Studio delle architetture GPU per la ricostruzione di particelle nella fisica delle alte energie

Workshop GARR Calcolo Storage-Distribuito

Dott. Silvio Pardi

INFN-Napoli

Parallelismo nella Fisica delle Alte Energie

Attività a livello mondiale di R&D per esplorare le possibilità di accelerare i framework di analisi mediante l'ausilio di paradigmi di calcolo parallelo e sfruttando a pieno le tecnologie multi/many core.

- Identificare quali sono I requisiti degli esperimenti HEP futuri
- Identificare le tecnologie potenzialmente interessanti.
- Identificare similitudini tra diversi progetti software, individuare sinergie.

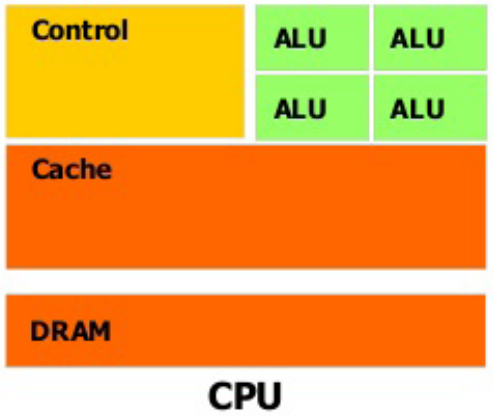
Centri attualmente coinvolti in questo forum di discussione sono FNAL, CERN, LBL, DESY, INFN.

Goal della sperimentazione

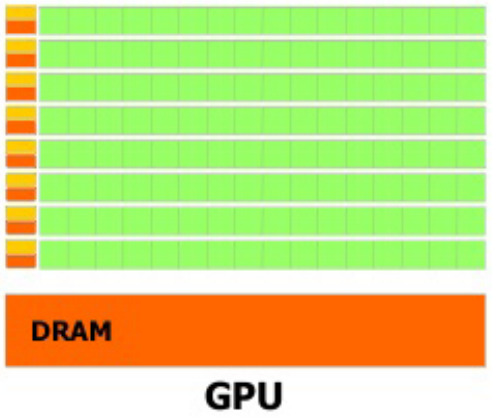
•Studiare la possibilità di utilizzare le architetture GPGPU per specifici task di interesse per la fisica delle alte energie.

•Studiare I setup migliori dal punto di vista sistemistico e capire come potranno incidere nella progettazione dei futuri datacenter

•Sviluppare un know-how interno



CPU



GPU

•multi-core Technology

•High speed and complex processing unit

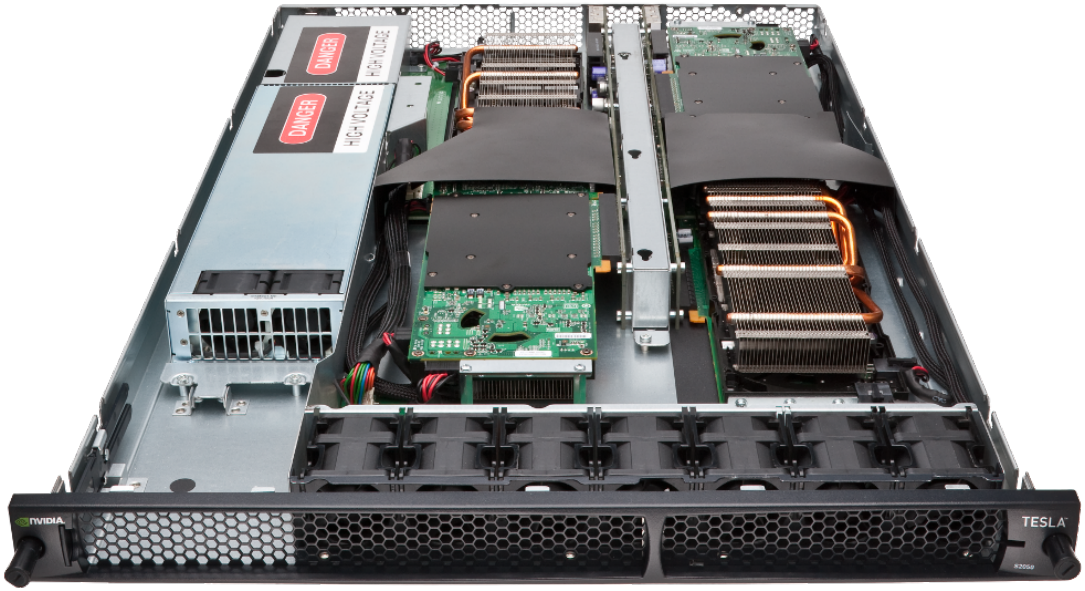
• General Purpose

•many-core technology

• Hundred of simple Processing Units

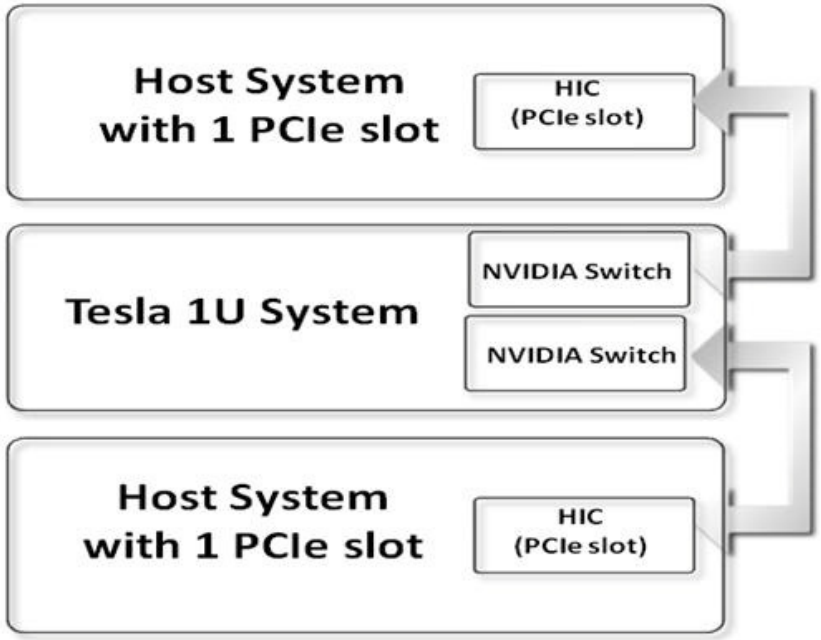
• Designed to macht the SIMD paradigm (Single Instruction Multiple Data)

The Hardware disponibile a Napoli



1U rack NVIDIA Tesla S2050

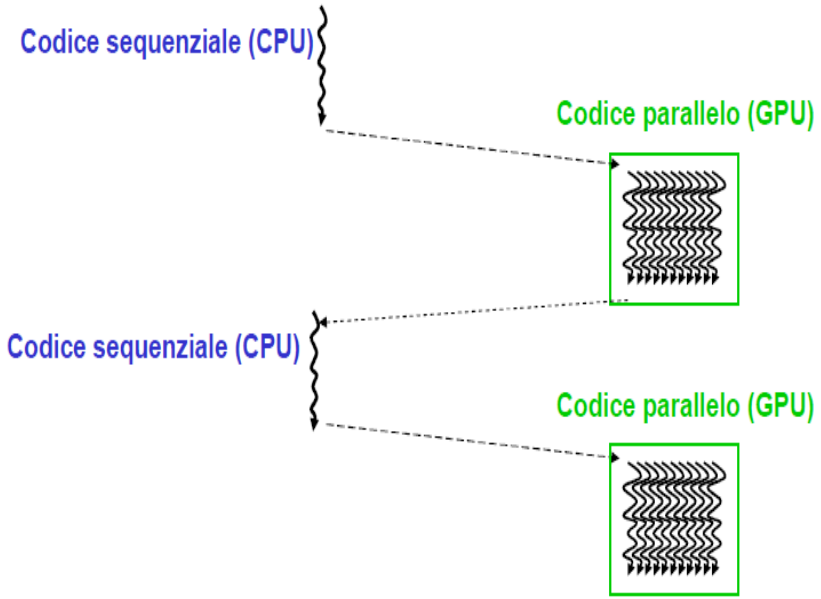
- 4 GPU Fermi
- Memory for GPU: 3.0 GB
- Core for GPU: 448
- Processor core clock: 1.15 GHz



