An SIMD parallel version of the VELO Pixel track reconstruction for the LHCb upgrade

Public Note

Issue: 1
Revision: 1

Reference: LHCb-PUB-2013-007
Created: April 15, 2013
Last modified: June 10, 2013

Prepared by: Royer Ticse[a], Daniel Hugo Cámpora Pérez[b], Rainer Schwemmer[b], Niko Neufeld[b]
[a]Universidade Federal do Rio de Janeiro, Brazil
[b]CERN, Switzerland

LHCb-PUB-2013-007
10/06/2013

Abstract

This note describes a new algorithm for the VELO Pixel track reconstruction for the LHCb upgrade. Our tracking algorithm implementation was designed with parallelism in mind for an easy adoption in many-core architectures, like GPGPUs. First, CPU vectorization units are explored as an attempt to speedup a critical section of the current implementation. Then, studies on a parallel processing technique are presented, with a draft design for many-core architectures.

Document Status Sheet

1. Document Title: An SIMD parallel version of the VELO Pixel track reconstruction for the LHCb upgrade			
2. Document Reference Number: LHCb-PUB-2013-007			
3. Issue	**4. Revision**	**5. Date**	**6. Reason for change**
Draft	1	April 15, 2013	First version. Introduction only, conclusion is missing.
Draft	2	June 10, 2013	Added conclusion.
Final	1	June 10, 2013	Checked English.

Contents

List of Figures

List of Tables

1 Introduction

The LHCb experiment has planned an upgrade in 2018 to allow operation at higher luminosity $L \approx 2 \times 10^{33} cm^2.s^{-1}$, with a very flexible software-based trigger. In order to improve the trigger efficiencies, the whole detector will be read out at the full bunch-crossing rate of 40 MHz (currently operating at $L = 2 - 4 \times 10^{32} cm^2 s^{-1}$, with a readout at 1.1 MHz).

Track reconstruction in the VELO is a crucial ingredient to the first software level trigger. A new kind of VErtex LOcator detector, based upon the use of pixel detectors with the Medipix / TimePix family of chips, in contrast to the current ϕ and R strips, is being considered. A first geometry implementation of the VELO pixel detector named VeloPix is available, and it has a matrix of 256×256 pixels of 55×55 μm^2 each. It will provide simultaneously information about the deposited energy and the timestamp, reaching an output rate at the hottest chip of more than 12 Gbit/s. More detailed VeloPix specifications are given in [1].

In this note, we present some preliminary studies on parallelizing the current sequential algorithm for VeloPix. We analyze the impact of using vector operations in the existing code, applying our changes to a critical section of the triggering codebase.

Then we focus on drafting a new design, compatible with state-of-the-art many-core architectures. We study a global method for tracking in the VELO, and we show the reconstruction efficiency and purity of our reconstructed tracks using such a method.

2 Vectorization of the current implementation.

Even before multi-core CPUs became mainstream, CPUs had the ability to process multiple values by using a Single Instruction Multiple Data (SIMD) architecture. The Streaming SIMD Extensions (SSE) are a set of 128-bit vector registers and an extension to the x86 Instruction Set Architecture (ISA) making use of these registers, introduced by Intel in 1999.

Header File	Extension name	Abbrev.
xmmintrin.h	Streaming SIMD Extensions	SSE
emmintrin.h	Streaming SIMD Extensions 2	SSE2
pmmintrin.h	Streaming SIMD Extensions 3	SSE3
tmmintrin.h	Supplemental Streaming SIMD Extensions 3	SSSE3
smmintrin.h	Streaming SIMD Extensions 4 (Vector math)	SSE4.1
nmmintrin.h	Streaming SIMD Extensions 4(String processing)	SSE4.2
gmmintrin.h	Advanced Vector Extensions Instructions	AVX

Table 1 Header files used for different Streaming SIMD Extensions versions. We use SSE3.

Most modern compilers expose a set of vector types and intrinsics to manipulate them. Here, we use the intrinsics defined by Intel, well documented in [2]. The intrinsics are enabled by including the correct header file. The name of the header file depends on the SSE version being targeted, see Table 1.

