GPU-based Dynamic Search on Adaptive Resolution Grids

Francisco M. García1, Mubbasir Kapadia2, and Norman I. Badler2

1School of Computer Science, University of Massachusetts - Amherst
2Computer and Information Sciences Department, University of Pennsylvania

Abstract—This paper presents a GPU-based wave-front propagation technique for multi-agent path planning in extremely large, complex, dynamic environments. Our work proposes an adaptive subdivision of the environment with efficient indexing, update, and neighbor-finding operations on the GPU to address several known limitations in prior work. In particular, an adaptive environment representation reduces the device memory requirements by an order of magnitude which enables for the first time, GPU-based goal path planning in truly large-scale environments (> 2048 m²) for hundreds of agents with different targets. We compare our approach to prior work that uses an uniform grid on several challenging navigation benchmarks and report significant memory savings, and up to a 1000X computational speedup.

I. INTRODUCTION

Path finding has been extensively researched and still remains one of the most fundamental problems in robotics, artificial intelligence, and computer animation. Finding an optimal route to a goal in a dynamic environment is a difficult problem, which is further exacerbated when we deal with a large number of goals and agents. In such scenarios, each agent performs a search to obtain an initial solution, detects environment changes that invalidate their current paths, and reacts accordingly in an independent fashion. Furthermore, traditional methods require agents to plan from scratch when the goal changes.

GPU capabilities can be exploited to mitigate the impact of multiple agents by simultaneously searching for an optimal path for all agents sharing a common goal. The work in [1] presents a GPU based planning technique that can compute paths for multiple agents with the same goal simultaneously, while efficiently repairing solutions to accommodate dynamic changes in the environment. However, due to its considerable memory overhead of using a uniform grid discretization for the environment, it is limited to agents traveling to the same target, or small-scale environments. In addition, even simple environments with large open spaces require the search to explore an enormous state space, which significantly impacts the computational performance, while yielding little gains to ensure path optimality.

In this paper, we present a GPU based dynamic planner which preserves the same search properties of the previous work. We show a method that is able to handle a very large number of goals with a large number of agents by addressing the limitations described in [1]. We adaptively discretize the environment into variable resolution quads, using finer resolution only where necessary, thus significantly reducing the size of the state space. This also reduces considerably the memory footprint allowing us to handle very large environments in GPU memory and also accelerates the search process.

Using an adaptive environment representation on the GPU has two main challenges: indexing is no longer constant time and handling dynamic changes is computationally expensive, since the number and location of neighbor quads varies as obstacles move in the world. Given these properties, it is impossible to know ahead of time how many neighbors a given quad has, which ones those neighbors are, and how many quads are needed to represent the state space. In addition, dynamically allocating memory on the GPU to accommodate these changes can be very expensive. We address these challenges by: (1) using a quadcode scheme for performing efficient indexing, update, and neighbor finding operations on the GPU, and (2) efficiently handling dynamic environment changes by performing local repair operations on the quad tree, and performing plan repair to resolve state inconsistencies without having to plan from scratch.

We demonstrate the benefits of our method on a variety of challenging benchmarks with empirical results and discuss the trade-offs between using an adaptive-resolution and an uniform grid. Our work addresses several important challenges that furthers the efficient use of next generation hardware for massively parallel path planning of autonomous agents in large-scale dynamic environments.

II. PREVIOUS WORK

There has been a considerable amount of work in path planning leading to current efforts. A* provides strict optimality guarantees, but it is unable to handle dynamic world updates without discarding previous search efforts. AD* [2] addresses this issue while satisfying time constraints by quickly generating sub-optimal solutions and gradually converging to an optimal result. Other search techniques, [3], [4] based on expanding trees have been developed and improved by exploiting multiple processors, but these methods are not well suited for massive parallelization on graphics cards.

Wavefront based method are ideally suited for parallelization and have successfully been implemented on the GPU. The work in [5] presents a model for analyzing performance of wavefront algorithms. [6] proposes a focused wavefront expansion for path planning, where the expansion of the algorithm is directed towards the solution instead of being
spread across the entire environment. Several GPU accelerated planners were proposed in [7] which provide great performance boosts. Crowd simulation techniques [8], [9] exploit multi-core CPU’s and GPU’s for parallelizing local collision avoidance.

