Universidade Federal de Minas Gerais

ParallelME Runtime: User Manual

User manual for ParallelME
low-level runtime: instructions and application example.

Final Report submitted to LGE as part of the Project *Scheduling heterogeneous tasks on hybrid device environments.*

Belo Horizonte, MG
June 04, 2016

ParallelME Runtime: User Manual

1 Introduction

This document aims to present practical information to instruct a programmer on how to implement a particular application directly using the ParallelME low-level runtime. The document includes instructions for environment set up, a step by step guide of the implementation of an application using the runtime and finally some important observations.

2 Grayscale Image Converter

The application that will be implemented as an example is a grayscale image converter. The application will get an image bundled with the application and run a task with the ParallelME Runtime to convert the image to grayscale.

The following algorithm will be used to convert the image to grayscale:

```
1 for(auto pixel : image) {
2     auto luminosity = 0.21 * pixel.r + 0.72 * pixel.g + 0.07 * pixel.b;
3     pixel.r = pixel.g = pixel.b = luminosity;
4 }
```

Following the algorithm, for each pixel in the image the luminosity of the pixel will be computed by a weighted sum of the RGB components of the pixel based on how much the eye sees each component. Then, all the components will be made equal to this luminosity, what will result in a gray color. After all the pixels execute this algorithm, the entire image will be in shades of gray.

As this algorithm is embarrassingly parallel, it can be easily optimized using the ParallelME Runtime.

3 Environment Configuration

3.1 Finding a suitable device

The ParallelME Runtime requires a device with OpenCL support to work. Because of this, the runtime won't work on simulators, and not all Android devices have OpenCL support. To check if your device has OpenCL support, please see the OpenCL-X benchmark[1] app. If an Android device doesn't support OpenCL, trying to instantiate the `parallelme::Runtime` class will throw an exception.

[1] `https://play.google.com/store/apps/details?id=com.maxtrium.opencldevicetest`

3.2 Configuring Android Studio

For the application we'll be using Android Studio[2], the official Android IDE. However, the C++ support isn't enabled by default. To enable this support, the Android NDK Toolkit must be installed and configured.

The following instructions show how to configure Android Studio (2.1) to properly use the NDK Toolkit:

1. Open the SDK Manager (Tools > Android > SDK Manager) and ensure the following packages are installed:
   - On the SDK Plaform tab, the latest Android SDK platform (except for preview versions).
   - On the SDK Tools tab, the latest version of the following packages:
     - Android SDK Build Tools
     - Android NDK
     - Android SDK Platform-Tools
     - Android SDK Tools
2. As of Android Studio 2.1, the Instant Run feature does not work well with the NDK Toolkit: the IDE doesn't always properly update the apk if the NDK code is recompiled while this option is active. Because of this, it is recommended to disable the feature by accessing Preferences > Build, Execution, Deployment > Instant Run and deselect the "Enable Instant Run" option.

3.3 Creating a project

Create a new project in Android Studio with the default project configurations and an empty activity. The ParallelME Runtime can be used with Android API 15 or newer.

After creating the project, open the `app/src/main` folder and create a new `jni` folder. This will be the folder that will contain the C++ code that uses the runtime.

As Android Studio automatic NDK compilation doesn't work properly, we want to disable it and instead write and call the compilation code manually. To disable the automatic compilation, add the following lines to the `build.gradle` file in the `app` directory, inside `android`:

```
android {
    // ...

    sourceSets.main {
        jni.srcDirs = [] // Disable automatic ndk-build
    }
}
```

This will remove all the JNI directories from the gradle script and stop Android Studio from automatically trying to compile the code. This also means that before compiling the Java code on

[2] `https://developer.android.com/studio`

Android Studio, the `ndk-build`[3] command should be run from inside the `app/src/main` folder to manually compile the C++ source.

The `ndk-build` command reads the `Android.mk`[4] and `Application.mk`[5] files inside the `jni` folder.

