Uso di GPGPU nell'esperimento LHCb

Stefano Gallorini
Università e INFN Padova

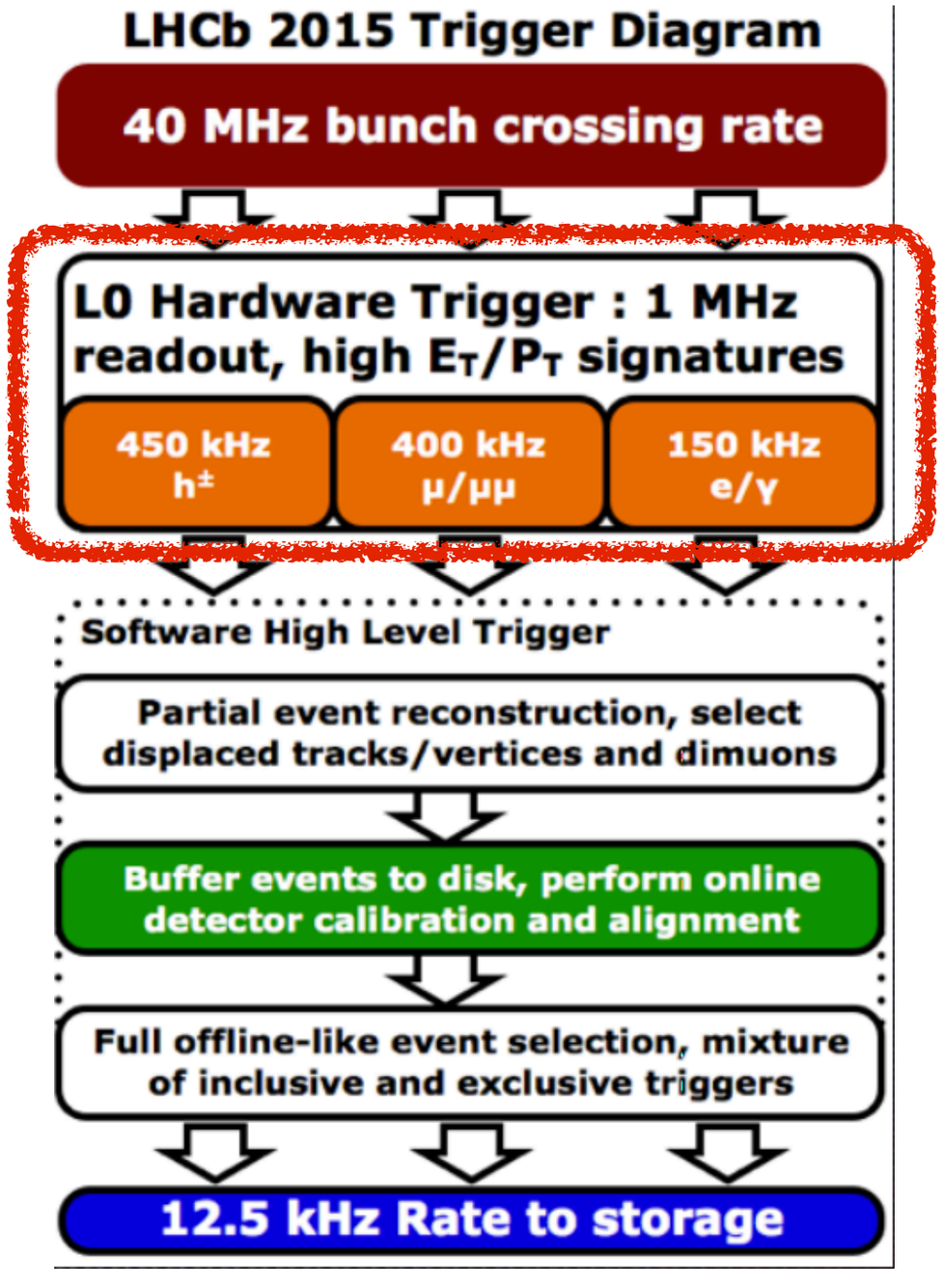
Incontri di Fisica delle Alte Energie - Roma - 10/04/2015

Introduzione

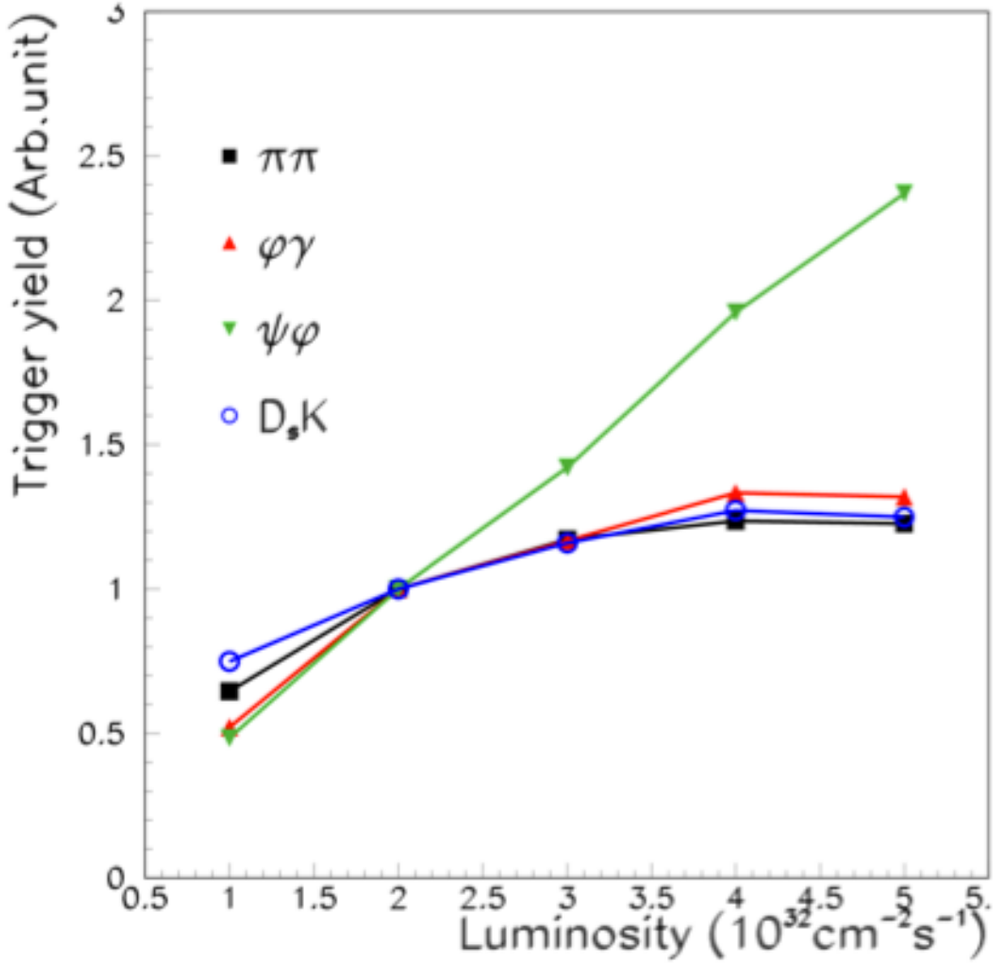
- Con l'aumento di luminosità di LHC previsto per il Run3, è indispensabile un uso più efficiente delle risorse di calcolo per riuscire a processare gli eventi (sia online che offline)
- Architetture many-cores, come ad es. GPGPU e Intel MIC, possono essere una valida alternativa alle CPU:
  - Per selezioni di trigger più raffinate e vicine all'offline
  - Per ottimizzare il rapporto costo/performance della farm di trigger
  - Per mitigare il rischio dovuto a possibli deviazioni dalla legge di Moore

Il trigger di LHCb upgrade (1)

Schema del trigger attuale (Run2)



