GPU-acceleration of Optimal Permutation-Puzzle Solving

HAYAKAWA Hiroki
Department of Information Network Systems
The University of Electro-Communications
Tokyo, Japan

ISHIDA Naoaki
Department of Communication Engineering and Informatics
The University of Electro-Communications
Tokyo, Japan

MURAO Hirokazu
Department of Communication Engineering and Informatics
The University of Electro-Communications
Tokyo, Japan

ABSTRACT

We first investigate parallelization of Rubik's cube optimal solver, especially for acceleration by GPU. To examine its efficacy, we implement a simple solver based on Korf's algorithm, with which CPU and GPU collaborate in IDA* algorithm and a large number of GPU cores are utilized for speedup instead of a huge distance table used for pruning. Empirical studies succeeded to attain sufficient speedup by GPU-acceleration.

There are many other similar puzzles of so-called permutation puzzles. The puzzle solving, i.e., restoring the original ordered state from a scrambled one is equivalent to the path-finding in the Cayley graph of the permutation group. We generalize the method used for Rubik's cube to much smaller problems, and examine its efficacy. The focus of our research interest is how efficient the parallel path-finding can be and whether the use of a large number of cores substitutes for a large distance table used for pruning, in general.

CCS Concepts

•**Theory of computation** → **Shortest paths;** *Massively parallel algorithms;* •**Computing methodologies** → **Game tree search;** •**Computer systems organization** → *Multicore architectures;* •**Mathematics of computing** → Permutations and combinations;

Keywords

GPU; parallel; permutation puzzle; Cayley graph, path-finding

1. INTRODUCTION

This paper introduces a new example of GPU application. A certain kind of puzzles, called permutation puzzles, can be treated by group theory, and solved as path-finding in the Cayley graphs. We shall apply GPU's computing power to this path-finding problem.

PASCO '15, July 10 - 12, 2015, Bath, United Kingdom

http://dx.doi.org/10.1145/2790282.2790289

Rubik's cube is well-known as a cubic mechanical puzzle invented in 1974 by Erno Rubik, and is often treated as a puzzle for competitive events or an abstract mathematical problem in group theory. From the viewpoint of mathematics, the puzzle constitutes a group of its own, by regarding moves as permutations of parts of cube, and the group theory is utilized for analysis of the structure conducting to theoretical solving algorithms. Because of the huge size of the problem, the use of computers is indispensable for group theoretical analysis and also for obtaining concrete maneuvers or its length. Besides the series of Rubik's cubes of different sizes, there have been developed various 3D mechanical puzzles such as Pyraminx, Megaminx, Skewb, and so on. All those puzzles can be treated in the same way, and generically termed as permutation puzzles.

From the viewpoint of group theory, solving a problem of any permutation puzzles, namely restoring the "home" position[1] of the puzzle from a given scrambled position, is equivalent to finding a path between the nodes corresponding to the two positions in the Cayley graph of the group of the puzzle. The graph is composed of the nodes of all possible positions of the puzzle and the edges representing all possible moves. In the case of $3 \times 3 \times 3$ Rubik's cube, the complete Cayley graph consists of over 4.33×10^{19} nodes, each of which branches out in 18 edges. The diameter of the graph, called "God's number", or its bound has long been a main research subject, where optimal solvers or shortest-path finding programs play a key role.

A milestone algorithm is presented by Thistlethwaite[2]. His algorithm consists of 4 major stages corresponding to the chain of nested subgroups, $G_4 =< 1 > \subset G_3 \subset \cdots \subset G_0 =$ Rubik's Cube group, defined by the respective sets of restricted moves. Sequences of actual moves in each stage are given by the respective look-up table, whose number of entries is equal to the number of cosets, $\mid G_i/G_{i+1} \mid$, and is quite feasible as 1,082,565 at most. The worst numbers of required moves are shown as 7, 13, 15 and 17, and the bound for the total is 52. In 1992, Thistlethwaite's algorithm was improved by Kociemba[5], by reducing the number of stages, i.e., the length of the subgroup chain. As its result, the number of cosets increased significantly, so that it became almost impossible for daily-use computers to maintain the lookup table of transitions for all cosets. Thus, in the two-phase algorithm, the path-search method is switched from table lookup to a method equivalent to IDA* algorithm[6], which performs efficient depth-first search with pruning by

[1]To represent a status of a puzzle, the terms *position* and *state* are used interchangeably.

heuristic function making use of some cost table. In 1985, Reid proved that the bound can be lowered to 29 by using the same two subgroups[10]. Besides the research for lower bounds, development of an optimal solver, a program to fix the smallest number of moves for a given position required to restore, is important. Optimal solvers are used also for decreasing the lower bounds for concrete positions[8]. There are two ways of counting God's number, half-turn metric and quarter-turn metric, and they are determined in 2010 and 2014. For the details of the history arriving at God's numbers, refer to [12, 1].

