# ACCELERATING DGM WITH GRAPHICS PROCESSING UNITS AND KOKKOS

PEDRO D. BELLO-MALDONADO*, CURTIS OBER†, AND KENNETH BELCOURT‡

**Abstract.** Numerical computing has seen a rapid increase in the past several years as simulations became the go to approach for product development and scientific discovery. Implemented in DGM, the Discontinous Galerkin method is one of the most popular computing tools due to its versatility in adapting to meshes with mixed elements as well as better accuracy and general numerical properties. A GPU parallelization with Kokkos was added to DGM achieving improvements of twice the speed when considering the communication time and 20x the speed without communication. This initial implementation aims to serve as proof of concept in order to demonstrate how GPU computing can significantly improve the execution time of DGM for further development in the future.

**1. Introduction.** Given the rise of computing power over the last 50 years, computer simulations have become a key component in the development of new products and services. Companies save millions of dollars every year by testing their products using computers instead of building disposable prototypes. Furthermore, better and better simulation tools and methods are developed by scientists and engineers to improve the quality of the results and reduce the time to completion. This trend is seen in all sectors of the economy from engineering to finance, and the demand for skilled workers in these fields is expected to grow significantly in the next years.

While the computing resources are mostly available to anyone through funding agencies like NSF and the XSEDE program or private clusters, utilizing these resources can be daunting for non-computing experts. Furthermore, with newer computing architectures emerging every year it is hard to determine what the best option is when choosing a platform for running a particular simulation. Different systems offer different features that may be well suited for a specific application while being the wrong choice for a different one. Even if it is clear that a certain platform is the right choice for an application developing code to maximize resource usage is not possible without significant effort. For this reason, new programming tools that aim to overcome this issue are being developed to maximize the performance of applications while keeping it simple for non-experts to use. Compilers are also getting smarter and smarter at determining how code translates to machine instructions that make use of the resources effectively. In this work we focus on one of these tools named Kokkos used for GPU computing and a computing method known as the Discontinous Galerkin (DG) method for the solution of partial differential equations. Our main objective is to accelerate the execution time of a DG code using GPUs.

The rest of this paper is organized as follows. In Section 2 we introduce the DG method at a very high level and the code that implements it named DGM. Also a survey of GPU computing tools is listed as to understand where Kokkos lies in the spectrum of programming tools for GPUs. We present here the basics of the Kokkos library and show a simple example so the reader appreciates how easy it is to use. Section 3 describes the approach used to accelerate a portion of DGM that fits well in the GPU programming paradigm. Results of our efforts are summarized in Section 4 with plots of the speedups achieved, and finally we

*Dept. of Computer Science, University of Illinois at Urbana-Champaign, belloma2@illinois.edu
†Sandia National Laboratories, ccober@sandia.gov
‡Sandia National Laboratories, kbelco@sandia.gov

conclude our work in Section 5.

**2. Background.** In the realm of Scientific Computing, the numerical solution of partial differential equations (PDEs) is at the core of scientific discovery and product development. These equations describe many physical systems involving multiple independent and dependent variables, making them almost impossible to solve in closed form (i.e. the solution is given by a function or set of functions satisfying the equations exactly). Many numerical methods have been proposed to address this issue. The general approach in these methods is to decompose the domain (i.e. the physical space where the physics is happening) into many well defined, disjoint elements and use them to solve the equations at a set of discrete points (e.g. nodal methods). Other approaches aim to approximate the solution using interpolation where the unknowns become the coefficients in the interpolating function (e.g. modal methods) [12, 2, 3].

In either case the computational complexity of the method can be very expensive if the solution is to be very accurate. This can be done by increasing the number of elements, known as $h$-refinement, or increasing the polynomial degree of the basis functions in each element, known as $p$-refinement. Furthermore, for time evolving PDEs, different time-stepping approaches are possible that can be of explicit nature or implicit nature, each with its own advantages and disadvantages [7, 9].

**2.1. The Discontinous Galerkin Method and DGM.** One of the numerical methods for the solution of PDEs is the Discontinous Galerkin Method or DG for short. DG methods combine features of the Finite Element method (FE) and the Finite Volume method (FV) to overcome the challenges each one has with respect to accuracy, $hp$-adaptivity, and explicit/implicit time-stepping. Solutions to the PDE are locally approximated in the Galerkin sense (i.e. the residual is minimized using the weighted residual formulation at each element) which causes discontinuous solutions at the interface of elements. In order to pick one solution fluxes are used as in the FV method and in this way the discretization is complete and the solution unique [5].

