Median filtering with very large windows: SKA algorithms for FPGAs

Tyrone Sherwin*, Kevin I-Kai Wang*, Prabu Thiagaraj†, and Oliver Sinnen*
**Department of Electrical and Computer Engineering*, University of Auckland
tshe835@aucklanduni.ac.nz, k.wang@auckland.ac.nz, o.sinnen@auckland.ac.nz
†University of Manchester; Raman Research Institute, Bangalore
prabuthiagaraj@gmail.com

***Abstract*—Large scale median filtering algorithms are investigated in the context of the Square Kilometre Array (SKA) pulsar search; a signal processing pipeline estimated to require more than 10POps on 60PB of search data collected per day. Real time performance is needed for rectangular median windows of 63 frequency channels across 1023 time steps, requiring at least 64 million values to be calculated and output per second. This paper proposes an algorithmic approach for large scale median filtering based on existing techniques, providing improvements for the heterogeneous system used and utilising a high-end FPGA accelerator. Taking advantage of OpenCL for rapid parameter sweeping, the design space was explored to find the best algorithmic approach. The evaluation results are promising and show output of up to 99.3 million values per second on an Arria-10 FPGA, coming close to the limits set by resources and bandwidth. These results are set into relation with GPU and CPU implementations for the same algorithm, taking advantage of the OpenCL portability, achieving up to 16.8 and 9.1Mvalue/s respectively.**

I. Introduction

Radio Frequency Interference (RFI) mitigation is an extremely important part of using telescopes in radio astronomy [11]. The need for real time RFI mitigation has grown in response to increasing data volume, sensitivity requirements, and RFI contamination. The SumThreshold algorithm, proposed in [11] is able to estimate the underlying, true signal by flagging contaminated data through an iterative thresholding process. [17] implements this algorithm on a GPU cluster for the LOw Frequency ARray (LOFAR), one of the Square Kilometre Array (SKA) pathfinders.

We look at the median filter in the context of RFI mitigation using SumThreshold. An FPGA implementation to be included in the SKA Pulsar Search Engine is designed through collaboration with the Time Domain Team (TDT). Algorithmic optimisations are implemented and tested on a real FPGA-host system to operate on very large window sizes of 63 frequency channels across 1023 time steps. Development was carried out in OpenCL, a parallel programming standard for heterogeneous computing systems. This allowed for rapid development and evaluation of different algorithms. The main contributions of this paper are as follows:

1) Proposal of first approach to computing very large, rectangular window medians on an FPGA with configurable parallelism.
2) Demonstrating the suitability of OpenCL to design and implement such a median filter on FPGAs, utilising analytical upper bounds to argue about theoretical peak performance.
3) Experimental evaluation of the proposed approaches on a real, high-end FPGA accelerator, comparing performance with CPU and GPU implementations.

Section II introduces the median filtering algorithm and the relevant state-of-the-art. An FPGA architecture and algorithmic approach is proposed in Section III. Modelling of theoretical upper bounds is shown in Section IV before the evaluation is described in Section V. Finally, conclusions are drawn and opportunities for future work are discussed in Section VI.

II. Median filtering

Median filters are used where it is desirable to remove extreme values whilst retaining the overall structure of the data such as edges; this is in contrast to filters such as the Gaussian filter which creates new values and blurs the input data together [7]. This is accomplished by replacing each value with the median of the adjacent input values. A number of techniques and optimisations have been developed over time to improve performance, but in its simplest form the median filter is a problem of sorting.

```
1 for $col = 1$ to $X$ do
2     for $row = 1$ to $Y$ do
3         for $x = col - \frac{n-1}{2}$ to $col + \frac{n-1}{2}$ do
4             for $y = row - \frac{m-1}{2}$ to $row + \frac{m-1}{2}$ do
5                 load input[$x$,$y$] into array A;
6         sort A;
7         output A[$\frac{N-1}{2}$];
```

Algorithm 1: Basic median filtering for $n \times m$ window

An example algorithm is provided as pseudocode in Algorithm 1; iterating through a 2D array with X columns and Y rows, loading all values in the window to a new array, sorting that array, and selecting the middle value for output; assuming the usual odd number of values. In general, best case performance for sorting requires $O(N \log N)$ complexity [4], where N is the total number of values. As shown in [8], this can be reduced to $O(\log n)$ through the reuse of data between adjacent $n \times m$ windows. Due to the computational complexity inherent in sorting, a number of other techniques have been developed in order to reduce complexity and improve performance. Many of these techniques are based on assumptions about the input; limiting the number of possible values which the input may take in order to reduce the search space [12].

