Journal of Real-Time Image Processing (2023) 20:119
https://doi.org/10.1007/s11554-023-01373-w

RESEARCH

Frappe: fast fiducial detection on low cost hardware

Simon Jones[1] · **Sabine Hauert**[1]

Received: 26 July 2023 / Accepted: 27 September 2023 / Published online: 24 October 2023

Abstract
Square fiducial markers are widely used in robotics to easily obtain pose and other information about the world from camera images. Processing the images to extract the markers is usually performed centrally with standard libraries but the code is typically aimed at PC-level hardware. Platforms with constrained processing power have difficulty handling multiple camera streams at real-time refresh rates. We introduce the **Frappe** (Fiducial Recognition Accelerated with Parallel Processing Elements) algorithm for detecting and decoding the popular ArUco tags. Designed to be implemented on the low cost hardware of the Raspberry Pi Zero, we show tag detection and decoding on images of 640×480 resolution exceeding 60 Hz, five times faster than the standard ArUco library, while maintaining similar detection performance and using much less energy. Using Frappe, we demonstrate improved real-world performance on a visual navigation task with our DOTS robot.

Keywords Image processing · Fiducial tags · Robot vision · Embedded processing · GPU acceleration

1 Introduction

Scaling up robot numbers in real-world environments requires both lowering the cost of robots, and improving their ability to perceive and interact with the world. One approach uses cheap vision hardware and augments the environment with markers. Square fiducial markers consisting of a grid with a binary pattern are widely used in robotics vision systems as a way of providing pose and navigation information from a camera image feed without the complexity and processing cost of full image comprehension techniques such as Visual SLAM. The popular ArUco library is widely used, but the processing cost is still significant in resource constrained robot systems, limiting the resolution and update rate that is possible, hindering the performance of real-time robot navigation.

The Raspberry Pi series of educational Single Board Computers (SBCs) has enabled many projects needing a small, cheap computer running Linux. Well supported, they have a camera interface supporting several models of camera. A Raspberry Pi Zero and OV5241 camera module can be purchased for around £16, providing 1080p60 streaming video. What is not widely utilised is the surprisingly capable Graphics Processing Unit (GPU) that all Pi models have, with around 24 GFLOPs processing power.

We design an image processing algorithm, called **Frappe**, Fiducial Recognition Accelerated with Parallel Processing Elements, to use the Raspberry Pi (RPi) Zero GPU for as much processing as possible. As proof-of-concept, we implement Frappe on our swarm of DOTS [1] robots designed for intralogistics applications. By re-engineering the visual navigation system of the DOTS, enabling higher detection frame-rates and resolutions than were previously possible, we enhance performance at a visual navigation task.

We make available an implementation of the algorithm and a complete Docker-based development environment[1]. This brings together the required specialised toolchains and provides a virtual environment for compiling GPU applications targeting the Raspberry Pi Zero. We provide this framework for others to make use of this underutilised processing power for visual processing and other edge processing applications.

This paper is organised as follows; Section 2 covers background and related material, Sect. 3 details the algorithm and its implementation, Sect. 4 compares the performance

✉ Simon Jones
simon2.jones@bristol.ac.uk

Sabine Hauert
sabine.hauert@bristol.ac.uk

[1] Department of Engineering Mathematics, University of Bristol, Bristol, UK

[1] https://bitbucket.org/simonj23/frappe/src/master/.

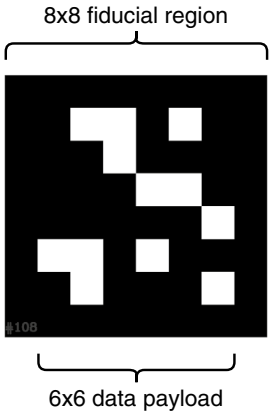

Fig. 1 Example of an ArUco fiducial marker from the `ARUCO_MIP_36h12` dictionary, showing the full 8x8 region, with the outer cells always black, and the inner 6x6 36 bit data payload

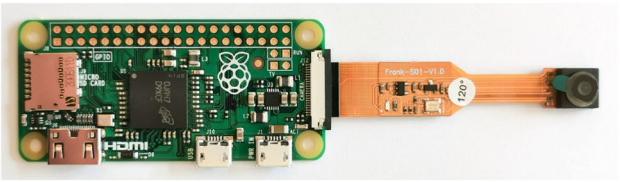

Fig. 2 Raspberry Pi Zero with attached camera, costing around £16 and capable of streaming up to 1080p60 video

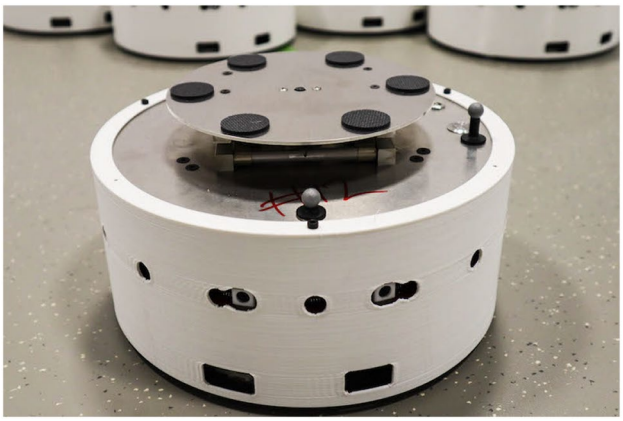

Fig. 3 DOTS robot, fast moving and low cost with 360° vision, enabling research into swarm intralogistics

of Frappe and ArUco on Raspberry Pi Zero hardware, before using Frappe in a larger robot system for enhanced performance, and Sect. 5 concludes the paper.

2 Background

Fiducial markers or tags are visually distinct objects placed in the environment to convey information or position or both. In robotics, what is often desired is to extract pose and position from a camera feed, in this case the fiducial must convey both accurate position and unique identity. The most common form is a monochrome square region with an internal bit pattern, an early system was ARToolKit [2], widely used examples include AprilTag [3, 4], ARTag [5], and ArUco, with [6] showing generation of dictionaries with near-optimal intermarker distance, and [7] accelerating detection. Circular forms are also common, such as InterSense [8], STag [9], and CCTag [10]. CCTag is also designed to be resistant to occlusion and motion blur. For widely used square tags such as ArUco, AprilTag, and ARTag, there has been work on blur resistant decoders with conventional [11] and machine learning approaches [12, 13]. See [14] for a recent review and examination of the comparative detection performance and resilience of some different tag systems. Although not directly comparable with our results, they show detection rates of 95% for ArUco in their test data. They don't directly report processing time, but do say that 640x480 detection at 20 Hz on a Raspberry Pi 3 was possible for ARTag and ArUco, but AprilTag was too computationally intensive. Regarding the speed of various detectors, [4] report AprilTag2 at 78 ms for a 640x480 image on an Intel Xeon E5-2640, [7] report ArUco at 0.9 ms for 640x480 on an Intel Core i7-4700HQ.

This work specifically addresses accelerating ArUco tag detection on low cost hardware, due to our existing systems and software using this tag. Figure 1 shows an example ArUco fiducial from the standard dictionary `ARUCO_MIP_36h12`, generated as described in [6]. It shows the 8x8 region of a marker, consisting of an outer perimeter of always black cells, with an inner 6x6 region containing the data payload. Each of the 250 unique symbols in the dictionary have a minimum Hamming distance of 12 from all other symbols, meaning that up to 6 erroneous bits out of the 36 can be corrected (Fig. 2).

