

GPU Exploration of Two-Player Games with Perfect Hash Functions

Stefan Edelkamp and Damian Sulewski

University of Bremen
{edelkamp,sulewski}@tzi.de

Cengizhan Yücel

Dortmund University of Technology
cengizhan.yuecel@googlemail.com

Abstract

In this paper we improve solving two-player games by computing the game-theoretical value of every reachable state.

A graphics processing unit located on the graphics card is used as a co-processor to accelerate the solution process. We exploit perfect hash functions to store the game states efficiently in memory and to transfer their ordinal representation between the host and the graphics card.

As an application we validate Gasser's results that Nine-Men-Morris is a draw on a personal computer. Moreover, our solution is strong, while for the opening phase Gasser only provided a weak solution.

Introduction

Thanks to a continuous improvement algorithms, but also because of the increasing powers of the *central processing units* (CPUs), search engines have been able to successfully cope with complexity and tackle a wide range of problems. Unfortunately, it seems that we cannot rely anymore on Moore's law that predicts doubling of the efficiency of the hardware each 18 months. The answer of the manufacturers to the Moore's law failure is focusing on further development of multi-core CPU systems that essentially contain multiple processors in one. These parallel processors are already part of the standard desktops and laptops. Such a *parallelism for the masses* offers immense opportunities for the improvement of search algorithm.

In the last few years there is a rising trend to exploit the *graphics processing unit* (GPU) not only for image processing but as a co-processor to support the CPU. A suitable graphics processor is frequently referred to as a *general purpose graphics processing unit*.

While current *multi-core* CPUs have up to 12 processor cores, current *many-core* GPUs comprise several hundreds of cores. To exploit their parallel processing power, programming interfaces like *CUDA*, *Stream*, or *OpenCL* have been developed. Significant speed-ups wrt. CPU calculations have been obtained in mathematics (Göddeke et al. 2008) or medicine (Owens et al. 2007).

In this work we *strongly solve* one prominent two-player zero-sum game on the GPU, i.e., assuming optimal play,

the solvability status of each reachable state is computed. We use perfect hash functions to save memory demands and to exchange the state information with the GPU efficiently. As far as the perfect hash function and its inverse are efficiently computable and the bitvector representation of the state space at least partially fits in RAM, the approach of ranking, unranking, expanding, and evaluating states on the GPU is general to many two-player zero-sum board games. As our application domain, we provide a strong solution to the game Nine-Men-Morris utilizing the GPU. For this problem ordinary hashing with full state storage would very likely exceed RAM. Space-efficient alternatives like Bloom filters (1970) are lossy and may yield a wrong result, while state vector sharing, e.g., in BDDs (Bryant 1986), often induces unacceptable large run times.

The paper is structured as follows. First, we recall perfect hashing that will be used to address states space-efficiently and to move state information between the host and the graphics card. In this context we introduce perfect hashing with multinomial coefficients and prove its correctness. To motivate the design of this approach we briefly review the architecture of modern GPUs. Next, we apply parallel BFS on the GPU and turn to solving Nine-Men-Morris with parallel retrograde search. We compare exploration results on the GPU with ones on the CPU and draw conclusions.

Perfect Hashing

Game states are often represented as a vector of variable assignments, which – considering a huge number of states – can consume a sizable amount of space. An apparent alternative are perfect hash functions (Knuth 1998, S. 513 ff.), which reduce the vector representation to an ordinal number.

More formally, a *hash function* is a mapping h of a set of all possible game states U to $\{0, \dots, m-1\}$ with $|U| \geq m$. The set of reachable game states $S \subseteq U$ is often smaller. A hash function $h : S \mapsto \{0, \dots, m-1\}$ is *perfect*, if it is injective, and *minimal*, if $|S| = m$, i.e., minimal perfect hash functions bijectively map m states to $\{0, \dots, m-1\}$.

The inverse of a perfect hash function is well defined through the injectivity of the mapping, for practical purposes, similar to computing the hash function itself, it should also be determined efficiently; best in time linear to the state vector size for a fast reconstruction given a hash value. In the case of invertible perfect hash functions, we often speak

of *ranking* and *unranking*.

As hash conflicts are avoided, the state information can also be assigned implicit to the address of a bitvector yielding a state space representation with only 1 bit per state.

Perfect hash functions have been used to compress state vectors for permutation games like the $(n^2 - 1)$ -Puzzle and the Pancake Problem (Korf and Schultze 2005; Myrvold and Ruskey 2001; Mares and Straka 2007), and selection games like Peg Solitaire and Frogs-and-Toads (Edelkamp, Sulewski, and Yücel 2010). In the latter work binomial hash functions as a precursor to multinomial hashing is applied.

Multinomial coefficients can be used to compress state vectors sets with a fixed but permuted value assignment, e.g., board games state (sub)sets where the number of pieces for each player does not change. For example, all state vectors of length 6 with 3 bits set result in the following perfect mapping:

Rank	State	Rank	State	Rank	State	Rank	State
0	111000	5	101010	10	011100	15	010011
1	110100	6	101001	11	011010	16	001110
2	110010	7	100110	12	011001	17	001101
3	110001	8	100101	13	010110	18	001011
4	101100	9	100011	14	010101	19	000111

The intuition is that *ranking* adds numbers of remaining paths in a directed grid graph. Any state corresponds to a path that starts from root $(6, 3)$ and ends in sink $(0, 0)$ and each node is assigned to some binomial coefficient, determining the number of remaining paths to $(0, 0)$. Given a path, its ordinal representation can be inferred by accumulating the numbers that label nodes *below* it, where below is defined as following an edge for zero-bit when the state vector enforces following an edge for a one-bit.

