GPU-based Dynamic Search on Adaptive Resolution Grids

Francisco M. García[1], Mubbasir Kapadia[2], and Norman I. Badler[2]

[1]School of Computer Science, University of Massachusetts - Amherst
[2]Computer 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 ($> 2048\ m^2$) 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 $g(s) = g(s') + c(s, s') \times cost(s)$, 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 $cost(s)$ 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 heirarchical 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

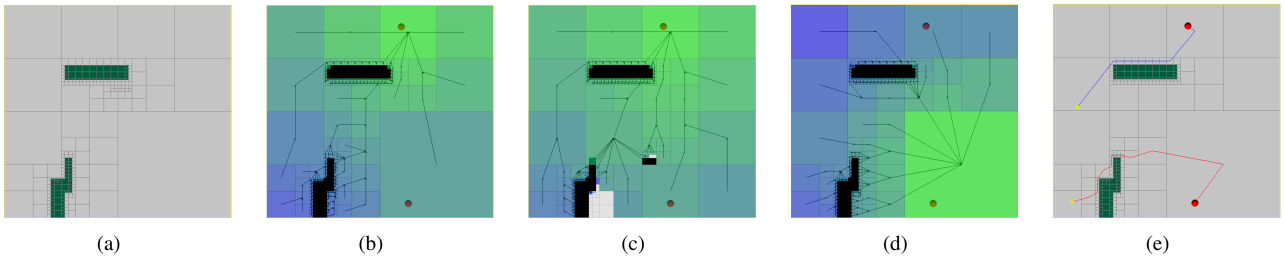
On the CPU, the environment is adaptively subdivided into a hierarchical quad tree representation, which is ported onto the GPU to compute an initial plan. Environment changes are monitored on the CPU which triggers local repairs in the quad tree. GPU memory is updated to reflect environment changes,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

A heirarchical representation of the environment for GPU computations presents several challenges. Quad tree traversal is inherently a recursive operation an 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 $\mathbb{Q}$, the number of quad divisions n_q, and the number of goals n_g to the GPU by calling $\mathbf{generateMap}(\mathbb{Q}, n_q, n_g)$and $\mathbf{createHashMap}(\mathbb{Q}, n_q)$.


(a)

(b)

(c)

(d)

(e)

Fig. 1. Method overview. 1(a) shows an initial subdivision of the environment into quads. 1(b) shows the plan computed for the top goal (red circle) for any quad. The arrows point to the predecessor quad and the colors correspond to each quad's g-value relative to a scale where *green* = 0, *blue* = max g-value and *white* = not computed. Figure 1(c) demonstrates an obstacle movement from the previous image, and how the corresponding quads are locally repaired. In 1(d) the plan for the bottom goal is depicted. Finally, figure 1(e) shows the computed plan for all agents..

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= ⟨(x,y),cost,code,index⟩, where (x, y) is the center position, $cost$ is the cost of transitioning to this quad, $code$ is a unique quad identifier, and $index$ is the position the quad occupies in the quad list. It is worth noting the importance of $code$ and $index$. The $code$ is used as the key into the hashmap to retrieve the quad structure, while $index$ 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 $index$ in the corresponding cost array. A cost structure is defined as ⟨g,pred_code⟩, where $pred\_code$ is the code of the neighbor quad q' with minimum g-value, and g is computed as:

$$g(q) = g(q') + c(q, q') \cdot cost(q),$$

where $cost(q, q')$ is the Euclidean distance between the centers of q and q', and $cost(q)$ 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 there are goals. While propagating the wave for plan computation, we only need to access the cost structures, instead of accessing the entire quad data structure. The method **createHashMap** initializes a hash map in device memory where the quad code $code$ is the key used to index into the associated quad structure.

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.

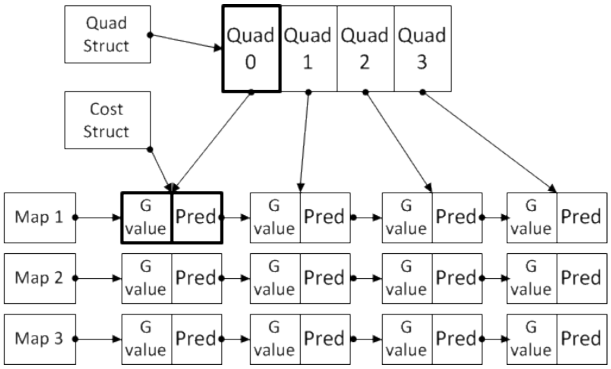


Fig. 2. Quad struct and cost struct layout. For each quad, there are as many cost structs as goals in the map.

