FPGA Co-processor for the ALICE High Level Trigger

G. Grastveit[1], H. Helstrup[2], V. Lindenstruth[3], C. Loizides[4], D. Roehrich[1], B. Skaali[5],
T. Steinbeck[3], R. Stock[4], H. Tilsner[3], K. Ullaland[1], A. Vestbo[1] and T. Vik[5]
for the ALICE Collaboration

[1] *Department of Physics, University of Bergen, Allegaten 55, N-5007 Bergen, Norway*
[2] *Bergen University College, Postbox 7030, N-5020 Bergen, Norway*
[3] *Kirchhoff Institut für Physik, Im Neuenheimer Feld 227, D-69120 Heidelberg, Germany*
[4] *Institut für Kernphysik Frankfurt, August-Euler-Str. 6, D-60486 Frankfurt am Main, Germany and*
[5] *Department of Physics, University of Oslo, P.O.Box 1048 Blindern, N-0316 Oslo, Norway*

The High Level Trigger (HLT) of the ALICE experiment requires massive parallel computing. One of the main tasks of the HLT system is two-dimensional cluster finding on raw data of the Time Projection Chamber (TPC), which is the main data source of ALICE. To reduce the number of computing nodes needed in the HLT farm, FPGAs, which are an intrinsic part of the system, will be utilized for this task. VHDL code implementing the *fast cluster finder* algorithm, has been written, a testbed for functional verification of the code has been developed, and the code has been synthesized.

1. Introduction

The central detectors of the ALICE experiment [1], mainly its large Time Projection Chamber (TPC) [2], will produce a data size of up to 75 MByte/event at an event rate of up to 200 Hz resulting in a data rate of ∼15 GByte/sec. This exceeds the foreseen mass storage bandwidth of 1.25 GByte/sec by one order of magnitude. The High Level Trigger (HLT), a massive parallel computing system, processes the data online doing pattern recognition and simple event reconstruction almost in real-time, in order to select interesting (sub)events, or to compress data efficiently by modeling techniques [3], [4]. The system will consist of a farm of clustered SMP-nodes based on off-the-shelf PCs connected with a high bandwidth low latency network. The system nodes will be interfaced to the front-end electronics via optical fibers connecting to their internal PCI-bus, using a custom PCI receiver card for the detector readout (RORC) [5].

Such PCI boards, carrying an ALTERA APEX FPGA, SRAM, DRAM and a CPLD and FLASH for configuration have been produced and are operational. Most of the local pattern recognition is done using the FPGA co-processor while the data is being transferred to the memory of the corresponding nodes.

Focusing on TPC tracking, in the conventional way of event reconstruction one first calculates the cluster centroids with a *Cluster Finder* and then uses a *Track Follower* on these space points to extract the track parameters [6], [7]. Conventional cluster finding reconstructs positions of space points from raw data, which are interpreted as the crossing points between tracks and the center of padrows. The cluster centroids are calculated as the weighted charge mean in pad and time direction. The algorithm typically is suited for low multiplicity data, because overlapping clusters are not properly handled. By splitting clusters into smaller clusters at local minima in time or pad direction a simple deconvolution scheme can be applied and the method becomes usable also for higher multiplicities of up to dN/dy of 3000 [3].

To reduce data shipping and communication overhead in the HLT network system, most of the cluster finding will be done *locally* defined by the readout granularity of the readout chambers. The TPC front-end electronic defines 216 data volumes, which are being read out over single fibers into the RORC.

We therefore implement the *Cluster Finder* directly on the FPGA co-processor of these receiving nodes while reading out the data. The algorithm is highly local, so that each sector and even each padrow can be processed independently, which is another reason to use a circuit for the parallel computation of the space points.

2. The Cluster Finder Algorithm

The data input to the *cluster finder* is a zero suppressed, time-ordered list of charge values on successive pads and padrows. The charges are ADC values in the range 0 to 1023 drawn as squares in figure 1. By the nature of the data readout and representation, they are placed at integer time, pad and padrow coordinates.

