Evaluation of likelihood functions for data analysis on Graphics Processing Units

Sverre Jarp, Alfio Lazzaro, Julien Leduc, Andrzej Nowak and Felice Pantaleo
CERN openlab
European Organization for Nuclear Research, CERN
Geneva, Switzerland
Emails: {sverre.jarp, alfio.lazzaro, julien.leduc, andrzej.nowak, felice.pantaleo}@cern.ch

***Abstract*—Data analysis techniques based on likelihood function calculation play a crucial role in many High Energy Physics measurements. Depending on the complexity of the models used in the analyses, with several free parameters, many independent variables, large data samples, and complex functions, the calculation of the likelihood functions can require a long CPU execution time. In the past, the continuous gain in performance for each single CPU core kept pace with the increase on the complexity of the analyses, maintaining reasonable the execution time of the sequential software applications. Nowadays, the performance for single cores is not increasing as in the past, while the complexity of the analyses has grown significantly in the Large Hadron Collider era. In this context a breakthrough is represented by the increase of the number of computational cores per computational node. This allows to speed up the execution of the applications, redesigning them with parallelization paradigms. The likelihood function evaluation can be parallelized using data and task parallelism, which are suitable for CPUs and GPUs (Graphics Processing Units), respectively. In this paper we show how the likelihood function evaluation has been parallelized on GPUs. We describe the implemented algorithm and we give some performance results when running typical models used in High Energy Physics measurements. In our implementation we achieve a good scaling with respect to the number of events of the data samples.**

***Keywords*-Likelihood; analysis; parallelization; GPUs**

I. Introduction

With the start-up of the Large Hadron Collider at CERN, the High Energy Physics (HEP) community has the great opportunity to look for possible effects predicted by several physics models, such as Higgs boson or SUSY particles, or totally unpredicted effects [1]. The real challenge is to extract such new phenomena from the data collected by the experiments, since they can be very rare and their contribution small if compared to the total amount of data. Data are a collection of independent *events*, an event being the measurement of a set of *variables* (energies, masses, spatial and angular variables...) recorded in a brief span of time by the physics detectors. The events can be classified in different *species*, which are generally denoted with *signals*, for the events of interest for their physics phenomena, and *backgrounds*, all that remains.

Over the last few years many complex techniques have been used to discriminate signal and background events, like maximum likelihood fits, neural networks, and boosted decision trees [2], [3]. The discrimination is obtained using particular variables (*discriminant variables*), or more in general combination of these variables, which have different characteristics for signal and background events. These techniques are implemented in several software applications, developed inside the HEP community, mainly as sequential algorithms. Increasing the number of events and variables in the samples and using advanced algorithms for discrimination require high CPU performance for the execution of such programs. In the past, the continuous gain in performance for each single CPU core kept pace with the increase on the complexity of the analyses. Therefore the execution time of the applications remained reasonable, without requiring parallel implementations of the algorithms. Nowadays, the performance of a single core is not increasing as in the past, while the complexity of the analyses has grown significantly. However, currently we have more computational units available in each single computational node, which can cooperate to execute the same application. CPU vendors like Intel and AMD are developing multi-cores. Currently we have up to 12 cores implemented in a single socket. Furthermore, Graphics Processing Units (GPUs) are also emerging as systems particularly suitable for intensive floating point algorithms which can benefit from their massive parallelization design. Therefore, it is clear that the existing HEP software applications for data analysis have to be redesigned to become parallel in order to take full advantage of the new computational systems. This is mandatory to speed up the execution time of these applications.

In this work we describe a strategy for the parallelization of a data analysis software based on the calculation of likelihood functions, e.g., in maximum likelihood fits. The common software used in HEP for likelihood-based analyses is RooFit [4], which is part of the general data analysis framework ROOT [5]. These analyses allow the estimation of unknown parameters using a given sample of data. An example of such parameters is given by the number of events belonging to each species and the parameters which characterize the probability density functions (PDFs) of the input variables, that can be related to the prediction obtained from physics models. Currently RooFit implements an algorithm for the evaluation of the likelihood function

which cannot take fully advantage from the vectorization and other code optimizations (like function inlining) due to its implementation based on C++ virtual methods [6]. To overcome these limitations, we have designed and implemented a new algorithm on CPUs, and parallelized it using a data parallelism paradigm implemented with OpenMP. In this paper we focus on the implementation of this algorithm on GPUs. Using the new algorithm, we are able to exploit data and task parallelism paradigms on GPUs, so that we can achieve a more fine-grain parallelism. For the comparison of the performance between CPU and GPU implementations, we take as reference the optimized parallel OpenMP implementation, which guarantees a *fair* comparison between CPU and GPU implementations.

Existing works in the literature report on the parallelization implemented on GPUs of the evaluation of likelihood functions in some specific scientific fields, such as phylogenetic analysis [7] and medical image reconstruction [8]. To the best of our knowledge, however, no previous work describes the GPU implementation of the evaluation of likelihood functions for general use in the case of maximum likelihood fit analyses. The goal of our work is to provide a general infrastructure for parallelization inside the RooFit packages so that users can use it for their specific data analysis.

This paper is organized as follows. In Section II we provide an overview of the maximum likelihood data analysis technique, a description of the RooFit package, and a detailed description of the implemented algorithms used for the likelihood function evaluation (the current available RooFit algorithm and our optimized algorithm). In Section III we give details on the parallelization of the new algorithm and in Section IV we describe the implementation of the parallelized algorithm on the GPU. Section V reports the results from tests with a benchmark analysis on a nVidia GeForce GTX 470 GPU. Conclusion are reported in Section VI.

