A Template for Scalable Continuum Dynamic Simulations in Multiple GPUs

Julián Becerra-Sagredo[1(✉)], Francisco Mandujano[2], Carlos Málaga[2], Jaime Klapp[1,3], and Irene de Teresa[4]

[1] "ABACUS" Centro de Matemáticas Aplicadas y Cómputo de Alto Rendimiento, Departamento de Matemáticas, Centro de Investigación y de Estudios Avanzados (CINVESTAV-IPN), Carretera México-Toluca Km 38.5, La Marquesa, 52140 Ocoyoacac, Estado de México, Mexico
juliansagredo@gmail.com
[2] Departamento de Física, Facultad de Ciencias, UNAM, Ciudad Universitaria, 04510 Mexico, D.F., Mexico
[3] Departamento de Física, Instituto Nacional de Investigaciones Nucleares, La Marquesa Ocoyoacac s/n, Ocoyoacac, Edo. de México, Mexico
[4] Department of Mathematical Sciences, University of Delaware, Newark, DE 19716, USA

Abstract. In this work we present a programming philosophy and a template code for achieving computational scalability when using multiple graphics processing units (GPUs) in the numerical solution of any mathematical system of equations found in continuum dynamic simulations. The programming philosophy exploits the principal characteristics of the GPU hardware, with emphasis in the delivering of threads with massive memory fetches, intense calculations using local registers and limited writes to global memory. The philosophy requires explicit formulas for calculations for which domain decomposition is trivial. The domains are decomposed in regions that use the local central processing unit (CPU) to communicate common interfaces using the message passing interface (MPI). A template code for the heat equation is established and tested for scalability. The novelty is that we show a series of codes, constructed from the basic template, that solve all the basic model equations found in continuum dynamics, and present illustrative results. The model equations are the heat equation, the Poisson equation, the shallow-water equations, the flow in porous media equations and the vorticity equations.

1 Introduction

The last decade has been witness to radical changes in number crunching hardware [1,2]. The graphics processing unit (GPU) has reached approximately an order of magnitude in peak, double precision, floating point operations per second (FLOPS), compared to the central processing unit (CPU). Today, the GPU can deliver more than 3000 GFLOPS, while the CPU delivers just above 500 GFLOPS. Additionally, the peak memory bandwidth, the velocity to fetch data

I. Gitler and J. Klapp (Eds.): ISUM 2015, CCIS 595, pp. 473–484, 2016.
DOI: 10.1007/978-3-319-32243-8_33

from memory, is around 500 GB/s for the GPU compared to under 80 GB/s for the CPU. And the trends are getting steeper for every new processor generation presented [3]. At the present time, the GPUs double their performance approximately every year while the CPUs do it almost every two years.

Supercomputers are increasingly relying upon the GPUs each year [2], mostly to increase the FLOPS delivered given a fixed monetary budget. Besides their highest performance, the GPUs bring savings because they deliver the best cost/operation and energy/operation ratios. Surprisingly, the transition to GPUs has been slow in the programming side and supercomputers are still dominated by distributed, multi-core applications. One of the main reasons behind the slow implementation of algorithms into the GPU is that the methodologies need to be adapted to the GPU architecture following a theoretical ideal GPU-programming philosophy [3]. Perhaps, the applied mathematics community has not paid sufficient attention to the changes in the hardware and has produced a series of methodologies that might be suited for some GPU acceleration [4,5], but not for exclusive, multiple GPU, scalable performance.

In the field of continuum dynamics, theoretical research has been mainly focused in weak formulations of partial differential equations and their discretization using finite elements [6–8] . The research groups in applied mathematics have produced a great variety of theorems and applications for this particular formulation [9–13]. The finite elements are highly attractive because complex geometries and a mixed systems of equations, including constrains, can be discretized and solved using implicit formulations, reducing stability constrains for the time steps. Once the numerical engine for the discretization and the inversion of the matrix has been implemented, the extension of the method to new applications is relatively simple, and new research can be directed to find the numerical convergence of the algorithms and the best choice for the preconditioning of the matrix. The finite element method produces a great sparse matrix with mixed entries that in most cases is solved using a General Minimum Residual (GMRES) algorithm [14].

