GPU for Monte Carlo Search

Lilian Buzer[1] and Tristan Cazenave[2]

[1] LIGM, Universite Gustave Eiffel, CNRS, F-77454 Marne-la-Vallee, France
[2] LAMSADE, Universite Paris Dauphine - PSL, CNRS, Paris, France

Abstract. Monte Carlo Search algorithms give excellent results for some combinatorial optimization problems and for some games. They can be parallelized efficiently on high-end CPU servers. Nested Monte Carlo Search is an algorithm that parallelizes well. We take advantage of this property to obtain large speedups running it on low cost GPUs. The combinatorial optimization problem we use for the experiments is the Snake-in-the-Box. It is a graph theory problem for which Nested Monte Carlo Search previously improved lower bounds. It has applications in electrical engineering, coding theory, and computer network topologies. Using a low cost GPU, we obtain speedups as high as 420 compared to a single CPU.

Keywords: Monte Carlo Search · GPU · Playouts.

1 Introduction

1.1 History of Monte Carlo Search algorithms

Monte Carlo Tree Search (MCTS) has been successfully applied to many games and problems [4]. It was used to build superhuman game playing programs such as AlphaGo [32], AlphaZero [33] and Katago [35]. It has been recently used to discover new fast matrix multiplication algorithms [21].

Nested Monte Carlo Search (NMCS) [6] is a Monte Carlo Search algorithm that works well for puzzles. It biases its playouts using lower-level playouts. Kinny broke world records at the Snake-in-the-Box applying Nested Monte Carlo Search [22]. He used a heuristic to order moves in the playouts. The heuristic is to favor moves that lead to a state where there is only one possible move. Other applications of NMCS include Single Player General Game Playing [24], Cooperative Pathfinding [1], Software testing [28], Model-Checking [29], the Pancake problem [2], Games [11], Cryptography [14] and the RNA inverse folding problem [27].

Online learning of playout strategies combined with NMCS has given good results on optimization problems [30]. It has been further developed for puzzles and optimization with Nested Rollout Policy Adaptation (NRPA) [31]. NRPA has found new world records at Morpion Solitaire and crossword puzzles. Edelkamp, Cazenave and co-workers have applied the NRPA algorithm to multiple problems. They have adapted the algorithm for the Traveling Salesman with Time

Windows (TSPTW) problem [12, 16]. Other applications deal with 3D Packing with Object Orientation [18], the physical traveling salesman problem [19], the Multiple Sequence Alignment problem [20], Logistics [17, 9], the Snake-in-the-Box [15], Graph Coloring [10], Molecule Design [8] and Network Traffic Engineering [13]. The principle of NRPA is to adapt the playout policy to learn the best sequence of moves found so far at each level. GNRPA [7] has much improved the result of NRPA for RNA Design [8].

1.2 Generating playouts

Monte Carlo based algorithms for game AI generate a large number of simulated games called *playouts*. The generation of a playout is presented in Algorithm 1. The main loop of this algorithm is used to play the different moves until the end of the game. At the beginning of each iteration, the game data is analyzed to detect all possible moves. Then, a given policy or a random strategy selects a move among the different possibilities. After that, the chosen move is played and we iterate. When the game ends, the list of played moves is returned with the score. The game AI will then analyze the results of many playouts to generate a new bunch of playouts with better scores.

Algorithm 1: Generation of a playout

```
  Data: Game: game object embedding rules and data
  Data: Policy: function that selects a move
1 Function CreatePlayout(Game, Policy):
2     Playout = [ ] // History of played moves
3     while $Game.IsNotTerminated()$ do
4         $L = Game.GetPossibleMoves()$
5         $ChosenMove = Policy.ChoseMove(L)$
6         $Playout.append(ChosenMove)$
7         $Game.Play(ChosenMove)$
8     return Playout, $Game.Score()$
```

1.3 Why GPUs have not been considered?

Monte Carlo based approaches for game AI are generally performed on high-end CPUs. Even if these algorithms have demonstrated their performance and quality, the possibility of creating a version on GPU has not been studied. There are many reasons for this:

- The capabilities of a CPU core, in terms of computing power or clock speed, greatly exceed the capabilities of a GPU core.

