

Parallel State Space Search on the GPU

Stefan Edelkamp

Technologie-Zentrum Informatik
Universität Bremen
edelkamp@tzi.de

Damian Sulewski

Fakultät für Informatik
TU Dortmund
damian.sulewski@tu-dortmund.de

Abstract

This paper exploits parallel computing power of graphics cards for the enhanced enumeration of state spaces. We illustrate that modern graphics processing units (GPUs) have the potential to speed up state space breadth first search significantly. For a bitvector representation of the search frontier, GPU algorithms with one and two bits per state are presented. Efficient perfect hash functions and their inverse are studied for enhanced compression. We establish maximal speed-ups of up to factor 30 and more wrt. single core computation.

Introduction

Each modern processor contains at least two cores and graphics cards with hundreds of fragment processors are a commercial standard. This “parallelism for the masses” offers great opportunities for state space search and is expected to have an enormous impact on the implemented programs.

In particular, in the last few years there has been a remarkable increase in the performance and capabilities of the graphics processing unit (GPU). Modern GPUs are powerful, parallel programmable processors featuring complex arithmetic computing and high memory bandwidths. Deployed on current graphics cards, GPUs have outpaced CPUs in numerical algorithms such as matrix operations and Fourier transformations (Owens *et al.* 2008).

The GPU’s rapid development has inspired researchers to map computationally demanding, complex problems to it. With interfaces like NVIDIA’s general purpose programming language CUDA, GPU computing is an apparent candidate to speed up state space search.

To tackle the intrinsic hardness of large search problems, sparse-memory and disk-based algorithms are used jointly. I/O-efficient BFS (for undirected search spaces) has been suggested for explicit search spaces stored on disk by Munagala & Ranade (1999) and implemented for AI domains by Korf (2003).

Frontier search applies duplicate detection schemes, being either delayed (Korf 2003) or structured (Zhou & Hansen 2004). Especially on multiple disks, instead of I/O waiting time due to disk latencies, the computational bottleneck for these external-memory algorithms is internal time,

so that a rising number of parallel search variants have been studied (Korf & Schultze 2005; Zhou & Hansen 2007).

External two-bit breadth-first search by Korf (2008) integrates a tight compression method into an I/O efficient algorithm. The approach for solving large-scale problems relies on an invertible and perfect hash function. It applies a space-efficient representation in breadth-first search with two bits per state (Kunkle & Cooperman 2007), an idea that goes back to work by Cooperman & Finkelstein (1992).

In this paper we propose a smooth interplay of a bitvector state space representation and parallel computation on the GPU. Our examples are permutation games. We show how to efficiently rank and unrank a permutation on the GPU and how to compute the parity on-the-fly. To map the search space to a bitvector, we study GPU-based two-bit BFS, and, for large state spaces, one-bit variants.

The paper is structured as follows. Firstly, perfect hashing and rank and unrank functions for a selection of permutation games are presented. For minimal perfect hashing we have a closer look on the change of parity in these games, and combine its computation with lexicographic and alternative orderings. We then turn to space-efficient state space search on a bitvector, including known variants such as two-bit BFS and new variants that require only one bit per state. Then, we recall GPU essentials to introduce the underlying computational model. Porting the algorithms to the GPU is explained next, and its effectiveness is shown in a range of experiments.

Perfect Hashing in Permutation Games

A minimal perfect hash function is a one-to-one mapping from the state space S to the set $\{0, \dots, |S| - 1\}$. For many AI search problem domains perfect hash functions and their inverses are available prior to the search.¹ For permutation games, including the ones shown in Figure 1, they are called *rank* and *unrank*.

For the design of rank and unrank functions in permutation games, parity will be a crucial concept.

Definition 1 (Parity) *The parity of the permutation π is defined as the parity of the number of inversions in π , where*

¹Botelho & Ziviani (2007) show that, given the state space on disk, minimal perfect hash functions with a few bits per state can be constructed I/O efficiently.

inversions are all pairs (i, j) with $i < j$ and $\pi_i > \pi_j$.

In all games we consider the time for generating a successor is dominated by the time for ranking and unranking.

Sliding-Tile Puzzle

The $(n \times m)$ sliding-tile puzzle (Hordern 1986) consists of $(nm - 1)$ numbered tiles and one empty position, called the blank. The task is to re-arrange the tiles such that a certain goal arrangement is reached. Swapping two tiles toggles the permutation parity and in turn the solvability status of the game. Thus, only half the states are reachable.

There is one subtle problem with the blank. Simply taking the parity of the entire board does not suffice, as swapping a tile with the blank is a move, which does not change it. A solution is to partition the state space wrt. the position of the blank, since for exploring the $(n \times m)$ puzzle it is equivalent to enumerate all $(nm - 1)!/2$ orderings together with the nm positions of the blank. If B_0, \dots, B_{nm-1} denote the set of “blank-projected” partitions, then each set B_i contains $(nm - 1)!/2$ states. Given the index r as the permutation rank and B_i it is simple to reconstruct the puzzle’s state. As a side effect, horizontal moves of the blank do not change the state vector, thus the rank remains the same.