The applied mathematician often overlooks the use of the GMRES algorithm as a black box and focuses mainly in establishing the weak formulation of the partial differential equations. Here is where we find the greatest obstacle for scalable simulations. The community argues that sooner or later they will have a general sparse matrix inverter, GMRES or else, working on multi-processors and they will be able to simply change the matrix inversion engine in their codes. The community had some success using domain decomposition for the inversion of the sparse matrix in systems of multiple processors but, to this date, this is still one of the most important open themes in the field [15,16].

Unfortunately for the domain decomposition of matrices, many-core architectures like GPUs are quite different than multi-cores. Multi-cores can assign a large region of elements in the sparse matrix to each core and use iterative matchings to achieve convergence. The GPU requires one code applied independently to each element, using massive memory fetches for the discretization and writing its result to memory, fully in parallel.

The use of methods like GMRES makes sense in GPUs for systems of linear or non-linear equations to be solved for every element, independently. That is, a local

inversion of a small matrix per thread. But large matrix inversions are not easy to implement in parallel. The main problem is that GMRES was designed and optimized as a sequential algorithm. It requires matrix-vector multiplications, rotations and the creation of a basis using the Gram-Schmidt algorithm. Matrix-vector multiplications can be implemented in parallel, but the rotations and creation of the basis vectors is always sequential [17]. Additionally, the domain decomposition of matrices using iterative algorithms faces a fundamental problem, the slow transmission of the long distance components of the solutions. For example, when solving a Poisson equation, the potential has always local components due to sources located at very long distances. Numerically, the computation must be able to communicate those sources rapidly when solving for the corrections at the intersection of the domains. Given the algorithm in its current form, only partial accelerations can be achieved for a large number of cores, computing matrix multiplications in the GPUs and reductions in the CPUs [17].

Intensive research is being done in the field and the reader must not loose attention to new developments. Finite elements can adapt to the GPU programming philosophy with some modifications of the general matrix strategy. Some alternatives are to produce explicit finite element methods and to reduce the equations complexity to allow the use of multigrid. Successful implementations can be found in this direction [18–23].

Other methods have evolved under relative isolation during the fertile decade of finite elements, these include finite differences, finite volumes [24] and Lagrangian advection [25,26]. The algorithms are popular among engineers but are ignored or regarded as low order by most mathematicians. In this direction, our team has been successful in producing new high-order numerical schemes that combine the best features of these methods, exploiting explicit integration, full multi-dimensionality, fast memory fetches and fine grain parallel processing, while avoiding memory latencies with a fixed memory model.

In this paper we show that the development of high-order, moment preserving, semi-Lagrangian schemes [27], combined with the explicit solution of diffusion equations and multigrid or multiscale algorithms [28], provide a high-order convergent framework for incompressible, compressible and constrained continuum dynamics, ideally scalable in an array of many-core architectures.

First, we present a general programming philosophy and a heat equation template for multiple GPUs, showing its scalability using a large number of units. After that, the programming philosophy is applied to a series of model equations that contain all the possible systems of equations found in continuum dynamics. These are pure hyperbolic systems like the shallow-water equations, elliptic-hyperbolic systems like the vorticity equations, and parabolic-hyperbolic systems like the porous media equations.

2 Programming Model

The programming model must exploit the hardware characteristics of the GPUs to achieve peak performance in each card, and communicate several GPUs

efficiently to obtain scalability. It is possible that we obtain acceleration using several GPUs, but the main goal must be the run of very large problems with small time penalties. We must be able to run a multiple sized problem in the same multiple number of GPUs for approximately the same computing time as the unit problem run in one. Therefore, we need to observe how the computational time varies with the number of mesh points for a fixed problem size per GPU, known as weak scalability.