Many codes are available for the discretization of PDEs with DG. In this work we focus on DGM developed by Scott Collis and his students at Rice University and now maintained at Sandia National Laboratories. The code features discretization in space using mixed nodal and modal representations, as well as a suite of time integrators with many examples of different classes of PDEs (e.g. Navier-Stokes, advection-diffusion, Poisson, etc.) [14].

**2.2. Parallel Computing and Accelerators.** Once the problem being solved gets too large to fit on a single machine or too expensive to run in a timely manner, high performance computing becomes the go to tool. In this computing paradigm, the problem is broken down and distributed across multiple processors that compute the solution in parallel. Significant speed improvements can be possible with enough computing units. Furthermore, specialized hardware designed to maximize computational throughput can be used to further accelerate the execution time of the application. Currently, two architectures are the most widely adopted in supercomputing clusters: graphics processing units (GPUs) and Intel Xeon Phi coprocessors (the newest version is not a coprocessor anymore. See Intel Knights Landing).

Different programming tools are available for GPU computing. For NVIDIA GPUs the Compute Unified Device Architecture (CUDA) programming model is the standard for programming the accelerators [6]. CUDA is an extension to the C/C++ language that

supports GPU computing and provides routines to manage memory and run functions (called kernels) on the GPU. A similar tool that supports other GPU vendors, as well as NVIDIA, is the OpenCL library. Both of these tools work very close to the hardware and require the user to carefully map computational work to the manycore architecture on the GPU. Higher level programming libraries such as OpenACC and OpenMP make it easier and faster for programmers to write GPU code without worrying about the underlying architecture at the expense of performance [13, 11]. In these cases the programmer instruments the sequential code with compiler directives that aid the compiler in deciding how to port the code to the GPU and run it.

A third category of programming models for GPUs and other multithreaded architectures aims to use abstractions to describe the parallel computation so it can be run on different architectures. These tools provide a "write once, run everywhere" approach that enables parallelism without writing code for each different platform. Some examples include the Kokkos library in Trilinos [10], VTK-m [8], and Loo.py [1].

**2.3. The Kokkos Library.** Developed by Carter Edwards and Christian Trott at Sandia National Lab, the Kokkos library implements a programming model in C++ for writing performance portable applications targeting all major high performance computing (HPC) platforms [4]. Architectural features are abstracted so the user need not worry about the specifics of the platform the code is to be run on.

Abstractions in Kokkos have the following forms:

- *Memory Space*: Describes where the data resides (e.g. device memory, host memory)
- *Execution Space:* Describes where the functions are executed (e.g. GPU, XeonPhi, CPU)
- *Execution Policy:* Describes how and where a user function is executed (e.g. concurrently call function $f(i)$ for $i = [0, n)$)
- *Pattern:* Describes the execution pattern (e.g. parallel for, reduce, scan, or task)

In Listing 1 a simple example of Kokkos code is presented. The `Kokkos::View` object is a smart pointer that allocates memory on the memory space defined at compile time or specified in one of the template parameters. Then after that we launch a kernel on the GPU using `Kokkos::parallel_for` with `num_elem` threads and use the indexing to assign an initial value to each element in the array. In this way, we are letting Kokkos define the underlying distribution of thread blocks (or teams depending on the architecture) on the device. More specific distributions of threads can be possible for finer control, as well. On the other hand, notice that it is possible to change the memory space and execution space by just recompiling the code with the appropriate flags without changing the source code at all. For further details please refer to the Kokkos documentation.

LISTING 1
*Kokkos example*

```
// Create a view object that allocates data on the device
Kokkos::View<double*> array("array", num_elem);

// Launch a kernel and initialize the data
Kokkos::parallel_for(num_elements, KOKKOS_LAMBDA (const int i)
{
    array[i] = initial_value;
});
```

**3. Methodology.** In order to support a large number of elements during simulation, DGM was implemented in parallel with MPI. The code has been run on hundreds of thousands of processors effectively and has been tested in different platforms. However, little effort has been put into migrating portions of the code to GPUs. Under the right circumstances, significant improvements can be achieved with GPUs and in this section we explain our approach to parallelize DGM with GPUs and Kokkos.