A. Related work

Median filtering is used in many signal processing applications such as 2D image processing [9], 1D speech processing [14], 3D medical imaging [3], and the 1D trace transform [7]. Many of these applications operate with very small window sizes. Square windows are common, ranging from 3×3 up to 7×7 [13]. In higher dimensional applications, sizes tend to be similar, such as $3 \times 3 \times 3$ [3]. Single dimension filters have a much larger range of size, ranging from 3-1023 points [7],

[14]. Applications utilising very large windows, such as those required here, are not very common. [19] and [12] are able to reduce the complexity so as to support such windows, providing results for square windows of radii up to 127, i.e. 255×255.

Huang et al. first introduced two widely used techniques in [9]; a sliding window technique for utilising data overlap, and a histogram based approach for storing values and calculating medians. Similar to stencil computations, each output value is determined by a window of input values. This also results in overlapping of input data for calculating adjacent outputs. As window sizes get larger this overlap increases. Taking a sliding approach, whereby at each iteration the values from the previous window would be used, allows for a significant degree of data reuse. A histogram of values in each window is able to reduce the median calculation to a linear complexity, independent of the window size. These techniques are explored further in Sections III-C and III-D.

III. Proposed architecture

This work aims to design a median filtering approach capable of processing data in real time for RFI mitigation in the SKA. Input data is represented by 8bit integers arranged in two dimensions, one being frequency channels and the other being time samples. For SKA1-Mid there are 4096 frequency channels, while the total number of time steps is dependent on the observation. This is further explored in Section III-E as the results can be buffered and broken into chunks. A window size of 63 frequency channels across 1023 time steps was chosen by the Pulsar Search team.

Achieving a high throughput is very important in the context of data processing in the SKA due to the real time requirements; processing must be at least as fast as the rate at which data is collected due to the continuous nature of the input data. As the input data consists of measurements across a number of frequency channels through time, the required throughput of the filter is determined by the time resolution of the data:

$$T = \frac{f_n}{t_{res}} \tag{1}$$

Where T is the throughput, f_n is the No. frequency channels, and t_{res} is the time resolution. For SKA1-Mid, using 4096 frequency channels, the required time resolution is ~64µs [10]. This equates to 64 million values calculated and output per second.

Many of the software implementations capable of operating with very large windows exhibited properties that made them undesirable for an FPGA architecutre. Existing FPGA implementations generally had high complexity making them undesirable for the large window sizes required. These factors led to the need for a new approach to be taken.

A. OpenCL and AOCL

OpenCL is a programming standard designed around the acceleration of parallel computing, targeting a wide variety of platforms [16]. AOCL is an OpenCL compilation framework developed by Intel FPGAcapable of describing high-performance computing applications on FPGAs [5]. The OpenCL computing paradigm describes applications as a host and some non zero number of kernels. Each kernel represents a computation to be performed by an accelerator controlled by the host. The use of AOCL automatically provides pipeline parallelism, and gives constructs for defining operation parallelism through the

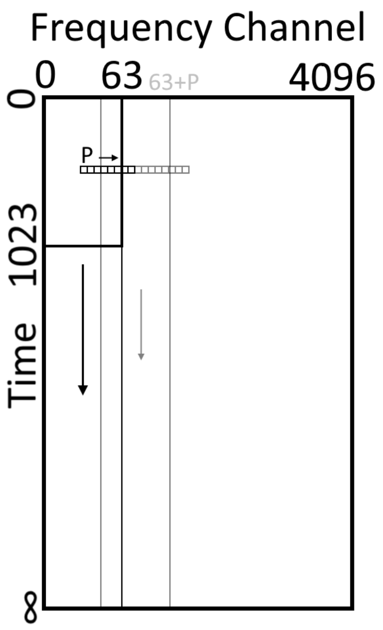

Fig. 1: The sliding window used, showing the 63 frequency channels and 1023 time steps. Sliding occurs through time, while P parallel values are calculated across multiple frequency channels.

OpenCL framework. More information about AOCL and its implementation of the OpenCL framework can be found in [5].

