Virtual Docking using General Purpose Graphic Processing Units

Martin Simonsen - 20030596

October 7, 2009

1 Introduction

Even though modern processors continue to follow Moore's law[1] the performance of our applications is stagnating because of the memory, frequency and power walls [4]. The three walls prevent microchip manufactures in increasing the performance of new processors by simply increasing the frequency as they have done the last 50 years. Instead microchip manufactures have been trying to find alternative ways of increasing performance in the last few years. One popular solution is to increase the number of transistors in new processors by adding more cores. In theory two cores yield twice the performance of one core. But to utilise more than one core we need to write concurrent applications but because modern multi-core CPU architectures are based on single core architectures they lack features to support efficient concurrent applications such as fast inter-core communication.

There have been more or less successful attempts to design processors with better multi-core performance in the last couple of years, but recently attention have been focused on a processor architecture which have been present in most desktop PCs the last decade. The GPU (Graphic Processing Unit) has existed for more than a decade and is essentially a multi-core processor on steroids without the constraints of normal CPU's. Since GPU's are designed to handle graphics it used to be difficult to make use of them for solving more general problems. But with the introduction of NVIDIA's Compute Unified Device Architecture (CUDA) in 2007, General Purpose GPUs have made the immense computing power available in modern GPU's more available.

In this project I aim to apply GPU's to Virtual Docking (VD) which is a problem requiring both large amounts of raw computing power and sophisticated search algorithms. The goal is to implement a crude VD algorithm loosely based on Molegro's Molecular Virtual Docking software (MVD) [5] for scanning large databases for possible drug candidates. Using a GPU for computationally heavy operations I hope to achieve a significant speed-up compared to a similar CPU implementation.

2 General Purpose Graphic Processing Units

This section introduces the basics of NVIDIAs CUDA and the CUDA API which was used to implement the VD GPU implementations. All information was found in [1, 2].

2.1 CUDA

Figure 1 illustrates the CUDA architecture. A device corresponds to a GPU and if the device memory is included it corresponds to a graphic card. Each device has a number of multi-processors (normally between 2 and 60) that contains a number of processors (normally 8 per multi-processor). Thus the total number of processors on a device is between 16 and 240 which has to be kept busy to obtain maximum performance. Compared to standard desktop CPU's, GPU processors are quite slow (clocked typically at 500 to 1500 Mhz) and only designed to perform huge amounts of sequential computations efficiently whereas CPU's are designed to perform a wide variety of tasks. GPU's are not flexible enough to replace CPU's (yet) but some computational tasks can be offloaded

[1] The number of transistors doubles every 2 years.

GPU's if the tasks can be parallelized to run on hundreds of processors and does not require complex computations.

