INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

Object Tracking Using a Many-Core Embedded System

Laercio Minozzo

Dissertação apresentada à Escola Superior de Tecnologia e de Gestão de Bragança para obtenção do Grau de Mestre em Sistemas de Informação.

Trabalho orientado por:
José Rufino, José Lima
Paulo Lopes de Menezes, Arnaldo Cândido Jr.

Esta dissertação não inclui as críticas e sugestões feitas pelo Júri.

Bragança
2017

INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

Object Tracking Using a Many-Core Embedded System

Laercio Minozzo

Dissertação apresentada à Escola Superior de Tecnologia e de Gestão de Bragança para obtenção do Grau de Mestre em Sistemas de Informação.

Trabalho orientado por:
José Rufino, José Lima
Paulo Lopes de Menezes, Arnaldo Cândido Jr.

Esta dissertação não inclui as críticas e sugestões feitas pelo Júri.

Bragança
2017

Abstract

Object localization and tracking is essential for many practical applications, such as man-computer interaction, security and surveillance, robot competitions, and Industry 4.0. Because of the large amount of data present in an image, and the algorithmic complexity involved, this task can be computationally demanding, mainly for traditional embedded systems, due to their processing and storage limitations. This calls for investigation and experimentation with new approaches, as emergent heterogeneous embedded systems, that promise higher performance, without compromising energy efficiency.

This work explores several real-time color-based object tracking techniques, applied to images supplied by a RGB-D sensor attached to different embedded platforms. The main motivation was to explore an heterogeneous Parallella board with a 16-core Epiphany co-processor, to reduce image processing time. Another goal was to confront this platform with more conventional embedded systems, namely the popular Raspberry Pi family. In this regard, several processing options were pursued, from low-level implementations specially tailored to the Parallella, to higher-level multi-platform approaches.

The results achieved allow to conclude that the programming effort required to efficiently use the Epiphany co-processor is considerable. Also, for the selected case study, the performance attained was bellow the one offered by simpler approaches running on quad-core Raspberry Pi boards.

Resumo

A localização e o seguimento de objetos são essenciais para muitas aplicações práticas, como interação homem-computador, segurança e vigilância, competições de robôs e Indústria 4.0. Devido à grande quantidade de dados presentes numa imagem, e à complexidade algorítmica envolvida, esta tarefa pode ser computacionalmente exigente, principalmente para os sistemas embebidos tradicionais, devido às suas limitações de processamento e armazenamento. Desta forma, é importante a investigação e experimentação com novas abordagens, tais como sistemas embebidos heterogéneos emergentes, que trazem consigo a promessa de melhor desempenho, sem comprometer a eficiência energética.

Este trabalho explora várias técnicas de seguimento de objetos em tempo real baseado em imagens a cores adquiridas por um sensor RBD-D, conectado a diferentes sistemas embebidos. A motivação principal foi a exploração de uma placa heterogénea Parallella com um co-processador Epiphany de 16 núcleos, a fim de reduzir o tempo de processamento das imagens. Outro objetivo era confrontar esta plataforma com sistemas embebidos mais convencionais, nomeadamente a popular família Raspberry Pi. Nesse sentido, foram prosseguidas diversas opções de processamento, desde implementações de baixo nível, específicas da placa Parallella, até abordagens multi-plataforma de mais alto nível.

Os resultados alcançados permitem concluir que o esforço de programação necessário para utilizar eficientemente o co-processador Epiphany é considerável. Adicionalmente, para o caso de estudo deste trabalho, o desempenho alcançado fica aquém do conseguido por abordagens mais simples executando em sistemas Raspberry Pi com quatro núcleos.

Acknowledgments

I am grateful to Professors José Rufino, José Lima, Paulo Lopes de Menezes, and Arnaldo Cândido Jr. for the guidance, dedication, patience, availability and support they have always shown during this work.

I also thank the various people involved in the double diploma program. Without them this work would not be possible. I would therefore like to express my gratitude to all those who, directly or indirectly, contributed to making this a reality.

Finally, I would like to thank all those who, in one way or another, have made it possible to carry out this dissertation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Object localization and tracking is a crucial task in several real-world domains, as augmented reality, human-computer interaction, security and surveillance, robot competitions and aerial vehicles, and Industry 4.0 [iS17], to name a few. The later, for instance, demands the collaboration between robots and humans; in this context, *perception* is one of the most important capabilities, that should be applied as fast as possible to guarantee expected/bounded reaction times.

One of the possible tasks to realize through a computer vision system is to recognize objects automatically. However, this task is not trivial, especially in the treatment of complex scenes, with variations in lighting, position, angle, scale, texture, shadows, deformations, occlusions and other characteristics. To address these issues, efforts are being made not only for image processing and computer vision, but also for areas such as pattern recognition, artificial intelligence, psychophysics, and cognitive sciences [CdFC95].

The growth of the processing capacity of embedded systems, and the parallel processing abilities of many modern Systems-on-Chip (SoCs), are playing an important role in real-time object localization and tracking, addressing more sophisticated image processing techniques, that are able to exploit the extra processing power available. In this regard, a recent trend is the emergence of energy efficient embedded heterogeneous systems, with powerful co-processors that may be used as accelerators in co-operation with their host.

This work documents the investigation, development and experimentation conducted on the use of a Parallella heterogeneous embedded system, connected to a Kinect sensor, in order to perform object tracking. It explores a simple technique (color segmentation), through several programming models (including hybrid programming), to take advantage of the concurrent/parallel computing capabilities of the Parallella host and its Epiphany co-processor. Furthermore, the portable nature of some of the developed approaches allowed its deployment and evaluation in several Raspberry Pi models. The ensuing comparison between the Parallela and the Raspberry Pi platforms allowed to derive important conclusions on the cost/benefit ratio of the most performant approaches in each platform.

After the introduction, this document is structured as follows:

- chapter 2 provides background information about object tracking, and on the hardware (embedded systems and RGB-D sensor) and software development tools used;
- chapter 3 starts with a high-level description of the tracking application; it then introduces an initial Pthreads version, along with a first round of evaluations and optimizations, paving the way for further enhancements (presented in chapter 4);
- chapter 4 enhances the initial application version, in the storage and computing domains; the resulting optimized Pthreads version becomes the foundation for hybrid versions, combining Pthreads with OpenMP, and/or with a low-level Parallella framework; intermediate evaluations supporting the choice of different implementation options are presented; the chapter ends with evaluation results from the final versions developed, deployed in the Parallella and in some Raspberry Pi models;
- chapter 5 concludes the document and points directions for future work.

Chapter 2

Background and Tools

This chapter introduces the background concepts and technological tools pertaining to this work. Specifically, these include concepts on object tracking and a presentation of the hardware platforms and software tools used.

2.1 Object tracking

In the context of this work, tracking can be defined as "the problem of estimating the trajectory of an object in an image" [AOM06]. There are several algorithms, applications and systems that solve the object tracking problem. These approaches depend on the object characteristics and features such as appearance, shape, context/environment or scenario, and the end use.

A well known target for object tracking is surveillance (human body tracking). In some of the works that address this topic [AC97, Gav99, MG01], human kinematics provide the basis for implementation, namely using articulated object models. Another object tracking end use is the learning of different views of an object, by training a set of classifiers, like support vector machines [Avi04] or Bayesian networks [PA04].

Object tracking can also be found in such different areas as industrial applications or robotics soccer. Specific examples include manipulators finding objects to perform pick and place operations (helping users in a collaborative task), or mobile soccer robots finding the ball and estimating its position.

Feature selection is the most critical role in tracking. The most desirable property of a visual feature is its uniqueness so that the objects can be easily distinguished in the environment. Feature selection provides a way to perform the image segmentation. This can be achieved using Mean-shift [CM99], Graph-cut [SM00] and Active Contours [CKS95]. Another approach is to use the colour (RGB or HSV) as a feature for histogram-based appearance representations, while for contour-based representation object edges are usually used as features. In general, many tracking algorithms use a combination of these features [Pas01].

Ready to use algorithms/routines can be found in well-known development platforms and frameworks. Mathworks' Matlab includes the Computer Vision System Toolbox, providing video tracking routines that can be used for tracking single or multiple objects. Matlab, however, runs only on x_86 platforms and is currently restricted to 64 bit running environments [Mat17], which prevents its use on most embedded systems, including the ones used in this work. OpenCV [Ope17a] also offers an API that performs object tracking [Ope17b]. However, this work did not use such API, due to the need of fine-grain control on the code, for parallelization purposes (OpenCV was still used, though for different tasks, like calibration, visualization and to support some data types).

2.2 Heterogeneous Embedded Systems

In recent years, advancements in embedded systems and, in particular, the emergence of Systems on Chip (SoC) (mostly propelled by the widespread adoption of mobile devices and the emergence of the Internet-of-Things), brought with it a growing processing capacity, coupled with low/modest power requirements. Nowadays, there a plethora of small single-board computers (SBCs) [bd17], built around these SoCs. They are usually open-platforms, running an open-source operating system (typically some distribution of Linux) which, coupled with standard development tools, offer tremendous flexibility, at a relatively low cost. In this regard, the Raspberry Pi [Fou17b] line of SBCs is perhaps the most well-established on the market and academia [Bro17], with an enormous ecosystem, ranging from industrial to educational and I&D scenarios.

Virtually all modern SBCs include a multi-core CPU, and the same happens with the Raspberry Pi, since the launch of the 4-core Raspberry Pi 2, in early 2015. Therefore, there has been an increasing interest in exploring the parallel computing capabilities of modern SBCs, to accelerate processing on demanding applications, including object tracking [INA$^+$16, TK17]. This trend also extends to heterogeneous embedded systems [KA11], where the main/host processor co-exists with additional devices, of a different architecture, that may be used to execute/accelerate tasks on its behalf. A common example of this is the presence of increasingly powerful Graphical Processing Units (GPUs) in SBCs, usable as numerical co-processors, beyond their native graphical capabilities [NVI]. Another example is provided by the Parallella board [ONUA14, L.11] – the main focus of this work –, coupling a 2-core ARM CPU with a grid of 16 (or more) Epiphany CPUs [Ada13a].

2.3 The Parallella Board

Parallella is a credit card sized, single-board computer (Figure 2.1) developed by Adapteva, running on Linux (Ubuntu-based). The intent behind its inception was to have high processing capabilities with low power consumption [ONUA14, L.11]. It was made available to general public in late 2013, after successful Kickstarter funding in 2012. There are several models currently available [Para], all offering an Epiphany III MIMD co-processor. This work explores the ”Parallella-16 Desktop Computer”, with USB 2.0 data ports (as required, to connect a Kinect sensor), and a micro-HDMI port (for visualization).

2.3.1 Hardware and Architecture

The Parallella board is a heterogeneous system, with processing elements based on two different architectures. The host component consumes up to 5W, and includes a 32 bit dual-core 667 MHz ARMv7 A9 microprocessor, serviced by 1 GByte of SDRAM.

Figure 2.1: Parallella board.

The co-processor consumes an additional 2W, and consists of a 16-core 2D-grid of 32 bit RISC CPUs, based on the Epiphany III architecture [Ada13a]. All Epiphany RISC cores (eCores or eNodes) interconnect through a Network-on-Chip (eMesh), that provides message passing. eCores run at 600 MHz, with 19.2 GFLOPS aggregated single-precision peak performance (2.74 GFLOPS/W, onve host and co-processor consume up to 7W).

The memory architecture of the Epiphany co-processor does not have an explicit hierarchy and has no caches; it is a distributed shared memory, with a partitioned global address space of 512 KBytes. Each eCore is assigned 32 KBytes of local memory (4 banks of 8 KBytes), for code, stack and data. Table 2.1 shows the memory map of an eCore's local memory space. Besides the four banks of 8 KBytes, it includes other slots reserved in the address space for register access and future expansion.

Fast inter-eCore local memory access is supported: an eCore may directly access the memory of another eCore using the eMesh Network-on-Chip. In order to do so, it is necessary to prefix the desired memory location with the ID of the target eCore in the eMesh Network-on-Chip. That ID is formed by concatenating two tags that depend on the coordinates of the eCore in the eMesh. Table 2.2 shows the 2D coordinates of each eCore (inside the table), and the corresponding tags (leftest column, and topmost line).

Description	Start Address	End Address	Size
Interrupt Vector Table	0x00000	0x0003F	64B
Bank 0	0x00040	0x01FFF	8KB-64B
Bank 1	0x02000	0x03FFF	8KB
Bank 2	0x04000	0x05FFF	8KB
Bank 3	0x06000	0x07FFF	8KB
Reserved for future memory expansion	0x08000	0xEFFFF	N/A
Memory Mapped Registers	0xF0000	0xF07FF	2048B
Reserved	0xF0800	0xFFFFF	N/A

Table 2.1: eNone Local Memory Map [Ada13a]

For instance, to access memory location `0x2000` of the eCore (0,1), the address to target would be `0x80902000`.

	80	90	A0	B0
80	(0,0)	(0,1)	(0,2)	(0,3)
84	(1,0)	(1,1)	(1,2)	(1,3)
88	(2,0)	(2,1)	(2,2)	(2,3)
89	(3,0)	(3,1)	(3,2)	(3,3)

Table 2.2: Prefix table for eCores remote memory addresses.

Buffers shared between the host and the eCores may be created in a 32 MBytes region (8 MBytes usable) of the host main memory, perceived by eCores as an external memory. These buffers may be used for the host and eCores to exchange data and synchronize (or only by eCores to directly store and manage private data), but access to this buffers is slow (even for the host), compared with inter-eCore local memory access. The host can also access directly the local memory of every eCore, but this is slower than access to the shared memory.

Each eCore contains two DMA general purpose channels to simplify data transfers to/from other eCores or shared memory. The DMA engine works at the same frequency as the eCore RISC CPU and can transfer 64-bits per clock cycle.

The two DMA channels have a 2D DMA mechanism that adds some flexibility to transfer data. This mechanism allows the data to be rearranged at the destination during the data copy by the DMA mechanism, depending on the configuration of some parameters in registers. This mechanism is implemented in hardware, but its operation can be represented by the pseudo-code in listing 2.1.

