Evaluation of likelihood functions on CPU and GPU devices

Sverre Jarp, Alfio Lazzaro, Julien Leduc, Andrzej Nowak and Yngve Sneen Lindal

CERN openlab, Geneva, Switzerland

E-mail: sverre.jarp@cern.ch, alfio.lazzaro@cern.ch, julien.leduc@cern.ch, andrzej.nowak@cern.ch, yngve.sneen.lindal@cern.ch

Abstract. We describe parallel implementations of an algorithm used to evaluate the likelihood function used in data analysis. The implementations run, respectively, on CPU and GPU, and both devices cooperatively (hybrid). CPU and GPU implementations are based on OpenMP and OpenCL, respectively. The hybrid implementation allows the application to run also on multi-GPU systems (not necessarily of the same type). The hybrid case uses a scheduler so that the workload needed for the evaluation of function is split and balanced in corresponding sub-workloads to be executed in parallel on each device, i. e. CPU-GPU or multi-CPUs. We present the results of the scalability when running on CPU. Then we show the comparison of the performance of the GPU implementation on different hardware systems from different vendors, and the performance when running in the hybrid case. The tests are based on likelihood functions from real data analysis carried out in the high energy physics community.

1. Introduction

The evaluation of a likelihood function is required in several data analysis techniques, such as the maximum likelihood fitting procedure [1]. These techniques are largely used in high energy physics (HEP) experiments to analyze the data they collect. 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 that can be related to the prediction obtained from physics models. 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. 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. It is usual to evaluate the equivalent function $-\ln(\mathcal{L})$, the *negative log-likelihood* (*NLL*)[1]:

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

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 in traversing the entire tree, first evaluating the leaves up to the root. The time spent for this evaluation depends on the number of events and the complexity of the PDFs. Furthermore, the entire evaluation is performed several times during the data analysis for different sets of the PDF parameters. Hence, it becomes important to speed-up the evaluation to have fast data analyses.

The common software used in HEP community for the evaluation of the *NLL* is RooFit [2], which is part of the general data analysis framework ROOT [3]. Currently RooFit implements an algorithm for the *NLL* evaluation that cannot take full advantage of data vectorization and other code optimizations (like function inlining). A redesign of the algorithm and the corresponding implementation, with vectorization and several code optimizations, was described in Refs. [4] and [5]. This algorithm was also parallelized for CPU and GPU execution, based on OpenMP and CUDA, respectively. An improved version of the OpenMP implementation and a new implementation for GPU based on OpenCL are described in Ref. [6]. In the same reference, there is a description of a new implementation that allows the evaluations of the likelihood function to run cooperatively on a CPU and GPU belonging to the same computational node (hybrid evaluation). In this paper, the OpenMP and OpenCL implementations are briefly reported, while an improved hybrid implementation, which allows running on multi-GPU systems (not necessarily of the same type) 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. single socket desktop with a GPU whose main target is computer gaming). Data analysts can fully exploit their systems when performing the hybrid evaluation.

This paper is organized as follows: Section 2 describes the OpenMP and OpenCL implementations and introduces the the new hybrid solution; Section 3 reports the results obtained from tests done with a benchmark analysis on a testing system with NVIDIA and ATI GPUs; conclusion are given in Section 4.

2. Parallel Implementations

This section describes the implementations for the evaluation on CPU, GPU, and the hybrid configuration. The code is implemented in C++ and all floating point operations are performed in double precision. Other details on these implementations can be found in Ref. [6].

2.1. OpenMP CPU Implementation

The parallelization on the CPU is based on OpenMP. Each type of PDF is implemented by a specific class. All inherit from a common interface that gives the public virtual method `getVal` and the protected pure virtual method `evaluate`. The former method computes the

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

normalization integral of the PDFs (in case of basic PDFs) and then it calls the latter method, which is overloaded inside each PDF class and it performs a loop for the evaluation of the function values. In total there are $S \times (O + 1) + 1$ loops that are implemented as `for` loops. There is only one parallel region for each evaluation, and this region starts at the root of the tree. This is implemented via a `#pragma omp parallel` directive. Therefore each loop executes a given subset of the N iterations. They are statically partitioned, i.e. each thread executes a fixed number of iterations. The partitioning is implemented in a way so that one thread can have at most one iteration of difference with respect to the other threads, to ensure an equal load-balancing. Eventually, the algorithm computes the logarithm of the final results, summing them up with a loop (reduction), which is also parallelized. To ensure the reproducibility of the results a specific 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 non-associative floating point arithmetic, using the double-double compensation algorithm 2Sum [7].