II. Likelihood Function Analysis

We consider a multidimensional random vector $\hat{x} = (x^1, \ldots, x^n)$ described by a distribution function $\mathcal{P}(\hat{x}|\hat{\theta})$, where $\hat{\theta}$ is a set of p real parameters. We assume $\mathcal{P}(\hat{x}|\hat{\theta})$ to be well known except for the set of parameters $\hat{\theta}$. So, the $\mathcal{P}(\hat{x}|\hat{\theta})$ expression represents, after normalizing it, the hypothesized PDF for the $\hat{x}$ variables. If we then perform an experiment where a measurement has been repeated N times, supplying $\hat{\mathbf{x}} = \hat{x}_1, \ldots, \hat{x}_N$ values. The joint PDF of $\hat{\mathbf{x}}$ is, by independence,

$$f(\hat{x}|\hat{\theta}) = \prod_{i=1}^{N} \mathcal{P}(\hat{x}_i|\hat{\theta}). \quad (1)$$

Note that, when the variables are uncorrelated, the PDF can be factorized as product of single PDFs dependent on each variable, such as:

$$\mathcal{P}(\hat{x}_i|\hat{\theta}) = \prod_{v=1}^{n} \mathcal{P}^v(x_i^v|\hat{\theta}). \quad (2)$$

When the variables $\hat{x}$ are replaced by the observed data $\hat{\mathbf{x}}$, then f is no longer a PDF, and it is usual to denote it by $\mathcal{L}$, being the *likelihood function*, which is now a function of $\hat{\theta}$ only: $\mathcal{L}(\hat{\theta}) = f(\hat{\mathbf{x}}|\hat{\theta})$.

An important case where the likelihood function is used is the estimation of the parameters, using the *maximum likelihood (ML) method*. The estimate of $\hat{\theta}$ is the set of values of parameters for which $\mathcal{L}(\hat{\theta})$ has its maximum, given the particular data sample of measurements $\hat{\mathbf{x}}$. The procedure to find the maximum represents the fitting procedure. We can use the ML method to estimate the number of events in a data sample belonging to the different species, i.e. signals or backgrounds. Considering s different species, and defining with n_j the number of events belonging to species j and with $\mathcal{P}_j(\hat{x}|\hat{\theta}_j)$ the PDF for the species j, the likelihood function becomes:

$$\mathcal{L} = \frac{e^{-\sum_{j=1}^{s} n_j}}{N!} \prod_{i=1}^{N} \sum_{j=1}^{s} n_j \mathcal{P}_j(\hat{x}_i|\hat{\theta}_j). \quad (3)$$

In this expression we have introduced the *extended* term to take into account that the number of observations N in the sample is itself a Poisson random variable with a mean value $\sum_{j=1}^{s} n_j$. So this function is called the *extended likelihood function*. It has to be maximized as a function of the free parameters n_j and the possible free parameters $\hat{\theta}_j$ of the PDFs [2].

The likelihood function applies also for interval estimation of the parameters. In this case we refer to *likelihood based confidence intervals*, and the multidimensional extension called *profile likelihood*. Unlike the maximum likelihood method, where we estimate the values of $\hat{\theta}$, in the interval estimation we want to find the region $\hat{\theta}_a \leq \hat{\theta} \leq \hat{\theta}_b$, which contains the true value $\hat{\theta}_0$ with probability β. Such a region is called *confidence region* for $\hat{\theta}$ with probability content β. The confidence region is given by

$$\ln \mathcal{L}(\hat{\theta}) = \ln \mathcal{L}_{\max} - \frac{1}{2}\chi^2_\beta, \quad (4)$$

where $\mathcal{L}_{\max}$ corresponds to the likelihood function value at its maximum as function of $\hat{\theta}$. The borders of the region are the values of set of parameters $\hat{\theta}$ which satisfies this equation. So for one standard deviation error we have $\ln \mathcal{L}(\hat{\theta}) = \ln \mathcal{L}_{\max} - 1/2$ [2].

A. Numerical Considerations

The search of a maximum for $\mathcal{L}$ and the confidence region estimation as a function of the unknown parameters can be carried out numerically. Usually, it is used to minimize the equivalent function $-\ln(\mathcal{L})$, the *negative log-likelihood* (NLL), which has a direct connection with the confidence

region estimation (4), (we omit the $N!$ term in the expression, which does not depend on the parameters). So the NLL to be minimized has the form:

$$NLL = \sum_{j=1}^{s} n_j - \sum_{i=1}^{N} \left(\ln \sum_{j=1}^{s} n_j \mathcal{P}_j(\hat{x}_i|\hat{\theta}_j) \right), \qquad (5)$$

that is, a sum of logarithms.

The most common method used in the HEP community to find the minimum of NLL is based on the MIGRAD algorithm inside the MINUIT package [9]. MIGRAD performs the minimization of a function using the *variable metric* method [10]. This method involves the calculation of the derivatives of the *NLL* for each free parameter. Since very often we are faced with minimizing a function for which no derivatives are provided, MIGRAD is able to estimate the derivatives of the function by finite differences. Details of the implementation of the method used in MIGRAD can be found elsewhere [9].

MINUIT uses the special algorithm, MINOS, for the profile likelihood determination. In this case, MINOS allows to determine the values of the parameters which satisfy (4), requiring to follow the log-likelihood all the way out to the edge of the confidence region in any number of dimensions. This is done applying several times MIGRAD for each parameter of interest [9].

These algorithms require several calls to the *NLL* as described in (5), which requires itself the calculation of the corresponding PDFs for each variable and each event of the data sample. Hence, depending on the complexity of the NLL function, the procedures can be very time-consuming.

B. NLL *Evaluation*

RooFit package is formed by a set of classes constructed on ROOT framework dedicated to likelihood-based analyses. This software has been developed with an object oriented coding technique, using the C++ language. We should underline that all floating point operations are performed in double precision.

All classes for PDFs inherit from a common abstract class (class `RooAbsPdf`), which provides the common interface. It includes the virtual methods:

- `evaluate`: it provides the non-normalized value (for internal use to the class, i.e. protected method to the class);
- `getVal`: it provides the normalized value, so essentially the value obtained from `evaluate` divided by PDF normalization integral;
- `getLogVal`: it provides the logarithm of the normalized value obtained using `getVal`.

