A Study on GPU-Accelerated Parallel $A^\star$ Search

Zhang, Zhenyuan
Wang, Yiqun

December 2021

1 Introduction

$A^\star$ search is a widely-used graph traversal and path search algorithm. It can be categorized as a kind of best-first-search with heuristic and memory. The algorithm maintains an open list and a close list. The open list contains all the states that are to be explored, and the close list contains the minimal costs of all states that have already been explored. In each iteration, the algorithm

1. Extracts the best state from its open list based on its value, i.e., the current cost plus the estimated cost-to-go;
2. Expands the picked state and gets a list of expanded states;
3. Eliminates the expanded state if it has already been visited and its values is less than what recorded in the close list, or keeps it and updates the close list;
4. Add the remaining expanded states to the open list.

$A^\star$ search is commonly used on AIs that solving puzzles and playing games. Both applications can take the advantage of parallelism: solving puzzles usually involve exploring a potentially exponential amount of states, and in a video game the machine may need to plan actions (or find paths) for thousands but not hundreds agents within the duration of several frames.

Modern GPUs can be programmed to process hundreds and thousands of data at the same time, making them good candidates to provide a vast speedup on $A^\star$ search. But until now it is not common to see games with $A^\star$ search implemented on a GPU. The main reason is that the traditional $A^\star$ algorithm is not trivially parallel (since picking and pushing into the open list is a serial operation usually implemented by priority queues). The only parts that can be trivially paralleled are the expansion of states and the evaluation of values. However, in applications like solving puzzles, the number of expanded states in each iteration is usually limited by some constant, which limits the ability to parallel the algorithm.

In 2015, Zhou and Zeng [2] proposed a framework for massively paralleling $A^\star$ search on GPUs. In each iteration, instead of using one open list, the new algorithm maintains thousands of open lists at the same time, and re-distribute the expanded states to all the open lists at the end of the iteration. In such a way we can expand and explore far more states in one iteration thanks to GPUs' highly paralleled architecture. In this project we generally follow the framework proposed in [2], with a more detailed focus on the impact of different implementation choices regarding to memory layout and data structures used.

Algorithm 1: $GA^\star$: Parallel $A^\star$ Search on a GPU

Input : Starting state s, Target node t, Number of parallel open lists k
Output: Shortest path from s to t

Let $\{Q_i\}_{i=1}^{k}$ be the k priority queues of open list
Let H be the hash table of close list
$\texttt{Push}(Q_1, s)$
$m \leftarrow \mathsf{None}$
while Q *not empty* **do**
  Let S be the list to store expanded states
  Let M be the list to hold candidate target states
  `// Extract and expand the current best states`
  for $i \leftarrow 1, \cdots, k$ *in parallel* **do**
    if $q_i \leftarrow \texttt{Pop}(Q_i) \neq \mathsf{None}$ **then**
      if $q_i.node = t$ **then**
        $M \leftarrow M + q_i$
      else
        $S \leftarrow S + \texttt{Expand}(q_i)$
      end if
    end if
  end for
  `// Determine finish conditions`
  $m \leftarrow \arg\min_{m' \in M} f(m')$
  if $m \neq \mathsf{None}$ *and* $f(m) \leq \min_{q \in Q} f(q)$ **then**
    return *path generated from* m
  end if
  `// Remove duplication`
  $T \leftarrow S$
  for $s' \in S$ *in parallel* **do**
    $h \leftarrow H[s'.node]$
    if $h \neq \mathsf{None}$ *and* $h.g < s'.g$ **then**
      Remove s' from T
    end if
  end for
  `// Re-insert expanded states`
  for $t' \in T$ *in parallel* **do**
    `// Compute values based on the heuristic function` h
    $t'.f \leftarrow t'.g + h(t', t)$
    $H[t'.node] \leftarrow t'$
    Distribute t' to one of Q
  end for
end while

