GPGPU
for track finding
in High Energy Physics

L. Rinaldi, M. Belgiovine, R. Di Sipio, A. Gabrielli, M. Villa
(Bologna University and INFN)

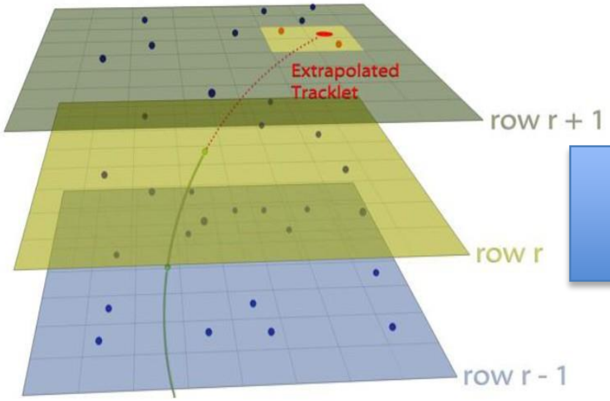
M. Negrini, F. Semeria, A. Sidoti
(INFN Bologna)

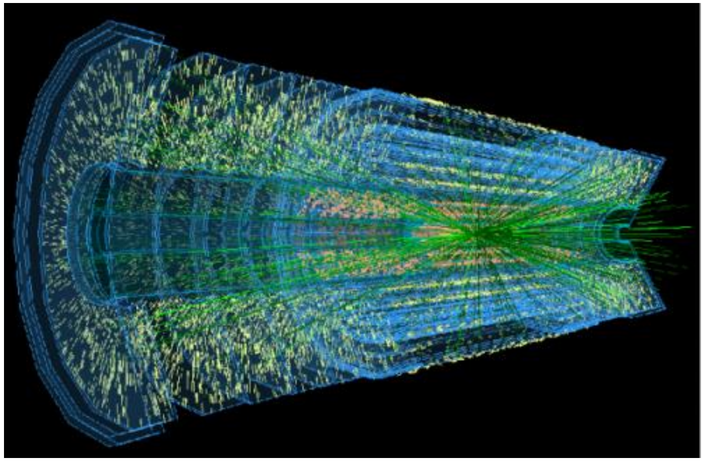
S. A. Tupputi
(INFN CNAF)

GPU Computing in High Energy Physics
Pisa, 10-12/9/2014

Outlook

A massive parallel approach based on GPGPU can be relevant for Tracking in High Energy Physics





Fast tracking is suitable for realtime data selection

In this contribution we will show a track finding algorithm based on the Hough Transform

Tracking in HEP experiments

Model based on a typical central track detector:

- Multi-layer cylindrical shape, with axis on the beamline and centered on the nominal interaction point
- Uniform magnetic field with field lines parallel to the beamline
- Charged particles will have helix trajectories (circles in the transverse plane wrt z-axis)

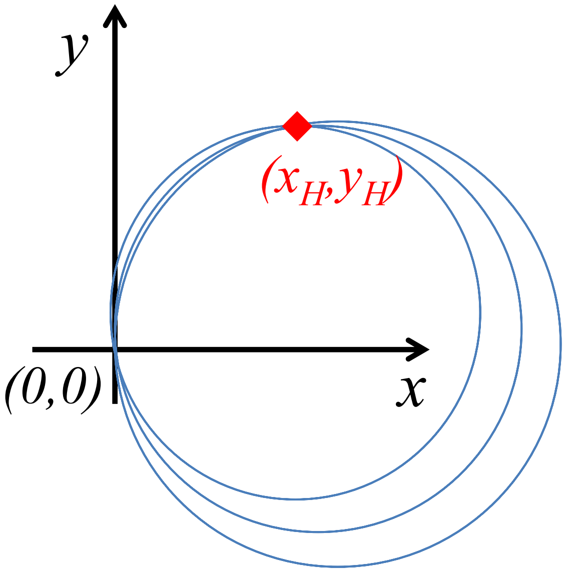
Several approaches used to extract track parameters from experimental data (fitting, associative memories, etc.)

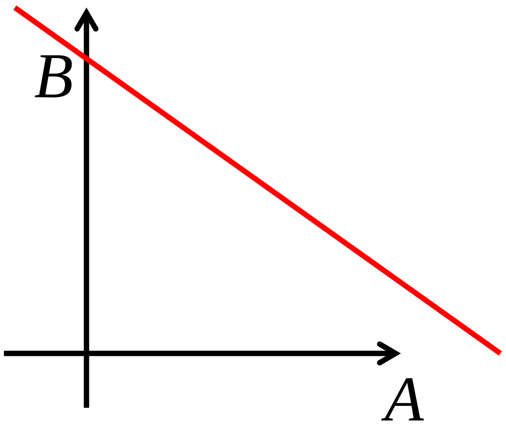
Hough Transform (HT) is yet another method

HT is a pattern recognition technique (60's) for *feature* extraction in *image* processing

The advantage: very massive parallelisation could be applied

The Hough Transform





In real space there are ∞ circles passing for each hit (x_H, y_H) and $(0,0)$:

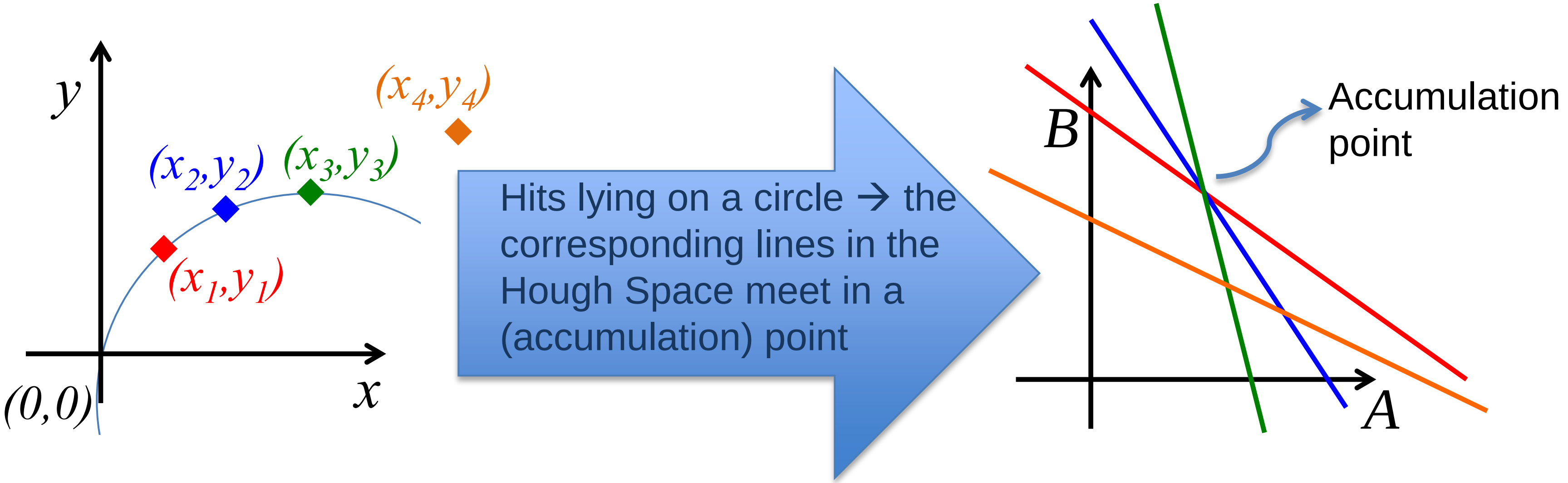
$$x_H^2 + y_H^2 - 2Ax_H - 2By_H = 0$$

The point of coordinates (A,B) is the center of the circle

In the A-B parameter space **(Hough space)**, for each hit, all the ∞ circles are represented with straight lines:

$$B = \frac{x_H^2 + y_H^2 - 2Ax_H}{2y_H}$$

The Hough Transform



Hits lying on a circle → the corresponding lines in the Hough Space meet in a (accumulation) point

