# We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,900
Open access books available

185,000
International authors and editors

200M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities




WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

## Interested in publishing with us?
## Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com



Chapter 6

# Hardware Accelerated Molecular Docking: A Survey

Imre Pechan and Béla Fehér

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/48125

## 1. Introduction

Hardware acceleration is the general concept of applying a specialized hardware for a given problem instead of an ordinary CPU in order to get lower processing time. General purpose CPUs can be considered as a totally general platform suitable for executing virtually any software or algorithm. Application specific accelerators have a custom architecture that fits the needs of a certain family of algorithms. As a consequence, they are able to outperform CPUs by orders of magnitude in a special application area but they are unfit for other, more general tasks. In contrast to normal CPUs, which are essentially serial machines executing instructions sequentially, hardware accelerators use parallel architectures which allow them to exploit the parallelism available in the given application by performing independent operations simultaneously.

The most important examples of hardware accelerators are graphics processing units (GPUs) and field-programmable gate array devices (FPGAs). GPUs are special many-core processors optimized for 3D rendering and image processing purposes. GPU devices are nowadays part of any desktop PC configurations and they can be programmed with general purpose programming languages. These facts make them an easily accessible and cost-effective accelerator platform and explain why they are used more and more frequently even in applications that are not graphics-related (general purpose GPU programming). FPGAs are programmable logic devices consisting of hundreds of thousands of general logic elements whose interconnection can be configured by the user. Thus FPGAs have a highly flexible architecture that allows to implement a totally custom digital hardware without the enormous cost of designing and manufacturing an application-specific integrated circuit (ASIC). When using an FPGA as a hardware accelerator a custom logic device is realized in the FPGA whose only purpose is to execute the algorithm to be accelerated as effectively as possible; thus the algorithm is usually implemented as pure hardware instead of software.

INTECH
open science | open minds

© 2012 Pechan and Fehér licensee InTech. This is an open access chapter distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Hardware accelerators such as GPUs or FPGAs are utilized in many scientific applications, when the time-consuming operations make it impractical or even impossible to use ordinary CPUs. Bioinformatics is not an exception; it includes many problems and algorithms which are computationally expensive due to the large amount of data to be processed or the complex operations involved. Typical examples are different sequence alignment algorithms, protein structure prediction algorithms and molecular dynamics simulations which were implemented on various accelerator platforms several times.

Molecular docking is another key field of bioinformatics whose purpose is to determine the binding geometry of molecules and is used by the pharmaceutical industry for identifying drug candidate compounds. Docking algorithms are usually computationally demanding since they consist of generating and evaluating a large amount of different molecule conformations and placements. However, these different placements can often be processed simultaneously and evaluating a single placement usually offers further parallelization possibilities. These facts make molecular docking an ideal target for hardware acceleration. In accordance with this, several GPU- and FPGA-based docking implementations were reported applying different approaches for hardware acceleration. In this chapter our purpose is to give a general overview of the most interesting implementations and to compare them with respect to the applied parallelization, applicability and achieved speedup. The remainder of this chapter is organized as follows. Section 2 surveys the concept and methods of molecular docking. Section 3 gives a general overview of FPGA and GPU devices. Section 4 and 5 introduce the existing FPGA- and GPU-based docking implementations, respectively. Finally, Section 6 surveys the current state and perspectives of hardware accelerated molecular docking.

## 2. Overview of molecular docking

Molecular docking is a computer simulation technique for determining the possible binding position and binding energy of molecules whose initial 3D spatial structure is known. Many docking methods and software exist, which may be different in several respects such as the size and number of molecules involved, the applied docking algorithm, the applied chemical model or the modeling of molecular flexibility.

Molecular docking usually refers to docking a molecule to another one, that is, to determine the binding pose of the former relative to the latter. In case of protein-protein docking both of the molecules are large macromolecules. The more typical case is the protein-ligand docking when one of them is a small ligand molecule whose binding pose needs to be determined within the active site of a receptor. Since the computational complexity (the number of atoms, the size of the search space, etc.) differ by orders of magnitude, protein-protein and protein-ligand docking usually require different approaches. Although the number of molecules involved in the docking problem is generally two, some protein-ligand docking software allow to dock more than one ligand to a macromolecule simultaneously. For some software a good starting position has to be provided manually which is then

refined by the algorithm; other ones are totally automated and try to find the docked position without any a priori knowledge.

Another important aspect is how the docking algorithm takes into account molecular flexibility. Rigid-body docking methods keep the structure of the molecules rigid, flexible algorithms consider one or both of the molecules flexible allowing their conformation to change. The two approaches correspond to the lock-key and the induced fit model, respectively. Rigid docking methods are usually much faster but may easily fail to find the proper binding position in case of molecules that actually undergo a conformational change upon binding. The most obvious way to model flexibility is to consider some bonds rotatable by allowing their torsional angle to change during docking. This method is effective in case of small ligands, but greatly increases the number of degrees of freedom and the computational complexity of the docking problem when applied for a large protein. As a consequence, protein flexibility is often taken into account only partially (allowing a few bonds of some side chains to rotate) or is modeled differently. One example is the soft receptor technique which allows small atomic collisions between neighboring protein and ligand atoms by reducing the repulsion energy term. The method is based on the assumption that the highly flexible protein could avoid the collision in practice by a low energy conformational change. Modeling flexibility in this way is computationally economic but may easily lead to invalid docked positions. Another straightforward technique is to keep the protein structure rigid and repeat the docking process with different pre-generated (or experimentally determined) protein conformations. Ultimately, this enables taking into account both protein and ligand flexibility even in case of rigid-body docking methods. The approach is also useful for considering the flexibility of rings within the ligand, which cannot be modeled with rotatable bonds; instead, a set of pre-generated, valid substructure conformations can be used during docking.

Although there are numerous different molecular docking algorithms, essentially each of them consists of two important components: a scoring function and a search method. The scoring function represents a chemical model and usually estimates the free energy of a geometrical arrangement of the molecules, thus it scores the given placement. The search method tries to find the ideal arrangement by sampling the search space according to a strategy. Docking can be viewed as an optimization problem where the global optimum of the scoring function is to be identified and the degrees of freedom are the variables describing the position, orientation and conformation of the molecules. Some docking methods apply one of the standard force fields as scoring function such as AMBER or CHARMM [1-4]. Other ones use empirical scoring functions that consist of a sum of terms representing different interaction types between the molecules; the term types are weighted with values determined empirically from a set of protein-ligand complexes [5, 6]. Knowledge-based functions are also typical which are derived from the statistical analysis of a large database containing molecular structures [7, 8]. The search methods applied by the different docking methods are also very diverse. One example is incremental reconstruction applied by the docking tools DOCK [4] and FlexX [9], which split the ligand to be docked and place the fragments one-by-one at the binding site. AutoDock [6] and GOLD [10] use

genetic algorithms as global optimization methods. AutoDock Vina [11] applies a quasi-Newton BFSG algorithm along with Monte Carlo simulation. Other standard algorithms such as simulated annealing, tabu search or particle swarm optimization techniques are also common. A good overview of the general terms and concepts of molecular docking can be found in references [12-14].

The most important application area of molecular docking is computer-aided, structure-based drug design. Docking can be used for identifying drug candidates (potential inhibitors) for a given target receptor molecule. During virtual screening the members of a large ligand database are docked one by one to the target; promising compounds are subjected to further experiments. Virtual screening is extremely time-consuming; accelerating it can make the drug design process more effective. Trivially, this can be done by executing the docking runs of different molecules in parallel utilizing a lot of CPU cores. The other method is to accelerate the applied docking algorithm itself, potentially by an FPGA- or GPU-based hardware accelerator.

## 3. Accelerator platforms

### 3.1. FPGA devices

A field-programmable gate array is a programmable logic device - an integrated circuit with a flexible hardware architecture that can be configured to implement a specific functionality. FPGAs represent a trade-off between highly flexible, general purpose microprocessors and high-performance application-specific integrated circuits (ASICs). FPGA devices execute the required computation with a specific hardware architecture just like ASICs. Although they are not as efficient in terms of performance and power consumption, implementing a custom hardware in a 100-1000$ FPGA does not require to manufacture a new chip which is affordable only in case of large-scale production. In addition, FPGAs can be reconfigured many times. Thus they can be considered general-purpose similarly to CPUs but due to the applied custom architecture they can be orders of magnitude faster in case of a specific application.