In the context of the SKA, [18] recommends High Level Synthesis (HLS), and shows that AOCL can provide performance approaching theoretical maxima. As many details are still to be confirmed during this prototyping stage [6], it is important to retain flexibility in hardware configuration. Using high level approaches accommodates the failure and replacement of components over the duration of the project's 50 year life span. It is also likely that new hardware will be added during the second phase of the project; SKA2 [15]. HLS allows for existing algorithms to be ported to new platforms and hardware configurations, and incorporation of new developments and improvements in algorithms; even by non-hardware experts.

B. Algorithmic approach

The algorithm is developed around three sections; loading, calculation, and output; where the output section is trivial in comparison to the other two. In conjunction with the high level approach, this allows for rapid development and evaluation of different combinations of techniques and optimisations across both the loading and calculation sections. Due to the large window sizes, an emphasis was placed on reducing the time spent loading data and minimising the complexity of calculation.

C. Sliding windows

A sliding approach for reducing loading requirements utilised by much of the other work in the area has been used here as well. [9] showed that in the case of a rectangular window the direction of sliding relative to the orientation of the window can make a significant difference. Moving orthogonal to the short edge of the rectangle, as shown in Fig. 1, allows for fewer values to be loaded at each iteration. Inspiration was drawn from [19] and many of the techniques investigated utilise a P parameter, indicating the number of parallel values being calculated. These values are

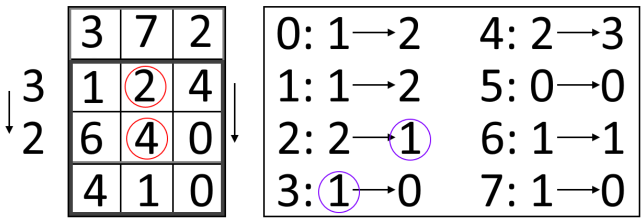

Fig. 2: 3×3 window, showing histogram approach with sliding update.

orthogonal to the direction of movement, or in the same axis as the short edge of the rectangle and is also shown in Fig. 1. The direction of movement here is the same as the direction of time.

Other techniques to further reduce loading were also investigated. Each row had to be loaded twice, reloading the oldest value from $row - \frac{n-1}{2}$ and loading a new value from $row + \frac{n-1}{2}$. These two values are then used before moving on to the next column. Once all columns in the window have been updated, the median can be found and work can begin on the next row. An alternative method utilises a buffered updated stage to first load values that need to be removed, and then to load all values that need to be added.

D. Histogram method

Given the very large sizes being implemented it is not practical to store all the values from a window, therefore some way to reduce the storage requirements is vital. First introduced by [9], the use of histograms decouples the calculation from the size of the input; instead being a function of the bit depth of the output value, with each bin storing the count of values for that bin. Fig. 2 shows a combination of this histogram technique with the sliding update as it is performed in [9]. The number of bins, B, is equal to 2^b, where b is the bit depth of the output value. For 8bit outputs, $B=256$ bins. This technique is quite similar to radix sorting, or bin sorting [4]. Algorithm 2 shows the changes made to Algorithm 1 in order to incorporate the sliding histogram approach, wherein the previous histogram is reused in the next iteration; replacing old values with new.

```
for $col = 1$ to $X$ do
    initialise histogram[1 to $B$] with padding values;
    for $row = 1$ to $Y$ do
        for $x = col - \frac{n-1}{2}$ to $col + \frac{n-1}{2}$ do
            decrement histogram[input[$x$,$row - \frac{m-1}{2} - 1$]];
            increment histogram[input[$x$,$row + \frac{m-1}{2}$]];
        $sum = 0$;
        for $bin = 1$ to $B$ do
            add histogram[$bin$] to $sum$;
            if $sum \geq \frac{N}{2}$ then
                $median = bin$;
                output $median$;
                break;
```

Algorithm 2: Approach taken

The histogram method has a calculation complexity of only $O(B)$, and also significantly reduces the amount of storage necessary. Each bin in the histogram need only be capable of storing at most the count of values in a window. For a 63×1023 window this requires 16bits per bin. For 8bit outputs, as is the case here each histogram requires 256 bins, giving an estimated size of 512Bytes per histogram. For contrast, approximately 64kByte is required to store all of the values in the window. Each of the P parallel calculations introduced earlier has an associated histogram, linearly increasing the memory requirements as the parallelism increases. To avoid double buffering of values to allow for simultaneous calculation and loading of adjacent windows, a so called 'little histogram' was introduced.