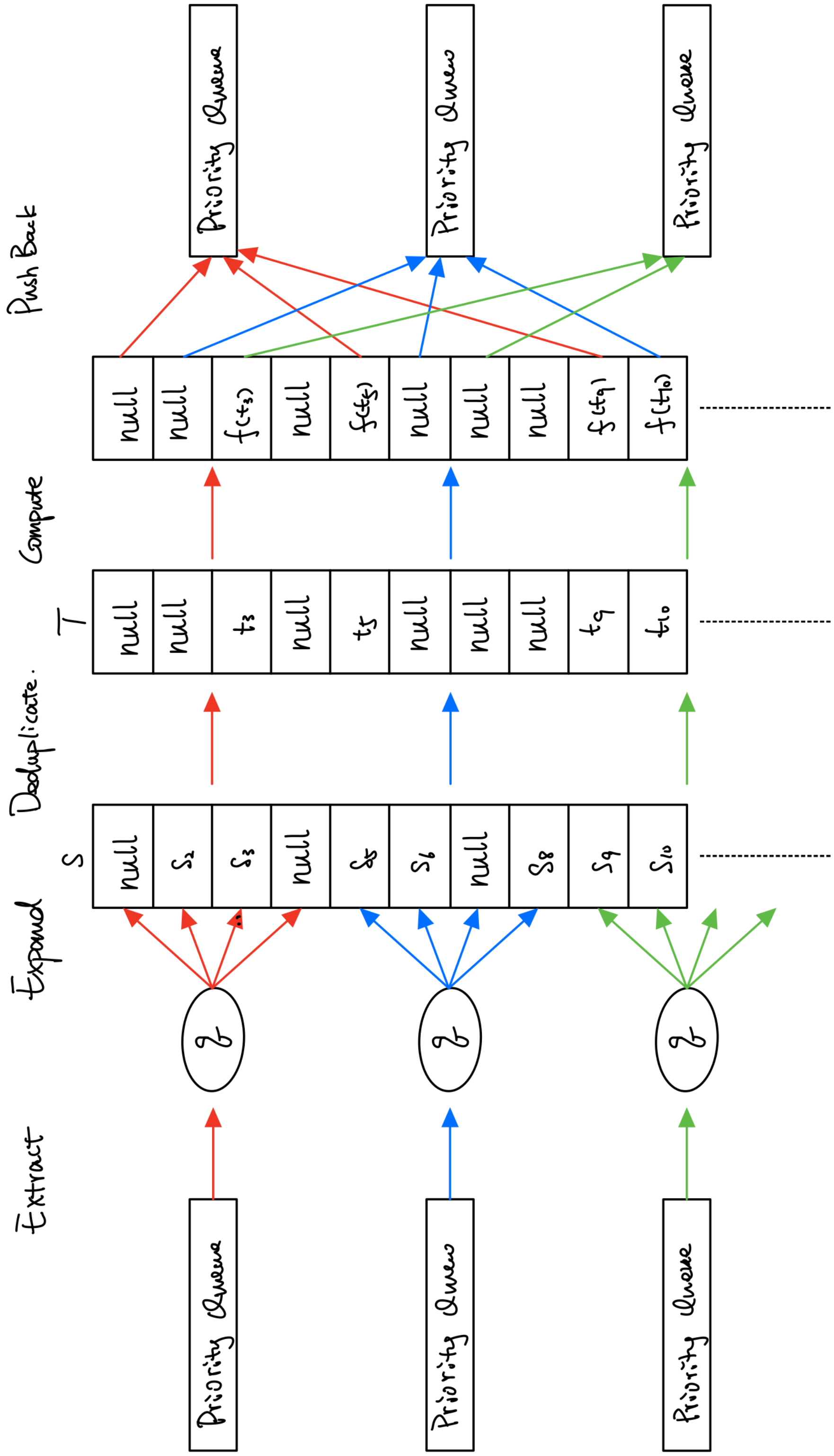

Figure 1: Data flow chart of $GA^{\star}$

2 Method

Algorithm 1 summarizes the main algorithm $GA^\star$ proposed in [2]. There are 5 stages in each iteration: *extraction, expansion, deduplicate, value computation and re-insertion.* In each of the stage we process in a data parallelism manner.

Figure 1 is the data flow diagram of algorithm 1. It also shows the memory layout of S and T lists and distributing strategy we used in our implementation. Details about the design choices are mentioned later this section.

Our implementation of algorithm 1 consists of 4 parallel functions and 1 data transfer operation:

1. *Extract/Expand*: Line 8-17.

   Heap extractions and expansion calculations.

2. *Verify Solution*: Line 19-22.

   Parallel scans [1] that find $\arg\min_{m' \in M} f(m')$ and $\min_{q \in Q} f(q)$.

3. *Copy*: Line 20-22.

   Transfer solution back to host to indicate the end of search.

4. *Remove Duplication*: Line 23-30.

   Hashtable lookup.

5. *Re-insert*: Line 31-37.

   Heuristic calculations, hashtable insertions and heap insertions.

