A Multi-Core FPGA-Based 2D-Clustering Implementation for Real-Time Image Processing

C.-L. Sotiropoulou, *Member, IEEE*, S. Gkaitatzis, A. Annovi, M. Beretta, P. Giannetti, K. Kordas, P. Luciano, S. Nikolaidis, *Senior Member, IEEE*, C. Petridou, and G. Volpi

***Abstract*—A multi-core FPGA-based 2D-clustering implementation for real-time image processing is presented in this paper. The clustering algorithm is using a moving window technique to reduce the time and data required for the cluster identification process. The implementation is fully generic, with an adjustable detection window size. A fundamental characteristic of the implementation is that multiple clustering cores can be instantiated. Each core can work on a different identification window that processes data of independent "images" in parallel, thus, increasing performance by exploiting more FPGA resources. The algorithm and implementation are developed for the Fast TracKer processor for the trigger upgrade of the ATLAS experiment but their generic design makes them easily adjustable to other demanding image processing applications that require real-time pixel clustering.**

***Index Terms*—Clustering methods, field programmable gate arrays, image analysis, multiprocessing systems, particle tracking.**

I. Introduction

THE development in image detector technology in the recent years has led to a great increase in resolution as well as the amount of produced data. These high-resolution detectors are used in a vast variety of applications from simple everyday use to the demanding applications of high-energy physics, astrophysics, security and biomedical science. Additionally, the majority of these applications call for real-time data capture at high input rates. Taking into account the computationally intensive image processing algorithms used to process the captured data, the demand for effective data reduction techniques, such as clustering [1]–[4] and edge detection [5]–[8], is increasing. In this paper we present a general-purpose high-performance 2D-clustering implementation for zero-suppressed input data, which is easily adjustable to a variety of application domains.

Manuscript received May 19, 2014; revised July 02, 2014; accepted October 16, 2014. Date of publication November 06, 2014; date of current version December 11, 2014. This work was supported in part by the Istituto Nazionale di Fisica Nucleare, in part by the Japan Society for the Promotion of Science and MEXT, Japan, under Grant-in-Aid for Scientific Research, and in part by the European Community FP7 People Grant FTK 324318 FP7-PEOPLE-2012-IAPP.

C.-L. Sotiropoulou, S. Gkaitatzis, K. Kordas, S. Nikolaidis, and C. Petridou are with the Department of Physics of the Aristotle University of Thessaloniki, Thessaloniki 54124, Greece (e-mail: lsoti@physics.auth.gr; stamatios.gkaitatzis@cern.ch).

A. Annovi and M. Beretta are with the INFN National Laboratory, Italy (e-mail: alberto.annovi@cern.ch, matteo.beretta@lnf.infn.it).

P. Giannetti and P. Luciano are with INFN, Pisa Section, Pisa 56127, Italy (e-mail: paola.giannetti@pi.infn.it; pierluigiluciano@gmail.com).

G. Volpi is with the University of Pisa, Italy (e-mail: guido.volpi@pi.infn.it).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNS.2014.2364183

A review of most clustering algorithms is presented in [1] by Casagrande *et al.* These are iterative algorithms where data are clustered with respect to an objective function. These algorithms are suitable for data mining but they are too complicated to be implemented in hardware for high data throughput applications. Hussain [9] presents a K-means clustering field programmable gate array (FPGA) implementation that requires 0.723 ms to group a 2905-point dataset to eight clusters. Shanthi [10] demonstrates an FPGA implementation of a histogram based clustering algorithm. This implementation executes segmentation of a 512×512 pixel input in a few seconds. FPGAs have been also used as hardware accelerators for image segmentation processes executed by a CPU, such as in [11], where the FPGA is only used to speed up specific mathematic functions. In [12] an image segmentation approach using logarithmic arithmetic is again implemented on an FPGA device, but the performance results are inconclusive because of lack of available FPGA area for the complete system. Yamaoka *et al.* [13] present a pattern matching implementation architecture on an FPGA for input images of 80×60 pixels. The implementation uses an image segmentation cell-network for a region-growing algorithm. The principle behind this network has similar characteristics with our approach. However, the cell-network in [13] is used only to extract the size of a previously identified cluster, while in our implementation a cluster identification window (grid) is used to detect the cluster itself. All the above algorithmic implementations target unprocessed data with common properties and have a low throughput because of iterative data processing. In the High-Energy Physics application domain, clustering is used to do a first-level data filtering such as in [14]. Gregerson *et al.* use two 2×2 clusters on adjacent towers from the cylindrical CMS detector to identify energy patterns that exceed a desired energy threshold on the CMS calorimeter. The pixels are not zero-suppressed and the cluster size and shape is fixed. Pattern matching is executed on the deposited energy values. A universal clustering engine for zero-suppressed pixel data is presented by Wassatsch and Richter [15] for the Belle II experiment.

The clustering algorithm and implementation presented in this work is designed to process data that are zero-suppressed. The algorithm finds clusters built from all non-suppressed pixels that are connected together sharing a side (first-order neighborhood) or sharing a corner (second-order neighborhood). The output of the clustering algorithm will be same as a single-linkage clustering [16] with a maximum allowed merging-distance of $\sqrt{2}$. The presented implementation is an evolution of a sliding clustering algorithm [2]. It uses an innovative moving

