Enabling Cognitive Autonomy on Small Drones by Efficient On-board Embedded Computing: An ORB-SLAM2 Case Study

Erqian Tang, Sobhan Niknam, Todor Stefanov
Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Leiden, the Netherlands
e.tang@liacs.leidenuniv.nl, s.niknam@liacs.leidenuniv.nl, t.p.stefanov@liacs.leidenuniv.nl

***Abstract*—In this paper, we present a case study which investigates whether/how Simultaneous Localization and Mapping (SLAM), e.g., the ORB-SLAM2 application, can be executed on a small, energy-efficient, multi-processor embedded platform with an ARM big.LITTLE architecture, e.g., the ODROID-XU4 platform, mounted on a small drone with a limited energy budget while meeting real-time performance requirements. More specifically, we model and implement ORB-SLAM2 as a Kahn Process Network (KPN) which exploits pipeline parallelism and enables efficient mapping and execution of ORB-SLAM2 onto ODROID-XU4. Moreover, our KPN model enables the application of generic model transformations to exploit data-level parallelism as well. Then, we propose and implement, on top of the Linux operating system, an environment for efficient execution of applications modeled as KPNs. Finally, we perform a simple design space exploration (DSE) to investigate the trade-off between system performance and power consumption when alternative ORB-SLAM2 KPNs are executed on different configurations of the ODROID-XU4 platform. The obtained results of this DSE clearly show the feasibility of running ORB-SLAM2 on ODROID-XU4 in real time with a limited power budget for a given range of flying time, thereby enabling cognitive autonomy on small drones.**

I. Introduction

In recent years, autonomous systems are emerging in both civil life and industrial production saving labour and providing convenience. Unmanned Aerial Vehicles (UAV), as one important representative of autonomous systems, can be utilized in many scenarios because of its portability and efficiency, for example, during search and rescue missions, military surveillance, or target detection and monitoring. UAVs can fly to a location and provide very fast and robust reaction. When executing missions, conventional UAVs often rely on a sensing system composed by an Inertial Measurement Unit (IMU) and the Global Positioning System (GPS). This technology is now also available in small UAVs, such as drones, for professional or recreational applications. However, GPS information is not sufficiently precise for altitude regulation when a small drone flies a few meters above the ground and is not always available or reliable in confined areas, such as cities, forests and buildings. Even if reliable GPS information was available, it would need to be combined with a precise map of the drone surroundings in order to identify obstacle-free trajectories [1].

Cognitive autonomy is a possible solution to the aforementioned problems. Relying on images captured by an on-board camera, a vision-based navigation method, called Simultaneous Localization and Mapping (SLAM) [2], enables to place an autonomous vehicle at an unknown location in an unknown environment, to build a map by only relative observations of the environment while the vehicle moves, and then to use this map simultaneously to navigate [3]. Typically, for real-life applications, the required image processing speed of SLAM is in the range of 10 to 20 frames per second (fps) where the typical size of a frame is around 3 MB [4], [5], [6], [7], [8]. However, due to its heavy computation workload, it is a challenge to implement and run SLAM in real time at 10 to 20 fps on-board of a small drone with limited energy budget and computational resources. One possible solution to this challenge is to execute SLAM off-board [9] by transmitting the images captured by the on-board camera to a ground station and run SLAM on a powerful computer/server. For example, when executing the SLAM parallel implementation given in [10], [11] on a desktop environment with 4 Intel Cores i7-6500U CPU @ 2.5GHz and 16GB RAM, the SLAM performance is 22 fps which is sufficient for real-time applications. Unfortunately, this solution may significantly limit the operational range of a drone due to limited transmission ability of the radio [12]. Moreover, some transmission latency exists inevitably which may influence negatively the system real-time performance. To avoid the above mentioned problems, another solution is to equip the small drone with a powerful embedded computing platform which enables the execution of SLAM on-board. For example, we can exploit GPU-based acceleration of SLAM on the Jetson TX2 embedded platform as in [13] or acceleration on FPGA-based platform as in [14] which gives the required real-time SLAM performance. However, such methods consume significant amount of power which cannot meet a limited power budget of 9 to 12 W (see in Section VI.B where this limited power budget comes from), available for SLAM, in order to enable small drones to fly 19 to 20 minutes on a typical small drone battery where a lot of power consumption goes to the motors.

Therefore, in this paper, we investigate whether/how SLAM can be implemented and executed on a small, energy-efficient, multi-processor embedded platform with an ARM big.LITTLE architecture [15] while meeting the real-time performance requirement of 10 to 20 fps with the power budget of 9 to 12 W. The specific SLAM, considered in this paper, is the

978-1-7281-2862-7/19/$31.00 ©2019 IEEE
DOI 10.1109/DSD.2019.00026

ORB-SLAM2 application [10], [11] because it is a versatile, complete, and the most accurate SLAM method supporting monocular, stereo, and RGB-D cameras, thus it is very suitable for use on UAVs including small drones. More specifically, the paper contributions are summarized as follows:

- We propose and realize a parallel implementation of the ORB-SLAM2 application using the Kahn Process Network (KPN) [16] model of computation. To the best of our knowledge, our work is the first to utilize the KPN model on SLAM. Having ORB-SLAM2 modelled and implemented as a KPN enables efficient mapping and execution of SLAM onto embedded multi-processor platforms as well as it enables the application of KPN model transformations [17], [18] to exploit different forms of parallelism thereby increasing the performance of the SLAM method on embedded multi-processor platforms.
- As a consequence of the above contribution, we show how to apply state-of-the-art and generic KPN model transformations such as process replication and splitting in order to exploit data-level and pipeline parallelism. By doing this, we can increase the performance of ORB-SLAM2 up to 18.21 fps on the existing popular embedded platform ODROID-XU4 [19] which is based on the ARM big.LITTLE multi-process architecture. Our achieved performance (up to 18.21 fps) is up to 3.72 times higher than the performance (around 4.9 fps) of the existing open-source thread-based parallel implementation of ORB-SLAM2 executed by us on the same platform.
- We propose and implement, on top of the Linux operating system, an environment for efficient execution of applications modelled as KPNs. We use this environment to do real implementations and tests of ORB-SLAM2 on the ODROID-XU4 platform and to obtain all experimental results presented in this paper.
- By using the three aforementioned contributions, we perform a simple design space exploration (DSE) to investigate the trade-off between system performance and power consumption when alternative ORB-SLAM2 KPNs are executed on different configurations of the ODROID-XU4 platform. The obtained results of this DSE clearly show that it is possible to run SLAM on a small embedded platform in real time (10 to 20 fps) with a limited power budget of 9 to 12 W, thereby enabling cognitive autonomy on small drones.

The remainder of the paper is organized as follows: Section II gives an overview of the related work. Section III gives background information including details about the embedded platform we use and the ORB-SLAM2 application we work on. Section IV presents the KPN model and transformations on ORB-SLAM2. Section V introduces our execution environment for KPNs. Section VI presents our DSE and obtained results. Finally, Section VII ends the paper with conclusion.

II. Related Work

To the best of our knowledge, our work is the first trying to efficiently implement and run the ORB-SLAM2 application on a multi-processor embedded platform based on the ARM big.LITTLE architecture. Therefore, in this section, we discuss some related works of implementing the ORB-SLAM2 application on other embedded platforms. Moreover, we consider that SLAM acceleration is very important in order to enable cognitive autonomy on small drones which is also one of our major contributions. So, we also discuss some related works on accelerating the SLAM algorithm itself.

Bourque [13] utilizes GPU devices on Jetson TX2 to enable on-board real-time visual ORB-SLAM. For the bottleneck Tracking thread, he moves the computational intensive part to the GPU by using asynchronous CUDA kernels for efficient acceleration. This approach achieves performance of up to 14.51 fps, on average, which meets real-time performance requirements. However, [13] does not consider the system power consumption. GPU devices consume a lot of power when doing large scale parallel computation which may cause reduced flying time for a small UAV system with limited energy budget. In contrast, we utilize the KPN model and its transformations for the ORB-SLAM implementation on a multi-process system. Our work on ORB-SLAM can achieve performance of up to 18.21 fps with only 8 processor cores. Moreover, we are able to perform DSE, thereby investigating the trade-off between system performance and power consumption and finding a system configuration with real-time performance and limited power budget for a given range of flying time.

Fang et al. [14] considers both real-time performance and power consumption when implementing and running ORB-SLAM. FPGA-based hardware is designed for dealing with the feature extraction computation - the most heavy task of ORB-SLAM. At the same time, as the FPGA hardware system consumes much energy when running, the system clock is set to only 203 MHz in order to meet the energy budget. However, the energy consumption of $5.6W$ is calculated only for this particular feature extraction hardware accelerator instead of the whole ORB-SLAM system. The energy consumption of the whole SLAM system including the un-accelerated parts executed on ARM processors still exceeds the energy budget limitation, in which case, it still cannot be utilized on a small drone. In contrast, our method balances acceleration and energy consumption for the whole SLAM system and is more credible and useful when running applications on a small drone. What is more, our method is based on software design and is more flexible when doing transformations like process replication and communication channel assignment. At the same time, we utilize the KPN model and its transformations on top of Linux, so our method is an application and platform agnostic method which can be easily utilized on other similar embedded platforms. Please note that some existing tool flows, e.g., Daedalus [20], [21], can automatically parallelize a given sequential application and provide the parallel KPN model of the application. However, these tool flows impose certain restrictions for a given application source code as input, e.g., an application written as a static affine nested loop program (SANLP) in C in Daedalus, and using such existing tool flows for parallelizing complex applications, e.g., SLAM written in C++, is either time consuming or infeasible. Therefore, manual parallelization is necessary and inevitable for complex applications.

Some researchers have attempted to accelerate SLAM by reducing the complexity of the algorithm itself. For example, DP-SLAM [22] is an algorithm based on distributed particle mapping instead of traditional predetermined landmarks. Real-time performance and accuracy are achieved. However, this algorithm cannot be utilized on a small drone due to its dependency on a laser range finder - a relatively heavy weight equipment (appropriate for a ground autonomous robot systems) that cannot be mounted on small drones. In contrast, our approach can be utilized on a small drone because the ORB-SLAM2 application relies only on a monocular on-board camera, which is much lighter and can be easily carried by a small drone into the sky.

