Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

MCTS-NC: A thorough GPU parallelization of Monte Carlo Tree Search implemented in Python via numba.cuda

Przemysław Klęsk

Faculty of Computer Science and Information Technology, West Pomeranian University of Technology in Szczecin, ul. Żołnierska 49, 71-210 Szczecin, Poland

ARTICLE INFO

Keywords:
MCTS parallelization
CUDA
Python
Numba
Bounds on action values estimates

ABSTRACT

With CUDA computational model in mind, we introduce MCTS-NC (Monte Carlo Tree Search–`numba.cuda`). It contains four, fast-operating and thoroughly parallel, variants of the MCTS algorithm. The design of MCTS-NC combines three parallelization levels (leaf / root / tree parallelizations). Additionally, all algorithmic stages – selections, expansions, playouts, backups – employ multiple GPU threads. We apply suitable *reduction* patterns to carry out summations or max / argmax operations. The implementation uses very few device-host memory transfers, no atomic operations (is lock-free), and takes advantage of threads cooperation. In the mathematical part of this article, we demonstrate how the confidence bounds on estimated action values become tightened by both the number of independent concurrent playouts and the number of independent concurrent trees. The experimental part reports the performance of MCTS-NC on two game examples: Connect4 and Gomoku. All computational results can be exactly reproduced.

Code metadata

Current code version	1.0.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00708
Permanent link to Reproducible Capsule	
Legal Code License	CC-BY-4.0 license
Code versioning system used	git
Software code languages, tools, and services used	Python, Numba (numba.cuda)
Compilation requirements, operating environments & dependencies	Python >= 3.8.10, numba >= 0.58, nvcc >= 11.4, numpy >= 1.22, CUDA drivers in OS
If available Link to developer documentation/manual	https://pklesk.github.io/mcts_numba_cuda
Support email for questions	pklesk@zut.edu.pl

Software metadata

Current software version	1.0.1
Permanent link to executables of this version	https://github.com/pklesk/mcts_numba_cuda/releases/tag/v1.0.1
Permanent link to Reproducible Capsule	
Legal Software License	CC-BY-4.0 license
Computing platforms/Operating Systems	Linux, Microsoft Windows
Installation requirements & dependencies	Python >= 3.8, numba >= 0.58, nvcc >= 11.4, numpy >= 1.22, CUDA drivers in OS
If available, link to user manual — if formally published include a reference to the publication in the reference list	https://github.com/pklesk/mcts_numba_cuda#readme
Support email for questions	pklesk@zut.edu.pl

E-mail address: pklesk@zut.edu.pl.

https://doi.org/10.1016/j.softx.2025.102139
Received 30 December 2024; Received in revised form 13 March 2025; Accepted 18 March 2025
Available online 1 April 2025

1. Motivation and significance

Recent years have shown important, often unexpected, discoveries due to reinforcement learning (RL) and algorithms for sequential decision problems. Those discoveries pertain to numerous and quite versatile domains, including games and game theory, robotics, bioinformatics, economics, physics, operational research, and many others [1–5].

Monte Carlo Tree Search (MCTS) is one of RL algorithms [6–8]. In brief, MCTS grows an asymmetric game tree by sampling actions selectively and reestimating their values based on random playouts carried out from the expanded leaves. Apart from games and famous Alpha Go / Zero projects [1,2], MCTS led to successes in various fields: bioinformatics [3,4], energetics [5], traffic engineering [9] or algebra [10]. MCTS is *non-dogmatic* — requires no evaluation function, no human knowledge; is purely driven by the terminal rewards (e.g., losses, draws, wins).

Our project, named MCTS-NC, contributes to the computational elements of Monte Carlo tree searches. The fast-operating implementation of MCTS-NC takes advantage of Numba (https://numba.pydata.org), a just-in-time Python compiler, and its `numba.cuda` package. We believe the project has a strong potential for recognition among researchers and practitioners. The combination of reasons presented in the paragraphs to follow justifies the significance of MCSTS-NC software.

First of all, the analysis of literature from the period of 2012–2023 (e.g., [11–16]) shows that the known ideas of leaf-/root-/tree-paralleliz ationfor MCTS are still under active research. The first two types translate into more accurate estimates of action values via multiple concurrent playouts or trees, respectively. The third pertains to nodes processing aspects. Nevertheless, all three types are seldom fully integrated, and GPU-only realizations of MCTS are also uncommon. Often, the leaf-parallelization runs on a GPU, while the root-parallelization runs on a multi-core CPU. In contrast, the MCTS-NC software constitutes a complete and GPU-only implementation (Fig. 1 provides a high-level intuition on MCTS-NC and its CUDA computations).

Secondly, surprisingly little has been written about how CUDA-specific techniques – *reductions*, *threads cooperation*, and other – can enhance the performance of individual stages of MCTS. Note, for example, that even for a large branching factor b, the UCB values[1] at one level of the selection stage can be computed *simultaneously*[2] for all children, hence in $O(1)$ time (constant time, regardless of b), and the subsequent max / argmax-reduction can be computed in $O(\log b)$ time. Also, the backup stage can be performed cheaply without a loop — in $O(1)$ time, provided that the path of selected nodes has been memorized. Savings alike are present in MCTS-NC. Beside the aforementioned properties, MCTS-NC implementation uses no atomic operations, no mutexes (lock-free implementation), and very few device-host memory transfers.

Finally, it is worth remarking that, in general, CUDA implementations of Monte Carlo simulations are naturally low-level ones. They tend to be specialized, tailored to specific problems or games (e.g., electron–photon interactions [17], chromatin 3D modeling [4], Go [11], Amazon Chess [18], Da Vinci Code [19]). On the other hand, MCTS software frameworks [9,20–22] strive for user convenience via object-oriented APIs, and understandably "end up" in CPU-dedicated implementations. In our opinion, MCTS-NC constitutes a decent attempt at coupling CUDA's low-levelness (kernel functions, device-side functions) with an Object-Oriented-like design and generality.

[1] upper confidence bounds on action values

[2] Here, "simultaneously" refers to a stronger property than "concurrently" — counts of CUDA cores and SMs in contemporary GPU devices allow it even for games with very large branching factors.

2. Mathematical and algorithmic description

We begin this section with mathematical preliminaries that led to the development of the UCT approach (upper confidence bounds for trees), and in consequence the MCTS algorithm. We end it with the algorithmic description of MCTS-NC and its properties.

2.1. Hoeffding's inequalities

For the mean $\overline{X}_n = 1/n(X_1 + \cdots + X_n)$ of n independent random variables X_i, bounded as follows: $\forall_i\, 0 \leqslant X_i \leqslant 1$, true are the following two versions of one-sided Hoeffding's inequalities:

$$P\left(\pm(\overline{X}_n - \mathbb{E}\overline{X}_n) \geqslant \epsilon\right) \leqslant e^{-2\epsilon^2 n}, \tag{1}$$

where $\epsilon > 0$ denotes a certain small amount by which the mean deviates from its expectation (more details given in Appendix A). By substituting the right-hand side of (1) with small probability $\delta > 0$, $e^{-2\epsilon^2 n} := \delta$, and solving for ϵ, one can equivalently state that: with probability at least $1 - \delta$ the bound $\mathbb{E}\overline{X}_n \leqslant \overline{X}_n + \sqrt{-\log\delta/(2n)}$ holds true. In particular, the bound is true also when X_is are not only independent but also identically distributed (i.i.d.), which implies $\mathbb{E}X_i = \mathbb{E}X$, and allows to write:

$$\mathbb{E}X \leqslant \overline{X}_n + \sqrt{-\log\delta/(2n)}. \tag{2}$$

In the context of games or decision problems, inequality (2) is a useful tool allowing one to upperbound the true, but unknown, expected values of *actions*, based on the observed means of rewards generated by Monte Carlo simulations.

2.2. Multi-armed bandits, regret, UCB1

Consider a gambling machine with b independent arms and let $a \in \{1, \ldots, b\}$ denote the index of an arm (or action for further contexts). Let $X_{a,1}, X_{a,2}, \ldots, X_{a,n_a}$ be the sequence of rewards coming from the ath arm, and n_a be the number of times a was selected within the total of n plays, $n = \sum_{a=1}^{b} n_a$. Expected rewards of arms $q_a = \mathbb{E}(X_a)$ are unknown and one would like to find a policy (strategy) that picks the best arm associated with $q^* = \max_{1\leqslant a\leqslant b} q_a$ as frequently as possible.

