Hardware Assisted Video Tracking

Matt Jacobsen, Mike Rose
mdjacobs@ucsd.edu, msrose@ucsd.edu

December 10, 2009

Abstract

We describe an implementation of video tracking using FPGA hardware and embedded software to track a target in real-time from a 640x480 @ 60 Hz video stream with less than 17 ms latency. The implementation runs primarily on the FPGA using a soft core CPU to support embedded software execution and a custom IP core to acquire and update video stream pixel data. This embedded implementation performs tracking 60 times faster than the same tracking algorithm run on a workstation.

1 Introduction

Object tracking is used in many applications and often using Kalman or particle filtering. These algorithms can be adapted to perform in real-time and used to track projectiles, aircraft or other time sensitive objects. Tracking can also be used to track slower moving objects, such as humans. Our interest in tracking is to provide the basis for gesture recognition for interactive displays.

Interactive displays allow users to input directives into computer applications in a natural way, using their limbs, hands, fingers, and even eyes. Natural gesture input has become a popular area of research and has spurred commercial interest, such as the 3DV Z-cam[3] and Microsoft’s Natal project[2]. We believe that high resolution video tracking can lead to a number of applications involving gesture recognition.

There are two dimensions on which tracking for real-time applications must perform well. The first is how quickly the tracking algorithm can calculate the new position. This ultimately dictates the number of frames per second the algorithm can update. The greater the number of updates per second, the more smooth and responsive the interaction. Frequent updates have an added benefit in that there is less time between updates for an object to move. Thus the algorithm need only consider a smaller area per update.

The other dimension is latency. This is the measure of time between the real life motion and the update the algorithm generates based on that motion. Any perceptible latency in interactive applications usually renders the application useless. Small amounts of latency are tolerable, but for must humans the amount of visual interaction latency must not exceed 100 ms before it becomes undesirable.

We therefore designed and built a video tracking device that uses a field programmable gate array (FPGA) with a soft core CPU to provide particle filter based tracking. The design uses IP cores to acquire pixel data from the incoming video stream and update the outgoing stream. The particle filtering algorithm is run entirely in software on the soft core CPU. We have tested our implementation using an input video stream of 640x480 @ 60 Hz. Our design can calculate a new position for the tracked point in 1.2 ms. At 60 Hz, the video is updated every 16.7 ms. Therefore, our tracking can provide updates every frame, though it can only do so in the subsequent frame.

2 Particle Filtering

Tracking in video for the purposes of interactive display is bound by two requirements. It must update frequently and it must not have any perceptible latency. To that end, we chose to use particle filtering to perform the tracking and found it well suited for our limited resources. Our design supports tracking
one point using 50 particles. The weighting function is NormalHedge, an algorithm devised at UCSD\cite{1}. Instead of performing numerous complex weighting calculations, NormalHedge allows a weight to be calculated using a simple exponential calculation.

\[ w_i \propto [R_i]\exp([R_i]^2/2c) \]

The weights \( w \) are proportional to the right hand side of the equation. The value of regret \( R \) has been calculated earlier. This calculation however relies on a constant, \( c \), that must be rediscovered each iteration. Luckily this constant is bounded and can be discovered through binary search. An appendix including the NormalHedge particle filtering algorithm is included at the end of this paper.

The algorithm runs in time \( O(n^2) \). Where \( n \) is the number of particles. These operations are dominated by the memory reads and writes. There are precisely \( 18n + n^2 \) memory accesses in the algorithm. As is the case with most real-time applications, the bottleneck is getting access to the data, not the time to compute an arithmetic operation. We leveraged this fact in our design to improve performance.

3 Hardware Design

Off the shelf video capture devices can perform well for mainstream applications. Most of these applications involve decoding and digitizing analog video then encoding and storing that video onto disk storage. The hardware and software are designed to buffer as much data as possible to allow the CPU enough time and access to recent frames to encode the data in a pipelined fashion. Rarely is the video modified, inspected, or filtered. Thus the video capture devices on the market are explicitly designed so has to induce high latency. Further, because encoding and moving video data is such a computationally intensive task, most devices don’t support capture at a frame rate higher than 30 frames per second (FPS). If the workstation CPU cannot handle data at a rate higher than 30 FPS, why build hardware that can capture at that rate? As such, most off the shelf video capture devices are ill suited for providing the hardware support to perform fast response, low latency tracking.

There are however, a handful of so called high performance video capture devices that focus on reducing latency. These devices are typically PCIe bus based PC cards that capture video (typically NTSC or PAL) fields in a frame buffer, then move the captured frame to main memory via DMA. The CPU then can access these frames, inspect, encode, or save as necessary. Often a software development kit (SDK) or library is provided for users to build custom applications using these devices. While these are much better offerings than their less performant brothers (that use firewire or USB buses), they still face the fundamental problem that processing and then updating a video stream faces several bottlenecks in a workstation.

