Accelerated Particle Filters

Adam Smallhorn and Steph McArthur

May 2016

Abstract

Particle filters are an attractive approach for object tracking because of their superior performance with noisy data and non-linear systems. By maintaining multiple hypothesised locations simultaneously they have superior object tracking properties, yet also have extensive computational requirements that inhibit their application to real-time applications. In order to overcome this limitation, an integrated FPGA implementation of a particle filter is demonstated that offers considerable performance enhancements over a CPU alone and enables real-time applications of up to 60fps.

Contents

1 Introduction

1.1 Applications

Object tracking is an area of intense interest in the computer vision field due to its numerous applications : autonomous vehicles, robotics and surveillance; just to name a few. There are several prevalent approaches to object tracking including Kalman filters, Extended Kalman filter and Mean shift but chief among these is the Particle filter because of its ability to track the motion of non-linear systems, as well as those with a high level of clutter. Part reason for this is the Particle filter's use of a probabilistic model of motion that maintains multiple hypotheses concurrently allowing for recovery from failure or temporary distraction. However, maintaining multiple concurrent hypotheses comes at a great computational cost which can impact the applicability of Particle filters to real-time or time sensitive applications.

1.2 Project Goals

Particle filters are a natural candidate for hardware acceleration due to the volume and nature of the computation required. The intention of this research was to build a particle filter in hardware that was capable of tracking moving objects in real-time, without delay, and without effecting the frame rate. The aim was to achieve a particle filter with at least 10 particles and 30 frames per second with a resolution of at least 320 x 240 pixels. What follows is a design for a Particle filter that spans software and hardware with the purpose of accelerating performance such that it is suitable for real-time applications.

1.3 Previous Work

Hardware accelerated Particle filters have been described in the literature in both standalone [JUCJ07, AAD05a] and integrated designs with a CPU [AAD05b, Jac14]. Notably M. Jacobsen [Jac14] was able to achieve object tracking of multiple targets at 60 Hz with a design partitioned across a CPU and FPGA. Our implementation however, provides a more robust estimate of object state, which is also processed in hardware.

1.4 The Particle filter

Particle filters provide robust object tracking of non-linear systems. What follows is an explanation of particle filters.

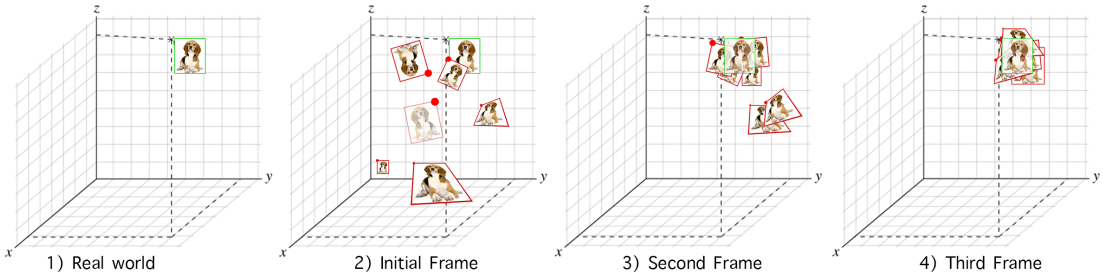

Figure 1: A visualisation of a particle filter with 6 particles.

Particle filters consist of multiple hypotheses of the object location, each hypothesis containing information about the object's possible translation and rotation in three-dimensional space. These hypotheses are known as particles, and a typical application uses from 10 to 1000 particles.

For the initial frame of the video, the particles are distributed across the image and every particle is assigned a score that represents how accurately it depicts the target object's location. This score is assigned by an arbitrary cost function. To estimate the object's location for the first frame, the algorithm uses a weighted average of the current location of all the particles.

Before the next frame is processed, all particles are re-sampled with a weighed distribution that favours the particles with a higher score. This redistributes the particles to areas that result in higher scores, which are more likely to be close to the object's actual location. Each particle is also mutated slightly to account for the motion of the object since the last frame. The score for each particle is calculated again, and the algorithm repeats.

2 Design

2.1 The Algorithm

This implementation is a standard particle filter with some notable features.

The algorithm first requires a template image, which will be used by the cost function during tracking. Once it has been set, the object can be tracked from frame to frame.

