Android RenderScript on LLVM

Shih-wei Liao
sliao@google.com

Linux Foundation: Collab-Summit, April 7, 2011

Outline

- Overview of RenderScript (what, why, how, ...)
- Components
  - Offline Compiler
  - Online JIT Compiler
  - RenderScript Runtime
- Put Everything Together:
  - Producing a .apk: HelloCompute Example
  - Running the .apk: Fast launch time & On-device linking
- Conclusions

Google™

What is RenderScript?

- It is ~~the future of~~ Android 3D Rendering and Compute
  - Portability
  - Performance
  - Usability
- C99 plus some extensions
  - Vector operations
  - Function overloading
  - rsForEach
- Current clients include:
  - Books app
  - YouTube app
  - MovieStudio app
  - Live wallpapers + 3D Launcher
- Challenging compiler + runtime work

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 (math functions, etc.)

Google™

Offline Compiler: llvm-rs-cc

- Based on Clang/LLVM
  - Clang and LLVM are popular open source compiler projects
  - Clang - C/C++/ObjC compiler frontend
  - LLVM - Optimizing compiler backend

Key features:

- Cleverly reuse Clang abstract syntax tree (AST) to reflect information back to Java layer
- Embeds metadata within bitcode (type, ...)

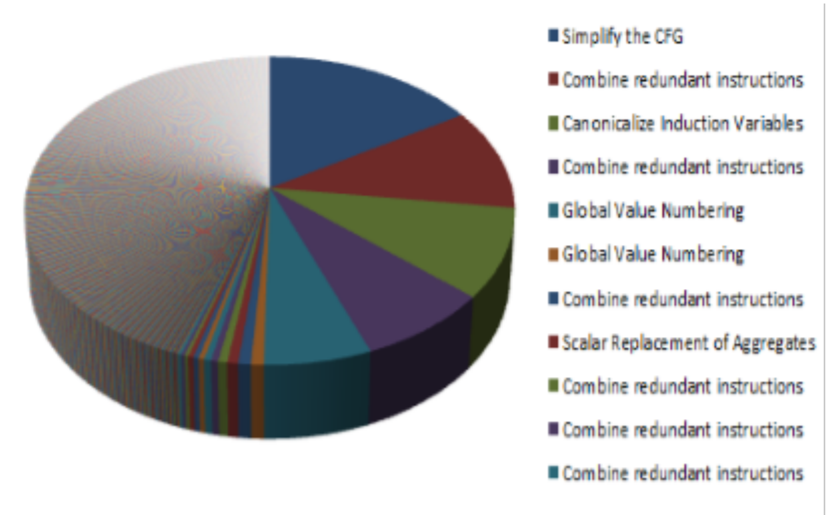
All bitcode supplied as a resource within .apk container

llvm-rs-cc (Cont'd)

Performs aggressive machine-independent optimizations on host before emitting portable bitcode

