A 64-mW DNN-Based Visual Navigation Engine for Autonomous Nano-Drones

Daniele Palossi, Antonio Loquercio, Francesco Conti, *Member, IEEE*, Eric Flamand, Davide Scaramuzza, *Member, IEEE*, and Luca Benini, *Fellow, IEEE*

***Abstract*—Fully autonomous miniaturized robots (e.g., drones), with artificial intelligence (AI)-based visual navigation capabilities, are extremely challenging drivers of Internet-of-Things edge intelligence capabilities. Visual navigation based on AI approaches, such as deep neural networks (DNNs) are becoming pervasive for standard-size drones, but are considered out of reach for nano-drones with a size of a few cm^2. In this paper, we present the first (to the best of our knowledge) demonstration of a navigation engine for autonomous nano-drones capable of closed-loop end-to-end DNN-based visual navigation. To achieve this goal we developed a complete methodology for parallel execution of complex DNNs directly on board resource-constrained milliwatt-scale nodes. Our system is based on GAP8, a novel parallel ultralow-power computing platform, and a 27-g commercial, open-source Crazyflie 2.0 nano-quadrotor. As part of our general methodology, we discuss the software mapping techniques that enable the DroNet state-of-the-art deep convolutional neural network to be fully executed aboard within a strict 6 frame-per-second real-time constraint with no compromise in terms of flight results, while all processing is done with only 64 mW on average. Our navigation engine is flexible and can be used to span a wide performance range: at its peak performance corner, it achieves 18 frames/s while still consuming on average just 3.5% of the power envelope of the deployed nano-aircraft. To share our key findings with the embedded and robotics communities and foster further developments in autonomous nano-unmanned aerial vehicles (UAVs), we publicly release all our code, datasets, and trained networks.**

***Index Terms*—Autonomous UAV, CNNs, end-to-end learning, nano-UAV, ultralow-power.**

Demo Video & Open Source Release

Demonstrative video at: https://youtu.be/57Vy5cSvnaA.

The project's code, datasets and trained models are available at: https://github.com/pulp-platform/pulp-dronet.

Manuscript received February 4, 2019; revised April 19, 2019; accepted April 29, 2019. Date of publication May 15, 2019; date of current version October 8, 2019. This work was supported by EC H2020 OPRECOMP Project under Grant 732631 and ALOHA Project under Grant 780788, in part by the Swiss National Center of Competence Research Robotics, and in part by the SNSF-ERC Starting Grant. *(Corresponding author: Daniele Palossi.)*

D. Palossi is with the Integrated Systems Laboratory, ETH Zürich, 8092 Zürich, Switzerland (e-mail: dpalossi@iis.ee.ethz.ch).

A. Loquercio and D. Scaramuzza are with the Robotic and Perception Group, Department of Informatics, University of Zürich, 8006 Zürich, Switzerland, and also with the Department of Neuroinformatics, University of Zürich, 8006 Zürich, Switzerland, and ETH Zürich, 8050 Zürich, Switzerland.

F. Conti and L. Benini are with the Integrated Systems Laboratory, ETH Zürich, 8092 Zürich, Switzerland, and also with the Department of Electrical, Electronic and Information Engineering, University of Bologna, 40136 Bologna, Italy (e-mail: f.conti@unibo.it, luca.benini@unibo.it).

E. Flamand is with the Integrated Systems Laboratory, ETH Zürich, 8092 Zürich, Switzerland, and also with GreenWaves Technologies, 38190 Villard-Bonnot, France (e-mail: eflamand@iis.ee.ethz.ch).

Digital Object Identifier 10.1109/JIOT.2019.2917066

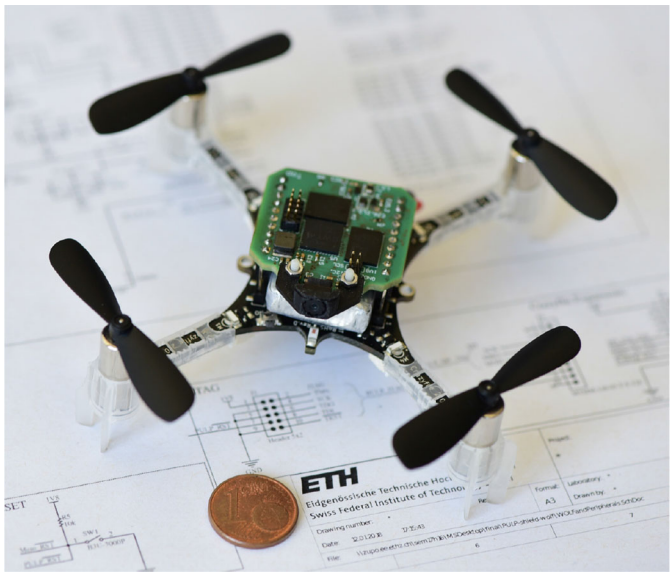

Fig. 1. Our prototype based on the COTS *Crazyflie 2.0* nano-quadrotor extended with our *PULP-Shield*. The system can run the *DroNet* [2] CNN for autonomous visual navigation up to 18 frames/s using only *onboard* resources.

I. Introduction

With the rise of the Internet-of-Things (IoT) era and rapid development of artificial intelligence (AI), embedded systems *ad-hoc* programmed to act in relative isolation are being progressively replaced by AI-based sensor nodes that acquire information, process, and understand it, and use it to interact with the environment and with each other. The "ultimate" IoT node will be capable of autonomously navigating the environment and, at the same time, sensing, analyzing, and understanding it [1].

Fully autonomous nano-scale unmanned aerial vehicles (UAVs) are befitting embodiments for this class of smart sensors: with their speed and agility, they have the potential to quickly collect information from both their onboard sensors and from a plethora of devices deployed in the environment. Nano-UAVs could also perform advanced onboard analytics, to preselect essential information before transmitting it to centralized servers [3]. The tiny form-factor of nano-drones is ideal both for indoor applications where they should safely operate near humans (for surveillance, monitoring, ambient awareness, interaction with smart environments, etc.) [4] and for highly populated urban areas, where they can exploit complementary

TABLE I
ROTORCRAFT UAVS TAXONOMY BY VEHICLE CLASS-SIZE

Vehicle Class	⌀ : Weight [cm:kg]	Power [W]	Onboard Device
std-size [11]	$\sim 50 : \geq 1$	≥ 100	Desktop
micro-size [12]	$\sim 25 : \sim 0.5$	~ 50	Embedded
nano-size [13]	$\sim 10 : \sim 0.01$	~ 5	MCU
pico-size [14]	$\sim 2 : \leq 0.001$	~ 0.1	ULP

sense-act capabilities to interact with the surroundings (e.g., smart-building, smart-cities, etc.) [5]. As an example, in this IoT scenario, a relevant application for intelligent nano-size UAVs can be the online detection of wireless activity, from edge nodes deployed in the environment, via onboard radio packet sniffing [6].

Commercial off-the-shelf (COTS) quadrotors have already started to enter the nano-scale, featuring only few centimeters in diameter and a few tens of grams in weight [7]. However, commercial nano-UAVs still lack the autonomy boasted by their larger counterparts [2], [8]–[10], since their computational capabilities, heavily constrained by their tiny power envelopes, have been considered so far to be totally inadequate for the execution of sophisticated AI workloads, as summarized in Table I.

The traditional approach to autonomous navigation of a UAV is the so-called *localization-mapping-planning* cycle, which consists of estimating the robot motion using either off-board (e.g., GPS [15]) or onboard sensors (e.g., visual-inertial sensors [16]), building a local 3-D map of the environment, and planning a safe trajectory through it [10]. These methods, however, are very expensive for computationally constrained platforms. Recent results have shown that much lighter algorithms, based on convolutional neural networks (CNNs), are sufficient for enabling basic reactive navigation of small drones, even without a map of the environment [2], [17]–[20]. However, their computational and power needs are unfortunately still above the allotted budget of current navigation engines of nano-drones, which are based on simple, low-power microcontroller units (MCUs).

Wood *et al.* [14] indicated that, for small-size UAVs, the maximum power budget that can be spent for onboard computation is 5% of the total, the rest being used by the propellers (86%) and the low-level control parts (9%). The problem of bringing state-of-the-art (SoA) navigation capabilities on the challenging classes of nano- and pico-size UAVs is therefore strictly dependent on the development of energy-efficient and computationally capable hardware, highly optimized software and new classes of algorithms combined into a next-generation navigation engine. These constraints and requirements depict the same scenario faced in deploying high-level computation capabilities on IoT edge-nodes/sensors. Moreover, in the case of a flying miniature robot, the challenge is exacerbated by the strict real-time constraint dictated by the need for fast reaction time to prevent collisions with dynamic obstacles.

Whereas standard-size UAVs with a power envelope of several hundred Watts have always been able to host powerful high-end embedded computers like Qualcomm Snapdragon,[1] Odroid, NVIDIA Jetson TX1, and TX2, etc., most nano-sized UAVs have been constrained by the capabilities of microcontroller devices capable of providing a few hundred Mop/s at best. Therefore, CNN-based autonomous vision navigation was so far considered to be out of reach for this class of drones.

In this paper, we propose a novel visual navigation engine and a general methodology to deploy complex CNN on top of COTS resources-constrained computational edge-nodes such as a nano-size flying robot. We present what, to the best of our knowledge, is the first deployment of a SoA, fully autonomous vision-based navigation system based on deep learning on top of a UAV visual navigation engine consuming less than 284 mW at peak (64 mW in the most energy-efficient configuration), fully integrated and in closed-loop control within an open source COTS *Crazyflie 2.0* nano-UAV. Our visual navigation engine, shown on the top of the *Crazyflie 2.0* in Fig. 1, leverages the *GreenWaves Technologies GAP8* SoC, a high-efficiency embedded processor taking advantage of the emerging parallel ultralow-power (PULP) computing paradigm to enable the execution of complex algorithmic flows onto power-constrained devices, such as nano-scale UAVs.

This paper provides several contributions beyond the SoA of nano-scale UAVs and serves as a proof-of-concept for a broader class of AI-based applications in the IoT field. In this paper:

1) we developed a general methodology for deploying SoA deep learning algorithms on top of ultralow-power embedded computation nodes, as well as a miniaturized robot;
2) we adapted *DroNet*, the CNN-based approach for autonomous navigation proposed in Loquercio *et al.* [2] for standard-sized UAVs, to the computational requirements of a nano-sized UAV, such as fixed-point computation;
3) we deployed DroNet on the *PULP-Shield*, an ultralow-power visual navigation module featuring the GAP8 SoC, an ultralow-power camera and off-chip Flash/DRAM memory; the shield is designed as a pluggable PCB for the 27 g COTS *Crazyflie 2.0* nano-UAV;
4) we demonstrate our methodology for the DroNet CNN, achieving a comparable quality of results in terms of UAV control with respect to the standard-sized baseline [2] within an overall PULP-Shield power budget of just 64 mW, delivering a throughput of 6 frames/s and up to 18 frames/s within 284 mW;
5) we field-prove our methodology presenting a closed-loop fully working demonstration of vision-driven autonomous navigation relying only on onboard resources.