2U rack Dell PowerEdge R510

- Intel Xeon E5506 eight-core @ 2.13 GHz
- 32.0 GB DDR3
- 8 hard disk SATA (7200 rpm), 500 GB

CUDA C Technology

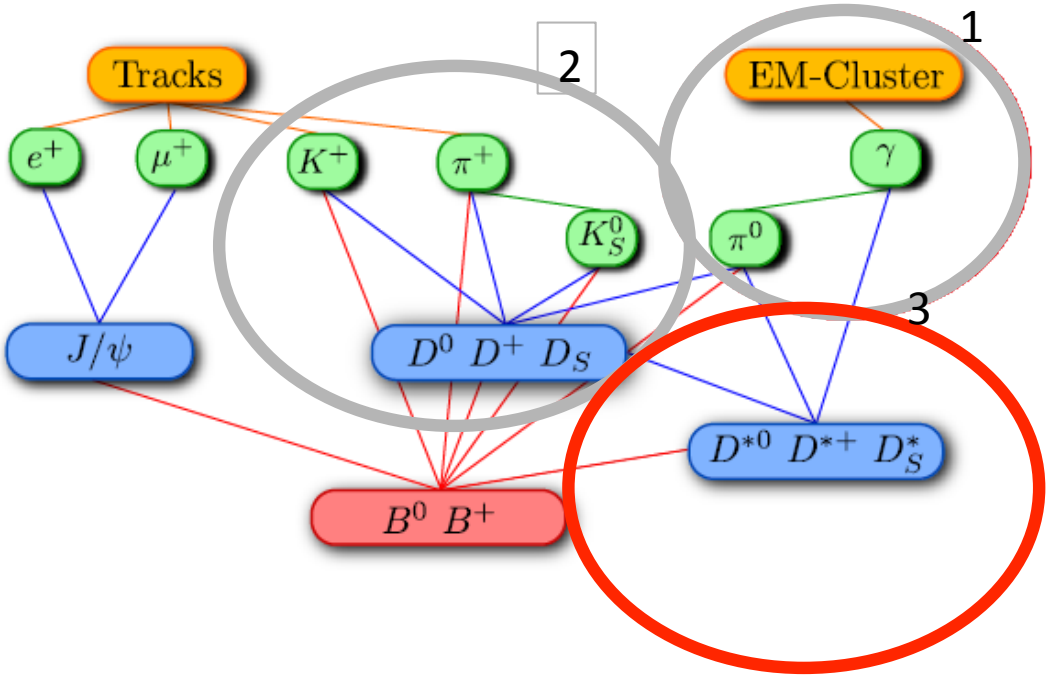


Hybrid code : Combinazione di codice C standard e direttive **CUDA** necessarie per programmare ed interagire con I GPU Device.

Quattro step principali caratterizzano un codice CUDA C :

- GPU Memory Allocation
- Data transfers from CPU to GPU
- GPU Kernel execution
- Data transfers from GPU to CPU

Algoritmo di ricostruzione del mesone B



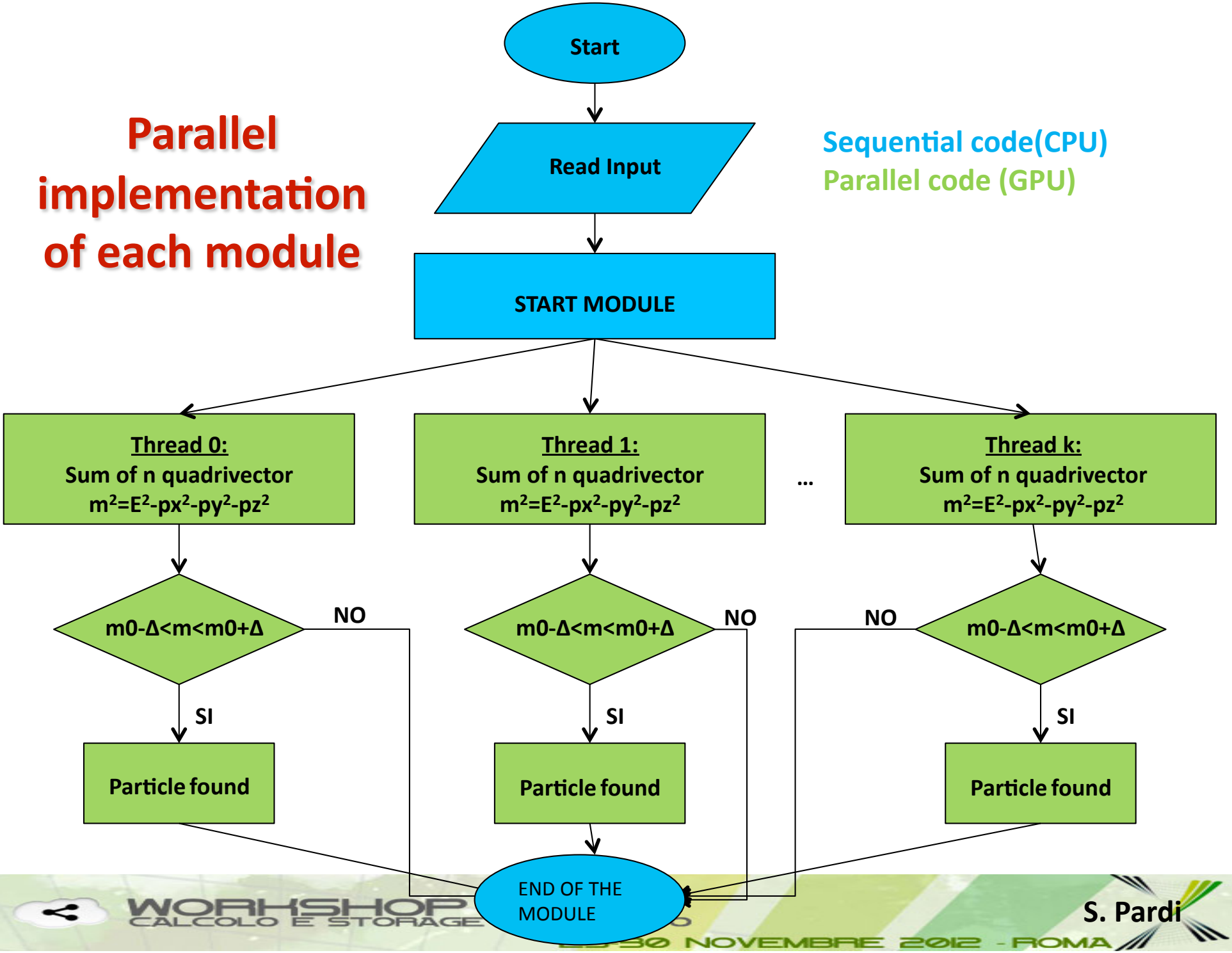
stage	particelle
1	tracks, K_S, γ, π^0
2	$D^{\pm}_{(s)}$, D^0, e J/ψ
3	$D^{*\pm}_{(s)}$ e D^{*0}
4	$B^{\pm}$ e B^0

Problema Combinatorio

Problema di Modellizzazione: dai N quadrivettori (3 componenti spaziali e una per l'energia), combinare tutte le coppie senza ripetizioni.Quindi calcolare la massa della nuova particella e verificare se si trova in un range specifico dato in input.