There have been developed several optimal solvers. The key strategy is described in [7]. The performance of the algorithm heavily depends on the distance tables of the smallest path lengths for all possible positions and cosets, especially on their sizes. Reid's solver [10] uses 70MB table, and Kociemba's does 1.8GB table. The size of the table is further extended to 8GB in Rokicki's later implementation in 2010, which reveals much higher performance than any other predecessors. These practices indicate a pragmatic fact that the execution speed of the algorithm is highly dependent on the amount of available memory and that the increase of memory is a key to execution speedup.

In this paper, we shall take a different approach. We try to obtain higher performance by making use of GPU as an accelerator, but without employing a huge expansion of the tables. In order to investigate the efficacy of GPU-acceleration, we implemented a version of Korf's solver, in a simpler style than the one by M.Reid, and parallelized for GPU with CUDA. Evaluation is done by comparing the timings of the non-parallel version executed on a single CPU core and of the parallel version executed using GPU. The attempt and the evaluation are made mainly on $3 \times 3 \times 3$ Rubik's cube. In the evaluation of parallel processing, practical results may imply much more meaning and have much more importance than simple analysis in theory, in general, although they heavily depend on the computing environment actually used. In this paper, we shall show and investigate some experimental results obtained by using an average CUDA-enabled GPU of a former generation. We observe that the path-finding algorithm can be reasonably accelerated and, we point out that some graph algorithms can be a good application area of GPU. Another topic in this paper is to investigate whether our method can be generalized to other permutation puzzles, and, if possible, what is its effect. We extend our method and apply to Pyraminx and Skewb, and present empirical results. With these puzzles, computerized solving is so simple that the overhead of the data transfer to/from GPU as an external device is dominant.

Organization of Paper

The paper is organized as follows. In Section 2, we characterize the permutation puzzles treated in this paper, and show how they can be modeled and represented in computers. In that, Korf's algorithm for Rubik's cube plays an important role, and we describe in some detail. In Section 3, we consider how we can construct optimal solvers, and Section 4 is devoted to describe our method used for parallelization for GPU. Section 5 is for empirical study, where we describe how some implementation parameters for GPU processing are determined and present timing data. Section 6 concludes this paper.

2. PUZZLES, MODELLING, AND REPRESENTATION

We describe the permutation puzzles treated in this paper, and how these puzzles are modeled and represented in computers very briefly. The puzzles treated are $2 \times 2 \times 2$ and $3 \times 3 \times 3$ Rubik's cube, Pyraminx, and Skewb. For the details of these puzzles, refer to [13], especially for beautiful pictures of physical models, and for mathematical treatment, refer to [4]. An easy way to represent permutation puzzles is to give a label to each face of moving parts, and to represent some move by a permutation of the labels of moved parts. This method is suited for group theoretical analysis using computer algebra systems such as SAGE, but not good for huge amount of calculations of path-finding. We first describe the well-known method for $3 \times 3 \times 3$ Rubik's cube, as an introductory study.

2.1 $3 \times 3 \times 3$ Rubik's cube

We shall use the standard terminology and convention[2]. A $3 \times 3 \times 3$ Rubik's cube consists of 26 visible *cubies*(small cubes), and each face of a cube is divided into 9 *facelets* (faces of cubies). Of 26 cubies, 8 are *corner cubie*s with 3 visible facelets, 12 are *edge cubie*s with 2 visible facelets, and 6 are *center cubie*s having one visible facelet in the center of a cube face. The center facelets are considered to be fixed. In the *home* position or solved (goal) position of the cube, all facelets in a face have the same color.

For the names of the moves and of the parts of the cube, we use Singmaster notation[2] as usual. Faces of the cube are labeled as U(p), D(own), R(ight), L(eft), F(ront), and B(ack), and the same letter is used to represent the move of clockwise quarter turn of the corresponding face. Let $S = \{ U, D, R, L, F, B \}$ denote a set of these moves. For each move $x \in S$, the corresponding half turn move and anticlockwise quarter turn move of the same face are denoted by x^2 and x^{-1} respectively. Notice that x^4 is an identity move for any $x \in S$. There are 18 kinds of moves, any of which move 20 facelets at a time and can be regarded as a permutation of facelets. In *half-turn metric*, all moves are counted as one move, while in *quarter-turn metric*, half-turn moves x^2 for $x \in S$ are counted as 2 moves. Any subset of S generates a group, and especially, a group generated by S, usually denoted by $< U, D, R, L, F, B >$, is called Rubik's Cube group. There is one-to-one correspondence between the group elements and the cube states.

There are many well-known facts and techniques commonly used for programming.

- The order of Rubik's Cube group or the number of possible cube states is $8! \times 3^7 \times 12! \times 2^{10} \approx 4.3 \times 10^{19}$.
- God's number is 20 in half-turn metric, and 26 in quarter-turn metric.
- A cube state is identified by the positions and the orientations of 8 corner cubies and 12 edge cubies, and may be represented by 40 variables. Each corner cubie can take one of 8 possible positions and one of 3 possible orientations, while each edge cubie can do 12 positions and 2 orientations.
- The orientation of one corner cubie is determined by those of other 7, and that of one edge cubie is by those of other 11. Therefore, there are 3^7 and 2^{11} possible

combinations of the orientations for the corner cubies and the edge cubies respectively, and each combination of the orientations can be numbered and encoded compactly as a number less than the corresponding amount.

