Dynamic Search on the GPU

Abstract— **Path finding is a fundamental, yet computationally expensive problem in robotics navigation. Often times, it is necessary to sacrifice optimality to find a feasible plan given a time constraint due to the search complexity. Dynamic environments may further invalidate current computed plans, requiring an efficient planning strategy that can repair existing solutions. This paper presents a massively parallelizable wavefront-based approach to path planning, running on the GPU, that can efficiently repair plans to accomodate world changes and start movement, without having to restart the wavefront propagation process. In addition, we introduce a termination condition which ensures minimum number of GPU iterations while maintaining strict optimality constraints on search graphs with non-uniform costs.**

I. INTRODUCTION

Pathfinding is a fundamental problem in robot navigation, with a large variety of proposed approaches [1] that balance computational performance, problem domain complexity, and plan optimality. Graph-based search methods such as A* [2] provide strict optimality guarantees but cannot handle dynanmically changing environments. Real-time planners [3] have been proposed which provide anytime solution guarantees, and can efficiently repair existing plans to accomodate world changes and agent movement. However, these approaches are difficult to parallelize. Parallel search algorithms [4], [5] exploit multiple computer resources to greatly reduce compuational cost, but sacrifice optimality guarantees. Additionally, they have no mechanism to efficiently handle world changes and agent movement.

This paper presents a massively parallelizable, wavefront-based approach to path planning that can exploit graphics hardware to considerably reduce the computational load, while still maintaining strict optimality guarantees, and performing efficient updates to accomodate world changes and agent movement, while reusing previous computations. We introduce a termination condition which ensures that the plans returned are strictly optimal, even on search graphs with non-uniform costs, while requiring minimum GPU iterations. Furthermore, the computational complexity of our approach is independent of the number of agents, facilitating optimal, dynamic path planning for a large number of agents in complex dynamic environments, opening the possibiity to large-scale crowd applications. This paper makes the following contributions:

- A wave-front based search technique that can efficiently handle world changes and agent movement, while reusing previous efforts, and is amenable to massive parallelization.
- A termination condition which enforces strict optimality guarantees, even for non-uniform search graphs, while requiring minimum number of GPU iterations.
- Extension to handle any number of moving agents, at no additional computational cost.

To our knowledge, this is the first massively parallelizable, dynamic search technique with strict optimality guarantees for non-uniform search graphs.

II. RELATED WORK

There has been considerable amount of research in robot motion planning [1] that provide different tradeoffs in computational cost, domain complexity, and solution optimality. Discrete search methods like A* [2] provide strict optimality guarantees but cannot efficiently handle dynamic world updates and agent movement. ARA* [6] provides anytime properties by quickly generating a sub-optimal solution while satisfying strict time constraints, and gradually converging to optimality while reusing previous plan efforts. D* Lite [7] efficiently handles world changes and agent movement, while AD* [3] is an anytime dynamic planner that satisfies strict time constraints while efficiently repairing existing solutions to accomodate dynamic events. Tree-based search algorithms such as breadth-first search techniques [8] and alpha-beta pruning [9] have been optimized to exploit multiple processors. However, these approaches are not amenable to massive parallelization using graphics hardware.

GPU accelerated path planning algorithms [5] provide tremendous performance boost, enabling the solutions of higher dimensional problem domains, but return sub-optimal paths. The work in [10] demonstrates shortest path calculations for graph based searches on the GPU. The work in [11] uses a blocked recursive elimination strategy to utilize the compuational power and memory bandwidth of the GPU. Randomized searches [12], [4] have been successfully ported to the GPU, by doing multiple short-range searches in parrallel, but provide no optimality guarantees. Distance fields can be used to solve multi-agent planning on the GPU [13]. Crowd simulation techniques [14], [15] exploit GPU hardware to accelerate local collision avoidance for crowds but does not handle global planning. Hierarchical planning approaches [16] perform map abstraction and refinement to adaptively subdivide the search space into smaller grids, each of which can be resolved in parallel.