Parallel implementation of each module

Sequential code(CPU)
Parallel code (GPU)



Methodologia usata

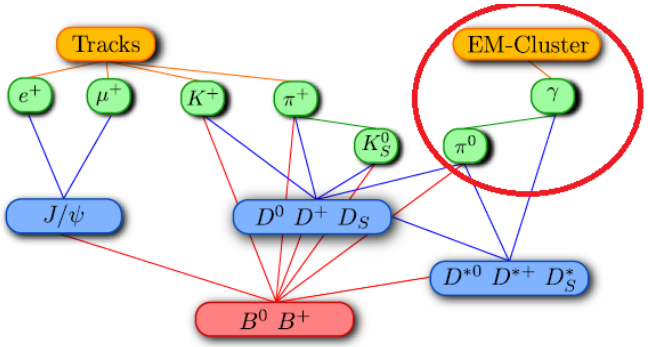
- Memory in input
- CudaMalloc input
- CudaMalloc output
- Load input array
- cudaMemcpy() hostTOdevice
- cudaMemset()
- Kernel
- cudaMemcpy() deviceTOhost
- cudaFree()
- free()
- Total time

Sono implementate le versioni sequenziale e parallelo su GPU dello stesso codice estrapolando l'algoritmo dal framework di Babar.

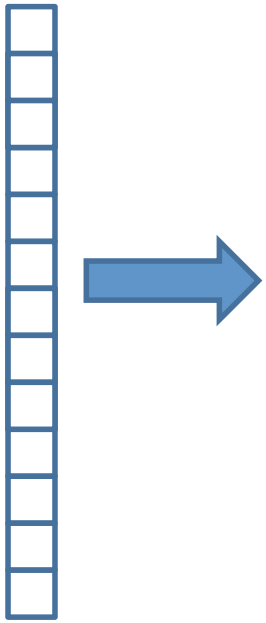
Gli algoritmi vengono testati in prima analisi incrementando I dati in input al fine di individuare la minima dimensione per un dataset su quale le GPU riescono ad accelerare il calcolo.

Misurare il tempo speso da ogni funzione caratterizzante di CUDA al fine di misurare l'overhead aggiunto dal sistema.

Π^0 Module from a Gamma Vector



Gamma Vector
n elements



Numero di possibili coppie=*n (n-1)/2*

n(n-1)/2 tread on GPU. Ogni tread combina 2 particelle attraverso un operazione di saxpy quindi calcola la massa in base all'energia e verifica se essa ricade in un range prestabilito preso in input

CODE EXAMPLE

Tread ID

```
sh=threadIdx.x+blockDim.x*threadIdx.y;
tid = blockDim.x*blockDim.y*(blockIdx.x + gridDim.x* blockIdx.y) + sh;

//Algoritmo per il calcolo degli indici giusti in base al thread
if (tid<N2){
```

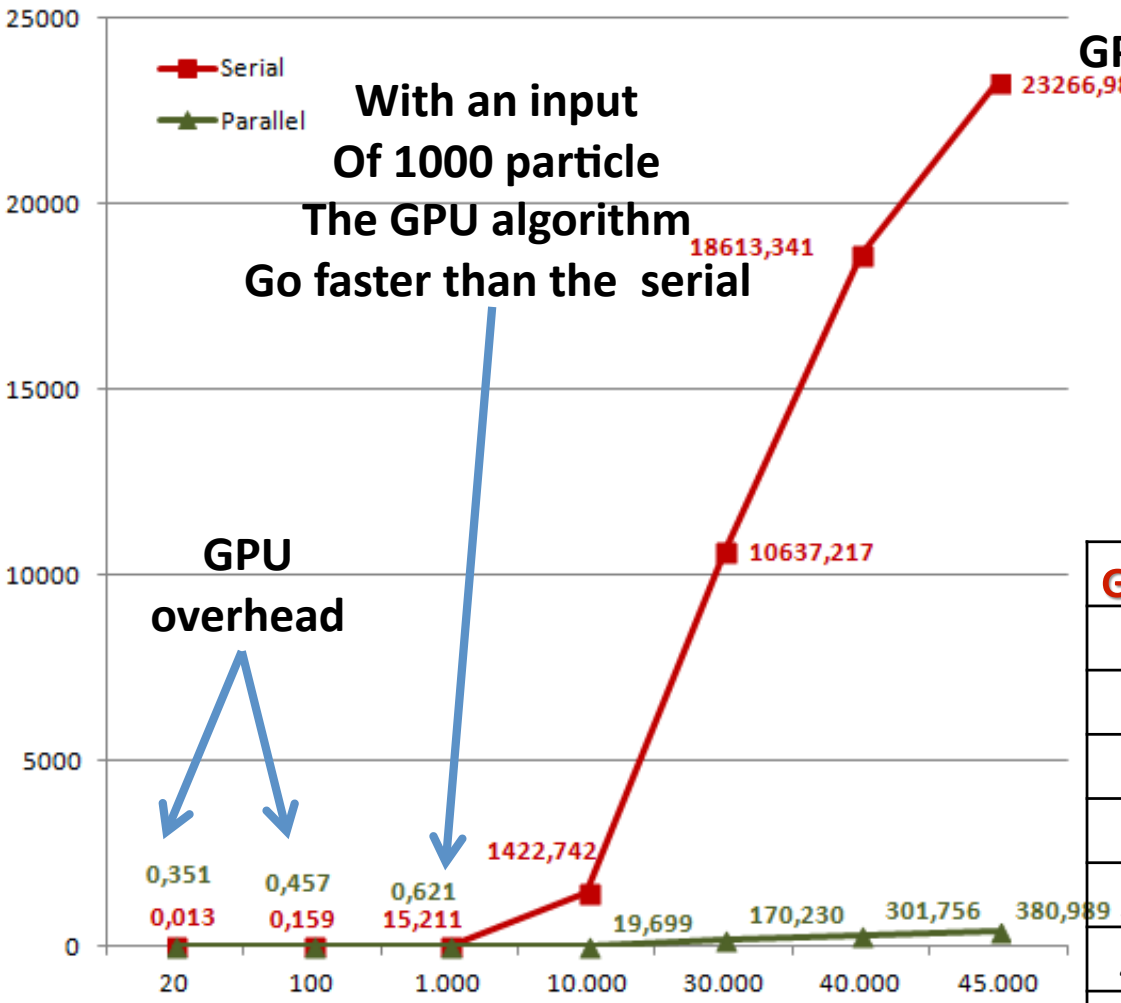
Define the work for each tread

```
        linIdx=N2-tid;
        i=int(N - 0.5 - sqrt(0.25 - 2 * (1 - linIdx)));
        z=(N+N-1-i)*i;
        j=tid - z/2 + 1 + i;

        if (i==j){
                i=i-1;
                j=N-1;
        }
```

