GPU Performance Increases for Autodock4 ECE 569 Spring 2017

Josiah Sellstrom, David Hung, Chien Hsun Ho, Zheng Li, Kejia Li

Electrical and Computer Engineering Department, The University of Arizona
1230 E. Speedway Blvd.
P.O. Box 210104
Tucson, AZ 85721-0104

josiahsellstrom@email.arizona.edu
chienhsunho@email.arizona.edu
davidhung@email.arizona.edu
zhengli233@email.arizona.edu
jacob1230@email.arizona.edu

Abstract— **Protein docking simulation software is useful for diminishing sample sizes in lab testing for molecule to protein bonding strengths. Autodock4 is a protein docking simulation software which has been partially adapted for GPU CUDA implementation to take advantage of the parallel nature of the problem. Within the CUDA based adaptations performance increasing tactics were looked at and attempted.**

Keywords— **CUDA, GPU, Autodock4**

I. Introduction

Docking simulations allow the user to narrow the focus of laboratory tests for protein to molecule docking trials. This narrower focus helps users save time. This approach does very little to effectively save researchers time, however, if the simulation software is too slow for the benefits to become apparent. Many current protein docking simulation softwares are open source, however, they are structured for CPU based implementation which is time consuming for large sets of input data. In order to decrease the overhead for these sets it is necessary to port Autodock4 to a GPU based implementation. One such project already exists which has ported two of the Autodock4 functions into CUDA, however, the performance is still relatively low. This project seeks to increase the performance of these ported functions.

Autdock4 is a protein to protein or protein to small molecule docking simulation software. This software analyzes the resultant energy state of the two objects being tested for the lowest energy state (the strongest bond) using a Lamarckian genetic algorithm coupled with two search functions, one large, the other small. The large search function takes the whole problem space and finds areas which are more likely to have beneficial matches, the small search function finds the areas within these selected mappings to focus in even further. The genetic algorithm then analyzes the energy states of the molecule based on pseudo random orientations of the molecule within the focused mapping.

In order to increase performance for the CUDA portions of the GPU adapted Autodock4 it was necessary to understand what these function were responsible for in the larger context of the program and then perform changes and see how those changes affected performance. During this project it was discovered that the primary area which could be improved was the memory allocation systems which did not utilize constant and shared memory within the CUDA ported functions.

II. Previous Project Analyses

A. Concept and Solutions

Many things need to be considered before porting Autodock and related medical domains into CUDA. The development concept for the CUDA porting project delineated between external and internal aspects of the code. The external part describes the consideration, analysis and decisions based on the background information such as the Genetic Algorithm. This project analyzed requirements on data parallelism and control parallelism, domain and functional decomposition, explicit and implicit Parallel Programing and the corresponding results. The internal part, however, is more about CUDA implementation strategy based on external analysis and design, two essential problems have been analyzed here. 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. Figure 1 shows the finalized concept selection of the porting project [1].

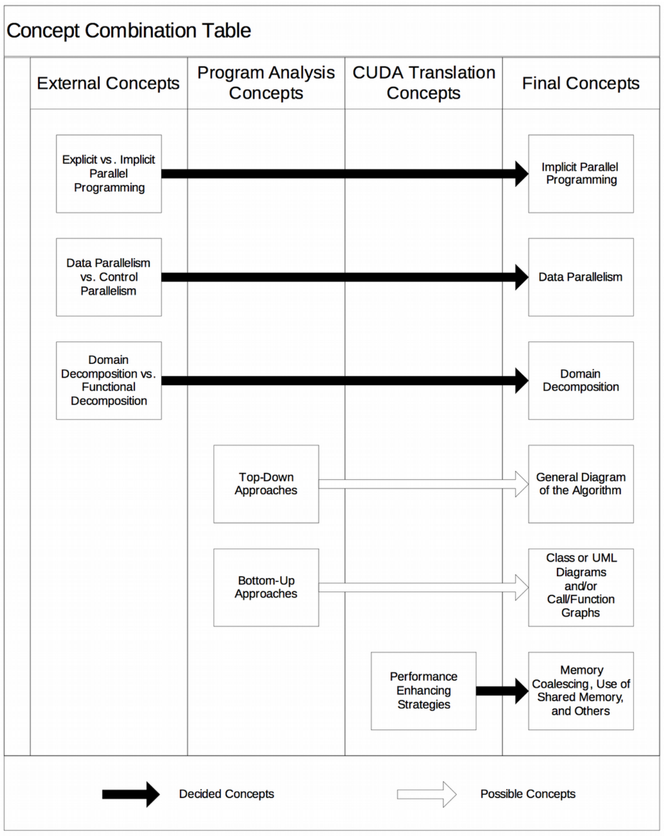
Concept Combination Table
External Concepts
Program Analysis Concepts
CUDA Translation Concepts
Final Concepts
Explicit vs. Implicit Parallel Programming
Implicit Parallel Programming
Data Parallelism vs. Control Parallelism
Data Parallelism
Domain Decomposition vs. Functional Decomposition
Domain Decomposition
Top-Down Approaches
General Diagram of the Algorithm
Bottom-Up Approaches
Class or UML Diagrams and/or Call/Function Graphs
Performance Enhancing Strategies
Memory Coalescing, Use of Shared Memory, and Others
Decided Concepts
Possible Concepts