Each GPU has a very large memory bandwidth and a very large number of low power processors. The programming model for each GPU must encourage the massive parallelization of the work space into the maximum number of threads, each using of the maximum number of memory fetches, intense computations in local registers and a few memory writes. If global iterations are needed, the iterations are better implemented on top of the GPU routines, after full workspace operations are finished.

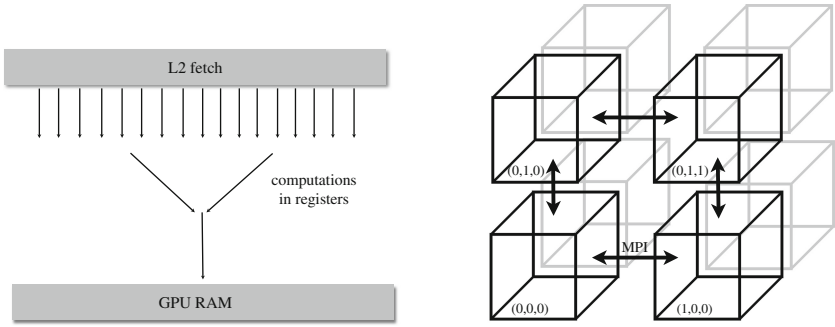

Fig. 1. Programming model for a single GPU (left) and message passing model for many GPUs (right). For each node in the numerical domain, massive L2 memory fetches are combined with intensive operations in registers and limited writes to global memory. The data missing in each face is transferred by the CPUs with MPI.

Figure 1 (left) shows the programming model for a single GPU. The run is carried out entirely on the GPU without large data transfers to the CPU. Inside the GPU, each thread represents a node of the numerical grid. For every thread, there are many memory reads using a read-only array, allowing the state-of-the-art compiler to assign a fast L2-memory transfer without the need to declare texture transfer arrays. This allows fast and massive memory transfers to local registers. Then the registers are used for the numerical operations and the results are written back to a write-only array in global memory. Every algorithm must be converted in every possible way to this philosophy in order to efficiently exploit the capabilities of the GPUs.

The multiple GPU model is based on domain decomposition of the computational domain and communication using the message passing interface MPI. Domain decomposition of continuum dynamics problems always involve communications between domains to update the domain border data. Depending on the

approximation order of the algorithm, more than one line of border data can be communicated. The border data is stored in a special array that is transferred to the local CPU and communicated to the corresponding neighbor CPU using MPI. Once the data has been transferred between CPUs, each CPU loads the border data to their respective slave GPU.

Figure 1 (right) shows a three-dimensional domain decomposition and a sketch of the communication model using MPI. The domain decomposition can be done in one, two or three dimensions, as desired, but it must be kept in mind that it could be important to minimize the area to be communicated. A full three-dimensional domain decomposition provides the minimal area of communication.

Bounded irregular domains can be handled including the irregular domain in a bounding box and eliminating the unnecessary nodes. Infinite domains must be handled using adequate boundary conditions.

3 Template Code

We establish a template code for the chosen programming philosophy. A programmable building block, basis of all the algorithms presented here, used for diagnostics of performance and scalability.

The model problem chosen for the template is parabolic, known as the heat diffusion equation

$$\frac{\partial u}{\partial t} + \kappa \nabla^2 u = f, \tag{1}$$

where $u = u(x,t)$, $x \in [0, L_1] \times [0, L_2] \times [0, L_3]$, $t = [0, \infty]$, $u(x,0) = u_0(x)$, and the given source $f = f(x,t)$.

We use first order forward differences in time and second order, 27-point stencil finite differences in space. We obtain an explicit algorithm that reads the values of 27 nodes, operates the finite differences and writes the solution for the next time step.

This algorithm is stable for time steps $\Delta t < \frac{(\Delta x)^2}{2\kappa}$, with $\Delta x = max_i(\Delta x_i)$, and therefore has a strong time step restriction. We use two arrays to distinguish present and future and use them only for reading and writing, respectively, at every time step. Given the characteristics of the algorithm, threaded for every node in the domain, a single GPU achieves peak performance and the penalties of the time step are reduced in the overall computational time. The algorithm is also perfectly scalable for GPUs with different numbers of processors.

