RenderScript
Writing high performance applications with Android

Ghilotti Giorgio
Matr. 765836, (giorgio.ghilotti@mail.polimi.it)

Report for the master course of Embedded Systems
Reviser: PhD. Patrick Bellasi (bellasi@elet.polimi.it)

Received: September 26, 2013

Abstract

This work is an overview of RenderScript(RS), Google's framework for Android developers that allows to perform intensive computation using heterogeneous architectures through a two-step compilation. This paper aims to describe how it works, the main components of an RS application, achievable performance, and why it should be used.

1 Introduction

In the last seven years GPUs have become useful, thanks to their greater FLOPS and memory bandwidth versus CPUs, for many more applications than just traditional graphics. They are really good for general data parallel tasks, high performance computing and supercomputing, and now these programmable GPUs are arriving on tablets and phones. Compared to desktop/server architectures, mobile device's CPU and GPU share the same pool of physical memory (see **Figure 1**), so they can transfer data to each other without pass through a PCI express bus, making more convenient managing the computational load on different processors. In mobile environment we may also have additional processors available, such as a camera ISP or programmable DSP; when possible we would like to use them because they are relatively fixed function and they may provide very good perf/watt.

Another important difference between mobile and desktop is the number of architectural solutions. In mobile there are several variants of ARM: with and without VFP, with and without NEON, and with various register counts. Beside ARM, there are other CPU architectures like x86, several GPU architectures, and even more DSP architectures. For that reason the more a code is optimized to run very well on a specific device, the less it performes on any other certain type of SoC.

Android provides a platform-indipendent framework called RenderScript that allows us to write high performance computational code. This means that developers don't have to worry about on which kind of architecture the code will run, they just have a computation needs to run very fast and the RS runtime will mind of using the best processor available at that moment. Since RS is a fairly new component in Android, it is subject of continuous improvements, so we don't aim here to give an exaustive technical manual. Indeed, while you read this paper, new features could have been added and others could be deprecated. For up-to-date informations we suggest to read the documentation on the official site for Android developers.

In section 2 we present the main features of the framework and explain when and why it should be used. Section 3 explains the procedure to write a RS file. In Section 4 are illustrated achievable performance with RS. Finally in section 5 we try to do some remarks.

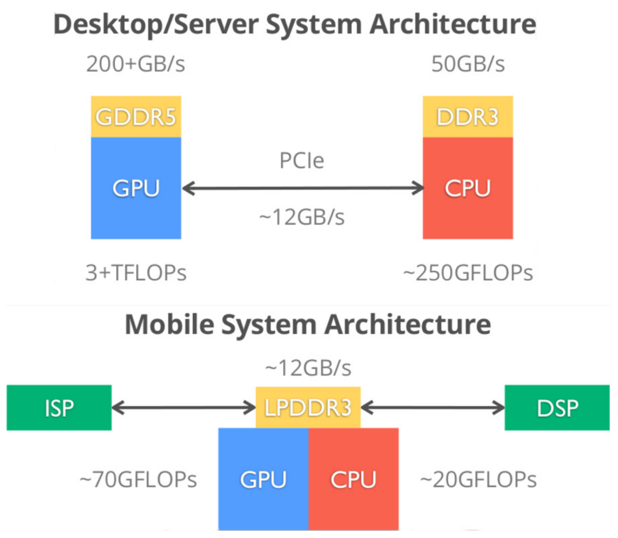


Figure 1: desktop/server architecture vs mobile architecture.

2 Why Renderscript

The purpose of RenderScript is to develop high performance applications for a wide variety of SoCs without sacrificing portability. RS framework provides a platform-indipendent computation engine that operates at native level.

As explained in [], we can point out three main goals, from most to least important:

Portability: application code should be able to run across all devices, even those with radically different hardware.
The first thing one notices about the RS API is that it is focused on the system: one doesn't get a list of devices, or a big collection of device properties so you don't have to try to figure out at run time which device you should use. The RS runtime handles that for you. You simply have a computation that you want to run quickly and the RS engine will manage to execute it on the best processor available.

Performance: get as much performance as possible within the costraints of portability.
RS code is firtly compiled to an intermediate and architecture-indipendent bytecode by the llvm compiler that runs as a part of an Android build. When your app runs on a device, the bytecode is compiled again just-in-time to machine code by another llvm compiler that resides on the device and optimizes the code for the particular target. Google works with SoC vendors and gets drivers for their GPUs, DSPs, ISPs, and makes them available on tablets and phones. This way, the RS runtime knows about whatever processors and capabilities it can use.