In addition, several techniques have been devised to efficiently traverse and update quadtrees and octrees. The work in [10] demonstrates a method to quickly perform point location, search and insertion operations in randomized and deterministic quad-trees by using compressed quad-trees. Efficient adjacency detection algorithms for quad-trees were proposed in [11] based on a series of arithmetic operations performed on quadcodes [12]. The work in [13] describes a method of rapidly creating an adaptive mesh based on quadtrees, but is not well suited for a GPU implementation.

III. DYNAMIC SEARCH ON THE GPU FOR UNIFORM GRIDS

The work in [1] presents an efficient GPU wavefront-based planner by taking advantage of the parallel nature of a grid representation of the state space. The cost $g(s)$ of reaching a given state $s$ is defined as $g(s) = g(s') + c(s, s') \times \text{cost}(s)$, where $s$ is the current state, $s'$ its predecessor, $c(s, s')$ is the cost of going from $s'$ to $s$. The cost of a particular state $\text{cost}(s)$ encodes untraversable regions such as obstacles and areas of high cost (e.g., rough terrain, water). For a given environment, a wave is propagated from the goal to all other states in the environment to compute the cost of reaching the goal from any other state. Once $g(s)$ is computed, the optimal path to the goal from any state in the environment can be efficiently computed by tracing the least cost path to the goal. A novel termination condition is proposed which minimizes the number of GPU iterations for wave propagation needed to guarantee optimal paths for all agents in the environment. This approach requires multiple copies of the map for each new goal, since $g(s)$ is computed by propagating a wave from a particular target.

Limitations. One of the major limitations of this approach stems from the memory usage necessary to keep track of all states in the environment. The entire map is subdivided in cells of a fixed size, regardless of whether or not an obstacle is present in that region. For large uniform spaces this is extremely wasteful, both in terms of memory and computation, as the wave needs to propagate through each state in the region of same cost, which could be estimated in a single iteration, at a slight compromise to path optimality. Furthermore, this problem is exacerbated if we attempt to handle multiple objectives. In these situations, a different copy of the state space needs to be created for each goal. In addition to wasting the limited GPU memory, this approach makes the process much slower, for there are many more states that need to be considered before reaching a solution.

Proposed Solution. Instead of representing the state space as a uniform grid, we use an adaptive resolution grid which reduces the number of states required to represent the same map. This, in turn, significantly reduces the amount of memory needed in the GPU to keep track of search efforts and greatly increases performance. There are several challenges for porting a hierarchical representation of the environment to the GPU, including the efficient indexing of neighbor quads and update to accommodate dynamic environment changes. We address these challenges and propose a GPU-based wave propagation technique on adaptive resolution grids for multi-agent planning in large, complex, dynamic environments. Our solution preserves all the properties of [1], while addressing the aforementioned limitations, enabling the real-time planning of hundreds and thousands of autonomous agents in extremely large, complex, dynamic environments.

IV. METHOD OVERVIEW

On the GPU, the environment is adaptively subdivided into a hierarchical quad tree representation, which is ported onto the GPU to compute an initial plan. Environment changes are monitored on the CPU which triggers local repairs in the quad tree. GPU memory is updated to reflect environment changes, and the plan is efficiently repaired by updating only the costs which have been invalidated as a result of the change.

A hierarchical representation of the environment for GPU computations presents several challenges. Quad tree traversal is inherently a recursive operation on the number of neighbors of a quad cannot be known in advance, which makes it difficult to use a wavefront-propagation technique in a massively parallel fashion. To address these challenges, we use an efficient quadcode method for performing indexing, update, and neighbor finding operations. Quadcodes can be efficiently computed using simple arithmetic operations in an independent fashion, which makes them amenable for GPU processing. The main components of our system are enumerated below, and elaborated in subsequent sections: Figure 1 provides an overview of the GPU-based planning framework using adaptive-resolution grids on a simple benchmark.

