Analiza i eksperymenty nad zrównoleglonym algorytmem MCTS-NC

Wojciech Bartoszek Jarosław Kołdun

Akademia Górniczo-Hutnicza w Krakowie
Wydział Informatyki

Przeznaczenie i cel rozwiązania

1. Przeznaczenie rozwiązania (MCTS-NC)

Artykuł źródłowy:
MCTS-NC: A thorough GPU parallelization of Monte Carlo Tree Search implemented in Python via numba.cuda (P. Klęsk, SoftwareX 2025).

Cel biblioteki:

- Implementacja algorytmu Monte Carlo Tree Search (MCTS) w środowisku wysoce zrównoleglonym (GPU).
- Umożliwienie szybkiego podejmowania decyzji w problemach o dużej przestrzeni stanów.

1. Przeznaczenie rozwiązania (cd.)

Obszary zastosowań:

- **Gry decyzyjne:** Szachy, Go, Connect4 (badane w projekcie).
- **Bioinformatyka:** Modelowanie 3D chromatyny, zwijanie RNA.
- **Logistyka i energetyka:** Optymalizacja przepływu ruchu, zarządzanie sieciami Smart Grid.

Narzędzia i implementacja

2. Narzędzia użyte do realizacji równoległości

Rozwiązanie opiera się na nowoczesnym stosie technologicznym Python, unikając konieczności pisania kodu w C++.

Kluczowe technologie

- **Python:** Język "gospodarza"(host code).
- **CUDA:** Platforma obliczeń równoległych na GPU firmy NVIDIA.
- **Numba:** Kompilator JIT (Just-In-Time), który tłumaczy podzbiór kodu Pythona i NumPy bezpośrednio na kod maszynowy GPU (kernele CUDA).

2. Narzędzia (cd.) – Dlaczego Numba?

Dlaczego Numba?

- Pozwala na pisanie kerneli CUDA bezpośrednio w Pythonie.
- Zapewnia wydajność zbliżoną do C/C++.
- Łatwa integracja z ekosystemem Data Science (NumPy).

Model zrównoleglenia

3. Sposób zrównoleglenia algorytmu

Algorytm MCTS składa się z 4 faz: *Selection, Expansion, Simulation (Playout), Backpropagation*. MCTS-NC zrównolegla **wszystkie** te etapy.

Poziomy zrównoleglenia:

1. **Leaf Parallelism:** Wiele wątków wykonuje symulacje z tego samego liścia.
2. **Root Parallelism:** Wiele niezależnych drzew budowanych równolegle.
3. **Tree Parallelism:** Wątki współpracują przy eksploracji jednego drzewa (unikając hazardów pamięciowych).

3. Sposób zrównoleglenia (cd.) – Szczegóły techniczne

Szczegóły techniczne:

- **Model pamięci:** Brak blokad (lock-free), wykorzystanie operacji atomowych tylko tam, gdzie to konieczne (lub unikanie ich przez redukcję).
- **Komunikacja:** Minimalny narzut transferu Host $\leftrightarrow$ Device (transfery tylko na początku i końcu).
- **Grid-stride loops:** Skalowalność niezależna od rozmiaru siatki wątków.

Analiza wydajności i wyniki

4. Opis eksperymentów (Wariant OCP_THRIFTY)

Zgodnie z podziałem zadań, nasza grupa skupiła się na wariancie **OCP_THRIFTY** (One-Child Policy, Memory Efficient).

Konfiguracja testowa:

- **Problem:** Gra Connect4.
- **Sprzęt:** GTX 1650 (4GB VRAM)
- **Monitorowanie:** NVML przez pynvml.

Badane parametry:

- N (liczba playoutów/symulacji) w zakresie od 32 do 2048.
- Wpływ na jakość decyzji (wartość Q) oraz wydajność (playouts/sec).

Wariant: OCP_THRIFTY

OCP (One Child Playout)

- Komputer wybiera jedno "dziecko"i wykonuje playout tylko dla niego.
- Pozwala na masowe zrównoleglenie symulacji na GPU — wiele "płytkich"analiz zamiast jednej głębokiej.

THRIFTY (Oszczędny)

- Alokacja pamięci na bieżąco — nie rezerwuje dużych tabel w VRAM.
- Mniej pamięci, więcej komunikacji CPU $\leftrightarrow$ GPU; stabilne działanie przy dużych drzewach bez OOM.

Podsumowanie: Wybraliśmy wariant **OCP_THRIFTY** jako kompromis między zrównolegleniem a oszczędnością pamięci.