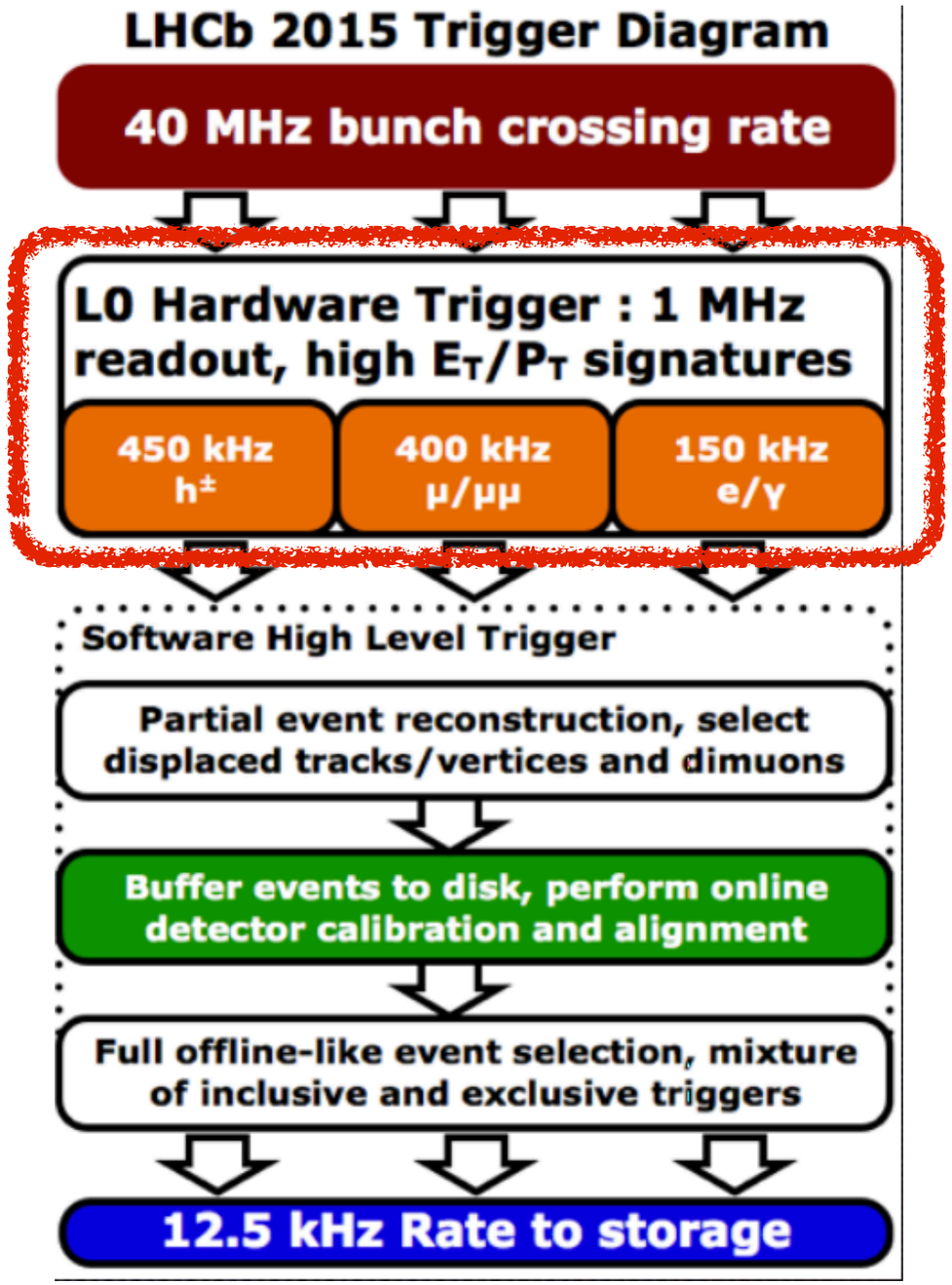
Il max. rate in input all'HLT è limitato dalle schede di DAQ dell'L0 (1MHz)

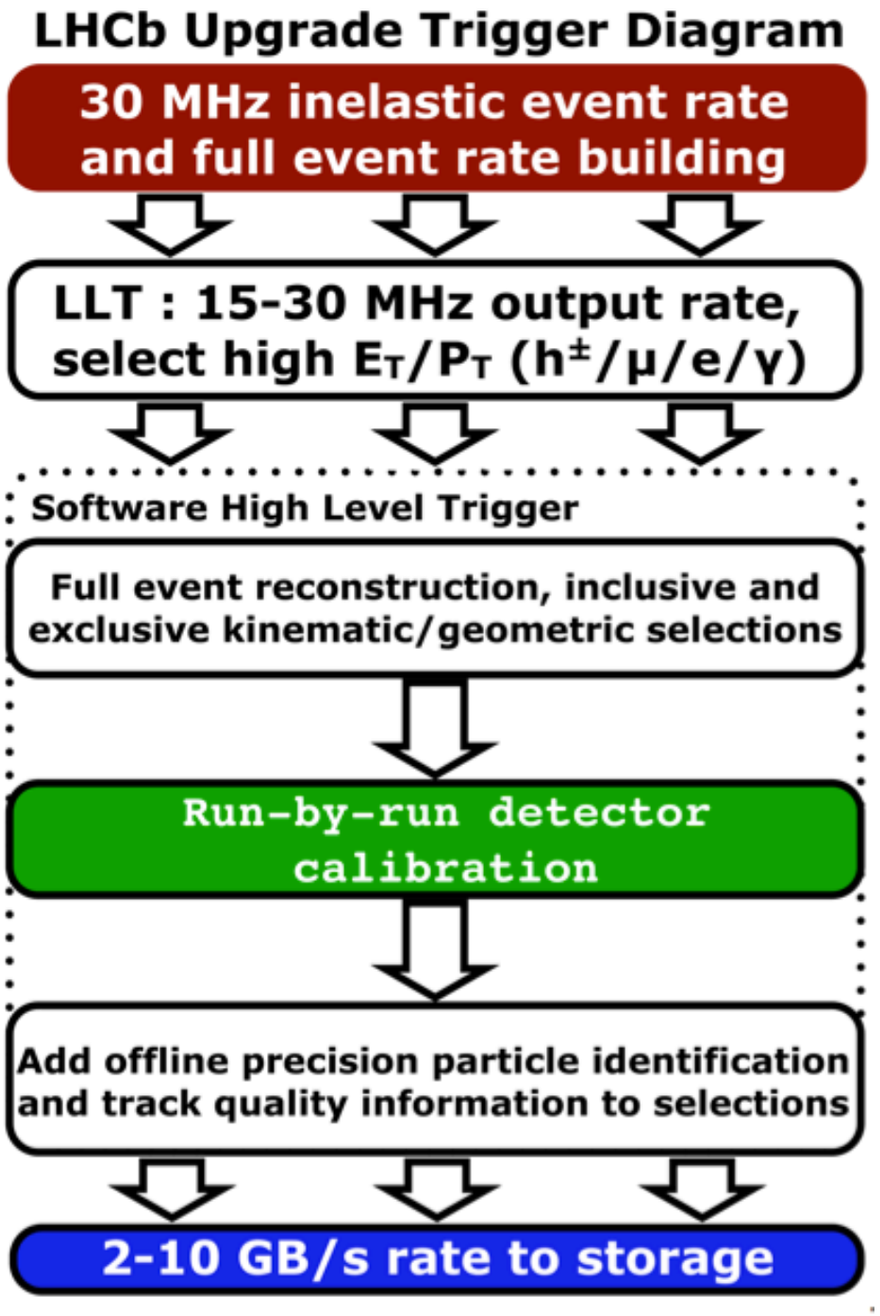


Il trigger di LHCb upgrade (2)

Goal dell'upgrade trigger:

