Use of modern GPUs in Design Optimisation

Aissa, Mohamed; Verstraete, Tom; Vuik, Kees

Publication date
2014

Document Version
Final published version

Published in
Proceedings of the 10th ASMO UK Conference Engineering Design Optimization

Use of modern GPUs in Design Optimization

M.H. Aissa* and T. Verstraete†
Von Karman Institute for Fluid Dynamics, Sint-Genesius-Rode, 1640, Belgium

K. Vuik‡
Delft University of Technology, 2628 CD Delft, the Netherlands

Graphics Processing Units (GPUs) are a promising alternative hardware to Central Processing Units (CPU) for accelerating applications with a high computational power demand. In many fields researchers are taking advantage of the high computational power present in GPUs to speed up their applications. These applications span from data mining to machine learning and life sciences. The field of design optimization in particular benefits from this alternative hardware. The automated search on the design space has been delegated to GPUs or to a system of CPUs assisted by GPUs. This paper is among the firsts to review the use of GPUs especially for design optimization. The focus is on topology optimization, shape optimization and multidisciplinary design optimization (MDO). The target is to provide an overview not only on the progress made in design optimization using GPUs but also to highlights limitations that researchers have to cope with and the areas that require more research.

Nomenclature

CG	=	Conjugate-Gradient solver
CPU	=	Central Processing Unit
CUDA	=	Compute Unified Device Architecture
CSM	=	Computational Structural Mechanics
FD	=	Finite Difference
HPC	=	High Performance Computing
GPU	=	Graphics Processing Unit
MDO	=	Multidisciplinary Design Optimization
MG	=	Multigrid
MPI	=	Message Passing Interface
PCG	=	Preconditioned Conjugate Gradient solver
SIMP	=	Solid Isotropic Microstructure with Penalization

I. Introduction

Design engineers are interested in identifying rapidly a design with optimal performance under specified constraints. This problem is easily formulated with the help of one or more objective functions that depend on typically a large number of design variables. The optimization process then requires finding a design from the design space, which minimizes the objective function respecting the set of constraints. For engineering problems, the design space is large and the objective function is relatively complicated, which leads to a computational intensive problem.

Optimization methods can be roughly classified by the information required from the evaluation process. Zero-order methods require only the function evaluation of the objective function. While first-order methods require additionally the gradients of the objective function with respect to all design variables.

Most of Zero-order optimization methods are nature-inspired and based on meta-heuristic. Some gradient-free methods improve a single design by exploiting its neighborhood in the design space through local search such as Tabu search and simulated annealing [1]. Other zero-order methods are population-based such as evolutionary algorithms and swarm intelligence [1]. These methods explore the design space with a multitude of interacting and evolving designs.

Communicating author: aissa@vki.ac.be
* Research Engineer, Turbomachinery Department.
† Assistant Professor, Turbomachinery Department.
‡ Professor, Institute of Applied Mathematics.

The first order optimization methods, such as steepest descent and Conjugate Gradient, have a better convergence behavior at an extra computational cost for derivative information.
The objective function itself is problem specific. It can be as simple as a minimization of a total pressure loss in a channel ($\int(p_{02} - p_{01})\, dA$) or also more complicated nonlinear functions. In topology optimization, approaches such as the general Solid Isotropic Microstructure with Penalization (SIMP) help formulating the objective function. In shape optimization the design can be changed within a fixed topology (e.g. number of holes). The objective function can be derived from aerodynamics considerations, structural considerations, heat transfer considerations or other disciplines. In MDO the objective functions are originating from the interaction of different disciplines (e.g. structure mechanics, aerodynamics), where different levels of interactions exist (uncoupled, one way coupled or strongly coupled).

The definition of the objective function often depends on the solution of partial differential state equations (PDE). For topology optimization, the function evaluation is in many cases delegated to a method of Computational Structural Mechanics (CSM) for which Finite Element discretization are the most popular. The evaluation of the objective function requires then to solve a set of linear equations of the form $A.x = b$ with x the results from which the objective function depends, matrix A the problem specific system matrix depending on the design variables and vector b a problem specific right-hand side potentially depending on the design variables. In aerodynamic shape optimization, a CFD method performs the evaluation solving the non-linear governing equations (Navier-Stokes, Euler). In multidisciplinary design optimization (MDO), CFD, CSM and eventual other methods can work in a segregated or interactive manner to perform the function evaluation.

The complexity of the optimization problem, as described above, leads to algorithms with large demand on computational resources. Thus high computational power and large memory resources are required to solve repeatedly the CSM and CFD models responsible for the objective evaluation. Three types of algorithm optimization[2] concerning the convergence of the methods encourage the wide application of design optimization: mathematically, physically and computer science based optimizations. Physically-based optimization reduces the complexity of the objective function evaluation by replacing it with a less complicated model (metamodel) that generates a faster but less accurate design evaluation. The delicate task is to combine high fidelity (original objective function evaluations) and low fidelity (metamodel) evaluations to accelerate the design optimization process and keep the evaluation accurate enough. A mathematical-based optimization takes advantage of preconditioner and multigrid techniques to accelerate solving the system of linear equations. Finally a computer-science motivated optimization is to use high performance computing. The latter type of optimization is the central focus of this review.

The high performance systems can be classified roughly following the memory architecture. The two main systems are then shared-memory and distributed-memory systems. For the shared-memory configuration, a set of processor shares the same memory area. Modern duo-core and quad-core CPU belong to this group. The programming interface (API) OpenMP[3] handles the parallelization in this system. Few simple compiler directives (#pragmas) surrounding sequential for-loops divide automatically the work between available cores. Every core processes a part of the loop. Processor communication is very simple since they are all sharing the same memory contingent. The maximal available number of cores for this system is however too low to cover large-scale problems (maximum by Xeon phi with 60 cores[4]). In the distributed-memory configuration, better known as cluster, every processor has its own memory. The communication between processors occurs through message passing (MPI). A decomposition of work (better computational domain) is essential for the parallelization on distributed-memory systems. Every processor contributes to the solving of the optimization problem by running a part of the program. A high number of cores (e.g. cluster of CPUs) could speed up the whole process significantly. But the parallelization increases also the programming burden, since the designer has to distribute the computational work among the available CPU processors and regulate the communication using MPI[5]. For a real life application, a large number of cores is essential and MPI is the most implemented paradigm on today HPC systems. A hybrid system, consisting of a cluster of shared-memory system combining the two systems, knows an increasing appliance today.

