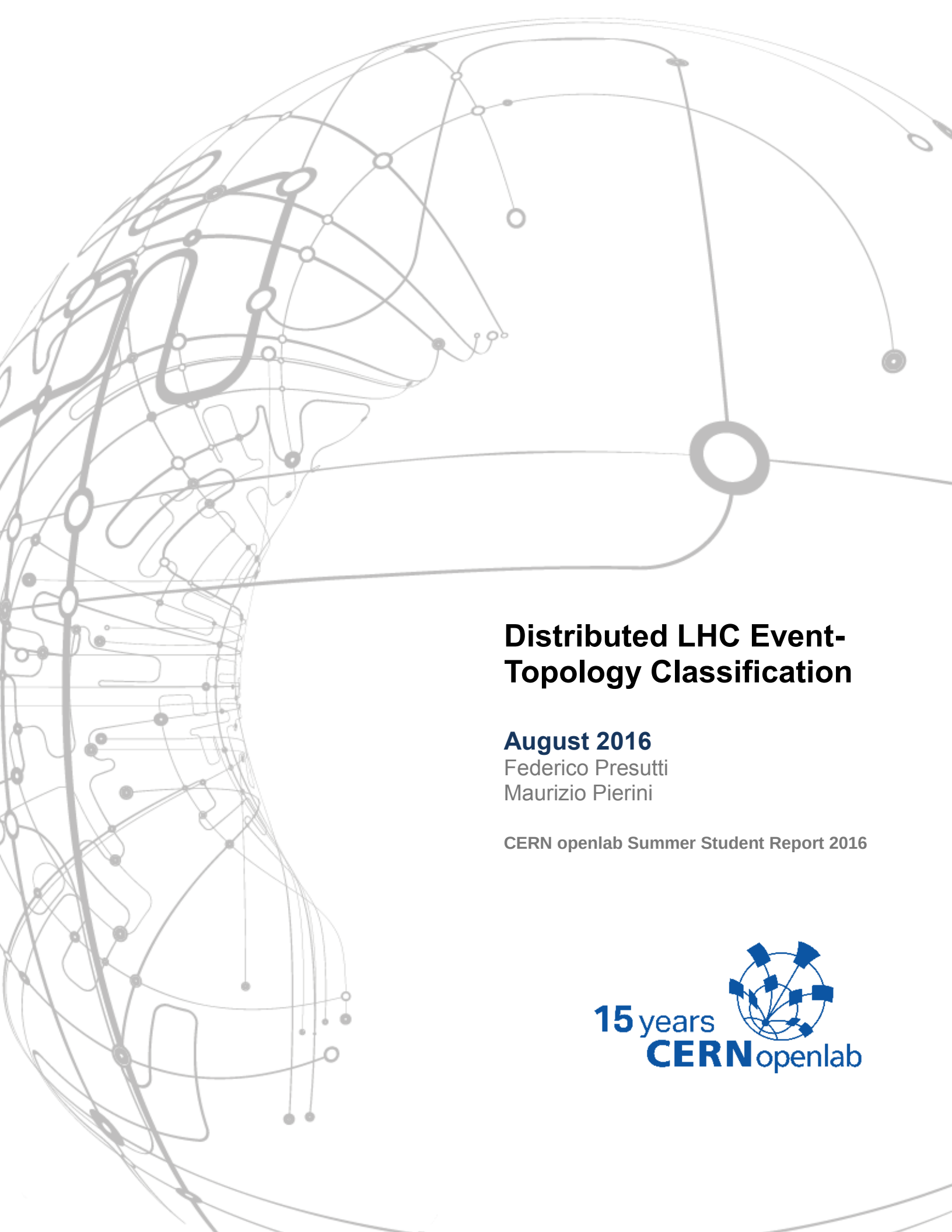


Distributed LHC Event-Topology Classification

August 2016

Federico Presutti
Maurizio Pierini

CERN openlab Summer Student Report 2016

15 years CERN openlab

Abstract

High data volumes and data throughput are a central feature of the CMS detector experiment in the search for new physics. The aim of this project is to develop prototype systems capable of speeding up and improving the quasi-real-time analyses performed by the triggers during the data-acquisition stage of the experiment. This is of importance as the high luminosity upgrade of the LHC is expected to increase the raw data throughput significantly. The options explored to improve the trigger farm performance are the use of GPUs for parallelization of razor variable analysis, and inference based on distributed machine learning algorithms.

