

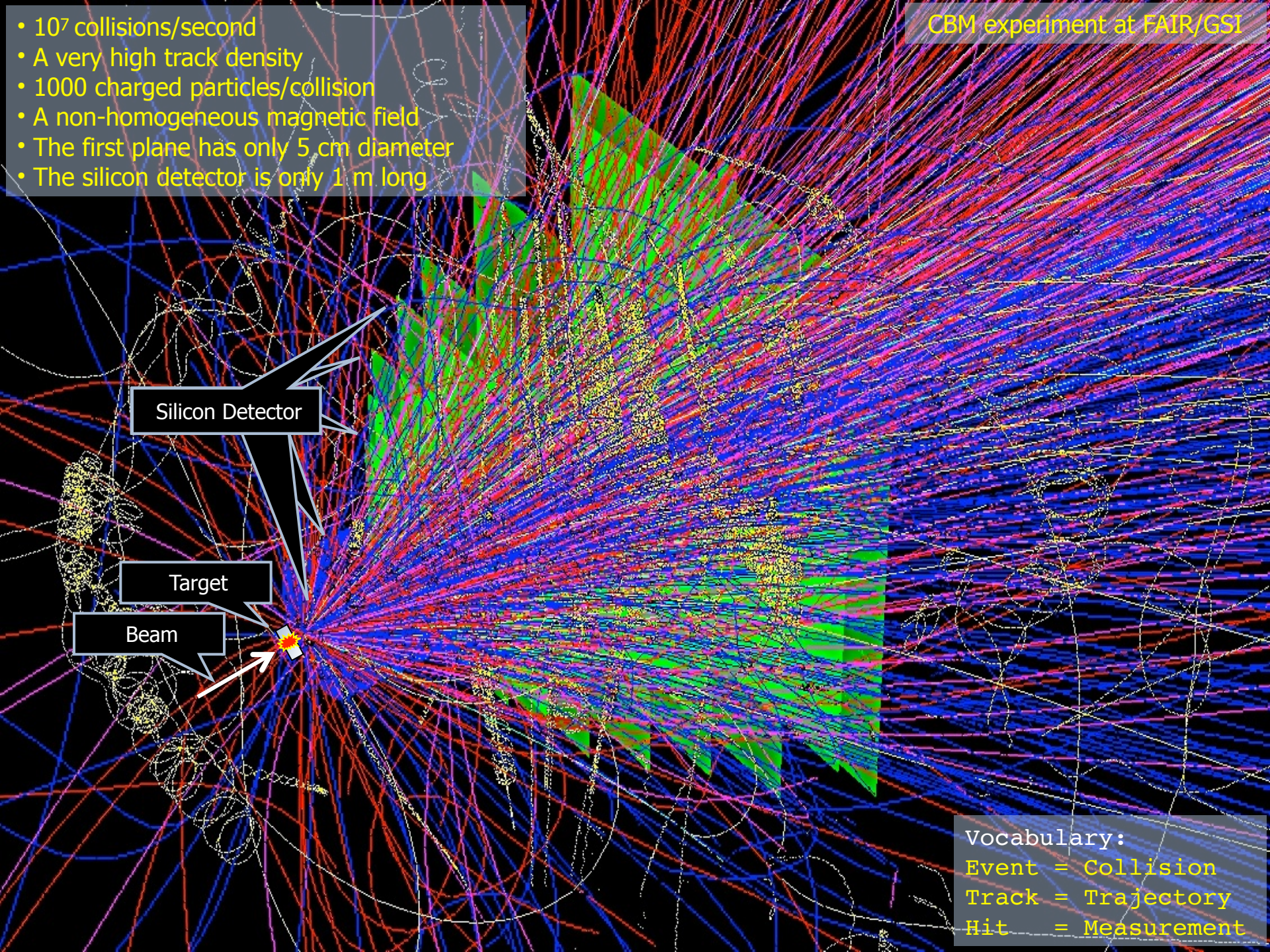
# **Workflows for efficient pattern recognition under different computing architectures**

**Prof. Dr. Ivan Kisel**

Goethe University Frankfurt am Main  
FIAS Frankfurt Institute for Advanced Studies  
GSI Helmholtz Center for Heavy-Ion Research, Darmstadt  
Helmholtz Research Academy Hesse, HFHF



- $10^7$  collisions/second
- A very high track density
- 1000 charged particles/collision
- A non-homogeneous magnetic field
- The first plane has only 5 cm diameter
- The silicon detector is only 1 m long



Silicon Detector

Target

Beam

Vocabulary:

Event = Collision

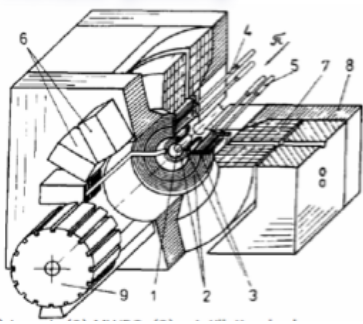
Track = Trajectory

Hit = Measurement

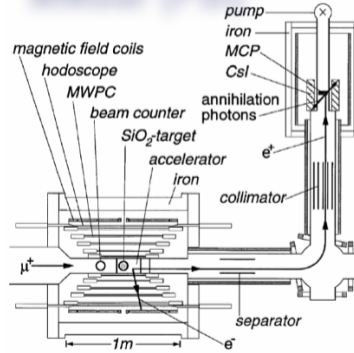


# My research experience

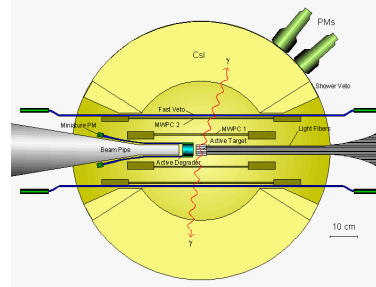
**ARES (JINR)**



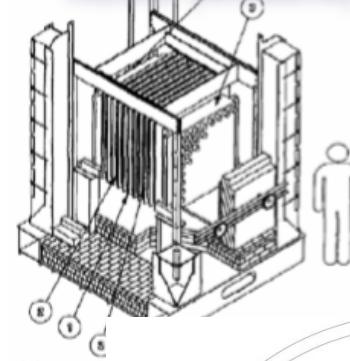
**MMbar (PSI)**



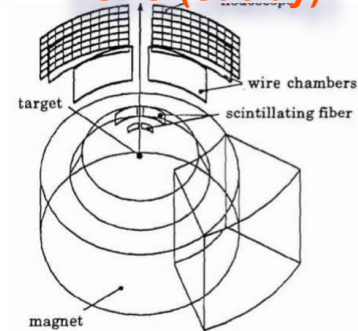
**PiBeta (PSI)**



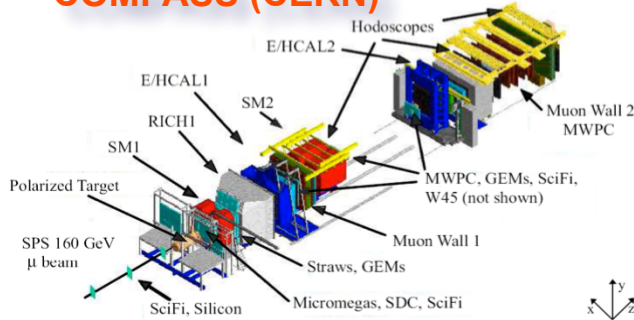
**NEMO (Modane)**



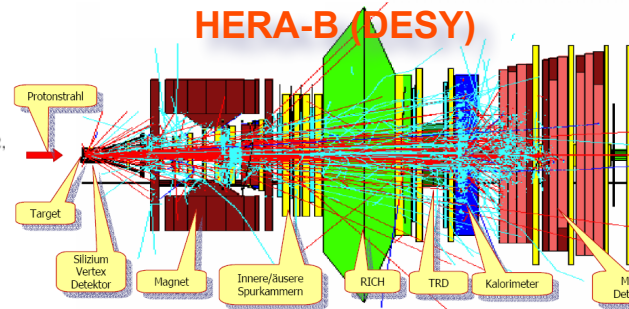
**DISTO (Saclay)**



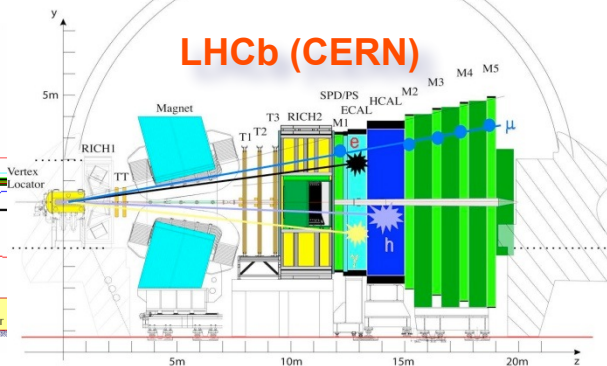
**COMPASS (CERN)**



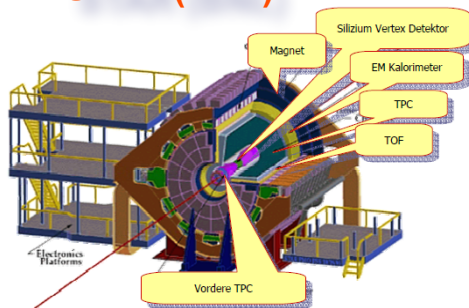
**HERA-B (DESY)**



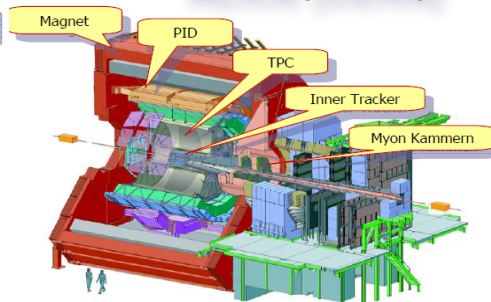
**LHCb (CERN)**



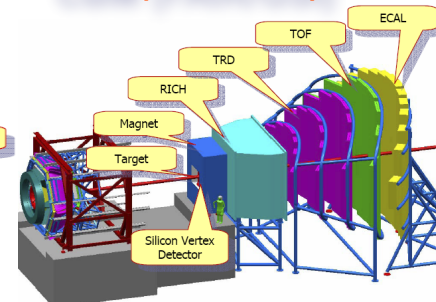
**STAR (BNL)**



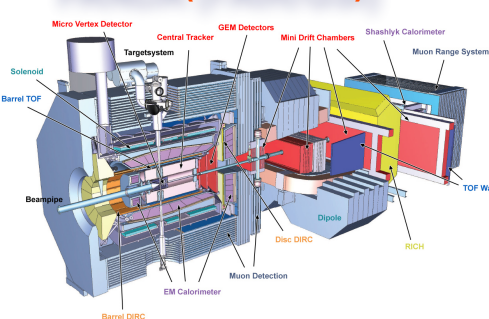
**ALICE (CERN)**



**CBM (FAIR/GSI)**

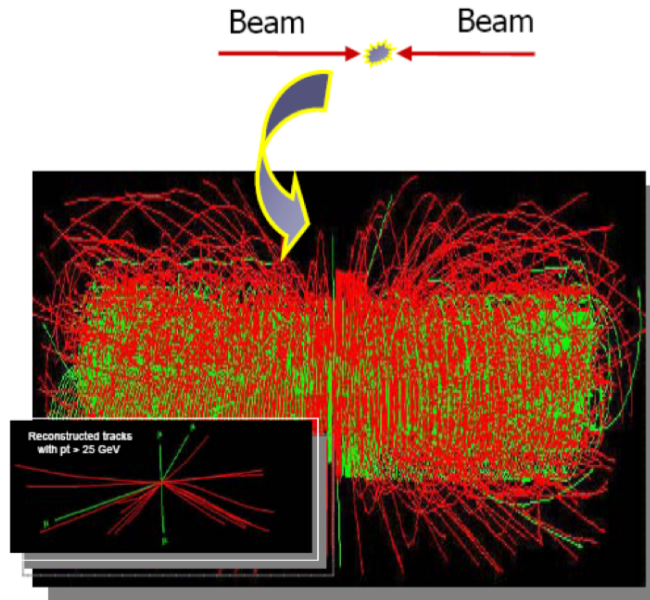


**PANDA (FAIR/GSI)**





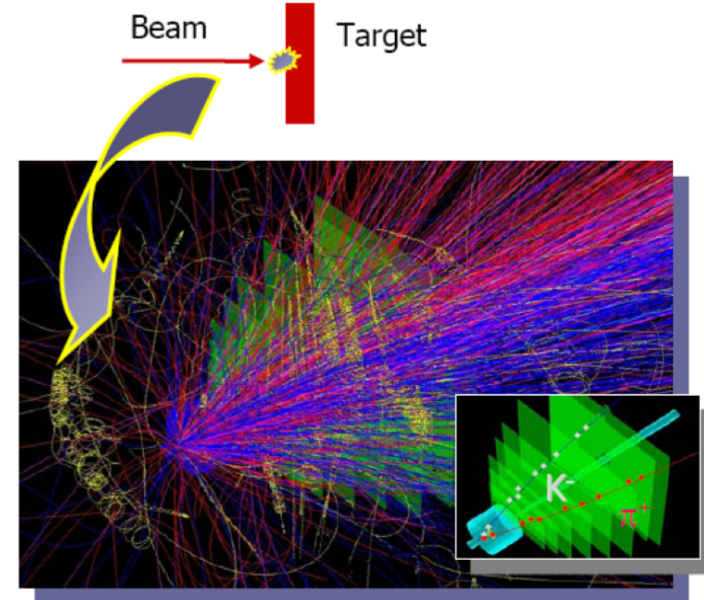
# HEP Experiments: Collider and Fixed-Target



Inelastic collisions  
 $10^7 - 10^9$

$10^{11}$

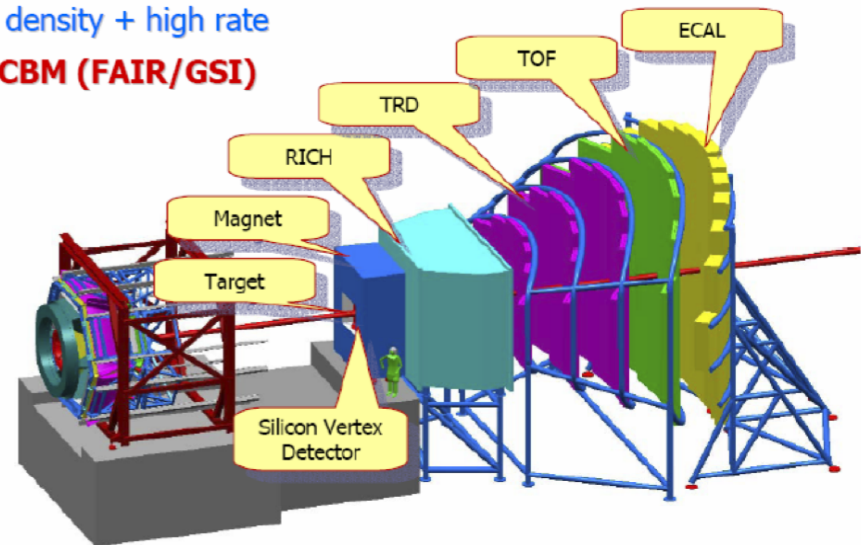
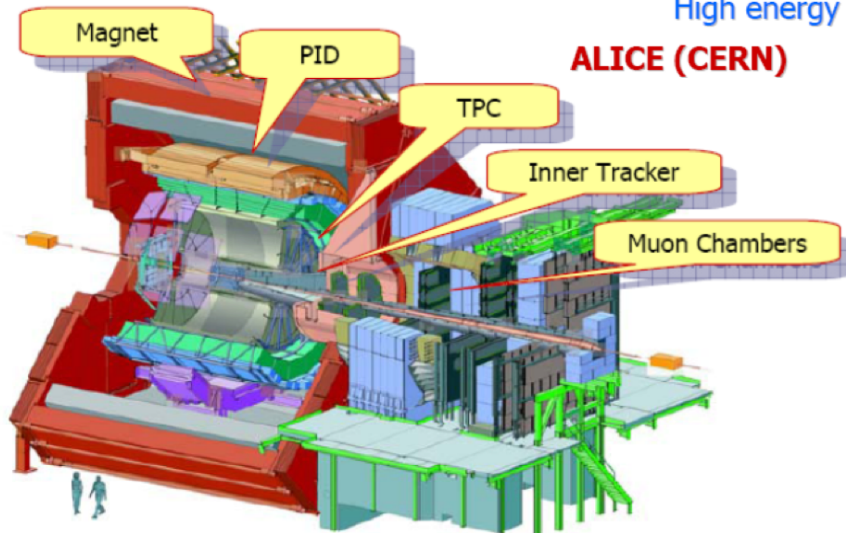
Signal events  
 $10^2 - 10^{-2}$