Usability: simplify development as much as possible.
Java classes are reflected from RS code for an easy integration with existing applications. This allows the control of RS runtime management and execution directly through Java APIs without relying on JNI or some other low-level interface. Furthermore, a collection of RS files (called intrinsics) is provided for the standard graphic computational functions like convolution, YUV to RGB conversion and Gausian blur filtering.

3 Writing RenderScript file

RS files are written in a C99 derived language (C99 is the standard C from 1999, the current standard is C11, since 2011). The main component of an RS file is a special function called kernel, here is the place where we have to write the computation we want to performe as fast as possible. Kernel functions take as input a special object called Allocation, which is nothing but a memory space with a number of elements of the same type (see **Table 1**). The normal behavior of RS when a kernel function is called from a java application, is to launch it across all the elements inside the allocation passed by input.
Some key features of the RS runtime libraries include:

- Memory allocation request features.
- A large collection of math functions with both scalar and vector typed overloaded versions of many common routines.
- Conversion routines for primitive data types, vectors and matrix routines, date and time routines.
- Data types and structures to support the RenderScript system such as Vector types for defining two-, three-, or four-vectors.
- Logging functions.

See RS runtime API reference for more information on the available functions.

Object	Description
Element	An Element is essentially a C type; it can be a scalar type (e.g. an int), a vector type (e.g. an int4 or a float4) or a C struct declared on your RS code. It represents one cell of a memory allocation.
Allocation	An Allocation is a portion of memory containing a collection of a single type of element. It provides the memory to get data from java into RenderScript, making it processesable by one ore more kernels.
Type	A Type is a memory allocation template and constitutes the size of the allocation. It consists of an element and one or more dimentions and describes the layout of the memory. You can assign the X, Y, Z dimensions to any positive integer value within the constrains of available memory. So you can have a vector, a matrix or a cube of elements. That allows to do safety checking on copies and kernel launches and also to control how much parallel work actually gets launched by a kernel.

Table 1: android object types for RS.

3.1 RenderScript Kernel

An RS kernel is the portion of code that requires extensive computational horsepower and that we want to execute at high performance. It tipically resides in a *.rs* file (also called script) in the *project_root/src/* directory. Each script contains is own set of kernels, variables and functions. A script can contain:

- A pragma declaration (*#pragma version(1)*) that declares the RenderScript version used in the script. So far 1 is the only value available.
- A pragma declaration (*#pragma rs java_package_name (com.example.app)*) that indicates the package for the java classes reflected from this script.
- Some invokable functions, i.e. single-threaded fuctions often used for initial setup.

- A certain number of script globals equivalent to a global variable in C and almost used to pass parameters from the java side to RS kernels.
- Some number of compute kernels, that are parallel fuctions that executes across every Element within an Allocation.
- An optional *init()* function, a special type of invokable function that runs when the script is first instantiated. This allows for some computation to occur automatically at script creation.
- static script globals and functions. They can be used normally in any kernel or invokable function in the script, but are not exposed to the Java API. If a script global or function does not need to be called from Java code, it is highly recommended that they are declared static.

A simple kernel may look like the following:

```
uchar4 __attribute__((kernel)) invert
  (uchar4 in,uint32_t x, uint32_t y) {
        uchar4 out = in;
        out.r = 255 - in.r;
        out.g = 255 - in.g;
        out.b = 255 - in.b;
        return out;
}
```

The attribute kernel applied to the function prototype denotes that is an RS kernel instead of an invokable fuction. The "in" argument is a special argument automatically filled in based on the input Allocation passed to the kernel launch. The returned value is automatically written to the appropiate location in the output Allocation. A kernel may not have more than an input and an output Allocation. A kernel may access to the coordinates of the current execution using x, y and z argument. These are optional, but the type must be uint32_t.

The code written in RenderScript generates reflected Java code in Android framework in order to menage the RenderScript lifetime eliminating JNI glue code. We explain this deeper in the following subsections.

