ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

An energy-efficient parallel algorithm for real-Time near-optimal UAV path planning

This is the final peer-reviewed author’s accepted manuscript (postprint) of the following publication:

*Published Version:*
An energy-efficient parallel algorithm for real-Time near-optimal UAV path planning / Palossi, Daniele; Furci, Michele; Naldi, Roberto; Marongiu, Andrea; Marconi, Lorenzo; Benini, Luca. - STAMPA. - (2016), pp. 392-397. (Intervento presentato al convegno ACM International Conference on Computing Frontiers, CF 2016 tenutosi a Como, Italy nel 16-19 May 2016) [10.1145/2903150.2911712].

*Availability:*
This version is available at: https://hdl.handle.net/11585/588239 since: 2019-05-02

*Published:*

DOI: http://doi.org/10.1145/2903150.2911712

*Terms of use:*

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

03 May 2024

This is the peer reviewed version of the following article:

Palossi, D., A. Marongiu, L. Marconi, M. Furci, R. Naldi, and L. Benini. “An Energy-Efficient Parallel Algorithm for Real-Time near-Optimal UAV Path Planning.” 2016 ACM International Conference on Computing Frontiers - Proceedings, 392–97

© ACM 2016. This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive Version of Record was published in CF16 Proceedings of the ACM International Conference on Computing Frontiers

http://dx.doi.org/10.1145/2903150.2911712

# An Energy-Efficient Parallel Algorithm for Real-Time Near-Optimal UAV Path Planning

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

Michele Furci[*]
michele.furci@unibo.it

Roberto Naldi[*]
roberto.naldi@unibo.it

Andrea Marongiu[†‡]
amarongiu@iis.ee.ethz.ch

Lorenzo Marconi[*]
lorenzo.marconi@unibo.it

Luca Benini[†‡]
lbenini@iis.ee.ethz.ch

[†]IIS, ETH Zürich
Zürich, Switzerland

[*]CASY, University of Bologna
Bologna, Italy

[‡]DEI, University of Bologna
Bologna, Italy

## ABSTRACT

We propose a shortest trajectory planning algorithm implementation for Unmanned Aerial Vehicles (UAVs) on an embedded GPU. Our goal is the development of a fast, energy-efficient global planner for multi-rotor UAVs supporting human operator during rescue missions.
The work is based on OpenCL parallel non-deterministic version of the Dijkstra algorithm to solve the Single Source Shortest Path (SSSP). Our planner is suitable for real-time path re-computation in dynamically varying environments of up to 200 $m^2$. Results demonstrate the efficacy of the approach, showing speedups of up to 74x, saving up to $\sim 98\%$ of energy versus the sequential benchmark, while reaching near-optimal path selection, keeping the average path cost error smaller than 1.2%.

## Keywords

Energy-Efficient Autonomous UAV; Parallel Path Planning

## 1. INTRODUCTION

The interest in the use of UAVs in dangerous scenarios, such as natural disasters or hazardous areas is growing quickly [18]. Autonomous robots can exploit complementary sense-act capabilities, supporting human operators in accomplishing surveillance and rescue tasks [16].
Our target scenario is research and rescue activities, where the rescuer has to operate in real-world, hostile environment, using robotic helpers. We address an unstructured and dynamically changing environment, where real-time environment analysis is required (e.g. after a landslide or an earthquake). The human operator must be considered as not always available to supervise the robotic platform, thus robots need to have a high degree of autonomy. Small multi-rotor UAVs (i.e. quadrotors) are used to give the rescuer the ability of monitoring a large area, as "flying eyes", keeping costs and risks at a reasonable level and improving his perception and knowledge.
Autonomous navigation and cognitive capabilities play a crucial role to increase the efficiency and reliability of humans in see-and-listen missions. Required navigation skills are: patrolling, path following and "follow me" capability. Cognitive skills are: obstacle avoidance [11], ambient awareness [21], feature detection and Simultaneous Localization And Mapping (SLAM) [22]. All requirements involve performing computationally demanding tasks in real-time.
Multi-rotor UAVs are characterized by the presence of a sensing visual system, that captures information from a preferential point of view, allowing on-board computation. The embedded processing device is in charge of both analyzing the visual information detected and calculating the plan of flight, as in Fig. 1. These kinds of robots are characterized by constrained resources: payload, batteries autonomy, operative radius and on-board computational capability.
Our work focuses on the UAVs navigation skill. We propose a novel real-time parallel path planner, which allows autonomous navigation using only on-board computation, handling the entire mission area. We demonstrate the energy-efficiency and the path optimality of the proposed implementations. Due to the given use case, we selected the Intel Graphics HD 4000 GPU to execute our non-deterministic version of the Dijkstra algorithm [8], developed with OpenCL.
We believe that reducing the power consumption of the control system in UAVs will become increasingly important as the size of the vehicle is scaled down. Indeed, reducing the UAV size quickly leads to order-of-magnitude reduction of the power spent on the propellers. Beside, the computational load required to implement the UAV cognitive skills does not vary with the vehicle size, and will thus constitute an increasingly larger fraction of the total system power consumption. In this context, the proposed solution is particularly convenient for ultra-low-power embedded devices.
The remainder of the paper is organized as follows: Sec. 2 summarizes the related work and the state-of-the-art in the trajectory planning field for UAVs. The used hardware and software architectures are described in Sec. 3, the algorithm and our implementations are summarized in Sec. 4, and experimental results are given in Sec. 5. Finally, Sec. 6 concludes the paper.

