Solving the Linearized Poisson-Boltzmann Equation on GPUs using CUDA

Colmenares, Jose
jose.colmenares@iit.it
Italian Institute of Technology

Ortiz, Jesus
jesus.ortiz@iit.it
Italian Institute of Technology

Decherchi, Sergio
sergio.decherchi@iit.it
Italian Institute of Technology

Rocchia, Walter
walter.rocchia@iit.it
Italian Institute of Technology

March 20, 2013

1 INTRODUCTION

The Poisson-Boltzmann equation (PBE) is a widely used tool to estimate the electrostatic energy of molecular systems in ionic solution [5]. This last decade experienced a rapid growth of available structural data concerning proteins and other biological macromolecules as well as the availability of new and affordable computational architectures that can be extremely useful in treating increasingly larger and more complex systems [4] as well as in collecting and analyzing electrostatic features throughout structural databases such as the Protein Data Bank [7,11]. In this context, we present an implementation of a linearized PBE solver based on a Finite-Difference scheme on a GPU architecture. Even though there are GPU implementations of the linearized PBE solution (see for example [13]), and suggestions on how to port a PBE solver on GPU [1], to our knowledge a full description of a finite difference (FD) implementation followed by integration and comparison with a widely used tool is lacking in the literature. We follow the approach of the DelPhi PBE solver [8, 10], which exploits the checkerboard structure of the FD discretization of the Laplace differential operator and adopts a Successive Over Relaxation (SOR) scheme to converge to the solution.

We implemented the program on a NVIDIA GPU using the CUDA programming language. CUDA enables C-style programming and gives access to

the GPU computing power, simplifying the process of parallel programming [2]. The code in CUDA is written in kernels that are executed in parallel on the GPU. The kernels are executed in blocks of threads, and each block is executed on a grid. The distribution of blocks and threads is configurable in 1D, 2D and 3D. The threads belonging to the same block have access to the same shared memory, a sort of cache memory, with lower latency than the global device memory. We finally interfaced our implementation with the DelPhi solver and tested the achieved performance on a set of proteins of different sizes, so to assess the efficiency of our solution.

1.1 The sequential Poisson-Boltzmann solver

Under the approximation of continuous dielectric media, the PBE describes the electrostatic potential generated by fixed charges in a solute surrounded by an aqueous solution containing a thermally averaged ionic distribution. In its linearized form, which is valid for low salt concentration, it has the following form:

$$\nabla \cdot [\epsilon(\mathbf{x}) \nabla \Phi(\mathbf{x})] + \frac{\rho^{fixed}}{\epsilon_0} = \frac{1}{\lambda^2} \Phi(\mathbf{x}) \tag{1}$$

where Φ refers to the electrostatic potential, $\epsilon(\mathbf{x})$ is the space-varying relative dielectric constant, ϵ_0 is the permittivity of vacuum, ρ^{fixed} is the fixed charge density on the solute, $\mathbf{x}$ is the position, and λ is the Debye length of the ionic solution, a quantity describing the electrostatic screening made by the ionic cloud in the solution [9]. The right hand side of equation (1) is present only if $\mathbf{x}$ is located in the ionic solution. Our implementation uses the same FD algorithm described in [8]. The sequential version is already fast and has been successfully parallelized using the MPI paradigm with remarkable results [6]. The PBE can be discretized on a uniform grid as follows:

$$\left[\sum_{i=1}^{6} \epsilon_i + \left(\tfrac{h}{\lambda}\right)^2 \right] \Phi_j - \sum_{i=1}^{6} \epsilon_i \Phi_i - \frac{q_j}{\epsilon_0 h} = 0 \tag{2}$$

where Φ_j refers to the electrostatic potential at the node j, which has assigned a net charge q_j. The λ containing term is present only if the node j belongs to the solvent and ϵ_i is the relative dielectric constant at one of the midpoints between the node j and its six nearest neighbors on the grid, h is the grid spacing. This discretized relationship can be used to build a linear system of equations $A\mathbf{\Phi} = \mathbf{b}$ where a suitable mapping converting three dimensional to one dimensional indices has to be adopted. The matrix A can then be decomposed into $A = D + L + U$, where D is the diagonal of A, U and L are the strict upper and lower triangular parts of A, respectively. According to the Successive Over-Relaxation method, the iterative equation is given by:

$$\mathbf{\Phi}^{(n+1)} = (D + \omega L)^{-1} \left\{ \omega \mathbf{b} - [\omega U + (\omega - 1) D] \, \mathbf{\Phi}^{(n)} \right\} \tag{3}$$

where ω is the over-relaxation factor and bracketed superscripts indicate iteration number. The term $(D + \omega L)^{-1}$ can be calculated using forward substitution since $D + \omega L$ is a lower triangular matrix implying that the iterative

scheme must be consistent with the previously described mapping, which makes parallelization difficult. The iteration stencil becomes:

$$\Phi_j^{(n+1)} = \omega \left(\frac{\sum_{i=1}^{6} \epsilon_i \Phi_i^{(n)} + \frac{q_j}{\epsilon_0 h}}{\sum_{i=1}^{6} \epsilon_i + \left(\frac{h}{\lambda}\right)^2} \right) + (1-\omega)\, \Phi_j^{(n)} \tag{4}$$

The best over-relaxation factor can be obtained from the highest eigenvalue of the iteration matrix [12], which in turn can be calculated using the Connected-Moments Expansion [8].
The following observations helped us significantly increase the efficiency of the algorithm. First, the PBE discretized according to the FD scheme is endowed with the so called checkerboard structure. All even grid points depend only on their neighboring grid points, which are odd, and *vice versa*; this means that we can iterate alternately on grid points of different parity until convergence. Due to this property, we can break the order required by the SOR formula and apply the parallelism inside each of the two steps. Second, it is worth pointing out that in most points there are no charges and ϵ_i is independent on i. In fact, ϵ varies only in points close or at the molecular surface. In this way the stencil effectively becomes:

$$\Phi_j = \frac{\sum_{i=1}^{6} \Phi_i}{6 + \kappa_j^2} \tag{5}$$

where

$$\kappa_j = \begin{cases} \left(\frac{h}{\lambda}\right) & \text{if } j \text{ is inside the ionic solution,} \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

which allows for a fast parallelization. After each run of this uniform stencil corrections have to be made at the, few, points where charges are present and where ϵ_i changes. This solution is faster than using the full non uniform stencil everywhere.

2 IMPLEMENTATION

We have implemented the algorithm both in the CPU and in the GPU. In figure 1 we can see the flow diagram of the computation. At the beginning of the calculation there are some pre-processing steps that run always in the CPU (with blue background). These steps are:

- **Determine inside/outside**. Here we determine which grid points are on the solute or in the solvent. This involves calculating the molecular surface of the solute (see [3] for a summary of the different possibilities), and store a flag related to its position. If there is salt in the solution we also calculate the κ factor.
- **Find dielectric boundaries**. In this block we look for the midpoints in which ϵ_i varies. That's equivalent to find the grid points closest to the dielectric boundary. Once the boundary grid points are found, the corresponding entry has to be corrected at each iteration.

- **Set boundary conditions**. This block sets up the boundary conditions to be used. See [9] for a description of the possibilities.

- **Prepare charges correction**. We calculate the correction to be made at the grid points where charges have been assigned.

If we are running the program in GPU mode, the next step is the initialization of the graphic card. Here we allocate the necessary memory on the device and transfer the initial system state and all the necessary information to start the calculation loop: phi (even and odd), charges (position and value), dielectric boundaries (position, factors and temporary vectors for convergence test) and the *solvent* flag (only when there is salt in the solution).

After that, we enter in the main loop, where we update alternatively the odd and even points of the grid. The computation blocks of the main loop are programmed for both CPU and GPU (blocks with orange background). In this way we can run the whole algorithm on the GPU and reduce to the minimum the data transfer between the host (main memory of the computer) and the device (global memory of the graphic card). The only memory transfer in the main loop is carried out every several iterations (typically set to 10) to test the convergence. The steps of the main loop are the following:

- **Save dielectric boundaries**. This block saves the state of the dielectric boundary points to a temporary vector.

