Massively Parallel A* Search on a GPU: Appendix

Yichao Zhou Jianyang Zeng

A1 Pseudocode of Traditional A* Search

Algorithm A1 Sequential A* graph search algorithm

```
1:  procedure A*(s, t)                          ▷ find the shortest path from s to t
2:      Let Q be a priority queue               ▷ Q is the open list
3:      PUSH(Q, s)
4:      Let H be a linked hash table            ▷ H is the close list
5:      H ← ∅
6:      while Q is not empty do
7:          q ←EXTRACT(Q)
8:          if q = t then
9:              return the path found
10:         end if
11:         R ← EXPAND(q)
12:         S ← R − H                           ▷ deduplication (needless for A* tree search)
13:         for s ∈ S do
14:             PUSH-BACK-OR-UPDATE(Q, s, f(s))   ▷ f is the heuristic function
15:         end for
16:         H ← H ∪ {q}
17:     end while
18: end procedure
```

Algorithm A1 gives the pseudocode of the traditional A* search algorithm. In each round, we extract the state with the minimum f value from the open list (Line 7). Then we expand its outer neighbors and check for node duplication (Line 12). The step for node deduplication is not required for tree search. Finally, we calculate the f values of the new expanded states and push them back to the open list (Line 14). If the nodes of some states are already in the open list, we only update their f values for the open list.

In order to find the global optimal solution with an *admissible* heuristic function rather than a *consistent* one, in Line 12 we need to keep the state that has a shorter path from the starting node to the duplicated node (Russell and Norvig, 2009).

A2 Node Duplication Detection

A2.1 Parallel Cuckoo Hashing

The lookup operations of traditional cuckoo hashing can be performed in constant time in the worst case. We only need to check whether the query node exists in any of the tables or not. Below is a sequential implementation of the insertion procedure in the cuckoo hashing scheme using two tables with hash functions $h_1(x)$ and $h_2(x)$, respectively:

1. Let $z \in \{1, 2\}$. Initially, $z = 1$;
2. Let $H[h_z(x)]$ denote the bucket indexed by the hash function $h_z(x)$. If any of the buckets $H[h_1(x)]$ or $H[h_2(x)]$ is empty, we place node x into that bucket and terminate the procedure;
3. Otherwise, we displace the node currently occupying position $H[h_z(x)]$ and place x into $H[h_z(x)]$;
4. Alternate z between 1 and 2; Go back to Step 2 again to insert the node displaced.

There is a small possibility that the above procedure runs into a cycle and never ends. A result in closed form of this possibility was derived in (Kutzelnigg, 2006) when $d = 2$ (i.e., using two hash tables). In practice, we can solve this problem by terminating the procedure after a certain number of iterations and rebuilding the hash tables using a new set of hash functions.

Algorithm A2 describes the pseudocode of parallel cuckoo hashing, in which CUCKOO-HASH-INSERT stands for the partition procedure and INSERT stands for the parallel insert procedure. The parameter $\mathcal{H}$ of CUCKOO-HASH-INSERT is an array of cuckoo hashing instance, and $\mathcal{H}[i]$ stands for the cuckoo hashing instance on block i. Parameter T is a list of nodes that need to be inserted. The third parameter k stands for the total number of the blocks available on the GPU platform. Lines 2-7 assign nodes to different blocks according to $\hbar(x)$ and stores the partitioned node temporarily in $\mathcal{T}[\hbar(x)]$. Lines 8-12 launch the parallel insertion process.

Parameter H in the procedure INSERT is the actual cuckoo hashing instance for the block of current thread and t is the node to be inserted. The variable d is the number of hash tables in cuckoo hashing. Lines 17-23 scan for an empty bucket to insert the current node. The **thread synchronization** operation in Line 24 is a common operation in a GPU algorithm to wait until all other threads in the same block have reached that line. The check in Line 25 is necessary because other threads may write to the same bucket in Line 19. Finally, if we fail to find a suitable bucket to write, we evict a node randomly from these valid buckets and place the current node into its position in Lines 28-29. The function ATOMIC-SWAP swaps two parameters in the memory with one machine instruction to avoid the race hazard. After that, we go back to Line 16 and try to insert the evicted node.

For simplicity, the procedure of rebuilding the hash tables is not included in Algorithm A2. Also, we need to guarantee that the input T of CUCKOO-HASH-INSERT does not contain two identical nodes. This can be done by sorting all elements in T and then removing the same adjacent nodes to deduplicate locally in advance. The GPU radix sort (Sintorn and Assarsson, 2008) and the GPU scan (Sengupta et al., 2007) can be used in this local deduplication stage.