Finding the particle crossing points breaks down into two tasks: Determine which ADC values belong together, forming a cluster, and then calculate the center of gravity as the weighted mean using

$$G_P = \sum q_i p_i \Big/ Q \qquad (1)$$

$$G_T = \sum q_i t_i \Big/ Q\,, \qquad (2)$$

where G_T is the center of gravity in the time-direction,

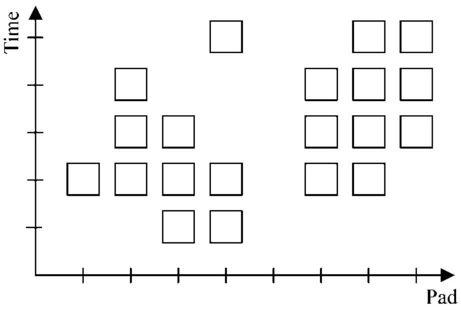

Figure 1: Input charge values in the time-pad plane

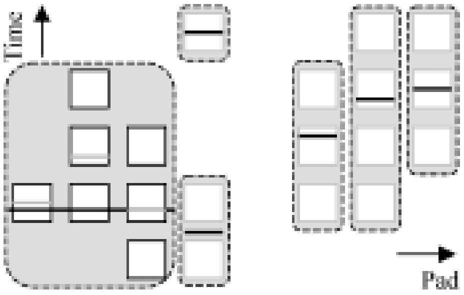

Figure 2: Grouping input data into sequences, and merging neighboring sequences into clusters.

G_P in the pad direction, $Q = \sum q_i$ is the total charge of the cluster.

2.1. Grouping charges into sequences

Because of the ALTRO[8] data format, we choose to order the input data first along the time direction into *sequences*, before we merge sequences on neighboring pads into clusters.

A sequence is regarded as a continuous (vertical) stack of integer numbers. They are shown in figure 2. For each sequence the center of gravity is calculated using eqn. (2). The result is a decimal number. This is illustrated by the horizontal black lines.

2.2. Merging neighboring sequences

Searching along the pad direction (from left to right in the figures), sequences on adjacent pads are merged into a starting cluster if the difference in the center of gravity is less than a selectable *match distance* parameter[1]. The center of gravity value of the last appended (rightmost) sequence is kept and used when searching for matches on the following pad, also shown in figure 2.

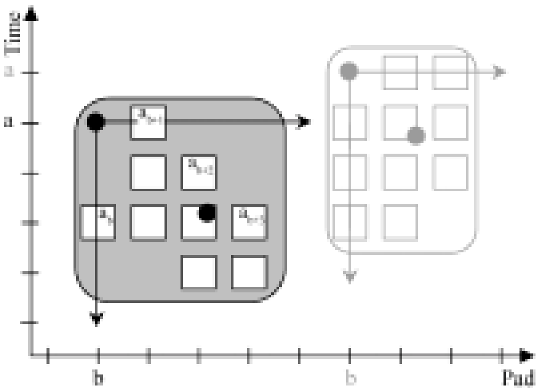

Figure 3: Finished clusters in the time-pad plane, where the black dots in the middle mark the centroid. a and b specify the absolute position of the cluster, a_i are the time positions of the sequences that form the cluster.

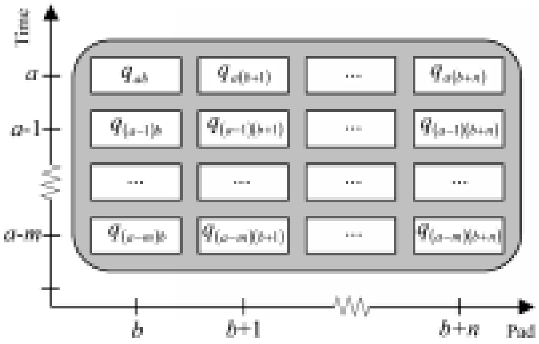

Figure 4: The definition of a general cluster used to simplify the algorithm for the FPGA

A cluster that has no match on the next pad is regarded as finished. If it is above the noise threshold (requirements on size and total charge), the final center of gravities in time and pad according to eqns. (1) and (2) are calculated. This is –besides other definitions– illustrated in figure 3. The space-point, given by these two values transformed into real space, and the total charge of the cluster are then used for tracking [6].

3. Algorithm adaptions for VHDL

To transform the algorithm into a version which is suitable for a computation in hardware, we define a general cluster as a matrix in the time-pad plane according to figure 4.

Of course, clusters can be any size and shape and can be placed anywhere within the time-pad plane. The upper left corner is chosen as the reference point

---

[1] Typically we set the match distance parameter to 2.

since this charge will enter the cluster finder circuit first. The values a and b define the absolute placement of the corner, the size of the cluster is given by m and n. Allowing charge values to be zero covers varying shapes, but since only one sequence per pad is merged, zero charge values in the interior of the clusters can not occur.