Each thread executes the entire evaluation from the root to the leaves within its own partition only and it 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 increase in the memory footprint of the application when running it in parallel. This implementation can lead to consequences that may be problematic in the case of a complex C++ code like RooFit. Indeed, the parallel region covers a larger portion of the execution, so it is crucial not to modify member variables of the object the method is running on, or global variables, without carefully assuring that race conditions are avoided.

Several arrays of temporary results and O arrays of observables have to be managed for each evaluation, each array composed by N double precision values. The amount of data to manage becomes significant in the case of complex models and large data samples, and it becomes crucial to have optimal treatment of the data inside the cache memories. Tests have proved that there is a significant penalty to the scalability when running with a high number of threads in a system where the largest cache memory is shared among the cores. Some different optimizations have been considered in order to reduce the load on memory. The composite PDFs, which have to combine arrays of results with just a simple operation, 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 parent (results propagation). The main benefit of this change is that each basic PDF does not have to store its own results. Another optimization consists in splitting the data domain into blocks so that the entire procedure of evaluation is done one by one of them (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 $S \times (O + 1)$ 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 a GPU. Each loop is thereby replaced by a corresponding OpenCL kernel which runs the N iterations using GPU threads. Also this implementation takes advantage of the results propagation as explained in Section 2.1 for the OpenMP implementation. The block splitting is not considered because it does not fit with the way a GPU does computations. The results propagation for the composite PDFs is not feasible for OpenCL (it requires more coding with respect to the OpenMP implementation, e.g. because templates are not supported). There is an independent kernel to launch for each PDF, therefore $S \times (O + 1)$ kernels, and a corresponding equal number of arrays of results to manage. The reduction is performed in parallel using a tree-based algorithm. It is deterministic and takes into account the non-associativity problem described in the OpenMP implementation.

The key reason for using OpenCL is the portability between vendors, which is particularly interesting for users using commodity systems. The implementation is fully performance-portable between NVIDIA and AMD GPUs. 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. Then they are 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. 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 queued for execution on the GPU. The possibility of interleaving CPU and GPU computations reduces the impact of the operations that are not in loops, i.e. only executed by CPU, like the integral calculation for the normalization.

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 good 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. The number of blocks per kernel is then computed from the number of events divided by the number of threads per block, rounded to the greatest integer number.

2.3. Hybrid Implementation

The two implementations described in the previous sections make it possible to fully use the CPU and GPU computational devices independently. In the case of the GPU implementation, the CPU runs only a single thread, so a multicore CPU would be underutilized. Therefore, it is interesting to explore the possibility to fully load the CPU in an implementation-pleasant way. Another interesting possibility is to cooperatively use GPUs in a multi-GPU system. The hybrid solution described here allows to achieve these goals, running simultaneously the OpenMP and OpenCL implementations described in the previous sections.

In general a *compute device* is defined as some device capable of computation inside one physical computer. In the context of this work, it can be a CPU or a GPU. The physical computer has a single CPU and several possible GPUs. It then becomes important to have correct load balancing of the workload between such devices. Load balancing is a very fundamental and old problem, and much research has been done on it (Refs. [8, 9, 10] describe some examples of load balancing scenarios within computer science). The hybrid implementation described in this work uses a refining static load balancing for the evaluation of the *NLL*. The whole evaluation can be regarded as data-parallel, and therefore the whole domain is split into partitions so that each compute device is responsible for computing one and only one partition. A central concept is the importance of result determinism. It is *crucial* that each compute device has a fixed range before actual result computation starts, since different ranges in the same run would lead to different results when the final reduction is done. This means that the balancing must be an initial phase before a real evaluation is performed. The strategy used for implementing the heterogeneous data-parallel load-balancer in this work consists of:

- Starting an initial balancing phase by assigning to each compute device a range of length approximately N/K, where N is the total number of events, and K is the number of compute devices. This will lead to timing value t_k for each computational device, with $k = 1, \dots K$. The distribution of timing values will most probably be highly sub-optimal if the computational capabilities of the devices differ.

