Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)

Perfect Hashing for State Space Exploration on the GPU

Stefan Edelkamp
TZI, Universität Bremen
edelkamp@tzi.de

Damian Sulewski
TZI, Universität Bremen
dsulews@tzi.de

Cengizhan Yücel
Technische Universität Dortmund
cengizhan.yuecel@googlemail.com

Abstract

This paper exploits parallel computing power of graphics cards to accelerate state space search. We illustrate that modern graphics processing units (GPUs) have the potential to speed up 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 explored in order to achieve enhanced compression. We report maximal speed-ups of up to a factor of 27 wrt. single core CPU computation.

Introduction

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, and programmable processors featuring complex arithmetic computing and high memory bandwidths. Deployed on current graphics cards, and conveniently accessible, e.g., through CUDA (Lindholm et al. 2008) NVIDIA's programming environment, GPUs have outpaced CPUs in numerical algorithms such as Fourier transformations (Owens et al. 2008) or sorting (Leischner, Osipov, and Sanders 2009).

To tackle the intrinsic hardness of large search problems, sparse-memory and disk-based algorithms are often used together. I/O-efficient breadth-first search (BFS) has been suggested for explicit undirected graphs by Munagala and Ranade (1999) and for implicit graphs by Korf (2003). The duplicate detection schemes in these frontier search algorithms can either be delayed or structured (Zhou and Hansen 2004). Especially on multiple disks, instead of the 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 has been studied (Korf and Schultze 2005; Zhou and Hansen 2007).

External two-bit breadth-first search by Korf (2008) integrates the RAM compression method by Cooperman and Finkelstein (1992) into an I/O-efficient algorithm. The approach for solving large-scale problems relies on reversible perfect hash functions. It applies a space-efficient representation with two bits per state e.g. to prove lower bounds in Rubik's Cube (Kunkle and Cooperman 2007).

Due to the large number of processors, memory access is a bottleneck on current graphics cards. All data accessed by the GPU have to be copied to the card, however, the amount of memory is very limited when compared to the system's RAM. GPUs are designed for high throughput, while latency is the main guiding principle in a CPU's design.

Based on the two-bit approach by Cooperman and Finkelstein (1992), we put forward two algorithms: a generalized one-bit reachability algorithm, and a one-bit BFS algorithm. We show how to speed up the search by parallelizing the algorithms and computing reversible minimal perfect hash functions on the GPU. We analyze our approach on several perfect hash functions, ranging from permutations to binomial coefficients for solving search problems.

The paper is structured as follows. Firstly, the used perfect hashing functions are introduced and extended. For minimal perfect hashing we have a closer look on the change of parity in some 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. We recall GPU essentials and the underlying computational model. Execution of the algorithms on the GPU is explained next, and its effectiveness is shown with the help of a wide range of experiments.

Perfect Hashing Functions

For the search in a bitvector we require certain characteristics of hash functions.

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

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

Definition 2 (Perfect Hash Function) *A hash function* $h : U \to [0, \ldots, m-1]$ *is* perfect*, if for all* $s \in S$ *with* $h(s) = h(s')$ *we have* $s = s'$.

Given that every state can be viewed as a bitvector, and, in turn, be interpreted as a number in binary, a simple but space-inefficient design for a perfect hash function would be to use this number as a hash value.

Definition 3 (Space Efficiency) *The space efficiency of* h *is the proportion* $\lceil m/|S| \rceil$ *of available hash values to states.*

Definition 4 (Minimal Perfect Hash Function) *A perfect hash function h is* minimal *if its space efficiency is* 1.

A minimal perfect hash function is a one-to-one mapping from the state space S to the set of indices $\{0, \ldots, |S| - 1\}$, i.e., $m = |S|$. In contrast to open-addressed or chained hash table, perfect hash functions allow direct-addressing of bit-state hash tables. This allows compressing the set of visited states *Closed*. The other important property is that the state vector can be reconstructed given the hash value, which allows us to also compress the list of frontier nodes *Open*.

