Hybrid implementation of the VEGAS Monte-Carlo algorithm

Gilles Grasseau[1], *Stanislav Lisniak*[1], *David Chamont*[1]

[1]LLR, Laboratoire Leprince-Ringuet, Ecole polytechnique, 91128 Palaiseau, France

DOI: http://dx.doi.org/10.3204/DESY-PROC-2014-05/19

Cutting edge statistical methods involving Monte-Carlo integrations are commonly used in the LHC analysis chains. Dealing with high dimensionality integration, the computing time can be a bottleneck to deploy such statistical methods at large scales. In this paper we present the first bricks to build an HPC implementation of the well known VEGAS algorithm. Thanks to the open programming standards

OpenCL and MPI, we target to deploy such tool on cluster of nodes handling various hardware like GPGPU or 'many-core' computing accelerators.

1 Motivations

Multidimensional integration based on Monte-Carlo (MC) techniques [1] are widely used in High Energy Physics (HEP) and numerous other computing domains. In HEP, they naturally arise from the multidimensional probability densities or from the likelihoods often present in the analysis. Due to computing intensive integrations and large data sets containing millions of events, the situation is difficult for an analysis team if the processing of all samples exceeds 2-3 weeks (elapsed time).

Today, HPC programming requires dealing with computing accelerators

like GPGPU or 'many-core' processors, but also taking into account the development portability and the hardware heterogeneities with the use of open programming standards like OpenCL. Among the available MC algorithms (MISER, Markov Chain, etc.) the choice has been driven by the popularity and the efficiency of the method. The 'VEGAS' algorithm [2, 3] is frequently used in LHC analyses as it is accessible from the ROOT environment [4], while providing reasonably good performance.

The parallel implementation of VEGAS for computing accelerators presents no major obstacle [5], even though some technical difficulties occur when dealing with portability and heterogeneity, mainly due to the lack of libraries and development tools (like

performance analysis tools).

Combining MPI and OpenCL, we present a scalable distributed implementation. Performance will be shown on different platforms (NVidia K20, Intel Xeon Phi) but also on heterogeneous platform mixing CPUs, and different kinds of computing accelerator cards.

The presented work is a canvas to integrate various multidimensional

functions for different analysis processes. It is planned to integrate and exploit this implementation in the future analyses by the CMS (Compact Muon Solenoid) experiment at CERN.

2 Software design

2.1 Introduction to the VEGAS MC algorithm

The ROOT-based MC integration environment is popular within the High Energy Physics community. This environment actually provides an encapsulation of the GNU Scientific Library (GSL) MC integration functions [6]. GSL offers 3 kinds of MC algorithms : classical (or PLAIN), MISER, VEGAS. Since VEGAS converges more rapidly than the two other methods, and is widely used in the CMS collaboration data analysis, we will focus on this MC integration method.

A High-dimensional integral $I = \int d^n\vec{x}\, f(\vec{x})$ can be approximated by evaluating $f(\vec{x})$ at M points $\vec{x}$, drawn randomly in the domain Ω with the probability density $p(\vec{x})$:

$$\overline{I} = \overline{f_p} = \frac{V}{M}\sum_{\vec{x}} \frac{f(\vec{x})}{p(\vec{x})},\ V \text{ is the volume of } \Omega,$$

with its estimated variance $\overline{\sigma}^2$:

$$\overline{\sigma}^2 = \frac{\overline{f_p^2} - (\overline{f_p})^2}{M-1} \simeq \sigma^2,\ \text{where } \overline{f_p^2} = \frac{V}{M}\sum_{\vec{x}}(\frac{f(\vec{x})}{p(\vec{x})})^2.$$

The classical MC integration method (PLAIN in GSL library) uses a uniform density probability $p(\vec{x}) = cst$ which ensures the process' convergence: $\lim_{M\to+\infty} \overline{f}_p \longrightarrow I$. Nevertheless, the convergence is slow and requires a great number of points M to obtain a good approximation of I.

In the VEGAS algorithm, two main ideas are used to reduce the variance and, as a result, accelerate the convergence of the estimated integral $\overline{I}$.

1. *Importance sampling*: the variance tends to zero if the probability density has the form $p(\vec{x}) \propto |f(\vec{x})|$ (quick justification for $f(\vec{x}) > 0$, $\frac{f(\vec{x})}{p(\vec{x})} = cst, \Rightarrow \sigma^2 = 0$, see [1] for more details) . In other words, this means that the function sampling must be concentrated on the largest magnitudes of the function $f(\vec{x})$. Starting with $p(\vec{x})$ uniform, $p(\vec{x})$ gradually approximates $\frac{|f(\vec{x})|}{cst}$, thanks to the contributions of the different function evaluations.

