Computers and Structures 157 (2015) 9–18

Contents lists available at ScienceDirect

Computers and Structures

journal homepage: www.elsevier.com/locate/compstruc

Fine-grained GPU implementation of assembly-free iterative solver for finite element problems

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

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

ARTICLE INFO

Article history:
Received 17 December 2014
Accepted 5 May 2015
Available online 27 May 2015

Keywords:
GPU computing
High performance computing
Matrix-free methods
Finite element method

ABSTRACT

This paper proposes a fine-grained implementation of matrix-free Conjugate Gradient (CG) solver for Finite Element Analysis (FEA) using Graphics Processing Unit (GPU) architectures. The use of GPU computing in FEA is today an active research field. This is primary due to current GPU sparse solvers are partially parallelizable and can hardly make use of Data-Level Parallelism (DLP) for which GPU architectures are designed. The proposed GPU instance takes advantage of Massively Parallel Processing (MPP) architectures performing well-balanced parallel calculations at the Degree-of-Freedom (DoF) level of finite elements. The numerical experiments evaluate and analyze the performance of diverse GPU instances of the matrix-free CG solver.

1. Introduction

Nowadays, parallel processing has become the dominant paradigm for High-Performance Computing (HPC). This is primarily due to the decrease of the progress of single-core processor performance and the diminishing marginal returns in Instruction-Level Parallelism (ILP) [1]. The former issue, known as *power wall*, is related with the end of the frequency scaling factor as a result of the physical constraints preventing excessive power dissipation at GHz clock rates. Such a wall was reached in the early twenty-first century, stopping the trend of exponential frequency growth at just below 4 GHz [2]. The latter issue, known as *ILP wall*, is attributed to the disproportionate resources, power dissipation costs and required complexity to obtain some benefits. In addition to these issues, the *memory wall* [3], where the processor speed improvement exceeds the memory speed improvement, was also reached by the end of the twentieth century. These facts allow us to affirm that serial computing has today reached its zenith in performance [4] and future performance increases must largely come from increasing the number of processors (multi-core architectures) rather than making faster cores [5].

By increasing the number of processors, Thread-Level Parallelism (TLP) and Data-Level Parallelism (DLP) can be exploited. The TLP, also known as function parallelism, aims to divide large problems into smaller ones, whose code is then distributed across multiple processors and solved in parallel. On the other hand, the DLP, also known as Massively Parallel Processing (MPP), focuses on distributing the data across different parallel computing nodes, which perform the same task on different pieces of distributed data. The TLP-based techniques permit flexible parallel computing, while higher speedups are possible with DLP-based approaches. For these reasons, task and data parallelisms are often combined [6] to achieve performance increments.

Graphics Processing Unit (GPU) architectures are specially designed to exploit and deal with DLP. These architectures have emerged as a competitive platform for non-graphics HPC applications [7], the so-called General-Purpose computing on Graphics Processing Units (GPGPUs) [8]. GPU computing has been successfully applied in a wide spectrum of scientific, engineering and enterprise applications [9–11]. This has been facilitated by the tremendous advances made in the programming languages and tools for GPU developments. Even though these languages and tools make the developments using GPU considerably easier, the programming strategy is a must and the problem formulation is crucial to obtain acceptable results.

Despite the exponential increase in the computational resources provided by the Moore's law [12] last years, the computation cost, memory requirements and time constraints of Finite Element Analysis (FEA) are still long-standing and cutting edge challenges due to the ever-increasing complexity of finite element models [13]. The Finite Element Method (FEM) [14] involves two computationally intensive tasks: the assembly of the local element equations into a global system of equations and the system resolution. For large scale finite element models these tasks can lead to an unaffordable problem. The resolution of large system of equations is normally addressed using iterative solvers, which require

* Corresponding author.
E-mail addresses: jesus.martinez@upct.es (J. Martínez-Frutos), pedro.castejon@upct.es (P.J. Martínez-Castejón), david.herrero@upct.es (D. Herrero-Pérez).

http://dx.doi.org/10.1016/j.compstruc.2015.05.010
0045-7949/© 2015 Elsevier Ltd. All rights reserved.

of less memory than direct methods at the cost of increasing the computational cost. Such a computational time is often reduced using distributed computing [15,16]. The unaffordable assembly processes due to insufficient memory have been traditionally addressed using assembly-free methods, which were developed for low memory computers in the early eighties [17]. These methods obviate the assembly step, saving the required memory, at the cost of increasing the computational cost meaningfully. A key feature of matrix-free methods is that they are parallelizable.

The weak scalability of GPU sparse calculations has constrained the acceleration of FEA using GPU architectures [18,19]. Significant speedups have been achieved in the creation and assembly of the global stiffness matrix [20]. However, the time spent in this stage is much shorter than the time spent in the solving stage [21]. The GPU instances of iterative solvers using sparse-matrix representation aim to speedup matrix–vector and inner-vector product operations [22–27] considering limited transfer rates over data channels, concurrency and memory coalescing problems. GPU computing has also been successfully applied to solve time dependent problems, such as elastodynamic simulations [28], where concurrent read/write (r/w) access to nodal information is of paramount importance.

