J Sign Process Syst (2018) 90:913–929
DOI 10.1007/s11265-017-1267-1

Image Processing Units on Ultra-low-cost Embedded Hardware: Algorithmic Optimizations for Real-time Performance

Suraj Nair[1] · Nikhil Somani[1] · Artur Grunau[2] · Emmanuel Dean-Leon[2] · Alois Knoll[2]

Received: 18 August 2016 / Revised: 17 March 2017 / Accepted: 12 July 2017 / Published online: 1 August 2017

Abstract The design and development of image processing units (IPUs) has traditionally involved trade-offs between cost, real-time properties, portability, and ease of programming. A standard PC can be turned into an IPU relatively easily with the help of readily available computer vision libraries, but the end result will not be portable, and may be costly. Similarly, one can use field programmable gate arrays (FPGAs) as the base for an IPU, but they are expensive and require hardware-level programming. Finally, general purpose embedded hardware tends to be under-powered and difficult to develop for due to poor support for running advanced software. In recent years a new option has surfaced: single-board computers (SBCs). These generally inexpensive embedded devices would be attractive as a platform on which to develop IPUs due to their inherent portability and good compatibility with existing computer vision (CV) software. However, whether their performance is sufficient for real-time image processing has thus far remained an open question. Most SBCs (especially the ultra-low-cost ones which we target) do not offer CUDA/OpenCL support which makes it difficult to port GPU-based CV applications. In order to utilize the full power of the SBCs, their GPUs *need* to be used. In our attempts at doing this, we have observed that the CV algorithms which an IPU uses have to be re-designed according to the OpenGL support available on these devices. This work presents a framework where a selection of CV algorithms have been designed in a way that they optimize performance on SBCs while still maintaining portability across devices which offer OpenGL ES 2.0 support. Furthermore, this paper demonstrates an IPU based on a representative SBC (namely the Raspberry Pi) along with two CV applications backed by it. The robustness of the applications as well as the performance of the IPU are evaluated to show that SPCs can be used to build IPUs capable of producing accurate data in real time. This opens the possibilities of large scale economically deployment of vision system especially in remote and barren lands. Finally, the software developed as a part of this work has been released open source.

Keywords Embedded vision · Image processing units · Human tracking

✉ Suraj Nair
suraj.nair@tum-create.edu.sg

[1] TUMCREATE, Singapore, Singapore

[2] Technische Universität München, München, Germany

1 Introduction

The term *image processing unit* (IPU) lacks a formal definition in computer vision (CV) literature. It has been used only in a generic sense to describe components carrying out image processing tasks: a camera developed for spacecrafts [1], image compression [2], or a multimedia processor accelerating image transformations in hardware [3].

We propose that an image processing unit be defined as a device that combines the following functions:

- image acquisition, for instance from an attached camera
- image processing for obtaining high-level information required by a specific CV application

– remote access to processed image data and extracted image characteristics

Consequently, IPUs are closely related to smart cameras. Every smart camera has an embedded IPU at its core, and adds a layer of CV logic on top of it. An IPU, however, does not have to be an embedded device, and is not an active system that generates events or makes decisions on its own [4]. Both PC-based and embedded IPUs find applications in industrially-relevant areas: machine vision, video surveillance, and human-computer interfaces. Improving their performance [5], and reducing their cost [6] and power consumption [7] has been a research focus.

Current-generation IPUs are either specially-programmed heterogeneous embedded systems combining field-programmable gate arrays (FPGAs), digital signal processors (DSPs) and microprocessors; or PCs running CV software, frequently accelerated by exploiting general-purpose computing on graphics processing units (GPGPU) techniques. The former are employed when portability is paramount, the latter when ease of programming is the primary concern.

Despite continuing improvements, IPU development has always been an exercise in trade-offs. Compact IPUs are difficult to program since widely available CV software does not run on FPGAs or DSPs, which instead must be programmed using such low-level languages as VHDL and assembly. At the same time, PC-based IPUs are too unwieldy for many applications. Moreover, the need for processing to have real-time performance effectively precludes the use of inexpensive hardware.

Nevertheless, ongoing developments in the single-board computer (SBC) space have the potential to disrupt established IPU design and development practices. The SBC market has seen rapid growth in recent years due to the release of the Raspberry Pi,[1] a $35 computer (Fig. 1). As hardware specifications and software compatibility of SBCs improved, so did their feasibility for computer vision applications.

Currently, SBCs are a platform that promises to offer the best of two worlds: the ease of programming of PCs, and the small footprint of FPGAs and DSPs. Consequently, they would be perfectly poised for development of portable, low-cost IPUs if it were not for their performance. Unfortunately, while existing CV software can often be ported to run on modern SBCs, their CPUs are not fast enough to achieve real-time processing.

Luckily, modern SBCs are composed of not only a CPU, but also many auxiliary multimedia processors, including a graphics processing unit (GPU). As a result, selected GPGPU techniques could be used to offload certain kinds of computation onto a GPU — in much the same way as CV applications are optimised to run on mobile devices [8] — considerably improving the image processing performance of SBCs.

[1] http://www.raspberrypi.org/

Figure 1 Raspberry Pi Model B. Photograph by Jwrodgers (CC BY-SA).

Given all of the above facts, it should be clear that the process of developing IPUs could be simplified and streamlined. The use of SBCs for this purpose holds a lot of promise, even though the adequacy of their performance remains an open question. Constant progress in this area has thus far not prompted the CV community to build a real-time, SBC-based IPU, mainly due to the fact that standard CV algorithms would have to be adapted to run on it efficiently. In addition one can observe that although popular SBCs like the Raspbery Pi have evolved in terms of CPU power, the GPU architecture still remains unchanged. This fact reiterates the necessity of CV algorithms which are reengineered and optimised for low cost SBCs.

Consequently, in this paper we demonstrate that SBC-based IPUs are feasible by implementing and evaluating an IPU based on the Raspberry Pi along with two example CV applications that use it to extract high-level information from camera images.

We selected the Raspberry Pi (Model B) for this task because it can be considered representative of existing SBCs performance- and capability-wise. However, there are good reasons why one might prefer it for development over similar or more powerful computers. For example, at $35 the Raspberry Pi is one of the cheapest SBCs available today, and its performance/price ratio is unrivalled.

More importantly, however, the Raspberry Pi has — partly because its main focus is on education — by far the biggest and most thriving community of all SBCs. The advantages of this should not be underestimated: software support is first-class, one can easily find an answer to virtually any question regarding the computer, and there are countless projects pushing the Raspberry Pi to its limits.

At the core of the Raspberry Pi (Model B) is the Broadcom BCM2835 system on a chip (SoC) which consists of an ARM1176JZF-S 700 MHz processor, a VideoCore IV

GPU, and 512 MB of RAM. Due to its low frequency, and the fact that it only has a single core, the CPU's performance is not good enough to achieve real-time processing in CV applications without delegating some of the work to other processing units. Fortunately, the VideoCore IV GPU supports OpenGL ES 2.0, which allows the Raspberry Pi to use its GPU for general-purpose computation by framing image processing algorithms as rendering operations.

2 Related Work

The shortcomings of current approaches to IPU development are widely acknowledged, and there have been several attempts to improve selected aspects of the situation. For instance, [9] introduced a new high-level methodology for implementing vision applications on smart camera platforms. It proposed a holistic approach to designing and programming heterogenous embedded IPUs that results in a simpler development process.

An FPGA-based IPU built using commercial off-the-shelf components was described in [1]. Its design indicates that it is possible to create "small, low-cost, high-performance" IPUs, although they still have to be programmed using various low-level languages.

Several specialised CV systems integrating SBCs in some capacity have appeared in recent years. For example, [10] describes a project in which a stereo vision application for small water vehicles was developed by combining a Raspberry Pi with a PC-104 unit. All image processing was done on the SBC using the OpenCV library. Sadly, due to the fact that only the CPU was utilised, the application operated at a rather low framerate of 2-3 frames per second (FPS).

Similarly, [11] outlines an embedded IPU developed to provide visual odometry and localization of moving objects for unmanned aerial vehicles. It ran feature detection and parameter estimation algorithms on an overclocked Raspberry Pi. Although implementation details of the system were not divulged, it is known that it processed captured camera images at approximately 1 FPS.

Finally, there have been successful attempts to accelerate augmented reality applications on smart phones using GPGPU techniques. For instance, [12] compared 2 implementations of the SURF local feature detector: the one that ships with the OpenCV library, and a custom version designed to run on low-power GPUs and written in OpenGL ES GLSL. A range of mobile devices was used to benchmark the 2 implementations, and the results showed that the GPU version was up to 14× faster.

There was also a related project, documented in [13], that dealt with creating panoramic images in real-time on mobile devices. It used the FAST corner detector for feature point extraction, and a motion model to track camera orientation. The use of OpenGL shaders in the project's implementation allowed it to process high-resolution images (2048px×512px) at 20 FPS on a Samsung Galaxy S2.