Wavefront based algorithms [17] are amenable to parallelization and have been demonstrated in a wide variety of problem domains [18], not just navigation, yielding substantial performance benefits over serial algorithms.

Comparison to Prior Work. Our work provides the benefits of both dynamic search techniques [3] and wavefront style algorithms [17] to provide a search technique which is massively parallelizable and can efficiently update search efforts to accomodate dynamic world changes and agent movement.

III. METHOD OVERVIEW

Our method relies on appropriate data transfer between the CPU and GPU at specific times. In the initial setup, the CPU calls $generateMap(rows, columns)$ which allocates $rows \times columns$ states in the GPU to represent the entire world. Initially, all free states s have an associated cost of -1, $g(s) = -1$, which represents a state that needs to be updated, while obstacles have infinite cost. Givent an environment configuration with start and goal state(s), ***computePlan*** is executed which repeatedly invokes ***plannerKernel*** (a GPU operation) until a solution is achieved. We keep two copies of the world map and use one to read state costs and the other to write updated states costs to. After each iteration (i.e. kernel execution), the two maps are swapped. This strategy addresses the synchronization issues inherent in GPU programming, by ensuring that the main kernel does not write to the map we use for read operations.

Once the planner is done executing, each agent can just follow the least cost path form the goal to its own state to find the generated plan. If an obstacle moves from state s to state s', we update the GPU map by setting $g(s') = \infty$ and $g(s) = -1$. This means that s' is now an obstacle and the cost for s is invalid and needs to be updated. In addition, we check the neighbors of s' and mark them as inconsistent if they had s' as their least cost predecessor. The planner kernel monitors states that are marked as inconsistent and efficiently computes their updated costs (while also propagating inconsistency) without the need for resetting the entire map. Agent movement (change in start) is also efficiently handled by performing the search in a backward fashion from the goal to the start, and marking the previous state as inconsistent to ensure a plan update. Algorithm 1 provides the pseudocode for ***computePlan***.

IV. GPU BASED WAVEFRONT ALGORITHM

Existing graph based search [2] guarantee optimality and work well for dynamic environments [3]. However, they are not amenable to massive parallelization. The wavefront algorithm [17] takes its name as an analogy of the way it behaves. It sets up a map with a initial state which contains an initial cost. At each iteration, every state at the frontier is expanded computing its cost relative to its predecessors cost. This process repeats until the cost for every state is computed, thus creating the effect of a wave spreading across the map. Wavefront based approaches are inherently parallelizable, but existing techniques require the entire map to be recomputed to handle dynamic world changes and agent movement. Figure 1 visualizes the wavefront propagation process in a simple environment.

Algorithm 1 ***computePlan***$(*m_{cpu})$

```
$m_r \leftarrow m_{cpu}$
$m_w \leftarrow m_{cpu}$
repeat
    $flag \leftarrow 0$
    plannerKernel($m_r$, $m_w$, $flag$)
    swap ($m_r$,$m_w$)
    $incons \leftarrow false$
    for all $s$ in $m_r$ do
        if $incons(s) = true$ then
            $incons \leftarrow true$
            break
        end if
    end for
until $(flag = 0 \wedge incons = false)$
$m_{cpu} \leftarrow m_r$
```

Our Approach. Algorithm 2 describes the shortest path wavefront algorithm ported to the GPU. The planner first initializes the cost of every traversable state to a default value, $g(s) = -1$, indicating it needs to be updated. States occupied by obstacles take a value of infinity, $g(s) = \infty$, and the goal state is initialized with a value of 0, $g(s) = 0$. The planner finds the value g of reaching any state s from the $goal$ by launching a kernel at each iteration that computes $g(s)$ as follows:

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

for each successor that has been updated (i.e. $g \geq 0$). This process continues until all states have been updated at which point the planner terminates execution. To address the concurrency problem inherent in a massive parallel application, we created two copies of the maps in device memory, one used as write-only and the other one as read-only. Let us refer to these maps as m_w and m_r, for write-only and read-only operations respectively. Each thread in the kernel reads the necessary values to calculate the cost of its corresponding state from m_r, and writes it to its given state in m_w. This ensures that the map we are reading from will not change as we are executing the kernel. Once the kernel finishes execution, we swap m_r and m_w, thus allowing the threads to read the most recent map while preventing race conditions.

