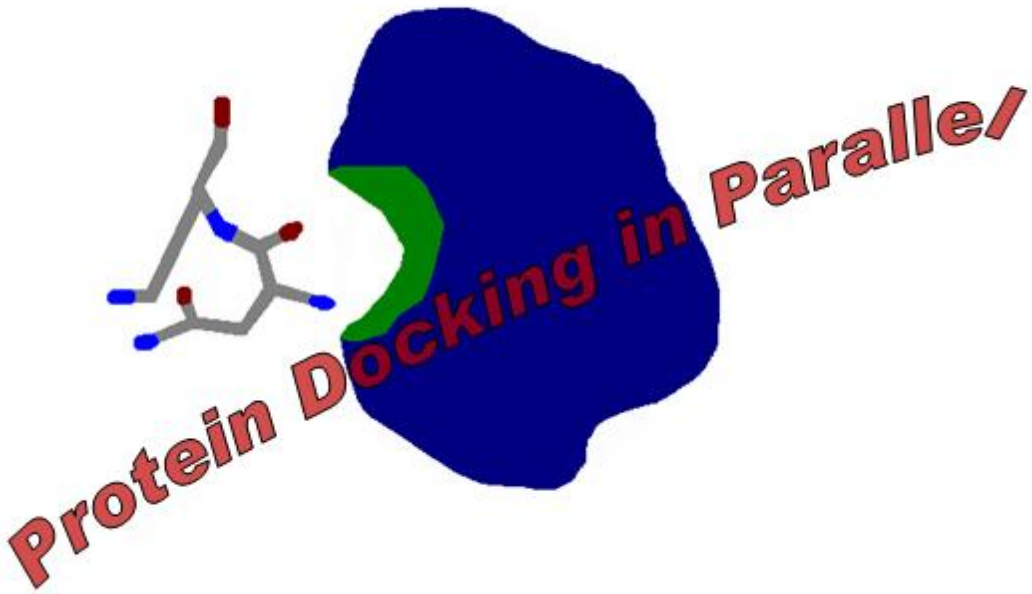
Improving Protein Docking Efficiency with General Purpose Computation for Graphics Processing Units- Formal Report



Project Team E-51:
Timothy Blattner
David Hartman
Andrew Kiel
Adam Mallen
Andrew St. Jean

Faculty Advisor:
Dr. Craig Struble

Table of Contents

I. Executive Summary

Dr. Struble commissioned our team to increase the efficiency of protein docking experiments by implementing parallel processing techniques in the existing AutoDock docking software. Currently AutoDock uses serial processing only and a significant increase in speed is expected with the implementation of parallel computation. For our work this semester, parallelization was facilitated by the CUDA API designed by NVIDIA which allows a program written in C to be automatically compiled in a suitable manner to execute pieces of the program in parallel on a NVIDIA GPU [1]. GPUs are inherently suited to the task of parallel processing due to their many individual cores which act independently [2].

GPUs are an attractive solution to the problem of cost and time effective parallel processing because they are common, thus cheap, and the CUDA API allows for an easy transition into learning to program for them [1]. With our completion of this project, any current AutoDock user with capable hardware (many workstations already exist that meet minimum specifications) can download and compile our new code and use AutoDock as they always have with no apparent change in functionality to the user except for faster run time for experiments. The value of this optimization becomes apparent when one considers the fact that users typically run huge numbers of docking simulations within one experiment and the cost of using powerful supercomputers or large clusters is expensive.

With the release of our final product, as well as our testing results, we are confident that users whom already have supported hardware will have no hesitation in adopting our version. While we hope our improvements will be significant enough to warrant hardware upgrades, the average user is expected to be able to take advantage of our product with no additional cost.

II. Background

Docking

Docking is a general term referring to a variety of operations involving protein-ligand and protein-protein binding simulations. Computer simulation is extremely valuable in structural drug design which works on the principal that two compounds will bind if their structures have compatible surfaces. Docking based on structural compatibility is also used to research the general functions of molecules and macromolecules, and it is widely held that the structure of a molecule is the main factor determining what functions a molecule has and what other molecules it interacts with.

The science of docking, whether protein-protein or protein-ligand interactions are studied, starts with x-ray crystallography. The structure of a molecule is determined using this technique so that each atom in the molecule can be plotted in a 3-D Cartesian system. Then the coordinates are saved to a file for use by a docking program. Once the atomic structure of a protein is known the computer simulation brings the protein and ligand (which in some cases is another protein) into contact with each other (within binding distance). The process used analyzes the structural information and assigns charges based on the binding properties of the elements to each atom in order to compute interaction forces between the atoms when the two molecules are brought together. Each interaction between the atoms of the protein and the ligand creates small forces that minutely change the structure. The changes in structure will create strain energy which must be stored within the atomic bonds. Like all things in nature a protein complex is most likely to form when the final energy of the bonded structure is the lowest. So when a docking program finds a combination of protein and ligand whose surfaces conform well the complex can be labeled as a probable occurrence in nature. These probable conformations are what are useful to researchers as they indicate the likely interactions that a ligand will have with a given protein in a biological environment.

AutoDock

A popular program used to perform docking simulations is AutoDock which is open source software. Open sourced software provides benefits to all those involved with the project such as increased feedback to developers of the software and thus, usually, a more reliable, useful, and effective product. Therefore open sourced software is a valuable tool. AutoDock has been recently credited with being the most referenced docking software available today[1] which alone is one factor that proves its performance capabilities. The software can perform a simulation in as little as a minute and although this is quite fast a drug company can go through millions of individual simulations before having the requisite data to start physically synthesizing a new drug compound. Obviously time is valuable and so we will be using parallel computing techniques to further speed up AutoDock.

Parallel versus Serial Processing

Parallel processing utilizes multiple processors acting simultaneously in order to solve a computational problem. This is opposed to serial processing which must execute instructions one after another on a single processor (as illustrated in Figure 1). By running a process in parallel, problems are solved faster because different pieces of the problem can be solved at the same time (as illustrated in Figure 2). The problem with this strategy comes from determining which pieces of a problem are independent from the result of others. Two processes need to be independent in order to run simultaneously. The algorithms used in the AutoDock software never take advantage of this strategy, though examination of these algorithms shows potential for optimization using parallel processing techniques.

Graphics Processing Units

Graphics Processing Units (GPUs) provide parallel processing over numerous processor cores [2]. When programming for the GPUs, it is useful to utilize their multi-core architecture by modifying pieces of the code to run in parallel over the multiple processors. With the high amount of bandwidth provided and the massive number of graphics processor cores at our disposal on GPUs, parallelism is both feasible and relatively easy to implement.