The appearance of programmable graphics processing units (GPU) enabled at relatively low price to access a new high computational power system. GPUs are indeed a shared-memory system but with a larger number of cores than CPU shared-memory systems. These GPU-cores are available in large numbers (up to 2400 cores[6]) and specialized on arithmetic computation, unlike the more powerful but general purpose CPU cores. The work of the design engineer is then to successfully divide the global optimization problem on small work packages that can be handled by a GPU core in a massively parallel manner. The problems that are easily divided on small and independent work packages are called *embarrassingly* parallel (e.g. simple image processing functions). If the work packages are not independent and need intercommunication, the problem is called *coarse-grained* parallel (e.g. low-order PDE solvers). If the communication demand increases, it is then called *fine-grained* parallel (e.g. High order PDE solvers).

The aim of this work is to introduce researchers active on design optimization to GPU computing while highlighting the GPU advantages and challenges. This review should first help researchers to evaluate in how

far their application can benefit from GPUs and at the same time allow them to recognize possible bottlenecks and how other researchers managed to solve them.

The remainder of this work focuses on the use of GPUs to accelerate optimization methods in topology optimization, shape optimization and multidisciplinary design optimization. The first section of this paper defines the term "modern GPU" as used in the title of this paper. The second section deals with the domain of application of design optimization covered in this work. The next section emphasizes the different optimization methods used in the literature and their parallelization potential. The third section discusses the advantages derived from the GPU use and highlights the areas requiring more research toward a better use of GPU high computational power.

II. Modern Graphics Processing Unit (GPU)

Today's GPUs evolved from graphic cards installed in most of computers starting from the eighties. The recent GPUs have a reduced heritage of old graphic cards but a historical point of view helps better defining the term "modern GPUs" as used in the title of this paper.

A graphic card is a complex electric circuit that processes graphical data starting with 3D raw information about scene content sent from the CPU to render display pictures with increasing quality, effects and refreshing frequency. The process is called a graphics pipeline and comprises in a simplified way a vertices shader and a pixel shader. A shader is a program part capable of processing many data at the same time (e.g. apply lighting and shadow). A high pressure on graphic cards for fast refreshing of pictures (mainly video games) caused the spectacular increase of computational power reflected by the large number of cores packed in graphic cards.

The high computational power attracted also scientists and engineers looking for low-cost high performance ways to speed up their applications. These first graphics cards were fixed-function devices and the user had to present his problem as a graphics problem to the card, which implicated a change on data storage and programming language. The term GPGPU, which stands for General Purpose Graphics Processing Unit, was established for this type of use of graphics cards. In response to this emerging demand, Nvidia released in 2007 the first fully programmable open graphics processing units in a C-based programming language called CUDA[6]. At the same time AMD released its programmable GPU with OpenCL[7], an open-source equivalent to CUDA. The term "modern GPUs" designates these fully programmable GPU with high-level programming language. Instead of the closed shaders as used initially, nowadays users write programs called kernels, which are run on GPUs.

Compared to a CPU core, a GPU core is less powerful. A GPU includes however a large number of these cores which combined address a higher computational power (see picture Computational Power). The second advantage is the specialization of GPU cores. While CPUs are inherently responsible for a wide spectrum of tasks requiring large cache and flow control, a GPU is mostly dedicated to fixed point arithmetic calculation. This is reflected in the high achieved computation performance measured in Float Operations per Second (FLOPs). From a HPC point of view, modern GPUs are a series of multiprocessor connected to a global memory. Every multiprocessor grouping a number of streaming processors has own shared-memory and read-only texture and constant cache. Recent GPUs have also a L1 and L2 caches. The bottleneck of any GPU is the latency of Global memory access and the small shared-memory.

The programming model CUDA is specialized for Nvidia GPU whereas OpenCL can be run on AMD and Nvidia GPUs. This portability has a price on the programming overhead and the peak performance gain[8]. If peak performances are required a GPU under CUDA is more promising. But if portability is more important OpenCL is better fitted. The CUDA programming model for example starts for every kernel a high number of threads (smallest computation units) grouped in blocks. The GPU manage the threads in groups of 32 called warp. Every warp executes the same instruction.

A coalesced access to data has to be followed in order to take advantage of the full bandwidth and special care has to be given to shared memory. If many threads access the same memory position simultaneously to write information (e.g. an incrementation) only a part will be saved and the rest is discarded. This non deterministic behavior is called a race condition and is very harmful for calculation, since a high amount of information can be lost and a random result may occur. Authors in design optimization, as in other disciplines, developed methods to deals with this problem and many other issues staying in the way for a high speedup of scientific application as it will follow in next section. Figure 1 depicts the key aspects and challenges for the use of GPUs.

The CUDA C programming language adds a set of extensions to the standard C programming language. Recently is also CUDA Fortran[9] available under commercial license. More information and examples are in the Cuda Programming Guide[6] the CUDA C Best Practice Guide[10] and other dedicated textbooks[11, 12]

OpenACC provides a similar approach to OpenMP for programming GPU in more generic way. However accessing high speedups on OpenACC is not trivial and the directive is open but the compiler for this directive is under commercial license, which reduces its impact on scientific community compared to CUDA or OpenCL.

Libraries also can assist developers to speed up there application. Many libraries that exist for CPU have their equivalent for GPU such as CUFFT[13], CUBLAS[14] and CUSPARSE[15].

GPU optimization methods are well referenced and many times reviewed (Nvidia) but every area has its own governing equations and specific algorithms. A look on used GPU optimization in a specific field is essential to learn the adaptation of optimization method on everyday problems and to notice challenges.

