RenderScript

In this document

Related Samples

- Hello Compute

RenderScript is a framework for running computationally intensive tasks at high performance on Android. RenderScript is primarily oriented for use with data-parallel computation, although serial workloads can benefit as well. The RenderScript runtime parallelizes work across processors available on a device, such as multi-core CPUs and GPUs. This allows you to focus on expressing algorithms rather than scheduling work. RenderScript is especially useful for applications performing image processing, computational photography, or computer vision.

To begin with RenderScript, there are two main concepts you should understand:

- The *language* itself is a C99-derived language for writing high-performance compute code. Writing a RenderScript Kernel describes how to use it to write compute kernels.
- The *control API* is used for managing the lifetime of RenderScript resources and controlling kernel execution. It is available in three different languages: Java, C++ in Android NDK, and the C99-derived kernel language itself. Using RenderScript from Java Code and Single-Source RenderScript describe the first and the third options, respectively.

Writing a RenderScript Kernel

A RenderScript kernel typically resides in a `.rs` file in the `<project_root>/src/` directory; each `.rs` file is called a *script*. Every script contains its own set of kernels, functions, and variables. A script can contain:

- A pragma declaration (`#pragma version(1)`) that declares the version of the RenderScript kernel language used in this script. Currently, 1 is the only valid value.
- A pragma declaration (`#pragma rs java_package_name(com.example.app)`) that declares the package name of the Java classes reflected from this script. Note that your `.rs` file must be part of your application package, and not in a library project.
- Zero or more ***invokable functions***. An invokable function is a single-threaded RenderScript function that you can call from your Java code with arbitrary arguments. These are often useful for initial setup or serial computations within a larger processing pipeline.
- Zero or more ***script globals***. A script global is similar to a global variable in C. You can access script globals from Java code, and these are often used for parameter passing to RenderScript kernels. Script globals are explained in more detail here.
- Zero or more ***compute kernels***. A compute kernel is a function or collection of functions that you can direct the RenderScript runtime to

execute in parallel across a collection of data. There are two kinds of compute kernels: *mapping* kernels (also called *foreach* kernels) and *reduction* kernels.

A *mapping kernel* is a parallel function that operates on a collection of `Allocations` of the same dimensions. By default, it executes once for every coordinate in those dimensions. It is typically (but not exclusively) used to transform a collection of input `Allocations` to an output `Allocation` one `Element` at a time.

- Here is an example of a simple **mapping kernel**:

```
uchar4 RS_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;
}
```

  In most respects, this is identical to a standard C function. The `RS_KERNEL` property applied to the function prototype specifies that the function is a RenderScript mapping kernel instead of an invokable function. The `in` argument is automatically filled in based on the input `Allocation` passed to the kernel launch. The arguments `x` and `y` are discussed below. The value returned from the kernel is automatically written to the appropriate location in the output `Allocation`. By default, this kernel is run across its entire input `Allocation`, with one execution of the kernel function per `Element` in the `Allocation`.

  A mapping kernel may have one or more input `Allocations`, a single output `Allocation`, or both. The RenderScript runtime checks to ensure that all input and output Allocations have the same dimensions, and that the `Element` types of the input and output Allocations match the kernel's prototype; if either of these checks fails, RenderScript throws an exception.

> **NOTE:** Before Android 6.0 (API level 23), a mapping kernel may not have more than one input `Allocation`.

  If you need more input or output `Allocations` than the kernel has, those objects should be bound to `rs_allocation` script globals and accessed from a kernel or invokable function via `rsGetElementAt_type()` or `rsSetElementAt_type()`.

  NOTE: `RS_KERNEL` is a macro defined automatically by RenderScript for your convenience:

```
#define RS_KERNEL __attribute__((kernel))
```

A *reduction kernel* is a family of functions that operates on a collection of input `Allocations` of the same dimensions. By default, its accumulator function executes once for every coordinate in those dimensions. It is typically (but not exclusively) used to "reduce" a collection of input `Allocations` to a single value.

- Here is an example of a simple **reduction kernel** that adds up the `Elements` of its input:

```
#pragma rs reduce(addint) accumulator(addintAccum)

static void addintAccum(int *accum, int val) {
  *accum += val;
}
```

  A reduction kernel consists of one or more user-written functions. `#pragma rs reduce` is used to define the kernel by specifying its name (`addint`, in this example) and the names and roles of the functions that make up the kernel (an `accumulator` function `addintAccum`, in this example). All such functions must be `static`. A reduction kernel always requires an `accumulator` function; it may also have other functions, depending on what you want the kernel to do.

  A reduction kernel accumulator function must return `void` and must have at least two arguments. The first argument (`accum`, in this example) is a pointer to an *accumulator data item* and the second (`val`, in this example) is automatically filled in based on the input `Allocation` passed to the kernel launch. The accumulator data item is created by the RenderScript runtime; by default, it is initialized to zero. By default, this kernel is run across its entire input `Allocation`, with one execution of the accumulator function per `Element` in the `Allocation`. By default, the final value of the accumulator data item is treated as the result of the reduction, and is returned to Java. The RenderScript runtime checks to ensure that the `Element` type of the input Allocation matches the accumulator function's prototype; if it does not match, RenderScript throws an exception.

  A reduction kernel has one or more input `Allocations` but no output `Allocations`.

  Reduction kernels are explained in more detail here.

Reduction kernels are supported in Android 7.0 (API level 24) and later.

A mapping kernel function or a reduction kernel accumulator function may access the coordinates of the current execution using the special arguments `x`, `y`, and `z`, which must be of type `int` or `uint32_t`. These arguments are optional.

A mapping kernel function or a reduction kernel accumulator function may also take the optional special argument `context` of type rs_kernel_context. It is needed by a family of runtime APIs that are used to query certain properties of the current execution -- for example, rsGetDimX. (The `context` argument is available in Android 6.0 (API level 23) and later.)