CUDA

One can send instructions directly to their NVIDIA graphic cards using the CUDA development environment [3]. For our project we plan on using the CUDA application programming interface (API) as a means of accessing the parallel processing power of the GPUs. In this way we can modify the existing AutoDock C++ code to harness the power of parallel computing and improve the efficiency of the program.

III. Target Specifications

The following metrics table contains a list of all specifications formulated by our list of different customer needs. Each entry contains the metric number, the name of the specification, the type of units used to measure the specification, a pair of marginal and ideal values which represent our project's goal, and a relative weight of the importance of meeting each goal (1 is the least important and 5 is the most important).

Metric No.	Metric	Units	Marginal Value	Ideal Value	Weight
1	Difference in mean value between our AutoDock results and original AutoDock results	P-value of T-statistic	(1-P) < 0.05	(1-P) < 0.01	5
2	Difference in variance between our AutoDock results and original AutoDock results	P-value of F-statistic	(1-P) < 0.05	(1-P) < 0.01	5
3	Precision of our AutoDock results, measured as significant digits	Significant Digits	>= 5	>= 10	5
4	On high-end hardware, factor of speed up	Percentage	> 1000%	> 1500%	4
5	On low-end hardware, factor of speed up	Percentage	> 200%	> 1000%	1
6	Runs old AutoDock as normal on machines without supported hardware	Binary	Yes	Yes	5
7	Interface is identical (or similar)	Binary	Yes	Yes	4
8	Released as open source	Binary	Yes	Yes	5
9	Coding style is consistent with proper comments	Binary	Yes	Yes	2
10	Provide measurements of speed tests for customer review	Binary	Yes	Yes	2
11	Provides differences in accuracy and precision for customer review	Binary	Yes	Yes	2

IV. Technical Development & Documentation

This section contains the information regarding the modification made to the AutoDock software. Included is an updated schedule based on our progress made throughout this semester. Also included is a summary of the processes we used to meet our target specifications.

Project Schedule

We adopted an Agile Software Development plan for our project. This involves an iterative and adaptive approach to addressing and completing the tasks we identified. Notice that in our schedule we partitioned the semester into separate two week sections. Each of these counted as a single iteration according to the agile software development plan. Our understanding of this system led us to a plan where each iteration ended with an analysis of the results of that iteration followed by a decision of the specific goals for the upcoming one. With this approach we followed the Agile Software Development's Manifesto by reflecting after each iteration on how to become more effective in the next iteration and then adjust our methods accordingly [4]. In this way we embraced the Agile Software Development's Manifesto by staying flexible in order to welcome and adapt to changing requirements in our project [4]. This means we left our plan open to the possibility of adapting to unforeseen issues as they showed up over the course of the project. However, in order to avoid a loss of efficiency and lack of motivation from the removal of hard time constraints on tasks, we also placed emphasis on concrete results at the end of each iteration showing progress in at least one of the major sections identified in our work breakdown structure.

Code Review

At the beginning of the semester we dedicated two iterations to code review. This involved analyzing the Genetic Algorithm used by AutoDock to determine which portions of the code can be run in parallel using GPUs. There are five main steps in the algorithm, three of which can be run in parallel. Reading Daniel Micevski's article on parallelizing AutoDock, we found the relative CPU time taken for each step [5]. These results are outlined in Figure 4. From this information, we decided to parallelize the two most time intensive functions, eintcal() and trilinterp().

Preparing AutoDock code for CUDA

One of the main challenges of this project was the incompatibility between languages. The CUDA API only interprets C code while AutoDock is written entirely in C++. Unlike C, C++ is an object-oriented language; this means that we had to strip all the parallelizable code of its objects. We built wrapper functions to extract data from the C++ objects and arranged this information in arrays which can be used in C. Using these arrays, we were able to allocate and copy memory locally onto the Graphics Card.

Implementation of CUDA

Once we had the data in CUDA usable format, we began building CUDA functions that would run the code directly on the Graphics Card. This involved rewriting the eintcal and trilinterp functions in order for them to use our prepared data. We used specialized CUDA functions to call our eintcal and trilinterp functions with our prepared data. This enabled us to run the most time consuming portions of AutoDock on GPUs in parallel.

Further Optimizations

Our implementation of CUDA was naive and there was much room optimization. Some calls in particular (i.e. CUDAMEMCPY), which involve memory transfers from RAM to the Graphics Card, are highly time intensive. We spent half an iteration minimizing these calls and removing any unnecessary or redundant code. The other half of this iteration was spent optimizing AutoDock's use of the GPUs. CUDA runs parallel portions of an application on a device as kernels. These kernels are executed as a grid of thread blocks which synchronize their execution and share memory [3]. CUDA allows the programmer to specify the thread block size, which ultimately determines the number of threads allowed to run per block. Traditionally the size is assigned as a multiple of 32. We ran preliminary speed tests using variable sizes and determined that a thread block size of 128 was optimal for AutoDock (illustrated in Figure 4).

Additional Information

Environmental Impact Statement

Our project is improving an existing software package. The only environmental impact that could be foreseen from our project is that users may purchase higher end graphics cards to run our specialized code. In this case, there could be increased power consumption by the computer while running AutoDock.

Applicable Standards

The teams development had to comply with the standards of the CUDA API. Our code had to follow particular syntax and follow certain protocols in order to ensure proper use of the graphics card.

Regulatory Compliance

AutoDock is released under the GNU General Public License, which is a widely used free software license. We therefore were obliged to release the code under the same license.

Safety Evaluations

Our version of AutoDock has the same amount of risk as running the original software package. If the end-users computer does not support our modified version, the program will gracefully exit.

Legal Issues

We faced no legal issues during the course of our development process. AutoDock is an open-source software package and we released our version under the same license.

IV. Validation & Verification

The following description of our validation and verification process is split up into four main parts. The first is an outline of the simple tests performed to validate the binary target specifications. Next, is a description of the final and thorough statistical analysis we have performed in order to provide conclusive empirical evidence that our modifications to the program have not altered the reliability of the results. The third section describes the results of our speed analysis on the difference in run times between our program and the original over a variety of input sizes. The fourth and last part describes the way in which we have released our program to the public and how the release meets our remaining target specifications.

Part (1)

Metrics 6, 7, and 9 are binary criteria and thus can be analyzed by simple inspection to determine whether the team has met the requirement or not. The option to compile the original program or our modified version is made at compile time, so users who want to run the original code can do so in exactly the same way they always have. Users who want to run our modified program need only enter a simple command line option during compilation to run our version. After this small difference, running our version of AutoDock is identical to the original. In this way, our changes to the program are transparent to the user.

