# Pyopencl in a non-root environment

Hinrichsen, Jesse

10. März 2016

**Abstract**

This is a small guide to install and use OpenCL[1] for Nvidia[2] GPU's in a non-root environment under Linux[3] using the python bindings pyopencl[4].The primary goal is, to execute GPU enhanced computations on a remote cluster. The second section gives an idea how OpenCL performs compared to a CPU based on an example

## Inhaltsverzeichnis



[1] https://www.khronos.org/opencl/
[2] http://www.nvidia.com/content/global/global.php
[3] In our case we use Ubuntu 14.04.4 LTS
[4] https://documen.tician.de/pyopencl/

# 1 Installation

This shall be a rough overview off the important parts during the setup.

## 1.1 Prerequisites

Due to the fact that we primarily want to run code on an existing cluster, we assume that the following packages are already available.

- Python[5]
- CUDA Toolkit[6]
- ICD file[7]: This refers to a file in `/etc/OpenCL/vendors` called `nvidia.icd` that contains only the name of the driver shared object. In this case: `libnvidia-opencl.so.1`

## 1.2 Virtualenv

To use virtualenv[8] follow the instructions on `https://virtualenv.readthedocs.org/en/latest/installation.html` under "To use locally from source:".

```
$ python virtualenv.py gpupy
$ source gpupy/bin/activate
```

This will create a folder gpupy and activate the virtualenv.

## 1.3 PyOpenCL

We will use pip to install pyopencl and all dependencies. In order to compile successfully we need to set the following environment variables.

- Path for C and C++ compiler to look for header files

```
$ export C_INCLUDE_PATH=$C_INCLUDE_PATH:/path/to/cuda/include
$ export CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH:/path/to/cuda/include
```

- Path to libraries[9]

```
$ export LIBRARY_PATH=$LIBRARY_PATH:/path/to/cuda/lib
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/cuda/lib
```

- Path to executables

```
$ export PATH=$PATH:/path/to/cuda/bin
```

It is advised to set additional CUDA relevant variables like `CUDA_HOME`.
Now running

```
$ pip install pyopencl
```

[5] On most systems a Python version comes natively. Since we will proceed in a virtualenv there is no version requirenment

[6] `https://developer.nvidia.com/cuda-toolkit`

[7] Normally the path, where to search for the icd files can be defined via OPENCL_VENDOR_PATH. Unfortunately the ICD loaders of Nvidia can't handle that environment variable. Alternatively it is possible to install a custom ICD Loader. For example: | http://manpages.ubuntu.com/manpages/trusty/man7/libOpenCL.7.html

[8] `https://virtualenv.readthedocs.org/en/latest/`

[9] `libOpenCL.so` and `libOpenCL.so.1` should exists. `libOpenCL.so.1` can be just a symlink to `libOpenCL.so`

will install the newest pyopencl version and all dependencies.
To specifically choose a version use for example `pyopencl==2015.1`[10].
To use additional compiler flags use the `--global-option` with `build_ext` of pip. For example, if you need OpenCL-OpenGL interoperation write

```
$ pip install --global-option=build_ext --global-option="-DHAVE_GL=1" pyopencl
```

# 2 Computing example

On `https://github.com/Jesse-jApps/gpu-computing` you can find a script to get some informations about the available devices on your machine. Furthermore we want to illustrate the potential of calculations on the GPU on a small example. You can find all source code in the above given url as well. This is a rather uncommon example for GPU computation, but easily demonstrate the advantages of parallel computing in physics.

In physics, numerical integration plays a very important role. While solving a single ODE on the CPU can be done very fast, we are often confronted with numerous parameters which can be set. As a consequence we might want to solve the same ODE for many different parameter sets and display the results for example in a field plot[11]. For a field plot with 100 times 100 points, we need to integrate 10,000 ODE's.

This can be done now in parallel on the GPU. In our example we take the Hamiltonian $H_0$ of a Josephson Junction, solve the eigenvalue-problem in k-space and integrate the time-dependant Schrödinger-Equation for the groundstate and the Hamiltonian $H = H_0 + H_{dr}$ where $H_{dr}$ is time-dependant. We will omit the physical details of the model and just compare performance. The used code is very rudimental as well, allowing many optimizations.