The kernel also takes as a parameter a $flag$ which is set depending on the termination condition used:

Exit when goal reached. The flag is originally set to 1 before each kernel run. If we find that goal state was updated, that means we have a path to it and can terminate execution. We do so by marking the $flag$ as 0. This will produce considerably fewer iterations but will not guarantee optimality on search graphs with non-uniform costs. Since each thread corresponds to only one state, only one thread is able to modify this flag and no race condition is possible.

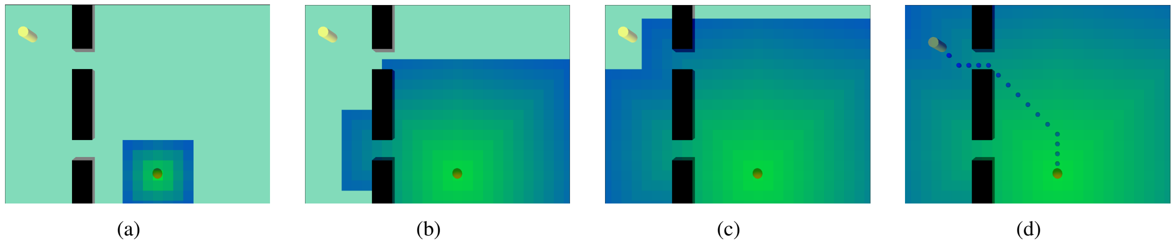
Fig. 1. Wavefront expansion process. (a) 3 iterations. (b) 11 iterations. (c) 15 iterations. (d) 18 iterations.

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

```
$s \leftarrow threadState$
if $s \neq obstacle \wedge s \neq goal$ then
    for all $s'$ in $neighbor(s)$ do
        if $s' \neq obstacle$ then
            $newg \leftarrow g(s') + c(s, s')$
            if $(newg < g(s) \vee g(s) = -1) \wedge g(s') > -1$ then
                $pred(s) \leftarrow s'$
                $g(s) \leftarrow newg$
                { evaluate_termination_condition }
            end if
        end if
    end for
end if
```

$$\textbf{if}(s == goal)\texttt{flag} = 0$$

Exit when whole map converges. An alternate exit condition is to continue propagating until the whole map has been successfully updated with accurate g values. This will guarantee optimality but with a considerable increase in the number of iterations. The flag is set to 0 before each kernel run. If there is any update in a given iteration, the flag is set to 1 thus ensuring that the planner keeps running until no further update is possible. In other words, it will terminate only when the cost computation for the entire environment has converged. This will compute costs for unnecessary parts of the environment but will guarantee optimal solutions.

$$\texttt{flag} = 1$$

Minimal Map Convergence with Optimality Guarantees. The naive approach discussed previously does much more work than it is necessary to find an optimal path. For large environments, this is prohibitively expensive. We introduce a termination condition that can greatly reduces the number of iterations required to find an optimal plan in large environments with non-uniform search graphs. If at any iteration, we find that the minimum g-value expanded corresponds to that of the agent, this means that a path to that agent is available and any other possible path would yield a higher cost. To make sure that the agent state is expanded at each iteration (to compare to the other states expanded), we give it a g-value of -1 before each kernel run, marking it as a state that needs to be updated. To implement this strategy, it is enough to just adjust the condition that would set the flag that terminates the execution:

$$\textbf{if}(g(s) < g(start) \vee g(agent) = -1) flag = 1$$

Once the planner has finished executing, an agent can simply generate its plan by following the path of least cost from the goal to its position.