Our DOTS swarm robots [1], shown in Fig. 3, are designed to enable research into swarm intralogistics. They are low cost, capable of fast agile movement, able to carry loads, and have a ROS2-based control system running on RockPi 4 SBC. 250 mm in diameter, they are equipped with four cameras for 360° vision. Although recent trends in robot vision have moved towards high speed event cameras [15] and deep learning, the cost and computational requirements are still considerable, and way beyond our price point—we attempt to maximise the capabilities of very cheap commodity consumer electronics. Hence each camera is a low cost OV5241 module with a wide-angle lens attached to a Raspberry Pi Zero (Fig. 2) and streams video to the central RockPi 4 SBC, shown in Fig. 7. This architecture was chosen based on both cost, and the possibility of performing embedded image processing as described in this work.

Extracting real-time pose information from the camera feeds using techniques such as Visual Simultaneous Localisation and Mapping (SLAM) is computationally expensive, running e.g. ORB-SLAM [16] or LSD-SLAM [17] is beyond

the reach of the computational ability of the DOTS, so we use ArUco tags and library [7] for navigation and world comprehension, chosen as the fastest available library. Even so, processing four camera feeds on the central RockPi SBC necessitated limiting the resolution to 320×240 and frame rate to 15 Hz. In this work, we focus specifically on achieving a system capability of 640×480 pixels with a frame rate of 30 Hz over all four cameras, an eight-fold increase in the aggregate pixel processing rate, by delegating fiducial recognition to the RPi Zeros.

The Raspberry Pi [18] series are small SBCs based on Broadcom System-On-Chips (SoCs), initially aimed at education. They have become immensely popular with makers, in education and academia, with hobbyists, and also in industry, due to the low cost and good support, both from the Raspberry Pi Foundation and from the large user community [19–25] . The support and longevity has meant that a large ecosystem of peripherals and applications has grown up around them. There are more powerful SBCs, but they are often short-lived, with poor support from the manufacturers. The RPi Zero, shown in Fig. 2, is of particular interest to makers and roboticists because of its small form factor and low cost.

The Broadcom SoC is typical of many that were aimed at the mobile phone market, in that it includes a camera image processing pipeline, and a OpenGLES2-compliant GPU. All the Pi series except for the RPi 4 use the same processing block; the VideoCore IV, or VC4. This contains two major subsystems, the Vector Processing Unit (VPU); a dual core vector processor for running system code and handling 2D image and video data, and the GPU; 12 parallel Quad Processing Units (QPU) and support blocks for handling 3D rendering. Attached to the VC4 block are one or more ARM CPUs, in the case of the RPi Zero a single ARM1176 core. A simplified view of the architecture is shown in Fig. 4.

Although some documentation was released by Broadcom [26], this only covered the QPUs of the 3D core. Since then, much work has been done to reverse engineer details of the VPU instruction set [27], and develop tools and applications; a QPU assembler [28], open source VPU firmware [29], a port of the GCC compiler to the VPU [30], and optimised FFT library GPU_LIB on the QPUs [31], a QPU programming language QPULib [32], use of the QPUs for basic image processing [33], and other information about the hardware architecture [34, 35].

Despite this, there are few works within the formal literature making use of this processing power. The language QPULib is used in [36] implement a simple convolution and demonstrate 27x speedup and 35x less energy usage compared to a CPU-only implementation, and in [37] to implement part of a vision algorithm in the QPUs. The performance trade-offs of running FFTs on the QPUs or CPUs of a Raspberry Pi 3B for a cross-correlation task are investigated in [38]. We can find no works using the VPU at all, which motivates the design and implementation of Frappe as a complete demonstration application, and the making of this, and a collated set of development tools, freely available.

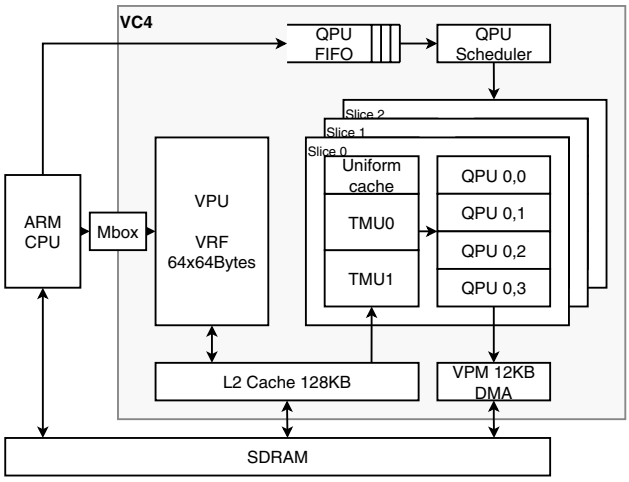

Fig. 4 Raspberry Pi CPU and VC4 subsystems. The ARM CPU on the left communicates with the VPU via the *mailbox* interface and with the QPUs via the QPU FIFO. The VPU and the 12 QPUs operate autonomously and in parallel with the CPU

3 Materials and methods

The Frappe algorithm is relatively conventional in image processing terms, but each part has been chosen such that they can be optimised using the available processing blocks of the VideoCore processor, if possible. The principles underlying the optimisation are these: CPU processing is relatively slow, and should be used only where operations cannot be performed on the QPUs or the VPU, and memory traffic should be minimised. We favour constant-time operations wherever possible.

We now detail the GPU hardware and how it can be used, then describe the algorithm and the specifics of implementation on the RPi Zero.

3.1 GPU hardware

The VPU is focussed mainly towards general purpose and 2D image processing. It consists of two scalar cores, a shared vector processing core, and two vector register

files. In normal operation of the RPi, it is responsible for booting the system from firmware, then starting the Linux kernel on the ARM core. It is then responsible for providing services such as OpenGLES2, video codecs, camera, and video composition. There is a documented[2] *mailbox* interface from the ARM CPU to the VPU firmware which includes a method to directly execute user code on the VPU.

The main power of the VPU comes from its vector core and register files. Scalar and vector code can be freely intermixed, the single vector core being transparently shared by the two scalar cores. The vector core is a 16 lane SIMD, with each lane being 32 bits, giving 16 operations per clock. Operations are primarily integer. The Vector Register Files (VRF) are 64×64 bytes and can be accessed in a flexible 2D fashion, with both horizontal and vertical slices of 8, 16, and 32 bit words being unpacked and packed. Data can be streamed into and out of the VRF at very high bandwidths, reaching 70% of peak theoretical from SDRAM.[3]

The QPUs are focussed mainly on providing 3D graphics. Each of the 12 QPUs is architecturally a 16 lane (4 lane physical) dual-issue SIMD, with each lane 32 bits wide, with 64 registers, supporting 32 bit floating point and many pack and unpack operand modes to facilitate e.g. 8-bit integer to 32 bit floating point conversion. The QPUs are organised as three slices of four, with each slice sharing some special purpose hardware. Maximum parallelism across the QPUs is thus 96 operations per clock. Memory access on the QPUs is suited to their intended purpose as GPU processing engines. There are two Texture Memory Units (TMUs) per slice, shared by four QPUs. These provide read-only access to 2D texture buffers, with pixel interpolation and format conversion. They also allow direct memory access of 16 arbitrary addresses per request from a QPU, with up to four requests per QPU allowed in flight at any time. The Vertex Pipe Memory (VPM), shared by all QPUs, is a 12 KByte block of memory that has Direct Memory Access (DMA) engines to read and write main memory. The QPU can access sequential rows or columns of the VPM, but has no direct access to main memory. A stream of *uniforms*, 32 bit constants automatically fetched from memory, is available to read from within a QPU program invocation. Programs are executed by submitting program descriptors to the QPU *scheduler* 16 entry queue, which allocates programs to the next available QPU.

[2] https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface.

[3] LPDDR2 32 bit @450 MHz = 3.6 GB/s, VPU block read of 4 MBytes takes 1.65 ms = 2.5 GB/s.

