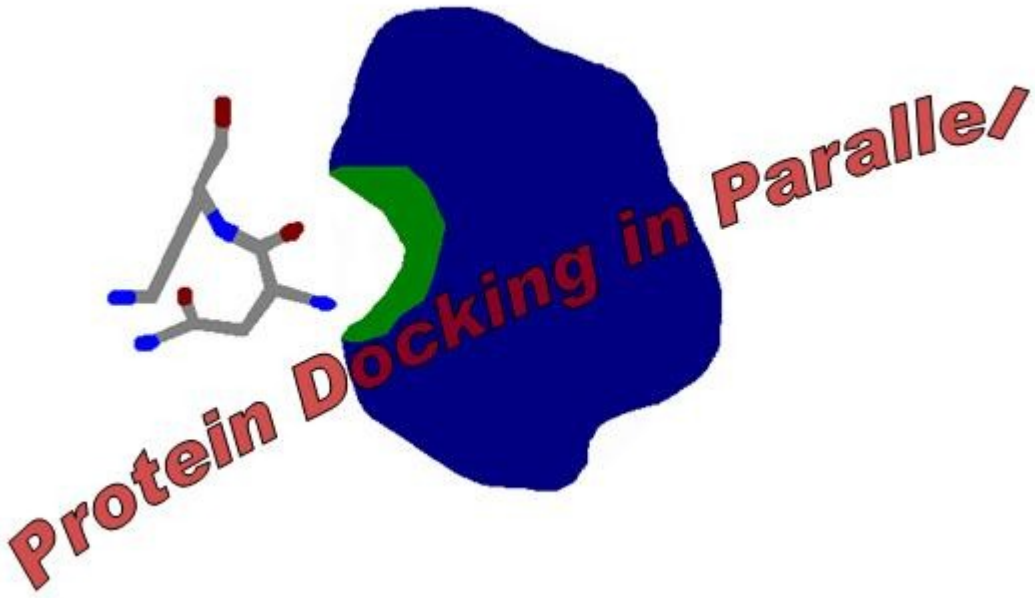
Improving Protein Docking Efficiency with General Purpose Computation for Graphics Processing Units – Generated and Final Concepts



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

Faculty Advisor:
Dr. Craig Struble

Table of Contents

I. Introduction

The Customer Needs and Target Specification Document provides a layout of the problem, the needs provided by the customers for improving the AutoDock software, and our specific quantifiable goals for satisfying these needs.

The following sections are included in this document:

- Project Background
- Project Objective Statement
- Customer Needs
- Customer Needs Classification
- Target Specifications

II. Background

AutoDock

AutoDock can perform a docking simulation in as little as a minute. 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 [1]. We will be using parallel computing techniques to speed up AutoDock.

Parallel Processing

Parallel processing uses multiple processors acting simultaneously to solve a computational problem [2]. By running a process in parallel, problems are solved faster because different pieces of the problem can be solved at the same time [2]. The algorithms used in the AutoDock software never take advantage of this strategy, though examination of these algorithms shows potential for speed up using parallel processing techniques.

Graphics Processing Units

Graphics Processing Units (GPUs) provide parallel processing over numerous processor cores [3]. When programming for the GPUs, it is useful to use their multi-core architecture by modifying pieces of the code to run in parallel over their multiple processors. Unfortunately, GPUs only support single-precision floating point values [4] in comparison to programming on double-precision floating point CPUs.

CUDA

One can send instructions directly to NVIDIA graphic cards using the CUDA Application Programming Interface (API) [5]. The CUDA API will be used to access 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. Project Objective Statement

Problem Statement

The popular AutoDock software uses serial processing techniques to simulate protein docking. AutoDock does not take advantage of the parallel processing power of modern computer architecture such as multi-core processor chips and graphical processing units.

Project Objective

Reduce docking time in the AutoDock simulation by using parallel processing on GPUs, which will be tested and implemented by working 20 man hours per week until April 29, 2009 without funding.