- Most Monte Carlo based algorithms are written using an iterative approach. GPU processing is based on parallelism and switching from one programming style to the other is difficult.
- On average, a GPU core has only 1 KB of memory cache. It seems difficult to store all the game data inside. Moreover, GPU global memory has a notoriously slow latency.
- Parallel GPU threads must access contiguous data to achieve efficiency. However, depending on the game, it is not always easy to meet this requirement.

These accumulated difficulties do not bode well for the performance. Nevertheless, this paper aims to show that implementing playout simulations on GPU is possible with little difficulty and with performance gain.

1.4 Our contribution

In this paper, we carefully present the first, to our knowledge, proof of concept showing that it is possible to obtain significant performance for playouts generation on a GPU. We study with precision the performance losses induced by the GPU architecture relative to parallel simulations in Section 2, to computing power in Section 3 and to memory access in Section 4. We finally choose the game 'Snake in the box' to perform a series of benchmarks in Section 5. Most of the tests we use are classical tests, however, we use them in the very particular context of parallel simulations doing random memory accesses. The objective of this article is to determine precisely how a GPU behaves in such a specific situation.

2 Parallel execution

2.1 Warp

We briefly summarize NVIDIA GPUs architecture, the reader can find a more detailed description in [3, 26]. A NVIDIA GPU contains thousands of CUDA cores gathered in groups of 32 cores called *warps*. In a warp, all the cores process the same statement at the same time. Thus, the cores belonging to the same warp run the same source code in parallel, only their register values may differ.

When looking at Algorithm 1 used for playouts generation, three steps are required: list the possibles moves, choose one of them and play it. Thus, if a particular game performs the same sequence of instructions to carry out these steps, we can efficiently simulate 32 playouts in parallel in the same warp. But these 32 simulations will obviously differ in their number of rounds. So, when a playout ends before the others, one of the cores becomes idle. The inactive cores are accumulating until the longest playout ends. At this point, all cores wake up and start a new batch of 32 simulations. As opposed to a sequential processing where a new simulation starts as soon as the previous one has finished, some computing resource is wasted by the idle cores waiting for the last simulation to complete. In the following, we theoretically estimate the performance loss inherent to any playouts simulation performed in a warp to verify that this loss is acceptable.

2.2 Theoretical model

Let us assume that for a given policy, the time spent to simulate a playout can be modeled by a Gaussian distribution. So, the 32 simulations running in a warp can be modeled as different Gaussian random variables denoted $X_i = \mathcal{N}(\mu, \sigma)$ with $i = 1, \dots, 32$. Now, we want to model the time spent by a warp to complete the generation of a group of 32 simulations. For this, we define a new variable:

$$Z = [\max_i X_i]$$

To simplify the calculation, we use $X'_i = X_i - \mu$ and $Z' = [\max_i X'_i]$. By Jensen's inequality, we obtain:

$$e^{t\mathbb{E}[Z']} \leq \mathbb{E}[e^{tZ'}] = \mathbb{E}[\max_i e^{tX'_i}] = \sum_{i=1}^{n} \mathbb{E}[e^{tX'_i}] = ne^{t^2\sigma^2/2}$$

Thus, we can write:

$$\mathbb{E}[Z'] \leq \frac{\log(n)}{t} + \frac{t\sigma^2}{2}$$

With $t = \sqrt{2\log n}/\sigma$ and with $n = 32$, we finally obtain:

$$\mathbb{E}[Z] \leq \mu + \sigma\sqrt{7}$$

In comparison, the average time spent by the 32 simulations follows a Gaussian distribution equal to $Y = (\sum_i Xi)/32 = \mathcal{N}(\mu, \sigma/32)$. Using the empirical rule, we know that 99.7% of the time, the variable Y will satisfy: $Y \geq \mu - 3\sigma/32$. Thus, we can conclude that 99.7% of the time, the ratio Z/Y satisfies :

$$\frac{Z}{Y} = \frac{\max(Xi)}{\text{mean}(X_i)} \leq \frac{\mu + \sigma\sqrt{7}}{\mu - 3\sigma/32} = \frac{\mu + 2.64\sigma}{\mu - 0.09\sigma}$$

The ratio Z/Y bounds the α factor describing the time increase when simulations run on a warp in parallel. As an example, when the mean of the playouts length is equal to 100 moves with a standard deviation of 15, this bound is equal to 1.41. This bound is not tight and may be somewhat overestimated. Thus, we numerically estimate the α factor in the following Section.

