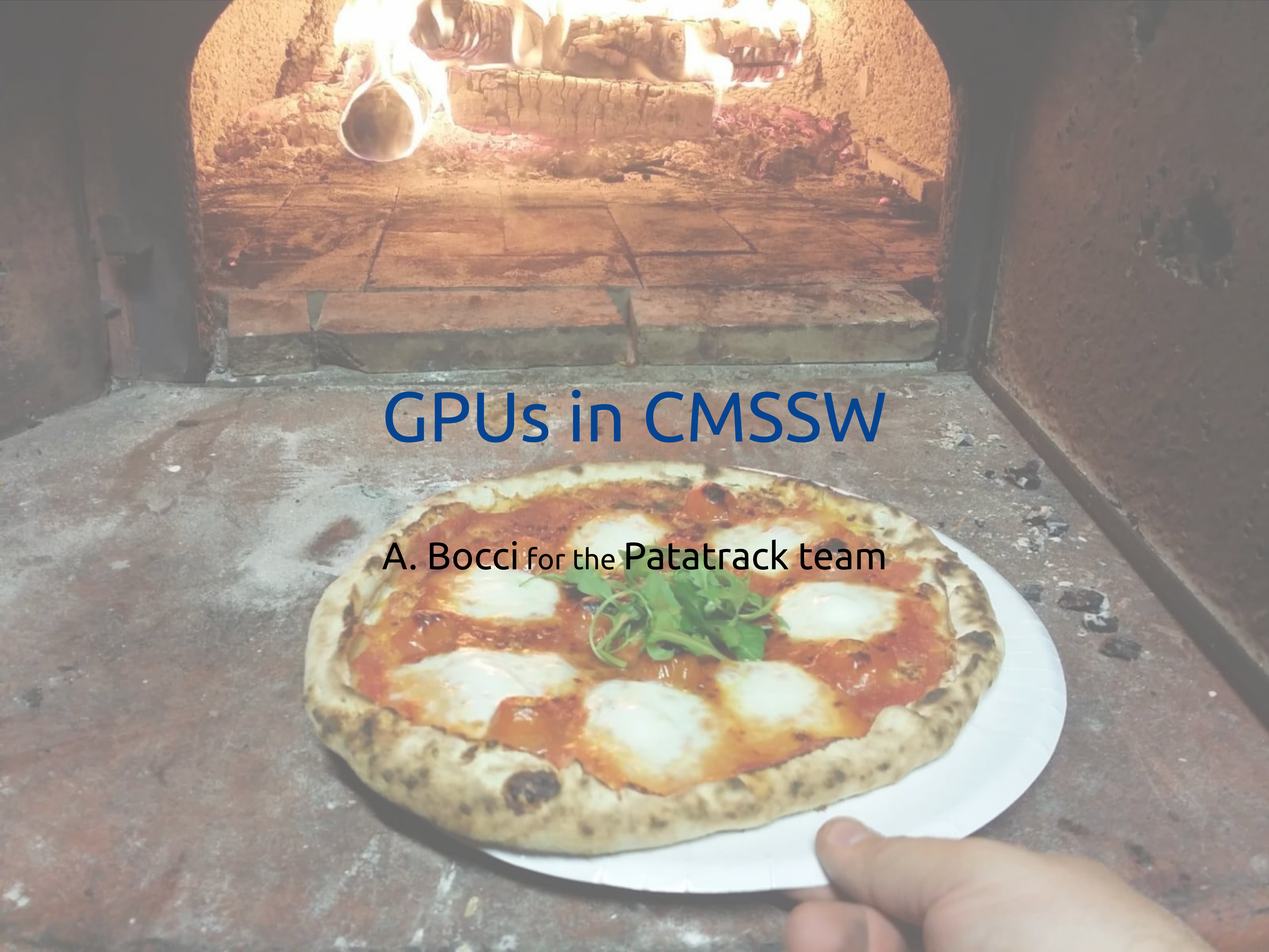


GPUs in CMSSW

A. Bocci for the Patatrack team

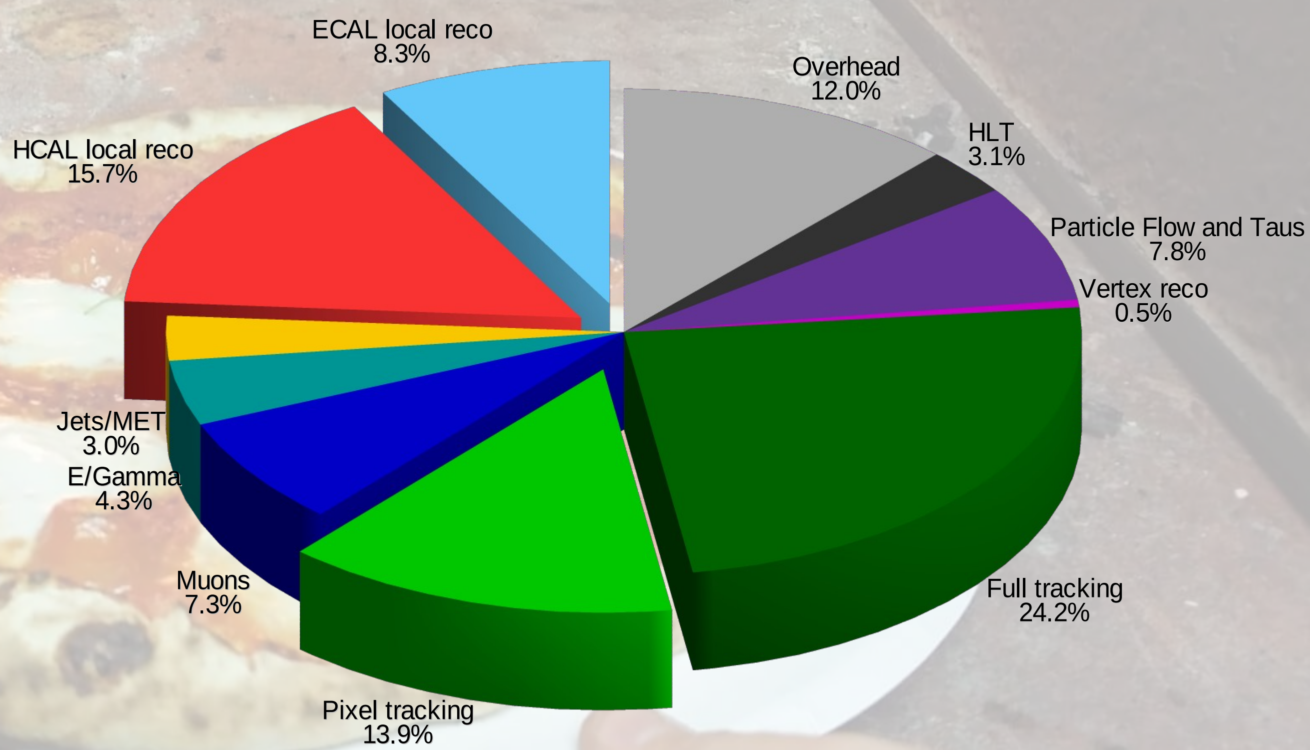
Patatrack !

- overview of the project
- what does CUDA code look like ?
- how to organise CUDA code in CMSSW
- data formats on GPUs
- framework support
  - acquire / produce semantics
  - HeterogeneousEDProducer
- future developments
  - build infrastructure
  - Unified Memory
- conclusions

current HLT menu

- time spent in the various areas in the current HLT menu
- HLT menu 2.2.0
  - CMSSW 10.1.4
  - 10k events from run 316457, ls 86

group	real time	fraction
ECAL local reco	38.9 ms	8.3%
HCAL local reco	73.9 ms	15.7%
Jets/MET	14.0 ms	3.0%
E/Gamma	20.4 ms	4.3%
Muons	34.2 ms	7.3%
Pixel tracking	65.7 ms	13.9%
Full tracking	114.2 ms	24.2%
Vertex reco	2.3 ms	0.5%
Particle Flow and Taus	36.8 ms	7.8%
HLT	14.7 ms	3.1%
Overhead	56.4 ms	12.0%
Total	471.5 ms	100.0%



the Patatrack team

- https://patatrack.web.cern.ch/patatrack/
- Ongoing activities
  - Heterogenous framework
    - Matti Kortelainen
  - Remote offloading
    - Serena Ziviani
  - Compilers and externals, integration
    - Andrea Bocci
  - Pixel tracking
    - Felice Pantaleo, Marco Rovere, Vincenzo Innocente
    - contributions from TIFR and SINP
  - HCAL local reconstruction
    - Viktor Khristenko, Vanessa Wong
  - ECAL local reconstruction
    - Andrea Massironi (expressed interest)
  - HGCAL
    - Felice Pantaleo, Marco Rovere

project goals

- September 2018
  - initial support for GPUs and CUDA in CMSSW
  - HLT pixel tracking running on GPUs
  - working demonstrator on a single machine