2. *Stratified sampling*: this other strategy samples the highest variance domains in Ω, then subdivides it in sub-domains to decrease the local variance and thus the global variance of $|f(\vec{x})|$. With this strategy, the system converges with a sub-domain partition of Ω which minimizes the variance σ^2. In GSL implementation, the probability distribution $p(\vec{x})$ is updated with the local sub-domain variance $\sigma^2(\vec{x})$; these sub-domains are called *boxes* and are used to discretize $p(\vec{x})$.

Depending on the number of points to evaluate (set by user), VEGAS chooses *Importance sampling* or *Stratified Sampling* according to the sampling density of the domain.

2.2 Parallelism and OpenCL considerations

The VEGAS algorithm can be sketched with 3 main embedded loops as shown in Fig.: 1. The internal loop evaluates the function $f(\vec{x})$ for M random points $\vec{x}$, according to the probability distribution $p_{k-1}(\vec{x})$. The accumulated values of $f(\vec{x})$ give the estimated integral $\overline{I_k}$. In the same loop, the estimated variance $\overline{\sigma_k}^2$, as well as the new probability distribution $p_k(\vec{x})$, are updated. Several evaluations of the $\overline{I_k}$ integral are performed in the intermediate loop, in order to compute the Chi-square χ^2 by degree of freedom, thus determining the consistency of the sampling. Finally, the outer-loop is used to control the integral convergence $(\overline{I}, \overline{\sigma}^2, \chi^2)$.

From the parallelization point of view, the most computing intensive loop, i.e. the internal one, must be spread among different computing units, handling the shared variables $\overline{I_k}$, $\overline{\sigma_k}^2$, $p_k(\vec{x})$ with care. It is well-known that opening a *parallel region* (with the `OpenMP` formalism) on the internal loop is much less efficient than opening a *parallel region* on the outer loop. In the same way, `OpenCL` (or `CUDA`) *kernels* must be as large as possible to avoid substantial overhead time to launch kernels and unnecessary work to split the initial kernel in several kernels. It is worth mentioning, in our algorithm, the embedded loops must be split in two *kernels* at the reduction step $(\overline{I_k}, \ldots)$ to synchronize the global memory between the different *work-groups*. As a result, split kernels will generate unexpected overhead time to launch and synchronize : *number_convergence_iterations* $\times$ *number_internal_iterations* $\times$ 2 *kernels*.

```
Compute: $I = \int_\Omega f(\vec{x})\, d^n x$, on
a $d$-dimensional integration domain $\Omega$
Loop until convergence $(\overline{\sigma}^2, \chi^2)$
    Loop internal iteration $k$
        $p_k(\vec{x}) \leftarrow 0$;
        Loop over $N$ points $\vec{x} \in \Omega$
            $\vec{x} \leftarrow$ rand();
            $\bar{f} \leftarrow \bar{f} + f(\vec{x})$;
            update $\overline{I_k}$, $\overline{\sigma_k}^2$, $p_k(\vec{x})$;
        End Loop
        update $\overline{I}, \overline{\sigma}^2, \chi^2$;
    End Loop
End Loop
```

Figure 1: the VEGAS algorithm. The integral $\overline{I}$ of $f(\vec{x})$ is evaluated on the d-dimensional domain Ω. The standard deviation $\overline{\sigma}^2$ and the χ^2 can be used as convergence criteria. $p_k(\vec{x})$ is the probability distribution discretized on a grid.

Writing and managing a single *kernel* which takes into account all steps of the VEGAS algorithm, presents no difficulty. It only requires that each computing element evaluates several points contributing to the integral. In addition it substantially increases the computing load per computing unit. However, the only way to synchronize the shared variables is to perform the computations in a single *work-group* (or *block* in `CUDA`). Although this solution works well on `Intel Xeon Phi` (thanks to `OpenCL` 1.2), it does not work properly on `NVidia` hardware for 2 reasons:

- The *work-group* size is limited (hardware limit, generally 1024).
- Two `OpenCL` *kernels* cannot be run simultaneously on `NVidia GPGPUs` (the situation is even more dramatic, two `OpenCL` *kernels* cannot be run simultaneously on two different `GPGPUs` in the same process).

With such limitation on `NVidia OpenCL` driver we were constrained to split the kernel and lose efficiency.

2.3 `OpenCL` and `MPI` event dispatchers

The expected speed-up factor of a single accelerator card will not be sufficient to minimize user waiting time when dealing with data-sets containing each 10^6 events to process. The CMS analysis team cannot afford to wait for long processing chains to end (each chain needs several weeks to be processed). With this requirement, it is mandatory to use several accelerator cards simultaneously, and even more, using several nodes themselves handling several accelerator cards to build a high performance computing application.