- Initialization. Data structures used to represent the environment, GPU memory allocation, and indexing to reduce the performance impact of dynamically changing environments.
- Plan Computation. Wavefront propagation on an adaptive resolution environment representation, with minimal number of GPU iterations for plan computation.
- Plan Repair. Efficient plan repair to accommodate dynamic changes in the environment.

V. METHOD INITIALIZATION

The environment is initially subdivided using a quad-tree up to a predefined maximum depth, which prevents the tree from growing to the point of practically becoming a uniform grid in highly populated environments. Once the tree is created, we transfer all the leaf nodes which correspond to quad regions in the environment $Q$, the number of quad divisions $n_q$, and the number of goals $n_g$ to the GPU by calling $\text{generateMap}(Q, n_q, n_g)$ and $\text{createHashMap}(Q, n_q)$.
The function `generateMap` allocates memory for \( n_q \) quad structures and \( n_q \times n_g \) cost structures in host memory. A quad structure is defined as \( Q = \langle (x,y), \text{cost}, \text{code}, \text{index} \rangle \), where \((x,y)\) is the center position, \( \text{cost} \) is the cost of transitioning to this quad, \( \text{code} \) is a unique quad identifier, and \( \text{index} \) is the position the quad occupies in the quad list. It is worth noting the importance of \( \text{code} \) and \( \text{index} \). The \( \text{code} \) is used as the key into the hashmap to retrieve the quad structure, while \( \text{index} \) indexes into the computed costs for all goals with respect to this quad.

As with a uniform grid method, a different copy of the map has to be created for each goal in order to handle multiple goals. Each copy of a map is represented as an array of costs, and the cost associated with a particular goal for the same quad can be retrieved by using \( \text{index} \) in the corresponding cost array. A cost structure is defined as \( \langle g, \text{pred\_code} \rangle \), where \( \text{pred\_code} \) is the code of the neighbor quad \( q' \) with minimum g-value, and \( g \) is computed as:

\[
g(q) = g(q') + c(q, q') \cdot \text{cost}(q),
\]

where \( \text{cost}(q, q') \) is the Euclidean distance between the centers of \( q \) and \( q' \), and \( \text{cost}(q) \) is the cost of transitioning to \( q \). The reason why we create these two structures is twofold. First, we save memory by creating several smaller representations of a given quad when we have multiple goals (since many attributes do not change for a quad regardless of its goal), and only keeping one copy of the whole quad structure. Second, accessing these smaller structures in device memory can be done much faster on GPU because memory becomes highly coalesced. As can be seen in Fig. 2, there is only one quad structure per quad and as many cost structures as goals in the map.

The final initialization step is to compute the neighbors for each quad. Given a predetermined maximum number of neighbors per quad (we used 20), we allocate an array large enough to be able to support our needs throughout the lifetime of the planner. This array contains the indices at which each neighbor is located in the cost struct array. An index of -1 means that no neighbor has been computed at that position. The neighbors for each quad are computed as described in [11], and the planner is run to generate the initial plan. Fig. 3 shows how a quad can identify its corresponding neighbor. The image shows a simple map divided into four quads and the corresponding tree. Each node in the tree can calculate an offset, given its position, in the neighbors array and find all adjacent quads. Having a predefined maximum number of neighbors allows us to easily identify the starting offset that corresponds to each quad. We can iterate through the neighbor indices to find all neighbors for a given quad; if we find a neighbor index of -1 that means that there are no more neighbors for that particular quad.
VI. ENVIRONMENT REPRESENTATION

We subdivide the environment by using a quad-tree up to a predefined maximum depth. In our tests a depth of 7 gave a good trade-off between plan quality and performance. As the tree is being created, a code is assigned to each quad in the following manner: the root is assigned 0 and it is always subdivided into four children. The top left child is assigned a code of 1, top right is assigned 2, bottom left child 3 and the bottom right child is assigned a code of 4. For each child node if we find that the quad contains an obstacle, it is subdivided into four child quads, and its code is equal to the code based on its relative location to its parent, appended to the parent’s code. Quadcodes are an efficient method for computing unique identifiers for quads using simple arithmetic operations in an independent fashion, which can be used for indexing, checking if a particular quad exists, and computing neighbors in parallel. For more details on how quadcodes are computed, please refer to [11], [12].