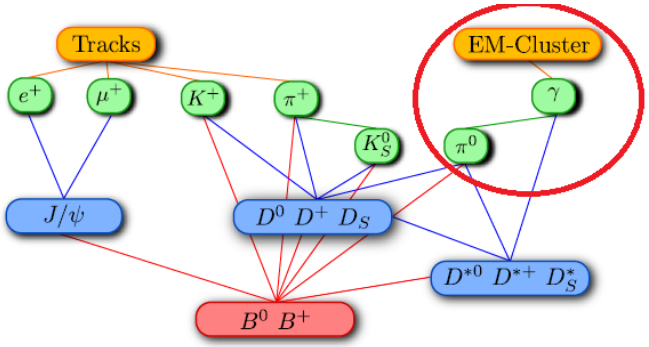
saxpy

```
        //Somma di due quadrivettori
        Candidato.x=Gamma[i].x+Gamma[j].x;
        Candidato.y=Gamma[i].y+Gamma[j].y;
        Candidato.z=Gamma[i].z+Gamma[j].z;
        Candidato.Ene=Gamma[i].Ene+Gamma[j].Ene;

        //Controllo massa
massa=(Candidato.Ene*Candidato.Ene)-(Candidato.x*Candidato.x)-
(Candidato.y*Candidato.y)-(Candidato.z*Candidato.z);

        if(sqrt(massa)>M0-del && sqrt(massa)<M0+del && (*dimPI0)
<=Tmax_PI0
```

Algorithm evaluation

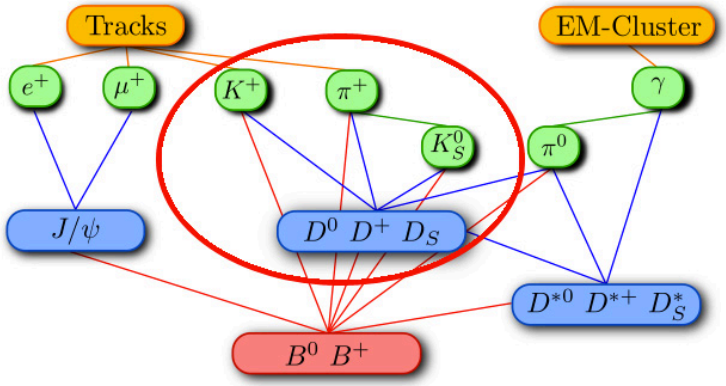


In the best case the GPU Algorithm is 72 time faster than the serial version



GAMMA	SERIAL	PARALLEL	S/P
20	0,013 ms	0,351 ms	0,04
100	0,159 ms	0,457 ms	0,35
1.000	15,211 ms	0,621 ms	24,49
10.000	1.422,742 ms	19,699 ms	72,22
30.000	10.637,217 ms	170,23 ms	62,49
40.000	18.613,341 ms	301,756 ms	61,68
45.000	23.266,988 ms	380,989 ms	61,07

D^0 KPi - Module from a Gamma Vector



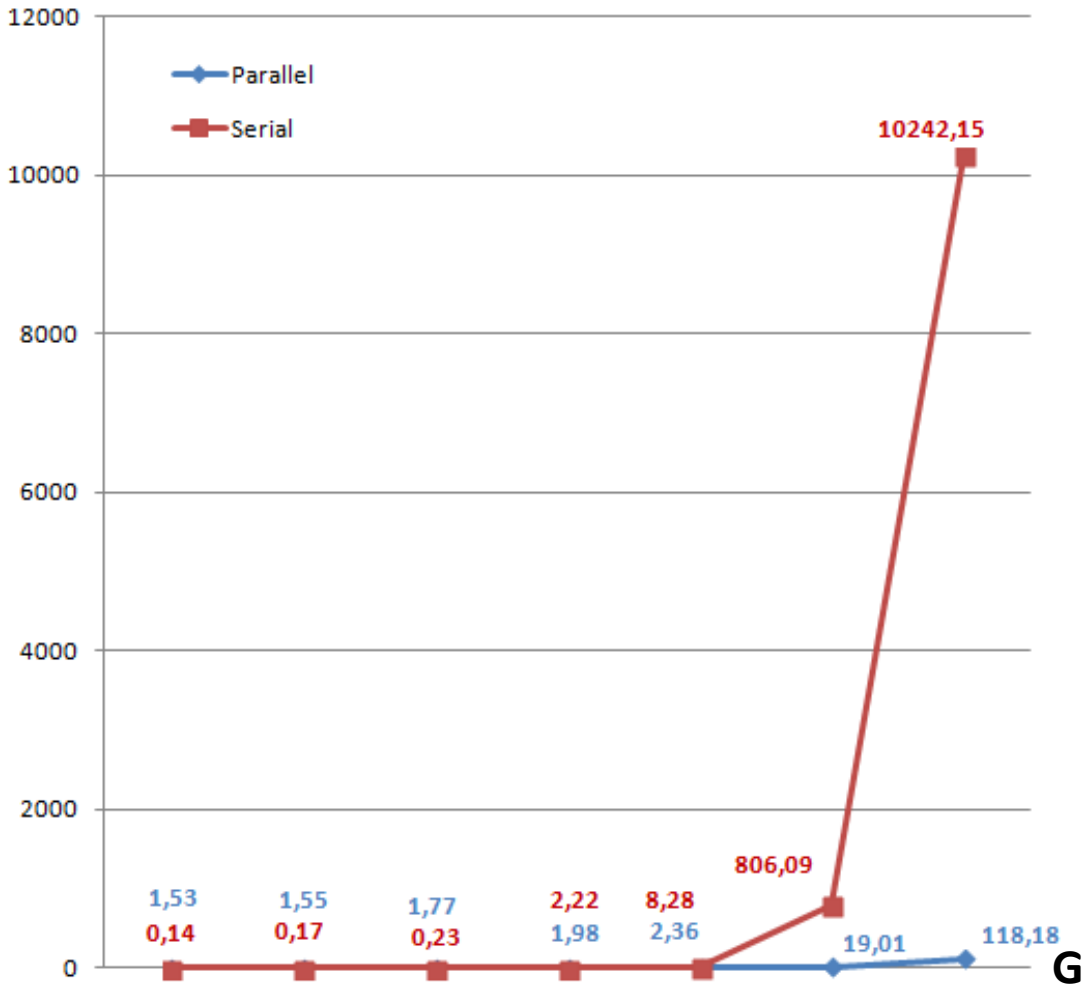
Vettore di n Pi+
Con 20% k

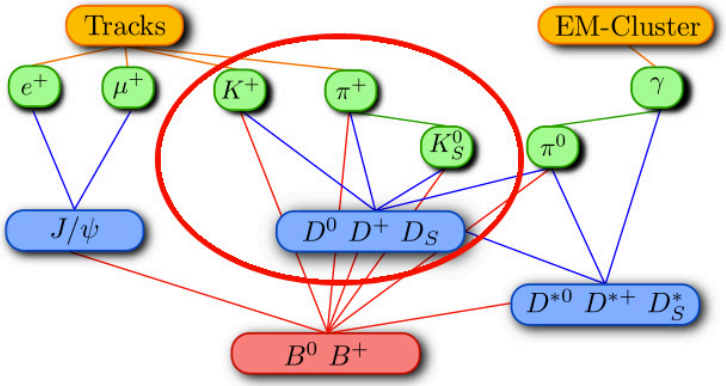
Vettore di m Pi-
Con 20% k

Tutte le possibili
Coppie di K+ Pi- e
Pi+ K-

*aK*m +bK*n tread on GPU. Ogni tread combina 2 particelle attraverso un operazione di* saxpy quindi calcola la massa in base all'energia e verifica se essa ricade in un range prestabilito preso in input

