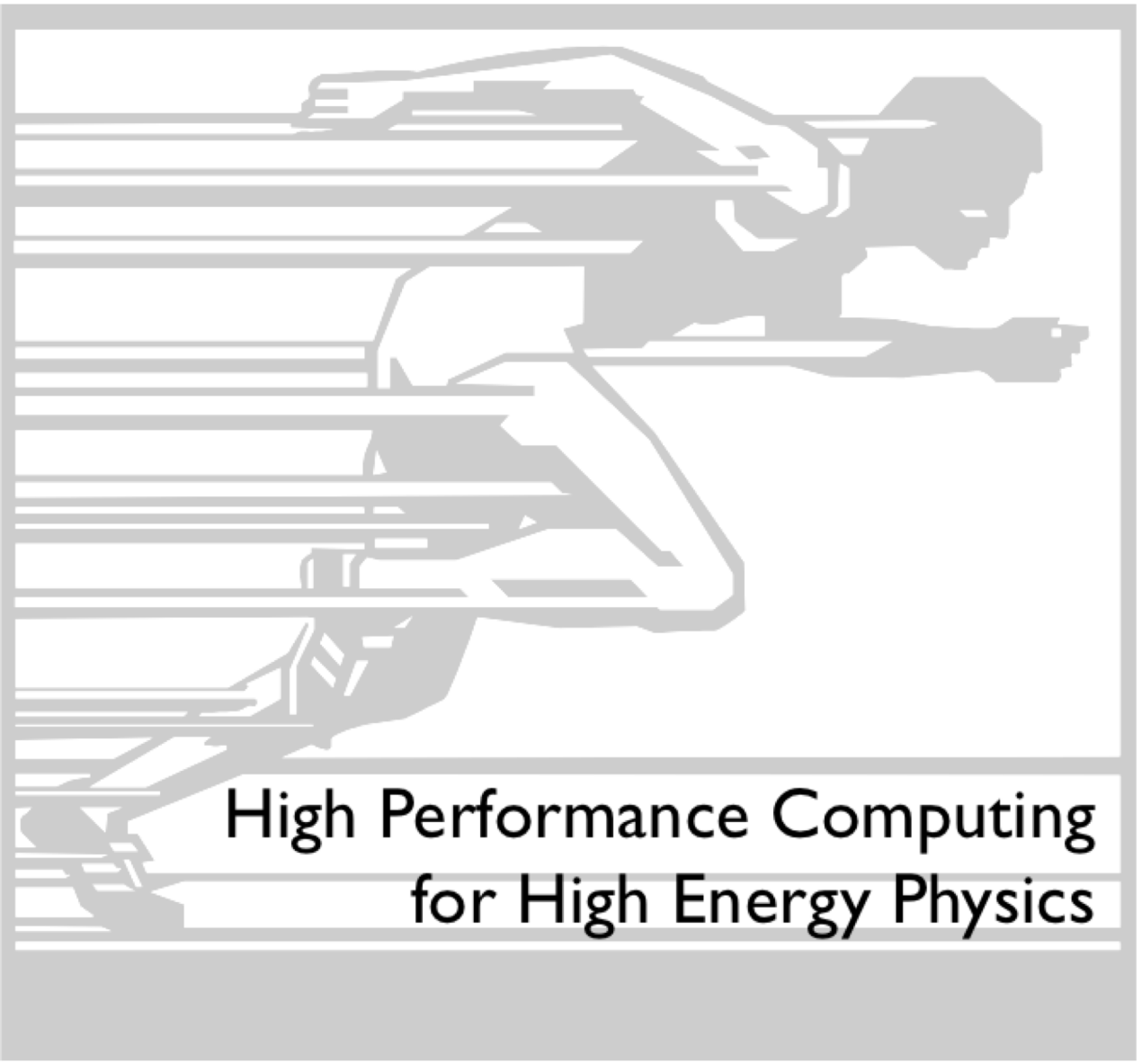
Running cuda kernels on CPU

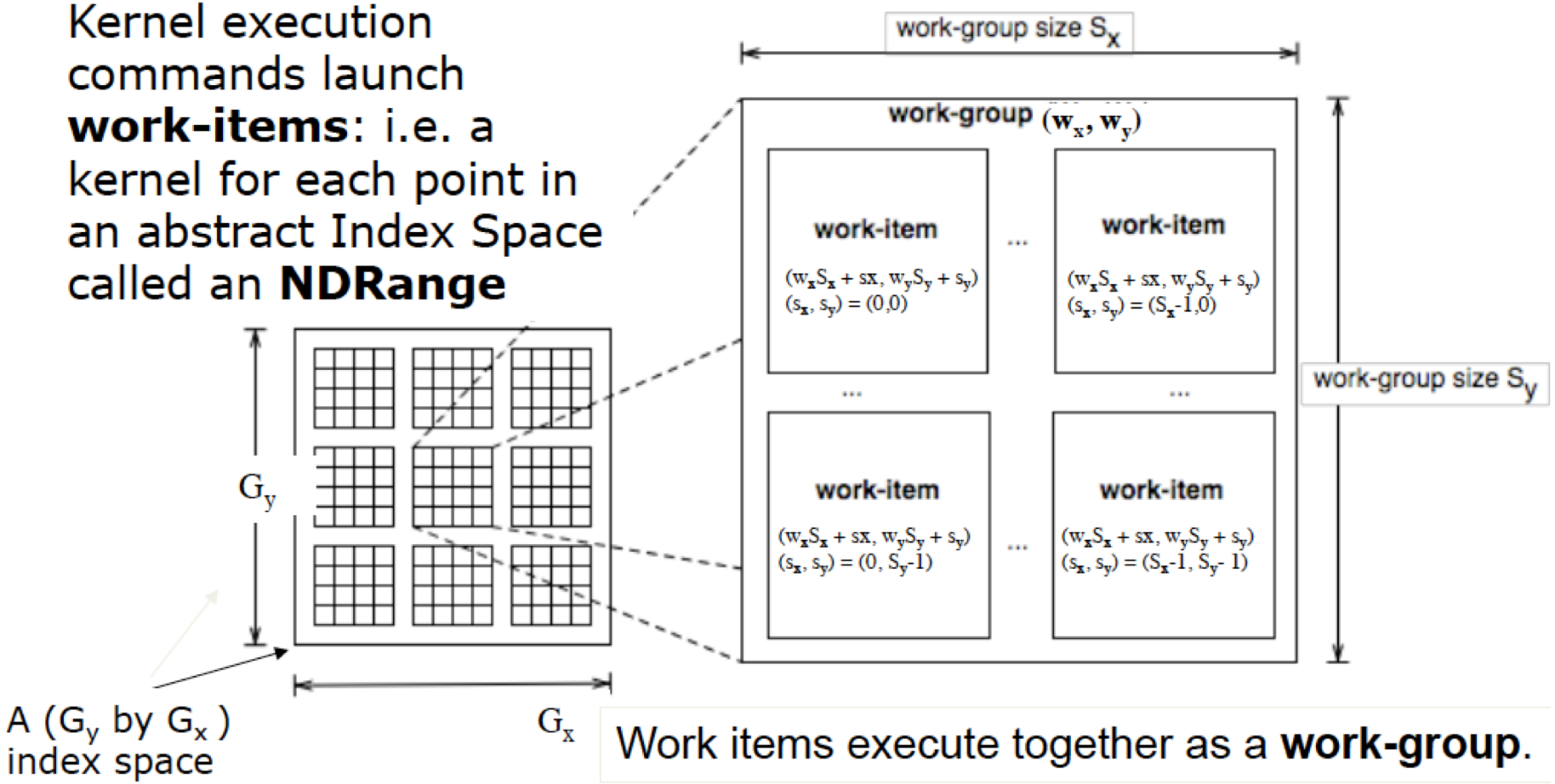


Vincenzo Innnocente
CERN/EP/SFT
CMS Experiment

OpenCL Execution Model

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue

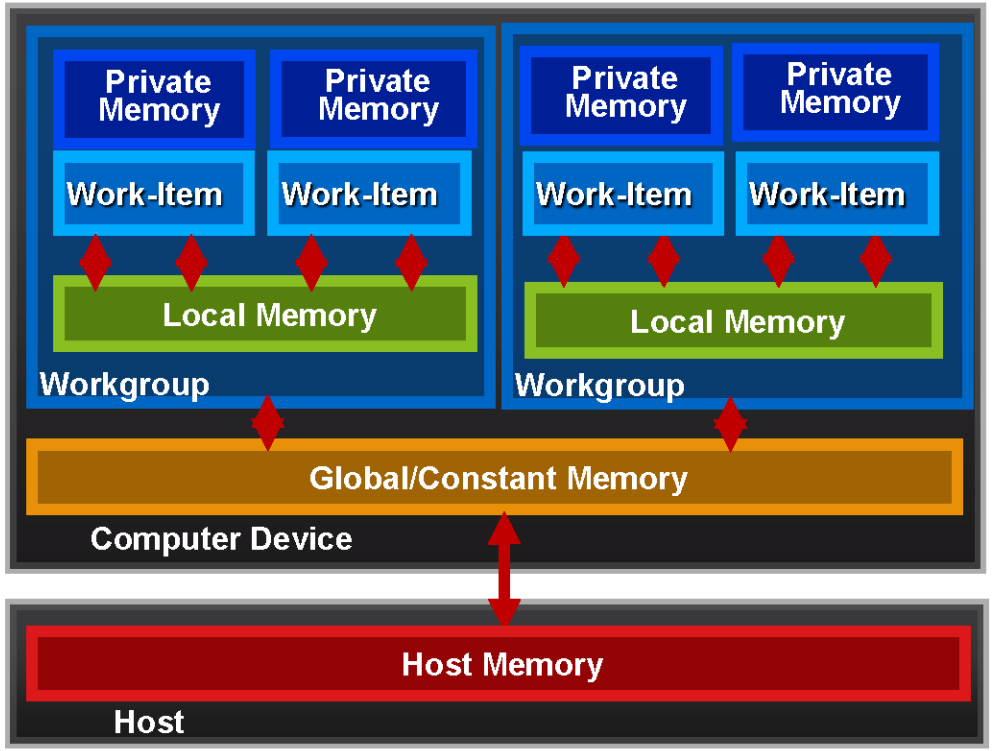
Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**



Work items execute together as a **work-group**.

OpenCL Memory Model

- **Private Memory** GPU thread
  - Per work-item
- **Local Memory** GPU block
  - Shared within a workgroup
