IOPscience iopscience.iop.org

Home Search Collections Journals About Contact us My IOPscience

Matrix element method for high performance computing platforms

This content has been downloaded from IOPscience. Please scroll down to see the full text.

2015 J. Phys.: Conf. Ser. 664 092009

(http://iopscience.iop.org/1742-6596/664/9/092009)

View the table of contents for this issue, or go to the journal homepage for more

Download details:

IP Address: 92.31.191.132
This content was downloaded on 15/05/2017 at 21:51

Please note that terms and conditions apply.

You may also be interested in:

The Matrix Element Method at the LHC: status and prospects for Run II
Sébastien Wertz

The automated Matrix-Element reweighting and its applications at the LHC
Alexandre Mertens

Bose operator expansions of tensor operators in the theory of magnetism. II
P -A Lindgard and A Kowalska

Momentum dependence of generalised oscillator strengths of atomic K shells using the local plasma approximation
C M Kwei and C J Tung

Top quark mass results at Tevatron
Pavol Bartoš and Cdf and DØ collaborations

Quasiprobability distribution for the photon-number operator and quadrature operator in a coherent state
V Perinová, A Luks and J Krepelka

Line-fitting procedures for intermediate structure
J E Lynn and J D Moses

The Low Lying Electronic States of GeO
A Lagerqvist and I Renhorn

Computational particle physics for event generators and data analysis
Denis Perret-Gallix

Journal of Physics: Conference Series **664** (2015) 092009 doi:10.1088/1742-6596/664/9/092009

Matrix element method for high performance computing platforms

G Grasseau[1], D Chamont[1], F Beaudette[1], L Bianchini[2], O Davignon[1], L Mastrolorenzo[1], C Ochando[1], P Paganini[1] and T Strebler[1], on behalf of the CMS Collaboration

[1]Laboratoire Leprince-Ringuet, Ecole polytechnique, Palaiseau, France
[2]ETH Zurich, Switzerland

Abstract. Lot of efforts have been devoted by ATLAS and CMS teams to improve the quality of LHC events analysis with the Matrix Element Method (MEM). Up to now, very few implementations try to face up the huge computing resources required by this method. We propose here a highly parallel version, combining MPI and OpenCL, which makes the MEM exploitation reachable for the whole CMS datasets with a moderate cost. In the article, we describe the status of two software projects under development, one focused on physics and one focused on computing. We also showcase their preliminary performance obtained with classical multi-core processors, CUDA accelerators and MIC co-processors. This let us extrapolate that with the help of 6 high-end accelerators, we should be able to reprocess the whole LHC run 1 within 10 days, and that we have a satisfying metric for the upcoming run 2. The future work will consist in finalizing a single merged system including all the physics and all the parallelism infrastructure, thus optimizing implementation for best hardware platforms.

1. Brief introduction to MEM for CMS H→ττ analysis

The Matrix Element Method (MEM) [1] is a well-known approach in high energy physics, used to analyse data arising from particle collisions. As opposed to other machine learning methods requiring training, the MEM allows a finer direct comparison between theory and observations, but unfortunately it consumes much more CPU time. This makes it hardly exploitable, unless implemented with parallel technologies [2]. The work described in this article is applied to the analysis of the CMS events produced through the so-called Vector Boson Mecanism. A Vector Boson Fusion (VBF) H→ττ candidate can be seen in figure 1 below.

For each observed event x (real or simulated), we want to compute a weight quantifying the probability that this event arises from a given theory model Ω (here H→ττ). A simplified formulation is:

$$P(\mathbf{x}|\Omega) = \frac{1}{\sigma_\Omega} \iiint dx_1 dx_2 d\mathbf{y} \mathcal{P}_s(x_1, x_2) |\mathcal{M}_\Omega(x_1, x_2, \mathbf{y})|^2 W(\mathbf{x}, \mathbf{y})$$

Within the integral above, the P_s function describes the probability that the initiating partons take a fraction *x1* and *x2* of the incoming proton momenta, the M_Ω function is the matrix element squared for obtaining a final state y, and W expresses the probability density of measuring the event x in the detector given the final state particles y.