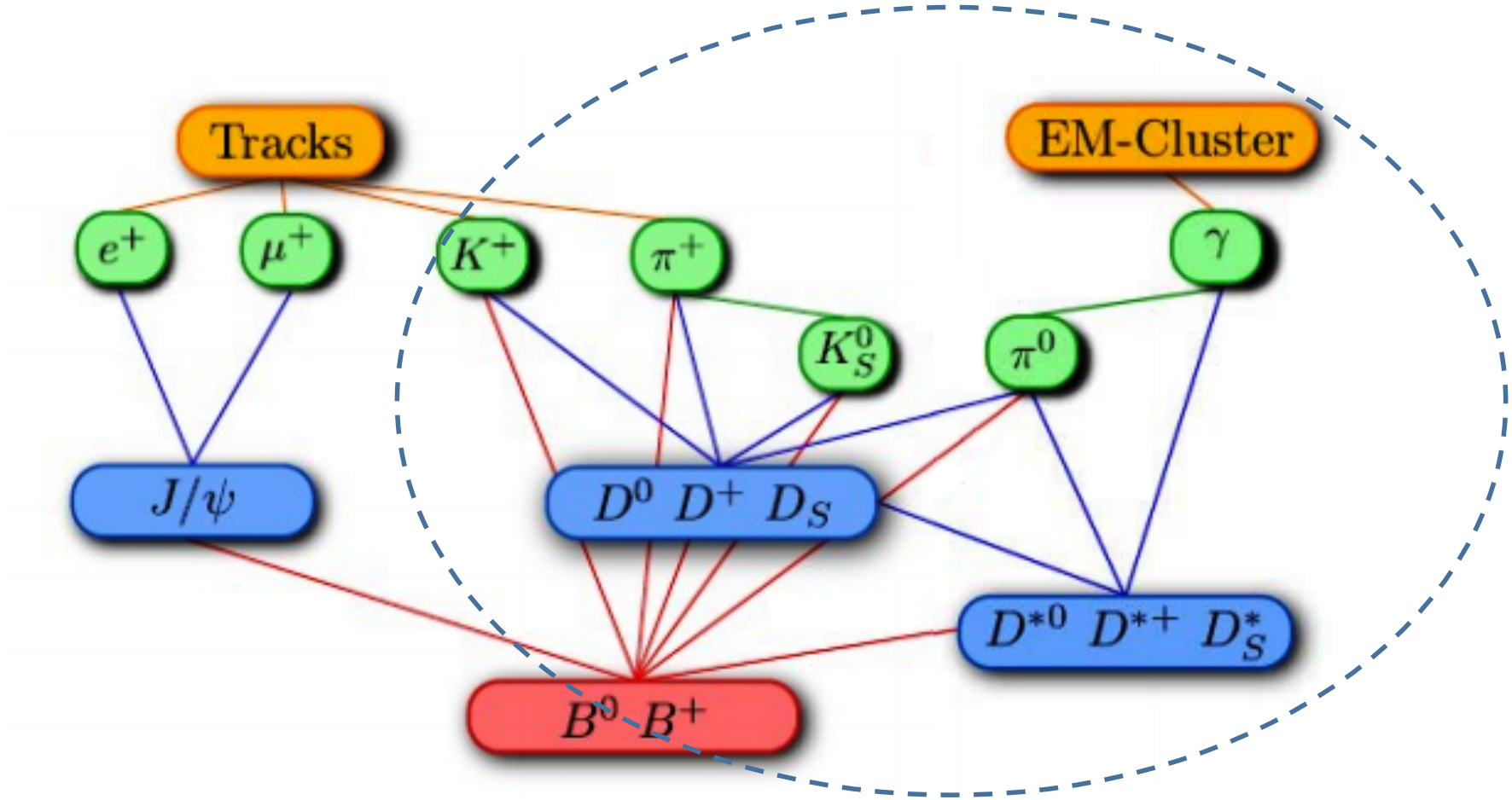
Algorithm evaluation



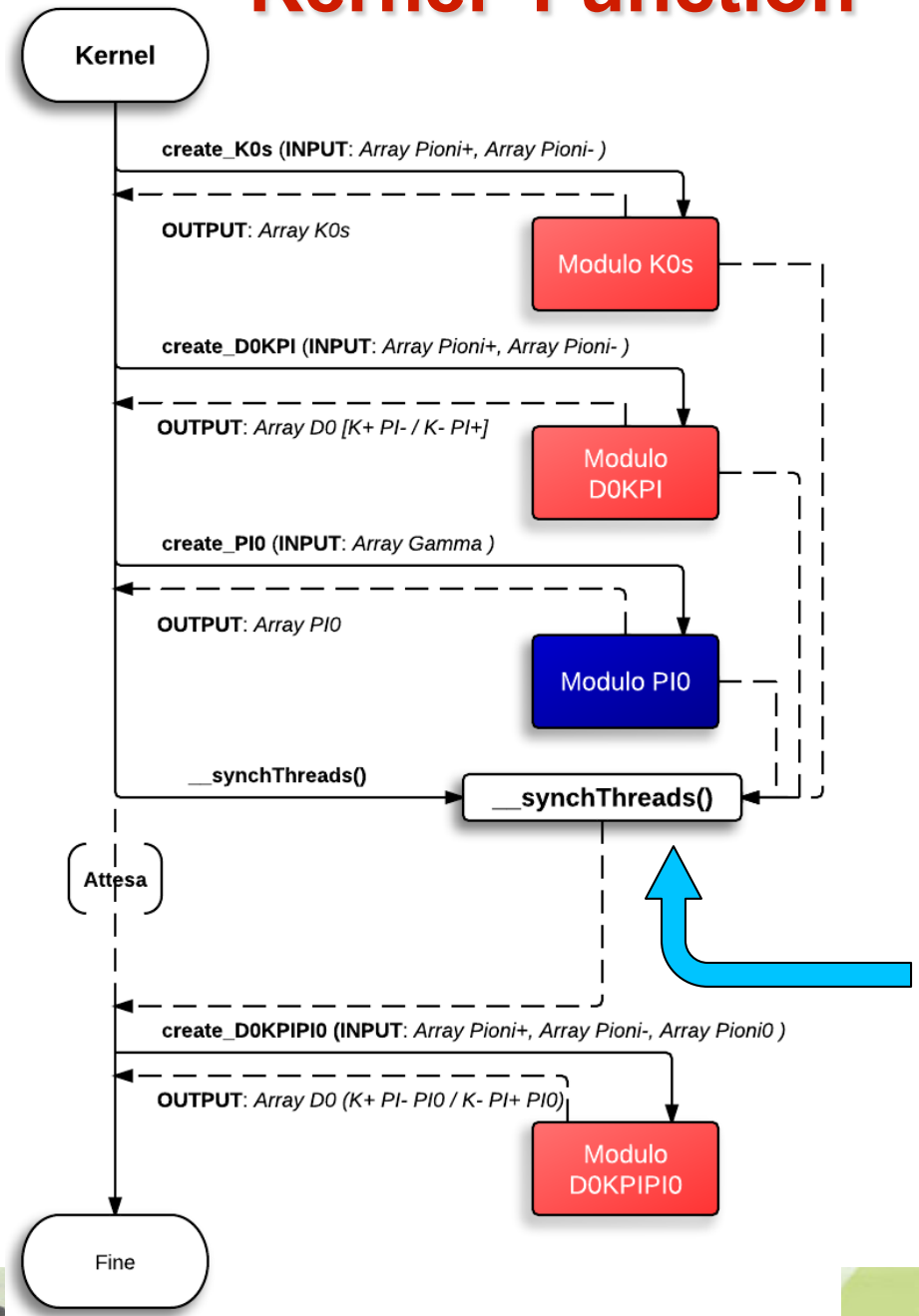


Pi+/P-	SERIAL	PARALLEL	S/P
5	0,14 ms	1,53 ms	0,09
50	0,17 ms	1,55 ms	0,11
100	0,23 ms	1,77 ms	0,13
500	2,22 ms	1,98 ms	1,12
1000	8,28 ms	2,36 ms	3,51
10000	806,09 ms	19,01 ms	42,40
40.000	10242,15 ms	118,18 ms	86,67