The two major FPGA vendors, Xilinx and Altera offer a wide range of FPGAs and FPGA families with different capabilities, the performance and complexity of the devices is also continuously growing; however, the basic architecture remains the same. FPGA devices consist of a large number of similar basic logic blocks or cells arranged usually in rows and columns on the chip and a configurable interconnect structure. Figure 1. shows a simplified diagram of the basic logic block (slice) of a Xilinx Virtex-4 FPGA. The slice consists of two 4-input LUTs (look-up tables), two D flip-flops, carry logic supporting chaining of neighboring slices for high-performance arithmetic operations and routing resource configurable by multiplexers. A 4-input look-up table is a simple $2^4$=16 bit memory element that can realize any four-variable logic functions when initialized with the truth table of the corresponding function. D flip-flops are 1-bit registers that capture and store the value of the D input at every active CLK clock edge. Thus LUTs are the basic resources of the FPGA for implementing combinational logic and D flip-flops for sequential logic, respectively. In

addition to the general logic resources FPGAs usually include special purpose cells such as dedicated memory blocks or DSP (digital signal processing) blocks consisting of adders and multipliers for arithmetic-intensive applications. FPGA-based accelerator cards are usually equipped with high-capacity external memory modules and high-speed interfaces like PCIe in addition to the FPGA.

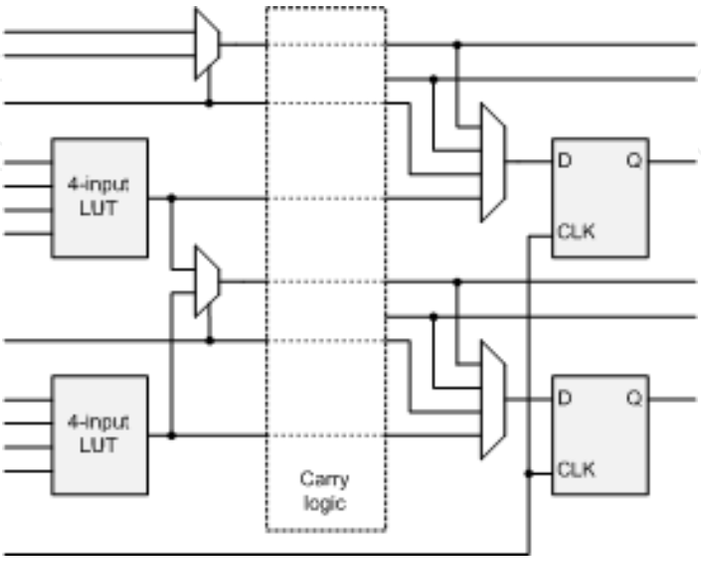



**Figure 1.** Virtex-4 slice

FPGA devices have an inherently parallel architecture which makes them suitable for high-performance computing applications. Different parts of an algorithm are executed by different hardware elements or modules; the execution can be simultaneous if the operations are independent. In data-parallel applications, where the same steps need to be performed on different data elements, the data can be distributed among many identical processing elements in the FPGA. In this case the achievable parallelism is limited only by the capacity of the device and the speed of the interface providing the input data. Another typical design concept is to apply a pipeline consisting of serially connected stages, which execute different steps of the same algorithm on different independent data elements.

Implementing an algorithm in an FPGA instead of a CPU may lead to a much shorter execution time; however, it usually requires more programming time and effort. The FPGA configuration can be defined with hardware description languages (HDL) such as VHDL and Verilog. HDLs allow the designer to describe the operation and interconnection of general digital circuits at a relatively high level (called register-transfer level). The HDL description is then mapped to the FPGA architecture by automatic tools. Further information regarding FPGA architectures, programming languages and design methodologies can be found in references [15-16].

## 3.2. GPU devices

Graphics processing units are massively parallel processors consisting of hundreds of processing cores, thus capable of executing hundreds of threads in parallel. Their

architecture is optimized for data-parallel applications, which consist of instructions that have to be carried out on many different data elements. GPU operation is akin to the SIMD (single instruction multiple data) behavior – the parallel threads execute the same code but process independent input data. There are two main GPU manufacturers, AMD and NVIDIA, and although there are differences between the GPU architectures, the basic concepts are very similar. The same is true for the two widely used programming languages, CUDA and OpenCL. The former is developed by NVIDIA and is applicable to NVDIA devices only. OpenCL in turn is a standard parallel programming language supporting not only both GPU architectures but also multicore CPUs and heterogeneous platforms in general. The remainder of this section gives an overview of NVIDIA GPUs and CUDA since this is used by the majority of the GPU-based molecular docking implementations introduced in Section 5. However, the basic methodology and design patterns are very similar in case of OpenCL, only the terminology differs.

CUDA (Compute Unified Device Architecture) is the computing architecture of NVIDIA GPUs, which defines a parallel programming model based on high-level programming languages. CUDA C gives minimal extensions to the standard C language and provides an API, which enable the user to write a CUDA program consisting of serial code and special parallel functions called kernels. The former runs on the host CPU, the latter are executed K-times parallel by K different CUDA threads on the GPU. Threads of a kernel are grouped into thread blocks; the blocks in turn form a grid. Threads within the same block can communicate and synchronize with each other. This is not possible between different blocks of threads, since these are scheduled and executed in a random, non-deterministic order based on run-time decisions. This leads to automatic scalability; among ideal circumstances a GPU with twice as many processing cores can execute the same kernel twice faster.

The simplified hardware architecture can be seen on Figure 2. An NVIDIA GPU consists of multiprocessors. Each multiprocessor includes several processing cores, a large amount of registers, shared memory and a scheduler. In addition, each multiprocessor can access the external memory and has caches for texture and constant data access. When a kernel is launched, a certain number of thread blocks is assigned to every multiprocessor and becomes active. A multiprocessor executes its active blocks logically in parallel, and it manages, schedules and executes the threads of its active blocks in groups of 32 threads called warps. Warps are executed physically in parallel, that is, a multiprocessor is able to execute the same operation of every 32 thread within a warp simultaneously in one or a few clock cycle. However, if threads of a warp take different execution paths after a conditional branch statement, the different instructions get serialized, that is, they are executed sequentially (warp divergence).

Keeping the number of active blocks and warps high is important since this helps keeping every multiprocessor of the GPU busy as well as since the scheduler can hide the instruction and memory access latencies by switching between active warps. The maximal number of blocks that can be active on a multiprocessor is limited by the register and shared memory usage of the block since these resources are split among the active blocks. On the other

hand, internal register and shared memory access is very fast. Threads can access their own registers in parallel; shared memory is divided into banks, and can be accessed also in parallel, as long as parallel threads access different memory banks. This suggests that data should be stored in registers and shared memory whenever possible. External memory access is much slower, but if threads of a warp read from or write to a contiguous memory space, the memory operations can be coalesced and executed as a single access, which can greatly increase the effective memory bandwidth. Constant data access is faster than ordinary memory read operations since it is cached. All of the aspects mentioned above have to be taken into account when choosing data storage areas, grid and block sizes. Further information about GPU architectures and programming can be found in references [17-19].