Considering the state of the art, the primary contribution of this work is twofold. First of all, we demonstrate how CV algorithms can be adapted to run on SBCs to achieve real-time performance, thus making SBC-based IPUs feasible. Second, we describe how multiple networked IPUs can be used to carry out computation in parallel, which further improves the performance of CV systems.

3 Algorithm Adaptation Strategies

Having selected the Raspberry Pi to be the SBC on which the 2 test applications would run, we needed to decide what techniques to use to implement the required CV algorithms. The approach outlined in this section tried to balance good performance with ease of programming.

Non-critical algorithm fragments, such as pre- and post-processing of small amounts of data, could be allowed to execute on the CPU. We decided to use OpenCV, the well-known open source library of CV functions, for selected small tasks. Although it lacked hardware acceleration on the Raspberry Pi, it did not introduce any bottlenecks due to the limited amount of work it was given.

Most image processing operations, however, had to be optimised in order to get satisfactory performance. In such cases OpenGL ES 2.0, or more specifically its shading language, GLSL, can be used to send data to be processed on the Raspberry Pi's GPU.

3.1 OpenGL ES 2.0 Pipeline

As OpenGL ES 2.0 GLSL's main purpose is graphics rendering, computer vision algorithms need to be expressed in terms of drawing 3D shapes and manipulating the resulting pixels. Fortunately, the problem of accelerating image processing algorithms using OpenGL is well understood and solutions have been described [14–17]. Consequently, this section only briefly describes how OpenGL-based GPGPU works.

The rendering process of OpenGL ES 2.0 consists of many stages, as illustrated on Fig. 2. When a draw call is issued, every vertex to be rendered first goes through a vertex shader. There its properties, such as position and colour, may be modified. Next vertices are assembled into primitives (e.g. lines or triangles). The visibility of the resulting shapes is then tested and they are culled or cropped if they lie outside of the viewport.

Subsequently, remaining primitives are rasterised, i.e. turned into fragments which may eventually become pixels.

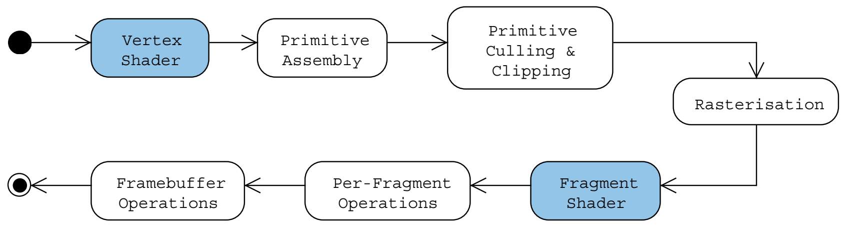


Figure 2 Processing pipeline of OpenGL ES 2.0. Programmable stages have a blue background.

Fragments are first processed by a fragment shader which either decides their final colour, or discards them. Additional specialised per-fragment operations may further reduce the number of fragments that become visible. Finally, fragments reach a framebuffer where they either replace or are blended with its existing contents to become pixels.

Two of the stages mentioned above are programmable: vertex shading and fragment shading. They execute custom code to carry out their functions. As a result, they can be used for many kinds of calculations unrelated to graphics rendering, provided it is possible to formulate those calculations in terms of vertex and fragment processing.

One of the first things that need to be established when implementing algorithms in OpenGL ES 2.0 GLSL is how data that needs to be processed will be uploaded onto the GPU. Shaders have access to various sources of data, all of which are shown on Fig. 3.

Each data source has distinct characteristics and is suitable for storing and providing different types of data. Every vertex processed by a vertex shader has a set of vertex attributes associated with it. They store properties such as position or colour that usually need to change per vertex. For each vertex it processes, the vertex shader may generate varying variables that are propagated by interpolation to all fragments of the primitive to which the vertex belongs. This is the only way to share data between a vertex and a fragment shader.

Textures and uniform variables are accessible from both vertex and fragment shaders. Textures are 2D arrays that can be used to store large amounts of data, e.g. per-fragment properties. Uniform variables, on the other hand, contain

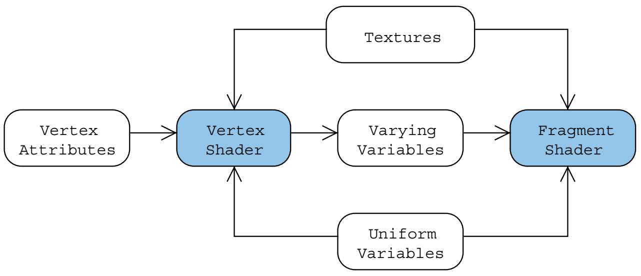


Figure 3 Data sources accessible to OpenGL ES 2.0 vertex and fragment shaders.

arguments that can change at most once per rendering call. As a result, they are often used to parameterise algorithm runs.

Finding the right way of expressing CV algorithms using vertex and fragment shaders, and feeding them data to be processed, is key to exploiting GPUs to improve computing performance. The sections that follow will describe how the algorithms needed by the test applications were adapted to offload most of their work onto the Raspberry Pi's GPU by utilising OpenGL ES 2.0.

4 Algorithm Adaptation Challenges

While implementing the required CV algorithms, we encountered several limitations of the Raspberry Pi that complicated the development to various degrees. Some of them can be attributed to the embedded nature of the Raspberry Pi, whereas others are a direct result of its low cost. What is more, they could be roughly divided into 2 categories: limitations related to OpenGL ES 2.0, and challenges associated with compilation and debugging.

As an example of an OpenGL ES 2.0-related limitation, the VideoCore IV GPU provides only 8 texture units. As they are the only location where large amounts of data can be stored, the number of data batches—such as camera frames—that can be processed togethes is rather low. This precludes the use of the GPU in certain types of background modelling, e.g. 1-G and Gaussian mixture models.

Moreover, the overall number of texture accesses in a single shader execution cannot exceed 64. This limit is further decreased proportionally to the shader's number of instructions, i.e. the more complex a shader is, the fewer texels it can access. As a result, reduction algorithms can only operate on texel blocks sized up to 8×8, which becomes 4×4 in practice. This usually results only in lower performance, but certain histogram generation algorithms need to be able to reduce blocks the size of a histogram, which makes it impossible to create histograms with more than a dozen buckets using such methods on the Raspberry Pi.

Another limitation of the OpenGL ES 2.0 implementation available on VideoCore IV is the fact that the only way to return data from the GPU's programmable units is to render into a single output texture. Consequently, a vertex shader cannot return values directly: instead, it must forward its results through a fragment shader. On top of that, all data returned from a fragment shader must fit into 1 texel. This makes some calculations more difficult to implement in GLSL than they should be because dummy fragment shaders and multiple rendering passes are needed to produce results.

Finally, the above problem is exacerbated by the lack of floating-point textures on the Raspberry Pi. As a result, texels are restricted to storing single-byte values. While floating-point data can still be uploaded onto the GPU using vertex attributes, without floating-point textures it is only possible to return small-range integers from OpenGL ES 2.0. This reduces arithmetic precision to levels at which statistical computation — which could be used in background modelling — is unfeasible.

Challenges associated with compilation arise from the Raspberry Pi's relatively slow CPU. Developing directly on the SBC becomes impractical before long due to the fact that it takes several seconds to compile and link even the simplest programs. Building more complex applications, such as OpenCV, can easily take many hours. Cross-compilation is therefore of special importance when developing for the Raspberry Pi: it is the only way to avoid lengthy build cycles and get feedback quickly when writing software.

Debugging is another area where the Raspberry Pi doesn't offer the best experience. Some useful diagnostic tools, such as Valgrind, do not work on the SBC at all. Others, such as the GNU Debugger, are very slow and require special configuration to work around certain quirks of the Raspberry Pi. All in all, debugging applications developed for the SBC is much more cumbersome than it would be on a PC.

5 Examples of Adaptated Algorithms

5.1 Colour Space Conversion

Colour space conversion is the primary building block of all other image processing algorithms discussed in this chapter. Due to the fact that it is a simple image transformation, it lends itself to straightforward implementation using OpenGL ES 2.0. For this work, shaders converting between the following colour spaces were written: RGB, YUV, YCoCg, and HSV.

The input to the algorithm is a texture containing raw data from a captured video frame and a uniform variable specifying what conversion to execute. The output is a texture of the same frame in a different colour space. Each input texture element (texel) can be mapped independently and in parallel using a selected conversion function. Such processing is best done by creating a fragment for every input texel and pushing it through a fragment shader that performs specific colour transformation.

A common technique to generate a set of fragments that cover a whole texture is to simply draw a full-screen rectangle the size of the texture. It will then be automatically broken up into the necessary fragments in the rasterisation process.

The functions used to transform texel colours are universally simple, sometimes to the point of requiring only a single matrix multiplication. Listing 1 shows an example fragment shader defining and using such a function.