First step: discretize the Hough space with a $N_A \times N_B$ Hough Matrix (or Vote Matrix)

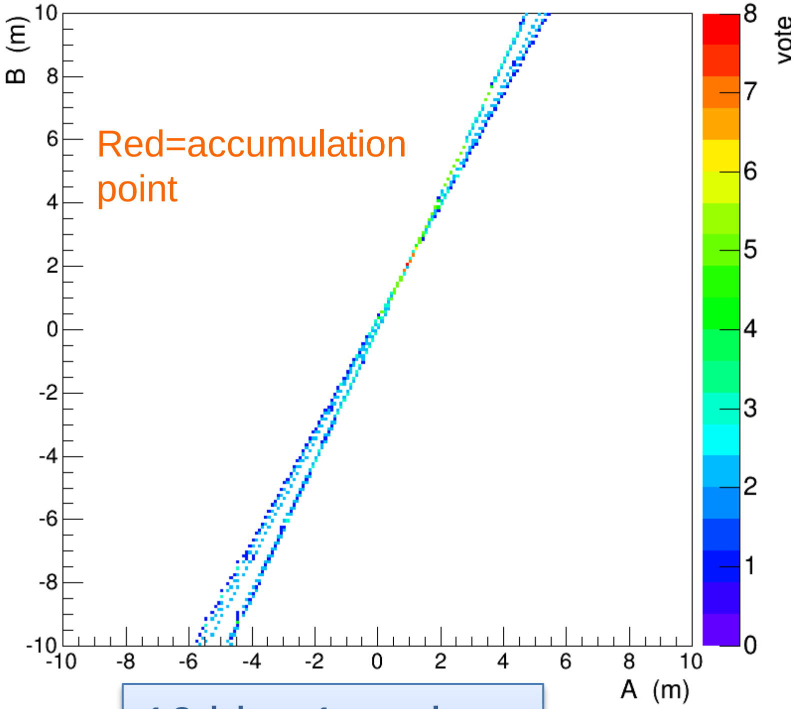
For each hit, all the matrix elements satisfying $B = \frac{x_H^2 + y_H^2 - 2Ax_H}{2y_H}$

are incremented by one unity (or weighted value).

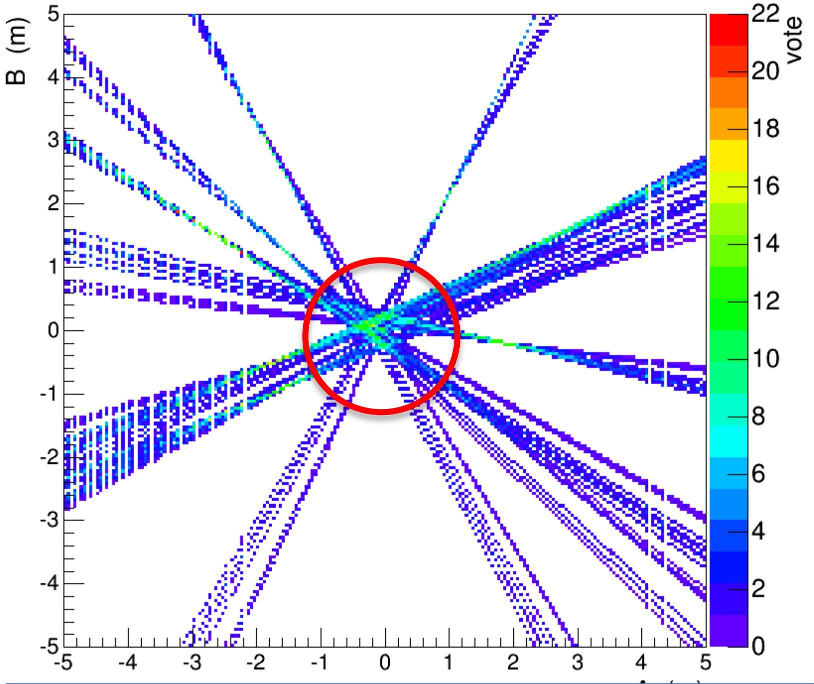
Accumulation points with high vote will correspond to real tracks

The Hough Transform

Second step: find local maxima of the Hough Matrix (each maximum corresponding to a real track)



Red=accumulation point



10 hits, 1 track
Very simple case

100 hits, 8 tracks, a little bit complex
(some cuts needed)

Test description

- Stand-alone testbed, not (yet) interfaced to any experiment framework
- Model based on a cylindrical 12-layer Si detector
  - 100 simulated events (pp collisions @ LHC energy, Minimum Bias sample with low p_T tracks)
  - Each event contains up to 5000 hits and O(100) tracks
  - Known quantities: *x*,*y*,*z* coord's of the hits
  - Hough-space divided in 4 iper-dimensions:

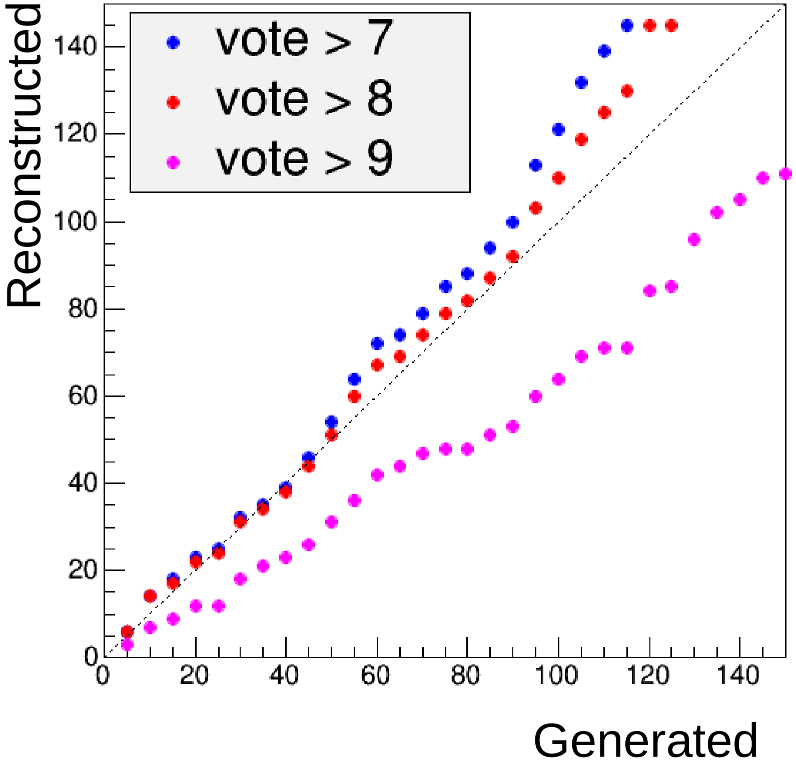
    the *A* and *B* parameters and the transverse (ϕ) and longitudinal (θ) planes
  - 4x16x1024x1024 $M_H(\phi,\theta,A,B)$ Hough-matrix