Table of Contents

1 Parallelization of Jet Kinematic Variables for the High Level Trigger

1.1 Introduction

QCD particle jets are commonplace in collision events found in the CMS experiment. They are nonetheless of interest since they may contain clues regarding the search for supersymmetric particles and "new physics". As such, analysis of the jets are used in some of the HLT algorithms, when the parameters are above a certain range that are deemed interesting. Since high energy particle physics is stochastic in nature, the number of jets in a given event can vary greatly. An event may have anywhere between 20 and no jets. Analysis is performed working under the assumption that jets from a single event have the same origin.

Razor variable analysis is a technique developed to aid in the search for model-independent SUSY events using jets. It essentially looks for a combination of missing energy and transverse momentum. Razor variable analysis mainly consists of arithmetic manipulation of experimental parameters and is thus fairly computationally simple given a list of 4-momenta of jets, except for one prior computationally intensive step. This step consists of partitioning the jets into two hemispheres such that the sum of the square of the masses of the jet is minimized.

Since partitioning is an NP-complete problem, this is an exponential search. Therefore, the algorithms that analyse the jets tend to be relatively slow. This is a concern since the HLT has to operate in quasi-real time, sifting through all the event data created at the experiment that has passed through the previous triggers.

Since the razor variable computations are relatively straightforward, an idea was to parallelize the variable computation for all possible partitions for one or more events simultaneously. This problem is well-suited for GPU computing, which can perform the same set of instructions, although not quite as quickly as on a CPU, on numerous cores at once.

The idea would thus be to systematically compute all possible partitions and subsequently compute the razor variables and mass terms for all partitions. At the end of this process one could search through the results, find the optimal partition given the mass term, and output the corresponding razor variable.

1.2 Implementation

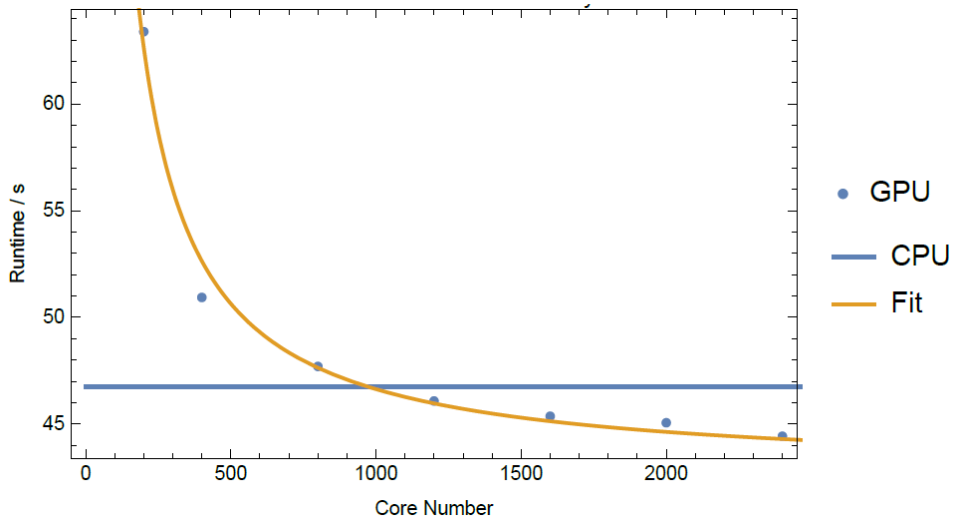
This idea was implemented and benchmarked by using the CUDA programming platform, on a single machine with 8 CPU cores and with NVIDIA TitanX GPUs, 3000 cores and at the time of writing one of the best GPU models available. A CMS dataset of approximately 62,000 events was used for the benchmark.

A high level overview of the algorithm is as follows, using a custom 4-momentum class implementation (since the ROOT classes are not implemented with GPUs in mind).

- The dataset is loaded in memory: an array of jet 4-momenta vectors are initialized.

- The program then iterates through the events, checking the number of jets in each one, and counting the number of additional operations the GPU needs to perform in parallel, and the associated memory cost.
- When either the number of parallel processes or the required memory passes a given threshold, there is a memory transfer of relevant data from main memory to GPU memory, and GPU computation is initiated.
- Information regarding the partitioning is stored efficiently as a bit-vector or integer. A 1 would indicate inclusion of a jet, and a 0 exclusion. Thus an array of 4-momenta for all events and an array of these bit-vectors are all that have to be stored on GPU memory.
- Razor variable analysis is performed and stored in an output array. This is then sent back to main memory. Since there is no way to parallelize a search through differently-size blocks (due to the variable number of jets per event), searching for the optimal partition is done on the CPU.