For p players in a game on n positions we use k_i with $1 \leq i \leq p$ to denote the number of game pieces owned by player i , and k_{p+1} for the remaining empty positions.

Definition 1 For $n, k_1, k_2, \dots, k_m \in \mathbb{N}$ with $n = k_1 + k_2 + \dots + k_m$ the multinomial coefficient is defined as

$$\binom{n}{k_1, k_2, \dots, k_m} := \frac{n!}{k_1! \cdot k_2! \cdot \dots \cdot k_m!}.$$

Since $\sum_{i=0}^{p+1} k_i = n$ we can deduce value k_{p+1} given $k_1, k_2, \dots, k_p$. We present multinomial hashing for $p = 2$ but the extension to three and more players is intuitive.

We will write $\binom{n}{k_1, k_2}$ for $\binom{n}{k_1, k_2, k_3}$ with $k_3 = n - (k_1 + k_2)$ and distinguish pieces by enumerating their *colors* with 1, 2, and 0 (empty).

Let S_{k_1, k_2} be the set of all possible boards with k_1 pieces of color 1 and k_2 pieces in color 2. The computation of the rank for states in S_{k_1, k_2} is provided in Algorithm 1. As above, the intuition for procedure *Rank* is to walk down through a 3-dimensional grid graph while accumulating the number of paths *below* the current one that reaches the sink. It defines h_{k_1, k_2} via counting with multinomial coefficient. For each position i we check, if a 2 (line 3), a 1 (line 5) or a 0 (line 9) is present in the state vector.

- If a 2 is found, the value in variable l_{twos} is decremented by 1, while r remains unchanged.

Algorithm 1 Rank

Input: Game state vector: $state[0, \dots, n-1]$,

number of pieces in color 1: l_{ones} ,

number of pieces in color 2: l_{twos}

Output: Rank: $r \in \{0, \dots, |S|\}$

```

1:  $i \leftarrow 0, r \leftarrow 0$ 
2: while  $i < n$  do
3:   if  $state[i] = 2$  then
4:      $l_{twos} \leftarrow l_{twos} - 1$ 
5:   else if  $state[i] = 1$  then
6:     if  $l_{twos} > 0$  then
7:        $r \leftarrow r + \binom{n-i-1}{l_{ones}, l_{twos}-1}$ 
8:      $l_{ones} \leftarrow l_{ones} - 1$ 
9:   else
10:    if  $l_{twos} > 0$  then
11:       $r \leftarrow r + \binom{n-i-1}{l_{ones}, l_{twos}-1}$ 
12:    if  $l_{ones} > 0$  then
13:       $r \leftarrow r + \binom{n-i-1}{l_{ones}-1, l_{twos}}$ 
14:     $i \leftarrow i + 1$ 
15: return  $r$ 
```

- In case of a 1, with the according multinomial coefficient we count the number of assignments, that have been visited and that contain a 2 at the current position. This is done only if there are still remaining 2s ($l_{twos} > 0$). Since we have seen a 1, variable l_{ones} is decremented by one.
- If a 0 is processed, we skip all visited 2s as long as $l_{twos} > 0$, and all 1s up to the current position, if $l_{ones} > 0$.

Theorem 1 The hash function defined in Algorithm 1 is bijective.

Proof Let $h_{k_1, k_2} : S_{k_1, k_2} \mapsto \mathbb{N}$ be the hash function defined by Algorithm 1. We show: 1) for all $s \in S_{k_1, k_2}$ we have $0 \leq h_{k_1, k_2}(s) \leq \binom{n}{k_1, k_2} - 1$; and 2) for all $s, s' \in S_{k_1, k_2}$: $s \neq s'$ implies $h_{k_1, k_2}(s) \neq h_{k_1, k_2}(s')$.

1) As r is initialized to 0 and increases monotonically, we only show the upper bound. The values that are added to r are $value_1 = \binom{n-i-1}{l_{twos}, l_{ones}-1}$ and $value_2 = \binom{n-i-1}{l_{twos}-1, l_{ones}}$. These values depend on the position $(i+1)$ of the currently considered state vector entry and on the number of non-processed pieces of color 1 (l_{ones}) and color 2 (l_{twos}). We additionally observe that the number of non-processed pieces referred to in the bottom line of the expressions decreases monotonically.

Similar to binomial coefficients, the coefficients of the expansions $(a + b + c)^n$ also form a geometric pattern. In this case the shape is a three-dimensional triangular pyramid, or tetrahedron. Each horizontal cross section of such tetrahedron is a triangular array of numbers, and the sum of three adjacent numbers in each row gives a number in the following row. In this structure we easily observe that for all $n \in \mathbb{N}^+$, and all $k_1, k_2, k_3 \in \mathbb{N}$ with $n = k_1 + k_2 + k_3$ we have

$$\binom{n}{k_1, k_2, k_3} \geq \binom{n-1}{k_1, k_2, k_3}$$

and for all $n, k_1 \in \mathbb{N}^+, k_2, k_3 \in \mathbb{N}$ with $n = k_1 + k_2 + k_3$:

$$\binom{n}{k_1, k_2, k_3} \geq \binom{n-1}{k_1-1, k_2, k_3+1},$$

Hence, $value_1$ and $value_2$ are maximized, if the first position of the state vector entry is maximized, followed by the second and so forth. Hence r is maximal, if at the first k_3 positions we have only 0s, while in the following k_1 positions we have only 1s and the remaining k_2 positions contain 2s.

