Università degli studi di Padova
Dipartimento di Fisica e Astronomia "Galileo Galilei"
Corso di laurea specialistica in fisica

Tesi per la Laurea Specialistica in Fisica

Algoritmi paralleli di trigger per decadimenti adronici di mesoni B nell'esperimento LHCb al CERN

Relatore:
Dott. Gianmaria Collazuol

Laureando:
Andrea Rugliancich 588361-SF

Correlatore:
Dott. Stefano Gallorini

Anno Accademico 2013-2014

Indice

Introduzione

L'esperimento LHCb (*Large Hadron Collider - beauty*) è uno dei principali esperimenti in corso presso il Large Hadron Collider (LHC) al CERN di Ginevra (Svizzera), assieme a CMS, ATLAS, ed ALICE. Lo scopo dell'esperimento è quello di effettuare studi approfonditi del fenomeno della violazione di CP e dei decadimenti rari degli adroni contenenti i quark b (b-hadrons) e c (c-hadrons).

Attraverso misure di elevata precisione, LHCb si propone di verificare le predizioni del modello standard (SM): eventuali deviazioni potranno mettere in luce fenomeni non previsti dal modello stesso; sarà quantomeno possibile restringere l'intervallo dei parametri relativi alle estensioni dello SM. Inoltre, lo SM non è in grado di spiegare l'origine dell'asimmetria tra materia e antimateria osservata nell'universo, che è legata alla violazione della simmetria CP [4]. Quasi tutte le estensioni dello SM introducono ulteriori sorgenti di violazione di CP rispetto a quelle proposte dal meccanismo CKM e, grazie a LHCb, queste predizioni potranno essere verificate.

LHCb studia le interazioni protone-protone a piccolo angolo rispetto all'asse di collisione (dette interazioni *in avanti*): è uno spettrometro a braccio singolo con copertura angolare azimutale compresa tra circa 10 e 300 mrad. È costituito da diversi rivelatori: vari tracciatori, tra i quali VELO (*Vertex Locator*), che è un sistema di tracciamento a strip di silicio in grado di misurare le tracce di particelle cariche nella regione prossima al punto d'interazione e che permette di determinare con precisione la posizione dei vertici di decadimento; due rivelatori Čerenkov (*RICH*), necessari per l'identificazione delle particelle cariche (*particle-id*); calorimetri elettromagnetici e adronici, e camere per la rivelazione dei muoni.

Il *trigger* è un sistema in grado di selezionare, tra tutte le collisioni, solamente quelle significative dal punto di vista fisico; i dati corrispondenti alle collisioni selezionate vengono memorizzati per una successiva analisi più approfondita. Il trigger di LHCb nella configurazione attuale è costituito da una parte hardware (*trigger di livello 0* o *L0*) e una parte software (*HLT*, o *High Level Trigger*), a sua volta suddivisa in due livelli (*HLT1* e *HLT2*). Il trigger L0 permette di passare da una frequenza di eventi di 40 MHz, corrispondente alla massima frequenza di *bunch crossing* di LHC, ad una frequenza di circa 1 MHz, che permette di raccogliere

le informazioni da tutti i rivelatori (*readout* dell'intero rivelatore). A questo punto i dati in uscita da L0 vengono processati dai livelli successivi, HLT1 e HLT2, che riducono ulteriormente la frequenza a 5 kHz, infine tali eventi vengono memorizzati su disco.

Per il 2020 è previsto un upgrade dell'elettronica dell'esperimento che permetterà di eliminare il trigger hardware L0 e raccogliere le informazioni dei rivelatori a 40 MHz. Di conseguenza sarà richiesto un potenziamento del trigger software HLT dell'esperimento. In tale prospettiva il lavoro di questa tesi è stato rivolto allo studio delle possibilità offerte dai moderni sistemi di calcolo parallelo per un loro possibile utilizzo nel trigger HLT. In particolare sono stati studiati alcuni algoritmi di trigger per GPU; una delle implementazioni proposte (*FastVeloGPU*) ricalca l'algoritmo esistente, introducendo alcune modifiche necessarie per sfruttare le caratteristiche delle architetture parallele, un'altra implementazione studiata (*FastVeloHough*) utilizza il metodo della trasformata Hough per implementare una parte del tracking.

In questa tesi, dopo un'introduzione alla fisica di LHCb (capitoli 1, 2, 3) si discutono la classificazione e le caratteristiche degli algoritmi di tracking (capitolo 4). Nel capitolo 5 si prende in esame l'algoritmo correntemente utilizzato in LHCb per il trigger con il VELO. Nei capitoli 7 e 8 sono discussi gli algoritmi proposti per il tracking del VELO su GPU. Infine nel capitolo 9 vengono analizzate e confrontate le prestazioni ottenute.

1

Cenni di fisica sui mesoni B

Il quark b (chiamato *beauty* o alternativamente *bottom*) fu scoperto nel 1977 nell'esperimento E288 presso il Fermilab; l'equipe sperimentale fu guidata da Leon M. Lederman [1]. La particella trovata, un mesone di massa di circa 9.4 GeV e composizione $b\bar{b}$, fu chiamata Υ.

Nell'esperimento LHCb le coppie $b\bar{b}$ create nelle collisioni protone-protone danno luogo ad una varietà adroni b, che possono essere sia barioni, composti da tre quark, come ad esempio Λ_b, oppure mesoni, composti da una coppia quark-antiquark. In questo capitolo ci limiteremo a discutere solamente i mesoni B, il cui processo di decadimento è governato dall'interazione debole. I principali mesoni B sono elencati nella tabella 1.1.

Nel capitolo sono descritte caratteristiche del modello standard inerenti la fisica dei mesoni B, il fenomeno dell'oscillazione $B - \bar{B}$, e i meccanismi che portano alla violazione di CP. L'ultima sezione 1.4 discute brevemente alcune differenze tra i collider pp e le beauty-factory e^+e^- (BaBar, Belle).

[1] media sugli autostati di massa B^0_L e B^0_H
[2] media sugli autostati di massa B^0_{sL} e B^0_{sH}

Mesone	Composizione	Massa (MeV)	Vita Media
B^+	$\bar{b}u$	5279.26 ± 0.17	1.641 ± 0.008 ps
B_c^+	$\bar{b}c$	6274.5 ± 1.8	0.452 ± 0.033 ps
B_d^0	$\bar{b}d$	5279.58 ± 0.17	1.519 ± 0.007 ps[1]
B_s^0	$\bar{b}s$	5366.77 ± 0.24	1.516 ± 0.011 ps[2]

Tabella 1.1: Alcune proprietà dei principali mesoni B [24].

1.1 Modello standard

La parte della lagrangiana del modello standard corrispondente alla corrente debole carica può essere scritta come [2]:

$$\mathcal{L}_{\text{Debole,CC}} = -\frac{g}{\sqrt{2}}\left[\overline{u}_i\gamma^\mu\frac{1-\gamma^5}{2}U_{ij}^{\text{CKM}}d_j + \overline{\nu}_i\gamma^\mu\frac{1-\gamma^5}{2}e_i\right]W_\mu + h.c. \tag{1.1.1}$$

La prima parte riguarda i vertici contenenti quark, la seconda parte i leptoni. Il parametro g è la costante di accoppiamento debole, γ^μ sono le matrici di Dirac, $\overline{u}_i$, d_j, $\overline{\nu}_i$, e_i, W_μ sono opportuni operatori di creazione o distruzione. È sottintesa la somma per tutte le generazioni di quark e leptoni (indice i).

La matrice CKM è una matrice di dimensioni 3×3. Gli elementi della matrice sono numeri complessi.

$$U^{\text{CKM}} = \begin{pmatrix} V_{ud} & V_{us} & V_{ub} \\ V_{cd} & V_{cs} & V_{cb} \\ V_{td} & V_{ts} & V_{tb} \end{pmatrix}. \tag{1.1.2}$$

La matrice CKM può essere parametrizzata mediante quattro numeri reali, tre angoli ed una fase: la "parametrizzazione standard" è definita come:

$$U^{\text{CKM}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & c_{23} & s_{23} \\ 0 & -s_{23} & c_{23} \end{pmatrix}\begin{pmatrix} c_{13} & 0 & s_{13}e^{i\delta_{13}} \\ 0 & 1 & 0 \\ -s_{13}e^{i\delta_{13}} & 0 & c_{13} \end{pmatrix}\begin{pmatrix} c_{12} & s_{12} & 0 \\ -s_{12} & c_{12} & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

dove compaiono quattro parametri reali (θ_{12}, θ_{23}, θ_{13}, δ_{13}).

$$c_{12} = cos\,\theta_{12} = cos\,\theta_C;$$

$$s_{12} = sin\,\theta_{12} = sin\,\theta_C;$$

$$c_{23} = cos\,\theta_{23}; \quad s_{23} = sin\,\theta_{23};$$

$$c_{13} = cos\,\theta_{13}; \quad s_{13} = sin\,\theta_{13}.$$

L'angolo θ_C ($= \theta_{12}$) è l'angolo di Cabibbo, pari a circa 13,0° [24].

Una parametrizzazione alternativa particolarmente utile è quella di Wolfenstein: è funzione di quattro parametri reali (λ, A, ρ, η) ed è legata nel seguente modo alla parametrizzazione standard:

$$\lambda = s_{12} = sin\,\theta_C \simeq 0.225;$$

$$A\,\lambda^2 = s_{23};$$

$$A\,\lambda^3(\rho - i\eta) = s_{13}e^{-i\delta_{13}}.$$

Lo sviluppo approssimato fino al terzo ordine in λ può essere scritto come:

$$U^{\mathrm{CKM}} = \begin{pmatrix} 1 - \frac{\lambda^2}{2} & \lambda & A\lambda^3(\rho - i\eta) \\ -\lambda & 1 - \frac{\lambda^2}{2} & A\lambda^2 \\ A\lambda^3(1 - \rho - i\eta) & -A\lambda^2 & 1 \end{pmatrix} + \mathcal{O}(\lambda^4). \tag{1.1.3}$$

Come si può facilmente vedere dall'espansione al terzo ordine in λ, gli elementi di matrice che hanno una fase non nulla sono V_{ub} e V_{td}. La presenza di una fase relativa non nulla tra le costanti di accoppiamento è legata alla violazione di CP.

Nello SM la matrice CKM è unitaria:

$$U^\dagger U = \begin{pmatrix} V_{ud}^* & V_{cd}^* & V_{td}^* \\ V_{us}^* & V_{cs}^* & V_{ts}^* \\ V_{ub}^* & V_{cb}^* & V_{tb}^* \end{pmatrix} \begin{pmatrix} V_{ud} & V_{us} & V_{ub} \\ V_{cd} & V_{cs} & V_{cb} \\ V_{td} & V_{ts} & V_{tb} \end{pmatrix} = I. \tag{1.1.4}$$

Esplicitando la condizione 1.1.4 abbiamo le seguenti nove equazioni, sei delle quali indipendenti:

$$|V_{ud}|^2 + |V_{cd}|^2 + |V_{td}|^2 = 1; \tag{1.1.5}$$

$$|V_{us}|^2 + |V_{cs}|^2 + |V_{ts}|^2 = 1; \tag{1.1.6}$$

$$|V_{ub}|^2 + |V_{cb}|^2 + |V_{tb}|^2 = 1. \tag{1.1.7}$$

$$V_{ud}V_{us}^* + V_{cd}V_{cs}^* + V_{td}V_{ts}^* = 0. \tag{1.1.8}$$

$$V_{ub}V_{us}^* + V_{cb}V_{cs}^* + V_{tb}V_{ts}^* = 0. \tag{1.1.9}$$

$$V_{ud}V_{ub}^* + V_{cd}V_{cb}^* + V_{td}V_{tb}^* = 0. \tag{1.1.10}$$

Le prime tre equazioni rappresentano la condizione di "universalità debole", cioè la somma delle costanti di accoppiamento debole di ogni quark di tipo up con quella di tutti i quark di tipo down è la stessa per tutte le generazioni[3]. Le ultime tre equazioni, che corrispondono alla somma di tre numeri complessi, possono essere interpretati sul piano dei numeri complessi come un triangolo. Tra le equazioni elencate, le più importanti sono quelle che contengono il fattore V_{ub} oppure V_{td}, perché hanno fasi non nulle nell'approssimazione 1.1.3. Tra di esse, quella più

[3] è un concetto simile a quello dell'universalità leptonica, che postula l'eguaglianza delle costanti di accoppiamento debole (ed elettromagnetico) per le tre generazioni di leptoni e, μ, τ.

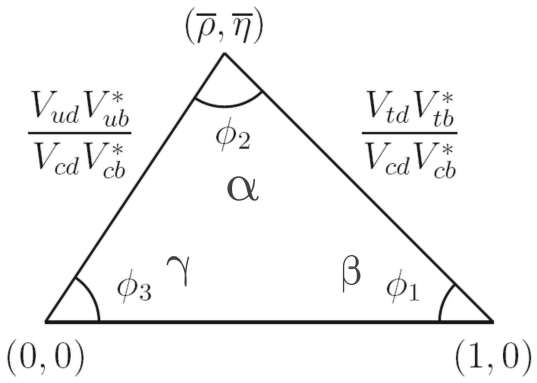

Figura 1.1.1: Uno dei triangoli di unitarietà.

importante è l'equazione 1.1.10 in quanto gli addendi sono di ordine λ^3 e pertanto i lati del triangolo hanno lunghezza dello stesso ordine in λ. Si può di dividere l'equazione 1.1.10 per $V_{cd}V_{cb}^*$ ottenendo:

$$\frac{V_{ud}V_{ub}^*}{V_{cd}V_{cb}^*} + \frac{V_{td}V_{tb}^*}{V_{cd}V_{cb}^*} + 1 = 0.$$

Il triangolo di unitarietà così ottenuto è rappresentato in figura 1.1.1.

1.2 Oscillazione $B^0 - \bar{B}^0$

Un fenomeno caratteristico dei mesoni B è l'oscillazione particella-antiparticella: un mesone B che al tempo $t = t_0$ è un autostato di flavour $|B^0\rangle$ può evolvere nello stato $|\bar{B}^0\rangle$ in un istante $t > t_0$. Il processo che porta ad una variazione del numero di flavour di due unità; nel caso dell'oscillazione $B - \bar{B}$ si ha $|\Delta B'| = 2$ dove B' è il numero di flavour *beauty*. L'oscillazione infatti porta ad una transizione $b \to \bar{b}$ e il numero di flavour è definito come la differenza tra numero di quark b e numero di antiquark b: $B' = -(N(b) - N(\bar{b}))$. L'oscillazione è descrivibile nell'ambito del modello standard con i diagrammi a box di figura 1.2.1.

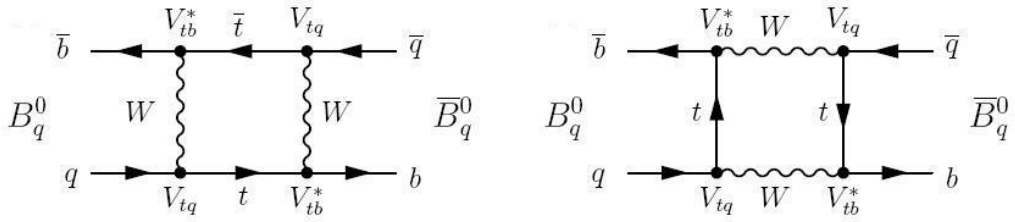

Figura 1.2.1: Diagrammi a box responsabili dell'oscillazione $B - \bar{B}$, all'ordine più basso. Il quark "q" può essere d (down), oppure s (strange).

È possibile ad ogni modo descrivere il processo di oscillazione tramite la seguente equazione di Schrödinger:

$$i\hbar\frac{\partial}{\partial t}\Psi(t) = \mathcal{H}\Psi(t),$$
$$\Psi(t) = a(t)\,|B^0\rangle + b(t)\,|\bar{B}^0\rangle\,.$$

Qui B^0 e $\bar{B}^0$ sono autostati di flavour. Assumendo che la simmetria CPT sia conservata, si può mostrare che l'hamiltoniana si può scrivere come [3]:

$$\mathcal{H} = \mathbf{M} + \frac{i}{2}\mathbf{\Gamma} = \begin{pmatrix} M - i\frac{\Gamma}{2} & M_{12} - i\frac{\Gamma_{12}}{2} \\ M_{12}^* - i\frac{\Gamma_{12}^*}{2} & M - i\frac{\Gamma}{2} \end{pmatrix}. \tag{1.2.1}$$

ove M, Γ, M_{12} e Γ_{12} sono parametri reali. Si può diagonalizzare l'hamiltoniana ottenendo i seguenti autostati di massa: (L sta per "light" e H per "heavy"):

$$\begin{aligned} |B_L\rangle &= p\,|B^0\rangle + q\,|\bar{B}^0\rangle \\ |B_H\rangle &= p\,|B^0\rangle - q\,|\bar{B}^0\rangle \end{aligned}$$

con $|q|^2 + |p|^2 = 1$. Il rapporto q/p è:

$$\left(\frac{q}{p}\right)^2 = \frac{M_{12}^* - \frac{i}{2}\Gamma_{12}^*}{M_{12} - \frac{i}{2}\Gamma_{12}}.$$

Gli autovalori μ_L, μ_H possono essere scritti come:

$$\mu_L = \mu - \frac{\Delta\mu}{2} = (M - \frac{\Delta M}{2}) - i\frac{(\Gamma - \frac{\Delta\Gamma}{2})}{2},$$
$$\mu_H = \mu + \frac{\Delta\mu}{2} = (M + \frac{\Delta M}{2}) - i\frac{(\Gamma + \frac{\Delta\Gamma}{2})}{2},$$

con

$$\begin{aligned} \mu &= M - i\frac{\Gamma}{2}; \\ \Delta\mu &= \mu_H - \mu_L = 2\sqrt{\left(M_{12} - \frac{i}{2}\Gamma_{12}\right)\left(M_{12}^* - \frac{i}{2}\Gamma_{12}^*\right)}; \\ \Delta m &= M_H - M_L = Re(\Delta\mu); \\ \Delta\Gamma &= \Gamma_H - \Gamma_L = -2\,Im(\Delta\mu). \end{aligned}$$

Dunque gli autostati di massa evolvono come:

$$\begin{aligned} |B_L(t)\rangle &= e^{-i\mu_L t}\,|B_L\rangle = e^{-(\Gamma_L/2)t}e^{-iM_L t}\,|B_L\rangle\,, \\ |B_H(t)\rangle &= e^{-i\mu_H t}\,|B_H\rangle = e^{-(\Gamma_H/2)t}e^{-iM_H t}\,|B_H\rangle\,. \end{aligned}$$

Partendo da uno stato $|B^0_{phys}(t)\rangle$ tale che sia uno stato puro a $t = 0$, $|B^0_{phys}(0)\rangle = |B^0\rangle$, lo stato del mesone al tempo t è:

$$|B^0_{phys}(t)\rangle = g_+(t)\,|B^0\rangle + (q/p)g_-(t)\,|\bar{B}^0\rangle\,, \tag{1.2.2}$$

con

$$g_{\pm}(t) = \tfrac{1}{2} \left(e^{-i\mu_H t} \pm e^{-i\mu_L t}\right), \tag{1.2.3}$$

$$|g_{\pm}(t)|^2 = \tfrac{1}{2} e^{-\Gamma t} \left[\cosh\left(\frac{\Delta\Gamma}{2}t\right) \pm cos\left(\Delta m\, t\right)\right], \tag{1.2.4}$$

dove il termine $\cosh\left(\frac{\Delta\Gamma}{2}t\right)$ può essere trascurato nel caso del B_d^0 in quanto $\Delta\Gamma \simeq 0$, ciò non è valido nel caso del B_s^0. Analogamente, per lo stato $|\bar{B}^0_{phys}(t)\rangle$ si ha:

$$|\bar{B}^0_{phys}(t)\rangle = g_+(t)\,|\bar{B}^0\rangle + (p/q) g_-(t)\,|B^0\rangle\,, \tag{1.2.5}$$

pertanto si ottiene:

$$\left|\langle B^0_{phys}(t)|\bar{B}^0\rangle\right|^2 = \left(\frac{q}{p}\right)^2 e^{-\Gamma t} \left[\cosh\left(\frac{\Delta\Gamma}{2}t\right) \pm cos\left(\Delta m\, t\right)\right],$$

$$\left|\langle \bar{B}^0_{phys}(t)|B^0\rangle\right|^2 = \left(\frac{p}{q}\right)^2 e^{-\Gamma t} \left[\cosh\left(\frac{\Delta\Gamma}{2}t\right) \pm cos\left(\Delta m\, t\right)\right].$$

Le ultime due espressioni descrivono esplicitamente l'oscillazione $B^0 - \bar{B}^0$ oppure $B_s^0 - \bar{B}_s^0$. La frequenza di oscillazione è data da $\nu = \omega/2\pi = \Delta m/2\pi$.
Per il mesone B^0 si ha $\Delta m = 0.507 \pm 0.004\, ps^{-1}$[24] e dunque una frequenza di oscillazione di circa 81 GHz. Per il mesone B_s si ha $\Delta m_s = 17.77 \pm 0.10\,(\text{stat}) \pm 0.07\,(\text{syst})$ ps^{-1}, che corrisponde ad una frequenza di circa $\nu = 2\pi/\omega = 2\pi/\Delta m = 2.8$ THz.

Per B_d^0 e B_s^0 la differenza di massa Δm è molto piccola per cui si può approssimare $M_H \simeq M_L \simeq M$, inoltre $\Gamma_H \simeq \Gamma_L$ per B_d^0, come accennato.

1.3 Violazione di CP

Si possono distinguere tre casi di violazione di CP, violazione diretta, indiretta e un terzo tipo di violazione che è dovuto all'interferenza dei processi di mixing e di decadimento. Introduciamo la seguente notazione per indicare le ampiezze di decadimento da uno stato iniziale (preso il mesone B_d^0 come esempio di stato iniziale):

$$A_f = A(B_0 \to f) = \langle f|\mathcal{H}|B_0\rangle\,,$$

$$A_{\bar{f}} = A(B_0 \to \bar{f}) = \langle f|\mathcal{H}|B_0\rangle\,,$$

$$\bar{A}_f = A(\bar{B}_0 \to f) = \langle f|\mathcal{H}|\bar{B}_0\rangle\,,$$

$$\bar{A}_{\bar{f}} = A(\bar{B}_0 \to \bar{f}) = \langle \bar{f}|\mathcal{H}|\bar{B}_0\rangle\,.$$

1.3.1 Violazione indiretta

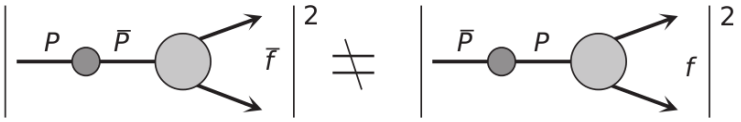

Figura 1.3.1: Violazione indiretta, per una generica particella P [1].

La violazione indiretta (detta anche "violazione nel mixing") è legata all'oscillazione particella-antiparticella. Storicamente, la prima violazione scoperta risale al 1964 ad opera di James Cronin e Val Fitch nell'ambito dello studio dei decadimenti dei kaoni neutri. Fu il primo tipo di violazione di CP a essere scoperto [1].

Sperimentalmente, possiamo misurare questo tipo di violazione nei decadimenti semileptonici misurando la seguente asimmetria:

$$\mathcal{A}_{\mathrm{SL}} = \frac{\Gamma(B^0(t) \to l^- \nu\, X) - \Gamma(\bar{B}^0(t) \to l^+ \nu\, X)}{\Gamma(B^0(t) \to l^- \nu\, X) + \Gamma(\bar{B}^0(t) \to l^+ \nu\, X)}. \tag{1.3.1}$$

In cui il mesone B^0 può decadere solamente in uno stato $l^+ \nu\, X$ solo un mesone mentre il mesone $\bar{B}^0$ (composizione: $b\bar{d}$) può decadere solo in $l^- \nu\, X$.

$$\mathcal{A}_{\mathrm{SL}} = \frac{\left|\langle \bar{B}^0 | B^0_{\mathrm{phys}}(t)\rangle\right|^2 - \left|\langle B^0 | \bar{B}^0_{\mathrm{phys}}(t)\rangle\right|^2}{\left|\langle \bar{B}^0 | B^0_{\mathrm{phys}}(t)\rangle\right|^2 + \left|\langle B^0 | \bar{B}^0_{\mathrm{phys}}(t)\rangle\right|^2}. \tag{1.3.2}$$

Questo rapporto è, a partire dalle equazioni 1.2.2 e 1.2.5:

$$\mathcal{A}_{\mathrm{SL}} = \frac{|g_-(t)|^2 \, |q/p|^2 - |g_-(t)|^2 \, |p/q|^2}{|g_-(t)|^2 \, |q/p|^2 + |g_-(t)|^2 \, |p/q|^2},$$

pertanto (considerando il caso in cui $g_-(t) \neq 0$):

$$\mathcal{A}_{\mathrm{SL}} = \frac{|q/p|^4 - 1}{|q/p|^4 + 1}. \tag{1.3.3}$$

che è diverso da zero quando $\left|\frac{q}{p}\right| \neq 1$ (o equivalentemente $|q| \neq |p|$).

Sperimentalmente, per i mesoni B neutri la violazione di CP nel mixing è molto piccola ($|q/p| - 1 \simeq 10^{-4}$) e difficilmente misurabile sperimentalmente. Al contrario, è relativamente facile da rilevare nel caso dei mesoni K. Questo è dovuto al fatto che K_{L} e K_{S} hanno vite medie molto diverse: $\tau(K_{\mathrm{S}}) \simeq 90\,\mathrm{ps}$, $\tau(K_{\mathrm{L}}) \simeq 51\,\mathrm{ns}$. Per produrre un fascio di K_{L} è sufficiente aspettare che la componente K_{S} decada; questo non è possibile per i mesoni B, visto che hanno vite medie comparabili: $\Delta\Gamma/\Gamma \simeq 4 \times 10^{-3}$ per B^0_d e $\Delta\Gamma/\Gamma \simeq 0.1$ per i mesoni B^0_s.

1.3.2 Violazione diretta di CP

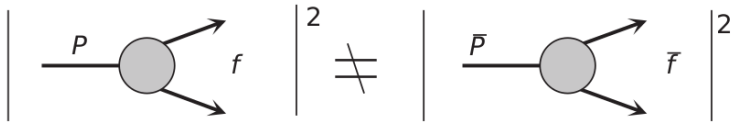

Figura 1.3.2: Violazione diretta di CP [1].

La violazione diretta di CP non ha bisogno dell'oscillazione per verificarsi ed è legata solamente al processo di decadimento. Possiamo dire che si ha, in termini di decay rate:

$$\Gamma(i \to f) \neq \Gamma(\bar{i} \to \bar{f}) \tag{1.3.4}$$

dove i rappresenta lo stato iniziale, o in termini di ampiezze:

$$|A(i \to f)|^2 \neq \left|A(\bar{i} \to \bar{f})\right|^2 \tag{1.3.5}$$

Nei mesoni B carichi non può esserci mixing e in questo caso l'asimmetria corrisponde a:

$$\mathcal{A} = \frac{\Gamma(B^+ \to f) - \Gamma(B^- \to \bar{f})}{\Gamma(B^+ \to f) + \Gamma(B^- \to \bar{f})}. \tag{1.3.6}$$

Un esempio di candidato per la violazione diretta di CP da parte dei mesoni B è ad esempio il decadimento $B^+ \to K^+\pi^0$.

È importante precisare che non può esserci violazione diretta di CP in presenza di un unico canale verso lo stato finale: infatti possiamo scrivere l'ampiezza come:

$$A(i \to f) = |A|\, e^{i\delta} e^{i\phi_W}; \tag{1.3.7}$$

qui δ sta per una fase forte[4] che interviene durante il decadimento e che si assume CP-invariante. L'ampiezza relativa allo stato coniugato di carica sarà:

$$A(\bar{i} \to \bar{f}) = |A|\, e^{i\delta} e^{-i\phi_W}; \tag{1.3.8}$$

come si può notare la fase debole cambia di segno ma la fase forte rimane invariata. Pertanto in presenza di un unico canale $|A(f)| = \left|\bar{A}(\bar{f})\right|$ e non c'è violazione di CP.

Se invece abbiamo due processi concorrenti che portano allo stesso stato finale,

$$A(i \to f) = |A_1|\, e^{i\delta_1} e^{i\phi_{W,1}} + |A_2|\, e^{i\delta_2} e^{i\phi_{W,2}} \tag{1.3.9}$$

mentre per il processo CP coniugato abbiamo (le fasi $\phi_{W,1}$ e $\phi_{W,2}$ cambiano di segno):

$$A(\bar{i} \to \bar{f}) = |A_1|\, e^{i\delta_1} e^{-i\phi_{W,1}} + |A_2|\, e^{i\delta_2} e^{-i\phi_{W,2}}$$

[4] chiamate anche interazioni dello stato finale. Una esempio di interazione dello stato finale è il cosiddetto *rescattering*.

$$|A(i \to f)|^2 = |A_1|^2 + |A_2|^2 + 2\,|A_1|\,|A_2|\,e^{i(\delta_1+\delta_2)}e^{i(+\phi_{W,1}+\phi_{W,2})}$$
$$\left|A(\bar{i} \to \bar{f})\right|^2 = |A_1|^2 + |A_2|^2 + 2\,|A_1|\,|A_2|\,e^{i(\delta_1+\delta_2)}e^{i(-\phi_{W,1}-\phi_{W,2})}$$

Dopo alcuni passaggi si ottiene:

$$|A(i \to f)|^2 - \left|A(\bar{i} \to \bar{f})\right|^2 = -4\,|A_1|\,|A_2|\,sin(\delta_1 - \delta_2)sin(\phi_{W,1} - \phi_{W,2}). \tag{1.3.10}$$

Come si può notare la violazione diretta ci può essere soltanto se contemporaneamente $\delta_1 \neq \delta_2$ e $\phi_{W,1} \neq \phi_{W,2}$.

1.3.3 Violazione dovuta all'interferenza di mixing e decadimento

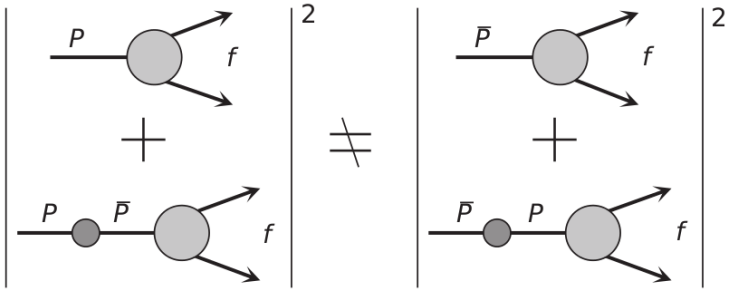

Figura 1.3.3: Violazione di CP dovuta all'interferenza dell'ampiezza di mixing e quella del decadimento [1].

Anche se non vi è violazione diretta o indiretta, può comunque verificarsi la violazione di CP a causa dei termini di interferenza tra le ampiezze di mixing e decadimento. Si può legare il termine di violazione al seguente parametro:

$$\lambda_f \quad = \quad \frac{q}{p}\frac{A(\bar{i} \to f)}{A(i \to f)}.$$
$$Im(\lambda_f) \neq 0 \implies \text{Violazione di } CP.$$

Infatti la larghezza di decadimento per i processi $i \to f$ e $\bar{i} \to f$ può scriversi come :

$$\frac{d\Gamma_{i\to f}(t)}{dt\,e^{-\Gamma\,t}} = \frac{1}{2}\,|A(i \to f)|^2\,(1 + |\lambda_f|^2)\left[\cosh\left(\frac{\Delta\Gamma}{2}t\right) - D_f \sinh\left(\frac{\Delta\Gamma}{2}t\right),\right.$$
$$\left. + C_f \cos(\Delta m\,t) - S_f \sin(\Delta m\,t)\right], \tag{1.3.11}$$

$$\frac{d\Gamma_{\bar{i}\to f}(t)}{dt\,e^{-\Gamma\,t}} = \frac{1}{2}\,|A(\bar{i} \to f)|^2\left|\frac{p}{q}\right|^2(1 + |\lambda_f|^2)\left[\cosh\left(\frac{\Delta\Gamma}{2}t\right) - D_f \sinh\left(\frac{\Delta\Gamma}{2}t\right)\right.$$
$$\left. - C_f \cos(\Delta m\,t) + S_f \sin(\Delta m\,t)\right], \tag{1.3.12}$$

dove

$$\Gamma = \frac{\Gamma_H + \Gamma_L}{2}, \qquad C_f = \frac{1 - |\lambda_f|^2}{1 + |\lambda_f|^2}, \qquad S_f = \frac{2\,\mathcal{I}m\,(\lambda_f)}{1 + |\lambda_f|^2}, \qquad D_f = \frac{2\,\mathcal{R}e\,(\lambda_f)}{1 + |\lambda_f|^2}.$$

I termini $\cosh\left(\frac{\Delta\Gamma}{2}t\right) - D_f \sinh\left(\frac{\Delta\Gamma}{2}t\right)$ possono essere trascurati nel caso del B_d^0, perché $\Delta\Gamma_H \simeq \Delta\Gamma_L$, mentre per B_s^0 non si possono trascurare.

Si consideri ad esempio l'asimmetria dipendente dal tempo relativa ad un decadimento dove lo stato iniziale i è $|B_d^0\rangle$ ed f è autostato di CP:

$$\mathcal{A}_{f_{CP}}(t) = \frac{\Gamma(B_d^0(t) \to f_{CP}) - \Gamma(\bar{B}_d^0(t) \to f_{CP})}{\Gamma(B_d^0(t) \to f_{CP}) + \Gamma(\bar{B}_d^0(t) \to f_{CP})}.$$

Si può assumere che sia trascurabile sia la violazione di CP indiretta, $|p| \simeq |q|$, sia quella diretta, $\left|A(B_d^0 \to f_{CP})\right| \simeq \left|A(\bar{B}_d^0 \to f_{CP})\right|$; ciò implica anche che $|\lambda_{f_{CP}}| \simeq 1$. Utilizzando le relazioni 1.3.11 e 1.3.12, abbiamo $\mathcal{A}_{f_{CP}} = C_{f_{CP}} \cos(\Delta m\, t) - S_{f_{CP}} \sin(\Delta m\, t)$, e poiché $|\lambda_{f_{CP}}| \simeq 1$ allora $C_{f_{CP}} \simeq 0$. Si ottiene:

$$\begin{aligned} \mathcal{A}_{f_{CP}} &\simeq S_{f_{CP}} \sin(\Delta m\, t), \\ S_{f_{CP}} &\simeq \mathcal{I}m(\lambda_{f_{CP}}), \end{aligned}$$

pertanto $Im(\lambda_f) \neq 0$ implica una violazione di CP dipendente dal tempo.

1.4 Differenze tra collisori pp e collisori e^+e^-

Gli esperimenti collisori di elettroni asimmetrici come BaBar (presso SLAC, Menlo Park, Stati Uniti) e Belle (presso l'acceleratore KEKB, a Tsukuba, Giappone) usano la risonanza $\Upsilon(4S)$ per produrre coppie di mesoni B_d. Il mesone $\Upsilon(4S)$ ha composizione $b\bar{b}$ e decade nella quasi totalità dei casi in B_d-$\bar{B}_d$ oppure in B^+-B^-.

In un collider pp il processo che porta alla creazione dei mesoni B è più complicato, essendovi coinvolti contributi QCD, e, nonostante venga necessariamente creata una coppia $b - \bar{b}$, essa può adronizzare in modo differente, ad esempio $p + p \to B_s + \bar{B}_d + X$. (vedi figura 1.4.1).

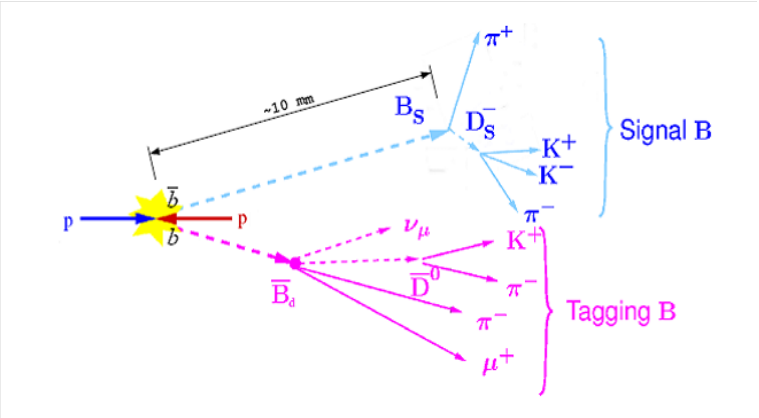

Figura 1.4.1: Creazione incoerente di coppie $B - \bar{B}$ in un collider pp.

Il *tagging* di una particella è un processo che permette di identificare il flavour, cioè di distinguere un mesone B da un $\bar{B}$. Un metodo per distinguere un B da un $\bar{B}$ è quello di usare

un decadimento flavour-specifico. Un esempio di decadimento flavour-specifico è il decadimento semi-leptonico. Ad esempio, $B_d \rightarrow D^-\mu^+\nu_\mu$ implica un decadimento $\bar{b} \rightarrow \bar{c}\mu^+\nu_\mu$ e l'aver osservato un μ^+ implica che, al momento del decadimento, il mesone fosse B_d e non $\bar{B}_d$.

2

L'esperimento LHCb

In questo capitolo viene descritto l'esperimento LHCb, elencando le varie componenti del rivelatore, le modifiche hardware previste per l'upgrade e il software di LHCb.

2.1 Introduzione

L'esperimento LHCb è uno degli esperimenti in corso presso l'acceleratore LHC (*large hadron collider)* al CERN (Ginevra). L'LHC è progettato per far collidere fasci di protoni (pp) all'energia nel centro di massa $\sqrt{s} = 14\,\text{TeV}$ con una luminosità massima di $10^{34}\ \text{cm}^{-2}\text{s}^{-1}$. Il tunnel che ospita il collider ha una circonferenza di 27 km ed è scavato ad una profondità media di 100 m. Uno schema del complesso di LHC è riportato in figura 2.1.2.

L'LHCb è uno spettrometro a braccio singolo con copertura angolare di tra 10 e 300 mrad nel piano orizzontale (piano di curvatura del magnete) e tra 10 e 250 mrad nel piano verticale. Il corrispondente intervallo di pseudorapidità[1] è approssimativamente $2 < \eta < 5$. La luminosità dei fasci in collisione presso LHCb è inferiore rispetto agli altri esperimenti di LHC, nel 2011 e nel 2012 è stata compresa $2 \times 10^{32}\,\text{cm}^{-2}\text{s}^{-1}$ e $4 \times 10^{32}\,\text{cm}^{-2}\text{s}^{-1}$; essa è mantenuta costante

[1] la pseudorapidità è definita come $\eta = -ln(tan(\theta/2))$, dove θ è l'angolo di accettanza, misurato rispetto all'asse del fascio.

durante un *fill* variando dinamicamente la distanza trasversale tra un fascio e l'altro (*lumi-leveling*). Nel 2011 l'energia dei fasci è stata di 3.5 TeV (7 TeV nel centro di massa), mentre nel run del 2012 è stata di 4 TeV per fascio (8 TeV nel centro di massa).

I rivelatori di LHCb sono ottimizzati per misurare tracce di particelle *in avanti*, che formano cioè angoli piccoli con il fascio originario. La scelta è determinata dal fatto che la maggior parte degli adroni con *beauty* vengono emessi a piccoli angoli ed entrambi nella stessa direzione. La distribuzione angolare attesa è rappresentata nella figura 2.1.3 (a destra). La sezione d'urto per i processi del tipo $pp \to b\bar{b} + X$ a 7 TeV è dell'ordine dei 300 μbarn [5] (vedi anche figura 2.1.3, a sinistra).

La disposizione dei sub-detector di LHCb è schematizzata in figura 2.1.1. Il sistema di coordinate globale di LHCb, definito in coordinate cartesiane (x, y, z), è orientato nel seguente modo: l'asse z è disposto lungo la direzione del fascio (direzione positiva verso il magnete, negativa verso il VELO), l'asse y è rivolto verso l'alto e l'asse x, guardando il rivelatore VELO dal magnete, è rivolto verso destra. Il sistema di coordinate è dunque destrorso. L'origine degli assi è posta nella zona in cui i due fasci vanno a collidere, che si trova all'interno del VELO.

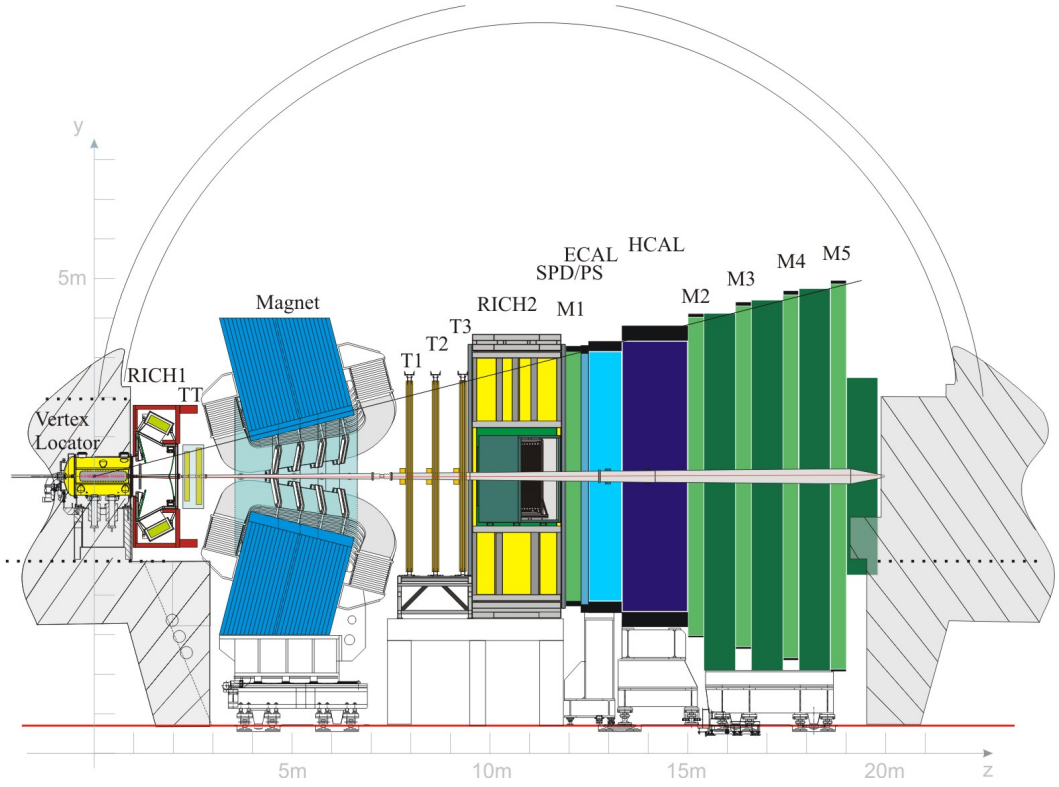

Figura 2.1.1: Disposizione dei sub-detector di LHCb.