- The permutations of the corner cubies and of the edge cubies must have an equal parity.
- Permutations of n can be represented by a sequential number in $\{0, \ldots, n!-1\}$ via factorial number system. This method can be used as a compact representation for a set of positions of corner cubies or of edge cubies.

With these methods for compact representation, a state of the cube can be represented by 4 kinds of data, and the data translation corresponding to a single move is done by using multiple tables.

2.2 $2 \times 2 \times 2$ Rubik's cube

$2 \times 2 \times 2$ Rubik's cube is equivalent to a puzzle using 8 corner cubies of $3 \times 3 \times 3$ puzzle. The cube has $6 \times 4 = 24$ facelets, and a single move is regarded as a permutation of half of them. The orientation of one cube is determined by those of other 7, and all 24 possible positions and orientations of a fixed corner cubie are equivalent. Therefore, the number of possible states of the cube is $8! \times 3^7/24 = 7! \times 3^6 = 3,674,160$. God's number is known to be 11 in half-turn metric and 14 in quarter-turn metric. Compact representation of a state of the cube requires only two variables, and therefore, two tables are used for data translation of moves.

2.3 Pyraminx

Pyraminx is a puzzle of tetrahedron, of which each edge is trisected and each face, having one color in the home state, is divided into $(1 + 3 + 5)$ identical triangles, called facets. With each vertex, 1/3 and 2/3 sub-tetrahedron can be twisted by 120 or 240 degrees. For each vertex $v \in \{ U(p), R(ight), L(eft), B(ack) \}$, move v is defined to be a clockwise twist of 2/3 sub-tetrahedron by 120 degrees, and an anticlockwise twist or two-times repetition v^2 of v can be its inverse v^{-1}. The tetrahedron itself is composed of 4 trivial corner pieces of 1/3 sub-tetrahedron, 6 edge pieces with 2 facets, and 4 axial pieces with 3 facets, where each axial piece neighbors to one corner piece. We ignore the move of trivial corner pieces, because they can be twisted independently and restored trivially.

- There are $4 \times 9 = 36$ triangle facets, and a single move can be regarded as a permutation of 12 facets.
- A state of Pyraminx is identified by the positions and orientations of 6 edge pieces and the orientations of 4 axial pieces, may be stored in 16 variables in total. Notice that the positions of all axial pieces are fixed. There are 6 possible positions and 2 possible orientations for each edge piece, and 3 possible orientations for each axial piece.
- The orientation of one edge piece is determined by those of other 5. Therefore, there are 2^5 possible orientation combinations with edge pieces, and each combination can be represented by a number in $\{0, ..., 2^5-1\}$. Likewise, the orientation combination of axial pieces can be represented by a number in $\{0, ..., 3^4 - 1\}$.
- The permutation of the edge pieces must be of even parity. The positions of all edge pieces can be configured in 6!/2 distinct ways, and each configuration can be represented by a number in $\{0, ..., 6! - 1\}$.
- The problem space is a group $< U, R, L, B >$ generated by 4 kind of moves, and its order, i.e., the number of possible states, is equal to $(6!/2) \times 2^5 \times 3^4 = 933,120$.
- God's number is 11.

Compact representation of a state of Pyraminx uses three variables, and three tables are used for data translation corresponding to a single twist.

2.4 Skewb

According to the Web-page "Skewb" of [13], Skewb is described as follows.

> The Skewb is a cube where you don't turn the faces, but instead turn exactly half the cube around the corners. The cube is bisected 4 ways, perpendicular to each of the 4 main diagonals. It therefore consists of 8 corner pieces and 6 square face pieces.

Note the well-known fact that 4 corner pieces have fixed positions in the form of a tetrahedron but can be rotated. Each square face of the cube has 4 cuts between the middle points of adjacent edges, and is divided into a single center square facet of half of the face and 4 corner right-triangle facets. A move (or, turn) is a clockwise twist around one of 8 corners, and two times repetition of a move has the same effect as an anticlockwise twist and works as the inverse of the move.

- A move around a corner changes the positions and orientations of 3 adjacent corner pieces and of 3 adjacent square face pieces, and it can be regarded as the product of the permutations of 4×3 triangle facets and of 3 square facets.
- A state of Skewb is identified by the positions and orientations of 8 corner pieces and the positions of 6 square face pieces[2].
- There are 4 and 6 possible positions for each movable corner piece and each square face piece, and there are 3 distinct orientations for each corner piece.
- The permutations of square face pieces and of 4 movable corner pieces must be of even parity.
- Eight corner pieces can be regarded as constituting two tetrahedrons (of 4 fixed pieces and of 4 movable pieces), and the orientation combination of each tetrahedron is determined by three corners of each.
- The positions of square face pieces and of corner pieces can be encoded as numbers in $\{0, ..., 6!-1\}$ and $\{0, ..., 4!-1\}$.
- The number of possible states of Skewb is $(6!/2) \times (4!/2) \times 3^6 = 3,149,280$.

[2]The orientations of square face pieces are ignored, and the puzzle taking this orientations into account is called "Super-Skewb".

