Parallel Hough transform for track detection in LHCb's VELO Pixel detector

Matthias Ebert

September 18, 2014

Abstract

The following text describes the first try in implementing a track-detection algorithm, based on the Hough transform method for finding straight lines, that can be efficiently run in parallel on GPUs.

0.1 Track-Detection using Hough transform

Here, track detection is accomplished by employing the Hough transform technique for straight lines in 2-dimensional space. This method relies on the possibility of uniquely describing any line going through a point (x, y) in the 2-dimensional plane by a set of two new coordinates (r, θ). Here θ represents the angle of the vector from the origin to the closest point on the line while r is the length of this vector (see Fig(1)). This mapping from x,y- to r, θ-space (or Hough-space) is done via the Hough transformation which in this case comes down to a simple polar coordinate transformation

$$r(\theta) = x * cos(\theta) + y * sin(\theta) \tag{1}$$

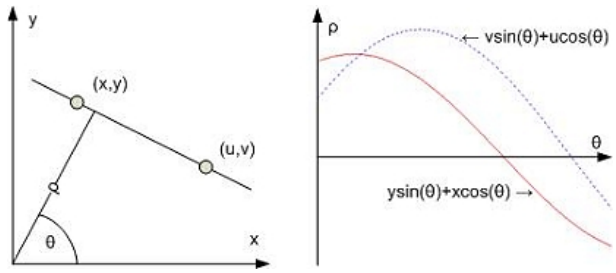

Figure 1: Transformation from real-space to Hough-space using (1) as a mapping

This transformation (1) for $\theta \epsilon \{0, \pi\}$ maps every point (x, y) in normal space to a continuous line in Hough-space which represents all the straight lines that could pass through the given point. One could also describe this transformation as mapping points to lines and lines to points.
Now, if we had two points in real space which by (1) would map to two lines in Hough-space and those two lines would cross at some place (r, θ) this coordinates would represent a straight line (in real-space) which passes through both of the original two points.

0.2 Algorithmic implementation

The basic idea of an algorithm using the above concept for track-finding would be to calculate, with a finite resolution, all the possible lines through each point(hits in the detector) resulting in a set of discretized curves in Hough-space. One for each point. Then all thats left is to find intersections of more than a desired number of curves which will reveal the (r, θ) coordinates of a line passing through the same number of points in Real-space.
To find all those intersections and the set of points on the respective straight lines without many (calculation intensive) comparisons an approach using a histogram appeared to be rather efficient.
Here, the idea is to create a 2-dimensional histogram with a bin for each pair

(r, θ) that are within relevant limits. Then, while doing the Hough transform for all the points the resulting coordinates (r_i, θ_j) will map to one bin in the histogram so the latter can be incremented. All thats left to do is look for bins with a value that is higher than the minimum number of points you are willing to call a line.

Unfortunately, in our case this is not enough since the desired output of the algorithm should not only contain every possible straight line but also all the Hit-IDs of the points that lie on each line.
To keep track of this information a second histogram is needed. This time represented by a 3-dimensional data structure. 2 dimensions are identical to the one above and represent every possible bin but instead of incrementing the number in the bins the third dimension is used to stack up the Hit-IDs of the points that would have resulted in an increment.

Furthermore, since equation (1) only serves as a mapping into Hough-space for 2 dimensional lines while the actual tracks live in 3 dimensions the problem will have to be solved in two steps. First, the Hits will be projected on the XZ-plane as well as the YZ-plane. There lines can be generated using the 2-dimensional Hough transform.
Then, as a second step the set of XZ-lines has to be compared against the set of YZ-lines. If some Hit-IDs appear in two lines this will indicate a track in 3-dimensional space.

0.2.1 Kernel for Hough transform

For the full code see Appendix A.

In this implementation the kernel is invoked for each angle used for the Hough transform. So all the possible $sin(\theta)$ and $cos(\theta)$ values can be stored in an array which will be passed to the kernel, while the global ID of each thread is used to retrieve the right value (see line 8, 9).
Then, each thread uses those values to compute the respective radius in Hough-space (line 17). Each thread does this once for all the points (for-loop in line 11). In lines 18, 19 this radius is transformed into a positive integer in order for it and θ to map to one unique histogram bin.
Since the Hits have been projected on a 2-d plane there is no longer any guarantee that the maximum number of points on a line will be equal to the number of detectors. This is accounted for in line 21. Finally, the ID of each point has to be written at the appropriate position in the 3-d histogram $Histo$ (line 23). To get the right position and to keep track of the number of IDs in each bin a 2-d histogram $Counts$ is used (lines 21, 23, 25).