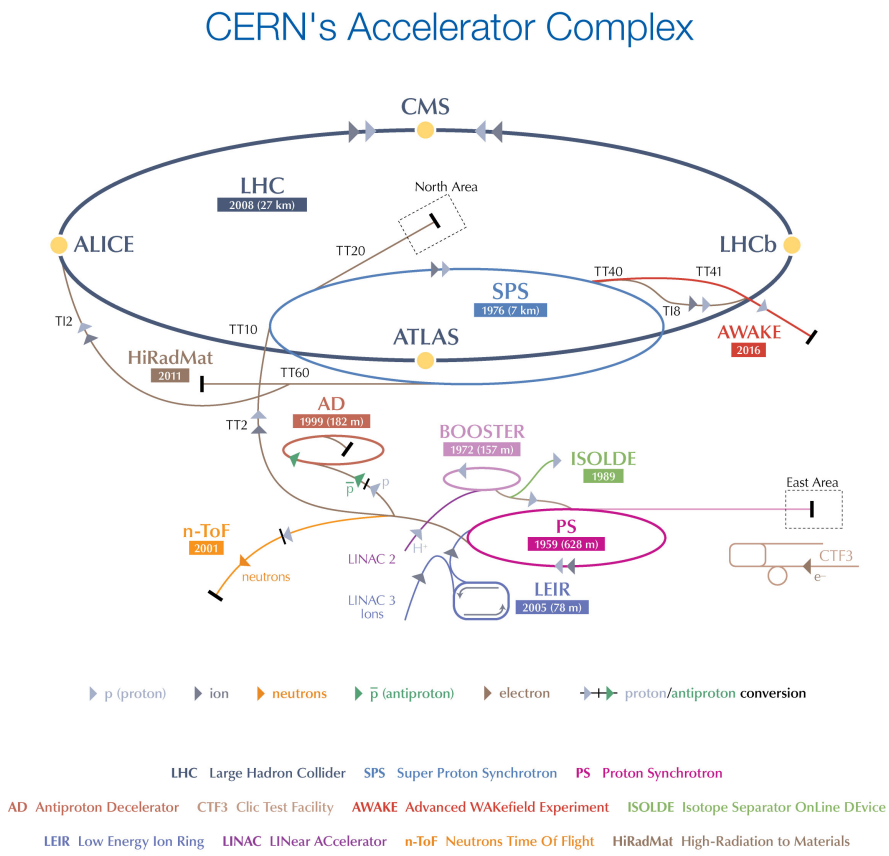

Figura 2.1.2: Complesso dell'acceleratore del CERN.

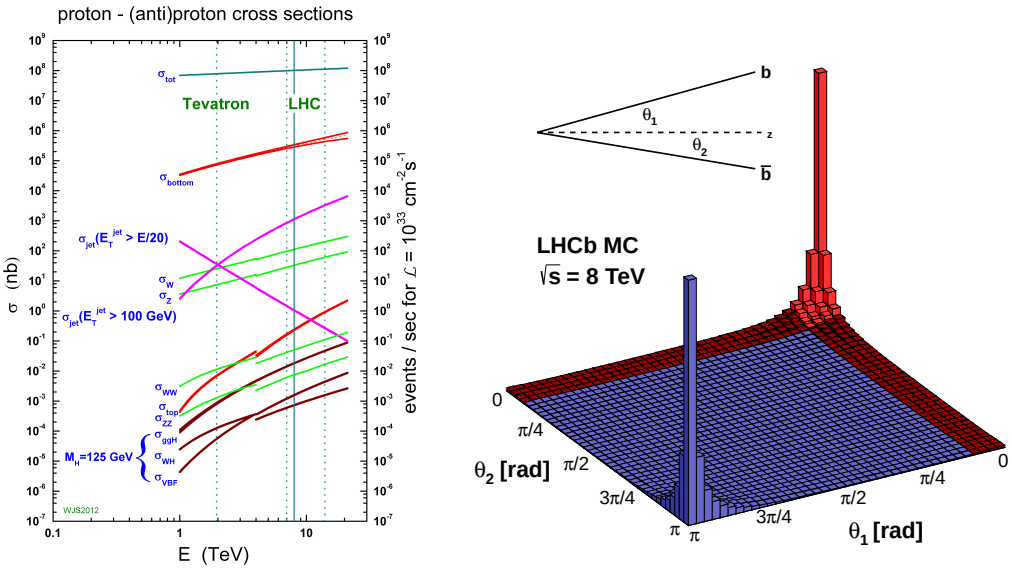

Figura 2.1.3: A sinistra: sezione d'urto per vari processi nelle collisioni pp in funzione dell'energia nel centro di massa $\sqrt{s}$ [52]. A destra: distribuzione angolare delle particelle contenenti quark b o $\bar{b}$ ottenute per mezzo di una simulazione. L'angolo θ è l'angolo tra la traccia e l'asse del fascio. L'intervallo di accettanza di LHCb è evidenziato in rosso [53].

2.2 Configurazione attuale

In questa sezione sarà descritta la configurazione attuale del rivelatore di LHCb, usata nel Run 1 (Novembre 2009 - Febbraio 2013[2]). I sottosistemi che compongono il rivelatore sono quello del tracking (VELO, stazione TT, tracker principale composto da T1, T2, T3), quello del particle-id (rivelatori RICH1 e RICH2), i calorimetri (SPD/PS, ECAL, HCAL) e le camere a muoni (M1-M5).

2.2.1 Sistemi di Tracking

I sistemi di tracking sono costituiti dai rivelatori VELO (*VErtex LOcator*), il rivelatore TT (chiamato *Trigger Tracker* o anche *Tracker Turicensis*) e il tracker principale, composto dalle stazioni T1, T2, T3 (vedi figura 2.1.1).

2.2.1.1 Il rivelatore di vertici VELO (VErtex LOcator)

La funzione del rivelatore di vertici VELO (VErtex LOcator) [6, 37] è quella di determinare con precisione la posizione dei vertici di decadimento (primari e secondari) e i corrispondenti parametri di impatto. Ricordiamo infatti che i mesoni B hanno vite molto brevi, la lunghezza media di decadimento (data da $L = c\tau\beta\gamma$, dove τ è la vita media) è dell'ordine del millimetro. Infatti prendendo come momento tipico di un mesone B 100 GeV ($\beta\gamma = \frac{|\vec{p}|}{m} \sim 20$), si ha $L \sim 10mm$. La risoluzione sul parametro di impatto è migliore di 35 μm per tracce con momento trasverso $p_T > 1$ GeV. La corrispondente risoluzione sul tempo proprio di decadimento è dell'ordine di 30-50 femtosecondi. Il campo magnetico in cui si trova il VELO è piccolo, inferiore a 0.05 T in modulo (confrontare figura 4.3.4), pertanto le tracce possono essere considerate rettilinee.

Il rivelatore è composto da 42 sensori di tipo R e 42 sensori di tipo ϕ, più altri quattro sensori R di "pileup" o "veto station" che non sono considerati non facenti parte del rivelatore VELO vero e proprio. I sensori di tipo R hanno forma semicircolare e dispongono di 2048 strip di silicio a forma d'arco di cerchio; le strip sono disposte in quattro settori circolari composti da 512 strip. I sensori ϕ, invece, hanno 2048 strip disposte radialmente e in due corone circolari; quella interna contiene 682 strips, mentre quella esterna 1366 (vedi figura 2.2.2). I sensori sono disposti come riportato nella figura 2.2.1. Sono organizzati in 42 coppie di sensori r e ϕ definite *moduli*; ciascun modulo $r\phi$ copre un semicerchio. Una coppia di moduli destro e sinistro affiancati copre un cerchio ed è chiamata *stazione*.

L'elettronica del VELO è basata sul chip di frontend chiamato BEETLE [35], che gestisce 128 canali analogici con preamplificatori del tipo "charge-sensitive" e shaper a basso rumore.

[2] la maggior parte dei dati sono stati raccolti nel 2011 e nel 2012

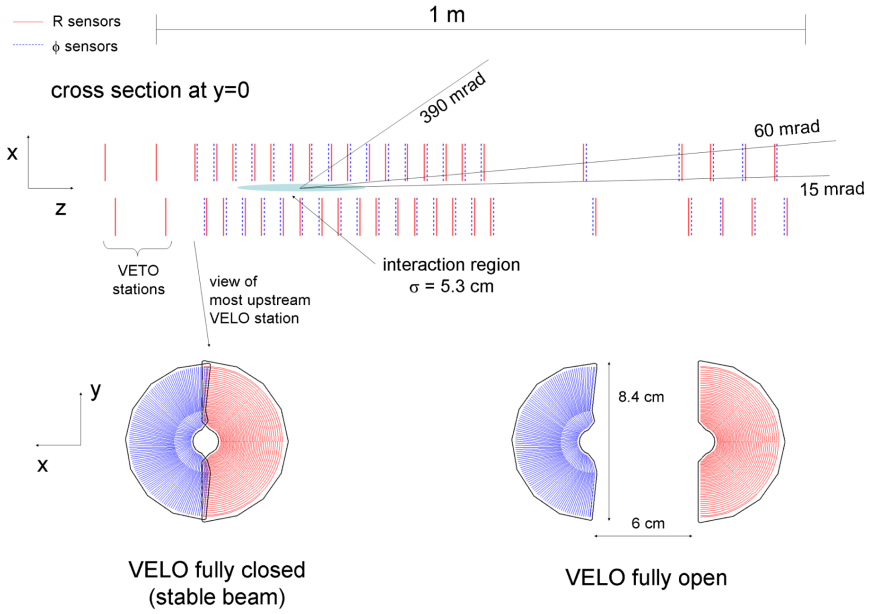

Figura 2.2.1: Disposizione dei moduli all'interno del rivelatore VELO.

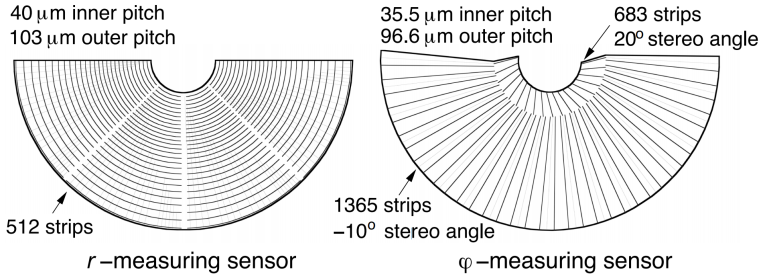

Figura 2.2.2: Struttura di un singolo sensore R e ϕ [54].

Ogni canale del BEETLE misura il segnale proveniente da una strip; pertanto ogni sensore, che ha 2048 strip, contiene 16 chip BEETLE.

Il sensore è diviso in due parti mobili, che posso essere ritratte durante la fase di iniezione del fascio, e sono separate dal vuoto primario da una lamina in lega di alluminio spessa 300 μm, detta *RF foil* (*radio-frequency foil*). La lamina è corrugata in modo da ridurre il materiale attraversato dalle particelle prima della misura.

2.2.1.2 Trigger Tracker (TT)

Il tracker TT ([38], capitolo 5), si trova tra il detector VELO ed il magnete (confrontare figura 2.1.1); contrariamente al VELO il campo magnetico in cui si trova TT non è trascurabile (confrontare figura 4.3.4). Il detector è particolarmente utile per ricostruire tracce prodotte da particelle con basso momento, che vengono curvate dal campo magnetico fuori dall'ango-

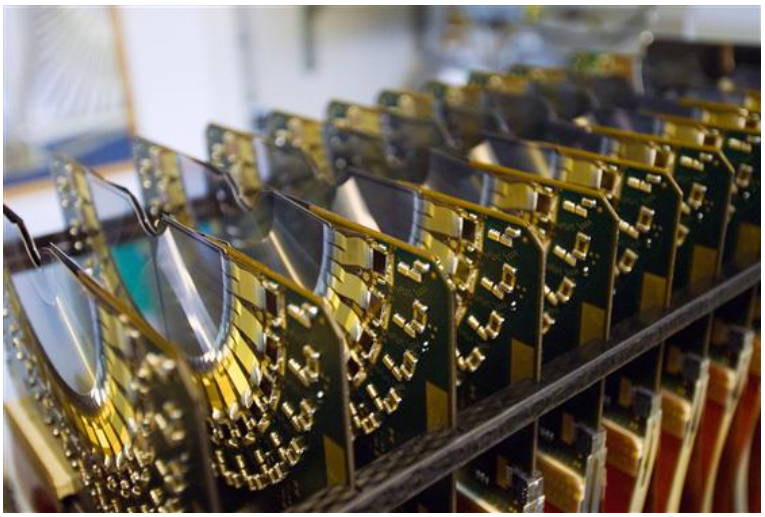

Figura 2.2.3: Foto di alcuni sensori costituenti del rivelatore VELO. Si possono notare le coppie di sensori R e ϕ affiancati, distanziati di qualche millimetro (fonte: CERN).

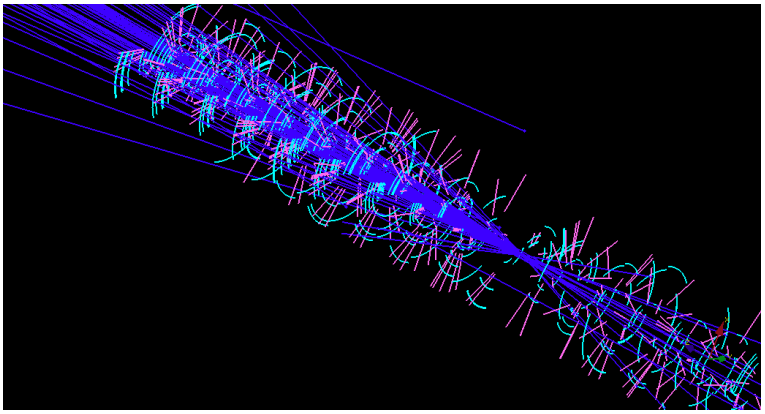

Figura 2.2.4: Alcune tracce ricostruite (in blu) a partire dalle informazioni sulle strip colpite (in ciano quelle di tipo R, magenta quelle di tipo ϕ). Non sono evidenziati i bordi esterni e interni dei sensori.

lo di accettanza di LHCb, oppure è utile per misurare particelle che hanno una vita media relativamente lunga e che decadono all'esterno del VELO.

È composto da due stazioni distanti circa 27 cm lungo l'asse z; ciascuna stazione possiede due strati di sensori a strip di silicio (per un totale di quattro strati). Ciascuno dei quattro strati copre un'area di circa 2 m^2 e ciascun sotto-sensore ha un'area di circa 9.44×9.64 cm^2 con uno spessore di 0.5 mm lungo l'asse z. Il pitch delle strip è di circa 183 μm, permettendo una risoluzione di circa 50 μm nella misura coordinata x di una traccia. I quattro strati sono organizzati secondo la configurazione $xuvx$ (detta configurazione *stereo*), cioè le strip del primo e dell'ultimo strato sono orientate verticalmente, ed effettuano la misura della coordinata x, mentre le strip dei due strati intermedi sono inclinate di -5°(+5°) rispetto alla verticale per lo strato u (v). In questo modo è possibile determinare la terna di coordinate (x, y, z) caratteristiche di una traccia, privilegiando tuttavia la misura della coordinata x, utile per misurare il momento (il magnete curva la traiettoria delle particelle cariche solamente nella proiezione xz).

2.2.1.3 Tracker principale (T1, T2, T3)

Il tracker principale consiste di tre stazioni di tracking chiamate T1, T2, T3, poste tra il magnete e i calorimetri (confrontare figura 2.1.1). Ciascuna della stazioni è suddivisa in due parti, una detta *IT* (*inner tracker*) [40], che è basata su strip di silicio ed un'altra detta *OT* (*outer tracker*) [39] , che è invece un rivelatore a *straw tube* (camera a drift a forma di tubo sottile).

Il rivelatore IT è costituito da quattro strati di sensori al silicio del tutto simili a quelli installati nel detector TT. I moduli sono disposti a formare una croce nella regione prossima al beam pipe (vedi figura 2.2.5), dove il flusso di particelle è massimo, allo scopo di minimizzare l'occupanza dei sensori. Le strip sono orientate secondo lo schema (x, u, v, x), già descritto nella sezione relativa al rivelatore TT. Il pitch delle strip è 198 μm, e porta ad una risoluzione sulla misura della coordinata x simile a quella del detector TT (50 μm).

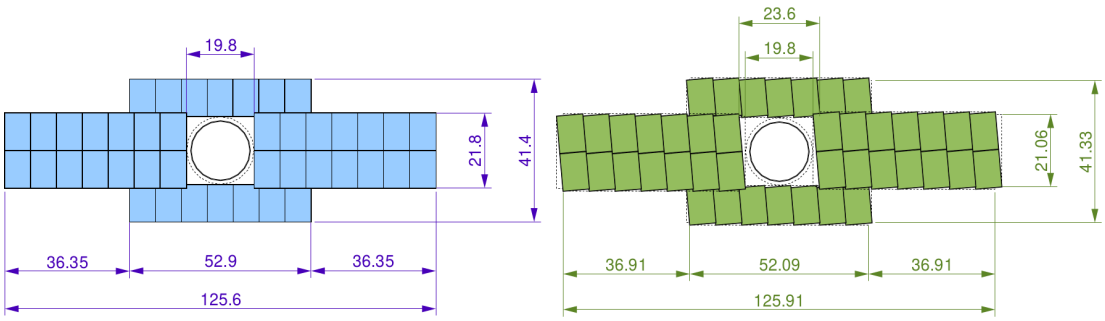

Figura 2.2.5: Disposizione dei moduli nei layer rivelatore IT, a destra: layer x, a sinistra: layer u, inclinato di 5° rispetto alla verticale [40]. Dimensioni in cm.

Ciascuna delle tre stazioni OT è costituita da quattro strati (layer) orientati secondo lo

schema (x, u, v, x). Il detector OT copre la regione di accettanza angolare non coperta da IT. Le dimensioni sono di circa 480 cm lungo l'asse y e 595 cm lungo l'asse x (vedi figura 2.2.6). Un layer consiste in un insieme di moduli di dimensione variabile; ciascun modulo contiene fino a 64 *straw tube*. Una straw tube è rivelatore a gas di forma cilindrica. Ogni straw tube ha un catodo con raggio di 2.45 mm e l'anodo ha un raggio di 12.7 μm, la lunghezza è variabile a seconda della dimensione del modulo; il gas usato è una mistura 70:30 di Argon e anidride carbonica. Il tempo di drift è di circa 50 ns, corrispondente al doppio del tempo due bunch crossing. La risoluzione sulla misura della coordinata x è dell'ordine dei 200 μm.

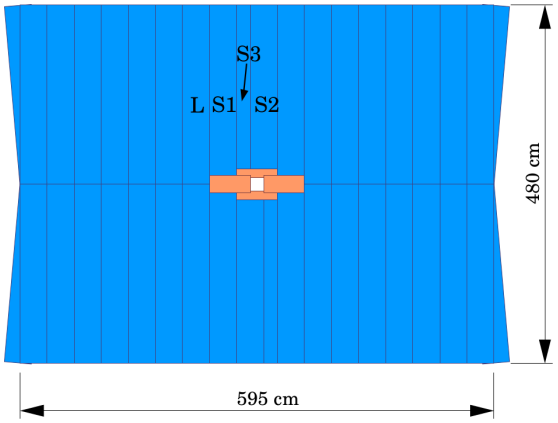

Figura 2.2.6: Geometria del detector OT (in blu) [39].

2.2.2 I rivelatori RICH (Ring-Imaging Čerenkov detector)

Esistono due sensori RICH [41, 33], uno posizionato tra il VELO ed il magnete (RICH1) e un altro tra i tracker T1, T2, T3 e i calorimetri. Il compito principale di questi rivelatori è l'identificazione delle particelle (*particle-id*, o *PID*). Uno degli impieghi tipici è quello di distinguere i kaoni dai pioni.

Il sensore RICH1, schematizzato nella parte sinistra della figura 2.2.7, è dedicato alla rivelazione di particelle di basso momento, in particolare quelle che vengono curvate fuori dall'accettanza dal magnete. Il RICH1 ha come elementi radianti una lastra di aerogel di silice (indice di rifrazione $n = 1.030$ per $\lambda = 400$ nm), che permette la rivelazione di particelle con basso momento (dell'ordine di 1 GeV/c), ed il resto del rivelatore è riempito con perfluorobutano (C_4F_{10}), con indice rifrazione $n = 1.005$. Per misurare il segnale luminoso sia RICH1 che RICH2 usano dei rivelatori ibridi di fotoni (HPD, Hybrid Photon Detector).

In figura 2.2.8 è mostrata la distribuzione dell'angolo Čerenkov in funzione del momento, informazione che permette di distinguere kaoni e pioni.

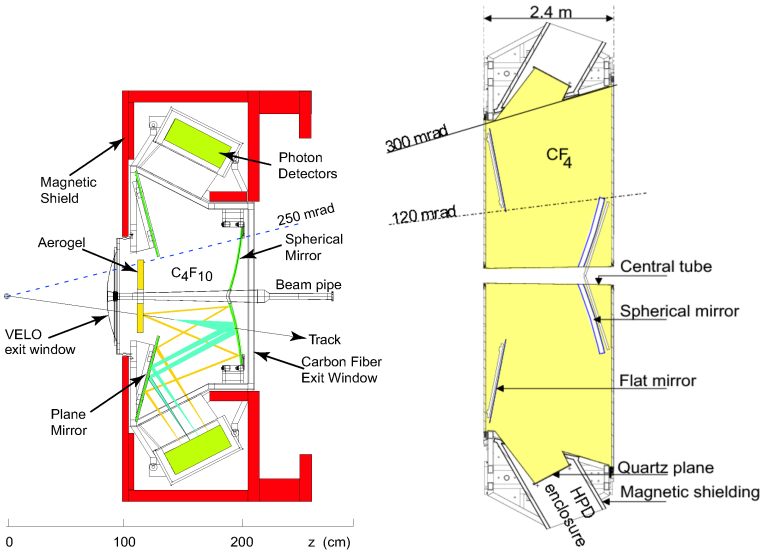

Figura 2.2.7: I rivelatori RICH1 (a sinistra) e RICH2 (a destra).

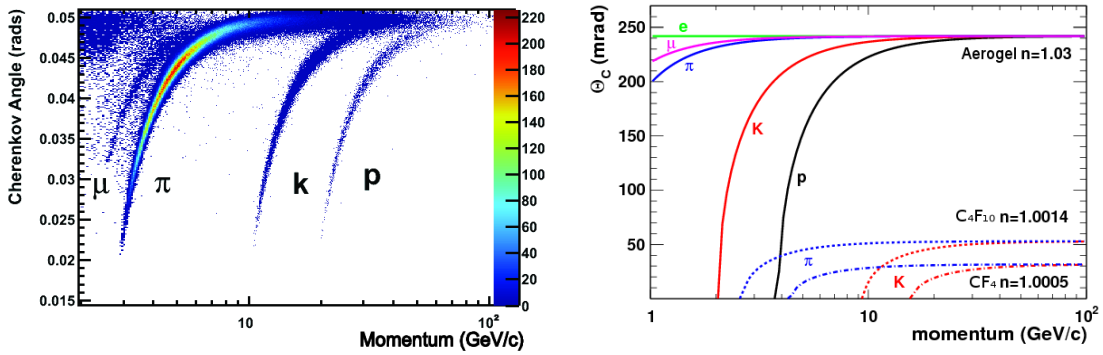

Figura 2.2.8: A sinistra: distribuzione dell'angolo Čerenkov in funzione del momento, per diverse particelle, nel gas C_4F_{10}. Il colore indica il numero di campioni misurati. A destra: angolo Čerenkov in funzione del momento per i materiali radiatori usati nei rivelatori RICH.

2.2.3 Calorimetri

Per calorimetro si intende un sistema in grado di misurare l'energia di una particella. Le particelle interagiscono con il materiale di cui è costituito il calorimetro generando uno sciame di particelle; la radiazione prodotta viene raccolta e misurata permettendo di stimare l'energia della particella incidente. I materiali di cui sono costituiti i calorimetri sono caratterizzati da due grandezze, la lunghezza di interazione elettromagnetica (o lunghezza di radiazione) X_0 e la lunghezza di interazione adronica λ_I[3]. I materiali dei calorimetri elettromagnetici sono

[3]la lunghezza di radiazione rappresenta il cammino medio richiesto da un elettrone ad alta energia ($\gtrsim 1\,\mathrm{MeV}$) per perdere via bremsstrahlung la propria energia fino a raggiungere $1/e$ volte il valore iniziale. La lunghezza di

	numero celle	spessore lungo z	L/X_0	L/λ_I	dimensione celle interne (mm)	dimensione celle centrali (mm)	dimensione celle esterne (mm)
SPD	6016	180 mm	2.0	0.1	40.4×40.4	60.6×60.6	121.2×121.2
PS	6016	180 mm	2.0	0.1	40.4×40.4	60.6×60.6	121.2×121.2
ECAL	6016	835 mm	25	1.1	40.4×40.4	60.6×60.6	121.2×121.2
HCAL	1488	1650 mm	–	5.6	131.3×131.3	–	262.6×262.6

Tabella 2.1: Caratteristiche del rivelatore SPD, PS, ECAL e HCAL [12].

progettati per ottenere $L/X_0 \simeq 25$, dove L è la profondità del rivelatore, in modo da assorbire completamente lo sciame elettromagnetico, mentre quelli adronici massimizzano L/λ_I.

Nell'esperimento LHCb sono presenti un calorimetro elettromagnetico (*electromagnetic* calorimeter o *ECAL*) ed uno adronico (*hadronic calorimeter* o HCAL), ad una distanza di circa 12 metri dal punto d'interazione. Sono inoltre presenti due detector a pad scintillatori, PS (*preshower*) e SPD (*scintillating pad detector*) che aiutano a distinguere fotoni, elettroni ed adroni.

2.2.3.1 I rivelatori SPD (scintillating pad detector) e PS (preshower)

Il rivelatore SPD (*scintillating pad detector*) e il PS (*preshower*) [42, 34] sono posti a monte dei calorimetri (confrontare figura 2.1.1) e permettono di distinguere elettroni, fotoni, e adroni. I due rivelatori SPD e PS sono identici e sono costituiti da 6016 celle di grandezza diversa; la segmentazione è identica a quella del calorimetro ECAL (vedi tabella 2.1). Le celle sono costituite da scintillatore plastico in forma di mattonella (*pad*). Per raccogliere la luce emessa dagli scintillatori e inviarla ai tubi fotomoltiplicatori sono usate fibre a spostamento di lunghezza d'onda WLS[4] (*wavelength-shifting fibers*). I due rivelatori PS e SPD sono intervallati da una lastra di piombo spessa 12 mm.

Il funzionamento del sistema PS/SPD è il seguente: le particelle cariche che raggiungono l'SPD producono luce di scintillazione, contrariamente alla particelle neutre. Attraversando la lastra di piombo, solo i fotoni e gli elettroni provocano una cascata di particelle (*particle shower*). L'informazione proveniente dal rivelatore SPD unita a quella del PS permette di distinguere i tre tipi di particelle, come schematizzato in figura 2.2.9.

interazione adronica, analogamente, rappresenta il cammino libero medio di un adrone prima di ridurre l'energia fino a $1/e$ volte il valore iniziale.

[4]sono fibre ottiche fluorescenti in grado di convertire radiazioni elettromagnetiche ad alta frequenza ed emettere radiazione a frequenza inferiore.

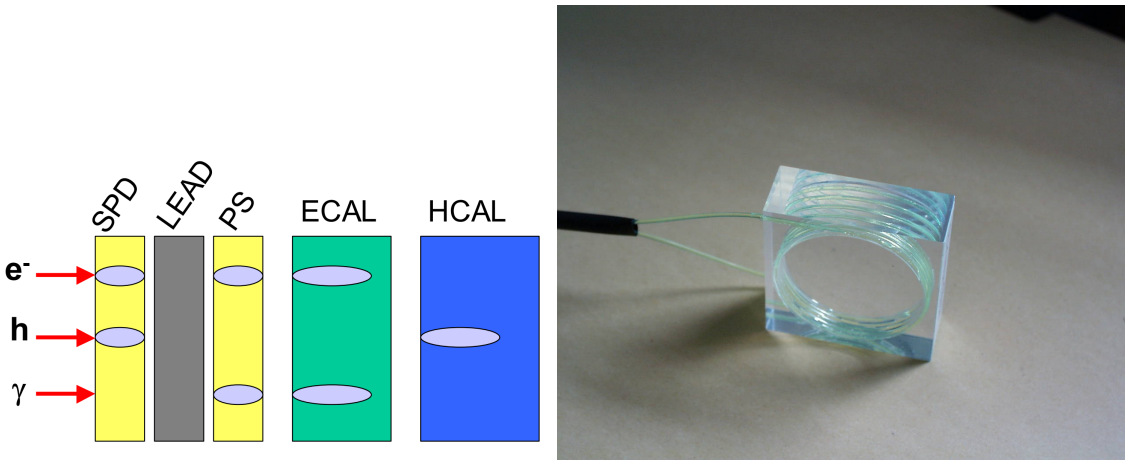

Figura 2.2.9: A sinistra: segnale atteso sulle differenti sezioni del calorimetro, nel caso di un elettrone, di un adrone e di un fotone. A destra: mattonella di scintillatore corredata di fibra spostamento di lunghezza d'onda.

2.2.3.2 Calormetri elettromagnetici

Il compito del calorimetro elettromagnetico (*ECAL*) [36, 42] è quello di misurare l'energia di fotoni ed elettroni.

I moduli costituenti il calorimetro (celle) consistono in un'alternanza di piani di materiale scintillatore e di piombo forati; nei fori sono fatte passare delle fibre WLS che convertono e convogliano la radiazione raccolta dagli scintillatori ai tubi fotomoltiplicatori. Ciascuna cella è formata da 66 lamine di piombo spesse 2 mm e 67 lamine di scintillatori spesse 4 mm. Questa configurazione è denominata "shashlik" ed è rappresentata in figura 2.2.10 (a sinistra). Il calorimetro dispone di 6016 celle di dimensione diversa (vedi tabella 2.1), la segmentazione del rivelatore è la stessa dei rivelatori SPD e PS (figura 2.2.10, a destra).

La risoluzione sulla misura E dell'energia ottenuta è:

$$\frac{\sigma_E}{E} \sim 10\%/\sqrt{E} \oplus 1\%$$

con E espresso in GeV.

2.2.3.3 Calorimetri adronici

Il calorimetro adronico (*HCAL*) [42] è costituito da piani di ferro, scintillatori e fibre a spostamento di lunghezza d'onda (WLS) che raccolgono la radiazione prodotta dagli scintillatori (vedi figura 2.2.12, a sinistra). Lo spessore in ferro costituisce 5.6 lunghezze d'interazione adroniche; il numero di celle è 1468, la segmentazione è riportata in figura 2.2.12.

La risoluzione sull'energia può essere parametrizzata come:

$$\frac{\sigma_E}{E} \sim 80\%/\sqrt{E} \oplus 10\%$$

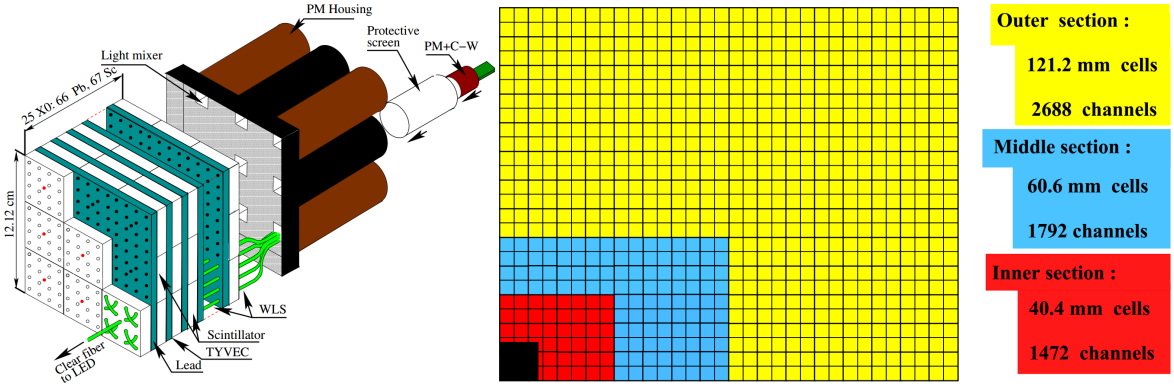

Figura 2.2.10: A sinistra: struttura di un modulo "inner ECAL" del calorimetro elettromagnetico con tecnologia "shashlik". Figura da [36]. A destra: segmentazione di uno dei quattro quadranti del calorimetro elettromagnetico. Le regioni prossime al fascio hanno una granularità maggiore per far fronte alla maggior occupanza. Figura da [42].

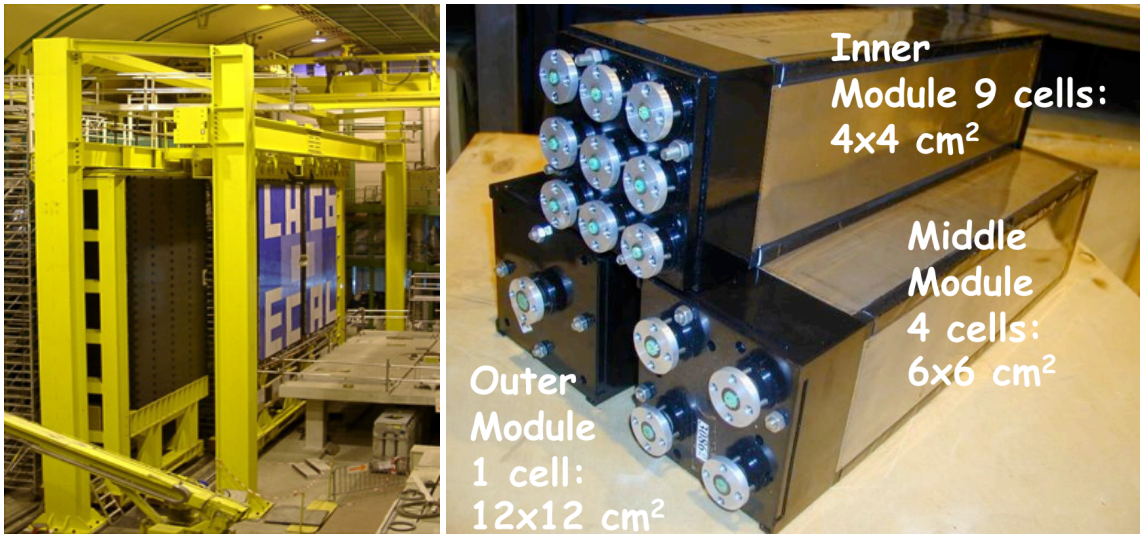

Figura 2.2.11: A sinistra: il calorimetro elettromagnetico (ECAL) di LHCb. A destra: elementi del calorimetro. Nelle zone più interne del calorimetro la risoluzione è maggiore, pertanto esistono elementi in grado di ottenere diverse risoluzioni spaziali.

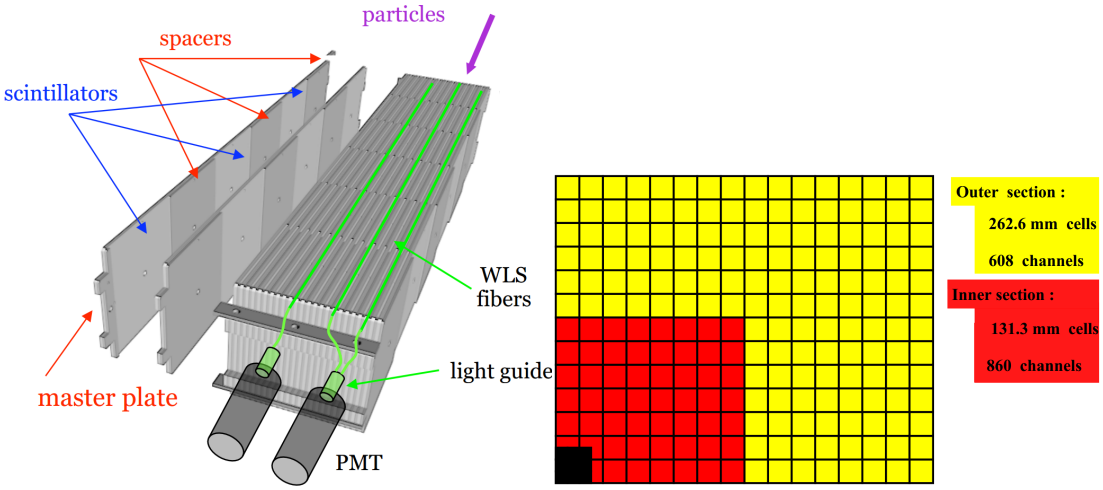

Figura 2.2.12: A sinistra: struttura di un elemento del calorimetro adronico (fonte: CERN). A destra: segmentazione delle celle costituenti il calorimetro. Figura tratta da [42].

con E espresso in GeV.

2.2.4 Camere a muoni

I rivelatori più esterni sono le camere a muoni [43]. Questi rivelatori sono fondamentali per effettuare le selezioni di trigger e l'analisi di canali importanti, quali ad esempio $B_s^0 \to J/\psi(\mu^+\mu^-)\phi$, e i decadimenti rari $B_s^0 \to \mu^+\mu^-$ e $B_d^0 \to \mu^+\mu^-$.

La prima stazione delle camere a muoni M1 si trova a monte dei calorimetri, mentre le rimanenti stazioni (M2-M5) si trovano a valle. Tutti i moduli (M1-M5) sensori sono del tipo MWPC (*Multi Wire Proportional Chambers*); il rivelatore M1 ha nella parte più interna, dove il flusso di particelle è massimo, anche dei rivelatori GEM (Gas Electron Multiplier). Le camere MWPC contengono una miscela di anidride carbonica, argon e tetrafluorometano (CF_4). Le stazioni dalla M1 alla M5 sono intervallate da assorbitori di ferro spessi 80 cm, aventi il compito di selezionare i muoni più energetici.

L'elettronica del trigger L0 effettua una misura del momento trasverso p_T dei muoni basandosi unicamente sull'informazione prodotta dalle camere a muoni assumendo che i muoni provengano dalla regione di interazione; la risoluzione ottenuta è dell'ordine del 20% [8].

2.2.5 Il sistema online (*online system*)

Il sistema online [45, 46] include tutte le componenti necessarie per trasferire i dati dalle schede di front-end all'archiviazione permanente su disco e può essere diviso in tre componenti principali: il sistema di acquisizione dati (*data acquisition* o *DAQ*), il sistema di temporizzazione e controllo (*TFC, timing and fast control system*) e il sistema di controllo dell'esperimento (*ECS, experiment control system*). Uno schema del sistema online è mostrato in figura 2.2.14.

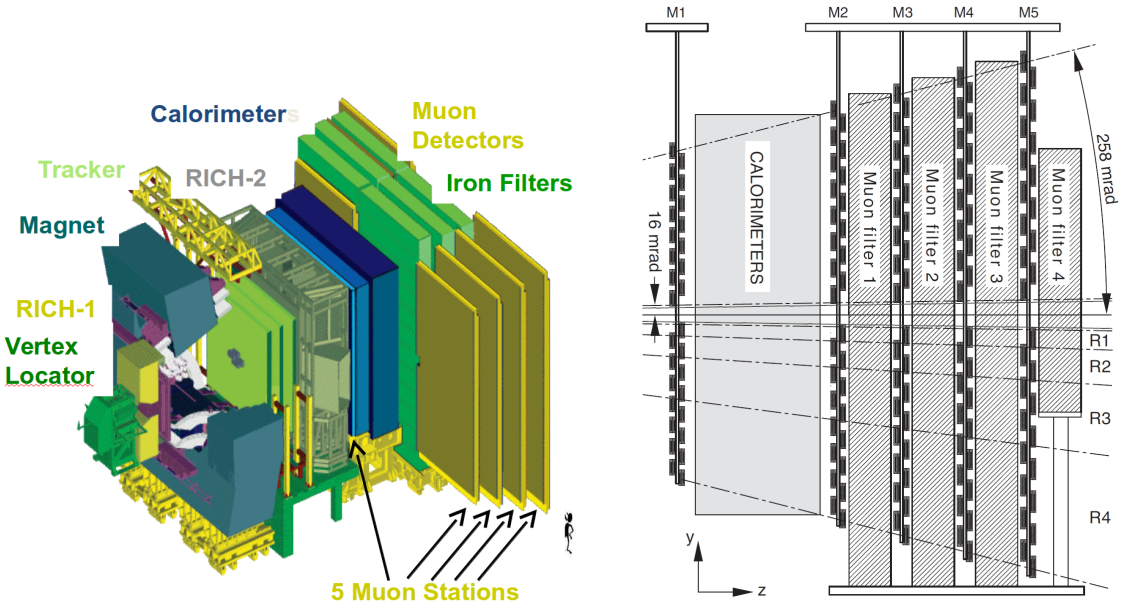

Figura 2.2.13: A sinistra: struttura del rivelatore dei muoni. Sono visibili le lastre di ferro (in verde) che filtrano solo i muoni più energetici. A destra: sezione laterale [31].

