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
Numerical 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 and the second models the side chain flexibility by energy-minimizing 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.

1. INTRODUCTION

Discovering a new drug involves finding a site on a given protein which will bind a small molecule inhibitor with high affinity. Moreover, it 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, drug discovery involves docking-based screening of millions of candidate ligands for a given protein.

Molecular docking refers to the computational prediction of the least energy pose between two interacting proteins, in this case, a protein and a ligand. It involves searching the entire protein surface for the binding site, often requiring exhaustive 3D search. This is a computationally demanding process, requiring many hours to days of CPU runtime.

An important observation, however, is that certain regions on a protein surface, called hot-spots, 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 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, each ligand can then be docked on this local region or the hotspot region, 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 on 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 the GPU.

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 computation performed per iteration and a large fraction of it being serial accumulations. Most parallel accumulation schemes on GPUs require large amount 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 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.

2. COMPUTATIONAL MAPPING OF BINDING SITES

Computational mapping of binding site refers to the process of finding druggable binding-sites on the surface of a protein. Such binding sites, or hot-spots, 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.

2.1 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 2), 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]. The steps performed by PIPER for each protein-probe pair are shown in the left half of Figure 1. 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 \( \alpha, \beta, \gamma \) of the small molecule relative to the protein) is computed as a 3D correlation sum between the two grids

\[
E(\alpha, \beta, \gamma) = \sum_p \sum_{i,j,k} R_p(i,j,k)L_p(i+\alpha, j+\beta, k+\gamma)
\]  

(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.

Figure 2. Lock-and-Key Model for Rigid Docking

Thus for each rotation, \( O(N^3) \) translations are performed, each requiring \( O(N^3) \) computations. To speed-up the computations, the translations and correlations for a given rotation are replaced with an 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.

2.1.1 PIPER Energy Functions

During rigid docking, simple shape complementarity functions are often insufficient to discriminate true negatives from false negatives. In other words, simply eliminating some poses based on a poor shape complementarity score can lead to the loss of near-native solutions. Docking programs therefore often retain thousands of conformations for further evaluations using more complex energy functions.

To reduce the number of conformations requiring the more expensive second level evaluations without losing the near-native solutions, PIPER employs more complex energy functions during 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).

\[
E = E_{\text{shape}} + w_2 E_{\text{elec}} + w_3 E_{\text{desol}}
\]  

(2)

Both the shape complementarity and the electrostatics terms are computed as a weighted sum of two components each. Further, desolvation is represented as a sum of various pairwise potential terms. Up to 18 pairwise terms can be used, though using 4 terms is often sufficient. 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 \( L_p \) 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 based on score and distribution.

2.1.2 PIPER Profiling

Figure 3 (b) shows the distribution of time per rotation for different steps of PIPER rigid-docking phase. 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 desolvation terms 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.
2.2 FTMap Energy Minimization

2.2.1 Overview of Energy Minimization

In FTMap, the rigid docking step is followed by the energy minimization of the top scoring conformations. During energy minimization, the larger molecule is held constant and the side chains of the probe are free to move. This is done to model the flexibility in the side chains of the small molecule probe. From each rotation of the rigid docking phase, FTMap retains 4 top scoring conformations for energy minimization. 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 3 (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]. During each minimization iteration, the total potential energy of the complex is computed. In order to evaluate the potential energy of the system rapidly, it is often represented using force-fields. A force-field represents each atom in a molecular system as a point charge and the total potential energy of the system as a sum of various two, three or four-particle interactions [9]. Various force fields have been developed, with the more popular ones being CHARMM [3] and AMBER [5]. Due to the popularity of the CHARMM force fields, energy minimization is often referred to as minimization of the CHARMM potential or simply as CHARMM minimization.

Minimization 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 (right half of Figure 1). Forces acting on the atoms are obtained by differentiating the potential energy function with respect to the atom coordinates. 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.

During minimization, the move to the next iteration can be made using one of many optimization approaches such as steepest descent, conjugate gradient, quasi-Newtonian, or Newton-Raphson. Depending on the method chosen, minimization requires computing the first and, in some cases, the second derivatives of the energy functions. The choice of iteration method also affects the rate at which the energy of the system converges.

2.2.2 Energy Minimization ≠ Molecular Dynamics

Though the underlying computation of energy minimization is superficially similar to molecular dynamics (MD), it differs from MD simulations in various ways. These distinctions can be classified into two categories – the geometry of the problem and the way the computation is performed. From the point of view of the problem geometry, there are three differences. First, unlike MD where the system typically consists of millions of particles, energy minimization is often performed on a local region of the complex, resulting in only a few thousand atoms begin simulated, and requiring only up to a few tens of thousands of atom-pair evaluations per iteration. This makes accelerating this computation far harder since the computation per iteration is very small and the particle-filtering schemes often used in MD do not provide much benefit. Second, the neighborhood associated with each atom is much smaller compared to MD. Finally, since minimization is a refinement step, simply to model the flexibility in atom side chains, the motions are small.

