Porting Autodock to CUDA

CONFERENCE PAPER · AUGUST 2010

DOI: 10.1109/CEC.2010.5586277 · Source: IEEE Xplore

Raghavendra Ganji
HCL Technologies Limited
1 PUBLICATION **9** CITATIONS

WCCI 2010 IEEE World Congress on Computational Intelligence
July, 18-23, 2010 - CCIB, Barcelona, Spain CEC IEEE

Porting Autodock to CUDA

Sarnath Kannan, Raghavendra Ganji

Abstract—**This paper is a report on the migration of the molecular docking application, "Autodock" to NVIDIA CUDA. Autodock is a Drug Discovery Tool that uses a Genetic Algorithm to find the optimal docking position of a ligand to a protein. Speedup of Autodock greatly benefits the drug discovery process. In this paper, we show how significant speed up of Autodock can be achieved using NVIDIA CUDA. This paper describes the strategy of porting the Genetic Algorithm to CUDA. Three different parallel design alternatives are discussed. The resultant implementation features ~50x speedup on the fitness function evaluation and 10x to 47x speedup on the core genetic algorithm.**

I. Introduction

AUTODOCK is a molecular docking application that is widely used in drug discovery. It is available as open source software under GPL. Autodock is developed and maintained by TSRI [1].

In this paper, we present a CUDA [2] enabled version of Autodock (*version 4.2.1*) that speeds up the core Genetic Algorithm by two orders of magnitude. We chose to parallelize Autodock at an architectural level than at a functional level. We explored 3 different approaches for efficient fitness calculation on GPU before choosing the best. Our implementation gives a speedup of ~50x in fitness evaluation for the typical usage. We also present a CGPU (*CPU+GPU*) memory management approach suitable for handling memory for large projects.

Our speedup results are a huge leap compared to the open source project *gpuAutodock* [3]. *gpuAutodock* achieves 4% (*1.04x*) speedup for typical population sizes and a similar speedup for higher population sizes.

Autodock Vina [4] is a new docking tool from TSRI that achieves two orders of magnitude speedup even on existing hardware. Our speedups are similar to Vina. However this is *not* an apple to apple comparison. Vina uses efficient *local search* that reduces the evaluations needed to arrive at the solution. As on today, Vina is distributed only in binary.

Local search algorithms are usually iterative and adaptive and easily take up considerable amount of time in docking. Much of this time is spent in fitness evaluation. By providing efficient fitness evaluation on CUDA, we have laid the ground-work for implementing efficient local search on GPUs.

This paper does not focus on Local Search. The hunt for a suitable local search method for GPU (*Graphics Processing Unit*) itself is a separate research work and deserves a separate paper. This paper only focusses on implementation of the Genetic Algorithm and the fitness function on CUDA. Towards the end of paper, we briefly discuss some strategies for efficient local search on the GPU.

II. CUDA Architecture

CUDA stands for ***Compute Unified Device Architecture***. It is a GPGPU [5] technology pioneered by NVIDIA Corporation. CUDA programs seamlessly run on a family of GPUs from NVIDIA.

A. *Physical View*

At the physical level, the NVIDIA GPU can be seen as a set of multi-processors (*MP*). Each MP has 8 SIMD cores. MPs execute hardware threads as dictated by the CUDA software. A bunch of 32 threads called a "warp" are always executed together by the MP.

All MPs share a common global memory (GPU's main graphics memory). Each MP has 16KB of high-speed shared memory placed under program control. These are unlike CPU caches which are program un-aware. All MPs have an 8KB texture cache to accelerate texture fetches. The global memory offers a very high bandwidth. This bandwidth is manifold of CPU's main memory bandwidth.

Some of the commonly used math library functions are directly implemented on the hardware. For example *sqrtf*, *sinf*, *cosf*, *reciprocal sqrtf* etc.

The hardware is capable of hiding latencies by overlapping computation from different threads with memory access. Scheduling a lot of threads also help to avoid "Read-After-Write" dependencies in the execution pipeline resulting in superior performance. The concept is very similar to Hyper-threading but is done massively and efficiently by the CUDA hardware [6].

B. *Logical View*

CUDA programming model allows programmers to launch software kernels to run on the GPU. The programmer can specify the number of *thread blocks* that need to be spawned by the hardware while launching the CUDA kernel. The GPU hardware takes care of spawning the *thread blocks* and executing the kernel. A *thread block* is a bunch of hardware threads tied to a shared memory region. The physical number of MPs in the underlying hardware has no

Manuscript received January 31, 2010. This work was fully supported by the Center of Excellence for High Performance Computing, HCL Technologies.

