SnuCL-Tr: CUDA to OpenCL Quick Start Guide

Requirements

The CUDA to OpenCL translator in SnuCL-Tr requires CUDA Toolkit 7.0. You can download the CUDA Toolkit from https://developer.nvidia.com/cuda-toolkit-70.

If cuBLAS is used in your application, you can use clBLAS for the translated OpenCL application. You can download clBLAS from https://github.com/clMathLibraries/clBLAS/releases.

How to Install

Perform the following steps to install the CUDA to OpenCL 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 require a long time to compile.

```
$ cd cuda2opencl/llvm-3.3/build
$ ../configure --enable-optimized
$ make
```

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

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

For example,

```
$ ../configure --enable-optimized CC=gcc CXX=g++
$ make CC=gcc CXX=g++
```

3. Set Environment Variables

There are three environment variables to be set to use the CUDA to OpenCL translator; CUDA_DIR, CUDA_TO_OPENCL_DIR, and CLBLAS_DIR.

Open your .bashrc to edit.

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

At the bottom of the file, insert the three lines below. Note that the locations should be specified to where you installed each software.

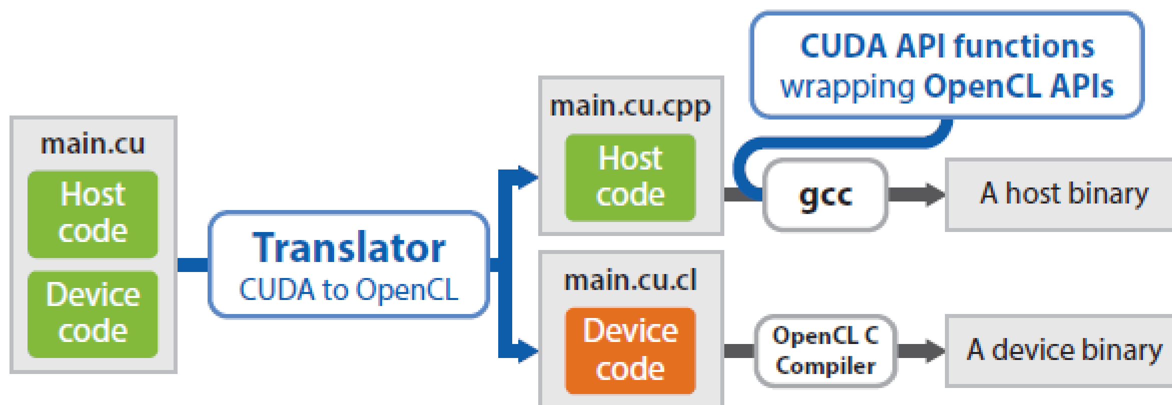
```
export CUDA_DIR=/usr/local/cuda
export CUDA_TO_OPENCL_DIR=$HOME/cuda2opencl
export CLBLAS_DIR=$HOME/clBLAS
```

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

```
$ source .bashrc
```

Understanding the CUDA to OpenCL Translator

In CUDA, the host and the device code are mixed in source files. Our source-to-source translator separates the device code and host code from mixed source files. For each source file, the translator generates two new files. See the figure below.



For example, assume that we have a CUDA source file named “main.cu”. Our translator generates an OpenCL host code file named “main.cu.cpp” and an OpenCL device code file named “main.cu.cl”. Then, a native compiler, such as gcc, compiles the host code and generates an executable binary. At this time, our OpenCL wrapper library functions are linked to the executable.

How to Build a Program Using the CUDA to OpenCL Translator

- Makefile

In **sample** directory, a Makefile template for the CUDA to OpenCL translator is provided with a sample application. The sample application adds two vectors using CUDA. The result is then transferred to the host and verified.

Variables in **Makefile** in the sample application are described below. At the last line, **Makefile** includes **$(CUDA_TO_OPENCL_DIR)/wrapper/Makefile**. If you want to know the detail of how the translator generates an executable, you can take a look into that file.

