Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

Massively Parallel A* Search on a GPU

Yichao Zhou and **Jianyang Zeng***
Institute for Interdisciplinary Information Sciences
Tsinghua University, Beijing, P. R. China

Abstract

A* search is a fundamental topic in artificial intelligence. Recently, the general purpose computation on graphics processing units (GPGPU) has been widely used to accelerate numerous computational tasks. In this paper, we propose the *first* parallel variant of the A* search algorithm such that the search process of an agent can be accelerated by a single GPU processor in a *massively parallel* fashion. Our experiments have demonstrated that the GPU-accelerated A* search is efficient in solving multiple real-world search tasks, including combinatorial optimization problems, pathfinding and game solving. Compared to the traditional sequential CPU-based A* implementation, our GPU-based A* algorithm can achieve a significant speedup by up to 45x on large-scale search problems.

1 Introduction

A* search is one of the most widely used best-first search algorithms in artificial intelligence, guided by a heuristic function (Hart, Nilsson, and Raphael 1968). Intrinsically, A* search is a sequential algorithm that is hard to be parallelized efficiently. In this paper, we propose the *first* parallel variant of the A* search algorithm that is able to run on a graphics processing unit (GPU) processor in a *massively* parallel fashion, called GA*.

1.1 General-Purpose Computation on GPUs

General-purpose computation on graphics processing units (aka GPGPU) is a new technology to use graphics processing units to accelerate traditional CPU-based computational tasks. CPUs and GPUs handle a computational task differently. A CPU usually contains several highly optimized cores for sequential instruction execution, while a GPU typically contains thousands of simpler but more efficient cores that are good at manipulating different data at the same time. So we need to modify search algorithms originally designed for a CPU to exploit such a large amount of parallelism brought by a GPU.

A GPU typically performs better in floating-point operations than a CPU around the same price. For example, an NVIDIA GeForce GTX 580M has six times more theoretical GFLOPS (giga floating-point operation per second) than that of Intel Core i7-3960 (Intel Corporation 2011; NVIDIA Corporation 2013). In the domain of A* search, some applications, such as protein design (Leach and Lemon 1998), require heavy floating-point operations in the stages of computing heuristic functions. When the computation of heuristic functions becomes the main bottleneck of the whole algorithm, these applications can significantly benefit from the advantage of GFLOPS on a GPU.

In addition, a GPU has a memory system which is independent of that of its CPU. Such a design provides a higher bandwidth for accessing the global memory. In other words, cores of a GPU can retrieve and write data from/to the global memory much faster than a CPU. Therefore, even for those A* search applications which use only integer heuristic functions, they can still benefit a lot from the acceleration provided by a GPU. This is because A* search usually requires massive access to the global memory for storing and retrieving states from/to both open and closed lists, and higher global memory bandwidth can lead to a faster expansion rate during A* search.

On the other hand, using an independent global memory system has certain disadvantage. Before and after computation, data need to be transferred between the memories of CPU and GPU systems through a relatively slow PCI-E bus. Nevertheless, for those applications which have to explore exponential search space, A* search usually has a relatively small ratio between the amount of time used to transfer input/output data and the amount of time spent on computation. Thus, in these applications, the data transfer overhead is usually negligible.

1.2 Related Work

Previous work on parallelization of A* search and best-first search mainly focused on CPU-based implementations. One of the earliest parallel A* search algorithms was developed in (Kumar, Ramesh, and Rao 1988). With the advent of multi-core CPUs in the commercial desktop processor market, Burns et al. (2009) proposed a general approach for best-first search on a multi-core system with shared memory. Recently, Kishimoto, Fukunaga, and Botea presented a distributed A* search algorithm, called HDA*, on a CPU cluster by assigning internal states to different machines using a

*Corresponding author. Email: zengjy321@tsinghua.edu.cn.

hashing function (Kishimoto, Fukunaga, and Botea 2013).