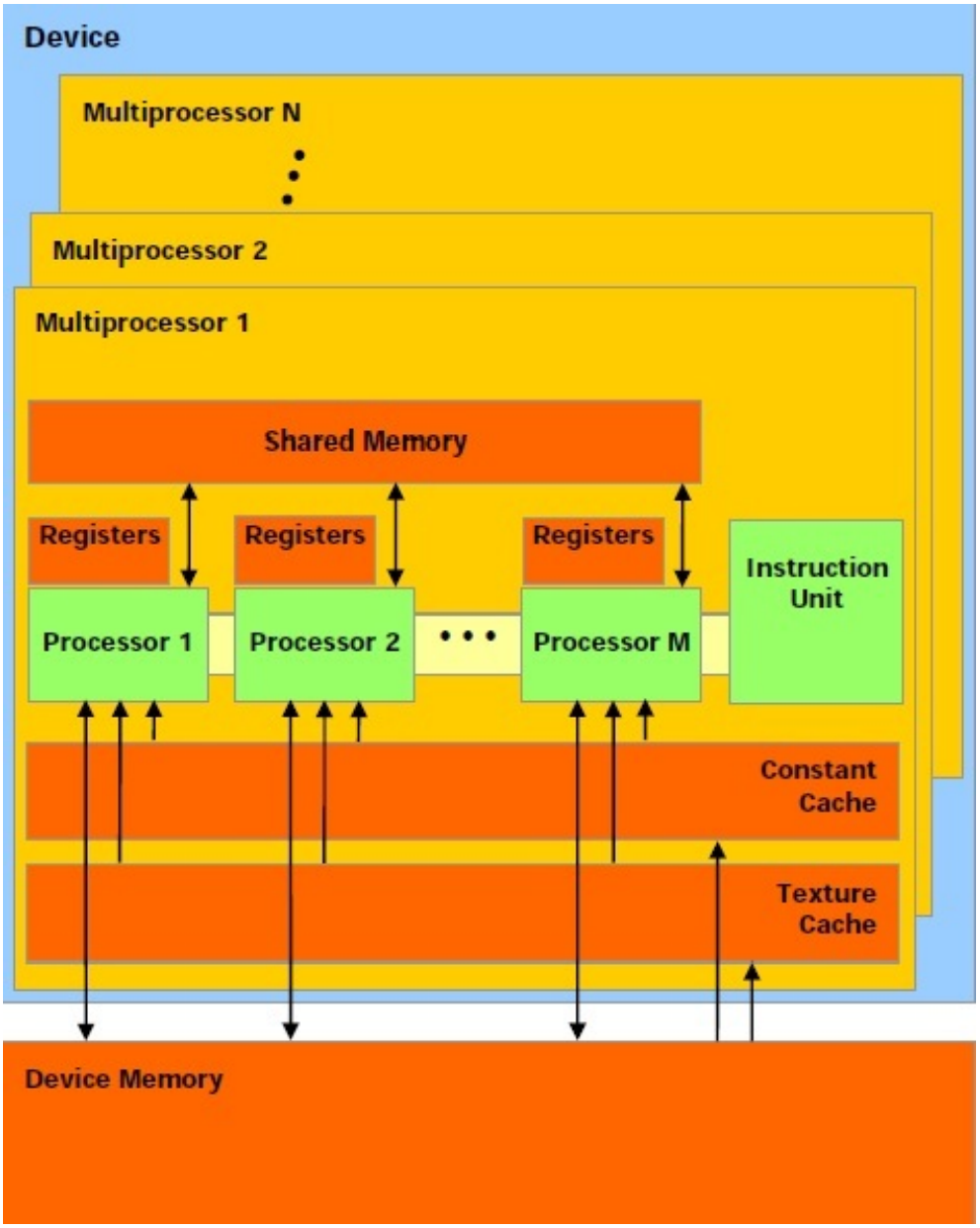

Figure 1: Illustration of the CUDA architecture

CUDA expose several different types of memory to the application programmer which are controlled by software to some degree. Each multi-processor has 8-16KB registers and 16KB of shared memory which are divided between the 8 processors. The main memory is called the "device memory" and is typically between 256MB and 1GB in size. The device memory can be allocated as one

of three types of memory, global, texture and constant memory. Global memory is not cached and is the simplest type of memory whereas both constant and texture memory have their own cache of size 6-8 KB per multi-processor. Constant memory is limited to a size of 64KB, while texture memory is just a special way of allocating global memory. Both registers, shared memory and caches are on chip and have very low latency. The device memory is accessed through a memory bus with a high bandwidth but also a high latency. Besides being cached, texture memory has several special properties such as the possibility to access allocated memory as a continuous 2D grid with automatic interpolation between neighbouring points performed by hardware in the GPU.

2.2 API

CUDA makes use of an architecture which is called Single-Instruction Multiple-Thread (SIMT). SIMT is a clever way of hiding both the Single Instruction Multiple Data architecture that each GPU processor implements and the Multiple Instruction Multiple Data (MIMD) architecture implemented by the multi-processors. Developers simply define a division of the workload using multi-threading scheme without worrying too much about the GPU architecture. Threads are executed in *warps* of 32 which execute the same instruction on different data (SIMD). If the threads in a warp does not agree on the execution path, the threads are serialised into as many warps as necessary. Different warps execute instructions independently (MIMD) so the execution path of a thread in one warp does affect threads in other warps. Threads are organised by the software into ”blocks” and blocks are organised into grids. Blocks have 1, 2 or 3 dimensions and can contain op to 512 threads while grids have 1 or 2 dimensions each with a maximum size of 65536. It is possible to create millions of threads and unlike normal CPU architectures, there is little overhead in handling a large number of threads on a GPU. Figure 2 illustrates the relationship between threads, blocks and grids.

Each block of threads is scheduled to run on an available multi-processor and up to 8 blocks can be executed concurrently on a single multi-processor where the block is split into warps that are also executed concurrently. Concurrent execution of blocks and warps helps hide idle clock cycles caused by thread synchronisation and memory latencies. Hiding memory latencies is important when threads are accessing uncached data in the global memory. Instead of waiting for data to arrive, multi-processors will start executing the next warp of threads if any warps are available.

But the number of blocks that can be executed concurrently on a multi-processor is often limited by the number of registers available and the amount of free shared memory. It is therefore important to minimise the use of these resources in each thread. To achieve a high performance there must be enough blocks to keep all multiprocessors fully occupied. For a standard GPU with around 16 multi-processors more than 3000 threads are typically needed to have a high utilisation all multi-processors.