By design, outside the class we can access only the normalized value obtained using `getVal` or its logarithm using `getLogVal`. Combinations of PDFs are possible with classes for adding, multiplying and convoluting basic PDFs.

Finally, RooFit provides a class for the *NLL* calculation (class `RooNLLVar`), which basically does the reading of the data (by means of the class `RooAbsData`), and the loop over the events to calculate the *NLL* sum term with the corresponding calls to `getLogVal` of the total PDF.

Data are organized in memory like a matrix where the columns contain the values for each variable, and the rows represent the values of the variables belonging to each event (see Fig. 1). This structure has the advantage of keeping the values of the variables belonging to the same event in adjacent memory location. In order to calculate the *NLL* from the formula (5), the current available RooFit algorithm consists of the following steps (in order):

1) For a given set of values of parameters of the model, loop over the events $i = 1...N$:
   - read the values of the variables for event i, i.e. read the row i of the data matrix;
   - calculate the PDFs $\mathcal{P}$'s for the event i (this step requires the calculation and caching of the normalization integrals of the PDFs);
   - combine, by means of addition and multiplication, the results of the individual PDFs to calculate the total PDF value for the event i;
   - calculate the logarithm of the total PDF value;
   - calculate the partial term of the sum in the *NLL* and accumulate this value to calculate the final value of the sum.
2) Finalize the calculation of *NLL* by including the extended term contribution.
3) Iterate the calculation of *NLL* for a choice of the values of the model parameters.

The key part of this procedure is the calculation of *all* PDFs for *each* event, and then there is a single loop over all events (performed inside the class `RooNLLVar`). Since this is done by having recourse to calls of the virtual methods `evaluate` and `getVal` of each PDF, this algorithm does not allow particular code optimization, like inlining and vectorization, and it introduces the obvious overhead due to the virtual method calls (there are two calls to virtual methods per each PDF and per each event, plus a call to the virtual method `getLogVal` per each event).

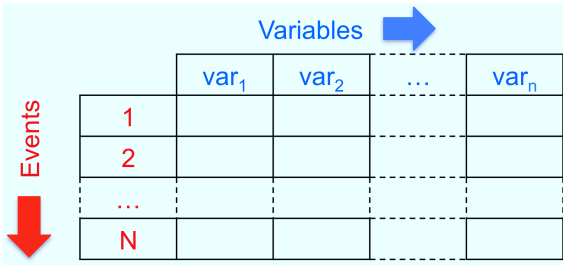

Figure 1. Data representation as matrix, where the variables are organized in columns and events in rows.

In order to take benefit from code optimization, we redesigned the algorithm to reduce the number of calls to virtual methods. Furthermore, the data are stored differently: the values of each variable are organized in independent arrays, so that we can profit from the coalescing of memory accesses for each variable. This is particularly useful in the CPU cache memories data access when calculating the values for an individual PDF. The new algorithm follows a different procedure for the first step with respect to the RooFit algorithm described above:

- For a given set of values of the parameters and a given PDF, we evaluate the PDF on each event of the data sample (which means calculate the PDF on the corresponding arrays of variables), and we save the results of this calculation in an array. So we do a loop over all events $i = 1...N$ and calculate the PDF for each of them.
- Repeat the previous step for all PDFs, so we end up with several arrays of partial results (an array for each PDF). Each array of results is composed by N elements, i.e. a result for each event.
- Combine, by means of addition and multiplication, all arrays of partial results, corresponding to each event, providing a final array of results, i.e. the array of results of the total PDF.
- Calculate the logarithm of the total PDF results.
- Do the sum of the total PDF results.

The key part of this procedure is the calculation of *each* PDF for *all* events, so that instead of one single *global* loop over the events, now we have independent *local* loops for each PDF (and their combinations). To implement this new algorithm, we implement a new virtual method `evaluate` for each PDF class with a reference to the data sample as parameter. Inside this method we perform the local loop over all values of the variables of the corresponding PDF, storing the results of the calculations in an array of partial results. Since the virtual method `evaluate` is called just once, and then within local loops we perform the calculations of the mathematical functions for all events, we can conclude that the number of calls to virtual methods per PDF does not depend by the number of events. Actually, the mathematical function itself is declared inside an inline method (`evaluateLocal`), which is made private to the PDF class. Furthermore, thanks to the new data structure organized as arrays for each variable, this code can easily be vectorized. An example of implementation for a PDF is shown in Fig. 2. As consequence of the new implementation of the method `evaluate`, we implement new virtual methods `getVal` and `getLogVal` in the class `RooAbsPdf` which do the loops over the partial results to apply the normalization and calculate the logarithm, respectively (see Fig. 3). The loop over the final results of the total PDF to calculate the *NLL* sum term and the extended term is done in the usual class `RooNLLVar`. Finally, we modify the classes `RooAbsData` and `RooAbsPdf` so that they can manage the arrays of data and the arrays of results, respectively, using C arrays. Indeed, we should note that a drawback of this new algorithm is that we have to manage all the arrays for the temporary results.

