The blocks of the grid, invoked by each kernel, are distributed to SMs depending on their execution capacity, which includes on-chip memory resources. The use of on-chip memory, much faster than SGRAM memory, is of paramount importance to increase significantly the GPU performance.

For that reason, the CUDA memory hierarchy, shown in Figure 1(b), is crucial to optimize memory access and achieve a reasonable performance. We can observe that each SM has the following on-chip memory: one set of registers per processor and a shared memory, a read-only constant cache and a read-only texture cache. These memory resources are shared by all cores of such a SM. This fact implies that the amount of blocks a SM can process at once depends on the number of registers per thread and the shared memory per block required for a given kernel. For this reason, the use of shared memory can show relatively poor performance for computation using large arrays. CUDA cannot schedule more blocks to SMs than the multiprocessors can support in terms of shared memory and register usage, and thus the occupancy (number of active warps) is deteriorated. A key point for the proposed GPU instance is that the constant memory is stored in SGRAM but data are read through each multiprocessor constant cache, which is on-chip memory. Constant memory is also optimized for broadcast, i.e. when warp of threads read same location, but it is however serialized when warp of threads read in different locations.

The software developments using CUDA consist of the following steps: i) memory allocation and transaction, ii) kernel execution on GPU and iii) copy

back the results to the host. The strategies to optimize code in GPU computing can be summarized as follows: i) optimization of parallel execution to achieve maximum use of cores, ii) optimization of memory management to facilitate coalesced memory accesses, iii) optimization of instruction usage to achieve maximum instruction performance, and iv) optimization of communications to achieve minimal synchronization between parallel executions. The different effects of the proposed GPU implementation can be explained using these optimization criteria.

3. Evolutionary topology optimization driven by stress isosurfaces

The techniques adopted to address the topology optimization of three-dimensional continuum structures are briefly described below.

3.1. Fixed-Grid Finite Element Analysis (FGFEA)

The FGFEA method [42, 43] is a technique that allows to make fast re-evaluations of modified meshes [44]. It permits to analyze complex finite element models using a structured grid. This is done by superimposing the structural domain Ω, shown in Figure 2(a), over a regular grid of rectangular/cubic equally sized elements, as shown in Figure 2(b). Three types of elements can be created: elements located inside Ω (I elements), elements located outside Ω (O elements), and boundary intersected elements (B elements). The elemental stiffness matrix is given by

$$\begin{aligned} \mathbf{K}^e &= (\xi^e + \Delta(1-\xi^e)) \int_{\Omega^e} \mathbf{B}^T\mathbf{D}\mathbf{B}\ \mathrm{d}\Omega^{\mathrm{e}} \\ &= d^e \int_{\Omega^e} \mathbf{B}^T\mathbf{D}\mathbf{B}\ \mathrm{d}\Omega^{\mathrm{e}} = d^e\mathbf{K}_0^e, \end{aligned} \tag{1}$$

where $\mathbf{B}$ is the strain-displacement matrix, $\mathbf{D}$ is the constitutive material matrix, $\mathbf{K}_0^e$ is the common local stiffness matrix, Ω^e is the element domain, ξ^e is the volume fraction of the element, d^e is the design fraction inside the element, and Δ is a small magnitude close to zero. The common local stiffness matrix $\mathbf{K}_0^e$

is similar for all of the elements due to the regularity of the grid. The design fraction is full ($d^e = 1$) for inside elements I whereas it is a small magnitude ($d^e = \Delta$) for outside elements O. For the boundary elements B the design fraction d^e is given by $\xi^e + \Delta(1 - \xi^e)$, where the volume fraction $\xi^e = V_I^e/V^e$ is the ratio between the elemental volume enclosed by the boundary V_I^e of the real design Γ and the total volume of the element V^e.

The global stiffness matrix $\mathbf{K}$ of linear elasticity problems can be then obtained from the elemental stiffness matrices $\mathbf{K}^e$ using the assembly operator A [45] as follows

$$\mathbf{K} = \mathop{\mathsf{A}}_{e \in \mathscr{E}} \mathbf{C}^{eT} \mathbf{K}^e \mathbf{C}^e, \tag{2}$$

where $\mathscr{E}$ denotes the set of elements and the matrix $\mathbf{C}^e$ represents the transition between local and global numbering of DoFs for the e-th element.

The use of a regular grid permits to compute this $\mathbf{K}^e$ matrix only once at the beginning of the optimization. The disassociation of the boundary and the physical domain from the mesh allows analyzing modifications of boundary model without the regeneration of the mesh. This increases the efficiency of FEA processes that require of mesh regeneration, such as structural topology optimization. In fact, this technique has already been used to alleviate the shortcomings of ESO method related to its computational efficiency [46].

The linear system of equations resulting from the finite element discretization of the linear elasticity system is then as follows

$$\mathbf{K}\mathbf{u} = \mathbf{f}, \tag{3}$$

where $\mathbf{u}$ is the vector of unknown displacements and $\mathbf{f}$ the vector of nodal forces. The resolution of large systems of equations normally requires the use of iterative algorithms, which reduces the memory requirements of the matrix inversion operation at the cost of increasing the processing time of the solve step. The PCG method using different preconditioning techniques is studied in this

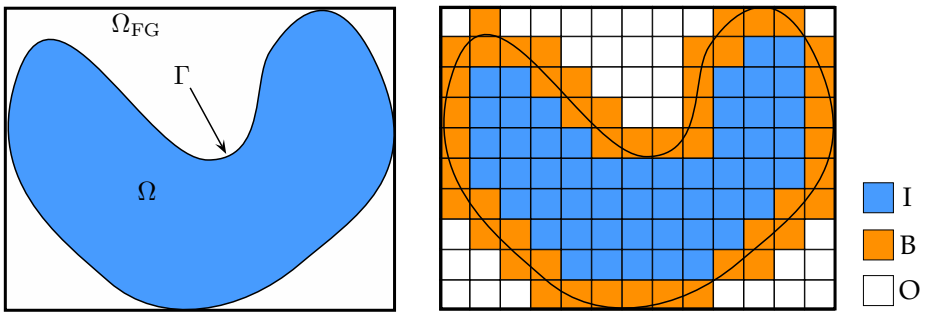


Figure 2: (a) Fixed grid domain and (b) discretization of such a domain.

work to evaluate the performance of structural analysis using GPU computing in evolutionary topology optimization.

The vector $\mathbf{u}$ of unknown displacements is then used to calculate the components of the stress tensor $\sigma_e^{(n)}$ for each node n of the element e of the regular grid as follows

$$\sigma_e^{(n)} = d^e \mathbf{D} \mathbf{B}_e^{\iota_e(n)} \mathbf{u}^{(e)}, \tag{4}$$

where $\iota_e(:)$ is the global-to-local mapping operator indexing the nodes of each element, $\mathbf{B}_e^{\iota_e(n)}$ is the strain-displacement matrix evaluated at each node of the elements, $\mathbf{u}^{(e)}$ is the vector of unknowns and d^e is the design fraction of the element e. The components of the stress tensor $\sigma^{(n)} = [\sigma_x^{(n)}, \sigma_y^{(n)}, \sigma_z^{(n)}, \tau_{xy}^{(n)}, \tau_{yz}^{(n)}, \tau_{xz}^{(n)}]$ at each node are then calculated as

$$\sigma^{(n)} = \frac{1}{k} \sum_{e \in \boldsymbol{\mathscr{E}}^{(n)}} \sigma_e^{(n)}, \tag{5}$$

where k is the number of elements $e \in \boldsymbol{\mathscr{E}}^{(n)}$ connected to the node n. The von Mises stress or equivalent tensile stress $\sigma_{\text{VM}}^{(n)}$ for each node is finally calculated as

$$\sigma_{\text{VM}}^{(n)} = \sqrt{\sigma_x^2 + \sigma_y^2 + \sigma_z^2 - \sigma_x\sigma_y - \sigma_y\sigma_z - \sigma_x\sigma_z + 3(\tau_{xy} + \tau_{xy} + \tau_{xy})^2}\,, \tag{6}$$

where $\sigma = \sigma^{(n)}$ for clarity.

3.2. Marching Cubes (MC) algorithm

The MC algorithm [47] is a well-known cell-by-cell method for extraction of isosurfaces from scalar volumetric data sets [48]. An isosurface can be defined as the surface with constant value, called isovalue, within a volume of space. MC algorithm provides a set of triangles representing such an isosurface. It consists of marking the eight vertices of each cube with 256 (2^8) possible marking scenarios. Each cube marking scenario encodes a cube-isosurface intersection pattern, which provides the edges on which the vertices of triangles lies. For performance reasons, this facetization information is typically stored in a lookup table. The position of each vertex on the edge is estimated using interpolation between the scalar values of the endpoints of the edge.

The early work of Lorensen and Cline [47] considers 15 marking scenarios due to reflective and rotational symmetry. These symmetries provide equivalent cube-isosurface intersection patterns for different marking scenarios, and thus reducing to only 15 unique cube-isosurface intersection patterns the 256 possible marking scenarios. However, some of these basic intersection topologies can be facetized in multiple ways [49]. This ambiguity problem in standard MC algorithm is of paramount importance because inconsistent intersection patterns on the shared face between cells can produce holes in the isosurface. The exploitation of only rotational symmetry – or the non-exploitation of reflective symmetry – overcomes this key problem without using face ambiguity resolution methods [50]. Figure 3 shows the 23 intersection topologies, with circles denoting marked vertices, of the variant of MC algorithm exploiting rotational symmetry.

We have adopted a cell-by-cell strategy to calculate the volume fraction enclosed by the isosurface. The volume fraction of partial cells ξ^e is calculated for the 23 intersection topologies of the MC algorithm, and then the volume fraction of the whole domain is obtained as the addition of partial volume fraction ξ^e of voxels. The volume of partial cells ξ^e only depends on the scalar field of

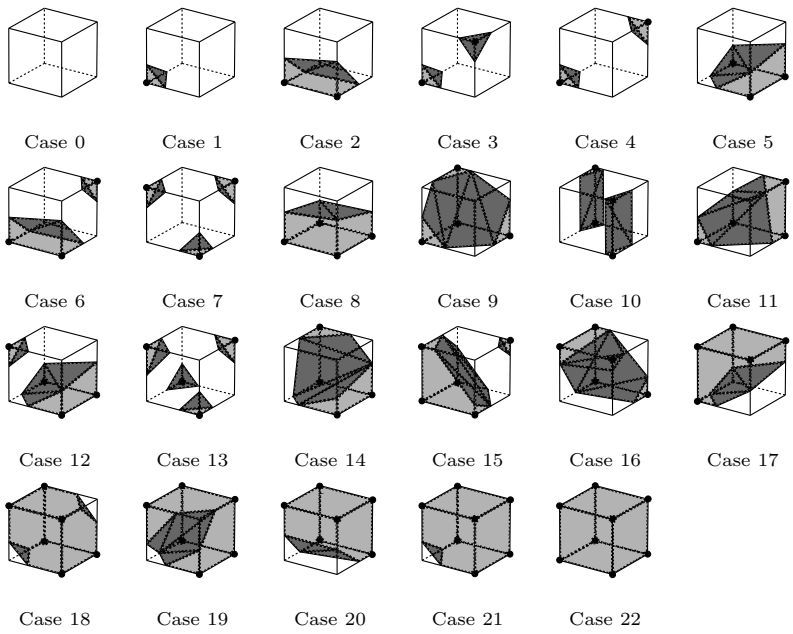


Figure 3: The 23 intersection topologies when only rotational symmetry is exploited and the volume enclosed by the isosurface using MC algorithm.

individual voxels, which allows preserving the benefits, especially the scalability, of MC algorithm. The calculation of partial cells ξ^e is performed by selecting the most suitable case to divide the problem into simplest ones, and then calculating the volume in an efficient way. The volume selection consists of choosing between the volume enclosed or unenclosed by the triangles, up to five, resulting from the MC algorithm. This selection is performed for the 23 intersections topologies, and ensures that the volume of partial cells ξ^e is composed of up to four tetrahedra and one polytope. The volume of each tetrahedron V_{TH} is obtained by

$$V_{TH} = \frac{1}{3!} \begin{vmatrix} a_1 & a_2 & a_3 & 1 \\ b_1 & b_2 & b_3 & 1 \\ c_1 & c_2 & c_3 & 1 \\ d_1 & d_2 & d_3 & 1 \end{vmatrix} \tag{7}$$

where $\mathbf{a} = (a_1, a_2, a_3)$, $\mathbf{b} = (b_1, b_2, b_3)$, $\mathbf{c} = (c_1, c_2, c_3)$ and $\mathbf{d} = (d_1, d_2, d_3)$ are the vertices of the tetrahedron, which are stored using a lookup table for the intersection topologies. The volume V_P enclosed by the polyhedron $P \in \mathbb{R}^3$ is calculated using the divergence theorem. Such a theorem provides important advantages with respect to the popular approach of the tetrahedralization of P. In particular, the divergence theorem does not require that P be convex. Some of the 23 intersections topologies are not convex, and thus the tetrahedralization approach induces significant errors. Besides, the divergence theorem is much more efficient than the tetrahedralization approach. Representing the surface enclosing the polyhedron $P \in \mathbb{R}^3$ as a set of N triangular faces with area A_i, i = {0, ..., N-1}, defined by the vertices (x_i, y_i, z_i) ordered counterclockwise, the volume V_P enclosed by such a polyhedron P is given by

$$V_P = \int_P dv = \frac{1}{3} \sum_{i=0}^{N-1} \int_{A_i} x_i \cdot n_i = \frac{1}{6} \sum_{i=0}^{N-1} x_i \cdot \hat{n}_i, \tag{8}$$

where $\hat{n}_i = (y_i - x_i) \otimes (z_i - x_i)$ is the outer normal to P on each A_i and $n_i = \hat{n}_i / |\hat{n}_i|$ is the outer unit normal. The volume selection ensures that the surface of the polyhedron requires up to 20 triangular faces for the 23 intersection topologies. The vertices of these faces ordered counterclockwise are also stored using a lookup table for performance reasons.