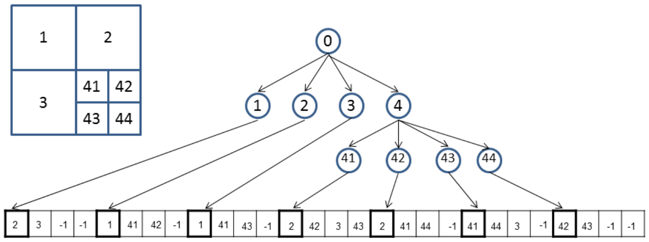
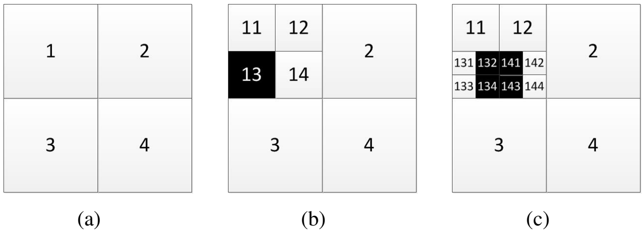


Fig. 3. Mapping neighbor array to a given quad. The index of each quad corresponds to an offset on the neighbor array. A neighbor quad can be found using the quad code in the GPU hashmap.

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 an neighbor index of -1 that means that there are no more neighbors for that particular quad.



(a) (b) (c)

Fig. 4. Quadcode computation for dynamic quads. The black obstacles subdivides the tree differently depending on its position. The numbers in each quad correspond to its specific quadcode.

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 their positions are appended to the parent code, resulting in quads 11, 12, 13 and 14. The obstacle moves again in 4(c) and now intersects quads 13 and 14, requiring them to subdivide into four more quads each.

At the end of the process, a reference for each quad is inserted into the hashmap which resides in GPU memory. The quad code serves as the key and the index in the host memory array is the value. The quads themselves reside in an array in CPU memory. In the case of large environments, a given quad can contain multiple obstacles since it cannot be divided further after a maximum depth is reached. In these scenarios, rather than considering a quad as either an obstacle or a free space, we compute its cost as the relative area that is occupied by obstacles, producing high g-values for highly populated quads. This allows the planner to favor a path through empty quads, but we allow an agent to go through a quad containing obstacles if needed. If a quad area is occupied by obstacles more than a given threshold (we used 90%), the entire quad is deemed untraversable.

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 computePlan(m_{cpu})

```
$m_r \leftarrow m_{cpu}$
$m_w \leftarrow m_{cpu}$
repeat
    $flag \leftarrow 0$
    plannerKernel($quads, neighbors, m_r,\ m_w,\ flag$)
    swap ($m_r$,$m_w$)
until ($flag = 0$)
$m_{cpu} \leftarrow m_r$
```

Wavefront Expansion

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

$$cost(q) = \texttt{MAX\_COST} \cdot \frac{\sum\limits_{\forall o \in q} \mathbf{area}(o)}{\mathbf{area}(q)}$$

where a value of MAX_COST 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 $g(\text{q})$ as follows:

$$g(q) = \min_{q' \in succ(q) \wedge g(q') \geq 0} (g(q') + c(q, q') \cdot cost(q))$$

for each successor $q' \in succ(q)$ that has been updated (i.e. $g \geq 0$), where $cost(q, q')$ is the Euclidean distance between the centers of q and q', and $cost(q)$ is the cost of traversing q. This process continues until the minimum g-value expanded in an iteration is larger than the g-value found for the agent quad.

We utilize two copies of the map representation (arrays of cost structs): a write-only map m_w and a read-only map m_r, to deal with concurrency issues that are common in parallel applications. We also pass to the kernel the array of quad structs to access quad information that is goal independent, and an array containing all the precomputed neighbor indices for efficient neighbor access. Each thread in the kernel reads the necessary values to calculate the cost of its corresponding quad from m_r, and writes it to its corresponding quad in m_w. This ensures that the data we read is always consistent. Once the kernel is done executing, we swap m_r and m_w, thus allowing the threads to read the most recent map while preventing race conditions.