- **Run Poisson or Poisson-Boltzmann**. This is the main calculation block. The CUDA implementation is detailed in the subsections 2.1 and 2.2.

- **Adjust dielectric boundaries**. In this block we update the potential value of the grid points located at the dielectric boundary, this is done at the end of each iteration. In the GPU implementation, the position of the boundary points is stored in a linear integer texture. And the 6 factors are stored in 3 floating point textures with two components each. Each thread accesses one dielectric boundary point corresponding to its unique thread ID.

- **Add charges**. Here we add the charge terms to the grid points that were predefined as "charged". Similarly to the previous calculation block, in the GPU implementation, the position of the charges is stored in a linear integer texture and the value of the charges is stored in a linear floating point texture.

- **Calculate potential difference at the dielectric boundary**. Here we calculate the absolute differences between the current potential values at the dielectric boundary with the one saved previously on a temporary data structure. This is done since the boundary is the region where the convergence is expected to be slower.

- **Copy differences from device to host**. The absolute differences calculated before are transferred entirely from the device to the host memory.

- **Check convergence**. The maximum absolute difference is compared to the threshold to test the convergence and decide whether to stop the iterative procedure. This operation is done always in the CPU.

- **Copy the final solution from device to host**. Once the convergence is reached, we transfer the matrix of the calculated potential phi from the device to the host memory.

We programmed two different CUDA kernels to implement separately Poisson and Poisson-Boltzmann equations; depending on the specific case, we apply the corresponding one, however both of them share most of the optimizations.

2.1 Poisson calculation

Poisson calculation is based on a Laplace iteration corrected for local charge and dielectric variation. When we use the checkerboard structure to solve Laplace equation, each point of the grid uses the 6 neighbors of the opposite parity. In other words, we can say that each point is used 6 times (except for the points at the boundary of the grid). In our solution, we try to minimize the number of memory accesses to increase the speed of the calculation. For that reason, the target is to read each point only once and use it 6 times. The practical way to do this is to copy the points from the global device memory to the shared memory. The shared memory is faster than the global memory but it is shared only between the threads from the same block and its size amounts only to $16kB$.

The solution adopted in this work makes use of a bi-dimensional distribution of threads and blocks, so as that each thread calculates a block of points of the grid, corresponding to the third dimension. In the figure 2 we can see the bi-dimensional distribution. The continuous lines are the threads that are actually computing points of the grid, while the dotted lines mark the contour points that are not updated during the calculation. The size of the blocks is Bx and By. The X_s and Y_s coordinates correspond to the local coordinates of the thread inside its block that match the coordinates within the shared memory. The X and Y coordinates are the global coordinates on the grid.

As we can see in figure 2 the grid (shaded in gray) could be smaller than the blocks of threads. This is because all of these latter must have the same dimension and it is not always possible to fit them into the grid dimension. Due to this fact, there will be some threads with a global coordinate outside the grid (that is $X > Nx$ or $Y > Ny$). These threads will terminate their execution without performing any calculation. As it can be expected, the performance is affected by the number of threads of this type, but their impact decreases with the size of the problem, since the percentage of discarded threads decreases with the grid size. In principle, we could choose a different block size to reduce or even eliminate this problem, but we would then hit an even bigger problem, because we would be calculating with a suboptimal block size. The occupancy of the GPU is an important factor, so bigger blocks usually are to be preferred. The performance also increases with multiples of 32 threads, because the scheduler of the GPU allocates the threads in groups of 32, called warps. In our implementation, a block size of 16x16 was found to provide the best results.

The coordinates are calculated in the following way:

$$\begin{cases} X_s = threadIdx.x \\ Y_s = threadIdx.y \end{cases} \tag{7}$$

$$\begin{cases} X = threadIdx.x + blockIdx.x \cdot (blockDim.x - 2) \\ Y = threadIdx.y + blockIdx.y \cdot (blockDim.y - 2) \end{cases} \tag{8}$$

where (following the CUDA notation) $threadIdx$ are the 2D coordinates of the thread inside the block, $blockIdx$ are the 2D coordinates of the block inside the grid and $blockDim$ is the dimension of the block.

