# Efficient topology optimization using GPU computing with multilevel granularity

Jesús Martínez-Frutos, Pedro J. Martínez-Castejón, David Herrero-Pérez*

*Computational Mechanics & Scientific Computing Group, Department of Structures and Construction, Technical University of Cartagena, Campus Muralla del Mar, 30202 Cartagena, Murcia, Spain*

---

**Abstract**

This paper proposes a well-suited strategy for High Performance Computing (HPC) of density-based topology optimization using Graphics Processing Units (GPUs). Such a strategy takes advantage of Massively Parallel Processing (MPP) architectures to overcome the computationally demanding procedures of density-based topology design, both in terms of memory consumption and processing time. This is done exploiting data locality and minimizing both memory consumption and data transfers. The proposed GPU instance makes use of different granularities for the topology optimization pipeline, which are selected to properly balance the workload between the threads exploiting the parallelization potential of massive parallel architectures. The performance of the fine-grained GPU instance of the solving stage is evaluated using two preconditioning techniques. The proposal is also compared with the classical CPU implementation for diverse topology optimization problems, including stiffness maximization, heat sink design and compliant mechanism design.

*Keywords:* GPU computing, Topology optimization, Compliance, Compliant mechanism, Heat conduction.

---

*Corresponding author

*Email addresses:* `jesus.martinez@upct.es` (Jesús Martínez-Frutos), `pedro.castejon@upct.es` (Pedro J. Martínez-Castejón), `david.herrero@upct.es` (David Herrero-Pérez)

## 1. Introduction

Topology optimization aims to find the optimal distribution of material within a design domain such that an objective function is minimized under certain constraints [1]. Contrary to size and shape optimization methods, topology optimization permits to obtain a material distribution without assuming any prior structural configuration. This provides engineering designers with a powerful tool to find innovative and high-performance conceptual designs at the early stages of the design process. Not to mention the great impact of the optimization of geometry and topology on the structural performance. This problem has sparked a broad interest since the early work of Bendsøe and Kikuchi [2], giving rise to a multitude of studies in a wide range of physics problems, such as stiffness maximization of structures [3], design of compliant mechanisms [4], maximization of temperature diffusivity [5], and minimization of acoustic emission [6], to name but a few [7].

Shape and topology optimization methods can be broadly classified into three main categories depending on the representation used to describe the shapes they involve: *density-based methods*, *Eulerian methods* and *Lagrangian methods*. The methods included in the first category operate on a fixed grid of finite elements and seek an optimal void/solid material distribution that minimizes an objective function. The homogenization method [2, 8] and the Solid Isotropic Material Penalization (SIMP) method [1, 9] are some examples of the most popular topology optimization approaches included in this category. The second category is composed of methods that use an implicit representation of the structural boundary. Such a boundary can be modified by tracking the motion of a level-set function, as is done in the Level-Set Method (LSM) [10, 11, 12], or by evolving the interfacial dynamics of phase field equations, as occurs in the phase field models [13]. The third category is composed of methods that use an explicit representation of the structural shape by means of a computational mesh or CAD model [14, 15]. This work deals with the efficient computation of density-based topology optimization methods using GPU computing.

Despite the great advances made in theory and practical application of topology optimization in the past decade, the computational requirements still remain as a primary challenge [7]. This is due to some demanding tasks involved in the topology optimization pipeline, such as the solving of large systems of equations, the computation of sensitivities and the filtering strategy. Such tasks may increase meaningfully the computation time of the topology optimization process, which may takes hours or even days for relatively large models. High Performance Computing (HPC) is then needed to address the topology optimization process, normally making use of task-level parallel computing to address the computationally intensive tasks [16, 17, 18].

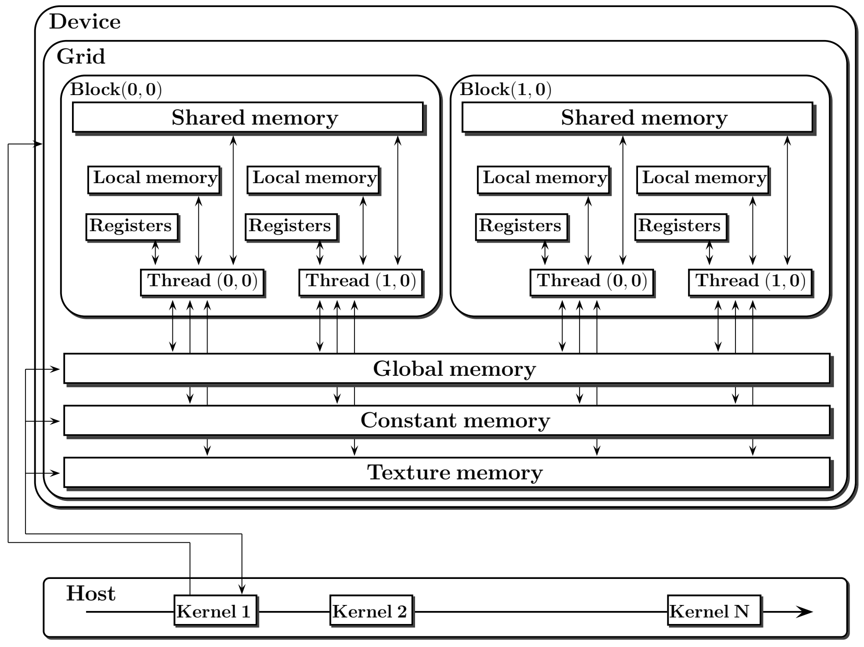
The use of Graphics Processing Units (GPUs) for non-graphics applications is rapidly growing in popularity [19, 20]. This is due to the high computing capacity of these graphics cards for Massively Parallel Processing (MPP) at reasonable cost. GPU computing consists of the use of a GPU together with a CPU to accelerate compute intensive applications. This is not a simple goal since 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 [21], which constraint severely the GPU performance.

GPU computing has been successfully used in diverse engineering and scientific problems requiring numerical analysis. One can mention the acceleration of the solving of parametric integral equations in elasticity [22], system of equations in finite element problems [23] and peridynamic systems in peridynamic models of solid mechanics [24, 25, 26]. These graphics devices are also successfully used for real-time simulation and haptic feedback of soft tissue deformations [27, 28], which are especially useful for the development of realistic simulators. Besides, relevant results are obtained for the structural solver of finite element explicit dynamics problems using GPU computing [29]. These graphics cards have also shown promising results in heterogeneous systems addressing large-scale problems in Finite Element Analysis (FEA) [30]. The use of these devices to speedup computationally demanding tasks in the topology optimization pipeline has sparked a broad interest last years, giving rise to several studies.

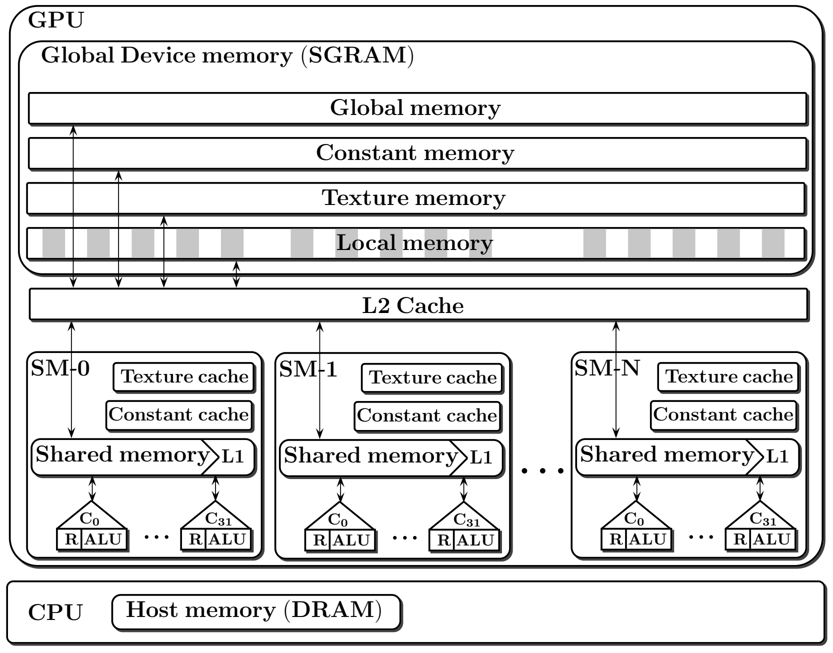
The early work of Wadbro and Berggren [31] aims to solve large topology optimization problems using a gradient-based optimality criterion method. This early work implements a Preconditioned Conjugate Gradient (PCG) method on GPU to solve high resolution finite element models arising in heat conduction 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 implementation for the solver of the SIMP method is proposed in [32]. Applied to the minimization of the structural compliance problem, this GPU instance achieves significant speedups following the strategy of loading three successive 2D slices of the third dimension into shared memory to perform the computations required by the middle slice efficiently. Such a slice-wise and nodal-based strategy is also adopted in [33] achieving speedups of one order of magnitude for the solving of the system of equations of elasticity. GPU computing is also used to increase the tractable computational resolution of topology optimization problems using discrete level-set methods [34] and evolutionary structural optimization methods [35].

GPU computing using the sparse-matrix representation permits to efficiently assembly and solve the system of equations of elasticity [36]. 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 accesses 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 the global device memory, which affects seriously the GPU performance. For these reasons, matrix-free GPU implementations 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. [37] 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]. The GPU instance is based on the node-wise GPU parallelization proposed by Dick et al. [38], where the grid regularity is exploited to perform coarsening and matrix-vector 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 density-based topology optimization methods. On the one hand, a fine-grained GPU implementation of matrix-free PCG solver for structural analysis is adopted. The regularity of the grid permits to exploit data locality maximizing the GPU performance for FEA [39]. The granularity of matrix-vector multiplication operations is at the Degree of Freedom (DoF) level, which allows reducing and balancing the workload for all the threads of the MPP architecture [23, 40]. Another key point for increasing the GPU performance is that matrix-vector multiplication operations are launched by three-dimensional kernels using cache data in shared memory for the corresponding three-dimensional blocks. This strategy permits to increase the use of on-chip memory, i.e. the cache data in shared memory, by the threads performing the operations for the corresponding DoF. This achieves a significant improvement for solving the system of equations using GPU computing. The performance of the GPU instance for the solving stage is evaluated in terms of speedup and wall-clock time analyzing two preconditioning techniques; in particular, Jacobi



(a)



(b)

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

preconditioner and geometric multigrid preconditioner. On the other hand, the calculation of sensitivities, filter and density update are also implemented using GPU computing in order not to limit the theoretical speedup according to Amdahl's law [41]. The granularity of such tasks is at the finite element level due to the nature of the operations do not allow us to reduce the grain size, which usually improves the GPU performance. The proposed matrix-free GPU instance is compared to the classical sparse-matrix CPU implementation for diverse topology optimization problems, including stiffness maximization, heat sink design and compliant mechanism design. The speedups and relative wall-clock time in density-based topology optimization pipeline is also studied for such topology optimization problems.

The paper is organized as follows. Section 2 provides an overview of the GPU architecture and the CUDA programming model. The bases and the theoretical background of density-based topology optimization methods are briefly reviewed in section 3. The proposed GPU implementation of SIMP method is presented in section 4. Section 5 is devoted to the numerical experiments and the performance evaluation of the proposed matrix-free GPU instance with respect to the classical sparse-matrix CPU implementation. Finally, the conclusion of the proposed GPU instance is presented in section 6.

## 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 at a reasonable cost. Currently, the use of Nvidia devices and its programming model, Compute Unified Device Architecture (CUDA) [42], is the prevailing tendency, which is adopted in the developments presented in this work. Such a programming model allows 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) taken from [43], should specify the 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 (R) per processor (C) and a shared memory, a read-only constant cache and a read-only texture cache. These memory resources are shared by all cores ($\mathrm{C_i}$) of such a SM. This fact implies that the amount of blocks that 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. For these reasons, the proposed GPU instance uses constant memory for storing the common elemental stiffness matrix, whereas the use of shared memory is limited to unknowns and elemental densities. Such a strategy achieves a significant GPU performance for the calculations involved in density-based topology optimization.

The software developments using CUDA consist in 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. Density-based topology optimization

Topology optimization can be defined as a binary programming problem that aims to find the optimal material layout (solid and void) that minimizes an objective function. Such a material layout should satisfy a set of prescribed constraints in the design domain. Density-based methods are the most widely used topology optimization methods due to its conceptual simplicity, which has facilitated its application in industrial software [44]. In these methods, the integer-based topology optimization problem is relaxed to a formulation based on artificial continuous material densities, which permits the use of gradient-based solvers. The topology optimization problem can be stated as

$$\begin{aligned} \min_{\boldsymbol{\rho}} \quad & f(\boldsymbol{\rho}, \mathbf{u}) \\ \text{s. t.}: \quad & \mathbf{K}(\boldsymbol{\rho})\mathbf{u} = \mathbf{f} \\ : \quad & V(\boldsymbol{\rho}) \leq V^* \\ : \quad & 0 \leq \boldsymbol{\rho}(\mathbf{x}) \leq 1, \ \mathbf{x} \in \mathscr{D} \end{aligned} \tag{1}$$

where $f$ is the objective function, $\boldsymbol{\rho}$ is the vector of density design variables, $\mathbf{u}$ is the system response, $\mathbf{K}$ is the global stiffness matrix, $\mathbf{f}$ is the force vector and $\mathbf{x}$ is the vector of finite elements. The design domain is denoted by $\mathscr{D}$ and the volume of material $V(\boldsymbol{\rho})$ is constrained to be smaller than a prescribed target $V^*$. The unknown densities, $\boldsymbol{\rho}(\mathbf{x})$, are used to scale the stiffness of the finite elements of the regular grid. In practice, this parametrization leads to designs with large areas of intermediate densities which, even though being numerical optimal, are impossible to manufacture. This problem is normally addressed using implicit relaxation/penalization techniques, which drive the topology design towards solid/void configurations. The SIMP method [9, 45] makes use of such implicit penalization techniques by a power-law interpolation function between void and solid to determine the stiffness matrix of each element $\mathbf{K}_e$ as follows

$$\mathbf{K}_e = \mathbf{K}_{min} + {\rho_e}^p \left(\mathbf{K}_0 - \mathbf{K}_{min}\right), \tag{2}$$

where $\mathbf{K}_0$ and $\mathbf{K}_{min} > 0$ are the stiffness matrix of solid and void material respectively, and $p > 1$ is the penalization power. For problems where the volume constraint is active, Bendsøe and Sigmund [46] prove that the power-law interpolation function is perfectly valid when p is sufficiently large. In particular, p $\geq 3$ is usually required to obtain black-and-white designs.