full detector readout a 40MHz e full software trigger



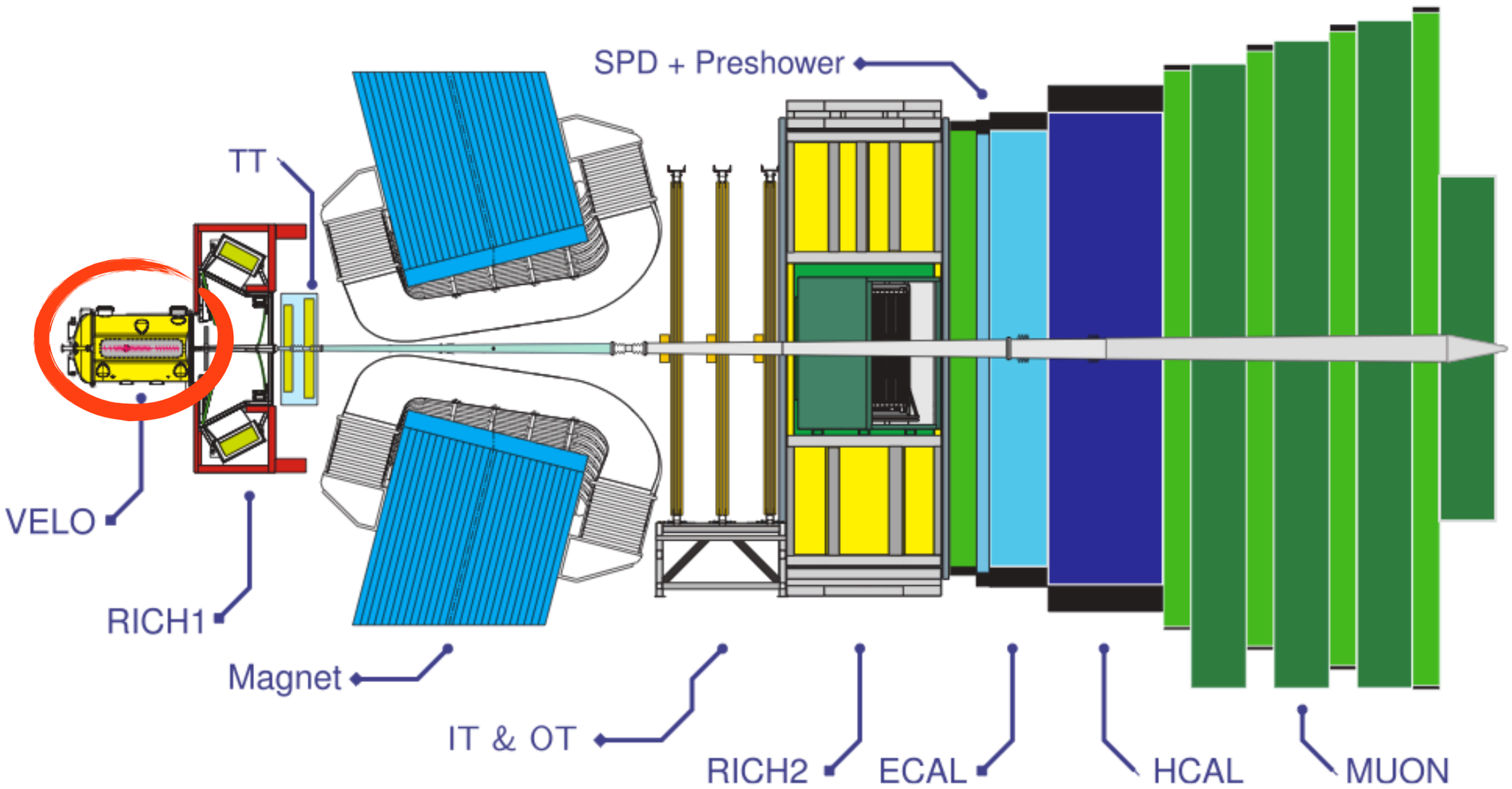


Attività many-cores in LHCb

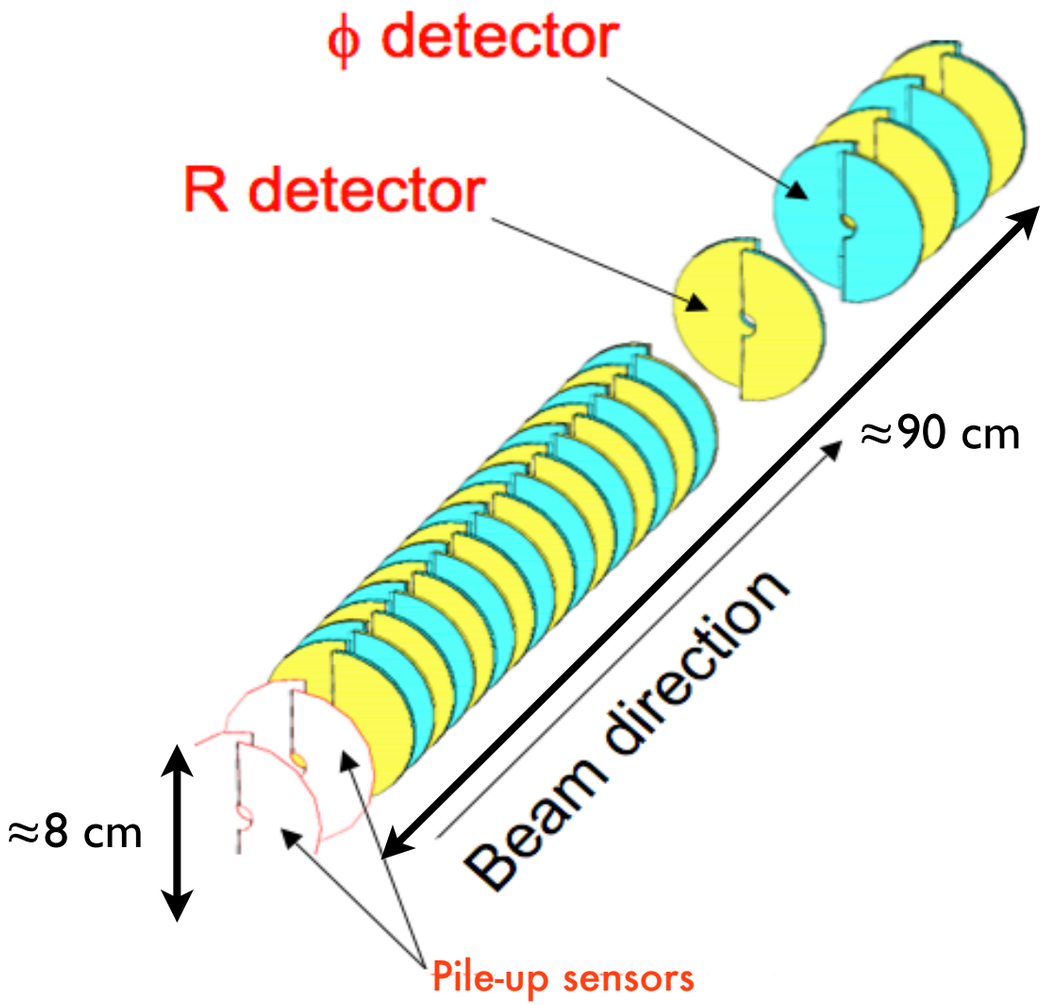
- Goal:
    - Studiare la fattibilità dell'uso di GPU commerciali (o Intel MIC, sistemi eterogenei, ...) nel trigger per l'upgrade di LHCb
    - Per far ciò stiamo sviluppando algoritmi di tracking su GPU per studiare le performances durante il Run2 (modalità parassita)

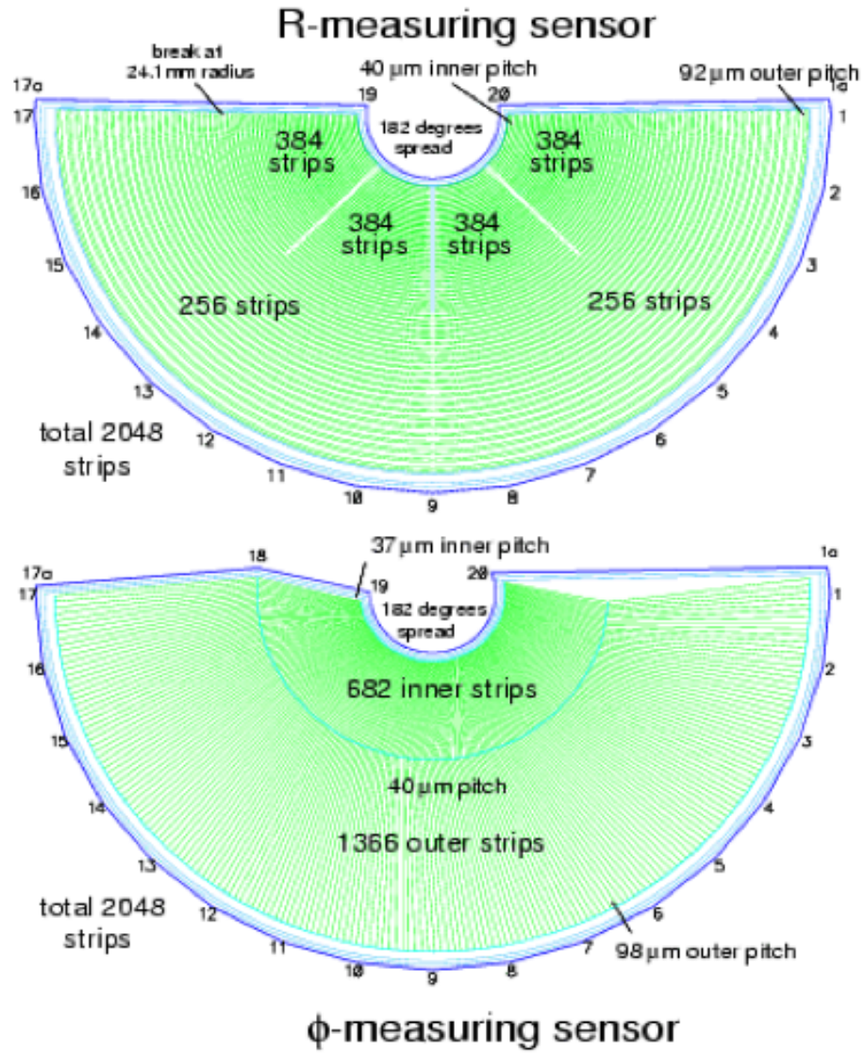
- Due algoritmi in fase di studio:
    1. Ricostuzione delle tracce nel VELO
    2. Ricostruzione delle tracce nelle T-stations

