Batch Random Walk for GPU-Based Classical Planning

Ryo Kuroiwa, Alex Fukunaga
Graduate School of Arts and Sciences
The University of Tokyo

Abstract

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 show experimentally that straightforward parallelizations of MRW perform poorly. Next, we 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 on BMRW performs comparably with previous state-of-the-art MRW-based planners. Then, we propose BMRW$_G$, which uses a GPU to perform random walks. We show that BMRW$_G$ achieves significant speedup compared to BMRW and achieves competitive performance on a number of IPC benchmark domains.

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 et al. 2010; Kishimoto, Fukunaga, and Botea 2013), but these previous approaches for parallel BFS cannot 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 and Zeng 2015). 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 and Fukunaga 2017), this is too small to hold even a single state for most domain-independent planning domains. Sulewski et al. (2011) 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 and Müller 2009). 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 and Müller 2013). 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 evaluate several implementations of a straightforward approach to parallelizing MRW on a GPU, and show that it is nontrivial to improve upon single-CPU performance with this straightforward approach.

Next, we 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. Then, 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.

Monte-Carlo Random Walk Planning

Monte-Carlo Random Walk Planning (MRW) was first proposed in the Arvand planner (Nakhost and Müller 2009). The state-of-the-art Arvand13 MRW-based planner (Nakhost and Müller 2013), 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 and Nebel 2001) 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. This trick is called Monte-Carlo Helpful Actions(MHA). In addition, local and global start thresholds are set adaptively.

Parallel Monte-Carlo Random Walk

We first investigate a straightforward approach to parallelizing MRW on the GPU. A natural approach executes an independent random walk on each GPU thread. However, the architectural features of a GPU impose several constraints. The main bottleneck is the relatively small amount of GPU RAM available per thread. For example, the GTX1080 we used has 8GB, but this global RAM must be shared among all threads – the GTX1080 has 2560 CUDA cores (20 Streaming Multiprocessors, 128 cores/SM), so roughly 3125KB/thread.

A random walk does not require much memory for state information, as there is no open/closed list during the walk. However, biasing the random walks using helpful actions is nontrivial, because helpful-action based biasing requires storing/updating the $Q(a)$-value for each ground action. Storing $Q(a)$ per thread is infeasible due to the limited amount of GPU global RAM per thread, while sharing/maintaining a global $Q(a)$ table among threads requires atomic updates (mutex accesses) which incur a large cost.

Also, warp divergence (threads within a warp performing different instructions) is quite costly, so although each random walk in Arvand13 has a local restart probability selected from $\{0.1, 0.01, 0.001\}$ according to the ϵ-greedy strategy, our parallel MRW implementation uses a fixed walk length selected from $\{10, 100, 1000\}$ for all threads.

Without helpful actions and local restarts, all of our attempts to implement the FF heuristic on the GPU have resulted in very poor performance (results not shown due to space, but worse than all results in Table 1. Thus, we used the Landmark Count (LMC) heuristic (Hoffmann, Porteous, and Sebastia 2004), which can be computed quickly on the GPU and requires relatively little memory. At the beginning of the search, the landmark graph is constructed on the CPU, transferred to the GPU global RAM, and shared by all GPU threads. Computing the LMC value on each parallel MRW thread only requires incrementally updating the number of unreached landmarks for the current walk state. This requires $O(\#facts)$ memory.

Experimental Evaluation of Parallel MRW

We evaluated our parallel MRW implementations and compared them to sequential MRW implementations as follows. We used a workstation with a Xeon E5-2650 v2 @ 2.60 GHz CPU and a GTX1080 GPU running Ubuntu 16.04.4 LTS, C++ (g++ 5.4.0, C++11, and CUDA 8.0). All GPU experiments used a Nvidia GTX1080 (2560 CUDA cores, 20 streaming multiprocessors, 8GB GPU RAM). We used benchmark instances from IPC11 and IPC14, excluding domains with conditional effects. All GPU algorithms used 5120 threads, which is a convenient multiple of the 2560 CUDA cores on the GTX1080. For each CPU algorithm, the average of 3 runs is shown. For domains duplicated in IPC11 and IPC14, we used the 2014 versions. All search algorithms were implemented from scratch, except for Arvand13, which we obtained from (Nakhost 2013). We used the PDDL to SAS+ translator from Fast Downward.