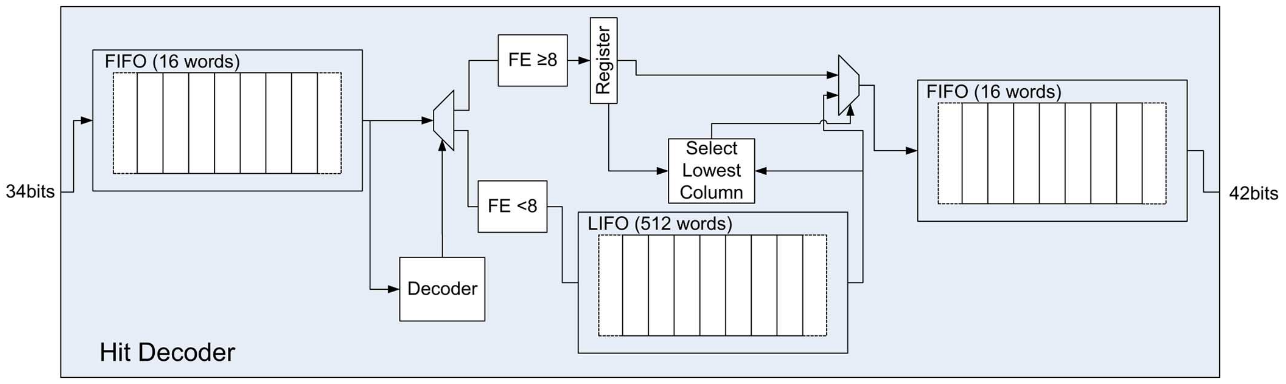

Fig. 1. The 2D-clustering hit decoder module.

window approach to reduce the FPGA resources required for the cluster identification process.

II. Pixel Clustering For The ATLAS Fast TracKer

The clustering implementation presented here is part of the Fast TracKer processor (FTK) [17]. FTK is an approved ATLAS [18] upgrade. The ATLAS experiment captures the outcome of proton-proton collisions occurring every 25 ns at its center with 100 million detector elements arranged in cylindrical layers around the collision point. The data acquired by these elements every 25 ns are called an “event”. The Level-1 trigger of ATLAS accepts events at a rate of 100 kHz and FTK has to provide a complete list of tracks to the ATLAS High-Level Trigger (HLT) within $\sim 100\ \mu$s.

The FTK will receive data from the pixel and microstrip detectors read out drivers (RODs) over 380 S-LINKs [19] running at 2.0 Gb/s and, thus, the total input rate will be 760 Gb/s. Since the hits from the silicon detectors need to be processed by subsequent algorithms, an early reduction of data optimizes the FTK processing. The natural data reduction replaces a cluster of contiguous hits with the position of the cluster centroid and the size of the cluster. This is the target of the proposed 2D-clustering implementation. The hard real-time requirement is achieved by a multicore parallel architecture that can be implemented on the available FPGA devices.

The presented 2D-clustering implementation receives data from ATLAS pixel detector modules. The ATLAS pixel module [20] has 144×328 pixels. In worst-case conditions the occupancy is about 0.4% pixels above the signal-to-noise threshold. Data from a pixel module is equivalent to an image frame and is processed independently of other modules. Thanks to the zero suppression and depending on occupancy conditions, each input link will deliver pixel module data at rate from 0.7 to 2.6 MHz. Our implementation performs clustering at full speed processing all input data on the fly.

The 2D-clustering is implemented on the FTK Input Mezzanine cards (FTK_IM) which are installed on the FTK Data Formatter boards [21][22]. On each FTK_IM card there are two Xilinx Spartan-6 LX150 T FPGAs [23] that receive 2 S-LINKs each, one from a microstrip ROD [24] and one from a pixel ROD [20]. The latter transfers hits as 32 bit words at 40 MHz rate. Each ROD will transmit data of level-1 accepted events in sequence. The event data for one ROD contain the hits of the detector modules connected to that ROD. The main challenge of the clustering implementation is to achieve the required 40 MHz hit processing rate for each S-LINK. We will evaluate the implementation performance using simulated Monte Carlo data for 80 overlapping proton-proton collisions that correspond to the maximum LHC luminosity foreseen until 2022 [25]. For ATLAS all overlapping collisions produce data simultaneously and are seen as one event.

III. Clustering Implementation

The 2D-clustering implementation will identify groups of contiguous pixels compatible with a cluster and then reduce the hit data to a single set of coordinates: the cluster centroid plus a few bits of cluster shape and size information. The data are received by an S-LINK decoder and are forwarded to a FIFO, which is the source of data for the clustering implementation presented here. Each hit is a 32 bit word which carries the hit coordinates (column, row) and the Time-Over-Threshold (ToT) value that contains the deposited charge information.

The clustering implementation is designed with a pipeline of three separate modules: a) the hit decoder module; b) the grid clustering module; c) the centroid calculation module.

A. Hit Decoder Module

The hit decoder (Fig. 1) transforms the incoming data from the ATLAS raw data format to a format useful to the following processing step. It is a pre-processing step that selects, formats and organizes the information that is used by the clustering algorithm such as start/end event words (the flag words that mark the beginning and the end of an event in the bytes stream), module headers/trailers (the flag words that mark the beginning and the end of hits from one pixel module as well as the module number) and of course the pixel hits. This module is robust against bit errors in the input data and it is tolerant to errors in control words.

The most important role of the hit decoder module is to properly align all the incoming data. Each ATLAS pixel module is read out by 16 Front End chips (FE). The FE chips are organized in two rows of 8 chips each. The data from each chip are read out in column pairs. The data coming from each column pair are not geographically sorted. FE chips are numbered from the bottom right corner to the upper right corner in a clockwise cycle and they are read out in the same sequence. This leads to half of the pixel module data arriving in reverse column order than the other half. The hit decoder module needs to restore the column pair order of the hits since the grid clustering algorithm is based on the sole assumption that the hits arrive sorted by column pair.

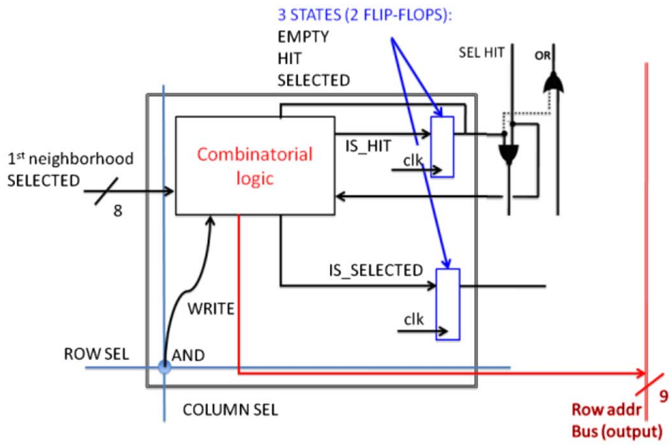

Fig. 2. Elementary pixel cell.