Definition 5 (Reversible Hash Function) *A perfect hash function h is* reversible*, if given $h(s)$, the state $s \in S$ can be reconstructed. A reversible minimum perfect hash function is called* rank*, while the inverse is called* unrank*.*

We will see that for a traversal of the search space in which array indices serve as state descriptors, reversible hash functions are required.

Definition 6 (Move-Alternation) *Property $p : S \to \{0, 1\}$ is* move-alternating*, if it toggles for all applied actions.*

In other words, for all successors s' of s, we have $p(s') = 1 - p(s)$. As a result, $p(s)$ is the same for all states s in one BFS layer, so that states s' in the next BFS layer can be separated from the ones in the current one, exploiting $p(s') = x \neq y = p(s)$. A stronger criterion is the following one.

Definition 7 (Layer-Selection) *A property $p : S \to \mathbb{N}$ is* layer-selecting*, if it determines the BFS-layer for a state, in other words $p(s) =$ BFS-layer(s).*

An example is the number of unoccupied holes in the *Peg-Solitaire* game. In some cases perfect hash functions can be partitioned along the properties (Korf and Schultze 2005).

Definition 8 (Orthogonal Hash Functions) *Two hash functions h_1 and h_2 are* orthogonal*, if for all states $s, s' \in S$ with $h_1(s) = h_1(s')$ and $h_2(s) = h_2(s')$ we have $s = s'$.*

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

Proof. We start with two hash functions h_1 and h_2. Let s, s' be any states in S. 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 $s = s'$. □

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 state space, and a smaller hash value h_2 to be stored explicitly. Orthogonality can cooperate with bitvector representation of the search space, as a perfect function h_2 can be used as a secondary index.

Definition 9 (BFS-Partitioning) *A perfect hash function h is* alternation partitioning*, if there is a move-alternation property p that is orthogonal to h. A perfect hash function h is* layer partitioning*, if there is a layer-selection property p that is orthogonal to h.*

For a given perfect hash function h for the full state space this leads to further compression, and in some cases memory advances when applying frontier search, depending on the *locality* of the search space (Zhou and Hansen 2006).

If p is a move-alteration property, we can partition S into parts $S_0 = \{s \mid p(s) = 0\}$ and $S_1 = \{s \mid p(s) = 1\}$ with $S_0 \cup S_1 = S$ and $S_0 \cap S_1 = \emptyset$, such that $h(s_0) < h(s_1)$ for $(s_0, s_1) \in S_1 \times S_2$. This defines two bit-vector compression functions $h_0(s) = h(s)$ and $h_1(s) = h(s) - |\{h(s) \mid s \in S_0\}|$ that can be used in odd and even layer of the search. Similarly, we can extend the observation to hash functions $h_1, h_2, h_3, \ldots$, if p is layer partitioning.

Belazzougui, Botelho, and Dietzfelbinger (2009) show that, given a state space, minimal perfect hash functions with a few bits per state can be constructed I/O efficiently.

Unfortunately, the hash functions are hardly reversible. For many domain-independent problem domains, however, perfect hash functions (and their inverses) can be derived.

Permutation Rank as a Hash Function

For the design of rank and unrank functions for permutation games (including the ones shown in Figure 3) parity is a crucial concept.

Definition 10 (Parity) *The* parity *of the permutation π of length N is defined as the parity of the number of inversions in π, where inversions are all pairs (i, j) with $0 \leq i < j < n$ and $\pi_i > \pi_j$.*

Definition 11 (Parity Preservation) *A permutation problem is* parity preserving*, if all moves preserve the parity of the permutation.*

Parity-preservation allows separating solvable from insolvable states in several permutation games. Examples are the sliding-tile and the (n, k) Top-Spin puzzles (with even value of k and odd value of n). If the parity is preserved, the state space can be compressed.

Korf and Schultze (2005) use two lookup tables with a space requirement of $O(2^N \log N)$ bits to compute lexicographic ranks (and their inverse). Bonet (2008) discusses time-space trade-offs and provides a uniform algorithm that takes $O(N \log N)$ time and $O(N)$ space. As we are not aware of any $O(N)$ time and $O(N)$ space algorithm for lexicographic rank and unrank, we studied the ordering induced by the *rank1* and *unrank1* functions from Myrvold and Ruskey (2001).

