Fast Binding Site Mapping using GPUs and CUDA[†*]

Bharat Sukhwani Martin C. Herbordt

Computer Architecture and Automated Design Laboratory
Department of Electrical and Computer Engineering
Boston University, Boston, MA 02215

Abstract*—*Binding site mapping **refers to the computational prediction of the regions on a protein surface that are likely to bind a small molecule with high affinity. The process involves flexibly docking a variety of small molecule probes and finding a consensus site that binds most of those probes. Due to the computational complexity of flexible docking, the process is often split into two steps: the first performs rigid docking between the protein and the probe; the second models the side chain flexibility by energy-minimizing the (few thousand) top scoring protein-probe complexes generated by the first step. Both these steps are computationally very expensive, requiring many hours of runtime per probe on a serial CPU. In the current article, we accelerate a production mapping software program using NVIDIA GPUs. We accelerate both the rigid-docking and the energy minimization steps of the program. The result is a 30x speedup on rigid docking and 12x on energy minimization, resulting in a 13x overall speedup over the current single core implementation.**

I. INTRODUCTION

Discovering a new drug involves finding a site on a given protein that will bind a small molecule inhibitor with high affinity. This involves docking the candidate inhibitor to the protein, often requiring exhaustive 3D search. Moreover, drug discovery also requires finding the appropriate small molecule inhibitor, or the ligand, that will bind to that site and alter the function of the protein, thus curing the disease. Thus, discovering a new drug involves docking-based screening of millions of candidate ligands for a given protein, requiring many hours to days of CPU runtime.

An important observation, however, is that certain regions on a protein surface, called "hotspots", are major contributors to the total binding energy between the protein and the ligand, and that they bind a wide variety of small molecule probes [2][11]. Thus, a hotspot on a protein surface can be found by docking some number of small molecule probes and finding a consensus region that binds most of these probes with high affinity. This process is called *binding site mapping*. The advantage of this scheme is that the likely binding site on a protein surface can be found independent of the actual ligand. During drug-screening then, each ligand can be docked on this local region or the hotspot, without having to search the entire 3D space. This reduces the screening time significantly, enabling faster drug discovery. Though the identification of hotspots is also possible with experimental methods such as NMR or X-ray crystallography, such methods are very expensive and computational methods are explored as more cost-effective alternatives.

Mapping of binding sites is a computationally expensive process, requiring many hours of runtime on a single CPU. In the current article, we present the GPU based acceleration of a production mapping code called FTMap [2]. FTMap employs a complex rigid docking routine, followed by CHARMM-potential based minimization of few thousand top scoring docked conformations. On a single processor core, FTMap typically requires around 18 hours to finish mapping of a protein. FTMap is a production mapping code, with a web-based server setup for free public use. Currently, it runs on a 1024 node IBM Blue Gene cluster. In the current article, we present a more cost effective, desktop alternative to the cluster implementation, with potential application as the backend for the web-server on a GPU based cluster.

We present acceleration of both the rigid docking and the energy minimization steps. In our previous work, we have published acceleration of a rigid docking program using GPUs [16] and preliminary results on the acceleration of electrostatics energy computation for energy minimization [17]. Here, we extend the acceleration of energy minimization to include the van der Waals energy evaluation on GPUs. Though the energy minimization uses similar force fields as the widely studied molecular dynamics simulation (MD), the underlying problem geometry is very different and hence the acceleration techniques used in MD are mostly not applicable here.

Parallelization and acceleration of energy minimization is difficult due to the very small amount of computation performed per iteration and the large fraction that is serial accumulations. Most parallel accumulation schemes on GPUs require large amounts of data communication, leading to poor overall performance. We address this by changing the underlying data-structures and statically mapping the work on GPU threads in a way that allows parallel energy evaluations and fast, parallel accumulations. In this work, we also integrate the accelerated energy minimization with

[†]This work was supported in part by the NIH through award #R01-RR023168-01A1. Web: www.bu.edu/caadlab. Email: {herbordt|bharats} @bu.edu

* For more details, please see TR2010_1 at www.bu.edu/caadlab/publications.html

accelerated rigid docking to enable fast mapping on a desktop class workstation. We achieve a factor of 32x speedup on the rigid docking step and 12.5x on the energy minimization step, resulting in the overall speedup of 13x of the FTMap program.

II. BINDING SITE MAPPING

Computational mapping refers to the process of finding druggable binding-sites on the surface of a protein. Such binding sites, or "hotspots", are regions that bind inhibitor molecules with high affinity. The process involves flexibly docking a wide variety of small molecule probes to a given protein and finding the consensus region that binds most of these probes with high affinity. Due to the computational complexity of flexible docking, the mapping task is usually performed in two steps. The first step assumes the interacting molecules to be rigid and performs exhaustive 3D search to find the best pocket on the protein that can fit the probe. This step is called rigid docking. The top scoring conformations from this step are saved for further evaluation in the second step.

The second step models the flexibility in the side chains of the probes by allowing them to move freely and minimizing the energy between the protein-probe complex. This is an iterative process wherein the side chains are progressively moved towards the least energy conformation. This is often referred to as CHARMM-potential minimization or simply *energy minimization*. The FTMap program also follows the two step approach just described.

