GPU-Accelerated High-Accuracy Molecular Docking using Guided Differential Evolution

Martin Simonsen, Mikael H. Christensen, René Thomsen, and
Christian N. S. Pedersen

Abstract The objective in molecular docking is to determine the best binding mode of two molecules *in silico*. A common application of molecular docking is in drug discovery where a large number of ligands are docked into a protein to identify potential drug candidates. This is a computationally intensive problem especially if the flexibility of the molecules is taken into account. We show how MolDock, which is a high accuracy method for flexible molecular docking using a variant of differential evolution, can be parallelised on both CPU and GPU. The methods presented for parallelising the workload result in an average speedup of 3.9x on a 4-core CPU and 27.4x on a comparable CUDA enabled GPU when docking 133 ligands of different sizes. Furthermore, the presented parallelisation schemes are generally applicable and can easily be adapted to other flexible docking methods.

1 Introduction

In a modern drug discovery process, *in silico* identification of ligands (small molecules) which bind to a target protein is often used in the search for novel drug candidates. Such ligands are likely to change the function of the target protein and

Martin Simonsen
CLC Bio, Finlandsgade 10–12, Katrinebjerg, DK-8200 Aarhus N, e-mail: msimonsen@clcbio.
Work done while affiliated to the Bioinformatics Research Centre (BiRC), Aarhus University.

Mikael H. Christensen
CLC Bio, Finlandsgade 10–12, Katrinebjerg, DK-8200 Aarhus N, e-mail: mchristensen@clcbio.com

René Thomsen
CLC Bio, Finlandsgade 10–12, Katrinebjerg, DK-8200 Aarhus N, e-mail: rthomsen@clcbio.com

Christian N. S. Pedersen
Bioinformatics Research Centre (BiRC), Aarhus University, C. F. Møllers Allé 8, DK-8000 Aarhus C, e-mail: cstorm@birc.au.dk

may therefore be used to design new pharmaceuticals. From a known 3D structure of a target protein, virtual screening methods can be used to search large ligand databases and create a ranked list of drug candidates. By focusing subsequent *in vitro* experiments on the top ranked ligands, the cost of experimental testing can be reduced.

A common virtual screening approach is to use molecular docking methods to identify binding modes between a ligand and a protein. Molecular docking methods use scoring functions to identify likely binding modes. Rigid docking methods consider only the translation and orientation of the ligand relative to the protein while the shape of both molecules is fixed during the docking process. In flexible molecular docking the conformation, i.e. the internal structure, of at least one of the molecules is allowed to change. It is common to consider only ligand flexibility while the protein is kept rigid to reduce the number of parameters which have to be optimised. Heuristic methods such as simulated annealing [7, 4], ant colony optimisation [2, 8] and evolutionary algorithms [16, 18] are commonly used to efficiently sample the large search space in flexible docking.

In molecular docking experiments the accuracy of the method is a primary concern but performance in terms of running time per ligand plays a significant role in virtual screening, where many ligands have to be docked into one or more target proteins. Docking methods that take molecular flexibility into account typically use several minutes per ligand, which makes it necessary to employ distributed computing for large virtual screening experiments. One way of reducing the need for expensive hardware in virtual screening is to accelerate molecular docking methods by taking advantage of modern general purpose GPUs.

A first application of GPUs to accelerate molecular docking is [9] where shaders are used to compute the scoring function of PLANTS [8] which is a flexible molecular docking method. Another application of GPUs to flexible docking is [6] where both the genetic algorithm (GA) and scoring function used in AutoDock [3] are accelerated. GPUs are also used in [17] to accelerate the scoring function of PIPER [10] which performs rigid docking using fast Fourier transforms. Other applications of GPUs in molecular modelling are reviewed in [15].

In our work, we explore parallelisation on CPUs and GPUs of MolDock [18] which is a flexible molecular docking method. We present parallelisation schemes for both the differential evolution (DE) algorithm and scoring function used in MolDock which are also applicable to other flexible docking methods. Experiments with a GPU-accelerated version of MolDock show an average speedup of 27.4x for docking a ligand into a protein compared to a similar implementation running on a single core of a CPU. Compared to previous work in this area, the GPU parallelisation scheme presented here results in a better utilisation of GPU cores and hence a better speedup.

The rest of this chapter is organized as follows: In Section 2, we introduction the basic concepts of flexible molecular docking. In Section 3, we present the MolDock method for flexible molecular docking, its score function, and guided DE. In Section 4, we present our parallelisation schemes for the DE and score function of MolDock using CUDA. In Section 5, we present the obtained experimental results.

2 Flexible Molecular Docking

We model ligands as one or more rigid structures of atoms connected by rotatable covalent bonds, where each bond has a dihedral angle associated with it. The dihedral angels define the ligand conformation and by allowing them to change, the flexibility of the ligand is taken into account. With a flexible ligand and a rigid protein, the optimal binding mode is found by optimising the translation, orientation and configuration of dihedral angles of the ligand with respect to a scoring function. Fig. 1 shows an example of docking a ligand into a protein cavity.

A scoring functions is needed to estimate the interaction strengths and internal ligand energy. The most common approach is to use energy force fields which model molecular energy forces such as electrostatic and van der Waals forces that compute the interaction energy between the ligand and the target protein by summing over the energy contributions from each pair of atoms in the ligand and protein [5, 12]. The number of atoms in the protein is usually considerably larger than in the ligand, and precomputed energy grids are often used to speed up energy computations. Energy grids are constructed for each atom type by placing a grid over the protein and computing the energy contribution from all atoms in the protein for each point in the grid. The energy of a ligand atom can then be obtained by interpolating between the grid points closest to the position of the atom.