3.3. Iso-stress driven ESO using isosurfaces

The iso-stress driven ESO using isolines is an iterative algorithm that gradually add and/or remove material depending on the shape and distribution of the contour isolines of the desired structural behavior [37]. Such a method is adapted for topology optimization of three-dimensional continuum structures using the FGFEA technique for structural analysis and the MC algorithm for isosurface extraction and volume fraction calculation. This method uses a smooth boundary (isosurface) to represent the structural design, which facilitates the topology interpretation [51, 52]. The topology design method is summarized into the following steps:

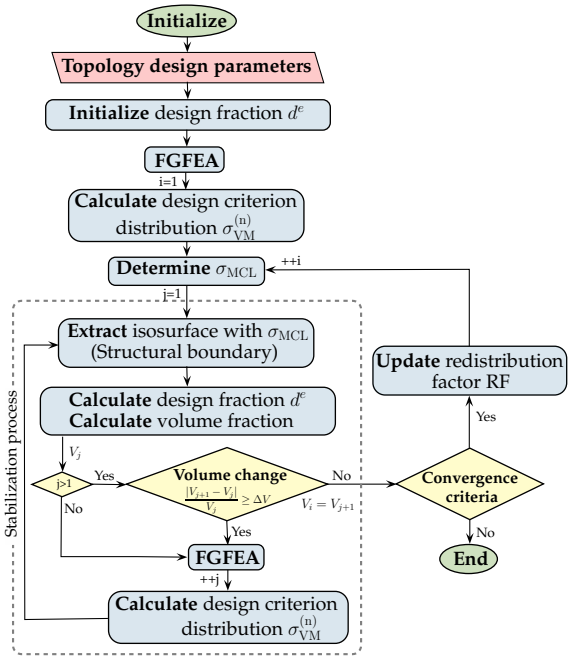


Figure 4: Flowchart of iso-stress driven ESO using isosurfaces.

1. The response of the structural design is calculated using FGFEA. The design criteria distribution within the design domain is then calculated, in particular the von Mises stress distribution $\sigma_{\mathrm{VM}}^{(\mathrm{n})}$ in all of the n nodes enclosed by the design domain.

2. The Minimum Criteria Level (MCL) σ_{MCL} of the design criteria distribution $\sigma_{\mathrm{VM}}^{(\mathrm{n})}$ that produces a new structural boundary, redistributing and removing material, is calculated as

$$\sigma_{\mathrm{MCL}} = RF \times \sigma_{\mathrm{VM}_{\mathrm{max}}}^{(\mathrm{n})}, \tag{9}$$

where $\sigma_{\mathrm{VM}_{\mathrm{max}}}^{(\mathrm{n})}$ is the maximum nodal criterion value of von Mises stress distribution and RF $\in]0, 1[$ is the redistribution factor. This factor is updated in each i iteration as

$$RF_{i+1} = \begin{cases} RF_0 & \text{if } i = 1, \\ RF_i & \text{if } i > 1 \text{ and } \dfrac{|V_{i+1} - V_i|}{V_i} \geqslant \Delta V, \\ RF_i + \Delta RF & \text{otherwise,} \end{cases} \tag{10}$$

where $\Delta RF > 0$ is the increment in the redistribution factor and $\Delta V > 0$ is the minimum volume change between two consecutive iterations. The RF_0, ΔRF and ΔV are empirical values that should be adjusted for each problem. The σ_{MCL} is used as isovalue to obtain the structural boundary (isosurface) from the scalar field of nodal von Mises stress distribution.

3. The structural boundary (isosurface) is then used to obtain the design fraction inside each element d^e of FGFEA following (1), which permits to update the finite element model, removing or redistributing material, and reevaluate the structural response efficiently.
4. The σ_{MCL} is modified according to (9), and the iterative process is repeated from step 1 to step 3 until the desired volume V_T is reached and the following convergence criterion

$$error_i = \frac{|PI_i - PI_{i-1}|}{PI_i} \leqslant \varepsilon \tag{11}$$

is satisfied, where ε is a prescribed tolerance and PI is the performance index. Such an index is defined as $PI = \frac{1}{CV}$ [53], where C and V are the compliance and the volume of the current design.

The flowchart of the structural optimization process is shown in Figure 4. The initialization consists of the tessellation of the design domain using the regular grid of FGFEA method, the information about non-design domain, the material properties, and the loads and boundary conditions of the finite element model. The topology design parameters include the target volume V_T, the initial redistribution factor RF_0, the redistribution factor ΔRF and the minimum volume change limit ΔV. The analysis using FGFEA method requires the design fraction d^e of each element, which is initialized as full ($d^e = 1$) for design domain elements and a small magnitude ($d^e = \Delta$) for non-design domain elements. The

calculation of the design fraction d^e of elements during the optimization process is obtained from the partial volume fraction ξ^e of voxels/elements of MC algorithm. The stabilization loop is an iterative process of reanalysis and material distribution to evolve the structural boundary to the corresponding minimum criteria level σ_{MCL}. This is done by obtaining the structural response $\sigma_{\mathrm{VM}}^{(\mathrm{n})}$ for successive structural boundaries using a certain isovalue σ_{MCL} until the volume change in successive iterations is lower than the empiric value ΔV. The number of iterations of optimization process depends on the empirical parameter ΔRF. The stopping criterion is satisfied when the target volume V_T is reached and the converge criterion of (11) is achieved. The redistribution factor is updated following (10), the new σ_{MCL} is calculated as (9) and the stabilization process is performed until the stopping criterion is satisfied.

4. GPU implementation

GPU computing is used to accelerate the computationally intensive tasks of ESO method driven by stress isosurfaces. Such tasks are the iterative method to solve the system of equations, the calculation of the design criteria distribution $\sigma_{\mathrm{VM}}^{(n)}$ and the isosurface extraction and calculation of the volume enclosed by such an isosurface.

4.1. Fixed-Grid Finite Element Analysis (FGFEA)

The FEA is a demanding task in terms of computation time and memory requirements, mainly due to the assembly and the solving of the system of equations. Iterative solvers and assembly-free methods have been widely used for reducing the memory requirements at the cost of increasing the processing time of the solving stage, which is normally alleviated using parallel computing. The principal computational bottleneck for iterative solvers is the number of iterations required to converge for ill-conditioned problems, which is typically addressed using preconditioning techniques. However, the computation time of the preconditioning may take more wall-clock time than the time spent in

Algorithm 1: DbD PCG algorithm (dbdPCG)

Data: $\mathbf{K}_0^e$, $\mathbf{f}_0$, $\mathbf{u}_0$, $\mathbf{d}$, tol, k_{max}, μ_1, μ_2, n_ℓ, ω, $\mathbf{I_c}$, $\mathbf{I}$, $\mathbf{C}$
Result: $\mathbf{u}$

$\rho_0 \leftarrow 0$; $\gamma_0 \leftarrow 0$; $k \leftarrow 0$; $\ell \leftarrow 0$; // Host initialization
$\mathbf{u} \leftarrow \mathbf{u}_0$; $\mathbf{f} \leftarrow \mathbf{f}_0$; // Device initialization
$\mathbf{r} \leftarrow \mathbf{dbdMVP}(\mathbf{u}, \mathbf{d}, \mathbf{K}_0^e, \ell, \mathbf{I_c}, \mathbf{I}, \mathbf{C})$; // $r = K^e u$
foreach $u \in \mathscr{U}$ **do** // DbD CUDA kernel
  $r^{(u)} \leftarrow f^{(u)} - r^{(u)}$;
end
if *Jacobi* **then** // Jacobi preconditioning
  $\mathbf{z} \leftarrow \mathbf{dbdJacP}(\mathbf{d}, \mathbf{K}_0^e)$;
else if *Multigrid* **then** // Multigrid preconditioning
  $\mathbf{z} \leftarrow \mathbf{VCycle}(\mathbf{K}_0^e, \mathbf{r}, \mathbf{d}, \ell, n_\ell, \omega, \mathbf{I_c}, \mathbf{I}, \mathbf{C}, \mu_1, \mu_2)$;
end
foreach $u \in \mathscr{U}$ **do** // DbD CUDA kernel
  $\rho_0 \leftarrow \rho_0 + z^{(u)} r^{(u)}$; // Atomic operation
  $\gamma_0 \leftarrow \gamma_0 + f^{(u)} f^{(u)}$; // Atomic operation
  $p^{(u)} \leftarrow r^{(u)}$;
end
while ($\sqrt{\rho_k} > tol \cdot \sqrt{\gamma_k}$) **and** ($k < k_{max}$) **do**
  $k \leftarrow k + 1$;
  $\mathbf{a} \leftarrow \mathbf{dbdMVP}(\mathbf{p}, \mathbf{d}, \mathbf{K}_0^e, \ell, \mathbf{I_c}, \mathbf{I}, \mathbf{C})$; // $a = K^e p$
  $\phi_k \leftarrow 0$;
  foreach $u \in \mathscr{U}$ **do** // DbD CUDA kernel
    $\phi_k \leftarrow \phi_k + a^{(u)} p^{(u)}$; // Atomic operation
  end
  $\alpha_k \leftarrow \rho_{k-1} / \phi_k$;
  foreach $u \in \mathscr{U}$ **do** // DbD CUDA kernel
    $u^{(u)} \leftarrow u^{(u)} + \alpha_k p^{(u)}$;
    $r^{(u)} \leftarrow r^{(u)} - \alpha_k a^{(u)}$;
  end
  if *Multigrid* **then** // Multigrid preconditioning
    $\mathbf{z} \leftarrow \mathbf{VCycle}(\mathbf{K}_0^e, \mathbf{r}, \mathbf{d}, \ell, n_\ell, \omega, \mathbf{I_c}, \mathbf{I}, \mathbf{C}, \mu_1, \mu_2)$;
  end
  foreach $u \in \mathscr{U}$ **do** // DbD CUDA kernel
    $\rho_k \leftarrow \rho_k + z^{(u)} r^{(u)}$; // Atomic operation
  end
  $\beta_k \leftarrow \rho_k / \rho_{k-1}$;
  foreach $u \in \mathscr{U}$ **do** // DbD CUDA kernel
    $p^{(u)} \leftarrow r^{(u)} + \beta_k p^{(u)}$;
  end
end

Algorithm 2: DbD Jacobi preconditioner (dbdJacP)

Data: $\mathbf{d}$, $\mathbf{K}_0^e$
Result: $\mathbf{M}$ // Preconditioner

```
foreach u ∈ 𝒰 do                                              // Loop 1
    M^(u) ← 0;
    ℰ^(u) ← Determine index of elements containing u;
    foreach e ∈ ℰ^(u) do                                      // Loop 2
        𝒰^(e) ← Determine the unknowns of e ;
        i ← Extract index of u from 𝒰^(e) ;
        M^(u) ← M^(u) + d^(e) K^e_0ii ;
    end
    M^(u) ← 1/M^(u);
end
```

computing the iterations saved. Besides, the preconditioning methods normally increase the required memory, which can affect seriously the GPU performance. Therefore, a trade-off between memory requirements and computation time is mandatory for the proper GPU implementation of the solving stage.