Top-Spin Puzzle

The next example is the (n, k) -Top-Spin Puzzle (Chen & Skiena 1996), which has n tokens in a ring. In one twist action k consecutive tokens are reversed and in one slide action pieces are shifted around. There are $n!$ different possible ways to permute the tokens into the locations. However, since the puzzle is cyclic only the order of the different tokens matters and thus there are only $(n - 1)!$ different states in practice. After each of the n possible actions, we thus normalize the permutation by cyclically shifting the array until token 1 occupies the first position in the array.

Depending on the value k and an odd value of n , a twist will always change the parity or not. We observe that for an even value of k (the default), only a twist on token 1 may change the parity.

Theorem 1 (Parity Anomaly in Top-Spin) *For an even value of k and odd value of $n > k + 1$, the (normalized) (n, k) Top-Spin Puzzle has $(n - 1)!/2$ reachable states.*

Proof. To ease notation, w.l.o.g., the proof is done for $k = 4$. Let $n = 2m + 1$ and $(x_0, x_1, \dots, x_{2m})$ be the normalized state vector. Thus, due to normalization, $x_0 = 0$. First of all, given that 0 is not counted, only three elements change their position and lead to 3 transpositions. For $(0, x_1, x_2, x_3, \dots, x_{2m})$ we have four critical successor states:

- $(x_3, x_2, x_1, 0, x_4 \dots x_{2m})$,
- $(x_2, x_1, 0, x_{2m}, x_3, \dots, x_{2m-1})$,
- $(x_1, 0, x_{2m}, x_{2m-1}, x_2, \dots, x_{2m-2})$, and
- $(0, x_{2m}, x_{2m-1}, x_{2m-2}, x_1, \dots, x_{2m-3})$.

In all cases, normalization has to move 3 elements either the ones with low index to the end of the array to postprocess the twist, or the ones with large indices to the start of the array to

preprocess the operation. The number of transpositions for one such move is $2m - 1$. In total we have $3(2m - 1) + 3$ transpositions. As each transposition changes the parity and the total of $6m$ transpositions is even, all critical cases have even priority. \square

As the parity is odd and even for a move in the (normalized) (n, k) Top-Spin Puzzle for an odd value of $n > k + 1$, we obtain the entire set of $(n - 1)!$ reachable states.

Pancake Problem

The n -Pancake Problem (Dweighter 1975) is to determine the number of flips of the first k pancakes (with varying $k \in \{1, \dots, n\}$) necessary to put them into ascending order. The problem has been analyzed e.g. by (Gates & Papadimitriou 1979). It is known that $(5n + 5)/3$ flips always suffice, and that $15n/14$ flips are necessary. In the burned pancake variant, the pancakes are burned on one side and the additional requirement is to bring all burned sides down. It is known that $2n - 2$ flips always suffice and that $3n/2$ flips are necessary. Both problems have n possible operators. The pancake problem has $n!$ reachable states, the burned one has $n!2^n$ reachable states. For an even value of $\lceil (k - 1)/2 \rceil$, $k > 1$ the parity changes, while for an odd one, the parity remains the same.

Efficient Ranking and Unranking

For ranking and unranking permutations, efficient time and space algorithms have been designed (Bonet 2008). For the design of a minimal perfect hash function for the sliding-tile puzzle, we observe that in a lexicographic ranking every two adjacent permutations π_{2i} and π_{2i+1} have a different solvability status.

Definition 2 (Lexicographic Rank, Inverted Index) *The lexicographic rank of permutation π (of size N) is defined as $\sum_{i=0}^{N-1} d_i \cdot (N - 1 - i)!$ where the vector coefficients d_i are called the inverted indices.*

The coefficients d_i are uniquely determined. The parity of a permutation is known to match $(\sum_{i=0}^{N-1} d_i) \bmod 2$.

In order to hash a sliding-tile puzzle state to $\{0, \dots, (nm)!/2 - 1\}$, we can compute the lexicographic rank and divide it by 2. Unranking is slightly more complex, as it has to determine, which of the two permutations π_{2i} and π_{2i+1} of the puzzle with index i is reachable.

Korf & Schultze (2005) use two lookup tables with a space requirement of $O(2^N \log N)$ bits to compute lexicographic ranks. Bonet (2008) discusses time-space trade-offs and provides a uniform algorithm that takes $O(N \log N)$ time and $O(N)$ space.

We observed that existing ranking and unranking algorithms wrt. the lexicographic ordering are rather slow. Hence, we study the more efficient ordering of Myrvold & Ruskey (2001) in more detail, and show that the parity of a permutation can be derived on-the-fly.² For faster execution

²In our results, we always refer to Myrvold and Ruskey’s rank1 and unrank1 functions.