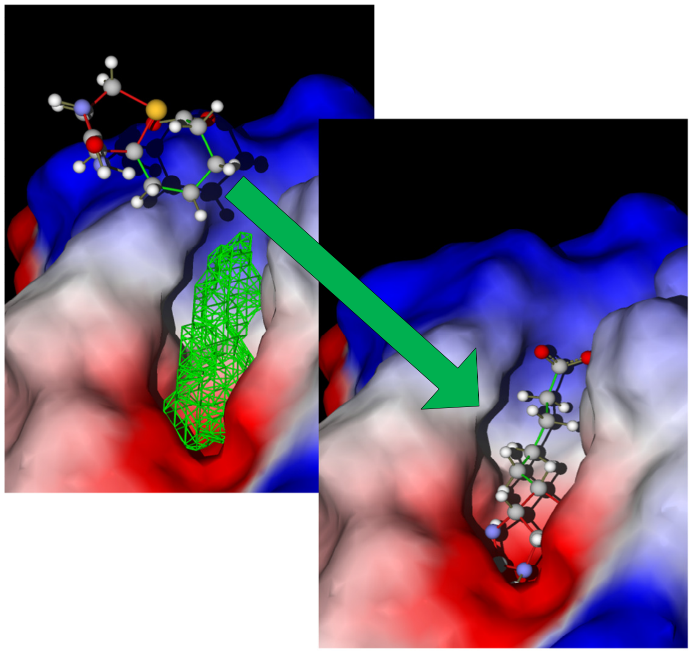


Fig. 1 Illustration of a ligand outside a protein cavity and a docking of the ligand in the cavity.

3 MolDock

MolDock [18] is a method for performing high accuracy flexible molecular docking which uses a variant of DE called *guided* differential evolution as search heuristic. The method takes full ligand flexibility into account while the protein remains rigid during the docking process.

3.1 Guided Differential Evolution

The guided DE algorithm used in MolDock is based on the original DE optimisation method by Storn and Price [16] but uses knowledge of cavities in the target protein to constrain the search space. As in the original DE method, guided DE starts out with an initial population of candidate solutions (individuals). In MolDock, each solution is a parameter vector which describes a pose (candidate binding mode). The vectors each contain a 3D translation, an orientation (a rotation by an angle around a unit vector), and a dihedral angle for each rotatable bond in the ligand.

```
procedure Differential Evolution
    initialise population with random individuals
    evaluate individuals
    while (not termination-condition)
        for ($i = 0$; $i < popsize$; $i$++)
            create offspring $O[i]$ from parent $P[i]$ by procedure below
            evaluate offspring $O[i]$
            if (offspring $O[i]$ is better than parent $P[i]$)
                replace parent with offspring
            else
                keep parent in population
            end if
        end for
    end while

procedure create offspring $O[i]$ from parent $P[i]$
    randomly select parents $P[i_1]$, $P[i_2]$, $P[i_3]$, where $i_1 \neq i_2 \neq i_3 \neq i$
    for ($j = 0$; $j < n$; $j$++)
        if ($U(0,1) < CR$)
            $O[i][j] = P[i_1][j] + F \times (P[i_2][j] - P[i_3][j])$
        else
            $O[i][j] = P[i][j]$
        end if
    end for
```

Fig. 2 Pseudo code for the generic DE algorithm. (The function $U(0,1)$ is a uniformly distributed value between 0 and 1, CR is the crossover rate, F is the scaling factor, and n is the number of problem parameters.)

Initially, all parameters are set to random values and the corresponding poses are evaluated using the scoring function described in Section 3.2. The parameters in each vector are then optimised with respect to the scoring function as follows: Trial vectors (offspring) for each individual are created in a *mutation* and *crossover* step. These vectors are then evaluated using the scoring function and used to update the population in a *selection* step. During selection each vector in the population is replaced with the corresponding trial vector if the trial vector resulted in a better score. An outline of the generic DE algorithm and the offspring creation scheme is shown in Fig. 2.

The difference between DE and guided DE lies in the way translation parameters for trial vectors are computed. Binding sites are often located in cavities in the protein and this knowledge is used in guided DE to restrict the translation of poses as follows. If all atoms in a pose are positioned outside cavity areas, the pose is translated such that a random ligand atom is placed inside the cavity area. Cavities are predicted as described in [18] and represented as points in a 3D-grid with a resolution of 0.8Å where each grid point has a boolean value indicating if the point is in a cavity or not. If no cavities are detected, pose positions are restricted to a cubic search volume.

3.2 Scoring Function

The scoring function used by MolDock to estimate the interaction energy of poses is defined as:

$$E_{score} = E_{inter} + E_{intra} \,. \tag{1}$$

E_{inter} is the intermolecular energy, i.e. the interaction energy between heavy atoms in the ligand and protein. It is computed as:

$$E_{inter} = \sum_{i \in ligand} \sum_{j \in protein} \left(E_{PLP}(r_{ij}) + 332.0 \frac{q_i q_j}{4 r_{ij}^2} \right) . \tag{2}$$

The PLP term, E_{PLP}, is a piecewise linear potential which given the distance between two atoms, r_{ij}, approximates the van der Waals forces and the potential energy of hydrogen bonds between atoms (see [18] for further details). The second term in the summation is a Coulomb potential with a distance dependent dielectric constant which approximates the electrostatic energy between two atoms, where q_i and q_j are the electric charges of the two atoms. In this work, E_{inter} is computed using trilinear interpolation on the precomputed energy grids.