Gra: Connect4

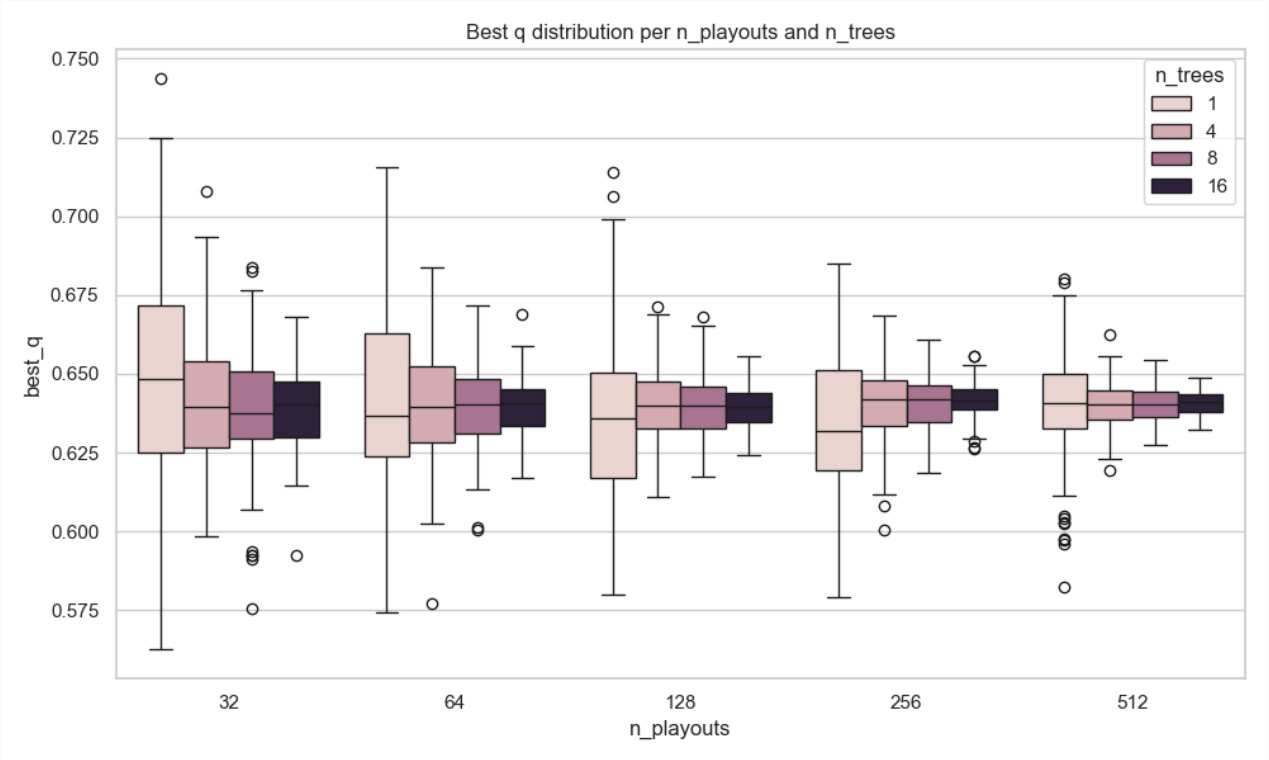
Zasady:

- Plansza: 6 wierszy x 7 kolumn.
- Cel: ułożyć cztery pionki w linii (poziomo, pionowo lub ukośnie).
- Ruchy: na przemian, pionki spadają do najniższego wolnego pola w kolumnie.

Przykładowa pozycja w Connect4.

Wyniki: Jakość decyzji (Best Q)

Badanie rozkładu wartości Q dla najlepszego ruchu w zależności od liczby symulacji.



Rysunek 1: Rozkład wartości Q. Wyższe N zmniejsza wariancję.