Any SLAM algorithm relies on a place recognition procedure [23]. A fast place recognition technique is presented in [24], detecting loop closure of the real-time position of a moving robot and establishing feature point correspondence between image sequences in real time by only using a conventional CPU and a single camera. Both reliability and efficiency of the localization and the building of a map are shown on very different public datasets depicting indoor, outdoor, static and dynamic environments. However, this approach has a limitation when applied on small drones because the fast place recognition technique cannot handle severe situations like inevitable vibration and drastic movement of a small drone. The reason is that this technique lacks rotation and scale invariance for an image sequence. This limits the flying speed and movement amplitude of a small drone to a large extent. In contrast, the ORB-SLAM2 application, we use, has a complete feature point correspondence between images which makes a small drone relatively robust in unstable situations caused by drastic movement, vibration, and high speed.

III. Background

In this section, we introduce the hardware platform, the functionality/open source reference implementation of the ORB-SLAM2 application we consider, as well as the EuRoC MAV Dataset we utilize when performing experiments.

A. Hardware Platform

ODROID-XU4 [19] is the heterogeneous multi-processing embedded platform, we use, to execute our application, to increase the performance, and to explore the design space in order to find the trade-off between the system real-time performance and power consumption. The ODROID-XU4 has the Samsung Exynos5422 chip, where there are two clusters on the chip, one quad core Cortex-A15 (big) and one quad core Cortex-A7 (LITTLE). The big core runs at a maximal frequency of 2.0 GHz and is considered as performance-efficient core. The LITTLE core runs at a maximal frequency of 1.4 GHz and is considered as energy-efficient core. Basically, this ARM big.LITTLE architecture is utilized on many portable devices like mobile phones and robots. The cores in each cluster operate at the same voltage and frequency level. The voltage and frequency level of a cluster can be changed within a specific range. Different combinations of utilized big.LITTLE cores and operating frequencies give us a diversity of possible system configurations to perform DSE in order to find configurations with system performance and power consumption which meet our requirements. Therefore, in this paper, we utilize ODROID-XU4 as our hardware platform for obtaining real-world and credible experimental results.

B. Application

ORB-SLAM2 [10], [11] is a versatile and accurate, complete SLAM application supporting monocular, stereo, and RGB-D cameras which are commonly used in automation and robotic systems. It is an ORB feature-based method [25]. ORB are binary features invariant to rotation and scale (in a certain range), resulting in a very fast feature recognition with good invariance to view points. Thus, ORB-SLAM2 guarantees efficiency when computing and matching feature points for every frame in an image sequence. This ORB-SLAM2 application first pre-processes the input image sequence to extract ORB features, and then the same ORB features are used in the following image processing tasks. In general, there are three main tasks in the ORB-SLAM2 application: Tracking, Local Mapping, and Loop Closing. Tracking is responsible for localizing a robot with the help of finding feature matches in every frame by comparing the current frame features with previously extracted ORB features stored in a local map. Also, it decides when to insert a new key frame for the whole map. Local Mapping takes over the new key frames and perform local Bundle Adjustment [26] to reconstruct the surrounding environment optimally. Loop Closing searches for loops in a robot movement path by analyzing every new key frame. If a loop is detected, a similarity transformation is then performed, and accumulated drift in the robot path is reported to the SLAM system for correction. In this way, a run-time accurate algorithm is provided.

When this ORB-SLAM2 application is implemented as described in [11], Tracking, Local Mapping, and Loop Closing are realized as three main threads that run concurrently. These three threads constitute the SLAM system, as shown in Fig.1. For each frame entering the SLAM system, only task-level parallelism is exploited when the frame is processed. However, neither data-level nor pipeline parallelism is exploited to process multiple frames simultaneously.

Even though the ORB-SLAM2 application supports various input sensors like monocular, stereo, and RGB-D cameras, our study considers only a monocular camera because it is light enough to be installed on a small drone. The evaluation on 29 popular public datasets shows that ORB-SLAM2 is able to achieve state-of-the-art accuracy, thereby being the existing most accurate SLAM solution in most cases and is suitable for UAV computer vision.

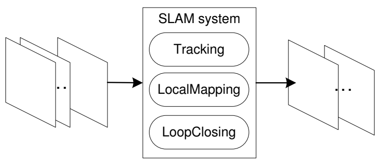

Fig. 1: Original SLAM [11]

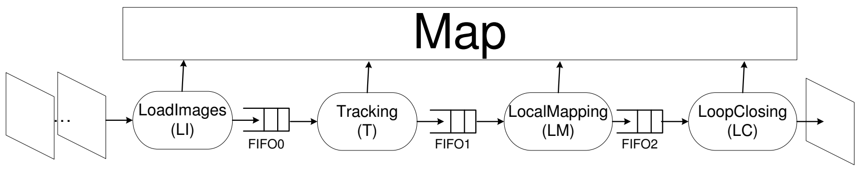

Fig. 2: KPN model of ORB-SLAM2

C. Datasets

For our experiments, we use the EuRoC MAV Dataset [27], designed by the autonomous system lab of ETH Zurich. It contains images, synchronized IMU measurements, and accurate motion and structure ground-truth. This dataset is captured by a real drone in an indoor environment, thus it is very suitable for our case study.

