EECS 587 Term Project
Solve Rubik's Cube with Multi-GPU

Zhitian Xu, Chenkai Shao

February 2, 2020

1 Problem Description

Rubik's Cube is a famous 3-D combination puzzle that is known to have an extremely large amount of possible states. Although Rubik's Cube may be used on different types of "cube" nowadays, such as $2 \times 2 \times 2$, $7 \times 7 \times 7$ and non-cubicle puzzles, it originally refers to the standard $3 \times 3 \times 3$ cube that was invented by Hungarian sculptor and professor of architecture Ernő Rubik back in 1974. A $3 \times 3 \times 3$ may reach approximately 4.3252×1019 different states if only the six sides are turned [1]. Although humans can solve Rubik's Cube fairly quickly now, it is nearly impossible for humans to be able to consistently find an optimal solution for any sufficiently scrambled cube. There exists programs that solve Rubik's Cube optimally but none of them utilizes parallelization approaches or GPU capabilities to our knowledge. In our project, we abstract and represent the $3 \times 3 \times 3$ Rubik's Cube in an efficient way and try to find the optimal solution of randomly scrambled cubes on multiple GPUs.

2 Rubik's Cube Notion and Structure

A standard $3 \times 3 \times 3$ cube consists of six centerpieces, eight corner pieces and twelve edges pieces. The six centerpieces are fixed on a six-direction axis and thus their relative positions are fixed. Since any turn of the side faces does not affect the position of the centerpieces, we only need to consider solving the corner and edge pieces, which are called "cubies," into their corresponding position with the correct orientation. The optimal solution is commonly defined as the least number of 90 or 180 turns required to solve a cube from a scrambled state. Under this particular definition, the maximum number of turns in an optimal solution of any scrambled cube is proven to be 20, which is known as the God's Number [2].

On a $3 \times 3 \times 3$ cube, each corner cubie has 8 possible positions and 3 possible orientations at each position, resulting in $3 \times 8 = 24$ possible states; each edge cubie has 12 possible positions and 2 possible orientations at each position, resulting in $2 \times 12 = 24$ possible states. One subtlety worth mentioning is that by nature, the position AND orientation of the last corner is determined by the rest of the corners and the same applies to edges. This rule significantly reduces the problem size and we need to exploit it in our solver implementation.

3 Cube State Representation

We assign a number to each sticker, or facet, on the corner and edge cubies, illustrated in Fig.1 below. A cube is represented with an array of 20 numbers, each representing the facet number of a position on the cube indexed in the way shown in Fig.2. For example, the first element of the array represents the color of the facet at the position labeled with 0 in Fig.2. If it is the green facet of the orange-white-green cubie, the value of the first element in the array would be 5. The number assignment in Fig.1 is arranged in a way that, for corners,