Computing resources

	Local resources	INFN-CNAF HPC-Cluster	
Device specification	**NVIDIA GeForce GTX770**	**NVIDIA Tesla K20m (2x)**	**NVIDIA Tesla K40m (2x)**
Performance (Gflops)	3213	3524	4291
Mem. Bandwidth (GB/s)	224.3	208	288
Connection	PCIe3	PCIe3	PCIe3
Mem. Size (MB)	2048	5120	12228
Number of Cores	1536	2496	2880
Clock Speed (MHz)	1046	706	745

HT algorithm performance

Number of tracks



Number of reconstructed tracks strongly dependent on algorithm parameters:

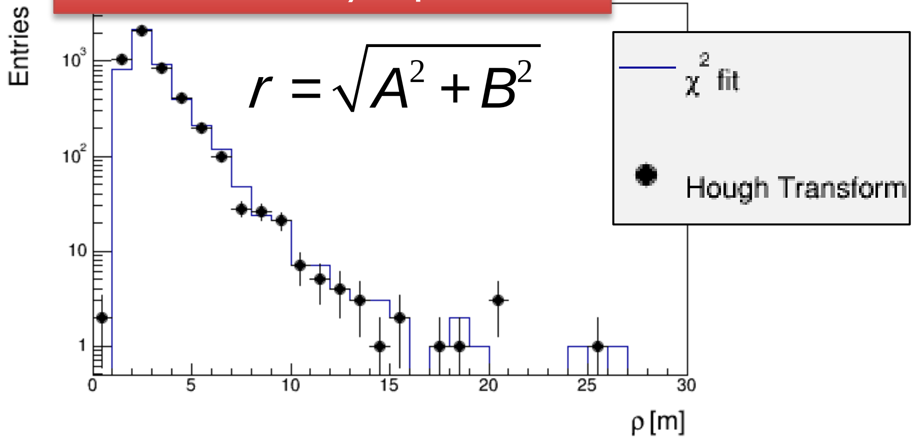
- Hough Matrix Dimension
- Vote threshold

Reconstruction slightly overestimated:

A solution could be to add more constraints from other event features

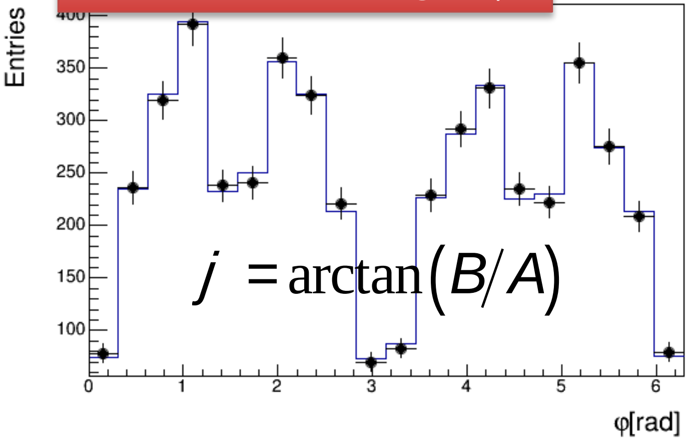
HT vs χ^2 fit

Track radius ρ spectrum



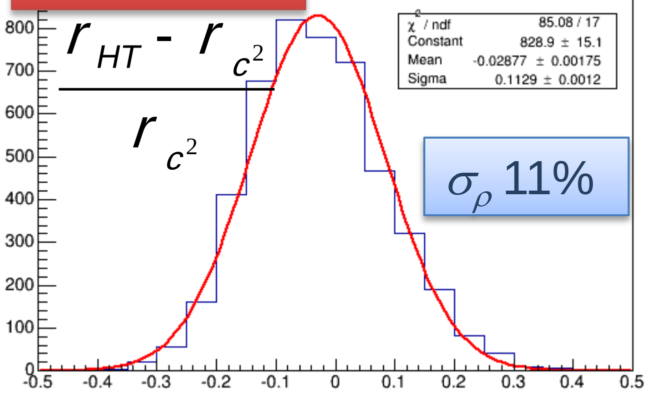
$$r = \sqrt{A^2 + B^2}$$

Track center angle φ



$$j = \arctan(B/A)$$

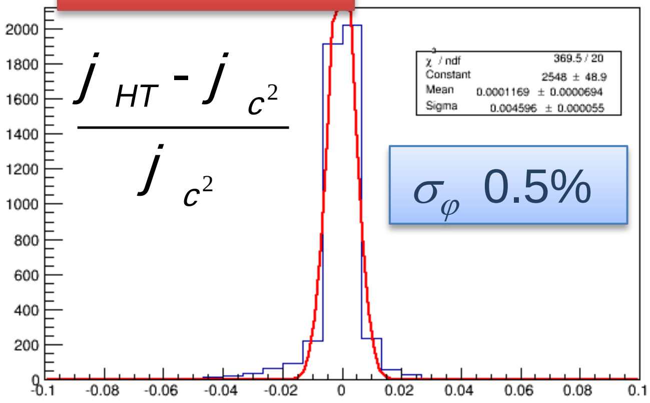
ρ resolution



$$\frac{r_{HT} - r_{c^2}}{r_{c^2}}$$

σ_ρ 11%

φ resolution

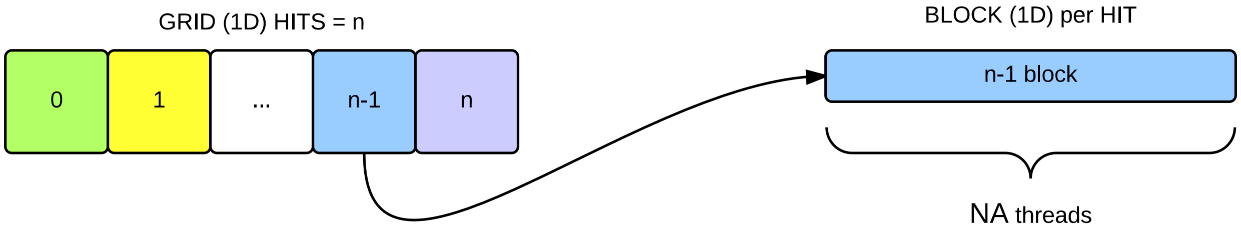


$$\frac{j_{HT} - j_{c^2}}{j_{c^2}}$$

σ_φ 0.5%

CUDA 6.0 coding

Hough Matrix filling (Vote):