Although the use of material interpolation schemes enables to obtain almost solid-and-void designs, they destroy the convexity of the optimization problem increasing the risk of ending in local minima. However, it is common to use *continuation methods* to mitigate the premature convergence to local minima when solving the optimization problem, see e.g. [47]. According to [48], continuation methods take "global" information into account and are more likely to ensure "global" convergence or at least convergence to better designs. Different continuation methods have been proposed based on the idea of gradually change the optimization problem from a convex problem to the original non-convex problem.

The topology optimization problem should also be regularized using additional constraints on the density field to avoid numerical difficulties and modeling problems, such as mesh-dependency of solutions and checker-board patterns [1] respectively. The sensitivity filter [4] is adopted in this work because it has proven to be effective in practice producing mesh-independent solutions. Moreover, the filtering of gradients has a continuum mechanics motivation and may promote convergence of some length scales over others, and thereby speeds up convergence [49]. Furthermore, the sensitivity filter has computational advantages because it is not included in the OC updating scheme loop. One drawback is, however, that there remain discrepancies between the filtered sensitivities and the actual sensitivities, i.e. the modified sensitivities do not completely correspond to the objective function. In theory, this may lead to some divergence problems though proper designs are obtained in practice. The sensitivity filter applies a smoothing filter to the derivatives of the objective function as follows

$$\widehat{\frac{\partial f(\rho)}{\partial \rho_e}} = \frac{\sum\limits_{i \in NB_e} w(\mathbf{x}_i, \mathbf{x}_e)\rho_i \dfrac{\partial f(\rho)}{\partial \rho_i}}{max(\gamma, \rho_e) \sum\limits_{i \in NB_e} w(\mathbf{x}_i, \mathbf{x}_e)}, \tag{3}$$

where $NB_e$ is the neighborhood set of an element $e$, $w(\mathbf{x}_i, \mathbf{x}_e)$ is a weighting function and $\gamma > 0$ is a small value to prevent the division by zero. The linear weighting function is used in this work, which is defined as

$$w(\mathbf{x}_i, \mathbf{x}_e) = \begin{cases} R - ||\mathbf{x}_i - \mathbf{x}_e|| & \text{if} ||\mathbf{x}_i - \mathbf{x}_e|| \leq R \\ 0 & \text{if} ||\mathbf{x}_i - \mathbf{x}_e|| > R \end{cases},$$

whereas the neighborhood set of an element $e$ is defined as

$$NB_e := \{i \mid dist(i,e) \leq R\}, \tag{4}$$

where $R$ is the filter size and $dist(i,e)$ is the Euclidean distance between the center of element $i$ and the center of element $e$.

The SIMP method can be applied to diverse physics problems. The numerical experiments of this work address the stiffness maximization of continuum structures, the heat sink design cooled by heat conduction and the compliant mechanism design problems. The objective function for the minimization of structural compliance (maximization of stiffness) and the minimization of thermal compliance (maximization of heat transfer) is given by

$$f = c = \mathbf{f}^{\mathrm{T}}\mathbf{u}, \tag{5}$$

whereas the objective function for the compliant mechanism design, consisting of the maximization of output displacements, is as follows

$$f = -u_{out} = -\mathbf{l}^{\mathrm{T}}\mathbf{u}, \tag{6}$$

where $\mathbf{f}$ and $\mathbf{u}$ are the global force/thermal load and displacement/temperature vectors respectively for elasticity/heat transfer problems, and $\mathbf{l}$ is a vector with ones at the Degrees of Freedom (DoF) of the output displacements $u_{out}$ and zeros in all other positions. Considering the discretized linear state system $\mathbf{Ku} = \mathbf{f}$ and using the adjoint state method, the sensitivity of (5) and (6) with respect to the state variables $\boldsymbol{\rho}$ is as follows

$$\mathbf{f}_\rho = \frac{\partial f}{\partial \boldsymbol{\rho}} = -\mathbf{u}^{*\mathrm{T}}\frac{\partial \mathbf{K}}{\partial \boldsymbol{\rho}}\mathbf{u} = -\mathbf{u}^{*\mathrm{T}}(p\boldsymbol{\rho}^{p-1}\left(\mathbf{K}_0 - \mathbf{K}_{min}\right)\mathbf{u}, \tag{7}$$

where $\mathbf{u}^*$ is given by the solution of the adjoint problem as follows

$$\mathbf{K}\mathbf{u}^* = \frac{\partial f}{\partial \mathbf{u}}. \tag{8}$$

The right hand side of the adjoint problem is $\partial f/\partial \mathbf{u} = \mathbf{f}$ for the minimization of both structural and thermal compliance, which means that these problems are self-adjoint and the solution of (8) is $\mathbf{u}^* = \mathbf{u}$. Conversely, the right hand side of (8) is $\partial f/\partial \mathbf{u} = \mathbf{l}$ for the compliant mechanism design, which requires the solving of the adjoint system (8) to obtain $\mathbf{u}^*$.

The sensitivities given by (7) permit to update the design variables $\boldsymbol{\rho}$ using sequential convex approximations, such as the Sequential Quadratic Programming (SQP) [50] and the Method of Moving Asymptotes (MMA) [51]. The Optimality Criterion (OC) updating scheme proposed by [52] and modified by [53] is adopted in this work due to its numerical efficiency. The OC updating scheme is as follows

$$\rho_{e_{k+1}} = \begin{cases} \max\{(1-\zeta)\rho_{e_k}, 0\} & \text{if} \ \ \rho_{e_k} B_{e_k}^{\eta} \leq \max\{(1-\zeta)\rho_{e_k}, 0\}, \\ \min\{(1+\zeta)\rho_{e_k}, 1\} & \text{if} \ \ \min\{(1+\zeta)\rho_{e_k}, 1\} \leq \rho_{e_k} B_{e_k}^{\eta}, \\ (\rho_{e_k} B_{e_k}^{\eta})^q & \text{otherwise,} \end{cases} \tag{9}$$

where $\zeta$ is a positive step width, $\eta$ is a numerical damping coefficient, $q$ is a penalty factor to further achieve black-and-white topologies (typically $q = 2$) and

$$B_{e_k} = -\frac{\partial f(\boldsymbol{\rho})}{\partial \rho_e} \left( \lambda \frac{\partial V(\boldsymbol{\rho})}{\partial \rho_e} \right)^{-1} = 1 \tag{10}$$

is the Karush-Kuhn-Tucker (KKT) optimality condition. The Lagrange multiplier $\lambda$ is found using the bisection method. The algorithm stops when the maximum number of iterations is reached or when the change variable $||\rho_{e_{k+1}} - \rho_{e_k}||_\infty$ and the change in the objective function $|f_{k+1} - f_k|$ fall below a prescribed value.

The FEA is the principal bottleneck of the topology optimization pipeline. This stage involves two computational intensive tasks: the assembly of the local element equations into a global system of equations and the solving of such a system. These computational intensive tasks can lead to an unaffordable problem in terms of computation time and memory consumption. This problem is exacerbated when dealing with large-scale models [54] or when the system response needs to be re-evaluated, as occurs in topology optimization. Iterative solvers and assembly-free methods have been extensively used for reducing the memory requirements of FEA at the cost of increasing the processing time of the solve step, which is commonly alleviated using parallel computing.

## 4. GPU implementation of SIMP method

GPU computing is used to accelerate the computationally intensive tasks involved in the SIMP method. Such tasks are shown in the flowchart depicted in Figure 3; in particular, the Finite Element Analysis (FEA), the calculation of the sensitivities, the filtering strategy and the OC updating scheme. The custom-developed CUDA kernels and the techniques adopted for the efficient implementation of these computationally intensive tasks on GPU architectures are detailed below.

### *4.1. Finite Element Analysis (FEA)*

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 geometric multigrid 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 [55]. However, the performance of these methods deteriorates with increasing contrast in material properties [56]. This is attributed to the coarsening across discontinuities which affects to the coarse grid correction [57]. Nevertheless, the use of geometric multigrid as preconditioning technique shows good convergence rates for topology optimization problems using a sufficiently strong smoothing operator [58].

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 elemental properties $\mathbf{d} = E_{void} + \boldsymbol{\rho}^p(E_{solid} - E_{void})$ (for simplicity) for each analysis. This reduces meaningfully the use of device memory and permits to exploit the data locality [39]. 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. In particular, 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.

**Algorithm 1:** DbD PCG algorithm (dbdPCG)

```
Data: $\mathbf{K}_0^e$, $\mathbf{f}_0$, $\mathbf{u}_0$, $\mathbf{d}$, $tol$, $k_{max}$, $\mu_1$, $\mu_2$, $n_\ell$, $\omega$, $\mathbf{I_c}$, $\mathbf{I}$, $\mathbf{C}$, $DoF_n$, $n_x$, $n_y$, $n_z$
Result: 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
$\mathbf{r} \leftarrow$ dbdMVP($\mathbf{u}$, $\mathbf{d}$, $\mathbf{K}_0^e$, $\ell$, $\mathbf{I_c}$, $\mathbf{I}$, $\mathbf{C}$, $DoF_n$, $n_x$, $n_y$, $n_z$);     // $r = K^e u$

$u \leftarrow threadId + BlockDim \times BlockId$;                    // CUDA kernel (dbdKer1)
if ($u < N_{DoF}$) then
    $r^{(u)} \leftarrow f^{(u)} - r^{(u)}$;
end
if Multigrid then                                                      // Multigrid preconditioning
    $\mathbf{z} \leftarrow$ VCycle($\mathbf{K}_0^e, \mathbf{r}, \mathbf{d}, \ell, n_\ell, \omega, \mathbf{I_c}, \mathbf{I}, \mathbf{C}, \mu_1, \mu_2, n_x, n_y, n_z, DoF_n$);
else if Jacobi then                                                    // Jacobi preconditioning
    $\mathbf{z} \leftarrow$ dbdJacP($\mathbf{d}, \mathbf{K}_0^e$);
end

$u \leftarrow threadId + BlockDim \times BlockId$;                    // CUDA kernel (dbdKer2)
if ($u < N_{DoF}$) then
    $p^{(u)} \leftarrow r^{(u)}$;
    $b^{(u)} \leftarrow z^{(u)} r^{(u)}$;
    $c^{(u)} \leftarrow f^{(u)} f^{(u)}$;
end

$\rho_0 \leftarrow \rho_0 + \sum_{u=0}^{N_{DoF}-1} b^{(u)}$;                  // Reduction using thrust library
$\gamma_0 \leftarrow \gamma_0 + \sum_{u=0}^{N_{DoF}-1} c^{(u)}$;
while ( $\sqrt{\rho_k} > tol \cdot \sqrt{\gamma_k}$ ) and ( $k < k_{max}$ ) do
    $k \leftarrow k + 1$;
    $\mathbf{a} \leftarrow$ dbdMVP($\mathbf{p}$, $\mathbf{d}$, $\mathbf{K}_0^e$, $\ell$, $\mathbf{I_c}$, $\mathbf{I}$, $\mathbf{C}$, $DoF_n$, $n_x$, $n_y$, $n_z$);    // $a = K^e p$
    $\phi_k \leftarrow 0$;

    $u \leftarrow threadId + BlockDim \times BlockId$;                // CUDA kernel (dbdKer3)
    if ($u < N_{DoF}$) then
        $b^{(u)} \leftarrow a^{(u)} p^{(u)}$;
    end

    $\phi_k \leftarrow \phi_k + \sum_{u=0}^{N_{DoF}-1} b^{(u)}$;              // Reduction using thrust library
    $\alpha_k \leftarrow \rho_{k-1} / \phi_k$;

    $u \leftarrow threadId + BlockDim \times BlockId$;                // CUDA kernel (dbdKer4)
    if ($u < N_{DoF}$) then
        $u^{(u)} \leftarrow u^{(u)} + \alpha_k p^{(u)}$;
        $r^{(u)} \leftarrow r^{(u)} - \alpha_k a^{(u)}$;
    end
    if Multigrid then                                                  // Multigrid preconditioning
        $\mathbf{z} \leftarrow$ VCycle($\mathbf{K}_0^e, \mathbf{r}, \mathbf{d}, \ell, n_\ell, \omega, \mathbf{I_c}, \mathbf{I}, \mathbf{C}, \mu_1, \mu_2, n_x, n_y, n_z, DoF_n$);
    end

    $u \leftarrow threadId + BlockDim \times BlockId$;                // CUDA kernel (dbdKer5)
    if ($u < N_{DoF}$) then
        $b^{(u)} \leftarrow z^{(u)} r^{(u)}$;
    end

    $\rho_k \leftarrow \rho_k + \sum_{u=0}^{N_{DoF}-1} b^{(u)}$;              // Reduction using thrust library
    $\beta_k \leftarrow \rho_k / \rho_{k-1}$;

    $u \leftarrow threadId + BlockDim \times BlockId$;                // CUDA kernel (dbdKer6)
    if ($u < N_{DoF}$) then
        $p^{(u)} \leftarrow r^{(u)} + \beta_k p^{(u)}$;
    end
end
```