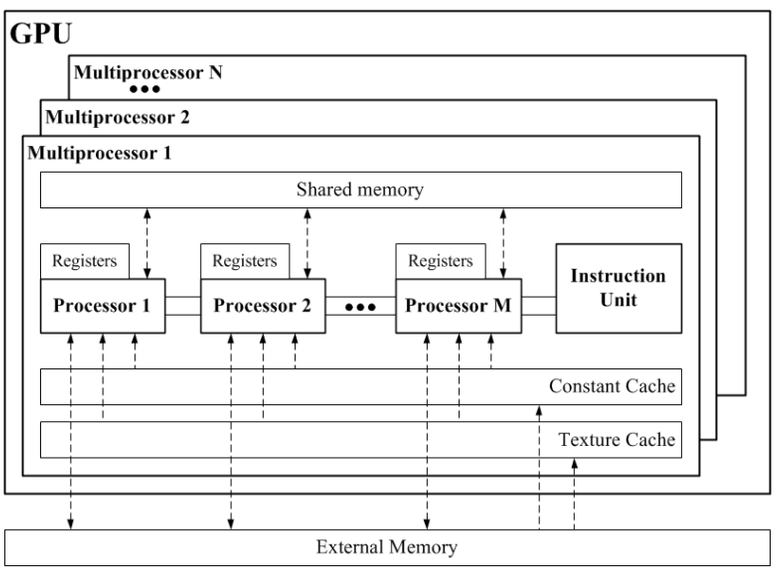



**Figure 2.** NVIDIA GPU architecture

## 4. Molecular docking on FPGA platforms

We believe that there are only three FPGA-based docking implementations which have been published until now. This chapter introduces all of them: a docking engine using 3D correlation, its successor, the FPGA-based implementation of the PIPER [20] docking program, and the FPGA-based acceleration of AutoDock.

### 4.1. Docking with 3D correlation

This implementation is described in references [21, 22]. The applied algorithm uses 3D correlation which is a common rigid-body docking technique. The molecules to be docked are represented with 3D grids whose voxels consist of pre-calculated values expressing some property of the molecule at the corresponding spatial location related to the binding affinity. In order to evaluate an arrangement the two grids are shifted relative to each other, then the voxels are multiplied pairwise and the values are summed to get the final score. By calculating the whole correlation array every possible translational position is evaluated.

This process has to be repeated for each orientation to be investigated, which requires the rotation of one of the grids periodically. In case of correlation-based docking methods the applied search method is essentially exhaustive search. Obviously the molecules are treated rigid during docking, since their structure is hard-coded in the grids.

The CPU-based docking programs using correlation usually replace it with Fourier transformation (FFT) and multiplication, which can be much faster on serial machines. The described FPGA-based implementation, however, performs direct correlation which can be effectively implemented with a highly parallel systolic chain in the FPGA. Another advantage of this method is that, by avoiding FFT, the operation for determining the voxel-voxel interaction is not restricted to multiplication; even non-linear functions can be used. In order to exploit this the implementation has a flexible structure; the design can be easily configured to adapt different scoring schemes. The initial implementation [21] used a very simple voxel type consisting of only two bits that distinguish molecule interiors from exteriors and mark the surface of the molecules. The final version allows using voxels with tuple data type that represent different effects including directional interactions like hydrogen bonding.

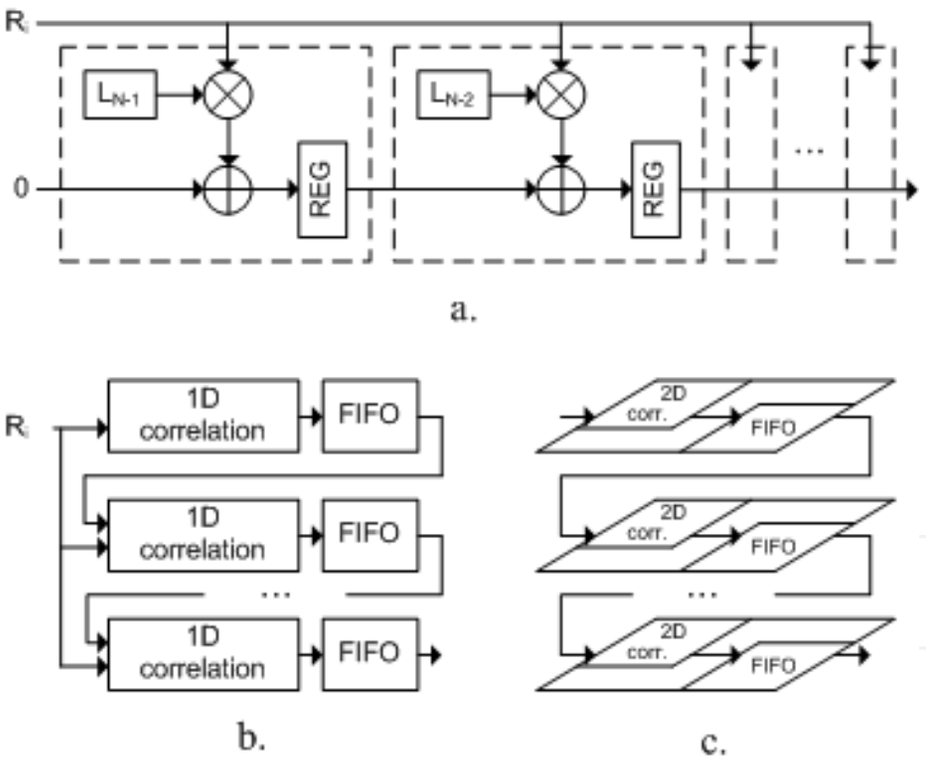



**Figure 3.** Systolic 3D array architecture [22, 23]

The core element of the implementation is a systolic 3D correlation array consisting of cells. Each cell stores one voxel of the grid corresponding to the smaller (ligand) molecule. The receptor grid is stored in external memory. Instead of rotating the ligand at the beginning of each new correlation cycle, rotated orientation is obtained by reading the receptor voxels in rotated order. Thus rotation is performed on the fly by the address logic and uses only little

of the FPGA resources. The receptor voxels read from the memory are also rotated in case of directional data types; then they are passed to the systolic array. Figure 3/a shows a 1D correlation array consisting of pipelined cells. Each cell executes a pairwise operation (in this case, a multiplication) defined by the scoring method on its ligand voxel ($L_i$) and the receptor voxel ($R_i$) available at its input, then adds it to the sum received from the previous cell and stores the result in a register. 1D correlation arrays are connected by FIFO delay lines to form a 2D correlation plane; planes in turn are connected by delay planes to obtain the 3D correlation array (Figure 3/b and 3/c). Due to the pipeline-based structure, the systolic array produces the result of one position evaluation (correlation) in each clock cycle, and the achieved parallelism is proportional to the number of ligand grid voxels. Due to the large amount of output data the resulting grids are not sent to the host machine directly. Instead, a data reduction filter module detects the best score (local maximum) within each subblock of the correlation array; only these promising docked positions are returned for further analysis. Certain parts of the FPGA design are configurable according to the applied voxel word with and type, score data type, and the applied pairwise scoring operation in order to support various force laws and scoring schemes.

Performance tests were carried out with a Xilinx Virtex-II Pro XC2VP70-5 FPGA. Results were compared to a software running on a 3 GHz Xeon CPU. The software applied direct correlation since FFT proved to be slower for the applied small problem sizes. FPGA speedup varied between ×100-1000 according to the scoring method used.

### 4.2. PIPER on FPGA

The docking engine described in reference [23] is a modified, extended version of the docking core introduced in Section 4.1 and implements the PIPER software [20]. PIPER is based on 3D correlation and calculates it with the standard FFT method. The scoring function of PIPER consists of the weighted sum of different terms represented with separate grids; as a consequence, several independent forward FFTs have to be performed during evaluation. In addition to the van der Waals repulsion and attraction terms and the electrostatic interaction, desolvation effect is taken into account as well. The latter is described by a pairwise potential which is transformed to correlation grids with eigenvalue-eigenvector decomposition. Grids corresponding to low eigenvalues are often discarded which reduces computational complexity but retains the accuracy of the algorithm [20].

The original implementation described in Section 4.1 was modified in a variety of ways to support multiple energy grids as well as to allow docking of larger molecules which would not fit in the systolic 3D array, thus permitting even protein-protein docking. The basic cell element of the systolic array is extended to process the independent grids in parallel; as a consequence, each correlation is performed simultaneously. At the end of every 1D correlation array a new weighted scorer module sums the partial correlation results with respect to the weights defined in PIPER. New FIFOs are used to propagate the output of a scorer module to the input of the next one. Calculating the weighted sum at the end of every

1D array requires a balanced amount of multipliers and FIFOs (block RAMs) of the FPGA keeping the resource utilization optimal. To support large molecules both the receptor and the ligand are stored in external memories. The ligand grid is partitioned into subgrids small enough to fit the size of the correlation array. Correlation is performed piece-wise; each subgrid is first loaded to the FPGA, then the receptor voxels are streamed through the array.