As for such maximal r we have 0s for the first k_3 positions, the according values l_{ones} and l_{twos} in the corresponding multinomial coefficient are constant. These positions thus add the following offset $\Delta_{0,max}$ to r ($k_1 = l_{ones}$ and $k_2 = l_{twos}$):

$$\sum_{i=1}^{k_3} \left(\binom{n-i}{k_1, k_2-1, k_3+1-i} + \binom{n-i}{k_1-1, k_2, k_3+1-i} \right)$$

At the following k_1 positions for such maximal r all 1s are scanned, while the value k_2 remains constant at l_{twos} . Value l_{ones} matches k_1 initially and is decremented by 1 for each progress in i . Obviously, 0s are no longer present, such that the offset $\Delta_{1,max}$ equals

$$\sum_{i=1}^{k_1} \binom{n-k_3-i}{k_1+1-i, k_2-1, 0}$$

As the multinomial coefficient can be expressed as a product of binomial coefficients.

$$\binom{n}{k_1, k_2, \dots, k_r} = \binom{k_1+k_2}{k_2} \dots \binom{k_1+k_2+\dots+k_r}{k_r}$$

we rewrite the summands for $\Delta_{0,max}$ to

$$\sum_{i=1}^{k_3} \left(\binom{k_1+k_2-1}{k_2-1} \binom{n-i}{k_3+1-i} \right)$$

and

$$\sum_{i=1}^{k_3} \left(\binom{k_1+k_2-1}{k_2} \binom{n-i}{k_3+1-i} \right).$$

For binomial coefficients we have

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

and

$$\sum_{i=1}^{k_3} \binom{n-i}{k_3+1-i} = \sum_{i=1}^{k_3} \binom{n-k_3-1+i}{n-k_3-1}.$$

Using the identity $\sum_{i=k+1}^n \binom{i}{k} = \binom{n+1}{k+1} - 1$ this implies that $\Delta_{0,max}$ is equal to

$$\begin{aligned} & \sum_{i=1}^{k_3} \left(\binom{n-i}{k_3+1-i} \cdot \left(\binom{k_1+k_2-1}{k_2-1} + \binom{k_1+k_2-1}{k_2} \right) \right) \\ &= \sum_{i=1}^{k_3} \left(\binom{n-i}{k_3+1-i} \binom{k_1+k_2}{k_2} \right) \\ &= \binom{k_1+k_2}{k_2} \cdot \left(\binom{n}{n-k_3} - 1 \right) \end{aligned}$$

Algorithm 2 Unrank

Input: Rank r ,

number of pieces in color 1: l_{ones} ,

number of pieces in color 2: l_{twos}

Output: Game state vector: $state[0 \dots n-1]$

```

1:  $i \leftarrow 0$ 
2: while  $i < n$  do
3:   if  $l_{twos} > 0$  then
4:      $value_2 \leftarrow \binom{n-i-1}{l_{ones}, l_{twos}-1}$ 
5:   else
6:      $value_2 \leftarrow 0$ 
7:   if  $l_{ones} > 0$  then
8:      $value_1 \leftarrow \binom{n-i-1}{l_{ones}-1, l_{twos}}$ 
9:   else
10:     $value_1 \leftarrow 0$ 
11:   if  $r < value_2$  then
12:      $state[i] \leftarrow 2$ 
13:      $l_{twos} \leftarrow l_{twos} - 1$ 
14:   else if  $r < value_1 + value_2$  then
15:      $state[i] \leftarrow 1$ 
16:      $r \leftarrow r - value_2$ 
17:      $l_{ones} \leftarrow l_{ones} - 1$ 
18:   else
19:      $state[i] \leftarrow 0$ 
20:      $r \leftarrow r - (value_1 + value_2)$ 
21:      $i \leftarrow i + 1$ 
22: return  $state$ 

```

and $\Delta_{1,max}$ is equal to

$$\begin{aligned} & \sum_{i=1}^{k_1} \left(\binom{k_1-i+1}{k_1-i+1} \binom{k_1+k_2-i}{k_2-1} \binom{k_1+k_2-i}{0} \right) \\ &= \sum_{i=1}^{k_1} \binom{k_1+k_2-i}{k_2-1} \\ &= \binom{k_1+k_2}{k_2} - 1. \end{aligned}$$

Hence, the maximal possible value for r is

$$\begin{aligned} r_{max} &= \Delta_{0,max} + \Delta_{1,max} \\ &= \binom{k_1+k_2}{k_2} \cdot \left(\left(\binom{n}{n-k_3} - 1 \right) + \binom{k_1+k_2}{k_2} \right) - 1 \\ &= \binom{k_1}{k_1} \binom{k_1+k_2}{k_2} \binom{n}{k_3} - 1 \\ &= \binom{n}{k_1, k_2, k_3} - 1. \end{aligned}$$

2) Consider two states $s_1, s_2 \in S_{k_1, k_2}$ and the smallest possible index in which the two states differ, i.e.,

$$i' := \min \{i \mid 0 \leq i \leq (n-1) \wedge state_{s_1}[i] \neq state_{s_2}[i]\}$$

The values of r computed up to i' are the same. Let $k'_1 \leq k_1$ and $k'_2 \leq k_2$ be the remaining pieces of the respective color. We have $r_{s_1, i'} = r_{s_2, i'}$, where $r_{s, i'}$ denotes

the value r computed for state s before evaluating position i' . At position i' we have the following three cases.

In the first case, $state_{s_1}[i'] = 0$ and $state_{s_2}[i'] = 1$. The difference of the r values is

$$r_{s_1, i'+1} = r_{s_2, i'+1} + \binom{n-i'-1}{k'_1-1, k'_2}$$

Following the above derivations, we know that r_{s_2} increases by at most $\binom{n-i'-1}{k'_1-1, k'_2} - 1$ in the $(n-i'-1)$ remaining positions with (k'_1-1) and k'_2 pieces of the according color, such that $r_{s_1, j} \neq r_{s_2, j}$ for $j > i'$.