2.1 Extraction

The extraction stage consists of extracting the current best states from priority queues. Though it is hard and inefficient to parallelize insertions and queries to one priority queue on a GPU, it is possible that multiple threads operating on their own priority queues without interfering each other. Another thing to notice is that priority queues can be efficiently implemented by binary heaps, which can be made of a chunk of continuous memory. Continuous memory layout is friendly for GPUs, because as we will see, fractional heap memory allocations on GPUs are extremely slow and unsafe.

2.2 Expansion

In the expansion stage, rules from specific problems are used to generate successive states from the extracted states. Expansion of an extracted state can either be done by the same thread that did the extraction, or be assigned to new individual threads. We are not expecting to see significant performance differences between the two when the computational resources are saturated.

The first question we encountered is about how to manage the resources of the newly created states. The first approach we tried is similarly to what we would do in `c++` programming: using heap-allocated memory with atomically reference-counted (ARC) smart pointers. In regular `c++` programming, RAII technique is used to ensure that the resource is freed at the right moment. But later we found that this didn't work for GPU because shared buffers we used are fundamentally unsafe in RAII sense. Moreover, as experiment results turned out, using heap allocation extremely slowed down the execution, and it usually failed to allocate the requested memory.

The solution we come out is to manually implement a memory pool with reference counting. It consists of pre-allocated continuous chunks of memory for data and reference counters, and a header pointer. In each allocation call, the head pointer atomically advances (and circulates if going beyond the end)

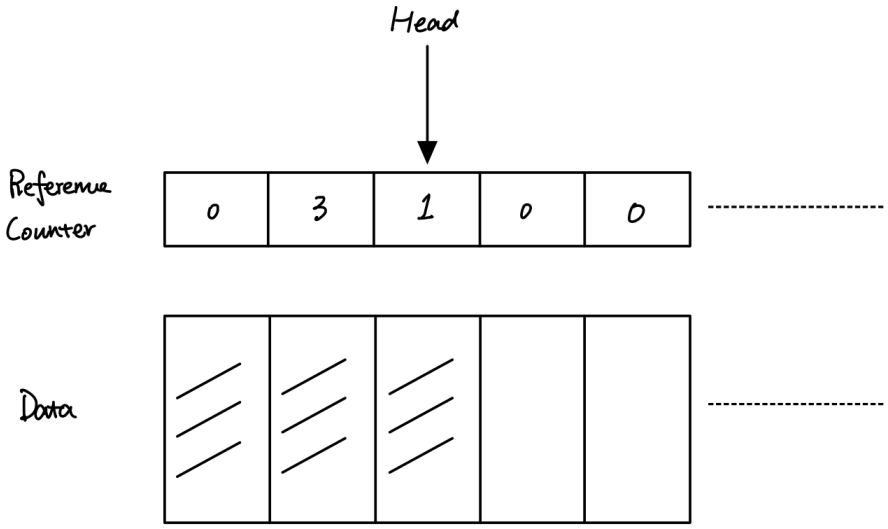

Figure 2: Illustration of a GPU memory pool

until a slot with zero reference count is met and returns the pointer to the data slot. A sufficiently large pool speeds up memory allocation for over 1000 times comparing to heap allocation.

Another design choice is the data structure for S and T lists which store pointers to the expanded states. The conceptually best way is to use an append/consume buffer: a ring buffer with an atomically advancing append head and a consume head. As this approach stores data compactly, it is useful in the cases where memory is limited or the number of expansions are unknown. It also enables pipelining: the expansion stage and the subsequent stages runs in parallel, while the expansion stage constantly feeds the downstream with data.

However, we end up using a sparse array as shown in figure 1 instead because we think that append/consume buffer is too complicated for our use case (i.e., puzzle solving and path finding) where the maximum expansion number is known and limited, and atomic operations may also lower the performance. The sparse array allows us to do insertion and removal in a embarrassingly parallel manner. Moreover, the fact that the expanded states are distributed evenly inside the array saves us a great amount of work re-distributing the states in the last stage.

2.3 Hashtable Implementation

$A^\star$ algorithm avoids re-exploration the same node by keeping tracking of costs of the states it visited in the close list. It won't expand a state the second time unless there is a shorter path than what's in its memory. A hashtable is the most suitable data structure for the close list on GPUs because of its simplicity and efficiency. It generally takes $O(1)$ time to lookup and insert into a hashtable, and what's better, it uses continuous memory. On GPUs, we can compute hash functions in parallel, but what about lookup and insertion operations?