The docking engine was implemented on an Altera Stratix-II EP2S180 and was validated against the original PIPER software. FPGA performance, however, was determined with post place-and-route simulations supposing an Altera Stratix-III EPSL340. Performance was compared to the original PIPER code, its multithreaded version, as well as a GPU-based implementation of PIPER (introduced in Section 5.1). The host CPU was a quad-core Intel Xeon 2 GHz CPU, the GPU code run on an NVIDIA Tesla C1060 device. The measured FPGA speedup depended greatly on the ligand grid size. In case of a $4^3$ ligand grid speedup of the correlation task only and that of the whole application was almost ×1000 and ×37, respectively, compared to the single-core PIPER. However, it dropped exponentially with respect to the ligand size, decreasing below the ×16 speedup of the GPU at grid edge size 16 and below the ×3 speedup of the quad-core version at grid edge size 32. The reason for this is that the FFT method applied by the CPU and the GPU became greatly superior to direct convolution at this problem sizes.

### 4.3. AutoDock on FPGA

References [24, 25] introduce our own FPGA-based docking implementation, the acceleration of the AutoDock [6] docking software. AutoDock is applicable basically for protein-ligand docking and models molecular flexibility with rotatable bonds. AutoDock uses a semi-empirical scoring function that consists of weighted terms representing van der Waals and electrostatic interactions, hydrogen bonding and desolvation. The scoring function gives the energy contribution of one non-bonded atom pair; this value has to be summed over all movable atom pairs of the system to determine the score. To reduce computational complexity AutoDock represents the rigid part of the receptor molecule with pre-calculated potential grids. Thus the energy contribution of a given ligand atom and the whole receptor can be determined with trilinear interpolation and iterating over the receptor atoms is not necessary. AutoDock uses a standard genetic algorithm (GA) as search method. Genetic algorithms generate sets of potential solutions (generations of entities) iteratively. Solutions are represented with values of the degrees of freedom (called genes) and are created by combining the genes (crossover) of selected previous entities (selection) and altering them randomly (mutation). In addition to the genetic algorithm AutoDock subjects some selected entities of each generation to an iterative local search method (LS) similar to hill climbing, which greatly increases the effectiveness of the algorithm.

AutoDock was implemented on the SGI RASC RC100 module on a Xilinx Virtex-4 LX200 FPGA. The design consists of four main blocks organized as a three stage pipeline (Figure 4). The first pipeline stage executes the genetic algorithm, that is, it generates the genes of a

new entity periodically. The second stage calculates the positions of the atoms of the ligand based on the input gene values. This step consists mainly of performing atomic rotations according to the positions of rotatable bonds and to the orientation of the ligand. The third stage includes two modules. One of them determines the receptor-ligand interaction energy based on the potential grids stored in external memory, that is, it performs a trilinear interpolation for each ligand atom; the other one calculates the energy contribution of each movable ligand atom pair by evaluating the scoring function directly. Each of the four modules consists of massively parallel, fine-grained internal pipelines; as a consequence, all of them are able to produce a new result of the realized operation in each clock cycle. The first module generates a new gene value, the second one performs the rotation of an atom, the other ones calculate the interaction energy of a ligand atom and the receptor molecule or that of an internal ligand atom pair in every clock cycle.

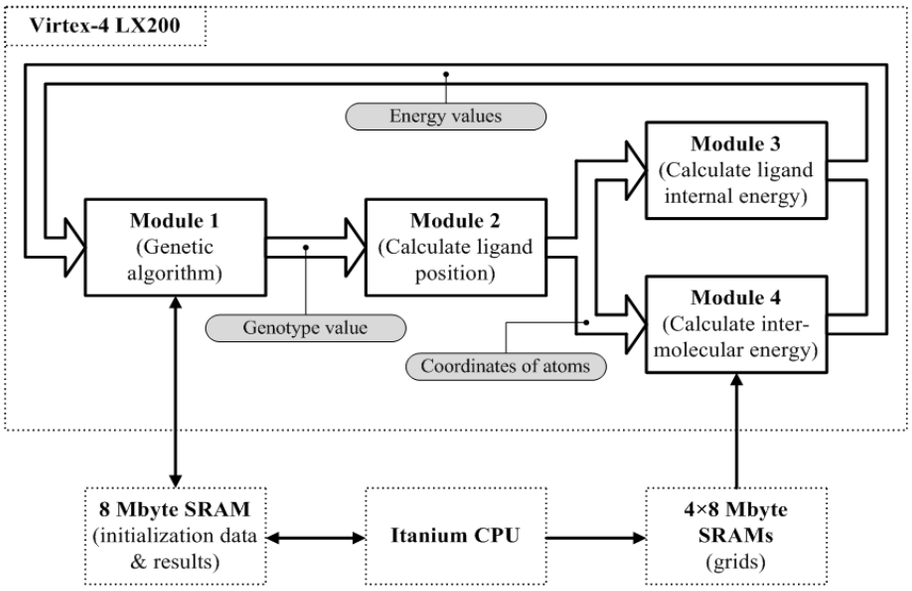



**Figure 4.** FPGA core implementing AutoDock [24]

In order to increase the performance of the docking engine the implemented algorithm slightly differs from the original AutoDock code and uses fixed-point arithmetic that fits better the FPGA architecture. According to test runs these differences does not degrade the accuracy of docking. Performance tests showed that the FPGA-based implementation yields an average speedup of ×23 over AutoDock running on a 3.2 GHz Intel Xeon CPU; the actual speedup varied between ×10-40 according to the structure and size of the molecules.

## 5. GPU-based implementations

Compared to the relatively small number of FPGA-accelerated docking engines, quite a lot of GPU-based solutions have been reported, which clearly indicates the advantages of GPUs over FPGAs in terms of accessibility and programming effort. It is neither reasonable nor possible to introduce every one of them. Instead, we aim at describing a wide variety of different approaches, and we tried to select the most promising implementations. Two of

the docking codes introduced in the following subsections were implemented also on FPGA.

### 5.1. PIPER on GPU

The authors of the FPGA-based PIPER (Section 4.2) published also a GPU-based version [26]. In case of the FPGA the FFT applied in PIPER was replaced with direct correlation, which can be executed by a very effective standard structure in the FPGA. On GPU, both FFT and direct correlation were implemented and they proved to be advantageous at different ligand grid sizes. Other steps such as summing the grids and filtering the results by identifying local maxima also run on the GPU; although they comprise only a few percent of total PIPER runtime, executing them on the CPU would have limited the achievable speedup. The only exception is re-calculation of the ligand grid according to the current orientation and charges, which run on the host CPU.

3D correlation includes a lot of parallelism which can be exploited on a GPU as easily as on FPGA. Each voxel of the result grid can be calculated by a different thread simultaneously; in addition, correlation of grids representing different terms can be performed in parallel. In this implementation two different approaches are applied whose performance turned out to be similar: assigning each 2D plain to a different tread block, and assigning the same part of each 2D plain to a thread block. Receptor grid is stored in the external memory of the GPU due to its size and since it has to be available for each thread block (multiprocessor). Ligand grid is stored in shared or constant memory if possible; if the grid is small enough grids corresponding to multiple ligand orientations are stored and processed in parallel, which leads to further performance improvement.

Forward and inverse FFT is executed with the standard NVIDIA CUFFT library consisting of optimized FFT-related CUDA functions. Receptor grids are calculated by the CPU, moved to the GPU memory and transformed by the GPU only once at initialization. Ligand grids are re-calculated, copied and transformed for each ligand orientation. Voxels of the transformed receptor and ligand grids are multiplied pairwise by the GPU. The CUDA implementation is trivial, since each voxel pair can be processed independently by a different thread. Finally the product grid is inverse transformed.