Fig. 4 depicts a simple quad-tree being subdivided by a moving obstacle. In 4(a) we observe the initial tree subdivision where no obstacles are present. We divide the root of the tree in four high-level quads with their corresponding quadcodes. In 4(b) an obstacle enters the region defined by quad 1, thus dividing it into four quads and the code corresponding to its parent is appended to the parent’s code. Quadcodes are a good trade-off between plan quality and performance. As the tree is being created, a code is assigned to each quad and the code is equal to the code based on its relative location to its parent, appended to the parent’s code. Quadcodes are an efficient method for computing unique identifiers for quads using simple arithmetic operations in an independent fashion, which can be used for indexing, checking if a particular quad exists, and computing neighbors in parallel. For more details on how quadcodes are computed, please refer to [11], [12].

At the end of the process, a reference for each quad is inserted into the hashmap which resides in GPU memory. In the case of large environments, the tree is being created, a code is assigned to each quad and the code is equal to the code based on its relative location to its parent, appended to the parent’s code. Quadcodes are an efficient method for computing unique identifiers for quads using simple arithmetic operations in an independent fashion, which can be used for indexing, checking if a particular quad exists, and computing neighbors in parallel. For more details on how quadcodes are computed, please refer to [11], [12].

Alg. 1 computesPlan(mcpu)

\[
\begin{align*}
    m_r & \leftarrow m_{cpu} \\
    m_w & \leftarrow m_{cpu} \\
    \text{repeat} \\
    \quad \text{flag} & \leftarrow 0 \\
    \quad \text{plannerKernel}(\text{quads, neighbors, } m_r, m_w, \text{ flag}) \\
    \quad \text{swap} (m_r, m_w) \\
    \text{until} (\text{flag} = 0) \\
    m_{cpu} & \leftarrow m_r
\end{align*}
\]

Wavefront Expansion

Algorithm 2 describes the plannerKernel. The planner initializes the cost of every quad \( q \) to a default value, \( g(q) = -1 \), indicating that it needs to be updated. Quads containing obstacles \( o \) take a transition value relative to the total area occupied by obstacles:

\[
g(q) = \max_{\text{cost}(q)} \sum_{o \in q} \frac{\text{area}(o)}{\text{area}(q)}
\]

where a value of \( \max_{\text{cost}(q)} \) indicates that the entire quad is occupied by an obstacle. The goal quad is initialized with a value of 0, \( g(\text{goal}) = 0 \). The planner finds the value \( g \) of reaching any quad \( q \) from the goal by launching a kernel at each iteration that computes \( g(q) \) as follows:

\[
g(q) = \min_{q' \in \text{successor}(q) \land g(q') \geq 0} (g(q') + c(q, q') \cdot \text{cost}(q))
\]

for each successor \( q' \in \text{successor}(q) \) that has been updated (i.e. \( g \geq 0 \)), where \( c(q, q') \) is the Euclidean distance between the centers of \( q \) and \( q' \), and \( \text{cost}(q) \) is the cost of traversing \( q \). This process continues until the minimum \( g \)-value expanded

VII. DYNAMIC SEARCH ON THE GPU FOR
ADAPTIVE RESOLUTION GRIDS

The same underlying approach used with uniform grids [1] is applied here, where individual states represent quads of varying sizes instead of a grid cell. The propagation of the wave accounts for the cost of transitioning the quad as well as its area. At the end of the propagation process, the minimal cost of reaching any quad center from the goal is computed which is used to efficiently trace paths to the goal from any position in the environment.

A. Main Kernel

The main kernel creates two copies of the entire map of cost structs in device memory, one is used as read-only and the other one as write-only to avoid synchronization issues when running the kernel on multiple threads. It also creates a copy in device memory of all quad structures to access the data that is common to all cells regardless of which goal is associated with them. At the end of the kernel execution, these maps are swapped and the kernel is run again until the termination condition is satisfied and the costs converge to their optimal values.