As GPGPU is a relatively new technology, feasibility of applying A* search on a GPU system is rarely explored. Researchers from NVIDIA Corporation described an efficient GPU implementation for multi-agent A* search, i.e., finding the shortest paths between multiple pairs of nodes in parallel in a sparse graph, based on the CUDA programming environment (Bleiweiss 2008; Pan, Lauterbach, and Manocha 2010). Although their methods were able to solve multiple small A* search problems simultaneously, they cannot parallelize an individual A* search process to solve a large-scale problem (e.g., the protein design problem described in Section 5.3). Another related work is (Sulewski, Edelkamp, and Kissmann 2011), which parallelized the node expansion step on a GPU and performed the remaining task on a CPU for integer-cost Dijkstra. On the other hand, our method is a pure GPU algorithm for general A* search, and thus it does not require exchanging much data between a CPU and a GPU.

2 Traditional A* Search

Traditional implementations of A* search usually use two lists to store the states during its expansion, i.e., the open list and the closed list. The closed list stores all the visited states, and is used to prevent unnecessary repeated expansion of the same state. This list is often implemented by a linked hash table to detect the duplicated nodes. The open list normally stores the states whose successors have not been fully explored yet. The open list uses a priority queue as its data structure, typically implemented by a binary heap. States in the open list are sorted according to a heuristic function $f(x)$:

$$f(x) = g(x) + h(x), \tag{1}$$

where the function $g(x)$ is the distance or cost from the starting node to current state x, and the function $h(x)$ defines the estimated distance or cost from current state x to the end node. We call the function value of $f(x)$ the f value. If the f values of a problem are small integers, the open list can also be efficiently implemented with buckets (Dial 1969).

In each round of A* search, we extract the state with the minimum f value from the open list, expand its outer neighbors and check for duplication. The step of node duplication detection is not required for A* tree search. After that, we calculate the heuristic functions of the resulting states and then push them back to the open list. If the nodes of some states have already been stored in the open list, we only update their f values for the open list. The pseudocode of traditional A* search is provided in Section A1 of the appendix (Zhou and Zeng 2014).

In A* search, we require the heuristic function to be *admissible* for optimality, i.e., $h(x)$ is never greater than the actual cost or distance to the end node. If the search graph is not a tree, a stronger condition called *consistency*: $h(x) \leq d(x, y) + h(y)$, where $d(x, y)$ represents the cost or distance from x to y, guarantees that once a state is extracted from the queue, the path that it follows is optimal (Pearl 1984).

3 Parallel A* Search

3.1 Parallelized Computation of Heuristic Functions

In some applications of A* search, computing the heuristic functions is quite expensive and becomes the bottleneck of the whole algorithm. An example of such applications is shown in Section 5.3. In these applications, the first step to parallelize A* search on a GPU is to parallelize the computation of heuristic functions. This step is straightforward, and the rationality is simply based on the observation that the computation of heuristic functions for each expanded state is mutually independent.

3.2 Parallel Priority Queues

After incorporating the procedure described in Section 3.1, we get a simple parallel algorithm. However, this A* algorithm is still inefficient on the GPU computational framework. Here are the two problems. First, the degree of parallelism is limited by the outer degree of each node in the search graph. As described in Section 1.1, a GPU processor usually contains thousands of cores. On the other hand, in some applications, such as pathfinding in a grid graph, the degree of a node is usually less than ten, which limits the degree of parallelism on a GPU platform. Second, this simple algorithm still has many sequential parts which have not been fully parallelized yet. For example, the EXTRACT and PUSH-BACK operations for the open list take $O(\log N)$ time to finish using a binary heap, where N is the total number of elements in the list. For those applications in which the computation of heuristic functions is relatively cheap, the priority queue operations will become the most time-consuming parts in A* search. Sequential operations are inefficient for a GPU processor because it will only exploit a tiny proportion of the GPU hardware, and the single-thread performance of a GPU is far worse than that of a CPU.

The first problem may be solved by extracting multiple states from the priority queue to increase the degree of parallelism in the computation of heuristic functions. However, the priority queue operations still run in a sequential mode. One may want to solve the second problem by using a concurrent data structure for the priority queue. Unfortunately, existing lock-free concurrent priority queues, such as those proposed in (Sundell and Tsigas 2003), cannot run efficiently on the SIMD architecture of a modern GPU processor, as they require the usage of compare-and-swap (CAS) operations.