Algorithm A2 Parallel cuckoo hashing on a GPU

```
1:  procedure CUCKOO-HASH-INSERT(ℋ, T, k)
2:      for i ← 0 to k − 1 do
3:          𝒯[i] ← ∅
4:      end for
5:      for all t ∈ T do                          ▷ dispatch nodes according to ℏ(x)
6:          PUSH-BACK(𝒯[ℏ(t)], t)
7:      end for
8:      for i ← 0 to k − 1 in parallel do
9:          for j ← 1 to |𝒯[i]| in parallel do
10:             call INSERT(ℋ[i], 𝒯[i][j]) on block i
11:         end for
12:     end for
13: end procedure
14: procedure INSERT(H, t)
15:     z ← −1
16:     while true do
17:         for i ← 0 to d − 1 do
18:             if H[h_i(t)] is empty then
19:                 H[h_i(t)] ← t
20:                 z ← i
21:                 break
22:             end if
23:         end for
24:         thread synchronization
25:         if z ≠ −1 and H[h_z(t)] = t then
26:             return
27:         end if
28:         Let r be a random number in {0, 1, . . . , d − 1}
29:         ATOMIC-SWAP(H[h_r(t)], t)
30:     end while
31: end procedure
```

A2.2 Parallel Hashing with Replacement

Algorithm A3 gives the pseudocode of parallel hashing with replacement. To increase the occupancy factor, multiple hash functions are also used. As no synchronization operation is required, we no longer need multiple hash table instances. The input H is the hash table and T is the list of nodes to be inserted. The procedure HASH-WITH-REPLACEMENT-DEDUPLICATE does two jobs. First, it inserts all the nodes from T to H. Second, it detects the duplication and returns a new list T' after removing the redundant nodes in T.

Lines 2-10 scan for an empty bucket for inserting $T[i]$. Lines 11-12 use an atomic swap operation to assign the current node to the hash table. If there are duplicated nodes t_0 in T,

Algorithm A3 Parallel hashing with replacement on a GPU

```
1:  procedure HASH-WITH-REPLACEMENT-DEDUPLICATE(H, T)
2:      T' ← T
3:      for i ← 0 to |T| in parallel do
4:          z ← 0
5:          for j ← 0 to d − 1 do
6:              if H[h_j(T[i])] ∈ {T[i], nil} then
7:                  z ← j
8:                  break
9:              end if
10:         end for
11:         t ← T[i]
12:         ATOMIC-SWAP(t, H[h_z(T[i])])
13:         if t = T[i] then
14:             remove T[i] from T'
15:             continue
16:         end if
17:         for j ← 0 to d − 1 do
18:             if j ≠ z and H[h_j(T[i])] = T[i] then
19:                 remove T[i] from T'
20:                 break
21:             end if
22:         end for
23:     end for
24:     return T'
25: end procedure
```

the atomic operation here can guarantee to deduplicate them if the hash values of other nodes in T are not in collision with that of t_0 (see Theorem A5.1). Finally, Lines 13-23 examine the hash table to do the actual node deduplication task.

A3 Analysis of GA*

A3.1 Correctness of GA*

Here we re-state the pseudocode of GA* (as given in Algorithm A4 and then provide its theoretical analysis.

Lemma A3.1. *Let $h'(x)$ denote the optimal distance or cost from current state x to the target state. If the given heuristic function is* admissible*, i.e., $h(x) \leq h'(x)$ for each state x, there exists a state t' in one of the priority queues such that $f(t') \leq f(t)$ before executing Line 19 in Algorithm A4, where t is an optimal target state.*

Proof. Let $d(x, y)$ denote the optimal distance or cost from state x to state y. For each state t'