- An optional `init()` function. The `init()` function is a special type of invokable function that RenderScript runs when the script is first instantiated. This allows for some computation to occur automatically at script creation.
- Zero or more ***static script globals and functions***. A static script global is equivalent to a script global except that it cannot be accessed from Java code. A static function is a standard C function that can be called from any kernel or invokable function in the script but is not exposed to the Java API. If a script global or function does not need to be accessed from Java code, it is highly recommended that it be declared `static`.

Setting floating point precision

You can control the required level of floating point precision in a script. This is useful if full IEEE 754-2008 standard (used by default) is not required. The following pragmas can set a different level of floating point precision:

- `#pragma rs_fp_full` (default if nothing is specified): For apps that require floating point precision as outlined by the IEEE 754-2008 standard.
- `#pragma rs_fp_relaxed`: For apps that don't require strict IEEE 754-2008 compliance and can tolerate less precision. This mode enables flush-to-zero for denorms and round-towards-zero.
- `#pragma rs_fp_imprecise`: For apps that don't have stringent precision requirements. This mode enables everything in `rs_fp_relaxed` along with the following:
  - Operations resulting in -0.0 can return +0.0 instead.
  - Operations on INF and NAN are undefined.

Most applications can use `rs_fp_relaxed` without any side effects. This may be very beneficial on some architectures due to additional optimizations only available with relaxed precision (such as SIMD CPU instructions).

Accessing RenderScript APIs from Java

When developing an Android application that uses RenderScript, you can access its API from Java in one of two ways:

- `android.renderscript` - The APIs in this class package are available on devices running Android 3.0 (API level 11) and higher.
- `android.support.v8.renderscript` - The APIs in this package are available through a Support Library, which allows you to use them on devices running Android 2.3 (API level 9) and higher.

Here are the tradeoffs:

- If you use the Support Library APIs, the RenderScript portion of your application will be compatible with devices running Android 2.3 (API level 9) and higher, regardless of which RenderScript features you use. This allows your application to work on more devices than if you use the native (`android.renderscript`) APIs.
- Certain RenderScript features are not available through the Support Library APIs.
- If you use the Support Library APIs, you will get (possibly significantly) larger APKs than if you use the native (`android.renderscript`) APIs.

Using the RenderScript Support Library APIs

In order to use the Support Library RenderScript APIs, you must configure your development environment to be able to access them. The following Android SDK tools are required for using these APIs:

- Android SDK Tools revision 22.2 or higher

- Android SDK Build-tools revision 18.1.0 or higher

Note that starting from Android SDK Build-tools 24.0.0, Android 2.2 (API level 8) is no longer supported.

You can check and update the installed version of these tools in the Android SDK Manager.

To use the Support Library RenderScript APIs:

1. Make sure you have the required Android SDK version and Build Tools version installed.
2. Update the settings for the Android build process to include the RenderScript settings:
   - Open the `build.gradle` file in the app folder of your application module.
   - Add the following RenderScript settings to the file:

     ```
     android {
         compileSdkVersion 23
         buildToolsVersion "23.0.3"

         defaultConfig {
             minSdkVersion 9
             targetSdkVersion 19

             renderscriptTargetApi 18
             renderscriptSupportModeEnabled true

         }
     }
     ```

     The settings listed above control specific behavior in the Android build process:

     - `renderscriptTargetApi` - Specifies the bytecode version to be generated. We recommend you set this value to the lowest API level able to provide all the functionality you are using and set `renderscriptSupportModeEnabled` to `true`. Valid values for this setting are any integer value from 11 to the most recently released API level. If your minimum SDK version specified in your application manifest is set to a different value, that value is ignored and the target value in the build file is used to set the minimum SDK version.
     - `renderscriptSupportModeEnabled` - Specifies that the generated bytecode should fall back to a compatible version if the device it is running on does not support the target version.
     - `buildToolsVersion` - The version of the Android SDK build tools to use. This value should be set to `18.1.0` or higher. If this option is not specified, the highest installed build tools version is used. You should always set this value to ensure the consistency of builds across development machines with different configurations.
3. In your application classes that use RenderScript, add an import for the Support Library classes:

   ```
   import android.support.v8.renderscript.*;
   ```

Using RenderScript from Java Code

Using RenderScript from Java code relies on the API classes located in the `android.renderscript` or the `android.support.v8.renderscript` package. Most applications follow the same basic usage pattern:

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 at all 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 RenderScript code and vice-versa. `Allocation` objects are typically created using `createTyped()` or `createFromBitmap()`.

3. **Create whatever scripts are necessary.** There are two types of scripts available to you when using RenderScript:
   - **ScriptC**: These are the user-defined scripts as described in *Writing a RenderScript Kernel* above. Every script has a Java class reflected by the RenderScript compiler in order to make it easy to access the script from Java code; this class has the name `ScriptC_filename`. For example, if the mapping kernel above were located in `invert.rs` and a RenderScript context were already located in `mRenderScript`, 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 `createFromBitmap()`, an Allocation is populated with empty data when it is first created. To populate an Allocation, use one of the "copy" methods in `Allocation`. The "copy" methods are synchronous.
5. **Set any necessary script globals.** You may set globals using methods in the same `ScriptC_filename` class named `set_globalname`. For example, in order to set an `int` variable named `threshold`, use the Java method `set_threshold(int)`; and in order to set an `rs_allocation` variable named `lookup`, use the Java method `set_lookup(Allocation)`. The `set` methods are asynchronous.
6. **Launch the appropriate kernels and invokable functions.**

   Methods to launch a given kernel are reflected in the same `ScriptC_filename` class with methods named `forEach_mappingKernelName()` or `reduce_reductionKernelName()`. These launches are asynchronous. Depending on the arguments to the kernel, the method takes one or more Allocations, all of which must have the same dimensions. By default, a kernel executes over every coordinate in those dimensions; to execute a kernel over a subset of those coordinates, pass an appropriate `Script.LaunchOptions` as the last argument to the `forEach` or `reduce` method.