Metric 3, similarly can be validated through simple inspection. It turns out that the original AutoDock program's log files store the results of the free binding energy with only 2 decimal places. Since all our preliminary trial runs of our version of the program also produce results with 2 decimal places, we consider this target specification validated.

Part (2)

The bulk of our validation involved a thorough analysis of the difference in the results of the original and modified versions of the program over a variety of input ligands-protein pairs and performance parameters. These tests are designed to validate metrics 1 and 2. We include both our methodology and results for user review to serve as final validation that we have satisfied our major specification: our modified program yields equally reliable results as the original program. The input data and parameter files are included for download on our website *http://gpuautodock.sourceforge.net*. The experiments referred to as Input-1, Input-2, and Input-3 come from data and parameter files marked as input1, input2, and input3 respectively on our website.

The following outlines the major steps of the final experiments and analysis we have performed:

Experiment Outline: 3 experiments with different ligand/protein inputs that came as examples with the AutoDock 4 download. Each consist of a relatively large number (the number differs between each experiment due to time constraints) of control runs using the default parameter files that came with those examples with the following changes:

ga_pop_size set to 1500
ga_num_evals set to 250000000
ga_num_generations set to 270000
ga_elitism set to 10

Also, with each experiment we ran a relatively smaller number (again, this number differs between each experiment because of time constraints) of experimental runs using our modified program with the same ligand/protein inputs and parameter files as the control runs. We would have liked to have run as many experimental runs as control runs, but because of the lengthy run times of AutoDock simulations we were not able to. We overcame this problem for the control runs by utilizing the MSCS Condor Pool to use over 420 days (over 11,000 hours) of computing time in just a few weeks of real time by running multiple control runs simultaneously across a pool of machines in the MSCS department. However, since we only have access to one machine with capable graphics hardware, the experimental runs must all be run sequentially. This, of course, takes longer to generate as many results as the control runs which could be run in parallel on the collection of dozens of machines in the Condor Pool with AutoDock installed.

Statistical Analysis Outline:

On each of these experiments we performed a non-parametric Wilcoxon rank-sum test on the resulting free binding energy as well as the conformation of the ligand that yielded that energy result. We chose this nonparametric test because it does not assume that the distributions generating the samples are normal. After performing some exploration of the results, it was not obvious that the distributions were normal and so a T-test was not appropriate. The P-values returned by this non-parametric test are similar to those returned by the T-test in that high P-values denote high confidence in the hypothesis that the two samples came from the same distribution and low P-values denote low confidence in the same hypothesis.

For the free binding energies we simply compared the control and experimental distributions of resulting energies using the rank-sum test. However, to test the hypothesis that the resulting conformations from our program and the original did not differ, we had to first transform the data into simpler distributions of single values in order to run the rank-sum test. Originally the conformations are represented as a set of multiple values which correspond to the ligand's binding position, orientation, and torsional bond rotations relative to the protein to which it is docking. First we normalized each of these values to the [0,1] interval. Then we computed a “centroid” conformation by averaging all corresponding values from the distribution of control results. To complete our transformation we took the euclidean distance of each control and experimental conformation from this “centroid” conformation to get two clean distributions of single values instead of multiple values. Then, on this transformed data we ran the same rank-sum statistical test.

Figures 5 – 17 show the histograms of the distributions of the results of the 3 different experiments. The following table describes some statistics and results from each experiment:

	Input-1	Input-2	Input-3
Number of control runs	216	1520	248
Number of experimental runs	10	40	22
P-value for free binding energy test	0.78	0.64	0.5
P-value for conformation test	0.77	0.86	0.86
Minimum control free binding energy	-17.07	-8.02	-6.29
Maximum control free binding energy	-6.38	-7.97	-6.28
Minimum experimental free binding energy	-15.81	-8.02	-6.29
Maximum experimental free binding energy	-8.54	-7.97	-6.28

The first thing to notice about these results is the fact that none of the P-values we computed are larger than 0.95 which was our marginal value in the Target Specifications section. However, the preliminary results of some further exploration suggests that even when performing the rank-sum test on two different distributions of control values the resulting P-values were less than 0.95. This would mean that for some input parameters the variation from the random nature of the genetic algorithm would make it nearly impossible for distributions of our program's results to get P-values higher than 0.95 when compared with control distributions since it would also be nearly impossible for two control distributions to get P-values higher than 0.95. Unfortunately, we did not have enough time in the semester to follow up on this conjecture with anything conclusive.

The next thing to notice, though, is that all the P-values for the conformation tests are above 0.75. This grants relatively high confidence to the hypothesis that these two samples come from the same distribution, even if it does not meet our original specifications. However, comparing these samples of data points with several dimensions of values is a difficult task, and our centroid distance transformation method is a very simple and naive approach to solving this problem. Though the normalization step converts all the different dimension's values to the “same” scale, outlier values can skew a certain dimension's relative weight to the distance calculation. Also, in the realistic situation where resulting conformations cluster around more than one point, our approach would compute each conformation's distance from some average point somewhere between the actual cluster centers, regardless of which actual cluster the conformation came from. A more sophisticated method of measuring the similarity of conformations from our program and the original may be able to handle these problems with our naive method and yield more conclusive results.

The last important thing we would like to point out is the spread of values for the free binding energies. Notice that in every case the entire sample of energies from our experimental runs were contained within the minimum and maximum energies of the sample from the control runs. Furthermore, in Input-2 and Input-3 the range of energy values from the experimental and control samples are exactly the same. So, even though the P-values are very inconclusive for our tests on the free binding energies, this shows that all of the output energies from our program fall into the range of values produced by the original program. This result gives confidence that our program's output is as reliable as the original and suggests that we were, again, using the wrong test statistic to measure the reliability of our program's results.

Part (3)

Part 3 describes our experiments for testing the speed-up achieved by our modifications to the code. That is, it describes our validation of metric 4. Notice that we do not include a description of experiments for validating metric 5. Due to time constraints and because it was ranked as the least priority metric on our list of target specifications, we dropped validation of this metric from our schedule. Our run time experiments were conducted as follows:

Each benchmark run consisted of 6 control and 6 experimental runs. We ran these using 3 different input population sizes of 1,000, 10,000, and 100,000 to see the impact that population size had on the run time. We chose this specific parameter to alter for the reason that our code was designed to parallelize across individual population members. This experiment was performed on input 3 described in the previous section. We chose not to run these benchmark experiments with input 1 or input 2 because we did not have enough time to run a sufficient number of trials for either.