1: $scaled \leftarrow$ ADAPTIVE_SCALE($input$)
2: $ec \leftarrow$ CANNY_SHI_TOMASI($scaled$) QPU1,2 28%
3: $mask \leftarrow$ FIND_EMPTY_TILES($scaled$) VPU1 5%
4: **repeat** raster scan ec CPU1 51%
5: $cand \leftarrow$ TRACE_CONTOUR(ec)
6: **if** GOOD_CANDIDATE(cand) **then**
7: $candidates += cand$
8: **end if**
9: **until** end of ec
10: CORNER_REFINE($candidates$) CPU2 8%
11: PERSPECTIVE_WARP($candidates$) QPU3 4%
12: BINARISE_DECODE($candidates$) VPU2 4%

Algorithm 1 Frappe

The RPi SoC has a unified memory architecture, meaning both the CPU and GPU can see the same memory without need to copy data from one to the other. In order to have parts of the algorithm executing on different functional blocks, we make use of kernel-supported[4] zero-copy VideoCore Shared Memory (VCSM) buffers. In this way, we can freely access a memory buffer from both CPU and GPU with zero copy cost, provided we pay attention to cache maintenance issues.

In order to best achieve parallelism, any algorithm should ensure that the 12 QPUs are each operating on different areas of data simultaneously, and that processing of data overlaps the storage of previous results and the loading of the next inputs.

3.2 Frappe algorithm

To detect square fiducials, we find contiguous borders that have exactly four corners, perspective correct, extract and binarise the information payload, then look up the value in a dictionary of valid fiducials. The process we use is outlined in Algorithm 1, along with functional unit and approximate time percentage per algorithm step, and is illustrated in Figs. 5, 6. Parameter values are shown in Table 1.

One of the insights of the ArUco paper [7] is that when operating on video, frames are often similar, so if a large fiducial was detected in one frame, later frames can be scaled down for performance increases from processing fewer pixels while still being able to detect similarly sized fiducials. In step 1, ADAPTIVE_SCALE, we look at the smallest fiducial edge length l_{min} detected in the previous frame and use that as the basis for setting the scale factor $r_{\text{scale}} = \beta / l_{\text{min}}$, where β is the target size of the scaled fiducial in pixels. To handle the situation where a smaller fiducial enters a frame already

[4] Up until kernel 5.4.83.

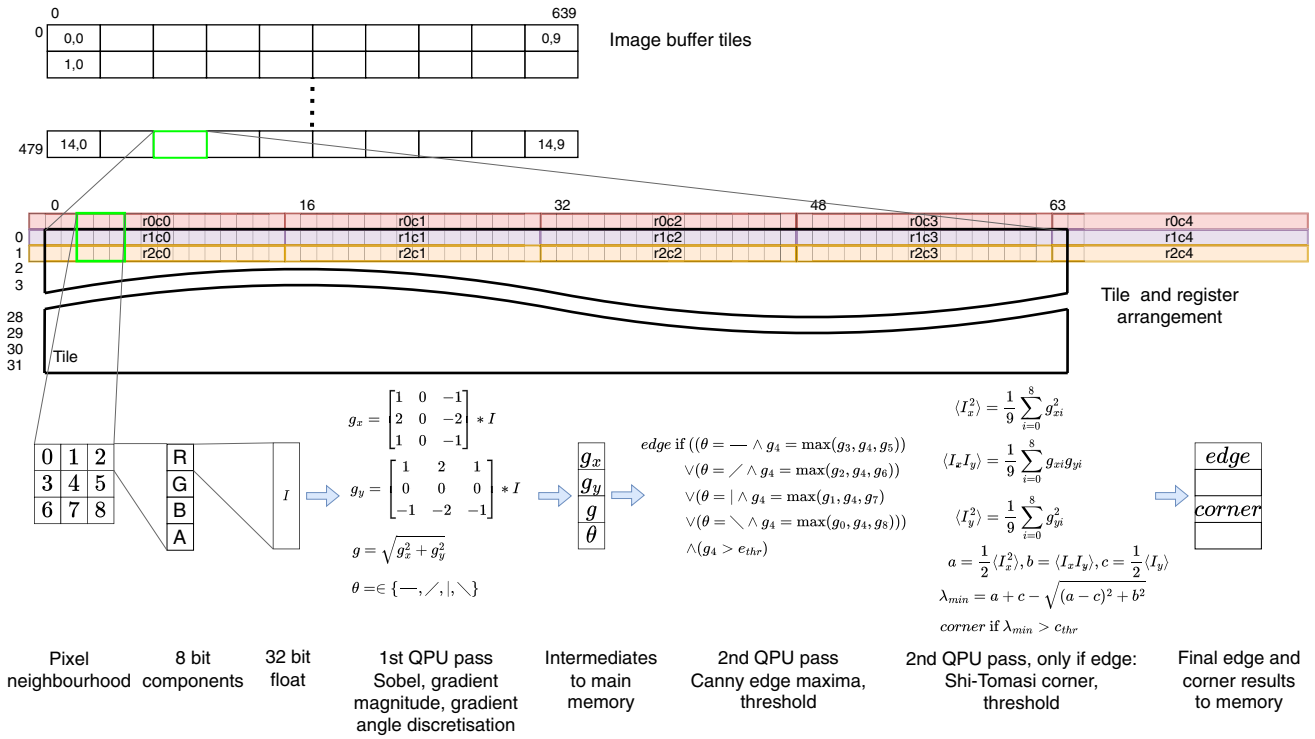

Fig. 5 Data layout for QPU processing passes. Data is organised as 64x32 pixel tiles, operated on independently and in parallel by the QPUs. Pass 1 fetches pixels, takes intensity information and produces gradients and discretised gradient angles. Pass 2 completes the Canny edge and Shi-Tomasi corner detector stages

containing a larger one, we set a maximum number of scaled down frames that can be processed contiguously $n_{\text{max\_scaled}}$.

In step 2, `CANNY_SHI_TOMASI`, we scale down the input image by r_{scale}, and then perform Canny [39] edge detection and Shi-Tomasi [40] corner detection. These operations are performed as two passes with the QPUs over the input image. Processing on the QPUs is organised as follows, illustrated in Fig. 5: Each QPU is allocated 256 bytes of the VPM memory, corresponding to 64 pixels. The output of a pass is organised as 64×32 pixel *tiles*, each of which is processed by a single QPU program invocation issued to the QPU scheduler in raster order. There are a total of 150 tiles for a full 640×480 image, with each invocation having associated uniforms specifying source and destination buffer addresses and strides, and, for scaling, fractional accumulation buffers and increments. The tile size of 64×32 was chosen empirically for best performance but represents the largest tile geometry possible while ensuring the tile data for the 12 QPUs fits within the 128 KByte L2 cache.

Processing is performed on 16 pixel wide parallel slices, with the pixels corresponding to the 3×3 neighbourhood region (66×34 pixels in total) being fetched from the TMU. The cost of fetches outside the boundary of a tile are hidden by the L2 cache. Each 16 pixel result is written to the VPM, after a complete tile row of 64 pixels has been calculated, the VPM DMA is triggered to write the data to main memory. TMU reads for future rows are arranged to take place during computation of current rows to minimises stalls.

Pass 1 fetches input pixels to the QPUs with locations chosen to achieve the required scaling factor, no interpolation is used, making scaling essentially a free operation. Within a 3×3 window on the fetched pixel data, we apply the Sobel [41] kernel in both x, and y directions to the image I to obtain gradients and gradient angle: $g_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * I$, $g_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * I$, $g = \sqrt{g_x^2 + g_y^2}$, $\theta = \frac{\pi}{4} \text{round}\left(\frac{4}{\pi} \tan^{-1} \frac{g_y}{g_x}\right)$. The gradient angle θ is discretised into four possible directions (horizontal, vertical, 45°, 135°) for the edge-thinning stage of the Canny algorithm. g_x,g_y,g, and θ are output to memory as 8-bit components of 32-bit pixels.

Pass 2 performs the Canny edge-thinning using the gradient angle to examine gradient magnitudes either side of candidate edge pixels, suppressing all but the maximum.