3.2 Reflected Layer

The reflected layer is a set of classes that the Android build tools generates to allow access to the RS runtime from the Android framework. This layer also provides methods and constructors that allow you to allocate and work with memory for pointers that are defined in your RS code. The following list describes the major components reflected:

- Every *.rs* file is generated into a class named *project_root/gen/package_name/ScriptC_renderscript_filename* of type ScriptC. This file is the *.java* version of your *.rs* file, which you can call from the Android framework. This class contains the following items reflected from the *.rs* file:
  - Non-static functions. Functions cannot have a return value, because the RenderScript system is designed to be asynchronous. When your Android framework code calls into RenderScript, the call is queued and executed when possible. This restriction allows the RS system to run without constant interruption and increases efficiency. If functions were allowed to have return values, the call would be blocked until the value was returned. If you want the RS code to send a value back to the Android framework, use the *rsSendToClient()* function.
  - Non-static global RenderScript variables. Getter and Setter methods are generated for each variable so you can read and write them from Android framework. If a global variable is initialized at the RenderScript runtime layer, those values are used to initialize the corresponding values in the Android framework layer. If global variables are marked as const, then a set method is not generated.
  - Global pointers. A get method and a special method named *bind_pointer_name* (instead of a set() method) are also generated. The second method allows you to bind the memory that is allocated in the Android VM to the RS runtime (you cannot allocate memory in your *.rs* file).
- A struct is reflected into its own class named *<project_root>/gen/package/name/ScriptField_struct _name*. This class represents an array of the struct, which allows you to allocate memory for one or more instances of this struct.

3.3 Using RenderScript from java code

Managing RS lifecycle from Android side relies on the API classes located in the android.renderscript, for devices running Android 3.0 (API level 11) and higher, or in the android.support.v8.renderscript package through a Support Library, for devices running Android 2.2 (API level 8) and higher. Most applications follow the same basic usage patterns:

1. **Initialize a RenderScript context.** The RenderScript context, created with *create(Context)*, ensures that RenderScript can be used and provides an object to control the lifetime of all subsequent RenderScript objects. You should consider context creation to be a potentially long-running operation, since it may create resources on different pieces of hardware; it should not be in an application's critical path if possible. Typically, an application will have only a single RenderScript context at a time.

2. **Create at least one Allocation to be passed to a script.** An Allocation is a RenderScript object that provides storage for a fixed amount of data. Kernels in scripts take Allocation objects as their input and output, and Allocation objects can be accessed in kernels using *rsGetElementAt_type()* and *rsSetElementAt_type()* when bound as script globals. Allocation objects allow arrays to be passed from Java code to RS code and vice-versa. Allocation objects are typically created using *createTyped(RenderScript, Type)* or *createFromBitmap(RenderScript, Bitmap)*.

3. **Create whatever scripts are necessary.** There are two types of scripts available when using RenderScript:

   - **ScriptC**: These are the user-defined scripts as described in section 3.1 above. Every script has a Java class reflected by the RS compiler in order to make the script easy to access from Java code; this class will be named *ScriptC_filename*. For example, if the kernel above was located in *invert.rs* and a RenderScript context was already located in mRS, the Java code to instantiate the script would be:

```
ScriptC_invert invert =
 new ScriptC_invert(mRenderScript);
```

   - **ScriptIntrinsic**: These are built-in RenderScript kernels for common operations, such as Gaussian blur, convolution, and image blending. For more information, see the subclasses of ScriptIntrinsic.

4. **Populate Allocations with data**. Except for Allocations created with android.renderscript, an Allocation will be populated with empty data when it is first created. To populate an Allocation, use one of the copy methods in Allocation.

5. **Set any necessary script globals**. Globals may be set using methods in the same *ScriptC_filename* class with methods named *set_globalname*. For example, in order to set an int named elements, use the Java method *set_elements(int)*. RenderScript objects can also be set in kernels; for example, the *rs_allocation* variable named *lookup* can be set with the method *set_lookup(Allocation)*.

6. **Launch the appropriate kernels**. Methods to launch a given kernel will be reflected in the same *ScriptC_filename* class with methods named *forEach_kernelname()*. These launches are asynchronous, and they will be serialized following the order in which they are launched. Depending on the arguments passed to the kernel, the method will take either one or two Allocations. By default, a kernel will execute over the entire input or output Allocation; to execute over a subset of that Allocation, pass an appropriate *Script.LaunchOptions* as the last argument to the *forEach method*.

   Invoked functions can be launched using the *invoke_function_name* methods reflected in the same *ScriptC_filename* class.

7. **Copy data out of Allocation objects**. In order to access data from an Allocation from Java code, that data must be copied back to Java buffers using one of the copy methods in Allocation. These functions will synchronize with asynchronous kernel and function launches as necessary.

