© International Journal of Combinatorial Optimization Problems and Informatics, Vol. 9, No. 1, Jan-April 2018, pp. 3-11. ISSN: 2007-1558.

A GPU Parallel Finite Volume Method for a 3D Poisson Equation on Arbitrary Geometries

M. A. Uh Zapata, F. J. Hernandez-Lopez
Consejo Nacional de Ciencia y Tecnología (CONACYT),
Centro de Investigación en Matemáticas A.C (CIMAT), Unidad Mérida, PCTY, 97302
Sierra Papacal, Merida, Yucatan, México
angeluh@cimat.mx; fcoj23@cimat.mx

Abstract. In this paper, we present a parallelization technique developed to solve an unstructured-grid based Poisson equation on arbitrary three-dimensional geometries using CUDA over a GPU. The Poisson problem discretization is based on a second-order finite-volume technique on prisms elements, consisting of triangular grids in the horizontal direction and bounded by an irregular free surface and bottom in the vertical direction. The resulting linear system is solved using the Jacobi method, and the parallelization is designed by using different GPU memory management as shared, texture and managed memory. Numerical experiments are conducted to test and compare the performance of the proposed parallel technique.

Keywords: GPU computing, PGI CUDA Fortran, 3D Poisson equation, Unstructured grid, σ transformation.

1 Introduction

Fast and accurate numerical solutions of the Poisson equation are important parts for the resolution of many computational physics problems. Typically, the CPU (Central Processing Unit) is used to perform sequential and parallel simulations [1, 2]; however, the current trend is to perform parallel computations using GPUs (Graphics Processing Unit) due to their increasing computational power and low cost. In this paper, a GPU-CUDA (Compute Unified Device Architecture) parallel solver is presented for the finite volume (FV) discretization of the 3D Poisson equation on arbitrary geometries.

Many methods exist to solve linear systems associated with the discretization of the 3D Poisson equation for both structured and unstructured meshes [3–5], but there are few works using parallel methods with a finite volume approach on 3D irregular domains [6–9]. Traditionally, large linear systems are solved using iterative solvers. The Generalized Minimal Residual method and the Conjugate Gradient are popular methods; however, classical stationary iterative methods such as Jacobi, Gauss-Seidel, and Successive Over-Relaxation (SOR) methods have shown stability and high efficiency for this kind of problem [10]. Although the Jacobi method has slow convergence, the simplicity of its algorithm allows us to parallelize the program without any changes.

There are several techniques used to parallelize the iterative linear solver such as GPU-CUDA, MPI, OpenMP, hybrid methods, etc. As an initial parallel approach, in this paper, we propose a Jacobi method in which all the calculations were performed using several kernels in a GPU-CUDA platform. For each kernel, we are allowed to configure several aspects of the grid, including the number of threads per block, as well as the number of blocks. In the proposed algorithm, control volumes are processed in rectangular blocks of data assigned to threads whose configuration was chosen to take advantage of the element distribution of the problem and easy implementation of the code. Therefore, the novelty of this work is to present a strategy and the results of our effort to exploit the computational capabilities of GPUs under the CUDA environment in order to solve the 3D Poisson equation using a finite volume method on arbitrary geometries.

This paper is organized as follows. The second section presents the Poisson equation in the σ-coordinate system, the FV discretization, and the Jacobi linear solver. The third section deals with the parallelization method using GPUs. The fourth section is devoted to numerical examples. Finally, the last section includes the conclusions and future work.

Received Dec 2, 2017 / Accepted Jan 22, 2018

2 The Finite Volume Discretization

Let us consider a Poisson equation in a 3D irregular domain (1) of the form

$$\nabla \cdot (\Gamma(\mathbf{x})\nabla\phi) = f(\mathbf{x}), \quad \mathbf{x} \in \Omega \tag{1}$$

where Γ is the dispersion coefficient. The free surface elevation and the irregular bottom of the domain are changed to a new coordinate using the σ-coordinate transformation (1). It is given by $x* = x$, $y* = y$, and $\sigma = (z + h)/H\ (x, y)$, where $H = h + \eta$ is the total depth with h being a reference depth and η the free surface [11]. Thus, the 3D Poisson equation in σ-coordinates domain Ω∗ is

$$\nabla \cdot \left(\Gamma(\mathbf{x}^*)H(x^*, y^*) \begin{pmatrix} 1 & 0 & \sigma_x \\ 0 & 1 & \sigma_y \\ \sigma_x & \sigma_y & \sigma_x^2 + \sigma_y^2 + \sigma_z^2 \end{pmatrix} \nabla\phi \right) = f(\mathbf{x}^*), \quad \mathbf{x}^* \in \Omega^*, \tag{2}$$

where σ_x, σ_y , σ_z are the partial derivatives of the transformation.

