Parallel State Space Search on the GPU

Stefan Edelkamp
Am Fallturm 1
University of Bremen

Damian Sulewski
Otto-Hahn-Str. 14
Dortmund University of Technology

Abstract

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

Introduction

Since each modern processor contains at least two cores and graphics cards with hundreds of fragment processors are a commercial standard, one can expect an enormous impact on the programs that we will use. This "parallelism for the masses" offers great opportunities for state space search.

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 high arithmetic capabilities and memory bandwidths. Deployed on current graphic cards, GPUs have outpaced CPUs in numerical algorithms like matrix operations and Fourier transformations (Owens *et al.* 2008).

The GPU's rapid increase in both programmability and capability 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 in joint use. 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 for clusters (Edelkamp & Jabbar 2006) and/or multi-core processors (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 inversible and perfect hash function. It applies a space-efficient representation in breadth-first search with two frontier bits per state (Kunkle & Cooperman 2007), an idea that goes back to Cooperman & Finkelstein (1992).

Edelkamp & Sulewski (2008) investigated the exploitation of the GPU for accelerated delayed duplicate detection. Since moving vectors within the GPU's global memory is slow, the authors illustrate that existing GPU-sorting algorithms designed for floating point data often fail on sorting state vectors. As a bypass, they propose a refined bucket-sort algorithm on the GPU, that performs a hybrid of hash- and sorting-based duplicate detection.

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 permutation on the GPU and to how to compute the parity on-the-fly. To map the search space to a bit-vector, we study GPU-based two-bit BFS, and, for limited space, one-bit variants.

The structure of the paper is as follows. First, we recall GPU essentials to introduce the underlying computational model. Then, perfect hashing and devise rank and unrank functions for a selection of permutation puzzles. We reflect properties required for searching with a state space bitvector. For minimum perfect hashing, we have a closer look on the change of parity in theses games, and turn to integrating its computation for lexicographic and alternative orderings. We then turn to space-efficient state space search on a bitvector, including known variants like two-bit BFS and new variants that require only one bit per state. We then port the algorithms to the GPU and show its effectiveness in a range of experiments.

GPU Essentials

GPUs have multiple cores, but the programming and computational model are different from ones on the CPU. GPU

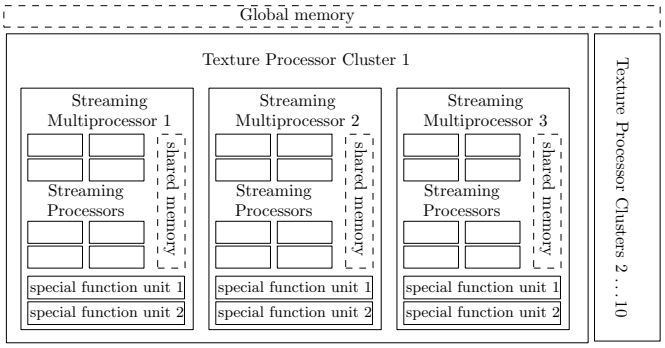

Figure 1: Sample GPU Architecture.

programming requires a special compiler, which compiles the code to native GPU instructions. The GPU architecture mimics a vector computer with the same function running on all processors. The architecture support different layers for accessing memory. GPUs forbid common writes to a memory cell but support some limited form of concurrent read.

The number of cores one the GPU exceeds the ones on the CPU, but they are limited to streamed processing. While cores on a multi-core processor work autonomously, the operation of cores on the GPU is different. For example, we observe that many if-then-else branching points or large branches in the GPU code lead to limited parallelism.

If we consider the G 200 chipset, as found in state-of-the-art NVIDIA GPUs, and illustrated in Fig. 1, a core is a simple 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 arithmetic) to a streaming multiprocessor. Each of the 10 texture processor clusters (TSCs) combines 3 SMs and a second cache, yielding 240 cores on one chip. Memory 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 shared RAM (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 they are not shared between SPs. Data has to be copied to 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 on its streaming multiprocessors in a way that none of the SMs runs more than 1,024 threads, and a block is not distributed among different SMs. This way, taking into account that the maximal *blockSize* is 512, at most 2 blocks can be executed by one SM on 8 SPs. Each TSC schedules 24 threads to be executed in parallel, providing the code to the SMs. 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 24 threads have executed the chunk the next kernels are executed. Note that threads which are waiting for data can be parked by the TSC, while the SPs work on threads, which have already received the data.