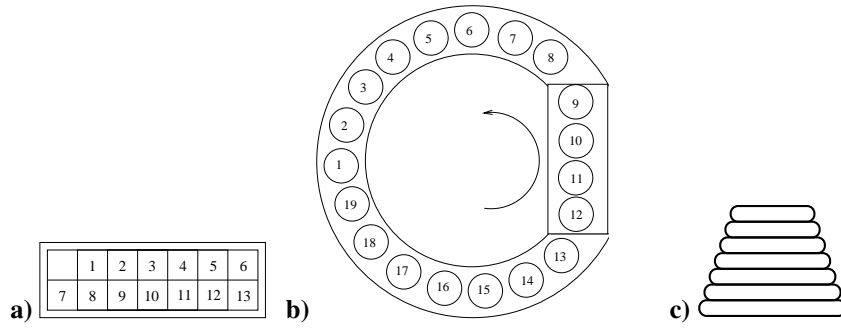


Figure 1: Permutation Games: a) Sliding Tile Puzzle, b) Top-Spin Puzzle c) Pancake Problem.

(on the graphics card), we additionally avoid recursion (see Alg. 1).

Algorithmus 1 $unrank(r)$

```

1:  $\pi := id$ 
2:  $parity := false$ 
3: while  $N > 0$  do
4:    $i := N - 1$ 
5:    $j := r \bmod N$ 
6:   if  $i \neq j$  then
7:      $parity := \neg parity$ 
8:      $swap(\pi_i, \pi_j)$ 
9:      $r := r \div N$ 
10:   $N := N - 1$ 
11: return  $(parity, \pi)$ 

```

Theorem 2 (Parity in Myrvold & Ruskey's Unrank)

The parity of a permutation for a rank r in Myrvold & Ruskey's ordering can be computed on-the-fly in the $unrank$ function depicted in Alg. 1.

Proof. In the $unrank$ function swapping two elements u and v at position i and j , resp., with $i \neq j$ we count $2(j - i - 1) + 1$ transpositions (u and v are the elements to be swapped, x is a wild card for any intermediate elements): $uxx \dots xxv \rightarrow xux \dots xxv \rightarrow \dots \rightarrow xx \dots xxw \rightarrow xx \dots xxvu \rightarrow \dots \rightarrow vxx \dots xxu$. As $2(j - i - 1) + 1 \bmod 2 = 1$, each transposition either increases or decreases the parity of the number of inversions, so that the parity toggles for each iteration. The only exception is if $i = j$, where no change occurs. Hence, the parity of the permutation can be determined on-the-fly in our algorithm. \square

Theorem 3 (Folding Myrvold & Ruskey) Let $\pi(r)$ denote the permutation returned by Myrvold & Ruskey's $unrank$ function given index r . Then $\pi(r)$ matches $\pi(r + N!/2)$ except for swapping π_0 and π_1 .

Proof. The last call to $swap(\pi_{N-1}, \pi_r \bmod N)$ in Myrvold and Ruskey's $unrank$ function is $swap(\pi_0, \pi_r \bmod 1)$, which resolves to either $swap(\pi_1, \pi_1)$ or $swap(\pi_1, \pi_0)$. Only the latter one induces a change.

If r_1, \dots, r_{N-1} denote the indices of $r \bmod N$ in the iterations $1, \dots, N - 1$ of Myrvold and Ruskey's $unrank$ function, then $r_{N-1} = \lfloor \dots \lfloor r / (N - 1) \rfloor \dots / 2 \rfloor$, which resolves to 1 for $r \geq N!/2$ and 0 for $r < N!/2$. \square

State Space Search on a Bitvector

For search with an implicit search frontier in form of a bitvector there are some hidden assumptions, so that we formalize required characteristics for hash functions.

Definition 3 (Hash Function) A hash function h is a mapping of some universe U to an index set $[0, \dots, m - 1]$.

The set of reachable states S is a subset of U , i.e., $S \subseteq U$. The first aspect are hash functions that are injective.

Definition 4 (Perfect Hash Function and Space Efficiency) A hash function $h : U \rightarrow [0, \dots, m - 1]$ is perfect, if for all $s \in S$ with $h(s) = h(s')$ we have $s = s'$. The space efficiency of h is the proportion $\lceil m / |S| \rceil$ of available hash values to states.

Given that every state can be viewed as a bitvector and interpreted as a number, one inefficient design of a perfect hash function is immediate. The space requirements of the corresponding hash table are usually too large. An space-optimal perfect hash function is bijective.

Definition 5 (Minimal Perfect Hash Function) A perfect hash function is minimal if its space efficiency $\lceil m / |S| \rceil = 1$.

Efficient and minimal perfect hash functions allow direct-addressing in a bit-state hash table instead of taking an open-addressed or chained hash table. The index uniquely identifies the state.³ Frontier search requires the locality of the search space (Zhou & Hansen 2006) being bounded.

³The approach of Botelho & Ziviani (2007) is applicable only if the state space has been seen before and requires some constant number of bits per state for storing the hash function. A perfect hash function does not have to be minimal to be efficient. There is recent research showing that space-efficient hash functions can be better than minimal ones, since the constant number of bits per state for the hash function becomes smaller than the loss in accuracy.

Whenever the averaged number of required bits per state for a perfect hash function is smaller than the number of bits in the state encoding, an implicit representation of the search space is advantageous, assuming that no other tricks like frontier search or orthogonal hashing are applied.

Definition 6 (Orthogonal Hash Functions) Hash functions h_1 and h_2 are orthogonal, if for all states s, s' with $h_1(s) = h_1(s')$ and $h_2(s) = h_2(s')$ we have $s = s'$.