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

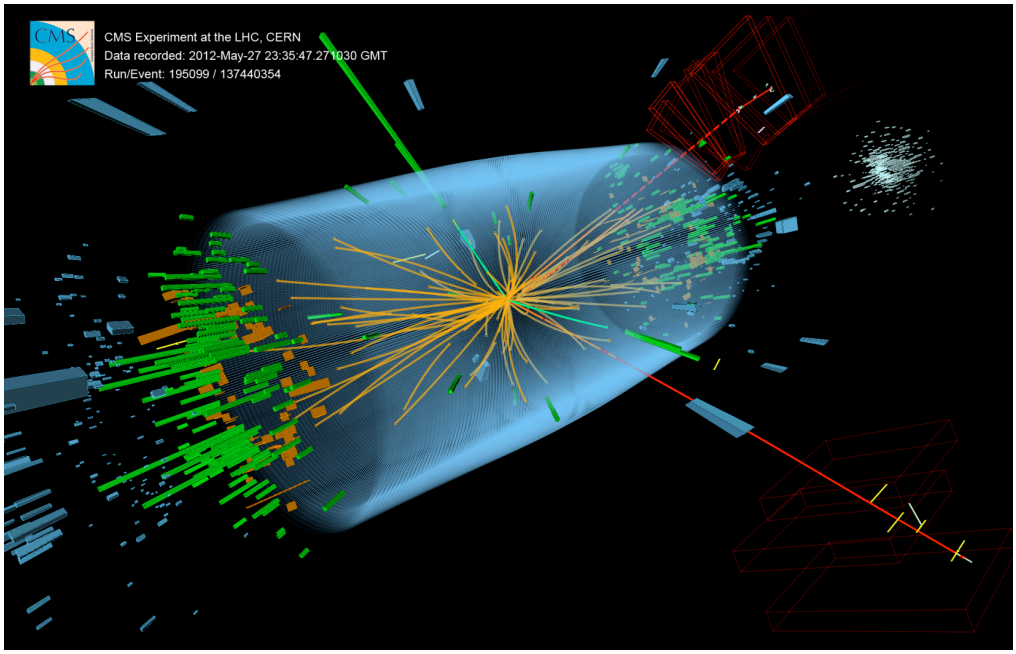

Figure 1. A CMS H→ττ candidate event.

The MEM has never been applied before to final states with tau leptons at the LHC.

2. Computing aspects

For each event, one must compute a multi-dimensional integral with a Monte Carlo approach [3]. In our use-case, the integral has been simplified down to 5 dimensions, and one needs at least 20k points for its Monte-Carlo evaluation.

Several different software packages were used as a starting point for our software system:

- For the parton distribution function *Ps*: LHAPDF [4][5].
- For the matrix element M_{Ω}: the code generated with MadGraph [6][7].
- The transfer function *W* is specific to the studied channel and the detector features. It requires a custom-made development.
- For the Monte-Carlo evaluation of the integral: the GSL [8] implementation of the VEGAS [9][10][11] algorithm, as provided by ROOT [12].

The complete run 1 dataset, collected between 2011 and 2012 at a center-of-mass energy of 7-8 TeV, consists of about 200k events, which must be processed twice: once to evaluate the probability *Ps* assuming that the event is a signal one (H→ττ), once to evaluate the probability *Pb* assuming that the event comes from background. The processing time estimation for one event being about half an hour, the complete run 1 reprocessing, sequentially, would require about 4100 days.

For the Run 2, to be acquired from 2015 to 2018 at a center-of-mass energy of 13-14 TeV, the processing time of each single event should be similar to Run 1, but we expect about x5 more events.

Obviously enough, we must parallelize our software so as to accumulate every computing resource available (nodes and devices), and perform the analysis within an acceptable time-frame.

3. Key technological choices

In order to achieve this goal, we made two technological choices: OpenCL [13] to take profit of the resources within a node (server), and MPI [14] to accumulate all the nodes available. Combined use of MPI and OpenMP is already a common practice. The combined use of MPI and OpenCL goes one step further, because it can also manage the computing accelerators, also called co-processors or "devices" in the OpenCL terminology.

3.1. OpenCL

The OpenCL standard, specified by the Khronos Group, provides a way to write portable code which can run on all traditional CPUs as well as GPGPU and many-core devices. An additional benefit of OpenCL is its abstract model which is well suited for the SIMD instructions. Unfortunately the initial price to pay is rather high: most of the code must be translated into a subset of C99.

3.2. MPI

The Message Passing Interface is the most widely used way to distribute computation (distributed memory model) between different computing servers (nodes) linked with a high bandwidth network.

Currently, we mainly use MPI to distribute the events to be analysed, and this is embarrassingly parallel, so we may rely on grid or big data middleware. Yet we preferred to start from the beginning with MPI, which can also handle problems which require fast communication between the nodes.