We evaluated the following algorithms:

Arvand13: original Arvand13 code from (Nakhost 2013).

MRW13$_C$/FF/MHA: our reimplementation of Arvand13 (uses the FF heuristic and biases actions using helpful actions, same as the original Arvand13).

MRW13$_C$/FF: same as MRW13$_C$/FF/MHA, but does **not** use helpful actions and chooses a successor uniformly.

MRW13$_C$/FF/FIX: same as MRW13$_C$/FF, but the random walk length is fixed and selected from $\{10, 100, 1000\}$ according to the ϵ-greedy strategy used in Arvand13.

MRW13$_C$/LMC: same as MRW13$_C$/FF, but uses the LMC heuristic.

MRW13$_C$/LMC/FIX: same as MRW13$_C$/FF/FIX, but uses the LMC heuristic.

MRW13$_C$/LMC/FIX/SH: same as MRW13$_C$/LMC/FIX, but selects the best state from a set of random walks (5120 walks in this experiment). The random walk length is selected in the same way as MRW13$_C$/LMC/FIX and shared by all random walks in the set.

MRW13$_G$/LMC/FIX: GPU adaptation of MRW13$_C$/LMC/FIX, where each GPU thread almost independently executes the search strategy of MRW13$_C$/LMC/FIX. However, all threads perform random walks of the same length at the same time.

MRW13$_G$/LMC/FIX/SH: GPU parallelization of MRW13$_C$/LMC/FIX/SH. All random walks in a set

are performed in parallel.

The coverage results are shown in Table 1. MRW13$_C$/FF/MHA, our reimplementation of Arvand13, is competitive with the original Arvand13 implementation, so it is an appropriate baseline for evaluating both sequential and parallel variants.

MRW13$_C$/FF, which does not use helpful actions to bias random walks, has significantly lower (189) coverage than MRW13$_C$/FF/MHA (229), and MRW13$_C$/FF/FIX, which performs fixed length random walks, performs worse (163) than MRW13$_C$/FF. Furthermore, in the sequential MRW implementations without helpful actions, the heuristic used (FF vs. LMC) does not have a large effect (MRW13$_C$/FF: 189 vs. MRW13$_C$/LMC: 188, MRW13$_C$/FF/FIX:163 vs. MRW13$_C$/LMC/FIX 180).

Next, we consider the performance of the GPU-parallel versions of MRW. MRW13$_G$/LMC/FIX, which simply executes MRW13$_C$/LMC/FIX independently on each thread, has the worst coverage (100) among all configurations, and is significantly worse than the single-thread CPU version, MRW13$_G$/LMC/FIX (180). This shows that trivially executing many independent instances of MRW is insufficient – merely running (in effect) many restarts in parallel is not enough to overcome the slow speed of each GPU core.

In contrast, MRW13$_G$/LMC/FIX/SH, which runs a set of random walks in parallel and selects the best result as the next start state, has significantly better coverage (179) than MRW13$_G$/LMC/FIX. Furthermore, MRW13$_G$/LMC/FIX/SH significantly outperforms its equivalent, sequential implementation MRW13$_C$/LMC/FIX/SH, showing that MRW13$_G$/LMC/FIX/SH utilizes the GPU resources effectively enough so that it is at least faster than a 1-core CPU implementation of the exact same search strategy. Nevertheless, MRW13$_G$/LMC/FIX/SH does not outperform the best sequential MRW implementations.

Therefore, these results show that *straightforwardly parallelizing MRW to the GPU is insufficient, and can result in worse than sequential performance. A search strategy which can better exploit the GPU is necessary.*

Batch Monte-Carlo Random Walk (BMRW)

We now propose BMRW, a generalization of MRW. BMRW (Alg. 1), maintains a h-value based priority queue openList, initially containing the successors of the initial state s_0. Each iteration of the main loop (lines 22–38) 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 (lines 11-18).[1] A random walk (Alg. 1, Walk function) of up to l steps is performed from each start point in batch, and the results are stored in walkres.

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

In addition, BMRW uses an *elite insertion* policy (lines 34-38), 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 32-33). 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 batchSize $= 1$ and a special openList limited to size 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 26-27, the for loop is executed in parallel on the GPU. After each node in the batch selected in line 25 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.

