Parallel Likelihood Function Evaluation on Heterogeneous Many-core Systems

Sverre JARP [a], Alfio LAZZARO [a], Julien LEDUC [a], Andrzej NOWAK [a] and Yngve SNEEN LINDAL [a,b]

[a] *CERN openlab, European Organization for Nuclear Research, Geneva, Switzerland*
[b] *Norges Teknisk-Naturvitenskapelige Universitet, Trondheim, Norway*

Abstract. This paper describes a parallel implementation that allows the evaluations of the likelihood function for data analysis methods to run cooperatively on heterogeneous computational devices (i.e. CPU and GPU) belonging to a single computational node. The implementation is able to split and balance the workload needed for the evaluation of the likelihood function in corresponding subworkloads to be executed in parallel on each computational device. The CPU parallelization is implemented using OpenMP, while the GPU implementation is based on OpenCL. The comparison of the performance of these implementations for different configurations and different hardware systems are reported. Tests are based on a real data analysis carried out in the high energy physics community.

Keywords. Likelihood function, OpenMP, OpenCL, heterogeneous parallelization

CERN-IT-2011-012
02/11/2011

Introduction

Current high energy physics experiments are collecting unprecedented large amounts of data, which gives a great opportunity to look for effects predicted by several physics models or totally unpredicted effects. It is crucial to properly analyze the data, since the new phenomena can be very rare and their contribution small compared to the total amount of data. The data samples are a collection of N independent *events*, an event being the measurement of a set of O *observables* $\hat{x} = (x^1, \ldots, x^O)$ (energies, masses, spatial and angular variables...) recorded in a brief span of time by physics detectors. The events can be classified in S different *species*. Each observable 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 parameters of the PDF. Several data analysis techniques can be used to distinguish between the events belonging to each species, using particular observables that have different PDF distributions for the species [1]. The maximum likelihood (ML) fitting procedure is a popular statistical technique used to identify events and to estimate the number of events belonging to each species and the parameters $\hat{\theta}_s = (\hat{\theta}_s^1, \ldots, \hat{\theta}_s^O)$ of the PDFs, that can be related to the prediction obtained from physics models.

This work describes a strategy for the parallelization of the likelihood function evaluation for ML fits for execution on CPU and GPU computational devices. It represents a continuation of a previous work, which described the algorithm and the corresponding implementations for CPU and GPU based on OpenMP and CUDA, respectively [2]. Here

an improved version of the CPU implementation and a new implementation for GPU based on OpenCL are described. Furthermore, a novel implementation that allows the evaluations of the likelihood function to run cooperatively on both devices belonging to the same computational node (hybrid evaluation) is introduced. It is worthwhile to point out from the beginning that the implementations are specifically optimized for running on commodity systems, i.e. systems than can be considered, in terms of cost and power consumption, easily accessible to general data analysts (e.g. a single socket desktop with a GPU whose main target is computer gaming). Performing the hybrid evaluation, data analysts can fully exploit their systems.

Existing works in literature report on the parallelization implemented on GPUs for the evaluation of likelihood functions in some specific scientific fields, such as phylogenetic analysis [3] and medical image reconstruction [4].

This paper is organized as follows. Section 1 gives an overview of likelihood function definition in the case of ML fits and the algorithm used for its evaluation. Section 2 describes the changes of the OpenMP implementation and introduces the OpenCL implementation and the hybrid solution. Section 3 reports the results obtained from tests done with a benchmark analysis on a testing system with two GPUs from two main vendors. Conclusion are given in Section 4.

1. Likelihood function evaluation

The description of the data analysis techniques based on likelihood function can be found elsewhere [1]. Here a short description in case of ML fits is given. If the observables are uncorrelated, then the total PDF for the species s is expressed by

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

The PDFs are normalized over their observables, as function of their parameters, which implies an analytical or numerical evaluation of their integral. Then the evaluation of the PDFs can be considered in two steps: evaluation of the non-normalized function values and their normalization. The *extended likelihood function* is

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

where n_s are the number of events belonging to each species. Then the ML technique allows to estimate the values of the parameters by maximizing this function with respect to the parameters to estimate. 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]:

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

[1] The $N!$ term in the expression is omitted, since it does not depend on the parameters.

that is a sum of logarithms. The terms of the sum can graphically be visualized as a tree, where the leaves are the PDFs $\mathcal{P}_s^j(x^j;\hat{\theta}_s^j)$ (basic PDFs), which are then linked to the corresponding product PDFs $\mathcal{P}_s(\hat{x};\hat{\theta}_s)$, and finally the root that is $\sum_{s=1}^{S} n_s \mathcal{P}_s(\hat{x}_i;\hat{\theta}_s)$ (sum PDF). Product and sum PDFs are denoted as composite PDFs. Therefore, the root has S child nodes, each with O children, which means that in the tree there are $S \times (O + 1) + 1$ nodes in total. The evaluation of the term in the sum of logarithms consists of traversing the entire tree, first evaluating the leaves up to the root.