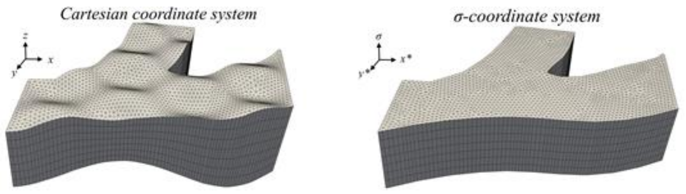

Fig. 1. Example of a physical domain in the Cartesian and σ-coordinate system.

A FV method is chosen to discretize the Poisson problem (2) using a cell- center collocated grid. Thus, the domain Ω∗ is discretized using prism-shaped control volumes: triangular cells in the horizontal domain and a rectangular grid in the vertical domain, see Fig. 2 (b). Following a standard FV scheme, we can write the Poisson equation (2) as a balance equation and apply Green's theorem as follow

$$\int_{\partial b} \Gamma\mathbf{K}\nabla\phi \cdot \mathbf{n}\ dS = \int_b f\ dV, \quad \text{for all } \ b \in \Omega^*. \tag{3}$$

This discretization involves the knowledge of derivatives at the boundary ∂b (consisting of the five faces of a prism) which are approximated as

$$\int_{\partial b} \Gamma\mathbf{K}\nabla\phi \cdot \mathbf{n}\ dS = \sum_{i=1}^{5} \int_{S_i} \Gamma\mathbf{K}\nabla\phi \cdot \mathbf{n}\ dS \approx \sum_{i=1}^{5} (\Gamma\mathbf{K}\nabla\phi)_i \cdot \mathbf{n}_i A_i,$$

where Ai is the area of the interface region S_i, $\mathbf{n}_i$ is the outward normal unit vector and the $(\Gamma\,\mathbf{K}\ \varphi)_i$ approximate the functions over the interface between two neighboring control volumes. The approximation of Γ_i and $\mathbf{K}_i$ can be calculated averaging the two neighbors. However, there is no way to directly calculate the numerical flux φ_i in terms of the cell-center values. In this paper, we approximate the gradient as an average over a new 3D region delimited by the two cell-centered points and the vertices of the interface as shown in Fig. 2 (b) [10].

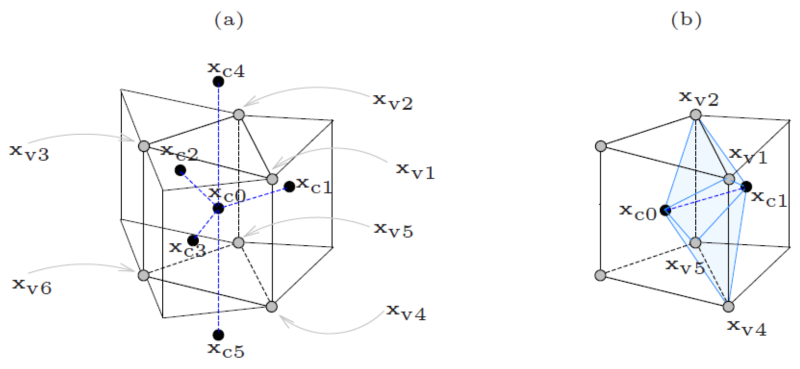

Fig. 2. (a) 3D control volume and (b) region to approximate the interface gradient.

