Exploiting GPUs for Multi-Agent Path Planning on Grid Maps

Giuseppe Caggianese
Dipartimento di Ingegneria e Fisica dell'Ambiente
Università degli Studi della Basilicata
Potenza, Italy
giuseppe.caggianese@unibas.it

Ugo Erra
Dipartimento di Matematica e Informatica
Università degli Studi della Basilicata
Potenza, Italy
ugo.erra@unibas.it

Abstract—**Multi-agent path planning on grid maps is a challenging problem and has numerous real-life applications ranging from robotics to real-time strategy games and non-player characters in video games. A* is a cost-optimal forward search algorithm for path planning which scales up poorly in practice since both the search space and the branching factor grow exponentially in the number of agents. In this work, we propose an A* implementation for the Graphics Processor Units (GPUs) which uses as search space a grid map. The approach uses a search space decomposition to break down the forward search A* algorithm into parallel independently forward sub-searches. The solution offer no guarantees with respect to completeness and solution quality but exploits the computational capability of GPUs to accelerate path planning for many thousands of agents. The paper describes this implementation using the Compute Unified Device Architecture (CUDA) programming environment, and demonstrates its advantages in GPU performance compared to GPU implementation of Real-Time Adaptive A*.**

Keywords—**path-finding; GPU acceleration; A* algorithm; grid maps; search space decomposition;**

I. Introduction

Navigation-planning techniques for multi-agents have been traditionally studied in the domain of robotics and in recent years have been increasingly applied to real-time strategy games and non-player characters in video games. In these applications, the principal challenge is the safe navigation of an agent to its target location while avoiding collision with static or dynamic obstacles and other moving agents. Agents are therefore not directly controlled by humans but rely instead on path-planning algorithms.

Single-agent path planning, where the size of the search space is bounded by the map size, can be tackled with a search algorithm such as A* [4]. Multi-agent path planning is much more challenging than the single agent case, being a PSPACE-hard problem [5]. The number of states and the branching factor grow exponentially with the number of agents on a map. As the number of agents and the size of the search space increase, planning tends to become computationally burdensome or even intractable. Trading the method completeness and the solution optimality for improved performance is a typical feature of decentralized approaches, including the method described in this paper.

In the last few years, Graphics Processor Units (GPUs) have increased rapidly in popularity because they offer an opportunity to accelerate many algorithms. In particular, applications that have large numbers of parallel threads that do similar work across many data points with limited synchronization are good candidates with which to exploit GPU acceleration. All of this means that in a multi-agent scenario, we can take into account the idea that exploration of different paths can be performed in parallel with large numbers of threads that explore simultaneous subpaths. In this way, scalability to larger maps can be achieved with decentralized approaches, which decompose the initial problem into a series of A* searches. However, a path planning GPU-based must takes into account that it has limited access to the GPU and memory resources, which are allocated with higher priority to other game modules such as the graphics engine and more recently to the physics engine.

In this paper, we describe an approach for a path-planning system on grid maps for many thousands of agents that uses A*. The approach is based on the search space decomposition of the input grid map into blocks. For all blocks, we perform simultaneous A* searches to obtain all potential subpaths of the input agents toward the goal state that traverse these blocks. In this way, given the start positions of agents and a goal position as input, we are able to break down the forward search A* algorithm into parallel independently forward sub-searches. This approach fits well with the parallel architecture of GPUs, where many hundreds of threads are necessary to exploit the GPU fully. In addition, our method is simple and easy to implement using a GPU programming model such as the platform chosen in this work, NVIDIA's Compute Unified Device Architecture (CUDA). We compare our approach with Real-Time Adaptive A* (RTAA*), an algorithm for multi-agent path-planning on grid maps. The empirical results demonstrate the GPU performance-speed up advantages for large numbers of agents compared to GPU implementations of RTAA*.

The rest of the paper is organized as follow. Section II relates similar work on parallel path-planning problems. Section III describes the fundamentals of our parallel path-planning method. In Section IV, we present some details about the implementation on the GPU. In Section V, we assess performance trade-off and results on quality. Finally, in Section VI, we present our conclusions and future research directions.

II. Related Work

Multi-agent path planning using decentralized approach can significantly reduce computation by decomposing the problem into several subproblems. These approaches are faster but yields suboptimal solutions and loses the completeness. One such example is prioritized planning [2], which uses prioritization to assign an order in which the objects move. Another approach coming from game industry is Local Repair A* [11] which performs an expensive full A* for every replan. In [10], Ryan introduces an approach which is complete but restricted to specific search graphs, which can be decomposed into structures such as chains or rings.