Listing 2.1: 2D DMA data transfer

```
for(int i=0;i<OUT_COUNTER;i++){
    for(int j=0;j<IN_COUNTER;j++){
        *dst=*src;
        dst+=IN_DST_STRIDE;
        src+=IN_SRC_STRIDE;
    }
    dst+=OUT_DST_STRIDE;
    src+=OUT_SRC_STRIDE;
}
```

The `OUT_COUNTER`, `IN_COUNTER`, `IN_DST_STRIDE`, `IN_SRC_STRIDE`, `OUT_DST_STRIDE`, and `OUT_SRC_STRIDE` represent the parameters that must be configured in special registers before the start of a DMA transaction. The `src` pointer holds the address of the source memory region and the `dst` pointer holds the address of the destiny memory region. This mechanism can transfer 8, 16, 32 or 64 bits for each cycle, needing to align the steps with the size used through the configuration of the parameters mentioned above.

Finally, it is possible to connect several Parallella boards in a cluster (aiming at performance gains), whether by using a fast inter-chip ”eMesh” interconnect (that extends the logical grid of Epiphany cores), or the on-board 1Gbps Ethernet port (for a more traditional cluster configuration). This work used a single board, postponing the exploration of those configurations for future work.

2.3.2 Application Development

Applications that take advantage of the Epiphany device may be developed using the low-level Epiphany Software Development Kit [Ada13b], or higher-level frameworks [Parb].

Epiphany Software Development Kit

The Epiphany Software Development Kit (eSDK) is a set of tools to assist in software development on the Parallella board. The eSDK is based on standard development tools including an optimizing C-compiler, functional simulator, and debugger. The Epiphany C-Compiler is based on GNU/GCC, and the programs are written in ANSI-C and C++ (partial support), not requiring any language extensions. The eSDK include also two important libraries: the Epiphany Hardware Abstraction Layer (eHAL) and the Hardware Utility Library (eLib). These two libraries provide a set of routines and data structures necessary to the development of C-based applications.

Epiphany Hardware Abstraction Layer The eHAL is a C-library that provides functionality of the Epiphany chip to the host side. The library interface is defined in the *e-hal.h* header file. This library manages the Epiphany chip based on the supplied hardware description file (HDF), which contains all the information about the Epiphany platform, such as chip arrangement, memory locations and defined sizes. This way, an application can be easily migrated between different Epiphany chips. This file is loaded on the initialization function to get the actual Epiphany chip attributes. The Epiphany chip attributes can be retrieved after with some API functions.

During program execution the application can allocate space in the shared memory. This space is defined from the beginning of the shared memory, and the eHAL provides functions to allocate, write, read and free external memory buffers. After an allocation in shared memory, the program can write and read data in allocated space only with special functions provided by the library. The same functions that access the shared memory can also access the local memory of every Epiphany eCores (a write operation is only possible if the eCore involved is idle; to ensure that eCores are idle at the beginning of program execution, it is necessary a reset operation after the initialization operation).

Hardware Utility Library The eLib is the C-library that provides functionality and hardware abstraction in the Epiphany eCores side. The eLib provides essential Epiphany architecture tasks that are not present in standard C and C++. These tasks are as follows: system register functions that allow write and read information of all eCore registers; the interrupt service functions, that are used to attach or detach interrupt-handlers; timer functions, that allow to set/get the timers values beside starting and stopping the two timers systems; DMA functions, that allow to control the DMA data transfer system; mutex system functions, that allow exclusive access of a single eCore to a shared resource, as well as a barrier function for synchronization between eCores program execution; coreId and work groups functions, allowing the programmer access to IDs and coordinates.

High-Level Approaches

Higher-level approaches and frameworks to explore the Epiphany accelerator include, among others, i) ePython [Bro16] (for Python-based parallel programming), ii) OpenSHMEM [RR16] (particularly suitable to Epiphany's partitioned global address space), iii) OpenMP [Ope13] via the OMPi compiler [APD15] (for automatic thread-based parallelization via code pragmas), and iv) MPI [mpi15] using the Threaded MPI implementation [RRPS15] (for distributed parallel computing on the Epiphany grid based on message passing). This flexibility results from the MIMD nature of Epiphany's architecture, that supports many parallel programming models.

Hybrid Approaches

Different approaches may also be mixed, in line with an Hybrid Programming approach, often used in Heterogeneous Systems. For instance, at the host-side, the application developed for this work explores conventional POSIX Threads (Pthreads) [But97], either isolated, or combined with OpenMP threads. For co-processor offloading, both the low-level eSDK eLib and the OMPi compiler were selected. These approaches give rise to several implementation combinations, later fully described and evaluated in this work.

2.3.3 Use Cases

Due to its specificities, like its memory architecture (see section 2.3.1), the Parallella board often requires a low-level approach to application development. There are, however, many examples that were able to explore its potential. For instance, Gener et al.

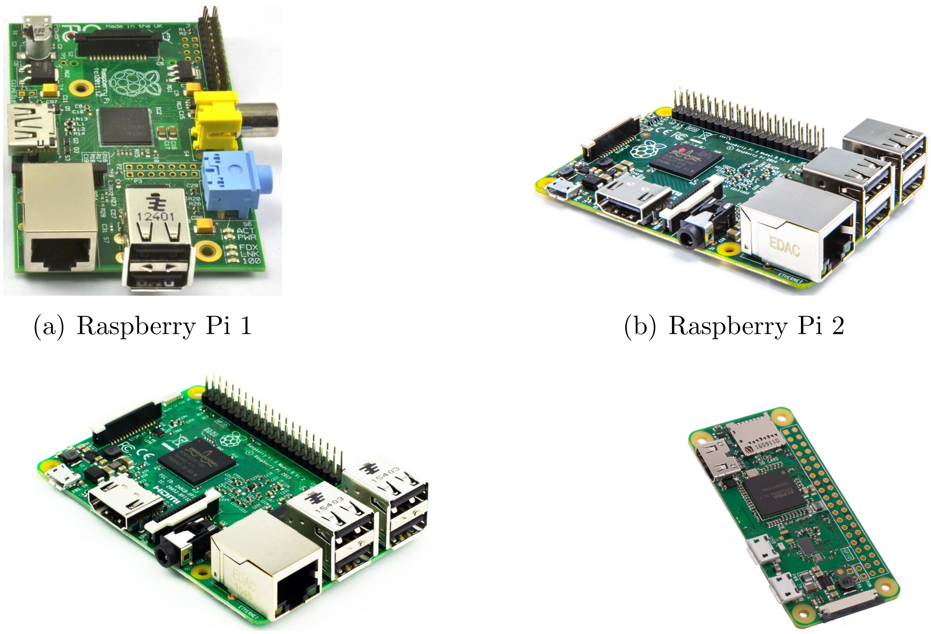
[GYG15] compared the Parallella with a GPU for spatial domain video filtering, showing that in such scenario the Parallella offers an efficient low-cost and low-power alternative. Taking advantage of the Epiphany co-processor, by using a combination of task and data parallelization, and fine-grained data pipelining, Brauer et al. [BLM16] were able to decrease the processing latency of a typical signal processing chain in more than 50%. Vaas et al. [VRH$^+$16] investigated if smart control units used for frequency conversion could benefit from highly parallel hardware accelerators, namely the Epiphany co-processor, and achieved speedups of 1.78 with a limited increase (9%) on power consumption.

2.4 The Raspberry Pi family of SBCs

Raspberry Pi is a very popular line of single-board computers, with all the core hardware components (CPU, RAM, GPU and IO controllers) integrated into a single small card, and supporting connection of several external devices (monitor, mouse and keyboard, and others through a General Purpose Input Output (GPIO) connector). These SBCs were created with the goal of promoting a simple and easy platform to learn programming and become familiar with computer technology [Joh12]. However, an increasing number of companies are taking advantage of Raspberry Pi technology and uses these boards as part of their end products.

The first version (Raspberry Pi 1) was available to the public in 2013, having a 700 MHz single core ARM CPU and 512MB of RAM. In February 2015, the Raspberry Pi 2 was launched, with a quad-core 900 MHz ARM CPU and 1 GByte of RAM. This was followed by the Raspberry Pi Zero in November 2015, with a 1 GHz single-core ARM CPU and 512 MB of RAM, the goal being to promote a very cheap computer. The last significant evolution on the Raspberry Pi line is represented by the Raspberry Pi 3, launched in February 2016, with a quad-core 1.2 GHz ARM CPU and still 1 GByte of RAM. These four models, illustrated in Figure 2.2, are in fact the ones used in this work.

Raspberry Pi can run many free and open-source Linux or FreeBSD-based operating systems. The most popular is Raspbian, a Debian-based Linux operating system, with



(a) Raspberry Pi 1

(b) Raspberry Pi 2

(c) Raspberry Pi 3

(d) Raspberry Pi 0

Figure 2.2: Main Raspberry Pi models

over 35,000 pre-compiled packages available [Fou17b]. Raspbian was used in this work, with the same configuration across all four Raspberry Pi boards.

There are many case studies that cover object tracking assisted by Raspberry Pi SBCs, like [MPG15, SS16], to cite just a few. This, and the popularity of these SBCs, make them an obvious choice for comparison with alternative platforms, like the Parallella board.

2.5 RGB-D Sensor

RGB-D sensors were introduced in 2010 by Microsoft's Kinect device, a peripheral originally developed for the XBOX 360 gaming console, but that may also be connected to a computer using a simple USB adapter and an external power supply. The RGB-D sensors include an RGB sensor that is able to capture frames with three colour channels (Red, Green and Blue), and a depth sensor that can get an image frame that represents objects distance to the camera. Besides the RGB and depth sensors, Kinect also has a microphone array.

In 2013 a second version of the Kinect device was launched, along with the new XBOX One console. However, this work still explores the original version, once it is enough to accomplish the work goals. The Kinect device used is shown in Figure 2.3, and its main specifications are provided in Table 2.3.

Figure 2.3: Microsoft Kinect

Frames per second	30
Available IR distance	0.5m - 4.5m
Image size	640 x 480
Dimensions	24.9cm x 6.6cm x 6.7cm
Weight	1.4kg
Horizontal viewing field	57°
Vertical viewing field	43°

Table 2.3: Microsoft Kinect specifications (version 1).

2.5.1 libfreenect

`libfreenect` is an open source library that enables the Kinect device to be used with Windows, Linux, and Mac systems. This library is available for C, C++, Java, and Python languages. It is maintained by the OpenKinect community, that consists of over 2000 members. In this work, `libfreenect` was used with its C/C++ bindings.

For the C language, `libfreenect` provide a "synchronous" interface where the program executes the function `int freenect_process_events (freenect_context *ctx)` by which it blocks and returns data through callback functions previously configured. This behaviour can be implemented by using a thread to handle the callbacks and a buffer to provide an interface for the client.

`libfreenect` for C++ implements a thread, created in the device class constructor, and that dies when the object is destroyed. The programmer needs to define the callbacks methods in its device class implementation. It should be pointed out that in the C++ class is impossible to control the image buffers; thus, one needs to implement a special separate buffer and to copy the image there during the execution of the callback function. On the other hand, with `libfreenect` for C, is possible to directly set, access and manipulate the image buffers, facilitating further processing and improving performance.

2.6 Profiling Tools

2.6.1 Valgrind

Valgrind is a free and open-source software that helps to detect errors in programs due to incorrect dynamic use of memory, such as memory leaks, incorrect allocation and deallocation, and access to invalid areas. It uses a virtual machine to simulate the memory access of the program under test, eliminating the need to use other auxiliary libraries or drastic code changes. Made for C or C++ coded programs, the virtual machine allowsf possible to use Valgrind with programs that have been encoded in other languages, such as Java. By using other tools that come with Valgrind, it is possible to optimize the use of the processor cache, locate regions of memory accessed concurrently, and obtain memory usage statistics, as well as measure the execution time of parts of a program [Val17b].

The code of the programs that are being executed by Valgrind do not execute directly on the processor of the machine, being before translated to another intermediate representation, named *ucode*, where that code is executed by Valgrind. This justifies the great loss of performance when running programs with Valgrind, while allowing full monitoring of the running program of libraries without connecting additional libraries.

Throughout the development of this work, Valgrind was regularly used in order to validate the robustness of the code developed, detecting possible bad memory management and threading bugs. Initially, Valgrind was also used to measure execution times, using the Callgrind tool. Callgrind is a built-in Valgrind tool, capable of measuring the execution time and generating function call history (and call graphs) with information about an application's runtime [Val17a]. To run the application through `callgrind` one executes the `valgrind --tool=callgrind <prog>` command, where `<prog>` is the application. The result of this is a call graph file, with information about the functions that spend more time during the execution. This call graph can be visualized using the KCachegrind program [Wei17]. In this work, `callgrind` was only used in the initial development stages, and was quickly abandoned in favour of `gprof`, once this tool has a much less noticeable overhead when used in the Parallella environment.

2.6.2 Gprof

Gprof is a tool of the GNU Binutils binary tool-kit [Fou14] that allows the analysis of binary programs by collecting informations on the most requested functions, including number of calls, and execution time (absolute and relative) [IBM17].

To collect runtime data, Gprof needs the executables to have been generated with profiling support; when using the GNU C++/C compilers, this is ensured by the `-pg` compilation option; this option will instruct the compiler to add debugging flags and some extra code in the executable that produces the profiling information. After the execution of the program (if it does not end with errors), it is generated the file named `gmon.out` with information about the execution; this file is interpreted by the Gprof tool to generate a text file with a table that contains the profiling data in an friendly format, and also data that may be used to generate a call graph.

2.7 Robot Operating System

The Robot Operating System (ROS) is a framework designed to make it easier to write robotics software. It is a set of tools, libraries, and conventions that simplify the creation of complex and distributed robot systems. It was created because software development for robots is "hard" and from the robot's perspective, problems that seem trivial to humans often vary wildly between instances of tasks and environments [Fou17a].

This work has put considerable effort trying to port to and run ROS in the Parallella Linux environment, with the aim of simplifying and automating, as much as possible, the capture, pre-processing and pos-processing of frames from the Kinect sensor. Unfortunately, such effort was not 100% effective, the conclusion being that there's seems to exist a fundamental incompatibility between the Parallella runtime environment and ROS (most probably at the USB subsystem). This failure lead the work towards a lower-level custom approach, directly exploring `libfreenect` facilities.

Chapter 3

General Structure and Initial Version

This chapter presents the general structure of the object tracking application developed in this work, along with a first functional version and results of some preliminary tests. This made possible the identification of some bottlenecks, and opportunities for improvements that are necessary for execution under the constraints of the Parallella platform.