The FV discretization results in a linear system $A\boldsymbol{\varphi}_c = \mathbf{f}$ where an unknown value φ_{c_0} is related to the surrounding cell-centered values φ_{c_j} $(j = 1, 5)$ and auxiliary vertex values φ_v $(j = 1, 6)$ of a control volume by

$$\begin{aligned} a_0\phi_{c_0} + a_1\phi_{c_1} + a_2\phi_{c_2} + a_3\phi_{c_3} + a_4\phi_{c_4} + a_5\phi_{c_5} \\ +b_1\phi_{v_1} + b_2\phi_{v_2} + b_3\phi_{v_3} + b_4\phi_{v_4} + b_5\phi_{v_5} + b_6\phi_{v_6} = f_{c_0}, \end{aligned} \quad (4)$$

where a_j $(j = 1, 5)$ and b_j $(j = 1, 6)$ are the FV coefficients relating the cell- centered and vertex values, respectively. The value at any vertex is obtained by averaging over all surrounding cell-centered nodal values as

$$\phi_{v_i} = \sum_s \omega_s \phi_s \Big/ \sum_s \omega_s, \quad (5)$$

with $\omega_s = 1/L_s$ as weighting value, where L_s is the distance between the vertex and the cell-centered node. The number of values of s depends on each vertex. Finally, Dirichlet conditions are imposed at the boundary, and they are included in the problem as updates of ghost cell-centered values.

As was mentioned before, we find the solution of the resulting linear system by the Jacobi method. In this study, we consider the vertex values as a part of the right-hand side of our problem. However, these values are not fully considered as a fixed term as was the case for the vector $\mathbf{f}$. It should be updated at each iteration using interpolation (5) of the previous iteration unknowns. Thus, the Jacobi method solves the new iteration as:

$$\phi_{c_i}^{(m+1)} = \phi_{c_i}^{(m)} + r, \quad i = 1, 2, \ldots, N_c \quad (6)$$

where N_c is the total number of cell-centered points and r is the residual given by

$$r = \frac{1}{a_{ii}} \left(f_{c_i} - \sum_{j=1}^{N_v} b_{ij} \phi_{v_j}^{(m)} - \sum_{j=1}^{N_c} a_{ij} \phi_{c_j}^{(m)} \right). \quad (7)$$

The $(m + 1)$ values are from the current iteration, and (m) values are from the previous iteration. The iteration will be stopped only when the convergence is obtained. In this iterative solver, each component at the new iteration depends on only previously computed components from the same iteration. It has slow convergence; however, it offers a simple parallel implementation

3 The Finite Volume Discretization

Now we discuss the parallel implementation of iterative solver (6)-(7) using GPU-CUDA. In order to enable CUDA programming directly in Fortran using a GPU, NVIDIA and Portland Group, Inc. (PGI) created CUDA Fortran [12]. A GPU contains multiple transistors for the arithmetic logic unit, which is exploited when multiple data are managed from one simple instruction in parallel. Typical GPU processing involves first transferring the input data from CPU memory to GPU memory; next the data are processed into a kernel function, and then the output data are transferred from GPU memory to CPU memory. The CUDA programming model can run thousands of threads that are grouped into blocks; threads in each block cooperate in an organized way by using the shared memory and can be synchronized into the kernel function. For this particular problem, elements are processed in rectangular blocks of data assigned to threads as shown in Fig. 3. This configuration was chosen to take advantage of the element distribution of the problem and allow for easy code implementation.

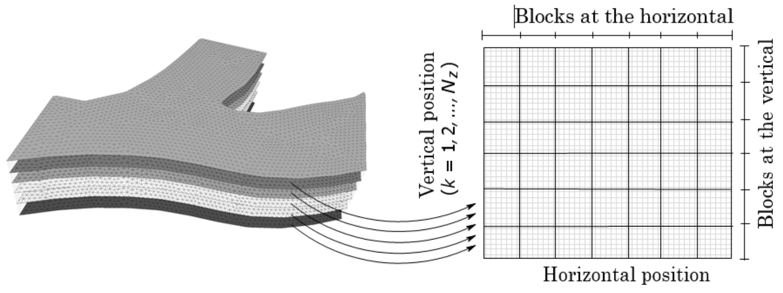

Fig. 3. Mapping of a 3D computational domain to a 2D matrix for the topological distribution of the blocks and threats in the CUDA parallelization.