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 that the contributions to the matrix of coefficients are bounded by 8 elements and by 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 $3 \times 3$. 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 [38], 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 $\ell$ 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 elemental properties $\mathbf{d}$ of the topology optimization. This is done for both preconditioners and allows exploiting data locality alleviating mostly of memory related problems in GPU

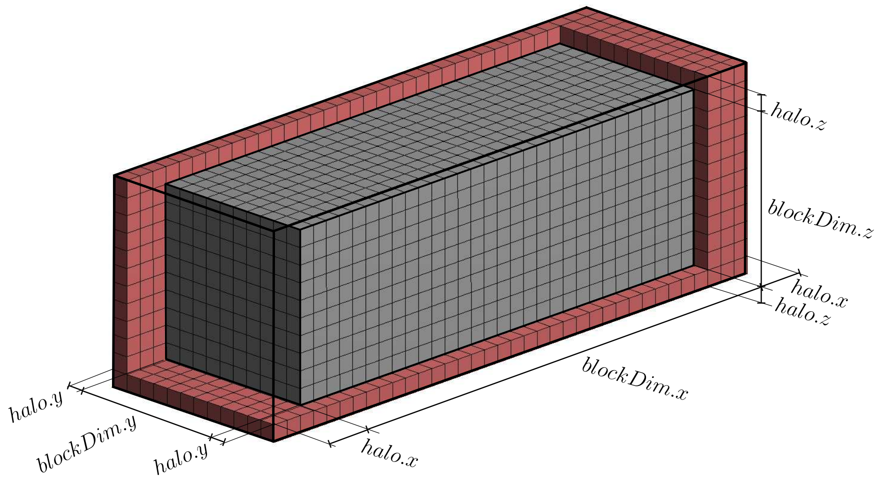


Figure 2: Cache data in shared memory by 3D block.

architectures.

The pseudocode of the DoF-by-DoF (DbD) PCG or dbdPCG GPU instance using both preconditioners for FEA is shown in Algorithm 1. Such an algorithm assumes that the hierarchical grids are composed of equally sized first-order isoparametric hexahedral elements. For the finest grid, this tessellation provides a set $\mathscr{E}$ of $N_{ele}$ elements, a set $\mathscr{N}$ of $N_{nod}$ nodes and a set $\mathscr{U}$ of $N_{DoF}$ unknowns. 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 elemental stiffness matrix $\mathbf{K}_0^e$, the vector of forces $\mathbf{f_0}$, an initialization of displacements $\mathbf{u_0}$, the vector of elemental properties $\mathbf{d}$ of the topology optimization, the vector $\mathbf{I_c}$ indicating the boundary conditions per node, the number of divisions of the grid ($n_x$,$n_y$,$n_z$) and the number of DoF per node $DoF_n$ ($DoF_n = 3$ for elasticity and $DoF_n = 1$ for heat conduction problems). Additionally, for the coarser level the vector $\mathbf{I}$ of adjacent nodal indexes per node and the assembled matrices of coefficients $\mathbf{C}$ per level are also included as input data. Note that the data of the coarser levels is not needed for the Jacobi preconditioner. Finally, 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 $n_\ell$, the number of pre- and post-smoothing steps $\mu_1$ and $\mu_2$, and the damping factor for Jacobi smoothing $\omega$. 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.

**Algorithm 2:** DbD Jacobi preconditioner (dbdJacP)

```
Data: $\mathbf{d}$, $\mathbf{K}_0^e$
Result: $\mathbf{M}$                                          // Preconditioner
1  $u \leftarrow threadId + BlockDim \times BlockId$;         // CUDA kernel
2  if ($u < N_{DoF}$) then
3      $M^{(u)} \leftarrow 0$;
4      $\mathscr{E}^{(u)} \leftarrow$ Determine index of elements containing $u$;
5      foreach $e \in \mathscr{E}^{(u)}$ do
6          $\mathscr{U}^{(e)} \leftarrow$ Determine the unknowns of $e$ ;
7          $i \leftarrow$ Extract index of $u$ from $\mathscr{U}^{(e)}$ ;
8          $M^{(u)} \leftarrow M^{(u)} + d^{(e)} K^e_{0_{ii}}$;
9      end
10     $M^{(u)} \leftarrow 1/M^{(u)}$;
11 end
```

Additionally, the GPU instance requires the calculation of diverse vector arithmetic operations which are implemented using custom developed CUDA kernels, labeled with "dbdkerX", with granularity at the DoF level. Some of these kernels require the synchronization of the threads involved in the computation to add the resulting data of all these threads. This can be 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. However, its use should be minimized because the operations are serialized when the same memory address is accessed at the same time, which can deteriorate the performance of the kernel execution. For this reason, the addition is computed as a reduction using the thrust library.

**Algorithm 3:** Vcycle Preconditioner (Vcycle)

```
Data: $\mathbf{K}_0^e$, ${}^{\ell}\mathbf{r}$, $\mathbf{d}$, $\ell$, $n_\ell$, $\omega$, $\mathbf{I_c}$, $\mathbf{I}$, $\mathbf{C}$, $\mu_1$, $\mu_2$, ${}^{\ell}n_x$, ${}^{\ell}n_y$, ${}^{\ell}n_z$, $DoF_n$
Result: ${}^{\ell}\mathbf{z}$                                                        // Preconditioner
${}^{\ell}\mathbf{z} \leftarrow 0$;
foreach $i = 1 : \mu_1$ do
    ${}^{\ell}\mathbf{s} \leftarrow$ dbdDJS(${}^{\ell}\mathbf{z}, {}^{\ell}\mathbf{r}, \mathbf{d}, \mathbf{K}_0^e, \ell, \mathbf{I_c}, \mathbf{I}, \mathbf{C}, {}^{\ell}n_x, {}^{\ell}n_y, {}^{\ell}n_z, DoF_n$);
    ${}^{\ell}\mathbf{z} \leftarrow {}^{\ell}\mathbf{s}$;
end
${}^{\ell}\mathbf{z} \leftarrow$ dbdMVP(${}^{\ell}\mathbf{z}$, $\mathbf{d}$, $\mathbf{K}_0^e$, $\ell$, $\mathbf{I_c}$, $\mathbf{I}$, $\mathbf{C}$, ${}^{\ell}n_x$, ${}^{\ell}n_y$, ${}^{\ell}n_z$, $DoF_n$);     // ${}^{\ell}z = {}^{\ell}K^e\,{}^{\ell}z$

$u \leftarrow threadId + BlockDim \times BlockId$;                       // CUDA kernel (dbdV1)
if ($u < {}^{\ell}N_{DoF}$) then
    ${}^{\ell}v(u) \leftarrow {}^{\ell}r(u) - {}^{\ell}z(u)$;
end

${}^{\ell+1}\mathbf{v} \leftarrow \mathbf{R}_{\ell}^{\ell+1}({}^{\ell}\mathbf{v})$;              // CUDA kernel – Restriction (nbnVRest)
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
    ${}^{\ell+1}\mathbf{w} \leftarrow$ VCycle($\mathbf{K}_0^e, {}^{\ell+1}\mathbf{v}, \mathbf{d}, \ell+1, n_\ell, \omega, \mathbf{I_c}, \mathbf{I}, \mathbf{C}, \mu_1, \mu_2, {}^{\ell+1}n_x, {}^{\ell+1}n_y, {}^{\ell+1}n_z, DoF_n$)
end

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

$u \leftarrow threadId + BlockDim \times BlockId$;                       // CUDA kernel (dbdV2)
if ($u < {}^{\ell}N_{DoF}$) then
    ${}^{\ell}z(u) \leftarrow {}^{\ell}z(u) + {}^{\ell}v(u)$;
end
foreach $i = 1 : \mu_2$ do
    ${}^{\ell}\mathbf{s} \leftarrow$ dbdDJS(${}^{\ell}\mathbf{z}, {}^{\ell}\mathbf{r}, \mathbf{d}, \mathbf{K}_0^e, \ell, \mathbf{I_c}, \mathbf{I}, \mathbf{C}, {}^{\ell}n_x, {}^{\ell}n_y, {}^{\ell}n_z, DoF_n$);
    ${}^{\ell}\mathbf{z} \leftarrow {}^{\ell}\mathbf{s}$;
end
```

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 $\mathbf{d}$ containing the elemental Young's modulus/thermal conductivity for elasticity/heat conduction problems. 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 performs the matrix-vector operation "on-the-fly" for the finest level ($\ell = 0$) using the vector $\mathbf{d}$ 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 at the DoF level.

A key point to perform the matrix-vector multiplication operations efficiently using GPU computing is the maximization of the use of on-chip memory. This is done in the proposed GPU instance by launching a three-dimensional CUDA kernel for such an operation. The displacements $\mathbf{p}$ and elemental properties $\mathbf{d}$ involved in the calculation required by the threads of the block are cached on shared memory. This requires to store some halo values of the unknowns of neighbor nodes. Figure 2 shows the dimension of the thread block in gray color whereas the size of shared memory required by elasticity problems operating at the DoF level should include the halo of neighbor nodes, which is depicted in red color. The unknown displacements are cached on the $x$ dimension, which requires a higher halo than the other dimensions. The block size of three-dimensional kernel can be tuned to maximize the use of shared memory by the threads of each block, which can improve significantly the GPU performance as shown in the numerical experiments.

**Algorithm 4:** DbD Damped Jacobi Smoother (dbdDJS)

```
Data: $\mathbf{z}$, $\mathbf{r}$, $\mathbf{d}$, $\mathbf{K}_0^e$, $\ell$, $\omega$, $\mathbf{I_C}$, $\mathbf{I}$, $\mathbf{C}$, $n_x$, $n_y$, $n_z$, $DoF_n$
Result: $\mathbf{s}$
$hp < x, y, z > \leftarrow < DoF_n, 1, 1 >$;  $hd < x, y, z > \leftarrow < 1, 1, 1 >$;

$idx \leftarrow threadIdx.x + blockDim.x \cdot blockIdx.x$;                                   // CUDA kernel
$idy \leftarrow threadIdx.y + blockDim.y \cdot blockIdx.y$;
$idz \leftarrow threadIdx.z + blockDim.z \cdot blockIdx.z$;
                              // Copy to shared memory per block
$z\_s[blockDim.x + 2 \cdot hp.x][blockDim.y + 2 \cdot hp.y][blockDim.z + 2 \cdot hp.z] \leftarrow \mathbf{z}$;
$d\_s[blockDim.x/hp.x - 1 + 2 \cdot hd.x][blockDim.y - 1 + 2 \cdot hd.y][blockDim.z - 1 + 2 \cdot hd.z] \leftarrow \mathbf{d}$;
__syncthreads();                                         // Synchronize threads in the block
if $(idx < (n_x + 1))$ && $(idy < (n_y + 1))$ && $(idz < (n_z + 1))$ then
    $u \leftarrow (idz * ((n_x + 1) * (n_y + 1))) + (idy * (n_x + 1)) + idx$ ;
    $n1 \leftarrow$ 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 1
        $n2 \leftarrow {}^{\ell}\mathbf{I}_k^{n1}$;
        $\mathbf{w} \leftarrow$ 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
                        $\mathbf{A} \leftarrow \mathbf{A} + d\_s^{(e)} \left(\mathbf{K}_0^e\right)_{\mathbf{v},\mathbf{w}}$;
                    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 2
                $s^{(u)} \leftarrow s^{(u)} - \omega M A_{i,j} \left(z\_s^{(\mathbf{w})}\right)_j$;
            end
        end
    end
    $s^{(u)} \leftarrow s^{(u)} + \omega M r^{(u)}$;
end
```

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 vector $\mathbf{d}$, the common elemental 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 neighbor nodal indexes per node, the residual ${}^{\ell}\mathbf{r}$ and the parameters $\omega$, $\mu_1$ and $\mu_2$ 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 nodal-based GPU instance of prolongation operator $\mathbf{P}_{\ell+1}^{\ell}$ : ${}^{\ell+1}\Omega \rightarrow {}^{\ell}\Omega$ and restriction operator $\mathbf{R}_{\ell}^{\ell+1}$ : ${}^{\ell}\Omega \rightarrow {}^{\ell+1}\Omega$ 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 $\ell$ 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 $\ell$ 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 $\omega$.