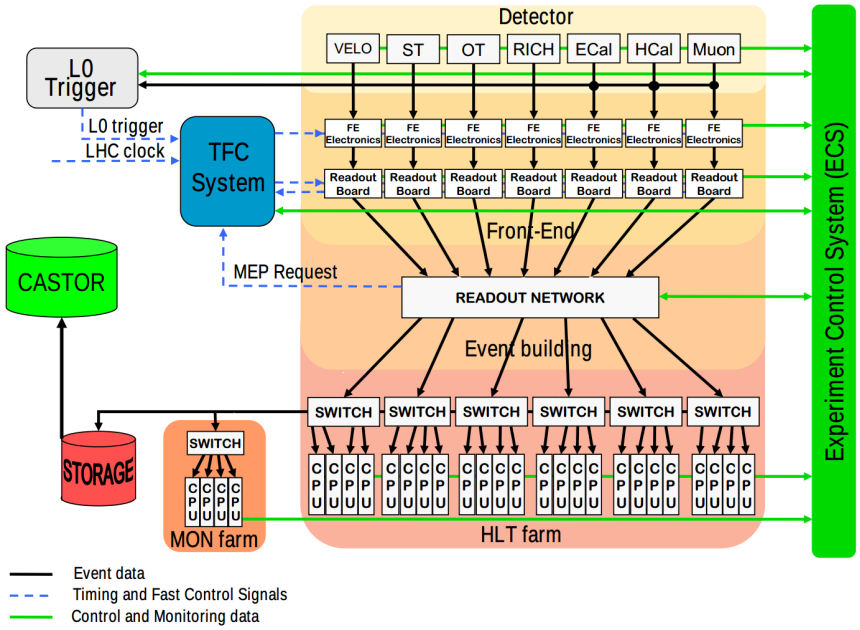

Figura 2.2.14: Infrastruttura dell'*online system* di LHCb.

Il ruolo del sistema di acquisizione dati (*data acquisition system,* o *DAQ*) è raccogliere i dati sugli eventi misurati dai vari sotto-rivelatori e assemblarli per formare eventi completi, ed infine sottoporli al trigger HLT. Il sistema DAQ è composto dall'elettronica di frontend dei rivelatori, dalle schede di readout (TELL1 e UKL1), dalla rete di readout, dalla farm di computer che

implementa il trigger di alto livello HLT (detta *event filter farm* o *EFF*), e dal sistema di archiviazione permanente dei dati.

Il sistema TFC (*timing and fast control system*) opera alla stessa frequenza di clock di LHC ovvero quella del trigger L0 (40 MHz), e il suo compito è quello di specificare una destinazione per i dati prodotti da ciascuna scheda di readout. La destinazione identifica uno dei computer della farm che si occupano del processo di event-building; tutti i frammenti di dati appartenenti allo stesso evento sono inviati ad un unico nodo della farm.

L'ECS (*experiment control system*) è il sistema in grado di monitorare e controllare l'intero sistema online, il trigger e altri parametri utili come pressione, temperatura, tensioni, ecc.

2.2.5.1 DAQ: Elettronica di front-end

L'elettronica di *front-end* (*FEE*, *front-end electronics*) è specifica per ciascun rivelatore e interfaccia ciascun rivelatore con la scheda di read-out (TELL1 o UKL1). Una importante caratteristica della configurazione attuale è il fatto che ogni scheda di FE deve disporre di un buffer nel quale accumulare i dati relativi agli eventi per un tempo pari alla latenza del trigger L0 (4 μs). Poiché la frequenza di bunch crossing nominale è 40 MHz, corrispondente ad un periodo di 25 ns, il buffer deve poter accumulare i dati relativi ad almeno $\frac{4\,\mu\text{s}}{25\,\text{ns}} = 160$ eventi.

2.2.5.2 DAQ: Schede di read out

Le schede di readout usate in LHCb sono di tipo TELL1 [30] oppure UKL1 e si occupano di acquisire il segnale (analogico o digitale) proveniente dai front-end. La scheda TELL1 è una scheda di readout sviluppata appositamente per l'esperimento LHCb ed è usata in tutti i sotto-detector dell'esperimento (eccetto alcuni, come il RICH che usano la scheda UKL1). Le funzionalità della scheda sono finalizzate allo sfruttamento della massima frequenza del trigger L0 (1.11 MHz). L'ingresso di ogni TELL1 consiste in 24 link ottici alla frequenza 1.6 GHz oppure 64 canali di input analogico con ADC a 10 bit e con frequenza di campionamento di 40 MHz. La scheda è basata su FPGA ed è usata per la sincronizzazione degli eventi, per provvedere un buffer di input e output, e per preprocessare i dati applicando la soppressione dello zero[5]. La scheda dispone inoltre di quattro porte Gigabit Ethernet per il collegamento alla rete di readout e può essere alloggiata in un crate standard VME 9U.

Per ridurre l'overhead a carico della rete di readout, la scheda di readout impacchetta più frammenti relativi ad eventi consecutivi in un insieme di *multi-event packet* (*MEP*). Il formato MEP è un protocollo di trasporto simile all'UDP ed è descritto in [21].

[5]la maggioranza degli elementi che costituiscono i sensori non misurano alcun segnale in un determinato evento. È opportuno evitare di trasmettere queste misure nulle ripetute ("zeri") per risparmiare banda.

2.2.5.3 DAQ: Rete di readout

La rete di readout (*readout network*) si occupa della distribuzione dei dati provenienti dai rivelatori. Uno switch Gigabit Ethernet (*GbE*) si occupa di smistare i dati provenienti dalle schede di readout. Sono presenti $\sim$ 300 schede TELL1, e ciascuna di esse possiede 4 porte Gigabit Ethernet. Tutte le schede di readout sono connesse ad una coppia di switch (detti "core router") Gigabit ethernet Dell (Force10) E1200i [32]. I "core router" si interfacciano a ciascuna sotto-farm attraverso degli ulteriori switch TOR ("top of rack") Force 10 S60 e ciascun nodo della farm è connesso tramite un singolo collegamento GbE.

2.2.5.4 DAQ: La farm di CPU

La farm di computer che realizza il trigger di alto livello HLT1 e HLT2 è detta *event filter farm* (EFF) è attualmente costituita da un insieme di server (nodi); in totale la farm possiede circa 29000 core logici [8] (le CPU dotate di *hyperthreading* permettono di implementare due core logici per ogni core fisico). È divisa in 50 sotto-farm.

2.3 Upgrade per il 2020

L'upgrade dei detector di LHCb avverrà durante il periodo di spegnimento del 2018-2019 (*long shutdown 2* o *LS2*). La nuova configurazione sarà utilizzata nel "run 3", previsto per il 2020. In questa sezione saranno elencate le principali modifiche hardware previste. Per un resoconto più dettagliato si può fare riferimento alla lettera d'intento per l'upgrade [47] e al "framework TDR" [48]. Per alcuni rivelatori sono già disponibili i TDR (*techincal data report*) per l'upgrade (ad esempio, per il VELO [49], per il sistema di tracking [50] e il sistema online e di trigger [51]).

2.3.1 Upgrade del VELO

Il VELO attuale, basato sulle strip di silicio, sarà sostituito da un rivelatore basato su pixel di silicio, chiamato *VeloPixel* [49]. La disposizione dei sensori è simile a quella del VELO attuale ed è mostrata in figura 2.3.1. Il chip di frontend che si occupa di processare il segnale proveniente dai sensori a pixel sarà basato sensori sugli ASIC (*Application Specific Integrated Circuit*) denominati *VeloPix*.

Un'importante modifica riguarda l'RF foil, che verrà assottigliato il più possibile; attualmente ha uno spessore di $300\mu m$ e si prevede di raggiungere uno spessore possibilmente inferiore ai 200 μm. L'assottigliamento dell'RF foil è molto importante perché la risoluzione sui vertici di decadimento (e quindi sul parametro d'impatto) dipende fortemente dalla quantità di materiale attraversato dalle particelle prima di raggiungere gli elementi sensibili (pixel o strip) del VELO.

Un'altra caratteristica che contraddistingue il sensore VeloPixel è il sistema di raffreddamento a base di anidride carbonica. Nel sistema proposto, l'anidride carbonica percorre dei micro-canali scavati nel silicio, che funge da supporto meccanico. Lungo i canali, l'anidride carbonica cambia di stato permettendo di raffreddare i chip di frontend VeloPix e i sensori a pixel. La disposizione delle varie componenti del sensore è riportata in figura 2.3.2.

VeloPixel è inoltre progettato per sopportare gli effetti del danneggiamento da radiazione a luminosità fino a $2 \times 10^{33} cm^{-2} s^{-1}$ per la durata prevista del run 3.

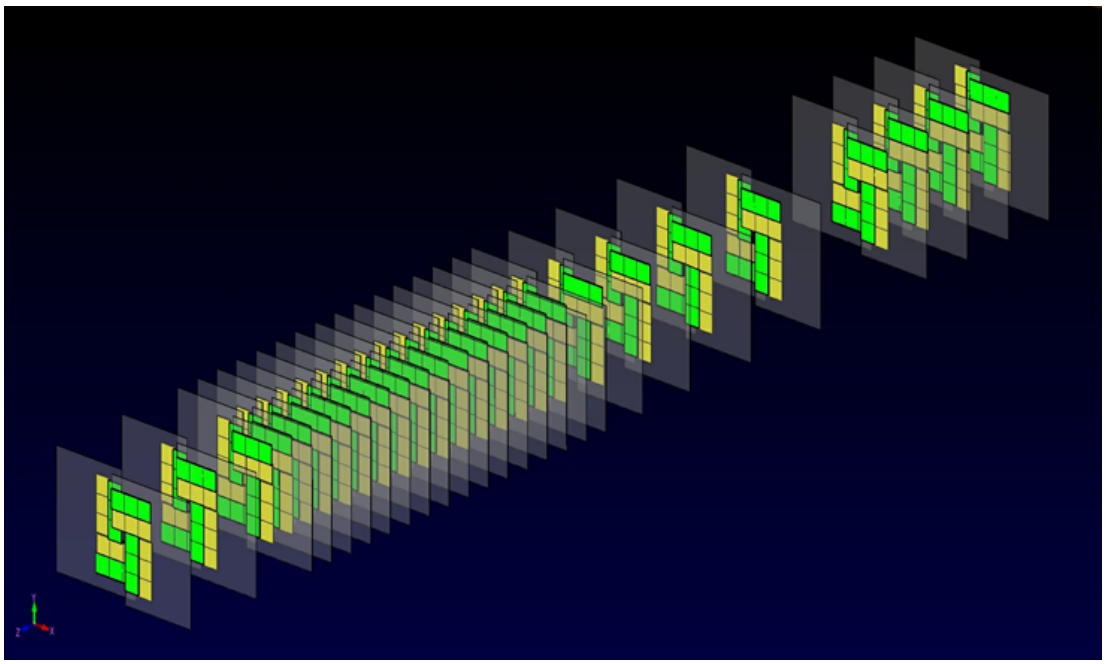

Figura 2.3.1: Struttura proposta del rivelatore VELO basato sui pixel. I rivelatori sono 48 tutti identici organizzati in 24 stazioni (più 2 stazioni di pileup con 4 sensori).

2.3.1.1 Upgrade dei rivelatori TT e IT

Il tracker TT, posto a monte del magnete, sarà sostituito da un nuovo detector chiamato UT (*upstream tracker*) [50], basato su micro strip di silicio, e caratterizzato da aumento della regione di accettanza angolare.

A valle del magnete al magnete, le stazioni IT e OT saranno sostituite da un nuovo rivelatore basato su fibre scintillanti, chiamato SciFi [50] (*scintillating fibers)* che è costituito da fibre lunghe 2.5 m la cui luce di scintillazione è misurata da fotomoltiplicatori al silicio (*SiPM*).

2.3.2 Upgrade dei calorimetro

La struttura del calorimetro rimarrà invariata, il principale problema da affrontare per l'upgrade sarà sostituire l'elettronica di front-end con una in grado di inviare dati al sistema DAQ alla

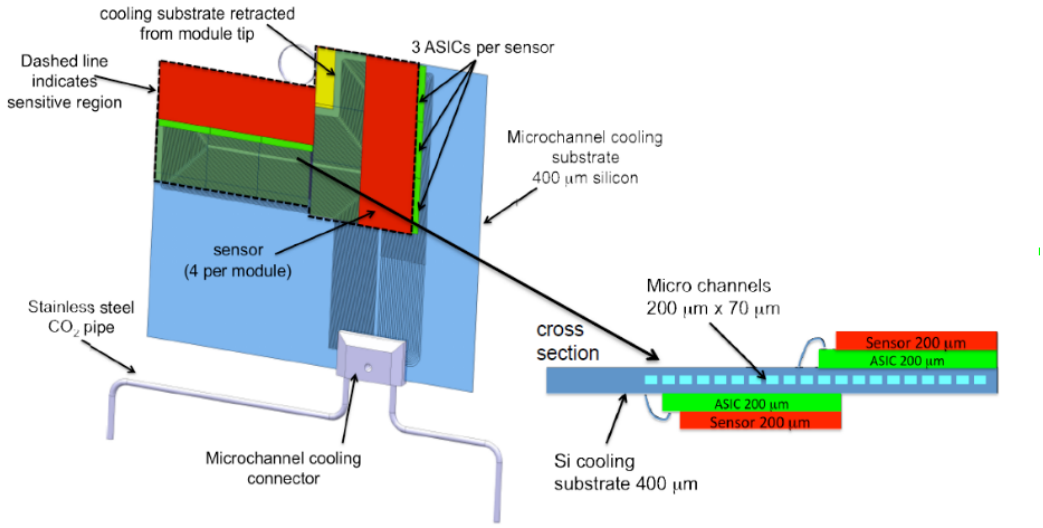

Figura 2.3.2: Una delle soluzioni proposte per il raffreddamento del sensore VeloPixel, basato su micro-canali scavati nel silicio nei quali avviene la transizione di fase di anidride carbonica.

frequenza di 40 MHz [48]. Il rivelatore a pad scintillanti (SPD) e il rivelatore preshower (PS) verranno probabilmente rimossi. In questo modo la capacità di distinguere particelle di basso momento trasverso peggiorerà, tuttavia si ritiene che la riduzione del materiale di fronte ai calorimetri possa migliorare la risoluzione dell'ECAL, e facilitare la calibrazione di entrambi i calorimetri.

2.3.3 Upgrade del RICH

Per quanto riguarda il detector RICH1, si prevede di rimuovere la lastra di aerogel per far fronte alla maggiore occupanza e l'ottica conseguentemente sarà riprogettata [48]. Si prevede invece di lasciare invariata la struttura detector RICH2.

I fotosensori ibridi (*HPD*) saranno sostituiti da fotomoltiplicatori multianodo (*MaPMT*) e l'elettronica di front-end dovrà essere modificata per poter essere compatibile con il readout a 40 MHz.

2.3.4 Upgrade delle camere di muoni

Nell'upgrade, la stazione M1 verrà probabilmente rimossa, perché la risoluzione sul momento dei muoni sarà migliorata attraverso le stazioni di tracking, e l'aumento della molteplicità corrispondente all'aumento di luminosità comunque pregiudicherebbe le prestazioni di M1 [47].

Il readout dell'elettronica di front-end delle camere a muoni è già effettuato a 40 MHz, in quanto partecipa alla decisione del trigger L0, tuttavia la frequenza massima con cui viene

effettuato il readout completo è 1 MHz. Per questo motivo l'elettronica dovrà essere comunque modificata per essere compatibile con il readout completo a 40 MHz. Ad ogni modo, come strategia globale, si prevede di mantenere il più possibile le caratteristiche del sistema esistente.

2.3.5 Upgrade del sistema online

Le modifiche alla parte di DAQ [51] che riguarda la parte del frontend e delle schede di readout sono principalmente legate al fatto che si vuole supportare il readout di tutti i sottorivelatori a 40 MHz. Una modifica prevista per la parte DAQ consiste nello spostare le funzioni di "zero suppression" direttamente nel front-end. Questa scelta permette di ridurre il prima possibile il volume di dati da trasferire alla scheda di readout. Un'altra modifica consiste nella rimozione del buffer nel frontend, necessario per accumulare di dati delle misure in attesa della decisione del trigger L0 (vedi figura 2.3.3).

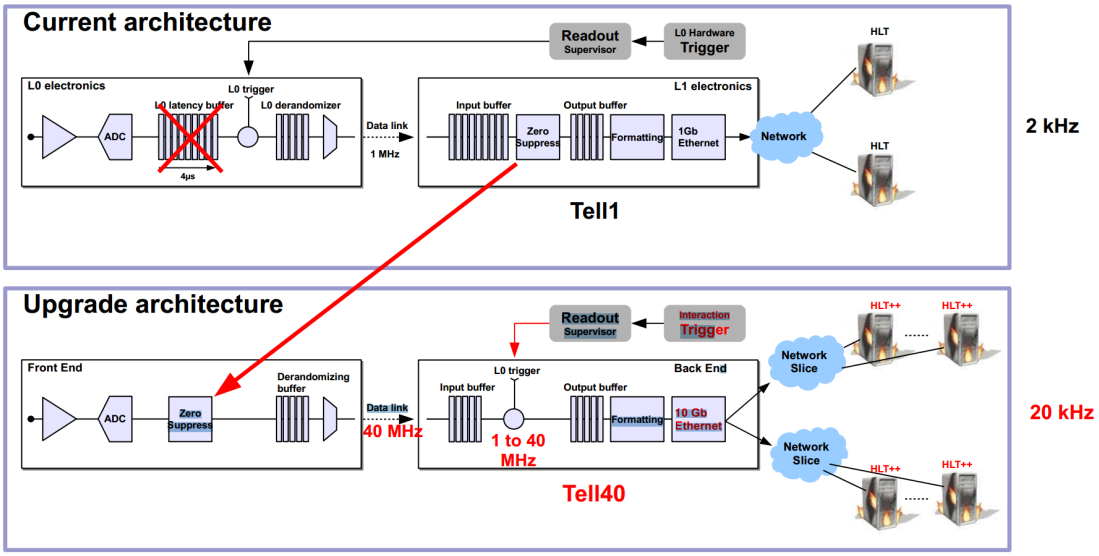

Figura 2.3.3: Sintesi delle modifiche relative all'upgrade del DAQ.

La rete di readout subirà un upgrade per sopportare l'incremento di un fattore ∼40 nella banda di dati in ingresso. Per quanto riguarda la trasmissione dei dati layer fisico la soluzione "baseline" è basata sullo standard 10 Gigabit Ethernet (10GbE). Un'opzione alternativa si basa sui link e switch di tipo InfiniBand, si attende una decisione entro il 2015.

2.4 Software di LHCb: il framework Gaudi

Gaudi [55] è una infrastruttura di sviluppo software, o framework, sviluppata dal CERN per gli esperimenti di fisica delle alte energie ed è dedicato all'analisi dei dati prodotti dagli esperimenti. Il framework Gaudi è utilizzato per realizzare applicazioni di simulazione, trigger di alto livello,

ricostruzione, e analisi. Gaudi è per gran parte costituito da codice scritto in C++, e dispone di una interfaccia python. Le principali applicazioni per LHCb sviluppate con il framework Gaudi sono:

- Gauss [56] è il programma simulazione che permette di generare decadimenti simulati. Comprende Pythia, simulatore di collisioni, Evtgen, programma specializzato nella simulazione di decadimenti del mesone B, e Geant4, simulatore del passaggio di particelle attraverso la materia.

- Boole [57] è il programma di "digitalizzazione", cioè simula la risposta dei rivelatori al passaggio delle particelle, l'input di Boole è costituito dalle particelle simulate da Gauss;

- Moore [58] è il programma che implementa il trigger ad alto livello (HLT), è in grado anche di emulare il trigger di basso livello (L0) nel caso di eventi simulati;

- Brunel [59] è il programma di ricostruzione offline[6], il suo compito è ricostruire il percorso compiuto dalle particelle nel rivelatore;

- DaVinci [60] è il programma di analisi che permette di estrarre le informazioni fisiche d'interesse.

[6]per *online* si intende l'analisi di eventi nel momento in cui arrivano dai rivelatori per poi applicare la decisione di trigger, mentre per *offline* si intende l'analisi di eventi già sottoposti al trigger e registrati su disco oppure nastro.

3

Il trigger di LHCb

In questo capitolo saranno discusse le caratteristiche della del sistema trigger di LHCb. Nella prima sezione sarà decritta la configurazione usata nel Run 1 (2009-2013). Nella seconda sezione saranno discussi i limiti dell'implementazione attuale. Nella terza sezione saranno discusse le modifiche proposte per il run 2 del 2015 e quelle dell'upgrade per il run 3 del 2020.

3.1 Il trigger attuale (2010-2012)

Il trigger di LHCb è suddiviso in due parti, il trigger di livello zero (*L0*), e la seconda di alto livello (*High Level Trigger*, o *HLT*). La prima parte è di tipo hardware, ossia è implementata per mezzo di elettronica dedicata. L'input di L0 non proviene da tutti i rivelatori: è costituito dalle informazioni fornite dalle camere a muoni (M1..5), dai calorimetri adronico e elettromagnetico (HCAL ed ECAL), più i pad scintillatori (SPD) e il sensore preshower (PS). Il trigger L0 è anche predisposto per la ricezione delle informazioni dai sensori di pile-up del VELO, tuttavia questa informazione non è attualmente utilizzata nel trigger.

Il trigger L0, selezionando opportunamente gli eventi riduce il flusso di eventi da quello nominale (40 MHz) a solo un milione di eventi per secondo (il valore massimo nominale è 1.11 MHz). A questo punto la frequenza è sufficientemente bassa da permettere all'elettronica di

Anno	Frequenza di output di L0	Frequenza di Output HLT1	Frequenza di output HLT2
2011	$\simeq$ 870 kHz	$\simeq$ 43 kHz	$\simeq$ 3 kHz
2012	$\simeq$ 1 MHz	$\simeq$ 80 kHz	$\simeq$ 5 kHz

Tabella 3.1: Frequenze di output (eventi/s) di L0, HLT1 e HLT2 [7, 8].

trasmettere i dati (*readout*) di *tutti* i rivelatori di LHCb al livello trigger successivo.

Il trigger di alto livello (HLT), al contrario è un software che viene eseguito in una farm di computer che elabora gli eventi che hanno già passato la selezione di L0. Rispetto al trigger L0, HLT è più flessibile può essere facilmente configurato e modificato.

La frequenza degli eventi in output degli stadi L0, HLT1, e HLT2 negli run del 2011 e 2012 è riportata nella tabella 3.1. Sono valori differenti rispetto a quelli della fase progettuale, nella quale si stimava una frequenza di 40 kHz per l'HLT1 (una volta denominato L1) e 200 Hz per l'output di HLT2 [44].

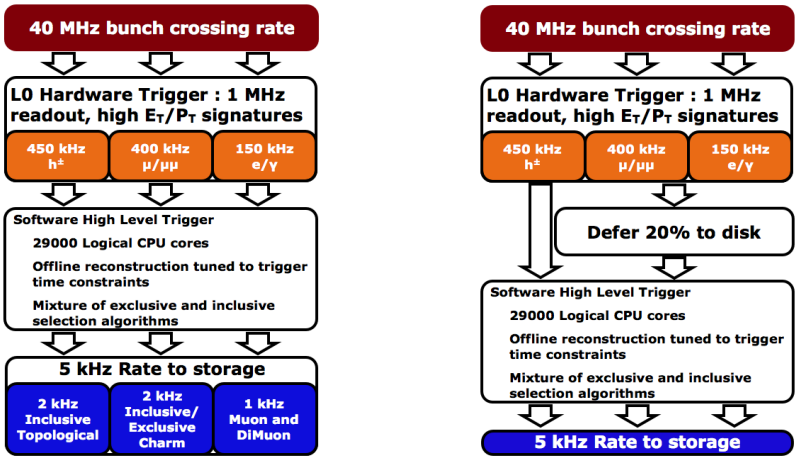

Figura 3.1.1: A sinistra: struttura del trigger di LHCb. A destra: struttura del trigger di LHCb con la strategia del *deffered triggering*.

Nel 2012 è stata implementata una nuova procedura chiamata *deferred trigger*. Poiché l'acceleratore permette di ottenere dei fasci di particelle stabili soltanto per il 30% del tempo, è possibile fare in modo che parte degli eventi vengano salvati temporaneamente su disco, evitando il trigger HLT (operazione detta *buffering*). Gli eventi così registrati vengono processati dal trigger mentre il fascio non è stabile. In questo modo si ottimizza l'utilizzo delle risorse di calcolo: in precedenza la farm di computer era inutilizzata durante le fasi di iniezione del fascio (*inter-fill*). La strategia del *deferred trigger* permette pertanto di dimensionare la procedura tenendo conto del valore medio delle risorse di calcolo utilizzate, non quello di picco, ed è stata testata con successo nel 2012.

Nelle seguenti sottosezioni sarà analizzato in maggiore dettaglio il trigger L0 e HLT.

3.1.1 Il trigger di livello zero (L0)

Il trigger L0 utilizza le informazioni solamente tre sotto-rivelatori: il calorimetro, le camere a muoni e il sistema di pile-up. Le soglie utilizzare per selezionare i diversi canali sono regolate opportunamente per non superare la massima frequenza con la quale effettuare il readout dell'intero rivelatore, cioè circa 1 MHz. È possibile regolare le soglie di L0 a seconda della priorità che si vuole dare ai vari tipi di canale.

La decisione finale è determinata dall'unità chiamata Level-0 Decision Unit (L0DU), che combina le informazioni provenienti dai rivelatori che partecipano alla decisione di livello 0.

3.1.1.1 Trigger L0: veto in base al pile-up

La probabilità che un evento venga accettato dal trigger L0 diventa più alta nel caso in cui vi siano interazioni multiple; eventi con un numero eccessivo in interazioni *pp* sono più complicati da analizzare, sia offline che nel trigger HLT. Pertanto può risultare conveniente filtrarli a livello di trigger, imponendo una soglia massima per il numero di interazioni primarie. La distribuzione prevista di numero di interazioni primarie visibili per bunch crossing in funzione della luminosità è riportata nel grafico 3.1.2. Il veto basato sul pileup non è stato utilizzato durante la fase di presa dati 2011-2012.

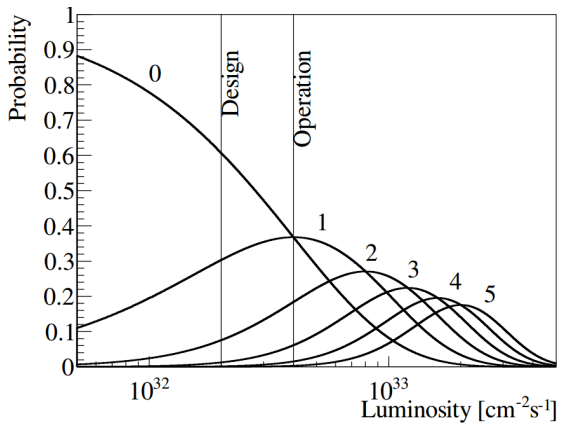

Figura 3.1.2: Distribuzione di probabilità del numero interazioni visibili (μ) in funzione della luminosità, ottenute tramite simulazione Monte Carlo.

3.1.1.2 Trigger L0: trigger calorimetrico

Il trigger calorimetrico per il trigger di livello zero usa le informazioni provenienti dal calorimetro elettromagnetico, quello adronico, il detector *preshower* (*PSD*) e i *pad scintillatori* (*SPD*). L'energia trasversa, sia per il calorimetro elettromagnetico e adronico, è calcolata considerando il segnale depositato di cluster di celle 2×2 della stessa dimensione. L'energia trasversa è

definita come:

$$\mathrm{E_T} = \sum_{i=1}^{4} \mathrm{E_i} sin\,\theta_i,$$

dove l'indice i corre sulla cella i-esima, $\mathrm{E_i}$ è l'energia depositata sulla cella i-esima, e l'angolo θ è l'angolo formato tra l'asse del fascio e la direzione della particella (si assume che la particella provenga dalla zona d'interazione e segua una traiettoria rettilinea verso la cella i-esima).

Il fotone candidato per il trigger L0 è associato al massimo valore di $\mathrm{E_T}$ registrato, solo se ci sono hit sul detector preshower PS e invece sono assenti nei pad SPD, come plausibile per una particella neutra. L'elettrone candidato per il trigger L0 (L0-electron) è selezionato in modo simile a quello dei fotoni, con la differenza che sono richiesti hit anche sul pad scintillatore SPD. Per quanto riguarda l'adrone candidato per il trigger L0, poiché parte dell'energia può essere depositata anche sul calorimetro elettromagnetico, se è misurata una corrispondenza geometrica tra un hit del calorimetro adronico e uno del calorimetro elettromagnetico, allora si considera la somma delle due energie.

Particelle candidate	Soglia 2011	Soglia 2012	Rate
adrone (E_T)	3.5 GeV	3.7 GeV	∼490 kHz
elettrone (E_T)	2.5 GeV	3.0 GeV	∼150 kHz
fotone (E_T)	2.5 GeV	3.0 GeV	(elettroni+fotoni)

Tabella 3.2: Soglie per il trigger calorimetrico per il 2011 e il 2012 [8].

3.1.1.3 Trigger L0: trigger muonico

Il trigger muonico utilizza l'informazione proveniente dalle stazioni M1-M5 per identificare i muoni. L'elettronica di L0 ricerca una sequenza di hit che forma una linea retta attraverso le cinque camere a muoni M1-M5 e che punti verso la regione di interazione nel piano y-z (dove non c'è curvatura da parte del magnete). Nel piano x-y la ricerca è limitata ai muoni con momento maggiore di 0.5 GeV. La posizione di una traccia nelle prime due stazioni permette di determinare il momento trasverso con una risoluzione di circa il 25% rispetto alla risoluzione ottenuta nell'analisi offline.

Nella tabella 3.3 sono riportate le soglie utilizzate negli anni 2011 e 2012 in modo da ottenere circa 400 kHz di eventi con singolo o doppio muone.

Particelle candidate	Soglia 2011	Soglia 2012	Rate
singolo muone, p_{T}	1.48 GeV	1.76 GeV	∼400 kHz
doppio muone ($p_{\mathrm{T_1}} \times p_{\mathrm{T_2}}$)	$(1.296)^2$ GeV^2	$(1.6)^2$ GeV^2	(singolo + doppio muone)

Tabella 3.3: Soglie per il trigger L0 per tigger muonico, per il 2011 e il 2012 [8].

3.1.2 Il trigger di alto livello 1 (HLT1)

Il trigger di alto livello HLT1 è diviso in più *alley*, ossia categorie di selezioni, una dedicata ai canali adronici, una ai canali muonici e una ai decadimenti radiativi (decadimenti che contengono un fotone, come $B_s \to \phi\gamma$). La strategia è quella di *confermare* la decisione del trigger L0. Gli algoritmi che si occupano delle selezioni in HLT sono chiamati *linee di trigger*.

3.1.2.1 Linee di trigger adroniche

HLT1 contiene una linea di trigger che è eseguita per tutti gli eventi accettati da L0, chiamata `Hlt1TrackAllL0`, e la sua funzione è quella di individuare canali adronici, cioè decadimenti che non contengono muoni nello stato finale e pertanto non possono essere selezionati dalle linee di trigger muoniche. Nel 2012, la selezione si basa su un taglio sul momento trasverso $p_{\mathrm{T}} > 1.6$ GeV e sul parametro d'impatto. Le banda totale utilizzata è 58 kHz su un totale di 80 kHz in uscita da HLT1, pertanto le linee adroniche sono la componente dominante in uscita da HLT1 [8]. Uno schema che sintetizza l'alley adronica di HLT1 è presentato in figura 3.1.3. Le componenti principali dell'alley sono: la misura delle tracce nel VELO, che permette di determinare il parametro d'impatto, e il forward tracking (vedi sezione 4.3.4), indicato come "PatForwardTool" nella figura, che permette di misurare il momento trasverso.

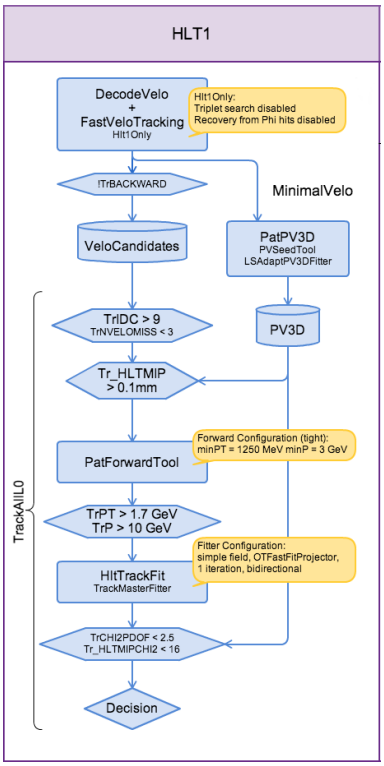

Figura 3.1.3: Diagramma di flusso dell'algoritmo di trigger usato nel 2012 per la parte adronica.

3.1.2.2 Altre linee di trigger

Le linee di trigger muoniche sono eseguite solamente per gli eventi selezionati come singolo o doppio muone dal trigger L0. Viene effettuato un matching tra i muoni candidati e le tracce misurate nel VELO, estrapolando la traccia del VELO verso le camere a muoni e scegliendo una opportuna finestra di ricerca, ma rilassando le soglie sul momento trasverso p_{T}. Altre linee di trigger invece sono dedicate ad eventi classificati come fotoni o elettroni da L0, rilassando anche in questo caso sul momento trasverso minimo p_{T}.

3.1.3 Il trigger di alto livello 2 (HLT2)

Il trigger di livello 2 usa la stessa sequenza di algoritmi usata per la ricostruzione offline. Dopo la ricostruzione effettuata HLT2, sono applicate delle selezioni inclusive ed esclusive[1], che permettono di passare ad una frequenza di eventi tale da poter essere registrata permanentemente per le analisi offline[2] [8].

Le più importanti linee di trigger riguardano un trigger topologico generico, atto a selezionare canali contenenti adroni contenenti il quark b. Nel 2012, la frequenza di eventi in output allocata per questo tipo di trigger è stata 2 kHz su 5 kHz totali. Un altra linea di trigger seleziona i canali muonici, eseguendo un analisi identica a quella effettuata nella fase offline; nel 2012 la frequenza di output è stata di 1 kHz. Una terza linea di trigger seleziona canali contenenti il quark charm c; nel 2012 la frequenza allocata per i canali charm è stata di 2 kHz.

3.2 Limitazioni del trigger corrente

Finora LHCb ha acquisito circa $1\,\mathrm{fb}^{-1}$di dati con $\sqrt{s} = 7\,\mathrm{TeV}$ e $2\,\mathrm{fb}^{-1}$ circa 2 con $\sqrt{s} = 8\,\mathrm{TeV}$. La luminosità istantanea è rimasta compresa tra $2\times10^{32}\mathrm{cm}^{-2}\mathrm{s}^{-1}$ e $4\times10^{32}\mathrm{cm}^{-2}\mathrm{s}^{-1}$. Per il run 2 e il run 3 si prevede di aumentare la luminosità e l'energia del fascio. A causa delle caratteristiche del trigger attuali, l'incremento di luminosità porterà ad una resa[3] del trigger (*trigger yield*) superiore solo i canali muonici, mentre i canali adronici non subiranno miglioramenti. Come si può notare dalla figura 3.2.1, all'aumentare della luminosità la resa del trigger per i canali completamente adronici non è proporzionale alla luminosità come i canali contenenti muoni ma rimane pressoché costante. Questo è dovuto al fatto che la massima frequenza di eventi in uscita dal sistema di readout (TELL1) è di 1 MHz, che deve essere rispettata dalla frequenza di trigger di L0. Ora L0 è molto selettivo e puro per i canali muonici, viceversa per i canali

[1] per *esclusiva* si intende una selezione che coinvolge un canale specifico, mentre per *inclusiva* si intende una selezione che include una *classe* di canali (per esempio una selezione inclusiva è $B_s \to D_s + X$ dove X indica qualsiasi insieme di particelle che può far parte dello stato finale).

[2] vedi tabella 3.1

[3] per *trigger yield* si intende il numero di eventi fisici d'interesse selezionati per unità di tempo.

adronici non ha molto potere di reiezione del fondo. Dunque all'aumentare della luminosità le soglie per i canali adronici devono essere aumentate al fine di ridurre il rate di eventi di fondo. In questo modo però anche l'efficienza per eventi di segnale cala e il trigger yield rimane costante. Per l'upgrade si prevede dunque di rimuovere la limitazione in banda a 1 MHz e di effettuare il readout degli gli eventi a 40 MHz.

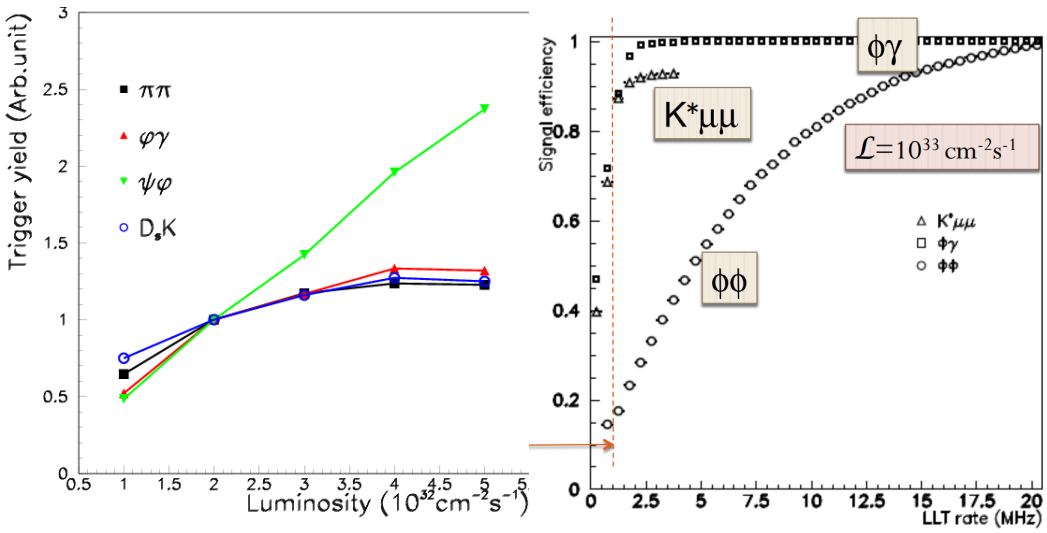

Figura 3.2.1: A sinistra: Efficienza del trigger in funzione della luminosità per alcuni canali di decadimento con il corrente sistema trigger di LHCb. La resa del trigger per i canali adronici ($\pi\pi$, $\phi\gamma$, D_sK) non migliora di molto al crescere della luminosità rispetto ai canali che contengono muoni nello stato finale, a causa delle caratteristiche del trigger. A destra: efficienza del trigger in funzione del rate del Low Level Trigger.

3.3 Upgrade del trigger

In questa sezione saranno descritti brevemente i propositi riguardanti l'upgrade del trigger per il Run 2 (2015) e per il Run 3 (2020).

3.3.1 Upgrade del trigger per il 2015

Una importante modifica consiste nel rendere indipendente l'esecuzione di HLT1 e HLT2, in modo da permettere di applicare il *deferred trigger* non agli eventi selezionati da L0, ma a quelli che hanno già passato la selezione HLT1. Uno schema del trigger proposto per il 2015 è riportato in figura 3.3.1.

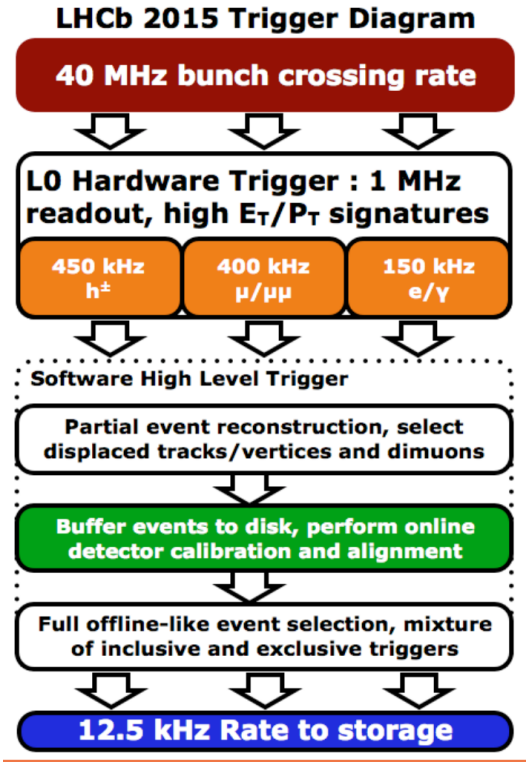

Figura 3.3.1: Struttura prevista del trigger che sarà usato nel 2015. Include la strategia del "deferred trigger".

3.3.2 Upgrade del trigger per il 2020

Nella nuova configurazione proposta [51], che secondo i piani entrerà in funzione nel 2020 dopo il long shutdown 2 (LS2), è schematizzata sinteticamente nella figura 3.3.2. Come accennato, si prevede di eliminare il trigger L0 e di incrementare la capacità di calcolo della farm, stimando un incremento da 29000 a 50000 in termini di core logici disponibili nella farm. Se nel 2020 non sarà disponibile una potenza di calcolo sufficiente per analizzare eventi alla frequenza di 40 MHz, una possibilità è quella di inserire un trigger simile a L0, ma con banda in uscita (cioè numero eventi al secondo in uscita) regolabile da 1 a 40 MHz, in modo da adattare il sistema alle capacità di calcolo della farm.