V. EFFICIENT PLAN REPAIR FOR DYNAMIC ENVIRONMENTS AND MOVING AGENTS

We extend the algorithm to handle dynamic environments where obstacle changes may invalidate plans that are currently being executed and it is also able to create a new plan if an agent diverges from a previous plan. To handle obstacle movements we incorporate a flag that marks every state as either inconsistent or not. We define a state to be inconsistent if its predecessor is not the neighbor with lowest cost or if its successor is inconsistent. If an obstacle moves from state s to state s', then we set $g(s') = \infty$ and $g(s) = -1$ (marking it for update). Then, for each successor s'' of s', we mark s'' as inconsistent if s' is the predecessor of s'' and set $g(s'') = -1$, forcing it to update. In other words, if any of the neighbor states has s' as a predecessor, we mark them as inconsistent, thus mandating an update. The main kernel then propagates in the same fashion until the termination condition is satisfied and no inconsistent states along the optimal path are left. We can append the following code to the end of the ***plannerKernel*** (Algorithm 3), to guarantee that node inconsistency is propagated and resolved in the entire map. Keep in mind that the following code will run in parallel, and that all read and write operations are done in two distinct maps.

Algorithm 3 Algorithm to propagate state inconsistency

```
$state \leftarrow threadState$
$incons(s) \leftarrow false$
if $incons(pred(s)) = true$ then
    $incons(pred(s)) \leftarrow false$
    $incons(s) \leftarrow true$
    $g(s) \leftarrow -1$
end if
```

Handling agent movement is straightforward. For the non optimized planner, the cost to reach every state has already

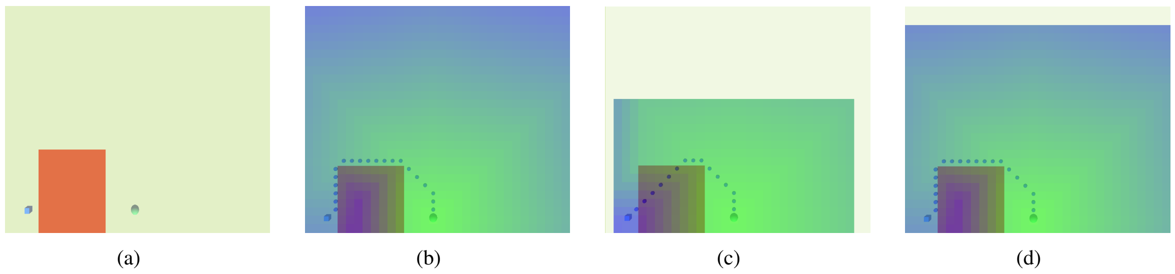

Fig. 2. Comparison of termination conditions. (a) Non-uniform state space. The states shown in red are of much higher cost as compared to other states. (b) The planner terminates after updating g values for the whole map, producing an optimal path with significantly more iterations = 17. (c) Plan termiation as soon as it finds a path to the goal, producing a sub-optimal path. Total number of iterations = 8. (d) The proposed termination condition requires minimal iterations, while providing strict optimality guarantees. Number of iterations = 12.

been computed, so the agent would only require to reconstruct its path again. In the case of the optimized version of the planner it is necessary to run the planner again so that any state between the goal and the new agent position that has not been expanded, gets a chance to update its cost.