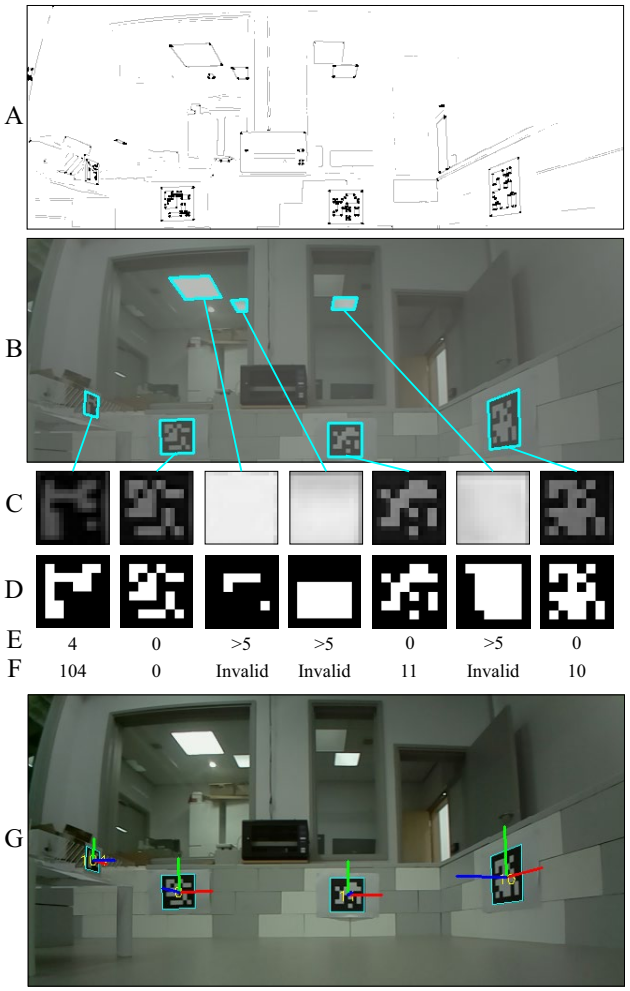

Fig. 6 Illustration of the stages of processing. **A** Input image after edge (grey) and corner (black) detection. **B** Good candidates after contour tracing. **C** Perspective warping into 16x16 pixel regions. **D** Binarisation for decoding. **E** Hamming distance from a valid symbol. **F** Decoded ID of symbol if valid. **G** Annotated input

Table 1 Frappe algorithm parameter values

Parameter	Symbol	Value
Edge threshold	e_{thr}	0.27
Corner threshold	c_{thr}	0.19
Maximum error correction	h_{thr}	5 bits
Scaled fiducial size	β	28 pixels
Maximum scaled frames	$n_{\max\_scaled}$	5

This is thresholded using a single value, e_{thr}, unlike the original Canny two threshold approach, which would have required another stage of processing. Provided this pixel is an edge, we form the structure matrix A for Shi-Tomasi corner detection $A = \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_x & I_y^2, \end{bmatrix}$ by multiplying gradients appropriately then performing a box filter on the 3×3 neighbourhood, then we solve for the the smallest eigenvalue $\lambda_{\min}$, which represents cornerness. Let $a = \frac{A_{11}}{2}, b = A_{12}, c = \frac{A_{22}}{2}, \lambda_{\min} = a + c - \sqrt{(a-c)^2 + b^2}$ This is also thresholded, with c_{thr}. Edge and corner values are output as 8-bit R and B components of 32-bit pixels, with the G, and A components left at zero.

Step 3 of the algorithm, `FIND_EMPTY_TILES`, uses the VPU to scan 16×16 blocks of pixels to see if they have any edges in them and generate a mask. Empty blocks can then be skipped by the contour tracing step. This is very well suited to the VPU, costing about 1 ms to scan a complete frame. The benefits are substantial, on a typical image we save more than 5 ms in contour tracing.

Steps 4–9 generates candidate quadrilaterals by tracing the contours in the image formed by edge pixels. This the most processing-intensive part and cannot be easily performed on the QPUs or the VPU. The image is scanned raster-style (step 4), skipping empty blocks, and edges are traced (step 5 `TRACE_CONTOUR`) with the Suzuki algorithm [42] using a modified version of the OpenCV `findContour` function. As each edge is followed we keep track of the pixels on it that qualify as corners, typically between 1 and 4 pixels on an edge near a corner will have been marked as such. If a traced edge forms a closed contour, we cluster all corner pixels within Euclidean distance of 4 pixels of each other and keep contours that have exactly four corner clusters (step 6 `GOOD_CANDIDATE`).

Step 10, `CORNER_REFINE` uses the standard OpenCV function `cornerSubPix` to refine the locations of candidate corners with subpixel accuracy. This has been shown [43] to give accurate corner locations to around 0.17 pixels. Corners are also sorted to ensure they have a consistent clockwise winding order.

Then in step 11 candidate quadrilaterals undergo `PERSPECTIVE_WARP` to remove the effect of perspective on potential fiducials and turn each into a square 16×16 pixel region to attempt decoding. For each candidate, we use OpenCV `getPerspectiveTransform` to obtain the perspective correction matrix from the refined corner coordinates, then use one QPU invocation per candidate to apply the matrix giving 256 sample points. These are fetched from the original input image using the TPUs to give hardware accelerated bilinear pixel interpolation for the warping process. Interpolated regions are written to a fixed target buffer of 128×64 pixels, sufficient for 32 post-warp candidates.

Finally at step 12, `BINARISE_DECODE`, we use the VPU to binarise each candidate by taking the minimum and maximum pixel values $p_{\min}, p_{\max}$ within the 6x6 information bearing region of the fiducial, corresponding to pixel locations $x, y \in \{2..13\}$, and establishing a threshold of $p_{thr} = p_{\min} + \frac{1}{2}(p_{\max} - p_{\min})$. The 2x2 pixel region of each

fiducial bit is averaged and the threshold applied to give a binary string. This simple approach proved just as effective as Otsu's method [44] and faster. Each binary string is manipulated to give four variants for the four possible rotational orientations, and these are checked for Hamming distance against the dictionary of valid symbols. The symbol in the dictionary with the lowest Hamming distance that is at or below the threshold value h_{thr} is regarded as a correctly decoded symbol.

Parameter values, shown in Table 1, were chosen empirically across different source material. The edge, corner, and error correction thresholds apply to both operating modes. Edge threshold e_{thr} represents a trade-off between speed and sensitivity since edge-tracing is the most expensive single part of the algorithm. It was chosen such that sensitivity was broadly similar to the ArUco library. Corner threshold c_{thr} was rather insensitive and placed mid way between values where detection reliability fell. Given the inter-symbol minimum Hamming distance of 12 bits of the `ARUCO_MIP_36h12` dictionary, we chose a correction threshold of $h_{thr} = 5$ bits to maximise sensitivity while still providing some buffer against wrong symbol identification. Relevant only for mode `ASCALE`, the scaled fiducial size β at 28 pixels represents the minimum fiducial size that could reliably be detected, and $n_{\max\_scaled} = 5$ a compromise between responsiveness to changes in scene and framerate gain.

Implementation of the algorithm was in C++ for the main code running on the CPU, with QPU code written in assembly and hand-optimised, and VPU code a combination of C++ and hand-optimised assembly. Optimisation was aided by the use of hardware performance counters detailed in [26] to find data starvation issues. Classes were implemented to encapsulate zero-copy buffers and pass these between CPU, QPU, and VPU parts of the implementation. The detector code was compiled to a library to be linked against applications. We also packaged a complete fiducial detection application using the Frappe library together with a minimal Linux to enable booting of the RPI Zero over USB without the need for an SD card.