E_{intra} is the intramolecular energy of the ligand, i.e. the internal energy of the ligand which is computed as:

$$E_{intra} = \sum_{i \in ligand} \sum_{j \in ligand} \left(E_{PLP}(r_{ij}) + E_{clash}(r_{ij})\right) + \sum_{\theta \in rotatable\ bonds} A\left[1 - cos(m \cdot \theta - \theta_0)\right] . \quad (3)$$

The double summation runs over all heavy atom pairs in the ligand which are more than two bonds apart. E_{clash} is a fixed penalty of 1000 given to atom pairs that have a mutual distance of less than 2.0Å to prevent unrealistic conformations of the ligand. The second summation is the torsional energy of all rotatable bonds in the ligand, where the A, θ_0 and m parameters are set as described in [18] and θ is the dihedral angle for a given rotatable bond.

The original MolDock scoring function also takes the directionality of hydrogen bonds into account by scaling the PLP contribution of hydrogen bonds according to the angles between atoms involved in the bond. However, this requires an iteration over all protein atoms involved in hydrogen bonds with ligand atoms. In this work, the directionality of hydrogen bonds are therefore not taken into account.

3.3 Flexible Docking with MolDock

Initially a ligand and protein is loaded into memory and preprocessed. The preprocessing involves detecting all rigid structures and rotatable bonds in the ligand, cavity detection in the protein and computation of energy grids. Additionally, a search space for the DE algorithm needs to be created but this cannot always be done without user interaction. In this work we used the center of the experimentally known ligand as the center for a search space with size 30Å×30Å×30Å.

The next step is energy minimisation using guided DE and the MolDock scoring function. Because of the stochastic nature of DE algorithms, multiple energy minimisation runs increase the chance of finding the lowest scoring binding mode. In [18] 10 runs were performed for each ligand and after each run the solution with the best score was stored. In the original implementation, an additional final step re-ranked the stored solutions using a more accurate scoring function in order to produce a ranked list of solutions.

3.4 Parallelisation of Scoring Function and Differential Evolution

Table 1 shows the time consumption of the different steps in the MolDock algorithm. The time used on re-ranking poses is not shown as this step has been omitted in this work. However, re-ranking is fast and only has to be performed once for every run. The time used on preprocessing the protein is also not shown as this only has to be done once for each protein and when multiple ligands are docked into the same protein this time become insignificant. The protein used in Table 1 contains 2163 heavy atoms and here cavity prediction takes 0.93s while computation of energy grids takes 24.06s. Parallelising the computation of energy grids, which is the most time consuming step of the two, is easily done since each entry in a grid can be computed independently.

The most time consuming step of docking a ligand into a preprocessed protein is pose evaluation (computation of pose atom positions and scoring of poses). Pose evaluation can be parallelised in a straightforward way as each individual in the population is evaluated independently (population level parallelisation). The mutation, recombination and selection steps in DE can also be parallelised on population level, i.e. the parameter vector of each individual in the population can be computed independently. However, to take advantage of the hundreds of cores in modern GPUs it is necessary to increase the parallelisation granularity. By using a large number of threads it is possible to hide the high memory latency by switching to other threads while waiting for memory access. CUDA, and GPUs in general, provide fast synchronisation between threads which makes fine grained parallisation schemes, such as the one presented in Section 4, possible. On CPUs, high granularity parallelisation using multi-threading often give rise to a high overhead due to thread synchronisation which results in lower performance. SIMD (Single Instruction Multiple Data) instructions could be used to implement a fine grained parallelisation scheme for CPUs, but this has not been done in the current work. A much simpler approach for parallelisation of MolDock on CPUs is to create multiple processes each running an instance of the MolDock application and then let each instance execute one or more energy minimisation runs. If multiple ligands are docked, the total number of runs needed will be large enough to fully utilise all cores in most modern CPUs. However, this approach is not suitable for GPUs.

To reduce the overhead of thread synchronisation and increase the workload for each step of the MolDock algorithm, multiple runs of energy minimisation can be performed in parallel. Performing r runs of energy minimisation in MolDock with a population of p individuals correspond to a single energy minimisation run with a population of $p \cdot r$ where mutation and crossover are restricted to groups of p individuals. As shown in Section 5.3 the use of this strategy is important to achieve good performance on GPUs.

Table 1 Performance profile of MolDock. Running times were obtained using one core of the CPU as described in Section 5. Computation time for energy grids is not shown.

Step	Time	%
Ligand preprocessing	<0.01s	0.00%
Pose computation	4.44s	16.53%
Scoring function	21.82s	81.24%
DE	0.60s	2.23%
Total	26.86s	

Runs = 10; Population size = 64; DE iterations = 3,000;
Ligand: heavy atoms = 26; rotatable bonds = 8

4 Parallelisation of MolDock with CUDA

This section describes the CUDA kernels used to perform energy grid computation, pose evaluation and guided differential evolution on a CUDA enabled GPU. The main objective of the kernel designs is to maximise parallelisation granularity, minimise thread serialisation and ensure coalesced access to data stored in the global memory. The dimensions of the thread block grids used in the kernels have no impact on performance and were chosen to reduce complexity of index computations. Thread blocks contain 64 threads (unless otherwise stated) which gave the best overall performance. In our experiments both the pose evaluation and the DE are executed on the GPU. However, as shown in Fig. 3, it is also possible to execute only the pose evaluation on the GPU and let the DE runs on the CPU.