The state of each particle is stored using x, y and z coordinates for the translation, and x, y, and z values for rotations about the three orthogonal axes. These are known as state values. These values are used in a perspective transform to find the corresponding pixels in the camera frame. Using a perspective transform allows the object to be detected even if it is tilted towards or away from the camera, and if it rotates or changes in size. This was necessary as preliminary experiments demonstrated that simpler methods of locating the pixel in the frame (such as x,y translation, or an affine transform) were inadequate for modeling motion that moved in the z direction, or that caused distortion near the edges of the camera.

The cost calculation compares the template image with the pixels that surround the particle in the current frame. The cost is calculated as the sum of the squared difference between the pixel values in the red, green and blue channels. The square difference was chosen as the cost function because it was relatively easy to implement in hardware, and in testing, was significantly more effective than other simpler cost functions that were analysed, such as the difference function.

The re-sampling function selects the new particles based on the probability distribution of the particles, ensuring the higher scoring particles are more likely to be picked.

Mutation is performed by adding a random value (within a reasonable bounds of expected variance) to each of the state values for each particle. The addition of this mutation models a particle's hypothesis about where the object will travel in the next frame relative to the current frame.

2.2 Architecture

As can be seen in Figure 1, the overwhelming majority of time is spend calculating the perspective transform for each pixel, as well as the square difference. These functions are performed on hundreds of pixels, for each particle, for each frame. The more particles used, the better the tracking performance - but this limits the real-time application of the particle filter. Hence, this design implements the pixel perspective transformation and square difference calculation on the

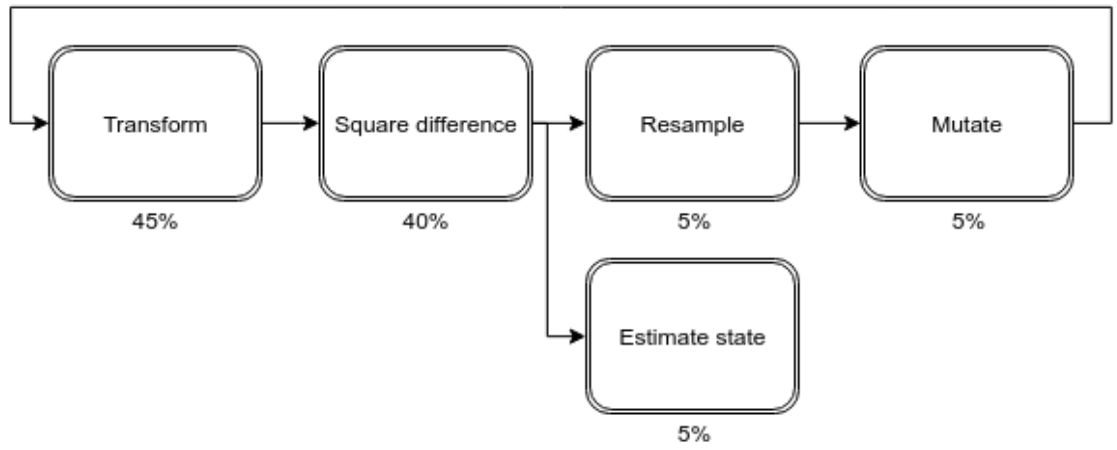

Figure 2: High level components of Particle filter algorithm and corresponding relative execution time for CPU-only implementation.

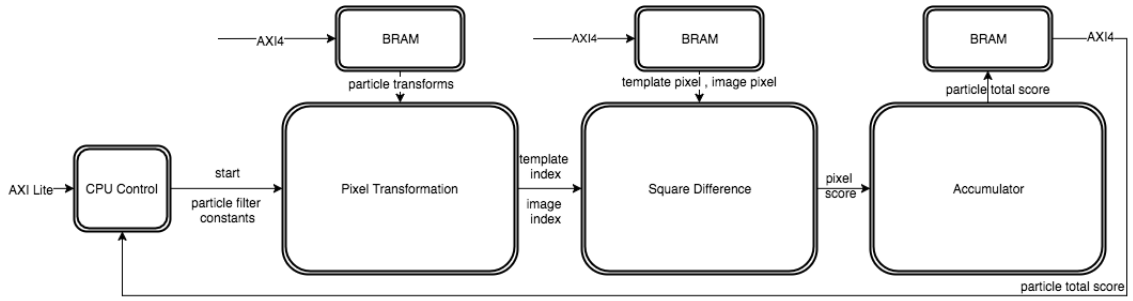

Figure 3: High level block diagram showing the three main hardware components implemented in the FPGA.