**3.1. Element Blocks.** Meshing a domain significantly impacts the performance of any numerical method for PDEs. Good meshes represent the physical domain as closely as possible, but using some element types may be too expensive or too hard to mesh. DGM supports mixed elements enabling a large number of possibilities for the programmer to achieve good representation. This feature in DGM allows for meshing portions of the domain with the exact same element while other portions may have more unstructured ones. Computationally speaking this is very efficient since many properties of the element are shared among identical elements thus reducing recomputing of such properties. An example of this is the Jacobian of the element which is used to map the local element to the reference element. In DGM, an `ElementBlock` class is provided for this purpose. All the elements in an `ElementBlock` object are aligned in memory as to facilitate accessing the data of all the elements with a single pointer. This is a key feature in the parallelization with GPUs since a single copy is possible from the host to the device without worrying about padding the field data of each element in a contigous array on the device.

**3.2. Functions of Interest.** Maximizing efficiency on GPUs in order to achieve excellent speedups is not always possible if the task being parallelized doesn't fit the computing paradigm. A good understanding of the code and the problem it solves is necessary before parallelization efforts are put in place. Here, we show the functions of interest that we propose to port to the GPU and their mathematical meaning.

**3.2.1. The `inner_product` Function.** For integration reasons, field data of each element needs to be multiplied by the basis functions in the discretization process. This is accomplished by the `inner_product` function in DGM. Mathematically speaking the following steps achieve this. Let $q_a$, $q_b$, and $q_c$ be the quadrature points in the $x$, $y$, and $z$ directions, respectively. Also based on these quadrature points, the polynomial degrees of the basis functions take the form $L_a = q_a - 1$, $L_b = q_b - 1$, and $L_c = q_c - 1$. Let $\boldsymbol{u}^{(e)}$ be a vector of size $q_a q_b q_c$ with the field values for element $e$ at the quadrature points, and $e = 1, \ldots, E$ with $E$ being the total number of elements in the block. Let $\hat{\boldsymbol{u}}^{(e)}$ be the vector of size $L_a L_b L_c$ with field values at the modes for each element. Finally, let J be the weighted Jacobian vector at the quadrature points and, $\boldsymbol{B}_a$ (size $L_a \times q_a$), $\boldsymbol{B}_b$ (size $L_b \times q_b$), and $\boldsymbol{B}_c$ (size $L_c \times q_c$) be the matrices used to project onto the basis functions. Algorithm 1 implements this procedure.

**3.2.2. The `backward_transform` Function:.** While projecting field data onto the basis modes is accomplished with the `inner_product` function, the inverse operation is possible, as well. This is what the `backward_transform` function does. Using the same notation as before, Algorithm 2 shows the implementation in this case.

**3.3. Parallelization.** As described, both Algorithm 1 and 2, can be parallelized in several ways. For starters, element computations are independent of each other. Each individual element could potentially be assigned to a thread and computed in parallel.

**Algorithm 1** `inner_product` function description

**Input:** Number of quadrature points in each dimension, $q_a$, $q_b$, and $q_c$. Number of modes in each dimension, $L_a$, $L_b$, and $L_c$. Weighted Jacobian of the element, J. Matrices to project onto the basis functions, $\boldsymbol{B}_a$, $\boldsymbol{B}_b$, and $\boldsymbol{B}_c$. Field, $\boldsymbol{u}$, for each element. Number of elements, $E$. All matrices and vectors are assumed to be layed out in memory in row major ordering for reshaping purposes.