Algorithm 2 **plannerKernel**($quads$, $neighbors$, m_r, m_w, $flag$)

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

```
$q \leftarrow threadQuad$
$computeNeighbors(q)$
for all $q' \in neighbors(q)$ do
    if code$(q') > 0$ then
        $index \leftarrow hashmap$[code$(q')$]
        if $index \geq 0 \land q' \notin inserted$ then
            $computeNeighbors(q')$
```

An obstacle movement could leave a quad q in an inconsistent state. A quad is considered inconsistent when

$$g(q) \neq g(q') + c(q, q') \cdot cost(q)$$

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,

Algorithm 4 **UpdateAfterObstacleMove**($quads$, $hashmap$, $propagateUpdate$)

```
$q \leftarrow threadQuad$
**if** pred_code$(q) > 0$ **then**
  $predIndex \leftarrow hashmap[\text{pred\_code}(q)]$
  **if** $predIndex < 0 \vee g(q) \neq g(q') + c(q, q') \times cost(q)$ **then**
    $g(q) = -1$
    pred_code$(g) = 0$
    $propagateUpdate = 1$
```

VIII. MULTI-AGENT AND MULTI-GOAL PLANNING

The proposed planner is able to handle multiple agents, similar to [1], while scaling to handle much larger environments, as well as multiple goals. We use RVO2 [14] for local collision avoidance and interleave planning and execution on background threads.

We create a copy of the environment cost map for each goal, but since this structure has a very small memory footprint, we can handle many goals as well as much larger environments. The running time which depends on the number of GPU iterations required for map convergence is greatly reduced as the number of states linking the goal from the farthest agent is much smaller in an adaptive environment representation. Each agent is also able to change its target to any other preexisting goal configuration without any additional computation, by simply assigning the agent to the cost map associated with that particular goal. This is particularly useful for simulating large crowds of agents that travel between predefined locations in the environment.

Path Smoothing. Our current implementation of the system searches the neighbors of a given quad in the four cardinal directions. The plan quality can be improved by considering diagonal directions as well. To improve path smoothness, we perform a raycast from the agents current position to its next set of waypoints along the path to check for the farthest waypoint that does not have an obstacle along a straight-line path. This allows us to disregard waypoints along the path which would otherwise produce jagged agent movement. There exist many other smoothing techniques which can be employed to further improve the quality of the simulations.

IX. RESULTS

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: $128 \times 128, 256 \times 256, 512 \times 512, 1024 \times 1,024$ and $2,048 \times 2,048$. 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 128×128 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.

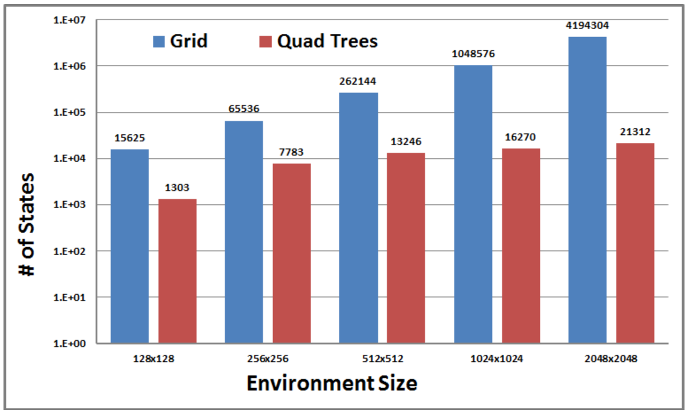


Fig. 5. Comparison of number of states required to represent the same map using a uniform and adaptive grid. For a world of 128×128, an adaptive grid need about 10 times fewer states, while for a world of 2048×2048 it utilizes about 200 times fewer states.

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 512×512 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

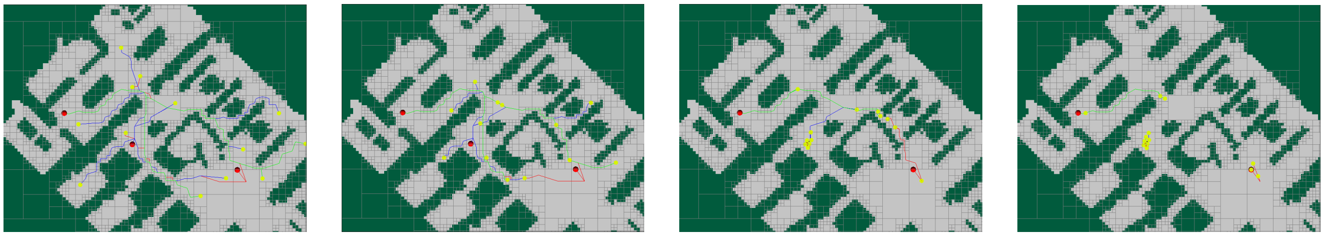


Fig. 8. Several frames of a multi-agent simulation in a game environment. The environment is not axis-aligned which is handled by the quad subdivision of the environment.