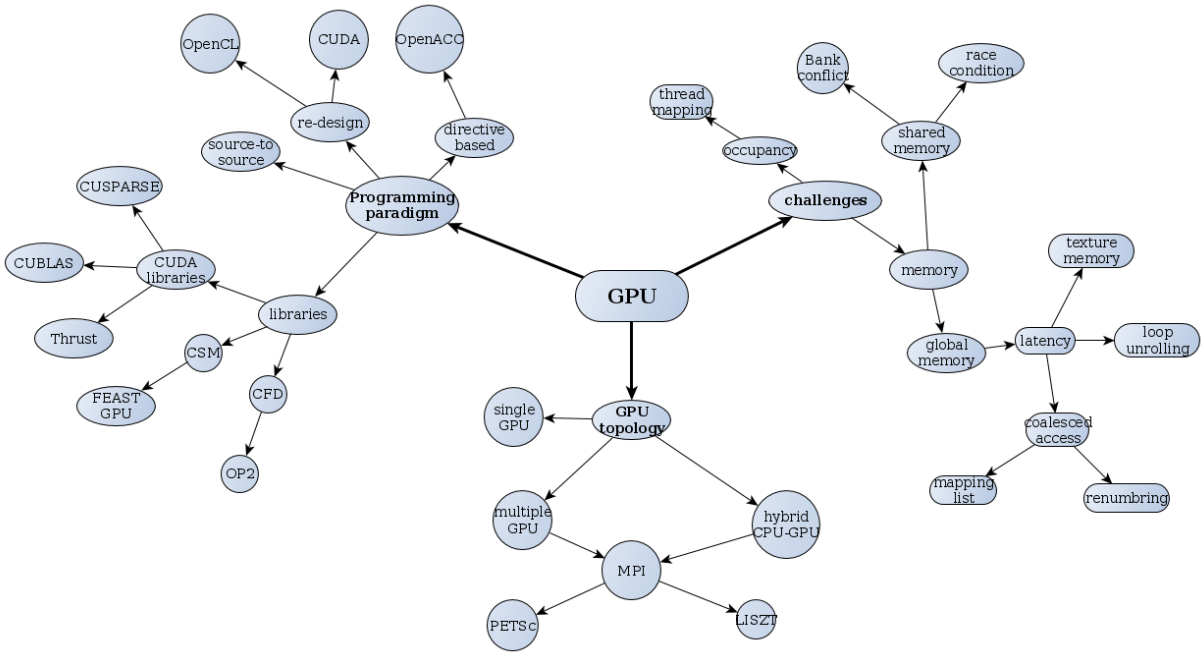

Figure 1. Chart of GPU key themes grouped in programming paradigm, GPU topology and challenges. *The programming could be performed through reediting of CPU code on CUDA or OpenCL, but also extended with special openACC directive or compiled with a source-to-source compiler to provide GPU code. The challenges of porting to GPU are related to global and shared memory efficient use.*

III. GPU in Design Optimization

Design optimization problems are generally classified as topology optimization, shape optimization or sizing optimization. The topology optimization is very important in the early design conception phase. It generates indeed a blue print of the general appearance of the design. The result of this process can be fine-tuned later with a shape optimization method. This section reviews GPU use in topology and shape optimization. It comments also on the absence of GPU in MDO.

A. Topology Optimization

Topology optimization is about evolving a start design toward an optimal one in regard of objective minimization. Many approaches have been developed to capture and guide the evolution of the topology during the optimization process (see fig. 3). Two of the main methods are level-set[16] and Solid Isotropic Microstructure with Penalization (SIMP)[17,18]. These methods have also been ported to GPU. The GPU potential of other methods such as the bubble method[19] and evolutionary structural optimization [20](ESO) are still to be explored. This section introduces briefly the two first methods and focuses on their adaptation to GPU highlighting the challenges and advantages.

1. Solid Isotropic Microstructure with Penalization method (SIMP)

The SIMP method is based in a homogenization approach. It describes structures as a combination of solid-void micro-elements. A pseudo-density variable (ρ) characterizes the material (solid ρ=1, void ρ =0) and is assumed to be constant within each element of the structure. The optimization process is about finding the optimal material distribution satisfying predefined constraints. This type of integer-programming is highly computational expensive and needs therefore to be avoided for large scale problems. A relaxation reduces the computational cost of the problem. Relaxation performs indeed an extension of possible values for the design variable from two values {0, 1} to the entire range [0, 1]. The problem becomes convex and can be consequently optimized with a gradient-based method. Through relaxation intermediate values for the pseudo density are indeed tolerated but physically not interpretable. A penalization solves this issue by promoting the

integer value 0 and 1 for the pseudo-density. One of the many penalization methods is *power law*[18] (ρ^{p}). The contribution of elements with intermediate densities is increasingly discarded with higher penalization factor *p*. The problem formulation with penalization is however not convex anymore. A remedy resides on applying a local filter on the density distribution averaging neighbor elements inside a predefined radius. An optimizer, such as optimality criteria[21] (OC) or Method of Moving Asymptotes[22] (MMA), is then responsible for the update of the design variables to locate the optimum solution.

The linear elasticity state equations are solved with Finite Element Methods (FEM). Finite Elements Analysis (FEA) is the central component of the optimization process. It generates the structure answer to loads (e.g. displacements), which is essential to the evaluation of the objective function.

The applications of the SIMP methods for large-scale problems with millions of design variables are computationally demanding and therefore require a high performance system. Work of Mahdavi et al. [23] is an example of parallelization of topology optimization application on CPU. GPUs as low-cost-high-performance alternative for HPC system have been also tested for solving topology optimization problems with SIMP method. The implementation from Schmidt et al. [24] of SIMP on structured meshes with a matrix-free conjugate gradient solver is faster on GPU than 48 cores shared memory CPU. The GPU implementation from Wadbro et al. [25] of SIMP method with Preconditioned Conjugate Gradient solver applied to a 2D plate with heat source yield a speedup of 20x against single CPU and 3x against multi-threaded CPU. Zegard et al. [26] implemented the SIMP approach for unstructured meshes in GPU focusing on assembly.