The use of GPU computing for solving the system of equations of linear elasticity problems using matrix-free and Jacobi PCG methods already has shown its advantages for FGFEA in [38]. In this work, the performance of the solving stage is evaluated using a matrix-free PCG method with two different preconditioning techniques: geometric multigrid preconditioner and Jacobi preconditioner. The use of geometric multigrid methods is especially suitable for regular grids. These methods are based on the *smoothing property* and the *coarse grid principle*. The former reduces the high frequency error components whereas the latter approximates the low frequency error components on coarser grids, which are then prolonged to the finer grids. Their major advantage is that they have an asymptotically optimal complexity of $\mathcal{O}(N)$ and provide mesh-independent convergence and good parallel scalability [54]. However, the performance of these methods deteriorates with increasing contrast in material properties [55], which is the case of FGFEA where outside O and inside I elements have very different properties. The performance decrement is attributed to the coarsening

Algorithm 3: Vcycle Preconditioner (Vcycle)

Data: $\mathbf{K}_0^e$, ${}^{\ell}\mathbf{r}$, $\mathbf{d}$, ℓ, n_ℓ, ω, $\mathbf{I_c}$, $\mathbf{I}$, $\mathbf{C}$, μ_1, μ_2

Result: ${}^{\ell}\mathbf{z}$ // Preconditioner

${}^{\ell}\mathbf{z} \leftarrow 0$;

foreach $i = 1 : \mu_1$ **do**

  ${}^{\ell}\mathbf{s} \leftarrow \mathbf{dbdDJS}({}^{\ell}\mathbf{z}, {}^{\ell}\mathbf{r}, \mathbf{d}, \mathbf{K}_0^e, \ell, \mathbf{I_c}, \mathbf{I}, \mathbf{C})$;

  ${}^{\ell}\mathbf{z} \leftarrow {}^{\ell}\mathbf{s}$;

end

${}^{\ell}\mathbf{z} \leftarrow \mathbf{dbdMVP}({}^{\ell}\mathbf{z}, \mathbf{d}, \mathbf{K}_0^e, \ell, \mathbf{I_c}, \mathbf{I}, \mathbf{C})$; // ${}^{\ell}z = {}^{\ell}K^{e\,\ell}z$

foreach $u \in {}^{\ell}\mathscr{U}$ **do** // DbD CUDA kernel

  ${}^{\ell}v^{(u)} \leftarrow {}^{\ell}r^{(u)} - {}^{\ell}z^{(u)}$;

end

${}^{\ell+1}\mathbf{v} \leftarrow \mathbf{R}_\ell^{\ell+1}({}^{\ell}\mathbf{v})$; // DbD CUDA kernel (Restriction)

if $\ell + 1 == n_\ell$ **then** // Coarsest level

  ${}^{\ell+1}\mathbf{v} \leftarrow$ Copy to Host memory;

  ${}^{\ell+1}\mathbf{w} \leftarrow$ Solve system$\left({}^{n_\ell}\mathbf{K}\right)\left({}^{\ell+1}\mathbf{w}\right) = {}^{\ell+1}\mathbf{v}$; // Direct solver

  ${}^{\ell+1}\mathbf{w} \leftarrow$ Copy to Device memory;

else // Recursion

  ${}^{\ell+1}\mathbf{w} \leftarrow \mathbf{VCycle}(\mathbf{K_0^e}, {}^{\ell+1}\mathbf{v}, \mathbf{d}, \ell+1, \mathbf{n}_\ell, \omega, \mathbf{I_c}, \mathbf{I}, \mathbf{C}, \mu_\mathbf{1}, \mu_\mathbf{2})$

end

${}^{\ell}\mathbf{v} \leftarrow \mathbf{P}_{\ell+1}^{\ell}({}^{\ell+1}\mathbf{w})$; // DbD CUDA kernel (Prolongation)

foreach $u \in {}^{\ell}\mathscr{U}$ **do** // DbD CUDA kernel (correction)

  ${}^{\ell}z^{(u)} \leftarrow {}^{\ell}z^{(u)} + {}^{\ell}v^{(u)}$;

end

foreach $i = 1 : \mu_2$ **do**

  ${}^{\ell}\mathbf{s} \leftarrow \mathbf{dbdDJS}({}^{\ell}\mathbf{z}, {}^{\ell}\mathbf{r}, \mathbf{d}, \mathbf{K}_0^e, \ell, \mathbf{I_c}, \mathbf{I}, \mathbf{C})$;

  ${}^{\ell}\mathbf{z} \leftarrow {}^{\ell}\mathbf{s}$;

end

Algorithm 4: DbD Damped Jacobi Smoother (dbdDJS)

Data: $\mathbf{z}$, $\mathbf{r}$, $\mathbf{d}$, $\mathbf{K}_0^e$, ℓ, ω, $\mathbf{I_c}$, $\mathbf{I}$, $\mathbf{C}$

Result: $\mathbf{s}$

foreach $u \in \mathscr{U}$ **do** // Loop 1
  $n1 \leftarrow$ Determine node containing u;
  $\mathbf{v} \leftarrow$ Determine the unknowns of $n1$;
  $i \leftarrow$ Extract index of u from $\mathbf{v}$;
  foreach $k = 0 : 26$ **do** // Loop 2
    $n2 \leftarrow {}^{\ell}\mathbf{I}_k^{n1}$;
    $\mathbf{w} \leftarrow$ Determine the unknowns of $n2$;
    if $n2 > -1$ **then**
      if $\ell == 0$ **then** // assembly on-the-fly
        foreach $e \in \mathscr{E}^{(n1)}$ **do**
          if $n2 \in \mathscr{N}^{(e)}$ **then**
            $\mathbf{A} \leftarrow \mathbf{A} + d^{(e)} \left(\mathbf{K}_0^e\right)_{\mathbf{v},\mathbf{w}}$;
          end
        end
        $\mathbf{A} \leftarrow$ Impose Dirichlet BC from $\mathbf{I_c}$;
      else
        $\mathbf{A} \leftarrow {}^{\ell}\mathbf{C}_k^{n1}$; // (3X3) coefficients matrix
      end
      if $n2 == n1$ **then**
        $M \leftarrow 1/A_{i,i}$;
      end
      foreach $j = 0 : 2$ **do** // Loop 3
        $s^{(u)} \leftarrow s^{(u)} - \omega M A_{i,j} \left(z^{(\mathbf{w})}\right)_j$;
      end
    end
  end
  $s^{(u)} \leftarrow s^{(u)} + \omega M r^{(u)}$;
end

Algorithm 5: DbD Matrix-Vector Product (dbdMVP)

Data: $\mathbf{p}$, $\mathbf{d}$, $\mathbf{K}_0^e$, ℓ, $\mathbf{I_c}$, $\mathbf{I}$, $\mathbf{C}$

Result: $\mathbf{a}$ `// a = Kp`

foreach $u \in \mathscr{U}$ **do** `// Loop 1`

  $n1 \leftarrow$ Determine node containing u;

  $\mathbf{v} \leftarrow$ Determine the unknowns of $n1$;

  $i \leftarrow$ Extract index of u from $\mathbf{v}$;

  foreach $k = 0 : 26$ **do** `// Loop 2`

    $n2 \leftarrow {}^{\ell}\mathbf{I}_k^{n1}$;

    $\mathbf{w} \leftarrow$ Determine the unknowns of $n2$;

    if $n2 > -1$ **then**

      if $\ell == 0$ **then** `// on-the-fly`

        foreach $e \in \mathscr{E}^{(n1)}$ **do**

          if $n2 \in \mathscr{N}^{(e)}$ **then**

            $\mathbf{A} \leftarrow \mathbf{A} + d^{(e)}\left(\mathbf{K}_0^e\right)_{\mathbf{v},\mathbf{w}}$;

          end

        end

        $\mathbf{A} \leftarrow$ Impose Dirichlet BC from $\mathbf{I_c}$;

      else `// device memory`

        $\mathbf{A} \leftarrow {}^{\ell}\mathbf{C}_k^{n1}$; `// (3X3) coefficients matrix`

      end

      foreach $j = 0 : 2$ **do** `// Loop 3`

        $a^{(u)} \leftarrow a^{(u)} + A_{i,j}\left(p^{(\mathbf{w})}\right)_j$;

      end

    end

  end

end

across discontinuities which affects to the coarse grid correction [56]. Nevertheless, the use of geometric multigrid as preconditioning technique shows good convergence rates for topology optimization problems, given a sufficiently strong smoothing operator [57].

The use of a regular grid permits to calculate and store the common elemental stiffness matrix at the finest grid $\mathbf{K}_0^e$ only once at the beginning of the optimization, whereas the global matrix $\mathbf{K}$ at the finest grid can be calculated "on-the-fly" using the design fraction of elements $\mathbf{d}$ for each analysis. This reduces meaningfully the use of device memory and permits to exploit the data locality [38]. Such an approach is enough for the Jacobi preconditioning but the geometric multigrid preconditioner requires the assembled coefficients at the coarser levels, which are computationally intensive to calculate "on-the-fly" and require significant memory resources when they are stored. One naive alternative is making use of the geometric relationship between successive grids assigning the material properties of coarser-grid elements as the average of those for the underlying fine-grid elements. However, this procedure is a clear deviation from the Galerkin projection [56] and converge rate can be compromise due to the contrast in the system properties. Therefore, a Galerkin-based coarsening is required and the assembled coefficients at the coarser levels need to be stored. This deteriorates the GPU performance due to the global memory accesses through large memory, which does not permit to exploit data locality.

The GPU instance to calculate and store the assembled matrices of coefficients $\mathbf{C}$ at the coarser levels is of paramount importance for an efficient geometric multigrid implementation. The use of a regular grid permits to know a priori the contributions to the matrix of coefficients per element and per node, which are bounded by 8 elements and 27 nodes per node. This permits to set the maximum size of global stiffness coefficients per node, which is bounded by 27 matrices of dimension 3×3. Such 27 matrices are related to the contributions of the 3^3 grid neighborhood of the node. This storage scheme requires the indexes of the adjacent nodes, which are stored on a vector $\mathbf{I}$ of integers where -1 means that the node does not exit. The storage of the global stiffness

matrix $\mathbf{C}$ per node has a similar size than using a sparse-matrix representation but permits to allocate the required memory for the assembly at the beginning, which has significant computational benefits for GPU computing.

The coefficient matrices for the coarser levels are obtained from the finer levels using a Galerkin-based coarsening following ${}^{\ell+1}\mathbf{C} = \mathbf{R}_{\ell}^{\ell+1}\ {}^{\ell}\mathbf{C}\ \mathbf{P}_{\ell+1}^{\ell}$, where $\mathbf{R}$ and $\mathbf{P}$ are the restriction and prolongation operators respectively. Following [35], the coarsening operation is computed in a node-by-node matrix-free fashion using a two-step approach. Firstly, a linear combination of the 3^3 fine grid neighborhood of considered node is performed, corresponding to a linear combination of the rows of ${}^{\ell}\mathbf{C}$. Secondly, these coefficients are interpolated to the coarser grid vertices, corresponding to a linear combination of the columns of ${}^{\ell}\mathbf{C}$. These operations require the vector $\mathbf{I_e}$ of indexes of the elements contributing to each node. The GPU instance for the coarsening is performed assigning one CUDA thread to the calculation of each one of the 27 matrices of coefficients per node of the coarser level. This fine granularity provides good performance in the calculation of the matrices of coefficients at the coarser levels. The assembled matrices of coefficients ${}^{\ell}\mathbf{C}$ at the coarser levels ℓ are calculated and stored in the device memory. The global matrix $\mathbf{K}$ of the finest grid is calculated "on-the-fly" using the elemental matrix $\mathbf{K}_0^e$ and the design fraction of elements $\mathbf{d}$. This is done for both preconditioners and allows exploiting data locality alleviating mostly of memory related problems in GPU architectures.

The pseudocode of the DoF-by-DoF (DbD) PCG or dbdPCG GPU instance using both preconditioners for FGFEA is shown in Algorithm 1. Such an algorithm assumes that the hierarchical grids are composed of equally sized first-order isoparametric hexahedral elements. This tessellation provides for the finest grid a set of elements $\mathscr{E}$, a set of nodes $\mathscr{N}$ and a set of unknowns $\mathscr{U}$, which are calculated "on-the-fly". The vector $\mathbf{I_c}$ indicating the boundary conditions per node is also required to impose the Dirichlet conditions to the corresponding DoFs at the finest level. Thus, the input data of the dbdPCG algorithm for the finest level are the common stiffness matrix $\mathbf{K}_0^e$, the vector of forces $\mathbf{f_0}$, an initialization of displacements $\mathbf{u_0}$, the vector of elemental densities $\mathbf{d}$ and the

vector $\mathbf{I_c}$ indicating the boundary conditions per node. For the coarser levels, the input data are the vector $\mathbf{I}$ of adjacent nodal indexes per node and the assembled matrices of coefficients $\mathbf{C}$ per level. Note that the data of the coarser levels is not needed for the Jacobi preconditioner. The algorithm also requires the tolerance tol and the maximum number of iterations k_{max} for the stopping criteria of the iterative method, the number of grid levels n_ℓ, the number of pre- and post-smoothing steps μ_1 and μ_2, and the damping factor for Jacobi smoothing ω. The GPU instance of PCG requires the matrix-vector product (dbdMVP) for both preconditioners. Besides, it also requires the diagonally preconditioner (dbdJacP) for the Jacobi preconditioning and the Vcycle preconditioner (Vcycle) for the geometric multigrid preconditioning. Additionally, the GPU instance requires the calculation of diverse vector arithmetic operations within the labeled loops with "DbD CUDA kernel".