2.3 Numerical estimation

We now try to numerically estimate the α factor corresponding to the ratio between the time spent by a warp to generate 32 playouts and the average time used by the same 32 playouts without considering the idle time. For this, let us assume that the length of each simulation can be modeled as Gaussian variable

$X_i = \mathcal{N}(\mu, \sigma)$. In this manner, the α factor is equal to $\mathbb{E}[\max(Xi)/\text{mean}(X_i)]$. Note that the α factor is invariant by scaling the X_i variables. Thus, we present some estimations of the α factor for different Gaussian random variables $X_i = \mathcal{N}(1, \sigma/\mu)$ in Table 1. When the ratio σ/μ increases, the α factor also increases. The known bound from the previous Section for the case $\sigma/\mu = 0.15$ is equal to 1.41, but the numerical estimation is more favorable with an α factor equal to 1.31. This ratio can be considered as an acceptable overhead of 31%.

Table 1. Estimations of the α factor for Gaussian random variables $X_i = \mathcal{N}(\mu, \sigma)$

σ/μ	0.05	0.1	0.15	0.20	0.25	0.30	0.35	0.40
α	1.1	1.2	1.31	1.41	1.51	1.62	1.72	1.82

3 Expected performance

We compare the performance of a NVIDA GTX 3080 GPU and an AMD Ryzen 9 3900X CPU. Comparisons between CPU and GPU performance can be found in the literature, but usually in a specific context such as linear algebra [23] or neural networks [5]. To our knowledge, no study has been published relative to our specific context of random memory access. Thus, we first benchmark the computing power of these two devices and then, we evaluate their memory latency. All this information allows us to determine what performance gain we may expect, for what problem size and in what way.

3.1 Computing power

We want to compare the computing power of a GPU core against a CPU core. For this purpose, we set up a function that computes the sums of all possible subsets of $\{k \in \mathbb{N} : k \leq n\}$ without performing any memory access. In this way, we test the performance of two common operations in puzzle games: additions and logical tests. Our two test platforms are a NVIDIA GTX 3080 GPU and an AMD Ryzen 9 3900X CPU. In our test scenario, each CPU or GPU thread performs the same calculations. We perform different tests with 1 or 2 threads per core and with integer or float values. We present the time spent to complete all the threads in Table 2. We choose as reference time the scenario with 1 thread per CPU core and with integer numbers. We notice that using 2 threads per core instead of one seems to be more efficient for both GPUs and CPUs. Finally, we can conclude that GPU execution is about 4 to 5 times longer compared to CPU.

3.2 CPU and GPU memory cache size

A memory cache is used to reduce the average time to access data in memory. The logic behind memory cache is simple: when a core requests data in RAM,

Table 2. Relative duration of the performance benchmark.

	1 Thread / Core		2 Threads / Core	
	Integer	Float	Integer	Float
CPU vs GPU	1 vs 3.9	1.3 vs 5.6	1.3 vs 6.3	2.0 vs 10.1

it first checks whether a copy exists in the L1 cache and in case of success, this process saves memory access time. For the AMD Ryzen 9, the AMD EPYC and the Intel Xeon Platinum family, the L1 cache size is 64KB per core. For the NVIDIA GTX family, the L1 data cache size is 128KB but it is shared among 128 CUDA cores. Thus, on average each CUDA core has 1 KB of L1 data cache which is far less than a CPU L1 cache of 64KB. So if we want a GPU to be able to compete with a CPU, we should process problems with small data size.

3.3 Estimating memory latency

After having compared CPU and GPU cache size, we now focus on their response time also called latency. For this, we use the P-Chase method presented in [25, 34] which continuously performs the read statement $i = A[i]$ as shown in Algorithm 2. To simulate random memory accesses, we initialize the values in A to perform a random walk of this array as in the example $A[\,] = \{6, 5, 7, 2, 0, 4, 3, 1\}$. We set up a second test scenario to analyze the latency of Read+Write operations. For this, we still conduct a random walk, but this time each memory read is followed by a memory write at the same location. This behavior simulates a game which is updating the data of its gameboard.