A. *Rigid Docking Using PIPER*

The rigid docking step aims at finding a pocket on the protein surface that can fit the small molecule. It follows the lock-and-key model (see Figure 1), wherein the two interacting molecules are considered to be rigid. The task is to find the relative offset and rotation (pose) of one molecule with respect to the other that results in the strongest interaction between the two molecules. In addition to the geometries of the two molecules, various other energy functions, such as electrostatics and desolvation, are modeled to determine the strength of the interaction between the two molecules in a given orientation.

FTMap performs the rigid docking step using a program called PIPER [10]. Like many other rigid docking programs, PIPER maps the surface and other properties of the two interacting proteins onto 3D grids. Exhaustive 3D search is performed by rotating one of the grids by an incremental angle and translating the grid with respect to the other along the 3 axes.

The score of a pose (a rotation and a relative translation α, β, γ of the small molecule relative to the protein) is computed as a 3D correlation sum between the two grids

$$E(\alpha,\beta,\gamma)=\sum_{p}\sum_{i,j,k}R_P(i,j,k)L_P(i+\alpha,j+\beta,k+\gamma) \quad (1)$$

where $R_p(i,j,k)$ and $L_p(i+\alpha, j+\beta, k+\gamma)$ are the components of the correlation function defined on the protein and the small molecule, respectively.

Thus for each rotation, $O(N^3)$ translations are performed, each requiring $O(N^3)$ computations. These are performed using FFT, reducing the complexity for each rotation to $O(N^3 \log N)$. By default, PIPER evaluates tens of thousands of rotations, typically requiring many hours of CPU time. To limit the computation requirements, FTMap performs rotation at a higher granularity of incremental angle, performing a total of 500 rotations. This results in about 30 minutes of serial runtime per probe for the rigid docking phase. From each rotation, the 4 top scoring poses (relative translations) are retained for the energy minimization phase.

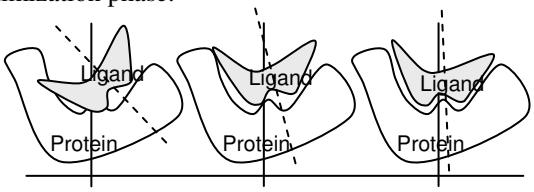

Figure 1. Lock-and-Key Model for Rigid Docking

The scoring function used in PIPER is based on three criteria: shape complementarity, electrostatic energy, and desolvation energy. The total pose score is computed as a weighted sum of these three energy functions (Equation 2).

$$E = E_{shape} + w_2 E_{elec} + w_3 E_{desol} \quad (2)$$

Both the shape complementarity and the electrostatics terms are computed as a weighted sum of two components each and the desolvation energy is computed as a sum of 4 to 18 pairwise potential terms. Computing scores of each of these terms requires independent correlation sums, leading to up to 22 FFT correlations per rotation.

For every rotation, PIPER computes the ligand energy function *Lp* on the grid and performs repeated FFT correlations to compute the scores for the different energy functions. For each pose, these energy functions are combined to obtain the overall energy for that pose. Finally, a filtering step returns some number of poses per rotation based on score and distribution.

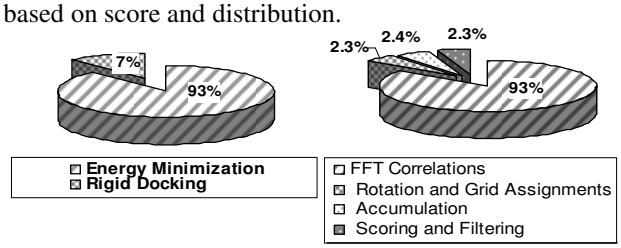

Figure 2. (a) Profiling of FTMap program, (b) Runtimes per rotation for different steps in rigid docking

The distribution of time per rotation for different steps of PIPER rigid-docking phase is shown in Figure 2 (b). Clearly, the most time consuming step is FFT correlation, requiring about 93% of the time. Of the remaining, almost 5% is spent in accumulation of pairwise potential terms for desolvation energy and scoring and filtering. In our GPU accelerated PIPER, we accelerate all these per rotation steps except rotation and grid assignment. As discussed later, the FFT correlation step is replaced with direct correlation.

B. FTMap Energy Minimization

In FTMap, the rigid docking step is followed by the energy minimization of the top scoring conformations. From each rotation of the rigid docking phase, FTMap retains 4 top scoring conformations. This results in 2000 conformations to be minimized per probe, with typical runtimes of many hours per probe. With 16 probes to be mapped, the energy minimization phase is clearly the more time consuming step of the mapping task. As shown in Figure 2 (a), energy minimization step constitutes about 93% of total FTMap runtime.

Energy minimization is an iterative process which aims at computing the configuration of the atoms in a complex that corresponds to the minimum potential energy [7]. It involves computing the potential energy of the complex at a point, updating the forces acting on the atoms, and adjusting the atom-coordinates according to the total forces acting on them. This process of energy evaluation and of force and position updates is repeated for many iterations until the energy of the system converges to within a threshold.