The pseudocode of the Jacobi preconditioner (dbdJacP) kernel is detailed in Algorithm 2. It calculates the Jacobi preconditioner "on-the-fly" using the common stiffness matrix $\mathbf{K}_0^e$ and the vector of elemental densities $\mathbf{d}$. This simple preconditioner is computationally cheap and only requires storing a vector of the dimension of unknowns. The pseudocode of the matrix-vector product (dbdMVP) kernel is shown in Algorithm 5. This algorithm also performs the matrix-vector operation "on-the-fly" for the finest level ($\ell = 0$) using the elemental densities $\mathbf{d}$ and the common stiffness matrix $\mathbf{K}_0^e$. For the coarser levels, it takes the matrix of coefficients contributing to the node to perform the operation. Nevertheless, the grain size of the operations is performed at the DoF level.

The pseudocode of the geometric multigrid preconditioner (Vcycle) kernel is shown in Algorithm 3. Such a preconditioning is carried out by a recursive call to the V-cycle algorithm. The algorithm requires as input data the elemental densities $\mathbf{d}$, the common stiffness matrix $\mathbf{K}_0^e$, the vector $\mathbf{I_c}$ of boundary conditions for the finest level ($\ell = 0$) and the assembled matrix of coefficients $\mathbf{C}$ for the coarser levels. It also needs the vector $\mathbf{I}$ of adjacent nodal indexes per node, the residual ${}^{\ell}\mathbf{r}$ and the parameters ω, μ_1 and μ_2 for all the levels.

The algorithm performs a matrix-vector product (dbdMVP) and the multigrid smoother (dbdDJS) for each level. Besides, diverse vector arithmetic operations are performed using custom developed kernels. To transfer information between two consecutive grids ${}^{\ell+1}\boldsymbol{\Omega}$ and ${}^{\ell}\boldsymbol{\Omega}$, a prolongation operator $\mathbf{P}_{\ell+1}^{\ell} : {}^{\ell+1}\Omega \to {}^{\ell}\Omega$ and a restriction operator $\mathbf{R}_{\ell}^{\ell+1} : {}^{\ell}\Omega \to {}^{\ell+1}\Omega$ are introduced. The geometric relationship between hierarchical grids allows us to avoid storing the prolongation operator $\mathbf{P}_{\ell+1}^{\ell}$ and the restriction operator $\mathbf{R}_{\ell}^{\ell+1}$ and to work with the stencils instead, which are constant or can be computed "on-the-fly" when needed. The number of levels ℓ is selected in order to ensure the coarsest level is small enough to be solved using a sparse LU decomposition on CPU. When the number of levels ℓ is properly selected, the number of DoFs in the coarsest grid is relatively small and the system of equations can be solved with a direct method on CPU. The pseudocode of the multigrid smoother (dbdDJS) kernel is shown in Algorithm 4. Such a smoother is based on the damped Jacobi method, which uses the inverse of the diagonal of global stiffness matrix with a relaxation parameter ω.

The implementation of all these CUDA kernels is performed at the DoF level. This is a key point to distribute properly the workload between the threads of the MPP architecture, especially for the matrix-vector product operation [26]. This granularity also permits to optimize the implementation of FGFEA by excluding outside O elements, i.e. DOFs attached to nodes unenclosed by the isosurface. Such an exclusion provides significant improvements in computation time due to the reduction of DoFs of the finite element model and the reduction in the condition number of the system matrix which is related with the number of iterations performed by the PCG algorithm [43]. This is done by simply assigning CUDA threads to the unknowns $u \in \mathscr{U}$ which are attached to inside I and boundary B elements of FGFEA.

The labeled loops with "DbD CUDA kernel" indicate that the computation is performed on GPU with granularity at the DoF level. These loops are independent of the grid connections, and thus their implementation as CUDA kernels is straightforward. The arithmetic operations labeled with "Atomic op-

eration" require the synchronization of the threads involved in the computation to add the resulting data of all these threads. This is done using atomic addition in CUDA, which permits to read, modify, and write a value back to device memory without the interference of any other threads.

4.2. Design criteria calculation

Algorithm 6: NbN stress calculation (nbnStress)

Data: $\mathbf{u}$, $\mathbf{d}$, n_e, $\mathbf{DB}^{(i)}$

Result: $\boldsymbol{\sigma}_x$, $\boldsymbol{\sigma}_y$, $\boldsymbol{\sigma}_z$, $\boldsymbol{\tau}_{xy}$, $\boldsymbol{\tau}_{xz}$, $\boldsymbol{\tau}_{yz}$, $\boldsymbol{\sigma}_{vm}$

$\boldsymbol{\sigma}_{\mathbf{x}} \leftarrow 0$; $\boldsymbol{\sigma}_{\mathbf{y}} \leftarrow 0$; $\boldsymbol{\sigma}_{\mathbf{z}} \leftarrow 0$; $\boldsymbol{\tau}_{\mathbf{xy}} \leftarrow 0$; $\boldsymbol{\tau}_{\mathbf{yz}} \leftarrow 0$; $\boldsymbol{\tau}_{\mathbf{xz}} \leftarrow 0$; $\boldsymbol{\sigma}_{\mathbf{vm}} \leftarrow 0$;

foreach $n \in \mathscr{N}$ **do** // Loop 1

  $i_e \leftarrow 0$;

  $\mathscr{E}^{(n)} \leftarrow$ Determine index of elements containing n;

  foreach $e \in \mathscr{E}^{(n)}$ **do** // Loop 2

    $\mathscr{U}^{(e)} \leftarrow$ Determine unknowns of e ;

    $\mathscr{N}^{(e)} \leftarrow$ Determine nodes attached to e ;

    $i \leftarrow$ Determine local index $\iota_e(n)$, $n \in \mathscr{N}^{(e)}$;

    foreach $j \in \{1, \ldots, n_e\}$ **do** // Loop 3

      ${\sigma_x}^{(n)} \leftarrow {\sigma_x}^{(n)} + d^{(e)} DB_{1j}^{(i)} u_j^{(e)}$;

      ${\sigma_y}^{(n)} \leftarrow {\sigma_y}^{(n)} + d^{(e)} DB_{2j}^{(i)} u_j^{(e)}$;

      ${\sigma_z}^{(n)} \leftarrow {\sigma_z}^{(n)} + d^{(e)} DB_{3j}^{(i)} u_j^{(e)}$;

      ${\tau_{xy}}^{(n)} \leftarrow {\tau_{xy}}^{(n)} + d^{(e)} DB_{4j}^{(i)} u_j^{(e)}$;

      ${\tau_{yz}}^{(n)} \leftarrow {\tau_{yz}}^{(n)} + d^{(e)} DB_{5j}^{(i)} u_j^{(e)}$;

      ${\tau_{xz}}^{(n)} \leftarrow {\tau_{xz}}^{(n)} + d^{(e)} DB_{6j}^{(i)} u_j^{(e)}$;

    end

    $i_e \leftarrow i_e + 1$;

  end

  ${\sigma_x}^{(n)} \leftarrow {\sigma_x}^{(n)}/i_e$; ${\sigma_y}^{(n)} \leftarrow {\sigma_y}^{(n)}/i_e$; ${\sigma_z}^{(n)} \leftarrow {\sigma_z}^{(n)}/i_e$;

  ${\tau_{xy}}^{(n)} \leftarrow {\tau_{xy}}^{(n)}/i_e$; ${\tau_{yz}}^{(n)} \leftarrow {\tau_{yz}}^{(n)}/i_e$; ${\tau_{xz}}^{(n)} \leftarrow {\tau_{xz}}^{(n)}/i_e$;

  ${\sigma_{vm}}^{(n)} \leftarrow$ Compute von Misses stress according to (5);

end

The calculation of the stress tensor components $\sigma^{(n)}$ and the equivalent tensile stress $\sigma_{\mathrm{VM}}^{(n)}$ is accelerated using GPU computing. This is done by performing the operations from (4) to (6) at the node level. The regularity of the first-order

isoparametric hexahedral elements composing the regular grid implies that the strain-displacement matrix $\mathbf{B}^{(i)} = \mathbf{B}_e^{(i)}$, evaluated at the $i = \{1, \ldots, 8\}$ nodes, is similar for all the e elements. Since the constitutive material matrix $\mathbf{D}$ is also constant for all the elements of the regular grid, the result of the products $\mathbf{DB}^{(i)}$ can be stored in a lookup table to save this computation, which is calculated only once at the beginning of the optimization.

The pseudo-code of the GPU instance of equivalent tensile stress calculation Node-by-Node (NbN) stress or nbnStress is shown in Algorithm 6. The input data of this algorithm are the resulting displacements $\mathbf{u}$ of the analysis, the design fraction $\mathbf{d}$ of elements, the number n_e of DoFs per element and the result of the products $\mathbf{DB}^{(i)}$. The algorithm is designed as three nested loops. The first loop applies to each node $n \in \mathscr{N}$ for which the elements $e \in \mathscr{E}^{(n)}$ connected to such a node n are determined. The contribution of these connected elements $\sigma_e^{(n)}$ to the stress tensor components $\sigma^{(n)}$ of each node n is averaged according to equation (5). The equivalent tensile stress $\sigma_{\mathrm{VM}}^{(n)}$ is then calculated following the expression (6). The second loop operates on each element $e \in \mathscr{E}^{(n)}$ to determine the nodes and unknown displacements contained by such an element. This information is used to obtain the contribution of each element $\sigma_e^{(n)} = d^{(e)}\mathbf{DB}^{(i)}\mathbf{u}^{(e)}$ to the stress tensor components in the third loop. The algorithm is implemented as a CUDA kernel assigning one thread to each node of the first loop, i.e. the granularity of the GPU implementation is at the node level.

4.3. Isosurface extraction and volume calculation

The isosurface extraction from the design criteria field and the calculation of the volume enclosed by such an isosurface is also performed using GPU computing. This work proposes an efficient method for the calculation of the volume enclosed by the isosurface obtained from MC algorithm. The GPU instance is implemented at the element level, which is called Element-by-Element (EbE) volume calculation or ebeVol GPU instance. The pseudocode of this GPU instance is shown in Algorithm 7. The input data are the design criteria distribution $\sigma_{\mathrm{VM}}^{(n)}$, the isovalue σ_{MCL} and the cell volume V_c for the regular grid.

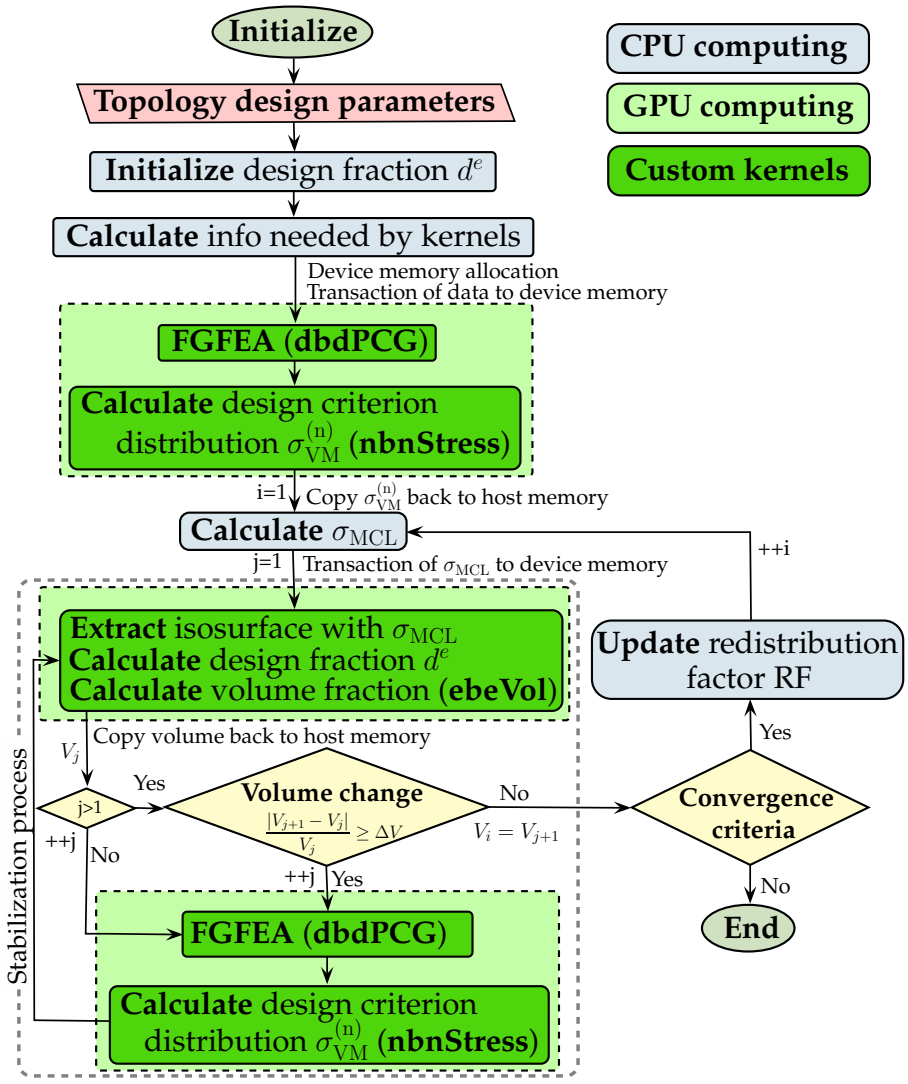


Figure 5: Flowchart of GPU instance of ESO using isosurfaces.