VI. Multi-Agent Planning

Our GPU planner is able to handle multiple agents and create paths for each of them in such a way that they avoid collisions with each other. We interleave planning and execution by running the kernel every time the agents move. Each agent is able to update its own plan by following the least cost path from its position to the goal each time the planner kernel is run. We extend our planner implementation by making a slight modification to the termination condition to account for multiple agents. We execute the kernel until the agent with the largest g has a smaller g that any of the non-agent states that were expanded for that iteration, and all agent states had a chance to be reached:

$$\textbf{if}((g(s) < max_{a_i \in \{a\}} g(a_i)) \vee (g(a_i) = -1 \forall a_i \in \{a\}))$$

This means that the number of iterations it will take the planner to finish execution will depend on the distance from the goal to the farthest agent. When the map is updated, each agent simply follows the least cost path from the goal to its position to find an optimal path. It is important to note that our approach requires no additional computational cost to handle many agents.

Multi-Agent Simulation. Since our approach can efficiently plan paths for a large number of agents, and use existing plan efforts to efficiently repair plan efforts due to changes in world state, we can easily extend our approach to simulate a crowd of autonomous agents. Grid states that are currently occupied by an agent incur an additional cost, which impacts the manner in which the wave propagates through it. Each agent is simulated to move along its current path using a simple particle simulator (the path is guaranteed to avoid obstacles and agents). At each frame, the map is efficiently repaired to accommodate world changes and agent movement, thereby repairing the paths of all agents.

Multiple Target Locations. Our framework is currently limited to a small number of target locations, since a separate map needs to be maintained for each target, resulting in a significant memory overhead. Possible extensions to mitigate this issue include an adaptive quad-based environment representation, which would reduce the computation of wave propagation in large worlds, and significantly reduce the memory footprint. Efficiently porting an adaptive-quad based environment representation to the GPU is the subject for future exploration. We face a limitation regarding memory when we attempt to handle multiple agents each with its own goal. In this case, each agent would need its own representation of the environment, which means that one map per agent would be needed, requiring great amount of memory.

VII. Results

We ran our planner on several challenging navigation benchmarks [19] to showcase the benefits and limits over traditional methods using two different GPUs. Table I gives the specification of both units:

TABLE I
Graphics Processing Units Specifications

Information	GPU 1	GPU 2
Type	GeForce GTX680	Geforce GT 650M 2GB
Warp Size	32	32
Threads/Block	1024	1024
Global Mem	2147483648 Bytes	2147483648 Bytes
MultiProcessors	8	2
Mem Clock Rate	3004000 KHz	900000
Mem Bus Width	256 bits	128 bits
Chip Clock Rate	1058500 KHz	950000 KHz

Figure 3 demonstrates the scalability of our approach with increase in number of agents on a 256×256 world map. We observe that there was no noticeable increase in the computational cost with increase in number of agents.

Figure 4 illustrates the scalability of our approach to accomodate large environments. We tested it with a single agent with a goal distance of $N/2$ in a $N \times N$ world map. We observe that the use of the minimal yet sufficient exit condition (EXIT A) produces significant performance improvements over EXIT B as the planner does not have to wait

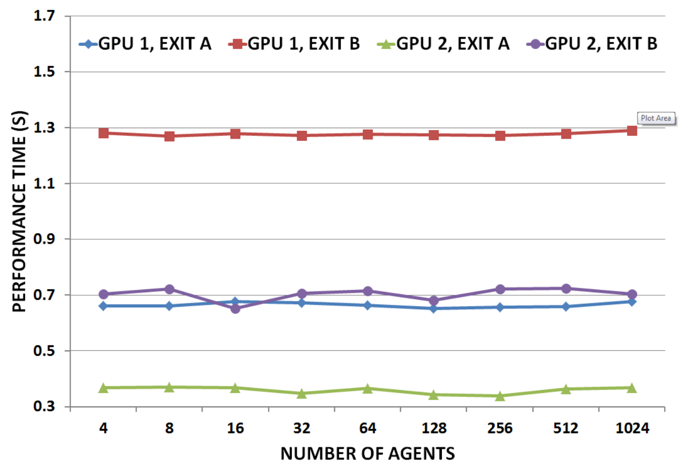

Fig. 3. GPU planner performance with increase in number of agents. `EXIT A`: Exit condition that checks for convergence of only agent states. `EXIT B`: Exit conndition which checks for convergence of whole map. All solutions returned are optimal paths. Experiment performed on a 256×256 environment. GPU memory = 5120 KB. CPU memory varies from 2048 KB to 2080 KB.