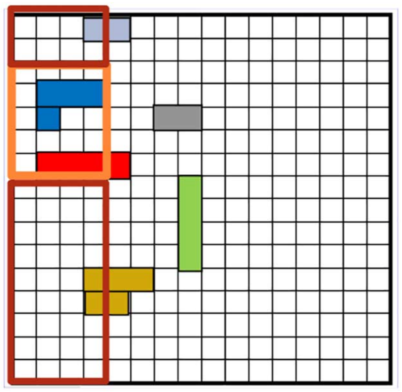

Fig. 3. Discrimination between hits that belong to the "window" and hits that do not. The hits in the darker colored boxes are loaded in the circular buffer.

There is no requirement on the order within the column pair because the grid clustering module can compensate for it.

To reverse the hit sequence a LIFO is used to store all the hits that arrive from FEs with numbers from 0 to 7 (Fig. 1). When a hit arrives from a FE chip with number from 8 to 15 it is stored in a separate register. The value of the register is compared with the last value stored in the LIFO and the hit with the smallest column value is propagated to the next processing module. In this way increasing column sequence is restored. The LIFO size is 512 words, which is largely sufficient for the expected hit occupancy in the pixel modules for up to 80 overlapping proton-proton collisions. In the rare case that LIFO size is exceeded, the logic will empty the LIFO completely and then continue to process incoming data normally. The column order would then be broken, but still valid for the two blocks of data before and after the LIFO full. This would cause only the split of those few clusters that have hits in both data blocks. This condition will be recorded in an end-event-word error flag. Two small FIFOs (16 words each) are added as input and output buffering stages for synchronization purposes.

B. Grid Clustering Module

The grid clustering module is the one that actually identifies the clusters and it is the most computationally intensive block of the implementation. The module uses a "moving window", which is actually a rectangular grid of pixel cells of generic size. Its size depends on the maximum expected cluster size per application with the requirement to be large enough to fit this cluster size in both dimensions. The pixel cell (Fig. 2) is a Finite State Machine cell with three possible states ("empty", "hit" and "selected") and is directly connected to all neighboring cells. The "window" is "moving" in the sense that during the several passes of the cluster identification process it is virtually placed on different coordinates on the pixel module and it is filled every time with data from different areas of the pixel module plane.

This module works properly under the assumption that input data is sorted by column pair. The grid logic processes data from two sources: an input FIFO that receives the hits from the hit decoder and a circular buffer. Since incoming hits are sorted by column pair, the grid and the circular buffer always store data with column number smaller or equal to that of the hits that are still in the FIFO. When the grid clustering module starts processing the first data from a new pixel module, the circular buffer is empty.

The clustering identification procedure is the following:

Reference hit selection: The identification of a cluster starts with the selection of a reference hit. This hit must be the leftmost available hit. When the circular buffer is empty the hit from the input FIFO is selected as reference. Otherwise the leftmost hit stored in the circular buffer is selected.

Grid placement: The grid is virtually aligned to have the reference hit placed on either column 0 or column 1 of the window. The alignment column depends on whether the hit belongs to an even or odd column of the pixel module. This is due to the double column scrambling of the data from the pixel modules, to allow for one column space for preceding hits arriving later.

Grid loading from the circular buffer: Hits are first read from the circular buffer once. They are loaded in the grid if their coordinates are within its boundaries. Hits outside the grid are stored again in the circular buffer without using additional clock cycles (Fig. 3). Every time the circular buffer is written the leftmost hit is stored in a separate register. When a hit is stored in a cell, it changes its state from "empty" to "hit".

Grid loading from the input FIFO: Hits from the FIFO are read until a hit with column coordinate beyond the grid's right boundary is found or until an end-of-module word is received. In case the FIFO becomes empty the architecture waits for data. Identically to the hits loaded from the circular buffer, the hits from the input FIFO are loaded either into the grid if their coordinates are within its boundaries or in the circular buffer.

At this point in the flow the grid is loaded with all hits that belong to it. In addition, all data that fall outside the grid but within the same column span are temporary stored in the circular buffer.

Cluster identification: The cluster identification process starts by using the reference hit as "seed" [Fig. 4(a): shadowed cell]. The reference hit, which is always in one of two positions on the grid, is changed to "selected" state via a dedicated control line.

Cluster readout: The "selected" state is propagated to the 8 neighboring pixel cells [Fig. 4(b), (c): arrow demonstrating the propagation]. Those that are in "hit" state change to "selected" state. While the "selected" state propagates, the grid readout logic is used to readout "selected" cells [Fig. 4(c), (d): black dotted cells]. The cells that are readout return to the "empty" state [Fig. 4(d): cell marked with horizontal lines].

At this point in the flow, only the cluster that was "seeded" is found and readout. Hits that remain in the grid are not clustered yet.

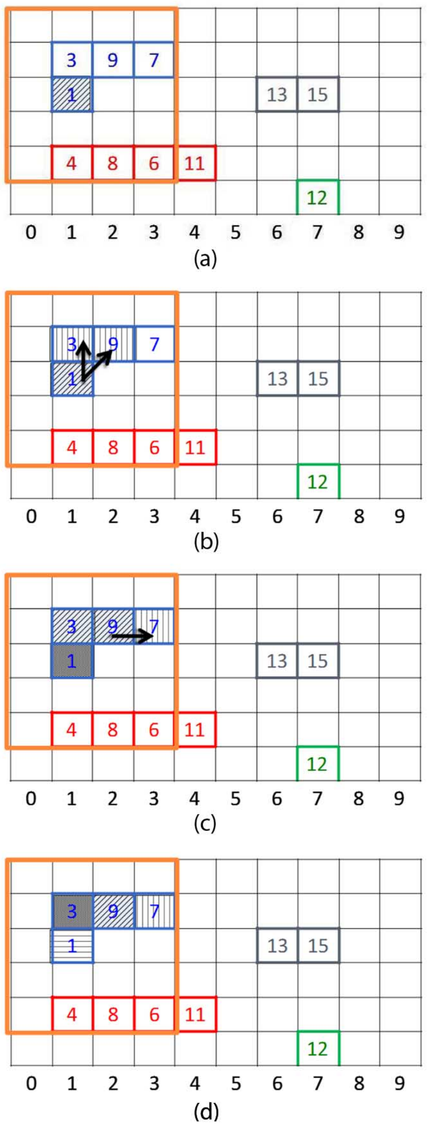

Fig. 4. Cluster read out process.

