Tiny Triplet Finder (TTF) – A track segment recognition scheme and its FPGA implementation developed in the BTeV level 1 trigger system

Jinyuan Wu, Z. Shi, M. Wang, H. Garcia and E. Gottschalk

Fermi National Accelerator Laboratory, Batavia, IL 60510, USA
jywu168@fnal.gov

Abstract

We describe a track segment recognition scheme called the Tiny Triplet Finder (TTF) that involves the grouping of three hits satisfying a constraint, for example, forming a straight line. The TTF performs this $O(n^3)$ function in $O(n)$ time. The logic element usage in FPGA implementations of typical track segment recognition functions are $O(N^2)$, where N is the number of bins in the coordinate considered, while that for the TTF is $O(N\log(N))$, which is significantly smaller for large N. The TTF is also suitable for software implementation and many other pattern recognition problems.

I. Introduction

Track segment finding is an essential process in many trigger systems for high-energy physics experiments. In the Fermilab BTeV [1], trigger system, for example, we need to identify track segments from the coordinates of pixel detector hits from three adjacent detector planes forming a straight-line segment in the non-bend view. For a given track segment, the following relationship holds:

$$u_A + u_C = 2u_B$$

where u_A, u_B and u_C are the hit coordinates on planes A, B and C in the non-bend view. Such segments consisting of three hits are referred to as "triplets" (See Fig. 1, 2) [3].

Straightforward software implementation of such a function would require $O(n^3)$ execution time, where n is number of hits per plane, in order to examine all possible combinations of three hits using three layers of nested loop. In a hardware implementation, this execution time can be reduced to $O(n)$ proportional to the time required to fetch the data. This is accomplished by "unrolling" two layers of loops, consuming a significant portion of the silicon resources in the device. The number of logic elements needed in many typical triplet finding implementations is $O(N^2)$ where N is the number of bins that each plane is divided into.

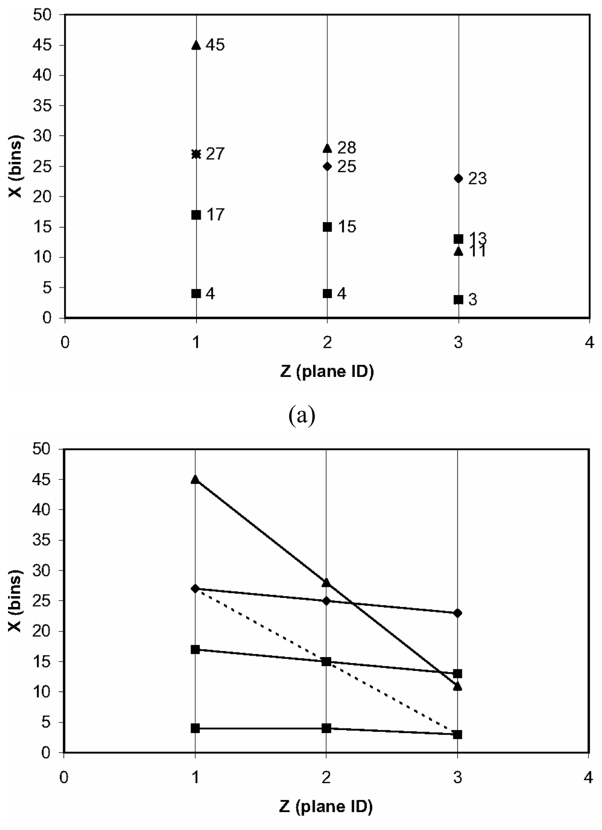

Figure 1: Triplet finding.

In this article, we describe a new algorithm that performs the triplet finding function, which we will refer to as the Tiny Triplet Finder (TTF). We also describe a sample hardware implementation of the TTF using the low cost Altera Cyclone [7] family of FPGA devices. Logic element usage in this implementation is $O(N\log(N))$ which is significantly smaller than $O(N^2)$ when N is large.

II. Principle

Consider the three equally spaced detector planes in the non-bend view as shown in Fig. 2. We first divide the two outer detector planes, Plane_A and Plane_C, into N bins ($N = 64$ in this example), choosing a bin as a unit of the coordinate in the non-bend view, and rounding off to integer units.