5.2 Colour Thresholding

Colour thresholding is used extensively by the simple smart camera test application (see Section 6.1). It builds on colour

Listing 1 Fragment shader converting from the RGB to the YUV colour space.

```
uniform sampler2D texture;
varying vec2 textureCoordinates;

const mat3 rgb2yuvMatrix = mat3(
  0.299, 0.587, 0.114,
  -0.14713, -0.28886, 0.436,
  0.615, -0.51499, -0.10001
);

vec3 rgb2yuv(in vec3 rgbColor) {
  return rgb2yuvMatrix * rgbColor;
}

void main(void) {
  vec4 color = texture2D(texture, textureCoordinates);
  vec3 yuvColor = rgb2yuv(color.rgb);
  gl_FragColor = vec4(yuvColor, 1);
}
```

space conversion: thresholding is done in the HSV colour space. The algorithm is another example of an image transformation, and as a result can be expressed in OpenGL ES 2.0 without difficulty.

The algorithm takes as input a texture containing HSV data from a captured video frame and two uniform variables specifying the lower and upper thresholds for the Hue, the Saturation, and the Value channels, respectively. In the case of colour thresholding, processing follows the same pattern as with colour space conversion: a particle is created for every input texel, and a fragment shader decides how to transform it.

The fragment shader used to threshold captured video frames is shown on Listing 2. It works by comparing each texel's HSV colour to a given lower and upper threshold. The texel is then kept as is if its colour lies between the 2 thresholds, or discarded — leaving transparent background it its place — if it does not.

5.3 Background Subtraction

Background subtraction is an image transformation algorithm that analyses all pixels of an image to determine which ones belong to the background and should be discarded. As a result, it can be expressed in OpenGL ES 2.0 quite naturally. The specific algorithm we implemented was 1-G background subtraction [18].

1-G background subtraction first of all needs to build a background model that describes — using one Gaussian per texel's channel — change characteristics of the background. This model is created before processing starts by capturing several frames containing no foreground objects and finding, for every channel at each texel coordinate, a normal distribution that best fits the channel's values at that coordinate across all captured frames. This background model generation is executed on the CPU; the GPU is not involved due to its limitations described in the previous section (Fig. 4).

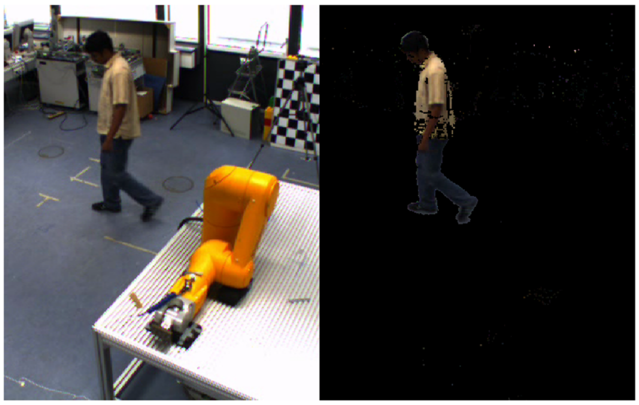


Figure 4 Captured video frame before and after background subtraction.

The Gaussians that model background for the purpose of 1-G background subtraction have their parameters estimated using the maximum likelihood method. Background model generation and subsequent background subtraction operate on texels whose values are represented in the YCoCg colour space. This choice of colour space results in changes in luminance and chromaticity being considered separately, which in turn makes background subtraction more robust.

Once a background model has been created, it is serialised into 2 textures: one storing the means of all Gaussians, one

Listing 2 Fragment shader filtering out fragments whose colour lies outside a specified threshold.

```
uniform sampler2D texture;
varying vec2 textureCoordinates;

uniform vec3 lowerHsvThreshold;
uniform vec3 upperHsvThreshold;

void main(void) {
  vec3 hsvColor = texture2D(texture,
                            textureCoordinates).xyz;
  bvec3 lowerThreshold = bvec3(step(lowerHsvThreshold,
                                    hsvColor));
  bvec3 upperThreshold = bvec3(step(hsvColor,
                                    upperHsvThreshold));

  if (all(lowerThreshold) && all(upperThreshold))
    gl_FragColor = vec4(color, 1.0);
  else
    discard;
}
```

containing their variances. They become, together with a texture containing YCoCg data from a video frame to be processed, input to the background subtraction process. To execute it, the standard technique of creating a particle for every input texel, and letting a fragment shader transform it, is used.

The function that classifies texels as foreground or background bases its decisions on their deviation from the background model. For each fragment, all channels of the corresponding texel are compared with the mean values stored in the background model at the texel's coordinates. If the channel's values lie within 3 standard deviations — as recorded in the second texture of the background model — of their associated mean values, the texel is considered background and discarded. Otherwise, it is output without modification as foreground.

5.4 Occlusion Testing

When a test application is tracking multiple targets, it must be able to detect when they start occluding each other in, or leaving the field of view of, any camera used. This information is needed for the application to know which cameras cannot see certain objects, and should not be relied upon when tracking those objects.

Occlusion testing is relatively easy to implement on the GPU due to the fact that the functionality is often used in computer graphics and is therefore available as part of OpenGL ES 2.0. To exploit it one only needs to pre-process their input data before passing it to OpenGL ES 2.0, and then post-process results retrieved from the GPU.

The occlusion testing algorithm implemented as part of this work takes only 1 input: a set of vertex attributes specifying the locations of all currently tracked targets. The algorithm's output is a list of floating-point numbers that contains an entry for each target indicating its visibility percentage.

The algorithm's operation involves 2 steps. To begin with, a destination texture the size of a camera frame is created. Next, each target's pose is projected onto that texture as a filled polygon of a different colour. At this point, OpenGL ES 2.0 automatically calculates each polygon's distance from the camera, and uses a technique called *depth buffering* to ensure that polygons closer to the camera are drawn on top of those farther away.

Finally, the resulting texture is fetched from the GPU, and the number of texels of each colour is counted. Each obtained count is then associated with a target based on their common colour, and divided by the number of pixels the target's pose was expected to generate. This step produces a list of target visibility percentages, which the algorithm subsequently returns (Fig. 5).

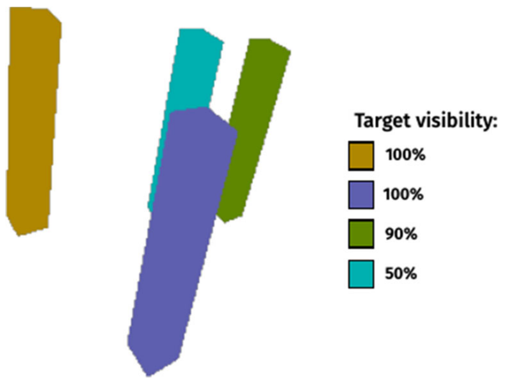


Figure 5 Target poses are drawn in order of their distance from the camera (*left*); next, the percentage of visible pixels is computed for each pose (*right*).

5.5 Foreground Coverage

Foreground coverage computation is a typical example of a reduction algorithm. As a result, there exist best practices [19, 20] that can be applied to implement it using OpenGL ES 2.0. At its most basic, the algorithm requires 2 inputs: an RGB foreground texture, and a set of vertex attributes specifying the location of a scan grid cell whose foreground coverage is to be computed. The output of the algorithm is, unlike all previously discussed algorithms, a single number derived from all relevant input texels. As a result, determining foreground coverage is considerably more complex than the simple image transformation analysed so far.

Here the foreground coverage of a 3D Polygon (scan grid cell) projected on a 2D surface is computed. The specific method used to compute foreground coverage of a scan grid cell involves several steps. First of all, the inputs need to be combined and pre-processed to make them ready for reduction. This is achieved by projecting the cell onto the foreground texture, cutting out the resulting 2D polygon, and colour-coding the polygon's area and any foreground texels it contains using red and green, respectively. Figure 6 illustrates the transformation.

Next, the pre-processed texture undergoes reduction based on a pyramid approach [19]. To begin with, it is divided into square blocks of 16 texels. Next, a fragment is generated for every block by drawing a fullscreen rectangle that is 1/16th the size of the texture. Each fragment is then processed by a shader that averages the colour values (red and green) of its corresponding texels to produce a new texel. Finally, all values returned from the fragment shader are collected into a temporary texture that is 16 times smaller than the pre-processed texture. The whole process is visualised on Fig. 7.

The reduction operation is repeated several times, with each temporary result texture becoming the input of

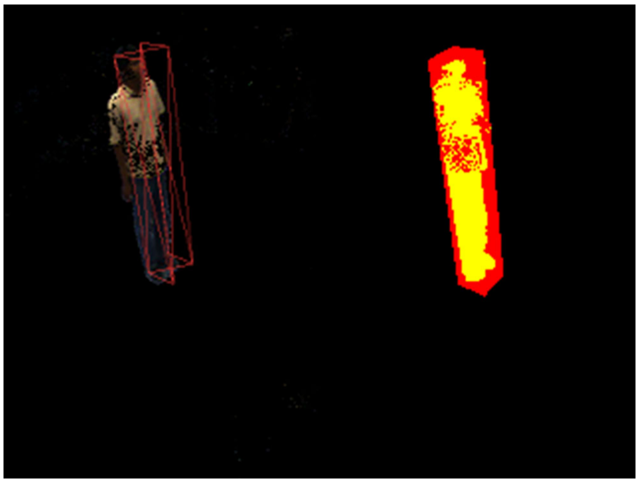


Figure 6 Foreground texture with a scan grid cell marked (*left*) is pre-processed to prepare it for reduction (*right*).