3.1 Main Components and Stages

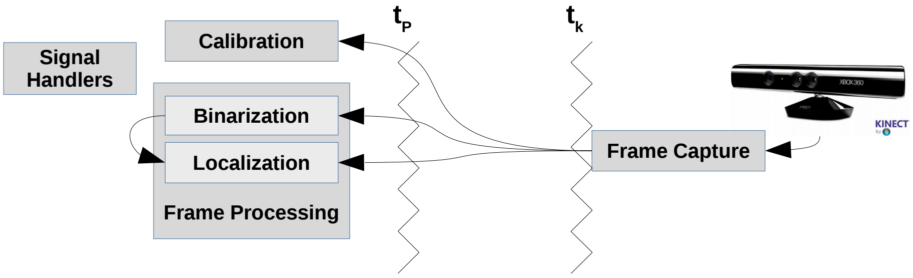


Figure 3.1: General Structure of the Tracking Application.

The general structure of the tracking application developed in this work is represented in Figure 3.1. It includes several components, namely: i) Frame Capture, that captures frames from the Kinect sensor; ii) Calibration, active only at the beginning of execution; iii) Frame Processing, that makes each frame go through several transformations and processing stages (Binarization and Localization) in order achieve object tracking (it also

also allows for the visualization of frames in all processing stages using OpenCV); iv) Signal Handlers, used to control application execution and to trigger feedback from it.

3.1.1 Concurrent Processing with Pthreads

The application components are bound to two threads: t_K, that loops through the Frame Capture code; t_P, that performs the initial Calibration and then loops through the various stages of Frame Processing. The main reason to split the application in two distinct threads is to avoid overloading frame capture with further frame processing, which could delay capture of subsequent frames and generate frame loss; thus, by relieving the frame capture thread from frame processing tasks, the capture of frames may be performed at the maximum rate supported by the Kinect sensor; this was confirmed by preliminary tests with the Parallella board: the frame rate of a single thread, running at the host-side, and performing frame processing in addition to frame capture would be under 30fps; thus, the use of more than one thread becomes imperative if no frame loss is admissible.

The t_K and t_P threads are based on the POSIX Threads (PThreads) standard [But97], for portability and performance reasons. In addition to the use of the PThreads model as the basic foundation for the developed application, other processing strategies were also intermixed (namely in the different stages of the frame processing), to improve performance. The hybrid approaches pursued in this work are addressed in the next chapter.

3.1.2 Frame Capture

The t_K thread captures frames from the Kinect sensor using libfreenect [Ope17c] facilities. This is achieved using two callback functions: one for RGB frames, another for Depth frames. These callback functions work with a pair of MUTEX locks and a condition variable to control access to the shared data structures where t_K writes Kinect frames and from where t_P collects them for further processing. Figure 3.2 provides a simplified representation of this frame transfer mechanism.

When a new frame is received from Kinect, the t_K thread tries to lock a specific mutex

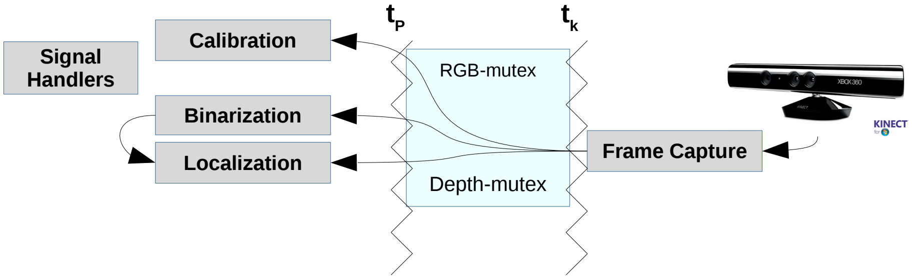


Figure 3.2: Frame transfer from t_K to t_P (general view).

(RGB-mutex, or Depth-mutex) using a non-blocking locking. If the locking succeeds, t_K performs the operation that provides the frame for t_P (a full copy of the frame in the initial version of the application, or a simple pointer switch coupled with a pair of buffers in the final version) and signalizes t_P that a new frame is available; otherwise, if looking fails, it means that the t_P thread has previously acquired the lock and is still accessing the critical memory region; in this case, the t_K thread proceeds to capture a new frame discarding the current frame.

The t_P thread performs a blocking lock to get an available frame. After succeeding, checks if the available frame is new; if it is a new frame it continues to do its processing; otherwise, t_P sleeps waiting to be notified from t_K thread that a new frame is available.

3.1.3 Calibration

The purpose of Calibration is to capture the RGB colours of the object to be tracked in the RGB frames. This is done once, by the t_P thread. The calibration uses a OpenCV-managed window, as shown in Figure 3.3, where the purple rectangle (drawn by the user) delimits the area containing the relevant colours; all the colours obtained are stored in a structure to be used in the Binarization stage.

A common characteristic shared by the various approaches tested for RGB Calibration was to ignore some least significant bits of each byte of an original object RGB pixel. The reason behind this is twofold: i) captured RGB frames usually show some colour noise to the extent that close colours may be considered equal; ii) having to store fewer bits per channel may translate in storage savings (which may be relevant in memory constrained

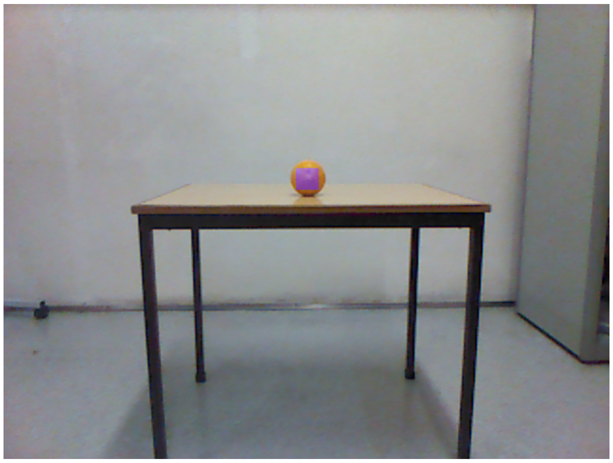


Figure 3.3: Initial RGB Calibration.

scenarios) and even faster searches (once there are less possible colour variants).

3.1.4 Frame Processing

As Figure 3.1 shows, t_P processes frames through two main different stages: i) (RGB) Binarization, and ii) (RGB+Depth) Localization. These stages are discussed next.

RGB Frame Binarization

The goal of this stage is to produce a new version of the original RGB frame, with only two colours: white for pixels belonging to the object tracked, and black for the remaining pixels – see Figure 3.4. Without any compression of the original frame, the size of this "binary" version of the frame will be 1/3 of the size of the original frame, once each pixel will be represented by a single byte with 255 (white) or 0 (black) values.

This is achieved by following a simple procedure: for each pixel of the original RGB frame, it is performed a search on the calibration structure, to check if the pixel colour is one of the detected colours during the calibration stage.

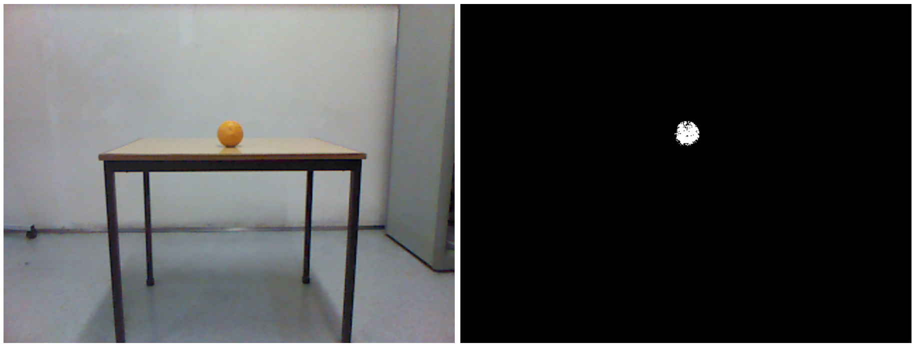


Figure 3.4: Binarization Process (before and after).

Object Localization

This stage takes the <X,Y> coordinates of all object pixels (white pixels) in the binary RGB frame (produced in the previous stage), and computes their average. The outcome of this stage is a single pair of coordinates, $< \overline{X},\overline{Y} >$.

The resulting coordinates, <X',Y'>, are then used to recover a Z' coordinate in the Depth frame as follows: to minimize the noise influence, it is calculated the average Z' of all Depth values of a tile of 3×3 centered on the <X',Y'> position in the Depth frame. This completes the localization of the object in the frame. Figure 3.5 shows the result of the location stage of the object, where the target is over the object tracked in the image, and its final <X',Y',Z'> position (in the context of the frame coordinates) is shown.

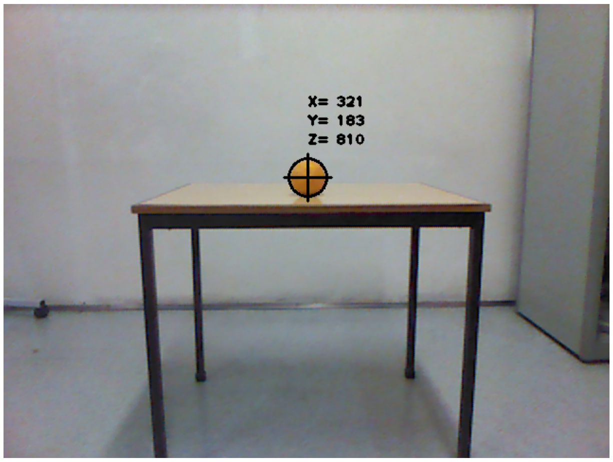


Figure 3.5: Representation of Object Position.

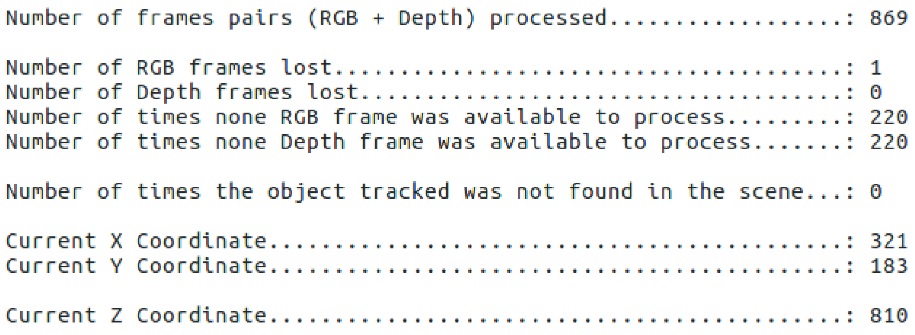
3.1.5 Signal Handlers

The Signal Handlers component includes routines to be executed by the thread t_P upon the arrival of certain POSIX signals (software generated interrupts). The generation of those signals is currently achieved through the `kill` shell command. The signals are sent to the application to trigger certain actions, as described in Table 3.1.

Signal Name	**Signal Number**	**Action**
SIGINT	2	Stop the Application (Clean Exit).
SIGALRM	14	Execute Signal 10 handler, and reprogram the next next Signal 14 arrival.
SIGUSR1	10	Show the current state of the program.
SIGUSR1	30	Show the last image incoming from Kinect (with a target over the object), and its binary version.
SIGUSR1	16	Start Calibration.

Table 3.1: POSIX Signals captured by the application and actions triggered.

As an example, Figure 3.6 shows an application output after receiving a signal 10.



```
Number of frames pairs (RGB + Depth) processed..................: 869

Number of RGB frames lost.......................................: 1
Number of Depth frames lost.....................................: 0
Number of times none RGB frame was available to process.........: 220
Number of times none Depth frame was available to process.......: 220

Number of times the object tracked was not found in the scene...: 0

Current X Coordinate............................................: 321
Current Y Coordinate............................................: 183

Current Z Coordinate............................................: 810
```

Figure 3.6: Output of the Signal 10.

The way in which the application detects and reacts to signals is as follows. Because signals are an asynchronous notification mechanism, it is desirable to execute the smallest possible amount of code, and only certain types of code (like changing the value of an integer variable), in order to avoid breaking the consistency of the application (which may theoretically get interrupted anywhere). Thus, when t_P receives a signal, only a global flag is changed, from zero to the respective signal number, and the action specified in Table 3.1 is postponed to the beginning of a new iteration of the processing loop. Before going after the next frame, the processing loop in t_P checks the flag; if the flag tells a signal has been received, it will execute the proper action and after it will reset the flag.

An exception occurs at the beginning of the application execution, where the Calibration needs to be executed before the image processing loop. In this case the flag assumes the value 16. The application verifies this flag and executes the calibration like if it had received a signal, and after it executes the image processing loop.

During signal handling, the t_K thread is not affected. This thread continues the capture of frames, but these end up getting discarded if t_P gets delayed.

3.2 Initial PThreads Version

This section documents the preliminary efforts towards a functional (prototype-level) object tracker, based on Pthreads. These efforts already include some preliminary evaluation tests that pave the way to several optimizations added in the final Pthreads version.

3.2.1 Frame Capture

Initially, frame capture was performed using facilities offered by `libfreenect` for C++. This library provides the `Freenect::Device` class that needs to implement two callback virtual methods: one for RGB frames (`void VideoCallback(void* _rgb, uint32_t timestamp)`), and the other for Depth frames (`void DepthCallback(void* _depth, uint32_t timestamp)`). These callbacks methods are invoked when a new RGB/Depth frame is made available by the Kinect sensor, receiving a reference (pointer) to the frame as input parameter. Before returning, the callbacks copy (by value) the frames to separate buffers. These are the RGB-buf and Depth-buf buffers represented in Figure 3.7.

In turn, the processing thread, t_P, will also copy the content of intermediate buffers RGB-buf and Depth-buf to its own buffers. The consistency of these operations is ensured by the mutual exclusion mechanisms put in place, already presented in section 3.1.2.

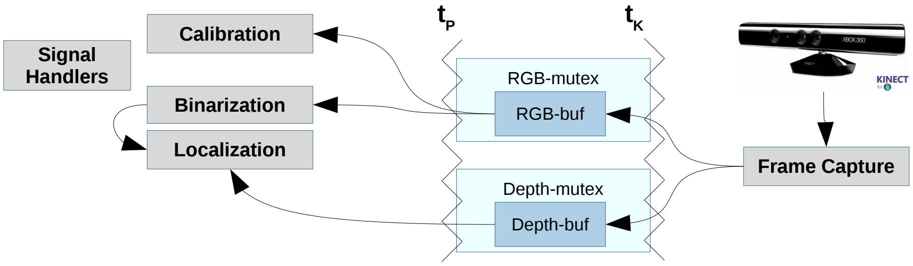


Figure 3.7: Frame transfer from t_K to t_P (initial version: by copy)

3.2.2 Calibration

Calibration implementation started by exploring two different data structures, with the goal of finding a way to store calibration data with minimal memory space and still good access performance. These preliminary calibration data structures are discussed next.