In this way each thread has assigned different X and Y coordinates, and they will iterate through the Z coordinate, skipping the points of opposite parity. Since we are calculating independently even and odd points, the starting Z coordinate will depend on the calculation parity and the X and Y coordinates as shown in table 1 (we skip the $Z = 0$ point because it corresponds to the grid boundary).

Table 1: Calculation of the initial Z coordinate

$X + Y$	Z_{even}	Z_{odd}
Even	1	2
Odd	2	1

With the three coordinates X, Y and Z we can calculate the index in the linear buffer were we stored the potential (phi) grid, as shown in equation (9). This equation is the same for even and odd points:

$$index = \left\lfloor \frac{X + Y \cdot Nx + Z \cdot Nx \cdot Ny}{2} \right\rfloor \tag{9}$$

Each thread, including those that are in the border area, copies the value of the grid point with opposite parity that has the same index to the shared memory. After that, all the threads of the same block synchronize to be sure that the shared memory is updated for all of them. Then, the interior points update their values following the Laplace rule. The coordinates of the six neighbors, codified in linear index rather than in 3D coordinates, are shown in tables 2 and 3 for even and odd points respectively. While applying the Laplace stencil, the *Left*, *Right*, *Back* and *Front* points are already in the shared memory, so we use the corresponding offset in the shared space. However, *Bottom* and *Top* points are missing, so we have to read them from the global memory using the offset in the global representation. Since we are iterating in the Z coordinate inside the thread, we can reuse the *Top* information since it corresponds to the *Bottom* point of the next Z coordinate. So, in each iteration we rewrite the *Bottom* point with the previous *Top* point and we read a new value with the indicated offset.

2.2 Linearized Poisson-Boltzmann calculation

The implementation of the Boltzmann term only requires considering the κ term in the updating rule (5) for points that lay within the solvent. Basically, there are two alternative ways to implement this algorithm. One is passing one value per point, and thus practically duplicating the use of memory. The second is

Table 2: Neighbor offsets for even points

Neighbor	Offset (global space)	Offset (shared space)
Left ($-X$)	-1	-1
Right ($+X$)	$+1$	$+1$
Back ($-Y$)	$-Nx$	$-Bx$
Front ($+Y$)	$+Nx$	$+Bx$
Bottom ($-Z$)	$-Nx \cdot Ny/2 - 1$	Not available
Top ($+Z$)	$Nx \cdot Ny/2$	Not available

Table 3: Neighbor offsets for odd points

Neighbor	Offset (global space)	Offset (shared space)
Left ($-X$)	-1	-1
Right ($+X$)	$+1$	$+1$
Back ($-Y$)	$-Nx$	$-Bx$
Front ($+Y$)	$+Nx$	$+Bx$
Bottom ($-Z$)	$-Nx \cdot Ny/2$	Not available
Top ($+Z$)	$Nx \cdot Ny/2 + 1$	Not available

passing just a flag stating whether in the corresponding updating rule κ is equal to zero or not. In this latter case, we introduce a branch division in the kernel. In the tests, the second option was found to be faster, besides the advantage of memory saving. As we will see in the next section, most of the gain comes from the reduction of the data transfer. In the first case we need to read a double precision floating point value (8 bytes) and in the second just a single byte.

References

[1] CUDA Lab Course Reference Manual 2011. `http://num.math.uni-goettingen.de/~stkramer/doc/autogen/CUDA_HPC_Praktikum/step_12.html#AdaptionofPB_ModeltoCUDA`.

[2] NVIDIA CUDA C Programming Guide. `http://www.nvidia.com/`.

[3] Sergio Decherchi, Jose Colmenares, Chiara E. Catalano, Michela Spagnuolo, Emil Alexov, and Walter Rocchia. Between algorithm and model: different molecular surface definitions for the poisson-boltzmann based electrostatic characterization of biomolecules in solution. *Commun. Comput. Phys.*, 13:61–89, 2012.