   Launch invokable functions using the `invoke_functionName` methods reflected in the same `ScriptC_filename` class. These launches are asynchronous.
7. **Retrieve data from `Allocation` objects and *javaFutureType* objects.** In order to access data from an `Allocation` from Java code, you must copy that data back to Java using one of the "copy" methods in `Allocation`. In order to obtain the result of a reduction kernel, you must use the `javaFutureType.get()` method. The "copy" and `get()` methods are synchronous.
8. **Tear down the RenderScript context.** You can destroy the RenderScript context with `destroy()` or by allowing the RenderScript context object to be garbage collected. This causes any further use of any object belonging to that context to throw an exception.

Asynchronous execution model

The reflected `forEach`, `invoke`, `reduce`, and `set` methods are asynchronous -- each may return to Java before completing the requested action. However, the individual actions are serialized in the order in which they are launched.

The `Allocation` class provides "copy" methods to copy data to and from Allocations. A "copy" method is synchronous, and is serialized with respect to any of the asynchronous actions above that touch the same Allocation.

The reflected *javaFutureType* classes provide a `get()` method to obtain the result of a reduction. `get()` is synchronous, and is serialized with respect to the reduction (which is asynchronous).

Single-Source RenderScript

Android 7.0 (API level 24) introduces a new programming feature called *Single-Source RenderScript*, in which kernels are launched from the script where they are defined, rather than from Java. This approach is currently limited to mapping kernels, which are simply referred to as "kernels" in this section for conciseness. This new feature also supports creating allocations of type `rs_allocation` from inside the script. It is now possible to implement a whole algorithm solely within a script, even if multiple kernel launches are required. The benefit is twofold: more readable code, because it keeps the implementation of an algorithm in one language; and potentially faster code, because of fewer transitions between Java and RenderScript across multiple kernel launches.

In Single-Source RenderScript, you write kernels as described in Writing a RenderScript Kernel. You then write an invokable function that calls `rsForEach()` to launch them. That API takes a kernel function as the first parameter, followed by input and output allocations. A similar API `rsForEachWithOptions()` takes an extra argument of type `rs_script_call_t`, which specifies a subset of the elements from the input

and output allocations for the kernel function to process.

To start RenderScript computation, you call the invokable function from Java. Follow the steps in Using RenderScript from Java Code. In the step launch the appropriate kernels, call the invokable function using `invoke_function_name()`, which will start the whole computation, including launching kernels.

Allocations are often needed to save and pass intermediate results from one kernel launch to another. You can create them using rsCreateAllocation(). One easy-to-use form of that API is `rsCreateAllocation_<T><W>(…)`, where *T* is the data type for an element, and *W* is the vector width for the element. The API takes the sizes in dimensions X, Y, and Z as arguments. For 1D or 2D allocations, the size for dimension Y or Z can be omitted. For example, `rsCreateAllocation_uchar4(16384)` creates a 1D allocation of 16384 elements, each of which is of type `uchar4`.

Allocations are managed by the system automatically. You do not have to explicitly release or free them. However, you can call `rsClearObject(rs_allocation* alloc)` to indicate you no longer need the handle `alloc` to the underlying allocation, so that the system can free up resources as early as possible.

The Writing a RenderScript Kernel section contains an example kernel that inverts an image. The example below expands that to apply more than one effect to an image, using Single-Source RenderScript. It includes another kernel, `greyscale`, which turns a color image into black-and-white. An invokable function `process()` then applies those two kernels consecutively to an input image, and produces an output image. Allocations for both the input and the output are passed in as arguments of type `rs_allocation`.

```
// File: singlesource.rs

#pragma version(1)
#pragma rs java_package_name(com.android.rssample)

static const float4 weight = {0.299f, 0.587f, 0.114f, 0.0f};

uchar4 RS_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;
}

uchar4 RS_KERNEL greyscale(uchar4 in) {
  const float4 inF = rsUnpackColor8888(in);
  const float4 outF = (float4){ dot(inF, weight) };
  return rsPackColorTo8888(outF);
}

void process(rs_allocation inputImage, rs_allocation outputImage) {
  const uint32_t imageWidth = rsAllocationGetDimX(inputImage);
  const uint32_t imageHeight = rsAllocationGetDimY(inputImage);
  rs_allocation tmp = rsCreateAllocation_uchar4(imageWidth, imageHeight);
  rsForEach(invert, inputImage, tmp);
  rsForEach(greyscale, tmp, outputImage);
}
```

You can call the `process()` function from Java as follows:

```
// File SingleSource.java

RenderScript RS = RenderScript.create(context);
ScriptC_singlesource script = new ScriptC_singlesource(RS);
Allocation inputAllocation = Allocation.createFromBitmapResource(
    RS, getResources(), R.drawable.image);
Allocation outputAllocation = Allocation.createTyped(
    RS, inputAllocation.getType(),
    Allocation.USAGE_SCRIPT | Allocation.USAGE_IO_OUTPUT);
script.invoke_process(inputAllocation, outputAllocation);
```

This example shows how an algorithm that involves two kernel launches can be implemented completely in the RenderScript language itself. Without Single-Source RenderScript, you would have to launch both kernels from the Java code, separating kernel launches from kernel

definitions and making it harder to understand the whole algorithm. Not only is the Single-Source RenderScript code easier to read, it also eliminates the transitioning between Java and the script across kernel launches. Some iterative algorithms may launch kernels hundreds of times, making the overhead of such transitioning considerable.