Algorithm 7: EbE volume calculation (ebeVol)

Data: $\boldsymbol{\sigma}_{vm}^{(n)}$, σ_{MCL}, V_c

Result: ξ

foreach $e \in \mathscr{E}$ **do**
  $\mathbf{m}^{(e)} \leftarrow$ Calculate marking scenario; // 2^8 cases
  $p^{(e)} \leftarrow LuT_1(\mathbf{m}^{(e)})$; // Intersection pattern - 23 cases
  switch $p^{(e)}$ **do**
    case *0* **do** // O element
      $\xi^{(e)} = 0$;
    end
    case *22* **do** // I element
      $\xi^{(e)} = 1$;
    end
    otherwise do // B element
    end
    $V^{(e)} \leftarrow 0$;
    $\mathbf{c}^{(e)} \leftarrow LuT_2(m^{(e)})$; // Max 5 connections
    $\mathbf{t}^{(e)} \leftarrow$ Interpolation of $\mathbf{c}^{(e)}$; // Triangles
    $\mathbf{t_h}^{(e)} \leftarrow LuT_3(m^{(e)})$; // Tretrahedra
    $V_{TH}^{(e)} \leftarrow$ Calculate volume of tetrahedra with $\mathbf{t_h}^{(e)}$;
    $\mathbf{p_h}^{(e)} \leftarrow LuT_4(m^{(e)})$; // Polyhedron
    $V_P^{(e)} \leftarrow$ Calculate volume of polyhedron with $\mathbf{p_h}^{(e)}$;
    $V^{(e)} \leftarrow V^{(e)} + V_{TH}^{(e)} + V_P^{(e)}$;
    if $LuT_5(m^{(e)})$ **then** // Volume calculation case
      $\xi^{(e)} = {}^{V^{(e)}}/_{V_c}$; // Normal case
    else
      $\xi^{(e)} = {}^{(V_c - V^{(e)})}/_{V_c}$; // Complementary case
    end
  end
end

The algorithm consists of a simple loop operating on each element. This loop calculates the marking scenario for the element $\mathbf{m}^{(e)}$, which is then used to determine the cube-isosurface intersection pattern $p^{(e)}$ using the lookup table LuT_1. The volume fraction $\xi^{(e)}$ is empty and full for the cases 0 and 22, shown in Figure 3, respectively. These cases correspond to the outside O and inside I elements of FGFEA method.

The volume fraction $\xi^{(e)}$ of boundary elements B is calculated using the triangles provided by MC algorithm. The edges $\mathbf{c}^{(e)}$ containing the vertices of MC triangles are calculated using the lookup table LuT_2 for the marking scenario $m^{(e)}$. The position of the vertices of MC triangles $\mathbf{t}^{(e)}$ is calculated by interpolation. The lookup table LuT_3 provides the vertices of the tetrahedra $\mathbf{t_h}^{(e)}$ for the marking scenario $m^{(e)}$, which are used to calculate the volume of tetrahedra $V_{TH}^{(e)}$ according to (7). The lookup table LuT_4 provides the vertices of the triangular faces of the polyhedron $\mathbf{p_h}^{(e)}$ for the marking scenario $m^{(e)}$, which are used to calculate the volume of polyhedron $V_P^{(e)}$ according to (8). The addition of both volumes is the volume $V^{(e)}$ enclosed by the triangles provided by MC algorithm. The lookup tables LuT_3 and LuT_4 provide the information to calculate the volume fraction $\xi^{(e)}$ or the complementary case following a simplicity criterion. The lookup table LuT_5 indicates the volume case, provided by LuT_3 and LuT_4, to calculate properly the volume fraction $\xi^{(e)}$ for the corresponding marking scenario $m^{(e)}$. The algorithm is implemented as a CUDA kernel assigning the operations on each cell to one thread, i.e. the granularity of the GPU implementation is at the element level.

4.4. Evolutionary topology optimization

The flowchart of the GPU instance of ESO driven by isosurfaces is shown in Figure 5. The GPU implementation consists of the development of custom designed kernels for the computationally demanding tasks involved in the algorithm. These CUDA kernels are the PCG solver (dbdPCG), the von Mises stress calculation (nbnStress) and the design fraction computation (ebeVol). The flowchart is designed to minimize the device memory allocation and the

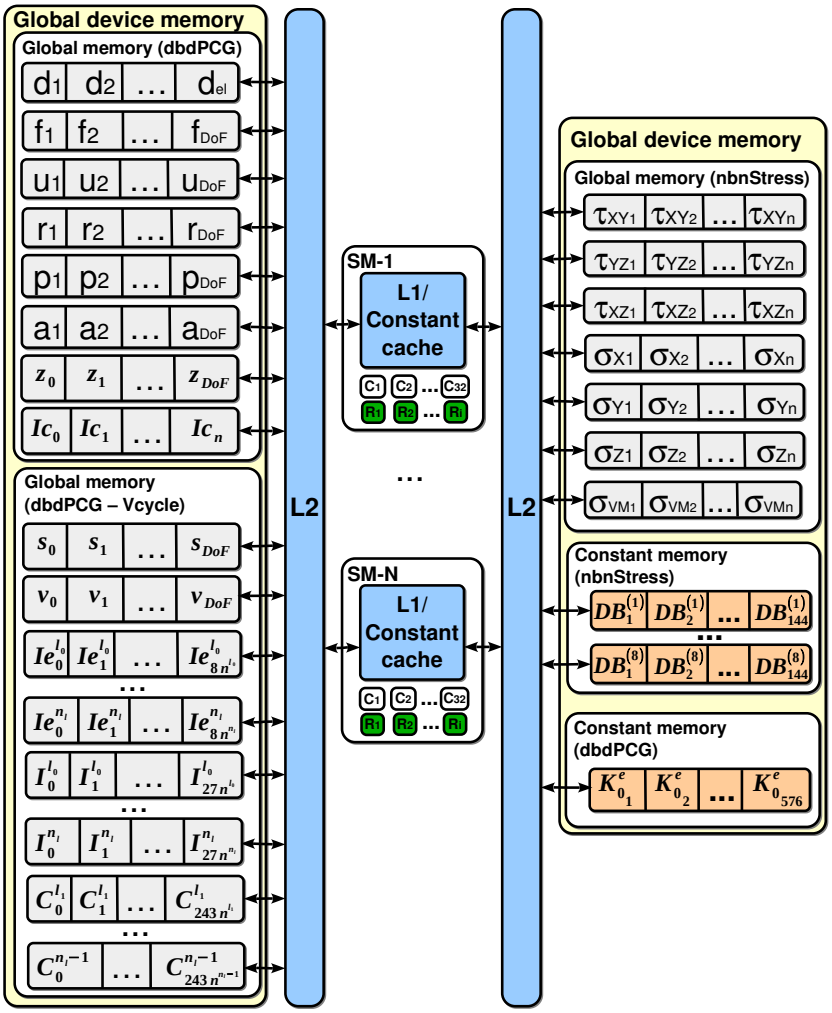


Figure 6: Memory required by GPU instance of dbdPCG and nbnStress.

memory transactions between host and device memory. This is of paramount importance to obtain reasonable results using GPU computing. One can observe that the information needed by the kernels is allocated and transferred to the device memory in the initialization of the optimization process. The iterations of the optimization process only require to copy back to host memory some scalar values and the design criteria distribution $\sigma_{\mathrm{VM}}^{(n)}$ for the implementation using the Jacobi preconditioner. In the case of the PCG using the geometric multigrid preconditioner, the vector of unknowns ${}^{\ell}\mathbf{v}$ at the coarsest level is also copied back to the host for solving the system of equations using a direct solver. This minimization of memory transactions increases notably the GPU performance.

The device memory allocated for dbdPCG and nbnStress kernels, obviating the allocation of scalar values, is shown in Figure 6. The dbdPCG kernel using the Jacobi preconditioner requires the storage in the global device memory of vectors $\mathbf{d}$, $\mathbf{f}$, $\mathbf{u}$, $\mathbf{r}$, $\mathbf{p}$, $\mathbf{a}$, $\mathbf{z}$, and $\mathbf{I_c}$ indicated in the pseudocode of Algorithm 1. When the geometric multigrid preconditioner is used, the vectors $\mathbf{s}$, $\mathbf{v}$, ${}^{\ell}\mathbf{I_e}$, ${}^{\ell}\mathbf{I}$ and ${}^{\ell+\mathbf{1}}\mathbf{C}$ are also stored in the global device memory for the corresponding n_l levels l. The common elemental stiffness matrix $\mathbf{K}_0^e$ is stored in constant memory. This permits to save bandwidth because constant memory is cached and consecutive reads of the same address does not incur any additional memory traffic. Besides, one single read from constant memory is broadcast to the threads of a half-warp. The nbnStress kernel requires the storage in the global device memory of vectors $\boldsymbol{\sigma}_x$, $\boldsymbol{\sigma}_y$, $\boldsymbol{\sigma}_z$, $\boldsymbol{\tau}_{xy}$, $\boldsymbol{\tau}_{xz}$, $\boldsymbol{\tau}_{yz}$ and $\boldsymbol{\sigma}_{vm}$. The length of these vectors is the number of nodes n of the fixed grid. The result of the products $\mathbf{DB}^{(i)}$ for the $i = \{1, \ldots, 8\}$ nodes of the common first-order isoparametric hexahedral element are also stored in constant memory, taking advantage of the same benefits that the common elemental stiffness matrix $\mathbf{K}_0^e$. The memory used by the five lookup tables of the ebeVol GPU implementation does not depend on the dimension of the grid. This information is also stored in global device memory.

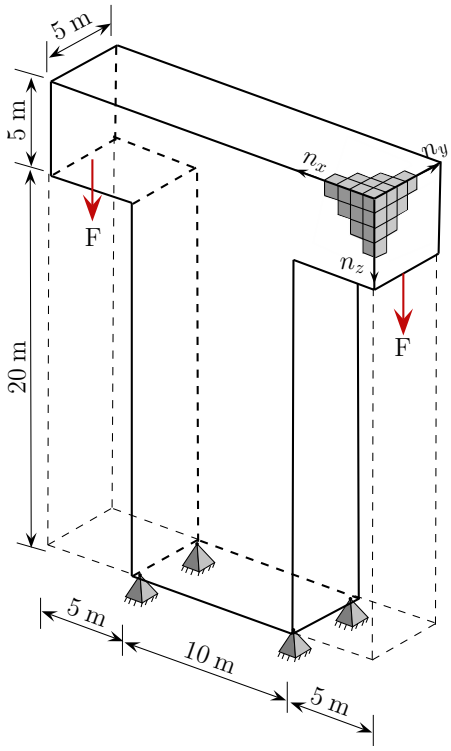
Nvidia GPU model	CUDA cores	Processor clock (MHz)	Memory clock (MHz)	FMA-DP (GFlops)
Quadro 4000 (GF100-100-KD)	256	475	1400	243
Tesla C2070 (GF100)	448	575	1566	515.2
Tesla K40 (GK110b)	2880	745	3004	1430

Table 1: GPU specifications for benchmark devices.

5. Numerical experiments

The performance of the proposal to accelerate the ESO method driven by stress isosurfaces using GPU computing is evaluated using three real-life topology optimization problems; in particular, the designs of the electric mast, the tied-arch bridge and the Messerschmidt–Bölkow–Blohm (MBB) aircraft floor beam. The solving of the system of equations is evaluated using two preconditioning techniques; in particular, the Jacobi preconditioner and the geometric multigrid preconditioner. The performance of the former is also evaluated by the inclusion/exclusion of outside O elements in the global system matrix of FGFEA. The GPU instance of these problems is compared with the classical CPU implementation, in which the global stiffness matrix of (3) is assembled and the sparse-matrix representation is used to perform the operations required by the instances of PCG solvers. The CPU implementation makes use of only one thread for the comparisons. The computationally demanding tasks involved in the algorithm are evaluated separately using GPUs with different parallel capabilities. This aims to evaluate the scalability of the GPU instance with respect to the capabilities of the graphics units. The first two experiments are relatively large benchmarks which are used to evaluate the performance of the GPU instances for usual models. The third problem is a large scale benchmark that aims to show the ability of the proposal to address high resolution topology optimization using only one computer equipped with one graphics card.