3.3 Development environment

Developing the Frappe algorithm for the RPi Zero requires the VC4 GCC toolchain [30], the VC4 QPU assembler [28], and the OpenCV libraries compiled from source with appropriate optimisations for the ARM1176 CPU. Given the low performance of the CPU, and the limited amount of RAM (512 Kbytes), compiling directly on the RPi Zero is extremely slow. We created a Docker-based development environment, which we make freely available for use, with all the necessary cross compilers, QPU assembler, and build scripts. This allows the full power of a host PC to be used to cross compile and link code ready-to-run on the target RPi Zero far faster than possible natively.

The full source of the Frappe library, together with simple applications for performance measurement and headless streaming of camera images with detected fiducials are included within this environment to provide concrete examples for others who wanting to make use of the parallel processing abilities of the RPi Zero for other image processing or edge applications.

3.4 Test methodology

The aim of developing the Frappe algorithm was to improve the visual navigation capabilities of our DOTS robots, which requires real-time pose information. We performed two sets of evaluations. Firstly we measured the standalone algorithm speed, accuracy, and energy efficiency compared to ArUco when running on the same hardware, by using two recorded video sequences on a Raspberry Pi Zero. Secondly, we looked at robot system performance, comparing the original ArUco-based DOTS visual navigation system with a re-engineered Frappe-based version, delegating fiducial detection to the four camera RPi Zeros. We use a task relevant to intralogistics applications whereby a robot is placed at multiple starting locations in an arena and has to find, navigate to, then accurately manoeuvre under a cargo carrier.

3.5 Algorithm performance

Two sets of video footage were used. The first, *Office*, was recorded using a mobile phone in an office environment under varying lighting conditions with fiducials attached to the walls around the space. Also attached to one wall was the complete ArUco camera calibration target, providing a dense set of 24 fiducials. The second, *Arena*, was recorded from a camera on one of our DOTS robots while it was moving around within a purpose-built arena, within which large (225 mm) fiducials were attached to the walls, and small (45 mm) fiducials were attached to the four sides of cargo carriers. Each sequence was scaled to 640x480 pixels resolution and was run with each detector under varying parameters on an RPi Zero. We used ArUco version 3.1.15 and OpenCV version 4.5.2, in both cases compiled specifically for the RPi Zero with appropriate optimisations. Fiducials used were from the standard recommended ArUco fiducial dictionary `ARUCO_MIP_36h12`, which has 250 unique symbols encoded on a 6x6 grid within an 8x8 area, with a minimum Hamming distance of 12 bits between each symbol.

The standard ArUco library offers three modes of operation; `DM_NORMAL`, with adaptive thresholding, `DM_FAST`, with fixed thresholding, and `DM_VIDEO_FAST`, with fixed thresholding and adaptive input scaling. Parameters were left

at their default values except for the Hamming error correction rate, which we set to 0.42 in order to achieve a maximum of 5 bits of error correction for the `ARUCO_MIP_36h12` dictionary. We operate the Frappe algorithm in two regimes; `NORMAL`, with no scaling, and `ASCALE`, with adaptive input scaling based on previous frames. Hamming correction was set to a maximum of 5 bits.

For speed, we measure the time taken t to process each frame of video across both detectors in all modes, using an RPi Zero. We define the sensitivity s of a detector as the proportion of all true positive detections $D_{textall}$. Since we have no ground truth, we assume a similar detection by both Frappe and ArUco is a true positive, then verify the small number of non-intersecting fiducial detections in each sequence for correctness by hand. To examine corner accuracy we look at how our detector differs from ArUco with the set of all intersecting detections and find the Mean Absolute Error (MAE) $\bar{e}$ for the corner locations in pixels.

To characterise the energy efficiency of the algorithm, we use two metrics, the total energy used to process each frame *epf*, and the energy used for each frame in excess of the idle energy use Δepf. The RPi Zero was allowed to boot and reach steady state with no applications running and average power consumption P_{idle} measured over a 60 s period. Every tenth frame of each sequence was loaded and each decoder run continuously on the frame while average power P_{process} was measured over a 2 s period. During this period, the average frame time t_{frame} was also measured to give $epf = P_{\text{process}} \cdot t_{\text{frame}}$, and $\Delta epf = (P_{\text{process}} - P_{\text{idle}}) \cdot t_{\text{frame}}$. The maximum power consumption P_{max} of each sequence was also recorded. Only non-video modes (ArUco `DM_NORMAL`, `DM_FAST`, and Frappe `NORMAL`), representing the worst case processing load, were tested since we are presenting the same frame repeatedly.

3.6 System performance

The original DOTS visual system is detailed on the left in Figure 7. Each camera feed is compressed to MJPEG by its RPi Zero and sent to the RockPi 4 over USB for processing, where it is decompressed and ArUco tag recognition performed. The corner locations of each tag, together with their known size and the camera intrinsic parameters are used to estimate the relative pose between camera and tag, made available to the ROS transform tree for navigation. Transforms are acquired continuously. The Frappe version of the system extracts the corner locations and IDs of fiducials on the RPi Zeros and sends this information directly to the pose extraction step. The same camera calibration intrinsics are used in both cases.

The navigation task, illustrated in Fig. 8, is for a robot in multiple possible starting positions (1), to locate the carrier fiducial, move to a predock location (2) in front of the fiducial, pausing to capture more accurate pose information, then move accurately to the dock location under the carrier (3). Docking is successful if the robot stops within 50 mm of the correct location.

Each move is only possible if there is a valid transform, i.e. a carrier tag has been recognised. The carrier is located with its fiducial at arena location (0, 0). The robot is automatically placed at starting poses on a 0.1m grid in range $x \in \{0.35, 0.45.., 1.95\}, y \in \{-1.0, -0.9, .., 1.0\}$ facing in the $-x$ direction and attempts the task. Ground truth feedback from an Optitrack motion capture system is used to navigate to the starting point, then the navigation task is completed

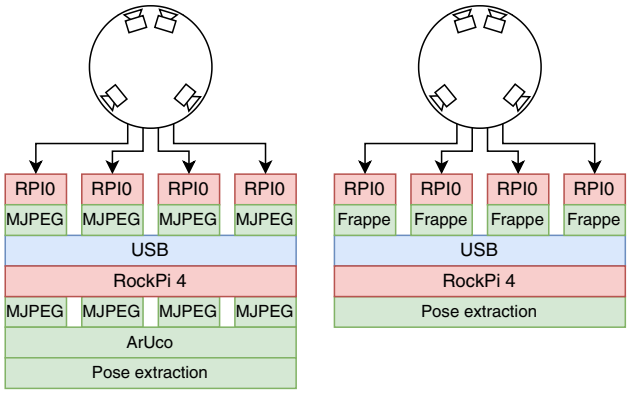

Fig. 7 DOTS vision system with four cameras. Left: original system with video compressed and streamed over USB to the central SBC for decompression and ArUco library detection. Right: New system with Frappe detection running locally on Raspberry Pi Zeros and fiducial positions streamed over USB

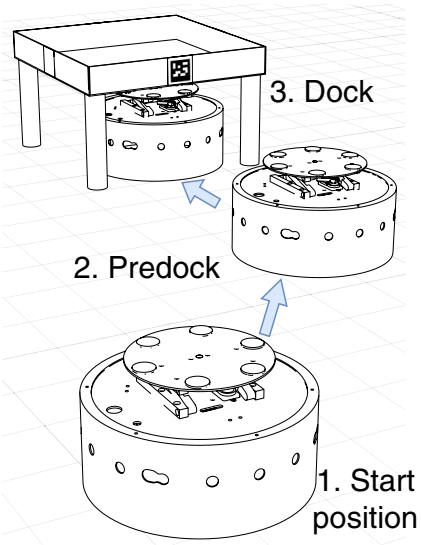

Fig. 8 System test. Robot starts from many different starting positions and must successfully navigate under the carrier using information from the vision system

