Struct Multidisc Optim (2013) 48:473–485
DOI 10.1007/s00158-013-0920-y

RESEARCH PAPER

Toward GPU accelerated topology optimization on unstructured meshes

Tomás Zegard · Glaucio H. Paulino

Received: 18 November 2011 / Revised: 25 February 2013 / Accepted: 2 March 2013 / Published online: 12 April 2013

Abstract The present work investigates the feasibility of finite element methods and topology optimization for unstructured meshes in massively parallel computer architectures, more specifically on Graphics Processing Units or GPUs. Challenges in the parallel implementation, like the parallel assembly race condition, are discussed and solved with simple algorithms, in this case greedy graph coloring. The parallel implementation for every step involved in the topology optimization process is benchmarked and compared against an equivalent sequential implementation. The ultimate goal of this work is to speed up the topology optimization process by means of parallel computing using off-the-shelf hardware. Examples are compared with both a standard sequential version of the implementation and a massively parallel version to better illustrate the advantages and disadvantages of this approach.

Keywords Topology optimization · Graphics processing units · Finite element method · FEM · GPU · CUDA

1 Introduction

Nowadays processors achieve continuous improvements by increasing the level of parallelism, that is, increasing the number of processing cores while maintaining the clock frequency. Many-core processors are a type of processors that evolved to a high level of parallelism. A class of many-core processors are the *Graphics Processing Units*, or GPUs for short. We note that GPUs and *Central Processing Units* (or CPUs) have different design approaches (Fig. 1): the CPU is a general-purpose multi-core processor with many high level instructions, whereas the GPU is a many-core processor with a faster and smaller set of instructions (more specialized type of hardware) but capable of handling a large number of concurrent threads.[1]

In the year 2007, NVIDIA (GPU developer) released the *Compute Unified Device Architecture* (or CUDA) (NVIDIA 2007). Essentially, CUDA allows the programmer to use NVIDIA GPUs to develop massively parallel computing applications. In order to take advantage of the high level of parallelism in these many-core processors, the code must be especially crafted so that several tasks can be concurrently executed without interfering with each other by making efficient use of the hardware memory and ideally establishing minimal communication between the tasks. The GPU architecture traditionally operates in single precision, and double precision can be used with a performance hit. In addition, the floating point arithmetic has a lower standard compared to the CPU. All these factors must be kept in mind when crafting codes for the GPU (NVIDIA 2009).

The present work investigates the feasibility of implementing *Topology Optimization* (or TOP), a highly demanding computational algorithm based on the *Finite Element Method* (or FEM) on the GPU. The purpose of this work is not to excel in each and every component of the algorithm, but to implement the complete algorithm with sufficiently acceptable results by setting a foundation from where the implementation could be further improved, keeping up with

T. Zegard · G. H. Paulino (✉)
Department of Civil and Environmental Engineering,
Newmark Laboratory, University of Illinois
at Urbana-Champaign, 205 N. Mathews Avenue,
Urbana, IL 61801, USA
e-mail: paulino@illinois.edu

[1]A thread is an independent *unit of processing* that can handle and process a task.

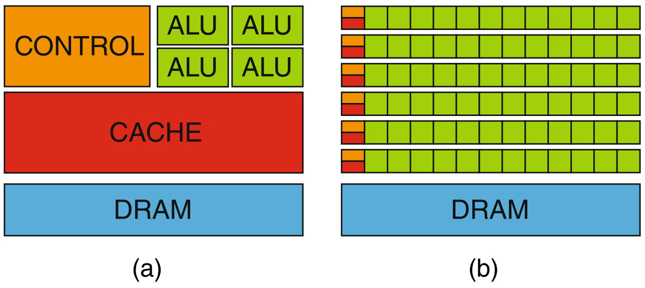


Fig. 1 Comparison of the basic diagram of a typical CPU and GPU [ALU: Arithmetic Logic Unit, DRAM: Dynamic Random-Access Memory] **a** Schematic of a CPU. **b** Schematic of a GPU

current and future trends in microprocessors. For that purpose, an implementation was developed to prove and test the concepts and ideas, as presented by Zegard and Paulino (2011).

The use of unstructured meshes allows the user to define complex domains, loads and restrictions. The practical advantage is clear over a structured mesh, especially for real life applications where the domain is rarely orthogonal, corners do not necessarily follow straight lines, and boundaries are curved. The characteristics of structured meshes can be exploited to easily address the parallel stiffness assembly problem, and the resulting global matrix has a structured profile that makes the system solution simple and efficient. Thus, implementations based on structured meshes show good performance results (Schmidt and Schulz 2011). However, the topology optimization algorithm loses flexibility with a structured mesh.

This work is organized as follows: Section 2 outlines the data flow, and discusses the main problems faced in the implementation of specific components of the algorithm. The examples in Section 3 compare the results obtained with this implementation and also benchmarks and profiles the different components. Finally, in Section 4 conclusions are given to condense the findings and describe future desirable changes and developments that would make the implementation more suitable for end-user requirements. The nomenclature, symbols and abbreviations used are listed in the Appendix A.

2 Computational implementation

The present implementation has two user-selectable compute chains, both with a possibility of an alternative solver, resulting in a total of 4 possible compute chains as depicted in Fig. 2. The topology optimization loop can take place entirely in the GPU, entirely in the CPU, or combining both. Previous work has been done using GPUs to speed up FEM routines (Cecka et al. 2011; Dziekonski et al. 2010;

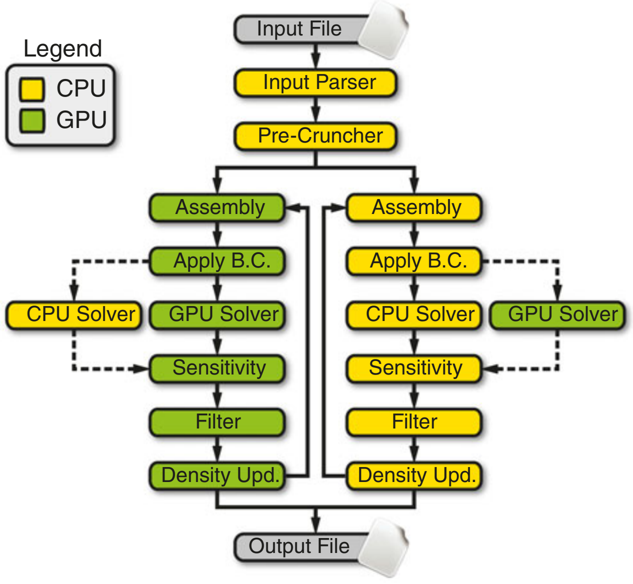


Fig. 2 Schematic flowchart of the topology optimization implementation developed for this work

Gödel et al. 2010; Guney 2010; Kakay et al. 2010; Liu et al. 2008). Concurrently, there have been large efforts into GPU based linear solvers (Carvalho et al. 2010; EM Photonics 2004; Tomov et al. 2009, 2010; Volkov and Demmel 2008). Topology optimization, being a computationally intensive task, has already been parallelized on traditional architectures (Mahdavi et al. 2006; Vemaganti et al. 2004) and on GPUs for structured meshes (Schmidt and Schulz 2011). The GPU architecture is better suited for structured problems or linear system of equations where the matrix follows some structure (banded, tri-banded, block-banded and others). This work explores feasibility and addresses topology optimization for unstructured two-dimensional meshes on GPUs. Nevertheless, most concepts can be extended to three-dimensional problems.