For multiple GPUs, the computational domain is partitioned in $N_1 \times N_2 \times N_3$ cubes or parallelepipeds, assigning a subdomain to every GPU. After a full mesh time iteration, the faces of the subdomains are loaded to the local CPU and communicated with MPI using send and receive commands. The corresponding neighboring CPU receives the data and loads it to its slave GPU. The transfer of the data is done in an ordered way for every direction.

We fix the spatial discretization and increase the number of nodes. The code is first run in one Tesla C2075 card and the results are shown in Fig. 2 with a light grey colored line. It shows a linear relation between the number of mesh points

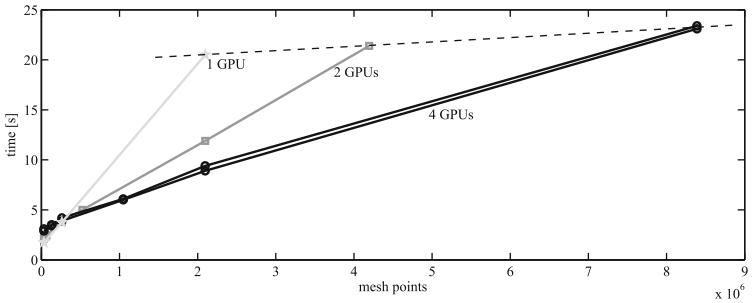

Fig. 2. Computational time against number of mesh points for one, two and four GPUs Tesla C2075. The dotted line shows the almost perfect scalability slope. Runs for 8 million mesh points can be performed by four GPUs almost in the same computational time than 2 million mesh points in one GPU. The small slope of the dotted line indicates the penalization of the message passing between GPUs.

and the computational time. Reaching approximately 20 s for 6400 time steps and two million mesh points. We run the code for two (grey line) and four (black lines) GPUs. We observe that for the same two million mesh points, two GPUs need almost half the computational time than one, accelerating the computation. But four GPUs are not able to do it in a fourth of the time. Accelerating computations using several GPUs is possible but it's not our main goal.

The use of several GPUs is needed to run problems with a large number of mesh points. In Fig. 2, the dotted line shows practically perfect scalability. The little slope shows the penalization time of the message passing to bind the subdomains. It means that we can run more than eight million mesh points in four GPUs, using the same computational time than two million points in one GPU. The slope of the dotted line could be used to estimate the computational time using many GPUs. The black line is double because the run in four GPUs is performed with one- (upper) and two-dimensional subdomain partitions, showing no considerable differences.

When using clusters with a large number of GPUs, we have observed perfect scalability as long as the GPUs share a motherboard. A large penalty can result of the message passing to distributed motherboards depending on the type of network. In such cases, perfect scalability is sacrificed against the possibility of a very large computation. Abacus I, an SGI cluster in Cinvestav, Mexico, achieves perfect scalability for two Tesla K40 cards that share motherboard. The computational time for the 20 s benchmark increases ten times for 27 cards in 14 nodes, computing 56 million points; and fifty times for 64 cards in 32 nodes, computing 134 million points.

4 Model Equations and Algorithms

We show a set of model equations that cover the whole range of the systems found in continuum dynamics simulations. The numerical algorithms are original

and have been developed following the programming model presented for several GPUs. Therefore the algorithms have the same scalability properties of the heat equation template. The heat equation template was our model for parabolic equations. Here, we extend the programming philosophy to elliptic equations, using a multiscale method close to a multigrid [28]; to hyperbolic equations, using a moment preserving high-order semi-Lagrangian scheme [27]; and to mixed systems of equations with constraints.

4.1 Elliptic Equations

The chosen model for elliptic systems of PDEs is the three-dimensional Poisson equation

$$\nabla^2 u = f, \tag{2}$$

for $u \in [0, L_x] \times [0, L_y] \times [0, L_z]$, given $f = f(x, y)$.