3.1 Workstation Video Capture

The workstation video capture pathway requires first the video to be captured and buffered in a frame buffer. If the video is NTSC or PAL encoded, the odd and even fields of a frame are read sequentially and stored in a frame buffer. Once the frame is complete, the video capture device driver can copy the frame using DMA into main memory. Note however, that at this point a frame’s worth of time has elapsed and the CPU hasn’t even begin processing the data\cite{2}. The bandwidth of the PCIe bus will play a role in how quickly this DMA transfer can complete. A single PCIe lane can transfer up to 250 MB/s. But the larger issues affecting the delay will be the size of DMA transfers (i.e. transfer size) and latency of DMA requests. This of course, will vary depending on video capture device.

After the data is transferred from the frame capture buffer to main memory, the CPU can access the data from main memory as it wishes. However, this requires the data be moved from off-chip into the register space of the CPU before it can be processed. Of course, the data will most likely be cached in L1 or L2 memory, but the cost of getting the data from memory is extremely high compared to the speed of modern CPUs. It takes on the order of 100 cycles to read page of memory. The memory bus typically has very high bandwidth but it still suffers from request latency delays. If the entire frame must be read (i.e. for searching the frame or encoding it), this can be an appreciable cost.\footnote{Some extremely expensive devices attempt to mitigate this delay by beginning the DMA transfer before the frame has been captured completely. But these devices are equally as rare as they are expensive.}
Once the data has been processed it needs to be displayed. The CPU must issue array copy commands to copy the pixel data from main memory into video memory (i.e. a frame buffer in the video/graphics card). This imposes an additional delay which again depends less on the bandwidth of the PCIe or AGP bus but on the transfer size and latency in transfer requests\(^2\).

Afterwards, the video/graphics card can display the video data via some monitor. This too will impose additional delay based on how the video/graphics card processes the input video frame buffer.

### 3.2 Embedded Device Video Capture

Let us contrast the workstation video capture pathway with our embedded device capture pathway. Figure 1 illustrates the two pathways. The major difference is that with our design, we can reduce or eliminate nearly all the bottlenecks found with the workstation capture path and use the embedded device like a stream processor.

First the video pixel data is decoded and digitized. But instead of being buffered until an entire frame is ready\(^3\), the pixels are sent immediately via a dedicated set of wires to the custom video capture IP core. This method is the most efficient because each pixel is sent immediately after it has been digitized.

The custom video capture IP core looks at every pixel in the frame (as it comes in) and simply copies it out to the DVI encoder via a dedicated set of wires. It also looks for a small region of interest in the frame. When this region is available, the core copies the region into a small Block RAM (BRAM). After the region has been copied, the core writes a flag to the same BRAM to let the soft core CPU know the data for the current frame is ready. Lastly, the core will occasionally alter the pixel data it sends to the DVI encoder based on values in the BRAM.

---

\(^2\)Again, some of the very expensive devices can DMA captured frame buffer data to two different locations (main memory and video memory). However this optimization only supports video pass-through applications.

\(^3\)Each odd/even field is upscaled and additional pixels are interpolated to provide a progressive (i.e. full frame) video stream.
A chief difference between copying data from a video capture buffer to main memory and saving data to BRAM is that writes to BRAM happen in step with the arrival of new pixel data. For every new pixel, a pixel of data can be saved to BRAM. Additionally, there is no time spent on reading the pixel. The pixel is passing from the input digitizer to the output DVI encoder. It is available regardless of whether it is saved or not.

Once the custom video capture IP core writes the flag, the soft core CPU begins running the particle filtering algorithm. The soft core CPU has one application running on it, the particle filtering algorithm. There is no operating system or scheduler or kernel of any kind. Thus the particle filtering algorithm is never interrupted by another process. It continually reads the flag value and starts the algorithm when the signal is read. At this point, the soft core CPU must request the individual data elements that were saved in BRAM via a Processor Local Bus (PLB). Because the BRAM is on-chip memory, these accesses take on the order of 5-7 cycles not 100. It is similar to accessing data from L2 cache for a workstation CPU.

After processing is complete, new values are written back to BRAM and the cycle repeats.

Comparing the two different video capture pathways, one can see that the workstation pathway copies video frames three times before it can be displayed on a monitor. Additionally, it does so on a frame by frame basis. In contrast, the embedded device pathway streams the video through from digitization to output on a pixel by pixel basis, without ever copying.

