

Analiza i eksperymenty nad zrównoległym algorytmem MCTS-NC

Wojciech Bartoszek
Jarosław Kołodun

Streszczenie

Niniejszy raport stanowi sprawozdanie z analizy i testów oprogramowania MCTS-NC (Monte Carlo Tree Search - numba.cuda). Dokument opisuje cel rozwiązania, zastosowany model zrównoleglenia GPU oraz narzędzia implementacyjne. Przedstawiono również wyniki własnych eksperymentów dla wariantu OCP_THRIFTY w grze Connect4, badając wpływ liczby drzew i playoutów na wydajność i jakość decyzji. Dodatkowo przeprowadzono analizę wydajności przy pomocy systemu monitorującego.

1 Przeznaczenie rozwiązania

Omawiane rozwiązanie, MCTS-NC, jest biblioteką służącą do realizacji algorytmu Monte Carlo Tree Search (MCTS) w środowisku wysoce zrównoległym. MCTS to algorytm uczenia ze wzmocnieniem (Reinforcement Learning), który buduje asymetryczne drzewo gry poprzez selektywne próbkowanie akcji i estymację ich wartości na podstawie losowych symulacji (playouts) [1].

Główne obszary zastosowań tego rozwiązania to:

- **Gry decyzyjne:** Szachy, Go, Connect4, Gomoku (weryfikowane w artykule źródłowym).
- **Bioinformatyka:** np. modelowanie 3D chromatyny czy zwijanie RNA.
- **Inżynieria ruchu i energetyka:** optymalizacja przepływu ruchu, zarządzanie siecią energetyczną.
- **Matematyka:** odkrywanie algorytmów mnożenia macierzy.

Celem MCTS-NC jest dostarczenie szybkiej, działającej wyłącznie na GPU implementacji, która eliminuje wąskie gardła związane z przesyłaniem danych między CPU a GPU.

2 Opis zrównoleglenia algorytmu

Algorytm został zrównoleglony w oparciu o model obliczeniowy **CUDA (Compute Unified Device Architecture)** Implementacja łączy trzy poziomy zrównoleglenia (Leaf, Root, Tree parallelization) w jeden spójny system.

2.1 Model podziału i mapowanie wątków

Podział pracy pomiędzy wątki GPU zależy od etapu algorytmu i wybranego wariantu:

- **Organizacja bloków:** Wątki są grupowane w bloki CUDA, które są indeksowane albo przez same indeksy drzew, albo przez pary (drzewo, akcja).
- **Etapy MCTS:** Wszystkie cztery etapy algorytmu (selekcja, ekspansja, symulacja/playout, wsteczna propagacja/backup) wykorzystują wiele wątków GPU.

- **Redukcje:** Do sumowania wyników oraz wyznaczania wartości maksymalnych ($\max/\operatorname{argmax}$) zastosowano wzorce redukcji (reduction patterns), co pozwala na obliczenia w czasie $O(1)$ lub $O(\log n)$.

2.2 Zarządzanie pamięcią i komunikacja

Implementacja jest typu **lock-free** (bez blokad) – nie używa operacji atomowych ani mutexów, co jest kluczowe dla wydajności przy masowej równoległości.

- **Współpraca wątków:** Wykorzystano mechanizm kooperacji wątków (threads cooperation) oraz szybką pamięć współdzieloną (shared memory) wewnątrz bloków.
- **Transfer danych:** Zminimalizowano komunikację Host-Device. W wariancie *Prodigal* transfery pamięci są niemal całkowicie wyeliminowane w głównej pętli.

2.3 Warianty algorytmiczne

Zaimplementowano cztery warianty różniące się sposobem alokacji zasobów:

- **OCP (One-Child Playouts) vs ACP (All-Children Playouts):** W OCP symulowany jest jeden losowy potomek liścia, w ACP – wszyscy potomkowie równolegle.
- **Thrifty vs Prodigal:**
 - **Thrifty (Oszczędny):** Liczba bloków jest dokładnie dopasowana do liczby legalnych akcji. Wymaga to jednak transferów pamięci do hosta w celu przeliczenia indeksów.
 - **Prodigal (Rozrzutny):** Alokuje nadmiarową siatkę bloków (T, B) . Marnuje pewne zasoby GPU, ale unika kosztownych transferów pamięci Host-Device.