Script Globals

A *script global* is an ordinary non-`static` global variable in a script (`.rs`) file. For a script global named *var* defined in the file *`filename`*`.rs`, there will be a method `get_`*`var`* reflected in the class `ScriptC_`*`filename`*. Unless the global is `const`, there will also be a method `set_`*`var`*.

A given script global has two separate values -- a *Java* value and a *script* value. These values behave as follows:

- If *var* has a static initializer in the script, it specifies the initial value of *var* in both Java and the script. Otherwise, that initial value is zero.
- Accesses to *var* within the script read and write its script value.
- The `get_`*`var`* method reads the Java value.
- The `set_`*`var`* method (if it exists) writes the Java value immediately, and writes the script value asynchronously.

> **NOTE:** This means that except for any static initializer in the script, values written to a global from within a script are not visible to Java.

Reduction Kernels in Depth

Reduction is the process of combining a collection of data into a single value. This is a useful primitive in parallel programming, with applications such as the following:

- computing the sum or product over all the data
- computing logical operations (`and`, `or`, `xor`) over all the data
- finding the minimum or maximum value within the data
- searching for a specific value or for the coordinate of a specific value within the data

In Android 7.0 (API level 24) and later, RenderScript supports *reduction kernels* to allow efficient user-written reduction algorithms. You may launch reduction kernels on inputs with 1, 2, or 3 dimensions.

An example above shows a simple addint reduction kernel. Here is a more complicated findMinAndMax reduction kernel that finds the locations of the minimum and maximum `long` values in a 1-dimensional `Allocation`:

```
#define LONG_MAX (long)((1UL << 63) - 1)
#define LONG_MIN (long)(1UL << 63)

#pragma rs reduce(findMinAndMax) \
  initializer(fMMInit) accumulator(fMMAccumulator) \
  combiner(fMMCombiner) outconverter(fMMOutConverter)

// Either a value and the location where it was found, or INITVAL.
typedef struct {
  long val;
  int idx;     // -1 indicates INITVAL
} IndexedVal;

typedef struct {
  IndexedVal min, max;
} MinAndMax;

// In discussion below, this initial value { { LONG_MAX, -1 }, { LONG_MIN, -1 } }
// is called INITVAL.
static void fMMInit(MinAndMax *accum) {
  accum->min.val = LONG_MAX;
  accum->min.idx = -1;
  accum->max.val = LONG_MIN;
  accum->max.idx = -1;
}

//
```

```
//-----------------------------------------------------------------------
// In describing the behavior of the accumulator and combiner functions,
// it is helpful to describe hypothetical functions
//   IndexedVal min(IndexedVal a, IndexedVal b)
//   IndexedVal max(IndexedVal a, IndexedVal b)
//   MinAndMax  minmax(MinAndMax a, MinAndMax b)
//   MinAndMax  minmax(MinAndMax accum, IndexedVal val)
//
// The effect of
//   IndexedVal min(IndexedVal a, IndexedVal b)
// is to return the IndexedVal from among the two arguments
// whose val is lesser, except that when an IndexedVal
// has a negative index, that IndexedVal is never less than
// any other IndexedVal; therefore, if exactly one of the
// two arguments has a negative index, the min is the other
// argument. Like ordinary arithmetic min and max, this function
// is commutative and associative; that is,
//
//   min(A, B) == min(B, A)               // commutative
//   min(A, min(B, C)) == min((A, B), C)  // associative
//
// The effect of
//   IndexedVal max(IndexedVal a, IndexedVal b)
// is analogous (greater . . . never greater than).
//
// Then there is
//
//   MinAndMax minmax(MinAndMax a, MinAndMax b) {
//     return MinAndMax(min(a.min, b.min), max(a.max, b.max));
//   }
//
// Like ordinary arithmetic min and max, the above function
// is commutative and associative; that is:
//
//   minmax(A, B) == minmax(B, A)                 // commutative
//   minmax(A, minmax(B, C)) == minmax((A, B), C)  // associative
//
// Finally define
//
//   MinAndMax minmax(MinAndMax accum, IndexedVal val) {
//     return minmax(accum, MinAndMax(val, val));
//   }
//-----------------------------------------------------------------------

// This function can be explained as doing:
//   *accum = minmax(*accum, IndexedVal(in, x))
//
// This function simply computes minimum and maximum values as if
// INITVAL.min were greater than any other minimum value and
// INITVAL.max were less than any other maximum value.  Note that if
// *accum is INITVAL, then this function sets
//   *accum = IndexedVal(in, x)
//
// After this function is called, both accum->min.idx and accum->max.idx
// will have nonnegative values:
// - x is always nonnegative, so if this function ever sets one of the
//   idx fields, it will set it to a nonnegative value
// - if one of the idx fields is negative, then the corresponding
//   val field must be LONG_MAX or LONG_MIN, so the function will always
//   set both the val and idx fields
static void fMMAccumulator(MinAndMax *accum, long in, int x) {
  IndexedVal me;
  me.val = in;
  me.idx = x;

  if (me.val <= accum->min.val)
    accum->min = me;
  if (me.val >= accum->max.val)
    accum->max = me;
}

// This function can be explained as doing:
//   *accum = minmax(*accum, *val)
```

```
//
// This function simply computes minimum and maximum values as if
// INITVAL.min were greater than any other minimum value and
// INITVAL.max were less than any other maximum value.  Note that if
// one of the two accumulator data items is INITVAL, then this
// function sets *accum to the other one.
static void fMMCombiner(MinAndMax *accum,
                        const MinAndMax *val) {
  if ((accum->min.idx < 0) || (val->min.val < accum->min.val))
    accum->min = val->min;
  if ((accum->max.idx < 0) || (val->max.val > accum->max.val))
    accum->max = val->max;
}

static void fMMOutConverter(int2 *result,
                            const MinAndMax *val) {
  result->x = val->min.idx;
  result->y = val->max.idx;
}
```

> **NOTE:** There are more example reduction kernels here.