**Algorithm 5:** DbD Matrix-Vector Product (dbdMVP)

```
Data: $\mathbf{p}$, $\mathbf{d}$, $\mathbf{K}_0^e$, $\ell$, $\mathbf{I_c}$, $\mathbf{I}$, $\mathbf{C}$, $n_x$, $n_y$, $n_z$, $DoF_n$
Result: $\mathbf{a}$                                                   // a = Kp
1  $hp < x, y, z > \leftarrow < DoF_n, 1, 1 >$;  $hd < x, y, z > \leftarrow < 1, 1, 1 >$;
2
3  $idx \leftarrow threadIdx.x + blockDim.x \cdot blockIdx.x$;              // CUDA kernel
4  $idy \leftarrow threadIdx.y + blockDim.y \cdot blockIdx.y$;
5  $idz \leftarrow threadIdx.z + blockDim.z \cdot blockIdx.z$;
              // Copy to shared memory per block
6  $p\_s[blockDim.x + 2 \cdot hp.x][blockDim.y + 2 \cdot hp.y][blockDim.z + 2 \cdot hp.z] \leftarrow \mathbf{p}$;
7  $d\_s[blockDim.x/hp.x - 1 + 2 \cdot hd.x][blockDim.y - 1 + 2 \cdot hd.y][blockDim.z - 1 + 2 \cdot hd.z] \leftarrow \mathbf{d}$;
8  __syncthreads();                               // Synchronize threads in the block
9  if $(idx < (n_x + 1))$ && $(idy < (n_y + 1))$ && $(idz < (n_z + 1))$ then
10     $u \leftarrow (idz * ((n_x + 1) * (n_y + 1))) + (idy * (n_x + 1)) + idx$ ;
11     $n1 \leftarrow$ Determine node containing $u$;
12     $\mathbf{v} \leftarrow$ Determine the unknowns of $n1$;
13     $i \leftarrow$ Extract index of $u$ from $\mathbf{v}$;
14     foreach $k = 0 : 26$ do                                          // Loop 1
15         $n2 \leftarrow {}^{\ell}\mathbf{I}_k^{n1}$;
16         $\mathbf{w} \leftarrow$ Determine the unknowns of $n2$;
17         if $n2 > -1$ then
18             if $\ell == 0$ then                                   // on-the-fly
19                 foreach $e \in \mathscr{E}^{(n1)}$ do
20                     if $n2 \in \mathscr{N}^{(e)}$ then
21                         $\mathbf{A} \leftarrow \mathbf{A} + d\_s^{(e)} \left(\mathbf{K}_0^e\right)_{\mathbf{v},\mathbf{w}}$;
22                     end
23                 end
24                 $\mathbf{A} \leftarrow$ Impose Dirichlet BC from $\mathbf{I_c}$;
25             else                                                // device memory
26                 $\mathbf{A} \leftarrow {}^{\ell}\mathbf{C}_k^{n1}$;                    // (3X3) coefficients matrix
27             end
28             foreach $j = 0 : 2$ do                                   // Loop 2
29                 $a^{(u)} \leftarrow a^{(u)} + A_{i,j} \left(p\_s^{(\mathbf{w})}\right)_j$;                 // a = Kp
30             end
31         end
32     end
33
34 end
```

## *4.2. Calculation of sensitivities*

The calculation of sensitivities (7) is decomposed into element-wise operations to exploit the parallelization potential of GPU architectures. The pseudo-code of the Element-by-Element (EbE) custom-developed CUDA kernel (ebeUKU) is detailed in Algorithm 6. The input data of such a CUDA kernel are: the vector $\mathbf{d}_\rho$, the result of the state $\mathbf{u}$ and adjoint state $\mathbf{u}^*$ equations, the number $DoF_e$ of DoFs per element, the elemental stiffness matrix $\mathbf{K}_0$, the number of divisions of the regular grid ($n_x$,$n_y$,$n_z$) and the number of DoF per node $DoF_n$. The common elemental matrix $\mathbf{K}_0$ is stored in constant memory. The algorithm is designed as a three-dimensional CUDA kernel using on-chip memory for the vector $\mathbf{d}$ and the state $\mathbf{u}$ and adjoint state $\mathbf{u}^*$ involved in the calculation of the thread block. The sensitivity of each block element is calculated by one thread, for which the unknowns $\boldsymbol{\mathscr{U}}^{(e)}$ attached to the element $e$ are determined making use of the grid regularity. The inner loops operate on each degree of freedom of the element to calculate the sensitivities $\mathbf{f}_\rho{}^{(e)}$ according to (7). Finally, the objective function $f$ is computed using reduction operation using the thrust library. The compliant mechanism synthesis also requires the calculation of the objective function ($u_{out}$) as well as the adjoint state ($\mathbf{u}^*$), which are calculated on GPU using the custom-developed CUDA kernel detailed in Algorithm 1.

**Algorithm 6:** EbE calculation of sensitivities (ebeUKU)

```
Data: $\mathbf{d}_\rho$, $\mathbf{u}^*$, $\mathbf{K}_0$, $\mathbf{u}$, $DoF_e$, $n_x$, $n_y$, $n_z$, $DoF_n$
Result: $\mathbf{f}_\rho, f$                                   // $f_\rho = -\hat{du}^* K_0 u$
$f \leftarrow 0$;

$idx \leftarrow threadIdx.x + blockDim.x \cdot blockIdx.x$;    // CUDA kernel
$idy \leftarrow threadIdx.y + blockDim.y \cdot blockIdx.y$;
$idz \leftarrow threadIdx.z + blockDim.z \cdot blockIdx.z$;
                    // Copy to shared memory per block
$u\_s[DoF_n \cdot (blockDim.x + 1)][blockDim.y + 1][blockDim.z + 1] \leftarrow \mathbf{u}$;
$u^*\_s[DoF_n \cdot (blockDim.x + 1)][blockDim.y + 1][blockDim.z + 1] \leftarrow \mathbf{u}^*$;
$d_\rho\_s[blockDim.x][blockDim.y][blockDim.z] \leftarrow \mathbf{d}_\rho$;
__syncthreads();                             // Synchronize threads in the block
if ($idx < n_x$) && ($idy < n_y$) && ($idz < n_z$) then
    $e \leftarrow (idz * ((n_x + 1) * (n_y + 1))) + (idy * (n_x + 1)) + idx$ ;
    $\mathscr{U}^{(e)} \leftarrow$ Determine the unknowns of $e$ ;
    foreach $i \in \{1, \ldots, DoF_e\}$ do                          // Loop
        foreach $j \in \{1, \ldots, DoF_e\}$ do                      // Loop
            ${f_\rho}^{(e)} \leftarrow f_\rho^{(e)} + d_\rho\_s^{(e)} u^*\_s_j^{(e)} K_{0_{ij}} u\_s_j^{(e)}$;
        end
    end
    ${f_\rho}^{(e)} \leftarrow -{f_\rho}^{(e)}$;                          // sensitivities
end

$f \leftarrow f + \sum_{u=0}^{N_{DoF}-1} f_\rho^{(u)}$;                 // Reduction using thrust library
```

*4.3. Filtering strategy*

The filtering strategy aims to prevent numerical artifacts in the optimal solution. The Algorithm 7 shows the pseudo-code of the EbE GPU implementation of the three-dimensional sensitivity filter (ebeFILTER) following (3). The input data of such an algorithm are: the density design variables $\boldsymbol{\rho}$, the vector of sensitivities $\mathbf{f}_\rho$, the finite element size ($d_x$, $d_y$, $d_z$), the number of divisions of the grid ($n_x$,$n_y$,$n_z$) and the filter radius $R$. Note that the $ebeFILTER$ kernel cannot be implemented using a three-dimensional launching approach. This is due to the radius size of (4) requires a large halo in all the dimensions of the cache data using shared memory, which normally exceeds the limit of such a resource or forces to launch very small three-dimensional blocks that do not make use of the parallelization potential of GPU computing. The kernel applies to each element $e \in \mathscr{E}$ going through each element that falls within the projection of the linear convolution function (3). The regularity of the grid permits to efficiently look for the neighbors of the corresponding element, avoiding the use of a global index table that requires a substantial number of memory accesses. The granularity of the ebeFILTER CUDA kernel is at the element level, assigning one CUDA thread to each element when invoking such a kernel.

**Algorithm 7:** EbE sensitivity filter (ebeFILTER)

```
Data: $\boldsymbol{\rho}$, $\mathbf{f}_\rho$, $d_x$, $d_y$, $d_z$, $n_x$, $n_y$, $n_z$, $R$
Result: $\hat{\mathbf{f}}_\rho$                                   // Filtered sensitivities
$s_x \leftarrow floor(R/d_x)$;
$s_y \leftarrow floor(R/d_y)$;
$s_z \leftarrow floor(R/d_z)$;
$\hat{\mathbf{f}}_\rho \leftarrow 0$;
$sum \leftarrow 0$;

$e \leftarrow threadId + BlockDim \times BlockId$;                 // CUDA kernel
if ($e < N_{ele}$) then
    $c_z \leftarrow floor(e/((n_x+1)(n_y+1)))$;
    $c_y \leftarrow floor((e - c_x(n_x+1)(n_y+1))/(n_x+1))$;
    $c_x \leftarrow e - c_z(n_x+1)(n_y+1) - c_y(n_x+1)$;
    for $r \leftarrow max(c_x - s_x, 0)$ to $min(c_x + s_x, n_x)$ do
        for $s \leftarrow max(c_y - s_y, 0)$ to $min(c_y + s_y, n_y)$ do
            for $t \leftarrow max(c_z - s_z, 0)$ to $min(c_z + s_z, n_z)$ do
                $i \leftarrow t(n_x+1)(n_y+1) + s(n_x+1) + r$;
                $w \leftarrow R - sqrt((c_x - r)^2 + (c_y - s)^2 + (c_z - t)^2)$ ;   // convolution
                  operator
                $sum \leftarrow sum + max(0, w)$;
                $\hat{f}_\rho^{(e)} \leftarrow \hat{f}_\rho^{(e)} + max(0, w)\rho^{(i)}\hat{f}_\rho^{(i)}$;
            end
        end
    end
    $\hat{f}_\rho^{(e)} \leftarrow \hat{f}_\rho^{(e)}/(\rho^{(e)} sum)$
end
```

## *4.4. Optimality Criterion (OC) update scheme*

The pseudo-code of the EbE GPU implementation of the density updating strategy (ebeOC) is shown in Algorithm 8. Such an implementation makes use of the bisection method and the optimality criterion according to (9) and (10). The input data of this algorithm are: the density design variables ($\boldsymbol{\rho}$), the interval bounds $[\lambda_l, \lambda_u]$ of the Lagrange multiplier, the numerical damping coefficient ($\eta$), a positive step width ($\zeta$), an intermediate density penalty factor ($q$), the finite element size ($d_x$, $d_y$, $d_z$), the objective function sensitivities ($\hat{\mathbf{f}}_\rho$) and the volume sensitivities ($\mathbf{V}_\rho$). The algorithm is composed of two CUDA kernels, highlighted in boxes labeled with "CUDA kernel", with element level granularity. The first kernel goes through each element of the regular grid to calculate the volume penalized with $\boldsymbol{\rho}$. The total volume is then calculated as the addition of partial volume computed as a reduction using the thrust library. The bisection method divides the interval repeatedly to calculate the midpoint Lagrange multiplier $\lambda_m$, which is then used to calculate the KKT optimality condition. The second kernel updates the element density $\boldsymbol{\rho}$ following (9) and copy back the calculated volume $V_{new}$ to update the Lagrange multiplier, which is used to evaluate the stopping criteria. This procedure is repeated until such a convergence criterion is satisfied. The vector $\boldsymbol{\rho}$ is finally copy back to host memory.

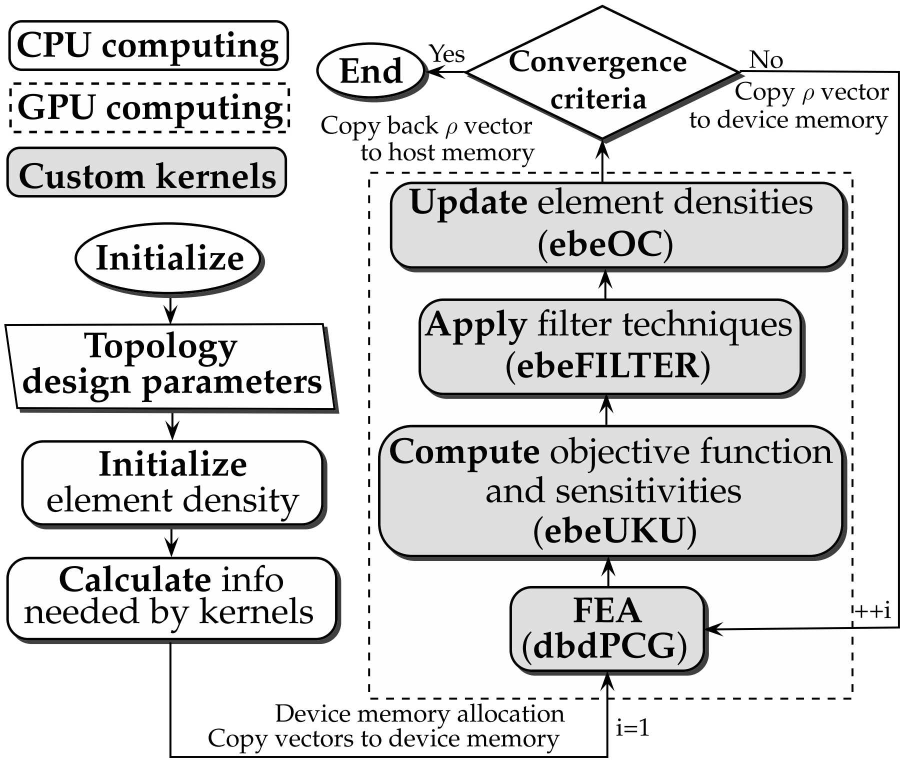

CPU computing
GPU computing
Custom kernels
Initialize
Topology design parameters
Initialize element density
Calculate info needed by kernels
Device memory allocation
Copy vectors to device memory
i=1
FEA (dbdPCG)
Compute objective function and sensitivities (ebeUKU)
Apply filter techniques (ebeFILTER)
Update element densities (ebeOC)
Convergence criteria
Yes
End
Copy back ρ vector to host memory
No
Copy ρ vector to device memory
++i

Figure 3: Flowchart of GPU instance of SIMP method.