In the best case the GPU Algorithm is 86 time faster than the serial version with a very large input


Tracks
EM-Cluster
e^+
μ^+
K^+
π^+
γ
K_S^0
π^0
J/ψ
D^0 D^+ D_S
D^{*0} D^{*+} D_S^*
B^0 B^+

Kernel Function



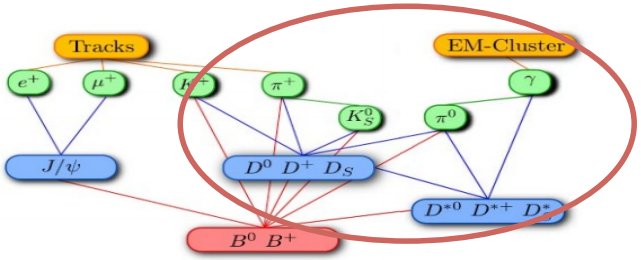
__synchThreads()

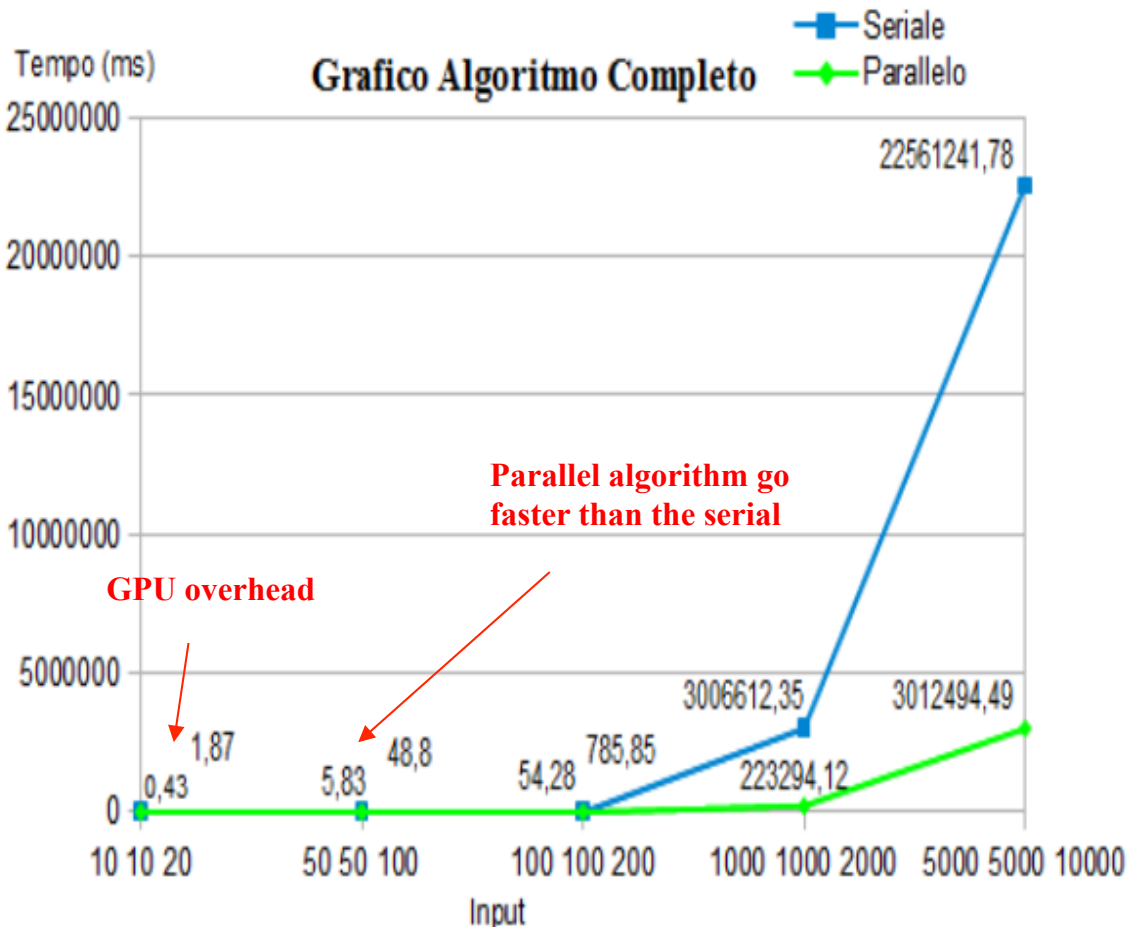
Overhead analysis

Pi+/Pi-	Gamma	Cuda Malloc	Cuda Mem Copy (Host to Device)	CudaMemset()	Cuda Mem Copy (Device to Host)	CudaFree	Total
5	20	0,376	0,111	0,056	0,239	0,832	1,614
10	100	0,368	0,113	0,05	0,28	0,918	1,729
50	200	0,568	0,113	0,05	0,341	0,959	2,031
100	400	0,577	0,113	0,05	0,416	1,055	2,211
200	800	0,703	0,122	0,05	0,565	1,291	2,731
500	2000	0,95	0,147	0,05	1,976	2,59	5,713