Il progetto del porting del software di trigger su architetture parallele (GPU o architetture multicore come Intel MIC) nasce appunto in previsione dell'ampliamento delle capacità di calcolo per l'upgrade del 2020. Le architetture parallele permettono di ottimizzare due parametri fondamentali: prestazioni di calcolo per costo dell'unità di calcolo e prestazioni per potenza elettrica richiesta. Chiaramente, il primo parametro influenza l'investimento iniziale, e il secondo permette di risparmio di energia elettrica nel lungo periodo.

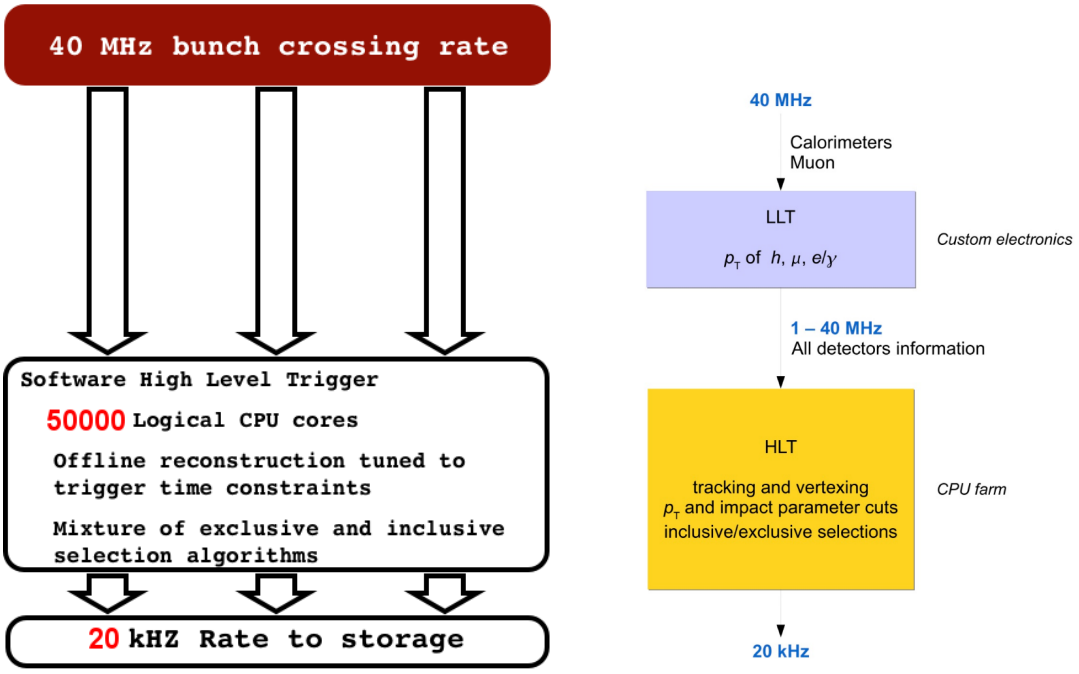

Figura 3.3.2: A sinistra: struttura proposta per il trigger nel 2020 (opzione priva del low level trigger LLT). A destra: schema della configurazione del trigger per l'upgrade. Il trigger L0 è sostituito dall'LLT (low level trigger) che può essere regolato (si parla di *throttling*) da 1 a 40 MHz, a seconda delle prestazioni ottenute da HLT.

4

Algoritmi di tracking

Lo scopo di un algoritmo di tracking è quello di ricostruire la traiettoria e l'impulso delle particelle prodotte durante le collisioni a partire dagli hit rilasciati nei rivelatori. Per *hit* si intende l'informazione sulla posizione di una particella che colpisce il rivelatore (per esempio un hit in un sensore del VELO corrisponde ad una o più strip che identificano la coordinata R o ϕ).

In questo capitolo verranno presentate brevemente le varie tipologie di algoritmi di tracking; inoltre verrà descritto il tracking in LHCb e introdotte alcune definizioni necessarie per valutarne le prestazioni.

4.1 Classificazione degli algoritmi tracking

Un algoritmo di tracking può in generale essere suddiviso in più parti:

- *pattern recognition*: è la parte dell'algoritmo che si occupa di formare una traccia a partire dalle distribuzioni spaziali degli hit (o pattern);

- *track fitting*: è la parte che calcola i parametri della traccia ed effettua l'interpolazione della traccia trovata dall'algoritmo di pattern recognition;

- *clone killing (*rimozione dei cloni): un clone è una copia ripetuta della stessa traccia; il concetto sarà approfondito nel paragrafo 4.4.5. Una fase di "clone killing" è necessaria quando il numero dei cloni generato dalla parte di pattern recognition è ritenuto troppo alto al fine delle successive analisi.

Gli algoritmi di pattern recognition possono essere suddivisi in tre classi:

- algoritmi con approccio **locale**: si ricostruisce una traccia dividendo l'algoritmo in più passi: il primo consiste nel determinare delle tracce candidate formate da un numero minimale di hit, chiamati *seed* o *tracklet*. Le tracce vengono quindi ricostruite a partire dai seed aggiungendo altri hit (è l'approccio seguito dall'algoritmo FastVelo, che sarà discusso in seguito).

- algoritmi con approccio **globale**: questo approccio consiste nel considerare tutti gli hit evitando di considerare sottoinsiemi più piccoli come nel caso dell'approccio locale. Algoritmi che rientrano in questa categoria sono la *trasformata di Hough* (vedi capitolo 8) e il metodo *retina* [25].

- algoritmi basati su altri metodi: uno di questi è il metodo dell'automa cellulare. Un automa cellulare consiste in una griglia di celle; ciascuna cella può assumere un numero finito di stati, come "acceso" e "spento". La griglia può essere realizzata in numero di dimensioni qualsiasi, oppure può essere sostituita da un grafo (nel grafo le celle corrispondono ai vertici e gli spigoli alle relazioni di adiacenza tra celle). Per ogni cella è definito un insieme di celle "vicine", tipicamente costituito dalle sole celle adiacenti. È definito uno stato iniziale (al tempo $t = 0$) assegnando uno specifico stato per ogni cella. Lo stato delle celle evolve nel tempo: lo stato di una cella al tempo $t + 1$ è funzione dello stato delle celle "vicine" al tempo t, e la funzione che determina l'evoluzione delle celle è scelta al fine di ottenere uno specifico risultato. Il più noto esempio di automa cellulare è il "Conway's Game of Life" [9]. Ai fini del pattern recognition, si utilizza un grafo dove ad ogni cella corrisponde un hit e la funzione di evoluzione è scelta in modo da implementare funzioni di pattern recognition. Questo approccio è stato utilizzato per la ricostruzione delle tracce nell'esperimento CMS [26] e in ALICE [10, 11].

In questa tesi abbiamo studiato sia algoritmi di tipo locale (FastVelo, capitolo 5 e FastVeloGPU, capitolo 7) che di tipo globale (trasformata di Hough, vedere capitolo 8).

Per quanto riguarda il *track fitting*, in LHCb l'algoritmo utilizzato è il filtro di Kalman [22]; in HLT1 si preferisce rimandare l'esecuzione del filtro di Kalman soltanto alla fine della catena (confrontare lo schema in figura 3.1.3), in quanto è un algoritmo particolarmente dispendioso in termini computazionali. Comuni alternative al filtro di Kalman sono interpolazioni in tre dimensioni, basate sulla minimizzazione di un opportuna funzione χ^2.

4.2 Algoritmi paralleli di tracking

In questo paragrafo saranno introdotti alcuni concetti utili per la *parallelizzazione* di un generico algoritmo di tracking. Per parallelizzazione di un algoritmo intendiamo un'implementazione che permetta di suddividere il lavoro da effettuare in più parti, facendo in modo che le parti risultanti siano processate simultaneamente dall'elaboratore, sia esso GPU, o CPU o altro. Lo scopo della parallelizzazione è (tipicamente) quello di velocizzare l'esecuzione dell'algoritmo.

Introduciamo alcune definizioni: per *processo* si intende l'istanza di un programma in esecuzione. Un processo può essere diviso in uno o più sottoprocessi, detti *thread.* Le CPU attuali sono concepite per eseguire un numero relativamente piccolo di thread in parallelo, $\mathcal{O}(10)$. Si può affermare che il numero di thread paralleli ottimale per sfruttare appieno una unità di calcolo basata su CPU debba essere o maggiore o uguale al numero di core logici disponibili. È comune trovare in commercio macchine quad-core, che dispongono di quattro core fisici più quattro core virtuali per un totale di otto core logici[1]. Queste macchine possono gestire in modo efficace un numero di thread pari a otto.

Una GPU, al contrario, è progettata per effettuare calcoli paralleli ed è in grado di gestire un numero di thread molto più elevato, $\mathcal{O}(10000)$. Ad esempio, la macchina usata durante lo sviluppo del codice GPU per questo lavoro di tesi[2] può gestire efficacemente fino a 28672 thread contemporaneamente (sono disponibili 14 multiprocessori e ciascuno di essi gestisce fino a 2048 thread, vedi capitolo 6). Queste macchine inoltre implementano la strategia *SIMD*[3] (*single instruction, multiple data*) a livello di blocchi di thread[4].

Il primo problema da affrontare nello sviluppo di un algoritmo parallelo è quello di suddividere il lavoro in un numero di parti compatibile con il numero di thread gestibili dalla macchina utilizzata. Nel caso del tracking, abbiamo in ingresso un numero elevato di eventi e possiamo scegliere tra le seguenti opzioni:

- analizzare più eventi alla volta, evitando di suddividere l'analisi di ciascun evento in più parti. Possiamo chiamare questa strategia "parallelizzazione per eventi" (figura 4.2.1);
- analizzare un evento alla volta in sequenza, suddividendo l'algoritmo in più parti indipendenti e parallelizzando su queste ultime (vedi figura 4.2.2);
- mescolare i metodi sopra elencati, come schematizzato in figura 4.2.3.

Per lo sviluppo del codice GPU riguardante questo lavoro di tesi è stato adottato il terzo approccio sopra elencato, al fine di massimizzare il numero di thread in esecuzione. È stato

[1] la tecnologia che permette di emulare un core virtuale per ogni core fisico è nota come *hyperthreading*

[2] NVIDIA GTX Titan

[3] NVIDIA preferisce classificare l'architettura delle proprie GPU con il termine *SIMT*, ossia *single instruction multiple threads*, cioè un approccio a metà strada tra *SMT* (*simultaeous multithreading*) e *SIMD* (*single instruction multiple data*). Confrontare [69].

[4] nelle schede NVIDIA un blocco SIMD è detto *warp* e gestisce 32 thread.

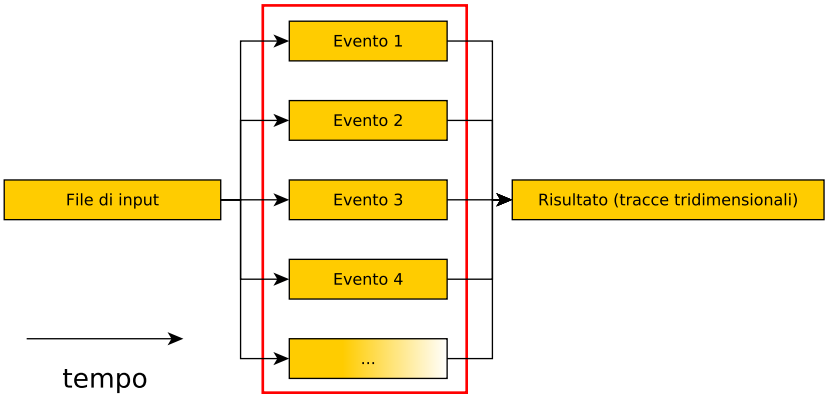

Figura 4.2.1: Parallelizzazione "per evento": ad ogni evento viene assegnato un thread indipendente.

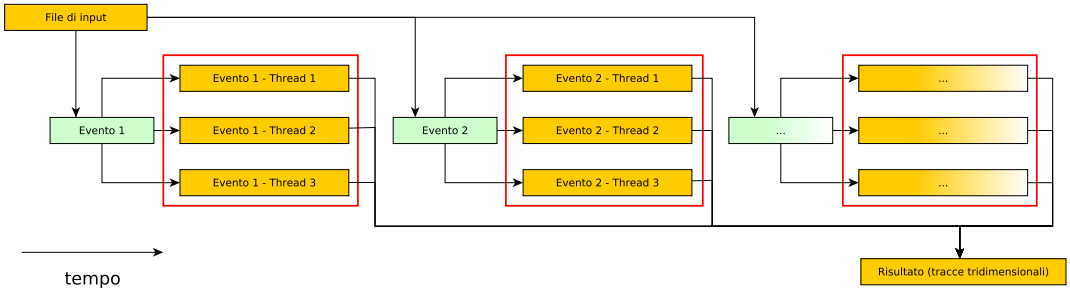

Figura 4.2.2: Parallelizzazione "per algoritmo": viene analizzato un evento per volta, ma l'esecuzione dell'algoritmo viene suddivisa in più thread. In rosso sono evidenziati i thread che vengono eseguiti contemporaneamente.

possibile caricare circa un migliaio di eventi nella memoria della GPU e gli algoritmi utilizzati sono stati opportunamente parallelizzati. La tipica parallelizzazione utilizzata è "per evento e per traccia". Dato che ogni evento contiene circa un centinaio di tracce, in questo modo si generano facilmente 10^5 thread indipendenti. Maggiori dettagli saranno dati nel capitolo 7.

Un secondo aspetto da affrontare nello sviluppo di codice parallelo e per GPU in particolare riguarda le strutture dati: è opportuno organizzare i dati in modo tale che thread appartenenti allo stesso *warp* accedano ad indirizzi contigui in memoria globale, per realizzare quella che viene chiamata "coalescenza nell'accesso alla memoria globale" (vedi capitolo 6).

Infine un ulteriore problema è che dovendo analizzare molti eventi contemporaneamente, lo spazio occupato in memoria globale è grande e impone un limite nel numero di eventi gestibili contemporaneamente. L'ottimizzazione dell'uso delle risorse di memoria diventa pertanto fondamentale per ottenere buone prestazioni con sistemi basati su GPU.

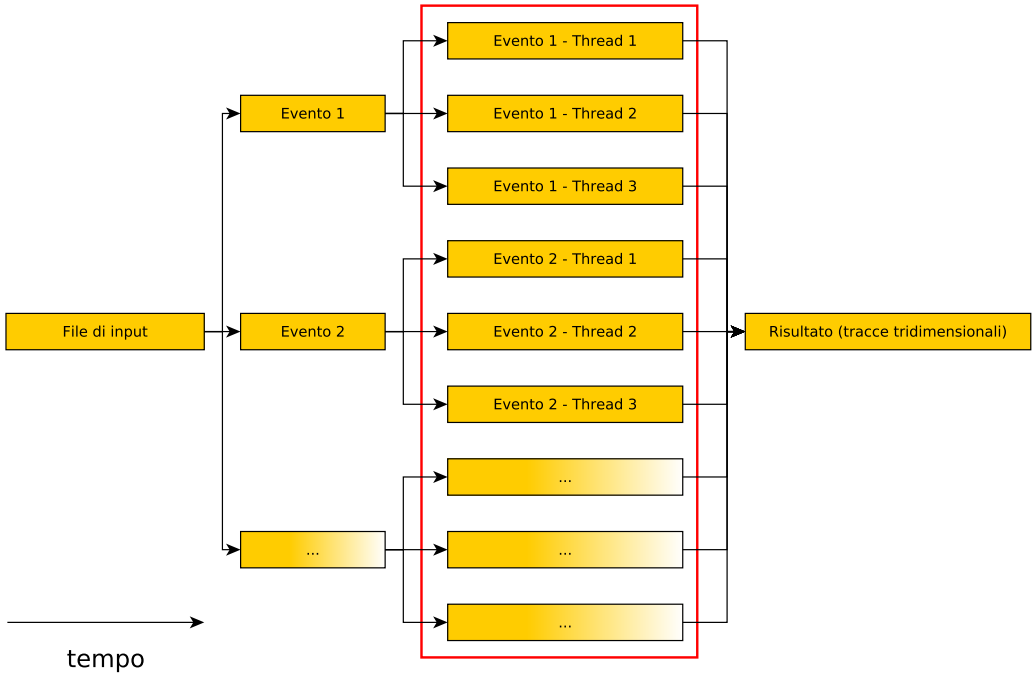

Figura 4.2.3: Parallelizzazione "per evento" e "per algoritmo": l'analisi di ogni evento viene suddivisa in ulteriori thread. In rosso sono evidenziati i thread che sono eseguiti parallelamente (tutti).

4.3 Tracking in LHCb

L'esperimento LHCb si occupa principalmente dello studio dei decadimenti di adroni contenenti il quark b e c. La maggior parte di questi adroni pesanti viene prodotta *in avanti* (cioè a piccoli angoli rispetto all'asse del fascio, vedi figura 2.1.3), pertanto i rivelatori di LHCb sono disposti in questa regione. Per misurare gli eventi di interesse il sistema di tracking di LHCb deve essere in grado di determinare con ottima precisione i vertici di decadimento e l'impulso delle particelle.

Le tracce più importanti per LHCb sono i prodotti di decadimento dei mesoni B (*b-daughters*): la distanza tra il vertice primario, cioè il punto in cui vengono prodotti, e il vertice secondario, punto in cui decadono, è dell'ordine della decina di millimetri. La distanza tra vertice primario e secondario è legata al parametro di impatto; quest'ultima grandezza costituisce il criterio principale con cui il trigger seleziona gli eventi: si ricercano tracce con grande parametro d'impatto e grande momento trasverso.

Il parametro d'impatto è definito come la distanza tra la retta su cui giace la traccia "figlia"[5] e il vertice primario in cui è prodotta la traccia "madre". Equivalentemente, il parametro

[5]per traccia figlia o daughter si intende una traccia prodotta dal decadimento di un'altra particella detta madre (mother).

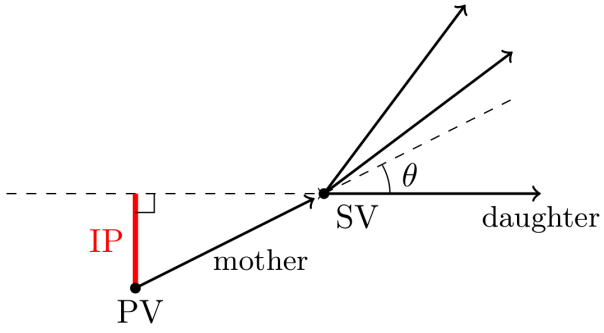

Figura 4.3.1: Definizione del parametro d'impatto. La traccia daughter e il vertice primario PV giacciono sul piano del disegno.

d'impatto è pari al prodotto della distanza tra vertice primario (PV) e secondario (SV) e il seno dell'angolo θ tra la retta sulla quale giace la traccia daughter e la retta sulla quale giace la traccia madre, come mostrato in figura 4.3.1, ovvero:

$$IP = |\vec{x}_{PV} - \vec{x}_{SV}| \, sin\,\theta,$$

dove $\vec{x}_{PV}$ e $\vec{x}_{SV}$ sono le posizioni del vertice primario e secondario.

La ricerca delle tracce è complicata dal fatto che assieme ai mesoni B viene prodotta una cospicua quantità di adroni contenenti quark leggeri (K, π, ...). Rispetto alle tracce che costituiscono il fondo (*background*), le b-daughters sono caratterizzate da un momento trasverso di circa 1 GeV più grande, come mostrato in figura 4.3.2.

Le prestazioni richieste dagli algoritmi di tracking per LHCb sono di natura diversa a seconda di due contesti, quello *online* e *offline*. La configurazione online è quella usata per il trigger, mentre la fase offline è quella in cui si analizzano dati già sottoposti al trigger e memorizzati su disco. Nel caso del trigger online si richiede che l'esecuzione dell'algoritmo rispetti stringenti vincoli temporali (deve cioè produrre un risultato in tempi molto rapidi). Si ricordi che l'HLT deve essere in grado di processare eventi e prendere la decisione di trigger ad una frequenza pari a circa 1.1 MHz, che caratterizza l'output del trigger L0. La farm attuale di HLT dispone di 29000 core di CPU [8], per cui ciascun core dispone di ≈ 30 millisecondi per analizzare un evento. Il numero degli eventi processati al secondo è il *throughput*, mentre per *latenza* si intende il tempo massimo a disposizione per processare un singolo evento prima che le informazioni vengano cancellate[6]. Nel caso del tracking online si ammette una certa inefficienza nella ricerca delle tracce purché vengano rispettati i limiti temporali richiesti dal trigger. Nel caso del tracking

[6]ad esempio, una CPU con 10 core può processare 10 eventi simultaneamente; se l'analisi termina dopo un millisecondo abbiamo un throughput di 10 eventi/millisecondo. Tuttavia, l'analisi un singolo evento ha una latenza di un millisecondo. In questo caso e in generale, in presenza di processi paralleli, il throughput non corrisponde all'inverso della latenza. Nota: in questo esempio si è trascurato il tempo di trasferimento dei dati.

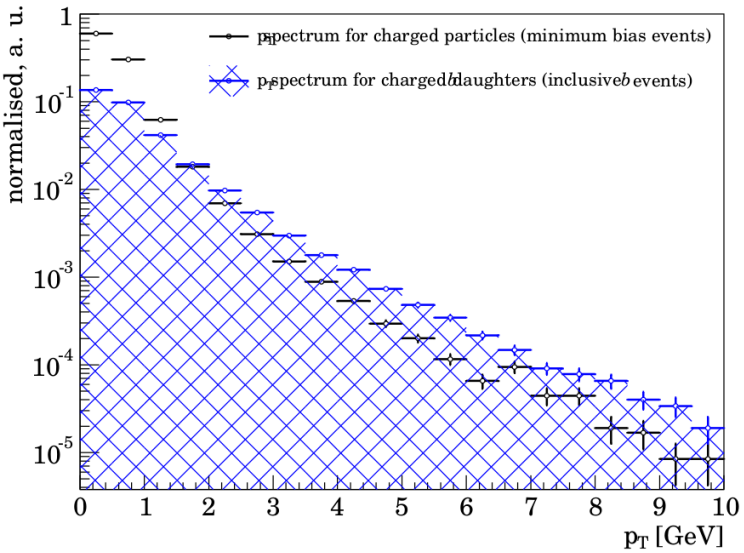

Figura 4.3.2: A sinistra: Distribuzione del modulo del momento delle tracce di un campione minimum bias rispetto ad un campione di b-daughters. Per campione *minimum bias* si intende un campione di dati registrato con selezioni minimali, in modo da non favorire un particolare canale.

offline, invece, non ci sono particolari restrizioni temporali, per cui si richiede che l'efficienza nella ricerca delle tracce sia la più alta possibile.

Nelle successive sezioni sarà introdotta la classificazione delle tracce e saranno descritte delle principali strategie di tracking di LHCb, nell'ultima sezione si farà cenno alle prestazioni nella misura dell'impulso.

4.3.1 Tipi di tracce

In base agli hit rilasciati da una traccia nei vari sub-detector che compongono il sistema di tracking (VELO, TT, T1, T2, T3), si possono distinguere cinque categorie di tracce. Facendo riferimento alla figura 4.3.3, abbiamo i seguenti casi:

- **Tracce lunghe**: tracce misurate in tutto il sistema di tracking, composto da VELO, dalle stazioni TT e i tracker T1, T2, e T3. Questo tipo di tracce è quello più significativo perché attraversano completamente la zona immersa nel campo magnetico e pertanto è possibile determinare il momento con la massima precisione;

- **Tracce VELO**: tracce ricostruite nel detector VELO. Le tracce VELO sono usate sia per la ricostruzione dei vertici di decadimento, sia come punti di partenza (*seed*) per ricostruire tracce lunghe;

- **Tracce T**: tracce ricostruite nel tracker principale composto dalle stazioni T1, T2, T3;

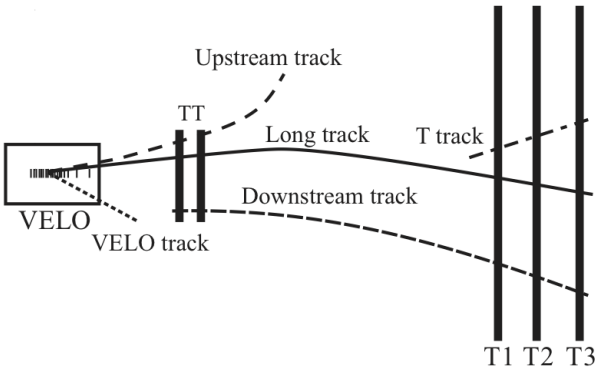

Figura 4.3.3: Schema per la classificazione delle tracce nell'esperimento LHCb [12].

- **Tracce upstream**: tracce misurate solamente nel velo e nel tracker TT. Queste tracce sono tipicamente tracce con basso momento ed escono dall'accettanza dopo aver attraversato il campo generato dal magnete;

- **Tracce downstream**: tracce misurate solamente nel tracker TT e nelle stazioni T1, T2, T3. Queste tracce possono essere prodotti di decadimenti di particelle neutre non rilevate nel VELO e con lunghezze di decadimento dell'ordine del metro, come il mesone K_S, che ha una vita media relativamente lunga, dell'ordine di $10^{-10}s$.

4.3.2 Tracking nel VELO

Il tracking del VELO è implementato per mezzo dell'algoritmo *FastVelo* ed è parte del trigger HLT1 e HLT2. L'algoritmo sarà discusso in dettaglio nel capitolo 5.

Il campo magnetico all'interno del VELO è piccolo (0.05 T) e può essere trascurato (vedi figura 4.3.4), pertanto le tracce all'interno del VELO possono essere considerate rettilinee, e questo permette di semplificare l'analisi. Il pattern recognition del VELO è però complicato dalla geometria $r - \phi$ del sensore.

4.3.3 Tracking nelle stazioni T

Il tracking nelle stazioni T consiste nel ricostruire le tracce considerando solamente gli hit rilasciati nei rivelatori T1, T2, e T3. Il detector T è posto nel campo residuo del magnete (vedi figura 4.3.4), pertanto l'algoritmo di pattern recognition deve tenere in conto la curvatura della traccia. L'algoritmo di tracking usato dal 2011 in poi per implementare il tracking si chiama *PatSeeding*, è documentato in [12, 13], ed è utilizzato sia nella componente HLT2 del trigger, sia nella ricostruzione offline.

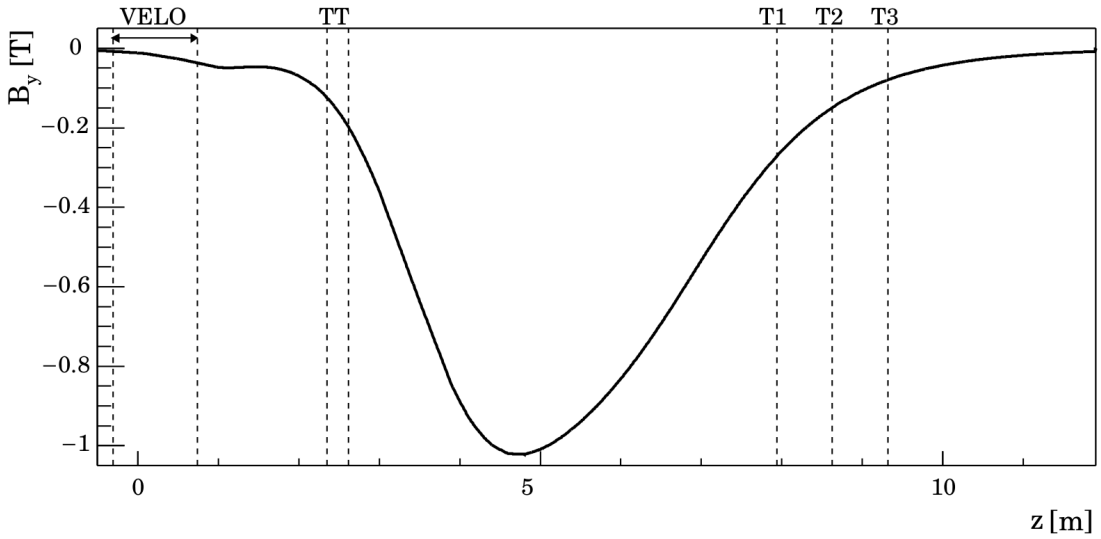

Figura 4.3.4: Campo generato dal magnete. All'interno del VELO è inferiore (in modulo) a 0.05 T e può essere trascurato.

4.3.4 Forward Tracking

Il tracking forward è uno dei due metodi per ricostruire le tracce lunghe. È chiamato così perché l'algoritmo di pattern recognition parte da una traccia del VELO e la estende "in avanti" verso il tracker principale T. Il tracking forward è utilizzato sia nella componente HLT1 che HLT2 del trigger; l'algoritmo corrispondente è chiamato *PatForward* (vedere schema in figura 3.1.3).

L'algoritmo di forward tracking [14, 15] utilizza una trasformata di Hough: in pratica, questo metodo consiste nel trasformare lo spazio degli osservabili, costituito dalle posizioni degli hit, nello spazio dei parametri, detto *spazio di Hough*. La trasformazione è scelta in modo tale che la distanza tra gli hit prodotti da una particella risulti minima se proiettata nello spazio di Hough.

L'idea alla base del forward tracking è che, se si trascurano i fenomeni di scattering e di perdita di energia, la traiettoria di una particella carica in un campo magnetico è completamente determinata se sono noti il campo magnetico e le condizioni iniziali. È sufficiente conoscere la posizione e il momento in un istante qualsiasi, oppure, alternativamente, la direzione e posizione a monte del magnete al tempo t_1 e una seconda posizione a valle del magnete al tempo t_2. Con questi dati è possibile determinare la posizione e la velocità della particella in qualsiasi punto lungo la traiettoria.

Sfruttando questo fatto, data una traccia VELO e la posizione di un hit in una stazione T (T1, T2 o T3), l'algoritmo calcola la coordinata x dell'intersezione della traiettoria di una particella con un piano a $z = 8250$ mm (piano di Hough). Se l'hit nella stazione T e la traccia VELO sono stati generati dalla stessa particella, la coordinata x calcolata sarà prossima alla

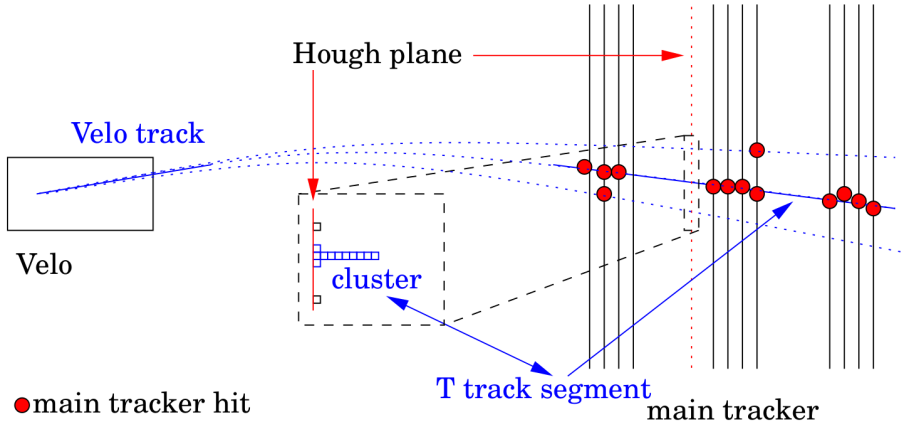

Figura 4.3.5: Schema che illustra la strategia del forward tracking. L'istogramma dei valori x calcolati presenta un picco in corrispondenza della traccia ricercata [12].

coordinata x "vera", altrimenti i valori x tenderanno a distribuirsi casualmente. Nell'istogramma dei valori x calcolati sarà presente un picco in corrispondenza della traccia voluta, come schematizzato in figura 4.3.5. È sufficiente limitarsi a considerare solo la coordinata x perché il campo magnetico curva le particelle principalmente nel piano $x - z$, mentre si può trascurare in prima approssimazione la deflessione nel piano $y - z$.

4.3.5 Track matching

Il track matching [17, 18] è un altro metodo per ricostruire le tracce lunghe. Diversamente dal forward tracking, sono usate le tracce T prodotte dall'algoritmo PatSeeding. Poiché le tracce prodotte da PatSeeding contengono una stima del momento, è possibile determinare la traiettoria attraverso il campo magnetico integrando numericamente l'equazione del moto. La traiettoria viene calcolata fino alla finestra di uscita del VELO, che si trova circa a $z \simeq 870\,mm$. A questo punto è effettuato il "matching" con le tracce del VELO, ossia per ogni traccia T viene cercata la traccia VELO più compatibile; la ricerca prosegue ricercando hit compatibili nel detector TT. L'algoritmo è usato nella componente HLT2 del trigger ed è chiamato *PatMatching.*

4.3.6 Tracking Downstream

Le tracce downstream sono ricostruite partendo da una traccia T per definire una regione (detta *region of interest* o *RoI*) in cui possono esistere potenziali hit TT appartenenti alla traccia; successivamente viene applicata una trasformata di Hough sugli hit TT candidati al fine di estendere la traccia T di partenza. L'idea è simile a quella del forward tracking, la differenza consiste nel fatto che l'algoritmo parte da una traccia dalla parte opposta rispetto al

magnete. Il tracking downstream è importante per ricostruire tracce di particelle a vita media lunga, come K_S^0 e Λ che spesso decadono all'esterno del VELO.

Il tracking downstream è più complesso rispetto al forward tracking perché il detector TT è composto da soli quattro strati di silicio, e pertanto non c'è ridondanza sufficiente a confermare con certezza la presenza di una traccia. Inoltre, la densità di tracce in TT è superiore a quella del tracker principale perché il campo magnetico agisce come filtro per le particelle a basso momento, curvandole fuori dalla regione di accettanza.

4.3.7 Tracking Upstream

Le tracce upstream sono ricostruite a partire dalle tracce VELO e gli hit TT. In modo simile al tracking downstream, ogni traccia VELO definisce una regione di interesse nel detector; agli hit contenuti in questa regione è applicata una trasformata di Hough. La differenza rispetto al forward tracking e il tracking downstream è che in questo caso il campo magnetico è costituito dal campo magnetico parassita presente tra il VELO e il TT (confrontare figura 4.3.4). L'algoritmo è descritto in dettaglio in [16]; sono disponibili configurazioni ottimizzate per il trigger e la ricostruzione offline. Nel primo caso, l'informazione sul momento che si ottiene può essere usata per selezionare le tracce con alto momento trasverso p_T e parametro d'impatto, tuttavia nel trigger attuale il tracking upstream non è usato in HLT, ma è utilizzato nella ricostruzione offline per recuperare tracce con basso momento che escono dalla regione di accettanza dopo aver attraversato il campo magnetico [12].

4.3.8 Misura dell'impulso

L'impulso delle tracce è misurato con diverse strategie a seconda del contesto (trigger o ricostruzione offline). La misura più grossolana è ottenibile stimando la curvatura media della traccia nel sistema di tracking, che è inversamente proporzionale al momento della particella. Questo metodo non tiene conto delle inomogeneità del campo magnetico, della perdita di energia e dello scattering multiplo, cosicché l'accuratezza sulla misura del momento non è ottimale. Per le tracce T, ad esempio, l'incertezza sulla stima del momento tramite la curvatura $dp/p = p_{\text{misurato}}/p_{\text{vero}}$-1, è dell'ordine del 14% [12]. Il metodo più accurato per la stima dell'impulso si basa sul fit effettuato per mezzo del filtro di Kalman, che tiene conto delle variazioni locali del campo magnetico, della perdita di energie e dello scattering. Per le tracce T si ottiene una risoluzione dp/p di circa 8%, mentre se applicato a tracce lunghe la risoluzione è dell'ordine dello 0.6%. Un terzo metodo si basa sulla misura dei parametri delle tracce calcolati con l'algoritmo del forward tracking, assumendo che le tracce provengano dall'origine del sistema di riferimento. Con questo sistema si ottiene una risoluzione dp/p del 3% [12].

Per quanto riguarda le prestazioni in termini temporali, il filtro di Kalman è il più lento, ma permette la misura più precisa nel caso delle tracce lunghe. Il terzo metodo rappresenta un buon compromesso in termini temporali ed è usato in varie parti del codice di HLT.

4.4 Figure di merito di un algoritmo di tracking

Per analizzare le prestazioni di un algoritmo di tracking è necessario generare eventi simulati per mezzo di un apposito programma di simulazione, che deve essere in grado di generare le particelle prodotte dalle collisioni, i loro decadimenti, la loro successiva interazione con gli apparati di misura e la risposta dei detector (vedi capitolo 2). I programmi di simulazione implementano il cosiddetto metodo Monte Carlo (MC), che è basato sulla generazione di numeri casuali, e pertanto si parla di eventi Monte Carlo e particelle Monte Carlo.

Tramite il MC è possibile avere accesso a tutte le informazioni "vere" di ogni particella (impulso, posizione di origine, etc...); la misura delle figure di merito si basa sul confronto fra le tracce "vere" e quelle ricostruite dall'algoritmo di tracking. Le prestazioni di un algoritmo di tracking sono definite da un certo numero di parametri, in particolare *efficienza*, *ghost rate* e *clone rate*. Verranno ora introdotte alcune definizioni necessarie per definire i parametri relativi all'efficienza.

4.4.1 Traccia ricostruibile

Si definisce traccia ricostruibile una traccia che ha prodotto un numero sufficiente di hit nel rivelatore. Il criterio deve essere definito in modo opportuno a seconda del rivelatore che si considera:

- nel caso del VELO una traccia è considerata ricostruibile se ha lasciato almeno tre hit sui sensori R e tre hit sui sensori ϕ;
- nel caso del tracker principale (stazioni T1, T2, T3) una traccia è considerata ricostruibile se ha rilasciato almeno un hit x e un hit stereo in ciascuna stazione (vedi capitolo 2). Le tracce trovate nelle stazioni T1, T2, T3 sono chiamate alternativamente tracce T o tracce *seed* in quanto possono essere utilizzate come punto di partenza per ricostruire tracce più lunghe;
- una traccia è ricostruibile come traccia upstream se è ricostruibile sia nel VELO che nel rivelatore TT;
- una traccia è ricostruibile come traccia downstream se è ricostruibile sia nel rivelatore TT che nel tracker principale (stazioni T).

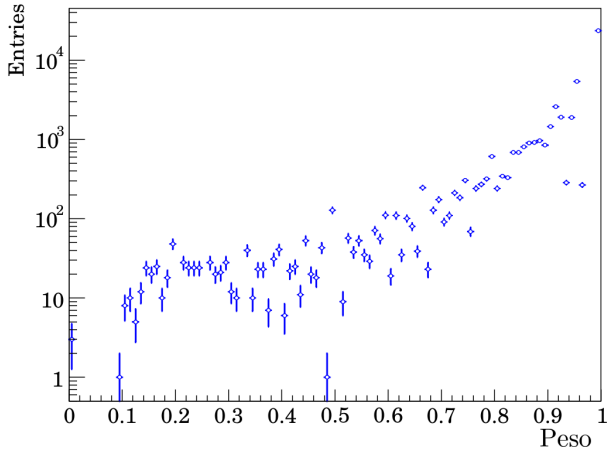

Figura 4.4.1: Massima frazione di hit di una traccia relativa ad particella specifica. Notare la scala logaritmica sull'asse delle ordinate [12].

4.4.2 Associazione alle tracce Monte Carlo

Per associare una traccia ad una "vera" simulata si sceglie di definire una traccia t **associata** ad una particella Monte Carlo v se almeno il 70% degli hit della traccia t sono stati prodotti dalla particella Monte Carlo v. Il valore esatto di soglia non è critico: le tracce associate contengono tipicamente tutti gli hit prodotti dalla particella. Se definiamo il *peso* w come la massima frazione di hit su una traccia t provenienti da una particella Monte Carlo, cioè

$$w(t) = \max_v \frac{N\text{hit prodotti dalla particella } v \text{ presenti in } t)}{N(\text{hit della traccia } t)}.$$

Il valore tipico per w è 1 (vedi grafico 4.4.1), cioè tipicamente tutti gli hit di una traccia provengono da un'unica particella Monte Carlo.

4.4.3 Efficienza di un algoritmo di tracking

Per efficienza ϵ di un algoritmo di tracking si intende il rapporto tra tracce ricostruite e tracce corrette ricostruibili:

$$\epsilon = \frac{N(\text{ricostruibili e ricostruite, che non siano elettroni})}{N(\text{ricostruibili, che non siano elettroni})}. \tag{4.4.1}$$

Dal computo sono esclusi gli elettroni perché perdono molta energia per Bremsstrahlung e pertanto le tracce corrispondenti sono più difficili da ricostruire.

Alcuni eventi sono caratterizzati da un'alta molteplicità (ossia da un gran numero di particelle prodotte); questo tipo di eventi è difficile da ricostruire ed è caratterizzato da una bassa efficienza. Per questo motivo, è significativo definire due metodi per calcolare l'efficienza: l'efficienza mediata sulle tracce e quella mediata sugli eventi. Dato un insieme di eventi, l'efficienza ϵ mediata sulle tracce si calcola considerando tutte le tracce di tutti gli eventi insieme secondo

l'espressione 4.4.1. L'altro modo consiste nel calcolare il parametro ϵ separatamente per ogni evento per poi calcolare il valore ϵ medio. L'efficienza mediata sulle tracce tende ad essere peggiore di quella mediata sugli eventi, perché gli eventi con maggiore molteplicità hanno un peso maggiore.

