The 32nd Annual Conference of the Japanese Society for Artificial Intelligence, 2018

Batch Random Walk for GPU-Based Classical Planning (Extended Abstract)

Ryo Kuroiwa Alex Fukunaga

Graduate School of Arts and Sciences, The University of Tokyo

Graphical processing units (GPUs) have become ubiquitous because they offer the ability to perform cost and energy efficient massively parallel computation. We investigate forward search classical planning on GPUs based on Monte-Carlo Random Walk(MRW). We first propose Batch MRW (BMRW), a generalization of MRW which performs random walks starting with many seed states, in contrast to traditional MRW which used a single seed state. We evaluate a sequential implementation of BMRW on a single CPU core and show that a sequential, satisficing planner based BMRW performs comparably with Arvand, the previous MRW-based planner. Then, we propose BMRW$_G$, which uses a GPU to perform random walks. We show that BMRWG achieves significant speedup compared to BMRW and achieves competitive performance on a number of IPC benchmark domains. This is an extended abstract of an ICAPS2018 paper [Kuroiwa 18].

1. Introduction

The use of Graphics Processing Units (GPUs) for general-purpose computing has become ubiquitous in many areas including AI, but their use in domain-independent planning has been quite limited. This seems largely due to the fact that there is a significant mismatch between the architecture of GPUs and forward heuristic search based algorithms commonly used for planning. For satisficing, classical planning, the most widely studied forward search strategy in recent years have been approaches such as Greedy Best First Search (GBFS), as well as many improvements which seek to avoid/escape local minima and plateaus.

In standard GBFS, Enhanced Hill-Climbing, and weighted A* approaches, each node expansion involves accessing global open/closed sets, which poses a challenge for efficient parallelization. Methods for efficiently distributing work in parallel best-first search (BFS) based planners on multi-core machines as well as clusters have been studied [Burns 10, Kishimoto 13], but these previous approaches for parallel BFS can not be straightforwardly applied to GPUs. One major issue is that GPUs provide thousands of cores/threads, but the amount of GPU RAM available per thread is quite limited. For example, a state-of-the-art Nvidia GTX1080 has 8GB global RAM, which must be shared by 2560 physical CUDA cores. This will be exhausted within a few seconds by a parallel BFS algorithm if the open/closed lists are stored on the GPU, as in GA*, a delayed duplicate detection based A* for the GPU [Zhou 15]. There is also a tiny amount of fast RAM per core ($\sim$375 bytes/core on a GTX1080), and although previous work has investigated performing domain-specific IDA* search using only this local memory [Horie 17], this is too small to hold even a single state for most domain-independent planning domains. Sulewski et al. ([Sulewski 11]) used a GPU to parallelize the successor generation step for breadth-first search in cost-optimal planning, with duplicate detection and open/closed list management performed on the CPU. A forward *heuristic* search method for satisficing planning which effectively uses the GPU has remained an open problem.

One approach to heuristic-driven forward search which encourages explorative search behavior and is suited for GPU parallelization is Monte Carlo Random Walk Planning (MRW) [Nakhost 09]. At each step, MRW starts at some state s (initially the initial state), performs a set of random walks from s, and then sets s to the best endpoint (according to a heuristic function) found by the random walks. State-of-the-art MRW-based planners have been shown to be competitive with GBFS-based approaches on some domains [Nakhost 13]. MRW appears to be suited for GPU parallelization because each random walk can be executed independently by a GPU thread.

In this paper, we first propose BMRW, a simple generalization of MRW-based search which combines an open list based search strategy with MRW. In BMRW, the search is driven by an open list, as in GBFS. In each iteration, a batch of nodes is selected from the open list, and random walks are performed starting with these nodes. Promising states found by the random walks are then inserted into the open list, and this cycle repeats until a goal is found or time runs out. We experimentally show that BMRW is a promising search strategy, and show that a planner using BMRW search on a single CPU core is competitive with Arvand13, the previous, state-of-the-art MRW based sequential planner.