The `Android.mk` file should contain two special lines when using the ParallelME Runtime. Between these two lines should come the normal content:

```
PM_JNI_PATH := $(call my-dir)/ParallelME
# Other Android.mk file contents, will be described later
include $(wildcard $(PM_JNI_PATH)/**/Android.mk)
```

These two lines will define where the ParallelME runtime is installed and call all the compilation scripts inside the `jni/ParallelME` folder, which will contain the runtime code.

The `Application.mk` file should specify at least the `clang` toolchain and the `c++\_static` STL. These are required when using the ParallelME Runtime, as it uses STL features that are not available on other STL implementations. An example file would contain:

```
NDK_TOOLCHAIN_VERSION=clang
APP_STL:=c++_static
# Other Application.mk directives, such as APP_ABI and APP_PLATFORM.
```

Now the `jni/ParallelME` folder should be created, and the runtime must be downloaded inside it by cloning the repository:

```
$ cd jni/ParallelME
$ git clone git://github.com/ParallelME/runtime.git # If not on a git repository
$ git submodule add git://github.com/ParallelME/runtime.git # If already in one
```

As the runtime repository already has a `Android.mk` file, the wildcard inclusion added to the `jni/Android.mk` file will include it and automatically compile the runtime when calling the `ndk-build` script. The runtime shared library will be called `libParallelMERuntime.so`, and the include folder will be at `jni/ParallelME/runtime/include`.

3.4 Copying the image into the project

The image that will be grayscaled needs to be put inside the `app/src/main/res/drawable` folder. This image should be in the JPEG or PNG format, and will be accessible in the source code by calling `R.drawable.<name_without_extension>`. Because of this, it is important that the file name doesn't contain spaces. In our case, we put a `rainbow.jpg` file inside the `res/drawable` folder, so that it can be accessed in the source code as `R.drawable.rainbow`.

[3] `https://developer.android.com/ndk/guides/ndk-build.html`
[4] `https://developer.android.com/ndk/guides/android_mk.html`
[5] `https://developer.android.com/ndk/guides/application_mk.html`

4 Implementing the Application

4.1 Adding the buttons and views to the layout

The application will have two buttons — one that displays the grayscaled image and one that displays the original image — and an `ImageView` that will be used to display the image. First, open the `res/values/strings.xml` and add the text that will be inside the two buttons:

```
<resources>
    <!- Other strings... ->

    <string name="button_grayscale">Grayscale</string>
    <string name="button_original">Original</string>
</resources>
```

Then, open the `res/layout/activity_main.xml` and replace the code inside the `RelativeLayout` that comes as default with the file with this:

```
<LinearLayout android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <LinearLayout android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">
        <Button android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/button_grayscale"
            android:onClick="showGrayscale" />
        <Button android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/button_original"
            android:onClick="showOriginal" />
    </LinearLayout>
    <HorizontalScrollView android:layout_width="wrap_content"
        android:layout_height="wrap_content">
        <ImageView android:id="@+id/image_view"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
    </HorizontalScrollView>
</LinearLayout>
```

This layout file will define first a vertical linear layout to organize the buttons and the image, then an horizontal linear layout with the two buttons inside it and finally a horizontal scroll view that contains the image view that will host the image.

Note that the first button uses the `button_grayscale` string we defined earlier and, when clicked, will call the `showGrayscale` function of the `MainActivity`. The second button uses the `button_original` instead and will call the `showOriginal` function when clicked.

The image view has an ID (`image_view`), so that it can be accessed in the code as the variable `R.id.image_view`, and is inside a horizontal scroll view to be able to scroll the image horizontally in case it is too big.

4.2 Implementing the MainActivity

As described in the last section, the main activity will need to implement two functions: `showOriginal` and `showGrayscale`. To implement these functions, we'll need two private members inside the activity: a reference to the `ImageView` defined in the layout, and an instance of the `GrayscaleOperator` class, a class that we'll write that will be responsible for calling the ParallelME Runtime code to process the image. These two members must be initialized in the `onCreate` function:

```
1  public class MainActivity extends AppCompatActivity {
2      private GrayscaleOperator operator;
3      private ImageView imageView;
4
5      @Override
6      protected void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.activity_main);
9
10         operator = new GrayscaleOperator();
11         imageView = (ImageView) findViewById(R.id.image_view);
12     }
13 }
```

The `imageView` object is just a reference to the view that we defined in the layout, and is retrieved by the `findViewById` method.