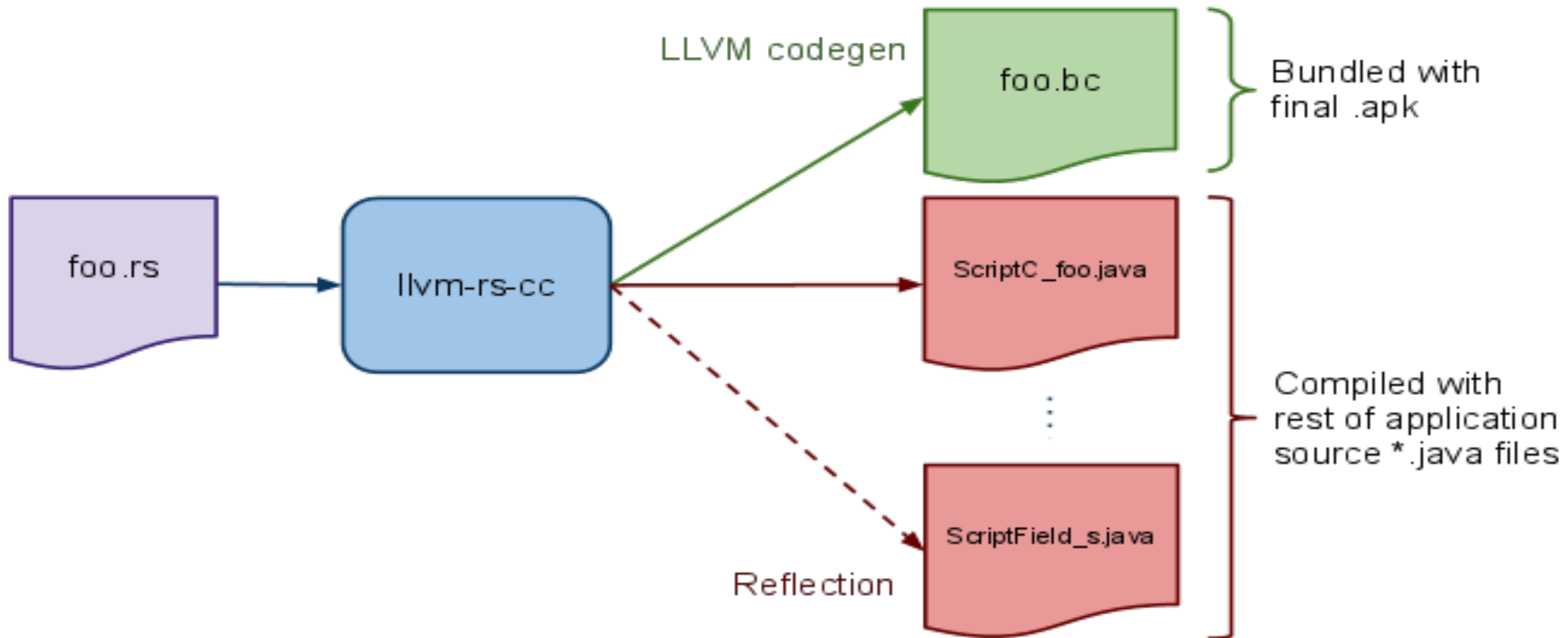
- So the online JIT on Android devices can be lighter-weight

```
==----------------------llvm-rs-cc----------------------------------------===
              ... Pass execution timing report ...
===-----------------------------------------------------------------------===
Total Execution Time: 0.0040 seconds (0.0077 wall clock)
 --System Time--   --System Time--    ---Wall Time---      --- Name ---
 0.0000 (  0.0%)     0.0000 (  0.0%)     0.0090 ( 15.8%)  Simplify the CFG
 0.0000 (  0.0%)     0.0000 (  0.0%)     0.0008 ( 10.5%)  Combine redundant instructions
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0007 (  9.0%)  Canonicalize Induction Variables
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0006 (  7.6%)  Combine redundant instructions
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0005 (  6.0%)  Global Value Numbering
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0004 (  0.7%)  Global Value Numbering
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0004 (  0.7%)  Combine redundant instructions
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0003 (  0.6%)  Scalar Replacement of Aggregates
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0003 (  0.5%)  Combine redundant instructions
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0002 (  0.4%)  Combine redundant instructions
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0002 (  0.4%)  Combine redundant instructions
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0002 (  0.3%)  Combine redundant instructions
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0002 (  0.3%)  Rotate Loops
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.3%)  Natural Loop Information
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.2%)  Interprocedural Sparse Conditional Constant Propagation
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.2%)  Loop Invariant Code Motion
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.2%)  Dominator Tree Construction
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.1%)  Canonicalize natural loops
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.1%)  Jump Threading
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.1%)  Sparse Conditional Constant Propagation
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.1%)  Reassociate expressions
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.1%)  Simplify the CFG
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.1%)  Jump Threading
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.1%)  Dominator Tree Construction
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.1%)  Deduce function attributes
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.1%)  Dominator Tree Construction
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.1%)  Dominance Frontier Construction
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0001 (  0.1%)  Basic CallGraph Construction
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0000 (  0.1%)  Dominance Frontier Construction
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0000 (  0.1%)  Dead Store Elimination
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0000 (  0.1%)  Dominator Tree Construction
 0.0000 (  0.0%)     0.0000 (  0.0%)   0.0000 (  0.1%)  Simplify the CFG
```