Researchers have recently developed parallel-based implementations of path-finding that use the computational power of the GPU. These approaches provide strong evidence that GPUs can substantially accelerate path-finding algorithms, particularly for real-time applications such as video games and real-time applications in robotics, which require efficient path-finding to support large numbers of agents moving through expansive and increasingly large dynamic environments. In 2008, Bleiweiss [1] implemented the Dijkstra and the A* algorithms using CUDA. After several benchmarks, he observed that the Dijkstra implementation reached a speed up of a factor 27 compared to a C++ implementation without SSE instructions, while A* implementation reached speed up of a factor 24 compared to a C++ implementation with SSE instructions. In [6], Katz et al. present a cache-efficient GPU implementation of the all-pairs shortest-path problem and demonstrate that it results in a significant improvement in performance. In [12], Stefan et al. obtained speedups for breadth-first search using a bit-vector representation of the search frontier on a GPU. In [7], Kider et al. present a novel implementation of a randomized heuristic search, namely R* search, that scales to higher-dimensional planning problems. They demonstrate how R* can be implemented on a GPU and show that it consistently produces lower-cost solutions, scales better in terms of memory, and runs faster than R* on a Central Processing Unit (CPU). In [3], Erra et al. propose an efficient multi-agent planning approach for GPUs based on an algorithm called RTAA*. The implementation of RTAA* enables the planning of many thousands of agents by using a limited memory footprint per agent. In addition, benchmarks support the GPU CUDA performance scale compared to multi-threading CPU implementation in running one, two, and four threads.

III. The Proposed Approach

In this section, we describe an approach to compute in parallel simultaneous path planning in a multi-agent scenario. As inputs we have a set of start states, a goal state, and a search space that represents the environment in which the agents move. We first describe the reference scenario for our path planning and how we decompose the search space. We then describe the necessary steps to perform the parallel search from the start states to the goal state.

A. Search space decomposition

The approach we propose is suitable for scenarios that are based on a grid map. In these scenarios, the environment is subdivided into small regular zones called tiles. Each tile represents a state s of the search space and is connected to all nearby tiles. The cost of moving from a tile to each of its neighbors is specified by an integer. This can be used to model terrain elements that are difficult or impossible to pass, for example hills and lakes. A common metric used to measure distance on grid maps, which we adopt for this work, is the Manhattan distance.

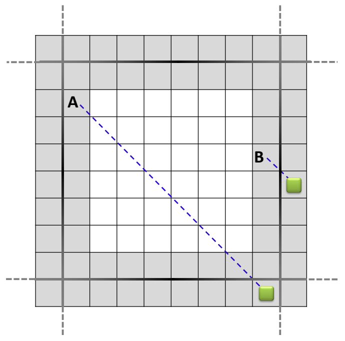
The grid map used as search space S is further divided into k regular regions B_i called *planning blocks*. Planning blocks are all of the same size and decompose the search space into non-overlapping search sub-spaces such that $S = B_1 \cup B_2 \cup \cdots \cup B_k$. We refer to the edge states of a planning block as *border tiles*, which enable a state transition from a planning block to a neighboring block, as illustrated in Fig. 1.

Given two distinct planning blocks B_i and B_j, let $p_i = \langle s_1, s_2, \ldots, s_m \rangle$ a subpath with $s_i \in B_i$ for $i = 1, \ldots, m-1$. We name p_i a *traversing subpath* of B_i if $s_1, s_m \in B_j$ are border tiles (Fig. 2a). We call p_i an *incoming subpath* of B_i if $s_m \in B_i$ and s_1 is a border tile (Fig. 2b). If $s_m \in B_i$ is a border tile we call p_i an *outgoing subpath* of B_i (Fig. 2c). Finally, if $s_m \in B_i$ and none of the states s_i is a border tile then p_i is an *internal subpath* of B_i (Fig. 2d). A path p from a start state to a goal state has subsequent subpaths p_i with $i = 1, 2, \ldots, n$. If $n = 1$, we have only an internal subpath. With $n \geq 3$, p_1 is an outgoing subpath, $p_2, p_3, \ldots, p_{n-1}$ are traversing subpaths, and p_n is an incoming subpath.