Table 2 Performance on two video sequences, best results bold

Sequence	Arena			Office		
Information						
n_{frames}		1252			1499	
True positives $\lvert D_{\text{all}} \rvert$		1918			7078	
Positives per frame		1.53			4.72	
Detection	n	s	f	n	s	f
Aruco `DM_NORMAL`	1628	0.85	0	5961	0.84	0
Aruco `DM_FAST`	**1703**	**0.89**	0	5925	0.84	0
Aruco `DM_VIDEO_FAST`	1543	0.80	0	5883	0.83	0
Frappe `NORMAL`	1649	0.86	2	**6364**	**0.90**	2
Frappe `ASCALE`	1533	0.80	4	5782	0.82	3
Corner MAE (pixels)	$\bar{e}$	σ		$\bar{e}$	σ	
Frappe `NORMAL`	0.108	0.26		0.105	0.37	
Frappe `ASCALE`	0.164	0.26		0.098	0.54	
Frame time (ms)	$\bar{x}$	σ	fps	$\bar{x}$	σ	fps
Aruco `DM_NORMAL`	64.3	9.3	15.6	90.8	32	11.0
Aruco `DM_FAST`	119	120	8.39	169	130	5.91
Aruco `DM_VIDEO_FAST`	116	120	8.65	164	130	6.11
Frappe `NORMAL`	13.1	2.3	76.3	18.8	4.0	53.1
Frappe `ASCALE`	**10.6**	4.3	**94.2**	**14.3**	6.3	**69.8**
Power (W)	P_{max}			P_{max}		
Aruco `DM_NORMAL`	**1.76**			**1.77**		
Aruco `DM_FAST`	1.85			1.87		
Frappe `NORMAL`	2.08			2.00		
P_{idle} (W)	1.19					
Energy (mJ)	epf	σ		epf	σ	
Aruco `DM_NORMAL`	101	11		142	48	
Aruco `DM_FAST`	190	200		274	220	
Frappe `NORMAL`	**22.4**	2.8		**31.0**	5.9	
Net energy (mJ)	Δepf	σ		Δepf	σ	
Aruco `DM_NORMAL`	32.0	3.2		43.8	13	
Aruco `DM_FAST`	59.7	63		85.6	69	
Frappe `NORMAL`	**8.66**	0.87		**11.2**	1.8	

n number of detections, s sensitivity, f false positives, $\bar{e}$ corner Mean Absolute Error (MAE), $\bar{x}$ frame time, P_{max} worst case power consumption, epf energy usage per frame, Δepf energy usage per frame in excess of idle

entirely using pose information from the visual system and odometry. There are a total of 357 grid points, covering a floor area of 3.57 m^2. Each starting grid point is tested at least 5 times for both ArUco and Frappe, giving a success rate r for each point.

4 Results and discussion

Table 2 shows that the detection sensitivity of Frappe is very similar to that of ArUco, differing by only a few percent. Frappe `NORMAL` mode is slightly better on the *Office* sequence but slightly worse on *Arena* than ArUco. Frappe `ASCALE` mode is substantially the same detection performance as ArUco `DM_VIDEO_FAST`. For both detectors, the adaptive scaling modes are slightly worse than the static modes. False positive detections were very low; zero for all ArUco modes, and a maximum of 4 for Frappe `ASCALE` on *Arena*. Corner accuracy MAE is less than 0.16 pixels in all cases, which is what we would expect from the results in [43]. It is important to note that we are looking at the differences between the detectors, rather than assuming that one is 'correct'; both ArUco and Frappe use OpenCV `cornerSubPix` to perform the position refinement.

In all cases, Frappe is substantially faster than ArUco, with the slowest Frappe mode 4.8x faster than the fastest ArUco mode (Fig. 9). The ArUco `DM_FAST` modes are slower than `DM_NORMAL` mode. Examining individual

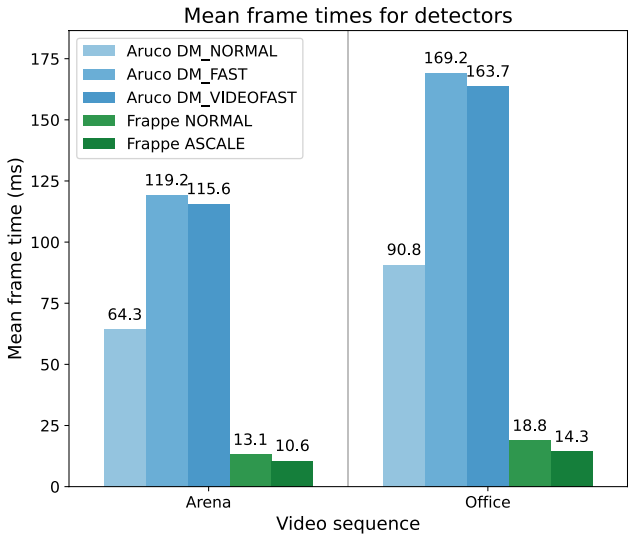

Fig. 9 Detector speed of ArUco and Frappe over all modes

frame results show very slow times for frames with no visible fiducials, we speculate because the detector in this mode tries up to three fixed thresholds per frame if there is no detection, costing more than the single adaptive threshold of `DM_NORMAL`. Although maximum power usage for the ArUco modes is slightly lower, the speed of Frappe means that Frappe uses considerably less energy to perform detection, at an average of 31 mJ per frame for the *Office* sequence, it is less than a quarter of the best performing ArUco mode.

We now consider the real-time performance, which we define as frame processing time never exceeding the camera frame rate of 30 Hz. In Fig. 10 we show how the frame times vary over both sequences when running Frappe in `NORMAL` mode. In no case does the time per frame exceed 33 ms, meeting our aim of 30 Hz from four cameras at 640x480 resolution. The QPU stages are almost constant time, and the VPU stages are a fairly small fraction. It is clear that the variation in frame times is dominated by CPU1, the contour

