Scaling Results for a Discontinuous Galerkin Finite-Element Wave Solver on Multi-GPU Systems

Max Rietmann, Olaf Schenk, Helmar Burkhart
{ max.riethmann | olaf.schenk | helmar.burkhart } @unibas.ch

University of Basel
Department of Mathematics and Computer Science
Klingelbergstrasse 50
CH-4056 Basel, Switzerland
http://www.cs.unibas.ch

Abstract. Among the various techniques for solving hyperbolic partial differential equations with inhomogeneous, irregularly-shaped domains, a relatively new type of finite element method has grown in popularity because of its flexibility and scalability across many parallel cores. Discontinuous Galerkin (DG) methods have shown themselves to be an effective scheme for the simulation of wave-propagation problems commonly arising in computational seismology. This paper presents performance results of a DG solver using one and more CUDA-based graphics processors (GPUs) and challenges associated with scaling the simulation across multiple GPUs.

Key words: Discontinuous Galerkin, Computational Seismology, CUDA, GPU

1 Introduction

Data locality is the key to any efficient simulation of wave phenomena on highly parallel architectures. Spectral finite elements lead to diagonal mass matrices, they are inherently parallel when combined with explicit time integration schemes. However, spectral elements are typically restricted to conforming hexahedral meshes, which are difficult and time-consuming to generate, in particular in the presence of complex topography or local geophysical features. Discretization based on discontinuous Galerkin (DG) finite elements accommodate tetrahedral meshes and even hanging nodes, crucial for accurately handling topography and using local refinement. In addition, they also lead to block-diagonal mass matrices and thus are essentially as efficient as spectral elements on massively parallel architectures.

In addition, there is great interest in using graphics processors in the simulation of wave phenomena because of their ability to deliver very high performance. As more supercomputing centers upgrade machines with GPUs such as the Cray XK6, developing codes that are able to utilize these multiple GPUs is very important. This paper examines scaling and performance results for the discontinuous Galerkin solver MIDG [1], which was adapted to work for the acoustic wave equation. MIDG provides both CPU and CUDA-based GPU solvers for hyperbolic partial differential equations. It uses MPI for parallelization and Parmetis [2] for mesh partitioning.

2 Discontinuous Galerkin

Discontinuous Galerkin is a type of finite-element method that has grown in popularity for solving wave-propagation problems [1]. It allows for irregular domains with inhomogeneous material constants and is well suited to simulate realistic wave phenomena. Further than this, it is easily parallelized through domain decomposition and can be structured to overlap computation with MPI-communication, which enhances parallel scalability.

The computational seismological problems that motivate this research are based on the elastic wave equation, but for simplicity in this paper we will explore the acoustic wave

equation, which has fewer terms and provides a simpler scalabilty testbed. After splitting into a system of first-order equations, we get

$$\frac{\partial v}{\partial t} + \nabla \cdot (c^2(x)\,\mathbf{w}) = f \tag{1a}$$

$$\nabla v + \frac{\partial \mathbf{w}}{\partial t} = 0, \tag{1b}$$

where $c(x)$ is the material-determined wave speed, f is an external forcing, v is a scalar field, and $\mathbf{w}$ is a vector field with 1,2, or 3 elements corresponding to the x, y, and z dimensions of the problem domain. The simulations themselves are based on unstructured grids which are a triangular decomposition of the desired simulation domain into K elements. For simplicity, this preliminary work will focus on two dimensions, but the result can be transfered directly to three dimensions. Following the standard Galerkin finite element treatment, we multiply our equations by test functions ϕ and ψ, integrate over the domain, do a higher-dimensional integration by parts, and a discretization yielding the so-called weak formulation for each element $k \in K$ [1]:

$$M^k \frac{d\mathbf{v}^k}{dt} - c^2 \sum_{\xi} S^k_{\xi} \mathbf{w}^k_{\xi} + M^{\partial K} \sum_{\xi} ((c^2 \mathbf{w}^k_{\xi})^* n_{\xi} = F^k \tag{2a}$$

$$M^k \frac{d\mathbf{w}^k_{\xi}}{dt} - S^k_{\xi} \mathbf{v}^k + M^{\partial K} (\mathbf{v}^k)^* n_{\xi} = 0 \quad \text{for } \xi = \{x, y, z\}, \tag{2b}$$