The search for the minimum for *NLL* can be carried out numerically [5]. The whole procedure of minimization requires several evaluations of the *NLL*, which themselves require the calculation of the corresponding PDFs for each observable and each event of the data sample.

The algorithm for the evaluation of the *NLL* is described in detail in the previous work [2]. For each *NLL* evaluation the tree is traversed sequentially only once. The algorithm starts evaluating the basic PDFs, belonging to a given product PDF, looping over all values of their observables, and storing the results in arrays (an array for each PDF). This part is done in two phases: first evaluating all the non-normalized function values and then looping on the corresponding arrays of results for the normalization, i.e. two loops per basic PDF. Then it does the evaluation of the corresponding product PDF, looping and combining the arrays of results of the daughter PDFs in a new array. It repeats the procedure for all product PDFs. After that it loops again and combines the arrays of results of the product PDFs to get a new array of results for the sum PDF (final results). So in total there are $S \times (2 \times O + 1) + 1$ loops. Eventually, the algorithm calculates the logarithm of the final results and their sum (reduction). Parallelism has been introduced in each loop (data parallelism). Also the reduction has been parallelized. The parallel reduction can affect the final value of the *NLL*, due to rounding problem in case of associative floating point arithmetics. In particular the result can depend on the number of threads used in the parallelization, and moreover it cannot be deterministic between two different evaluations, even with the same number of threads. This can lead to unpredictable behavior during the minimization procedure, i.e. unstable results of the ML fits. For this reason a new algorithm for the reduction has been implemented. It preserves the order of the operations for a given number of threads and it reduces the rounding problem due to associative floating point arithmetics using the double-double compensation algorithm 2Sum [6]. In this way the results are deterministic and stable in all tests.

2. Parallel Implementations

This section describes the implementations for the evaluation on CPU, GPU, and the hybrid. The code is implemented in C++ and all floating point operations are performed in double precision.

2.1. OpenMP CPU Implementation

The parallelization on the CPU is based on OpenMP. It has been improved to have better scalability and overall performance with respect to the previous work.

Recalling the description of the algorithm from the previous section, the $S \times (2 \times O + 1) + 1$ loops, which are implemented as `for` loops, have been paral-

lelized via the `#pragma omp parallel for` directive. Each loop iterates N times. The scheduling of the iterations is statically partitioned, i.e. each thread executes a fixed number of iterations. The partitioning is implemented such in a way that one thread can have maximum one iteration of difference with respect to the other threads, to ensure an equal load-balancing. The same technique is applied for the loop that computes the reduction. Each thread accesses consecutive elements of the arrays of observables and results, allowing coalescing of memory accesses and data vectorization of the loops. These arrays are shared among the threads, so that there is a negligible increment in the memory footprint of the application when running in parallel. Furthermore, race conditions can be easily avoided since the parallel region are confined to the loop iterations. However, this implementation has some limitations that reduce the overall performance:

1. For each *NLL* evaluation, $S \times (2 \times O + 1) + 2$ independent OpenMP parallel regions have to be considered. This could lead to a larger overhead than necessary, which drastically reduces the scalability. It is better to have as few OpenMP parallel regions as possible, since threading overhead should be kept at a minimum.
2. $S \times (O + 1) + 1$ arrays of results and O arrays of observables have to be managed, each array composed by N double precision values. The amount of data to manage becomes consistent in case of complex models and large data samples, and it becomes crucial to have an optimal treatment of the data inside the cache memories. Tests have proved that there is a significant penalty to the scalability when running with high number of threads in a system where the largest cache memory is shared among the cores. An analysis of the problem shows that the culprits are the loops of the composite PDFs, which have to combine arrays of results with just a simple operation.

