Android Renderscript

Stephen Hines, Shih-wei Liao,
Jason Sams, Alex Sakhartchouk
srhines@google.com
November 18, 2011
Outline

- Goals/Design of Renderscript
- Components
  - Offline Compiler
  - Online JIT Compiler
  - Renderscript Runtime
- Java Reflection + rsForEach
- HelloCompute Example
- LLVM Challenges
  - Source Differences
  - Adventures in AST Annotation (Reference Counting)
- Conclusions & Future Work
Renderscript Goals

● Develop a solution to providing a solid 60fps for our application frameworks
  ○ Has to work well with Dalvik
  ○ Has to work with existing hardware
  ○ Has to scale to future hardware without requiring developers to rewrite their applications
● Developed applications must be portable across a variety of hardware. ARM v5, v7, NEON, x86, SSE, GPUs, DSPs
● Good performance across all devices instead of peak performance for one device at the expense of others
● Build a forward looking 3D graphics and compute API
Renderscript Design Principles

● Portability
  ○ Applications need to be able to run on a wide range of hardware without changes
  ○ Applications should be able to take advantage of new instructions and processors without requiring a recompile or code changes

● Performance
  ○ Should be able to utilize advanced vector operations
  ○ Should be able to utilize non-CPU processors

● Usability
  ○ Within the above constraints what can we do to help developers?
Portability

To achieve portability, some code must be compiled on the device. This may happen either at install time, runtime, or possibly both.

- Leverage LLVM bitcode as our on-device portable format
- Generated by a Clang-based tool from C99 scripts running as part of the SDK build process

LLVM bitcode is not 100% portable

- Endian (we use little endian)
- Alignment (use size of type - i.e. 8-byte aligned double)
- sizeof(long) == 8 to match Java
Performance

- Started with C99 due to its friendliness with compiling to a variety of HW targets and achieving good performance
- SMP primitives (forEach) built into the API
- Memory model designed to allow multiple memory spaces
  - Explicit sync points between memory spaces
  - Only script space is directly visible to the user
- Runtime is asynchronous with the application's Dalvik code
  - All communication goes through FIFOs
- Runtime is designed so the developer is not aware of which processor a given script is running on
Usability

Our goal is to make this as easy to use for developers as possible:

● Portability and performance constraints
● Complicated HW-specific features such as local memory and thread launch types are deliberately not exposed
● We don't allow developers to control which processors their apps run on
  ○ This ties apps to specific HW
  ○ Creates forward portability problems
● Feature set is CPU like, not "mobile" GPU like.
  ○ Recursion, function pointers, full IEEE 754 fp32, fp64, ...
Renderscript Components

- Offline compiler (llvm-rs-cc)
  - Convert script files into portable bitcode and reflected Java layer
- Online JIT compiler (libbcc)
  - Translate portable bitcode to appropriate machine code (CPU/GPU/DSP/...)
- Runtime library support (librs)
  - Manage scripts from Dalvik layer
  - Also provide basic support libraries (graphics drawing functions, etc.)
Offline Compiler: llvm-rs-cc

- Leverage Clang abstract syntax tree (AST) to reflect information and functionality back to Java layer
- Embeds metadata within bitcode (type, ...)
- Performs aggressive machine-independent optimizations on host before emitting portable bitcode
- All bitcode supplied as a resource within .apk container
Offline Compiler Flow

- foo.rs
- llvm-rs-cc
- LLVM codegen
- foo.bc
- Compiled with final .apk
- Compiled with rest of application source * .java files
- Reflection
- ScriptC_foo.java
- ScriptField_s.java
- Bundled with final .apk
Online JIT Compiler: libbcc

- Based on LLVM
  - Currently supports ARM and x86
  - Future targets include GPU/DSP
- Performs target-specific optimizations and code generation
- Provides hooks for runtime to access embedded metadata
- Links dynamically against runtime library functions (graphics, math)
- Caches JITted scripts to improve startup time
- Uses MC CodeGen to generate .o
- librsloader is used to load the .o file
- libbccinfo provides bitcode translator + metadata extraction
Online JIT Compiler Flow

- APK file
  - foo.bc
  - ScriptC_foo.java
  - ScriptField_s.java
  - App Java source files
  - ... System Library .bc files
  - libbcc + librslloader
  - Dalvik JIT

- GPUs/DSPs
- Multicore CPUs
Renderscript Runtime: libRS

- Manages *Scripts* and other Renderscript objects
- Provides implementation of runtime libraries (math, time, drawing, ref-counting, ...)
- Allows allocation, binding of objects to *Script* globals
- Message-passing between control and runtime
Non-static global variables and functions are automatically reflected for use with Java app (control-side).