Google™

Offline Compiler Flow



Google

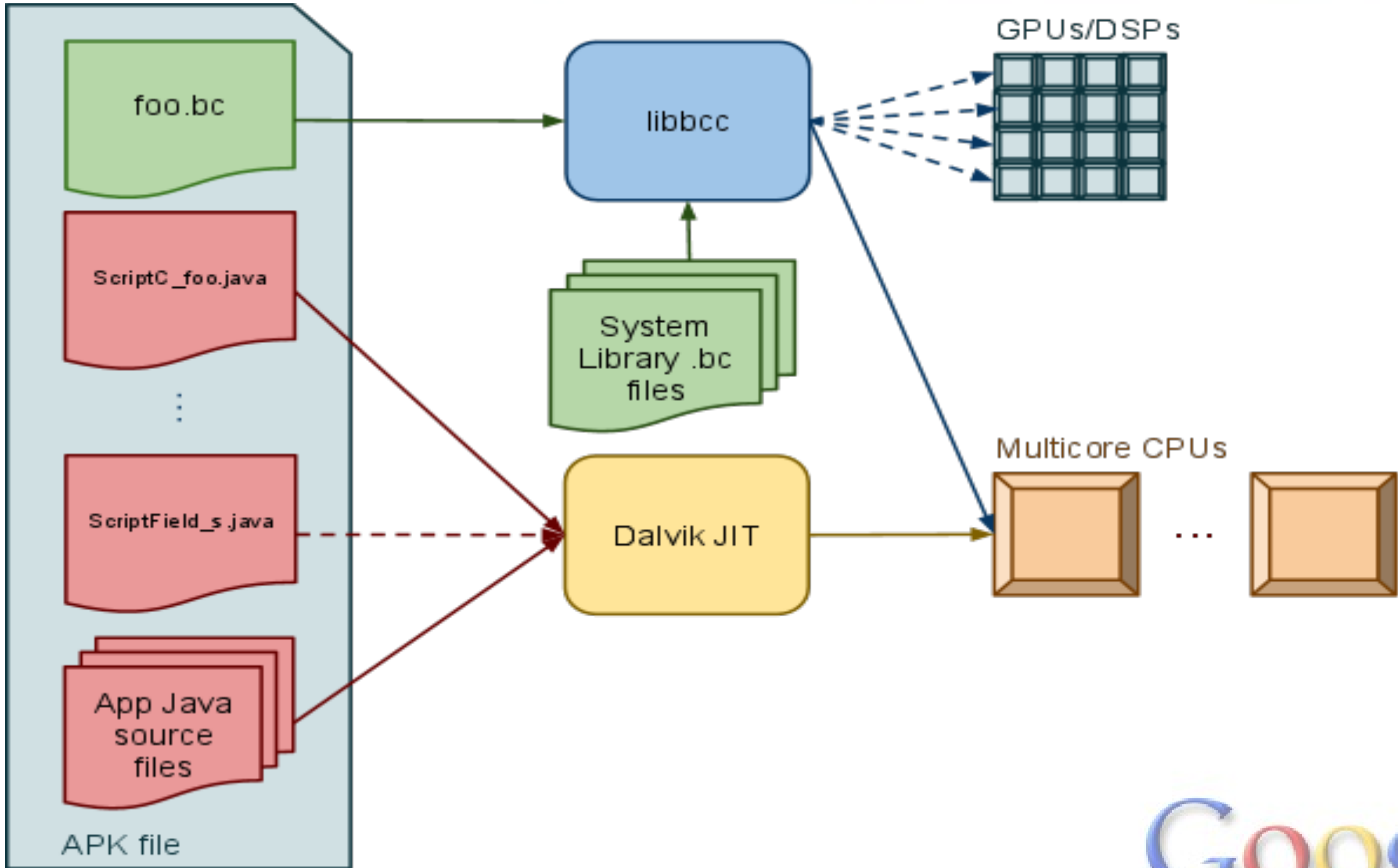
Online JIT Compiler: libbcc

- Based on LLVM: Lighten it to fit in Android devices
  - Currently supports ARM and x86
  - Future targets include GPU/DSP
- Performs target-specific optimizations and code generation

Key features:

- Reflection for libRS: Provides hooks for runtime to access embedded metadata
- Caches JITed scripts to improve startup time
- On-device linking: Links dynamically against runtime library functions (math)

Online JIT Compiler Flow



Google

RenderScript Runtime: libRS

- Manages Scripts and other RenderScript objects
    - Script, Type, Element, Allocation, Mesh, ProgramFragment, ProgramRaster, ProgramStore, ProgramVertex, Sampler, various matrix and primitive types
- Provides implementation of runtime libraries (math, time, drawing, ref-counting, ...)
- Allows allocation, binding of objects to script globals
- Work distribution (multi-threading)
- Message-passing between Dalvik and script-side

HelloCompute Example

- Available in Honeycomb 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

Sample Script (mono.rs)

```
#pragma version(1)
#pragma rs java_package_name(com.example.android.rs.hellocompute)

rs_allocation gIn;
rs_allocation gOut;
rs_script gScript;

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

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

   float3 mono = dot(f4.rgb, gMonoMult);
   *v_out = rsPackColorTo8888(mono);
}

void filter() {
   rsForEach(gScript, gIn, gOut, 0);
}
```