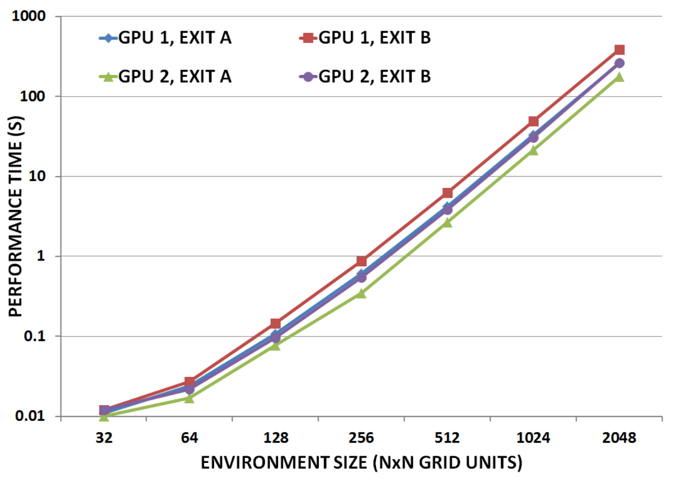

Fig. 4. The figure shows the time it took our planner to find an optimal solution for different map sizes on the average case. We can observe that on the average case, the larger the environment the larger the benefit of our planner.

until the g values of the whole map have converged, resulting in great savings. Figure 5 illustrates the performance of our approach on a variety of challenging benchmarks [19].

Figure 6 illustrates the overall advantage of using our method with a simple test scenario where we handle obstacle, agent and goal movement. We generated a random map of size 512×512 populated with 8 agents. We can observe that our method took fewer iterations to reach an optimal solution at each step, with a significant performance improvement on the initial plan and after goal movement, when the map needs to be reset and planned from scratch.

Figure 7 illustrates the memory requirements of our approach based on world size.

Figure 8 demonstrates path planning for 200 agents in a randomly generated environment of size 512×512. Since

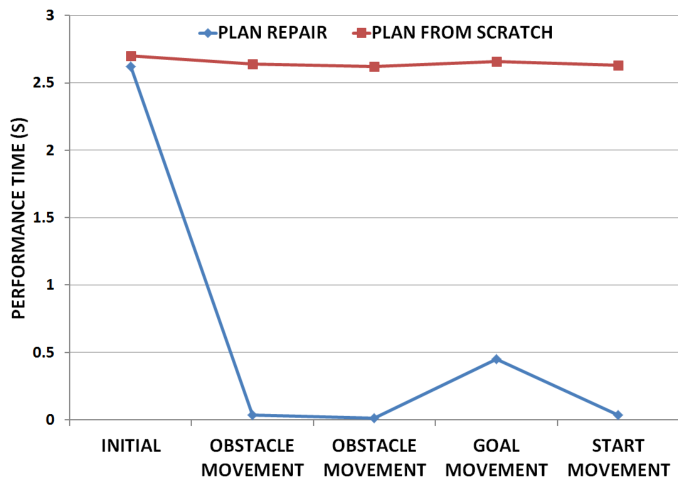

Fig. 6. GPU planner performance for dynamic simulation with changes in environment, start, and goal.