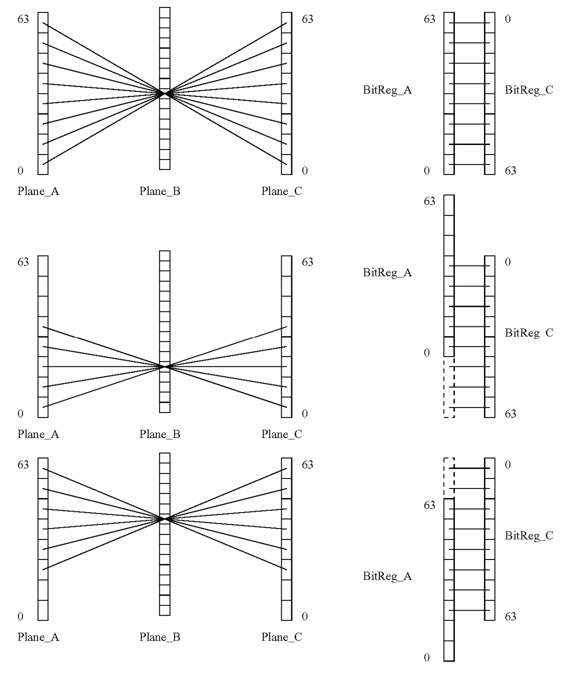

Figure 2: Tiny Triplet Finder.

In general, there exist N^2 possible track combinations, or "roads" in this configuration. A "road" is defined here as a line segment passing through one of the N bins in each of Plane_A and Plane_C. Directly implementing all possible combinations using either logic elements or content addressable memories (CAMs) would require a huge amount of silicon resources (64 X 64 = 4096 in this example).

In the Tiny Triplet Finder, two register arrays, BitReg_A and BitReg_C, are used to record the hits in the detector planes. When a hit coordinate from a detector plane is input, one of the 64 bits in the register array, corresponding to its position, is set. Once all hits in the Plane_A and Plane_C are recorded, the algorithm cycles through the hits from Plane_B one at a time. In the special case when the hit is at the mid-point of Plane_B (see the top of Fig. 2), there will be 64 possible track combinations or roads. Each possibility is checked through bit-wise coincident logic between the bit patterns recorded in BitReg_A and BitReg_C. If a pair of corresponding bits in BitReg_A and BitReg_C are both set, e.g., (0, 63), (1, 62) or (2, 61), etc., the bit-wise logic will output the pattern of the matching pair(s) corresponding to a possible track segment passing through the hits in Plane_A, Plane_C and Plane_B. The bit-wise coincident logic is primarily a bit-wise AND of the patterns in the two registers. In a real implementation, a bit-wise OR with the neighbouring bits in one pattern may first be performed to cover boundaries.

For hits that are not at the mid-point of Plane_B, the bit-wise coincident logic is identical, except that the positions of the bit patterns representing the hits on Plane_A and Plane_C relative to each other are shifted by an amount determined from the coordinate of the hit on Plane_B (see the middle and bottom configurations of Fig. 2). Rewriting the constraint for the triplet in the following form:

$$u_A = -u_C + 2u_B$$

we see that the relative shift between the bit patterns is $2u_B$. We also see that the orders of the two bit patterns relative to each other should be reversed due to the negative sign between u_A and u_C.

We should point out that hits from different tracks or noise hits can satisfy the coincident logic resulting in fake tracks. The simplest way to deal with these is to encode and output all of them performing arbitration at a later stage. The user can also use priority encoder to choose one depending on the physics requirement of the experiment.

In the Tiny Triplet Finder, only N (64) combinations are implemented in the bit-wise coincident logic, rather than N^2 (64 X 64=4096) combinations. Taking advantage of symmetry, we get all possible combinations by shifting the bit pattern.

In the time domain, the total execution time is taken up by the following processes:

1. Setting the bit patterns BitReg_A and BitReg_C.
2. Looping over hits in Plane_B, shifting the bit pattern in BitReg_A, performing the bit-wise coincident and decoding matching pair(s) found.

Although these are essentially $O(n)$ processes, there will be a small non-linear contribution when more than one pair is found by the bit-wise coincident logic.

Ignoring the small non-linear contributions, we see that the TTF unrolls two layers of loops so that an $O(n^3)$ process can now be executed in $O(n)$ time.

This is accomplished through the use of the bit-wise coincident logic that simultaneously finds all matching hits on Plane_A and Plane_C for each hit on Plane_B in a single operation, making the process time proportional to the number of hits n on Plane_B.