4.4.4 Tracce ghost

Una certa frazione delle tracce ricostruite può non essere associata ad alcuna particella Monte Carlo: può succedere infatti che una combinazione di hit prodotti da particelle diverse risultino sufficientemente allineati tali da essere riconosciuti come traccia nella fase di pattern recognition, ma non corrispondere ad alcuna particella MC.

La frazione di ghost (o *ghost rate*) è la frazione di tracce ghost rispetto a tutte le tracce prodotte dall'algoritmo di pattern recognition. Ad esempio l'algoritmo FastVelo che implementa il tracking del rivelatore VELO genera circa il 7% di ghost, facendo riferimento ad una simulazione che riproduce le condizioni del 2011 ($\nu = 2.5$) [19].

Gli eventi che hanno molteplicità più alta tendono a formare molte più tracce ghost, a causa del maggior combinatorio, quindi anche in questo caso si distinguono la frazione di ghost mediata per tracce e quella mediata per evento.

4.4.5 Tracce cloni

Si è in presenza di una traccia clone quando la stessa traccia fisica viene *associata* a più di una traccia ricostruita dall'algoritmo di tracking. In questo caso, viene considerata valida solo la traccia che contiene il maggiore numero di hit, perché contiene più informazioni delle altre, e le rimanenti vengono definite *cloni*. È opportuno che un algoritmo di tracking rimuova i cloni generati il prima possibile, anche per evitare di effettuare calcoli non necessari nelle successive fasi di analisi.

La definizione appena data è valida per tracce Monte Carlo, ma è possibile fornire una definizione alternativa di cloni valida anche nel caso di dati reali, per i quali non è disponibile l'informazione sulla traccia "vera". In questo caso possono essere definiti altri criteri:

- due tracce possono essere considerate cloni se il numero di hit in comune è superiore al 70% della traccia più corta;
- un altro criterio può essere quello di definire tracce cloni due tracce che hanno parametri compatibili entro un certo intervallo, indipendentemente dal numero di hit in comune.

L'intervallo tipico per la frazione di cloni generati da tutti algoritmi usati nel pattern recognition per il tracking di LHCb è dell'ordine dello 0.1% fino all'1% [50].

4.4.6 Altre definizioni

Per **purezza degli hit** (*hit purity*) si intende la percentuale di hit che fanno effettivamente parte della traccia Monte Carlo associata. Data una traccia t associata alla particella "vera" v si ha:

$$\text{purezza negli hit} = \frac{N(\text{hit prodotti dalla particella } v \text{ presenti in } t)}{N(\text{hit totali della traccia } t)}.$$

L'**efficienza negli hit** (*hit efficiency)* è definita come il rapporto tra tutti gli hit prodotti da una particella e il numero di hit della traccia. Data una traccia t associata alla particella "vera" v si definisce:

$$\text{efficienza negli hit} = \frac{N(\text{hit prodotti dalla particella } v \text{ presenti in } t)}{N(\text{hit totali prodotti dalla particella } v)}.$$

Per valutare l'efficienza degli algoritmi di tracking, le tracce sono classificate a seconda della particella di provenienza, come di seguito elencato:

- **B-daughters**: per b-daughters si intendono tracce che sono prodotti di decadimento di adroni contenenti il quark b;
- **Tracce "buone"**: per tracce "buone" si intendono tracce b-daughters i cui prodotti di decadimento siano tutti ricostruibili. Queste chiaramente sono tracce molto importanti, perché sono le tracce più utili alla fine di effettuare misure di precisione relative alla fisica dei mesoni B;
- **K^0_S/Λ daughters**: le tracce prodotte dal decadimento dei mesoni strani K^0_S e Λ sono particolarmente difficili da ricostruire, in quanto sono caratterizzate da vertici di decadimento molto lontani dal vertice primario nel quale sono prodotte le particelle madri K^0_S e Λ.

4.5 Caratteristiche tipiche degli eventi del VELO

Per la misura delle efficienze e del tempo di esecuzione del tracking, in questa tesi ci siamo concentrati su eventi Monte Calro $B_s \to \phi\phi$ generati con le condizioni del 2012 (ossia $\nu = 2.5$[7], $\sqrt{s} = 8$ TeV). Un altro campione studiato per la misura dei tempi del tracking è costituito da circa 8000 eventi di dati reali *no-bias*[8] con $\mu = 1.6$ raccolti durante il 2012.

Nel grafico 4.5.1 è riportata la distribuzione di cluster per sensore, che è legata all'occupanza: per *occupanza* dei sensori si intende il numero di elementi sensibili attivati rispetto al numero

[7]per ν si intende il numero medio di interazioni pp per evento, in realtà si è interessati al numero medio di interazioni *visibili* per evento, ossia misurabili dal detector, indicato solitamente con μ; si preferisce tuttavia classificare eventi Monte Carlo per mezzo di ν e non μ.

[8]per no-bias si intende un campione di dati scelto in modo da non favorire nessun canale specifico.

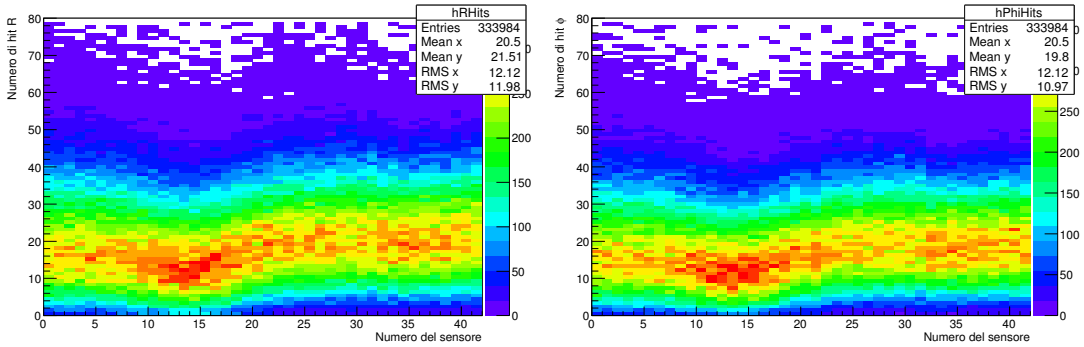

Figura 4.5.1: Distribuzione dei cluster per sensore. A sinistra: sensori R. A destra: sensori ϕ. Numerando i sensori da 1 a 42, l'origine degli assi si trova tra il sensore 12 e 13, e corrisponde approssimativamente alla centro zona d'interazione. Campione di dati *no-bias* con $\mu = 1.6$ del 2012.

totale di elementi nel sensore. Nel caso del VELO l'elemento sensibile corrisponde alla strip; un cluster è un insieme costituito da una a quattro strip contigue. Per il campione no-bias del 2012, il numero medio di cluster per sensore R è 21.5 mentre quello per sensore ϕ è 19.8.

Un parametro importante è il numero di tracce per evento: questo dato è in pratica direttamente proporzionale a ν (numero di interazioni per bunch crossing, usato per classificare eventi Monte Carlo) oppure a μ (numero di interazioni visibili per bunch crossing). Nel campione Monte Carlo il numero di tracce medio per evento trovate da FastVelo in modalità HLT1 è circa 80, mentre il campione di dati reali no-bias il numero di tracce medio è circa 100, confrontare i grafici di figura 4.5.3. Un altro parametro rilevante è il numero di hit per traccia, la cui distribuzione è riportata in figura 4.5.4. Si può notare che le tracce più comuni sono tracce corte con 4 hit R o 4 hit ϕ. Esistono anche tracce che contengono tre hit R, anche se in HLT1 non sono ricercate triplette nella proiezione RZ (vedi capitolo 5): questo è dovuto al fatto che un hit R presente nella quadrupletta iniziale può essere successivamente rimosso perché fornisce un contributo χ^2 troppo elevato durante la fase di track fitting.

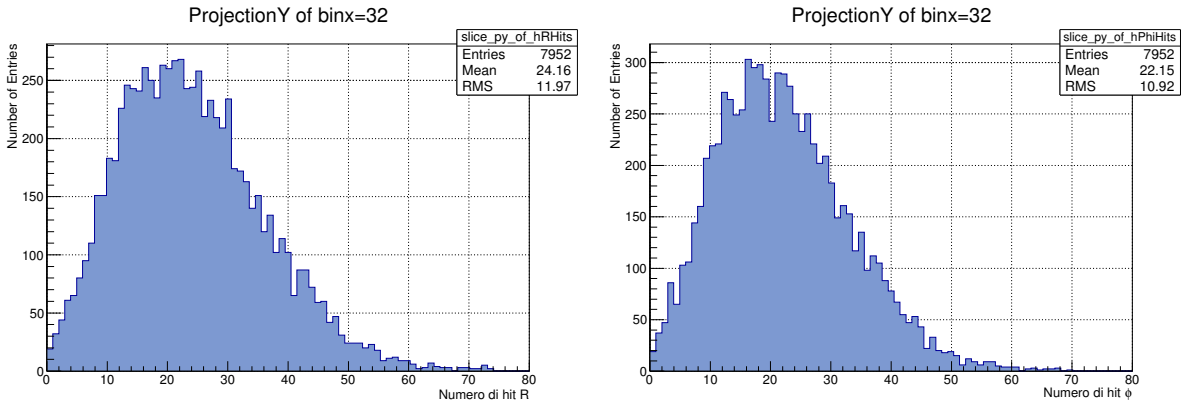

Figura 4.5.2: A sinistra: numero di cluster R nel sensore 32. A destra: numero di hit ϕ nel sensore 32. Campione usato: 7952 eventi *no-bias* con $\mu = 1.6$ del 2012.

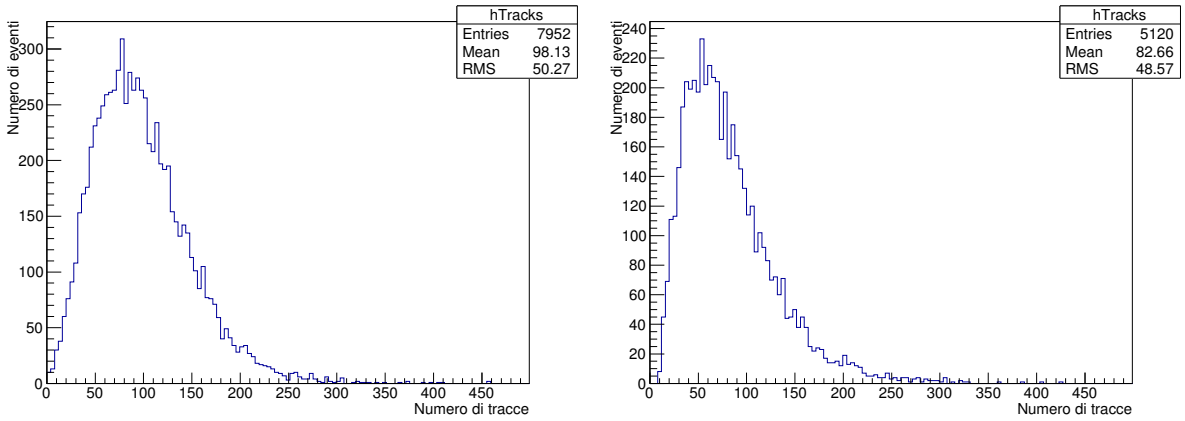

Figura 4.5.3: Numero di tracce per evento nel VELO ricostruito in HLT1. A sinistra: il numero di tracce per evento (campione di dati *no-bias* con $\mu = 1.6$ del 2012). A destra: numero di tracce per evento, campione di dati Monte Carlo di 5120 eventi $B_s \to \phi\phi$ con condizioni del 2012 ($\nu = 2.5$).

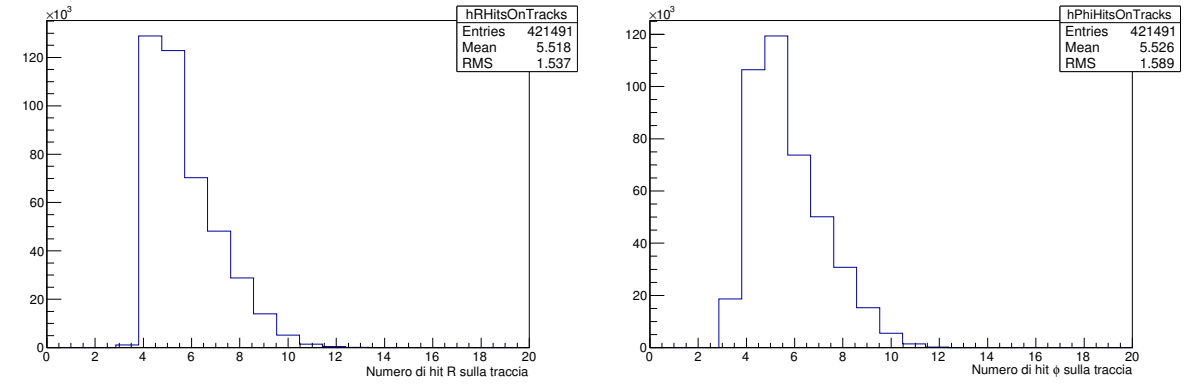

Figura 4.5.4: A sinistra: numero di hit R per traccia. A destra: numero di hit ϕ per traccia. Campione usato: 5120 eventi $B_s \to \phi\phi$ con condizioni del 2012 ($\nu = 2.5$).

5

FastVelo

L'algoritmo *FastVelo* [19] implementa il tracking nel VELO ed è stato con successo nei run del 2011 e del 2012. L'algoritmo rappresenta un'evoluzione di un precedente versione chiamata *PatVelo* usato fino al 2010. Rispetto alla versione precedente, FastVelo fornisce prestazioni in termini di efficienza e timing migliori, e si adatta a condizioni di più alta luminosità rispetto a quelle programmate per il 2011-2012. Infatti, inizialmente era previsto un numero medio di interazioni visibili p-p per bunch crossing pari a $\mu = 0.4$, mentre nel 2011-2012 si è raggiunto $\mu = 2.5$ (valore corrispondente ad una luminosità di $4 \times 10^{32}\text{cm}^{-2}\text{s}^{-1}$). L'algoritmo è implementato in C++ ed è sviluppato all'interno del framework Gaudi.

Nelle prossime sezioni sarà introdotta la struttura dell'algoritmo e saranno discusse in dettaglio le varie componenti. La discussione delle prestazioni dell'algoritmo è rimandata invece al capitolo 9, dove queste prestazioni saranno confrontate con quelle ottenute con la versione per GPU (*FastVeloGPU*).

5.1 L'algoritmo *Fastvelo*

L'input dell'algoritmo FastVelo sono le informazioni degli *hit* provenienti dal rivelatore, mentre l'output consiste in tracce tridimensionali nel formato *LHCb::Track* definito in Gaudi.

Le tracce sono classificate di due tipi: tracce *forward* e *backward*. Le prime sono tracce che si allontanano dalla zona di interazione propagandosi a valle, ovvero verso il magnete e il resto dei rivelatori. Le tracce backward invece si allontanano dal punto di interazione dirigendosi verso i sensori di pileup (veto stations), vedi figura 2.2.1.

L'algoritmo FastVelo è diviso in due parti principali. La *prima parte,* chiamata tracking RZ, coinvolge solamente i sensori di tipo R e ricerca tracce nella proiezione RZ, dove R è la distanza della strip dall'asse del fascio. Ogni sensore è posizionato in un piano perpendicolare al fascio[1], per cui ogni strip colpita corrisponde ad una specifica coordinata R corrispondente al raggio della strip e alla coordinata Z corrispondente al sensore. I risultati della prima parte dell'algoritmo sono un insieme di tracce nella proiezione RZ, che sono utilizzate successivamente come punto di partenza (*seed*) per costruire tracce nello spazio tridimensionale.

La *seconda parte* di FastVelo ricostruisce tracce spaziali, ossia le tracce tridimensionali che costituiscono l'output finale. Una traccia spaziale si ottiene combinando le informazioni date dai sensori R con le informazioni degli hit sui sensori ϕ. Le tracce spaziali vengono parametrizzate da quattro parametri reali (x_0, y_0, t_x, t_y), secondo l'equazione

$$\begin{cases} x(z) = x_0 + z\, t_x; \\ y(z) = y_0 + z\, t_y. \end{cases} \tag{5.1.1}$$

dove x_0 e y_0 sono i valori dell'intercetta a $z = 0$, mentre t_x e t_y sono i coefficienti angolari. Il fit dei parametri della traccia viene effettuato minimizzando un χ^2 basato sulla distanza tra la retta interpolante e le rette che parametrizzano le strip. Le strip ϕ sono parametrizzate tramite tre parametri a, b, c che descrivono la retta bidimensionale $ax + by + c = 0$ sulla quale giace la strip (vedi figura 5.1.1).

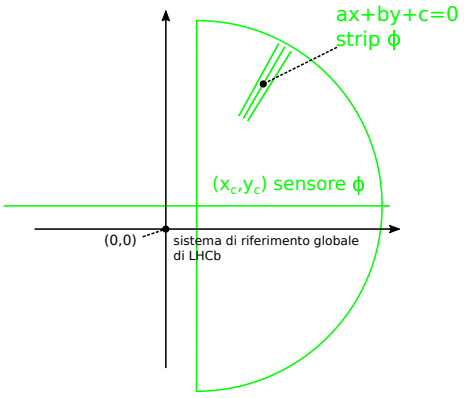

Figura 5.1.1: Parametrizzazione delle strip ϕ.

[1] il sensore non è perfettamente perpendicolare all'asse Z, né è centrato perfettamente nel piano XY, né il fascio è perfettamente centrato a $x = 0$, $y = 0$. Le informazioni relative al disallineamento sono misurate nelle fasi di calibrazione e registrate in un database; opportune correzioni tengono conto di questi effetti.

I parametri a, b, c sono calcolati a partire dai dati presenti nel database, che contengono le informazioni sulla geometria del sensore e le informazioni sul disallineamento. Per parametrizzare le strip R, che hanno la forma di un arco di cerchio, si approssimano con rette tangenti al punto dove si presume passi la retta interpolante, come mostrato in figura 5.1.2.

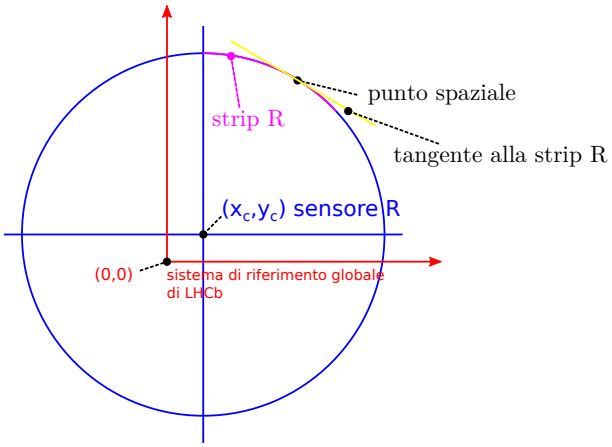

Figura 5.1.2: Approssimazione della strip R con una tangente, utile per il track fitting.

Il χ^2 della retta interpolante è calcolato come $\chi^2 = \sum_i \frac{(a_i x + b_i y + c_i)^2}{\sigma_i^2}$, dove l'indice i corre lungo la strip i-esima associata alla traccia, e σ_i è l'errore associato alla strip i-esima, proporzionale alla dimensione del cluster e al pitch come accennato nel paragrafo 2.2.1.1.

L'ultima parte dell'algoritmo si occupa di rimuovere i cloni: una parte si occupa di rimuovere i cloni provenienti dalla stessa traccia RZ (nel codice questo compito è svolto dalla funzione *mergeClones*, che è eseguita all'interno della funzione che si occupa del tracking spaziale *makeSpaceTracks*), successivamente la funzione *mergeSpaceClones* si preoccupa di rimuovere cloni provenienti da tracce RZ diverse. Infine si convertono le tracce spaziali in un formato compatibile con il framework Gaudi (*LHCb::Track).*

Un diagramma di flusso dell'algoritmo è presentato nella figura 5.1.3, mentre la descrizione dettagliata degli funzioni di FastVelo è riportata nei paragrafi successivi.

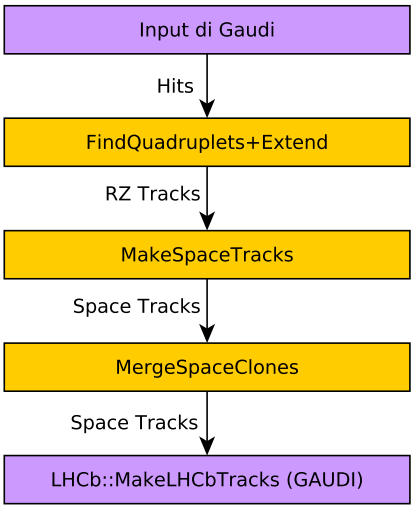

Figura 5.1.3: Struttura dell'algoritmo FastVelo originale in configurazione HLT1.

5.1.1 La funzione findQuadruplets

La funzione *findQuadruplets* realizza la fase di tracking RZ nella componente HLT1. Essa ricerca sequenze di quattro hit allineati in cinque sensori R contigui nello stesso settore (ricordiamo che i sensori R sono divisi in quattro settori da 45° che coprono un semicerchio, coppie di sensori R affiancati sono chiamate *stazioni* e coprono un cerchio). La ricerca viene effettuata settore per settore in quanto quasi tutte le tracce ($\approx 99.9\%$) rimangono nello stesso settore quando il VELO è chiuso [19]. L'algoritmo ammette che un hit in un sensore intermedio sia mancante, questo perché i sensori non sono perfettamente efficienti e può succedere che una particella attraversi un sensore senza lasciare hit. Considerando *quintuplette* di sensori R (vedi figura 5.1.4), possiamo definire i casi riportati nella tabella 5.1.

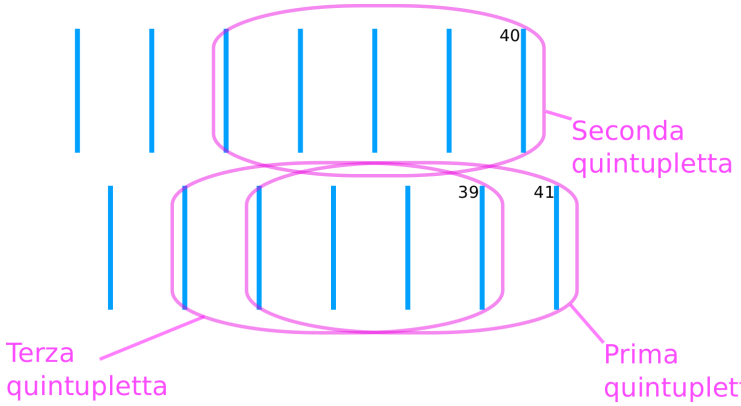

Figura 5.1.4: Schema Sequenza di ricerca delle quadruplette *forward*: sono mostrate le prime tre quintuplette considerate.

Caso	Configurazione
0	XXXX
1	X.XXX
2	XX.XX
3	XXX.X

Tabella 5.1: Possibili quadruplette in FastVelo. Una X corrisponde alla presenza di un hit e un punto ad uno mancante

Il caso in cui il primo sensore sia privo di hit non è considerato, poiché corrisponde a considerare la quintupletta successiva.

L'algoritmo è progettato in modo da evitare di generare cloni (cioè di trovare la stessa traccia più volte, vedi sezione 4.4.5). Per questa ragione sono poste due condizioni: il primo hit non deve essere già stato usato in una precedente quadrupletta, mentre l'ultimo hit deve essere inutilizzato se si sta cercando una quadrupletta con sensori mancanti (caso 1, 2, 3 nella tabella 5.1). In questo modo la condizione è più stringente nel caso in cui la quadrupletta abbia un hit mancante. La procedura di marcare gli hit come usati (detta *tagging*) è effettuata dopo aver trovato una quadrupletta e dopo averla estesa (vedi paragrafo successivo). Come vedremo nel capitolo successivo, la strategia di marcare gli hit usati non è ottimale nel caso di algoritmi paralleli e pertanto saranno implementati algoritmi che evitano di *taggare* gli hit usati.

L'algoritmo dà priorità alle tracce lunghe (definite nella sezione 4.3.1), in quanto sono le più importanti per le analisi in LHCb. A tal fine vengono considerate per prime le quintuplette di sensori più vicine al magnete e successivamente quelle più vicine alla zona di interazione, perché le tracce che raggiungono i sensori vicino al magnete sono quelle che con più probabilità possono formare tracce lunghe. Così facendo si evita che queste tracce vengano escluse a causa del *tagging*. In altre parole gli hit delle quadruplette sono ricercati nella direzione opposta a quella della propagazione delle particelle.

Avendo numerato i sensori R da 0 a 41 ordinati per z crescente (il sensore 41 è quello più vicino al magnete) la ricerca parte dal sensore 41 e la prima quintupletta considerata è costituita dai sensori 41-39-37-35-33 (in questo caso si considera solo il lato destro, si veda la figura 2.2.1). Nel primo caso (vedi tabella 5.1) saranno ricercate quadruplette nei sensori 41-39-37-35, successivamente nei sensori 41-37-35-33 (manca l'hit nel sensore 39), poi saranno considerati i sensori 41-39-35-33, e così via. La ricerca prosegue partendo da un sensore diverso: il successivo è il 40 (sensore sinistro, situato a $x > 0$), poi il 39 (destro), etc.

La procedura si interrompe quando la coordinata z del sensore R è più piccola di $zMin = ZVertexMin$[2]$+200\,\text{mm}$. Il processo si ripete analogamente per le tracce backward.

[2]confrontare tabella 5.2

Uno pseudocodice semplificato dell'algoritmo è il seguente:

```
per ogni zona (da 1 a 4)
  per ogni iCase (da 0 a 3)
    per ogni hit h1 del sensore 1
      per ogni hit h4 del sensore 4
        cerca l'hit h2 nel sensore 2 più vicino alla posizione attesa data dalla
                                                    retta passante per h1 e h4;
        se l'hit non è stato trovato, passa al prossimo hit h4.
        cerca l'hit h3 nel sensore 3 più vicino alla posizione attesa data dalla
                                                    retta passante per h1 e h4;
        se è stato trovato un hit nel sensore 3, salva la quadrupletta (h1, h2, h3, h4).
```

Il criterio per trovare l'hit più compatibile è il seguente: si traccia la retta passante per h1 ed h4, cioè i punti (z_1, r_{h1}) e (z_4, r_{h4}) nella proiezione RZ, dove z_1 e z_4 sono le coordinate corrispondenti al sensore 1 e 4 mentre r_{h1} e r_{h4} sono le coordinate degli hit corrispondenti. La posizione predetta ai sensori intermedi ("sensore 2" o "sensore 3" nello pseudocodice) dell'hit è semplicemente $r_{\text{pred}} = r_{h1} + (z - z_1)\frac{r_{h4}-r_{h1}}{z_4-z_1}$. Viene scelto l'hit con la coordinata r più vicina alla posizione aspettata all'interno dell'intervallo di tolleranza $r_{\text{pred}} \pm r_{\text{tol4}}$, dove $r_{\text{tol4}} = rMatchTol4 \times rPitch$, dove $rMatchTol4$ è un parametro configurabile (vedi tabella5.2) e $rPitch$ è lo spessore delle strip dei sensori R in corrispondenza della coordinata prevista r_{pred}.

Nel codice sono presenti alcune condizioni che permettono di eliminare tracce che hanno pendenze troppo elevate. Il parametro che controlla questo tipo di taglio è $MaxRSlope$.

5.1.2 Estensione delle quadruplette

Una volta trovata una quadrupletta valida, essa viene estesa, cioè sono ricercati ulteriori hit compatibili con la retta che interpola le quattro hit della quadrupletta ottenuta in precedenza. Il parametro che controlla la tolleranza con la quale vengono aggiunti nuovi hit durante l'estensione è il parametro $rExtraTol$ (vedi 5.2). L'estensione avviene solamente verso un lato, cioè verso la zona d'interazione, perché si suppone che gli hit presenti nel lato opposto siano già usati nelle ricerche precedenti.

Un caso speciale riguarda le tracce che si trovano nella regione di overlap dei sensori R. È una piccola regione nel piano (x, y) che è coperta sia dai sensori R destri che da quelli sinistri. Se una traccia passa attraverso questa zona, ci aspettiamo hit provenienti dalla stessa traccia in entrambi i lati del sensore. Per tener conto di questo fatto il codice tenta di estendere la traccia cercando corrispondenze di hit anche nel lato opposto del sensore.

Parametro	Valore Predefinito	Commento
ZVertexMin	-170.0 mm	Specifica il minimo valore per la coordinata z del vertice primario determinando l'intervallo di sensori R nei quali si ricercano tracce forward.
ZVertexMax	120.0 mm	Specifica il massimo valore per la coordinata z del vertice primario determinando l'intervallo di sensori R nei quali si ricercano tracce backward.
ZVertexMaxBack	1200.0 mm	Limite per la z del vertice primario delle tracce backward per le tracce BeamGas.
MaxRSlope	0.45	Imposta il coefficiente angolare massimo per tracce RZ.
rMatchTol4	1.0	Imposta la tolleranza usata per la ricerca delle quadruplette.
rExtraTol	4.0	Imposta la tolleranza usata per l'*estensione* delle quadruplette.
rOverlapTol	1.0	Imposta la tolleranza usata quando si ricercano quadruplette in overlap.
MaxMissed	1	L'estensione delle tracce (funzione extendTrack) si interrompe quando si incontrano più di *MaxMissed* sensori privi di hit allineati.
MinToTag	4	Minima lunghezza della traccia RZ affinché gli hit vengano marcati come usati (se impostato a 4 la condizione è sempre verificata nella funzione *findQuadruplets*).

Tabella 5.2: Parametri configurabili relativi alla funzione findQuadruplets

5.1.3 La funzione *makeSpaceTracks*

Data una traccia RZ, la funzione *makeSpaceTracks* associa i possibili hit ϕ ricostruendo la traccia tridimensionale. Il primo passo è quello di selezionare un opportuno intervallo di sensori ϕ nel quale cercare gli hit. La traccia RZ trovata in precedenza è usata per definire tale intervallo: si cercano hit ϕ nello stesso intervallo di *stazioni* occupato dalla traccia RZ, ammettendo la ricerca in una stazione aggiuntiva in entrambi i lati (a monte e a valle).

5.1.3.1 Selezione e calcolo dei punti spaziali

È necessario selezionare dai molti hit ϕ presenti in una stazione quelli effettivamente associabili alla traccia, in modo da semplificare la successiva fase che si occupa della ricerca di triplette. Poiché ogni traccia RZ è localizzata in un settore circolare di 45°, si ricercano hit ϕ che si occupano la stessa regione nel piano xy.

Un esempio di filtro "veloce" per la selezione di hit ϕ è richiedere che la proiezione centro della strip sulla bisettrice del settore circolare abbia un valore minimo, nel codice si è scelto il valore $(0.4 \times (rMin+rMax))$, dove $rMin = 8.17$ mm è raggio il minimo del sensore e $rMax = 42$ mm è quello massimo.

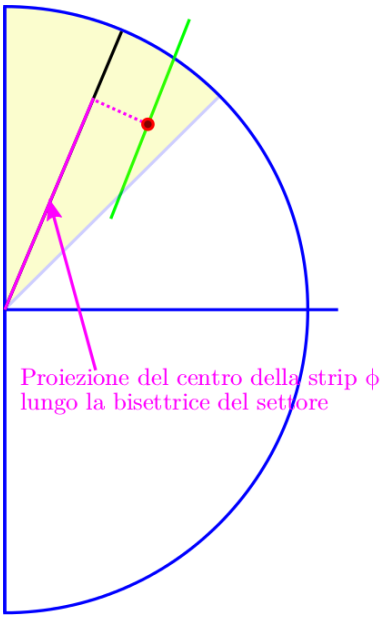

Figura 5.1.5: Uno dei di selezione degli punti spaziali.

La coordinata r al sensore ϕ è ottenuta dall'estrapolazione della traccia RZ alla z del sensore ϕ; a questo punto è possibile calcolare un punto spaziale (x, y, z) per ogni ϕ selezionato su cui si baserà la successiva fase di ricerca delle tracce. Il metodo usato per calcolare la coppia (x, y) consiste nel trovare l'intersezione tra una retta e una circonferenza, risolvibile tramite un'equazione di secondo grado, come schematizzato nella figura 5.1.6.

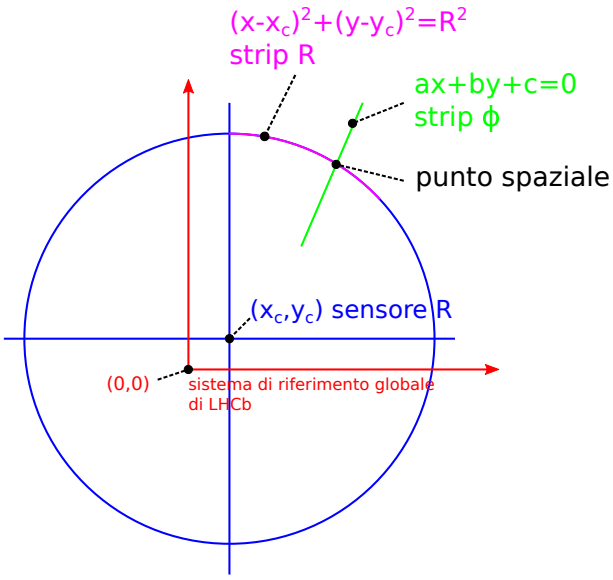

Figura 5.1.6: Calcolo delle coordinate spaziali (x, y)

I parametri (a, b, c) per ogni strip ϕ di ogni sensore ($= 2048 \times 48 = 98304$ strip totali) sono pre-calcolati dal programma all'inizio dell'algoritmo a partire dalle informazioni presenti nel database di Gaudi (è necessario calcolarli una volta sola, non vengono ricalcolati ad ogni evento). Conoscendo anche la posizione del centro del sensore, ricavata anche essa dal database, è possibile determinare l'intersezione punto/circonferenza. C'è da notare che il programma per determinare la posizione finale, a partire dalla coppia (x, y) così ottenuta, corregge la coordinata z dell'intersezione tenendo conto dell'inclinazione dei sensori.

5.1.3.2 Ricerca delle triplette

L'approccio usato per la ricerca delle triplette di hit ϕ è simile a quello delle quadruplette, ed anche in questo caso si fa uso del tagging (gli hit ϕ usati per creare una traccia spaziale vengono marcati come usati e non vengono più utilizzati).

Come accennato nel paragrafo precedente, a questo punto dell'algoritmo sono disponibili un insieme di punti spaziali (x, y, z) che corrispondono alle intersezioni dei cluster R e ϕ nella stessa stazione[3]. La ricerca delle triplette avviene nella direzione opposta a quella della propagazione della traccia come avviene per le quadruplette in RZ. Diversamente dall'algoritmo findQuadruplets, le triplette vengono ricercate solamente nel primo blocco di quattro stazioni, come mostrato in figura 5.1.7.

[3]in realtà le strip R e strip ϕ non possono fisicamente intersecarsi, in quanto si trovano su due sensori affiancati che si trovano ad una distanza di circa un millimetro l'uno dall'altro. Confrontare anche la figura 2.2.3.

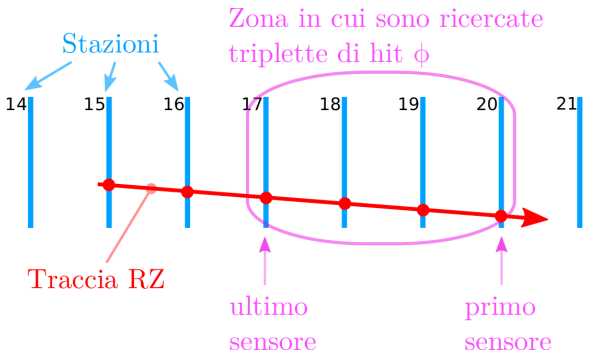

Figura 5.1.7: Ricerca delle triplette di punti per formare tracce spaziali.

La ricerca delle triplette procede considerando cinque casi, riassunti nella tabella 5.3. Nel primo caso si ricercano triplette di hit ϕ non usati che hanno colpito i primi tre sensori, nel secondo caso si rilassa la condizione e si cercano hit usati e non usati nei primi tre sensori, e negli ultimi tre casi si ammette che un hit sia mancante in uno dei sensori ϕ.

Caso	Configurazione	Commento
0	XXX	Solo hit non usati
1	XXX	Tutti gli hit
2	XX.X	Tutti gli hit
3	X.XX	Tutti gli hit
4	.XXX	Tutti gli hit

Tabella 5.3: Possibili triplette. Una X corrisponde alla presenza di un hit e un punto ad un hit mancante.

Per determinare una tripletta si considera una coppia di hit ϕ, *h1* e *h2*, appartenenti al primo e al secondo sensore (ad esempio i sensori 19 e 20 in figura5.1.7). Quindi viene calcolata la quaterna di parametri (x_0, y_0, t_x, t_y) della retta passante per *h1* e *h2*, dopodiché per ogni hit *h3* nel terzo viene calcolato il $\chi^2 = \frac{(ax+by+c)^2}{\sigma_i^2}$ dove x e y sono fornite da 5.1.1, e a, b, e c parametrizzano il cluster corrispondente ad *h3*. Abbiamo $\chi^2 = \frac{(a(x_0+t_x z)+b(y_0+t_x z)+c)^2}{\sigma_i^2}$. Viene scelto l'hit *h3* con χ^2 minimo, purché sia minore di quattro volte il parametro *maxChi2ToAdd* (vedi tabella 5.4).

5.1.3.3 Estensione delle triplette

La tripletta viene poi estesa in modo simile a quanto avviene nell'algoritmo findQuadruplets. La ricerca si interrompe quando non è possibile trovare hit compatibili in due stazioni successive. Per essere aggiunti gli hit devono avere un contributo χ^2 inferiore a *MaxChi2ToAdd*, con l'eccezione dei sensori con grande gap z, per i quali il limite è quattro volte *MaxChi2ToAdd*. Se

alla fine dell'analisi di un caso della tabella 5.3 una traccia estesa contiene lo stesso numero di hit R e ϕ, la ricerca è considerata conclusa e il ciclo sui casi viene abortito.

5.1.3.4 Fit della traccia tridimensionale

Il fit della traccia spaziale è calcolato minimizzando il seguente χ^2:

$$\chi^2 = \sum_i \frac{(a_i x + b_i y + c_i)^2}{\sigma_i^2},$$

dove x e y sono forniti dall'equazione della retta interpolante tridimensionale 5.1.1. L'indice i corre sugli hit della traccia, sia R che ϕ. Entrambi i tipi di hit sono parametrizzati da una retta; come accennato, poiché le strip R hanno la forma di un arco di cerchio, ai fini del track fitting l'hit R è parametrizzato come una retta tangente alla strip nel punto spaziale (x, y) calcolato in precedenza. Pertanto ogni hit è parametrizzato da una quaterna (a, b, c , z). Il peso $\frac{1}{\sigma_i^2}$ è funzione dello spessore e il numero delle strip costituenti il cluster.

Sostituendo x e y si ha:

$$\chi^2 = \sum_i \frac{(a_i(x_0 + t_x z_i) + b_i(y_0 + t_x z_i) + c_i)^2}{\sigma_i^2}.$$

Calcolando le derivate parziali rispetto ai parametri (x_0, y_0, t_x , t_y) della retta interpolante si ottiene un insieme di quattro equazioni in quattro incognite, risolvibile per sostituzione. La soluzione è funzione di coefficienti che sono somme di termini che dipendono da a, b, c e z; per aggiungere o rimuovere un hit dall'interpolazione è sufficiente aggiornare queste somme. L'implementazione della regressione lineare in tre dimensioni tramite il metodo della sostituzione risulta nettamente più veloce di altri metodi, quali ad esempio la decomposizione di Choleski [19].

5.1.3.5 Tracce in overlap non promosse a tracce spaziali

Se una traccia RZ che si trovava nella regione di overlap non è stata promossa a traccia spaziale, si prendono tutti gli hit filtrati precedentemente e si effettua un'interpolazione di tutti gli hit ϕ, ignorando la precedente ricerca basata sulle triplette. Una volta effettuata l'interpolazione si rimuove l'hit con il più grande contributo χ^2 finché il contributo di ciascun hit è inferiore a *MaxChi2PerHit*. Vengono rimossi prima gli hit nei sensori che contengono più di un hit, successivamente i rimanenti.

Se il recupero fallisce, questo vuol dire che la traccia è probabilmente una coincidenza di due tracce, una nella metà destra del VELO e una nella metà sinistra, ma aventi la stessa proiezione RZ. In questo caso la traccia RZ iniziale viene separata in due tracce, una con gli hit R provenienti dal lato destro, e una con gli hit R provenienti dal lato sinistro. La funzione *makeSpaceTracks* esegue due chiamate ricorsive per riprocessare le due tracce appena create.

5.1.3.6 Selezione finale

Se ci sono più candidati possibili per una singola traccia RZ, è possibile che alcuni di essi siano dei cloni. Per questo motivo è inserito un controllo realizzato dalla funzione *mergeClones*: se due tracce candidate condividono una frazione di hit superiore al parametro *FractionForMerge*, quella più corta, oppure quella con χ^2 peggiore, viene eliminata.

Infine le tracce vengono nuovamente interpolate e sono eliminati gli hit con χ^2 maggiore di *MaxChi2PerHit*. Se rimangono almeno 6 hit R o ϕ dopo quest'ultimo passaggio, la traccia è considerata valida e può passare alla fase successiva.