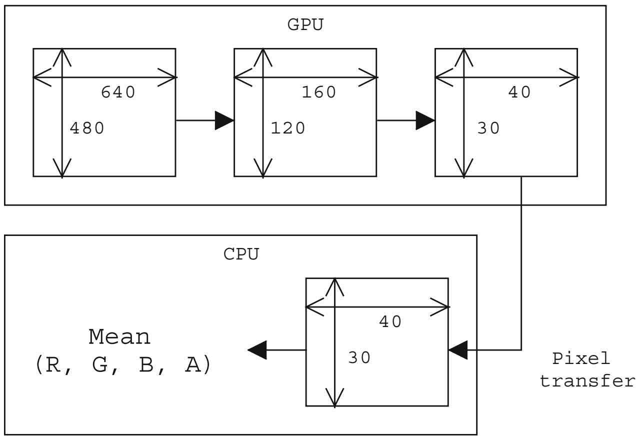


Figure 8 Several reduction steps are required to compute the average colour of a 640px×480px texture.

the next iteration. Eventually, the original texture (sized 640px×480px) is turned into an average texture small enough (40px×30px) to be post-processed on the CPU. At that point, the last result texture's data is fetched from the GPU into a pixel array. The colour values of the retrieved pixels are subsequently averaged to produce a single mean pixel. Figure 8 shows the individual reduction steps.

Due to the way colour channels were assigned at the pre-processing stage, the value of the red channel of the computed mean pixel can be interpreted as the percentage of frame image taken up by the scan grid cell whose foreground coverage is being determined. Similarly, the green channel's value is the percentage of frame image filled by the foreground contained in that cell. If we divide the latter number by the former one, we will arrive at the fraction the algorithm is looking for: the foreground coverage of the grid cell provided for processing.

While the described implementation of the foreground coverage algorithm exhibits good performance, but would incur too much overhead if used to process in one run the hundreds of cells that make up a scan grid: several rendering passes followed by a data fetch from the GPU would be needed per cell.

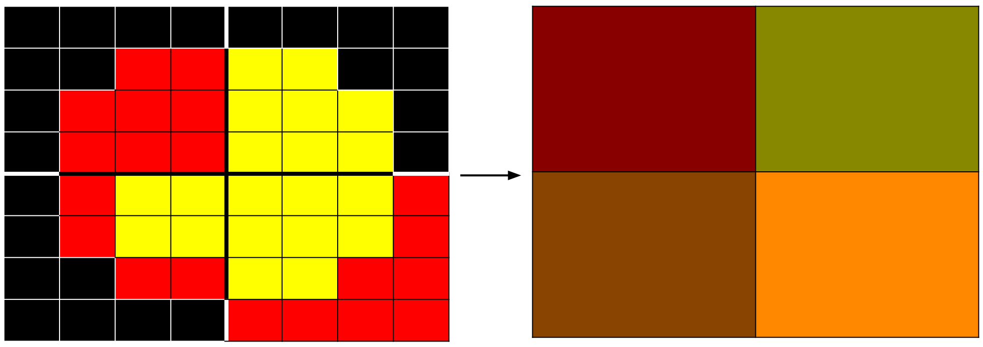
Fortunately, it is possible to optimise the algorithm to process scan grid cells in batches. First of all, the fact that only 2 colour channels are needed to compute foreground coverage of a single cell can be exploited: data associated with 2 cells can be stored in one texture at the pre-processing stage if the blue and the alpha channels are used for the second cell.

Moreover, as the input texture's size (640px×480px) is well below the maximum supported by OpenGL ES 2.0 (2048px×2048px), several pre-processed textures can be tiled and stored in one big texture sheet for the purpose of reduction. Benchmarking revealed that using a 4x4 sheet in this manner gives best results. Combining both optimisations allowed us to process 32 cells at a time, greatly increasing performance of foreground coverage computation.

5.6 Histogram Generation

Since histograms find many uses in graphics and CV applications, and because generating them on the GPU is not trivial, some research into accelerating histogram generation using GPGPU techniques has has been carried out

Figure 7 4 blocks of 16 texels are reduced to 4 texels by averaging.



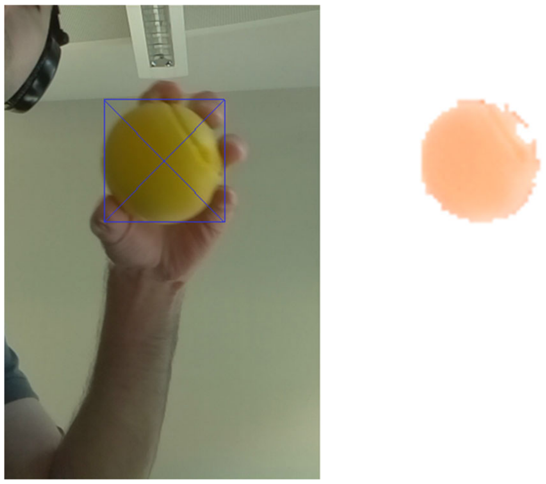


Figure 9 Colour-thresholded texture with a projected model marked (*left*) is pre-processed to prepare it for histogramming (*right*).

over the years [21–23]. While most presented methods require Nvidia CUDA, several could be implemented on the Raspberry Pi using OpenGL ES 2.0.

The histogram computed here is a 2D Joint Probability Histogram. In our implementation we initially intended to follow the approach described in [22]. It is based on the idea of dividing the image to be histogrammed into blocks the size of the desired histogram, generating local histograms for those blocks, and combining them using a reduction technique similar to the one described in Section 5.5.

However, due to the problems mentioned in Section 4, the approach introduced in [23] was taken in the end. The inputs of the resulting algorithm, in its standard version, are: an HSV foreground or thresholded texture, and a set of vertex attributes specifying the pose of the object model whose projected region histogram is to be computed. The output of the algorithm is, unlike all previously discussed algorithms, a 2D array of floating-point numbers representing the generated histogram.

The algorithm consists of a number of steps. First of all, the inputs need to be combined and pre-processed. This is achieved by projecting the object model whose projected/warped region histogram is being generated onto the input texture, cutting out the resulting 2D polygon, and removing all channels apart from hue and saturation from the created texture. Figure 9 illustrates the transformation.

Next, an output texture in which a histogram will be stored on the GPU is set up. It has the same size as the histogram, i.e. 10×10 in our case. Each of its texels is treated as a histogram bucket: its initial value of 0 gets incremented by 1 for every fragment rendered into it.

Afterwards, a set of points covering the the pre-processed texture are drawn. A special vertex shader is then used to transform them: it selects a bucket in the output texture for each point based on the the hue and the saturation of its corresponding texel from the pre-processed texture. Rasterisation turns these points into fragments that increment the values of their assigned buckets in the output texture. This main stage of the algorithm is illustrated on Fig. 10.

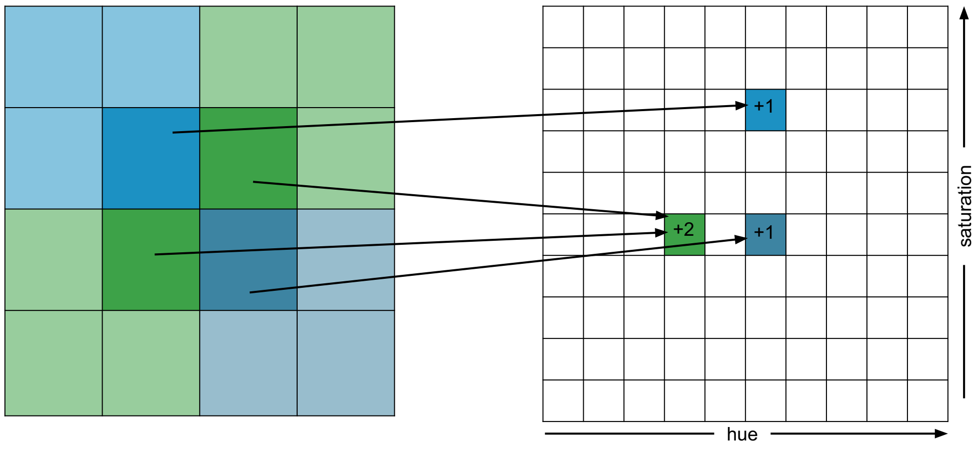
Finally, the output texture is fetched from the GPU, and turned into a 2D array of floating-point numbers that represent the normalised result histogram.