Figure 18 outlines the average run time of the control and experimental programs over the 3 different population sizes. There are two important points to take away from these results. The first is the fact that even on the largest sized population our program took, on average, about 96% as much time as the control run to complete. This means that our program only achieved about 4% speed-up, while our target specification called for over 1000% speed-up in the marginal case. However, the second important detail is the fact that, as the population size increased, the difference in performance between the control and experimental runs also increased. This means that, though we did not achieve even close to the speed-up we were planning on, we have succeeded in creating a program which outperforms the original to an increasing degree as the population size increases.

Part (4)

This section describes the way we have released our software, documents, and scripts to the public in order to meet our remaining target specifications. The manifestation of our key deliverable consists of a web page on *http://gpuautodock.sourceforge.net* which will make available for free public download the following items:

The original AutoDock4 source code.
Our modified AutoDock4 source code, as well as all files needed to make/compile our version.
The readme documentation which includes instructions to make/compile our program.
All documents and deliverables turned in for this course during the year, including this report.
All validation experiments and their results (included in this report).
All input data and parameter files used for testing discussed in this document.
All scripts used for the tests and analysis discussed in this document.
An "Optimization" document outlining ideas for future optimization for other groups interested in continuing this project.
A "Flow Diagram" document outlining the function call flow relevant to our modifications.

V. Conclusion & Future Enhancements

Throughout the design and development of our project, many of our initial ideas for changes to AutoDock had to be cut due to time constraints. These changes could be implemented by a future senior design team or, as this is an open-source software package, by other programmers interested in optimizing AutoDock. Some of these changes include: using our CUDA implementation in more locations in the software, parallelizing other portions of the genetic algorithm, adding support for multiple or higher performance graphics cards and implementing optimization techniques in the graphics card kernels. For more information regarding our team's ideas for further improvements see our "Optimization" document on our website.

Another important issue that requires additional work lies in addressing the problems in our validation techniques for the accuracy and reliability of our program's results discussed earlier in this document. More investigation into the P-values of rank-sum tests between samples of control runs would give us a better idea of what is considered a "good" P-value for validating the reliability of our program. A more sophisticated technique for measuring similarity between samples of conformations would further help in validating reliability. Also, simply running more tests with a larger variety of inputs would help to the same end.

Overall our project can be considered a success. Team E-51 worked well together in order to meet many of our target specifications and deliver a reliable and working product to our customers. Although we did not meet some of the specifications, we were successful in running parts of the program on the graphics card and yielded similar and reliable results. With this first step in the project we hope to have opened to door for further work on this project.

VII. References

[1] AutoDock. 24 September 2008.
<http://www.AutoDock.scripps.edu>

[2] “What is CUDA?” CUDA Zone. NVIDIA. 14 October 2008.
<http://www.nvidia.com/object/cuda_what_is.html>

[3] “CUDA education.” CUDA Zone. NVIDIA. 14 October 2008.
<http://www.nvidia.com/object/cuda_education.html>

[4] “Principles behind the Agile Manifesto” Manifesto for Agile Software Development. 31 Jan. 2009.
<http://agilemanifesto.org/principles.html>

[5] Micevski, Daniel, “Optimizing Autodock with CUDA.” Victorian Partnership For Advanced Computing. Jan–Feb 2009.
<http://www.vpac.org/files/OptimizingAutodockwithCUDA.pdf>

