White Paper | Get started with OpenCL on i.MX6

Rev.1.0

Table of Contents

1 Introduction

This white paper will introduce the proper step by step instructions to the user regarding how to start OpenCL development on i.Mx6 Board.

This paper talks about hardware and software requirements and their setup. It demonstrates development of first OpenCL application using them.

1.1 What is OpenCL?

Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs and other processors. Open CL provides parallel computing using task based and data based parallelism.

In order to visualize the heterogeneous architecture in terms of the API and restrict memory usage for parallel execution, OpenCL defines multiple cascading layers of virtual hardware definitions
The basic execution engine that runs the kernels is called a Processing Element (PE).
A group of Processing Elements is called a Compute Unit (CU).Finally a group of Compute Unit is called Compute Device.
A host system could interact with multiple Compute Devices on a system (e.g., a GPGPU and a DSP), but data sharing and synchronization is coarsely defined at this level

1.2 Understanding i.MX6

The i.MX 6 series of applications processors are scalable multicore platform that includes single-, dual- and quad-core families based on the ARM® Cortex™-A9 architecture.
They have superior 3D graphics performance with up to quad shaders performing 200 Mb/s and OpenCL support.
They have separate 2D and/or Vertex acceleration engines for an optimal user interface also,experience and stereoscopic image sensor support for 3D imaging

1.3 Developing First OpenCL Application

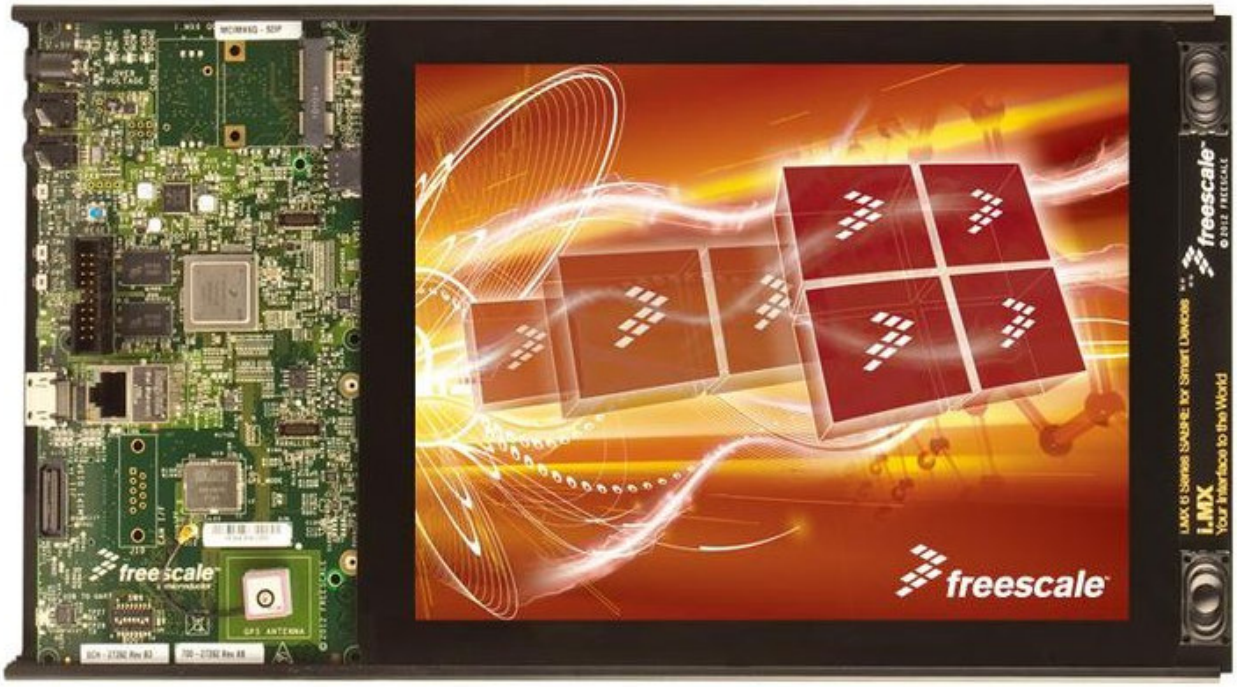
- Setup hardware and software environment for development
- Write the host and kernel programs
- Write functions for
    - Hardware Initialization
    - Kernel Compilation
    - Kernel Execution
    - OpenCL finish