2. Level Set

The level set method, developed originally by Osher et al.[27] as a scheme to advance the motion of an interface, was applied later to topology optimization. In topology optimization, the level-set method optimizes a given structure by a sequence of guided motion of the structure boundaries. The guided evolution converges to an optimum solution by minimizing the objective function[16]. The flexible boundaries can represent complex shapes with the ability to create new topology through inserting new holes, or structure splitting and merging. Refer to the review of founder author[28] for a detailed insight and work of Allaire[29] for CPU applications.

The method is built on two fundamental equations: the boundary evolution and the state equations. The state equations are often discretized with FE methods. The boundary evolution is far simpler and can be solved with a finite difference method (FD). FD is a local method. It acts on a reduced group of elements of the mesh, which makes it suitable for efficient GPU processing [30]. A GPU interpretation of this method is in the work of Herrero et al.[31]. Challis et al.[32] solved an inverse homogenization problem with a GPU implementation of a level-set method targeting high resolution topology optimization. An increasing speedup with size is reported reaching 13x for 3D problem with over 4 million design variables[32].

3. Underlying FEM

Independently of the method (SIMP, level set) followed to formulate and solve the topology optimization problem, common steps exist: (1) An objective function is formulated reflecting the domain of application. (2) The design variable in this function is most of the time evaluated by solving a linear system A.x=f (3) an optimization scheme updates the design variable. The level set however advances also a boundary equation in time. The solver of the FEM discretized state equations is the most time consuming part of the optimization and should be the focus during the adaptation to GPU architecture. The rest of the procedures such as the optimizer (e.g. OC, MMA), boundary evolution (level set) is not time consuming so not directly relevant toward getting a good acceleration through GPU. Georgescu et al. provide a review on the use of GPU in all FEM steps from preprocessing to solving and post-processing. This section focuses on the assembly, the solver and the mesh importance.

Solver:

The system of equations discretized with FEM can be solved directly[34] (e.g. LU factorization) or iteratively (e.g. Conjugate gradients). Direct solvers are not appropriate for large scale problems since the full-size system matrix has to be stored. The memory usage scales with number of variable [25] following O ($N^{3/2}$). The large amount of inter-processors communication makes the direct solver challenging for parallelization [23]. Conjugate Gradient solvers on the other hand require less memory (O(N)) and are faster than direct methods which explain their large use in GPU [32,35]. The system matrix is symmetric and positive definite for CG consequently a preconditioned Conjugate Gradient solver (PCG) is more favorable[25]. Multigrid has been also implemented on GPU to accelerate the convergence of linear system solver [36, 37].

Cevahir et al.[38] accelerated CG sparse iterative solver for unstructured mesh in a multi-GPU cluster. Hypergraph partitioning[39] was used to reach a fair load balance and reduce the CPU-GPU communication targeting an implementation in 32 GPU of 16 nodes faster than 16 nodes of 16 CPU per node (16x16=256).

The system matrix is sparse and the sparse matrix vector multiplication (SpMV) is a key feature. A first attempt to take advantage of GPU regarding SpMV is to write kernels for GPU performing the multiplication.

This approach provides a large flexibility and can be adapted to specific needs of the problem to be solved in regard of data storage layout and assembly. Good performance requires however an important development effort [35, 36]. Geveler et al. based all the solver computation on one kernel for SpMV. This simplified the optimization. All effort was invested on this kernel and the entire solver could take advantage. This approach brings a programming overhead to turn all solver stages to SpMV functions. Efficient SpMV GPU implementations exist already: CUDA libraries (CUSPARSE) or Petsc. Wadbro et al. [25] used the CUBLAS library, the CUDA version of the linear algebra library BLAS, for inner products in the PCG. The active libraries OP2[41] and LISZT[42] provide a high-order abstraction for matrix vector multiplication.

The GPU global memory is reduced to a maximal of 6GB for 1 card and therefore an explicit building of global system matrix is highly limiting the size of problems that could be treated. FEM problems in topology optimization are inherently local, thus a matrix-free solver of state equation is feasible[24] and can be even one order of magnitude faster than a full-matrix solver [43].

Assembly:

Standard solvers build the system matrix computing node contributions and summing them to central elements for a global assembly. The solver is definitely the most time consuming part of a structure optimization method but the assembly part is also important. An adaptation of the assembly phase to GPU architecture saves computational time and improves the solver itself.

During the assembly, a problem occurs if nodes are contributing in a parallel manner to build element stiffness. Two or more nodes could add their values to the same elements at the same time, which causes a race condition and an information loss. Zegard et al. [26] used a graph coloring technique to avoid such a case coloring a set of non-racing nodes with the same color. Different colors cover the entire computational domain. All node of the same color can be run safely in parallel. Another approach is to assemble the stiffness matrix element-wise which never causes a race condition but results on nodes contribution calculated many times for different adjacent elements. Cecka et al. focus more on FEM assembly on GPU presenting low-level code optimization on CUDA targeting a speedup of 30x with single precision GPU code against double precision CPU version.

Mesh:

The mesh is crucial toward better use of GPU potential in structure analysis and optimization. An unstructured mesh provides certainly more flexibility to mesh complex domains surrounding complex geometries. It represents at the same time some challenges for the use of a GPU. The absence of ordered indexing and regular neighboring makes the memory access for node or cell data irregular and thus uncoalesced (see fig. 2).

For unstructured meshes, the cell-based approach presents some regularity compared to vertices-based meshes (e.g. number of cell neighbors). A vertex can have different number of related vertices depending on its place in the mesh. On the other hand a cell with *m* edges will have *m* neighbors within the computational domain and less in the boundaries (*m-1,m-2*).