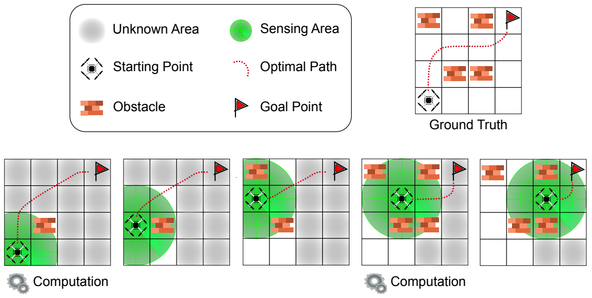

**Figure 1:** Planning strategy: computation is performed only when a new obstacle is detected nearby the current trajectory.

## 2. RELATED WORK

The increasing number of contributions show growing interest of advanced motion planning algorithms for UAVs.

ACM ISBN .
DOI: http://dx.doi.org/10.1145/2903150.2911712

A widely used approach in literature is the layered planner. This combines low-level reactive local planning for fast reactivity but limited range (i.e. obstacles avoidance), with a high-level global planning for long range but high computation time [17, 19]. The global path planner typically requires some deliberation time and may not find a solution in time to prevent a collision. Thus, the reactive local planner keeps the vehicle safe from immediate threats by reacting quickly, in a potentially suboptimal manner. In [19], the global planner follows generalized Voronoi diagram graph while local planner is based on potential field theory [1]. In this approach, the global planner computational time (up to 10 seconds) is not suitable to be used for dynamic environments moreover the generated trajectories are not optimal. Several other approaches, surveyed in [10], assume complete knowledge of the environment geometry, giving a strict limit to the applicative scenario.

When dynamic real-world populated environments are considered, planning the motion of the vehicle may require solving complex optimization problems with high computational load [15]. A way to solve high computation problem consists in discretizing the environment space and/or the operative space of robots, resulting in a discrete manifold described by graph or lattice. Graph search or discrete techniques can then be used to find solution on the discrete manifold in a computationally optimized way [14].

Graph optimization, using parallel devices, are highly multi-domain and considered in several works [6, 7, 3]. They deal with huge dimension graphs in order to exploit all the computational power delivered by high performance GPUs. In [7], they reorder the vertices, before the path computation, to reduce the number of cache misses, but the algorithm is efficient only if there are sufficient computations to amortize the preprocessing cost. In [3], the problem is tackled only in terms of minimal cost path, reducing the memory usage and unconcerning information to generate the route.

A graph approach is considered in [5]. The graph considered connects vertices of surfaces tangent to obstacles. The graph size is reduced in comparison with uniform discretization of maps but the trajectories are limited to be tangent to obstacles, and the optimality could be considered only for minimum length problems. Another "discrete" planner can be found in [13]. The author proposes a strategy to dynamically dodge obstacles by generating an escape waypoint when a new obstacle is detected. This algorithm aims at fast computation using voxel discrete structure and simple avoid strategy, but it does not take into account the feasibility and optimality of the solution.

A similar approach to the algorithm presented in this paper is given in [11], where the kinematic of the robot is directly taken into account in the planning stage. However, we propose a solution that does not use any heuristic for the planning, getting the optimal path. In addition, we are also able to compute a map 8-10 time bigger in the same time.

The power consumption of the UAV control system plays a crucial role in the overall system energy efficiency when the size of the vehicle is scaled down. The power analysis of the on-board feedback control and sensing of a miniature robotic platform is given in [2]. In [23], the authors estimate in 5 mW the power budget to perform navigational skills on pico-size UAVs, endorsing the need for energy-efficient solutions.

In this work we present an energy-efficient optimal global path planner, suitable for real-time decisions, capable of handling a dynamically varying environment using only on-board computational resources exploiting parallel programming techniques.

# 3. SYSTEM ARCHITECTURE

Our planner is tailored to the architectural model depicted in Fig. 2. The key components are: the *Sensing System*, the *Processing Device* and the *Autopilot*.

The *Sensing System* is composed by *Laser-scan* and *Stereo-rig*. They represent the sensors through which the robot "sees" the physical space. The output of the sensing system are the *Point Cloud* (i.e. 3-D model of the surrounding environment) and the *Visual Information* (e.g. obstacles, relevant features detected and the position itself of the quadrotor). *Sensing System* information, including *Measures* coming from the *Inertial Sensors*, represent information gathered by on-board sensors that are used by the *Visual & Map* module to build the internal map representation. This map is based on a space discretization to reduce the computational complexity and it can be expressed either by an *Occupancy Grid* or by a 3D occupancy map [12].