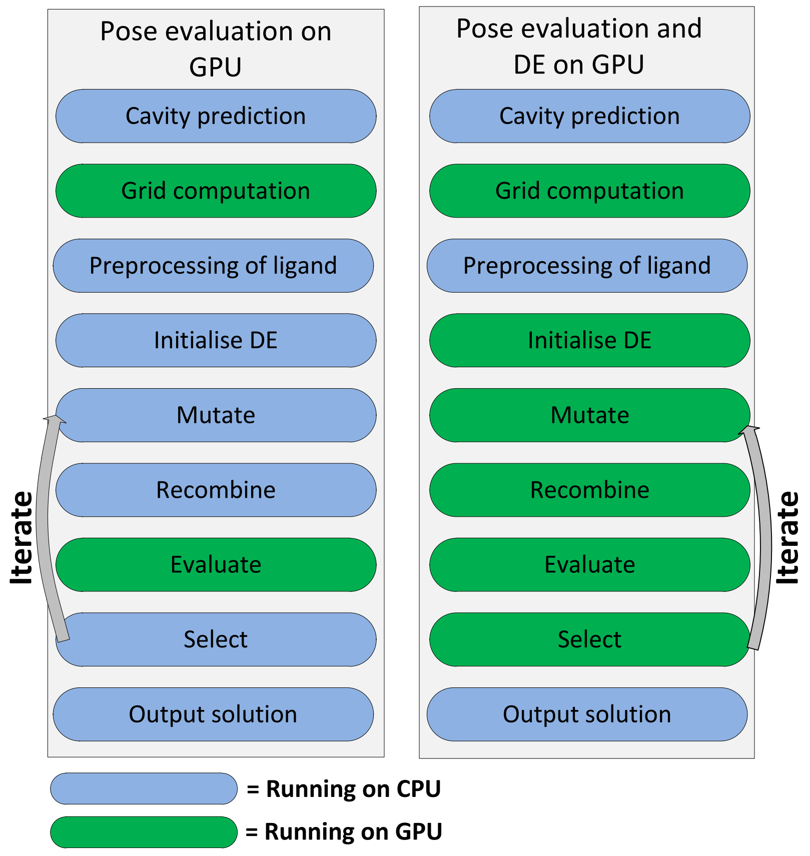


Fig. 3 The program flow of the two GPU accelerated versions of MolDock.

In [18] 10 runs where performed with a population size of 50. This allows a maximum of 500 independent units of work using the parallelisation of MolDock on population level described in Section 3.4. On modern GPUs, 500 threads are not enough to fully utilise all cores, hence the parallelisation granularity needs to be increased. This can only be done to some degree when computing pose atom positions (see Section 4.3) but in case of both the scoring function and DE algorithm, a high granularity can be achieved. Fine grained parallelism gives rise to dependencies between threads in different thread blocks. Consequently, some steps are implemented in multiple kernels to allow global thread synchronisation.

4.1 Preprocessing and Data Structures

In the preprocessing phase we compute energy grids, perform cavity detection and create a representation of the ligand which is suitable for CUDA enabled GPUs. Energy grids are computed on the GPU while all other steps are performed on the CPU.

Computation of energy grids is done using a single CUDA kernel, where each thread-block contains 128 threads and computes all grid points for a fixed z-coordinate. After all grids have been computed they are stored in 3D textures to enable the use of hardware accelerated trilinear interpolation. Cavity points are represented using a 3D grid and also stored in a 3D texture to allow cached access using the CUDA point filter access mode.

Preprocessing of ligands consists of several steps. First, internal rigid structures (*RS*) are identified along with rotatable bonds which are the single covalent bonds that connect rigid structures. Fig. 4 shows a decomposition of an example ligand molecule into rigid structures. The atoms in the ligand are represented as an array containing 3D coordinates, the atom types and the charge of each atom. Data for atoms in the same rigid structure are stored sequentially such that data for all atoms in a rigid structure can be located using two indices stored in a list. Rotatable bonds are represented as a list of index-pairs which identify the two heavy atoms in each bond. For each rotatable bond we also store information on all possible dihedral angles to enable computation of torsional energies. All these data structures are relatively small and can therefore be stored in the GPUs constant memory to allow fast cached access.

When computing the internal energy of the ligand, only atoms more than two bonds apart are considered. To avoid computing the bond distance between all atom pairs more than once, a list of atom pairs that need to be evaluated is created. As this list can become fairly large it is stored in global memory to prevent pollution of the constant memory cache.

4.2 Guided Differential Evolution

We use the parallelised Mersenne Twister [11, 14] from the CUDA SDK to generate uniformly distributed random numbers. To reduce the number of calls to the Mersenne Twister kernel each call generates enough random numbers for 100 iterations of the DE algorithm (where 100 was chosen because it gave the best performance in our experiments). The random numbers are stored in global memory and loaded by the DE kernel when needed.

Three kernels are used to implement the three steps of DE (mutation, crossover and selection) in CUDA. All kernels use one thread for each parameter in each individual where threads in the same block handle the same parameter type in different individuals. There are some challenges associated with handling random numbers when DE is parallelised in this way. In most cases, threads require unique random numbers which is easily handled. But in the mutation and crossover steps, threads working on the same individual must share some random numbers.

In the mutation kernel, each thread must select three random individuals which are mutually distinct and also distinct from the individual handled by the thread. Additionally, for threads working on parameters in the same individual these three individuals must be the same. This is achieved by loading the same three random numbers in threads working on the same individual and then mapping the numbers to parameter vector indices satisfying the above constraints. Next, all parameters in the population of the type handled by a thread are loaded into shared memory to allow fast random access by threads in a block. Mutated parameters are then computed and stored in global memory. The mutation kernel is also responsible for updating the minimum energy of each individual as this cannot be done in the selection kernel (see below) because of concurrent access to pose energies by threads in different thread blocks.