Parametro	Valore predefinito	Commento
PhiMatchZone	0.410	Imposta la massima distanza di una cluster ϕ dalla bisettrice del settore.
PhiCentralZone	0.040	Imposta la massima distanza di una strip ϕ dalla bisettrice del settore nel caso di tracce in overlap.
MaxChi2ToAdd	40	Massimo valore di χ per il terzo hit che forma una tripletta.
FractionFound	0.70	Rapporto minimo di hit ϕ rispetto al numero di stazioni che possono essere colpite dalla traccia.
MaxDelta2	0.05	Massima distanza (componente perpendicolare al piano bisettore) tra il primo e il secondo hit di una tripletta in ϕ.
MaxChi2PerHit	12	Massimo contributo al χ^2 per un singolo hit dopo aver eseguito il fit su una tripletta.
MaxQFactor	6	Massimo fattore di qualità per una traccia spaziale estesa.
MaxQFactor3	3	Massimo QFactor per una traccia che ha solo 3 hit R e 3 hit ϕ.
DeltaQuality	0.5	Da aggiungere a MaxQFactor quando ci sono più candidati con lunghezza superiore a 3.
FractionForMerge	0.70	Determina la percentuale minima di hit ϕ in comune perché due candidate tracce spaziali vengano unite.

Tabella 5.4: Parametri configurabili relativi alla funzione makeSpaceTracks

5.1.4 La funzione mergeSpaceClones

La funzione *mergeSpaceClones* ha il compito di ridurre il numero di cloni di tracce spaziali in uscita da *makeSpaceTracks*. Questa funzione considera tutte le coppie di tracce e se una certa coppia soddisfa un certo criterio di vicinanza (descritto più avanti), tale coppia viene fusa in un unica traccia, e viene effettuata nuovamente l'interpolazione. Se l'algoritmo FastVelo viene privato della funzione *mergeSpaceClones*, si ottiene un clone rate pari al 1-2% contro lo ∼0.5%.

Il criterio di vicinanza scelto è il seguente: in primo luogo le tracce candidate per la fusione devono essere "vicine", ossia i loro coefficienti angolari devono essere simili e il punto di minima distanza deve essere inferiore ad un certo valore limite (confrontare tabella 5.5). Se la condizione appena descritta viene soddisfatta, sono calcolati i seguenti valori: *nRCommon*, hit R in comune tra le due tracce, *nPhiCommon,* ossia hit ϕ in comune tra le due tracce, *nRInBoth*, numero di sensori R in cui ci siano hit provenienti da entrambe le tracce e *nPhiInBoth*, numero di sensori ϕ in cui ci siano hit provenienti da entrambe le tracce [4]. A questo punto la decisione di fondere la coppia è basata sulla seguente condizione.

$$\begin{aligned}((nRInBoth &= 0) \vee (nRCommon > 0.4 \times nRInBoth)) \wedge \\ &((nPhiInBoth = 0) \vee (nPhiCommon > 0.4 \times nRPhiInBoth))\end{aligned}$$

Tale condizione tiene conto del fatto che le tracce cloni possono essere allineate ma avere hit su insiemi di sensori disgiunti (condizioni $nRInBoth = 0$ e $nPhiInBoth = 0$) oppure avere effettivamente molti hit in comune (in questo caso gli hit in comune devono essere presenti in almeno il 40% dei sensori in comune).

Parametro	Valore predefinito	Commento
MaxDeltaSlopeToMerge	0.002	Se la differenza tra i coefficienti angolari (t_x o t_y) delle le due tracce spaziali supera questo valore, non sono fuse.
MaxDistToMerge	0.1 mm	Se minima distanza tra le due tracce spaziali supera questo valore, non sono fuse.

Tabella 5.5: Parametri configurabili relativi alla funzione mergeSpaceClones.

[4] ogni traccia possiede al più un hit per sensore R o ϕ

6

Note sulla programmazione della GPU

Una GPU (Graphics Processing Unit) è un dispositivo elettronico specializzato nell'elaborare immagini. Le GPU trovano impiego in diverse applicazioni, tra le quali personal computer, workstation e telefoni cellulari. Nell'ultimo decennio le prestazioni computazionali dei dispositivi legati alla grafica dei personal computer sono aumentate vertiginosamente, trainate dalla fertile industria del *gaming* (videogiochi).

Nella prima sezione di questo capitolo sono introdotte le caratteristiche delle GPU, nella seconda sezione sono date alcune definizioni, nella terza sezione sono descritte le risorse di memoria e nell'ultima sezione si forniscono alcuni criteri per l'ottimizzazione delle prestazioni.

6.1 Prestazioni a confronto

La struttura di una GPU è essenzialmente diversa da quella della CPU (central processing unit): mentre la prima è specializzata nell'effettuare calcoli paralleli, la seconda è pensata per eseguire algoritmi "seriali".

L'utilizzo delle GPU per affrontare problemi generici sfruttando le potenzialità del calcolo parallelo è noto come GPGPU (*general-purpose computing on graphics processing units*); sono stati sviluppati linguaggi di programmazione appositi, i principali attualmente sono OpenCL

(*Open Computing Language*) e CUDA (*Compute Unified Device Architecture*).

La prima versione di CUDA è stata rilasciata nel 2007, mentre la pubblicazione delle prime specifiche del linguaggio OpenCL risale al 2008.

Possiamo paragonare le GPU e le CPU in termini di potenza di calcolo; un parametro che può essere usato come indicatore delle prestazioni di un sistema di calcolo è il GFLOP/s (Giga FLoating point OPeration al secondo), cioè miliardi di operazioni in virgola mobile al secondo. Le GPU tipicamente offrono una potenza di calcolo nettamente superiore alle CPU, come mostrato nel grafico di figura 6.1.1.

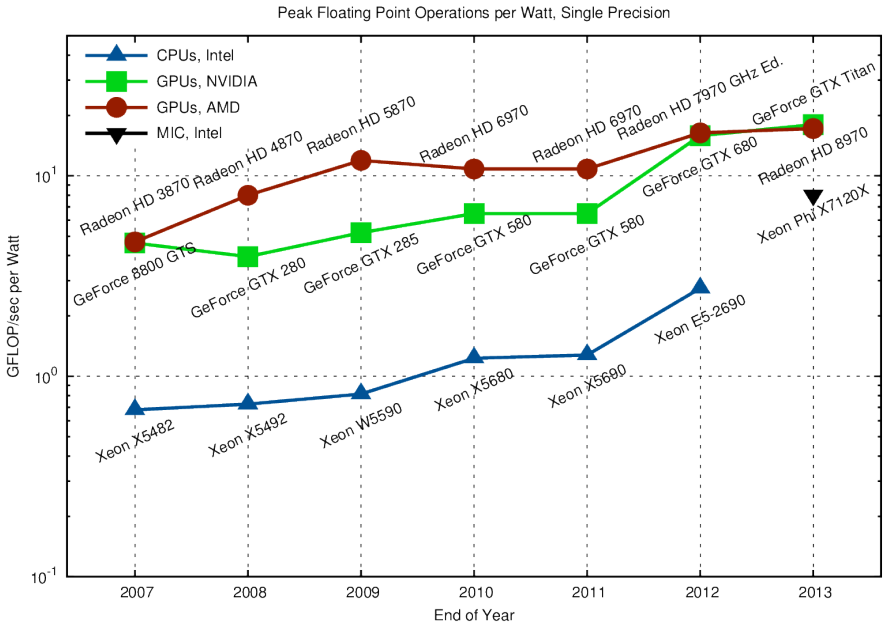

Figura 6.1.1: Confronto delle prestazioni (GFLOPS/Watt) di sistemi GPU e CPU Multicore [68].

Per gli studi discussi in questa tesi è stata usata una scheda NVIDIA GTX Titan. Questa scheda è stata introdotta nel 2013 ed ha le caratteristiche riportate nelle tabelle 6.1 e 6.2. Uno schema del processore grafico che costituisce il cuore della scheda è mostrato in figura 6.1.2. Per semplicità, gli esempi numerici riportati nel seguito di questo capitolo faranno sempre riferimento alle caratteristiche di questa scheda.

6.2 Alcune definizioni

In questa sezione verranno introdotte alcune definizioni [62]. Faremo qui riferimento alla nomenclatura propria del linguaggio CUDA; tuttavia le definizioni sono simili anche per il linguaggio OpenCL [70]:

Parametro	Valore
Codename	GK110
Architettura	Kepler2
Compute capability	3.5
SMX (Streaming multiprocessors)	14
Core CUDA (single precision) per SMX	192
CUDA core totali (single precision)	2688
Memoria RAM	6144 MB GDDR5
Memory bandwitdh (peak)	288 GB/s
Base Clock (Boost Clock)	836 MHz (876 MHz)
Numero di transistor	7.1×10^9
Costo unitario	$\simeq 800$ €

Tabella 6.1: Specifiche tecniche della NVIDIA GTX Titan

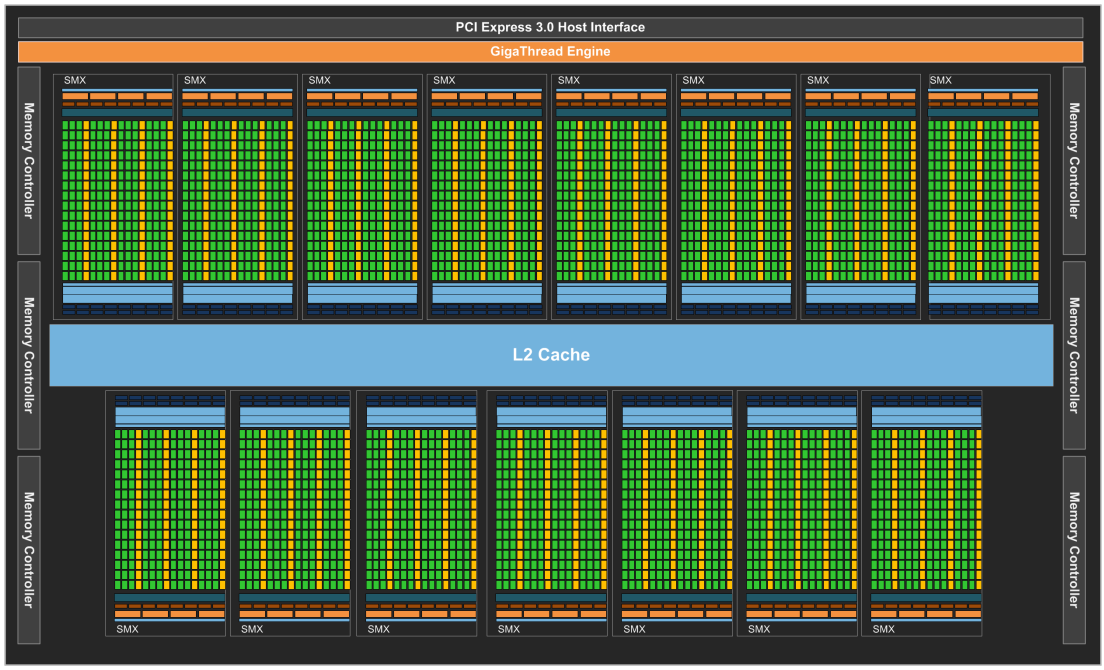

Figura 6.1.2: Struttura del processore grafico GK-110 (architettura Kepler2). Nella GTX Titan uno dei 15 SMX è disabilitato. In verde sono indicati i core single precision (float), in giallo quelli double precision (double).

Parametro	Valore
Numero di registri per SMX (32 bit ognuno)	65536
Memoria shared per SMX (massima)	48 KiB
Numero massimo di blocchi per SMX	16
Numero massimo di thread per blocco	1024
Numero massimo di thread per SMX	2048

Tabella 6.2: Alcuni parametri relativi alla compute capability 3.5

- per *host* si intende il calcolatore che ospita su CPU;
- per *device* si intende il dispositivo GPU che è in grado di eseguire il codice parallelo;
- un *kernel* è una funzione che può essere invocata dall'*host* ed essere eseguita sul *device;*
- un *thread* corrisponde all'esecuzione di un kernel dato un particolare indice i. Tipicamente, ad ogni thread è assegnato un indice i che viene utilizzato per accedere all'elemento i-esimo di un array in input[1]; l'insieme di tutti i thread in esecuzione cooperano per elaborare l'insieme dei dati di input;
- un *blocco di thread* (*thread block*) è un insieme di thread. Non è possibile prevedere l'ordine in cui i vari thread di un blocco vengano eseguiti: possono essere eseguiti in modo concorrente o seriale senza seguire un ordine particolare[2]. Tuttavia, i thread di un blocco possono essere sincronizzati[3] tramite l'istruzione _syncthreads() di CUDA. La sincronizzazione è utile quando è necessario scambiare informazioni tra uno o più thread in esecuzione concorrente;
- una *griglia* (*grid)* è un gruppo di blocchi. Non c'è alcun meccanismo di sincronizzazione tra blocchi diversi. I blocchi di una griglia sono implicitamente sincronizzati quando l'esecuzione di un kernel termina. Quando l'host invoca un kernel, questo viene applicato ad una griglia, della quale sono specificate le dimensioni, in termini di numero di blocchi e numero di thread per blocco;
- uno *streaming multiprocessor* (SM[4]) è unità di calcolo autosufficiente; ciascun SM si occupa di eseguire un certo numero di blocchi di thread. È importante notare che uno SM può

[1] l'indice in generale può essere multidimensionale, e corrispondentemente l'array può essere multidimensionale.

[2] tuttavia, i blocchi di thread sono a loro volta suddivisi gruppi di thread chiamati *warp*. All'interno di un warp l'esecuzione è di fatto sincrona (confrontare sezione 6.4.1).

[3] cioè si può fare in modo che tutti i thread completino di eseguire le istruzioni fino ad un certo punto nel codice.

[4] chiamati "SMX" nel caso della scheda GTX Titan.

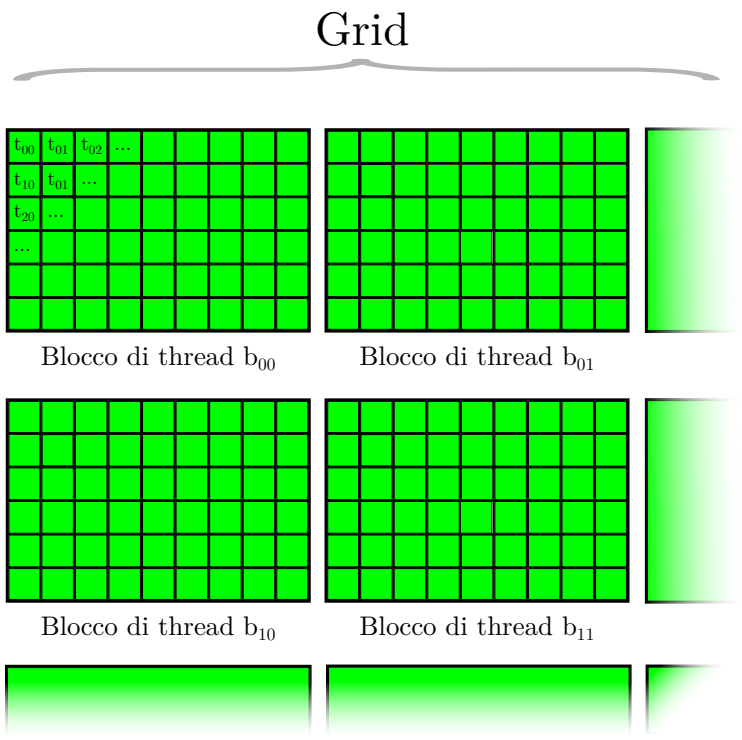

Figura 6.2.1: Suddivisione della grid in blocchi di thread.

eseguire un nuovo blocco solamente quando l'esecuzione di thread del blocco precedente è terminata.

6.3 Risorse di memoria della GPU

La memoria disponibile in un dispositivo GPU è di diverso tipo: possiamo distinguere diversi tipi di memorie a seconda di due caratteristiche: dimensione e banda. Per banda si intende la velocità di trasferimento, misurabile in byte al secondo. I tre principali tipi di memoria sono elencati nella tabella 6.3.

La componente di dimensioni maggiori (nel nostro caso la dimensioni è di 6 GB) è la cosiddetta memoria globale: è chiamata così perché è condivisa da tutte le unità di calcolo. È nota anche come *off-chip* memory, nel senso che non si trova nello stesso chip dove risiedono le unità di calcolo. Nel nostro caso la velocità di trasferimento dalla memoria globale alla memoria più performante (registri o memoria shared) si attesta all'incirca sui 250 GB/s. Il trasferimento tra la memoria globale dalla memoria della CPU (host) è più lento, circa 12 GB/s ed è limitata dalle caratteristiche del bus PCI-express del computer. La memoria invece più veloce è quella dei registri, che permette di manipolare i dati nel modo più veloce possibile. Nel nostro caso, ogni multiprocessore SMX contiene al massimo 65536 registri a 32 bit. È una risorsa limitata,

Tipo	Banda	Scope
Memoria globale	bassa host $\leftrightarrow$ globale: $\simeq$ 12 GB/s registri/shared $\leftrightarrow$ globale: $\simeq$ 250 GB/s	globale
Memoria shared	alta (paragonabile ai registri)	locale al blocco (*block-local*)
Registri	massima	locale al thread (*thread-local*)

Tabella 6.3: Principali tipi di memoria della GPU

perché è condivisa da tutti i thread che sono in esecuzione su di un multiprocessore. Un terzo tipo di memoria è la memoria shared: è *on-chip*, ossia risiede nello stesso chip dove si trovano le unità di calcolo, ed ha prestazioni paragonabili, ma inferiori, a quelle dei registri. Viene in genere utilizzata per immagazzinare informazioni comuni a tutti i thread di un blocco. Uno schema delle risorse di memoria è riportato in figura 6.3.1.

Per quanto detto, al fine di massimizzare le prestazioni di un programma per GPU, è necessario limitare al minimo indispensabile i trasferimenti più lenti, in primo luogo quelli tra host e memoria globale e sfruttare il più possibile la memoria veloce (registri e memoria shared).

6.4 Ottimizzazione delle prestazioni

La prestazione di una generica applicazione GPU può essere limitata da diversi fattori. I principali sono:

- Larghezza di banda e/o latenza di trasferimento nella memoria della GPU (in particolare tra memoria globale e registri o shared memory)

- Calcolo in virgola mobile (in termini di GFLOP/s), in singola precisione (float) o doppia precisione (double) oppure dalla velocità di esecuzione di istruzioni in generale.

Tipicamente una applicazione è limitata da una di queste due caratteristiche o una combinazione di esse, si parla pertanto di applicazioni *memory-bound* (limitate dalle caratteristiche della memoria) o *compute-bound* (limitate dal calcolo) oppure *instruction-bound* (limitate dalla velocità di esecuzione di istruzioni in generale).

Esistono ulteriori limitazioni dovute alle caratteristiche dell'hardware della GPU. Nei prossimi paragrafi discuteremo il concetto della *località dei dati*, la definizione di *occupanza,* e il fenomeno della cosiddetta *divergenza dei thread.*

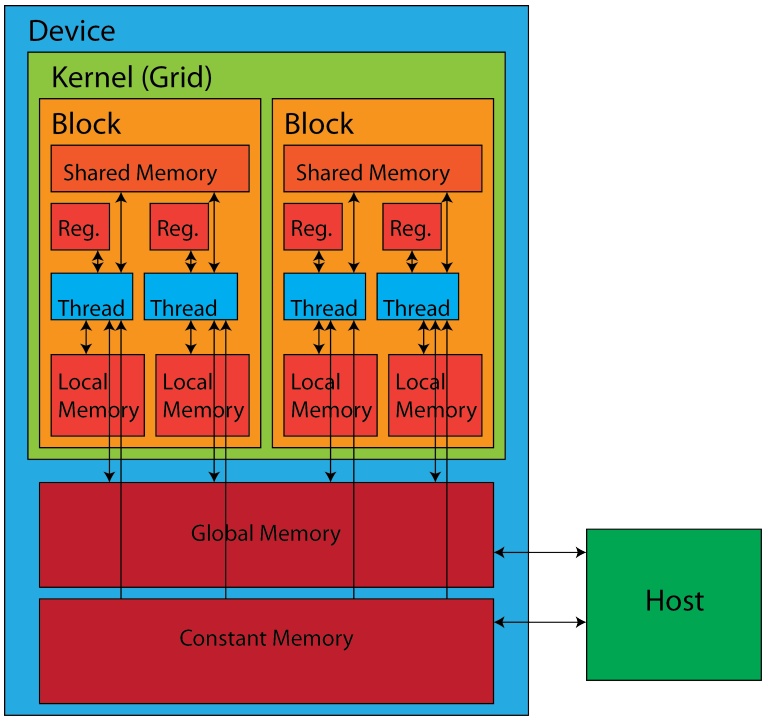

Figura 6.3.1: Struttura delle risorse di memoria della GPU. Nota: la *Local Memory* e la *Constant Memory* in realtà risiedono in memoria globale (memoria *off-chip*), nella figura è evidenziato il campo di visibilità (*scope*).

6.4.1 Divergenza dei thread

I thread di un blocco vengono a loro volta suddivisi ed eseguiti in blocchi da 32 threads, detti warp. A livello di warp ogni thread esegue necessariamente la stessa istruzione, ma su dati diversi. Si parla in questo caso di *SIMD* (*single instruction, multiple data*). Nel caso in cui sia presente una istruzione condizionale, se parte dei thread segue un certo percorso e parte un altro percorso, il warp è costretto ad eseguire entrambi "serializzando" l'esecuzione. Questo problema è invece assente se si fa in modo che tutti i branch di uno stesso warp seguano lo stesso percorso, anche in presenza di istruzioni condizionali. Se il codice contiene un numero eccessivo di istruzioni che portano alla creazione di numero elevato *branch* divergenti, le prestazioni del codice degradano. Il concetto è illustrato in figura 6.4.1.

6.4.2 Località dei dati

È possibile avvicinarsi alla banda teorica riportata nella tabella 6.1 solamente in particolari condizioni: i thread di un warp devono accedere a dati contigui in memoria globale o comunque vicini. Quando questa condizione è verificata si parla di *coalescenza* nell'accesso, in quanto più transazioni vengono raggruppate e ottimizzate. La distanza tra un dato e l'altro in memoria globale è definita *stride* e le prestazioni di trasferimento variano di molto a seconda di questo

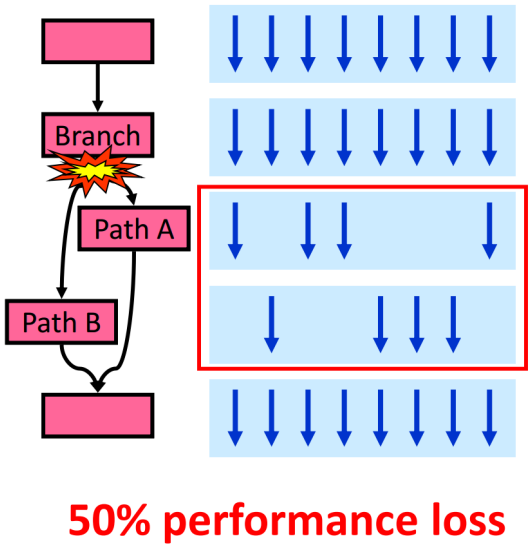

Figura 6.4.1: Illustrazione del concetto di divergenza

parametro, come riportato in [66].

6.4.3 Occupanza

L'occupanza è definita come numero di warp che sono in esecuzione contemporanea rispetto al numero massimo possibile in uno SM (streaming multiprocessor). È in pratica equivalente definire questo parametro come rapporto tra il numero di thread eseguiti contemporaneamente e il massimo di thread eseguibili contemporaneamente da uno SM.

Una buona occupanza permette di migliorare le prestazioni nel caso di un codice che abbia come collo di bottiglia la banda di memoria[5], mentre nel caso in cui il codice sia essenzialmente dominato dal "calcolo" (cioè operazioni in virgola mobile) aumentare l'occupanza non permette di migliorare sensibilmente le prestazioni.

È possibile determinare l'occupanza di un kernel tramite il foglio elettronico [64]. Una possibile combinazione di parametri che massimizza l'occupanza è riportata in tabella 6.4.

Purtroppo non sempre è possibile rispettare i limiti imposti dall'hardware ed è necessario giungere ad un compromesso.

6.4.3.1 Numero massimo di blocchi per streaming multiprocessor

Consultando le caratteristiche riportate in tabella, si nota che esiste un limite massimo (16) di blocchi per multiprocessore. Per fare in modo che vengano eseguiti il massimo numero di

[5] la banda ottenuta è legata alla latenza di accesso alla memoria; avere più thread in esecuzione permettere di "nascondere" la latenza mettendo in coda e temporizzando opportunamente gli accessi alla memoria globale.

Parametro	Valore
Compute Capability	3.5
Shared Memory Size Config	49152 byte
Threads Per Block	128
Registers Per Thread	32
Shared Memory Per Block	3072 byte
Occupanza teorica	100%

Tabella 6.4: Esempio di configurazione che massimizza l'occupanza.

thread contemporaneamente, la dimensione *minima* dei blocchi deve essere 128. In questo modo 128×16 = 2048 thread, e in questo modo si raggiunge il numero massimo di thread per SM.

6.4.3.2 Registri

Se non intervengono altri fattori limitanti, un multiprocessore può eseguire 2048 thread contemporaneamente, e ogni thread ha a disposizione 65536/2048 = 32 registri. Se il kernel utilizza più di 32 registri, la macchina è costretta ad eseguire meno thread contemporaneamente per mancanza di risorse. Al fine di massimizzare l'occupanza è pertanto necessario limitare il numero di registri utilizzati.

6.4.3.3 Memoria shared

La memoria shared come accennato è on-chip, è configurabile ed è di default impostata con dimensioni 48 KiB = 48×1024 byte = 49152 byte per SM. Se un multiprocessore lancia N blocchi, ogni blocco avrà a disposizione 49152 / N byte di memoria shared. Se un blocco di thread ne usa di più, la macchina sarà costretta a lanciare meno blocchi contemporaneamente.

Ad esempio, se il nostro blocco ha 1024 thread e usa 24 KiB = 24 × 1024 byte = 24576 byte, la macchina potrà lanciare 2 blocchi contemporaneamente, raggiungendo la massima occupanza (1024×2 = 2048 thread per multiprocessore) e occupando totalmente la memoria shared (48 KiB). Se il blocco richiede invece solo un byte in più, ossia 24577 byte, ogni streaming multiprocessor sarà costretto a lanciare solo un blocco da alla volta.

6.4.3.4 Thread inattivi

Un ultimo fattore di cui tener conto è la distribuzione del lavoro tra i thread di un blocco. È opportuno fare in modo che un blocco che tutti i thread siano sottoposti ad un carico di lavoro simile, o in altre parole è necessario minimizzare il numero di thread inattivi (*idle threads*)

per unità di tempo. È infatti necessario che tutti i thread di un blocco terminino la propria esecuzione prima che un nuovo blocco possa essere processato.

La presenza di thread inattivi e codice divergente possono fare in modo che l'occupanza effettiva (numero di thread effettivamente in esecuzione rispetto al numero massimo di thread gestibili) sia inferiore all'occupanza teorica, che è determinata dall'utilizzo dei registri, della memoria shared e dal numero di thread per blocco. Per determinare l'occupanza effettiva (assieme ad altri parametri legati alle prestazioni) è possibile utilizzare un profiler; per CUDA è disponibile "NVIDIA Visual Profiler" [65].

7

FastVeloGPU

Lo scopo del lavoro di tesi è stato riscrivere l'algoritmo FastVelo per una piattaforma GPU. Come primo approccio si è deciso di implementare il codice cercando di mantenere la struttura originaria, seppure fortemente seriale, evitando di modificare in modo sostanziale l'algoritmo esistente.

È stato sviluppato inizialmente un porting preliminare che era in grado di riprodurre esattamente i risultati dell'algoritmo, parallelizzando soltanto sugli eventi (ad ogni evento viene assegnato un singolo thread, e l'algoritmo non viene modificato). Tuttavia, seguendo questo approccio semplificato non si ottengono prestazioni soddisfacenti, per diverse ragioni: in primo luogo non si genera un numero sufficiente di thread, perché il numero di eventi caricabili in memoria è $\mathcal{O}(1000)$ mentre la macchina è in grado di eseguire $\mathcal{O}(10000)$ thread contemporaneamente[1]. Pertanto la strategia utilizzata è stata quella di parallelizzare sia a livello di "evento" che di "algoritmo". Ad esempio, per la funzione makeSpaceTracks e MergeSpaceClones si è seguita la strategia di parallelizzare sia sugli eventi (tra 1000 e 2000) che sulle tracce (circa 80), generando circa 80000 thread indipendenti.

In questo capitolo saranno descritte in dettaglio le modifiche apportate all'algoritmo ne-

[1] ad esempio la scheda NVIDIA GTX Titan, in condizioni ideali è in grado di gestire $2048 \times 14 = 28672$ thread contemporaneamente.

cessarie per poter eseguire l'algoritmo su piattaforma GPU. Nella sezione 7.1 sarà descritta l'infrastruttura necessaria per poter interfacciarsi con il software esistente basato sul framework Gaudi e il formato di input e di output dei dati, mentre nella sezione 7.2 saranno analizzate in dettaglio le modifiche apportate all'algoritmo FastVelo originale. Infine l'analisi delle prestazioni sarà discussa nel capitolo 9.

7.1 Framework per GPU

FastVeloGPU è stato progettato in modo da essere indipendente dal framework Gaudi, in quanto Gaudi è progettato per analizzare un evento alla volta[2], mentre in questo lavoro di tesi abbiamo avuto l'esigenza di analizzare più eventi in parallelo. Per questa ragione si è deciso di accedere ai dati "grezzi" nel modo più diretto possibile senza l'ausilio del framework Gaudi.

È stato deciso di utilizzare un particolare formato di input, il formato MDF, che sarà descritto nel paragrafo 7.1.2, e di implementare le operazioni di decodifica e di tracking sulla GPU[3]. L'output generato dall'algoritmo viene convertito solo alla fine in un formato compatibile con Gaudi. La strategia seguita è schematizzata nella figura 7.1.1.

7.1.1 Interfaccia con il database

Il codice originale si interfaccia con il database del sistema Gaudi. Tipicamente, quando viene lanciato una componente di Gaudi che opera sui dati, è necessario specificare le condizioni sperimentali di presa dati (per esempio, la geometria del detector). Anche nel caso di una simulazione è comunque necessario scegliere un particolare insieme di condizioni sperimentali.

Esiste un database che contiene le informazione che descrivono la geometria e le proprietà dei diversi rivelatori ed è chiamato *Detector Description Database* (DDDB). Un secondo database, chiamato *Condition Database* (CondDB) descrive invece gli allineamenti, cioè la posizione esatta dei sensori e dei moduli che li compongono. Per quanto riguarda il sensore VELO, nel database CondDB sono registrate diverse informazioni, tra le quali le posizioni dei sensori R e ϕ, misurate con precisione a livello del μm.

L'impostazione che determina in GAUDI la scelta del database opportuno può essere impostata tramite gli script python usati per configurare applicazioni quali MOORE o BRUNEL, ad esempio:

[2] più specificamente ogni processo di Gaudi (es Moore o Brunel) è a thread singolo e analizza un evento per volta, si noti che una macchina con CPU multicore è comunque in grado di eseguire più processi in modo parallelo, pertanto anch'essa è in grado analizzare più eventi simultaneamente.

[3] eventualmente alcune parti dell'algoritmo di decodifica o di tracking potranno essere spostate su CPU, se ciò dovesse risultare conveniente.

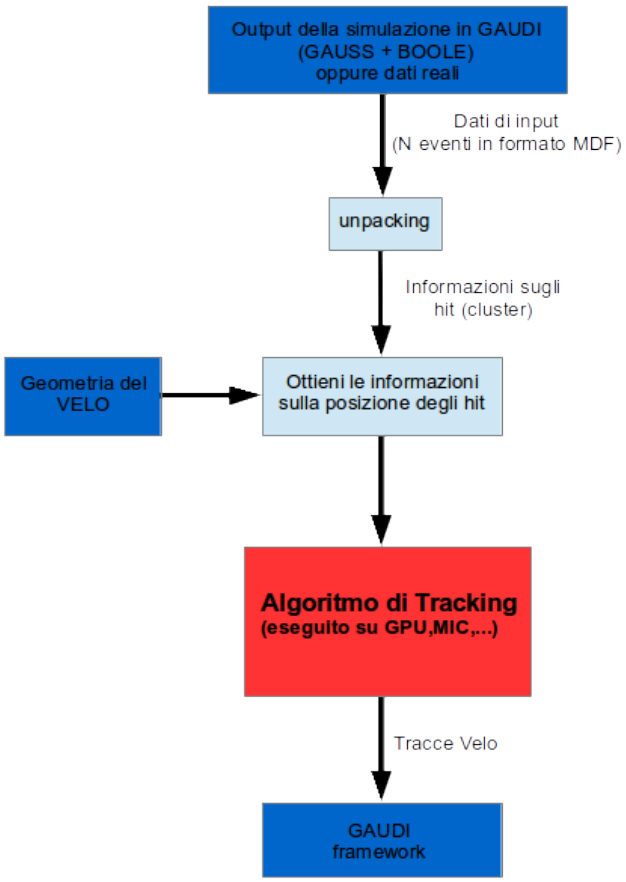

Figura 7.1.1: FastVeloGPU: interfaccia con il framework Gaudi.

```
Moore().DDDBtag   = "Sim08-20130503-1"
Moore().CondDBtag = "Sim08-20130503-1-vc-md100"
```

Il codice originale di Fastvelo ha bisogno del database di Gaudi per caricare le informazioni sulla geometria e l'allineamento dei sensori. Il codice di FastVeloGPU non realizza un accesso al database, ma utilizza alcuni file appositamente generati contenenti solamente le informazioni della geometria del VELO. I file sono generati tramite una versione modificata di FastVelo, eseguita su CPU. La dimensione dei file contenti le informazioni sulla geometria del VELO così prodotte è di circa 2 MB; il contenuto di questi file è riassunto nella tabella 7.1.

Le informazioni comuni per ogni sensore R sono: il raggio interno $innerRadius = 8.17$ mm, il raggio esterno $outerRadius = 42$ mm, la distanza tra le strip ($pitch$) nella zona interna ($innerPitch = 40\ \mu m$), e la variazione del $pitch$ con il raggio ($pitchSlope \simeq 1.82\ \mu m/mm$). Per i sensori ϕ le informazioni comuni sono $innerRadius = 8.17$ mm, $outerRadius = 42$ mm, $rMiddle = 17.25$ mm. L'ultimo parametro individua il raggio che separa le due zone a forma di corona di cerchio. Tra le informazioni specifiche di ciascun sensore ricavabili dal database, quelle più

File	Descrizione
RCommon.bin	Contiene le informazioni comuni ad ogni sensore R .
RSensors.bin	Contiene le informazioni specifiche di ogni sensore R.
PhiCommon.bin	Contiene le informazioni comuni ad ogni sensore ϕ.
PhiSensors.bin	Contiene le informazioni specifiche di ogni sensore ϕ. Qui è specificata anche la parametrizzazione a, b, c per ogni strip, più il centro della strip (x_s, y_s), per ogni strip e per ogni sensore.

Tabella 7.1: File che contengono le informazioni sulla geometria del VELO.

importanti sono la posizione del centro in coordinate globali (x, y, z) e l'inclinazione del sensore (dx/dz e dy/dz). Per le strip ϕ sono disponibili ulteriori informazioni come riassunto in tabella 7.1.

Le informazioni contenute in questi file vengono infine caricate nella memoria globale della GPU, dove sono disponibili per un accesso rapido da parte del kernel che si occupa della decodifica e del calcolo delle coordinate degli hit. Alcune di queste informazioni sono caricate nella *constant memory* della GPU, che è una porzione della memoria globale ottimizzata per immagazzinare dati costanti.

7.1.2 Formato di input

Come formato di input per i dati da caricare su GPU è stato scelto il formato MDF. Tale formato è molto simile al formato "raw" che proviene dal front-end dei rivelatori ed è il più agevole da decodificare.

Il formato MDF consiste in una sequenza di *header* (48 byte) e di *body*, di dimensione variabile; ciascun body contiene tutte le informazioni relative ad un singolo evento. La struttura del file è schematizzata nella figura 7.1.2. Per lo sviluppo del codice GPU non è stato necessario decodificare il contenuto degli header, è stato sufficiente decodificare il *body*.

I vari body sono in formato che possiamo chiamare *RawEvent*; il formato è descritto in [20]. Un blocco RawEvent contiene tutte le informazioni relative ad un evento, ossia le informazioni provenienti da tutti i rivelatori. Un RawEvent è costituito da una sequenza di RAW Data Bank

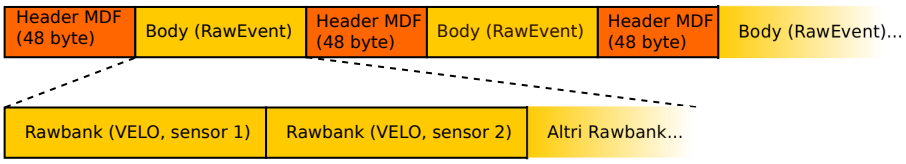

Figura 7.1.2: Struttura di un file MDF.

contigui, e ciascuno di questi contiene le informazioni relative ad un sotto-rivelatore. La parte

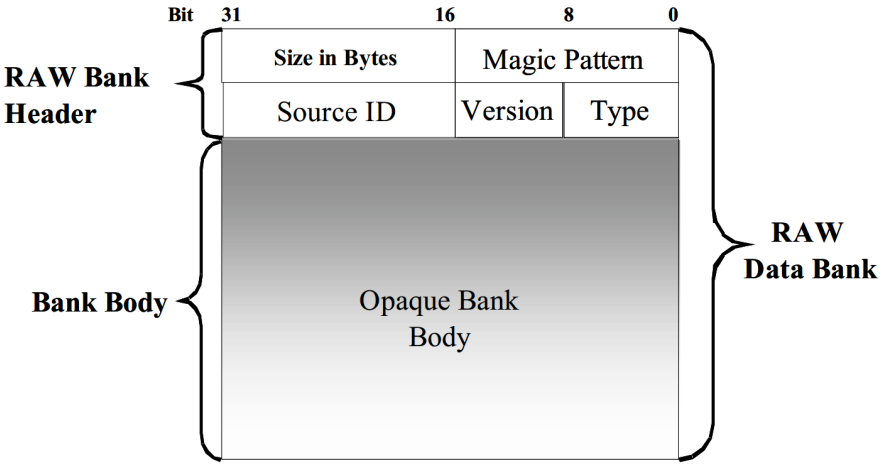

Figura 7.1.3: Struttura di un generico Raw Data Bank (RawBank)[20].

indicata con "opaque bank body" contiene le informazioni relative ad uno specifico sensore (o sotto-sensore). Nel caso del VELO, *type* = 8 e *sourceID* identifica il numero del sensore del VELO: i sensori R sono numerati da 0 a 41, mentre i sensori ϕ sono numerati da 64 a 105, i sensori pileup da 128 a 131. Nel codice pertanto è necessario leggere tutti gli header RAW Bank e selezionare quelli con *type* = 8. Il contenuto del *bank body* di tipo VELO è descritto in [23]. Nella tabella 7.2 è schematizzata la struttura di un VELO RawBank.

8 bits	8 bits	8 bits	8 bits
R 8 bits	PCN 8bits	Nr. of clusters 16 bits	
SOver 1bit | Cl. size 1bit | Cluster position cluster 1 14bits		SOver 1bit | Cl. size 1bit | Cluster position cluster 0 14bits	
SOver 1bit | Cl. size 1bit | Cluster position cluster 3 14bits		SOver 1bit | Cl. size 1bit | Cluster position cluster 2 14bits	
Padding 16bits		SOver 1bit | Cl. size 1bit | Cluster position cluster 4 14bits	
EOC 1bit "1" | ADC0 cluster 2 7bits	EOC 1bit "1" | ADC0 cluster 1 7bits	EOC 1bit "1" | ADC1 cluster 0 7bits	EOC 1bit "0" | ADC0 cluster 0 7bits
Padding 8bits	EOC 1bit "1" | ADC0 cluster 4 7bits	EOC 1bit "1" | ADC1 cluster 3 7bits	EOC 1bit "0" | ADC0 cluster 3 7bits

Tabella 7.2: Struttura di un VELO Raw Data Bank (Velo RawBank).

7.1.3 Decodifica

Al fine di decodificare i dati di input, abbiamo bisogno sia delle informazioni sulle strip colpite, sia i dati della geometria dei sensori.

Nel prototipo di codice realizzato abbiamo scelto di copiare interamente il file MDF di input nella memoria della GPU. Tenendo conto che la dimensione di un evento generato con $\nu = 2.5$ è circa 40 kByte, 1000 eventi occupano circa 40 MByte, che è una quantità di memoria accettabile, dato che la memoria a disposizione della GPU ha una dimensione di circa 6 Gigabyte. Il file MDF viene dapprima letto e copiato interamente in memoria RAM *pinned*[4] del computer host, dopodiché viene copiato interamente nella memoria della GPU. La banda ottenuta nel trasferimento di 1000 eventi è di circa 12 Gigabyte al secondo.

Dalle prove effettuate risulta che i dati del VELO costituiscono mediamente il circa il 10% delle dimensioni totali del file, cioè solamente ∼4 kB in un evento di dimensioni di 40 kB. Ad ogni modo, è stato deciso di copiarlo interamente per semplicità ed anche perché in futuro si prevede di sviluppare codice GPU per gli altri algoritmi di tracking, per esempio il *forward tracking* (vedi sezione 4.3), che richiede l'accesso ad ulteriori dati, non solo quelli relativi al VELO.

7.1.4 Formato dei dati in output

Il formato dei dati in output è costituito da un formato binario custom. Nel file sono contenute le informazioni relative al numero di tracce spaziali e, per ogni traccia, sono specificati i parametri spaziali (pendenza, intercetta) e la lista hit R e ϕ associati. L'informazione su ciascun hit associato ad una traccia consiste in un numero identificativo (*LHCbId)* che specifica il sensore e la strip di provenienza, utile per il calcolo delle efficienze e per i successivi algoritmi di tracking.

Le efficienze ottenute con l'algoritmo per GPU sono calcolate facendo uso del software BRUNEL. A tal fine è necessario fornire a BRUNEL anche le informazioni Monte Carlo della simulazione, ossia l'esatta natura e la posizione delle tracce "vere" simulate. Si può riassumere il procedimento nella schema 7.1.4.