FPGA.

The calculation of the object's location (i.e. state estimation), the re-sampling, the mutation of particle state and calculation of transform matrices (used in the hardware pixel transformation) are calculated in software.

This partition between software and hardware is optimal for the largest speed-up with a relatively quick development time.

2.3 Hardware Design

2.3.1 Overview

The hardware architecture implemented in the FPGA is shown in Figure 2 and comprises three main components : Pixel Transformation, Square Difference and Accumulator.

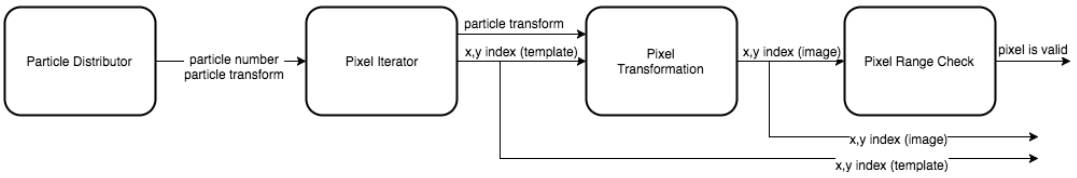

Figure 4: Design of the Pixel Transformation block.

The Pixel transformation block takes a start signal from the CPU control block, as well as some other constants such as the number of particles, maximum template and image size. The block fetches the transform for the current particle (which was pre-calculated on the CPU) and calculates a corresponding frame x,y index for each template x,y index. These values will be used in the square difference block to locate the comparison pixels.

The Square Difference block takes the x,y index for the frame and the template and uses these values to retrieve, from block RAM, a pixel for the template and a pixel for the frame. These 'pixels' are the three R G and B channels. The square of the difference for each channel is summed and the result it sent to the accumulator as the score for that particular pixel.

The accumulator sums all pixel results until it receives a last pixel signal, after which it will write the result to block RAM as the total particle score.

2.3.2 Pixel Transformation block

The internal components of the Pixel Transformation block can be seen in Figure 4. The first component is the particle iterator. The particle iterator is controlled by its own state machine and will fetch the first particle transform after it receives a start signal from the CPU control block. The particle iterator will pass this information on to the pixel iterator after it receives a request signal from it. This separation allows for parallel pixel iterators with a single particle distributor in future designs.

The pixel iterator iterates through all x,y index values. These x,y indexes are used to locate a pixel in the template image, and are also transformed into indexes for the frame image in next block.

The pixel transform block takes the current x,y indexes and the particle transform that was fetched earlier to compute what the indexes for the frame image will be. Figure 7 shows the formula used to transform the pixels.

Since it is possible for a pixel's indexes to be transformed to something that does not lie within

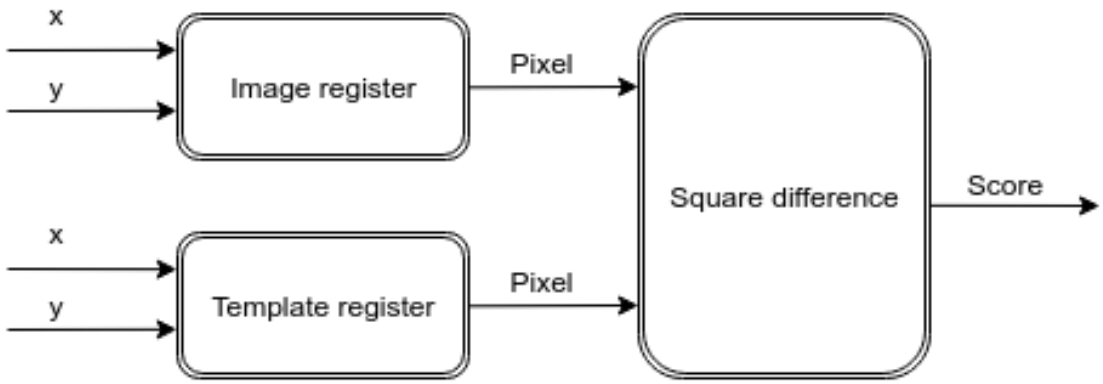

Figure 5: The square difference block.