Tracking del VELO su GPU



Tracking del VELO su GPU





- Il VELO è un detector a strips di silicio, situato vicino alla regione di interazione
- R-ϕ layout, 21 stazioni, ognuna con 2 sensori R e 2 sensori ϕ

Tracking del VELO su GPU (1)

- L'algoritmo originale per il tracking del VELO ("FastVelo") è stato adattato per sfruttare al meglio l'architettura delle GPU (GPU NVidia)

- Per fare ciò, sia la logica che le strutture dati dell'algoritmo sono state in parte modificate

- Le performances (efficienze e tempi di esecuzione) sono state confrontate con l'algoritmo originale (single/multiple CPU cores)

Tracking del VELO su GPU (2)

- La ricostruzione delle tracce nel VELO è suddivisa in due parti:

  1. **RZ tracking:** ricerca di tracce nel piano R-Z (solo sensori R)
  2. **Space tracking:** le tracce spaziali sono ricostruite a partire dalle tracce in RZ aggiungendo l'informazione delle hits nei sensori ϕ

- **Strategia di parallelizzazione:**

  1. Processare più eventi in parallelo (ovvio)
  2. Ricerca in parallello delle tracce RZ (seeding + track following)
  3. Space tracking in parallelo per ogni traccia RZ (più dettagli in backup)

Tracking del VELO su GPU (3)

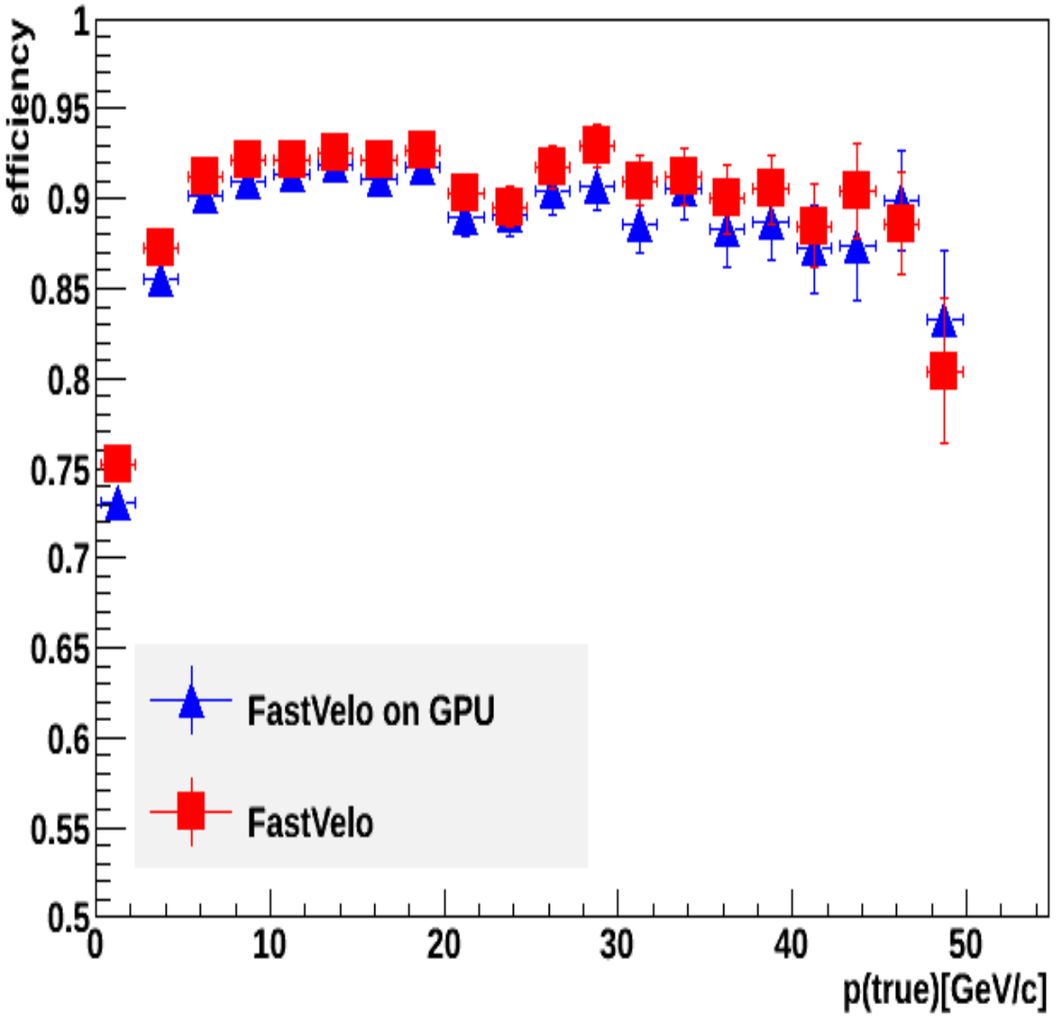
- **Set-up:**
  - GPU: NVidia Titan (14 SMX, 192 CUDA cores/SM, 6GB di memoria)
  - CPU: **single core** (Intel i7, 3.40 GHz, nello stesso PC che ospita la GPU) e **multicore CPU** (Intel Xeon E5-2600, 24 cores w/ Hyper Threading, @CNAF, simile ad un nodo HLT)

- **Risultati** (1000 evt MC $B_s \rightarrow \Phi\Phi$, HLT1 mode):

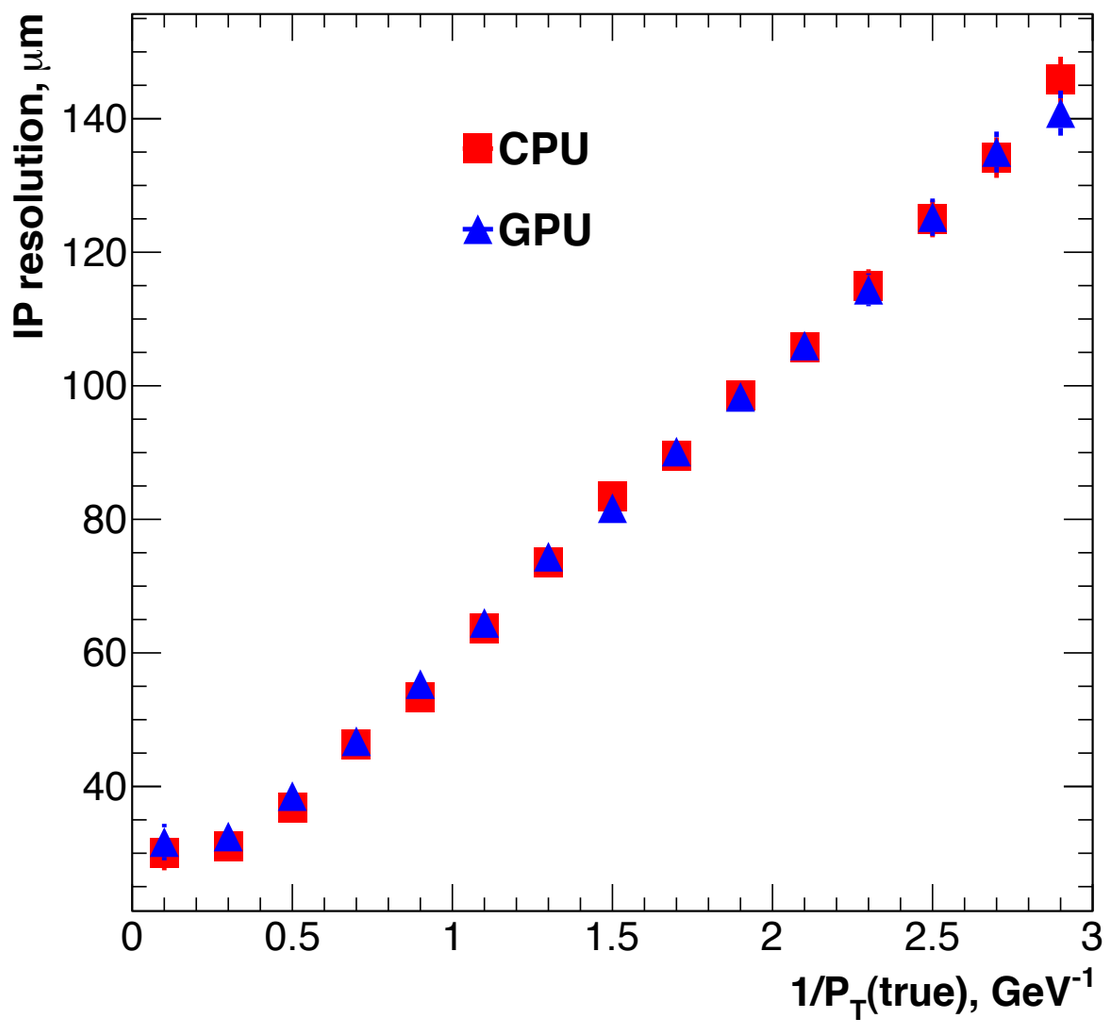
Track category	FastVelo on GPU		FastVelo on CPU	
	Efficiency	Clones	Efficiency	Clones
VELO, all long	86.6%	0.2%	88.8%	0.5%
VELO, long, p > 5 GeV	89.5%	0.1%	91.5%	0.4%
VELO, all long B daughters	87.2%	0.1%	89.4%	0.7%
VELO, long B daughters, p > 5 GeV	89.3%	0.1%	91.8%	0.6%
VELO, ghosts	7.8%		7.3%	