Theorem 4 (Orthogonal Hashing implies Perfect Hashing) If the two hash functions $h_1 : U \rightarrow [0, \dots, m_1 - 1]$ and $h_2 : U \rightarrow [0, \dots, m_2 - 1]$ are orthogonal, their concatenation (h_1, h_2) is perfect.

Proof. We start with two hash functions h_1 and h_2 . Let s be any state in U . Given $(h_1(s), h_2(s)) = (h'_1(s), h'_2(s))$ we have $h_1(s) = h_1(s')$ and $h_2(s) = h_2(s')$. Since h_1 and h_2 are orthogonal, this implies $s_1 = s_2$. \square

In case of orthogonal hash functions, with small m_1 the value of h_1 can be encoded in the file name (Korf), leading to a partitioned layout of the search frontier, and a smaller hash value h_2 to be stored explicitly. Orthogonality can cooperate with bitvector representation of the search space, as function h_2 can be used as an index. For frontier search, the space-efficiency is smaller than with full state space memorization.

Definition 7 (Locality and Space Efficiency) The BFS locality (Zhou & Hansen 2006) l is defined as $\max\{\text{layer}(s') - \text{layer}(s) + 1 \mid s \in S; s' \in \text{successors}(s)\}$, where $\text{layer}(s)$ denotes the depth d of s in the BFS layer Layer_d . For frontier search, the space efficiency of $h : U \rightarrow [0, \dots, m - 1]$ is defined as $\lceil m / \max_d |\text{Layer}_d| + \dots + |\text{Layer}_{d+l}| \rceil$.

The other important property of a perfect hash function for an implicit state space is that the state vector can be reconstructed given the hash value.

Definition 8 (Invertible Hash Function) A perfect hash function h is invertible, if given $h(s)$, $s \in S$ can be reconstructed. The inverse h^{-1} of h is a mapping from $[0, \dots, m - 1]$ to S .

For an implicit encoding of the search space (Cooperman & Finkelstein 1992), in which array indices serve as state descriptors, invertible hash functions are required.

Definition 9 (Parity Preservation) A permutation problem is parity-preserving, if all moves preserve the parity of the permutation.

Parity-preservation allows to separate solvable from insolvable states in several permutation games. Examples are the sliding-tile puzzles, the Rubik's cube, the Top-Spin puzzle (with even k and odd n), and the Pancake problem (with k always being even). If the parity is preserved, the state space can be compressed.

Definition 10 (Move Alternation Property) A property p is move-alternating, if the parity of $p(s)$ for all actions from s to s' toggles, i.e., $p(s') \bmod 2 = (p(s) + 1) \bmod 2$.

As a result, $p(s)$ is the same for all states s in one BFS layer. In a mixed representation of two subsequent layers, states s' in the next BFS layer can be separated by knowing $p(s') = x \neq y = p(s)$. Two examples for a move-alternation property are the heuristic value or the index of the blank in the sliding tile puzzle. Moreover, many pattern database heuristics (?) have the property either to increase or to decrease by one with each move in the original space.

Two-Bit Breadth-First Search

In the domain of Caley graphs, Cooperman & Finkelstein (1992) show that, given a perfect and invertible hash function, two bits per state are sufficient to conduct a complete breadth-first exploration of the search space. The running time of their approach (shown in Alg. 2) is determined by the size of the search space times the maximum breadth-first layer, times the efforts to generate the children. Each node is expanded at most once. The algorithm uses two bit encoding numbers from 0 to 3, with 3 denoting an unvisited state, and 0,1,2 denoting the current depth value modulo 3. The main effect is that this allows to distinguish newly generated states and visited states from the current layer.

For non-minimal perfect hash functions, determining all reachable states is important to distinguish the good from the bad ones. This includes filtering of terminal states in two player games like tic-tac-toe. Here 5,478 states are reachable. A simple hash function maps tic-tac-toe positions to $|\{O, X, -\}|^9 = 19,683$. In this case, the efficiency is $\lceil 19,683/5,478 \rceil = 4$ so that this implicit representation is fortunate compared to explicit representations which need more bits per state.

A complete BFS traversal of the search space is very important for the construction of pattern databases. Korf (2008) has applied the algorithm to generate the state spaces for hard instances of the Pancake problem I/O efficiently.

Algorithmus 2 Two-Bit-Breadth-First-Search (init)

```

1: for all  $i := 0, \dots, N! - 1$  do
2:    $\text{Open}[i] := 3$ 
3:    $\text{Open}[\text{rank}(\text{init})] := \text{level} := 0$ 
4:   while Open has changed do
5:      $\text{level} := \text{level} + 1$ 
6:     for all  $i := 0, \dots, N! - 1$  do
7:       if  $\text{Open}[i] = (\text{level} - 1) \bmod 3$  then
8:          $\text{succs} := \text{expand}(\text{unrank}(i))$ 
9:         for all  $s \in \text{succs}$  do
10:          if  $\text{Open}[\text{rank}(s)] = 3$  then
11:             $\text{Open}[\text{rank}(s)] := \text{level} \bmod 3$ 

```

Two-bit breadth-first search indicates the use of bit-state tables for compressed pattern databases. If we store the mod-3 value of the BFS-level, we can determine its absolute value by backward construction of its generating path. One shortest path predecessor with mod-3 value of BFS-level k appears in level $k - 1 \bmod 3$. By having the BFS-level as the lookup value of the initial state, the pattern database lookup-values can then be determined incrementally.