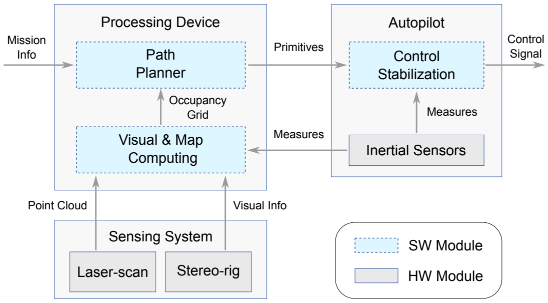

**Figure 2:** Overall HW & SW architecture of the UAVs.

The *Processing Device* performs the high level computation, including the visual analysis and exploration strategy. The *Path Planner* represents the heart of the control and it provides information about the feasibility of the mission, evaluating the optimal path. *Mission Information* is pushed to the *Path Planner* either by a human operator or by the system itself and represents the start point, destination and way-points to be visited during the patrolling path. The result of this stage is the best primitive sequence to achieve the mission with the minimal cost. Thus the *Path Planner* feeds the *Autopilot* with a sequence of *Primitives*, which represents elementary movements the robot is capable of doing. The *Autopilot* is in charge of the feedback control to stabilize and track the desired trajectory. It includes also the *Inertial Sensors* module (i.e. accelerometers and gyroscopes). Inertial *Measures* and *Primitives* are both sent to the *Control Stabilization* block, that connects each different primitive so as to obtain a sufficiently smooth path and converts the geometrical information into a time-law *Control Signal*. As introduced in [9], the control is able to produce a smooth path, dealing with trajectory discontinuities and keeping the real trajectory close to the primitives sequence.

For the class of UAVs considered in this work, with a total power budget of 300 Watts, the selected *Processing Device* is an embedded modular computer consuming, at most, one tenth of the total power. The device is based on an Intel Core Processor (Ivy Bridge[1]), with a Thermal Design Power (TDP) of 35 Watts. Furthermore, the Intel x86 architecture enables the exploitation of a wide range of legacy software layers, like the Robot Operating System (ROS) [20]. The processor integrates 2 CPU cores and a GPU that consists of 2 *sub-slices*, each one has 8 ALUs called execution units (EUs), each of which can perform 16 flop/cycle. Thus, the whole GPU can perform 256 flops/cycle.

The integrated GPU supports OpenCL, using Beignet[2] : an open source implementation of the OpenCL specification for Linux OS. The OpenCL Work-Groups (WGs) are mapped onto EU threads and then are distributed across EUs in the sub-slice. Each EU can run up to 8 EU threads to hide latencies and increase utilization. However, the number of Work-Items (WIs) in a EU thread is kernel-dependent. Each EU thread has access to a 4 kB register file. The compiler maps either 8, 16 or 32 WIs to one EU thread depending upon register usage of each WI, in a Single Instruction Multiple Data fashion (i.e. SIMD 8/16/32). Each sub-slice has access to 64 kB of local memory which is divided into 16 banks and provides 64 bytes/cycle of bandwidth. The global memory is allocated from system DDR memory.

---

[1] http://ark.intel.com/products/67355/Intel-Core-i5-3210M

[2] http://www.freedesktop.org/wiki/Software/Beignet

The exploration strategy developed, is aimed at exploring the unknown environment performing the path computation on-line, both at start time and when new relevant information is detected. New information could be represented either by obstacles detected or by updating mission information like the destination. The strategy is realized through an Application Programming Interface (API) that puts the control back to the *Visual & Map* module, allowing it to invoke a new path computation. The *Visual & Map* module pushes the *Occupancy Grid* to the *Path Planner* where it is used to build and update the map automaton.

The map automaton, Fig. 3 (a), represents the flying zone discretized to a fixed granularity. A single grain corresponds to a state (i.e vertex) and events (i.e. edges) represent elementary movements on the 2-D space. The quadrotor automaton, shown in Fig. 3 (b), is "hard-coded" into the planner module and represents the robot model and the interaction between its primitives and the movements on the map. The quadrotor and the map automaton are merged together using the supervisory control theory [4], producing the composition automaton. Finally, the *Path Planner* performs the graph exploration of the composition automaton in order to feed the *Control Stabilization* with the optimal path sequence.

## 4. ALGORITHMS & IMPLEMENTATIONS

In this section we present the algorithmic flow made of two main stages: the automaton synchronous composition and the SSSP. The first stage delivers a complete representation of the system in graph form, including forbidden zones. Then, we perform the SSSP exploration, for a given source and destination, running our OpenCL parallel implementation of the Dijkstra algorithm.

### 4.1 Automaton Synchronous Composition

Starting from the discrete map and the quadrotor automaton, Fig. 3, we perform the automaton synchronous composition to synchronize both automata in a new one that represents the whole system: the *composition automaton*. In the synchronous composition [4], a common event (i.e. events that occur in both automata) can only be executed if two automata both execute it simultaneously. Thus the two automata are "syncronized" by the common events. Private events (i.e. events that occur exclusively in an automaton) are not subject to such a constraint and can be executed whenever possible.