- **Local Global/Constant Memory**
  - Visible to all workgroups
- **Host Memory**
  - On the CPU



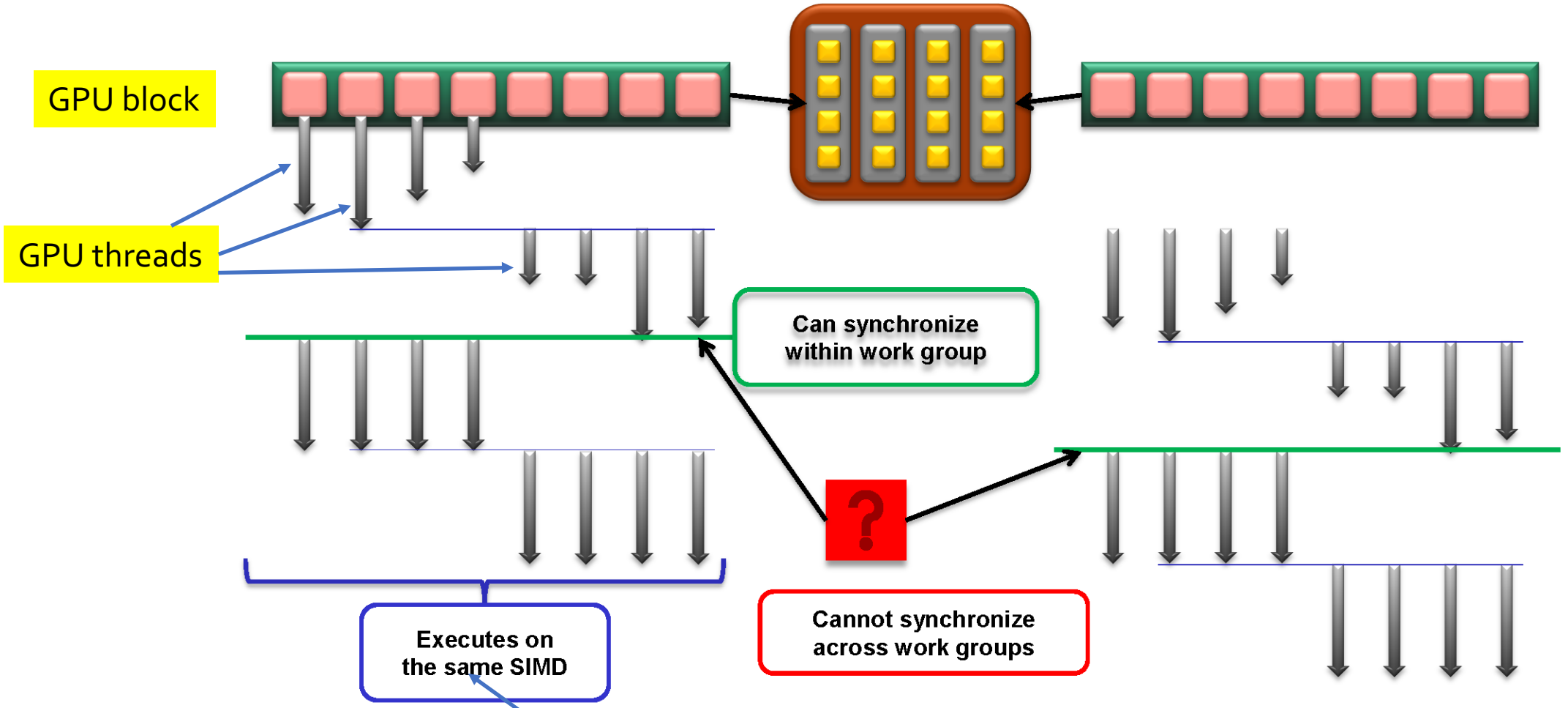
- **Memory management is explicit**
  You must move data from host -> global -> local *and* back