To address both problems, we propose a new algorithm, called GA*, to further exploit the parallelism of the GPU hardware. Instead of just using one single priority queue for the open list, we allocate a large number of (usually thousands of, for a typical GPU processor) priority queues during A* search. Each time we extract multiple states from individual priority queues, which thus parallelizes the sequential part in the original algorithm. Meanwhile, the GA* algorithm also increases the number of expanding states at each step, which further improves the degree of parallelism for the computation of heuristic functions as the part described in Section 3.1.

Algorithm 1 describes the framework of our algorithm using parallel priority queues and Figure 1 provides the data flow of the open list. For each state s in the open list or closed list, s.node stands for the last node in the path represented by s, $s.f$ and $s.g$ store the values of $f(s)$ and $g(s)$, respectively, and s.prev stores the pointer to the previous state that expanded s, which is used to regenerate a path from a given state. List S stores the expanded nodes and list T stores the nodes after the removal of duplicated nodes. Lines 24-29 detect the duplicated nodes, where $H[n]$ represents the state in the closed list in which the last node in its path is node n. Through synchronization operations, which are computationally cheap on GPUs, we can push nodes expanded from the same parent into different queues (Line 32), as nodes with the same parent tend to have similar priority values.

Algorithm 1 GA*: Parallel A* search on a GPU

```
1:  procedure GA*(s, t, k)
                    ▷ find the shortest path from s to t with k queues
2:      Let {Q_i}_{i=1}^k be the priority queues of the open list
3:      Let H be the closed list
4:      PUSH(Q_1, s)
5:      m ← nil                          ▷ m stores the best target state
6:      while Q is not empty do
7:          Let S be an empty list
8:          for i ← 1 to k in parallel do
9:              if Q_i is empty then
10:                 continue
11:             end if
12:             q_i ←EXTRACT(Q_i)
13:             if q_i.node = t then
14:                 if m = nil or f(q_i) < f(m) then
15:                     m ← q_i
16:                 end if
17:                 continue
18:             end if
19:             S ← S + EXPAND(q_i)
20:         end for
21:         if m ≠ nil and f(m) ≤ min_{q∈Q} f(q) then
22:             return the path generated from m
23:         end if
24:         T ← S
25:         for s' ∈ S in parallel do
26:             if s'.node ∈ H and H[s'.node].g < s'.g then
27:                 remove s' from T
28:             end if
29:         end for
30:         for t' ∈ T in parallel do
31:             t'.f ← f(t')
32:             Push t' to one of priority queues
33:             H[t'.node] ← t'
34:         end for
35:     end while
36: end procedure
```

By assigning more priority queues (i.e., increasing the parameter k in Algorithm 1), we can increase the degree of parallelism to further exploit the computational power of the GPU hardware. However, we cannot increase the degree of parallelism infinitely, as extracting multiple states in parallel rather than a single state with the best f value so far can lead to the overhead on the number of expanded states. The more degree of parallelism we have, the more states GA* needs to generate to find the optimal path. So we need to balance the trade-off between the overhead of extra expanded states and the degree of parallelism to exploit the computational power of the GPU hardware. In Section 5, empirical studies will show that our new parallel A* search algorithm can significantly improve the efficiency of the traditional A* search approach with an acceptable range of space overhead.

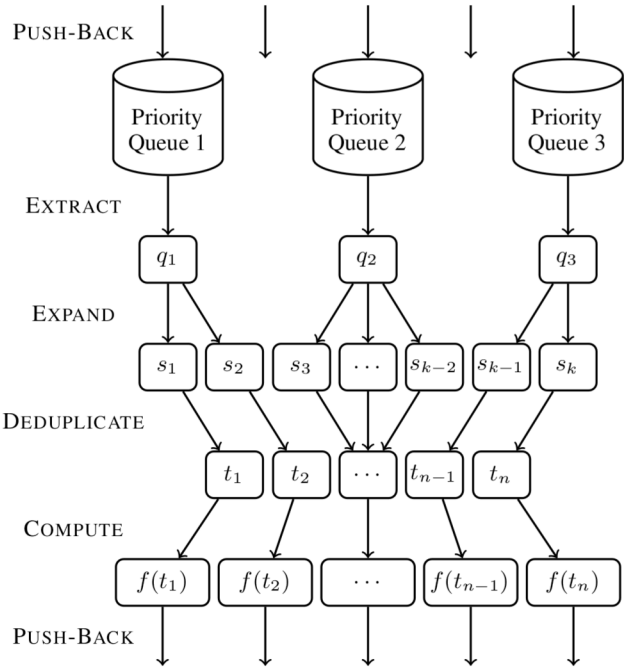

Figure 1: Data flow of the open list. The symbols in this diagram matches those in Algorithm 1. In this example, the number of the parallel priority queues is three. In practice, we usually use thousands of parallel priority queues for a typical GPU processor.