Sarnath Kannan is with HCL Technologies, Bangalore, India (phone: +91-80-41584000; e-mail: k_sarnath@hcl.in)
Raghavendra Ganji is with HCL Technologies, Bangalore, India (e-mail: raghavendrag@hcl.in)

bearing on the number of *thread blocks* that can be launched. CUDA provides for synchronization and memory consistency primitives for all threads inside a *thread block*. However, the *thread blocks* themselves are completely independent of each other and cannot be synchronized. Recent versions of CUDA hardware support global atomics and global memory consistency primitives enabling *thread blocks* to communicate and synchronize with each other.

Frequently used data can be stored in shared memory to accelerate computations. The CUDA programming model requires adjacent threads to access adjacent memory locations for the best memory bandwidth. This concept is called *Memory Coalescing* and it reduces the number of trips to memory. Applications requiring random access can benefit from textures. CUDA offers 1D, 2D and 3D read-only textures optimized for spatial locality. Control statements like "if", "for" and "while" can split warps causing loss of hardware cycles.

III. AUTODOCK BACKGROUND

Autodock uses a Lamarckian Genetic Algorithm to search for the optimal docking position of a ligand inside a protein. The ligand is also referred as *small molecule*. The protein is also referred as *Macro Molecule* or the *Receptor*. This paper will focus only on the core Genetic Algorithm (*GA*). The Lamarckian variant pertains to local search which is "not" a focus of this paper.

A. Solution Space

Autodock attempts to find the optimal docked position of the *ligand* inside a 3D grid box specified in the active site of the *protein*. The 3D grid box represents the solution space. This box is a continuous space and hence the total number of possible solutions is infinite. Each 3D point inside this grid box is a potential candidate where the *ligand* can be centered. Autodock uses discrete 3D energy points for fitness evaluation purposes and uses tri-linear interpolation to arrive at approximate energies.

B. Genetic Code (chromosome)

Genetic Algorithms usually identify the notion of an individual and its genetic code (*Chromosome*) with respect to the problem's solution space. The genetic code must uniquely identify a candidate solution.

Autodock treats a particular orientation of the ligand inside the protein as an "*individual*" in its GA. Hence Autodock's genetic code uniquely identifies the orientation of a ligand.

Autodock represents the chromosome as a vector of real numbers. Each element in the vector stands for a gene. A pictorial representation of the chromosome can be found in Figure 1. Following parameters are used as genes contained in an individual's chromosome.

- X,Y,Z-Translations of the center of ligand inside the 3D grid box (*3 genes*)
- Quaternion Information (*4 genes – w, x, y, z*) that specifies the overall 3D orientation of the ligand
- Torsion angle information for each rotatable bond in the ligand (*"Rotatable bonds" number of genes*)

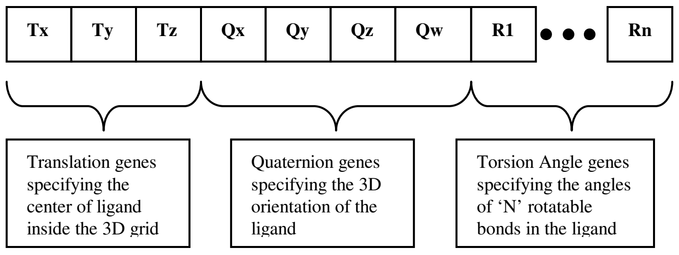
Tx
Ty
Tz
Qx
Qy
Qz
Qw
R1
Rn
Translation genes specifying the center of ligand inside the 3D grid
Quaternion genes specifying the 3D orientation of the ligand
Torsion Angle genes specifying the angles of 'N' rotatable bonds in the ligand

Figure 1 - Autodock GA Chromosome Representation

C. Genetic Operators

Autodock performs all the genetic operators namely, *Selection*, *Crossover* and *Mutation* on the genetic code.

D. Fitness Evaluation

Fitness evaluation evaluates the total energy of a particular orientation of the ligand inside the protein. Lower is better.

E. Typical GA parameters

Here are some typical parameters used while running Autodock [7].