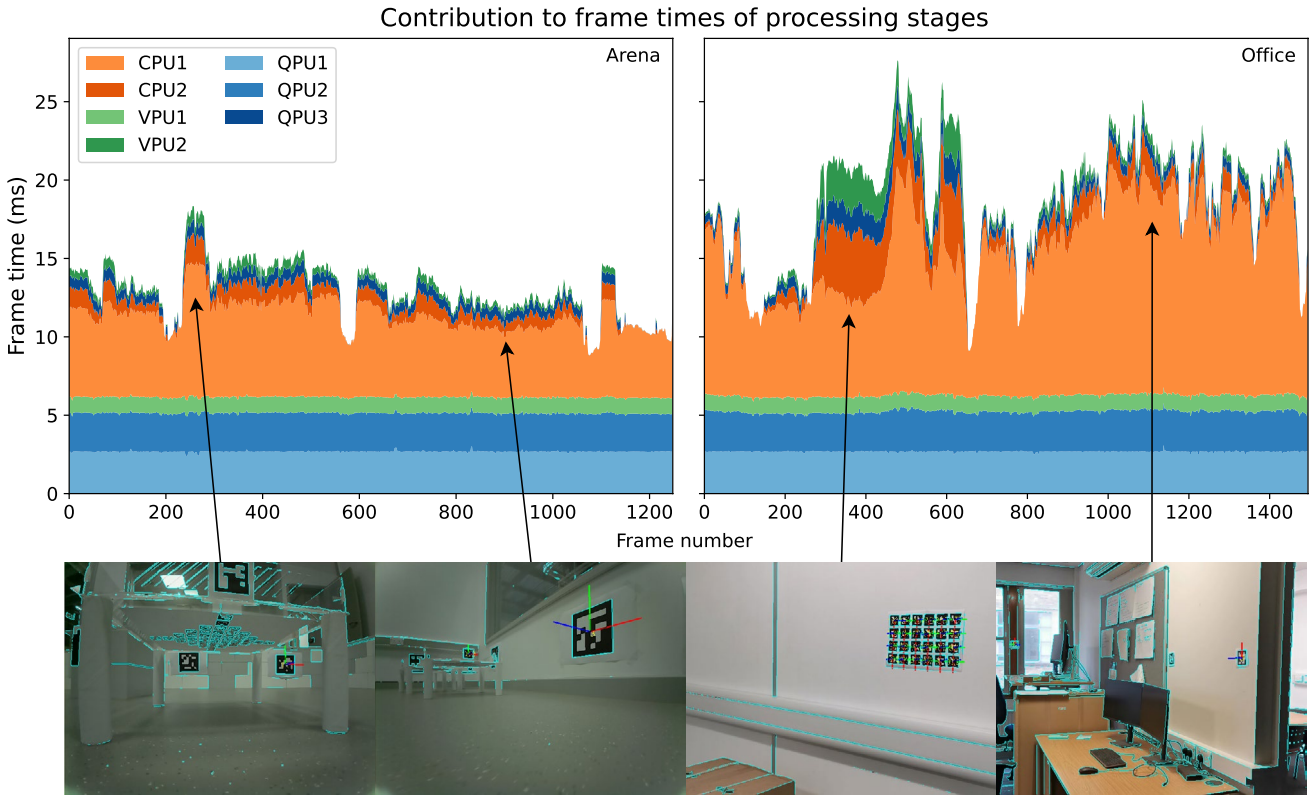

Fig. 10 Contribution of processing stages to frame times of Frappe in `NORMAL` mode. Below are sample frames from the *Arena* and *Office* sequences, with detected contours shown in cyan. QPU1: `CANNY_SHI_TOMASI` pass 1, QPU2: `CANNY_SHI_TOMASI` pass 2, VPU1: `FIND_EMPTY_TILES`, CPU1: `TRACE_CONTOUR`, CPU2: `CORNER_REFINE`, QPU3: `PERSPECTIVE_WARP`, VPU2: `BINARISE_DECODE`. The contour tracing stage is generally more expensive in the *Office* sequence, due to the more cluttered environment having more edges to trace around. Processing in the QPU stages and first VPU stage is virtually constant, and VPU2 decode stage showing an increase with the dense set of fiducials in the first *Office* sample frame

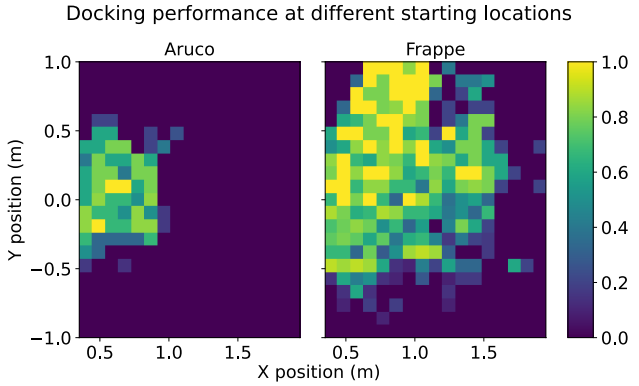

Fig. 11 Successful docking trials for different starting positions out of five attempts. Target is at location (0,0). Area from which docking attempt is likely successful is 3.5 times higher for the Frappe system (1.52m^2 vs 0.43m^2)

tracing stage, and the *Office* sequence spends much more time on this. Examination of intermediate outputs showed that this was mainly due to the *Office* environment having more clutter and high contrast edges. The data in Table 2 show that running in Frappe `ASCALE` achieves better than 60 Hz framerate, but this is an average—every fifth frame is forced to be processed at full resolution, giving the same frame times as shown in Fig. 10. Thus additional aggregate performance is available from this mode, provided the application can handle this greater variation of individual frame processing times.

Although the implementation of Frappe described here has a fixed resolution of 640×480 pixels, there is no reason this could not be extended to other resolutions, limited by the maximum hardware texture size supported of 2048×2048 pixels. Assuming constant pixel processing rate, XGA resolution 1024×768 could be processed at 27 *fps* and HD resolution 1920×1080 could be processed at 10 *fps*.

Also of interest in Fig. 10 is the fact that there is only a weak dependence of frame processing times on the number of fiducials. The worst case is *Office*, around frames 300–450, where the camera calibration target is in frame, shown in the third image along the bottom. The 24 fiducials result in a visible increase in CPU2—`CORNER_REFINE` and VPU2— `BINARISE_DECODE`. Another observation are the occasional dips in frame time, with no CPU2, QPU3, or VPU2 usage, e.g around frame 550 in *Arena*. Examination of these frames show this was due to fast panning causing blurring, resulting in few edges that could be traced, and no good candidates for decode.

For the robot navigation task, Fig. 11 shows the success rate r for each grid starting point for ArUco and Frappe. The area covered where the $r > 0.5$ for ArUco is 0.43 m^2 (12%) vs Frappe at 1.52 m^2 (43%), 3.5 times better. Frappe is clearly able to perceive fiducials from a further distance, likely due to the higher resolution of 640×480 vs 320×240 pixels, but also increasing the frame rate will increase the likelihood of a good detection.

5 Conclusion

The Raspberry Pi Zero has capable but underused processing ability which is of interest for low-cost edge computing and image processing applications. We demonstrate how this can be used to accelerate a common robotics vision task of recognising square fiducial markers, designing and implementing the Frappe algorithm to specifically target this processing ability. We show greatly improved speed over the standard ArUco library, running nearly five times faster on the same hardware, while using less than a quarter of the energy per frame.

Integrating the Frappe detector into our DOTS robot, we apply this additional performance to enable the vision system collectively to process camera feeds at four times the resolution, and twice the framerate of the previous system. This results in much better visual navigation abilities, with the robot able to perceive and navigate to targets over substantially greater distances.

There are several ways the algorithm and implementation could be improved which we intend to explore. Firstly, there is very little overlap of the use of the GPU hardware and the CPU. For example, pipelined operation could have the CPU performing the costly contour tracing of the current frame while the QPU was working on initial processing steps of the next frame. This would result in similar overall frame latency but support higher framerates. The current fixed resolution restriction could be removed. Contour tracing could potentially be accelerated on the VPU as a parallel connected-component labelling operation [45]. Adding greater resilience to motion blur is an area we intend to explore.

We look forward to developing other applications, such as implementations of alternate fiducial tag systems, and even potentially some computationally efficient CNN models. We make this software freely available to spread knowledge of the potential of the Raspberry Pi Zero parallel processing hardware and to encourage the use of this resource.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s11554-023-01373-w.

Acknowledgements SJ and SH are funded by UKRI Grant 10038942 and EU Grant 101070918.