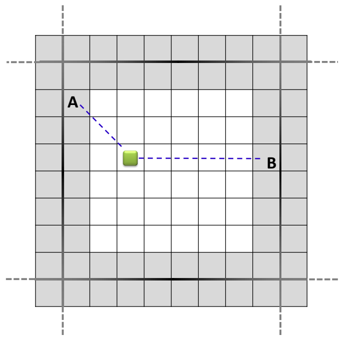
The rationale behind the planning blocks is to break down the search of a single path, computing independently all its subpaths. By using simultaneous A* searches for all planning blocks, we compute in parallel all potential subpaths. This may pose a problem because of the dependence of the planning blocks. During an A* search a path is discovered sequentially, and if a subpath p_{i-1} inside the planning block B_{i-1} then precedes a subpath p_i inside the planning block B_i, we are unable to launch simultaneous A* searches for B_{i-1} and B_i. However, in a multi-agent scenario it is natural to expect portions of a path to be shared between agents. Thus, we can take advantage of this scenario, computing in parallel all subpaths potentially able to traverse all planning blocks.

B. The parallel search

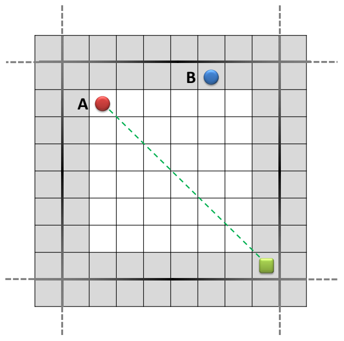
To perform searches for many thousands of agents in parallel, we need to find a way of using all planning blocks



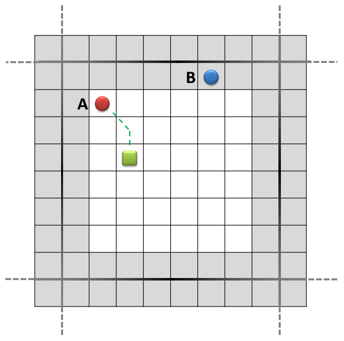
(a) Traversing subpath.



(b) Incoming subpath.



(c) Outgoing subpath.



(d) Internal subpath.

Figure 2. Left (a–b): border-to-border step. All the simultaneous A* searches start from the border tiles and terminate in a border tile of a neighboring planning block or in a goal state. Right (c–d): start-to-border step. The simultaneous A* searches start from the agent's positions and terminate on a border tile of the same planning block or in a goal state.

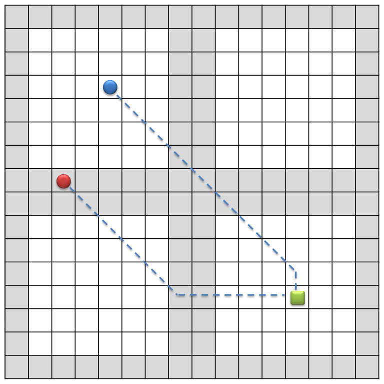


Figure 1. Example of a 16×16 grid map with four 8×8 planning blocks. Gray indicates border tiles. The circles are two agents in their start positions. The square is a goal position, and dotted lines are paths that traverse planning blocks.

simultaneously. Our solution exploits the fact that, given a set of start states and one goal state, it is likely that the discovered paths share subpaths. We attempt to determine these shared subpaths, computing in parallel all potential subpath types inside the planning blocks, taking into account that they must converge toward the goal direction. This strategy is achieved in two steps: *border-to-border* search and *start-to-border* search.

In the border-to-border step (Fig. 2a–b), we compute for all planning blocks the traversing subpaths and incoming subpaths using multiple A* searches. In particular, for a given planning block, A* searches have as start states the border tiles, and the searches determine all states along the way toward the goal position. A single A* search terminates when a border tile belonging to a neighboring planning block is discovered or when the search discovers a goal position. At the end of this step, we can assemble a path from any border tile toward the goal position, assembling a sequence of zero or more traversing paths and an incoming path. Note that this step is independent from the start positions and can be performed off-line if the goal is not expected to move anywhere.

In the start-to-border step (Fig. 2c–d), we compute the outgoing subpaths and internal subpaths using multiple A* searches. In this case, for all the input agents, A* searches have as start states the start positions of all agents, and the searches determine, as described above, all states along the way toward the goal position. A single A* search terminates when a border tile belonging to the same planning block is discovered or when the search discovers a goal position. Thus, the objective of this step is to search the paths of all agents from their start positions to the nearest border tiles.