VIII. Figures

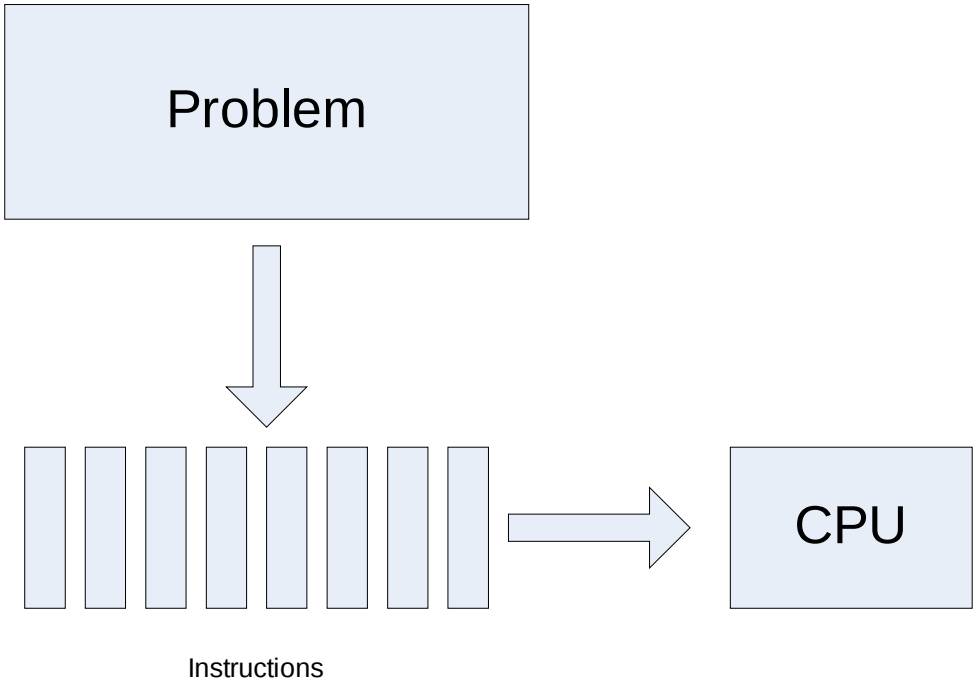


Figure 1: Serial Processing - sequentially solving each piece of a problem

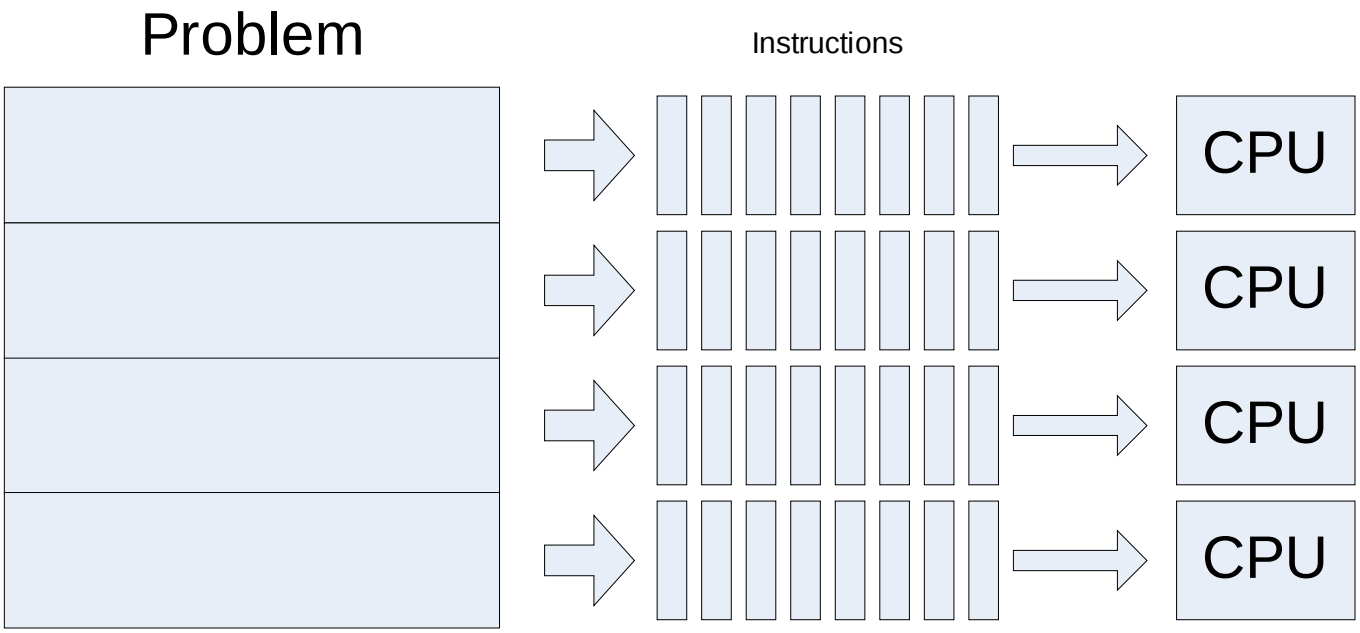


Figure 2: Parallel Processing - simultaneously solving pieces of a problem

Function	Total % Program Run Time
eintcal()	36.69%
trilinterp()	24.41%
RealVector::clone()	5.95%
torsion()	4.17%
ignlgi()	2.38%