Optimizing the implementation of the new algorithm, we are able to reach a speed-up of 4.5x in a common data analysis with respect to the original RooFit implementation [11]. In the next section we will briefly show how the new algorithm can be parallelized using OpenMP for the CPUs, and then we will focus on its implementation for the GPUs.

```
// Inline method for the Gaussian PDF calculation,
// defined inside the class RooGaussian
inline double evaluateLocal(const double x,
                const double mu,
                const double sigma) const
{
  return std::exp(-0.5*std::pow((x-mu)/sigma,2));
}

// Virtual method for the calculation of the
// Gaussian PDF on a single event
// (this is the current RooFit implementation)
virtual double evaluate() const
{
  return evaluateLocal(m_x->getVal(),
               m_mu->getVal(),
               m_sigma->getVal());
}


// Virtual method for the calculation of the
// Gaussian PDF on all events
// (new implemented algorithm)
virtual bool evaluate(const RooAbsData& data)
{
  // retrive the data array of values for the variable
  const double *dataArray = data.GetDataArray(*m_x);
  // check if there is an array for the variable
  if (dataArray==0)
    return false;

  // retrive the number of events
  int nEvents = data.GetEntries();
  // retrive the array for the partial results
  double *resultsArray = GetResultsArray();

  // loop over the events to calculate the Gaussian
  for (int idx = 0; idx<nEvents; ++idx) {
    resultsArray[idx] = evaluateLocal(dataArray[idx],
                    m_mu->getVal(),m_sigma->getVal());
  }

  return true;
}
```

Figure 2. Representative parts of the methods for the evaluation of the Gaussian PDF, implemented in the RooGaussian class in RooFit. We show the original RooFit implementation and our new implementation. Similar implementation is used for all basic PDFs.

III. Parallelization Strategy

The parallelization of the implementation of the new algorithm is relatively straight-forward, since the iterations

```
// Original RooFit implementation
double RooAbsPDf::getVal()
{
  // Apply the normalization
  return evaluate()/GetIntegral();
}

// Original RooFit implementation
double RooAbsPdf::getLogVal()
{
  return std::log(getVal());
}


// New implemented algorithm
double* RooAbsPdf::getVal(const RooAbsData& data)
{

  // Call the evaluate method, defined in the PDF
  if (!evaluate(data))
    return 0;

  // Retrive the normalization integral
  double integral = GetIntegral();
  // retrive the number of events
  int nEvents = data.GetEntries();
  // retrive the array for the partial results
  double *resultsArray = GetResultsArray();

  // Apply the normalization
  if (integral!=1.)
    for (int idx = 0; idx<nEvents; ++idx) {
      resultsArray[idx] /= integral;
    }

  return resultsArray;

}

// New implemented algorithm
double* RooAbsPdf::getLogVal(const RooAbsData& data)
{

  // Do the calculation of the normalized PDF
  if (0==getVal(data))
    return 0;

  // retrive the number of events
  int nEvents = data.GetEntries();
  // retrive the array for the partial results
  double *resultsArray = GetResultsArray();

  // Do the Log of the results
  for (int idx = 0; idx<nEvents; ++idx) {
    resultsArray[idx] = std::log(resultsArray[idx]);
  }

  return resultsArray;

}
```

Figure 3. Representative parts of the methods for the evaluation of a PDF, implemented in the RooAbsPdf class in RooFit. We show the original RooFit implementation and our new implementation.

in the loops inside the methods `evaluate`, `getVal`, and `getLogVal` are independent. Therefore, we can parallelize them on CPUs via the `#pragma omp parallel for` OpenMP directive. Arrays of data and results are shared among the threads, so that there is a negligible increment in the global memory footprint of the application. The application scales with the number of threads as expected by the Amdahl's law, being limited by the not parallelizable part (mainly the calculation of the normalization integrals and the handling of the data) [11]. Because of the strategy adopted for the parallelization, we denote it as event-level strategy.

To efficiently exploit the GPU architecture, the number of threads must be maximized. For this reason we have extended the strategy of the parallelization to also include the evaluation in parallel of several PDFs. This solution adds a further degree of parallelism, since the PDFs can be calculated independently. More explicitly, for a given PDF, we evaluate it in parallel over the events of the data sample, as explained in the event-level strategy. Then in parallel we execute this procedure for the several PDFs. After that we combine properly all partial results of the arrays, corresponding to each event, providing a final array of results.

To explain better the strategy, we show here how it works for a simple example. We consider two variables, x^1 and x^2, and a single species, with the following total PDF:

$$\mathcal{P} = \mathcal{P}^1(x^1) \times \left[\mathcal{P}^2(x^2) + \mathcal{P}^3(x^2)\right], \tag{6}$$

that is a sum of two PDFs and the product with another PDF (so, basically, we have 5 PDFs in total). We have to evaluate this expression on the data sample composed by N events to calculate the *NLL* function. The execution steps are:

1) start the parallel calculation for $\mathcal{P}^1$, $\mathcal{P}^2$, $\mathcal{P}^3$;
2) for each PDF, make parallel calculation of the results on the data;
3) synchronize the results calculation for $\mathcal{P}^2$ and $\mathcal{P}^3$, producing two arrays of results of N elements, $\hat{V}^2$ and $\hat{V}^3$, respectively;
4) combine the results in $\hat{V}^2$ and $\hat{V}^3$ in a single array of results $\hat{V}^S$, corresponding to the sum $\mathcal{P}^S = \mathcal{P}^2 + \mathcal{P}^3$;
5) synchronize the results calculation of $\mathcal{P}^1$ and $\mathcal{P}^S$ in the two corresponding arrays, $\hat{V}^1$ and $\hat{V}^S$, respectively;
6) combine the results in $\hat{V}^1$ and $\hat{V}^S$ in a single array of results $\hat{V}$, corresponding to the product $\mathcal{P} = \mathcal{P}^1 \times \mathcal{P}^S$;
7) do the logarithms of the results in $\hat{V}$ and then calculate the final sum for the *NLL*.

We can define this strategy as based on PDF-event parallelism.

The advantage of the PDF-event parallel strategy is the possibility to have more fine-grain parallelism with respect to the event-level strategy described above. We separate the execution in *tasks*. For a given event, there are tasks carrying out the evaluation and normalization of a given PDF, the combination of the results, and the calculation of the logarithm of the final results. The algorithm requires intermediate synchronizations between the tasks, which do not introduce significant overhead (in any case we do

synchronize all tasks at the end of the evaluation for the final *NLL* calculation). Tasks belonging to PDFs which are part of composite PDFs are grouped is common *streams*, so that can be synchronized independently. Using this technique, the strategy for the parallelism is very suitable for massive parallel computation systems based on task parallelism, i.e. GPUs.

