

Day 2

1st NextGen Hackathon



NextGen
Next Generation Triggers

```
#define  
GENERATE_SOA_LAYOUT(  
TheBestTeam,
```

```
SOA_COLUMN(Members,
```

```
(Jolly Chen, Leonardo Beltrame, Oliver Rietmann, Simone Balducci,  
Simone Rossi Tisbeni, Luca Ferragina, Aurora Perego, Davide Gadioli)
```

```
))
```



NexTGen
Next Generation Triggers

What happened today

- Exploring **SoAs** in CMS
 - SoA generation with Boost::PP library
 - Allocating host/device memory using PortableCollections
- Discussing **planned works** and **new ideas** for NextGenTrigger's SoA library
 - Views of sub-ranges for SoA
 - Print the SoA (operator <<)
 - Device/Host memory management
 - Compatibility with the standard library (e.g. iterators)
 - Support for AoS

Plan for tomorrow

- Presentation on SoA with template metaprogramming
- Hands On: Try out our library
- Compare to the other approaches
- Discuss new features

Running event generators on GPU

Daniele Massaro

Davide Di Croce

Johannes Junggeburth



NexTGen
Next Generation Triggers

Introduction to your topic

- MadGraph is a Monte Carlo event generator for high-energy physics.
- It is an automatic meta-code that writes the code for computing the cross-section and generating events for any process at colliders.
- It can:
 - generate all the feynman diagrams for a certain process;
 - automatically compute the cross section for that process;
 - generate events and store them in LHE format;
 - interface with other tools (like for shower and hadronisation - Pythia).
- Originally, MadGraph has been developed with only CPUs in mind, and the event generation can be parallelised on multiple cores.
- Recently, IT-FTI released a new plugin that allows MadGraph to use hardware acceleration for part of the event generation.
- This means using both GPUs and CPUs with vectorised instructions.

Plan for the week

- [MadGraph introduction](#);
- [Getting familiar with MadGraph](#):
 - how to install;
 - first event generation;
 - change the configuration;
 - install and link additional tools (e.g. Pythia);
 - AOB and any suggestions welcome.
- [The cudacpp plugin](#), a.k.a. running MadGraph on GPU/vectorized CPU:
 - the MadGraph plugin system and how to install;
 - details on hardware acceleration in MadGraph;
 - perform a MadGraph run using `cudacpp` plugin;
 - various possible settings.
- [Finding bottlenecks in the code](#):
 - profile MadGraph;
 - using [Adaptyst](#) to obtain flamegraphs;
 - discuss future updates.

Plan for tomorrow

- Finalize running MadGraph on GPUs:
 - inspect throughput and GPU usage.
- Explore profiling options, and create flamegraphs for the main executable.
- Hands-on with Adaptyst profiler.

Fast ML Inference

Lorenzo Moneta, Anastasiia Petrovych, Enrico Lupi,
Alessandro Crespi, Sanjiban Sengupta
Jamie Gooding, Jonas Rembser



NexTGen
Next Generation Triggers

What happened today?

- Explore SOFIE IR
- Develop Matrix operations using SOFIE - Gradient development
- SOFIE Standalone
 - Experimentation with linking ROOT binary with SOFIE (inspiration from RooFit)
 - Experimentation with Dockerfile build
 - Discussion on its build structure
- Broadcast parametric tensors in SOFIE
- Integrated SOFIE generated code into SBI statistical inference framework

Plan for tomorrow

- Release SOFIE Standalone
- Continuing Matrix operations support
- SOFIE Memory Optimization
- Experiment SOFIE with user models- Smart Pixels, UNet
- Continuing on SOFIE Dynamic Computation support

Fast ML Inference using hls4ml

Dimitrios Danopoulos, Roope Niemi



NextGen
Next Generation Triggers

Introduction to your topic

- Hls4ml is a package for ultra-low latency targeting FPGA inference
 - Parses a trained model in ONNX, Keras or PyTorch
 - Generates optimized inference code (HLS) for the target FPGA

- PQuant is a library for training compressed ML models (public from today)
 - Helps optimizing the models (pruning, quantization) before deploying them to hls4ml
 - Currently supports PyTorch (TF under development)