3 Narzędzia realizacyjne

Do realizacji równoległości wykorzystano następujące technologie:

- **Język:** Python (wersja ≥ 3.13).
- **Biblioteka równoległości: Numba** (moduł `numba.cuda`). Jest to kompilator JIT (Just-In-Time), który tłumaczy kod Pythona na instrukcje PTX wykonywalne na kartach graficznych NVIDIA.
- **Sprzęt:** RTX 1650; CUDA 12.8.

W przeciwieństwie do rozwiązań opartych na MPI czy OpenMP działających na CPU, użycie `numba.cuda` pozwala na bezpośrednie pisanie jąder (kernels) CUDA w Pythonie.

4 Przebieg eksperymentów

Celem eksperymentów (demo) było ocenienie zachowania wariantu `OCP_THRIFTY` dla gry Connect4 w różnych konfiguracjach: `n_trees = {1,4,8,16}` i `n_playouts = {32,64,128,256,512}`. Każda konfiguracja została wielokrotnie testowana (100 prób).

4.1 Metodologia eksperymentalna

- Wariant algorytmu: `OCP_THRIFTY`
- Gra: Connect 4
- Miary: `playouts_per_second`, `best_q` (średnia wygranych dla wybranej akcji).

4.2 Analiza wydajności i monitoring

W ramach projektu przeprowadzono analizę wydajności aplikacji przy użyciu systemu monitorującego. Poniżej przedstawiono plan, proponowane metryki oraz wyniki pomiarów z instrumentacji.

4.2.1 Cele i metryki

Analiza skupi się na następujących metrykach: wykorzystanie GPU (%), wykorzystanie pamięci GPU, wykorzystanie CPU (%), zużycie pamięci RAM procesu, liczba playoutów na sekundę (playouts/s), czasy etapów (select/expand/playout/backup), transfery host \leftrightarrow device oraz, o ile to możliwe, czasy poszczególnych kernelów.

4.2.2 Plan eksperymentów i wyniki

Po uruchomieniu eksperymentów z instrumentacją wygenerowano poniższą tabelę podsumowującą miary monitoringu oraz wykres wykorzystania GPU.

metric	mean	std / max
CPU (%)	18.8	6.5
GPU util (%)	51.9	4.9
GPU mem mean (MB)	1846.6	max 2339.4
RSS mem mean (MB)	438.2	

Wnioski: GPU było umiarkowanie obciążone podczas eksperymentów; rozważamy optymalizację parametrów bloków oraz redukcję transferów host \leftrightarrow device w celu lepszego wykorzystania dostępnych zasobów GPU.

4.2.3 Proponowane ulepszenia — hipotezy do weryfikacji

- Zredukowanie transferów host \leftrightarrow device i minimalizacja kopiowania danych (oczekiwane zwiększenie przepustowości i zmniejszenie opóźnień).
- Dopasowanie parametrów bloków/wątków (tpb) oraz rozkładu pracy w kernelach w celu lepszego wykorzystania SM GPU.
- Usprawnienie generatora losowości (np. pre-allocacja stanów RNG) i optymalizacja inicjacji per-thread, aby uniknąć kosztów w pętli programu.
- Profilowanie kernelów i rozważenie łączenia prostych kernelów (kernel fusion) lub przeniesienia części logiki na host w celu redukcji liczby wywołań kerneli.

4.3 Zestawienie wyników

Poniżej wstawiono tabelę podsumowującą średnie wartości ($mean \pm std$) dla szybkości playoutów i jakości akcji.