CUDA kernels are written in a c-like language which support most features in the c-language. Even though the API tries to hide the CUDA architecture, there are several things which developers are forced to handle. The different types of memory can be accessed by threads in any order, but if the access pattern does not follow some specific patterns the performance degrade dramatically. In general, threads in a block should use a coalesced memory access pattern and avoid accessing the same element. One important exception to this rule is that all threads in a block can read the same element in shared memory efficiently. There are several other memory access patterns which are

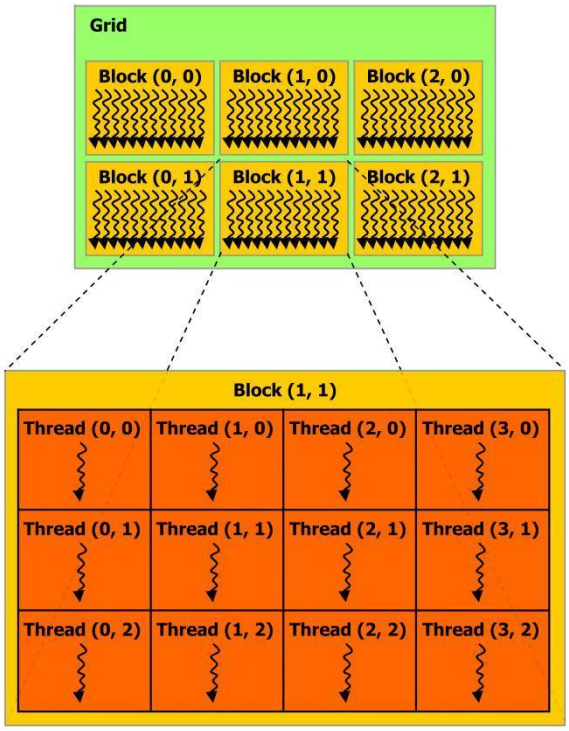

Figure 2: Illustration of the thread-block-grid hierarchy

also efficient depending on the memory type but these are beyond the scope of this report.

Designing algorithms such that each thread in a block takes the same path through branches is also important for performance. If one or more threads follow different execution paths there's a risk that it will hurt performance as warps needs to be serialised. In general CUDA kernel code should not contain too many branches and access to device memory should be kept at a minimum.

Thread synchronisation in blocks is handled efficiently though the GPU hardware using the CUDA API and threads in a block can easily share data using the shared memory. However, synchronising threads and sharing data between different blocks is expensive as it can only be done using global memory. The missing support for fast inter-block synchronisation is of course a hardware design choice that results in better performance for CUDA kernels that are completely parallisable, but it makes some simple tasks difficult to perform efficiently on a GPU, such as summing a vector of numbers (See Sec. 4.1).

3 Virtual Docking

Virtual screening is widely used in drug discovery research. By approximating the interaction between drug candidates and molecular targets *in silico* the use of expensive experimental method can be reduced. Virtual Docking (VD) is an important part of virtual screening where the optimal pose (conformation, position and rotation) of a ligand[2] and a target molecule (A large protein in my case)

[2] A small molecule which is able to bind to another (larger) molecule

is predicted *in silico* (See Fig. 3). Finding the optimal pose can be done in several different ways but it often involves maximising an optimising criteria by searching a huge parameter space.

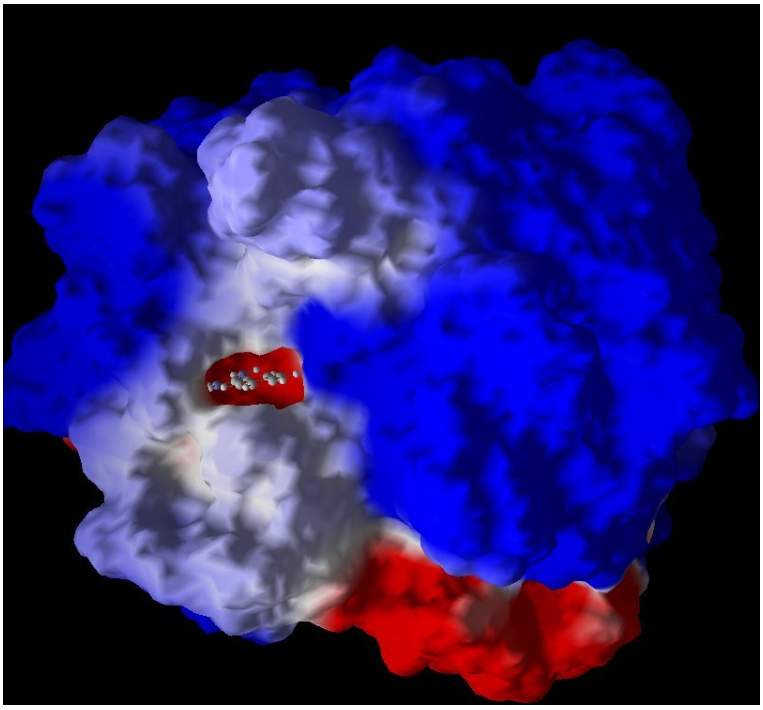

Figure 3: An illustration of a ligand docked in a cavity of a protein.