Dynamic RGB Vector

In this approach the specific RGB colours of the tracked object are inserted, one-by-one, into a dynamically allocated vector, based on the `std::vector<colorID>` type, where `colorID` is a 32 bit unsigned integer. Colours are sorted by the colorID 32 bit value and the class used provides access to a specific colorID using internally a binary search.

The internal structure of the colorID data type is shown in Figure 3.8. It includes 1 byte per each RGB channel, and a 1 byte padding. The padding ensures the colorID type consumes 32 bits, thus being memory aligned, which improves access speed; moreover, it makes searching for colours easier (once it is enough to compare full 32 bit integers), and faster (because the CPU can use only one instruction to make comparisons).

In line with what was stated in section 3.1.3, only 6 bits per RGB channel are considered; the others are ignored (zeroed, in fact); this is conveyed, in Figure 3.8 by the special symbol 'X'. However, in this approach those ignored bits still consume storage space.

Static RGB Cube

This variant uses a 3-dimensional static array, also known as RGB cube in this context. (a geometric representation of the RGB cube is shown in Figure 3.9). The presence/absence

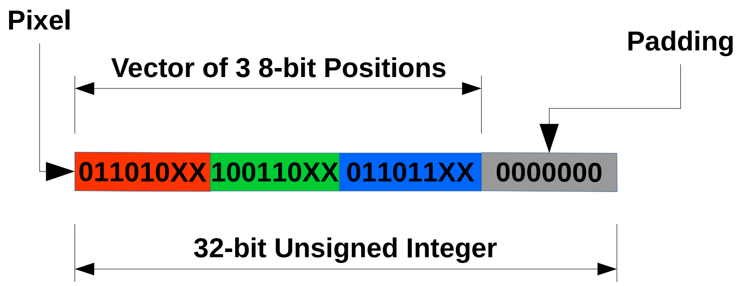


Figure 3.8: Internal structure of the colorID data type.

of a specific RGB colour in the object is represented as a True (1) / False (0) boolean value in the specific RGB cube cell of that colour.

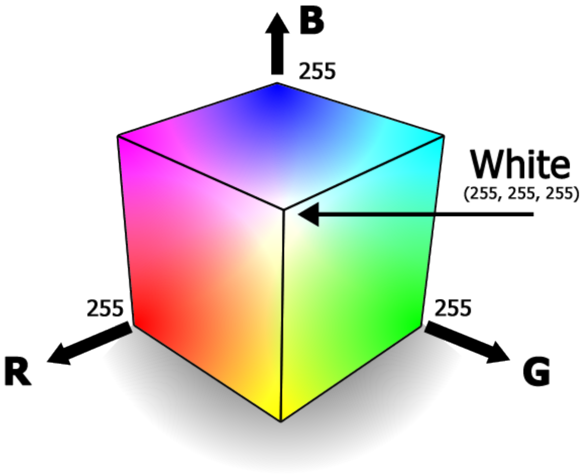


Figure 3.9: RGB colour space

Considering 8 bits per each RGB channel, the RGB cube would have $2^8 \times 2^8 \times 2^8 = 2^{24} = 16M$ elements of 1 boolean byte each, thus consuming 16 MBytes of storage. However, each channel byte of an RGB pixel of the tracked object is still stripped of the least 2 significant bits when defining the RGB cube coordinates that will have the value True. This means that the RGB cube can be much smaller, with only $2^6 \times 2^6 \times 2^6 = 2^{18} = 256K$ elements, thus consuming only 256 KBytes. Nevertheless, the Static RGB Cube still consumes much storage space than the Dynamic RGB Vector. The big advantage of the Static RGB Cube approach is that it supports direct access to the intended element, which is faster than the binary search used in the Dynamic RGB Vector approach.

3.2.3 RGB Frame Binarization

The Binarization stage relies on access to the data structures produced during Calibration to perform the binarization of the RGB frames incoming from Kinect. Thus, with two different Calibration data structures, there are at least two different approaches to Binarization: with the Dynamic RGB Vector, or with the Static RGB Cube.

Those data structures may, however, benefit from extra information gathered during Calibration, that may be used to accelerate the Binarization process. This extra information consists on the lowest and highest Red, Blue and Green values observed for the tracked object. The idea is then to immediately discard, as not belonging to the tracked object, a frame RGB pixel if at least one of its Red, Blue and Green values are out of the ranges observed during Calibration; otherwise, the Calibration data structures will still have to be searched, to confirm if the RGB pixel belongs to the object; this confirmation is necessary because it is not enough for the Red, Green and Blue values to fall in the expected ranges: typically, only some of the combinations of those in-range values are valid. The Binarization variants that take advantage of this simple segmentation method carry the suffix ”with Threshold” on their name.

A sum up of the four initial approaches to RGB Frame Binarization is presented next.

Dynamic RGB-vector

To check if the RGB colour of a RGB frame pixel belongs to the tracked object, convert that RGB colour to the ColorID format and trigger a search for it in the Dynamic RGB-vector, taking advantage of the binary search provided by the C++ STL [Int17].

Dynamic RGB-vector with Threshold

Like the previous approach, but preceded by an extra test: only conduct the search in the Dynamic RGB-vector if the Red, Green and Blue values of the RGB pixel are all within the valid ranges, for each channel, identified during Calibration.

Static RGB-cube

To check if the RGB colour of a RGB frame pixel belongs to the tracked object, convert each RGB channel from 8 bits to 6 bits and then use the three 6 bits values as coordinates for direct access to the Static RGB-cube.

Static RGB-cube with Threshold

Like the previous approach, but preceded by an extra test: only conduct the search in the Dynamic RGB-vector if the Red, Green and Blue values of the RGB pixel are all within the valid ranges, for each channel, identified during Calibration.

This technique virtually searches a "sub-cube or sub-parallelepiped" inside the RGB-cube, that encompass all colours within the ranges identified during Calibration.

3.2.4 Preliminary Evaluation

This section presents the results of an evaluation of the initial application version.

The evaluation was conducted on a Parallella-16 Desktop Computer board, taking advantage only of its dual core ARM CPU, once the initial application version was only based on Pthreads and thus not yet capable of exploring the Epiphany co-processor.

In all tests the Kinect sensor was pointed to the same scenario, and the object tracked was the same as well as the luminosity conditions (once the object was static). In order to ensure fairness in the evaluation of the four variants, the number of colours picked during Calibration was always the same[1]: 2048 RGB colours; this number was chosen having in mind memory constraints for a future Epiphany version (the space necessary to store a vector of 2048 RGB colours – with 4 bytes per colour, due to memory alignment requisites – would be 8 KBytes, that is, the capacity of an eCore local memory bank).

[1] Sometimes requiring the capture of more than one frame during Calibration.

The executables were generated with GNU compilers (`g++` for C++ code, and `gcc` for C code), using the `-Os` option (enables all `-O2` optimizations that do not increase code size, which is adequate for embedded systems, that typically have limited memory). Execution times were measured using `gprof` [FSB98].

Frame Capture Time

Initially, the application was fully based on C++, and only `g++` was used to generate the executable(s); however, the frame capture times of this version were measured to be very high, being around 63ms, well above the time necessary (33 ms) to ensure a frame capture rate of 30 fps; the culprit was then found to be the time spent by the RGB and Depth callbacks to copy the frames to the intermediate buffers shared with the processing thread (RGB-buf and Depth-buf in Figure 3.7). After some unsuccessful attempts to solve the problem, the drastic option was taken to rewrite the frame capture module exclusively in C, and compile it with `gcc`. The net result was the decrease of the frame capture time from $\approx$ 63 ms to $\approx$ 43 ms. Despite the improvement, this late value is still above the desired time (33 ms), and extra optimization efforts were necessary (see next chapter).

Frame Processing Time

To evaluate the frame processing time, four variants of the initial application were tested, each using one of the four approaches (presented in section 3.2.3) for the RGB Frame Binarization stage. The results, measured in milliseconds, are shown in Figure 3.10, as averages, considering 5 runs of each executable and a limited sample of 2000 frame pairs (RGB+Depth) per run. The results also include the time spent by the the t_P thread to copy data from the intermediate buffers shared with the capture thread (RGB-buf and Depth-buf in Figure 3.7), to its own buffers; together, these times are less than 10 ms, and so the bulk of the time is really spent in the Binarization and Localization stages.

The results clearly show that using a Dynamic RGB-vector, even with the Threshold optimization, is always outperformed by the Static RGB-cube approach.

The underwhelming performance of the Dynamic RGB-vector approaches is due to the

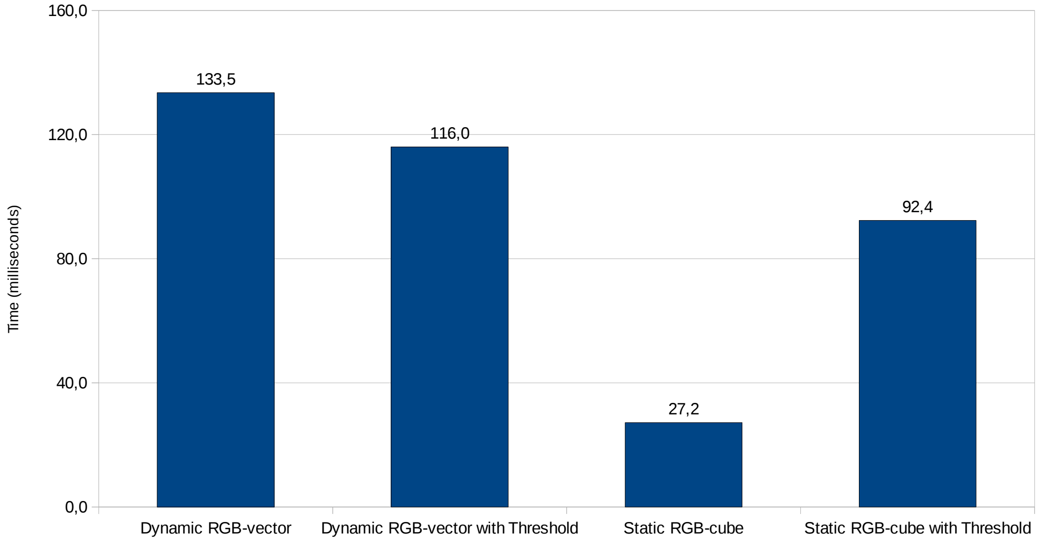


Figure 3.10: Frame processing time in the initial version (ms).

time spent in binary searches: with 2048 RGB colours and no Threshold optimization, the maximum number of comparisons is 11, and this is the effective number of comparisons most of the time, because the object tracked is typically small in comparison to the scene; on the other hand, with the Threshold optimization, there will always be 6 initial comparisons, that eliminate the vast majority of candidates (and so only a small number will pass the test, and imply a binary search); however, the impact of those 6 comparisons is still very high, to the extent that the Dynamic RGB-vector with Threshold approach has a speed-up of only 133,5/116 = 0,1509 over the Dynamic RGB-vector approach .

The cost of the 6 initial comparisons is also very high in the Static RGB-cube with Threshold approach, such that it pays off to always access the RGB cube, as done by the Static RGB-cube approach. The speed-up relative to the Static RGB-cube with Threshold approach is, in this case, 92,4/27,2=3,3971. As such, the Static RGB-cube approach was chosen as the base approach for the final application version, with adaptations necessary to cope with the memory constraints of the Epiphany co-processor (see next chapter).

Finally, it should be said that the average frame processing time of 27,2 ms ensured by the Static RGB-Cube approach is already bellow the limit of 33,(3) ms for a single frame, as imposed by the Kinect nominal sampling rate of 30 fps. As threads t_P and t_K

are able to run simultaneously (if at least two cores are available), this means that, as soon a t_K has captured a frame, t_P will immediately grab it and will become ready to grab the next frame slightly before it arrives. It also means that there's a delay of $\approx$ 27,2 ms from the moment that a frame is made available (by t_K) to the moment that its processing ends (and its possible effects manifest) or, equivalently, "processing is always one frame behind". Shrinking this gap makes the application to follow more closely what happening in the real scene, and may even free enough time for other features, like visualization. In the next chapter, several optimizations are explored, including parallel processing techniques, in order to achieve even lower frame processing times.

Chapter 4

Optimized and Hybrid Versions

This chapter starts by presenting a set of optimizations that decrease the storage requisites and enhance the performance of the PThreads version of the objet tracking application, as shown by a second round of tests. It then introduces the parallel programming models used to try to further improve the performance of PThreads version: OpenMP and the Epihany eSDK. For each hybrid approach, performance results are presented. The chapter ends with a final discussion on the performance achieved by the different approaches used. A comparison is also provided with several models of the Raspberry Pi platform.

4.1 Optimized PThreads Version

As shown in section 3.2.4, the way in which frames are transfered, from the t_K thread to the t_P thread, in the initial version of the application, imposes a significant performance penalty, even preventing frame capture to be conducted at the Kinect sensor nominal rate (33 fps). On the other hand, the size of several important data structures, namely the captured frames and the RGB-cube produced/used during Calibration and Binarization, are still inadequate to the memory constraints of the Epiphany grid of eCores.

The problems above identified are solved in the final iteration of the tracking application. The general structure of that optimized version is presented in Figure 4.1: it is based on a new strategy for the transfer of frames between t_K and t_P, and there's also a

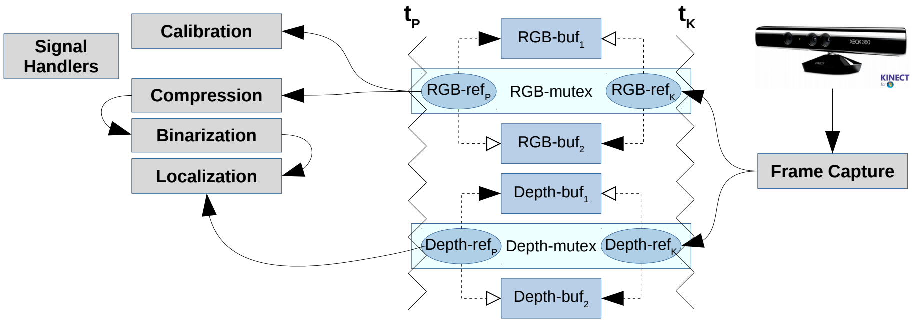


Figure 4.1: General Structure of the Optimized Version.

new Compression processing stage. These enhancements are discussed along this section.

4.1.1 Frame Capture

The change of the code base of the frame capture module from C++ to C allowed not only to decrease the frame transfer times (although still not enough), but also opened the opportunity to explore the facilities of `libfreenect` for C in order to directly control the pointers that Kinect uses to reference the buffers where captured frames are deposited.