**Output:** Projected field onto the basis modes, $\hat{\boldsymbol{u}}$, for all the elements

**procedure** INNER_PRODUCT
  **for** $e = 1$ to $E$ **do**
    $\boldsymbol{u}_w = \text{diag}\,(\text{J})\,\boldsymbol{u}^{(e)}$
    $\boldsymbol{U}_w =$ reshape $\boldsymbol{u}_w$ from $q_a q_b q_c$ to $q_a \times q_b q_c$
    $\boldsymbol{H}' = \boldsymbol{B}_a \boldsymbol{U}_w$
    $\boldsymbol{H}'' =$ zero matrix of size $L_a \times L_b q_c$
    **for** $l_a = 1$ to $L_a$ **do**
      $\boldsymbol{H}^{(l_a)} =$ reshape $\boldsymbol{H}'^{(l_a)}$ from $q_b q_c$ to $q_b \times q_c$
      $\boldsymbol{R} = \boldsymbol{B}_b \boldsymbol{H}^{(l_a)}$
      $\boldsymbol{H}''^{(l_a)} =$ reshape $\boldsymbol{R}$ from $L_b \times q_c$ to $L_b q_c$
    **end for**
    $\boldsymbol{H}'' =$ reshape $\boldsymbol{H}''$ from $L_a \times L_b q_c$ to $L_a L_b \times q_c$
    $\hat{\boldsymbol{U}} = \boldsymbol{H}'' \boldsymbol{B}_c^{\top}$
    $\hat{\boldsymbol{u}}^{(e)} =$ reshape $\hat{\boldsymbol{U}}$ from $L_a L_b \times L_c$ to $L_a L_b L_c$
  **end for**
  **return** $\hat{\boldsymbol{u}}$
**end procedure**

Furthermore, the individual computations of each element are basic linear algebra routines that benefit from manycore architectures. Listing 2 and 3 show the implementation of two key portions of Algorithm 1: multiplying the Jacobian by the field (an element to element vector multiplication that is represented as a diagonal matrix multiplied by a vector) and a simple matrix-matrix multiplication. A very similar code is used for `backward_transform` since the math is effectively the same.

In Listing 2 we see a couple of new Kokkos features not shown before. Mainly the `TeamPolicy` object. This object defines how threads are distributed and organized in teams. The number of teams and the number of threads per team is given in the constructor of the object. For each team, a `TeamThreadRange` object organizes the threads and gives them IDs to be referenced in the body of the function where they are used. The CUDA equivalent of teams and thread range, in this case, is thread blocks and threads in a one dimensional configuration. Kokkos manages the mapping automatically and takes care of the correct indexing at the boundaries. Finally, in the body of the function each element is indexed by `e * (q_a * q_b * q_c)` and each quadrature point is multiplied by the Jacobian. Notice that this is only possible because the elements in an `ElementBlock` have consecutive memory addresses separated by blocks of size $q_a q_b q_c$. We are not showing the view allocation here, but it suffices to say that `u` and `wJ` reside in global memory on the device and have the correct data.

For Listing 3, a basic matrix-matrix multiplication is implemented for each element. The

**Algorithm 2** `backward_transform` function description

**Input:** Number of quadrature points in each dimension, $q_a$, $q_b$, and $q_c$. Number of modes in each dimension, $L_a$, $L_b$, and $L_c$. Matrices to project onto the basis functions, $\boldsymbol{B}_a$, $\boldsymbol{B}_b$, and $\boldsymbol{B}_c$. Projected field, $\hat{\boldsymbol{u}}$, for each element. Number of elements, $E$. All matrices and vectors are assumed to be layed out in memory in row major ordering for reshaping purposes.

**Output:** Field value on the grid points, $\boldsymbol{u}$, for all the elements

**procedure** BACKWARD_TRANSFORM
  **for** $e = 1$ to $E$ **do**
    $\hat{\boldsymbol{U}} =$ reshape $\hat{\boldsymbol{u}}^{(e)}$ from $L_a L_b L_c$ to $L_a \times L_b L_c$
    $\hat{\boldsymbol{H}}' = \boldsymbol{B}_a^\top \hat{\boldsymbol{U}}$
    $\hat{\boldsymbol{H}}'' =$ zero matrix of size $q_a \times q_b L_c$
    **for** $q = 1$ to $q_a$ **do**
      $\hat{\boldsymbol{H}}^{(q)} =$ reshape $\hat{\boldsymbol{H}}'^{(q)}$ from $L_b L_c$ to $L_b \times L_c$
      $\hat{\boldsymbol{R}} = \boldsymbol{B}_b^\top \hat{\boldsymbol{H}}^{(q)}$
      $\hat{\boldsymbol{H}}''^{(q)} =$ reshape $\hat{\boldsymbol{R}}$ from $q_b \times L_c$ to $q_b L_c$
    **end for**
    $\hat{\boldsymbol{H}}'' =$ reshape $\hat{\boldsymbol{H}}''$ from $q_a \times q_b L_c$ to $q_a q_b \times L_c$
    $\boldsymbol{U} = \hat{\boldsymbol{H}}'' \boldsymbol{B}_c$
    $\boldsymbol{u}^{(e)} =$ reshape $\boldsymbol{U}$ from $q_a q_b \times q_c$ to $q_a q_b q_c$
  **end for**
  **return** $\hat{\boldsymbol{u}}$
**end procedure**