One-Bit Reachability

The simplification in Algorithm 3 allows to generate the entire state space using one bit per state.

As we do not distinguish between open and closed nodes, the algorithm may expand a node multiple times. It is able to determine reachable states if a bijective hash function is present. Additional information extracted from a state can

Algorithmus 3 One-Bit-Reachability (*init*)

```
1: for all  $i := 0, \dots, N! - 1$  do
2:    $Open[i] := \text{false}$ 
3:    $Open[rank(init)] = \text{true}$ 
4:   while  $Open$  has changed do
5:      $i := 0, \dots, N! - 1$ 
6:     if  $Open[i] = \text{true}$  then
7:        $succs := \text{expand}(unrank(i))$ 
8:       for all  $s \in succs$  do
9:          $Open[rank(i) \bmod (N!/2)] := \text{true}$ 
```

improve the running time by decreasing the number of re-opened nodes.

If the successor's rank is smaller than the rank of the actual one, it will be expanded in the next scan, otherwise in the same.

Theorem 5 (Number of Scans in 1-Bit Reachability)

The number of scans in the algorithm One-Bit-Reachability is bounded by the maximum BFS layer.

Proof. Let $L_b(i)$ be the BFS-layer and $L_o(i)$ be the layer in the algorithm *One-Bit-Reachability*. Inductively, we have $L_o(i) \leq L_b(i)$. Evidently, $L_o(init) = L_b(init) = 0$. For any path (s_0, \dots, s_d) generated by BFS, we have $L_o(s_{d-1}) \leq L_b(s_{d-1})$ by induction hypothesis. All successors of s_d are generated in the same iteration (if their index value is larger) or in the next iteration (if their index value is smaller) such that $L_o(s_d) \leq L_b(s_d)$. \square

One-Bit Breadth-First Search

In cases with the move-alternation property (Def. 10), we can perform BFS using only one bit per state. We exemplify the considerations in the sliding-tile puzzles. We select the permutation ordering of Myrvold and Ruskey.

Since the parity does not change in this puzzle we need another alternating property, and find it in the position of the blank. The partition into buckets B_0, \dots, B_{nm-1} has the advantage that we can determine whether the state belongs to an odd or even layer and which bucket a successor belongs to (Zhou & Hansen 2004). Numbering the positions from 0 to $nm - 1$ and reducing the problem to puzzles with an odd number of columns we see that the successors of a state where the blank is at position 0 will have its blank either at position 1 (right move) or at position n (down move). For puzzles with an even number of columns the position of the blank also indicates the parity of the BFS layer. We observe that the blank position in puzzles with an odd number of columns at an even breadth-first level is even and for each odd breadth-first level it is odd.

For such a factored representation of the sliding-tile puzzles, a refined exploration in Alg. 4 retains the breadth-first order, by means that a bit for a node is set for the first time in its BFS layer. The bitvector $Open$ is partitioned into nm parts, which are expanded depending on the breadth-first *level* (line 7). The directions in which the blank can move (R-right, L-left, D-down, U-up, see line 9), are expanded in parallel using different threads.

As mentioned above, the rank of a permutation does not change by a horizontal move of the blank. This is exploited in line 11 writing the ranks directly to the destination bucket using a bitwise-or on the bitvector from layer *level* - 2 and *level*. The vertical moves are unranked, moved and ranked from line 13 onwards. When a bucket is done, the next one is skipped and the next but one is expanded. The algorithm terminates when no new successor is generated.

Algorithmus 4 One-Bit-Breadth-First-Search (*init*)

```
1: for  $blank = 0, \dots, nm - 1$  do
2:   for  $i = 0, \dots, (nm - 1)!/2 - 1$  do
3:      $Open[blank][i] := \text{false}$ 
4:    $Open[blank(init)][rank(init) \bmod (nm - 1)!/2] := \text{true}$ 
5:    $level := 0$ 
6:   while  $Open$  has changed do
7:      $blank := level \bmod 2$ 
8:     while  $blank \leq nm$  do
9:       for all  $d \in \{R, L, D, U\}$  do
10:         $dst := \text{newblank}(blank, d)$ 
11:        if  $d \in \{L, R\}$  then
12:           $Open[dst] := Open[dst] \text{ or } Open[blank]$ 
13:        else
14:          for all  $i$  with  $Open[blank][i] = \text{true}$  do
15:             $(valid, \pi) := \text{unrank}(i)$ 
16:            if  $\neg \text{valid}$  then
17:               $\text{swap}(\pi_0, \pi_1)$ 
18:               $\text{succ} := \text{expand}(\pi, d)$ 
19:               $r := \text{rank}(\text{succ}) \bmod (N - 1)!/2$ 
20:               $Open[dst][r] := \text{true}$ 
21:             $blank = blank + 2$ 
22:           $level = level + 1$ 
```

Even though some states are expanded several times, the following result is immediate. Let the population count pc_l of level l be the number of bits set after the l -th scan. Then the number of states in BFS-level l is $|Layer_l| = pc_l - pc_{l-1}$.