The recombination kernel creates a trial parameter vector for all individuals. Each parameter in a trial vector is selected from either the mutant or target parameter vector of the corresponding individual. The source vector is selected randomly using

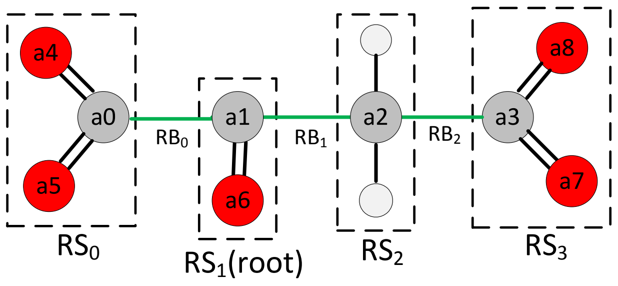


Fig. 4 The decomposition of an example ligand molecule into rigid structures.

a random number and a crossover constant as follows. Each thread loads a unique random number from the global memory and use it to select a random source vector. As specified by the original DE method, at least one random parameter must be chosen from the mutated vector for each individual. Using the same strategy as in the mutation kernel, each thread reads another random number that is identical for threads associated with the same individual. This number is then used to choose a parameter from the mutated vector. The final trial parameter vectors are stored in global memory in a single 1D array.

In the selection kernel, threads use interaction energies, computed by the scoring function, to choose between trial vectors and the target vectors. Each thread reads the interaction energies of a trial and target vector and if the energy of the trial vector is lower than the target vector, the trial parameter handled by the thread is used to overwrite the corresponding target parameter in global memory.

Finally, we need to determine if a pose has at least one atom inside a cavity area. This is done in a different way than in the original MolDock method to enable a better parallelisation. Instead of translating poses into cavities, poses are simply discarded if they lie outside all cavity areas. However, this step requires access to the positions of all atoms in a pose and is therefore better handled by the scoring function kernel (see Section 4.4).

4.3 Pose Computation

In order to evaluate the ligand energies, explicit atom coordinates must be computed. These coordinates are computed using two kernels. The first kernel computes a transformation matrix for each rigid structure in all poses using the parameter vectors from the DE kernels. Threads in the same thread block compute transformation matrices for the same rigid structure in different poses as shown in Fig. 5(a). The RS closest to the center of mass in the ligand is denoted RS_{root}. Transformation matrices for $RS \neq RS_{root}$ rotates rigid structures around a rotatable bound while transformation matrices for RS_{root} translate and rotate the whole molecule. Each transformation matrix is computed using a single thread which stores the result temporarily in shared memory. When all threads in a block have computed their assigned transformation matrix, the matrices are written to the global memory.

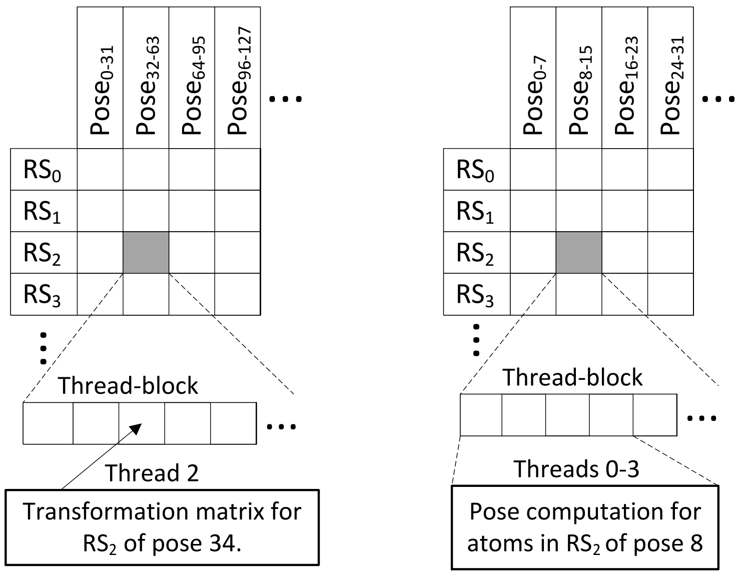
The second kernel computes the position of atoms in each pose using the transformation matrices stored in global memory as shown in Fig. 5(b). For each rigid structure, i, in pose, j, a transformation matrix which transforms the coordinates of all atoms in $RS_{i,j}$ is computed using multiplication of the transformation matrices computed by the first kernel and four threads for each matrix. Next, three of the four threads are used to compute atom coordinates for atoms in $RS_{i,j}$ which are then stored in global memory.

4.4 Scoring Function

The scoring function is computed using two kernels. In both kernels atom coordinates are loaded from the global memory into shared memory as needed using coalesced reads and accessed there.

The first kernel, illustrated in Fig. 6, performs two different functions. The first function is computation of inter- and intra-molecular energies for each atom excluding torsional energies. Here, one thread is used for each atom in a pose where threads in the same thread block work on the same ligand atom but from different poses. The intermolecular energy is computed using trilinear interpolation in energy grids. Intramolecular energy is computed as the PLP of atom pairs which are more than two bonds away from the atom handled by a thread. All threads in a block runs through the same list of atom pairs as these threads handle the same ligand atom. The total energy contribution of each atom is stored in global memory as a final step in this kernel.

The second function performed by the kernel is computation of the torsional energy term. This is done in parallel with computation of atom specific energies using additional thread blocks where the grid coordinates of a thread block is used



(a) Grid layout for the transformation matrix computation

(b) Grid layout for the atom position computation

Fig. 5 The two kernels used for computing pose atom positions.

to determine the function of each block (see Fig. 6). In a thread block computing torsional energies, all threads compute the energy of one possible dihedral angel from the same bond but for different poses. The torsional energy contribution is also stored in global memory along with the atom specific energies.

The second kernel is used to compute the total interaction energy of poses and to handle cavities. A single thread is used to sum the atom specific and torsional energies for each pose. Each thread also determines if at least one atom in the pose is inside a cavity by performing a lookup in the cavity grid for each atom. If no atoms are inside a cavity, a penalty of 10,000 is given to the pose interaction energy, thereby discarding the pose. In this way the hard constraint used in MolDock where ligands are always positioned in cavities is replaced by a soft constraint which can be efficiently computed on a GPU. Finally, the total interaction energy of each pose is stored in the global memory.