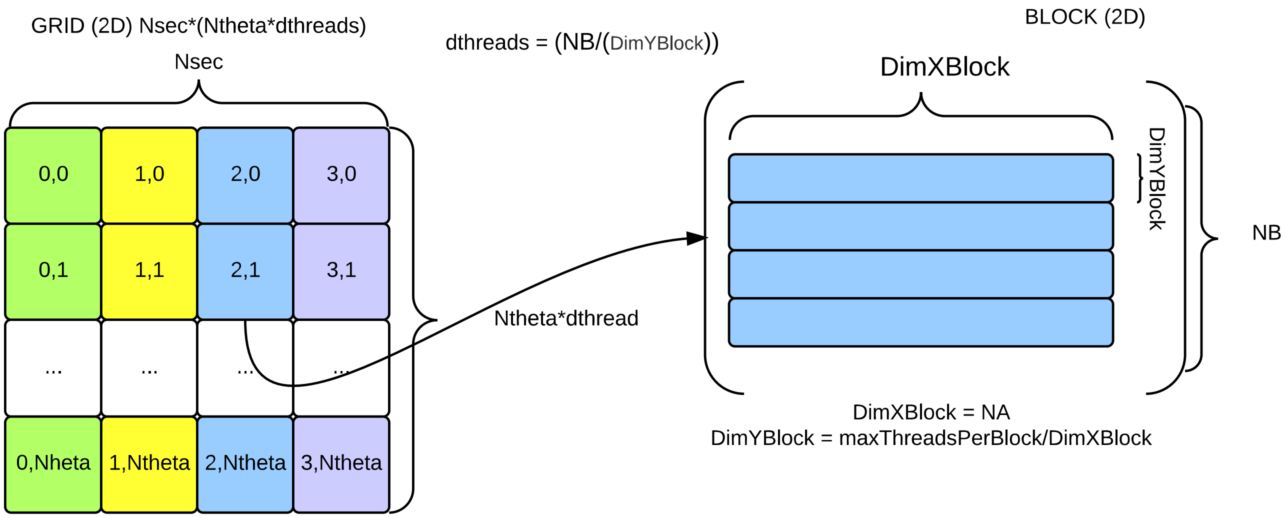
- 1D grid over hits

  (grid dimension = number of hits)
- At a given (ϕ, θ), threadblock over A .

  For each A, a corresponding B is evaluated
- The $M_H(\phi, \theta, A, B)$ Hough-Matrix element is incremented by a unity with CUDA atomicAdd()
- Matrix initialization once at first iteration with cudaMallocHost (pinned memory) and initialized on device with cudaMemset

CUDA 6.0 coding

Local Maxima search



GRID (2D) Nsec*(Ntheta*dthreads)

Nsec

0,0	1,0	2,0	3,0
0,1	1,1	2,1	3,1
...	...	...	...
0,Nheta	1,Ntheta	2,Ntheta	3,Ntheta

Ntheta*dthread

dthreads = (NB/(DimYBlock))

BLOCK (2D)

DimXBlock

DimYBlock

NB

DimXBlock = NA

DimYBlock = maxThreadsPerBlock/DimXBlock

- 2D-grid over (ϕ, θ)

Grid dimension: N_ϕ×(N_θ*N_A*N_B/maxThreadsPerBlock)

- 2D-threadblock, with dimXBlock= N_A, dimYBlock=maxThreadsPerBlock/N_A
- Each thread compares the $M_H(\phi, \theta, A, B)$ element to neighbours, the bigger is stored in the GPU shared memory and eventually transferred back.
- I/O demanding – several kernel may access matrix together

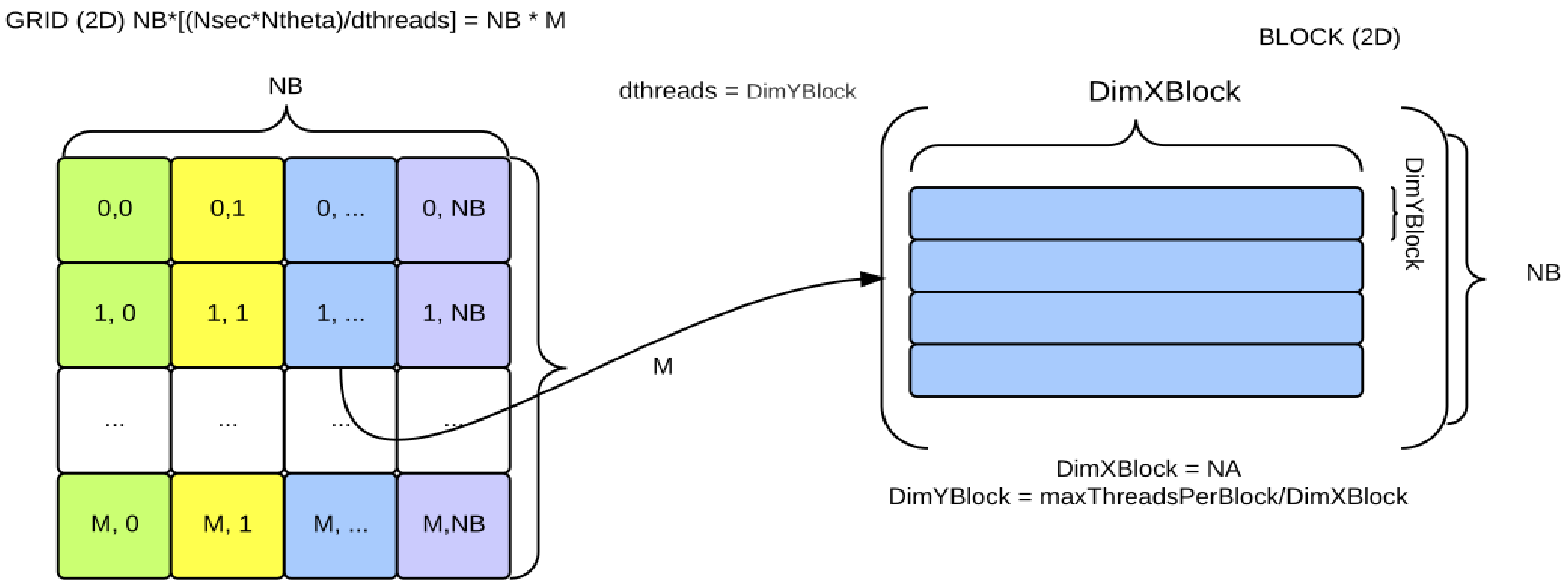
OpenCL 1.1 coding

- Translation from CUDA to OpenCL had to be done carefully:

  No direct pinning memory API for vote and relative maxima matrices:

  - The OpenCL workaround: mapping a device buffer to an already memalloc'ed host buffer
  - Ad hoc kernels used for initializing the matrices in the device memory
    - Such kernels' execution times go into initialization time
- Memory buffers H2D allocation performed concurrently and asynchronously in OpenCL, saving overall transferring time
- Respect to CUDA, working principle of the kernels is unchanged, except for block/thread settings

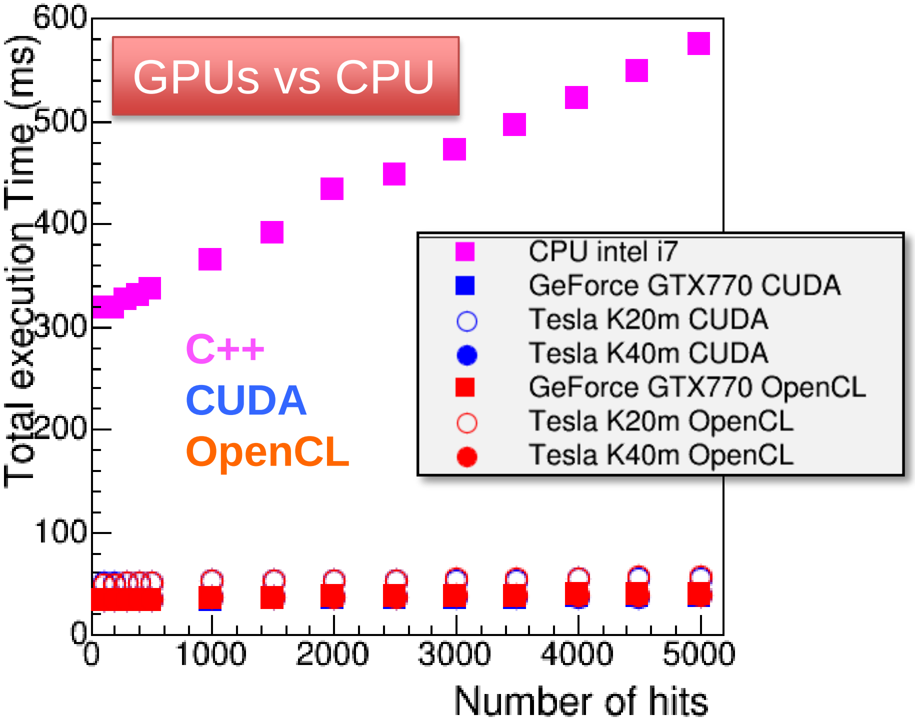
OpenCL 1.1 coding

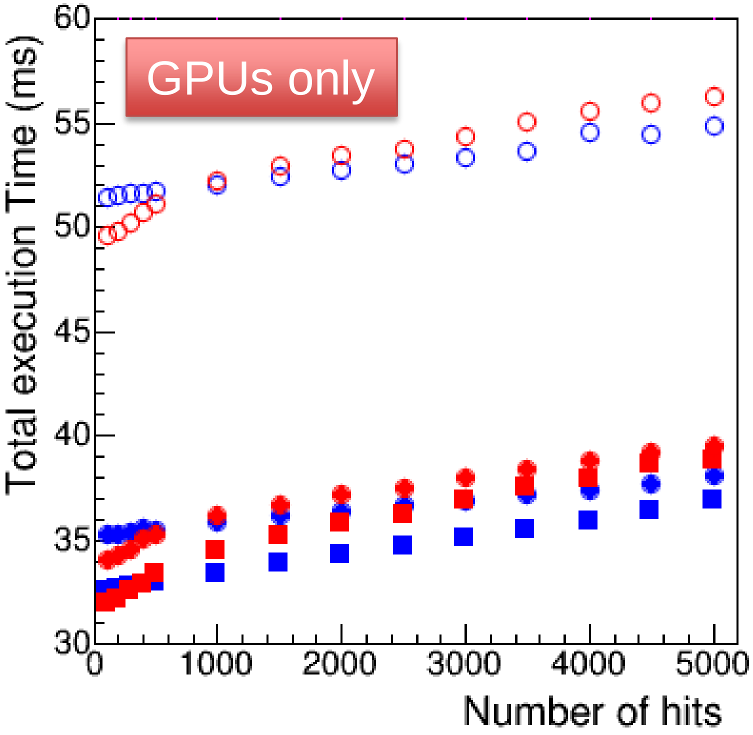

GRID (2D) NB*[(Nsec*Ntheta)/dthreads] = NB * M
NB
dthreads = DimYBlock
M
BLOCK (2D)
DimXBlock
DimYBlock
NB
DimXBlock = NA
DimYBlock = maxThreadsPerBlock/DimXBlock

0,0	0,1	0, ...	0, NB
1, 0	1, 1	1, ...	1, NB
...	...	...	...
M, 0	M, 1	M, ...	M,NB

- The block/thread counting OpenCL APIs made such arrangement more useful and easy-to-manage
  - Local and global thread (work-items in OpenCL) numbers are considered instead of thread and blocks (work-group in OpenCL)
- Indexes have been managed so to have coalesced memory kernels I/O access thus speeding up overall execution
  - Useful both in OpenCL and CUDA versions