- Population Size – 150
- Crossover Rate – 80% of individuals undergo Crossover on every generation
- Mutation Rate – 2% of the total genes in a population undergo mutation on every generation

IV. AUTODOCK GA ON CUDA

This section is broadly structured as follows

- High Level Guidelines adopted for CUDA Migration
- Fitness function on CUDA
- Genetic Operators on CUDA
- Results

The results shared in this paper used the following configurations

- AMD Athlon, 2.41GHz, 64 KB L1 I-Cache, 64 KB L1 D-Cache, 512KB L2 Cache, 2GB RAM
- Tesla C1060, 240 cores @ 1.3GHz, 4GB Graphics RAM
- The ind-hsg ligand protein pair that comes with Autodock tutorial was used for docking. The "ind" molecule has 12 rotatable bonds (torsions) and "hsg" has a flexible side chain as well. Together they represent a typical docking pair.
- Proportional Selection
- Uniform 2-point Crossover with a rate of 80% of individuals per generation

- Mutation rate of 2% of total genes per generation

A. *High Level Guidelines adopted for CUDA Migration*

1) *Floating Point Precision*

One of the earliest design choices that we made was to use single-precision arithmetic. This is harmless because GA depends on relative goodness among individuals' energies and single precision may not affect the accuracy of GA path significantly.

2) *GA State Maintenance*

We decided to keep as much GA State in the GPU itself and perform the GA operators using GPU kernels. This decision is *critical* because this eliminates un-necessary memory copies back and forth GPU memory and host PC's main memory. GA State includes the chromosomes of all individuals in the current population, Energy values of individuals, chromosomes of all individuals in new population and so on.

3) *Phenotypes*

The Genetic Algorithm used in Autodock does not differentiate between a genotype and a phenotype. Autodock merely duplicates the genotype to create the phenotype. Logically, the atomic coordinates of each participating atom is the phenotype resulting from a genotype. Although Autodock does generate the logical phenotype eventually, it does "not" represent the same as a "phenotype" in the software. Instead the software merely duplicates the genotype to generate the phenotype. In our implementation, we did away with this redundancy and maintained only one single copy. This helped in minimizing un-necessary memory copies.

4) *Random Numbers*

The random number generator used in Autodock maintains state information and is difficult to port to GPU. For testing purposes, we required the CPU and GPU implementations to generate the same stream of random numbers. Hence we retained the random number generation in the CPU itself. This has two advantages:

- Enables one to one comparisons of CPU and GPU results. This is very important in validating the accuracy of GPU implementation.
- Reduce the design, coding and validation effort of generating random numbers on GPU.

5) *CGPU Memory Manager*

The Autodock program involves a lot of memory buffers. Majority of them are one time copies needed only during initialization, for example, the energy map files. Some of others are dynamic and need to be transferred to and from GPU and host PC's main memory. We came up with the idea of a CGPU memory manager, implemented as a C++ object which features the following:

- Avoid GPU memory fragmentation by pooling related memory requests
- Support Alignment for individual memory requests
- Support for pinned memory
- Copy related buffers between CPU and GPU in one shot, typically using operator overloading
- Free related buffers in 1 shot

The CGPU memory manager object could be constructed in one of the following ways

- Specifying the location as CPU or GPU
- Pairing with another CGPU memory manager object. The pair could be homogeneous or heterogeneous with respect to the location, the default being heterogeneous. A use case for homogeneous pair can be found in the Selection operator section below.
- Specifying a vector of allocation requests (the location defaults to GPU unless otherwise specified)

All of the construction interfaces offer an optional flag to signal a 'pinned' memory allocation.

6) *Root Structures*

Root Structures hold information and pointers to data used in the GA. Our implementation used two Root structures, namely *CPURoot* and *GPURoot*. The GA initialization module creates memory objects, allocates memory, copies data, retrieves the memory pointers and populates them in the CPURoot and GPURoot structures. Both the structures were made available globally to other modules of the GA. This helped us in the following.

- To clearly demarcate CPU and GPU pointers
- To create a hierarchy of pointers and data that gave structure to the code

B. *Fitness Function (Energy Evaluation) on CUDA*

Energy evaluation is the fitness function used by Autodock's GA. For every generation, the energies of all the individuals are re-calculated. In practice, only the modified ones get re-calculated. With 80% *crossover* rate, majority of the population gets modified every generation, which is a sizeable number. Our strategy was to evaluate all the individuals in a population regardless of modifications. This strategy is also helped by the fact that the entire GA state resides inside the GPU.