High energy = high density + high rate

**ALICE (CERN)**

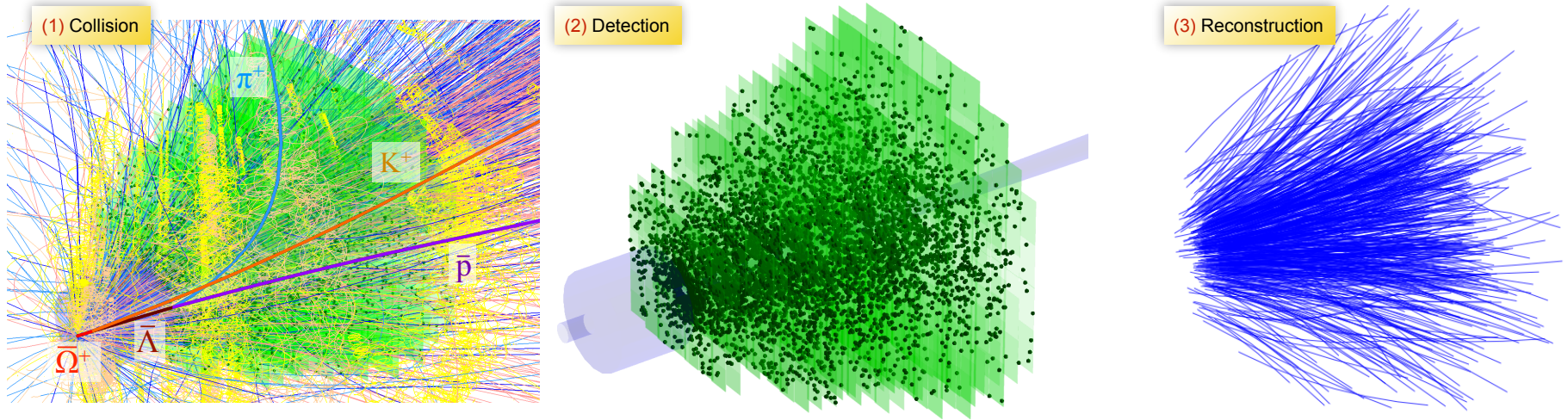
**CBM (FAIR/GSI)**



HEP Experiments: select interesting physics on-line



# Reconstruction challenge in CBM at FAIR/GSI

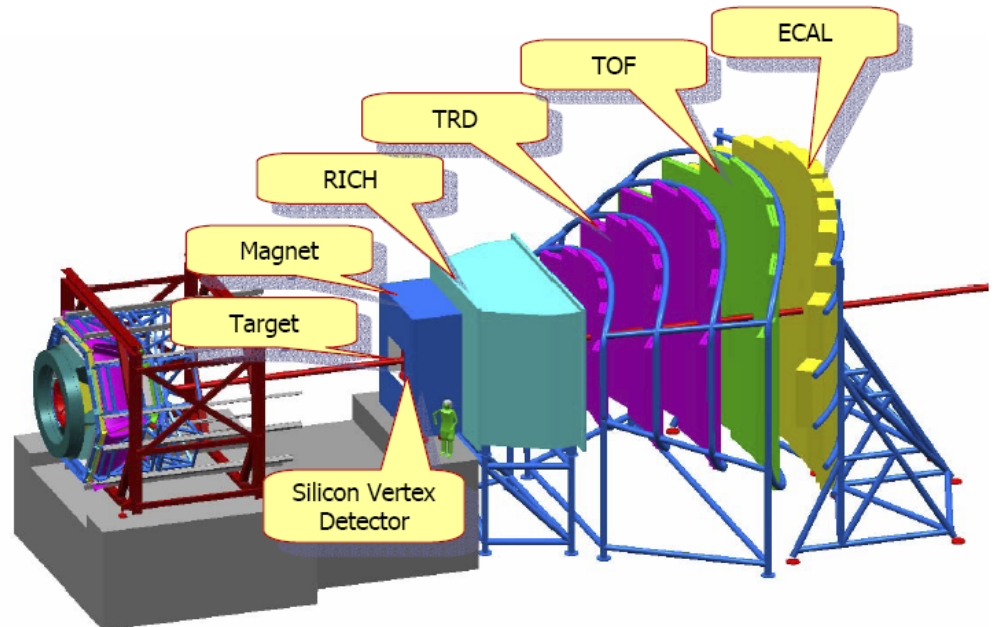


- Future **fixed-target heavy-ion** experiment
- $10^7$  Au+Au collisions/sec
- $\sim 1000$  charged **particles/collision**
- **Non-homogeneous** magnetic field
- **Double-sided strip** detectors (85% fake space-points)

Full event reconstruction will be done **on-line** at the First-Level Event Selection (**FLES**) and **off-line** using the same **FLES** reconstruction package.

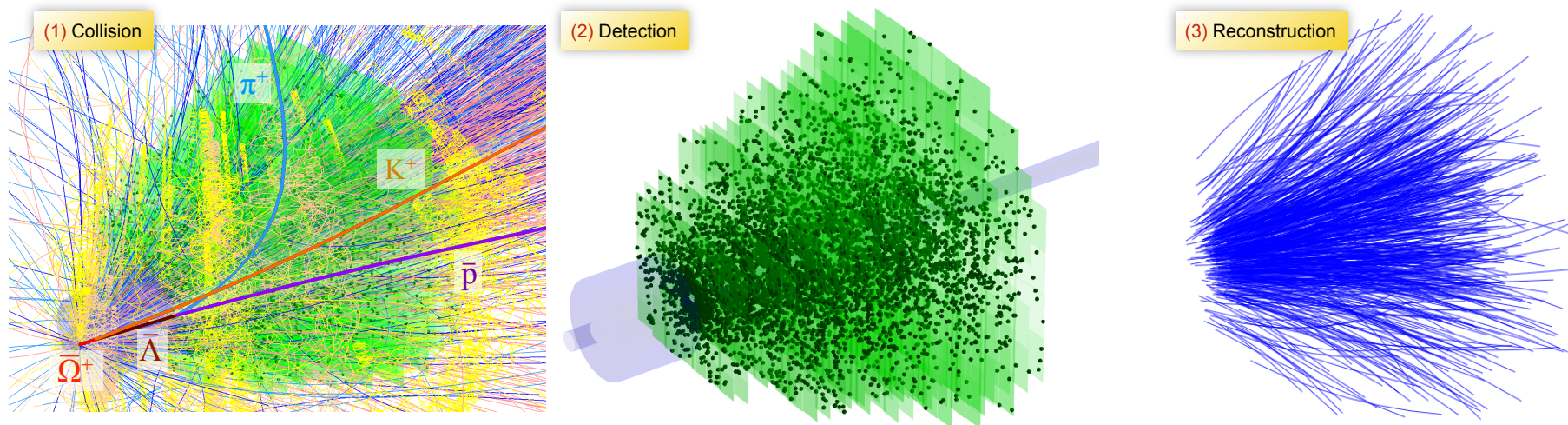
Cellular Automaton (CA) Track Finder  
Kalman Filter (KF) Track Fitter  
KF short-lived Particle Finder

All reconstruction algorithms are **vectorized** and **parallelized**.





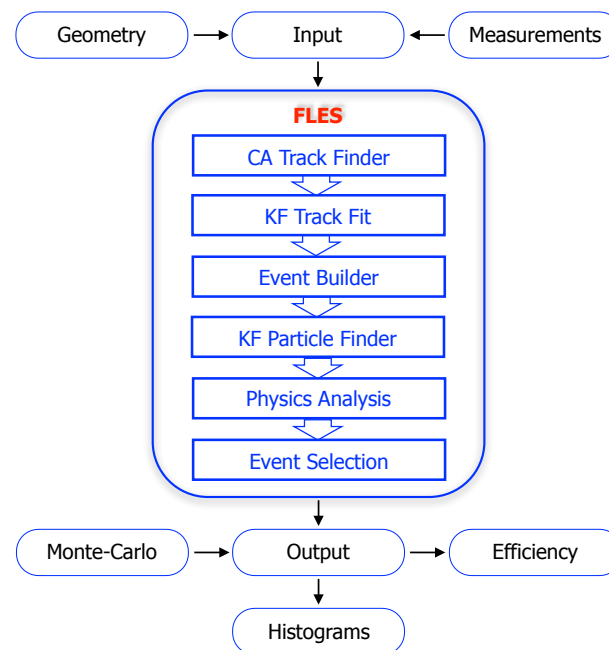
# Reconstruction challenge in CBM at FAIR/GSI



The full event reconstruction will be done **on-line** at the **First-Level Event Selection (FLES)** and **off-line** using the same **FLES** reconstruction package.

- Cellular Automaton (CA) Track Finder
- Kalman Filter (KF) Track Fitter
- KF short-lived Particle Finder

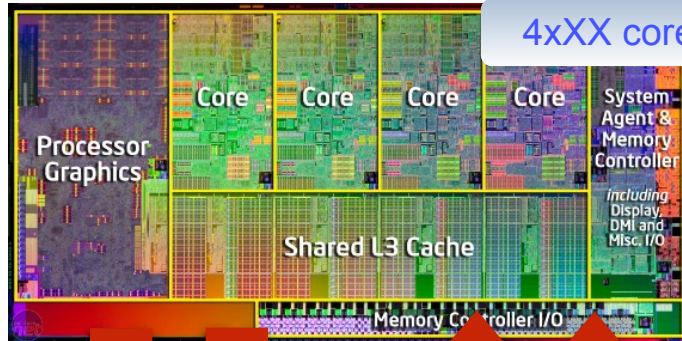
All reconstruction algorithms are **vectorized** and **parallelized**.





# Many-Core CPU/GPU Architectures

Intel/AMD CPU



Math

Memory

- Optimized for low latency access to cache data sets
- Control for out-of-order and speculative execution

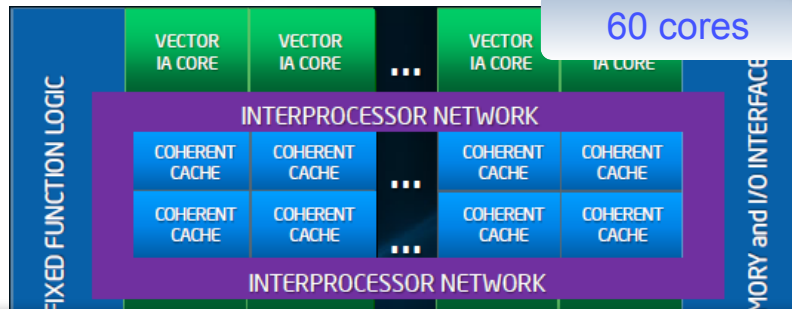
Parallelism

Math

Memory

#Cores

Intel Phi



Nvidia/ATI GPU

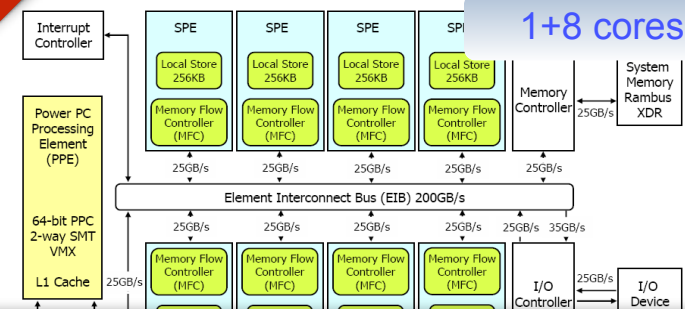


- Optimized for data-parallel, throughput computation
- More transistors dedicated to computation

Stability

Memory

IBM Cell



- General purpose RISC processor (PowerPC)
- 8 co-processors (SPE, Synergistic Processor Elements)
- 128-bit wide SIMD units

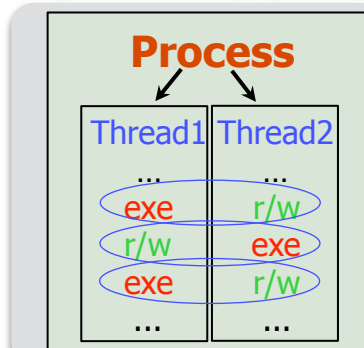
Future systems are heterogeneous. Fundamental redesign of traditional approaches to data processing is necessary



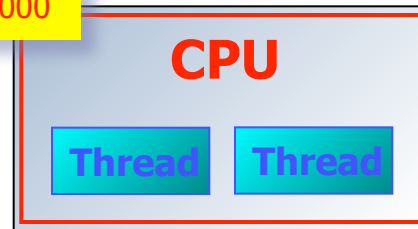
# Many-core HPC: Cores, Threads and Vectors

HEP experiments work with high data rates, therefore need High Performance Computing (HPC) !

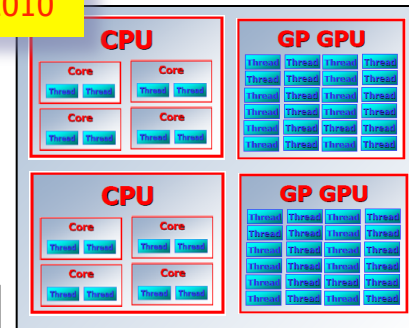
Cores and Threads = task level parallelism



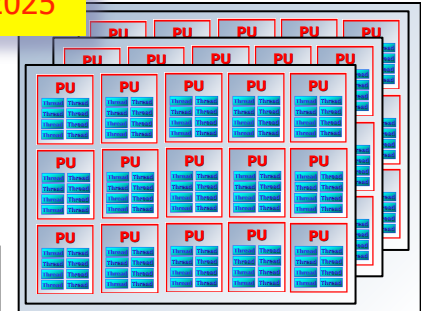
2000



2010



2025

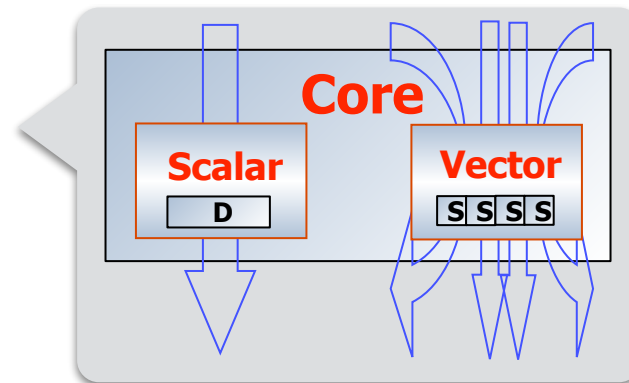


Some remarks:

- + hardware parallelism for free;
- many-core machine is not a batch farm;
- + use core/thread parallelism at the event level;
- scalar code is useless;
- + use vector programming;

Speed-up = (1 + HT) \* N cores \* N sockets \* SIMD width

S lxr075@gsi = (1 + 0.3) \* 10 \* 4 \* 4 = 200



SIMD = Single Instruction, Multiple Data

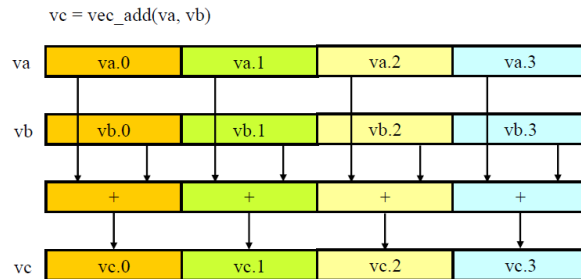
Vectors (SIMD) = data level parallelism

Fundamental redesign of traditional approaches to data processing is necessary



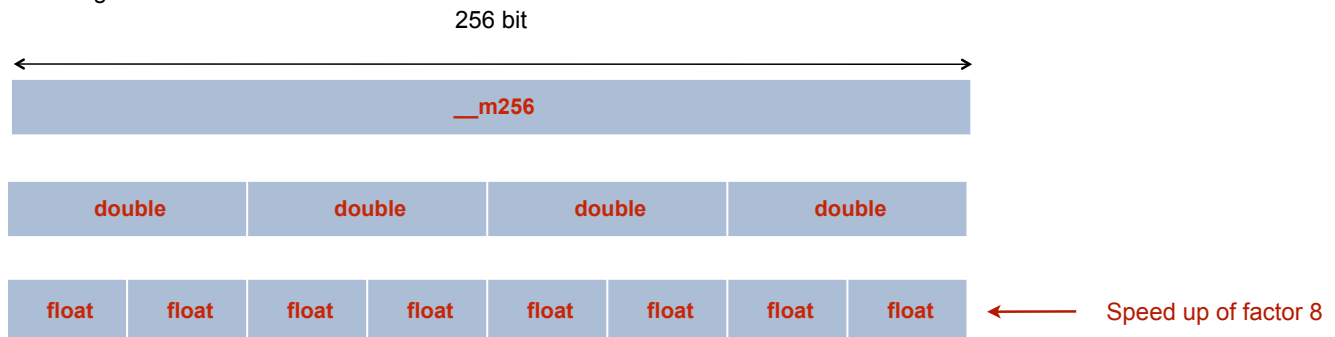
# SIMD Registers

$vc = va + vb$



```
void main()
{
    __m256  f;    // float[8]
    __m256d f;    // double[4]
    __m256i f;    // char[32], short int[16], int[8], uint64_t[4]
}
```

SIMD register:





# Basic Single Precision SIMD Instructions

## Arithmetic:

<code>_mm_add_ps (Va,Vb)</code>	$a + b$
<code>_mm_sub_ps (Va,Vb)</code>	$a - b$
<code>_mm_mul_ps (Va,Vb)</code>	$a * b$
<code>_mm_div_ps (Va,Vb)</code>	$a / b$

## Logical:

<code>_mm_and_ps (Va,Vb)</code>	$a \& b$
<code>_mm_or_ps (Va,Vb)</code>	$a   b$
<code>_mm_xor_ps (Va,Vb)</code>	$a \wedge b$

## Comparison:

<code>_mm_cmplt_ps (Va,Vb)</code>	$a < b$
<code>_mm_cmple_ps (Va,Vb)</code>	$a \leq b$
<code>_mm_cmpgt_ps (Va,Vb)</code>	$a > b$
<code>_mm_cmpge_ps (Va,Vb)</code>	$a \geq b$
<code>_mm_cmpeq_ps (Va,Vb)</code>	$a == b$

## Extra:

<code>_mm_min_ps (Va,Vb)</code>	$\min(a, b)$
<code>_mm_max_ps (Va,Vb)</code>	$\max(a, b)$
<code>_mm_rcp_ps (Va)</code>	$1/a$
<code>_mm_sqrt_ps (Va)</code>	$\sqrt{a}$
<code>_mm_rsqrt_ps (Va)</code>	$1/\sqrt{a}$

There are much more instructions, including instructions for SIMD vectors with double precision values.