The new approach still uses separate callback functions for RGB frames and Depth frames. Each callback uses a pair of buffers (<RGB-buf$_1$, RGB-buf$_2$>, and <Depth-buf$_1$, Depth-buf$_2$>) and each pair of buffers is referenced by a pair of global pointers (<RGB-ref$_P$, RGB-ref$_K$>, and <Depth-ref$_P$, Depth-ref$_K$>), visible in both t_K and t_P. Pointers RGB-ref$_K$ and Depth-ref$_K$ reference the buffers that t_K will use to receive the next RGB and Depth frames. Pointers RGB-ref$_P$ and Depth-ref$_P$ reference the buffers that hold the previous RGB and Depth frames to be processed by t_P. The pointers switch the buffers they point to, for each new frame captured. This way, there is no need for memory copies that would delay the processing of the current frame and the capture of the next. The same pair of locks (RGB-mutex and Depth-mutex) are still used, to protect access to the pointers (and their pointed buffers), following the same logic described in section 3.1.2.

4.1.2 Calibration

The preliminary evaluation discussed in section 3.2.4 showed the Static RGB-cube approach to be the most performant. On the other hand, even with only 6 bits per RGB channel, the RGB-cube still consumes 256 KBytes, making it impossible to fit one copy in the limited 32 KBytes local memory of each eCore (other options, like placing the RGB-cube in shared memory, or even to scatter it among the 16 eCores, would entail worse performance, specially the last alternative). Therefore, extra compression is needed.

First, the size of Red and Blue coordinates is further reduced one bit, so that, in the end, the Red, Green and Blue bytes loose the least significant 3, 2 and 3 bits, respectively; this transformation can be represented by $< R_8, G_8, B_8 > \rightarrow < R_5, G_6, B_5 >$ (this finds ground on the fact that captured RGB frames usually show some color noise, to the extent that close colors may be considered equal). This reduces the overall number of $< R, G, B >$ coordinates to $2^5 \times 2^6 \times 2^5 = 2^{16}$, producing a new RGB calibration data structure, that takes only 64 KBytes (still using 1 boolean byte per RGB colour). This new data structure is 1-dimensional, indexed by a $R_5G_6B_5$ 2-byte coordinate.

Further compression is achieved by replacing each boolean byte by a boolean bit, in the RGB calibration data structure, as it shows Figure 4.2. This final structure, hereafter named **RGB-bitmap**, thus becomes a 1-dimensional vector, with 64 Kbits, now taking only 8 KBytes, thus perfectly fitting in one of the four local memory banks of each eCore.

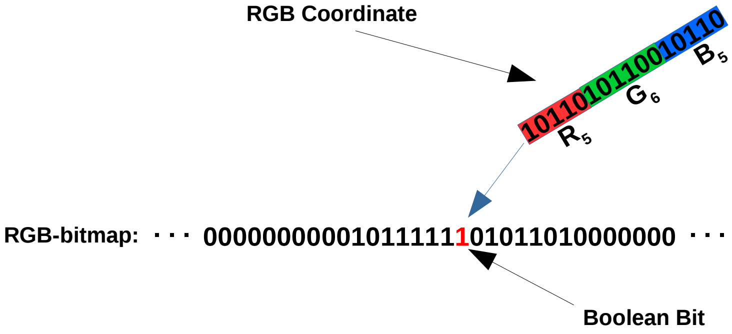


Figure 4.2: RGB-bitmap calibration data structure.

4.1.3 RGB Frame Compression

This new stage reduces the RGB frame size. It aims to allow the RGB image (not the original, but a compressed version) to fit into each eCore's local memory (1/16th of the image, per eCore). Besides reducing frame size, compression also reduces color noise and increases the processing speed of further stages (as they have less RGB pixels to process).

RGB frame compression is achieved as follows: the original 640 × 480 RGB frame is downscaled to 320 × 240 (each dimension is halved), by averaging tiles of 2×2 adjacent pixels, producing a smaller frame with 1/4th of the size of the original (900 → 225 KBytes) as shown in Figure 4.3.

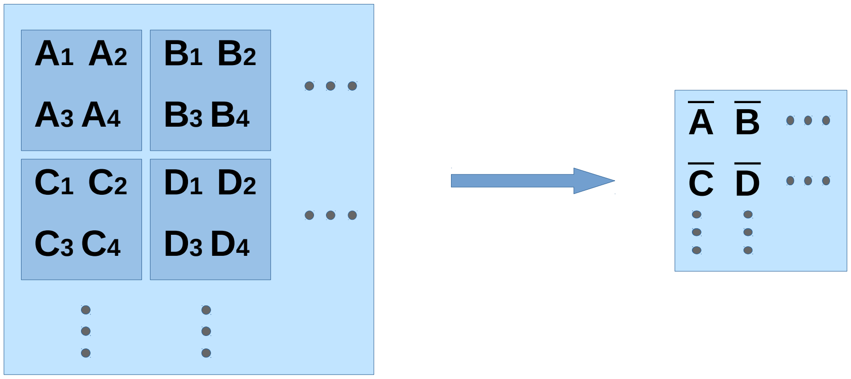


Figure 4.3: First phase of the RGB Frame Compression: Pixel Averaging.

A second phase of compression is then performed over the outcome of the first phase: the RGB colour of each RGB pixel is downsized from 24 bits to 16 bits, using the same technique applied to the RGB coordinates of the RGB-bitmap (see above). This further reduces the frame size in 2/3[1], going down from 225 KBytes to 150 KBytes (9,375 KBytes per Epiphany core). Furthermore, the $R_5G_6B_5$ pixels in the final reduced frame can now be used as RGB coordinates for direct access to the RGB calibration bitmap. The Figure 4.4 shows the outcome of the two phases of the compression, Although presented here separately, the two phases of the RGB Frame Compression stage are performed together, for each pixel, in a single loop, for performance reasons.

[1] For a total reduction of $1/4 \times 2/3 = 1/6$ with relation to the original size of 900 KBytes.

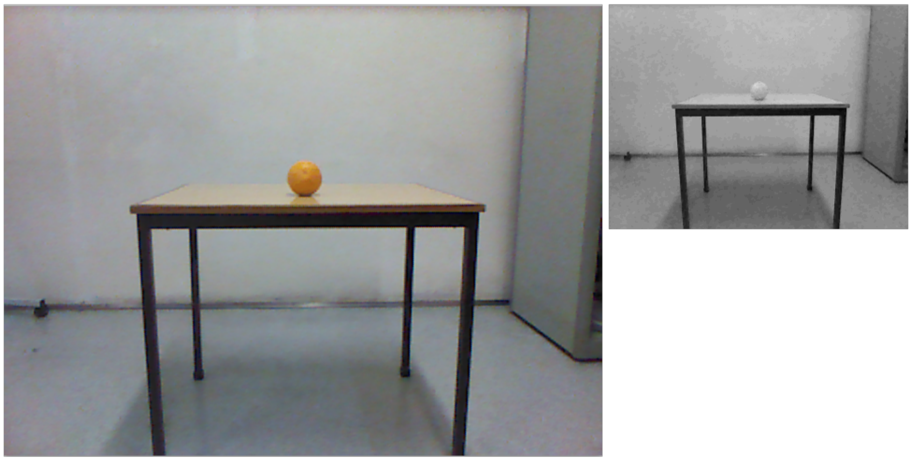


Figure 4.4: Frame Compression Process (before and after).

4.1.4 RGB Frame Binarization

The Binarization stage now needs to take into account the new RGB-bitmap as the optimized Calibration data structure: it goes through each $R_5G_6B_5$ 16 bit colour of each pixel of the RGB frame compressed, and uses that 16 bit value as a 1-dimensional coordinate in the RGB-bitmap, to get the corresponding boolean bit. Because the RGB-bitmap is primarily byte-addressed, it is first necessary to compute the index of the byte that holds the intended bit (see Equation 4.1); then, the value (0 or 1) of the intended bit may be easily extracted from the referenced byte (see Equation 4.2).

$$byteIndex = RGBcolor/8 \tag{4.1}$$

$$bitValue = (128 >> (RGBcolor\%8))\ \&\ RGBbitmap[byteIndex] \tag{4.2}$$

The size of the binary frame will be 1/2 of the RGB compressed frame, thus taking 150 KBytes / 2 = 75 KBytes, or 1/12 of the original uncompressed RGB frame.

Object Localization

This stage follows the description provided in section 3.1.4, with an additional adjustment: the coordinates <X',Y'> are doubled before recovering the Z' coordinate in the Depth frame, at position <2X',2Y'> (<X',Y'> refers to a 320 × 240 binary frame, but the Depth frame was not compressed and thus preserves its original 640 × 480 resolution).

4.1.5 Preliminary Evaluation

To measure with more accuracy the frame processing time, it was decided to replace the usage of `gprof` by instrumenting the code with calls to the POSIX function `gettimeofday`. The processing times presented are still averages of 2000 samples per run, and each test is still executed in 5 runs. This methodology was adopted for the remaining of the work.

The frame processing times of the initial and optimized Pthreads version, measured under the new methodology, are shown in Figure 4.5.

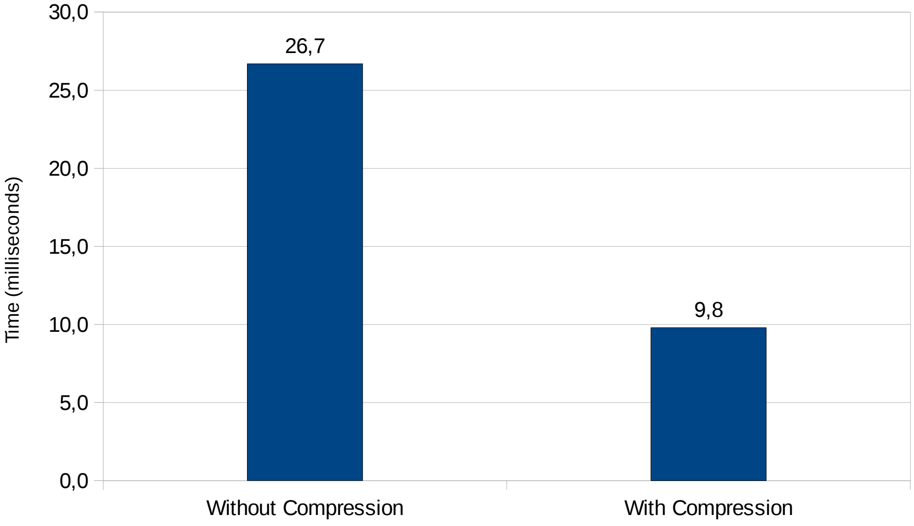


Figure 4.5: Frame processing time (ms): initial vs optimized Pthreads version.

The new evaluation methodology is clearly less intrusive, once the time measured for the initial version is now 26,7 ms, slightly bellow the 27,2 ms previously measured (see Figure 3.10). Regarding the optimized version, the two optimizations applied (a new frame transfer strategy between t_K and t_P, and the use of Compression on the captured frame and the Calibration data structure), were able to provide a respectable speed-up of 26,7/9,8=2,7. And, as important as this performance improvement, the main data structures have now adequate sizes to be stored in the local memory of the Epiphany eCores, thus making possible to finally explore the Epiphany co-processor.

Trying to further improve the performance of the optimized Pthreads version, three hybrid approaches were developed that build on this version: the first explores OpenMP; the second explores the low-level Epiphany Software Development Kit (eSDK); the final one mixes OpenMP and the eSDK. These approaches are described in the next sections.

4.2 Parallel Processing with OpenMP

OpenMP was explored as a straightforward way for automatic (many-)thread parallelization of all frame processing stages. These stages are particularly suitable to parallelization due to independent accesses and processing of the data structures involved.

In the Parallella board, OpenMP may be used to take advantage either of the dual-core ARM CPU or of the Epiphany many core co-processor. This implies the use of two different OpenMP implementations: the ARM CPU was targeted by the native OpenMP facilities of the C/C++ compilers used; the Epiphany 16-core co-processor required the use of OMPi, a separate OpenMP implementation. These two ways used to explore OpenMP are explained next.

4.2.1 Host-only OpenMP

The OpenMP version that targets the ARM CPU was used for automatic parallelization of `for` loops in the Compression, Binarization and Localization (RGB only) stages, by prepending each loop with proper ”pragma” directives. For the Compression and Binarization stages, the directive was `#pragma omp parallel for`, in a single loop in each stage. To compute the average coordinates ($\overline{X}$,$\overline{Y}$) in the RGB Localization, a more specific directive (reduction) was used to sum automatically the results of every thread in a single variable for each result: `#pragma omp parallel for reduction(+:sumX,sumY,numElems)`.

These directives allow to separate data and work automatically among the processing cores available in the platform. The number of threads the application uses is set up by an ambient variable (by default, its value is the number of processing cores available).

It is possible to change the way that these directives work, as controlling the amount of data that each OpenMP thread will process, or turn the operation of separating the data from dynamic (default) to static. Several tests showed that the default way to use the directives is the most effective way.

4.2.2 Epiphany-specific OpenMP

When using OMPi, it is possible to compile only C 99 code; however, the frame processing thread was based on C++ 11. In order to solve this mismatch, the following strategies were applied: i) the frame processing thread code was rewritten in C language; unfortunately, the OMPi compiler seems to have a bug that makes impossible to compile code that uses, at the same time, `pragma` directives and calls to the C-based OpenCV library; ii) to solve the previous problem, the code with `pragma` directives was moved to a different file and compiled separately; however, the OMPi compiler was unable to link that file with files having calls to the C-based OpenCV library; iii) as a last attempt, code with pragmas was kept in C and was compiled with the OMPi complier, and for the remaining code its C++ version was used again and compiled with `g++`; the linkage of the object files was made with OMPi; this convoluted strategy solved the compilation problem and produced and executable that runs without crashes.

Another problem found was that vectorization was not fully implemented by the OMPi compiler. In its current development state, OMPi can only send a statically allocated vector (as if it were a simple scalar datum) to the Epiphany co-processor (providing a pointer to a vector, along with its size, does not work). Thus, to send the compressed RGB frame and the RGB-bitmap to the co-processor, it was necessary to declare global static instances for those data structures, in the application module having the `pragma` directives. In the C++ application module, `extern` directives allow to access those static structures through pointers.