2 Requirements

2.1 Hardware

SABRE Platform for Smart Device Based on i.MX6 Series.
The kit contains:

Item	Description
Smart device	platform Main board, LVDS display and speaker board, pre-assembled
Cable	USB cable (micro-B to standard-A)
Power supply	5 V/5 A universal power supply
8 GB SD card	Bootable demonstration code for smart device platform



2.2 Software

1. Install and configure BSP and LTI

3 Developing Environment

Before starting to develop the first OpenCL application, we need to set up environment for that.
1. Installing Board Support Package (BSP) and Linux Target Image Builder (LTIB)

Steps:
1.Download L3.0.35_4.1.0_ER_SOURCE_BSP.tar.gz from Freescale Website.
2.Extract .tar to folder on your host e.g. L3.0.35_4.1.0_130816_source
3. Go to that folder where you extracted tar and enter command
./install
4. Continue installation by typing Y and accept EULA agreement.
5. Enter the path to install LTIB (Your LTIB will be installed there)
6. Go inside ltib directory and enter command
./ltib
It will appear as GUI, if everything went right.
I faced following issues before this screen:

Error1 : Error with host Packages :
Make sure to have following host packages installed in your machine,

Package	Minimum ver
libstdc++	0
gcc-c++	2.96
zlib	0
zlib-devel	0
rpm	0
rpm-build	0
ncurses-devel	0
m4	0
bison	0
tcl	0

Install all above packages using command :
Sudo apt-get install *package-name*

Error2: Error in installation of zlib
I installed liblz-dev and zlib1g-dev but still getting following error with zlib.

Package	Minimum ver	Installed info
zlib	0	not installed

The solution for zlib error can be found on this community thread.
https://community.freescale.com/thread/302017

Follow the instructions on this thread to resolve the issue.

Once you resolve all errors your host packages will be installed on your machine this may take up to an hour.
Install any missing package if you get error.

7.After successful installation type command :

./ltib –c

You will get GNU/Linux Target Image Builder window.

8. Select Platform : Freescale imx Reference Board (default)
Select <Exit > and press Enter.

9. Select imx6q as platform type and package profile is min (default)

10. Select Target Image generation , change target image to NFS only.

11. Select package as **gpu-viv-bin-mx6q**

Save configuration and let it rebuild

Error3:Busy Box Error
If you get busy box error: Solution for it as below
@ARGV = grep { `file $_` =~ m,ASCII C program text, } @ARGV;
to
@ARGV = grep { `file $_` =~ m,ASCII\s+.*text, } @ARGV;
on files:
<ltib>/dist/lfs-5.1/base_libs/base_libs.spec

<ltib>/dist/lfs-5.1/glibc/glibc-2.3.2.spec

<ltib>/dist/lfs-5.1/glibc/glibc.spec

Error 4: Error with mtd-utils package
If you are facing issue with installation of any package try installing it by using following command.
./ltib –p *package-name* –m prep
./ltib –p *package-name* –m scbuild
./ltib –p *package-name* –m scdeploy

4 First OpenCL Application

OpenCL application has two main parts host program and kernel program.
Host programs executes on host in our case CPU and Device programs runs on device in our case GPU.
Device program contains all the magic of OpenCL.

4.1. Kernel Program

Developer writes Device program in form of special functions called 'Kernel'. Kernel is reserved keyword.
Kernels are only functions that can be called by host. They always return void.
Kernel code can be written inline inside the host program or in external file. We will write in external file and read that file in host program.

This kernel program is for 'Buffer Copy' function. Input is coming from input buffer and output is coming from another buffer using GPU.
We are using 2D matrix as input hence for processing each element of it we need to have position of each element in matrix, we will get it by using int id=(y*width)+x
Get_global_id(0) = returns the index of current element to be processed.

```
__kernel void helloworld (     __global uchar *input,
                        __global uchar *output,
                        int width,
                        int height
                    )
{
      int y = get_global_id (0);
      int x = get_global_id (1);
      int id = (y * width) + x;

      output[id] = input[id];
}
```

4.2. Host Program

Host controls the execution of kernel. We have to create environment to run kernel.
We must include following OpenCL header with normal C headers.

```
#include<CL/cl.h>
```

We must declare following global variables before writing host program.

```
    cl_platform_id  platform_id;
    cl_device_id  device_id;
    cl_context     context;
    cl_command_queue cq;
    cl_program    program;
```

Vivante Specifications:
The i.MX 6Q contains a Vivante GC2000 GPGPU capable of running OpenCL 1.1 Embedded Profile (EP). The Embedded Profile has 4 shader cores which mean there are 4 processing elements per compute unit.

