2010 IEEE International Conference on Robotics and Automation
Anchorage Convention District
May 3-8, 2010, Anchorage, Alaska, USA

High-dimensional Planning on the GPU

Joseph T. Kider Jr., Mark Henderson, Maxim Likhachev, and Alla Safonova
University of Pennsylvania

Abstract— **Optimal heuristic searches such as A* search are commonly used for low-dimensional planning such as 2D path finding. These algorithms however, typically do not scale well to high-dimensional planning problems such as motion planning for robotic arms, computing motion trajectories for non-holonomic robotic vehicles and motion synthesis for humanoid characters. A recently developed randomized version of A* search, called R* search, scales to higher-dimensional planning problems by trading off deterministic optimality guarantees of A* for probabilistic sub-optimality guarantees. In this paper, we show that in addition to its scalability, R* lends itself well to a parallel implementation. In particular, we demonstrate how R* can be implemented on the GPU. On the theoretical side, the GPU version of R*, called R*GPU, preserves all the theoretical properties of R* including its probabilistic bounds on sub-optimality. On the experimental side, we show that R*GPU consistently produces lower cost solutions, scales better in terms of memory, and runs faster than R*. These results hold for both motion planning for a 6DOF robot arm planar as well as 2D path finding.**

I. INTRODUCTION

A* search is a widely-used tool for finding a least-cost path in a graph. This technique is a provably optimal algorithm in terms of both the quality of the path it finds as well as the amount of work (state computations) the search has to do to in order to guarantee the optimality of the solution [1]. The provable optimality of the solution however comes at the expense of thorough exploration of a large fraction of states in the graph. Such exploration prohibits the application of A* search to higher-dimensional planning problems in which even a small fraction of the space contains too many states to explore all in run-time. To address this, a number of suboptimal variants of A* have been proposed [2], [3], [4], [5], that aim for the provable suboptimality of the solution instead. While they can often scale to much larger problems [3], [5], [6], [7], [8], they still perform a thorough exploration of the (much smaller) state-space and rely heavily on the guiding power of the heuristic function.

In contrast to A* and its variants, randomized motion planning techniques such as Probabilistic Roadmaps and RRTs [9], [10] explore the state-space sparsely. Sparse exploration makes these approaches suitable to high-dimensional planning but completely gives up the optimality guarantees. R* search [11] is a recently developed randomized version of A* search that offers a compelling compromise. R* explores the state-space in a sparse fashion. Scheduling the exploration in a clever way provides probabilistic guarantees on suboptimality and good cost minimization.

In this paper, we show that the structure of R* search makes it well-suited to a parallel implementation. R* search was originally designed for sequential processing. It operates by decomposing the usual single-shot A* search into a series of properly-scheduled *short-range and easy-to-solve* searches, each guided by the heuristic function towards a randomly chosen goal. It turns out that this decomposition lends itself naturally to a parallel implementation. First, each short-range search is independent of others which makes them suitable for running in parallel. Second, each search is short-range and easy-to-solve. This means that each search doesn't require vast amounts of memory. This allows for multiple searches to share the DRAM on the GPU. Finally, each search in R* discards its memory after it exits. This eliminates the need for time-consuming transfers of memory and makes it ideal for running in the DRAM on the GPU.

The paper is structured as follows. We first briefly describe the related work including several GPU-based graph search implementations. We then give a short overview of GPU architecture and an overview of R* search and the guarantees it makes. In section IV, we present the implementation of R* search on the GPU, that we refer to as R*GPU. In sections V and VI, we evaluate the performance of R*GPU experimentally on two domains: motion planning for a 6DOF simulated planar robot arm and a 2D path finding. Both domains show that R*GPU consistently produces lower cost solutions, scales better in terms of memory, runs faster, has a much higher chance of finding a feasible solution, and can solve much harder problems than R*. Since R*GPU also preserves all the theoretical guarantees of R*, our experimental analysis concludes that R*GPU is a better choice for planning if GPU hardware is available.

II. RELATED WORK

A. Parallel-based Planning

Planning tends to become computationally burdensome. This serves as a good motivation for a number of researchers working on developing Parallel-based implementations of path finding. For example, Bleiweiss [12] presented a GPU-based implementation of A* in the context of global 2D planning. His GPU-based A* implementation showed roughly an order of magnitude improvement to a single and two-threaded CPU-based implementation of A* using C++. Similarly, Katz and Kider [13] presented a cache-effcient GPU implementation of the all-pairs shortest-path problem and demonstrated that it results in a significant performance improvement. Edelkamp and Sulewski [14] obtained speed-ups for breadth-first search using a bitvector representation of the search frontier on the GPU. Burns et al. [15], [16] presented a parallel version of best-first search for an 8 core machine that combines duplication detection and speculative expansion. Zhou and Hansen [17] parallelized graph search using structured duplicate detection. Evett et al. [18] designed a variant of A* search on SIMD Connection Machine. Kishimoto et al. [19] parallelize A* by distributing and scheduling work among processors based on a hash function of the search state.

The work we present in this paper differs in that we are concerned with problems that cannot afford thorough exploration of the state-space characteristic of A* and its variants. We therefore present a GPU-based implementation of an R* search that combines A*-type search with randomized sparse exploration of the state-space. It sacrifices deterministic guarantees on optimality and sub-optimality for the ability to plan in large and/or high-dimensional state-spaces. In addition to reduced time complexity, R* (and consequently R*GPU) imposes drastically smaller memory requirements than optimal graph searches such as A*, making it even more suitable to GPU hardware.