Whiteboard => Notebooks => Whiteboad

Memory Consistency

- **"OpenCL uses a relaxed consistency memory model; i.e.**
  - the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times."
- **Within a work-item:**
  - Memory has load/store consistency to its private view of memory
- **Within a work-group:**
  - Local memory is consistent between work-items at a barrier
- **Global memory is consistent within a work-group, at a barrier, but not guaranteed across different work-groups**
- **Consistency of memory shared between commands (e.g. kernel invocations) are enforced through synchronization (events)**

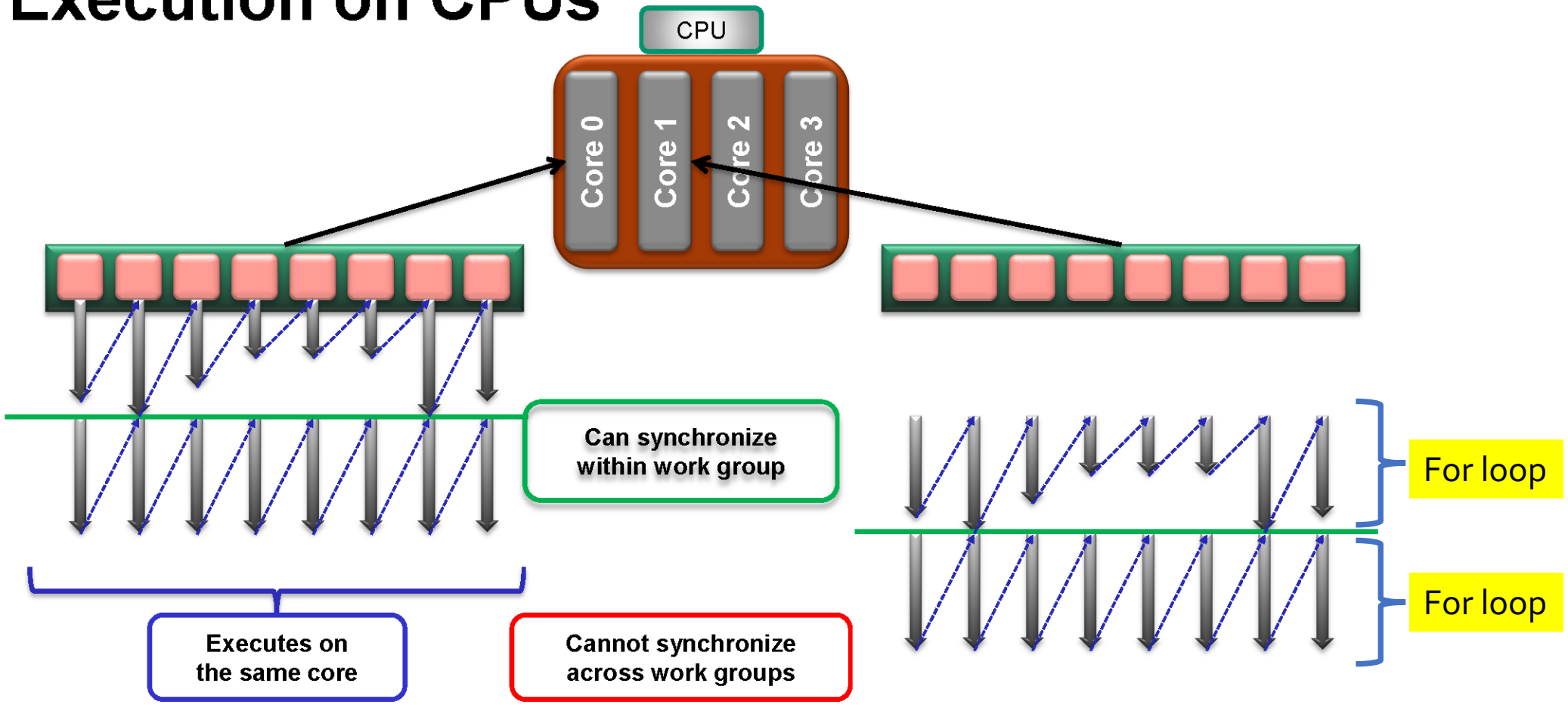
Execution on GPUs



- Page 34

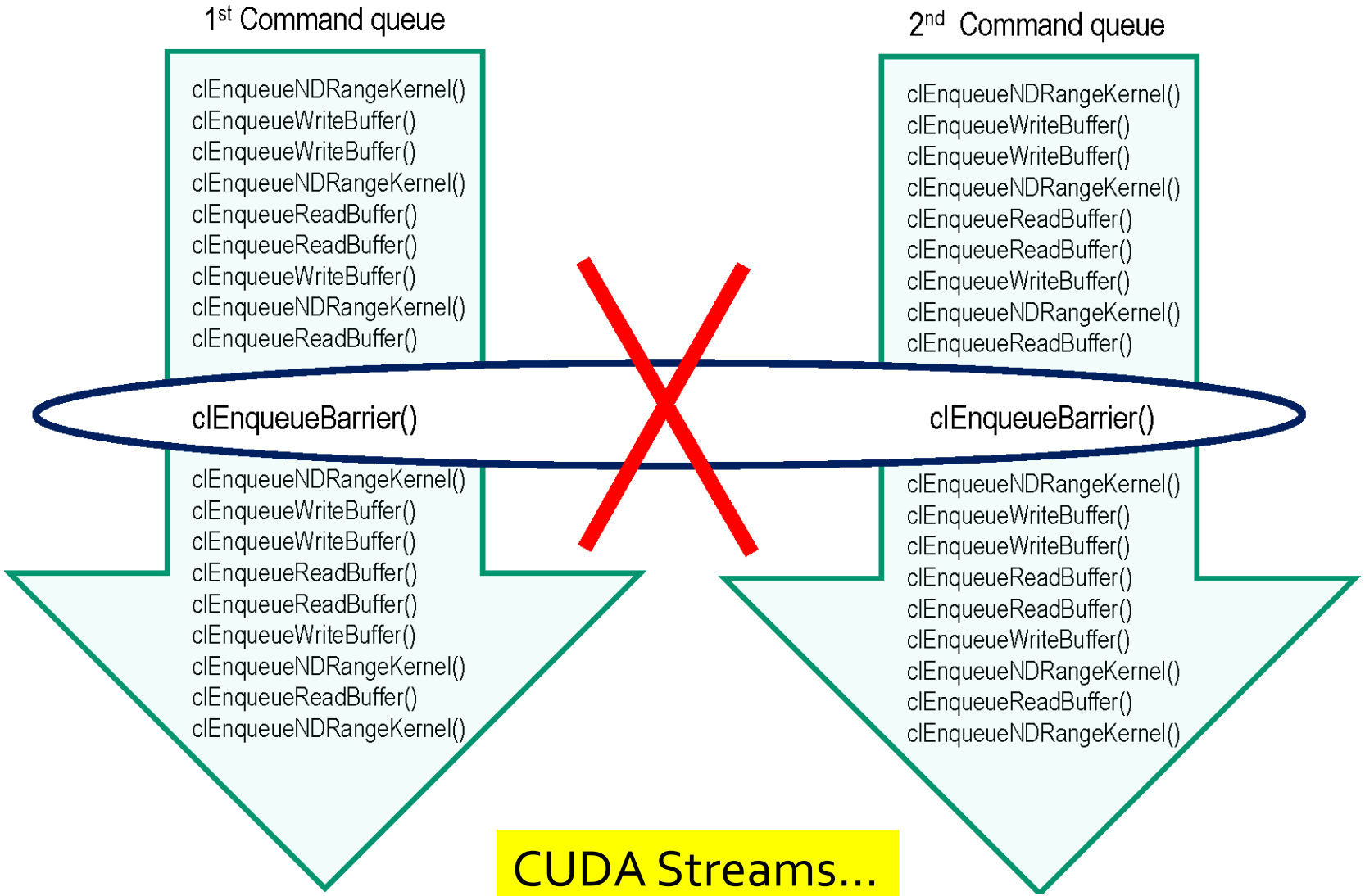
Not true, a block is a logical unit that can span many "warp"