Fig. 1 Concept of Targeted Project [1]

Specifically, for the CUDA translation, the Memory Coalescing, Shared Memory and the use of Barriers are considered and analyzed. In the real implementation, depending on the nature and requirement of the original algorithm, some of them are applied properly and some are not. Detailed analysis are mentioned in following sections and lead to changes in the right direction for optimized code performance where possible.

B. *Code Structure and operation*

Scrutinizing the Cuda ported code showed two major kernels responsible for energy analyses calculations. The first kernel utilizes a trilinear interpolation method, which means it operates energy calculations along each axis on a three dimensional plot. This kernel records the interactive energy produced by combining two components of the atoms selected. Another kernel, eintcal, defines internal energy calculations for each protein/molecule ligand interaction. Most of the input information is brought from the CPU structured code, which is in C++, and adapted to C for use in the CUDA code. This porting from C++ to C means that objects, which are not supported within C, needed to be restructured as float arrays for use. The threads assigned in global memory are in serial form. One potential optimization strategy was to use strided indexing for coalesced memory access.

C. *Potential Optimizations*

The CUDA ported structures only utilized global memory, this included the read only input array crdsgpu which was only allocated to the global memory and is used frequently in energy calculations. A constant or shared memory implementation of this array would allow for somewhat significant speed-up due to its iterative re-use within the CUDA ported kernels. Moving this array to constant memory was first attempted. However, the array exceeded the constant memory size in K20 of 65536 bytes so other options needed to be pursued. It was discovered the array size was too large for constant memory by printing out the array length.

The utilization of some variable arrays in shared memory was not possible due to their size. Other arrays were too small and too infrequently used to show any real benefits. As a result, other consideration of potential optimizations such as bank conflict and divergence control are not needed anymore.

III. Preliminary Testing Strategies

A. *Autodock4/raccoon*

In order to test any changes made to the CUDA ported functions within Autodock4 it was necessary to run the original code and run any modified code as well. Several strategies were tried in order to do this.

The first strategy was to run Autodock4 through the Raccoon and Raccoon2 GUIs. The original GUI, Raccoon, was shown to have compatibility issues when interfacing with Autodock4 (the original, non-GPU ported version). Autodock2 did not have these compatibility issues but proved too difficult to work with when interfacing with even the non-ported software because it mandated a server connection. This server connection could not be made with the elgato server because hpc requires a two part authentication and Raccoon2 only allows for one password entry field. There was no method for establishing a connection without the second part of the authentication process, therefore it was decided not to use Raccoon2 to run the program.

The next attempt to run Autodock with the GPU ported functions was to download the entire program onto elgato and attempt to make and configure it within that environment to generate the correct executables. This method proved to work, however, it became difficult to run the program within any reasonable amount of time due to outages with elgato and the demanding nature of the program, therefore, a third method was needed.

The third method used for running the GPU ported Autodock code was to run it on a CUDA compatible GPU. During this time it was also seen as beneficial to attempt to separate out the specific functions being targeted in order to better analyze the performance increases made by the team. Autodock4 functions were written in C++ and CUDA is modified C which means that the team who ported Autodock to CUDA had to convert object oriented programs to float array variables within the CUDA functions. Most of the functions within autodock also take advantage of the outputs of other functions, which means that it is extremely difficult to run any function, and especially the CUDA ported functions separately from the other functions.