Returning to the previous example, we have in that case $N \times 9$ tasks to execute, which are divided in: $N \times 6$ tasks to calculate the results of the 3 PDFs ($N \times 3$ for the evaluation of the functions and $N \times 3$ for their normalization), N tasks for the sum PDF, N tasks for the product PDF, N tasks for the logarithm calculations.

IV. GPU Code Implementation

In this section we describe the GPU implementation of the PDF-event parallel algorithm inside the RooFit package. We added the new implementation together with the CPU event-level parallel algorithm implementation, so that data analysts can indifferently choose which algorithm to use for their data analyses.

For the GPU implementation we use C for CUDA language provided by nVidia. The software application was implemented in a Linux environment using CUDA toolkit v3.2. These rules were followed during the development of the code:

- All data in the calculation are in double precision floating point numbers.
- Same source code must compile using host compilers (e.g. Intel C++ Compiler) and using nVidia C Compiler (i.e. `nvcc`). This implies that device-related code must be enclosed inside `#ifdef ... #endif` statements, to protect it when compiling the application using host compilers. In this case only the event-level parallel algorithm will be available on the host.
- Data analysts can choose the host or device algorithm by using a flag at runtime. They do not need to change their applications, i.e. there is a common interface for the user to the two implementations.

The first modification of the code is made inside `RooAbsData` and `RooAbsPdf` classes so that they can manage the arrays of input data sample and the arrays of partial results on the device, using C arrays. The former class takes care of copying the host data to the device global memory, and vice versa for the latter, using synchronous functions. The dimension of the data sample is given by the number of variables times the number of events, where each variable is of `double` type. The arrays of input data are read-only during the entire execution of the application, so we can copy them once to the device memory at the beginning and then use them for all *NLL* calculations. The arrays of partial results can be kept resident in the device memory, except for the array of the final results which has to be copied to the host memory for the final sum of the *NLL*. Of course all temporary arrays can be cached in the device memory, avoiding to calculate them again in case the PDF does not change in consecutive calls. With this implementation we are able to strongly reduce the time spent for the communication between host and device memories.

We implemented a CUDA kernel for each method `evaluate`, which takes care of the tasks for the evaluation of the corresponding PDF function. The call to this kernel is done by the method `evaluate` itself. Inside the same method we also implement the event-level parallel algorithm with OpenMP, with a flag used for choosing which algorithm to execute. Each task gets evaluated by a CUDA thread, i.e. there is a correspondence one to one between tasks and threads. The fact that a task represents the calculation on a single event, and that the variables of the events and the partial results are organized in arrays, allows the CUDA compiler to coalesce memory accesses because the threads access to adjacent memory locations for the variables and the partial results. The kernel is defined as `friend` of the PDF class, so that it can access to the private method `evaluateLocal`. Therefore this method is compiled for the host and device. Following the example of Fig. 2, we show the corresponding implementation of the method `evaluate` and his CUDA kernel in Figs. 4 and 5, respectively. A similar implementation is repeated for the methods `getVal` and `getLogVal` of the class `RooAbsPdf`, where we add calls to the CUDA kernels used for the execution of the normalization and calculation of logarithm tasks, respectively.

Intermediate synchronizations of the results in case of composite PDFs are done by grouping in CUDA streams the corresponding threads belonging to the PDFs of the composition. Creation/destruction of these streams and synchronizations are performed inside the `evaluate` method of the composite PDFs. For a given composite PDF, the stream is propagated to the corresponding PDFs of the composition as parameter of their public method `getVal` and, from here, to their protected method `evaluate`. The synchronization of all threads is done inside the method `getLogVal`, which also requires the copy of the final results from the device to the host memory, returning the pointer to the host array of the final results. After that, the class `RooNLLVar` performs the loop on these results for calculating the sum term of the *NLL*. Finally, the data analysts can specify which implementation to use for the *NLL* evaluation when they instantiate the `RooNLLVar` object. This choice is then propagated all over the evaluation.

The number of threads per block of the CUDA kernels depends on the maximum shared-memory size, number of registers per thread, and number of threads required to use full thread warps of the CUDA architecture. All these factors are directly connected to the complexity of the *NLL* evaluation, i.e. which PDFs are involved in the calculation and the dimension of the input data sample. Therefore this number

depends on the user analyses. From our tests we have found a very small improvement ($< 1\%$) on the performance when we tune this number for each kernel. So, we have decided to simplify the procedure using for all kernels a common value for the number of threads per block, independently by the tasks carried out by the kernels. This number is specified in the static method `CUDA::GetNThreadsPerBlock`. The data analysts can set this number using the corresponding method `CUDA::SetNThreadsPerBlock` (the default values is 256). The number of blocks per kernel is then calculated from the number of events divided by the number of threads per block, rounded to the greatest integer number.

V. Tests

Tests are executed on the following hardware:

- CPU: Intel i7 965 @ 3.2 GHz (4 cores)
- Memory: 3x2048MB DDR3 @ 1333MHz
- Motherboard: Intel Desktop Board DX58SO
- Graphic Card: ASUS ENGTX470

The GeForce GTX 470 is based on GF100 "Fermi" architecture (compute capability 2.0). The card features reference clock speeds of 607 MHz core clock and 837 MHz on the 1280 MB of GDDR5 memory that runs on a 320-bit memory interface.