The advantage of this approach is that it can be easily implemented in parallel because all the searches in the border-to-border step and in the start-to-border step can be performed simultaneously. We have removed the dependence between planning blocks taking into account all possible subpaths inside the planning blocks. Furthermore, the ability to use small search areas ensures faster searches and limits memory use. Finally, in the case of a change in start position, we need only to perform a start-to-border search where the change occurred.

C. Paths Reconstruction

At the end of the border-to-border and the start-to-border step, we have for each border tile its immediate predecessor in the best path found so far. This information can be used to reconstruct the subpaths of all planning blocks by working backwards from the last border tile to the initial border tile or start positions of all agents. Thus, an agent can use immediately these subpaths to move along the trajectory from its start position to the goal position without additional computation. An advantage of this approach is that when the knowledge of the agent about the search space changes or the search space itself changes along the resulting trajectory, we can replan only blocks where these changes occur with a new border-to-border and/or start-to-border step.

A problem associated with this approach is the formation of loops at the end of the border-to-border step, as illustrated in Fig. 3. There are two possible strategies to tackling this problem. The first is to handle the loop during the movements of agents, e.g., prevent the agent from becoming trapped in the loop. The second is to increment the heuristic associated with the border tiles that form the loop and then perform a new border-to-border step only for the planning blocks involved. In this way, these border tiles will not be taken into account in the new paths because of their higher movement cost.

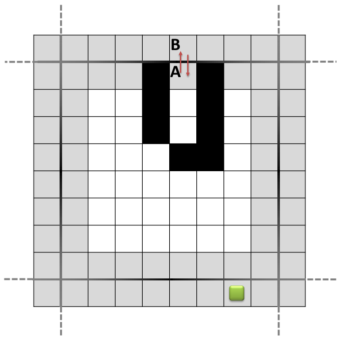


Figure 3. A dead-end causes the A* search from border tile A to stop in border tile B. Conversely, A* search from border tile B stops in border tile A.

IV. Implementation Details

NVIDIA's CUDA is the platform chosen for exploiting data parallelism in the GPU. From the point of view of programming model, CUDA is a minimal extension to C language which permits the writing of a serial program called kernel which are executed in parallel across a set of parallel threads. Further details on the GPU architecture and CUDA programming model are available in NVIDIA's CUDA Programming Guide [9].

Given the CUDA programming model, implementation of the approach described above on a GPU programming model is straightforward. To execute all searches simultaneously, we associate a single GPU thread with each search; in fact, each search executes the same instructions but with different data. In the border-to-border step, we couple a single thread block for each planning block, as illustrated in Fig. 4. This decision enables us to execute concurrently all border-to-border searches and to share information through shared memory. Indeed, heuristic values $h[s]$, used to estimate the goal distance for each state s, are stored as a single array in the large shared memory as a planning block. This is possible because in the border-to-border step all the searches are local, i.e., a search starts and stops in the same planning block. These single arrays are initialized on the CPU and copied to the GPU. Note that each thread, after A* search is also in charge of the path reconstruction. In such way, at the end of border-to-border all the subpath are copied from the GPU to the CPU.

Conversely, in the start-to-border step, agents' start positions are not all located in the same planning block. However, even in this case, the array for heuristic values can be shared between agents. In fact, the array may become as large as the entire map and must then be stored in a global memory because its dimensions are too large for shared memory.