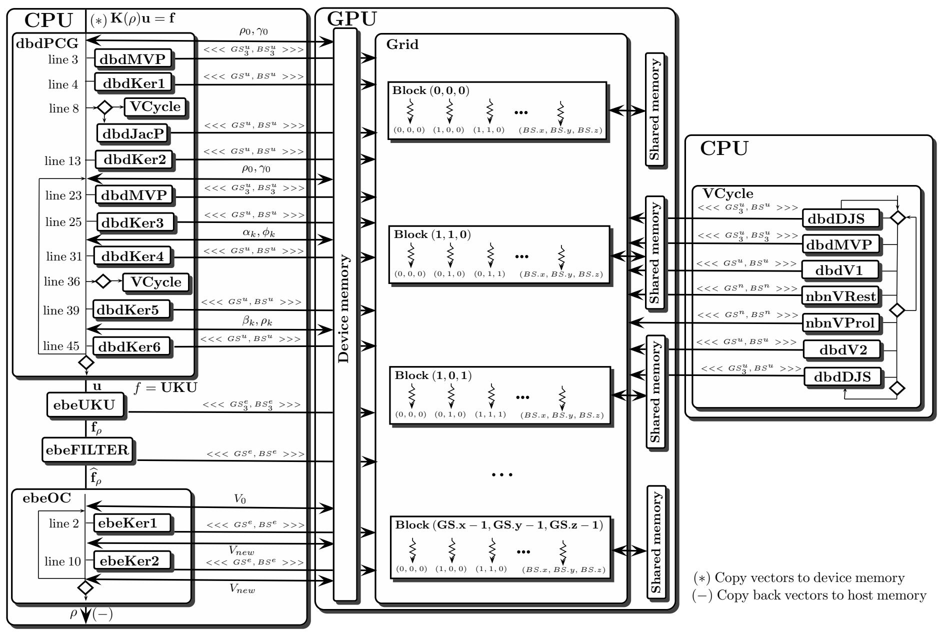


Figure 4: Kernel invocation and memory transfer for each iteration of SIMP method.

**Algorithm 8:** EbE density update (ebeOC)

```
Data: $\boldsymbol{\rho}$, $\lambda_l$, $\lambda_u$, $\eta$, $\zeta$, $q$, $d_x$, $d_y$, $d_z$, $\hat{\mathbf{f}}_\rho$, $\mathbf{V}_\rho$
Result: $\boldsymbol{\rho}$                                              // updated densities
1  $V_0 \leftarrow 0$ ;
2
3  $e \leftarrow threadId + BlockDim \times BlockId$;                    // CUDA kernel (ebeKer1)
4  if ($e < N_{ele}$) then
5      $b^{(e)} \leftarrow d_x d_y d_z \rho^{(e)}$;
6  end
7
8  $V_0 \leftarrow V_0 + +\sum_{u=0}^{N_{DoF}-1} b^{(u)}$;              // Reduction using thrust library
9  while ($(\lambda_u - \lambda_l)/(\lambda_u + \lambda_l) > 10^{-6}$) do
10     $\lambda_m \leftarrow (\lambda_u + \lambda_l)/2$;
11     $V_{new} \leftarrow 0$ ;
12
13     $e \leftarrow threadId + BlockDim \times BlockId$;                // CUDA kernel (ebeKer2)
14     if ($e < N_{ele}$) then
15         $\rho^{(e)} \leftarrow max(0, max(\rho^{(e)} - \zeta, min(1, min(\rho^{(e)} + \zeta, (\rho^{(e)}(-\hat{f}_\rho^{(e)}/V_\rho^{(e)}/\lambda_m)^\eta)^q))))$;
16         $b^{(e)} \leftarrow d_x d_y d_z \rho^{(e)}$;
17
18     end
19
20     $V_{new} \leftarrow V_{new} + \sum_{u=0}^{N_{DoF}-1} b^{(u)}$;   // Reduction using thrust library
21     if $V_0 > V_{new}$ then
22         $\lambda_l = \lambda_m$;
23     else
24         $\lambda_u = \lambda_m$;
25     end
26 end
```

*4.5. GPU implementation and memory management*

The GPU instance of density-based topology optimization consists of the custom-developed CUDA kernels for the computationally demanding tasks involved in the algorithm. Figure 3 shows the flowchart of the algorithm and the relevant memory allocation and memory transfer of large vectors during the optimization, whereas Figure 4 details the custom-developed kernels, including memory transfer of scalar values in each iteration of the topology optimization and the invocations from the host. One can observe that the information needed by the custom-developed CUDA kernels is allocated and transferred to the device memory in the initialization of the optimization process, and that the memory transaction between host and device memory of large vectors is reduced to the $\boldsymbol{\rho}$ vector in each iteration of the optimization to evaluate the stopping criteria. This is of paramount importance to obtain reasonable results. One also can observe that the iterations of the optimization process only requires to copy back to host memory some scalar values. This minimization of memory transactions increases notably the GPU performance. The CUDA kernels are invoked from the host assigning the corresponding grid size $GS$ and block size $BS$ to fit the granularity of the custom-developed kernels, which are tuned to empiric values that provide good performance.

The device memory allocated for the custom-developed kernels, obviating the allocation of scalar values, is shown in Figure 5. 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 ${}^{\ell+1}\mathbf{C}$ are also stored in the global device memory for the corresponding $n_l$ levels. 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. Additionally, the vectors $\boldsymbol{\rho}$, $\mathbf{d}_{\boldsymbol{\rho}}$, $\mathbf{u}^*$, $\mathbf{f}_\rho$ and $\hat{\mathbf{f}}_\rho$ required by the kernels ebeUKU, ebeFILTER and ebeOC are also stored in the global

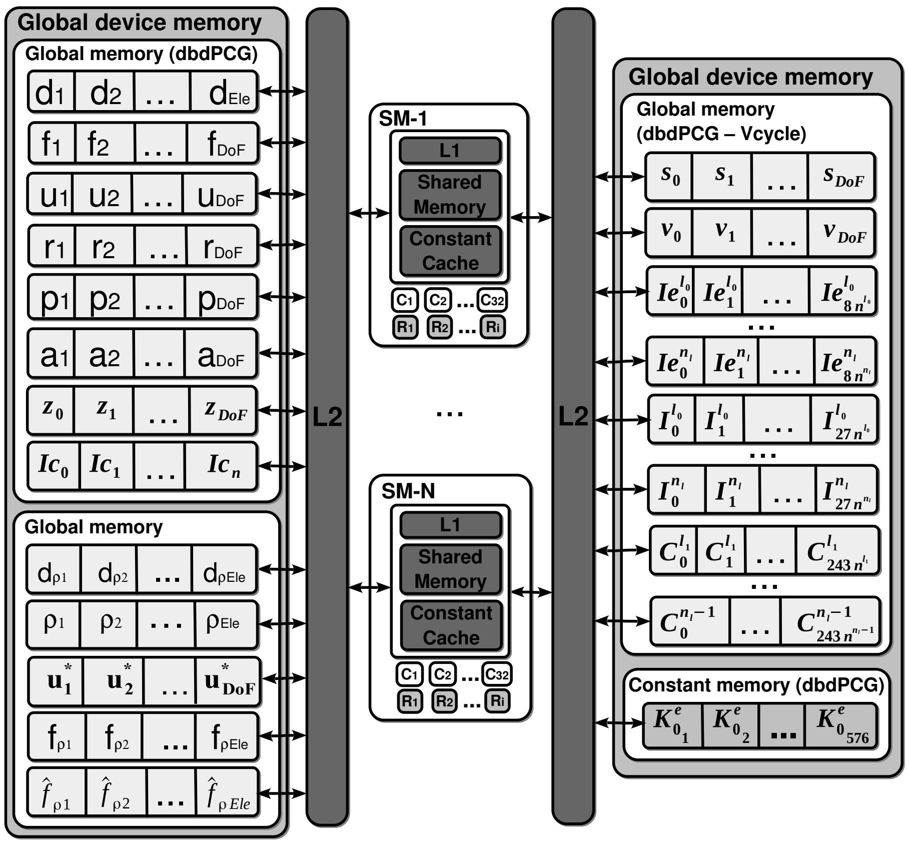

Global device memory
Global memory (dbdPCG)
$d_1$ $d_2$ … $d_{Ele}$
$f_1$ $f_2$ … $f_{DoF}$
$u_1$ $u_2$ … $u_{DoF}$
$r_1$ $r_2$ … $r_{DoF}$
$p_1$ $p_2$ … $p_{DoF}$
$a_1$ $a_2$ … $a_{DoF}$
$z_0$ $z_1$ … $z_{DoF}$
$Ic_0$ $Ic_1$ … $Ic_n$
Global memory
$d_{\rho 1}$ $d_{\rho 2}$ … $d_{\rho Ele}$
$\rho_1$ $\rho_2$ … $\rho_{Ele}$
$u^*_1$ $u^*_2$ … $u^*_{DoF}$
$f_{\rho 1}$ $f_{\rho 2}$ … $f_{\rho Ele}$
$\hat{f}_{\rho 1}$ $\hat{f}_{\rho 2}$ … $\hat{f}_{\rho Ele}$
L2
SM-1
L1
Shared Memory
Constant Cache
C1 C2 … C32
R1 R2 … Ri
…
SM-N
L1
Shared Memory
Constant Cache
C1 C2 … C32
R1 R2 … Ri
L2
Global device memory
Global memory (dbdPCG – Vcycle)
$s_0$ $s_1$ … $s_{DoF}$
$v_0$ $v_1$ … $v_{DoF}$
$Ie^{l_0}_0$ $Ie^{l_0}_1$ … $Ie^{l_0}_{8n^{l_0}}$
…
$Ie^{n_l}_0$ $Ie^{n_l}_1$ … $Ie^{n_l}_{8n^{n_l}}$
$I^{l_0}_0$ $I^{l_0}_1$ … $I^{l_0}_{27n^{l_0}}$
…
$I^{n_l}_0$ $I^{n_l}_1$ … $I^{n_l}_{27n^{n_l}}$
$C^{l_1}_0$ $C^{l_1}_1$ … $C^{l_1}_{243n^{l_1}}$
…
$C^{n_l-1}_0$ … $C^{n_l-1}_{243n^{n_l-1}}$
Constant memory (dbdPCG)
$K^e_{0_1}$ $K^e_{0_2}$ … $K^e_{0_{576}}$

Figure 5: Device memory required by the kernels of the proposed GPU instance.

device memory. In the case of compliance minimization problems, the memory allocation for the adjoint state $\mathbf{u}^*$ can be obviated because such problems are self-adjoint.

## 5. Numerical experiments

The performance of the proposed GPU instance of density-based topology optimization is evaluated using three topology optimization problems in different fields. In particular, the stiffness design of continuum structures, the heat sink design cooled by heat conduction and the compliant mechanism synthesis. The two first two numerical experiments aim to analyze the use of different $DoF_n$, whereas the last one aims to explore the use of the proposal with multi-GPU systems. The solving of the system of equations using GPU computing is evaluated using two preconditioning techniques; in particular, the Jacobi preconditioner and the geometric multigrid preconditioner. Besides, the way to launch the kernel for the matrix-vector multiplication operation is studied to make use of the parallel potential of massive parallel architectures in this demanding operation. In addition, the proposed GPU instance is compared with the classical CPU implementation, in which the global stiffness matrix is assembled and the sparse-matrix representation is used to perform the operations required by the PCG solver. 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 massive parallel capabilities. This aims to evaluate the scalability of the GPU instance with respect to the capabilities of the graphics units.

The three numerical experiments are performed using a computer with an Intel Core i7-5820k 3.33 GHz and 32 GB of RAM memory. Three Nvidia GPUs are installed in the computer to perform the experiments: GF100-100-KD (Quadro 4000), GF100 (Tesla C2070) and GK110b (Tesla K40). The first two graphics cards use the Fermi micro-architecture, whereas the third one makes use of the Kepler micro-architecture. Table 1 summarizes the most relevant specifications

| **Nvidia GPU model** | **CUDA cores** | **Processor clock (MHz)** | **Memory clock (MHz)** | **FMA-DP (GFlops)** |
|---|---|---|---|---|
| GF100-100-KD | 256 | 475 | 1400 | 243 |
| GF100 | 448 | 575 | 1566 | 515.2 |
| GK110b | 2880 | 889 | 3004 | 1430 |

Table 1: GPU specifications for benchmark devices.

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 and the numerical experiments are run on 64 bits Linux OS with the NVIDIA Driver Version 340.76. It is important to remark that the development environment and the graphics driver updates often show significant performance improvements.

A continuation strategy for parameter $p$ is adopted for all the numerical experiments; in particular, the evolution of the parameter $p$ in the continuation step, according to [53], is as follows

$$p_{k+1} = \begin{cases} 1 & \text{if } k \leqslant 20 \\ \min(3, \gamma \cdot p_k) & \text{if } k > 20 \end{cases} . \tag{11}$$

This continuation strategy is represented in Figure 6 for different $\gamma$ values. By modifying the parameter $p$, the optimization problem is gradually changed from a convex problem to the original non-convex problem, which is governed by the parameter $\gamma$. Based on the experience provided by Groenwold and Etman [53], such a $\gamma$ parameter is set to 1.02 in all the numerical experiments.

