CERN Summer Student Programme 2014

Gábor Bíró

Investigation of OpenCL support in the VecGeom geometry package

---

Supervisor:

Dr. Mihály Novák
CERN PH-SFT

CERN, 2014
30.June -- 26.September

Abstract. High energy physics always needed a significant amount of computational resources, and with the upgrades of the experiments this need is still growing. Though the computational capacity of the hardwares are increasing as well, since the increasing of the clock frequency of the CPUs stopped a few years ago it needs more effort from the sofware side to exploit all the capabilities. The goal of the VecGeom package is to offer a geometry package that can take advantage most of the modern computing technologies. This report describes the benefits and handicaps of the implementation of the OpenCL support in VecGeom, bearing in mind that it has to be as generic as possible and it has to fit in the already existing generic templated structure.

1 Introduction

Geant4 (GEometry ANd Tracking) is the simulation framework that is used by the main LHC experiments and many others around the world. It's main purpose is to simulate the passage of particles through matter, using Monte Carlo methods. It is a worldwide collaboration and it has many application areas (for example space science or medicine), not just particle physics. The development of the simulation package started in 1994, and since 2012 there are studies for the possible successor, the Geant Vector Prototype project[1].

Since in a typical detector simulation the geometry calculations need a huge amount of computational resources (40-50% of the overall time spent with geometry calculations), it is very important to have an effective geometry code that exploits most of the available computational capacity.

The VecGeom (Vectorized Geomety) is the geometry package that was started to develop in the context of the Geant Vector Prototype project in 2013. When it will be finished, it will be a standalone library that provides a GPU support via the CUDA compiler as well[2].

In the geometry calculations the already vectorized code executes single instructions on multiple data (SIMD) in parallel, so a good speedup is already achieved on CPUs that support vector units, compared to the scalar handling of the same data elements.

Nowadays the GPGPU (general-purpose computing on graphics processing units) is widely used in cases where highly parallel intensive computations are needed[3]. The computations with GPUs are effective in cases where the size of the input to be processed is big (compared to the CPUs), and since in geometric calculations all the particles are independent, it is reasonable to prepare the code for such cases and implement the support for GPUs.

A CUDA (Compute Unified Device Architecture) backend already exists in VecGeom. Because CUDA

supports only NVIDIA devices, during my stay at CERN I had to investigate how could we increase the portability and the performance on multiple architectures, with the usage of the OpenCL.

2 Project goals

OpenCL (Open Computing Language) is an open standard with the developer can make very effectively massively parallelised code. It is maintained by Khronos Group, which is a non-profit consortium with the purpose of ,,creating open standards for the authoring and acceleration of parallel computing, graphics, dynamic media, computer vision and sensor processing on a wide variety of platforms and devices''[4]. It has about 100 members (including Apple, who was initially the developer of OpenCL, NVIDIA and AMD) and currently it handles 15 standards.

The 1.0 version of OpenCL was released in the middle of 2009 and based on the C programming language. Since the version 1.2 there is a C++ Wrapper for the device side codes[5, 6, 7]. The latest released version is 2.0[8].

My project was to study wether we can achieve OpenCL support in VecGeom without deverging from the generic C++ coding. Because of the OpenCL kernel language based on C, I did experiments with the AMD's OpenCL C++ Kernel Language Extension[9], with that one can develop OpenCL kernels also with C++ features, like templates or kernel overloading. However, there are still many unsupported features, that made difficult to unify the OpenCL execution model with the generic structure of VecGeom. In order to understand the results of the investigation, in the following I give a short brief about the structure of an OpenCL application.

2.1 Hierarchy of models

In OpenCL the following models are used to describe the workflow:

Platform model. There are two basic elements: the host, which is connected to at least one device. The device can be any computing device that has an OpenCL implementation (GPUs and CPUs as well). The device is than divided to smaller computing units. The most basic elements are the processing elements: they can execute instructions as SIMD units and can be grouped into computing units.

Execution model. OpenCL is a multisource standard, which means that the host program defines the context, queries the available devices, manages the memory objects and submit the kernels, that run on the device(s). After a kernel is submitted, an index space is defined, and an instance of the kernel (a work-item) will be executed on each point of the index space. Each work-item has a global ID and an equivalent combination of local ID and and work-group ID. In order to create a continuous workflow, the two main tasks during the development of an OpenCL program are to define the index space properly and to ensure that all the availbale devies are utilized effectively.

(maybe more description here about contexts, commandqueues, memory objects and other stuff?)

Memory model. In OpenCL the developer has a lot of freedom to determine the relationship of the individual memory objects. Each memory region has its own allocation and memory access capability. This also implies that the developer has to use barrier objects and syncronizing commands.

For further details see [6, 8].

2.2 Experiences.

First I had to get familiar with the generic structure of VecGeom. It uses trait classes to determine the workflow, depending on the used architecture. (maybe figure and/or snippets) Since CUDA can implement all the C++ commands, with clever designing and macro usage there is no need to use a different code for CUDA: with the help of `typedef` commands the behaviour can be determined in compile time. It is a very important fact, because the code maintenance is much feasible if there is only one single source code for all of the supported architectures.

I used this principle as a baseline and I tried to integrate the OpenCL code to the already existing code. As a working area I used the methods of the box shape, for example the `DistanceToOut`, which computes a points minimal distance that is needed to reach the border of a box (of course the point is inside the box). First I created a header file that contained most of the functions that is needed to submit a kernel. My goal was to implement a simple and reusable way to launching kernels, that can be used in the existing implementations of the shapes. The basic idea was that in the main code one has to write only short commands like

```
BoxMethods<double>::DistanceToOut( ...geometry specific parameters... )
```

and all the OpenCL specific commands remain hidden.

