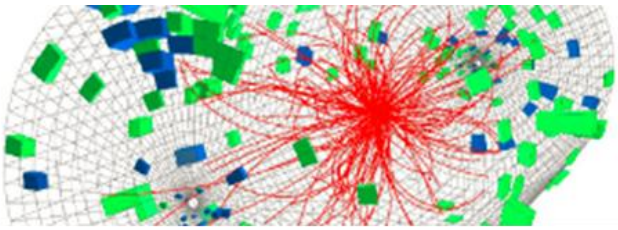
A GPU-based track reconstruction in the core of high p_T jets in CMSSW

S. Donato, B. Hegner, V. Innocente, A. Meyer, F. Pantaleo, A. Pfeiffer, A. Rizzi, A. Schmidt

felice@cern.ch

GPU in HIGH ENERGY PHYSICS PISA 10-12 SEP 2014



Bundesministerium für Bildung und Forschung

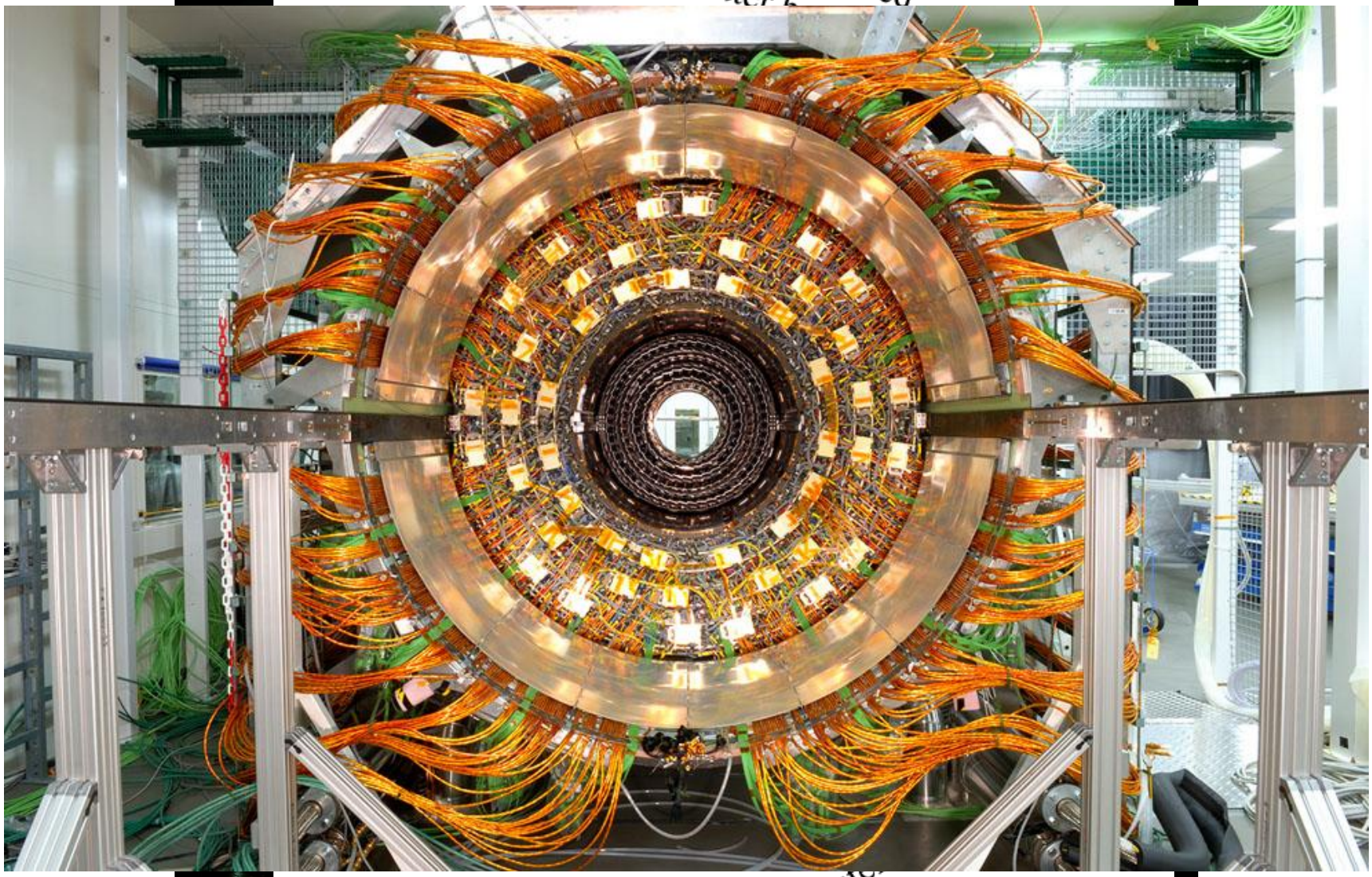
UH

Outline

- Physics motivation
- Software and Hardware Threading on GPUs
- Cluster splitting on GPU
- Scheduling
- Conclusion