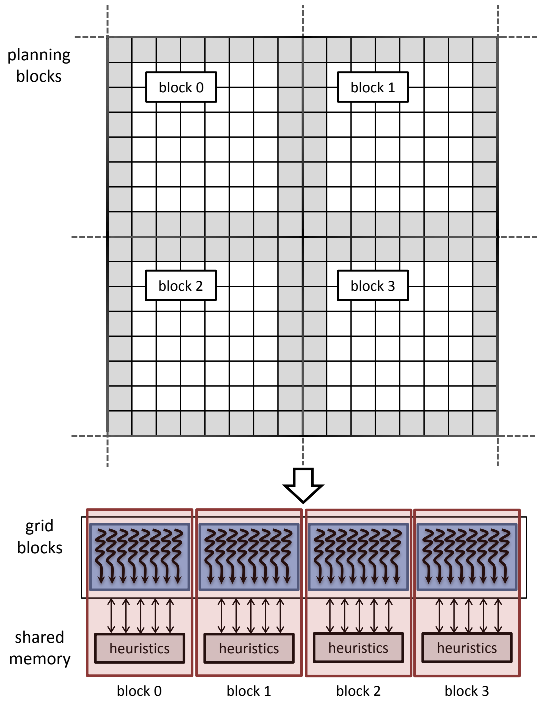


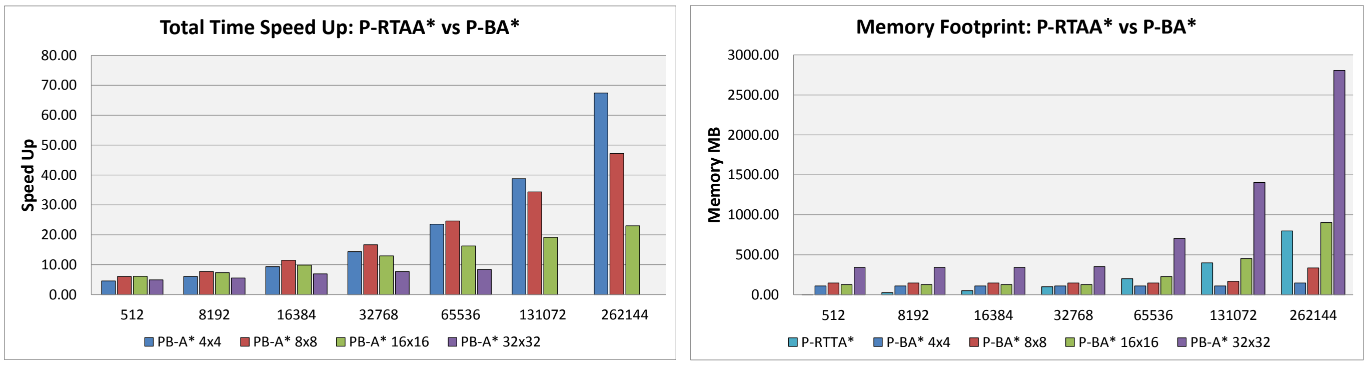
Figure 4. Parallelizing the border-to-border step using GPU. A planning block is associated with a thread block, and a thread executes a search for each border tile. This mapping enables us to use the shared memory in a thread block to store the estimated heuristics of a planning block. Finally, all planning blocks are gathered in a single CUDA grid.

Also in this step, the threads are in charge of the paths reconstruction and at the end of start-to-border all the subpaths are copied from the GPU to the CPU. For both types of steps, another element that is shared between all agents and stored in a global memory is the map that retains the cost of movement from each tile to its neighbors and therefore also the positions of obstacles.

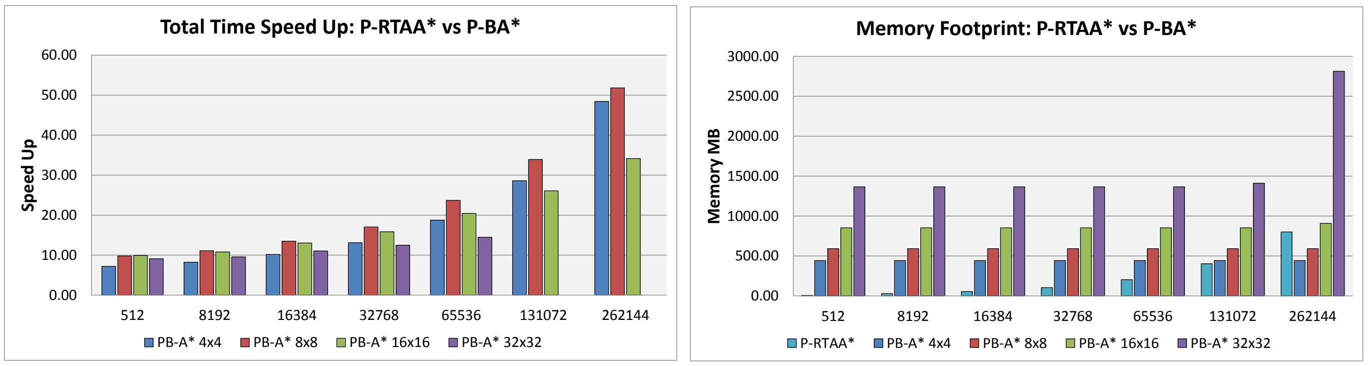
The planning-block size affects the behavior of searches because it determines the number of states in a single planning block, the array dimensions for shared memory used to maintain the heurists, and the number of border tiles. If the planning block is too large, the A* algorithm is required to explore too many states, while if it is too small, there are fewer states to explore but many searches to execute in the same length of time. However, to optimize time execution, we select the planning-block size to always be a power of 2 so that we can use CUDA bitwise operations to replace integer division and modulo operation, which are too expensive to run and are necessary to retrieve tile coordinates and consequently the planning-block ids.