IV. KPN MODEL AND TRANSFORMATIONS

In this section, we explain our KPN model and the transformations, we apply on the ORB-SLAM2 application.

A. KPN Model of ORB-SLAM2

In order to exploit efficiently the parallelism available in ORB-SLAM2 by processing multiple frames simultaneously, we propose and realize a parallel implementation of the ORB-SLAM2 application using the KPN model of computation. The KPN model of computation is a network of concurrent autonomous processes that communicate data in a point-to-point fashion over FIFO channels, using blocking read/write synchronization primitives. The read primitive blocks the execution of a process if the current channel from which a process reads data is empty. Similarly, the write primitive blocks the execution of a process if the current channel to which a process writes data is full. In this way, the autonomous concurrent processes of the KPN model operate and synchronize based on the status (full/empty) of the communication channels.

Our specific KPN model of ORB-SLAM2 is shown in Fig. 2 and it consists of four main processes, namely, LoadImages (LI), Tracking (T), LocalMapping (LM), and LoopClosing (LC). They are connected via three communication channels, namely, FIFO0, FIFO1, and FIFO2. Also, a shared memory is accessible to all four processes. Each process in the network is specified as a sequential program that executes concurrently with other processes. Process LoadImages reads an image sequence captured by a camera and formats the image data for further processing. Please note that LoadImages is a part of the Tracking process in the original SLAM shown in Fig. 1. However, to improve the application performance, we split the Tracking process in the original SLAM into two processes LoadImages and Tracking that are executing in pipeline fashion. The computation part of the following processes, Tracking, LocalMapping, and LoopClosing, is similar to the one explained in Section III.2. When executing, these processes produce key frames, key feature map points, and a covisibility graph which together constitute a Map of the surrounding environment. This Map is stored and updated in the shared memory during image sequence processing. Writing data to the map is synchronized by mutexes to avoid conflicts. At the same time, the key frames are the only data tokens communicated between processes via the FIFOs. No data tokens are communicated between different processes via (hidden) global shared memory structures.

Our KPN model of the ORB-SLAM2 application has the following favourable characteristics:

- It is deterministic, which means that irrespective of the chosen schedule to execute the processes, always the same input/output relation exists. This gives us a lot of scheduling freedom that we can exploit when mapping and executing the processes onto a multiprocessor system.
- The inter-process synchronization is done by a blocking read and a blocking write on FIFO channels. This is a very simple synchronization protocol that can be realized easily and efficiently in hardware or software.
- Different processes of the KPN model can run autonomously and explicitly communicate/synchronize via the communication FIFO channels. No data tokens are communicated between different processes via other hidden channels. Therefore, a generic transformation such as process replication can be easily, systematically, and effectively applied on our KPN model.

Compared with the ORB-SLAM2 implementation in [11], very briefly described in Section III.2, which exploits only task-level parallelism inside one frame, our KPN model of ORB-SLAM2 in Fig. 2 exploits pipeline parallelism by processing multiple frames simultaneously, i.e., four frames are processed at the same time. As a consequence, our KPN model of ORB-SLAM2 achieves accelerated performance of up to 7.1 fps compared to 4.9 fps of the original ORB-SLAM2 in Fig. 1. For a real-world scenario with a small drone, this improved performance is still insufficient to meet the real-time performance requirement of 10 to 20 fps. However, our KPN model of ORB-SLAM2 allows to apply generic transformations for further acceleration.

B. Transformation of ORB-SLAM2 KPN model

In our initial KPN model of ORB-SLAM2 in Fig. 2, there are four processes in the main pipeline utilizing four CPU cores when executed on the ODROID-XU4 platform. Given the fact that further acceleration of ORB-SLAM2 is needed to reach real-time performance and only 4 out of the 8 CPU cores on ODROID-XU4 are utilized, we transform our KPN model by replicating bottleneck processes in order to create more parallel processes, thereby speeding up the computations. By monitoring the status of FIFO0, FIFO1 and FIFO2 in our initial KPN model of ORB-SLAM2, we find that the LoadImages process is blocked most of the time on writing to FIFO0

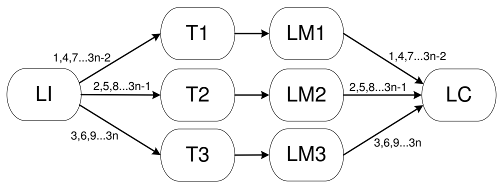

Fig. 3: Transformed KPN model

because FIFO0 is constantly full, as well as the LoopClosing process is blocked most of the time on reading from FIFO2 because FIFO2 is constantly empty. This means that processes Tracking and LocalMapping are the bottleneck processes in the pipeline. Therefore, we replicate each bottleneck process twice in order to fully utilize the 8 CPU cores on ODROID-XU4 by having 8 concurrent processes as shown in Fig. 3. By this KPN model transformation, data-level and pipeline parallelism is exploited when executing our transformed model. This is because 8 frames are processed simultaneously, compared to only 4 frames in our initial KPN model shown in Fig. 2.