III. FPGA Implementations of the Tiny Triplet Finder

The block diagram of the Tiny Triplet Finder implemented in a FPGA device is shown in Fig. 3.

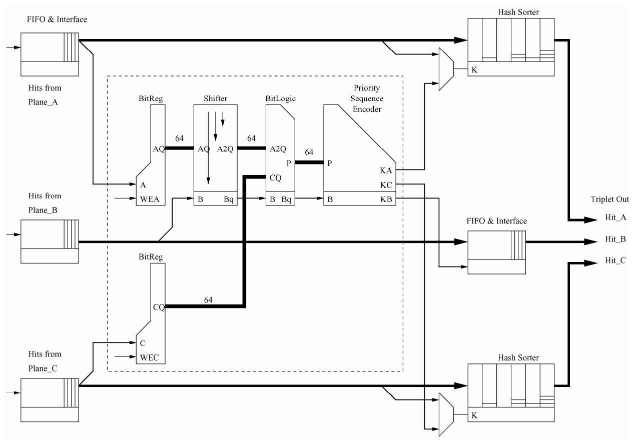

Figure 3: Block diagram of Tiny Triplet Finder.

A. Bit Array Filling

As the hit data from Plane_A and Plane_C are fetched from input FIFO's, a bit corresponding to each hit is set in the BitReg blocks. The resulting hit patterns are presented at the output ports on buses AQ and CQ. Meanwhile, the full hit data are stored into memory buffers called "Hash Sorters" [5] for fast retrieval later. For simplicity, one may think of the Hash Sorters as memory areas that are each divided into 64 bins. When a hit sets a bit in the BitReg register array, the full hit data are written into the corresponding bin in the Hash Sorter.

B. Looping B Hits and Shifting Bit Pattern

After all the hits from Plane_A and Plane_C have been written into the Hash Sorters, the hits from Plane_B can now be fetched from the input FIFO's. The coordinates of the hits from Plane_B are used to determine the relative shift distance between the two bit patterns AQ and CQ. The shifter shifts the bit pattern AQ by this amount and presents the shifted pattern at port A2Q. The full hit data from Plane_B are also stored, for later retrieval, in a buffer which can either be a hash sorter or a regular output FIFO.

The shifter is implemented in a two-stage pipeline to increase operation frequency. Although the shifter requires a relatively large amount of logic elements ($O(N\log(N))$) in comparison to the other blocks in this design, it is still much smaller than typical implementations where $O(N^2)$ logic elements are needed.

C. Bit-wise Coincident Logic

The bit pattern CQ and the shifted pattern of AQ, A2Q, are sent to the "BitLogic" block in which the bit-wise coincident logic is performed. The coincident logic is essentially a bit-wise AND. The OR logic among the neighbouring bits in A2Q is included to cover the boundaries. The detailed logic is P[k]=CQ[k]& (A2Q[k] + A2Q[k-1]), where k is the bit index.

Any non-zero bit in the resulting bit pattern P indicates a found triplet. The location of this bit represents the coordinate of the Plane_C hit belonging to the triplet. The coordinate of the Plane_A hit can be derived from this location and the distance of shift.

D. Priority Sequence Encoder

The locations of the non-zero bits are encoded in the "Priority Sequence Encoder" block which can accommodate situations with more than one triplet. When there is only one non-zero hit in the bit pattern P, the encoder outputs the location of the bit. If there are two or more non-zero bits, the encoder will insert a wait signal to halt earlier pipeline stages, allowing the locations of all the non-zero bits to be reported sequentially.

This block is also implemented as a pipeline. Although it takes 6-clock cycles to encode the non-zero bit(s) in P, the block accepts one P pattern each clock cycle, as long as the wait signal is not inserted.

IV. Test Designs and Silicon Resource Usage

We have test compiled the Tiny Triplet Finder with *N*=64 and *N*=128 bins in an Altera EP1C4 Cyclone device [7].

The full simulation of the Tiny Triplet Finder is shown in Fig. 4. The simulation uses hit coordinates given in Fig. 1 as an example. The coordinates for Plane_B are multiplied by 2 to obtain the shift distance. All 4 real triplets in this example are found plus a fake one which also satisfies the triplet condition KA + KC = KB and is represented by the dashed line in Fig. 1 with hits (27, 15, 3).