V. Experiments and Results

In this section, we provide the results of two experiments. The first type of experiment demonstrates the efficiency of our approach, while the second is related to the quality of the trajectory found using the search space decomposition. Our tests were performed on an Intel Core i7 CPU 1.6GHz,



(a) GPU total time speed-up and memory footprint (MB) for a 512×512 map.



(b) GPU total time speed-up and memory footprint (MB) for a 1024×1024 map.

Figure 5. GPU total time speed-up values and memory footprint compared to GPU implementation of RTAA* with groups of agents ranging in size from 512 to 262144. We also include the time to transfer data from CPU to GPU and vice versa. Memory footprints are for border-to-border and start-to-border searches. Note that in two cases the required memory is higher than the memory available for computation.

NVIDIA Fermi GTX 480 1.5GB, Windows 7. All the kernels were written in CUDA 4.0 and using Microsoft's Visual C++ 2010 compiler.

The first experiment compares our GPU parallel approach based on planning blocks (P-BA*) with a GPU implementation of RTAA* (P-RTAA*) [8]. RTAA* is a real-time heuristic search method that selects its local search spaces in a very fine-grained way. The basic principle is to update the heuristics of all states in the local search space swiftly and to save the heuristics so as to speed up future A* searches. This approach uses a variable called `lookahead`, which specifies the largest number of states to expand during A* searches, and was used in the GPU implementation to reduce the memory footprint required for each agent. We chose to compare our approach with a GPU parallel version of RTAA* because in previous work [3] this implementation was found to be faster than a parallel GPU implementation of A* [1].

Two grid maps measuring 512×512 and 1024×1024 with several groups of agents ranging in size from 512 to 262144 were used to assess performance and memory footprint with planning blocks measuring 4×4, 8×8, 16×16, and 32×32. In P-RTAA* the number of searches and then threads run on the GPU is always equal to number of agents. Conversely, in P-BA* the number of agents determines only the number of start-to-border searches because border-to-border searches depend on the planning-block dimensions and on the number of border tiles. For instance, in the 1024×1024 map we have 786432, 458752, 245760, and 126976 threads for all planning-block sizes tested.

In all configurations, start positions were randomly chosen in the grid map, whereas the stop tile was always the center tile of the map. Also, the heuristic values $h[s]$ are precomputed off-line and stored in a matrix large as the grid maps. Figure 5 reports GPU total speed-up time and memory footprint compared to P-RTAA*. GPU implementation of RTAA* is always executed with `lookahead` = 3, which is the optimal value for achieving best performance as described in [3]. The results indicate that our approach is faster than P-RTAA*. The average speed up acceleration for each group of agents ranging from 5X to 45X in the 512×512 map and from 9X to 44X in the 1024×1024 map; measured time values include memory transfer time (CPU to GPU and vice versa) and kernel execution time. However, although we observed that shared memory improves performance, its use implies a degree of variability across the tested configurations. Conversely, P-RTTA* exhibited a better memory footprint in most cases, because of the greater amount of memory required to store the searches generated in the border-to-border step compared with P-RTAA*. However, as the number of agents increases, the number of searches is expected to rise considerably, and