This equation is solved using the multiscale algorithm described in [28], closely related to a multigrid. The multiscale algorithm consists of solving heat equations iteratively, using coarse nested discretizations, helped by interpolations to the rest of the mesh, in a descending cycle, until the heat equation is solved in the whole mesh. The process has been seen to converge to machine precision when full cycles are repeated a few more times for the residual function.

In Fig. 3, we show an illustration for the elliptic equation's solver constructed from the template. It shows the solution of the Poisson equation for a singular source with the shape of Mexico. Neumann boundary conditions are used because the solution is used to deform a mesh. The mesh is deformed using equations of motion for each mesh point, with a force computed from the potential found as solution to the Poisson equation.

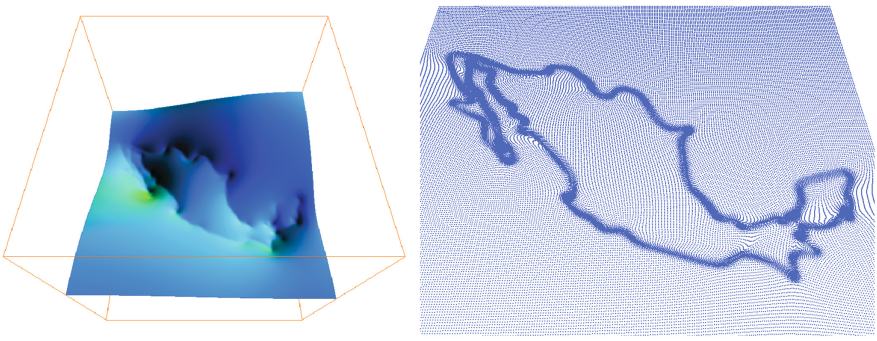

Fig. 3. Potential function, solution of the Poisson equation with Neumann boundary conditions and a singular one-dimensional source with the shape of Mexico. Illustrative for a multigrid, elliptic equation's solver constructed from the template. The solution is used as the potential of a force to move the mesh points of a regular grid and obtain adaptivity.

4.2 Hyperbolic Equations

The chosen model for hyperbolic equations are the two-dimensional shallow-water equations: the conservation of water volume

$$\frac{Dh}{Dt} = -h\nabla \cdot v, \tag{3}$$

and the acceleration of the water column

$$\frac{Dv}{Dt} = -\nabla h, \tag{4}$$

where h is the height of the water column, and v its velocity. The total time derivate, known as the material derivative $D/Dt = \partial/\partial t + v \cdot \nabla$, requires the solution of the trajectories $dx/dt = v$.

The equations are solved using a semi-Lagrangian, moment preserving, numerical scheme described in [27]. The scheme makes use of fluid elements that move and deform, starting from a Cartesian set, the corners of the elements are treated as movable nodes that travel in space following the given equations with total derivatives in time. As the points travel, the element that they describe deforms, and after a few time steps a non-linear exact map is performed to see the new element in a space where coordinates are orthogonal, where high-order, moment preserving interpolations are performed to restart the mesh points in a reference mesh.

The algorithm requires 216 neighbors. It uses arrays for the position of the nodes and computes trajectories and interpolations explicitly. It has a very relaxed CFL time step restriction, as long as the trajectory is well integrated.

In Fig. 4, we present the dynamic solution to the shallow water waves in a tank with variable floor. The floor has an obstacle, a transverse bump with a centered aperture. The solution shows the formation of a diffraction pattern. The waves are originated at the left face, moving the wall with an oscillating piston. The end of the channel has a dissipation zone to kill the waves.