Though the host-side code can be implemented in a relative easy way, soon I realized that making kernel code from the existing methods (without creating too much extra code) are much more difficult and have more issues. The first and most problematic is that even the AMD's C++ language extension doesn't allowe me to include STL and standard C++ headers in the kernel files. For this first it seemed as an obvious solution to cut all of the troubled parts with macros and create a suitable environment for the kernel codes, very similar to the case of CUDA, but since the structure of an OpenCL kernel is quite different than the templated vectorized code (see Table 1), soon I had almost a double amount of code, so for the time being I rejected this solution.

```
distance = kInfinity;
Vector3D<Float_t> inverseDirection =
Vector3D<Float_t>(
1. / (direction[0] + kTiny),
1. / (direction[1] + kTiny),
1. / (direction[2] + kTiny));
Vector3D<Float_t> distances = Vector3D<Float_t>(
(dimensions[0] - point[0]) * inverseDirection[0],
(dimensions[1] - point[1]) * inverseDirection[1],
(dimensions[2] - point[2]) * inverseDirection[2]);
MaskedAssign(direction[0] < 0,
(-dimensions[0] - point[0]) * inverseDirection[0],
&distances[0]);
MaskedAssign(direction[1] < 0,
(-dimensions[1] - point[1]) * inverseDirection[1],
&distances[1]);
MaskedAssign(direction[2] < 0,
(-dimensions[2] - point[2]) * inverseDirection[2],
&distances[2]);
distance = distances[0];
MaskedAssign(distances[1] < distance, distances[1],
&distance);
MaskedAssign(distances[2] < distance, distances[2],
&distance);
```

```
int i = get_global_id(0);
double distX = (( copysign(dimensions[0], dirX[i]) -
posX[i] ) * ( 1. / ( dirX[i]+1e-30 )));
double distY = (( copysign(dimensions[1], dirY[i]) -
posY[i] ) * ( 1. / ( dirY[i]+1e-30 )));
double distZ = (( copysign(dimensions[2], dirZ[i]) -
posZ[i] ) * ( 1. / ( dirZ[i]+1e-30 )));
distances[i] = fmin( distX, fmin (distY, distZ));
```

Table 1: The core elements of the same method in the existing vectorized code (left) and in an OpenCL kernel (right)

Instead of trying to minimize the amount of new code, I decided to ,,rewrite'' the already existing ones and try to make at least one effective OpenCL supported method. I find this way much more feasible: with the freedom of adding more new codes most of the problems can be solved. In addition, I find this way is more beneficial for the kernel functions as well, because they can be more effectively fine tuned for the advantage of OpenCL.

Thought the performance is still an issue and contrary to the goals this method means more additional code, the support of OpenCL could work with the re-implementation of the methods.

2.3 SYCL

Since one of the main apsects is the code maintenance, I would like to mention SYCL[10]: when it will be finished, it will be an abstraction layer over OpenCL, developed by the Khronos member Codeplay. The provisional specification is available since the first quarter of 2014.

My opinion is that if we want to keep as many from the existing structure as possible, the OpenCL support in VecGeom will be reasonable - according to the specification - with the usage of SYCL.

3 Conclusions

During the summer I showed that in order to create the OpenCL support for VecGeom it is necessary to modify (or even extend) the existing generic templated code, because the current version of OpenCL doesn't fit well in the current structure. However, with some effort the OpenCL supported geometry could be very effective on CPUs and GPUs as well and it could provide at least as good performance as the vectorized code combined with the CUDA backend.

I also managed to make a few tests with the new beta version of the SYCL compiler, with that one will be able to write OpenCL code based on C++. Regarding that this is a beta version compiler still under development, much more investigation is needed - according to the preliminary specification, when it will be completed, with the usage of SYCL the OpenCL support could be done with much less code modification.

4 Acknowledgements

Firstly I would like to thank my supervisor, Mihály Novák for the fruitful conversations and for the improvised lessons, Sandro Wenzel for the guidance he gave me during the last months and Johannes de Fine Licht for being so patient with me when I had a lot of questions. Last but not least I would like to thank the MTA Wigner FK - GPU-Labor in Budapest for the opportunity they gave me to test some of my codes.

During my stay at CERN I received a great amount of knowledge from various fields. My ablilties like templated metaprogramming, GPGPU programming, CMake usage improved greatly and among others I learned a lot about particle transport simulations.

References

[1] J. Apostolakis, R. Brun, F. Carminati, A. Gheata, and S. Wenzel, ''A concurrent vector-based steering framework for particle transport,'' *ACAT2013 Proceeding*, 2013.

[2] J. Apostolakis, R. Brun, F. Carminati, A. Gheata, and S. Wenzel, ''Vectorising the detector geometry to optimize particle transport,'' *arXiv:1312.0816v1*, 2013.

[3] B. Oancea, A. Tudorel, and D. Raluca Mariana, ''GPGPU computing,'' *arXiv:1408.6923*, 2012.

[4] Khronos Group.

[5] Khronos OpenCL Working Group, *OpenCL Introduction*, 2014.

[6] Khronos OpenCL Working Group, *The OpenCL 1.2 Specification*, 2012.

[7] B. R. Gaster and L. Howes, *The OpenCL C++ Wrapper API*, 2012. Version 1.2.6.

[8] N. Trevett, *OpenCL 2.0 Specification*, 2013.

[9] O. Rosenberg, B. R. Gaster, B. Zheng, and I. Lipov, *OpenCL Static C++ Kernel Language Extension*, 2011.

[10] Khronos OpenCL Working Group, *SYCL Provisional Specification*, 03 2014.