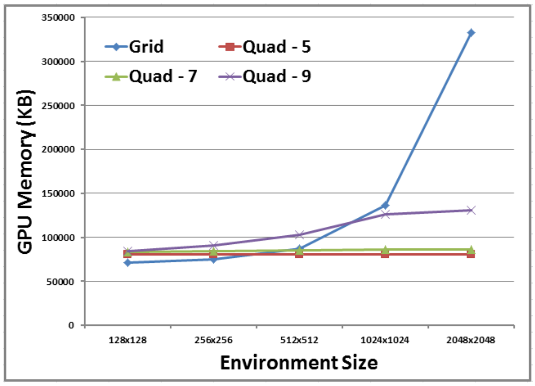


Fig. 6. GPU memory used for different world sizes. We compare results between a uniform grid and a quad-tree of depths 5, 7 and 9.

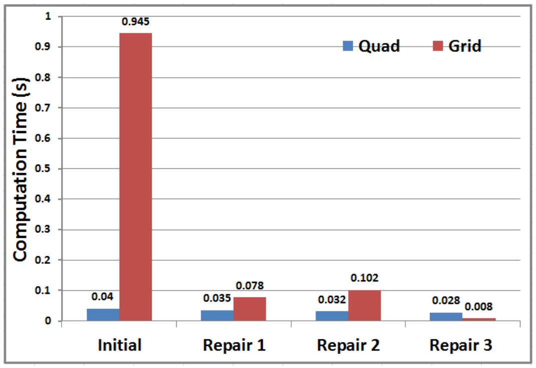


Fig. 7. Comparison of running times between uniform grid and quad-trees of the initial plan and several repairs in a map of size 512 × 512 units.

the uniform grid. From the data we can observe that the world size is not the major determinant of the required running time for quad trees, and instead mostly depends on obstacle distribution. We observe a significant performance boost when using an adaptive grid, with a 1000X speedup for a world size of 2048 × 2048. In the worst case our proposed method will perform as poorly as a uniform grid, but this scenario rarely happens in practice. Figure 9 compares the solution obtained using a uniform and adaptive grid. Although, plan quality is sacrificed by using a coarser environment representation, the performance benefits far outweigh the reduction in quality, which is reasonable for real-time applications. Fig. 10 compares the path lengths of the adaptive grid in various scenarios with the uniform grid.

TABLE I

ALGORITHM PERFORMANCE FOR DIFFERENT ENVIRONMENTS. (TIME IN SECONDS)

World Size	**GPU**		
	Uniform Grid	Quad Tree - depth 7	Reduction as %
128 × 128	0.9	0.012	1.3
256 × 256	1.25	0.04	3.2
512 × 512	10.8	0.187	1.73
1024 × 1024	14.12	0.04	0.28
2048 × 2048	196.14	0.234	0.11

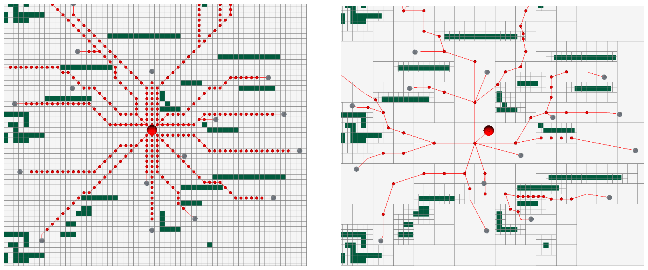


Fig. 9. Comparison of paths obtained using an uniform grid (a) vs. an adaptive resolution grid (b). Red dots mark each waypoint in the final plan. (a) retrieves optimal paths by searching many more states. (b) sacrifices plan quality but results in enormous memory and performance gains.