Algorithm 1 computesPlan(mcpu)

\[
\begin{align*}
    m_r & \leftarrow m_{cpu} \\
    m_w & \leftarrow m_{cpu} \\
    \text{repeat} \\
    \quad \text{flag} & \leftarrow 0 \\
    \quad \text{plannerKernel}(\text{quads, neighbors, } m_r, m_w, \text{ flag}) \\
    \quad \text{swap} (m_r, m_w) \\
    \text{until} (\text{flag} = 0) \\
    m_{cpu} & \leftarrow m_r
\end{align*}
\]

Wavefront Expansion

Algorithm 2 describes the plannerKernel. The planner initializes the cost of every quad \( q \) to a default value, \( g(q) = -1 \), indicating that it needs to be updated. Quads containing obstacles \( o \) take a transition value relative to the total area occupied by obstacles:

\[
g(q) = \max_{\text{cost}(q)} \sum_{o \in q} \frac{\text{area}(o)}{\text{area}(q)}
\]

where a value of \( \max_{\text{cost}(q)} \) indicates that the entire quad is occupied by an obstacle. The goal quad is initialized with a value of 0, \( g(\text{goal}) = 0 \). The planner finds the value \( g \) of reaching any quad \( q \) from the goal by launching a kernel at each iteration that computes \( g(q) \) as follows:

\[
g(q) = \min_{q' \in \text{successor}(q) \land g(q') \geq 0} (g(q') + c(q, q') \cdot \text{cost}(q))
\]

for each successor \( q' \in \text{successor}(q) \) that has been updated (i.e. \( g \geq 0 \)), where \( c(q, q') \) is the Euclidean distance between the centers of \( q \) and \( q' \), and \( \text{cost}(q) \) is the cost of traversing \( q \). This process continues until the minimum \( g \)-value expanded

![Fig. 4. Quadcode computation for dynamic quads. The black obstacles are divided into children quads at each iteration.](image)
Once the kernel is done executing, we swap $m$ quad from $m_r$ and $m_w$, for efficient neighbor access. Each thread in the kernel reads and an array containing all the precomputed neighbor indices structs to access quad information that is goal independent, applications. We also pass to the kernel the array of quad to deal with concurrency issues that are common in parallel applications. It is not necessary to run the Algorithm 2 plannerKernel($quads$, $neighbors$, $m_r$, $m_w$, flag)

1. $q \leftarrow \text{threadQuad}$
2. if $q \neq \text{INVALID} \land q \neq \text{goal}$ then
3. for all $q' \in \text{neighbor}(q)$ do
4. if index($q'$) $\geq 0$ then
5. newg $\leftarrow g(q') + c(q, q') \cdot \text{cost}(q')$
6. if (newg $< g(q) \lor g(q) = -1) \land g(q') > -1$
7. then
8. pred_code($q$) $\leftarrow \text{code}(q')$
9. $g(q) \leftarrow \text{newg}$
10. if $g(s) < g(\text{start}) \lor g(\text{agent}) = -1$ then
11. flag = 1

Minimal Map Convergence. It is not necessary to run the planner until the entire map costs converge to their optimal values. At every planner iteration, we keep track of the minimum g-value expanded for any quad. If at a given iteration the minimum g-value is larger than the g-value of the agent quad and the agent quad has been expanded, then we can terminate the search and follow the least cost path to retrieve the solution. This ensures that only those quad costs are considered which are required to compute the least cost path from the goal to all agent positions. This exit condition is specified in Lines 9,10 of Algorithm 2 which ensures that the planner terminates execution once the two previously mentioned conditions have been met. The error of the plan found by using an adaptive resolution environment scheme depends on the size of the largest quad found as part of the solution.

