Published in final edited form as:

Methods Mol Biol. 2017 ; 1529: 265–277. doi:10.1007/978-1-4939-6637-0_13.

Parallel Computational Protein Design

Yichao Zhou¹, Bruce R. Donald^{2,3}, and Jianyang Zeng^{1,*}

¹Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, P. R. China

²Department of Computer Science, Duke University, Durham, North Carolina, United States of America

³Department of Biochemistry, Duke University Medical Center, Durham, North Carolina, United States of America

Abstract

Computational structure-based protein design (CSPD) is an important problem in computational biology, which aims to design or improve a prescribed protein function based on a protein structure template. It provides a practical tool for real-world protein engineering applications. A popular CSPD method that guarantees to find the global minimum energy solution (GMEC) is to combine both dead-end elimination (DEE) and A* tree search algorithms. However, in this framework, the A* search algorithm can run in exponential time in the worst case, which may become the computation bottleneck of large-scale computational protein design process. To address this issue, we extend and add a new module to the OSPREY program that was previously developed in the Donald lab [1] to implement a GPU-based massively parallel A* algorithm for improving protein design pipeline. By exploiting the modern GPU computational framework and optimizing the computation of the heuristic function for A* search, our new program, called *gOSPREY*, can provide up to four orders of magnitude speedups in large protein design cases with a small memory overhead comparing to the traditional A* search algorithm implementation, while still guaranteeing the optimality. In addition, *gOSPREY* can be configured to run in a bounded-memory mode to tackle the problems in which the conformation space is too large and the global optimal solution cannot be computed previously. Furthermore, the GPU-based A* algorithm implemented in the *gOSPREY* program can be combined with the state-of-the-art rotamer pruning algorithms such as iMinDEE [2] and DEEPer [3] to also consider continuous backbone and side-chain flexibility.

Keywords

protein design; A*; dead-end-elimination; GPGPU; CUDA; parallel computing

²To whom correspondence should be addressed. zengjy321@tsinghua.edu.cn.

1 Introduction

1.1 Structure-Based Computational Protein Design

Computational structure-based protein design (CSPD) is an important task in computational biology. In this problem, we want to find new amino acid sequences that have the prerequisite features to perform certain desired functions by substituting a number of residues from a wild-type protein structure with new amino acids. CSPD has many exciting real-world applications in protein engineering, such as design of protein-protein interactions [4], drug design [5], drug resistance prediction [6,7], vaccine development [8,9], and enzyme synthesis [10].

The CSPD problem can be formulated into finding a sequence of side-chain conformations based on a given energy function so that the energy of the resulting protein structure is minimized. Such an optimal solution is often called the *global minimum energy conformation* (GMEC). In an ideal case, we hope to consider all possible backbone positions and continuous side-chain conformations for searching the GMEC solution. However, it is almost impossible to sample over all these parameters with high precision because of the huge computational burden. Therefore, simplified protein design models with reasonable assumptions are often used. In practice, we often ignore the displacement of the backbone structure to assume a rigid backbone, and limit the rotational degrees of freedom of side-chain conformations to a set of common discrete conformations, called *rotamer library*.

Having the rigid backbone structure and discrete side-chain conformation assumptions, the protein design problem can be formulated into a combinatorial optimization problem. Equation (1.1) defines the objective function of this problem:

$$E_T(A) = E_0 + \sum_{i_r \in A} E_1(i_r) + \frac{1}{2} \sum_{i_r \in A} \sum_{j_s \in A} E_2(i_r, j_s), \quad (1.1)$$

where A represents a conformation in the search space, i.e., a set of discrete side-chain rotamers of all residues, $E_T(A)$ represents the total energy of conformation A , E_0 represents the backbone energy, $E_1(i_r)$ represents the self-energy term of residue i_r which is the sum of its intra-energy and residue-to-backbone energy, and $E_2(i_r, j_s)$ represents the pairwise energy between rotamer i_r and j_s .