3.3 Node Duplication Detection on a GPU

In A* graph search, it is possible that we try to expand a state whose node has been already visited. If the f value of the new state is not smaller than that of the existing state in the closed list, it is safe to prevent this state from being visited again. This procedure is called *node duplication detection*.

Depending on individual applications, the difficulty of node duplication detection varies. In A* tree search, this step is not necessary. For A* graph search in which the whole graph fits into the memory, we can simply use an array table to accomplish this task. Node duplication detection becomes more complicated when search space is too large to fit into the memory. Examples include the applications of A* search in solving the Rubiks cube and sliding puzzle problems. Such applications often have exponential search space, which makes it unrealistic to use a preallocated table to detect the duplication.

Node duplication detection requires a data structure that can support both INSERT and QUERY operations. INSERT inserts a key-value pair to the data structure. QUERY asks whether a key is in this data structure or not. If so, it returns its associate value. On a CPU platform, we often use a linked hash table or a balanced binary search tree (such as red-black tree). However, it is quite difficult to extend these data structures to parallel node duplication detection on a GPU. For example, it will be tricky to handle the situation in which two different states are inserted into a same bucket simultaneously in a linked hash table. So we need to resort to a more appropriate data structure for efficient node duplication detection on a GPU.

In this paper, we propose two different node duplication detection mechanisms on a GPU, each having its own virtues. The first scheme is called *parallel cuckoo hashing*, which is a parallelized version of the traditional cuckoo hashing algorithm (Pagh and Rodler 2001). The second method is called *parallel hashing with replacement*, which is a probabilistic data structure particularly designed for simplicity on a GPU. The main advantage of parallel cuckoo hashing is that it guarantees to detect all the duplicated nodes. Therefore, less redundant nodes will be generated, and thus the algorithm occupies less memory space. On the other hand, though parallel hashing with replacement may miss a small number of duplicated nodes during detection, it runs faster and is easier to implement.

Parallel Cuckoo Hashing *Cuckoo hashing* (Pagh and Rodler 2001) is a relatively new hashing algorithm. In this hashing scheme, we use multiples hash tables and functions to combat collision. In other words, each node is placed into one of hash tables with different hash functions. Similar to the behavior of cuckoo chicks in pushing eggs out of their hatches, in cuckoo hashing, a new inserted node can push the old node to a different position. There is a small possibility that nodes will push each other forever and the insertion cannot be finished. In that case, we need to rebuild the hash tables with another set of hash functions.

Implementation of cuckoo hashing on a GPU is a bit complicated due to the specific architecture of GPUs. As a typical GPU has thousands of logic cores, which are further divided into blocks. Sharing data among cores inside a block is much faster than sharing data between blocks. Our parallel cuckoo hashing scheme on a GPU is an online version of (Alcantara et al. 2009). More specifically, we assign one cuckoo hashing instance (i.e., $\mathcal{H}(i)$ in Algorithm A2 of the appendix (Zhou and Zeng 2014)) to each block so that the synchronization can be done efficiently. Suppose that the GPU has k blocks. Then we use a partition hash function $\hbar(x) \in \{0, 1, \ldots, k-1\}$ to assign nodes to different blocks. After that, nodes assigned to the same block can be inserted to the particular hash buckets of the cuckoo hashing instance in that block in parallel. The idea of using a hash function to assign states to different blocks also occurs in (Kishimoto, Fukunaga, and Botea 2013). We can also place multiple cuckoo hash instances in a GPU block to obtain a better performance. The detailed pseudocode of parallel cuckoo hashing is provided in Algorithm A2 of the appendix (Zhou and Zeng 2014). In Section 4, we will provide the theoretical analysis of the failure rate of parallel cuckoo hashing, which shows that the probability of rebuilding the hash table is quite small.