In order to run a reduction kernel, the RenderScript runtime creates *one or more* variables called ***accumulator data items*** to hold the state of the reduction process. The RenderScript runtime picks the number of accumulator data items in such a way as to maximize performance. The type of the accumulator data items (*accumType*) is determined by the kernel's *accumulator function* -- the first argument to that function is a pointer to an accumulator data item. By default, every accumulator data item is initialized to zero (as if by `memset`); however, you may write an *initializer function* to do something different.

> **Example:** In the addint kernel, the accumulator data items (of type `int`) are used to add up input values. There is no initializer function, so each accumulator data item is initialized to zero.

> **Example:** In the findMinAndMax kernel, the accumulator data items (of type `MinAndMax`) are used to keep track of the minimum and maximum values found so far. There is an initializer function to set these to `LONG_MAX` and `LONG_MIN`, respectively; and to set the locations of these values to -1, indicating that the values are not actually present in the (empty) portion of the input that has been processed.

RenderScript calls your accumulator function once for every coordinate in the input(s). Typically, your function should update the accumulator data item in some way according to the input.

> **Example:** In the addint kernel, the accumulator function adds the value of an input Element to the accumulator data item.

> **Example:** In the findMinAndMax kernel, the accumulator function checks to see whether the value of an input Element is less than or equal to the minimum value recorded in the accumulator data item and/or greater than or equal to the maximum value recorded in the accumulator data item, and updates the accumulator data item accordingly.

After the accumulator function has been called once for every coordinate in the input(s), RenderScript must **combine** the accumulator data items together into a single accumulator data item. You may write a *combiner function* to do this. If the accumulator function has a single input and no special arguments, then you do not need to write a combiner function; RenderScript will use the accumulator function to combine the accumulator data items. (You may still write a combiner function if this default behavior is not what you want.)

> **Example:** In the addint kernel, there is no combiner function, so the accumulator function will be used. This is the correct behavior, because if we split a collection of values into two pieces, and we add up the values in those two pieces separately, adding up those two sums is the same as adding up the entire collection.

> **Example:** In the findMinAndMax kernel, the combiner function checks to see whether the minimum value recorded in the "source" accumulator data item `*val` is less then the minimum value recorded in the "destination" accumulator data item `*accum`, and updates `*accum` accordingly. It does similar work for the maximum value. This updates `*accum` to the state it would have had if all of the input values had been accumulated into `*accum` rather than some into `*accum` and some into `*val`.

After all of the accumulator data items have been combined, RenderScript determines the result of the reduction to return to Java. You may write an *outconverter function* to do this. You do not need to write an outconverter function if you want the final value of the combined accumulator data items to be the result of the reduction.

> **Example:** In the addint kernel, there is no outconverter function. The final value of the combined data items is the sum of all Elements of the input, which is the value we want to return.

> **Example:** In the findMinAndMax kernel, the outconverter function initializes an `int2` result value to hold the locations of the minimum and maximum values resulting from the combination of all of the accumulator data items.

Writing a reduction kernel

`#pragma rs reduce` defines a reduction kernel by specifying its name and the names and roles of the functions that make up the kernel. All such functions must be `static`. A reduction kernel always requires an `accumulator` function; you can omit some or all of the other functions, depending on what you want the kernel to do.

```
#pragma rs reduce(kernelName) \
  initializer(initializerName) \
  accumulator(accumulatorName) \
  combiner(combinerName) \
  outconverter(outconverterName)
```

The meaning of the items in the `#pragma` is as follows:

- `reduce(kernelName)` (mandatory): Specifies that a reduction kernel is being defined. A reflected Java method `reduce_kernelName` will launch the kernel.
- `initializer(initializerName)` (optional): Specifies the name of the initializer function for this reduction kernel. When you launch the kernel, RenderScript calls this function once for each accumulator data item. The function must be defined like this:

  ```
  static void initializerName(accumType *accum) { … }
  ```

  `accum` is a pointer to an accumulator data item for this function to initialize.

  If you do not provide an initializer function, RenderScript initializes every accumulator data item to zero (as if by `memset`), behaving as if there were an initializer function that looks like this:

  ```
  static void initializerName(accumType *accum) {
    memset(accum, 0, sizeof(*accum));
  }
  ```

- `accumulator(accumulatorName)` (mandatory): Specifies the name of the accumulator function for this reduction kernel. When you launch the kernel, RenderScript calls this function once for every coordinate in the input(s), to update an accumulator data item in some way according to the input(s). The function must be defined like this:

  ```
  static void accumulatorName(accumType *accum,
                              in1Type in1, …, inNType inN
                              [, specialArguments]) { … }
  ```

  `accum` is a pointer to an accumulator data item for this function to modify. `in1` through `inN` are one *or more* arguments that are automatically filled in based on the inputs passed to the kernel launch, one argument per input. The accumulator function may optionally take any of the special arguments.

  An example kernel with multiple inputs is `dotProduct`.

- `combiner(combinerName)` (optional): Specifies the name of the combiner function for this reduction kernel. After RenderScript calls the accumulator function once for every coordinate in the input(s), it calls this function as many times as necessary to combine all accumulator data items into a single accumulator data item. The function must be defined like this:

  ```
  static void combinerName(accumType *accum, const accumType *other) { … }
  ```

  `accum` is a pointer to a "destination" accumulator data item for this function to modify. `other` is a pointer to a "source" accumulator data item for this function to "combine" into `*accum`.

  > **NOTE:** It is possible that `*accum`, `*other`, or both have been initialized but have never been passed to the accumulator function; that

> is, one or both have never been updated according to any input data. For example, in the findMinAndMax kernel, the combiner function `fMMCombiner` explicitly checks for `idx < 0` because that indicates such an accumulator data item, whose value is INITVAL.