Note that though energy minimization superficially seems similar to the widely studied molecular dynamics (MD) simulations, the underlying geometry of the problem and the computational structures are quite different. Unlike MD, energy minimization is a refinement step and is performed on a local region of the protein surface and the motions are very small. Due to this, the filtering techniques employed in MD, e.g. cell lists, are not employed in energy minimization. Moreover, even though energy minimization, like MD, uses neighbor-lists, they are seldom updated.

In energy minimization, the system to be simulated consists of a number of atoms; the total energy of the system is a sum of various bonded and non-bonded energies of all the atoms (Equation 3).

$$E^{total} = \underbrace{E^{vdw} + E^{elec}}_{non\text{-}bonded} + \underbrace{E^{bond} + E^{angle} + E^{torsion} + E^{improper}}_{bonded} \quad (3)$$

Minimization involves repeated evaluation of this expression, once during each iteration. Figure 3 shows the profiling results for the energy minimization step of the FTMap program. As shown in Figure 3 (a), most of the minimization runtime is spent in evaluating these energy terms and the forces. Of these, the non-bonded energy evaluation step is the most computationally intensive, requiring more than 99% of total energy evaluation time (Figure 3 (b)). This includes the electrostatics and the van der Waals energies.

The non-bonded energy of each atom is the sum of the contributions due to neighboring atoms within a cutoff distance. The total non-bonded energy of the system is the sum of the non-bonded energies of all the atoms. In the current article, we accelerate the evaluation and accumulation of these non-bonded energies and the corresponding force calculations by mapping these computations on a GPU. Bonded energy evaluation is a small fraction of the total runtime and is left to be executed on the host.

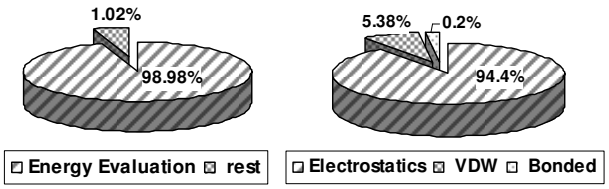

Figure 3: (a) Profiling of Energy Minimization step of Serial FTMap program, (b) Distribution of the energy evaluation time.

As stated earlier, the non-bonded energy is a sum of the electrostatic and van der Waals energy terms. The electrostatic energy of a solute with N charges can be decomposed into two components; a sum of the self energies E_i^{self} of all the charges and a sum of pairwise interaction energies E_{ij}^{int} [13]. (see Equation 4).

$$E^{elec} = \sum_i E_i^{self} + \sum_{i<j} E_{ij}^{\text{int}} \quad (4)$$

For the computation of the electrostatic energy, FTMap employs the Analytic Continuum Electrostatics (ACE) model [13], wherein the self-energy of an atom is represented as a sum of its Born self-energy in the solvent and the sum of effective pairwise interactions, E_{ik}^{self}, due to all the other solute atoms (see Equation 5) [13].

$$E_i^{self} = \frac{q_i^2}{2\varepsilon_s R_i} + \sum_{k \neq i} E_{ik}^{self} \quad (5)$$

$$E_{ik}^{self} = \frac{\tau q_i^2}{\omega_{ik}} e^{-\left(\frac{r_{ik}^2}{\sigma_{ik}^2}\right)} + \frac{\tau q_i^2 \tilde{V}_k}{8\pi}\left(\frac{r_{ik}^3}{r_{ik}^4 + \mu_{ik}^4}\right)^4 \quad (6)$$

Here q_i represents the charge on atom 'i', r_{ik} is the distance between atoms 'i' and 'k', $\tilde{V}_k$ is the size of the solute volume associated with atom 'k', ω_{ik} and σ_{ik} determine the height and width of the Gaussian that approximates E_{ik}^{self}, and μ_{ik} is an atom-atom parameter.

The pair-wise interaction energy is given by the generalized Born (GB) equation, which is the sum of Coulomb's law in a dielectric and the Born equation [14]:

$$E_{ij}^{\text{int}} = 332\sum_{j\neq i}\frac{q_i q_j}{r_{ij}} - 166\tau\sum_{j\neq i}\frac{q_i q_j}{\sqrt{r_{ij}^2 + \alpha_i\alpha_j e^{-\left(r_{ij}^2/4\alpha_i\alpha_j\right)}}} \quad (7)$$

where α_i and α_j represent the Born radius for atoms 'i' and 'j', respectively. These in turn depend on the self-energy of the atom.

For computing the van der Waals energies of the atoms, FTMap uses a variant of the Lennard-Jones 6-12 potential. This is shown in Equation (8).

$$E_{vdw_{ik}} = eps_{ik}\left(\frac{rm_{ik}^6}{r_{ik}^{12}} - \frac{8\,rm_{ik}^6/r_c^6}{r_{ik}^6} + \frac{rm_{ik}^6}{r_c^{12}}\left[1 + \frac{2r_{ik}^6}{r_c^6}\right]\right) \quad (8)$$

$$eps_{ik} = eps_i . eps_k \quad (9) \qquad rm_{ik} = (rm_i + rm_k)^2 \quad (10)$$

where eps_i and rm_i represent the van der Waals parameters of atom *'i'*, r_{ik} is the distance between the two atoms and r_c is the cut-off distance.