After that, the two functions called by the buttons must be implemented. The `showOriginal` function is very simple, and just loads the image as a bitmap (from the `R.drawable.<image_name>` resource described in section 3.4) and makes the image view display it:

```
1  public void showOriginal(View view) {
2      Bitmap image = BitmapFactory.decodeResource(getResources(), R.drawable.rainbow);
3      imageView.setImageBitmap(image);
4  }
```

The `showGrayscale` function is almost the same, but calls the `operator.grayscale()` function (that we'll implement shortly) on the bitmap before displaying it. This function is responsible for calling the grayscale algorithm implementation using the ParallelME Runtime.

```
public void showOriginal(View view) {
    Bitmap image = BitmapFactory.decodeResource(getResources(), R.drawable.rainbow);
    operator.grayscale(image);
    imageView.setImageBitmap(image);
}
```

4.3 Implementing the GrayscaleOperator Java class

The `GrayscaleOperator` class will be responsible to initialize the runtime and make the bridge between the Java application and the C++ code.

The runtime and the compiled OpenCL program will be stored inside a C++ structure. Because there is no way to store data per Java class instance directly in the C++ code, we'll employ a small hack and store a struct pointer inside the Java class. This pointer will contain the address of a dynamically allocated structure that can be accessed by the C++ code.

We'll define three native functions: a `nativeInit`, that will initialize the runtime and return the pointer to the C++ structure (as a long integer), a `nativeCleanUp`, that will release the C++ structure when the Java class is destroyed and a `nativeGrayscale`, the function that will execute the grayscale algorithm.

```
public class GrayscaleOperator {
    private long dataPointer = nativeInit();

    private native long nativeInit();
    private native void nativeCleanUp(long dataPointer);
    private native void nativeGrayscale(long dataPointer, Bitmap bitmap,
                                        int width, int height);

    // ...
}
```

Note that the dataPointer is initialized by calling `nativeInit` when the class is constructed.

The `nativeCleanUp` function needs to be called when the `finalize` java function is called. It supplies the dataPointer with the structure as the first parameter, and is responsible for deleting the structure.

```
@Override
protected void finalize() throws Throwable {
    nativeCleanUp(dataPointer);
    super.finalize();
}
```

A public `grayscale` function will exist to interface with the `nativeGrayscale` function, by supplying the dataPointer, the bitmap and the bitmap sizes as arguments:

```
public void grayscale(Bitmap image) {
    nativeGrayscale(dataPointer, image, image.getWidth(), image.getHeight());
}
```

Finally, the shared library that contains the implementation of the native methods must be loaded. This can be done by the following code (assuming a "libGrayscale.so" shared library):

```
static {
    System.loadLibrary("Grayscale");
}
```

4.4 Generating the JNI interface header

After implementing the Java code in the previous sections, hit the `Make Project` button and wait until the compilation finishes. The project should compile successfully. However, the application will not open because the native methods are still not implemented. To do this, enter the `jni` folder and generate the JNI header that specifies the native functions of the GrayscaleOperator class that should be implemented. This can be done by running the `javah` command. For example, if the `GrayscaleOperator` class is inside the `org.parallelme.samples.grayscale` package (and the `Make Project` command has been executed), the command would be (from inside the jni folder):

```
$ javah -classpath <path_to_android_sdk>/platforms/android-23/android.jar:../../../
build/intermediates/classes/debug org.parallelme.samples.grayscale.GrayscaleOperator
```

Note that the correct path to the Android SDK must be supplied. Here in the example I used the Android 23 version, but any version can be used, so use the one you downloaded when you installed the Android SDK packages back in section 3.2.

In the example, a file `org_parallelme_samples_grayscale_GrayscaleOperator.h` would be created, with the prototypes of the three native functions of the `GrayscaleOperator` class that need to be implemented in a .cpp file.