What happened today

- Install the dependencies and build hls4ml and PQuant libraries ✓
- Profile the models, identify bottlenecks/problematic layers
- Perform optimization on the SW level (use PQuant, HGQ)
 - We ran the PQuant tutorial, experimented with using it in some of the projects below
- fAD: trying to port normalizing flows to FPGAs for the first time and testing them in a pseudo-realistic scenario of anomaly detection at 40MHz in CMS

| Project title / ID | Description | DL framework | Type of model + params | Plan |
|--------------------|---|------------------------|---|--|
| QDips | Optimize jet-tagging model, as an exploration of Deep Sets architecture | QKeras / TF | Conv1D -> custom sum -> dense | Make model definition more hls4ml friendly and remove manual quantization definition in model hls4ml config (and use QKeras one). |
| NGTWP.2.2 | Enhance L0 MDT Trigger | Keras | Conv1D,Conv2D,RNN,LSTM,Dense | 1. Test optimization on Convolutional models. 2. test LSTM layers |
| 2 FCNs | Convert 2 FCN models for Jets Reconstruction | PyTorch/Brevitas/QONNX | Conv2D, MaxPool2D, AvgPool2D, Upsample/ConvTranspose2D, BatchNorm2D, ReLU | 1. Identify problematic layers 2. Resolve issues 3. Begin optimization |
| CaloJetSSD | Calorimeter Jet Single Shot Multibox Detector for ATLAS Trigger System | PyTorch/Brevitas | Too Many | Continue to try and convert it with HLS4ML |
| fAD | flow-based Anomaly Detection for triggers | PyTorch | Dense; but custom forward? | 1. get a decently performing model 2. understand if I want a custom forward to solve the ODE or I should switch to a FF model 3. distill it to a good complexity level 4. test the actual export to FPGA |

Plan for tomorrow

Continue experimenting with PQuant

- Test different pruning methods / bitwidths, apply layer-wise quantization

Replace problematic layers with hls4ml-friendly equivalents

Begin synthesizing models for FPGA

- Experiment with optimization strategy, reuse factor, precision
- Identify bottlenecks, check resource usage, etc.

Torch Alpaka Inference in CMSSW

Leonardo Beltrame, [Lukasz Michalski](#), Davide Valsecchi, Christine Zeh



NextGen
Next Generation Triggers

Introduction to your topic

Torch + Alpaka Fast ML Inference

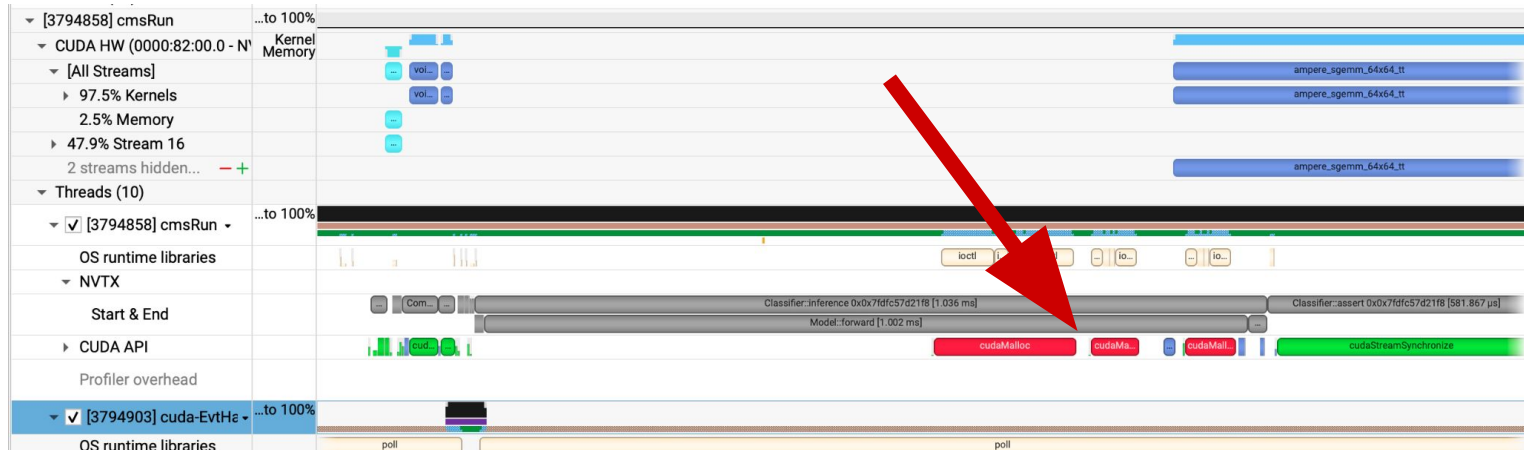
- **Memory Optimizations:**
 - Interface for direct conversion of SoA data into PyTorch tensors without explicit data movement.
 - Leverages GPU memory and uses memory-stealing to reduce unnecessary copy operations.
 - Supports optimized memory layout, computation of tensor strides, and flexible column/tensor list ordering.
 - Simplifies integration with ML models, especially for users of PortableCollection concepts in CMSSW.
- **CMSSW Integration:**
 - Fully integrated into the CMSSW framework, ensuring controlled execution without hidden multithreading or process spawning by PyTorch
 - Prevents use of Torch's internal thread pool, preserving CMSSW resource management strategy.
 - Enables fast and seamless use of industry-standard ML libraries inside the framework.
 - Aligned with ongoing improvements in Alpaka, promoting future-proof, portable module development.