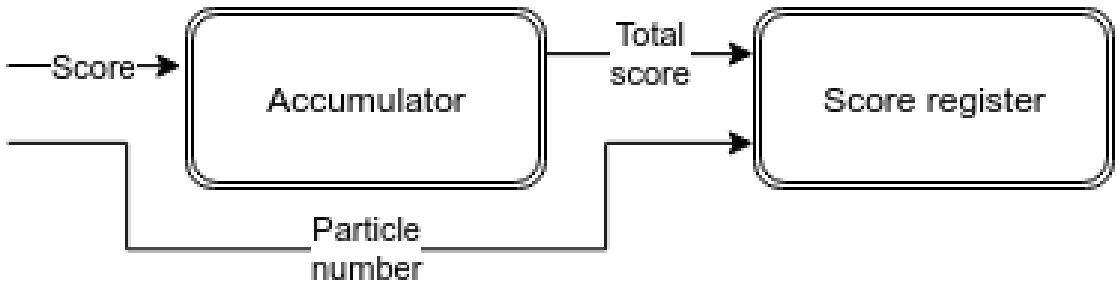

Figure 6: The Score Accumulator block.

the frame image (for instance it may mean that the particle is partially, or fully off screen) the pixel is checked in the Pixel Range Check block to see if it is a valid pixel or not. This determines whether a score is recorded in the accumulator, as off screen particles should receive no score.

2.3.3 Square Difference block

The square difference block, as seen in Figure 5, comprises two register blocks that take in x,y indexes for the template and for the image, and return to pixel values. These pixel values actually comprise three separate intensity values for the red, green and blue colour channels. Each channel of both pixels is compared and the difference between the two pixel is squared and all channel values are summed to provide a final score for the pixel. This is output to the accumulator block.

2.3.4 Score Accumulator block

The Score accumulator block, as seen in Figure 6, takes the pixel score from the square difference block and will sum each additional value with its total until it receives a last pixel signal, after which it will clear the current value, and add the total particle score to the score register in block RAM for access by the CPU.

$$\begin{bmatrix} \mathbf{d}_x \\ \mathbf{d}_y \\ \mathbf{d}_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(-\theta_x) & -\sin(-\theta_x) \\ 0 & \sin(-\theta_x) & \cos(-\theta_x) \end{bmatrix} \begin{bmatrix} \cos(-\theta_y) & 0 & \sin(-\theta_y) \\ 0 & 1 & 0 \\ -\sin(-\theta_y) & 0 & \cos(-\theta_y) \end{bmatrix} \begin{bmatrix} \cos(-\theta_z) & -\sin(-\theta_z) & 0 \\ \sin(-\theta_z) & \cos(-\theta_z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} \mathbf{a}_x \\ \mathbf{a}_y \\ \mathbf{a}_z \end{bmatrix} - \begin{bmatrix} \mathbf{c}_x \\ \mathbf{c}_y \\ \mathbf{c}_z \end{bmatrix} \right)$$

Figure 7: The perspective transform calculation. The result of the first three matrix multiplications is calculated in software. D is the transformed location of the particle. A is the location of the template summed with translation state variables of the particle. C is the location of the camera. Theta is the rotation state of the particle. Image from [Wik16].

2.4 Software

The following functions of the Particle filter are performed in software : the calculation of the object's final state, the mutation of particles, and the calculation of the transformation matrices for the pixel transformation.

In order to calculate the object's state for the current frame, the software needs to fetch the scores for each particle that have been calculated on the FPGA. These scores are used to weight the average of the state variables to provide an aggregate estimate of the object's location.

The particle states, which are stored in software, are mutated via a random number generator that produces Gaussian distributed numbers with a specified standard deviation. For example, x becomes x + noise. This ensures a variety of hypotheses for the object's position exist for each frame and that the population of particles do not degrade to a small number of states.

The transformation matrices are required in the FPGA by the pixel transformation block in order to determine which pixel in the frame corresponds with which pixels in the template. These transformation matrices are generated for each particle, based on its state variables, and is derived from a perspective transform formula, as seen in Figure 7. Since the first three maxtices require sinusoidal functions, these are calculated on the CPU and the result is computed as a single 3x3 matrix, which is sent to the FPGA. This resultant 3x3 matrix contains all the information about the particle's state, and is sufficient to determine where the particle would exist in the frame. The final matrix multiplication is then performed on each pixel's indexes in hardware.