The performance boost usually seen in programs using vector operations, comes from the reduction in the overhead of issuing the functional units in the CPU, and the capability of storing data in additional XMM 128-bit wide registers. The abbreviation SIMD describes how the instructions are issued, a single instruction specifies the operation that is to be executed simultaneously on more than one value.

In the SSE vector extension, instructions come in two flavours, aligned and unaligned, which refer to the indirection method to access memory. In PCs, as a generalization memory is accesed by 32 or 64-bit pointers, which identify one-byte-wide cells. The data contained in memory is said to be aligned to some width if the pointer to its starting location is a multiple of the width. If alignment cannot be guaranteed, some part of the performance gain achieved by processing multiple data elements in parallel would be lost, because either the compiler or assembly programmer would have to use unaligned move instructions.

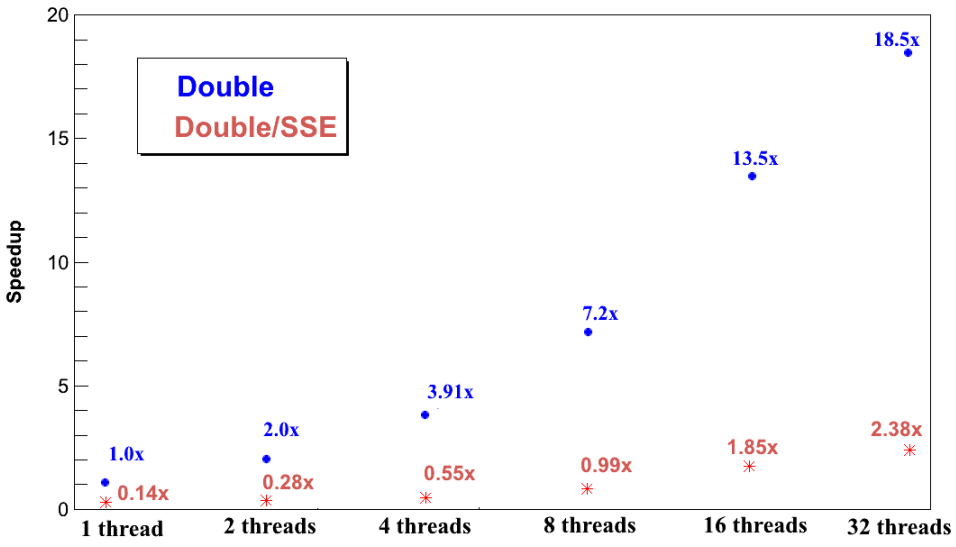

Figure 1 Result for the pixel sequential algorithm using double floating point and Streaming SIMD Extensions.

In our codebase, double floating point numbers (doubles) are used to perform the calculations. The data-width of doubles is 64-bit, therefore fitting an XMM register with only two

numbers. The translation from our codebase to SSE instructions is shown in Figure 1 shows the speedup obtained by using SSE instructions.

The results show a decrease in the performance by using SSE instructions. Memory alignment is an important feature to allow for efficient data loading and store on SSE registers, and in our implementation, data has not been prepared beforehand to be parallelized in vectors, causing a need for a data preparation stage and a decrease in the performance as a consequence of it.

The data preparation stage would potentially be smaller if single point instructions were used instead of double floating point precision.

The pixel sequential project, developed by Daniel Cámpora, is an analysis of the current implementation of the VELO pixel subdetector tracking algorithm (by Olivier Callot, under the package *Rec/Tf/PatPixel v1r1*). The aim of this project was to study the feasability of parallelism in the algorithm at a level of events, focusing on using single floating point precision and enabling parallelization in the algorithm. It allows the execution of parallel track processing, fully utilizing the CPU resources on the reconstruction of tracks. Further documentation on this project is available [5].

