CUDA Checkers

Game Rules

The rules used are based on American Checkers.

- 8×8 board.
- Men move one square forward only.
- Kings move one square forward or backward.
- Capturing is mandatory — any of the longest capture sequences can be chosen.
- Draw occurs after 50 moves without a capture.

Adopting the American rules, as opposed to variants with flying kings, allows for certain interesting optimizations (discussed later).

Data Structures

Board

The board is represented in 12 bytes:

```
typedef struct {
    uint32_t white;
    uint32_t black;
    uint32_t kings;
} Board;
```

The checkers board is 8×8, but only half of the squares are playable, so 32 bits are sufficient to represent all piece positions.

For example, to check if a white king occupies position `x`:

```
(1 << x) & board.white & board.kings != 0
```

At first glance, a straightforward indexing might look like:

```
   28  29  30  31
 24  25  26  27
   20  21  22  23
 16  17  18  19
   12  13  14  15
 08  09  10  11
   04  05  06  07
 00  01  02  03
```

Notice that each piece can move ± 4 (except pieces on the top and bottom rows). Additionally, in even-numbered rows (starting from 0 at the bottom) moves can be ± 3, while in odd-numbered rows they can be ± 5. This complicates the move-generation algorithm and reduces performance.

It turns out that a better indexing scheme exists:

```
   11  05  31  25
 10  04  30  24
   03  29  23  17
 02  28  22  16
   27  21  15  09
 26  20  14  08
   19  13  07  01
 18  12  06  00
```

Here, each piece can move ± 1 or ± 7, regardless of row parity, simplifying move generation.

Move

A first idea for representing a move (16 bytes):

```
typedef struct {
    uint8_t path[10];
    uint8_t path_len;
    uint32_t captured;
} Move;
```

`path` is an array of indices along the piece's trajectory (start, intermediate, end squares). `captured` is a bitmask of opponent pieces captured. The maximum `path` length is 10, since at most 9 pieces can be captured in a single move under these rules.

A more memory-efficient representation (8 bytes) might be:

```
typedef struct {
  uint32_t path;
  uint8_t begin;
  uint8_t end;
} Move;
```

Here, `path` encodes the intermediate squares and captured pieces, while `begin` and `end` are the start and end indices.

This reduces memory usage (8 vs 16 bytes) at the cost of slightly more complex handling. In this representation, we cannot represent king moves if they were allowed to move any number of squares.

Monte Carlo Algorithm

Assume we play white and our opponent plays black. At some point in the game, we must select one of N available moves.

"Flat" Monte Carlo

The simplest approach is to simulate k random games for each possible move and then choose the move that maximizes wins minus losses:

$$\text{argmax}_{i=1,\dots,N} \sum_{j=1}^{k} s_j, \quad s_j = \begin{cases} +1 \text{ if } j\text{-th game resulted in white winning} \\ -1 \text{ if } j\text{-th game resulted in white losing} \\ \ \ 0 \text{ if } j\text{-th game resulted in a draw} \end{cases}$$

This approach maps naturally to GPUs — each thread plays a game to the end on its own board and reports the result. A kernel is launched for each initial move, and after completion, the results are aggregated.

Monte Carlo Tree Search

The flat method has a significant drawback — it spends the same time on every move. After a certain number of simulations, we often know which moves are promising and should focus more simulations on them.

Monte Carlo Tree Search (MCTS) can be divided into four phases:

1. *Selection*: Starting from the root (current position), select child nodes based on the outcomes of their subtrees.

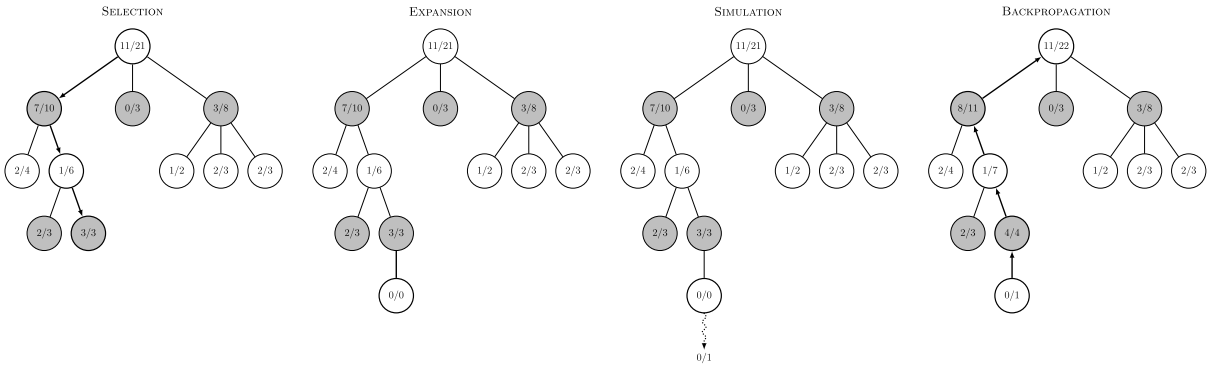
Typically, selection uses:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln(N_i)}{n_i}}$$

where $\frac{w_i}{n_i}$ is the win ratio for the subtree (favoring better moves), and $c\sqrt{\frac{\ln(N_i)}{n_i}}$ encourages exploration of less-visited nodes (N_i is the number of simulations for the parent).

2. *Expansion*: If the selected node is not a terminal game, perform a random move to create a new node.
3. *Simulation*: Play out one simulation from the new node to a terminal state.
4. *Backpropagation*: Update the nodes' statistics — wins and total simulations.

Draws can count as 0.5 wins, or all values can be scaled x2 to avoid using floats.



Implementation:

1. CPU selects a number of leaf nodes and *expands* each.
2. GPU simulates a fixed number of games (e.g., 32) for each selected leaf.
3. CPU performs *backpropagation*.
4. Repeat until the time limit for the move expires.

Random Number Generation

The algorithm needs a method to generate random numbers to select moves.

Speed is more important than high-quality randomness. A good choice seems to be xorshift32, which uses three shifts and three XORs:

```
uint32_t xorshift32(uint32_t *state) {
    uint32_t x = *state;
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    return *state = x;
}
```

`state` can be initialized per thread:

```
int id = blockIdx.x * blockDim.x + threadIdx.x;
uint32_t state = 0x9E3779B9 ^ id;
```

The magic constant `0x9E3779B9` is $\left\lfloor \frac{2^{32}}{\varphi} \right\rfloor$, often used for “random-looking” bits.