GPU Essentials

GPUs have multiple cores, but the programming and computational model are different from the ones on the CPU. Programming a GPU requires a special compiler, which translates the code to native GPU instructions. The GPU architecture mimics a single instruction multiple data (SIMD) computer with the same instructions running on all processors. It supports different layers for memory access, forbids simultaneous writes but allows concurrent reads from one memory cell.

If we consider the G200 chipset, as found in state-of-the-art NVIDIA GPUs and illustrated in Figure 2, a core is a streaming processor (SP) with 1 floating point and 2 arithmetic logic units. 8 SPs are grouped together with a cache structure and two special function units (performing e.g. double precision arithmetics) to one streaming multiprocessor (SM), and used like ordinary SIMD processors. Each of the 10 texture processor clusters (TSCs) combines 3 SMs, yielding 240 cores in one chip.

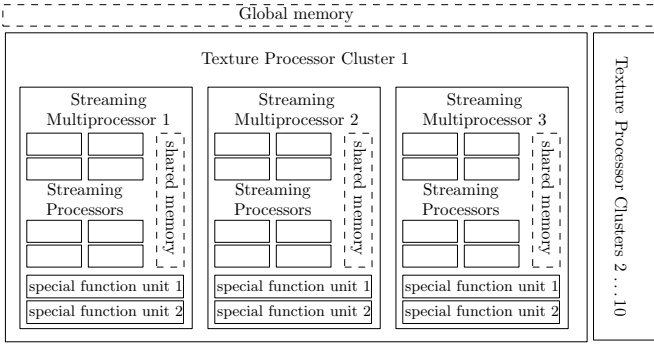


Figure 2: Sample GPU Architecture (G200 Chipset).

Memory, visualized dashed in the figure, is structured hierarchically, starting with the GPU’s global memory (video RAM, or VRAM). Access to this memory is slow, but can be accelerated through *coalescing*, where adjacent accesses with less than 64 bits are combined to one 64-bit access. Each SM includes 16 KB of memory (SRAM), which is shared between all SPs and can be accessed at the same speed as registers. Additional registers are also located in each SM but not shared between SPs. Data has to be copied from the systems main memory to the VRAM to be accessible by the threads.

The GPU programming language links to ordinary C-sources. The function executed in parallel on the GPU is called *kernel*. The kernel is driven by threads, grouped together in *blocks*. The TSC distributes the blocks to its SMs in a way that none of them runs more than 1,024 threads and a block is not distributed among different SMs. This way, taking into account that the maximal *blockSize* of 512, at most 2 blocks can be executed by one SM on its 8 SPs. Each SM schedules 8 threads (one for each SP) to be executed in parallel, providing the code to the SPs. Since all the SPs get the same chunk of code, SPs in an else-branch wait for the SPs in the if-branch, being idle. After the 8 threads have completed a chunk the next one is executed. Note that threads waiting for data can be parked by the SM, while the SPs work on threads, which have already received the data.

Port on the GPU

Let us consider porting the above algorithms on the GPU. To profit from coalescing, threads should access adjacent memory contemporary. Additionally, the SIMD like architecture forces to avoid if-branches and to design a kernel which will be executed unchanged for all threads. These facts lead to the implementation of keeping the entire or partitioned state space bitvector in RAM and copying an array of indices (ranks) to the GPU. This approach benefits from the SIMD technology but imposes additional work on the CPU. One additional scan through the bitvector is needed to convert its bits into integer ranks, but on the GPU the work to unrank, generate the successors and rank them is identical for all threads. To avoid unnecessary memory access, the rank given to expand should be overwritten with the rank of the first child. As the number of successors is known be-

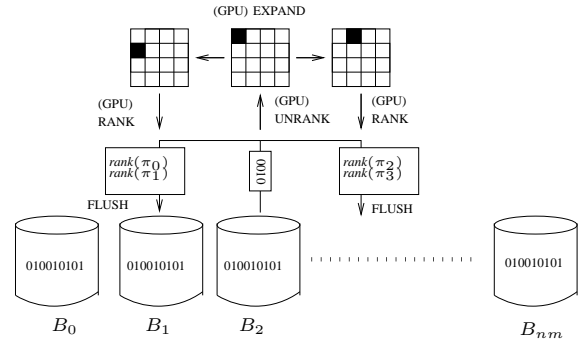


Figure 3: GPU Exploration of a Sliding-Tile Puzzle State Space Search stored as a Bitvector in RAM.

forehand, with each rank we reserve space for its successors. For smaller BFS layers this means that a smaller number of states is expanded.

In larger instances that exceed RAM capacities we additionally maintain write buffers to avoid random access on disk. Once the buffer is full, it is flushed to disk. In one streamed access, all corresponding bits are set.

The setting is exemplified for the sliding-tile puzzle domain in Figure 3. We see the “blank-partitioned” breadth-first state space residing on disk that is read into RAM, converted to integer ranks, copied to the GPU to be unranked, expanded and ranked again.