Algorithm 1 illustrates that the parity of a permutation can be derived on-the-fly. For faster execution (especially on the graphics card), we additionally avoid recursion.

Theorem 2 (Parity in Myrvold & Ruskey's Unrank) *The parity of a permutation for a rank according to Myrvold & Ruskey's ordering can be computed on-the fly in the unrank function shown in Algorithm 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 acts as a wild card): $uxx \ldots xxv \to xux \ldots xxv \to \ldots \to xx \ldots xxuv \to xx \ldots xvxu \to \ldots \to vxx \ldots xxu$. As $2(j-i-1)+1 \bmod 2 = 1$, each transposition either increases

Algorithm 1 *Permutation-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: if ¬parity then
12:   swap(π_0, π_1)
13: return π
```

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 is determined on-the-fly in Algorithm 1. □

Theorem 3 (Folding Myrvold & Ruskey) *Let $\pi(r)$ denote the permutation computed by Algorithm 1 given index r at line 11. Then $\pi(r) = \pi(r + N!/2)$ except of 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*(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 $r \bmod N$ 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 $r < N!/2$. □

Hashing with Binomial Coefficients

For states consisting of a fixed number of Boolean variables, it suffices to store only the variables that are assigned *true*, in order to identify each state. Traversing the search graph and generating successors flips the status of individual state variables depending on the successor generating function.

If the order of the variables is fixed and the number of satisfied bits are given, we can identify their position using a *binomial coefficient*. A binomial coefficient $\binom{n}{k}$ is the number of possible k-sets in a set of n elements. Algorithm 2 describes how to assert a unique rank to a given state. Since the number of k-sets in a n-set is known, we can impose an ordering on these k-sets. This ordering is given by the position of the variables that are satisfied. The algorithm starts with a rank $r = 0$ and uses the variable t to count the number of satisfied variables. For each unsatisfied variable r is increased by the binomial coefficient given by the position of this entry and the number of the remaining satisfied variables.

The according unrank function is displayed in Algorithm 3. For proving the correctness, we show surjectivity and injectivity.

Theorem 4 (Surjectivity) *For each $s \in S$ with t satisfied variables, the maximal rank is bounded by $r(s) \leq \binom{n}{n-t} - 1$.*

Proof. Value r is increased only if no satisfied variable is found. For all $n, t \in \mathbb{N}$ with $n \geq t$, we have $\binom{n}{t} \geq \binom{n-1}{t}$. We get the maximal rank when the first $n - t$ variables are not satisfied, resulting in $r = \binom{n-1}{t-1} + \ldots + \binom{n-t}{t-1}$. Using the known characteristic of binomial coefficients $\sum_{i=0}^{m} \binom{n+i}{n} = \binom{n}{n} + \ldots + \binom{n+m}{n} = \binom{n+m+1}{n+1}$ we have $\sum_{i=0}^{t} \binom{(n-t)-1+i}{(n-t)-1} = \binom{n}{n-t}$, and, by applying Algorithm 2, we deduce $\binom{n-t}{(n-t)-1} + \ldots + \binom{n-1}{(n-t)-1} = \binom{n}{n-t} - \binom{(n-t)-1}{(n-t)-1} = \binom{n}{n-t} - 1$. □

Theorem 5 (Injectivity) *Given a number of satisfied variables t binomial ranking induces a collision free hash function, such that for all $s, s' \in S_t$, we have $s \neq s'$ implies $r(s) \neq r(s')$.*

Proof. Assume the contrary. Then there exists $s, s' \in S_t$ with $s \neq s'$ and $r(s) = r(s')$. Since s and s' are different, an entry at a minimal position i exists with $s_i \neq s'_i$. If w.l.o.g s_i is not satisfied, equation $r(s_{0..i}) = r(s'_{0..i}) + \binom{n-i}{t_i-1}$ applies, where t_i is the number of satisfied elements left, but the maximal increase for $r(s')$ is, due to the same arguments as in Theorem 4, $\binom{(n-j)-1}{t_i-2} + \ldots + \binom{t_i-1}{t_i-2} = \binom{n-j}{t_i-1} - 1$. □

Algorithm 2 *Binomial-Rank*(s)