1) The little histogram: Given the sliding technique requires updating of the previous window, a loop carried dependency occurs in the loading section. When the AOCL compiler detects such a dependency the Initiation Interval (II) is extended in order to avoid potential memory conflicts. To allow for more effective utilisation of the generated pipelines, an intermediate variable can be introduced This intermediate variable stores only the changes in bin counts during sliding. Updating of a full histogram is then performed during the calculation, as the data is used, and is simply a case of adding the little histogram.

The little histogram only needs to store count values up to the number of values being added and subtracted at each step; being ±63 in 7bits per bin for the window size used here. Due to the version of AOCL,8bit values are used for each little histogram.

2) Output precision: It is also possible to reduce the computational complexity by reducing the output bit depth, b. As mentioned earlier, a lower number of bins, B, reducesboth the number of iterations for calculation and the memory requirements for storing histograms. Input values are matched to bins based on their b most significant bits. Values are less precise since fewer bits are used, with padding of the least significant bits being added. This tradeoff may be acceptable for some applications, and can be used alongside more complex changes.

E. Other considerations

To operate on continuous input data, there are two options available. The first is to operate on chunks of size $X\times Y$, re-initialising at each launch and incurring a penalty. The other option is to buffer input values, continuing to run the kernel as long as there continues to be new data to process. Due to the overlapping nature of the processing, this would require at least triple buffering of data, and storing of histograms between input chunks. In theory this can be extended, keeping as few as $m+1$ rows in global memory and replacing the oldest row at each time step. However without sufficient local memory to maintain all of the histograms this would requre significant overheads.

In addition the two different types of OpenCL kernel are investigated in AOCL, i.e. NDRange and single work item kernels [16]. In the NDRange case the input was broken into a series of work independent work groups. Each work group comprises a number of work items, each representing a column of the input, which are then able to share input data as in the P parallelism. A single work item kernel is then one where the number of work groups and work items are both set to 1.

Multiple variations on the median filter are evaluated in Section V. These variations are listed in Table I with their differences highlighted.

TABLE I: Design variations shown in evaluation

Design	Type	Details
MEDIAN	FPGA	Basic median filtering
LTL	FPGA	Implementation of little histogram
-BUFF	Modifier	Uses buffered input values
-ND	Modifier	Uses an NDRange kernel

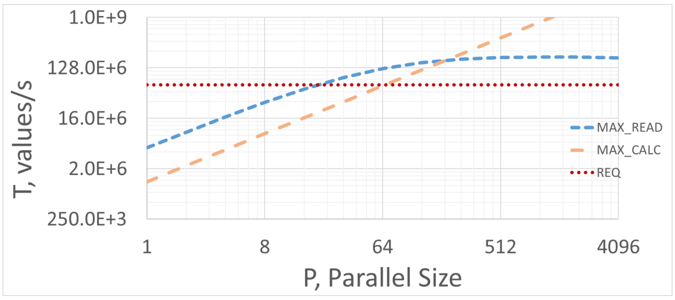

Fig. 3: Performance modelling of a 63×1023 window with calculation and read limitations for varying P shown with the required throughput.

IV. Performance model

Models were developed to determine the theoretical performance of our median filter, and are shown in Fig. 3. Equations (2) and (3) describe the relationship between the rate of output and the rate of input, while (4) and (5) describe the relationship between the rate of calculation and the rate of input.

$$T_{\text{READ}} = \left(\frac{l}{L} * P\right) * f \tag{2}$$

$$L = (n + P - 1) * 2 \tag{3}$$

where T_{READ} is the throughput in values per second with read limitations, f is the estimated operating frequency (here assumed to be 300MHz), L is the number of load operations required to calculate P parallel values, and l is the load utilisation measuring the average number of loads performed in each cycle.

$$T_{\text{CALC}} = \left(\frac{c}{C} * P\right) * f \tag{4}$$

$$C = B = 2^b \tag{5}$$

where T_{CALC} is the throughput with calculation limitations, c is the calculation utilisation measuring the average number of calculations performed per P in each cycle, C is the number of calculation operations required per output value, B is the number of bins in each histogram, and b is the bit depth.