Perfect Hashing

A minimal perfect hash function is a one-to-one mapping from the state space S to the set $\{0, \ldots, |S| - 1\}$. Recent results (Botelho, Pagh, & Ziviani 2007; Botelho & Ziviani 2007) show that given the state space on disk, minimum perfect hash functions with a few bits per state can be constructed I/O efficiently. For many AI search problem domains, however, perfect hash functions and their inverses are available prior to the search. Examples are rank and unrank functions for permutation games, including the ones shown in Fig. 2. Permutation parity will be a helpful concept.

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

Rubik's Cube Rubik's Cube, invented in the late 1970s by Erno Rubik, is a known challenge for single-agent search (Korf 1997). Each face can be rotated by 90, 180, or 270 degrees and the goal is to rearrange a scrambled cube such that all faces are uniformly colored. Solvability invariants are that a single corner and a single edge sub-cube (cubi) cannot be twisted. Exchanging two cubis also destroys solvability. For the last issue the parity of the permutation is crucial and leads to $8! \cdot 3^7 \cdot 12! \cdot 2^{11}/2 = 43 \cdot 10^{18}$ solvable states.

Sliding-Tile Puzzle The $(n \times m)$ sliding-tile puzzle (Korf 2003) consists $(nm - 1)$ 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. A minimal perfect hash function for a sliding-tile problem has to map puzzle instances to a value in $\{0, \ldots, (nm)!/2 - 1\}$.

There is one subtle problem with the blank. Simply taking a minimum perfect hash the entire board does not suffice, as swapping a tile with the blank is a move and changes the priority. 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 $B_0, \ldots, B_{nm}$ denote the set of "blank-projected" partitions, then each set B_i contains $(nm - 1)!/2$ states. Given the index i as the permutation rank it is simple to reconstruct the puzzle's state. As a fortunate side effect, neither move of the blank to the left or right changes the state vector, such that for these moves we do not need any ranking or unranking.

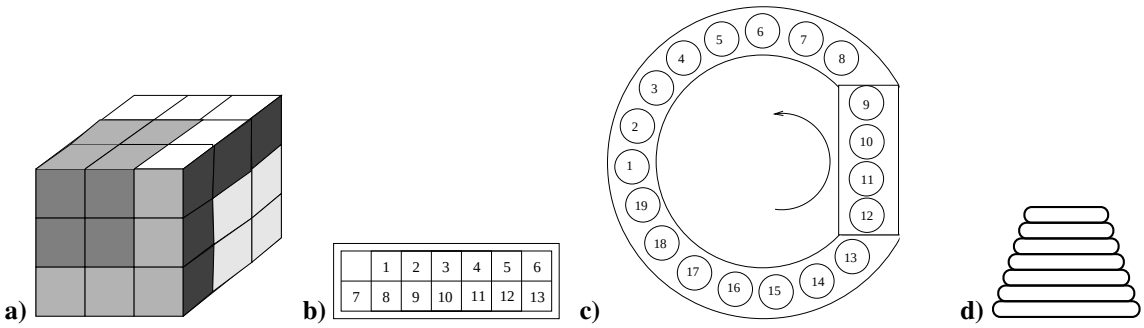

Figure 2: Permutation Games: a) Rubik's Cube, b) Sliding Tile Puzzle, d) Top-Spin Puzzle c) Pancake Problem.

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 relative location 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 moving token 0 to the start of the array, and mapping the rest.

Depending on value k, a twist will change the parity. We observe that for an even value of k (the default), only a twist on token 0 may change the parity.

Theorem 1 *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 $(x_0, x_1, \ldots, x_{2m})$ be the normalized state and, 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 transposition. For $(0, x_1, x_2, x_3, \ldots, x_{2m})$ we have four critical successor states:

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

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 $6m$ transposition is even, all critical cases have even priority. □

For even values of n some solutions checks the permutation parity at the beginning and toggle if it is odd.

Pancake Problem The n-Pankake Problem (Dweighter 1975) is to determine the number of flips of the first k pancakes (with varying $k \in \{1, \ldots, n)$ 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 pankake 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 neceessary. 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 and for an odd value $\lceil (k-1)/2 \rceil$ the parity remains the same.