As a surplus, we used *pthreads* as additional multi-threading support. The partitioned state space was divided on multiple hard disks to increase the reading and writing bandwidth and to enable each thread to use its own hard disk.

To sample a move-alternation property in contrast to the blank’s position in the sliding-tile puzzles, the Manhattan distance heuristic value has been computed on the GPU by processing the unranked permutation. Even though the estimate can be computed incrementally in constant time, for the sake of generality we prefer computing it from scratch by cumulating absolute distances for all tiles.

The option of computing the heuristic value efficiently on the GPU suggests to also accelerate heuristic search. By the large reduction in state space due to the directedness of the search and by the lack of a perfect hash function for explored part of the state space in heuristic search – at least for simpler instances – a bitvector compression for the entire search space is not the most space-efficient option. However, as bitvector manipulation is fast, for hard instances we obtain runtime advances on the GPU.

For our case study we have ported breadth-first heuristic search (BFHS) (Zhou & Hansen 2006) to the GPU. For a given upper bound U on the optimal solution length and current BFS-level g the GPU receives the value $U - g$ as the maximal possible h -value, and marks states with larger h -value as invalid.

Experiments

We conducted the experiments on an AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ system with 4 GB RAM and

840 GB external storage, divided on 4 hard disks. The GPU we used is an NVIDIA N280GTX MSI with 1 GB VRAM and 240 cores.

For measuring the speed-up on a matching implementation we compare the GPU performance with a CPU emulation on a single core⁴.

Sliding-Tile Puzzle

The first set of experiments in Table 1 shows the gain of integrating bitvector state space compression with BFS in different instances of the Sliding-Tile puzzle.

We run the one- and two-bit breadth-first search algorithm on various instances of the sliding-tile-puzzle with RAM requirements from 57 MB up to 4 GB. The 3×3 version was simply too small to show significant advances, while even in partitioned form a complete exploration on a bit vector representation of the 15-Puzzle requires more RAM than available. Moreover, the predicted amount of 1.2 TB hard disk space is only slightly smaller than the 1.4 TB of frontier BFS search reported by Korf & Schultze (2005).

We first validated that all states were generated and equally distributed among the possible blank positions. Moreover, the numbers of BFS layers for symmetric puzzle instances match (53 for 3×4 and 4×3 as well as 63 for 2×6 and 6×2).

For the 2-Bit BFS implementation we observe a moderate speed-up by a factor between 2 and 3, which is due to the fact that the BFS layers of the instances are too small. For such small BFS layers, side processing like copying the indices to the VRAM is expensive compared to the gain achieved by parallel computation on the GPU. Unfortunately, the next larger instance (7×2) was too large for the amount of RAM in the machine (it needs $3 \times 750 \text{ MB} = 2250 \text{ MB}$ for *Open* and 2 GB for reading and writing indices to the VRAM).

In the 1-Bit BFS implementation the speed-up increases to a factor between 7 and 10 in the small instances. Many states are re-expanded in this approach, inducing more work for the GPU and exploiting its potential for parallel computation. Partitions being too large for the VRAM are split and processed in chunks of about 250 millions indices (for the 7×2 instance). A quick calculation shows that the savings of GPU computation are large. We noticed that the GPU has the capability to generate 83 million states per second (including unranking, generating the successors and computing their rank) compared to about 5 million states per second of the CPU. As a result, for the CPU experiment that ran out of time (o.o.t), which we stopped after one day of execution, we predict a speed-up factor of at least 16, and a running time of over 60 hours.

For BFHS, we measure the effect of computing the estimate together with the expansion on the GPU (see Table 2). For the puzzles we chose the hardest instances located in

⁴This way, the same code and work was executed on the CPU and the GPU. The emulation was run with one thread to minimize the work for thread communication on the CPU. In future releases of its compiler, NVIDIA will release an emulation mode, which utilizes multi-cores. So far we are forced to use only one core with the current emulation mode.

Table 1: Comparing GPU with CPU Performances in 1-Bit and 2-Bit BFS in the Sliding-Tile Puzzle Domain (times given in seconds, o.o.m means out of memory, while o.o.t denotes out of time).

Problem	2-Bit Time		1-Bit Time	
	GPU	CPU	GPU	CPU
(2×6)	70	176	163	1,517
(3×4)	55	142	98	823
(4×3)	64	142	104	773
(6×2)	86	160	149	1,110
(7×2)	o.o.m.	o.o.m.	13,590s	o.o.t.

Table 2: Comparing GPU with CPU Performances in 1-Bit BFHS in the Sliding-Tile Puzzle Domain (times given in seconds).

Problem	Rank	Blank	GPU Time	CPU Time
(2×6)	18,295,101	5	33	118
(3×4)	5,840,451	9	41	220
(4×3)	1,560,225	3	43	257
(6×2)	799,911	1	32	117
(2×7)	2,921,466,653	6	119	2,711

the deepest BFS layer from a previous BFS run as the initial state (its rank is provided in Table 2 column 2 and 3). The speed-up ranges in between 3 and 6 for small puzzle sizes, this can be attributed to the fact that for small problems the number of states copied to the GPU is limited. It scales up to 22 for large puzzles, where the effect of parallel computation is clearly visible. Even the additional burden of computing the Manhattan distance heuristic from scratch is negligible.