Tracking del VELO su GPU (4)

Efficenza vs P(true)



Risoluzione IP vs $1/P_T$(true)

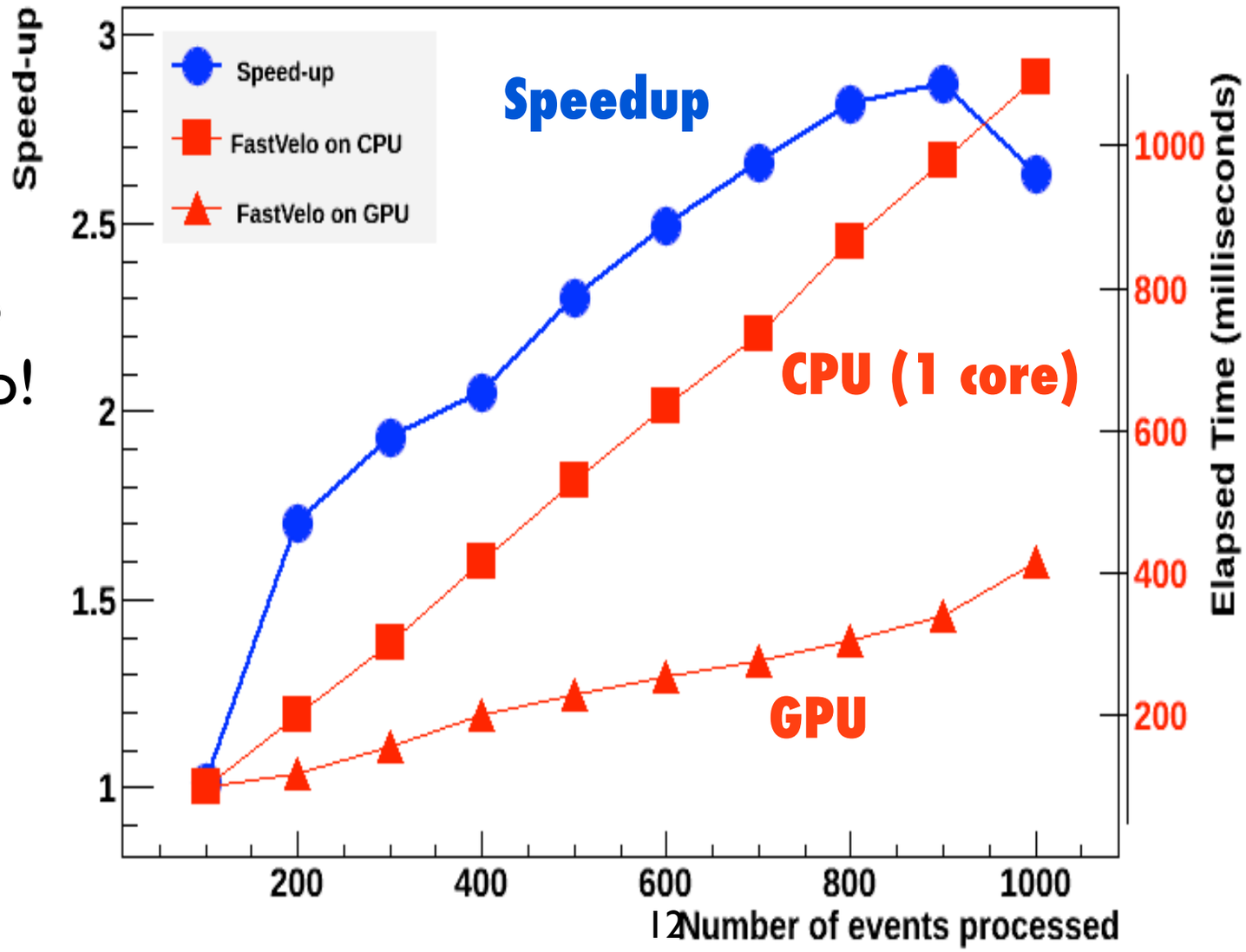


Performances compatibili con l'algoritmo originale (ottimizzato)

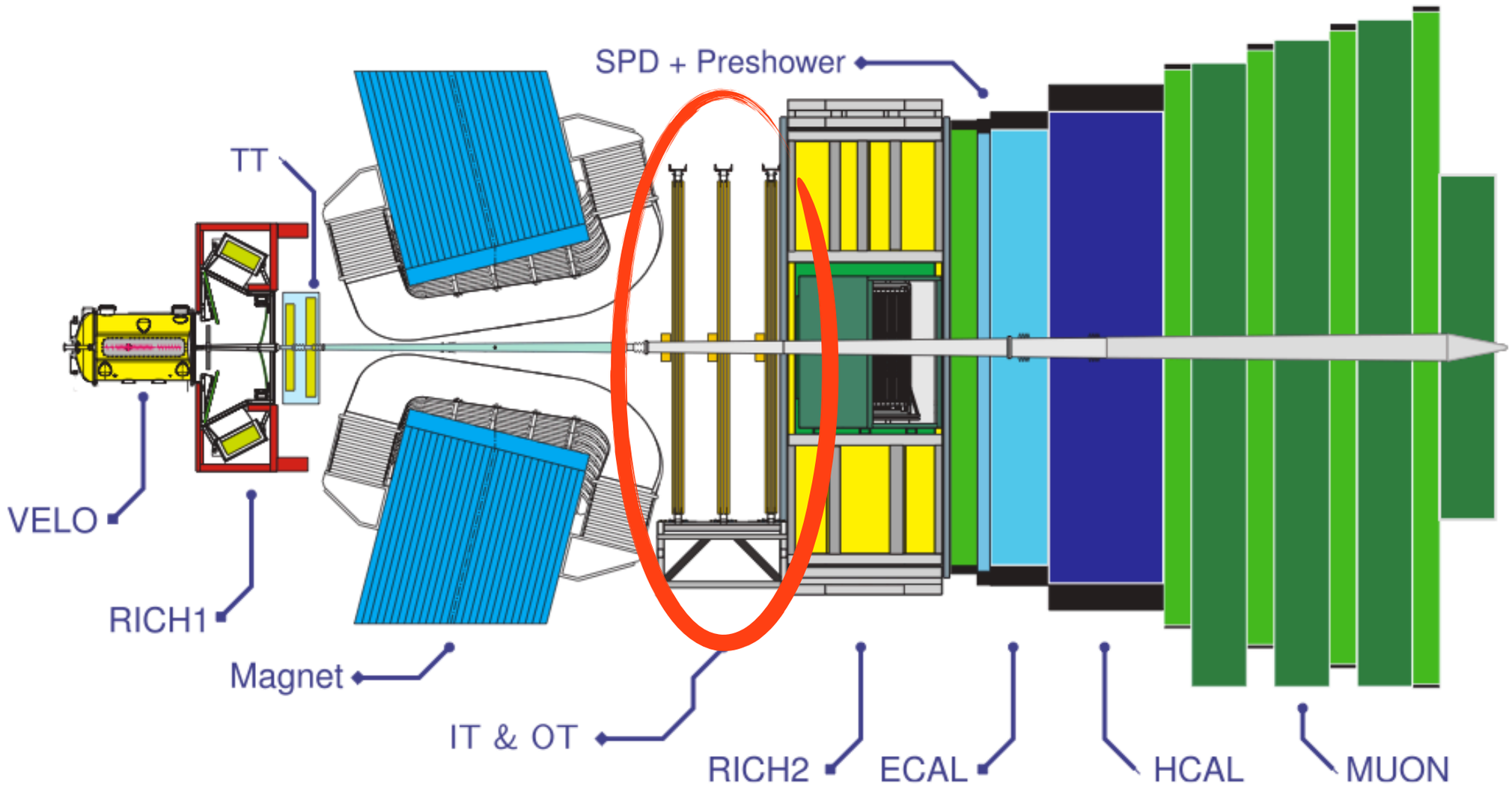
Tracking del VELO su GPU (5)