Fortunately, in the duplication removal stage the close list is read-only, so we can implement the lookup functionality in a lock-free manner. Nevertheless, we do need locks for the insertion operation. For each slot we set a spin lock which protects the data from being read or written by other threads during the insertion process.

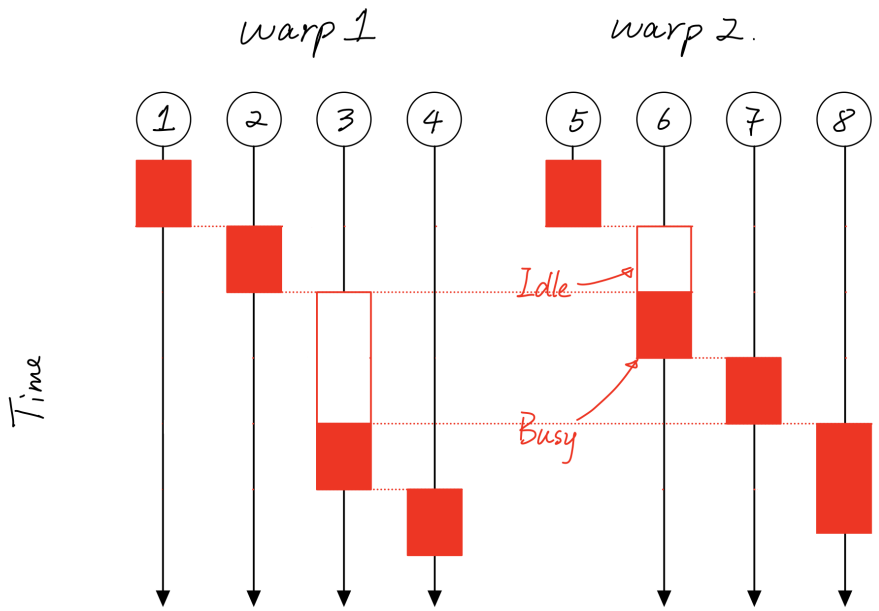

Figure 3: An illustration of how we resolve race conditions when inserting into a hashtable. Threads 1-4 and 5-8 are two wraps; Threads 2/6 and 3/7 acquire the same locks. Note how threads within a wrap are executed in an ordered manner.

However, there is a pitfall in this implementation. GPUs threads in a wrap are executed in a step lock. If two threads in the same wrap are waiting on the same lock, none of the threads in this wrap can proceed, which causes a deadlock. The first solution we came up with was to hash threads in one wrap to different slots. This turned out not solving the deadlock fundamentally. Since the execution order of threads in a wrap is undetermined when a branch divergence has been encountered, it's still possible that two wraps with two pairs of inter-blocking threads block on each other. Eventually we take the safest yet slightly slow approach: we enforce the insertion orders of threads in the same wrap.

Hash collisions are expected to occur. [2] proposed that in memory-limited cases, one can directly replace the collided slot with new data. This may cause re-expansion of visited states, but in practise, we find that as long as the states are hashed evenly, there is no performance differences between using a significantly large hashtable and an overwhelmingly large one.

2.4 Re-distribution of Expanded States

The last stage of the algorithm is to re-distribute the expanded states into the open lists. To avoid distributing all states to one open list throughout all iterations, we let adjacent lists read adjacent slots in the T list, with a stride of k, as shown in figure 1. As mentioned above, we can assume that the states are evenly distributed in the T sparse array, so this re-distribution pattern should keep the size of open lists mostly the same.

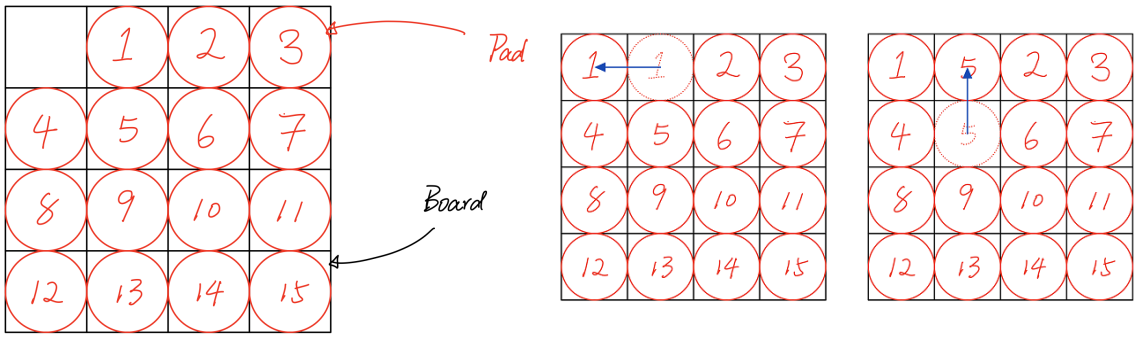

Figure 4: Sliding pad puzzle

3 Results

To see how GPUs are suitable for such kind of work, we test our implementation with a mini sliding pad puzzle.