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 × 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

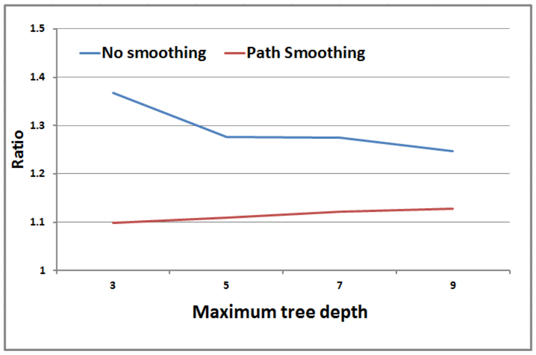


Fig. 10. Comparison of path lengths in adaptive resolution with and without path smoothing vs. uniform grid. The figure shows the ratio of the average path lengths for all agents in various test cases with the average path lengths for the uniform grid.

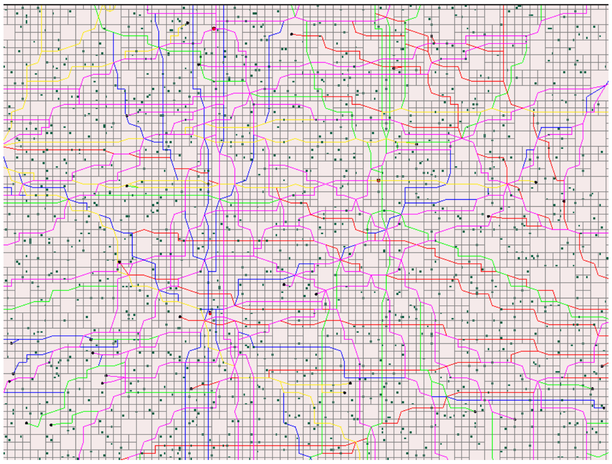


Fig. 11. Computed plans for 700 agents traveling to different goals in a 1024 × 1024 environment. Lines show the computed plan and line color corresponds to different destinations.

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

[1] Mubbasir Kapadia, Francisco Garcia, Cory D. Boatright, and Norman I. Badler. Dynamic search on the gpu. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, IROS '13. IEEE, 2013.

[2] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic a*: An anytime, replanning algorithm. In *ICAPS*, 2005.

[3] Curt Powley, Chris Ferguson, and Richard E. Korf. Depth-first heuristic search on a simd machine. *Artif. Intell.*, 60(2):199–242, 1993.

[4] Chris Ferguson and Richard E. Korf. Distributed tree search and its application to alpha-beta pruning. In *AAAI*, pages 128–132, 1988.

[5] Adolfy Hoisie, Olaf M. Lubeck, and Harvey J. Wasserman. Performance analysis of wavefront algorithms on very-large scale distributed systems. In *Wide Area Networks and High Performance Computing*, pages 171–187, 1998.

[6] Anshika Pal, Ritu Tiwari, and Anupam Shukla. A focused wave front algorithm for mobile robot path planning. In *HAIS (1)*, pages 190–197, 2011.

[7] Giuseppe Caggianese and Ugo Erra. Gpu accelerated multi-agent path planning based on grid space decomposition. *Procedia CS*, 9:1847–1856, 2012.

[8] Stephen J. Guy, Jatin Chhugani, Changkyu Kim, Nadathur Satish, Ming C. Lin, Dinesh Manocha, and Pradeep Dubey. Clearpath: Highly parallel collision avoidance for multi-agent simulation. In *ACM SIGGRAPH/EUROGRAPHICS SCA*, pages 177–187, 2009.

[9] Mubbasir Kapadia, Shawn Singh, William Hewlett, Glenn Reinman, and Petros Faloutsos. Parallelized egocentric fields for autonomous navigation. *The Visual Computer*, 28(12):1209–1227, 2012.

[10] Michael T. Goodrich and Jonathan Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *In Proc. 21st ACM Symposium on Computational Geometry*, pages 296–305. ACM, 2005.

[11] Shu-Xiang Li and Murray H. Loew. Adjacency detection using quadcodes. *Commun. ACM*, 30(7):627–631, July 1987.

[12] Shu-Xiang Li and Murray H. Loew. The quadcode and its arithmetic. *Commun. ACM*, 30(7):621–626, July 1987.

[13] Pascal J. Frey and Loc MARECHAL. Fast adaptive quadtree mesh generation. In *in: Proceedings of the Seventh International Meshing Roundtable*, pages 211–224, 1998.

[14] Jur Berg, StephenJ. Guy, Ming Lin, and Dinesh Manocha. Reciprocal n-body collision avoidance. In Cdric Pradalier, Roland Siegwart, and Gerhard Hirzinger, editors, *Robotics Research*, volume 70 of *Springer Tracts in Advanced Robotics*, pages 3–19. Springer Berlin Heidelberg, 2011.