As with the best parallel MRW variant we found above (MRW13$_G$/LMC/FIX/SH), BMRW uses the LMC heuristic, and runs random walks with uniform lengths on all cores.

Experimental Evaluation of BMRW

We evaluated BMRW (same settings as the sequential and parallel MRW experiments above). For reference, we also include LAMA(Richter and Westphal 2010), the LAMA11 configuration of Fast Downward.

Evaluation of BMRW on a single CPU core

We evaluated the following algorithms:

BMRW$_C$/LMC/1/NC: BMRW$_C$/LMC/1 without closed list (i.e., duplicate random walk start states are not detected), batchSize $= 1$.

BMRW$_C$/LMC/1: BMRW with batchSize $= 1$, LMC heuristic.

BMRW$_C$/LMC/NE: BMRW with batchSize $= 5120$ without elite insertion – all successors of random walk results inserted into openList instead of only the elite n successors.

BMRW$_C$/LMC: CPU implementation of BMRW, as described in Alg. 1, LMC heuristic, batchSize $= 5120$.

[1] If openList contains $<$ batchSize nodes some nodes are repeatedly put in the batch, so that in the GPU version of BMRW, there are no idle GPU cores .

	Arvand13	MRW13$_C$/FF/MHA	MRW13$_C$/FF	MRW13$_C$/FF/FIX	MRW13$_C$/LMC	MRW13$_C$/LMC/FIX	MRW13$_C$/LMC/FIX/SH	BMRW$_C$/LMC/1/NC	BMRW$_C$/LMC/1	BMRW$_C$/LMC/NE	BMRW$_C$/LMC	BMRW$_C$/FF	MRW13$_G$/LMC/FIX	MRW13$_G$/LMC/FIX/SH	BMRW$_G$/LMC	LAMA	RW_{spdup}
total	204.3	229.7	189.3	163.0	188.7	180.3	111.7	129.3	203.7	202.3	223.3	130.7	100	179	266	292	
elevators	**20.0**	17.3	0.0	0.0	2.0	1.0	2.7	6.3	9.3	9.7	**10.0**	0.0	1	9	10	**20**	27.83
nomystery	8.7	8.7	**10.7**	9.3	10.3	8.7	7.3	0.0	5.0	11.0	**15.3**	13.7	6	15	**19**	11	3.43
parcprinter	**16.0**	**16.0**	8.3	0.0	4.7	4.0	3.7	0.0	0.0	15.3	**18.0**	**18.0**	0	11	**20**	19	4.34
pegsol	**19.3**	19.0	19.0	19.0	17.3	17.3	16.7	18.0	18.7	19.0	18.7	**20.0**	3	16	19	**20**	15.31
scanalyzer	17.3	17.0	17.0	16.3	**20.0**	19.7	16.7	16.3	**20.0**	**20.0**	**20.0**	13.0	**20**	**20**	**20**	**20**	43.96
sokoban	**2.7**	2.0	1.0	1.0	0.7	0.3	0.0	0.0	4.0	**4.7**	0.7	3.7	0	0	3	**16**	N/A
tidybot	12.7	13.3	13.3	13.7	**18.7**	18.0	2.7	6.7	**16.7**	11.3	13.3	0.0	7	11	**18**	16	3.26
woodworking	14.0	4.0	18.7	**19.3**	12.0	11.3	4.3	**19.0**	18.0	16.0	18.3	12.0	1	10	**20**	**20**	8.53
barman	15.0	**18.0**	0.0	0.0	0.0	0.0	0.0	0.0	0.3	4.7	**8.3**	0.0	0	0	12	**19**	1.52
childsnack	2.3	**3.7**	1.3	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.7	**1.7**	0	2	3	**5**	5.94
floortile	**5.7**	5.0	5.0	4.7	0.0	0.0	0.0	0.0	0.0	0.3	0.0	**6.0**	0	0	1	**2**	N/A
ged	0.0	0.0	0.0	0.0	**20.0**	19.3	17.7	**20.0**	**20.0**	**20.0**	**20.0**	0.0	16	**20**	**20**	**20**	17.78
hiking	17.7	18.0	19.7	**20.0**	18.0	16.0	14.0	3.3	**20.0**	**20.0**	**20.0**	**20.0**	19	18	**20**	15	19.02
maintenance	6.0	**16.3**	6.0	3.3	0.0	0.0	5.0	0.0	0.0	4.7	12.0	**15.3**	0	**12**	11	0	31.04
openstacks	14.0	**20.0**	**20.0**	14.0	**20.0**	**20.0**	0.0	19.7	**20.0**	0.0	0.0	0.0	6	9	10	**20**	N/A
parking	**0.7**	0.0	0.3	0.0	0.0	0.0	0.0	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	0	0	0	**20**	N/A
tetris	5.0	5.7	17.7	17.0	**20.0**	19.7	15.7	16.3	**20.0**	**20.0**	**20.0**	2.3	12	15	**20**	4	2.68
thoughtful	16.7	**19.7**	12.3	7.3	5.0	5.0	5.3	1.0	1.7	**18.0**	16.7	5.0	5	7	**19**	16	6.49
transport	10.7	**17.0**	0.0	0.0	0.0	0.0	0.0	2.0	10.0	7.7	**11.3**	0.0	0	0	**17**	10	27.58
visitall	0.0	9.0	19.0	17.0	**20.0**	**20.0**	0.0	0.7	**20.0**	0.0	0.0	0.0	4	4	4	**19**	N/A