### 4 Runtime Comparison

After our implementation was complete, we conducted performance tests to see how well the system performed as compared to a workstation running the same algorithm. To that end, we implemented our particle filtering algorithm on a workstation and configured a 1.8 GHz AMD workstation to perform single point tracking using 50 particles. We ran the workstation version against a stored video movie file. We had hoped to run it against a real video stream similar to that used with the embedded device, but we were unable to acquire the necessary video capture device resources within time. Instead we treat the video movie file as the live data. In our experiment, we read each movie frame in from disk and ensure it is in main memory. We then run the particle filtering algorithm and record the amount of time the algorithm takes to run. This process repeats and we take the average time as our measurement.

The time it takes our workstation implementation to run is 70 ms on average. This is obviously longer than 16.7 ms (the time of a frame at 60 Hz), thus the best our workstation implementation can do is update 14 times per second.

Our workstation experiment does not measure the video capture latency associated with a video capture device. Unfortunately, this measurement is quite difficult to come by in the literature as well. Opinions from users on forums and authors of video capture and editing software\(^4\) have claimed they have seen delays of 100 ms to 300 ms for video capture devices performing pass-through video display. This would be a rough estimate of the delay imposed by copying the frames through the workstation out to the video frame buffer.

We then ran our embedded implementation. To measure the time taken in just the algorithm, we counted clock cycles before and after. Knowing the frequency of the soft core CPU we were able to determine that tracking a single particle took 1.2 ms on average. This is 58.3 times faster than the workstation. The clock frequency of our soft core CPU is 125 MHz. Our soft core CPU is nearly 18 times slower than the workstation’s CPU, but completes the same algorithm nearly 60 times faster. This strongly indicates the the algorithm is data bound. Furthermore, our embedded implementation copies

\(^4\)http://www.virtualdub.org
data directly into BRAM. Therefore the cost of retrieving that data is considerably lower than for a faster clocked workstation to read the same data from main memory.

Our embedded experiment also does not measure the video capture latency. However, based on our design, we believe any latency will be in the digitization and/or rasterization of the video stream. Earlier experiments using high frame rate video cameras indicated that the same FPGA with a pass-through design resulted in 100 ms delay from real motion to display on the monitor\(^5\). We therefore expect our design to have similar delay. Table 1 lists a comparison between the two tracking approaches.

5 Conclusion

We have implemented a real-time video tracker using FPGA hardware and embedded software that tracks objects in 640x480 video streams at 60 Hz with only 1 frame of delay. This represents about a 60 times speed up over the same video tracker implemented using a workstation. The speed up comes not from parallelization of calculations, but from operating on the video data in stream processor fashion and moving the data path of the video data as close as possible to the CPU.

We believe this real-time tracker will provide the necessary fast response time, low latency base for natural input gesture recognition. We feel it will bring us closer to more natural computer interfaces.

References


\(^5\)Much of that 100 ms delay is believed to be from the LCD display we used.
1 Particle filtering for tracking in video

Algorithm 1 describes the general particle filtering algorithm. The details of its application to tracking in video are described below. We have not yet tuned any of the specific constants.

\begin{algorithm}
\textbf{input} \quad n \in \mathbb{N} (\text{number of particles}); \alpha \in (0, 1) (\text{discounting parameter}); F \subset \mathcal{X} (\text{set of fixed particles, with } |F| \leq n); x_0 \in \mathcal{X} (\text{initial state}); \text{Sample} : \mathcal{X} \rightarrow \mathcal{X} (\text{sampling oracle}); \text{Score} : \mathcal{X} \times \mathbb{N} (\text{scoring oracle}); \text{Trans} : \mathcal{X} \rightarrow \mathcal{X} (\text{transition function}); \text{Weight} : \mathbb{R}^n \rightarrow \Delta^n (\text{weighting oracle}).
\text{Let } E_0 \leftarrow F \cup \{\text{Sample}(x_0) : n - |F| \text{ times}\}.
\text{Let } R_{x,0} \leftarrow 0 \text{ and } w_{x,0} \leftarrow 1/n \text{ for all } x \in E_0.
\text{for } t = 1, 2, \ldots \text{ do}
\quad \text{Let } s_x \leftarrow \text{Score}(x, t) \text{ for all } x \in E_{t-1}.
\quad \text{Let } \bar{s} \leftarrow \sum_{x' \in E_{t-1}} w_{x',t-1}s_{x'}.
\quad \text{Let } R_{x,t} \leftarrow (1 - \alpha)R_{x,t-1} + (s_x - \bar{s}) \text{ for all } x \in E_{t-1}.
\quad \text{Let } E' \leftarrow \{x \in E_{t-1} \setminus F : R_{x,t} \leq 0\}.
\quad \text{Let } E'' \leftarrow \emptyset.
\quad \text{Let } \tilde{w}_x \leftarrow w_x \cdot 1(R_{x,t} > 0) \text{ for all } x \in E_{t-1}.
\text{for } x \in E' \text{ do}
\quad \text{Randomly select } x' \in E_{t-1} \text{ with probability proportional to } \tilde{w}_{x'}.
\quad \text{Let } x'' \leftarrow \text{Sample}(x').
\quad \text{Let } R_{x'',t} \leftarrow (1 - \alpha)R_{x',t-1} + (\text{Score}(x'', t) - \bar{s}). \quad \text{(Alternative: let } R_{x'',t} \leftarrow R_{x',t}.)
\quad \text{Let } E'' \leftarrow E'' \cup \{x''\}.
\text{end for}
\text{Let } E_t \leftarrow (E_{t-1} \setminus E') \cup E''.
\text{Let } (w_{x,t} : x \in E_t) \leftarrow \text{Weight}((R_{x,t} : x \in E_t)).
\text{Report } E_t \text{ and } (w_{x,t} : x \in E_t) \text{ to the user.}
\text{Let } E_t \leftarrow \{\text{Trans}(x) : x \in E_t\}.
\text{end for}
\end{algorithm}