```
1: i := 0; r := 0
2: t := number of true values in s
3: while t > 0 do
4:    i := i + 1
5:    if s_i = 1 then
6:       t := t − 1
7:    else
8:       r := r + C(n−i, t−1)
9: return r
```

Algorithm 3 *Binomial-Unrank*(r, t)

```
1: i := 0
2: while t > 0 do
3:    if r < C(n−i−1, t−1) then
4:       s_i := true
5:       t := t − 1
6:    else
7:       s_i := false
8:       r := r − C(n−i−1, t−1)
9:    i := i + 1
10: while i < |s| do
11:    s_i := false
12:    i := i + 1
13: return s
```

Bitvector State Space Search

Minimal and reversible perfect hash functions are bijective mappings of the set of reachable states to a set of available indices.

Two-Bit Breadth-First Search

Cooperman and Finkelstein (1992) show that, given a minimal reversible perfect 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. 4) is determined by the size of the search space times the maximum breadth-first layer (times the efforts to generate the children). The algorithm uses two bits, encoding numbers from 0 to 3, with 3 denoting an unvisited state, and 0, 1, and 2 denoting the current depth value modulo 3. The main effect is that this allows us to separate newly generated states and visited states from the current layer.

Algorithm 4 *Two-Bit-BFS* (*init*)

```
 1: for all i := 0, ..., |S| − 1 do
 2:     Open[i] := 3
 3: Open[rank(init)] := layer := 0
 4: while Open has changed do
 5:     layer := layer + 1
 6:     for all i := 0, ..., |S| − 1 do
 7:         if Open[i] = (layer − 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)] := layer mod 3
```

Two-bit breadth-first search suggests the use of bit-state tables for compressing pattern databases (Breyer and Korf 2009). If we store the mod-3 BFS value, we can determine its absolute value first by construction of its generating path and then by incremental updates. By having the BFS-layer as the lookup value of the initial state, the pattern database lookup-values can be determined incrementally.

One-Bit Reachability

The simplification in Algorithm 5 allows us to generate the entire state space while using only one bit per state.

Algorithm 5 *One-Bit-Reachability* (*init*)

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

As we do not distinguish between *open* and *closed* nodes, the algorithm may expand a node several times. Worse, in one iteration of the algorithm, states in different BFS-layers can be expanded. 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 one. Nevertheless, if a bijective hash function is available, the algorithm is able to determine all reachable states. Additional information extracted from a state can improve the running time by decreasing the number of reopened nodes.

Theorem 6 (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 *Layer*(i) be the breadth-first search layer of a state with rank i and *Scan*(i) be the layer in the algorithm *One-Bit-Reachability*. Evidently, *Scan*(*rank*(*init*)) = *Layer*(*rank*(*init*)) = 0. For any path $(s_0, \ldots, s_d)$ generated by BFS, we have *Scan*(*rank*$(s_{d-1})) \leq$ *Layer*(*rank*$(s_{d-1}))$ by induction hypothesis. All successors of s_{d-1} are generated in the same iteration (if their index value is larger than i) or in the next iteration (if their index value is smaller than i) such that *Scan*(*rank*$(s_d)) \leq$ *Layer*(*rank*$(s_d))$. □

One-Bit Breadth-First Search

Given a move-alternation or layer-selection property, we can perform BFS using only one bit per state.

Assuming a layer-selection property, Algorithm 6 retains the breadth-first order, i.e., a bit identifying a node is set in the BFS-layer where the node is reached for the first time. The bitvectors *Open* are expanded depending on the breadth-first *layer*. The algorithm terminates when no new successor is generated.

Algorithm 6 *One-Bit-BFS* (*init*)