- Iterating and adjusting the partitioning based on these timings. Hopefully, this will converge within some time threshold, assuring a nearly optimal execution, i.e. a work partitioning that minimizes the difference between all timing values within some threshold.

Therefore, the determination of the optimal balancing is performed using *a priori* self-adjusting static load-balancing, so that the computation domain is first decomposed in such a way that each device will spent almost same amount of time to execute its own sub-workload. Then the decomposition is kept fixed (static) during all *NLL* evaluations. The downside of this strategy is that the entire subsequent evaluation will be based on this phase. If conditions in the system change, e.g. false timings during the initial phase or external load on the hardware during evaluation, the balancing can be sub-optimal.

The actual method used for dividing work between compute devices based on their respective execution times was originally described in [11] and was also used with seemingly good results in [12]. Let N_k^b denote the partition for compute device k (i.e. the number of events for that device) for a given balancing cycle b, so that the events are split in $\left[1, N_1^b\right], \left[1 + N_1^b, \sum_{k=1}^{2} N_k^b\right], \dots \left[1 + \sum_{k=1}^{K-1} N_k^b, N\right]$ group for each device, respectively. Then each device evaluates its partition by mean of the specific implementation, i.e. OpenMP or OpenCL, spending the execution time t_k^b. The *relative power* for each device (i.e. its relative compute capability) is defined by:

$$RP_k^b = \frac{N_k^b}{t_k^b} \tag{4}$$

and the total power as

$$SRP^b = \sum_{k=0}^{K} RP_k^b. \tag{5}$$

Now each new partition can be computed by

$$N_k^{b+1} = N * \frac{RP_k^b}{SRP^b}. \tag{6}$$

The procedures ends when the maximum and the minimum of the distributions of t_k^b differ within some threshold. Each implementation runs the entire *NLL* tree evaluation. 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*). The threshold value is increased when the condition is not satisfied after a given number of cycles of the load-balancer because of large fluctuations with respect to a small threshold. In any case, the minimum value of the threshold can be related to the time to execute a minimum number of events (granularity). Tests presented in this paper have shown that the load-balancer is able to reach convergence after only 3 cycles.

Once the best partition is found, the hybrid implementation consists in three different steps:

(i) Decomposition of the N iterations of the loops in groups of iterations N_k^{best} for each device k.

(ii) Each device runs the corresponding implementation for the entire *NLL* tree evaluation, considering its iterations. Then it runs the reduction.

(iii) The result of the reductions from each device 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. The second step executes the OpenMP and OpenCL implementations already described on different ranges on the observables, and the last step is trivial.

The hybrid implementation is itself based on OpenMP. Essentially, the OpenMP implementation starts $P + K - 1$ threads, where P is the number of threads used for the *NLL* evaluation, while the other threads are used to run the GPU implementation for each GPU devices ($K - 1$ GPUs). 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 imposes a minimal overhead. If this holds, then users can decide to run $P + K - 1$ threads on a CPU with P cores.

3. Tests

The following tests a statistical model based on the *BABAR* analysis for the branching fraction measurement of the B meson to $\eta' K$ decay [13]. There are 3 observables and 5 species. In total there are 21 PDFs: 7 Gaussians, 5 polynomials, 3 Argus functions, combined by 5 PDFs for multiplication and one for addition. All PDFs have an analytical integral. The number of events considered ranges between 10k and 1M. The model is implemented by a simplified version of the RooFit package. It is an easy-to-compile implementation which preserves the same programming interface of RooFit but without all the physics-related details. Tests are done considering 1000 *NLL* evaluations so that it is possible to compare the results for different number of events. The entire tests are repeated 5 times 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 supports SMT, which means that it has the ability to physically execute 8 threads simultaneously. Two GPUs are used: NVIDIA GeForce GTX470 and AMD Radeon HD5870. The code is compiled to run on the CPU using Intel Composer XE 2011 (icpc v12).