B. Plan Repair

Quickly repairing the quad-tree and updating the GPU hashmap is imperative to handle dynamic environments. We locally repair the tree as follows: when an obstacle moves, we find the quads the obstacle previously occupied and examine its parent. If the parent contains an obstacle we finish traversing the tree, otherwise, we remove all children for that parent and recursively repeat the process for its parent. Then, we find the quads to which the obstacle has moved, and construct the sub-trees corresponding to each of those quads. As we repair the tree, we keep track of which quads have been added, removed, and the ones that need to be updated. A quad may need to be updated when an obstacle movement invalidates its current g-value, requiring its cost to be recomputed. The list of quads that have been inserted, removed, and need to be updated, are processed on the GPU as follows:

1) For each quad to be updated, we compute its quad-code to find its index by querying the hashmap and invalidate its cost values by setting them to $-1$, which mandates a recomputation the next time the planner kernel is launched.

2) A kernel is launched to account for all removed quads which retrieves their indices and invalidates them in the hash map, as well as its corresponding g-values in the cost map. Additionally, all references to invalidated quads in the precomputed neighbor list is detected and removed.

3) We insert the new quads into our quads array and cost map. We recycle memory that is occupied by invalidated quads that were previously removed, or append the newly created quads at the end of the list. We also keep track of the locations where these new quads were inserted and launch a kernel that updates the hashmap with the indices and quadcodes for each quad.

4) A kernel is launched (Algorithm 3) which computes the neighbors of each inserted quad, and updates the neighbors of surrounding quads, if needed.

Algorithm 3 UpdateNeighborsKernel(inserted, hashmap)

$q \leftarrow \text{threadQuad}$

computeNeighbors($q$)

for all $q' \in \text{neighbors}(q)$ do

if code($q'$) $> 0$ then

index $\leftarrow \text{hashmap}[\text{code}(q')]$

if index $\geq 0$ $\land$ $q' \notin \text{inserted}$ then

computeNeighbors($q'$)

An obstacle movement could leave a quad $q$ in an inconsistent state. A quad is considered inconsistent when $g(q) \neq g(q') + c(q, q') \cdot \text{cost}(q)$

Repairing Inconsistent States. After updating the neighbors, we fix any inconsistency in the state space and propagate this repair until there are no inconsistent states left, as described in Algorithm 4. The algorithm runs in a loop which sets the flag propagateUpdate to 0 before running the kernel. The loop is repeatedly executed until no changes are made in the most recent kernel execution (i.e. the flag is not modified). The kernel checks if the g-value of a quad is inconsistent or its predecessor had been invalidated. If so, the quad g-value is set to $-1$ and its pred_code is set to $0$, which marks it for update the next time the planner kernel is launched,
We tested our planner on several challenging navigation benchmarks to demonstrate the benefits of our proposed method over a uniform grid. Tests where performed on different environments sizes: $128 \times 128, 256 \times 256, 512 \times 512, 1024 \times 1024$ and $2048 \times 2048$. All sizes are given in units, where a unit maps to a single grid cell in a uniform grid. Figure 5 shows the number of states required to represent the state space using a quad-tree of depth 7 vs. a uniform grid. On a $128 \times 128$ world map, the uniform grid needs 16,384 states to represent the map while a quad-tree representation only requires 1,303 states. This difference keeps increasing with larger sizes. We can observe an enormous difference for a map of size $2,048 \times 2,048$, where a uniform grid requires 4,194,304 states while only 21,312 is required for a quad-tree. It is worth noting that the number of states for a uniform grid depends directly on the size of the environment, while the number of quads in the quad-tree depends on the distribution of obstacles in the map.

Figure 6 compares GPU memory usage for different world sizes using different tree depth. We can observe that for smaller world sizes, our approach requires some extra memory compared to the uniform grid. This is due to the fact that we maintain a precomputed list of neighbors and a hashmap in GPU memory that is not required for the uniform grid. Starting at a grid size of $512 \times 512$ we observe the benefits of representing the state space with quad-trees. For quad-trees with a maximum depth of 5 and 7, the final plan is not as accurate for large worlds because of lack of resolution. For a quad-tree with depth 9, the resulting plan resembles the result of the uniform grid while utilizing significantly fewer resources.