Algorithm A4 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 close 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 i ← 1 to |S| in parallel do
26:            s ← S[i]
27:            if s.node ∈ H and H[s.node].g < s.g then
28:                remove s from T
29:            end if
30:        end for
31:        T ← S
32:        for i ← 1 to |T| in parallel do
33:            t ← T[i]
34:            t.f ← f(t)
35:            Push t to one of priority queue
36:            H[t.node] ← t
37:        end for
38:    end while
39: end procedure
```

that is on an optimal path from s to t, we have

$$\begin{aligned} f(t') &= g(t') + h(t') \\ &\leq d(s,t') + h'(t') \\ &= d(s,t') + d(t',t) \\ &= d(s,t) \\ &= f(t). \end{aligned}$$

Thus it is sufficient to prove that there always exists a state t' in one of the priority queues along an optimal path from s to t. At the beginning, the state with starting node s satisfies such a condition. At any time, if Line 12 in Algorithm A4 pops such a state t', Line 35 will generate another state that is also on an optimal path from s to t, which is then pushed back into the queues. Thus, such a state always exists in one of the priority queues. □

Theorem A3.2. *Let $h'(x)$ denote the optimal cost from current state x to the target current state. If the given heuristic function is* admissible*, i.e., $h(x) \leq h'(x)$ for each state x, the first solution returned by GA* must be the optimal solution.*

Proof. We prove this theorem by *contradiction*. There exists two possible situations that may violate our conclusion:

1. GA* never ends. In this case, suppose that the minimum cost of an edge is δ, then the number of steps from the starting node to the current node with minimum f value in the open list is never greater than $\lceil f(t)/\delta \rceil$, where t is the target state. This is because if such node t_0 exists, i.e., $f(t_0) > f(t)$, and according to Lemma A3.1, there exists a state with smaller f value. Let A be the set of all the states that are reachable from the starting state within $f(t)/\delta$ steps. In each expansion round, GA* will extract at least one state in A from one of the priority queues. However, $|A|$ must be finite because the number of paths from the starting state within $\lceil f(t)/\delta \rceil$ steps is finite. Thus this situation is impossible.

2. When GA* terminates, it returns a non-optimal solution. In this case, assume that our algorithm returns a state t_1, while the optimal solution is state t_2. Because t_1 is not optimal, we have $f(t_1) > f(t_2)$. However, according to Lemma A3.1, we have a state t' in one of the priority queues that satisfies $f(t') \leq f(t_2) < f(t_1)$, which violates the condition in Line 21 of Algorithm A4.

□

The reason why GA* can guarantee the optimality without the requirement of a consistent heuristic function is that GA* keeps the state that has a shorter path from the starting node to the current node during the node deduplication process.

A3.2 Theoretical Bound on the Number of Nodes Expanded by GA*

Lemma A3.3. *In the same expansion round of sequential A* (Line 11 of Algorithm A1) and GA* (Line 19 of Algorithm A4) on the same input, the state expanded by A* will also be expanded in the same round or has already been expanded by GA*, provided that the heuristic function is* consistent.

Proof. We prove this lemma by contradiction. Let r be the first expansion round that we have such a situation. Suppose that the state s is expanded in the sequential A* algorithm but not in the GA* algorithm. There are two possibilities why s is not expanded in the GA* algorithm:

1. The state s is not in the open list by the GA* algorithm. Let the parent of s be s'. Because r is the first round of this situation, s' must have already been expanded by the GA* algorithm. As s is a child of s', it must be in the open list;

2. The state s is in the open list, but has not been extracted by the GA* algorithm. Then, there must exist another node s'' in the open list of the GA* algorithm such that $f(s'') < f(s)$. Let us check the expansion situation of s'' in the sequential A* algorithm. If s'' has already been expanded in the sequential A* algorithm, s'' must also have been expanded in the GA* algorithm, because r is the first round that we have such a situation. If s'' has not been expanded but exists in the open list of the A* algorithm, the sequential A* algorithm will expand s'' rather than s in this round. If s'' does not exist in the open list of the sequential A* algorithm, then $f(s'') < f(s)$, which violates the property of consistency that the f values of the nodes extracted from the priority queue are monotone increasing.

□

Theorem A3.4. *The number of expansion rounds of expansion in GA* is less than or equal to that in the sequential A* algorithm on the same input, provided that the heuristic function is* consistent.