3 Implementation

The system was implemented on the ZedBoard and used a Logitech C920 USB webcam to capture live data for object tracking. A Linux port for the ZedBoard, Xillybus, provided USB webcam drivers and a VGA driver so we could display results in real time. The FPGA logic was created in Vivado and used a combination of Xilinx IP and VHDL modules. Notable implementation details are below.

3.1 Memory management

In order to transfer data from the CPU to the FPGA fabric, an AXI bus is memory mapped in Linux to a certain address range, and accesses by the CPU to this address range will create AXI bus requests to an AXI BRAM Controller which provides a transparent interface to the block RAM. The AXI bus can transfer data at up to 200 MB/s. Although the memory map regions are set to non-cacheable in Linux it seems that writing multiple values to the same address quickly will result in only the last value being written. This is only a problem for manipulating the control registers and is solved by calling msync on the offending bytes after each write.

As the block RAM is true dual port SRAM, the AXI BRAM Controller and the FPGA logic can access the block RAM simultaneously and with different clock frequencies. This allows us to run the AXI4 bus at 100MHz whilst our FPGA logic can run at the slower max frequency of 50MHz. The FPGA logic can randomly access the data that has been placed in block RAM with one cycle latency. Furthermore, the FPGA logic can simultaneously access multiple block RAMs which facilitates effective pipelining. To allow the logic to efficiently locate data structures in block RAM, data is padded to power of two address boundaries.

3.2 Perspective Transform

Calculating the perspective transform requires several multiplications, additions and a division of decimal numbers. The DSP slices on the ZedBoard can calculate the multiplication or addition of 2 32bit numbers in 1-2 clock cycles (2 clock cycles increases max clock frequency). We used 32 bit fixed point numbers instead of floating point numbers in order to utilise these slices which maximise multiplication performance. 32 bit multiplication results in a 64 bit vector, so results were truncated by taking the 32 bits around the fixed point[1].

[1]For example multiplying 2 20:12 fixed point numbers would give a 40:24 result. Truncation should not change the value represented, just reduce its precision back to 20:12, so bits 44 down to 12 are selected by truncation out of the 40:24 64 bit fixed point number

Division is an expensive operation on an FPGA so this design uses a Xilinx division IP core that provides the quotient and remainder. Although the IP is heavily pipelined (70 stages long) it limits our clock frequency to 50MHz.

3.3 Pipelining and Parallelisation

The whole design is pipelined to provide one pixel score per clock tick, with 100 ticks latency. We can run the clock at 50MHz, which generates 50 million results a second. In addition to this some sections are parallelised where possible. Examples include the row/column multiplications and additions for the matrix multiplication, prefetching of the next transform state during the previous particle and the fetching of the template and image pixels at the same time.

3.4 Software Interface

Since only part of the particle filter is implemented in the FPGA we need to interface it with the software part of the algorithm. Section 3.1 describes how the particle state and current frame from the camera is transferred to block RAM using an AXI BRAM Controller. A memory mapped AXI4-LITE bus is used to set a start bit in control registers on the FGPA, which triggers the algorithm to run on the FPGA. These control registers are polled by the CPU for a done signal. Polling is used for simplicity, as interrupts would allow the CPU to do other work at the same time but are harder to set up (on Linux this would require a custom device driver). Once a done value has been read a reset value is written to the control register, this is used by the FGPA logic to reset state ready for the next iteration. Results (particle scores) are read out using an AXI BRAM Controller. The software can directly use the particle scores in the same manner as the original software implementation.

4 Testing

Testing was performed in several stages to reduce bugs where they would be too time consuming to debug (e.g. on the FPGA). Each IP block that was developed had a test bench with unit tests that could be visually inspected in the waveform viewer for correctness. An Intergrated Logic Analyser (ILA) IP block was used to probe signals within IP blocks when testing on the FPGA. Vivado can set up triggers in the ILAs to record data when certain events happen, allowing us to debug timing issues not apparent in simulation.