Equations (6), (7) and (8) represent the main computations that need to be performed for all atom-atom pairs to evaluate the total electrostatic and van der Waals energies of a given conformation. In addition, the energy gradients need to be computed to determine the forces acting on the atoms and to update the atom coordinates.

III. RIGID DOCKING ON GPUs

As in the original FTMap software, we split the FTMap on GPUs into two steps: rigid-docking and energy minimization. Here we describe the mapping of rigid-docking on GPUs. Energy minimization on GPUs is discussed in section 4.

As stated earlier, the FTMap program performs rigid docking using PIPER, which computes multiple FFT-correlations to obtain the pose score for different energy functions. Though FFT reduces the computational complexity from $O(N^6)$ to $O(N^3 \log N)$, our prior work on accelerating PIPER using FPGAs[15] and GPUs[16] indicates that, if the ligand grid is smaller than a certain size, direct correlation can perform better than FFT correlation, especially if multiple correlations are to be performed. This is due to many reasons: direct correlation lends itself well to parallelization, multiple correlation scores can be computed together, multiple rotations can be scored in a single pass of the protein grid and large data reuse amortizes the overhead of data fetch and kernel launch. Since the probes used by FTMap are very small, we use direct correlation for docking on GPUs.

A. *Direct Correlation on GPUs*

Direct correlation on a GPU replaces the steps of forward FFT, modulation, and inverse FFT. It translates one of the grids over the other and computes a sum of all the voxel-voxel interactions for each translation. Note that multiple energy functions can be evaluated together for each translation, requiring only a single pass through the grids.

To distribute the task of computing the correlation scores for all the translations along the 3-axes, we represent the task as the 3D result grid that needs to be computed. Here, each grid point represents the correlation score for a translation (or multiples scores, one for each energy function). The distribution of work on different GPU threads can now be seen as distributing the computation of different portions of the result grid across multiple threads and thread blocks. This can be performed in various ways. We tried two different schemes, as shown in Figure 4.

In both schemes, we launch the kernel with a 2D array of thread blocks, each with a 3D array of threads. In the first scheme, each thread block is responsible for computing a part of the 2D result plane for all the 2D planes in the 3D result grid. In the second scheme, we assign different 2D planes to different thread-blocks. The threads on each of those thread blocks compute a larger part of the 2D plane, but only for the planes assigned to the current thread block. Both distributions result in similar runtimes, though one or the other can have better performance for various non-cubic grids.

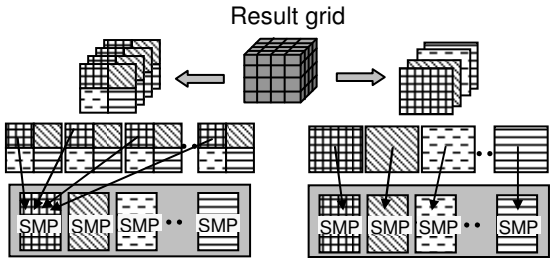

Figure 4. Distribution of work on a GPU for direct correlation.

As the protein and the probe grids are generated on the host, they must be transferred from the host memory to the GPU memory. In the case of the protein grid, this is done only once. The ligand grid, however, is rotated on the host and remapped. Thus, this transfer is required for every rotation. Since every multiprocessor needs access to both grids, they need to be stored either in the device's global memory, accessible by all the multiprocessors, or duplicated in the local shared memory of each of the multiprocessor. Due to the relatively large sizes of the protein grids and the limited amount of shared memory, we store these grids on the global memory. The ligand grids are much smaller, however, and can fit in device's shared memory or constant cache. Both of these provide much faster access compared with global memory. We found that access time from constant memory and shared memory is identical.

Due to the small sizes of the shared and the constant memories, we can fit a ligand grid of size up to 7^3 in shared memory and up to 8^3 in constant memory. Since the probes are never bigger than 4^3 this is not an issue for mapping. The small probe grids, in fact, allow us to perform a further optimization: storing the voxel grids for multiple rotations in the constant memory. This enables the correlation inner loop to compute multiple scores in each iteration.

Storing and evaluating multiple rotations together has two-fold benefits. First, the loop and kernel launch overhead is amortized over multiple rotations. Second, each protein voxel fetched from the global memory (which is not cached) gets reused multiple times, reducing the number of higher latency accesses to the global memory. This results in significant performance improvement. For 4^3–sized probe grids, we can perform 8 rotations in each pass, achieving a speedup of 2.7x over direct correlation performed one rotation at a time.

B. *Scoring and Filtering on GPUs*

Scoring refers to computing the weighted sum of correlation scores for different energy functions and filtering refers to selecting the top scores from different regions on the result map. Filtering is performed by selecting the best score and then excluding its neighbors while selecting the next best score. Such exclusion is done to avoid selecting multiple best scores from the same region (see Figure 5).

Though scoring and filtering amount to a small fraction of total runtime for rigid-docking, it is critical to accelerate

this step to achieve overall good performance. Performing filtering on the GPU has the further advantage of reducing the amount of data that must be transferred back to the host after correlation. After filtering, only the top few scores need to be transferred, as opposed to the entire 3D score grid.