The minimum compliance design problem consists of finding the material density distribution that minimizes the deformation of the structure, a tied-arch bridge in our case, under the prescribed loading and boundary conditions. Figure 7(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. The bases of bridge abutments and the bottom part of the bridge deck

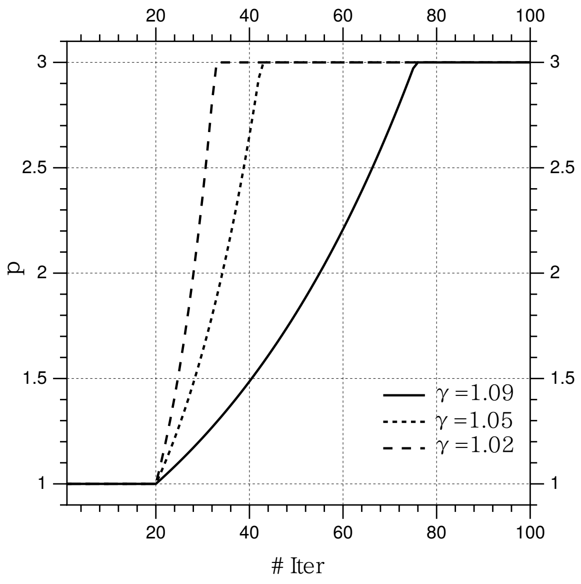
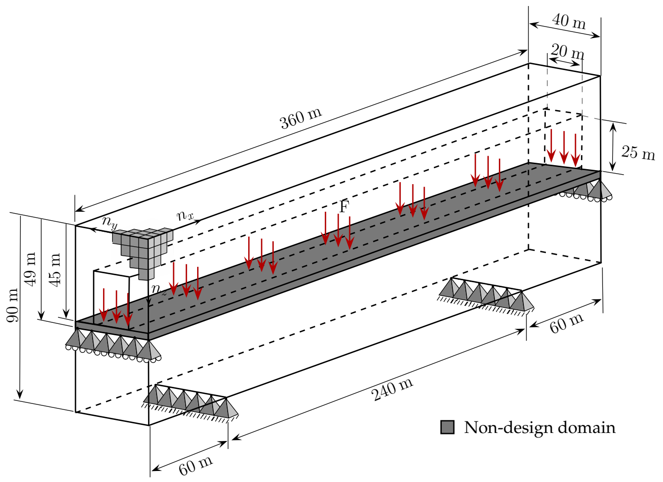


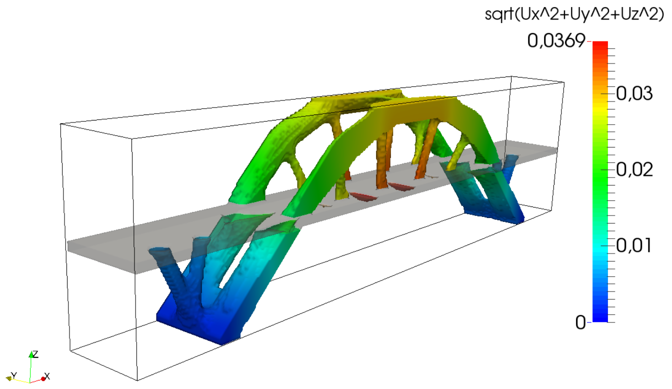
Figure 6: Evolution of penalty parameter $p$ using continuation on $p$.

are simply-supported along the edges located at 60m from left face and right face respectively. A uniformly distributed load is applied to the top of the non-optimizable bridge deck. The solving of this problem makes use of one of the vertical planes of symmetry to only analyze the half of the finite element model. The half design domain is discretized using $184 \times 40 \times 90$ eight-node hexahedral linear brick elements, i.e. 662,400 elements and about 2 million of DoFs. The material parameters are $\mathrm{E_{solid}} = 210$ GPa, $\mathrm{E_{void}} = 2.1 \cdot 10^{-4}$ GPa and $\nu = 0.31$. The maximum residual error is set to $10^{-8}$ for the PCG algorithm and the calculations are performed using double-precision floating-point format. The target volume is the 15% of the volume of the design domain. The parameters of the bisection method used to infer the Lagrange multiplier $\lambda$ are: $\eta = 0.5$, $\zeta = 0.2$, $\lambda_l = 0$, $\lambda_u = 10^9$ and $\lambda_{min} = 10^{-40}$. The topology optimization parameters are: filter radius $R = 3$ m and factor $q$ as follows

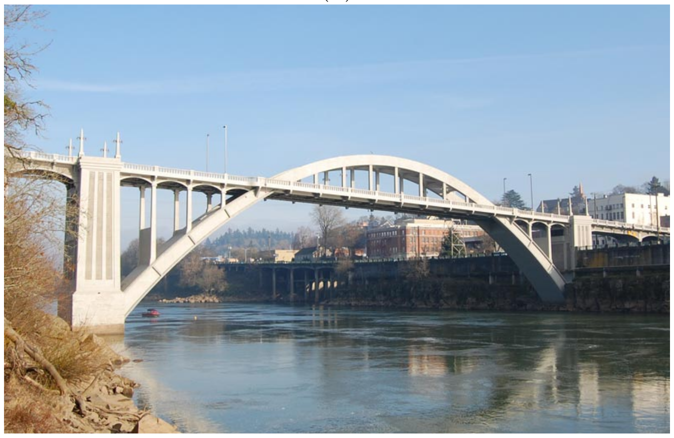
$$q_{k+1} = \begin{cases} 1 & \text{if } k \leqslant 15 \\ \min(3, 1.01 \cdot q_k) & \text{if } k > 15 \end{cases} . \tag{12}$$



(a)



(b)



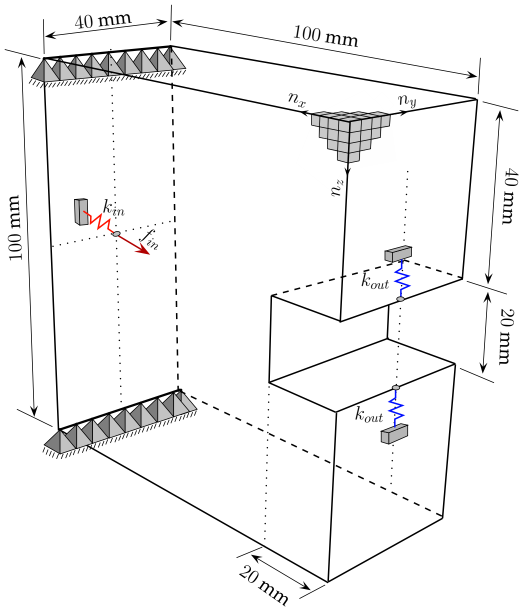
(c)

Figure 7: Tied-arch bridge benchmark: (a) the design domain and boundary conditions, (b) the topology design with threshold $\rho = 0.99$ from isometric view and (c) the Oregon city bridge.

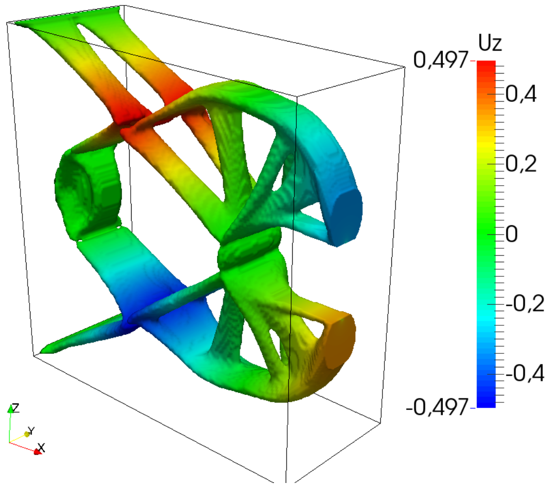
The resulting structural design of the tied-arch bridge is shown in Figure 7(b), where the final design is thresholded at $\rho = 0.99$ and colored by the magnitude of the displacement field. This structural design resembles the topology of this kind of bridges, as shown in the real bridge of Figure 7(c).

The compliant mechanism design problem consists of maximizing the output displacements $u_{out}$ for mechanisms under given forces $f_{in}$ applied to the input actuators. The numerical experiment aims to provide the optimal topology compliant mechanism design for a gripper. The design domain and the boundary conditions for this benchmark are shown in Figure 8(a). This design domain is simply supported at the upper and lower edges of left face. The input actuator and the output port are modeled as springs with different degrees of stiffness, in particular $k_{in}$ and $k_{out}$ respectively. These degrees of stiffness are fixed to $k_{in} = k_{out} = 1$kN/mm in the springs. The input force $f_{in} = 1$kN is applied to the center of a given face, for which the top and bottom edges are fixed. The output ports are located in the opposite face as indicated in Figure 8(a). The material considered is nylon with Young's modulus $\mathrm{E}_{\mathrm{solid}} = 3$ GPa and Poisson's coefficient $\nu = 0.4$. For the void material, a Young's modulus $\mathrm{E}_{\mathrm{void}} = 10^{-3}$ GPa is considered. Only the half of the design domain is analyzed using the horizontal plane of symmetry, which is discretized using $160 \times 40 \times 80$ eight-node hexahedral linear brick elements, i.e. 512,000 elements or 1,604,043 DoFs. The target volume is about 10% of the volume of the design domain. The topology optimization parameters are: filter radius $R = 4$ mm and factor $q$ following (12). The parameters of the bisection method used to infer the Lagrange multiplier $\lambda$ are: $\eta = 0.3$, $\zeta = 0.1$, $\lambda_l = 0$, $\lambda_u = 10^9$ and $\lambda_{min} = 10^{-40}$. The resulting mechanism design is shown in Figure 8(b), where the final design is thresholded at $\rho = 0.99$ and colored by the magnitude of the displacement field. One can observe that the gripping "jaws" along with the hinges resemble the common topology of grippers.

The heat sink design considering conduction heat transfer, also known as volume-to-point heat conduction problem [59], consists of the minimization of thermal compliance to maximize the heat conduction transfer from the heat sink.



(a)

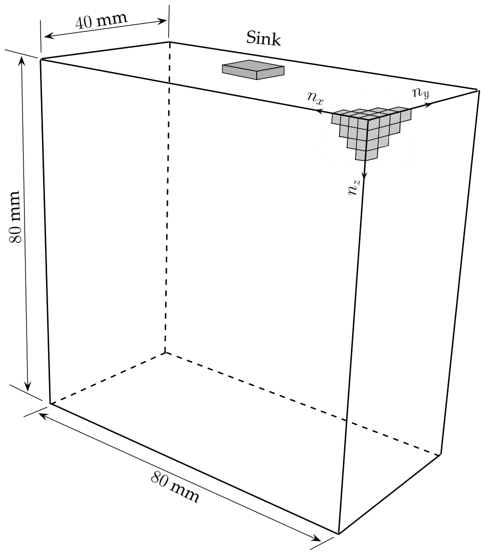


(b)

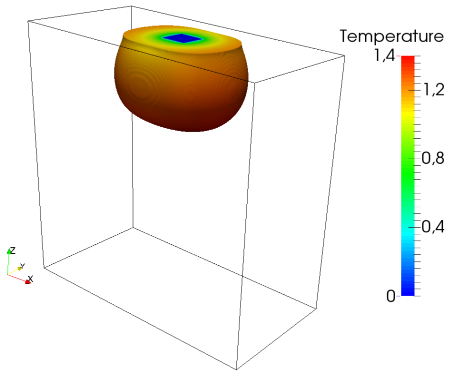
Figure 8: 3D compliant gripper benchmark: (a) the design domain and boundary conditions and (b) the topology design with threshold $\rho = 0.99$ from isometric view.

The volume is subjected to a heat generation rate at every point of the design domain and cooled through a small patch (heat sink) located in the middle of its upper face. The thermal conductivity matrix of each element is considered, according to (2), as the distribution of two material phases comprising a good thermal conductor ($\mathbf{K}_0$) and a poor conductor ($\mathbf{K}_{min}$). Both material phases are considered homogeneous and isotropic without temperature effect on their conductivities. Therefore, the topology optimization problem aims to find the optimal distribution of "good" thermal conductor that minimizes the highest temperature under a volume constraint. The design domain and the boundary conditions for this benchmark are shown in Figure 9(a). The thermal conductivities are $k_{min} = 0.1\ W/m^2K$ and $k_0 = 100\ W/m^2K$. The heat generation rates for the different phases are similar with a magnitude of $F = 10\ kW/m^3$. All the boundaries are adiabatic with the exception of the heat sink, in which $T = 0$. Only one quarter of the design domain is analyzed using the two vertical planes of symmetry, which is discretized using a regular grid of $128 \times 64 \times 256$ eight-node hexahedral linear brick elements, i.e. 2,097,152 elements or 6,464,835 DoFs. The target volume is about the 30% of the volume of the design domain. The topology optimization parameters are: filter radius $R = 5$ mm and factor $q$ following (12). The parameters of the bisection method used to infer the Lagrange multiplier $\lambda$ are: $\eta = 0.5$, $\zeta = 0.2$, $\lambda_l = 0$, $\lambda_u = 10^9$ and $\lambda_{min} = 10^{-40}$. The resulting topology design along with intermediate designs to illustrate the evolution of the optimization process are shown in Figure 9. The intermediate and final designs are thresholded at $\rho = 0.99$ and colored by the magnitude of the temperature field (°C). One can observe how the final design is a "thermal tree" composed of conductivity branches that move the heat away from the heat source.