In the second case, we have $state_{s_1}[i'] = 0$ and $state_{s_2}[i'] = 2$. This implies

$$r_{s_1, i'+1} = r_{s_2, i'+1} + \binom{n-i'-1}{k'_1-1, k'_2} + \binom{n-i'-1}{k'_1, k'_2-1}$$

Value r_{s_2} increases by at most $\binom{n-i'-1}{k'_1, k'_2-1} - 1$ on the remaining $(n-i'-1)$ positions with k'_1 and (k'_2-1) pieces of the according color. We again have $r_{s_1, j} \neq r_{s_2, j}$ for $j > i'$.

The remaining case is $state_{s_1}[i'] = 1$ and $state_{s_2}[i'] = 2$ with

$$r_{s_1, i'+1} = r_{s_2, i'+1} + \binom{n-i'-1}{k'_1, k'_2-1},$$

where the argumentation of the second case applies. $\square$

Algorithm 2 is the inverse of Algorithm 1 and used to compute h_{k_1, k_2}^{-1} in form of assignments to a state vector. As the Unrank procedure subtracts the multinomial coefficients that match the ones that have been added in Rank, the inverse h_{k_1, k_2}^{-1} is computed correctly.

GPU Basics

Starting from the 1970s where computing devices displayed text only, graphics standards have grown over the years. In 1987, SVGA and resolutions of 800×600 together with several colors have been obtained, followed by XVGA, UVGA, SXGA and UXGA etc. (Eickmann 2007). To cope with the computational requirements, in 1999, NVIDIA presented a graphics accelerator that autonomously transformed data into a 2D image. Enlarging the capabilities of the transformations has lead to highly parallel systems. In about 2002/03 first ideas were born to use GPUs for more than only graphics processing¹.

Current GPUs obey a *SIMD*-architecture (Single Instruction, Multiple Data), that is, all processors execute the same code on different portions of the data. The cores are called *streaming processors* (SP). Take for example the NVIDIA GeForce GTX 285 architecture, where a *streaming multiprocessor* (SM) is composed of 8 SPs and every 3 SMs give a *TPC* unit (Texture/Thread Processing Cluster). One GPU has 10 TPC units. For the computation, each SP has one Floating Point Unit and two Arithmetic Logic Units, while a SM contains two Special Function Units and local shared memory, the *SRAM*, that is exclusively used by its SPs. Additionally, each SP has its own registers, which allow an independent execution of so-called *Threads* within one SM.

¹www.extremetech.com/article2/0,2845,1091392,00.asp

Algorithm 3 GPU Breadth-First Search

Input: Set $hash_{cpu}$, initialized with hash value of initial state

```

1: while  $hash_{cpu} \neq \emptyset$  do
2:    $hash_{gpu} \leftarrow hash_{cpu}$ 
3:   for all  $r \in hash_{gpu}$  do in parallel
4:      $s \leftarrow Unrank(r)$ 
5:     for all  $s' \in successors(s)$  do
6:        $successors_{gpu} \leftarrow successors_{gpu} \cup Rank(s')$ 
7:    $hash_{cpu} \leftarrow successors_{gpu}$ 
8:   for all  $r \in hash_{cpu}$  do
9:     Update information on BFS-Layer
10:    Remove duplicates from  $hash_{cpu}$ 
11: return
```

The top level memory, called *video RAM* (VRAM), is often limited to 1.5 GB and preferably accessed streamed² The number of instructions per second as well as the throughput of data are considerably large (NVI 2008).

Based on the GPU's parallel hardware design and on its hierarchical memory, there are limitations to GPU programming. E.g. due to the SIMD architecture large conditional branches should be avoided.

CUDA, a programming interface from NVIDIA, uses a hierarchy of *threads* that are clustered into thread *blocks*, which in turn are clustered into a *grid*. Threads within a block can be synchronized. The instructions are written in a *kernel*, whose call has to specify the dimensions of the grid. In order to select individual data items to work on, threads can extract their own ID.

State Space Search on the GPU

The design of a search algorithm (and the subsequent retrograde analysis) is related to how parallelism should be exploited on a GPU. Hash tables are good if they distribute, so that we prefer them not to stay on the graphics card. On the other hand, the computation of values (at least considering this amount of data) is a localized, comparably hard operation that should be parallelized. Duplicate detection is delayed. Moreover, depth-first search is inherently sequential, so that we prefer a breadth-first exploration for which all successors in a layer can be computed in parallel.

On the CPU we, therefore, maintain a set $hash_{cpu}$, that contains all hash values for the actual BFS layer, which is either maintained in RAM or on external media. On the GPU, we maintain $hash_{gpu}$ and a multi-set $successors_{gpu}$ with hash values of the generated successors in the VRAM.

Cooperman and Finkelstein (1992) have shown that 2 bits per state are sufficient to distinguish four types of information, *unreached*, *reached-and-to-be-processed*, *reached-and-to-be-processed-later* and *reached-and-processed* in a BFS algorithm. Algorithm 3 sketches such BFS on the GPU, where duplicates are eliminated by the CPU using a bitvec-

²There are NVIDIA offerings, e.g., NVIDIA Tesla C1060 comes with 4 GB of VRAM.

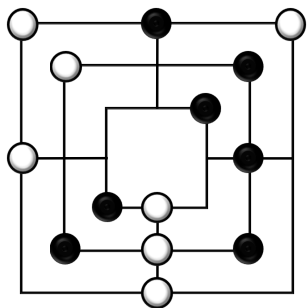


Figure 1: Nine Men's Morris.

tor representation of the state space.³ In the I/O setting, also investigated by Korf (2008), we sort successor ranks before merging them with the information stored on disk.