# Wrapped C++ Headers

The instructions can be wrapped by C++ class, which overloads standard operators (+, -, \*, \, >, etc.) for the convenience of user.

## Partial source code of P4\_F32vec4.h

```
44 class F32vec4
45 {
46     public:
47
48     __m128 v;
49
50     float & operator[]( int i ){ return (reinterpret_cast<float*>(&v))[i]; }
51     float  operator[]( int i ) const { return (reinterpret_cast<const float*>(&v))[i]; }
52
53     F32vec4( ):v(_mm_set_ps1(0)){}
54     F32vec4( const __m128 &a ):v(a) {}
55     F32vec4( const float &a ):v(_mm_set_ps1(a)) {}
56
57     F32vec4( const float &f0, const float &f1, const float &f2, const float
&f3 ):v(_mm_set_ps(f3,f2,f1,f0)) {}
58
59     ...
60
61     /* Arithmetic Operators */
62     friend F32vec4 operator +(const F32vec4 &a, const F32vec4 &b) { return _mm_add_ps(a,b); }
63
64     ...
65
66     /* Square Root */
67     friend F32vec4 sqrt ( const F32vec4 &a ){ return _mm_sqrt_ps (a); }
68
69     ...
70
71     } __attribute__ ((aligned(16)));
72
73
74     typedef F32vec4 fvec;
75     typedef float  fscal;
76     const int fvecLen = 4;
```



# Headers - overloading operators

```
typedef F32vec4 Fvec_t;
/* Arithmetic Operators */
friend F32vec4 operator +( const F32vec4 &a, const F32vec4 &b ) { return _mm_add_ps(a, b); }
friend F32vec4 operator -( const F32vec4 &a, const F32vec4 &b ) { return _mm_sub_ps(a, b); }
friend F32vec4 operator *( const F32vec4 &a, const F32vec4 &b ) { return _mm_mul_ps(a, b); }
friend F32vec4 operator /( const F32vec4 &a, const F32vec4 &b ) { return _mm_div_ps(a, b); }
/* Functions */
friend F32vec4 min( const F32vec4 &a, const F32vec4 &b ){ return _mm_min_ps(a, b); }
friend F32vec4 max( const F32vec4 &a, const F32vec4 &b ){ return _mm_max_ps(a, b); }
/* Square Root */
friend F32vec4 sqrt( const F32vec4 &a ){ return _mm_sqrt_ps (a); }
/* Absolute value */
friend F32vec4 fabs( const F32vec4 &a ){ return _mm_and_ps(a, _f32vec4_abs_mask); }
/* Logical */
friend F32vec4 operator&( const F32vec4 &a, const F32vec4 &b ){ // mask returned
    return _mm_and_ps(a, b);
}
friend F32vec4 operator|( const F32vec4 &a, const F32vec4 &b ){ // mask returned
    return _mm_or_ps(a, b);
}
friend F32vec4 operator^( const F32vec4 &a, const F32vec4 &b ){ // mask returned
    return _mm_xor_ps(a, b);
}
friend F32vec4 operator!( const F32vec4 &a ){ // mask returned
    return _mm_xor_ps(a, _f32vec4_true);
}
friend F32vec4 operator||( const F32vec4 &a, const F32vec4 &b ){ // mask returned
    return _mm_or_ps(a, b);
}
/* Comparison */
friend F32vec4 operator<( const F32vec4 &a, const F32vec4 &b ){ // mask returned
    return _mm_cmplt_ps(a, b);
}
```

SIMD instructions



# Code (Part of the Kalman Filter)

```
inline void AddMaterial( TrackV &track, Station &st, Fvec_t &qp0 )
{
    const mass2 = 0.1396*0.1396;
```

```
    Fvec_t tx = track.T[2];
    Fvec_t ty = track.T[3];
    Fvec_t txtx = tx*tx;
    Fvec_t tyty = ty*ty;
    Fvec_t txtx1 = txtx + ONE;
    Fvec_t h = txtx + tyty;
    Fvec_t t = sqrt(txtx1 + tyty);
    Fvec_t h2 = h*h;
    Fvec_t qp0t = qp0*t;
```

```
    const c1=0.0136, c2=c1*0.038, c3=c2*0.5, c4=-c3/2.0, c5=c3/3.0, c6=-c3/4.0;
```

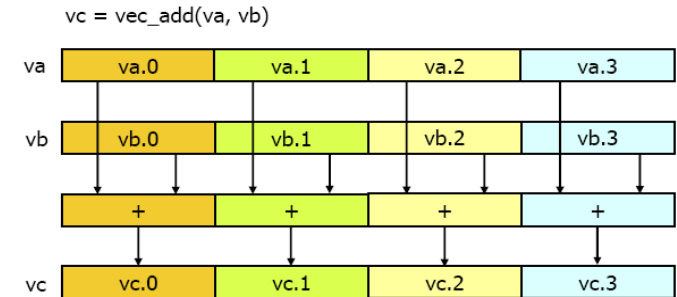
```
    Fvec_t s0 = (c1+c2*st.logRadThick + c3*h + h2*(c4 + c5*h + c6*h2) )*qp0t;
```

```
    Fvec_t a = (ONE+mass2*qp0*qp0t)*st.RadThick*s0*s0;
```

```
    CovV &C = track.C;
```

```
    C.C22 += txtx1*a;
    C.C32 += tx*ty*a; C.C33 += (ONE+tyty)*a;
```

```
}
```



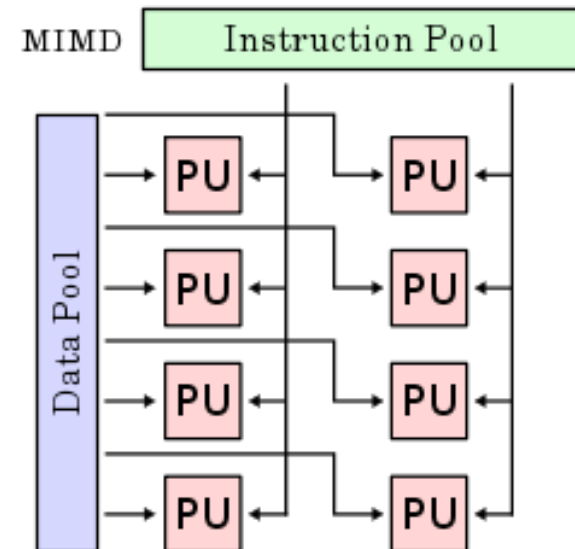
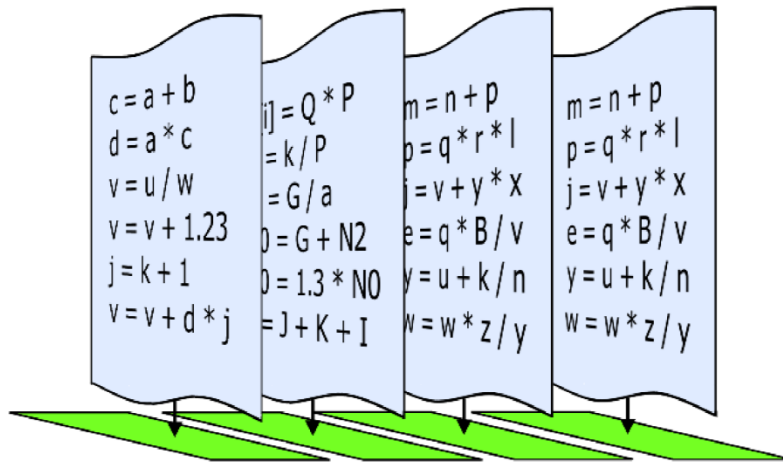
Use of headers to vectorize the code

# Task Parallelism

Parallelization between cores → Task Parallelism, Parallelization, MIMD

Tools:

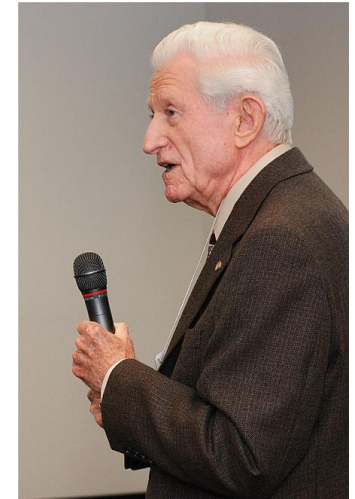
- OpenMP
- OpenCL
- Intel Threading Building Blocks (ITBB)
- Pthreads
- Intel Cilk
- Intel Array Building Blocks
- MPI





# Performance Characterization

**Gene Myron Amdahl** (born November 16, 1922) is an American computer architect and high-tech entrepreneur, chiefly known for his work on mainframe computers at **IBM (System/360)** and later his own companies, especially Amdahl Corporation. He formulated **Amdahl's law**, which states a fundamental limitation of parallel computing.

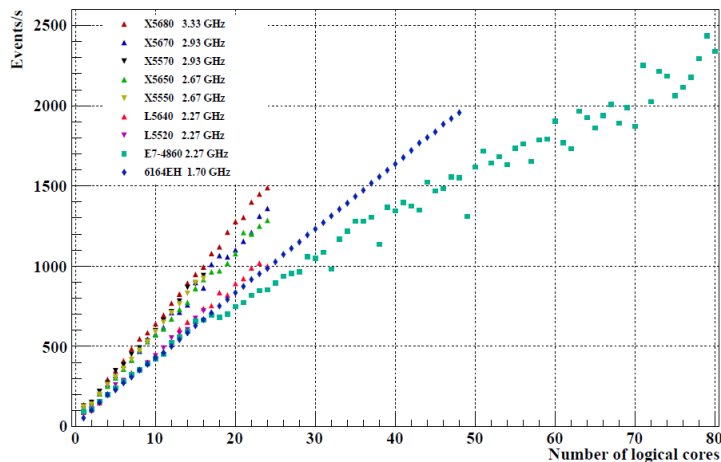


- Performance increasing is characterized with speedup factor
- In ideal case – perfect linear speedup
- Super-linear speedup (usually cache effect)
- Speedup in presence of a serial part
- Number of CPUs is infinite ( $P \rightarrow \infty$ ) – the maximum speedup (**Amdahl's law**)

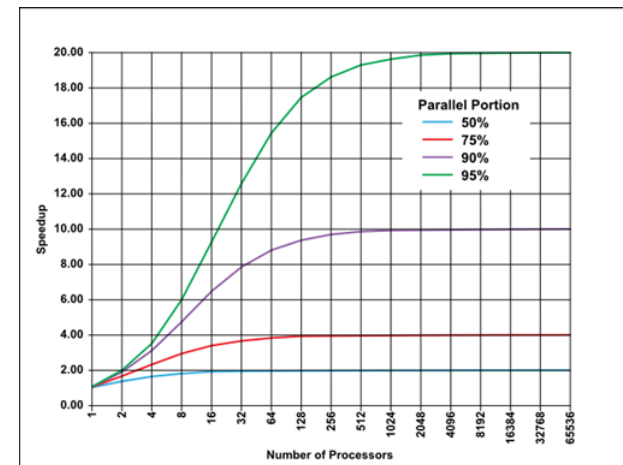
$$S(P) = \frac{\text{Time}(1)}{\text{Time}(P)} \quad \begin{matrix} S(P) = P \\ S(P) > P \end{matrix}$$

$$S(P) = \frac{1}{\alpha + \frac{1-\alpha}{P}} \quad S(P) = \frac{1}{\alpha}$$

Real-life example:  
scalability of the CBM track finder on different platforms



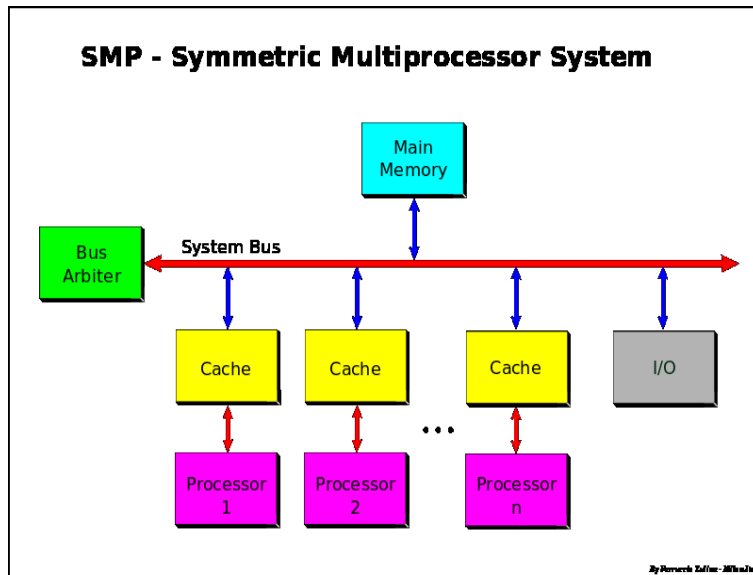
Amdahl's law



# CPU systems

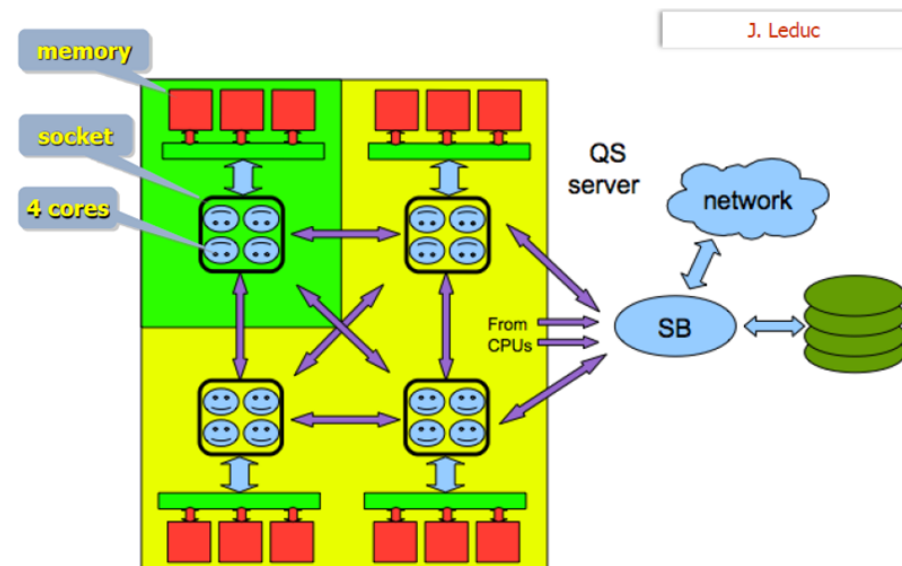
SMP (symmetric multiprocessor) systems:

- Homogeneous
- “Equal-time” access for each processor to any part of the memory



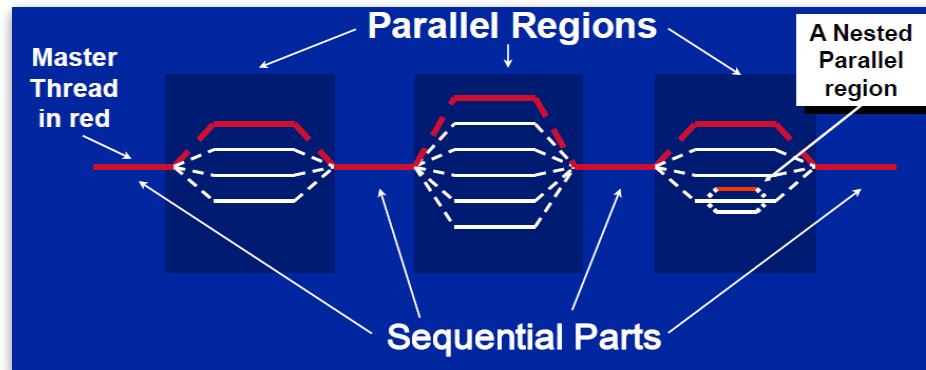
NUMA (non uniform memory access) systems:

- Heterogeneous
- Non uniform access to different parts of the main memory – different speed, data should be close to the CPU





# OpenMP



- OpenMP (Open Multi-Processing) technology is the most popular technology of parallelization nowadays.
- A sequential program is taken as a basis and a set of compiler directives, environment variables and library procedures intended for programming multithreaded applications on multiprocessor systems with shared memory are used for its parallelization.
- Memory in OpenMP is divided into: local and shared memory. Local memory is only available for one thread, and shared memory is available for several threads.
- OpenMP gives an opportunity to use multiprocessor computing systems with shared memory most efficiently, providing the possibility to use shared data for parallel executed threads without any difficulties for inter-processor data transfers.
- OpenMP allows incremental (step-by-step) development of parallel programs. It means that at the early stages of development you can already get a parallel program, due to OpenMP directives, which can be added into a sequential program step by step.
- A program created with the use of OpenMP technology consists of sequential (single-threaded) and parallel (multi-threaded) sections.
- In OpenMP the parallelization model „Fork - Join“ is used. At first, there is only a single thread called the initial thread or the main thread or master thread (thread is a lightweight process). As soon as a thread meets a parallel construction in the program code, it creates a group of threads and becomes the main thread. In the created group all the threads, including the main one, execute the program code. After executing a parallel construction in the code, only the main thread continues to work.