- Functions like `set_GlobalName()`
  - Set a runtime global to a new value
  - Not reflected for `const` variables

- Corresponding `get_GlobalName()`
  - Returns the most recently set control-side value
  - Starts out as initialized script-side value (zero-init is on by default because of C99)
Java Reflection (continued)

- Functions like `bind_GlobalPtr()`
  - Allows control side to bind `Allocations` to runtime global pointers
  - Implements call-by-reference (as opposed to `set()`'s call-by-value)
  - Pointer types also reflect `get()` functions appropriately (that return the most recently bound `Allocation`)
- Functions like `invoke_FuncName()`
  - Allows control side to execute a particular runtime function (complete with parameter passing)
Explicit parallelism (based on rs_allocation dimensions)

Invokes `root()` function of script on each cell in allocation

- `void root(const void *ain, void *aout,
  const void *usrData,
  uint32_t x, uint32_t y)`

ICS - Reflect a Java helper to do parallel launch

- `void forEach_root(Allocation ain,
  Allocation aout)`

Need at least 1 of input/output allocation (determines launch dimensions)

ICS - Improved type and dimensionality verification in reflected Java helper
Also available in rs_core.rsh to the C99-based scripts:

```c
extern void __attribute__((overloadable))
  rsForEach(rs_script script,
            rs_allocation input,
            rs_allocation output,
            const void * usrData,
            size_t usrDataLen);

extern void __attribute__((overloadable))
  rsForEach(rs_script script,
            rs_allocation input,
            rs_allocation output,
            const void * usrData,
            size_t usrDataLen,
            const rs_script_call_t *);
```
HelloCompute Example

- Available in Android SDK samples
- Converts a bitmap image to grayscale
- Exploits parallelism by using rsForEach on every pixel of an allocation (simple dot product of RGB values)
- mono.rs -> mono.bc + reflected to ScriptC_mono.java
#pragma version(1)
#pragma rs java_package_name(com.example.android.rs.hellocompute)

const static float3 gMonoMult = {0.299f, 0.587f, 0.114f};

void root(const uchar4 *v_in, uchar4 *v_out, uint32_t x, uint32_t y) {
  float4 f4 = rsUnpackColor8888(*v_in);

  float3 mono = dot(f4.rgb, gMonoMult);
  *v_out = rsPackColorTo8888(mono);
}
package com.example.android.rs.hellocompute;

import android.renderscript.*;
import android.content.res.Resources;

public class ScriptC_mono extends ScriptC {
    // Constructor
    public ScriptC_mono(RenderScript rs, Resources resources, int id) {
        super(rs, resources, id);
        __U8_4 = Element.U8_4(rs);
    }

    private Element __U8_4;
    private final static int mExportForEachIdx_root = 0;
    public void forEach_root(Allocation ain, Allocation aout) {
        // check ain
        if (!ain.getType().getElement().isCompatible(__U8_4)) {
            throw new RSRuntimeException("Type mismatch with U8_4!");
        }
    }

    ...
}
// check aout
if (!aout.getType().getElement().isCompatible(__U8_4)) {
    throw new RSRuntimeException("Type mismatch with U8_4!");
}

// Verify dimensions
Type tIn = ain.getType();
Type tOut = aout.getType();
if ((tIn.getCount() != tOut.getCount()) ||
    (tIn.getX() != tOut.getX()) ||
    (tIn.getY() != tOut.getY()) ||
    (tIn.getZ() != tOut.getZ()) ||
    (tIn.hasFaces() != tOut.hasFaces()) ||
    (tIn.hasMipmaps() != tOut.hasMipmaps()) ) {
    throw new RSRuntimeException("Dimension mismatch between input and output parameters!");
}
forEach(mExportForEachIdx_root, ain, aout, null);
public class HelloCompute extends Activity {

    private void createScript() {
        mRS = RenderScript.create(this);

        mInAllocation = Allocation.createFromBitmap(mRS, mBitmapIn,
            Allocation.MipmapControl.MIPMAP_NONE,
            Allocation.USAGE_SCRIPT);
        mOutAllocation = Allocation.createTyped(mRS,
            mInAllocation.getType());

        mScript = new ScriptC_mono(mRS, getResources(), R.raw.mono);

        mScript.forEach_root(mInAllocation, mOutAllocation);
        mOutAllocation.copyTo(mBitmapOut);
    }