Hit recovery: The hits that remain in the grid and do not belong in the cluster are readout and stored in the circular buffer ready for the next processing step.

After one cluster is found the process is restarted.

For the current 2D-clustering module implementation a "window" of 8 $\times$ 21 pixels is used, 8 for the z or r direction and 21 for the $r-\varphi$, ATLAS uses a right-handed coordinate system with its origin at the nominal interaction point (IP) in the centre of the detector and the z-axis along the beam pipe. The x-axis points from the IP to the centre of the LHC ring, and the y-axis points upwards. Cylindrical coordinates (r,φ) are used in the transverse plane, φ being the azimuthal angle around the beam pipe. Most clusters recorded by the ATLAS pixel module (95%) fit within a box of 5 columns and 6 rows. The bigger grid is used to allow identification of the rare large clusters and clusters generated by merging hits from two or more clusters. Clusters bigger than the grid size, i.e., clusters extending from the reference hit beyond one of the grid edges, will be split. Clusters that touch a grid edge will be identified by a flag in the output, called the "split flag". The "split flag" bit is a dedicated bit in the cluster flag word.

C. Centroid Calculation Module

The centroid calculation module is the post-processing step in the 2D-clustering implementation that performs the data reduction process. It is the module where the cluster data are replaced with one set of coordinates, the centroid coordinates. For each cluster a centroid value is calculated. The centroid value is calculated as the center of each cluster's bounding box. The pixels of the ATLAS pixel modules have two different sizes on the x axis: the pixels at the edges of the FEs are 600 μm long, while all others are $400-\mu$m long. The centroid calculation module corrects this difference by calculating the centroid coordinates in normalized units of 25 μm for the x axis and 6.25 μm for the y axis. The divisions required for the normalization process are implemented in a Look-Up-Table (LUT).

The centroid can be corrected taking into account the charge deposition in each hit measured by the Time-over-threshold (ToT) information. This is done to replicate the ATLAS offline code for centroid calculation that uses charge interpolation described in detail in [26].

The post-processing step can be tailored on the application (e.g., center-of-mass [8], median calculation [27], etc.).

IV. Parallel Implementation

One fundamental characteristic of this 2D-clustering implementation is that different clustering engines can work independently and in parallel on data from different modules, therefore, increasing performance by exploiting more FPGA resources. It has been noted that the pixel data arrive through an S-LINK. Additionally, the Data Formatter board also expects data through a single data stream [21][22]. Choosing the appropriate parallelization strategy is critical for the implementation's performance.

The parallelization strategy chosen for the presented implementation is to instantiate multiple clustering engines (grid clustering modules) that work independently on data from separate pixel modules (Fig. 5). The data acquired by each detector module can be considered an independent image because clusters are entirely contained in a single module. To achieve this, data parallelizing (demultiplexing) and data serializing (multiplexing) logic modules are necessary. There are two issues that require special attention. The execution cycles required for the clustering identification process are data dependent. This leads to unequal processing time per pixel module per clustering engine. Additionally, the recovered single data flow after the parallel processing must be in the correct event order. To tackle these issues a special parallel data distributor module and a data merger module were designed.

A. Parallel Data Distributor Module

The pixel data arrive in a single data stream, packed by event header and event trailer control words. The event header at the input of the 2D clustering code is a single word made of the ATLAS Level 1 ID number (event number) with a specific start event flag. Within each event the pixel module data are packed within module header and module trailer control words. The processing time of one engine is strongly data dependent and, therefore, it is impossible to predict which parallel clustering engine will finish processing first. In addition, the data merger module must be able to restore the data stream in the received