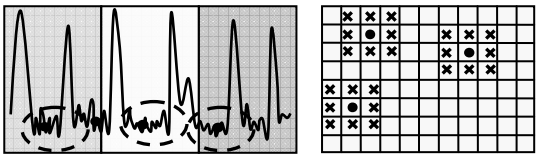

Figure 5. Filtering of top-scores from different regions on the result map. Cells surrounding the selected score are marked for exclusion.

As in the case of computing the correlation score, we divide the N^3 points of the result grid equally among M threads. Each thread computes N^3/M weighted scores, finds the best scores within its subset and stores it in the shared memory (Figure 6). These scores are then gathered by a master thread and the best among these is selected. To simplify this gather process, we distribute the scoring task to threads on only one multiprocessor. Though this is a heavy under-utilization of the available GPU computation power, it simplifies the process of assembling these scores from different threads. Distribution across multiple multiprocessors would incur large communication overhead and nullify any performance benefits achieved from the increased parallelism.

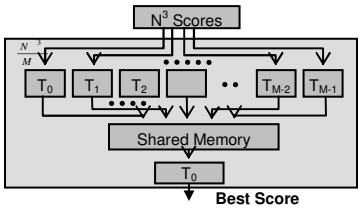

Figure 6. Scoring and Filtering on a GPU. Scores distributed across different threads and accumulated by a master thread.

The master thread (thread 0) performs an additional task of flagging the cells for exclusion. Exclusion is determined by maintaining an array of length N^3, with one entry for each of the cell, for constant time lookup. Since N = 128 is typical, this array does not fit in the GPU shared memory and is stored in the global memory.

IV. ENERGY MINIMIZATION ON GPUs

The computations to be performed per iteration of energy minimization can be divided into six tasks: (i) computing the self-energies for all the atoms, (ii) computing the pairwise interactions energies, (iii) computing the van der Waals energies, (iv) computing the energy gradients (v) updating the forces acting on the atoms, and (vi) performing the optimization move and updating the atom-coordinates based on the force values. To amortize the GPU kernel launch overhead and to reuse the common computations, we divide the six tasks into three GPU kernels: (a) computing atom self energies and the corresponding energy gradients, (b) computing the pairwise interactions (which is a part of the electrostatic energy) and the van der Waals energies along with the energy gradients, and (c) updating the atom forces. The computation structures used by these kernels are similar and the techniques discussed here apply to all these computations. Two computations - the optimization move and the atom-coordinate updates, are left on the host, though in the future we plan on performing these on the GPU as well.

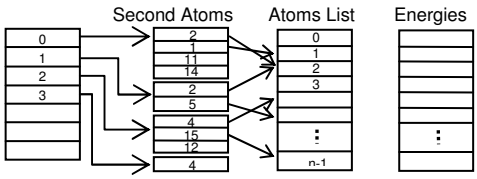

Figure 7. Array of Neighbor-Lists

For efficient computation of atom energies, serial FTMap arranges the atoms in a neighbor list format, where each atom (the "first" atom) has an associated list of neighbors (the "second" atoms) that contribute to its energy (see Figure 7). As the positions of the atoms change, the neighbor lists are updated. The FTMap program cycles through different atom-pairs in the neighbor-list and computes the partial energies. These partial energies are accumulated, as they are computed, into an array storing the total energies of all the atoms. Though there are various ways to map this neighbor-list computation structure onto GPU threads for parallel energy evaluations, most of them run into two serious problems: (i) memory conflicts during parallel updates from different threads and (ii) serialization during the accumulation of the partial energies into the energy array.

There are several reasons why the neighbor-list structure is not suited for mapping to the GPUs. First, we need the individual total self energies of all the atoms, not just the total self energy of the system. This requires multiple accumulations, one for each entry of the energy array. Second due to the random occurrences of the "second" atoms in the neighbor-lists (see Figure 7), the energy array cannot be distributed into the shared memories of different GPU multiprocessors. Rather, it must be present in the GPU global memory, accessible from all the multiprocessors. And third, having the energy array in the global memory can potentially lead to write conflicts, since a particular "second" atom can be present in the neighbor-lists of more than one "first" atom (see Figure 7).

For efficient mapping of these computations onto GPU threads, and to enable fast and parallel energy updates and accumulations, we modified the original neighbor-list into a different data structure: we refer to it as a *pairs-list*. Before we discuss this structure, we briefly describe our initial solution for mapping the original neighbor-lists onto the GPUs.

A. *Mapping Neighbor-lists on GPUs*

To enable parallel updates and accumulations on different GPU multiprocessors, we map only one "first" atom onto a multiprocessor at a time. On each multiprocessor we have two different energy arrays in the

shared memory: one for the partial energies of the current first atom and the second for the partial energies of all the second atoms (Figure 8).

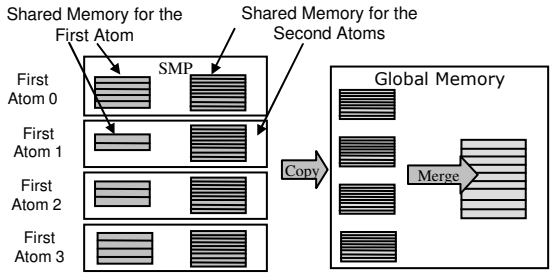

Figure 8. Mapping the Neighbor-lists onto GPU threads. Replicating energy arrays to enable parallel updates.