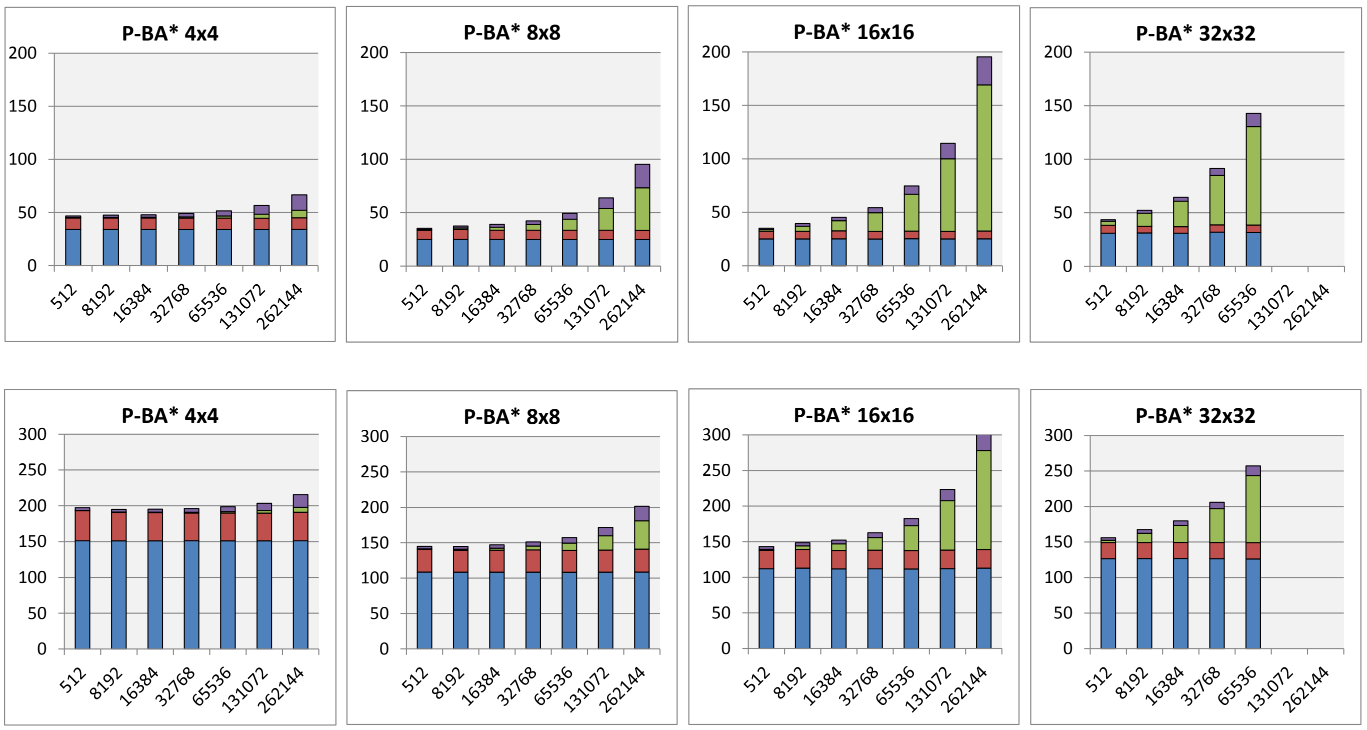


Figure 6. GPU times (ms) broken down into different steps on map 512 × 512 (up) and 1024 × 1024 (down) using a groups of agents ranging in size from 512 to 262144. Times are grouped in border-to-border searches (blue), GPU to CPU memory transfer for border-to-border outputs (red), start-to-border search (green), and GPU to CPU memory transfer for the start-to-border outputs (violet).

so also the memory footprint. Note that in general a planning block measuring 8×8 offers better performance in terms of acceleration and memory footprint.

The second experiment computes the computational times of our GPU implementation broken down in different steps as illustrated in Figure 6. It is clearly shown how border-to-border is independent from the number of agents and depends only on the planning blocks size. On the other hand, start-to-border is strongly dependent on the number of agents.

The last experiment concerns the lengths of paths obtained via our approach through the introduction of planning blocks and the number of paths computed respect to A* algorithm. We measured the average path length using 32×32, 64×64, 128×128, 256×256, and 512×512 grid maps with an increasing rate of obstacles. One A* search and three P-BA* searches were performed with planning blocks measuring 4×4, 8×8, and 16×16 with the upper-left corner as the start position and the lower-right corner as the goal position. Table I lists the average path lengths for 100 runs. For each run, we placed obstacles chosen randomly, and because of this, there may have been maps where there was no path from start to goal positions. We report the number of the paths found as A* paths. The results indicate that the length of the path retrieved with our approach is substantially the same as calculated with sequential A* and increasing the size of planning blocks involves a path length near the optimal solution. On the other hand, some paths are not returned especially as the number of obstacles increase. This is due to the formation of the loops described above.

These experiments suggest that our approach finds paths whose difference from the optimal path length is not significant. This deficiency is compensated for in terms of efficiency, as shown in the performance experiments. Fine-tuning the planning-block dimensions allows the user to trade off speed against path optimality. For example, in real-time applications, speed is the highest priority and suboptimal paths may thus be acceptable.