The notion of *regret* related to this problem —

$$R_n = nq^* - \mathbb{E}\left(\sum_{a=1}^{b}\sum_{i=1}^{n_a} X_{a,i}\right) = nq^* - \sum_{a=1}^{b} q_a \mathbb{E}(n_a) = \sum_{a=1}^{b}(q^* - q_a)\mathbb{E}(n_a). \tag{3}$$

— quantifies the loss caused by a policy not always picking the best arm. Lai and Robbins [23] showed that for a large class of rewards' distributions the best possible regret grows at least logarithmically, i.e.: $R_n \sim \Omega(\log n)$. Hence, a policy is deemed to properly solve exploration–exploitation trade-off if its regret is within a constant factor of the best possible: $R_n \sim \Theta(\log n)$. Building upon [23], Auer et al. [24] proposed several policies for the bandits problem, in particular a simple deterministic one named UCB1 (*upper confidence bound*) that, after n rounds, dictates to pick the arm:

$$\arg\max_{1\leqslant a\leqslant b}\left(\overline{X}_{a,n_a} + \sqrt{2\log n/n_a}\right), \tag{4}$$

where $\overline{X}_{a,n_a} = 1/n_a \sum_{i=1}^{n_a} X_{a,i}$. The width $\sqrt{2\log n/n_a}$ of the bound is chosen cleverly. Firstly, when equating it with $\sqrt{-\log\delta/(2n_a)}$ from (2) one obtains $\delta = n^{-4}$. This implies that the upper bound on unknown expectation: $q_a \leqslant \overline{X}_{a,n_a} + \sqrt{2\log n/n_a}$ is satisfied with a high probability of at least $1 - 1/n^4$. Secondly, it guarantees a proper exploration–exploitation balance. A high value of $\overline{X}_{a,n_a}$ suggests exploiting a. When an arm keeps on being selected over many plays, and both n and n_a are incremented simultaneously, then the bound tightens towards q_a due to the denominator in the second term growing faster than the numerator. On the other hand, for all non-selected arms, the $1/n_a$ expression remains fixed and only $\log n$ grows, which causes any such arm to be eventually selected (explored).

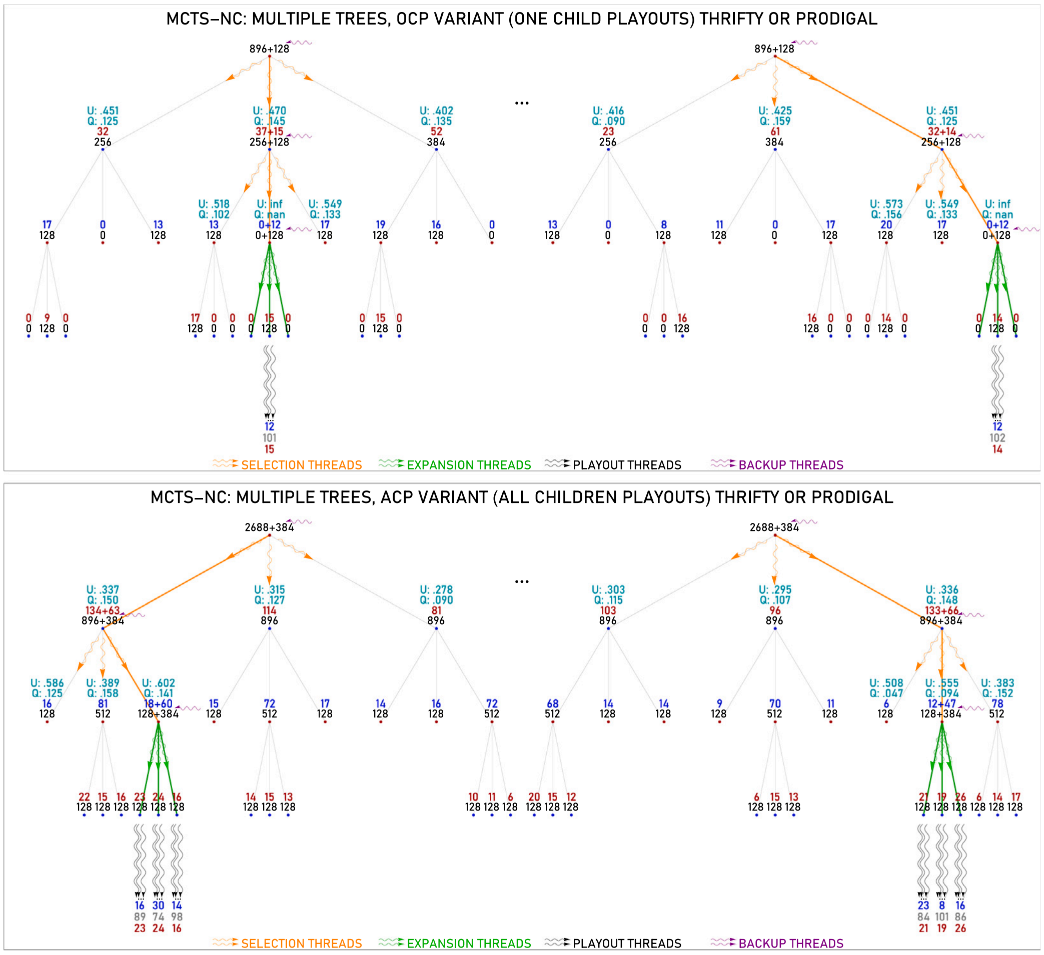
MCTS–NC: MULTIPLE TREES, OCP VARIANT (ONE CHILD PLAYOUTS) THRIFTY OR PRODIGAL
SELECTION THREADS
EXPANSION THREADS
PLAYOUT THREADS
BACKUP THREADS
MCTS–NC: MULTIPLE TREES, ACP VARIANT (ALL CHILDREN PLAYOUTS) THRIFTY OR PRODIGAL

Fig. 1. High-level view on two main algorithmic variants present in MCTS-NC: OCP and ACP (each having two subvariants: thrifty or prodigal).

2.3. *UCT — Upper confidence bounds for trees*

Kocsis and Szepesvári [6] wondered whether UCB1 (or a similar bound) can be applied to MCTS at *all* internal tree nodes and correctly guide the process via selective sampling of actions. They pointed out the non-stationarity of the bandits problem that occurs in this case — as the algorithm progresses, the sampling probabilities of actions below each given node *drift* in time. They proved that under some mild assumptions (presented in Appendix B) the algorithm is still consistent[3] despite the drift, provided that a parametric form $c_{n,n_a} = 2C_0\sqrt{\log n/n_a}$ for the bound width is used with an appropriate choice of $C_0 > 0$ constant. Probabilities of too large deviations can be then expressed as

$$P\left(\pm(q_a - \overline{X}_{a,n_a}) \geqslant c_{n,n_a}\right) \leqslant e^{-2n_a \frac{4C_0^2 \log n}{n_a}} = n^{-8C_0^2}, \tag{5}$$

which coincides with UCB1 when $C_0 = 1/\sqrt{2}$.

The contemporarily popular MCTS formulation is such that the returned scores are in fact estimates of *win probabilities* conditional on actions. Hence, the commonly met convention for writing UCT is:

$$q_a \leqslant \hat{q}_{a,n_a} + C\sqrt{\frac{\log n}{n_a}} = \frac{w_a}{n_a} + C\sqrt{\frac{\log n}{n_a}}, \tag{6}$$

where: the factor of 2 is skipped for simplicity ($C = 2C_0$), $\hat{q}_{a,n_a}$ corresponding to $\overline{X}_{a,n_a}$ is an estimate of action-value, i.e., of win probability, and w_a denotes the number of wins experienced after taking action a.[4] For strictness we should explain that in the above notation some state s, corresponding to a tree node, is always implicitly considered. This means that: $n \equiv n(s)$ denotes the number of visits in s or equivalently (for now) the number of performed simulations (playouts) going through s, $q_a \equiv q(s,a)$ is the true probability of win after taking action a in s and $\hat{q}_{a,n_a}$ is its estimate under the UCT-based tree policy, $n_a \equiv n(s,a)$ is the number of times a was taken in s. At implementation level, when $n_a = 0$, the right-hand side of (6) is defined to be ∞ (e.g., `np.inf`) to make a selected in the first order (with ties of infinities broken arbitrarily).

2.4. *High-level intuition on MCTS-NC*

Fig. 1 provides a high-level intuition on MCTS-NC and the two main variants according to which it conducts the playouts: OCP (*One Child Playouts*), ACP (*All Children Playouts*). Each of them has two subvariants, named "thrifty" and "prodigal", described later on. In both OCP and ACP, multiple independent trees are grown concurrently (for readability just two are shown in each illustration). Wavy arrows distinguished by different colors represent CUDA threads working for different stages of MCTS algorithm: orange for selection, green for expansion, black for playouts, and purple for backup. In MCTS-NC, threads are grouped in CUDA blocks that are indexed either by tree indexes alone, or tree-action pairs, depending on the stage and variant / subvariant. In the OCP variant, exactly one random child of each

[3] It guarantees: (a) small error probability when stopped prematurely, (b) convergence to the best action given sufficient time.

[4] Commonly met choices for C in (6) are: $1/\sqrt{2}$, 1, $\sqrt{2}$, 2.

expanded leaf node (across different trees) becomes played out. In ACP, all such children become played out. In the figure, terminal rewards from playouts are colored in: blue (losses of the first 'red' player), gray (draws) or red (wins of the first player). Their counts suitably update the statistics at ancestor nodes. For example, in the top left tree of the figure, the 15 wins by the red player that occurred within 128 playouts from the selected leaf, update the estimate of the middle action at the root level to become: $(37 + 15)/(256 + 128)$. For shortness, `Q` stands for an action-value estimate and `U` for its upper confidence bound.

2.5. Tighter bounds on action values due to CUDA parallelization (multiple playouts and trees)

In general, by introducing multiple playouts (within each iteration of MCTS) one aims at obtaining more accurate estimates of action values and tighter bounds on them. This guides the search process better and thereby improves the quality of play.

