Optimization and Implementation of a Moving Mean Identifier and Hampel Identifier for Profile-Laser Scan Data, Using HLS

Jeremy Ford & Elakkia Kanaka Parthipan (with guidance from Isaac Phillips of Cognex)
CSE237c - Final Report
March 20, 2020

1. Introduction

The use of machine vision for noncontact measurement has become an integral part of many manufacturing processes. In particular, laser-profile scanners profive a robust solution for monitoring and verifying production lines. These laser-profile scanners utilize structured light and laser triangulation to obtain high-precision digital images and create cross-sectional profiles , which can be combined to achieve a 3D surface representation of the products. Outliers and artifacts are inevitable in the use of laser-profile scanners. These outliers and artifacts can be generated for a variety of reasons, including: sensor noise, distortion during processing, and human factors. Laser-profile scans contaminated with such outliers will lead to further errors in the data processing pipeline, and as such, must be removed from the data to improve the overall accuracy and precision of the measurement. For this project, we have teamed with an engineer from Cognex, a leading producer of laser-profile scanners to address the problem of outlier identification and replacement.

Several methods have been described for the recognition of outliers. We have explored two algorithms, the *Moving Mean Identifier* and the *Hampel Identifier*, which have shown promise as real-time outlier identification and removal methods for laser-profile data [1]. We then applied architectural and HLS optimizations to these algorithms to achieve lower latencies and initiation intervals. Finally, we implemented our design on the XILINX PYNQ - Z2.

2. Moving Mean Identifier

One approach that can be used to recognize and replace the outliers in a given laser-profile sance is the use of a moving mean identifier. The moving mean identifier relies on a sliding-window technique to process the data. For this project, we considered the window size of the filter to be nine. That is, in order to sample one data input, we will consider four data inputs before it, and four after it to determine whether the central data point is an outlier or not. The local mean of such a window is calculated as μ_i and the local standard deviation is calculated as σ_i. Then, the rule for determining if x_i is an outlier is as follows:

$$y_i = \begin{cases} x_i, & |x_i - \mu_i| \le 3\sigma_i, \\ \text{outlier}, & |x_i - \mu_i| > 3\sigma_i. \end{cases}$$

The optimisations done and performance achieved for the moving mean filter is shown below. The baseline version of the Mean filter refers to no optimisations being done, LUT implementation refers to using a precomputed array to determine the indices for inserting data into the window, rather than using shift register or modulus and division calculations (which proved to greatly increase latency and II). Best Mean Filter refers to the best HLS optimised version. For the best version of optimisation, we have unrolled all the loops, partitioned the arrays completely and pipelined the contents of the functions. Each input pixel is considered to be **ap_ufixed<16,12>** (as required by Cognex).

	Baseline Mean Filter	LUT Mean Filter	Best Mean Filter
Estimated Clock (ns)	6.972	6.972	6.823
Latency (cycles)	181	133	44
Interval (cycles)	181	133	3
BRAM (%)	0	1	0
DSP (%)	6	2	0
FF(%)	6	4	0
LUT (%)	12	12	1

Table 1. Synthesis results for moving mean filter.

The following images show the functional results of the moving mean filter:

