GPU DOCK

Saad Amin

Abstract

Molecular docking is a crucial tool in drug discovery. However, it happens to be very slow. One observation about docking is that all of its subprocesses (e.g. processing one particular orientation) are independent from one another. These processes are also high in arithmetic complexity and low in branching complexity, making them a perfect match for GPUs. In this short write-up, I go over efforts to port DOCK to the GPU and their results. Based on these results, I propose ways to move forward with GPU DOCK.

Introduction

In this section, I give the necessary background for molecular docking. I then discuss the components of DOCK and how they work.

An Overview of Medicinal Chemistry

Proteins and enzymes are molecules that provide crucial functions for sustaining life. Proteins, for instance, are useful in producing a variety of structures for organisms. Sometimes, we may want to adjust the structure or function of such molecules to achieve a certain task. For example, one protein is essential in helping bacteria build cell walls. If we can disable this protein, we can help avoid bacterial infections and help a sick person recover from a cold. Molecules such as proteins and enzymes are known as receptors.

Changing the structure or function of receptors is the task of drugs. Drugs are molecules that bind to proteins and enzymes to incur a change. In the case of bacteria, we might want to bind a drug to the protein that makes it unable to produce cell walls.

Finding these drugs is nontrivial and mainly a process of trial and error. Even if a drug might work, it can have unintended consequences on the rest of the human body. Initially, many drugs are tested out experimentally in petri dishes full of sample tissue. The drugs that show promising results are further tested, and may advance to clinical trials before coming into production.

Experimentally testing drugs in petri dishes is a slow and expensive process. To speed up this process, we can test out many molecules in a simulation to filter out non-candidate molecules. The goal of docking computer programs is to run these simulations on large datasets of molecules.

Generally, any molecule that binds to a receptor is known as a ligand. Drugs are a special class of ligands that have a medicinal purpose. However, docking programs run with any receptor and ligand combination for any purpose, and hence we will use these generalized terms for the rest of this paper.

Outline of Docking

The first step in docking is to find orientations in which ligands and receptors might bind. Docking algorithms assume that both receptors and ligands have atoms known as hotspots. Hotspots on the ligand have the capability to create bonds with hotspots on the receptor.

Once these potential orientations are found, the docking program scores how “good” the orientation is. Goodness of the orientation depends on multiple factors, including whether the bonds are strong or weak, or whether there is a spatial collision between the ligand and receptor.

Docking programs want to find the best orientation for each ligand, and the matching algorithm is not sufficient for this. Instead, the program takes a set of promising matches and runs an optimizer on the orientations. The program represents the orientation as a 6D input consisting of 3 values for rotation and 3 values for transformation, and then runs a simplex optimizer on it.

Now, I will go into more detail on each phase of the docking program.

Matching

The first step in the docking algorithm is to generate matches. This is done with distance matrices [1] and often takes up the bulk of the time of the program. What roughly happens (from my understanding) is that distance matrices of both the receptor and the ligand are generated. Then the ligand distances that are within a certain tolerance level are paired up with the receptor distances. This array of pairs is used to generate potential matches, which are then converted into rotational and translational transformations for the ligand.

Scoring

Once a match has been generated, it is necessary to score it. For each match, the program considers a set of structural deformations of the ligand and receptor. Let’s now discuss some terminology for DOCK

- A conformation of a ligand is the ligand with a particular set of bond rotations (i.e. structural deformations)
- A set is a way to mathematically represent the conformation. Each set contains an array of confs

- A conf (not to be confused with conformation) is a collection of spatially local atoms. These collections of atoms are rigid (i.e. a conf cannot structurally deform) and connect to other confs via rotatable bonds

Now that we know DOCK terminology, let's take a look at the scoring loop (in this code, ligscore is a data structure that contains scores):

```
function score mol(ligscore):
     for each match
          ligscore.insert(score sets(current match))
     return ligscore


function score sets(match):
     for each conf
          for each receptor
               conf record(receptor, conf) = score conf(match)

     for each set
          for each receptor
               for each conf in set
                    conf score = conf record(receptor, conf)
                    set score(receptor, set) += conf score

     for each combination
          for each set/receptor pair in the combination
               whole score(combination) += set score(receptor, set)

     return whole score
```

To represent multiple conformations, DOCK first picks a conf known as the "rigid conf", which is an "anchor" for transformations. It is not translated. Then, DOCK starts rotating the bonds of confs connected to rigid conf, and then recursively rotates the subbonds of those confs to generate multiple confs (note that bond rotations are sampled at discrete angles). Hence, in the process it explores the tree of all possible bond rotations and saves them to memory.

When the program rotates a conf, it needs a way to remember the translated positions of those atoms. DOCK takes a crude approach to this and duplicates that conf. Hence, the same rigid portion of atoms might be represented thousands of times in memory.

Combinations

One thing to note in the molecule scoring loop is about combinations. When a ligand docks with a protein, it undergoes a variety of states in which both the protein and ligand might deform over time. To account for this, DOCK discretizes the time interval of the docking process. It then samples and scores a variety of set and receptor deformations at each time step, and then accumulates their scores in a final pass.

The Scoring Process

By default, only 3 scoring options are enabled:

- Van der waals score
- Electrostatics
- Solvation score

Each of these scores involve a tedious process to accurately compute. As such, DOCK relies on an approximation. It represents the area around the ligand using a grid and then samples the score at each grid point. When the time to score comes, it samples this grid using linear interpolation.