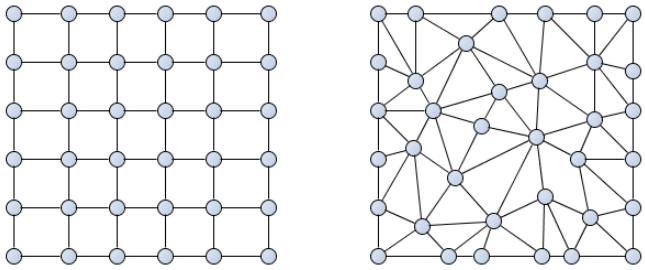

Figure 2. On the left a structured mesh with fixed number of neighbors for cells and vertices. On the right an unstructured mesh with same number of nodes. *Number of cell neighbors for unstructured mesh is almost the same for interior elements (m=3) but for interior vertices the number of neighbors varies from 5 to 7.*

The structured mesh is better fitted to a GPU, since the GPU threads can be mapped to the cell or nodes indices. Efficient unstructured mesh applications are more difficult for the GPU than the CPU since the GPU cache is reduced. The programmer should just avoid race conditions between threads. For the unstructured mesh and especially the vertex-based scheme a series of challenges are faced. Node renumbering and index list can help to keep a partly coalesced memory access.

The process of meshing has been often the responsibility of CPU. In this field a large set of mesh optimization are available on CPU. D'Amato et al. [45] used however GPU for meshing. The mesh influences the storage layout of data structure. Reguly et al. [43] analyze the different data structures such as ELLPACK and CSR and how they affect both assembly and solution. Goddeke et al.[46] propose a different renumbering than CSR reporting a speedup of 2x.

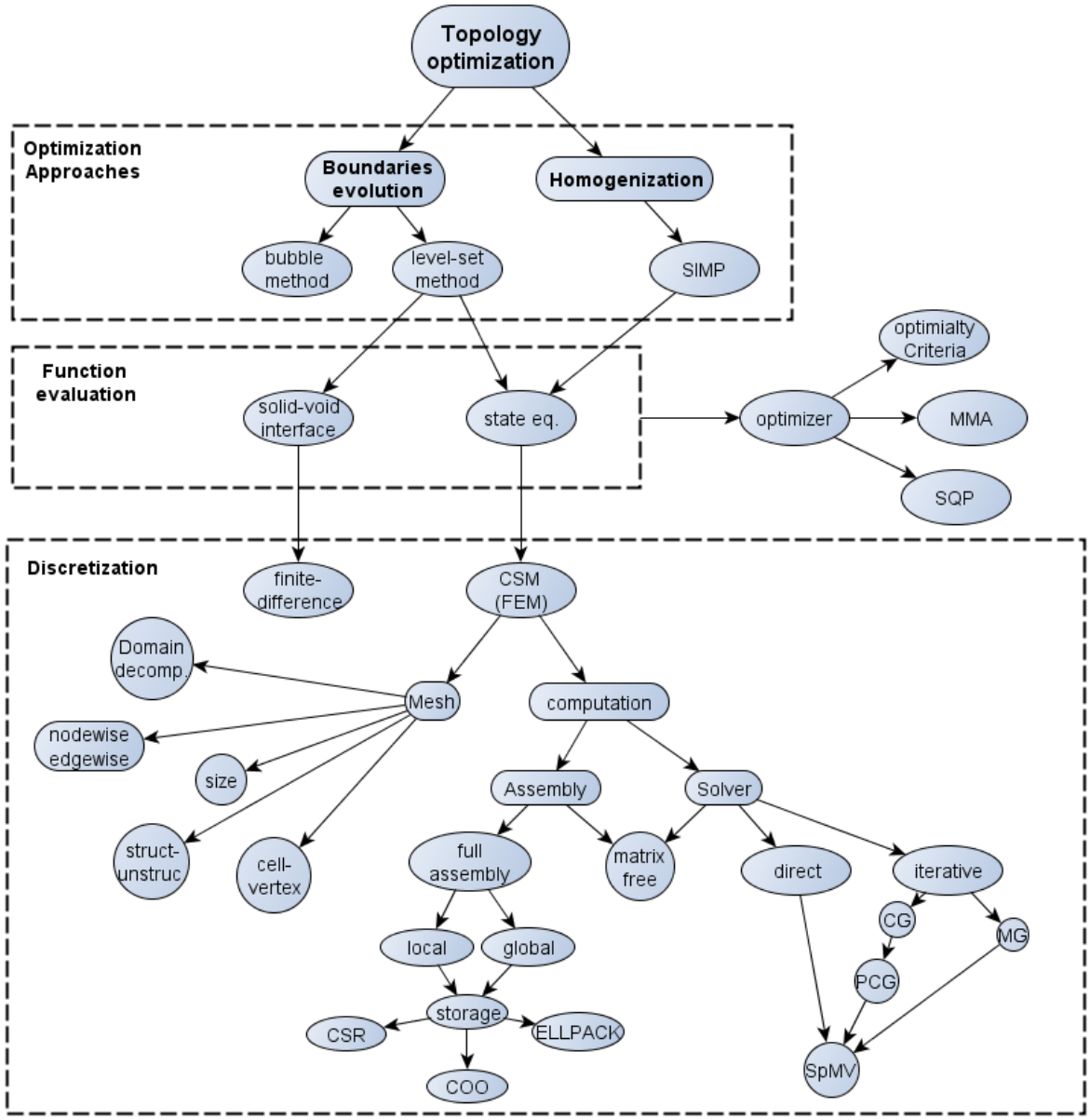

Figure 3. Chart of different topology optimization approaches and common techniques for function evaluation such as CSM computation with FEM.

B. Shape Optimization

The shape optimization is more restricted than topology optimization in the sense that the topology will remain untouched throughout the optimization process. The number of structure parts and the number of holes for example will be unchanged. Only the shape is changing to meet the problem specific constraints and objectives. The shape is parameterized through mathematical function (e.g. Bezier splines). The parameters controlling the geometrical construct, such as spline control points or curvature, build the design variables of the shape optimization problem. These design variables have often a clear geometrical interpretation which explains the strong coupling between mesh and design variables in shape optimization. A change in a spline curvature for example will affect all adjacent cells throughout the entire spline. This leads in the end to solving a non-linear system of equation $A(u).u = b(u)$ in order to evaluate the objective function. The system matrix and the forcing term can depend on the design variables. With the increase in complexity, compared to topology optimization, linear algebra libraries can no more be applied, unless a linearization is performed.