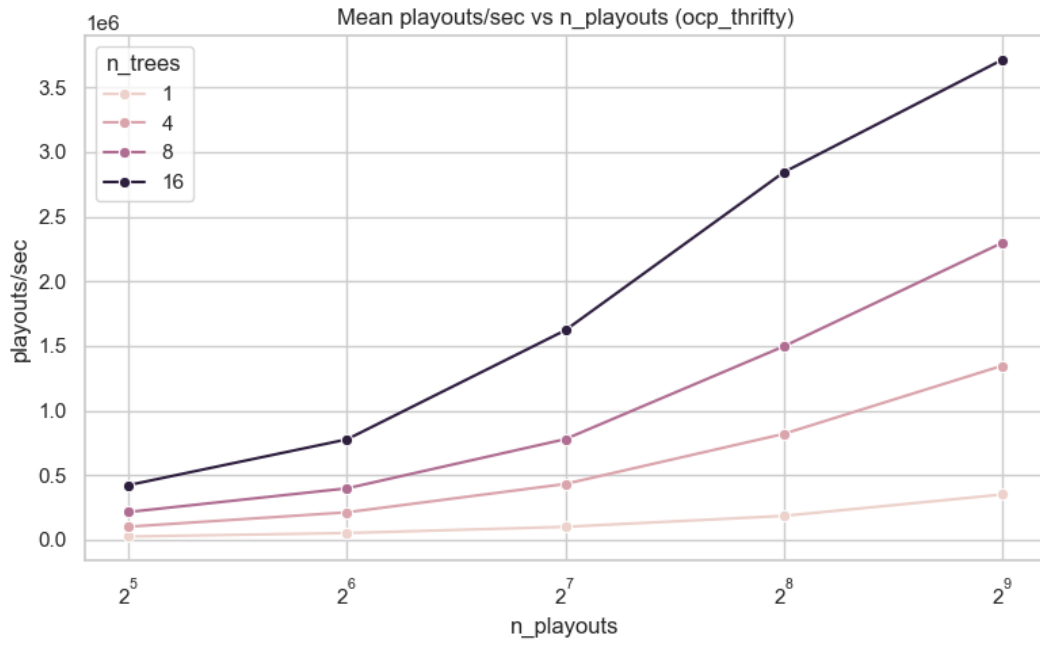
n_trees	n_playouts	playouts/s (mean \pm std)	best_q (mean \pm std)	trials
1	64	52,265 \pm 5,265	0.6416 \pm 0.0302	100
1	128	100,643 \pm 8,357	0.6356 \pm 0.0257	100
1	256	185,010 \pm 11,775	0.6337 \pm 0.0229	100
1	512	350,306 \pm 23,495	0.6399 \pm 0.0176	100
4	32	101,651 \pm 7,923	0.6407 \pm 0.0219	100
4	64	212,774 \pm 15,353	0.6389 \pm 0.0187	100
4	128	432,024 \pm 31,429	0.6402 \pm 0.0126	100
4	256	819,187 \pm 53,015	0.6405 \pm 0.0122	100
4	512	1,346,204 \pm 99,740	0.6398 \pm 0.0079	100
8	32	215,212 \pm 15,660	0.6379 \pm 0.0195	100
8	64	398,186 \pm 47,392	0.6393 \pm 0.0132	100
8	128	778,756 \pm 88,164	0.6396 \pm 0.0098	100
8	256	1,495,930 \pm 118,487	0.6411 \pm 0.0083	100
8	512	2,296,589 \pm 133,618	0.6407 \pm 0.0059	100
16	32	423,327 \pm 31,679	0.6387 \pm 0.0131	100
16	64	777,088 \pm 71,423	0.6396 \pm 0.0092	100
16	128	1,620,980 \pm 101,277	0.6398 \pm 0.0066	100
16	256	2,842,605 \pm 185,688	0.6416 \pm 0.0060	100
16	512	3,712,266 \pm 170,840	0.6409 \pm 0.0037	100