Parallel Hashing with Replacement The parallel cuckoo hashing is a heavy-weight solution for node duplication detection on a GPU platform. Here, we provide a light-weight alternative in the GA* algorithm, called *parallel hashing with replacement*. The key observation here is that we do not need to guarantee that all the duplicated nodes must be detected. This is acceptable because having some states with duplicated nodes in the open list does not affect the correctness of the whole algorithm. If we are allowed to miss some nodes, then the rigorous collision detection process is not strictly required.

Basically, parallel hashing with replacement is similar to cuckoo hashing, except that when a new node has to occupy the position of an old one, instead of pushing the old node to a different position, parallel hashing with replacement simply drops the old node. This modification makes it simpler, faster, and easier to be parallelized. The detailed pseudocode of parallel hashing with replacement can be found in Algorithm A3 of the appendix (Zhou and Zeng 2014).

The choice between parallel cuckoo hashing and parallel hashing with replacement depends on individual applications. If the space limitation is the main concern, one can choose parallel hashing with replacement so that it can detect all the duplicated nodes. On the other hand, parallel hashing with replacement can be chosen if speed or simplicity is the major consideration.

3.4 Memory Bounded A* Search on a GPU

A* search often suffers from the memory bottleneck when it needs to save a huge number of visited states in the closed list, which are necessary for detecting duplicated nodes and regenerating the optimal path. When the system runs out of memory, it is impossible to continue the search process to find the optimal solution, which is very likely to happen when search space is exponential.

Several methods have been proposed to address this problem in sequential A* search. For examples, iterative-deepening A* (IDA*) (Korf 1985) and recursive best-first search (RBFS) (Korf 1993) mimic the behavior of traditional A* using the depth-first-search scheme. Both approaches may visit the same state multiple times due to the lack of closed lists. Although this issue can be solved through *transposition tables* (Reinefeld and Marsland 1994; Akagi, Kishimoto, and Fukunaga 2010), it is still hard to extend their depth-first-search schemes on the GPU platform.

Another option is to sacrifice the optimality when memory is inadequate. Algorithms such as simplified memory bounded A* (SMA*) (Russell 1992) and frontier search (Korf et al. 2005) belong to this category. Among them, we found that frontier search is simpler and more suitable for a GPU implementation. In this search mechanism, we discard all the nodes in the closed list and only keep the most promising states in the open list according to their f values. The *scan primitive* (Sengupta et al. 2007) and *GPU radix*

sort can be used to select the requisite states efficiently in parallel on the GPU platform.

4 Analysis

In this section, we provide several theoretical results about our algorithm. Due to space limitation, their proofs are provided in the appendix (Zhou and Zeng 2014).

Theorem 4.1. *Let $h'(x)$ denote the optimal cost from x to the target node. If the given heuristic function is* admissible*, i.e., $h(x) \leq h'(x)$ for each node x, the first solution returned by GA* must be the optimal solution.*

Theorem 4.1 states that our algorithm can guarantee to find the global optimal solution as long as the heuristic function is admissible. As in the sequential A* search algorithm, *consistency* is still an important property for the heuristic function used by the GA* algorithm. Here, we also provide a theoretical bound on the number of nodes expanded by GA*, given that the heuristic function is consistent.

Corollary 4.2. *The GA* algorithm with k parallel queues on a graph with N nodes will expand at most kN states, provided that the heuristic function is* consistent*.*

We also derive the failure probability for the parallel cuckoo hashing algorithm when $d = 2$ (i.e., using only two hash functions).

Theorem 4.3. *The probability that the parallel cuckoo hashing algorithm with k cuckoo hashing instances, in which each instance has $n = (1 - \varepsilon)m$ nodes and two hash tables of size m, fails to construct is equal to $c(\varepsilon)\frac{k^2}{m} + O\left(\frac{1}{m^2}\right)$, where $\varepsilon \in (0, 1)$ and $c(\cdot)$ is a function of ε.*

By Theorem 4.3, we can conclude that if $k^2 = o(m)$, which is normally the case, the probability of rebuilding the hash tables is very low.

Finally, we prove that our parallel hashing with replacement algorithm is able to handle the situation with the duplicated nodes in its input in most cases, which means that the local duplication detection is not necessary for parallel hashing with replacement.