Nine-Men-Morris

The game *Nine Men's Morris* (see Fig. 1) has a board of three concentric squares that are connected at the mids of their sides. The 12 corner and 12 side intersections are the game positions (see Figure). Initially, each player picks 9 pieces in one color. The game divides into an opening (I), a middle (II) and an ending (III) phase. In all phases a player may close *mills*, i.e., align three pieces in his color horizontally or vertically, even across the diagonals where no lines are marked. In this case, he can remove one of the opponent's pieces from the board provided that it is not contained in a mill (for the case of having two mills closed in one move, only one piece can be taken, and if the opponent only has mills, they can be destroyed). Once a piece is removed from the board it takes no further part in the game. The game ends when one player is reduced to two pieces and so can no longer form a mill. A player who is blocked, i.e. is unable to move any piece, also loses the game.

The opening phase begins with an empty board. Each player has nine pieces which are placed one at a time in turn on any vacant point on the board until both have played all nine. The middle phase starts when all the pieces have been used. Play continues alternately with the opponents moving one piece to any adjacent point.

Once a piece is removed from the board it takes no further part in the game. Firstly, once a mill is formed it may be opened by moving one piece from the line and closed by returning it to its original position in the next move. Alternatively, in a running mill opening one mill will close another one so that an opponent's piece is removed on every turn.

The ending phase allows a player with only three pieces to jump, i.e. to move one piece to any empty point on the board regardless of position. The other player must continue to move normally unless both are reduced to three pieces.

By an exhaustive enumeration on a parallel architecture, the game was shown to be a draw by Gasser (1996), but his results have never been validated (Gasser himself called for

³Experiments show that storing the bitvector on the GPU yields inferior results, since random access to the VRAM is slow.

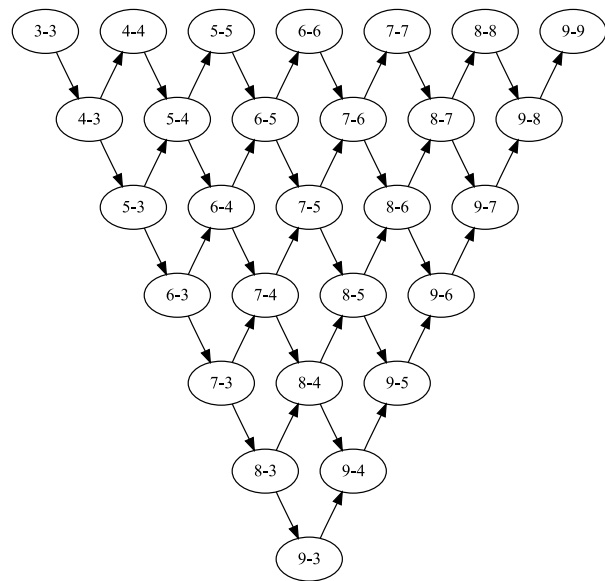


Figure 2: Partitioning for Phases II and III.

an independent proof). He partitioned the state space in sets S_{k_1, k_2} , which contains k_1 pieces of the first player, k_2 pieces of the second player and $n - (k_1 + k_2)$ empty intersections. Obviously, S_{k_1, k_2} and S_{k_2, k_1} are symmetrical, so that it is sufficient to consider S_{k_1, k_2} with $k_1 \geq k_2$ only. We inherit Gasser's organizational structure, but address states in the sets S_{k_1, k_2} via multinomial perfect hash functions.

After generating the state space, the game is solved bottom-up. As the number of pieces in the phases II and III can only decrease, Gasser has applied such retrograde analysis to these two phases. For phase I, however, he applied *AlphaBeta* pruning, which weakly solved the game. In contrast, we strongly solve it, and determine the game-theoretical value for each reachable game state.

While for acyclic two-player games 2 bits are sufficient to encode won, lost and draw positions and to conduct a retrograde analysis, Nine-Men-Morris requires a *progress measure* to avoid infinite computations. We adapt Gasser's 1 byte encoding.

The partitioning in Fig. 2 indicates that for $i, j \in \{1, 2\}$ and $i \neq j$ predecessors of $S_{k_1, k_2, i}$ (Player i to move) are contained either in $S_{k_1, k_2, j}$ or, if player j has closed a mill, in $S_{k_1+1, k_2, j}$ (given $i = 1$) or $S_{k_1, k_2+1, j}$ (given $i = 2$).

Scanning Fig. 2 from left to right may be interpreted as a variant of space-efficient *frontier search* (Korf 1999; Korf and Zhang 2000). To analyse S_{k_1, k_2} with $k_1 > 3$ or $k_2 > 3$ we thus only require the results left to S_{k_1, k_2} to be present. Due to symmetry, S_{k_1, k_2} with $k_1 < k_2$ needs not to be considered again, so that a copy from S_{k_1, k_2} suffices to evaluate a state.