In the tests we look at the comparison of performances obtained from the ratio of the execution time of the application when running on the CPU and GPU the respective algorithms. We take as reference for the CPU algorithm the optimized parallel OpenMP implementation, requiring four parallel threads for the application, so that we fully load the available CPU. From the hardware point of view, we are comparing two systems which can be considered commodity systems: a single GPU, whose main target is for computer gaming, versus a standard single socket desktop system with 4 cores. Both systems are not the top of their product lines. Although we are just comparing the performance in terms of the ratio between the execution time, without considering power consumption and price of the systems, we should consider that the target of our work is to provide a parallel application that data analysts can run on easy accessible system, i.e. not supercomputing facilities, such as their desktops or even laptops. Given these considerations, we judge our tests a fair comparison between the CPU and GPU implementations of the application.

The system is running 64-bit Scientific Linux CERN 5.5 (SLC5), based on Red Hat Enterprise Linux 5 (server). The default SLC5 Linux kernel (`2.6.18-194.8.1.el5`) is used for all the measurements. We use the Intel C++ compiler version `11.1` for the host compilation. nVidia video drivers version is 260.19.29 for linux-x86_64.

In the following tests we do not include the time spent for the initialization of the application (mainly reading of data sample and declaration of the data analysis model),

```
// NOTE:
// The flag __USECUDA__ allows to switch off
// the compilation of CUDA part in case
// CUDA compiler is not used

// This method is compiled for host and device
#ifdef __USECUDA__
__host__ __device__
#endif
inline double evaluateLocal(const double x,
                const double mu,
                const double sigma) const
{
  return std::exp(-0.5*std::pow((x-mu)/sigma,2));
}

// RooAbsPdf::ImpAlgo is {kOpenMP, kGPU} and it allows
// to choose which implementation has to be executed.
// The CUDA stream is used for the
// intermediate synchronization
virtual bool evaluate(const RooAbsData& data,
              RooAbsPdf::ImpAlgo impAlgo
#ifdef __USECUDA__
              , cudaStream_t& stream
#endif
              )
{
  // retrive the data array of values for the variable
  // from the host or the device
  const double *dataArray = data.GetDataArray(*m_x,
                            impAlgo);
  if (dataArray==0)
    return false;

  int nEvents = data.GetEntries();
  // retrive the array for the partial results
  // from the host or the device
  double *resultsArray = GetResultsArray(impAlgo);

#ifdef __USECUDA__
  // Run the CUDA implementation
  if (impAlgo==RooAbsPdf::kGPU) {

    // Launch the CUDA kernel as part of
    // the given CUDA stream.
    // CUDA::GetNBlocks(nEvents) and
    // CUDA::GetNThreadsPerBlock() are static methods
    // of the class CUDA used to determinate
    // the number of blocks for a given number of
    // events and threads per block.
    KernelEvaluateGaussian<<<CUDA::GetNBlocks(nEvents),
      CUDA::GetNThreadsPerBlock(),0,stream>>>
      (this,m_mu->getVal(),m_sigma->getVal(),
       dataArrays,resultsArray,nEvents);

    return true;
  }
#endif

  // Loop over the events to calculate the Gaussian
  // Use the default OpenMP algorithm
#pragma omp parallel for
  for (int idx = 0; idx<nEvents; ++idx) {
    resultsArray[idx] = evaluateLocal(dataArray[idx],
                m_mu->getVal(),m_sigma->getVal());
  }

  return true;
}
```

Figure 4. This case shows the implementation in OpenMP and CUDA in case of the methods for the evaluation of the Gaussian PDF reported in the Fig. 2. Similar implementation is used for all basic PDFs.

```
// NOTE:
// The flag __USECUDA__ allows to switch off
// the compilation of CUDA part in case
// CUDA compiler is not used

#ifdef __USECUDA__
// The Kernel is declared as friend to the
// class RooGaussian, so that it can call
// the private method evaluateLocal
__global__
void KernelEvaluateGaussian(const RooGaussian *pdf,
                const double mu, const double sigma,
                const double *data, double *results,
                const int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) {
    results[idx] = pdf->evaluateLocal(data[idx],
                      mu,sigma);
  }

}

#endif
```

Figure 5. CUDA kernel used for the evaluation of the Gaussian PDF function. Similar implementation is used for all basic PDFs.

which is in any case a small fraction (usually less than 1% in common data analyses) of the total execution time and it is in common for the CPU and GPU implementations. In case of the GPU implementation, we include in all tests the time spent for the copy of the events from host memory to the device memory and for the copy of the array of final results back to the host memory. We should also mention that calculation of the normalization integral and the final sum of the results for the *NLL* calculation are execute only by the host in sequential. Finally, we remind that we do everything in double precision floating point operations, which gives a penalty factor of 8x (2x) on the peak performance of the GPU (CPU) used in our tests with respect to single precision.

The first synthetic test consists in understanding the device algorithm performance. For that we do the evaluation of the *NLL* with a linear polynomial as PDF in the form $ax + b$, where a and b are two fixed parameters. The data sample is composed of 1,000,000 events (one variable). We consider a single species in (5). The application repeats this evaluation 1000 times, forcing the calculation for each iteration, i.e. not using cached values. The CUDA kernels execution is organized in blocks of 512 threads. We reach 22 GFLOPS (peak performance on the GeForce GTX 470 is about 136 GFLOPS). If we do not consider the time spent for the communications (mainly the communication of the final results per each iteration, since the input data sample is communicated just once at the beginning), we reach 112 GFLOPS (82% of the peak performance).

Then we run tests with different PDFs, usually used in HEP [2]. Also in this case we use a sample of 500,000 events (one variable), and we repeat the *NLL* evaluation 1000 times, forcing the calculation for each iteration. The number of threads per block is 512. We show the comparison of performance between the CPU and GPU implementations in table I. Furthermore, we report the percentage of the execution time spent by the kernels, i.e. the portion of the application which is executed on the device. The CPU implementation scales at 4 parallel threads with an efficiency around 97.5% (speed-up 3.9x). We can note a relation between the percentage of the execution time spent by the kernels execution and the complexity of the PDFs. For simpler PDFs (e.g. Breit-Wigner and polynomials) the total execution time is dominated by the data transfers and the host part of the implementation. This effect becomes less relevant for complex PDFs (e.g. Argus PDF). Since the time spent for the data transfer does not depend on the PDFs (it depends on the number of events), we can conclude that we can reach a better ratio of CPU vs GPU execution time for complex PDFs.