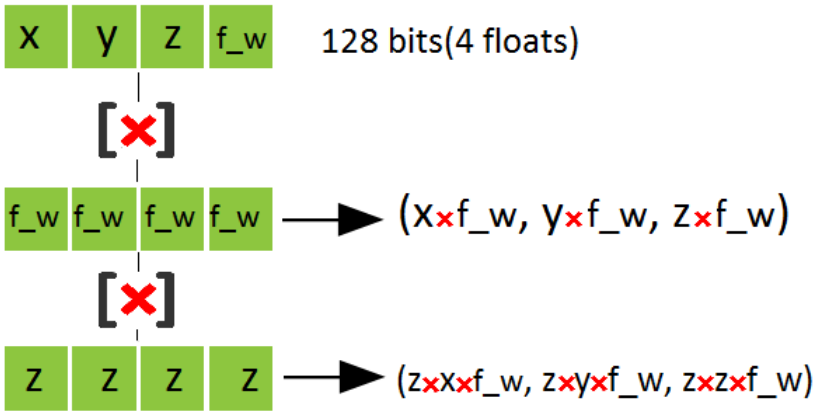

Figure 2 Reducing operations with SIMD. Our strategy was to use three vector units (registers) to reduce recurring operations for each reconstructed track.

Making use of this project, we have implemented a solution using single floating point operations. Our implementation strategy consists in reducing the number of operations in a critical section, executed for each χ^2 calculation in the process of reconstructing tracks. As shown in figure 2, using the SIMD programming model nine multiplication operations are reduced to two instructions. The implementation details are shown in algorithm 1, in page 7. Figure 3 shows the new speedup, against the number of threads used to process in parallel.

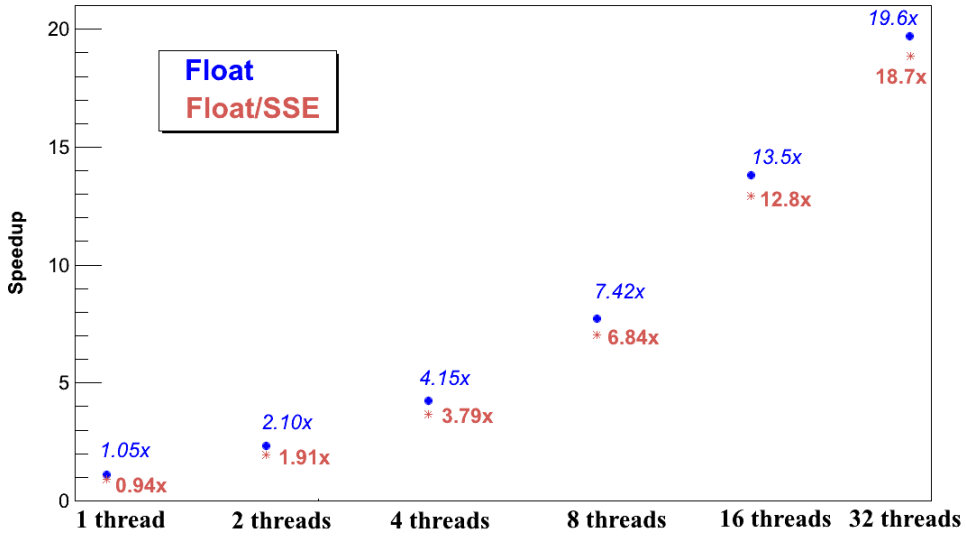

Figure 3 Result for the pixel sequential algorithm using single precision and Streaming SIMD Extensions.

The results show a decrease in the speed, in comparison to the existing implementation. We attribute this to the inexisting memory alignment preparation, which indicate deeper changes would have to be made to the existing codebase in order to account for vectorization speedups.

Table 3 shows the execution times to run 200 Montecarlo events in a single thread with different implementation strategies and precision formats. The best execution times are obtained using single floating point precision.

Precision	Execution time (s)
Double	0.088
Double using SSE	0.564
Float	0.053
Float using SSE	0.061

Table 2 Execution times for one thread using different implementation strategies and precisions.

3 A Global Track Reconstruction method

Many-core architectures, like General Purpose Graphics Processing Units (GPGPUs) or the recent Intel MIC, are built around the philosophy of delivering a larger raw computing power, benefiting a larger density of Arithmetic Logic Units (ALUs) in detriment of control and cache units, on a per-processor basis. As a consequence, they exhibit a different programming model, where functional units are grouped (controlled) in bigger groups, issuing the same instruction to the whole group each cycle.