The results unfortunately do not favour GPUs strongly enough to warrant a change of hardware. The figure below demonstrates the results.



Runtime for razor variable analysis

Note that the slowest step in each trial was the loading of data into memory, so the absolute runtime is meaningless. Since this step is equally slow for all trials.

Because the number of GPU instructions in this analysis are constant, the main slowdown is attributed to the number of times the GPU was called. Note also that GPU memory transfer speeds are linear with respect to the size with a constant overhead. Hence the runtime should scale approximately as 1/N, with a constant factor, which is visible in the plot above.

The constant term in the fit performed resulted in a speedup of 4 seconds with respect to the CPU runtime. Spread across 62,000 events, this is a rather trivial speedup. This is aggravated by the fact that in a real implementation, if the algorithm were to wait for real-time events in order to fill up its GPU cores, the advantage would probably be lost. For offline analysis, the speed advantage is offset by the difficulty of GPU programming.

There are a few possible reasons for why the GPU is not achieving significant improvements. The first is that large data transfers are relatively slow, and must be done synchronously, because the analysis cannot begin without all the data present in GPU memory. Secondly, parallelization causes redundancies and inefficiencies. Avoiding divergence demands that each core must have its own copy of the set of jet 4-momenta corresponding to its event. Additionally, it does not allow for the last step, the optimal-partition search, to be done on the GPU, which causes further data transfer slowdowns and a large process to be performed serially, defeating the purpose of parallelization.

2 Distributed Event-Topology Classification

2.1 Introduction

The CMS detector and its algorithms currently allow for a high resolution reconstruction of the multitude of particles created in collision events. We know that different types of physical interactions will result in the creation of different sets of particles. Nonetheless, since interactions in high energy physics are a stochastic process, the particles created and picked up by the detector will vary immensely. However, a machine learning algorithm might be able to learn to distinguish between these different collections of particles.

A machine learning algorithm trained to recognize the types of physical processes producing the events it is fed would have useful applications in triggers, since it could be used to classify events on the fly. Furthermore, it could be used with other applications such as anomaly detection. If an event is not recognized as a process the algorithm was trained on, it could indicate either a malfunction or an unusual physics event of interest.

The difficulty in the problem lies in the sheer number of particles created at the high energies of the LHC. The size of the input to the algorithm depends on the final design choice, however, without any truncation, the input size could consist of data for hundreds of thousands of particles. Training a deep neural network of this size to a high degree of accuracy takes considerable computation power. Thus it would be invaluable to develop a system for distributed learning, which refers to machine learning algorithms that can harness the power of multi-node computing. The most important reason to use distributed learning is that it would allow for training on supercomputers in parallel. Since the community is finding progressively more applications machine learning in high energy physics, having such a setup would make training and implementing future ideas easier.

There are a number of stages to realize such a prototype:

- To show that such a neural network can be successful in classifying events based on simulated data.
- To build a system that performs distributed training with a cluster.
- Quantitatively demonstrate that machines equipped with GPUs are a more effective solution than those without GPUs; i.e. show that the cost to performance ratio using GPUs for training is superior to using cluster using only CPUs, in order to justify the cost GPUs.

2.2 Implementation

The input of such a deep neural network would be a list of particles features: particle ID, 4-momenta, impact parameter. Training data was generated using the DELPHES fast-simulation framework for CMS detector. The training data was prepared was in the order of a hundred gigabytes.

A 98% accuracy was obtained using training with Python's Keras on a single machine with a single GPU using a relatively simple DNN model. These results are promising, but training in this manner is relatively slow and it is not possible to optimize the DNN's hyper-parameters.

For distributed learning, we wanted to continue using Keras, but would have to implement the distribution ourselves. There have been efforts in the open source community to make such tools. The most attractive option for distributing jobs in a cluster is Apache Spark. There is an open source package which seeks to combine Keras and Spark, which is called Elephas, but is still rather limited scope.

Collaborating with CERN IT, which provided a Hadoop cluster, we sought to implement and expand on this package. The main limitation to using Spark is that it is made to work solely in memory and works with only select data formats. This is not an acceptable size for the scope of this project. Hundreds of gigabytes of data are needed in order to sufficiently train a network, and these are all stored as compressed, binary data files. A workaround is in progress to feed the records but is not yet ready.