8. **Tear down the RenderScript context**. The RenderScript context can be destroyed with *destroy()* or by allowing the RenderScript context object to be garbage collected. This will cause any further use of any object belonging to that context to throw an exception.

4 Performance

So far, there aren't many results available about the performance achievable from RenderScrit, because of the young age of the framework. In [4] we can find a comparison between the three programming models in Android: Java in SDK, C++ in NDK and Renderscript. In order to compare them a representative application is developed in all the models, and run in an Android dual-core tablet with Honeycomb 3.2 OS. The application chosen is *Balls*, which simulate the movement of several hundred of bodies according to the gravity to the ground and repulsion among them. **Figure 2** shows the best achieved performance. The device sets 60 as the maximal FPS. When the computation for one frame is faster, the device still gives 60 FPS since that is the maximal. We can see that the NDK and RS versions are quite close in performance, while the SDK version lags far behind.

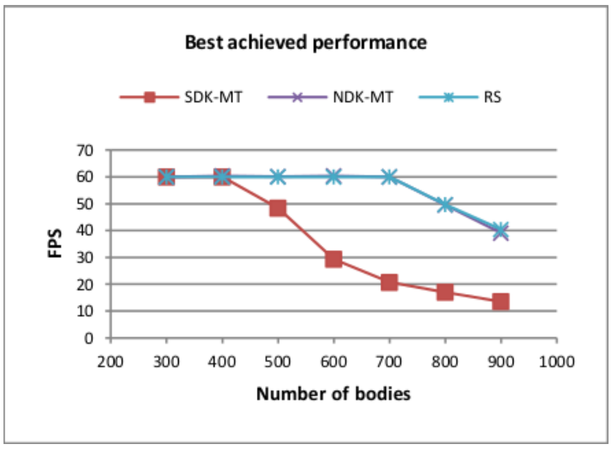


Figure 2: best achieved performance.

However this work is quite old since in the meantime Google spent many efforts to improve performance [5], especially for intrinsic kernels, as we can see in **Figure 3**, which shows the evolution of RS performance on Galaxy Nexus across different Android platform versions.

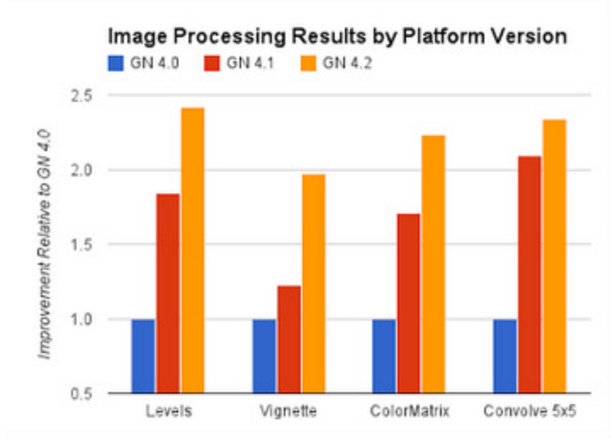


Figure 3: performance evolution across different Android versions.

5 Conclusion

RenderScript seems to be very useful when your goal is writing an application that require high-performance computational tasks in order to develop a single program that runs well on a large number of different target architectures. In the contrary, if you mean to optimize an application for a specific target-architecture, RS is not your tool, since it decrease peak performance on specific target platform in order to guarantee very good average performace across a larger number of SoCs. Anyway, the choice to use RS should be well pondered, since this framework brings some serious drawbacks. First of all it increases project's complexity due to the introduction of a new set of APIs that you have to learn. While the syntax of C99 is well known, the learning curve of RS system is pretty steep, because its APIs are not. Besides the programmer must manage memory allocation manually from java layer. Another issue is that RenderScript can potentially execute on processors other than the main CPU (such as the GPU) and, if this occurs, debugging becomes a problem. Given the nature of RenderScript and how it works with multiple cores, this is not a huge surprise, but this can make finding and eliminating bugs more difficult. Ultimately, the choice to use RenderScript, the NDK, or stay within the realm of Java is entirely dependent on the project's specifications. It's an application design decision with considerable ramifications, it effects what programming language you can use, what devices your application can run on, and how complicated your source project is from a maintenance perspective.

References

[1] Google Inc. android developers. `http://developer.android.com/index.html`. Accessed: 2013-09.

[2] Google Inc. high performance applications with renderscript. `https://developers.google.com/events/io/sessions/331954522`. Accessed: 2013-09.