4.4 Najważniejsze obserwacje

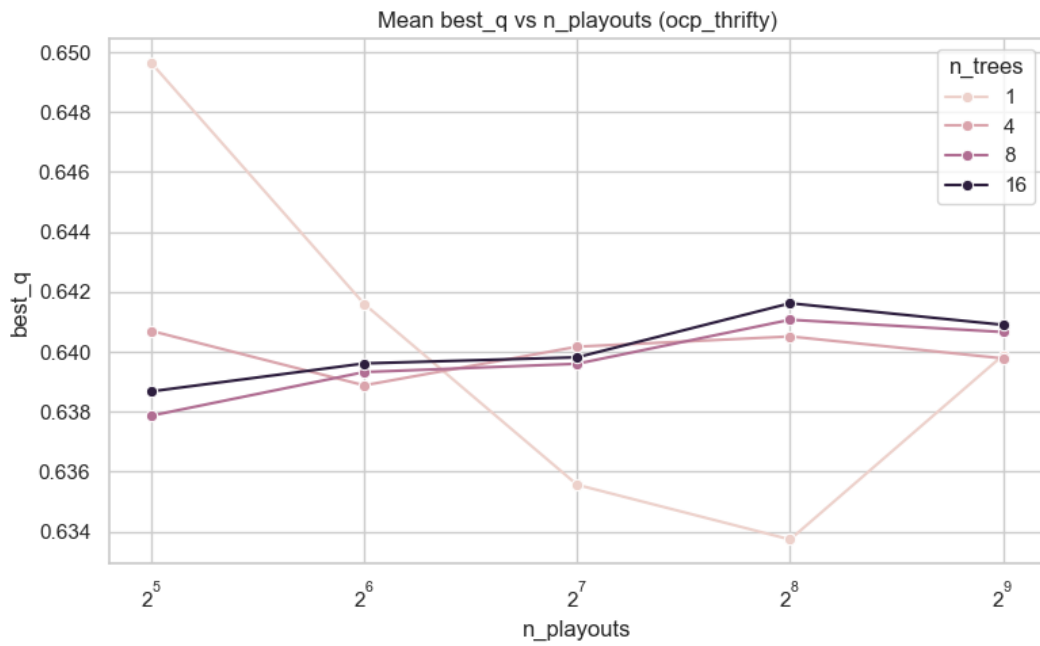
- Najwyższą średnią liczbę playoutów na sekundę zaobserwowano dla konfiguracji **n_trees=16**, **n_playouts=512**.
- Najwyższą średnią jakość decyzji (**best_q**) uzyskano dla **n_trees=1**, **n_playouts=32** (średnia około 0.6496).
- Ogólnie: szybkość playoutów rośnie wraz ze wzrostem **n_trees** i/lub **n_playouts**, natomiast średnia jakość (**best_q**) jest względnie stabilna w mierzonych konfiguracjach.

4.5 Wykresy

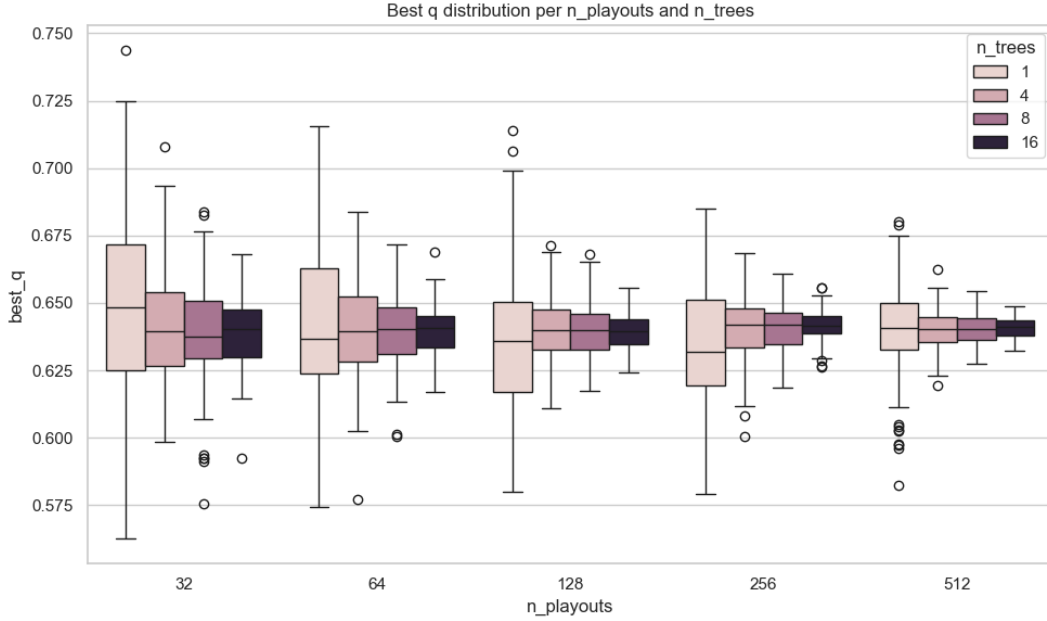
Poniżej zamieszczono kluczowe wykresy wygenerowane podczas eksperymentów.