In order to take full advantage of their computing power, divergent branches should be avoided, and data must be processed avoiding dependencies (like Read After Write). Therefore, in the creation of parallel software, the factors which have an impact on the design of an algorithm change from conventional software development techniques. The algorithm under study in the previous section, makes use of a forward tracking method upon performing the VELO tracking. In surface, it produces tracks by starting on hits on the farthest Z sensor, and sequentially computes hits on previous sensors, forming initially tracklets (or track segments) and potentially tracks. Then, it applies the same criteria on hits of the following sensors, partially excluding the hits already assigned to other tracks.

This approach is not suitable for a many-core architecture, due to the fact data dependencies exist. In this section we propose a global method based upon the creation of tracklet groups, with a further exclusion based on meeting certain criteria.

The algorithm design can be subdivided in three separate stages, Tracklet Construction, Tracklet Grouping and Track Formation. For each of the stages, we outline how a parallel algorithm would not present any dependencies, and thus process it concurrently. We also discuss design improvements for obtaining better tracking efficiency.

The souce files for this project and documentation is available [4]. In our current setup, an input file containing events information, generated by a modification on the Brunel Sequencer, is being used. The implementation is made in C++.

3.1 Tracklet Construction

Our tracking algorithm starts with a combinatorial search for tracklets. We define neighbour sensors as every pair of adjacent sensors on the same side. A tracklet is then a pair of hits of any two neighbouring sensors. In order to keep the multiplicity low, all the possible combinations are required to have a maximum acceptable slope in the (x,y) axis of (300, 400) mrad.[3], according with the angular coverage of the VELO.

We also define a virtual plane E, parallel to the sensors, and placed after the last sensor on the Z+ end. The use of this virtual plane will become clear in the Tracklet Selection stage.

In figure 4a, tracklets are formed from pairs of hits in neighbouring sensors. Then, for each tracklet we define four parameters, as per table 3. The pair (dx, dy) identify the gradient in the X and Y plane, and (ex, ey) is the extrapolation of the tracklet into the E plane. This is depicted in figure 4b.

The process of creation of tracklets is independent from one another, and has no data dependencies. Therefore, it can be carried out in parallel. The average multiplicity (hit $\times$ hit) per event is of around 800 for the Montecarlo dataset we are considering, and this can be processed on a thread-by-hit-pair basis.

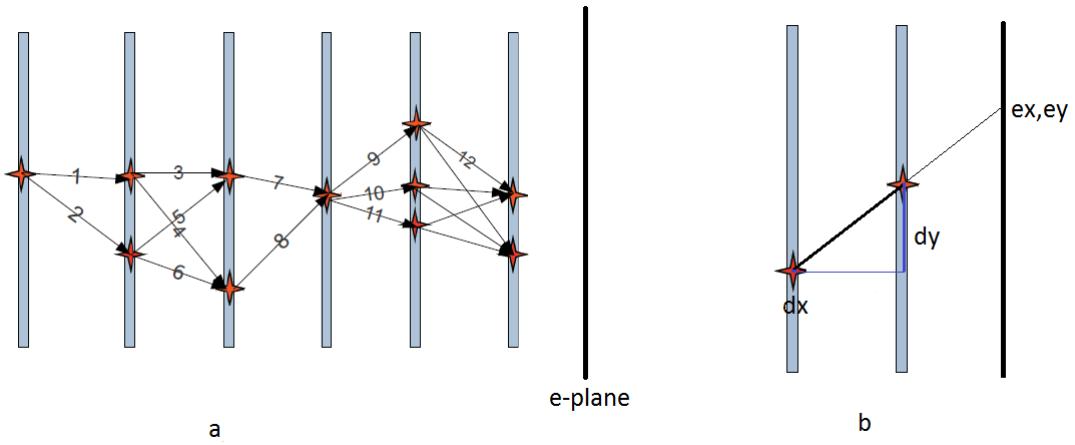

Figure 4 a. Tracklet construction, all possible combinations between two hits in continuous sensors. b. Each tracklet has four variables dx, dy and the projection on the e-plane, ex, ey.