- God's number of Skewb is 11.

With compact representation of Skewb, 3 variables are used for the state of the cube and 3 tables for the translation of moves.

3. DESIGN OF OPTIMAL SOLVER

The problem space of a permutation puzzle corresponds to the Cayley graph of the corresponding permutation group. Solutions to a permutation puzzle are the paths between the nodes of a given scrambled state and of the home state in its Cayley graph, and optimal solutions mean the shortest paths. If the problem space is of manageable size, path-finding may be completed by exhaustive breadth-first or depth-first search, and in other cases, good heuristic function is required to perform efficient search algorithm such as A*. A distance table for all reachable nodes to the solution(home) node serves good heuristics, however it can often be too big, as in the case of Rubik's cube. The subgroup method is useful for permutation puzzles to reduce the sizes of distance tables. The method introduces a chain of subgroups from the original group to the trivial (sub)group $< 1 >$ of the solution, and performs path-finding in multiple stages. For $3 \times 3 \times 3$ Rubik's cube, Thistlethwaite's algorithm which uses a chain of length 4 has become a breakthrough to the later research, and one of the subgroups has become common to current solving programs. The order of the subgroup is $2,217,093,120$. Today's computers afford to manage tables of this number of entries. We shall use such a table with Korf's algorithm. In the following, we describe Korf's algorithm for optimal solutions to $3 \times 3 \times 3$ Rubik's cube in some detail.

3.1 Korf's Algorithm

An algorithm for optimal solutions is designed by Korf[7], and the details of a similar algorithm can be found in [5]. They employ the so-called IDA*(Iterative-Deepening A*) algorithm[6], which is described in [7] as follows:

> IDA* is a depth-first search that looks for increasingly longer solutions in a series of iterations, using a lower-bound heuristic to prune branches once their estimated length exceeds the current iteration bound.

Namely, a depth-first search is repeated by setting the limit of the search depth initially 1 and increasing by 1.

The algorithm is characterized by two major strategies designed for efficiency: the one is for avoiding repeated visit to the same node, and the other is about pruning. Multiple visits to a node is usually avoided by remembering visited nodes, which requires a large amount of memory. Korf's algorithm does not use such a non-strategic method to completely avoid duplication, but takes such a strategy to restrict the choice of moves so as to reduce the generation of duplications. The restrictions are placed only on consecutive twisting moves, and they are as simple as the following:

- do not twist the same face twice in a row, and
- consecutive twists of opposite faces must be done in a fixed order.

The former one is rationalized by the fact that two consecutive moves for a single face can be represented by a single move. The rationale of the latter will be also clear. The effect of the consecutive twisting operations of opposite faces is independent of the operation order, e.g., the operations L after R and R after L are equivalent, and therefore we have only to consider either one of two orders. Even with these two simple conditions, the effect is turned out sufficient in practice; only 5% duplication is observed at depth 10. Also note that memory access for checking the visited nodes is not required, which critically affects on efficiency.

Pruning, the second strategic point, is done by using some heuristics. To obtain a shorter path, IDA* algorithm requires such a heuristic function that computes, for a given node, the lower bound of path lengths to the home node. In [7], Korf suggests the use of a table of $8! \times 3^7 = 88,179,840$ possible states of corner cubies. Most of the current major solvers use a two-phase subgroup method, with which IDA* algorithm is used in the first phase by making use, as a heuristic function, of a complete table of distances, called *distance table*, to some element in $H =< U, D, R^2, L^2, F^2, B^2 >$. This method only gives shortest paths from an arbitrary state into H, but notice they are not necessarily an optimal one to the home. For optimal solutions, we take a different approach.

More concrete description is given by Algorithm 1, where the current status Σ given to the algorithm is represented as a tuple of three elements, position of the puzzle s, preceding move m, and the length k of moves so far, and a set of possible moves is

$$M = \{m^j \mid m \in \{F, B, R, L, U, D\}, j = 1, 2, -1\}.$$

Algorithm 1 depth-first path-search

Input: the status $\Sigma_k = (s_k, m_k, k)$, and the limit value *lim* of search depth. We assume the set M of all possible moves is defined as above.

Output: A sequence of moves if the *home* position is reached within *lim* moves, and otherwise nothing.

```
 1: if s_k is home position then
 2:     return the sequence m_1, ..., m_k of moves.
 3: else if k ≥ lim then
 4:     return with no result.
 5: end if
 6: for all m ∈ M do
 7:     if m is not allowed just after m_k then
 8:         skip this move m.
 9:     end if
10:     Let s_new be the position obtained
                by applying the move m to s_k.
11:     Evaluate the distance d of s_new
                to the home position.
12:     if d is obtained and k + d ≥ lim then
13:         skip this move m.
14:     end if
15:     Apply this algorithm recursively
                to the new status Σ' = (s_new, m, k + 1).
16:     if any result is returned then
17:         return the result by breaking the for loop.
18:     end if
19: end for
```

At line 7, the move m is checked against the restictions

stated above. Notice that our simplified IDA* algorithm does not memorize the newly created positions to save the memory usage, and works with a small overhead. A newly position s_{new} is not checked if ever tested, and in the execution of the search algorithm to the depth limit d, all the search processes of depth $1, ..., d-1$ are re-executed from the initial scrambled position. In the case that search depth is appropriately bounded and we use a GPU with a large number of processing cores but with a limited amount of memory, recalculation is easier than the complicated management of subproblems and of memory space.