If you do not provide a combiner function, RenderScript uses the accumulator function in its place, behaving as if there were a combiner function that looks like this:

```
static void combinerName(accumType *accum, const accumType *other) {
  accumulatorName(accum, *other);
}
```

A combiner function is mandatory if the kernel has more than one input, if the input data type is not the same as the accumulator data type, or if the accumulator function takes one or more special arguments.

- `outconverter(outconverterName)` (optional): Specifies the name of the outconverter function for this reduction kernel. After RenderScript combines all of the accumulator data items, it calls this function to determine the result of the reduction to return to Java. The function must be defined like this:

  ```
  static void outconverterName(resultType *result, const accumType *accum) { … }
  ```

  `result` is a pointer to a result data item (allocated but not initialized by the RenderScript runtime) for this function to initialize with the result of the reduction. *resultType* is the type of that data item, which need not be the same as *accumType*. `accum` is a pointer to the final accumulator data item computed by the combiner function.

  If you do not provide an outconverter function, RenderScript copies the final accumulator data item to the result data item, behaving as if there were an outconverter function that looks like this:

  ```
  static void outconverterName(accumType *result, const accumType *accum) {
    *result = *accum;
  }
  ```

  If you want a different result type than the accumulator data type, then the outconverter function is mandatory.

Note that a kernel has input types, an accumulator data item type, and a result type, none of which need to be the same. For example, in the findMinAndMax kernel, the input type `long`, accumulator data item type `MinAndMax`, and result type `int2` are all different.

What can't you assume?

You must not rely on the number of accumulator data items created by RenderScript for a given kernel launch. There is no guarantee that two launches of the same kernel with the same input(s) will create the same number of accumulator data items.

You must not rely on the order in which RenderScript calls the initializer, accumulator, and combiner functions; it may even call some of them in parallel. There is no guarantee that two launches of the same kernel with the same input will follow the same order. The only guarantee is that only the initializer function will ever see an uninitialized accumulator data item. For example:

- There is no guarantee that all accumulator data items will be initialized before the accumulator function is called, although it will only be called on an initialized accumulator data item.
- There is no guarantee on the order in which input Elements are passed to the accumulator function.
- There is no guarantee that the accumulator function has been called for all input Elements before the combiner function is called.

One consequence of this is that the findMinAndMax kernel is not deterministic: If the input contains more than one occurrence of the same minimum or maximum value, you have no way of knowing which occurrence the kernel will find.

What must you guarantee?

Because the RenderScript system can choose to execute a kernel in many different ways, you must follow certain rules to ensure that your kernel behaves the way you want. If you do not follow these rules, you may get incorrect results, nondeterministic behavior, or runtime errors.

The rules below often say that two accumulator data items must have "the same value". What does this mean? That depends on what you want the kernel to do. For a mathematical reduction such as addint, it usually makes sense for "the same" to mean mathematical equality. For a "pick any" search such as findMinAndMax ("find the location of minimum and maximum input values") where there might be more than one occurrence of identical input values, all locations of a given input value must be considered "the same". You could write a similar kernel to "find the location of *leftmost* minimum and maximum input values" where (say) a minimum value at location 100 is preferred over an identical

minimum value at location 200; for this kernel, "the same" would mean identical *location*, not merely identical *value*, and the accumulator and combiner functions would have to be different than those for findMinAndMax.

The initializer function must create an *identity value*. That is, if `I` and `A` are accumulator data items initialized by the initializer function, and `I` has never been passed to the accumulator function (but `A` may have been), then

- `combinerName(&A, &I)` must leave `A` the same
- `combinerName(&I, &A)` must leave `I` the same as `A`

> **Example:** In the addint kernel, an accumulator data item is initialized to zero. The combiner function for this kernel performs addition; zero is the identity value for addition.

> **Example:** In the findMinAndMax kernel, an accumulator data item is initialized to `INITVAL`.
>
> - `fMMCombiner(&A, &I)` leaves `A` the same, because `I` is `INITVAL`.
> - `fMMCombiner(&I, &A)` sets `I` to `A`, because `I` is `INITVAL`.
>
> Therefore, `INITVAL` is indeed an identity value.

The combiner function must be *commutative*. That is, if `A` and `B` are accumulator data items initialized by the initializer function, and that may have been passed to the accumulator function zero or more times, then `combinerName(&A, &B)` must set `A` to the same value that `combinerName(&B, &A)` sets `B`.

> **Example:** In the addint kernel, the combiner function adds the two accumulator data item values; addition is commutative.

> **Example:** In the findMinAndMax kernel,
>
> ```
> fMMCombiner(&A, &B)
> ```
>
> is the same as
>
> ```
> A = minmax(A, B)
> ```
>
> and `minmax` is commutative, so `fMMCombiner` is also.

The combiner function must be *associative*. That is, if `A`, `B`, and `C` are accumulator data items initialized by the initializer function, and that may have been passed to the accumulator function zero or more times, then the following two code sequences must set `A` to the same value:

```
combinerName(&A, &B);
combinerName(&A, &C);
```

```
combinerName(&B, &C);
combinerName(&A, &B);
```

> **Example:** In the addint kernel, the combiner function adds the two accumulator data item values:
>
> ```
> A = A + B
> A = A + C
> // Same as
> //   A = (A + B) + C
> ```
>
> ```
> B = B + C
> A = A + B
> // Same as
> //   A = A + (B + C)
> //   B = B + C
> ```
>
> Addition is associative, and so the combiner function is also.

Example: In the findMinAndMax kernel,

```
fMMCombiner(&A, &B)
```

is the same as

```
A = minmax(A, B)
```

So the two sequences are

```
A = minmax(A, B)
A = minmax(A, C)
// Same as
//   A = minmax(minmax(A, B), C)
```

```
B = minmax(B, C)
A = minmax(A, B)
// Same as
//   A = minmax(A, minmax(B, C))
//   B = minmax(B, C)
```

`minmax` is associative, and so `fMMCombiner` is also.