Listing 2
*Element-to-element multiplication of the field, $\boldsymbol{u}$, by the weighted Jacobian,* J

```
typedef Kokkos::Cuda Space;
typedef Kokkos::TeamPolicy<Space> TeamPolicyExec;
typedef Kokkos::TeamPolicy<Space>::member_type MemberTypeExec;

Kokkkos::parallel_for(TeamPolicyExec(num_elem, num_threads),
    KOKKOS_LAMBDA (const MemberTypeExec& team_member)
{
    const int e = team_member.league_rank();

    Kokkos::parallel_for(Kokkos::TeamThreadRange(team_member, q_a * q_b * q_c),
        [=] (const int i)
    {
        u[e * (q_a * q_b * q_c) + i] *= wJ[i];
    });
});
```

structure is similar to that of the element-to-element multiplication code but another layer of parallelism is included. The `ThreadVectorRange` object adds an extra dimension to the thread teams. For that reason in the constructor of the `TeamPolicy` object two arguments are given defining the block size in each dimension. The Kokkos library computes the number of blocks necessary to cover the ranges in `TeamThreadRange` and `ThreadVectorRange`,

LISTING 3
*Matrix-matrix multiplication of the basis function matrix and the reshaped Jacobian-multiplied field*

```
Kokkos::parallel_for(TeamPolicyExec(num_elem, block_y, block_x),
    KOKKOS_LAMBDA (const MemberTypeExec& team_member)
{
    int e = team_member.league_rank();

    Kokkos::parallel_for(Kokkos::TeamThreadRange(team_member, num_modes),
        [=] (const int i)
    {
        Kokkos::parallel_for(Kokkos::ThreadVectorRange(team_member, q_b * q_c),
            [=] (const int j)
        {
            double* A = Ba;
            double* B = u + e * (q_a * q_b * q_c);
            double* C = Hp + e * (L_a * q_b * q_c);

            double sum = 0.0;

            for (int k = 0; k < q_a; k++)
            {
                sum += A[i * q_a + k] * B[k * (q_b * q_c) + j];
            }

            C[i * (q_b * q_c) + j] = sum;
        });
    });
});
```

automatically. In the body of the function, a simple loop is used to compute each element of the output matrix which is the equal to the inner product of a row of the first matrix dotted with the column of the second matrix. In this way, the order of operations does not affect the value of the output compared to the sequential implementation and both values should be exactly the same. This was not by design and some accuracy could be sacrificed in spite of more parallelism but that is left for future optimizations of the code. The remaining portions of the routine have similar implementations and thus are not shown here.

**4. Results.** Our tests were run on the Shannon test bed in interative mode. The specifications per node are:

- Two 8-core Sandy Bridge Xeon E5-2670 @ 2.6GHz
- 128GB DDR3-1600MHz
- 2x NVIDIA K20 GPUs
  - 2688 CUDA Cores
  - 732MHz Clock rate
  - 5760MB Global memory
- RHEL 6

We performed 5 runs of each test and averaged the results. In analyzing the effectiveness of our algorithm, the number of elements is varied as well as the polynomial degree of the basis functions in each element. DGM provides an executable to generate a cubic mesh with hex elements of the same structure. Listing 4 shows the command to generate a computational mesh. Notice that the code generates a linear grid of elements of size $1 \times 1 \times 1$ in the $x$-direction. Other testings with a cubic grid of elements did not influence the speedups and the results were equivalent.