3.2 Distance Table and Approach

A distance table contains the least numbers of moves required to reach a solution (home node) for each node of the search tree. In IDA* algorithm, when seeking a solution of a fixed length, such information are used for pruning, i.e., those nodes whose total length to the home node, computed as the sum of the depth in the search path and the number in the table, exceeds the value of length will be discarded from the search paths. The numbers contained in the table are precomputed for all possible positions of some typical set of parts of a cube of a subgroup. We may expect higher performance if the table size increases.

It is well-known that in the above subgroup H, the orientations of all edge and corner cubies are constrained and 4 edge cubies in the UD-slice, a slice between the U and D faces, stay isolated in that slice. Therefore, there are $2^{11} \times 3^7 \times {}_{12}C_4 = 2,217,093,120$ cosets of H in Rubik's Cube group. Each coset is identified by the orientations of edge cubies and of corner cubies, and the combination of the edge cubies, and is represented by a simplified coordinate of these 3 elements. The numbers of possible states of each element is shown as factors in the above formula, and is used as the upper limits of encoded numbers. Move is translated into this coordinate, and the maximum number of moves required from an arbitrary node to H is known to be 12. Therefore the distance table of the cosets requires about 1GB memory. Execution of IDA* algorithm with this distance table is usually the first phase of the two-phase algorithm. For optimal solutions, suboptimal paths in this phase will be required.

In order to attain the optimality of the results of path-finding, we directly use IDA* algorithm, until a solution (a path to the home) is found, instead of further searching in two-phases. Here, we notice

- the identity element corresponding to the home node can be reached as a member of H,
- this path-finding can be done by IDA* algorithm with a distance table of the cosets of H,
- where the state of a cube and moves are represented in terms of a simplified coordinate of 3 elements, while the check for the identity must be done with complete coordinate,
- and, if found, the path is definitely shortest.

Some notices are listed below.

- Conversion of coordinate from a complete tuple of 4 elements to a simplified tuple with 3 elements is so simple that only the positions of edge cubies need be translated into a non-negative integer less than ${}_{12}C_4$. The translation of position is a bit complicated but computed very easily.
- The standard orientation ought to be chosen so as to be unchanged by any of basic operations of H.
- Shrink of the table size by the symmetry of a cube can be a trade-off with the complication of computation. In our implementation, symmetry is not taken into account for simplicity and to observe the pure effect of GPU-acceleration.
- The distance table can be used in 3 directions, i.e., U-D, R-L and F-B, and the application in 3 directions serves better heuristics.
- Furthermore, use of multiple different distance tables can be better heuristics in general. The table for the rare cases called *superflip* can be realized by 42MB memory.

4. PARALLELIZATION FOR GPU

Korf's algorithm basically traverses the search tree in a depth-first manner. The tree is characterized by the independency among the nodes of the same depth. A set of nodes of a certain depth are regarded as a number of independent subproblems, to which parallel processing by GPU is applied. We design a collaborative algorithm for CPU and GPU, of the model depicted in Figure 1. There are many publications on parallel IDA*, e.g., [11, 9], however, most of them are for older parallel machines of different architecture and for homogeneous multiprocessor systems. There is a recent research for a similar purpose [3], and our design may be regarded as its iterative version. Recently, a smart parallel A* algorithm is designed for GPU, where the main focuses are laid on the problem of node duplication detection in A* search and memory bounded A* search[14]. Our treatment is simple, as mentioned before.

On CPU, search is performed to a certain depth to collect subproblems, and, whenever a certain amount of subproblems, say W, are collected, a GPU kernel of the solver is invoked to perform efficient parallel search. The search length limit ℓ of subproblems must be kept equal to make their computing costs balance among the running GPU kernels. Notice that if the limit is increased by 1, then the cost increases by a factor more than 10. The limit of search depth performed on CPU is given by God's number minus ℓ, and this parameter ℓ will be determined by experiment so as to make the total processing most efficient. Also, the number of subproblems processed in a single kernel call is to be determined so that the kernel call is completed within one second in order to prevent GPU timeout. In practice, several thousands to several millions of subproblems will processed per invocation.