The puzzle has 15 marked pads on a 4-by-4 board. Apparently there is always an empty slot on the board if we fill 16 slots with 15 pads. In each step, we move one pad adjacent to the empty slot, and repeat until the desired layout is achieved. We are interested to find the minimum number of steps possible for a desired layout.

In the layout illustrated in figure 4, 15 pads are placed row-wisely, leaving the top-left slot empty. We can fill the empty slot either by moving Pad 1 left or moving Pad 4 up. If we choose to move Pad 1 left first, then move Pad 5 up (to fill the new empty slot), we can get a new layout of the pads.

Each unique layout of the pads on board can be regarded as a unique state. If we consider all the layouts that can be achieved from the starting state, there are $16!/2 \approx 1.05 \times 10^{13}$ states. We expect the number of states needed to be explored to grow approximately exponentially to the number of steps to the desired layout. We define a solution to the desired layout to be the series of layouts from the start to the end. Accordingly, the solution length is the number of steps.

3.1 General Search Performance

For comparison, we also implement a single-threaded CPU version of $A^\star$ algorithm. We conduct experiment on 3 devices: Intel® Core™ i7-8700, NVIDIA® Quadro RTX™ 4000 and NVIDIA® GeForce GTX™ 960. We manually (but arbitrarily) create a problem set consists of desired layouts whose optimal solution lengths range from 24 to 32. In all experiments of this section, we fix the GPU number of parallel open lists to $k = 1024$.

We measure the total time (in milliseconds) and iterations it takes to compute the solution. Though not being directly measured, we assume that the total states explored are proportional to the number of iterations, with the factor k, which is the number of parallel open lists. We can thus derive several indices showing the speedup and efficiency of the algorithm:

1. *Explore Rate*: The number of states explored per millisecond. This value shows whether the parallelism of GPUs helps explore more of the problem space than CPUs do.
2. *Explore Efficiency*: The total number of states GPUs explored compared to its CPU counterpart. Though GPUs are able to explore more states, only a small part of it turns out to be useful.

The results are summarized in table 1.

To take a better look at the time and speedup, we plot the time on logarithm scale in figure 5. We can see that all three devices obey linear increment in the graph (exponential increase in time vs. solution length), but the time for GeForce increases faster as the solution length grows. The speedups of Quadro are range from 6 to 14, and the speedups of GeForce are from 3 to 10. Nevertheless, **all GPU implementations are faster than the CPU in real time**.

In terms of the explore rate, we find that GPUs can generally explore thousands of states within a millisecond while the CPU can only do about 50. However, the explore rates of GPUs drop significantly when that of the CPU slightly increases at the solution length grows. The main reason is that, as the exploration goes on, the size of heaps increases. Insertion and extraction operations on large heaps cause non-local memory access and GPUs are really not good at that.

We see a reversed trend on the explore efficiency, though. This is also expected, because the maximum number of states being explored in each iteration is fixed, as the exploration space grows, the algorithm focuses more on promising states.