The final step of each orientation evaluation is to sum the result grids according to the PIPER coefficients and find the voxels with the best scores corresponding to the best translational poses. PIPER uses several different sets of weights; these are assigned to different thread blocks in the GPU. Each block performs averaging according to the given set of weights. Individual threads process different parts of the grid. During averaging each thread identifies the best score of the grid part assigned to it and stores it in shared memory. Finally a single thread iterates over the scores to find the best one. Clearly the last filtering step could be implemented on the GPU the less effectively; if the number of coefficient sets is less than that of the multiprocessors, certain processors are not utilized, and serial steps such as finding the very last best score leads to idle threads. The majority of the algorithm, however, suits well the GPU architecture.

Performance tests were carried out on the platforms already mentioned in Section 4.2; the CUDA code run on a Tesla C1060 GPU and was compared to the FPGA-based version and to PIPER running on a single core and on all the four cores of a 2 GHz Intel Xeon CPU. Speedup of the correlation task was about ×300 compared to the single core version at a minimal ligand grid size of 4, but decreased exponentially with respect to ligand size similarly to the FPGA-based implementation. FPGA speedup was about ×1000 in case of a $4^3$ ligand grid, so in case of direct correlation the FPGA outperformed the GPU. The FFT-based GPU code achieved a speedup of about ×30 regardless the ligand size and proved to be faster than GPU-based direct correlation above ligand grid size $8^3$. Worst-case speedup of the whole GPU application was ×17.7 and ×6.1 versus single core and quad-core PIPER, respectively, and was faster than the FPGA accelerated version if ligand grid size was above $8^3$.

## 5.2. A general FFT-based approach

In reference [27] another CUDA implementation is presented that applies FFT for performing the correlation-based rigid docking algorithm. The approach is very similar to the one described in Section 5.1. The scoring function is very simple; it consists of two terms which represent the shape of the molecules and the electrostatic field. These terms are calculated over the 3D grid for the receptor and for each orientation of the ligand. Again, FFT is executed with the CUDA library.

The test environment consisted of a dual-core AthlonX2 3600+ CPU and an NVIDIA GeForce9800GT GPU. The GPU speedup proved to be about ×3-4, depending on the grid size and the angle step size between different ligand orientations. That is, for the same search space size a finer discretization of the grids (meaning higher number of grid voxels) and a finer discretization of the ligand orientation (leading to more different orientations to be evaluated) resulted higher speedup. The reason is that in this case the FFT-grid multiplication-IFFT steps became more dominant compared to the whole GPU algorithm, and these can be executed the most effectively on the GPU.

The achieved GPU performance seems to be lower with respect to the GPU-based PIPER (Section 5.1). Although the applied algorithms and implementation methods are similar, the achieved speedups are hard to compare due to the different hardware platforms. The GeForce 9800 GT includes about half the number of multiprocessors than Tesla C1060. CPU frequencies are the same but the architectures are very different; the applied AMD CPU is older than the Intel used in case of PIPER. The other possible explanation of the different performance improvements is that in case of PIPER several grids has to be processed during docking, which leads to more parallelism and requires more FFT computation; thus the advantages of the GPU can be exploited more effectively.

## 5.3. AutoDock on GPU

AutoDock is one of the best-known docking software; it was the most cited docking program in the ISI Web of Science database in 2005 [28]. This explains why it is a popular

subject for GPU-based acceleration. There is even a related SourceForge project called gpuautodock. The following subsections focus on three different AutoDock implementations; each of them maps different parts of the original algorithm to the GPU architecture.

### *5.3.1. Acceleration based on profiling*

This AutoDock implementation is described in a case study [29]. The authors followed a traditional way – they profiled the original code in order to identify the most time-consuming functions and ported only these to GPU. Two functions were selected – eintcal() and trilininterp() – which together accounted for about 63% of the total runtime. The former calculates the internal energy of the ligand molecule, that is, it evaluates the scoring function for each ligand atom pair whose distance can change due to rotatable bonds. The latter is called for each ligand atom during the calculation of receptor-ligand intermolecular energy to perform interpolation based on the pre-calculated potential grids.

Each time these functions are called the corresponding CUDA kernel is executed instead of the original function. In both cases the number of threads within the kernel equals to the number of ligand atoms. This molecule usually consists of a few tens of atoms, which is a very low number compared to the GPU capabilities leading to a poor GPU utilization ratio. In addition, before each kernel call some data is transferred from the main memory to the GPU according to the current ligand position; these frequent memory transfer operations further decrease the performance.

According to test runs, which were executed on an NVIDIA GeForce GTX 280 GPU, the GPU accelerated application could not achieve speedup but was slower than the CPU for typical ligand sizes. Performance improvement was obtained only if the number of atoms (threads) was in the range of $10^4$, which is not a realistic use case. The reasons are mentioned above. Accelerating only a few computationally expensive functions without restructuring the original code is straightforward and does not require much programming effort; however, it does not allow to exploit all the parallelism available in the algorithm, and also limits the maximal achievable speedup according to Amdahl's law.

### *5.3.2. Acceleration excluding local search*

AutoDock includes further parallelism that is not exploited by the implementation described in Section 5.3.1. It uses a genetic algorithm as search method, which can be parallelized easily; each entity of the next generation can be created and evaluated simultaneously by different processing cores. The default population size is 150, which makes this approach promising with respect to GPU-based acceleration. However, AutoDock also applies an iterative local search method in addition to the GA, which is executed only on a few percent of the population (6%, that is, averagely 9 entities by default). Executing local search of different entities in parallel is possible, but would lead to low GPU utilization. In addition, performing the local search algorithm on an entity may

consist of hundreds of iterations (energy evaluations); that is, executing the whole LS on CPU would greatly reduce the achievable speedup.

To overcome this problem, the authors of reference [30] chose to exclude the local search from the algorithm, but implement virtually every other part of AutoDock (the genetic algorithm, the ligand position calculation and the scoring function evaluation) on the GPU. Although the genetic algorithm (generating the degrees of freedom according to the GA rules) is usually not time-critical, leaving it on the host CPU would require periodic CPU-GPU memory transfer operations, which is avoided if it is executed by the GPU.

The main idea behind the implementation is to assign a different thread block to each entity of the new generation, whose threads cooperatively execute the different steps on the given entity. Another scheme was also tried where different entities were assigned to different threads, but this leads to low GPU utilization in case of typical population sizes – using default size, the number of parallel threads would be only 150 instead of 150 multiplied by the thread block size. The coordinates of ligand atoms are stored in the fast shared memory, which is crucial since every step of the scoring function evaluation modifies or reads this data and each thread of the block has to access it. During evaluation, each thread block first determines the atom positions (using two kernels for calculating the ligand conformation and orientation). Independent rotations of different atoms can be executed by different threads of the block. Then each thread performs trilinear interpolation for a different ligand atom (determining the atom-receptor intermolecular energy), and each thread evaluates the scoring function directly for a different ligand-ligand atom pair. Trilinear interpolation offers a further optimization, since NVIDIA GPUs support the fast access of 3D data by hardware.

Parallelization of the GA operators (selection, crossover and mutation) is also straightforward. Selection requires to calculate the relative score (fitness) of the entities compared to the average score, which can be performed for each entity simultaneously. Genes of the new entities can be generated by crossover and mutation in parallel by threads of the block assigned to the entity.

The test platform included an AMD Athlon 2.4 GHz and an NVIDIA Tesla C1060. Validation of the CUDA code was performed by using the same random seeds and comparing the output to that of original AutoDock. The results differed slightly only due to the single precision arithmetic applied in the GPU. The speedup of the different kernels depended highly on the population size. At default size speedup of the scoring function evaluation proved to be ×50, the selection and crossover ×1.25 and ×2.75. In case of mutation no speedup was obtained. The overall speedup of the algorithm was ×10 for a population size 50, it increased to ×20 for the default size and become saturated at 10000 yielding a speedup of ×47 over the CPU. On one hand, the GA operators could be implemented on the GPU much less effectively than the fitness evaluation. The probable reason is that they are much more control-intensive than the different steps of the scoring function evaluation consisting of a lot of arithmetic operations. On the other hand, executing GA on the CPU

would certainly decrease the speedup due to the additional transfer operations between the CPU and GPU memory.