LISTING 4
*Mesh generation with DGM (Python string formating)*

```
"dgm_mesh.exe -nsd 3 -nx %d -ny 1 -nz 1 -x0 0.0 -y0 0.0 -z0 0.0 -Lx %f -Ly 1.0 -Lz
    1.0 -r 3d" % (E, E)
```

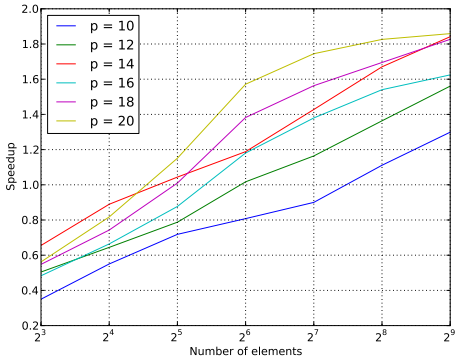


FIG. 4.1. *Overall speedup of* inner_product *function*

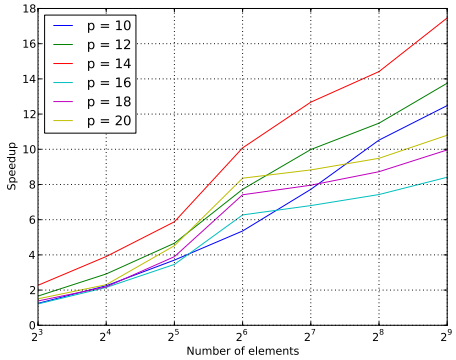


FIG. 4.2. *Speedup of* inner_product *function without communication*

Figures 4.1-4.4 show the measured speed improvements for a `HexBlock` of size 8, 16, 32, 64, 128, 256, and 512, with and without the cost of communication for both functions of interest. In this case, communication means the data transfer between the CPU and the GPU. As we can see, with more grid points to work with, the speed improvement increases since the communication cost has less effect on the overall computation. Furthermore, a higher polynomial degree also means better performance, as a higher polynomial means more grid points per element. However, this trend is not seen when the communication cost is not taken into account. As we can see from Figure 4.2 or 4.4 the configuration with the best performance is a hex element with $p = 14$. This is true regardlessof the number of elements. We believe that, due to the fact that $p = 14$ means $p + 2 = 16$ grid points in each direction, an element with $16^3 = 4096$ grid points fits exactly in one memory page and so it is very efficient, but further investigation is needed.

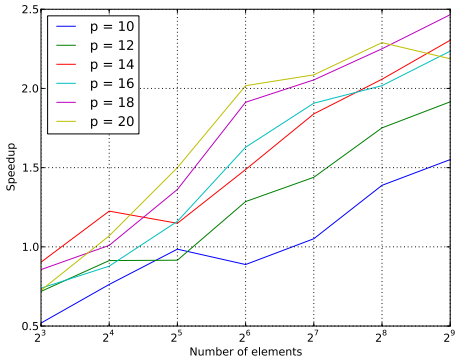


FIG. 4.3. *Overall speedup of* backward_transform *function*

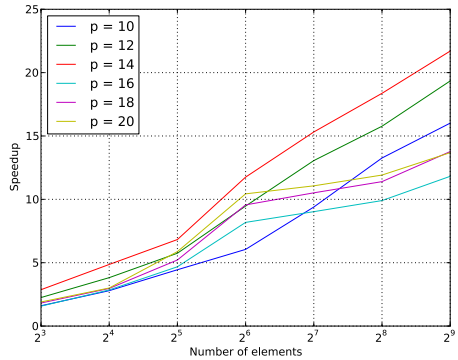


FIG. 4.4. *Speedup of* backward_transform *function without communication*

When looking at the overall execution and not just each individual function an overall speedup gain is also achieved under the right conditions. Figure 4.5 shows these results. As we can see, a total of 5% improvement is seen for a run with 1000 time-steps. We expect that with more portions of the code ported to the GPU larger improvements will be seen.

**5. Conclusions.** Numerical applications benefit greatly from the computing power of GPUs since computation can be broken into small, lightweight pieces that can be mapped efficiently onto the many core architecture of the GPU. However, in order to maximize performance, a careful mapping of the tasks is needed and this can't be done without significant understanding of the architecture. Tools like Kokkos facilitate the parallelization process by abstracting the hardware and generalizing parallel work so multiple platforms can be covered. We showed that improvements of twice the speed are possible even when the communication is present and a relatively low number of elements are available. The results show that increasing the total amount of work manifests into higher speedups. Furthermore, if communication is ignored huge improvements are seen since transferring data from the CPU to the GPU is the bottleneck of GPU computing in general. Future generations of GPUs will share the memory space with the CPU so we expect to see even faster GPU code in the near future. On the other hand, this initial approach to GPU parallelization of DGM