Different threads of a thread group compute the partial energy of the current atom due to one of the second atoms in its neighbor-list and that of the second atom due to the first. As the energies are computed by different threads, they are updated in these shared memory arrays. Note that since a second atom will appear in the neighbor list of a particular atom only once, no two threads of a particular thread block will update the same shared memory location at the same time. This enables parallel, conflict free updates.

Once all the second atoms of the current first atom are processed, a barrier is reached and a master thread accumulates the partial energy of the first atom by accumulating the values in the first atom energy array. The energies in the second atom array, however, are for different second atoms and are only partial. Analogous partial arrays are present on the shared memories of all the other multiprocessors and must be combined to compute the total energies of the second atoms (see Figure 8). This is done by copying the second atom arrays from the shared memories of the different multiprocessors to the global memory. The corresponding values from these arrays are then added to obtain the total energies.

Though this scheme allows parallel execution and updates, it has three problems. First, since only one first atom is processed by a multiprocessor, the GPU threads are heavily underutilized and the distribution of work on different multiprocessor is uneven. This is because different "first" atoms have widely varying number of "second" atoms in their neighbor-lists, ranging from a few to a few hundred. Second, transferring multiple large second atom arrays from shared to global memory incurs high data transfer cost per iteration. Finally, accumulation from the global memory is slow. Overall this method results in poor performance and is not preferred.

B. Improved Data-Structures for Efficient Mapping on GPUs

Since the computation per iteration is very small, only a few milliseconds on a serial computer, obtaining high speed-up requires efficient distribution of work to maximize parallelism and reduce the communication cost. We now discuss two schemes for doing just that. Both modify the original neighbor-list data structure to enable better distribution of work over GPU threads and more efficient accumulations.

	Atom index		Atom Energy	
Pair Id	Atom 1	Atom 2	Atom 1	Atom 2
0	0	2		
1	0	1		
2	0	11		
3	0	14		
4	1	2		
5	1	5		
6	2	4		
7	2	15		
8	2	12		
9	3	4		

Figure 9. Atom-pairs list

In the first scheme, we replace the neighbor-list structure with a pairs-list. It contains a list of atom-pairs that need to be processed, along with fields to store the partial energies of the two atoms involved in the pair. This is shown in Figure 9. Different atom-pairs are independent of each other and can be processed in parallel. We distribute these pairs equally among different GPU threads. The pairs-list is stored in the GPU global memory. Each thread processes the pair assigned to it and stores the partial energies of the two atoms at the corresponding index in the global memory.

Once all the pairs have been processed, we accumulate these partial energies to compute the total energy of each individual atom. This needs to be done serially, mainly due to the unordered occurrences of the second atoms in the pairs-list.

Since the energy values are stored in the GPU global memory, accumulation requires multiple accesses to the slow GPU global memory. Also, since the accumulation is done serially by a single thread, it turns out that this accumulation is actually faster on the host. Accumulation on the host, however, requires transferring the two arrays of atom-energies from the GPU to the host in each iteration. This scheme thus enables parallel energy computation and updates but still requires serial accumulation of energies. With accumulation on the host, it results in a speedup of around 3x over the original serial code.

To enable faster and parallel accumulations from the GPU shared memory, we further modified the data structure. In our second approach, we still use the pairs-list of Figure 9 but make two changes in how the pairs get mapped to the GPU threads.

The first change is to split the pairs-list into two separate pairs-lists. Notice that the serialization during accumulation is mainly due to the random occurrences of second atoms in the neighbor-lists (now the pairs-list). The first atoms still appear in an ordered fashion. Thus, to add determinism in how the atoms appear in the list, we split the lists into two separate lists and process each one separately.

The first pairs-list is based on the original neighbor-list and is called the forward list. The second list is generated by reversing the original neighbor-list, i.e., by treating each second atom of the original neighbor list as a first atom for the reverse neighbor list. We call this second list the reverse list. While processing a list, only the energy of the first atom in each pair is computed and updated. This way, the energies of the first atoms (in the original list) get updated

while processing the forward list and those of the second atoms (in the original list) while processing the reverse list. This is shown in Figure 10. Note that there is no column for the energies of the second atom in the pair.

	Atom index		Energy
Pair id	Atom 1	Atom 2	Atom 1
0	0	2	
1	0	1	
2	0	11	
3	0	14	
4	1	2	
5	1	5	
6	2	4	
7	2	15	
8	2	12	
9	3	4	

	Atom index		Energy
Pair id	Atom 1	Atom 2	Atom 1
0	1	0	
1	2	0	
2	2	1	
3	4	2	
4	4	3	
5	5	1	
6	11	0	
7	12	2	
8	14	0	
9	15	2	

Figure 10. Split pairs-lists. (Left) Forward list, (Right) Reverse list.

The second modification involves statically mapping the pairs from the new pairs-lists onto the GPU threads. This comes from the observation that the pairs in the new lists can be grouped by the first atoms. This can be done since we now care only about computing the energies of the first atoms in the pair and not the second atoms.