The accumulator function and combiner function together must obey the *basic folding rule.* That is, if *A* and *B* are accumulator data items, *A* has been initialized by the initializer function and may have been passed to the accumulator function zero or more times, *B* has not been initialized, and *args* is the list of input arguments and special arguments for a particular call to the accumulator function, then the following two code sequences must set *A* to the same value:

```
accumulatorName(&A, args);  // statement 1
```

```
initializerName(&B);        // statement 2
accumulatorName(&B, args);  // statement 3
combinerName(&A, &B);       // statement 4
```

Example: In the addint kernel, for an input value *V*:

- Statement 1 is the same as `A += V`
- Statement 2 is the same as `B = 0`
- Statement 3 is the same as `B += V`, which is the same as `B = V`
- Statement 4 is the same as `A += B`, which is the same as `A += V`

Statements 1 and 4 set *A* to the same value, and so this kernel obeys the basic folding rule.

Example: In the findMinAndMax kernel, for an input value *V* at coordinate *X*:

- Statement 1 is the same as `A = minmax(A, IndexedVal(V, X))`
- Statement 2 is the same as `B = INITVAL`
- Statement 3 is the same as

  ```
  B = minmax(B, IndexedVal(V, X))
  ```

  which, because *B* is the initial value, is the same as

  ```
  B = IndexedVal(V, X)
  ```

- Statement 4 is the same as

```
A = minmax(A, B)
```

which is the same as

```
A = minmax(A, IndexedVal(V, X))
```

Statements 1 and 4 set *A* to the same value, and so this kernel obeys the basic folding rule.

Calling a reduction kernel from Java code

For a reduction kernel named *kernelName* defined in the file `filename.rs`, there are three methods reflected in the class `ScriptC_filename`:

```
// Method 1
public javaFutureType reduce_kernelName(Allocation ain1, …,
                                        Allocation ainN);

// Method 2
public javaFutureType reduce_kernelName(Allocation ain1, …,
                                        Allocation ainN,
                                        Script.LaunchOptions sc);

// Method 3
public javaFutureType reduce_kernelName(devecSiIn1Type[] in1, …,
                                        devecSiInNType[] inN);
```

Here are some examples of calling the addint kernel:

```
ScriptC_example script = new ScriptC_example(mRenderScript);

// 1D array
//   and obtain answer immediately
int input1[] = …;
int sum1 = script.reduce_addint(input1).get();  // Method 3

// 2D allocation
//   and do some additional work before obtaining answer
Type.Builder typeBuilder =
  new Type.Builder(RS, Element.I32(RS));
typeBuilder.setX(…);
typeBuilder.setY(…);
Allocation input2 = createTyped(RS, typeBuilder.create());
populateSomehow(input2);  // fill in input Allocation with data
script.result_int result2 = script.reduce_addint(input2);  // Method 1
doSomeAdditionalWork(); // might run at same time as reduction
int sum2 = result2.get();
```

Method 1 has one input `Allocation` argument for every input argument in the kernel's accumulator function. The RenderScript runtime checks to ensure that all of the input Allocations have the same dimensions and that the `Element` type of each of the input Allocations matches that of the corresponding input argument of the accumulator function's prototype. If any of these checks fail, RenderScript throws an exception. The kernel executes over every coordinate in those dimensions.

Method 2 is the same as Method 1 except that Method 2 takes an additional argument `sc` that can be used to limit the kernel execution to a subset of the coordinates.

Method 3 is the same as Method 1 except that instead of taking Allocation inputs it takes Java array inputs. This is a convenience that saves you from having to write code to explicitly create an Allocation and copy data to it from a Java array. *However, using Method 3 instead of Method 1 does not increase the performance of the code.* For each input array, Method 3 creates a temporary 1-dimensional Allocation with the appropriate `Element` type and `setAutoPadding(boolean)` enabled, and copies the array to the Allocation as if by the appropriate `copyFrom()` method of `Allocation`. It then calls Method 1, passing those temporary Allocations.

> **NOTE:** If your application will make multiple kernel calls with the same array, or with different arrays of the same dimensions and Element type, you may improve performance by explicitly creating, populating, and reusing Allocations yourself, instead of by using Method 3.

javaFutureType, the return type of the reflected reduction methods, is a reflected static nested class within the `ScriptC_filename` class. It represents the future result of a reduction kernel run. To obtain the actual result of the run, call the `get()` method of that class, which returns a value of type *javaResultType*. `get()` is synchronous.

```
public class ScriptC_filename extends ScriptC {
  public static class javaFutureType {
    public javaResultType get() { … }
  }
}
```

javaResultType is determined from the *resultType* of the outconverter function. Unless *resultType* is an unsigned type (scalar, vector, or array), *javaResultType* is the directly corresponding Java type. If *resultType* is an unsigned type and there is a larger Java signed type, then *javaResultType* is that larger Java signed type; otherwise, it is the directly corresponding Java type. For example:

- If *resultType* is `int`, `int2`, or `int[15]`, then *javaResultType* is `int`, `Int2`, or `int[]`. All values of *resultType* can be represented by *javaResultType*.
- If *resultType* is `uint`, `uint2`, or `uint[15]`, then *javaResultType* is `long`, `Long2`, or `long[]`. All values of *resultType* can be represented by *javaResultType*.
- If *resultType* is `ulong`, `ulong2`, or `ulong[15]`, then *javaResultType* is `long`, `Long2`, or `long[]`. There are certain values of *resultType* that cannot be represented by *javaResultType*.

javaFutureType is the future result type corresponding to the *resultType* of the outconverter function.