Only the Binarization and RGB Localization stages were parallelized, using `pragma` directives for loop parallelization, similar to those used with ARM CPUs. However, these directives were preceded by three additional OMPi specific directives: i) `pragma omp target data map(to:staticRGBframe,staticRGBbitmap) map(from:sumX,sumY,numPixelsObj)`, ii) `pragma omp parallel private(staticRGBbitmap)`, iii) `pragma omp target`. The first two directives ensure the transfer of a copy of the RGB calibration bitmap and 1/16th of the compressed RGB frame to the local (private) memory of the Epiphany cores, and the recovery of the outcome of the RGB Localization. The third directive ensures that subsequent parallel loops execute on the Epiphany device.

Unfortunately, the OMPi compiler has bugs (yet unresolved) that prevent, in the Epiphany, the parallelization of further loops after the first one: subsequent loops execute only in a single of the Epiphany cores. As a tentative to solve this problem, it was tried the use of the `sections` directives. These allow to manually assign specific tasks to each target core. In this context, this would mean to assign the same code, but with specific data, into each eCore, with the objective to force the parallelization. However, the result was the same as the one observed when using `for` loops. After this, no more efforts were done in order to use OMPi to explore the Epiphany under the OpenMP model, and this approach was not considered in the evaluation of section 4.3.3.

4.3 Parallel Processing with the Epiphany SDK

Another processing strategy explored in this work (and its main motivation) was to make use of the low-level Epiphany Software Development Kit (eSDK) for the parallelization of Binarization and Localization (RGB only) on the Epiphany co-processor. This approach allows finer control of the processing resources and data distribution in the Epiphany, than the OMPi based one (that ultimately couldn't be made to work as expected), promising better performance, although at the cost of higher programming complexity.

In this approach (and, more generally, in any one that makes use of the Epiphany device), it becomes crucial to minimize data transfers between the ARM host and the Epiphany device, and to take the most advantage of data copied to / residing in the small local memory of the Epiphany eCores (this kind of constraints is typically found when developing applications for heterogeneous systems). These were the primary reasons for having aggressive compression applied to the RGB calibration structure and to the captured RGB frames. Without compression, frame processing could still be offloaded to the Epiphany co-processor, by splitting each original frame (and the calibration structure) into slices, and processing each one in as much offloading rounds. But that would hardly pay off, due to the numerous data exchanges needed between host and co-processor.

Thus, using the eSDK, the first task right after Calibration finishes in the host, is to copy the RGB-bitmap (8 KBytes) to one local memory bank, at each eCore. This is done once, before starting the main loop, that captures Kinect frames. Then, for each frame acquired and compressed, a horizontal slice of this frame (1/16 of the original frame, taking 9,375 KBytes) is copied to the local memory of an eCore (making full use of an extra memory bank, and consuming 1,375 KBytes of yet another), for further processing.

As soon as each eCore has a local copy of the RGB-bitmap, and also a 1/16 slice of the RGB frame compressed, it can perform the Binarization of this slice, and all eCores do this in parallel. The outcome, in each eCore, is a binary horizontal slice, that needs 75 KBytes / 16 = 4,6875 KBytes of storage; considering that 8 KBytes are reserved for code and heap, 8 KBytes for the RGB-bitmap, 9,375 KBytes for the compressed slice, this leaves 32 KBytes - 25,375 KBytes = 6,625 KBytes for the stack and for the binary slice; however, to avoid corrupting the stack, it was decided to overwrite the compressed slice with the binary slice as binarization progresses.

After completing Binarization, each eCore moves to the Localization stage. This stage is still executed in the grid of eCores, because it is an ”embarrassingly parallel” operation; moreover, moving out each binary slice from each eCore to the host (so it performs Localization), would take considerable time and, even using its full two CPU-cores, the host would not be fast enough to amortize the previous communication delays (this was confirmed during the development of this hybrid version).

During Localization, each eCore e (with $e = 0..15$) scans its binary horizontal slice and returns to the host the sum of the global X and Y coordinates (that is, coordinates in the context of the full binary frame) of each pixel that belongs to the object tracked ($sumX_e$ and $sumY_e$), and also the number of those pixels ($numPixels_e$). The host will reduce these values to average global values ($\overline{X} = \sum\limits_{e=0}^{15} sumX_e / \sum\limits_{e=0}^{15} numPixels_e$, and $\overline{Y} = \sum\limits_{e=0}^{15} sumY_e / \sum\limits_{e=0}^{15} numPixels_e$) and incorporate Depth information to produce the final X, Y, and Z coordinates of the object, as discussed in section 3.1.4.

The way in which each eCore conducts Localization is next detailed. The full binary frame has 320 columns by 240 lines. Thus, each binary slice, at each eCore, has 320 columns by 15 lines (each binary slice is a horizontal slice with 1/16th of the lines of the full binary frame). Each eCore scans it binary slice line-by-line. In each line, as it founds a pixel belonging to the object, it adds the local X and Y coordinates (that is, coordinates in the context of the binary slice) to the $sumX$ and $sumY$ accumulators; the local X coordinate is the same as the global X coordinate, an so it is added to sum_X without any transformation; however, the local and global Y coordinates differ, and the first is converted to the second, using the simple relation given by equation 4.3; in this equation, e is a unique application-level identifier, specific to each eCore, in the range 0 to 15 (this identifier is derived from the ID of the eCore in the eMesh Network-on-Chip – see section 2.3.1), and $Slice_{height} = 240 \ / \ 16 = 15$.

$$Y_{global} = e \times Slice_{height} + Y_{local} \tag{4.3}$$

4.3.1 Data Exchange via Shared Memory

Data exchanges between the host and the Epiphany device can be done in two ways: using the shared memory as intermediate data buffer, or with direct access to local memory. Both strategies were explored and evaluated. This section discusses the first strategy, and the next section describes the second. A performance comparison is given in section 4.3.3.

Using the 32 MBytes shared memory between the ARM host and the Epiphany coprocessor, for data exchange, is conducted via one or more buffers created on that memory zone (mapped in the host main memory). The size of those buffers must be multiple of 8 (padding may be required), once eCores local memory is 8 byte-aligned and host-device DMA transfers also require this alignment. In the application developed in this work, this data exchange strategy used only one buffer divided in two regions: i) a *control region*, for host-device synchronization, and ii) a *data region*, for host-device data exchanges.

In the control region, one byte is used by the host to encode a specific task to be performed by the device; eCores inspect this byte and will act accordingly; when the task is completed, each eCore will change a specific byte flag in the control region: the host actively monitors these flags (an asynchronous notification mechanism is not available) to know when a task submitted to the Epiphany device has completed. The data region is wide enough to accommodate the different structures that host and device exchange: RGB bitmap, RGB frames compressed, and (partial) RGB localization results.

The tasks are device-side functions, that i) grab from the host a copy of the RGB calibration bitmap, ii) grab from the host a specific slice (1/16th) of a RGB frame compressed, and perform binarization on it, and iii) inspect a binary RGB slice on local memory, and return partial localization data (the number of object points detected, and the sum of its X and Y coordinates), to the host.

Host-device data transfers are DMA-based, from the perspective of the device. When copying the RGB calibration bitmap, a simple blocking DMA operation is used; this means that each eCore is blocked, waiting for its own DMA controller to finish the transfer. However, when loading a compressed RGB frame slice, it is more efficient to use a non-blocking DMA operation: this allows the DMA controller to grab one line, while the eCore is busy with the binarization of the line previously transferred, effectively overlapping communication and computation Finally, when each eCore returns RGB localization data to the host, it does it using, again, blocking DMA (although, in this last stage, the use of DMA is only marginally better than the eCore doing the transfer itself).

From the perspective of the DMA mechanism, the RGB frame compressed stored in the shared memory is a 1-dimensional vector of contiguous slices, with $Slice_{width} \times Slice_{height}$ pixels per slice, where $Slice_{width} = 320$ and $Slice_{height} = 240/16$; each eCore will get the lines of its slice, one-by-one; the beginning position (to the byte) of each line l (with $l = 0..Slice_{height} - 1$) in the 1-dimensional vector is given by equation 4.4, as follows,

$$Line_{address} = Slice_{address} + l \times Slice_{width} \times Pixel_{size} \tag{4.4}$$

where $Pixel_{size} = 2$ bytes (16 bits) and $Slice_{address}$ is given by equation 4.5, as follows

$$Slice_{address} = e \times (Slice_{width} \times Slice_{height}) \times Pixel_{size} \tag{4.5}$$

This strategy allow the DMA controllers of all eCores to read the shared memory in parallel, once access is performed to mutually exclusive regions (slices). Also, the same strategy was used, but in the reverse direction (and with $Pixel_{size} = 1byte$), for the debugging of Binarization, writing its outcome in the shared memory.

Due to restrictions on how the four 8 KBytes banks of local memory may be used, and also to ensure maximum performance on local memory accesses, carefully planning is needed to chose the best local storage approach. In an eCore local memory, the first bank (bank 0) stores the interrupt vector plus code, banks 1 and 2 are free, and the stack is assigned to the last bank (bank 3), that fills top-bottom. With this in mind, it was decided to assign the RGB bitmap to bank 1, and the compressed RGB frame slice to banks 2 and 3, alternated (odd lines to bank 2, and even lines to bank 3, taking about 50% of each banks capacity, once 9,375 KBytes / 2 = 4,6875 KBytes). This alternation allows the eCore and its DMA controller to access local memory at the same time, without conflicts: the eCore processes (read access) a slice line in a bank, while the DMA controller writes a new line to another bank. This organization can be seen in Figure 4.6.

Multi Line Access Evaluation

To gather insight on the performance impact of different number of DMA requests per slice (and thus different amounts of data transferred per request), an evaluation was made

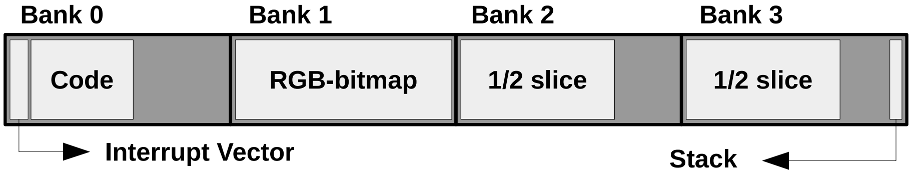


Figure 4.6: eCore local memory map when using DMA.

focused on the Binarization, thus including the transfer of slice lines from shared memory to local memory via DMA, and the simultaneous processing of previously transfered lines.

Figure 4.7 shows the Binarization time per frame (that is, considering all eCores working at the same time, in their own slice lines of the same frame), measured for 1, 3, 5, 15, and 20 lines transferred per DMA operation. The different numbers of lines considered were chosen to cover two different situations: i) each eCore still ends processing, in the end, 15 lines, so that all eCores are busy (1, 3, 5 and 15 lines, per DMA transfer); ii) only 12 eCores are busy (each one processes 20 lines, transferred in a single DMA operation), and 4 eCores are idle; it should be noted that it was not possible to test this scenario with more than 20 lines due to the limited local memory available in the eCores.

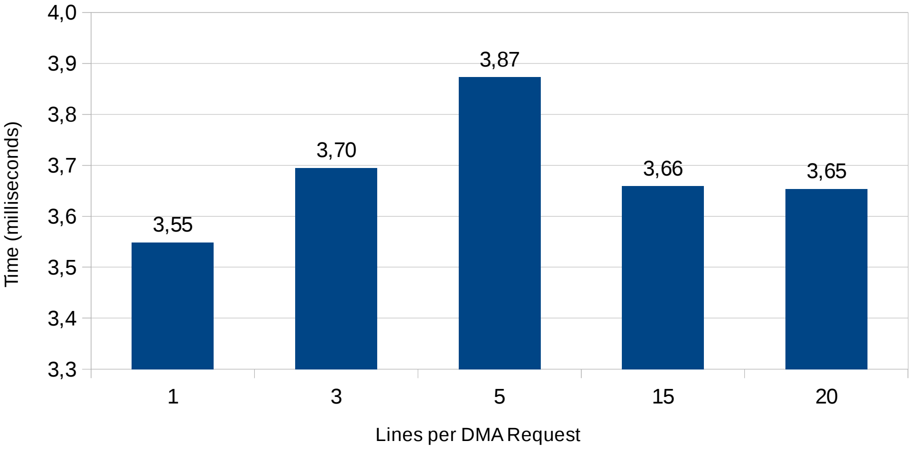


Figure 4.7: Number of lines per DMA transfer: impact on Binarization times.

The results show that transferring one slice line per DMA request is the most efficient approach. When an eCore requests a DMA operation (through its own DMA controller) with only one line, it will succeed and release the bus very quickly, allowing another eCore to succeed; there is contention in the access to the external shared memory, but each eCore has to wait a short amount of time for its next transfer slot; also, transferred data (a single slice line) is quickly processed. With 3 or 5 lines transferred per DMA request, each transfer will take more time, and so each eCore will have to wait more time for its next transfer slot; and, despite the fact that fewer transfers are necessary, the overall times increase. Notably, when there is only one DMA transfer per eCore (15 or 20 lines per transfer), the times improve, when compared to the transfer of 3 or 5 lines; but the fact that times are similar with 15 lines (16 eCores busy) and 20 lines (12 eCores busy) is an indication that the processing load of the Binarization in the image processing scenario of this work is not high enough to justify the full use of the Epiphany grid.

In light of the results of this evaluation, and unless otherwise stated, the results presented in the remaining of this dissertation, produced by tests conducted with the Epiphany co-processor using DMA transfers, always used 1 slice line per DMA transfer.

4.3.2 Direct Access to Local Memory

In this strategy all interactions between host and eCores are performed through regions of the eCores local memory space. Figure 4.8 shows the organization of the local memory.

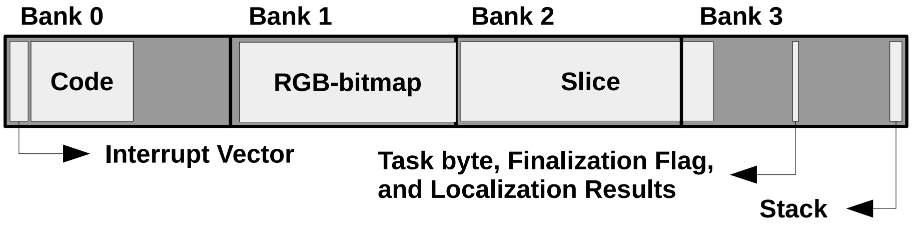


Figure 4.8: eCore local memory map when using direct access to local memory.

Banks 0 and 1 play exactly the same role as in the approach based on shared memory. Banks 2 and 3 are still used to accommodate the compressed RGB frame slice (9.375 KBytes), but this time contiguously (8 KBytes in bank 2 followed by 1.375 KBytes in bank 3), once there won't be alternate access to half-slices. However, in addition to holding up the stack, bank 3 now also holds a region (starting at local address `0x7000`) similar to the *control region* used in shared memory; this region will also receive the (partial) RGB localization results, whereas the "binarization" slice will overlap the RGB slice in bank 2.