# OpenMP - HelloWorld

```
#include <omp.h>
#include <iostream>
using namespace std;

int main() {

    #pragma omp parallel num_threads(10)
    {
        int id = 0;

        cout << " Hello world " << id << endl;
    }

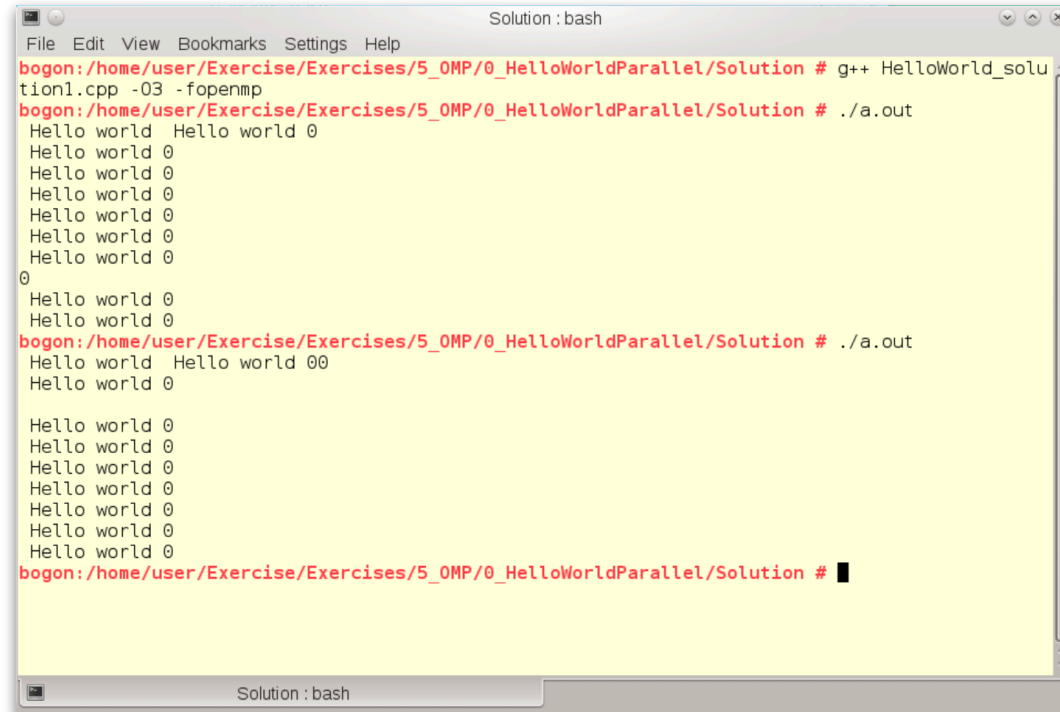
    return 0;
}
```

Heloworld program parallelized with OpenMP for 10 threads.

omp.h header must be included to operate with OpenMP constructs.



# OpenMP - HelloWorld



```
Solution : bash
File Edit View Bookmarks Settings Help
bogon:/home/user/Exercise/Exercises/5_OMP/0_HelloWorldParallel/Solution # g++ HelloWorld_solution1.cpp -O3 -fopenmp
bogon:/home/user/Exercise/Exercises/5_OMP/0_HelloWorldParallel/Solution # ./a.out
Hello world Hello world 0
Hello world 0
Hello world 0
Hello world 0
Hello world 0
Hello world 0
Hello world 0
0
Hello world 0
Hello world 0
bogon:/home/user/Exercise/Exercises/5_OMP/0_HelloWorldParallel/Solution # ./a.out
Hello world Hello world 00
Hello world 0

Hello world 0
Hello world 0
Hello world 0
Hello world 0
Hello world 0
Hello world 0
Hello world 0
bogon:/home/user/Exercise/Exercises/5_OMP/0_HelloWorldParallel/Solution # █
```

Such a program will print symbols on the screen chaotically, because all 10 threads try to do this at the same time and only one of them can print at the current moment. To prevent such situation the threads should be synchronized.

# OpenMP - HelloWorld

```
#include <omp.h>
#include <iostream>
using namespace std;

int main() {

    #pragma omp parallel num_threads(10)
    {
        int id = omp_get_thread_num();

        #pragma omp critical
        cout << " Hello world " << id << endl;
    }

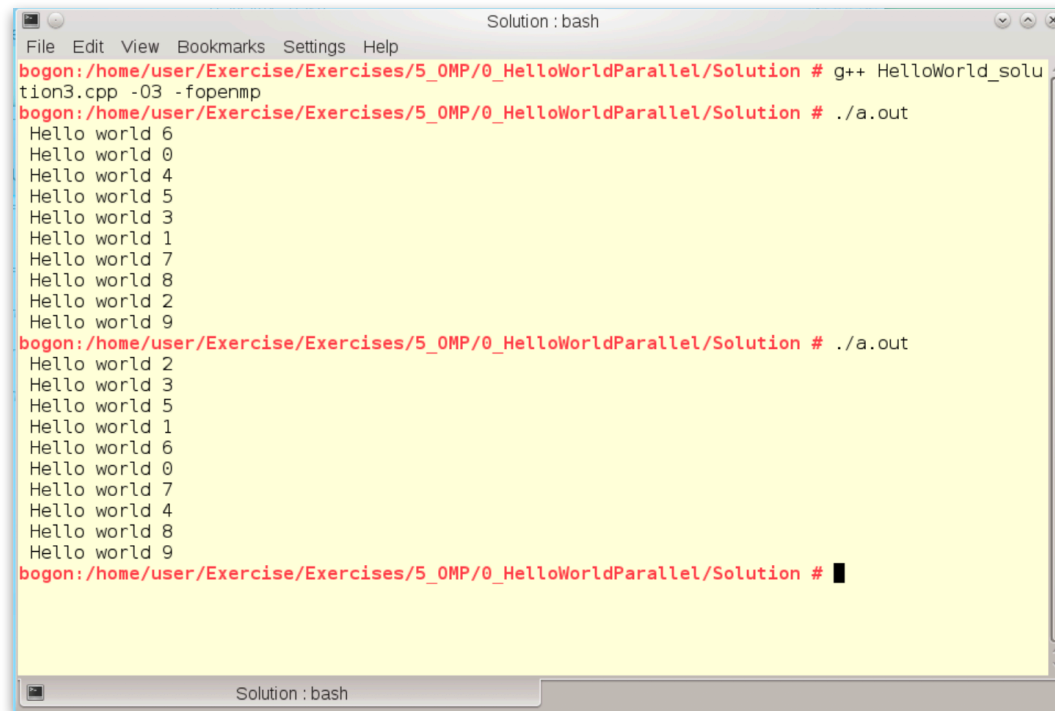
    return 0;
}
```

`omp critical` can be added to the parallel region just before the printing to synchronise threads.

To obtain the id of the thread the function `omp_get_thread_num( )` should be used. Each thread has its own id, therefore the function should be called in the parallel region by each thread individually. Then id is stored to the local variable of each thread and printed on the screen.



# OpenMP - HelloWorld

A terminal window titled "Solution : bash" with a menu bar (File, Edit, View, Bookmarks, Settings, Help). The terminal shows the compilation and execution of a C++ program using OpenMP. The first run shows 10 "Hello world" messages in a non-sequential order. The second run shows another 10 "Hello world" messages in a different non-sequential order.

```
Solution : bash
File Edit View Bookmarks Settings Help
bogon:/home/user/Exercise/Exercises/5_OMP/0_HelloWorldParallel/Solution # g++ HelloWorld_solution3.cpp -O3 -fopenmp
bogon:/home/user/Exercise/Exercises/5_OMP/0_HelloWorldParallel/Solution # ./a.out
Hello world 6
Hello world 0
Hello world 4
Hello world 5
Hello world 3
Hello world 1
Hello world 7
Hello world 8
Hello world 2
Hello world 9
bogon:/home/user/Exercise/Exercises/5_OMP/0_HelloWorldParallel/Solution # ./a.out
Hello world 2
Hello world 3
Hello world 5
Hello world 1
Hello world 6
Hello world 0
Hello world 7
Hello world 4
Hello world 8
Hello world 9
bogon:/home/user/Exercise/Exercises/5_OMP/0_HelloWorldParallel/Solution # █
```

Now threads are synchronised, but they order stays unpredictable and would be different for every run.

# OpenCL - Host Program Structure

Program:  
create

```
// Returns the OpenCL program
cl_program clCreateProgramWithSource (cl_context context,
                                     cl_uint count,           // number of files
                                     const char **strings,      // array of strings, each one is a file
                                     const size_t *lengths,     // array specifying the file lengths
                                     cl_int *errcode_ret)       // error code to be returned
```

Program:  
build

```
cl_int clBuildProgram (cl_program program,
                      cl_uint num_devices,
                      const cl_device_id *device_list,
                      const char *options, // Compiler options, see specifications
                      void (*pfn_notify)(cl_program, void *user_data),
                      void *user_data)
```

Error log:

```
cl_int clGetProgramBuildInfo (cl_program program,
                              cl_device_id device,
                              cl_program_build_info param_name, // The parameter we want to know
                              size_t param_value_size,
                              void *param_value, // The answer
                              size_t *param_value_size_ret)
                              CL_PROGRAM_BUILD_STATUS
                              CL_PROGRAM_BUILD_OPTIONS
                              CL_PROGRAM_BUILD_LOG
```

Kernel:  
create

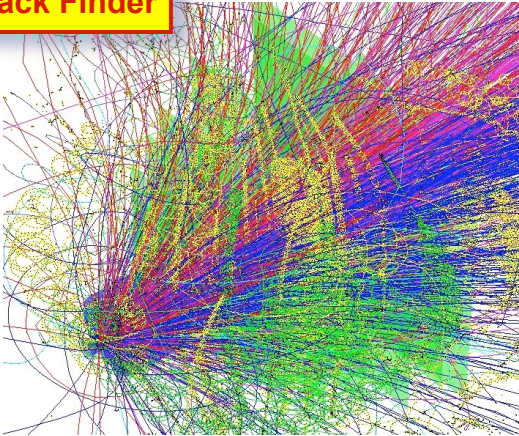
```
cl_kernel clCreateKernel (cl_program program, // The program where the kernel is
                          const char *kernel_name, // The name of the kernel
                          cl_int *errcode_ret)
```



# Stages of Event Reconstruction

1

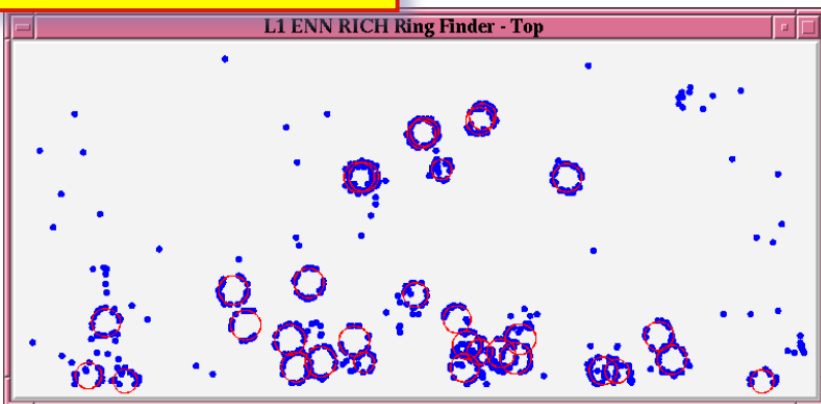
// Track Finder



- Conformal Mapping
- Hough Transformation
- Track Following
- **Cellular Automaton**

3

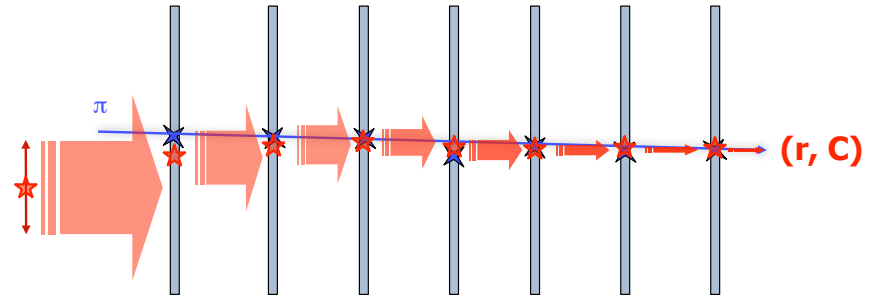
// Ring Finder (Particle ID)



- Hough Transformation
- Elastic Neural Net

2

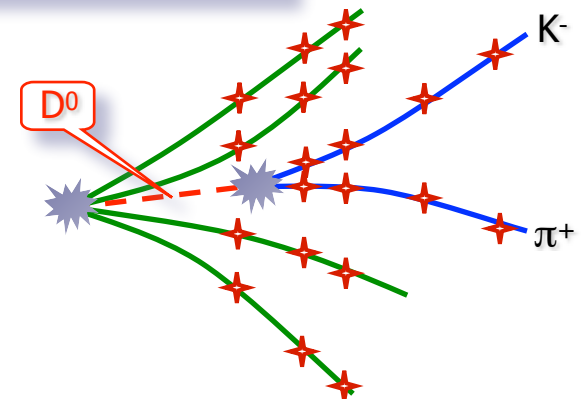
// Track Fitter



- Kalman Filter

4

// Short-Lived Particles Finder



- Kalman Filter

# Cellular Automaton (CA) Track Finder

0. Hits (CBM)

1000 Hits

0. Hits

Detector layers

Hits

1. Segments

2. Counters

3. Track Candidates

4. Tracks

Cellular Automaton:

1. Build short track segments.
2. Connect according to the track model, estimate a possible position on a track.
3. Tree structures appear, collect segments into track candidates.
4. Select the best track candidates.

