SnuCL-Tr: OpenCL to CUDA Quick Start Guide

Requirements

The OpenCL to CUDA translator in SnuCL-Tr requires that you have the latest CUDA Toolkit installed. You can download the CUDA Toolkit from https://developer.nvidia.com/cuda-downloads.

How to Install

Perform the following steps to install the OpenCL to CUDA translator in SnuCL-Tr and verify the installation.

1. Untar Source Code

   Download the SnuCL-Tr source code from http://snucl.snu.ac.kr/snucl-tr.html. Untar it on your preferred location.

   ```
   $ tar xvzf snucl-tr.tar.gz
   ```

2. LLVM

   You need to configure the LLVM compiler first and then compile the program. It may take a long time to compile for LLVM.

   ```
   $ cd opencl2cuda/build
   $ ../llvm.mod/configure
   $ make BUILD_EXAMPLES=1
   ```

   Note: If your system already has Clang, then you need to configure and build the LLVM compiler manually with the following flags.

   ```
   CC=gcc CXX=g++
   ```

   Example:

   ```
   $ ../llvm.mod/configure CC=gcc CXX=g++
   $ make CC=gcc CXX=g++ BUILD_EXAMPLES=1
   ```

3. The Runtime Library

   Build the runtime library (i.e., wrapper functions) at the location below.

   ```
   $ cd opencl2cuda/common/common/
   $ make
   ```

   As a result, the shared library will be created in the following location:

- opencl2cuda/common/lib/libsnuclOC.a

4. Set Environment Variables

There are two environment variables that have to be set to use the OpenCL to CUDA translator: OPENCL_TO_CUDA and OPENCL_TO_CUDA_GPU_ARCH (i.e., your GPU's compute capability). You can check the GPU's compute capability at https://developer.nvidia.com/cuda-gpus.

Open your .bashrc to edit.

```
$ vi $(HOME)/.bashrc
```

At the bottom of the file, insert the two lines shown below.

```
export OPENCL_TO_CUDA=$(HOME)/opencl2cuda
export OPENCL_TO_CUDA_GPU_ARCH=compute_xx
```

For example, if your GPU's compute capability is 3.0, then modify the OPENCL_TO_CUDA_GPU_ARCH variable as shown below.

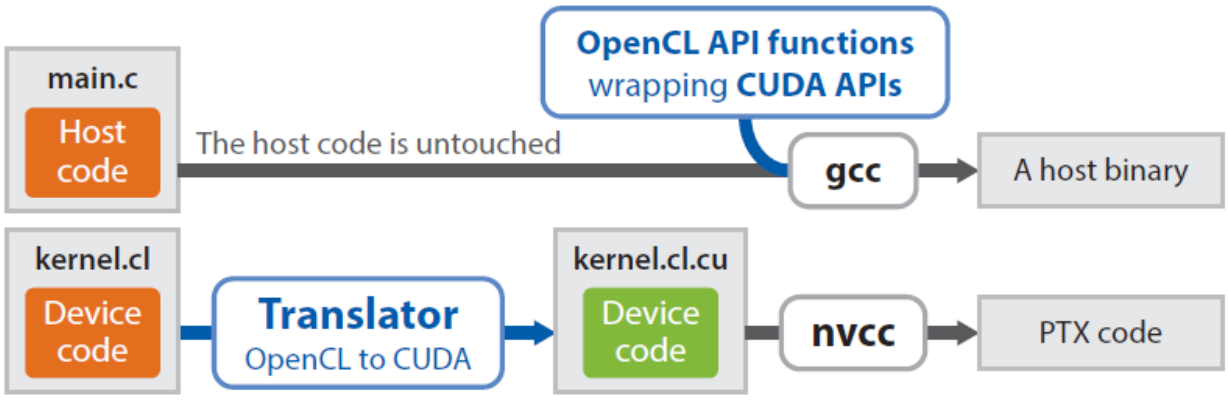
```
export OPENCL_TO_CUDA_GPU_ARCH=compute_30
```

To apply modified environment variables, go to your home directory and execute the following command.

```
$ source .bashrc
```

Understanding the OpenCL to CUDA Translator

In OpenCL, the host code and device code are separated. Hence, we translate them separately. The OpenCL device code (e.g., kernel.cl) is translated to the CUDA device code (e.g., kernel.cl.cu) by our source-to-source translator. The OpenCL host API functions are implemented as wrapper functions. We use CUDA driver API functions to implement the wrappers.



How to Build a Program Using the OpenCL to CUDA Translator

- Makefile Template

  In **sample** directory, a Makefile template for the OpenCL to CUDA translator is provided with a sample application. Makefile can be written as you deem fit, but there are four things you have to follow to use the translator.

  - Use g++ compiler

    ```
    CC = g++
    ```

  - Add CUDA and the runtime library path to search

    ```
    -L$(CUDA_INSTALLED_PATH)/lib64 –L$(OPENCL_TO_CUDA)/common/lib
    ```

  - Link CUDA and the runtime library

    ```
    -lcuda –lcudart –lsnuclOC -lpthread
    ```

  - Add CUDA header file path

    ```
    -I$(CUDA_INSTALLED_PATH)/include
    ```

Translation Result for the Sample Application

When you build your program with the OpenCL to CUDA translator, translated source files will be named "*.cu". For example, when you build the sample application, "__temp_kernel.cu" will be generated. You can open it with text-editor to see how it is translated. The figures shown below are the original device code and the translated device code of the sample application.

- kernel.cl (original kernel code)

```
__global__ void vecAdd(
  __global int* A, __global int* B, __global int* C, const int N) {
    int i = get_global_id(0);
    C[i] = A[i] + B[i];
}
```

- __temp_kernel.cu (translated kernel code)

```
__constant__ char __snucl_const_mem[16384];
extern __shared__ char __snucl_shared_mem[];
__device__ unsigned int __snucl_group_id_offs[2] = {0, 0};
__device__ int get_global_id( int index) {
    switch (index){
        case 0: return (blockIdx.x + __snucl_group_id_offs[0]) *
                  blockDim.x + threadIdx.x;
        case 1: return blockIdx.y * blockDim.y + threadIdx.y;
        case 2: return blockIdx.z * blockDim.z + threadIdx.z;
        default: return -1;
    };
}

extern “C” __global__ void vecAdd(int* A, int* B, int* C, const int N){
    int id = get_global_id(0);
    C[i] = A[i] + B[i];
}
```