To gather the computing power available, a two-level event dispatcher has been implemented: the `OpenCL` *event dispatcher* dealing with the distribution of events among several devices (i.e. several accelerator cards and `CPUs`) and the `MPI` *event dispatcher* to distribute events among the computation nodes.

The `OpenCL` *event dispatcher* takes advantage of the `OpenCL` abstract model, in which *queues* manage *kernels* to be executed on the hosted *devices*. In this way, we can benefit directly of the `CPUs` power without adding a multi-thread programming paradigm to build a hybrid application.

The `OpenCL` dispatcher feeds the device with an event to be processed, as soon as a queue is free. All copies from host node to accelerator cards and all kernels are launched asynchronously, with a dependence graph built thanks to `OpenCL` *events*. The computation of the integral is considered to be finished when the copy of the result - from the *device* to the host - is completed.

The second dispatcher level, called `MPI` *event dispatcher* aggregates computing power form different nodes by distributing sets of events among several MPI processes (generally one `MPI` process per node). In the same way as the `OpenCL` dispatcher, the *master* `MPI` process sends a set of events to be processed, as soon as a *worker* `MPI` process is idle. Then, the `MPI` worker (or `MPI` master) delegates the treatment of its event set to the `OpenCL` dispatcher.

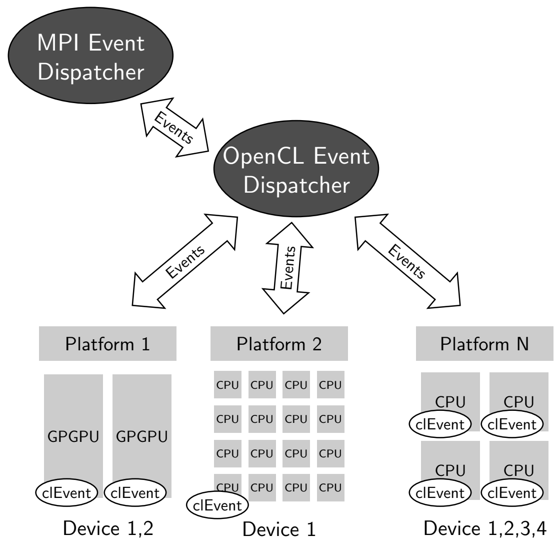

Figure 2: `OpenCL` dispatcher: the events are dispatched to different device queues. The different kernels to be processed are run asynchronously. The synchronization is performed thanks to `OpenCL` *events*

3 Performance studies

3.1 Benchmark environment

As far as the integration is concerned, we choose for this benchmark the special function *sinus integral* to test the adaptive scheme for an oscillating function, whose result is known:

$$I = \int_0^{2\pi} \prod_{i=0}^{n} \frac{\sin(x_i)}{x_i} d^n x \simeq 5.7360, \text{ with } n = 5.$$

Often, the integration is done iteratively to control the convergence towards acceptable values of the standard deviation σ and chi-square χ^2. In this test case, we want to

Product	Version/Options
Compiler	icc-13.0.1 / `-O3`
`MPI`	`OpenMPI` 1.6.5
`OpenCL`	`Intel` 1.2
`OpenCL`	`NVidia` 1.1

Table 1: Compiler and library versions.

perform a fixed number of iterations, evaluating 5×10^5 points per integration. For these parameters, VEGAS draws 2 points per box (discretized volume element of $p_k(\vec{x})$), this means that only 12 boxes are used per dimension for the grid integration domain ($2 \times 12^5 = 497664$).

Three kinds of clusters are available on our platform, called `GridCL`, for benchmarking:

- Two nodes connected with an `InfiniBand` link, each hosting 2 `Intel Xeon E5-2650` processors (8 cores for each processor rated at 2.0 GHz). In addition, each node hosts 2 `NVidia K20M GPGPU` cards.
- Same as above concerning the node characteristics. Each node hosts 2 `Intel Xeon Phi 5110P` accelerator cards.
- The last node based on two `Intel Xeon E5-2650 v2` (Ivy Bridge) processors (8 cores for each processor rated at 2.6 GHz) hosts 6 `NVidia Titan GPU` cards.

The software configuration used to build the VEGAS hybrid application is presented in Tab. 1.

3.2 Results

As shown in Tab. 2, the `OpenCL` implementation of VEGAS presents very good performance on `CPUs` (column 2). This improvement is not only accountable to `OpenCL` parallelization, but also to the `Intel OpenCL` implementation which both *parallelizes and vectorizes* the *kernels*, as `Intel` was claiming [7] (a speed-up of 18.5 is obtained with 32 `MPI` instantiations of the `GSL` implementation - column 1).