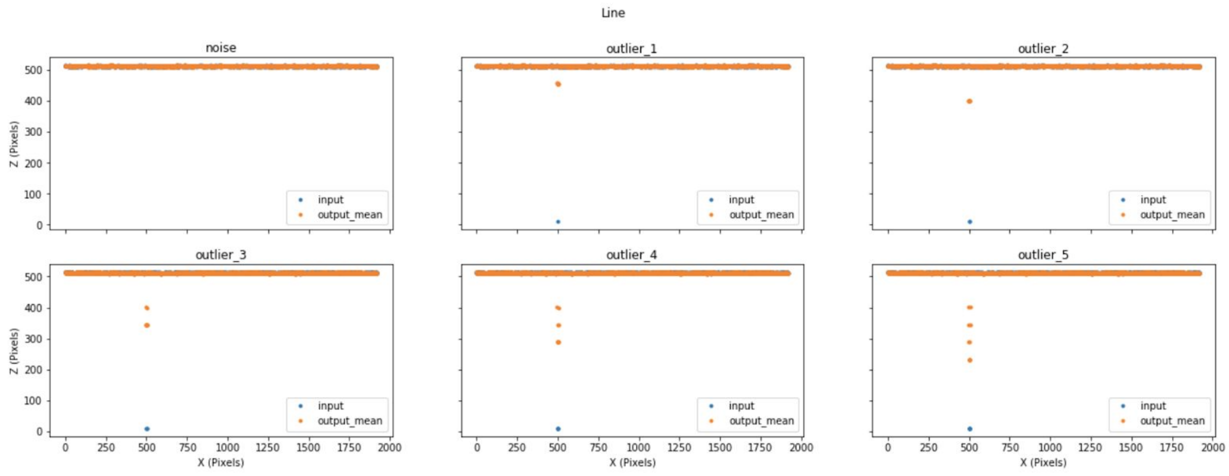

Figure 1. Moving mean filter; input = line + noise, with increasing numbers of outliers.

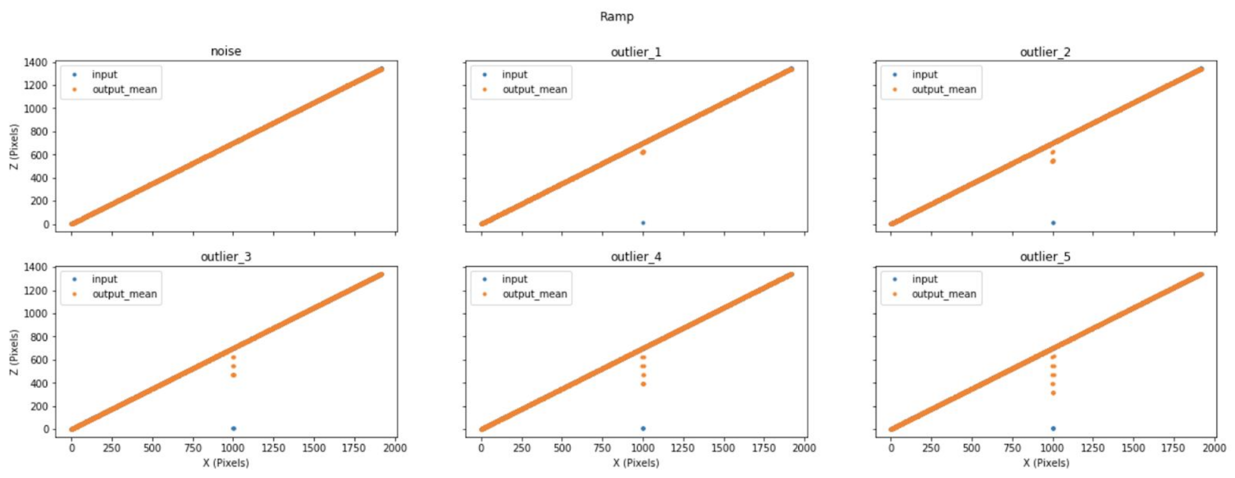

Figure 2. Moving mean filter; input = ramp + noise, with increasing numbers of outliers.