3.1 Finding the Optimal Pose

The docking process of ligands is highly complex and it is currently infeasible to take all parameters into account [3]. Instead of trying to simulate the docking process, most VD-methods search for poses where the most significant molecular forces have optimal values. This can be accomplished using a number of different approaches such as empirical methods and optimisation of force field scoring functions.

In force field based scoring functions, which I used in my implementations, the interaction between a ligand and target molecule is modelled as a combination of different force fields that approximate basic molecular forces such as Van der Waals forces and hydrogen bond energies. A widely used example of a force field scoring function is the Lennard-Jones (L-J) potential which approximates the potential energy between a pair of atoms. The L-J function is

$$V(r) = 4\varepsilon\left(\frac{\sigma}{r}^{12} - \frac{\sigma}{r}^{6}\right) \tag{1}$$

Where r is the distance between the two atoms, σ the distance at which the potential is zero and ε the depth of the potential well. The r^{12} term describes Pauli repulsion which is short range repulsive forces caused by overlapping electron orbitals. The r^6 term describes the long range attractive forces such as Van der Waals forces. When using the L-J potential in VD the goal is to find a pose that

minimise the potential corresponding to a binding configuration where the ligand is affected by strong attractive forces and weak repulsive forces.

In rigid docking both the target molecule and ligand have a fixed form determined by e.g. X-ray crystallography. Using a force field scoring function the goal is to find a position and rotation of the ligand which optimises a scoring function. A good value of the scoring function corresponds to a strong interaction between the ligand and target molecule and could imply that the ligand is able to affect the function of the target molecule, i.e. cure a disease. Today rigid docking can be solved using a systematic search approach as there's only 6 degrees of freedom. However, rigid docking ignores many important factors where the flexibility of ligands is one of the most important. An optimal pose is often located in cavities of target molecules (see Fig. 3), and if flexibility is ignored docking algorithms are often unable to fit ligands in cavities with a different shape than the ligand. Flexible ligands introduce several extra degrees of freedom which makes the naive exhaustive search approach infeasible.

Docking where flexibility of the molecules is taken into account is called flexible docking and is standard today. Flexibility of molecules can be modelled by allowing some single bonds to rotate i.e. the dihedral angels are changed. To handle the large parameter space that flexible docking introduces, we need an intelligent way to navigate the search space.

Efficient systematic search methods for flexible docking are available. Instead of performing an exhaustive search of the entire parameter space, the ligand is divided in smaller pieces which are then docked separately. The docked ligand is then found by combining compatible docked pieces in a final step. Libraries of pre-generated ligand conformations can also be used to reduce flexible docking to rigid docking.

Stochastic search methods is a popular way of overcoming the large search space in VD [5, 3]. These methods use randomised search algorithms to quickly navigate the search space and can often find high scoring poses very quickly.

Monte Carlo(MC) methods is a type of stochastic methods used in VD. Here random poses are generated from the search space and then scored using some scoring function. The scored poses are then compared to a set of previously saved poses, after which the set of saved poses is updated to contain the best scoring poses. How the search space is sampled and how new poses are evaluated differs but most MC methods follow this basic algorithm.

Genetic algorithms is another type of stochastic search methods which is also the type of search algorithm used in MVD. A genetic algorithm starts by sampling a number of random poses and scoring them. The best poses are then combined into new poses in a way which is analogous to gene recombination and mutation. In this way good poses pass on their parameters to the next generation with some degree of randomness. The new poses are then evaluated and the best are allowed to replace old poses.

3.2 Representation of Data

An atomic representation is a simple way of representing molecular data where each molecule is represented as a set of atoms and bonds. This representation is mostly used in conjunction with force field energy functions but usually only in the final optimisation stages of docking algorithms as this representation requires the energy between all pairs of atoms in the ligand and target molecule to be computed which can be computationally expensive [3].

Energy grids is an representation which is closely related to the atomic representation. Here an energy grid replaces the target molecule (which is usually the largest). At each point in the grid the total potential energy contribution of each atom in the target molecule is stored. Evaluation of a pose can now be done by reading the energy in the grid point closest to each atom in the ligand or by interpolating the three closest grid points. Energy grids are significantly faster than a pure atomic representation but cannot be used for flexible target molecules.

Surface representations are also used in some VD implementations. Here both molecules are represented as a 3D surface which is defined by e.g. Van der Waals forces or the ”Accessible Surface Area”.

4 Virtual Docking on a GPU

MVD use a score function based on two simple stepwise linear potentials (force field scoring functions) which model Van der Waals forces potential (steric potential) and hydrogen bond potential (see Fig. 4). Each potential roughly corresponds to a term in the L-J potential but is faster to compute. For each atom-pair one potential is used depending on the type of atoms. For atom-pairs capable of forming a hydrogen bonds, the hydrogen bond potential is used, otherwise the Van der Waals potential is used.