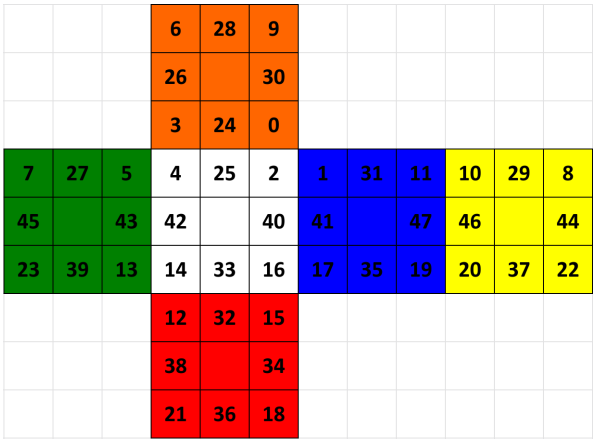

Figure 1: Facet representation

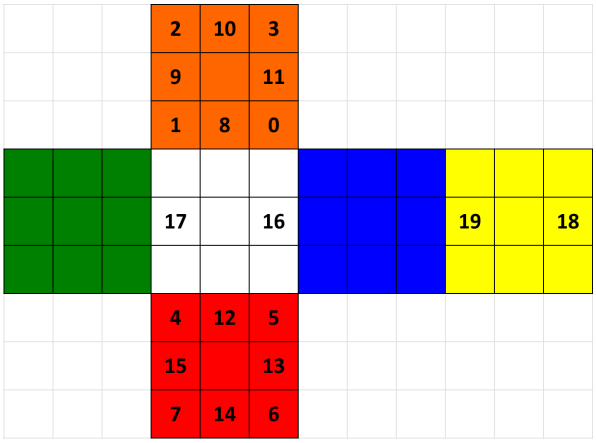

Figure 2: Cube array index

the number divided by 3 can represent the position of the cubie and that modulo 3 can represent the orientation. For edges, the position and orientation can be induced by dividing and moduloing 2. This approach significantly simplifies calculations in our algorithm and reduces the constant lookup tables we need to store in the constant memory on the GPUs.

4 Algorithm

Finding a solution or an optimal solution for a scrambled cube is essentially a search problem. The entire space of all the possible cube states can be considered as an enormous graph with each state as a node and each cube turn as an edge. Therefore, an optimal solution becomes the shortest distance between the node of the scrambled state and that of the solved state, only one of which exists.

We choose to use the Iterative Deepening A* algorithm (IDA*) to solve this search problem. A* search is a depth first search based on heuristic. We use the iterative deepening technique to ensure that the solution we find first is actually the optimal solution.

Due to the sheer amount of nodes in the entire graph, we need to use heuristics for branch pruning and load balancing. The heuristic should return an estimation of the number of turns needed to solve a cube from a given state. In addition, the heuristic has to be admissible, meaning that it needs to represent a lower bound of the number of turns. In order to prune search branches, we need to compare this lower bound to either a running upper bound of the nodes already searched or a globally known upper bound. Since there is no easy way

to derive an upper bound of the optimal solution for a given cube state, we can only use the God's Number 20 initially and any solution, optimal or not, found on the GPU as a running upper bound. With this approach, we can safely prune a search branch if the sum of 1) number of turns already used to get to a state and 2) number of future turns that the heuristic returns is larger than the running upper bound.

5 Limitation of Our Work

As suggested in [1], the number of nodes in the search graph with IDA* increases with a branching factor of about with a branching factor of about 13.34847 as the searching depth increases. For example, depth 4 has 43,254 nodes and depth 5 has 577,368, which roughly 43154 times the branching factor. Therefore, for cubes with an optimal solution close to the God's Number 20, the search space increases dramatically. This suggests that good heuristics are needed for pruning branches at run time. As discussed before, the upper bound we can use to compare with our heuristic is fixed to the depth of the current iteration. Thus, the search speed is largely determined by the quality of the heuristic, with heuristics with higher expected value resulting in more branches getting pruned.

In [1], two types of heuristics are discussed - Manhattan distance and pattern database. The Manhattan distance heuristic simply calculates the average Manhattan distance of all corners or edges to their solved facet. The expected value of the Manhattan distance of the edges is 5.5. We only need a 2-D array of size 12×24 as a lookup table for the Manhattan distance between facets. The pattern database heuristic, on the other hand, builds a huge database of the exact number of minimum turns required for each state considering a subset of the cube. For example, a 1-D array database can be built with the length of the optimal solution for all possible states of only the corners combined. This will result in a lookup table of 42 megabytes [1].

We did not choose to use the database heuristic in our project for the following reasons. First, the database heuristic essentially pre-computes a subset of the problem, which can be considered cheating in terms of pure search of the optimal solution. Second, this 42 megabytes of database can only be stored in the global memory instead of the constant memory resulting in slowing access to it. Because any entry in the database may be accessed by all the threads processing nodes on the GPU, there is no way of pre-loading a certain portion of the database in shared memory and cache would not help either. In addition, the time required to calculate the index into the database from the state of a cube is significantly larger than simply looking up the Manhattan distance table, which can be pre-loaded into the shared memory. Third, it would take a fair bit amount of time to implement the code that could generate the database and verify that the order of the entries are correct. We can not use tables generated by other programs because they use different cube representation methods from ours.

Therefore, we use the average Manhattan distance of all edges as our heuristic, which is not able to prune too many branches. Our serial version can only run up to a depth of 11 within a reasonable amount of time, which takes 42 seconds. In order to compare the speedup of using the GPU, we limit the max depth to a number smaller than the God's Number by only using cubes scrambled with no more than a certain number of turns. We

also use the number of scramble turns as the way to scale the problem size.