[4] Lucy R. Forrest and Thomas B. Woolf. Discrimination of native loop conformations in membrane proteins: decoy library design and evaluation of

effective energy scoring functions. *PROTEINS: Structure, Function, and Genetics*, 52:492–509, 2003.

[5] Pawel Grochowski and Joanna Trylska. Continuum molecular electrostatics, salt effects, and counterion binding–review of the poisson-boltzmann theory and its modifications. *Biopolymers*, 89:93–113, 2007.

[6] Chuan Li, Lin Li, Jie Zhang, and Emil Alexov. Highly efficient and exact method for parallelization of grid-based algorithms and its implementation in delphi. *Journal of Computational Chemistry*, 2012.

[7] G. Neshich, W. Rocchia, A. L. Mancini, M. E. B. Yamagishi, P. R. Kuser, R. Fileto, C. Baudet, I. P. Pinto, A. J. Montagner, J. F. Palandrani, J. N. Krauchenco, M. C. Vianna, S. Souza, R. C. Togawa, and R. H. Higa. Javaprotein dossier: a novel web-based data visualization tool for comprehensive analysis of protein structure. *Nucleic Acids Research*, 32:W595–W601, 2004.

[8] A. Nicholls and B. Honig. A rapid finite difference algorithm, utilizing successive over-relaxation to solve the poisson-boltzmann equation. *Journal of Computational Chemistry*, 12:435–445, 1991.

[9] Walter Rocchia. Poisson-boltzmann equation boundary conditions for biological applications. *Mathematical and Computer Modelling*, pages 1109–1118, 2005.

[10] Walter Rocchia, Emil Alexov, and Barry Honig. Extending the applicability of the nonlinear poisson-boltzmann equation: Multiple dielectric constants and multivalent ions. *J. Phys. Chem. B*, 105:6507–6514, 2001.

[11] Walter Rocchia and Goran Neshich. Electrostatic potential calculation for biomolecules creating a database of pre-calculated values reported on a per residue basis for all pdb protein structures. *Genet. Mol. Res.*, 6:923–936, 2007.

[12] J. Stoer and R. Bulirsch. *Numerical Mathematics*. Springer, 2002.

[13] Rio Yokota, J.P. Bardhan, M. G. Knepley, L. A. Barba, and T. Hamada. Biomolecular electrostatics using a fast multipole bem on up to 512 gpus and a billion unknowns. *Comput. Phys.*, pages 1271–1283, 2011.