4.5 Implementing the kernel

Now it is time to implement the OpenCL kernel that will run the grayscale algorithm. This kernel will receive just one parameter, a pointer to the buffer containing the image, and update the pixel specified by the global ID:

```
__kernel void grayscale(__global uchar4 *image) {
    int gid = get_global_id(0);
    uchar4 pixel = image[gid];

    uchar luminosity = 0.21f * pixel.x
        + 0.72f * pixel.y + 0.07f * pixel.z;
```

```
    pixel.x = pixel.y = pixel.z = luminosity;

    image[gid] = pixel;
}
```

The kernel will get the pixel specified by the current global ID, run the algorithm from section 2 and save the results back to the image.

4.6 Implementing the native functions

To implement the native functions specified in the header file generated with the `javah` tool, we'll create a C++ file, in this case called `org_parallelme_samples_grayscale_GrayscaleOperator.cpp`. This file should include the header files and define the kernel source code. The easiest way to do this is to just supply the kernel source as a string in a global variable.

```
#include "org_parallelme_samples_grayscale_GrayscaleOperator.h"
#include <parallelme/ParallelME.hpp>
using namespace parallelme;

const static char gKernels[] =
    "__kernel void grayscale(__global uchar4 *image) { \n"
    "    int gid = get_global_id(0);                   \n"
    "    uchar4 pixel = image[gid];                    \n"
    "                                                  \n"
    "    uchar luminosity = 0.21f * pixel.x            \n"
    "        + 0.72f * pixel.y + 0.07f * pixel.z;      \n"
    "    pixel.x = pixel.y = pixel.z = luminosity;     \n"
    "                                                  \n"
    "    image[gid] = pixel;                           \n"
    "}                                                 \n";
```

Then, the data structure that the C++ code will use and return to java should be defined. This structure will just contain pointers to the ParallelME Runtime and the compiled OpenCL program.

```
struct NativeData {
    std::shared_ptr<Runtime> runtime;
    std::shared_ptr<Program> program;
};
```

After that, we are going to implement the `nativeInit` function. This function will get a pointer to the JavaVM (as the Runtime needs this on the constructor), create a new Runtime instance and create a Program instance that will contain the compiled OpenCL code (from the source we defined as a global variable before). The structure holding all this will be returned as a jlong.

```
JNIEXPORT jlong JNICALL
Java_org_parallelme_samples_grayscale_GrayscaleOperator_nativeInit
(JNIEnv *env, jobject self) {
    JavaVM *jvm;
    env->GetJavaVM(&jvm);
    if(!jvm) return (jlong) nullptr;

    auto dataPointer = new NativeData();
    dataPointer->runtime = std::make_shared<Runtime>(jvm);
    dataPointer->program = std::make_shared<Program>(dataPointer->runtime, gKernels);

    return (jlong) dataPointer;
}
```

The `nativeCleanUp` function will get the pointer and delete it.

```
JNIEXPORT void JNICALL
Java_org_parallelme_samples_grayscale_GrayscaleOperator_nativeCleanUp
(JNIEnv *env, jobject self, jlong dataLong) {
    auto dataPointer = (NativeData *) dataLong;
    delete dataPointer;
}
```

Finally, the `nativeGrayscale` function will run the grayscale algorithm. First, the function will get the data pointer, calculate the image size, create a Runtime Buffer as big as the input image and mark the bitmap to be copied into the buffer.

```
JNIEXPORT void JNICALL
Java_org_parallelme_samples_grayscale_GrayscaleOperator_nativeGrayscale
(JNIEnv *env, jobject self, jlong dataLong, jobject bitmap, jint width, jint height) {
    auto dataPointer = (NativeData *) dataLong;
    auto imageSize = width * height;
    auto bitmapBuffer = std::make_shared<Buffer>(Buffer::sizeGenerator(imageSize,
                                                                       Buffer::RGBA));
    bitmapBuffer->setAndroidBitmapSource(env, bitmap);
```

The `setAndroidBitmapSource` function saves a global reference to the bitmap, to be used later as the data source of the buffer. This bitmap will be copied right before the kernel execution in the target device, so it must be kept alive until the kernel executes.