7.2 FastVelo su GPU

In questa sezione discuteremo le strutture dati scelte per immagazzinare i dati nella memoria globale della GPU, dopodiché saranno descritti i vari kernel che compongono il software GPU: il kernel *findoffsets, unpack*, *findQuadruplets*, *delQuadrupletsClones, extendQuadruplets*, *delRZTracksClones*, *makeSpaceTracks* e *delSpaceClones*.

[4] consiste in un accesso più diretto alla memoria RAM che permette di sfruttare al meglio la banda del bus PCI-Express [63].

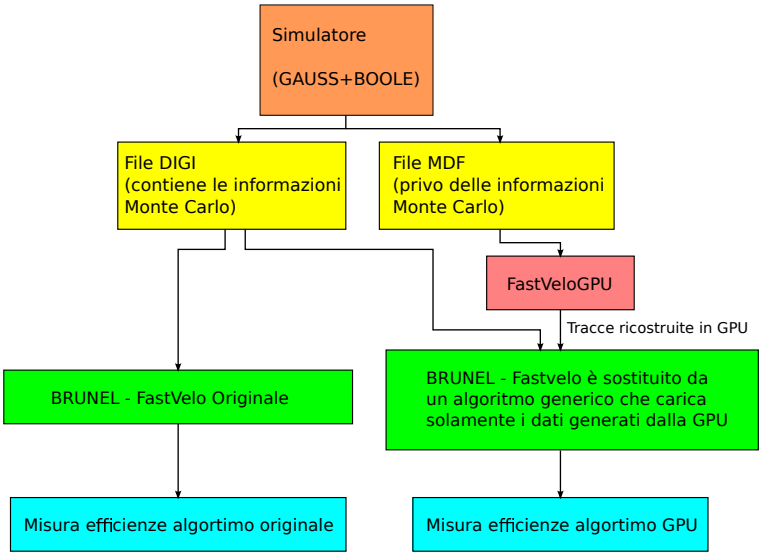

Figura 7.1.4: Analisi delle efficienze dell'output della GPU via GAUDI.

7.2.1 Strutture dati per GPU

Il kernel *unpack* si occupa di decodificare i dati di ingresso in strutture più agevoli da maneggiare successivamente. I dati di ingresso per il rivelatore VELO non sono altro che una sequenza di *hit* (chiamati alternativamente *cluster*). Il codice originale per CPU utilizzava un'unica struttura per immagazzinare sia gli hit R che gli hit ϕ, chiamata *VeloHit*. Nel codice GPU, al fine di ottimizzare l'utilizzo della memoria, sono stati creati due contenitori appositi per le due specie, ossia due classi *RHit* e *PhiHit*.

Una caratteristica che differenzia in modo sostanziale l'implementazione CPU e GPU è che tutti i contenitori (array) dei dati che risiedono nella memoria globale della GPU hanno dimensione fissa e sono tipicamente riempiti solo parzialmente; i limiti sono riportati in tabella 6.3. Nel caso in cui processando un evento in GPU venga superato uno di questi valori limite, si può decidere di salvarlo di comunque per poi riprocessarlo. Nella versione originale per CPU è usata quasi sempre *std::vector* della libreria STL che sfrutta invece l'allocazione dinamica (cioè porzioni di memoria di dimensione variabile vengono "richieste" ed "affidate" nel momento in cui servono).

La classe *RHit* contiene le informazioni specifiche di un hit R, quelle principali sono la coordinata r e quella z in coordinate globali, più il peso associato *weight*; la classe *PhiHit* contiene invece le informazioni di un hit Phi, cioè le tre coordinate a, b, c (i parametri della retta bidimensionale su cui giace la strip), più il peso *weight*. I pesi associati sono proporzionali a $1/\sigma^2$, dove σ è proporzionale alla dimensione del cluster e al *pitch* corrispondente alla strip(vedi capitolo 2). Il contenitore degli hit include un numero prefissato di "slots" in cui immagazzinare gli hit, ed è in pratica un array multi-indice strutturato nel seguente modo:

```
RHit[MAX_EVENTS][R_SENSORS_NUMBER][R_ZONES_NUMBER][MAX_HITS_PER_R_ZONE];
PhiHit[MAX_EVENTS][PHI_SENSORS_NUMBER][PHI_ZONES_NUMBER][MAX_HITS_PER_PHI_ZONE];
```

Purtroppo utilizzando questa strategia tale array non viene riempito completamente, ossia, dato un particolare evento, sensore e zona del sensore, il numero di hit effettivamente memorizzati sarà inferiore al limite (*MAX_HITS_PER_R_ZONE* oppure *MAX_HITS_PER_PHI_ZONE*). Infatti il numero medio di hit per zona R è circa 4, mentre quello per la zona ϕ è circa 8; alcuni eventi possono però contenere oltre 50 hit per zona (vedere anche sezione 4.5). Nella tabella 7.3 sono riportati i limiti massimi per la dimensioni dei contenitori usati in FastVeloGPU.

Parametro	Valore	Commento
MAX_EVENTS	1024	Il massimo numero di eventi da analizzare contemporaneamente
MAX_INPUT_SIZE_BYTES	10^8	La massima dimensione (in byte) del file MDF in input
MAX_HITS_PER_R_ZONE	80	Specifica il numero massimo di hit (cluster) R per un sensore di tipo R.
MAX_HITS_PER_PHI_ZONE	80	Specifica il numero massimo di hit (cluster) per una zona di un sensore ϕ.
MAX_RHITS_PER_TRACK	30	Specifica il numero massimo di hit (cluster) R per tracce RZ o spaziali.
MAX_PHIHITS_PER_TRACK	30	Specifica il numero massimo di hit (cluster) ϕ per una traccia spaziale.
MAX_RZ_TRACKS	3500	Specifica il numero massimo di tracce RZ.
MAX_SPACETRACKS_PER_EVENT	500	Specifica il numero massimo di tracce spaziali.
MAX_GOODSPACEPOINTS	45	Specifica il numero massimo di hit candidati a formare una traccia spaziale.

Tabella 7.3: Valori massimi impostati per diversi parametri di FastVeloGPU. Se uno di questi valori limite è superato, l'evento non viene processato.

7.2.2 I kernel findoffsets e unpack

Questi due kernel si occupano di decodificare i dati in ingresso e immagazzinarli in strutture dati più maneggevoli, *RHit* e *PhiHit*, discusse nel paragrafo precedente. Il kernel *findoffsets* si

occupa di trovare l'offset[5] corrispondente ad all'inizio di ciascun evento nel file MDF (gli offset non sono noti a priori perché la dimensione di ogni evento è variabile). Il successivo kernel *unpack* si occupa di popolare le strutture $RHit$ e $PhiHit$ integrando i dati presenti nel file di input con le informazioni sulla geometria. Grazie all'informazione sugli offset di ciascun evento ottenuta precedentemente, è possibile eseguire il kernel *unpack* parallelizzando sugli eventi.

Poiché il tempo di decodifica risulta piccolo ($\sim 5\%$) rispetto al tempo impiegato per l'esecuzione dell'algoritmo di tracking (RZ tracking e tracking spaziale), i kernel findoffsets e unpack non saranno discussi ulteriormente.

7.2.3 Il kernel findQuadruplets

L'algoritmo originale ricerca una quadrupletta di hit per volta. Una volta che è stata trovata una combinazione di hit valida, vengono marcati gli hit corrispondenti come usati (*tagging*).

Nel codice GPU si è invece interessati ad eseguire più ricerche di quadruplette contemporanee per massimizzare il numero di thread lanciati in parallelo. Tuttavia in un algoritmo parallelo non è possibile prevedere l'ordine in cui gli hit vengono marcati come usati, pertanto l'approccio del tagging va abbandonato, in quanto genera *race condition*[6].

Avendo rimosso il tagging degli hit usati, è possibile suddividere la ricerca in più thread. È stato scelto di suddividere la ricerca per zona (4 zone per sensore ogni sensore R, corrispondenti ad un settore di 45°), per sensore da cui far partire la ricerca (35 sensori[7]) e per "Caso" (descritto nella sezione 5.1.1). È possibile creare un blocco di dimensioni $4 \times 4 \times 35 = 560$ thread che gestisce un evento.

Avendo rimosso il tagging degli hit usati, il problema che si ottiene a questo punto è che le quadruplette trovate comprendono molti cloni. I cloni generati in questa fase sono rimossi mediante un kernel apposito chiamato *delQuadrupletsClones*.

7.2.4 Il kernel delQuadrupletsClones

Questo kernel controlla ogni coppia di quadruplette trovate e, se una coppia di tracce contiene più di ϵ hit in comune, elimina quella con χ^2 maggiore. Il numero di hit in comune minimo è stato impostato a $\epsilon = 2$. Il pseudocodice corrispondente è (N è il numero di quadruplette):

```
1 per i da 1 ad N−1:
2   per j da i+1 ad N:
```

[5] l'offset non è altro che la distanza (in byte) rispetto ad un punto stabilito (all'inizio, se non specificato) di una sequenza di dati.

[6] Per race condition si intende una condizione per la quale il risultato finale dell'esecuzione di un sistema basato su processi multipli concorrenti dipende da fattori incontrollabili, quali la temporizzazione o l'ordine in cui i processi sono eseguiti.

[7] il numero di sensori da cui far partire la ricerca delle quadruplette è determinato dai parametri ZVertexMin e ZVertexMax; con le impostazioni predefinite il numero dei sensori usati come punto di partenza è 35.

```
3     hitInComune = 0;
4     per ogni hit phi h1 della traccia t[i]:
5       per ogni hit phi h2 della traccia t[j]:
6         se (h1 == h2) hitInComune = hitInComune + 1;
7     minimaLunghezza = min( numeroDiHitR(t[i]), numeroDiHitR(t[j]) )
8     se (hitInComune > 2):
9       se ( Chi2(t[i]) < Chi2(t[j]) ): marca t[j] come non valida;
10        altrimenti: marca t[i] come non valida;
```

Questo algoritmo richiede un tempo di esecuzione piuttosto lungo, in quanto deve effettuare $\mathcal{O}(N^2)$ confronti tra tracce[8], dove N è il numero di tracce medio per evento.

Per quanto riguarda le prestazioni, notiamo che per la presenza di cicli *for* con limiti dipendenti dai dati e per la presenza di numerose istruzioni condizionali, non possiamo aspettarci prestazioni eccellenti dal kernel *delSpaceClones*. Tale scelta comporta però il costo (temporale) di trasferire i dati su CPU ed il trasferimento dei risultati nuovamente in memoria globale della GPU.

7.2.5 Il kernel extendQuadruplets

Questo kernel esegue l'estensione delle tracce RZ come nel codice originario, all'interno della funzione findQuadruplets; nel codice GPU si è scelto di implementare la funzione in un kernel separato. Differentemente dal codice originale, si è scelto di estendere la quadrupletta in entrambe le direzioni, sia nella direzione di propagazione delle particella che in quella contraria[9]. Questa scelta è legata al fatto che il tagging degli hit usati è stato rimosso. Dalle prove effettuate risulta che effettuando l'estensione solo da un lato si ottiene un numero di ghost più alto (1-2%) e bassa efficienza negli hit[10] ($\sim$87% contro il $\sim$97%).

È stato possibile parallelizzare questo kernel in funzione di due indici: l'indice dell'evento e l'indice della quadrupletta di partenza. A ciascun blocco sono assegnate tutte le quadruplette generate da un evento. Poiché il numero di quadruplette medio è superiore ad 80 (vedi sezione 4.5), si ottiene un numero elevato di thread per blocco (circa 1000 eventi $\times$ 80 quadruplette/evento $\times$ 1 thread/quadrupletta = 80000 thread).

7.2.6 Il kernel delRZTracksClones

Il kernel *delRZTracksClones* per ogni coppia di tracce RZ di un certo evento controlla il numero di hit R in comune. Se il numero di hit in comune è maggiore del 70% del numero di hit R

[8] per un insieme di N elementi, il numero di coppie in cui non conta l'ordine è $\frac{N(N-1)}{2} = \mathcal{O}(N^2)$.
[9] si assume che tutte le particelle si allontanino dalla zona di interazione.
[10] per la definizione di "efficienza negli hit" far riferimento alla sezione 4.4.6

della traccia più corta, allora la traccia più corta, oppure quella con il χ^2 peggiore, è eliminata. Lo pseudocodice che descrive l'algoritmo è (N rappresenta il numero di tracce):

```
1  per i da 1 ad N−1:
2    per j da i+1 ad N:
3      hitInComune = 0;
4      per ogni hit phi h1 della traccia t[i]:
5        per ogni hit phi h2 della traccia t[j]:
6          se (h1 == h2) hitInComune = hitInComune + 1;
7      minimaLunghezza = min( numeroDiHitR(t[i]), numeroDiHitR(t[j]) )
8      se (hitInComune > 0.7 ∗ minimaLunghezza ):
9        se ( numeroDiHitR(t[i]) > numeroDiHitR(t[j]) ): marca t[j] come non valida;
10       altrimenti: marca t[i] come non valida;
11     altrimenti:
12       se ( Chi2(t[i]) < Chi2(t[j]) ): marca t[j] come non valida;
13       altrimenti: marca t[i] come non valida;
```

Il numero delle tracce candidate cala di un fattore 6-7 dopo entrambe le selezioni compiute da delQuadrupletsClones e delRZTracksClones.

7.2.7 Il kernel makeSpaceTracks

La versione parallela di makeSpaceTracks è molto simile alla funzione originale; una modifica apportata è la rimozione del tagging durante la ricerca delle triplette nei sensori ϕ: il tagging non è compatibile con la parallelizzazione per tracce, per la stessa ragione descritta nel caso della ricerca della delle quadruplette. Un'altra modifica apportata è la rimozione della chiamata ricorsiva nel caso di tracce in overlap che non sono state promosse a tracce spaziali; si prevede di implementare un kernel separato che analizzi questo caso speciale. La funzione makeSpaceTracks utilizza come indici su cui parallelizzare l'evento e la traccia RZ. Si è scelto di creare blocchi da 128 thread che analizzano un gruppo di 128 tracce. Nel codice il kernel è lanciato nel seguente modo:

```
1  grid = dim3( h_nEvents, (MAX_RZ_TRACKS+127)/128); // (event, RZ−track tile)
2  block = dim3( 128, 1 );       // (RZ−tracks/tile, None)
3  makeSpaceTracks <<<grid, block>>> (d_geometry, d_PhiHits, d_RZtracks, d_VeloTracks);
```

In questo segmento di codice viene definita una grid bidimensionale di dimensioni[11] h_nEvents × ((MAX_RZ_TRACKS+127) / 128) e un blocco di thread unidimensionale da 128 thread. Il primo blocco relativo ad un determinato evento analizza le tracce 0-127, il secondo le tracce 128-255, etc. Poiché il numero massimo impostato di tracce RZ in ingresso è molto grande

[11] il "+127" serve per tenere conto dei casi in cui MAX_RZ_TRACKS non è un multiplo di 128.

(3200, configurabile), vengono creati $((3200 + 127)/128) = 25$ blocchi per ogni evento, tuttavia la maggior parte degli eventi contiene meno di 128 tracce, pertanto spesso viene di fatto sfruttato soltanto il primo blocco e solamente in modo parziale (vedi sezione 6.4.3.4). Una soluzione alternativa che permette di aumentare il numero di thread attivi in un blocco è quella di creare un'unica lista di tracce, per poi processarle tutte assieme, definendo una grid e un blocco unidimensionali. In questo modo però si perde la possibilità di sfruttare la memoria shared per memorizzare informazioni comuni a tutte le tracce di evento, perché seguendo questa strategia non necessariamente tutte le tracce di un blocco apparterrebbero allo stesso evento.

7.2.8 Il kernel delSpaceClones

La funzione *mergeSpaceClones* del codice originale per CPU, descritta nel capitolo 5, ha il compito di ridurre i cloni di tracce spaziali prodotte dalla funzione makeSpaceTracks. Nel codice per GPU, *mergeSpaceClones* è sostituita da un kernel chiamato *delClones* più semplice. Il kernel *delClones* determina il numero di hit ϕ in comune per ogni coppia di tracce spaziali di un certo evento. Se il numero di hit in comune è maggiore del 70% del numero di hit ϕ della traccia più corta, allora la traccia più corta, oppure quella con il χ^2 peggiore, è eliminata. L'implementazione si può descrivere con il seguente pseudo-codice:

```
1  per i da 1 ad N−1:
2    per j da i+1 ad N:
3      hitInComune = 0;
4      per ogni hit phi h1 della traccia t[i]:
5        per ogni hit phi h2 della traccia t[j]:
6          se ( h1 == h2 ) hitInComune = hitInComune + 1;
7      minimaLunghezza = min( numeroDiHitPhi(t[i]), numeroDiHitPhi(t[j]) )
8      se ( hitInComune > 0.7 * minimaLunghezza ):
9        se ( numeroDiHitPhi(t[i]) > numeroDiHitPhi(t[j]) ): marca t[j] come non valida;
10       altrimenti: marca t[i] come non valida;
11     altrimenti:
12       se ( qFactor(t[i]) < qFactor(t[j]) ): marca t[j] come non valida;
13       altrimenti: marca t[i] come non valida;
```

Rispetto alle funzioni precedenti, il conteggio degli hit in comune riguarda gli hit ϕ e non più quelli R. Il parametro *qFactor*, invece, è il χ^2 calcolato su tutti gli hit, sia R che ϕ.

L'algoritmo appena descritto è piuttosto crudo e si prevede eventualmente in futuro di supportare la *fusione* delle tracce cloni spaziali, come descritta nella sezione 5.1.4. L'esecuzione è parallelizzata "per evento" e "per traccia"; la traccia su cui si parallelizza nel codice per GPU nello pseudocodice corrisponde al primo ciclo *for* (cioè "per i da 1 ad N-1").

8

FastVeloHough

In questo capitolo sarà introdotta la trasformata di Hough bidimensionale e sarà descritto il tentativo di realizzare il tracking RZ su GPU basandosi su questa tecnica, mentre il rimanente tracking spaziale viene eseguito con l'algoritmo originale makeSpaceTracks. A questo fine è usato del codice GPU per il tracking spaziale in grado di riprodurre identicamente l'algoritmo originale per CPU. L'analisi delle prestazioni sarà discussa nel capitolo 9.

8.1 Trasformata di Hough: introduzione

Il metodo della trasformata di Hough tratta il problema del pattern recognition (vedi sezione 4.1) trasformando l'informazione caratteristica di un *hit* nel corrispondente punto nello spazio dei parametri delle curve interpolanti. La trasformata di Hough fu introdotta nel 1959 come metodo per l'analisi automatica delle immagini prodotti da camere a bolle [27], ed è tuttora utilizzata in diverse applicazioni tra le quali la visione computerizzata (vedi ad esempio il software per la visione computerizzata OpenCV [28, 29]). Nella seguente sezione sarà descritta la trasformata di Hough bidimensionale, mentre si è già brevemente accennato alla trasformata di Hough unidimensionale nel caso del "forward tracking", nella sezione 4.3.4.

8.2 Trasformata di Hough bidimensionale

Dato un insieme di punti nello spazio bidimensionale, il metodo della trasformata di Hough permette di trovare i parametri delle curve in grado di interpolare la distribuzione spaziale dei punti, riducendo in questo modo la dimensionalità delle informazioni (*feature extraction*). Nel caso del tracking, un punto nello spazio corrisponde all'informazione spaziale sull'hit, mentre la curva interpolante corrisponde ad una traccia. Le curve interpolanti possono essere di diversa natura a seconda dell'applicazione: tipicamente sono rette oppure traiettorie di particelle cariche in un campo magnetico. In questa sezione ci limiteremo a considerare solamente il caso dell'interpolazione tramite rette.

Le rette interpolanti possono essere parametrizzate con due valori reali (m, q), che sono i coefficienti dell'equazione della retta $y = mx+q$, dove m è il coefficiente angolare e q l'intercetta. Ad ogni hit (x, y) corrisponde un luogo di punti (m, q) nello spazio dei parametri che soddisfano l'equazione. L'equazione del luogo dei punti è dunque $q = -xm + y$ (x e y fissati), cioè il luogo dei punti nello spazio dei parametri (m,q) è anch'esso una retta con coefficiente angolare $-x$ e con intercetta y.

Il metodo della trasformata di Hough consiste nel rappresentare[1], sovrapponendole, le rette nello spazio dei parametri (una per ogni hit). Per poter implementare questa tecnica con un calcolatore è necessario discretizzare lo spazio dei parametri creando una opportuna griglia di punti. Ciascuno dei punti della griglia deve poter *accumulare* un valore (che possiamo chiamare *voto*), ciò permette di stabilire facilmente il numero di rette che passano per un determinato punto della griglia.

I punti dello spazio dei parametri in cui più rette vanno a intersecarsi individua un punto (m, q) che individua una retta interpolante (traccia) candidata. Il procedimento descritto è schematizzato nella figura 8.2.1.

Un'altra parametrizzazione molto usata nell'ambito dell'analisi delle immagini, alternativa a quella appena discussa, è la parametrizzazione polare. Si parametrizza la retta come indicato in figura 8.2.2 (a sinistra); l'equazione corrispondente è:

$$r = x\cos\theta + y\sin\theta.$$

Un esempio di trasformata con parametrizzazione polare è schematizzato in figura 8.2.3.

8.3 Implementazione della trasformata su GPU

Come accennato, è possibile utilizzare la trasformata di Hough bidimensionale per effettuare il tracking RZ del sensore VELO. La parte di tracking spaziale è invece effettuata in tre dimensioni

[1] in informatica, la rappresentazione di enti geometrici (come linee, triangoli) su una griglia di punti è detta *rasterizzazione*.

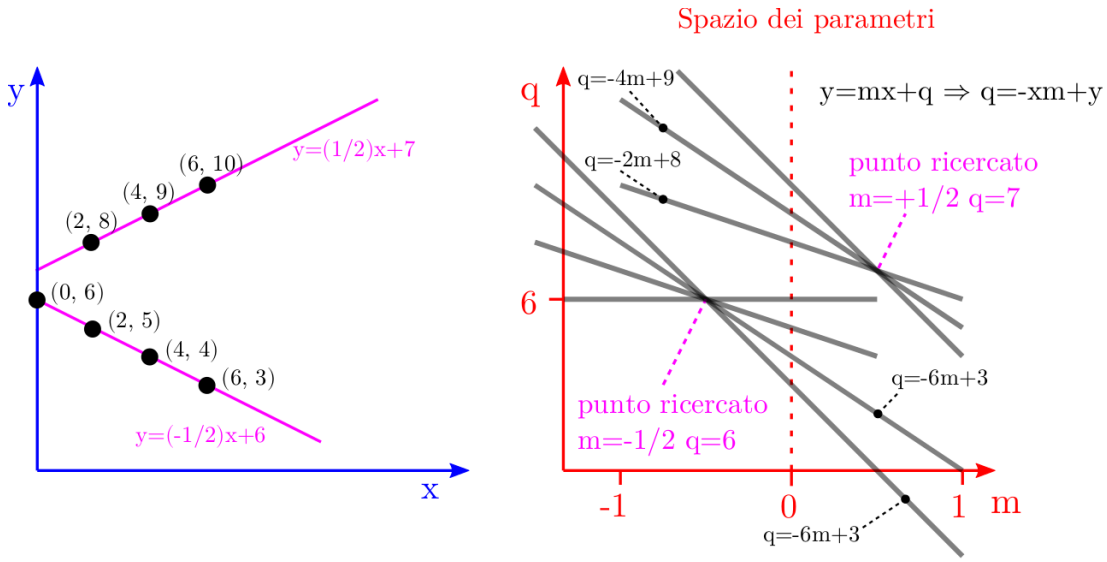

Figura 8.2.1: Illustrazione del concetto della trasformata di Hough

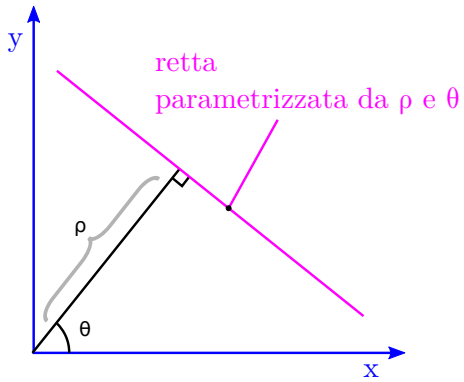

Figura 8.2.2: Parametrizzazione polare.

e quindi non trattabile con la trasformata di Hough bidimensionale, a meno di non effettuare una proiezione.

8.3.1 Rappresentazione delle rette nello spazio dei parametri

Per realizzare il tracking RZ tramite la trasformata di Hough è necessario introdurre una parametrizzazione opportuna nello spazio RZ e stabilire i limiti entro i quali campionare valori nello spazio delle fasi. È stata scelta la seguente parametrizzazione di una retta nello spazio (r,z), come in figura 8.3.1:

$$z = mr + z_0,$$

dove z_0 è l'intercetta sull'asse ed m è il coefficiente angolare.

Nello spazio dei parametri si ha in corrispondenza, fissate le coordinate r, z di un hit:

$$z_0 = -r\,m + z,$$

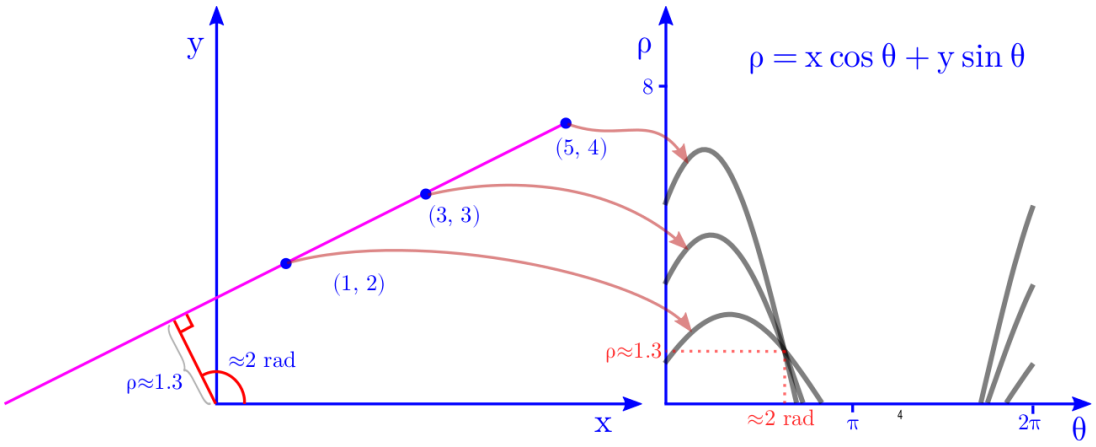

Figura 8.2.3: Trasformata di Hough con parametrizzazione polare, esempio con tre punti allineati.

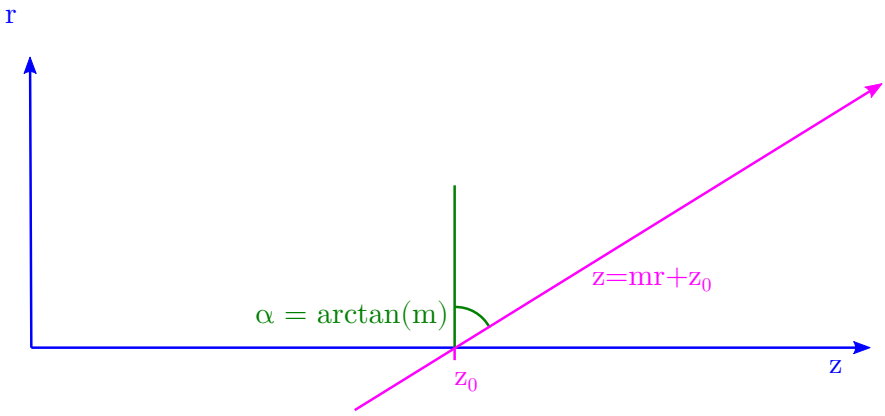

Figura 8.3.1: Parametrizzazione per il tracking RZ tramite trasformata di Hough.

Non è stata scelta la parametrizzazione alternativa ($r = m'z + r_0 \implies r_0 = -m'z + r$) perché la trasformata di Hough campiona con maggiore granularità la regione in cui $|m| \gg 1$, e noi siamo di fatto interessati alle tracce che formano un piccolo angolo rispetto al fascio. L'angolo formato rispetto al fascio è dato da $\theta = 90° - \arctan(m) = \arctan(1/m)$. Si è scelto come limite superiore $m = 80$, che corrisponde ad un angolo di $\xi = \arctan(80) = 12\,\text{mrad}$ rispetto al fascio.

Un ulteriore accorgimento consiste nel tener conto della tolleranza ammessa per l'associazione di un hit R ad una traccia RZ (confrontare sezione 5.1.1), dovuto all'incertezza sperimentale sulla coordinata r (ricordiamo che l'incertezza Δr è proporzionale al prodotto del pitch e dimensione del cluster di strip). L'equazione diventa pertanto:

$$z_0 = -(r \pm \Delta r)\, m + z.$$

È necessario rappresentare la retta opportunamente sfumata attorno al valore centrale, in modo da riprodurre l'incertezza nel coefficiente angolare, come mostrato in figura 8.3.2. Nel codice la

sfumatura è stata approssimata sovrapponendo tre rette distinte per ogni hit invece di una:

$$\begin{aligned} z_0 &= -(r - \Delta r)\, m + z, \\ z_0 &= -(r)\, m + z, \\ z_0 &= -(r + \Delta r)\, m + z. \end{aligned}$$

Il voto è accumulato in un numero intero, nel codice alla retta centrale viene associato un voto pari a 6 mentre le due laterali hanno un peso pari a 3. Se Δr è sufficientemente piccolo viene disegnata un'unica retta con peso 10.

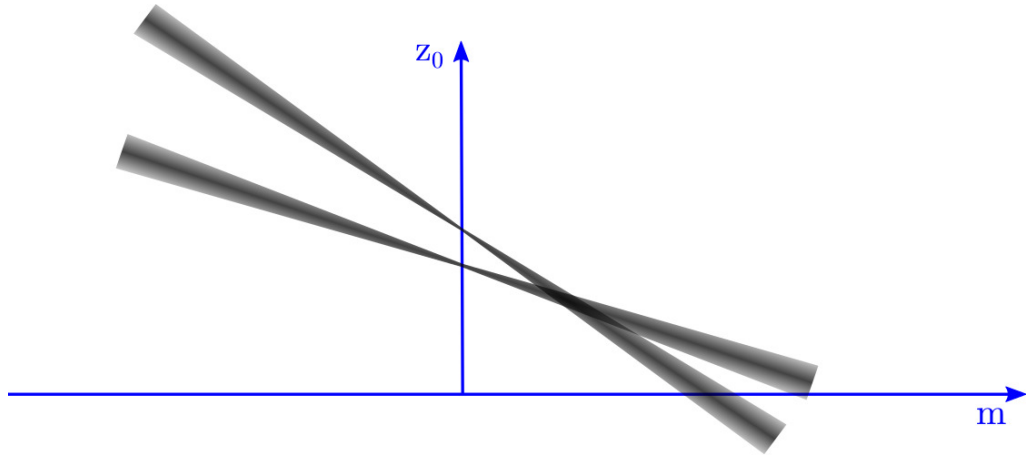

Figura 8.3.2: Allargamento delle rette per riprodurre la tolleranza sulla coordinata r.

Un esempio di rappresentazione delle rette in una griglia di dimensione 1000×600 celle è mostrata in figura 8.3.3. L'intervallo per l'intercetta è $-250 < z_0 < 350$ mentre l'intervallo per il coefficiente angolare $m = \partial z/\partial r$ è $-20 < m < 1$ oppure $1 < m < 80$, corrispondente all'intervallo angolare $177° < \theta < 135°$ oppure $0.72° < \theta < 45°$ rispetto al fascio.

Un'ulteriore necessità è quella di essere in grado di determinare quali hit contribuiscono al voto di ciascun punto dello spazio dei parametri, che è un'informazione necessaria per effettuare il successivo tracking spaziale. Si è scelto di registrare questa informazione allo stesso tempo in cui le rette vengono rappresentate sulla griglia; pertanto il contenitore dello spazio di Hough è stato così strutturato:

```
1 struct HoughSpace {
2   unsigned int value[DIM_Z*DIM_M];
3   unsigned int nhit[DIM_Z*DIM_M];
4   int hits[DIM_Z*DIM_M][MAX_HIT_PER_TRACK];
5 };
```

Dove DIM_Z è il numero di campioni per la coordinata z_0 (600 campioni), DIM_M è il numero di campioni per la coordinata m, e MAX_HIT_PER_TRACK è il numero massimo

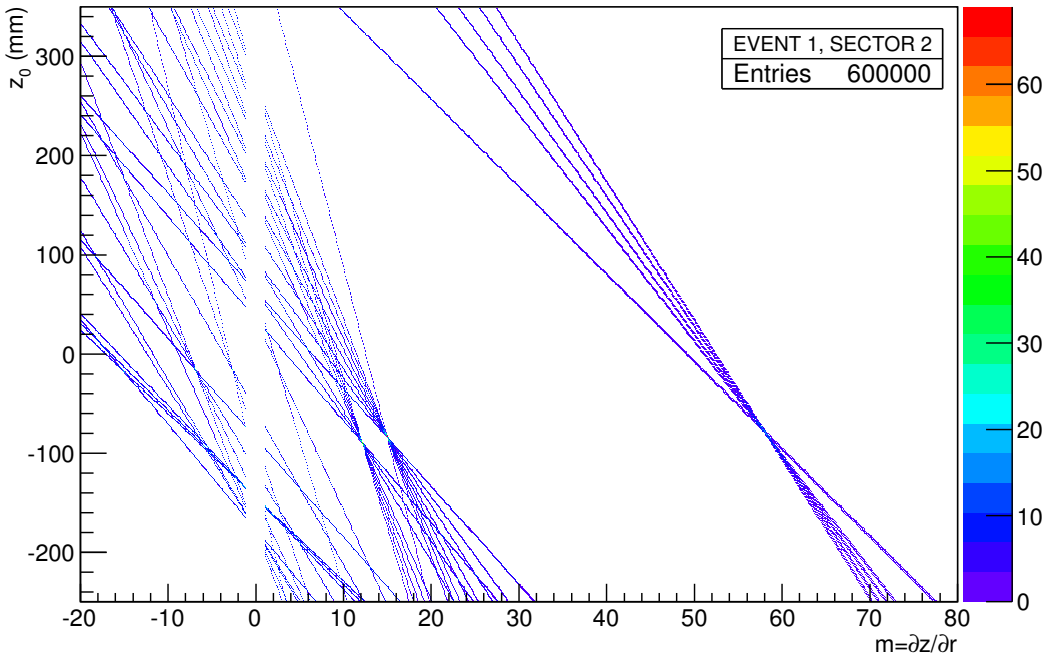

Figura 8.3.3: Trasformata di Hough per la ricerca di tracce RZ realizzata su GPU.

di hit per traccia, impostato a 12. Il voto è salvato come un intero (*value*); per ogni cella sono salvati il numero di hit associati (*nhit*) e la lista di hit (*hits*). Ciascun valore contenuto nel vettore *hits* permette di individuare un particolare hit in un particolare sensore. Nel codice GPU la rappresentazione delle rette è parallelizzata per coordinata m (1000 campioni), per stazioni (21) e per settori (8).

8.3.2 Individuazione dei massimi locali

Al fine di determinare il punto in cui più rette vanno ad intersecarsi, è necessario campionare la matrice di punti e determinare le zone in cui il valore accumulato è massimo. Vanno inoltre applicate opportune condizioni al fine di ridurre la possibilità di falsi positivi. Il seguente pseudocodice rappresenta la soluzione realizzata nel codice per GPU:

```
1 per ogni punto p della matrice:
2   se il voto è maggiore di VOTE_THRESHOLD:
3     se il numero di hit associati a p è maggiore o eguale a HIT_THRESHOLD:
4       se ognuna delle otto celle vicine ha un voto inferiore:
5         salva il punto p come candidato rappresentativo di una traccia.
```

Il criterio usato per le celle vicine (vedi figura 8.3.4) permette di selezionare massimi locali nella griglia dei voti. Si è trovato che nel nostro caso i valori che massimizzano l'efficienza di

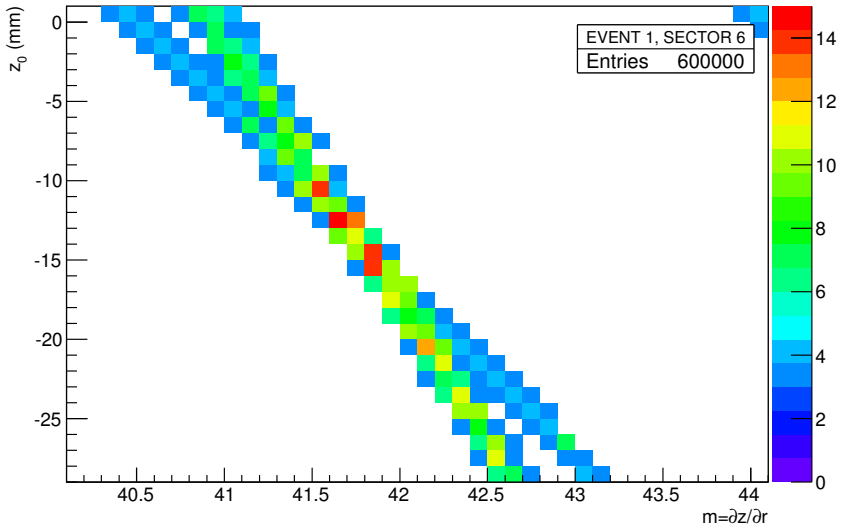

Figura 8.3.5: Cloni generati dal metodo della trasformata di Hough.

tracking e minimizzano il ghost rate sono VOTE_THRESHOLD = 13 e HIT_THRESHOLD = 4 hit (si richiede che la traccia sia una quadrupletta).

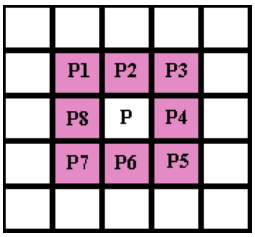

Figura 8.3.4: Criterio dei pixel vicini.

8.3.3 Rimozione dei cloni

Purtroppo le condizioni imposte finora non sono sufficienti ad evitare la generazione di cloni. Come esempio, possiamo analizzare la figura 8.3.5, in cui si possono distinguere facilmente più elementi della griglia che soddisfano le condizioni di massimo locale, mentre ci aspetteremmo di trovare un unico massimo locale. Il numero medio di cloni generato con le impostazioni descritte è piuttosto alto (sono prodotti all'incirca 4 massimi locali per ogni massimo "vero").

È stato necessario includere una funzione di rimozione cloni del tutto simile a quella usata in FastVeloGPU per le tracce RZ (vedi sezione 7.2.6). L'output finale consiste in tracce RZ; ciascuna traccia contiene un insieme di hit dei sensori R che costituisce l'input per la successiva fase di tracking spaziale.

9

Confronto delle prestazioni

In questo capitolo saranno confrontate le prestazioni del codice per GPU (FastVeloGPU e FastVeloHough) con quelle del codice per CPU originale, sia in termini dei valori di merito (efficienza, ghost rate, clone rate), introdotti nella sezione 4.4, sia in termini del tempo di esecuzione. Nella sezione 9.4 saranno descritte le prestazioni attese da server di calcolo multi-core. Nell'ultima sezione saranno discussi eventuali miglioramenti da applicare al codice GPU per incrementarne le prestazioni.

9.1 Campioni di dati

I campioni di dati sono stati scelti in modo da riprodurre condizioni realistiche per il trigger, sia per quanto riguarda il run del 2012, che per il futuro run del 2015, caratterizzato da alto numero di interazioni per bunch crossing. Nel caso del 2015, è stato scelto un campione di dati con decadimenti b-inclusivi. Nella tabella 9.1 sono riassunte le caratteristiche dei campioni utilizzati per la misura delle prestazioni.

Decadimento	Condizioni	Tipo	Numero eventi
$B_s \to \phi\phi$	2012, $\nu = 2.5$	Monte Carlo	1024
dati no-bias	2012, $\mu = 1.6$	Dati reali	1024
b-inclusivi	2015, $\nu = 4.8$	Monte Carlo	1024

Tabella 9.1: Campioni di dati usati per il confronto delle prestazioni.

9.2 Prestazioni di FastVeloGPU

In questa sezione saranno analizzate le prestazioni del codice FastVeloGPU in termini di figure di merito e in termini temporali.

Per misurare le efficienze sui risultati ottenuti dal nostro codice per GPU abbiamo utilizzato un codice che permette il calcolo delle efficienze, basato sul framework Gaudi e in particolare su il programma di ricostruzione BRUNEL che è in grado, dato un campione di dati simulati in formato DIGI, di valutare vari parametri degli algoritmi di tracking tra i quali FastVelo. I risultati ottenuti per la versione in sviluppo sono riportati[1] nella tabella 9.2. In figura 9.2.1 è

Parametro	FastVeloGPU	FastVelo
Ghost Rate	10.0(1) %	7.3(1) %
Efficienza sulle tracce lunghe	86.5(2) %	88.8(2) %
Efficienza tracce lunghe $>$ 5 GeV	89.5(2) %	91.5(2) %
Efficienza tracce B-daughter lunghe	87.3(5) %	89.4(4) %
Efficienza tracce B-daughter lunghe $>$ 5 GeV	89.4(5) %	91.9(4) %
Efficienza tracce B-daughter lunghe e buone	90.6(8) %	93.4(7) %
Efficienza tracce B-daughter lunghe e buone $>$ 5 GeV	90.7(9) %	93.6(7) %
Efficienza su tracce lunghe di Kaoni o Λ	67(1) %	68(1) %
Efficienza su tracce lunghe di Kaoni o $\Lambda > 5$ GeV	73(2) %	74(2) %
Efficienza su tracce lunghe con grande IP	85(2) %	88(2) %
Efficienza su tracce lunghe con grande IP, p $>$ 5 GeV	85(3) %	90(2) %
Frequenza dei cloni su tutte le tracce lunghe	0.2 %	0.2 %
Purezza degli hit su tutte le tracce lunghe	99.5 %	99.5 %
Efficienza negli hit su tutte le tracce lunghe	96.6 %	95.9%
Throughput[2]	2960 $\pm$ 8 eventi/s	1130 $\pm$ 30 eventi/s

Tabella 9.2: Prestazioni FastVelo HLT1 e FastVeloGPU a confronto; campione scelto: 1024 eventi $B_s \to \phi\phi$ con $\nu = 2.5$, $\sqrt{s} = 8$ TeV.