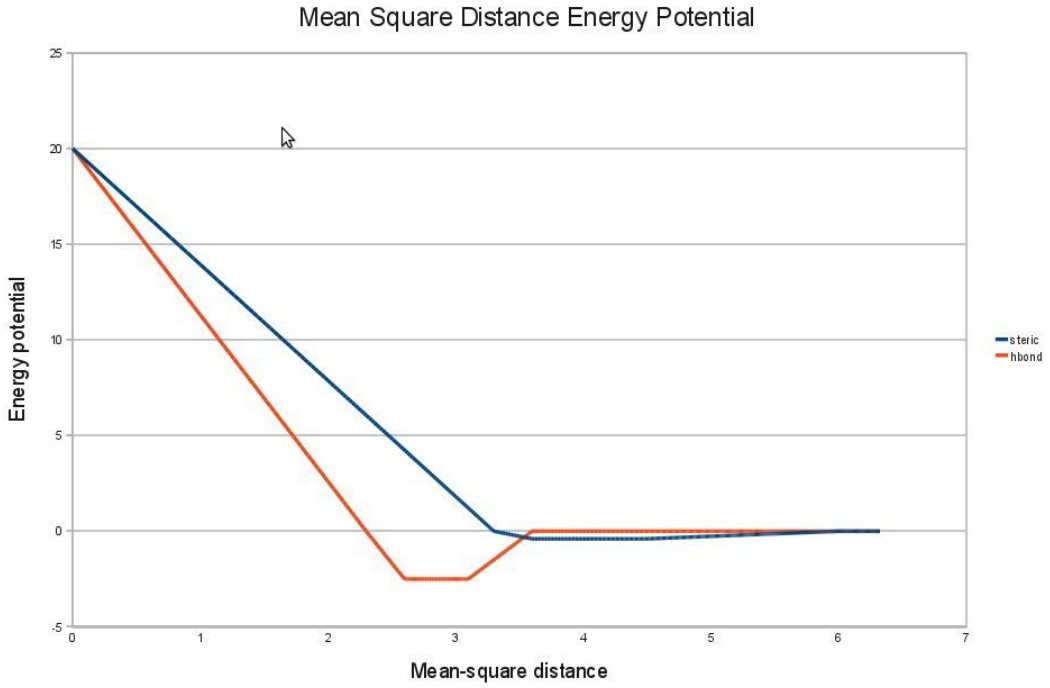

Figure 4: Illustration of the energy function used in MVD software.

The initial idea was to used a GPU to compute the score function using an atomic molecule representation. The typical problem size is between 10 to 50 atoms for ligands and 1-4000 atoms

for target molecules resulting in tens of thousands evaluations of the scoring function for each pose. In MVD this is done using an energy grid that reduces computational complexity considerably. However, computing the score function using an atomic representation seemed as a good way of learning GPU programming and investigate GPU vs CPU performance.

4.1 Computing the Scoring function

I implemented the scoring function in a GPU kernel which was called by a CPU application. The CPU ran a simple systematic search algorithm where the ligand was translated a fixed number of times in a small 3D area close to the target molecule. Initially all atoms in the target molecule were copied to the GPU global memory. Translation of the ligand was done on the CPU and after each translation all ligand atoms were copied to the GPU global memory.

1D thread blocks with 128 threads was used to compute the potential energy between 128 atoms of the target molecule and all atoms in the ligand molecule. A 1D grid was used to create enough thread blocks to handle all atoms of the target molecule. The position of a thread in a block and the position of a block in the grid was used to uniquely assign a target molecule atom to each thread. To prevent each thread in a block in loading all ligand atoms from the global memory, ligand atoms were stored in shared memory. I also experimented with using both constant and texture memory for the target molecule atoms to take advantage of the cache associated with these types of memory, but I could not register any change in performance. The reason for this is probably that most target molecules contains more than 1000 atoms which are accessed sequentially and only once for each kernel execution. With only 6-8KB of cache and an atom size of 16 bytes this results in a series of cache misses as not all atoms can fit in the cache.

When a thread finished scoring all assigned atom-pairs the result was stored in shared memory. Using a parallel reduction algorithm each thread block summed the score of all threads within the block and stored the total score in global memory. Because there is no built in synchronisation between blocks, the total score of all blocks are computed on the CPU by copying the score from each block from the GPU memory to the CPU memory.

4.2 Computing the Score of Multiple Ligands and Target Molecules

Computing the score of a single ligand and target molecules only give rise to 10-20 thread blocks which is not enough to keep all multi-processors in a GPU fully occupied because of memory latencies. Computing the score of multiple ligands and target molecules increases the number of blocks significantly and allows the GPU to fully utilise all multi-processors.