6 Utilizing GPU

For cube state transformation and Manhattan distance heuristic calculation, we need to use several constant tables of size no larger than 400 bytes each. The total size of constant tables we use can easily fit in the 64 KB constant memory on the GPU and in fact the 8 KB cache working set per multiprocessor for constant memory is enough to store our tables, further reducing the time spent fetching the table values.

We experimented with various grid sizes and block sizes on up to two GPUs of a single node on Great Lakes. In each run, the program first generates a single scrambled cube with a given number of turns, which determines the problem size as discussed above. After that, the CPU starts running the IDA* algorithm from the single node that represents the state of the scrambled cube. It expands the search tree until there are leaf nodes of number equal to the total number of threads used in this run. The CPU then dispatches the nodes to the GPU(s) so that each thread starts with one node. We did not balance the workload between GPUs, if two are used, or between threads before dispatching for the reasons that 1) the Manhattan distance heuristic is not a good indicator to rank the approximate number of turns needed when the IDA* search depth is small and 2) work balancing between two GPUs does not have much advantage over randomly splitting the nodes in half given the large number of nodes in total.

After each thread gets its initial node, it starts to expand the search tree with IDA* algorithm until it has found a solution or pruned all the nodes. However, we do not let the threads keep running on their own until they finish. Instead, a thread would temporarily stop searching if it has explored a certain amount of nodes, which is a parameter we change in our experiment, and has not found a solution. The number of nodes explored before checking the global variable functions as the interval between periodic checks in the threads. In this case, the thread would check a variable in the global memory which represents the length of the solution already found in other threads. This global variable is updated with an atomic CUDA operation whenever a thread finds a solution.

For a single GPU, due to the fact that the first solution found IDA* is always the local optimal solution within a thread, we can use this variable as a way of communicating the discovery of a solution. It is also considered as the run-time upper bound used for pruning the branches of other threads. For two GPUs, this global variable provides more benefits. Since the problem space is partitioned to two GPUs, a single GPU does not know if it has the global optimal solution. Without frequent communication between them, this variable in each GPU's global memory stores the length of local optimal solution. If, for example, GPU 2 does not get the initial node that could lead to the global optimal solution, it would keep searching beyond the depth of the global optimal solution. In this case, a local upper bound, other than the God's number, that can be used to prune the nodes on GPU 2 becomes crucial. Considering the benefit of communication between the GPUs, we copy the length of the local optimal solution to another GPU whenever it is updated on one GPU. This does not incur too much overhead because this local upper bound is not updated frequently.

An approach that we have considered but did not adopt is work stealing between threads.

If we want to transfer work between threads, there needs to be memory reserved for a work pool either in shared memory or global memory, depending on if the pool is shared within blocks or across the whole GPU. There are a couple of problems with a work pool. First of all, on an NVIDIA GV100 GPU with Volta architecture, we can run maximum 163,840 threads (64 warps per SM $\times$ 32 threads per warp $\times$ 80 SMs per GPU) concurrently. The number of nodes that may be expanded on a single thread is in the order of 18 ^ (number of turns used in scramble - number of turns run on the CPU). Keeping such a huge work pool in the memory is not practical. Some clever optimizations, such as only pushing the nodes with a heuristic above a certain threshold to the poll, could be done, but our Manhattan distance heuristic is not expressive enough and the benefit of doing so will be very limited. Furthermore, the work pool would need to be frequently updated by all the threads, which requires the use of lock on memory. Mutually exclusive access on memory is not easy to implement on GPUs and the introduced overhead and time blocked in threads can be quite large. Lastly, in order to ensure complete balance, the workload on different threads needs to be sorted by number of nodes and the potential future load of each node expressed by their heuristic. Again, the overhead of maintaining such a priority queue surpasses the benefit that we may get from it.

7 Evaluation

Before explaining the evaluation results, we need to introduce some symbols.
We use *num of gpu* to represent the number of GPUs we used on the board, *grid size* to represent the GPU kernel dimension (assuming one dimension), *block size* to represent the block size (assume one dimension), *search depth* to represent the number of turns used to scramble the cube.