B. Planning Algorithms

R*, and consequently R*GPU, fall into the category of heuristic searches such as A* [1] and its suboptimal variants [2], [3], [4], [5]. These searches examine states in the search-space thoroughly and in a systematic manner and therefore return solutions that are often of much better quality than those found by randomized planners. However, the methodological exploration limits them from being widely applicable to high-dimensional planning.

Within the class of heuristic searches, R* is somewhat related to K-best-first search [20]. The latter limits the number of successors for each expanded state to at most K states. R* also does this, but these successors are remote, and each transition to them is computed via a separate short-range search. All short-range searches run in parallel in R*GPU.

Both R* and R*GPU are also closely related to the family of randomized motion planners [9], [10]. These algorithms have gained tremendous popularity in the last decade. They have been shown to consistently solve impressive high-dimensional motion planning problems. In addition, these methods are simple, fast and general enough to solve a variety of motion planning problems. R* and R*GPU differ from these algorithms in several aspects. Most importantly, the current randomized planners are mainly concerned with finding any feasible path rather than minimizing the cost of the solution. In addition, they provide no guarantees on the sub-optimality of the solution. In contrast, R* and R*GPU try to find the solutions with minimal cost and provide probabilistic guarantees on the quality of the solution. These two aspects of these algorithms are important when solving planning problems for which the minimization of the objective function is important (such as planning dynamically-feasible trajectories for unmanned vehicles [8], motion planning for highly articulated robots [21], [22], motion synthesis of human characters in animation [23] and others).

```
1  select unexpanded state $s \in \Gamma$ (priority is given to states not labeled AVOID)
2  if path that corresponds to the edge $bp(s) \rightarrow s$ has not been computed yet
3    try to compute this path
4    if failed then label state $s$ as AVOID
5    else
6      update $g(s)$ based on the cost of the found path and $g(bp(s))$
7      if $g(s) > w\,h(s_{\text{start}}, s)$ label $s$ as AVOID
8  else //expand state $s$ (grow $\Gamma$)
9    let $SUCCS(s)$ be $K$ randomly chosen states at distance $\Delta$ from $s$
10   if goal state within $\Delta$, then add it to $SUCCS(s)$
11   for each state $s' \in SUCCS(s)$, add $s'$ and edge $s \rightarrow s'$ to $\Gamma$, $bp(s') = s$
```

Fig. 1. Single iteration of R*

The most relevant work to ours is a very recently and independently developed Randomized A* algorithm [24]. The major difference from our work is that Randomized A* mainly targets continuous domains and does not provide the analysis of bounds on sub-optimality. Diankov and Kuffner's work contains a number of interesting ideas including the use of statistical learning to learn the heuristic function in order to avoid tweaking its parameters.

III. Overview

A. G80 Architecture and CUDA

With the introduction of NVIDIA G80 GPU architecture an abundance of scientific and numeric general purpose GPU applications found performance gains due to the graphics hardware's streaming data-parallel organizational model. The graphics pipeline now features a single unified set of processors that function as vertex, geometry, and fragment processors. Additionally, the release of the Compute Unified Device Architecture (CUDA) API [25] on the G80 architecture allows developers to easily develop and manage general purpose scientific and numerical algorithms without formulating solutions in terms of nontrivial shaders and graphic primitives making the programming model much more programmer friendly.

The GPU serves, to an extent, as a coprocessor to the CPU programmed through the CUDA API. A single program known as a kernel is compiled to operate on the GPU device to exploit the massive data parallelism inherent to Single Instruction, Multiple Data (SIMD) architecture. Groups of threads then execute the kernel on the GPU. This batch of threads is organized as a grid of thread blocks. Data in a block is shared through a high-speed shared memory region (16 kB in size) on the G80 architecture accessible to the programmer and explicitly managed through CUDA. CUDA also defines per thread registers, read only constants, and texture memory, and per grid global memory. This memory hierarchy facilitates higher bandwidth and overall performance gains. Therefore, algorithms must manage memory effectively to achieve maximum performance.

B. R* Algorithm

R* operates by constructing a small graph Γ of sparsely placed states, connected to each other via edges. Each edge represents a path in the original graph in between the corresponding states in Γ. In this respect, Γ is related to the graphs constructed by randomized motion planners [9], [10]. The difference is that R* constructs Γ in such a way as to provide explicit minimization of the solution cost and probabilistic guarantees on the suboptimality of the solution. To achieve these objectives, R* grows Γ in the same way as A* grows a search tree.

Every iteration, R* selects the next state s to expand from Γ (see figure 1). While normal A* expands s by generating all the immediate successors of state s, R* expands s by generating K states residing at some distance Δ from s (lines 8-11). The distance Δ is some metric that measures how far two states are from each other. This metric can be domain dependent or independent, such as the difference in heuristic values of two states. The metric should be applicable to whatever domain we are solving, be it a discrete or continuous one. If a goal state is within Δ from state s then it is also generated as the successor of s. R* grows Γ by adding these successors of s and edges from s to them.

A path that R* returns is a path in Γ from the start state to the goal state. This path consists of edges in Γ. Each such edge, however, is actually a path in the original graph. Finding each of these (local) paths may potentially be a challenging planning task. R* postpones finding these paths until necessary and tries to concentrate on finding the paths that are easy to find instead. It does this by labeling the states to which it cannot find paths easily as AVOID states. Initially, when generating K successors, none of these states are labeled as AVOID - R* does not try to compute paths to *all* of the generated states. Instead, only when state s is selected for expansion does R* try to compute a path from the predecessor of s, stored in the backpointer of s $bp(s)$, to state s (lines 2-7).