Using the general cluster definition, the total charge Q is the sum of the total sequence charges Q_j according to

$$Q = \sum_{j=b}^{b+n} Q_j \quad \text{where} \quad Q_j = \sum_{i=a-m}^{m} q_{ij} \tag{3}$$

and the formulas for the center of gravity change into

$$\begin{aligned} G_P &= \sum_{j=b}^{b+n} jQ_j \bigg/ Q \qquad (4) \\ G_T &= \sum_{j=b}^{b+n} \sum_{i=a-m}^{a} iq_{ij} \bigg/ Q \qquad (5) \end{aligned}$$

By calculating the center of gravity relative to the upper left corner of the cluster, the multiplicands i and j in (4) and (5), which start counting from a and b respectively, are exchanged by indexes starting at 0. Using the definitions (3) and figure 3, we get for the center of gravities the following two equations:

$$\begin{aligned} G_P &= b + \sum_{k=0}^{n} kQ_{b+k} \bigg/ Q \qquad (6) \\ G_T &= a - \sum_{j=b}^{b+n} \sum_{k=0}^{m} kq_{(a_j-k)j} + (a-a_j)Q_j \bigg/ Q \qquad (7) \end{aligned}$$

These formulas are better suited for the FPGA implementation, because we have reduced the number of multiplications, and for the remaining the range of one of the multiplicands k and $(a - a_j)$ is restricted. It is kept within the range of the height and width of a cluster. When the relative centroid has been determined, it has to be transformed to the absolute coordinate system (using a and b). Thus, the FPGA implementation has to calculate a, b, Q and the result of the two sums. Per cluster these 5 integer values will be sent to the host, which in turn uses (6) and (7) for the final result.

4. VHDL implementation

The FPGA implementation has four components as shown in figure 5. The two main parts are the *Decoder*, which groups the charges into sequences (section 2.1), and the *Merger*, which merges the sequences into clusters (section 2.2).

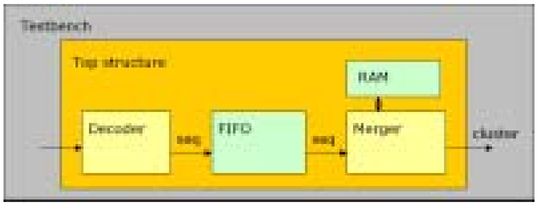

Figure 5: The block diagram of the FPGA cluster finder

4.1. Decoder

The *Decoder* handles the format of the incoming data and computes properties of the sequences. We see from (7) that two parts are "local" to every sequence. These are Q_j and $\sum_{k=0}^{m} kq_{(a_j-k)j}$, where the total charge of the sequence is also used in (6). Calculation of these two sequence properties need the individual charges q_{ij} but does not involve computations that require several clock cycles. The two properties are therefore computed as the charges enter the circuit, so that the charges have not to be stored. Additionally, as required by the algorithm, the center of gravity of the sequences needs to be calculated. Only calculating the geometric middle, is sufficiently precise and faster than the exact computation.

When all information about a sequence is assembled, it will be sent to the second part, the *Merger*. That way the narrow, fixed-rate data stream is converted into wide data words of varying rate. That rate is dependent on the length of the sequences. Long sequences give low data rate and vice versa. Because of this, and also because of varying processing speed of the *Merger*, a small FIFO is used to buffer the sequence data. Since the Decoder handles data in real time as they come, the sequences will also be ordered by ascending row, ascending pad and descending time.

4.2. Merger

The *Merger* decides which sequences belong together and merges them. The merging in effect increases the k index of (6) and j index of (7) by one, adding one more sequence to the sums in the numerators. It also updates the total charge of the cluster Q.

Since the *Merger* is processing the data in arriving order, merging will be done with sequences on the preceding pads. And more precisely, since by definition a cluster does not have holes, only the immediately preceding pad needs to be searched. Therefore we need only two lists in memory (see figure 6). One list contains the started cluster of the previous pad; the other list contains started clusters already processed at the current pad. Clusters are removed from the search range when a match is found or when the cluster is

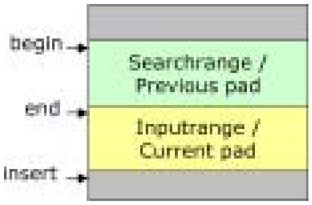

Figure 6: The ring buffer storing clusters of the previous and the actual pad.