Execution on CPUs



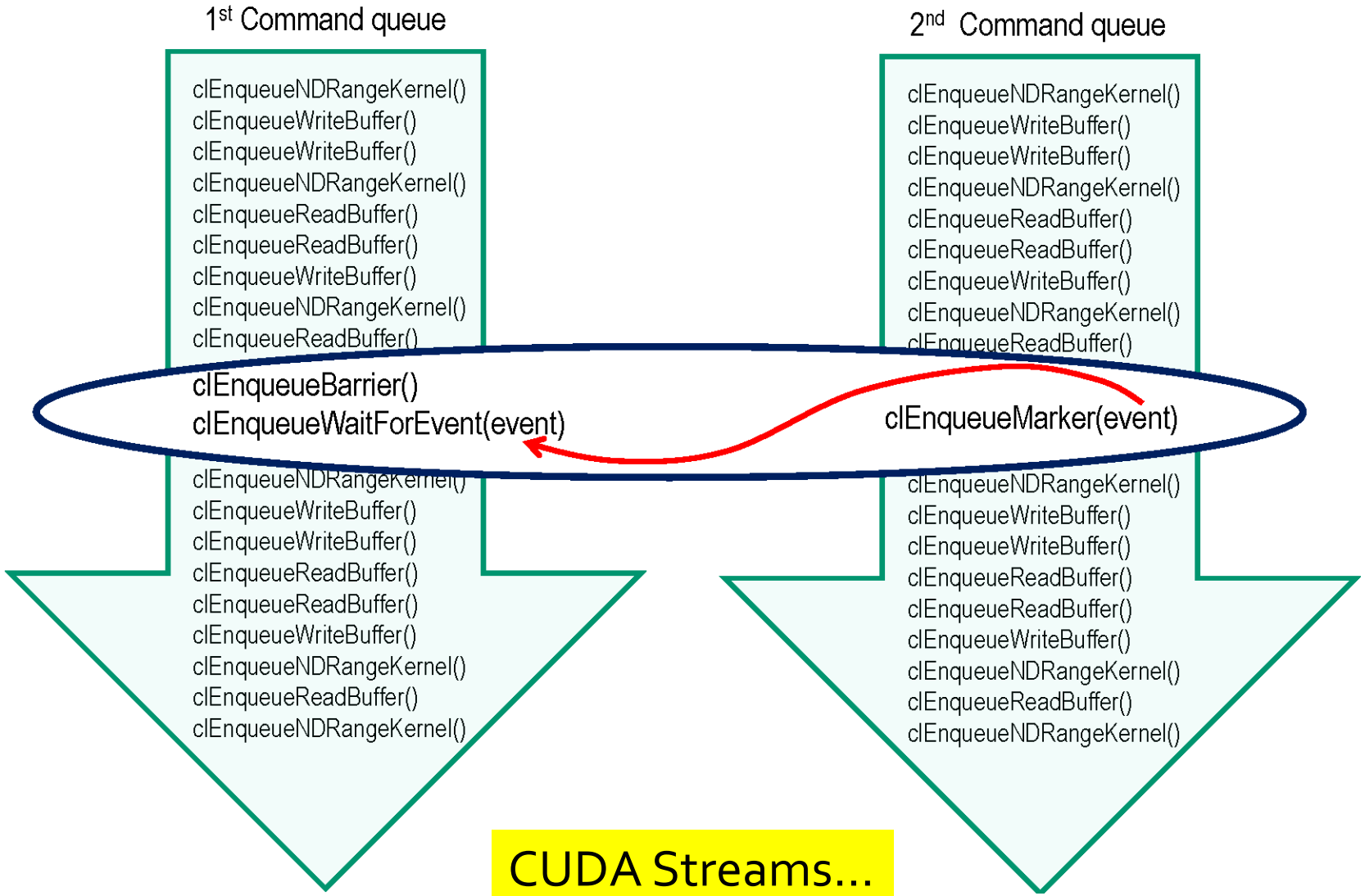
Note: Work-items may be merged by the compiler for parallel execution on SSE/AVX lanes.