Proof. Suppose the sequential A* algorithm terminates when expanding the target state t in round k. From Lemma A3.3, GA* algorithm will expand in the same round or have already expanded t in round k. □

Corollary A3.5. *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.

Proof. When the heuristic function is consistent, the sequential A* algorithm will terminate in N expansion rounds. According to Theorem A3.4, GA* will also terminate in less than or equal to N expansion rounds. In each expansion round, GA* expands at most k nodes from the open list. Thus, the total number of the nodes expanded by GA* is less than or equal to kN. □

A4 Failure Rate of Parallel Cuckoo Hashing

In this section, we derive the theoretical analysis on the failure rate of the parallel cuckoo hashing algorithm for the case $d = 2$, i.e., using only two hash functions.

Lemma A4.1 (Kutzelnigg 2006)**.** *The probability that a single cuckoo hashing instance with $n = (1-\varepsilon)m$ nodes and two hash tables of size m successes to construct, where $\varepsilon \in (0,1)$, is equal to*

$$1 - \frac{(2\varepsilon^2 - 5\varepsilon + 5)(1-\varepsilon)^3}{12(2-\varepsilon)^2\varepsilon^3}\frac{1}{m} + O\left(\frac{1}{m^2}\right). \tag{1}$$

The proof of this lemma was first given by Kutzelnigg (2006), which was derived using generating functions and other advanced mathematic tools.

Theorem A4.2. *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*

$$c(\varepsilon)\frac{k^2}{m} + O\left(\frac{1}{m^2}\right), \tag{2}$$

where $\varepsilon \in (0,1)$ and $c(\varepsilon) = \frac{(2\varepsilon^2-5\varepsilon+5)(1-\varepsilon)^3}{12(2-\varepsilon)^2\varepsilon^3}$

Proof. Let $p(m,\varepsilon)$ be the success probability of a single cuckoo hashing instance with $n = (1-\varepsilon)m$ nodes and two hash tables of size m. From Lemma A4.1, we know that $p(m,\varepsilon) = 1 - \frac{c(\varepsilon)}{m} + O(\frac{1}{m^2})$. Then the success rate of all cuckoo hashing instances in the parallel cuckoo hashing algorithm is

$$\begin{aligned}
&\left[p\left(\frac{m}{k},\varepsilon\right)\right]^k \\
=\ &\left(1 - \frac{c(\varepsilon)k}{m} + O\left(\frac{1}{m^2}\right)\right)^k \\
=\ &\left(1 - \frac{c(\varepsilon)k}{m}\right)^k + O\left(\frac{1}{m^2}\right) \\
=\ &\binom{k}{0} - \binom{k}{1}\frac{c(\varepsilon)k}{m} + \cdots + O\left(\frac{1}{m^2}\right) \\
=\ &1 - c(\varepsilon)\frac{k^2}{m} + O\left(\frac{1}{m^2}\right).
\end{aligned}$$

Thus, the failure rate of the parallel cuckoo hashing is equal to $c(\varepsilon)\frac{k^2}{m} + O\left(\frac{1}{m^2}\right)$. □

A5 Analysis of Parallel Hashing with Replacement

In the Analysis Section of the main paper, we have stated Theorem 4.4 about the behaviors of parallel hashing with replacement. Here we first re-describe an alternative statement (i.e., Theorem A5.1) and then provide its proof.

Theorem A5.1. *If the parameter T of Algorithm A3 contains duplicated nodes, namely there exists $i_1 < i_2 < \cdots < i_n$ such that $T[i_1] = T[i_2] = \cdots = T[i_n] = t_0$, then the return value T' will contain at most one instance of t_0 if there does not exist a number $z \in \{0, 1, \ldots, d-1\}$ and a node $t_1 = T[j]$ such that $h_z(t_0) = h_z(t_1)$.*

Proof. Because there does not exist a number z and $t' = T[j]$ such that $h_z(t) = h_z(t')$, we can just focus on the threads dealing with $T[i_1], T[i_2], \ldots, T[i_n]$ without considering the behaviors of other threads. We label these threads as thread 1, thread 2, ..., and thread n, respectively.

We first consider the situation in which t_0 is in H before HASH-WITH-REPLACEMENT-DEDUPLICATE is called. Threads 1 to n will find this t_0 in Line 6. In that case, Line 12 will not modify the hash table and all the t_0 nodes will be removed from T'.

Now let us consider the second situation in which t_0 is not in H before HASH-WITH-REPLACEMENT-DEDUPLICATE is called. We will discuss the following different cases:

1. If any bucket for t_0 is still empty, the first thread that execute Line 12 will insert it into an empty bucket $h_u[t_0]$. For this thread, $t =$nil and its t_0 will be kept in T'. For all other threads, when they check the bucket $h_u[t_0]$, it is either filled with t_0 or empty. In either case, Line 7 will set $z \leftarrow u$. Because they execute Line 12 later than the first thread, after Line 12, t will be set to t_0 and Line 14 will be executed. So there will only one t_0 left in T'.

2. If none of the buckets for t_0 is empty, Line 7 will never be executed in these threads. So z is equal to zero in Line 12. Then the first thread that execute Line 12 will insert t_0 into bucket $h_0[t_0]$ and keep its own t_0 in T', while the other threads will remove their t_0 from T'.

□

References

Kutzelnigg, R. (2006). Bipartite Random Graphs and Cuckoo Hashing. *DMTCS Proceedings*.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

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

Sintorn, E. and Assarsson, U. (2008). Fast Parallel GPU-Sorting Using a Hybrid Algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388.