Saturday, June 1, 2013

**Tutorial: Simple start with OpenCL and C++**

To begin programming in OpenCL is always hard. Let's try with the basic example. We want to sum two arrays together.  
  
At first you need to install the OpenCL libraries and other files. AMD has for CPU's and their GPU's  AMD APP: <http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/downloads/>. Intel has their OpenCL libraries at <http://software.intel.com/en-us/vcsource/tools/opencl-sdk>. And Nvidia has everything at <https://developer.nvidia.com/cuda-downloads>. In some cases the graphic drivers already include all the files you need. I recommend that you continue with the next step and if anything will go wrong return to this step and install the needed OpenCL SDK toolkits.  
  
  
  
We will program in C++11. To ease everything we will use OpenCL C++ binding 1.1 from[www.khronos.org/registry/cl/api/1.1/cl.hpp](http://www.khronos.org/registry/cl/api/1.1/cl.hpp) . manual for this binding is available at[www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf](http://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf). It might happen that cl.hpp is already installed at your computer. If not then simply download C++ binding to folder of your project. Don't forget to turn on the C++11. In case of QtCreator add next line into the .pro file: 

QMAKE\_CXXFLAGS += -std=c++0x

Also don't forget to use OpenCL library. In case of QtCreator add next line into the .pro file:

LIBS+= -lOpenCL

If you get any errors you need to adjust system variable to point to folder of OpenCL installation. You can also manually set path to OpenCL library path:

LIBS+= -Lpath\_to\_openCL\_libraries

Or you can simply write hard-coded path to OpenCL library: 

LIBS+=/usr/.../libOpenCL.so

Let's start with coding. We will create simple console program which will use OpenCL to sum two arrays like C=A+B. For our simple sample we will need only two headers:

#include <iostream> #include <CL/cl.hpp>

Everything else will happen inside main function. At start we need to get one of the OpenCL platforms. This is actually a driver you had previously installed. So platform can be from Nvidia, Intel, AMD....

int main(){ //get all platforms (drivers) std::vector<cl::Platform> all\_platforms; cl::Platform::get(&all\_platforms); if(all\_platforms.size()==0){ std::cout<<" No platforms found. Check OpenCL installation!\n"; exit(1); } cl::Platform default\_platform=all\_platforms[0]; std::cout << "Using platform: "<<default\_platform.getInfo<CL\_PLATFORM\_NAME>()<<"\n";

Once we selected the first platform (**default\_platform**) we will use it in the next steps. Now we need to get device of our platform. For example AMD's platform has support for multiple devices (CPU's and GPU's). We will now select the first device (**default\_device**):

//get default device of the default platform std::vector<cl::Device> all\_devices; default\_platform.getDevices(CL\_DEVICE\_TYPE\_ALL, &all\_devices); if(all\_devices.size()==0){ std::cout<<" No devices found. Check OpenCL installation!\n"; exit(1); } cl::Device default\_device=all\_devices[0]; std::cout<< "Using device: "<<default\_device.getInfo<CL\_DEVICE\_NAME>()<<"\n";

Now we need to create a Context. Imagine the Context as the runtime link to the our device and platform:

cl::Context context({default\_device});

Next we need to create the program which we want to execute on our device:

cl::Program::Sources sources;

Actual source of our program(**kernel**) is there:

// kernel calculates for each element C=A+B std::string kernel\_code= " void kernel simple\_add(global const int\* A, global const int\* B, global int\* C){ " " C[get\_global\_id(0)]=A[get\_global\_id(0)]+B[get\_global\_id(0)]; " " } ";                                                                          ";

This code simply calculates C=A+B. As we want that one thread calculates sum of only one element, we use get\_global\_id(0). get\_global\_id(0) means get id of current thread. Id's can go from 0 to get\_global\_size(0) - 1. get\_global\_size(0) means number of threads. What is 0? 0 means first dimension. OpenCL supports running kernels on 1D, 2D and 3D problems. We will use 1D array! This means 1D problem.    
  
Next we need our kernel sources to build. We also check for the errors at building:

sources.push\_back({kernel\_code.c\_str(),kernel\_code.length()}); cl::Program program(context,sources); if(program.build({default\_device})!=CL\_SUCCESS){ std::cout<<" Error building: "<<program.getBuildInfo<CL\_PROGRAM\_BUILD\_LOG>(default\_device)<<"\n"; exit(1); }

For arrays A, B, C we need to allocate the space on the device:

// create buffers on the device cl::Buffer buffer\_A(context,CL\_MEM\_READ\_WRITE,sizeof(int)\*10); cl::Buffer buffer\_B(context,CL\_MEM\_READ\_WRITE,sizeof(int)\*10); cl::Buffer buffer\_C(context,CL\_MEM\_READ\_WRITE,sizeof(int)\*10);

Arrays will have 10 element. We want to calculate sum of next arrays (A, B).

int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; int B[] = {0, 1, 2, 0, 1, 2, 0, 1, 2, 0};

We need to copy arrays from A and B to the device. This means that we will copy arrays from the host to the device. **Host** represents our **main**. At first we need to create a queue which is the queue to the commands we will send to the our device:

//create queue to which we will push commands for the device. cl::CommandQueue queue(context,default\_device);

Now we can copy data from arrays A and B to buffer\_A and buffer\_B which represent memory on the device:

//write arrays A and B to the device queue.enqueueWriteBuffer(buffer\_A,CL\_TRUE,0,sizeof(int)\*10,A); queue.enqueueWriteBuffer(buffer\_B,CL\_TRUE,0,sizeof(int)\*10,B);

Now we can run the kernel which in parallel sums A and B and writes to C. We do this with KernelFunctor which runs the kernel on the device. Take a look at the *"simple\_add"*this is the name of our kernel we wrote before. You can see the number 10. This corresponds to number of threads we want to run (our array size is 10):

cl::KernelFunctor simple\_add(cl::Kernel(program,"simple\_add"),queue,cl::NullRange,cl::NDRange(10),cl::NullRange);

Here we actually set the arguments to kernel **simple\_add** and run the kernel: 

simple\_add(buffer\_A, buffer\_B, buffer\_C);

At the end we want to print memory C on our device. At first we need to transfer data from the device to our program (host):

    int C[10]; //read result C from the device to array C queue.enqueueReadBuffer(buffer\_C,CL\_TRUE,0,sizeof(int)\*10,C); std::cout<<" result: \n"; for(int i=0;i<10;i++){ std::cout<<C[i]<<" "; } return 0; }

This is it. Complete code is there:

1. #include <iostream>
2. #include <CL/cl.hpp>
4. int main(){
5. //get all platforms (drivers)
6. std::vector<cl::Platform> all\_platforms;
7. cl::Platform::get(&all\_platforms);
8. if(all\_platforms.size()==0){
9. std::cout<<" No platforms found. Check OpenCL installation!\n";
10. exit(1);
11. }
12. cl::Platform default\_platform=all\_platforms[0];
13. std::cout << "Using platform: "<<default\_platform.getInfo<CL\_PLATFORM\_NAME>()<<"\n";
15. //get default device of the default platform
16. std::vector<cl::Device> all\_devices;
17. default\_platform.getDevices(CL\_DEVICE\_TYPE\_ALL, &all\_devices);
18. if(all\_devices.size()==0){
19. std::cout<<" No devices found. Check OpenCL installation!\n";
20. exit(1);
21. }
22. cl::Device default\_device=all\_devices[0];
23. std::cout<< "Using device: "<<default\_device.getInfo<CL\_DEVICE\_NAME>()<<"\n";

26. cl::Context context({default\_device});
28. cl::Program::Sources sources;
30. // kernel calculates for each element C=A+B
31. std::string kernel\_code=
32. " void kernel simple\_add(global const int\* A, global const int\* B, global int\* C){ "
33. " C[get\_global\_id(0)]=A[get\_global\_id(0)]+B[get\_global\_id(0)]; "
34. " } ";
35. sources.push\_back({kernel\_code.c\_str(),kernel\_code.length()});
37. cl::Program program(context,sources);
38. if(program.build({default\_device})!=CL\_SUCCESS){
39. std::cout<<" Error building: "<<program.getBuildInfo<CL\_PROGRAM\_BUILD\_LOG>(default\_device)<<"\n";
40. exit(1);
41. }

44. // create buffers on the device
45. cl::Buffer buffer\_A(context,CL\_MEM\_READ\_WRITE,sizeof(int)\*10);
46. cl::Buffer buffer\_B(context,CL\_MEM\_READ\_WRITE,sizeof(int)\*10);
47. cl::Buffer buffer\_C(context,CL\_MEM\_READ\_WRITE,sizeof(int)\*10);
49. int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
50. int B[] = {0, 1, 2, 0, 1, 2, 0, 1, 2, 0};
52. //create queue to which we will push commands for the device.
53. cl::CommandQueue queue(context,default\_device);
55. //write arrays A and B to the device
56. queue.enqueueWriteBuffer(buffer\_A,CL\_TRUE,0,sizeof(int)\*10,A);
57. queue.enqueueWriteBuffer(buffer\_B,CL\_TRUE,0,sizeof(int)\*10,B);

60. //run the kernel
61. cl::KernelFunctor simple\_add(cl::Kernel(program,"simple\_add"),queue,cl::NullRange,cl::NDRange(10),cl::NullRange);
62. simple\_add(buffer\_A,buffer\_B,buffer\_C);
64. //alternative way to run the kernel
65. /\*cl::Kernel kernel\_add=cl::Kernel(program,"simple\_add");
66. kernel\_add.setArg(0,buffer\_A);
67. kernel\_add.setArg(1,buffer\_B);
68. kernel\_add.setArg(2,buffer\_C);
69. queue.enqueueNDRangeKernel(kernel\_add,cl::NullRange,cl::NDRange(10),cl::NullRange);
70. queue.finish();\*/
72. int C[10];
73. //read result C from the device to array C
74. queue.enqueueReadBuffer(buffer\_C,CL\_TRUE,0,sizeof(int)\*10,C);
76. std::cout<<" result: \n";
77. for(int i=0;i<10;i++){
78. std::cout<<C[i]<<" ";
79. }
81. return 0;
82. }