R* uses the weighted A* search with heuristics inflated by w to compute local paths. R* stops the search, however, if it fails to find the path easily. (Different heuristics can be used to establish when to stop the search, for example a time limit or number of state expansions. In our experiments, we used a threshold of 1024 generated states to detect that a path can not be found easily.) If the search does fail, then R* labels state s as an AVOID state since it assumes that it will be time-consuming to find a path to state s. If the weighted A* search does find a path, then the cost of the found path can be used to assign the cost of the edge $bp(s) \rightarrow s$. The cost of the edge and the cost of the best path from s_{start} to $bp(s)$, stored in $g(bp(s))$, can then be used to update $g(s)$ in the same way A* updates g-values of states.

R* provides probabilistic guarantees on the suboptimality of the solution. The uncertainty in the guarantee is purely due to the randomness of selecting K successors during each expansion. For a given graph Γ, on the other hand, R* can state that the found path is no worse than w times the cost of an optimal path that uses only the edges in Γ.

Suppose $g(s) \leq wh(s_{\text{start}}, s)$. Then the cost of the found path from s_{start} to s via the edges in Γ is clearly no worse than w times the cost of an optimal path, since $h(s_{\text{start}}, s)$ is supposed to be no more than the cost of the optimal path. Suppose now $g(s) > w\,h(s_{\text{start}}, s)$. This means that a path from s_{start} to s may not be w suboptimal. To prove otherwise, similarly to weighted A*, R* needs to expands all the states s' in Γ with $f(s') = g(s') + w\,h(s') \leq g(s) + w\,h(s) = f(s)$. Expanding all of these states, however, is computationally expensive, because some of these states are labeled AVOID and therefore require the computation of hard-to-find local paths to them. Moreover, it may even be unnecessary to use state s. For a given w, there often exist a wide spectrum of solutions that satisfy w suboptimality though some are easier to find than others. Therefore, R* considers the states s with $g(s) > wh(s_{\text{start}}, s)$ as the states it should also avoid expanding. It labels these states as AVOID (line 7).

```
1  procedure UpdateState($s$)
2  if ($g(s) > w\,h(s_{\text{start}}, s)$ OR
           ($path_{bp(s),s} = \textbf{null}$ AND $s$ is labeled AVOID))
3    insert/update $s$ in OPEN with priority $k(s) = [1,\ g(s) + w\,h(s, s_{\text{goal}})]$;
4  else
5    insert/update $s$ in OPEN with priority $k(s) = [0,\ g(s) + w\,h(s, s_{\text{goal}})]$;

6  procedure ReevaluateState($s$)
7  $[path_{bp(s),s}, c_{low}(path_{bp(s),s})]$ = TrytoComputeLocalPath($bp(s)$, $s$);
8  if ($path_{bp(s),s} = \textbf{null}$ OR
           $g(bp(s)) + c_{low}(path_{bp(s),s}) > w\,h(s_{\text{start}}, s)$)
9     $bp(s) = \arg\min_{s' | s \in SUCCS(s')}(g(s') + c_{low}(path_{s',s}))$;
10    label $s$ as AVOID state;
11  $g(s) = g(bp(s)) + c_{low}(path_{bp(s),s})$;
12  UpdateState($s$);

13 procedure RStarSearch()
14 $g(s_{\text{goal}}) = \infty$, $bp(s_{\text{goal}}) = bp(s_{\text{start}}) = \textbf{null}$, $k(s_{\text{goal}}) = [1, \infty]$;
15 $OPEN = CLOSED = \emptyset$;
16 $g(s_{\text{start}}) = 0$;
17 insert $s_{\text{start}}$ into OPEN with priority $k(s_{\text{start}}) = [0, w\,h(s_{\text{start}}, s_{\text{goal}})]$;
18 while ($k(s_{\text{goal}}) \geq \min_{s' \in OPEN} k(s')$ AND $OPEN \neq \emptyset$)
19   remove $s$ with the smallest priority from OPEN;
20   if $s \neq s_{\text{start}}$ AND $path_{bp(s),s} = \textbf{null}$
21     ReevaluateState($s$);
22   else //expand state $s$
23     insert $s$ into CLOSED;
24     let $SUCCS$ = set of $K$ randomly chosen states at distance $\Delta$ from $s$
25     if distance from $s_{\text{goal}}$ to $s$ is smaller than or equal to $\Delta$
26       $SUCCS(s) = SUCCS(s) \cup \{s_{\text{goal}}\}$;
27     $SUCCS(s) = SUCCS(s) - SUCCS(s) \cap CLOSED$
28     for each state $s' \in SUCCS(s)$
29       $[path_{s,s'}, c_{low}(path_{s,s'})] = [\textbf{null}, h(s, s')]$;
30       if $s'$ is visited for the first time
31         $g(s') = \infty$, $bp(s') = \textbf{null}$;
32       if $bp(s') = \textbf{null}$ OR $g(s) + c_{low}(path_{s,s'}) < g(s')$
33         $g(s') = g(s) + c_{low}(path_{s,s'})$; $bp(s') = s$;
34         UpdateState($s'$);
```

Fig. 2. The pseudocode of R*

To provide the suboptimality guarantees and minimize solution costs while avoiding as much as possible the states labeled AVOID, R* selects states for expansion in the order of smaller $f(s) = g(s) + w\,h(s)$), same as in weighted A*. However, it selects these states from the pool of states not labeled AVOID first. Only when there are no more such states left, R* starts selecting AVOID states (in the same order of f-values).