4.2.1 Hardware initialization:
Creating platform using function clGetPlatformIDs
What is Platform?
The host plus a collection of devices managed by the OpenCL framework that allow an application to share resources and execute kernels on devices in the platform.
Platforms are represented by a cl_platform object, which can be initialized using the following function

```
ret = clGetPlatformIDs (1, &platform_id,  &platforms );
```

Here, Number of platform entries =1, &platform_id= pointer to platform object , &platform = returns number of platforms.

Device: is represented by cl_device objects, initialized with the following function.
clGetDeviceIDs obtains list of available devices.

```
ret = clGetDeviceIDs (platform_id ,CL_DEVICE_TYPE_GPU,1,&device_id,&devices);
```

CL_DEVICE_TYPE_GPU = Bitfield identifying the type.
For the GPU we use Number of devices, typically 1
&device_id =Pointer to the device object
&devices=Puts here the number of devices matching the device_type

Properties: Specifies a list of context property names and their corresponding values. Each property name is immediately followed by the corresponding desired value.
The list is terminated with 0. *Properties* can be NULL in which case the platform that is selected is implementation-defined.

```
cl_context_properties properties[] = {CL_CONTEXT_PLATFORM, (cl_context_properties)platform_id, 0};
```

Now we create context using function clCreateContext
What is context?
Context defines the entire OpenCL environment, including OpenCL kernels, devices, memory management, command-queues, etc. Contexts in OpenCL are referenced by an cl_context object, which must be initialized using the following function:

```
context = clCreateContext(properties, 1,&device_id,NULL,NULL,&ret);
```

Properties=Bitwise with the properties (see specification)
Number of devices =1
&device_id =Pointer to the device object
&ret = error code result

What is Command-Queue?
The OpenCL command-queue, as the name may suggest, is an object where OpenCL commands are enqueued to be executed by the device. The command-queue is created on a specific device in a context having multiple command-queues allows applications to queue multiple independent commands without requiring synchronization.

```
cq = clCreateCommandQueue(context, device_id,0,&ret);
```

Context=must be a valid OpenCL context.
device_id=must be a device associated with context
&ret =error code result

After creating queues we finish Hardware Initialization.

4.2.2 Kernel Compilation
ret = FSLCL_LoadKernelSource ((char *)"helloworld.cl", &app_kernel);

To Submit the source code of the kernel to OpenCL we use clCreateProgramWithSource

```
program = clCreateProgramWithSource (context,  1, (const char **)&app_kernel.src, 0,&ret)
```

4.2.3 Kernel Execution
clBuildProgram function builds (compiles and links) a program executable from the program source or binary.

```
ret = clBuildProgram (program,
```

program= program object
device_id=A pointer to a list of devices that are in program. NULL = all devices associated with the program
The number of devices listed in device_list =1

We will set data on buffers after creation.
clEnqueueWriteBufferEnqueue commands to write to a buffer object from host memory, hence write data will be processed from CPU to GPU.

```
ret = clEnqueueWriteBuffer (cq, buffer_input, CL_TRUE, 0, size_2d, data, 0, NULL, NULL);
```

cl_command_queue cq = Refers to the command-queue in which the write command will be queued. *command_queue* and *buffer* must be created with the same OpenCL context.
Buffer_input = valid buffer object
blocking_write Indicates if the write operations are *blocking* or *nonblocking*.
If *blocking_write* is CL_TRUE, the OpenCL implementation copies the data referred to by *ptr* and enqueues the write operation in the command-queue. The memory pointed to by *ptr*can be reused by the application after the clEnqueueWriteBuffer call returns.
Size_2d= the size of data in bytes being written

clEnqueueNDRangeKernel function enqueues a command to execute a kernel on a device

```
ret = clEnqueueNDRangeKernel(cq, kernel, dimension, NULL, &global, &local, 0, NULL, NULL);
```

Cq= valid command queue.The kernel will be queued for execution on the device associated with *command_queue*
Kernel = valid kernel object
dimension= The number of dimensions used to specify the global work-items and work-items in the work-group. it must be greater than zero and less than or equal to three