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 different characteristics for hash functions.

Definition 2 (Hash Function) *A* hash function h *is a mapping of some universe U to an index set* $[0..m-1]$.

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 3 (Perfect Hash Function) *A hash function $h : U \rightarrow [0..m-1]$ 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 the every state can viewed as a bitvector and interpreted as a number, one inefficient design of a perfect hash functions is immediate. The space requirements of the corresponding hash table are usually too large. An optimal space-efficient perfect hash function is bijective.

Definition 4 (Minimal Perfect Hash Function) *A perfect hash function is* minimal*, if $|S| = m$.*

Efficient and minimal perfect hash functions allow direct-addressing a bit-state hash table instead of taking an open-addressed or chained hash table. The index uniquely identifies the state. Note that the approach in (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 function can be better than

minimal ones, since the constant number of bits per state for the hash function becomes smaller than the loss in accuricy.

Whenever the efficiency is smaller than the number of bits in the state encoding, an implicit representation of the search space is advantageous, assumming that no other tricks like frontier search or orthogonal hashing apply. Frontier search requires the locality of the search space is bounded.

Definition 5 (Locality) *The BFS locality is defined as* $\max\{layer(s) - layer(s') + 1 \mid s \in S; s' \in successors(s)\}$*, where* $layer(s)$ *denotes the depth* d *of* s *in BFS layer* $Layer_d$*. For frontier search, the space efficiency of* $h : U \to [0..m-1]$ *is defined as* $\lceil m / \max_d |Layer_d| + \ldots + |Layer_{d+l}| \rceil$.

Definition 6 (Orthogonal Hash Functions) *Two 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 2 (Orthogonal Hashing imply Perfect One) *If the two hash functions* $h_1 : U \to [0..m_1 - 1]$ *and* $h_2 : U \to [0..m_2 - 1]$ *are orthogonal, their concatenation* (h_1, h_2) *is perfect.*

Proof. We start with two hash 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 $s_1 = s_2$. □

In case of orthogonal hash functions, with small m_1 the value of h_1 can be encoded in the file name, leading to a partitioned layout of the search frontier, and a smaller hash value h_2 to be stored explicitly. Both orthogonality and frontier search mainly improve hardly cooperate with bitvector representation of the search space.

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

Definition 7 (Inversible Hash Function) *A perfect hash function* h *is* inversible*, if given* $h(s)$*,* $s \in S$ *can be reconstructed. The inverse* h^{-1} *of* h *is a mapping from* $[0..m-1]$ *to* S.

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

For permutation games, linear time and space algorithms compute the index (rank) and its inverse (unrank). For the design of a minimum perfect hash function for the sliding-tile puzzle, we observe that in a lexicographic ordering every two adjacent permutations π_{2i} and π_{2i+1} have a different solvability status.

Definition 8 (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 index*.*

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

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

Definition 9 (Parity Preserving) *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 heuristic often have the property either to increase or to decrease by one with each move in original space.

Efficient Ranking and Unranking with Parity

While experimenting with the GPU, we observed that existing ranking and unranking algorithms wrt. the lexicographic ordering are rather slow. Hence, we study the more efficient Myrvold Ruskey ordering in more detail, and show that the parity of a permuation can be derived on-the-fly.[1] For GPU execution, we additionally avoid recursion (see Alg. 1).

Algorithm 1 *unrank*(r)

```
1:  π := id
2:  parity := false
3:  while N > 0 do
4:      i := N − 1
5:      j := r mod N
6:      if i ≠ j then
7:          parity := ¬parity
8:          swap(π_i, π_j)
9:          r := r div N
10:     N := N − 1
11: return (parity, π)
```

Theorem 3 *The parity of a permutation for an rank* r *in the Myrvold & Ruskey's ordering can be computed on-the fly in the unrank function depicted in Alg. 1.*

[1] In our results, we always refer to Myrvold and Ruskey's rank1 and unrank1 functions.