An essential part of maximizing the performance of our parallel implementation is the GPU memory management (Fig. 4). Kernel functions can read and write in the GPU device global memory using a controller integrated into the GPU. Each thread block has a limited shared memory space (16 to 48 KB) which can be read and written only inside of the block, and is 10 times faster than accesses to global memory [13]. The texture is a read-only cache, which is basically a reference to a CUDA array and contains information on how the array should be interpreted. On the other hand, the unified memory defines a managed memory space in which all CPU and GPU processors see a single coherent memory image with a common address space. Note that for using the unified memory it is not necessary to transfer data from CPU memory to GPU global memory and vice-versa.

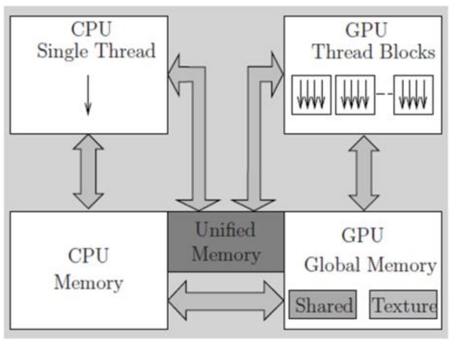

Fig. 4. Memory management between CPU and GPU

We develop three versions of the parallel Jacobi method according to the type of memory used. In the first version (CUDA D), we only use the device global memory, in the second version (CUDA DT), we use texture memory and device global memory, and in the third version (CUDA DTM), we use managed memory in addition to texture and device global memory.

The following three kernels are used for each iteration:

1. k_JacobiMethod. Used to implement all calculations for a single iteration of Jacobi method and to implement one of the most complicated tasks in the code, the calculation of the error. The proposed parallel versions are as follows:

– In the CUDA D version, the values $\phi_{cj}^{(m+1)}$, $\phi_{cj}^{(m)}$, ϕ_{vj} , $a_{ij,}$ b_{ij} and f_{ci} required in (6), reside in matrices loaded in the GPU device global memory. The error is calculated as the sum of residuals (7) of all elements. We calculate part of this sum using a variable in the shared memory space, computing partial reductions in each thread block. Then, we transfer the partial reductions to the CPU memory and the rest of the sum is computed by the CPU.

– In the CUDA DT version, we are using texture memory for $\phi_{cj}^{(m)}$, ϕ_{vj} , $a_{ij,}$ b_{ij} and f_{ci} values because are not modified inside the kernel, i.e. they can be managed as read-only memory. The values $\phi_{cj}^{(m+1)}$ are used as GPU device global memory and the errors are calculated as in the CUDA D version.

– In the CUDA DTM version, the iteration update is calculated the same as in the CUDA DT version; however, the calculation of the error differs. In previous versions, the rest of the sum is computed by the CPU. Meanwhile, in our new version, we are using managed memory for saving the partial reductions. Then the rest of the sum is computed by the GPU in the unified memory, taking advantage of the fact that the host intrinsic function “sum” is overloaded to accept device or managed arrays, when the CUDA for the module in PGI Fortran is used.

2. k_BoundaryCondition. Used to compute the boundary conditions. The Dirichlet condition is applied at all boundaries using a ghost element technique. First, we compute the vertical bounded conditions over φ_c. Next, the threads are synchronized inside the kernel, in order to compute the horizontal bounded conditions. Here, we are using a pre-computed function for the boundary cell center points. In our CUDA DT and DTM versions, this function is loaded in texture memory.
3. k_Interpolation. Used to compute the interpolation technique from the cell-centered values to the vertex values. The calculation effort to obtain the vertex values is comparable with the update solution per computed element. For computing the interpolation of the vertex values φ_v (5) in each call center, we fix the maximum number of surrounding neighborhoods (nine in our experiments). In this way, we unroll the loop which traverses the surrounding neighborhoods, considering weighting value $\omega_s = 0$ when there is not a surrounding neighbor. In this kernel we also apply the boundary conditions over the vertexes, using a pre-computed function which can be loaded in texture memory as in the previous kernel k_BoundaryCondition.

Algorithm 1 shows a PGI CUDA Fortran code excerpt from the CUDA DTM version of the parallelized Jacobi method, where we specify the type of memory to use and the kernel calls. Note that the thread block size tBlock is the same for all launched kernels. The size is fixed according to the best performance found in our numerical experiments. The grid size grid is calculated by dividing the number of cell-centered points by the block size in x and y dimensions, while the grid size gridv is calculated by dividing the number of vertex points by the block size (Fig. 3).