In the case of the MAX_READ model T_{READ} is used with l set to 2. This reflects loading 1 new value and reloading 1 old value at the same time. In the MAX_CALC model, T_{CALC} is used with c set to 1, this represents the sequential nature of the median finding process being limited to 1 calculation step per cycle. As can be seen from Fig. 3, the constant C causes T_{CALC} to increase linearly with P. Since L is dependent on P, increasing P for the same l results in a drop off in throughput for T_{READ}.

This lead to another model, MAX_ABS, giving the lower of the two, with the turning point at $l/L = c/C$. For the values of l and c chosen, this is equivalent to $P < B - n + 1$ using T_{CALC}, and those above using T_{READ}. For $n = 63$ this point is at $P = 194$, as can be seen in Fig. 3. It can also be seen that $P > 64$ is required for the desired throughput of 64Mvalue/s. Each of these models are listed in Table II for reference in Section V.

V. Evaluation

A. Hardware/Software setup

For the purposes of this prototype development, the Nallatech 385A FPGA PCIe accelerator card was used in conjunction with an Intel Core i7-6700K CPU@4.0GHz, and 64GB of RAM. This is a high-end, modern platform representative of the level of technology anticipated to be used in the SKA [6]. Exploration of the design space was done to identify possible regions of adjustment

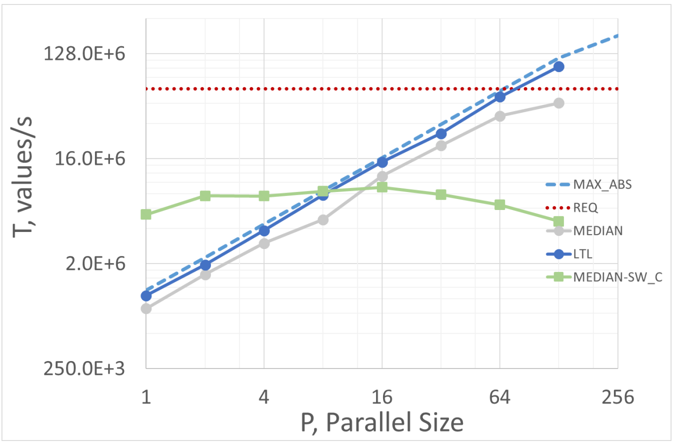

Fig. 4: Throughput comparisons for generated hardware of selected methods. Included is the MAX_ABS performance model.

based on the specific hardware used. Evaluation was carried out under CentOS version 7.3.1611, using AOCL version 16.0.2.

1) Functional correctness: To ensure functional correctness, a software reference (written in C++) was developed in tandem with the OpenCL kernel. In both cases, this initial implementation was largely derived from [9], utilising only the sliding window and histogram techniques. Both versions were implemented with a configurable P term to represent the degree of parallelism. In the OpenCL kernel this manifested as parallel pipelines, while in the software each P was calculated sequentially while still allowing for an overall reduction in loading, similar to tiling as in [2].

2) Evaluation metrics: The main metric examined was the overall throughput of the median filter, measured as output values per second or value/s. As mentioned in Section III-B the data throughput rate is quite important in the SKA context. Resources used were a secondary metric such that fewer resources allowed more duplication and higher performance, but generally were not examined on their own. The use of throughput as a measure of performance also allowed for an examination of the effects of varying X and Y, representing the total number of processed frequency channels and time steps respectively. For the purposes of this investigation, the focus was on determining viability of an FPGA accelerator; the methods of communication have yet to be decided and so this overhead has been ignored, however at present the AOCL runtime handles batched transfer of data over PCIe.

B. Initial Evaluation

Throughput comparisons were made between the initial implementation derived from [9] for both software (C++) and hardware (AOCL) versions, shown in Fig. 4 as MEDIAN-SW_C and MEDIAN respectively. The initial focus was on the effects of varying P and confirming the hardware was synthesised as expected. It was anticipated that the degree of P parallelism would be limited by the resources available in the form of logic elements (LEs), memory bits, and on chip RAM blocks, all of which are estimated by the AOCL compiler and reported

TABLE II: Model variations shown in evaluation

Design	Type	Details
REQ	Model	Required performance of 64Mvalue/s
MAX_READ	Model	Model with input rate limitation
MAX_CALC	Model	Model with calculation rate limitation
MAX_ABS	Model	Lower performance of above models

TABLE III: Throughput (value/s) for different versions of MEDIAN. Unless otherwise mentioned all results use parameter ($b = 8$). Tests where compilation was not possible are shown with a dash, while empty cells were not tested.