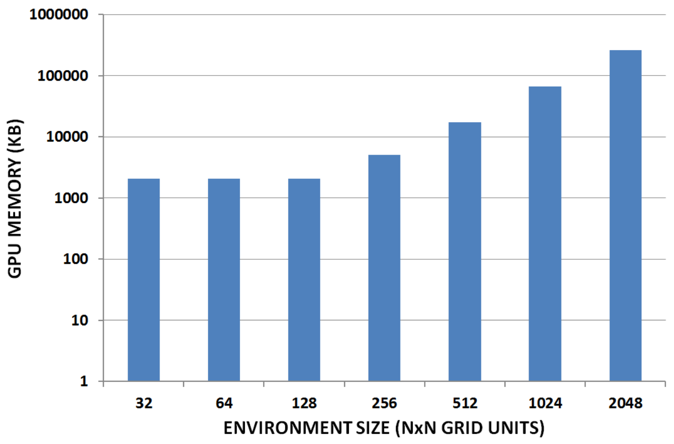

Fig. 7. Here we can observe that for world sizes over 128x128, we see an increase on the amount of memory required by the GPU. We ran the planner up to a size of 2048x2048 where the total amount of memory used in the GPU was 263.16 MB.

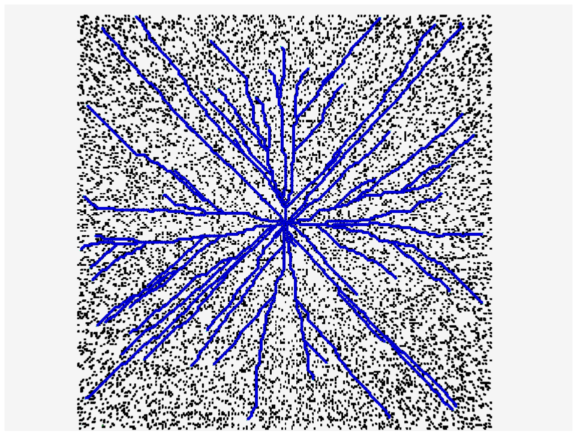

Fig. 9. The figure shows how our planner is able to handle large complex environments. In this case we show a world of 512×512 with 200 agents. The goal is in the center of the map, and the computed paths are shown in blue.

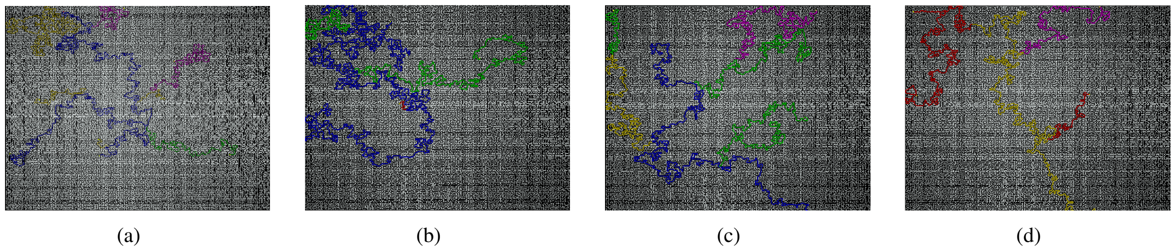

Fig. 5. Global Navigation for multiple agents on a variety of challenging benchmarks [19] of size 512×512.

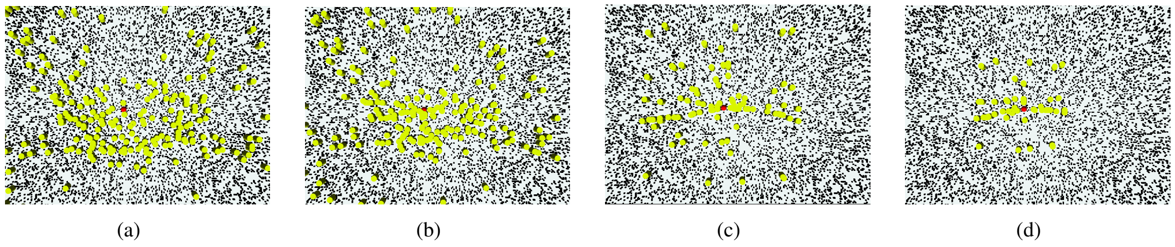

Fig. 8. Global Path Planning and Simulation of 200 agents on a complex navigation benchmark.