As shown in Fig. 3, we use a map automaton (a) with 4 events types (i.e. *Up*, *Down*, *Left*, *Right*) and a quadrotor automaton (b) made of 21 states and 12 events types (i.e. 8 primitives and the map's events). In the map automaton the number of the states depends on the dimension of the environment explored and on the step-size used during the discretization. The primitives of the quadrotor are: *go_0*, *go_45*, *go_90*, *go_135*, *go_180*, *go_225*, *go_270*, *go_315* and they represent straight lines at 45 degrees each. As shown in Fig. 3 (b), diagonal movements (i.e. *go_45*, *go_135*, *go_225*, *go_315*) have three states more than perpendicular movements in order to prevent possible collisions with obstacles lying on lateral locations, respect to the diagonal.

Before performing the synchronous composition, we build the map automaton starting from the occupancy grid. This stage is performed off-line on the CPU only at start time. During the flight (on-line), every time a new obstacle is revealed by sensors the map automaton is updated removing the state related to the occupied cell. Then, if the new obstacle lies either in the current path or in its side, a new graph exploration is executed on the GPU.

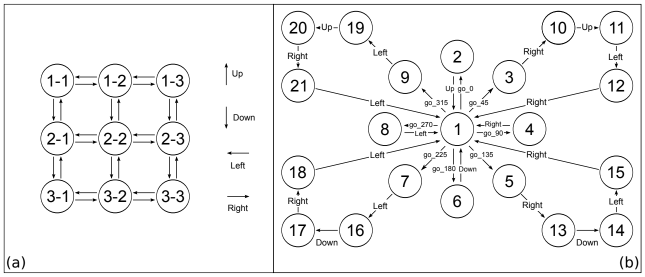

**Figure 3:** Map (a) and Quadrotor (b) automaton.

### 4.2 Single Source Shortest Path

In this section we present three different SSSP parallel implementations based on the Dijkstra algorithm. The Dijkstra algorithm has been chosen due to its global optimality property and for its adoption in *state-of-the-art* works like ROS [20]. 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* ($c : E \to \mathbb{R}_+$), 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 *neighbours* if $(v, w) \in E$, i.e., if there exists an edge between them. The graph used here is the outcome of the automaton synchronous composition where, defining $Q$ the number of states of the quadrotor and $M$ the number of states of the map automaton, the used graph has $\mathcal{O}(Q \cdot M)$ vertices.

We represent both vertices and edges using a state-transaction matrix stored in array form: $\vec{T}$. The matrix is built with one column for each edge type (i.e. 12 events) and one row for each vertex (state) of the composition automaton. As depicted in Fig. 3 (b) only central nodes can be reached by more than one edge (i.e. 8 edges in general). Thus, during the parallel graph exploration, the access to these nodes (i.e. nodes with an id multiple of 21 in the composition automaton) can turn out to be a race condition and consequently register of a not minimal cost for that vertex. In that case a non-optimal and non-deterministic path would be obtained. In order to avoid this behavior we propose two distinct parallel implementations: *Parallel Atomic* and *Parallel Multi-Buffer*. We also present a last implementation where race conditions are permitted: *Parallel Non-Deterministic*. In Sec. 5 we evaluate the non-deterministic solution in terms of performance respect to path optimality.

Auxiliary structures used in our implementations are: a mask array $\vec{M}$ and a temporary mask array $\vec{M}_{temp}$, a cost array $\vec{C}$ and a temporary cost array $\vec{C}_{temp}$. The mask arrays are used to keep track of the next nodes to be explored, using only one Byte for each vertex. In addition to updating the cost array, we also have to keep track of the predecessor of each element so that we can reconstruct the minimal cost path. Thus, we have to make sure that, when a thread is updating the predecessor id of a vertex $w$ in memory, while that thread is reading the value $\vec{C}[w]$, comparing it with its cost and writing the predecessor to memory, no other thread can change either of those variables. For this, however, atomic operations are not enough. We have to use a critical section, which serializes a part of the parallel execution.

To avoid this solution, we decided to package the two pieces of information into one 32-bit element. The 20 right-most bits are used to store the vertex, or *vertex id*, while the remaining bits store the corresponding cost, which means that we can again use atomic function. In this way we save both bandwidth and memory, still being able to compare costs, since only the left-most bits affect this comparison. We manipulate this packed data with fast bitwise operations. However, we have introduced two new constraints: the maximum dimension allowed for the composition automaton of $2^{20}$ vertices and the maximum cost allowed of $2^{12}-1$. These two new constraints do not represent an actual limitation because the real-time requirement introduces itself stricter upper bounds. Then, during the initialization stage, only the left-most 12 bits of $\vec{C}$ and $\vec{C}_{temp}$ are initialized to the maximum value allowed and all the id bits to 0. All proposed implementations share the same algorithmic structure on the host side (i.e CPU): a main loop handles the kernel invocations, where the termination is based on the change in cost. As reported in Algo. 1, additional structures are created on the host side and then initialized on the GPU, avoiding transfer overhead. The final result stored in $\vec{C}$ is then used by the host to compute the final path just performing a backward exploration starting from the destination vertex.