The topology optimization problem, using *Solid Isotropic Material with Penalization* (SIMP) (Bendsøe and Sigmund 1999; Bruns 2005) for minimum compliance, is as follows (Bendsøe 1989; Bendsøe and Kikuchi 1988; Bendsøe and Sigmund 2003; Rozvany 1997):

$$\begin{aligned}
\min_{\rho}: \quad & c(\rho) = \sum_{e=1}^{n} (\rho_e)^p \{\mathbf{u_e}\}^T [\mathbf{k_{e0}}] \{\mathbf{u_e}\} \\
\text{s. t.}: \quad & \sum_{e=1}^{n} (\rho_e V_e) - f V_0 = 0 \\
& 0 < \rho_{\min} \leq \rho \leq 1 \\
\text{with}: \quad & [\mathbf{K}]\{\mathbf{u}\} - \{\mathbf{f}\} = 0
\end{aligned} \tag{1}$$

where the objective function c is the compliance, $[\mathbf{K}]$ is the global stiffness matrix, $\{\mathbf{f}\}$ is the nodal force vector, ρ_e is the element density (design variable), p is a penalization factor to prevent *gray* densities, $\{\mathbf{u_e}\}$ are the element displacements, and $[\mathbf{k_{e0}}]$ is the local stiffness assuming $\rho = 1$. The constraints are the global FEM equilibrium, volume constraint and density constraint, respectively. The parameter V_e is the element volume, f is the specified volume fraction for the problem, V_0 denotes the volume or area of the entire domain and $\rho_{\min}$ is a minimum value of density that the design variable can attain to prevent system singularity.

The topology optimization loop was subdivided into six modules: FEM Assembly, Boundary Conditions, Solver, Sensitivity, Filter and Density (ρ) Update. For each one of these components, a CPU and a GPU version was written, the exception being the CPU Solver, where LAPACK, a well known Linear Algebra Package (AMD 2009; Anderson et al. 1999) distributed in the *AMD Core Math Library* (ACML), was used. Prior to the topology optimization loop, some calculations must be done. These calculations are done in the *Pre-cruncher* module. The current implementation runs entirely on single precision, on both the CPU and GPU components of the code. A file parser for ABAQUS input files (SIMULIA (Dassault Systèmes) 1978) (or any other commercial FEM software to that effect) was implemented to ease the data input, but the implementation is independent of the parser, and data may be supplied manually, or other parsers can be easily developed.

2.1 Preliminary computations

Preliminary calculations are performed by the *Pre-cruncher* module. The pre-cruncher obtains essential information for the implementation to run. Whether the CPU or GPU chain is selected, information from the pre-cruncher is required. The pre-cruncher runs in the CPU because it is used only once, and any improvement will have a minimal impact on the overall runtime. The main tasks done by the pre-cruncher are:

- Build element communications graph
- Graph coloring
- Calculate the bandwidth of [**K**]
- Calculate the element's area or volume
- Build the filter list

The communication graph (Paulino et al. 1994a, b) is used by both the graph colorer and the filter list construction. The graph coloring is required to assemble the global stiffness matrix [**K**] in a parallel way without falling into race conditions (further explained in the next section). The bandwidth of [**K**] is required by the assembly process and the solver to make the memory usage of [**K**] smaller and to traverse the entire matrix efficiently. The element and domain areas are required by the filter in the convolution function and in the material update module to enforce the volume fraction constraint. The filter list is required by the filter module to avoid searching through the mesh at every iteration (details on the filter are explained in a later section).

2.2 Graph coloring

There is a *race condition*[2] present in the assembly of the stiffness matrix [**K**] and the computation of the force vector {**f**}, the assembly of the global stiffness matrix being the most challenging to tackle.

There are different approaches possible for computing and assembling the local stiffness matrices. These approaches can be divided into two groups: the nodal approach and the element approach. The problem with the nodal approach is that there is a lot of over computation: each element's local stiffness matrix is computed as many times as it has nodes (for Q4 elements, each matrix is computed 4 times). An element-wise approach, on the other hand, does not discard any computation, but must be careful enough to make sure that no other thread is assembling to the same nodes at a given time. Thus

$$(DOF^i) \cap (DOF^j) = \emptyset \qquad \forall i, j \text{ with } i \neq j \tag{2}$$

To address this problem, an approach based on graph coloring is studied. Graph coloring is an attractive solution to the problem due to its simplicity and ability to render results with little overhead (Cecka et al. 2011; Oliker and Biswas 2000). The idea consists of assembling a specific set of elements that comply with (2) in parallel (each node has two *degrees of freedom*, or DOFs, corresponding to displacements in two-dimensional problems), and then move to a new set of elements (or new color in terms of graph coloring). With enough groups, the entire global stiffness matrix [**K**] can be assembled with no race condition problems.

When assembling the stiffness matrix of an element, no neighboring element can be assembled at the same time. The neighboring elements are stored by the *element communication matrix* or *graph*. The element communication graph $\mathbf{L}(G^C)$ is bidirectional (symmetric) and has a 1 (*or true*) if two elements i and j share one or more nodes ($\mathbf{L}_{i,j} = \mathbf{L}_{j,i} = 1$) or 0 (*or false*) if not (Paulino et al. 1994a, b). Depending on the specific algorithm, the diagonal may be filled with zeroes or ones, or used to store

[2] A race condition or *race hazard* is a flaw where the output and/or result of a process is wrong because two events that cannot take place at the same time race against each other to influence the result. In FEM this typically occurs when two local stiffness matrices [$\mathbf{k^e}$] are being added at the same time to the same positions in the global [**K**].

the total number of neighbors of each element. The greedy coloring algorithm (Gebremedhin et al. 2005), used in the present work, makes no use of this information, and therefore we take it to be 0 ($\mathbf{L}_{i,i} = 0$). For large meshes, $\mathbf{L}(G^C)$ is mostly composed of zeroes, and therefore a sparse storage scheme is used. Since $\mathbf{L}(G^C)$ is a *binary* matrix, the value does not require storage, only the positions where a 1 (or *true*) exists.

An example of this scheme can be seen for the mesh in Fig. 3a, where after numbering the elements in any desired order, the equivalent communication graph can be generated as in Fig. 3b. From the communication graph, the communication matrix can be easily derived:

$$\mathbf{L}(G^C) = \begin{bmatrix} 0&1&1&1&1&1&1&1\\ 1&0&0&1&1&1&0&1\\ 1&0&0&0&1&0&1&0\\ 1&1&0&0&0&1&0&0\\ 1&1&1&0&0&1&1&1\\ 1&1&0&1&1&0&0&1\\ 1&0&1&0&1&0&0&0\\ 1&1&0&0&1&1&0&0 \end{bmatrix}$$

The optimal solution for the graph coloring problem is the *chromatic number*, and is denoted by $\chi(G)$, for a given graph G (in our case the element communication graph). Graph coloring is in fact an NP-Complete problem (Dailey 1980), that is, the complexity scales in a non-polynomial way, and in most cases, the FEM problem size is such that the optimal solution cannot be found in reasonable time. Greedy coloring algorithms are sensitive to the way the

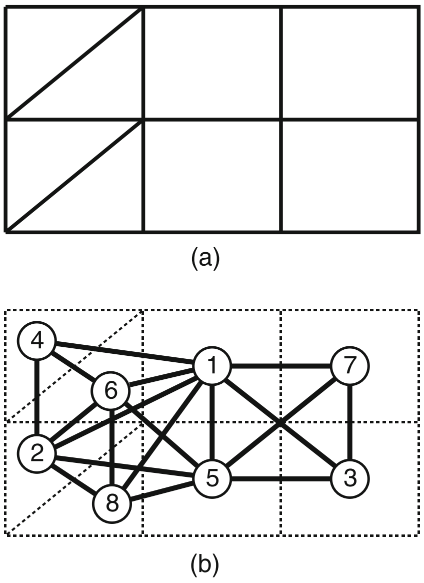


Fig. 3 Communication graph example (extracted from Paulino et al. (1994a)). **a** FEM mesh. **b** Element-based communication graph