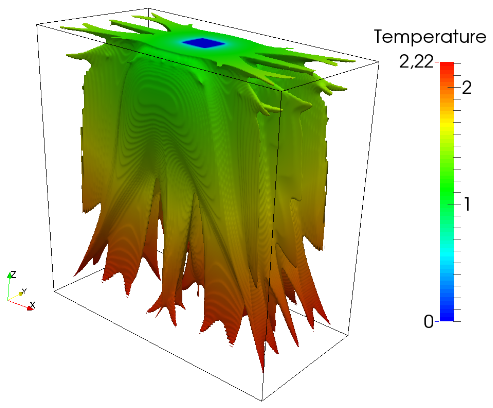
Figure 10 shows the evolution of the Measure of Non-Discreteness ($M_{nd}$) [60] for the numerical experiments. This value indicates whether an optimized design has converged to a discrete solution following



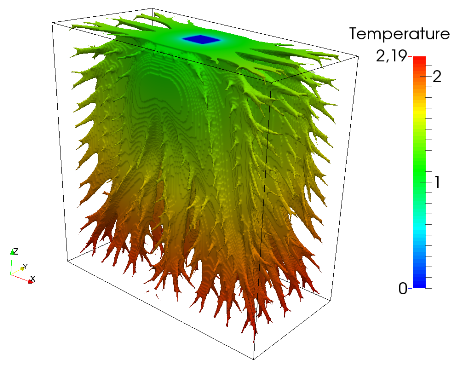
(a)



(b) Iteration 20



(c) Iteration 50



(d) Iteration 250

Figure 9: Heat sink design benchmark: (a) the design domain and boundary conditions and (b-d) the optimized topology designs with threshold $\rho = 0.99$ at different iterations of the optimization process.

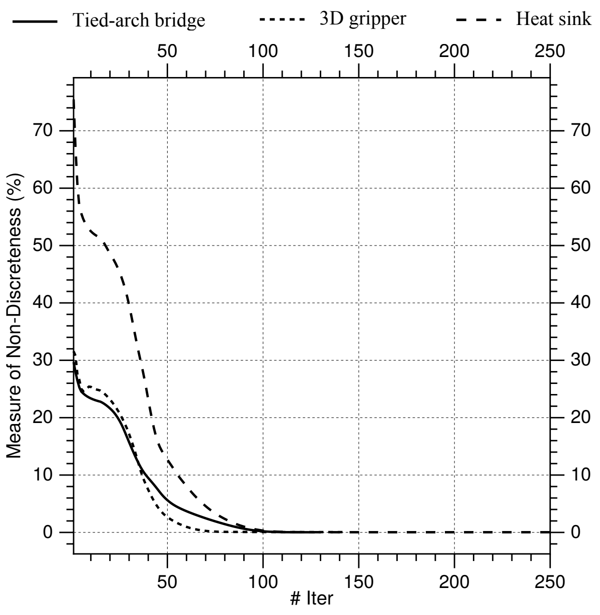


Figure 10: Measure of Non-Discreteness ($M_{nd}$) for the numerical experiments.

$$M_{nd} = \frac{\sum_{e=1}^{N_{ele}} 4\rho_e(1-\rho_e)}{N_{ele}} \times 100\% \;, \tag{13}$$

where $M_{nd} = 100\%$ indicates that the design is totally gray and $M_{nd} = 0\%$ means that the design is fully discrete. One can observe in Figure 10 that the numerical experiments show stable convergence to fully discrete designs. Figure 11 shows the evolution of the objective function for the different topology optimization problems. One can observe that all the topology optimization problems show stable convergence of the objective function. All the numerical experiments are performed using $\gamma = 1.02$ following (11). We can observe in Figure 6 how the parameter $p$ is fixed to $p = 1$ for the 20 initial iterations and then it is increased until $p = 3$ in the iteration 75. The objective function for the different topology optimization problems converges to a minimum with intermediate densities, as shown in Figure 10, in the iteration 20. As the penalty parameter $p$ is increased one can observe that the objective function converges

| **Block size** | **Grid size** | **Shared memory (bytes)** | **Threads per block** | **Wall-clock time (ms)** |
|---|---|---|---|---|
| $111 \times 3 \times 3$ | $5 \times 14 \times 33$ | 28264 | 999 | 17.2 |
| $63 \times 4 \times 4$ | $9 \times 11 \times 25$ | 24272 | 1008 | 18.0 |
| $39 \times 5 \times 5$ | $15 \times 9 \times 20$ | 21672 | 975 | 18.7 |
| $24 \times 7 \times 6$ | $24 \times 6 \times 17$ | 21312 | 1008 | 21.6 |
| $21 \times 7 \times 6$ | $27 \times 6 \times 17$ | 19136 | 882 | 22.1 |
| $21 \times 6 \times 6$ | $27 \times 7 \times 17$ | 16960 | 756 | 25.0 |
| $18 \times 6 \times 6$ | $31 \times 7 \times 17$ | 15032 | 648 | 25.1 |

Table 2: Wall-clock time of dbdMVP configuring different grid and block size for tied-arch bridge experiment.

to a local minima with a lower $M_{nd}$ until a fully discrete design is found.

The efficient calculation of matrix-vector multiplication operations is crucial to obtain a reasonable performance for the solving stage. The performance of this operation using shared memory is evaluated modifying the block size used to launch the $dbdMVP$ kernel. This is done launching one FEA of the tied-arch bridge problem using different three-dimensional block size configurations. Table 2 shows the battery of experiments performed to tune the block size, which is sorted by wall-clock time. As a general rule, the performance is maximized using the maximum amount of on-chip memory and the maximum number of threads. However, the GPU performance of $dbdMVP$ kernel is maximized as increasing the $x$ dimension of block size due to the unknowns are stored in such a dimension.

Table 3 details the thread hierarchy used for invoking the CUDA kernels in the different benchmarks. One can observe that the three-dimensional kernels with granularity at the DoF level use the block size configuration obtained from the experiment shown in Table 2. Three-dimensional kernels with granularity at the element level use symmetric kernels to maximize the use of shared memory. These kernels do not operate over information with a dimension higher than the others. The kernels that are not using on-chip memory are launched as

| | | | | | |
|---|---|---|---|---|---|
| **Tied-arch bridge** | | | | | |
| | $BS^e$ | $BS^n$ | $BS^u$ | $BS^e_3$ | $BS^u_3$ |
| Tessellation | (512,1,1) | (512,1,1) | (512,1,1) | (10,10,10) | (111,3,3) |
| $184 \times 40 \times 96$ | $GS^e$ | $GS^n$ | $GS^u$ | $GS^e_3$ | $GS^u_3$ |
| | (1380,1,1) | (1438,1,1) | (4312,1,1) | (5,14,33) | (19,4,10) |
| **3D gripper** | | | | | |
| | $BS^e$ | $BS^n$ | $BS^u$ | $BS^e_3$ | $BS^u_3$ |
| Tessellation | (512,1,1) | (512,1,1) | (512,1,1) | (10,10,10) | (111,3,3) |
| $160 \times 40 \times 80$ | $GS^e$ | $GS^n$ | $GS^u$ | $GS^e_3$ | $GS^u_3$ |
| | (800,1,1) | (841,1,1) | (2522,1,1) | (16,4,8) | (5,11,27) |
| **Heat sink** | | | | | |
| | $BS^e$ | $BS^n$ | $BS^u$ | $BS^e_3$ | $BS^u_3$ |
| Tessellation | (512,1,1) | (512,1,1) | (512,1,1) | (10,10,10) | (8,8,8) |
| $128 \times 64 \times 256$ | $GS^e$ | $GS^n$ | $GS^u$ | $GS^e_3$ | $GS^u_3$ |
| | (4096,1,1) | (4209,1,1) | (12627,1,1) | (13,7,26) | (17,9,33) |

Table 3: Thread hierarchy for the benchmarks. The reader is referred to Figure 4 in order to find the block and grid sizes used by each kernel.

blocks with only one dimension. The grid size of all CUDA kernels is adjusted to process the corresponding information.

The performance of solving the system of equations is evaluated using the Jacobi and the geometric multigrid preconditioning techniques. Table 4 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. One can observe that the best performance considering the wall-clock time is obtained using the geometric multigrid preconditioner. The solving stage of the compliant mechanism design problem (3D gripper) is also evaluated using a multi-GPU system, which exploits the task-level parallelism involved in the computation of the direct and adjoint problems. The multi-GPU system consists of a master-worker configuration [40] installing two Nvidia Tesla K40 (GK110b) on the same host. The workers solve the governing equations of the finite element model using the matrix-free PCG

| **Tied-arch bridge** | | | | | |
|---|---|---|---|---|---|
| | dbdPCG (sec) | | ebeUKU | ebeFILTER | ebeOC |
| | Jacobi | Multigrid | (sec) | (sec) | (sec) |
| CPU (Sparse) | 69991.92 | 5104.26 | 84.69 | 316.11 | 331.65 |
| Quadro 4000 | 10769.99 | 3762.71 | 9.15 | 10.39 | 220.54 |
| Tesla C2070 | 4831.15 | 1867.22 | 8.10 | 4.96 | 214.17 |
| Tesla K40 | 3509.48 | 1256.81 | 6.75 | 2.76 | 193.59 |
| **3D gripper** | | | | | |
| | dbdPCG (sec) | | ebeUKU | ebeFILTER | ebeOC |
| | Jacobi | Multigrid | (sec) | (sec) | (sec) |
| CPU (Sparse) | 320039.48 | 9704.29 | 146.92 | 754.21 | 308.24 |
| Quadro 4000 | 51998.92 | 7009.23 | 22.76 | 35.14 | 151.91 |
| Tesla C2070 | 25404.44 | 3212.67 | 17.78 | 14.62 | 114.12 |
| Tesla K40 | 16528.05 | 2243.24 | 15.46 | 7.69 | 95.10 |
| 2 × Tesla K40 | 8549.52 | 1153.67 | ‘’ | ‘’ | ‘’ |
| **Heat sink** | | | | | |
| | dbdPCG (sec) | | ebeUKU | ebeFILTER | ebeOC |
| | Jacobi | Multigrid | (sec) | (sec) | (sec) |
| CPU (Sparse) | 26396.97 | 3006.49 | 58.76 | 28699.76 | 1435.07 |
| Quadro 4000 | 3198.65 | 4453.50 | 22.28 | 1105.31 | 857.11 |
| Tesla C2070 | 1938.04 | 1849.62 | 20.04 | 437.73 | 803.86 |
| Tesla K40 | 1172.15 | 1113.96 | 17.34 | 176.85 | 719.47 |

Table 4: Total wall-clock time for the topology optimization stages.

solver. The results show how the use of a multi-GPU platform permits to scale up the acceleration of the solver stage preserving the speedups obtained for a single analysis.

Figure 12 shows the speedups with respect to the sparse-matrix CPU implementation for the topology optimization stages using diverse graphic cards. These speedups are calculated for different graphics units to evaluate the scalability with respect to GPU capabilities. One can observe that all the computationally intensive tasks are accelerated significantly. Besides, the speedup increases with the massive parallel capabilities of the graphics unit. Speedups between 98x and 162x are obtained, in the 3D gripper and heat sink experiments respectively, for the $ebeFILTER$ kernel using the most recent graphics card (Nvidia K40-GK110b) of the devices evaluated. The solving of the system of equations limits the global speedup for the topology optimization; the numerical experiments requiring the linear elastic analysis achieve speedups of 4x and 20x using the geometric multigrid and the Jacobi preconditioning respectively. The speedup of the solving stage for the heat conduction problem is 2.7x and 22.5x using the geometric multigrid and the Jacobi preconditioning respectively. The increment in the speedup using Jacobi preconditioning is attributed to the higher size of the finite element model. Besides, the lower number of non-zero elements in the global thermal conductivity matrix reduces the number of operations and device memory accesses using GPU computing. The number of non-zero elements is 13 per DoF for the thermal experiment, whereas is 40 per DoF for elasticity problems. On the contrary, the decrement of the speedup using the geometric multigrid preconditioner is attributed to the higher size of the stiffness matrices of coefficients, which are stored in global device memory. The memory accesses of these large arrays dominate the potential speedup.

Figure 13 shows the percentage of the total wall-clock time of each kernel in the topology optimization using the Jacobi and the geometric multigrid preconditioning. As expected, the principal bottleneck of the topology optimization algorithm is the solving of the system of equations of the finite element model. The wall-clock time for elasticity problems is around the 98% of the total wall-

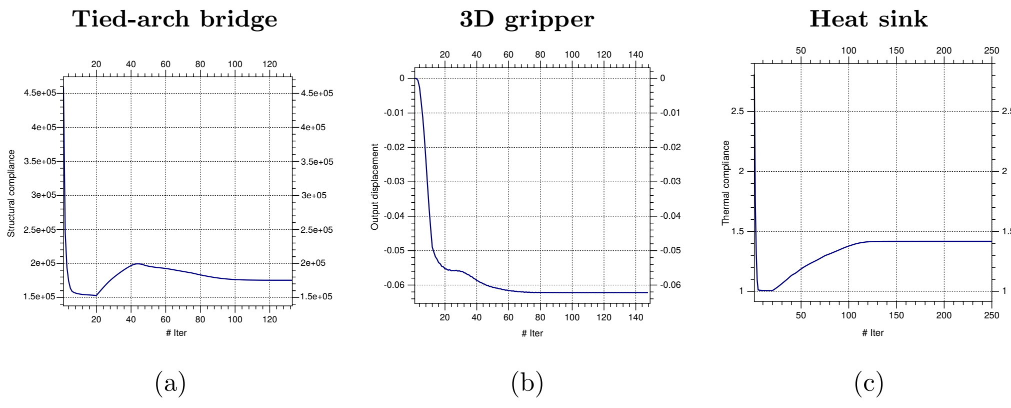


Figure 11: Evolution of the objective function for the numerical experiments.