Total execution time vs CPU





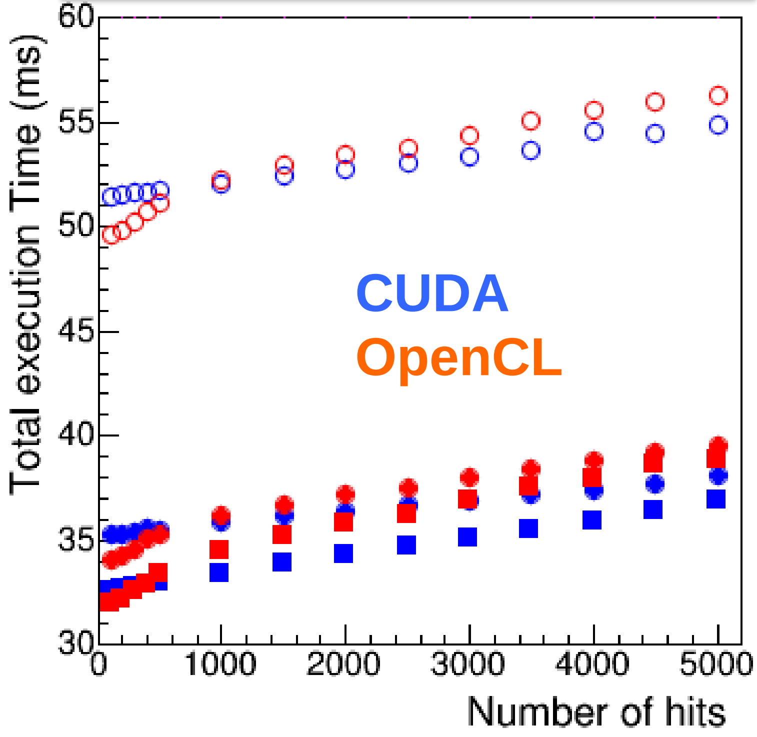
GPU%CPU speed up over 15x

CPU timing scales with number of hits

GPU timing almost independent on number of hits

CUDA vs OpenCL

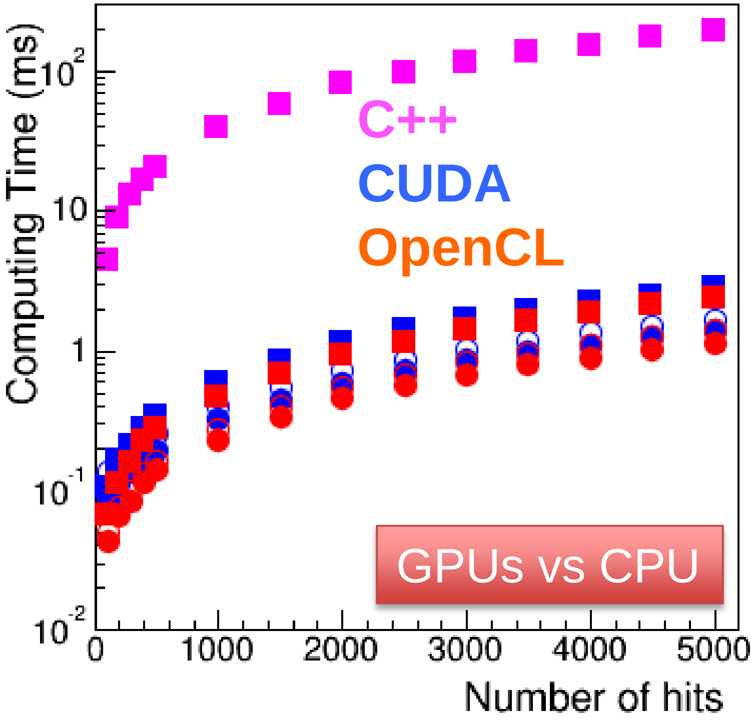
Total execution time

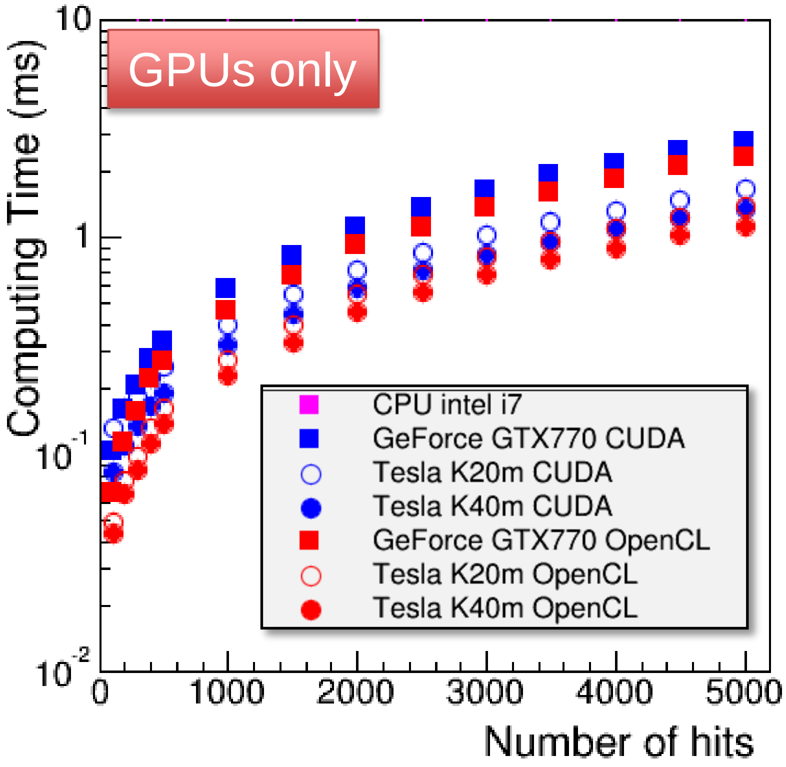


Best performance of our code on GTX770, CUDA-coded

For large numbers hits, CUDA performs better on all devices

Kernel: Hough Matrix Filling





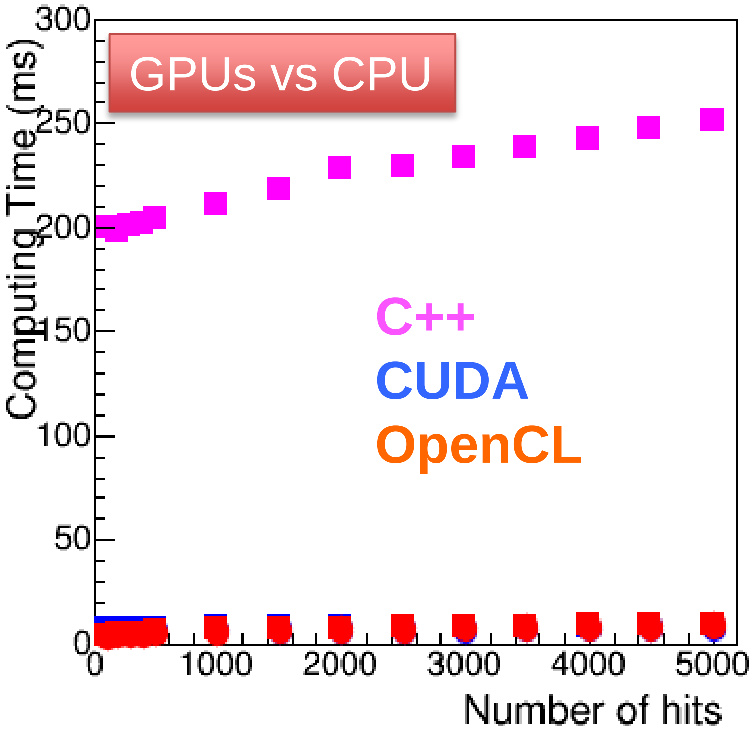
Up to GPU%CPU 200x speedup

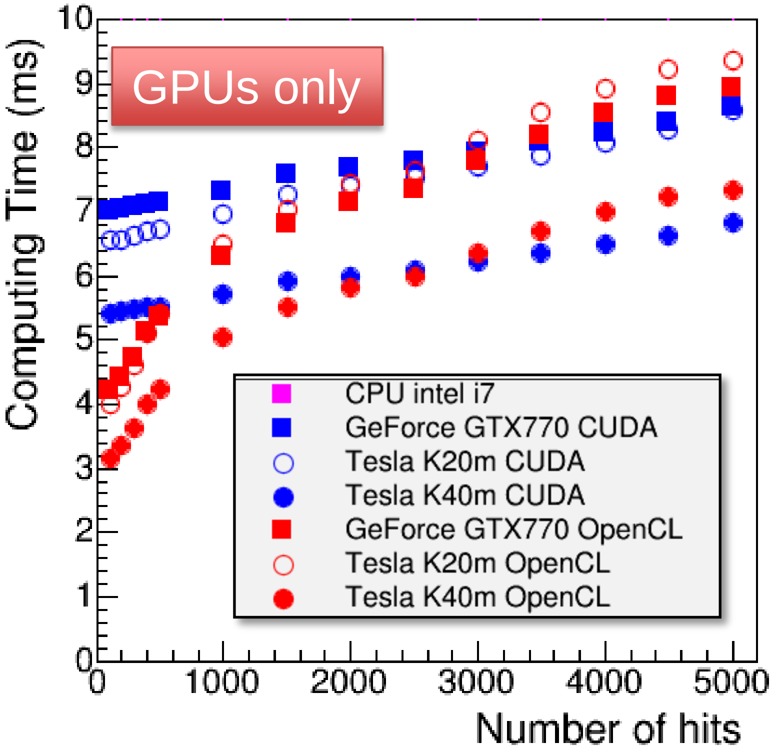
Linear dependence on number of hits

Good performace of Tesla's

OpenCL code better optimized on nested loop

Kernel: Relative Maxima





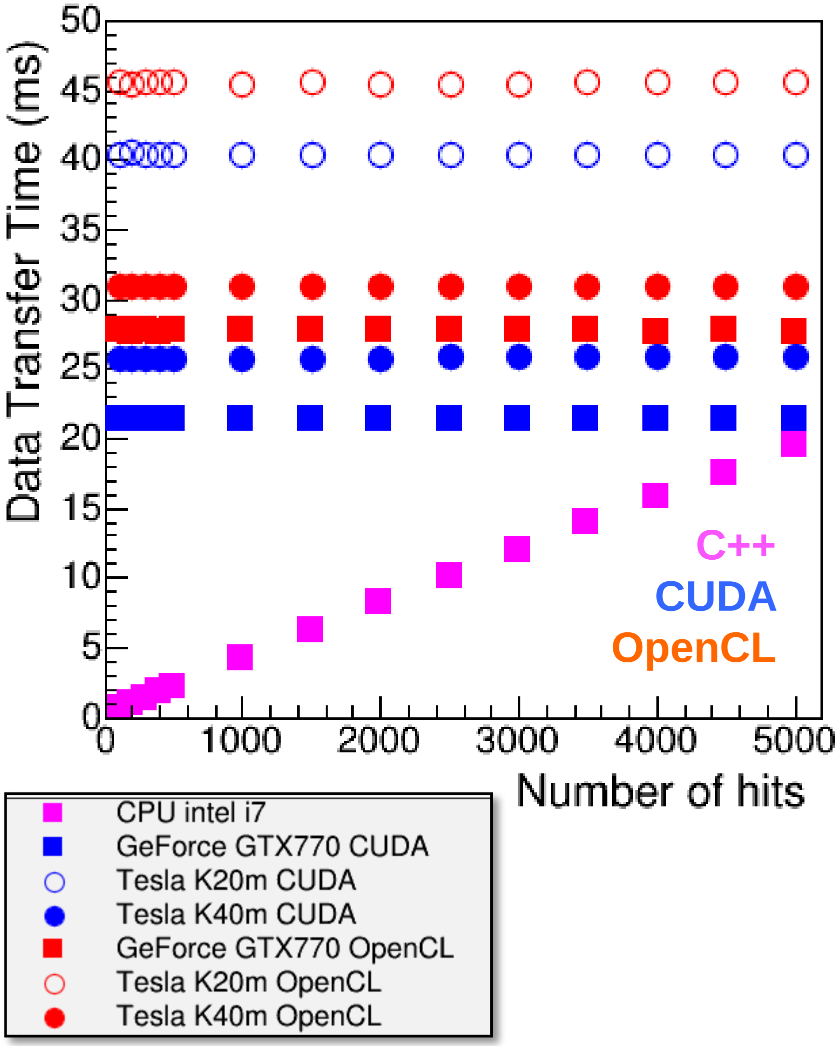
Up to GPU%CPU 60x speedup

Linear dependence on number of hits

Good perfomance on Tesla k40m

CUDA and OpenCL comparable when processing large numbers of hits