where M^k and S^k are element-local matrices defined as

$$S^k_{\xi}(j,i) = \int_{\Omega_k} \frac{\partial \phi^h_j}{\partial d\xi} \phi^h_i d\Omega_k, \quad M^k(i,j) = \int_{\Omega_k} \phi_i \phi_j d\Omega_k \quad \text{and,}$$

and $\mathbf{v}^k$ and $\mathbf{w}^k_{\xi}$ are vectors representing the nodes within an element. A unique property of DG, which separates it from a traditional FEM, is that the nodes on the element boundary are unique despite sharing a location (x, y, z) with the nodes in the neighboring element boundary. These doubly-defined nodes are coupled via the numerical flux $(c^2 w_{\xi})^*$, but their independence mitigates the need for any sort of atomic update or mesh-coloring, which is usually required in the assembly stage of a FEM solver when threading is enabled because element-boundary nodes are shared.

The DG-algorithm consists of a local element update, followed by the calculation of numerical flux, which requires information from neighboring elements. When only a single CPU thread or GPU is being used for calculation, synchronization is only required across GPU threads, as the CPU simply iterates over all elements linearly. Parallelization beyond a single CPU-thread or single GPU is achieved through domain decomposition, and, in the case of MIDG, using MPI and Parmetis [4]. Each sub-domain is then composed of two sub-domains $K = K_{\text{outer}} \cup K_{\text{inner}}$, where K_{outer} is made of elements that share a boundary with elements of another sub-domain.

The CPU or GPU code first updates elements $k \in K_{\text{outer}}$, which is followed by asynchronous MPI-communication. The code completes the update on elements $k \in K_{\text{inner}}$, which should contain more elements than K_{outer} so that the MPI communications have time to finish before the computation completes. Once the inner domain and communication is finished, the numerical flux calculation for the nodes along the partition boundary can be computed.

Within the each sub-domain, the CPU code is structured such that it simply loops over elements k, then loops over nodes within each element. The GPU code is structured quite differently. Each element is assigned to a CUDA block, and each thread is assigned a node [3].

2.1 Numerical Results

The scaling tests were carried out at the Swiss National Supercomputing Centre (CSCS) on the Eiger GPU cluster. The specific machines used were dual socket AMD Opteron Magnycours systems with 24-cores and two high-end NVIDIA C2050/70 Fermi GPU cards per node. The code, which can be run using either CPUs or GPUs, was scaled across multiple nodes using two mesh sizes with the number of elements $K = \{20'664, 46'629\}$. The number of degrees of freedom (DOF) this presents is then simply $K \times \frac{(N+1)(N+2)}{2}$, where N is the order of the polynomial used as a basis function for the scheme. Note that in a standard finite-element method, the element boundaries are shared between elements, where in DG the element boundary is unique to each element. This makes the total number of DOF for a standard FEM less than with DG (a possible disadvantage of DG). These simulations had $N = 4$ or 15 nodes per element and $15 \times K$ DOF.