IV. Customer Needs Classification

Customer Needs Categories and Details

1. Accuracy/Precision
    - AutoDock software should not cause a significant loss of accuracy in the program's output.
    - Customers should be satisfied with the output despite the loss of precision from performing computations on single precision GPUs instead of double precision CPUs.
    - Provide results of CUDA AutoDock test data and differences in comparison to old AutoDock results for customer review.
2. Compatibility
    - Customers without the necessary CUDA capable GPU hardware will be able to compile and run AutoDock the same as before our changes to the code.
    - CUDA AutoDock should make changes transparent to the user.
    - Command line tools of AutoDock are consistent with prior version.
    - Code that we add/modify in the software should follow the same styles as the existing code and provide clear comments for the customer to understand.
    - Released as open source software.
3. Cost
    - We should provide a low cost mechanism for parallel processing in order to speed up the runtime of an AutoDock docking simulation. We plan on using both high-

end and low-end GPUs to meet this need. So, since this design decision has already been made, the real customer need is making sure that the cost of the CUDA capable GPU hardware is worth the speed up trade-off.
    - Our modified AutoDock code should be released as open-source software.
4. Speed
    - CUDA AutoDock will provide a speedup of at least 10 times.
    - We should provide measurements of speedups achieved for customer review.

Prioritized Customer Needs

1. Has minimal loss of accuracy in results
2. Is precise enough for customer to use
3. Cost of CUDA capable hardware is worth the speed up our modifications provide
4. Runs as usual if customer does not have CUDA capable hardware
5. User interaction with AutoDock commands when running our modified program is identical (or at least similar) to previous AutoDock interface
6. Results in a speed up of an order of magnitude or more with high-end hardware
7. Released as open source software
8. Coding style remains consistent with previous AutoDock code and provides clear comments
9. Provides differences in accuracy and precision for customer review
10. Provide measurements of speed tests for customer review

V. Benchmarking and Testing

Benchmarking

In our project, the competitive benchmark is the original AutoDock program. Our target specifications are almost exclusively relative to the this original program. In other words, the metrics we use in our target specifications are all relational, comparing the original program to what we want to see in our modifications of it.

Testing

To compare our modified AutoDock with the original program we must first have controlled test runs on the original program and compare these results with tests on the modified program. Even though we plan on using the same seed for random number generation—in an attempt to minimize the differences between results from separate runs with the same input—parts of the algorithm are still stochastic. This will yield some randomness in the results which makes it important to statistically compare the results of a number of test runs; this means statistically comparing the distribution of results from the controlled test runs with the distribution of results from our modified program. It is obvious that we must use the same set of inputs on both tests and run both tests on the same machine for comparisons. We will also want to run the same experiments on a machine with low-end GPU hardware so we can compare the speed up factors of low-end and high-end GPU hardware. For our project we plan on using Dr. Struble's gpgpu machine for tests on high-end GPU hardware and the Systems Lab's MicroWulf cluster for tests on low-end GPU hardware (specifications for both of these machines can be found in the appendix). For sample input we will be using two data sets from researchers here at Marquette which include sample ligands and proteins from Dr. Struble's Bio-Informatics lab and sample ligands and proteins from Dr. Sem and his assistant Andrew Olson's research.

However, because of the complexity and lengthy run times of these experiments we have not yet performed them in order to acquire benchmark values. This means that the specifications listed in the following Target Specifications section are measured in one of two ways. Either they are measured in proportions (percentages) of results from our modified program and results of the original program, or they are measured in statistical scores which reflect the statistical comparisons of results from our modified program and the results of the original program. Because we have not run the controlled tests yet, we are assuming that the distributions of results will be normal. The specifications in the following section include T-test and F-test statistical analyses which assume normal distributions. However, we still plan on running normal probability tests on the distributions first, to make sure that our assumptions are correct. If they are not, then we plan on recomputing the statistical specifications listed in the next section to be measured using non-parametric statistical analyses instead of the traditional T-test and F-test analyses.

VI. 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 associated customer need, 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.	Need	Metric	Units	Marginal Value	Ideal Value	Weight
1	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	1	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	2	Precision of our AutoDock results, measured as significant digits	Significant Digits	>= 5	>= 10	5
4	3,6	On high-end hardware, factor of speed up	Percentage	> 1000%	> 1500%	4
5	3	On low-end hardware, factor of speed up	Percentage	> 200%	> 1000%	1
6	4	Runs old AutoDock as normal on machines without supported hardware	Binary	Yes	Yes	5
7	5	Interface is identical (or similar)	Binary	Yes	Yes	4
8	7	Released as open source	Binary	Yes	Yes	5
9	8	Coding style is consistent with proper comments	Binary	Yes	Yes	2
10	9	Provide measurements of speed tests for customer review	Binary	Yes	Yes	2
11	10	Provides differences in accuracy and precision for customer review	Binary	Yes	Yes	2

VII. Problem Decomposition

Introduction

Before generating concepts and deciding on a final solution, we need to decompose our problem. Once the sub-problems have been identified, the following section will address the sub-problems for which solution methods have already been decided upon. The next section will discuss the generated concepts the team has come up with as possible solutions to the remaining, unsolved, sub-problems.