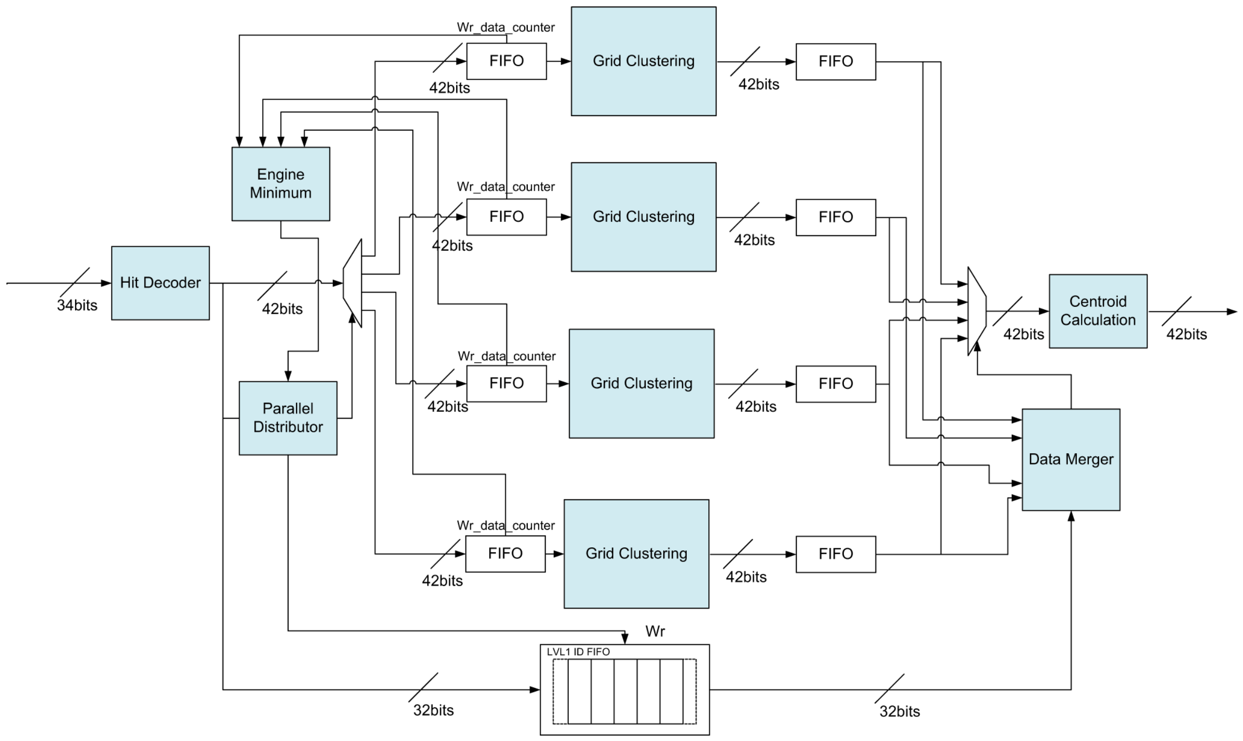

Fig. 5. 2D-clustering implementation parallel engines block diagram for the four-engine case.

event sequence. The parallel data distributor module must propagate through the engines the event and module control words in a way such that the data merger can retrieve the proper order.

Each parallel clustering engine is buffered by two FIFOs at input and output. This is done to facilitate the design routing and temporarily store input and output data while the clustering engine is busy or the data merger has put the clustering engine on hold. The input FIFO from each parallel clustering engine has a write data counter activated so that the parallel data distributor module can monitor which parallel engine has less data queued at the input. As soon as the first event header appears at the input of the parallel data distributor it is written to a LVL1ID FIFO (Level 1 ID FIFO) as a reference for the received event header sequence. The parallel data distributor chooses the clustering engine where all pixel hits for one module will be propagated. On first run, this is the engine with the smallest index number. On all subsequent runs it is the engine with the smallest write data counter value (clustering engine which currently has less data queued at input). To identify the smallest write data counter value a generic binary tree comparator was implemented. The pixel module data are sent packed between the module header and the module trailer words. If there are more than one pixel module data in the event, another clustering engine is chosen by the binary tree comparator that will receive the next pixel module data. Multiple modules can be sent to each engine for each event. For each engine receiving at least one module, the event header is also sent before the first module in order to keep the module-event association. When the event trailer arrives it is sent to all clustering engines that have been assigned data belonging to this event, to close the event for all engines. The parallel distributor assigns to the engines pixel module data for new events while the engines themselves are still processing previous event(s). The event trailer word is made by the LVL1ID and an event trailer specific flag bit.

B. *Data Merger Module*

The data merger module begins its operation as soon as the LVL1ID FIFO and one of the parallel clustering engines output FIFOs have an event header at the output. It then compares the event header with the one at the output of the LVL1ID FIFO and if it matches it will start reading the data. If more than one parallel engine has the same event header then the engine with the lowest index number is chosen (by using a priority encoder). After the data from one pixel module are read out, the data merger checks whether there are any more engines that have the same event header or whether a second pixel module is loaded in the same engine and reads out the corresponding FIFO. While the data merger is busy reading the output FIFO of one parallel engine, the other parallel engines continue normal operation and write their outputs to the corresponding output FIFO, until this FIFO is almost full. When the FIFO becomes almost full, backpressure is applied to the grid clustering module (read out operation is stalled), until the corresponding FIFO has free space for data. When all the parallel clustering engines output FIFOs that had data from the same event have the event trailer at the output the event trailer is read out and the LVL1ID FIFO is read to get the next event header and restart the event reading process. In the extremely rare case when no engines have the same event header as the LVL1ID FIFO this is declared a fatal error, it is flagged in the output error word and the clustering module is restarted.

V. Results

The 2D-clustering implementation has been developed, verified by simulation and tested on the FTK_IM board in both single and parallel versions. Implementation results are presented for both the single and parallel flow versions. The most

TABLE I
2D-CLUSTERING IMPLEMENTATION RESULTS FOR SINGLE FLOW

	FF	LUT	BRAM (18 kbit)	BRAM (9 kbit)	Maximum Frequency (MHz)
Hit Decoder	306	486	1	3	120.0
Grid Clustering	700	2257	-	2	81.5
Centroid Calculation	539	492	-	2	185.0
Total System	1580	3128	1	8	81.5
FF: Flip Flops, **LUT**: Look-Up-Tables, **BRAM**: Block RAMs					

TABLE II
2D-CLUSTERING IMPLEMENTATION RESULTS FOR 4-ENGINE PARALLEL FLOW

	FF	LUT	BRAM (18 kbit)	BRAM (9 kbit)	Maximum Frequency (MHz)
Parallel Data Distributor	51	79	-	-	270.0
Binary Tree Comparator	84	51	-	-	334.8
Data Merger	70	113	-	-	247.5
Total System	5739	10583	15	21	80.5
FF: Flip Flops, **LUT**: Look-Up-Tables, **BRAM**: Block RAMs					

fundamental features of the FPGA fabric (flip-flops, LUTs, and BRAMs) are used as metrics for the implementation size.

A. Single Flow Results

The 2D-clustering single flow implementation (results presented in Table I) occupies 0.9% of the device's FFs, 3.4% of the LUTs, 0.4% of the 18 kb BRAMs and 2.9% of the 9-kb BRAMs. The small differences between the sum of resources of the separate modules and the complete system are due to different routing choices applied by the Xilinx PAR (place and route) tool when the complete system is implemented. The extra 9-kb BRAM belongs to the output FIFO implemented after the grid clustering module. The operational frequency is defined by the grid clustering module's critical path. The centroid calculation module calculates the center of the cluster bounding box in normalized coordinates without taking into account the charge deposition. In addition to the centroid calculation module, an indicative implementation of a center-of-mass calculation is presented in [8] and of a median calculation in [27]. Both implementations are for the same FPGA device and have an operating frequency much higher than 81.5 MHz. The center-of-mass implementation uses 237 FFs and 409 LUTs while the median implementation uses 215 FFs and 307 LUTs for a 32 $\times$ 32 pixel detection window. The center-of-mass, the median calculation and the centroid calculation module have comparable area occupation and shorter critical path than the grid clustering module, therefore, it can be assumed that all three can be used as a final processing step for the clustering implementation.