Algorithm 1. CUDA DTM version for the Jacobi parallel implementation

```
          use cudafor
          real*8,dimension(:),allocatable                          ::h_phi real *
          8,dimension(:),device,target,allocatable ::d phi_real*8,
          dimension(:),device,target,allocatable ::d phiNew real*8,te
          xture,pointer                                            ::t_phi(:)
          real*8,dimension(:,:),managed,allocatable                ::m_error

          d_phi = h_phi   ! Copy memory from CPU to GPU t p
          h_i =>d phi_ ! Texture points to the device

500       continue
          error =0.0d0
          call k Jacobi Method <<<grid, t Block, shared_mem >>>()
          call k Boundary Condition <<<grid, t Block >>>()
          call k_Interpolation <<<gridv, tBlock >>>() d p
          hi = d phi_New
          error = sum(m_error)
          if (error.gt.1e-6)  goto 500

          h_phi = d_phi   ! Copy memory from GPU to CPU
```

4 Numerical Experiments and Results

In this section, we discuss various numerical experiments comparing our solver in sequence and parallel. We report the results of the examples on a twenty-core 3.0 GHz Intel Xeon server with a Tesla K40 and Ubuntu 14.04 (64-bits). CUDA kernels were compiled using the pgf90 compiler from PGI CUDA Fortran 2016. The results have been obtained after averaging over several simulations of the same cases. The Jacobi method is used with a tolerance value of $\varepsilon = 10^{-6}$.

For validation and comparison purposes we applied our parallel solvers to an example with a known analytical solution. We solve the Poisson problem (1) with diffusive coefficient $\Gamma = 1$. The right-hand side forcing function and its corresponding analytical solution are given by the functions.

$$f(x,y,z) = -3\pi^2 \sin(\pi x)\sin(\pi y)\sin(\pi z),$$
$$\varphi\,(x,\ y,\ z) = \sin(\pi x)\sin(\pi y)\sin(\pi z).$$

The domain is designed to test the numerical approximation over an irregular configuration. In this example, we consider the domain as shown in Fig. 5. The horizontal domain consists of a curvilinear section of the rectangular domain $[0,1]\text{x}[0,1]$. The irregular bottom and the free surface were chosen using $H(x,\ y) = 0.1[\sin(\pi(x+y))+(x+y) - 3.5]$, and $\eta(x,\ y) = 0.05[\sin(2\pi x)\sin(4\pi y)+ 5]$, respectively.

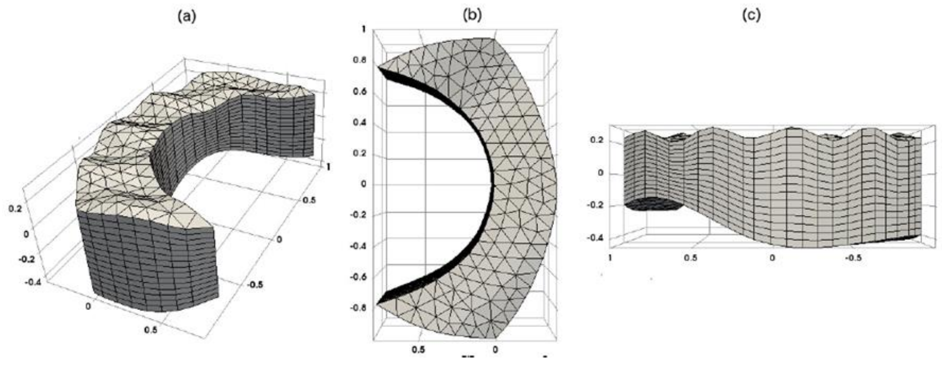

Fig. 5. Domain with irregular free surface and bottom using different viewpoints.

The numerical simulations were performed using different mesh resolutions in the horizontal direction and different divisions of the vertical direction. Details of the horizontal meshes are given in Table 1 and Fig. 6. In this table, we do not include the number of ghost cells added to the domain to deal with the boundary conditions. The finest resolution (Mesh 3 and $Nz = 128$) has approximately 12 million cell-centered points and 6 million vertices.

Table 1. Size of the 3D control volumes used in the numerical experiments.