The following diagram outlines the basic subdivisions of our problem.

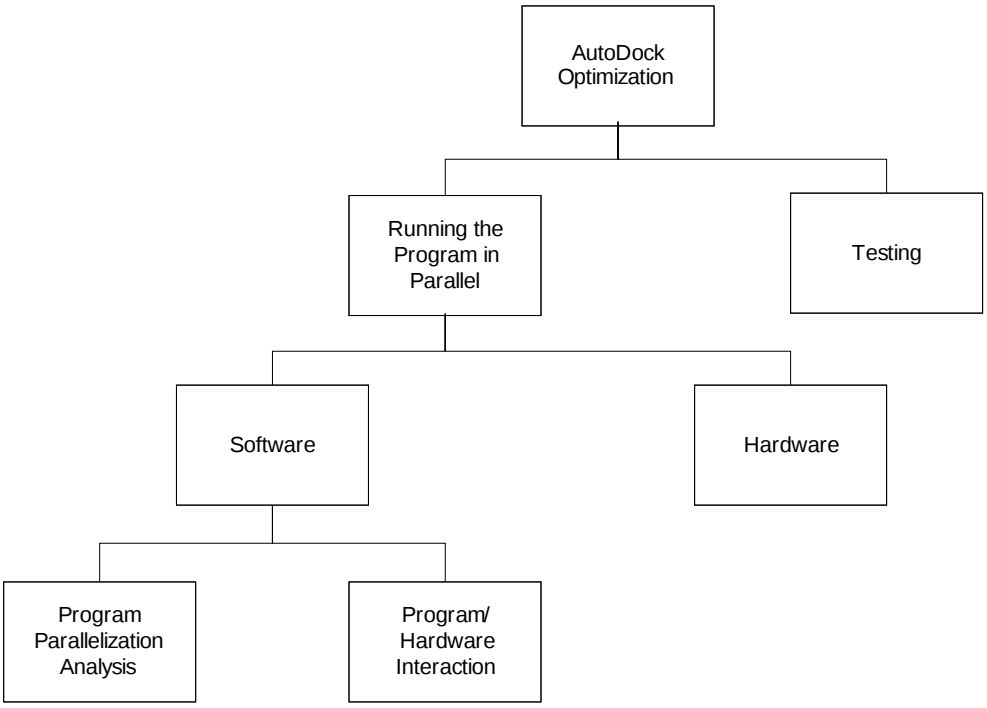


Figure 7.1 Problem Decomposition for Optimizing AutoDock

AutoDock Optimization

At the most basic level our project requires solving two distinct problems. One is the actual parallelization of the program, and the other is the methods and analysis of testing the differences in speed, accuracy, and precision between our program and the original.

Running the Program in Parallel

There are two aspects of the process of parallelizing the program which we have made into separate sub-problems. The first problem is deciding what hardware to use. The other sub-problem involves the software characteristics of parallelizing the program. This software aspect includes both the problem of where to run code in parallel and how to run that code on the chosen hardware.

Hardware

The basic hardware problem this project faces is the decision of a hardware medium which allows for the running of (and possibly optimizes for) parallelized programs.

Program/Hardware Interaction

Running parallelized programs is in essence different from running serial programs because it requires a fundamentally different interaction between software and hardware. In order to run a parallelized program, special code is needed to actually run the software on hardware which is capable of running such programs in parallel. Determining how the parallelized software will run on the hardware has been identified as one of our critical sub-problems.

Program Parallelization Analysis

An essential problem facing the project is identifying which pieces of the program can be run in parallel without ruining its functionality. Deciding what methods of code analysis will allow the team to efficiently discover the sections of the program which can be run in parallel is the basic aspect of this sub-problem.

Testing

Besides the actual modifications to the program, the other problem facing the project is decided on methods of testing the differences between our modified AutoDock program and the original. Because of the stochastic nature of the algorithms used by the program, intense testing is required to convince customers that the results of our program are not statistically different from those of the original program. The problem is basically defined as the decision of an appropriate testing method which will yield sufficient results to convince customers that there is no significant difference between the programs.

