Final published version in ACM Digital Library available at https://dl.acm.org/citation.cfm?id=3079072
DOI: http://dx.doi.org/10.1145/3078659.3079072

On the Accuracy of Near-Optimal GPU-Based Path Planning for UAVs

Daniele Palossi[a]
dpalossi@iis.ee.ethz.ch

Andrea Marongiu[ab]
a.marongiu@iis.ee.ethz.ch

Luca Benini[ab]
lbenini@iis.ee.ethz.ch

[a]IIS, ETH Zürich
Gloriastrasse 35
Zürich, Switzerland 8092

[b]DEI, University of Bologna
Viale del Risorgimento 2
Bologna, Italy 40136

ABSTRACT

Path planning is one of the key functional blocks for any autonomous aerial vehicle (UAV). The goal of a path planner module is to constantly update the route of the vehicle based on information sensed in real-time. Given the high computational requirements of this task, heterogeneous many-cores are appealing candidates for its execution. Approximate path computation has proven a promising approach to reduce total execution time, at the cost of a slight loss in accuracy. In this work we study performance and accuracy of state-of-the-art, near-optimal parallel path planning in combination with program transformations aimed at ensuring efficient use of embedded GPU resources. We propose a profile-based algorithmic variant which boosts GPU execution by up to $\approx 7\times$, while maintaining the accuracy loss below 5%.

KEYWORDS

Path Planning, Embedded GPU, Parallel Graph Exploration, UAVs

1 INTRODUCTION

The interest in autonomous vehicles is growing constantly, with lots of practical applications appearing on the marketplace and many more being actively studied in academia, industry and military research departments. Unmanned aerial vehicles (UAVs) constitute a representative example of such technology [6, 8], already widely used for tasks such as aerial mapping, entertainment, surveillance, and rescue missions.

One of the key functional blocks for autonomous navigation is the path planner [4, 9], that constantly updates the route of the vehicle based on information sensed in real time. Besides selecting the "best"[1] path to the desired destination, the path planner is responsible for preventing collisions with dynamic, unexpected obstacles, adjusting the current trajectory as soon as on-board sensors detect them. Therefore, the reactivity of the UAV depends on the path planner response time.

Planning the route through the surrounding environment can be abstracted as the exploration of a large graph, which is known to be a compute-intensive task. For this reason, many techniques take advantage of parallel graph exploration on embedded graphic processing units (GPUs) [4, 5]. In addition to parallelism, to meet the application's real-time requirements on top of low-end accelerator devices, researchers are investigating *near-optimal* algorithms [3, 4, 7] to allow some degree of accuracy loss in path optimality in exchange for faster computation. A near-optimal path might be slightly longer than the shortest path, but the safety of the vehicle operation is not affected (on the contrary, a near-optimal path planner reacts faster to dynamic obstacles, which ultimately strengthens system safety).

In this paper we study the performance and accuracy of a state-of-the-art, near-optimal parallel path planner and propose an extension that enables more efficient utilization of GPU resources. Specifically, we observed that the original algorithm makes poor use of the GPU compute power due to a "sparse" workload distribution. Here, each thread continuously attempts to visit a node of the graph, but the operation is blocked until at least one of the predecessors has been visited and had its cost updated. This ultimately translates in limited thread usage and highly divergent control flow within thread *warps*. Moreover, it relies on a fine-grained synchronization scheme, which further limits parallelism.

To overcome this limitation, we propose an extension of the original algorithm that relies on a profile-based, offline stage to increase thread usage. During this stage we compute ahead of time *exploration frontiers*, which represent the set of nodes that can be visited at the current iteration of the algorithm. Frontiers introduce two important benefits: i) they expose dense, parallel workloads (dependencies for their node-sets are all resolved before computation starts); ii) they allow for a coarser synchronization scheme.