Unfortunately, even under the rigid backbone and discrete side-chain conformation assumptions, finding the GMEC solution has still been proven as an NP-hard problem [11,12], which means that most likely there does not exist an algorithm that can guarantee to solve it in polynomial time. The solutions to this issue can be divided into two categories. One common scheme is to apply heuristic algorithms in hopes of generating high-quality solutions [13–16]. The weakness of this scheme is that these algorithms often provide no guarantee of solution quality, as they may be trapped into local optima.

The alternatives to the heuristic algorithms are the provable algorithms, which can assure to output the GMEC solution. Examples are integer linear programming [17], branch-and-

bound [18,19], tree decomposition [20], dead-end elimination [21,2], and A* tree search [1,22,23]. Among them, the combination of dead-end elimination and A* tree search has been popularly used to solve the design problem [24]. The major advantage of this pipeline is that it not only guarantees to find the global minimum energy conformation solution, but also is to output all the sub-optimal solutions in a gap-free sorted order in an efficient way. This is an important feature because sub-optimal solutions are necessary to fight against the errors in the energy functions and from the model assumptions (e.g., rigid backbone structure) for real-world applications.

In our protein design pipeline, a set of predefined dead-end elimination criteria [25,26] is first applied to prune all the rotamers that can be proved not to be in the GMEC solution. After that, A* tree search algorithm is applied to traverse the remaining conformation space to find the GMEC solution (and other sub-optimal solutions within a given energy cutoff from the GMEC solution). Although A* search guided by an admissible heuristic function usually only needs to visit a small portion of search space to find the optimal solution, it can still run in exponential time in the worst-case scenario due to the difficulty of this problem. Our tool `gOSPReY` provides an effective method to address this computational bottleneck by fully exploiting the massively parallel computational power on the GPU platform to accelerate the computational protein design process.

1.2 General-Purpose Computing on a Graphic Processing Unit

In recent years, the general-purpose computing on a graphic processing unit is becoming popular in numerous scientific computation scenarios. The main difference between a traditional CPU computational framework and a GPU computation framework lies in the way they deal with a computational task. CPUs are often optimized to efficiently execute the input instructions one by one with little parallelism, while GPUs are designed to process thousands of similar tasks simultaneously in an efficient fashion.

Because of such design difference, each core of a GPU is simpler and more efficient than that of a CPU. Therefore, the overall throughput of a GPU platform can be much higher than a CPU platform if the intrinsic parallelism has been fully exploited [27,28]. Furthermore, a GPU platform usually has its own memory system with high memory bandwidth that is independent of the normal memory system used by the CPU platform. Such a design scheme may require data to be transferred back and forth between CPU and GPU through a slow interface called the PCI-E bus. Our algorithm implemented in `gOSPReY` is a pure GPU algorithm, thus we only need to transfer the initial input data and the final output results between CPU and GPU, in which the data transferring time is generally a negligible overhead compared to the large amount of time spent in the floating-point arithmetic operations require by A* search.

2 Materials

2.1 Hardware

`gOSPReY` requires a NVIDIA's CUDA-capable GPU to enable its GPU acceleration feature. The minimum requirement of the CUDA compute-capability of the GPU is version 1.2.

2.2 Software

Currently gOSPREY depends on the following software environment:

- Linux operation system
- NVIDIA CUDA SDK
- Java Development Kit 1.7
- gcc-4.7 or newer
- CMake 2.8 or newer

2.3 Installation

In order to install gOSPREY, the user needs to enter the working directory and execute the following commands:

```
$ git clone  
  
https://github.com/zhoul3/gOSPREY.git  
  
$ cd gOSPREY  
$ mkdir build  
$ cd build  
$ cmake -DCMAKE_INSTALL_PREFIX=/usr ..  
$ make  
$ sudo make install
```

After that, if everything goes smoothly, the user should be able to find that `osprey.jar` has been generated under the `build` directory and the library `libMSAStar.so` has been installed to the system's library directory.