**Algorithm 1** OCL_SSSP (Graph G(V,E,W), Source S)

```
Create: $M$, $M_{temp}$, $C$, $C_{temp}$ for all $V$
Initialize: $M$ to 0, $C$ and $C_{temp}$ to $UINT\_MAX$ « 20
$M[S] \leftarrow M_{flag} \leftarrow 1$
$C[S] \leftarrow C_{temp}[S] \leftarrow 0$
while $M_{flag} = 1$ do
    for each vertex $V$ in parallel do
        invoke OCL_K_x.1
        invoke OCL_K_x.2 /*Not for Non-Deterministic*/
    end for
end while
```

The best OpenCL configuration experienced is, as expected, scheduling the maximum number of WIs that match the EU threads. Thus, every WI handles more vertices during the same execution.

### 4.2.1 Parallel Atomic

This implementation is composed of two kernels. The first kernel, Algo. 2, starts re-triggering the termination variable $M_{flag}$, shared among all WGs, to the termination value. In this way the execution will terminate only if no WI will update any vertex cost. Then, during each iteration and for each vertex, the mask array $\vec{M}$ is checked. If the current value is "*to be explored*", the cost and the neighbor's weight are fetched respectively from $\vec{C}$ and $\vec{W}$. The cost of each neighbor is updated if greater than the cost of the current vertex plus the weight of the incoming edge.

**Algorithm 2** OCL_K_1.1 ($\vec{T}, \vec{M}, \vec{C}, \vec{C}_{temp}, \vec{W}, M_{flag}$)

```
$M_{flag} \leftarrow 0$
$tid \leftarrow$ get_global_id(0)
if $M[tid] \neq 0$ then
    $M[tid] \leftarrow 0$
    for all neighbors $nid$ of $tid$ from $T$ do
        $C_{new} \leftarrow (C[tid] \gg 20 + W[nid]) \ll 20 \vee tid$
        atomic_min($C_{temp}[nid]$, $C_{new}$)
    end for
end if
```

**Algorithm 3** OCL_K_1.2 ($\vec{M}, \vec{C}, \vec{C}_{temp}, M_{flag}$)

```
$tid \leftarrow$ get_global_id(0)
if $C[tid] > C_{temp}[tid]$ then
    $C[tid] \leftarrow C_{temp}[tid]$
    $M[tid] \leftarrow M_{flag} \leftarrow 1$
end if
$C_{temp}[tid] \leftarrow C[tid]$
```

Due to our custom representation of the cost array, the new cost $C_{new}$ is computed using the bitwise operations *shift* and *or*. The $C_{new}$ is not reflected in the cost array but is updated in the temporary array $\vec{C}_{temp}$ through the atomic OpenCL operation `atomic_min`. The atomic operation compares the two costs (passed as parameters) and stores, in the location of the first argument, the minor value. The second kernel, shown in Algo. 3, compares cost $\vec{C}$ with temporary cost $\vec{C}_{temp}$. It updates the cost $\vec{C}$ only if it is bigger than $\vec{C}_{temp}$ and marks the corresponding entry, in the mask $\vec{M}$, as "*to be explored*". Therefore, the $M_{flag}$ is set to produce another kernels round. The temporary cost array reflects the cost array after each kernel execution for consistency. The second kernel is required as there is no synchronization between OpenCL WGs. Updating the cost at the time of modification itself can result in *read-after-write* inconsistencies.

### 4.2.2 Parallel Multi-Buffer

In this implementation we prevent race conditions by allowing extra-size temporary cost buffer. For each vertex that can be reached by more than one edge (i.e. 8 edges), we store the costs in continuous locations of $\vec{C}_{temp}$ (multi-buffer), as reported in Algo. 4. The storage location inside the multi-buffer is determined exploiting a Look-Up-Table (LUT), that is used only for vertex id multiple of 21. For others vertices, the new cost is always stored in the only location available, in the same way of Algo. 2.

**Algorithm 4** OCL_K_2.1 ($\vec{T}, \vec{M}, \vec{M}_{temp}, \vec{C}, \vec{C}_{temp}, \vec{W}, \vec{L}, M_{flag}$)

```
$tid \leftarrow$ get_global_id(0)
$M_{flag} \leftarrow 0$
if $M[tid] \neq 0$ then
    $M[tid] \leftarrow 0$
    for all neighbors $nid$ of $tid$ from $T$ do
        $C_{new} \leftarrow (C[tid] \gg 20 + W[nid]) \ll 20 \vee tid$
        if $C_{new} < C[nid]$ then
            $L_{id} \leftarrow 0$
            if $nid$ is multiple of 21 then
                $L_{id} \leftarrow L[tid \bmod 21]$
            end if
            $C_{temp}[nid + L_{id}] \leftarrow C_{new}$
            $M_{temp}[nid] \leftarrow 1$
        end if
    end for
end if
```

In a second kernel, Algo. 5, for each vertex that can be reached by more than one edge, is selected the minimum cost stored in the temporary cost array among its values. The remaining kernel's operations are the same described in the *Parallel Atomic* version. Also in this case, the second stage of the kernel execution is required because there is no synchronization between the OpenCL WGs.