Since frontiers are computed for a given static snapshot of the map (e.g., an obstacle-free map), while obstacles appear dynamically during vehicle operation, we introduce a two-phase parallel exploration on the GPU. In the first phase we explore the graph one frontier after the other, where the nodes inside a frontier are visited in parallel. Since dynamic obstacles might alter the required order of exploration defined ahead of time in the frontier, we register the affected nodes in a separate data structure. In the second phase we complete the exploration of the deferred nodes by running a smaller instance of the original algorithm.

We study path optimality focusing on two sources of accuracy degradation: i) race conditions due to non-protected parallel path cost updates (as discussed in the original work); ii) dynamic obstacles that change the map on-line and alter the order in which nodes are visited (compared to the profile-based stage) during the parallel graph exploration. Experimental results show that i) race conditions play a very small role in the overall error; ii) the proposed method can lead to an improvement in performance up to $\approx 7\times$, keeping the error in path optimality lower than 5%, always guaranteeing the safety of the mission.

[1] According to specific metrics such as safety, distance, speed, energy savings, etc.

2 BACKGROUND

We start from a reference path planner implementation [4] based on a two-step process. The first step is the *automata synchronous composition* [1], where a graph representing a discretized topology of the environment (i.e., the map) is merged with a second graph representing the kinematics of the robot. Thus, the path returned by the path planner is also guaranteed to be compliant with maneuvers that the robot is able to perform (i.e., no need for a posteriori validation of kinematics). The price for this is an increased size of the graph to be explored. In the reference implementation, the *composition automaton* (i.e., the combined graph) is 21 times larger than the map size.

The second step is the exploration of the composition automaton, which implies solving the single source shortest path (SSSP) problem. Finding a feasible path on the composition automaton means finding a path that is feasible both in term of obstacle-free locations and in term of maneuvers that the robot is able to perform. The SSSP problem with non-negative weights can be stated as follows: given a weighted graph $G = (V, E, c)$, where V is the set of *vertices* or *nodes*, E the set of *edges* (i.e., pairs of nodes) and c the *cost* ($c : E \rightarrow \mathbb{R}_+$), find a minimal weight path from one chosen node $s \in V$, called the *source node*, to all other nodes in V. We say that the nodes $v, w \in V$ are *neighbors* if $(v, w) \in E$, i.e., if there exists an edge between them. The core algorithm to solve the SSSP problem is Dijkstra [2].

The main data structure used is a sparse *state-transition matrix* used to represent both vertices and edges. This matrix contains the information about which neighbors that can be reached from any reference node. The transition matrix is stored in the *global memory*, that is mapped in system DRAM. The information about which nodes are "to be visited" is kept in an auxiliary array called *mask array*. The information regarding the cost (i.e., the weight, or distance) to reach each node is stored in the *cost array*. While the mask array indicates that there are still vertices to be explored, another parallel iteration is performed. The vertices to be explored in each iteration are referred to as *reference nodes*, and for each reference node, all neighbors are explored by comparing the current cost to reach that neighbor (stored in the cost array) with the cost to reach the neighbor through the path of the current reference node. If the new cost is lower than the previous cost, the cost array is updated and the neighbor is marked in the mask array.

The authors of the original paper showed that a non-deterministic version of this algorithm, where parallel updates to the cost array are not protected for mutually exclusive operation, is 3.7 times faster than deterministic implementations, at the cost of a small error in the path optimality (lower than 1.2% on average).

Our case study is based on this non-deterministic version, which permits race conditions during the costs update of the graph to increase the degree of parallelism, which is particularly well suited for GPU execution. In the following, we refer to this baseline approach as the *Naive* algorithm.

In practical applications (e.g., UAV), in order to deal with dynamically changing environments, each time a new obstacle is detected by on-board sensors the transition matrix is updated, invalidating the vertices related to the occupied locations on the map. A new parallel exploration is thus performed, where all the nodes previously invalidated are skipped. The overhead introduced for on-line updates to the transaction matrix is negligible w.r.t. the time required by the parallel exploration (i.e., respectively few μs and tens of ms). Thus, without loss of generality our experiments focus on exploration time and accuracy for a given transaction matrix (i.e., a given map with a given percentage of obstacles).