VI. Conclusion

In this work, we have demonstrated a parallel implementation based on the A* algorithm that fits well with GPU parallel architecture. By using it to explore each potential subpath per thread, the method offers a simple and powerful way of planning trajectories for many thousands of agents in parallel. Our results show that the GPU implementation improves by up to 45 times on that of RTAA*. The proposed solution follows other studies that have examined large multi-agent path planning problems trading the completeness for an improved efficiency. We believe that this is the first study that tries to exploit the GPU to obtain in practice a solution to the multi-agent path planning problem, allowing a real-time use of a vast number of agents in applications such as video games.

Future work may investigate the management of dynamic obstacles that occur in the grid map and in particular agent collisions. One of the advantages of this approach is that once

TABLE I
AVERAGE STEPS AND NUMBER OF PATHS FOUND OF A* AND P-BA*.

Map	Obstacle Rate	A*	P-BA*-4	P-BA*-8	P-BA*-16	A* Paths	P-BA*-4 Paths	P-BA-8 Paths	P-BA*-16 Paths
32×32	10%	33.850	33.897	33.838	33.778	100	97	99	99
	15%	35.102	35.500	35.489	35.277	98	84	92	94
	20%	37.553	36.822	37.016	37.181	85	45	61	72
	25%	39.338	38.000	38.441	38.625	74	18	34	48
64×64	10%	68.53	68.67	68.29	68.45	97	85	90	95
	15%	70.66	71.88	71.32	71.44	93	64	73	87
	20%	74.66	73.50	73.83	73.75	85	30	42	53
	25%	79.78	77.40	77.29	78.24	69	10	17	29
128×128	10%	138.12	138.88	138.48	138.09	99	82	86	94
	15%	142.84	143.38	142.84	142.92	91	47	63	79
	20%	151.02	148.60	149.32	149.41	85	10	19	34
	25%	161.43	-	155.00	153.00	69	0	3	7
256×256	10%	277.10	277.93	276.73	276.49	100	54	8	14
	15%	287.48	288.60	287.93	286.56	97	15	13	13
	20%	301.57	292.00	296.40	298.64	88	1	9	15
	25%	324.21	-	314.00	309.75	61	0	1	3
512×512	10%	557.17	561.42	556.53	556.33	99	33	45	61
	15%	577.04	580.50	571.88	573.86	99	4	16	28
	20%	605.31	-	-	601.33	91	0	0	3
	25%	639.09	-	-	-	90	0	0	0

the system recognizes the presence of an obstacle, only in the planning blocks where they occur will a new border-to-border and/or start-to-border step be performed. Thus, agents should be able to adapt swiftly to changes in the map. Also, we plan to investigate an approach to detect and remove loop formation before the path is available to an agent. Finally, we plan to validate this approach in larger simulations and experiments on a collection of maps extracted from popular video games and to release this implementation over an open source license.

REFERENCES

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

[2] M. Erdmann and T. Lozano-Prez, "On multiple moving objects," *Algorithmica*, vol. 2, pp. 477–521, 1987.

[3] U. Erra and G. Caggianese, *Real-time Adaptive GPU multi-agent path planning*, GPU Computing Gems Jade Edition ed. Morgan Kaufmann Publishers Inc., 2011, vol. 2, ch. 22, pp. 295–308.

[4] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100 –107, july 1968.

[5] J. E. Hopcroft, J. T. Schwartz, and M. Sharir, "On the complexity of motion planning for multiple independent objects: PSPACE-hardness of the "warehouseman's problem"," *International Journal of Robotics Research*, vol. 3, no. 4, pp. 76–88, 1984.

[6] G. J. Katz and J. T. Kider, Jr, "All-pairs shortest-paths for large graphs on the GPU," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, ser. GH '08. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 47–55.

[7] J. Kider, M. Henderson, M. Likhachev, and A. Safonova, "High-dimensional planning on the GPU," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, may 2010, pp. 2515 –2522.

[8] S. Koenig and M. Likhachev, "Real-Time Adaptive A*," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, ser. AAMAS '06. New York, NY, USA: ACM, 2006, pp. 281–288.

[9] Nvdia, *NVIDIA CUDA Compute Unified Device Architecture - Programming guide*.

[10] M. R. K. Ryan, "Exploiting subgraph structure in multi-robot path planning," *J. Artif. Int. Res.*, vol. 31, no. 1, pp. 497–542, Mar. 2008.

[11] D. Silver, "Cooperative pathfinding," in *AIIDE*, 2005, pp. 117–122.

[12] E. Stefan and S. Damian, "Parallel state space search on the GPU," in *International Symposium on Combinatorial Search (SoCS)*, 2009.