4.3 Model for a Parabolic-Hyperbolic Systems of Equations

The chosen model for a parabolic-hyperbolic system of equations is the three-dimensional flow in porous media: the conservation of mass

$$\frac{Dn}{Dt} + n\nabla \cdot v = q, \tag{5}$$

for the effective density $n = \phi\rho$, where ϕ is the porosity, v is the intrinsic velocity, and q is a mass source; and the pressure equation

$$[(1-\phi)C_r + \phi C_f]\,\frac{\partial P}{\partial t} = \phi\left(-\nabla \cdot v + \frac{q}{n}\right), \tag{6}$$

where C_r and C_f are the compressibiity of the solid and the fluid, respectively. The system is closed using Darcy's law for the intrinsic velocity

$$v = \boldsymbol{K} \cdot (-\nabla P + \boldsymbol{f}), \tag{7}$$

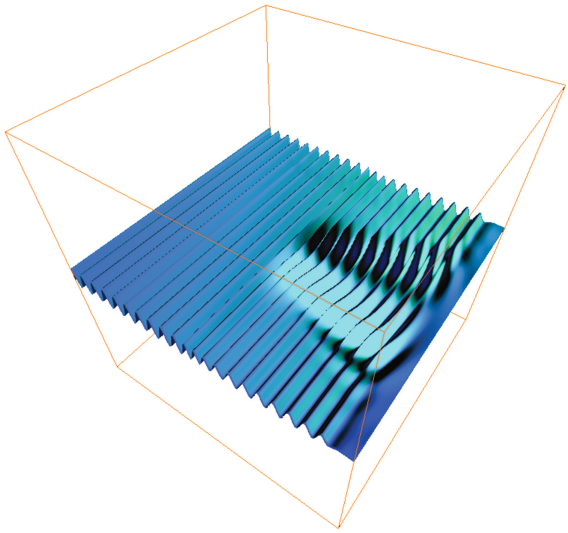

Fig. 4. Height of the water column for shallow water waves in a tank, illustrative of a hyperbolic system of equations' solver constructed from the template. The left wall is moved with an oscillating piston, and the opposite side has a dissipation zone to kill the waves. The floor has a transverse bump with a centered aperture. The waves show diffraction.

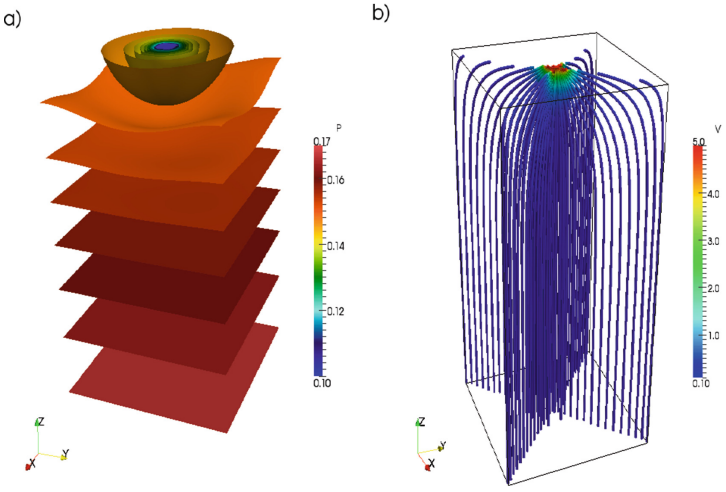

Fig. 5. Solution to the porous media equations for one fluid. (a) Graph of the contours of pressure for an extraction point in the top. (b) Graph of the streamlines colored with the velocity magnitude.

where K is the permeability over the porosity and f is an external force.

We solve the equations using a heat equation solver for the parabolic pressure equation combined with the semi-Lagrangian scheme for the hyperbolic part. In Fig. 5, we show the contours of pressure and the streamlines for a flow in porous media with a sink at the top's face center.

4.4 Model for an Elliptic-Hyperbolic Systems of Equations

The chosen model for an elliptic-hyperbolic system of equations is the two-dimensional vorticity equation

$$\frac{D\omega}{Dt} = 0, \tag{8}$$

coupled to the Poisson equation

$$\nabla^2 \phi = -\omega, \tag{9}$$

with the velocity vector given by $v = \nabla \times \phi \hat{k}$.