Read back from GPU memory to CPU memory using clEnqueueReadBuffer

```
ret = clEnqueueReadBuffer(cq, buffer_output, CL_TRUE, 0, size_2d, data2, 0, NULL, NULL);
```

4.2.4. OpenCL finish
Free all memory by clearing buffers

```
clFlush( cq);
clFinish(cq);

clReleaseMemObject (buffer_input);
clReleaseMemObject (buffer_output);

clReleaseContext(context);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cq);
```

Allocating Memory for 2D array before writing host Program using malloc

```
data = (char **) malloc (buffer_width * sizeof (char *));
data2 = (char **) malloc (buffer_width * sizeof (char *));

for (i = 0; i < buffer_width; i++)
{
data[i] = (char *) malloc (buffer_height * sizeof (char));
data2[i] = (char *) malloc (buffer_height * sizeof (char));
}
```

5 How to compile and run

First export ROOTFS_DIR using following command
Export ROOTFS_DIR=/your_path/ltib/rootfs

Create a make file as following

```
APPNAME                 = helloworld

TOOLCHAIN               = /opt/freescale/usr/local/gcc-4.6.2-glibc-2.13-linaro-multilib-2011.12/fsl-linaro-
toolchain/
CROSS_COMPILER          = $(TOOLCHAIN)/bin/arm-fsl-linux-gnueabi-

CC                      = $(CROSS_COMPILER)gcc
DEL_FILE                = rm -rf
CP_FILE                 = cp -rf

TARGET_PATH_LIB         = $(ROOTFS_DIR)/usr/lib
TARGET_PATH_INCLUDE     = $(ROOTFS_DIR)/usr/include

CFLAGS                  = -DLINUX -DUSE_SOC_MX6 -Wall -O2 -fsigned-char -march=armv7-a -mfpu=neon -
mfloat-abi=softfp \
                          -DEGL_API_FB -DGPU_TYPE_VIV -DGL_GLEXT_PROTOTYPES -
DENABLE_GPU_RENDER_20 \
                          -I../include -I$(TARGET_PATH_INCLUDE)

LFLAGS                  = -Wl,--library-path=$(TARGET_PATH_LIB),-rpath-link=$(TARGET_PATH_LIB) -lm \
                          -lglib-2.0 -lOpenCL -lCLC -ldl -lpthread

OBJECTS                 = $(APPNAME).o

first: all

all: $(APPNAME)

$(APPNAME): $(OBJECTS)
        $(CC) $(LFLAGS) -o $(APPNAME) $(OBJECTS)

$(APPNAME).o: $(APPNAME).c
        $(CC) $(CFLAGS) -c -o $(APPNAME).o $(APPNAME).c

clean:
        $(DEL_FILE) $(APPNAME)
```

Compile using make command , it will create binary executable file ,copy these two files in same folder and run .exe file on board to see results.

6 Complete Host Implementation

Host Code

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <CL/cl.h>

#define FSLCL_ERROR -1
#define FSLCL_SUCCESS CL_SUCCESS

cl_mem buffer_input = NULL;
cl_mem buffer_output = NULL;
cl_kernel buffer_kernel = NULL;
size_t buffer_size = 0;
int buffer_width = 1024;
int buffer_height = 1024;

cl_platform_id          platform_id;
cl_device_id            device_id;
cl_context              context;
cl_command_queue cq;
cl_program              program;
cl_kernel   kernel;

struct fsl_kernel_src_str
{
        char *src;
        long size;
};
typedef struct fsl_kernel_src_str fsl_kernel_src;

cl_int FSLCL_LoadKernelSource (char *filename, fsl_kernel_src *kernel)
{

        FILE *fp = NULL;

        fp = fopen (filename, "rb");

        if (fp == NULL)
        {
                printf ("\nFailed to open: %s\n", filename);
                return FSLCL_ERROR;
        }

        fseek (fp, 0, SEEK_END);
        kernel->size = ftell (fp);
        rewind (fp);

        kernel->src =  (char *) malloc (sizeof (char) * kernel->size);
        if (! kernel->src)
        {
                printf ("\nError Allocating memory to load CL program");
                return FSLCL_ERROR;
        }

        fread (kernel->src, 1, kernel->size, fp);
        kernel->src[kernel->size] = '\0';
        fclose (fp);

        return FSLCL_SUCCESS;
}
```

```
int main (int argc, char **argv)
{
        int dimension = 2;
        size_t global[2] = {buffer_width, buffer_height};
        size_t local[2] = {4, 16};
         int size_2d = buffer_width * buffer_height;
        cl_int ret;
        cl_int platforms;
        char **data;
        char **data2;
        int i, j;