It is clear that *search depth* represents the upper bound of the solution, but not the optimal solution because the scramble is not guaranteed to be reverse of the optimal solution. Therefore, it is possible that we randomly turn the cube 20 times but the optimal solution takes only 3 steps to solve the cube, though the probability of such an extreme case is very low. When we run the experiments, it is possible that we get different execution time with the same parameters. In all the experiments listed below, we assume that *search depth* represents the optimal solution, which means we run the experiments several times to pick the worst case, where the length of the optimal solution is close to the number of scramble turns.

Besides, we use *initial upper bound of solution* to represent the estimated upper bound for each GPU. We do not need this if we only have 1 GPU, because this GPU will do all the work. If we have multiple GPUs on a board, and each GPU handles part of the problem space, it is possible that GPU 1 finds that optimal solution is 10 while GPU2 finds that optimal solution is 20. As discussed in the GPU section, the communication of local optimal solutions between GPUs helps improve the performance. Back to our example, if we set the estimated upper bound to 12, then GPU 1 will find the optimal solution 10 and quickly finish its job, G PU2 will not find the solution since we use upper bound 12 to cut 20, which will enhance the performance of GPU 2.

Finally, we use *number of nodes required exploration every check* to represent the fre-

quency that each thread checks the progress of other threads in GPU. For each thread, it needs to check the latest upper bound of the optimal solution to prune nodes in the search tree. However, the frequency is a metric related to the problem size. If the problem size is relatively small, we might need to check the progress of other threads more frequently to ensure we get the latest update. If the problem size is relatively large, which means each thread needs more time to find the answer, then frequent check will cause much more overhead.

For the experiment results, we assume that the grid size is 512, the block size is 1024, the *initial upper bound of solution* is always *search depth* plus one, and the number of nodes required exploration is 1000.

parallel performance				
number of GPU	search depth	initial upper bound	nodes required exploration	timing results(ms)
0	7			10
0	8			40
0	9			870
0	10			2580
0	11			42480
1	7	8	1000	7.12208
1	8	9	1000	62.2082
1	9	10	1000	262.382
2	7	8	1000	2.6911
2	8	9	1000	20.48
2	9	10	1000	40.2382
2	10	11	1000	617.874
2	11	12	1000	5448.83

From the table above, we can see the scaling of performance of using different number of GPUs. One exception worth noticing is that, using 1 GPU with search depth 8 is slower than the serial version. This is reasonable because in this case the serial version is pretty fast, and the overhead of parallelization in the GPU reduces its performance.

In addition, for the fixed search depth, we do experiments on different upper bounds. The experiment results show that the closer the upper bound is to the number of scramble turns, the more nodes are pruned earlier and thus the better performance we can get.

relation between search depth and upper bound				
number of GPU	search depth	initial upper bound	nodes required exploration	timing results(ms)
2	10	10	1000	94.3432
2	10	11	1000	1068.73
2	10	12	1000	11155.2
2	10	13	1000	14005.2
2	10	14	1000	20445.5

Finally, we do some experiments on the relation between the number of nodes a thread explore between every check on the local optimal solution and input size, which depends on search depth and upper bound.

relation between number of nodes required exploration and input size				
number of GPU	search depth	initial upper bound	nodes required exploration	timing results(ms)
2	7	8	100	2.05792
2	7	8	500	5.23638
2	7	8	1000	8.696
2	7	8	5000	11.7814
2	7	8	10000	15.6995
2	8	9	100	3.86986
2	8	9	500	6.91632
2	8	9	1000	18.2116
2	8	9	5000	37.3685
2	8	9	10000	15.8982
2	9	10	100	206.113
2	9	10	500	131.3
2	9	10	1000	30.7569
2	9	10	5000	286.781
2	9	10	10000	59.2846

From the table above, we can see that there is a balance between input size and nodes required to be expanded. If the input size is small, we need to check the running upper bound of the solution more frequently. If the input size is larger, then the effect of overhead cost will become significant.

8 References

[1] Richard E. Korf. 1997. Finding optimal solutions to Rubik's cube using pattern databases. In Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence (AAAI'97/IAAI'97). AAAI Press 700-705.
[2] Davidson, Morley; Dethridge, John; Kociemba, Herbert; Rokicki, Tomas. "God's Number is 20". cube20.org.