Next, we propose BMRW$_G$, an efficient, parallel implementation of BMRW for GPUs. BMRW$_G$ maintains the open list in CPU memory, but uses the GPU for the random walks. We show that BMRW$_G$ achieves significant speedup compared to BMRW, and that BMRW$_G$ achieves competitive performance on a number of standard IPC benchmark domains.

2. Monte-Carlo Random Walk Planning

Monte-Carlo Random Walk Planning (MRW) was first proposed in the Arvand planner [Nakhost 09]. The state-of-the-art Arvand13 MRW-based planner, which performs significantly better than the original Arvand planner, works as follows: Given a state s, a basic *random walk* repeatedly generates successors of s, chooses one of the successors s' of s and transitions to s' ($s \rightarrow s'$). From the current node s (initially set to the start state), the Arvand13 MRW algorithm perform a set of random walks. Each state on the walk is evaluated according to a heuristic evaluation function h (the FF heuristic [Hoffmann 01] was used), and the walk returns

either when (a) it encounters a state with a better h-value than the random walk start state, in which case that state becomes the start point for the next random walk, or (b) with some probability (i.e., local restart). A global restart is triggered when h does not improve after some number of random walks. Arvand13 used an enhanced random walk which, instead of uniformly randomly choosing a successor, biased successor choice according to *helpful actions* identified by the FF heuristic. In addition, local and global start thresholds are set adaptively.

3. Batch Monte-Carlo Walk (BMRW)

We now propose BMRW, a generalization of MRW. BMRW, shown in Alg. 1, maintains a priority queue $openList$, initially containing the successors of the initial state s_0. In each iteration of the main loop (lines 21–36), BMRW first checks if $openList$ is empty, and if so, initializes it with the successors of the start state s_0, i.e., it performs a *global restart*. Then, a *batch* of $batchSize$ nodes is selected from $openList$. A random walk (Alg. 1, Walk function) of up to l steps is peformed from each start point in $batch$, and the results are stored in $walkres$.

The main differences between BMRW and MRW are: (1) MRW performs random walks from the same start state, while BMRW performs a set of random walks based on a batch of start states selected from $openList$. (2) MRW only keeps and updates a single "current state" (which is used as the start point for random walks), while BMRW maintains an $openList$, similar to GBFS-based approaches. Thus, BMRW can be viewed as a hybrid between MRW and GBFS. (3) In MRW, every random walk is followed by a possible update of the current state (jump to the state returned by the walk), whereas BMRW performs an entire batch of walks at a time.

In addition to the above basic scheme, BMRW uses an *elite insertion* policy (lines 32-36), where for the best n results (according to h-value) of the random walk return, the successors of those nodes are inserted into $openList$ instead of the nodes themselves. This is intended to strongly encourage further exploration of these "elite" nodes by pushing its many successors into $openList$ (because these successors also have good h-values, they are likely to be expanded soon).

Furthermore, a $closedList$ is used in order to prevent duplicate states from being pushed into the open list (lines 30-31). This ensures that each random walk starts from a different start state, promoting exploration of the search space.

MRW can be viewed as a special case of BMRW with batch size $b = 1$ and a special $openList$ limited to size 1.

3.1 BMRW$_G$: BMRW on a GPU

In principle, BMRW can be efficiently implemented on a GPU, due to the independence of each random walk. In Alg. 1 lines 25-26, the for loop is executed in parallel on the GPU. After each node in the batch selected in line 24 is copied to the GPU, each node in the batch is assigned to a GPU thread, and each random walk is performed by a single GPU thread, after which the result of the walk is copied back to the CPU. Everything else is performed on the CPU.