For many shape optimization problems, the flow is the driving factor through the optimization. For external flows over whole planes or wings, an aerodynamic shape optimization problem is solved to reduce the drag and increase the lift. For internal flows through engines or channels, an internal flow shape optimization problem is solved mainly to increase engine efficiency. Depending on the nature of the flow and the leading phenomena different equations have to be solved (see fig. 4).

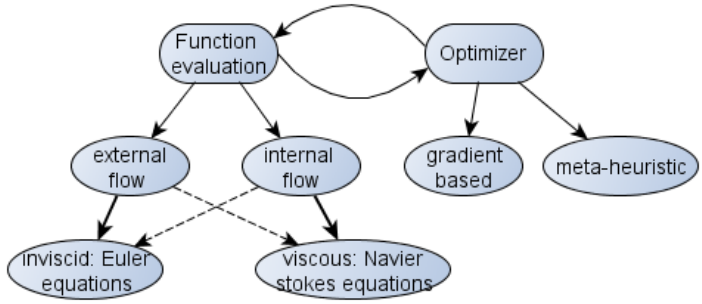

Figure 4. Aerodynamic shape optimization based on the function evaluation and the optimization method. *The function evaluation is mainly about solving Euler equation for external flows and Navier-Stokes equation for internal flows.*

Lefebre et al. [47] optimized a 2D/3D Euler solver for GPU. Brandvik et al.[48] implemented a source-to-source compiler to execute CPU Navier-Stokes solver on GPU with speedup of one order of magnitude in turbomachinery application. Asouti et al.[49] used a NS solver with an evolutionary algorithm optimizer for both external (airfoil) and internal applications (compressor cascade airfoil) achieving 27x speedup.

Shape optimization in GPU raises similar issue as topology optimization. Race conditions are avoided through cell or vertices coloring or through the cell-based calculation of fluxes.

C. Multidisciplinary Design Optimization (MDO)

MDO methods optimize designs with regards to multiple disciplines. Most spread method is a bi-disciplinary optimization making use of structure analysis and flow analysis[50]. No GPU work on this field was known to the author when preparing this review. Some initiatives exist of porting one of the disciplines (CFD or CSM) on GPU[51]. But generally when the complexity increases, developer try to decrease the scope of the development by working first on the MDO application for standard CPU. Exploring the GPU potential on established MDO CPU implementation is however promising in term of performance increase.

IV. GPU in Meta-heuristics

Metaheuristic methods can be classified as population-based and trajectory or single solution based. Population-based methods, such as Evolutionary algorithm; swarm intelligence and particle swarm, manage a set of interacting individuum (solution), that are improved in every optimization iteration. These methods focus more on the exploration of the solution space than in the exploitation of the neighborhood of one solution. Single solution-based such as advanced local search, simulated annealing, Tabu search, update continuously only one solution toward finding an optima. Some trajectory-based methods keep track of all intermediate solution (TABU) other just replaces old with new solution. These methods focus more on the refinement of a solution through exploitation of local neighborhood than a general global exploration of solution domain. Hybrid methods combining both features exist. They start with an exploration and then perform a refinement.

As seen in shape optimization some metaheuristic methods are applied for design optimization in GPU[49]. Many other meta-heuristic methods have been already used in shape and topology optimization but only implemented for CPU (Differential evolution[50]). The GPU implementations of many meta-heuristic methods in other area [52, 53] are encouraging to apply in design optimization. Ant colony optimization (ACO) is widely used[54] along with Genetic Algorithm[55], local search[56, 57] and Particle Swarm Optimization[58]. Taillard et al. explored the GPU potential for hybrid metaheuristic methods [59].

V. Discussion

CUDA enables developer to write code that is highly tuned for the used hardware. The CUDA developer community is increasing and useful tools help designing well performing codes: an integrated editor, debugger, profiler and memory checker. The prompt change in the GPU hardware scene from generation to generation requires however a continuous updating of high level programming skills. This effort can distract from discipline specific work. Whereas the most important issue is that a set of optimization techniques resulting from tremendous work can be made insignificant with a next hardware generation. The example of computing precision is well representative. First GPUs of 2007-2009 did not support double precision calculation. An important effort has been invested to mix GPU single precision and CPU double precision to achieve fast results without accuracy loss. This work is no more of interest, since recent GPUs support double precision. Some changes in GPU memory layouts, such as a larger shared memory or a larger L1 cache, can require a code to be retuned to keep best performance. CUDA itself was a relief from the graphics programming burden of early GPUs. A further abstraction seems unavoidable. The hardware specific optimization should be separated from the algorithm itself. This hardware oriented code tuning should be a responsibility of low-level system, while the user focuses on the high-level algorithm. One solution is to use directive-based tools such as openACC. The

high level abstraction is implemented in the openACC standard. The specific compiler of this standard is under commercial license. Another alternative is to apply toolbox libraries such as CUSARSE and Thrust, and also vendor independent libraries[60]. A trade-off between peak performance hardware specific tuned code and a portable easy maintained code is inevitable. For large developer teams a computer science expert can focus only on the optimization of the implementation of algorithm developed by the rest of the team. But a high level of abstraction should not make from a GPU a black box for users. Learning the used hardware specifications helps always to design adapted algorithms and to recognize algorithms with high parallelism potential.

VI. Conclusion

This paper covered the use of GPU in the acceleration of design optimization problems focusing on topology optimization, shape optimization and multidisciplinary design optimization. Interesting speedup of 1 to 2 orders of magnitude have been reported in the literature for topology optimization and aerodynamics shape optimization problems. The core of the optimization process is often the simulation (CFD or CSM). Porting an optimization application is then more about porting the FEM solver or the CFD solver. For this purpose established libraries are available such as CUSPARSE and OP2 to prevent developer from getting distracted by purely technical aspects of programming.

Concerning multidisciplinary design optimization a rarer use of GPUs is observed, which is basically influenced by the higher complexity of MDO problems and the multitude of tools and simulation involved in comparison with single-discipline design optimization.

The number of metaheuristic optimizer running on GPU increased in many disciplines but not in design optimization. Especially population-based methods are inherently adapted to GPU, since the same procedure is repeated for independent individuum. This scheme maps very well to the thread structure of GPUs.