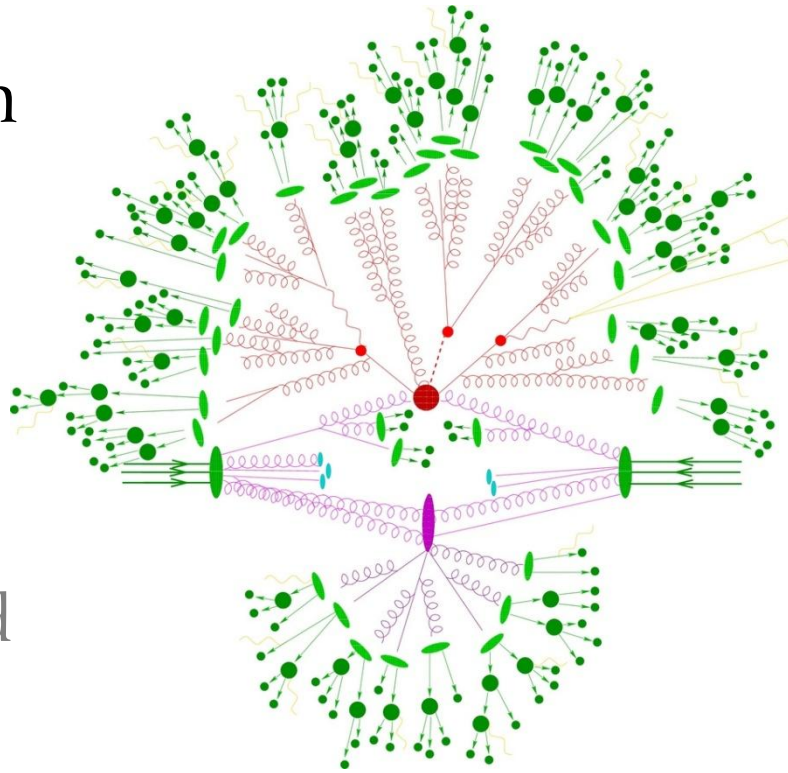
Physics Motivation

CMS Tracking system



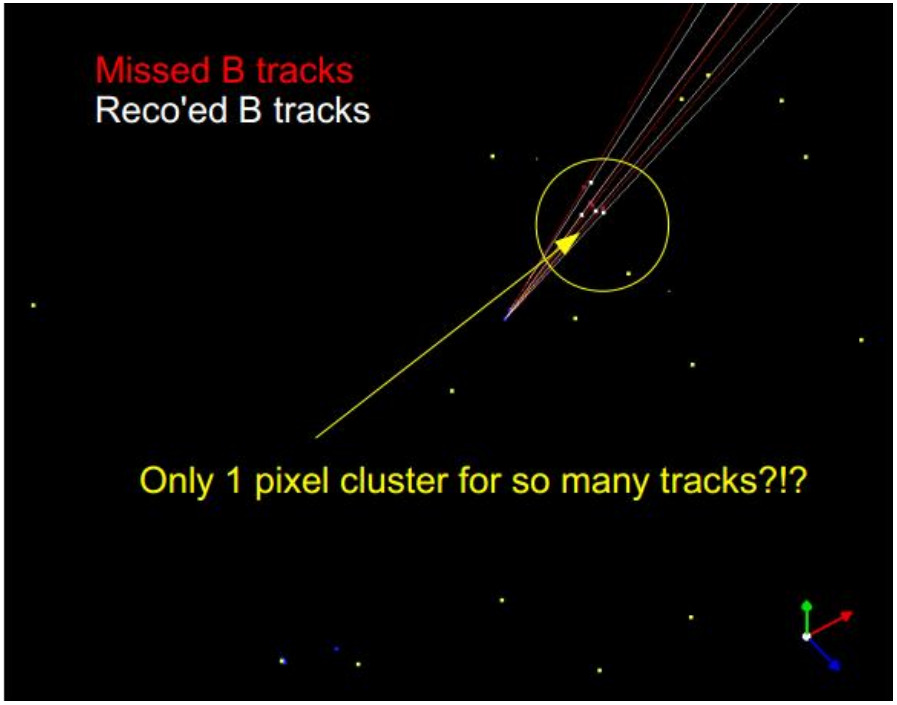
B-tagging

- Full reconstruction of 3^{rd} generation-quarks decaying in high-p_T jets
  - Searches of NP at the Energy Frontier
  - More important @ 13 TeV
  - Tracks become more collimated



Tracking vs p_T

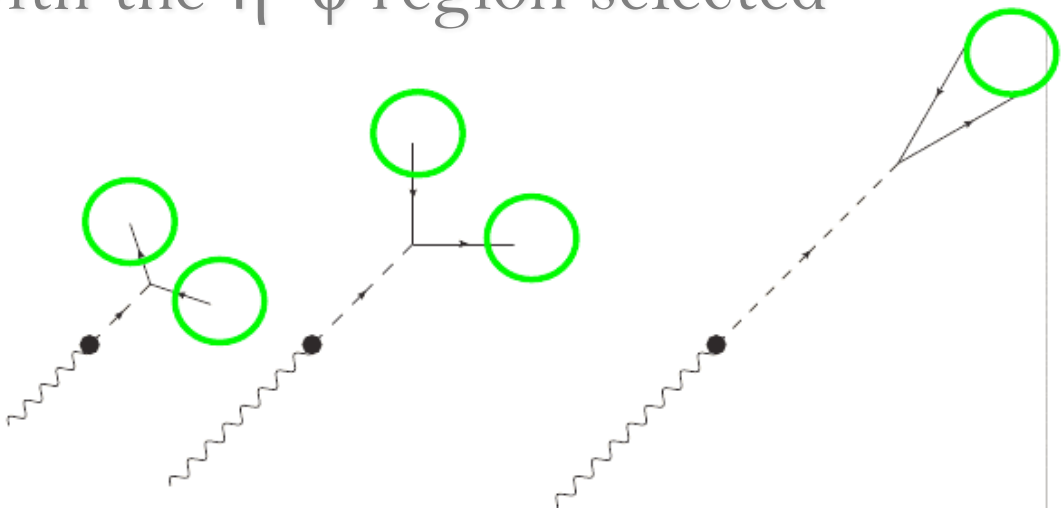
- At the moment, the same generic tracking algorithm runs for both low-p_T and high-p_T jets
  - Tracks from B-decays get more and more collimated as p_T increases (perf degradation ~30% at high p_T)
  - Fake rate increases



By A. Rizzi et al.

Cluster splitting

- Tracks leave clusters in the hitted subdetectors
- Collimated tracks in the core of high-p_T jet generate very close clusters that could be accidentally treated one single cluster
- Need to split the cluster in subclusters
  - only hits compatible with the η–ϕ region selected by the core of the jet
  - tracks with high p_T



Software and Hardware Threading on GPUs

Hardware vs Software

- From a programmer's perspective:
  - Blocks
  - Kernel
  - Threads
  - Grid
- Hardware implementation:
  - Streaming multiprocessors (SMX)
  - Warps

Thread Assignment

- Threads assigned to execution resources on a block-by-block basis.
- CUDA runtime automatically reduces number of blocks assigned to each SMX until resource usage is under limit.
- Runtime system:
  - maintains a list of blocks that need to execute
  - assigns new blocks to SM as they compute previously assigned blocks

Example of SMX resources:

- threads/block or threads/SMX or blocks/SMX
- number of threads that can be simultaneously tracked and scheduled
- shared memory

Context switching

- Registers and shared memory are allocated for a block as long as that block is active
  - Once a block is active it will stay active until all threads in that block have completed
- Context switching is very fast because registers and shared memory do not need to be saved and restored
- Goal: Have enough transactions in flight to saturate the memory bus
  - Latency can be hidden by having more transactions in flight
  - Increase active threads or Instruction Level Parallelism (ILP)