After loading images and formating them into frames, process LoadImages distributes the frames to the replicas of the Tracking processes. As shown in Fig. 3, from LoadImages (LI) to Tracking 1 (T1), Tracking 2 (T2), and Tracking 3 (T3), we distribute the frames of the input sequence alternately. Then, when the frames are processed by Tracking and LocalMapping, they will be collected by LoopClosing (LC) at the end of the pipeline in exactly the same order.

When our transformed KPN model in Fig. 3 is executed in the ODROID-XU4 platform, we map the processes to the CPU cores as follows: LoadImages (LI), Tracking 1 (T1), LocalMapping 1 (LM1), and LoopClosing (LC) are mapped on the four big cores. Tracking 2 (T2), LocalMapping 2 (LM2), Tracking 3 (T3), and LocalMapping 3 (LM3) are mapped on the four LITTLE cores. By doing this, all of the 8 cores on the ODROID-XU4 platform are fully utilized and the system performance reaches 18.21 fps, thereby meeting our real-time performance requirement of 10 to 20 fps.

Thanks to our KPN model of ORB-SLAM2 and the possibility to apply transformations on it, if more CPUs are available, for example on other hardware platforms, we can easily and fully utilize them to get the maximum performance out of the platform by continuing to transform the model and revealing/exploiting more parallelism.

V. Execution Environment for KPNs

In this section, we present the execution environment, we have designed and developed, for specifying an application as a KPN and running it on top of Linux. First, in Section V.1, we describe how a KPN topology is specified by creating FIFOs and processes, connecting them, and launching the execution of the specified KPN. Second, we give more details on how the functional behavior of a KPN process is specified in Section V.2. Finally, in Section V.3, we show how the FIFO communication and blocking read/write synchronization mechanism between KPN processes is realized by introducing

Listing 1: Example of KPN topology specification

```
 1  void main(int argc, char **argv){
 2  /* Create FIFO0 */
 3  size_fifo0_in_tokens = 10;
 4  size_token0=sizeof(KeyFrame)/sizeof(int)
    +(sizeof(KeyFrame)%sizeof(int)
    +(sizeof(int)-1))/sizeof(int);
 5  size_fifo0 = size_fifo0_in_tokens*
        size_token0;
 6  //Allocate memory for FIFO0
 7  fifo0 = calloc(size_fifo0+2, sizeof(int));
 8  /* Create FIFO1 and FIFO2*/ {...}
 9  /* Create and launch process Tracking */
10  threadInfo.core_id = 1;
11  threadInfo.inFIFO = fifo0;
12  threadInfo.outFIFO = fifo1;
13  threadInfo.size_token_inFIFO = size_token0;
14  threadInfo.size_token_outFIFO = size_token1
        ;
15  threadInfo.size_inFIFO = size_fifo0;
16  threadInfo.size_outFIFO = size_fifo1;
17  thread Tracking(&Tracking, &threadInfo);
18  /* Create and launch process ReadImages,
        LocalMapping, and LoopClosing */{...}
19  /* Final process synchronization */
20  ReadImages.join(); Tracking.join();
21  LocalMapping.join(); LoopClosing.join();}
```

the software read/write communication and synchronization primitives we have designed.

A. KPN topology specification and launching

The topology of a KPN model is specified using the C++ language in the top-level `main()` function and consists of three main parts. The first part specifies the creation of all FIFOs and the allocation of memory for them. Taking data type `int` to be the minimum data unit which can be stored in a FIFO, the size of the FIFO is calculated and the necessary memory is allocated using the standard `calloc()` function. The second part specifies the connections of the created FIFOs to KPN processes as well as the creation and launching of the processes as threads. We use POSIX Threads (Pthreads) [28] and the corresponding API integrated in Linux. Finally, in the third part, we synchronize the different threads by waiting for all of them to finish before exiting the main program.

Consider the KPN model of ORB-SLAM2 in Fig. 2 as an example. The KPN topology specification and launching is illustrated in Listing 1. Firstly, we create and set parameters for the three FIFOs (see Lines 2 to 8). For example, consider FIFO0 specified in Lines 2 to 7. We set the required size of FIFO0 in number of data tokens as shown in Line 3. Then the size in `int` of one token (in this example a complex data object KeyFrame) is calculated as shown in Line 4. Having these two parameters, we can calculate the size of FIFO0 in number of `int` (Line 5) and allocate the necessary memory (Line 7) for FIFO0 by calling the standard function `calloc()` provided in Linux. Here, we allocate two more memory cells for FIFO0 in order to store the `read/write`

Listing 2: Example of KPN process specification

```
1 void Tracking(void *threadarg){
2 threadInfo = (threadInfo *)threadarg;
3 setaffinity(threadInfo.core_id);
4 while(1){
5 /* read data from FIFO and store in CPU */
6   readSWF_CPU(threadInfo.inFIFO,
    memobj_cpu, threadInfo.size_token_inFIFO,
    threadInfo.size_inFIFO);
7 /* Computation part of Tracking */{...}
8 /* write data from CPU to FIFO */
9   writeSWF_CPU(threadInfo.outFIFO,
    memobj_cpu, threadInfo.size_token_outFIFO,
    threadInfo.size_outFIFO);}}
```