We can better understand the relation between the complexity of the PDFs and the ratio of executions times of CPU and GPU applications considering a test where we do an incremental combination of PDFs. The total PDF is $\mathcal{P} = \prod_v \mathcal{G}^v(x)$, where $\mathcal{G}^v$ are Gaussians with different parameters calculated on the same variable x. We do the test varying the number of Gaussians. The sample is composed by 500,000 events and we repeat the *NLL* evaluation 1000 times, forcing the calculation for each iteration. The number of threads per block is 512. The results are shown in Fig. 6. We should underline that we do the evaluation of the *NLL* for the same data sample, since all Gaussians depend on the same variable. Therefore, increasing the number of Gaussians in the product results in a corresponding increase of the complexity of the total PDF, and hence the time spent for its calculation. As we can see from the plot, the GPU implementation gives better performance with respect to the CPU implementation when increasing the number of Gaussians in the total PDF. This can be directly correlated to the percentage of time spent in the kernel calculation, also shown in the plot. With 10 Gaussian PDFs the GPU implementation runs 4x faster than the CPU implementation.

The last test we present is based on a "real" data analysis model, published in [12]. It consists in a multivariate analysis of 3 variables (denoted by x, y, z) and 4 species (a, b, c, d). The total PDF is the following:

$$\begin{aligned} & n_a DG_{1,a}(x) AG_{1,a}(y) AG_{2,a}(z) + \\ & n_b G_{1,b}(x) BW_{1,b}(y) G_{2,b}(z) + \\ & n_c AR_{1,c}(x) P^1_{1,c}(y) P^2_{2,c}(z) + \\ & n_d P^4_{1,d}(x) G_{1,d}(y) AG_{1,d}(z), \end{aligned}$$

where G is Gaussian, DG is a double Gaussians (i.e. sum of two Gaussians), AG is Asymmetric Gaussian, BW is Breit-Wigner, AR is Argus, and P^n is polynomial of n^{th} order. In total there are 13 basic PDFs and 6 composite PDFs.

Table I

RESULTS OF THE EXECUTION FOR DIFFERENT PDFS. THE FIRST TWO COLUMNS ARE THE NAME AND THE FORMULA OF EACH PDF (x IS VARIABLE, BEING THE OTHERS PARAMETERS). THIRD COLUMN REPORTS THE RATIO BETWEEN THE EXECUTION TIME OF CPU AND GPU, AND THE LAST COLUMN IS THE PERCENTAGE OF THE EXECUTION TIME SPENT BY THE KERNELS, I.E. THE PORTION OF THE APPLICATION WHICH IS EXECUTED ON THE DEVICE. CPU IMPLEMENTATION RUNS IN PARALLEL WITH 4 THREADS, WITH SPEED-UP $\sim$ 3.9X.

PDF Name	Formula	CPU vs GPU time ratio	kernels execution time portion
Gaussian	$\exp\{-\frac{(x-\mu)^2}{2\sigma^2}\}$	1.45	24.1%
Asymmetric Gaussian	$\begin{cases} \exp\{-\frac{(x-\mu)^2}{2\sigma_l^2}\} & x \leq \mu \\ \exp\{-\frac{(x-\mu)^2}{2\sigma_r^2}\} & x > \mu \end{cases}$	1.39	26.6%
Breit-Wigner	$\frac{1}{(x-x_0)^2+\Gamma^2/4}$	1.53	17.7%
Linear Polynomial	$ax + b$	1.47	16.4%
Parabolic Polynomial	$ax^2 + bx + c$	1.54	17.1%
Argus	$\sqrt{\left(1-\frac{x^2}{c^2}\right)}\exp\left\{-\frac{1}{2}\eta\left(1-\frac{x^2}{c^2}\right)\right\}$	1.80	26.3%

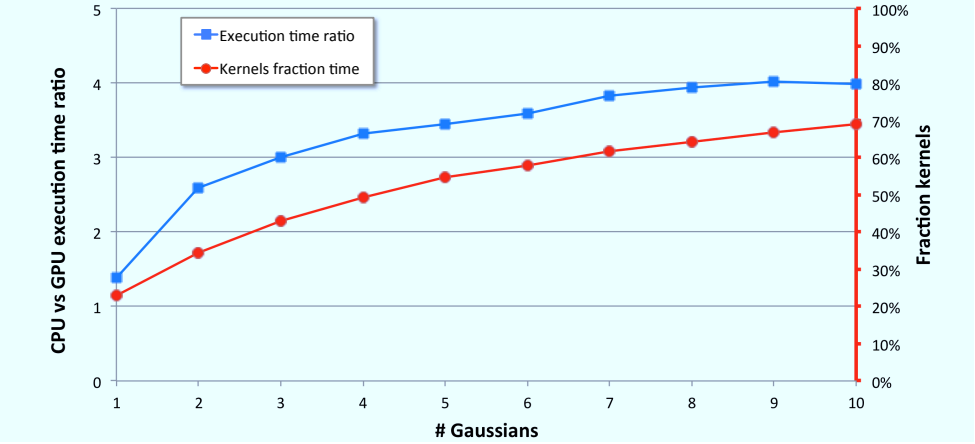

Figure 6. The blue line with square markers represents the ratio between the execution time of CPU and GPU implementations when executing the *NLL* evaluations, whose total PDF is the product of an incremental number of Gaussians (1 to 10), calculated on a common variable (500,000 events in the data sample). The reference CPU execution time is taken when running the application using 4 active parallel threads (speed-up 3.9x). The red line with circle markers, which refers to the right axis, shows the time fraction for executing the kernels of the total execution time.