As was the case with foreground coverage computation, the described implementation of histogram generation exhibits good performance, but would incur too much overhead if used to process in one run the tens of projected models regions produced by algorithms such as particle filters [24] on each iteration: several rendering passes followed by a data fetch from the GPU would be needed per particle. Fortunately, the technique described in the previous subsection can be used to process particles in batches.

6 Experiments and Evaluation

We developed an IPU based on the Raspberry Pi running the described algorithms, and evaluated them through

Figure 10 Rendered points are assigned to histogram buckets, and rasterised into fragments to increment their values.



experiments conducted on 2 CV applications dealing with detection and tracking. The two applications operated in dissimilar environments under distinct assumptions, and dealt with different types of targets (inanimate objects versus humans). The experiments were performed both on synthetic video streams aswell as ones from a multi camera setup in a real environment. Synthetic data was used to facilitate zero error ground truth for evaluating the accuracy of the tracking. Poisson noise was added to synthetic data in order to recreate camera noise. A snapshot from the synthetic and real world experimental setup for the two applications is illustrated in Fig. 11.

The algorithms in the test applications were based on the work in [25] and were effectively re-engineered and optimised for the IPU architecture. Their adapted implementations, simplified to adjust them for our use cases, are briefly discussed in this section.

Trackers in both test applications are built around similar colour-based particle filters. However, while in the first simple smart camera application 2D positions and scales of targets were tracked, the second test application's tracker handled 3D target locations. Still, the particle filter implementations used in the two cases were modeled on the same systems, described in [26] and [24], and are discussed together in the following paragraphs.

To generate a new set of particles — representing hypotheses about a target's location — in every iteration of tracking, each particle filter used by the 2 test applications employed a simple prediction model which added white Gaussian noise to a re-sampled set of particles from the previous iteration. Next, histograms of the new particles were obtained by executing the histogram generation algorithm described in Section 5.6. Each particle's histogram was then compared with the reference histogram of the tracked target, and their Bhattacharyya distance B [27] was computed. Finally, particle likelihoods were evaluated under a Gaussian model in the overall residual:

$$P(s_t^i) = exp(-\lambda B^2)$$

where s_t^i is the ith particle in the tth iteration, and λ corresponds to the covariance.

Once particle likelihoods were available, the particle filter assigned weights to the particles based on their likelihoods. The estimated target pose was then computed as the weighted average of all particles:

$$\bar{s}_t = \sum_i \omega_t^i s_t^i$$

where ω_t^i is the weight of the ith particle in the tth iteration.

After every iteration, the particle set was deterministically re-sampled to maintain a good distribution of particles. The initial set of particles was derived from the target's detection pose.

The following subsection provide discussion on the two applications and their evaluation results:

6.1 Single IPU Based Detection and Tracking

This application involved a standalone camera based IPU running tracking and detection algorithms. It was implemented to determine whether a single SBC could process camera images, and run high-level algorithms on the extracted data simultaneously. The detector used by the simple smart camera application performed contour detection on a colour-thresholded image. If a large enough contour was found, the detector instantiated a tracker for it and switched the application into tracking state.

The data set used to evaluate was an animation of a ball moving against a vibrant, static background. It was programmatically generated and consisted of 660 frames sized 640px×480px. As the data set was computer generated, zero-error ground truth for it was known.

A video showing the application being executed on a live camera is attached and also available online http://youtu.be/kwmSKxxhAsg. The evaluation video showing the application running on the benchmarking sequence is attached and also available online http://youtu.be/CZyW8a5zezw.

The application was executed on the animation data set and the tracked target positions it generated were recorded.

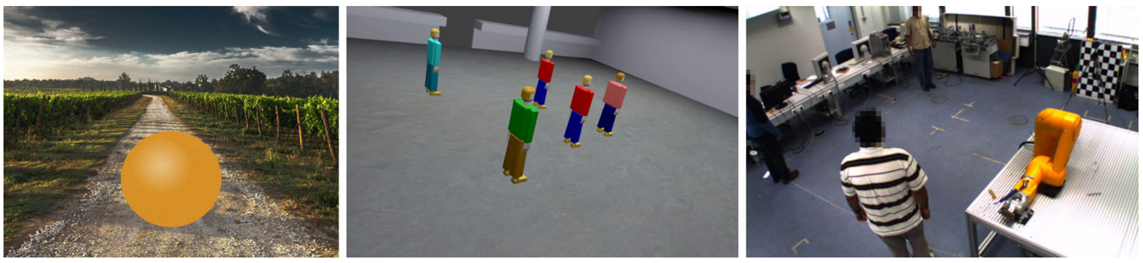

(a) (b) (c)

Figure 11 Overview of developed applications: **a** Tracking a ball in 2D. Tracking multiple people in 3D using 4 cameras over **b** synthetic and **c** real sequences.

Table 1 Mean and mean absolute tracking error observed when the application was executed on the animation data set.

Target property	Mean error (standard deviation)	Mean absolute error (standard deviation)
X coordinate	0.4px (5.6px)	4.4px (3.4px)
Y coordinate	1.3px (4.8px)	3.9px (3.1px)
Scale	−0.002 (0.027)	0.021 (0.017)

Next, they were compared to the ground truth to calculate position and scale tracking errors. The mean and mean absolute tracking error observed are listed in Table 1.

The size of the ball tracked by the application was 128px×122px. Taking that into consideration, the mean absolute tracking error along the X and the Y axes was less than 5% of the target size. Performance measurements showed that the IPU took less than 100ms (usually around 50ms) to process a single frame.

6.2 Multiple Human Tracking Using Distributed IPUs

This application deals with detection and tracking of humans using multiple cameras. It involves 4 IPUs connected to a master node that requests high-level image information from them and applies detection and tracking algorithms to it. This scenario was designed to test how well SBC-based IPUs perform in distributed environments, where they act as data pre-processing modules for external processing nodes. The real environment consisted of 5x5 meter area observed using a 4 camera setup. Within this area, 3 humans move in close proximity to each other. The animated sequence was generated using Blender[2] modeling an environment with 5 targets from the perspective of 4 cameras located in different corners of the lab. It consisted of 900 frames (sized 752px×480px) per camera. As the data set was computer generated, zero-error ground truth for it was known.

This application performed detection and tracking in 3D. The detector included more logic. It was connected to 4 IPUs, and asked them to execute—on the same scan area—the foreground coverage algorithm described earlier. It then iterated through all cells of the scan area, and if any had foreground coverage above 70% on at least 3 IPUs, it was considered a potential location of a human target. Next, the detector filtered out those potential targets which were occluded on some of the IPUs that reported high foreground coverage for them. Finally, a tracker was instantiated for each target that passed that test. A master node then combined and ran detection and tracking algorithms on high-level image information received from the 4 IPUs to decide whether any human-sized objects were visible, locate them, and track their changing positions in real time as long as they stayed in view.

A video showing the application being executed in the real environment is attached and can be found online http://youtu.be/4ZxP4nR1r8I. In addition, a video showing results from the animated sequence used for benchmarking is attached and available online http://youtu.be/NSTznYNcr1Y.

Contrary to the previous one this application was evaluated with greater detail. The individual aspects are discussed below:

– Data Sets: Two data sets, originating from [28], were used to evaluate the application. The first one was a video sequence generated using Blender, modeling a lab environment with 5 targets from the perspective of 4 cameras located in different corners of the lab. It consisted of 900 frames (sized 752px×480px) per camera. As the data set was computer generated, zero-error ground truth for it was known. A video showing the application being executed with the data set is available on line [29]. The second data set was a video recording of a real-world lab environment from the perspective of 4 cameras located in different corners of the lab. It consisted of 2700 frames (sized 752px×480px) per camera. 3 targets appeared in the data set, however no ground truth was available to evaluate how well they were tracked.
– Tracking Accuracy: First of all, the application was run with different configurations of the particle filter to decide how many particles to use for target tracking with the first data set. In each run the the length of every target's first path tracked after detection was recorded. Figure 12 shows how the observed lengths change in relation to the number of particles used.

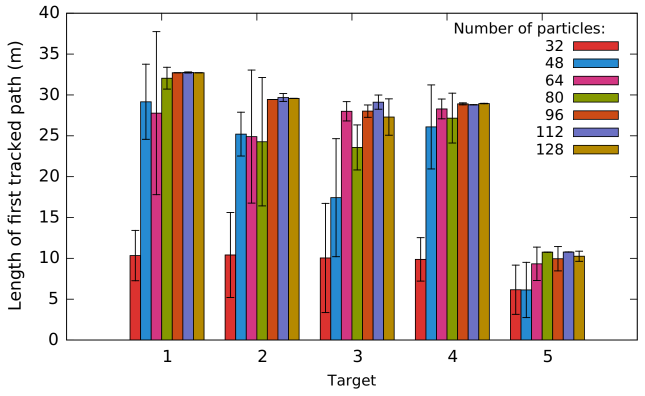


Figure 12 Mean length of the path tracked between a target was detected for the first time and lost, depending on the target and the number of particles used for tracking. To obtain these results the application was executed on the first data set 5 times for each number of particles.