References

1. Jones, S., Milner, E., Sooriyabandara, M. and Hauert, S.: DOTS: An Open Testbed for Industrial Swarm Robotic Solutions. 2022. https://arxiv.org/abs/2203.13809
2. Kato, H. and Billinghurst, M.: Marker tracking and HMD calibration for a video-based augmented reality conferencing system. In: Proceedings 2nd IEEE and ACM International Workshop on Augmented Reality (IWAR'99). IEEE, pp. 85–94 (1999)
3. Olson, E.: AprilTag: a robust and flexible visual fiducial system. IEEE Int. Conf. Robot. Autom. **2011**, 3400–3407 (2011)
4. Wang, J. and Olson, E.: AprilTag 2: efficient and robust fiducial detection. In: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, pp. 4193–4198 (2016)
5. Fiala, M.: ARTag, a fiducial marker system using digital techniques. In: 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), vol. 2. IEEE, pp. 590–596 (2005)
6. Garrido-Jurado, S., Munoz-Salinas, R., Madrid-Cuevas, F.J., Marin-Jimenez, M.J.: Automatic generation and detection of highly reliable fiducial markers under occlusion. Pattern Recogn. **47**(6), 2280–2292 (2014)
7. Romero-Ramirez, F.J., Muñoz-Salinas, R., Medina-Carnicer, R.: Speeded up detection of squared fiducial markers. Image Vis. Comput. **76**, 38–47 (2018)
8. Naimark, L. and Foxlin, E.: Circular data matrix fiducial system and robust image processing for a wearable vision-inertial self-tracker. In: Proceedings. International Symposium on Mixed and Augmented Reality. IEEE, pp. 27–36 (2002)
9. Benligiray, B., Topal, C., Akinlar, C.: STag: a stable fiducial marker system. Image Vis. Comput. **89**, 158–169 (2019)
10. Calvet, L., Gurdjos, P., Griwodz, C. and Gasparini, S.: Detection and accurate localization of circular fiducials under highly challenging conditions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 562–570 (2016)
11. Romero-Ramirez, F.J., Muñoz-Salinas, R., Medina-Carnicer, R.: Tracking fiducial markers with discriminative correlation filters. Image Vis. Comput. **107**, 104094 (2021)
12. Li, B., Wu, J., Tan, X., and Wang, B.: Aruco marker detection under occlusion using convolutional neural network. In: 2020 5th International Conference on Automation, Control and Robotics Engineering (CACRE). IEEE, pp. 706–711 (2020)
13. Mondéjar-Guerra, V., Garrido-Jurado, S., Muñoz-Salinas, R., Marín-Jiménez, M.J., Medina-Carnicer, R.: Robust identification of fiducial markers in challenging conditions. Expert Syst. Appl. **93**, 336–345 (2018)
14. Kalaitzakis, M., Cain, B., Carroll, S., Ambrosi, A., Whitehead, C., Vitzilaios, N.: Fiducial markers for pose estimation. J. Intell. Robot. Syst. **101**(4), 1–26 (2021)
15. Gallego, G., Delbrück, T., Orchard, G., Bartolozzi, C., Taba, B., Censi, A., Leutenegger, S., Davison, A.J., Conradt, J., Daniilidis, K., et al.: Event-based vision: a survey. IEEE Trans. Pattern Anal. Mach. Intell. **44**(1), 154–180 (2020)
16. Mur-Artal, R., Montiel, J.M.M., Tardos, J.D.: ORB-SLAM: a versatile and accurate monocular SLAM system. IEEE Trans. Robot. **31**(5), 1147–1163 (2015)
17. Engel, J., Schöps, T. and Cremers, D.: LSD-SLAM: large-scale direct monocular SLAM. In: European conference on computer vision. Springer, pp. 834–849 (2014)
18. Severance, C.: Eben upton: raspberry Pi. Computer **46**(10), 14–16 (2013)
19. Parham, K.E., Ferri, A.M., Fan, S., Murray, M.P., Lahr, R.A., Grguric, E., Swamiraj, M., Meyers, E.: Critical making with a raspberry Pi—towards a conceptualization of librarians as makers. Proc. Am. Soc. Inf. Sci. Technol. **51**(1), 1–4 (2014)
20. Jolles, J.W.: Broad-scale applications of the Raspberry Pi: a review and guide for biologists. Methods Ecol. Evolut. **12**(9), 1562–1579 (2021)
21. Taheri Tajar, A., Ramazani, A., Mansoorizadeh, M.: A lightweight Tiny-YOLOv3 vehicle detection approach. J. Real-Time Image Process. **18**(6), 2389–2401 (2021)
22. Rubino, E.M., Álvares, A.J., Marín, R., Sanz, P.J.: Real-time rate distortion-optimized image compression with region of interest on the arm architecture for underwater robotics applications. J. Real-Time Image Process. **16**, 193–225 (2019)
23. Paull, L., Tani, J., Ahn, H., Alonso-Mora, J., Carlone, L., Cap, M., Chen, Y.F., Choi, C., Dusek, J., Fang, Y. et al.: Duckietown: an open, inexpensive and flexible platform for autonomy education and research. In: 2017 IEEE International Conference on Robotics and Automation (ICRA). IEEE, pp. 1497–1504 (2017)
24. Millard, A.G., Joyce, R.A., Hilder, J.A., Fleseriu, C., Newbrook, L., Li, W., McDaid, L. and Halliday, D.M.: The Pi-puck extension board: a Raspberry Pi interface for the e-puck robot platform. In: Maciejewski, T., (ed.) IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2017), Vancouver, Canada: IEEE (2017)
25. Alhafnawi, M., Hunt, E.R., Lemaignan, S., O'Dowd, P. and Hauert, S.: MOSAIX: a swarm of robot tiles for social human-Swarm interaction. In: 2022 International Conference on Robotics and Automation (ICRA). IEEE, pp. 6882–6888 (2022)
26. Broadcom, VideoCore IV 3D Architecture Reference Manual, Broadcom, 2013. [Online]. Available: https://docs.broadcom.com/doc/12358545
27. Hermitage; H., et al.: VideoCore IV Programmers Manual, 2012. Available: https://github.com/hermanhermitage/videocoreiv/wiki/VideoCore-IV-Programmers-Manual
28. Müller, M.: vc4asm - Macro assembler for Broadcom VideoCore IV. 2014. . Available: https://github.com/maazl/vc4asm
29. Brooks, K.: Minimal Raspberry Pi VPU firmware. 2016. Available: https://github.com/christinaa/rpi-open-firmware
30. Brown, J.: VC4 GCC toolchain. 2016. Available: https://github.com/itszor/vc4-toolchain
31. Holme, A.: GPU FFT. 2014. Available: http://www.aholme.co.uk/GPU_FFT/Main.htm
32. 'mn416' and Rijnders, V.: QPULib. 2016. Available: https://github.com/mn416/QPULib
33. Seneral. VideoCore IV Computer Vision framework. 2020. Available: https://github.com/Seneral/VC4CV
34. "VC4 caches," 2019. Available: https://forums.raspberrypi.com/viewtopic.php? t=234167#p1432851
35. "VPU information," 2020. Available: https://forums.raspberrypi.com/viewtopic.php? t=287399#p1738410
36. Polat, A., Bayar, S.: A fast and energy efficient parallel image filtering implementation on Raspberry Pi's GPU. Eur. J. Tech. (EJT) **10**(2), 322–330 (2020)
37. Li, Y., Huang, D., Huang, S., Huang, S., Li, Y., Zhou, X., et al.: Sub-pixel gear parameter measurement based on Zemike moment.

In: 2019 IEEE International Conference on Mechatronics and Automation (ICMA). IEEE, pp. 2336–2341 (2019)
38. Faerman, V., Shvetsov, M., and Tsavnin, A.: Computations of cross-correlation functions on a single board Raspberry Pi computer. In: Journal of Physics: Conference Series, vol. 1615, no. 1. IOP Publishing, p. 012004 (2020)
39. Canny, J.: A computational approach to edge detection. IEEE Trans. Pattern Anal. Mach. Intell. **6**, 679–698 (1986)
40. Shi, C., Jianbo; Tomasi. Good Features to Track. In: 1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition. IEEE, pp. 593–600 (1994)
41. Sobel, I. and Feldman, G.: An Isotropic 3x3 Image Gradient Operator," Stanford AI Project (1968)
42. Suzuki, S., et al.: Topological structural analysis of digitized binary images by border following. Comput. Vis. Graph. Image Process. **30**(1), 32–46 (1985)
43. Kallwies, J., Forkel, B., and Wuensche, H.-J.: Determining and improving the localization accuracy of AprilTag detection. In: 2020 IEEE International Conference on Robotics and Automation (ICRA). IEEE, pp. 8288–8294 (2020)
44. Otsu, N.: A threshold selection method from Gray-level histograms. IEEE Trans. Syst. Man Cybern. **9**(1), 62–66 (1979)
45. He, L., Ren, X., Gao, Q., Zhao, X., Yao, B., Chao, Y.: The connected-component labeling problem: a review of state-of-the-art algorithms. Pattern Recogn. **70**, 25–43 (2017)