We are going to compare the bound (6) corresponding to a standard single-threaded MCTS execution (with 1 tree and 1 playout per step) against the bounds implied by multiple playouts and trees — for convenience, let us call that case: 'CUDA-based'. We shall write m to denote the number of concurrent playouts per one expanded child (e.g., $m = 128$), and T to denote the number of trees (e.g., $T = 8$). Having any node in mind, n will be understood as the number of times that node has been reached. In particular, for the root node this coincides with the number of steps (iterations) performed so far. For the purpose of a strict comparison, suppose for a certain action a that the decision to take it was made the *same* number of times – n_a – in both the standard and the CUDA-based execution. Hence, the total of playouts originated by a in the latter case is $N_a = n_a m$. We remark that this assumption is not obligatory. The bounds below are asymptotically correct also when the two counts are only approximately equal, $N_a \approx n_a m$, agreeing up to a constant factor, i.e. $N_a \sim \Theta(n_a m)$.

For multiple playouts the upper confidence bound is

$$q_a \leqslant \hat{q}_{a,N_a} + C'\sqrt{\frac{\log N}{N_a}} = \frac{W_a}{N_a} + C'\sqrt{\frac{\log N}{N_a}} = \frac{W_a}{n_a m} + C'\sqrt{\frac{\log n + \log m}{n_a m}}, \tag{7}$$

where W_a is the number of wins due to a experienced within N_a playouts, $N = \sum_a N_a = nm$, and C' is an appropriate constant. Division of second terms in (6) and (7) shows that the bound tightens proportionally to $C'/C\sqrt{(1 + \log m/\log n)/m}$, hence with an $O(\sqrt{\log m/m})$ factor. Note also that already for a fairly small number of visits n in the parent node of action a, s.t. $n \gtrsim m$, the factor becomes $O(1/\sqrt{m})$.

For the scenario with multiple trees, we consider only the actions at the root node level. Their statistics $W_{a,t}$, $N_{a,t}$ (number of wins due to a and times it was taken in tree t, respectively) can be aggregated by a CUDA sum-reduction and the following bound can be computed:

$$q_a \leqslant \frac{\sum_{t=1}^{T} W_{a,t}}{\sum_{t=1}^{T} N_{a,t}} + C''\sqrt{\frac{\log(NT)}{\sum_{t=1}^{T} N_{a,t}}} = \frac{\sum_{t=1}^{T} W_{a,t}}{T\,\bar{N}_{a,T}} + C'' \cdot \sqrt{\frac{\log(NT)}{T\,\bar{N}_{a,T}}} = \frac{\sum_{t=1}^{T} W_{a,t}}{T\,\bar{N}_{a,T}} + C''\sqrt{\frac{\log T + \log n + \log m}{T\,\bar{n}_{a,T}\,m}}, \tag{8}$$

where $\bar{N}_{a,T} = 1/T\sum_{t=1}^{T} N_{a,t}$. Now, by comparing (8) against (6), and assuming $n_a \approx \bar{n}_{a,T}$ for asymptotics, one sees the bound tightens with

$$\frac{C''}{C}\sqrt{\frac{1 + (\log T + \log m)/\log n}{T\,m}}, \tag{9}$$

hence with an $O\left(\sqrt{(\log T + \log m)/(Tm)}\right)$ factor, or just $O\left(1/\sqrt{Tm}\right)$ when $n \gtrsim Tm$.

In MCTS-NC software, bound (8) is valid for the OCP variant. As regards the ACP variant, one can see that if a node picked at the selection stage (of any tree) branches into b children then the root level action leading to that node results in $b \cdot m$ concurrent playouts, instead of only m (OCP case). Therefore, the asymptotic factor with which the analogical upper confidence bound tightens becomes $O\left(\sqrt{(\log T + \log b + \log m)/(Tbm)}\right)$, or just $O\left(1/\sqrt{Tbm}\right)$ when $n \gtrsim Tbm$.

2.6. MCTS-NC algorithms

Algorithms 1 and 2 present two (out of four) variants of Monte Carlo tree search available in MCTS-NC, named: OCP-THRIFTY and ACP-PRODIGAL, respectively.

Algorithm 1

```
algorithm OCP-THRIFTY(root_board, root_extra_info, root_turn)
  copy root state to device
    ⦚ dev_root_board = cuda.to_device(root_board)
    ⦚ dev_root_extra_info = cuda.to_device(root_extra_info)
  reset root nodes of trees to new root state
    ⦚ _reset[$T$, tpb_r](dev_root_board, dev_root_extra_info, root_turn, ...)
  while within computational budget do
    selections
      ⦚ _select[$T$, tpb_s](...)
    expansions (substage 1, thrifty indexing of actions)
      ⦚ _expand_1_ocp_thrifty[$T$, tpb_e1](...)
    copy actions expanded to host
      ⦚ dev_trees_actions_expanded.copy_to_host(ary=trees_actions_expanded)
    if first step of loop then
      memorize actions expanded at root node(s)
        ⦚ _memorize_root_actions_expanded[1, $B$ + 2](
        ⦚   dev_trees_actions_expanded, dev_root_actions_expanded)
    calculate total number of actions expanded over all
    trees: $A = \sum_{t=1}^{T} b_t$
      ⦚ $A$ = np.sum(trees_actions_expanded[:, -1])
    expansions (substage 2, thrifty number of blocks)
      ⦚ _expand_2_thrifty[$A$, tbp_e2](...)
    playouts ($m$ per tree)
      ⦚ _playout_ocp[$T$, $m$](...)
    backups
      ⦚ _backup_ocp[$T$, tpb_b2](...)
  copy actions expanded at root node(s) to host
  to know their count
    ⦚ dev_root_actions_expanded.copy_to_host(ary=root_actions_expanded)
    ⦚ n_root_actions = int(root_actions_expanded[-1])
  do sum-reduction over trees (thrifty number of blocks)
    ⦚ _reduce_over_trees_thrifty[n_root_actions, tpb_rot](...)
  do max/argmax-reduction over actions
    ⦚ _reduce_over_actions_thrifty[1, tpb_roa](n_root_actions, ...)
  copy best action to host (and reindex due to thriftiness)
    ⦚ best_action = dev_best_action.copy_to_host()[0]
    ⦚ best_action = root_actions_expanded[best_action]
  return best action
```

Input arguments `root_board`, `root_extra_info`[5] describe the root state from which the search starts, whereas `root_turn` $\in \{-1, 1\}$ indicates the player, minimizing or maximizing, to act first. Under every algorithmic step, we write down the selected details important at the CUDA code level — invocations of kernel functions, memory transfers. In compliance with the syntax of `numba.cuda`, we write the counts of blocks per grid and threads per block in square brackets following the names of invoked kernels Wherever applicable, we stick to mathematical notations defined so far, such as T, m, etc., instead of programming ones. Additionally, we introduce B to denote the maximum branching factor,[6] and $b_t \in \{0, \ldots, B\}$ to denote the number of legal actions

[5] meant for any additional information not implied by the contents of the board itself (e.g., possibilities of castling or en-passant captures in chess, the contract in double dummy bridge, etc.), or technical information useful to generate legal actions faster

[6] E.g., 361 for Go, 13 for double-dummy bridge, etc. One can assume that $B \leqslant 512$.

Algorithm 2

```
algorithm ACP-PRODIGAL(root_board, root_extra_info, root_turn)
  copy root state to device
    dev_root_board = cuda.to_device(root_board)
    dev_root_extra_info = cuda.to_device(root_extra_info)
  reset root nodes of trees to new root state
    _reset[$T$, tpb_r](dev_root_board, dev_root_extra_info, root_turn, ...)
  while within computational budget do
    selections
      _select[$T$, tpb_s](...)
    expansions (substage 1, prodigal indexing of actions)
      _expand_1_acp_prodigal[$T$, tpb_e1](...)
    if first step of loop then
      memorize actions expanded at root node(s)
        _memorize_root_actions_expanded[1, $B$ + 2](
          dev_trees_actions_expanded, dev_root_actions_expanded)
    expansions (substage 2, prodigal number of blocks)
      _expand_2_prodigal[($T$, $B$), tbp_e2](...)
    playouts ($m \cdot b_t$ per tree, prodigal number of blocks)
      _playout_acp_prodigal[($T$, $B$), $m$](...)
    backups (substage 1, sum-reduction due to ACP)
      _backup_1_acp_prodigal[$T$, tpb_b1](...)
    backups (substage 2)
      _backup_2_acp[$T$, tpb_b2](...)
  do sum-reduction over trees (prodigal number of blocks)
    _reduce_over_trees_prodigal[$B$, tpb_rot](...)
  do max/argmax-reduction over actions
    _reduce_over_actions_prodigal[1, tpb_roa](...)
  copy best action to host
    best_action = dev_best_action.copy_to_host()[0]
  return best action
```

expanded from the selected node of the t-th tree. Also, let us overload slightly the so far meaning of s, so that it represents the amount of memory needed to represent a state (proportional to its board shape and amount of extra information), rather than the state itself. The aforementioned counts of threads per block differ depending on the operation or stage to be performed. Listing 1 shows how those counts become established based on the relevant constants — B, s, T, and `tpb_default` typically equal to[7] 512.

```
tpb_max_actions = int(2**ceil(log2(B)))
tpb_r = min(2**ceil(np.log2(s)), tpb_default)
tpb_s = tpb_default
tpb_e1 = min(max(tpb_r, tpb_max_actions), tpb_default)
tpb_e2 = tpb_r
tpb_b1 = tpb_max_actions
tpb_b2 = tpb_default
tpb_rot = int(2**ceil(log2(T)))
tpb_roa = tpb_max_actions
```

Listing 1: Counts of threads per block for different operations or stages of MCTS-NC.