[2]http://www.blender.org

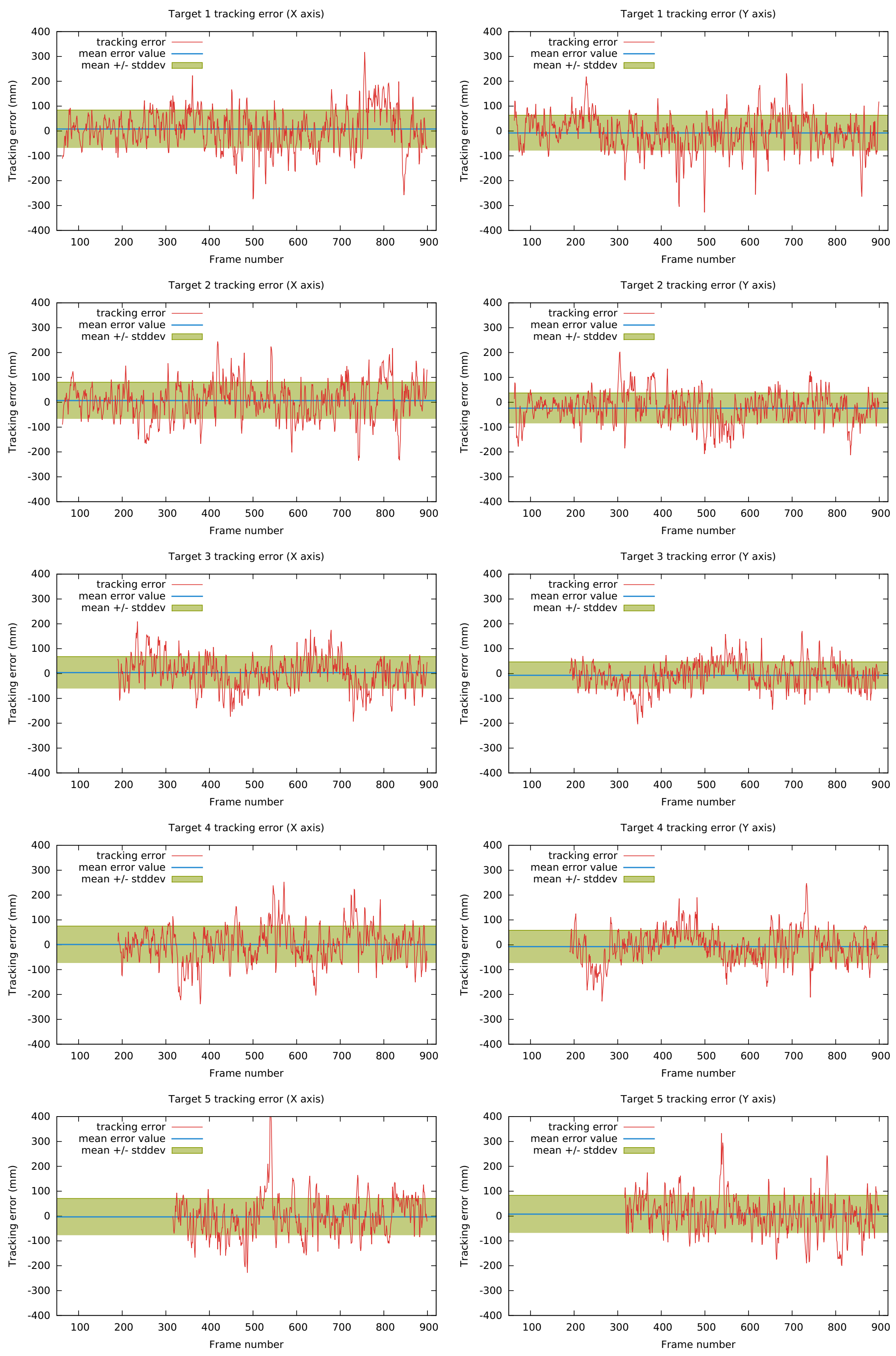


Figure 13 Tracking error on the x and the y axis for the 5 targets from the first data set. The mean error and its corresponding standard deviation are also marked on all plots.

Table 2 Mean and mean absolute tracking error observed when the application was executed on the first data set.

Target	Coordinate	Mean error (standard deviation)	Mean absolute error (standard deviation)
1	X	8 mm (76 mm)	59 mm (48 mm)
	Y	−8 mm (71 mm)	54 mm (46 mm)
2	X	7 mm (74 mm)	58 mm (46 mm)
	Y	−24 mm (61 mm)	50 mm (42 mm)
3	X	4 mm (64 mm)	50 mm (40 mm)
	Y	−7 mm (54 mm)	43 mm (34 mm)
4	X	1 mm (74 mm)	57 mm (47 mm)
	Y	−7 mm (66 mm)	51 mm (42 mm)
5	X	−1 mm (79 mm)	60 mm (52 mm)
	Y	8 mm (76 mm)	57 mm (50 mm)

The collected data indicates that when a very low number of particles (e.g. 32) are used for tracking, targets are invariably lost rather quickly. Medium particle counts (such as 48, 64, 80) resulted in tracked paths that were longer, but still varied considerably due to targets being prematurely lost. Only the last three particle counts tested (96, 112, 128) consistently yielded long paths lasting until the end of the data set. Consequently, 96 particles (3 batches for the purpose of histogram generation) were used for tracking as that number provided a good trade-off between accuracy and performance.

Next, the application was executed on the first data set and the tracked target positions it generated were recorded. They were then compared to the ground truth to calculate position tracking errors. The 5 targets were detected in frames 63, 63, 190, 190, and 309. All of them were tracked until the end of the data set.

Figure 12 shows how close tracked target positions follow ground truth locations. The difference between the two was calculated and plotted on Fig. 13 to visualise the tracking error. The mean and mean absolute tracking error observed when the application was executed on the first data set are also listed in Table 2.

The observed tracking errors are in line with the results reported in [28]. The standard deviation of the tracking errors remains below 100mm, and the mean absolute tracking error is less than 60mm.

– Performance: While the application was being executed on both data sets, the performance of the algorithm implementations it used was measured and recorded. Figures 14 and 15 illustrate how the performance of histogram generation changes while running detection and tracking throughout the first and the second data set, respectively. The plots are split into 2 areas to visualise

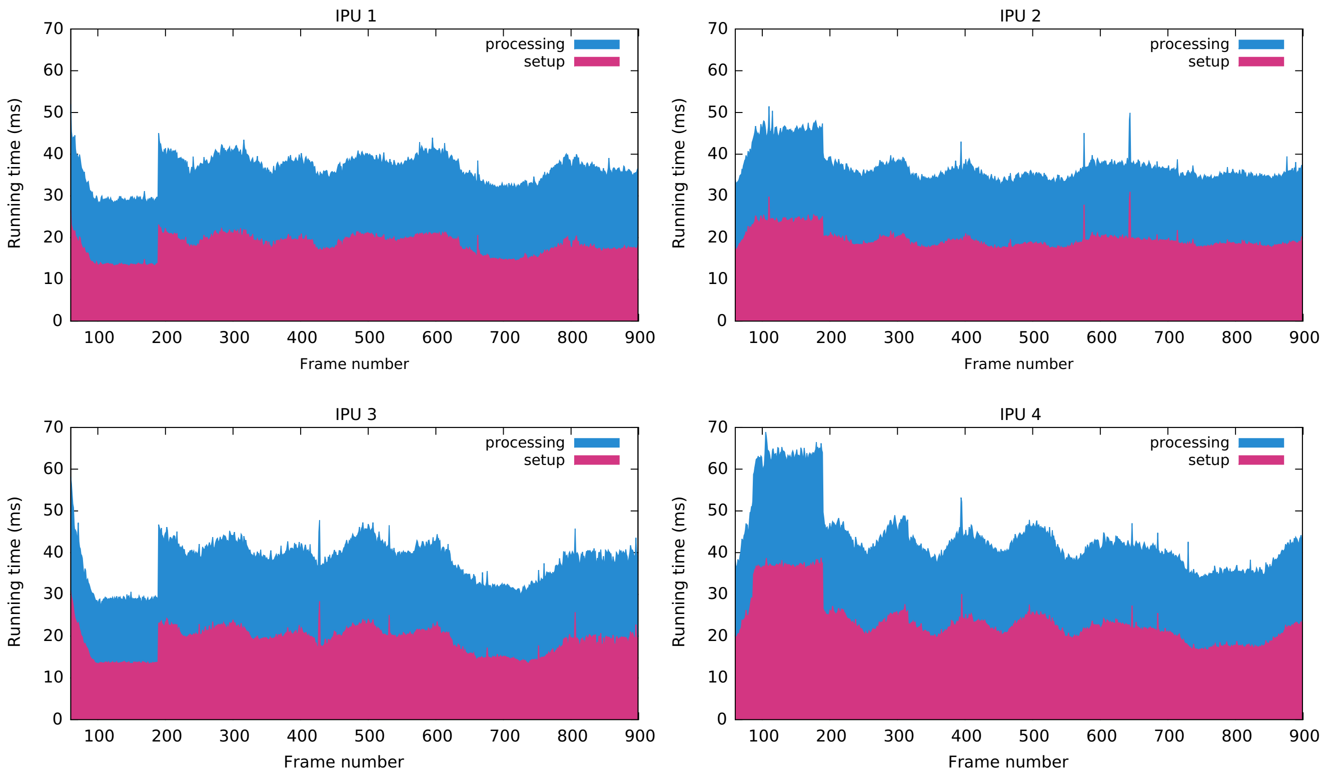


Figure 14 Time needed to generate histograms for 96 particles used to track targets throughout the first data set. The overall value for each frame consists of the setup and the processing time.