3 PROFILE-BASED APPROACH

In the *Naive* parallel algorithm each thread is assigned a-priori a (set of) node(s) from the transition matrix to be visisted. However, this workload distribution criterium does not take into account the dependencies created by the visit order imposed by the Dijkstra algorithm. Thus, many threads are blocked for several iterations, until such dependencies are solved (i.e., at least one of the predecessors has had its cost updated. Besides the inherent poor use of the GPU computational resources implied by this parallelization scheme, divergent control flow within the same warp further limits the performance, as well as a fine-grained synchronization scheme required to check visit order dependencies. Such a scenario is depicted in Figure 1 (B).

To overcome the poor usage of the compute power of the *Naive* implementation, we reorder the elements of the transition matrix so as to ensure a "dense" workload. To do this, we introduce a profiling stage which performs an offline exploration of a static snapshot of the map, and we introduce the concept of *exploration frontiers*. The exploration frontier is an enumeration of sets of vertices F, where all vertices in F_n have been visited from at least one vertex in F_m for any $m : 0 \geq m < n$. As shown in Figure 1 (A), the base case is F_0 which contains only the source vertex. The next frontier is constructed by all the vertices that can be reached from the source vertex, and then the remaining frontiers are in turn populated by the vertices that can be reached by the previous frontier. The exploration of the graph in a "frontier-step" fashion, where all the nodes inside the same frontier are explored in parallel, enables higher computational parallelism.

The organization of the parallel work in frontiers permits also the adoption of a coarser synchronization scheme w.r.t. the *Naive* implementation. In fact, we can relay on the insertion of breakpoints in the streaming transition matrix, at which points all previous vertices must have been explored before the exploration can continue beyond that point in the stream, thus ensuring that nodes are not visited out of order.

The static map snapshot used to populate the frontiers in this offline stage can be taken at the beginning of the vehicle's operation, or it could represent a predefined environment with known, static obstacles, etc. Without loss of generality in this work we consider as a static snapshot an empty map (i.e. without any obstacles). As some vertices of the graph may be explored multiple times, to keep the streaming property of the transition matrix, these vertices must be added multiple times (in Section 4 we will discuss on the resulting increase in memory usage).

We refer to this new version of the near-optimal path planning algorithm as *Prof.* A visual representation of its execution model is presented in Figure 1 (C), which shows that each warp now explores effectively multiple vertices, delivering higher GPU utilization.