Energy of a particular individual is the sum of *inter-molecular* energy (*between atoms of ligand and receptor*) and *intra-molecular* energy (*amongst movable atoms in the ligand and flexible side-chain in the receptor*).

- Fitness evaluation starts by locating the atom coordinates of each atom in the ligand based on its genetic code. (torsion() , qtransform() functions)
- Once the coordinates of the atoms in the 3D box are known, the inter-molecular energies of each atom in the ligand are summed up. The inter-molecular energy values are pre-generated by Autogrid [8] for every atom type in the ligand and available as energy map files. Since the atomic coordinates are in continuous space, Autodock performs tri-linear

interpolation based on values in the energy map files and the atom type. (trilinterp() function)

- Autodock runs through the non-bond list evaluating intra-molecular energies. (eintcal() function).

1) Design Alternatives

We explored 2 different approaches to evaluate the fitness function on CUDA. One was to dedicate one GPU thread to calculate the energy of one individual (*PerThread)*. The other approach was to use one GPU *Thread Block* for calculating the energy of one individual (*PerBlock). The final design was a variant of PerBlock approach and is explained in the next section*. Here is a brief comparison of the two parallel approaches.

Table 1 - Comparison of 2 approaches to Parallel breakdown of the Fitness function

	PerThread Parallel Breakdown Approach	PerBlock Parallel Breakdown Approach
1	Parallelism exploited at inter-individual level. Parallelism is proportional to the population size.	Parallelism exploited at intra-individual level. This is usually a function of the number of atoms participating in the docking (*atoms in ligand + atoms in flexible side chain*)
2	Requires population size in thousands to saturate the GPU.	A modest population in hundreds is enough to saturate the GPU
3	Only common data to all individuals can be placed in shared memory.	All data pertaining to a particular individual can be placed in shared memory. Common data can be accessed via textures.
4	Applying Torsions is straight forward. Each thread applies the necessary torsions for the individual it controls.	Since an atom's coordinates could be dictated by more than one torsion entry, parallelism is restricted to the atoms affected by a single torsion. (*which could be very less*)
5	Summing the energies is straight forward. Each thread sums up the energy values for the individual it controls.	Summing the energies require a *reduction* [9] towards the end.

We designed the following set of 5 kernels for both the approaches

1. Atomic coordinate initialization
2. Applying *Torsion*
3. Applying *qtransform*
4. *Tri-linear Interpolation*
5. *Internal energy calculation*

We used structure of arrays so that *PerThread* kernels accessed memory in a coalesced way.

Figure 2 and Figure 3 show the breakdown of typical time taken for the 5 steps for the *PerThread* and *PerBlock* cases on the GPU. A population size of 5760 was used in *PerThread* case. A population size of 300 was used in *PerBlock* case. These numbers were chosen to saturate the GPU completely. Tri-linear interpolation and internal energy calculation dominate the computation in both cases. Both the steps are memory intensive and that explains why they take the most time. The *PerBlock* implementation spends relatively more time in *torsion*. This can be attributed to the 4th point in Table 1.

Figure 4 shows the performance results of the two approaches. *PerBlock* approach performs ~12x faster than the *PerThread* for the typical population size of 150.

It is worth noting that CUDA hardware provides hardware support for 3D Textures and tri-linear interpolation. The tri-linear interpolation function can be replaced by a single "*tex3D*" instruction on the GPU. However the GPU hardware performs the interpolation with reduced precision. Our implementation can be compiled with/without hardware interpolation. In any case, 3D textures accelerated *tri-linear interpolation* by a great amount due to the *3D spatial locality* involved in interpolation. This feature was exploited in both *PerThread* and *PerBlock* kernels. We *never* enabled the hardware interpolation in any of the tests discussed in this paper because it caused the results to differ significantly.