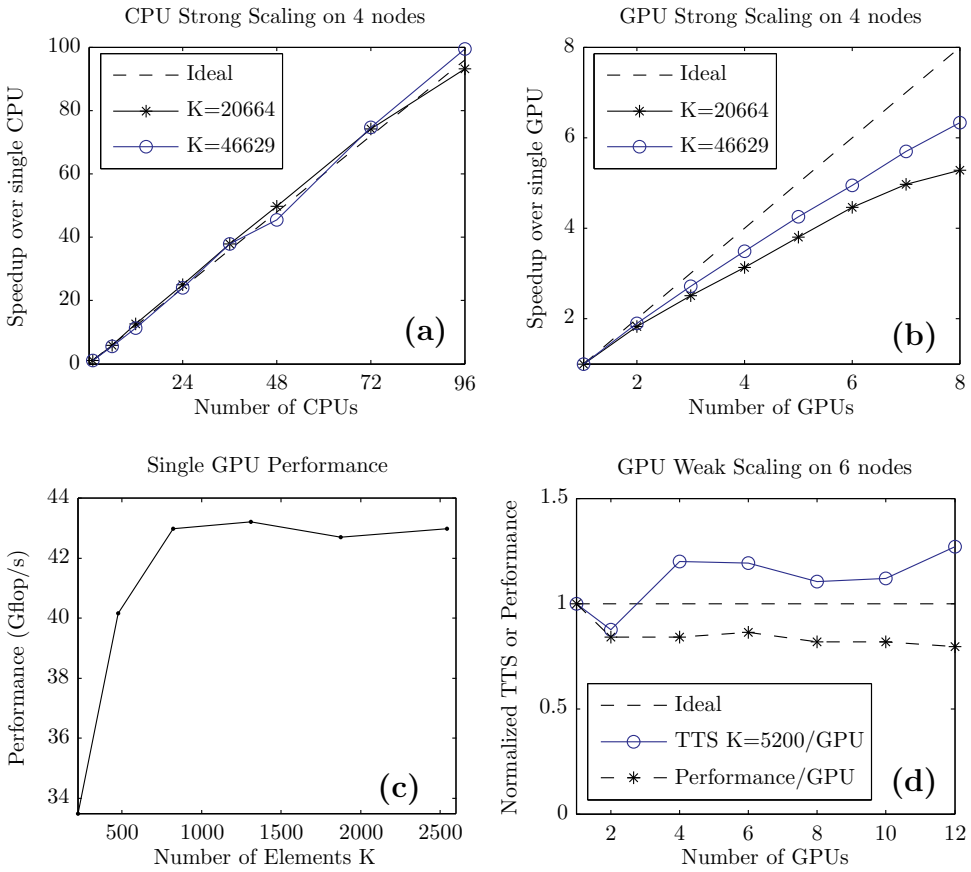

Fig. 1. Scaling Curves for CPU and GPU simulations.

The CPU code scaled extremely well across both cores and nodes, as can be seen in panel (a) of Fig. 1. As the GPU code was similarly structured to the CPU code, the impressive scaling of the CPU code allows us to more objectively analyze the GPU code (i.e., MPI may not by the culprit for poor GPU scaling performance). The code was scaled from 1 processor, up to four nodes, each with twenty-four AMD Magnycours cores at 2.2 GHz with a total of 96 cores. In panel (b), we repeated the experiment, but with GPUs instead of CPUs. Here we see how the problem size affects the ability of the GPU version of the code to scale, where the larger problem scales more efficiently than the smaller problem (circles vs. stars). In Panel (c) we note that the GPU solver performance saturates for $K > 1000$. As a performance comparison, each CPU MPI-process was able to achieve approximately

2.0 Gflop/s, which is 48 Gflop/s per node. The GPU code is able to achieve $\approx$ 70 Gflop/s per node using two Fermi C2070s.

Finally, Panel (d) demonstrates weak scaling (solid with circles), where the number of elements per GPU was kept at a constant $K \approx 5'200$ as GPUs were added to the problem. Shown is the normalized time-to-solution (TTS) (solid with circles) as well as the normalized performance of each GPU (dashed with stars) as the problem was grown as compared to the single GPU solution (the performances were similar across GPUs and the number represents an average). The performance of the single-GPU was 44 Gflop/s, and the twelve GPU simulation had an average of 35 Gflop/s each. From the initial drop of the performance from one to two GPUs, it is clear that the introduction of MPI (and thus the need for device $\Leftrightarrow$ host memory transfers) introduces a performance penalty of about 15%.

2.2 Conclusion and Future Work

This paper showed that an acoustic wave-equation solver based on the MIDG code performs well on both CPUs and GPUs. The MIDG scales excellent on CPUs, whereas the GPU version still needs work, especially if simulations are going to be run on a larger cluster with hundreds of GPUs. However, the simplicity of the code should provide a good testing ground for improving this scaling, which may also be taken to other similarly structured GPU-based solvers.

One possible way to improve the scaling performance is through the use of pinned host memory and asynchronous data transfers. This should allow the data transfer to be overlapped along with the MPI communications. It will also be useful to explore whether any of the thread coarsening and register optimizations used in the implementation of the Matrix-Vector, and Matrix-Matrix CUBLAS 3.x routines could be useful here [5].

In addition, despite the impressive scaling of the code across all 24 cores of each node using purely MPI, it may be worthwhile to add OpenMP threading to the element loop within each MPI-process in order to reduce communication and improve scaling for larger simulations.

References

1. J.S. Hesthaven and T. Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. Springer Verlag, 2007.
2. G. Karypis, K. Schloegel, and V. Kumar. ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 3.1. *University of Minnesota, Minneapolis*, 2003.
3. A. Klöckner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863–7882, 2009.
4. K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.
5. Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.