```
 1: for all i := 0, ..., |S| − 1 do
 2:     Open[0][i] := false
 3: Open[0][rank_0(init)] := true
 4: layer := 0
 5: while S_layer ≠ ∅ do
 6:     layer := layer + 1
 7:     for all i := 0, ..., |S| − 1 do
 8:         Open[layer][i] := false
 9:     for all i := 0, ..., |S_layer−1| − 1 do
10:         if Open[layer − 1][i] = true then
11:             succs := expand(unrank_layer−1(i))
12:             for all s ∈ succs do
13:                 r := rank_layer(s)
14:                 Open[layer][r] := true
```

One important observation is that not all visited states that appear in previous BFS layers are removed from the current one. Hence, there are states that are reopened; in the worst case once for each BFS layer. Even though some states may be re-expanded several times, the following result is immediate.

Theorem 7 (Population Count In One-Bit-BFS) *Let the population count pc_l be the number of bits set after the l-th scan in Algorithm* One-Bit-BFS. *Then the number of states in BFS-layer l is $|Layer_l| = pc_l - pc_{l-1}$.*

GPU Essentials

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

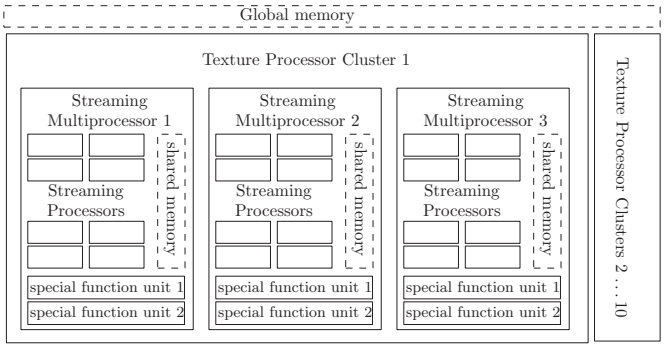

Figure 1: Sample GPU Architecture (G200 Chipset).

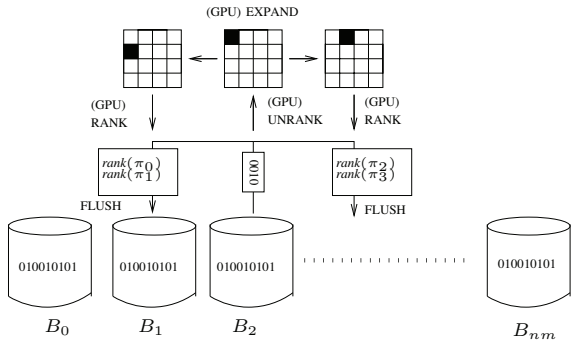

Figure 2: GPU Exploration of the Sliding-Tile Puzzle Domain represented as a Bitvector.

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 1, a core is a streaming processor (SP). 8 SPs are grouped together to one streaming multiprocessor (SM), and used like ordinary SIMD processors. Each of the 10 texture processor clusters (TPCs) combines 3 SMs, yielding 240 cores in one chip.

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 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 function executed in parallel on the GPU is called *kernel*. The kernel is driven by threads, grouped together in *blocks*. The TPC distributes the blocks to its SMs in a way that a block is not distributed among different SMs. All the SPs get the same chunk of code, so that SPs in an else-branch wait for the SPs operating in the according if-branch, being idle. After all threads have completed a chunk of code, the next one is executed. Threads waiting for data can be parked by the SM, while the SPs work on threads, which have already received the data.

Implementation on the GPU

Let us now consider how to implement 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 is executed unchanged for all threads. These facts lead to keeping the entire or partitioned state space bitvector in RAM and copying an array of ranks to the GPU. This approach benefits from the SIMD technology but imposes additional work on the CPU. One scan through the bitvector is needed to convert its bits into ranks, but on the GPU the work to unrank, generate the successors and rank them is identical for all threads. To avoid unnecessary memory accesses, the value given to expand should be overwritten with the value of the first child. If the number of successors is known in advance, we can reserve space for these successors in advance.

In larger instances that exceed RAM capacities we employ I/O-efficient disk-based algorithms, additionally maintaining write buffers to avoid random access on disk. Once the buffer is full, it is flushed to disk. In one streamed file access, all corresponding bits in the buffer are set.

The setting is exemplified for the sliding-tile puzzle domain in Figure 2. We see the "blank-partitioned" BFS layer of the state space resides on disk. It is read into RAM, converted to integer ranks, and copied to the GPU to be unranked, expanded and ranked again.