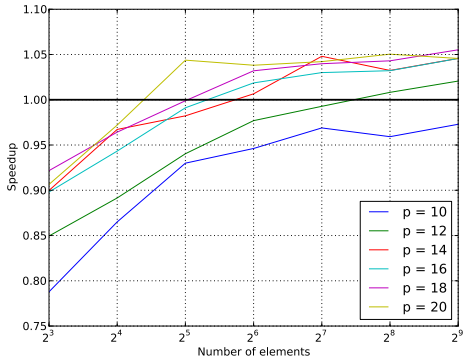


FIG. 4.5. *Total performance gain for a test run with 1000 time-steps. Time measured from the beginning of the application to the end including all aspects of the execution (e.g. reading input files, setup, etc)*

aims to demonstrate how GPUs can be used with Kokkos in accelerating the application. Based on the results obtained, we can move forward and target more portions of the code that fit well on the GPU and expect to see a reduced computing time of DGM as a whole.

## REFERENCES

[1] ANDREAS KLÖCKNER, *Loo.py: Transformation-Based Code-Generation for GPUs and CPUs*, in Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming, Edinburgh, Scotland., 2014, Association for Computing Machinery.

[2] CONSTANTINE POZRIKIDIS, *Introduction to Finite and Spectral Element Methods Using MATLAB, Second Edition*, Chapman and Hall/CRC, Boca Raton, FL, USA, 2014.

[3] DAVID V. HUTTON, *Fundamentals of Finite Element Analysis*, Tata McGraw Hill India, 2003.

[4] H. CARTER EDWARDS AND CHRISTIAN R. TROTT, *Kokkos: Enabling performance portability across manycore architectures*, in 2013 Extreme Scaling Workshop (xsw 2013), Aug 2013, pp. 18–24.

[5] JAN S. HEASTHAVEN AND TIM WARBURTON, *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*, Springer-Verlag New York, 2008.

[6] JOHN NICKOLLS AND IAN BUCK AND MICHAEL GARLAND AND KEVIN SKADRON, *Scalable Parallel Programming with CUDA*, Queue, 6 (2008), pp. 40–53.

[7] K. SUBBARAJ AND M.A. DOKAINISH, *A Survey of Direct Time-Integration Methods in Computational Structural Dynamics-II: Implicit Methods*, Computers & Structures, 32 (1989), pp. 1387–1401.

[8] KENETH MORELAND AND CHRISTOPHER SEWELL AND WILLIAM USHER AND LI-TA LO AND JEREMY MEREDITH AND DAVID PUGMIRE AND JAMES KRESS AND HENDRIK SCHROOTS AND KWAN-LIU MA AND HANK CHILDS AND MATTHEW LARSEN AND CHUN-MING CHEN AND ROBERT MAYNARD AND BERK GEVECI, *VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures*, IEEE Computer Graphics and Applications, 36 (2016), pp. 48–58.

[9] M.A. DOKAINISH AND K. SUBBARAJ, *A Survey of Direct Time-Integration Methods in Computational Structural Dynamics-I: Explicit Methods*, Computers & Structures, 32 (1989), pp. 1371–1386.

[10] MICHAEL A. HEROUX AND ROSCOE A. BARTLETT AND VICKI E. HOWLE AND ROBERT J. HOEKSTRA AND JONATHAN J. HU AND TAMARA G. KOLDA AND RICHARD B. LEHOUCQ AND KEVIN R. LONG AND ROGER P. PAWLOWSKI AND ERIC T. PHIPPS AND ANDREW G. SALINGER AND HEIDI K. THORNQUIST AND RAY S. TUMINARO AND JAMES M. WILLENBRING AND ALAN WILLIAMS AND KENDALL S. STANLEY, *An Overview of the Trilinos Project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.

[11] OPENMP ARCHITECTURE REVIEW BOARD, *OpenMP Application Program Interface Version 4.5*, November 2015.

[12] PHILIPPE G. CIARLET, *Finite Element Method for Elliptic Problems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[13] Sandra Wienke and Paul Springer and Christian Terboven and an Mey, Dieter, *OpenACC: First Experiences with Real-World Applications*, in Proceedings of the 18th International Conference on Parallel Processing, Euro-Par'12, Berlin, Heidelberg, 2012, Springer-Verlag, pp. 859–870.
[14] Scott Collis, *DGM*. For more information contact sscoll@sandia.gov.