Proof. In the *unrank* function we always have $N-1$ element exchanges. For 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): $uxx \ldots xxv \rightarrow xux \ldots xxv \rightarrow \ldots \rightarrow xx \ldots xxuv \rightarrow xx \ldots xxvu \rightarrow \ldots \rightarrow vxx \ldots xxu$. As $2(j-i-1)+1 \ mod \ 2 = 1$, 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 Myrvold & Ruskey's algorithm. □

Theorem 4 *Let $\pi(r)$ denote the value returned by Myrvold & Ruskey's* unrank *function given index r. Then $\pi(r)$ matches $\pi(r + N!/2)$ except of swapping π_0 with π_1.*

Proof. The last call to *swap*$(\pi_{N-1}, \pi_{r \ mod \ N})$ in Myrvold and Ruskey's *unrank* function is *swap*$(\pi_0, \pi_{r \ mod \ 1})$, which resolves to either *swap*(π_1, π_1) or *swap*(π_1, π_0). Only the latter one induces a change.

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

Two-Bit Breadth-First Search

In the domain of Caley graphs, Cooperman & Finkelstein (1992) show that, given a perfect and inversible 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 bits 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-minmimal perfect hash functions, determining all reachable states is important to distinguish to good for the bad ones. This includes filtering of terminal states in two player games like tic-tac-toe. For tic-tac-toe 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 an implicit representation is fortunate. Once generated, retrograde analysis can be applied, calling for another few 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.

One-Bit Reachability

The simplification in Algorithm 3 allows to generate the entire state space using one bit. If the successor's position is smaller than the actual one, it will be expanded in the next run, otherwise in the same run.

Algorithm 2 Two-Bit-Breadth-First-Search (*init*)

```
 1: for all i := 0, ..., N! − 1 do
 2:    Open[i] := 3
 3: Open[rank(init)] := level := 0
 4: while Open has changed do
 5:    level := level + 1
 6:    for all i := 0, ..., N! − 1 do
 7:       if Open[i] = (level − 1) mod 3 then
 8:          succs := expand(unrank(i))
 9:          for all s ∈ succs do
10:             if Open[rank(s)] = 3 then
11:                Open[rank(s)] := level mod 3
```

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

```
1: for all i := 0, ..., N! − 1 do
2:    Open[i] := false
3: Open[rank(init)] = true
4: while Open has changed do
5:    i := 0, ..., N! − 1
6:    if Open[i] = true then
7:       succs := expand(unrank(i))
8:       for all s ∈ succs do
9:          Open[rank(i) mod (N!/2)] := true
```

As we do not distinguish between open and closed nodes, the algorithm may expand a node multiple times.

Theorem 5 *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 $(s_0, \ldots, s_d)$ to the state with index i 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 $L_o(s_d) \leq L_b(s_d)$ □

One-Bit Breadth-First Search

In the advent of a 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 Myworld and Ruskey.

The partition $B_0, \ldots, B_{nm}$ into buckets has the additional advantage that we can determine, which bucket a successor belongs to (Zhou & Hansen 2004). Moreover, 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 partioned 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 to write the ranks directy 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.

Algorithm 4 One-Bit-Breath-First-Search (*init*)

```
 1: for blank = 0, ..., nm do
 2:   for i = 0, ..., (nm − 1)!/2 − 1 do
 3:     Open[blank][i] := false
 4: Open[blank(init)][rank(init) mod (nm−1)!/2] := true
 5: level := 0
 6: while Open has changed do
 7:   blank := level mod 2
 8:   while blank ≤ nm do
 9:     for all d ∈ {R, L, D, U} do
10:       dst := newblank(blank, d)
11:       if d ∈ {L, R} then
12:         Open[dst] := Open[dst] or Open[blank]
13:       else
14:         for all i with Open[blank][i] = true do
15:           (valid, π) := unrank(i)
16:           if ¬valid then
17:             swap(π_0, π_1)
18:           succ := expand(π, d)
19:           r := rank(succ) mod (N − 1)!/2
20:           Open[dst][r] := true
21:     blank = blank + 2
22:   level = level + 1
```

Even though some states are expanded several times the following result is immediate.

Theorem 6 *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 $|Layer_l| = pc_l - pc_{l-1}$.*

Port on the GPU

Let us consider porting the above algorithms on the GPU. After some initial experiments, we chose the design to keep the entire or partitioned state space bitvector in RAM and to move the array indices (ranks) to the GPU. For smaller BFS layers this means that a smaller amount of states are expanded. As the number of successors is known in advance with each rank value we reserve space for its successors.