A GPU kernel executes the IDA* algorithm with a non-recursive version of Algorithm 1. The current version of CUDA supports recursive call in a kernel call, however for safety, we realize a non-recursive version by preparing a stack of statuses. Figure 2 describes processing flow on the host CPU and GPU. By breaking up the process for collecting small subproblems on CPU and the search process on GPU into small parts to be executed in multiple steps,

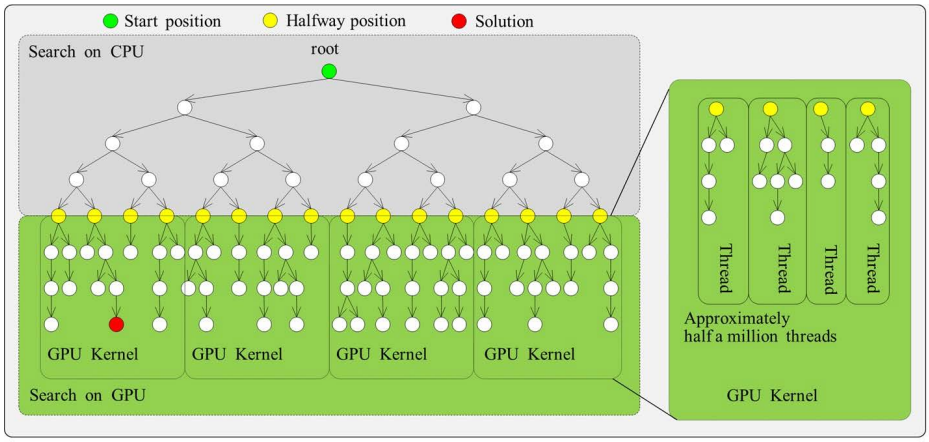

Figure 1: Model of parallel search with GPU

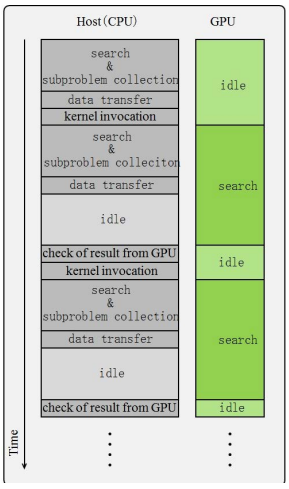

Figure 2: Processing Flow

Algorithm 2 Modified version of depth-first search for CPU

Input and Output: same as those of Algorithm 1

1: **if** s_k is *home* position **then**
2: **return** the sequence $m_1, ..., m_k$ of moves.
3: **else if** $k + \ell \geq$ *God's number* **then**
4: put the status $\Sigma_k = (s_k, m_k, k)$ into the queue of subproblems. If the queue contains W problems, then invoke the GPU kernel to process the subprolems.
5: **end if**
6: ...

the time for search on CPU and the latency of data transfer can be hidden, which enables efficient execution of the whole search processing. Data representing subproblems is transferred from CPU to GPU in the form of a sequence of moves used for transition from the root node to a subproblem, instead of the cube state corresponding to a subproblem, to decrease the amount of data transfer. There are 18 kinds of moves and one move can be represented using 5bit. The number of moves is bounded by 20 and 26 in half- and quarter-turn metrics respectively, and therefore, two and three 32-bit words suffice to represent the sequence. On GPU, each subprolem is recovered from the initial scrambled state and the sequence, and if a solution is encountered, the whole sequence of moves are written to be sent back to CPU.

CUDA defines memory hierarchy of different access speeds. The fastest one is register file shared by CUDA cores, and its capacity depends on the CUDA architecture. In any case, much more registers are available on GPU than on usual CPU, and ought to be utilized extensively via temporary variables in kernel functions. Constant memory is the second fastest read-only memory, and ought to be used for frequently-accessed tables, e.g., for transition table for move operations.

5. EMPIRICAL STUDY

The computing environment used for our experiment is summarized below.

OS	Ubuntu 14.04
CPU	Core i7 4970 (3.6GHz, 8MB L1-cache)
RAM	2*8GB=16GB
GCC	ver 4.8.2, `-O3`
GPU	GTX780Ti(VRAM:3GB
CUDA	ver 7.0

The GPU we used is a bit outdated, whose Kepler architecture is one generation behind from the latest Maxwell. However, overall configuration is similar, and most of the differences lie in the numbers of component parts. In order to conform to various GPU architectures, it is common to introduce parameters into programs and adjust them for GPU used in practice. We shall investigate how we can improve the efficiency of GPU-parallel execution very briefly.

5.1 Puzzles of Small Problem Size

Puzzles used are as follows:

(R2) $2 \times 2 \times 2$ Rubik's cube,

(Py) Pyraminx,

(Sk) Skewb, and

(R3) $3 \times 3 \times 3$ Rubik's cube,

and hereinafter, we use these labels to identify. The problem space of (R2), (Py) and (Sk) is sufficiently small, and we can use the complete distance tables of respective sizes 42MB, 912KB and 18MB in memory. Any of transition tables for orientation and position required only $1 \sim 200$KB.

Distance Table and Pruning

We investigate how effective the distance tables are for pruning. We use 1,000 problems generated by 100 random moves. We prepare non-complete distance table of the following subgroups.