Since the bottleneck on the GPU is the memory not the computing power we suggest toing additional useful operations after expanding the state. One option is computing the heuristic value and to also accelerate heuristic search. By the large reduction in state space due to the directness 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 also ported breadth-first heuristic search (BFHS) (Zhou and Hansen 2006) to the GPU. For a given upper bound U on the optimal solution length and current BFS-layer g the GPU receives the value $U - g$ as the maximal possible h-value, and marks states with larger h-value as invalid.

Experiments

To test the efficiency of our approach, we implemented the hash functions for several search problems and compared the running times for GPU and CPU execution. We conducted the experiments on an Intel Core i7 CPU 920 @2.67GHz system with 12 GB RAM and 1.5 TB external storage, distributed on 4 hard disks. The GPU we used is an NVIDIA N285GTX rev a1 with 1 GB VRAM and 240 cores.

For measuring speed-ups, we compare the GPU performances with the run-times of a SIMD CPU implementation, either running on a single or on multiple cores (8 in our case,

Table 1: GPU vs. CPU Performance using 1-Bit BFS (o.o.t denotes out of time).

Domain	Instance	Times		
		GPU	CPU	CPU
		GPU	1 Core	8 Cores
Sliding-Tile	(3×4)	$66s$	$427s$	$217s$
	(4×3)	$78s$	$475s$	$187s$
	(2×6)	$93s$	$1,114s$	$374s$
	(6×2)	$114s$	$1,210s$	$284s$
	(7×2)	$14,215s$	o.o.t.	$22,396$
Peg-Solitaire		$44s$	$360s$	

using *pthreads*). We used this to grant fairness between the two implementations, since a conceptually different CPU algorithm would not be comparable to the GPU one.

Permutation-type Problems

Extracting a state from a permutation rank, generating the children and computing their ranks was tested on three different single player games gaining speed-ups of more then one order of magnitude.

Sliding-Tile Puzzle The $(n \times m)$ sliding-tile puzzle (see Fig. 3a) 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 of the states are reachable.

The move-alternating property we used in this puzzle is the position of the blank. Note that there is one subtle problem with ranking 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 $B_0, \ldots, B_{nm-1}$ 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.

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-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 and Schultze (2005).

Table 2: GPU vs. CPU Performance using 1-Bit BFHS in the Sliding-Tile Puzzle Domain.

Instance			Times	
Size	Rank	Blank Pos.	GPU	CPU
(2×6)	$18,295,101$	5	$25s$	$98s$
(6×2)	$799,911$	1	$21s$	$95s$
(3×4)	$5,840,451$	9	$30s$	$208s$
(4×3)	$1,560,225$	3	$27s$	$202s$
(2×7)	$2,921,466,653$	6	$114s$	$2,188s$

For the 1-Bit BFS implementation the speed-up achieves 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 ranks) compared to about 5 million states per second of the CPU (utilizing one core). 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. We also implemented a multicore version of the algorithm utilizing the available 8 cores and showing the benefit of the GPU implementation.

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 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 Puzzle The next example is the (n, k)-Top-Spin Puzzle (see Fig. 3b) (Chen and Skiena 1995), 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.

The results for the (n, k)-Top-Spin problems for $k = 4$ are shown in Table 3. Since no layer-selection or move-alternating property is known, we perform 2-bit BFS. We additionally compare the GPU performance to a parallel CPU implementation, where the algorithm utilizes all the avail-

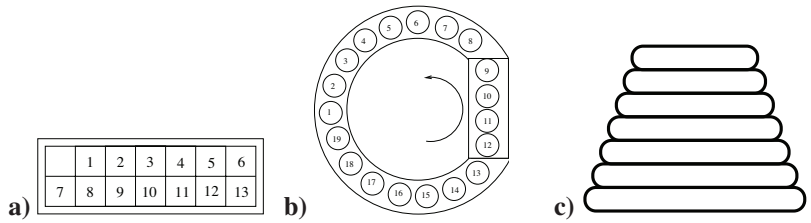

Figure 3: Permutation Games: a) Sliding Tile Puzzle, b) Top-Spin Puzzle c) Pancake Problem.