Barriers between queues: clEnqueueBarrier doesn't work



CUDA Streams…

Barriers between queues: this works!



CUDA Streams...

Two execution models

- Work items are completely independent
  - No "Local Memory"
  - No sync
  - (Atomic) updates only to global memory
    - Cuda optimizes that…
- Explicit Task model that decomposes problem in work-groups
  - Usage of "Local Memory"
    - Mostly as working space
  - Explicit sync
  - Some actions performed by one work-item only
    - Typically (Atomic) updates of global memory

Independent Work-Items

```
__global__
void sum (float const * x, float * y, int n) {
  auto i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i>=n) return;  // assume gridDim.x*blockDim.x >=n
  y[i]+=x[i];
}
```

GPU

```
int threadsPerBlock = 256;  // a magic number
int blocksPerGrid =
  (numElements + threadsPerBlock - 1) / threadsPerBlock;
sum<<<blocksPerGrid, threadsPerBlock>>>(x,y, numItems);
```

```
sum<<<1,1>>>(x,y, numItems);
```

Works,
Sequential
Useful for relative benchmarks

```
__global__
void sum (float const * x, float * y, int n) {
  int first = blockDim.x * blockIdx.x + threadIdx.x;
  for (int i = first; i < n; i += gridDim.x*blockDim.x)
    y[i]+=x[i];
}
```

CPU

As easy as is
Fully sequential
may vectorize

```
threadIdx.x= blockIdx.x=0;
gridDim.x=blockDim.x=1;
sum(x,y, numItems);
```

Should work…
Can vectorize
Is of any use in CMSSW?
For sure is not generic

```
threadIdx.x= blockIdx.x=0; blockDim.x=gridDim.x =1;
tbb::parallel_for(
        tbb::blocked_range<size_t>(0,numItems),
        [&](const tbb::blocked_range<size_t>& r) {
       threadIdx.x =r.begin();
       sum(x,y,r.end()); // "n" is not what kernel expects
       };)
```

Independent Work-Items

```
__global__
void sum (float const * x, float * y, int n) {
  auto i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i>=n) return;  // assume gridDim.x*blockDim.x >=n
  y[i]+=x[i];
}
```

GPU

```
__global__
void sum (float const * x, float * y, int n) {
  int first = blockDim.x * blockIdx.x + threadIdx.x;
  for (int i = first; i < n; i += gridDim.x*blockDim.x)
    y[i]+=x[i];
}
```

CPU

```
int threadsPerBlock = 256;  // a magic number
int blocksPerGrid =
  (numElements + threadsPerBlock - 1) / threadsPerBlock;
sum<<<blocksPerGrid, threadsPerBlock>>>(x,y, numItems);
```

```
threadIdx.x= blockIdx.x=0; blockDim.x=1;
gridDim.x = numItems; // or simply MAX_INT
tbb::parallel_for(
        tbb::blocked_range<size_t>(0,numItems),
        [&](const tbb::blocked_range<size_t>& r) {
        for (size_t i=r.begin();i<r.end();++i) {
          threadIdx.x =i;
          sum(x,y, numItems);
        }
       };)
```

generic, not easy for the compiler to vectorize
Is of any use in CMSSW?

Explicit Decomposition

```
__global__  void findCluster (….) {
  if (blockIdx.x >= Nmodules) return;
  auto firstPixel = moduleStart[blockIdx.x];
  auto lastPixel = moduleEnd[blockIdx.x];
  auto first = firstPixel + threadIdx.x;
  for (int i = first; i < lastPixel; i += blockDim.x) { ….
```

- Map each "module" to a Block (work-group)
- Local Memory (__shared__) used as working space
- Synchronization after each loop on pixels

GPU

CPU

```
int threadsPerBlock = 256;  // a magic number
int blocksPerGrid = MaxNumModules;
findCluster<<<blocksPerGrid, threadsPerBlock>>>(…);
```

```
findCluster<<< MaxNumModules,1>>>(…);
```