For the encoding of a rank r , 34 bits are sufficient, so that a 64-bit integer suffices to contain all state information. This integer actually stores pairs (r, v) with the additional state information v having 8 bits. The remaining bits are

Algorithm 4 BFS for Phase I

```

1:  $reached \leftarrow \emptyset$ 
2: for  $t \leftarrow 1$  to 4 do
3:    $reached \leftarrow reached \cup (\lceil \frac{t}{2} \rceil, \lfloor \frac{t}{2} \rfloor, t)$ 
4:   Mark all states in  $bits_{\lceil \frac{t}{2} \rceil, \lfloor \frac{t}{2} \rfloor, t}$  with reached
5: for  $t \leftarrow 5$  to 18 do
6:   for all  $(k_1, k_2, t') \in reached$  with  $t' = t - 1$  do
7:      $ranks_{cpu} \leftarrow \emptyset$ 
8:     for  $i \leftarrow 0$  to  $\binom{n}{k_1, k_2} - 1$  do
9:       if  $bits_{k_1, k_2, t'}(i) = reached$  then
10:         $ranks_{cpu} \cup Rank(i)$ 
11:       $expand_{GPU}(ranks_{cpu})$ 
12:      for all  $r' \in ranks_{cpu}$  do
13:        Determine  $k'_1, k'_2$  according to  $r'$ 
14:        if  $(k'_1, k'_2, t) \notin reached$  then
15:           $reached \leftarrow reached \cup (k'_1, k'_2, t)$ 
16:          Initialize vector  $bits_{k'_1, k'_2, t}$  with 'not reached'
17:          Mark  $r'$  in  $bits_{k'_1, k'_2, t}$  with 'reached'

```

used to store numbers of successors. The GPU expects pairs (r, v) for expansion and returns triples (r', v', c) , where c is the number of successors for the state represented in r' (still fitting into 64 bits). The CPU reads value c if needed for encoding r' more efficiently.

Phase I is not completely analyzed in Gasser (1996). Arguing that closing mills is unfortunate, his analysis was reduced to games with 8 or 9 of the 9 pieces for each player. In contrast, we analyze this phase completely. The BFS starts for phase I with an empty board and determines for all depth $t \in \{1, \dots, 18\}$ which sets $S_{k_1, k_2, t}$ are to be considered and which states are then reached in that set. The partition into sets $S_{k_1, k_2, t}$ is different to the one obtained in the other two phases and respects that some partitions may be encountered in different search depth.

The BFS traversal is shown in Algorithm 4. For depths 1 to 4 the state space is initialized with *reached*. Only in depth $t > 4$ closing mills is possible, so that the successors of a set in the two sets of the next depth are possible and, therefore, a growing number of state spaces are to be considered.

The sets are themselves computed in BFS, utilizing a set *reached* of triples (k_1, k_2, t) with piece counts k_1, k_2 and obtained search depth t . An according entry (k_1, k_2, t) denotes that BFS has reached all states in $S_{k_1, k_2, t}$. This allows us to compute the according state spaces incrementally.

If $S_{k_1, k_2, t}$ is encountered for the first time (line 14), prior to its usage in (line 17) the responsible bitvector is allocated and initialized as *not-reached*. In line 13 of the Algorithm 4 the outcomes for different k'_1, k'_2 are combined. If depth t is odd we have $k'_1 = k_1 + 1$ but both $k'_2 = k_2$ and $k'_2 = k_2 - 1$ are possible (depending on a mill being closed or not). We take an additional bit in the encoding of the ranks to denote if a mill has been closed to accelerate the determination of values k'_1, k'_2 of rank r' .

In principle, one bit per state is sufficient. Subsequent to the BFS a backward chaining algorithm determines the game-theoretical values. We use two bits per state to en-

Algorithm 5 Retrograde Analysis Phase I

Input: $bits_{k_1, k_2, t}, reached, bytes_{k_1, k_2, i}$

```

1: for all  $(k_1, k_2, t') \in reached$  with  $t' = 18$  do
2:   if  $k_1 \geq k_2$  then
3:     for all  $j \in \{j \mid bits_{k_1, k_2, t}(j) = reached\}$  do
4:        $bits_{k_1, k_2, t}(j) \leftarrow bytes_{k_1, k_2, 1}(j)$ 
5:   else
6:     for all  $j \in \{j \mid bits_{k_1, k_2, t}(j) = reached\}$  do
7:       Compute Rank  $j'$  of the inverted game state for state with rank  $j$ 
8:        $bits_{k_1, k_2, t}(j) \leftarrow bytes_{k_2, k_1, 2}(j')$ 
9:   for  $t \leftarrow 17$  to 1 do
10:    for all  $(k_1, k_2, t') \in reached$  with  $t' = t$  do
11:       $ranks_{cpu} \leftarrow \emptyset$ 
12:      for  $j \leftarrow 0$  to  $\binom{n}{k_1, k_2} - 1$  do
13:        if  $bits_{k_1, k_2, t'}(j) = reached$  then
14:           $ranks_{cpu} \cup Rank(j)$ 
15:         $ranks_{gpu} \leftarrow expand_{GPU}$ 
16:        for all  $r' \in ranks_{cpu}$  do
17:          Compute  $k'_1, k'_2$  associated with  $r'$ 
18:           $bits_{k'_1, k'_2, t}(r) \leftarrow bits_{k'_1, k'_2, t+1}(r')$ 

```

code the four cases *not-reached*, *won-for-player-1*, *won-for-player-2*, and *draw*. As we already use 1 bit for state-space generation, these demands are already allocated.

In the backward traversal described in Algorithm 5, first all state sets $S_{k_1, k_2, t}$ with depth $t = 18$ are initialized wrt. the data computed for phases II and III. As player 1 starts the game, he will also start phase II. For the initialization of $S_{k_1, k_2, t}$ with $t = 18$ and $k_1 \geq k_2$ we scan the corresponding bitvector and consider each state marked *reached* at position i the value stored with position i in the bytevector inferred for $S_{k_1, k_2, 1}$ from solving phases II and III. Depth and successor count information is ignored. We are only interested in whether a state is won, lost or a draw.