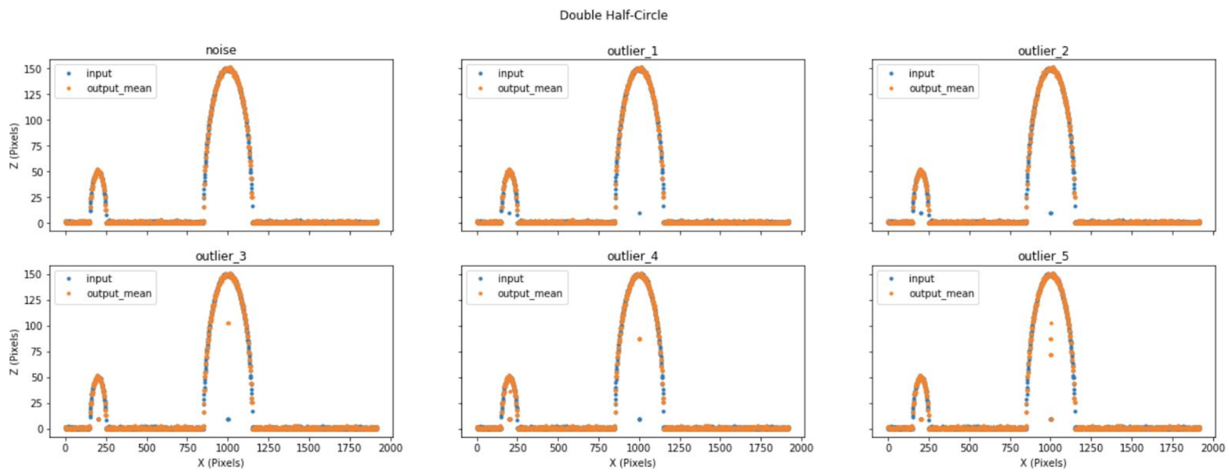

Figure 3. Moving mean filter; input = double half-circle + noise, with increasing numbers of outliers.

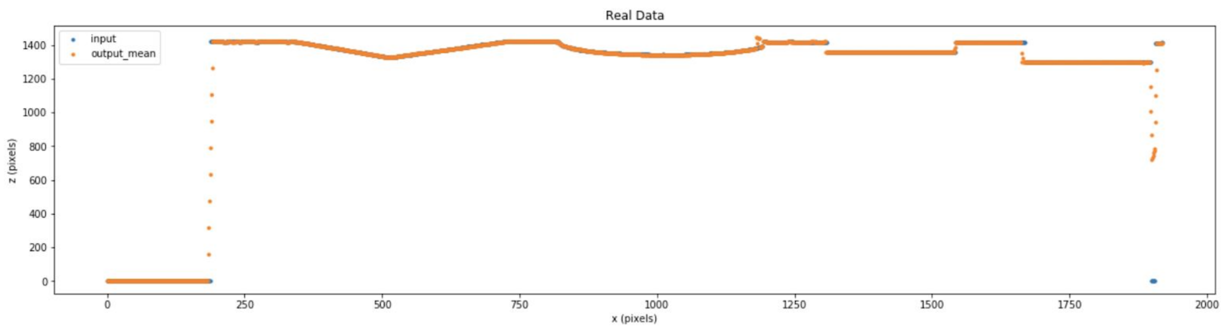

Figure 4. Moving mean filter; input = Cognex laser-profile scan.

As can be seen, the moving mean filter is poor at identifying outliers when the window size is just nine. In-order to catch an outlier in a window, we would have to use a window size between 15 and 30. This makes it extremely computationally heavy and therefore less desirable as a real-time algorithm.

3. Hampel Identifier

The Hampel Identifier uses the median and the median absolute deviation as a robust estimate of the location and spread of the outliers, that is, using the median to estimate the data position and using the median absolute deviation to estimate the standard deviation of the data, thereby effectively identifying upto four outliers with a window size of nine. First we compute the median of the window as:

$$m_i = \text{median}(x_{i-k}, x_{i-k+1}, x_{i-k+2}, \ldots, x_i, \ldots, x_{i+k-2}, x_{i+k-1}, x_{i+k-1}, x_{i+k}).$$

Then we calculate the median standard deviation Si as:

$$S_i = k \cdot \text{median}(|x_{i-k} - m_i|, \ldots, |x_{i+k} - m_i|),$$

k = 1.428 here is considered to be the unbiased estimation of the gaussian distribution.