Figure 3: Program Run Time per Function

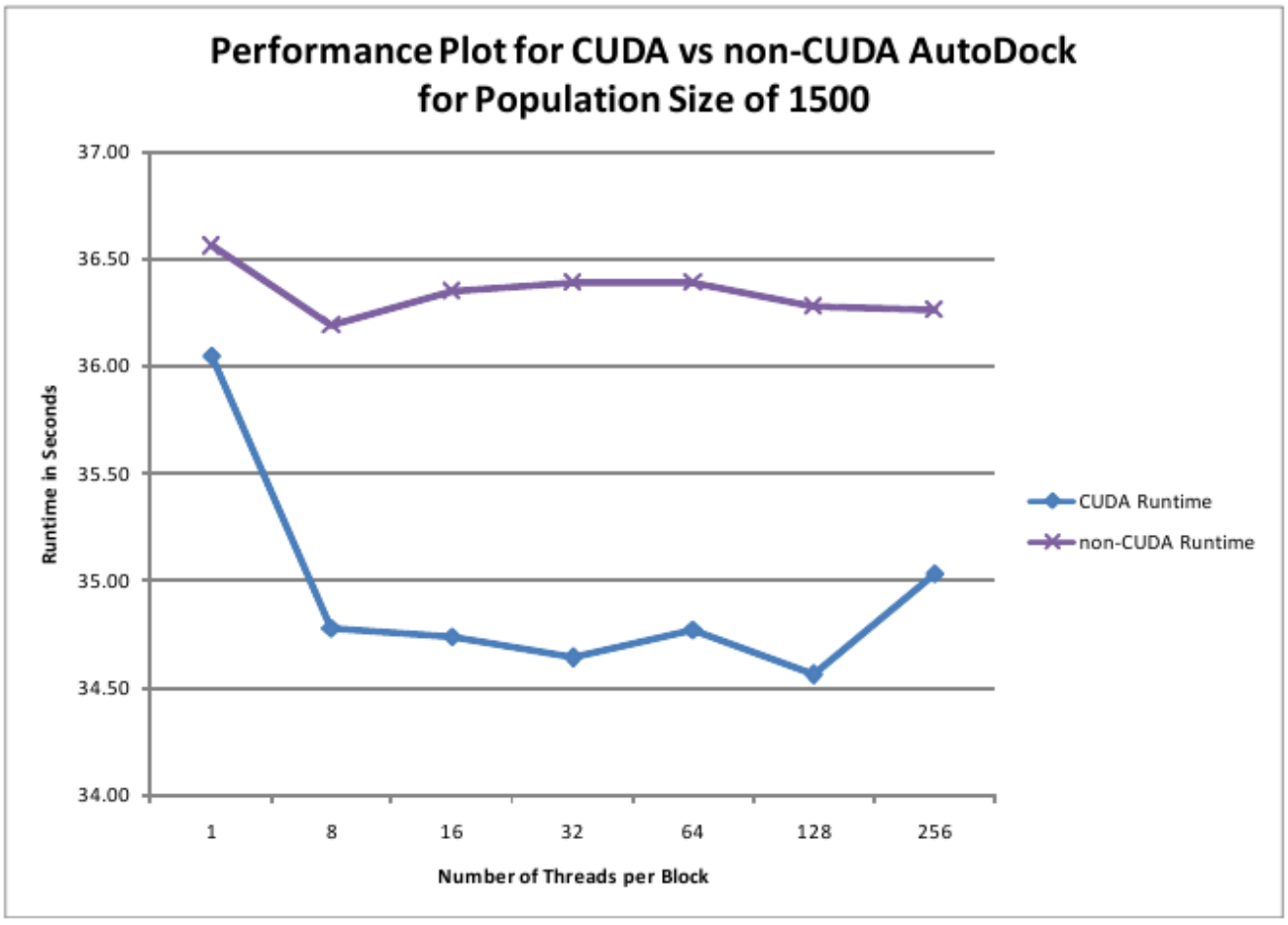


Figure 4: Determining optimal thread block size

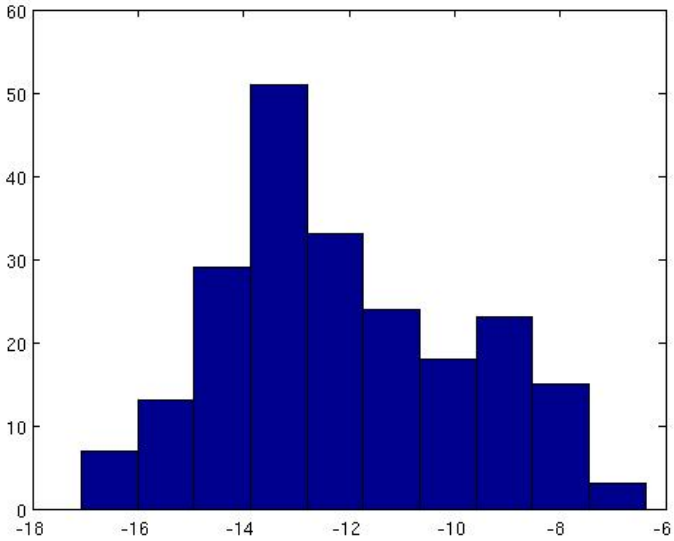


Figure 5: Input-1 Control Energies Histogram

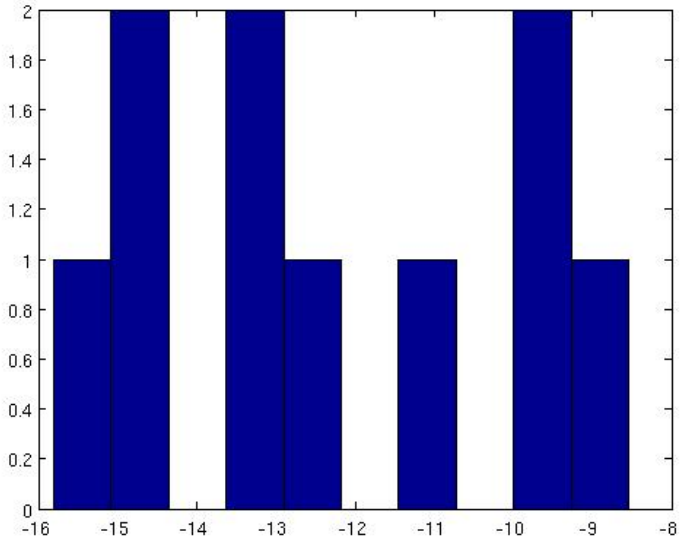


Figure 6: Input-1 Experimental Energies Histogram

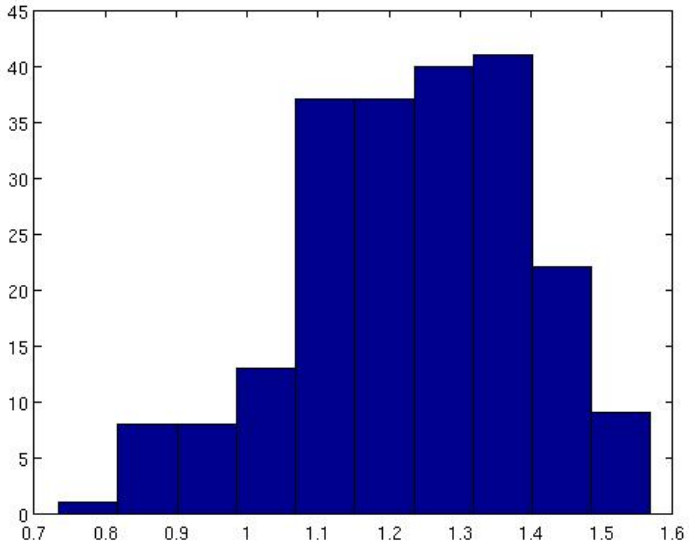


Figure 7: Input-1 Control Conformation Histogram

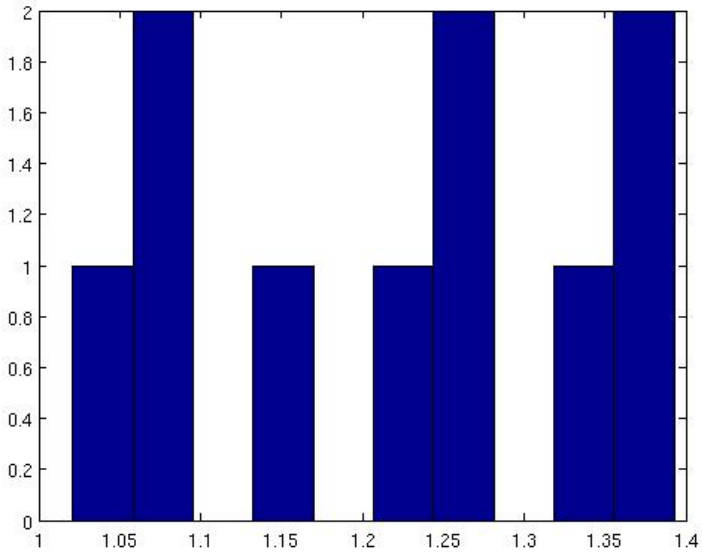


Figure 8: Input-1 Experimental Conformation Histogram