Implementation Details The pseudocode of the R* algorithm is given in figure 2. The algorithm first goes through the initialization of variables. The g-values are estimates of the distance from the start state to the state in question, same as in normal A* search. bp-values are backpointers in graph Γ that can be used to backtrack the solution after the search terminates. k-values are priorities used to select states for expansion from *OPEN* - the list of states in Γ that have not been expanded yet. A state with the minimum priority is always selected for expansion first. Priorities are two-dimensional values and are compared according to the lexicographical ordering (first dimension is compared first, and the second is used to break ties). Whenever a state is labeled AVOID, the first dimension of its priority is set to 1. Otherwise, it is 0. This way, the states labeled AVOID are only selected for expansion if there are zero states not labeled AVOID left to expand. The second dimension of the priority $k(s)$ is $f(s) = g(s) + w\,h(s)$, where h-values are heuristic values and must be consistent [1].

As in weighted A*, R* expands states until the priority of the goal state is smaller than the smallest priority in *OPEN*. The lines 22-34 correspond to a normal expansion of state s. It generates K random successors of state s, that haven't been expanded (closed) previously and goal state if within Δ from s. For each generated state s', it then sets $path_{s,s'} =$ **null** to represent that no path from s to s' has been found yet. The cost of the edge $s \rightarrow s'$ is therefore set to the heuristic estimate of the distance, $c_{low}(path_{s,s'}) = h(s, s')$, which is an admissible estimate. Finally, similar to (weighted) A*, R* tries to decrease the g-value of state s if it has been already generated previously.

If R* selects a state s for expansion and the path to s from its parent $bp(s)$ in Γ has not been computed yet, then R* tries to compute this path first by calling the function ReevaluateState(s) on line 21. If path is found, R* updates the cost $c_{low}(path_{bp(s),s'})$ of the edge $bp(s) \rightarrow s$ based on the cost of the found path. If not found, weighted A* search is supposed to return the smallest (un-inflated) f-value of a state in its queue which can be used to set the edge cost $c_{low}(path_{s,s'})$ to an improved estimate of the path. Depending on whether R* successfully finds the path and how costly the path is, R* labels s as AVOID or not (described above). If it does set the state as AVOID, it re-computes the best predecessor of s (in case there are multiple predecessors) and sets $bp(s)$ accordingly. Afterwards, $g(s)$ is updated based on the g-value of $bp(s)$ and the cost of the edge in between $bp(s)$ and s.

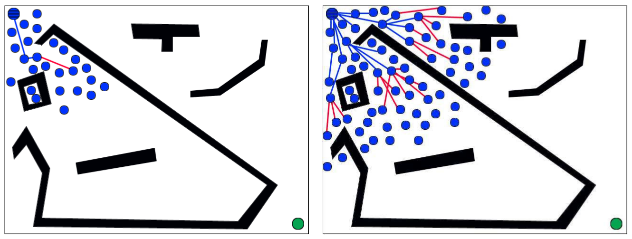
After the while loop of R* terminates, the solution can be re-constructed by following backpointers bp backwards starting at state s_{goal} until s_{start} is reached. The path is given in terms of edges in R*, but each edge is guaranteed to have a local path computed. Thus, the solution in the original graph can be re-constructed.

IV. Parallelization of R*GPU

It turns out that the decomposition of a single-shot search into a series of easy-to-solve short-range searches lends itself naturally to a parallel implementation on GPU. In particular, while the main loop (figuring out what short-range search to run next) can run on CPU, each of the short-range searches can run on a thread in CUDA. This results in significant speedups for the following reasons. First, each short-range search is independent of others, which makes it suitable for running them in parallel. Second, each search is short-range and easy-to-solve search. This means that each search does not require vast amounts of memory. This allows for multiple searches to share states in the DRAM on the GPU so there are no unnecessary expansions. Finally, each search in R* discards its memory after it exits. This eliminates the need for time consuming transfers of memory and makes it ideal for running in the DRAM on the GPU.

Figure 4 shows the high-level pseudocode of R*GPU. R*GPU runs the high level loop on CPU while performing all short-range (local) weighted A* searches in parallel on GPU (line 6). The search performs the same initialization as R*. In addition, before the main loop is executed, R*GPU generates K random successors of the start state. In the main loop, R*GPU repeatedly selects M states from *OPEN* with minimum priorities. For all the states s for which the paths from their respective predecessors $bp(s)$ to s have not been found yet, R*GPU uses GPU to run local weighted A* to compute these paths (line 6). The main difference between R*GPU and R* is that unlike the sequential R*, R*GPU computes up to M paths to the selected M states, all at the same time. To compute all M paths simultaneously, we run local weighted A* searches on GPU in parallel using one of the two methods described in our implementation section. After the GPU is done processing the weighted A* searches, the costs of the found paths are retrieved from GPU, the state values are updated and states are re-inserted into *OPEN*. Figure 3 shows graphically the difference between R* and R*GPU. After three timesteps (where each timestep corresponds to the time allocated to a single local weighted A* search), R* computes graph Γ that contains only three local paths (Figure 3(a)). In contrast, R*GPU computes a graph Γ that contains many more edges (Figure 3(b)). As a result, R*GPU searches the state-space far more effectively than its sequential version R*.

When the main loop terminates, the path from start to goal is computed. At this point, the path is given by a series of edges in the high-level graph Γ. To reconstruct the actual path, R*GPU re-executes local weighted A* searches to find the actual paths that correspond to each of the edges in the path in graph Γ (line 13). Once again, this operation is done on GPU in parallel. The found paths can now be combined into a single path that represents a path from start to goal in the original graph.



(a) graph Γ generated by R* (b) graph Γ generated by R*GPU

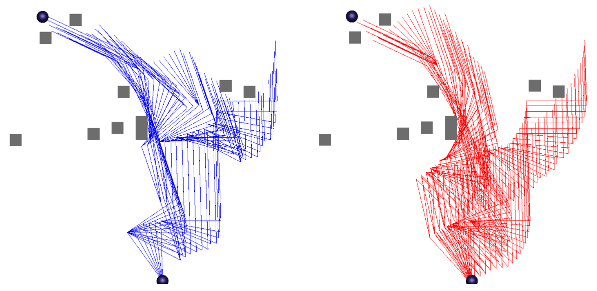
Fig. 3. High-level graphs Γ generated for 2D planning after 3 iterations of the main loop in R* and R*GPU. The circles represent generated states.