In the case of the OpenMP CPU implementation the speed-up is defined as the ratio between the execution times spent by the application running with a single thread and with a certain number of threads in parallel. The fraction of the execution time spent in code that can be executed in parallel is 96%. The results are: 1.8x with 2 threads and 3.6x with 4 threads. They are consistent with the theoretical speed-up values obtained by the Amdahl's law, i.e. 1.9x for 2 threads and 3.6x for 4 threads. Using the SMT threads, the overall speed-up becomes 4.7x with 8 threads, which means a benefit of 30% due to SMT. All results do not depend on the number of events used in the test.

In the case of the OpenCL GPU and the hybrid implementations the speed-up results are obtained from the ratio of the execution time of the exclusive OpenMP CPU implementation with 4 threads and the execution time of the configurations with GPUs. The reason of this choice is to see how fast we can go using the GPUs with respect to the parallel execution on the CPU only. The results for the OpenCL GPU implementation are shown in figure 1. For this case only, 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 CPU implementation with 4 threads (speed-up < 1.0x), while it reaches a speed-up between 3.0x–3.6x for an high number of events. This is a direct consequence of the need to copy the final value over the PCI-Express bus. An 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 is that the computation is completely memory-bound so all the ALUs on the cards are starved waiting for memory reads [14]. Figure 2

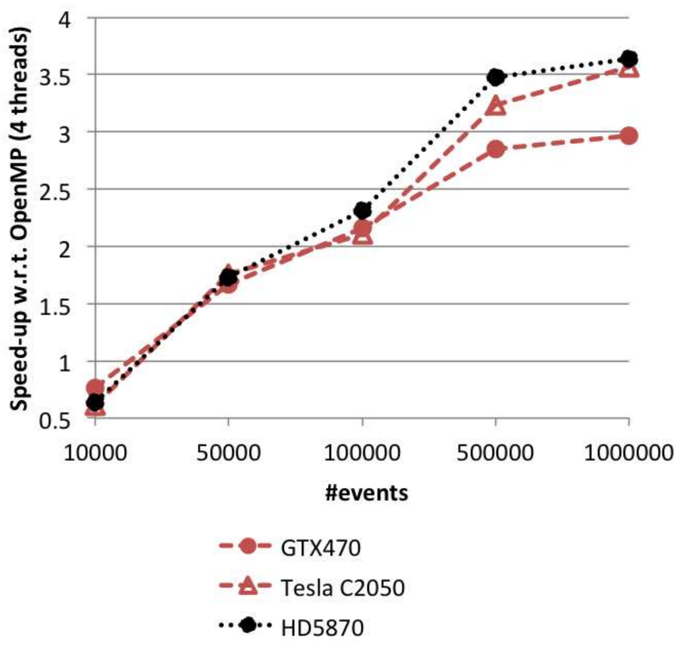

Figure 1. Comparison between OpenMP CPU and OpenCL GPU implementations. The reference is the OpenMP implementation with 4 threads.

shows the results from the hybrid implementation. A general observation is that N must be large to gain anything since the overhead must be amortized. Balancing between the GTX470 and the CPU (with 4 threads in the OpenMP implementation) is very beneficial, achieving nearly perfect balancing for high workloads. In this case the speed-up increases from 3.0x when using the exclusive OpenCL GPU implementation to 3.8x of the hybrid case (4.0x is the theoretical maximum). Running with HD5870 and the CPU however leads to an almost negligible gain, i.e. the speed-up does not increase with respect to the exclusive OpenCL GPU implementation. This is due to an inefficiency of the AMD GPU driver for the OpenCL execution, which implies a significant overhead for the its execution on the CPU. The multi-GPUs configuration gives the best performance, reaching a speed-up of 6.1x, close to the theoretical maximum of 6.8x.