Algorithm 2: Random memory read latency estimation

```
Data: $A$: array of Integer, $m$: number of reads to perform, $p$: random start
1 Function P-ChaseReadOnly($A, m, p$):
2     for $m/3$ do
3         $p = A[p]$;    $p = A[p]$;    $p = A[p]$;              // 3 reads
4 Function P-ChaseReadWrite($A, m, p$):
5     for $m/3$ do
6         $p2 = A[p]$;    $p3 = A[p2]$;    $p4 = A[p3]$;         // 3 reads
7         $A[p] = p3$;    $A[p2] = p4$;    $A[p3] = p2$;         // +3 writes
8         $p = p4$
```

3.4 Random access and CPU L1 cache latency

The L1 memory cache on modern CPUs is very efficient. Thus, playout generation can take full advantage of the acceleration provided by the L1 cache. We present the average latency estimated using the P-Chase method in Table 3. We consider different scenarios with 1 or 2 threads per core and with read only or read and write. The estimation we obtain are very stable as long as data resides entirely in the L1 cache. We notice that with 1 or 2 threads per core, performing 1 read or 1 read + 1 write access, the memory latency is very similar. This confirms that Ryzen 9 CPU family is able to handle read and write in parallel, with 2 threads per core and with random access without loss of performance.

Table 3. L1 cache CPU latency for random access.

Threads per core	1	2
Latency in ns - Read Only	1.46	1.53
Latency in ns - Read+Write	1.54	1.60

3.5 Random access and GPU latency

The NVIDIA GTX 3080, has 68 Streaming Multiprocessors (SM). Each of these SMs has an internal memory of 128KB that can be partitioned into L1 cache and shared memory. The SM L1 cache behaves like a CPU L1 cache. Shared memory can be seen as a user-managed memory space that all threads of the same SM have access to. Its size is limited to 100KB on the 3080 GPU. We know that our parallel playouts simulations will generate mainly random access in memory. In our benchmark, each thread performs its own P-Chase using its own array. In this manner, each thread behaves as if it was performing its own game simulation in a private memory space. So we use the P-Chase algorithm to precisely estimate the memory latency in such a scenario, this information being not documented by NVIDIA. Thus, we have two test scenarios: one where data mainly resides in the L1 cache and another one where data are allocated in shared memory. In the first scenario, we can exceed the size of the L1 cache and use global memory. So, we test arrays up to a size of 2K which requires 16-bit indexing. In the second scenario, data must totally reside in the shared memory space, so we limit arrays to 256 bytes in order to use only 8-bits indexing. We also test the two variants of the P-Chase algorithm: read only or read+write. We present all estimated latency in Table 4.

What are our observations ? When using the L1 cache, latency of read only access is stable until the L1 occupancy remains below 100%. When occupancy is beyond 100%, latency increases rapidly to over 1000ns. When running many threads on the same core, a mechanism called latency hiding is triggered by the GPUs to improve performance. This way, when the active thread is put on hold due to a memory access, a waiting thread can rapidly take its place avoiding a core being idle. Thus with 2 threads per core, we see that the latency reduces by half. Nevertheless, using 2 threads per core divides the memory available for each thread by a factor 2 which increases the strain on the available memory space for each thread. When we perform the Read+Write test, we notice that the performance becomes very bad with a latency nearly 10 times longer. It is not easy to explain this behavior but in any case it seriously harms playouts simulation. When data relies in shared memory, the latency in the read only scenario is better and stable with about 18 ns. But most importantly, the latency during Read + Write tests remained very good with 25ns.

Table 4. GPU latency in nanoseconds for random memory access.

L1 cache + RAM - 16 bits value - X means > 1000									
Buffer size	8	16	32	64	128	256	512	1K	2K
Occupancy	2%	4%	8%	16%	33%	66%	131%	262%	524%
1T/Core Read	26	27	28	30	31	32	524	895	X
1T/Core R+W	79	144	258	296	287	294	X	X	X
Occupancy	4%	8%	16%	33%	66%	131%	262%	524%	1048%
2T/Core Read	13	13	14	17	21	310	868	X	X
2T/Core R+W	46	144	248	297	309	X	X	X	X
SHARED - 8 bits value - X means > 1000									
Buffer size	8	16	32	64	128	256			
Occupancy	1%	2%	4%	8%	16%	33%			
1T/Core Read	17	18	18	19	19	19			
1T/Core R+W	23	24	24	25	25	25			

3.6 Synthesis

We can conclude that the use of shared memory is a wise choice because it provides an optimal latency for random memory accesses, even when reads and writes are performed at the same time. Using shared memory, we must respect the constraint of 100KB maximum for 128 cores. This will force us to greatly reduce the storage of game data in memory. When looking for performance, we should focus on problems with less than 1 KB of data per simulation.

