CUDA Warcaby

Zasady gry

Przyjęte zasady to warcaby amerykańskie.

- Plansza 8x8.
- Pionki ruszają się o jeden tylko do przodu.
- Króle ruszają się o jeden do przodu i do tyły.
- Przymus bicia – można wybrać dowolne z najdłuższych.
- Remis po 50 ruchach bez bicia.

Przyjęcie zasad amerykańskich w odróżnieniu do innych (w których dla przykładu króle mogą poruszać się dowolną ilość pól) pozwala na pewne ciekawe optymalizacje (o tym później).

Struktury danych

Plansza

Plansza reprezentowana jest na 12 bajtach:

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

Plansza warcabów ma wymiary 8x8 ale tylko połowa z tych pól jest grywalna zatem do zareprezentowania pozycji pionka potrzeba nam 32 bitów.

Przykładowo aby sprawdzić czy na pozycji o indeksie `x` znajduje się biały król:

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

Na pierwszy rzut oka intuicyjnym indeksowaniem wydawało by się coś w stylu:

```
   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
```

Zauważmy, że każdy pionek może poruszyć się o ± 4 (oprócz odpowiednio górnego i dolnego wiersza). Ale dodatkowo te w parzystych wierszach (numerowane od 0 od dołu) mogą ruszać się ± 3 a te w nieparzystych o ± 5. Powoduje to znaczne komplikacje algorytmu generowania dozwolonych ruchów (a co za tym idzie gorsza wydajność).

Okazuje się, że istnieje lepsze indeksowanie:

```
   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
```

Zauważmy, że tutaj każdy pionek (bez względu na parzystość wiersza) może ruszać się o ± 1, ± 7

Ruch

Pierwszym pomysłem jest następująca reprezentacja (16 bajtów):

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

`path` to tablica indeksów na ścieżce pionka (pole początkowe, pola pośrednie, pole końcowe). `captured` to maska bitowa pozycji zbitych pionków przeciwnika. Maksymalna długość tablicy `path` to 10, bo w jednym ruchu da się zbić maksymalnie 9 pionków przy przyjętych zasadsach.

Prawdopowobnie lepiej będzie przyjąć bardziej skompresowaną wersję (8 bajtów):

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

Tutaj `path` reprezentuje pola pośrednie w ścieżce oraz zbite pionki. `begin` i `end` to odpowiednio indeksy początku i końca ścieżki.

Uzyskujemy znacznie lepsze zużycie pamięci (8 vs 16 bajtów) kosztem nieco trudniejszego korzystania. W tej reprezencacji nie możemy na przykład zareprezentować ruchów króli jeżeli mogliby poruszać się o dowolną ilość pól.

Algorytm Monte Carlo

Przyjmijmy bso. że gramy białymi a nasz przeciwnik czarnymi pionkami. Jesteśmy w pewnym momencie rozgrywki i musimy wybrać jeden z N dostępnych ruchów.

"Flat" Monte Carlo

Najprostszym pomysłem jest rozegrać dla każdego z możliwych posunięć k losowych partii a następnie wybrać ruch maksymalizujący ilość wygranych minus ilość przegranych:

$$\text{argmax}_{i=1,\dots,N} \sum_{j=1}^{k} s_j, \quad s_j = \begin{cases} +1 \text{ jeśli } j\text{-ta gra zakończyła się wygraną białego} \\ -1 \text{ jeśli } j\text{-ta gra zakończyła się przegraną białego} \\ \ \ 0 \text{ jeśli } j\text{-ta gra zakończyła się remisem} \end{cases}$$

Takie podejście można w naturalny sposób napisać dla GPU – każdy wątek gra do końca na swojej planszy i raportuje wynik (wygrana/przegrana/remis). Wywołujemy kernel dla każdego możliwego ruchu początkowego i po wywołaniu każdego zliczamy jego wynik.

Monte Carlo Tree Search

Powyższa metoda ma znaczącą wadę – spędza tyle samo czasu na każdym ruchu. Po wykonaniu pewnej liczby symulacji często wiemy już które ruchy wydają się być lepsze od innych i to na nich powinniśmy się skupiać (alokować większą ilość symulacji).

Monte Carlo Tree Search można podzielić na 4 etapy:

1. *Selection*: zaczynamy z pozycji początkowej (korzenia) i wybieramy kolejne plansze (dzieci)

biorąc pod uwagę rezulataty gier w ich poddrzewach:

Najczęściej używana jest w selekcji wartość wyrażenia:

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

gdzie $\frac{w_i}{n_i}$ - wygrane w stosunku do ilości rozgrywek w poddrzewie (faworyzuje lepsze ruchy), $c\sqrt{\frac{\ln(N_i)}{n_i}}$ - faworyzuje mniej eksplorowane poddrzewa (N_i to liczba symulacji w rodzicu).

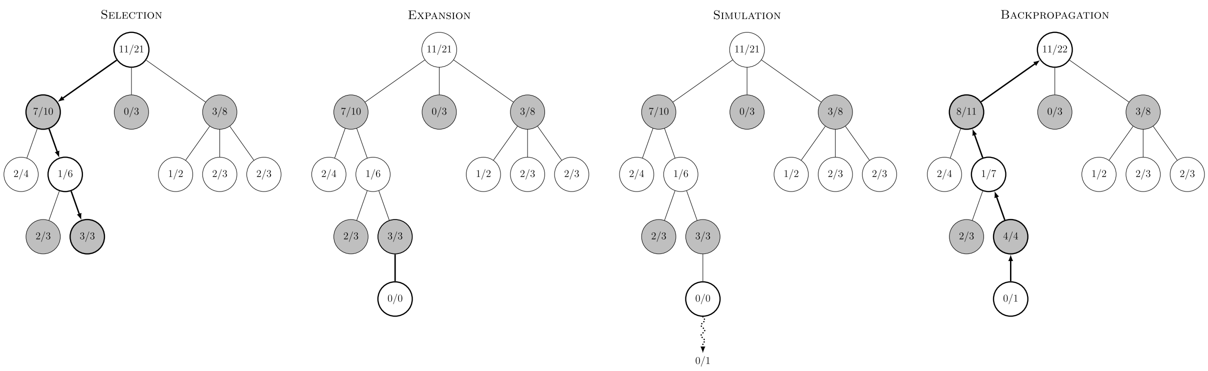
2. *Expansion*: Jeśli wybrany wierzchołek nie jest zakończoną grą, robimy losowy ruch,

tworząc nowy wierzchołek.

3. *Simulation*: Dla nowego wierzchołka wykonujemy jedną symulację (do zakończenia gry).

4. *Backpropagation*: Znając rezultat symulacji, aktualizujemy dane w wierzchołkach –

liczba wygranych i liczba rozgrywek. W przypadku remisu zwiększamy liczbę wygranych o 0.5 (ewentualnie można przeskalować wszystkie wartości x2 aby nie używać `float`ów).



TODO: napisać o tym, że tą metodę można przerwać w dowolnym momencie.

Generowanie liczb losowych

Algorytm potrzebuje metody generowania losowej liczby do wybrania jednego z dostępnych ruchów.

Liczy się dla nas raczej szybkość niż "jakość" generowania. Dobrym wyborem wydaje się xorshift32, który używa jedynie trzech shiftów i trzech xorów:

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

gdzie `seed` może być zainicjalizowany w każdym wątku:

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

Magiczna stała `0x9E3779B9` to $\lfloor 2^{32}/\varphi \rfloor$, często używana jako "losowe" bity.