	Information.
(dx, dy)	Gradient in X and Y.
(ex, ey)	Extrapolation of tracklet in the E plane.

Table 3 Tracklet information. Each tracklet has four variables dx, dy, ex and ey.

3.2 Tracklet Grouping

The Tracklet Grouping stage takes care of grouping the tracklets in sets, according to their distance on a gradient and projection 2D plots.

As formula 1 states, given two tracklets t_m and t_n, the gradient distance is an application from $\mathbb{R}^2$ in $\mathbb{R}$, defined as the euclidean distance in the XY plane for their (dx, dy) values.

$$g : \mathbb{R}^2 \to \mathbb{R}, \qquad g(dx, dy) = \sqrt{(dx_m - dx_n)^2 + (dy_m - dy_n)^2} \tag{1}$$

The projection distance, given two tracklets t_m and t_n, is the euclidean distance in the XY plane for their (ex, ey) coordinates. These are then normalized in relation to the distance to the central point of the E plane, as depicted in formula 2.

$$p : \mathbb{R}^2 \to \mathbb{R}, \qquad p(ex, ey) = \frac{\sqrt{(ex_m - ex_n)^2 + (ey_m - ey_n)^2}}{normalization\ \ distance} \tag{2}$$

The gradient distance relates to the straightforward conception for which the gradient in any segment within a track should be constant, with an error e proportional to the sensors' resolution. Therefore, we consider two segments to be part of the same track the closer they are in the gradient norm. A similar principle applies to the projection distance. Any foreseable track should present a minimal distance between any two tracklets composing it.

Figure 5 shows a dy VS dx plane, for all tracklets on a Montecarlo generated event. The density of the points on the E plane is bigger around the origin (0, 0), due to the collision point. The normalization applied gets rid of this effect, normalizing the distance between tracklet projections.

The calculation of the gradient and projection distances is independent from each track, and can thus be processed in parallel. For each tracklet, the distance to the rest could be processed independently, storing a list of neighbours for each.

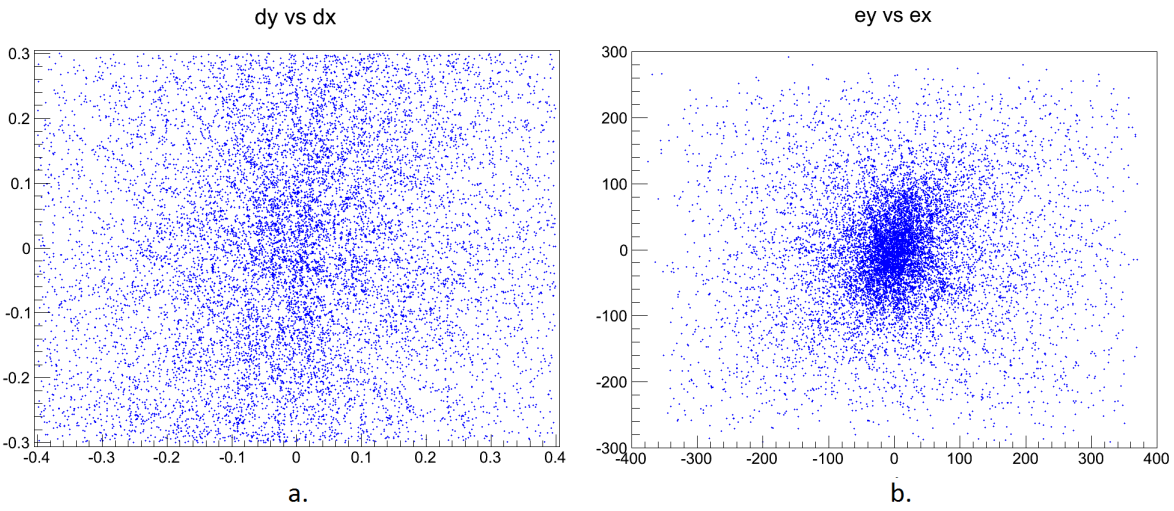

Figure 5 All tracklets formed for one montecarlo generated event. a. dy vs dx plane, b. ey vs ex plane.