        data = (char **) malloc (buffer_width * sizeof (char *));
        data2 = (char **) malloc (buffer_width * sizeof (char *));

        for (i = 0; i < buffer_width; i++)
        {
          data[i] = (char *) malloc (buffer_height * sizeof (char));
          data2[i] = (char *) malloc (buffer_height * sizeof (char));
        }

        for (i = 0; i < buffer_width; i++)
        {
                for (j = 0; j < buffer_height; j++)
                {
                        data[i][j] = 0;
                        data2[i][j] = 0;
                }
        }

        ret = clGetPlatformIDs (1, &platform_id, &platforms  );
        assert (ret == CL_SUCCESS);

        cl_int devices;

        ret = clGetDeviceIDs (platform_id,CL_DEVICE_TYPE_GPU,1,&device_id,                &devices);
        assert (ret == CL_SUCCESS);

cl_context_properties properties[] = {CL_CONTEXT_PLATFORM,  (cl_context_properties)platform_id, 0};

context = clCreateContext(properties,       &device_id, NULL,NULL,&ret);
assert (ret == CL_SUCCESS);

 cq = clCreateCommandQueue(context,      device_id, 0, &ret);
assert (ret == CL_SUCCESS);
 fsl_kernel_src app_kernel;

ret = FSLCL_LoadKernelSource ((char *)"helloworld.cl", &app_kernel);

// Submit the source code of the kernel to OpenCL
program = clCreateProgramWithSource (context, 1, (const char **)&app_kernel.src, 0,&ret);
if (ret == CL_SUCCESS)

// and compile it (after this we could extract the compiled version)
ret = clBuildProgram (program, 1, device_id, NULL, NULL, NULL);
if (ret < 0)
{
printf ("Failed\n");
printf ("\nReturn: %d\n", ret);
clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, app_kernel.size,    app_kernel.src, NULL);
printf ("\n%s", app_kernel.src);
}
assert(ret == CL_SUCCESS);
```

```
buffer_input = clCreateBuffer (context, CL_MEM_READ_ONLY, size_2d, NULL, &ret);
assert (ret == CL_SUCCESS);

buffer_output = clCreateBuffer (context, CL_MEM_WRITE_ONLY , size_2d, NULL, &ret);
assert (ret == CL_SUCCESS);


// get a handle and map parameters for the kernel
kernel = clCreateKernel(program, "helloworld", &ret);
assert (ret == CL_SUCCESS);

clSetKernelArg (kernel, 0, sizeof(cl_mem), &buffer_input);
clSetKernelArg (kernel, 1, sizeof(cl_mem), &buffer_output);
clSetKernelArg (kernel, 2, sizeof(int), &buffer_width);
clSetKernelArg (kernel, 3, sizeof(int), &buffer_height);


for (i = 0; i < buffer_width; i++)
{
        for (j = 0; j < buffer_height; j++)
        {
                data[i][j] = rand () % 10;
        }
}


ret = clEnqueueWriteBuffer (cq, buffer_input, CL_TRUE, 0, size_2d, data, 0, NULL, NULL);
assert (ret == CL_SUCCESS);

ret = clEnqueueNDRangeKernel(cq, kernel, dimension, NULL, &global, &local, 0, NULL, NULL);
assert (ret == CL_SUCCESS);

ret = clEnqueueReadBuffer(cq, buffer_output, CL_TRUE, 0, size_2d, data2, 0, NULL, NULL);
assert (ret == CL_SUCCESS);


printf ("\nResult:\n");
for (i = 0; i < buffer_width; i++)
{
        for (j = 0; j < buffer_height; j++)
        {
                printf ("\n%d -- %d", data[i][j], data2[i][j]);
        }

}

clFlush( cq);
clFinish(cq);

clReleaseMemObject (buffer_input);
clReleaseMemObject (buffer_output);

clReleaseContext(context);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cq);


return 0;

}
```

7 Conclusion

This is how we can start creating first OpenCL application. Kernel contains the logic. User needs to write kernel every time as host functions remain same and we can use earlier host code with little modification.

8 Appendix

https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/

http://cache.freescale.com/files/32bit/doc/user_guide/IMX6SABREINFOQSG.pdf?&Parent_nodeId=&Parent_pageType=