Design	Parameter varied								
	$P = 1$	2	4	8	16	32	64	128	256
MAX_READ	4.7M	8.9M	16.7M	30.5M	52.9M	85M	122.8M	157.7M	183M
MAX_CALC	1.2M	2.3M	4.4M	8.4M	16.3M	31.5M	60.9M	117.7M	227.4M
MAX_ABS	1.2M	2.3M	4.4M	8.4M	16.3M	31.5M	60.9M	117.7M	183M
MEDIAN	0.8M	1.6M	3M	4.8M	11.3M	20.7M	37.2M	48.1M	-
LTL	1.1M	1.9M	3.8M	7.8M	15.0M	26.3M	54.5M	99.3M	-
LTL-BUFF	1.1M	1.9M	3.9M	7.9M	15.6M	28.2M	37.7M	86.6M	-
LTL($b = 7$)	2.03M		7.86M			55.7M	90.7M	97.3M	121M
	$W = 1$	2	4	8	16	32	64		
LTL-ND($P = 1$)	0.78M	0.67M	0.66M	0.82M	0.86M	0.88M	0.88M		
	$I = 1$	2	4	8	14	16	24	28	32
LTL($P = 128$)	7.73M	7.10M	21.2M	40.1M	97.4M	91.0M	70.0M	92.5M	87.4M
	$b = 8$	7	6	5	4	3	2	1	
MAX_ABS($P = 1$)	1.17M	2.34M	4.69M	4.69M	4.69M	4.69M	4.69M	4.69M	
LTL($P = 1$)	0.94M	2.03M	2.43M	2.98M	3.69M	4.29M	4.35M	4.32M	

prior to synthesis. During evaluation it was discovered that the limiting factor was often routing, which was not reported.

Initial results for the hardware MEDIAN were on the expected order of magnitude, with loop carried dependencies causing inefficient generation of hardware. This resulted in the development of the little histogram adaptation, LTL, providing throughput which closely follows the performace model up to $P = 128$, shown in 4. A throughput of 99.3Mvalue/s is achieved, surpassing the required 64Mvalue/s for the pulsar search pipeline.

As the system must fully process input data, the latency is dependent on the total input size, $X \times Y$. For 16,384 time steps of SKA1-Mid data, approximately 1s buffer time, LTL($P = 128$) takes 0.70s to process. Further design space exploration was performed for the FPGA platform which will now be discussed.

C. Design space exploration

Where possible, the AOCL compiler would attempt to replicate memory in order to better utilise generated pipelines. Results for LTL($P = 128$) with a varying number of full column iterations, I, are given in Table III. Compilation logs indicate that an initiation interval of 14 is used when loading values, and the memory is replicated 16 times to compensate. It can be seen that when the number of iterations is less than 14 the performance is reduced. There is also a drop in performance between 14 and 28 iterations. Despite the number of iterations being known at compile time, the compiler would only reduce replication when there was a single iteration. For $P > 256$, this initiation interval may result in reduced performance, further compounding diminishing returns, however no tests were able to achieve such a degree of parallelism.

Locally storing all necessary values in the two rows being added and removed, rather than accessing the input array directly was evaluated and shown in Table III as LTL-BUFF. As can be seen from the table, this provides improved performance over LTL only in some cases, believed to be a result of the randomness of heuristics used in synthesis rather than indicative of better performance.

For an NDRange kernel, it is possible to vary the number of work items assigned to each work group, this value is provided as W. Overlapping data is shared across W parallel work items in a similar way as P. This can be seen from the throughput of LTL-ND($P = 1$) in Table III, where the value/s increases as W increases. Based on the obtained performance, it is believed that each work item is pipelined sequentially rather than performed in parallel as with P, giving the same amount of logic usage with an increase in memory usage and associated overhead.

Reducing B, the number of bins, is another method for reducing C, albeit with diminishing returns as P increases owing to the dropoff in T_{READ}. Table III shows the effect of the bit depth, b, where $P = 1$, improving performance by up to a factor of 4. It can be seen that the synthesised performance achieves this speedup at a much slow rate than the model suggests; achieving a two fold increase from the first bit reduction, but not achieving the full speed increase for another five bit reductions. In addition, the significant reduction in memory requirements for $b < 8$ meant that LTL($P = 256$) was able to be compiled.