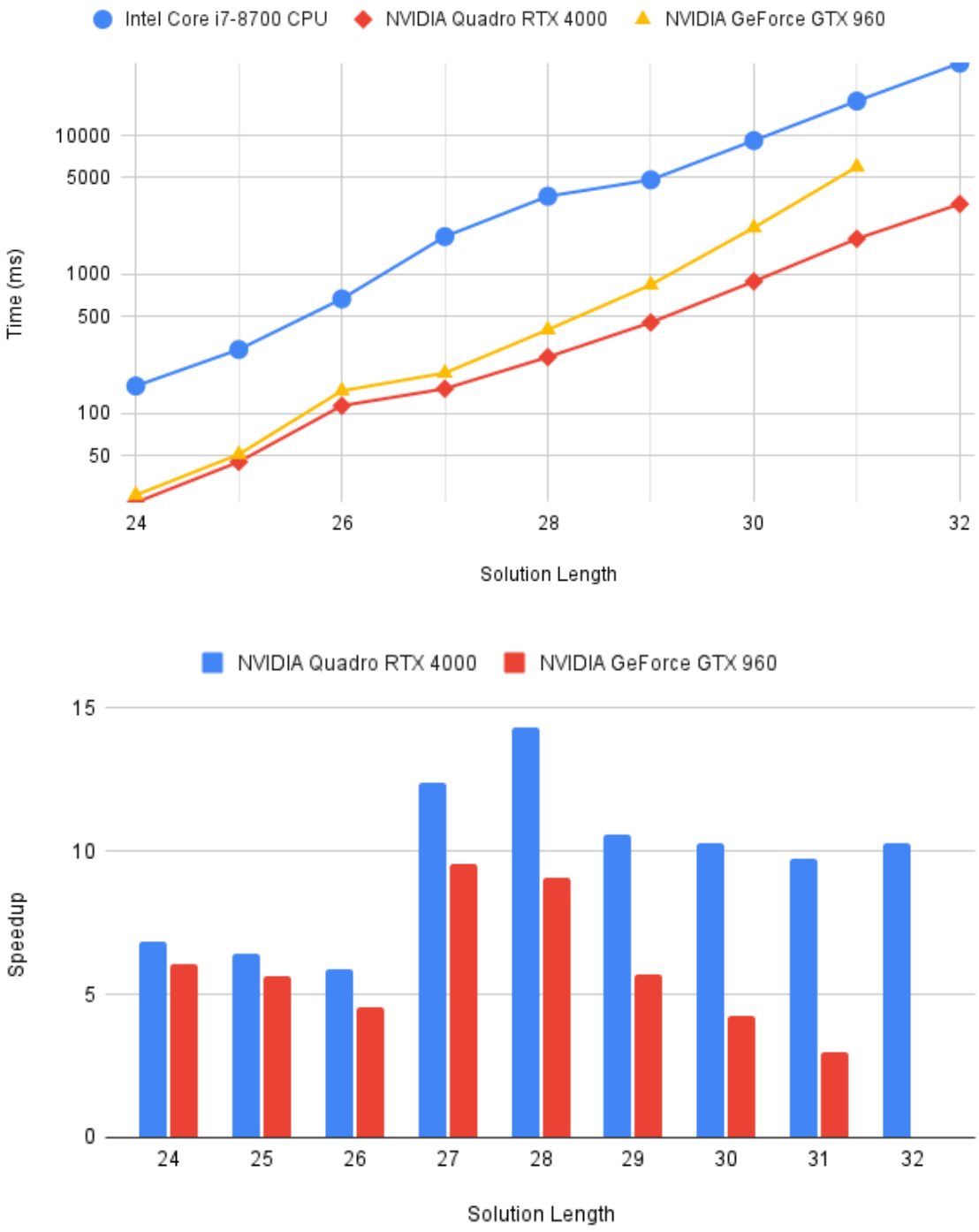

Figure 5: Time and speedup with different devices ($k = 1024$ for GPUs)

Solution Length	Iterations	Time	Explore Rate	Explore Efficiency
Intel® Core™ i7-8700 CPU @ 3.20GHz				
32	1716151	33026	51.96	
31	945720	17656	53.56	
30	497285	9174	54.21	
29	253819	4792	52.97	
28	183519	3650	50.28	
27	92713	1873	49.50	
26	30366	668	45.46	
25	14028	289	48.54	
24	7070	158	44.75	
NVIDIA® Quadro RTX™ 4000				
32	12473	3211	3,977.69	13.44%
31	7726	1807	4,378.21	11.95%
30	4369	892	5,015.53	11.12%
29	2310	452	5,233.27	10.73%
28	1385	255	5,561.73	12.94%
27	739	151	5,011.50	12.25%
26	557	114	5,003.23	5.32%
25	239	45	5,438.58	5.73%
24	119	23	5,298.09	5.80%
NVIDIA® GeForce GTX™ 960				
32	-	-	-	-
31	7717	5940	1,330.34	11.97%
30	4357	2170	2,056.02	11.15%
29	2313	844	2,806.29	10.72%
28	1371	401	3,501.01	13.07%
27	739	196	3,860.90	12.25%
26	557	146	3,906.63	5.32%
25	239	51	4,798.75	5.73%
24	119	26	4,686.77	5.80%

Table 1: Results of general performance experiment ($k = 1024$ for GPU)