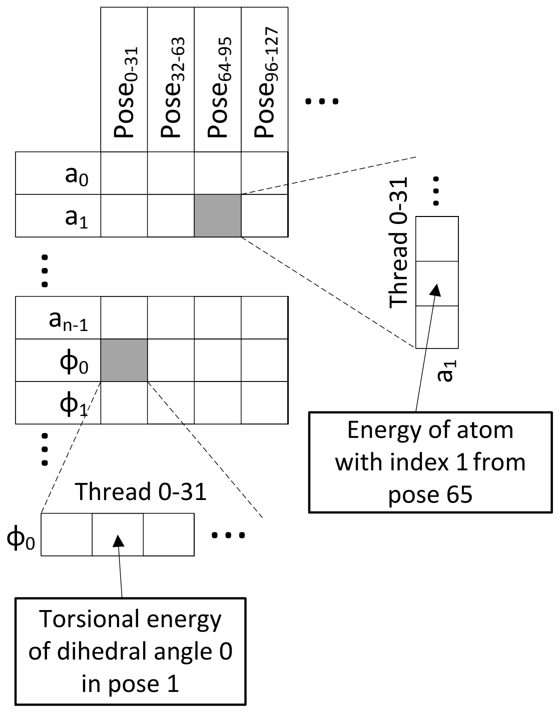


Fig. 6 Grid layout for computing both atom specific energies and torsional energies.

5 Results and Discussion

Our parallelisation schemes handle functions which are common for several popular flexible docking methods similar to MolDock such as PLANTS [8], GEMDOCK [19] and AutoDock [3] and are thus generally applicable. In particular, the parallelisation scheme for computation of pose atom coordinates can be directly used in most flexible docking methods. The parallelisation scheme for the MolDock scoring function is also fairly easy to apply on other methods (fx PLANTS, GEMDOCK and AutoDock) and can also be used in combination with other optimisation methods than DE.

An optimised implementation of MolDock was made for x86-64 compatible CPUs in *C*++ and parallelised on population level with OpenMP [1] as described in Section 3.4. The kernels described in Section 4 were implemented using the CUDA SDK 3.2 and used to accelerate pose evaluation and DE steps from the CPU implementation (see Fig. 3). All code was compiled with the -O3 optimisation option using GCC 4.4.3 and nvcc 3.2. Both the CPU and GPU implementations use single precision floating point numbers and compute multiple runs of the energy minimisation step by creating one large population as described in Section 3.4. Unless stated otherwise, experiments using the GPU perform both pose evaluation and DE on the GPU. The crossover rate (CR) and scaling factor (F) for the DE algorithm were set to 0.9 and 0.5 respectively which is the same values used for experiments in [18]. These parameters affect the rate of convergence and the docking accuracy but they do not have any direct influence on time consumption.

The experiments were performed on a machine containing an Intel Core 2 Quad Core (Q9450) CPU @ 2.66 GHz with 4GB RAM running Ubuntu 10.04 and equipped with a Geforce 8800GT GPU (112 cores @ 1.5 GHz). The CPU and GPU are both mainstream products in the same price range and therefore provides a good basis for comparison.

5.1 Dataset

We used the original GOLD benchmark dataset [13] to perform all experiments. This dataset contains the same 77 co-crystallized protein-ligand complexes used for experiments in [18] plus an additional 56 complexes. When the energy grids have been computed the running time of MolDock depends mainly on the number of heavy atoms and rotatable bonds in the ligand. The GOLD dataset contains a

Table 2 Dataset statistics (133 complexes).

	min	max	avg.
#Heavy atoms	6	55	21.32
#Rotatable bonds	0	23	5.89

diverse range of ligand sizes as shown in Table 2 and is therefore well suited for benchmarking the parallelised MolDock implementation. In all experiments individuals in DE populations were initialised by randomly translating and orientating the co-crystallised ligand and assigning random dihedral angles to rotatable bonds.

5.2 Docking Accuracy

Several differences between the implementation used here and the one used in [18] could affect the accuracy. Both the use of energy grids and the omission of a scaling factor for hydrogen bonds change the energy landscape. Also, the use of trilinear interpolation and arithmetic functions with low precision on the GPU might affect the accuracy. Omission of the pose re-ranking step does, however, not affect the energy minimisation step as it is used in a post-processing step to select the best pose of all energy minimisation runs.

To determine any loss of accuracy we docked the same 77 complexes used in [18] on both the CPU and GPU. Table 3 shows the number of successfully docked complexes where successful means that a solution was found with a RMSD $\leq$2Å relative to the co-crystallized ligand. The results were obtained using a population size of 64 and 3,000 iterations of the DE algorithm. To enable a comparison with the results in [18], Table 3 shows both the number of successful docks using the best scoring pose of all runs (the "Score" column) and using the pose with the lowest RMSD of all runs (the "RMSD" column). With re-ranking of poses using a more precise scoring function, the number of successfully docked poses can be expected to lie between these two numbers. The original MolDock method successfully docked 67 of the 77 complexes and the results in Table 3 show that both the CPU and GPU implementations have comparable high accuracy with 10 runs if re-ranking is used.

In [6] a decrease in accuracy was observed when using hardware accelerated trilinear interpolation on the GPU to compute intermolecular energies. We did not observe any difference in accuracy when using this feature compared to performing

Table 3 The average number (and standard deviation) of successfully docked complexes out of 77 complexes using best-score ("Score" column) and best-RMSD ("RMSD" column) of all runs.