A. Implementation on GPU

We have implemented the local weighted A* searches in R*GPU using two different implementations. In the first implementation, we utilized a hash table that fully protects

```
initialize {lines 14-17 in Fig. 2}
generate $K$ random successors at distance $\Delta$ from $s_{\text{start}}$
Loop until $s_{\text{goal}}$ is expanded
  retrieve $M$ states $s$ with minimum priority $k$ from *OPEN*
  for every selected state $s$ s.t. $s \neq s_{\text{start}}$ AND $path_{bp(s),s} = \textbf{null}$
    run a local weighted A* on GPU to find a path from $bp(s)$ to $s$
  retrieve the costs of all found paths from GPU and re-insert them into *OPEN*
  for all selected states $s$ that have not been sent to GPU for processing
    expand $s$ {lines 22-34 in Fig. 2}
if $path_{bp(s),s_{\text{goal}}} \neq \textbf{null}$
  re-construct the path in $\Gamma$ from $s_{\text{start}}$ to $s_{\text{goal}}$
  for every state $s$ in the found path
    run a local weighted A* on GPU to find a path from $bp(s)$ to $s$
  retrieve the found paths
  combine the found paths into a single path from $s_{\text{start}}$ to $s_{\text{goal}}$
```

Fig. 4. The high-level pseudocode of R*GPU



(a) motion generated by R*GPU (cost=78) (b) motion generated by R* (cost=101)

Fig. 5. Motions generated for a simulated 6 DOF robot arm after 30 secs of planning

against having any duplicate states (in the experimental results section we will refer to this implementation as full). This method allows us to always find a state that has already been generated and therefore no state is ever generated more than once. This is a standard implementation of A* search. In the second implementation, we only performed a partial duplicate detection (we will refer to this implementation as part). In this implementation, whenever we generate state s and therefore need to see if it is already in the hash table, we only check against the most recently added state in the corresponding bin in the hash table. If this most recently added state is indeed state s, we then do not need to re-generate state s (i.e., duplicate is removed). Otherwise, s is added to the hash table, even if the hash table already contains it (i.e., duplicate is generated). This partial duplicate elimination procedure results in some states being expanded more than once, but drastically reduces the number of if-else statements that need to be executed on GPU. The current GPU architecture handles divergent branches poorly. In particular, Bleiweiss [26] describes how hash table construction is a single threaded operation and is very expensive on the GPU. Our partial duplicate elimination implementation therefore mitigates thread divergence and minimizes data accesses which are expensive. Since threads are running multiple weighted A* searches in parallel, we therefore avoid allocating memory on the fly using a structured array.

GPU Data Layout : We store our data in an efficient collection of structure-of-arrays (SoA). This improves the possibility of coalesced memory transactions on graphics hardware. Most of our data structures reside in global memory on the GPU so we can increase the amount of generated states per local search. Our search elements consist of simple structs aligned to 64 bit coalesced global memory access to foster aligned accesses on the graphics card. The rest of the variables reside in a single cycle register memory on the GPU to increase performance.

V. Experimental Results

We implemented our R*GPU algorithm in the CUDA programming language. All tests were performed on a machine with an NVIDIA Tesla C870 card with 1.5 GB of GPU memory and with a Core 2 Quad processor running at 2.33 GHz and 3 GB memory. In the first set of experiments, we have evaluated the performance of R*GPU on simple 2D planning on an eight-connected grid. In our tests, we dedicated each R* or R*GPU search 10 seconds for planning and then executed these R* and R*GPU searches repeatedly for 2 minutes each. This corresponded to running as many R* (R*GPU) searches as possible within 2 minutes to obtain the best cost across these runs. Since each R* (R*GPU) search comes with the probability of obtaining w-suboptimal solution, running repeated R* (R*GPU) searches increases this probability proportionally to the number of runs. Based on these runs, we reported three performance measures: (a) the best cost across all R* (R*GPU) searches executed within the 2 minutes; (b) the number of R* (R*GPU) searches that have successfully found solution within 10 seconds allocated to them; and (c) the total number of local weighted A* searches called by all of the R* (R*GPU) searches executed within the 2 minutes. The reported performance measures (Table I) are the (full) implementation of R* and (part)implementation of R*GPU, which were the best set of configuration parameters for both the CPU and GPU. We averaged over on 90 randomly generated gridworld maps of varying obstacle density (20% and 40%). In addition, we have also evaluated the performances of R* and R*GPU on both actual (simple) edgecost computations and artificially slowed-down (hard) edgecost computations. The actual edgecosts were Euclidean distances and were therefore cheap to compute. The artificially slowed-down computations in 2D involved a series of trigonometric calculations to simulate complicated planning domains. Motion planning for non-circular robots and motion planning for robotic arms involve significantly more computationally expensive cost evaluations. We analyze a more realistic problem domain, motion planning for a robot arm, in our detailed experimental analysis which includes computationally expensive edge cost computations. In all the experiments, the heuristic we used was based on Euclidean distance.

The results show that R*GPU outperforms the CPU version of R* as obstacle density increases and as the edgecost computation becomes time-consuming, which is often the case when planning for complex systems. The number of local weighted A* searches decreases drastically on the CPU as the complexity of edgecost computations increases. In these cases, R*GPU manages to perform over 30 times more of weighted A* searches and consequently finds solutions of smaller cost and with higher probability.