- Run 3
  - GPU-aware workflow completely integrated in CMSSW
  - run local reconstruction and pixel tracking on GPUs
  - deploy in production at the HLT

- Phase II / Run 4
  - support for local and remote heterogenous resources in CMSSW
  - large fraction of the HLT accelerated by GPUs and other devices
  - HGCAL reconstruction accelerated by GPUs and other devices

functions in CUDA

- kernels
  - interface between “host” and “device” code
  - declared as __global__ functions
  - called from the host (with a dedicated syntax or library call)
  - execute on the GPU on many parallel threads
- “host” code
  - default case, nothing changes
- “device” code
  - __device__ functions
  - support most C++ 14 features
    - no execptions, no std objects
  - can be called by __device__ or __global__ functions
  - can call only other __device__ functions
- code shared between host and device
  - __host__ __device__ functions
  - constexpr functions are implicitly considered as __host__ __device__

an example: square roots

- on a CPU:

```
#include <cmath>

void array_sqrt(std::vector<float> const& in, std::vector<float> & out) {
  out.clear();
  out.reserve(in.size());
  for (size_t i = 0; i < in.size(); ++i) {
    out.push_back(std::sqrt(in[i]));
  }
}
```

- in order to benefit from the parallelisation inherent to the GPU architectures, expose the parallelism in the CPU code:

```
#include <cmath>

void array_sqrt(std::vector<float> const& in, std::vector<float> & out) {
  out.resize(in.size());
  for (size_t i = 0; i < in.size(); ++i) {
    out[i] = std::sqrt(in[i]);
  }
}
```

an example: square roots

- on a GPU, in a .cu file:

```
#include <cuda_runtime.h>

__host__ __device__
float my_sqrt(float x) {
  ...
}

__global__
void kernel_sqrt(float const* in, float* out, size_t n) {
  size_t i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n) {
      out[i] = my_sqrt(in[i]);
  }
}

void array_sqrt(std::vector<float> const& in, std::vector<float> & out) {
  size_t size = in.size();
  out.resize(size);
  size_t blocks = (size + 1023) / 1024;
  kernel_sqrt<<<blocks,1024>>>(in.data(), out.data(), size);
}
```

not shown...

...GPU memory allocations

...memory copies

...synchronisations

- note
  - the kernel launch is asynchronous and returns immediately
  - allows the CPU to do something else while the GPU is working

an example: square roots

- on a GPU with c++ 14:

```
#include <cuda_runtime.h>

constexpr
float my_sqrt(float x) {
  ...
}

__global__
void kernel_sqrt(float const* in, float* out, size_t n) {
  size_t i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n) {
      out[i] = my_sqrt(in[i]);
  }
}

void array_sqrt(std::vector<float> const& in, std::vector<float> & out) {
  size_t size = in.size();
  out.resize(size);
  size_t blocks = (size + 1023) / 1024;
  kernel_sqrt<<<blocks,1024>>>(in.data(), out.data(), size);
}
```

- constexpr functions are implicitly __host__ __device__

CUDA code organisation

- device code and kernels should go in .cu files
  - can be compiled with nvcc or clang
  - caveat: nvcc does not support all root and framework headers
    - .cu files can include only a subset of the CMSSW header files

- host code stays in .h and .cc files
  - compiled as usual by gcc or clang
    - __host__, __device__, __global__ etc. attributes are ignored by non-CUDA compilers
    - can be safely used in host code

- all the device code involved in a kernel launch must be in the same library

CUDA code in CMSSW

- example 1
  - .../interface:
    - utils.h ← __host__ __device__ function declarations and data structures
  - .../src:
    - utils.cu ← __host__ __device__ function definitions
  - .../plugins:
    - algo.h, algo.cu ← __global__ kernel
    - producer.cc ← kernel launch

CUDA code in CMSSW

- example 1
  - .../interface:
    - utils.h ← __host__ __device__ function declarations and data structures
  - .../src:
    - utils.cu ← __host__ __device__ function definitions
  - .../plugins:
    - algo.h, algo.cu ← __global__ kernel
    - producer.cc ← kernel launch

- today this does not work !