The graph shown in figure 7 compares the running time between quad-trees and uniform grid for computing an initial plan and performing several repairs due to dynamic changes in the environment. The adaptive environment representation outperforms the uniform grid with significant performance gains achieved during initial plan computation. However, it could be the case that an obstacle movement does not affect a plan in a uniform grid, but does affect it for quad-trees. If a change forces a quad that was part of the solution to subdivide, this will require some effort to repair the plan, but the same change could have no effect on a solution for uniform grid. In the last repair comparison we can observe the grid outperforming the quad-tree because of this situation.

Table I compares the running time of the different test scenarios for the uniform and adaptive grid. The last column shows the running time of the quadtree as a percentage of...
Fig. 6. GPU memory used for different world sizes. We compare results between a uniform grid and a quad-tree of depths 5, 7 and 9.

Fig. 7. Comparison of running times between uniform grid and quad-trees of the initial plan and several repairs in a map of size $512 \times 512$ units.

TABLE I

<table>
<thead>
<tr>
<th>World Size</th>
<th>Uniform Grid</th>
<th>Quad Tree - depth 7</th>
<th>Reduction as %</th>
</tr>
</thead>
<tbody>
<tr>
<td>$128 \times 128$</td>
<td>0.9</td>
<td>0.012</td>
<td>1.3</td>
</tr>
<tr>
<td>$256 \times 256$</td>
<td>1.25</td>
<td>0.04</td>
<td>3.2</td>
</tr>
<tr>
<td>$512 \times 512$</td>
<td>10.8</td>
<td>0.187</td>
<td>1.73</td>
</tr>
<tr>
<td>$1024 \times 1024$</td>
<td>14.12</td>
<td>0.04</td>
<td>0.28</td>
</tr>
<tr>
<td>$2048 \times 2048$</td>
<td>196.14</td>
<td>0.234</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Fig. 9. Comparison of paths obtained using an uniform grid (a) vs. an adaptive resolution grid (b). Red dots mark each waypoint in the final plan. (a) retrieves optimal paths by searching many more states. (b) sacrifices plan quality but results in enormous memory and performance gains.

We can observe that solutions given using quads is a little over 20% longer than the optimal solutions on average, but just above 10% when we use path smoothing.

Figure 11 shows the planner running with two hundred and fifty agents distributed among 8 goals. The world size for this scenario is $1024 \times 1024$. On the machine we tested, the uniform grid was not able to handle such a large problem while the quad planner returned a solution for all agents in under 4 seconds.

X. CONCLUSION AND IMPROVEMENTS

There has been a recent influx of work that exploits graphics processing hardware for other applications including pathfinding for autonomous agents. In this work, we address several known limitations that arise when porting wavefront-based algorithms for path planning on the GPU. Our proposed solution uses an adaptive environment representation with efficient queries for quad indexing and neighbor finding, which facilitates wave propagation for path computation in a
massively parallel fashion. While previous approaches were severely limited in environment size and number of goals due to its prohibitive memory requirements, our approach can scale to handle massively large, complex, dynamic environments with thousands of agents with different targets. In addition to addressing these memory limitations, our approach gains a significant performance boost and still preserves the dynamic properties, while keeping optimality guarantees within the resolution bounds of the adaptive environment representation.

There are still further improvements that could be done to our method. We currently allocate a predefined number of neighbors for each quad in GPU memory. This can be very wasteful for quads with only a few neighbors since we have to accommodate for a maximum number of expected neighbors. A possible solution would be to dynamically adjust the array of neighbors and have each quad know about its neighbor position in the array and how many neighbors it has. This, of course, would require more computation time, providing a tradeoff between memory and computation. Another possible improvement would be to replace the hashmap and use a combination of radix sort and binary search on the GPU. Insertions and queries in a GPU hashmap result in highly divergent branches. Radix sort and binary search have been optimized for the GPU and they could potentially bring better performance for insertion and indexing.

We described our method in terms of the application to path planning, but there are many applications where this technique could be beneficial. Some examples include: (1) crowd simulation for visual effects, urban evacuation, and security applications. (2) Multi-robot motion planning. (3) Planning in high-dimensional search spaces like kinematic linkages for end-effector motion planning.

REFERENCES