[1] gli errori sulle efficienze sono stati stimati con l'espressione relativa all'errore della distribuzione binomiale $\sigma = \sqrt{\frac{\epsilon(1-\epsilon)}{n}}$ dove $\epsilon = t/n$, t è il numero di tracce selezionate, mentre n è il numero di tracce totali.

mostrata la efficienza sulle tracce lunghe in funzione del modulo del momento, mentre in figura 9.2.2 è riportato un confronto della risoluzione sul parametro di impatto in funzione dell'inverso del momento trasverso. Le prestazioni CPU e GPU risultano del tutto compatibili.

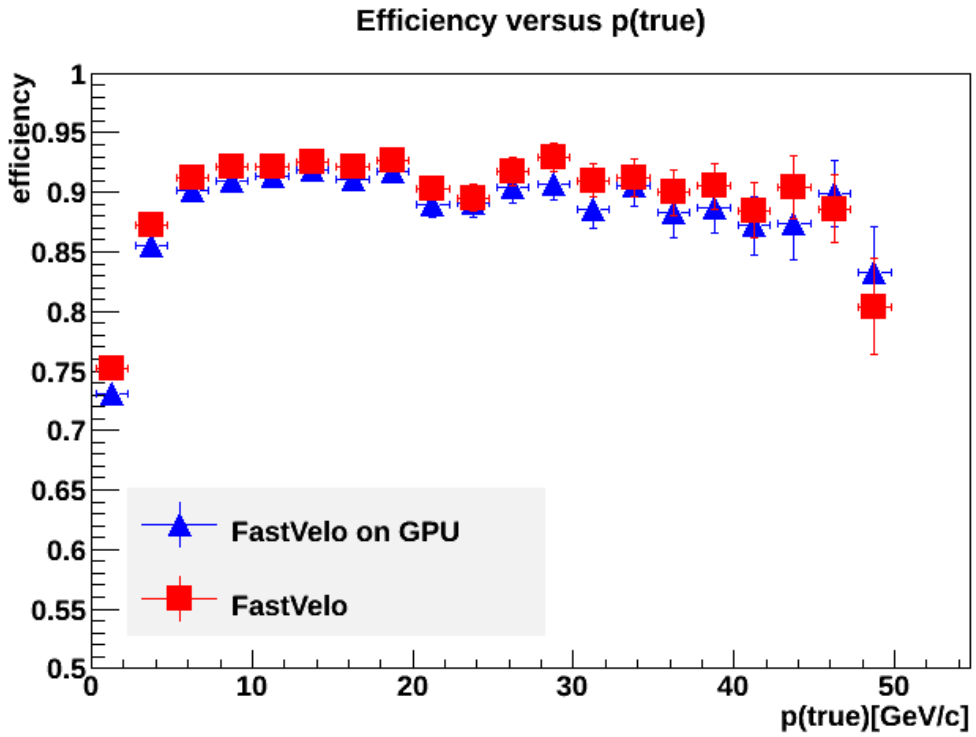

Figura 9.2.1: Confronto dell'efficienza rispetto al momento della particella, tra FastVeloGPU e FastVelo.

Per quanto riguarda le prestazioni in termini temporali il parametro più significativo consiste nel *throughput*, ossia la quantità di dati processati per unità di tempo; per il trigger HLT di LHCb la *latenza* non è rilevante, per cui quest'ultima non sarà discussa. Possiamo valutare il throughput in termini di eventi processati al secondo. Le caratteristiche dell'hardware usato per effettuare il confronto tra CPU e GPU sono riportate nella tabella 9.3, mentre i campioni usati sono quelli della tabella 9.1.

È possibile configurare MOORE o BRUNEL in modo tale da riportare le statistiche dettagliate sui tempi di esecuzione di FastVelo. Si ottiene in questo modo un'idea circa quale sia la componente che impiega più tempo. I grafici in figura 9.2.3 e 9.2.4 riportano il tempo impiegato per eseguire le varie componenti dell'implementazione CPU e GPU.

La legge di Amdahl [61], utile per determinare il massimo speedup[4] di un sistema quando

[2]GPU: NVIDIA GTX Titan; CPU: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 1 thread. Media su 100 campioni.

[3]viene cioè soltanto un thread e pertanto è attivo un solo core

[4]la legge di Amdahl può essere applicata non solo all'ottimizzazione in termini temporali (speedup), ma è applicabile all'ottimizzazione di qualsiasi parametro

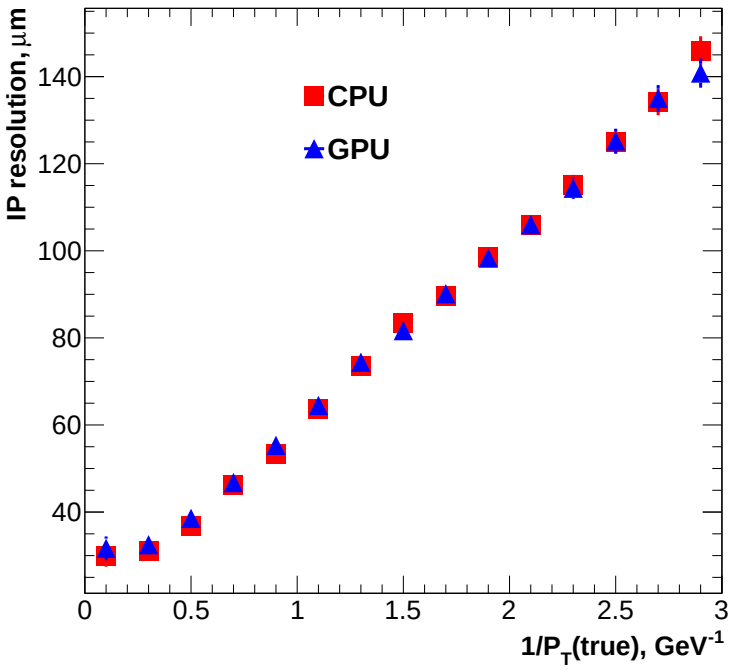

Figura 9.2.2: Confronto tra la risoluzione sui parametri di impatto tra FastVeloGPU e FastVelo.

solo una parte di esso viene ottimizzata (per esempio tramite parallelizzazione) può essere scritta come:

$$\text{Speedup} = \frac{1}{(1-P) + \frac{P}{N}},$$

dove P rappresenta la frazione ottimizzata del programma, mentre N rappresenta lo speedup ottenuto nell'ottimizzazione della sola frazione P. Alla luce di questa considerazione, dal diagramma a torta 9.2.4 risulta chiaro che le parti del programma che vanno ottimizzate per prime sono il kernel MakeSpaceTracks (42% del tempo totale) e le funzioni che si occupano di rimuovere i cloni, mentre l'ottimizzazione del kernel findquadruplets, che occupa il 15% del tempo di tracking, ha minore priorità.

	Configurazione 1	Configurazione 2
Unità di calcolo	Intel® Core™ i7-3770 (single-threaded[3])	NVIDIA GTX TITAN
Frequenza di clock	3.4 GHz	837-876 (Boost) MHz
Memoria principale	32 GiB GDDR3	6 GiB (GDDR5, bus a 384 bit)
Cache	8 MB	64 kB / SMX (L1 + shared memory)
Numero di core	4 (8 con hyperthreading)	14 SMX (Streaming multiprocessors)

Tabella 9.3: Specifiche tecniche dei dispositivi CPU e GPU usati per il confronto.

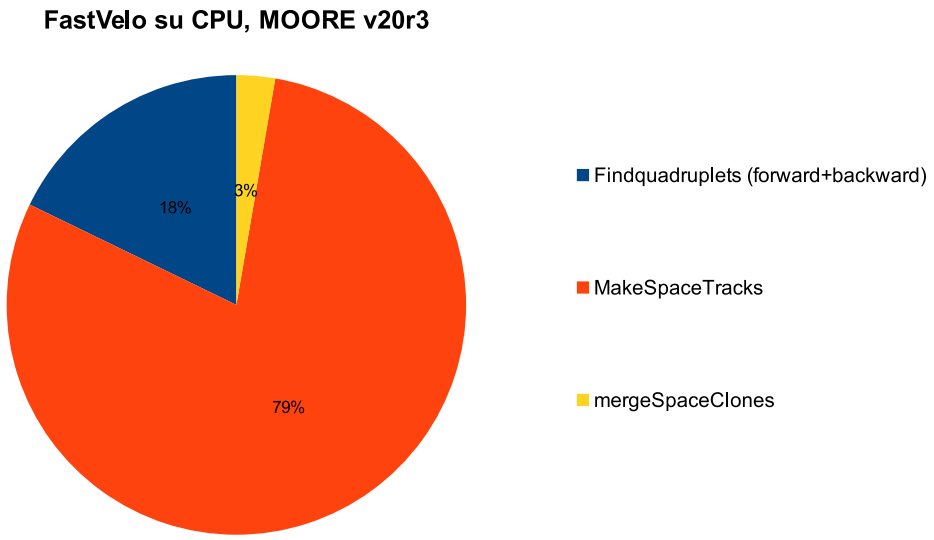

Figura 9.2.3: Distribuzione dei tempi di esecuzione delle componenti di FastVelo in modalità HLT1. Non è incluso il tempo di decodifica.

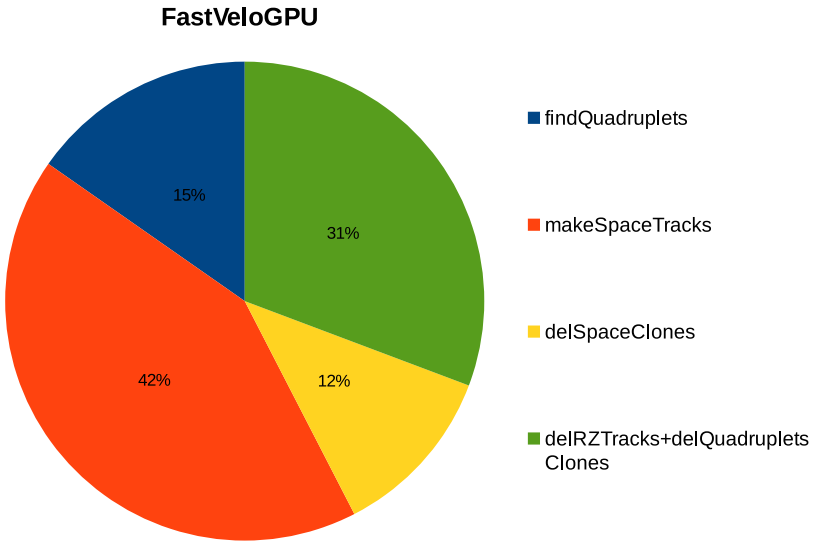

Figura 9.2.4: Distribuzione dei tempi di esecuzione delle componenti di FastVeloGPU. Non è stato incluso il tempo di decodifica (kernel *unpack*).

Nei grafici 9.2.6, 9.2.5, 9.2.7 sono riportate le prestazioni in termini temporali dell'algoritmo originale, lanciato come un'applicazione a singolo thread (cioè viene utilizzato solo un core della CPU) su una macchina Intel i7-3770 @ 3.40 GHz, con quelle dell'algoritmo sviluppato per GPU, per diversi campioni di dati.

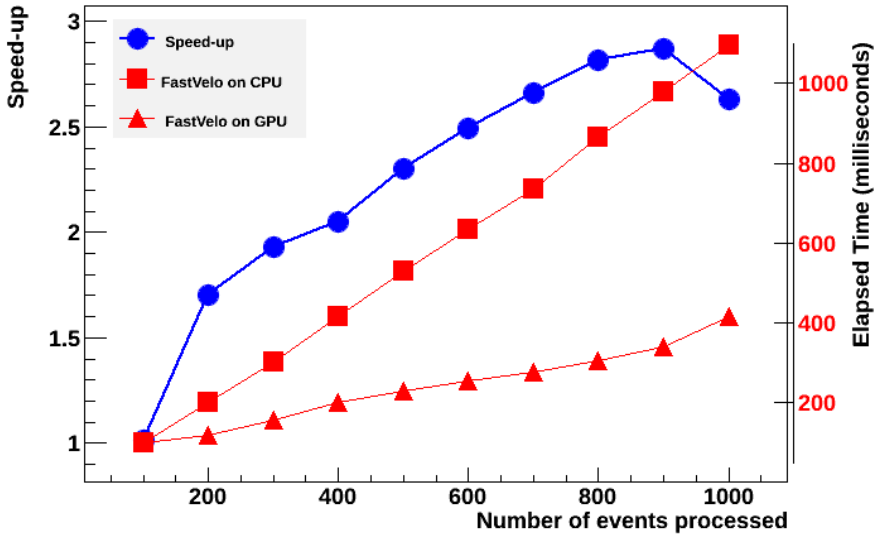

Figura 9.2.5: Prestazioni a confronto, CPU (single thread) e GPU nella versione in sviluppo. Campione di dati Monte Carlo relativi a decadimenti b-inclusivi generati con impostazioni del 2012 ($\nu = 2.5$). Nota: la scala dello speedup non parte da zero; gli errori sono piccoli e le barre di errore non sono visibili. Media su 20 campioni.

Per quanto riguarda il tempo di trasferimento dei dati tra host e device, non incluso nel computo dei tempi riportato nei grafici, con la banda misurata (12 Gbyte/s) questo tempo è dell'ordine della decina di millisecondi per un migliaio di eventi caricati. Tenendo conto della dimensione media dell'evento, 40 kB, si ottiene un tempo di circa 3 microsecondi ad evento, trascurabile rispetto al tempo di esecuzione dell'algoritmo, che si attesta tuttora a circa 400 μs/evento, come si può desumere anche dal grafico in figura 9.2.6. La dimensione dei dati in output è simile a quella dei dati in input, possiamo quindi stimare un tempo totale di trasferimento pari a 6 μs/evento, comunque piccolo rispetto al tempo totale. È inoltre possibile ovviare a questo problema usando l'interfaccia asincrona recentemente introdotta nel linguaggio CUDA: si creano due "canali di esecuzione" indipendenti, chiamati *stream* [63], ed è possibile caricare i dati in memoria in uno stream mentre l'altro stream esegue i calcoli. La strategia è illustrata nella figura 9.2.8.

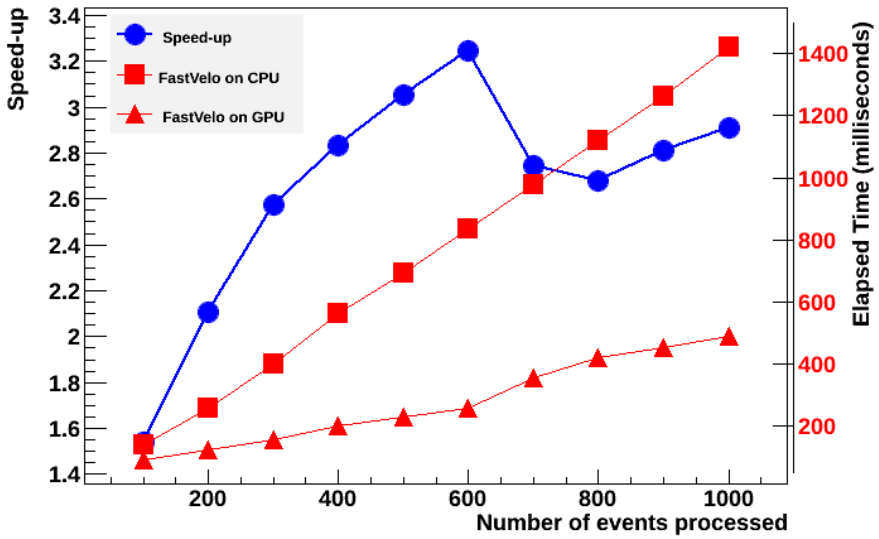

Figura 9.2.6: Prestazioni a confronto, CPU (single thread) e GPU nella versione in sviluppo. Campione no-bias raccolto durante il run del 2012 ($\mu = 1.6$). Nota: la scala dello speedup non parte da zero; gli errori sono piccoli e le barre di errore non sono visibili. Media su 20 campioni.

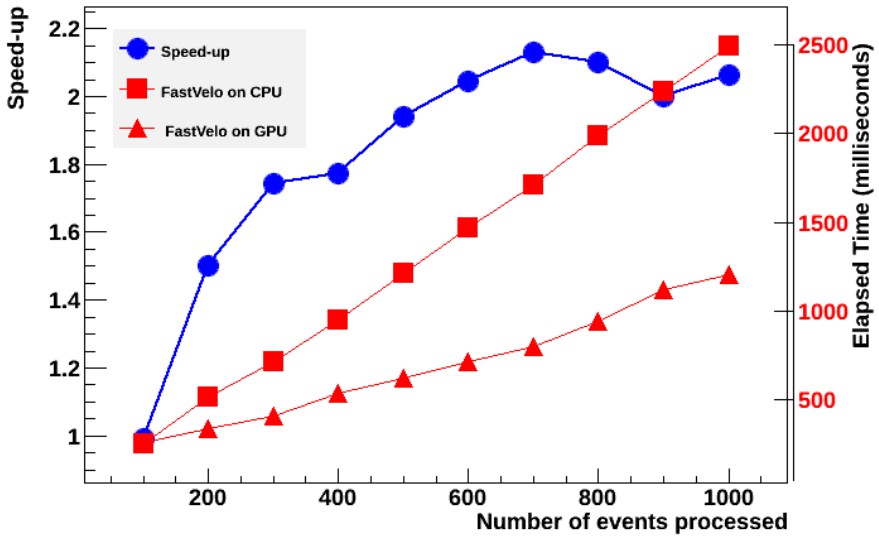

Figura 9.2.7: Prestazioni a confronto, CPU (single thread) e GPU nella versione in sviluppo. Campione di dati Monte Carlo relativi a decadimenti b-inclusivi generati con impostazioni del 2015 ($\nu = 4.8$). Nota: la scala dello speedup non parte da zero; gli errori sono piccoli e le barre di errore non sono visibili. Media su 20 campioni.

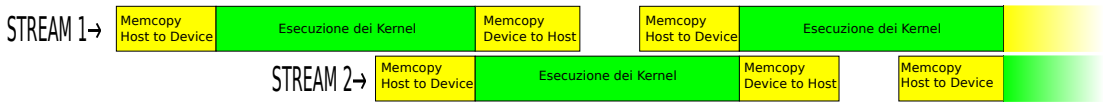

Figura 9.2.8: Interfaccia asincrona con cudaStreams. Il tempo di trasferimento di un migliaio di eventi è dell'ordine della decina di millisecondi, mentre l'esecuzione dei kernel per gli stessi mille eventi è di circa 400 ms.

9.3 Prestazioni di FastVeloHough

In questa sezione si descriveranno le prestazioni dell'algoritmo di Hough applicato al tracking RZ, la cui implementazione è descritta nel capitolo 8. Per ottenere le figure di merito dell'algoritmo (efficienza, ghost rate, ecc.) è necessario implementare anche il tracking spaziale. A questo fine è usato del codice GPU per il tracking spaziale in grado di riprodurre identicamente l'algoritmo originale per CPU, pertanto le figure di merito ottenute dipendono dall'algoritmo di Hough realizzato su GPU e dal codice originale per quanto riguarda il tracking spaziale. La misura delle prestazioni è riportata in tabella 9.4. Si noti che la misura del tempo riportata riguarda soltanto la parte del tracking RZ.

Parametro	FastVeloHough
Ghost Rate	12.6(1) %
Efficienza sulle tracce lunghe	84.1(2) %
Efficienza tracce lunghe > 5 GeV	88.0(2) %
Efficienza tracce B-daughter lunghe	82.4(5) %
Efficienza tracce B-daughter lunghe > 5 GeV	84.8(6) %
Efficienza tracce B-daughter lunghe e buone	87.6(1.0) %
Efficienza tracce B-daughter lunghe e buone > 5 GeV	87.9(1.0) %
Efficienza su tracce lunghe di Kaoni o Λ	65(1) %
Efficienza su tracce lunghe di Kaoni o $\Lambda > 5$ GeV	71(2) %
Efficienza su tracce lunghe con grande IP	81(2) %
Efficienza su tracce lunghe con grande IP, p > 5 GeV	83(3) %
Frequenza dei cloni su tutte le tracce lunghe	0.3%
Purezza degli hit su tutte le tracce lunghe	99.5 %
Efficienza negli hit su tutte le tracce lunghe	96.6 %
Tempo medio di esecuzione (solo tracking RZ, GPU: NVIDIA GTX Titan)	~2.4 ms/evento

Tabella 9.4: Prestazioni FastVeloHough, campione scelto: 1024 eventi $B_s \to \phi\phi$ con $\nu = 2.5$, $\sqrt{s} = 8$ TeV.

Il metodo della trasformata di Hough consente di ottenere prestazioni simili a quelle ottenute dall'algoritmo originale (confrontare tabella 9.2), tuttavia non è in grado di migliorare le prestazioni temporali: per il tracking RZ si misura un tempo di 2.4 ms, di molto superiore al tempo impiegato dall'implementazione di FastVeloGPU, pari a ∼0.15 ms/evento. Quest'ultimo valore è desumibile dal throughput riportato in tabella 9.2 e dal grafico 9.2.4; il tracking RZ è composto da "findQuadruplets" di "delRZclones+delQuadrupletsClones".

Le scarse prestazioni dell'algoritmo di Hough sono legate al fatto che ci aspettiamo che le prestazioni siano buone solamente quando il numero di tracce è molto alto. Nel caso del tracking RZ, tenendo conto delle caratteristiche degli eventi del VELO presentate nella sezione 4.5, poiché le stazioni R sono divise in otto settori da 45° e i settori sono considerati indipendentemente l'uno dall'altro, dato che il numero medio di tracce per evento dell'ordine di un centinaio, si hanno ~ 12.5 tracce per settore, che è un valore troppo piccolo affinché il metodo della trasformata di Hough possa risultare competitivo rispetto all'algoritmo originale. Una soluzione potrebbe essere quella di analizzare gli eventi con poche tracce con l'approccio "locale" e utilizzare invece la trasformata di Hough per gli eventi con grande molteplicità, avendo stabilito una opportuna soglia. Questa scelta è tuttavia discutibile in quanto potrebbe introdurre un bias, ossia favorire la ricostruzione di un certo tipo di tracce a seconda che venga o meno superata la soglia per la molteplicità.

9.4 Confronto con macchine multi-core

Per confrontare le prestazioni di macchina CPU muticore con una GPU, è bene tenere presente che le prestazioni non scalano linearmente con il numero di core che una macchina possiede: questo è dovuto al fatto che alcune risorse sono condivise (ad esempio, il bus dati). Le prestazioni effettive dipendono dall'applicazione specifica.

In questo lavoro di tesi sono state effettuate delle prove con delle macchine dual CPU Xeon E5-2620 con 32 Gbyte di RAM. Ognuna delle CPU ha sei core fisici più sei virtuali (grazie all'*hyperthreading*) pertanto dispone di un totale di 24 core logici. Nella figura 9.4.1 è riportata la misura del throughput (eventi/s) in funzione del numero di core attivi (ad ogni istanza del programma corrisponde un core attivo). Considerando la prestazione ottenuta con un solo core (408 eventi/s) ed estrapolando per il numero di core logici (24) potremmo aspettarci un throughput di $408 \times 24 \simeq 9792$ eventi/s, mentre in pratica abbiamo $\simeq 4840$ eventi/s, cioè circa la metà del valore atteso. Possiamo invece aspettarci che le prestazioni siano direttamente proporzionali al numero di macchine fisicamente distinte presenti in una farm.

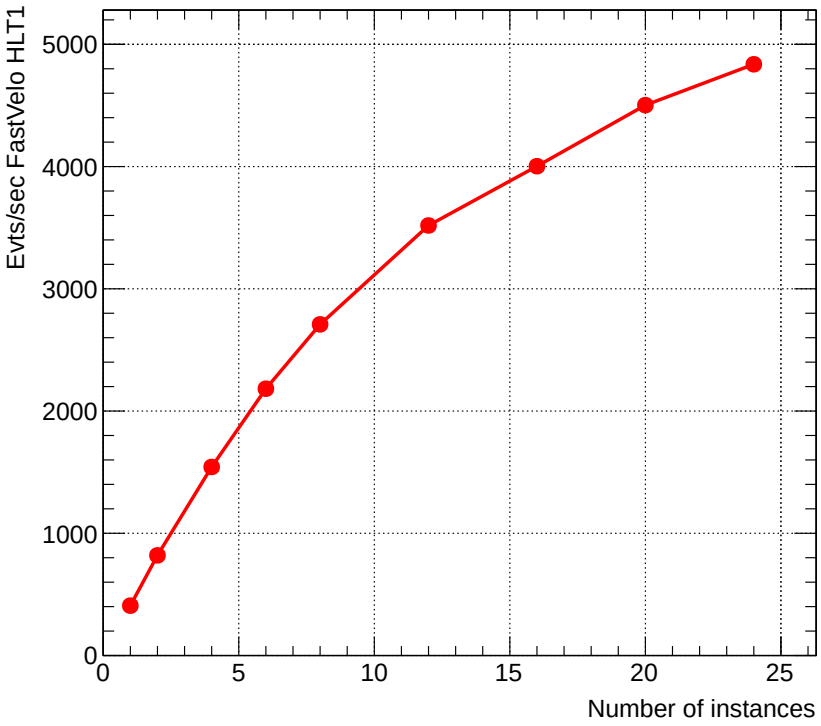

Figura 9.4.1: Prestazioni in termini di eventi processati al secondo in funzione del numero di core utilizzati. Ognuno dei core esegue un'istanza FastVeloHLT1 indipendente. Prova effettuata su un sistema Dual CPU con Xeon E5-2620 (24 core logici totali). Barre di errore non incluse perché non visibili.

9.5 Possibili miglioramenti del codice GPU

Alla luce delle prestazioni descritte, per giustificare il risultato ottenuto (speed-up solo di un fattore 2 o 3 rispetto alla versione CPU) è importante precisare che il codice originale è ricco di istruzioni condizionali, che purtroppo non possono essere implementate efficientemente su GPU, in quanto degradano le prestazioni del codice a causa del fenomeno della *divergenza dei thread* (descritto nella sezione 6.4.1). Un'altra delle caratteristiche negative è il fatto che il numero di tracce per evento e il numero di hit per traccia è variabile: in questo modo alcuni thread terminano la propria esecuzione prima degli altri e rimangono inattivi in attesa che tutti i thread di un blocco terminino la propria esecuzione, che è una condizione non vantaggiosa.

Un punto critico del codice attuale riguarda la rimozione dei cloni, che risulta un problema difficilmente parallelizzabile; nel codice GPU occupa tuttora circa il 40% del tempo totale di tracking, come visibile nel grafico 9.2.4. Date n tracce, si richiede un numero di confronti dell'ordine di $\mathcal{O}(n^2)$. Una possibile miglioria consiste nell'effettuare un ordinamento delle tracce rispetto ad un parametro caratteristico (ad esempio il coefficiente angolare t_x o t_y della retta interpolante) prima di effettuare il confronto traccia per traccia. In questo modo è possibile ridurre il numero di confronti necessari per la ricerca di hit in comune: possiamo assumere che due tracce con t_x e t_y molto diversi non possano avere molti hit R o ϕ in comune.

Un ulteriore miglioramento consiste nell'implementare la ricerca dell'"hit più vicino alla posizione attesa" mediante un efficiente metodo di riduzione parallela [67]. Questo problema ricorre sia nella fase di ricerca di quadruplette nella proiezione RZ sia durante la ricerca di triplette ϕ effettuata nel tracking spaziale.

È ancora da esplorare la possibilità di utilizzare la strategia dell'automa cellulare per il pattern recognition e il filtro di Kalman per il track fitting.

10

Conclusioni

Lo scopo di questo lavoro di tesi è quello di valutare la possibilità di utilizzare sistemi basati su GPU per il trigger di alto livello (HLT) nella *online farm* dell'esperimento LHCb, ed è un progetto nato come preparazione alla fase upgrade di LHCb del 2018: a partire dal 2020 (Run 3) si prevede la possibilità di spostare buona parte delle componenti del trigger su piattaforme parallele, che possono essere sia GPU, sia tecnologie concorrenti quali i coprocessori multicore, come la scheda Xeon Phi della Intel. La prima opportunità di mettere alla prova il sistema basato su GPU descritto in questa tesi si presenterà nel prossimo run, a partire dal 2015: in questa occasione gli algoritmi di tracking potranno essere eseguiti in modalità parassitica[1] durante le fasi di presa dei dati.

Il lavoro svolto ci ha permesso di comprendere i problemi relativi al porting di un programma inizialmente pensato per essere eseguito su CPU. Ci siamo resi conto del fatto che per adattare efficacemente il codice per l'esecuzione su una architettura parallela è stato necessario modificare alcuni algoritmi, in particolare quello relativo alla ricerca delle quadruplette e quello relativo alla ricerca delle tracce spaziali. Per questo motivo non è stato possibile riprodurre esattamente i risultati numerici del codice originale, tuttavia è stato ottenuto un ottimo accordo, come

[1] per parassitica si intende un'analisi ridondante effettuata a scopo dimostrativo, eseguita su una minima parte dei dati in ingresso e utilizzando un sistema di calcolo di dimensioni ridotte.

mostrato nella sezione 9.2.

Il metodo usato per confrontare la bontà di un algoritmo a livello di tracking è quello di misurare figure di merito quali *efficienze, ghost rate* e *clone rate*. I risultati per il tracking del VELO ottenuti con la versione per GPU sono simili a quelli ottenuti con la versione CPU: col campione utilizzato (5120 eventi $B_s \to \phi\phi$ con $\nu = 2.5$) l'efficienza è dell'ordine del 90% e il ghost rate è simile, 7% nel caso della CPU e 10% per il codice GPU.

Riguardo all'ottimizzazione del tempo necessario per effettuare il tracking, il parametro fondamentale è il throughput. Tuttora lo speedup ottenuto dalla GPU è dell'ordine di circa $\times 3$ rispetto ad un singolo core di una CPU. Il confronto è stato effettuato tra una GPU NVIDIA GTX Titan e una CPU Intel i7-3770 @ 3.40 GHz.

Naturalmente, dal punto di vista dell'esperimento, l'ultimo parametro che va considerato sono i costi, sia il costo della macchina in sé, sia eventualmente il consumo energetico, che diventa rilevante a lungo termine.

Dunque il criterio di valutazione è il seguente: a parità di prestazioni dell'algoritmo a livello di tracking, e a parità di costi, quale è la macchina che consente un maggiore throughput in termini di eventi processati per unità di tempo? Tenendo conto degli attuali costi dell'hardware, l'adozione di sistemi basati su GPU risulta decisamente un'interessante alternativa.

Bibliografia

[1] I. I. Bigi, A. I. Sanda, *CP violation* (second edition), Cambridge University Press, ISBN-13 978-0-511-58069-7 (2009)

[2] Franz Mandl, Graham Shaw, *Quantum Field Theory*, 2nd Edition, ISBN: 978-0-471-49683-0 (2010)

[3] P. K. Kabir, Appendix A of: *The CP Puzzle*, Academic Press (1968)

[4] A.D. Sakharov, *Violation of CP Symmetry, C-Asymmetry and Baryon Asymmetry of the Universe*, JETP Lett 5, 24-27 (1967)

[5] R. Aaij et al., *Measurement of* $\sigma(pp \to b\bar{b}X)$ *at* $s = 7$ *TeV in the forward region*, Physics Letters B, 694(3):209–216 (2010)

[6] M.G. van Beuzekom, *Status and prospects of the LHCb vertex locator*, Nuclear Instruments and Methods in Physics Research, Section A, Accelerators Spectrometers Detectors and Associated Equipment 01/2008 (2008)

[7] R. Aaij et al., *The LHCb Trigger and its Performance in 2011*, JINST 8 P04022 (2012)

[8] J. Albrecht et al., *Performance of the LHCb High Level Trigger in 2012*, arXiv:1310.8544 [hep-ex] (2013)

[9] A. Adamatzky, *Game of Life Cellular Automata*, Springer, ISBN 978-1-84996-216-2 (2010)

[10] S.Gorbunov et al., *ALICE HLT High Speed Tracking on GPU,* IEEE Trans. Nucl. Sci., Volume 58 (4) p. 1845 (2011)

[11] T.Kollegger, D.Rohr, *Alice High Level Trigger Tracking*, Proceedings of CNNA2012 (2012)

[12] M. T. Schiller, *Track reconstruction and prompt* K_S^0 *production at the LHCb experiment*, Ph. D. thesis, http://www.physi.uni-heidelberg.de//Publications/diss.pdf (2011)

[13] O. Callot, M. Schiller, *PatSeeding : A Standalone Track Reconstruction*, LHCb note LHCb-2008-042 (2008)

[14] O. Callot, S. Hansmann-Menzemer, *The Forward Tracking: Algorithm and Performance Studies*, Tech. rep. LHCb-2007-015, CERN (2007)

[15] M. Benayoun, O. Callot, *The forward tracking, an optical model method*, Tech. rep. LHCb 2002-008, CERN (2002)

[16] O. Callot, M. Kucharczyk, M. Witek, *VELO-TT track reconstruction.* Tech. rep. LHCb-2007-010 CERN (2007) http://cdsweb.cern.ch/record/1027834?ln=en

[17] M. Needham, J. Van Tilburg, *Performance of the track matching*, Tech. rep. LHCb-2007-020 (2007) http://cdsweb.cern.ch/record/1020304?ln=en

[18] M. Needham, *Performance of the Track Matching*, Tech. rep. LHCb-2007-129, CERN (2007) http://cdsweb.cern.ch/record/1060807?ln=en

[19] Olivier Callot, *FastVelo, a fast and efficient pattern recognition package for the Velo*, LHCb-PUB-2011-001 (2011)

[20] Olivier Callot et al., *Raw-data Format*, EDMS 565851.5 (2005)

[21] LHCb DAQ Group, *Raw-data Transport Format*, EDMS 499933 (2004)

[22] R. E. Kalman, *A New Approach to Linear Filtering and Prediction Problems,* J. Fluids Eng. 82(1), 35-45 (1960)

[23] Doris Eckstein, *VELO Raw Data Format and Strip Numbering*, EDMS 637676 (2005)

[24] J. Beringer et al. (Particle Data Group), *PR D86, 010001 (2012) and 2013 partial update for the 2014 edition* http://pdg.lbl.gov

[25] L. Ristori, *An artificial retina for fast track finding*, Nuclear Instruments and Methods in Physics Research, Section A, Volume 453, Issue 1-2, p. 425-429 (2000)

[26] Hauth, Thomas et al., *Parallel track reconstruction in CMS using the cellular automaton approach*, 20th International Conference on Computing in High Energy and Nuclear Physics (CHEP2013)

[27] P.V.C. Hough, *Machine Analysis of Bubble Chamber Pictures*, Proc. Int. Conf. High Energy Accelerators and Instrumentation, 1959 (1959)

[28] OpenCV, *Hough Line Transform* documentation, pagina web http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html

[29] OpenCV, *Hough Circle Transform* documentation, pagina web http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/hough_circle/hough_circle.html

[30] Guido Haefeli *et al.*, *TELL1 Specification for a common read out board for LHCb*, LHCb 2003-007, IPHE 2003-02 (2003)

[31] LHCb Collaboration, *Muon Identification in the LHCb experiment*, http://arxiv.org/abs/1005.2585

[32] G. Antichi *et al.*, *Time structure analysis of the LHCb DAQ network*, International Conference on Computing in High Energy and Nuclear Physics (CHEP) 2013 (2013) http://hal.archives-ouvertes.fr/docs/00/90/83/83/PDF/chep_proceeding_netfpga_paper.pdf

[33] The LHCb RICH Collaboration, Performance of the LHCb RICH detector at the LHC, Phys. J. C (2013)

[34] Eduardo Picatoste Olloqui, *LHCb Preshower(PS) and Scintillating Pad Detector (SPD)*: commissioning, calibration, and monitoring, J. Phys.: Conference Series 160 012046 (2009)

[35] D. Baumeister *et al.*, *Characterization of the Beetle-1.0 Front End Chip*, LHCb 2001-049 (2001)

[36] Irina Machikhiliyan, *The LHCb electromagnetic calorimeter*, J. Phys.: Conf. Ser. 160 012047 (2009)

[37] LHCb Collaboration, *LHCb VELO (VErtex LOcator): Technical Design Report*, CERN-LHCC-2001-010; LHCb-TDR-4 (2001)

[38] LHCb Collaboration, *LHCb Reoptimized Detector Design and Performance Technical Design Report*, CERN/LHCC 2003-030; LHCb-TDR-9 (2003)

[39] LHCb Collaboration, *LHCb Outer Tracker Technical Design Report*, CERN-LHCC-2001-024; LHCb-TDR-6 (2001)

[40] LHCb Collaboration, *LHCb Inner Tracker Technical Design Report*, CERN-LHCC-2002-029; LHCb-TDR-8 (2002)

[41] LHCb Collaboration, *LHCb RICH Technical Design Report*, CERN-LHCC-2000-037 LHCb-TDR-3 (2000)

[42] LHCb Collaboration, *LHCb calorimeters: Technical Design Report*, CERN-LHCC-2000-036; LHCb-TDR-2 (2000)

[43] LHCb Collaboration, *Muon System Technical Design Report*, CERN-LHCC-2001-010; LHCb-TDR-4 (2001)

[44] LHCb Collaboration, *LHCb Trigger System Technical Design Report*, CERN-LHCC-2003-031; LHCb-TDR-10 (2003)

[45] LHCb Collaboration, *LHCb online system technical design report: Data acquisition and experiment control*, CERN-LHCC-2001-040 (2001)

[46] LHCb Collaboration, A*ddendum to the LHCb online system technical design report*, CERN-LHCC-2005-039, CERN-LHCC-2001-040-Add1 (2005)

[47] LHCb Collaboration, *Letter of Intent for the LHCb Upgrade*, CERN-LHCC-2011-001 (2011)

[48] LHCb Collaboration, *Framework TDR for the LHCb Upgrade*, CERN-LHCC-2012-007 ; LHCb-TDR-12 (2012)

[49] LHCb Collaboration, *LHCb VELO Upgrade Technical Design Report*, CERN-LHCC-2013-021; LHCB-TDR-013 (2013)

[50] LHCb Collaboration, *LHCb Tracker Upgrade Technical Design Report*, CERN-LHCC-2014-001; LHCB-TDR-015 (2014)

[51] LHCb Collaboration, *Trigger and Online Upgrade Technical Design Report*, CERN-LHCC-2014-016; LHCb TDR-16 (2014)

[52] James Stirling, *Parton Luminosity and cross section plots*, http://www.hep.ph.ic.ac.uk/~wstirlin/plots

[53] LHCb Collaboration, $\bar{b}b$ *production angle plots*, web page, http://lhcb.web.cern.ch/lhcb/speakersbureau/html/bb_ProductionAngles.html

[54] L.B.A. Hommels, *The Tracker in the Trigger of LHCb*, Ph. D. thesis, Cambridge Printing, Cambridge ISBN: 978-0-9553948-0-5

[55] Gaudi project home page, http://proj-gaudi.web.cern.ch/proj-gaudi/

[56] Gauss project home page, http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/gauss/

[57] Boole project home page, http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/boole/

[58] Moore project home page, http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/moore/

[59] Brunel project home page, http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/brunel/

[60] Da-vinci project home page, http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/davinci

[61] Gene M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, in Proceedings of the April 18-20 1967 Spring Joint Computer Conference, AFIPS, pages 483-485, New York, NY, USA (1967)

[62] Wen-mei Hwu, David Kirk, *Programming Massively Parallel Processors*: *A Hands-on Approach,* Elsevier Inc. ISBN: 978-0-12-415992-1 (2013)

[63] NVIDIA Corporation, *CUDA C Programming Guide*, http://docs.nvidia.com/cuda/cuda-c-programming-guide

[64] NVIDIA Corporation, *CUDA Occupancy Calculator*, http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

[65] NVIDIA Corporation, *NVIDIA Visual Profiler*, https://developer.nvidia.com/nvidia-visual-profiler

[66] Mark Harris, *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*, blog, http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/ Archiviato su: http://www.webcitation.org/6Qile2Xoa

[67] Justin Luitjens, *Faster Parallel Reductions on Kepler*, blog, http://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/ Archiviato su: http://www.webcitation.org/6Qz9G8J74

[68] Karl Rupp, *CPU, GPU and MIC Hardware Characteristics over Time*, blog, http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/ Archiviato su: http://www.webcitation.org/6QilnTT7P

[69] Yossi Kreinin, *SIMD < SIMT < SMT: parallelism in NVIDIA GPUs,* blog, http://www.yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html Archiviato su: http://www.webcitation.org/6QilWUalh

[70] Sami Rosendahl, *CUDA and OpenCL API comparison*, presentation for T-106.5800 Seminar on GPGPU Programming, Aalto University, Helsinki, Finland, spring 2010 (2010) https://wiki.aalto.fi/download/attachments/40025977/Cuda+and+OpenCL+API+comparison_presented.pdf Archiviato su: http://www.webcitation.org/6QilFUBQD