In terms of computing power, we can conclude that a CPU core is five times faster than a GPU core. When data resides in shared memory, GPU latency (25ns) is 16 times slower compared to CPU latency (1.5ns). Thus memory latency, even when using shared memory, remains the main bottleneck when we speak about performance. We recall that the NVIDIA GTX 3080 has 8704 CUDA cores, thus, when a game performs mainly memory accesses, its computing power will be equivalent to 8704/16/1.3 = 420 times a single CPU core, the ratio 1.3 being the Warp performance loss factor we present in Chapter 2. This estimation is an approximation, but it gives the level of performance we can expect.

4 Snake in the box

4.1 Performance Benchmark

We have chosen the game 'Snake in the box' game for several reasons:

- The game rules are intuitive and quickly understandable.
- The source code is easily readable and can be used as a pedagogical example.
- This game generates mainly memory access and finally very few computations in comparison. So, this game allows us to test our scenario where memory latency is the main performance bottleneck.

4.2 Game rules

A d dimensional hypercube is an analogue of a cube in dimension d with 2^d nodes, each node having d neighbors. The Snake in the box problem consists in searching a longer path among the edges of a hypercube. There are two additional constraints : we cannot turn back and we can not select a new node which is adjacent to a previously visited node (*connectivity constraint*). The score of a playout corresponds to its number of edges in the path.

4.3 Data Structure

We can code each node of a d-dimensional hypercube by an integer value of d bits. The code of two connected corners only differ by one bit. This way, the neighbors of the node 0010b are 1010b, 0110b, 0000b, and 0011b. We associate with each node a 1 bit value named $Usable[i]$ indicating whether that node can be visited. As GPU programming requires optimization of data in the L1 cache/shared memory, we use a bitfield of 2^d bits to store the array $Usable$. In the same way, as there are at most d possible moves at each turn, we can store the sequence of moves using only 4 bits per move when $d < 16$. As the NVIDIA GTX 3080 has a maximum of 100KB of shared memory, we can test our approach for a value of d ranging from 8 to 11 as shown in Table 5.

Table 5. L1 cache occupancy relative to the dimension.

Dimension of the Snake in the box	8	9	10	11	12
Number of nodes	256	512	1024	2048	4096
Longuest known path	98	190	370	707	1302
Bitfield size in bytes	32	64	128	256	512
Sequence size in bytes	49	95	175	354	651
Data size in KB - 128 playouts	10	20	39	76	145
Shared memory occupancy - 100KB	10%	20%	40%	78%	149%

5 Nested Monte-Carlo Search

NRPA algorithm uses a lot of memory and it does not suit our constraints. Thus we focus on the NMCS algorithm which is memory efficient and provides very good results for the Snake In the Box problem.

5.1 Algorithm

A Nested Monte-Carlo Search, $NMCS$, returns a sequence of moves used to finish a game. The $NMCS$ algorithm takes two arguments: an integer indicating its recursion level and a game G where n moves have already been played. To complete the game, the NMCS algorithm iteratively plays moves. To choose the next move, the algorithm analyzes all the possible moves. For each move, it creates a copy G' of the current game G, plays the candidate move and performs a recursive call to $NMCS(level-1, G')$. If the sequence returned by the recursive call is associated with a better score, the best known sequence is replaced. After all possible moves have been tested, the algorithm plays the *n+1*-th moves of the current best known sequence and iterates. We point out that at level 0, the $NMCS$ performs only a random playout to build a sequence of moves.