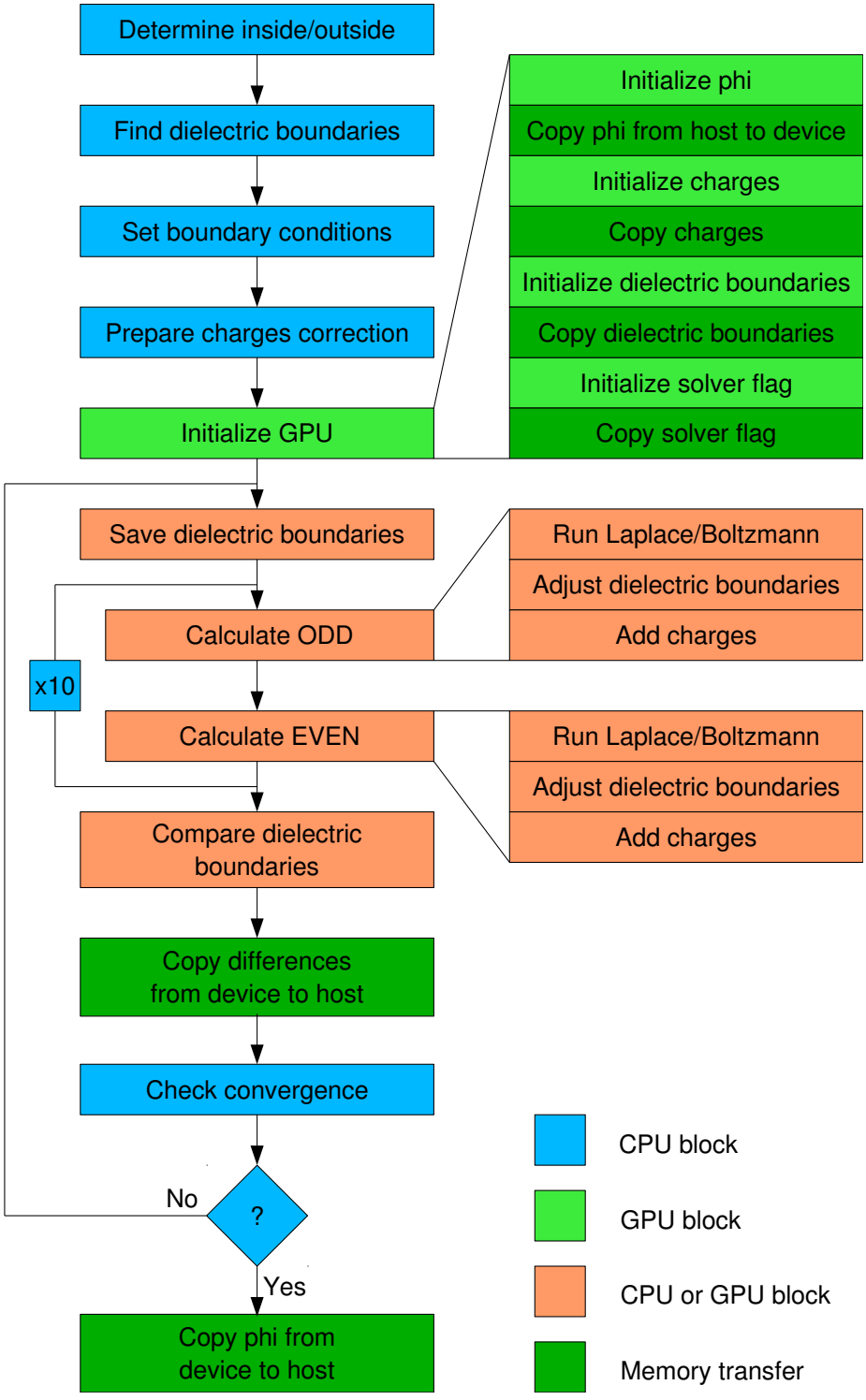

Figure 1: Computation flow diagram

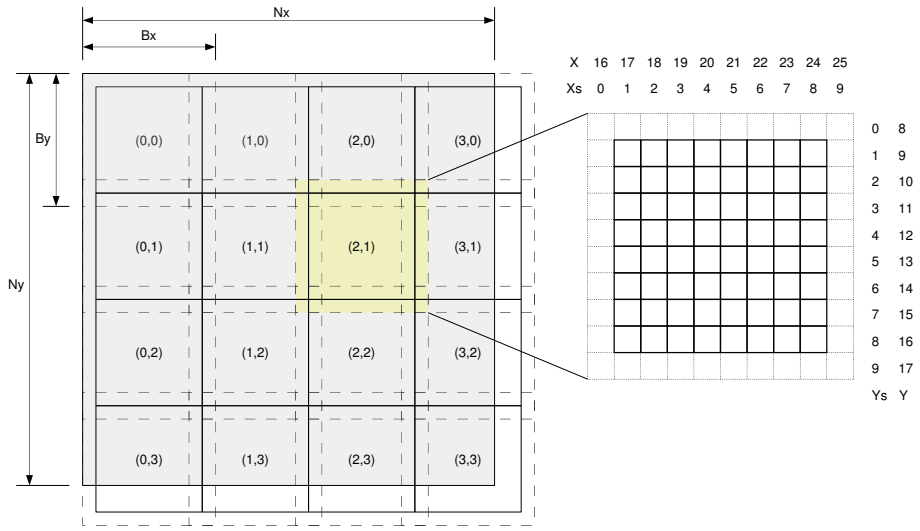

Figure 2: *Tartan* distribution of the blocks of threads in CUDA