The main bottleneck in implementing BMRW on a GPU is the relatively small amount of GPU RAM available per thread. Although GPUs have a substantial amount of global memory available on the GPU, (the GTX1080 we used has 8GB), this

Algorithm 1 Batch MRW

```
1: function WALK(s, goals, l)
2:     s_best ← s
3:     for i ← 1 to l do
4:         if s ∈ goals then return s
5:         if DeadEnd(s) then s ← s_best
6:         else s ← RANDOMSELECT(successors(s))
7:         if h(s) < h(s_best) then s_best = s
8:     return s_best
9:
10: function FILLBATCH(openList, batchSize, batch)
11:     offset ← 0
12:     for i ← 1 to batchSize do
13:         if openList is Empty then
14:             if offset = 0 then State offset = i
15:             batch[i] ← batch[i − offset]
16:         else
17:             batch[i] ← POP(openList)
18:
19: function BATCHMRW(s_0, goals, l, batchSize, n)
20:     openList, closedList, batch, walkres ← φ
21:     loop
22:         if openList is Empty then
23:             openList ← successors(s_0)
24:         FILLBATCH(openList, batchSize, batch)
25:         for i ← 1 to batchSize do
26:             walkres[i] ← WALK(batch[i], goals, l)
27:         i ← 1
28:         for all s ∈ walkres do
29:             if s ∈ goals then return s
30:             if s ∈ closedList then continue
31:             INSERT(closedList, s)
32:             if i ≤ n then
33:                 PUSH(openList, successors(s), h(s))
34:                 i ← i + 1
35:             else
36:                 PUSH(openList, s, h(s))
```

global RAM must be shared among all threads. The GTX1080 has 2560 CUDA cores (20 Streaming Multiprocessors, 128 cores/SM), roughly 3125KB/thread. Although a random walk does not require much memory for state information, as there is no open/closed list during the walk, it is necessary to maintain data structures for fast, incremental computation of heuristic values during the walk. For example, to compute the FF heuristic using the generalized Dijkstra method [Liu 02, Keyder 08], the cost and unsatisfied preconditions of each ground action is required. On the GTX1080, if we use 2560 threads, each thread can handle approximately 390,000 ground actions – this is sufficient for most IPC benchmark domains. However, this implies that we currently can not run BMRW with tens of thousands of threads to take even better advantage of the ability of GPUs to overlap communication and memory access. Note that larger domains can be handled by reducing the number of threads, at the cost of leaving some cores idle.

We used the Landmark Count (LMC) heuristic [Hoffmann 04] for BMRW$_G$. The landmark graph is built on the CPU once at the beginning of search, transferred to the GPU global RAM, and shared by all GPU threads. Computing the LMC value only requires incrementally updating the number of unreached landmarks for the state. This requires $O(\#facts)$ memory. As shown below, BMRW with the LMC heuristic significantly outperforms BMRW with the FF heuristic.

4. Experimental Results

In our experimental evaluation (please see the full paper [Kuroiwa 18]), we evaluated a sequential implementation of BMRW on a single CPU core and showed that a sequential, satisficing planner based BMRW performs comparably with Arvand, the previous MRW-based planner. Then, we showed that BMRW$_G$ achieves significant speedup compared to BMRW and achieves

competitive performance on a number of IPC benchmark domains.

5. Conclusion

In order to exploit GPUs in domain-independent planning, we first proposed BMRW, a generalization of MRW which combines GBFS and random walk by performing a GBFS-like, $openList$-driven search, which at each iteration performs batches of random walks in order to explore the search space. *We showed that BMRW is competitive with previous random walk strategies, including Arvand13. We then showed that BMRW$_G$, a heterogeneous CPU/GPU implementation of BMRW, achieves significant speedup compared to BMRW and a straightforward parallelizatoin of MRW13.*

We have shown that random walk using the relatively lightweight Landmark Count heuristic can be efficiently implemented *entirely on the GPU*. In fact, MRW13, the baseline parallelization of Arvand13, runs almost entirely on the GPU, except for the top level loop, and in BMRW$_G$, only the global $openList$ and $closedList$ management are performed by the CPU. Our primary objective was to demonstrate the feasibility of GPU-based forward heuristic search, so we focused on efficient implementation on the GPU side, and the current implementation only uses a single CPU core. Effective, simultaneous usage of multiple CPU cores along with the GPU in a more heterogeneous algorithm is an avenue for future work. For example, $openList$ and $closedList$ management can be parallelized, as in [Sulewski 11].