	CPU		GPU	
#Runs	Score	RMSD	Score	RMSD
1	56.0 ±1.5	56.0 ±1.5	55.9 ±2.0	55.9 ±2.0
5	59.9 ±2.1	68.0 ±1.8	60.0 ±1.7	67.7 ±1.4
10	59.6 ±1.1	69.6 ±1.2	59.6 ±1.2	70.1 ±0.9
15	58.4 ±1.4	71.9 ±0.8	59.5 ±1.7	72.3 ±0.9
20	58.8 ±1.2	71.7 ±0.9	58.0 ±0.8	72.4 ±1.2

The numbers are the average of 10 individual experiments.

interpolation in software on the GPU. The performance is, however, increased by up to 68% when hardware accelerated interpolation is used.

5.3 Benchmark Results

Table 4 shows the time consumption of the different steps in MolDock when docking a single complex of average size. In case of the CPU, running times using both a single core and four cores for the pose evaluation step are shown. Parallelisation of MolDock with threads on the CPU does not fully utilise all cores because of thread synchronisation overhead. The DE algorithm was not parallelised in the CPU implementation as experiments showed that the overhead of thread synchronisation caused the running time of DE to increase by up to 4x. However, as shown in Fig. 7, a speedup of close to 4x with 4 cores can be achieved by running multiple instances of the program (one for each core) in parallel and is therefore the preferable way to parallelise MolDock on a CPU.

In case of the GPU, Table 4 shows the running times for performing only pose evaluation on the GPU and when both pose evaluation and DE are running on the GPU. With DE running on the CPU, data must be copied between the CPU and GPU after each iteration of the DE algorithm. Here, the time used on copying data and running the DE algorithm on the CPU is responsible for 45% of the total time consumption. By running the DE algorithm on the GPU, the time consumption of the DE algorithm is reduced and most data transfers avoided which increases performance significantly.

The preprocessing time of the protein is also improved by parallelising the computation of energy grids. Compared to 20.04s using one core on the CPU, the energy grids can be computed in 12.23s when 4 cores are used and in only 1.42s on the GPU. Consequently, the total time (including the time used on initialising all data

Table 4 Time consumption of the steps for docking a single ligand in MolDock on CPU and GPU.

	CPU		GPU	
Step	1-core	4-cores	PE	PE+DE
Ligand pre-processing	<0.01s	<0.01s	<0.01s	<0.01s
Pose computation	4.44s	1.16s	0.34s	0.36s
Scoring	21.82s	6.36s	0.61s	0.61s
DE	0.60s	0.61s	0.59s	0.15s
GPU data copy	N/A	N/A	0.18s	∼0s
Total time	26.86s	8.13s	1.72s	1.12s
Speedup	1.0x	3.3x	15.6x	24.0x

PE = Pose Evaluation

Runs = 10; Population size = 64; DE iterations = 3,000;

Ligand: heavy atoms = 26; rotatable bonds = 8 structures on the CPU and GPU) needed to dock the complex used in Table 4 is reduced from 52.2s using a single core on the CPU to 3.96s on the GPU which is a 13.2x speedup.

Fig. 7 shows the average speedup achieved on all 133 complexes in the GOLD dataset by both CPU and GPU as a function of the number of runs performed on each complex. The running times were measured as the time used on energy minimisation plus the time used on preprocessing the ligand, i.e. the total time needed to dock a ligand into a preprocessed protein. When performing 10 runs, multi-threading on the CPU resulted in an average speedup of 3.3x whereas an average speedup of 3.9x was achieved by running multiple instances of the program on the CPU each docking a different ligand.

With only 4 cores available in the CPU, it is easy to achieve a good utilisation of all cores. However, on the GPU a large number of thread blocks is needed to fully utilise all 112 cores which can be achieved by increasing the number of runs. When DE is running on the CPU it is a bottleneck and for more than 10 runs the performance does not increase significantly. However, by running DE on the GPU, the bottleneck is removed and we achieve an average speedup of 27.4x and 33.1x with 10 and 20 runs respectively. Minimum and maximum speedup for docking a single ligand with 10 runs and both pose evaluation and DE running on the GPU is 7.7x and 37.5x respectively.

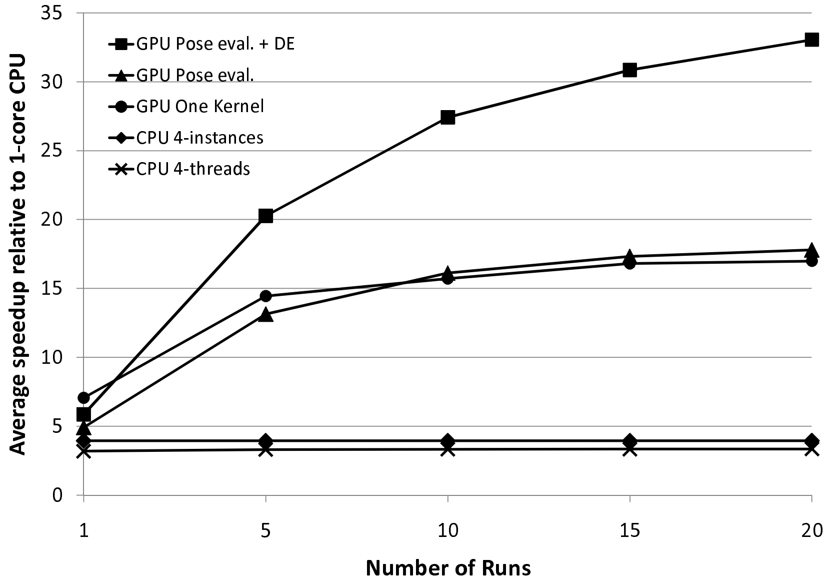


Fig. 7 Average speedups relative to the CPU implementation using one core when docking 133 complexes with different number of runs.