Initially I implemented a kernel which could handle multiple ligands by creating a 2D grid of thread blocks and using the position of each thread block in the grid to determine which ligand to compute. Sadly CUDA does not support 3D grids which I planned on using for handle multiple target molecules. Instead I used a 2D grid with dimensions x, y where $x = \lceil \frac{\#target\_atoms}{128} \rceil$ and $y = \#ligands * \#targets$. Each thread could then determine which ligand and target molecule it was assigned to by knowing the size of all molecules. The score function was computed as described above.

4.3 Searching the Parameter Space

It is not hard to reduce the computational complexity of the score function by using an energy grid, and after some discussion with Molegro I found that the algorithm used for searching the parameter space might be just as time consuming to compute as the score function. MVD use differential evolution as their search algorithm which is a type of genetic algorithm. The default max population size in MVD is 50 poses and after each iteration a number of new poses are created from the old population. Creation of new poses is similar to scoring the poses in terms of computational complexity and using GPU's to improve performance of this step could be interesting. I did not have time to implement a GPU enabled version of MVD, so I tried to implement some simple search algorithms instead.

4.3.1 Systematic Search

The first search algorithm used a systematic approach on a single ligand and target molecule. I implemented a simple rotation algorithm that rotated the ligand around the x, y and z-axis. and defined a search space as a 3 dimensional box around a cavity in each target molecule identified by MVD. All combinations of positions and rotations was then searched using a fixed resolution of both translations and rotations.

The first implementation used the GPU to do most of the work as follows. For each point in a 2D plane formed by the x and y axis and some resolution, a thread block was created. Each thread block translated the ligand to the assigned position and scored all rotations in that point. A pose was scored by assigning an equal number of protein-ligand atom-pairs to each thread and using a parallel reduction algorithm to compute the total potential energy of all threads in a block. The highest scoring pose was then copied to the CPU memory and the CPU computed the best pose from the result of all blocks. By launching a GPU kernel for each point on the z-axis the whole parameter space was searched.

There were two problems with this approach. First of all the kernels took a long time to finish if the rotational resolution was too high, resulting lock up of the system because the GPU was unable to handle calls from the OS. Secondly, the number of variables required to keep track of the current state of each thread was too large. A variable corresponds to using a register in a multiprocessor which is a limited resource. If too many registers are used, multi-processors cannot process enough thread blocks concurrently to hide memory latencies resulting in loss of performance.

To solve these problems I moved the search of rotations from the GPU to the CPU. Now the CPU rotated the ligand and copied the rotated ligand to the GPU memory. Similar to the above algorithm, the GPU computed the score of the rotated ligand in a 2D plane for each point on the z-axis. This approach had a much better performance and because the CPU only needed to run through all rotations once and not once for each point in the 3D search area, most of the workload was still assigned to the GPU.

4.3.2 A Simple Hill Climbing Search Algorithm

Even when using a fast GPU for a systematic search of the parameter space, it is still very time consuming and not feasible for a large number of ligands. I did not have time to implement an

advanced search algorithm that took advantage of a GPU, such as a genetic algorithm, so I went for a simple hill climbing algorithm.

My hill climbing algorithm works as follows. Initially a ligand with a random position and rotation is created. The ligand is then in turns translated one step along the tree axis in both directions using a suitable resolution. At each position the score of all possible rotations is computed and the best saved. The ligand is then translated to the position where the best score was found. By repeating these steps the ligand slowly moves to a local optima. This algorithm is of course very inefficient compared to algorithms like gradient decent and the simplex algorithm but it is easy to implement on a GPU.

Several runs are needed to locate a global maximum but because runs are independent, it is easy to parallelise the algorithm and utilise a GPU. In a CPU-only implementation each run was performed sequentially but could easily have been parallelised using multi-threading (which I didn't have time to do).

A GPU enabled implementation was implemented as follows. The CPU was used to create an initial population of ligands with random positions and rotations. For each ligand in the population 6 thread blocks were created, one for each possible translation of the ligand. The blocks were placed in a grid with dimensions $\{\#ligands, 6\}$ providing a straightforward way of assigning a workload to each block. Because all ligands in the population were processed in parallel all multi-processors on the GPU could be fully utilised if the population were sufficiently large. The CPU was used to rotate the ligands, finding the best score for each ligand and translate the ligand in the direction with the best score.

In the CPU implementation the algorithm ran until the potential energy increase (ΔE) dropped below a threshold of $0.5cm^{-1}$. The resolution of rotations was increased by a factor 1.5 when ΔE dropped below $2.0cm^{-1}$ the first time to increase the precision of the final poses. The GPU implementation used the same threshold to terminate the algorithm but here the algorithm ran until $\Delta E < 0.5$ for all ligands in the population. When ΔE of a ligand dropped below $2.0cm^{-1}$ the first time for a ligand, this ligand was removed from the population. When the population size reached 0 the first time, all ligands were reinserted into the population and the rotation resolution increased by a factor of 1.5.