Cellular Automaton:

- local w.r.t. data
- intrinsically parallel
- extremely simple
- very fast

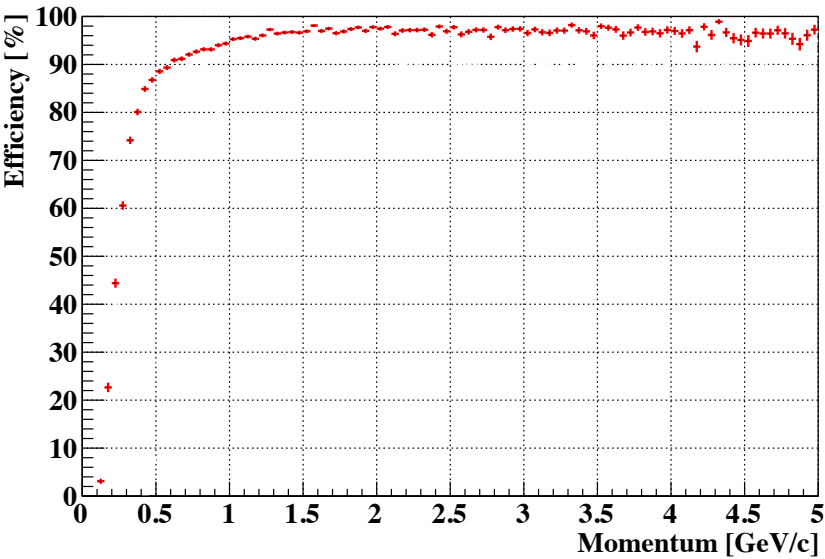
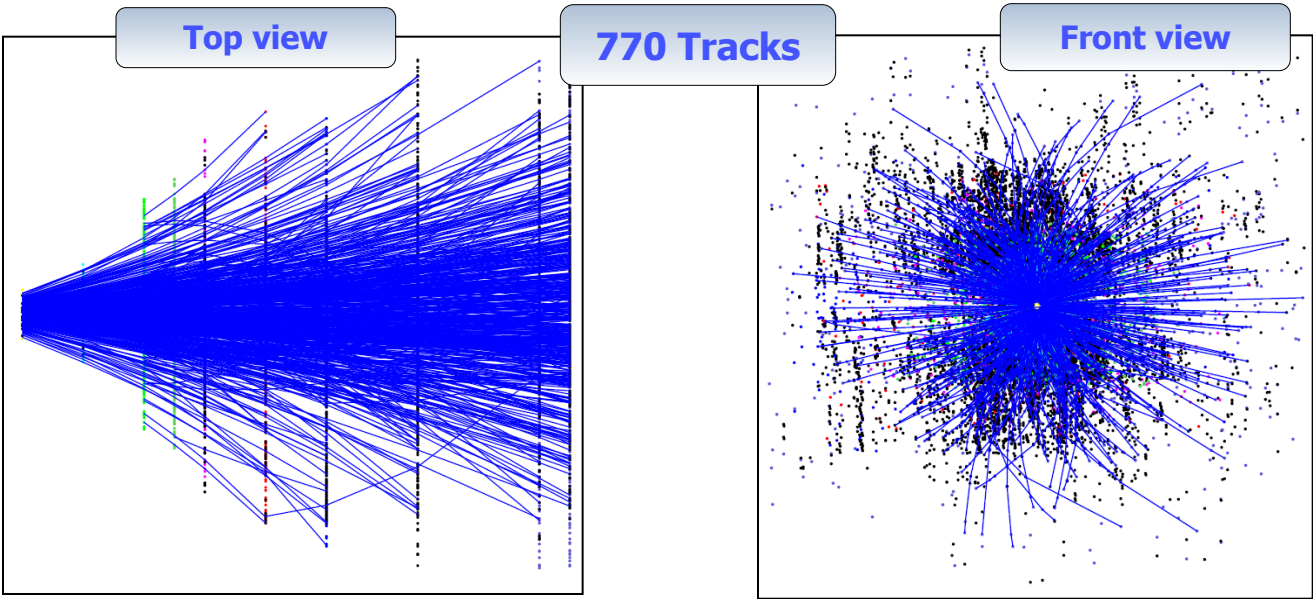
4. Tracks (CBM)

1000 Tracks

Deeply appropriate for many-core CPU/GPU

Useful for complicated event topologies with heavy combinatorics

# Cellular Automaton (CA) Track Finder



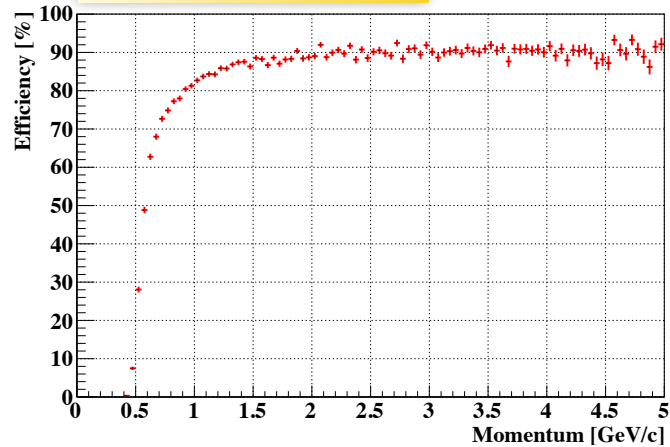
Track category	Eff, %
All tracks	90.9
Primary high- <i>p</i>	97.5
Primary low- <i>p</i>	92.6
Secondary high- <i>p</i>	91.1
Secondary low- <i>p</i>	63.8
Clone level	0.4
Ghost level	5.9
MC tracks found	134
Time, ms/ev	10

Fast and efficient track finder



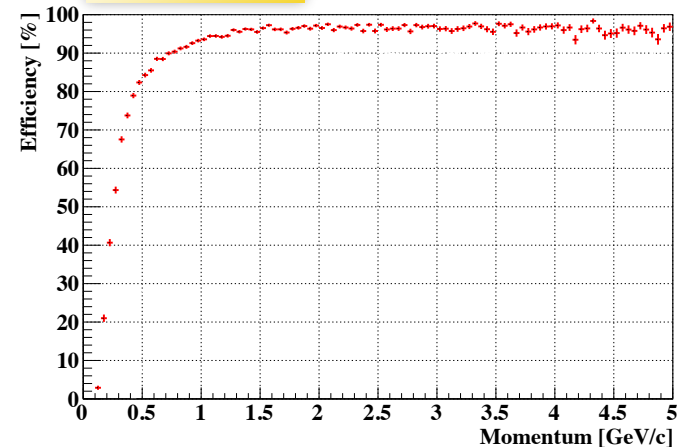
# CBM CA Track Finder

(1) high-momentum primary tracks



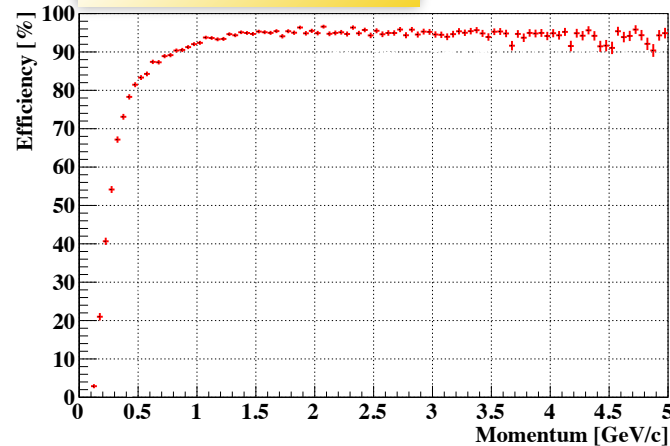
Track category	Eff. %
All tracks	70.4
Primary high- $p$	94.9
Primary low- $p$	56.8
Secondary high- $p$	49.7
Secondary low- $p$	13.0
Clone level	0.3
Ghost level	0.3
MC tracks found	103
Time, ms/ev	4

(3) secondary tracks



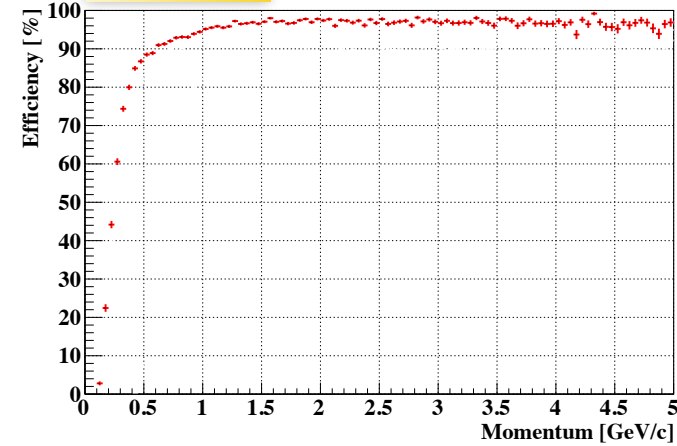
Track category	Eff. %
All tracks	89.2
Primary high- $p$	97.5
Primary low- $p$	92.4
Secondary high- $p$	86.6
Secondary low- $p$	54.7
Clone level	1.0
Ghost level	5.5
MC tracks found	131
Time, ms/ev	7

(2) low-momentum primary tracks



Track category	Eff. %
All tracks	87.8
Primary high- $p$	95.8
Primary low- $p$	91.4
Secondary high- $p$	84.5
Secondary low- $p$	54.2
Clone level	0.9
Ghost level	5.6
MC tracks found	129
Time, ms/ev	6

(4) broken tracks

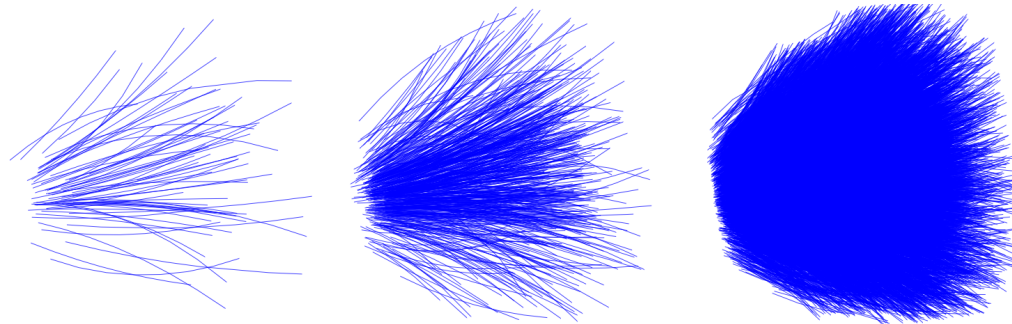


Track category	Eff. %
All tracks	90.9
Primary high- $p$	97.5
Primary low- $p$	92.6
Secondary high- $p$	91.1
Secondary low- $p$	63.8
Clone level	1.0
Ghost level	5.9
MC tracks found	134
Time, ms/ev	8

Efficient and stable event reconstruction

# CA Track Finder at High Track Multiplicity

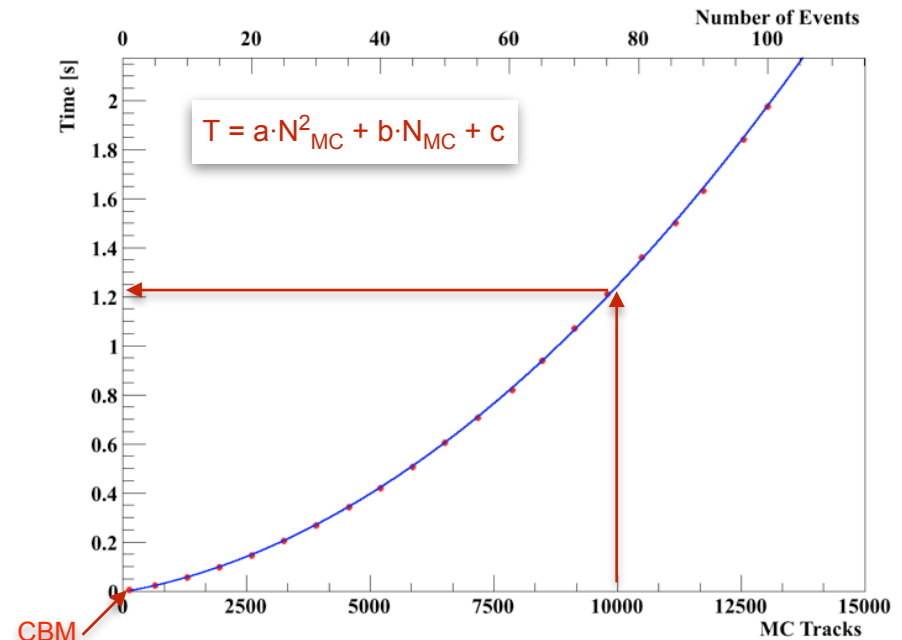
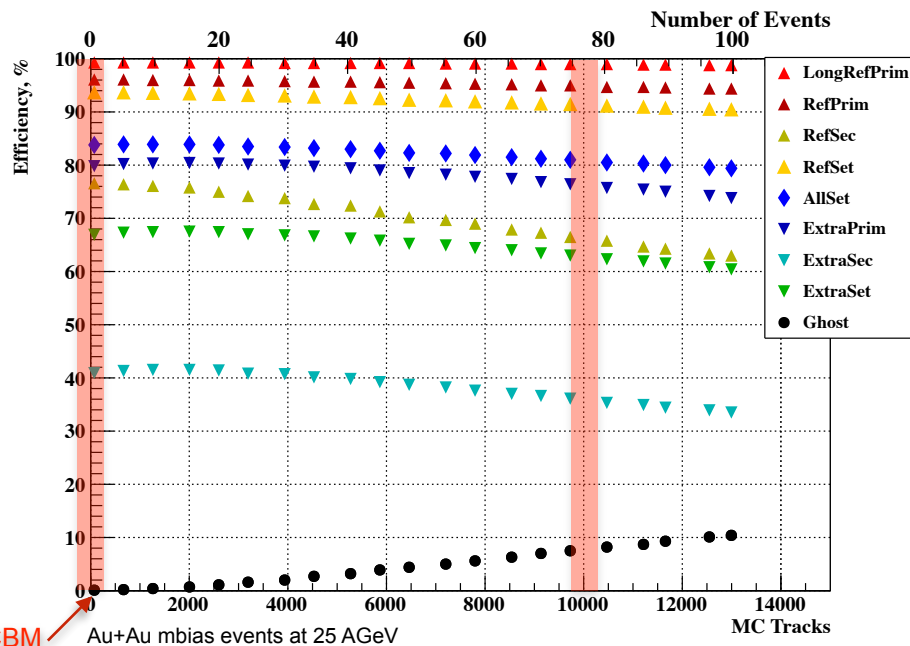
A number of minimum bias events is gathered into a group (super-event), which is then treated by the CA track finder as a single event.



1 mbias event,  $\langle N_{\text{reco}} \rangle = 109$

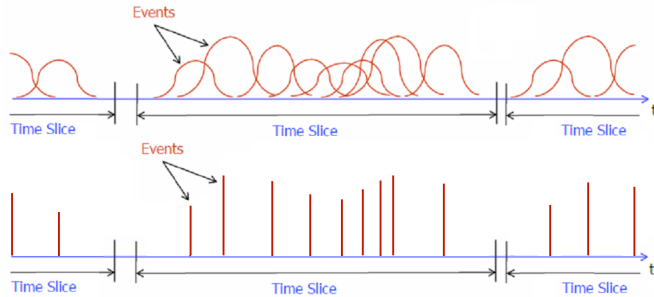
5 mbias events,  $\langle N_{\text{reco}} \rangle = 572$

100 mbias events,  $\langle N_{\text{reco}} \rangle = 10340$



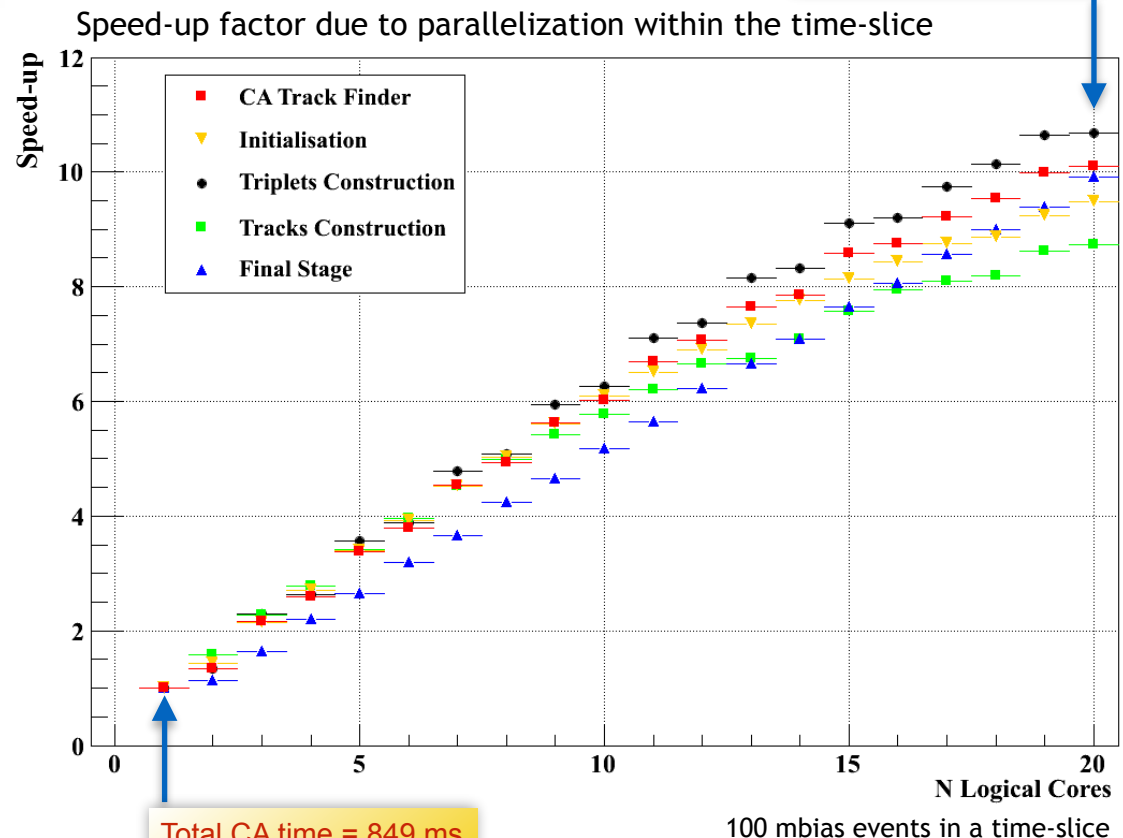
Reliable reconstruction efficiency and time as a second order polynomial w.r.t. to the track multiplicity

# Time-based (4D) Track Reconstruction



- The **beam** in the CBM will have **no bunch structure**, but continuous.
- Measurements in this case will be **4D** ( $x, y, z, t$ ).
- Significant **overlapping of events** in the detector system.
- Reconstruction of **time slices** rather than events is needed.