Table 3: GPU vs. CPU Performance using 2-Bit BFS.

Domain	Instance	Times		
		GPU	CPU 1 Core	CPU 8 Cores
Top-Spin	10	$0s$	$2s$	$0s$
	11	$1s$	$10s$	$3s$
	12	$12s$	$272s$	$63s$
	13	$87s$	$2,404s$	$510s$
Pancake	10	$0s$	$4s$	$2s$
	11	$9s$	$52s$	$14s$
	12	$130s$	$832s$	$164s$
	13	$1,819s$	$11,771s$	$2,499s$
Frogs and Toads		$686s$	$8,880s$	

able cores. For large values of n, we obtain a significant speed-up of more than factor 27 wrt. the single core computation, and a factor of 5 compared to the 8 core computation.

Pancake Problem The n-Pancake Problem (see Fig. 3c) (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. It is known that $(5n + 5)/3$ flips always suffice, and that $15n/14$ flips are necessary.

The GPU and CPU running time results for the n-Pancake problems are shown in Table 3. In contrast to the Top-Spin puzzle for a large value of n, we obtain a speed-up factor of 7 wrt. running the same algorithm on one core of the CPU.

Binomial-type Problems

Two games were analyzed to test the effectiveness of the binomial ranking. Since calculating large binomial coefficients is a computationally intensive task, necessary values were precomputed and stored in an array while initializing the experiments.

Peg-Solitaire Peg-Solitaire (see Fig. 4a), is a single-agent problem invented in the 17th century. The game asks for the minimum number of pegs that is reachable from a given initial state. The set of pegs is iteratively reduced by jumps. Solutions for the initial state (shown in Fig. 4a) with one peg remaining in the middle of the board are widely known (Berlekamp, Conway, and Guy 1982). An optimal player for all possible states has been generated by Edelkamp and Kissmann (2007).

In this problem the number of pegs decreases with each move by one. Using this layer-selection property we can assign each state immediately to its BFS layer. We performed a 1-bit BFS on the complete state space and show the results in Table 1. The GPU achieves a significantly faster exploration speed then a CPU exploration.

Frogs and Toads The Frogs and Toads puzzle (see Fig. 4b) is a generalization of the Fore and Aft puzzle, which has been made popular by the American puzzle creator Sam Loyd. It is played on a part of the 7x7 board consisting of two 4x4 subarrays at diagonally opposite corners. They overlap in the central square. One square has 15 black pieces and the other has 15 white pieces, with the centre left vacant. A move is to slide or jump over another pieces of any color (without removing it). The objective is to reverse the positions of pieces in the lowest number of moves. This game was originally an English invention in the 18th century. Henry Ernest Dudeney discovered a quickest solution of just 46 moves for Fore and Aft. To our knowledge full explorations of larger instances of Frogs-and-Toads have not been reported.

In this problem, each field can be either blank or occupied with a black or a white piece. Hence a straight-forward binomial ranking would not suffice to distinguish the three different options. To overcome this challenge, we exploit the fact that the number of pieces remains unchanged during the entire exploration, and divide the search space into disjoint sets, identified by the position of the blank. This way we can define one color as the satisfied variable in order to use the binomial ranking. Table 3 shows the results.

Conclusion

In this paper we studied the application of GPU computation in selected search domains. We have shown how to apply GPU-based BFS. The speed-ups of up to factor 27 with respect to one core CPU computation compare well with the speeds we obtained on a multiple core CPU (Korf and Schultze 2005; Zhou and Hansen 2007).