In larger instances that exceed main memory capacities, we additionally maintain write buffers in RAM to avoid random access on disk. Once the buffer is full, it is flushed to disk. Then, in one streamed access, all corresponding bits are set. We first thought that sorting the ranks on the GPU with GPU sorting was faster, but then resort to flusing bitvector partitions.

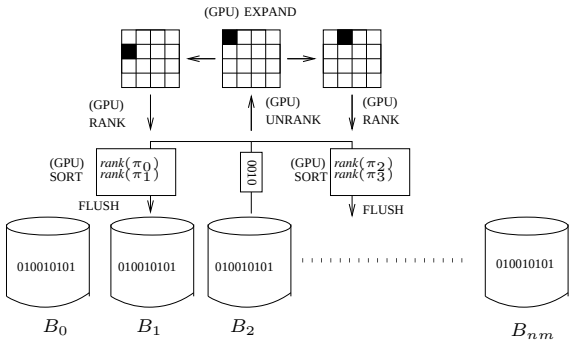

Figure 3: GPU Exploration of a Sliding-Tile Puzzle State Space Search stored as a Bitvector in RAM. (GPU sorting the indices to flush is optional and was not used in the experiments).

The setting is exemplified for the sliding-tile puzzle domain in Fig. 3. We see the "blank-partitioned" breadth-first state space residing on disk that is read into RAM and moved to the GPU to be unranked, expanded and ranked again.

We have used pthreads as an additional multi-threading support. The partitioned state space was divided on multiple hard disks to increase the reading and writing bandwith and to enable threads to use its own hard disk.

To sample a move-alternation property different to the blank's position, in the sliding-tile puzzles the Manhattan distance heuristic value has be 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.

Breadth-First Heuristic Search

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 for the set of reachable state space in heuristic search, at least for simpler instances bitvector compression for entire search space is often not the most space-efficient option. However, as bitvector manipulation is fast, for hard instances we have obtained runtime advances wrt. single core experiments.

A disk-based variant of A* with delayed duplicate detectionis due to Edelkamp, Jabbar, & Schrödl (2004). Alternatives are frontier A* by Korf 2004, and external-memory breadth-first heuristic search by Zhou & Hansen 2005.

For our case study, we have ported breadth-first heuristic search (BFHS) 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+ sytem with 4 GB RAM and 840 GB storing space, divided on 4 hard disks. The GPU

used was a NVIDIA N280GTX MSI with 1GB VRAM and 240 cores.

Rubik's Cube

Unfortunately, we have not been able to conduct an experiment in Rubik's cube. Assuming one bit per state, an impractical amount of 4.68 exabytes storage for performing full reachability. For generating upper bounds, however, bitvector representations of subspaces have been shown to be efficient (Kunkle & Cooperman 2007).

Sliding-Tile Puzzle

The first set of experiments in Table 1 shows the gain of integrating bitvector state space compression with in 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 versio 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).

For measuring the speed-up on a matching implementation we compare the GPU performance with a CPU emulation on a single core[2].

We first validated that all states were generated and equally distributed among the possible blank positions. Moreover, the maximum BFS layer for symmetric puzzles matched (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 3×750 MB = 2250 MB for *Open* and 2 GB for reading and writing indices to the VRAM).

For 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 exploits its potential of 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.

[2]This 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 support a multi-core emulation mode, so far we are deemed to use the single core one.

	2-Bit		1-Bit	
Problem	Time GPU	Time CPU	Time GPU	Time CPU
(2×6)	70s	176s	163s	1517s
(3×4)	55s	142s	98s	823s
(4×3)	64s	142s	104s	773s
(6×2)	86s	160s	149s	1110s
(7×2)	o.o.m.	o.o.m.	13590s	o.o.t.

Table 1: Comparing CPU with GPU Performances in 1 and 2-Bit BFS in the Sliding-Tile Puzzle Domain.

Problem	Index	Blank	Time GPU	Time CPU
(2×6)	18295101	5	33s	118s
(3×4)	5840451	9	41s	220s
(4×3)	1560225	3	43s	257s
(6×2)	799911	1	32s	117s
(2×7)	2921466653	6s	119s	2711s

Table 2: Comparing CPU with GPU Performances in 1-Bit BFHS in the Sliding-Tile Puzzle Domain.