Top-Spin Problems

The results for the (n, k) -Top-Spin problems for a fixed value of $k = 4$ are shown in Table 3. We see that the experiments validate the theoretical statement of Theorem 1 that the state spaces are of size $(n - 1)!/2$ for n being odd⁵, and $(n - 1)!$ for n even. For large values of n , we obtain a significant speed-up of more than factor 30.

Pancake Problems

The GPU and CPU running time results for the n -Pancake problems are shown in Table 4. Similar to the Top-Spin puzzle for a large value of n , we obtain a speed-up factor of more than 30 wrt. running the same algorithm on the CPU.

Conclusion

In this paper we studied the application of GPU computation in selected AI search domains. We have shown how to apply

⁵At least the Top-Spin implementation of Rob Holte and likely the one of Ariel Felner/Uzi Zahavi do not consider parity compressed state spaces.

Table 3: Comparing GPU with CPU Performances in 2-Bit BFS in the Top-Spin Domain (times given in seconds).

n	States	GPU Time	CPU Time
6	120	0	0
7	360	0	0
8	5,040	0	0
9	20,160	0	0
10	362,880	0	6
11	1,814,400	1	35
12	39,916,800	27	920

Table 4: Comparing GPU with CPU Performances in 2-Bit BFS in Pankake Problems (times given in seconds).

n	States	GPU Time	CPU Time
9	362,880	0	4
10	3,628,800	2	48
11	39,916,800	21	641
12	479,001,600	290	9,187

GPU-based BFS, enjoy an implicitly encoded search frontier, and obtain significant speed-ups. The speed-ups of up to factor 30 and more compare well with speeding-up external-memory with parallel search on multiple cores (Korf & Schultze 2005; Zhou & Hansen 2007).

Two-bit BFS is applicable if invertible and perfect hash functions are available. One-bit reachability shows an interesting time-space trade-off, and one-bit BFS is applicable if a move alternation property can be obtained.

Due to the small amount of available shared RAM of 16 KB on the GPU, we prefer the space requirements for ranking and unranking to be small. To compute invertible minimal perfect hash functions for the permutation games, we studied took the more efficient one by Myrvold and Ruskey.

Future work will consider efficient ranking and unranking functions for general state spaces by forcing a BDD representing all reachable states to work as a perfect hash function. For simple reachability analysis this does not provide any surplus, but in case of more complex algorithms, like the classification of two-player games, perfect hash function based on BDDs show computational advantages in form of (internal or external) memory gains.

References

- Bonet, B. 2008. Efficient algorithms to rank and unrank permutations in lexicographic order. In *AAAI-Workshop on Search in AI and Robotics*.
- Botelho, F. C., and Ziviani, N. 2007. External perfect hashing for very large key sets. In *CIKM*, 653–662.
- Chen, T., and Skiena, S. 1996. Sorting with fixed-length reversals. *Discrete Applied Mathematics* 71(1–3):269–295.
- Cooperman, G., and Finkelstein, L. 1992. New methods for

using Cayley graphs in interconnection networks. *Discrete Applied Mathematics* 37/38:95–118.

Dweighter, M. 1975. Problem e2569. *American Mathematical Monthly* (82):1010.

Edelkamp, S., and Jabbar, S. 2006. Large-scale directed model checking LTL. In *SPIN*, 1–18.

Gates, W. H., and Papadimitriou, C. H. 1979. Bounds for sorting by prefix reversal. *Discrete Mathematics* 27:47–57.

Hordern, E. 1986. *Sliding Piece Puzzles*. Oxford University Press.

Korf, R. E., and Schultze, T. 2005. Large-scale parallel breadth-first search. In *AAAI*, 1380–1385.

Korf, R. E. 1997. Finding optimal solutions to Rubik’s Cube using pattern databases. In *AAAI*, 700–705.

Korf, R. E. 2003. Breadth-first frontier search with delayed duplicate detection. In *MoChArT*, 87–92.

Korf, R. E. 2008. Minimizing disk I/O in two-bit-breath-first search. In *AAAI*, 317–324.

Kunkle, D., and Cooperman, G. 2007. Twenty-six moves suffice for Rubik’s cube. In *ISSAC*, 235 – 242.

Munagala, K., and Ranade, A. 1999. I/O-complexity of graph algorithms. In *SODA*, 687–694.

Myrvold, W., and Ruskey, F. 2001. Ranking and unranking permutations in linear time. *Information Processing Letters* 79(6):281–284.

Owens, J. D.; Houston, M.; Luebke, D.; Green, S.; Stone, J. E.; and Phillips, J. C. 2008. GPU computing. *Proceedings of the IEEE* 96(5):879–899.

Zhou, R., and Hansen, E. A. 2004. Structured duplicate detection in external-memory graph search. In *AAAI*, 683–689.

Zhou, R., and Hansen, E. A. 2006. Breadth-first heuristic search. *AI* 170(4-5):385–408.

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *AAAI*, 1217–1222.