Parallelization of maximum likelihood fits with OpenMP and CUDA

Sverre Jarp, Alfio Lazzaro, Julien Leduc, Andrzej Nowak and Felice Pantaleo

CERN openlab, Geneva, Switzerland

E-mail: sverre.jarp@cern.ch, alfio.lazzaro@cern.ch, julien.leduc@cern.ch, andrzej.nowak@cern.ch, felice.pantaleo@cern.ch

Abstract. Data analyses based on maximum likelihood fits are commonly used in the high energy physics community for fitting statistical models to data samples. This technique requires the numerical minimization of the negative log-likelihood function. MINUIT is the most common package used for this purpose in the high energy physics community. The main algorithm in this package, MIGRAD, searches the minimum by using the gradient information. The procedure requires several evaluations of the function, depending on the number of free parameters and their initial values. The whole procedure can be very CPU-time consuming in case of complex functions, with several free parameters, many independent variables and large data samples. Therefore, it becomes particularly important to speed-up the evaluation of the negative log-likelihood function. In this paper we present an algorithm and its implementation which benefits from data vectorization and parallelization (based on OpenMP) and which was also ported to Graphics Processing Units using CUDA.

1. Introduction

The maximum likelihood (ML) fitting procedure is a popular statistical technique used to estimate parameters of a statistical model on a given data sample [1]. Data samples are a collection of N independent *events*, an event being the measurement of a set of *variables* $\hat{x} = (x^1, \ldots, x^n)$ (energies, masses, spatial and angular variables...) recorded in a brief span of time by 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. Each variable x^j is distributed for the given species s with a probability distribution function (PDF) $\mathcal{P}_s^j(x^j; \hat{\theta}_s^j)$, where $\hat{\theta}_s^j$ are free (not constant) parameters of the PDF. If the variables are uncorrelated each other, then the total PDF for the species s is expressed by

$$\mathcal{P}_s(\hat{x}; \hat{\theta}_s) = \prod_j \mathcal{P}_s^j(x^j; \hat{\theta}_s^j). \tag{1}$$

The ML technique allows to estimate the values of the free parameters, as well the number of events belonging to each species n_s, by maximizing the function

$$\mathcal{L} = \frac{e^{-\sum_s n_s}}{N!} \prod_{i=1}^{N} \sum_s n_s \mathcal{P}_s(\hat{x}_i; \hat{\theta}_s), \tag{2}$$

which is called the *extended likelihood function*. We should underline that $\hat{x}_i$ are measured and the $\mathcal{P}_s^j$ functions are well-known, so $\mathcal{L}$ only depends on the free parameters we want to fit on the data sample.

The search for the maximum for $\mathcal{L}$ can be carried out numerically. Usually, it is used to minimize the equivalent function $-\ln(\mathcal{L})$, the *negative log-likelihood* (*NLL*). So the *NLL* to be minimized has the form[1]:

$$NLL = \sum_s n_s - \sum_{i=1}^{N} \left(\ln \sum_s n_s \mathcal{P}_s(\hat{x}_i; \hat{\theta}_s) \right), \tag{3}$$

that is a sum of logarithms. The most common method used in the high energy physics (HEP) community for the minimization is based on the MIGRAD algorithm inside the MINUIT package. MIGRAD performs the minimization using the *variable metric* method [2]. This method involves the calculation of the derivatives of the *NLL* for each free parameter. Since very often we deal with minimizing a function for which no derivatives are provided, MIGRAD is able to estimate the derivatives of the function by finite differences [3]. The whole procedure of minimization requires several evaluations of the *NLL*, which requires themselves 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 with several free parameters, many independent variables and large data samples, the minimization procedure can be very time-consuming. In this case it is important (or even mandatory) to speed-up the evaluation of the *NLL* [4].

The common software used in HEP community for the evaluation of the *NLL* is RooFit [5], which is part of the general data analysis framework ROOT [6]. Currently RooFit implements an algorithm for the *NLL* evaluation which cannot take full advantage of data vectorization and other code optimizations (like function inlining) due to its implementation based on C++ virtual methods [4]. To overcome these limitations, we have designed and implemented a new optimized algorithm, and parallelized it using a data parallelism paradigm implemented with OpenMP. The algorithm has been also implemented to run on a Graphics Processing Unit (GPU) device by using the CUDA language provided by NVIDIA. In this paper we describe the algorithm and the two implementations. Finally we show the comparison of the performance between our CPU implementation and RooFit and the comparison of the performance between CPU and GPU implementations.