3 Methods

3.1 The Algorithm

In order to find the global minimum energy conformation (GMEC) solution, we often need to search over a large conformational space. To reduce the search space and speed up the design process, our search scheme follows a popular protein design pipeline in the literature [1]: First, a set of dead-end elimination criteria is applied to prune the rotamers that are provably not part of the optimal solution and thus can significantly reduce the magnitude of the search space. Then, a combinatorial optimization algorithm, namely A* tree search, is used to traverse the remaining conformational space and guarantees to find the GMEC solution. In the traditional A*-based tree search algorithm for protein design, an A* search tree is visited for searching the global optimal solution.

An example of the A* search tree is shown in Figure 1a. Each internal node of the tree represents a partial conformation in which the side-chain conformations of some residues

have not been decided yet, while each leaf node represents a full conformation, in which the side-chain rotamer conformations of all residues have been determined. Thus, computing the global minimum energy conformation is equivalent to finding the leaf node with the minimum energy function value in the last layer.

In order to traverse this search tree efficiently, the A* search algorithm with an admissible heuristic function is usually used [29]. The traditional A* search algorithm uses a priority queue to decide which node should be visited next. In the priority queue, nodes are sorted according to the following heuristic function:

$$f(x)=g(x)+h(x), \quad (1.2)$$

where $g(x)$ represents the energy term among the residues that have already been decided and $h(x)$ represents the lower bound of the energy term that involves the undecided residues. Intuitively, the heuristic function $f(x)$ provides a quantified estimation about whether the children of node x have low energy values. Therefore, A* search can give higher priority to these nodes with lower $f(x)$ values during the node expansion process. In our protein design problem, the functions of $g(x)$ and $h(x)$ are defined as follows:

$$g(x)=E_0+\sum_{i\in D(x)} E_1(i_s)+\frac{1}{2}\sum_{i\in D(x)}\sum_{j\in D(x)} E_2(i_r,j_s)$$

$$h(x)=\sum_{i\in U(x)} \min_s \left(E_1(i_s)+\sum_{j\in D(x)} E_2(i_r,j_s)+\sum_{k\in U(x)} \min_u E_2(i_s,k_u) \right), \quad (1.3)$$

where $D(x)$ represents the set of residues whose side-chain rotamer conformations have already been decided and $U(x)$ represents the set of residues whose side-chain rotamer conformations are still undecided.

In the A* search process, the root node is placed in the priority queue initially. In each round, the A* algorithm extracts the node with the minimum $f(x)$ value, expands its child nodes, computes their heuristic function values, and pushes them back to the priority queue. These steps are repeated until a leaf node is extracted. If the heuristic function is *admissible*, which is the case for Equation (1.3), we can prove that the A* search algorithm can find the global optimal solution in our protein design problem [29]. Figure 1b gives an example of the node expansion operation in the traditional A* search algorithm.

From Figure 1b, we know that the calculation of heuristic function $f(x)$ for each expanded node is simply a series of independent arithmetic operations, which can be directly parallelized on a GPU platform. However, the degree of the parallelism is still limited by the number of children for each node, which is equal to the number of rotamers for each residue in the protein design problem. In general, this number is far smaller than the number of cores in a normal GPU processor, thus the parallelization of the heuristic function calculation alone does not fully exploit the parallelism of a GPU.

To further speedup the search process, *gOSPReY* creates another level of parallelism to exploit the computational power of a GPU platform. Instead of using only one priority queue to perform node expansion, the parallel A* search algorithm allocates hundreds of priority queues in parallel to accelerate the A* search process. Figure 2 provides an example of the parallel node expansion operations in our algorithm.

Our GPU-based A* search algorithm first launches k threads to extract the nodes with the minimum $f(\cdot)$ values from k independent priority queues in parallel, where k is a parameter that can be set by the user. Then each thread expands the child nodes of each extracted node in parallel. After that, the GPU-based A* algorithm launches p threads to compute the heuristic function values for each expanded node, where p is the number of total expanded nodes. Finally, the algorithm launches k threads to push these expanded nodes with the computed $f(\cdot)$ values back to the k priority queues.