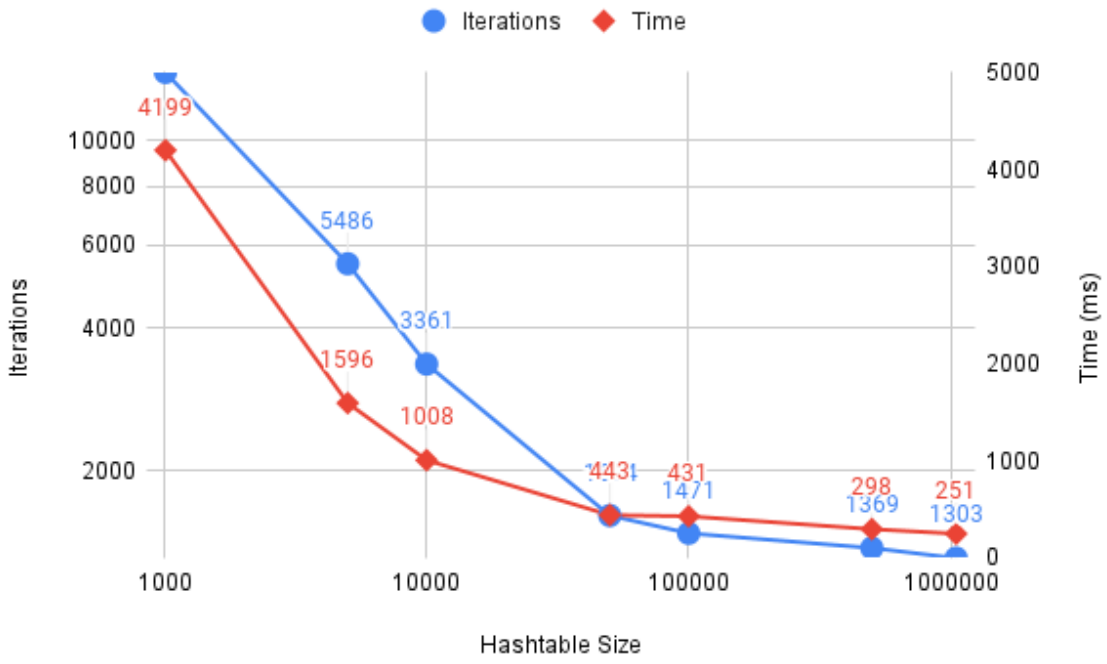

Figure 6: Impact on performance due to hash collisions (Quadro, $k = 1024$)

3.2 Impacts on Hash Collisions

The size of hashtable is limited by the memory size of the GPU. As explained in 2.3, the threads will compete for the lock and then override the old data with the new in the case of a collision. We postulate that this behavior influences the performance in two aspects:

1. *Number of Iterations*

   The override behavior harms our capability to reject defected states early in the process. The number of iterations needed to solve the puzzle can increase with potentially more states to explore.

2. *Time per Iteration*

   The threads don't get the lock remain idle in case of a collision till the lock is released, leaving other threads in the wrap waiting behind. Threads are synchronized globally at the end of each iteration, so the time spent on each iteration may grow with the frequency of collisions.

However, as indicated in the end of 2.3, when the hashtable is sufficiently large, the possibility of collision remains at an acceptable low level. Therefore, allocating space for a larger hashtable starts off to be essential for a better performance. After that it gradually becomes a costly way to improve the overall efficiency.

In this experiment, we fix the problem to be one with a solution length 28, and use hashtables of different sizes. The smaller the hashtable, the more frequently hash collisions. Figure 6 shows our results. We can see clearly that **both iterations and time increase like crazy when the hashtable size gets sufficiently small, but a large hashtable doesn't add too much to the performance**.