Algorithm Evaluation

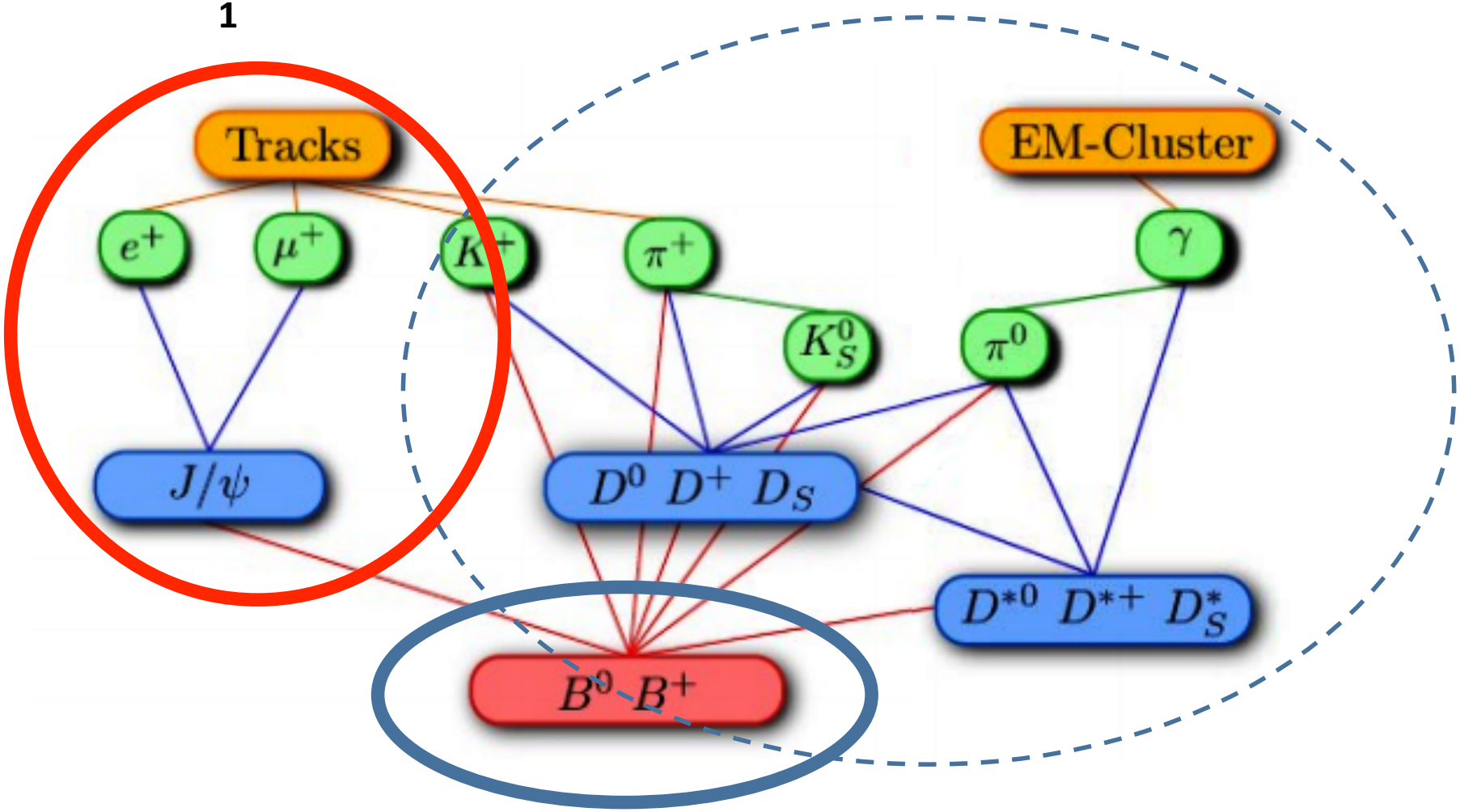
Pi+/P-	Gam.	SERIAL	PARALLEL	S/P
10	20	0,43 ms	1,87 ms	0,23
50	**100**	**48,8 ms**	**5,83 ms**	**8,37**
100	200	785,85 ms	54,28 ms	14,48
1000	2000	3006,6 s	223,2 s	13,46
5000	10000	22561,2 s	3012,4 s	7,49





On going Work

1



2

Sommario

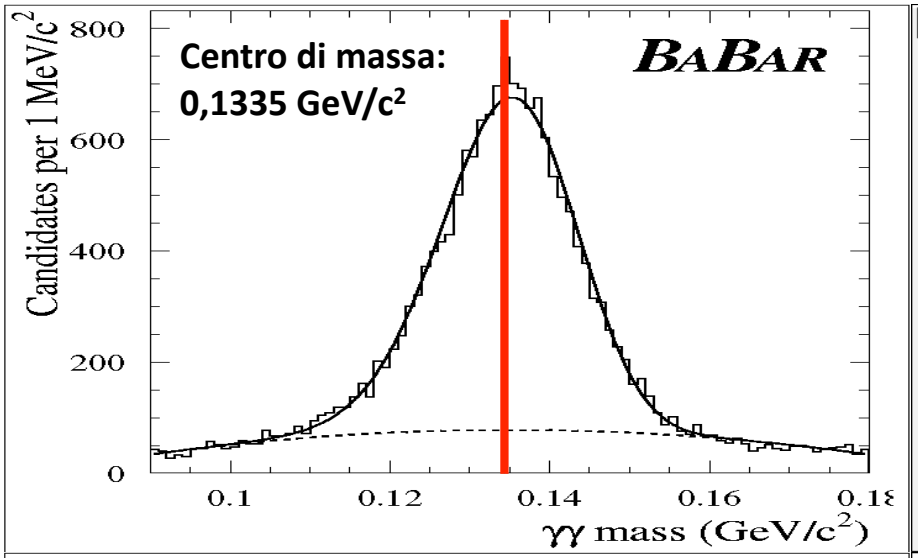
Su input piccoli siamo dominati dall'overhead

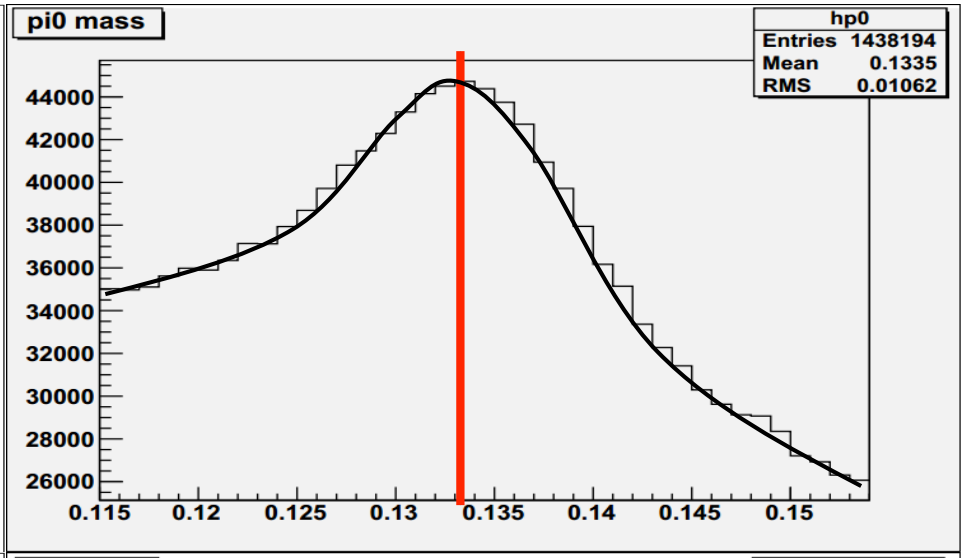
Input sufficientemente grandi traiamo benefici evidenti dalle GPU ma siamo limitati nella crescita dello speed-up dalla componente sequenziale del problema.

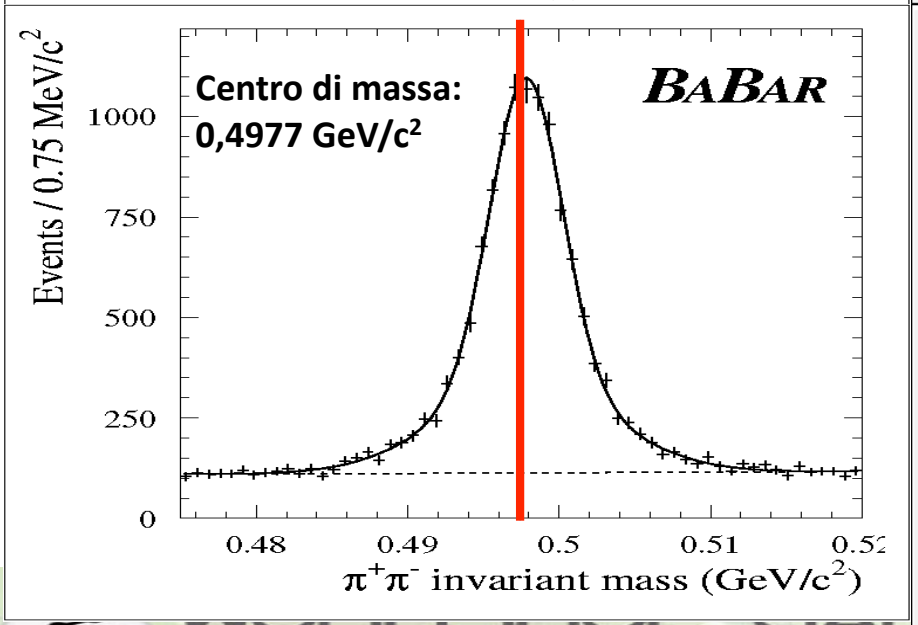
L'algoritmo non è completo manca di una parte ulteriore che aggiunge una componente parallella e che dovrebbe portare benefici nella crescita dello speed-up.