For BFHS, we measure the effect of computing the estimate together with the expansion on the GPU (see Table 2). For the puzzles we choose hardest instances located in the deepest BFS layer from a previous BFS run as the initial state (its rank is provided in Table 2). The speed-up ranges in between 3 and 6 for small puzzle sizes and scales to 22 for large puzzles. This can be attributed to the fact that for small problems the number of states copied to the GPU is limited. There, the effect of parallel computation is clearly visible and we also notice that the additional burden of computing the Manhattan distance heuristic from scratch is neglectible.

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 an odd value of n[3]. For an even value we have $(n-1)!$ as expected. For large values of n, we obtain a significant speed-up of GPU computation 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 some AI search domains. We show how to apply GPU-

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

n	States	Time GPU	Time CPU
6	120	0s	0s
7	360	0s	0s
8	5040	0s	0s
9	20160	0s	0s
10	362880	0s	6s
11	1814400	1s	35s
12	39916800	27s	920s

Table 3: Comparing CPU with GPU Performances in 2-Bit BFS in the Top-Spin Domain.

n	States	Time GPU	Time CPU
9	362880	0s	4s
10	3628800	2s	48s
11	39916800	21s	641s
12	479001600	290s	9187s

Table 4: Comparing CPU with GPU Performances in 2-Bit BFS in Pankake Problems.

based BFS, enjoy an implicitly encoded search frontier, and obtain significant speed-ups.

Two-bit BFS is applicable if inversible and perfect hash functions are available. One-bit reachability shows an interesting time-space trade-off, and one-bit BFS is applicable if we can find an efficient move-alternation property.

The speed-ups of up to factor 30 and more compare well with speeding-up external-memory with parallel search on multiple cores and/or clusters (Korf & Schultze 2005; Zhou & Hansen 2007; Edelkamp & Jabbar 2006).

To compute inversible minimal perfect hash functions for the permutation games, we studied two different orders and took and extended the more efficient one by Myrvold and Ruskey. For parallel expansion on the GPU, due to the little amount of available shared RAM of 16K, we preferred the space requirements for ranking and unranking to be small.

The one- and two-bit breadth-first search results indicate the use of bit-state tables to compress pattern databases in regular domains. Using two bits per state and by storing 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 $k-1$ *mod* 3. Having the BFS-level as the lookup value of the initial state, the pattern database lookup-values can then be determined incrementally.

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.

Botelho, F. C.; Pagh, R.; and Ziviani, N. 2007. Simple and space-efficient minimal perfect hash functions. In *WADS*, 139–150.

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 *Model Checking Software (SPIN)*, 1–18.

Edelkamp, S., and Sulewski, D. 2008. Model checking via delayed duplicate detection on the GPU. Technical Report 821, University of Dortmund.

Edelkamp, S.; Jabbar, S.; and Schrödl, S. 2004. External A*. In *German Conference on Artificial Intelligence (KI)*, 233–250.

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

Korf, R. E., and Schultze, T. 2005. Large-scale parallel breadth-first search. In *National Conference on Artificial Intelligence (AAAI)*, 1380–1385.

Korf, R. E. 1997. Finding optimal solutions to Rubik's Cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI)*, 700–705.

Korf, R. E. 2003. Breadth-first frontier search with delayed duplicate detection. In *Model Checking and Artificial Intelligence (MOCHART)*, 87–92.

Korf, R. E. 2004. Best-first frontier search with delayed duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*, 650–657.

Korf, R. E. 2008. Minimizing disk I/O in two-bit-breath-first search. In *National Conference on Artificial Intelligence (AAAI)*, 317–324.

Kunkle, D., and Cooperman, G. 2007. Twenty-six moves suffice for Rubik's cube. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, 235 – 242.

Munagala, K., and Ranade, A. 1999. I/O-complexity of graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, 687–694.

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 *National Conference on Artificial Intelligence (AAAI)*, 683–689.

Zhou, R., and Hansen, E. A. 2005. External-memory pattern databases using structured duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*, 1398 – 1405.

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*, 1217–1222.