### *5.3.3. Acceleration including local search*

Implementing AutoDock on CUDA without local search as in Section 5.3.2 clearly offers a straightforward parallelization scheme that avoids GPU underutilization. However, the local search process usually increases docking accuracy of AutoDock significantly [31]. In order to include the LS in the implemented algorithm and simultaneously achieve a high speedup we ported AutoDock to CUDA exploiting a further high-level parallelization possibility [25]. Due to the heuristic nature of the search algorithm, often several (10-100) different docking runs are performed with AutoDock for the same receptor-ligand complex. This increases the reliability of the results as well as helps identifying multiple valid docked poses. Since these docking runs are totally independent from each other, they can be executed in parallel.

Our implementation includes two CUDA kernels. In each generational cycle, first Kernel A is launched that creates and evaluates a whole population; then Kernel B is launched for performing LS on the selected entities. The two kernels call the same CUDA functions for scoring function evaluation; they differ only in how the degrees of freedom are generated (using either GA or LS rules). Basically, our implementation is quite similar to the one introduced in Section 5.3.2. Each thread block of the kernels is assigned to a different entity. Threads within a thread block generate different gene values, calculate independent rotations, and process different ligand atoms or atom pairs during scoring function evaluation. However, in case of kernel A a thread block is launched for each new entity of every independent docking run; in case of kernel B a block is launched for each entity of every run which is selected for local search.

The advantage of this method is that local search is included which allows preserving docking accuracy, and even significant performance improvement can be achieved if the number of independent runs is high enough. The performance improvement, however, depends strongly on this number and in case of too few parallel runs the GPU is underutilized during LS, which leads to a low speedup.

Test runs were carried out on an NVIDIA GeForce GTX 260 GPU; performance was compared with that of AutoDock running on a 3.2 GHz Intel Xeon CPU. In case of only one docking run the GPU achieved a low, ×2-5 speedup depending on the ligand structure and size. In case of 10 and 100 independent runs the average speedup proved to be ×30 and ×65, respectively, for a large set of ligands.

Our FPGA-based AutoDock implementation described in Section 4.3 achieved an average speedup of ×23. This value does not depend on the number of docking runs since the FPGA executes only one at a time. Due to the applied three stage pipeline (Figure 4) only three entities are processed simultaneously in the FPGA. On the contrary, the GPU applies a brute force approach by processing each entity of every run in parallel. The low level (per

rotation, per atom, etc.) parallelization possibilities are exploited by fine-grained pipelines in the FPGA very effectively; this allows the FPGA-based implementation to achieve a significant speedup regardless the number of runs. As a consequence, the FPGA is faster than the GPU for a low number of runs. Further advantage of the FPGA architecture is that implementing local search is not problematic. However, if the number of runs is high enough, the GPU outperforms the FPGA; that is, similarly to the FPGA and GPU-based PIPER (Section 4.2 and 5.1) the two platforms are advantageous at different parameter ranges.

## 5.4. MolDock on GPU

Reference [32] describes the GPU-based acceleration of the MolDock [33] docking software. MolDock is very similar to AutoDock: it models molecular flexibility with rotatable bonds, its scoring function consists of the summation of pairwise energy terms, it uses pre-calculated potential grids for representing the receptor during docking and it applies a genetic (evolutionary) algorithm as search method. Differences are the actual form of the energy terms (which is virtually irrelevant from the point of view of parallelization) and the lack of local search.

Due to the similar algorithms the basic implementation schemes are practically the same as the ones described in Section 5.3.2 and 5.3.3; the gene values and atoms of every entity are distributed among the threads and are processed in parallel. Although no local search process is used, independent docking runs are performed in parallel to increase GPU utilization ratio (like in Section 5.3.3). Due to these similarities the implementation is not described here in more details. However, we would like to emphasize an apparent difference regarding how different jobs are aligned to the threads of the kernel.

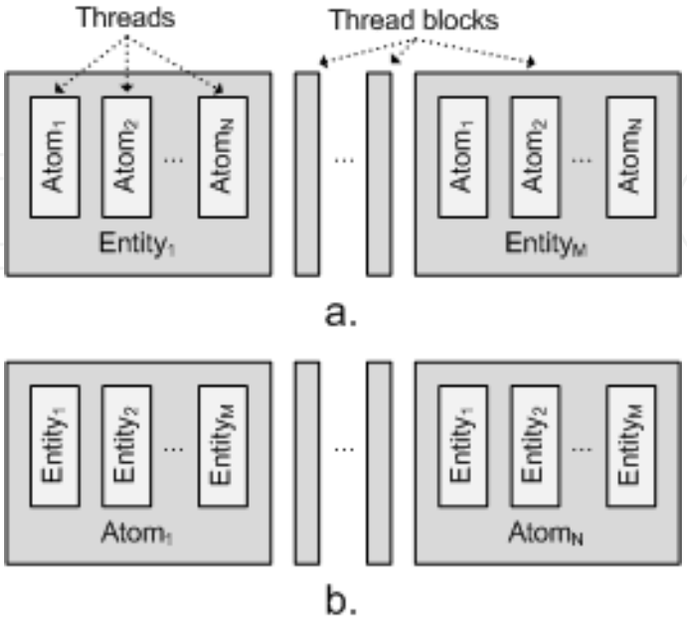



**Figure 5.** Job alignment comparison

In the GPU-based AutoDock implementations each thread block processes a different entity. In case of receptor-ligand energy calculation, for example, threads within the block perform trilinear interpolation for different atoms of the same ligand orientation (Figure 5/a). On the contrary, in this implementation threads within the same block perform interpolation for the same ligand atom of different entities (orientations) (Figure 5/b). Parallelization of other steps (genetic operators, internal energy calculation, etc.) also follows this scheme. This makes orientation calculation more effective; its disadvantage is that data corresponding to a given entity has to be stored in external GPU memory.

Performance tests were carried out using a 2.66 GHz Intel Core 2 Quad CPU and an NVIDIA GeForce 8800 GT. Average GPU speedup was ×5, ×27 and ×33 for 1, 10 and 20 parallel docking runs, respectively. The speedup, that is, GPU utilization showed a similar saturating tendency as in case of our GPU-based AutoDock implementation (Section 5.3.3).

### 5.5. PLANTS on GPU

PLANTS [34] stands for Protein-Ligand ANT System; it is a docking software using ant colony optimization (ACO) as search method. ACO is an optimization technique that mimics the behavior of ants when they collectively found the shortest path between the food source and the nest. At initialization, the degrees of freedom of the problem are discretized, and the same probability (pheromone level) is assigned to each discrete value of every degree of freedom. Then in each iteration a set of ants (potential solutions of the problem) choose a value for each degree of freedom according to the probability distribution. At the end of the iteration most of the probabilities are decreased (pheromone evaporation), but the ones corresponding to the best solution (shortest route) of the current iteration are increased, making it more likely that these values will be chosen by the ants in the next iteration. In PLANTS each solution is subjected to a local search algorithm at the end of each ACO iteration; then in a refinement step the LS is repeated for the best solution, which potentially further increases its fitness. PLANTS models flexibility with rotatable bonds and uses two different empirical scoring functions. One of them includes terms for protein-ligand steric interactions, torsions and clashes of the ligand (representing ligand internal energy), in addition, steric interactions and side-chain clashes of the protein (representing protein internal energy). The other scoring function is similar but models hydrogen bonds as well. The protein is represented with 3D grids during docking making the protein-ligand energy calculation more effective. From the point of view of parallelization ACO is similar to genetic algorithms: the ants can be generated, evaluated and subjected to local search in parallel like the entities of the GA.

The GPU accelerated PLANTS is described in reference [35]. The authors followed the traditional way of GPU programming using OpenGL and the NVIDIA Cg shading language. This method is less flexible than using CUDA; input data has to be encoded as textures, and functionality is implemented as shader programs processing these textures. The receptor grids, for example, are stored in a four channel (red, green, blue and alpha) 3D texture; the channels correspond to the four atom types which the scoring function of

PLANTS distinguishes. The optimization algorithms run on the CPU. The degrees of freedom are generated for each ant, then they are mapped to textures and moved to the GPU memory. Different shader programs calculate the coordinates of atoms, the protein-ligand interaction energy (by exploiting the interpolation capabilities of the GPU), the ligand clash and torsional energy terms. Finally a shader sums the partial energy terms. These steps are executed for each ant of the current ACO or LS iteration in parallel.