The decision of whether an input data is an outlier or not is done based on the following condition:

$$y_i = \begin{cases} x_i, & |x_i - m_i| \le tS_i, \\ \text{outlier}, & |x_i - m_i| > tS_i. \end{cases}$$

The following table shows the results of the naive implementation of the median filter. Baseline version of the median filter refers to no optimisations being done and just executed, LUT implementation refers to using an array of values as indices for determining where an input should be inserted in the window, instead of using shift register or computations using modulus and division, which were a real bottleneck for the code. Best median Filter refers to the best optimised version. For the best version of optimisation, we have unrolled all the loops, partitioned the arrays completely and pipelined the contents of the functions.

	Baseline Naive Median Filter	LUT Naive Median Filter	Best Naive Median Filter
Estimated Clock (ns)	6.91	6.78	7
Latency (cycles)	509	435	65
Interval (cycles)	509	435	28
BRAM (%)	1	2	2
DSP (%)	3	0	0
FF (%)	3	1	3
LUT (%)	5	2	8

Table 2. Synthesis results for Hampel filter.

The following images show how effectively the median filter can remove the upto four outliers in a window.

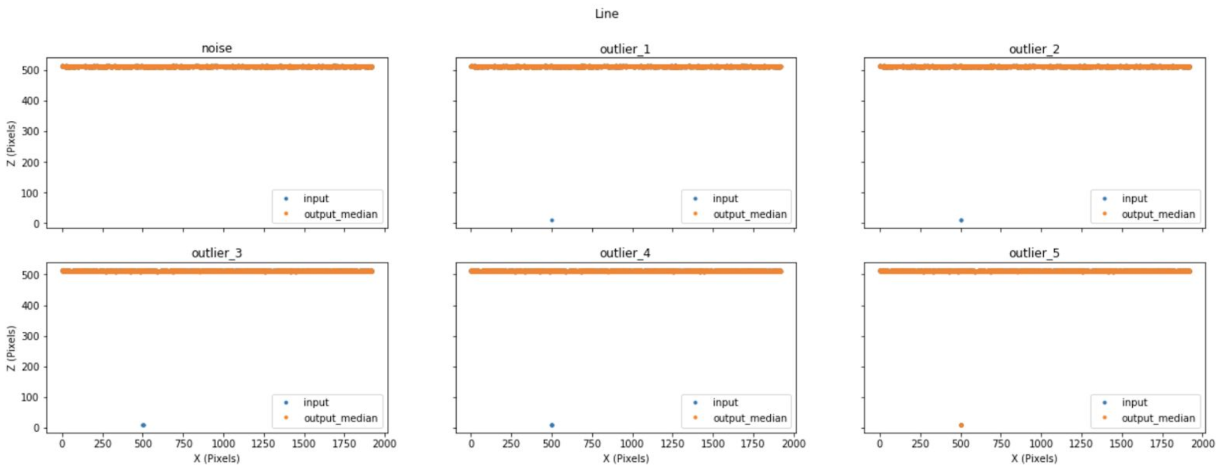

Figure 5. Hampel filter; input = line + noise, with increasing numbers of outliers.