Theorem 4.4. *If the input of the parallel hashing with replacement algorithm contains duplicated nodes t_0, then the result will contain at most one instance of t_0 if there does not exists any other node t'_1 such that $h_i(t_0) = h_j(t_1)$ where h_i and h_j are two hash functions used in this parallel hashing with replacement.*

5 Evaluation

In this section, we compared the performance of our parallel GPU-based A* search algorithm against that of the traditional CPU-based sequential A* search algorithm. We chose three classical problems with different characteristics for comprehensive comparisons.

The CPU used in our experiments was Intel Xeon E5-1620 3.6GHz with 16GB global memory. The graphic card (GPU) that we used was a single NVIDIA Tesla K20c with 4.8GB off-chip global memory and 2496 CUDA cores.

In these experiments, we evaluate GA* with different numbers of parallel priority queues, which were chosen to be multiple folds of the number of GPU cores to fully exploit the underlying thread-scheduling mechanism of a GPU platform. More specifically, when a core is waiting for memory access of a thread, the GPU driver can schedule another thread to this core.

The reason why we did not compare our method with existing CPU-based parallel A* search algorithms is that it is impractical to port an algorithm originally designed a CPU to a GPU platform. This is because GPUs are designed as an SIMD machine, i.e., a group of cores can only execute the same instruction at the same time, while CPUs do not have such a restriction.

5.1 Sliding Puzzle

Sliding puzzle has been widely used to test the performance of heuristic search algorithms (Korf 1985; Burns et al. 2009). We tested both 15-puzzles and 24-puzzles in our experiments. For 15-puzzles, the test data were randomly generated. For 24-puzzles, we manually created several test data sets by moving the tiles from the target state to control the number of steps below a current value so that both CPU- and GPU-based A* search algorithms can generate the optimal solutions within available memory for fair comparisons.

The heuristic function we used here is called the *disjoint pattern database* (Korf and Felner 2002), which uses the sum of the results from multiple pattern databases (Culberson and Schaeffer 1998) and still guarantees consistency. Pattern databases are often used in heuristic search applications, such as solving Rubik's Cube (Korf 1997), and are much more efficient than the Manhattan distance based heuristic functions.

	Steps	Time	# of States	Exp. Rate
4x4-1 CPU		1,687	1,990,351	1,179.8
4x4-1 GA1	61	184	3,295,248	17,899.0
4x4-1 GA2		174	3,135,554	18,010.3
5x5-1 CPU		52,972	45,437,838	857.8
5x5-1 GA1	62	3,236	52,633,552	16,327.0
5x5-1 GA2		1,729	53,995,567	31,368.0
5x5-2 CPU		98,935	81,564,879	824.4
5x5-2 GA1	64	4,532	73,408,175	16,314.6
5x5-2 GA2		2,187	67,693,001	31,326.9

Table 1: The comparison results on sliding puzzle problems. Time is measured in millisecond and "Exp. Rate" indicates the number of nodes expanded per millisecond.

The comparison results between GA* and traditional A* search algorithms are provided in Table 1. The rows labeled with GA1 and GA2 represent the GA* algorithms with 2496 and 9984 parallel priority queues, respectively. Because of exponential search space, our GPU-based A* search algorithm was much more efficient than the single-thread CPU-based A* algorithm in solving this problem. We were able to achieve a near 30x speedup. When performing these experiments, we found that the number of expanded states on the GPU can fluctuate by as much as 20 percent. In Table 1, all

the items are averaged values over ten runs for each input. In experiment 5x5-2, the main reason that the parallel A* algorithm expanded less states than the sequential A* algorithm was that there were 51,237,009 states in the open list whose f values were equal to 64, and the CPU expanded more of these nodes than the GPU before finding the target state. We can break ties among states with the same f values using their h values to resolve this issue during the comparison of priority.

5.2 Pathfinding

Pathfinding is an another application in which A* search can be used to find the shortest path between two vertices in a graph. In our experiments we evaluated the running time required to find the shortest path between two vertices in a grid graph. The graphs in our experiments were 8-connected, i.e., a vertex can had a edge to another vertices that touch its edges or corners. The *diagonal distance* was used as the heuristic function.