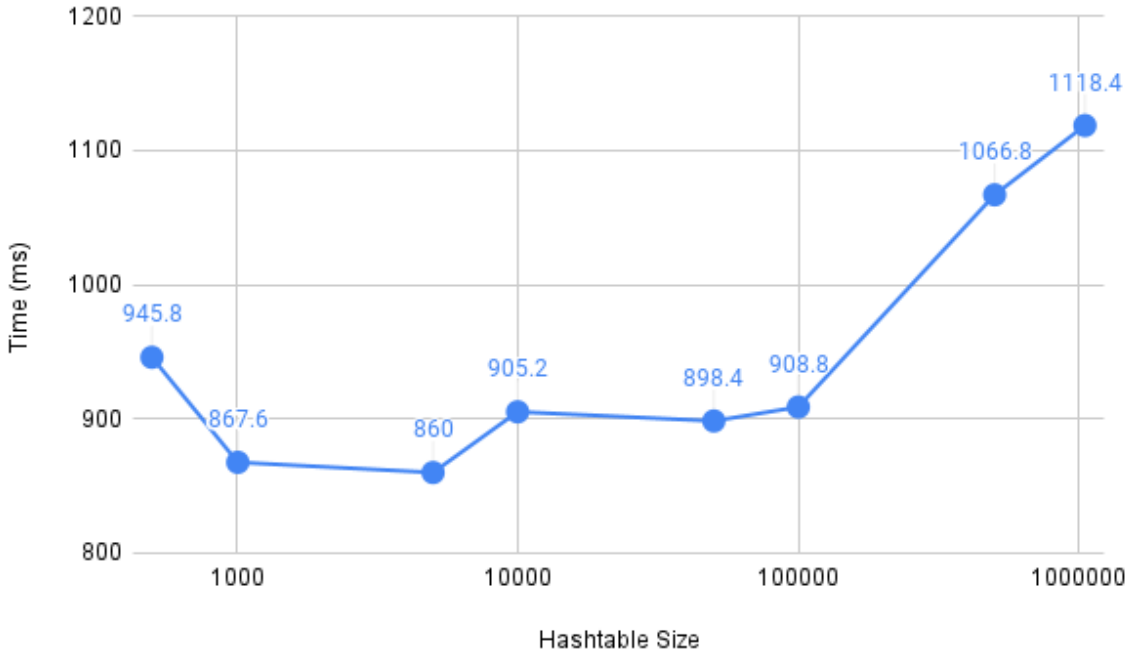

Figure 7: Results on inserting 67108864 states into hashtables of various sizes (Quadro, $k = 1024$). Smaller hashtables are generally faster.

To further study the influences of hashtable sizes, we solely test the efficiency with 67108864 insertions, which is the largest possible state number within 16384 iterations. The results are plotted in figure 7. To our surprise, **more frequent hash collisions themselves do not affect the performance of insertion**; instead, smaller hashtables are faster due to their memory locality.

3.3 Efficiency of Individual Components

Studying which operations are likely to be the bottleneck of the algorithm may lead to potential improvements. We postulate that there are two main slow operations:

1. *Transfer Data through PCI-e Bus*

   In each iteration we copy the flag indicating whether the search has ended to the host. It is slow because transferring data through the PCI-e bus is slow, and this causes the consequent tasks block until the previous calculations are done. But actually it is not necessary to do this that frequently. We can transfer it for, say, every 64 iterations with the cost of some additional iterations.

2. *Heap Insertion/Extraction*

   These operations involve a lot of branching and low memory locality. GPUs don't like that.

We measure time for each individual parallel function on a problem with solution length 28, a hashtable size of 1048573 and $k = 1024$ parallel open lists. We test under a set of distinct exit condition copy intervals (0, 64, 128, 512, 1024). The results are shown in figure 8.

We see that it takes a great amount time to transfer data through the PCI-e bus (see how big the

Figure 8: Time for operations under various exit condition copy interval

yellow chunks are when the copy interval is 0). This amount of time vanishes as soon as we begin to limit the frequency of data transfer. The reason why the overall time increases with the copy interval is the fact that more iterations are run. With this optimization, **we get an average improvement of** 35 % **when the solution length is greater than 26**.

Also notice that the most time-consuming operations for a GPU are *Extract/Expand* and *Re-insert*, which are both mainly heap operations. On the hand, on a CPU the time it takes for those operations are almost the same as *Remove Duplication*. This suggests that our previous insight is correct: **heaps are far more inefficient than hashtables on a GPU**.

4 Discussions

As demonstrated in section 3, the parallel $A^\star$ search algorithm on GPU outperforms the traditional $A^\star$ on the CPU. For further study, we consider reduce the time spent in the two heap-related stages: *Extract/Expand* and *Re-insert*. As shown in 8, these two stages dominate the time of a search. After we almost eliminated the copy time by introducing copy interval as described in 3.3, the two heap-related stages take more than 80 % of the total time. One possible way is transfer the heap operation to a more powerful CPU. Besides, we can make a pipeline by setting the devices to focus on certain stages for a more stable memory environment and simpler instructions within the device. It can be a pipeline of CPUs and GPUs, a manager-worker structure, or a combination of them. All these are promising directions for further study.

References

[1] W Daniel Hillis. “Steele Jr”. In: *GL: Data Parallel Algorithms. CACM* 29.12 (1986), pp. 1170–1183.

[2] Yichao Zhou and Jianyang Zeng. “Massively parallel $A^\star$ search on a GPU”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 29. 1. 2015.