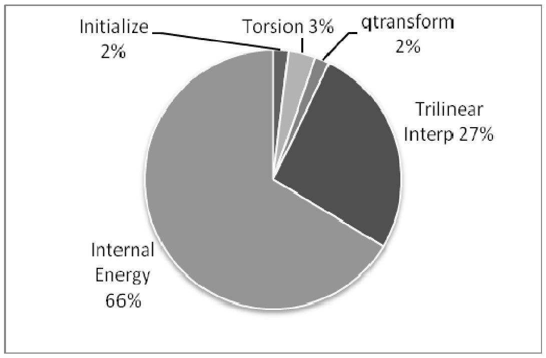

Figure 2 – Fitness function on GPU: Time breakdown for PerThread Approach

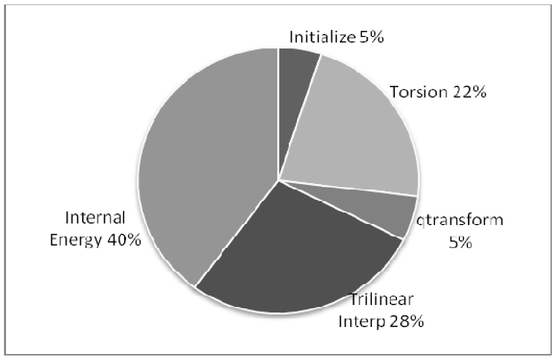

Figure 3 – Fitness function on GPU: Time breakdown for PerBlock Approach

2) Final Design

The final design is an improvement over the "*PerBlock*" approach. All the 5 *PerBlock* kernels *write* and/or *read* all the atomic coordinates pertaining to an individual. This overhead can be completely eliminated if one chooses to launch only *one* kernel and keep all the atomic coordinates cached in shared memory. We refer this implementation as "*PerBlockCached*". This approach outperforms the *PerBlock* implementation consistently.

However this approach limits the number of atoms that can be supported for docking. CUDA hardware provides 16KB of shared memory per MP. Out of this roughly 14KB is used for storing 3D atomic coordinates. This can help store about a 1000 atoms. Autodock supports a maximum of 2000 atoms. However for practical purposes 1000 is enough. With the advent of Fermi [10], CUDA applications can use 48KB of shared memory that can easily accommodate 2000 atoms.

3) Results of Energy Evaluation (Fitness)

Figure 4 shows the comparative results of the three different approaches. The *PerBlockCached* kernel outperforms the other two for both smaller and relatively larger population sizes. The *PerBlockCached* kernel runs ~19x faster than *PerThread* and ~1.55x faster than the *PerBlock* kernel for the typical population size of 150. The *PerThread* kernel gives the best performance for very high population sizes. The *PerBlockCached* kernel outperforms the *PerBlock* kernel by a factor of ~1.2x to ~1.5x for the considered range of population sizes.

Applying torsions takes significant amount of time in *PerBlock and PerBlockCached* versions. This is one reason why *PerThread* version performs much better for higher population sizes. Our final implementation dynamically selected the kernels according to the population size specified by the user. There is still scope for improving *torsion* operation in *PerBlock* variants.

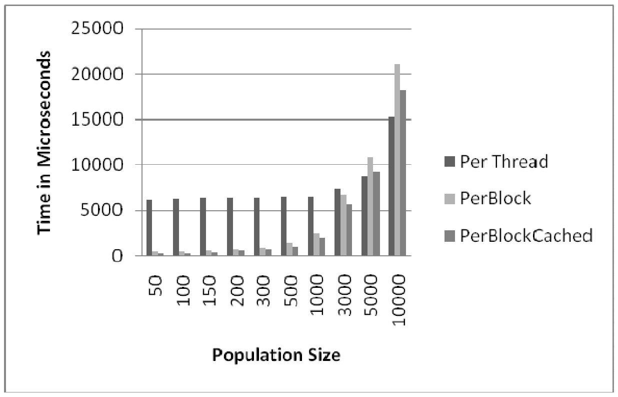

Figure 4 – Fitness function on GPU: Comparison of 3 parallel breakdown approaches

Figure 6 shows speedups achieved for various GA operators with respect to the CPU. The fitness operator in Figure 6 corresponds to the *PerBlockCached* variant. It yields a speedup of **~50x** over CPU for the typical population size of 150. The "*dip*" in the speedup at the population size of 200 can be explained by the fact that CUDA hardware *idles* for certain population sizes (*which is equal to the number of thread blocks*). One can also see that the fitness speed up saturates at *60x* for huge population sizes.

C. Selection

Selection process involves calculation of relative merit of each individual and selecting the eligible ones from the current population and copying them to the new population. Autodock supports multiple selection modes. Our GPU implementation supports only the "*selection proportional*" method as this is the default selection mode in Autodock.

The *selection* process was implemented as a GPU kernel that copied marked individuals from old population to the new population. The CPU merely indicated which individuals need to be copied. Our implementation used GPU to calculate the population average and relative merit of each individual and this was overlapped with other *selection* related work like permuting the ordering array on the CPU.