	(R2)				(Py)					(Sk)			
#moves	#	(a)	(b)	(c)	#	(a)	(b)	(c)	(G)	#	(a)	(b)	(c)
3	1	0.003	0.002	0.008	0	—	—	—	—	0	—	—	—
4	1	0.002	0.002	0.027	2	0.002	0.002	0.030	0.638	2	0.002	0.007	0.025
5	2	0.004	0.005	0.183	9	0.003	0.003	0.172	0.888	4	0.002	0.003	0.189
6	12	0.003	0.009	1.141	60	0.006	0.005	1.163	1.186	19	0.003	0.005	1.111
7	58	0.004	0.034	7.750	209	0.006	0.012	7.199	1.431	110	0.004	0.016	7.285
8	238	0.004	0.158	49.35	523	0.008	0.051	40.85	1.696	371	0.004	0.072	45.08
9	509	0.005	0.742	280.9	196	0.009	0.194	168.8	1.925	469	0.005	0.312	218.4
10	178	0.006	2.533	1,062	1	0.020	1.313	1,484	1.823	25	0.005	1.155	841.8
11	1	0.006	12.098	5,340	0	—	—	—	—	0	—	—	—

Table 1: Timings (msec) when using different distance tables

depth	(R2)	(Py)	(Sk)
1	9 → 9	8 → 8	8 → 8
2	90 → 63	72 → 56	72 → 56
3	819 → 387	584 → 344	584 → 344
4	7,380 → 2,331	4,680 → 2,072	4,680 → 2,072
5	66,429 → 13,995	37,448 → 12,440	37,448 → 12,440
6	597,870 → 83,979	299,592 → 74,648	299,592 → 74,648
7	5,380,839 → 503,883	2,396,744 → 447,896	2,396,744 → 447,896
8	48,427,560 → 2,023,307	19,173,960 → 2,687,384	19,173,960 → 2,687,384
9	32,075,174 → 1,005,995	12,431,320 → 1,082,144	24,542,114 → 2,122,358

Table 2: The numbers of nodes generated without using distance table

$\ell = 12$			$\ell = 13$			$\ell = 14$		
W	timing		W	timing		W	timing	
2^{19}	0.09	5.97	2^{16}	0.14	5.02	2^{12}	0.15	9.81
2^{20}	0.17	5.95	2^{17}	0.25	4.58	2^{13}	0.17	5.90
2^{21}	0.34	5.91	2^{18}	0.46	4.25	2^{14}	0.27	4.70
2^{22}	0.61	5.92	2^{19}	0.85	3.91	2^{15}	0.51	4.69
2^{23}	1.28	6.16	2^{20}	1.40	3.88	2^{16}	1.06	4.81

Table 3: Timing dependency on search length on GPU and the number of subproblems

#thread/block	timing(sec)
16	5.61
32	4.52
64	3.87
128	3.91
256	3.95
512	4.01

Table 4: Timing dependency on the number of threads/block

length	#	(A)CPU	(G)GPU	(A)/(G)
16	49	0.0322	0.0444	0.73
17	53	0.275	0.068	4.0
18	54	2.27	0.161	14.1
19	54	24.2	0.937	25.8
20	76	133	4.85	27.4
21	43	743	25.6	29.0
22	20	2548	97.8	26.1

Table 5: Speedup ratio

- for $2 \times 2 \times 2$ Rubik's cube, a table for only the position of cubies, i.e., orientations are ignored.
- for Pyraminx, a table for the posision and orientation of only edge pieces.
- for Skewb, a table for the orientation of corner pieces and the position of movable corner pieces.

Table 1 summarizes the number of problems for each of the required number of moves and the timings of the following cases:

(a) uses the complete distance table,

(b) uses the distance table of the above subgroup, and

(c) does not use any distance table.

With Pyraminx, timings using GPU are included in the column (G). The performance of distance tables are clear.

Next, we observe the effect of the first strategy used in Korf's algorithm to avoid the generation of duplicated nodes. We counted the numbers in each search depth for a single problem of length 9. Table 2 shows the difference of the numbers of nodes generated with and without using the strategy. Notice that the effect of complete distance table includes that of the first strategy.

GPU processing

We should mention the effect of the use of GPU. As can be seen in Table 1 and as can be anticipated, any problem of these small puzzles can be solved so fast that we cannot expect the acceleration by GPU, and it was confirmed via our experiment. Such a situation is due to the overhead of the generation of subproblems and data transfer. Compare the timings in (a) and (G) of Pyraminx. Timings are taken with the above 1,000 problems with $\ell = 2$, where the most part of path-finding is done by CPU. So, the timing difference corresponds to the overhead for data transfer.

5.2 $3 \times 3 \times 3$ Rubik's Cube(Quarter-Turn Metric)

Parameter Settings for GPU processing

For the puzzles with large problem space, we determine some parameters by experiment to utilize GPU highly and efficiently, as explained in the previous section. Let ℓ denote the limit of search depth to be executed purely on GPU, i.e., the limit set to the problem in IDA* minus the path length from root node to the subproblem, and let W do the number of subproblems processed in a single kernel call. W must be chosen so that any kernel call is completed within one second. Experiment is done for some typical cases with $W = 2^j$, $j = 12, ..., 23$ and $\ell = 11, ..., 15$, where the number of threads per block is chosen 128, and a problem requiring 20 moves is used. For a fixed ℓ, the execution time of a single kernel call increases monotonically as W increases, while the total execution time decreases as an effect of parallel execution. With $\ell = 11$, a single kernel call is completed within 0.2 seconds but the total execution took > 20 seconds for any W, and this choice of ℓ turned out useless. With $\ell = 15$, GPU's workload is so high that a single kernel call requires > 1 second even for $W = 2^{12}$ and the total execution time does > 8 seconds for $W = 2^{13}$, therefore, this choice of ℓ is useless. Table 3 summarizes some useful timings (in seconds); timings of a single kernel call on the left and of the total execution on the right. The best choice is $\ell = 13$ and $W = 2^{19} = 524,288$.