With respect to the computations and the data structures, there are four major distinctions. First, unlike molecular dynamics where the movement of the atoms is based on Newtonian dynamic laws and produces a trajectory based on kinetic energy, minimization simply adjusts the atom coordinates so as to lower the total energy of the system [3][7]. Second, even though the energy terms computed in minimization are similar to those in molecular dynamics, the actual energy expressions evaluated during minimization are often quite different. Third, unlike MD where cell-lists are heavily utilized to organize the computations, minimization routines often do not employ cell-lists at all. Finally, even though energy minimization, like MD, uses neighbor-lists, they are seldom updated. The last two points are consequences of the different problem geometry and affect the way parallelization of the algorithm is approached.

2.2.3 Computations in Energy Minimization

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).

\[ E_{\text{total}} = E_{\text{non-bonded}} + E_{\text{bonded}} + E_{\text{angle}} + E_{\text{torsion}} + E_{\text{improper}} \] (3)

where: \( E_{\text{non-bonded}} \) is the non-bonded energy, \( E_{\text{bonded}} \) is the bonded energy, \( E_{\text{angle}} \) is the angle energy, \( E_{\text{torsion}} \) is the torsion energy, \( E_{\text{improper}} \) is the improper energy.
Energy minimization involves the repeated evaluation of this expression, once during each minimization iteration. As stated earlier, moving to the next iteration requires moving the atoms in the direction of the least energy conformation. Thus at each iteration, the total force acting on each atom is also computed and the atoms are moved in the direction of those forces.

Figure 4 shows the profiling results for the energy minimization step of the FTMap program. As shown in Figure 4 (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 4 (b)). This includes the electrostatics and the van der Waals energies.

The second term is the sum of the N(N-1)/2 pair-wise interaction terms, each proportional to the product of the two charges involved in the pair. Both the self-energy and the pair-wise interaction energy terms depend on the geometry of the solute [13]. The total electrostatic energy of a solute is thus given as a sum of all the self-energies, $E_i^{\text{self}}$, and the pair-wise interaction energies, $E_{ij}^{\text{int}}$ [13] (see Equation 4).

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

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

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

To compute the pair-wise interaction portion of the self energy, ACE defines atom charges as Gaussian distributions. $E_{ik}^{\text{self}}$ can then be computed by integrating the energy density of the electric field. For efficient computation, this is approximated as the sum of a short-range term, that approximates the Gaussian, and a long range term (first and second terms in Equation 6 respectively).

$$E_{ik}^{\text{self}} = \frac{q_i q_j}{\omega_{ik}} e^{-\left(\frac{r_{ik}^2}{\sigma_{ik}^2}\right)} + \frac{q_i^2 q_j}{8\pi} \left(\frac{r_{ik}^3}{r_{ik} + \mu_{ik}}\right)^4$$

$$E_{ik}^{\text{self}} = \frac{q_i q_j}{\omega_{ik}} e^{-\left(\frac{r_{ik}^2}{\sigma_{ik}^2}\right)} + \frac{q_i^2 q_j}{8\pi} \left(\frac{r_{ik}^3}{r_{ik} + \mu_{ik}}\right)^4$$

Here $q_i$ represents the charge on atom ‘i’, $r_{ik}$ is the distance between atoms ‘i’ and ‘k’, $\mu_{ik}$ is the size of the solute volume associated with atom ‘k’, $\omega_{ik}$ and $\sigma_{ik}$ determine the height and width of the Gaussian that approximates $E_{ik}^{\text{self}}$, and $\mu_{ik}$ 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]:

$$E_{ij}^{\text{int}} = 332 \sum_{j \neq i} \frac{q_i q_j}{r_{ij}} - 1667 \sum_{j \neq i} \frac{q_i q_j}{r_{ij}^2} + \sum_{j \neq i} \frac{\alpha_j}{\alpha_i} e^{-\left(\frac{r_{ij}^2}{\alpha_i \alpha_j}\right)}$$

$$E_{ij}^{\text{int}} = 332 \sum_{j \neq i} \frac{q_i q_j}{r_{ij}} - 1667 \sum_{j \neq i} \frac{q_i q_j}{r_{ij}^2} + \sum_{j \neq i} \frac{\alpha_j}{\alpha_i} e^{-\left(\frac{r_{ij}^2}{\alpha_i \alpha_j}\right)}$$
where \( \alpha_i \) and \( \alpha_j \) 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. The most general form of the LJ potential is given as

\[
E_{vdw} = \sum_{\text{non-bonded - pairs}} \left( \frac{A_{ik}}{r_{ik}^{12}} - \frac{C_{ik}}{r_{ik}^6} \right)
\]