It should be explained that data structures underlying the algorithms are *arrays*, as is suitable for CUDA computations. In MCTS-NC, the overall representation of trees consists of 16 device-side arrays, storing, e.g.: parent–children links, sizes of trees, flags of terminal states and leaves, selection paths, statistics of playouts, etc. Random generators and information on root actions' values are stored in 11 additional device-side arrays.[8] Subsets of all the arrays constitute arguments for kernels' invocations (which we skipped for clarity in algorithms 1 and 2). For example, `dev_trees[,,]` is a three-dimensional array storing parent–children integer links for all trees: `dev_trees[`t`, `i`,0]` yields the parent index for the ith node of tth tree, while the remaining entries `dev_trees[`t`, `i`,1:]` store links to children nodes associated with particular actions.

The difference between "thrifty" and "prodigal" subvariants boils down to the manner in which CUDA blocks are allocated to some operations. In the "thrifty" approach, the number of blocks for expansions and ACP playouts is accurately tailored to the total of legal actions expanded in all trees: $A = \sum_{t=1}^{T} b_t$. For the sum-reduction over trees, it is tailored to the number of actions at the root node. This seems appropriate in terms of resource utilization but requires device-host memory transfers. In the "prodigal" approach the number of blocks is declared with an overhead implied by B. Note in particular the invocations: `_expand_2_prodigal[(`T`, `B`), tbp_e2](...)` `_playout_acp_prodigal[(`T`, `B`), `m`](...)`, where the tuple (T, B) imposes an excessive two-dimensional grid of blocks. This approach wastes some CUDA resources, but avoids device-host memory transfers (except for the final copy of the best action). It should be remarked though that the excessive blocks terminate their jobs very quickly (conditional returns) if they pertain to non-expanded actions, and can be soon rescheduled by GPU processors. At implementation level, the array named `dev_trees_actions_expanded[,]` is responsible for thrifty or prodigal actions' indexing. In either case, its last entry of each row holds the number of actions expanded by a given tree, i.e.: `dev_trees_actions_expanded[t,-1]=`b_t. At thrifty indexing, the b_t initial entries are filled with expanded actions ordinally from the left, for example: `dev_trees_actions_expanded[`t`, :]=[7,13,*, ..., *, 2]` when actions $\{7, 13\}$ are the only legal ones (`*`s are arbitrary entries from past uses). At prodigal indexing, we would store `dev_trees_actions_expanded[`t`, `a`]=`a for $a \in \{7, 13\}$, and `dev_trees_actions_expanded[`t`, `a`]=-1` for all $a \notin \{7, 13\}$.

Table 1 outlines all four algorithmic variants of MCTS-NC and kernel functions involved in them. There are 20 distinct kernels altogether, some of them reusable across variants. Under the kernels we write down their computational complexities (assuming sufficient GPU resources). Two new constants are d and D, denoting respectively the average depth that one "climbs down" to during selections, and the maximum possible depth (the latter relevant for playouts). We remind that due to UCT and the logarithmic regret principle, $d \sim \Theta(\log n)$ after n iterations of the main loop.

Listing 2 exemplifies the source code of `_select` kernel (for sources of other kernels we address the reader to the repository). In lines 17–30, all the threads compute UCB values simultaneously, in $O(1)$ time, and store them in the fast shared memory. Lines 32–41 perform the max/argmax-reduction in $O(\log B)$ time. One can note that only thread no. 0 is responsible for memorizing successive elements of the selection path in shared memory (lines 43–44), but later, outside the main `while` loop, all the threads cooperate to copy path elements from shared to global device memory (lines 46–51). For typical depths, not exceeding `tpb_s` (512 by default), every thread writes at most 1 path element, and the `for` loop performs exactly 1 iteration. For exceptionally long paths, that loop performs at most 4 iterations (note `MAX_TREE_DEPTH` constant), hence in any case this fragment is of constant-time complexity. Overall, the complexity of the whole `_select` kernel is $O(d \log B) = O(\log n \log B)$.

```
 1  @staticmethod
 2  @cuda.jit(void(float32, int32[:, :, :], boolean[:, :], int32[:, :], int32[:, :],
         int32[:], int32[:, :]))
 3  def _select(ucb_c, trees, trees_leaves, trees_ns, trees_ns_wins, trees_nodes_selected
         , trees_selected_paths):
 4      # 512 - assumed limit on max branching factor B, 2048 - MAX_TREE_DEPTH
 5      shared_ucbs = cuda.shared.array(512, dtype=float32)
 6      shared_best_child = cuda.shared.array(512, dtype=int32)
 7      shared_selected_path = cuda.shared.array(2048 + 2, dtype=int32)
 8      ti = cuda.blockIdx.x # tree index
 9      tpb_s = cuda.blockDim.x
10      thi = cuda.threadIdx.x
11      state_max_actions = int16(trees.shape[2] - 1)
12      node = int32(0)
13      depth = int16(0)
14      if thi == 0:
15          shared_selected_path[0] = int32(0) # path always starting from root
16      while not trees_leaves[ti, node]:
17          if thi < state_max_actions:
```

[7] set to 1/2 of `cuda.get_current_device().MAX_THREADS_PER_BLOCK` in order to preserve a safety margin of memory for kernels' registers

[8] For details see `init_device_side_arrays` in `mctsnc.py`.

```
            child = trees[ti, node, 1 + thi]
            shared_best_child[thi] = child
            if child == int32(-1):
                shared_ucbs[thi] = -float32(inf)
            else:
                child_n = trees_ns[ti, child]
                if child_n == int32(0):
                    shared_ucbs[thi] = float32(inf)
                else:
                    shared_ucbs[thi] = trees_ns_wins[ti, child] / float32(child_n)
                        + ucb_c * math.sqrt(math.log(trees_ns[ti, node]) / child_n)
        else:
            shared_ucbs[thi] = -float32(inf)
        cuda.syncthreads()
        stride = tpb_s >> 1 # half of tpb_s
        while stride > 0: # max-argmax reduction pattern
            if thi < stride:
                thi_stride = thi + stride
                if shared_ucbs[thi] < shared_ucbs[thi_stride]:
                    shared_ucbs[thi] = shared_ucbs[thi_stride]
                    shared_best_child[thi] = shared_best_child[thi_stride]
            cuda.syncthreads()
            stride >>= 1
        node = shared_best_child[0]
        depth += int16(1)
        if thi == 0:
            shared_selected_path[depth] = node
    path_length = depth + 1
    pept = (path_length + tpb_s - 1) // tpb_s # path elements per thread
    e = thi
    for _ in range(pept):
        if e < path_length:
            trees_selected_paths[ti, e] = shared_selected_path[e]
        e += tpb_s
    if thi == 0:
        trees_nodes_selected[ti] = node
        trees_selected_paths[ti, -1] = path_length
```

Listing 2: CUDA kernel for selections stage in MCTS-NC.

3. Software decription

3.1. Class MCTSNC

The software comes as 9 Python modules with the key class `MCTSNC` placed in `mctsnc.py` file. When constructing instances of `MCTSNC`, the user is allowed to impose: the traditional constraints on computational budget (time or steps), the parallelization-related settings (e.g., number of trees / playouts, amount of GPU memory), and to choose one of four algorithmic variants labeled by strings: `"ocp_thrifty"`, `"ocp_prodigal"`, `"acp_thrifty"`, `"acp_prodigal"`. For the complete list of constructor parameters see Appendix C.

Before running a search, an explicit call of the `init_device_side_arrays()` method is required. It allocates all the necessary device arrays based on relevant constants and available memory.[9] The actual search becomes executed by the method `run(root_board, root_extra_info, root_turn)`, returning the index of the best action. Additional information on other actions or time performance are accessible from dictionaries named `actions_info` and `performance_info` — properties of an `MCTSNC` instance.

Private functions of `MCTSNC` class are named with a single leading underscore (e.g.: `_set_cuda_constants`, `_make_performance_info`, `_playout_acp_prodigal`, etc.). Among them, the kernel functions are additionally described by `@cuda.jit` decorators coming from `numba` module. Exact specifications of types come along with the decorators.

3.2. Defining a custom game or search problem

MCTS-NC allows the user to define his custom game or search problem. This can be accomplished by implementing a set of five device functions, decorated by `@cuda.jit(device=True)`, named: `is_action_legal`, `take_action`, `compute_outcome`, `legal_actions_playout`, `take_action_playout`. The first three are self-explanatory. The last two are designed with faster playouts in mind and are meant to cooperate[10] — for some games, the evidence of legal moves / actions can be updated instead of being regenerated after each move. Examples of how those functions are implemented for the games of Connect 4 and Gomoku are provided in the file `mctsnc_game_mechanics.py`.

Since the CUDA model does not allow to use addresses (pointers) of device functions at the host side, the MCTS-NC software requires that the user provides his implementations either directly as bodies of the aforementioned functions, or writes his own device functions and forwards the calls.

[9] In particular, the maximum tree sizes are calculated from T, B and available memory. Leaves that cannot expand due to exhausted memory, do not cause program failures and simply execute playouts.