Some methods are easily ported others are very hard to port and need a lot of redesigning. First interest of this paper was to recognize in design optimization the parts ready for GPU porting and the others that require more adapting effort. This makes possible the evaluation of the need for GPU. The next interest is to encourage developer to work in heterogeneous system. Tasks are distributed among CPUs and GPUs taking advantage of both systems. Cooperation between systems is wished rather than competition.

Acknowledgments

This research activity is funded by a Marie Curie Action as part of the European Union's Framework 7 research program (AMEDEO: Project No. 316394).

References

[1]Talbi, EG., *Metaheuristics: from design to implementation* , John Wiley & Sons, 2009

[2]Dominique T., Gábor J. (ed.), *Optimization and Computational Fluid Dynamics*, Springer Verlag, AIAA, Berlin, 2008.

[3]OpenMP Architecture Review Board, "OpenMP Application Program Interface" URL: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf [cited 27 June 2014]

[4]Jeffers, J., Reinders, J. *Intel Xeon Phi coprocessor high performance programming*, Newnes, 2013

[5]Message P Forum. "Mpi: a Message-Passing Interface Standard," Technical Report. University of Tennessee, Knoxville, TN, USA.

[6] Nvidia, "CUDA C Programming Guide" http://docs.nvidia.com/cuda/cuda-c-programming-guide/ [cited 27 June 2014]

[7]Khronos OpenCL Working Group, "The opencl specification" URL: https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf [cited 27 June 2014]

[8]Fang, J., Varbanescu, A. L., Sips, H. "A comprehensive performance comparison of CUDA and OpenCL", *In Parallel Processing (ICPP), 2011 International Conference on*, 2011, pp. 216,225, IEEE.

[9]The Portland group, "CUDA Fortran Programming Guide and Reference"

[10]Nvidia, "CUDA C Best Practices Guide" http://docs.nvidia.com/cuda/cuda-c-programming-guide/ [cited 27 June 2014]

[11]Sanders, J., Kandrot, E., *CUDA by example: an introduction to general-purpose GPU programming*, Addison-Wesley Professional, 2010.

[12]Kirk, D. B., Wen-mei, W. H., *Programming massively parallel processors: a hands-on approach*, Newnes, 2012.

[13]Nvidia, "cuFFT" URL: http://docs.nvidia.com/cuda/cufft/ [cited 27 June 2014]

[14]Nvidia, "cuBLAS" URL: http://docs.nvidia.com/cuda/cublas/ [cited 27 June 2014]

[15]Nvidia, "cuSPARSE" URL: http://docs.nvidia.com/cuda/cusparse/ [cited 27 June 2014]

[16]Allaire, G. "Topology optimization with the homogenization and the level-set methods," *Nonlinear homogenization and its applications to composites, polycrystals and smart materials*. Vol. 170, 2005, pp. 1,13.

[17]Bendsøe, M. P., Kikuchi, N. "Generating optimal topologies in structural design using a homogenization method", *Computer methods in applied mechanics and engineering*, Vol. 71, No. 2, 1988, pp. 197,224.

[18]Bendsoe M. P., Sigmund O., *Topology Optimization: Theory, Methods and Applications*, Springer, Berlin 2003

[19]Eschenauer, H. A., Kobelev, V. V., Schumacher, A. "Bubble method for topology and shape optimization of structures", *Structural optimization*, Vol. 8, No. 1,1994, pp. 42,51.

[20]Xie, Y. M., Steven, G. P. "A simple evolutionary procedure for structural optimization", *Computers & structures*,Vol. 49 No. 5, 1993, pp. 885,896.

[21]Yin, L., Yang, W., "Optimality criteria method for topology optimization under multiple constraints", *Computers & Structures*, Vol. 79, No. 20, 2001, pp. 1839, 1850.

[22]Svanberg, K. "The method of moving asymptotes - a new method for structural optimization", *International journal for numerical methods in engineering*, Vol. 24, No. 2, 1987, pp. 359, 373.

[23]Mahdavi, A., Balaji, R., Frecker, M., Mockensturm, E. M., "Topology optimization of 2D continua for minimum compliance using parallel computing", *Structural and Multidisciplinary Optimization*, Vol. 32 No. 2, 2006, pp. 121, 132.

[24]Schmidt, S., Schulz, V., "A 2589 line topology optimization code written for the graphics card", *Computing and Visualization in Science*, Vol. 14, No. 6, 2011, pp. 249, 256.

[25]Wadbro, E., Berggren, M., "Megapixel topology optimization on a graphics processing unit", *SIAM review*, Vol. 51, No. 4, 2009, pp. 707,721.

[26]Zegard, T., Paulino, G. H. "Toward GPU accelerated topology optimization on unstructured meshes", *Structural and Multidisciplinary Optimization*,Vol. 48, No. 3, 2013, pp. 473,485.

[27]Osher, S., Sethian, J.A. , "Front propagating with curvature dependent speed: algorithms based on Hamilton-Jacobi formulations," *Journal of Computational Physics*, Vol. 78, No. 1, 1988, pp. 12, 49.

[28]Osher, Stanley, and Ronald P. Fedkiw. "Level set methods: an overview and some recent results." *Journal of Computational physics*, Vol. 169, No. 2, 2001, pp. 463, 502.

[29]Allaire, G., "Topology optimization with the homogenization and the level-set methods", *In Nonlinear homogenization and its applications to composites, polycrystals and smart materials*, 2005, pp. 1,13, Springer Netherlands.

[30]Micikevicius, P. (2009, March). "3D finite difference computation on GPUs using CUDA", *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, March 2009, pp. 79,84, ACM.

[31]Herrero,D, Martinez, J. , Marti, P., "An implementation of level set based topology optimization using GPU", *10th World Congress on Structural and Multidisciplinary Optimization*, Orlando, Florida, USA, May 19 - 24, 2013

[32]Challis, V. J., Roberts, A. P., Grotowski, J. F. "High resolution topology optimization using graphics processing units (GPUs)", *Structural and Multidisciplinary Optimization*, Vol. 49, No. 2, 2014, pp. 315,325.