Algorithm 3: NMCS algorithm

```
Function NMCS(level, Game G):
    Input: G game in progress (partially started or nearly finished)
    Output: B game completed
    B = G.copy()                                   // Current best game
    if level == 0 then
        B.playout()                 // At level 0, NMCS performs a playout
    else
        n = G.Sequence.size               // n turns have been performed
        while not G.Terminated() do
            for move in G.GetPossiblesMoves() do
                G' = G.copy()                        // Create a subgame
                G'.play(move)                          // Test this move
                NMCS(level-1,G')                // Evaluation from level-1
                if G'.score() > B.score() then
                    B = G'
            G.play(B.sequence[n])     // n-th move of best known sequence
            n += 1
    return B
```

5.2 NMCS with Parallel Leaf

To run multiple NMCS algorithms in parallel, we are faced with several difficulties. First, a level-4 NMCS algorithm requires storing 5 games in memory which means the L1 memory will quickly saturate. Second, the NMCS algorithm has been designed as an incremental and also recursive algorithm which makes it almost impossible to migrate to a parallel version. However, we can set up a *parallel leaf* version. For this, instead of building only one playout at level 0, we generate 32 playouts in parallel and select the best one. Other levels of the NMCS algorithm remains in single thread mode. The NMCS with Parallel Leaf remains effective because most of the computation time is spent at level 0.

We recall that the NVIDA GTX 3080 contains 68 Streaming Multiprocessors containing 4 warps of 32 cores. We can run one parallel leaf NMCS per warp to obtain $68 \times 4 = 272$ NMCS running in parallel on this GPU. Each thread in a warp generates a playout. Then when the 32 playouts are over, a single thread, named the *the master thread*, analyzes their results and select the best sequence to be returned to the upper level. As specified in the NVIDIA specification, threads within a warp that wish to communicate via memory must execute the dedicated CUDA function *__syncwarp__*(). In our case, this function has to be called by the master thread to correctly analyze the playouts.

5.3 NMCS on GPU

While it may seem easy to set up 32 threads running in parallel, there remains a little challenge to address when programming the NMCS algorithm on GPU. In fact, inside a warp, a GPU can easily reduces the number of running threads due to an if statement. But, for the NMCS algorithm, we operate in reverse. Indeed, the higher levels of the NMCS algorithm use only one master thread, and after some recursive calls, playouts generation requests the use of 32 threads in parallel. It is not the usual way a GPU works.

For this, we use a specific trick: when a processing must be performed by the master thread of the algorithm, we precede it with a *filter test* that verifies that the current thread corresponds to the master thread. But, we must keep the 32 threads active until the level 0 of the NMCS algorithm. For this, all threads in the warp must execute recursive calls and their enclosing loops. Threads outside the master thread should do nothing. As the filter test prevents them from performing any processing, they remain active and follow the master thread without performing any processing until level 0.

5.4 Performance comparison

We compare the performance of a NVDIA GTX 3080 GPU relative to one core of a Ryzen 9 3900X CPU. We validate our GPU implementation by comparing the mean score obtained by the CPU and the GPU versions. Any important deviation is associated to an implementation problem. We set up our GPU version using shared memory in order to obtain better performance.

We show performance gains in Table 6 for the Snake In The Box problem in dimension 8, 9 and 10 using level 1 and 2 of the NMCS algorithm. For level 1, the GPU was able to achieve performance gains by a factor $\times 390$ which is of the same magnitude as the ratio $\times 420$ we estimate in Section 3. In a surprising way, we notice that in level 2 of the NMCS algorithm, performance increases reaching $\times 480$. This behavior remains unexplained because the time spent by the higher level is normally negligible. We need to conduct more sophisticated experiments to analyze this phenomenon.

Table 6. Performance gain for the 3080 GPU relative to one CPU core.

Dimension	8	9	10
$\widetilde{\text{N}}$MCS Level 1	$\times 380$	$\times 387$	$\times 382$
$\widetilde{\text{N}}$MCS Level 2	$\times 471$	$\times 498$	$\times 521$

5.5 Implementation

To set up our GPU implementation, on the first try, we choose not to optimize data structures for GPU architecture. We thought this task not very useful because the performance bottleneck mainly comes from random memory access. So we import our CPU/C++ source code into our CUDA program. We embed game data and game functions into a C++ class called *SnakeInTheBox* to improve the structure of the code and its readability. We also use a C++ structure called *Info* to gather input and output information of each thread. This first version reaches interesting performance but half the efficiency we show in Table 6.