- **TARGET** specifies the filename of the executable binary. When you compile your program with **Makefile**, the executable will be generated as the name specified in **TARGET**.

```
TARGET=vector_add
```

- **C_SRCS** specifies source files written in C. Files specified here will be translated and compiled. If you don't have any source files written in C, leave it empty. If you have more than two files, they should be separated one or more whitespaces. This is the syntax of Makefile.

```
C_SRCS=
```

- **CXX_SRCS** specifies source files written in C++. Files specified here will be translated and compiled. If you don't have any source files written in C++, leave it empty.

```
CXX_SRCS=
```

- **CU_SRCS** specifies source files written in CUDA. Files specified here will be translated and compiled. If you don't have any source files written in CUDA, leave it empty.

```
CU_SRCS=vector_add.cu
```

- **EXTERN_LIBS** specifies any files that you want to link together when the target is linked. If you have a file that does not need to be translated, you can specify that file in this variable. This eliminates unnecessary translation processes.

```
EXTERN_LIBS=library_a.c library_b.o library_c.a
```

- **COMMON_CFLAGS** specifies any compile flags you want to specify when the source code is compiled. This flags are applied when the source files are translated and compiled.

```
COMMON_CFLAGS=-I.
```

After you have done editing Makefile, just type "make". The CUDA to OpenCL translator will translate files specified in **C_SRCS**, **CXX_SRCS**, and **CU_SRC** and compile those files automatically. The executable for OpenCL will be generated in the same directory. Then, you can run the executable.

Note that you should execute the program at the directory where you type "make". It is because the device code files are generated at that directory, and our runtime builds device code files in the current working directory. If you execute the program in another directory, our runtime cannot build device code files.

Translation Result for the Sample Application

When you build your program with the CUDA to OpenCL translator, translated source files will be named as "*.cu.cpp" or "*.cu.cl". For example, when you build the sample application, "vector_add.cu.cpp" and "vector_add.cu.cl" will be generated. You can open them with a text editor to see how they are

translated. The below figure shows the original source code and translated source code of the sample application.

- vector_add.cu (original source code)

```
int* d_A;
int* d_B;
int* d_C;

// device code
__global__ void vec_add(int* A, int* B, int* C, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// host code
int main(int argc, char** argv) {
    int N = 100000;

    int* h_A = (int*)malloc(N * sizeof(int));
    int* h_B = (int*)malloc(N * sizeof(int));
    int* h_C = (int*)malloc(N * sizeof(int));

    for (int i = 0; i < N; ++i) {
        h_A[i] = rand();
        h_B[i] = rand();
    }

    cudaMalloc((void**)&d_A, N * sizeof(int));
    cudaMalloc((void**)&d_B, N * sizeof(int));
    cudaMalloc((void**)&d_C, N * sizeof(int));

    cudaMemcpy(d_A, h_A, N * sizeof(int),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, N * sizeof(int),
cudaMemcpyHostToDevice);

    int threads_per_block = 256;
    int blocks_per_grid = (N + threads_per_block - 1) /
threads_per_block;
    vec_add<<<blocks_per_grid, threads_per_block>>>(d_A, d_B,
d_C, N);

...
```

- vector_add.cu.cpp (translated host code)

```
#include "cuda_for_clang.h"

 int *d_A;
 int *d_B;
 int *d_C;
 int main( int argc,  char **argv) {
    {
        InitOpenCL();
    }
     int N = 100000;
     int *h_A = (int *)malloc(N * sizeof(int));
     int *h_B = (int *)malloc(N * sizeof(int));
     int *h_C = (int *)malloc(N * sizeof(int));
    for ( int i = 0; i < N; ++i) {
        h_A[i] = rand();
        h_B[i] = rand();
    }
    cudaMalloc((void **)&d_A, N * sizeof(int));
    cudaMalloc((void **)&d_B, N * sizeof(int));
    cudaMalloc((void **)&d_C, N * sizeof(int));
    cudaMemcpy(d_A, h_A, N * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, N * sizeof(int), cudaMemcpyHostToDevice);
     int threads_per_block = 256;
     int blocks_per_grid = (N + threads_per_block - 1) /
threads_per_block;
     cl_kernel vec_add_0;
    {
         int file_idx =
getKernelFileIndex("vector_add.ocl_device.cl");
         int kern_idx = getKernelIndex(file_idx, "vec_add");
        vec_add_0 = cl_kernels[file_idx][kern_idx];
    }
    cl_error = clSetKernelArg(vec_add_0, 0, sizeof(cl_mem), (void
*)&d_A);
    if (cl_error != CL_SUCCESS)
        fatal_CL(cl_error, __FILE__, __LINE__);

...
```

- vector_add.cu.cl (translated kernel code)

```
__kernel void vec_add(
   __global int *A, __global int *B, __global int *C, int N) {
     int i = get_local_size(0) * get_group_id(0) + get_local_id(0);
    if (i < N)
        C[i] = A[i] + B[i];
}
```