2D Planning				
Performance Measurement	Obstacle Density	Planner	Simple Expand	Hard Expand
Best Cost	20%	R*GPU	321.31	**330.80**
		R*	**316.75**	344.19
	40%	R*GPU	**347.72**	**361.64**
		R*	349.90	392.39
# of succ R*	20%	R*GPU	69.87	**5.94**
		R*	**2461.55**	3.55
	40%	R*GPU	23.56	**3.21**
		R*	**45.54**	2.15
# of Local A*	20%	R*GPU	**79808.71**	**7114.38**
		R*	50327.09	204.42
	40%	R*GPU	26020.94	**1469.31**
		R*	**54045.52**	185.58

TABLE I
EXPERIMENTAL RESULTS: RESULTS FOR THE 2D PLANNER

In the second set of experiments, we compared the performance of R*GPU with the CPU version of R* on a simulated 6 degree of freedom (DOF) planar robotic arm (shown in Figure 5) [11]. Each R* or R*GPU search was dedicated 30 seconds for planning, but just as in the first experiments, we ran as many R* (R*GPU) searches as possible within 5 minutes. Based on these runs, we computed the best cost across the runs, the number of successful R* (R*GPU) searches, and the number of local weighted A* searches performed across all R* (R*GPU) runs within the 5 minutes. We averaged these results over both (full and part) implementations on 53 randomly generated maps of varying obstacle placement.

As shown in Figure 5, in all of the experiments, the base of the arm was fixed, and the task was to move its endeffector to the goal (small circle on the left) while navigating around obstacles (indicated by grey rectangles). The resulting state-space was over 3 billion states. The cost of each change in a joint angle was 1. We tested our algorithm using three settings of w. A smaller value of w relates to a better bound on suboptimality and therefore makes the search harder. The results shown in Table II demonstrate that R*GPU produces consistently a significantly lower "best cost", and within five minutes we execute over 38 times more of successful R*GPU searches and over 64 times more of weighted A* searches than when executing R* searches on CPU (Note that the shown numbers are ratios). As a result, R*GPU has a much higher chance of finding a feasible solution, and can solve much harder problems than R* could. Our detailed experimental analysis in the following section illustrates this result in more detail.

VI. DETAILED EXPERIMENTAL ANALYSIS

In this section, we present the detailed experimental study of the performance of R*GPU on a simulated 6 degree of freedom planar robotic arm [11]. In these experiments, we have tested our algorithm using three settings of w and two thresholds for the number of generated states within each local weighted A* search: 512 states and 1024 states. As before, the smaller value of w relates to a better bound on suboptimality and therefore makes the search harder. The larger threshold on states generated within each local weighted A* search allows for the search to search longer before a state is deemed as an AVOID state. However, this comes at the cost of more expensive memory accesses. We tested both (full and part) implementations of R* and R*GPU. All 12 (3 values of w, 2 values of thresholds, and 2 implementations) variants of R* and R*GPU were tested on 53 randomly generated maps of varying obstacle placement, three times each. For each trial we ran as many R* (R*GPU) searches as possible within five minutes and restricted each R* (R*GPU) to no more than 30 seconds of planning.

[1]The goal state is partially specified in 2D by the end effector of the robot arm.

6 DOF Robot Arm		
Performance Measure	w	R*GPU/R*
Best Cost	2	0.965
	4	0.921
	6	0.918
# of Succ R*	2	38.5556
	4	37.516
	6	24.917
# of Local A*	2	24.899
	4	44.268
	6	64.262

TABLE II
EXPERIMENTAL RESULTS: AVERAGE OF PER RUN RATIO (R*GPU / R*) FOR 3 DIFFERENT w SETTINGS FOR THE 6 DOF ROBOT ARM.

Number of Successful R* Searches: Figure 6 compares the number of R* searches and the number of R*GPU that completed successfully in the allotted time of 30 seconds per search over a total five minutes of running time. For cases with smaller bounds on suboptimality ($w = 2$), R*GPU completes 36 to 38 times more often than R*.

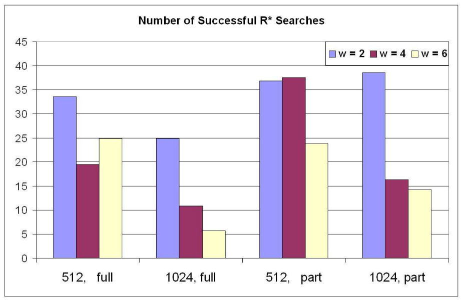


Fig. 6. Average ratio of successful searches (R*GPU / R*) within 5 min.

Total Number of A* Searches: Figure 7 compares the total number of local weighted A* searches performed by R* and R*GPU across all runs within five minutes. According to the results, R*GPU was able to run two to five times more local weighted A* searches than its sequential version.

Best Cost: Figure 8 shows that the motions generated by five minutes of repeated R*GPU searches are consistently of lower costs across all settings of R* and R*GPU algorithms. Though, as w decreases (in other words, the search aims for a better suboptimality bound), the gap between solution costs

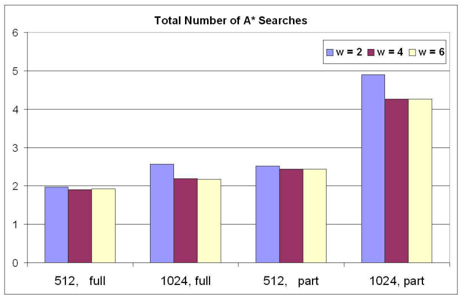


Fig. 7. The average ratio of the total number of local weighted A* searches (R*GPU / R*).

returned by R*GPU and R* becomes smaller.