	Vertex				Cell-centers			
		Vertical (N_Z)				Vertical (N_Z - 1)		
Mesh	2D	32	64	128	2D	31	63	127
1	526	16,832	33,664	67,328	922	28,582	58,086	117,094
2	12,246	391,872	783,744	1,567,488	23,801	737,831	1,499,463	3,022,727
3	48,050	1,537,600	3,075,200	6,150,400	94,699	2,935,669	5,966,037	12,026,773

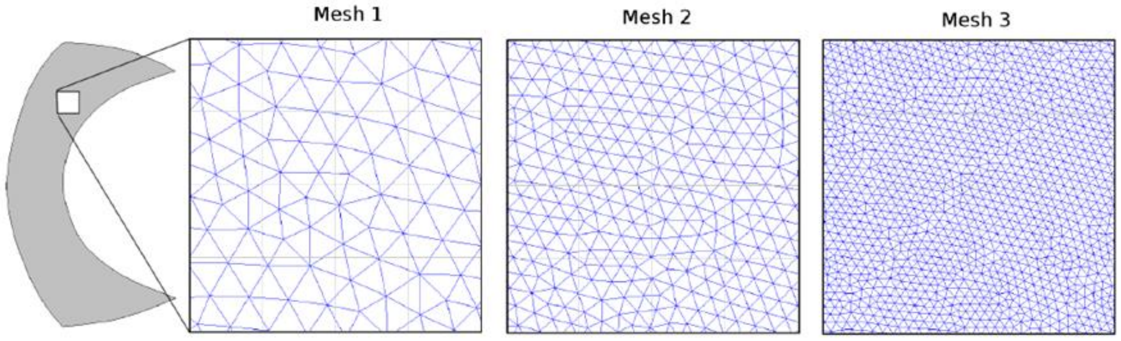

Fig. 6. Different mesh resolutions used in the horizontal coordinate.

The FV approximation, the analytical solution and the absolute errors of the solution of the Poisson equation using Mesh 1 and Nz = 32 are shown in Fig. 7. The numerical results show that the proposed method approximates very well the solution. Precise results are also obtained using different mesh resolutions.

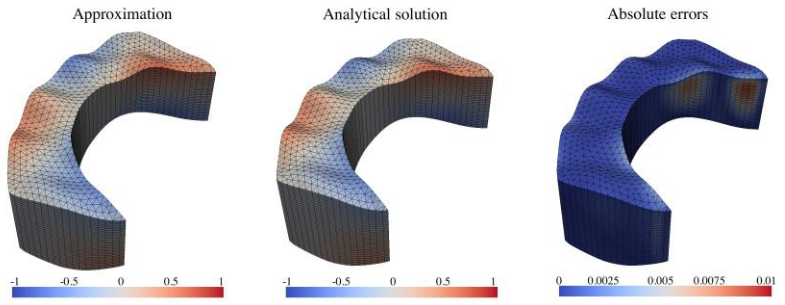

Fig. 7. Numerical approximation, analytical solution and absolute errors for the solution of the Poisson equation using Mesh 1 and Nz = 32.

Table 2 shows the comparison between the analytical solution, Jacobi iterative solver proposed in this paper. In these simulations, the analysis of the error have been performed using the L^{∞}-norm and L^2-norm [14]. For comparison, the sequential SOR method (with relaxation parameter ω = 1.1) is also calculated. The number of iterations and simulation time is also displayed in Table 2. It is important to remark that exactly the same errors were obtained for both sequential and parallel programs. It can be noticed that the accuracy increases as finer meshes are used. However, the number of iterations increases as the number of control volumes of the unstructured grid is increased; consequently, more simulation time is required. For example, the Jacobi simulation with the finest resolution required 60135.84 seconds (about 17 hours) to converge to the solution. We can also notice in Table 2 that the number of iterations of the Jacobi and SOR method is quite different. As we expected, the Jacobi method takes many more iterations for the same residual. However, the GPU-CUDA parallel version of the Jacobi method works so fast that it becomes attractive for applications.

Table 2. Norms of the error, simulation time and iterations of the different solvers for the solution of the Poisson equation.