Listing 3: Read Primitive

```
1 void readSWF_CPU(void* pFIFO,void* memobj_cpu
    , int size_token,int size_FIFO){
2 while(((int*)pFIFO)[0]==((int*)pFIFO)[1])
3  {pthread_yield();}
4 int r_cnt=((int*)pFIFO)[1];
5 for (int i=0; i<size_token; i++){
6   ((int*)memobj_cpu)[i]=
    ((int*)pFIFO)[(r_cnt & 0x7FFFFFFF)+2+i];}

7 r_cnt+=size_token;
8 if((r_cnt & 0x7FFFFFFF)==size_FIFO){
9   r_cnt &= 0x80000000; r_cnt ^= 0x80000000;}
10 ((int*)pFIFO)[1]=r_cnt;}
```

pointers explained in Section V.3. The creation of FIFO1 and FIFO2 is done in the same way as described above. Secondly, we create and launch the four processes as threads in Lines 9 to 18. For example, consider process Tracking specified in Lines 10 to 17. We define a data structure `threadInfo` to store several items, needed to launch the process and connect it to FIFOs, including `core_id`, `inFIFO`, and `outFIFO`. `core_id` specifies on which CPU core process Tracking should run (Line 10). `inFIFO` and `outFIFO` are pointers to the FIFO channels connected to process Tracking, for reading and writing data tokens, respectively (see Lines 11 to 16). Process Tracking is created and launched as Pthread Tracking in Line 17. The data structure `threadInfo` and a pointer to function `Tracking()`, explained in Section V.2, are the two parameters passed to Pthread Tracking. We follow the aforementioned approach to create and launch the other processes (ReadImages, LocalMapping, and LoopClosing). Finally, in order to make sure that all processes finish before exiting from the main program, we use the `thread.join()` method, provided by the Pthread API (see Lines 19 to 21).

B. Behaviour specification of KPN processes

The functional behaviour of a KPN process is specified in C++ as a function. The data structure `threadInfo`, explained in Section V.1, is the only parameter passed to the function. Within the function, there are no limitations on the C++ code that can be used to describe the computation done by the process. However, the process must communicate data tokens and synchronize with other processes by using only our `read/write` software primitives introduced in Section V.3. Consider again process Tracking in Fig. 2 as an example. Listing 2 shows how we specify its functional behavior. Firstly, we instruct Linux on which CPU core to run process Tracking by `setaffinity()` in Line 3. Then, process Tracking starts a while loop to do its computation (in Lines 4 to 9). Inside the loop, process Tracking reads a data token from FIFO0 (`inFIFO`) and stores it to local memory `memobj_cpu` (Line 6), does computation (Line 7), and writes data from local memory `memobj_cpu` to FIFO1 (`outFIFO`)(Line 9). Our software primitives `readSWF_CPU()` and `writeSWF_CPU()` are used for

Listing 4: Write Primitive

```
1  void writeSWF_CPU(void* pFIFO,void*
       memobj_cpu, int size_token,int
       size_FIFO){
2  while(((int*)pFIFO)[1]==
   (int)(((int*)pFIFO)[0]^0x80000000))
3  {pthread_yield();}
4  int w_cnt=((int*)pFIFO)[0];
5  for (int i=0; i<size_token; i++){
6  ((int*)pFIFO)[(w_cnt&0x7FFFFFFF)+2+i]=
   ((int*)memobj_cpu)[i];}
7  w_cnt+=size_token;
8  if((w_cnt & 0x7FFFFFFF)==size_FIFO){
9  w_cnt &= 0x80000000; w_cnt ^= 0x80000000;}
10 ((int*)pFIFO)[0]=w_cnt;}
```

data communication and synchorinization. `readSWF_CPU()` blocks the execution of process Tracking when `inFIFO` is empty and `writeSWF_CPU()` blocks the process when `outFIFO` is full. If the process is blocked, the control is returned back to Linux to try to execute other processes mapped on the same CPU core or to continue executing the currently blocked process later when data/space is available on its FIFOs.

C. Data Communication and Synchronization between KPN processes

As we discussed in Section V.1 and Section V.2, we have designed read and write primitives for data communication and synchronization between KPN processes. These primitives are shown in Listing 3 and Listing 4. Four parameters are passed to the primitives, i.e., a pointer to the memory array allocated for a FIFO (`pFIFO`), a pointer to local CPU memory (`memobj_cpu`), the size of the data tokens communicated via the FIFO (`size_token`), and the size of the FIFO (`size_FIFO`). The primitives realize a FIFO as a circular buffer [29] using the memory array (`pFIFO`) allocated for the FIFO. The read and write primitives implement the read/write access to the data stored in the circular buffer by manipulating two pointers (write and read) to the memory array. These two pointers are stored in the first two cells of the memory array

TABLE I: Mappings considered in DSE

	Cortex-A15 (big)				Cortex-A7 (LITTLE)			
mapping	core4	core5	core6	core7	core0	core1	core2	core3
map1	LI	T	LM	LC	–	–	–	–
map2	LI	T1	LM1	LC	T2	LM2	–	–
map3	LI	T1	LM1	LC	T2	LM2	T3	LM3

(`pFIFO[0] and pFIFO[1]`). Consider the read primitive in Listing 3. First, it checks whether the FIFO is empty by comparing the relative position of the read and write pointers (Line 2). As long as the FIFO is empty, the read primitive gives control back to Linux by `pthread_yield()` (Line 3). When there is data available in the FIFO, one data token at the head of the FIFO is copied from the FIFO to the local CPU memory in Lines 5 to 6. Finally, the position of the read pointer (`r_cnt`) is updated in Lines 7 to 10 to point to the new head of the FIFO. The write primitive in Listing 4 has similar behaviour with the one described above.