For example, when we will address the computation of the normalization integral, which has even more dimensions, we will study the distribution of sub-domains of the integral.

Also, MPI has proved to be useful to dynamically control the load balancing between the nodes, and to solve an issue with the poor NVidia OpenCL implementation, which cannot handle several devices.

4. Two lines of development

We pursue two developments in parallel, one focused on physics and one focused on parallel programming. Of course, the ultimate goal is to merge them into a single software system.

The **MEM-MPI** pre-production code contains all the physics for the channel $H \rightarrow \tau\tau$, fully interfaced with CMS software and data, but only MPI instrumentation. It is driven by physicists. For this preproduction code, MPI is currently used both to connect multiples nodes, and also to use all the cores within a node (until OpenCL is ready).

The **MEM-MPI-OCL** prototype contains all the machinery to opportunely accumulate any processing hardware available, combining MPI and OpenCL, but only a subset of the physics has been ported to OpenCL today. It is driven by engineers of the team.

5. The MEM-MPI first results

The results of this code have been validated by the physicists of the team (see figure 2), and it is under certification with CMS official simulated and real data.

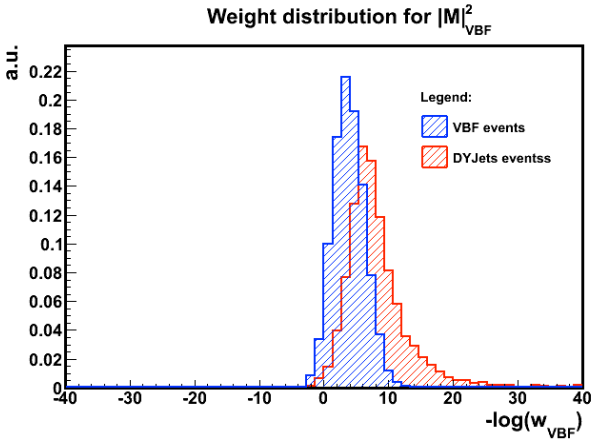

Figure 2. Distribution of $-\log(W_{VBF})$ for signal (VBF - in blue) and background (DY - in red) simulated events. W_{VBF} corresponds to the $P(X|\Omega)$ under VBF hypothesis.

For what concerns the computing time, we tried to use an increasing number of simultaneous MPI processes, each time processing a global number of events equal to three times the number of processes, so that each process analyses three events. The Monte Carlo evaluation of the integral was done with 20k points.

When less than 16 MPI processes are considered, they are submitted on the cores of a single node. Beyond 16, the processes are distributed over several nodes.

As one can see in the figure 3, we have observed that the performance scales very well from 1 to 16 cores within a single node, and until 4x16 cores with 4 full nodes. This basically confirms that our problem is very well suited for parallelism.

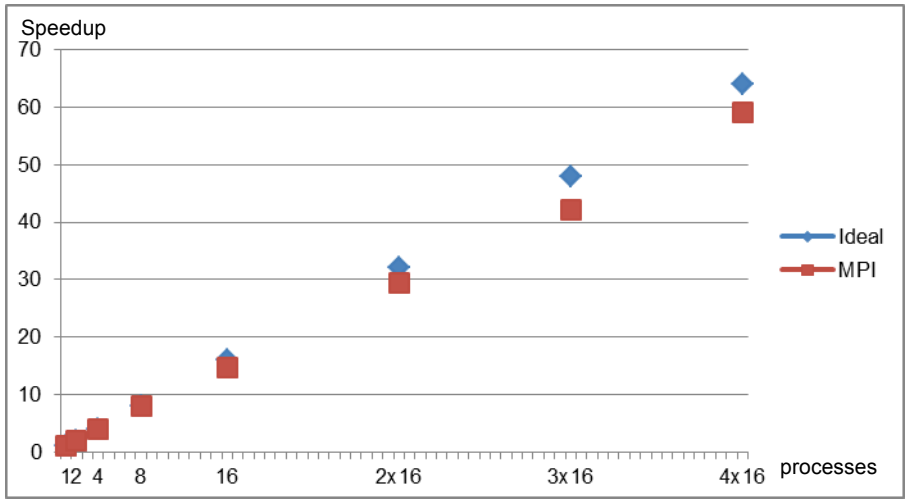

Figure 3. MEM-MPI speedup when increasing the number of MPI processes.