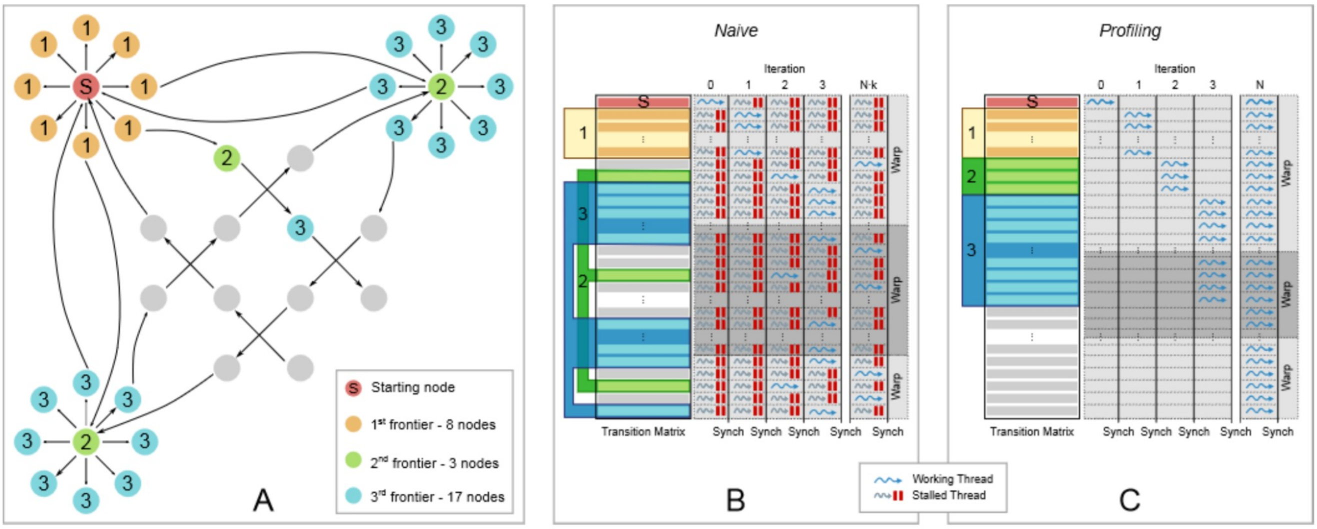

Figure 1: A) An example on how the frontiers are derived in *Prof.* B-C) A visualization on the threads' workload, respectively for the *Naive* (B) and the *Prof* (C) version.

Since dynamic obstacles might alter the visit order defined ahead of time by the frontiers, we defer the exploration of all the nodes scheduled for the exploration but not yet reached to a second phase of the algorithm. This second phase is implemented as a smaller instance of the original algorithm, where we limit the exploration to the deferred nodes, avoiding any further cost propagation. This mechanism on one side limits the overhead introduced but on the other side represents a new source for inaccuracy. This source of non-optimality will be also considered in Section 4 in addition to the classical *race conditions* [4].

There are obviously many ways to perform the exploration during the offline profiling, with different resulting transition matrices and frontiers. The proposed solution is based on the classical sequential Dijkstra exploration, fetching at each iteration the neighbor with the minimum cost first (namely *Prof-Min*). For the sake of completeness we also evaluate a different approach, where we revert the previous approach fetching at each iteration the neighbor with the maximum cost first (namely *Prof-Max*). In this latter case, we penalize performance in favor of accuracy due to the insertion of a node multiple times in different frontiers. In Section 4 we will investigate also the effect of both approaches and their impact on performance and accuracy.

4 RESULTS & DISCUSSION

We conduct our experiments on the NVIDIA Tegra TX1, a state-of-the-art heterogeneous, many-core SoC featuring 4-core ARM Cortex A57 and a Maxwell GPU[2].

We compare the original *Naive* algorithm to the two variants of the proposed *Prof* algorithm, scaling the map sizes up to 100×100 (the maximum size considered in the original non-deterministic planner paper). For all the experiments we run 1024 CUDA threads, the maximum we can schedule within the same block and which makes best use of the available cores and memory bandwidth.

Similar to the reference work [4], we consider a deadline for computing a new path in presence of dynamic obstacles of $250ms$. This has been chosen considering a vehicle speed of $4m/s$ and a minimum obstacle detection distance of 1 meter. In addition to that, we consider a second, stricter deadline of $50ms$. This is representative of a cutting-edge commercial quadcopter[3] with advanced autonomous navigation capabilities, capable of moving at $20m/s$.

Figure 2 shows execution time (left Y-axis, coloured bars) and percent error (right Y-axis, coloured markers) for the various algorithms. We show four different plots, for an increasing obstacle rate. This obstacle rate refers to the (percent) number of obstacles (dynamically appearing on the map) as compared to the static map snapshot used for the off-line profiling stage. In these experiments we consider the zero-obstacle map as a reference snapshot, the results for which are shown on the top-left plot (A).

Focusing on this plot, which represents a best case for the proposed *Prof* approach (only the first stage of the algorithm is executed), we observe up to ≈ 7× faster execution than *Naive*. In absence of dynamic obstacles, this approach ensures optimal path calculation (zero error). In line with what is published in [4], *Naive* shows an error that is below 1% (on average around 0.27%). It has to be reminded that the error to which we refer here never affects safety, only optimality of the computed route (i.e., shortest path).