The 2D-clustering single flow engine has been verified by both behavioral and post place and route simulations and measurements on the actual FTK_IM board. A bit-accurate simulation model of the 2D-clustering has also been developed to verify firmware operation. Post place and route simulations using Monte Carlo files of 80 overlapping proton-proton collisions have demonstrated a worst case estimate of roughly $\sim$ 85 ns processing time per data word. The firmware has been tested on the FTK_IM board where a gigabit fiber was used to deliver the data to the FTK_IM and read out the data from the Data Formatter board. The Data Formatter board is acting as a pass through and a computer is used as data source and data sink. For the board test a 40 MHz clock was used for the input FIFO of the hit decoder and a 80 MHz clock for the rest of the implementation. In all the tests the output results have been confirmed by comparison with the bit-accurate simulation.

The input rate of the pixel clustering system is 40 MHz. This means that at maximum throughput the pixel clustering system will receive one data word every 25 ns. It can be calculated that the ratio of processing time per data word over the input data time (the maximum input data rate for the single flow implementation) is $85\ \text{ns}/25\ \text{ns} = 3.4$. The single flow 2D-clustering implementation is 3.4 times slower than the maximum input rate. Since this is an iterating algorithm a deeper pipeline is not an option. Therefore, a parallel implementation is essential to achieve the required performance. As the single flow implementation is 3.4 times slower than the maximum input data rate it can be anticipated that the parallel engine implementation must have at least 4 clustering engines working in parallel to achieve the required performance.

B. Parallel Flow Results

A four-engine version was implemented first. Then we explored the FPGA resource cost and maximum clock speed of versions with 8 and 16 engines. Each version was designed and verified through the Xilinx validation chain and its performance was measured with post place and route simulation by using worst case ATLAS simulated data. We initially found that the 4 engines did not have enough buffering at the output since the data merger was applying backpressure. We explored the performance as function of the output buffer and found a strong correlation between the size of the output FIFOs on each clustering engine and system performance. A sufficient size for the output FIFO was determined by simulation and testing measurements to be 1024 words for each clustering engine.

This parallel flow version of four parallel clustering engines was implemented and measured on the FTK_IM board. In Table II the implementation results for the 4-engine parallel flow with large buffering are presented. The new modules that were introduced for the parallel design version, the parallel data distributor, the binary tree comparator and the data merger, occupy a very small percentage of FPGA resources with respect to the total system. The total system occupies 3.1% of the device's FFs, 11.5% of the LUTs, 5.6% of the 18-kb BRAMs and 3.5% of the 9-kb BRAMs. It can be seen that the number of BRAMs that are required for the implementation with the four parallel engines is greater than the number required for the single flow when multiplied by four because of the extra required buffering at the input and at the output of each parallel clustering engine (FIFOs before and large FIFOs after Grid Clustering modules in Fig. 5).

The clock performance is again defined by the grid clustering module and the maximum frequency drops a little due to routing to 80.5 MHz. The 4-engine parallel clustering flow has been tested on the FTK_IM board. The board test was executed with the same clock configuration as with the single flow, 40 MHz clock for the input FIFO and 80 MHz for the rest of the implementation. By using the same input data as with the single

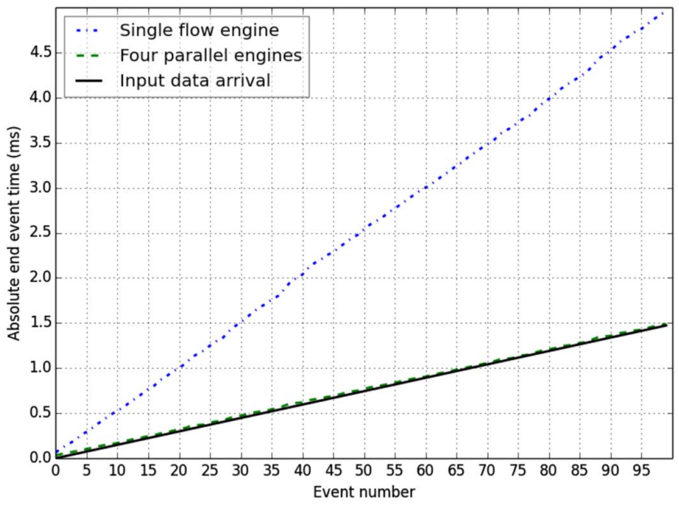

Fig. 6. Performance plot for the 2D-Pixel clustering implementation. Each line represents the transit time of the last event word. The black continuous line is input time. The blue dash dotted line is output time for the single flow. The green dashed line is output time for the 4 parallel flow.

flow clustering implementation a worst case of $\sim$ 25 ns processing time per data word was estimated. This performance covers the given specifications of maximum input data rate of 40 MHz. In Fig. 6 a performance plot for the 2D-pixel clustering implementation is presented. For this performance plot simulation data from a hard scattering process (ZH $\rightarrow \nu\bar{\nu}b\bar{b}$) overlapping with 80 pile up proton-proton collisions were used. On the x-axis the event number is presented, while on the y-axis the absolute time an event exited the 2D-clustering implementation (end_event output time). The continuous black line is the reference line that demonstrates the maximum input data rate, one word every 25 ns. As it can be seen the single flow (dash-dot blue line) cannot follow the input rate with a much higher slope and a processing time of $\sim$ 85 ns per data word. The 4 parallel engine flow (dashed green line) is almost identical to the reference line, demonstrating that it can fully respond to the maximum input data rate. The above performance results demonstrate that 4 parallel engines with large buffering are sufficient for the ATLAS detector pixel modules. From these 100 events a total of 21050 clusters were identified out of 58890 data words (hits and control words). The results have been confirmed by bit-accurate simulation.