[3] Shane Conder and Lauren Darcey. Writing native android code ndk vs. renderscrip. `http://www.developer.com/ws/android/development-tools/writing-native-android-code-ndk-vs.-renderscript.html`. Accessed: 2013-09.

[4] Xi Qian, Guangyu Zhu, and Xiao-Feng Li. Comparison and analysis of the three programming models in google android. In *First Asia-Pacific Programming Languages and Compilers Workshop (APPLC)*, 2012.

[5] R Jason Sams. Android Developers Blog evolution of renderscript performance. `http://android-developers.blogspot.it/2013/01/evolution-of-renderscript-performance.html`. Accessed: 2013-09.

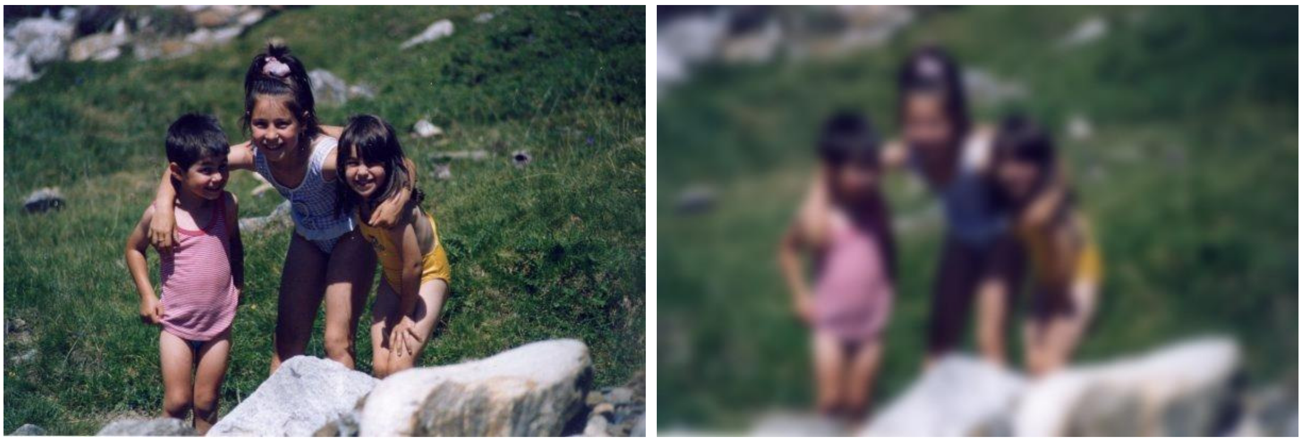
[6] Dave Smith and Jeff Friesen. Working with android ndk and renderscript. In *Android Recipes*, pages 743–804. Springer, 2012.

Appendix A Example of an intrinsic RenderScript kernel usage

Suppose we want to apply a Gaussian blur to a Bitmap image. Since Android gives an intrinsic script to implement this task we don't have to write a RenderScript file.
Here is the auto-esplicative code to use on Java side in order to correctly utilize Android API.

```
// Create a Context
RenderScript mRS = RenderScript.create(this);

// Create a RenderScript Allocation to hold the input image.
Allocation inputAllocation = Allocation.createFromBitmap(mRS, myInputBitmap,
        Allocation.MipmapControl.MIPMAP_NONE,
        Allocation.USAGE_SHARED |
        Allocation.USAGE_GRAPHICS_TEXTURE |
        Allocation.USAGE_SCRIPT);

// Create an output Allocation, notice the USAGE flags are different
Allocation outputAllocation = Allocation.createFromBitmap(mRS, myOutputBitmap,
        Allocation.MipmapControl.MIPMAP_NONE,
        Allocation.USAGE_SHARED |
        Allocation.USAGE_SCRIPT);

// RenderScript has built in support for Blur so use that
ScriptIntrinsicBlur mBlur;

// First, we need to create the intrinsic
mBlur = ScriptIntrinsicBlur.create(mRS, Element.U8_4(mRS));

// And now we configure the intrinsic to perform our blur
mBlur.setRadius(20.f);
mBlur.setInput(inputAllocation);

// Now run the blur
mBlur.forEach(outputAllocation);

// Copy the output to our bitmap if necessary
outputAllocation.copyTo(myOutputBitmap);
```

And this is our image before and after the filter application:



Appendix B   Example of a RenderScript kernel writing and usage

In this example we compute the luminance histogram of an image, but we want to do it in two separate steps:

- in the first step the image is splitted in stripes and for each one the histogram is computed.
- the second step sums the histograms obtained from the previously step.