VI. Design Space Exploration

In this section, we describe the simple DSE, we have performed, in order to investigate the trade-off between system performance and power consumption when alternative ORB-SLAM2 KPNs are executed on different configurations of the ODROID-XU4 platform. The goal of this DSE is to verify that, thanks to our contributions presented in this paper, ORB-SLAM2 can run on a small embedded platform in real time (10 to 20 fps) with a limited energy budget. First, we explain the setup for our DSE in Section VI.1. Then, in Section VI.2, we justify the considered constraints in our DSE based on a real-life scenario. Finally, in Section VI.3, we present the results of our DSE.

A. DSE setup

By using our execution environment (Section V), we map and run the ORB-SLAM2 KPN model and its transformed alternatives (Section IV) onto the ODROID-XU4 small embedded platform (Section III.1) which features an ARM big.LITTLE 8-core system, and we explore 84 system configurations. Each configuration is described by a tuple $(mapping, f_{big}, f_{LITTLE})$, where $mapping \in \{map1, map2, map3\}$ is a specific KPN model of ORB-SLAM2 mapped onto specific cores of the ARM big.LITTLE system, $f_{big} \in \{0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}$ is the frequency in GHz at which the big cores run, and $f_{LITTLE} \in \{0.8, 1.0, 1.2, 1.4\}$ is the frequency in GHz at which the LITTLE cores run. The three mappings $map1$, $map2$, and $map3$ considered in our DSE are given in Table 1. For example, $map1$ is the KPN model in Fig. 2 mapped onto the 4 big cores and $map3$ is the KPN model in Fig. 3 mapped onto all 8 cores. The symbol ”–” in the table denotes an unutilized core.

For each system configuration, we obtain its system performance and average power consumption by performing real measurement when executing ORB-SLAM2 on the ODROID-XU4 platform. The system performance in fps is the inverse

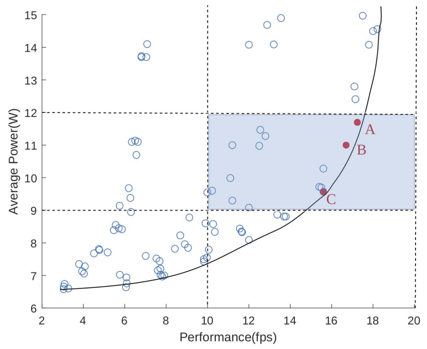

Fig. 4: Performance-Power Consumption Trade-off

of the measured average time interval needed for processing a single frame, using `steady_clock::now()` function in Linux. The average power consumption of the ODROID-XU4 platform is $P_{ave} = (V \times \int_0^t I(t)dt)/t$, where the current $I(t)$ is obtained by precisely measuring (sampling) the current drawn by the ODROID-XU4 platform during the time interval t of the ORB-SLAM2 execution under the platform operating voltage V.

B. Considered constraints in DSE

By performing our DSE, we look for system configurations where the ORB-SLAM2 performance is constrained in the interval of 10 to 20 fps ([10, 20]fps) with a constrained power consumption budget of 9 to 12 W ([9, 12] W). The performance constraint of [10, 20]fps comes from the required image processing speed of real-life applications using SLAM as discussed in Section 1. The power consumption constraint of [9, 12]W comes from a real-life scenario where a small drone has to fly for 19 to 20 minutes ([19, 20]min) without recharging the battery. In order to enable such small drone to fly for [19, 20]min, the constrained power budget for our ODROID-XU4 embedded platform is determined as follows. We mount the ODROID-XU4 platform, which weight is around 100 g, on a small quad-copter which weight is 573 g and it has 4 T-motors [30] and a total battery capacity of 1800 mAh. Thus, the total weight of the considered quad-copter is $100 + 573 = 673g$, which can be lifted in the sky by the 4 T-motors where the power consumption of the motors is $P_{motors} = 71W$. Then, the power budget left for the ODROID-XU4 platform is $P_{odroid} = (C \times V)/t - P_{motors} = (1800mAh \times 14.8V)/[19, 20]min - 71W = [9, 12]W$, where C is the battery capacity, t is the required flying time range, and V is the operating voltage of the battery.