## 2.1 Increase of dimensions

One performance crucial parameter is the dimension of the Hilbertspace. Figure 1 shows the behavior of execution time needed for different hilbertspace sizes. While we chose a fixed grid size of 4 times 4 systems, the execution time for the OpenCL kernel is, for small dimensions, more than 5 times smaller. With increasing dimension the Numpy calculation stays at a fixed execution time, while the OpenCL execution time rapidly grows. This is due to matrix operations, which are optimized in Numpy especially for sparse matrices. If we would optimize the OpenCL kernel for sparse matrices as well, we would expect a much slower increase of execution time.

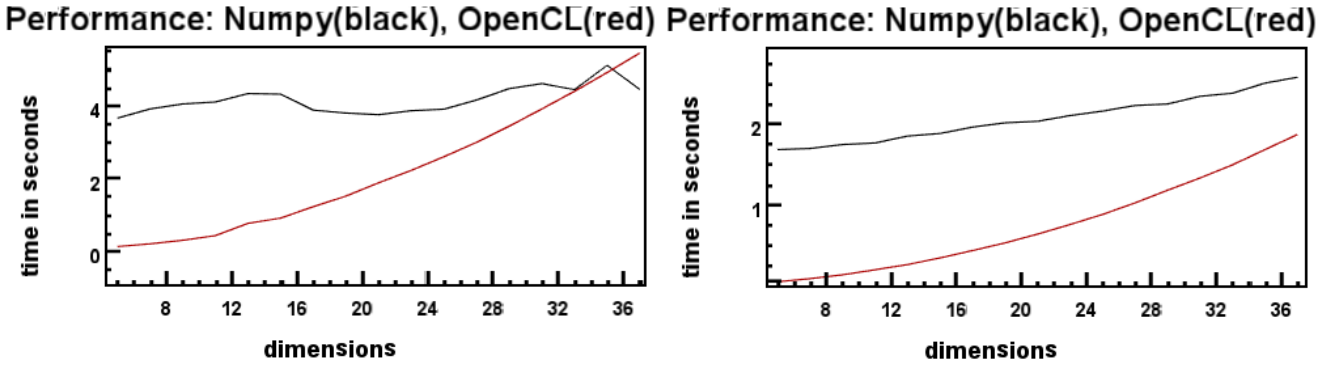


**Abbildung 1:** Execution time for 16 system over size of hilbertspace. Each system is integrated 1000 times with a stepwidth of $h = 0.005$ using Runge-Kutta-3. Left: MacBook Pro (2,4 GHz Intel Core i7-2760QM, AMD Radeon HD 6770M), left: Lenovo U31 (2,4 GHz Intel Core i7-5500U, Nvidia GK208M Geforce 920M)

[10] While 2015.1 still uses float2, double2, etc. to emulate complex numbers, when using `#include<pyopencl-complex.h>`, 2015.2 and later will use custom structs. Thereby already written code in 2015.1 will most likely not work anymore in 2015.2 and later.
[11] Due to the fact, that the calculations are made on the GPU, it is very easy to directly show the results using OpenGL buffers.

## 2.2 Increase of grid size

With an increasing number of systems, the execution time of non-parallel computations will increase in the same manner. Parallel kernels in contrast will stay at the same execution time independent of the number of systems until a certain hardware limit. Figure 3 show the increase of execution time for different numbers of systems.