VIII. Sub-Problems with Solutions

Introduction

This section outlines the solutions to sub-problems already decided upon by the team before this step of the project's development. Some of these problems' solutions were decided arbitrarily because calculated solutions to these problems lie outside of the scope of the project

and others were decided on in the writing of previous deliverables. Other aspects that we've already arbitrarily decided on are basic concepts which satisfy some of our simple binary target specifications. These include the decision to release our program as open-source along with our testing procedure and test results, the decisions to keep the interface the same, and the decision to keep the same coding styles.

Hardware Solutions

The original problem statement from our advisor already assumed that the parallelization of the program would be done on NVIDIA GPUs using their CUDA API to run our modified code on them in parallel. So the sub-problems outlined above which dealt with hardware issues (the choice of hardware and how the software would interact with it) were arbitrarily chosen by our advisor who wrote the problem statement for this project. This means that an analysis of other possible solutions to these problems lies outside of the scope of the problem laid out in the problem statement.

Testing

The problem of deciding on a testing method was already solved in the previous section. For our methods of testing see Section V on Benchmarking and Testing. Because the team put significant effort in the generation of these testing methods earlier we decided to accept them and not spend time considering alternatives at this stage of development.

IX. Concepts for Sub-Problems without Solutions

External Concepts

A number of outside readings have led to some different approaches to parallelization considered by the team.

Explicit vs. Implicit Parallel Programming

- Explicit programming requires the algorithm to specify how the processors will cooperate [7].
- Implicit parallel programming forces the compiler to create the constructs necessary to run a sequential program on a parallel architecture. CUDA allows for this by hiding the work of actually distributing tasks on the NVIDIA GPU processors from the programmer [7].

Data Parallelism vs. Control Parallelism

- Data parallelism consists of identical processing done on different input data simultaneously [7]. Because CUDA forces programmers to mark off sections of sequential code that will be run in parallel, the only difference between parallel executions of CUDA code can be the input data.
- Control parallelism consists of different pieces of the algorithm being run simultaneously, possibly with the same input data [7].

Domain Decomposition vs. Functional Decomposition

- Domain decomposition is a parallelization paradigm in which the programmer divides up the *data* of a program to be operated on concurrently [8].
- Functional decomposition is a parallelization paradigm in which the programmer divides up the *functionality* of the program and runs those pieces concurrently [8].

Although domain decomposition and data parallelism may seem the same and functional decomposition and control parallelism may seem the same, our team made a distinction between these two design decisions. In regards to our project the concept of data parallelism refers to the lower level data decomposition forced on the programmer by the way CUDA runs the same code in parallel making control parallelism impossible. However, the other design decision refers to the higher level decision of where to run pieces of the algorithm in parallel. Here, domain decomposition refers to an approach which isolates pieces of the algorithm where the same computations are performed many times on a different small piece of a large data-structure, whereas functional decomposition refers to an approach which isolates different functions in the algorithm that can be run simultaneously and still yield consistent and correct results.

Internal Concepts

Two essential problems remain for the team to solve. One is the method of program analysis which will result in an indication of sections of the program which can be run in parallel. The other is how to translate those pieces of the algorithm into CUDA code to run in parallel.

Program Analysis Concepts

Top-Down Approaches:

These concepts describe approaches which analyze a general diagram of the algorithm for pieces which can be run in parallel and then find the corresponding AutoDock code to modify.

1. Analyze a general diagram of the algorithm put together by contacting outside sources. Examples of such outside sources include our advisor Dr. Struble, Dr. Sem, or other AutoDock users.

2. Analyze our own general diagram of the algorithm put together according to how the team would have organized the program. This would require the team to decide how we would have decomposed the program into separate functions if we had been the programmers.

Bottom-Up Approaches:

These concepts describe approaches which take the actual AutoDock source code and create general diagrams of the algorithms which can be analyzed for sections which can be run in parallel.

1. Use reverse engineering to create Class or UML diagrams from the source code. We could analyze these diagrams to find sections of the program that could be run in parallel. Some examples of software we could look into using for this are the following: Rational Rose, StarUML, Argo, and Together J.