(8)

where \( r_{ik} \) is the distance between the two atoms under consideration and \( A_{ik} \) and \( C_{ik} \) are atom-atom constants. The expression evaluated by FTMap to compute the van der Waals energy of an atom pair is

\[
E_{vdw, ij} = \epsilon_{ik} \left( \frac{r_{ik}^6}{r_{ik}^{12}} - \frac{8r_{ik}^6}{r_{ik}^{12}} + \frac{1}{2} \frac{2r_{ik}^6}{r_{ik}^6} \right)
\]

(9)

where

\[
\epsilon_{ik} = \epsilon_{i}\epsilon_{k}, \quad r_{ik} = (r_{i} + r_{k}) / 2
\]

and \( \epsilon_{i} \) and \( r_{i} \) 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 (9) 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.

2.2.5 Data Structures for Non-bonded Energy Evaluation

The non-bonded energy of an atom is the sum of the contributions due to neighboring atoms within a cutoff distance. For efficient computation, atoms are often arranged in neighbor lists, where each atom (the “first” atom) has an associated list of neighbors (the “second” atoms) that contribute to its energy (see Figure 5). As the positions of the atoms change, the neighbor lists are updated.

To compute the self-energies of the atoms, the original FTMap program cycles through the list of first atoms. For each first atom, it traverses the list of second atoms, updating the self-energies of both the first atom as well as the second atom. After the self-energies of all the atoms have been computed, they are accumulated and the total self-energy of the system is obtained.

Similarly, the neighbor list array is again traversed to compute the pair-wise interaction energies using the generalized Born equation. Note that the neighbor lists must be traversed twice since the evaluation of pair-wise interaction energy requires that the individual total self-energy of each atom be known. Finally the list is traversed once more to compute the van der Waals energies of all the atoms.

As is shown in Section 3.2, mapping the computations onto the GPU in the form of neighbor-lists does not yield an efficient implementation and must be modified.

3. MAPPING FTMAP ONTO A GPU

As in the original FTMap software, we split the FTMap on GPU into two steps: rigid-docking and energy minimization. We describe each of these in separate sub-sections.

3.1 Rigid-Docking on GPUs

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 \( O(N^3) \) 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.

Direct correlation outperforms FFT correlation on smaller ligand grid sizes 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, direct correlation is used here.

3.1.1 Direct Correlation on GPUs

![Figure 6. Multiple Correlation scores generated together.](image)
Direct correlation on a GPU replaces the following steps: forward FFT, modulation, and inverse FFT. The direct 3D-correlation translates one of the grids over the other and computes a sum of all the voxel-voxel interactions for each translation (Figure 6). Note that, unlike FFT correlation, the grids need not be padded to the size of the correlation. Further, multiple energy functions can be evaluated together.

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. For different energy functions, different result grids must be computed, one for each. The total pose score is then a weighted sum of these different grids.

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 7.

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.

Each GPU kernel fetches the appropriate voxels from the protein and the probe grids and computes the voxel-voxel score. Since the grids are generated on the host, they must be transferred from the host memory to the global 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. Since receptor grids are large and the shared memory per multiprocessor is relatively small, it is not possible to copy these grids to the shared memories; rather, we store the receptor grids in the global memory. The ligand grids are much smaller, however, so we tried to store them variously in the 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.

Both the shared and the constant memory, however, are small and thus limit the size of the largest ligand that can fit in its entirety. An important consideration is the complexity of the energy function used. With 4 pairwise potential terms and thus total 8 energy grids, 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 benefits. First, the loop and kernel launch overhead is amortized over multiple rotations. And second, each protein voxel fetched from the global memory (which is not cached) gets used multiple times, thus reducing the overall fetch time by the number of rotations that can be computed at once. Since access to global memory has higher latency, reducing accesses to global memory 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.

### 3.1.2 Scoring and Filtering on GPUs

For each rotation, the four top scoring conformations are retained for the energy minimization phase. The score of a conformation is obtained by computing the weighted sum of correlation scores for different energy functions. These weighted scores are then filtered to obtain top scores.
Filtering refers to selecting the top scores from different regions on the result map, rather than simply selecting the highest scores. This 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 and is shown in Figure 8.

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.

To perform scoring on GPU, the scoring weights (see Equation 2) must be transferred to the GPU. Since these are constant, we store them in GPU constant memory. Copying the weights from host to constant memory is performed only once.