One can infer from Figure 6 that *selection* performs almost as good as CPU for lesser population sizes but slowly outperforms the CPU as the population size increases. The time complexity of *Selection* depends on the fitness of the population. That explains why the speedup graph for *Selection* is non-linear.

Selection requires two population instances to exist inside the GPU. Having one population instance causes read-write parallel hazards as GPU threads copy individuals (read) from old population and update (write) the new population. This forms a use case for support for *homogeneous pairs* in CGPU memory manager.

D. Crossover

Crossover process involves exchanging genetic code between two selected individuals at selected gene locations. *Crossover* was implemented as a GPU kernel. Our GPU implementation supports only the *2-point crossover mode* which is the default mode used by Autodock.

The GPU kernel exploited the pair-level parallelism. Autodock generates unique *crossover* pairs which is a necessary condition for running *Crossover* in parallel on GPU. One can infer from Figure 6 that Crossover operation on GPU linearly increases in speedup. This is expected because an increase in population size increases the number of Crossover operations and GPU gets loaded fully.

We would also like to share that we identified a bug in the *Crossover* implementation of Autodock during this process [11].

E. Mutation

Autodock implements the *Mutation* process by modifying random genes chosen from random individuals. The GPU implementation of the *Mutation* process exploits the per-mutation parallelism. However, Autodock allows a gene to be mutated more than once. These repeated mutations cause a parallel hazard in exploiting per-mutation parallelism.

This can be avoided in 2 ways. One approach would be to exploit per-gene parallelism instead of per-mutation parallelism. This approach will accommodate multiple mutations to the same gene. The other approach would be to avoid multiple-mutations. Our implementation chose the second one.

The GPU implementation features a Mutation Manager that avoids more than 1 mutation to a gene. The manager re-maps a colliding mutation to an un-mutated gene. For this reason, the GPU implementation results deviate from CPU implementation when mutations are enabled.

1) Mutation Manager

Mutation Manager simply re-maps colliding gene mutations. It uses a bit array to identify genes that are already mutated. An alternate approach would be to replace multiple mutations occurring on a same gene with a single mutation that causes the combined effect. This is quite possible if the mutation operator is *associative*. Non-associative mutation operations still need to be managed with re-mappings. We have *not* investigated the associative properties of the mutation operator in Autodock.

Figure 6 shows that mutation in GPU performs worse than the CPU. Experiments show that GPU mutation runs almost 6x slower than the CPU when 1% of total genes in a population undergo mutation. The number decreases to 4x when 2% undergo mutation. The increase in parallelism works to GPU's benefit in the 2% case. The slowness is mainly due to the overhead of Mutation Manager. Since mutation corresponds to a very minimal part of the GA, this does not affect the overall speedup. This can be inferred from Figure 7

F. Population swapping

When the new population is ready to go, our implementation merely swaps out the old and new *pointers* instead of copying data. This is in sharp contrast with Autodock that copies the new population to old population and destroys the newly created object. The current CPU implementation of Autodock can be easily sped up by employing an *object reference* swapping scheme instead of copying and destroying.

G. Results

1) Accuracy

CPU and GPU Results were comparable for smaller number of generations. However, deviations were observed when the number of generations increased to thousands. This was traced to energy evaluation results. Difference in order of floating point calculations (*the reductions*) and the single precision nature of computation (*Intel CPUs use 80-bits precision internally*) causes GPU evaluated energy values to slightly differ from CPU. This causes a change in the *Selection* process that affects the course of the GA. We believe that this will *not* be an issue and that several runs of Autodock with random seeds would help in identifying good ligand-protein conformations.

2) Speedups

Figure 5 shows the net speedup of the GA with respect to the population size. The core GA used by Autodock got accelerated by **10x** to **47x** depending on the chosen population size.

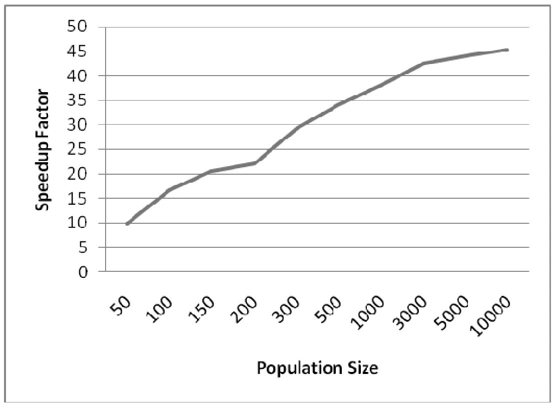

Figure 5 - Overall GA Speedup

Figure 6 shows the speedup of various GA phases with respect to the population size. To make one to one comparisons, we changed the CPU implementation *not* to evaluate energies in *selection*, *Crossover* and *mutation* phases. But the energies were evaluated for the entire population after the GA phases ended (*as opposed to only modified ones*).