Cluster splitting on GPU

Some considerations

1. Amount of charge in the cluster ~linear wrt #subclusters
   - Expected number of subclusters
2. Each of the firing pixels could be the center of a subcluster
3. The probability that a specific position (pixel) is the core of a track depends on the fraction of charge left in that position
   - Useful to order pixels by charge
4. The number of possible combinations of k subclusters in n positions is given by:

$$C''_{n,k} = \binom{n+k-1}{k} = \binom{n+k-1}{n-1}$$

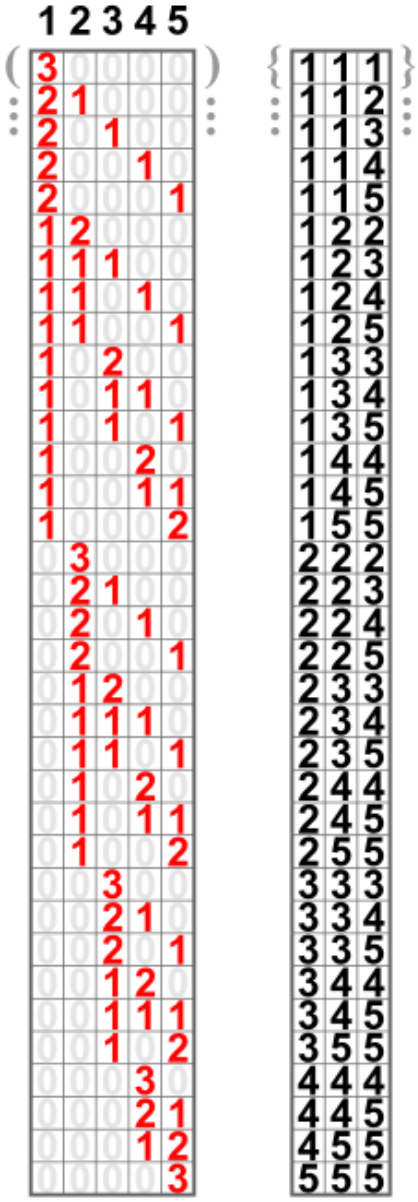
Some considerations (ctd.)

5. Linear mapping (indexing) of ordered combinations not possible
6. Overcommitting memory and execution is not an option:

$$C_R(64,6) \sim 120 \times 10^6$$

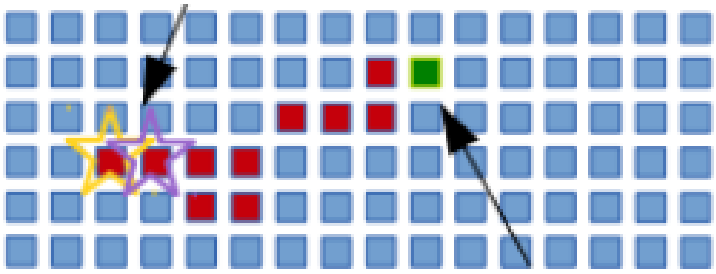
$$P_R(64,6) = 64^6 \sim 69 \times 10^9$$

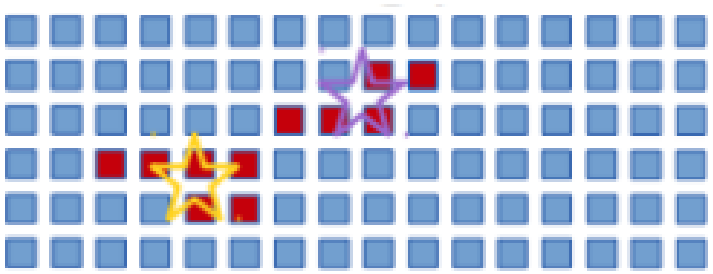
No pointers arithmetic allowed.



Simplified Algorithm Flow

```
For numClusters in [avgClusters – 1, avgClusters + 1]:
        For I in Combinations:
                For pixel in hittedPixels:
                        Compute(chi2);
                Minimize(chi2);
        Minimize(chi2);
```