While we focused on a relatively simple search strategy which is basically GBFS with (batched) random walk lookahead, random walk has been embedded as an exploration mechanism in other forward heuristic search variants such as RW-LS (ArvandLS) [Xie 12] and GBFS-LE [Xie 14]. It should be possible to combine these more complex algorithms with the basic idea of applying batches of random walks on the GPU with a diverse set of start points and a lightweight heuristic.

References

[Burns 10] Burns, E., Lemons, S., Ruml, W., and Zhou, R.: Best-first heuristic search for multicore machines, *Journal of Artificial Intelligence Research*, Vol. 39, pp. 689–743 (2010)

[Hoffmann 01] Hoffmann, J. and Nebel, B.: The FF Planning System: Fast Plan Generation through Heuristic Search, *J. Artif. Intell. Res.(JAIR)*, Vol. 14, pp. 253–302 (2001)

[Hoffmann 04] Hoffmann, J., Porteous, J., and Sebastia, L.: Ordered Landmarks in Planning, *J. Artif. Intell. Res.*, Vol. 22, pp. 215–278 (2004)

[Horie 17] Horie, S. and Fukunaga, A.: Block-Parallel IDA* for GPUs, in *Proceedings of the Tenth International Symposium on Combinatorial Search*, pp. 134–138 (2017)

[Keyder 08] Keyder, E. and Geffner, H.: Heuristics for Planning with Action Costs Revisited, in *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, pp. 588–592 (2008)

[Kishimoto 13] Kishimoto, A., Fukunaga, A., and Botea, A.: Evaluation of a simple, scalable, parallel best-first search strategy, *Artificial Intelligence*, Vol. 195, pp. 222–248 (2013)

[Kuroiwa 18] Kuroiwa, R. and Fukunaga, A.: Batch Random Walk for GPU-Based Classical Planning, in *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)* (2018)

[Liu 02] Liu, Y., Koenig, S., and Furcy, D.: Speeding Up the Calculation of Heuristics for Heuristic Search-based Planning, in *Eighteenth National Conference on Artificial Intelligence*, pp. 484–491, Menlo Park, CA, USA (2002), American Association for Artificial Intelligence

[Nakhost 09] Nakhost, H. and Müller, M.: Monte-Carlo Exploration for Deterministic Planning, in *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pp. 1766–1771, San Francisco, CA, USA (2009), Morgan Kaufmann Publishers Inc.

[Nakhost 13] Nakhost, H. and Müller, M.: Towards a Second Generation Random Walk Planner: An Experimental Exploration, in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pp. 2336–2342, AAAI Press (2013)

[Sulewski 11] Sulewski, D., Edelkamp, S., and Kissmann, P.: Exploiting the Computational Power of the Graphics Card: Optimal State Space Planning on the GPU., in *Proceedings of the International Conference of Automated Planning and Scheduling(ICAPS)* (2011)

[Xie 12] Xie, F., Nakhost, H., and Müller, M.: Planning via random walk-driven local search, in *Proceedings of the International Conference of Automated Planning and Scheduling(ICAPS)* (2012)

[Xie 14] Xie, F., Müller, M., and Holte, R.: Adding Local Exploration to Greedy Best-First Search in Satisficing Planning, in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pp. 2388–2394 (2014)

[Zhou 15] Zhou, Y. and Zeng, J.: Massively Parallel A* Search on a GPU, in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pp. 1248–1255 (2015)