These two changes allow better and more uniform distribution of atom-pairs on the GPU and enable parallel and much faster accumulations in GPU shared memory, as discussed next.

Once we have the forward and reverse pairs lists, we statically distribute them to GPU threads running on different multiprocessors. The static mapping scheme groups together all the pairs in a list having the same first atom and maps the entire group onto the threads in the same thread block. More than one group of pairs can be mapped onto a particular thread block, provided there are enough threads to accommodate all the pairs of those groups. If the current thread block does not have enough threads left to accommodate the entire group, it is mapped onto the next available thread block. Unused spaces on the thread blocks are claimed by other smaller pair-groups. Having all the pairs of a group on the same thread block allows us to perform accumulation in the shared memory, since all the partial energies are present within the same multiprocessor.

To determine the assignment of work for different GPU threads, we generate a new data-structure called the assignment table (see Figure 11). The table contains one row per thread id which contain 5 fields: pair id, indices of the two atoms, a master field indicating if this thread is the first thread for this pairs-group, and the number of pairs in the pair-group. The master thread field and the number of pairs in group are used to accumulate the energies of the atoms in the shared memory.

The table in Figure 11 is stored in the GPU global memory. One table is generated from each of the forward and the reverse pairs-lists and is transferred to the GPU only once at the beginning of the minimization process. There is no further data transfer per iteration, unless the neighbor list is updated, in which case we regenerate the assignment tables and transfer them to the GPU. This happens only a few times per 1000 minimization iterations; thus the transfer time is negligible.

Each thread works on the pair assigned to it in the assignment table. In case the number of pairs is larger than the number of threads, each thread would be responsible for multiple rows. Energies computed by different threads are stored in an array in the GPU shared memory. The length of this array is equal to the number of threads in the thread block, with each thread storing the computed energy at the index equal to its local thread id (id within the block).

	Thread Id	Pair Id	Atom 1	Atom 2	Master	Pairs	
Thread Block 0	0	0	0	2	1	4	Group 0
	1	1	0	1	0	4	
	2	2	0	11	0	4	
	3	3	0	14	0	4	
	4	9	3	4	1	1	Group 3
Thread Block 1	5	4	1	2	1	2	Group 1
	6	5	1	5	0	2	
	7	6	2	4	1	3	Group 2
	8	7	2	15	0	3	
	9	8	2	12	0	3	

Figure 11. Work Assignment Table for the GPU.

Once all the threads have finished processing their assigned pairs, the master threads execute the accumulation round. Each master thread reads the number of atoms for the group associated with it and accumulates that many values from the shared memory, starting from its local thread id. This way, many threads perform accumulation in parallel and from the shared memory, resulting in significant speedup compared to previous schemes. The master threads then store the accumulated values in the GPU global memory. Note that we can use this scheme only because we are computing and updating the energies of only the first atom. For the second atom, we repeat this process with the assignment table corresponding to the reverse pairs-list.

Calling the kernel twice leads to repeating some of the computations. We tried to avoid this by storing those values in the GPU global memory during the first kernel call and reusing them during the second call. This resulted, however, in a slowdown due to slower global memory access.

V. RESULTS

We present our results from accelerating the rigid docking and energy minimization steps of FTMap by mapping to the NVIDIA GPUs. The serial times were obtained by running the original unaccelerated FTMap code on a single core of a 3GHz quad-core Intel Xeon Harpertown processor. The code is written in C language and was compiled using Microsoft Visual Studio 8.

Currently the FTMap production code supports only coarse-grained parallelism through distributing rotations across nodes of a server. In previous work [15][16] we created a multicore version of the docking phase for comparison. The code was compiled using Microsoft Visual Studio 8 with standard optimizations (release mode). Docking, however, is only about 7% of the total computation in mapping. For the energy minimization step, creating an efficient multicore version appears to be challenging because of the small ratio of computation to communication.

Our GPU-accelerated code runs on a NVIDIA TESLA C1060 GPU, containing 240 processor cores @ 1.3 GHz. The GPU is housed in a Dell Precision workstation with a 3GHz quad-core Intel Xeon Harpertown processor running Windows XP. The GPU code was written using NVIDIA CUDA and compiled using Microsoft Visual Studio 8 with standard optimizations and the NVIDIA nvcc compiler.

A. *Speed-ups on Rigid-Docking Step*

Speedups achieved on various per-rotation tasks of the docking phase are shown in Table 1. Rotation and grid assignment are left on the host and thus have a speedup of 1. Correlation on the original PIPER was performed using an FFT whereas on the GPU it was implemented as direct correlation. As discussed earlier, GPU resources are underutilized during scoring and filtering leading to the modest speedup. The overall speedup achieved in docking is 32x. The speedups reported are for a probe grid size of 4^3 and a total correlation grid size of 128^3, which are typical for FTMap probes and proteins.

Comparing against our FFT based multicore implementation of PIPER, running on same quad-core processor as before, the GPU PIPER speed-up reduces to 11x. On multicore, as in the case of GPUs, we observed that for small ligand sizes, direct correlation is faster than FFT. Comparing against direct correlation based PIPER on multicore, the GPU PIPER speedup further reduces to 6x.

Table 1. Speedups for various computations in rigid docking.