To remove the potential overhead due to OpenMP, the entire *NLL* evaluation was redesigned using a different pattern: there is only one parallel region for each evaluation, and this region will start at the root of the tree. The partitioning of the iterations is done as before, but now each thread executes the entire evaluation from the root to the leaves within its own partition only. This implementation can lead to consequences that may be problematic. Indeed, the parallel region covers a larger portion of the execution, so it is crucial to not modify member variables of the object the method is running on, or global variables, without carefully assuring that race conditions are avoided.

Three different optimizations have been considered in order to reduce the load on memory. First, in each PDF the loop for the evaluation of the non-normalized values was merged with the loop of the normalization in a single loop (loop fusion). In this way computation and memory accesses overlap. The second optimization specifically regards the composite PDFs. The code was changed so that these PDFs can send their results array "down" to the children, which then do their own evaluation and the corresponding combination directly on the array of value of the mother (results propagation). The main benefit of this change is that each basic PDF does not have to store its own results anymore. The last optimization consists of splitting the data domain into blocks so that the entire procedure of evaluation is done one by one (block splitting). This optimization directly targets cache misses, since it increase locality and thereby cache efficiency. With these optimizations the number of total loops is reduced to $S \times (O+1)$ times the number of blocks and the number of arrays of results to $S + 1$.

2.2. *OpenCL GPU Implementation*

In the OpenCL implementation the PDF loops are offloaded to be executed on the GPU. Each loop is thereby replaced by a corresponding OpenCL kernel, which runs the N iterations using GPU threads. This implementation also takes advantage of the loop fusion, as explained in the OpenMP implementation. The results propagation and block splitting are not considered. The former is tedious to implement in OpenCL (it would require a consistent duplication of the code in plain C), and the latter does not fit with the way a GPU does computations. Therefore, there are $S \times (O + 1) + 1$ kernels to launch and a corresponding same number of arrays of results to manage. Also the reduction is done in parallel (tree-based reduction) on the GPU. This reduction is deterministic and it takes into account the rounding problem described in the OpenMP implementation.

It is important to point out that the implementation is fully performance-portable between NVIDIA and AMD GPU cards. Tests have shown marginal improvements (less than 5%) when doing specific optimizations, e.g. using native vector types, which lead to different implementations for each device with respect to a common implementation.

The arrays of the observables are copied from the host to the GPU global memory using synchronous functions. These arrays are read-only during the entire execution of the application, so only one copy at the beginning is needed. They are then used for all *NLL* evaluations. For each evaluation, the CPU traverses the *NLL* tree and it launches the corresponding kernels to be executed on the GPU, following the algorithm described in Section 1. The arrays of results for each PDF can be kept resident in the GPU global memory. Eventually the reduction is done on the final array of results and the value of the sum is copied back to the host memory for the finalization of the *NLL* value. The kernels are asynchronously executed, i.e. the evaluation of the tree is non-blocking and the kernels are just enqueued for execution on the GPU. Then an implicit synchronization will occur at the end when the final reduction result has to be copied. The possibility of interleaving the CPU and GPU computations reduces the impact of the operations that are not of the loops, i.e. only executed by CPU, such as the integral calculation for the normalization.

The optimization of the occupancy is not an easy task to achieve for two main reason. First of all, it can depend on the GPU architectures from different vendors. Then it depends on what kind of PDFs analysts decide to use for their analysis. Because of that, a very general procedure, based on a heuristic approach, has been used to decide the size of the workgroups. The rule is that if a kernel contains a transcendental function, the workgroup size is set to a “low” number. If the kernel does not contain transcendentals, but rather only basic arithmetics, the workgroup size is set to a “slightly higher” number. Tests have shown that 64 and 128, respectively, provided pleasant results, with occupancy numbers ranging from 0.33 to 0.67 depending on the kernel. A comparison between using these numbers and the OpenCL default numbers gives about 14% improvement in performance.

2.3. *Hybrid Implementation*

The two implementations described in the previous sections give the possibility to fully use the CPU and GPU computational devices, but independently. In the case of the GPU implementation, the CPU runs only a single thread, so a multicore CPU would be under-

utilized. Therefore, it is interesting to explore the possibility of fully loading the CPU in a hopefully implementation-pleasant way. Although it is possible to program both CPU and GPU with OpenCL, this would lead to a worsening of the performance with respect to using OpenMP implementation since several optimizations included in OpenMP are not easy to implement in OpenCL. For this reason the hybrid solution described here allows simultaneous use of the OpenMP and OpenCL described in the previous sections.