A parallel version with 16 engines was also integrated for potential more computationally demanding applications (such as the Insertable B-Layer [28]). The goal of this implementation is to demonstrate feasibility and to observe the achievable clock speed. The implementation results for the parallel 2D-clustering version with 16 clustering engines are presented in Table III. This system version is implemented with output FIFOs of 128 words as the buffering requirements will be recalculated after extensive testing with IBL data. The implementation uses 9.5% of the device's FFs, 38.8% of the device's LUTs, 6.3% of the 18-kb BRAMs and 16.4% of the 9-kb BRAMs. As this implementation uses a significant percentage of the FPGA device resources and has modules with a 16 times bus fan out, it is important to keep the critical path buffered in order to avoid performance degradation. With the current parallel configuration a 80 MHz clock frequency was achieved.

TABLE III
2D-Clustering Implementation Results for 16-Engine Parallel Flow

Slice Type	Used Slices	Total Slices	Used Slices(%)
FFs	17558	184304	9.5
LUTs	35724	92152	38.8
BRAMs (18 kbit)	17	268	6.3
BRAMs (9 kbit)	88	536	16.4
FF: Flip Flops, **LUT**: Look-Up-Tables, **BRAM**: Block RAMs			

TABLE IV
Extrapolated Results for Sliding Clustering Algorithm

Grid Size	FFs	LUTs	Maximum Frequency (MHz)
8 x 21 (current)	700	2257	81.5
8 x 328	10933	35252	7.4
FF: Flip Flops, **LUT**: Look-Up-Tables			

The current implementation is an evolution of a sliding clustering algorithm that had a much higher cost in terms of FPGA resources [2]. In the sliding clustering algorithm a grid of 8×328 pixels was considered and it would have a comparable fraction of split clusters. The grid clustering module of the old sliding algorithm version has been re-implemented on a Spartan-6 LX150 T device for direct comparison with the current implementation. The extrapolated FPGA resources and clock frequency results are presented in Table IV. Accounting for the fact that the processing time of the sliding window algorithm would be on average 3 clock cycles per hit, we can determine that our implementation reduces the required logic by a factor approximately 64.

The presented implementation works on zero-suppressed data, therefore, it cannot be directly compared with the previously described algorithms in [9]–[14]. Wassatsch and Richter [15] have presented a clustering implementation targeting zero-suppressed data. In their approach the whole pixel frame has to be read before the clusters can be identified, which imposes a significant algorithmic delay. From the imposed delay derives a hard requirement for a 400 MHz clock that required the implementation of the algorithm on ASIC technology (TSMC 65 nm). A direct area comparison cannot be made between [15] and our proposed implementation, but an estimated 10^6 logic gates are needed for the Wassatsch and Richter implementation, while around $15 \cdot 10^3$ logic cells are used in our design on a Spartan-61 $\times$ 150t FPGA device. An important difference between the two implementations is that the Wassatsch and Richter implementation does not have a cutoff on the cluster size, while the presented clustering algorithm does. The cutoff is not a limitation since for the typical HEP pixel detector meaningful clusters are small, e.g., smaller than 7×7[26].

C. Statistics

The outputs of the 2D-clustering implementations have been verified by a bit-accurate simulation model. Using the same model as well as the outputs of the FPGA firmware statistics for the 2D-clustering performance were gathered. The most important aspect of these statistics is the percentage of the identified clusters that have an active "split flag" with the current 8×21 pixel detection window specifications. The "split flag" is active whenever a cluster touches the grid (detection window) edge. This is an indication, but not a confirmation, that the cluster might be split. It was estimated that at most $\sim$ 1.5% of all clusters will be flagged as split. Not all clusters with the "split flag" are actually split. It is possible to reduce the fraction of split clusters using a larger pixel clustering grid. For example a 12×31 grid would reduce the fraction of potentially split clusters to $\sim 0.4\%$. Our design is totally generic, therefore, adjusting the size is easily achievable. An implementation with this larger grid was realized. The FPGA maximum clock speed is reduced to 65 MHz (along with the fact that the number of cycles required to process a hit increases to 4.7–by reference to the 40 MHz input clock). This means that 8 engines can fit on the Spartan-6 and process pixel data at full input speed, by occupying 43% FPGA resources (LUTs). We have chosen the 8×21 grid size, that doesn't split any cluster of size up to 7×11, since larger clusters are not of interest. They originate mostly from tracks that are not from beam collision or from low transverse momentum particles [26] that are not reconstructed by the FTK. Furthermore, what matters is the fraction of split clusters for tracks with transverse momentum above 1 GeV/s. The effect on tracking performance is under study.

VI. Conclusions and Future Developments

A multi-core FPGA-based 2D-pixel clustering implementation is presented. The implementation targets the ATLAS Fast TracKer but it is generic enough to be used in various images processing applications where pixel clustering is required. The implementation targets zero-suppressed data and uses a moving window technique to reduce the FPGA resources required for the cluster identification process. The proposed implementation is an improvement by a large factor over previous approaches to zero-suppressed data clustering, as it uses 64 times less logic resources in comparison to [2]. The innate flexibility of the implementation is that different clustering engines can work independently and in parallel to improve performance while the post processing step can be adapted to different applications' needs. Therefore, the proposed implementation can be used for various applications where 2D clustering of zero-suppressed data is required with different performance requirements. One potential application of this algorithm is the 2D-clustering for the Insertable B-Layer (IBL) of the ATLAS Inner Detector with a maximum pixel hit output rate of 100 MHz. A version with 4 parallel engines is currently implemented in the FTK system while a 16 engine implementation is proposed for the IBL detector.

References

[1] D. Casagrande, M. Sassano, and A. Astolfi, "Hamiltonian-based clustering: Algorithms for static and dynamic clustering in data mining and image processing," *IEEE Control Syst.*, vol. 32, no. 4, pp. 74–91, Aug. 2012.

[2] A. Annovi *et al.*, "A fast FPGA-based clustering algorithm for real time image processing," in *IEEE Nuclear Science Symp. (NSS/MIC) Conf. Rec.*, Oct.-Nov. 24–1, 2009-2009, pp. 4138–4141.