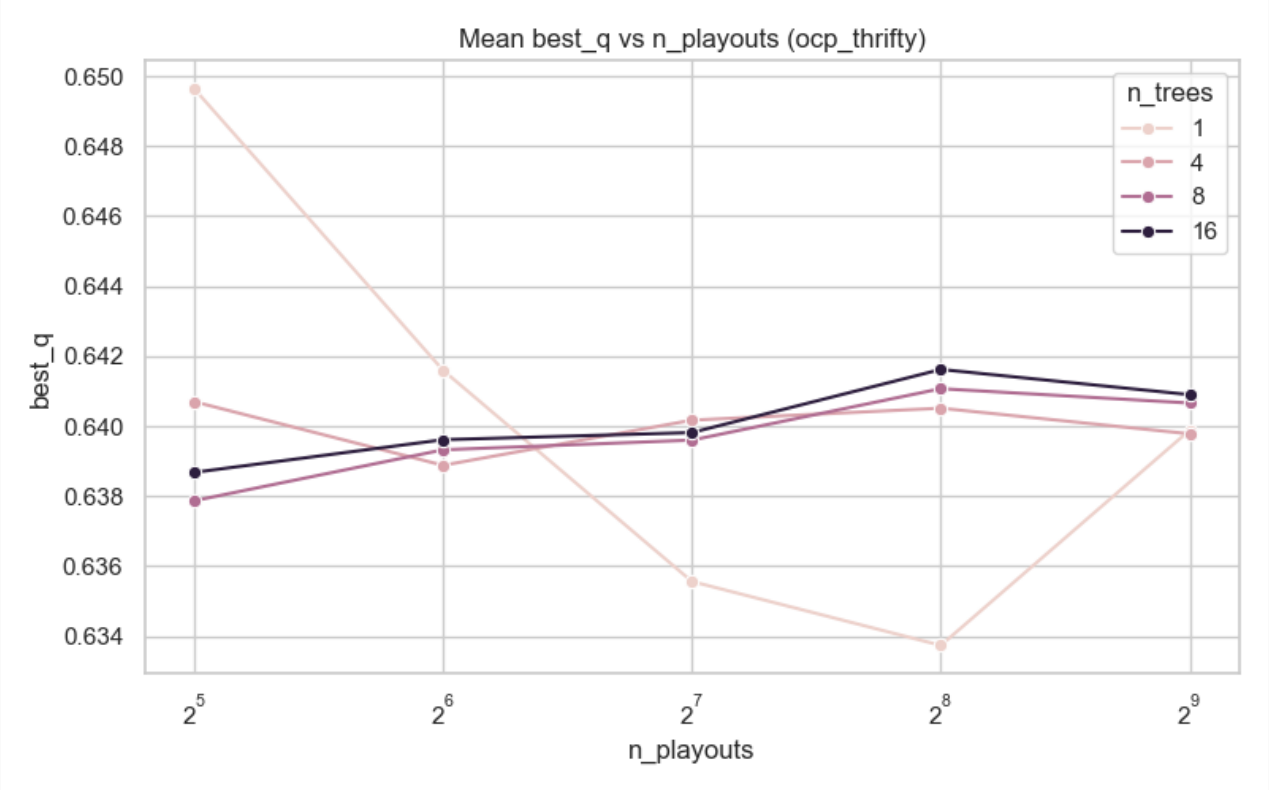
Wyniki: Zbieżność wartości Q

Zależność średniej wartości Q od liczby playoutów (N).

Wnioski:

- Wraz ze wzrostem N, wartość Q stabilizuje się.
- Dla wariantu Thrifty, przy $N > 512$ zyski jakościowe maleją (prawo malejących przychodów), ale rośnie pewność decyzji.