The strategy for the hybrid implementation consists of three different steps:

1. Decomposition of the N iterations of the loops in two groups of iterations, $N_{\rm CPU}$ and $N_{\rm GPU}$ to be executed by OpenMP and OpenCL implementations, respectively.
2. Each implementation runs the entire *NLL* tree evaluation, considering the iterations $[0, N_{\rm CPU}[$ for OpenMP and $[N_{\rm CPU}, N[$ for OpenCL.
3. The result of the reductions from the two implementations are collected and summed together to finalize the *NLL* evaluation on the CPU.

In this strategy, the part that needs to be implemented is represented by the first step, since the second step requires execution of the already described OpenMP and OpenCL implementations on different ranges on the observables and the last step is trivial.

The determination of $N_{\rm CPU}$ and $N_{\rm GPU}$ is done using a load-balancer, which determines the best decomposition, i.e. both implementations spend the same amount of time. In order to have result determinism, the reductions must be executed for the same static configuration during all *NLL* evaluations in a ML fit. This means that users can run the load-balancer once at the beginning to determine the decomposition, and then fixing it for their ML fits.

The load-balancer starts assuming $N^{(1)}_{\rm CPU} = N^{(1)}_{\rm GPU} = N/2$, where the apex index denoted that is the first configuration. Then the two implementations are executed for this configuration. Their execution times are $t^{(1)}_{\rm CPU}$ and $t^{(1)}_{\rm GPU}$, respectively. If the ratio $\max(t^{(1)}_{\rm CPU}, t^{(1)}_{\rm GPU})/\min(t^{(1)}_{\rm CPU}, t^{(1)}_{\rm GPU})$ is less then a given threshold, then an optimal decomposition was reached. Otherwise, the procedure is repeated with a new configuration [7]:

$$N^{(2)}_{\rm D} = N \times \frac{N^{(1)}_{\rm D}/t^{(1)}_{\rm D}}{N^{(1)}_{\rm CPU}/t^{(1)}_{\rm CPU} + N^{(1)}_{\rm GPU}/t^{(1)}_{\rm GPU}}, \tag{4}$$

where the subscript D represents CPU or GPU, and so on until the threshold condition is satisfied. The measurement of the execution times of both CPU and GPU implementations is done using the `omp_get_wtime` OpenMP function. It is possible to consider averages from several *NLL* evaluations to reduce the fluctuations of these timings (the number of evaluation can be set by the users depending on the complexity of their *NLL*). In the case the threshold condition is not satisfied after a given number of cycles of the load-balancer, because of large fluctuations with respect to a small threshold, the procedure ends and the last configuration is returned. Tests presented in this paper have shown that the load-balancer is able to reach the convergence after only 3 cycles.

The hybrid implementation is itself based on OpenMP. Essentially, the OpenMP implementation starts $P + 1$ threads, where P is the number of threads used for the *NLL* evaluation, while the other thread is used by the OpenCL implementation. This is possible since in the OpenMP implementation starts a single OpenMP parallel region at

the root of the *NLL* tree, so that it is possible to branch between the two implementations at the beginning of the evaluation. Then the OpenMP standard guarantees an implicit synchronization at the end of the parallel region. It is important to remember that the OpenCL kernel calls are non-blocking and ideally consume minimal CPU time, i.e. the evaluation of the tree on the CPU impose a minimal overhead. If this holds, then users can decide to run $P + 1$ threads on a CPU with P cores.

3. Tests

In the following tests a statistical model based on the BABAR analysis for the branching fraction measurement of the B meson to $\eta' K$ decay is used [8]. There are 3 observables and 5 species. In total there are 21 PDFs: 7 Gaussians, 5 polynomials, 3 Argus functions, which are combined in 5 PDFs for multiplication and one for addition, respectively. All PDFs have an analytical integral. The number of events considered ranges between 10k and 1M events. Each run does 1000 times a pure *NLL* function evaluation, and we time 5 runs to achieve an accurate timing result.

The CPU system is an Intel Core i7 965 Nehalem running at 3.2 GHz, with 2GB DDR3 RAM. It is a quad-core CPU and it also supports SMT, which means that it has the ability to physically execute 8 threads simultaneously and on a per-core basis (with 2 threads). Two GPUs are used: NVIDIA GeForce GTX470 and AMD Radeon HD5870.

The OpenMP CPU implementation improves the previous one for a single thread by a factor 1.8x. The new speed-up results are: 1.8x with 2 threads, 3.6x with 4 threads, and 4.7x with 8 threads, independently by the number of events used in the test. In the previous implementation the maximum speed-up was 2.5x with 8 threads.