3.3 Track Formation

The procedure to create tracks is based on the following: The tracklets formed by its hits must be close to each other, in their projection and gradient distance.

As figure 6 shows, a minimum distance is determined for grouping tracklets, forming sets of tracklets. Two sets are formed in this example, D1 and D2, using distances in the DXDY plane, and two other sets come from the EXEY plane. T1 and T2 would be indistinguishable by only using information from their gradient, since it is similar, in the same way T0 and T1 are indistinguishable by their projection distance. However, by combining these two methods via the intersection of the sets, figure 6b, we can identify the tracks. Figure 7 shows a comparison between our method found tracklet, and the tracklets from the tracks reconstructed by the algorithm developed by O. Callot. The output from our code (blue) and the existing codebase (red) present similarities.

The reconstruction efficiency and purity of this method varies upon the normalization distance for the projection plane, and the density of hits in the event. Fixing the normalization distance to the distance to the center, as formula 3 shows, figure 8 shows the distances with the cuts applied,

$$p : \mathbb{R}^2 \to \mathbb{R}, \qquad p(ex, ey) = \frac{\sqrt{ex_m - ex_n)^2 + (ey_m - ey_n)^2}}{\sqrt{(\frac{ex_m - ex_n}{2})^2 + (\frac{ey_m - ey_n}{2})^2 + 1}} \tag{3}$$

Two performance indicators have been studied, as depicted in table 4 for different hit densities.
The reconstruction efficiency is defined as:

$$\epsilon = \frac{\#(F \cap R)}{\#R} \tag{4}$$

The symbol # denotes the cardinal of the set, F is the set of reconstructed tracks and R is the set of real tracks, i.e. the set of tracks which is expected to be reconstructed. The purity is defined as:

$$purity = \frac{N_{correct}}{N_{total}} \tag{5}$$

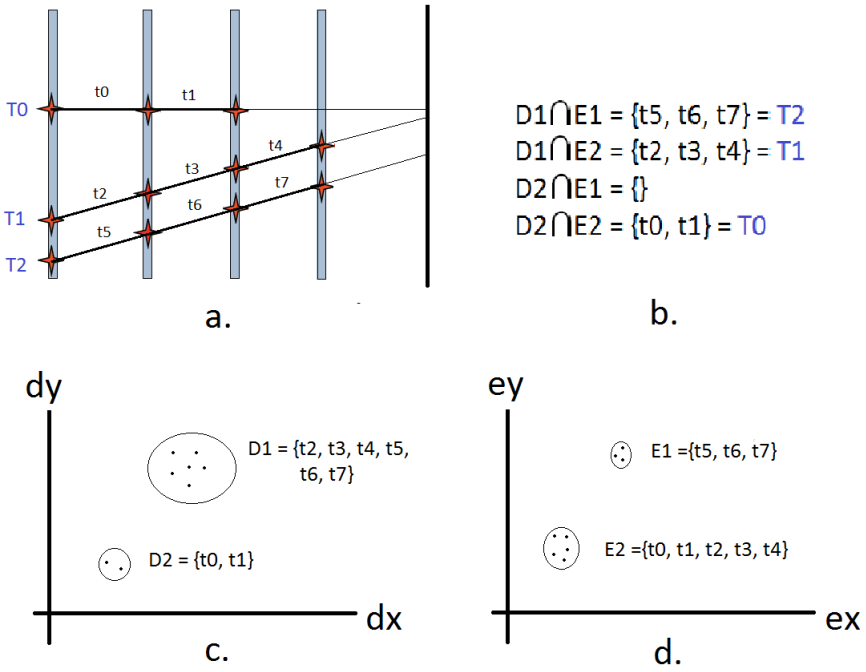

Figure 6 Example. a.Three tracks and their respective hits on the sensors, forming eight tracklets. b.Using the information in each plane, intersecting the possible candidates, we have the tracks. c.The tracklets are grouped according to minimum gradient distance. d.Also in the projection E-plane, tracklets are grouped according to minimum projection distance.