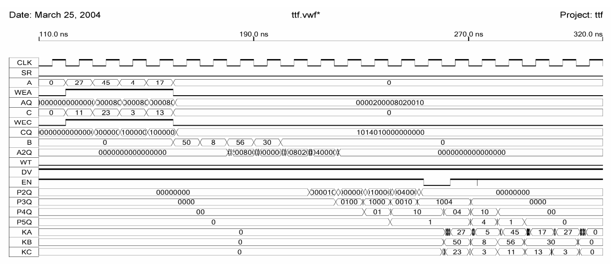

Figure 4: Full Simulation of the Tiny Triplet Finder.

The outputs of the Priority Sequence Encoder, KA, KB, and KC, are the bin numbers where the original hit data are stored in the Hash Sorters (or FIFO for Plane_B hits). These numbers are used as addresses to read out the hit data in the corresponding bins to send to later stages for further processing.

In case there is more than one hit stored in a bin, the Hash Sorter will output all the hits in the bin so that later stages can make better choice. In this case, the pipeline in earlier stages will be halted, allowing multiple hits to be read out.

Another interesting point shown in this example is that we have found a triplet (5,8,3) corresponding to the input (4,8,3). One of the input coordinates is off by 1 bin due to a boundary effect and/or a round-off error. Our bit-wise coincident logic covers this kind of difference. To trace back the original hits in the Plane_A at bin 4, the hash sorter will check both bin KA and KA-1, i.e., both bin 5 and bin 4 in this example.

The compilation results are shown in Table 1 for all functional blocks shown within the dashed box in Fig. 3. As we can see, the Tiny Triplet Finder can easily be accommodated in currently available middle-sized FPGA's.

Table 1: Silicon Usage of Triplet Finder Implementations

Devices: Price: (04/2004)	**EP1C4** $35.90	**EP2A40** $1200	
	Logic Cells (4000)	Logic Cells (30,855)	Embedded System Blocks (160)
TTF (64 bits)	944 (23%)	944 (3%)	-
TTF (128 bits)	1681 (42%)	1681 (5%)	-
CAM using ESB (64 bits)	Not fit		128 (80%)
Hough Trans. (64 bits)	Not fit	16384 (53%)	

The resource usages for two other typical implementations are also shown for comparison. The first one uses Content Addressable Memories (CAM) which can be implemented fairly efficiently with Altera Embedded System Blocks (ESB's) [8]. For this case, we calculated silicon usage assuming 64 X 64 =4096 roads without considering boundary effects and including other supporting logic.

The second other implementation uses the Hough transform scheme [6]. The number shown includes only the 2-D histogram, assuming each bin can be implemented with 4 logic cells. Decoder and other supporting logic are not included.

Since these two other implementations do not fit in the EP1C4 device, we picked an EP2A40 APEX II device [8], which is 7 times larger, to accommodate them.

Furthermore, as the bin number increases from 64 to 128, the logic cell usage of the Tiny Triplet Finder will increase only by about a factor of 2 while that for the other two implementations will increase by a factor of 4.

V. Configurations with Unequally Spaced Detector Planes

The Tiny Triplet Finder may also be used in triplet finding problems with non-uniformly spaced detector planes.

Let the distance between the planes A and B be d_1 and that between B and C be d_2, as shown in Fig. 5.

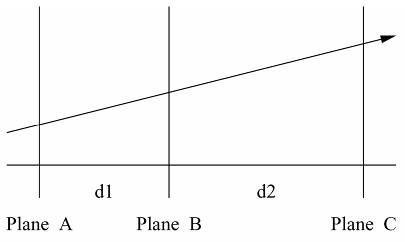

Figure 5: Detector configuration with unequally spaced planes.

For a given track segment, the following relationship exists:

$$\frac{2d_2}{d_1+d_2}u_A + \frac{2d_1}{d_1+d_2}u_C = 2u_B$$

where u_A, u_B and u_C are the hit coordinates on planes A, B and C. We can define:

$$K_A = \frac{2d_2}{d_1+d_2}u_A; \quad K_C = \frac{2d_1}{d_1+d_2}u_C; \quad K_B = 2u_B$$

where K_A, K_C and K_B are integers after eliminating fractional bits. We then have:

$$K_A + K_C = K_B$$

This shows that the Tiny Triplet Finder can be applied for arbitrarily spaced detector planes. The units of bins for plane A and C are as defined above. The relative shift between the two bit patterns is still K_B.