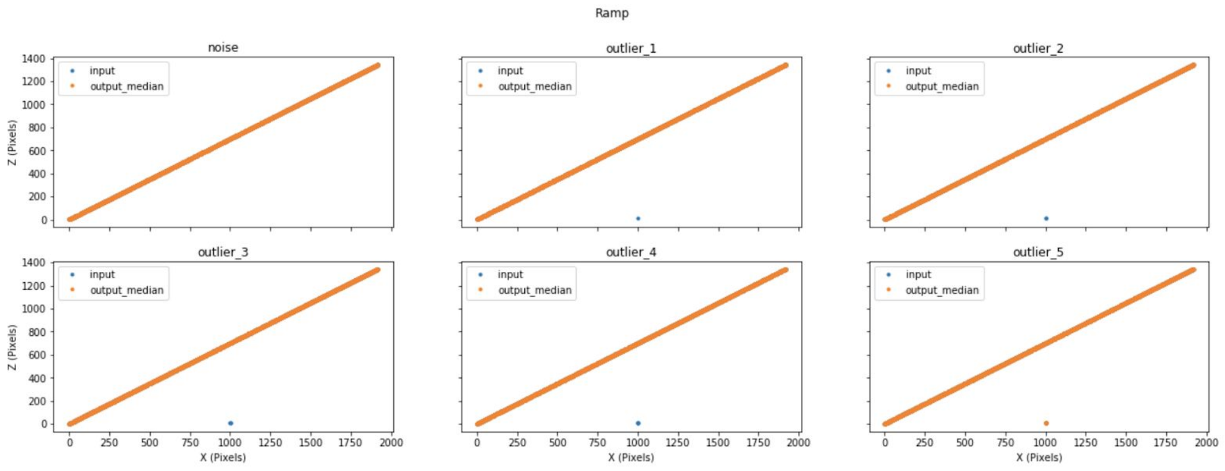

Figure 6. Hampel filter; input = ramp + noise, with increasing numbers of outliers.

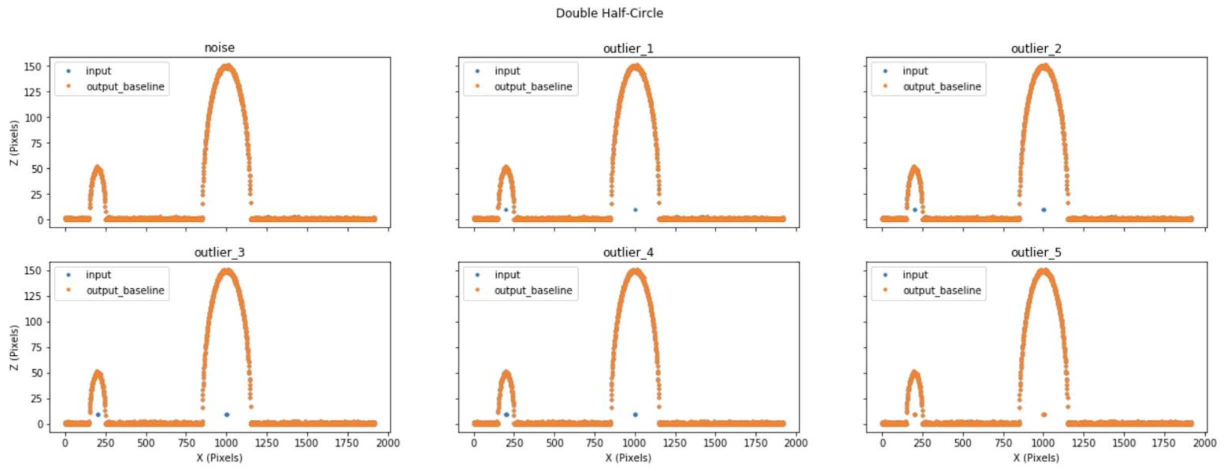

Figure 7. Hampel filter; input = double half-circle + noise, with increasing numbers of outliers.

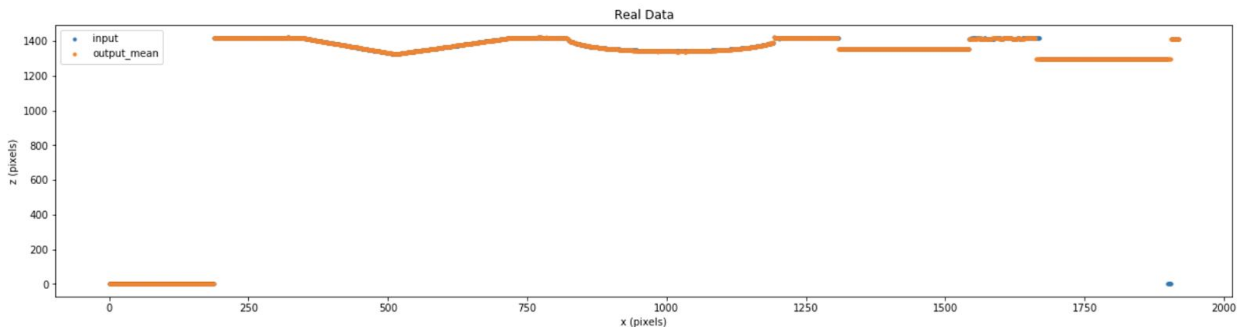

Figure 8. Hampel filter; input = Cognex laser-profile scan.