		Error		Jacobi		SOR	
Mesh	Nz	L^{∞}-norm	L^2-norm	Time (s)	Iter.	Time (s)	Iter.
1	32	1.02×10^{-2}	1.50×10^{-3}	4.52	3,360	0.97	738
2	64	3.12×10^{-3}	3.08×10^{-4}	1738.82	25,409	726.93	11,288
3	128	1.40×10^{-3}	1.52×10^{-4}	60135.84	107,730	24,625.36	46,971

The performance of the Jacobi method is investigated in terms of the total time simulation and speedup. We have analyzed the behavior of the Jacobi algorithm with respect to the number of threads in each block. The influence of the number of threads in each block is significant but not critical, the best performance is obtained using 2 threads in x dimension and Nz threads in y dimension for each block (2 × Nz threads in each block). Note that this number is lower than the total number of threads allowed in a GPU (512 or 1024 depending on the GPU architecture).

Time and speedup of the Jacobi method for different mesh sizes are shown in Table 3. We can see the same qualitative behavior for each version of our method: the speedup improves with the size of the problem and it is within the same order of magnitude with respect to the Nz chosen. The best results are obtained using Mesh 3 with $Nz = 128$ (the finest mesh). The problem can be solved more than 35× faster than its sequential version. Notice that the speedup of Mesh 2 with $Nz = 128$ is similar to the speedup of Mesh 3 with $Nz = 32$ (~ 18×) because the total number of control volumes in both cases is of the same order.

We analyze the use of texture (CUDA DT) and managed memory (CUDA DTM) separately. The performance improvement obtained with texture memory is significant compared with the use of device global memory, since almost all the values required in the kernel k_JacobiMethod are loaded into read-only texture memory, and these values are accessed for a large number of iterations. Note that the speedup of the problem can be increased by 10 times. On the other hand, when we use managed memory for the total error computation, the results are worse than employing the shared memory for partial reductions and the CPU to reduce the rest of the sum (simulations are not shown here). In contrast, the use of managed memory only to reduce the remainder sum leads to a slight improvement (CUDA DTM).

Table 3. Performance of the GPU-CUDA Jacobi method using different grids.

		Sequential	CUDA D		CUDA DT		CUDA DTM	
Mesh	Nz	Time (s)	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
1	32	4.52	0.60	7.53 ×	0.54	8.37 ×	0.53	8.53 ×
1	64	27.33	3.56	7.67 ×	3.21	8.51 ×	3.19	8.57 ×
1	128	193.37	35.42	5.45 ×	30.75	6.29 ×	30.14	6.42 ×
2	32	743.19	41.83	17.76 ×	26.50	28.04 ×	25.03	29.69 ×
2	64	1,738.82	91.78	18.94 ×	55.30	31.44 ×	53.82	32.31 ×
2	128	5,018.89	277.83	18.06 ×	163.56	30.69 ×	161.47	31.08 ×
3	32	11,734.84	626.34	18.74 ×	384.01	30.56 ×	364.93	32.16 ×
3	64	25,511.24	1276.06	20.00 ×	756.57	33.72 ×	737.09	34.61 ×
3	128	60,135.84	2934.38	20.49 ×	1708.39	35.20 ×	1681.11	35.77 ×

In figure 8 shows the results of the three CUDA parallel versions using Mesh 2 and varying Nz values from 32 to 128. We notice that the vertical size initially is not a significant factor that influences parallel performance. The reason is that the gap between the number of control volumes of two resolutions is not so large. However, if more vertical points are used, then there is a slight improvement in the speedup.

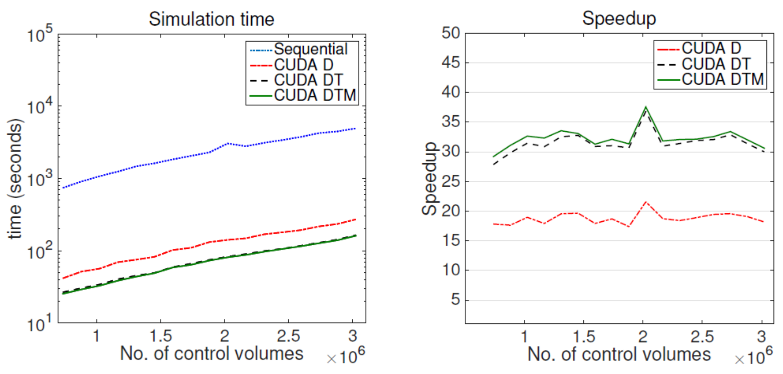

Fig. 8. (a) Simulation times and (b) speedups of the Jacobi method for different CUDA parallel versions.