- If *resultType* is not an array type, then *javaFutureType* is `result_resultType`.
- If *resultType* is an array of length *Count* with members of type *memberType*, then *javaFutureType* is `resultArrayCount_memberType`.

For example:

```
public class ScriptC_filename extends ScriptC {
  // for kernels with int result
  public static class result_int {
    public int get() { … }
  }

  // for kernels with int[10] result
  public static class resultArray10_int {
    public int[] get() { … }
  }

  // for kernels with int2 result
  //   note that the Java type name "Int2" is not the same as the script type name "int2"
  public static class result_int2 {
    public Int2 get() { … }
  }

  // for kernels with int2[10] result
  //   note that the Java type name "Int2" is not the same as the script type name "int2"
  public static class resultArray10_int2 {
    public Int2[] get() { … }
  }

  // for kernels with uint result
  //   note that the Java type "long" is a wider signed type than the unsigned script type "uint"
  public static class result_uint {
    public long get() { … }
  }

  // for kernels with uint[10] result
  //   note that the Java type "long" is a wider signed type than the unsigned script type "uint"
  public static class resultArray10_uint {
    public long[] get() { … }
  }

  // for kernels with uint2 result
  //   note that the Java type "Long2" is a wider signed type than the unsigned script type "uint2"
  public static class result_uint2 {
    public Long2 get() { … }
  }

  // for kernels with uint2[10] result
  //   note that the Java type "Long2" is a wider signed type than the unsigned script type "uint2"
  public static class resultArray10_uint2 {
    public Long2[] get() { … }
  }
}
```

If *javaResultType* is an object type (including an array type), each call to `javaFutureType.get()` on the same instance will return the same object.

If *javaResultType* cannot represent all values of type *resultType*, and a reduction kernel produces an unrepresentible value, then `javaFutureType.get()` throws an exception.

Method 3 and *devecSiInXType*

devecSiInXType is the Java type corresponding to the *inXType* of the corresponding argument of the accumulator function. Unless *inXType* is an unsigned type or a vector type, *devecSiInXType* is the directly corresponding Java type. If *inXType* is an unsigned scalar type, then *devecSiInXType* is the Java type directly corresponding to the signed scalar type of the same size. If *inXType* is a signed vector type, then *devecSiInXType* is the Java type directly corresponding to the vector component type. If *inXType* is an unsigned vector type, then *devecSiInXType* is the Java type directly corresponding to the signed scalar type of the same size as the vector component type. For example:

- If *inXType* is `int`, then *devecSiInXType* is `int`.
- If *inXType* is `int2`, then *devecSiInXType* is `int`. The array is a *flattened* representation: It has twice as many *scalar* Elements as the Allocation has 2-component *vector* Elements. This is the same way that the `copyFrom()` methods of `Allocation` work.

- If *inXType* is `uint`, then *deviceSiInXType* is `int`. A signed value in the Java array is interpreted as an unsigned value of the same bitpattern in the Allocation. This is the same way that the `copyFrom()` methods of `Allocation` work.
- If *inXType* is `uint2`, then *deviceSiInXType* is `int`. This is a combination of the way `int2` and `uint` are handled: The array is a flattened representation, and Java array signed values are interpreted as RenderScript unsigned Element values.

Note that for Method 3, input types are handled differently than result types:

- A script's vector input is flattened on the Java side, whereas a script's vector result is not.
- A script's unsigned input is represented as a signed input of the same size on the Java side, whereas a script's unsigned result is represented as a widened signed type on the Java side (except in the case of `ulong`).

More example reduction kernels

```
#pragma rs reduce(dotProduct) \
  accumulator(dotProductAccum) combiner(dotProductSum)

// Note: No initializer function -- therefore,
// each accumulator data item is implicitly initialized to 0.0f.

static void dotProductAccum(float *accum, float in1, float in2) {
  *accum += in1*in2;
}

// combiner function
static void dotProductSum(float *accum, const float *val) {
  *accum += *val;
}
```

```
// Find a zero Element in a 2D allocation; return (-1, -1) if none
#pragma rs reduce(fz2) \
  initializer(fz2Init) \
  accumulator(fz2Accum) combiner(fz2Combine)

static void fz2Init(int2 *accum) { accum->x = accum->y = -1; }

static void fz2Accum(int2 *accum,
                     int inVal,
                     int x /* special arg */,
                     int y /* special arg */) {
  if (inVal==0) {
    accum->x = x;
    accum->y = y;
  }
}

static void fz2Combine(int2 *accum, const int2 *accum2) {
  if (accum2->x >= 0) *accum = *accum2;
}
```

```
// Note that this kernel returns an array to Java
#pragma rs reduce(histogram) \
  accumulator(hsgAccum) combiner(hsgCombine)

#define BUCKETS 256
typedef uint32_t Histogram[BUCKETS];

// Note: No initializer function --
// therefore, each bucket is implicitly initialized to 0.

static void hsgAccum(Histogram *h, uchar in) { ++(*h)[in]; }

static void hsgCombine(Histogram *accum,
                       const Histogram *addend) {
  for (int i = 0; i < BUCKETS; ++i)
    (*accum)[i] += (*addend)[i];
}

// Determines the mode (most frequently occurring value), and returns
// the value and the frequency.
//
// If multiple values have the same highest frequency, returns the lowest
// of those values.
//
// Shares functions with the histogram reduction kernel.
#pragma rs reduce(mode) \
  accumulator(hsgAccum) combiner(hsgCombine) \
  outconverter(modeOutConvert)

static void modeOutConvert(int2 *result, const Histogram *h) {
  uint32_t mode = 0;
  for (int i = 1; i < BUCKETS; ++i)
    if ((*h)[i] > (*h)[mode]) mode = i;
  result->x = mode;
  result->y = (*h)[mode];
}
```