4. Optimizations

The naive median filter utilizes an insertion sort algorithm to first sort all values in the window, and then takes the middle value as the median. Although this algorithim works well to identify the median, optimizations can be performed to eleminate the use of the insertion sort and thereby reduce the total compuational complexity of the algorithm.

4.1. Optimization 1:An FPGA Implementation of a Fast 2-Dimensional Median Filter by Raju,Phukan, Baurah

Let us consider the window of nine as a three-by-three window. Here, we sort each of the rows separately, processing the maximum of the minimums of each of the three windows (MAX), minimum of the maximums of each window (MAX), and the median of the median values of each window (MED). Finally we find the median of MIN, MED, and MAX to get the overall median of the original window.

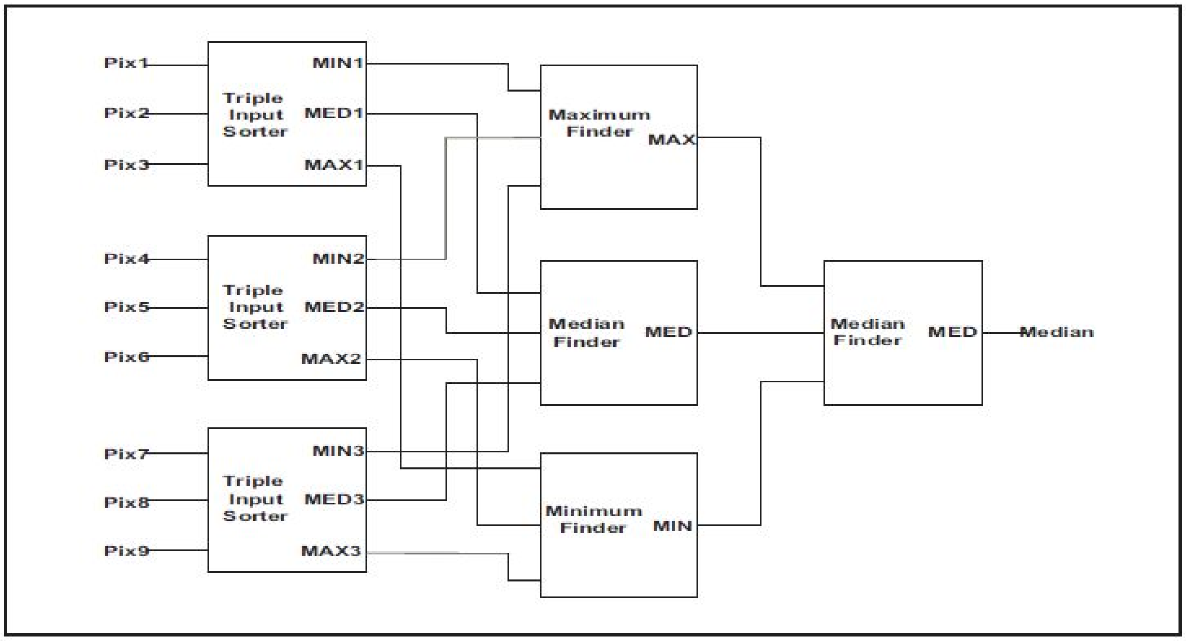

Figure 9. Block Diagram of median optimization 1.

As we can see, only for the processing of the input data, we have to sort all three sub windows. After that we insert the data into any one of the 3 sub windows and sort only that sub window, this eliminates a lot of the redundant processing. This is followed by the next stage of computing the medians from the maximum, median and the minimum of the three sub-windows are shown in the above picture.
On implementing this architecture for the median filter, we go the below performance:

	Baseline Median Filter - 1	LUT Median - 1	Best Median Filter - 1
Estimated Clock (ns)	6.91	6.78	6.532
Latency (cycles)	188	114	12
Interval (cycles)	189	115	3
BRAM (%)	1	2	1
DSP (%)	3	0	0
FF(%)	4	1	3
LUT (%)	8	5	8

Table 3. Synthesis results for Hampel filter - optimization 1.