Efficiency, %	3D	4D
All tracks	83.8	83.0
Primary high- $p$	96.1	92.8
Primary low- $p$	79.8	83.1
Secondary high- $p$	76.6	73.2
Secondary low- $p$	40.9	36.8
Clone level	0.4	1.7
Ghost level	0.1	0.3
Time/event/core, ms	8.2	8.5



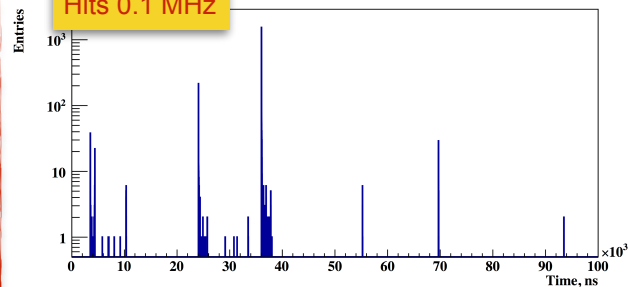
The reconstruction time 8.2 ms/event in 3D is recovered in 4D case as well



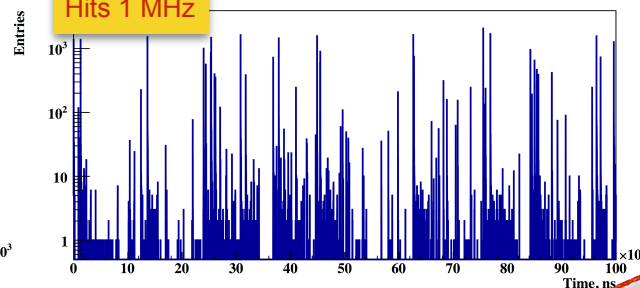
# 4D Event Building at 10 MHz

## Hits at high input rates

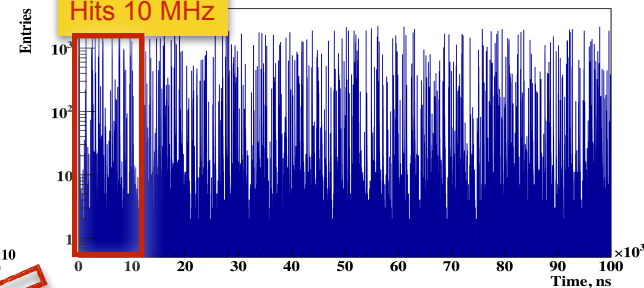
Hits 0.1 MHz



Hits 1 MHz

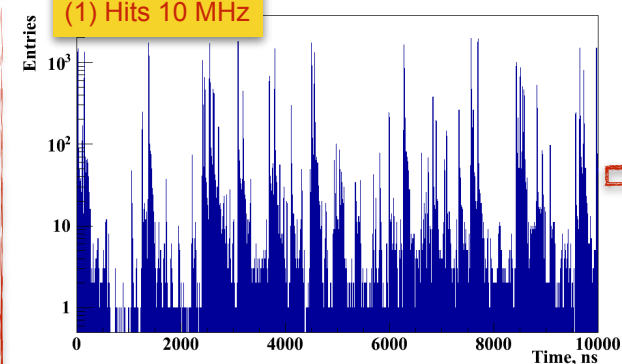


Hits 10 MHz

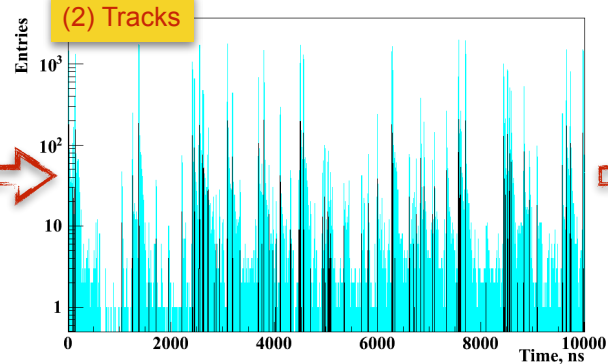


## From hits to tracks to events

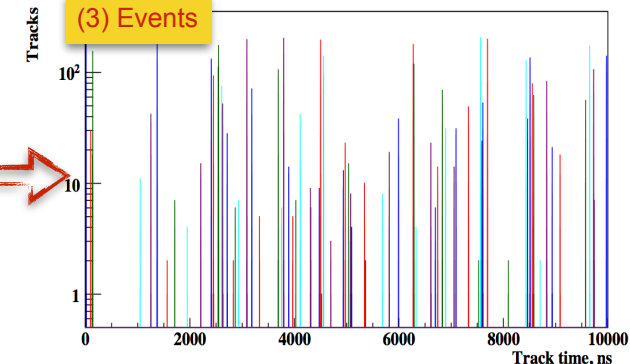
(1) Hits 10 MHz



(2) Tracks



(3) Events



Reconstructed tracks clearly represent groups, which correspond to the original events

# Kalman Filter Algorithm

The Kalman filter is a **recursive** estimator – only the estimated state from the previous time step and the current measurement are needed to compute the estimate for the current state.

mean value over  $n$  measurements



$$\mu_n = \frac{1}{n} \sum_{i=1}^n x_i$$



$$\mu_{n+1} = \frac{1}{n+1} \sum_{i=1}^{n+1} x_i = \frac{n}{n+1} \left( \frac{1}{n} \sum_{i=1}^n x_i + \frac{1}{n} x_{n+1} \right) = \frac{n}{n+1} \mu_n + \frac{1}{n+1} x_{n+1} = \mu_n + \frac{1}{n+1} (x_{n+1} - \mu_n)$$

mean value over  $n+1$  measurements

previous estimation

new measurement

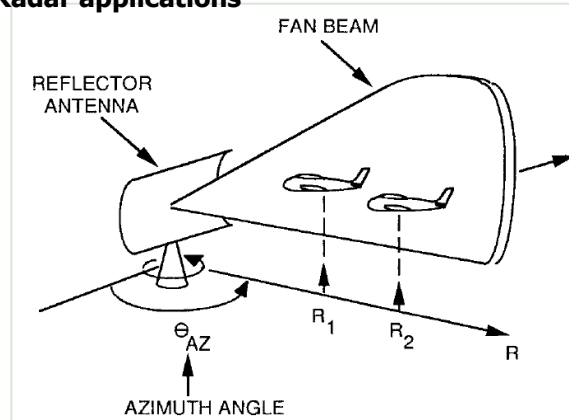
weight

correction

For this work, U.S. President **Barack Obama** rewarded Rudolf Kálmán with the **National Medal of Science** on October 7, 2009.



## Radar applications



state vector:

$$\mathbf{r} = \{ x, y, z, v_x, v_y, v_z \}$$

covariance matrix:

$$\mathbf{C} = \begin{Bmatrix} \sigma_x^2 & & & & & \\ & \sigma_y^2 & & & & \\ & & \sigma_z^2 & & & \\ & & & \sigma_{v_x}^2 & & \\ & & & & \sigma_{v_y}^2 & \\ & & & & & \sigma_{v_z}^2 \end{Bmatrix}$$

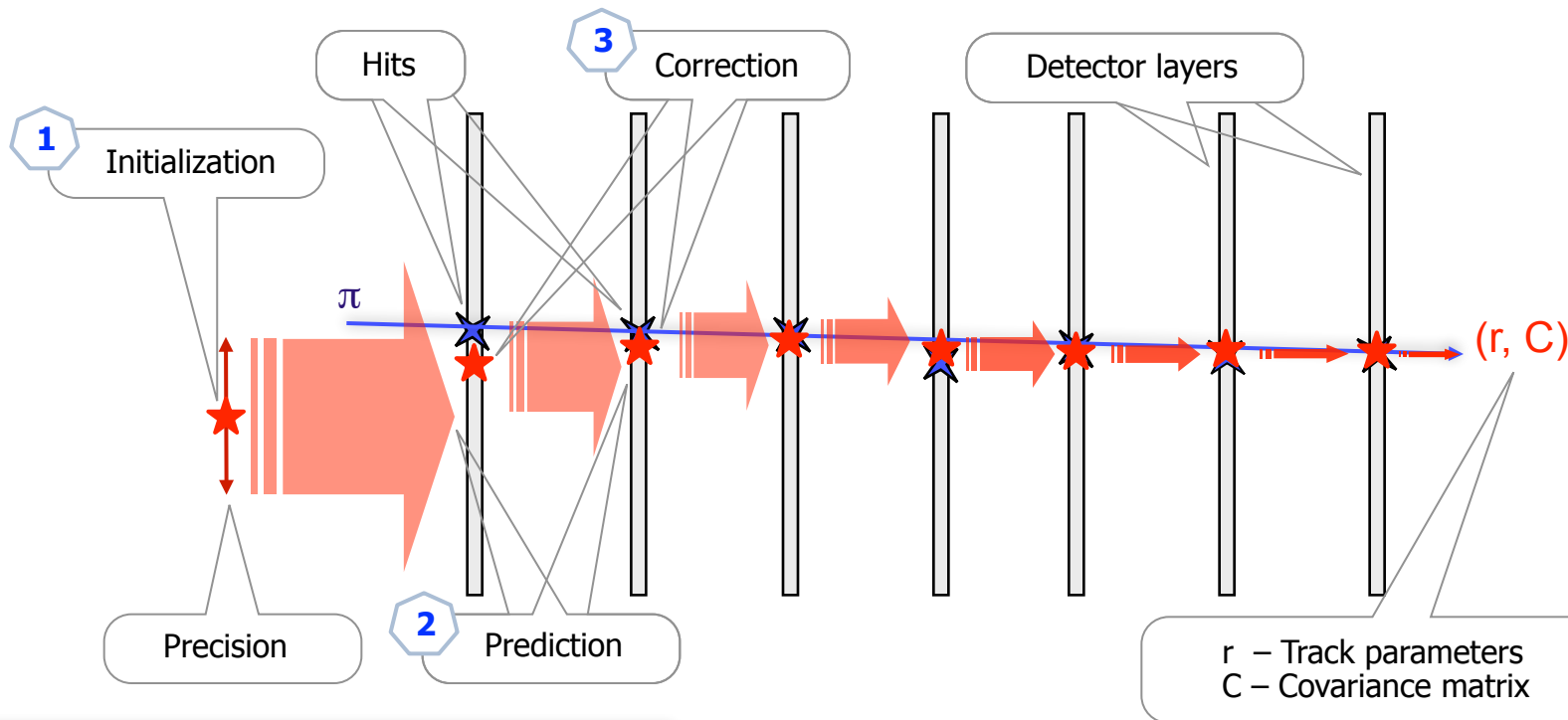
**Apollo Flight Journal**

December 21, 1968. The Apollo 8 spacecraft has just been sent on its way to the Moon.

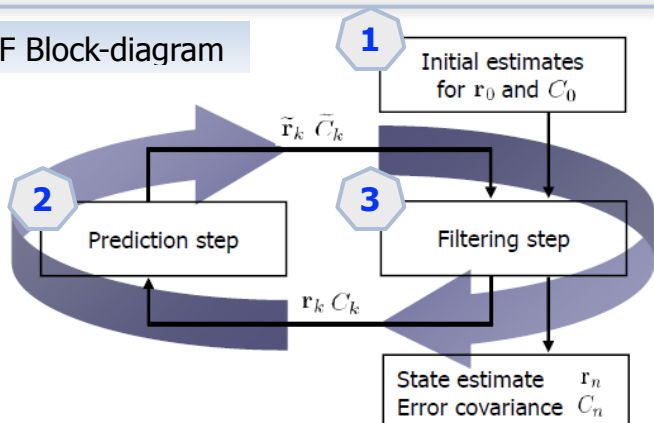
**003:46:31 Collins:** Roger. At your convenience, would you please go P00 and Accept? We're going to update to your W-matrix.

# Kalman Filter (KF) based Track Fit

Estimation of the track parameters at one or more hits along the track – Kalman Filter (KF)



## KF Block-diagram



KF as a recursive least squares method

State vector

Position, direction and momentum

$$r = \{x, y, z, p_x, p_y, p_z\}$$

Kalman Filter:

1. Start with an arbitrary initialization.
2. Add one hit after another.
3. Improve the state vector.
4. Get the optimal parameters after the last hit.

Nowadays the Kalman Filter is used in almost all HEP experiments



# Kalman Filter Track Fit on Cell

Stage	Description	Time/track	Speedup
Intel	Initial scalar version	12 ms	–
	1 Approximation of the magnetic field	240 $\mu$ s	50
	2 Optimization of the algorithm	7.2 $\mu$ s	35
	3 Vectorization	1.6 $\mu$ s	4.5
	4 Porting to SPE	1.1 $\mu$ s	1.5
Cell	5 Parallelization on 16 SPEs	0.1 $\mu$ s	10
	Final simdized version	0.1 $\mu$ s	120000

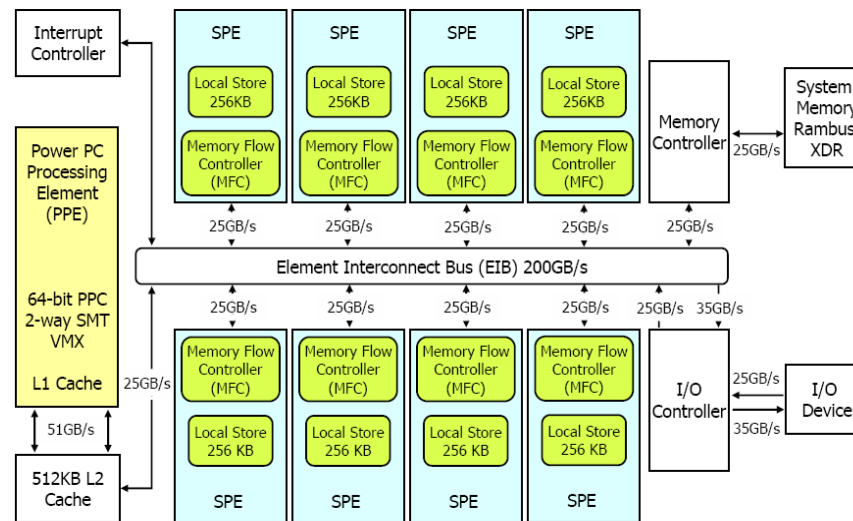
10000x faster  
on any PC

The KF speed was increased by 5 orders of magnitude

Comp. Phys. Comm. 178 (2008) 374-383



blade11bc4 @IBM, Böblingen:  
2 Cell Broadband Engines, 256 kB LS, 2.4 GHz



Emulator  
Assembler

Motivated by, but not restricted to Cell !