For implementing the program in the way described above we need to write two different kernel functions.

This is the code for the RenderScript runtime framework, which resides in our .rs file:

```
// The histogram script requires a number of buffers

// The source and destination images.
rs_allocation gSrc;
rs_allocation gDest;

// A buffer for the intermediate sums
// Integer values, [256][steps] in size
rs_allocation gSums;

// Final sum buffer, Integer, [256] cells in size
rs_allocation gSum;

// The height and width of the image
int gWidth;
int gHeight;

// The step, which is the number of lines processed by each thread
int gStep;

// The number of steps in total, roughly height / step
int gSteps;

// The start of the first kernel
void __attribute__((kernel)) pass1(int in, uint x, uint y) {
  // Note that x and y will indicate the coordinates of the pixel being processed
  // This kernel will be run on a range of x = [0] and y = [0 to steps]
  // Clear our output buffer for this thread.
  for (int i=0; i < 256; i++) {
    // Set the value at i,y to 0
    rsSetElementAt_int(gSums, 0, i, y);

  }

  // Iterate over our image
  for (int i= 0; i < gStep; i++) {
    int py= y*gStep + i;          // Compute the y coordinate in the image
    if (py>= gHeight) return;    // Might be out of range if this is the last step

    // Walk one scanline
    for (int px=0; px < gWidth; px++) {
      // Get the pixel and convert to luminance
      uchar4 c = rsGetElementAt_uchar4(gSrc, px, py);
      int lum = (77 * c.r + 150 * c.g + 29 * c.b) >> 8;

      // Add one to the bucked for this luminance value
      int old = rsGetElementAt_int(gSums, lum, y);
      rsSetElementAt_int(gSums, old+1, lum, y);
    }
  }
}

// This kernel is run on the Sum allocation , so its called once
// for each of the 256 levels
// Note, this is a 1D kernel
int __attribute__((kernel)) pass2(uint x) {
  int sum = 0;

  // For this level , add in the sum from each of the
  // separate partial sums
  for (int i=0; i < gSteps; i++) {
    sum += rsGetElementAt_int(gSums, x, i);
  }

  // Return the sum for this level. Since this is a kernel
  // return value, it will automatically be placed in the allocation.
  return sum;
}

// This is an invokable function. It will be called single threaded
void rescale() {
  // Find our largest bucket value
  int maxv = 0;
  for (int i=0; i < 256; i++) {
    maxv = max(maxv, rsGetElementAt_int(gSum, i));
  }

  // Compute the rescale value to to convert bucket values into bar heights.
  float overMax = (1.f / maxv) * gHeight;

  for (int i=0; i < 256; i++) {
    int t = rsGetElementAt_int(gSum, i);
    t = gHeight - (overMax * rsGetElementAt_int(gSum, i));
    t = max(0, t);
    rsSetElementAt_int(gSum, t, i);
  }
}
```

This is the code we have to write in the Java application that allows to launch our scripts:

```
// Create and load the script
mScript = new ScriptC_histogram(mRS);

// Setup the globals
mScript.set_gWidth(width);
...

// Create the [256][steps] buffer of integers for the partial sums
Type.Builder tb = new Type.Builder(mRS, Element.I32(mRS));
tb.setX(256).setY(steps);
Type t = tb.create();
```

```
mSums = Allocation.createTyped(mRS, t);

// Create the 1D [256] buffer for the final Sums
mSum = Allocation.createSized(mRS, Element.I32(mRS), 256);

// Set the buffers for the script
mScript.set_gSums(mSums);

// This first pass should be clipped because we want [step] threads
// not [256][step] threads

// To do this we have the ability to clip our launch
// using a LaunchOptions class
Script.LaunchOptions lo = new Script.LaunchOptions();

// Set the range in the X dimension to be 0 to 1
// Note: the end is exclusive so this says only run X value 0
lo.setX(0, 1);

// Run the kernel with our launch options
// This will spawn one thread per Y coordinate, each with X=0
mScript.forEach_pass1(mSums, lo);

// Once the first pass is complete, we need to add up our partial sums

// The pass2 launch is unclipped.
// It spawns one thread per cell in the mSum buffer
// for a total of 256 threads.
mScript.forEach_pass2(mSum);

// Finally, we call our rescale function
mScript.invoke_rescale();
```

This is an example of an image with its luminance histogram: