HPL tutorial

Moisés Viñas Buceta
Universidade da Coruña

December 11, 2014

1 Installation

HPL can be downloaded from `http://hpl.des.udc.es`. We will download it using the command

```
$   wget http://hpl.des.udc.es/page2__assets/hpl__20131014b.tar.gz
```

Now decompress the library using the command

```
$   tar -xzvf hpl__20131014b.tar.gz
```

At this point, we can compile the library and launch some tests distributed with the library. First, to compile the library, we need to generate the Makefile of the library. So we put:

```
$   cd hpl
$   ./configure
```

Finally, we compile the library [1]:

```
$   module load cuda55/toolkit/5.5.22
$   export CPLUS__INCLUDE__PATH=$CPLUS__INCLUDE__PATH:/cm/shared/package/amd-app-sdk/2.7/include
$   cd src
$   make
```

Now we are ready to compile and execute our tests.

[1]HPL needs a C++ compiler that supports C++11 standard (GNU g++ >= version 4.6.0). In DAS4 cluster, you can do `module load gcc/4.8.0` to use this compiler.

2 Compile and execute tests

HPL has available several tests (in /tests folder) to help the user to start using the library. For example, you can compile and execute the `HPLDeviceQuery` test. To launch it on a node with a GPU, you will need a job file to launch the program with the qsub command. You can use `job_gpu.job` of the web `http://gac.udc.es/~moises/courses_en.html`[2]:

```
$   make HPLDeviceQuery
$   wget http://gac.udc.es/~moises/ficheros/tudelft/job_gpu.job
$   qsub job_gpu.job
$   Your job 3394769 ("job_gpu.job") has been submitted
$   cat job_gpu.job.o3394769
-----------------------------------------
GPU number 0
Platform 3 DeviceType 1 Device number 0
Device name.......................... GeForce GTX 480
Global memory size (in bytes)........ 1610285056 unified=0
Local memory size (in bytes)......... 49152 dedicated=1
Maximum buffer size (in bytes)....... 402571264
Global memory cache size (in bytes).. 245760
Global memory cache type............. read-write
Compute units........................ 15
Domain dimensions supported.......... 3
Local domain sizes................... (1024,1024,64)
Maximum size of local domain......... 1024
Test is PASSED.
```

This program shows the main information of the OpenCL devices found in the system. The great majority of the tests delivered with HPL, launch one or several kernels in the OpenCL device found and comparing the result with a sequential CPU implementation. If the system has several OpenCL devices, HPL will choose a GPU in first place. You can try for example, Matmul that computes the product of two matrices. The program will show "Test is PASSED" or "Test is FAILED" if the data comparison was successful or not.

3 VectorAddition

This is the first program. Basically, you have to define three arrays `a`, `b` and `c` and compute `c = a + b`. In the web, you have a good starting point.

[2]The nodes with GPUs are scarce resources, and probably you will have to wait so much. To avoid this wait, you can use the nodes without GPUs using the CPU as OpenCL devices. For that, you only need to replace the `job_gpu.job` by `job_cpu.job` available on the web as well.

```
$ wget http://gac.udc.es/~moises/ficheros/tudelft/VectorAddition_skel.cpp
```

`VectorAddition_skel.cpp` is the test that you have to develop but it has some empty spaces (identified by "...") that you will have to fulfill. The easiest way to compile and execute these codes is copying them to the `/tests` folder and modifying the `/tests/Makefile` like the `http://gac.udc.es/~moises/ficheros/tudelft/Makefile` including the name of the new tests. In this first program, you will have to complete only the kernel code.

4 Caesar

The Caesar cipher is a simple encrypting algorithm. It consists in adding an offset to each letter of an input message obtaining the encrypted message. For this program, the first we need is the message that you will find in the `/tmp` folder of the head node of the cluster. So, first we are going to copy it to our accounts:

```
$ cp /tmp/original.data $HOME/hpl/tests
```

Again, in the web you have an incomplete HPL implementation. `Caesar_skel.cpp`[3] reads from a input file (original.data) the message to be encrypted. First it encrypts the message by means of a sequential implementation executed on the CPU. After this, the message is also encrypted on the GPU with the kernel that you will have to develop. So, for this second test, You will have to define the kernel entirely. Additionally, there are more empty spaces in the host code.

Finally, to verify your results, you can compare the two encrypted messages: `sequential.data` and `hpl.data`. [4]

```
$ diff sequential.data hpl.data
```

5 More?

If you want to know more about HPL, you can try more tests by yourself or for example, you can do the decryption kernel for the Caesar cipher.

[3] wget http://gac.udc.es/~moises/ficheros/tudelft/Caesar_skel.cpp

[4] Probably you noted that the execution time in HPL is higher than the sequential. There are two reasons to explain this behaviour. The first one is that HPL compiles the kernel code the first time that it is launched. Thus, this compilation time is included in the total execution time. The second one is that the original data is very small file. If you uses a file bigger (for example 512 MB) you will see how your HPL code is several times faster than sequential code.