The results for the OpenCL GPU implementation are shown in Figure 1 on the left plot. Only in this case tests are also executed on a NVIDIA Tesla C2050. This implementation is not beneficial for a low number of events with respect to the OpenMP implementation. This is a direct consequence of the need to copy the final value over the PCI-Express bus. Another interesting result is that neither the HD5870 nor the Tesla C2050 give any higher speedups than the GTX470, although theoretically they are almost 4x faster than the GTX470 when performing double-precision arithmetic. The reason for this is that the computation is completely memory-bound, so all the ALUs on the cards are starved while waiting for memory reads.

Figure 1 shows also the results from the hybrid implementation on the right plot. A general observation is that N must be large to gain anything on this, so that the overhead can be amortized. Running the GTX470 (with 4 threads in the OpenMP implementation) and the HD5870 (3 threads) in a hybrid scenario is very beneficial, achieving nearly perfect balancing for high workloads.

4. Conclusion

Two new implementations for the *NLL* evaluation have been presented in this paper, based on OpenMP and OpenCL. It is also described a novel approach to the hybrid evaluation for GPU and CPU computational devices. The OpenMP implementation gives better performance and scalability with respect to the previous implementatin reported

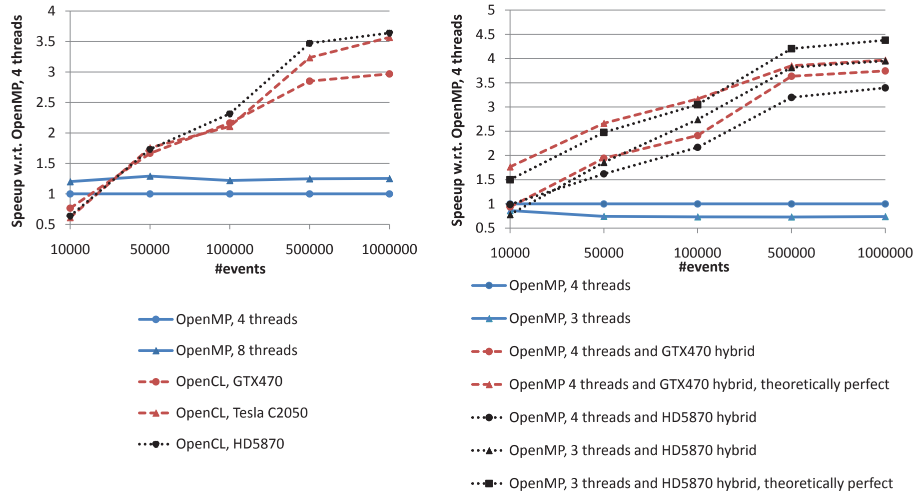

Figure 1. Comparison between CPU and GPU for the OpenCL implementation (left plot) and the hybrid implementation (right plot). The reference is the OpenMP implementation with 4 threads. The "theoretically perfect" lines are obtained when summing the performance of the OpenMP and OpenCL implementation alone, i.e. no considering the overhead from the hybrid implementation.

in [2]. Comparing this version to an OpenCL implementation, it is possible to conclude that the GPU should be used for a sufficient number of events, and this is suitable since the need for computing power increases with N. The possibility of using the hybrid implementation further improves the performance.

References

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

[2] S. Jarp *et. al.*, *Parallelization of Maximum Likelihood Fits with OpenMP and CUDA*, proceeding of "The International Conference on Computing in High Energy and Nuclear Physics", Taipei (Taiwan), 18-22 October, 2010, to be published on Journal of Physics: Conference Series. CERN-IT-2011-009; S. Jarp *et. al.*, *Evaluation of Likelihood Functions for Data Analysis on Graphics Processing Units*, ipdpsw, pp. 1349–1358, 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, 2011. CERN-IT-2011-010.

[3] 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.

[4] 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.

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

[6] P. Kornerup *et. al.*, *On the computation of correctly-rounded sums*, IEEE Transactions on Computers, 1 February, 2001.

[7] I. Galindo *et. al.*, *Dynamic load balancing on dedicated heterogeneous systems*, in volume 5205 of *Lecture Notes in Computer Science*, pp. 64–74, Springer, 2008.

[8] B. Aubert *et. al.*, *B meson decays to charmless meson pairs containing η or η' mesons*, Phys. Rev. D80, 112002, 2009.