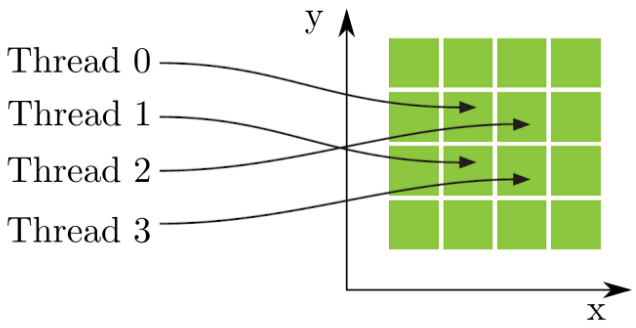
LUTs and constant data

- Constant information about the jet stored in constant memory
- A couple of big LUTs (a few MBs) need to be loaded onto the GPU
  - way too big to be loaded in constant or shared memory

Kepler architecture allows read-only data caching by using `const __restrict__` qualifiers:

Global memory (12GB) loaded through the same cache used by the texture pipeline

- no need to bind a texture beforehand
- no sizing limitations of standard textures



Dealing with exponential trend

The lack of pointer arithmetic, the need to parallelize to benefit from the GPU architecture and the impossibility to overcommit memory naturally leads to the choice of:

- exploiting fast context switching by overcommitting CUDA blocks
- warp synchronous programming
- atomics

Overcommit CUDA blocks

Overcommitting CUDA Blocks does not introduce much latency

- When a block finishes its execution another block is scheduled on the fly to run on the SMX
- Running on a grid, e.g.:

```
dim3 grid(numPositions,numPositions,1);
```

And then selecting the active blocks does not affect latency

```
if(blockIdx.x < numPositions && blockIdx.y <=
blockIdx.x && threadIdx.x <= blockIdx.y)
```

Warp synchronous techniques

- Each thread in a specific block computes the χ^2 for a specific combination
- Threads in a warp are executed in a SIMD fashion
- Shared memory is shared among the threads of a block

Warp synchronous techniques (ctd.)

Inside each block, the χ^2 minimization can be achieved by means of a warp synchronous reduction O(logN):

https://stikked.web.cern.ch/stikked/view/b9312f7b

- This technique allows to avoid barriers hence increasing kernel performance
- Each block has now a single `Chi2Comb` object that contains the minimum χ^2 and the combination information.

Atomics

On Kepler architecture, atomics can be performed as quickly as memory loads

- 64-bits maximum size
- has to contain all the combination information needed

```
struct __attribute__((__packed__)) Chi2Comb {
        int16_t chi2;
        int8_t comb[6];
};
```

- Atomics run faster on global memory than shared memory.

- As soon as each block has computed its own minimum χ^2 it is compared atomically with a global one.

https://stikked.web.cern.ch/stikked/view/a63373da

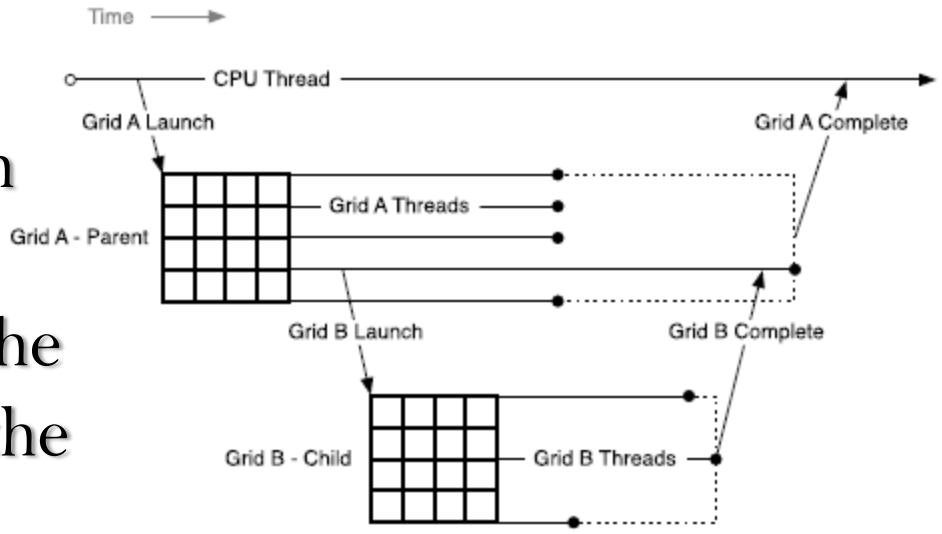
- CUDA blocks are not synchronized: atomic serialization is not a problem

CUDA Dynamic parallelism

- Enables a CUDA kernel to create and synchronize new nested work.
- A child CUDA kernel can be called from within a parent CUDA kernel
  - optionally synchronize on the completion of that child CUDA Kernel