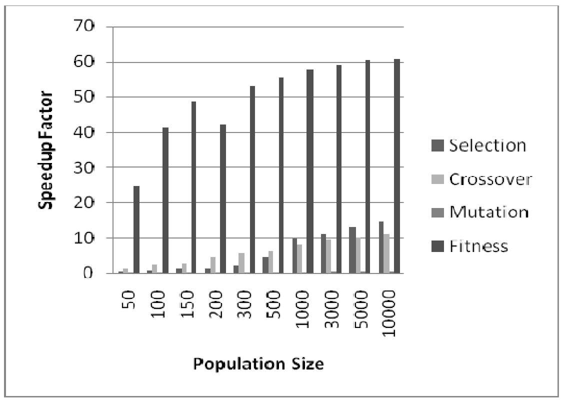

Figure 6 – Speedup of the Genetic Operators

It can be inferred from Figure 6 that *fitness evaluation* offers superior speedups followed by *Selection and Crossover*. For the typical population size of 150, *fitness evaluation* offers ~50x speedup, Selection offers ~1.25x speedup and Crossover offers ~2.75x speedup. *Selection* and *Crossover* outperforms CPU for higher population sizes. This is mainly due to the superior memory bandwidth offered by GPU compared to CPU.

Figure 7 and Figure 8 are pie charts showing the percentage of time spent by CPU and GPU implementations in various phases of the GA.

Two important points emerge from Figure 7.

- Energy evaluation dominates the GA completely in the CPU.
- The population swapping operation takes the combined time of selection, crossover and mutation. Our experiments reveal that this phenomenon is observed for all kinds of population sizes.

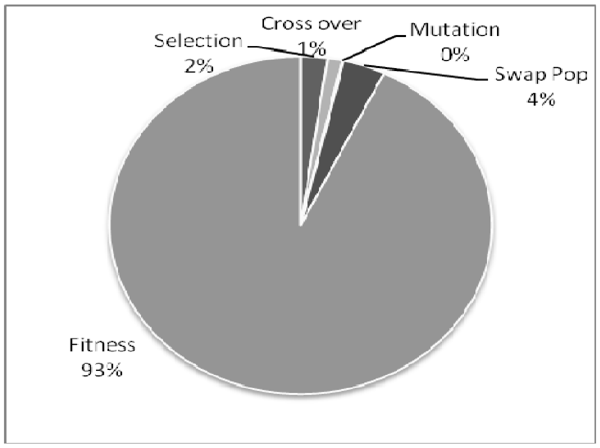

Figure 7 – GA on CPU: Time breakdown for pop size 150

Figure 8 tells that *selection* and *fitness* dominate the GA for a population size of 150 in GPU. Our experiments show that as the population size increases, *fitness*' domination increases close to 50%. Nonetheless, it is important to note that selection is also definitely a bottleneck for the GA in GPU.

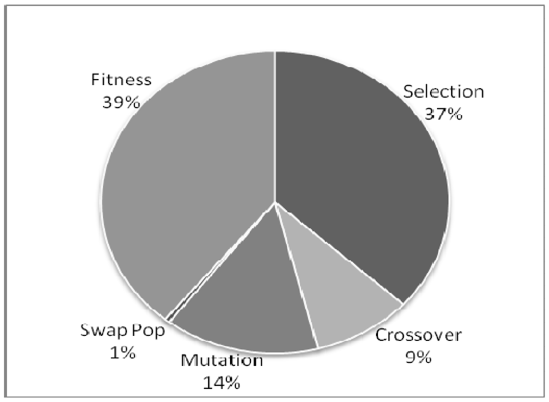

Figure 8 – GA on GPU: Time breakdown for pop size 150

V. TESTING

Testing was done with the *ind-hsg* ligand protein pair. We started testing by creating a CPU version of Autodock that would output necessary debug data that we wished to see and one that would run only the global search. We also made the CPU implementation evaluate energies in one shot during the GA process. The original implementation evaluated energies of individuals as and when they got modified during various phases of the GA. We changed this behavior to make the sequence of steps much like the GPU implementation. Both CPU and GPU random number generators were seeded with same numbers to make sure that results match.