**Algorithm 5** OCL_K_2.2 ($\vec{M}, \vec{M}_{temp}, \vec{C}, \vec{C}_{temp}, M_{flag}$)

```
$tid \leftarrow$ get_global_id(0)
if $M_{temp}[tid] = 1$ then
    $M_{temp}[tid] \leftarrow 0$
    $C[tid] \leftarrow$ min($C_{temp}[tid+0]$, ..., $C_{temp}[tid+7]$)
    $M[tid] \leftarrow 1$
    $M_{flag} \leftarrow 1$
end if
```

**Algorithm 6** OCL_K_3.1 ($\vec{T}, \vec{M}, \vec{C}, \vec{W}, M_{flag}$)

```
$tid \leftarrow$ get_global_id(0)
$M_{flag} \leftarrow 0$
if $M[tid] \neq 0$ then
    $M[tid] \leftarrow 0$
    for all neighbors $nid$ of $tid$ from $T$ do
        $C_{new} \leftarrow (C[tid] \gg 20 + W[nid]) \ll 20 \vee tid$
        if $C_{new} < C[nid]$ then
            $C[nid] \leftarrow C_{new}$
            $M[nid] \leftarrow M_{flag} \leftarrow 1$
        end if
    end for
end if
```

### 4.2.3 Parallel Non-Deterministic

The last implementation, shown in Algo. 6, allows race conditions with no guaranties about the path optimality. Only one kernel is needed to perform the algorithm and the operations here executed are the same presented in the the first *Parallel Atomic* version. The main difference, respect to others versions, is that we do not use temporary cost array $\vec{C}_{temp}$. Indeed, we store the cost directly in the final cost array $\vec{C}$, incurring in possible *read-after-write* inconsistencies.

# 5. RESULTS

We tested our parallel (GPU) implementations against a sequential (CPU) Dijkstra algorithm. Performed experiments measure the energy-efficiency and the speed-up brought by the proposed parallel versions and show the maximum map size supported to enable real-time computation. Finally, is carried out an experiment in order to evaluate the precision, in terms of path optimality, for the *Non-Deterministic* version. Due to the intrinsic nature of this last version, all the relative measurements (i.e. execution time and path optimality) are presented as the average of multiple executions (1000 runs). We tested all our parallel implementations against the simplest sequential implementation of Dijkstra algorithm. The sequential implementation runs on one CPU core and has time complexity $\mathcal{O}(|V|\log|V| + |E|)$. The specific devices configurations employed as detailed in Sec. 3 is summarized in Table 1.

| | Ivy Bridge | HD Graphics 4000 |
|---|---|---|
| # Cores | 2 | 128 |
| Core Frequency | 2.50 GHz | 650 MHz |
| L3 Cache | 3072 KB | 256 KB |
| System DDR Memory | 8192 MB | |

**Table 1:** Devices for the experiments.

## 5.1 CPU vs. GPU evaluation

We present quantitative results by measuring the energy-efficiency, the overall speed-up and the execution time of all proposed implementations. All experiments are presented with a growing squared map resolution (x-axis), from 10x10 to 100x100 discretization size, increasing by ten both dimensions at each step.

In Tab. 2, we report the power consumption measured with the Intel *Power Governor*[3] tool. The considered power domains are represented by the *Core*, the *Graphics* accelerator, the last level cache and memory controller (i.e. *Un-Core*) and the *Package* that is approximatively the sum of all the previous. All the reported measurements are given as average values over multiple iterations. As can be seen from Fig. 4, the energy saved by the parallel implementations, w.r.t. the sequential one, is between 95 and 98%. The *Non-Deterministic* version exhibits the best behavior requiring, at most, 3.4 J. If we consider these results in relation with the package power consumption reported in Tab. 2, is clear how the main contribution in the energy saving is given by the reduced execution time of the parallel implementations, as also shown in Fig. 6.

| | Package | Core | Graphics | Un-Core |
|---|---|---|---|---|
| Sequential | 10.65 | 8.11 | 0.27 | 2.27 |
| Non-Deterministic | 12.39 | 5.58 | 4.10 | 2.71 |
| Atomic | 8.96 | 3.21 | 3.33 | 2.43 |
| Multi-Buffer | 11.79 | 4.60 | 4.66 | 2.53 |

**Table 2:** Power consumption, in Watt, per power domain.

The speed-up, in Fig. 5, is computed as the ratio between the CPU measured execution time and the GPU one.

Results denote the efficiency of the proposed GPU optimizations, with peak speed-ups of 20x and 21x for deterministic versions (respectively *Atomic* and *Multi-Buffer*). The best performance is reached by the *Non-Deterministic* version with a speed-up of 74x. Both deterministic implementations benefit less from the parallel optimizations. On one hand, for the *Atomic* version, the limited performance gain is ascribed to the low efficiency of OpenCL atomic operation used. Moreover, when more threads execute the *atomic_min*, addressing the same memory location, the execution is serialized. On the other hand, for the *Multi-Buffer* version, we pay for extra writes in global memory and for additional operations such as *modulo* operation to compute the LUT index. The *Non-Deterministic* version, does not suffer from any of the discussed limitations and it uses only one kernel.