Could be employed to compute the chi2 inside each combination

- Actually useless because the parallel-intensive loop is the outer one



Tests

HW setup

- CPU: Intel Haswell i7 4771
  - 4 physical cores
  - 3.5GHz
- GPU: NVIDIA K40
  - 12GB GDDR5 ECC
  - 875 MHz
  - 2880 CUDA GPU cores
  - CUDA compute capability 3.5
- Communication Bus: PCI Express 3

Some numbers

	4 clusters	
# of positions	sequential	GPU (ms)
32	278.55	11.93
36	879.45	21.17
48	4980	72.14

Factor 35ish between the GPU version an optimized sequential version

	5 clusters		6 clusters	
# of positions	sequential	GPU (ms)	sequential	GPU (ms)
32	28692.35	357.67	61837.64	1759.17
48	85384.9	1620.04	301399.75	8348.24
64	149908.5	4238.4	1165312.37	41258.91

Job Scheduling

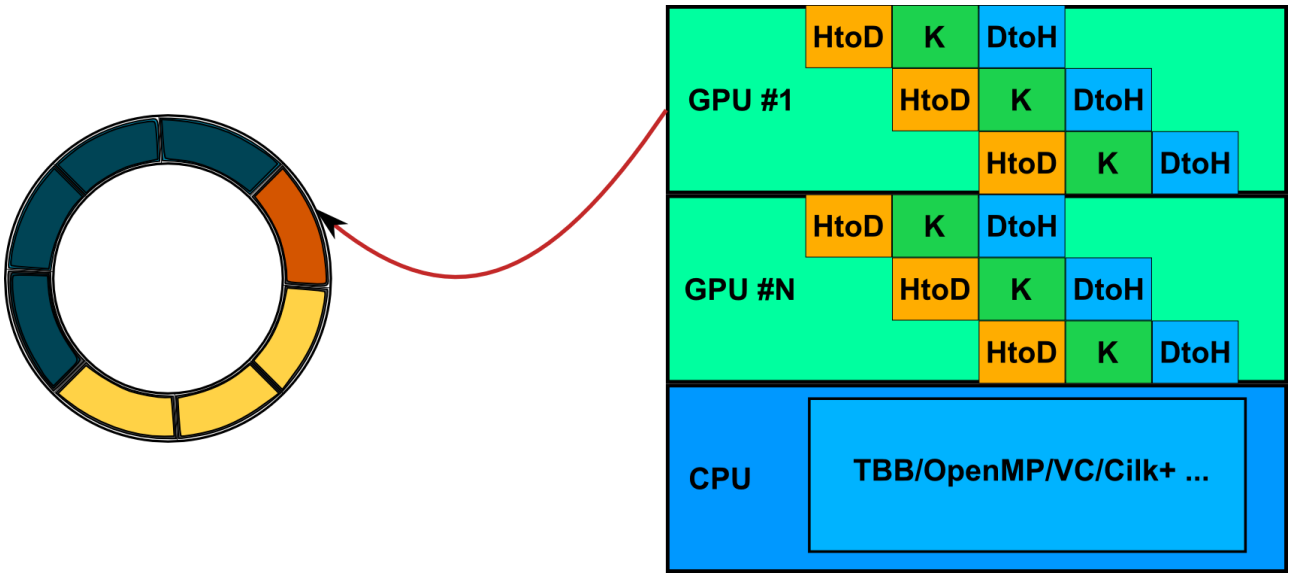
Scheduling

Take advantage of a centralized scheduler:

- Make constant memory thread-safe
- Hide latency through ILP
- Shared data between streams
- Load balancing
- Use otherwise idle CPU
- Use multiple GPUs

Scheduling (ctd.)

- tbb::concurrent_bounded_queue of JobDescriptors
- JobDescriptor contains info about input, output, device
- Pthreads pop from queue and start processing data as in JobDescriptor, asynchronously, concurrently
- Each pthread associated to one technology (e.g. TBB, CUDA, FPGA, OpenMP).
- Results returned by std::futures and std::promises



Conclusion

Conclusion

- CUDA fully integrated within CMSSW and working out-of-the-box
  - Target hybrid software design
- Some Combinatorial problems can be though to be parallelized
- Speedup of a factor ~35x wrt optimized sequential version
- GPU programming is a lot fun…