4. Illustrative examples

We have carried out experiments with MCTS-NC on two two-person games known as Connect 4 and Gomoku (illustrations in Fig. 2). The details of hardware and software of the experimental environment were as follows: Ubuntu Server 20.4.03, 4 CPUs: AMD EPYC 7H12 64-Core (2.6 GHz), 62.8 GB RAM, NVIDIA GRID A100-7-40C vGPU; nvcc 11.4 (V11.4.48), Python 3.8.10, numpy 1.23.5, numba 0.58.1.

In Connect 4, players drop their discs into columns of a 'standing' board, hence for its standard size of 6×7 the branching factor is 7 for most of the play. It decreases towards the end when some columns become filled. Gomoku is typically played on a 15×15 board (similar to a Go board) and players place stones anywhere on empty intersections. Hence, it has a high initial branching factor of 225, decreasing by 1 with each stone placed. By classical rules, the objective in Gomoku is to form a line of exactly five stones; six or more do not count for a win. Both games are characterized by a significant advantage of the starting player.

Zipped log files from all experiments discussed below are available in the repository in folder `experiments/`. Computations of each experiment can be reproduced *exactly*. It means that obtained estimates of action values and match outcomes will be identical. To do so one can set `REPRODUCE_EXPERIMENT = True` in the `main.py` script (assuming the default randomization seed equal 0). We remark that it is feasible in spite of the time limits per move that we imposed in our original executions.[12]

4.1. Experiments on Connect 4

In the first experiment, MCTS-NC algorithms (GPU) played Connect 4 games against the referential standard MCTS (CPU, single-thread), for simplicity called "vanilla" from now on. We tested all pairs of m, T settings from $\{32, 64, 128, 256\} \times \{1, 2, 4, 8\}$, and for each match-up conducted 100 games, switching the starting player every new game. Purposely, MCTS-NC instances have been handicapped with a 1s time limit per move, while the vanilla MCTS had 5 s. Fig. 3 shows color maps of average scores (reported as percentages) from this experiment, where a single win counted as 1, a draw as 1/2, and a loss as 0. The higher the average score, the more intense the red color. Smaller numbers in each cell provide additional details, reporting averages of: performed playouts / steps (main loop iterations), and reached mean depths / maximum depths. For reference, we inform that for the vanilla MCTS the corresponding averages were: 17.14 k / 17.14 k (1 playout per step) and 5.97 / 7.98, respectively. Fig. 5 graphs estimates of best actions' values and depths along a sample game from this experiment.

Several comments can be made on the results from Fig. 3. All the MCTS-NC algorithmic variants outplayed the vanilla MCTS, despite the time handicap. A general tendency can be noticed that average

[10] exchanging information via `legal_actions_with_count` array

[11] https://www.howardwexlertoys.com, https://pl.wikipedia.org/wiki/Gomoku

[12] The reproducing functionality reads a log and imposes the actual number of performed steps instead of the time limit. Hence, the duration becomes mimicked only approximately on a similar GPU device, the computations become reproduced exactly regardless of the device.

Table 1
CUDA kernels (and their computational complexities) in particular operations MCTS-NC algorithmic variants.

	OCP-THRIFTY	ACP-THRIFTY	OCP-PRODIGAL	ACP-PRODIGAL
reset	_reset[T, tpb_r](…) $O(s/\texttt{tpb\_r}) = O(1), \quad \texttt{tpb\_r} \gtrsim s$			
selections	_select[T, tpb_s](…) $O(d \log B + d/\texttt{tpb\_s}) = O(d \log B) = O(\log n \log B), \quad \texttt{tpb\_s} \gtrsim d$			
expansions 1	_expand_1_ocp_thrifty[T, tpb_e1](…) $O(s/\texttt{tpb\_e1} + B) = O(B), \quad \texttt{tpb\_e1} \sim s$	_expand_1_acp_thrifty[T, tpb_e1](…)	_expand_1_ocp_prodigal[T, tpb_e1](…)	_expand_1_acp_prodigal[T, tpb_e1](…)
memorization of root actions (first step only)	_memorize_root_actions_expanded[1, B + 2](…) $O(1)$			
expansions 2	_expand_2_thrifty[A, tpb_e2](…) $O(s/\texttt{tpb\_e2}) = O(1), \quad \texttt{tpb\_e2} \sim s$		_expand_2_prodigal[(T, B), tpb_e2](…)	
playouts	_playout_ocp[T, m](…) $O(s/m + s + D - d + \log m) = O(s + D - d + \log m)$		_playout_acp_thrifty[A, m](…)	_playout_acp_prodigal[(T, B), m](…)
backups 1	n.a.	_backup_1_acp_thrifty[T, tpb_b1](…) $O(\log B)$	n.a.	_backup_1_acp_prodigal[T, tpb_b1](…) $O(\log B)$
backups 2	_backup_ocp[T, tpb_b2](…) $O(d/\texttt{tbp\_b}) = O(1), \quad \texttt{tpb\_b} \gtrsim d$	_backup_acp[T, tpb_b2](…) $O(d/\texttt{tbp\_b2}) = O(1), \quad \texttt{tpb\_b2} \gtrsim d$	same as in OCP THRIFTY	same as in ACP THRIFTY
reduction over trees	_reduce_over_trees_thrifty[n_root_actions, tpb_rot](…) $O(\log T)$		_reduce_over_trees_prodigal[B, tpb_rot](…) $O(\log T)$	
reduction over actions	_reduce_over_actions_thrifty[1, tpb_roa](…) $O(\log B)$		_reduce_over_actions_prodigal[1, tpb_roa](…) $O(\log B)$	

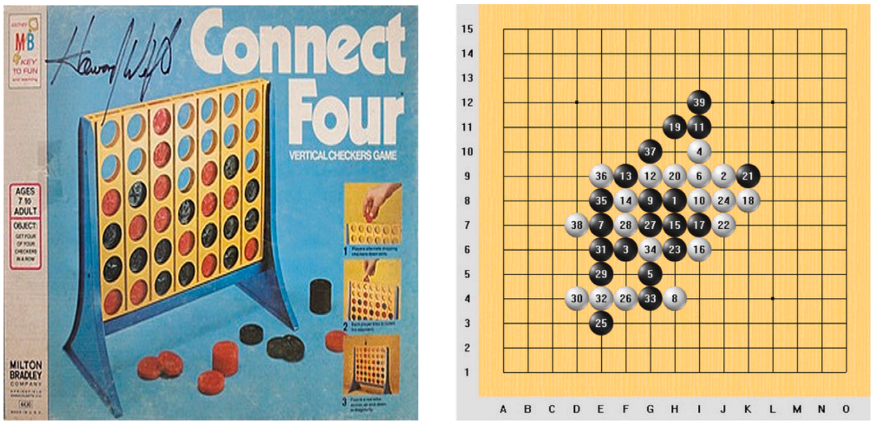

Fig. 2. Illustrations of Connect 4 and Gomoku.[11]