Host ← →Device throughput



The bottleneck

CPU I/O much faster than GPUs

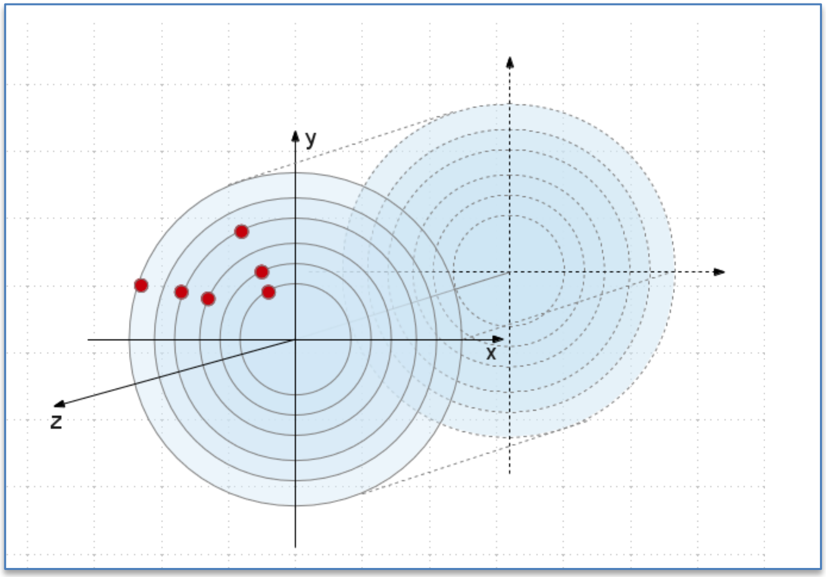
Our code transfers data faster on the GeForce GTX770

CUDA access to memory faster than OpenCL

Multi-GPU configuration

Physical motivation:

- split the transverse plane in sectors (at detector-readout level).
- Each sector processed separately (data independent across sectors)



A single Hough Transform executed for each sector (assigned to a single GPU)

Results merged when each GPU finishes its own process

Benefit:

- HT execution per sector overlapped
- Lightweight Hough Matrices and output structures per sector

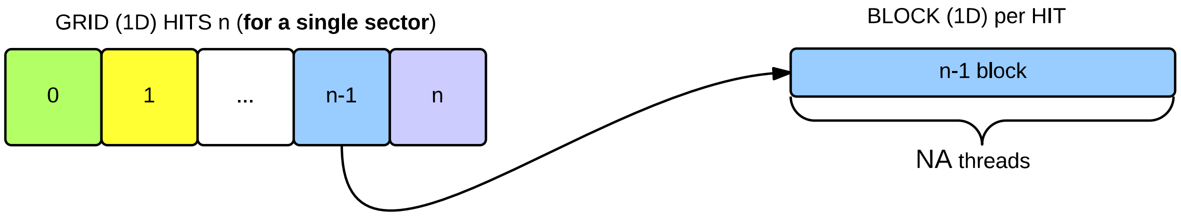
Multi-GPU configuration

Similar workload schema with 4 Hough Matrixes $M_H(\theta,A,B)$

4x(16x1024x1024)

CUDA implementtion only

voteHoughSpace() - single sector



GRID (1D) HITS n (**for a single sector**)