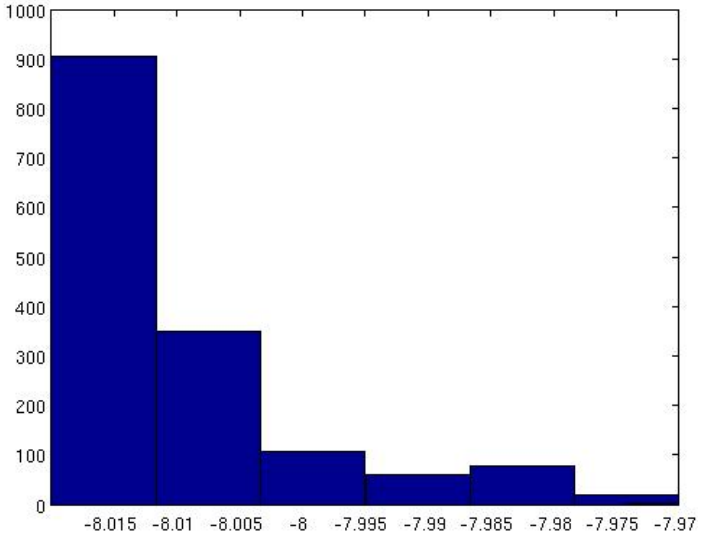


Figure 9: Input-2 Control Energies Histogram

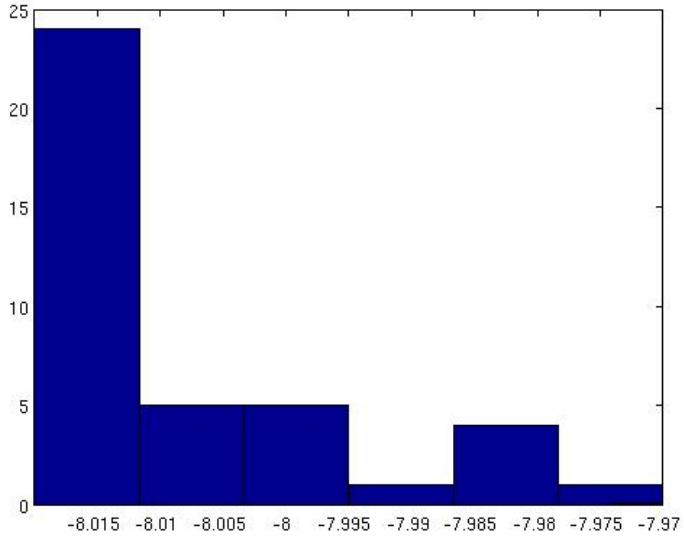


Figure 10: Input-2 Experimental Energies Histogram

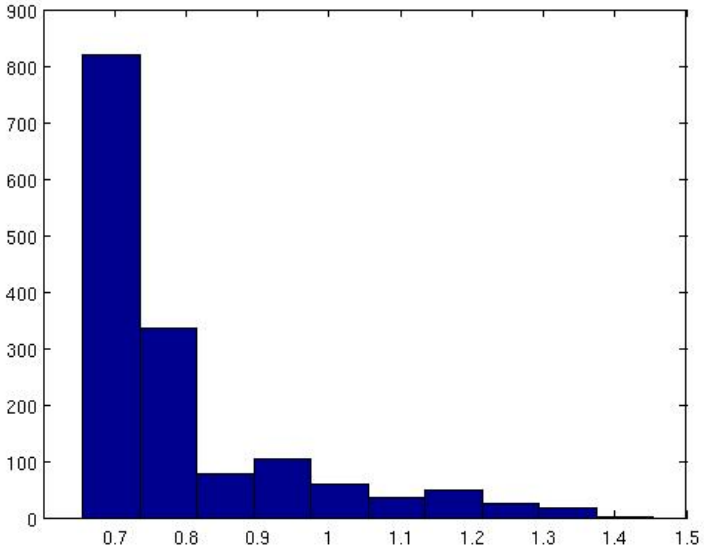


Figure 11: Input-2 Control Conformation Histogram

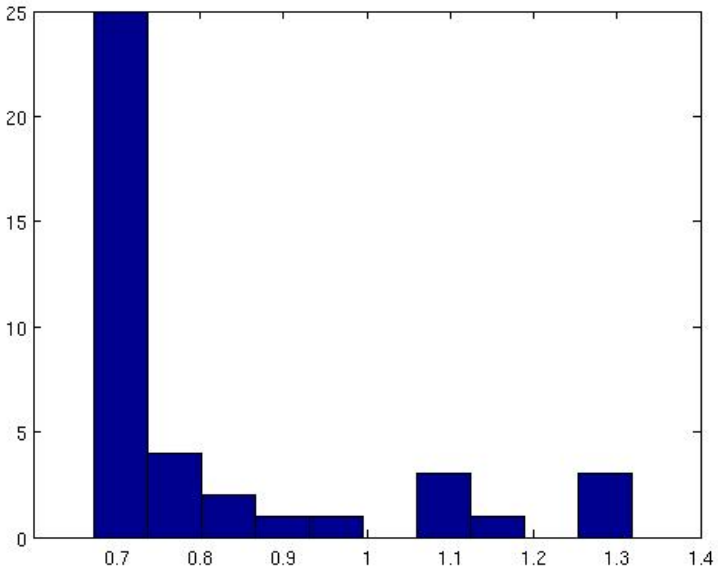


Figure 12: Input-2 Experimental Conformation Histogram

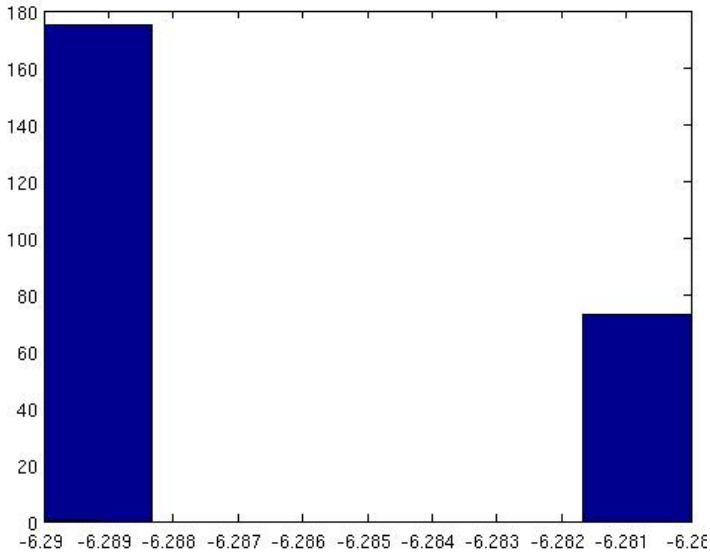


Figure 9: Input-3 Control Energies Histogram