As the percentage of obstacles increases, the speedup of *Prof* versus *Naive* diminishes as expected (as the second stage of our algorithm has increasingly more work to do). However, we still observe a net 2× speedup for 100×100 maps and 30% obstacles (Figure 2 (D)), which in practical

[2] http://www.nvidia.com/object/jetson-tx1-dev-kit.html

[3] https://www.dji.com/phantom-4/info

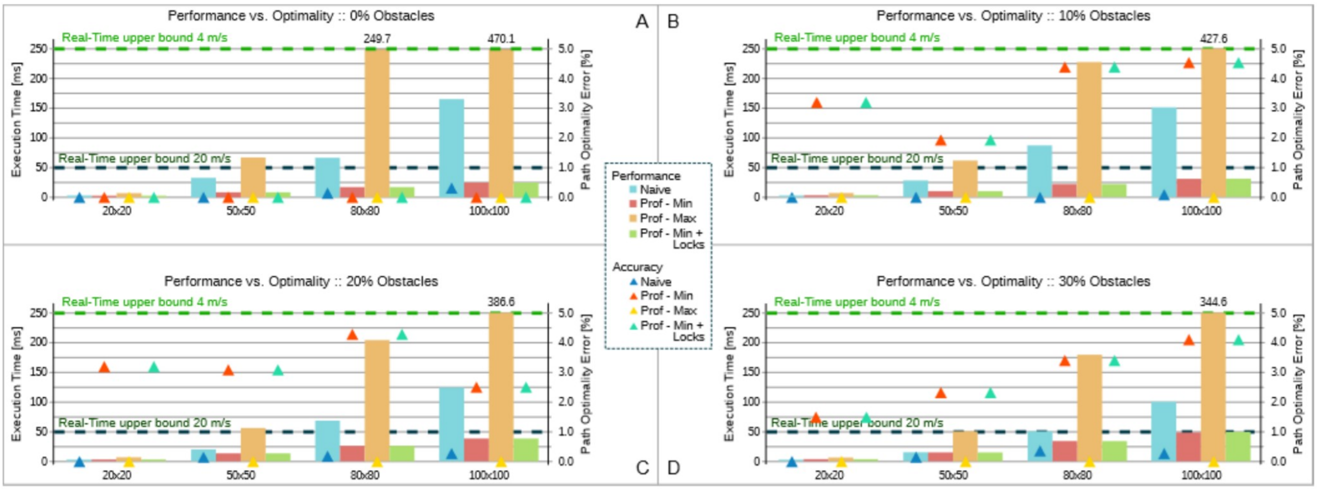

Figure 2: Performance and path optimality for *Naive* and *Prof* algorithms, with 0% (A), 10% (B), 20% (C) and 30% (D) of obstacles.

cases represents a very large value[4]. This is achieved at the expense of a modest loss of accuracy in path optimality, as the error is always below 5%. Note that while we show results for only up to 30% obstacles, for higher rates the error tends to diminish (with less feasible paths all methods increasingly converge to the optimal one). For 50% obstacles we have measured a worst-case error for *Prof-Min* of ≈ 0.5% (map size 100 × 100).

For all the considered map sizes and % obstacle rates, *Prof-Min* is the only approach capable of meeting the deadlines imposed by the two considered real-time constraints, while *Naive* only meets the requirements of the slower vehicle use-case ($4m/s$).

The *Prof-Max* variant, which is the only approach that ensures optimal path calculation in any circumstance, is up to 4× slower than *Naive*, which causes it to miss real-time deadlines already for small-medium map sizes (50×50).