In a second version, we update our code to use shared memory. We also create an implementation of the list of possible moves specific to GPU. This implementation uses *memory coalescing*, a technique where parallel threads accessing consecutive memory locations combine their requests into only one memory request. Considering all these improvements, we were able to achieve performance ratios shown in Table 6.

The data structures, P-Chase and NMCS algorithms, CUDA source codes and project files for Visual Studio 2022 are available for download at the URL http://anonymousdl.online/LION17/.

6 Conclusion

We have proven that running 32 simulations in parallel on a GPU warp loses an acceptable percentage of performance. Although random memory accesses are known to be extremely costly for a GPU, we were able to show that using shared memory could achieve a memory latencies 16 times slower than CPU memory latency, but with its 8704 cores, the NVIDIA GTX 3080 may achieve a speed of

$\times$420 compared to one Ryzen 9 3900X CPU core. All these observations allowed us to set up the first implementation of the NMCS algorithm on GPU. We test performance gain for the Snake In The Box problem in dimension 8,9 and 10. The performance we obtain corresponds to the order of magnitude that we had previously estimated, which in itself is a great success.

Using shared memory, we must respect the constraint of 100KB maximum for 128 cores which represents a very important constraint. This forces to greatly reduce game data in memory and to focus on problems with less than 1 KB of data per simulation. But on the other hand, the NVIDIA GTX-4090 card already offers twice as many CUDA cores compared to the 3080 and the next generation with the NVIDIA GTX-5090 will also double performance. We are probably at a technological tipping point where, for some games, it will be more efficient to generate playouts on a GPU than on a CPU. Indeed, the frantic race for performance that GPU founders are waging makes the power/price ratio more and more interesting compared to high-end CPUs.

References

1. Bouzy, B.: Monte-carlo fork search for cooperative path-finding. In: Computer Games Workshop at IJCAI. pp. 1–15 (2013)
2. Bouzy, B.: Burnt pancake problem: New lower bounds on the diameter and new experimental optimality ratios. In: SOCS. pp. 119–120 (2016)
3. Brodtkorb, A., Hagen, T., Schulz, C., Hasle, G.: Gpu computing in discrete optimization. part i: Introduction to the gpu. EURO Journal on Transportation and Logistics **2** (05 2013). https://doi.org/10.1007/s13676-013-0025-1
4. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games **4**(1), 1–43 (Mar 2012). https://doi.org/10.1109/TCIAIG.2012.2186810
5. Buber, E., Banu, D.: Performance analysis and cpu vs gpu comparison for deep learning. In: 2018 6th International Conference on Control Engineering & Information Technology (CEIT). pp. 1–6. IEEE (2018)
6. Cazenave, T.: Nested Monte-Carlo Search. In: Boutilier, C. (ed.) IJCAI. pp. 456–461 (2009)
7. Cazenave, T.: Generalized nested rollout policy adaptation. In: Monte Search at IJCAI (2020)
8. Cazenave, T., Fournier, T.: Monte Carlo inverse folding. In: Monte Search at IJCAI (2020)
9. Cazenave, T., Lucas, J.Y., Kim, H., Triboulet, T.: Monte Carlo Vehicle Routing. In: ATT at ECAI 2020. Saint Jacques de Compostelle, Spain (2020), https://hal.archives-ouvertes.fr/hal-03117515
10. Cazenave, T., Negrevergne, B., Sikora, F.: Monte Carlo graph coloring. In: Monte Search at IJCAI (2020)
11. Cazenave, T., Saffidine, A., Schofield, M.J., Thielscher, M.: Nested monte carlo search for two-player games. In: AAAI. pp. 687–693 (2016)
12. Cazenave, T., Teytaud, F.: Application of the nested rollout policy adaptation algorithm to the traveling salesman problem with time windows. In: Learning and Intelligent Optimization - 6th International Conference, LION 6. pp. 42–54 (2012)