The numerical experiments are performed using a computer with an Intel Core i7 980 3.33 GHz and 24 GB of RAM memory. Three Nvidia GPUs are installed in the computer to perform the experiments: Quadro 4000, Tesla C2070 and Tesla K40. The first two graphics cards use the Fermi microarchitecture,



(a)

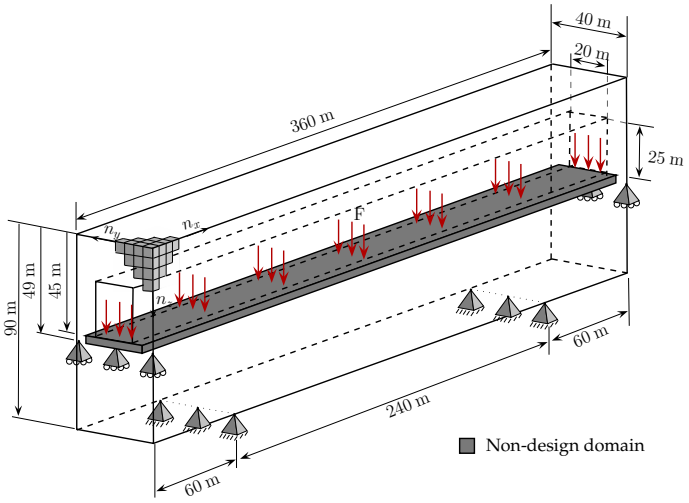
(b) (c)

Figure 7: Electric mast benchmark: (a) design domain and boundary conditions, topology design from (b) isometric and (c) front view.

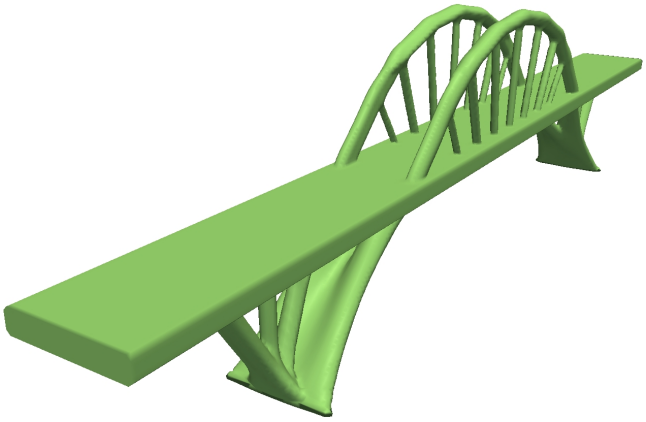
whereas the third one makes use of Kepler microarchitecture. Table 1 summarizes the most important specifications of such graphics units for scientific computation purposes; in particular, the number of cores, the processor and memory clocks, and the Double-Precision (DP) Fused Multiply Add (FMA) operations as specified in IEEE 754-2008. The GPU instance is compiled using the NVIDIA CUDA Toolkit 7.5. The numerical experiments are run on 64 bits Linux OS with the NVIDIA Driver Version 352.63. It is important to remark that the development environment and the graphics driver updates often show significant performance improvements.

The electric mast benchmark consists of finding the best structural design for a T-shaped design domain where the corners of the bottom part of the T-leg are simply-supported and two symmetric vertical loads are applied in the bottom middle part of T-shafts. The two loads F = 10KN represent the force applied by the wires on the mast. Figure 7(a) shows the T-shaped design domain with continuous line and the domain used by FGFEA with dashed line. It also shows the boundary conditions of the optimization problem. The domain used by FGFEA consists of a box shape of $20 \times 5 \times 25$ meters. The finite element model is reduced to the half by imposing the symmetric boundary condition, giving rise to a fixed grid of $60 \times 28 \times 148$ hexahedral elements with 790743 DoFs. The relative small size of the model only permits to perform three levels in the geometric multigrid preconditioning of the PCG solver up to a grid size of $15 \times 7 \times 37$ hexahedral elements.

The tied-arch bridge benchmark consists of finding the optimal layout of material for a box shape design domain where the base of bridge abutments are simply-supported and a uniformly distributed load is applied to the top of the non-optimizable bridge deck. The bottom part of the bridge deck is simply-supported. Figure 8(a) shows the box shape design domain and the boundary conditions of the optimization problem. It also shows the non-optimizable region over the top of the bridge deck, which represents the area needed to circulate the vehicles. This topology optimization problem also makes use of the symmetric boundary condition, and thus only half of the finite element model is analyzed.



(a)



(b)

(c)

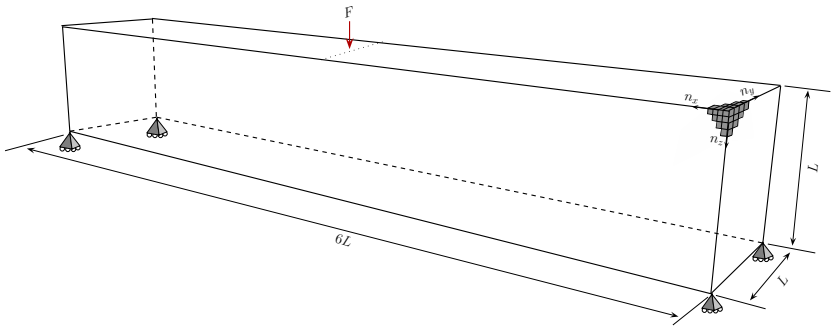
Figure 8: Tied arch bridge benchmark: (a) design domain and boundary conditions, topology design from (b) isometric and (c) front view.

Figure 9: Tied arch bridge taken from http://www.sellwoodbridge.org.

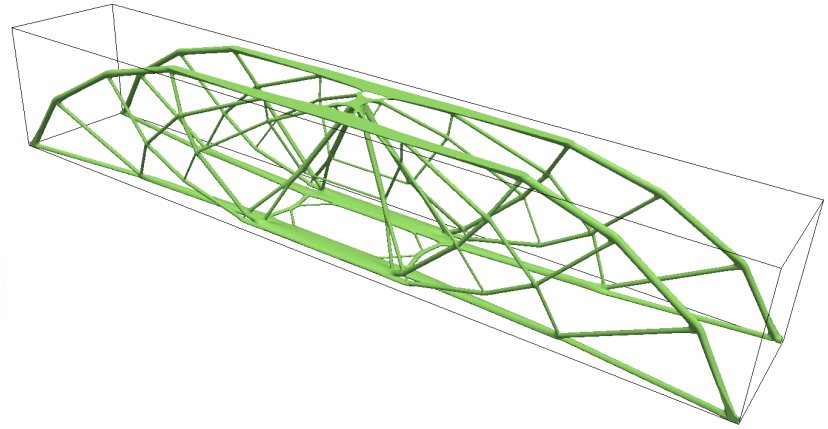
The half design domain is discretized using $184 \times 40 \times 96$ hexahedral elements with 2207235 DoFs. The size of this model allows performing four levels in the geometric multigrid preconditioning of the PCG solver up to a grid size of $23 \times 5 \times 12$ hexahedral elements.

The MBB beam benchmark consists of finding the best structural design for a box shape design domain which is loaded at the center of the top part and the corners of the bottom part are pinned supported. Figure 10(a) shows the box shape design domain and the boundary conditions of the optimization problem. The half design domain is discretized using $432 \times 144 \times 144$ (8.9 millions) hexahedral elements with about 27M DOFs. The size of the model permits to perform five levels in the geometric multigrid preconditioning of the PCG solver up to a grid size of $27 \times 9 \times 9$ hexahedral elements.

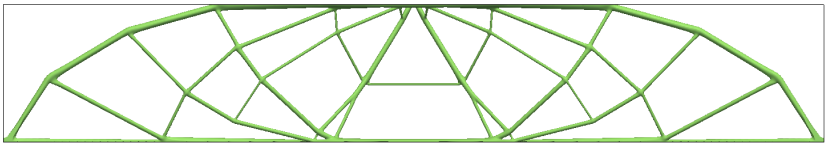
The topology optimization problems make use of the same material properties (E = 210GPa and $\nu = 0.31$) and small magnitude ($\Delta = 10^{-6}$) for FGFEA approach. The smoothing step is configured with a damping factor $\omega = 0.4$ for the electric mast and MBB beam benchmarks, whereas $\omega = 0.6$ is set for the tied-arch bridge experiment. All the calculations are performed using double-



(a)



(b)



(c)

Figure 10: MBB beam benchmark: (a) design domain and boundary conditions and topology design from (b) isometric and (c) front view.

Model	#DoFs	V_T (%)	#FEA	tol	Memory (MB)		
					Jacobi	Multigrid	Stress
Electric mast	790743	3	213	10^{-12}	33.07	153.86	22.01
Tied-arch bridge	2207235	3	340	10^{-12}	92.40	437.47	61.53
MBB beam	27311475	1.5	385	10^{-8}	1144.93	5504.45	762.92

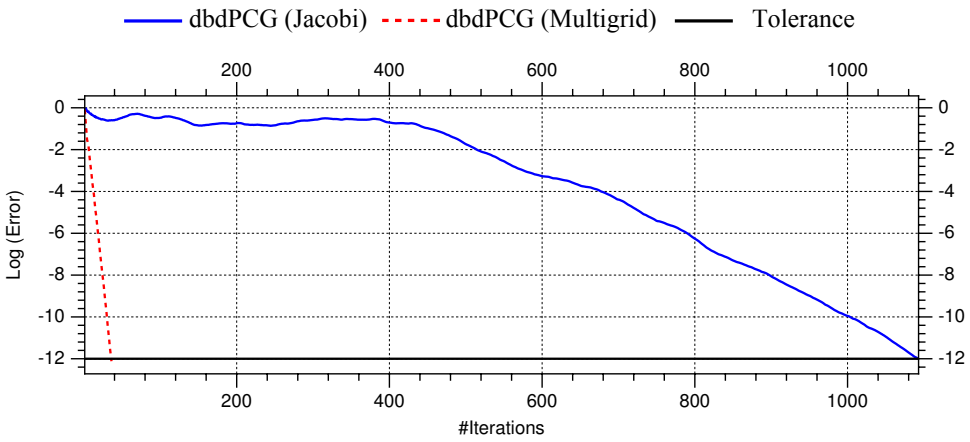
Table 2: Performance statistics for the benchmarks.

Electric mast					
	dbdPCG (sec)			nbnStress	ebeVol
	Jacobi**	Jacobi*	Multigrid**	(sec)	(sec)
CPU (Sparse)	108356.62	12049.59	2063.96	165.81	21.02
Quadro 4000	13098.81	4778.57	2058.61	69.81	3.39
Tesla C2070	8887.96	2814.34	1385.99	35.27	2.55
Tesla K40	6849.65	1648.65	1120.29	31.05	1.96
Tied-arch bridge					
	dbdPCG (sec)			nbnStress	ebeVol
	Jacobi**	Jacobi*	Multigrid**	(sec)	(sec)
CPU (Sparse)	461394.92	98376.13	14407.39	539.86	69.06
Quadro 4000	70153.70	25163.64	19564.32	232.58	10.49
Tesla C2070	36787.52	13904.94	11074.66	115.34	7.83
Tesla K40	25389.49	8387.96	7442.06	97.28	5.89
MBB beam					
	dbdPCG (sec)			nbnStress	ebeVol
	Jacobi**	Jacobi*	Multigrid**	(sec)	(sec)
Tesla K40	(–)	362454.28	168121.76	2100.06	77.3

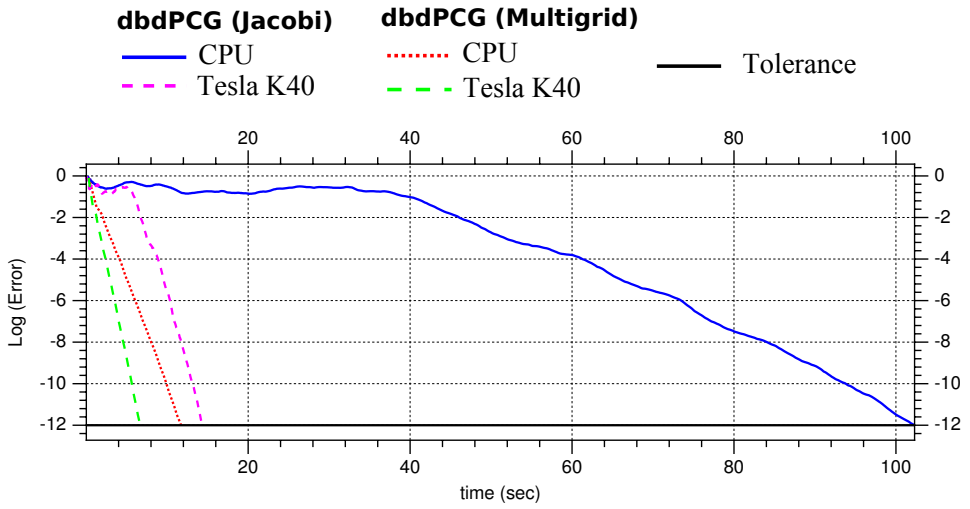
* Excluding outside O elements.

** Including outside O elements.

Table 3: Total wall-clock time for the topology optimization stages.