We do the test varying the number of events of the data sample. We repeat the *NLL* evaluation 1000 times, forcing the calculation for each iteration. The number of threads per block is 512. The results are shown in Fig. 7. The CPU implementation scales at 4 parallel threads with an efficiency around 90% (speed-up 3.6x). We can see how the GPUs algorithm behaves better for high number of events, which is due to the specific characteristics of the GPU architectures to take advantage from multiple threads. At 10,000 events the two implementations have comparable performance, and the GPU execution time breaks down in 36% device kernels, 60% host execution, 4% host-device communications. Instead, at 500,000 events the GPU algorithm is almost 6x faster, with execution time divided in 68% for device kernels, 21% for host execution, 11% for host-device communications.

VI. CONCLUSION

In this paper we have described a different strategy for the *NLL* evaluation with parallel execution, based on a PDF-event parallelism. This strategy give a more fine-grain parallelism with respect to a conventional event-based parallel algorithm. We implement the PDF-event parallel algorithm inside the RooFit package for a GPU device. The performance of this implementation has been compared with an optimized parallel OpenMP implementation on the CPU of the event-based parallel algorithm.

We run the comparisons for different data analysis models and number of events. The hardware at our disposal is based on commodity systems, that can be considered, in terms of price and power consumption, easily accessible to general

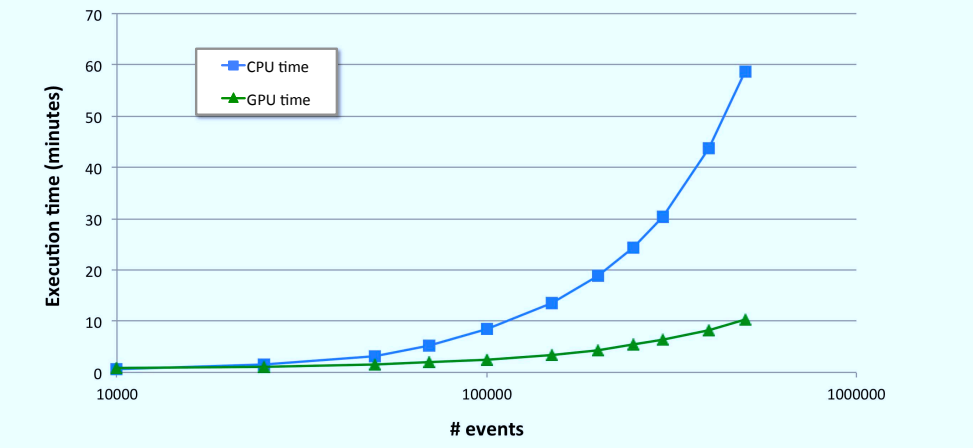

Figure 7. Execution time of the CPU (blue line with square markers) and GPU (green line with triangle markers) implementations when executing the *NLL* evaluation for a complex model as total PDF (see text for details), varying the number of events in the data sample. The CPU execution time is taken running the application with 4 parallel threads (speed-up 3.6x).

data analysts. The CPU application is executed in parallel, so that we can judge our tests a fair comparison between the CPU and GPU implementations of the application. The boost on execution time we reach with the GPU implementation with respect to the CPU implementation depends on the complexity of the data analysis model and the number of events in the data sample. In a test with a complex real model, we reach almost 6x speed-up. However, in case of simpler models and small data sample dimensions, the benefit in the performance from the GPU implementation becomes less evident. Therefore it is not possible to draw a general conclusion which can be applied in all data analyses. The analysts can easily choose which algorithm to use for their own data analysis in order to achieve the best performance. The expected increase in the near future of the data sample dimension and of the complexity of the data analyses, that will carry out at the experiments running at the Large Hadron Collider at CERN, leads to the conclusion that the GPU implementation will play an important role. For this reason we are collaborating with several groups in the community to have a wide coverage on the analyses for more specific tests, and we expect to officially release the code in the next ROOT releases.

REFERENCES

[1] G. Kane and A. Pierce, *Perspectives on LHC Physics*, 1st ed., World Scientific Publishing Company, 2008.

[2] G. Cowan, *Statistical Data Analysis*, 1st ed., Oxford University Press, 1998.

[3] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning*, 2nd ed., Springer, 2009.

[4] W. Verkerke and D. Kirkby, *The RooFit Toolkit for data modeling*, proceedings of PHYSTAT05, Imperial College Press, 2006.

[5] R. Brun and F. Rademakers, *ROOT An object oriented data analysis framework*, Nuclear Instruments and Methods in Physics Research Section A, Volume 389, Issue 1-2, p. 81, 1997.

[6] A. Lazzaro and L. Moneta, *MINUIT package parallelization and applications using the RooFit package*, J. Phys.: Conf. Ser. 219, 042044, 2010.

[7] F. Pratas *et. al.*, *Fine-grain Parallelism using Multi-core, Cell/BE, and GPU System: Accelerating the Phylogenetic Likelihood Function*, 2009 International Conference on Parallel Processing, pp. 9-17, 2009.

[8] L. Caucci *et. al.*, *Maximum Likelihood Event Estimation and List-mode Image Reconstruction on GPU Hardware*, 2009 Nuclear Science Symposium Conference, pp. 4072 - 4076, 2009.

[9] F. James, *MINUIT - Function Minimization and Error Analysis*, CERN Program Library Long Writeup D506, 1972.

[10] W. C. Davidon, *Variable Metric Method for Minimization*, SIAM J. Optim. Volume 1, Issue 1, p. 1, 1991.

[11] S. Jarp *et. al.*, *Parallelization of maximum likelihood fits with OpenMP and CUDA*, CERN-IT-2011-009, 2011, to be published on Journal of Physics: Conference Series.

[12] B. Aubert *et. al.*, *Observation of CP Violation in B^0 to $\eta' K^0$ Decays*, Phys. Rev. Lett. **98**, 031801, 2007.