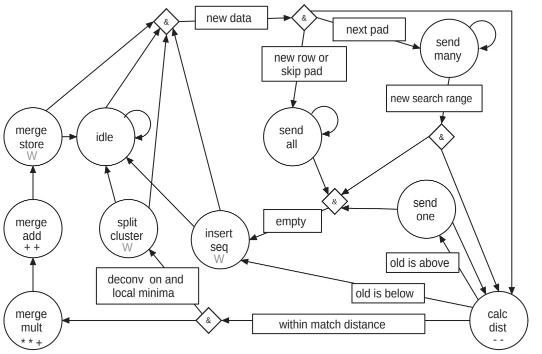

Figure 7: The state machine of the *Merger*. The undertaken arithmetical operations and write accesses are marked.

finished. Clusters are inserted in the input range after merging or when starting a new cluster. The list of clusters on the current pad, the input range, will be searched when a sequence on the next pad arrives. When that happens, there will be no matches for the remaining clusters on the preceding pad. Hence we output clusters in the old search range and exchange the lists. The old list becomes free memory, the current list becomes the old, and a new current list with no entries is created. At the end of a row or when a pad is skipped the clusters in both the lists must be finished. All the clusters are sent and both the lists are emptied. The two lists are implemented as a ring buffer stored in a dual-port RAM. The beginning and ending of the lists is marked by three memory pointers (*begin*, *end* and *insert* pointer).

The state machine of the *Merger* is shown in figure 7. To prevent overflowing the FIFO, the number of states the Merger machine needs to visit for each sequence is kept as low as possible. At the same time every state must be as simple as possible to enable a high clock rate. The ring buffer causes systematic memory accesses; read and write addresses are either unchanged or incremented. To keep the number of states down, the computations of merging are done in parallel, thereby using more of the FPGA resources. The resources needed are two adders (signed), and two "smart multipliers" (see 4.3).

There are three different types of states: States that remove a cluster from the search range sending it to the output (or discarding it if it is a noise cluster or overflow occurred), states that do arithmetic and states that insert a new cluster into the list of the current pad.

Arithmetic is done in three cases. When a new sequence on the current row and pad enters, the distance between starting times, $(a - a_j)$, is calculated. Because the data type is unsigned, a is kept at the top of the cluster by assigning it the highest of the starting times. The roles of the unfinished cluster and sequence are interchangeable. The distance between the middle of the incoming and the middle of the first in the search range is also calculated (*calc_dist*). If the result is within match distance, merging occurs.

In the *merge_mult* state the last part in (7), is computed for the lower of the unfinished cluster and the incoming sequence. The rest of the calculations are done unconditionally. They are finished in the *merge_add* state. After merging the resulting cluster is inserted into the current list (*merge_store*), at the same time the old unfinished cluster is removed from the search range by incrementing the *begin* pointer.

If a match is not found in the *calc_dist* state the ordering of data is crucial: The first cluster in the search range is always the one of highest time value, so if it is below the incoming, the rest of the clusters in the search range will also be below. Hence there can be no matches for the incoming sequence and it can be inserted into the current list (*insert_seq*). In the opposite case, if the cluster in the search range is higher than the incoming, all subsequent incoming sequences will be below or on other pads, so the cluster in search range must be finished. The cluster is output, and the *begin* pointer is incremented (*send_one*). The same procedure happens in three other states. By definition, on a change of row there will be no more matches neither in the search range nor the current list. Therefore all the clusters are sent (*send_all*). As described above, motivating the ring buffer: when there is a change of the pad, clusters in the search range are sent and the lists are renamed (*send_many*). The last case finishing a cluster occurs if convolution is turned on and a local minima in a cluster is detected. The *split_cluster* state is combination of *send_one* and *insert_seq*. The old cluster is sent to the output and the incoming is inserted into the search range.