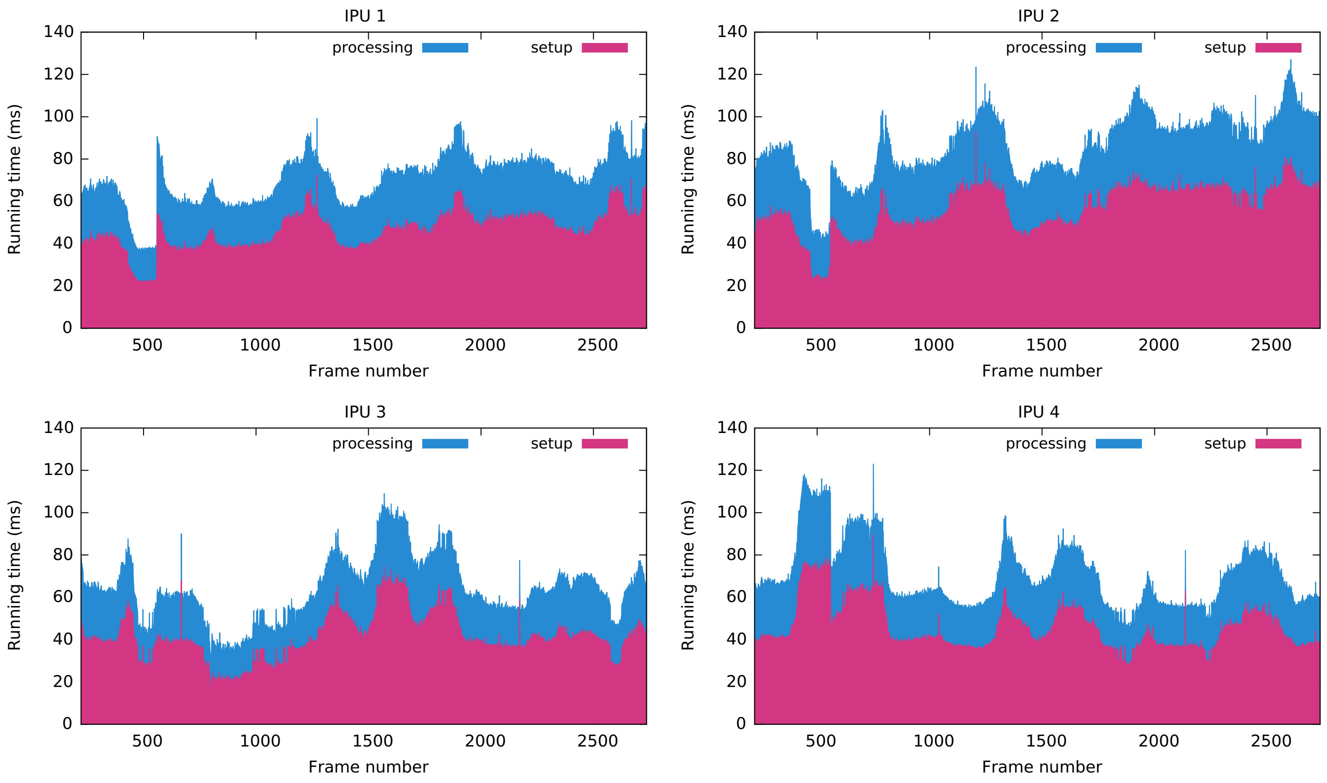


Figure 15 Time needed to generate histograms for 96 particles used to track targets throughout the second data set. The overall value for each frame consists of the setup and the processing time.

the time each of the 2 main stages of histogram generation takes. In the setup phase points covering all particles are created and uploaded onto the GPU. At the processing stage the points are scattered to generate histograms on the GPU.

The time needed to generate histograms for a set of particles varies due to the fact that targets change their distance from the IPU's cameras, which in turn causes projected particle image sizes to fluctuate as tracking progresses. Bigger particle images require more points to be rendered, which increases the setup and the processing times. The sudden rises and drops on the plots can be explained by new targets being detected: this happened in frames 63, 190, and 309 of the first data set; and frames 185, 554, and 986 of the second data set.

The times observed with the first data set are lower than the times for the second one because in the former video cameras are located further away from the scan area, resulting in smaller projected target and particle sizes.

The observed performance of other algorithms used by the application, as well as the time required to upload frame data onto the GPU, are plotted on Fig. 16.

Foreground coverage computation takes less time with the second data set due to the fact that it requires a smaller scan area: 12×11 cells (3.6m×3.3m) versus 20×20 cells (6m×6m) for the first data set.

The figures included in this section indicate that the 2 most time-consuming algorithms are foreground coverage computation and histogram generation. The former was executed at set intervals: every second for the first data set, and every 2s for the second data set. As a result, its relatively high running time was amortised over several frames, and did not significantly affect the overall performance of the application.

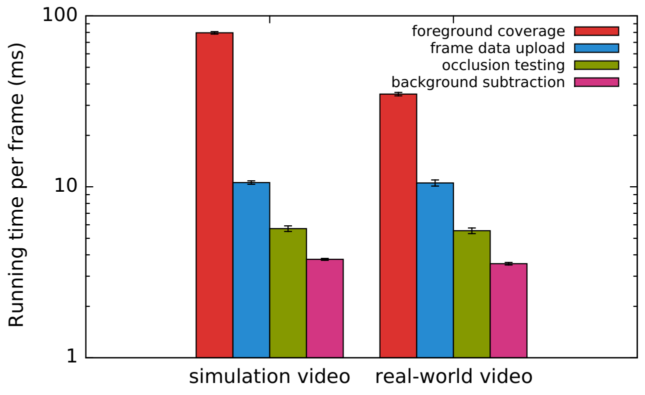


Figure 16 Average time spent executing the remaining algorithms used by the second test application on both data sets.

In contrast, histogram generation was executed every frame once the first target was detected and the tracking phase started. Its mean running time was 39ms with the first data set, and 72ms with the second one. Even after adding the frame data upload and background subtraction times, the IPU took on average less than 100ms to process a single frame.

7 Software

The software developed as a part of this project has be released open source and hosted on github. The source code and setup documentation can be found under https://github.com/cognitivesystems/smartcamera.

8 Conclusions

The implemented applications exhibited high detection and tracking accuracy. The observed tracking errors were as low as those reported by other sources for similar data sets. This is a strong indicator that the algorithms implemented to run on the Raspberry Pi-based IPUs produced good quality data and could be used in real-world scenarios.

Performance of the implemented algorithms was also determined to be satisfactory for the most part. The majority of them took only a negligible amount of time to execute on frames of standard resolutions, while others (e.g. histogram generation and foreground coverage computation) could have their running times reduced by carefully adjusting such application parameters as the size of the scan area or the number of particles used for tracking.

All in all, there is a strong case for the feasibility of SBC-based IPUs from the ease of programming, performance, and robustness standpoints. When their price and portability are taken into consideration, they become an attractive choice that is likely to become popular in the near future.