As in the case of computing the correlation score, the result grid is distributed across different threads, with each computing the weighted score and recording its local best score. Note that since we still need to gather these top scores from different threads, find the best of these best scores, and flag the neighboring cells for exclusion, we only use 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. More complex schemes to distribute the scoring task across multiple multiprocessors are possible and can potentially lead to faster computation. Though this is a heavy under-utilization of the available GPU computation power, it simplifies the process of assembling these scores from different threads. More complex schemes to distribute the scoring task across multiple multiprocessors are possible and can potentially lead to faster computation. The computation involved in scoring, however, is very small, and thus the amount of communication to assemble the scores from different multiprocessors will likely nullify any performance benefit.

We divide the \( N^3 \) grid points equally among \( M \) threads, with each thread computing \( N^3/M \) weighted scores and finding the best scores within its subset (see Figure 9). Each thread stores the best score and the corresponding score index in a shared memory array at an index equal to its thread id. Once all the threads have finished processing their subset of results, a master thread traverses the shared array to find the best of these best. The master thread also flags the cells surrounding the current best score, for exclusion during the next round.

For quickly determining whether a cell must be excluded, the original program maintains 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. We tried to solve this by replacing the flagging array with an exclusion array containing only the indices of the cells that need to be excluded. This is a much smaller array and fits in the GPU shared memory. This scheme, however, requires that the exclusion array be traversed every time to determine for exclusion of a cell, and hence results in slowdown with respect to filtering on host.

We divide the \( N^3 \) grid points equally among \( M \) threads, with each thread computing \( N^3/M \) weighted scores and finding the best scores within its subset (see Figure 9). Each thread stores the best score and the corresponding score index in a shared memory array at an index equal to its thread id. Once all the threads have finished processing their subset of results, a master thread traverses the shared array to find the best of these best. The master thread also flags the cells surrounding the current best score, for exclusion during the next round.

For quickly determining whether a cell must be excluded, the original program maintains 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. We tried to solve this by replacing the flagging array with an exclusion array containing only the indices of the cells that need to be excluded. This is a much smaller array and fits in the GPU shared memory. This scheme, however, requires that the exclusion array be traversed every time to determine for exclusion of a cell, and hence results in slowdown with respect to filtering on host.

3.2 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. The original FTMap program performs these tasks using separate function calls. Since the GPU kernel launch has some overhead, and since some of the computations are computed in multiple tasks, we combined them where possible.

Since the evaluation of pairwise interaction requires the knowledge of the total self energies of all
the atoms, these two computations cannot be combined. We therefore 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.

Serial FTMap uses neighbor lists to cycle through different atom-pairs and compute 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 the 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 5), 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 5).

For efficient mapping of these computations on the 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.

3.2.1 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 10).

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 10). 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.

3.2.2 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.

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 11. Different atom-pairs are independent of each other and can be processed in parallel. We distribute these pairs equally among the 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 11 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.

Figure 12. Split pairs-lists. (Left) Forward list, (Right) Reverse list.

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 12. Note that there is no column for the energies of the second atom in the pair.

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 pairs-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 13). 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 13 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).

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.

4. 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.

4.1 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 GPU, 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.

<table>
<thead>
<tr>
<th>Task (per rotation)</th>
<th>CPU Time (msec)</th>
<th>GPU Time (msec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotation and charge assignment</td>
<td>80</td>
<td>80</td>
<td>1</td>
</tr>
<tr>
<td>Correlations</td>
<td>3600</td>
<td>13.5</td>
<td>267x</td>
</tr>
<tr>
<td>Accumulation of desolvation terms</td>
<td>180</td>
<td>1</td>
<td>180x</td>
</tr>
<tr>
<td>Scoring and Filtering</td>
<td>200</td>
<td>30</td>
<td>6.67x</td>
</tr>
<tr>
<td>Total time per rotation</td>
<td>4060</td>
<td>125.5</td>
<td>32.6x</td>
</tr>
</tbody>
</table>

4.2 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.

<table>
<thead>
<tr>
<th>Computation</th>
<th>Serial Time</th>
<th>GPU Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self energies</td>
<td>6.15 ms</td>
<td>0.23 ms</td>
<td>26.7x</td>
</tr>
<tr>
<td>Pairwise</td>
<td>2.75 ms</td>
<td>0.19 ms</td>
<td>17x</td>
</tr>
<tr>
<td>van der Waals</td>
<td>0.5 ms</td>
<td>0.14 ms</td>
<td>6.7x</td>
</tr>
</tbody>
</table>

4.3 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. With 16 probes to be mapped to a protein, the total mapping runtime reduces from 116 hours to 8.8 hours, 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.

5. 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.

6. REFERENCES