13. Dang, C., Bazgan, C., Cazenave, T., Chopin, M., Wuillemin, P.: Monte carlo search algorithms for network traffic engineering. In: ECML PKDD 2021. Lecture Notes in Computer Science, vol. 12978, pp. 486–501. Springer (2021)
14. Dwivedi, A.D., Morawiecki, P., Wójtowicz, S.: Finding differential paths in arx ciphers through nested monte-carlo search. International Journal of electronics and telecommunications **64**(2), 147–150 (2018)
15. Edelkamp, S., Cazenave, T.: Improved diversity in nested rollout policy adaptation. In: KI 2016: Advances in Artificial Intelligence - 39th Annual German Conference on AI, Klagenfurt, Austria, September 26-30, 2016, Proceedings. pp. 43–55 (2016)
16. Edelkamp, S., Gath, M., Cazenave, T., Teytaud, F.: Algorithm and knowledge engineering for the tsptw problem. In: Computational Intelligence in Scheduling (SCIS), 2013 IEEE Symposium on. pp. 44–51. IEEE (2013)
17. Edelkamp, S., Gath, M., Greulich, C., Humann, M., Herzog, O., Lawo, M.: Monte-Carlo tree search for logistics. In: Commercial Transport, pp. 427–440. Springer International Publishing (2016)
18. Edelkamp, S., Gath, M., Rohde, M.: Monte-Carlo tree search for 3d packing with object orientation. In: KI 2014: Advances in Artificial Intelligence, pp. 285–296. Springer International Publishing (2014)
19. Edelkamp, S., Greulich, C.: Solving physical traveling salesman problems with policy adaptation. In: Computational Intelligence and Games (CIG), 2014 IEEE Conference on. pp. 1–8. IEEE (2014)
20. Edelkamp, S., Tang, Z.: Monte-Carlo tree search for the multiple sequence alignment problem. In: Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015. pp. 9–17. AAAI Press (2015)
21. Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., R Ruiz, F.J., Schrittwieser, J., Swirszcz, G., et al.: Discovering faster matrix multiplication algorithms with reinforcement learning. Nature **610**(7930), 47–53 (2022)
22. Kinny, D.: A new approach to the snake-in-the-box problem. In: ECAI 2012. Frontiers in Artificial Intelligence and Applications, vol. 242, pp. 462–467. IOS Press (2012)
23. Li, F., Ye, Y., Tian, Z., Zhang, X.: Cpu versus gpu: which can perform matrix computation faster—performance comparison for basic linear algebra subprograms. Neural Computing and Applications **31**(8), 4353–4365 (2019)
24. Méhat, J., Cazenave, T.: Combining UCT and Nested Monte Carlo Search for single-player general game playing. IEEE Transactions on Computational Intelligence and AI in Games **2**(4), 271–277 (2010)
25. Mei, X., Zhao, K., Liu, C., Chu, X.: Benchmarking the memory hierarchy of modern gpus. In: Hsu, C.H., Shi, X., Salapura, V. (eds.) Network and Parallel Computing. pp. 144–156. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
26. NVIDIA: Cuda c++ programming guide (November 2022), docs.NVIDIA.com/cuda/cuda-c-programming-guide, section: arithmetic-instructions
27. Portela, F.: An unexpectedly effective Monte Carlo technique for the RNA inverse folding problem. BioRxiv p. 345587 (2018)
28. Poulding, S.M., Feldt, R.: Generating structured test data with specific properties using nested Monte-Carlo search. In: GECCO. pp. 1279–1286 (2014)
29. Poulding, S.M., Feldt, R.: Heuristic model checking using a Monte-Carlo tree search algorithm. In: GECCO. pp. 1359–1366 (2015)

30. Rimmel, A., Teytaud, F., Cazenave, T.: Optimization of the Nested Monte-Carlo algorithm on the traveling salesman problem with time windows. In: EvoApplications. LNCS, vol. 6625, pp. 501–510. Springer (2011)
31. Rosin, C.D.: Nested rollout policy adaptation for Monte Carlo Tree Search. In: IJCAI. pp. 649–654 (2011)
32. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of go with deep neural networks and tree search. Nature **529**, 484–489 (2016)
33. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018)
34. Wong, H., Papadopoulou, M., Sadooghi-Alvandi, M., Moshovos, A.: Demystifying gpu microarchitecture through microbenchmarking. In: Performance Analysis of Systems and Software (ISPASS), 2010 IEEE International Symposium on. p. 235–246. IEEE (2010)
35. Wu, D.J.: Accelerating self-play learning in go. arXiv preprint arXiv:1902.10565 (2019)