The parameter k can affect the performance of *gOSPReY*. If we increase the value of k , we can further exploit the parallelism of the GPU platform. However, it may also increase the number of nodes that need to be expanded before finding the global optimal solution, which may lead to an extra memory overhead. In Section 3.3, we will provide a simple method to choose a suitable k for individual GPU platforms.

3.2 Performance Evaluation

Here we cite the test results from [30] to show the performance of our GPU-based protein design algorithm in *gOSPReY*. In this test, we used the CPU Intel Xeon E5-1620 3.6 GHz with 16GB memory and the GPU Tesla K20c with 2496 CUDA cores and 4.8G global memory. We evaluated both running time and memory usage of *gOSPReY* on several native sequence recovery problems. Figure 3 shows the comparison results between the traditional single-thread CPU-based A* search algorithm and the massively parallel GPU-based A* search algorithm implemented in *gOSPReY*. For more details about the comparison results with other protein design frameworks, please refer to the original paper [30].

As shown in Figure 3, the GPU acceleration achieved by *gOSPReY* was remarkable. Our GPU-based A* search algorithm was about 40 times faster than the traditional single-thread A* search algorithm on the large design problems. In addition, our benchmark tests showed that the GPU-based A* search had good scalability. The larger the size of the problem, the more speedup that GPU-based A* algorithm can achieve. For small design problems, launching the GPU code and copying the input and output data caused considerable overhead and thus the power of parallelism in the GPU platform was not fully exploited.

The result about the memory overhead of our GPU-based protein design algorithm was also promising. On small problems, the GPU-based A* search algorithm with 4992 priority queues used about six times more memory than the single-thread A* implementation. However, when the magnitude of the problems scaled up, the memory overhead gradually diminished. In the largest problem, the massively parallel GPU-based A* search algorithm only expanded 0.12 times more nodes than the CPU-based algorithm. The significant improvement in running time and the negligible memory overhead shown in this test made

the GPU acceleration in `gOSPRED` an appealing and practical tool for solving the large protein design problems.

As shown in Figure 3b, the GPU-based A* search algorithm with 4992 priority queues expanded more nodes than the A* search algorithm with 768 priority queues. This meant that the number of parallel priority queues should not be set to be too large, as this may result in an unacceptable memory overhead. In the next section, we will describe how to choose the appropriate number of parallel priority queues based on the available hardware environment.

3.3 Usage Examples

As `gOSPRED` is built based on `OSPRED` originally developed from the Donald lab [1], most of the configuration of `gOSPRED` is as same as that of `OSPRED`. Therefore, in this section we will mainly focus on the feature of `gOSPRED`, i.e., how to use the GPU acceleration feature of `gOSPRED`. For the guide and tutorial about how to set up other parameters, please refer to the original manual of `OSPRED` which is stored in `doc/manual.pdf`.

1. The first thing that the user needs to do is to know the hardware specification of the GPU system. This can be queried by a simple program called `deviceQuery` in the NVIDIA CUDA Samples, which is usually installed with the CUDA SDK. The `deviceQuery` program can be found in `$PATH_TO_CUDA_SAMPLE/1_Uutilities/deviceQuery`. The user may need to compile it manually using the `make` command. Here is an example of the `deviceQuery` result:

```
$ ./deviceQuery
deviceQuery Starting...

   CUDA Device Query (Runtime API) version (CUDA static
linking)
Detected 1 CUDA Capable device(s)
Device 0: "Tesla K20m"

   CUDA Driver Version / Runtime Version      5.5 /
5.5
   CUDA Capability Major/Minor version number: 3.5
   Total amount of global
memory:                    5120 MBytes (5368512512
bytes)
   (13) Multiprocessors, (192) CUDA Cores/MP: 2496
CUDA Cores
   GPU Clock
rate:
706 MHz (0.71 GHz)
   Memory Clock
rate:                    2600
```

```

Mhz
    Memory Bus
Width:                                     320-
bit
    L2 Cache
Size:
    1310720 bytes
    .....
    Concurrent copy and kernel
execution:                               Yes with 2 copy engine(s)
    Run time limit on
kernels:                                 No
    Integrated GPU sharing Host Memory:
No
    Support host page-locked memory mapping:      Yes
    Alignment requirement for
Surfaces:                               Yes
    Device has ECC
support:
Disabled
    Device supports Unified Addressing (UVA):
Yes
    Device PCI Bus ID / PCI location
ID:                                     2 / 0
    Compute Mode:
        < Default (multiple host threads can
use ::cudaSetDevice() with device sim-
ultaneously)>
        deviceQuery, CUDA Driver = CUDART, CUDA Driver Version =
5.5, CUDA Runtime Ver-
sion = 5.5, NumDevs = 1, Device0 = Tesla K20m
        Result = PASS