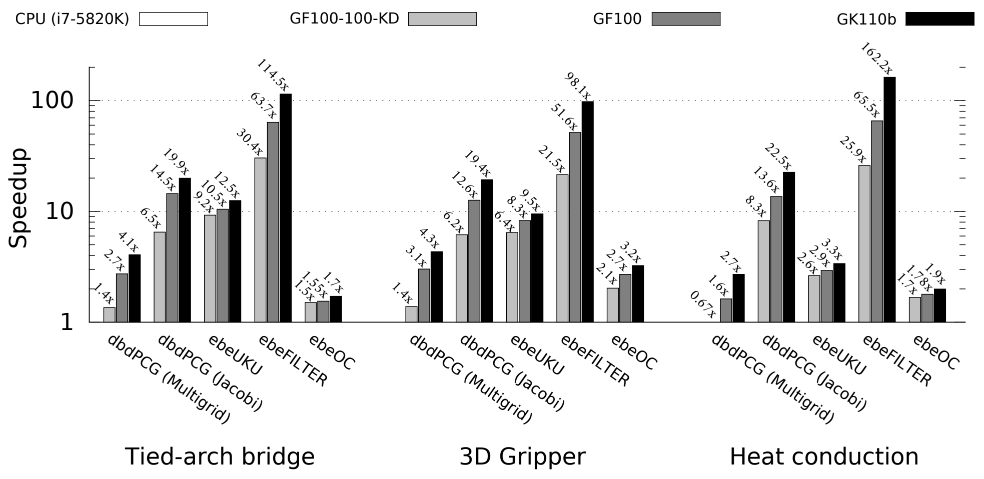


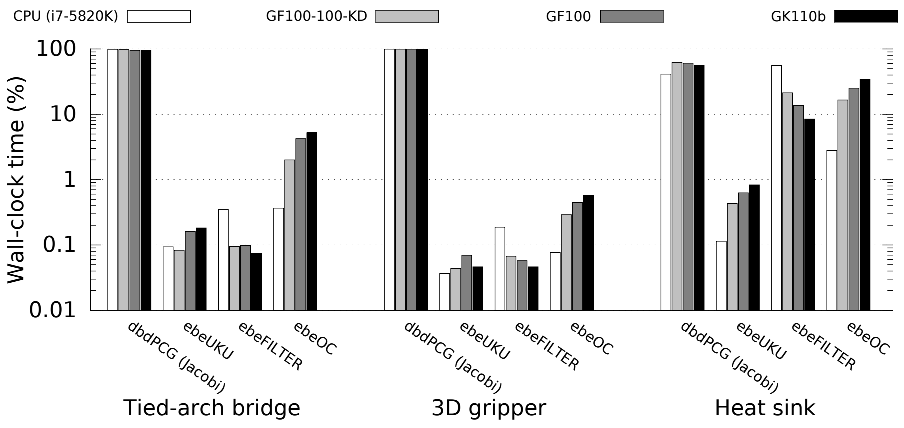
Figure 12: Speedup for the computationally demanding tasks in the numerical experiments.

clock time using the Jacobi preconditioning whereas this percentage is around the 87% using the geometric multigrid preconditioning. This percentage is considerably reduced in the heat conduction problem, where the solving stage using the most modern graphics device (GK110b) is about the 55% of the total wall-clock time using both preconditioners. This is mainly attributed to the fewer number of iterations required by PCG in the heat conduction problem to achieve convergence. As a consequence, the percentage of the total wall-clock time consumed by the filtering strategy and the bisection method become significant in the heat conduction problem, and the use of GPU computing for these stages can notably accelerate the topology optimization process.

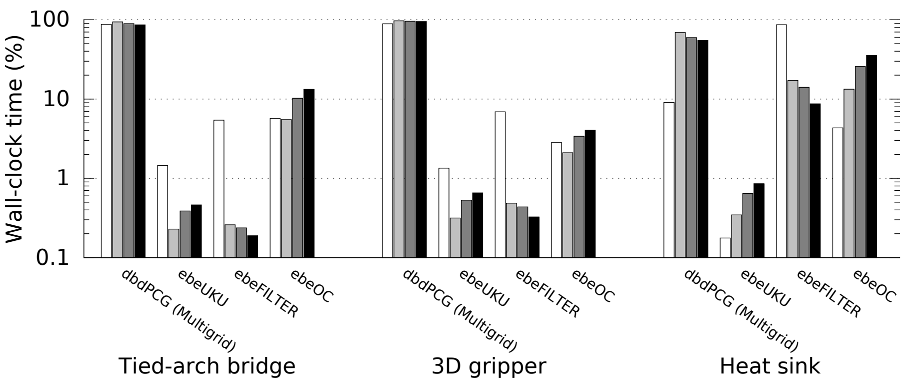
## 6. Conclusion

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 density-based topology optimization methods. The performance of the solving stage using GPU computing is analyzed using two preconditioning techniques; in particular, Jacobi preconditioner and geometric multigrid preconditioner. Different granularities are used to facilitate the exploitation of massive parallel architectures. The solving of the system of equations is implemented with granularity at the DoF level, whereas element grain size is used to process the calculation of sensitivities, the filtering strategy and the optimality criteria method. A nodal-based strategy is adopted for the coarsening using the geometric multigrid preconditioner and for prolongation and reduction operators. Significant speedup is achieved in the solving stage using constant memory for storing the common elemental matrix $\mathbf{K}_0$ and shared memory for storing the unknowns and the vector of elemental properties $\mathbf{d}$ of the topology optimization. Besides, the three-dimensional kernel to process the matrix-vector multiplication operations fits well for massive parallel processing showing good performance results.

The numerical results show that the proposed GPU instance achieves signifi-



(a)



(b)

Figure 13: Percentage of total wall-clock time using (a) Jacobi and (b) geometric multigrid preconditioning.

cant speedups for the computational demanding tasks involved in density-based topology optimization. Nevertheless, we have to remark that such speedups are calculated with respect to the sparse-matrix calculation using one CPU thread, and thus lower speedups are obtained using multiple CPU threads for the calculation. The numerical results also show that the best performance in wall-clock time is obtained using the geometric multigrid preconditioner in the GPU instance of the PCG. Moreover, the experiments show the efficiency of the proposed GPU implementation for solving topology optimization problems in three different fields: maximum stiffness design, heat sink design and compliant mechanism design. Furthermore, the numerical experiments show the scalability of the proposed GPU instance with the resources of the graphics units. This is a promising result to achieve higher speedups on new generations of graphics cards.

## 7. Acknowledgements

We gratefully acknowledge the support of NVIDIA Corporation with the donation of some of the GPU devices used for this research. Such a work has also been supported by the AEI/FEDER and UE under the contract DPI2016-77538-R and by the "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. Meth. Appl. Mech. Eng. 71 (1988) 197–224.

[3] F. Niu, S. Xu, G. Cheng, A general formulation of structural topology optimization for maximizing structural stiffness, Struct. Multidiscip. Optim. 43 (2011) 561–72.

[4] O. Sigmund, On the design of compliant mechanisms using topology optimization, Mech. Struct. Mach. 25 (1997) 493–524.

[5] T. Yamada, K. Izui, S. Nishiwaki, A Level Set-Based Topology Optimization Method for Maximizing Thermal Diffusivity in Problems Including Design-Dependent Effects, J. Mech. Des. 133 (2011) 1–9.

[6] M. Tinnsten, P. Carlsson, M. Jonsson, Stochastic optimization of acoustic response – a numerical and experimental comparison, Struct. Multidiscip. Optim. 23 (2002) 405–11.

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

[8] G. Allaire, E. Bonnetier, G. Francfort, F. Jouve, Shape optimization by the homogenization method, Numer. Math. 76 (1997) 27–68.

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

[10] G. Allaire, F. Jouve, A.-M. Toader, Structural optimization using shape sensitivity analysis and a level-set method, J. Comput. Phys. 194 (2004) 363–93.

[11] M. Y. Wang, X. Wang, D. Guo, A level set method for structural topology optimization, Comput. Methods Appl. Mech. 192 (2003) 227–46.

[12] J. Martínez-Frutos, D. Herrero-Pérez, M. Kessler, F. Periago, Robust shape optimization of continuous structures via the level set method, Comput. Methods Appl. Mech. Engrg. 305 (2016) 271–91.

[13] B. Bourdin, A. Chambolle, Design-dependent loads in topology optimization, ESAIM Contr. Optim. Calc. Var. 9 (2003) 19–48.

[14] M. K. Misztal, J. A. Bærentzen, Topology-adaptive interface tracking using the deformable simplicial complex, ACM Trans. Graph. 31 (2012) 1–12.

[15] A. N. Christiansen, M. Nobel-Jørgensen, N. Aage, O. Sigmund, J. A. Bærentzen, Topology optimization using an explicit interface representation, Struct. Multidisc. Optim. 49 (2014) 387–99.

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

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

[18] 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.

[19] D. Stošić, D. Stošić, T. Ludermir, B. Stošić, M. V. Milošević, GPU-advanced 3D electromagnetic simulations of superconductors in the Ginzburg–Landau formalism, J. Comput. Phys. 322 (2016) 183–98.

[20] B. Chrétien, A. Escande, A. Kheddar, GPU Robot Motion Planning Using Semi-Infinite Nonlinear Programming, IEEE Trans. Parallel Distrib. Syst. 27 (2016) 2926–39.

[21] 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.

[22] A. Kuzelewski, E. Zieniuk, GPU-based acceleration of computations in elasticity problems solving by parametric integral equations system, Adv. Eng. Software 79 (2015) 27–35.

[23] 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.

[24] W. Liu, J. W. Hong, Discretized peridynamics for brittle and ductile solids, Int. J. Numer. Methods Eng. 89 (2012) 1028–46.

[25] Q. V. Le, W. K. Chan, J. Schwartz, A two-dimensional ordinary, state-based peridynamic model for linearly elastic solids, Int. J. Numer. Methods Eng. 98 (2014) 547–61.

[26] M. D. Brothers, J. T. Foster, H. R. Millwater, A comparison of different methods for calculating tangent-stiffness matrices in a massively parallel computational peridynamics code, Comput. Meth. Appl. Mech. Eng. 279 (2014) 247–67.

[27] H. Courtecuisse, H. Jung, J. Allard, C. Duriez, D. Y. Lee, S. Cotin, GPU-based real-time soft tissue deformation with cutting and haptic feedback, Prog. Biophys. Mol. Biol. 103 (2010) 159–68.

[28] V. Strbac, J. V. Sloten, N. Famaey, Analyzing the potential of GPGPUs for real-time explicit finite element analysis of soft tissue deformation using CUDA, Finite Elem. Anal. Des. 105 (2015) 79–89.

[29] A. Bartezzaghi, M. Cremonesi, N. Parolini, U. Perego, An explicit dynamics GPU structural solver for thin shell finite elements, Comput. Struct. 154 (2015) 29–40.

[30] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. M. Buijssen, M. Grajewski, S. Turek, Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, Parallel Comput. 33 (2007) 685–99.

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

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

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

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

[35] J. Martínez-Frutos, D. Herrero-Peréz, GPU Acceleration for Evolutionary Topology Optimization of Continuum Structures Using Isosurfaces, Comput. Struct. 182 (2017) 119–36.

[36] F. Ramírez-Gil, E. 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.

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

[38] 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.

[39] 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.

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

[41] 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.

[42] N. Wilt, The CUDA Handbook: A Comprehensive Guide to GPU Programming, Addison-Wesley, 2013.

[43] N. Corporation, NVIDIA CUDA C Programming Guide, Version 7.5, 2015.

[44] G. I. N. Rozvany, A critical review of established methods of structural topology optimization, Struct. Multidiscip. Optim. 37 (2009) 217–37.

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

[46] M. P. Bendsøe, O. Sigmund, Material interpolation schemes in topology optimization, Archive of Applied Mechanics 69 (1999) 635–54.

[47] J. Petersson, O. Sigmund, Slope constrained topology optimization, Int. J. Numer. Methods Eng. 41 (1998) 1417–34.

[48] O. Sigmund, J. Petersson, Numerical instabilities in topology opitimization: A survey on procedures dealing with checkerboards, mesh-dependencies and local minima, Struc. Optim. 16 (1998) 68–75.

[49] O. Sigmund, K. Maute, Sensitivity filtering from a continuum mechanics perspective, Struct. Multidiscip. Optim. 46 (2012) 471–5.

[50] R. B. Wilson, A simplicial method for convex programming, Ph.D. thesis, Harvard University, 1963.

[51] K. Svanberg, The method of moving asymptotes–a new method for structural optimization, Int. J. Numer. Methods Eng. 24 (1987) 359–73.

[52] M. P. Bendsøe, Optimization of structural topology shape and material, Springer, New York, 1995.

[53] A. A. Groenwold, L. F. P. Etman, A quadratic approximation for structural topology optimization, Int. J. Numer. Methods Eng. 82 (2010) 505–24.

[54] M. Papadrakakis, G. Stavroulakis, A. Karatarakis, A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures, Comput. Meth. Appl. Mech. Eng. 200 (2011) 1490–508.

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

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

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

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

[59] A. Bejan, Constructal-theory network of conducting paths for cooling a heat generating volume, Int. J. Heat Mass Transf. 40 (1997) 799–816.

[60] O. Sigmund, Morphology-based black and white filters for topology optimization, Struct. Multidiscip. Optim. 33 (2007) 401–24.