Unfortunately, the test inputs provided by the original implementers were very large, resulting in execution times of an hour or more. This, in conjunction with the executions being largely CPU bound, made testing a difficult and time consuming process. In order to mitigate this, efforts were made towards isolating and extracting the kernel execution(s). In order to achieve this, the original code was modified to log and save each kernel launch, including the data passed to the kernel for processing. Then, the kernel code itself was extracted and placed into a simple test harness. The test harness fed the data from a single kernel launch to the extracted kernel code and output results and timing information. As a result, testing times were significantly reduced and optimization experiments could be carried out much more easily.

IV. Optimizations

A. *eval_tril_kernel*

Team Member	Optimization Approach
Chien Hsun Ho	Constant memory
Zheng Li	Register/Local memory
Kejia Li	Shared memory

Table 1 Optimization approaches used on eval_tril_kernel

i. Constant memory

To reduce the time for global memory reads, constant memory was used as the container for selected read-only arrays. For this the trilinear evaluation kernel, chargesgpu and ABSchargesgpu were the arrays which were small enough to be put in constant memory. However, these modifications did not reduce the operating time. The Crdsgpu array was the only array which made a considerable timing difference during overall execution. Looking at the histogram below, we could see the total execution time was reduced from 2600 seconds to 2380 seconds. Furthermore, the profiler was able to show a reduction in memory copy time from 55us to 26us.

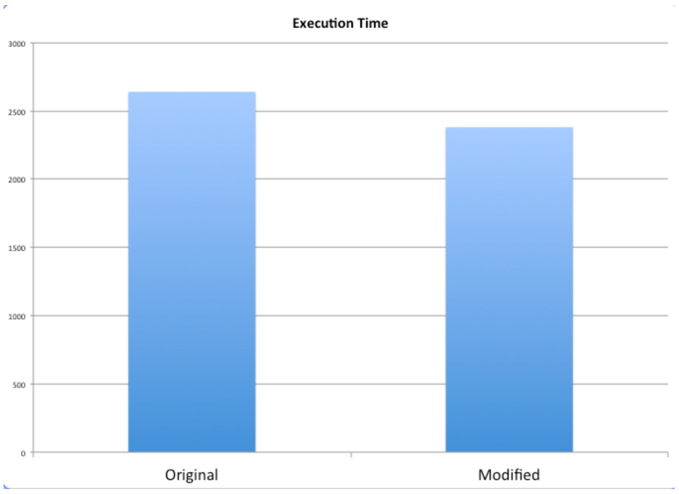

Fig. 8 Execution time comparison (seconds)

	Average CudaMemcpy
Original	55.03us
Modified	26.22us

ii. Register/Local memory

The eval_tril_kernel reads several arrays from global memory. However, reading from the global memory is quite expensive in terms of time. Therefore, it is neccesary to reduce global memory reading time as much as possible. When elements in an array are used in several calculations it is beneficial to read the data from the global memory once and write it to a register. THis way, the next time the data is needed, the kernel can read it from the register which is much faster than a global memory access.

The arrays pfloat_arraygpu and crdsgpu which are needed in the eval_tril_kernel, were stored in the global memory. Most elements from them are reused in an "for" loop. To reduce global read time new variables were created to store these arrays in registers.

iii. Shared memory

Although use of shared memory is not practical according to above-mentioned analysis, there are still some areas where it was worth experimenting. The below code snippet appears in the trilinear evaluation kernel eval_tri_kernel.

```
x -= (float)pfloat_arraygpu[FLOATINFO *
num_individualsgpu + 0];
y -= (float)pfloat_arraygpu[FLOATINFO *
num_individualsgpu + 1];
z -= (float)pfloat_arraygpu[FLOATINFO *
num_individualsgpu + 2];
```

The peak performance of different memory types in CUDA is ranked as follows:

Register > Shared > Constant > Texture >> Global [5]

In practice, when warps read the same location, constant memory, which is cached, can be exploited for optimization. The only condition for using constant memory is that this storage method does not allow for writes. To optimize the use of shared memory, things like bank conflict and divergence control need to be considered especially for tile-based strategies. Shared memory also has no advantage in memory size when compared to shared. Therefore, for the above data assignment shared memory might not be a better choice than constant memory.