Acknowledgements This work was financially supported by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

References

1. Kimura, S., Miyasaka, A., Funase, R., Sawada, H., Sakamoto, N., & Miyashita, N. (2011). High-performance image acquisition & processing unit fabricated using COTS technologies. *IEEE Aerospace and Electronic Systems Magazine*, *26*, 19–25.
2. Choy, C.S., Chan, W.K., & Lam, W. (1992). An image processing unit using an ICT chip set. In *TENCON '92. Technology enabling tomorrow: computers, communications and automation towards the 21st century. 1992 IEEE region 10 international conference*, (Vol. 2 pp. 1003–1007).
3. Freescale Semiconductor (2009). Image processing unit v3 (IPUV3) library.
4. Shi, Y., & Real, F.D. (2010). In *Smart cameras: fundamentals and classification*. US: Springer.
5. Holzer, M., Schumacher, F., Greiner, T., & Rosenstiel, W. (2012). Optimized hardware architecture of a smart camera with novel cyclic image line storage structures for morphological raster scan image processing. In *IEEE International conference on emerging signal processing applications (ESPA), 2012* (pp. 83–86).
6. Chan, W.K., & Chien, S.Y. (2006). High performance low cost video analysis core for smart camera chips in distributed surveillance network. In *IEEE 8th workshop on multimedia signal processing, 2006* (pp. 170–175).
7. Casares, M., & Velipasalar, S. (2010). An adaptive method for energy-efficiency in battery-powered embedded smart cameras. In *Proceedings of the fourth ACM/IEEE international conference on distributed smart cameras. ICDSC '10* (pp. 167–174). New York, NY, USA: ACM.
8. Cheng, K.T., Yang, X., & Wang, Y.C. (2013). Performance optimization of vision apps on mobile application processor. In *20th international conference on systems, signals and image processing (IWSSIP), 2013* (pp. 187–191).
9. Roudel, N., Berry, F., Serot, J., & Eck, L. (2010). A new high-level methodology for programming FPGA-based smart camera. In *13th euromicro conference on digital system design: architectures, methods and tools (DSD), 2010* (pp. 573–578).
10. Neves, R., & Matos, A. (2013). Raspberry pi based stereo vision for small size ASVs. In *Oceans - San Diego, 2013* (pp. 1–6).
11. Reboucas, R.A., Eller, Q.d.C., Habermann, M., & Shiguemori, E.H. (2013). Embedded system for visual odometry and localization of moving objects in images acquired by unmanned aerial vehicles. In *III Brazilian symposium on computing systems engineering (SBESC), 2013* (pp. 35–40).
12. Hofmann, R., Seichter, H., & Reitmayr, G. (2012). A GPGPU accelerated descriptor for mobile devices. In *IEEE international symposium on mixed and augmented reality (ISMAR), 2012* (pp. 289–290).
13. Reinisch, G., Arth, C., & Schmalstieg, D. (2013). Panoramic mapping on a mobile phone GPU. In *IEEE international symposium on mixed and augmented reality (ISMAR), 2013* (pp. 291–292).
14. Singhal, N., Park, I.K., & Cho, S. (2010). Implementation and optimization of image processing algorithms on handheld GPU. In *17th IEEE international conference on image processing (ICIP), 2010* (pp. 4481–4484).
15. Fung, J., & Mann, S. (2004). Computer vision signal processing on graphics processing units. In *IEEE international conference on acoustics, speech, and signal processing, 2004. Proceedings. (ICASSP '04)*, (Vol. 5 pp. 93–96).
16. Jargstorff, F. (2004). 27. In *A framework for image processing.* Addison-Wesley Professional (pp. 445–467).
17. Fung, J. (2005). 40. In *Computer vision on the GPU.* Addison-Wesley Professional (pp. 649–666).
18. Benezeth, Y., Jodoin, P.M., Emile, B., Laurent, H., & Rosenberger, C. (2010). Comparative study of background subtraction algorithms. *Journal of Electronic Imaging*, *19*, 033003.
19. Strengert, M., Kraus, M., & Ertl, T. (2006). Pyramid methods in GPU-based image processing. *Proceedings Vision, Modeling, and Visualization*, *2006*, 169–176.
20. Horn, D. (2005). 36. In *Stream reduction operations for GPGPU applications.* Addison-Wesley Professional (pp. 573–589).
21. Nugteren, C., van den Braak, G.J., Corporaal, H., & Mesman, B. (2011). High performance predictable histogramming on GPUs: Exploring and evaluating algorithm trade-offs. In *Proceedings of*

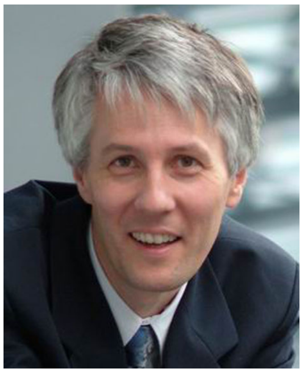
the fourth workshop on general purpose processing on graphics processing units. GPGPU-4 (pp. 1:1–1:8). New York: ACM.
22. Fluck, O., Aharon, S., Cremers, D., & Rousson, M. (2006). GPU histogram computation. In *ACM SIGGRAPH 2006 Research posters. SIGGRAPH '06*. New York, NY, USA: ACM.
23. Scheuermann, T., & Hensley, J. (2007). Efficient histogram generation using scattering on GPUs. In *Proceedings of the 2007 symposium on interactive 3d graphics and games. i3d '07* (pp. 33–37). New York: ACM.
24. Pérez, P., Hue, C., Vermaak, J., & Gangnet, M. (2002). Color-based probabilistic tracking. In Heyden, A., Sparr, G., Nielsen, M., & Johansen, P. (Eds.) *Computer vision — ECCV 2002. Volume 2350 of lecture notes in computer science* (pp. 661–675). Berlin: Springer.
25. Nair, S., Panin, G., Wojtczyk, M., Lenz, C., Friedlhuber, T., & Knoll, A. (2008). A multi-camera person tracking system for robotic applications in virtual reality TV studio. In *IEEE/RSJ International conference on intelligent robots and systems, 2008. IROS 2008* (pp. 3990–3996).
26. Nummiaro, K., Koller-Meier, E., & Van Gool, L. (2002). Object tracking with an adaptive color-based particle filter. In Van Gool, L. (Ed.) *Pattern recognition. Volume 2449 of Lecture Notes in Computer Science* (pp. 353–360). Berlin: Springer.
27. Bhattacharyya, A. (1946). On a measure of divergence between two multinomial populations. *Sankhyä: The Indian Journal of Statistics*, 401–406.
28. Nair, S. (2012). Visual tracking of multiple humans with machine learning based robustness enhancement applied to real-world robotic systems. Dissertation, Technische Universität München, München.
29. Nair, S., & Grunau, A. (2014). Human tracking simulation. https://www.youtube.com/watch?v=NSTznYNcr1Y.

Suraj Nair received his PhD in Robotics from the Technische Universität München (TUM), Germany. His thesis was focused in the area of computer vision applied to real world industrial robotic systems. During his research career, he has been involved in numerous research projects most of which were industry driven. His activities have been closely connected to industry in Germany and worldwide. Suraj served as the project lead from TUM-fortiss in the prestigious EU FP7 project SMErobotics. He has also been active in the area of medical robotics and was one of the founding members of the iRAM!S project. He currently serves as a Principal Investigator at TUMCREATE, Singapore. He also serves as a Co-Principal Investigator for the SERC Industrial Robotics Programme. His primary research interests are computer vision, artificial intelligence, autonomous vehicles and robotics.

Nikhil Somani is a Ph.D. student at the Technische Universität München (TUM) since 2013. He is currently working at the Cognitive Systems and Robotics group at TUM-CREATE in Singapore as the head of robotics system architecture. He received his Bachelor's degree from IIT Kharagpur, and Masters degree in Computer Science from TUM in 2011 and 2013 respectively. His research interests include computer vision, robot motion control, cognitive robotics and human-robot interaction.

Artur Grunau has been a software engineer at Microsoft since 2014. He currently works on database systems powering the Bing search engine. He received his Bachelor's degree from the Jagiellonian University and Master's degree in Computer Science from the Technische Universität München in 2010 and 2014, respectively. His professional interests include distributed systems, programming language design, and data analysis.

Emmanuel Dean-Leon (m) studied Mechatronics at the Center for Research and Advanced Studies of the National Polytechnic Institute (CINVESTAV-IPN) in Mexico, where he received his doctorate in 2006. He received the "Arturo Rosenblueth" Award in 2006 for the best PhD Theses. In 2009 he performed a postdoctoral research project at the Department of Computer Science, TUM. Since 2012, he has been involved in several EU projects related to advanced industrial robot applications. Since 2013 he has been a senior researcher at the Chair for Cognitive Systems in the Department of Electrical and Computer Engineering, TUM. His research interests include robotics, robot modeling and low level control design/implementation, sensor fusion, physical human-robot-interaction, and cognitive systems.



Alois Knoll received his diploma (M.Sc.) degree in Electrical/Communications Engineering from the University of Stuttgart and his PhD degree in Computer Science from the Technical University of Berlin. He served on the faculty of the computer science department of TU Berlin until 1993, when he qualified for teaching computer science at a university (habilitation). He then joined the Technical Faculty of the University of Bielefeld, where he was a full professor and the director of the research group Technical Informatics until 2001. Between May 2001 and April 2004 he was a member of the board of directors of the Fraunhofer-Institute for Autonomous Intelligent Systems. At AIS he was head of the research group "Robotics Construction Kits", dedicated to research and development in the area of educational robotics. Since autumn 2001 he has been a professor of Computer Science at the Computer Science Department of the Technische Universitaet Muenchen. He is also on the board of directors of the Central Institute of Medical Technology at TUM (IMETUM Garching). Between April 2004 and March 2006 he was Executive Director of the Institute of Computer Science at TUM. His research interests include cognitive, medical and sensor-based robotics, multi-systems, data fusion, adaptive systems and multimedia information information.