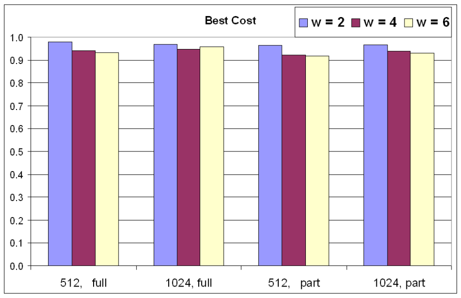


Fig. 8. The average Best Cost ratio (R*GPU / R*).

Percentage of Runs Completed: Figure 9 compares the percentage of runs completed within five minutes. The R*GPU (left bar) completes at a much higher rate across all settings. For example, for $w = 2$ with threshold set to 512 states, 85% of R*GPU searches successfully completed whereas only 40% of CPU R* searches were successful. This is because R*GPU executes more local weighted A* searches in parallel and therefore covers a larger area of the state space.

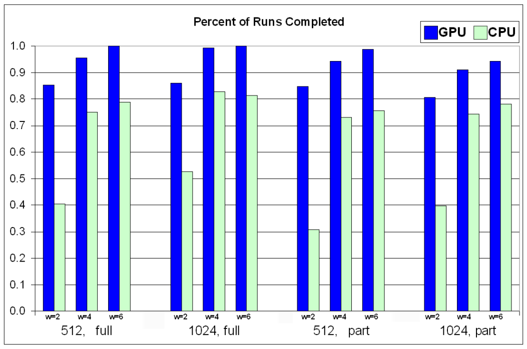


Fig. 9. The average percentage of R* and R*GPU searches completed within five minutes.

Number of Goal Paths: Since R* and R*GPU try to satisfy the bounds on suboptimality, they may actually reach the goal state before they terminate. In particular, they may find multiple paths to the goal. Only the least-cost path is finally recorded. However, looking at these statistics (shown in Figure 10), R*GPU reaches the goal state 65x - 309x more often than its CPU version depending on the setting. This is so, because each R*GPU runs many more local weighted A* searches. As a result, the GPU version of R* has a much higher chance of producing a lower cost path since it generates many more paths to the goal state.

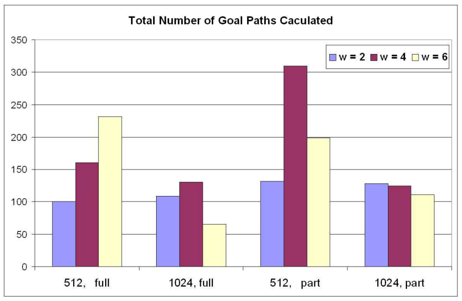


Fig. 10. The average ratio of total number of goal paths (R*GPU / R*).

Performance Over Time: Figure 11 plots the performance of three runs of five minute planning using repeated R*GPU and repeated R* searches on one of the random maps. The plots show that R*GPU consistently produces a lower cost solution quicker than R*. This implies that R*GPU may be executed fewer times than the CPU version of R* to reach the desired solution quality. Figure 12 plots all the solution costs returned by R* and R*GPU over the five minute window. This plot demonstrates that there are more R*GPU searches completed and they have a much higher probability of producing a lower cost.

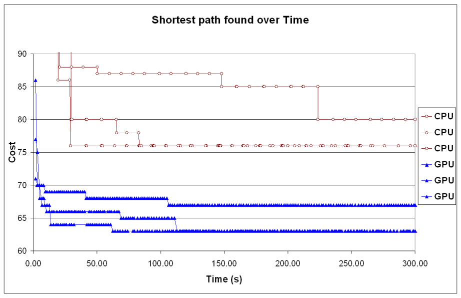


Fig. 11. Three trials showing the Best Cost tracked across five minute planning with R* and R*GPU.

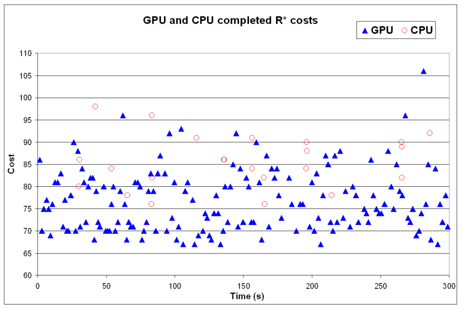


Fig. 12. Solution costs returned by R* and R*GPU in three trials, each consisting of five minute planning with repeated R* and R*GPU searches.

Problem Hardness: Table III outlines three performance metrics (best cost, # of Succ R* and # of local A*) for different w settings and how they relate to the complexity of the problem. The hardness of the planning problem was strongly correlated with obstacle density: the higher the density, the more difficult it was to find a collision-free motion for the robot arm. The results show that for easy maps the GPU version performs equally well in best cost, but the CPU version successfully completes many more easy R* searches. As maps become very hard however, the CPU version fails to solve any problems, whereas the R*GPU is able to solve many. The conclusion is that the R*GPU is better suited to planning for hard planning problems as opposed to the problems in which heuristics are already guiding the search well.