# Kalman Filter Track Fit Library

## Kalman Filter Methods

### Kalman Filter Tools:

- KF Track Fitter
- KF Track Smoother
- Deterministic Annealing Filter

### Kalman Filter Approaches:

- Conventional DP KF
- Conventional SP KF
- Square-Root SP KF
- UD-Filter SP
- Gaussian Sum Filter
- 3D (x,y,z) and 4D (x,y,z,t) KF

### Track Propagation:

- Runge-Kutta
- Analytic Formula

### Detector Types:

- Pixel
- Strip
- Tube
- TPC

## Implementations

### Vectorization (SIMD):

- Header Files
- Vc Vector Classes
- ArBB Array Building Blocks
- OpenCL

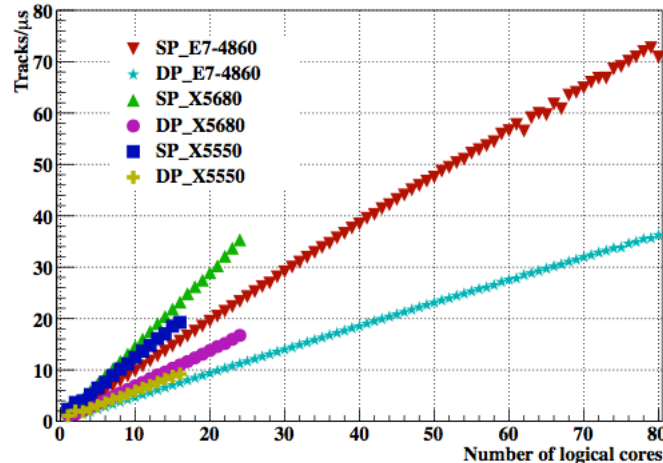
### Parallelization (many-cores):

- Open MP
- ITBB
- ArBB
- OpenCL

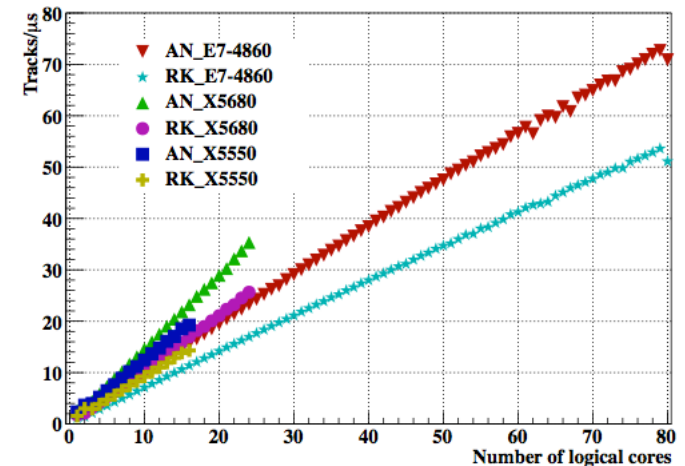
### Precision:

- single precision SP
- double precision DP

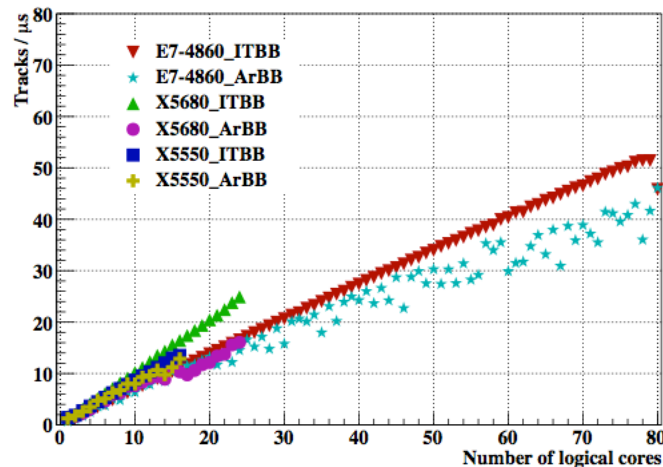
Conventional KF DP vs. SP



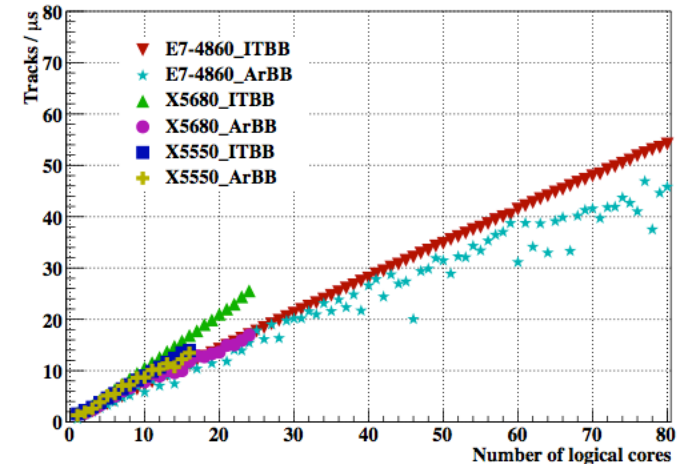
Conventional KF RK4 vs. Analytical



Square-Root KF

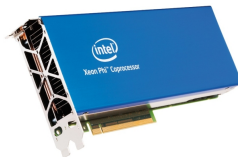
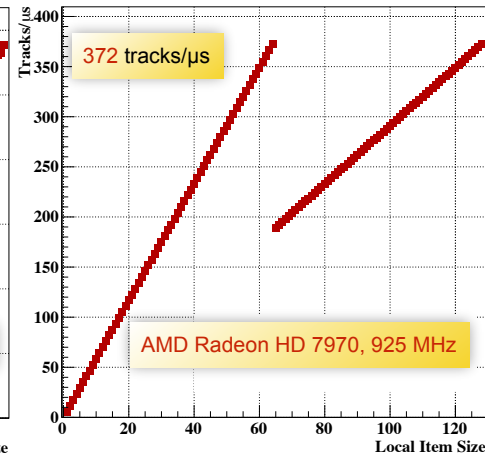
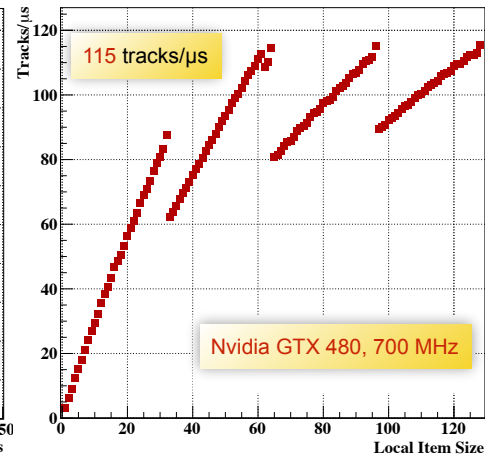
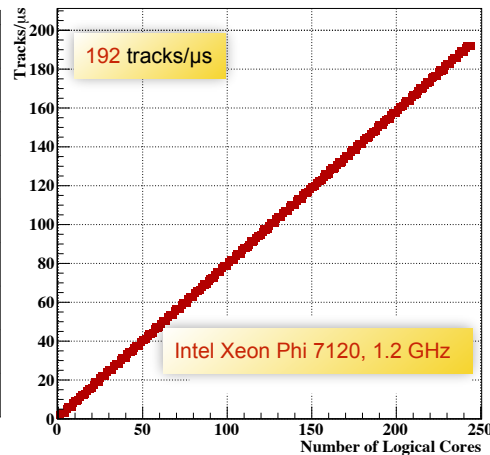
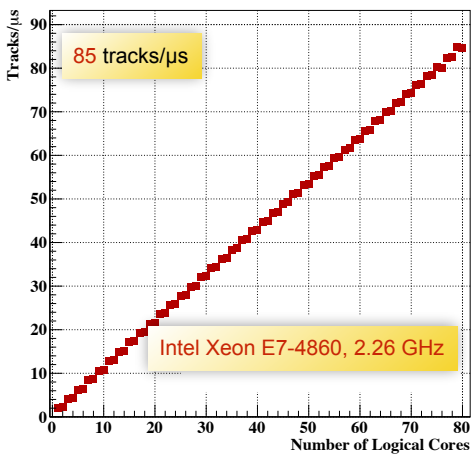


UD KF



Strong many-core scalability of the Kalman filter library

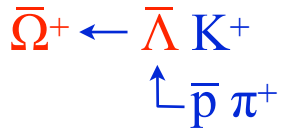
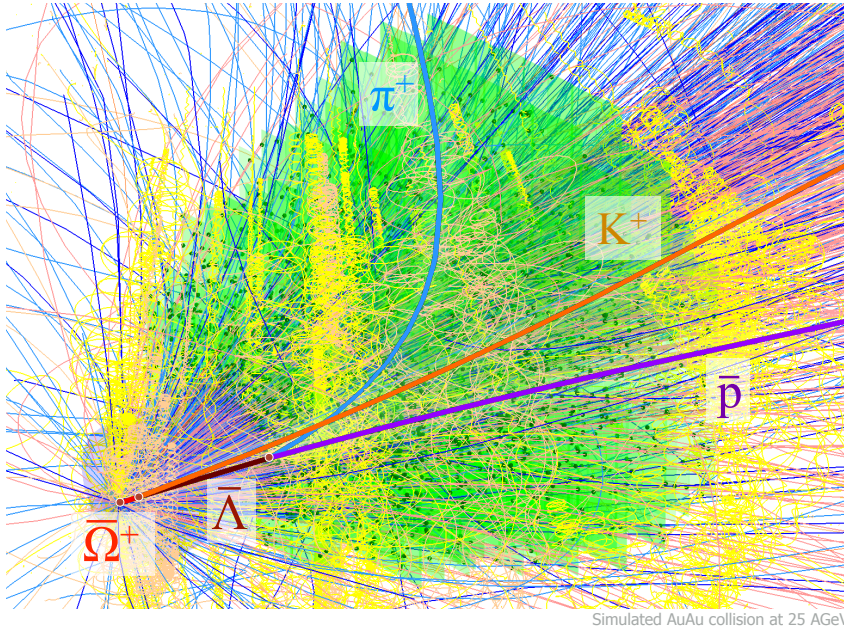
# Kalman Filter (KF) Track Fit



- Precise estimation of the parameters of particle trajectories is the core of the reconstruction procedure.
- **Scalability** with respect to the **number of logical cores** in a CPU is one of the most important parameters of the algorithm.
- The scalability on the **Intel Xeon Phi** coprocessor is **similar** to the **CPU**, but running **four threads per core** instead of two.
- In case of the **graphics cards** the set of tasks is divided into **working groups** of size **local item size** and **distributed among compute units** (or streaming multiprocessors) and the **load of each compute unit** is of the particular **importance**.
- The track fit performance on a single node:  **$2 * \text{CPU} + 2 * \text{GPU} = 10^9 \text{ tracks/s}$**  = (100 tracks/event) \*  $10^7 \text{ events/s}$  =  **$10^7 \text{ events/s}$** .
- **A single compute node is enough to estimate parameters of all particles produced at the maximum  $10^7$  interaction rate!**

The fastest implementation of the Kalman filter in the world

# KF Particle: Reconstruction of short-lived Particles



```
KFParticle Lambda(P, Pi);           // construct anti Lambda
Lambda.SetMassConstraint(1.1157);    // improve momentum and mass
KFParticle Omega(K, Lambda);        // construct anti Omega
PV -= (P; Pi; K);                   // clean the primary vertex
PV += Omega;                         // add Omega to the primary vertex
Omega.SetProductionVertex(PV);       // Omega is fully fitted
(K; Lambda).SetProductionVertex(Omega); // K, Lambda are fully fitted
(P; Pi).SetProductionVertex(Lambda); // p, pi are fully fitted
```

$$\mathbf{r} = \{ x, y, z, p_x, p_y, p_z, E \}$$

**State vector**

$$\mathbf{C} = \langle \mathbf{r} \mathbf{r}^T \rangle =$$

**Covariance matrix**

$$\begin{bmatrix} \sigma_x^2 & C_{xy} & C_{xz} & C_{xp_x} & C_{xp_y} & C_{xp_z} & C_{xE} \\ C_{xy} & \sigma_y^2 & C_{yz} & C_{yp_x} & C_{yp_y} & C_{yp_z} & C_{yE} \\ C_{xz} & C_{yz} & \sigma_z^2 & C_{zp_x} & C_{zp_y} & C_{zp_z} & C_{zE} \\ C_{xp_x} & C_{yp_x} & C_{zp_x} & \sigma_{p_x}^2 & C_{p_x p_y} & C_{p_x p_z} & C_{p_x E} \\ C_{xp_y} & C_{yp_y} & C_{zp_y} & C_{p_x p_y} & \sigma_{p_y}^2 & C_{p_y p_z} & C_{p_y E} \\ C_{xp_z} & C_{yp_z} & C_{zp_z} & C_{p_x p_z} & C_{p_y p_z} & \sigma_{p_z}^2 & C_{p_z E} \\ C_{xE} & C_{yE} & C_{zE} & C_{p_x E} & C_{p_y E} & C_{p_z E} & \sigma_E^2 \end{bmatrix}$$

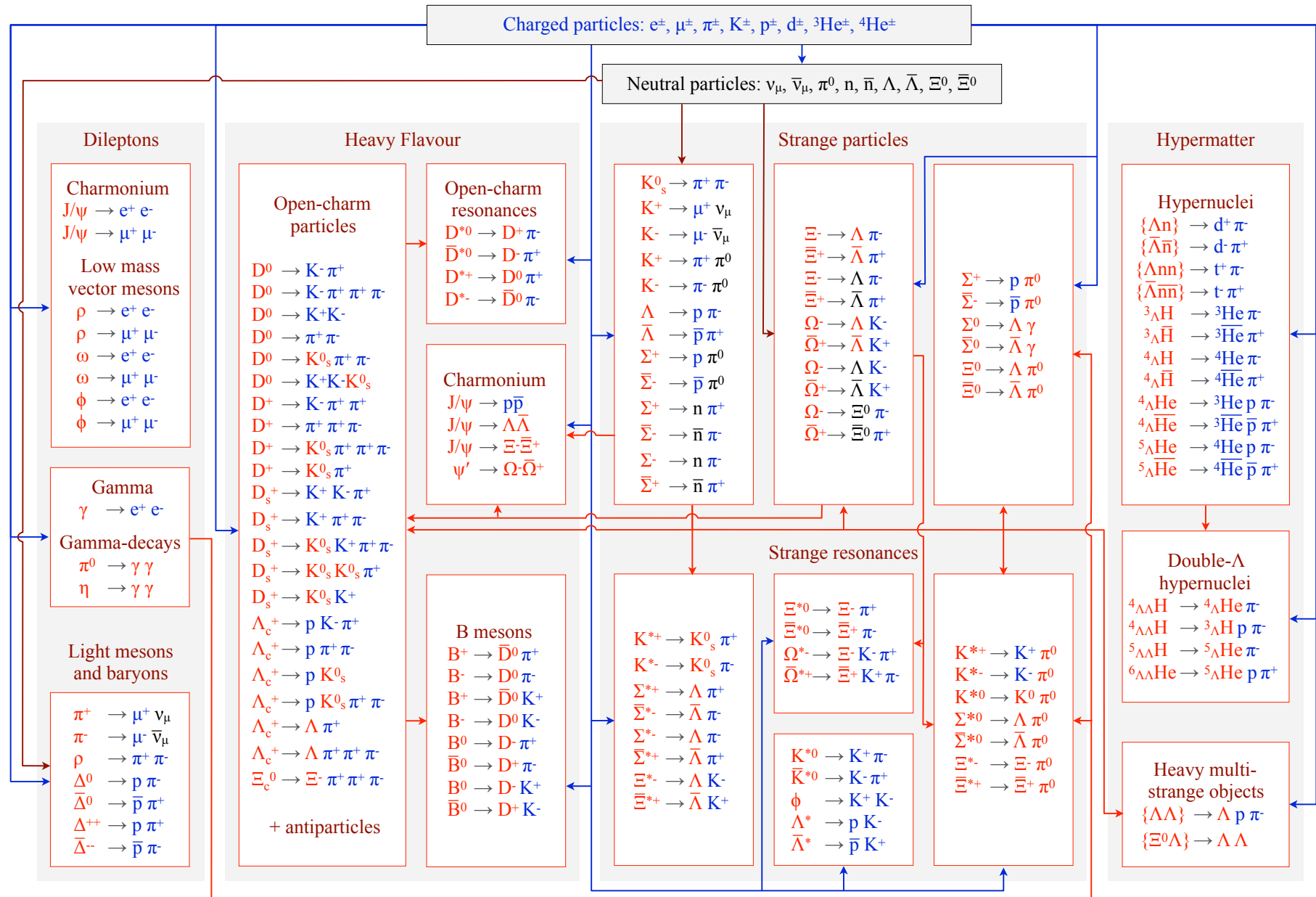
## Features:

- KF Particle class describes particles by the **state vector** and the **covariance matrix**.
- Covariance matrix contains essential information about tracking and **detector** performance.
- The method for **mathematically correct** usage of covariance matrices is provided by the KF Particle package based on the **Kalman filter** (KF).
- Heavy mathematics of KF requires **fast** and **vectorised** algorithms.
- **Mother** and **daughter** particles are treated in the same way.
- The **natural** and **simple interface** allows two reconstruct easily complicated decay chains.
- The package is geometrically independent and can be adapted to **different experiments** (CBM, ALICE, STAR).