An alternative parallelisation scheme proposed in [6] was also evaluated. Here, the evaluation of poses is done in a single kernel where each thread block is assigned to a pose and each thread is assigned to an atom. The threads compute atom positions and interaction energies for the assigned atom and share the computation of torsional energies. The advantage of this approach is that all intermediate results can be cached in shared memory which minimises access to the global memory. The main disadvantage is that computations on rotational bonds cannot utilise all threads in a block. Fig. 7 shows the speedup achieved by this parallelisation scheme (GPU One Kernel) with DE running on the GPU. For a single run it performs well but as the number of runs increases, the GPU runs out of resources due to a large number of idle threads. In our parallelisation scheme, all threads have no or few idle clock cycles which improves the performance by 74% on average with 10 runs.

The average docking time with 10 optimisation runs for the 133 ligands is 0.77s on the GPU. This is fast enough to perform large virtual screening experiments in a reasonable amount of time. However, screening e.g. a million ligands would still take more than a week using a single GPU. We therefore plan to investigate different methods for increasing the performance even further. As seen in Fig. 7 the GPU used in these experiments is not fully utilised with 10 runs and the increase in accuracy is minimal when using more than 10 runs on both the CPU and GPU. Therefore, we need another way of increasing the utilisation of GPUs than by increasing the number of runs. It is questionable if the parallelisation granularity for MolDock can be increased and unlikely that this approach will scale well for GPUs with more than 100 cores. Instead, multiple ligands could be docked simultaneously, either into the same or different proteins, which would increase the number of thread blocks that are executed concurrently. Factors such as an increased load on the GPU cache and complex indexing could become a bottleneck in this approach but it seems to be the most promising direction for further research.

6 Conclusion

In this chapter, we have presented methods for parallelising flexible molecular docking with MolDock on both CPUs and CUDA enabled GPUs which are applicable to other flexible molecular docking methods. When docking multiple ligands into a target protein on a four core CPU, we achieved an average speedup of 3.3x with multi-threading and 3.9x by running multiple instances of the program concurrently. Using a comparable GPU with 112 cores, the average speedup was increased to 27.4x which translates to an average docking time of less than a second per ligand. The modifications made to the MolDock method did not decrease the accuracy on neither the CPU or GPU. The reduction in running time per ligand achieved with a cheap consumer GPU, reduces the cost and the amount of hardware needed to perform virtual screening with MolDock significantly.

References

1. Dagum, L., Menon, R.: OpenMP: An industry standard API for shared-memory programming. Computational Science and Engineering, IEEE **5**(1), 46–55 (2002)
2. Dorigo, M., Birattari, M., Stutzle, T.: Ant colony optimization. IEEE Computational **1**(4), 28–39 (2006)
3. Goodsell, D.S., Morris, G.M., Olson, A.J.: Automated docking of flexible ligands: applications of AutoDock. Journal of Molecular Recognition **9**(1), 1–5 (1996)
4. Goodsell, D.S., Olson, A.J.: Automated docking of substrates to proteins by simulated annealing. Proteins **8**(3), 195–202 (1990)
5. Halperin, I., Ma, B., Wolfson, H., Nussinov, R.: Principles of docking: An overview of search algorithms and a guide to scoring functions. Proteins **47**(4), 409–443 (2002)
6. Kannan, S., Ganji, R.: Porting AutoDock to CUDA. In: Computational Intelligence on Consumer Games and Graphics Hardware (2010)
7. Kirkpatrick, S.: Optimization by simulated annealing: Quantitative studies. Journal of Statistical Physics **220**(4598), 671–80 (1984)
8. Korb, O.: PLANTS: Application of ant colony optimization to structure-based drug design. Chemistry Central Journal **3**(Suppl 1), O10 (2006)
9. Korb, O.: Efficient ant colony optimization algorithms for structure- and ligand-based drug design. Dissertation, Universität Konstanz (2008)
10. Kozakov, D., Brenke, R., Comeau, S., Vajda, S.: PIPER: An FFT-based protein docking program with pairwise potentials. Proteins **65**(2), 392–406 (2006)
11. Matsumoto, M., Nishimura, T.: Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation **8**(1), 3–30 (1998)
12. Muegge, I., Rarey, M.: Small molecule docking and scoring. In: K.B. Lipkowitz, D.B. Boyd (eds.) Reviews in Computational Chemistry, chap. 1, pp. 1–60. Wiley (2001)
13. Nissink, J.W.M., Murray, C., Hartshorn, M., Verdonk, M.L., Cole, J.C., Taylor, R.: A new test set for validating predictions of protein-ligand interaction. Proteins **49**(4), 457–471 (2002)
14. Podlozhnyuk, V.: Parallel Mersenne Twister. Tech. rep., NVidia (2007)
15. Stone, J.E., Hardy, D.J., Ufimtsev, I.S., Schulten, K.: GPU-accelerated molecular modeling coming of age. Journal of Molecular Graphics and Modelling **29**(2), 116–125 (2010)
16. Storn, R., Price, K.: Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. Journal of Global Optimization **11**(4), 341–359 (1997)
17. Sukhwani, B., Herbordt, M.C.: GPU acceleration of a production molecular docking code. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2, pp. 19–27. ACM Press (2009)
18. Thomsen, R., Christensen, M.H.: MolDock: a new technique for high-accuracy molecular docking. Journal of medicinal chemistry **49**(11), 3315–3321 (2006)
19. Yang, J.M., Chen, C.C.: GEMDOCK: a generic evolutionary method for molecular docking. Proteins **55**(2), 288–304 (2004)