Speed-up	`GSL` (1)	`OCL` (2)	`OCL` (3)	`OCL` (4)	`OCL` (5)	`MPI` (6)
# `CPUs`	32	32	-	-	32	-
# accelerators	-	-	1	2	2	2/6
`K20M` node	18.5	50.0	56.0	56.0	94.5	110
`Xeon Phi` node	18.5	50.0	27.3	54.1	80.3	-
`Titan` node	18.1	39.1	55.9	55.6	95.3	328

Table 2: Speed-up values obtained with different configurations (the reference time to calculate the speed-up values is the computing time obtained with the `GSL` library): (1) using `GSL` library executed on 32 processors (in fact 16 physical processors, 32 with hyper-threading), (2) `OpenCL` version on the 32 processors, (3) `OpenCL` version on a single accelerator card, (4) `OpenCL` with 2 accelerator cards, (5) `OpenCL` with all devices including `CPUs`, (6) 2 `MPI` processes (6 `MPI` processes on the `Titan` node), each handling one accelerator card (with `OpenCL`).

The acceleration obtained with the `NVidia K20M` card seems to be good, but compared with `CPUs` performance, the gain is not excellent. As announced in paragraph section 2.2, no gain is obtained with two `K20M` devices (column 4), assuming that the `NVidia OpenCL` driver does not handle simultaneously several cards properly. Fortunately, we can bypass the lack of functionality, by launching 2 `MPI` processes each handling one device (see column 6).

Concerning `Intel Xeon Phi`, the speed-up shows that we do not use theses accelerators optimally. We will rely on the `Intel` performance analysis software `VTune` to highlight the bottleneck when we will start to optimize our *kernels*. However, the `Intel OpenCL` driver deals with the 2 cards simultaneously (column 4). Considering all computing devices, good overall performance is obtained (column 5), thanks to the efficiency of `OpenCL CPUs`.

Benchmarking the computing node holding 6 `NVidia Titan` cards, the performance profile is the same as above : moderated speed-up for one card (speed-up = 56). With all 6 cards the speed up value is identical due to the `NVidia` driver issue, while the speed-up reaches 328 with 6 MPI processes.

The `OpenCL` and `MPI` event dispatchers provide good efficiency (see columns 5 and 6) even if the efficiency decreases when using several nodes. A time sampling has been added into the application to trace the event distribution, the overheads and the idle zones for future optimization work.

4 Conclusion

This preliminary version of our high performance `MC` integration implementation, based on `OpenCL` and `MPI`, already offers good efficiency on `CPUs` and `OpenCL` event dispatchers. The application however still needs several improvements to extract more computing power from accelerator cards, and requires tuning load-balancing parameters to efficiently run on `MPI` event dispatchers. Already substantial speed-up (> 300 - compared with sequential integrations based on `GSL` library) has been reached on the `GridCL` node hosting 6 `NVidia Titan` cards.

Before improving the application with all potential identified optimizations, we are going to focus our activity on a real application currently designed by the `LLR CMS` analysis team. Based on VEGAS integrations, the matrix-element methods (`MEM`) [8, 9], is a well known powerful approach in particle physics to extract maximal information from the events. Knowing that `MEM` require a huge computing power (processing one event can take $60s$), we aim to provide a drastic speed-up to the `MEM` processing chain.

Acknowledgments

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).

References

[1] Stefan Weinzierl. Introduction to Monte Carlo methods. *ArXiv e-prints*, 2000.

[2] G. Peter Lepage. A New Algorithm for Adaptive Multidimensional Integration. *J.Comput.Phys.*, 27:192, 1978.

[3] G. Peter Lepage. VEGAS: An Adaptive Multi-dimensional Integration Program. 1980. Cornell preprint CLNS 80-447.

[4] Rene Brun and Fons Rademakers. ROOT - An Object Oriented Data Analysis Framework. *Nucl. Inst. & Meth. in Phys. Res. A*, 389:81–86, 1997. `http://root.cern.ch/`.

[5] J. Kanzaki. Monte Carlo integration on GPU. *European Physical Journal C*, 71:1559, February 2011.

[6] M. Galassi et al. *GNU Scientific Library Reference Manual.* Network Theory Ltd, third edition edition, 2009. `http://www.gnu.org/software/gsl/`.

[7] Intel developper note. Writing Optimal OpenCL Code with Intel OpenCL SDK. page 10, 2011. `https://software.intel.com`.

[8] Abazov VM et al. A precision measurement of the mass of the top quark. *NATURE*, 429:638–642, 2004.

[9] D. Schouten, A. DeAbreu, and B. Stelzer. Accelerated Matrix Element Method with Parallel Computing. *ArXiv e-prints*, July 2014.