After that, a task made of kernels from the program we created in the `nativeInit` function is created and it's specified that it will run one kernel: the `grayscale` kernel.

```
    auto task = std::make_unique<Task>(dataPointer->program);
    task->addKernel("grayscale");
```

Then, we set the config function callback. This callback sets the first (and only) parameter of the `grayscale` kernel to point to the buffer created earlier and the number of work items of the task to be the number of pixels of the image.

```
    task->setConfigFunction([=] (DevicePtr &device, KernelHash &kernelHash) {
        kernelHash["grayscale"]
            ->setArg(0, bitmapBuffer)
            ->setWorkSize(imageSize);
    });
```

Finally, the task is submitted to the runtime. Soon after that a call to `Runtime::finish` is made, to wait for the execution to finish. This is not mandatory, and another task callback (the finish callback) could be set to be called after the task finishes execution. However, for simplicity, we're calling the `Runtime::finish` function here. At the end, after the `Runtime::finish` function is called, the contents of the buffer are copied to the bitmap.

```
    dataPointer->runtime->submitTask(std::move(task));
    dataPointer->runtime->finish();

    bitmapBuffer->copyToAndroidBitmap(env, bitmap);
}
```

4.7 FInishing the Android.mk file

We partially implemented the `jni/Android.mk` file in section 3.3. Now we'll finish the implementation and add the rest of the missing code.

The `jni/Android.mk` file started by specifying where the ParallelME code was located:

```
PM_JNI_PATH := $(call my-dir)/ParallelME
```

Then, we specify the local path where the code is and include the `CLEAR_VARS` variable to clean all compilation variables before compiling our code.

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
```

After that, we specify the name of the compiled library. In this example we'll name it as "Grayscale", so that the shared library name will be "libGrayscale.so".

```
LOCAL_MODULE := Grayscale
```

Then we must specify the ParallelME include directories so that the compiler can find the `<parallelme/ParallelME.hpp>` header file.

```
LOCAL_C_INCLUDES := $(PM_JNI_PATH)/runtime/include
```

It is also important to specify optimization flags, enable warnings and enable C++14, used in this example. Note that we add the "-Wno-unused-parameter" flag here, as implementing the JNI functions normally doesn't use all the function parameters.

```
LOCAL_CPPFLAGS := -Ofast -Wall -Wextra -Werror -Wno-unused-parameter -std=c++14
```

Next we specify a dependency on the `ParallelMERuntime` shared library and the source files we reated.

```
LOCAL_SHARED_LIBRARIES := ParallelMERuntime
LOCAL_SRC_FILES := org_parallelme_samples_grayscale_GrayscaleOperator.cpp
```

Finally, we include all the `Android.mk` files inside the `ParallelME` folder to compile all dependencies.

```
include $(wildcard $(PM_JNI_PATH)/**/Android.mk)
```

5 Finished Application

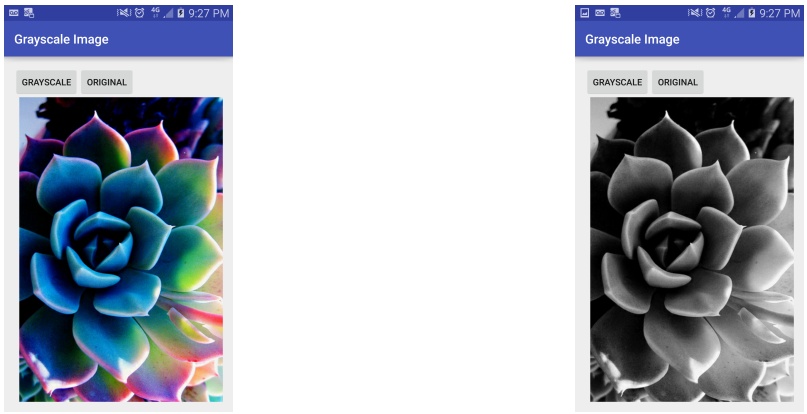

Figure 1: Finished application showing the original image to the left and the grayscaled image to the right.

The finished application implements a very efficient grayscale algorithm that can be seen in figure 1. The full source code for this example can be found at:
`https://github.com/ParallelME/samples/tree/master/Grayscale`.

Other ParallelME examples (not solely directly using the runtime) can be found at:
`https://github.com/ParallelME/samples`.