This paper demonstrates that parallel ultralow-power computing is a viable solution to deploy autonomous navigation capabilities on board nano-UAVs used as smart, mobile IoT

[1] [Online]. Available: https://developer.qualcomm.com/hardware/qualcomm-flight

end-nodes, while at the same time showcasing a complete hardware/software methodology to implement such complex workloads on a heavily power- and memory-constrained device. We prove in the field the efficacy of our methodology by presenting a closed-loop fully functional demonstrator in the supplementary material. To foster further research on this field, we release the PULP-Shield design and all code running on GAP8, as well as datasets and trained networks, as publicly available under liberal open-source licenses.

The rest of this paper is organized as follows. Section II provides the SoA overview both in terms of nano-UAVs and low-power IoT. Section III introduces the software/hardware background of this paper. Section IV presents in detail our CNN mapping methodology, including software tools and optimizations. Section V discusses the design of the visual navigation engine. Section VI-B shows the experimental evaluation of the work, considering both performance and power consumption, comparing our results with the SoA and also evaluating the final control accuracy. Finally, Section VII concludes this paper.

II. Related Work

The development of the IoT is fueling a trend toward edge computing, improving scalability, robustness, and security [1]. While today's IoT edge nodes are usually stationary, autonomous nano-UAVs can be seen as perfect examples of next-generation IoT end-nodes, with high mobility and requiring an unprecedented level of onboard intelligence. The goal of this paper is to make SoA visual autonomous navigation compatible with ultralow-power nano-drones, unlocking their deployment for IoT applications. Therefore, this section focuses on related work on nano-aircrafts [14] and the deployment of deep neural network (DNN) on top of low-power IoT nodes.

The traditional approach to autonomous navigation of nano-drones requires to offload computation to some remote, powerful base-station. For instance, Dunkley *et al.* [21] developed a visual-inertial simultaneous localization and mapping (SLAM) algorithm, for a 25 g nano quadrotor. The SLAM algorithm was used to stabilize the robot and follow a reference trajectory. All the computation was performed off-board, by streaming video and inertial information from the drone to a remote, power-unconstrained laptop. The main problems with this class of solutions are latency, maximum communication distance, reliability issues due to channel noise, and high onboard power-consumption due to the high-frequency video streaming.

Few previous works presented nano-size flying robots with some degree of autonomous navigation relying on onboard computation. McGuire *et al.* [13] developed a 4 g stereo-camera and proposed a velocity estimation algorithm able to run on the MCU on board a 40 g flying robot. If on one side this solution allows the drone to avoid obstacles during the flight, it still requires favorable flight conditions (e.g., low-flight speed of 0.3 m/s). In [22], an optical-flow-based guidance system was developed for a 46 g nano-size UAV. The proposed ego-motion estimation algorithm did not rely on feature tracking, making it possible to run on the onboard MCU. Unfortunately, the autonomous functionality was limited to hovering, and the method did not reach the accuracy of computationally expensive techniques based on feature tracking.

In [23], an application-specific integrated circuit (ASIC), called *NAVION*, for onboard visual-inertial odometry was presented. Although, this chip exposes enough computational power to perform state estimation up to 171 frames/s within 24 mW, this represents only one among other basic functionalities required by any UAV to be fully autonomous. Therefore, in a real use case, the proposed ASIC would still need to be paired with additional circuits, both for complementary onboard computation as well as for interacting with the drone's sensors. Moreover, to the date, the *NAVION* accelerator does not reach the same level of maturity and completeness of this paper; in fact, *NAVION* has not yet been demonstrated on a real-life flying nano-drone.

COTS nano-size quadrotors, like the *Bitcraze Crazyflie 2.0* or the *Walkera QR LadyBug*, embed on board low-power single core MCUs, like the *ST Microelectronics* STM32F4 [13], [21], [24]. While significant work has been done within academia [25]–[27] and industry (e.g., TensorFlow Lite[2] and ARM Compute Library[3]) to ease the embedding of DNNs on mobile ARM-based SoC's, there is no general consensus yet on how to "correctly" deploy complex AI-powered algorithms, such a DNNs, on this class of low-power microcontrollers. This is a "hard" problem both in terms of resource management (in particular, available working memory and storage) and the peak throughput achievable by single core MCUs. This problem is furthermore exacerbated by a lack of abstraction layers and computing facilities that are taken from granted by common deep learning tools, such as linear algebra libraries (e.g., BLAS, CUBLAS, and CUDNN) and preprocessing libraries (e.g., OpenCV).

ARM has recently released CMSIS-NN [28], which is meant to shrink this gap by accelerating deep inference compute kernels on Cortex-M microcontroller platforms, providing the equivalent of a BLAS/CUDNN library (in Section VI-B, we present a detailed SoA comparison between our results and CMSIS-NN). However, this effort does not curtail the difficulty of effectively deploying DNNs in memory-scarce platforms, which often requires particular scheduling/tiling [29], [30] and is still widely considered an open problem.

Pushing beyond the aforementioned approaches, in this paper we propose and demonstrate a visual navigation engine capable of sophisticated workloads, such as real-time CNN-based autonomous visual navigation [2], entirely aboard within the limited power envelope of nano-scale UAVs ($\sim$0.2 W). Such a kind of autonomous navigation functionality has been previously limited to standard-sized UAVs, generally equipped with power-hungry processors ($\geq$ 10 W) or relying on external processing and sensing (e.g., GPS) [6]. Our system relaxes both requirements: we use an onboard ultralow-power processor and a learning-based navigation approach.

[2][Online]. Available: https://www.tensorflow.org/lite

[3][Online]. Available: https://arm-software.github.io/ComputeLibrary

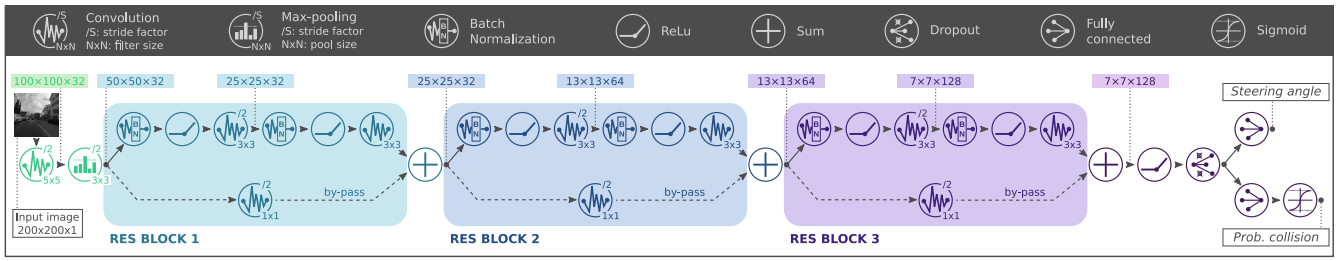

Fig. 2. *DroNet* [2] topology.

III. Background

In this section, we summarize the hardware/software background underlying our visual navigation engine. We first present the original *DroNet* CNN developed for standard-size UAVs. Then, we introduce the GAP8 SoC used on board of our nano-drone.

A. DroNet

DroNet is a lightweight residual CNN architecture. By predicting the steering angle and the collision probability, it enables safe autonomous flight of a quadrotor in various indoor and outdoor environments.

The DroNet topology, as illustrated in Fig. 2, was inspired by residual networks [31] and was reduced in size to minimize the bare image processing time (inference). The two tasks of steering and collision probability prediction share all the residual layers to reduce the network complexity and the frame processing time. Then, two separate fully connected layers independently infer steering and collision probabilities. Mean-squared error (MSE) and binary cross-entropy (BCE) have been used to train the two predictions, respectively. A temporal dependent weighting of the two losses ensures the training convergence despite the different gradients' magnitude produced by each loss. Eventually, to make the optimization focus on the samples that are most difficult to learn, hard negative mining was deployed in the final stages of learning. The two tasks learn from two separate datasets. Steering angle prediction was trained with the *Udacity* dataset,[4] while the collision probability was trained with the *Zürich bicycle* dataset.[5]

The outputs of DroNet are used to command the UAV to move on a plane with velocity in forwarding direction v_k and steering angle θ_k. More specifically, the low-pass filtered probability of collision is used to modulate the UAV forward velocity, while the low-pass filtered steering angle is converted to the drone's yaw control. The result is a single relatively shallow network that processes all visual information and directly produces control commands for a flying drone. Learning the coupling between perception and control end-to-end provides several advantages, such as a simple, lightweight system, and high generalization abilities. Indeed, the method was shown to function not only in urban environments but also on a set of new application spaces without any initial knowledge about them [2]. More specifically, even without a map of the environment, the approach generalizes very well to scenarios completely unseen at training time, including indoor corridors, parking lots, and high altitudes.

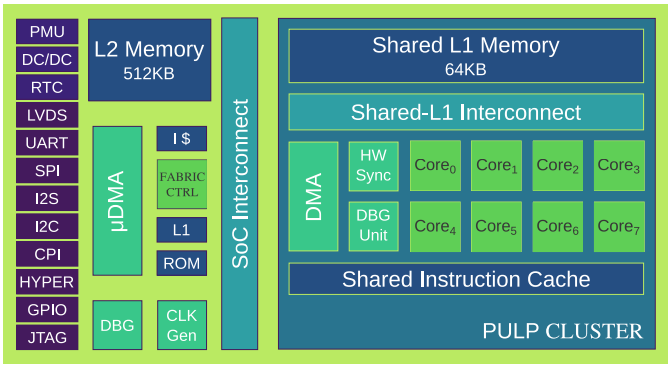

Fig. 3. Architecture of the GAP8 embedded processor.

B. GAP8 Architecture

Our deployment target for the bulk of the DroNet computation is GAP8, a commercial embedded RISC-V multicore processor derived from the PULP open source project.[6] At its heart, GAP8 is composed by an advanced RISC-V MCU coupled with a programmable octa-core accelerator with RISC-V cores enhanced for digital signal processing and embedded deep inference.

Fig. 3 shows the architecture of GAP8 in detail. The processor is composed of two separate power and clock domains, the FABRIC CTRL (FC) and the CLUSTER (CL). The FC is an advanced MCU featuring a single RISC-V core coupled with 512 KB of SRAM (*L2 memory*). The FC uses an in-order, DSP-extended four-stage microarchitecture implementing the RISC-V instruction set architecture [32]. The core supports the RV32IMC instruction set consisting of the standard ALU instructions plus the multiply instruction, with the possibility to execute compressed code. In addition to this, the core is extended to include a register–register multiply accumulate instruction, packed single instruction multiple data (SIMD) DSP instructions (e.g., fixed-point dot product), bit manipulation instructions, and two hardware loops. Moreover, the SoC features an autonomous multichannel I/O DMA controller (μDMA) [33] capable of transferring data between

[4][Online]. Available: https://www.udacity.com/self-driving-car
[5][Online]. Available: http://rpg.ifi.uzh.ch/dronet.html
[6][Online]. Available: http://pulp-platform.org

a rich set of peripherals (QSPI, I2S, I2C, HyperBus, and camera parallel interface) and the *L*2 memory with no involvement of the FC. The HyperBus and QSPI interfaces can be used to connect GAP8 with an external DRAM or flash memory, effectively extending the memory hierarchy with an external *L*3 having a bandwidth of 333 MB/s and capacity up to 128 Mbit. Finally, the GAP8 SoC also includes a dc/dc converter converting the battery voltage down to the required operating voltage directly on-chip, as well as two separate frequency-locked loops (FLLs) for ultralow-power clock generation [34].

The CL is dedicated to the acceleration of computationally intensive tasks. It contains eight RISC-V cores (identical to the one used in the FC) sharing a 64 KB multibanked *shared L*1 *scratchpad memory* through a low-latency, high-throughput logarithmic interconnect [35]. The shared *L*1 memory supports single-cycle concurrent access from different cores requesting memory locations on separate banks and a starvation-free protocol in case of bank contentions (typically $<10\%$ on memory-intensive kernels). The eight cores are fed with instruction streams from a single shared, multiported I-cache to maximize the energy efficiency on the data-parallel code. A cluster *DMA* controller is used to transfer data between the shared *L*1 scratchpad and the *L*2 memory; it is capable of 1-D and 2-D bulk memory transfer on the *L*2 side (only 1-D on the *L*1 side). A dedicated *hardware synchronizer* is used to support fast event management and parallel thread dispatching/synchronization to enable ultrafine grain parallelism on the cluster cores. CL and FC share a single address space and communicate with one another utilizing two 64-bit AXI ports, one per direction. A software runtime resident in the FC overviews all tasks offloaded to the CL and the μ*DMA*. On a turn, a low-overhead runtime on the CL cores exploits the hardware synchronizer to implement shared-memory parallelism in the fashion of OpenMP [36].

IV. CNN Mapping Methodology

In this section, we discuss and characterize the main methodological aspects related to the deployment of *DroNet* on top of the GAP8 embedded processor. This task showcases all the main challenges for a typical deep learning application running on resource-constrained embedded IoT node. Therefore, while our visual navigation engine is application-specific, the underlying methodology we present in the following of this section is general and could also be applied to other resource-bound embedded systems where computationally intensive tasks have to be performed under a real-time constraint on a parallel architecture.

A. Deploying DroNet on GAP8

Following an initial characterization phase, we calculated the original CNN to involve $\sim$41 MMAC operations per frame (accounting only for convolutional layers) and more than 1 MB of memory needed solely to store the network's weights, yielding a baseline for the number of resources required on our navigation engine.[7] To successfully deploy the CNN on top of GAP8, the execution of DroNet has to fit within the strict real-time constraints dictated by the target application, while respecting the bounds imposed by the on-chip and onboard resources. Specifically, these constraints can be resumed in three main points.

1) The *minimum real-time frame-rate* required to select a new trajectory on-the-fly or to detect a suspected obstacle in time to prevent a potential collision.
2) The native *quality-of-results* must be maintained when using an embedded ultralow-power camera (in our prototype, the *HiMax*—see Section V for details) instead of the high-resolution camera used by the original DroNet.
3) The *amount of available memory* on the GAP8 SoC, as reported in Section III-B we can rely on 512 KB of *L*2 SRAM and 64 KB of shared *L*1 scratchpad (TCDM), sets an upper bound to the size of operating set and dictates *ad-hoc* memory management strategy.

Therefore, it is clear there is a strong need for a strategy aimed at reducing the memory footprint and computational load to more easily fit within the available resources while exploiting the architectural parallelism at best to meet the real-time constraint. The original DroNet network [2] has been modified to ease its final deployment; we operated incrementally on the model and training flow provided by the original DroNet, based on Keras/TensorFlow.[8]

The first change we performed is the reduction of the numerical representation of weights and activations from the native one, 32-bit floating point (`Float32`), down to a more economical and hardware-friendly 16-bit fixed point one (`Fixed16`) that is better suited for the deployment on any MUC-class processor without floating point unit (FPU), like in our GAP8 SoC. By analyzing the native `Float32` network post-training, we determined that a dynamic range of ± 8 is sufficient to represent all weights and intermediate activations with realistic inputs. Accordingly, we selected a `Fixed16` Q4.12 representation, using 4 bits for the integer part (including sign) and 12 bits for the fractional part of both activations and weights (rounding down to a precision of 2^{-12}). Then, we retrained the network from scratch replacing the native convolutional layers from Keras to make them "quantization-aware," using the methodology proposed by Hubara *et al.* [37].

The second significant change with respect to the original version of DroNet is the extension of the collision dataset used in [2] (named *Original* dataset) with $\sim$1300 images (1122 for training and 228 for test/validation) acquired with the same camera that is available aboard the nano-drone (named *HiMax* dataset). Fine-tuning approaches, like dataset extension, have proved to be particularly effective at improving network generalization capability [38]. In our case, the original dataset is built starting form high-resolution color cameras whose images are significantly different from the ones acquired by

[7]The baseline MMAC count does not correspond to the final implementation's instruction count, because it does not account for implementation details such as data marshaling operations to feed the processing elements; however, it can be used to set an upper bound to the minimum execution performance that is necessary to deploy DroNet at a given target frame rate.

[8][Online]. Available: https://github.com/uzh-rpg/rpg_public_dronet

TABLE II
DRONET ACCURACY ON PULP. IN BOLD THE CONFIGURATION USED FOR THE FINAL DEPLOYMENT

Training			Inference - `Fixed16`					
Dataset	Max-Pooling	Data Type	Original Dataset				HiMax Dataset	
			EVA	RMSE	Accuracy	F1-score	Accuracy	F1-score
Original	3×3	`Float32`	0.758	0.109	0.952	0.888	0.859	0.752
	3×3	`Fixed16`	0.746	0.115	0.946	0.878	0.841	0.798
	2×2	`Float32`	0.766	0.105	0.945	0.875	0.845	0.712
	2×2	`Fixed16`	0.795	0.097	0.935	0.857	0.873	0.774
Original + HiMax	3×3	`Float32`	0.764	0.104	0.949	0.889	0.927	0.884
	3×3	`Fixed16`	0.762	0.109	0.956	0.894	0.918	0.870
	2×2	`Float32`	0.747	0.109	0.964	0.916	0.900	0.831
	2×2	**`Fixed16`**	**0.732**	**0.110**	**0.977**	**0.946**	**0.891**	**0.821**

the ULP low-resolution grayscale camera available in our navigation engine, particularly in terms of contrast. Therefore, we extended the training set and we evaluate our CNN for both datasets separately. Finally, we modified the receptive field of max-pooling layers from 3×3 to 2×2, which yields essentially the same final results while reducing the execution time of max-pooling layers by $2.2\times$ and simplifying their final implementation on GAP8.

Table II summarizes the results in terms of accuracy for all these changes. Explained variance[9] (EVA) and root-MSE (RMSE) refer to the regression problem (i.e., steering angle) whereas Accuracy and F1-score[10] are related to the classification problem (i.e., collision probability), evaluated on both *Original* and *HiMax* datasets. Regarding the *Original* dataset, it is clear that the proposed modifications are not penalizing the overall network's capabilities. Moreover, fine-tuning increases performance for almost all cases (both regression and classification), considering the test on the *HiMax* dataset, there is a definite improvement in terms of collision accuracy when training is done with the extended dataset. If we consider paired configurations, the fine-tuned one based is always outperforming its counterpart, up to 8% in accuracy (i.e., max-pooling 3×3, `Fixed16`). In Table II we also highlight (in bold) the scores achieved by the final version of DroNet deployed on GAP8.

B. AutoTiler

One of the most significant constraints in ULP embedded SoC's without caches is the explicit management of the memory hierarchy; that is, how to marshal data between the bigger—and slower—memories and the smaller—but faster—ones tightly coupled to the processing elements. A common technique is *tiling* [39], which involves: 1) partitioning the input and output data spaces in portions or tiles small enough to fit within the smallest memory in the hierarchy (in our case, the shared $L1$) and 2) setting up an outer loop iterating on tiles, with each iteration comprising the loading of an input tile into the $L1$, the production of an output tile, and the storage of the output tile into the higher levels of the memory hierarchy. Tiling is particularly effective for algorithms like DNNs exposing very regular execution and data access patterns. As part of this paper, we propose a tiling methodology that optimizes memory utilization on GAP8, while at the same time relieving the user from tedious and error-prone manual coding of the tiling loop and of the data movement mechanism.

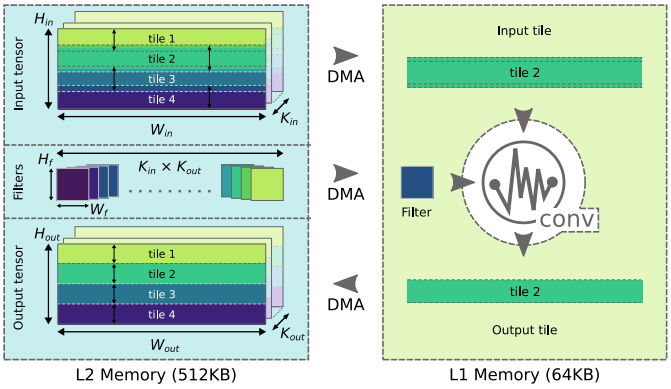

Fig. 4. Convolutional layer tiling.

Considering Fig. 4 as a reference, each layer in a CNN operates on a three-dimensional input tensor representing a *feature space* (with one *feature map* per channel) and produces a new 3-D tensor of activations as output. Convolutional layers, in particular, are composed of a linear transformation that maps K_{in} input feature maps into K_{out} output feature maps employing of $K_{\text{in}} \times K_{\text{out}}$ convolutional filters (or weight matrices). Therefore, in any convolutional layer, we can identify three different data spaces which can be partitioned in tiles in one or more of the three dimensions (i.e., W, H, and K in Fig. 4). Similar considerations can also be made for the other layers in a CNN, allowing to treating them in the same fashion.

As the design space defined by all possible tiling schemes is very large, we developed a tool called *AutoTiler* to help explore a subset of this space, choose an optimal tiling configuration, and produce C wrapping code that orchestrates the computation in a pipelined fashion as well as double-buffered memory transfers, taking advantage of the cluster DMA controller to efficiently move data between the $L2$ and $L1$ memories. The fundamental unit of computation assumed by the *AutoTiler* tool is the *basic kernel*, a function considering that all its working data is already located in the $L1$ shared memory.

[9] $\text{EVA} = [(\text{Var}[y_{\text{true}} - y_{\text{pred}}])/(\text{Var}[y_{\text{true}}])]$.
[10] $F - 1 = 2[(\text{precision} \times \text{recall})/(\text{precision} + \text{recall})]$.

```
# weight DMA-in
DMA_Copy(ŵ ← w)
for t in range(nb_tiles_H): # tiling over H
  # prologue operation (set bias value)
  ŷ ← BasicKernel_SetBias(ŷ)
  for j in range(nb_tiles_Kin): # tiling over K_in
    # input tile DMA-in
    DMA_Copy(x̂ ← x[j,t])
    for i in range(K_out):
      # body operation (convolution)
      ŷ ← BasicKernel_Conv_Spatial(ŷ)
  ŷ ← BasicKernel_ReLU(ŷ)
  # output tile DMA-out
  DMA_Copy(y[i,t] ← ŷ)
```

Listing 1. Example of *spatial* execution scheme. x, w, and y are the multidimensional input, weight, and output tensors in $L2$ memory; $\widehat{x}$, $\widehat{w}$, and $\widehat{y}$ are their respective tiles in $L1$ memory.

```
for i in range(nb_tiles_Kout): # tiling over K_out
  # weight DMA-in
  DMA_Copy(ŵ ← w[i])
  # prologue operation (set bias value)
  ŷ ←BasicKernel_SetBias(ŷ)
  for j in range(nb_tiles_Kin): # tiling over K_in
    # input tile DMA-in
    DMA_Copy(x̂ ← x[j])
    # body operation (convolution)
    ŷ ← BasicKernel_Conv_FeatWise(ŵ, x̂, ŷ)
  # epilogue operation (ReLU)
  ŷ ← BasicKernel_ReLU(ŷ)
  # output tile DMA-out
  DMA_Copy(y[i] ← ŷ)
```

Listing 2. Example of *feature-wise* execution scheme. x, w, and y are the multidimensional input, weight, and output tensors in $L2$ memory; $\widehat{x}$, $\widehat{w}$, and $\widehat{y}$ are their respective tiles in $L1$ memory.

Examples of basic kernels include convolution, max-pooling, ReLU rectification, and addition. To map the overall high-level algorithm to a set of basic kernels that operate iteratively on tiles, the *AutoTiler* introduces a second level of abstraction: the *node kernel*. The structure of the target algorithm is coded by the developer as a dependency graph, where each node (a node kernel) is a composition of one or more basic kernels together with a specification of the related iteration space over W, H, K_{in}, and K_{out}. For example, a node kernel for a convolutional layer can be composed of a first basic kernel for setting the initial bias, a central one to perform convolutions and a final one for ReLU rectification: in the *prologue*, *body*, and *epilogue*, respectively. The *AutoTiler* treats the tiling of each node kernel as an independent optimization problem constrained by the node kernel specification and the memory sizes. This approach allows to build complex execution flows reusing hand-optimized *basic kernels* and abstracting the underneath complexity from the developer.

C. Tiling, Parallelization, and Optimization

As introduced in Section III, the GAP8 SoC features $8+1$ RISC-V cores with DSP-oriented extensions. To develop an optimized, high-performance, and energy-efficient application for GAP8 and meet the required real-time constraint it is paramount that the most computationally intensive kernels of the algorithm are parallelized to take advantage of the eight-core cluster and are entirely using the available specialized instructions. For the purpose of this paper, we used the *AutoTiler* to fully implement the structure of the modified DroNet, therefore these optimization steps are reduced to hand-tuned parallelization and optimization of the basic kernels.

To exploit the available computational/memory resources at best, we constrain the *AutoTiler* to target the following general scheme: the input tensor is tiled along the H_{in} and K_{in} dimensions, while the output tensor is tiled along H_{out} and K_{out} ones. The stripes along H_{in} are partially overlapped with one another to take into account the receptive field of convolutional kernels at the tile border. Execution of the node kernel happens in either a *spatial* or *feature-wise* fashion, which differ in the ordering of the tiling loops and in the parallelization scheme

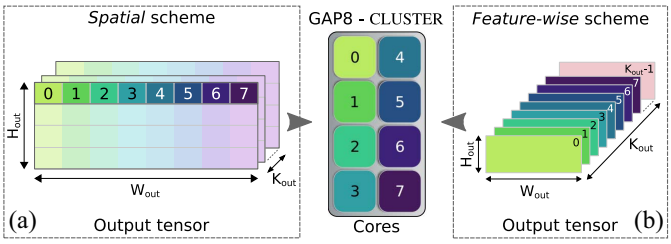

Fig. 5. Parallelization schemes utilized in the *DroNet* layers for deployment on GAP8; the different colors represent allocation to a different core.

that is applied. In the spatial scheme, work is split among parallel cores along the W_{out} dimension. Figs. 4 and 5(a) refer to this scheme, which is also exemplified in Listing 1. In the feature-wise scheme, which we only apply on full feature maps (i.e., the number of tiles in the H_{out} direction is 1), work is split among cores along the K_{out} dimension; this scheme is shown in Fig. 5(b) and Listing 2. The choice of one scheme over the other is influenced mostly by the parallelization efficiency: after an exploration phase, we found the best performance arose when using the spatial scheme for the first node kernel of DroNet (first convolution + max-pooling) while using the feature-wise approach for the rest. This choice is related to the fact that in deeper layers the feature map size drops rapidly and the spatial scheme becomes suboptimal because the width of each stripe turns too small to achieve full utilization of the cores.

To further optimize the DroNet execution, we made use of all the optimized signal processing instructions available in GAP8. These include packed-SIMD instructions capable of exploiting subword parallelism, as well as bit-level manipulation and shuffling, which can be accessed by means of compiler intrinsics such as `__builtin_pulp_dotsp2` (for 16-bit dot product with 32-bit accumulation), `__builtin_shuffle` (permutation of elements within two input vectors), and `__builtin_pulp_pack2` (packing two scalars into a vector).

D. L2 Memory Management Strategy

Given: 1) the residual-network topology of DroNet, which requires to increase the lifetime of the output tensors of some of the layers (due to bypass layers) and 2) the "scarcity" of

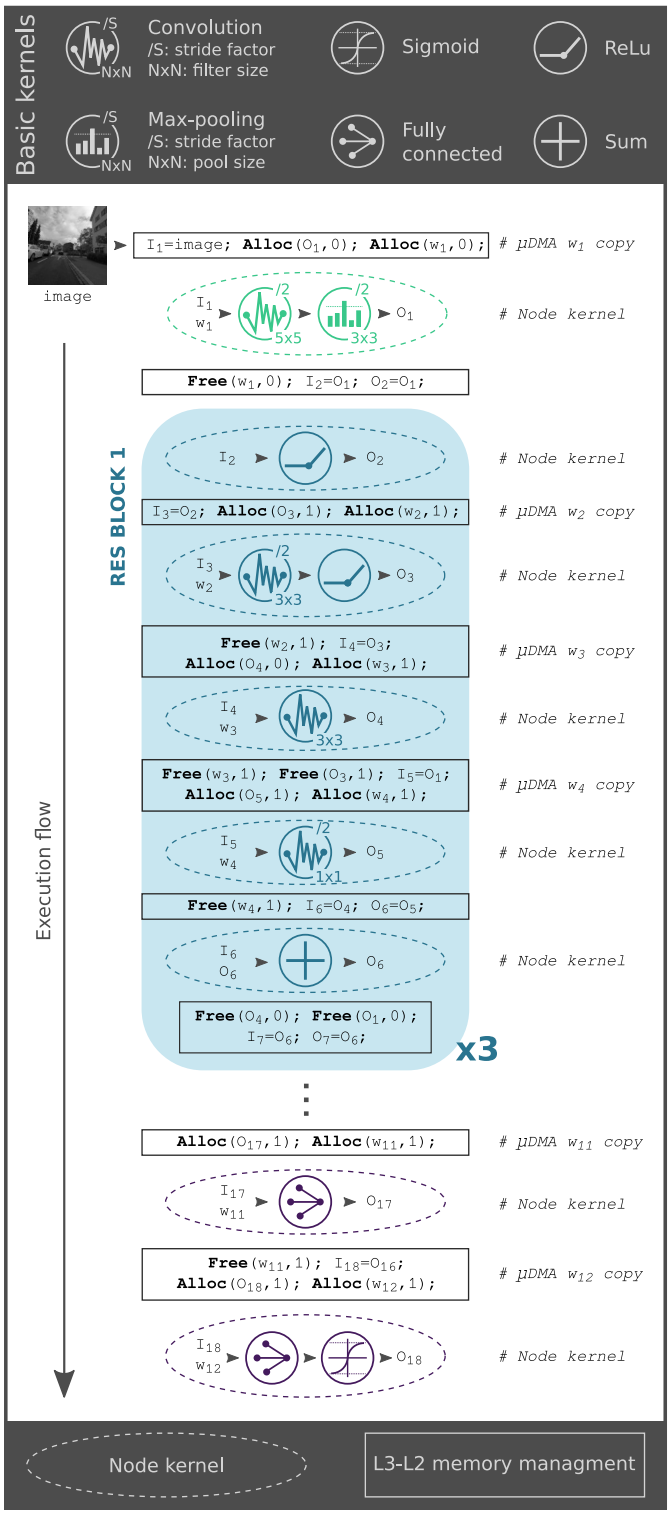

Fig. 6. *DroNet* on PULP execution graph (with pseudo-code).

L2 memory as a resource to store all weights and temporary feature maps (we would need more than 1 MB in view of 512 KB available), an *ad-hoc* memory management strategy for the *L2* memory is required, similar to what is done between *L2* and *L1* using the GAP8 *AutoTiler*. Due to the high energy cost of data transfers between *L3* and *L2*, the strategy needs to be aimed at the maximization of the *L2* reuse.

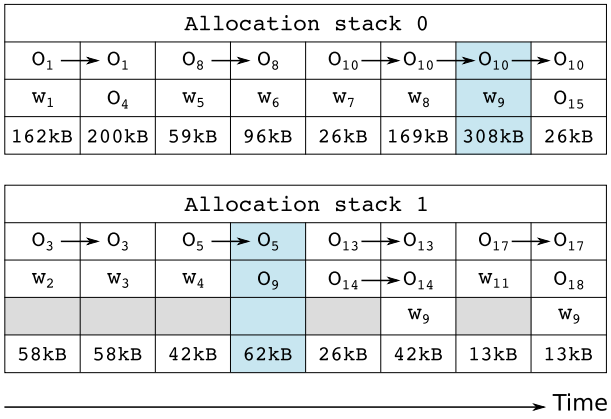

Fig. 7. *L2* memory allocation sequence.

At boot time, before the actual computation loop starts: 1) we load all the weights, stored in the external flash memory as binary files, in the *L3* DRAM memory and 2) we call from the fabric controller the *runtime* allocator to reserve two *L2* *allocation stacks* (shown in Fig. 7) where intermediate buffers will be allocated and deallocated in a linear fashion. The choice to use two allocation stacks instead of a single one is because in the latter case, we would need to keep alive up to 665 KB in *L2* due to data dependencies, which is more than the available space. Our allocation strategy updates the pointer of the next free location in the preallocated *L2* stack, avoiding the runtime overhead of library allocation/free functions. We differentiate our strategy between weights and feature maps: for the former, we allocate space just before their related layer and deallocate it just after the layer execution, as also shown in the pseudo-code blocks of Fig. 6. For the latter, due to the residual network bypasses, we often have to prolongate the lifetime of a feature map during the execution of the two following layers (node kernels in Fig. 6). Therefore, for each RES block, there will be an amount of time where three tensors have to be stored at the same time.

Fig. 6 shows the full execution flow of DroNet related to our solution, annotated with the sequence of node kernels and the *L3*/*L2* memory management blocks. For the sake of readability, in Fig. 6, we report only the first RES block, but this can be generalized also to the others with few minor modifications and updating input, output, and weights id. In the pseudo-code of Fig. 6, the second parameter of the `Alloc` and `Free` function specifies the allocation buffer (i.e., `Allocation stack 0` or `Allocation stack 1` in Fig. 7). Note that, the μDMA copies the weights from *L3* to *L2* just after the destination *L2* area is allocated. The buffers' memory allocation sequence is reported in Fig. 7 (from left to right) for the entire DroNet execution. The columns of the two stacks represent the data needed at each execution step, where O_i and w_j represent the input/output feature maps and weights, respectively. The last row of each stack reports the total amount of *L2* memory required at each step. Thus, the final dimension of each stack is given by the column with the biggest occupancy (highlighted in light blue in Fig. 7), resulting in 370 KB of *L2* memory. Therefore, our solution not only allows to the DroNet

execution within the *L2* memory budget constraint but results in leaving 142 KB of the *L2* still available (i.e., ∼28% of the total) for additional onboard tasks like target following [40], etc.

V. PULP-SHIELD

To host our visual navigation algorithm, we designed a lightweight, modular, and configurable printed circuit board (PCB) with highly optimized layout and a form factor compatible with our nano-size quadrotor. It features a PULP-based GAP8 SoC, two Cypress *HyperBus Memories*[11] and an ultralow-power *HiMax* CMOS image sensor[12] able to run up to 60 frames/s with a gray-scale resolution of 320 × 240 pixels with just 4.5 mW of power. Our pluggable PCB, named *PULP-Shield*, has been designed to be compatible with the *Crazyflie 2.0* (CF) nano-quadrotor.[13] This vehicle has been chosen due to its reduced size (i.e., 27 g of weight and 10 cm of diameter) and its open-source and open-hardware philosophy. The communication between the PULP chip and the main MCU aboard the nano-drone (i.e., *ST Microelectronics STM32F405*[14]) is realized via an SPI interface and two GPIO signals.

In Fig. 8, the schematic of the proposed PULP-Shield is shown. Two BGA memory slots allow all possible combinations of *HyperRAM*, *HyperFlash*, and hybrid *HyperFlash/RAM* packages. In this way, we can select the most appropriate memory configuration given a target application. We mounted on one slot a 64 Mbit *HyperRAM* (DRAM) chip and on the other a 128 Mbit *HyperFlash* memory, embodying the system *L3* and the external storage, respectively.

On the PCB [Fig. 8(b)], there is also a camera connector that allows the *HiMax* camera to communicate with the rest of the system through the parallel camera interface (PCI) protocol. Two mounting holes, on the side of the camera connector, allow plugging a 3-D printed camera holder that can be set either in front-looking or down-looking configuration. Those two configurations are representative of the most common visual sensors layouts typically embedded in any autonomous flying vehicles. The front-looking configuration can be used for many navigation tasks like path planning [41], obstacle avoidance [42], trajectory optimization [9], to name a few. Instead, the down-looking camera configuration is usually chosen for stabilization tasks like distance estimation [43], way-point tracking, and positioning [44], etc.

On the shield, there are also a JTAG connector for debug purposes and an external I2C plug for future development. Two headers, located on both sides of the PCB, grant a steady physical connection with the drone and at the same time, they bring the shield power supply and allow communication with the CF through the GPIOs and the SPI interface. The form factor of our final PULP-Shield prototype is 30 × 28 mm, and

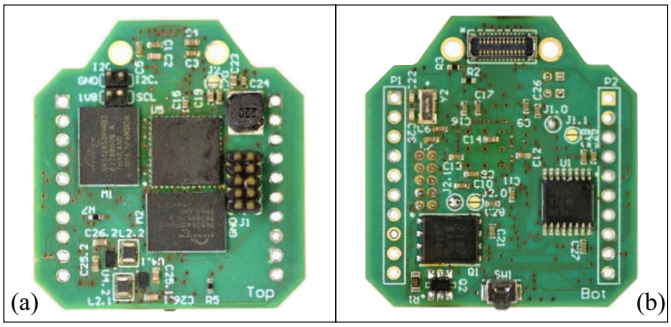

Fig. 8. *PULP-Shield* pluggable PCB. (a) Top view. (b) Bottom view.

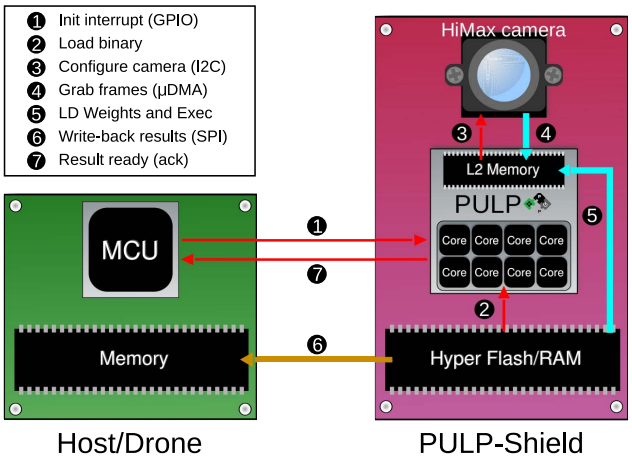

Fig. 9. Example of interaction between the *PULP-Shield* and the drone.

it weighs ∼5 g (including all components), well below the payload limit imposed by the nano-quadcopter.

Similarly to what has been presented in [36], the PULP-Shield embodies the *Host-Accelerator* architectural paradigm, where the CF's MCU offloads the intensive visual navigation workload to the PULP accelerator. As depicted in Fig. 9 the interaction starts from the host, which wakes up the accelerator with a GPIO interrupt ❶. Then, the accelerator fetches from its external *HyperFlash* storage the kernel (stored as a binary file) to be executed: DroNet in our case ❷. Note that, in this first part of the protocol the host can also specify which kernel should be executed, as well as a sequence of several preloaded ones available on the external Flash storage. At this point, the GAP8 SoC can configure the *HiMax* camera via an internal I2C ❸ and start to transfer the frames from the sensor to the *L2* shared memory through the μDMA ❹. All additional data, like the weights used in our CNN, can be loaded from the DRAM/Flash memory and parallel execution is started on the accelerator ❺. Lastly, the results of the computation are returned to the drone's MCU via SPI ❻, and the same host is acknowledged about the available results with a final interrupt over GPIO ❼. Note that, the transfer of a new frame is performed by the μDMA overlapping the CNN computation on the previous frame performed in the CL.

Even if the PULP-Shield has been developed specifically to fit the CF quadcopter, its basic concept and the functionality it provides are quite general, and portable to any drone based on an SPI-equipped MCU and more generally

[11][Online]. Available: http://www.cypress.com/products/hyperbus-memory

[12][Online]. Available: http://www.himax.com.tw/products/cmos-image-sensor/image-sensors

[13][Online]. Available: https://www.bitcraze.io/crazyflie-2

[14][Online]. Available: http://www.st.com/en/microcontrollers/stm32f405-415.html

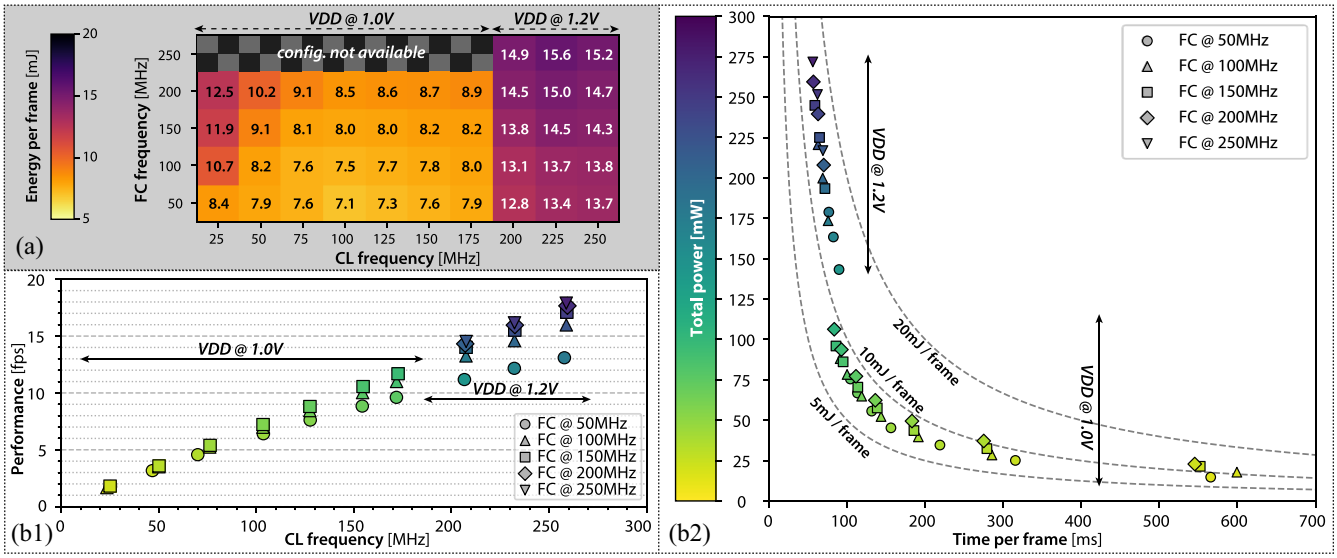

Fig. 10. (a) Heat map showing the energy per frame in all tested configurations of GAP8 with VDD@1.0 V and VDD@1.2 V. (b1) DroNet performance in frames/s in all tested configurations (coloring is proportional to total system power). (b2) DroNet total system power versus time per frame in all tested configurations; dashed gray lines show the levels of energy efficiency in mJ/frame.

to a generic IoT node requiring visual processing capabilities. The system-level architectural template it is based on is meant for minimizing data transfers (i.e., exploiting locality of data) and communication overhead between the main MCU and the accelerator—without depending on the internal microarchitecture of either one.

VI. Experimental Results

In this section, we present the experimental evaluation of our visual navigation engine, considering three primary metrics: 1) the capability of respecting a given real-time deadline; 2) the ability of performing all the required computations within the allowed power budget; and 3) the final accuracy of the closed-loop control, given as reaction time with respect to an unexpected obstacle. All the results are based on the PULP-Shield configuration presented in Section V.

A. Performance and Power Consumption

We measured wall-time performance and power consumption by sweeping between several operating modes on GAP8. We focused on operating at the lowest (1.0 V) and highest (1.2 V) supported core VDD voltages. We swept the operating frequency between 50 and 250 MHz, well beyond the GAP8 officially supported configuration.[15] Fig. 10 provides a complete view of the power consumption in all experimentally possible operating modes of GAP8 on the DroNet application while sweeping both FC and CL clock frequency, both at 1.0 and 1.2 V and the related achievable frame-rate. Fig. 10(a) shows the energy-efficiency of all available configurations as a heat map, where VDD@1.0 V, FC@50 MHz, and CL@100 MHz represent the most energy efficient one. In Fig. 10(b1), we report performance as frame-rate and total power consumption measured before the internal dc/dc converter utilized on the SoC. Selecting a VDD operating point of 1.2 V would increase both power and performance up to 272 mW and 18 frames/s. We found the SoC to be working correctly @ 1.0 V for frequencies up to ∼175 MHz; we note that as expected when operating @ 1.0 V there is a definite advantage in terms of energy efficiency. Therefore, for the sake of readability, in Fig. 10 we avoid showing configurations of VDD 1.2 V that would reach the same performance of VDD 1.0 V at a higher cost in terms of power. Similarly, in Fig. 10(b2) we report power consumption vs time to compute one frame.

In Fig. 11, we present the power traces for full end-to-end execution of DroNet, measured using a bench dc power analyzer.[16] The power traces are measured by powering the GAP8 SoC, with the most energy-efficient configuration at 1.0 V core voltage and operating at 50 MHz on FC and 100 MHz on CL. The detailed average power consumption (including both FC and CL domains) is reported in Table III. The peak power consumption of 47 mW is associated to the first convolutional layer; we used this value to compute the overall power envelope of our node. Instead, the minimum power consumption is given by the two last fully connected layers consuming 13 mW each. The average power consumption, weighted throughout each layer, is 39 mW, which grows to 45 mW including also the losses on the internal dc/dc converter (not included in Fig. 11). In the full DroNet execution, layers are interposed with *L3*-*L2* data transfers, happening with the CL cores in a clock-gated state, which accounts for ∼7% of the overall execution time. Therefore, power consumption for the entire board settles to 64 mW if we also consider the cost of *L3* memory access and the onboard ULP camera.

Fig. 12 reports the power break-down for the complete cyber-physical system and proposed PULP-Shield. Our

[15][Online]. Available: https://greenwaves-technologies.com/gap8-datasheet

[16][Online]. Available: www.keysight.com/en/pd-1842303-pn-N6705B

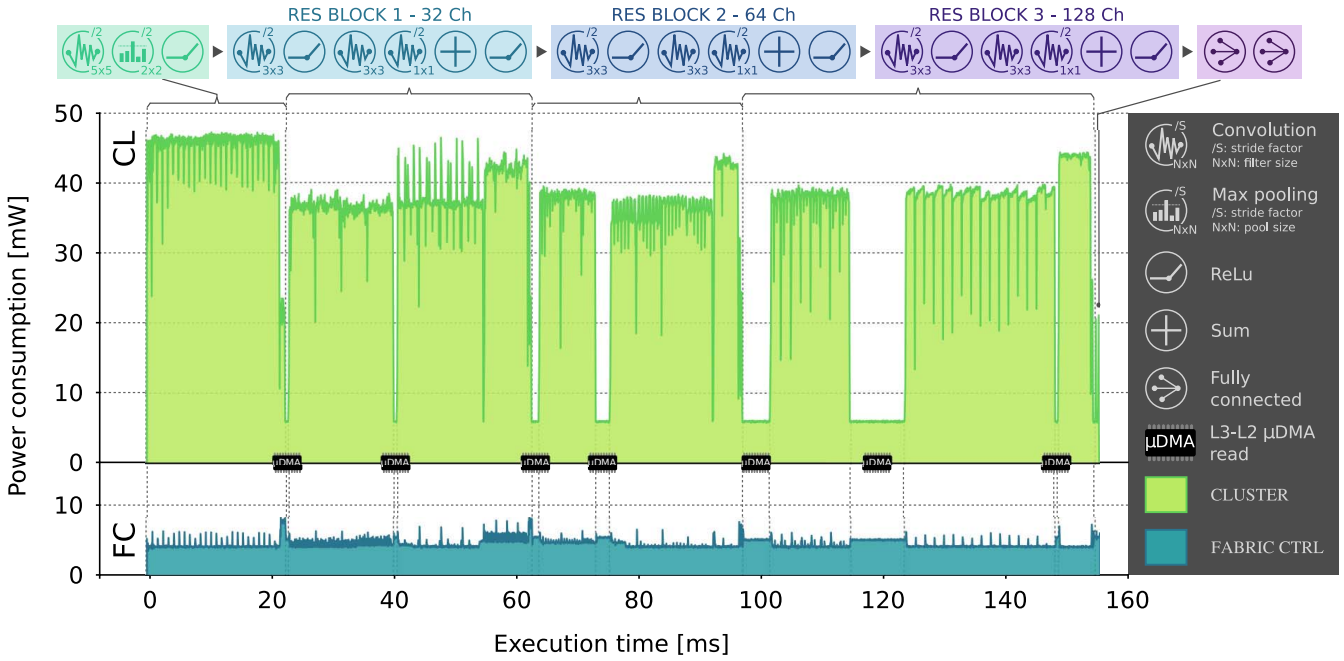

Fig. 11. Power traces per layer of *DroNet*, measured at VDD@1.0 V, FC@50 MHz, and CL@100 MHz, the symbols on top of the plot indicate the computation stage associated with each visible phase in the power trace. Measurements are taken after internal dc/dc converter (i.e., accounting for both FC and CL).

TABLE III
POWER CONSUMPTION AND EXECUTION TIME PER FRAME OF *DroNet* ON GAP8 VDD@1.0 V, FC@50 MHz, AND CL@100 MHz

Layer	AVG Power [mW]	Exec Time [ms]	L3-L2 Time [ms]
conv_1 + pool	47.1	22.6	0.1
ReLU	24.8	0.9	—
conv_2 + ReLU	38.7	17.3	0.6
conv_3	38.0	14.3	0.6
conv_4	43.6	7.3	0.1
add	38.9	0.3	—
ReLU	27.6	0.2	—
conv_5 + ReLU	37.7	9.3	1.2
conv_6	34.8	17.0	2.4
conv_7	32.7	4.2	0.2
add	24.3	0.3	—
ReLU	20.5	0.3	—
conv_8 + ReLU	33.1	13.0	4.7
conv_9	31.9	24.8	9.4
conv_10	41.9	5.4	0.5
add + ReLU	24.4	0.3	—
fully_1	13.0	0.1	0.4
fully_2	13.0	0.1	0.4

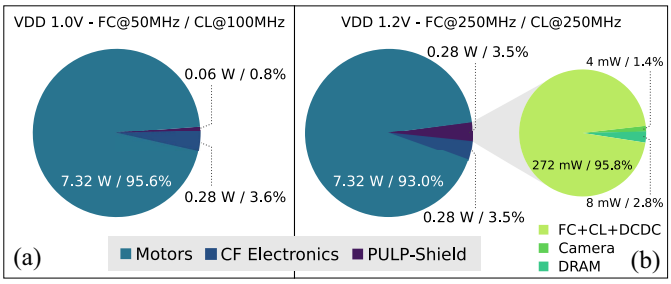

Fig. 12. Power envelope break-down of the entire cyber-physical system running at (a) FC@50 MHz-CL@100 MHz and (b) FC@250 MHz-CL@250 MHz with *PULP-Shield* zoomed-in view (on the right).

nano-quadcopter is equipped with a 240 mA h 3.7 V LiPo battery enabling a flight time of 7 minutes under standard conditions, which results in average power consumption of 7.6 W. The power consumption of all the electronics aboard the original drone amounts to 277 mW leaving ∼7.3 W for the four rotors. The electronics consumption is given by the two MCUs included in the quadrotor and all the additional devices (e.g., sensors, LEDs, etc.). In addition to that, introducing the PULP-Shield, we increase the peak power envelope by 64 mW using the most energy-efficient configuration and by 284 mW selecting the fastest setting (0.8% and 3.5% of the total, respectively). On the PULP-Shield, we consider the HyperRAM is operating at full speed only for the time required for *L3-L2* data transfers (as shown in Table III) with an average power consumption of 8 mW for the fastest configuration, as reported in Fig. 12(b). Notice that this is a worst-case figure, taking account of both GAP8 SoC and HyperRAM operating at full speed simultaneously. The power break-down of our visual navigation module can be seen on the right of Fig. 12(b), where we include the computational unit, the *L3* external DRAM memory, and the ultralow-power camera. As onboard computation accounts for roughly 5% of the overall power consumption (propellers, sensors, computation, and control, see Section I), our PULP-Shield enables the execution of the DroNet network (and potentially more) in all configurations within the given power envelope.

Finally, in our last experiment, we evaluate the cost in terms of operating lifetime of carrying the physical payload of the PULP-Shield and of executing the DroNet workload. To ensure a fair measurement, we decoupled the DroNet output from the nano-drone control and statically set it to *hover* (i.e.,

TABLE IV
Crazyflie (CF) LIFETIME WITH AND WITHOUT *PULP-Shield* (BOTH TURNED OFF AND RUNNING *DroNet* AT VDD@1.0 V, FC@50 MHz, AND CL@100 MHz)

	Original CF	CF + *PULP-Shield* (off)	CF + *PULP-Shield* (on)
Lifetime	∼440 s	∼350 s	∼340 s

TABLE V
CL-CYCLE BREAK-DOWN FOR PROCESSING ONE FRAME ON THE GAP8 BOTH FC AND CL @ 50 MHz

	μDMA L3/L2	DMA L2/L1	Computation	Total
Cycles	1.03 M	0.11 M	13.47 M	14.61 M

keep constant hight over time) at 0.5 m from the ground. We targeted three different configurations.

1) The original *Crazyflie* (CF) without any PULP-Shield.
2) PULP-Shield plugged but never turned on, to evaluate the lifetime reduction due to the additional weight introduced.
3) PULP-Shield turned on executing DroNet at VDD@1.0 V, FC@50 MHz, and CL@100 MHz.

Our results are summarized in Table IV and as expected the biggest reduction in the lifetime is given by the increased weight. The flight time of the original nano-drone, with one battery fully charged, is ∼440 s. This lifetime drops to ∼350 s when the drone is carrying the PULP-Shield (turned off) and to ∼340 s when the shield is executing DroNet. Ultimately, the price for our visual navigation engine is ∼22% of the original lifetime.

B. State-of-the-Art Comparison and Discussion

To compare and validate our experimental results with respect to the current SoA, we targeted the most efficient CNN implementation currently available for microcontrollers, namely CMSIS-NN [28]. At peak performance in a synthetic test, this fully optimized library can achieve as much as 0.69 MAC/cycle on convolutions, operating on *Fixed8* data that is internally converted to *Fixed16* in the inner loop.

By contrast, we operate directly on *Fixed16* and achieve a peak performance of 0.64 MAC/cycle/core in a similar scenario (on the sixth layer of DroNet, 3×3 convolution). The bypasses and the final layers are a bit less efficient, yielding an overall weighted peak throughput of 0.53 MAC/cycle/core on convolutional layers, which constitute the vast majority of the execution time.

In Table V, we report the execution breakdown per frame for all activities performed by our CNN. We can see how the *L3-L2* transfers (not overlapped to computation) and the nonoverlapping part of *L2-L1* transfers account for ∼1.14 Mcycles of the overall execution time. Then, considering ∼41 MMAC for the original CNN, in the ideal peak-throughput case of 4.28 MAC/cycle we would need ∼10 Mcycles for computing one frame, instead of our measured 13.47 Mcycles. The overhead is due to inevitable nonidealities such as suboptimal load balancing in layers exposing limited spatial parallelism as well as tiling control loops and the marshaling stage required by padded convolutions. Considering all of the effects mentioned above (i.e., computation nonidealities as well as memory transfers), we achieve a real throughput of 2.81 MAC/cycle in the DroNet execution—still 4× better than the CMSIS-NN peak performance.

To further concretize the comparison, we take as an example target a top-notch high-performance microcontroller: an STM32H7[17] sporting a Cortex-M7 core and capable of operating at up to 400 MHz. Without considering any data movement overhead, and taking into account only peak performance, this would be able to achieve up to 276 MMAC/s @ 346 mW. By comparison, our system can achieve an average performance of 281 MMAC/s with the most power-efficient configuration @ 45 mW, i.e., same performance within a 5.4× smaller power budget. Moreover, if we consider our peak-throughput configuration (where both FC and CL are running @ 250 MHz) we can deliver up to 702 MMAC/s @ 272 mW: 2.5× better with 21% less power. Even if it were possible to linearly up-scale the performance of this microcontroller to the same level of our system, it would consume ∼880 mW, which would constitute largely more than the 5% of power envelope typically dedicated to onboard computation on nano-UAV systems [14]. This confirms that the parallel-ultralow-power approach adopted in our visual navigation engine significantly outperforms sequential processing in terms of energy efficiency, without compromising programmability and flexibility.

C. Control Accuracy

To fully exploit the natural inertial agility of a lightweight nano-quadrotor as the *Crazyflie 2.0* used in our prototype, fast onboard perception is required. To evaluate the agility of our integrated system, we perform an experiment in which our flying platform is required to react to a sudden obstacle occluding its way. With this experiment, we aim to demonstrate that the PULP-Shield computational resources are enough to make full use of the platform agility. As mentioned in Section IV-A, for the final deployment of DroNet on the PULP-Shield, we select the network trained with `Fixed16` quantization, 2×2 max-pool receptive field, and fine-tuning dataset. The choice is justified by both quantization requirement of the GAP8 SoC and the model performance, superior to other viable alternatives (see Table II).

The experimental setting is as follows: we collect a dataset of images by manually flying the drone over a straight path of 20 m at an average speed of 4 m/s. At the beginning of the test, the path is entirely free from obstacles. At $T = 4$ s after the start of the experiment, an obstacle appears at the end of the track, leaving 4 m free for breaking and stopping. The system is then required to raise a stop signal soon enough to avoid the collision. As we show in the additional video, our integrated system can control the nano-drone in closed-loop. However, for safety reasons and to avoid damaging the platform, we do not control the nano-drone in closed-loop during this experiment. Instead, we process the frames collected with manual flight offline. The collected dataset is used to study

[17][Online]. Available: http://www.st.com/en/microcontrollers/stm32h7-series.html

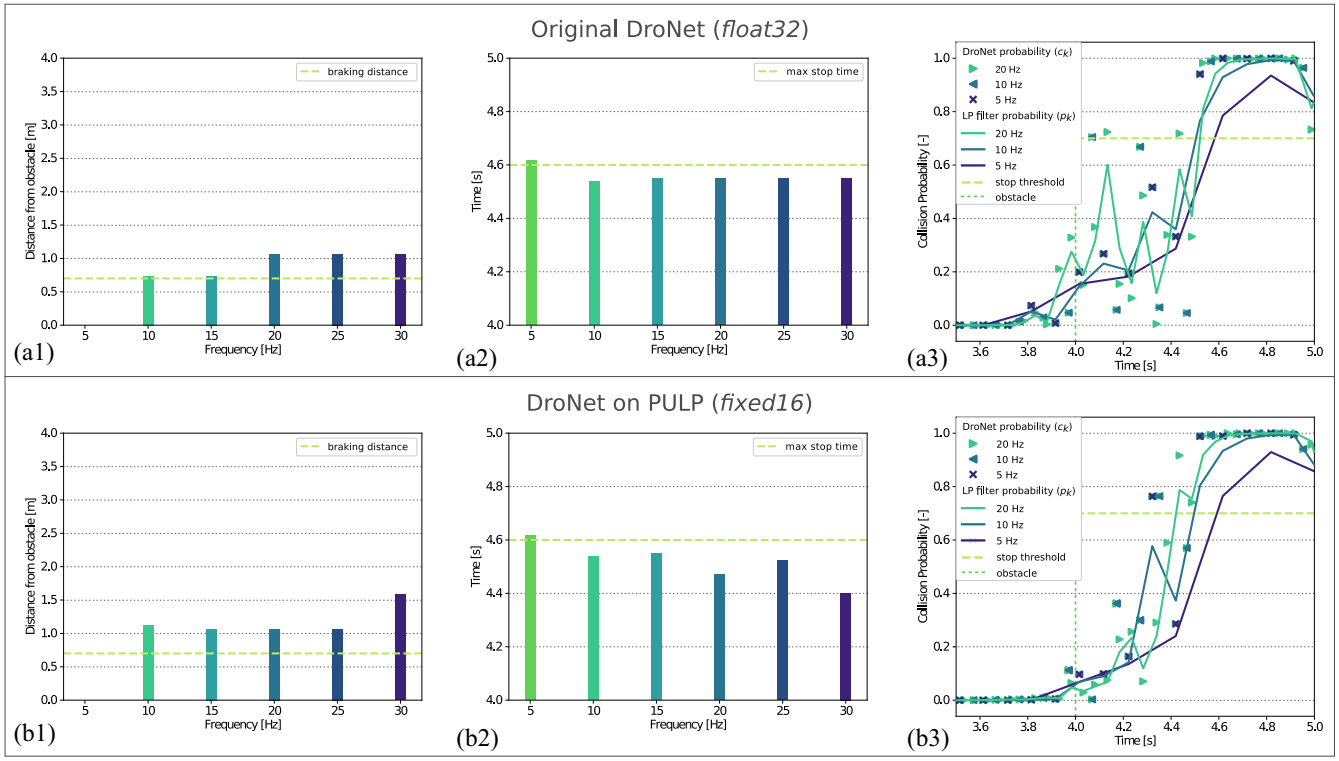

Fig. 13. Performance comparison between the (a1)–(a3) original and (b1)–(b3) quantized *DroNet* architectures. (a1) and (b1) Stop command time. (a2) and (b2) Minimum distance from the obstacle. (a3) and (b3) Collision probability as the output of both CNN and low-pass filter.

the relation between the system operational frequencies and the drone's reaction time.

As in the original implementation of [2], network's predictions are low-pass filtered to decrease high-frequency noise. In detail, the collision probability p_k is a low-pass filtered version of the raw network output c_k ($\alpha = 0.7$)

$$p_k = (1 - \alpha)p_{k-1} + \alpha c_k. \tag{1}$$

Fig. 13 (a3)–(b3) illustrates the predicted collision probability of the original and quantized DroNet CNN as a function of time. In the plots, we show both c_k and p_k at different frequencies, the former reported as markers, whereas the latter is shown as a continuous line. A horizontal dashed orange line shows the threshold for sending a stop signal to the control loop ($p_k > 0.7$), and a vertical red dashed line highlights the time at which the obstacle becomes visible ($T = 4$ s).

To quantitatively evaluate the performance of our system at different operational frequencies, we computed the maximum time and the minimum distance from the object at which the stop command should be given to avoid the collision. We deployed the *Crazyflie 2.0* parameters from [45] and the classical quadrotor motion model from [46] to *analytically* compute those two quantities. From this analysis, we derived a minimum stopping time of 400 ms and a braking distance of 0.7 m, assuming the platform moves with a speed of 4 m/s when it detects the obstacle.

In Fig. 13(a1) and (a2) and (b1) and (b2), we illustrate a performance comparison between our quantized system and the original implementation of [2]. Despite quantization, our network outperforms [2] in terms of collision detection, and can react more quickly to sudden obstacles even at low-operational frequencies. This is in accordance with the results of Table II, and mainly due to the fine-tuning of our network to the HiMax camera images.

Both quantized and original architecture share, however, a similar behavior at different operational frequencies. More specifically, both fail to detect obstacles at very low frequencies (i.e., 5 Hz), but successfully avoid the collision at higher rates. Interestingly, increasing the system frequencies does not always improve performance; it can be observed in Fig. 13(b2), where performance at 20 Hz is better than at 25 Hz. From Fig. 13, we can observe that inference at 10 Hz allows the drone to brake in time and avoid the collision. This confirms that our system, processing up to 18 frames/s, can: 1) make use of the agility of the *Crazyflie 2.0* and 2) be deployed in the same way as the original method to navigate in indoor/outdoor environments while avoiding dynamic obstacles. A supplementary video showing the performance of the system controlled in closed-loop can be seen at the following link: https://youtu.be/57Vy5cSvnaA.

VII. Conclusion

Nano- and pico-sized UAVs are ideal IoT nodes; due to their size and physical footprint, they can act as mobile IoT hubs, smart sensors and data collectors for tasks such as surveillance, inspection, etc. However, to be able to perform these tasks, they must be capable of autonomous navigation of environments such as urban streets, industrial facilities, and other hazardous or otherwise challenging areas. In this

paper, we present a complete deployment methodology targeted at enabling execution of complex deep learning algorithms directly aboard resource-constrained mW-scale nodes. We provide the first (to the best of our knowledge) completely vertically integrated hardware/software visual navigation engine for autonomous nano-UAVs with completely onboard computation—and thus potentially able to operate in conditions in which the latency or the additional power cost of a wirelessly connected centralized solution.

Our system, based on a *GreenWaves Technologies* GAP8 SoC used as an accelerator coupled with the STM32 MCU on the *Crazyflie 2.0* nano-UAV, supports real-time computation of DroNet, an advanced CNN-based autonomous navigation algorithm. Experimental results show a performance of 6 frames/s @ 64 mW selecting the most energy-efficient SoC configuration, that can scale up to 18 frames/s within an average power budget for computation of 284 mW. This is achieved without quality-of-results loss with respect to the baseline system on which DroNet was deployed: a COTS standard-size UAV connected with a remote PC, on which the CNN was running at 20 frames/s. Our results show that both systems can detect obstacles fast enough to be able to safely fly at high speed, 4 m/s in the case of the *Crazyflie 2.0*. To further paving the way for a vast number of advanced use-cases of autonomous nano-UAVs as IoT-connected mobile smart sensors, we release open-source our PULP-Shield design and all code running on it, as well as datasets and trained networks.

ACKNOWLEDGMENT

The authors would like to thank H. Müller for her contribution in designing the *PULP-Shield*, N. Brun for his support in making the camera-holder, and F. K. Gürkaynak for his assistance in making the supplementary videos.

REFERENCES

[1] N. H. Motlagh, T. Taleb, and O. Arouk, "Low-altitude unmanned aerial vehicles-based Internet of Things services: Comprehensive survey and future perspectives," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 899–922, Dec. 2016.

[2] A. Loquercio, A. I. Maqueda, C. R. del Blanco, and D. Scaramuzza, "DroNet: Learning to fly by driving," *IEEE Robot. Autom. Lett.*, vol. 3, no. 2, pp. 1088–1095, Apr. 2018.

[3] F. Conti *et al.*, "An IoT endpoint system-on-chip for secure and energy-efficient near-sensor analytics," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 64, no. 9, pp. 2481–2494, Sep. 2017.

[4] D. Palossi, A. Gomez, S. Draskovic, A. Marongiu, L. Thiele, and L. Benini, "Extending the lifetime of nano-blimps via dynamic motor control," *J. Signal Process. Syst.*, vol. 91, nos. 3–4, pp. 339–361, Feb. 2018.

[5] D. Floreano and R. J. Wood, "Science, technology and the future of small autonomous drones," *Nature*, vol. 521, no. 7553, pp. 460–466, May 2015.

[6] Z. Liu, Y. Chen, B. Liu, C. Cao, and X. Fu, "HAWK: An unmanned mini-helicopter-based aerial wireless kit for localization," *IEEE Trans. Mobile Comput.*, vol. 13, no. 2, pp. 287–298, Feb. 2014.

[7] M. Piccoli and M. Yim, "Piccolissimo: The smallest micro aerial vehicle," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, Singapore, May 2017, pp. 3328–3333.

[8] Y. Lin *et al.*, "Autonomous aerial navigation using monocular visual-inertial fusion," *J. Field Robot.*, vol. 35, no. 1, pp. 23–51, Jul. 2017.

[9] D. Falanga, E. Mueggler, M. Faessler, and D. Scaramuzza, "Aggressive quadrotor flight through narrow gaps with onboard sensing and computing using active vision," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2017, pp. 5774–5781.

[10] G. Loianno, D. Scaramuzza, and V. Kumar, "Special issue on high-speed vision-based autonomous navigation of UAVs," *J. Field Robot.*, vol. 35, no. 1, pp. 3–4, 2018.

[11] Y. Yang, Z. Zheng, K. Bian, L. Song, and Z. Han, "Real-time profiling of fine-grained air quality index distribution using UAV sensing," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 186–198, Feb. 2018.