Table 1: Results on IPC11/14 benchmarks (5min, 8GB CPU RAM). Coverage (# solved) is shown. RW_{spdup} is the speedup of the batch random walk (Alg. 1, lines 26-27) of BMRW$_G$ compared to BMRW$_C$ (walktime(BMRW$_C$)/walktime(BMRW$_G$)).

BMRW$_C$/FF: same as BMRW$_C$/LMC, but uses the FF heuristic(batchSize $= 5120$).

The results are shown in Table 1 for runs with a 5min, 8GB CPU RAM limit. The average of 3 runs is shown. The elite parameter n (Alg. 1, line 33) was 100, and the length of random walks l (Alg. 1, line 3) was 10. The batchSize was set to 5120 in order to allow easy comparison with the GPU implementation below. Tuning these control parameters for a CPU implementation of BMRW is future work.

Overall, BMRW$_C$/LMC performs comparably to both Arvand13 as well as our reimplementation, MRW13$_C$. We now consider the impact of several key features of BMRW, the closedList, the use of a batch, and elite insertion.

On the effect of detecting duplicate random walk start states using closedList BMRW$_C$/LMC/1/NC (batchSize $= 1$, no closedList, no elite insertion) performs significantly worse than BMRW$_C$/LMC/1 (batchSize $= 1$, no elite insertion). This shows that preventing duplicate random walk start states has a significant impact on performance.

On the effect of increasing batchSize (1 vs. 5120) Comparing BMRW$_C$/LMC/1 (batchSize $= 1$, no elite insertion) with BMRW$_C$/LMC/NE (batchSize $= 5120$, no elite insertion), we see that although increasing the batch size by itself does not significantly change total coverage, there is a significant change in the performance on individual domains. BMRW$_C$/LMC/1 (as well as MRW13$_C$/LMC, which behave somewhat similarly and use the same heuristic) perform well on `openstacks` and `visitall`. On the other hand, BMRW$_C$/LMC/NE, which differs from BMRW$_C$/LMC/1 only in increased batch size, fails to solve any instances in openstacks and visitall, but has significantly improved coverage on several domains (`parcprinter`, `thoughtful`, `nomystery`, `maintenance`). Thus, processing large batches of random walks results in significantly different search behavior compared to performing 1 random walk at a time in the BMRW framework.

On the effect of elite insertion Comparing BMRW$_C$/LMC/NE (batchSize $= 5120$, no elite insertion) and BMRW$_C$/LMC (batchSize $= 5120$, elite insertion), overall coverage improves, showing that inserting successors of top n nodes with lowest h-value into openList (elite insertion) vs. just inserting all the successors, helps focus the search on more promising areas of the search space.

On the effect of the heuristic BMRW$_C$/FF, which uses the FF heuristic, has a significantly lower coverage than BMRW$_C$/LMC. This is because evaluating the FF heuristic at every step of each random walk in the batch is quite expensive, and does not allow the search to explore the space as rapidly as with the faster LMC heuristic. Thus, the LMC heuristic is more suitable for BMRW$_C$ than the FF heuristic.