These equations represent the motion of incompressible fluids. The vorticity is advected using the semi-Lagrangian scheme and the potential is found solving the Poisson equation with multigrid.

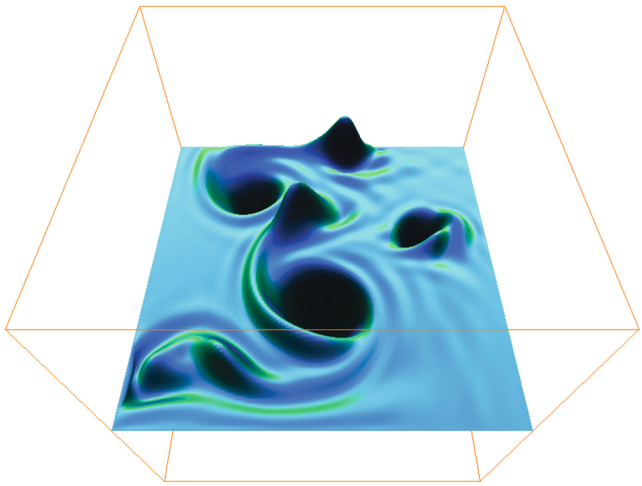

Fig. 6. Snapshot of the vorticity, illustration of the numerical solution of elliptic-hyperbolic systems of equations like those found in incompressible flow simulations, using a code based on the template. We solve the two-dimensional vorticity dynamic equations for an initial random distribution of Gaussian bell vortices with both signs. The box has slip wall boundary conditions.

In Fig. 6, we show a snapshot of vorticity, represented by the height of the mesh, starting from a random distribution of Gaussian bell vortices in positive and negative directions. The box has Dirichlet boundary conditions and therefore the walls have normal velocity equal to zero.

5 Conclusion

We presented a programming model using trivial domain decomposition with message passing for computations using multiple GPUs. The programming model has been proven to be perfectly scalable for GPUs that share a motherboard. In that case, the computational times remain in the same range for runs with increasing number of nodes using a large and fixed number of nodes in each GPU. For clusters of many GPUs in distributed motherboards, we have found a variable penalty associated to the message passings between nodes, dependent on the network. Problems of hundreds of millions of nodes can be solved sacrificing perfect scalability. The novelty of the work is that the programming philosophy is used to implement a diffusion equation solver for parabolic equations, a multiscale solver for elliptic equations, and a semi-Lagrangian advection scheme for hyperbolic equations. The three models are combined in schemes to solve systems of equations that model all the types of systems found in continuum dynamics, for incompressible, compressible and constrained flows.

Acknowledgements. This work was partially supported by ABACUS, CONACyT grant EDOMEX-2011-C01-165873.

References

1. Nickolls, J., Dally, W.J.: The GPU computing era. IEEE Micro **30**(2), 56–69 (2010)
2. Keckler, S.W., Dally, W.J., Khailany, B., Garland, M., Glasco, D.: GPUs and the future of parallel computing. IEEE Micro **31**(5), 7–17 (2011)
3. NVIDIA CUDA C Programming Guide, version 7.5. Nvidia (2015)
4. Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S.H.M., Grajewski, M., Turek, S.: Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. Parallel Comput. **33**(10–11), 685–699 (2007)
5. Lu, F., Pang, Y., Yin, F.: Scalability analysis of parallel algorithms on GPU clusters. J. Comp. Inf. Syst. **10**(14), 5947–5957 (2014)
6. Strang, G., Fix, G.: An Analysis of the Finite Element Method. SIAM, Wellesley Cambridge Press, Wellesley (1973)
7. Kuzmin, D., Hämäläinen, J.: Finite Element Methods for Computational Fluid Dynamics: A Practical Guide. Computational Science and Engineering. SIAM, Philadelphia (2014)
8. Löner, R., Morgan, K., Peraire, J., Zienkiewicz, O.C.: Recent developments in FEM-CFD. In: Fritts, M.J., Crowley, W.P., Trease, H. (eds.) The Free-Lagrange Method. Lecture Notes in Physics, vol. 238, pp. 236–254. Springer, Heidelberg (2005)
9. Yazid, A., Abdelkader, N., Abdelmadjid, H.: A state-of-the-art review of the X-FEM for computational fracture mechanics. Appl. Math. Model. **33**(12), 4269–4282 (2009)
10. Sukumar, N., Malsch, E.A.: Recent advances in the construction of polygonal finite element interpolants. Arch. Comput. Meth. Eng. **13**(1), 129–163 (2006)
11. Belytschko, T., Gracie, R., Ventura, G.: A review of extended/generalized finite element methods for material modeling. Modell. Simul. Mater. Sci. Eng. **17**(4), 1–24 (2009)