ScriptC_mono.java pt. 1 (generated)

```
package com.example.android.rs.hellocompute;

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

public class ScriptC_mono extends ScriptC {
   public  ScriptC_mono(RenderScript rs, Resources resources, int id) {
      super(rs, resources, id);
   }

   private final static int mExportVarIdx_gIn = 0;
   private Allocation mExportVar_gIn;
   public void set_gIn(Allocation v) {
      mExportVar_gIn = v;
      setVar(mExportVarIdx_gIn, v);
   }

   public Allocation get_gIn() {
      return mExportVar_gIn;
   }
...
```

ScriptC_mono.java pt. 2 (generated)

```
... // Skipping reflection of gOut
   private final static int mExportVarIdx_gScript = 2;
   private Script mExportVar_gScript;
   public void set_gScript(Script v) {
      mExportVar_gScript = v;
      setVar(mExportVarIdx_gScript, v);
   }

   public Script get_gScript() {
      return mExportVar_gScript;
   }

   private final static int mExportFuncIdx_filter = 0;
   public void invoke_filter() {
      invoke(mExportFuncIdx_filter);
   }
}
```

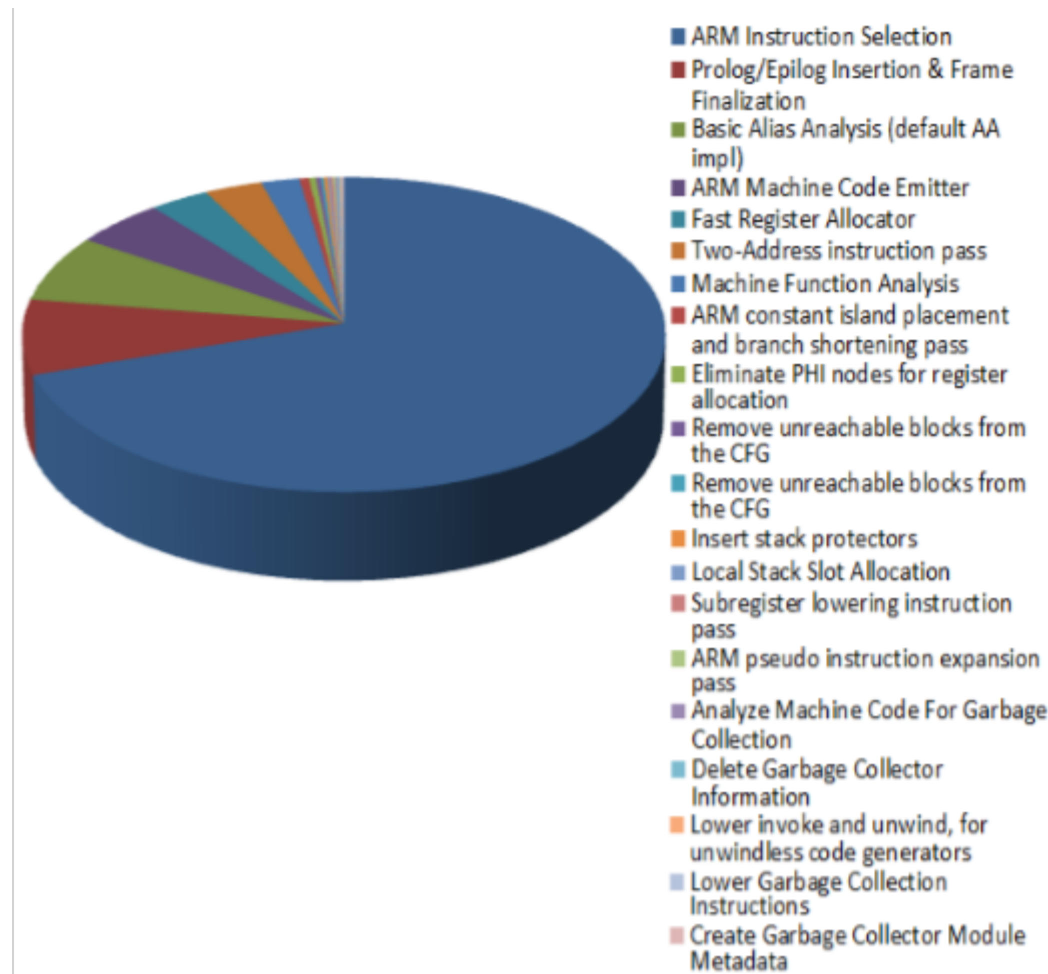
HelloCompute.java (partial source)

```
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.set_gIn(mInAllocation);
        mScript.set_gOut(mOutAllocation);
        mScript.set_gScript(mScript);
        mScript.invoke_filter();
        mOutAllocation.copyTo(mBitmapOut);
    }
    ...
}
```

Running .apk Needs Fast Launch, but...