In this method, the host needs to sequentially access each eCore local memory, to write data in there, to poll the finalization flag, and to read the localization data. The lack of parallelism in the interaction between host and eCores, coupled with the inherently slower communications inside the eMesh, make this strategy slower than using shared memory.

4.3.3 Evaluation of Data Exchange Strategies

Figure 4.9 shows the Binarization time per frame on the Epiphany, using the two approaches discussed for data exchange between host and eCores. The full Binarization time is a good indication of the time spent in data transfers (during Binarization) because, as further showed in Figure 4.10, the Binarization processing time is very small.

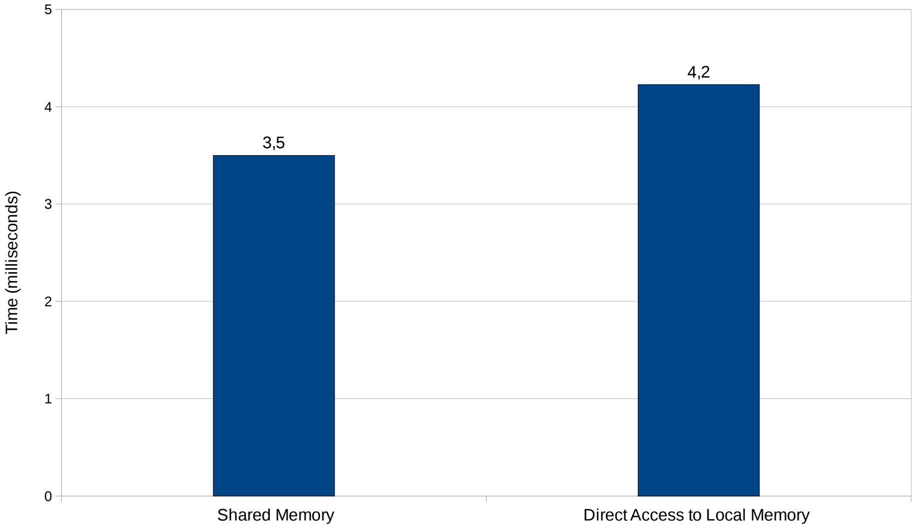


Figure 4.9: Binarization time per frame: impact of data exchange strategy.

As expected, the shared memory based approach is the fastest, but the difference to the direct access to the local memory is rather small (the speed-up is only 4,2/3,56=1,18). Again, this may have to do with the particular nature of the tested scenario, that is unable to generate enough load to bring out clearer differences between the tested approaches.

4.4 Parallel Processing with OpenMP and the Epiphany eSDK

This final hybrid version, that still primarily builds on the optimized Pthreads version, combines it with the two parallel processing approaches previously described: OpenMP on the ARM host, and the eSDK on the Epiphany co-processor. OpenMP is exclusively used to accelerate Compression, whereas the eSDK targets Binarization and Localization (in this regard, it applies the best processing options identified in section 4.3).

4.5 Final Evaluation

4.5.1 Optimized and Hybrid Approaches

With all optimizations applied and all intermediate evaluations performed (that allowed to identify the fastest implementation options) it becomes now possible to compare the performance of all versions of the object tracking application that run in the Parallella, either only on the ARM host, or also taking advantage of the Epiphany co-processor.

The application versions here considered are the ones discussed throughout this chapter (minus the one using OMPi, for reasons already discussed), and are identified as follows:

- Parallella - optimized Pthreads version; runs only on the ARM host;
- Parallella(OMP) - hybrid version, combining the optimized PThreads version with OpenMP; runs only on the ARM host;
- Parallella(eSDK) - hybrid version, combining the optimized PThreads version with the eSDK; runs both on the ARM host and in the Epiphany co-processor;
- Parallella(OMP+eSDK) - hybrid version, combining the optimized PThreads version with OpenMP and the eSDK; runs both on the ARM host and in the Epiphany co-processor;

These versions are summed up in Table 4.1, along with the maximum number of processing threads or cores involved in frame processing (thus excluding frame capture), in the last column. The later column provides information about the level of parallelism available and explored in each version (again, only for frame processing).

Application Version	*Execution Platform*	*Processing Strategy (Processing Stages)*	*Maximum Processing Threads or Cores*
Parallella	Parallella	Pthreads (all)	1
Parallella(OMP)	Parallella	OpenMP (all)	2
Parallella(eSDK)	Parallella + Epiphany	Pthreads (Compression) + eSDK (Bin. + Loc.)	1 + 16
Parallella(OMP+eSDK)	Parallella + Epiphany	OpenMP (Compression) + eSDK (Bin. + Loc.)	2 + 16

Table 4.1: Characteristics of the tracking application versions on the Parallella platform.

Figure 4.10 shows the results of the comparative evaluation. Experimental conditions were similar to those used in this chapter so far, except that the optimization level used with the `gcc/g++` compilers was changed from `-Os` to `-O2`. While this could potentially increase the code size too much (considering the memory limits of the Epiphany), such did not happen. At the same time, the extra optimizations performed by the `-O2` option allowed a measurable increase on the performance; this was observed in the Parallella version (the other versions lack a comparison reference), with an average frame processing time of only 7,5 ms, when compared with the previous time of 9,8 ms, measured with the `-Os` option (see Figure 4.5); this implied a speed-up of 9,8 / 7,5 = 1,30.

For each version, the processing time is represented in columns, split in its three main components (whose values are shown): Compression time (T_{Comp}), Binarization time (T_{Bin}), and Localization time (T_{Loc}). The overall processing time (the sum of these three components) is on the top of each column in parentheses.

For the Parallella(eSDK) and Parallella(OMP+eSDK) versions, $\mathrm{T}_{SharedCopy}$ is the time spent in transfers from host memory to shared memory, right after Compression and just before Binarization; also, T_{Bin} includes the time spent in transfers from shared memory to local memory (which overlaps with the time spent in the binarization itself in the eCores).

It is clear that using the Epiphany co-processor, ensures the smallest Binarization and

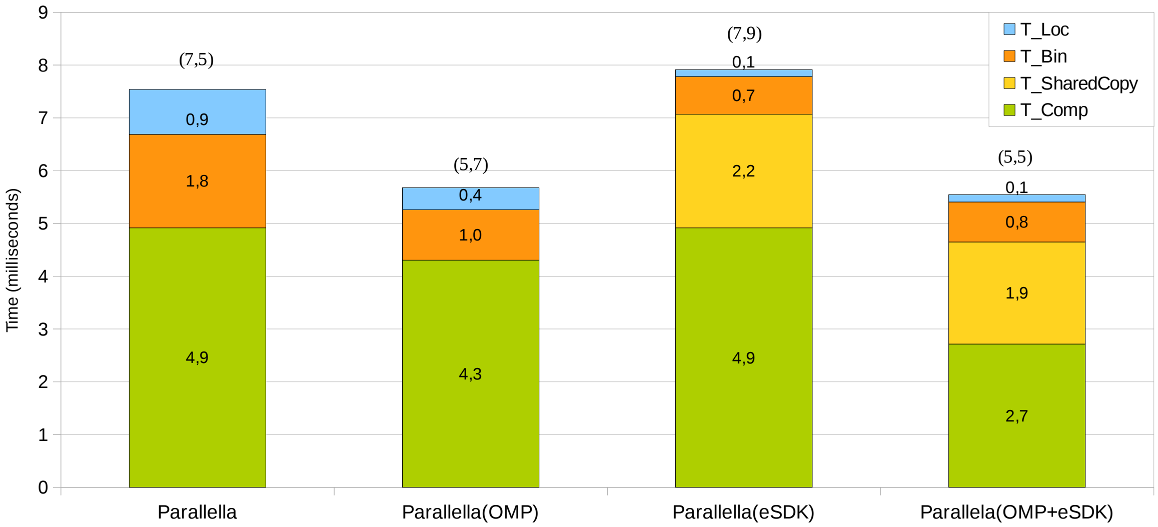


Figure 4.10: Parallella Versions Evaluation: (frame processing times), and decomposed frame processing times; times are averages, in milliseconds (ms).

Localization times (0,7 ms to 0,8 ms, and 0,1 ms). However, overall times are penalized by the time spent copying data from the host main memory to the shared memory (2,2 ms and 1,9 ms), to the extent that processing with only one POSIX thread takes less overall time (7,5 ms) than when offloading Binarization and Localization to the co-processor (7,9 ms in Parallella(eSDK)). Only when Compression is also parallelized with OpenMP on the host, the overall time decreases (to 5,5 ms). However, compared to fully using OpenMP in the host (which takes 5,7 ms), the performance gain was negligible (0,2 ms). Moreover, the decrease of the Compression time to 2,7 ms in the Parallella(OMP+eSDK) scenario is an oddity, once its OpenMP code is exactly the same as in Parallella(OMP), where it takes 4,9 ms. This is most probably a side effect, from the compiler rearranging code or doing other optimizations when OpenMP and eSDK code co-exist.

As observed, on the application versions that uses the eSDK, copies from the host main memory to shared memory impose a large performance penalty. If, however, the computations performed at the eCores were heavy enough, than the computation to communication ratio could become much more favourable. In order to demonstrate this, a simple test was made, by having the Binarization code to be executed several times for the same frame, in the Parallella version and in the Parallella(eSDK) version (noting that, in this scenario, copies to shared memory will still happen only one time per frame).

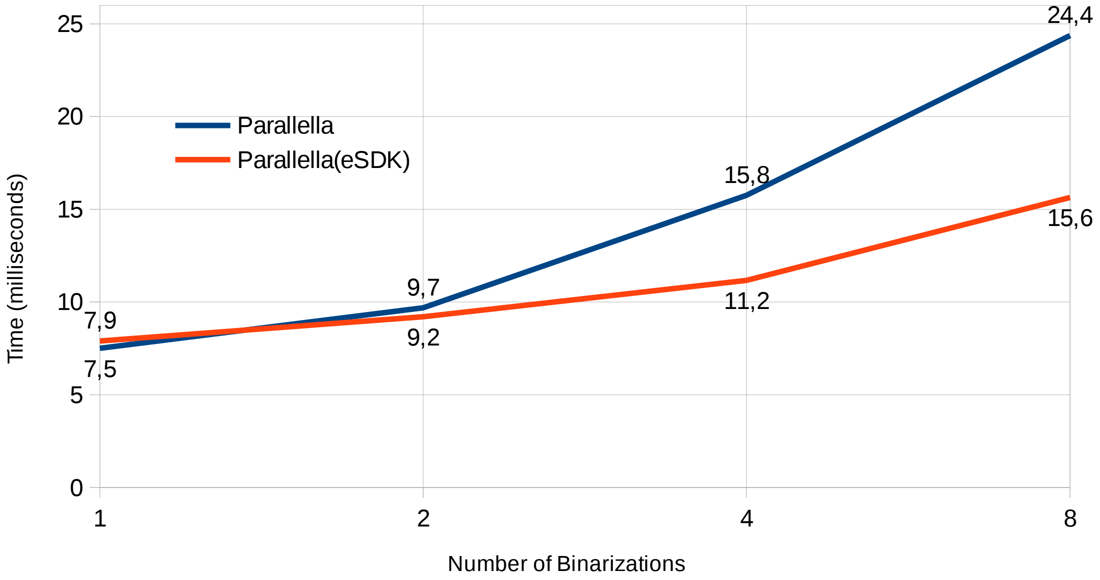


Figure 4.11: Binarization times when computing load increases.

As Figure 4.11 shows, if the computing load increases for the same data, it pays off to use the Epiphany, once the extra processing capacity amortizes the communication costs.

4.5.2 Comparison with the Raspberry-Pi SBCs

To get a comparison between the Parallella board and another embedded system, the tracking application was executed in several models of the popular Raspberry platform. The models considered were: a Raspberry Pi 1 (700 MHZ single-core, 512 MBytes RAM), a Raspberry Pi Zero (1 GHz single-core, 512 MBytes RAM), a Raspberry Pi 2 (900 MHz quad-core, 1 GByte RAM), and a Raspberry Pi 3 (1,2 GHz quad-core, 1 GByte RAM).

The benchmarking conditions were the same as the ones used to evaluate the Parallella platform, but the application versions tested were limited to the optimized Pthreads version, and the hybrid version with OpenMP. The following scenarios were thus tested:

- Rasp1 - Pthreads version on the Raspberry Pi 1;
- Rasp0 - Pthreads version on the Raspberry Pi 0;

- Rasp2 - Pthreads version on the Raspberry Pi 2;
- Rasp3 - Pthreads version on the Raspberry Pi 3;
- Rasp2(OMP) - hybrid version (Pthreads and OpenMP) on the Raspberry Pi 2;
- Rasp3(OMP) - hybrid version (Pthreads and OpenMP) on the Raspberry Pi 3.

These scenarios are summed up in Table 4.2 (similar to the Table 4.1 provided above).

Test Scenario	*Execution Platform*	*Processing Strategy (Processing Stages)*	*Maximum Processing Threads or Cores*
Rasp1	Raspberry Pi 1	Pthreads (all)	1
Rasp0	Raspberry Pi Zero	Pthreads (all)	1
Rasp2	Raspberry Pi 2	Pthreads (all)	1
Rasp2(OMP)	Raspberry Pi 2	OpenMP (all)	4
Rasp3	Raspberry Pi 3	Pthreads (all)	1
Rasp3(OMP)	Raspberry Pi 3	OpenMP (all)	4

Table 4.2: Characteristics of the testing scenarios on the Raspberry Pi platform.

Results of the evaluation on the Raspberry Pi SBCs are shown in Figure 4.12, together with the results measured on the Parallella platform, to ease a cross-platform comparison.

The figure includes an extra metric, in square brackets, on the top of the chart: the average frame capture time, in each scenario. Regarding this metric, the results reveal that only the Parallella and the Raspberry Pi 3 are able to keep with the frame generation rate of the Kinect sensor (of 30 fps, implying a capture time of around 33 ms). With average capture times ranging from 100 ms to 134 ms, the other boards would loose frames.