2. *NLL* Evaluation

The RooFit package is formed of a set of C++ classes constructed on top of the ROOT framework dedicated to likelihood-based analyses. Basically for each mathematical concept there is a corresponding C++ class, e.g. classes for the PDFs and variables definition. Then there is a special class which takes care of finalizing the *NLL* calculation. Furthermore RooFit provides an interface to the MINUIT package. We should underline that all floating point operations are performed in double precision. Data is 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. All classes for PDFs inherit from a common abstract class, which provides the common interface. So each class has a virtual method to get the value of the PDF. Combinations of PDFs are possible with classes for adding, multiplying and convoluting basic PDFs.

In order to calculate the *NLL* from the formula (3), the current available RooFit algorithm consists of the following steps (in order):

- For a given set of values of *NLL* free parameters, loop over the events $i = 1...N$:
  - read the values of the variables for event i;

[1] We omit the $N!$ term in the expression, which does not depend on the parameters.

  - calculate the PDFs for the event i;
  - 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, which is the term in the sum of the *NLL*;
  - accumulate the terms of the sum.
- Finalize the calculation of the *NLL*.

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. Since this is done by having recourse to calls of the virtual method of each PDF, this algorithm does not allow particular code optimization, like inlining and data vectorization, and it introduces the obvious overhead due to the virtual method calls.

In order to advantage from code optimization, we redesigned the algorithm to reduce the number of calls to virtual methods. Furthermore, the data is 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. The new algorithm follows a different procedure 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 calculating 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 and finalize the calculation of the *NLL*.

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). For the implementation of this new algorithm in RooFit, we add a new virtual method 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. Then the method returns a reference to this array. Since this new virtual method is called just once per each PDF during an *NLL* evaluation, 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 does not depend by the number of events. Furthermore, thanks to the new data structure organized as arrays for each variable, this code can easily be vectorized for the calculation of each PDF. The loop over the final results of the total PDF to calculate and finalize the *NLL* evaluation is done in the usual class for the *NLL* finalization. We should note that a drawback of this new algorithm is that we have to manage all the arrays for the temporary results.

3. Parallelization Strategy and Implementation

We parallelize the implementation of the new algorithm using OpenMP. Since the iterations in the loops inside the new virtual methods for the calculation of the PDFs are independent, we can parallelize them for the execution on CPUs via the `#pragma omp parallel for` directive. Each iteration in these loops calculates the corresponding PDFs for a single event. The scheduling of the iterations is static, and each thread does a fixed number of iterations and

accesses to consecutive elements of the arrays of input variables and partial results, allowing data vectorization of the loops. Arrays of input variables and results are shared among the threads, so that there is a negligible increment in the global memory footprint of the application when running it in parallel. We parallelize also the sum over the final results of the total PDF inside the class for the *NLL* finalization (parallel reduction). To mitigate rounding problems in the parallel reduction, which depend on the number of threads, we adopt the algorithm described in [7].

Together with the implementation based on OpenMP parallelization, we also implement a parallelization for a GPU device based on the CUDA language. Basically, the loops inside the new virtual methods for the calculation of the PDFs are substituted by calls to CUDA kernels. Hence, the kernels execute the calculation of the PDFs in parallel tasks, where a task represents the calculation of a given PDF on a single event. Each task gets evaluated by a CUDA thread, i.e. there is a one to one correspondence between tasks and threads. Of course the occupancy of the resources used by each CUDA kernel depends on the corresponding PDF function to be evaluated, which means that an optimization of the occupancy would require a custom task partitioning for each kernel. However, we have checked in simple cases (a kernel used for a Gaussian PDF) that the benefit from such an optimization is small, while it would require more efforts for implementing it. So we prefer to use the current configuration, leaving to next versions of the code the possibility to consider a better partitioning of the tasks per CUDA kernel. Threads are synchronized after the corresponding kernel call for each PDF. All data and calculations execute on the GPU are, as before, in double precision floating point. Arrays of the input variables are copied from the host to the device global memory using synchronous functions. These arrays are read-only during the entire execution of the application, so we can copy them once to the device memory at the beginning of the minimization procedure and then use them for all *NLL* evaluations. The arrays of partial results for each PDF are only allocated and kept resident in the global memory of the device, i.e. no copy from device to host, except for the array of the final results which has to be copied to the host memory for the final sum of the *NLL*. Also in this case the communication is done by using synchronous functions. Then, we perform the parallel reduction of the final results using the OpenMP implementation. To summarize, during the minimization there is a single copy of the arrays of the input variables and a copy of the the array of the final results for each *NLL* evaluation. With this implementation we are able to strongly reduce the time spent for the communication between host and device memories. 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 ($< 1\%$) of 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, independent of the tasks carried out by the kernels (the default value 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.