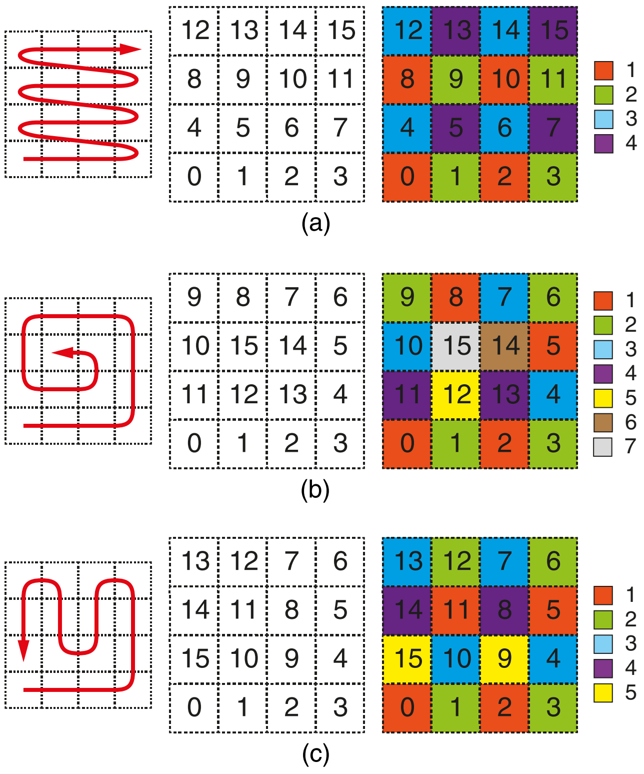


Fig. 4 Greedy coloring for a 4 × 4 structured mesh. Numbering schemes result in **a** 4 colors, **b** 7 colors and **c** 5 colors

data is traversed because they work in a *first-come first-served* basis (Gebremedhin et al. 2005; Kučěra 1991), with different outcomes depending on the numbering sequence (exemplified in Fig. 4). However, the hardware also limits the maximum number of concurrent threads which, in graph coloring terms, is the maximum number of elements a color can have. Numerical experiments indicate that the greedy algorithm subject to this constraint gives close to optimum solutions for very large problems, where the coloring is mainly dominated by the thread constraint.

2.3 Local stiffness matrices

In an element-wise approach, each thread computes a local stiffness matrix at a specific time step (or more correctly said, a *stride*). This requires local (temporal) memory to store and manipulate the local stiffness matrix $[\mathbf{k^e}]$, as well as the strain-displacement relation matrix $[\mathbf{B}]$ (spatial derivatives), the constitutive matrix $[\mathbf{D}]$, the Jacobian matrix $[\mathbf{J}]$, the Jacobian matrix inverse $[\mathbf{\Gamma}]$, the nodal coordinates $[\mathbf{XY}]$ and others. Current GPU architectures do not have a large local thread memory; thus the implementation must be efficient in its use to maximize the number of concurrent threads.

For a Q4 element, the size of $[\mathbf{k^e}]$ is 8×8. Using symmetry, only half is stored (lower triangular representation

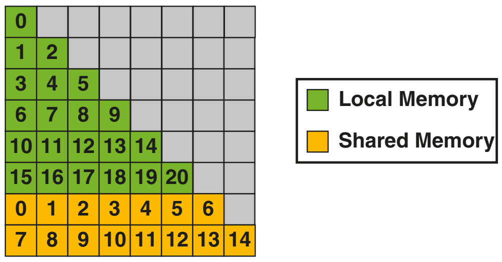


Fig. 5 Memory allocation for $\mathbf{k}^{\mathbf{e}}$

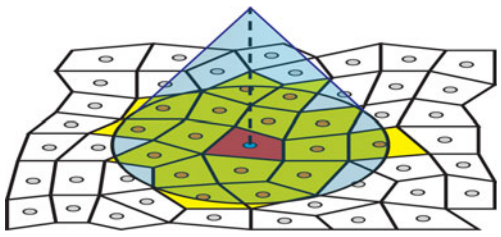


Fig. 6 Sensitivity filter with a linear convolution function (*cone*)

chosen), reducing the number of registers from 64 to 36. The local memory requirements per thread are still high. Therefore, the local stiffness matrices are split and stored half in shared memory and in registers as in Fig. 5, thus allowing more threads to be launched.

In addition to the registers used to store the local stiffness matrix, more registers are required to be used as counters, intermediate variables, other operations and such. The number of threads that are launched for the stiffness matrix assembly is 256, meeting the requirements of hardware and memory. In a SIMP approach, if the problem is not memory constrained, the computed local stiffness matrices can be stored and premultiplied by the penalized density prior to assembly.

2.4 Sensitivities filtering

To prevent the checkerboard pattern and address mesh dependence, a sensitivity filter (Bendsøe and Sigmund 2003; Sigmund 2001) is used. The sensitivity filter *blurs* the sensitivity distribution throughout the mesh, effectively eliminating the checkerboard pattern. It does not directly impose a member size constraint, but its effect is similar. If the mesh gets modified, but the filter parameters are kept, then the final topology should remain approximately the same (or exhibit little change).

The current implementation uses Q4 elements with a single material density variable per element located at the centroid (here the design variables are the density variables, but that is not a requirement). The filter requires the distance of an element's variable to that of a different element.

The filter for an element makes use of the sensitivities of all neighbors that fall within the projection of the convolution function. The convolution function typically used linearly decays to zero, being equivalent to a cone (Fig. 6), although other types of convolution functions can also be used. For this case the filter is:

$$\frac{\hat{\partial} c}{\partial \rho_i} = \left(\frac{1}{\rho_i V_i \sum_{j=1}^{n} \hat{H}_j} \right) \sum_{j=1}^{n} \hat{H}_j \rho_j V_j \frac{\partial c}{\partial \rho_j} \tag{3}$$

with the convolution operator (weight factor) $\hat{H}_j$ defined as:

$$\hat{H}_j = \begin{cases} R - \operatorname{dist}(i, j) \text{ if } \operatorname{dist}(i, j) \leq R \\ 0 \text{ if } \operatorname{dist}(i, j) > R \end{cases} \tag{4}$$

where dist$(\cdot)$ is the distance function (between the center points of the elements), R is the filter radius, ρ is the element's density, $\partial c/\partial \rho$ is the sensitivity, and $\hat{\partial} c/\partial \rho$ is the filtered sensitivity.

It is computationally expensive to find the elements that fall within an element's filter for an unstructured mesh. The communication matrix previously computed is used for an efficient search recursively through the element's neighbors. For a fixed mesh, the search can stored to be used at each iteration. The filter is applied in parallel on a per-element basis: a single thread reads the sensitivities of all the elements falling within a cone, applies the convolution, and stores the filtered sensitivity in a new vector.

2.5 Sensitivity

The sensitivity kernel shares most of its code with the assembly kernel, with a couple of exceptions. Since there is no assembly taking place, the kernel does not suffer from race condition problems. Also, this kernel computes the compliance; the kernel has to sum compliances from all elements and therefore a parallel binary reduction in shared memory is carried out. The sensitivity and the assembly kernel have similar memory requirements and the number of threads launched is also 256.

2.6 Optimality criteria

The topology optimization problem in (1) can be solved using Lagrange multipliers (Peressini et al. 1988). With the *adjoint method* (Haftka and Gürdal 1992), the objective function derivative is:

$$\frac{\partial \mathcal{L}}{\partial \rho_e} = -\{\mathbf{u_e}\}^T \frac{\partial [\mathbf{k_e}]}{\partial \rho_e} \{\mathbf{u_e}\} + \lambda V_e \tag{5}$$

where $\mathcal{L}$ is the Lagrangian and λ is a Lagrange multiplier.

Taking into account the box constraints, the condition of optimality is:

$$\begin{cases} \dfrac{\partial \mathcal{L}}{\partial \rho_e} \geq 0 \text{ if } \rho_e = \rho_{\min} \\ \dfrac{\partial \mathcal{L}}{\partial \rho_e} = 0 \text{ if } \rho_{\min} < \rho_e < 1 \\ \dfrac{\partial \mathcal{L}}{\partial \rho_e} \leq 0 \text{ if } \rho_e = 1 \end{cases} \tag{6}$$

Now the *Optimality Criteria* (OC) algorithm can be explained in light of expression (6) since it is essentially a fixed point iteration based on these results:

$$\rho_e = \left(\frac{1}{\lambda V_e} \{\mathbf{u_e}\}^T \frac{\partial [\mathbf{k_e}]}{\partial \rho_e} \{\mathbf{u_e}\} \right)^{\eta} \rho_{\text{old}} \tag{7}$$

where η is a numerical damping parameter, and good results are observed for $\eta = 1/2$. In addition, a *move limit* is imposed at each iteration to prevent divergence of the algorithm (Bendsøe and Sigmund 2003; Sigmund 2001):

$$\max(\rho_{\text{old}} - m, \rho_{\min}) \leq \rho_e \leq \min(\rho_{\text{old}} + m, 1) \tag{8}$$

where m is the move limit (to prevent the variables from changing too fast). The solution for λ is nonlinear, situation that gets worsened with the densities update in accordance with (8). Solving for λ is done by the *bisection method* (Sigmund 2001), modified to take advantage of the parallel architecture using the well known parallel binary reduction scheme. For each trial λ the total volume is obtained following a parallel binary reduction scheme, and depending on the value, a new half-space is selected for the new value of λ.

2.7 Solver

The global stiffness matrix $[\mathbf{K}]$ is symmetric and positive definite (requires $\rho_{\min} > 0$). The node numbering scheme will determine the bandwidth b_w. High levels of storage and computational efficiency can be attained if bandwidth reduction schemes are used (Cuthill and Mckee 1969; Gibbs et al. 1976a, b; Liu and Sherman 1976; Paulino et al. 1994a, b). The stiffness matrix is stored using a packed lower triangular banded scheme, with column-major ordering (to seamlessly interface with popular linear algebra libraries that typically use column-major ordering).

The modular nature of the implementation allowed for a variety of GPU and CPU solvers to be implemented. Comparing several commercial and handwritten solvers (Fig. 7), a Cholesky based solver in the GPU was selected to complete the topology optimization loop for unstructured meshes in the GPU (and the spbsv in LAPACK as the CPU counterpart). The following simple approach is used to compute the Cholesky decomposition of matrix $\mathbf{A} = \mathbf{L}\mathbf{L}^T$:

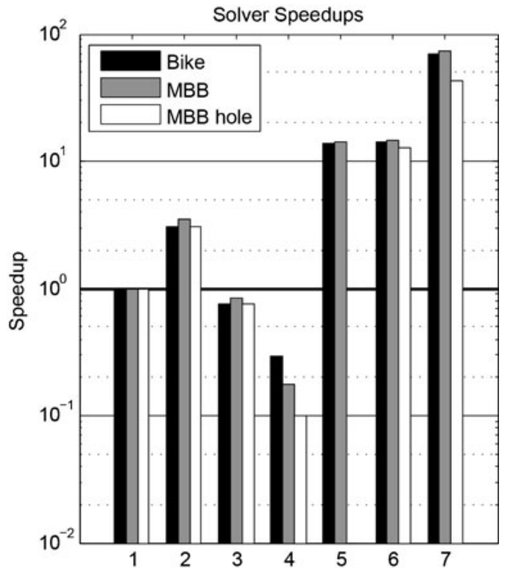


Fig. 7 Solver speedups for three problems of sizes $n = 41270, 87362$ and 118000 with $b_w = 350, 486$ and 690 respectively, with names *Bike*, *MBB* and *MBB hole*. Reference solver, *1* BLAS based PCGS. *2* Hand-written PCGS in CUDA. *3* cuBLAS based PCGS. *4* Hand-written PCGS in C++. *5* First iteration of a GPU Cholesky solver (Zegard and Paulino 2011), limited to $b_w \leq 513$. *6* Improved GPU Cholesky solver. *7* spbsv solver from LAPACK

$$\mathbf{L}_{i,i} = \sqrt{\mathbf{A}_{i,i} - \sum_{k=1}^{i-1} \mathbf{L}_{i,k}^2} \tag{9}$$

$$\mathbf{L}_{i,j} = \frac{\mathbf{A}_{i,j} - \sum_{k=1}^{j-1} \mathbf{L}_{i,k}\mathbf{L}_{j,k}}{\mathbf{L}_{j,j}} \tag{10}$$

If the matrix $\mathbf{A}$ is banded, then the lower triangular matrix $\mathbf{L}$ is also banded, and has the same bandwidth b_w.

The GPU solver has three components: the Cholesky decomposition for banded matrices (developed in the present work), and the solution of two triangular systems. The triangular system is solved using routines in cuBLAS (NVIDIA 2012), an implementation of the *Basic Linear Algebra Subprograms* (BLAS) in CUDA (Blackford et al. 2002), for maximum efficiency. Error accumulation should be prevented when possible in the implementation to solve large or badly conditioned systems (professional libraries such as BLAS, LAPACK or cuBLAS do this).

The Cholesky decomposition processes the matrix in parallel one column at a time, with a single thread propagating the computations across the rows as depicted in Fig. 8.

Exploiting the banded nature of the stiffness matrix is important in order to save computational cost and storage. Nonetheless, the use of this scheme has the consequence

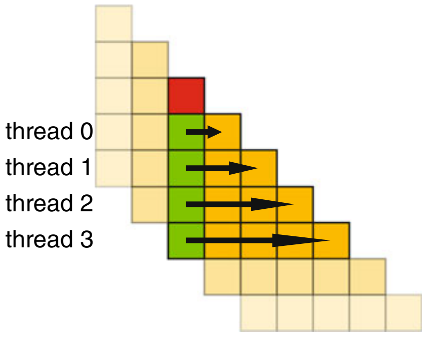


Fig. 8 Thread assignment for the Cholesky decomposition and substitution process

of hard to implement functions, poor performance and little attention for improvement in multi-threaded architectures (Remón et al. 2006, 2007). The nature of the current implementation is rather naïve, leaving the solver problem open. The complexity of the solver is $\mathcal{O}(nb_w^2)$; meaning that the solver will dominate the computation. In the present work attention was placed in the other components of the algorithm.

2.8 Loop stop criteria

In addition to a maximum number of iterations, an additional stopping criteria based on convergence is used. A change variable is monitored at each iteration and when it falls below a specified value, the iterations are stopped. One of the most typical change variables is the *infinity norm* of the change in densities ρ_e (Sigmund 2001):

$$change_1 = \|\rho^{\text{new}} - \rho_e\|_\infty$$
$$change_1 = \max(\rho_e^{\text{new}} - \rho_e) \tag{11}$$

This change variable is not suitable for unstructured meshes because it treats elements of different size equally. A problem occurs when material oscillates between a pair of very small elements. The GPU is more sensitive to this problem because of the lower floating point precision it operates with (e.g. no denormalization). Taking into account the element size, the change variable is improved:

$$change_2 = \|V_e(\rho_e^{\text{new}} - \rho_e)\|_\infty$$
$$change_2 = \max(V_e[\rho_e^{new} - \rho_e]) \tag{12}$$

The change variable presented in (12) has a violent drop whenever the algorithm is converging. In large problems, a