VI. Configurations with More Than Three Planes

The method used in the Tiny Triplet Finder is not restricted to three planes. By introducing more planes, additional constraints are added. The additional hit information can be also used to handle situations with missing hits due to detector inefficiencies.

Consider a 5-plane example as shown in Fig. 6.

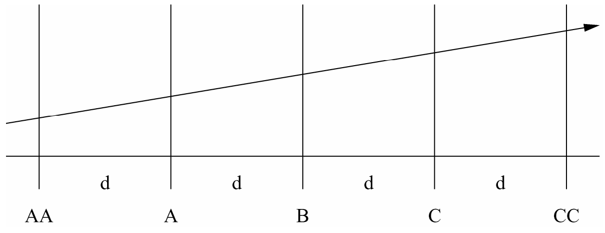

Figure 6: Detector configuration with unequally spaced planes.

We assume the planes in this example are uniformly spaced for simplicity. For arbitrary spacing, the method shown in the previous section can be applied.

For a given track segment, the following constraints can be written:

$$u_A = -u_C + 2u_B$$

$$\frac{1}{2}u_{AA} = -u_C + \frac{3}{2}u_B$$

$$-\frac{1}{2}u_{CC} = -u_C + \frac{1}{2}u_B$$

Again, the hit coordinates on planes AA, A, B, C and CC in the non-bend view are connected through these constraints.

The equations above provide the bin size and shift distance (relative to plane C, an arbitrary choice) for each plane. The bin sizes for planes AA and CC are twice those for A and C. The bin order for plane C and CC are reversed relative to AA and A. The shift distances for plane A, AA and CC are 2, 3/2 and 1/2 of the hit coordinate on B.

The bit-wise coincident logic can be implemented in the following 3 different ways:

- The highest level of constraint is a 4-fold bit-wise AND. This logic is best on eliminating fake triplets. However, the required efficiency of the detector must be very high.
- Another possible logic is 3-out-of-4 bit-wise majority. This allows 1 out of the 4 hits to be missing due to detector inefficiency. Two out of three constraints are used resulting in a lower fake triplet rate than the 3-plane situation.
- Using 2-out-of-4 bit-wise majority logic will produce as many fake triplets as in the 3-plane situation (perhaps even more). However, this allows 2 out of 4 hits missing, resulting in the highest triplet finding efficiency.

In practice, the second choice is a reasonable one since it maintains a good balance between the ability to deal with inefficiencies and the ability to reject fake triplets.

VII. Discussions

We have described an FPGA implementation of the Tiny Triplet Finder. Since the Tiny Triplet Finder algorithm uses no special logic operations other than shift and bit-wise AND/OR, it is also suitable for software implementation. In most CPU or DSP processors, the execution time will be reduced from $O(n^3)$ to $O(n)$. In addition to track segment finding, the TTF algorithm may also be used in hit recognition problems in wire chambers, time of flight counters, and GEM/MICROMEGAS detectors. We will discuss these applications in separate documents.

VIII. References

[1] Kulyavtsev et al., BTeV proposal, Fermilab, May 2000, BTeV-doc-66.

[2] G. Y. Drobychev et al., Update to BTeV proposal, Fermilab, March 2002, BTeV-doc-316.

[3] M. Wang, BTeV Level 1 Vertex Trigger Algorithm, BTeV-doc-1179.

[4] E.E. Gottschalk, BTeV detached vertex trigger, Nucl. Instrum. Meth. A 473 (2001) 167.

[5] J. Wu, M. Wang, E. Gottschalk, G. Cancelo and V. Pavlicek [for BTeV collaboration], “Hash sorter: Firmware implementation and an application for the Fermilab BTeV level 1 trigger system,” FERMILAB-CONF-03-357-E available: {http://www.slac.stanford.edu/spires/find/hep/www?r=fermilab-conf-03-357-e} Presented at IEEE 2003 Nuclear Science Symposium (NSS) and Medical Imaging Conference (MIC), Portland, Oregon, 19-24 Oct 2003}

[6] R. Fruhwirth et al., “Data Analysis Techniques for High-Energy Physics”, 2nd ed., Cambridge, 2000

[7] Altera Corporation, “Cyclone FPGA Family Data Sheet”, (2003)

[8] Altera Corporation, “APEX II Programmable Logic Device Family Data Sheet”, (2002)