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 17, 2013

Abstract

This is a normal text in 10pt type size and 12pt line spacing. Place here a summary of your report. Focus on what is the problem and how you propose to tackle it. Briefly resume achieved results.
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras dapibus. Donec fermentum mauris non lectus. Nulla porttitor pede id ante. Donec in erat pellentesque erat ultrices rutrum. Pellentesque id tellus. Donec vel mi non dui adipiscing tempor. Nunc laoreet pede ut lectus. Aenean aliquam quam a sem. Duis at elit? Ut dolor massa, dictum vel; convallis quis, ullamcorper id, urna.

1 Introduction

In the last seven years GPUs have become useful, due to their greater FLOPS and memory bandwidth versus CPUs, for a lot more applications than just traditional graphics. They are really good for general data parallel tasks, high performance computing and supercompiuting, and now these programmable GPUs are arriving on tablets and phones. In respect to desktop/server architectures, in mobile devices we have that CPU and GPU share the same pool of physical memory (see **Figure 1**), so they can transfer data each other without pass through a PCI express bus, making more convenient menaging the computational load on different processors. We also may have additional processors available to us in mobile, for example a camera ISP or a programmable DSP as well, and you would like to use these processors when possible, 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 you have several variants of ARM: with and without VFP, with and without NEON, and with various register counts. Beyond ARM, there are other CPU architectures like x86, several GPU architectures, and even more DSP architectures. For that reason it happens that the more a code is optimized to run very well on a specific device, the less it performes on an other certain type of SoC.

Android give us a framework platform-indipendent called RenderScript in order to write high performance computational code. This means that developers don't have to take care about upon what 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.

In section 1 we present the main features of the framework and explain when and why it could be used. In section 2 is illustrated the procedure to write a good code to perform our app at best. In section 3 we analyse performance achievable with RS. Eventually in section 4 conclusions are

2 Why Renderscript

The purpose of RenderScript is to develop high performance applications for a wide variety of SoCs without sacrificing performance and portability. RenderScript framework provides a platform-indipendent computation engine that operates at native level.
We can point out three main goals, from most to least important:

Portability: application code needs to be able to run across all devices, even those with radically different hardware.
The first thing you will notice about the RenderScript API is that is focused on the system: you don'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 something else will try to put that on the best processor it can. This gives developers a consistent target that will run well across any SoC.

Performance: to get as much performance as possible within the costraints of portability.
Your RS code is compiled a first time to 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 then 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 get drivers for their GPUs, DSPs, ISPs, and makes those available on tablets and phones. This way, the RS runtime knows about whatever processors and capabilities it can use.

Usability: to simplify development as much as possible.
Java classes are reflected from RS code for easy integration with existing applications. This allows the control of RS runtime management and execution directly from Java APIs without relying on JNI or some other low-level interface. Furthermore a collection of RS files (called intrinsics) are provided for the more usual computational functions like convolution, YUV to RGB conversion and Gausian blur filtering. For complete documentation on the Android framework APIs, see the android.renderscript package reference.

Instead the primarily disadvantages are:

Development complexity: RenderScript introduces a new set of APIs that you have to learn in order to use them.

Debugging visibility: RenderScript can potentially execute (planned feature for later releases) on processors other than the main CPU (such as the GPU),so if this occurs, debugging becames more difficult.

3 Writing Renderscript file

High-performance compute kernels are written in a C99-derived language.
They compute the function defined across all the elements of the allocation passed like input. Some key features of the RenderScript 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 on RS runtime API reference for more information on the available functions.
Performance critical kernels are written in a C99 based language.

3.1 Basic RS

RenderScript allows to write special functions called kernel functions, each of which have the porpuse to compute the same computation over a large amount of data of the same type in parallel mode.
An element is essentially a C type that 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 RenderScript code. It represents one cell of a memory allocation
An Allocation is nothig but a portion of memory containing a collection of a single type of element. It provides the memory to get data from java into RenderScript, so it can be processed by one ore more kernels.
A type is a memory allocation template and constitute the size of the allocation. It consists of an element and one or more dimentions and describes the layout of the memory but does not allocate the memory for the data that describes. You can assigne the X, Y, Z dimensions to any positive integer value within the costrains 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.

3.2 RenderScript Kernel

A RenderScript 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 that are single-threaded fuction 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 RenderScript kernels.
- Some number of compute kernels, that is a parallel fuction 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 those be 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 a RenderScript 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 value returned 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.

3.3 Reflected Layer

The reflected layer is a set of classes that the Android build tools generate to allow access to the RenderScript 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 RenderScript 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
  - Non-static global RenderScript variables. Getter and Setter metgods are generated for each variable so you can read and write them from Android framework. If a global variable is inizialized at the RenderScript runtime layer

3.4 Using RenderScript from java code

4 Performance

Because to keep the portability we can obtain good average performance to the detriment of the peak performance.

5 Conclusion

This is a citation [1] and here is another citation [2]. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
And this is the reference to a single column figure (see **Figure ??**). Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus dapibus convallis odio. Nunc sollicitudin laoreet ante! Vivamus dictum euismod orci.

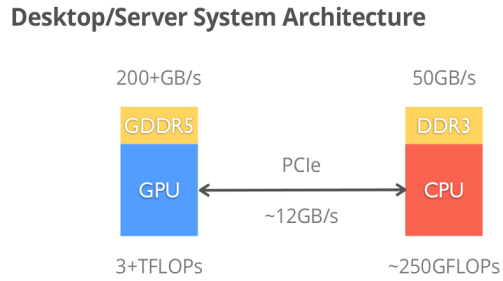

Figure 1: Some single-column figure caption.

And this is the reference to a single column figure (see **Figure 2**).

References

[1] Ramsey, N.: Learn technical writing in two hours per week. Technical report, Harvard University (2006)

[2] Hughes, S.P.J.J.L.J.: How to give a good research talk. SIGPLAN Notices (1993)

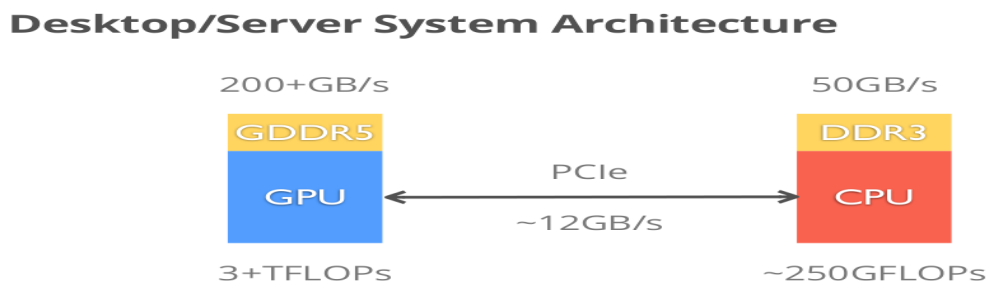

Figure 2: Some wide-figure caption.