4. Conclusion

Two implementations for the *NLL* evaluation have been presented based on OpenMP and OpenCL and run on CPU and GPU, respectively. A novel approach has been presented that allows the hybrid evaluation for CPU and GPU computational devices including multi-GPUs systems. One can conclude that the GPU can be used for a sufficient number of events, and this is suitable since the need for computing power increases with N. Using the hybrid implementation further improves the performance; the multi-GPUs configuration gives the best performance. More details on this work can be found in Ref. [14].

Acknowledgments

The authors would like to thank Filippo Spiga of Irish Centre for High-End Computing for providing us the system with the NVIDIA Tesla card and Felice Pantaleo for his suggestions in the GPU implementation.

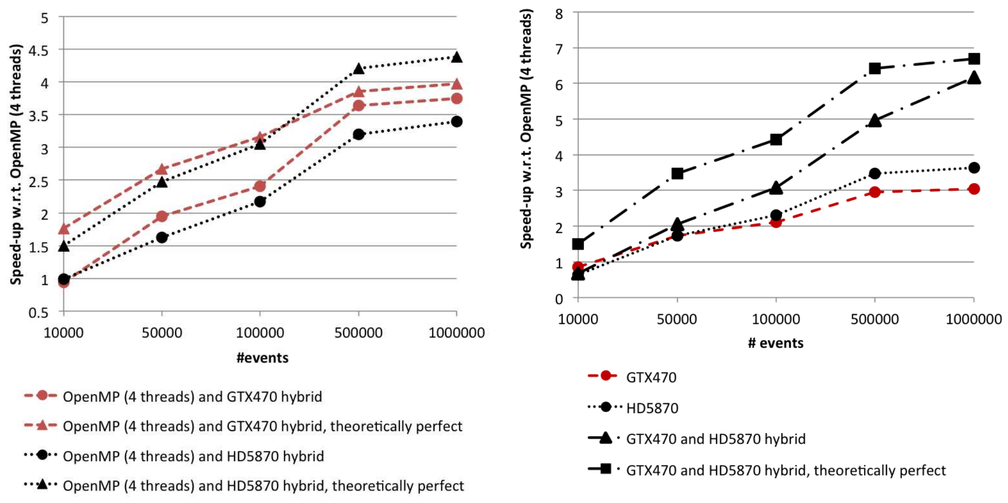

Figure 2. Comparison between OpenMP CPU, OpenCL GPU and the hybrid implementation (CPU + GPU on the left plot, multi-GPUs on the right plot). The reference is the OpenMP implementation with 4 threads. The "theoretically perfect" lines are obtained when summing the performance of the OpenMP and OpenCL implementation alone, i.e. no considering the overhead from the hybrid implementation.

References

[1] Cowan G 1998 *Statistical Data Analysis* (Oxford: Clarendon Press)

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

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

[4] Jarp S *et al.* 2011 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum pp 1349–1358 (*Preprint* `CERN-IT-2011-010`)

[5] Jarp S *et al.* 2012 Journal of Physics: Conference Series (*Preprint* `CERN-IT-2011-009`)

[6] Jarp S *et al.* 2011 Parallel likelihood function evaluation on heterogeneous many-core systems proceeding of International Conference on Parallel Computing, Ghent, Belgium (*Preprint* `CERN-IT-2011-012`)

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

[8] Aller B *et al.* 1997 *ACM Trans. Comput. Syst.* **15**(3) 253–285

[9] Ali R *et al.* 1995 *IEEE Computer Society Press*

[10] Ferrer R *et al.* 2011 *Proceedings of the 23rd International conference on Languages and compilers for parallel computing* 215–229

[11] Galindo I *et al.* 2008 *Lecture Notes in Computer Science* **5205** 64–74

[12] Acosta A *et al.* 2010 467–476

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

[14] Lindal Y S 2011 Optimizing a high energy physics toolkit on heterogeneous architectures (*Preprint* `CERN-THESIS-2011-153`)