TABLE II
ALGORITHM PERFORMANCE FOR DIFFERENT ENVIRONMENTS. (TIME IN SECONDS)

World Size	GPU 1		GPU 2	
	Exit A	Exit B	Exit A	Exit B
32×32	0.011	0.012	0.01	0.012
64×64	0.024	0.027	0.017	0.022
128×128	0.107	0.146	0.078	0.096
256×256	0.608	0.871	0.349	0.542
512×512	4.219	6.24	2.691	3.816
1024×1024	32.931	49.126	21.246	30.778
2048×2048	258.88	387.35	178.794	264.373

our approach can efficiently handle dynamic updates, we can interleave planning with execution to create a crowd simulator.

VIII. CONCLUSION AND FUTURE WORK

We have developed a massively parallel wave-front based planning technique which can efficiently handle world changes and agent movement by reusing previous computations. The computational cost of our approach is independent of number of agents, facilitating global path planning for hundreds and thousands of agents in very large, complex, dynamic environments. Furthermore, we demonstrate a prototype crowd simulator by interleaving planning with execution where the plans are efficiently updated to accomodate agent movement.

There are some limitations to our approach. A separate map needs to be maintained for each target location, resuting in a substantial memory and computational overhead. This makes our current approach intractable for agents with their own targets. One possible approach to attenuate the impact in memory would be to use a quad-based environment representation where open spaces can be represented as a coarser grid. Porting a quad-based environment representation to the GPU is the subject of future exploration.

REFERENCES

[1] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[2] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, July.

[3] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic a*: An anytime, replanning algorithm. In *Proc. Conference on Automated Planning and Scheduling (ICAPS*, 2005.

[4] Joseph Kider, Mark Henderson, Maxim Likhachev, and Alla Safonova. High-dimensional planning on the gpu. In *ICRA*, 2010.

[5] Leonardo G. Fischer, Renato Silveira, and Luciana Nedel. Gpu accelerated path-planning for multi-agents in virtual environments. In *Proc. Brazilian Symposium on Games and Digital Entertainment*, SBGAMES, pages 101–110. IEEE Computer Society, 2009.

[6] M. Likhachev, G. Gordon, and S. Thrun. ARA*: Anytime A* search with provable bounds on sub-optimality. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Proc. Conference on Neural Information Processing Systems (NIPS)*. MIT Press, 2003.

[7] Sven Koenig and Maxim Likhachev. D* lite. In *AAAI*, pages 476–483, 2002.

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

[9] Chris Ferguson and Richard E. Korf. Distributed tree search and its application to alpha-beta pruning. In Howard E. Shrobe, Tom M. Mitchell, and Reid G. Smith, editors, *AAAI*, pages 128–132. AAAI Press / The MIT Press, 1988.

[10] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato Fonseca F. Werneck. Phast: Hardware-accelerated shortest path trees. In *IPDPS*, pages 921–931, 2011.

[11] Aydin Buluc, John R. Gilbert, and Ceren Budak. Solving Path Problems on the GPU. 36:241–253, 2010.

[12] Maria Gini. Parallel search algorithms for robot motion planning. In *In Robotics: Current Approaches and Future Directions, IEEE ICRA*, 1996.

[13] Rafael P. Torchelsen, Luiz F. Scheidegger, Guilherme N. Oliveira, Rui Bastos, and João L. D. Comba. Real-time multi-agent path planning on arbitrary surfaces. In *Proc. ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D, pages 47–54, 2010.

[14] 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 Symposium of Computer Animation, pages 177–187, 2009.

[15] Avi Bleiweiss. Scalable multi agent simulation on the gpu. In *GPU Technology Conference*, 2009.

[16] Nathan R. Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *AAAI*, pages 1392–1397, 2005.

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

[18] 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.

[19] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012.