    ...
}
LLVM Challenges

● Bitcode versioning
  ○ Honeycomb
    ■ Only supported legacy JIT path
    ■ LLVM 2.7 - 2.9 bitcode depending on MR version
  ○ Ice Cream Sandwich
    ■ MC CodeGen
    ■ LLVM 3.0 bitcode - this one will last a while, right? ;)
  ○ Solution - provide a translator to go between versions
    ■ Partners only need to support latest bitcode format
    ■ llvm-rs-cc generates older bitcode for older target API

● Metadata extraction
  ○ Don't force partners rewrite this for every target backend
    + RS driver implementation
Source Differences

- android/external/clang
  - 32 line diff from Clang upstream!
    - Support for RGBA vector selection (28 lines)
    - Support flag to force "long" to 64-bit (4 lines)
    - Ready to upstream now

- android/external/llvm
  - 368 line diff from LLVM upstream!
  - Mostly patches for legacy JIT path (no longer used - essentially dead code)
  - Stripped down a bit to fit on current tablet/smartphone
  - No debugging support currently (since we don't emit DWARF from llvm-rs-cc today)
Adventures in AST Annotation

- Renderscript runtime manages a bunch of types
  - Allocations in the sample script (plus other things too)
  - How do we know when they can be cleaned up?
    - Java-Side ???
    - Script-Side ???

- Reference Counting
  - rsSetObject(), rsClearObject()
  - Developers do not want to micro-manage opaque blobs
  - Solution is to dynamically annotate script code to use these functions in the appropriate spots
Annotating the AST

- Update this in-place before we emit bitcode
- Need to do a few types of conversions on variables with an RS object type (rs_* types, not including rs_matrix*)
  - Assignments -> rsSetObject(&lhs, rhs)
  - Insert destructor calls as rsClearObject(&local) for locals
- Global variables get cleaned up by runtime after script object is destroyed
rs_font globalIn[10], globalOut;

void foo(int j) {
    rs_font localUninit;
    localUninit = globalIn[0];

    for (int i = 0; i < j; i++) {
        rs_font forNest = globalIn[i];

        switch (i) {
            case 3:
                return;
            case 7:
                continue;
            default:
                break;
        }
        localUninit = forNest;
    }

    globalOut = localUninit;

    return;
}
rs_font globalIn[10], globalOut;
void foo(int j) {
    rs_font localUninit;
    localUninit = globalIn[0];

    for (int i = 0; i < j; i++) {
        rs_font forNest = globalIn[i];

        switch (i) {
            case 3:
                return;
            case 7:
                continue;
            default:
                break;
        }
        localUninit = forNest;
    }
}

globalOut = localUninit;
return;
}
rs_font globalIn[10], globalOut;
void foo(int j) {
    rs_font localUninit;
    rsSetObject(&localUninit, globalIn[0]);  // Simple translation to call-expr

    for (int i = 0; i < j; i++) {
        rs_font forNest;
        rsSetObject(&forNest, globalIn[i]);  // Initializers must be split before conversion

        switch (i) {
            case 3:
                return;
            case 7:
                continue;
            default:
                break;
        }
        rsSetObject(&localUninit, forNest);
    }
    rsSetObject(&globalOut, localUninit);
}

return;
Insert Destructor Calls

rs_font globalIn[10], globalOut;

void foo(int j) {
    rs_font localUninit;
    rsSetObject(&localUninit, globalIn[0]);

    for (int i = 0; i < j; i++) {
        rs_font forNest;
        rsSetObject(&forNest, globalIn[i]);
        switch (i) {
            case 3:
                rsClearObject(&localUninit);  // Return statements always require that you
                rsClearObject(&forNest);      // destroy any in-scope local objects (inside-out).
                return;
            case 7:
                rsClearObject(&forNest);      // continue scopes to for-loop, so destroy forNest
                continue;
            default:
                break;                       // break scopes to switch-stmt, so do nothing
        }
        rsSetObject(&localUninit, forNest);
        rsClearObject(&forNest);       // End of for-loop scope, so destroy forNest
    }
    rsSetObject(&globalOut, localUninit);
    rsClearObject(&localUninit);    // End outer scope (before return)
    return;
}
Conclusions

- **Renderscript**
  - Portable, high-performance, and developer-friendly
  - 3D graphics + compute acceleration path
- **Hide complexity through compiler + runtime**
  - C99-based + forEach
  - Ample use of reflection
  - Library functions
  - Opaque managed types + reference counting
- **Future Work**
  - Debugging and profiling support
  - Improved use of vector intrinsics