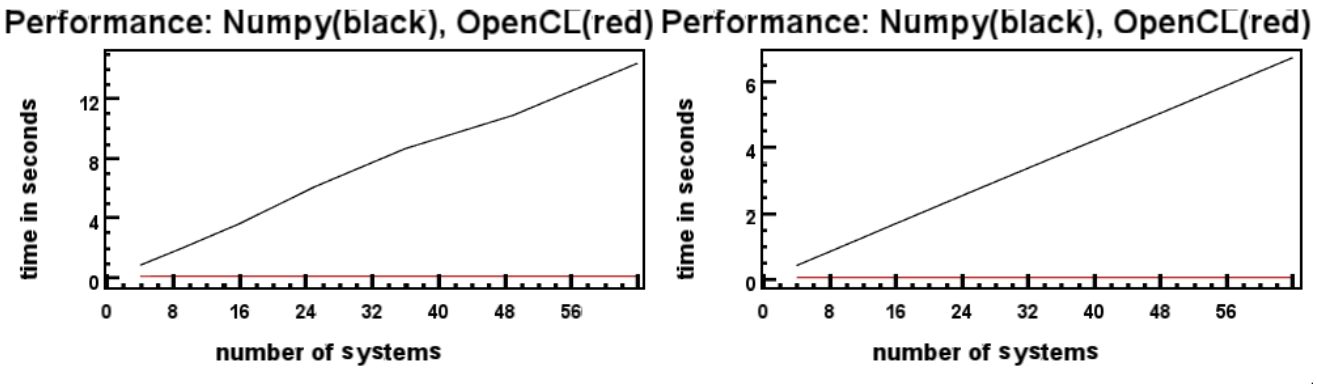


**Abbildung 2:** Execution time for different number of systems at a dimension of 5. Each system is integrated 1000 times with a stepwidth of $h = 0.005$ using Runge-Kutta-3. Left: MacBook Pro (2,4 GHz Intel Core i7-2760QM, AMD Radeon HD 6770M), left: Lenovo U31 (2,4 GHz Intel Core i7-5500U, Nvidia GK208M Geforce 920M)

## 2.3 Different graphic devices

Additionally here is an example how different GPU's compare. We compare the execution time for different amount of systems. Each system again evaluates the schrödinger equation for a state with 11 entries. Figure shows the result for AMD Radeon HD 6770M (Mac), Nvidia GK208M Geforce 920M (Lenovo), Nvidia Tesla K20m and Tesla M2050.

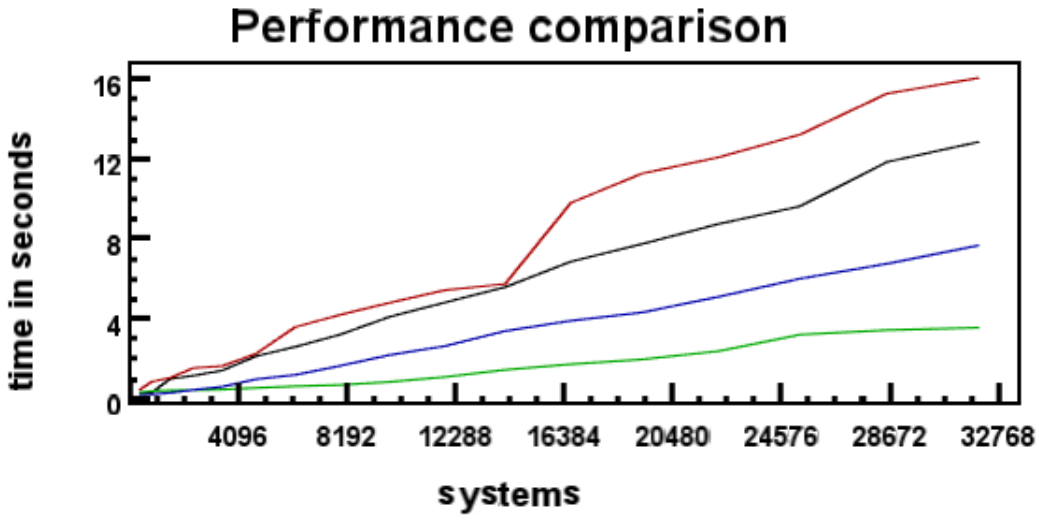


**Abbildung 3:** Execution time for different number of systems at a dimension of 11. Each system is integrated 1000 times with a stepwidth of $h = 0.005$ using Runge-Kutta-3. Red: MacBook Pro (AMD Radeon HD 6770M), black: Lenovo U31 (Nvidia GK208M Geforce 920M), blue: Nvidia Tesla M2050, green: Nvidia Tesla K20m

At a size of 36,100 systems calculated parallel we use the AMD Radeon HD 6770M as a reference. Therefore the Nvidia GK208M Geforce 920M needs 80% ,the Nvidia Tesla M2050 49% and the Nvidia Tesla K20m 25% for the calculation.

# 3 Conclusion

We conclude that as soon as a calculation can be split into independent smaller calculations, execution time on a GPU is significantly faster than on a usual CPU.
Additional use of local memory with a bus-transfer at least 2 times faster as bus-transfer to a global memory, will probably improve the execution time as well. If optimized algorithms are used, e.g. sparse matrix multiplication, and unnecessary data transfer is eliminated, we can expect to even better execution times at factor of 2 or 3.
While OpenCL is a vendor free, open source solution, CUDA will still be faster on Nvidia devices.