Without considering the if condition which makes no difference in performance comparisons, the original process for reading from the pfloat_arraygpu array reads n times from global memory where:

n = num_individualsgpu * natomsgpu

Num_individualsgpu here defines how many threads will participate in this operation and natomsgpu defines the number of iterations. When applied to constant memory, there is no change on the number of read operations since the same memory space will be copied from CPU to constant memory instead of global memory. As a contrast, in the shared memory scenario, n' times reads are needed where:

n' = nBlocks + num_individualsgpu * natomsgpu

This means that additional read operations from global to shared memory for each block are needed but any reads after this number has completed will be from shared memory. An additional barrier is also needed to ensure data integrity.

With a similar read speed for constant memory and shared memory, it is obvious that constant memory would be the best choice based on above analysis. However, shared memory usage still yields better performance than the original global memory approach--especially when the input leads to a large product of num_individualsgpu and natomsgpu.

B. *eintcal_kernel*

Team Member	Optimization Approach
David Hung	Constant & shared memory
Josiah Sellstrom	Additional constant and shared memory porting

Table 2 Optimization approaches used on eintcal_kernel

i. Constant and Shared Memory

The kernel relies heavily on global memory reads to perform its computations. In order to reduce the execution time spent on global memory accesses, a shared memory implementation was experimented with. In particular, the kernel was modified such that ***nonbondlist*** was first loaded into shared memory. By doing this, the access pattern for ***nonbondlist*** could be changed from its original strided access pattern to a coalesced access pattern.

It was additionally noted that the vast majority of global memory accesses are read only. This indicates that there could be performance gains in a constant memory implementation. Three arrays were targeted: nnb_array, ***nonbondlist***, and ***strsol_fn***. ***nnb_array*** is a very small array which is accessed at the same index by all threads in a warp. ***nonbondlist*** similarly has an identical access pattern across warps, but over a larger index range. strsol_fn is a precomputed function lookup table, and thus has a more random access pattern. Two constant memory implementations were tested: one, denoted const_1, placed all three arrays in constant memory, while another, denoted const_2, placed only ***nnb_array*** and ***nonbondlist*** in constant memory.

Kernel	Execution Time (ms)	Improvement
original	6.509	0%
shared	6.644	-2%
const_1	7.573	-16%
const_2	**5.804**	**11%**

Table3 Memory optimization results on eintcal_kernel. Execution times are averaged over 100 runs. const_1 denotes the constant memory implementation with nnb_array, nonbondlist, and strsol_fn all placed in constant memory, while const_2 denotes only nnb_array and nonbondlist in constant memory.

As seen in the results above, the shared memory implementation actually resulted in decreased performance as compared to the original kernel. It is hypothesized that the overhead of allocating and loading in the shared memory array outweighed any performance gains. Additionally, the const_1 implementation also displays (significantly) inferior performance. This is likely due to the random constant memory access pattern to the strsol_fn array, as memory accesses within half warps to different addresses within constant memory are serialized and thus have linearly time complexity. On the other hand, const_2 shows some performance gains. As the memory access pattern for both ***nnb_array*** and ***nonbondlist*** are identical within each half warp, each half warp is able to perform all of its constant memory reads in one cache hit.

V. CONCLUSIONS

Unfortunately, significant performance gains were not able to be achieved through optimizing the CUDA ported kernels alone. Both kernels are heavily reliant on global memory reads, but each read element is only used once with no opportunities for collaboration between threads. As a result, approaches such as tiling and shared memory are ineffective. Additionally, while moving data to constant memory does show slight improvements, performance gains are still small, and further gains are unlikely due to device limitations on constant memory size. Overall, Autodock is still a highly CPU bound application with little room for improvement in the existing kernels. Greater performance gains may be possible, but would require the conversion of more parallelizable CPU code to GPU code.

REFERENCES

[1] S. Kannan and R. Ganji, "Porting Autodock to CUDA," *IEEE Congress on Evolutionary Computation, Barcelona,* 2010, pp. 1-8.

[2] Stefano Forli, Ruth Huey, Michael E Pique, Michel F

Sanner, David S Goodsell & Arthur J Olson, “Computational protein–ligand docking and virtual drug screening with the AutoDock suite” , 1st May 2016

[3] Timothy Blattner David Hartman Andrew Kiel Adam Mallen Andrew St. Jean,”AutoDock Software In Parallel With GPUs” , 2009

[4] Forli, S. Raccoon for Processing Virtual Screening <http://autodock.scripps.edu/resources/raccoon> , 2013

[5] Justin McKennon, “GPU Memory Types – Performance Comparison”, *Parallel Code: Maximizing your Performance Potential*. 2013.