In the third experiment (Fig. 6) we investigate the maximum supported map size enabling real-time computation, defined as computation under a given upper bound (i.e. 250 [4] ms). The *Non-Deterministic* version is able to compute maps of 10000 elements (207612 vertices) in real-time. *Atomic* and *Multi-Buffer* versions are respectively able to compute maps having 1600 and 2500 elements (32472 and 51312 vertices) respecting the time constraint. As the quadrotor diameter is around 0.5 m, the map grid used has a cell size between 0.5 m$^2$ and 2 m$^2$, thus we are able to perform real-time planning of environments up to 200 m$^2$.

Results in Fig. 5 and 6 are based on the worst case in term of computational requirements: the absence of obstacles. In fact, as explained in Sec. 4.1, the more obstacles are present in the map the less vertices are in the graph.

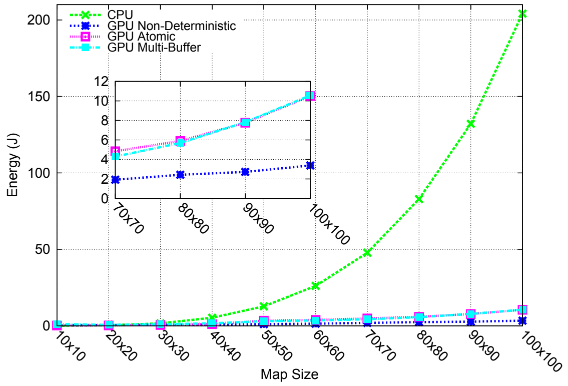

**Figure 4:** Energy-efficiency of the evaluated implementations.

## 5.2 Accuracy evaluation

In the experiment in Fig. 7, we evaluate the loss in optimality for the fastest *Non-Deterministic* implementation with respect to deterministic versions.

Results cover four different scenarios: *Random Path - 0% obstacles*, *Random Path - 25% obstacles*, *Random Path - 50% obstacles*, and *Diagonal Path - 0% obstacles*. *Random Path* means that the source and destination points are selected randomly from any available location in the map. *Diagonal Path* means cross the entire map, from a corner to the opposite one (i.e. the longest path evaluated). The percentage of obstacles represents the number of random forbidden locations in the map. Thus, introducing more obstacles means reducing the number of feasible paths, therefore the convergence between the non-deterministic and the deterministic path is increased. On the x-axis the map discretized resolution is reported, as in previous experiments.

The error is computed as the average of the percentage cost error with respect to the minimal optimal path cost, over 1000 iterations. Fig. 7 also shows, for each average error, the minimum and maximum percentage error retrieved. The minimum error is always equal to zero except for two diagonal configurations (i.e. 50x50 and 100x100), where it is of 0.68% and 1.0%, respectively. The largest error is always lower than 5.3% and is tied to a high variance. Finally, for all proposed configurations, the average error is always below

---

[3] https://software.intel.com/en-us/articles/intel-power-governor

[4] We consider a flight speed of 4 m/s and only 1 meter to prevent a collision with a dynamic obstacle (i.e. worst case).

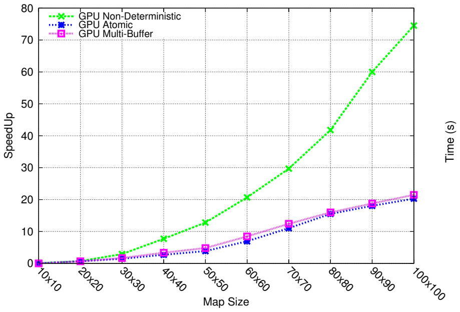

**Figure 5:** Speed-up yielded by the proposed GPU versions over the CPU implementation.

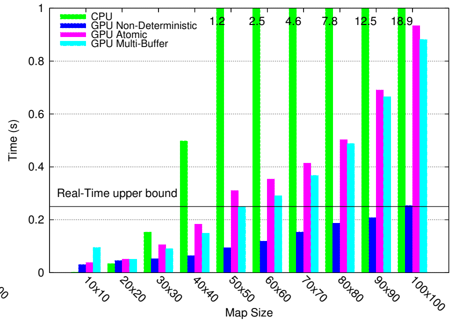

**Figure 6:** Measured execution times, with the real-time upper-bound.

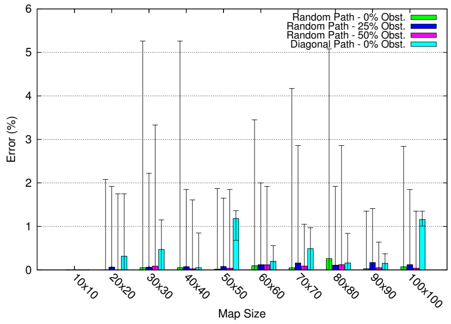

**Figure 7:** Path optimality error (%) for the Non-Deterministic version.

1.2%. As expected, the highest average error rate is related to the longest (diagonal), obstacle-free configurations, which represent the scenario with the highest number of feasible paths. The presented results enable the *Non-Deterministic* version to be a good trade-off between computing performance and path optimality, giving us always a feasible path (i.e. obstacle-free) with an average cost never higher than 1.2% w.r.t. the optimal cost.