Based on the results found here, it is likely that the throughput of the kernel will be higher than the rate at which data is input. This allows a multiple launch kernel to be used, allowing for lower complexity despite the additional overhead of relaunching. Preliminary testing for a single launch kernel also indicated a reduction in performance, further supporting this approach.

D. Comparison with GPU and CPU

A comparison was made with a GPU by taking the NDRange kernel developed for FPGA and compiling with another OpenCL compiler targeting AMD graphics cards. This involved making some modifications in order to allow compilation, and some steps were taken towards optimising for the resources available. Unlike the FPGA where each work item is pipelined, on a GPU each work item is treated as a separate thread and can be run in parallel. The OpenCL compiler used was also able to execute using the host processor to run the kernel directly, whereas AOCL sequentially emulates kernel execution on an FPGA by using the CPU.

Testing of the GPU was performed on another system consisting of an Intel® Xeon™ E5320 CPU@1.86GHz, 4GB memory, and AMD Radeon R7 370 GPU with 4GB RAM. Results can be seen in Fig. 5, for the two new series, Parallel Size refers to W, while for the other cases it refers to the P as before. To faciliate comparison between the platforms, this final FPGA implementation of LTL($P = 128$) requires 89,587 logic elements, 25MB of on chip memory, and 1,446 RAM blocks. This represents 21%, 47%, and 53% of the available resources respectively, while operating at a frequency of 209MHz. It should be noted that the results shown were provided using two different host PCs, as described here. For the MEDIAN-GPU and LTL comparison, this host PC is not relevant to performance as the entire algorithm described here is run on the accelerator.

In contrast to the previous work with an NDRange kernel, GPU performance improves as the parallelism increases. In the

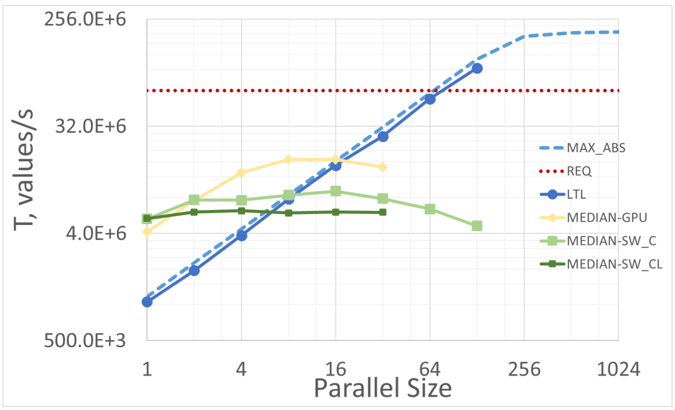

Fig. 5: Throughput comparisons for multiple platforms. LTL, MEDIAN-GPU, MEDIAN-SW_C, and MEDIAN-SW_CL correspond to the FPGA, GPU, CPU (C++) and CPU (OpenCL) implementations respectively.

case of the OpenCL execution on the CPU, there is negligible performance improvements, similar to the FPGA results for NDRange. Neither platform is able to compile for $W > 32$ due to resource constraints. In the case of the GPU, documentation lists availability of 16 compute units [1], agreeing with performance degradation for $W > 16$. It is unclear why performance drops off so rapidly before reaching 16. In both cases initial performance for low levels of parallelism is much higher than the FPGA implementation, likely a result of the significant difference in execution frequency, however scalability quickly becomes a problem.

It is believed that further optimisations to the GPU code could allow for increased performance, however it appears unlikely that significant improvements in scalability are possible for the algorithm described here. It is expected that a higher end GPU would be required in order to meet the throughput requirements set forth by the SKA and compete with the higher degree of parallelism available from the FPGA. An alternative algorithm may also be able to provide the required throughput by taking advantage of the GPU architecture, but no such approach is currently known for the very large windows required here.

VI. Conclusion

We investigated the design and performance of a large scale median filter on an FPGA for inclusion in the SKA Pulsar Search Engine, subject to real time constraints of 64 million values output per second with window sizes of 63×1023. These constraints were able to be met with the Arria-10 FPGA achieving up to 99.3Mvalue/s. The algorithm was developed using OpenCL which has been demonstrated to approach peak limits of performance of the hardware used. Increased parallelism may be possible, however theoretical models indicate diminishing returns.