Data	Length	Time	# of States	Exp. Rate
zigzag CPU		58,081	39,703,385	829.9
zigzag GA1	14,139	15,746	48,202,104	3,061.3
zigzag GA2		8,322	74,947,693	9,006.0
random CPU		32,562	28,083,916	862.5
random GA1	13,251	12,527	37,420,033	2,987.2
random GA2		9,563	76,438,961	7,993.2
empty CPU		8	3,000	375.0
empty GA1	9,999	4,475	18,480,573	4,129.7
empty GA2		8,114	75,075,995	10,553.3

Table 2: The comparison results on path-finding problems. Time is measured in millisecond and "Exp. Rate" indicates the number of nodes expanded per millisecond.

The size of the graphs we used in these experiments was 10000×10000. GA1 and GA2 represent the GA* algorithms with 2496 and 9984 parallel priority queues, respectively. As shown in Table 2, the performance of the GPU-based A* search varied a lot depending on the graphs. In the graph "zigzag1", the GPU-based A* search algorithm is about 7 times faster than the sequential CPU-based A* search algorithm. On the other hand, in an empty graph, the CPU-based A* search finished in almost no time while the GPU-based A* search still used 6.5 seconds. In this case, GA* expanded much more states than traditional A* search.

From the above results, we found that the GPU-based A* search algorithm is more efficient when the heuristic function of the problem deviates from the actual distance to some extent. In this case, more nodes will be expanded in A* search, which allows more parallelism for a GPU processor. Thus, GA* is able to provide a better worst-case performance compared to the traditional A* search algorithm. On the other hands, massively parallel A* search can have a bad performance in non-exponential search space when the heuristic function is closed to the actual distance, due to the lack of parallelism in such a case.

5.3 Protein Design

Protein design is an important problem in computational biology, which can be formulated into finding the most probable explanation of a graphical model with a complete graph. A* tree search has been developed to solve this problem (Leach and Lemon 1998; Donald 2011; Zhou et al. 2014).

In this problem, we want to minimize this energy function: $E_T = \sum_{i_r \in A} E_1(i_r) + \frac{1}{2} \sum_{i_r \in A} \sum_{j_s \in A} E_2(i_r, j_s)$, where A is the allowed node set, $E_1(i_r)$ is the self energy for the node r in position i, and $E_2(i_r, j_s)$ is the pairwise energy between nodes i_r and j_s.

In each level i of the A* search tree, the node at position i is fixed. The $g(x)$ function is the sum of energy among the fixed nodes in state x. We used the following heuristic function: $h(x) = \sum_{i \in U(x)} \min_r(E_1(i_r) + \sum_{j_s \in D(x)} E_2(i_r, j_s) + \sum_{k \in U(x)} \min_u E_2(i_r, k_u))$.

Data	Time	# of States	Exp. Rate
2CS7 CPU	100,677	24,941,885	247.7
2CS7 GPU	2,134	34,580,645	16,204.6
2DSX CPU	53,604	26,119,553	487.3
2DSX GPU	1,318	33,151,597	25,152.9
3D3B CPU	32,980	21,469,007	651.0
3D3B GPU	909	27,054,226	29,762.6

Table 3: The comparison results on protein design problems. Time is measured in millisecond and "Exp. Rate" indicates the number of nodes expanded per millisecond.

Table 3 shows the performance comparison results between different algorithms on this problem. In these experiments, GA* with 4992 priority queues was used. The heuristic function for the protein design problem was quite complex, and the computation of the heuristic function was the bottleneck of this problem, while a GPU can be effectively used to accelerate this computation. In addition, unlike the sliding puzzle experiments, in A* tree search it was impossible to expand a tree node multiple times, which can reduce the difference in the number of expanded states between the massively parallel version and the single-thread version. As shown in Table 3, we were able to achieve a near 45x speedup in this problem.

6 Conclusion and Future Work

In this paper, we propose the first A* search algorithm on a GPU platform. Our experiments have demonstrated that the GPU-based massively parallel A* algorithm can have a considerable speedup compared to the traditional sequential A* algorithm, especially when search space is exponential. By fully exploiting the power of the GPU hardware, our GPU-based parallel A* can significantly accelerate various computation tasks.

In the future, algorithms for the bidirectional A* search can be designed and evaluated on the GPU platform. The possibility of using A* search in a multi-GPU environment can also be explored.

Acknowledgement

We thank the three anonymous reviewers for their insightful and helpful comments. We thank Dr. Wei Xu for his help on parallel computation and discussions of the protein design problem.