# 6. CONCLUSION

This paper has proposed an energy-efficient shortest trajectory planning implementation, based on Dijkstra's algorithm, which exploits the Intel's HD Graphics accelerator for multi-rotor UAVs used in rescue missions, increasing *see-and-listen* capability of human operators.

Proposed parallel solutions show a saving of energy up to 98% of the energy required by the sequential version. Substantial speed-ups have been achieved for all proposed versions, up to 74x for the *non-deterministic* and 21x for the *deterministic* version, both with respect to the same CPU-based implementation. Selecting the cell size for the map discretization of 1 $m^2$, the proposed solutions allow the computation of 100 $m^2$ and 50 $m^2$ map sizes in real-time, for the non-deterministic and the deterministic version respectively. Moreover, the *non-deterministic* method proposed shows a maximum average error up to 1.2% w.r.t. the optimal cost. Finally, we demonstrate, that even if in the current UAV configuration the power consumption of the robot's engine is dominant w.r.t. the control system, the computational requirements are achieved in a very energy-efficient manner. This will be key to achieving advanced cognitive skills on heavily resource-constrained small-scale UAVs, where the power spent on the propellers is same-level with (or smaller than) that spent on the processing elements.

# 7. ACKNOWLEDGMENTS

This work has been supported by European Community under the H2020 project HERCULES (688860).

# 8. REFERENCES

[1] J. Barraquand, B. Langlois, and J.-C. Latombe. Numerical potential field techniques for robot path planning. October 1989.

[2] R. Bruhwiler, B. Goldberg, N. Doshi, O. Ozcan, N. Jafferis, M. Karpelson, and R. Wood. Feedback control of a legged microrobot with on-board sensing. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 5727–5733, Sept 2015.

[3] A. Buluc, J. R. Gilbert, and C. Budak. Solving path problems on the gpu. *Parallel Comput.*, 36(5-6):241–253, jun 2010.

[4] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2008.

[5] H. Cover, S. Choudhury, S. Scherer, and S. Singh. Sparse tangential network (spartan): Motion planning for micro aerial vehicles. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2820 – 2825. IEEE, 2013.

[6] A. Davidson, S. Baxter, M. Garland, and J. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 349–359, May 2014.

[7] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. Phast: Hardware-accelerated shortest path trees. Technical Report MSR-TR-2010-125, 2010.

[8] E. Dijkstra. A note on two problems in connexion with graph. *Numerische Mathematik*, 1:269–271, 1959.

[9] M. Furci, A. Paoli, and R. Naldi. A supervisory control strategy for robot-assisted search and rescue in hostile environments. In *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–4, Sept 2013.

[10] C. Goerzen, Z. Kong, and B. Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57(1-4):65–100, 2010.

[11] L. Heng, L. Meier, P. Tanskanen, F. Fraundorfer, and M. Pollefeys. Autonomous obstacle avoidance and maneuvering on a vision-guided mav using on-board processing. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 2472 – 2477. IEEE, 2011.

[12] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. Octomap: An efficient probabilistic 3D mapping framework based on octress. *Autonomous Robots*, 34:189–206, 2013.

[13] S. Hrabar. Reactive obstacle avoidance for rotorcraft uavs. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 4967–4974, Sept 2011.

[14] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006.

[15] S. M. LaValle. Motion planning: The essentials. *IEEE Robotics and Automation Society Magazine*, 18(1):79–89, 2011.

[16] E. Mueggler, M. Faessler, F. Fontana, and D. Scaramuzza. Aerial-guided navigation of a ground robot among movable obstacles. In *Safety, Security, and Rescue Robotics (SSRR), 2014 IEEE International Symposium on*, 2014.

[17] M. Nieuwenhuisen and S. Behnke. Hierarchical planning with 3d local multiresolution obstacle avoidance for micro aerial vehicles. In *Proceedings of Joint 45th International Symposium on Robotics (ISR)*, June 2014.

[18] J. Nikolic, M. Burri, J. Rehder, S. Leutenegger, C. Huerzeler, and R. Siegwart. A uav system for inspection of industrial facilities. In *Aerospace Conference, 2013 IEEE*, pages 1–8, March 2013.

[19] K. Ok, S. Ansari, B. Gallagher, W. Sica, F. Dellaert, and M. Stilman. Path planning with uncertainty: Voronoi uncertainty fields. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 4596–4601, May 2013.

[20] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[21] D. Scaramuzza, et al., M. W. Achtelik, L. Heng, G. H. Lee, S. Lynen, L. Meier, M. Pollefeys, R. Siegwart, and P. Tanskanen. Vision-Controlled Micro Flying Robots: From System Design to Autonomous Navigation and Mapping in GPS-Denied Environments. *IEEE robotics & automation magazine*, 21(3):26–40, Sept. 2014.

[22] F. Steinbrucker, J. Sturm, and D. Cremers. Volumetric 3d mapping in real-time on a cpu. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 2021–2028, May 2014.

[23] R. Wood, B. Finio, M. Karpelson, K. Ma, N. Pérez-Arancibia, P. Sreetharan, H. Tanaka, and J. Whitney. Progress on 'pico' air vehicles. *Int. J. Rob. Res.*, 31(11):1292–1302, Sept. 2012.