6 DOF Robot Arm						
Performance Measurement	w	Planner	Easy	Medium	Hard	Very Hard
Best Cost	2	R*GPU	**50**	**61**	**59.7**	N/A
		R*	**50**	67	N/A	N/A
	4	R*GPU	**51**	**63.3**	**65.7**	**89.7**
		R*	**51**	69.3	80	N/A
	6	R*GPU	**51**	**64**	**66.3**	**91**
		R*	**51**	70.7	79.5	N/A
# of Succ R*	2	R*GPU	330.3	**166**	**129.3**	N/A
		R*	**1,819**	15	N/A	N/A
	4	R*GPU	391.3	**218.7**	**257.3**	**7.3**
		R*	**12,554.7**	85.3	3.7	N/A
	6	R*GPU	353.3	**208.3**	**258.3**	**27.3**
		R*	**16,415.3**	69	1.0	N/A
# of Local A*	2	R*GPU	**128,412**	**113,887.7**	**137,810.7**	**179,778.3**
		R*	73,059.3	57,238.3	59,686.7	62,225.3
	4	R*GPU	**106,858.3**	**94,006**	**116,268.7**	**144,635.7**
		R*	68,654.7	52,449.0	57,931	61,117.7
	6	R*GPU	**107,342.3**	**93,833.3**	**114,4799.7**	**134,779.3**
		R*	69,766.7	44,451	60721.3	60,489.7

TABLE III

DETAILED EXPERIMENTAL RESULTS SHOWING THE DEPENDENCY ON PROBLEM HARDNESS.

VII. CONCLUSIONS

In this paper, we presented a novel implementation of a randomized heuristic search, namely R* search. Our experimental analysis shows that for easy problems with cheap edge cost computations, the need for the GPU version of R* is unnecessary. In fact, the CPU version of R* performs better. Many real planning problems, especially in robotics, are hard and involve expensive cost computations. For these problems, the GPU version of R* offers a significant improvement in performance in terms of lower solution cost and higher chances of finding a feasible solution. In the future, we intend to port the R*GPU search onto a real robotic manipulator and evaluate its performance there. We also plan on further investigating how and what planning in robotics can take advantage of parallel processing hardware that is readily available today.

VIII. ACKNOWLEDGEMENTS

This work was in part supported by Willow Garage. The authors would also like to thank NVIDIA for providing graphics hardware to help test our implementation. The authors would like to thank Norm Badler and Amy Calhoun for all their insightful help.

REFERENCES

[1] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[2] D. Furcy, "Chapter 5 of speeding up the convergence of online heuristic search and scaling up offline heuristic search," Ph.D. dissertation, Georgia Institute of Technology, 2004.

[3] R. Zhou and E. A. Hansen, "Multiple sequence alignment using A*," in *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2002, student abstract.

[4] R. Zhou and E. Hansen, "Beam-stack search: Integrating backtracking with beam search," in *ICAPS*, 2005, pp. 90–98.

[5] M. Likhachev, G. Gordon, and S. Thrun, "ARA*: Anytime A* with provable bounds on sub-optimality," in *Advances in Neural Information Processing Systems (NIPS) 16*. Cambridge, MA: MIT Press, 2003.

[6] S. Rabin, "A* speed optimizations," in *Game Programming Gems*, M. DeLoura, Ed. Rockland, MA: Charles River Media, 2000, pp. 272–287.

[7] J. Gaschnig, "Performance measurement and analysis of certain search algorithms," Carnegie Mellon University, Tech. Rep. CMU-CS-79-124, 1979.

[8] M. Likhachev and D. Ferguson, "Planning long dynamically-feasible maneuvers for autonomous vehicles," in *Proceedings of Robotics: Science and Systems (RSS)*, 2008.

[9] L. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[10] J. Kuffner and S. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2000.

[11] M. Likhachev and A. Stentz, "R* search," in *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2008.

[12] A. Bleiweiss, "GPU accelerated pathfinding," in *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 65–74.

[13] G. J. Katz and J. T. Kider, "All-pairs shortest-paths for large graphs on the GPU," in *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 47–55. [Online]. Available: http://portal.acm.org/citation.cfm?id=1413966

[14] S. Edelkamp and D. Sulewski, "Parallel state space search on the GPU," in *Proceedings of the International Symposium on Combinatorial Search*, 2009.

[15] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Parallel best-first search: Optimal and suboptimal solutions," in *Proceedings of the International Symposium on Combinatorial Search*, 2009.

[16] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Suboptimal and anytime heuristic search on multi-core machines," in *ICAPS*, 2009.

[17] R. Zhou and E. Hansen, "Parallel structured duplicate detection," in *National Conference on Artificial Intelligence (AAAI)*, 2007, pp. 1217–1222.

[18] M. Evett, A. Mahanti, D. Nau, J. Hendler, and J. Hendler, "PRA*: Massively parallel heuristic search," *Journal of Parallel and Distributed Computing*, vol. 25, pp. 133–143, 1995.

[19] A. Kishimoto, A. Fukunaga, and A. Botea, "Scalable, Parallel Best-First Search for Optimal Sequential Planning," in *Proceedings of the International Conference on Automated Scheduling and Planning ICAPS-09*, Thessaloniki, Greece, 2009, pp. 201–208.

[20] A. Felner, S. Kraus, and R. E. Korf, "Kbfs: K-best-first search," *Annals of Mathematics and Artificial Intelligence*, vol. 39, no. 1-2, pp. 19–39, 2003.

[21] D. Burfoot, J. Pineau, and G. Dudek, "RRT-plan: a randomized algorithm for strips planning," in *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2006.

[22] S. Morgan, "Sampling-based planning for discrete spaces," Ph.D. dissertation, Case Western Reserve University, 2004.

[23] A. Safonova and J. K. Hodgins, "Construction and optimal search of interpolated motion graphs," in *ACM Trans. Graph.*, 2007, p. 106.

[24] R. Diankov and J. Kuffner, "Randomized statistical path planning," in *Proceedings of IEEE/RSJ 2007 International Conference on Robots and Systems (IROS)*, October 2007.

[25] NVIDIA Corporation, "NVIDIA CUDA compute unified device architecture programming guide," Jan. 2007, http://developer.nvidia.com/cuda.

[26] Avi Bleiweiss, "Multi agent navigation on GPU," Apr. 2009, http://developer.download.nvidia.com/presentations/2009/GDC/ MultiAgentGPU.pdf.