```

Here we need to care about:

- a. “(13) Multiprocessors, (192) CUDA Cores/MP” tells us that this GPU has 13 multi-processors, and each of them has 192 CUDA cores. In total, we have $13 * 192 = 2496$ CUDA cores.
 - b. “Total amount of global memory” tells us the size of the global memory in the GPU.
2. The user also needs to configure the parameter files of `gOSPREY` in order to use the GPU acceleration. An `OSPREY` workspace usually contains three configuration files: `KStar.cfg`, `System.cfg`, and `DEE.cfg`. For GPU

acceleration, the user needs to modify `KStar.cfg`. Here is a list of options that are new to OSPREY and the user needs to append them to `KStar.cfg`:

```
enableAStarJava false
enableAStarNativeC false
enableAStarCUDA true
maxNativeCPUMemory 5032706048
maxNativeGPUMemory 5032706048
numGPUWorkGroup 26
numGPUWorkItem 192
numGPUWorkItem2 192
shrinkRatio 0.5
```

There are three different A* engines in the `gOSPREY` program: `enableAStarJava` indicates whether the original Java A* engine from OSPREY is enabled; `enableAStarNativeC` indicates whether our A* engine with the optimized computation of the heuristic function implemented in C programming language, which should be hundreds of times faster than the original Java A* engine, is enabled; `enableAStarCUDA` indicates whether our GPU-based A* engine is enabled, the performance benchmark of which is shown in Section 3.2.

In `KStar.cfg`, `maxNativeCPUMemory` and `maxNativeGPUMemory` set the maximum CPU and GPU memory that `gOSPREY` can occupy, respectively. For `maxNativeGPUMemory`, setting the value to 80% of the global memory size of the GPU indicated by the `deviceQuery` program is a safe choice.

The parameters `numGPUWorkGroup` and `numGPUWorkItem` together determine the number of parallel priority queues used in `gOSPREY`. In most case, it is reasonable to set `numGPUWorkItem` to be the number of CUDA cores per multi-processor. `numGPUWorkGroup` can be set to be one to two times the number of multiprocessors in the GPU platform. Parameter `numGPUWorkItem2` determines the block size of CUDA when computing the heuristic functions in parallel. In general, it can be set to be the same value as `numGPUWorkItem`.

Finally, `shrinkRatio` determines the fraction of frontier nodes kept in the memory-bounded A* search. After the memory occupied by the GPU-based A* search algorithm exceeds `maxNativeCPUMemory` or `maxNativeGPUMemory`, `gOSPREY` will discard a percentage of unpromising nodes in the priority queues depending on the value of `shrinkRatio`. Setting this parameter to be less than 1 enables this feature so that A* can continue to run even after the number of expanded nodes exceeds the global memory.

4 Note

1. Our GPU-based A* search algorithm can also output all the sub-optimal solutions within a given energy cutoff from the GMEC solution in a gap-free sorted order, using the same setting as in OSPREY [1].
2. gOSPREY can be combined with the iMinDEE [2] and DEEPPer [3] to further consider continuous backbone and side-chain flexibility, using the same framework as in OSPREY [1].
3. In gOSPREY, the GPU-based A* search algorithm only performs the single-precision floating-point arithmetic operations and typically runs for a limited period of time. Thus, the features of NVIDIA's expensive video cards Titan and Tesla, such as ECC memory and high-performance double-precision floating-point support, are not so useful to the program. Therefore, if the budget is the main concern, the high-end GeForce GTX video cards are a great choice for gOSPREY, and should have comparable or even better performance than the expensive Tesla and GeForce Titan cards.