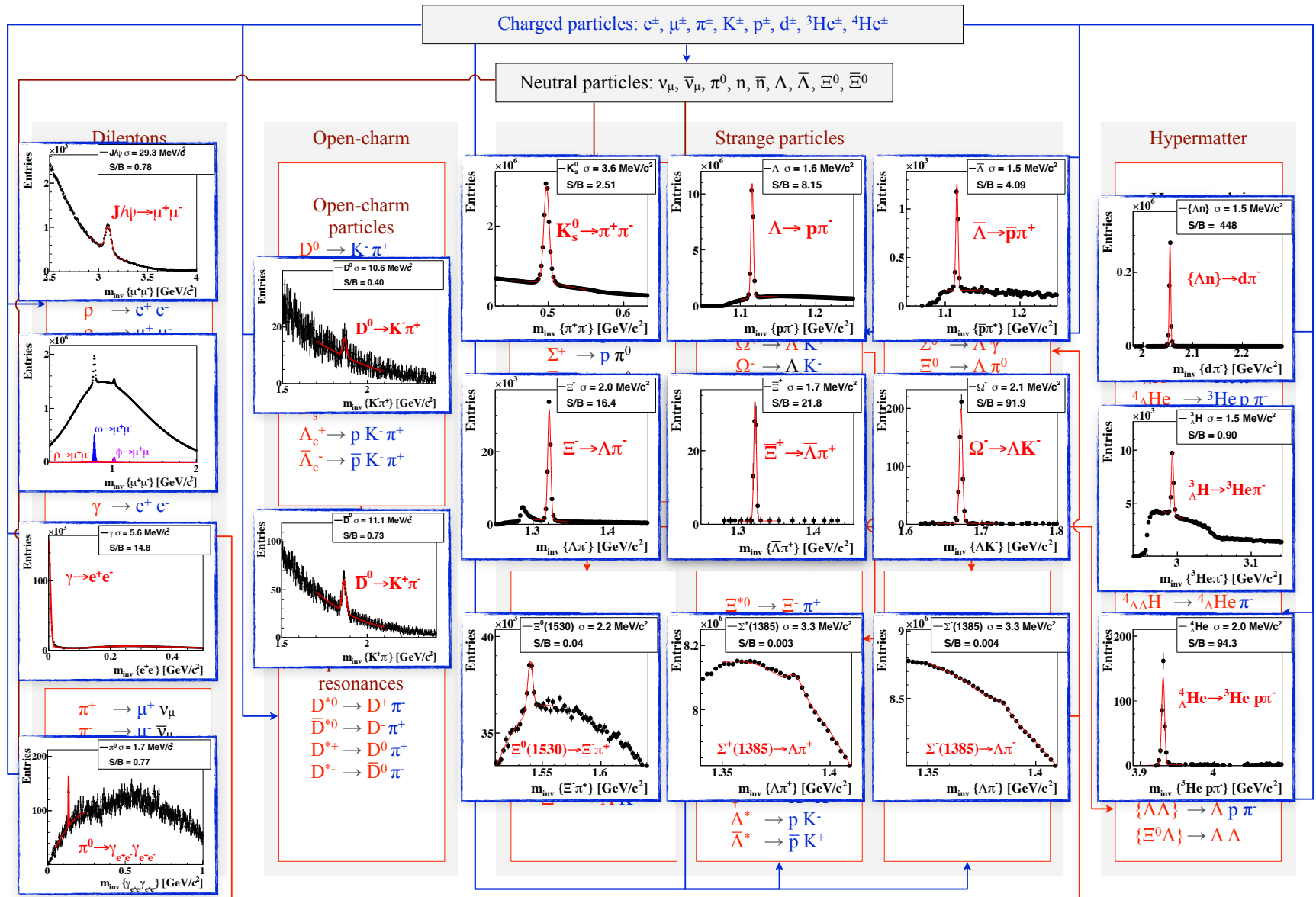
KF Particle provides a simple and very efficient approach to physics analysis



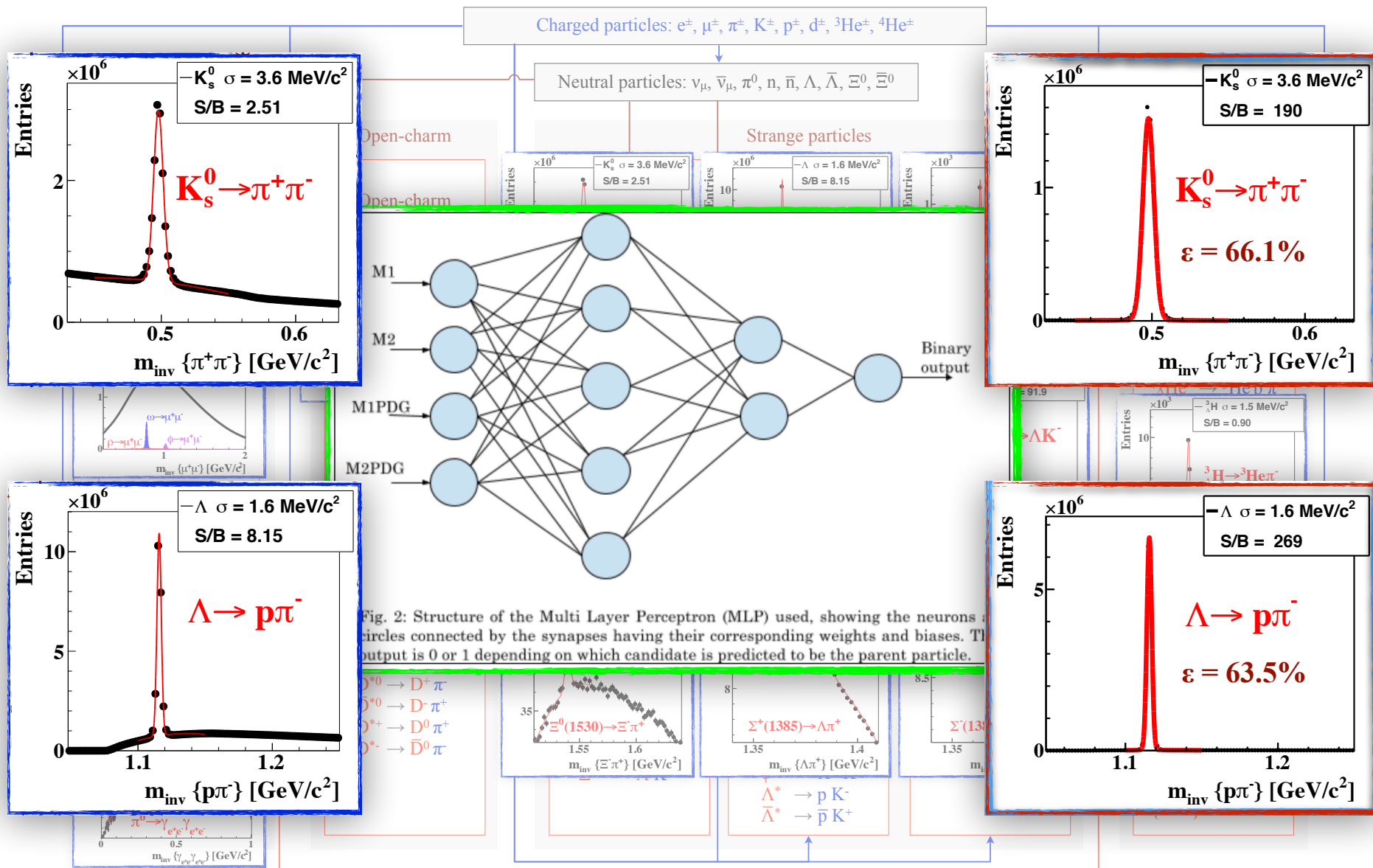
# KF Particle Finder for short-lived particles



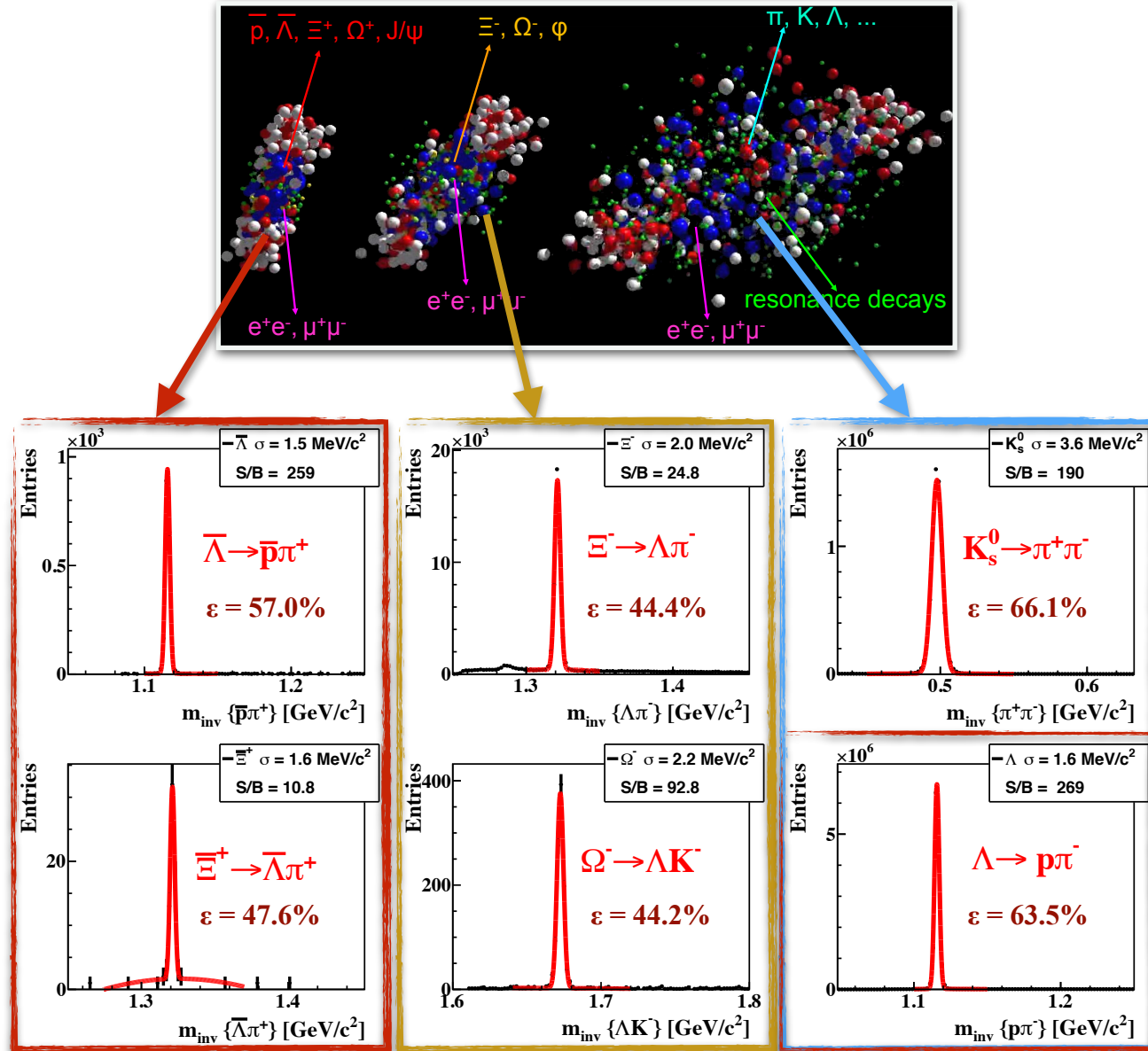
# KF Particle Finder for short-lived particles



# Artificial Neural Network



# Very Clean Probes of Collision Stages



AuAu, 10 AGeV, 3.5M central UrQMD events, MC PID



# Parallelization Challenge in the CBM Event Reconstruction

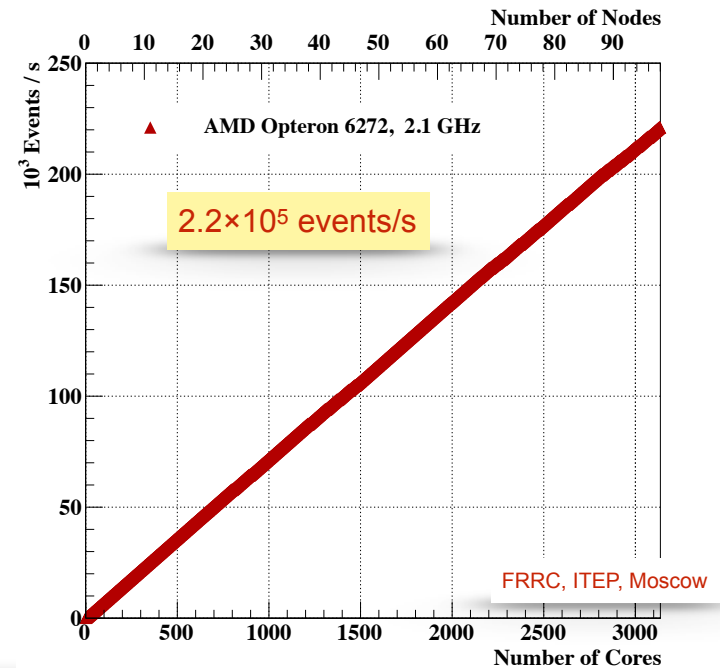
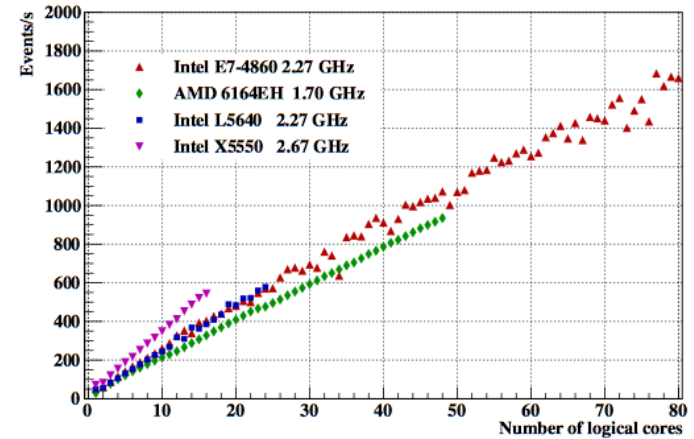
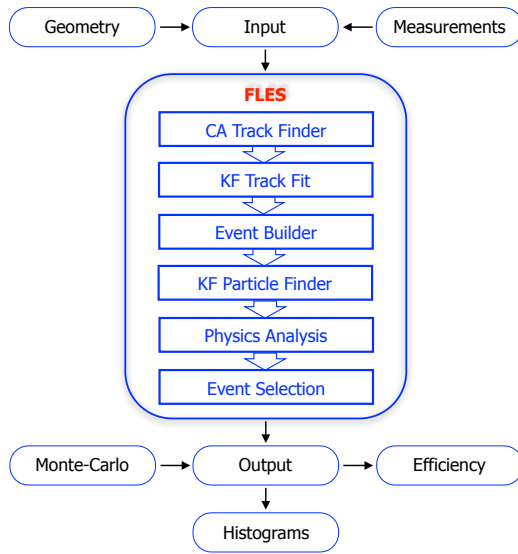
Location		Architecture	(Nodes·)sockets·cores·threads·SIMD	Data streams
CERN	Switzerland	AMD 6164HE	4·12·1·4	192
GSI	Germany	Intel E7-4860	4·10·2·4	320
ITEP	Russia	AMD 6272	100·(2·16·1·4)	12 800
FIAS	Germany	Intel E5-2600+Intel Phi 7120	2·8·2·8+2·61·4·16	256+7 808
BNL	USA	Intel E5-2680+Intel Phi 5110P	22·(2·12·2·8+2·60·4·16)	8 448+168 960

List of some heterogeneous HPC nodes, used in our investigations

Andrzej Nowak (OpenLab, CERN) by Hans von der Schmitt (ATLAS) at GPU Workshop, DESY, 15-16 April 2013							
	SIMD	Instr. Level Parallelism	HW Threads	Cores	Sockets	Factor	Efficiency
MAX	4	4	1.35	8	4	691.2	100.0%
Typical	2.5	1.43	1.25	8	2	71.5	10.3%
HEP	1	0.80	1	6	2	9.6	1.4%
CBM@FAIR	4	3	1.3	8	4	499.2	72.2%

Parallelization becomes a standard in the CBM experiment

# Running FLES on HPC Node/Farm



The FLES package is vectorized, parallelized, portable and scalable up to 3 200 CPU cores

# Conclusion

---

The future is parallel.  
The future is parallel.  
The future is parallel.  
The future is parallel.  
The future is parallel.