Plan for the week

- **Integrate Torch-Alpaka Inference** in cmssw, open PR to collect feedback
- **Improve interface for SoA to torch::Tensor** improvements possible with merged PR: <https://github.com/cms-sw/cmssw/pull/47306>
- **Profiling & Optimization:**
 - Ensure compatibility with CMSSW's execution model.
 - Ensure we are able to lock Torch internal optimizations (multithreading, Alpaka GPU queues).
- **Explore Ahead-of-Time (AOT) Compilation:** Investigate how AOT can improve model execution instead of relying on just-in-time (JIT) compilation.

What was achieved today

- Move forward with sanity checks regarding our implementation in cmssw.
 - Some issues detected that require more in depth investigations and understanding
- Playground with AOT compilation provided by torch and attempt to manually compile it in cmssw environment
 - Let's say progress is being made, albeit in small steps



What was achieved today

- Move forward with sanity checks regarding our implementation in cmssw.
 - Some issues detected that require more in depth investigations and understanding
- Playground with AOT compilation provided by torch and attempt to manually compile it in cmssw environment
 - Let's say progress is being made, albeit in small steps

```
void ATInductorModel::run_impl()
{
    AtanTensorHandle
    input_handles, // array of input AtanTensorHandle handles
    // are stolen; the array itself is borrowed
    AtanTensorHandle
    output_handles, // array for writing output AtanTensorHandle handles
    // will be stolen by the caller; the array itself is
    // borrowed
    DeviceStreamType stream,
    ATProxyExecutorHandle proxy_executor
    {
        auto inputs = steal_from_handles_to_handles(input_handles, 1);
        auto arg_L1 = std::move(inputs[0]);
        [inputs_unused] auto fcl_weight = constants_weight();
        [inputs_unused] auto fcl_bias = constants_bias();
        [inputs_unused] auto fcl_weight = constants_weight();
        auto arg_L1_size = arg_L1.size();
        int64_t n = arg_L1.size();
        int64_t n_batches = static_cast<int64_t>(n/n_batches_per_batch());
        const int64_t n_batches_per_batch() const { return n_batches_per_batch(); }
        static constexpr int64_t n_batches_per_batch() = 16L;
        AtanTensorHandle buf2_handle;
        AT_TENSOR_CODE_CHECK_ARGS(torch_ops_strided02, int_array_4, int_array_3, cached_torch_type_float32, cached_torch_device_type_cpu, this->device_id(), buf2_handle);
        RAIIAtanTensorHandle buf2(buf2_handle);
        // Topologically Sorted Source Nodes: (Linear), Original Atan: (aten.addmm)
        static constexpr int64_t int_array_4[] = {0L, 0L};
        static constexpr int64_t int_array_3[] = {0L, 0L};
        auto tmp_tensor_handle_0 = makeTensorHandle(fcl_weight, 2, int_array_4, int_array_3, 0L);
        RAIIAtanTensorHandle tmp_tensor_handle_0(tmp_tensor_handle_0);
        AT_TENSOR_CODE_CHECK_ARGS(torch_ops_addmm_buf2, fcl_bias, arg_L1, tmp_tensor_handle_0, 1L, 1L);