Acknowledgments

This work is supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61033001, 61361136003. This work is supported by a grant to B.R.D. from the National Institutes of Health (R01 GM-78031).

References

1. Gainza P, Roberts KE, Georgiev I, Lilien RH, Keedy DA, Chen CY, Reza F, Anderson AC, Richardson DC, Richardson JS, et al. OSPREY: protein design with ensembles, flexibility, and provable algorithms. *Methods in enzymology*. 2013; 523:87. [PubMed: 23422427]
2. Gainza P, Roberts KE, Donald BR. Protein design using continuous rotamers. *PLoS computational biology*. 2012; 8(1):e1002335. [PubMed: 22279426]
3. Hallen MA, Keedy DA, Donald BR. Dead-end elimination with perturbations (DEEPPer): A provable protein design algorithm with continuous sidechain and backbone flexibility. *Proteins: Structure, Function, and Bioinformatics*. 2013; 81(1):18–39.
4. Roberts KE, Cushing PR, Boisguerin P, Madden DR, Donald BR. Computational design of a PDZ domain peptide inhibitor that rescues CFTR activity. *PLoS computational biology*. 2012; 8(4):e1002477. [PubMed: 22532795]
5. Gorczynski MJ, Grembecka J, Zhou Y, Kong Y, Roudaia L, Douvas MG, Newman M, Bielnicka I, Baber G, Corpora T, et al. Allosteric inhibition of the protein-protein interaction between the leukemia-associated proteins Runx1 and CBF β . *Chemistry & biology*. 2007; 14(10):1186–1197. [PubMed: 17961830]
6. Frey KM, Georgiev I, Donald BR, Anderson AC. Predicting resistance mutations using protein design algorithms. *Proceedings of the National Academy of Sciences*. 2010; 107(31):13707–13712.
7. Reeve SM, Gainza P, Frey KM, Georgiev I, Donald BR, Anderson AC. Protein design algorithms predict viable resistance to an experimental antifolate. *Proceedings of the National Academy of Sciences*. 2015; 112(3):749–754.
8. Reardon PN, Sage H, Dennison SM, Martin JW, Donald BR, Alam SM, Haynes BF, Spicer LD. Structure of an HIV-1-neutralizing antibody target, the lipid-bound gp41 envelope membrane proximal region trimer. *Proceedings of the National Academy of Sciences*. 2014; 111(4):1391–1396.