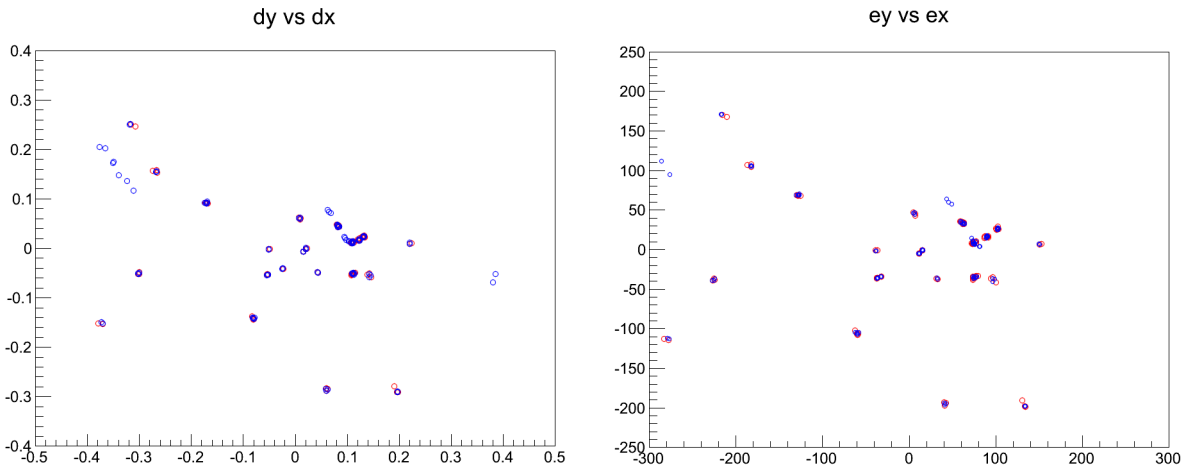

Figure 7 Tracks reconstructed from the sets of tracklets . Blue is the output of our code and red is Olivier's results

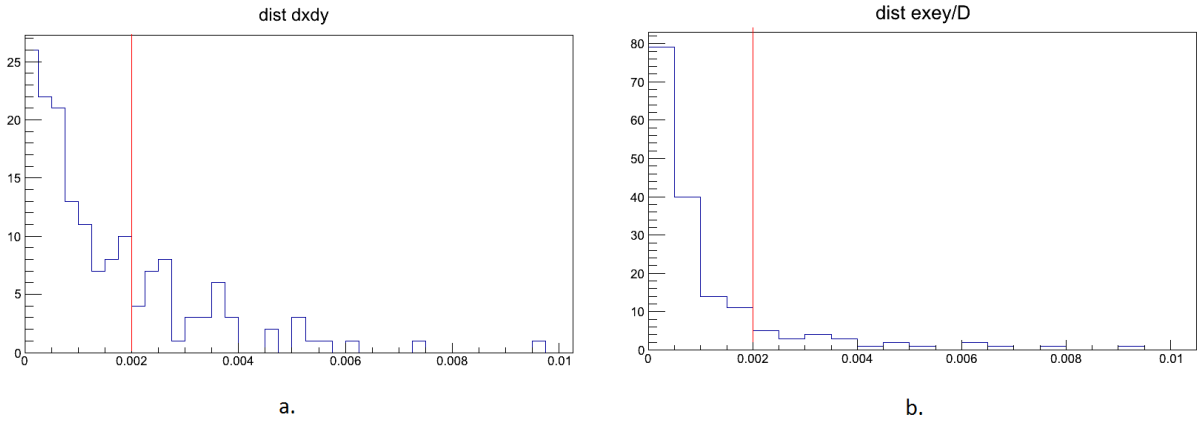

Figure 8 Use a cut $g(dx, dy) < 0.002$ and $p(ex, ey) < 0.002$ to select pairs of tracklets that form a set of tracklets that are candidates for tracks

where $N_{correct}$ is the correct number of tracks found and N_{total} is the total number of reconstructed tracks.

hit densities	# tracks	reconstruction efficiency (%)	purity (%)
55	8	88	88
143	23	87	87
462	67	82	48
623	91	76	22
968	119	69	3
1187	143	55	3

Table 4 Reconstruction figures.