[3] N. N. Gopal and M. Karnan, "Diagnose brain tumor through MRI using image processing clustering algorithms such as fuzzy C means along with intelligent optimization techniques," in *Proc. IEEE Int. Conf. Computational Intelligence and Computing Research (ICCIC)*, 2010, pp. 1–4.

[4] S. N. Sulaiman and N. A. M. Isa, "Adaptive fuzzy-K-means clustering algorithm for image segmentation," *IEEE Trans. Consum. Electron.*, vol. 56, no. 4, pp. 2661–2668, Nov. 2010.

[5] X. Fu, H. You, and K. Fu, "A statistical approach to detect edges in SAR images based on square successive difference of averages," *IEEE Geosci. Remote Sens. Lett.*, vol. 9, no. 6, pp. 1094–1098, Nov. 2012.

[6] M. Diwakar, P. K. Patel, and K. Gupta, "Cellular automata based edge-detection for brain tumor," in *Proc. Int. Conf. Advances in Computing, Communications and Informatics (ICACCI)*, 2013, pp. 53–59.

[7] C. Gentsos, C. L. Sotiropoulou, S. Nikolaidis, and N. Vassiliadis, "Real-time canny edge detection parallel implementation for FPGAs," in *Proc. 17th IEEE Int. Conf. Electronics, Circuits, and Systems*, 2010, pp. 499–502.

[8] C.-L. Sotiropoulou, L. Voudouris, C. Gentsos, A. M. Demiris, N. Vassiliadis, and S. Nikolaidis, "Real-time machine vision FPGA implementation for microfluidic monitoring on lab-on-chips," *IEEE Trans. Biomed. Circuits Syst.,*, vol. 8, no. 2, pp. 268–277, Apr. 2014.

[9] H. M. Hussain, K. Benkrid, A. T. Erdogan, and H. Seker, "Highly parameterized K-means clustering on FPGAs: Comparative results with GPPs and GPUs," in *Proc. 2011 Int. Conf. Reconfigurable Computing and FPGAs (ReConFig)*, 2011, pp. 475–480.

[10] K. Shanthi, L. Ashok, A. Anandu, and B. Gokul Das, "FPGA implementation of image segmentation processor," in *Proc. 2nd Int. Conf. Emerging Trends in Engineering and Technology (ICETET)*, 2009, pp. 364–367.

[11] S. Klupsch, M. Ernst, S. A. Huss, I. S. und Systeme, M. Rumpf, and R. Strzodka, "Real time image processing based on reconfigurable hardware acceleration," presented at the Workshop Heterogeneous Reconfigurable Systems on Chip (SoC), 2002.

[12] R. Bannister, D. Gregg, and A. Simon Wilson, "FPGA implementation of an image segmentation algorithm using logarithmic arithmetic," in *Proc. 48th Midwest Symp. Circuits and Systems*, 2005, vol. 1, pp. 810–813.

[13] K. Yamaoka, T. Morimoto, H. Adachi, T. Koide, and H. J. Mattausch, "Image segmentation and pattern matching based FPGA/ASIC implementation architecture of real-time object tracking," in *Proc. Asia and South Pacific Conf. Design Automation*, 2006, p. 6.

[14] A. Gregerson *et al.*, "FPGA design analysis of the clustering algorithm for the CERN large hadron collider," in *Proc. 17th IEEE Symp. Field Programmable Custom Computing Machines, FCCM'09*, 2009, pp. 19–26.

[15] A. Wassatsch and R. Richter, "DCE3-An universal real-time clustering engine," in *Proc. 2012 IEEE Int. Symp. Circuits and Systems (ISCAS)*, 2012, pp. 3242–3245.

[16] R. Sibson, "SLINK: An optimally efficient algorithm for the single-link cluster method," *The Comput. J.*, vol. 16, no. 1, pp. 30–34, 1973.

[17] A. Andreani *et al.*, "The fasttracker real time processor and its impact on muon isolation, tau and b-jet online selections at ATLAS," *IEEE Trans. Nucl. Sci.*, vol. 59, no. 2, pp. 348–357, Apr. 2012.

[18] The ATLAS Collaboration, "The ATLAS experiment at the CERN large hadron collider," *J. Instrum.*, vol. 3, no. S08003, 2008.

[19] E. Van der Bij, R. McLaren, and Z. Meggyesi, "S-LINK: A prototype of the ATLAS read-out link," [Online]. Available: https://cds.cern.ch/record/405075

[20] G. Aad *et al.*, "ATLAS pixel detector electronics and sensors," *J. Instrum.*, vol. 3, no. P07007, Jul. 2008.

[21] The ATLAS Collaboration, Fast TracKer (FTK) Tech. Design Rep. ATLAS-TDR-021, CERN-LHCC-2013-007 [Online]. Available: https://cds.cern.ch/record/1552953

[22] J. Olsen, T. Liu, and Y. Okumura, "A full mesh ATCA-based general purpose data processing board," *J. Instrum.*, vol. 9, no. 01, p. C01041, 2014.

[23] Xilinx Inc., "Spartan-6 family overview," [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf

[24] P. Haefner, "The ATLAS silicon microstrip tracker. Operation and performance," *J. Instrum.*, vol. 5, no. 12, p. C12050, 2010.

[25] The ATLAS Collaboration, Tech. Design Rep. for the Phase-I Upgrade of the ATLAS TDAQ System ATLAS-TDR-023 [Online]. Available: https://cds.cern.ch/record/1602235

[26] The ATLAS collaboration, and others, "A neural network clustering algorithm for the ATLAS silicon pixel detector," arXiv preprint arXiv:1406.7690 2014.

[27] C.-L. Sotiropoulou, C. Gentsos, and S. Nikolaidis, "High performance median FPGA implementation for machine vision applications," in *Proc. 20th IEEE Int. Conf. Electronics, Circuits, and Systems*, 2013, pp. 143–176.

[28] The ATLAS Collaboration, ATLAS insertable B-layer Tech. Design Rep. ATLAS-TDR-19, CERN-LHCC-2010-013 [Online]. Available: https://cds.cern.ch/record/1552953