To test correctness of the system we used a collection of simple test cases with known results

which we could visually verify. Examples of tests include monochrome template and images with varying particle states (all scores should be the same known value), gradient images (scores should be incrementing by known amounts) and images with known locations of coloured squares that match the template.

The final tests were the integration tests which involved running the full particle filter on test sequences and real data. The test sequences used a moving square on a plain background. Once this visually appeared to be tracking well we moved to a live camera feed and visually confirmed the object was tracked.

5 Evaluation

5.1 Comparison of Implementations

We have software only, bare metal & FPGA and Linux & FGPA implementations of our particle filter. For comparison tests we have run all versions on the zedboard and use generated test sequences for the image to avoid the overhead of the USB-camera driver (which runs at 13Hz by itself, limiting our tests). A microsecond resolution timer is used to measure the iteration times for these tests with 1000 samples used for statistics. We did not have time to implement the full Particle filter on the bare metal platform so results for the Linux implementation without resample and estimate state are included for comparison.

5.2 FPGA Resource Use

The main constraint to our design was the limited amount of block RAM available on the ZedBoard. Each parameter of the particle filter has different effects on the block RAM required:

1. Each particle: 16 bytes for transform and 4 bytes for result
2. Each template pixel: 4 bytes (3 bytes per channel and 1 byte padding)
3. Each image pixel: 4 bytes (3 bytes per channel and 1 byte padding)

There are 140 36kb blocks of BRAM on the ZedBoard, giving a total of 630KB to use. One example configuration is 2000 particles (128kB of transforms, 8kB of results), 128x128 image (65kB of pixels) and 32x32 template (4kB).

Another constrained resource is the DSP slices. Processing one particles at a time uses 56 out of the 220 available DSP slices. This is not a limit in our current situation but if we processed multiple particles in parallel (e.g. 4) then the design would have to be optimised to make the most efficient use of them.

The rest of the resources used on the FPGA were less than 5 percent (excluding debug cores) except for the divider which used 10 percent of available slices. This will only become a problem if the particle pipeline is replicated, so is not a constraint at the moment.

5.3 Discussion

The final design for this Particle filter exceeds our initial design objects by a factor of 100 for the number of particles, and a factor of 2 for the frame rate. However, the resolution is limited to 256 x 250 pixels by the amount of available block RAM. Figure 9 shows how the cost calculation step has been significantly sped up by the use of the FPGA compared to software.

Compared to the CPU-only version of the algorithm, the Particle filter running on Ubuntu and the FPGA have achieved a speed-up factor of approximately 2x. Arhmdal's Law dictates that other computational components of the Particle filter are becoming the limiting factors on the total speed-up, as seen in Figure 8. The difference between the Linux + FPGA results and Linux + FPGA (with no resampling) results are caused by the part of the algorithm that still remains in software. There is some overhead when running Bare Metal + FPGA and Linux + FPGA (with no resampling). This could be due to numerous reasons, including the possibility that Linux has a greater memory management overhead or it is not spending all its resources transferring and manipulating data as in the bare metal version.

Our results are consistent with findings in the literature. Jacobsen achieved a frame rate of 60Hz with 600 particles, however his implementation performs less of the Particle filter algorithm in hardware. Hence his design did not require random access to the frame and template in hardware, and therefore did not suffer from the added overhead of IO transfers.