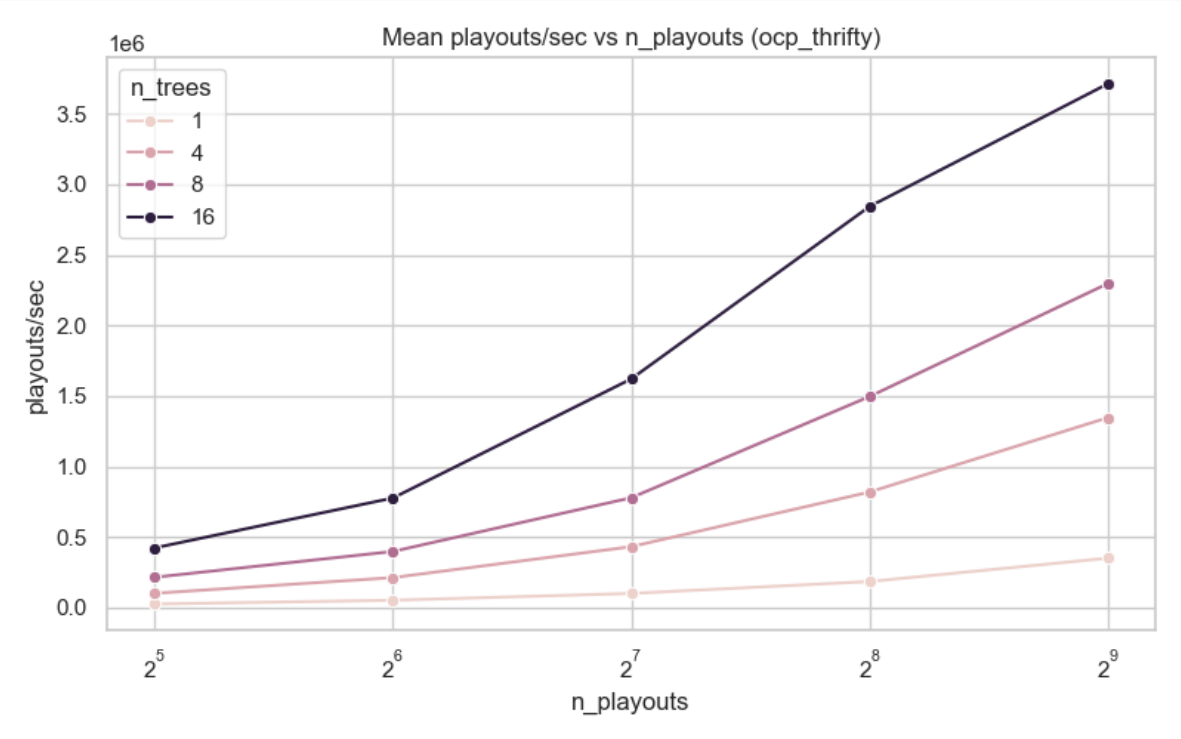
Wyniki: Zbieżność wartości Q – wykres



Rysunek 2: Średnie Best Q vs liczba playoutów.

Wyniki: Wydajność obliczeniowa

Przepustowość algorytmu (liczba symulacji na sekundę) w funkcji obciążenia (N).

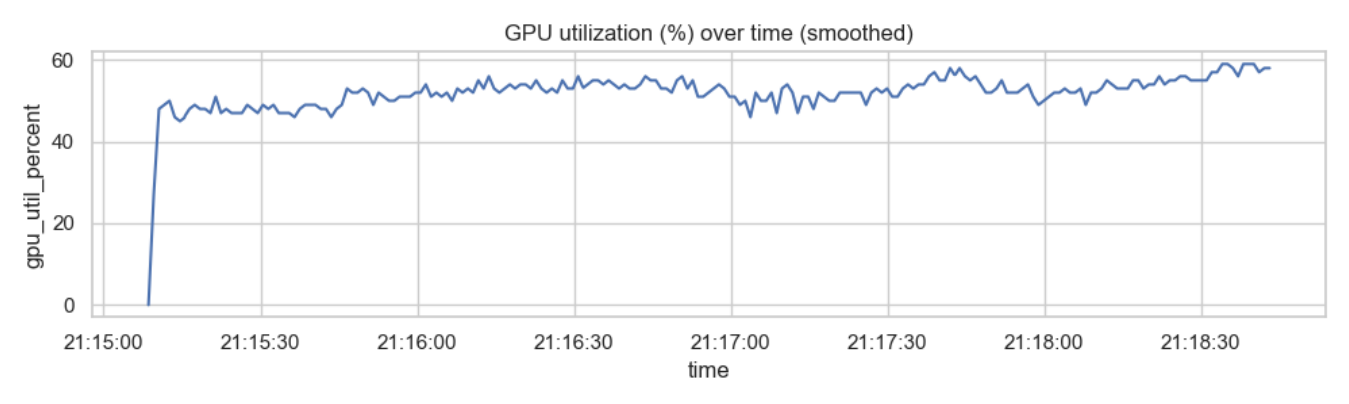


Rysunek 3: Liczba playoutów na sekundę.

Obserwacja: Wykres pokazuje nieliniowy wzrost.

Analiza obciążenia GPU (Monitoring)

Analiza utylizacji GPU w czasie trwania eksperymentu.



Rysunek 4: Utylizacja GPU w czasie.

- Średnie obciążenie utrzymuje się na średnim poziomie (55%).
- Obciążenie GPU jest stabilne w czasie, nie ma widocznych drastycznych spadków/wzrostów.

Propozycja ulepszenia

Propozycja ulepszenia wydajności

Na podstawie analizy wydajności i kodu źródłowego proponujemy optymalizację:

Weighted Depth Simulation (Ważona głębokość symulacji)

Problem
Obecnie symulacje (playouts) trwają do końca gry lub sztywnego limitu, marnując zasoby na oczywiste pozycje.

Propozycja ulepszenia – Rozwiązanie

Rozwiązanie

- Wprowadzenie tablicy wag dla głębokości przesyłanej do **pamięci stałej** GPU.
- Skalowanie pętli symulacji wewnątrz kernela CUDA w oparciu o dynamikę gry.

Oczekiwany efekt: Głębsze przeszukiwanie w krytycznych momentach przy tym samym budżecie czasowym.

Podsumowanie (1/2)

1. **MCTS-NC** to skuteczne narzędzie wykorzystujące Pythona i Numba do obliczeń GPGPU.
2. Zrównoleglenie na poziomie liści, korzenia i drzewa pozwala na pełne wykorzystanie architektury GPU.

Podsumowanie (2/2)

3. Eksperymenty dla wariantu **OCP_THRIFTY** w Connect4 wykazały:
   - Wysoką skalowalność przepustowości wraz ze wzrostem N.
   - Efektywne wykorzystanie GPU (niskie narzuty komunikacyjne).
4. Zaproponowano optymalizację zarządzania głębokością symulacji z użyciem pamięci stałej.