Works,
Sequential
Useful for relative benchmarks

```
int blocksPerGrid = MaxNumModules;
threadIdx.x= blockIdx.x=0; blockDim.x=1;
gridDim.x = blocksPerGrid;
for (;blockIdx.x<gridDim.x; ++blockIdx.x)
    findCluster(…..);
```

As easy as is
Fully sequential
may vectorize inner loops

Explicit Decomposition

```
__global__  void findCluster (….) {
  if (blockIdx.x >= Nmodules) return;
  auto firstPixel = moduleStart[blockIdx.x];
  auto lastPixel = moduleEnd[blockIdx.x];
  auto first = firstPixel + threadIdx.x;
  for (int i = first; i < lastPixel; i += blockDim.x) { ….
```

- Map each "module" to a Block (work-group)
- Local Memory (__shared__) used as working space
- Synchronization after each loop on pixels

GPU

CPU

```
int threadsPerBlock = 256;  // a magic number
int blocksPerGrid = MaxNumModules;
findCluster<<<blocksPerGrid, threadsPerBlock>>>(…);
```

```
int blocksPerGrid = MaxNumModules;
threadIdx.x= blockIdx.x=0; blockDim.x=1;
gridDim.x = blocksPerGrid;
tbb::parallel_for(
        tbb::blocked_range<size_t>(0, blocksPerGrid),
        [&](const tbb::blocked_range<size_t>& r) {
          for for (size_t i=r.begin();i<r.end();++i) {
            blockIdx.x=i; findCluster(…..);
          }
        });
```

Thread parallelization should work even in generic case
Is of any use in CMSSW?

Current “header” implementation

```
#ifndef __CUDACC__
#include<cstdint>
#include<algorithm>
#include<cstring>
#include "cuda_runtime.h”

// does not support concurrent blocks on CPU
namespace cudaCompat {

#ifndef __CUDA_RUNTIME_H__
  struct dim3{uint32_t x,y,z;};
#endif
  const dim3 threadIdx = {0,0,0};
  const dim3 blockDim = {1,1,1};
  extern thread_local dim3 blockIdx;
  extern thread_local dim3 gridDim;

  inline void __syncthreads(){}
  inline bool __syncthreads_or(bool x) { return x;}
  inline bool __syncthreads_and(bool x) { return x;}
  // more __ functions here

  inline void resetGrid() {
    blockIdx = {0,0,0};
    gridDim = {1,1,1};
  }
```

```
 template<typename T1, typename T2>
 T1  atomicInc(T1* a, T2 b) {auto ret=*a; if ((*a)<T1(b)) (*a)++; return ret;}
 template<typename T1, typename T2>
  T1  atomicAdd(T1* a, T2 b) {auto ret=*a; (*a) +=b; return ret;}
  template<typename T1, typename T2>
  T1  atomicSub(T1* a, T2 b) {auto ret=*a; (*a) -=b; return ret;}
  template<typename T1, typename T2>
  T1  atomicMin(T1* a, T2 b) {auto ret=*a; *a = std::min(*a,b);return ret;}
  template<typename T1, typename T2>
  T1  atomicMax(T1* a, T2 b) {auto ret=*a; a = std::max(*a,b);return ret;}
  // more atomics….
}

#ifndef __CUDA_RUNTIME_H__
#define __host__
#define __device__
#define __global__
#define __shared__
#define __forceinline__
#endif

#ifndef __CUDA_ARCH__
using namespace cudaCompat;
#endif
#endif // __CUDACC__
```

What is currently not supported?

- Atomic are not atomic
  - But in the Histogrammer…
  - Can be done, trivial, just a pain (and not necessary in most of the contexts)
  - Shall we use atomicXTZ_block if not-atomic on CPU?
- Specific warp level instructions
  - Usually part of high level algo (sort,prefix-scan)
- External shared
  - No clue how to implement it in pure C++
  - Can be most probably supported with some convention and ifdef
- Some other things I never used?
  - Dynamic Parallelism as it invoke kernels in native cuda syntax

summary

- "Calling" Cuda kernels from sequential C++ code turns out to be trivial provided few simple rules of code robustness are followed
  - I have successfully ported all "library" kernel developed in Patatrack
- Supporting multi-threading, using either tbb or openmp, is simple even if not always generic. Its implementation can be restricted to the driver code w/o any mods to kernel code
  - This will most probably not be required in any throughput-oriented data-processing application as coarse grain (at module level) multi-threading will suffice