CUDA code in CMSSW

- example 2
  - .../src:
    - utils.h ← __host__ __device__ function declarations and data structures
    - utils.cu ← __host__ __device__ function definitions
    - algo.h, algo.cu ← __global__ kernel
  - .../plugins:
    - producer.cc ← kernel launch

CUDA code in CMSSW

- example 2
  - …/src:
    - utils.h ← __host__ __device__ function declarations and data structures
    - utils.cu ← __host__ __device__ function definitions
    - algo.h, algo.cu ← __global__ kernel
  - …/plugins:
    - producer.cc ← kernel launch

- today this does not work, either

CUDA code in CMSSW

- example 3
  - .../src:
    - utils.h ← __host__ __device__ function declarations and data structures
    - utils.cu ← __host__ __device__ function definitions
    - algo.h, algo.cu ← __global__ kernel, kernel_wrapper() function to launch the kernel
  - .../plugins:
    - producer.cc ← call kernel_wrapper()

- finally, this works

- the downside is that we cannot build libraries for device code
  - e.g. DataFormats

CUDA code in CMSSW

- present approach
  - device code local to each package
    - inline / constexpr
    - object files linked only inside each package
- next step
  - support (static) libraries of CUDA device code
    - limitation to static libraries from the CUDA linker
  - supported use cases
    - device code and kernels in the same library or package
    - device code in one library, kernel in a second library, launch in a separate package
  - working proof-of-concept
    - using nvcc with gcc 7.x
    - using clang 7 (trunk)
  - to be implemented in SCRAM

data formats for a GPU

- memory allocations on a GPU are expensive
  - avoid std::vectors, std::maps, etc.

- memory copies to and from a GPU are expensive
  - avoid many small copies
  - avoid copying before and after each “module”

- our approach
  - use simple data structures (PODs, arrays, structs, SOAs, …)
  - preallocate buffers for the whole job
  - data stays on the GPU across multiple modules
  - pass around pointers into GPU memory
  - copy only the final results back to the host memory

parallel work on a CPU and GPU

- work on a GPU can be scheduled to run asynchronously
  - we should avoid blocking a host thread while the GPU is working

- framework support for EDProducers:
  - acquire(...) { ... }
    - run instead of the standard produce() method
    - reads data from the Event
    - schedules asynchronous copies from the host to the GPU
    - schedules kernel launches
    - register a callback to notify the framework when the work is done
  - produce(...) { ... }
    - run after the callback returns
    - optionally, copies data back from the GPU
    - put a product in the Event

- logically simple, somewhat complicated toimplement

HeterogeneousEDProducer

- wraps the common functionality in a base class
  - stream::EDProducer
  - acquire() / produce() semantic
  - produces a HeterogeneousProduct
    - keeps intermediate products on the GPU
    - copies final products to the host

- HeterogeneousCore/Producer/interface/HeterogeneousEDProducer.h

memory management

- traditional approach:
  - allocate on the host
  - fill on the host
  - allocate on the device
  - copy from host to device

- disadvantages
  - schedule explicit memory copies (can be asynchronous)
  - use different pointers on the host and device

- for a struct of buffers, we need multiple intermediate steps…
- pointer hell !

simplified memory management

- CUDA Unified Memory
  - automatically migrate memory pages among the host and the CUDA devices
  - greatly simplifies the device memory management

- Unified Memory approach
  - allocate with cudaMallocManaged()
  - use on the host
  - use on the device
  - …
  - the same pointer value is valid on the host and all devices !

- downside
  - page faults and on-demand copies are less performant
  - can be fixed with prefetching and hints

status of the project

- fully integrated in CMSSW
  - forked on GitHub: https://github.com/cms-patatrack/cmssw/
  - documented on the patatrack web site
- semi-automatic testing and regressions
  - compare the results of pixel tracking on the CPU and GPU
  - run for each pull request
    - e.g. https://github.com/cms-patatrack/cmssw/pull/87#issuecomment-401348619
- plan to submit for integration upstream after the Summer
- next event:

  4th Patatrack Hackathon, 3-5 September, IdeaSquare (CERN)