4.2. Optimisation 2: An FPGA Based Implementation for Median Filter Meeting the Real-time Requirements of Automated Visual Inspection Systems by Vega-Rodriguez, Sanchez-Perez, Gomex-Pulido
In this particular implementation of the median filter, we do the sorting and median calculation using only magnitude comparators. This particular implementation uses 19 magnitude comparators to find the median of a window of nine data points. Again, this implementation reduces much of the redundant calculation from the naive implementation.

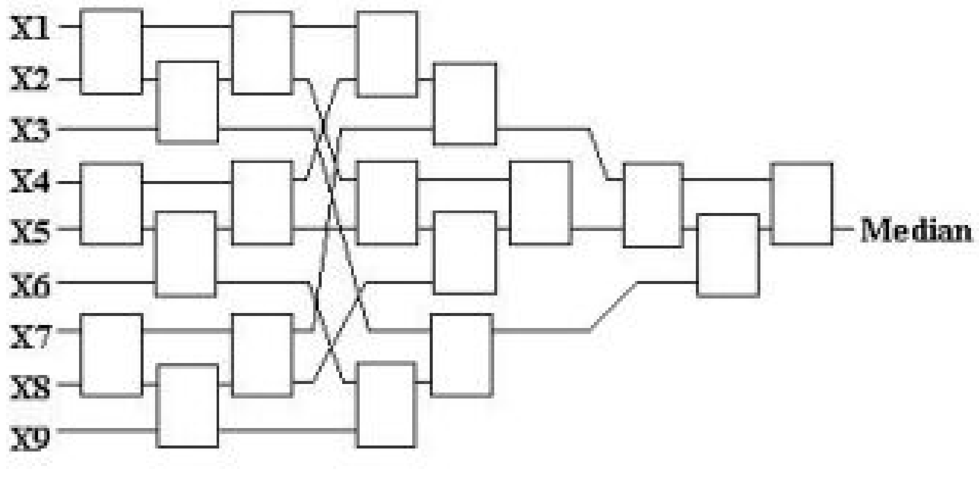

Figure 10. Block Diagram of median optimization 2.

The following results we obtained from using this architecture for the median filter:

	Baseline Median Filter - 2	LUT Median Filter - 2	Best Median Filter - 2
Estimated Clock (ns)	6.91	6.78	6.28
Latency (cycles)	153	116	23
Interval (cycles)	154	117	5
BRAM (%)	1	2	1
DSP (%)	3	0	0
FF(%)	4	1	6
LUT (%)	7	4	8

Table 4. Synthesis results for Hampel filter - optimization 2.

4.3. Optimization 1 vs. Optimization 2

We can see from the results that have been provided in the above two tables that, both the architectures work really well. We are able to get the entire median filter to give an output every three cycles for optimization 1, so we can say that it performs slightly better than optimisation 2.

5. Results

The following table shows the results of the best implementations of the different filters and architectures. We can see that the target of ten cycles per data sample has been achieved. In-fact we were able to three cycles per data sample.

	Best Moving Mean Filter	Best Naive Median Filter	Best Median Filter - 1	Best Median Filter -2
Estimated Clock (ns)	6.823		6.532	6.28
Latency (cycles)	44		12	23
Interval (cycles)	3		3	5
BRAM (%)	0		1	1
DSP (%)	0		0	0
FF(%)	0		3	6
LUT (%)	1		8	8

Table 4. Synthesis results for the best of each architecture.

6. PYNQ Implementation

We have implemented the Median Optimised version 1 on the Pynq Board. This has been implemented using the AXI Stream. We have displayed the results for three cases, the ramp, the double half circle and one of the real data profiles obtained from Cognex.