6. The MEM-MPI-OCL prototype

6.1. Porting status

The major task in this work is the translation to OpenCL of all the code involved (~30k lines of code). This translation is generally going through two steps :

1. From C++ and Fortran to C99.
2. From C99 to OpenCL.

The translation status of each big subset of the code is the following:

- VEGAS integration, ~2k lines of code: done
- LHAPDF, ~1.6k lines of code: we only lack the final numerical validation.
- MadGraph code generation, ~20k lines of code: the first step of translation towards C99 is done, but we still do the second by hand, and it is our priority to make it automatic.

Although our priority is yet to complete the OpenCL translation, we could not refrain from running a few preliminary benchmarks with a single of MadGraph matrix element process (uu->h->uu tau tau). The results are below. Absolute numbers are not useful alone, but the difference between the various hardware platforms is a good way to estimate future performance comparison. What we use as reference when talking about the "speed-up" is the sequential execution on one core of our motherboards (processor E5-2650).

6.2. First results with classical CPUs

The hardware is a motherboard with 2 processors E5-2650, with a total of 16 cores, which support AVX SIMD extension (able to apply basic operations to 4 doubles in parallel). The number of processes has been voluntary limited to 16 processes per node in this benchmark because, for upper values, MEM-MPI doesn't take advantage of the hyper-threading (meaning that the application does not benefit of virtual cores waiting for resources, mainly, data coming from memory) .

In the column "16 cores (MPI)" of table 1, we see the results when we run 16 MPI processes, the same as we did with the MEM-MPI code. As with this previous code, we notice a very good scaling.

In the second column "16 cores (OCL)", we run a single MPI process which sees the motherboard as 16 OpenCL devices. The results are even better, which we interpret as the automatic OpenCL support for SIMD. So, our OpenCL translation already brings a big benefit with classical CPUs, when compared to the original code.

Table 1. Speedup with classical CPUs

	16 cores (MPI)	16 cores (OpenCL)
s/event	27.00	0.53
Speedup (versus 1 core)	**14.6**	**46.6**

OpenCL is not magic. We would probably have similar results with MPI and an SIMD modified version of GSL+MadGraph. Yet, with the same code, we can now also run on accelerators, as presented in the following.

6.3. First results with CUDA hardware

For this test the hardware is composed of 2 NVidia K20M accelerators. The first column recalls the result when using all cores of the motherboard.

In the first column "1xK20 (OCL)" of table 2, we see the performance with a single accelerator (we use one core from the motherboard to manage the job). In the second column "2xK20 (MPI-OCL)", we see the results with two such accelerators. We needed to use two MPI processes on the motherboard, because the NVidia OpenCL implementation does not support the use of multiple accelerators from a single main process. Again: we did not make any kind of profiling and optimization. There are probably sources of optimization to be found.

Given that the price of two such accelerators is comparable to the price of the motherboard, we obtain a raw speedup of about 3.

Table 2. Speedup with CUDA hardware.

	16 cores (**OpenCL**)	**1xK20** (OpenCL)	**2xK20 (MPI-OpenCL)**
s/event	0.53	0.32	0.18
Speedup	**46.6**	**77.1**	**140.5**

6.4. First results with MIC hardware

At last, we tried the same with 2 MIC co-processors, model Phi 5110P.

In the first column "1xPhi (OpenCL)" of table 3, one see the performance of a single device, which we found a little disappointing, as compared to previous NVidia accelerators. When the OpenCL translation is completed, we will push further the investigation.

In the second column, "2xPhi (OpenCL)", one notices a normal scaling when using two devices. Also, in the third column, we observe another benefit of OpenCL: the straightforward use of both the devices and the cores of the motherboard.

Table 3. Speedup with MIC hardware.

	1xPhi (**OpenCL**)	**2xPhi** (OpenCL)	16 cores+2xPhi (O**pen**CL)
s/event	1.31	0.63	0.33
Speedup	**18.9**	**38.9**	**74.2**

7. Extrapolation

The estimation for the Run 1 sequential analysis was about 4100 days. If we assume that the K20 speedup (~77), obtained for a single MadGraph process, will be the same with all MadGraph processes, and if we do not encounter some new multi-node load balancing problem, then we can hope to reprocess the whole Run 1 in 10 days, either with ~28 nodes of 16 cores such as the one used for the tests above (4100/15/10), or with ~6 K20s (4100/77/10). Lets' notice that we were rather pessimistic and did not take a few favorables factors into account:

- We do not use the remaining CPU cores together with the K20s (under work).
- We did not yet profile and optimize the code.
- Physicists think they can improve factorization in the matrix elements, without compromising on the physics, and gain another x5.