DOCK also has an option to enable GIST scoring. GIST scoring is an accurate way to represent the cost of displacing the water molecules during docking. However, it requires a lot of memory to compute and is not fast. Furthermore, as we will discuss, it does not adapt well to GPUs.

Minimization

After all the scores have been computed from the above processes, DOCK then runs a minimization phase. It optimizes the ligand transformation matrix to achieve the best score. The optimizer in DOCK uses a stochastic simplex algorithm.

Current Status of GPU DOCK

In this section, I will go over efforts that have been put into moving DOCK to the GPU.

Overview

GPU DOCK processes ligands in batches. The CPU side program reads a set of ligands from the disk and matches them. Once the ligands are done being matched, the program uploads all of the matches to the GPU. The GPU scores these ligands and outputs them to a buffer. Once the scoring is done, the CPU downloads these scores and saves them into a buffer for processing.

Scoring

GPU DOCK only takes into account Van der Waals, electrostatics, and solvation. It does not take into account GIST. GIST requires a very large array to store a list of voxels, and thus will not port to the GPU well. A suggested alternative for GIST scoring includes a Guassian blurring method.

Memory

Each ligand has a large number of matches, and each match has a large number of sets. Since the GPU scores all sets at once, there is a large movement of memory from the GPU to the CPU. This often is in the hundreds of megabytes even for ligand batch sizes of 16.

To alleviate this problem, we prefiltered the matches. Before allocating space on the GPU to store all sets for all matches, we first checked what matches are bumped on the CPU by checking the score of the rigid conf. This allows us to allocate GPU memory accordingly to the matches we actually need to score. This also has the side benefit of skipping calculating the scores for bumped matches. Prefiltering is quick for many ligands (approximately less than 1 ms even for large batch sizes) and achieves a speedup from 8 seconds to 2 seconds for 16 ligands.

To keep the Fortran and C++ sides as independent from each other as possible, we implemented an interface for reading back the scores. The internal C++ implementation has two available methods for keeping track of the scores:

- A master score record, which organizes everything into a neat data structure. It is a 3D array that takes in a (ligand, match, set) index. The memory for this structure is allocated from a memory pool that keeps track of 4 MB buckets.
- Directly reading into the downloaded buffer. This is often more efficient and allocating the buffer is a required step in downloading anyway. The old version is used mainly to debug the program.

Results

In this section, we calculate $\%\, speedup = 100(\frac{t_{before}}{t_{after}} - 1)$.

We tested GPU DOCK and CPU DOCK on H17P050-N-laa. We found that GPU DOCK took 549.3760 seconds whereas CPU DOCK took 668.8986 seconds. This leads to approximately a 21% speedup.

Future Plans

GPU Matching

A further analysis of our results showed that GPU DOCK spent only 56 seconds uploading, scoring, and downloading. Hence, the remaining 500 seconds was spent generating matches and saving the scores. We believe the greatest gains can be made by porting the matching algorithm to the GPU.

Optimizing Scoring

Currently, the scoring loop does not take advantage of all features that GPUs have to offer. For example, grid sampling and interpolation is done manually. This can efficiently be done on texture memory, where the GPU can take advantage of texture caching and hardware-accelerated interpolation.

Another current bottleneck is the issue of storing the saved scores. The Fortran/C++ interface induces many bottlenecks and unnecessary memory duplications, and scores are a major example of this. If we can get the score saving and inserting working on the C++/GPU side, we can achieve major speed gains.

Minimization

We also need to implement minimization on the GPU. This can easily be done by assigning each thread its own match and implementing the optimization routines.

Complete GPU Pipeline

The grand vision for DOCK is to have all, if not most of DOCK running on the GPU asynchronously. We can have multiple sub-kernels running on the GPU that aim to move ligands along the scoring process.

We envision that we first upload a batch of ligands to the GPU and call the “full-docking” kernel. Once this kernel is launched, a set of blocks reads these ligands from memory and starts generating matches. These matches are then put in a buffer in global memory, where awaiting blocks can wake and pull these matches into local/shared memory for scoring and minimization. Once these threads are finished, they output to a data structure which keeps track of scores. If this data structure is nearing capacity, we can send a signal to the GPU to download it. The scoring threads then look for more matches to score. If the match buffer is nearing being empty and there are not many ligands left we send a flag to the CPU to upload more ligands.

Missing Features

There are some features in CPU DOCK which are not implemented in GPU DOCK. Namely:

- GIST-based scoring. This can be implemented using a Gaussian blurring method. Perhaps we can devise a hybrid method that does not rely too much on memory and more on arithmetic complexity and flow simplicity.
- Support for multiple receptors and combinations. This would significantly increase memory bandwidth and pressure on older hardware, however this is trivial on newer hardware.

We can also extend GPU DOCK beyond the original features. Some GPU docking programs use genetic algorithms to do docking. Others use a hybrid approach where the GPU does some portion of work while the CPU does some other portion of work.

Future of the Project as a Whole

My goal is to have a meaningful enough impact on DOCK that GPU DOCK becomes the next major iteration of DOCK 3.8. I also plan to get this published by October 2023. Since there are a lot of features to implement, I suggest expanding the resources for this project. I request 2-3 additional programmers working on GPU DOCK.

References

[1] R. G. Coleman, M. Carchia, T. Sterling, J. J. Irwin, and B. K. Shoichet, "Ligand pose and orientational sampling in molecular docking," *PLoS ONE*, vol. 8, no. 10, 2013.