When trying to initialize $S_{k_1, k_2, t}$ with $t = 18$ and $k_1 < k_2$ we observe that no corresponding set $S_{k_1, k_2, 1}$ from phases II and III has been computed. In this case, we traverse the bitvector for $S_{k_1, k_2, t}$, but consider the set $S_{k_2, k_1, 2}$ from phases II and III. In each scan of $S_{k_1, k_2, t}$ when encountering a state s marked *reached* we compute the rank j of its inverted representation, so that player 1 now plays color 2 and player 2 plays color 1. Similarly, in case $S_{k_1, k_2, 1}$ is not present, we consider position j in the bytevector, while inverting the state to be considered. For the translation of states in state vector representation, their inverted representation and the computation of their ranks j , we use the GPU.

After the initialization we go one step back and work on the state spaces $S_{k_1, k_2, t}$ with $t = 17$. We generate all successors of a state reached and determine the game-theoretical value in the bitvector stored depth 18 by considering all values at positions that correspond to the successor ranks.

The value of a state is determined by considering all its successors. If all successors of a state with player 1 to move are lost, the state itself is lost. If at least one successor is won, then the state itself is won. In all remaining cases, the game is a draw. We continue until we reach depth 1.

Table 1: Retrograde Analysis of Phases II and III (times in seconds, sizes given in GB).

Single HDD								
	Size	GPU	CPU	Ratio	Size	GPU	CPU	Ratio
9-9	6.54	31,576	—	—	8-4	1.34	749	8,021
9-8	8.41	62,894	—	—	9-3	0.58	933	8,567
8-8	9.47	36,604	—	—	6-5	1.15	601	1,530
9-7	8.41	31,607	—	—	7-4	0.80	487	4,425
8-7	8.41	80,747	—	—	8-3	0.40	707	6,125
9-6	6.54	21,713	—	—	5-5	0.48	36	22
7-7	6.73	26,429	63,303	2.39	6-4	0.40	178	445
8-6	5.89	25,077	59,476	2.37	7-3	0.23	398	3,148
9-5	3.93	2,384	25,216	10.57	5-4	0.16	13	4
7-6	4.28	4,914	23,170	4.71	6-3	0.11	78	620
8-5	3.21	1,984	20,239	10.20	4-4	0.05	1	1
9-4	1.78	949	10,712	11.36	5-3	0.04	18	137
6-6	2.50	1,293	4,160	3.21	4-3	0.01	4	26
7-5	2.14	1,404	11,742	8.34	3-3	.003	14	81

Software RAID								
	Size	GPU	CPU	Ratio	Size	GPU	CPU	Ratio
9-9	6.54	36,598	—	—	8-4	1.34	610	8,093
9-8	8.41	57,057	—	—	9-3	0.58	807	8,547
8-8	9.47	35,441	—	—	6-5	1.15	493	1,484
9-7	8.41	43,003	—	—	7-4	0.80	397	4,434
8-7	8.41	61,750	—	—	8-3	0.40	609	6,123
9-6	6.54	12,174	—	—	5-5	0.48	11	23
7-7	6.73	15,284	52,441	3.43	6-4	0.40	157	439
8-6	5.89	19,538	57,988	2.96	7-3	0.23	357	3,145
9-5	3.93	2,045	25,134	12.29	5-4	0.16	5	6
7-6	4.28	4,914	22,981	4.98	6-3	0.11	69	619
8-5	3.21	1,805	20,257	11.22	4-4	0.05	1	1
9-4	1.78	829	10,725	12.93	5-3	0.04	17	137
6-6	2.50	1,137	4,160	3.62	4-3	0.01	3	26
7-5	2.14	1,211	11,682	9.64	3-3	.003	12	80

Experiments

The GPU used is located on a GTX 285 (MSI) graphics card from NVIDIA, which contains 240 processor cores and 1 GB VRAM. The programming environment is CUDA, a c-like language linked to ordinary c/c++ and that abstracts from thread generation and initialization. All experiments were executed on a system with a Intel Core i7 CPU 920 clocked at 2.67GHz and a 1 TB SATA harddisk (according to `hdparm` at about 100 MB/s for sequential reading). We also experimented with a Linux software RAID(0) with two SSDs and the above HDD (according to `hdparm` yielding about 240 MB/s for sequential reading).

The time and space performances of the retrograde analysis are shown in Table 1. Since 12 GB were not sufficient to maintain all responsible sets in RAM, states were sequentially flushed to (and subsequently read from) disk.

The entire classification of the state space for phases II and III on the GPU required about 4 days and 7 hours on one HDD and 3 days 19 hours on the software RAID (of three hard and solid state disks). For both settings, the correspond-

Table 2: Retrograde Analysis in Phase I (time in seconds).

Single HDD					
Depth	BFS	Retrograde	Depth	BFS	Retrograde
1	<1	<1	10	199	193
2	<1	<1	11	444	440
3	<1	<1	12	1,043	1,028
4	<1	<1	13	1,782	1,815
5	<1	<1	14	3,251	3,227
6	1	1	15	4,594	4,521
7	5	5	16	6,737	6,652
8	20	21	17	8,317	8,087
9	62	61	18	-	17,267

Software RAID					
Depth	BFS	Retrograde	Depth	BFS	Retrograde
1	<1	<1	10	171	163
2	<1	<1	11	390	388
3	<1	<1	12	909	885
4	<1	<1	13	1,583	1,554
5	<1	<1	14	2,838	2,828
6	1	1	15	4,047	4,021
7	4	4	16	5,743	5,996
8	17	17	17	7,219	6,996
9	53	52	18	-	16,141

ing CPU computation on one core has not been completed and terminated after 5 days.

In the table, we see that the results on the software RAID are generally better than on a single HDD. We observe speed-ups of more than one order of magnitude (plotted in bold font), exceeding the number of cores on most current PCs⁴. On faster hardware, better speed-ups are possible. NVIDIA GPUs can be used in SLI and the Fermi architecture (e.g. located on the GeForce GTX 480 graphics card) is coming out which will go far beyond the 240 GPU cores we had access to.