[33]Georgescu, S., Chow, P., Okuda, H. "GPU Acceleration for FEM-Based Structural Analysis", *Archives of Computational Methods in Engineering*, Vol. 20, No. 2, 2013, pp. 111,121.

[34] Davis, T. A., *Direct methods for sparse linear systems (Vol. 2)*. Siam.,2006

[35]Bolz J., Farmer I., Grinspun I., and Schröoder P., "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," *ACM Transactions on Graphics*, New York, NY, 2003, pp. 917-924

[36] Ribbrock, D., Göddeke, D., Zajac, P., Turek, S. "Efficient finite element geometric multigrid solvers for unstructured grids on GPUs". Techn. Univ., Fak. für Mathematik.,2011

[37]Geveler, M., Ribbrock, D., Göddeke, D., Zajac, P., Turek, S., "Towards a complete FEM-based simulation toolkit on GPUs: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses", *Computers & Fluids*, Vol. 80, 2013, pp. 327,332.

[38]Cevahir, A., Nukada, A., Matsuoka, S. "High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning", *Computer Science-Research and Development*, Vol. 25No. 1, 2, 2010, pp. 83,91.

[39]Catalyurek, U. V., Aykanat, C. , "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication", *Parallel and Distributed Systems, IEEE Transactions on*, Vol. 10, No. 7, 1999, pp. 673, 693.

[40]Bell N., Garland M., "Efficient sparse matrix-vector multiplication on CUDA," Nvidia Technical Report NVR-2008-004, Nvidia Corporation 2 (5),2008.

[41]Mudalige, G. R., Giles, M. B., Bertolli, C., Kelly, P. H. "Predictive modeling and analysis of OP2 on distributed memory GPU clusters." *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40, No. 2, 2012, pp. 61,67.

[42]DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Hanrahan, P. "Liszt: a domain specific language for building portable mesh-based PDE solvers", *In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, November, pp. 9, ACM.

[43]Reguly, I. Z. , Giles, M. B., "Finite element algorithms and data structures on graphical processing units," *International Journal of Parallel Programming*, Dec., 2013, pp. 1, 37.

[44]Cecka, C., Lew, A. J., Darve, E., "Assembly of finite element methods on graphics processors", *International journal for numerical methods in engineering*, Vol. 85, No. 5, 2011, pp. 640,669.

[45]D'Amato, J. P., Vénere, M., "A CPU–GPU framework for optimizing the quality of large meshes", *Journal of Parallel and Distributed Computing*, Vol. 73, No. 8, 2013, pp. 1127,1134.

[46]Goddeke, D., Wobker, H., Strzodka, R., Mohd-Yusof, J., McCormick, P., Turek, S. "Co-processor acceleration of an unmodified parallel solid mechanics code with FEASTGPU", *International Journal of Computational Science and Engineering*, Vol. 4, No. 4, 2009, pp. 254,269.

[47]Lefebre, M., Guillen, P. Le Gouez, J.-M., Basdevant, C., "Optimizing 2D and 3D structured Euler CFD solvers on Graphical Processing Units," *Computers & Fluids*, Vol 70, 2012, pp. 136, 147.

[48]Brandvik, T., Pullan, G. "An accelerated 3D Navier–Stokes solver for flows in turbomachines", *Journal of Turbomachinery*, Vol. 133, No. 2, 2011, 021025.

[49] Asouti, V., Kontoleontos, E., Trompoukis, X., Giannakoglou, K., "Shape optimization using the one–shot adjoint technique on Graphics Processing Units", *In 7th GRACM International Congress on Computational Mechanics Conference*, Vol. 30, Athens, Greece.

[50]Verstraete, T. , "Cado: a computer aided design and optimization tool for turbomachinery applications", *In 2nd Int. Conf. on Engineering Optimization*, 2010, Lisbon, Portugal, September, pp. 6,9.

[51]Aissa, M. H., "Efficient high performance computing techniques for multi-disciplinary optimization" 5th Symposium of VKI Phd research, Sint-Genesius-Rode, Belgium, March, 2014, (to be published)

[52]Krömer, P., Platoš, J., Snášel, V. , "Nature-Inspired Meta-Heuristics on Modern GPUs: State of the Art and Brief Survey of Selected Algorithms", *International Journal of Parallel Programming*, Vol. 42, No. 5, 2014, pp. 681, 709.

[53]Talbi, EG., *Metaheuristics on Graphics Processing Units*, John Wiley & Sons, 2014.
[54]Cecilia, J. M., García, J. M., Nisbet, A., Amos, M., Ujaldón, M., "Enhancing data parallelism for ant colony optimization on GPUs", *Journal of Parallel and Distributed Computing*, Vol. 73, No. 1, 2013, pp. 42,51.
[55]Langdon, W. B., "Graphics processing units and genetic programming: an overview", *Soft Computing*, Vol. 15, No. 8, 2011, pp. 1657,1669.
[56]Van Luong, T., Melab, N., Talbi, E. G. "GPU computing for parallel local search metaheuristic algorithms", *Computers, IEEE Transactions on*, Vol. 62, No. 1,2013 , pp. 173,185.
[57]Czapiński, M., "An effective parallel multistart tabu search for quadratic assignment problem on CUDA platform", *Journal of Parallel and Distributed Computing*, Vol. 73 No. 11, 2013, pp. 1461,1468.
[58]Mussi, L., Daolio, F., Cagnoni, S. "Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture", *Information Sciences*, Vol. 181, No. 20, 2011, pp. 4642,4657.
[59]Taillard, E., Melab, N., Talbi, E. G., "Parallelization strategies for hybrid metaheuristics using a single GPU and multi-core resources", *In Parallel Problem Solving from Nature-PPSN XII*, 2013, pp. 368,377, Springer Berlin Heidelberg.
[60]Markall, G. R., Ham, D. A., Kelly, P. H. "Towards generating optimised finite element solvers for GPUs from high-level specifications", *Procedia Computer Science*, 2010,May, pp. 1815,1823.