The use of matrix-free methods on GPU architectures has already shown its advantages for FEA problems using the Element-by-Element (EbE) FEM technique [29]. Instead of assembling the global system matrix, the EbE FEM solution scheme makes the entire decomposition of some iterative solution scheme into element-level computations [30]. The refinement of GPU computation using this technique, performing calculations at node-level, has also shown its advantages [31]. The EbE solution scheme has also been applied to problems in which the computational bottleneck comes from the regeneration of the finite element model, such as those appearing in structural topology optimization [32,33]. Some disadvantages of the EbE scheme are shown in [33], where poor or null speedups are achieved. The reasons of these results are deeply studied in this work to determine the prevalent factors of GPU implementation for matrix-free solvers.

This work proposes a fine-grained GPU implementation of matrix-free Conjugate Gradient (CG) solver for EbE FEM technique. The matrix–vector operations of the iterative solver are performed at Degree of Freedom (DoF) level of finite elements. The underlying idea is that the slowest task determines the speed of the whole calculation. For this reason, the workload is reduced and well-balanced for all the threads of the MPP architecture. Besides, the concurrent memory access is studied both using graph labeling methods and reorganizing the concurrent accesses. The work aims to provide some guiding principles about the best way to exploit the parallelization capabilities of GPUs for implementing iterative solvers. The proposed GPU instance of EbE FEM technique is compared with other possible implementations to evaluate the effects of diverse factors, including granularity, concurrent memory access and synchronization overheads. This comparison aims to shed some light on the complex effects observed when implementing matrix-free iterative solvers on GPU architectures, which are difficult to understand because these factors have often opposite effects.

The paper is structured as follows. Section 2 presents the massive parallel GPU architecture and the parallel development environment. Section 3 is devoted to the study and discussion of granularity and concurrency in the implementation of EbE FEM technique on GPU architectures. The numerical experiments to evaluate diverse GPU instances are presented in Section 4. Finally, Section 5 presents some conclusions about the steps to follow for the successful GPU implementation of assembly-free CG solver for FEM.

2. GPU and CUDA architecture

GPU devices offer incredible computational resources for both graphics and non-graphics processing. This massively parallel architecture was initially designed to satisfy the market demand of real-time and realistic 3D visualization. The use of Nvidia devices and its programming model, Compute Unified Device Architecture (CUDA), is the prevailing tendency. Nevertheless, the low-level API Open Computing Language (OpenCL) for heterogeneous computing, available for different GPU manufacturers, permits us to launch parallel code using a limited subset of the C programming language. On the other hand, CUDA provides a comprehensive development environment for building GPU-accelerated applications using high-level C/C++ programming language.

The CUDA environment allows us to view the GPU as a compute device able to run a lot of threads using Single Instruction Multiple Data (SIMD) architecture, which typically exploits DLP. The parallel code (single instruction) is defined as a C Language Extension function, called kernel, which is executed by a lot of CUDA threads using different data (multiple data). The kernel call should specify the number of CUDA threads organized as a grid of thread blocks. These threads have only access to the device dynamic random-access memory (DRAM) and on-chip memory through the memory spaces depicted in Fig. 1. The blocks are batch of threads able to cooperate by sharing data through shared memory and to synchronize their execution coordinating memory access. The threads also have access to a fast private memory and to a rather slow memory. The latter is composed of constant and texture memory, which permit read-only memory access, and global memory, which allows r/w memory access. Fig. 1 shows how kernels are invoked from the host (CPU) to the device (GPU) organized as a batch of threads grouped as a grid of thread blocks.

The software developments using CUDA consist of the following steps: (i) memory allocation and transaction, (ii) kernel execution on the GPU and (iii) copy back the results to the host. The strategies to optimize GPU computing can be summarized as follows: (i) optimization of parallel execution to achieve maximum use of cores, (ii) optimization of memory access to avoid concurrency, (iii) optimization of instruction usage to achieve maximum instruction performance and (iv) optimization of communications to achieve minimal synchronization between parallel executions. The improvement and degradation of performance of GPU instances can be justified by some of these optimization criteria.

3. Computational implementation

Let consider the linear system of equations resulting from the finite element discretization of the linear elasticity system as follows

$$\mathbf{Ku} = \mathbf{f}, \tag{1}$$

where $\mathbf{K}$ is the symmetric positive-definite global stiffness matrix, $\mathbf{u}$ is the vector of unknown displacements, and $\mathbf{f}$ is the vector of nodal forces.

For large-scale systems of equations, the storage and manipulation of matrix $\mathbf{K}$ require of a lot of memory and computer time even with the use of sparse linear algebra. The memory limitation makes iterative solver a prevailing approach for large-scale simulations where direct methods would be prohibitively expensive even with parallel computing. Iterative solvers are flexible, easy to implement and convergence in a finite number of steps is theoretically demonstrated. Their implementation requires of matrix–vector products, which can make the problem computationally intensive for large system of equations.