2. Use reverse engineering to create call/function graphs or sequence diagrams from the source code. Similarly we could analyze these diagrams to run sections of AutoDock in parallel. We do not know of any software which can create these diagrams, but the software listed above may have features that do this that we are unaware of.

3. Split up the source code for a simple code review. This alternative to reverse engineering puts the responsibility of creating high level diagrams on the team instead of outside software.

CUDA Translation Concepts

1. Memory Coalescing is an approach which takes advantage of hardware optimizations on the NVIDIA GPUs. If threads simultaneously executing the same code normally perform memory read/write functions, they must be done sequentially which drastically reduces performance. However, if the memory operations meet certain criteria (the main one being that the *i*th thread performs the memory operation on the *i*th entry of the array of data) then all the threads' memory operations can be done simultaneously [9].

2. A performance gain is also accomplished by using the shared memory space of each thread block instead of the global memory space of the GPU. Each thread can access the shared memory space much faster than the global memory space. This means the initial overhead of copying data-structures into shared memory space can sometimes end up beneficial because it saves the time it would take for all the subsequent memory operations to go out to the graphics card's global memory [9].

Although we only list two CUDA “tricks” to improve performance over naive implementations, the CUDA reference manual and our weekly CUDA tutorial sessions with our advisor, Dr. Struble, will probably give the team more ideas in the future. We mention this because the team believes that this list of concepts will grow as we understand more of the reference manual and conduct more tutorial sessions with our advisor.

X. Concept Selection

After considering the needs of our customers and decomposing the various possibilities presented, we have selected concepts that will be implemented in the coming semester. Although they are not scored numerically, the forthcoming section will outline the reasoning behind the chosen concepts. Figure 10.1 represents our selection process from generated to final concepts.