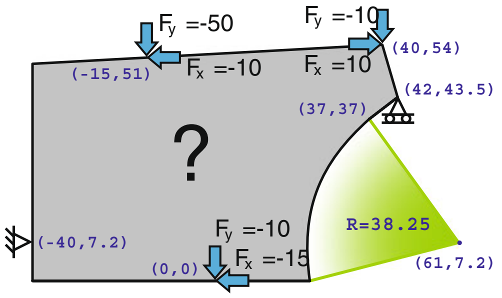


Fig. 9 Bike domain, loads and boundary conditions

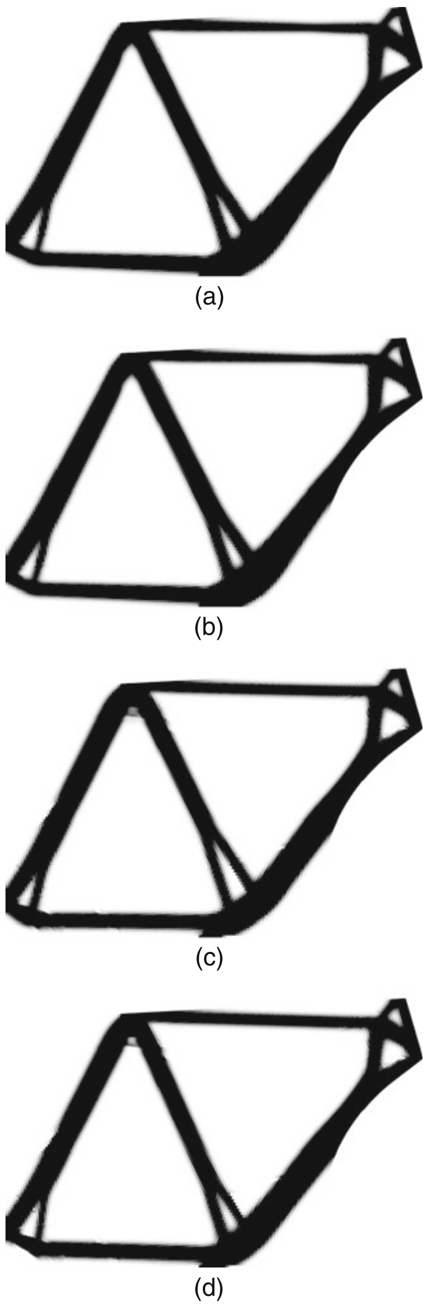


Fig. 10 Bike frame results after 30 iterations. **a** CPU **b** CPU -X **c** GPU and **d** GPU -X

finer control over the iterations is desirable. To address this, a different change variable based on the *1-norm* (also called *taxicab norm* or *manhattan norm*) can be used, and, to make the *1-norm* mesh-independent, it is divided by the number of elements n of the mesh:

$$change_3 = \frac{\|\rho_e^{\text{new}} - \rho_e\|_1}{n}$$
$$change_3 = \frac{\sum_{e=1}^{n} |\rho_e^{\text{new}} - \rho_e|}{n} \tag{13}$$

The change variable presented in (13) still suffers from the problem where very small elements can oscillate material among them and prevent convergence. Combining the ideas behind (12) and (13), we arrive at the resistant and controllable change variable that is used in the present work:

$$change = \frac{\sum_{e=1}^{n} V_e |\rho_e^{\text{new}} - \rho_e|}{V_0} \tag{14}$$

where V_0 is the domain volume, that is $V_0 = \sum_{e=1}^{n} V_e$. The change variables presented here (except for 12) range from 0 to the move limit m ($0 < change \leq m$). This makes the stop criteria simple by defining a tolerance and breaking the loop whenever $change \leq tolerance$. The computation of the change variable is done on a per-element basis, and then reduced following a parallel binary reduction scheme.

3 Examples & benchmarks

The performance, particularities and results for all possible compute chains are compared using 3 emblematic problems:

1. Bike frame
2. Messerschmitt–Bölkow–Blohm (MBB) beam
3. MBB beam with holes

The bike frame conceptual design is a somewhat applied problem that calls for an unstructured mesh analysis. The MBB beam is a typical problem in topology optimization (Bendsøe and Sigmund 2003), and it is of particular interest since an undesirable condition gets triggered on the implementation if certain parameters are used. The MBB beam with holes is a variation of the typical MBB problem, where an unstructured mesh is required to properly represent the domain restrictions.

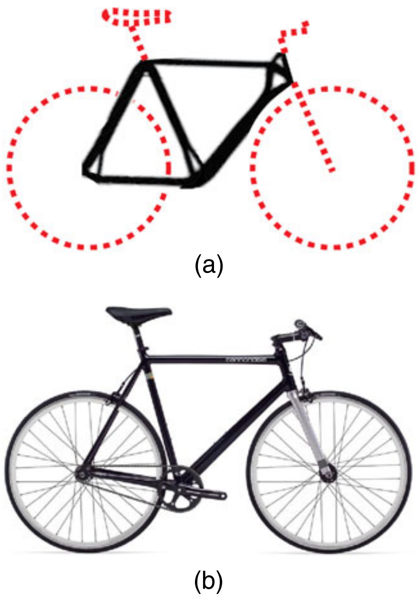


Fig. 11 **a** Final bike frame design with remaining components traced **b** Bike available in the market for comparison (2010 Cannondale Capo 2 urban commuter bike. Extracted from Cannondale (2010))

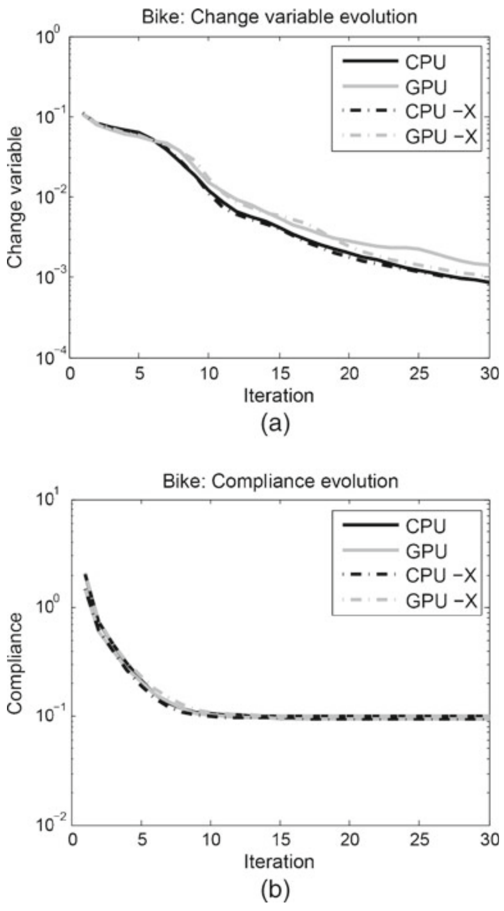


Fig. 12 Variable evolution for the bike frame problem. **a** Change variable and **b** compliance

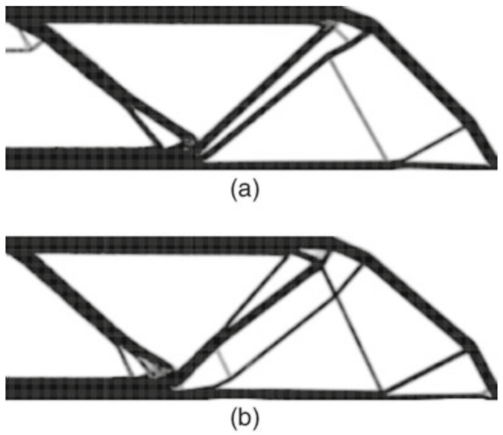


Fig. 13 MBB beam results after 30 iterations. **a** CPU and **b** GPU-X

The full implementation is benchmarked against the pure CPU chain. The compute chains (Fig. 2) available to the user and their labels are:

- TOP loop is entirely on the CPU: **CPU**
- TOP loop is entirely on the GPU: **GPU**
- CPU chain with GPU solver: **CPU -X**
- GPU chain with CPU solver: **GPU -X**

The penalization for all problems is $p = 3$, the minimal density is $\rho_{\min} = 0.001$, the move limit $m = 0.2$, and the maximum number of iterations is 30.[3]

3.1 Bike frame

Bike frames continuously seek for lighter and stronger designs. This is a simplified problem based on real bike frame geometries, and meets several domain restrictions (front wheel space to turn, location and inclination of the topmost bar, the total span of the frame and others). Loads are artificial but are expected to resemble a plausible loading scenario. The loads and the domain are explained in Fig. 9, with all units in centimeters and kilograms-force.