So, in the end both of the data structures mentioned in the above section were necessary.

0.2.2 Kernel for comparing 2-d lines to find 3-d tracks

As mentioned above, we are actually interested in tracks in 3-d space so the 2-d lines will have to be compared for mutual Hit-IDs. This has been done using, again, a 2-d histogram where each bin (i, j) corresponds to the number of equal IDs in both the i-th XZ-line and the j-th YZ-line.

The above task will be done by a kernel executable in parallel. The code for which is listed in Appendix B.

The kernel will be launched once for each pair of a XZ-line and YZ-line (if there are m XZ-lines and n YZ-lines it will be launched m*n times). So, by use of it's global IDs each thread can determine which lines to load form $In1$ (XZ-lines) and $In2$ (YZ-lines) respectively (see lines 14-18).
In lines 20-29 the kernel checks how many equal IDs both lines contain (common 0s do not count) and stores the number in *count*.
Finally, this *count* is stored in *Overlap* (see line 33) which has one bin for each possible pair of lines (m*n bins, same as the number of kernel launches).

Once the histogram is filled every row can be checked for the highest number, the position of which will tell us each combination of XZ-lines and YZ-lines that have the highest number of IDs in common. This allows us to create valid 3-d tracks.

0.3 Conclusion

As stated in the abstract at the beginning this has only been the first try in implementing a parallel track-detection algorithm based on Hough transform. Consequently, further improvements and optimizations will be necessary before any meaningful comparison against previous algorithms can be made.
Furthermore, the focus of this project has been on finding a more or less efficient way to generate the initial lines via Hough transform, while giving little thought on the other parts like the combination of 2d-lines to a 3d-track or the detection of duplicate lines.
Unfortunately the latter turned out to be, by far, more time expensive than the actual Hough transformation. So, as a next step it will be necessary to significantly reduce the execution time of the part responsible for the reduction of two sets of 2d-lines to one set of 3d-lines as well as making the check for duplicate lines more efficient.

0.4 Appendix

0.4.1 Appendix A

Full code of an OpenCL-kernel using Hough transform to fill a histogram with the IDs of points that lie on the same line in Real-space:

```
1 __kernel void createHisto(__constant float *Sin,
    __constant float *Cos, __constant int *PointData,
    __global int *Histo, __global int *Counts,
2 float radiusRes, int numRadii, int numPoints, int
    numAngles, int numDetectors){
3
4   int gid = get_global_id(0);  // determines the angle
      for hough transformation
5   int i, ID, x, y, pos;
6
7   float radius;
8   float sinTheta = Sin[gid];
9   float cosTheta = Cos[gid];
10
11  for(i = 0; i < numPoints; i++){
12
13    ID = PointData[i*3];
14    x = PointData[i*3 + 1];
15    y = PointData[i*3 + 2];
16
17    radius  = x*cosTheta + y*sinTheta;
18    pos = convert_int(radius/radiusRes);
19    pos += numRadii;     // Position in the array has to
        be a positive value
20
21    if(Counts[pos*numAngles + gid] < numDetectors){
22
23      Histo[pos*numAngles*numDetectors +
          gid*numDetectors + Counts[pos*numAngles + gid]]
          = ID;
24      //(void)atomic_inc(&Counts[pos*numAngles + gid]);
25      Counts[pos*numAngles + gid]++;
26    }
27  }
28 }
29 //
```

0.4.2 Appendix B

Full code of an OpenCL-kernel checking if two 2-d lines have feature overlapping points:

```
1 #include "Definitions.h"
2
3 __kernel void getOverlap(__global int *In1, __global int
    *In2, __global char *Overlap, int maxTrackSize){
4
5   int gid1 = get_global_id(0);
6   int gid2 = get_global_id(1);
7   int gsz2 = get_global_size(1);
8
9   int size = MAX_TRACK_SIZE;
10  int count = 0;
11
12  int Track1[24], Track2[24];
13
14  for(int i = 0; i < size; i++){
15
16    Track1[i] = In1[gid1*size + i];
17    Track2[i] = In2[gid2*size + i];
18  }
19
20  for(int i = 0; i < size; i++){
21
22    for(int j = 0; j < size; j++){
23
24      if((Track1[i] != 0) && (Track1[i] == Track2[j])){
25
26      count++;
27      }
28    }
29  }
30
31  if(count >= 3){
32
33    Overlap[gid1*gsz2 + gid2] = (char)count;
34  }
35 }//
```

0.5 References

Figure 1; http://campar.in.tum.de/Students/DaPentenrieder