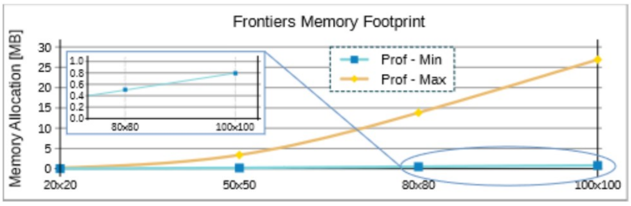

Figure 3: Frontiers memory usage for *Prof-Min* and *Prof-Max*.

Finally, *Prof-Min Lock* employs fine-grained locking to protect the updates to the cost array. This variant of the algorithm thus prevents the race conditions that in the original *Naive* algorithm caused the non-optimality of the computed path. The same race conditions are also present in all the other approaches. The results in Figure 2 show that different from the original *Naive* algorithm, race conditions contribute to the error in a negligible manner in the *Prof-Min* algorithm.

For completeness, Figure 3 shows the frontiers' memory footprint for the *Prof-Min* and *Prof-Max* algorithms. It can be seen that for the former the memory increase is a linear function of the map size, which practically introduces negligible overhead for the considered problem instances.

5 CONCLUSION

Near-optimal parallel path planning is being investigated by researchers as a technique to meet the real-time requirements on top of low-end accelerator. In this paper we have discussed a novel approach that extends state-of-the-art algorithms with the aim of ensuring efficient use of embedded GPU resources. This leads to an improvement of the overall performance of ≈ $7x$ at the price of a loss in path optimality of ≈ 5% never affecting the safety of the mission.

ACKNOWLEDGMENTS

This work has been funded by projects EC H2020 HERCULES (688860) and Nano-Tera.ch YINS. The authors thank Björn Forsberg for his support.

REFERENCES

[1] Christos G Cassandras and Stephane Lafortune. 2009. *Introduction to discrete event systems*. Springer Science & Business Media.

[2] E.W. Dijkstra. 1959. A Note on Two Problems in Connexion With Graph. *Numer. Math.* 1 (1959), 269–271.

[3] Ulises Orozco-Rosas, Oscar Montiel, and Roberto Sepúlveda. 2017. An Optimized GPU Implementation for a Path Planning Algorithm Based on Parallel Pseudo-bacterial Potential Field. In *Nature-Inspired Design of Hybrid Intelligent Systems*. Springer.

[4] Dynamic obstacles are expected to change the reference map layout very slowly over time. The measured worst-case execution time for the offline profiling stage on the ARM CPU is 896ms, which would allow to recompute the reference snapshot roughly every second. During this time frame, even considering the most advanced sensors, the number of dynamic obstacles detected along the path would remain well below 30%. Note that recomputing the reference snapshot would happen fully in parallel to GPU graph exploration.

[4] Daniele Palossi, Michele Furci, Roberto Naldi, Andrea Marongiu, Lorenzo Marconi, and Luca Benini. 2016. An Energy-efficient Parallel Algorithm for Real-time Near-optimal UAV Path Planning. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 6. DOI:https://doi.org/10.1145/2903150.2911712

[5] Daniele Palossi and Andrea Marongiu. 2016. Exploring Single Source Shortest Path Parallelization on Shared Memory Accelerators. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES '16)*. ACM, New York, NY, USA, 197–200. DOI:https://doi.org/10.1145/2906363.2915925

[6] D. Palossi, A. Marongiu, and L. Benini. 2017. Ultra low-power visual odometry for nano-scale unmanned aerial vehicles. In *2017 Design, Automation Test in Europe Conference Exhibition (DATE)*.

[7] V. Roberge, M. Tarbouchi, and G. Labonte. 2013. Comparison of Parallel Genetic Algorithm and Particle Swarm Optimization for Real-Time UAV Path Planning. *IEEE Transactions on Industrial Informatics* (2013). DOI:https://doi.org/10.1109/TII.2012.2198665

[8] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. 2011. *Introduction to autonomous mobile robots.* MIT press.

[9] Daniel Watzenig and Martin Horn. 2017. *Introduction to Automated Driving*. Springer International. DOI:https://doi.org/10.1007/978-3-319-31895-0_1