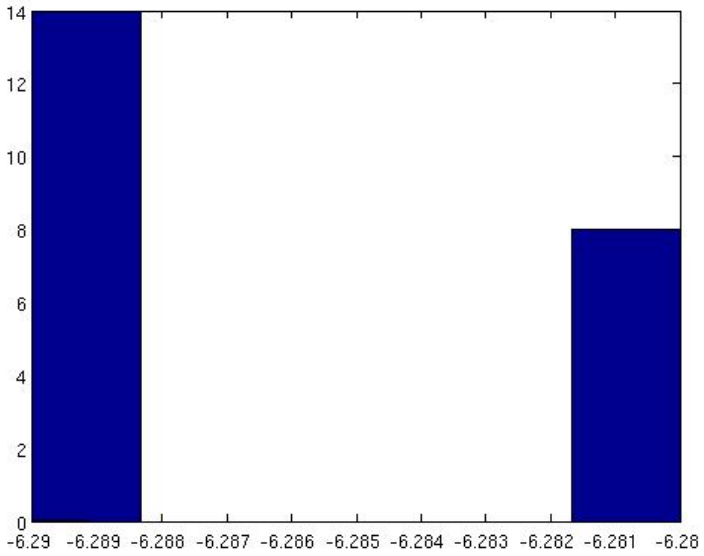


Figure 10: Input-3 Experimental Energies Histogram

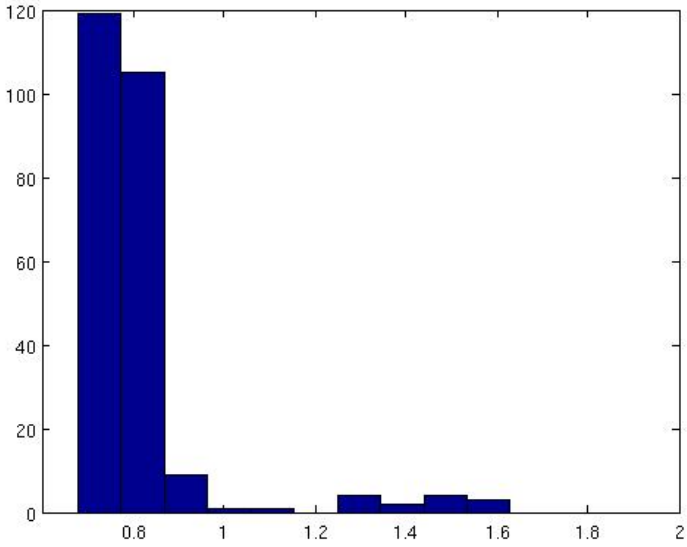


Figure 11: Input-3 Control Conformation Histogram

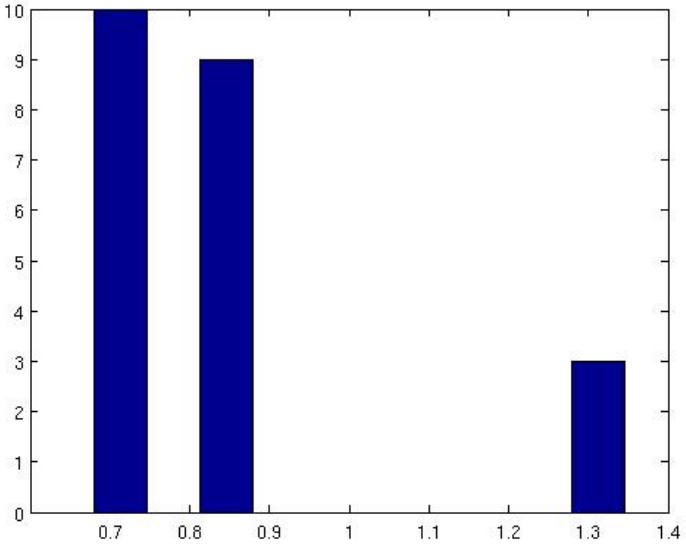


Figure 12: Input-3 Experimental Conformation Histogram

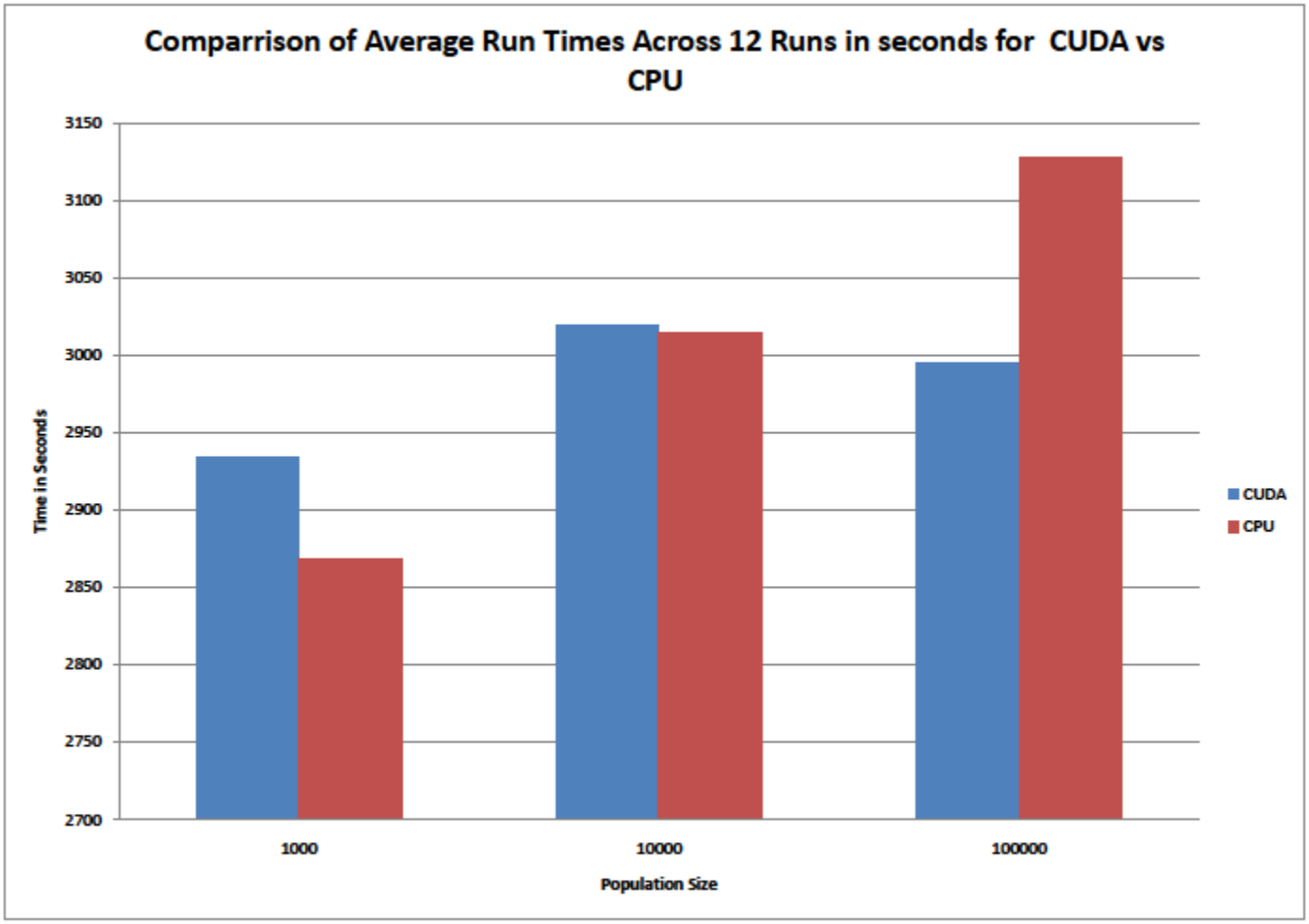


Figure 18: Comparison of Average Run Time between control and experimental AutoDock runs over 1,000, 10,000, and 100,000 member population sizes.