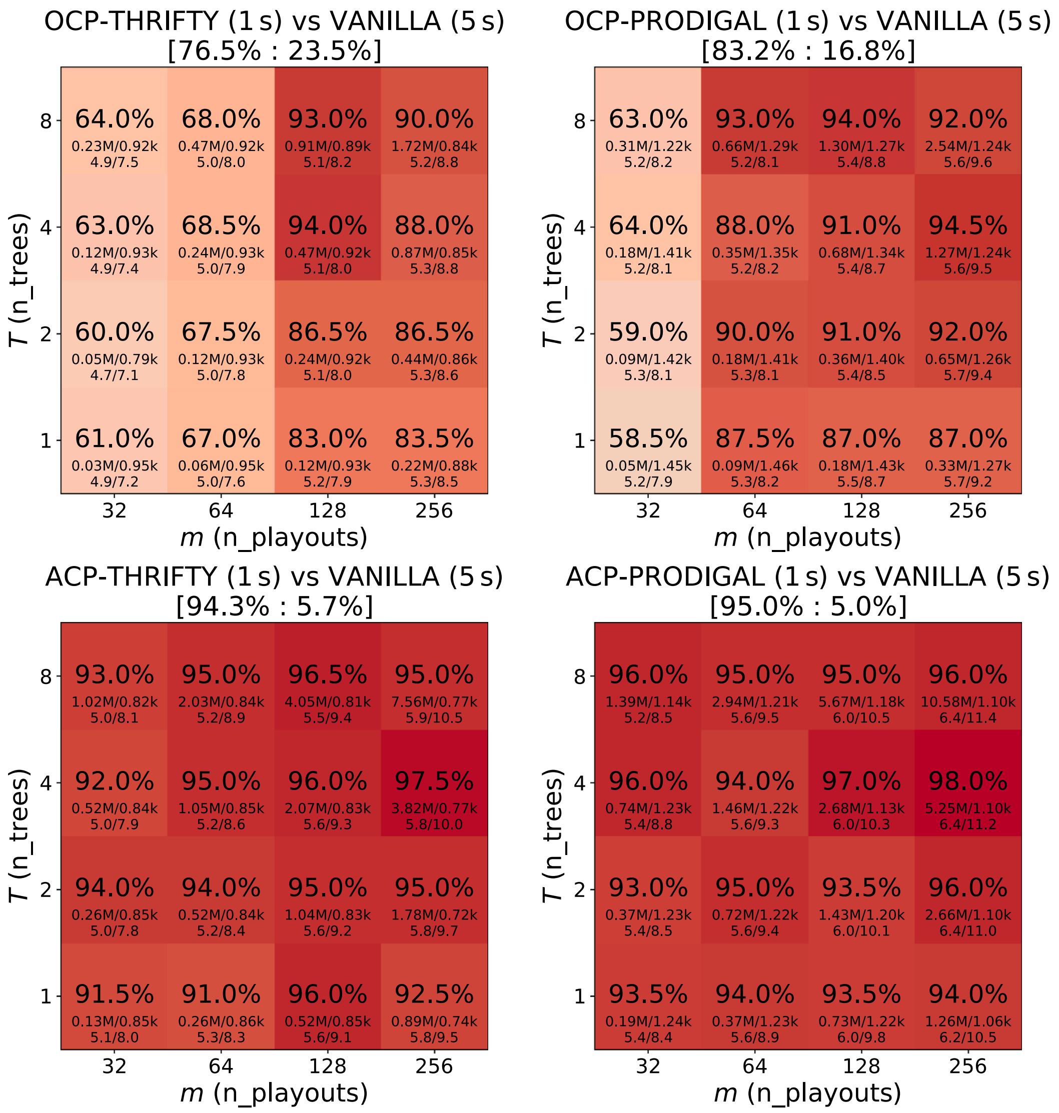

Fig. 3. Average scores over 100 games of Connect 4 (each cell) played by MCTS-NC algorithms (time limit: 1 s) against "vanilla" MCTS (time limit: 5 s). The intensity of red color correlates with the score. Smaller numbers report averages of: performed playouts/steps, and mean depths/maximum depths.

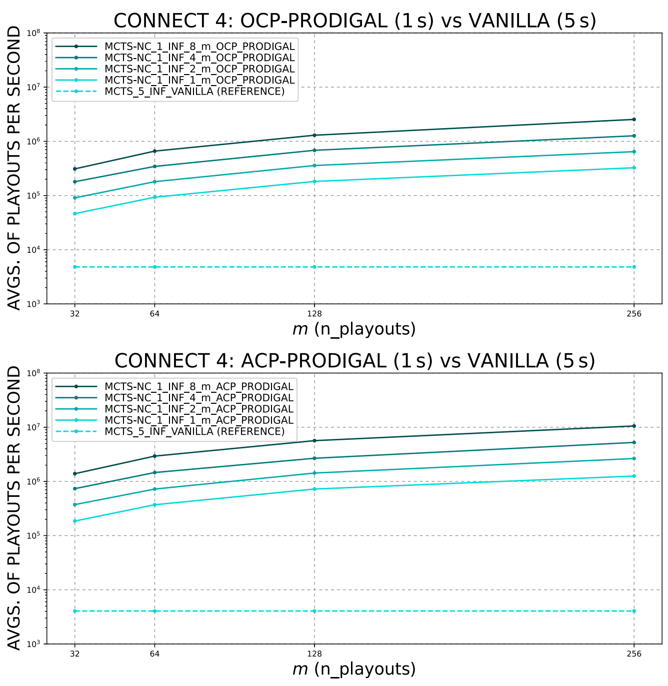

Fig. 4. Averages of playouts per second (plotted in logarithmic scale) over 100 games of Connect 4 played by prodigal MCTS-NC algorithms using 1, 2, 4 or 8 trees (time limit: 1 s) against "vanilla" MCTS (time limit: 5 s).

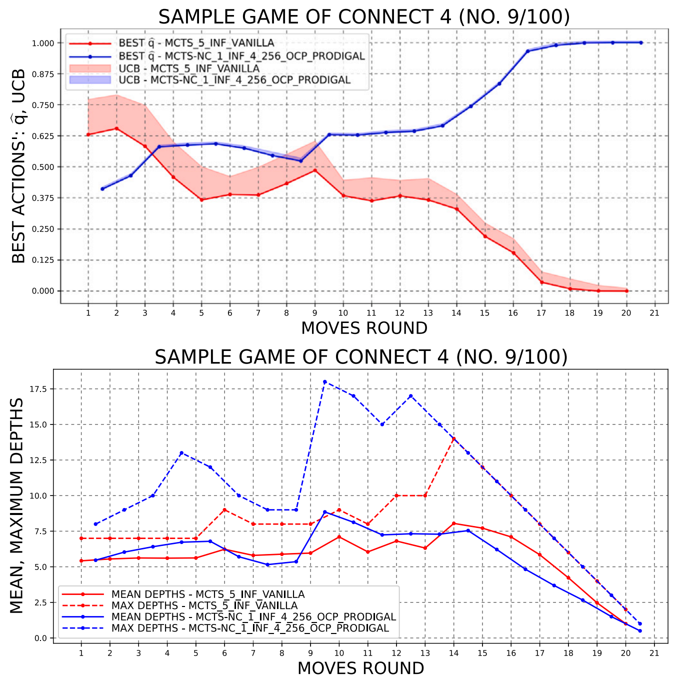

Fig. 5. Estimates on best actions' values and UCBs during a sample game of Connect 4 (top plot). Mean and maximum depths of tree nodes along the same game (bottom plot).

scores of MCTS-NCs improved as either of the parameters, m or T, grew, although some random fluctuations do exist caused by particular courses of games. Interestingly, the averages of reached mean and maximum depths also grew along with those parameters, even though the parameters translated into a higher workload for the GPU. This indicates that the algorithms became more selective about the actions within the searches. Counts of performed steps were decreasing only slightly (along with the parameters), which shows a high throughput of computations. For example, processing 8 trees on additional CUDA blocks rather than 1 tree reduced the number of steps on average by only $\approx 5\%$. Finally, it is worth noting that the "prodigal" variants were able to perform significantly more steps than the "thrifty" ones (by a factor of ≈ 1.46); and that the ACP variants achieved higher win rates than their OCP counterparts.

As regards the computational performance alone, one more important comment should be made. Forward model calls conducted within playouts are generally considered to be the most expensive part, computation-wise, of MCTS algorithms. In MCTS-NC algorithms, forward model calls are performed independently by each thread working for some `_playout_*` kernel function (see Table 1). This is done in a `while` loop until a decisive terminal outcome is reached. The body of that loop involves three calls to device functions: `legal_actions_playout`, `take_action_playout`, `compute_outcome` — which together can be seen as a single forward model call.

To show the scalability of our algorithms owed to CUDA resources, we report the averages of *playouts* conducted per second for different m and T settings. Note that the total of playouts is a "cousin" of the forward model calls total[13] and can be measured / counted naturally. This is owed to denominators of action values estimates that are registered in `dev_root_ns` array in our implementation. Fig. 4 graphs the *playouts per second* averages in logarithmic scale for OCP and ACP prodigal variants, averaged over 100 Connect 4 games. As expected, the observed quantity grows along with the m and T settings (hence, along with the invested CUDA resources), and can be greater by more than three orders of magnitude (for ACP) with respect to the referential baseline. On the other hand, since the shape of curves is sublinear in the logarithmic scale, one can realize that limits of the GPU throughput were in general reached during the conducted runs.

For the second experiment pertaining to Connect 4, we chose four instances of MCTS-NC (one instance per algorithmic subvariant) that achieved the highest win rates in the previous experiment, and played a tournament between them. For reference, we included the vanilla MCTS as the fifth player. Each match in the tournament consisted of 100 games with the time limit of 5 s per move for each side. Table 2 reports the results of this tournament. Below the total score for each pair of opponents, we report two additional subscores (separated by a comma), each obtained from 50 games, where the given player made the first and the second move, respectively.

As can be seen from the overall results, the ACP-PRODIGAL managed to beat all the opponents, achieving the win rate of $\approx 73.4\%$. Interestingly, the OCP-PRODIGAL had a higher average ($\approx 75.1\%$) despite losing a direct match to ACP-PRODIGAL. As regards the significance of the first move, the mean score for the starting player, averaged over all 1000 games of the tournament, was 55.45%.

4.2. *Experiment on Gomoku*

Our final experiment pertained to the game of Gomoku. Again, we conducted a tournament of AIs but due to Gomoku's very high branching factor we increased the time limit per move to 30 s. One should keep in mind that pure Monte Carlo searches do not apply any opening books nor pre-trained strategies (e.g., in the form of value or policy networks). Table 3 reports the results of this tournament.

Similarly as before, the ACP-PRODIGAL variant beat all the opponents, but this time the ACP-THRIFTY was the second best. The direct match between those two variants, consisting of 100 games, took almost 25 h of computations.[14] The experiment showed that concurrent samplings

[13] For example, for Connect 4 each playout may involve up to 42 forward model calls for the standard board.