The mesh has 20,378 elements, 20,635 nodes and the resulting stiffness matrix bandwidth is 350. The material is assumed to be Aluminum with a unit depth, elastic modulus $E = 700{,}000$ kgf/cm^2, and $\nu = 0.3$. Topologies for all compute chains with a volume fraction $f = 0.3$, filter radius $R = 1.2$ and 30 iterations are shown in Fig. 10. The different machine precision, floating point standards, and operations order (cancelation effects) make for slight differences in the final design. Figure 11a shows the result from the hybrid GPU chain (GPU -X), with the remaining bike components traced in dashed lines. Analysis of the change variable and the compliance show a stable and steady decrease for all cases (Fig. 12).

[3]The hardware used for all benchmarks consists of a dual-socket dual-core AMD Opteron 2216, 8 GB of RAM and a NVIDIA Tesla T10 GPU with 4 GB of RAM.

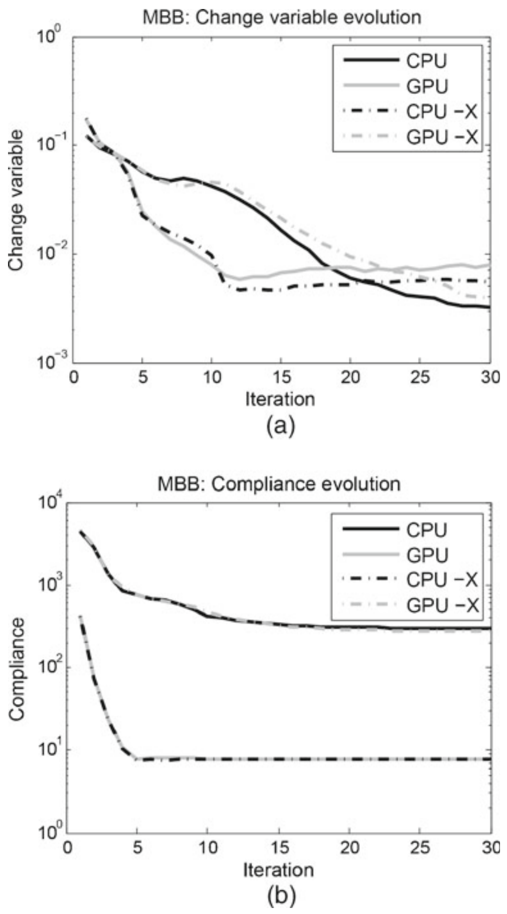


Fig. 14 Variable evolution for the MBB beam problem. **a** Change variable (14) and **b** compliance

3.2 Messerschmitt–Bölkow–Blohm (MBB) beam

The mesh for this problem is structured. Nevertheless, the current implementation makes no difference between structured or unstructured meshes. The problem parameters are typical: $\nu = 0.3$, $E = 100$, $f = 0.3$ and $height : length = 1 : 6$ that on the half-domain due to symmetry is $1 : 3$ (Bendsøe and Sigmund 2003; Sigmund 2001). The mesh has 43,200 elements, 43,681 nodes and a bandwidth of 486. The filter radius is set to $R = 0.02$, and interestingly enough, the problem fails to converge with the default parameters using the GPU solver. The standard GPU floating point arithmetic (single precision and no denormalization) have enough effect to make the system singular after a couple iterations. The stability can be regained if $\rho_{\min}$ is increased (note that $\rho^p_{\min} = 10^{-9}$) or if the filter radius R is increased. Results for the compute chains that converged are shown in Fig. 13.

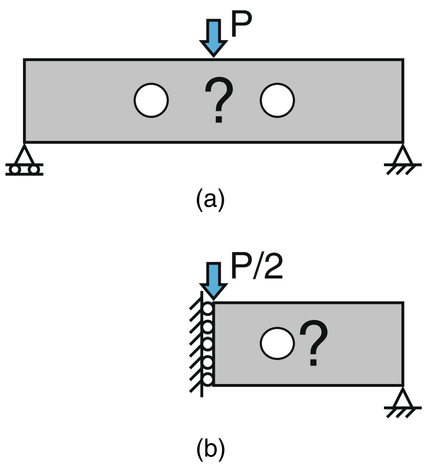


Fig. 15 MBB beam with holes problem. **a** Full beam and **b** half domain equivalent thanks to symmetry

The compliance and change variable in the CPU and GPU -X chains decrease monotonically (Fig. 14). The compliance and change variable have a noticeable drop once the system becomes singular for the compute chains that make use of the GPU solver.

3.3 MBB beam with holes

The MBB beam with holes problem is a variation of the typical MBB beam that includes two circular holes as a restriction in the design domain (one hole for the half domain). The domain, loading and boundary conditions are given in Fig. 15. The mesh has 55,200 elements, 55,900 nodes and a bandwidth of 690. The topologies were obtained for a volume fraction $f = 0.3$, filter radius $R = 0.02$ and 30 iterations (same as the MBB beam problem). Final converged topologies for this problem are given in Fig. 16. The problem again fails for the specified $\rho_{\min}$ and higher values or double precision are required for the problem to converge.

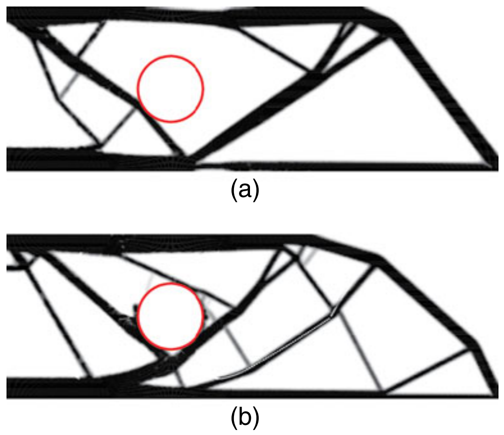
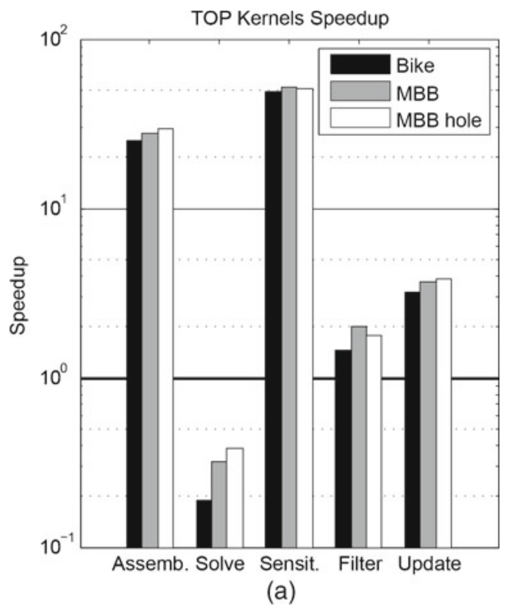


Fig. 16 Topology optimization results for MBB beam with holes after 30 iterations. **a** CPU and **b** GPU-X

Interestingly, the hybrid GPU chain (GPU -X) results in a design with a larger number of fine topologies, that has a closer similarity with a Michell-type solution (Hemp 1973; Michell 1904), despite having the same filter parameters of the CPU chain. Although this is probably just a coincidence.

3.4 Benchmarks

The speedups of each kernel compared to their CPU counterpart for each one of the three problems is shown in Fig. 17a. Speedup for the assembly kernel is comparable to results previously observed in the literature (Cecka et al.