[12] J. Conroy, G. Gremillion, B. Ranganathan, and J. S. Humbert, "Implementation of wide-field integration of optic flow for autonomous quadrotor navigation," *Auton. Robots*, vol. 27, no. 3, p. 189, 2009.

[13] K. McGuire, G. de Croon, C. D. Wagter, K. Tuyls, and H. Kappen, "Efficient optical flow and stereo vision for velocity estimation and obstacle avoidance on an autonomous pocket drone," *IEEE Robot. Autom. Lett.*, vol. 2, no. 2, pp. 1070–1076, Apr. 2017.

[14] R. J. Wood *et al.*, *Progress on 'Pico' Air Vehicles*. Cham, Switzerland: Springer Int., 2017.

[15] S. Scherer *et al.*, "River mapping from a flying robot: State estimation, river detection, and obstacle mapping," *Auton. Robots*, vol. 33, nos. 1–2, pp. 189–214, 2012.

[16] D. Scaramuzza *et al.*, "Vision-controlled micro flying robots: From system design to autonomous navigation and mapping in GPS-denied environments," *IEEE Robot. Autom. Mag.*, vol. 21, no. 3, pp. 26–40, Sep. 2014.

[17] D. Gandhi, L. Pinto, and A. Gupta, "Learning to fly by crashing," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Vancouver, BC, Canada, Sep. 2017, pp. 3948–3955.

[18] F. Sadeghi and S. Levine, "CAD2rl: Real single-image flight without a single real image," in *Proc. Robot. Sci. Syst. XIII*, Jul. 2017.

[19] N. Smolyanskiy, A. Kamenev, J. Smith, and S. Birchfield, "Toward low-flying autonomous MAV trail navigation using deep neural networks for environmental awareness," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Vancouver, BC, Canada, Sep. 2017, pp. 4241–4247.

[20] S. Ross *et al.*, "Learning monocular reactive UAV control in cluttered natural environments," in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2013, pp. 1765–1772.

[21] O. Dunkley, J. Engel, J. Sturm, and D. Cremers, "Visual-inertial navigation for a camera-equipped 25g nano-quadrotor," in *Proc. IROS Aerial Open Source Robot. Workshop*, 2014.

[22] A. Briod, J.-C. Zufferey, and D. Floreano, "Optic-flow based control of a 46g quadrotor," in *Proc. Workshop Vis. Based Closed Loop Control Navig. Micro Helicopters GPS Denied Environ. IROS*, 2013.

[23] A. Suleiman, Z. Zhang, L. Carlone, S. Karaman, and V. Sze, "Navion: A fully integrated energy-efficient visual-inertial odometry accelerator for autonomous navigation of nano drones," in *Proc. IEEE Symp. VLSI Circuits*, Honolulu, HI, USA, Jun. 2018, pp. 133–134.

[24] K. McGuire, G. de Croon, C. de Wagter, B. Remes, K. Tuyls, and H. Kappen, "Local histogram matching for efficient optical flow computation applied to velocity estimation on pocket drones," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2016, pp. 3255–3260.

[25] G. Hegde, N. Ramasamy, and N. Kapre, "CaffePresso: An optimized library for Deep Learning on embedded accelerator-based platforms," in *Proc. Int. Conf. Compliers Archit. Syth. Embedded Syst. (CASES)*, Pittsburgh, PA, USA, Oct. 2016, pp. 1–10.

[26] D. Kang, E. Kim, I. Bae, B. Egger, and S. Ha, "C-GOOD: C-code generation framework for optimized on-device deep learning," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, San Diego, CA, USA, 2018, pp. 1–105.

[27] A. Lokhmotov, N. Chunosov, F. Vella, and G. Fursin, "Multi-objective autotuning of mobilenets across the full software/hardware stack," in *Proc. 1st Reproducible Qual. Efficient Syst. Tournament Co Designing Pareto Efficient Deep Learn. (ReQuEST)*, Williamsburg, VA, USA, 2018, Art. no. 6.

[28] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for arm cortex-M CPUs," *arXiv:1801.06601 [cs]*, Jan. 2018.

[29] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Oct. 2013, pp. 13–19.

[30] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, Monterey, CA, USA, 2015, pp. 161–170.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[32] M. Gautschi *et al.*, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017.

[33] A. Pullini, D. Rossi, G. Haugou, and L. Benini, "uDMA: An autonomous I/O subsystem for IoT end-nodes," in *Proc. 27th Int. Symp. Power Timing Model. Optim. Simulat. (PATMOS)*, Sep. 2017, pp. 1–8.

[34] D. E. Bellasi and L. Benini, "Smart energy-efficient clock synthesizer for duty-cycled sensor SoCs in 65 nm/28nm CMOS," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 64, no. 9, pp. 2322–2333, Sep. 2017.

[35] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini, "A fully-synthesizable single-cycle interconnection network for shared-*L*1 processor clusters," in *Proc. Design. Autom. Test Europe*, Grenoble, France, Mar. 2011, pp. 1–6.
[36] F. Conti, D. Palossi, A. Marongiu, D. Rossi, and L. Benini, "Enabling the heterogeneous accelerator model on ultra-low power microcontroller platforms," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, Mar. 2016, pp. 1201–1206.
[37] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6869–6898, 2017.
[38] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "CNN features off-the-shelf: An astounding baseline for recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Columbus, OH, USA, Jun. 2014, pp. 512–519.
[39] L. Cecconi, S. Smets, L. Benini, and M. Verhelst, "Optimal tiling strategy for memory bandwidth reduction for CNNs," in *Advanced Concepts for Intelligent Vision Systems*, J. Blanc-Talon, R. Penne, W. Philips, D. Popescu, and P. Scheunders, Eds. Cham, Switzerland: Springer Int., 2017.
[40] D. Palossi, J. Singh, M. Magno, and L. Benini, "Target following on nano-scale unmanned aerial vehicles," in *Proc. 7th IEEE Int. Workshop Adv. Sensors Interfaces (IWASI)*, Jun. 2017, pp. 170–175.
[41] P. Kumar, S. Garg, A. Singh, S. Batra, N. Kumar, and I. You, "MVO-based 2-D path planning scheme for providing quality of service in UAV environment," *IEEE Internet Things J.*, vol. 5, no. 3, pp. 1698–1707, Jun. 2018.
[42] C. Yin, Z. Xiao, X. Cao, X. Xi, P. Yang, and D. Wu, "Offline and online search: UAV multiobjective path planning under dynamic urban environment," *IEEE Internet Things J.*, vol. 5, no. 2, pp. 546–558, Apr. 2018.
[43] D. Palossi, A. Marongiu, and L. Benini, "Ultra low-power visual odometry for nano-scale unmanned aerial vehicles," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, Mar. 2017, pp. 1647–1650.
[44] D. Falanga, A. Zanchettin, A. Simovic, J. Delmerico, and D. Scaramuzza, "Vision-based autonomous quadrotor landing on a moving platform," in *Proc. IEEE Int. Symp. Safety Security Rescue Robot. (SSRR)*, Oct. 2017, pp. 200–207.
[45] J. Förster, M. Hamer, and R. D'Andrea, "System identification of the crazyflie 2.0 nano quadrocopter," B.S. thesis, Inst. Dyn. Syst. Control, ETH Zürich, Zürich, Switzerland, 2015.
[46] R. Mahony, V. Kumar, and P. Corke, "Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor," *IEEE Robot. Autom. Mag.*, vol. 19, no. 3, pp. 20–32, Sep. 2012.

Daniele Palossi received the B.S. and M.S. degrees in computer science engineering from the University of Bologna, Bologna, Italy. He is currently pursuing the Ph.D. degree at the Department of Information Technology and Electrical Engineering, ETH Zürich, Zürich, Switzerland.

In 2012, he spent six months as a Research Intern with ST Microelectronics, Agrate Brianza, Italy, performing research on 3-D computer vision algorithms for the STM STHORM project. His current research interests include energy-efficient algorithms for autonomous vehicles and advanced driver assistance systems.

Mr. Palossi was a recipient of a one-year research grant of the University of Bologna, in 2013, with a focus on design methodologies for high-performance embedded systems.

Antonio Loquercio received the M.Sc. degree in robotics, systems, and control from ETH Zürich, Zürich, Switzerland, in 2017. He is currently pursuing the Ph.D. degree at the Robotics and Perception Group, University of Zürich, Zürich, under the supervision of Prof. D. Scaramuzza.

His current research interests include data-driven methods for perception and control in robotics.

Mr. Loquercio was a recipient of the ETH Medal for outstanding master thesis in 2017.

Francesco Conti (GS'13–M'15) received the Ph.D. degree from the University of Bologna, Bologna, Italy, in 2016.

He is currently a Post-Doctoral Researcher with the Integrated Systems Laboratory, ETH Zürich, Zürich, Switzerland, and the Energy-Efficient Embedded Systems Laboratory, University of Bologna, Bologna, Italy. He has coauthored over 30 papers in international conferences and journals. His current research interests include energy-efficient multicore architectures and applications of deep learning to low power digital systems.

Dr. Conti was a recipient of three Best Paper Awards at ASAP'14, EVW'14, and ESWEEK'18 and the 2018 HiPEAC Tech Transfer Award.

Eric Flamand received the Ph.D. degree in computer science from INPG, Grenoble, France, in 1982.

He was a Researcher with CNET, Grenoble, France, and CNRS, Grenoble, France, on architectural automatic synthesis, design and architecture, and compiler infrastructure for highly constrained heterogeneous small parallel processors. He then held different technical management positions within the semiconductor industry, first with Motorola, Chicago, IL, USA, where he was involved in the architecture definition and tooling of the StarCore DSP. Then with ST Microelectronics, Geneva, Switzerland, first being in charge of all the software development of the Nomadik Application Processor and then in charge of the P2012 corporate initiative aiming at the development of a many-core device. He is the co-founder and currently the CTO of Greenwaves Technologies, Villard-Bonnot, France, a French-based startup developing an IoT processor derived from the PULP project. He is also a part-time Research Consultant with ETH Zürich, Zürich, Switzerland.

Davide Scaramuzza (GS'06–M'08) was born in Terni, Italy, in 1980. He received the Ph.D. degree in robotics and computer vision from ETH Zürich, Zürich, Switzerland, in 2008.

He was a Post-Doctoral Researcher with the University of Pennsylvania, Philadelphia, PA, USA. He is a Professor of Robotics with the University of Zürich, Zürich, where he does research at the intersection of robotics, computer vision, and neuroscience. From 2009 to 2012, he led the European project sFly, which introduced the world's first autonomous navigation of microdrones in GPS-denied environments using visual-inertial sensors as the only sensor modality. He coauthored book entitled *Introduction to Autonomous Mobile Robots* (MIT Press, 2011).

Dr. Scaramuzza was a recipient of the SNSF-ERC Starting Grant, the IEEE Robotics and Automation Early Career Award, and the Google Research Award for his research contributions.

Luca Benini (S'94–M'97–SM'04–F'07) is the Chair of Digital Circuits and Systems with ETH Zürich, Zürich, Switzerland, and a Full Professor with the University of Bologna, Bologna, Italy. He has authored or coauthored over 1000 papers, 5 books, and several book chapters. His current research interests include energy-efficient system design for embedded and high-performance computing, energy-efficient smart sensors, and ultralow power VLSI design.

Dr. Benini was a recipient of the 2016 IEEE CAS Mac Van Valkenburg Award. He is a Fellow of the ACM and a member of Academia Europaea.