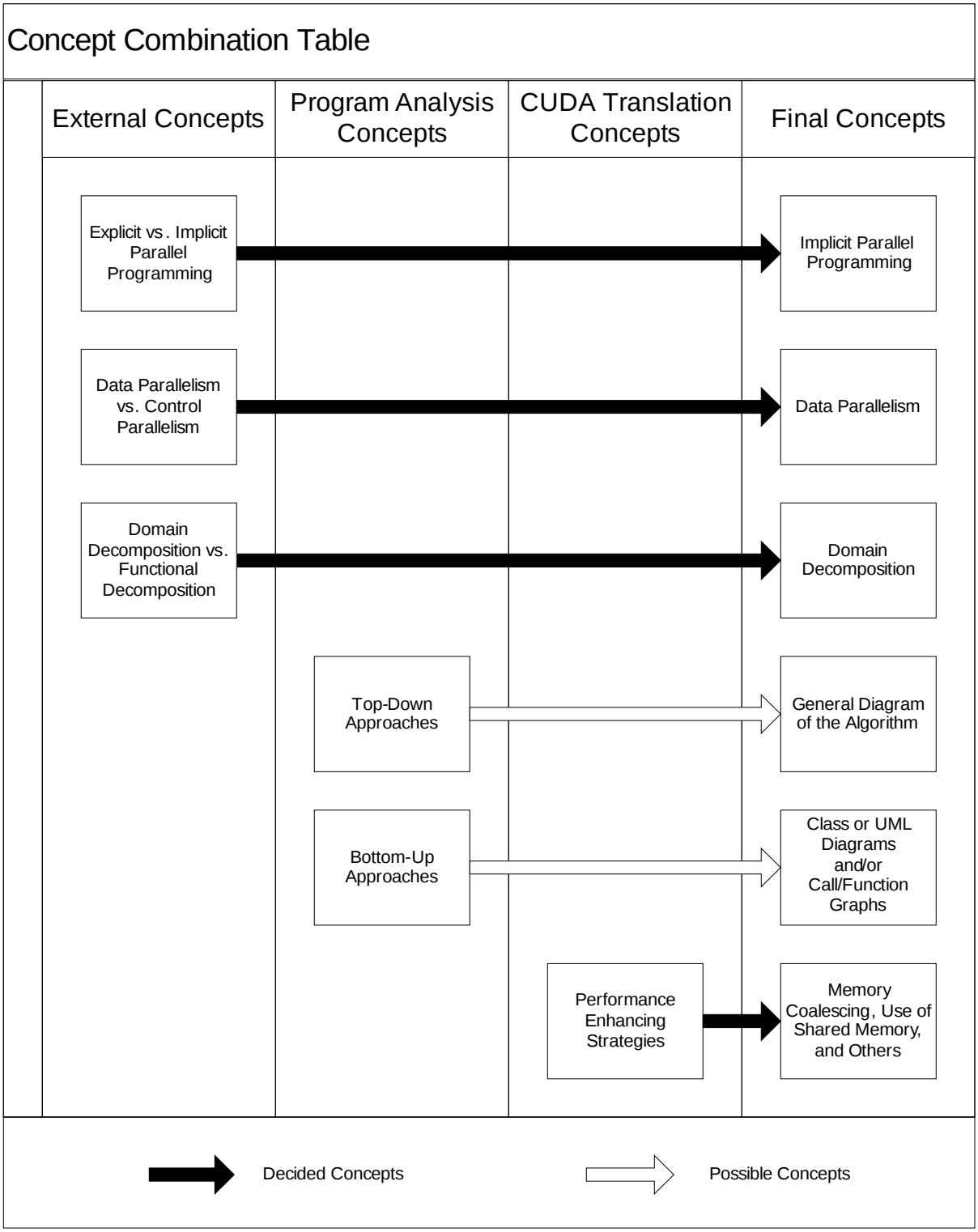


Figure 10.1 Concept combination table showing final concepts selected as well as possible concepts.

Implicit Parallel Programming

Because we have chosen to run our code on NVIDIA GPUs that are CUDA capable, which uses implicit parallel programming when compiling code, our final selection will also use implicit parallel programming. Explicit parallel programming does not follow in the constraints of the hardware that is defined in our project.

Data Parallelism

Data parallelism is ideal because CUDA forces programmers to mark off sections of sequential code that will be run in parallel. Due to the nature of CUDA's API, we are bound to using data parallelism in our implementation.

Domain Decomposition

We have concluded that using functional decomposition could cause problems in parallel computing. These problems might include conflicts in the AutoDock algorithm. By using domain decomposition, we are able to send independent portions of the docking procedure to be run in parallel on the hardware. Running independent portions is more ideal rather than portions that might be dependent on each other.

Program Analysis

In program analysis we have two approaches: bottom-up and top-down approach. The bottom-up approach consists of case diagrams of the code showing the interactions between the classes in the code. The top-down approach focuses on general diagrams of the algorithm. As we further analyze these approaches, we will discover the various locations where parallelism is ideal.

CUDA Translation

CUDA is the API of choice for NVIDIA GPUs. The group has studied CUDA and its interaction with the hardware and has discovered optimizations that can be implemented. One optimization that can greatly increase performance in memory interaction on GPUs is memory coalescing. As we continue to work with CUDA and learning its complex system, we will further discover more optimizations that we can use.

XI. Appendix

System Specifications for the gpgpu.mscs.mu.edu Machine

Processor	Dell Precision T3400 Convertible MiniTower Processor E4500, 2.20GHz, 800 2MB L2, 525W
Video Card	Dual nVidia Quadro FX3700 512MB dual DVI
Ram	2GB, 667MHz, DDR2 NECC SDRAM Memory, 2x1GB
Hard Drive	80GB SATA 3.0Gb/s with NCQ and 8MB DataBurst Cache

System Specifications for MicroWulf

Processor	Four AMD Phenom 64 9500 Quad-Core 2.2 GHz
Video Card	Four MSI NX8400GS nVidia GeForce
Ram	Four 1GB, 800MHz, DDR2 x2
Hard Drive	500GB Western Digital 7200 RPM

XII. References

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

[2] Blaise Barney., “Introduction to Parallel Computing.” 2007 https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

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

[4] Halfhill, Tom R. “Parallel Processing with CUDA, Nvidia's High-Performance Computing Platform Uses Massive Multithreading.” 1 January 2008. NVIDIA. 14 October 2008. <http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf>

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

[6] “Random seed.” Wikipedia.org. 14 October 2008. <http://en.wikipedia.org/wiki/Random_seed>

[7] Kumar, Vipin, Ananth Grama, Anshul Gupta, and George Karypis. Introduction to Parallel Computing: Design and Analysis of Algorithms. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc. 1994.

[8] Chandy, K. Mani and Stephen Taylor. An Introduction to Parallel Programming. Boston: Jones and Bartlett Publishers, Inc. 1992.

[9] Struble, Craig Ph.D. Personal interview. 31 Oct**.** 2008**.**