12. Schweitzer, M.A.: Generalizations of the finite element method. Cent. Eur. J. Math. **10**(1), 3–24 (2012)
13. Long, Y.Q., Long, Z.F., Cen, S.: Advanced Finite Element Method in Structural Engineering. Springer, Heidelberg (2009)
14. Saad, Y., Schultz, M.H.: GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear system. SIAM J. Sci. Stat. Comput. **7**, 856–869 (1986)
15. Hamandi, L.: Review of domain-decomposition methods for the implementation of FEM on massively parallel computers. IEEE Antennas Propag. Mag. **37**(1), 93–98 (1995)
16. Kruzel, F., Banaś, K.: Vectorized OpenCL implementation of numerical integration for higher order finite elements. Comp. Math. Appl. **66**(10), 2030–2044 (2013)
17. Khodja, L.Z., Couturier, R., Glersch, A., Bahi, J.M.: Parallel sparse linear solver with GMRES method using minimization techniques of communications for GPU clusters. J. Supercomput. **69**(1), 200–224 (2014)
18. Turek, S., Göddeke, D., Becker, C., Buijssen, S., Wobker, H.: UCHPC - unconventional high performance computing for finite element simulations. In: International Supercomputing Conference, ISC 2008 (2008)
19. Płaszewski, P., Macioł, P., Banaś, K.: Finite element numerical integration on GPUs. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part I. LNCS, vol. 6067, pp. 411–420. Springer, Heidelberg (2010)
20. Dick, C., Georgii, J., Westermann, R.: A real-time multigrid hexahedra method for elasticity simulation using CUDA. Simul. Model. Pract. Theory **19**(2), 801–816 (2011)
21. Banaś, K., Plaszewski, P., Maciol, P.: Numerical integration on GPUs for higher order finite elements. Comp. Math. Appl. **67**(6), 1319–1344 (2014)
22. Huthwaite, P.: Accelerated finite element elastodynamic simulations using the GPU. J. Comp. Phys. **257**(Part A), 687–707 (2014)
23. Martínez-Frutos, J., Martínez-Castejón, P.J., Herrero-Pérez, D.: Fine-grained GPU implementation of assembly-free iterative solver for finite element problems. Comput. Struct. **157**, 9–18 (2015)
24. LeVeque, R.J.: Finite Volume Methods for Hyperbolic Problems. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge (2002)
25. Oñate, E., Owen, R.: Particle-Based Methods: Fundamentals and Applications. Computational Methods in Applied Sciences. Springer, Heidelberg (2011)
26. Falcone, M., Ferretti, R.: Semi-Lagrangian Approximation Schemes for Linear and Hamilton-Jacobi Equations. Other Titles in Applied Mathematics. SIAM, Philadelphia (2013)
27. Becerra-Sagredo, J., Málaga, C., Mandujano, F.: Moments Preservind and high-resolution Semi-Lagrangian Advection Scheme (2014). arXiv: 1410.2817
28. Becerra-Sagredo, J., Málaga, C., Mandujano, F.: A novel and scalable Multigrid algorithm for many-core architectures (2011). arXiv:1108.2045