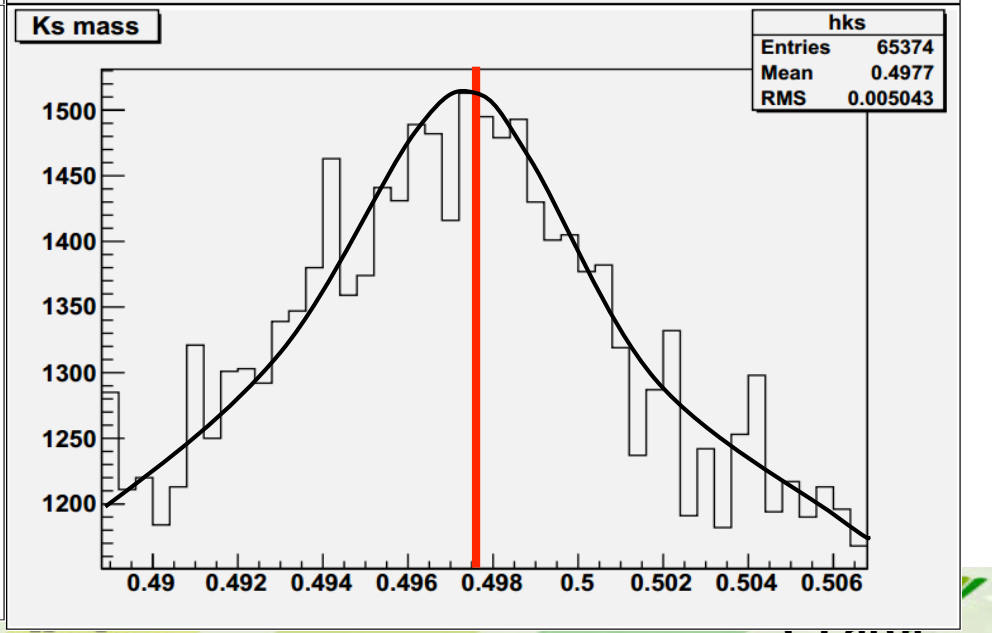
E' possibile far girare più kenel contemporanemente sul device GPU occupando eventuali tempi di IDLE di core. Questo approccio dovrebbe consentire Globalmente di aumentare il througphut in termini di eventi analizzati per unità di tempo.

First Evaluation test on a set of real data (100.000 events)









Conclusioni e prossimi sviluppi

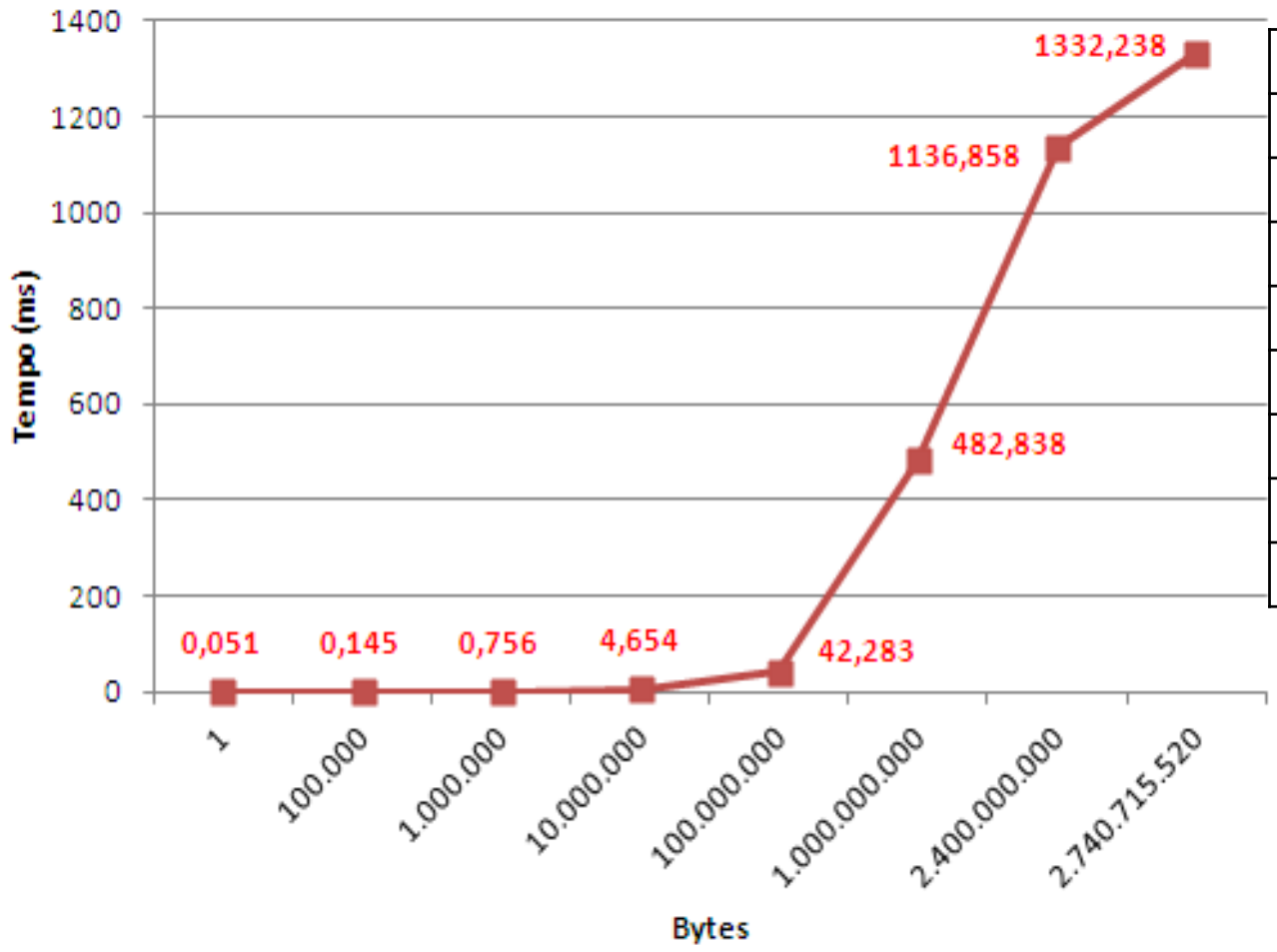
Le GPU hanno mostrato di riuscire ad accelerare il problema che stiamo studiando.

L'algoritmo è ancora incompleto e la parte che resta da sviluppare aggiunge una componente parallela che dovrebbe permetterci di migliorare ulteriormente lo speed-up globale.

Abbiamo cominciato a processare dati reali per verificare la consistenza dei risultati. Il prossimo step è quello di integrare l'algoritmo così definito nel codice del Framework di Babar e misurare le performance del codice in un ambiente completo di analisi.


MERRIE MELODIES
REG. U.S. PAT. OFF.
THE END
A WARNER BROS. CARTOON

Data Transfer CPU vs GPU



BYTE	TEMPO (ms)
1	0,051
100.000	0,145
1.000.000	0,756
10.000.000	4,654
100.000.000	42,283
1.000.000.000	482,838
2.400.000.000	1.136,858
2.740.715.520	1.332,238

TEST CUDAMEMCPY TEMPO (ms)

Comparison malloc e cudaMalloc