Table 4 summarizes the dependence of the total execution time on the number of threads per block. The case of 64 threads per block is most efficient and this value is used in our experiment.

With our former experiment using GPU of different architecture of one-generation behind, these values are $W = 2^{19} = 524,288$, $\ell = 11$ and $p = 128$ or 256. We should refer to some interesting result from the former experiment. We counted the number of generated nodes by changing the number of preceding moves used to avoid the generation of duplicated nodes up-to 4, and we found that the number varies by only 5%. Another result is about the effect of the utilization of register files and constant memory of GPU. By highly utilizing these fast memory execution time can be halved.

Timings for Optimal Solutions

Timings are taken for two cases of (A) CPU only and (G) CPU+VPU, using 350 problems, each 50 of which are shuffled by $16 \sim 22$ random moves. Table 5 summarizes the actual path lengths to solutions and the numbers of problems, where the lengths are counted in quarter-turn metric. This table contains, for each length, the average computing times spent in two cases (A) and (G), and the last column presents its ratios. Summing up, as a result of the effort for higher acceleration by GPU, nearly 30 times speedup was achieved.

6. CONCLUSION

In this paper, we first described how we represent the puzzles in computers and how we develop optimal solvers for shortest path-finding, for 4 kinds of permutation puzzles, and then parallelized for GPU. For the optimality of solutions, we simply employ Korf's algorithm, where IDA* algorithm plays an essential role, and we designed an IDA* algorithm for collaboration with GPU. It has been a common recognition that for path-finding, the use of larger distance table, used for pruning, is a key for better performance. However, we replace part of the table's role by path-finding computation on a large number of processor cores of GPU. Rich computing resource affords to perform unfruitful search.

Our experiments are twofold. For puzzles of small problem space, complete table of distance to the solution can be prepared. It was confirmed by experiments that the effect of distance table is very high in IDA* algorithm for pruning, and there cannot be found for GPU-acceleration because the overhead cannot be ignored or hidden. There is a technical method for fast data transfer, mainly used by experts, making use of pinned memory space, and we may expect this overhead problem can be diminished by the method. Furthermore, it is announced by NVIDIA that new interface will be introduced in future GPU architecture to speedup data transfer. If graphics processing unit as SIMT-style processor became an additional processing unit, like FP unit, rather than an external device, it could be utilized for path-finding.

Another aspect is within our major focus. We designed our IDA* algorithm for collaboration with GPU, and performed its experiment with a large scale permutation puzzle, $3 \times 3 \times 3$ Rubik's cube. We investigated a method for effective

use of GPU. A huge table for pruning is shared in GPU and can be used effectively. The experimental results approved our prospect, mentioned above, of GPU utilization.

7. ACKNOWLEDGMENTS

The authors would like to express sincere gratitude to anonymous reviewers for careful reading and suggestions to improve this paper. The last author is partly supported by JSPS KAKENHI Grant Number 26330144.

8. REFERENCES

[1] God's number is 20. http://cube20.org/.

[2] A. H. Frey, Jr. and D. Singmaster. *Handbook of Cubik Math*. Lutterworth Press, 2010.

[3] R. Inam. A* algorithm for multicore graphics processors. Master's thesis, Chalmers University of Technology, 2010.

[4] D. Joyner. *Adventures in Group Theory*. Johns Hopkins University Press, 2nd edition, 2008.

[5] H. Kociemba. The two-phase algorithm. http://kociemba.org/twophase.htm.

[6] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[7] R. E. Korf. Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of National Conference on Artificial Intelligence: AAAI-97*, pages 700–705, 1997.

[8] D. Kunkle and C. Cooperman. Twenty-six moves suffice for Rubik's cube. In *Proc. ISSAC'07*, pages 235–242. ACM Press, 2007.

[9] B. Mahafzah. Parallel multithreaded IDA* heuristic search: algorithm design and performance evalution. *International Journal of Parallel Emergent and Distributed Systems*, 26(1):61–82, 2011.

[10] M. Reid. Optimal Rubik's cube solver. http://www.cflmath.com/Rubik/optimal_solver.html, 2006.

[11] A. Reinefeld and V. Schnecke. AIDA* – asynchronous parallel IDA*. In *Proc. 10th Canadian Conf. on AI: AI'94*, pages 295–302, 1994.

[12] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge. The diameter of the Rubik's cube group is twenty. *SIAM Journal of Discrete Mathematics*, 27(2):1082–1105, 2013.

[13] J. Scherphuis. Jaap's Puzzle Page. http://www.jaapsch.net/puzzles/.

[14] Y. Zhou and J. Zeng. Massively parallel A* search on a GPU. In *Proceedings of the 29th AAAI Conference on Artifitial Intelligence*, pages 1248–1254, 2015.