[14] The whole experiment – all 10 matches – took almost 9 days.

```
  ABCDEFGHIJKLMNO
15+++++++++++++++15
14+++++++++++++++14
13+++++++++++++++13
12+++++++++++++++12
11++++++++●+○++++11
10+++++++++○+●+++10
 9++++○○●●●●○+○●+9
 8+++++●○○○●○○○○●8
 7+++++○●●○●○○●++7
 6+++○●●●●○●○+○++6
 5++++○+●●●○●●○●+5
 4++++○●+●●○●●○++4
 3++++●○○○○●++○++3
 2+++○+●●○●+○+●●+2
 1+++++++○+++++++1
  ABCDEFGHIJKLMNO
MCTSNC RUN...  [MCTSNC(search_time_limit=30.0, search_steps_limit=inf, n_trees=4, n_playouts=256, variant='acp_prodigal', device_memory=16.0, ucb_c=2.0, seed:  0)]
[actions info:
{
    0:  {'name':  'A1', 'n_root':  531590912, 'win_flag':  False, 'n':  309248, 'n_wins':  134121, 'q':  0.4337004604718543, 'ucb':  0.44982107084848894},
    1:  {'name':  'B1', 'n_root':  531590912, 'win_flag':  False, 'n':  309248, 'n_wins':  132358, 'q':  0.42799953435430466, 'ucb':  0.4441201447309393},
    2:  {'name':  'C1', 'n_root':  531590912, 'win_flag':  False, 'n':  309248, 'n_wins':  130400, 'q':  0.4216680463576159, 'ucb':  0.43778865673425055},
    ...,
    142:  {'name':  'H10', 'n_root':  531590912, 'win_flag':  False, 'n':  429235200, 'n_wins':  233188003, 'q':  0.543263933153665, 'ucb':  0.5436966335236032},
    ...,
    222:  {'name':  'M15', 'n_root':  531590912, 'win_flag':  False, 'n':  309248, 'n_wins':  132997, 'q':  0.4300658371274834, 'ucb':  0.44618644750411807},
    223:  {'name':  'N15', 'n_root':  531590912, 'win_flag':  False, 'n':  309248, 'n_wins':  130356, 'q':  0.42152576572847683, 'ucb':  0.4376463761051115},
    224:  {'name':  'O15', 'n_root':  531590912, 'win_flag':  False, 'n':  309248, 'n_wins':  130319, 'q':  0.42140612065397354, 'ucb':  0.4375267310306082},
    best:  {'index':  142, 'name':  'H10', 'n_root':  531590912, 'win_flag':  False, 'n':  429235200, 'n_wins':  233188003, 'q':  0.543263933153665, 'ucb':  0.5436966335236032}
}]
[performance info:
{
    steps:  3520,
    steps_per_second:  117.32477470948649,
    playouts:  531590912,
    playouts_per_second:  17718404.54204843,
    times_[ms]:  {'total':  30002.188444137573, 'loop':  30000.971794128418, 'reduce_over_trees':  0.15115737915039062, 'reduce_over_actions':  0.3490447998046875,
    'mean_loop':  8.523003350604665, 'mean_select':  0.10930977084419945, 'mean_expand':  0.2916635437445207, 'mean_playout':  7.889155975796959,
    'mean_backup':  0.22974115881052884},
    trees:  {'count':  4, 'mean_depth':  4.5354140118799595, 'max_depth':  8, 'mean_size':  519109.5, 'max_size':  522097}
}]
MCTSNC RUN DONE. [time:  30.002188444137573 s; best action:  142 (H10), best win_flag:  False, best n:  429235200, best n_wins:  233188003, best q:  0.543263933153665]
MOVE PLAYED: H10
```

Fig. 6. Sample printout of Gomoku moves analysis carried out by `MCTS-NC_30_INF_4_256_ACP_PRODIGAL`.

Table 2
Results of Connect 4 tournament played between the best variants of MCTS-NC (from Fig. 3) and vanilla MCTS for reference.

		A	B	C	D	E	avg. score	avgs. of: playouts / steps	avgs. of: mean depth / max depth
A	VANILLA (5 s)		04.0% 00.%, 08.%	04.5% 02.%, 07.%	02.0% 00.%, 04.%	01.0% 02.%, 00.%	02.875%	17.1 k / 17.1 k	5.97 / 7.98
B	OCP-THRIFTY (5 s, T=4, m=128)	96.0% 92.%, 100.%		16.0% 26.%, 06.%	35.0% 54.%, 16.%	28.5% 49.%, 08.%	43.875%	2.3 M / 4.5 k	6.63 / 11.85
C	OCP-PRODIGAL (5 s, T=4, m=256)	95.5% 93.%, 98.%	84.0% 94.%, 74.%		77.5% 77.%, 78.%	43.5% 33.%, 54.%	75.125%	6.4 M / 6.2 k	7.34 / 14.03
D	ACP-THRIFTY (5 s, T=4, m=256)	98.0% 96.%, 100.%	65.0% 84.%, 46.%	22.5% 22.%, 23.%		33.5% 57.%, 10.%	55.125%	15.7 M / 3.9 k	8.07 / 16.25
E	ACP-PRODIGAL (5 s, T=4, m=256)	99.0% 100.%, 98.%	71.5% 92.%, 51.%	56.5% 46.%, 67.%	66.5% 90.%, 43.%		73.375%	20.3 M / 5.4 k	8.62 / 17.54

Table 3
Results of Gomoku tournament played between the best variants of MCTS-NC (from Fig. 3) and vanilla MCTS for reference.

		A	B	C	D	E	avg. score	avgs. of: playouts / steps	avgs. of: mean depth / max depth
A	VANILLA (30 s)		00.0% 00.%, 00.%	00.0% 00.%, 00.%	00.0% 00.%, 00.%	00.0% 00.%, 00.%	00.00%	4.7 k / 4.7 k	2.95 / 3.00
B	OCP-THRIFTY (30 s, T=4, m=128)	100.0% 100.%, 100.%		51.0% 84.%, 18.%	26.0% 44.%, 08.%	28.0% 52.%, 04.%	51.25%	6.6 M / 13.0 k	3.04 / 3.50
C	OCP-PRODIGAL (30 s, T=4, m=256)	100.0% 100.%, 100.%	49.0% 82.%, 16.%		32.0% 62.%, 02.%	27.0% 48.%, 06.%	52.00%	14.2 M / 13.9 k	3.09 / 3.69
D	ACP-THRIFTY (30 s, T=4, m=256)	100.0% 100.%, 100.%	74.0% 92.%, 56.%	68.0% 98.%, 38.%		49.0% 86.%, 12.%	72.75%	411.8 M / 5.6 k	3.47 / 4.91
E	ACP-PRODIGAL (30 s, T=4, m=256)	100.0% 100.%, 100.%	72.0% 96.%, 48.%	73.0% 94.%, 52.%	51.0% 88.%, 14.%		74.00%	422.1 M / 5.6 k	3.49 / 4.93

over multiple children nodes are of great importance for Gomoku (and possibly other games with high branching factors). Even though OCP variants were able to compute about 13.5 k steps per move, the ACPs – computing only 5.6 k steps but coupled with over 400 M playouts –

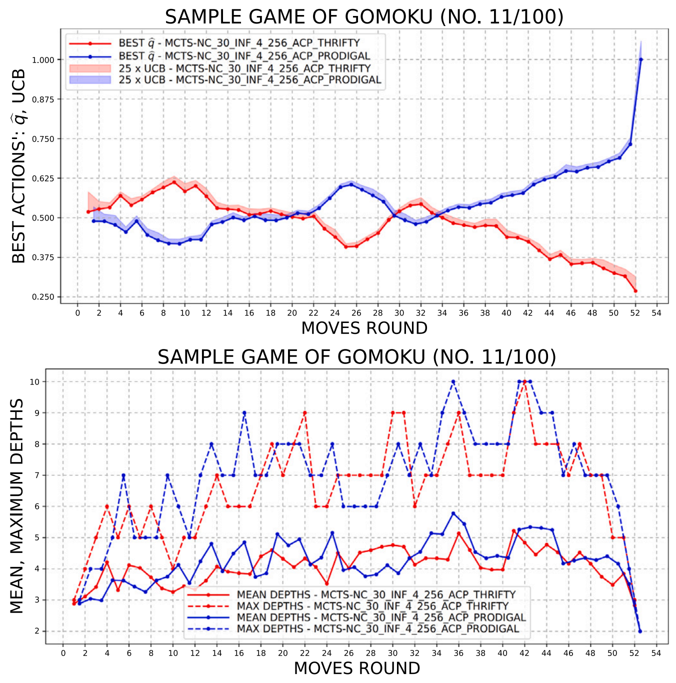

Fig. 7. Estimates on best actions' values and UCBs (enlarged 25×) during a sample game of Gomoku (top plot). Mean and maximum depths of tree nodes along the same game (bottom plot).

were clearly stronger than OCPs Gomoku-wise. As regards the significance of the first move, the mean score for the starting player was 66.3% (averaged over all 1000 tournament games), which indicates an even higher advantage comparing to Connect 4.

Fig. 7 graphs estimates of best actions' values and depths along a sample Gomoku game from this experiment. In the figure, the UCB intervals have been enlarged 25× for readability. Fig. 6 presents a sample console printout coming from that game.

5. Impact and conclusions

Multiple playouts and multiple trees applied in Monte Carlo Tree Search algorithm (MCTS) can obviously help in computing tighter bounds on action values [7,8]. With this purpose we have proposed and implemented four algorithmic variants of MCTS suitable for the CUDA computational model. The implementation, named MCTS-NC, integrates three parallelization types: leaf- / root- / tree-level. Even though the topic of those parallelizations has been an active research area for the last 12 years [11–16], the author of this paper has not come across a non-hybrid, GPU-*only* realization of MCTS that would include those three types fully integrated and also be lock-free (no mutexes).