Energy evaluation was tested first. This was done by copying intermediate values, like atomic coordinates, intermediate energy values to separate files. Similar strategies were followed for other phases of GA. The biggest challenge was to maintain the order of random number generation between the CPU and GPU. Even a small misstep would cause complete deviations in the final result.

VI. CONSIDERATIONS FOR LOCAL SEARCH ON GPUS

Local search is a local optimization procedure used in Autodock. It is very time consuming and much of this time is spent in energy evaluation. Autodock uses a hill climbing method to perform local search on selected individuals. Local search is usually performed on a very small percentage of the population and hence offers little scope for parallel breakdown. Hence local search needs to be implemented in a different way on GPUs. We list two possible methods below. The merits of these methods need to be evaluated against an exhaustive training set. This is the reason why we consider this as a topic for a separate paper.

A. *Brute Force*

One could consider evaluating a bunch of random individuals around a sphere of configurable radius centered at the individual selected for Local Search. This is an embarrassingly parallel problem and will scale well on the GPU.

B. *Gradient Method*

Using Gradients for Local Search is not a new idea. Autodock Vina [4] already uses a gradient based approach to optimize local search. Energy evaluation is a multi-variable function of genes. To find the local minima, it would be useful to find the partial derivative of this function with respect to all the genes and use that data intelligently to choose a local minimum. Finding the partial derivative with respect to all the genes for all selected individuals can be grouped together and executed on the GPU.

VII. CONCLUSION AND FUTURE WORK

The core Genetic Algorithm in Autodock has been accelerated using CUDA. Speedups ranging from **10x to 47x** have been observed. The core fitness evaluation function has been sped up **~50x** for typical population sizes. The efficient energy evaluation has laid the ground-work for implementing efficient local searches on GPU. The GPU power could also be used to explore new local search methods that were not viable to the scientists before. The general guidelines discussed in this paper will be useful to anyone who is considering migration to CUDA.

Future work can be to implement efficient GPU based local search and enhancement of results using double precision capability in GPUs. It would be interesting to see how the GPU implementation performs on a FERMI [10] which has 512 cores and 8x double precision power compared to current generation GPUs.

VIII. ACKNOWLEDGMENT

We would like to thank the management of HCL Technologies for funding our research work. We would like to thank the following people from HCL Technologies - Dr. Nagesh for his reviews, Rohini Srinivasan for introducing Autodock to us, Arjun, Madhusudan and Manjunatha Hebbar for supporting our work. We would also like to thank Dr. Garrett, Dr. Trott, Mike Pique and Ruth Huey from TSRI [1] for answering our questions time and again. Special thanks to NVIDIA forums. Finally, we would like to thank the anonymous reviewers for their useful comments.

REFERENCES

[1] TSRI, "*The Scripps Research institute*", http://www.scripps.edu
[2] CUDA, "The CUDA Zone", http://www.nvidia.com/cuda
[3] Timothy Blattner, D. H. (2009, 5 8). "AutoDock Software in Parallel with GPUs – Final Report", http://sourceforge.net/projects/gpuautodock/files/gpuautodock/GPUAUTODOCK.1.0.0/final_prototype_report.pdf/download
[4] Oleg Trott, Arthur J. Olson, "AutoDock Vina: Improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading", *Journal of Computational Chemistry*, vol. 31, No 2, 2010, pp. 455-461
[5] GPGPU, "General Purpose Computation on Graphics Processing Units", http://gpgpu.org
[6] Tom R. Halfhill, "Looking Beyond Graphics", http://www.nvidia.com/content/PDF/fermi_white_papers/T.Halfhill_Looking_Beyond_Graphics.pdf , page 4
[7] Autodock FAQ. "Official Autodock Website", http://autodock.scripps.edu/faqs-help/faq/which-values-of-the-genetic-algorithm-parameters-do-you-normally-use
[8] "Autogrid", http://autodock.scripps.edu/wiki/AutoGrid
[9] "Prefix Sums", *Wikipedia*, http://en.wikipedia.org/wiki/Prefix_sum
[10] NVIDIA Corporation, "*NVIDIA's next generation CUDA Compute Architecture: FERMI*", http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
[11] Sarnath Kannan, "Crossover bug", http://mgl.scripps.edu/forum/viewtopic.php?f=9&t=677&sid=ce704d82725f4255bd0ef5ad7b043811