The algorithm is highly flexible, operating on arbitrarily sized two dimensional windows and data, with variable degrees of parallelism; all supported by the OpenCL implementation. Algorithm performance is based on n, the number of columns in the window; b, the bit depth of output data; and P, the degree of parallelism. Total window size, N, affects the size of histogram bins and thus the degree of resource usage, but otherwise does not affect performance. Infinite amounts of data can be processed in chunks, where the chunk size determines the latency of the system.

In the future, we plan to better match the requirements set forth by the SumThreshold algorithm. This includes statically non deterministic numbers of values to support previously flagged data, and variable window sizes at run time. Other areas of improvement include the usage of arbitrary precision data, removing additional unneeded bits and reducing the amount of memory required. An improved loading step may also allow for fewer iterations and higher utilisation of the memory bandwidth available. While designed for the SKA Pulsar Search Engine, the algorithm developed here may find practical uses in other applications where a very large window is required and FPGA accelerators are available.

Acknowledgment

We gratefully acknowledge that this research was financially supported by the SKA funding of the New Zealand government through the Ministry of Business, Innovation and Employment (MBIE). T.S. thanks their colleagues, Leighton Hancock and Haomiao Wang, for their assistance. The authors would also like to thank Ben Stappers, Jayanta Roy, and the rest of the SKA-TDT for their input and work on the pulsar search pipeline.

References

[1] Inc Advanced Micro Devices. Amd radeon r7 series graphic cards designed for online gaming, 2015.
[2] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, pages 114–124, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
[3] Daniel Castano-Diez, Dominik Moser, Andreas Schoenegger, Sabine Pruggnaller, and Achilleas S. Frangakis. Performance evaluation of image processing algorithms on the gpu. *Journal of Structural Biology*, 164(1):153 – 160, 2008.
[4] Thomas H Cormen. *Introduction to algorithms*. 2009.
[5] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From opencl to high-performance hardware on fpgas. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534, Aug 2012.
[6] Peter Dewdney. Ska1 system baseline design. Technical report, SKA Office, 2013.
[7] Suhaib A Fahmy, Peter YK Cheung, and Wayne Luk. High-throughput one-dimensional median and weighted median filters on fpga. *IET computers & digital techniques*, 3(4):384–394, 2009.
[8] J. Gil and M. Werman. Computing 2-d min, median, and max filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(5):504–507, May 1993.
[9] T. Huang, G. Yang, and G. Tang. A fast two-dimensional median filtering algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 27(1):13–18, Feb 1979.
[10] L Levin, W Armour, C Baffa, E Barr, S Cooper, R Eatough, A Ensor, E Giani, A Karastergiou, R Karuppusamy, et al. Pulsar searches with the ska. *arXiv preprint arXiv:1712.01008*, 2017.
[11] A. R. Offringa, A. G. de Bruyn, M. Biehl, S. Zaroubi, G. Bernardi, and V. N. Pandey. Post-correlation radio frequency interference classification methods. *Monthly Notices of the Royal Astronomical Society*, 405(1):155–167, 2010.
[12] S. Perreault and P. Hebert. Median filtering in constant time. *IEEE Transactions on Image Processing*, 16(9):2389–2394, Sept 2007.
[13] Gilles Perrot, Stéphane Domas, and Raphaël Couturier. Fine-tuned high-speed implementation of a gpu-based median filter. *Journal of Signal Processing Systems*, 75(3):185–190, Jun 2014.
[14] L. Rabiner, M. Sambur, and C. Schmidt. Applications of a nonlinear smoothing algorithm to speech processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 23(6):552–557, Dec 1975.
[15] SKA. Frequently asked questions about the ska, 2017.
[16] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.
[17] R. V. van Nieuwpoort. Towards exascale real-time rfi mitigation. In *2016 Radio Frequency Interference (RFI)*, pages 69–74, Oct 2016.

[18] H. Wang, J. Gante, M. Zhang, G. FalcÃ£o, L. Sousa, and O. Sinnen. High-level designs of complex fir filters on fpgas for the ska. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 797–804, Dec 2016.

[19] Ben Weiss. Fast median and bilateral filtering. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 519–526, New York, NY, USA, 2006. ACM.