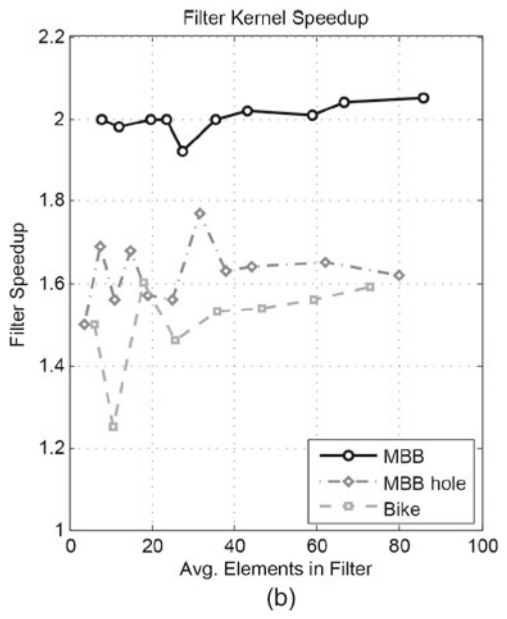


Fig. 17 GPU implementation benchmarks. **a** Individual kernel speedup and **b** filter speedup with filter size

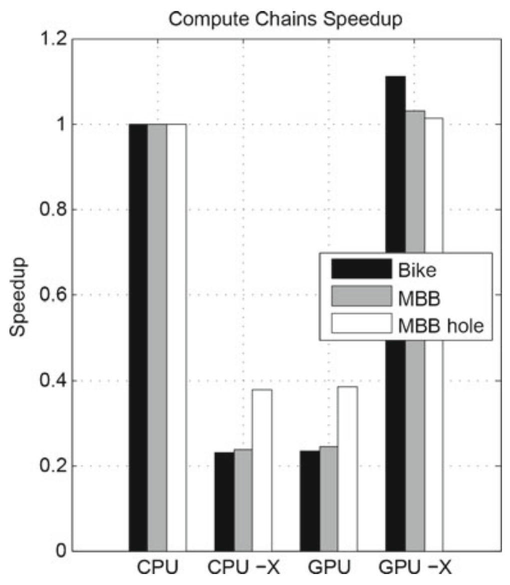


Fig. 18 GPU implementation speedup

2011), and because this kernel is similar to the sensitivity kernel, good results are observed there as well. There is no benchmark for the boundary conditions kernel since the runtime is often below 1 ms for the GPU. The filter kernel speedups from Fig. 17a are obtained taking the filter radius as used previously for each problem ($R = 1.2$ for the bike problem, and $R = 0.02$ for the MBB and the MBB with holes problem), nevertheless, to further analyze the filter speedup, its scaling with respect to the filter size is studied. Comparing the filter speedup with the average filter size of all the elements in the mesh (quasi mesh-independent parameter) as in Fig. 17b, the filter speedup is conservatively taken to be close to 1.6.

The total runtime speedups are in Fig. 18. Runtimes in seconds are available in Table 1. Good results and speedup were obtained for all modules but the solver. The solver takes about 90 to 95 % of the total runtime, and dominates the overall runtime speedup.

The GPU implementation scaling is provided in Fig. 19 for 5 iterations of an increasingly refined MBB problem (nodes optimized by the *Reverse Cuthill McKee* algorithm (Liu and Sherman 1976)). The largest problem has $n = 499{,}970$ and $b_w = 1158$. As indicated by the results in

Table 1 Runtime in seconds for all three problems for 30 topology optimization iterations

Problem	CPU	CPU -X	GPU	GPU -X
Bike	62.8	274.9	267.5	56.5
MBB	192.2	809.0	792.5	186.7
MBB hole	729.8	1,926.3	1,897.6	720.8

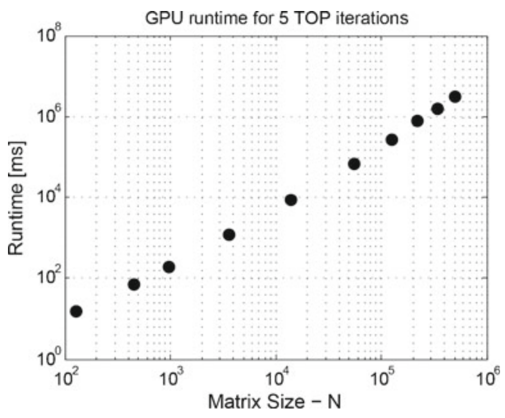


Fig. 19 GPU implementation scaling

Fig. 19, there is room for improvement in the linear solvers, however, this area is beyond the scope of this work.

4 Conclusions

Topology optimization in GPUs for unstructured meshes is indeed plausible. The solver dominates the overall speedup of the present implementation (it takes up approximately 90 % of the topology optimization runtime). Various solvers (some of them commercial) were tested and compared. Iterative solvers can be further explored (Remón et al. 2007) as they have fewer difficulties as the problem becomes ill-conditioned. The present GPU implementation shows significant speedups for the phases of assembly, sensitivities, filters and design updates.

The GPU operates under single precision and only supports normalized floating points in earlier NVIDIA architectures (based on the G80 core). This explains small differences in the resulting designs and, in some cases, difficulty to converge. Certain measures can alleviate this problem, but for an actual topology optimization solution, double precision has to be used.

Because the mesh is unstructured, some work must be carried out before the topology optimization loop. To prevent a possible race condition, a fast greedy coloring algorithm is used with excellent results due to the maximum number of threads constraint. In addition, filtering for unstructured meshes is not trivial because of the unordered memory access and the mesh search. Although this far-from-ideal situation for the GPU was encountered, speedup was obtained in the filter.

Despite the poor overall performance caused by the solver, the speedup obtained in the other components does not only concern topology optimization, and can be successfully ported to other applications such as traditional FEM (assembly kernel), or imaging and sampling (filter's convolution operation).

Future improvements in the GPU architectures are likely to eliminate most of the problems encountered here (mainly speed and precision) due to the rapid evolution of the GPU architecture compared to the CPU. Sparse algebra seems to be a move in the right direction considering the good performance obtained using numerical solvers in combination with sparse linear algebra (Schmidt and Schulz 2011), and it paves the way towards efficient three-dimensional problems.

The implementation could be further improved in many aspects. The changes that are most appealing to the authors include consideration of three-dimensional problems and support for the CAMD (Continuous Approximation of Material Distribution) approach (Matsui and Terada 2004). Further details along this line of reasoning can be found in Zegard (2010).

Acknowledgments We also thank Dr. Cameron Talischi for his help in the preparation of this manuscript. We acknowledge support from the National Science Foundation (NSF) under grant 1321661, and from the Donald B. and Elizabeth M. Willett endowment at the University of Illinois at Urbana-Champaign (UIUC).

Appendix A: Nomenclature

A.1 Symbols

$[\mathbf{B}]$	Strain-displacement matrix
b_w	Bandwidth of a matrix
c	Compliance
$[\mathbf{D}]$	Constitutive matrix
E	Young's modulus
f	Volume fraction
$\{\mathbf{f}\}$	Global force vector
$\hat{H}_j$	Convolution function
$[\mathbf{J}]$	Transformation Jacobian
$[\mathbf{K}]$	Global stiffness matrix
$[\mathbf{k}]$	Local stiffness matrix
$[\mathbf{L}]$	Lower triangular matrix
$\mathbf{L}(G^C)$	Communication matrix for graph G
m	Density move limit
n	Number of elements, matrix size or other depending on the context
p	Penalization factor for SIMP
R	Filter radius
u	Displacement
V	Volume
$[\mathbf{XY}]$	Nodal coordinates of an element
$[\Gamma]$	Inverse of the transformation Jacobian $[\mathbf{J}]$
η	Numerical damping parameter
$\mathcal{L}$	Lagrangian function
λ	Lagrange multiplier
ν	Poisson's ratio
ρ	Density
$\chi(G)$	Chromatic number for graph G