Evaluation of BMRW$_G$ (BMRW on the GPU)

First, we evaluated the speedup of BMRW$_G$ (5120 threads) compared to BMRW$_C$ on the CPU (batch size 5120, as

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 s is a dead end 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
15:                 offset = i
16:             batch[i] ← batch[i - offset]
17:         else
18:             batch[i] ← POP(openList)
19:
20: function BATCHMRW(s_0, goals, l, batchSize, n)
21:     openList, closedList, batch, walkres ← φ
22:     loop
23:         if openList is Empty then
24:             openList ← successors(s_0)
25:         FILLBATCH(openList, batchSize, batch)
26:         for i ← 1 to batchSize do
27:             walkres[i] ← WALK(batch[i], goals, l)
28:         SORT(walkres)              ▷ ascending order of h-value
29:         i ← 1
30:         for all s ∈ walkres do
31:             if s ∈ goals then return s
32:             if s ∈ closedList then continue
33:             INSERT(closedList, s)
34:             if i ≤ n then          ▷ top n best h-value nodes
35:                 PUSH(openList, successors(s), h(s))
36:                 i ← i + 1
37:             else
38:                 PUSH(openList, s, h(s))
```

in the previous experiment). Figure 1 compares the wall-clock runtimes for BMRW$_G$ and BMRW$_C$. As shown in the scatterplot, BMRW$_G$ achieves significant speedup compared to the CPU implementation, particularly on harder instances. To isolate the speedup on the random walks (excluding the sequential overhead of the rest of the algorithm), Table 1 shows RW_{spdup} per domain, the speedups of the *random walk computations only* (i.e., all CPU computations excluded), indicating that the random walks (including all heuristic computations) are sped up significantly by the GPU implementation.

Next, we evaluated the coverage of BMRW$_G$/LMC, the GPU implementation of BMRW as described above. BMRW$_G$ almost dominates BMRW$_C$, showing that BMRW clearly benefits from GPU parallelization.

Although LAMA has higher overall coverage than BMRW$_G$, BMRW$_G$ has a higher coverage on 9/20 domains (`nomystery`, `parcprinter`, `tidybot`, `hiking`, `openstacks`, `tetris`, `thoughtful`, `transport`), so

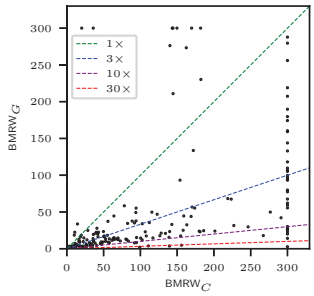

Figure 1: Comparison of search time: (seconds) between BMRW$_C$ vs. BMRW$_G$ (from same data as Table 1). Unsolved instances are shown as time=300.

BMRW performance is complementary to that of state-of-the-art GBFS-based search.

Conclusion

In order to exploit GPUs in domain-independent planning, we 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$_C$ and a straightforward parallelization 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$_G$, 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, Edelkamp, and Kissmann 2011).

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, Nakhost, and Müller 2012) and GBFS-LE (Xie, Müller, and Holte 2014). 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.

Acknowledgments

This research was supported by Kakenhi 17K00296.

References

Burns, E.; Lemons, S.; Ruml, W.; and Zhou, R. 2010. Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research* 39:689–743.

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

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *J. Artif. Intell. Res.* 22:215–278.

Horie, S., and Fukunaga, A. 2017. Block-parallel IDA* for GPUs. In *Proceedings of the Tenth International Symposium on Combinatorial Search*, 134–138.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2013. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence* 195:222–248.

Nakhost, H., and Müller, M. 2009. Monte-Carlo exploration for deterministic planning. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, 1766–1771. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Nakhost, H., and Müller, M. 2013. Towards a second generation random walk planner: An experimental exploration. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, 2336–2342. AAAI Press.

Nakhost, H. 2013. Arvand source code. https://github.com/nhootan/Arvand2011.

Richter, S., and Westphal, M. 2010. The lama planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Int. Res.* 39(1):127–177.

Sulewski, D.; Edelkamp, S.; and Kissmann, P. 2011. 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)*.

Xie, F.; Müller, M.; and Holte, R. 2014. 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.*, 2388–2394.

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

Zhou, Y., and Zeng, J. 2015. 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.*, 1248–1255.