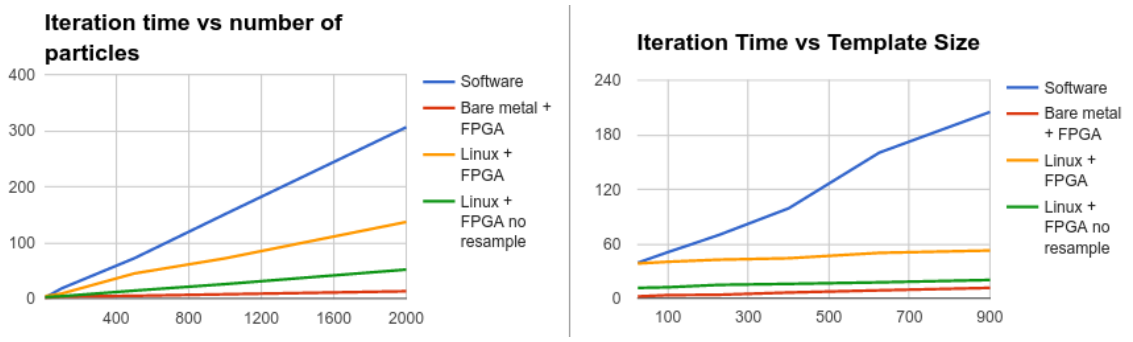

Figure 8: The left graph shows how the time per iteration of the particle filter varies with the number of particles. The test is performed with a 16x16 template and 128x128 image. The horizontal axis units are particles and left axis is in milliseconds. The right graph shows how the time per iteration of the particle filter varies with the number of pixels in the template. The test is performed with 500 particles and 128x128 image. The bottom units are the number of pixels whilst the left axis is in milliseconds.

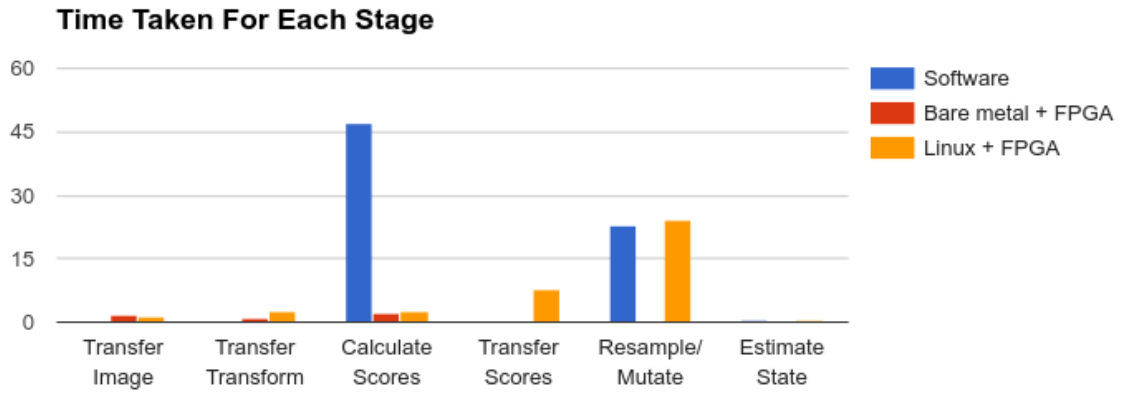

Figure 9: This bar chart shows how long each stage of the particle filter algorithm look for each implementation. Bare metal + FPGA does not implement resample and estimate state so results are not included for those. The left axis units are in milliseconds.

6 Conclusion

In this paper we have presented a Particle filter, implemented in software and accelerated via an FPGA, that is capable of performing almost real-time object tracking at 22 Hz. This presents represents a 2x speed-up over software alone. There is scope for optimisations and research to achieve higher throughput in the future, based on current research.

6.1 Future Work

There is scope for further investigation and development in the future.

A large proportion of execution time in this design is now the resample stage of the algorithm. A primary candidate for future research is moving this stage to hardware. This would reduce the overall iteration time of the particle filter to the same order of magnitude to the current score calculations on the FPGA. Such a modification would also remove the overhead of transporting the current particle state and results every iteration, which accounts for over half of the current memory transfer time overhead.

The design could be further augmented to support images of arbitrary size by dividing a frame into 4 or more chunks, processing each chunk sequentially, and summing the particle scores from each chunk. This would enable the Particle filter to handle High Definition camera sources of up to 1080 pixels.

Finally, the throughput of the Particle filter could be improved by processing multiple particles in parallel.

References

[AAD05a] Sangjin Hong Akshay Athalye, Miodrag Bolic and Petar M. Djuricn. Generic hardware architectures for sampling and resampling in particle filters. *EURASIP*, 2005.

[AAD05b] Sangjin Hong Akshay Athalye, Miodrag Bolic and Petar M. Djuricn. Generic hardware architectures for sampling and resampling in particle filters. *EURASIP*, 2005.

[Jac14] M. Jacobsen. Smart Frame Grabber: A Hardware Accelerated Computer Vision Framework. *Ph.D, University of California, San Diego*, 2014.

[JUCJ07] Xuan Dai Pham Jung Uk Cho, Seunghun Jin and Jae Wook Jeon. Multiple objects tracking circuit using particle filters with multiple features. *ICRA*, 2007.

[Wik16] Wikipedia. 3d projection, perspective projection, 2016.