C. DSE results

In Fig. 4, we show the obtained results for all 84 system configurations we have explored. Every point in Fig. 4 corresponds to a system configuration where the X coordinate is the system performance and the Y coordinate is the system average power consumption. The points near the solid black curve are the pareto-optimal design points from which we

TABLE II: Corresponding configurations of pareto points

Point	Mapping	Frequency		System speed	Average power	Flying time
		big	LITTLE			
A	$map3$	1.8GHz	1.0GHz	17.24 fps	11.70W	19min 20s
B	$map3$	1.8GHz	0.8GHz	16.70 fps	11.00W	19min 29s
C	$map3$	1.6GHz	0.8GHz	15.60 fps	9.57W	19min 50s

have to select corresponding system configurations which meet our design constraints (Section 6.2), i.e., system performance of 10 to 20 fps and power consumption of 9 to 12 W. All points (system configurations) within the shaded area in Fig. 4 meet these constraints but the solid red points A, B, and C correspond to the pareto-optimal system configurations that we look for. These configurations are described in Table 2. For example, configuration A in Table 2 indicates that ORB-SLAM2 achieves real-time performance of 17.24 fps consuming 11.70 W of power which guarantees a flying time of 19min and 20s of the quad-copter, described in Section 6.2, if the KPN model in Fig. 3 is run on the ODROID-XU4 platform mounted on the quad-copter. For this system configuration, the KPN processes are mapped on all ARM big.LITTLE cores according to $map3$ in Table 1, and the big and LITTLE cores run at 1.8 and 1.0 GHz clock frequency, respectively. The obtained results of this DSE clearly show that it is possible to run SLAM on a small embedded platform in real time (10 to 20 fps) with a limited power budget of 9 to 12 W, thereby enabling cognitive autonomy on small drones.

VII. Conclusions

In this paper, we presented an ORB-SLAM2 case study which clearly showed that cognitive autonomy on small drones is possible by efficient on-board embedded computing. To enable such cognitive autonomy, we modeled and implemented ORB-SLAM2 as a KPN which exploits pipeline parallelism and enables efficient mapping and execution of ORB-SLAM2 onto a small embedded multi-processor platform such as ODROID-XU4. Moreover, our KPN model enables the application of generic model transformations to exploit data-level parallelism as well. As a consequence, we achieved an increased performance of ORB-SLAM2 up to 18.21 fps. Finally, the results of our DSE for performance-power consumption trade-off showed the feasibility of running ORB-SLAM2 on the small embedded platform ODROID-XU4 in real time (10 to 20 fps) with a limited power budget of 9 to 12 W.

References

[1] D. Floreano and R. J Wood. Science, technology and the future of small autonomous drones. *Nature*, 521(7553):460, 2015.

[2] S. Thrun. Simultaneous localization and mapping. In *Robotics and cognitive approaches to spatial mapping*. Springer, 2007.

[3] M. Dissanayake et al. A solution to the simultaneous localization and map building (slam) problem. *IEEE Transactions on robotics and automation*, 2001.

[4] M. Mattamala et al. The nao backpack: An open-hardware add-on for fast software development with the nao robot. *arXiv preprint arXiv*, 2017.

[5] T. Caselitz et al. Monocular camera localization in 3d lidar maps. In *IROS*, 2016.

[6] S. Ross et al. Learning monocular reactive uav control in cluttered natural environments. In *ICRA*, 2013.

[7] J. Langelaan and S. Rock. Towards autonomous uav flight in forests. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2005.

[8] D. Hulens et al. How to choose the best embedded processing platform for on-board uav image processing? In *VISAPP*, 2015.

[9] A. Wada et al. A surveillance system using small unmanned aerial vehicle (uav) related technologies. *NEC Technical Journal*, 8(1), 2015.

[10] R. Mur-Artal et al. Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5), 2015.

[11] R. Mur-Artal and J. D Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[12] H. Chao et al. Band-reconfigurable multi-uav-based cooperative remote sensing for real-time water management and distributed irrigation control. In *IFAC World Congress*, volume 17, pages 11744–11749, 2008.

[13] D. Bourque. *CUDA-Accelerated Visual SLAM For UAVs*. PhD thesis, WORCESTER POLYTECHNIC INSTITUTE, 2017.

[14] W. Fang et al. Fpga-based orb feature extraction for real-time visual slam. In *ICFPT*, 2017.

[15] P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 17, 2011.

[16] K. Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[17] S. Meijer et al. Combining process splitting and merging transformations for polyhedral process networks. In *ESTIMedia*, 2010.

[18] S. Meijer et al. On compile-time evaluation of process partitioning transformations for kahn process networks. In *CODES+ISSS*, 2009.

[19] Odroid. https://www.hardkernel.com.

[20] H. Nikolov et al. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2008.

[21] T. Stefanovet et al. Daedalus: System-level design methodology for streaming multiprocessor embedded systems on chips. *Handbook of Hardware/Software Codesign*, 2017.

[22] A. Eliazar and R. Parr. Dp-slam: Fast, robust simultaneous localization and mapping without predetermined landmarks. In *IJCAI*, volume 3, 2003.

[23] J. H. Lee et al. Place recognition using straight lines for vision-based slam. In *ICRA*, 2013.

[24] D. Gálvez-López and J. Tardos. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 2012.

[25] E. Rublee et al. Orb: An efficient alternative to sift or surf. In *ICCV*, 2011.

[26] E. Mouragnon et al. Monocular vision based slam for mobile robots. In *ICPR*, 2006.

[27] M. Burri et al. The euroc micro aerial vehicle datasets. *I. J. Robotics Res.*, 2016.

[28] D. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[29] S. Bhattacharyya et al. Memory management for dataflow programming of multirate signal processing algorithms. *IEEE Trans. Signal Processing*, 1994.

[30] T-motor. http://store-en.tmotor.com/goods.php?id=388.