As table 4 shows, our method performs well with low hit densities, with a peak of 88% reconstruction efficiency and purity of tracks, using a simple, parallelizable method. Due to the fact we accept tracklets coming from any pair of hits between sensors, our reconstruction efficiency is much better than our purity for higher hit densities.

4 Design improvements

The track reconstruction method presented is immature, and presents inefficiencies for events containing a high density of hits. We consider there can be design improvements over the study carried out here. We propose a more thorough Tracklet Construction, and a Post-Processing stage.

The tracklet formation based upon two hits introduces noise tracklets. The Tracklet Construction stage could rather consider only tracklets consisting of three consecutive hits, potentially reducing the inefficiencies presented.

A Track Selection post-processing stage would be useful to account for several side effects of our algorithm, like long distant tracklets being considered part of the same track, or to check the duplicity of hits among tracks.

5 Conclusion

We have carried out a study of many-core parallelization over the VELO pixel upgrade proposal.

An SIMD acceleration for a critical section of the current implementation underperforms compared to the base implementation. Data preparation would be needed at earlier stages of the codebase, in order to account for efficient data load and store on vector units.

A study of a parallel design, using a novel global method for VELO tracking, has been presented. Reconstruction efficiency is 80% for events with smaller hit densities, and design improvements have been proposed to account for the side effects of our algorithm. A many-core architecture would take advantage of such an implementation, potentially speeding up our current reconstruction program.

6 References

[1] LHCb collaboration, Framework TDR for the LHCb upgrade, Technical Report CERN-LHCC-2012-007, LHCb-TDR-012, CERN, Geneva, 2012.

[2] Intel C++ Intrinsic Reference. Intel Corporation, 2007.

[3] Callot Olivier, FastVelo, a fast and efficient pattern recognition package for the VeLo. LHCb-PUB-2011-001.

[4] Global method documentation and repositories, *http://lbonupgrade.cern.ch/wiki/index.php/Global_method_approach*.

[5] Pixel-sequential project, *http://lbonupgrade.cern.ch/wiki/index.php/Pixel-sequential*.

7 Annex

Algorithm 1: SIMD implementation (single floating point precision)

```
1  float z = f_hit_Zs[offset];
2  float x = f_hit_Zs[offset];
3  float y = f_hit_Zs[offset];
4    __m128 m_tr_vec1, m_tr_vec2, m_tr_vec3, m_tr_vec5, m_tr_vec6;
5    float temp_vec1[4] = x, y, z, f_w;
6       m_tr_vec1 = _mm_load_ps(&temp_vec1[0]);
7       m_tr_vec2 = _mm_shuffle_ps(m_tr_vec1, m_tr_vec1, _MM_SHUFFLE(3, 3, 3, 3) );
8       m_tr_vec3 = _mm_shuffle_ps(m_tr_vec1, m_tr_vec1, _MM_SHUFFLE(2, 2, 2, 2) );
9       m_tr_vec5 = _mm_mul_ps(m_tr_vec1,m_tr_vec2);
10   float temp[4];
11      _mm_store_ps(&temp[0],m_tr_vec5);
12      m_tr_vec6 = _mm_mul_ps(m_tr_vec3,m_tr_vec5);
13      _mm_store_ps(&temp_vec1[0],m_tr_vec6);
14      tr->m_s0 += f_w ;
15      tr->m_sx += temp[0] ;
16      tr->m_sz += temp[2] ;
17      tr->m_sxz += temp_vec1[0] ;
18      tr->m_sz2 += temp_vec1[2] ;
19      tr->m_u0 += f_w ;
20      tr->m_uy += temp[1] ;
21      tr->m_uz += temp[2] ;
22      tr->m_uyz += temp_vec1[1] ;
23      tr->m_uz2 += temp_vec1[2] ;
```