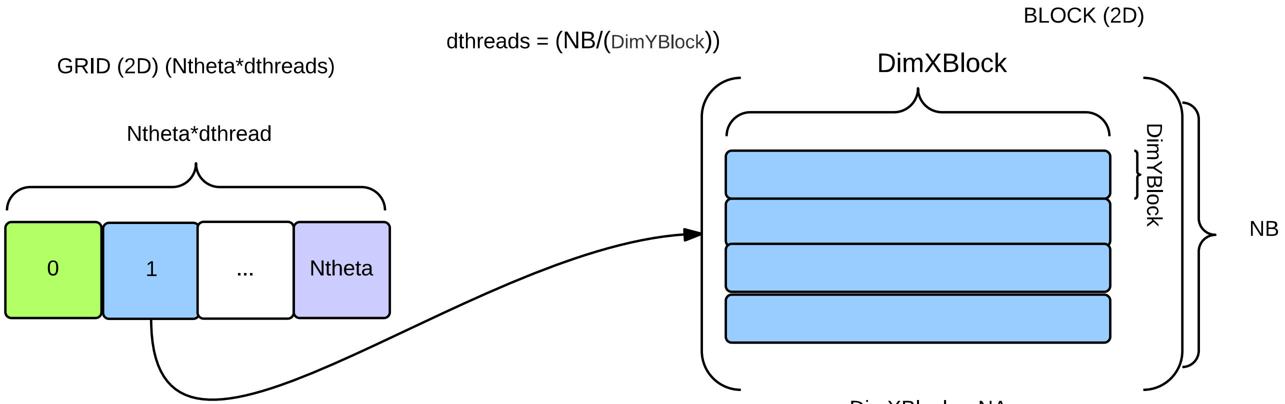
0 | 1 | ... | n-1 | n

BLOCK (1D) per HIT

n-1 block

NA threads

findRelativeMax() - single sector



GRID (2D) (Ntheta*dthreads)

Ntheta*dthread

0 | 1 | ... | Ntheta

dthreads = (NB/(DimYBlock))

BLOCK (2D)

DimXBlock

DimYBlock

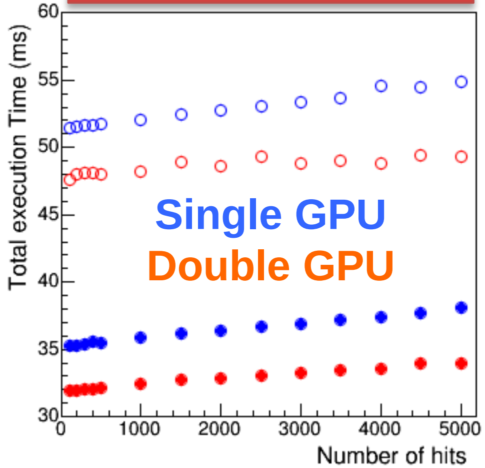
NB

DimXBlock = NA

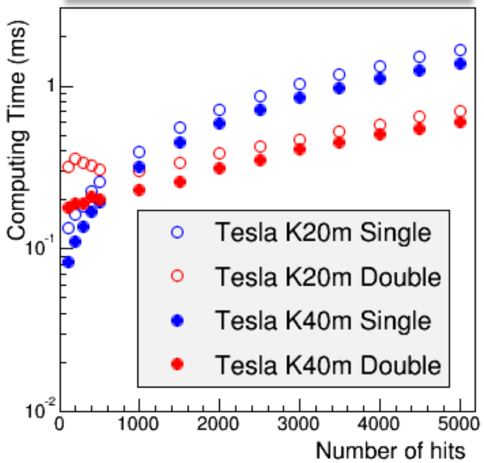
DimYBlock = maxThreadsPerBlock/DimXBlock

Multi-GPU results

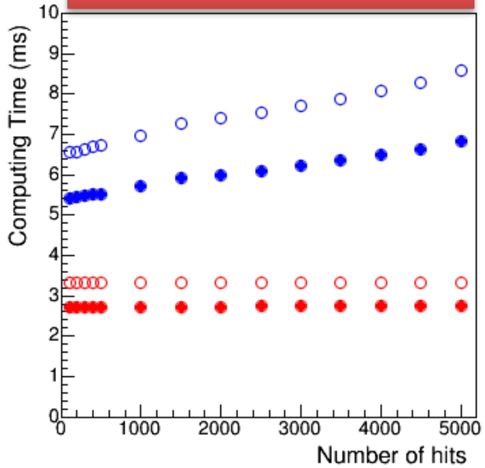
Total execution time



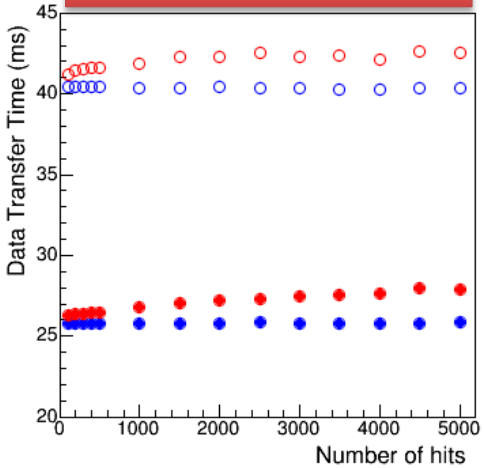
Hough Matrix filling



Relative Max Search



GPU←→CPU throughput



Test performed separtely with **2** Tesla k20m and **2** k40m (using CUDA)

Multi-GPU faster than single-GPU but not elevate gain

2x Speedup observed on kernels execution (M_H filling for large number of hits)

CUDA unified memory NOT used (specified instruction for memory access needed on multiple devices)

Summary and lesson learned

- Use of GPGPU can dramatically reduce track reconstruction time
  - A pattern recognition algorithm based on the Hough Transform has successfully implemented on CUDA and OpenCL, also using multiple devices

- Good performance obtained on pure computational algorithms
- GPU←→CPU transfers still conditioning total execution time
- Many handles for optimizing performance → Dependent on the GPU board specifications

Next steps

- Hough Transform method studied on a stand-alone simulation
  - Interface to a HEP experiment framework (both software and hardware)
  - Test with other accelerators/coprocessors
  - Introduce parallel reduction algorithm into RelMax kernel

backup

CUDA && OpenCL

- CUDA code ported to OpenCL1.1 (included in CUDA drivers)
- Each language has its own advantages, not necessarily unique:
  - CUDA *and* OpenCL rather than CUDA *or* OpenCL

- Advantages (non-exhaustive list) of OCL can be drawn from
  - Kernel just-in-time (JIT) compilation, allowing to optimize the kernel execution for the actual employed device
  - Easy-to-set asynchronous behavior, thus allowing to fully exploit resources
    - Asynchrony can go all the way with GPUs allowing to I/O while computing kernels (not in this talk)
  - Etc…
- Disadvantages
  - Harder learning curve than CUDA
  - JIT compilation increases debug time
  - Not all the HW features may be taken into account
  - Etc…