5 Results and Discussion

To compare the running times and precision of the CPU and GPU implementations I used some data sets provided by Molegro as test data. A comparison of my implementations with another VD software have been omitted as my implementations are too simple to be competitive. I did compare some pose with those found by MVD by comparing the poses visually using MVD, but as MVD use flexible docking my poses were usually far from the poses found MVD. However, if the ligands had very few rotatable bonds, my implementation were able to find poses close to the ones found by MVD.

5.1 Experimental Setup

The test system consisted of a quad core Intel core 2 system containing a GeForce 8800GT graphics card with 12 multi-processors. The CPU is a high-end CPU whereas the GPU is a mid/low-end GPU.

5.2 Results: Computation of the Score Function

Figure 5 and Fig. 6 shows the running time of the algorithm computing the MVD step-wise linear potential. When only a single ligand and protein was evaluated, the GPU achieved a 5.5x speed-up. By scoring multiple ligands in parallel the GPU could be utilised better which resulted in a 16x and 9x speed-up compared to a single-threaded and multi-threaded CPU implementation respectively. When scoring multiple ligands and multiple proteins an even larger workload is available for the GPU and a 22x speed-up was achieved compared to a single-threaded CPU implementation (12x speed-up with a multi-threaded implementation using all 4 cores).

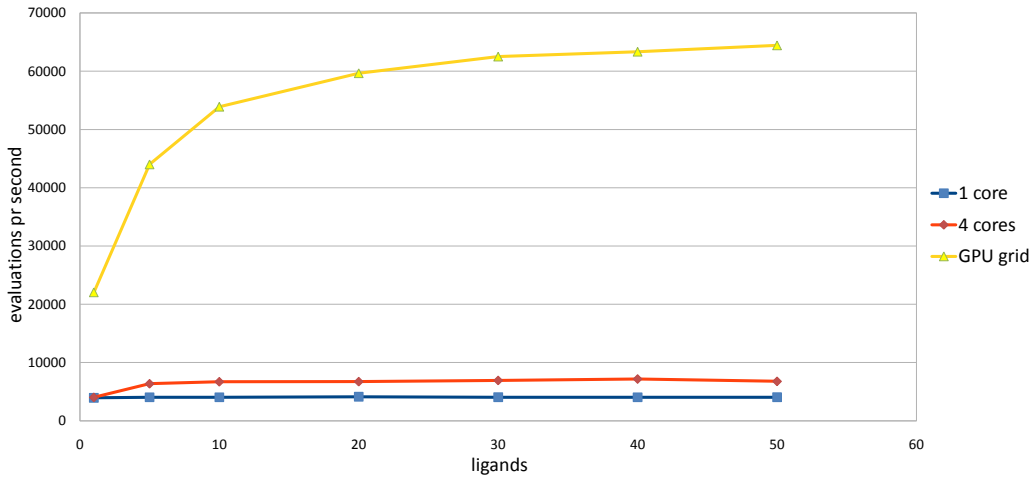

Figure 5: Evaluations per second of a ligand containing 46 atoms and a protein containing 1514 atoms using the MVD scoring function. The ligand was copied a number of times and each copy was scored against the protein 8000 times

It is clear that the GPU can be used to efficiently evaluate poses when using an atomic representation. Given more time it would be interesting to implement an energy grid representation of molecules and see if a similar speed-up could be achieved using a GPU. In MVD a grid with some fixed resolution is constructed once and then a simple 3D linear interpolation is used to estimate the value any point between the grid points. Because CUDA GPU's can perform 3D linear interpolation in hardware a significant speed-up could be achieved if the evaluation of the energy function was moved to a GPU.