Funding: This work was supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61033001, 61361136003 and 61472205, and China's Youth 1000-Talent Program.

References

Akagi, Y.; Kishimoto, A.; and Fukunaga, A. 2010. On transposition tables for single-agent search and planning: Summary of results. In *Third Annual Symposium on Combinatorial Search*.

Alcantara, D. A.; Sharf, A.; Abbasinejad, F.; Sengupta, S.; Mitzenmacher, M.; Owens, J. D.; and Amenta, N. 2009. Real-time Parallel Hashing on the GPU. In *ACM Transactions on Graphics (TOG)*, volume 28, 154. ACM.

Bleiweiss, A. 2008. GPU accelerated pathfinding. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 65–74. Eurographics Association.

Burns, E.; Lemons, S.; Zhou, R.; and Ruml, W. 2009. Best-first Heuristic Search for Multi-core Machines. In Boutilier, C., ed., *IJCAI*, 449–455.

Culberson, J. C., and Schaeffer, J. 1998. Pattern Databases. *Computational Intelligence* 14(3):318–334.

Dial, R. B. 1969. Algorithm 360: Shortest-path forest with topological ordering. *Communications of the ACM* 12(11):632–633.

Donald, B. R. 2011. *Algorithms in structural molecular biology*. The MIT Press.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on* 4(2):100–107.

Intel Corporation. 2011. *Intel Microprocessor Export Compliance Metrics*.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2013. Evaluation of a Simple, Scalable, Parallel Best-first Search Strategy. *Artificial Intelligence* 195:222–248.

Korf, R. E., and Felner, A. 2002. Disjoint Pattern Database Heuristics. *Artificial intelligence* 134(1):9–22.

Korf, R. E.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the ACM (JACM)* 52(5):715–748.

Korf, R. E. 1985. Depth-first Iterative-deepening: An Optimal Admissible Tree Search. *Artificial intelligence* 27(1):97–109.

Korf, R. E. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.

Korf, R. E. 1997. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In *AAAI/IAAI*, 700–705.

Kumar, V.; Ramesh, K.; and Rao, V. N. 1988. Parallel Best-first Search of State-space Graphs: A Summary of Results. In Shrobe, H. E.; Mitchell, T. M.; and Smith, R. G., eds., *AAAI*, 122–127. AAAI Press / The MIT Press.

Leach, A. R., and Lemon, A. P. 1998. Exploring the Conformational Space of Protein Side Chains Using Dead-end Elimination and the A* Algorithm. *Proteins Structure Function and Genetics* 33(2):227–239.

NVIDIA Corporation. 2013. *NVIDIA Tesla Technical Specifications*.

Pagh, R., and Rodler, F. F. 2001. *Cuckoo Hashing*. Springer.

Pan, J.; Lauterbach, C.; and Manocha, D. 2010. g-Planner: Real-time Motion Planning and Global Navigation Using GPUs. In *AAAI*.

Pearl, J. 1984. *Heuristics*. Addison-Wesley Publishing Company Reading, Massachusetts.

Reinefeld, A., and Marsland, T. A. 1994. Enhanced iterative-deepening search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 16(7):701–710.

Russell, S. 1992. Efficient Memory-bounded Search Methods. *Proceedings of the 10th European Conference on Artificial intelligence*.

Sengupta, S.; Harris, M.; Zhang, Y.; and Owens, J. D. 2007. Scan Primitives for GPU Computing. In *Graphics Hardware*, volume 2007, 97–106.

Sulewski, D.; Edelkamp, S.; and Kissmann, P. 2011. Exploiting the computational power of the graphics card: Optimal state space planning on the gpu. In *ICAPS*.

Sundell, H., and Tsigas, P. 2003. Fast and Lock-free Concurrent Priority Queues For Multi-thread Systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 11–pp. IEEE.

Zhou, Y., and Zeng, J. 2014. Massively parallel A* search on a GPU: Appendix. Available at http://iiis.tsinghua.edu.cn/%7Ecompbio/papers/aaai2015apx.pdf.

Zhou, Y.; Xu, W.; Donald, B. R.; and Zeng, J. 2014. An efficient parallel algorithm for accelerating computational protein design. *Bioinformatics* 30(12):255–263.