- Speedup (T_{CPU}/T_{GPU}) **~3x** rispetto ad un single CPU core
- Throughput: **≈5000 evts/sec** (full CPU) vs **≈2600 evts/sec** (GPU)

Trasferimento dati non incluso!

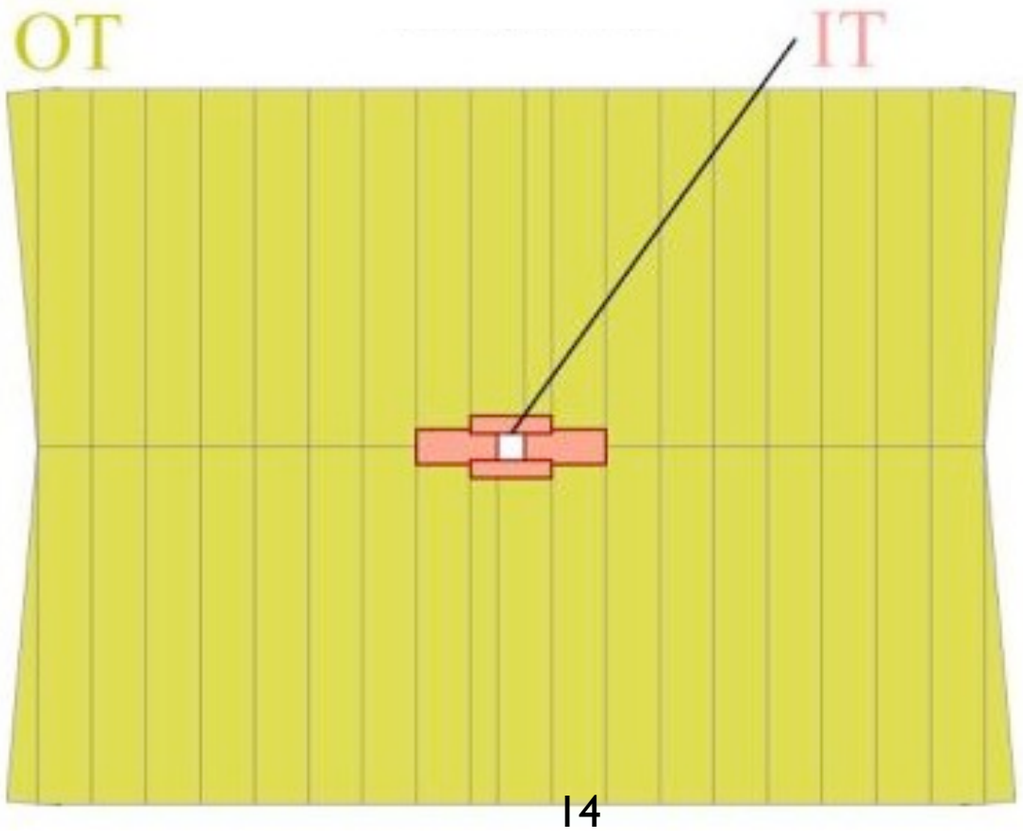


Tracking delle T-stations su GPU



Tracking delle T-stations su GPU

- Le T-stations sono detectors piuttosto complessi:
  - 3 stazioni, 4 layers/stazione (x, u, v, x)
  - Outer Tracker (OT): 2 piani di straw tubes/layer, large occupancy (ambiguità L/R, drift distance, ...)
  - Inner Tracker (IT): detectors di silicio, più semplice rispetto all'OT



Tracking delle T-stations su GPU (1)

- Attualmente, il tracking delle T-stations è responsabile del ~12% del tempo totale dell'HLT2

- Utilizzare il codice esistente in CPU adattandolo a sistemi many-cores è spesso molto complicato (uso estensivo di std library, AoS anzichè SoA, ...)

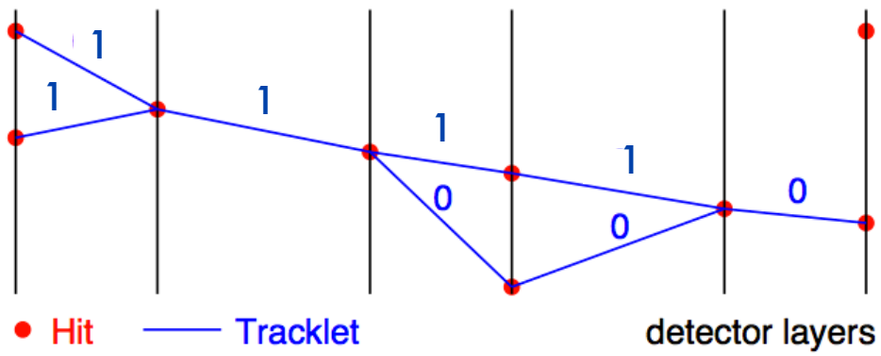
- Per il tracking delle T-stations abbiamo deciso di riscrivere l'algoritmo da zero, in modo da avere un codice "embrassingly parallel"

- L'approccio è ispirato all' Automa Cellulare

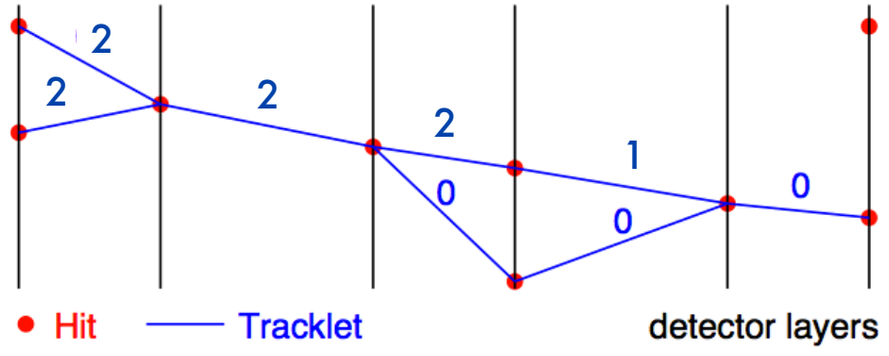
Tracking delle T-stations su GPU (2)

- L' Automa Cellulare (AC) procede in 3 passi:
  - Generazione di tutti i segmenti tra layers adiacenti ("le cellule"). Ogni segmento possiede un contatore ("stato della cellula") inizialmente inizializzato a zero
  - Ricerca dei segmenti vicini (segmenti adiacenti compatibili con il modello di propagazione della traccia)
  - Evoluzione (in step temporali). Il contatore (c1) di ogni segmento è incrementato di +1 se uno dei suoi vicini ha il contatore c2 ≥ c1