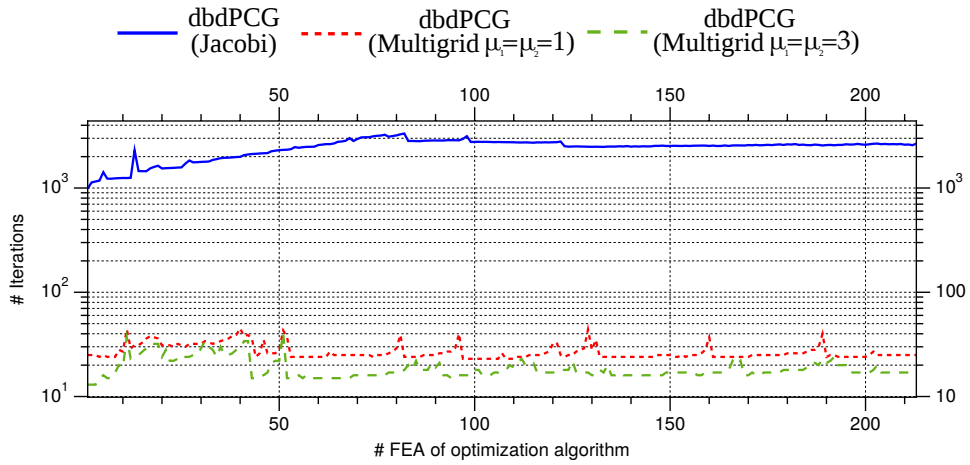


(a)

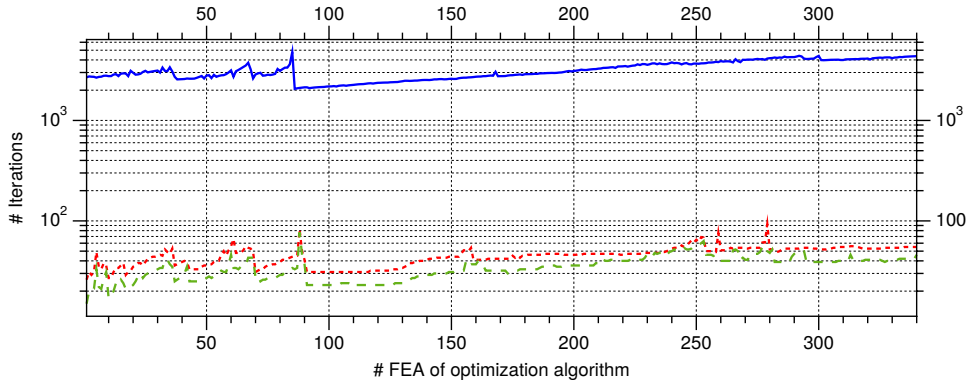


(b)

Figure 11: Convergence behavior of Jacobi PCG and geometric multigrid PCG iterative solvers for one FEA of the electric mast design.



(a)



(b)

Figure 12: Number of iterations required by Jacobi PCG and geometric multigrid PCG for all the FEAs of (a) electric mast and (b) tied-arch bridge designs.

precision floating-point format. The elements of the regular grid are eight-node hexahedral linear brick elements. The topology design parameters are also similar for all the benchmarks: $RF_0 = 10^{-3}$, $\Delta RF = 10^{-3}$, $\Delta V = 10^{-2}$. Table 2 shows some important parameters and performance statistics of the topology optimization experiments. The target volume V_T is about 3% of the volume of the design domain for the first two experiments and about 1.5% for the last one. Besides, the maximum residual error of PCG solver is set to 10^{-12} for the first two experiments and 10^{-8} for the last one. This configuration of the topology optimization requires a considerable number of FEAs to obtain the results.

Figure 7(b) and Figure 7(c) show the resulting structural design of the electric mast problem. The optimization algorithm leads to a truss-like design that resembles a real electric mast. Figure 8(b) and Figure 8(c) show the resulting structural design of the tied-arch bridge problem. This structural design resembles the typology of this kind of bridges, as shown in the real bridge of Figure 9. Figure 10(b) and Figure 10(c) show the resulting structural design of the MBB beam, which leads to a high resolution truss-like design.

The performance of solving the system of equations is evaluated using the Jacobi and the geometric multigrid preconditioning techniques. Besides, the performance of Jacobi preconditioner is evaluated including/excluding the outside O elements in the global system matrix of FGFEA. Figure 11(a) shows the convergence behavior (in terms of number of iterations) of PCG iterative solver using both preconditioning techniques for one FEA of the electric mast design with tolerance 10^{-12}. One can observe that the number of iterations to converge is reduced considerably when using the geometric multigrid preconditioner. On the other hand, Figure 11(b) shows the wall-clock time required to achieve the prescribed tolerance using the CPU and the proposed GPU instance with Tesla K40. Comparing Figure 11(a) and Figure 11(b), one can observe that the profile of Jacobi PCG is kept whereas the slope of geometric multigrid PCG is modified for the experiments using the CPU sparse-matrix based implementation. This is attributed to the fact that the wall-clock time per iteration of geometric multigrid PCG is higher than the Jacobi PCG in the CPU implementation. One

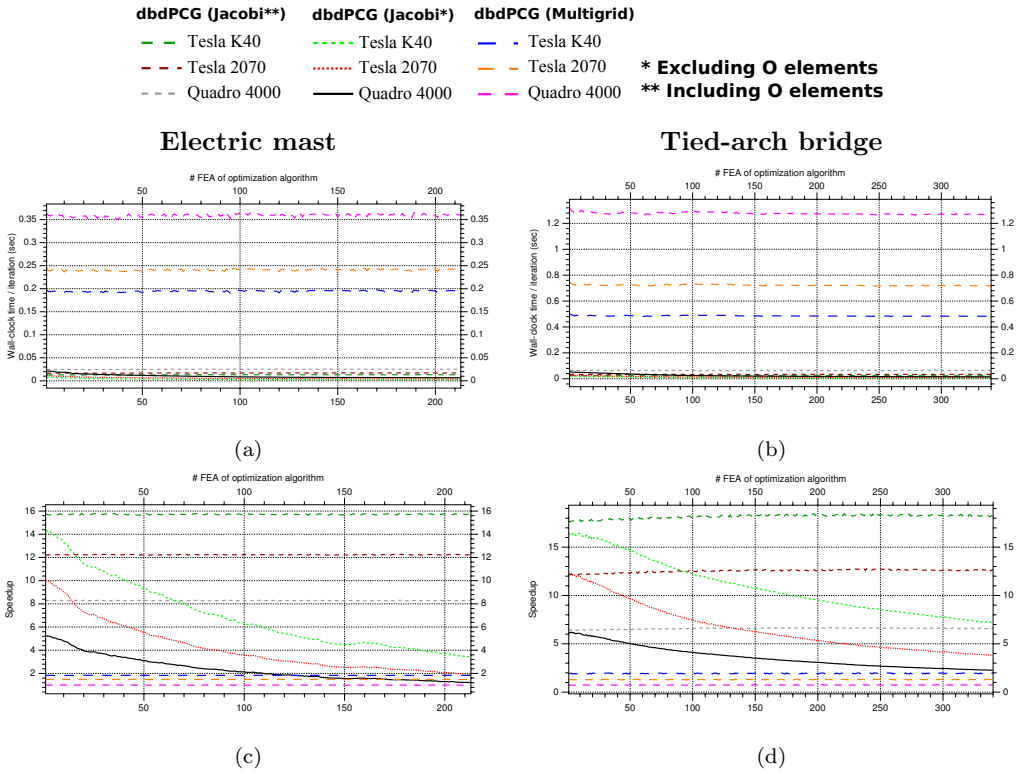
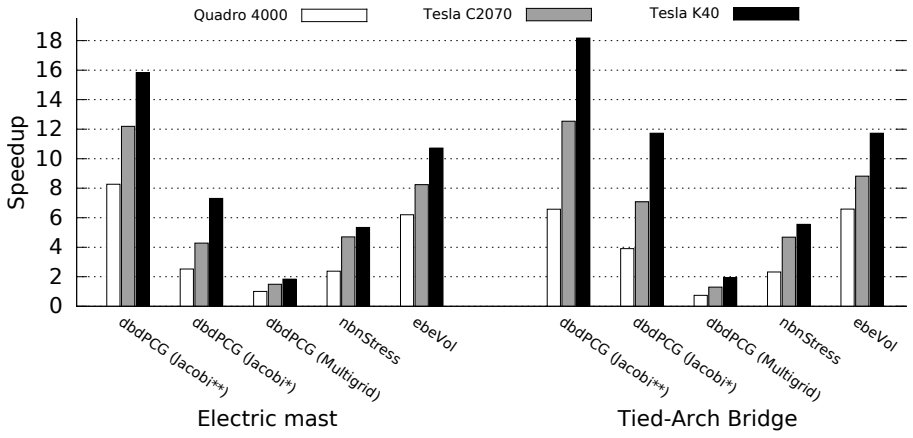


Figure 13: Wall-clock time per iteration of (a) electric mast and (b) tied-arch bridge benchmarks. Speedup per iteration of (c) electric mast and (d) tied-arch bridge benchmarks.



* Excluding outside O elements.

** Including outside O elements.

Figure 14: Speedup for benchmarks.

also can observe in Figure 11(b) that the speedup achieved in the GPU instance of the Jacobi PCG is significantly higher than the one obtained using multigrid PCG, and that the computation time for one FEA of the GPU instance of Jacobi PCG is only the double with respect to the GPU implementation of multigrid PCG for this example.

The number of iterations required by the PCG using both preconditioners for all the FEAs of the first two experiments is shown in Figure 12. One can observe how the PCG using the multigrid preconditioner requires a significantly lower number of iterations than the PCG using the Jacobi preconditioner. As expected, the number of iterations required by the multigrid solver fluctuate for each design cycle. This is due to the fact that convergence rate depends on the contrast of the stiffness distribution which changes as the design evolves. Nevertheless, the algorithm converges even for high-contrast layouts ($\triangle = 1e-6$). Besides, the variability of the number of iterations is relatively low for the PCG using the multigrid preconditioner; in particular, the iterations fluctuate between 35 and 50 for electric mast experiment and between 35 and around 100 for the tied-arch bridge experiment using $\mu_1 = \mu_2 = 1$. By increasing the number of pre- and post-smoothing steps these iterations can be reduced at the cost of increasing the computational cost per iteration.

The device memory required by the solving of the system of equations using both preconditioners and by the stress calculation is shown in Table 2. One can observe that the memory required by the PCG using the geometric multigrid preconditioner is much higher than the memory required by the PCG using the Jacobi preconditioner. This is mainly due to the storage of the I_e and I vectors and the assembled matrices of coefficients C for the coarser levels. Table 3 shows the wall-clock time for the topology optimization stages using diverse graphic cards, including the iterative solver using both preconditioners and the CPU implementation using sparse-matrix operations. It also shows the wall-clock time of the PCG using the Jacobi preconditioning excluding the O elements of the FGFEA to evaluate the improvement of this simplification. One can observe that the reduction in wall-clock time excluding the O elements of the FGFEA

is significant. However, the computation time is still considerably higher than adopting the geometric multigrid preconditioner for the experiments using the sparse-matrix CPU implementation.

On the other hand, the GPU instance of dbdPCG achieves higher speedups using the Jacobi preconditioner than using the geometric multigrid preconditioner. Indeed, the GPU instance on Tesla K40 provides computational times of similar order of magnitude for Jacobi PCG excluding outside O elements and multigrid PCG in the relatively small finite element models, such as the electric mast (800K DoFs) and the tied-arch bridge (2.2M Dofs) experiments. The MBB beam benchmark is only solved using the most modern GPU (Tesla K40) taking more than four days using the Jacobi preconditioner excluding the outside O element and almost two days using the multigrid preconditioner. The number of iterations per FEA required to solve this large model during the topology optimization is computationally prohibitive for the Jacobi preconditioner including the outside O elements. The wall-clock time for this large-scale model using the multigrid preconditioner is the half with respect to the Jacobi preconditioner excluding O elements of FGFEA. The wall-clock time of nbnStress and ebeVol are low order with respect to the solving of the system of equations. However, the efficient GPU implementation of these stages achieves significant speedups.

Figure 13 shows the speedup and wall-clock time per iteration of the GPU instances of solving stage during the topology optimization of the electric mast and the tied-arch bridge benchmarks. Figure 13(a) and Figure 13(b) show that the wall-clock time per iteration of FEA decreases during the optimization algorithm when the Jacobi PCG excluding the outside O elements is used. Such a reduction of wall-clock time also decreases the speedups during the topology optimization, as shown in Figure 13(c) and Figure 13(d). One also can observe that the wall-clock time per iteration of the FEAs using the geometric multigrid preconditioner is much higher than the wall-clock time per iteration using the Jacobi preconditioner. This is attributed to the higher amount of operations and the global memory accesses to device memory. However, the reduction of

iterations to converge compensates this increment of wall-clock time per iteration.