4.3. SmartMult

Both the *Decoder* and the *Merger* do multiplications where the range of one of the multiplicands is limited to the size of the cluster as intended by (6) and (7). To save resources (logic cells) on the FPGA and increase the clock speed, we replace the standard multipliers by *SmartMult*, which takes the two multiplicands as arguments and uses left shifts and one

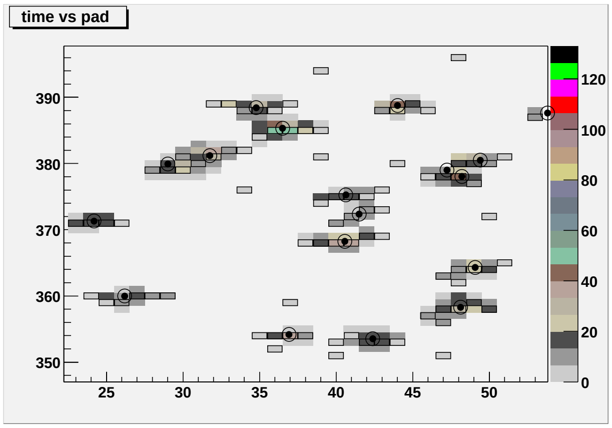

Figure 8: A part of a time-pad plane. Colored patches are input charges, squares mark the geometric middle of the sequences, points and circles mark the centroids found by the FPGA and the C code resp.

Table I Distribution of clock cycles spent in the various states of the *Merger*, corresponding to dN/dy of 2500 and 1000 and with/without deconvolution

state	**2500-n**	**2500-de**	**1000-n**	**1000-de**
idle	26,0	25,2	31,0	30,0
merge_mult	6,5	11,2	9,1	9,0
merge_add	6,5	11,2	9,1	9,0
merge_store	6,5	11,2	9,1	9,0
send_all	0,5	0,4	0,5	0,5
send_many	5,4	4,2	5,6	5,6
send_one	9,3	5,4	5,6	5,7
calc_dist	26,9	23,7	22,9	21,4
insert_seq	12,6	7,3	8,2	8,5
split_cluster	0,0	0,3	0,0	0,2

adder for the calculations. One argument, the short multiplicand, is limited to reduce the number of shifts needed. The limit for the short multiplicand determines the highest sequence that is allowed and the highest number of merges possible. Clusters larger than this are flagged as overflowed and ignored when finished.

4.4. Verification

For verification purposes the system is stimulated by a testbench. The testbench reads an ASCII file containing simulated ALIROOT raw data in an ALTRO like back-linked list, which is sent to the *Decoder*. Operation of the circuit is then studied in a simulator for digital circuits. Found clusters of the *Merger* circuit are directed back to the testbench which writes the results to a file. That file is then compared to a result file made by a C++ program running nearly the HLT C++ algorithm on the same simulated input data. Since the number of clusters is in the order of thousands and to eliminate human error, another C++ program compares the two result files. The found clusters agree as is graphically shown in figure 8.

4.5. Timing

The circuit has been synthesized and currently uses 1937 logic cells. That is 12% of the available resources on the APEX20KE-400-2x, which we are using for prototyping. The clock speed is 35 MHz. For the different input data sets taken for the verification, the *Merger* has been in the *idle* state more than 25% of the time. For two different data sets the distribution of the clock cycles is shown in table I for dN/dy of 2500 and 1000 and with/without deconvolution. As merging is the time critical part of the circuit, there is a safety margin for higher multiplicity data.

5. Conclusion

The fast cluster finder algorithm has been adapted for a hardware implementation. The synthesized circuit currently uses 12% (1937) of the logic cells on the APEX20KE-400-2x FPGA with a clock speed of 35 MHz. The time critical *Merger* circuit is idle more than 25% of the time, which implies a safety margin for higher multiplicity input data.

References

[1] ALICE Collab., *Technical Proposal*, CERN/LHCC/95-71 (1995).

[2] ALICE Collab., *Technical Design Report of the Time Projection Chamber*, CERN/LHCC 2000-001 (2000).

[3] V. Lindenstruth et. al., *Online Pattern Recognition for the ALICE High Level Trigger*, Proceedings of 13th IEEE-NPSS RealTime Conference, Montreal, Canada, May, 2003

[4] J. Berger et. al., *TPC Data Compression*, Nucl. Instr. Meth. A 489 (2002) 406

[5] A. Vestbø et. al., *High Level Trigger System for the LHC ALICE Experiment*, ACAT 2002 Workshop Proceedings, Moskow, June 2002.

[6] P. Yepes, *A Fast Track Pattern Recognition*, Nucl. Instr. Meth. A380 (1996) 582

[7] C. Adler et. al., *The STAR Level-3 trigger system*, Nucl. Instr. Meth. A499 (2003) 778

[8] R. Bosch, A. de Parga, B. Mota, L. Musa, *The ALTRO Chip: A 16-channel A/D Converter and Digital Processor for Gas Detectors*, submitted to the 2002 IEEE NSS/MIC, Nuclear Science Symposium, Norfolk, November 12-14, 2002