A.2 Abbreviations

ALU	Arithmetic Logic Unit
CAMD	Continuous Approximation of Material Distribution
CPU	Central Processing Unit
CUDA	NVIDIA's Compute Unified Device Architecture
DOF	Degree Of Freedom
DRAM	Dynamic Random-Access Memory
FEM	Finite Element Method
GPU	Graphics Processing Unit
MBB	Messerschmitt–Bölkow–Blohm
OC	Optimality Criteria
PCGS	Pre-conditioned Conjugate Gradient Solver
SIMP	Solid Isotropic Material with Penalization
TOP	Topology Optimization

References

AMD (2009) ACML - AMD Core Math Library v4.3.0. http://developer.amd.com/cpu/Libraries/acml/Pages/default.aspx. Accessed Jan 2010

Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999) LAPACK users' guide

Bendsøe MP (1989) Optimal shape design as a material distribution problem. Struct Optim 1:193–202

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

Bendsøe MP, Sigmund O (1999) Material interpolation schemes in topology optimization. Arch Appl Mech 69:635–654

Bendsøe MP, Sigmund O (2003) Topology optimization: theory, methods and applications. Engineering Online Library, 2nd edn. Springer, Berlin, Germany

Blackford LS, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A, Pozo R, Remington K, Whaley RC (2002) An updated set of basic linear algebra subprograms BLAS. ACM Trans Math Softw 28(2):135–151

Bruns TE (2005) A reevaluation of the SIMP method with filtering and an alternative formulation for solid-void topology optimization. Struct Multidiscip Optim 30(6):428–436

Cannondale (2010) Cannondale Bicycle Corporation. http://www.cannondale.com/. Accessed May 2011

Carvalho RF, Martins CAPS, Batalha RMS, Camargos AFP (2010) 3D parallel conjugate gradient solver optimized for GPUs. In: Digests of the 2010 14th biennial IEEE conference on electromagnetic field computation (CEFC). IEEE

Cecka C, Lew AJ, Darve E (2011) Assembly of finite element methods on graphics processors. Int J Numer Methods Eng 85(5):640–669

Cuthill E, Mckee J (1969) Reducing the bandwidth of sparse symmetric matrices. In: 24th national conference of the ACM, pp 157–172

Dailey DP (1980) Uniqueness of colorability and colorability of planar 4-regular graphs are NP-complete. Discrete Math 30(3):289–293

Dziekonski A, Lamecki A, Mrozowski M (2010) Jacobi and Gauss-Seidel preconditioned complex conjugate gradient method with GPU acceleration for finite element method. In: 40th European microwave conference, pp 1305–1308

EM Photonics (2004) CULA Tools - GPU Accelerated LAPACK. http://www.culatools.com/. Accessed Oct 2012

Gebremedhin AH, Manne F, Pothen A (2005) What color is your Jacobian? Graph coloring for computing derivatives. SIAM Rev 47(4):629–705

Gibbs NE, Poole Jr WG, Stockmeyer PK (1976a) A comparison of several bandwidth and profile reduction algorithms. ACM Trans Math Softw 2(4):322–330

Gibbs NE, Poole Jr WG, Stockmeyer PK (1976b) An algorithm for reducing the bandwidth and profile of a sparse matrix. SIAM J Numer Anal 13(2):236–250

Gödel N, Schomann S, Warburton T, Clemens M (2010) GPU accelerated Adams-Bashforth multirate discontinuous Galerkin FEM simulation of high-frequency electromagnetic fields. IEEE Trans Magn 46(8):2735–2738

Guney ME (2010) High-performance direct solution of finite element problems on multi-core processors. PhD thesis, Georgia Insitute of Technology, Atlanta, GA

Haftka RT, Gürdal Z (1992) Elements of structural optimization Solid mechanics and its applications series, 3rd edn. Kluwer, Norwell, MA

Hemp WS (1973) Optimum structures Oxford engineering science series. Clarendon Press, Oxford, UK

Kakay A, Westphal E, Hertel R (2010) Speedup of FEM micromagnetic simulations with graphical processing units. IEEE Trans Magn 46(6):2303–2306

Kučera L (1991) The greedy coloring is a bad probabilistic algorithm. J Algor 12(4):674–684

Liu W-H, Sherman AH (1976) Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. SIAM J Numer Anal 13(2):198–213

Liu Y, Jiao S, Wu W, De S (2008) GPU accelerated fast FEM deformation simulation. IEEE Asia Pac Conf Circ Syst606–609

Mahdavi A, Balaji R, Frecker M, Mockensturm EM (2006) Topology optimization of 2D continua for minimum compliance using parallel computing. Struct Multidiscip Optim 32(2):121–132

Matsui K, Terada K (2004) Continuous approximation of material distribution for topology optimization. Int J Numer Methods Eng 59(14):1925–1944

Michell AGM (1904) The limits of economy of material in frame-structures. Philos Mag Ser 8(47):589–597

NVIDIA (2007) CUDA programming guide. http://www.nvidia.com/. Accessed June 2009

NVIDIA (2009) CUDA C programming - best practices guide. http://www.nvidia.com/cuda/. Accessed June 2009

NVIDIA (2012) cuBLAS - CUDA Basic Linear Algebra Subroutines http://developer.nvidia.com/cublas. Accessed Oct 2012

Oliker L, Biswas R (2000) Parallelization of a dynamic unstructured algorithm using three leading programming paradigms. IEEE Trans Parallel Distrib Syst 11(9):931–940

Paulino GH, Menezes IFM, Gattass M, Mukherjee S (1994a) Node and element resequencing using the Laplacian of a finite element graph: part I - general concepts and algorithm. Int J Numer Methods Eng 37(9):1511–1530

Paulino GH, Menezes IFM, Gattass M, Mukherjee S (1994b) Node and element resequencing using the Laplacian of a finite element graph: part II - implementation and numerical results. Int J Numer Methods Eng 37(9):1531–1555

Peressini AL, Sullivan FE, Uhl Jr JJ (1988) The mathematics of nonlinear programming Undergraduate texts in mathematics series. Springer-Verlag, New York

Remón A, Quintana-Ortí E, Quintana-Ortí G (2006) Cholesky factorization of band matrices using multithreaded BLAS. In: PARA 2006, pp 608–616

Remón A, Quintana-Ortí E, Quintana-Ortí G (2007) The implementation of BLAS for band matrices. In: PPAM 07, pp 668–677

Rozvany GIN (1997) Topology optimization in structural mechanics. CISM International Centre for Mechanical Sciences. Springer, New York, NY

Schmidt S, Schulz V (2011) A 2589 line topology optimization code written for the graphics card. Comput Vis Sci 14(6): 249–256

Sigmund O (2001) A 99 line topology optimization code written in Matlab. Struct Multidiscip Optim 21(2):120–127

SIMULIA (Dassault Systèmes) (1978) Abaqus FEA. http://www.3ds.com/products/simulia/overview/. Accessed Dec 2012

Tomov S, Nath R, Du P, Dongarra J (2009) MAGMA users' guide v0.2. http://icl.cs.utk.edu/magma/. Accessed Dec 2012

Tomov S, Nath R, Ltaief H, Dongarra J. (2010) Dense linear algebra solvers for multicore with GPU accelerators. In: 2010 IEEE international symposium on parallel & distributed processing workshops and PhD forum (IPDPSW)

Vemaganti K, Lawrence WE, Parallel methods for topology optimization (2004). Comput Methods Appl Mech Eng 194(34–35):3637–3667

Volkov V, Demmel JW (2008) Benchmarking GPUs to tune dense linear algebra. In: 2008 ACM/IEEE conference on supercomputing

Zegard T (2010) Topology optimization with unstructured meshes on graphics processing units GPUs. Ms thesis, University of Illinois at Urbana-Champaign

Zegard T, Paulino GH (2011) GPU-based topology optimization on unstructured meshes. In: 11th US National Congress on computational mechanics