The averaged speedups of the computationally demanding tasks involved in the evolutionary topology optimization method driven by stress isosurfaces with different GPUs are shown in Figure 14. One can observe that the computationally intensive tasks of the topology optimization algorithm using the Jacobi preconditioner are accelerated significantly using the proposed GPU instance with respect to the classical implementation on CPU. Speedups between 7x and 19x are obtained for the solving of the system of equations using the Tesla K40 depending on the inclusion/exclusion of outside O elements. However, the speedups of the solving of the system of equations using the geometric multigrid PCG solver are around 2x for the Tesla K40. A key point is that the speedup of the GPU instance increases with the massive parallel capabilities of the graphics unit. The acceleration of the isosurface extraction and volume calculation tasks depends on the number of elements intersecting with the isosurface in the iterations of the optimization process. This speedup is relevant and similar for both experiments. The acceleration of the calculation of the design criteria distribution is similar for both benchmarks.

6. Conclusion

This paper has investigated about the proper strategy and techniques to achieve efficient calculation and reasonable speedups using GPU computing for the computationally intensive tasks of evolutionary topology optimization method driven by stress isosurfaces. Different granularities are used to facilitate the exploitation of massive parallel architectures. The solving of the system of equations, the von Misses stress calculation and the volume fraction calculation are implemented at the level of DoF, node and element, respectively. The common elemental stiffness matrix $\mathbf{K}_0^e$ and the result of the products $\mathbf{DB}^{(i)}$ are stored in the constant device memory only once at the beginning of the optimization to save memory bandwidth and exploit data locality. The numerical

results show the scalability of the proposed techniques with the resources of the graphics units. This is a promising result to achieve higher speedups on new generation of graphics cards or multi-GPU platforms.

The bottleneck of the topology optimization method is the solving of the system of equations of FEA. A comparison of the GPU implementation of PCG solver using Jacobi preconditioning and geometric multigrid preconditioning is provided. The former requires much iterations (and wall-clock time) to converge but a smaller amount of device memory. However, the wall-clock time is significantly reduced when outside O elements are excluded of the system of equations to solve. The latter reduces considerably the number of iterations to converge but these iterations have a higher computational cost in terms of wall-clock time and device memory requirements. The speedup using the geometric multigrid preconditioner is far lower than the speedups using Jacobi preconditioners including/excluding outside O elements. This is due to the GPU performance is deteriorated by the global memory accesses required by coarsening and coarser level operations. Therefore, diagonally preconditioned iterative solvers are a viable choice for not too large problems due to the reduced device memory requirements. Nevertheless, the geometric multigrid preconditioner presents better behavior for large-scale models, which permits to obtain high-resolution designs using evolutionary topology optimization algorithms.

Acknowledgment

We gratefully acknowledge the support of NVIDIA Corporation with the donation of some of the GPUs used for this research. Such a work has also been supported by the research support programmes of Ministry of Economy and Competitiveness under the contract DPI2016-77538-R and "Fundación Séneca – Agencia de Ciencia y Tecnología de la Región de Murcia" under the contract 19274/PI/14.

References

[1] M. P. Bendsøe, O. Sigmund, Topology Optimization – Theory, Methods, and Applications, second ed., Springer-Verlag Berlin Heidelberg, 2004.

[2] M. P. Bendsøe, N. Kikuchi, Generating optimal topologies in structural design using a homogenization method, Comput. Methods Appl. Mech. Eng. 71 (1988) 197–224.

[3] O. Sigmund, On the Design of Compliant Mechanisms Using Topology Optimization, Mechanics of Structures and Machines 25 (1997) 493–524.

[4] X. Huang, Y. Li, S. W. Zhou, Y. M. Xie, Topology optimization of compliant mechanisms with desired structural stiffness, Eng. Struct. 79 (2014) 13–21.

[5] A. Iga, S. Nishiwaki, K. Izui, M. Yoshimura, Topology optimization for thermal conductors considering design-dependent effects, including heat conduction and convection, Int. J. Heat Mass Transfer 52 (2009) 2721–32.

[6] G. H. Yoon, J. S. Jensen, O. Sigmund, Topology optimization of acousticstructure interaction problems using a mixed finite element formulation, Int. J. Numer. Methods Eng. 70 (2007) 1049–75.

[7] L. Shu, M. Y.Wang, Z. Ma, Level set based topology optimization of vibrating structures for coupled acoustic-structural dynamics, Comput. Struct. 132 (2014) 34–42.

[8] J. D. Deaton, R. V. Grandhi, A survey of structural and multidisciplinary continuum topology optimization: post 2000, Struct. Multidiscip. Optim. 49 (2014) 1–38.

[9] O. Sigmund, K. Maute, Topology optimization approaches: A comparative review, Struct. Multidiscip. Optim. 48 (2013) 1031–55.

[10] M. P. Bendsøe, Optimal shape design as a material distribution problem, Struct. Optim. 1 (1989) 193–202.

[11] M. Zhou, G. I. N. Rozvany, The COC algorithm, part II: topological, geometrical and generalized shape optimization, Comput. Methods Appl. Mech. Eng. 89 (1991) 309–36.

[12] N. P. van Dijk, K. Maute, M. Langelaar, F. van Keulen, Level-set methods for structural topology optimization: a review, Struct. Multidiscip. Optim. 48 (2013) 437–72.

[13] M. Burger, R. Stainko, PhaseField Relaxation of Topology Optimization with Local Stress Constraints, SIAM J. Control Optim. 45 (2006) 1447–66.

[14] N. P. van Dijk, K. Maute, M. Langelaar, F. van Keulen, Shape and topology optimization based on the phase field method and sensitivity analysis, J. Comput. Phys. 229 (2010) 2697–718.

[15] J. Sokolowski, A. Zochowski, On the Topological Derivative in Shape Optimization, SIAM J. Control Optim. 37 (1999) 1251–72.

[16] D. J. Munk, G. A. Vio, G. P. Steven, Topology and shape optimization methods using evolutionary algorithms: a review, Struct. Multidiscip. Optim. 52 (2015) 613–31.

[17] Y. M. Xie, G. P. Steven, A simple evolutionary procedure for structural optimization, Comput. Struct. 49 (1993) 885–96.

[18] P. Tanskanen, The evolutionary structural optimization method: theoretical aspects, Comput. Methods Appl. Mech. Eng. 191 (2002) 5485–98.

[19] O. M. Querin, G. P. Steven, Y. M. Xie, Evolutionary structural optimisation (ESO) using a bidirectional algorithm, Eng. Computations 15 (1998) 1031–48.

[20] X. Huang, Y. M. Xie, Evolutionary Topology Optimization of Continuum Structures: Methods and Applications, John Wiley & Sons, Ltd, United Kingdom, 2010.

[21] V. Young, O. M. Querin, G. P. Steven, Y. M. Xie, 3D and multiple load case bi-directional evolutionary structural optimization (BESO), Struct. Optim. 18 (1999) 183–92.

[22] R. Ansola, E. Vegueria, J. Canales, J. Tarrago, A simple evolutionary topology optimization procedure for compliant mechanism design, Finite Elem. Anal. Des. 44 (2007) 53–62.

[23] T. Borrvall, J. Petersson, Large-scale topology optimization in 3D using parallel computing, Comput. Methods Appl. Mech. Eng. 190 (2001) 6201–29.

[24] K. Vemaganti, W. E. Lawrence, Parallel methods for optimality criteria-based topology optimization, Comput. Methods Appl. Mech. Eng. 194 (2005) 3637–67.

[25] N. Aage, E. Andreassen, B. S. Lazarov, Topology optimization using PETSc: An easy-to-use, fully parallel, open source topology optimization framework, Struct. Multidiscip. Optim. 51 (2015) 565–72.

[26] J. Martínez-Frutos, P. J. Martínez-Castejón, D. Herrero-Peréz, Fine-grained GPU implementation of assembly-free iterative solver for finite element problems, Comput. Struct. 157 (2015) 9–18.

[27] A. R. Brodtkorb, T. R. Hagen, M. L. Sætra, Graphics processing unit (GPU) programming strategies and trends in GPU computing, J. Parallel Distrib. Comput. 73 (2013) 4–13.

[28] G. Pratx, L. Xing, GPU computing in medical physics: A review, Med. Phys. 38 (2011) 2685–97.

[29] E. Wadbro, M. Berggren, Megapixel Topology Optimization on a Graphics Processing Unit, SIAM Rev. 51 (2009) 707–21.

[30] S. Schmidt, V. Schulz, A 2589 line topology optimization code written for the graphics card, Comput. Vis. Sci. 14 (2011) 249–56.

[31] K. Suresh, Efficient generation of large-scale pareto-optimal topologies, Struct. Multidiscip. Optim. 47 (2013) 49–61.

[32] V. Challis, A. Roberts, J. Grotowski, High resolution topology optimization using graphics processing units (GPUs), Struct. Multidiscip. Optim. 49 (2014) 315–25.

[33] F. J. Ramírez-Gil, E. C. Nelli-Silva, W. Montealegre-Rubio, Topology optimization design of 3D electrothermomechanical actuators by using GPU as a co-processor, Comput. Methods Appl. Mech. Eng. 302 (2016) 44–69.

[34] J. Wu, C. Dick, R. Westermann, A System for High-Resolution Topology Optimization, IEEE Trans. Visual Comput. Graphics 22 (2016) 1195–208.

[35] C. Dick, J. Georgii, R. Westermann, A Real-Time Multigrid Finite Hexahedra Method for Elasticity Simulation using CUDA, Simulation Modelling Practice and Theory 19 (2011) 801–16.

[36] M. J. Garcia, O. Ruiz, G. P. Steven, Engineering design using evolutionary structural optimization based on iso-stress-driven smooth geometry removal, in: Proc. of NAFEMS World Congress, 2001.

[37] M. Victoria, P. Marti, O. Querin, Topology design of two-dimensional continuum structures using isolines, Comput. Struct. 87 (2009) 101–9.

[38] J. Martínez-Frutos, D. Herrero-Peréz, Efficient Matrix-free GPU implementation of Fixed Grid Finite Element Analysis, Finite Elem. Anal. Des. 104 (2015) 61–71.

[39] J. Martínez-Frutos, D. Herrero-Peréz, Large-scale robust topology optimization using multi-GPU systems, Comput. Methods Appl. Mech. Eng. 311 (2016) 393–414.

[40] G. M. Amdahl, Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities, in: AFIPS Conference Proceedings, volume 30, 1967, pp. 483–5.

[41] M. Victoria, O. M. Querin, P. Martí, Topology design of three-dimensional continuum structures using isosurfaces, Adv. Eng. Software 42 (2011) 671–9.

[42] V. R. Voller, C. R. Swaminathan, B. G. Thomas, Fixed grid techniques for phase change problems: a review, Int. J. Numer. Methods Eng. 30 (1990) 875–98.

[43] M. J. García-Ruíz, G. P. Steven, Fixed grid finite elements in elasticity problems, Engineering Computations 16 (1999) 145–64.

[44] H. Kim, M. J. Garcia, O. M. Querin, G. P. Steven, Y. M. Xie, Introduction of fixed grid in evolutionary structural optimisation, Eng. Computations 17 (2000) 427–39.

[45] O. C. Zienkiewicz, R. L. Taylor, J. Z. Zhu, The Finite Element Method: Its Basis and Fundamentals, Elsevier Butterworth Heinemann, Oxford, 2013.

[46] H. Kim, O. M. Querin, G. P. Steven, Y. M. Xie, Improving efficiency of evolutionary structural optimization by implementing fixed grid mesh, Struct. Multidiscip. Optim. 24 (2003) 441–8.

[47] W. E. Lorensen, H. E. Cline, Marching cubes: a high resolution 3D surface construction algorithm, Comput. Graph. 21 (1987) 163–9.

[48] T. S. Newman, H. Yi, A survey of the marching cubes algorithm, Comp. Graph. 30 (2006) 854–79.

[49] M. Dürst, Letters: Additional Reference to Marching Cubes, ACM SIGGRAPH Computer Graphics 22 (1988) 72–3.

[50] G. Nielson, On marching cubes, IEEE Trans. Visual Comput. Graphics 9 (2003) 283–97.

[51] M. J. de Ruiter, F. van Keulen, Topology optimization using a topology description function, Struct. Multidiscip. Optim. 26 (2004) 406–16.

[52] M. H. Hsu, Y. L. Hsu, Interpreting three-dimensional structural topology optimization results, Comput. Struct. 83 (2005) 327–37.

[53] X. Huang, Y. M. Xie, M. C. Burry, A New Algorithm for BiDirectional Evolutionary Structural Optimization, JSME Int J., Ser. C 49 (2006) 1091–9.

[54] S. Ashby, R. Falgout, A Parallel Multigrid Preconditioned Conjugate Gradient Algorithm for Groundwater Flow Simulations, Nucl. Sci. Eng. 124 (1996) 145–59.

[55] W. Briggs, V. Henson, S. McCormick, A Multigrid Tutorial, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 2000.

[56] R. S. Sampath, G. Biros, A Parallel Geometric Multigrid Method for Finite Elements on Octree Meshes, SIAM J. Sci. Comput. 32 (2010) 1361–92.

[57] O. Amir, N. Aage, B. S. Lazarov, On multigrid-CG for efficient topology optimization, Struct. Multidiscip. Optim. 49 (2014) 815–29.