We would like to underline that data analysts can choose which implementation to use for the *NLL* evaluation among the original RooFit algorithm, the OpenMP or the CUDA implementations, 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 three implementations.

4. Tests

In the following tests we use a statistical model based on the *BABAR* analysis for the branching fraction measurement of the neutral B meson to $\eta' K^0_S$ decay [8]. The model has 3 variables and 5 species. In total there are 29 PDFs: 11 Gaussians, 5 polynomials, 3 Argus functions, which are combined in 5 PDFs for multiplication and for addition, respectively. In our fit we leave 16 parameters free to float. The events for each species are generated from the corresponding PDFs using Monte Carlo generation techniques, not considering the time for the generation in the tests (we only consider the time spent for the fitting procedure). We use RooFit v3.14, which is part of ROOT v5.28. As minimizer we use MINUIT2 of the same ROOT distribution [9].

The first test we perform is a comparison between the original RooFit algorithm and the OpenMP implementations, running with a single thread. We run the tests on a dual socket Intel Westmere-based system: CPU (L5640) @ 2.27GHz (12 physical cores, 24 hardware threads in total), 10x4096MB DDR3 memory @ 1333MHz. 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 (20100414). In this test we are also interested to see the effect of the vectorization in the new algorithm. To do that we compile the application deactivating the data auto-vectorization made by the compiler (flag `-no-vec`). The results are shown in table 1. Due to different optimizations applied by the compiler, we expect small differences on the results when running in the three different cases (well below the parameters error values), which are negligible from the perspective of the fit results (small fractions of the free parameter errors), but they can lead to a different number of *NLL* evaluations required for the minimization. For this reason we compare the results using the wall clock time divided by the number of *NLL* evaluations. We findin that the new algoritm implementation with a single thread is about 2.5x and 4.5x faster than the RooFit implementation when running without and with vectorization, respectively. The speed-up due to the data vectorization is 1.8x, close to the theoretical maximum of 2x, since the CPU can handle in parallel two floating point operations with double precision using SIMD instructions.

Then we run another test to see the scalability of the OpenMP implementation (with vectorization) when running in parallel with a fixed number of events (100,000). We use the same system described before. The threads are pinned to the cores running them. When we reach the maximum number of physical cores (12), then the threads are pinned to hardware threads (24), still maximizing the amount of physical cores used. Also in this case the number of *NLL* evaluations required for the minimization is different due to the parallel reduction approximation (ranging between 10336 and 12665). For this reason the scalability is calculated considering the time spent for each *NLL* evaluation. The results are shown in figure 1. We reach a 8.5x speed-up with 12 threads (maximum number of physical cores) and 9.9x with 24 threads (maximum load for the system). The main limitation in the scalability is given by the remaining sequential part of the program (mainly the calculation of the normalization integral of each PDF) and from handling in memory the multiple arrays of results.

The last test we present is the comparison between running the fit with OpenMP and CUDA implementations on the CPU and GPU, respectively. In this case we use the following system: single socket Intel Nehalem-based CPU (i7 965) @ 3.2GHz (4 physical cores), with 2048MB DDR3 memory @ 1333MHz, and Graphics Card ASUS ENGTX470 (based on the NVIDIA GF100 "Fermi" architecture). The operating system and the compiler versions are the same used in the previous tests, and we use CUDA v3.2. We do the test varying the number of events of the data sample. The comparison is done considering the time spent for each *NLL* evaluation, because of the different number of *NLL* evaluations required for minimization for the two different implementations. (the difference in number of calls is up to 12% of the total calls). We take as reference the optimized parallel OpenMP implementation, requiring four parallel threads, so that we fully load the available CPU. In case of the CUDA implementation, we

Table 1. Results of the comparison executing the fit on different number of events for the three cases: original RooFit, OpenMP with one thread without vectorization, OpenMP with one thread with vectorization. The time per evaluation is obtained dividing the wall-clock time by the number of *NLL* evaluations required for the minimization, which is used in the RooFit versus OpenMP comparison.