Although the discussion presented so far reveals the advantages of using the parallel versus the sequential version of the Jacobi method, the proposed parallel algorithm is also competitive with the sequential SOR method. For example using Mesh 3 and Nz $Nz = 128$, the SOR method takes 24,625.36 seconds (almost 7 hours) to perform 46,971 iterations, only the 40% of the Jacobi sequential simulation time; however, the parallel CUDA DTM version takes only 1681.11 seconds (i.e., 14.6× faster than the sequential SOR method).

5 Conclusions

In this paper, we have presented results for parallelized Jacobi iterations applied to solve a finite volume discretization of the Poisson equation. Such a method provides an attractive way to obtain fast simulations for 3D irregular domains. Code written in Fortran was run for both sequential and parallel cases employing a GPU-CUDA code. Although the parallel algorithm is straightforward, GPU memory management was used to maximize the performance of our parallel implementation. The results were tested in different mesh sizes in the horizontal and vertical directions. Results are extremely promising. By numerical experiments, it was found that the final parallel method has a high speedup, being more than 35 times faster than its sequential version in fine grids and more than 15 times faster than the (sequential) SOR method. Further work will involve optimization techniques in the code to obtain faster simulations. The number of iterations of the Jacobi method is reduced by considering the SOR method, but this method is not well-suited to straightforward parallel implementation because residual values depend on the previous iteration. In future work, we plan to develop a GPU parallel implementation of the multi-color SOR method for unstructured grids.

References

1. H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman (2011). High performance computing using MPI and OpenMP on multi-core parallel sys- tems. Parallel Computing, 37(9): 562-575.
2. M. Uh Zapata, D. P. Van Bang and K. D. Nguyen (2016). Parallel SOR methods with a parabolic-diffusion acceleration technique for solving an unstructured-grid Poisson equation on 3D arbitrary geometries. International Journal of Computational Fluid Dynamics, 30(5): 370-385.
3. Y. Notay, A. Napov (2015). A massively parallel solver for discrete Poisson-like problems. Journal of Computational Physics, 281: 237-250.
4. T. Tang, W. Liu and J. M McDonough (2013). Parallelization of linear iterative methods for solving the 3-D pressure Poisson equation using various programming languages. Procedia Engineering, 61:136-143.
5. O. B. Fringer, M. Gerritsen, & R. L. Street (2006). An unstructured-grid, finite- volume, non-hydrostatic, parallel coastal ocean simulator. Ocean Modelling, 14(3): 139-173.
6. Y. Sato, T. Hino, K. Ohashi, (2013). Parallelization of an unstructured NavierStokes solver using a multi-color ordering method for OpenMP. Computers & Fluids, 88: 496-509.
7. M. Cheng, G. Wang, H. Hameed Mian, (2014). Reordering of hybrid unstructured grids for an implicit NavierStokes solver based on OpenMP parallelization, Computers & Fluids.
8. M. Su, J.-D. Yu, (2012). A parallel large eddy simulation with unstructured meshes applied to turbulent flow around car side mirror, Computers & Fluids, 55: 24-28.
9. J. Waltz (2004). Parallel adaptive refinement for unsteady flow calculations on 3D unstructured grids. International journal for numerical methods in fluids, 46(1): 37-57.
10. M. Uh Zapata, D. P. Van Bang and K. D. Nguyen (2014). An unstructured finite volume technique for the 3D Poisson equation on arbitrary geometry using a σ- coordinate system. Int J Numer Meth Fluid, 76(10): 611-631.
11. N. A. Phillips (1957). A coordinate system having some special advantages for numerical forecasting, J. Meteor., 14: 184-185.
12. PGI CUDA Fortran Compiler (2016, December 09), https://www.pgroup.com/ resources/cudafortran.htm.
13. N. Wilt (2013). The cuda handbook: A comprehensive guide to gpu programming. Pearson Education.
14. M. Zlhmal (1978). Superconvergence and reduce integration in the finite element method, Math. Comp., 32 (663).