        arg_L1.reset();
        auto buf2 = std::move(buf2); // reuse
        static constexpr int64_t int_array_4[] = {0L, 0L};
        AtanTensorHandle buf2_handle;
        AT_TENSOR_CODE_CHECK_ARGS(torch_ops_strided02, int_array_4, int_array_3, cached_torch_type_float32, cached_torch_device_type_cpu, this->device_id(), buf2_handle);
        RAIIAtanTensorHandle buf2(buf2_handle);
        // Topologically Sorted Source Nodes: (relu, Linear), Original Atan: (aten.relu, aten.addmm)
        static constexpr int64_t int_array_4[] = {0L, 0L};
        static constexpr int64_t int_array_3[] = {0L, 0L};
        auto tmp_tensor_handle_1 = makeTensorHandle(fcl_weight, 2, int_array_4, int_array_3, 0L);
        RAIIAtanTensorHandle tmp_tensor_handle_1(tmp_tensor_handle_1);
        AT_TENSOR_CODE_CHECK_ARGS(torch_ops_addmm_buf2, buf2, tmp_tensor_handle_1, 1L, 1L);
        buf2.reset();
        auto buf2 = std::move(buf2); // reuse
        // Note: handle is stolen; buf2 is not; this is a workaround
        output_handles[0] = buf2.release();
    } // ATInductorModel::run_impl()
} // namespace torch::inductor
```

```
#include "<torch/inductor_lmichals/2r/c2rn1spx43ivnz4u4ieul65kx65dfhfbbtbb5og4wk6rgebuxoo.h">
extern "C" void cpp_fused_relu_0(float* in_out_ptr0,
                                float* out_ptr0,
                                const int64_t ks0)
{
    for(int64_t x0=static_cast<int64_t>(0L); x0<static_cast<int64_t>(16L*ks0); x0+=static_cast<int64_t>(16L))
    {
        if(C10_LIKELY(x0 >= static_cast<int64_t>(0) && x0 < static_cast<int64_t>(16L*ks0)))
        {
            auto tmp0 = at::vec::Vectorized<float>::loadu(in_out_ptr0 + static_cast<int64_t>(x0), static_cast<int64_t>(16));
            auto tmp1 = at::vec::clamp_min(tmp0, decltype(tmp0)(0));
            tmp1.store(in_out_ptr0 + static_cast<int64_t>(x0));
        }
    }
    auto tmp0 = static_cast<float>(0.037419140338897705);
    out_ptr0[static_cast<int64_t>(0L)] = tmp0;
}
```

Plan for tomorrow

- Investigate cudaMalloc operations during inference in cmssw.
 - In principle we are reusing SoA mem buffers, torch is doing internal allocations for intermediate stuff - I had already seen such behavior in ONNX runtime, need check if we can get rid of that completely
- Update our conversion scheme with latest changes in SoA in cmssw,
 - Simplify interface by using SoA metadata rather than relying on user understanding of interface.
- Move forward with AOT topic and concept utilization in our use case
 - Computational graph capture - Matrix multiplications, convolutions, activations etc.
 - Benefit from precompiled kernels, layout, ops, fixed parameters/graph shape, unroll loops, branching etc.
 - Allocation and layout decisions can be frozen so that runtime has no overhead figuring that out.
 - No warm-up cost - plug and play no runtime compilation

C++ Coroutines

Mateusz Jakub Fila, Eric Cano, Axel
Naumann, Attila Krasznahorkay



NexTGen
Next Generation Triggers