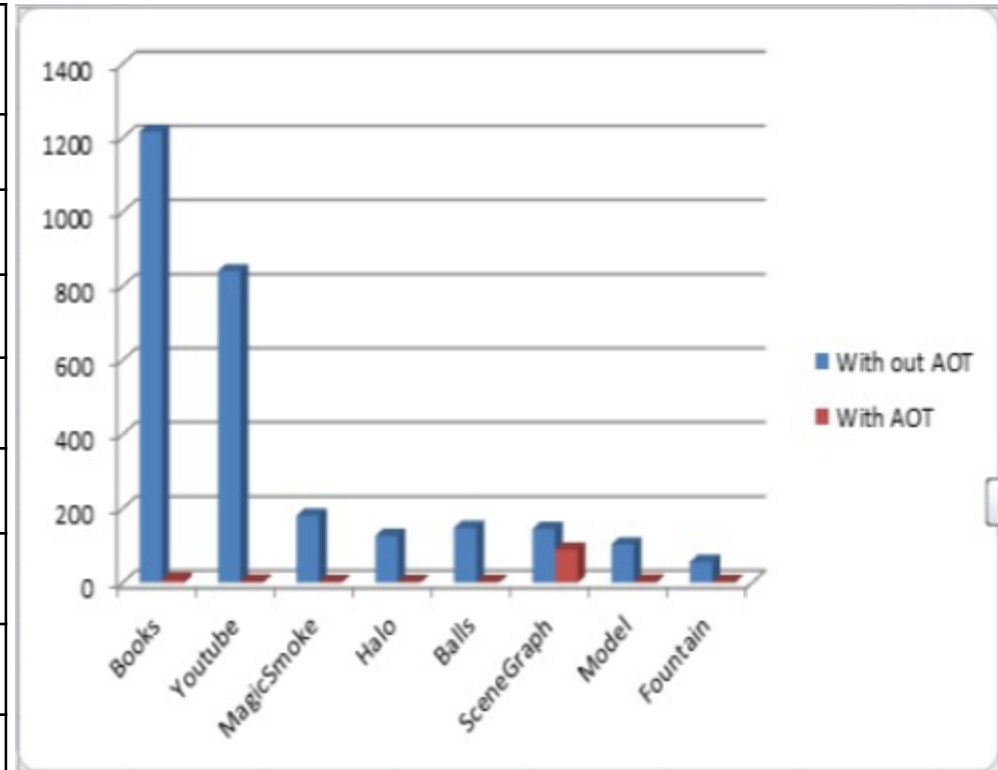
LLVM's optimizations may take time. E.g., for -O3, 750ms for 1500 lines of RenderScript app.

There are still many lines of LLVM code on a smaller app:



Caching: Launch fast, while allowing heavy optimizations ahead-of-time

Apps	*libbcc without AOT. launch time (libbcc)*	*libbcc with AOT. launch time (libbcc)*
Books	1218ms	9ms
Youtube	842ms	4ms
Wallpaper		
MagicSmoke	182ms	3ms
Halo	127ms	3ms
Balls	149ms	3ms
SceneGraph	146ms	90ms
Model	104ms	4ms
Fountain	57ms	3ms



Google

On-Device Linking

Design:
- Provide generic runtime library (lib*.bc), while
  - allow CPU/GPU vendors to provide their specialized lib*.bc, *and*
  - **reduce latency** of function invocation
- An example: RenderScript's runtime (libclcore.bc) comes with **vector** operations. Xoom's libclcore will have different CPU/GPU support (VFP3-16) than Nexus S's (NEON).

Implementation:
- **Linking at LLVM bitcode level**
- Using **Link-Time Optimization** to inline functions and perform constant propagation

Conclusions

- RenderScript
  - Portable, high-performance, and developer-friendly
  - 3D graphics + compute acceleration path
- Hide complexity through compiler + runtime
  - C99-based + rsForEach
  - Ample use of reflection
  - Library functions
  - Opaque managed types + reference counting
- Lightweight JIT:
  - Fast launch time
  - On-device linking
- See http://developer.android.com/ for the latest info

Backup

Google™

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

Google™

Annotating the AST

- AST - Abstract Syntax Tree - follows source code in Clang
- 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

Google™

Reference Counting Example

```
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 Object Local Variables

```
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;
}
```

Assignment -> rsSetObject()

```
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 destroy
    rsClearObject(&forNest);         // 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;
}
```