Two-bit BFS is applicable if reversible and perfect hash functions are available. Given a move-alternating or a layer-selection property 1-bit per state suffices for a BFS. One-bit reachability shows an interesting time-space trade-off.

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 reversible minimal perfect hash functions for the permutation games, we

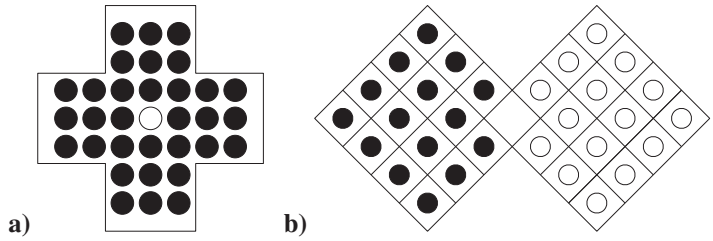

Figure 4: Binomial Problems: a) Peg Solitaire, b) Frogs and Toads.

thus studied the ranking proposed 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.

One open question is how domain-independent the results are. The algorithms require a perfect and reversible hash function that does not exceed SRAM. The mapping does not have to be minimal, but space-efficient to yield a memory advantage wrt. an explicit storage of the state vector. An automatic generation of such concise mapping seems difficult, but if we consider the hash function as part of the input, then all the algorithms above are domain-independent.

Acknowledgements Thanks to Shahid Jabbar and Peter Kissmann for proofreading and DFG for support in the project ED 74/8.

References

Belazzougui, D.; Botelho, F. C.; and Dietzfelbinger, M. 2009. Hash, displace, and compress. In Fiat, A., and Sanders, P., eds., *ESA*, volume 5757 of *Lecture Notes in Computer Science*, 682–693. Springer.

Berlekamp, E. R.; Conway, J. H.; and Guy, R. K. 1982. *Winning Ways for your Mathematical Plays*, volume 2. ISBN 0-12-091152-3: Academic Press. chapter 25.

Bonet, B. 2008. Efficient algorithms to rank and unrank permutations in lexicographic order. In *AAAI-Workshop on Search in AI and Robotics*.

Breyer, T., and Korf, R. E. 2009. 1.6-bit pattern databases. In *Symposium of Combinatorial Search*.

Chen, T., and Skiena, S. 1995. Sorting with fixed-length reversals. *Discrete Applied Mathematics* 71:269–295.

Cooperman, G., and Finkelstein, L. 1992. New methods for using cayley graphs in interconnection networks. *Discrete Applied Mathematics* 37/38:95–118.

Dweighter, H. 1975. Problem e2569. *American Mathematical Monthly* 82:1010+.

Edelkamp, S., and Kissmann, P. 2007. Symbolic exploration for general game playing in PDDL. In *ICAPS-Workshop on Planning in Games*.

Korf, R. E., and Schultze, P. 2005. Large-scale parallel breadth-first search. In Veloso, M. M.; Kambhampati, S.; Veloso, M. M.; and Kambhampati, S., eds., *AAAI*, 1380–1385. AAAI Press / The MIT Press.

Korf, R. E. 2003. Delayed duplicate detection: extended abstract. In *IJCAI'03: Proceedings of the 18th international joint conference on Artificial intelligence*, 1539–1541. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Korf, R. E. 2008. Minimizing disk I/O in two-bit breadth-first search. In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, 317–324. AAAI Press.

Kunkle, D., and Cooperman, G. 2007. Twenty-six moves suffice for rubik's cube. In *ISSAC '07: Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, 235–242. New York, NY, USA: ACM.

Leischner, N.; Osipov, V.; and Sanders, P. 2009. Gpu sample sort. *CoRR* abs/0909.5649.

Lindholm, E.; Nickolls, J.; Oberman, S.; and Montrym, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 28(2):39–55.

Munagala, K., and Ranade, A. 1999. I/O-complexity of graph algorithms. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, 687–694. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

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'04: Proceedings of the 19th national conference on Artifical intelligence*, 683–688. AAAI Press / The MIT Press.

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

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *AAAI'07: Proceedings of the 22nd national conference on Artificial intelligence*, 1217–1223. AAAI Press.