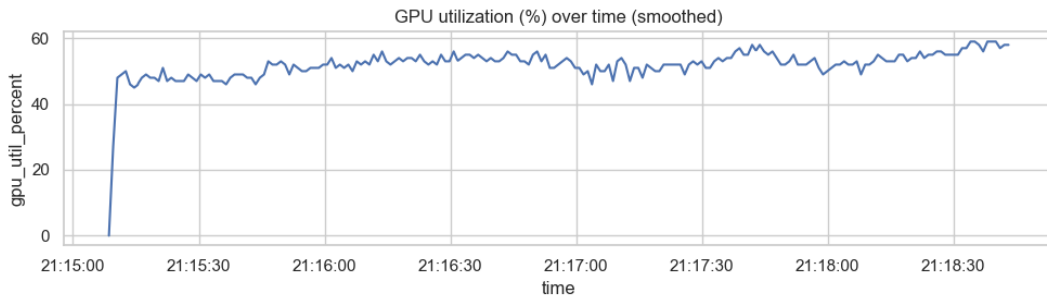
Rysunek 1: Średnie playouts/sec w zależności od $n_playouts$ dla różnych wartości n_trees .



Rysunek 2: Średnie $best_q$ w zależności od $n_playouts$ dla różnych wartości n_trees .



Rysunek 3: Boxploty rozkładu *best_q* dla różnych konfiguracji *n_trees* i *n_playouts*.



Rysunek 4: Wykres wykorzystania GPU w czasie podczas eksperymentu.

5 Dyskusja i propozycja ulepszenia

Wyniki eksperymentów wskazują, że samo zwiększanie przepustowości (liczby playoutów na sekundę) nie przekłada się liniowo na jakość decyzji w wariancie OCP. W związku z tym, zamiast prostej zmiany wariantu na bardziej zasobożerny (ACP), proponujemy optymalizację algorytmiczną.

5.1 Propozycja: Dynamiczne skalowanie liczby playoutów

Zgodnie z otwartymi pytaniami badawczymi postawionymi przez autora oprogramowania, proponujemy implementację mechanizmu **zmiennej liczby playoutów w zależności od głębokości drzewa**.

- **Problem:** Obecnie parametr m (liczba playoutów) jest stały dla każdego węzła. Powoduje to, że zasoby GPU są zużywane równomiernie, nawet w sytuacjach, gdzie ruchy są oczywiste lub mało znaczące dla wyniku końcowego.
- **Rozwiązanie:** Modyfikacja jądra `_playout_ocp` tak, aby parametr m był funkcją głębokości węzła d , np. $m(d) = m_{base} \cdot f(d)$.

- **Oczekiwany efekt:** Pozwoli to na "głębsze" przeszukiwanie w krytycznych momentach gry przy zachowaniu tego samego budżetu czasowego, co powinno podnieść wskaźnik wygranych bez konieczności drastycznego zwiększania pamięci (jak w wariancie Prodigal).
- **Plan wdrożenia:** Wprowadzenie tablicy wag dla głębokości przesyłanej do pamięci stałej (constant memory) GPU i skalowanie pętli symulacji wewnątrz kernela CUDA.

Literatura

- [1] Przemysław Kłęsk. "MCTS-NC: A thorough GPU parallelization of Monte Carlo Tree Search implemented in Python via numba.cuda". W: *SoftwareX* 30 (2025), s. 102139. ISSN: 2352-7110. DOI: <https://doi.org/10.1016/j.softx.2025.102139>. URL: <https://www.sciencedirect.com/science/article/pii/S2352711025001062>.
- [2] Przemysław Kłęsk. *Presentation on MCTS-NC: A thorough GPU parallelization of Monte Carlo Tree Search implemented in Python via numba.cuda*. Styczeń 2026.
- [3] Siu Kwan Lam, Antoine Pitrou i Stanley Seibert. "Numba: a LLVM-based Python JIT compiler". W: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. URL: <https://doi.org/10.1145/2833157.2833162>.
- [4] Maciej Swiechowski i in. "Monte Carlo Tree Search: A Review of Recent Modifications and Applications". W: *CoRR* abs/2103.04931 (2021). arXiv: 2103.04931. URL: <https://arxiv.org/abs/2103.04931>.