1.1 Discounting parameter ($\alpha$)

The discounting parameter $\alpha$ should be related to the video frame rate. Roughly, $1/\alpha$ should be approximately the number of frames in which we expect the states to be stationary.

For our 20–30 fps videos, we use $\alpha = 0.2$.

1.2 State space ($\mathcal{X}$) and transition function (Trans)

We use a five dimensional state space $\mathcal{X} \subset \mathbb{R}^5; \text{ for } x \in \mathcal{X}$:

1. $x_1$ is the horizontal position;
2. $x_2$ is the vertical position;
3. $x_3$ is the size;
4. $x_4$ is the horizontal velocity;
5. $x_5$ is the vertical velocity.

The transition function $\text{Trans} : \mathcal{X} \rightarrow \mathcal{X}$ is the linear function defined by

$$\text{Trans}(x) = (x_1 + x_4, x_2 + x_5, x_3, x_4, x_5). \quad (1)$$
1.3 Fixed particles (F)

We let the fixed particles $F$ be a covering of $\mathbb{R}^3 \times \{0\} \times \{0\}$ (i.e. the velocity values are fixed to zero). Since the relevant horizontal and vertical positions are bounded, we just use a uniform covering in these dimensions. We use a coarser covering in the size dimension.

For $n = 1000$, we let $|F| = 100$ and consider four sizes $\{64, 96, 128, 160\}$ in the covering. This allows for a $5 \times 5$ grid at each size.

1.4 Sampling oracle (Sample)

The sampling oracle $\text{Sample}(x)$ returns a random draw from $\mathcal{N}(x, \Sigma)$, where $\Sigma = \text{diag}(\sigma^2_x, \sigma^2_y, \sigma^2_z, \sigma^2_\dot{x}, \sigma^2_\dot{y})$.

For our $640 \times 480$ video frames at $\approx 20$–$30$ fps, we use $\sigma_x = \sigma_y = \sigma_z = 4$ and $\sigma_\dot{x} = \sigma_\dot{y} = 0.5$.

1.5 Scoring oracle (Score)

Given a state $x \in \mathcal{X}$ and time $t \in \mathbb{N}$, the scoring oracle $\text{Score}$ returns a detection score for a square centered at $(x_1, x_2)$ with side lengths $x_3$ in frame $t$.

When $(x_1, x_2)$ is beyond the dimensions of the video frame, we have $\text{Score}$ return a very small score ($-100$). When the specified square does not entirely fit inside the video frame, we just return the score of the closest box of the same size that does fit.

1.6 Weighting oracle (Weight)

We use the Normal-Hedge weighting scheme. Given $(R_1, \ldots, R_n)$, the weights $(w_1, \ldots, w_n)$ returned by $\text{Weight}$ are given by

$$w_i \propto [R_i]_+ \exp \left( \frac{[R_i]^2}{2c} \right)$$

(2)

(if the right side is 0 for all $i$, then set $w_i = 1/n$ for all $i$). The value $c > 0$ is solution to the equation

$$\frac{1}{n} \sum_{i=1}^{n} \exp \left( \frac{[R_i]^2}{2c} \right) = e.$$  

(3)

The left side of this equation is decreasing with $c > 0$, so a simple bisection search will approximately find the solution. The weights are normalized to be a probability vector.