8. Conclusion and future work

In order to tackle the prohibitive consuming of CPU time, we built a MEM application in two steps. The first one is a MEM parallel implementation already used in production mode which can be deployed, thanks to MPI standard, everywhere from local computing clusters to High Performance Computing centers. It offers the whole processing of a set of events in a single job.

In addition, mixing the MPI and OpenCL standards, we intend to use GPUs at large scales. In this way, we built an OpenCL preliminary version of MEM, which takes only the most CPU intensive part of the code into account: the integration computation of the matrix elements with the MadGraph generated code for only one sub-process: uu->h->uu tau tau. The first results are very promising. Firstly, on CPUs (without GPUs), the OpenCL implementation offers, thanks to the OpenCL abstract model and the SIMD instructions generation by modern OpenCL compilers, a speed-up greater than 3. So the whole application will benefit of this speed-up without hardware specific optimisations. Secondly, on GPUs, the preliminary version is accelerated by a factor greater than 5 on a single NVidia K20 GPU (speed-up of 77 compared with a MEM sequential implementation running on one CPU core).

Combining OpenCL and MPI, we plan to make MEM computations tractable by accelerating the required 2-months processing on a 4-nodes platform (with 16 real cores each) for Run 1 reprocessing, to 10 days on a 3-nodes platform equipped with 2 NVidia K20 accelerator per node. Some work remains before reaching this goal, mainly: complete the MadGraph generator in OpenCL (translation is already done automatically in C99) and translate ROOT utilities (like TLorentzVector algebra) in OpenCL.

Acknowledgements
This work has been funded by the P2IO LabEx (ANR-10-LABX-0038) in the framework « Investissements d'Avenir » (ANR-11-IDEX-0003-01) managed by the French National Research Agency (ANR).

This work has also been partly funded by the grant ANR-13-JS05-0002-01 ("CHAMPS") managed by the French National Research Agency (ANR)

References

[1] Abazov V M and others 2004 A precision measurement of the mass of the top quark *NATURE* **429** 638–642.
[2] Schouten D, DeAbreu A and Stelzer B 2014 Accelerated Matrix Element Method with Parallel Computing *ArXiv e-prints*
[3] Weinzierl S 2000 Introduction to Monte Carlo methods *ArXiv e-prints*
[4] Whalley M R, Bourilkov D and Group R C The Les Houches accord PDFs (LHAPDF) and LHAGLUE arXiv:hep-ph/0508110.
[5] Lai H L, Guzzi M, Huston J, Li Z and Nadolsky P M and others 2010 New parton distributions for collider physics *Phys.Rev.* D82 074024. arXiv:1007.2241, doi:10.1103/PhysRevD.82.074024.
[6] Alwall J, Herquet M, Maltoni F, Mattelaer O and Stelzer T 2011 MadGraph 5: Going Beyond *JHEP 1106* 128. arXiv:1106.0522, doi:10.1007/JHEP06 128.
[7] Hagiwara K, Kanzaki J, Li Q, Okamura N and Stelzer T 2013 Fast computation of MadGraph amplitudes on graphics processing unit (GPU) *European Physical Journal* C 73 2608. arXiv:1305.0708, doi:10.1140/epjc/s10052-013-2608-2.
[8] Galassi M and others 2009 GNU Scientific Library Reference Manual *Network Theory Ltd* third edition edition
[9] Peter Lepage G P 1978 A New Algorithm for Adaptive Multidimensional Integration. *J.Comput.Phys.* 27:192
[10] Lepage G P VEGAS: An Adaptive Multi-dimensional Integration Program 1980 *Cornell preprint* CLNS 80-447.
[11] Kanzaki J 2011 Monte Carlo integration on GPU *European Physical Journal* C, 71:1559
[12] Brun R and Rademakers F 1997 ROOT - An Object Oriented Data Analysis Framework *Nucl. Inst. & Meth. in Phys. Res.* A, 389:81–86
[13] Stone J E Gohara D and Shi G 2010 Opencl: A parallel programming standard for heterogeneous computing systems *IEEE Des. Test* 12 (3) 66–73. doi:10.1109/MCSE.2010.69.
[14] Gropp W, Lusk E and Skjellum A 1994 Using MPI: portable parallel programming with the message-passing interface *MIT Press* Cambridge MA, USA.