In order to exploit the capabilities of the GPU effectively the optimization algorithm was modified. The default value of ant colony size is 20 in PLANTS; to increase the number of solutions than can be evaluated in parallel multiple colonies are used, which sometimes exchange information by modifying the pheromone values of every colony according to the currently best solution. The refinement step was removed since it involves only one solution; in addition, the termination criterion of the LS was modified to prevent the parallel LS iterations from stopping after different number of steps. Although these modifications were necessary to achieve a high GPU utilization ratio, the altered algorithm turned out to be less effective than the original one; it requires a higher number of evaluations for finding the same solutions.

Test runs were performed on a 3.0 GHz dual core Pentium 4 CPU and an NVIDIA GeForce 8800 GTX GPU. For protein-ligand complexes the speedup of GPU accelerated steps was ×2-6 in case of 100 parallel solutions (5 colonies) and ×7-16 in case of 4000 parallel solutions (200 colonies), depending on the ligand structure. For protein-protein complexes with higher arithmetic intensity the speedup was ×10-20 and ×40-50 for 100 and 4000 parallel ants, respectively. The speedup of the whole GPU-based application with typically 400-500 parallel solutions proved to be about ×4 over the original PLANTS. This is an average value for a large set of protein-ligand complexes; in case of large and highly flexible ligands speedups over ×7 were observed.

## 5.6. Other approaches

As we mentioned, it is not possible to introduce every GPU-based docking solution reported; instead we try to give a general overview of the diverse methods applied in this field. In this subsection some further GPU-based implementations are mentioned, which in a way are different from the solutions described above. Instead of introducing these in details, we focus on the differences.

### *5.6.1. Hex on GPU*

Reference [36] describes the GPU-based acceleration of the Hex [37] program. Hex uses the FFT-correlation technique for docking. Instead of the ordinary Cartesian grids and translational correlation, however, Hex applies the spherical polar Fourier method based on rotational correlations, which allows to traverse not just the translational but also the orientational search space with FFT. The docking can be executed both with multiple 3D and with multiple 1D FFTs. Using 1D FFTs turned out to be much more advantageous on

the GPU, since it has a better memory read pattern than 3D FFT. The measured speedup on an NVIDIA GeForce GTX 285 was about ×45 compared to running Hex with 1D FFTs on a single CPU core.

### *5.6.2. Calculating pairwise potentials*

Reference [38] focuses on the acceleration of calculating the pairwise potentials between the protein-ligand atoms. In many docking applications this is performed with pre-calculated grids which reduces the $O(N_{prot}*N_{lig})$ complexity to $O(N_{lig})$ during docking (where N denotes the number of atoms of the molecule). This implementation, however, calculates the double sum directly; one protein atom is assigned to each CUDA thread, which iterates over the ligand atoms and calculates the corresponding potential values. Although the effectiveness of the approach is uncertain due to the increased complexity, it is interesting since it fits the GPU architecture perfectly. The number of protein atoms is usually high enough to keep the multiprocessors of the GPU busy; it is not necessary to evaluate multiple ligand positions simultaneously. In addition, the amount of input data is smaller (the number of protein atoms is usually lower than that of the grid points), making this approach less memory-intensive. Depending on the molecule sizes, speedups between ×10-260 were observed on an NVIDIA Tesla C1060 GPU, compared to the same algorithm running on an Intel Xeon E5530 CPU.

### *5.6.3. Using multiple GPUs*

Similarly to the previous section, reference [39] deals with accelerating only the pairwise potential calculation on GPU. The scoring function consists of two usual terms representing the van der Waals and electrostatic interaction. However, in this implementation two separate GPU devices are used; one of them calculates the van der Waals, the other one the electrostatic term. In a real docking application this approach would be probably impractical due to the required CPU-GPU memory transfer operations. Still, the applicability of multiple GPUs to the docking problem is intriguing; the most trivial way of utilizing them is to perform independent runs on the different devices. In case of this implementation overall speedup factors between ×118-193 were achieved; the test platform consisted of a 2.4 GHz Intel Core 2 Quad CPU and an NVIDIA GeForce 8800 GTX GPU.

## 6. Conclusion

Three FPGA-based and several GPU-based molecular docking implementations were surveyed in the previous sections. Although molecular docking algorithms are quite diverse in general, the methods introduced in this chapter actually fall into two categories. Both categories represent a docking approach which is easily parallelizable and thus suits well the architecture of accelerator platforms.

The first group includes the correlation-based methods (Section 4.1, 4.2, 5.1, 5.2 and 5.6.1). As it was shown, correlation is a massively parallel operation and can be implemented

effectively in FPGA; on GPU in turn it can be performed with optimized FFT kernels. This makes correlation-based docking algorithms ideal for hardware acceleration; the limitation is that they support only rigid-body docking.

The second group includes docking algorithms based on a global optimization algorithm which is inherently parallel (Section 4.3, 5.3-5.5). Both the evolutionary algorithms used by AutoDock and MolDock, and the ant colony optimization method of PLANTS operate on sets of potential solutions, which allows members of the set to be processed in parallel. The usual pairwise scoring functions applied by these programs offer further parallelization at the level of atoms or atom pairs. In addition, these methods support modeling of molecular flexibility, too.

Many of the introduced, accelerator-based docking implementations achieved significant speedup over single or even multi-core CPUs. The actual speedup value is always a matter of reference platform, of course; still, the results prove that molecular docking can effectively accelerated by hardware and often a performance improvement of 1-2 orders of magnitude can be obtained. However, this improvement is usually not constant; in many cases it was shown that it strongly depends on input parameters (number of atoms, size of search space, search exhaustiveness, etc.), making accelerators usually more suitable for larger problem sizes.

It should also be noted that performance improvement may come at a price: in some cases (4.3, 5.3.2, 5.5) the original algorithm had to be altered to make it more suitable for parallelization. Typically these changes were related to the local search in these cases, which is essentially a sequential algorithm. Such modifications are often necessary, however, they change the behavior and accuracy of the algorithm, which is sometimes unacceptable. Another typical necessity is that in addition to the computationally intensive but parallelizable steps that suit well the accelerator architecture, other parts must also be mapped to the accelerator in order to avoid that the host-accelerator bandwidth becomes a bottleneck. This, however may greatly increase the required programming effort.

Another interesting point is the applicability and performance of FPGAs vs. GPUs. In case of the PIPER implementations (Section 4.2, 5.1) the FPGA outperformed the GPU when both executed correlation directly; but due to the effective FFT-based approach the GPU implementation seemed to be more suitable since its performance scaled well with the problem size. In case of AutoDock (Section 4.3, 5.3.3) the GPU outperformed the FPGA in practical cases, although the latter exploited the low-level parallelism of the docking algorithm more effectively and thus was faster than the GPU if the number of parallel runs was low. All these results confirm that GPU devices represent a real competitor of FPGAs even when considering only performance. In addition, as it was mentioned in Section 3, FPGA programming usually requires hardware skills while GPUs can be programmed in C-like languages (although there are high-level C-based HDLs they are usually not as effective as VHDL or Verilog). GPU cards are cheaper by far than high-performance FPGA accelerators, and often they are already available in the desktop PC. All these facts suggest that GPUs are a better choice as accelerator platform than FPGAs in case of floating point-

intensive applications like the majority of the docking algorithms, although clearly there are problem domains where FPGAs remain superior.

## Author details

Imre Pechan
*evopro Informatics and Automation Ltd, Budapest, Hungary*

Béla Fehér
*Department of Measurement and Information Systems,*
*Budapest University of Technology and Economics, Budapest, Hungary*

## Acknowledgement

We would like to thank evopro Informatics and Automation Ltd for supporting our work and providing access to the necessary hardware and software tools.

## 7. References

[1] Brooks BR, Bruccoleri RE, Olafson BD, States DJ, Swaminathan S, Karplus M (1983) CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. J. Comput. Chem. 4: 187-217.