Task (per rotation)	CPU Time	GPU Time	Speedup
Rotation + grid assignment	80 ms	80 ms	1
Correlations	3600 ms	13.5 ms	267x
Accum. desolvation terms	180 ms	1 ms	180x
Scoring and Filtering	200 ms	30 ms	6.67x
Total time per rotation	*4060 ms*	*125.5 ms*	*32.6x*

B. *Speed-ups on Energy Minimization Step*

Table 2 shows the speedup achieved on various energy and force computations mapped onto GPU kernels. The runtimes presented are for a single iteration of energy minimization, which involves performing around 10,000 atom-atom computations for each of the energy term. Force update kernel updates forces for the 2200 atoms in the complex.

We also measured the overall energy minimization times for various different protein-probe complexes. The average time for minimizing 2000 conformations of a complex on the original FTMap program is around 400 minutes. On our GPU accelerated version, the energy minimization time reduces to 32 minutes, representing an overall speedup of 12.5x for the energy minimization phase.

Table 2. Speedups for different energy evaluation and force update steps of energy minimization.

Computation	Serial Time	GPU Time	Speedup
Self energies	6.15 ms	0.23 ms	26.7x
Pairwise	2.75 ms	0.19 ms	17x
van der Waals	0.5 ms		
Force updates	0.95 ms	0.14 ms	6.7x

C. *Overall Speed-up*

On our GPU accelerated mapping program, the time for mapping a probe on a protein reduces from 435 minutes to 33 minutes., representing an overall speedup 13x for the entire FTMap program. Comparing to the multicore version of the docking phase, the overall speed-up reduces to 12.3x.

VI. CONCLUSION

We present a fast, GPU-based implementation of FTMap, a production binding site mapping program. Both the rigid-docking and the energy minimization phases are accelerated, resulting in a 13x overall speedup of the entire application over the current single-core implementation. While an efficient multicore implementation of FTMap may be possible, it is certainly challenging: we estimate it would require an effort greater than what we spent on the GPU mapping.

Overall, this work provides a cost-effective, desktop-based alternative to the large clusters currently being used by production mapping servers. Essential to the success of this work is restructuring the original application in several places, e.g., to avoid the use of neighbor lists.

In the future, we plan on extending this work to a multi-GPU implementation and integrating it into a production web server.

VII. REFERENCES

[1] Born, M. Z. (1920) *Physics*, 1, 45.

[2] Brenke, R. et al. (2009) Fragment-based identification of druggable 'hot spots' of proteins using Fourier domain correlation techniques. *Bioinformatics*, 25(5), 621-627.

[3] Brooks, B.R. et al. (1983) CHARMM: a program for macromolecular energy, minimization, and dynamics calculations. *J. Comp. Chem.*, 4, 187–217.

[4] Constanciel, R., and Contreras, R. (1984) Self consistent field theory of solvent effects representation by continuum models: Introduction of desolvation contribution. *Theoret. Chim. Acta (Berl.)*, 65, 1-11.

[5] Cornell, W. D. et al. (1995) A Second Generation Force Field for the Simulation of Proteins, Nucleic Acids, and Organic Molecules. *J. Am. Chem. Soc*. 117, 5179-5197.

[6] Ershov, R. E. (1970) Self-energy of a "smeared" charge. *Russian Physics Journal*, 13 (6), 813-813.

[7] http://en.wikipedia.org/wiki/Energy_minimization.

[8]http://farside.ph.utexas.edu/teaching/em/lectures/node56.html

[9]http://www.wag.caltech.edu/publications/theses/alan/subsectional_4_0_2_1.html.

[10] Kozakov, D., Brenke, R., Comeau, S., and Vajda, S. (2006) PIPER: an FFT-based protein docking program with pairwise potentials. *Proteins Structure, Function, Genetics*, 65, 392-406.

[11] Landon, et. al. (2007) Identification of Hot Spots within Druggable Binding Regions by Computational Solvent Mapping of Proteins. *J. Med. Chem.*, 50, 1231–1240.

[12] Pappu, R.V., Hart, R. K., and Ponder, J. W. (1998) Analysis and Application of Potential Energy Smoothing and Search Methods for Global Optimization. *J. Phys. Chem. B*, 102, 9725–9742.

[13] Schaefer, M. and Karplus, M. (1996) A Comprehensive Analytical Treatment of Continuum Electrostatics. *J. Phys. Chem.*, 100 (5), 1578-1599.

[14] Still, W. C., et. al. (1990) Semianalytical treatment of solvation for molecular mechanics and dynamics. *J. Am. Chem. Soc*., 112 (16), 6127-6129.

[15] Sukhwani, B. and Herbordt, M. C. (2010) FPGA Acceleration of Rigid Molecular Docking Codes. *IET Computers and Digital Techniques* (in press).

[16] Sukhwani, B. and Herbordt, M. C. (2009) GPU Acceleration of a Production Molecular Docking Code. In Proceedings of the *Workshop on General-Purpose Computation on GPUs*.

[17] Sukhwani, B. Herbordt, M. C. (2009) Accelerating Energy Minimization using Graphics Processors. In Proc. *Symposium on Application Accelerators in High Performance Computing.*