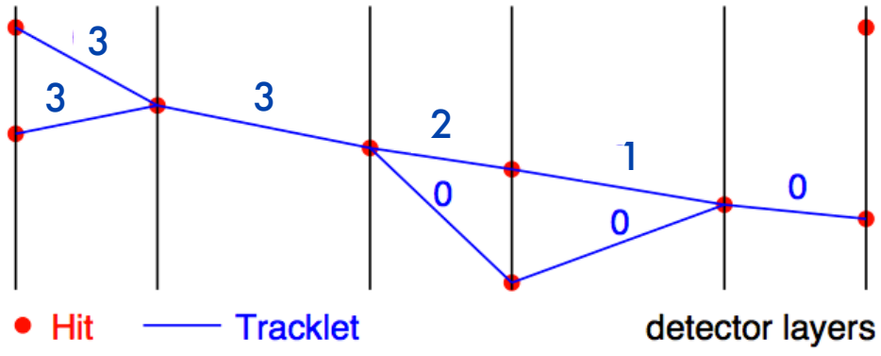
Step1



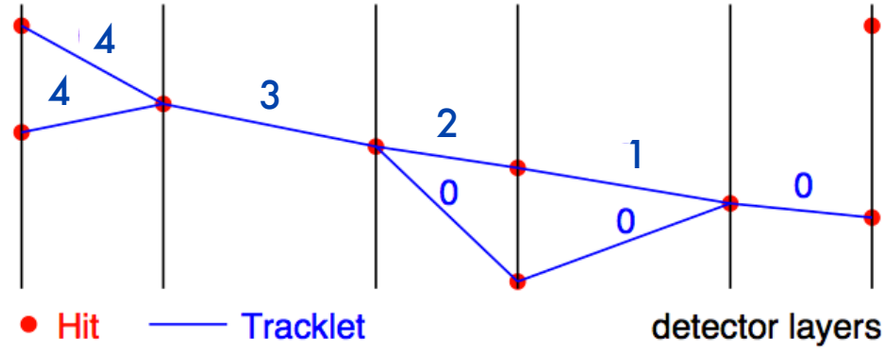
Step2



Step3

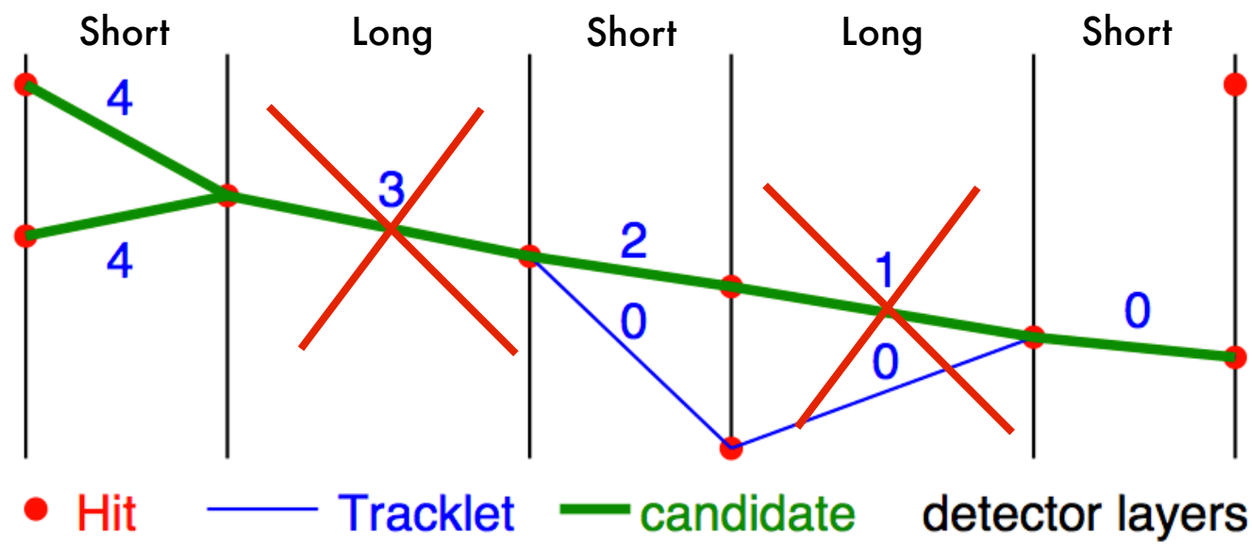


Step4



Tracking delle T-stations su GPU (3)

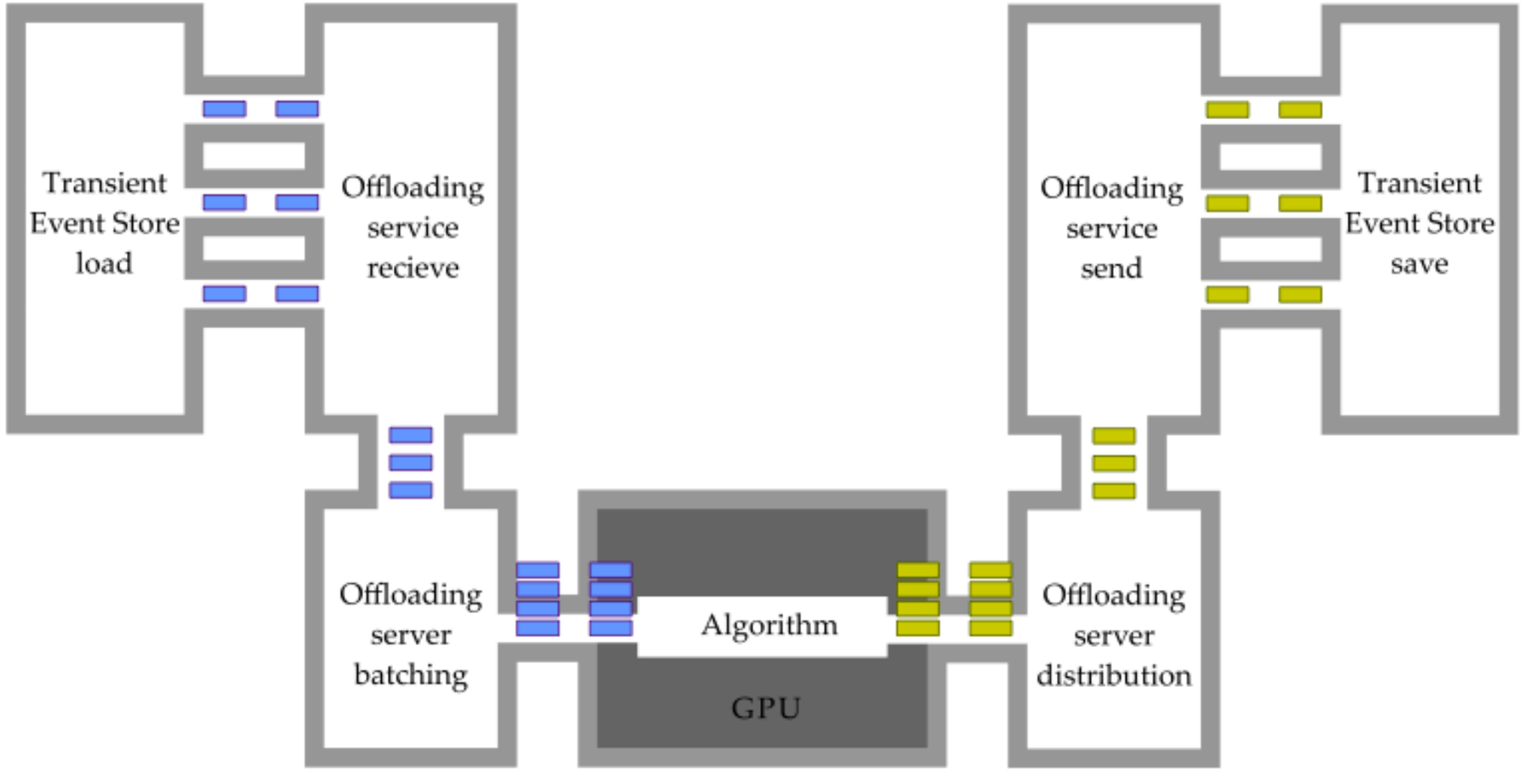
- Il CA è stato modificato per adattarlo alle caratteristiche del detector:
  - Per ridurre il combinatorio, i segmenti che connettono layers in stazioni diverse non vengono creati
  - Prima si trovano i segmenti tra x-layers, aggiungendo hits compatibili negli stereo layers
  - Fit dei segmenti e taglio sul χ^2
  - Ricerca dei vicini
  - Evoluzione e selezione delle tracce



- L'algoritmo è ancora in fase di sviluppo: il codice è stato scritto in CPU (C++), verrà poi sviluppato per GPU