An additional strong impact of the MCTS-NC software should be attributed to enhancements introduced directly within MCTS stages by means of CUDA-specific computational techniques. This aspect has been given little, or close to none attention in the literature. In MCTS-NC, those techniques include in particular:

- *sum-reductions* — employed in: playouts, ACP backups, and gathering statistics over multiple trees;
- *max / argmax-reductions* — employed in: selections (UCB values) and final choices of root-level actions;
- *threads cooperation* — employed in many places, in particular when: writing / reading information to / from shared memory, backing up outcomes along selection paths;
- *multiple random generators* (`xoroshiro128p`) — employed in: playouts and OCP expansions.
- *prodigal vs thrifty usage of CUDA blocks* — employed to minimize the amount of device-host memory transfers.

Please note that some of the employed techniques allow to carry out many partial operations in constant time or logarithmic complexity with respect to relevant constants. Detailed complexities of all 20 kernels have been reported in Table 1.

This paper and the preliminary experiments on Connect 4 and Gomoku raise new research questions, e.g.:

(1) *Given a fixed time budget, what is the proper balance between multiple playouts / samplings (m parameter) and the number of performed steps?* (the latter partially translates onto the reached depth),

(2) *Can a better quality of play be achieved by varying the number of playouts depending on the current depth?*

(3) *Thrifty or prodigal? OCP or ACP? Can the best variant be indicated without experimentations for a given game or search problem?*

Therefore, in our view the provided software has a significant potential to impact new research on games or sequential decision problems involving MCTS or reinforcement learning.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to express gratitude and give credit to the authors of Numba — an open-source JIT accelerating compiler for Python.

Appendix A. Hoeffding's inequality

For independent bounded random variables $X_1, \dots, X_n$, $\forall_i\ \alpha_i \leqslant X_i \leqslant \beta_i$, Hoeffding's inequality provides an upper bound on the probability that their sum $S_n = X_1 + \dots + X_n$ deviates from its expectation by more than a certain amount, say $\epsilon > 0$:

$$P\left(|S_n - \mathbb{E}S_n| \geqslant \epsilon\right) \leqslant 2e^{-2\epsilon^2 / \sum_{i=1}^{n}(\beta_i-\alpha_i)^2}. \tag{A.1}$$

When instead of a sum, a mean $\overline{X}_n = 1/n\,S_n$ is considered, and additionally when $\alpha_i = 0$, $\beta_i = 1$ for all i, the inequality transforms to the form $P\left(|\overline{X}_n - \mathbb{E}\overline{X}_n| \geqslant \epsilon\right) \leqslant 2e^{-2\epsilon^2 n}$.

Appendix B. Consistency conditions for UCT

The non-stationarity of the bandits problem that occurs within the MCTS algorithm requires that some mild assumptions need to be satisfied so that the UCT approach can be applied successfully. The needed assumptions – *drift conditions* – are:

1. for any fixed n_a expectations of observed averages $\overline{X}_{a,n_a}$ converge, i.e.: $q_{a,n_a} = \mathbb{E}(\overline{X}_{a,n_a})$ are well-defined;
2. the above expectations tend to true action values over time: $\lim_{n_a\to\infty} q_{a,n_a} = q_a$.

Results from [6] are existential — they do *not* provide a recipe how to choose C_0 in bound (5) (for a given game, its branching factor, etc.), but state that a good choice is always possible. Moreover, a property of such a choice is that there exists an integer n_0 such that for $n_a \geqslant n_0$ we have $c_{n_a,n_a} = 2C_0\sqrt{\log n_a/(n_a)} \geqslant 2|q_{a,n_a} - q_a|$. Therefore, the confidence interval embraces the true expectation q_a with high probability. This explains also the presence of factor 2 within the proposed form for c_{n,n_a}.

Appendix C. Constructor parameters

See Table C.4.

Table C.4
Constructor parameters for class `MCTSNC`.

parameter name	type (and default value if any)	description
`state_board_shape`	*tuple(int, int)*	shape of board for states in a given game, at most `(32, 32)`
`state_extra_info_memory`	*int*	number of bytes for extra information on states, at most `4096`
`state_max_actions`	*int*	maximum branching factor (corresponds to B), at most `512`
`search_time_limit`	*float* *default=5.0*	time limit in seconds (computational budget), `np.inf` if no limit
`search_steps_limit`	*float* *default=np.inf*	steps limit (computational budget), `np.inf` if no limit
`n_trees`	*int* *default=8*	number of independent trees (corresponds to T)
`n_playouts`	*int* *default=128*	number of independent playouts from an expanded child (corresponds to m), must be a power of two
`variant`	*{''ocp_thrifty'', ''ocp_prodigal'', ''acp_thrifty'', ''acp_prodigal''}* *default=''acp_prodigal''*	choice of algorithmic variant
`device_memory`	*float* *default=2.0*	GPU memory in gigabytes to be available for this instance
`ucb_c`	*float* *default=2.0*	C constants for upper confidence bounds on action values defined by (6)
`seed`	*int* *default=0*	randomization seed, used to initialize `xoroshiro128p` generators from `numba.cuda.random` module
`verbose_debug`	*bool* *default=False*	debug verbosity flag, if `True` then detailed information about each kernel invocation is printed to console (in each iteration)
`verbose_info`	*bool* *default=True*	verbosity flag, if `True` then standard information on actions and performance is printed to console (after a full run)
`action_index_to_name_function`	*callable* *default=None*	pointer to user-provided function converting action indexes to human -friendly names (e.g., ''`e2:e4`'' for chess)

References

[1] Silver D, et al. Mastering the game of Go with deep neural networks and tree search. Nat 2016;529:484–9.

[2] Silver D, et al. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. 2017, https://arxiv.org/abs/1712.01815.

[3] Portal F. An unexpectedly effective Monte Carlo technique for the RNA inverse folding problem. Cold Spring Harbor Laboratory; 2018, http://dx.doi.org/10.1101/345587, BioRxiv.

[4] Własnowski M, et al. cudaMMC: GPU-enhanced multiscale Monte Carlo chromatin 3D modelling. Bioinform 2023;39(10):btad588.

[5] Wijaya T, et al. Effective consumption scheduling for demandside management in the smart grid using non-uniform participation rate. In: Sustainable internet and ICT for sustainability. Springer-IEEE; 2013, p. 1–8.

[6] Kocsis L, Szepesvári C. Bandit based Monte-Carlo planning. In: Proceedings of the 17th European conference on machine learning. ECML '06, Berlin, Heidelberg: Springer-Verlag; 2006, p. 282–93.

[7] Świechowski M, et al. Monte Carlo Tree Search: a review of recent modifications and applications. Artif Intell Rev 2023;56:2497–562.

[8] Browne C, et al. A survey of Monte Carlo tree search methods. IEEE Trans Comput Intell AI Games 2012;4(1):1–43.

[9] Wen L, et al. TrafficMCTS: A closed-loop traffic flow generation framework with group-based Monte Carlo tree search. 2023, arXiv:2308.12797. https://arxiv.org/abs/2308.12797.

[10] Fawzi A, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. Nat 2022;610(7930):47–53.

[11] Zhou J. Parallel go on CUDA with Monte Carlo tree search. University of Cincinnati, School of Computing Sciences and Informatics; 2012.

[12] Barriga N, Stanescu M, Buro M. Parallel UCT search on GPUs. In: IEEE conference on computational intelligence and games. IEEE; 2014, p. 1–7.

[13] Mirsoleimani S, et al. Parallel Monte Carlo tree search from multi-core to many-core processors. In: 2015 IEEE trustcom/bigDataSE/ISPA. IEEE; 2015, p. 77–83.

[14] Mirsoleimani S, et al. A lock-free algorithm for parallel MCTS. In: Proceedings of the 10th international conference on agents and artificial intelligence. 2, SCITEPRESS; 2018, p. 589–98.

[15] Kurzer K, Hörtnagl C, Zöllner J. Parallelization of Monte Carlo tree search in continuous domains. 2020, arXiv:2003.13741. https://arxiv.org/abs/2003.13741.

[16] Buzer L, Cazenave T. GPU for Monte Carlo search. In: LION 17. Lecture notes in computer science, vol. 14286, Nice, France: Springer International Publishing; 2023, p. 179–93.

[17] Franciosini G, et al. GPU-accelerated Monte Carlo simulation of electron and photon interactions for radiotherapy applications. Phys Med Biol 2023;68(4):044001.

[18] Sun Y, et al. GPU acceleration of Monte Carlo tree search algorithm for Amazon chess and its evaluation function. In: 2022 international conference on artificial intelligence, information processing and cloud computing. 2022, p. 434–40.

[19] Zhang Y, et al. Development and application of a Monte Carlo tree search algorithm for simulating Da Vinci code game strategies. 2024, https://arxiv.org/html/2403.10720v1.

[20] Horcas J-M, et al. A Monte Carlo tree search conceptual framework for feature model analyses. J Syst Softw 2023;195(4):111551.

[21] Opolka F. Single-player Monte-Carlo tree search. 2021, https://github.com/FelixOpolka/Single-Player-MCTS.

[22] Sinclair P. MCTS 1.0.4. 2019, https://pypi.org/project/mcts/.

[23] Lai T, Robbins H. Asymptotically efficient adaptive allocation rules. Adv Appl Math 1985;6:4–22.

[24] Auer P, Cesa-Bianchi N, Fischer P. Finite time analysis of the multiarmed bandit problem. Mach Learn 2002;47(2–3):235–56.