9. Rudicell RS, Do Kwon Y, Ko SY, Pegu A, Louder MK, Georgiev IS, Wu X, Zhu J, Boyington JC, Chen X, et al. Enhanced potency of a broadly neutralizing HIV-1 antibody in vitro improves protection against lentiviral infection in vivo. *Journal of virology*. 2014; 88(21):12669–12682. [PubMed: 25142607]
10. Chen CY, Georgiev I, Anderson AC, Donald BR. Computational structure-based redesign of enzyme activity. *Proceedings of the National Academy of Sciences*. 2009; 106(10):3764–3769.
11. Chazelle B, Kingsford C, Singh M. A semidefinite programming approach to side chain positioning with new rounding strategies. *INFORMS Journal on Computing*. 2004; 16(4):380–392.
12. Pierce NA, Winfree E. Protein design is NP-hard. *Protein Engineering*. 2002; 15(10):779–782. [PubMed: 12468711]
13. Kuhlman B, Baker D. Native protein sequences are close to optimal for their structures. *Proceedings of the National Academy of Sciences*. 2000; 97(19):10383–10388.
14. Marvin JS, Hellenga HW. Conversion of a maltose receptor into a zinc biosensor by computational design. *Proceedings of the National Academy of Sciences*. 2001; 98(9):4955–4960.
15. Shah PS, Hom GK, Mayo SL. Preprocessing of rotamers for protein design calculations. *Journal of computational chemistry*. 2004; 25(14):1797–1800. [PubMed: 15362137]
16. Street AG, Mayo SL. Computational protein design. *Structure*. 1999; 7(5):R105–R109. [PubMed: 10378265]
17. Kingsford CL, Chazelle B, Singh M. Solving and analyzing side-chain positioning problems using linear and integer programming. *Bioinformatics*. 2005; 21(7):1028–1039. [PubMed: 15546935]
18. Hong EJ, Lippow SM, Tidor B, Lozano-Pérez T. Rotamer optimization for protein design through MAP estimation and problem-size reduction. *Journal of computational chemistry*. 2009; 30(12):1923–1945. [PubMed: 19123203]
19. Zhou Y, Wu Y, Zeng J. *Computational Protein Design Using AND/OR Branch-and-Bound Search*. Research in Computational Molecular Biology. 2015
20. Xu J, Berger B. Fast and accurate algorithms for protein side-chain packing. *Journal of the ACM (JACM)*. 2006; 53(4):533–557.
21. Desmet J, Maeyer MD, Hazes B, Lasters I. The dead-end elimination theorem and its use in protein side-chain positioning. *Nature*. 1992; 356(6369):539–542. [PubMed: 21488406]
22. Leach AR, Lemon AP, et al. Exploring the conformational space of protein side chains using dead-end elimination and the A* algorithm. *Proteins Structure Function and Genetics*. 1998; 33(2):227–239.
23. Lippow SM, Tidor B. Progress in computational protein design. *Current opinion in biotechnology*. 2007; 18(4):305–311. [PubMed: 17644370]
24. Donald, BR. *Algorithms in structural molecular biology*. The MIT Press; 2011.
25. Georgiev I, Lilien RH, Donald BR. Improved pruning algorithms and divide-and-conquer strategies for dead-end elimination, with application to protein design. *Bioinformatics*. 2006; 22(14):e174–e183. [PubMed: 16873469]
26. Georgiev I, Lilien RH, Donald BR. The minimized dead-end elimination criterion and its application to protein redesign in a hybrid scoring and search algorithm for computing partition functions over molecular ensembles. *Journal of Computational Chemistry*. 2008; 29(10):1527–1542. [PubMed: 18293294]
27. Gregg, C.; Hazelwood, K. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. *Performance Analysis of Systems and Software (ISPASS)*, 2011 IEEE International Symposium on; 2011. p. 134–144.
28. Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, Satish N, Smelyanskiy M, Chennupati S, Hammarlund P. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*. 2010; 383:451–460.
29. Hart PE, Nilsson NJ, Raphael B. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*. 1968; 4(2):100–107.
30. Zhou Y, Xu W, Donald BR, Zeng J. An efficient parallel algorithm for accelerating computational protein design. *Bioinformatics*. 2014; 30(12):i255–i263. [PubMed: 24931991]

Figure 1.

An illustration of the traditional A* tree search algorithm. (a) An example of the A* search tree at a certain time. Nodes in dark shade represent the nodes that have already been expanded and visited. Nodes in light shade represent the frontier nodes which are stored in the priority queue and waiting to be expanded. The node labeled with a is the node with minimum heuristic function value in the priority queue and thus is currently being expanded. (b) The workflow of the node expansion operation for current node a .

Figure 2.

The illustration of the parallel node expansion operations in our parallel A* search algorithm. (a) An example of the A* search tree at a certain time. The meaning of Figure 2a is as same as that of Figure 1a except that now the algorithm is expanding nodes a , b , and c in parallel. The nodes labeled with a , b , and c are the nodes with the minimum heuristic function values in their respective priority queues. (b) The workflow on how multiple nodes are expanded simultaneously.

Figure 3.

Semi-log plots about the ratios of speedups and memory overhead of our GPU-based A* search algorithm. The x axes represent the magnitude of the design problem, while the y axes of Figure 3a and 3b represent the ratios of speedups and memory overhead of our GPU-based algorithm comparing to the traditional sequential A* search algorithm, respectively. The circle and square marks represent the results of our GPU-based A* search algorithm with 768 and 4992 parallel priority queues, respectively.