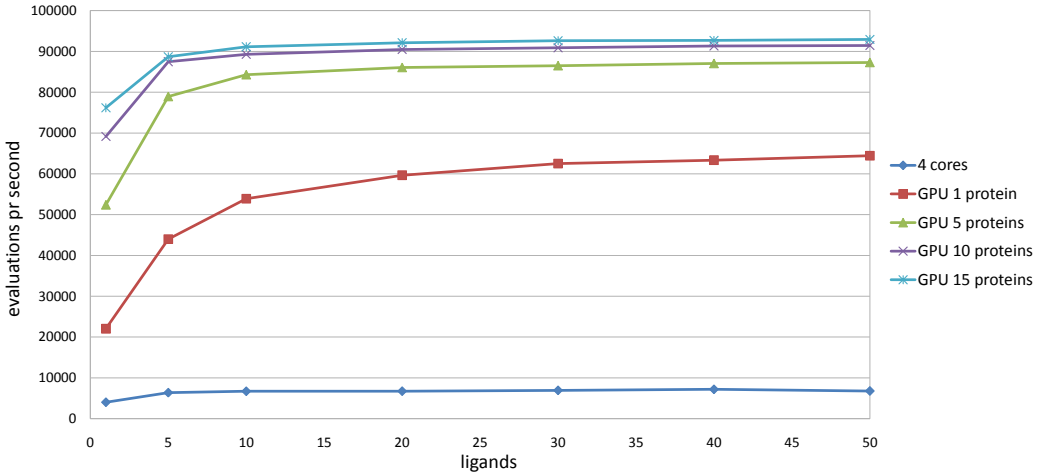

Figure 6: Evaluations per second of a ligand containing 46 atoms and a protein containing 1514 atoms using the MVD scoring function. Both the ligand and protein was copied a number of times and each pair of ligand and protein was scored 8000 times.

5.3 Results: Systematic Search and Hill Climbing

Two data sets of different sizes were used to evaluate the search algorithms. For each data set a near optimal pose is known and was used as the center of the search space. The search space was defined as a cube with dimensions 10Å x 10Å x 10Å. The resolution of translations and rotations was set to 0.25Å and $\frac{\pi}{3}$ respectively.

The result of experiments with the systematic search algorithm is shown in Table 1. The GPU implementation achieved a 26x speed-up compared to a single-threaded CPU implementation but I suspect that a multi-threaded CPU implementation could reduce this speed-up to under a factor 10 as systematic search is easily parallelised. Still, these results show that GPU's can be used to increase the performance of VD applications significantly if the workload is large and the computations relatively straightforward.

Arc.	Protein size	Ligand size	Running time	Score	Speed-up
CPU	3218	18	7h 3m	-165.537 cm^{-1}	-
GPU	3218	18	16m 5s	-165.537 cm^{-1}	26x
CPU	1741	31	6h 40m	-158.986 cm^{-1}	-
GPU	1741	31	15m 12s	-158.986 cm^{-1}	26x

Table 1: Results of experiments with the systematic search algorithm.

Table 2 show the result of experiments with the hill climbing algorithm with a population size of 50 ligands. The GPU enabled implementation achieved a 17x and 19x speed-up on the two data sets compared to the single-threaded CPU implementation. Compared to the systematic search

approach, the hill climbing algorithm has similar precision while being much faster to compute. The population size of 50 might not be enough if the search space is increased. But as both the CPU and GPU hill climbing implementations have running time $O(n)$, where n is the population size, the population size can easily be increased whereas it would become infeasible to use the systematic search implementations on a larger search space.

Arc.	Protein size	Ligand size	Running time	Best score	Speed-up
CPU	3218	18	7m 21s	-174.293 cm^{-1}	-
GPU	3218	18	25.5s	-176.819 cm^{-1}	17x
CPU	1741	31	8m 24s	-159.202 cm^{-1}	-
GPU	1741	31	26.5s	-158.081 cm^{-1}	19x

Table 2: Results of experiments with the hill climbing search algorithm.

It is not hard to improve my hill climbing algorithms by e.g. taking larger or smaller steps in the best direction and making it possible to move in more than 6 directions. However, implementing GPU kernels is quite time consuming and it difficult to analyse if a given approach will yield a better performance. I plan to continue with this project and implement a GPU enabled version of the genetic search algorithm used in MVD as some parts of algorithm seems to be well suited for GPU's. A GPU enables version of MVD is interesting even for a modest speed-up but I also aim to design a stripped down version of MVD which could be used for scanning large data bases.

6 Conclusion

Because GPU's continue to improve their practical performance as opposed to CPU's that currently only improve their theoretical performance, using GPU's seems to be one of the best ways of improving performance of VD algorithms and thereby enabling faster screening of the constant growing drug candidate databases. Constructing a GPU enabled VD algorithm proved to be time consuming and somewhat difficult. I succeeded in implementing two simple rigid docking algorithms which could find good poses using a simple force field scoring function in a small search space. My experiments showed that GPU's can be used to improve performance of VD algorithms if parts of an algorithm could be sufficiently parallised. However, because GPU's have many limitations compared to modern CPU's only the parts of a algorithms which contain few branches and requires a small state can be efficiently computed on a GPU. So one of the major challenges in designing GPU enabled algorithms is to only use techniques that can be massively parallelised and implemented using straightforward computations.

References

[1] *NVIDIA CUDA C Programming Best Practices Guide*.

[2] *NVIDIA CUDA Programming Guide 2.2*.

[3] Douglas B Kitchen, Hélène Decornez, John R Furr, and Jürgen Bajorath. Docking and scoring in virtual screening for drug discovery: methods and applications. *Discovery*, 3, 2004.

[4] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's*, 18(3), 2005.

[5] René Thomsen and Mikael H Christensen. Moldock: a new technique for high-accuracy molecular docking. *Journal of medicinal chemistry*, 49:3315–21, June 2006.