[2] Cornell WD, Cieplak P, Bayly CI, Gould IR, Merz KM, Ferguson DM, Spellmeyer DC, Fox T, Caldwell JW, Kollman PA (1995) A Second Generation Force Field for the Simulation of Proteins, Nucleic Acids, and Organic Molecules. J. Am. Chem. Soc. 117: 5179-5197.

[3] Grosdidier A, Zoete V, Michielin O (2011) Fast docking using the CHARMM force field with EADock DSS. J. Comput. Chem. 32: 2149-2159.

[4] Ewing TJA, Makino S, Skillman AG, Kuntz ID (2001) DOCK 4.0: Search Strategies for Automated Molecular Docking of Flexible Molecule Databases. J. Comput. Aided Mol. Des. 15: 411-428.

[5] Friesner RA, Banks JL, Murphy RB, Halgren TA, Klicic JJ, Mainz DT, Repasky MP, Knoll EH, Shelley M, Perry JK, Shaw DE, Francis P, Shenkin PS (2004) Glide: A New Approach for Rapid, Accurate Docking and Scoring. 1. Method and Assessment of Docking Accuracy. J. Med. Chem. 47: 1739-1749.

[6] Huey R, Morris GM, Olson AJ, Goodsell DS (2007) A Semiempirical Free Energy Force Field With Charge-Based Desolvation. J. Comput. Chem. 28: 1145-1152.

[7] Muegge I, Martin YC (1999) A General and Fast Scoring Function for Protein-Ligand Interactions: A Simplified Potential Approach. J. Med. Chem. 42: 791-804.

[8] Gohlke H, Hendlich M, Klebe G (2000) Knowledge-Based Scoring Function to Predict Protein-Ligand Interactions. J. Mol. Biol. 295: 337-356.

[9] Rarey M, Kramer B, Lengauer T (1997) Multiple Automatic Base Selection: Protein–Ligand Docking Based on Incremental Construction Without Manual Intervention. J. Comput. Aided Mol. Des. 11: 369-384.

[10] Jones G, Willett P, Glen RC, Leach AR, Taylor R (1997) Development and Validation of a Genetic Algorithm for Flexible Docking. J. Mol. Biol. 267: 727-748.
[11] Trott O, Olson AJ (2010) AutoDock Vina: Improving the Speed and Accuracy of Docking with a New Scoring Function, Efficient Optimization, and Multithreading. J. Comput. Chem. 31: 455-461.
[12] Teodoro ML, Phillips GN, Kavraki LE (2001) Molecular Docking: A Problem With Thousands of Degrees of Freedom. IEEE Int. Conf. on Robotics and Automation, 2001 May 21-26, Seoul, Korea.
[13] Dias R, de Azevedo WF (2008) Molecular Docking Algorithms. Curr. Drug Targets 9: 1040-1047.
[14] Kavraki LE (2007) Protein-Ligand Docking, Including Flexible Receptor-Flexible Ligand Docking. Receptor 1-19. Available: http://cnx.org/content/m11456/latest/. Accessed 2012. 04. 29.
[15] Hauck S, DeHon A (2007) Reconfigurable Computing - The Theory and Practice of FPGA-Based Computation. Morgan Kaufmann. 944 p.
[16] Kilts S (2007) Advanced FPGA Design - Architecture, Implementation, and Optimization. Wiley. 352 p.
[17] NVIDIA CUDA C Programming Guide. Available: http://developer.nvidia.com/nvidia-gpu-computing-documentation. Accessed 2012. 04. 29.
[18] Sanders J, Kandrot E (2010) Cuda by Example - An Introduction to General-Purpose GPU Programming. Addison-Wesley. 312 p.
[19] AMD Accelerated Parallel Processing OpenCL Programming Guide. Available: http://developer.amd.com/sdks/AMDAPPSDK/documentation/Pages/default.aspx. Accessed 2012. 04. 29.
[20] Kozakov D, Brenke R, Comeau SR, Vajda S (2006) PIPER: An FFT-Based Protein Docking Program with Pairwise Potentials. Proteins 65: 392-406.
[21] VanCourt T, Gu Y, Herbordt MC (2004) FPGA Acceleration of Rigid Molecule Interactions. 12th Ann. IEEE Symp. on Field-Programmable Custom Computing Machines, 2004 Apr. 20-23, Napa, USA.
[22] VanCourt T, Gu Y, Mundada V, Herbordt MC (2006) Rigid Molecule Docking: FPGA Reconfiguration for Alternative Force Laws. EURASIP J. on Applied Signal Processing 2006: 1-10.
[23] Sukhwani B, Herbordt MC (2010) FPGA Acceleration of Rigid-Molecule Docking Codes. IET Comput. Digit. Tech. 4: 184-195.
[24] Pechan I, Fehér B, Bérces A (2010) FPGA-Based Acceleration of the AutoDock Molecular Docking Software. Conf. on Ph.D. Research in Microelectronics and Electronics, 2010 July 18-20, Berlin, Germany.
[25] Pechan I, Fehér B (2011) Molecular Docking on FPGA and GPU Platforms. Int. Conf. on Field Programmable Logic and Applications, 2011 Sept. 5-7, Chania, Greece.
[26] Sukhwani B, Herbordt MC (2009) GPU Acceleration of a Production Molecular Docking Code. 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009 Mar. 8, Washington, USA.

[27] Feng Z, Tian X, Chang S (2010) A Parallel Molecular Docking Approach Based on Graphic Processing Unit. 4th Int. Conf. on Bioinformatics and Biomedical Engineering, 2010 June 18-20, Chengdu, China.
[28] Sousa SF, Fernandes PA, Ramos MJ (2006) Protein-Ligand Docking: Current Status and Future Challanges. Proteins 65: 15-26.
[29] Micevski D, Kuiper M (2009) Optimizing Autodock with CUDA. VPAC Case Study. Available: http://www.vpac.org/?q=node/290. Accessed 2012. 04. 29.
[30] Kannan S, Ganji R (2010) Porting Autodock to CUDA. IEEE Cong. on Evolutionary Computation, 2010 July 18-23, Barcelona, Spain.
[31] Morris GM, Goodsell DS, Halliday RS, Huey R, Hart WE, Belew RK, Olson AJ (1998) Automated Docking Using a Lamarckian Genetic Algorithm and an Empirical Binding Free Energy Function. J. Comput. Chem. 19: 1639-1662.
[32] Simonsen M, Pedersen CNS, Christensen MH, Thomsen R (2011) GPU-Accelerated High-Accuracy Molecular Docking Using Guided Differential Evolution. 13th Ann. Conf. on Genetic and Evolutionary Computation, 2011 July 12-16, Dublin, Ireland.
[33] Thomsen R, Christensen MH (2006) MolDock: A New Technique for High-Accuracy Molecular Docking. J. Med. Chem. 49: 3315-3321.
[34] Korb O, Stützle T, Exner TE (2007) An Ant Colony Optimization Approach to Flexible Protein–Ligand Docking. Swarm Intell. 1: 115-134.
[35] Korb O, Stützle T, Exner TE (2011) Accelerating Molecular Docking Calculations Using Graphics Processing Units. J. Chem. Inf. Model. 51: 865-876.
[36] Ritchie DW, Venkatraman V (2010) Ultra-Fast FFT Protein Docking on Graphics Processors. Bioinformatics 26: 2398-2405.
[37] Ritchie DW, Kozakov D, Vajda S (2008) Accelerating and Focusing Protein-Protein Docking Correlations Using Multi-Dimensional Rotational FFT Generating Functions. Bioinformatics 24: 1865-1873.
[38] Guerrero GD, Sánchez HP, Wenzel W, Cecilia JM, García JM (2011) Effective Parallelization of Non-bonded Interactions Kernel for Virtual Screening on GPUs. 5th Int. Conf. on Practical Applications of Computational Biology & Bioinformatics, 2011 Apr. 6-8, Salamanca, Spain.
[39] Roh Y, Lee J, Park S, Kim J (2009) A Molecular Docking System Using CUDA. Int. Conf. on Hybrid Information Technology, 2009 Aug. 27-29, Daejeon, Korea.