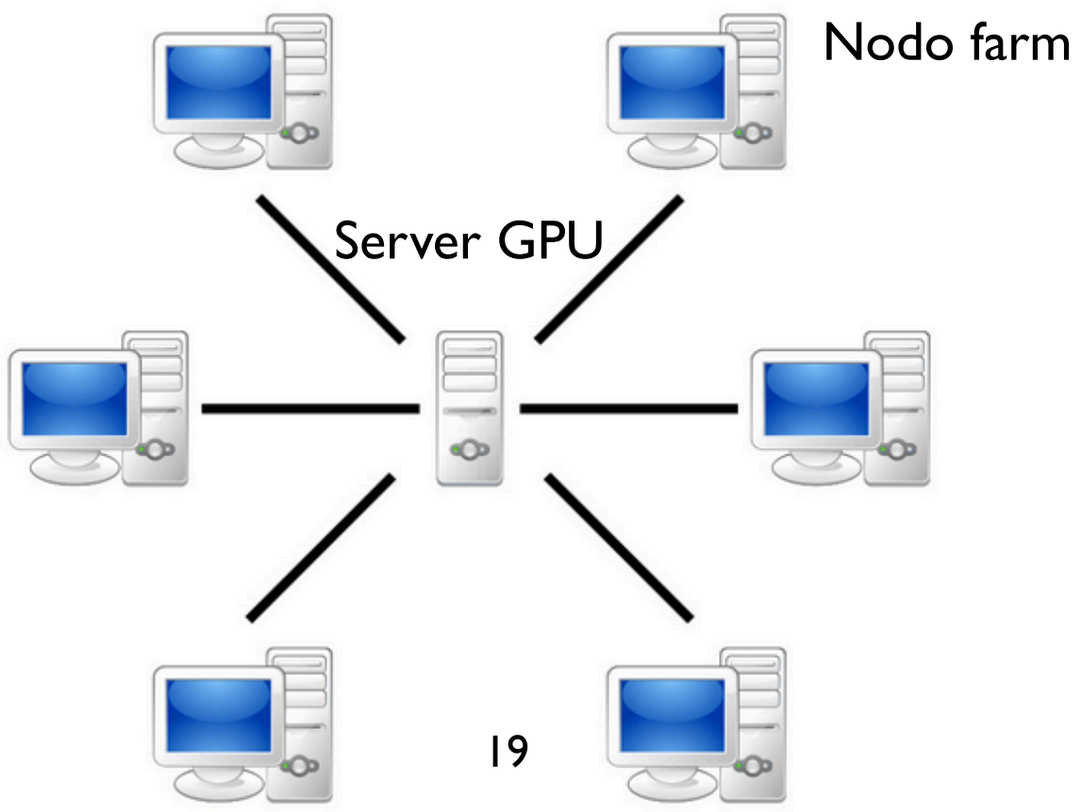
Integrazione nell'online farm (1)

- Un aspetto critico per l'utilizzo di acceleratori è l'integrazione con il framework software del trigger ("Gaudi")
- In LHCb stiamo sviluppando un sistema di "offloading" usando un sistema client-server ("GPUManager")



Integrazione nell'online farm (2)

- Ogni nodo dell'online farm agisce da client il quale manda l'evento al server GPU (es. le hits di un rivelatore)
- Il server riceve gli eventi e le accumula in un buffer
- Il bunch di eventi viene processato in GPU e il risultato rispedito a ciascun client



Integrazione nell'online farm (3)

Esempio di utilizzo:

I dati input/output devono essere convertiti in SoA dal client:

```
// compute total size and allocate memory
uint8_t * dst = (uint8_t *)&buffer[0];
const size_t noSensorsSize       = sizeof(int);
const size_t noHitsSize          = sizeof(int);
const size_t sensorZsSize        = m_event.sensorZs.size() * sizeof(int);
const size_t sensorHitStartsSize = m_event.sensorHitStarts.size() * sizeof(int);
const size_t sensorHitsNumsSize  = m_event.sensorHitsNums.size() * sizeof(int);
const size_t hitIDsSize          = m_event.hitIDs.size() * sizeof(int);
const size_t hitXsSize           = m_event.hitXs.size() * sizeof(float);
const size_t hitYsSize           = m_event.hitYs.size() * sizeof(float);
const size_t hitZsSize           = m_event.hitZs.size() * sizeof(float);

// serialize container contents
dst = copy(m_event.sensorZs,         dst);
dst = copy(m_event.sensorHitStarts, dst);
dst = copy(m_event.sensorHitsNums,  dst);
dst = copy(m_event.hitIDs,          dst);
dst = copy(m_event.hitXs,           dst);
dst = copy(m_event.hitYs,           dst);
dst = copy(m_event.hitZs,           dst);
```

Il client puo' quindi sottomettere i dati al server specificando il kernel:

```
gpuService->submitData("PrPixelCudaHandler", &m_serializedEvent[0], m_serializedEvent.size());
```

Integrazione nell'online farm (4)

- Nell'attuale implementazione, client e server comunicano tra loro tramite protocollo TCP-IP

- Si prevede di esplorare anche altri tipi di protocolli (es. Infiniband, MPI) per diminuire la latenza di trasmissione

- GPUManager è in piena fase di sviluppo/test e si prevede di utilizzarlo nel test-bed durante il Run2

- E' in fase di sviluppo (CERN, LHCb, ATLAS) anche una versione multi-threaded di Gaudi ("GaudiHive"), basata su tbb, in grado di supportare acceleratori.

Conclusioni (1)

- In conclusione, GPU e altri acceleratori possono fornire un valida alternativa alle CPU (anche in termini economici) in applicazioni real-time come il trigger
- Ottenere speedups molto elevati mantenendo le **stesse performances** fisiche è pero' tutt'altro che banale
  - Comunque il metro di giudizio è il throughput/costo HW!
- Per ottenere un buon speedup a parità di efficienze è necessario ristrutturare il codice CPU originale:
  - Quanto il codice vada cambiato dipende molto dall'algoritmo di partenza e da come è stato scritto...

Conclusioni (2)

- Un'altra opzione è implementare un nuovo algoritmo, manifestamente parallelo:
  - In questo caso occorre avere una buona conoscenza del problema (detector, ...) adattando l'algoritmo generico (Hough, Automa, ...) al caso particolare

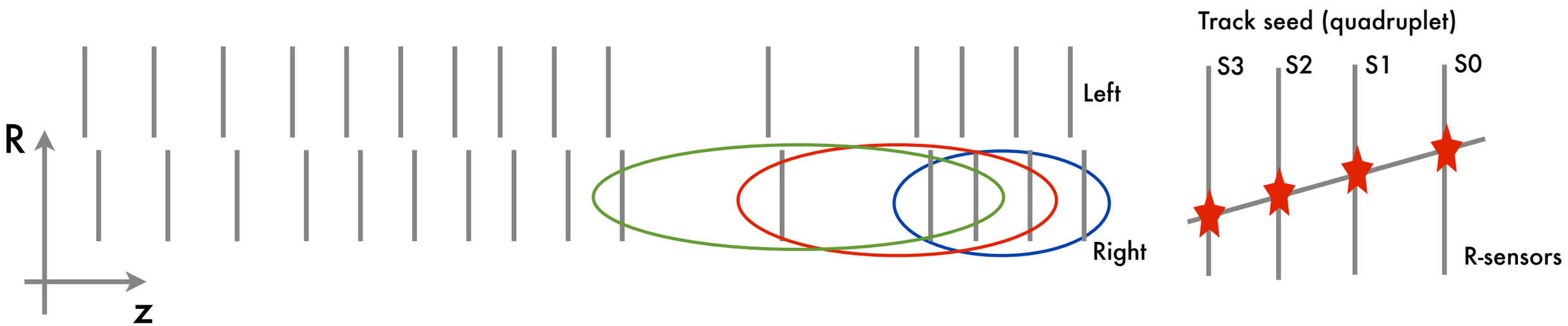
- Il Run2 fornisce un opportunità unica per testare le performances delle GPU e la loro integrazione nel sistema online

- Altri algoritmi, ad es. la ricostruzione del RICH, sono buoni candidati per architetture parallele e verranno esplorati in futuro

Back-up slides

Tracking del VELO su GPU (4)

- **RZ tracking:**
- Only R-sensors are used
- The algorithm looks for quadruplets of hits in four contiguous R-sensors (seed) on both halves.
  - Each thread works on a set of four contiguous R-sensors and find all quadruplets.
- Then each quadruplet is extended in parallel as much as possible adding the remaining R-sensors



Tracking del VELO su GPU (5)

- **Space tracking:**

- Add hits on ϕ-sensors

- Each RZ track is processed concurrently by assigning a space-tracking algorithm to each thread:

  - Search for a triplet of hits: for each hit in the first two ϕ-sensors, the candidate hit in the third sensor is the one most compatible with predicted position (best χ^2)

  - The track is extended and its parameters are found by minimizing $\Sigma_{points}\ \chi^2$ (linear system solved by substitution)

  - This part is almost a re-writing in CUDA of the original space-tracking code