For larger levels, we observe that the GPU performance degrades. When profiling the code, we identified I/O access as one limiting factor. For example, reading $S_{8,8}$ from one HDD required 100 seconds, while the expansion of 8 million states, including ranking and unranking, required only about 1 second on the GPU. We see that the RAID0 array indeed led to higher transfer rates and better speed-ups by reducing the amount of time needed for I/O.

We found some inconsistencies in the GPU performance, e.g., for the partition 9-7 and 9-9, where the RAID0 was inferior to the single HDD. According to our calculations, due to storing intermediate results, 12 GB RAM should be sufficient for the bitvector for breadth-first and retrograde analysis in the RAM, so that no further access to HDD for swapping should have been necessary. But there is additional memory needed for preparing and postprocessing the VRAM in RAM for copy purposes. Together with the needs

⁴This assertion is true for the dual 6-core CPUs available from Intel, but not on a dual Xeon machine with two quad-core CPUs creates 16 logical cores due to multi-threading

of the operating system this indicated that the system did swap.

For analyzing phase I the program required 19 hours on one HDD and little less than 17 hours on the software RAID0. Table 2 depicts the individual timings obtained by the GPU. All 24 states in depth 1 turned out to be a draw such that the result of Gasser (1996) has been validated. First non-optimal moves are possible in depth 2.

Conclusions and Discussion

In this paper GPUs have been shown to be effective for solving the Nine-Men-Morris two-player game. Despite his own desire, so far Gasser's computations that ran over weeks on a cluster of several computers have not been validated. Additionally, we have invented a class of time- and space-efficient perfect hash functions that was used to compress the state space in order to compute a strong solution.

There is increasing interest in GPU programming in AI, e.g., in answering probabilistic queries over Bayesian networks (Silberstein et al. 2008) and in state space search (Edelkamp, Sulewski, and Yücel 2010). This paper, however, pioneers solving a non-trivial game on the GPU.

Our approach provides a clear, provably efficient, correct, and extensible design of a hash function. In fact, we contributed a generic and extensible approach and thorough study of the class of invertible multinomial perfect hash functions that can be applied in other state space search areas as a sub-component. In Appendix B of the technical report Lake et al. (1993) provide insights to a bitvector encoding to construct checkers endgame databases. Their indexing scheme is in essence a multi-nominal hashing with four different piece types (checkers and kings for both black and white). In addition, they extend the hashing scheme to handle splitting the databases into sub-databases based on both number of different piece types on the board and by how far down the board the furthest advanced checkers are. The math becomes more complicated when one gets up to more pieces and – at least on the first glance – harder to generalize. Another parallel bitvector retrograde analysis (without GPU) has been applied to solve Awari (Romein and Bal 2003). The authors generally talk about Gödel numbers for states without going into details. In their case, binomial hashing, a special case of multinomial hashing is applicable.

Acknowledgments Thanks to Deutsche Forschungsgemeinschaft (DFG) for support in project ED 74/8.

References

- Bloom, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13(7):422–426.
- Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* 35:677–691.
- Cooperman, G., and Finkelstein, L. 1992. New methods for using Cayley graphs in interconnection networks. *Discrete Appl. Math.* 37-38:95–118.
- Edelkamp, S.; Sulewski, D.; and Yücel, C. 2010. Perfect hashing for state space exploration on the GPU. In *ICAPS*, 57–64. AAAI.
- Eickmann, U. 2007. *Untersuchung der Echtzeitfähigkeit von Budget-Grafikkarten*. München: Grin Verlag.
- Gasser, R. 1996. Solving Nine Men's Morris. *Computational Intelligence* 12:24–41.
- Göddecke, D.; Strzodka, R.; Mohd-Yusof, J.; McCormick, P.; Wobker, H.; Becker, C.; and Turek, S. 2008. Using GPUs to improve multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering* 4(1):36–55.
- Knuth, D. E. 1998. *Art of Computer Programming, Volume 3: Sorting and Searching*. Redwood City: Addison-Wesley Professional, 3rd edition.
- Korf, R. E., and Schultze, T. 2005. Large-scale parallel breadth-first search. In *AAAI*, 1380–1385.
- Korf, R. E., and Zhang, W. 2000. Divide-and-conquer frontier search applied to optimal sequence alignment. In *AAAI*, 910–916.
- Korf, R. E. 1999. Divide-and-conquer bidirectional search: First results. In *IJCAI*, 1184–1189.
- Korf, R. E. 2008. Minimizing disk I/O in two-bit-breath-first search. In *AAAI*, 317–324.
- Lake, R.; Schaeffer, J.; and Lu, P. 1993. Solving large retrograde analysis problems using a network of workstations. Technical Report 93-13, Department of Computing Science, University of Alberta.
- Mares, M., and Straka, M. 2007. Linear-time ranking of permutations. In *ESA*, volume 4698 of *LNCS*, 187–193. Springer.
- Myrvold, W., and Ruskey, F. 2001. Ranking and unranking permutations in linear time. *Information Processing Letters* 79(6):281–284.
- NVIDIA. 2008. *CUDA Programming Guide 2.0*. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA%20Programming_Guide_2.0.pdf.
- Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A. E.; and Purcell, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 26(1):80–113.
- Romein, J. W., and Bal, H. E. 2003. Solving Awari with parallel retrograde analysis. *Computer* 36(10):26–33.
- Silberstein, M.; Schuster, A.; Geiger, D.; Patney, A.; and Owens, J. D. 2008. Efficient computation of sum-products on GPUs through software-managed cache. In *22nd International Conference on Supercomputing*, 309–318.