Frame processing times with the different Raspberry Pi models are more diverse than with the Parallella. With a single CPU-core available, the Raspberry Pi 1 and Zero models are clearly outperformed by the quad-core Raspberry Pi 2 and 3, and by the Parallella platform. However, the Raspberry Pi 2 with a single processing thread (Rasp2 scenario) is already on-par with the best Parallella scenarios and, with four threads (Rasp2(OMP)), it more than halves the processing time (to 2,3 ms). The performance of the Raspberry Pi 3 with a single thread (Rasp3 scenario) is close (2,7 ms) to the best Raspberry Pi 2

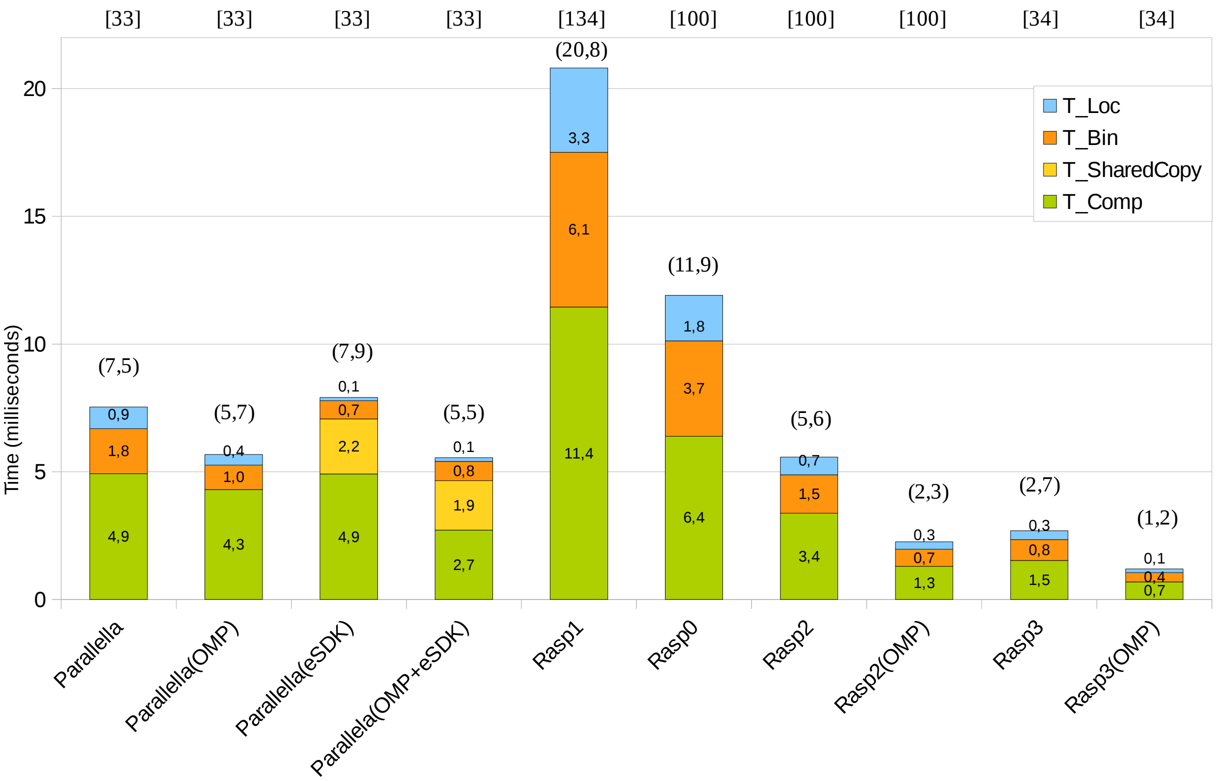


Figure 4.12: Multi-platform Results: [frame capture times], (frame processing times), and decomposed frame processing times; all times are averages, in milliseconds (ms).

scenario (2,3 ms), and with four threads (Rasp3(OMP) scenario), it more than halves the processing time (to 1,2 ms), in line with the Raspberry Pi 2 behavior.

When directly comparing the best Raspberry Pi 3 scenario (Rasp3(OMP)) with the best Parallella scenario (Parallella(OMP+eSDK)), the first attains a speedup of 5,5 / 1,2 = 4,58(3). Once the Raspberry Pi 3 has twice the number of ARM cores, and double clock frequency, the performance gap is understandable.

Finnaly, in terms of energy consumption at peak load, running our application, it was measured 6,5W in the Parallella(OMP+eSDK) test and 3,6W in the Rasp3(OMP) test (the Kinect sensor was powered by its own external power supply). Thus, the Raspberry Pi3 draw about half the power of Parallella.

Chapter 5

Conclusions and Future Work

This work introduced an initial version of an object-tracking application (prototype-level), based on the well-known Pthreads model, already capable of exploring (through concurrency) the ARM-based multi-core runtime of the Parallella board, for frame processing.

Through testing, it was possible to identify several bottlenecks, and opportunities for improvements, which made possible a second, optimized Pthreads version. Seeking to enhance its performance, this version was expanded through the OpenMP model, for automatic many-thread parallelization of several frame processing stages, in order to take full advantage of the Parallella dual-core architecture. This hybrid version, and the initial one are portable, meaning they can be executed on virtually any modern embedded system. However, they were still unable to take advantage of the Epiphany co-processor.

Two other hybrid versions were then developed, targeting the Epiphany accelerator: one combining the Pthreads optimized code, with routines and facilities offered by the low-level Epiphany Software Development Kit (eSDK); a subsequent variant of this version, bringing OpenMP into the equation, for a total of three programming models used at the same time; this last hybrid approach proved to be the fastest of all versions evaluated in the Parallella board, although by a very minimal margin when compared to the first OpenMP hybrid version.

Finally, for the portable versions developed (those based on Pthreads and/or OpenMP), its evaluation was extended to the Raspberry Pi family, to assess the merits of the Parallella heterogeneous platform in comparison to a popular conventional embedded platform.

By cross-comparing all evaluation results, it became clear that i) there was no meaningful performance advantage derived from using the Epiphany co-processor when running the tracking application in the Parallella board, and ii) the peak performance attained in the Parallella could be doubled and even (more than) quadrupled in the quad-core Rasperry Pi 2 and 3 models, respectively, using a simpler parallel programming approach.

Thus, considering the specific implementations developed for the case study targeted by this work, the tracking application seems to not take full advantage of the Parallella capabilities, which would be better exploited by workloads with higher computation/communication ratios. The need for costly data exchanges between the Parallella host and the eCores, and the scarce local memory at eCores (that dictated the need for time-consuming Compression), helped to turn the balance in disfavor of the Parallella.

To summit up, parallell programming with the Epiphany co-processor is hard and in order to reap its potential benefits the application domain must be carefully chosen.

Despite the somehow unsatisfactory results achieved in this work, academic and scientific literature provides abundant examples in which the Parallella and its Epiphany co-processor proved rewarding [Parc]. Therefore, in the future, the tracking problem will be revisited, by investigating better ways to unleash the full potential of the Parallella.

Bibliography

[AC97] J. K. Aggarwal and Q. Cai. Human motion analysis: a review. In *Proceedings IEEE Nonrigid and Articulated Motion Workshop*, pages 90–102, Jun 1997.

[Ada13a] Adapteva. Epiphany architecture reference (rev. 14.03.11), 2013.

[Ada13b] Adapteva. Epiphany sdk reference (rev. 5.13.09.10), 2013.

[AOM06] Yilmaz. Alper, Javed. Omar, and Shah. Mubarak. Object tracking: A survey. *ACM Comput. Surv.*, 38(4), December 2006.

[APD15] Spiros N. Agathos, Alexandros Papadogiannakis, and Vassilios V. Dimakopoulos. Targeting the parallella. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, pages 662–674, 2015.

[Avi04] S. Avidan. Support vector tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(8):1064–1072, Aug 2004.

[bd17] board db. The single board computer database. `https://www.board-db.org/`, 2017.

[BLM16] P. Brauer, M. Lundqvist, and A. Mällo. Improving latency in a signal processing system on the epiphany architecture. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 796–800, Feb 2016.

[Bro16] Nick Brown. epython: An implementation of python for the many-core epiphany coprocessor. In *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing*, PyHPC '16, pages 59–66, Piscataway, NJ, USA, 2016. IEEE Press.

[Bro17] E Brown. hacker board survey. `http://linuxgizmos.com/2017-hacker-board-survey-raspberry-pi-still-rules-but-x86-sbcs-make-gains/`, 2017.

[But97] David R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[CdFC95] Roberto Marcondes Cesar and Luciano da Fontoura Costa. A pragmatic introduction to machine vision, by r. jain, r. kasturi and b. g. schunck. *Real-Time Imaging*, 1(6):437 – 439, 1995.

[CKS95] V. Caselles, R. Kimmel, and G. Sapiro. Geodesic active contours. In *Proceedings of IEEE International Conference on Computer Vision*, pages 694–699, Jun 1995.

[CM99] D. Comaniciu and P. Meer. Mean shift analysis and applications. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1197–1203 vol.2, 1999.

[Fou14] Free Software Foundation. Gnu binutils. `https://www.gnu.org/software/binutils/`, 2014.

[Fou17a] Open Source Robotics Foundation. About ros. `http://www.ros.org/about-ros/`, 2017.

[Fou17b] Raspberry Pi Foundation. Raspberry pi - teach. learn, and make with raspberry pi. `https://www.raspberrypi.org/`, 2017.

[FSB98] Jay Fenlason, Richard Stallman, and Brent Baccala. Gnu gprof. `https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html`, September 1998.

[Gav99] D.M Gavrila. The visual analysis of human movement: A survey. *Computer Vision and Image Understanding*, 73(1):82 – 98, 1999.

[GYG15] Y. S. Gener, A. Yildiz, and S. Goren. Low-cost and low-power video filtering with parallel many cores. In *2015 9th International Conference on Electrical and Electronics Engineering (ELECO)*, pages 921–925, Nov 2015.

[IBM17] IBM. Introdução ao gprof. `https://www.ibm.com/developerworks/br/local/linux/gprof_introduction/index.html`, 2017.

[INA$^+$16] I. Iszaidy, R. Ngadiran, R. B. Ahmad, M. I. Jais, and D. Shuhaizar. Threading implementation on different hardware for travel time estimation purpose. In *2016 International Conference on Robotics, Automation and Sciences (ICORAS)*, pages 1–4, Nov 2016.

[Int17] Silicon Graphics International. Standard template library programmer's guide. `https://www.sgi.com/tech/stl/index.html`, 2017.

[iS17] i SCOOP. Industry 4.0: the fourth industrial revolution - guide to industrie 4.0. `https://www.i-scoop.eu/industry-4-0/`, 2017.

[Joh12] Bernadette Johnson. How the raspberry pi works. `http://computer.howstuffworks.com/raspberry-pi.htm`, 2012.

[KA11] D. Kaeli and D. Akodes. The convergence of hpc and embedded systems in our heterogeneous computing future. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*, pages 9–11, Oct 2011.

[L.11] Gwennap. L. Adapteva: More flops, less wats(white paper). `http://www.adapteva.com/wp-content/uploads/2011/06/adapteva_mpr.pdf`, 2011.

[Mat17] The MathWorks. System requirements for matlab r2017a. `https://www.mathworks.com/support/sysreq.html`, 2017.

[MG01] Thomas B. Moeslund and Erik Granum. A survey of computer vision-based human motion capture. *Computer Vision and Image Understanding*, 81(3):231 – 268, 2001.

[MPG15] V. Menezes, V. Patchava, and M.S.D. Gupta. Surveillance and monitoring system using raspberry pi and simplecv. In *Conference on Green Computing and Internet of Things (ICGCIoT)*, pages 1276–1278, 2015.

[mpi15] Mpi: A message-passing interface standard, version 3.1 ; june 4, 2015. `https://books.google.pt/books?id=Fbv7jwEACAAJ`, 2015.

[NVI] NVIDIA. Jetson tk1 embedded development kit. `http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html`.

[ONUA14] A. Olofsson, T. Nordström, and Z. Ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with epiphany. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 1719–1726, Nov 2014.

[Ope13] OpenMP. *OpenMP Application Program Interface, Version 4.0. 2013.* 2013.

[Ope17a] OpenCV. Opencv library. `http://opencv.org/`, 2017.

[Ope17b] OpenCV. Opencv: Tracking api. `http://docs.opencv.org/3.1.0/d9/df8/group__tracking.html`, 2017.

[Ope17c] OpenKinect. Openkinect/libfreenect. `https://github.com/OpenKinect/libfreenect`, june 2017.

[PA04] Sangho Park and J. K. Aggarwal. A hierarchical bayesian network for event recognition of human actions and interactions. *Multimedia Systems*, 10(2):164–179, Aug 2004.

[Para] Parallella. Parallella models. `https://www.parallella.org/parallella-models/`.

[Parb] Parallella. Parallella models. `https://www.parallella.org/programming/`.

[Parc] Parallella. Parallella publications. `https://www.parallella.org/publications/`.

[Pas01] G. Paschos. Perceptually uniform color spaces for color texture analysis: an empirical evaluation. *IEEE Transactions on Image Processing*, 10(6):932–937, Jun 2001.

[RR16] James Ross and David Richie. An openshmem implementation for the adapteva epiphany coprocessor. In *OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments - Third Workshop, OpenSHMEM 2016, Baltimore, MD, USA, August 2-4, 2016, Revised Selected Papers*, pages 146–159, 2016.

[RRPS15] David Richie, James Ross, Song Park, and Dale Shires. Threaded mpi programming model for the epiphany risc array processor. *Journal of Computational Science*, 9:94 – 100, 2015. Computational Science at the Gates of Nature.

[SM00] Jianbo Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, Aug 2000.

[SS16] A. Soetedjo and I. K. Somawirata. Implementation of face detection and tracking on a low cost embedded system using fusion technique. In *2016 11th International Conference on Computer Science Education (ICCSE)*, pages 209–213, Aug 2016.

[TK17] Mai Thanh Nhat Truong and Sanghoon Kim. Parallel implementation of color-based particle filter for object tracking in embedded systems. *Human-centric Computing and Information Sciences*, 7(1):2, Jan 2017.

[Val17a] Valgrind. Callgrind: a call-graph generating cache and branch prediction profiler. `http://valgrind.org/docs/manual/cl-manual.html`, 2017.

[Val17b] Valgrind. Valgrind user manual. `http://valgrind.org/docs/manual/cg-manual.html`, 2017.

[VRH$^+$16] S. Vaas, M. Reichenbach, J. Hofmann, T. Stadelmayer, and D. Fey. Embedded parallel computing accelerators for smart control units of frequency converters. In *ARCS 2016; 29th International Conference on Architecture of Computing Systems*, pages 1–5, April 2016.

[Wei17] Josef Weidendorfer. Kcachegrind. `http://kcachegrind.sourceforge.net/html/Home.html`, 2017.