# Events	10,000	25,000	50,000	100,000
RooFit				
# *NLL* evaluations	15810	14540	19041	12834
Time (s)	826.0	1889.0	5192.9	6778.9
Time per *NLL* evaluation (ms)	52.25	129.92	272.72	528.19
OpenMP (w/o vectorization)				
# *NLL* evaluations	15237	17671	15761	11396
Time (s)	315.1	916.0	1642.6	2397.3
Time per *NLL* evaluation (ms)	20.68	51.84	104.22	210.36
w.r.t. RooFit	2.5x	2.5x	2.6x	2.5x
OpenMP (w/ vectorization)				
# *NLL* evaluations	15304	17163	15331	12665
Time (s)	178.8	492.1	924.2	1536.9
Time per *NLL* evaluation (ms)	11.68	28.67	60.28	121.35
w.r.t. RooFit	4.5x	4.5x	4.4x	4.4x

include 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. 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. The results are shown in figure 2. We can see how the CUDA implementation behaves better for high number of events, which is due to the specific ability of the GPU architectures to take advantage of multiple threads.

5. Conclusion

In this paper we have described a different algorithm for the *NLL* evaluation in maximum likelihood fits with respect to the algorithm used in the RooFit package. We implemented this algorithm to run in parallel on CPU, using OpenMP, and GPU, using CUDA. In our test the OpenMP implementation with a single thread is about 4.5x faster than the RooFit implementation (table 1). Furthermore the OpenMP algorithm was executed in parallel, giving a speed-up of about 10x with respect to a single thread execution in our test on 12 cores (24 hardware threads) system (figure 1). The comparison between the OpenMP and CUDA implementations are made using commodity systems, that can be considered, in terms of price and power consumption, easily accessible to general data analysts. In this case, running the OpenMP implementation in parallel (with 4 threads), we were able to reach a boost of 2.8x with the CUDA implementation (figure 2).

References

[1] Cowan G 1998 *Statistical Data Analysis* (Oxford: Clarendon Press)
[2] Davidon W C 1991 *SIAM J. Optim.* **1** 1–17

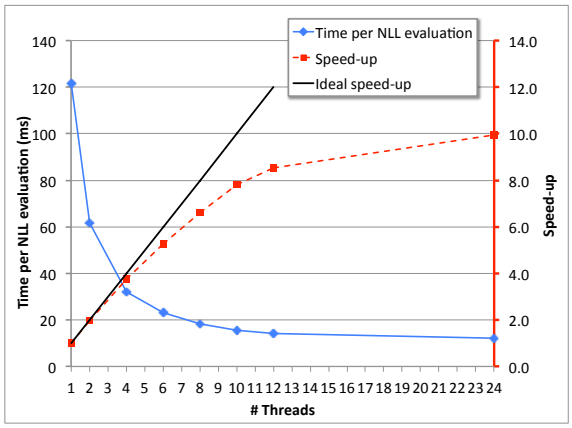

Figure 1. Time spent per *NLL* evaluation (blue line, left axis) and obtained speed-up (red line, right axis) when running the fit with 100,000 events in parallel.

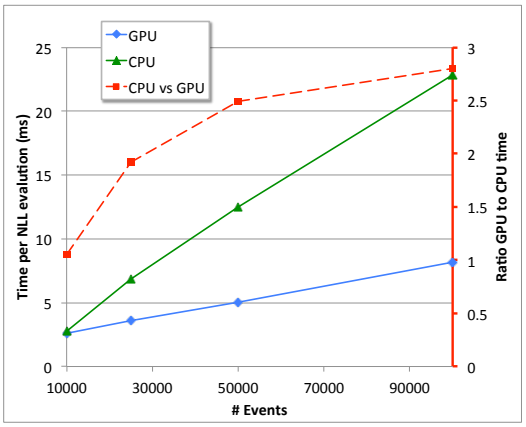

Figure 2. Time spent per *NLL* evaluation for the CPU (blue line, left axis) and GPU (green line, left axis) implementations for running the fit, varying the number of events of the data sample. The red dashed line (which refers to the right axis) shows the ratio of the values of the two lines (CPU time divided by GPU time). The CPU execution time is taken running the application with 4 parallel threads (speed-up 3.8x).

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

[4] Lazzaro A and Moneta L 2010 *J. Phys.: Conf. Series* **219** 042044

[5] Verkerke W and Kirkby D 2006 proceedings of PHYSTAT05 (London: Imperial College Press)

[6] Brun R and Rademakers F 1997 *Nuclear Instruments and Methods in Physics Research Section A* **389**(1-2) 81

[7] He Y and Ding C H Q 2001 *The Journal of Supercomputing* **18** 259–277

[8] Aubert B *et al.* (*BABAR* Collaboration) 2009 *Phys. Rev. D* **80** 112002

[9] Hatlo M *et al.* 2005 *IEEE Transactions on Nuclear Science* **52-6** 2818