Google I/O 13

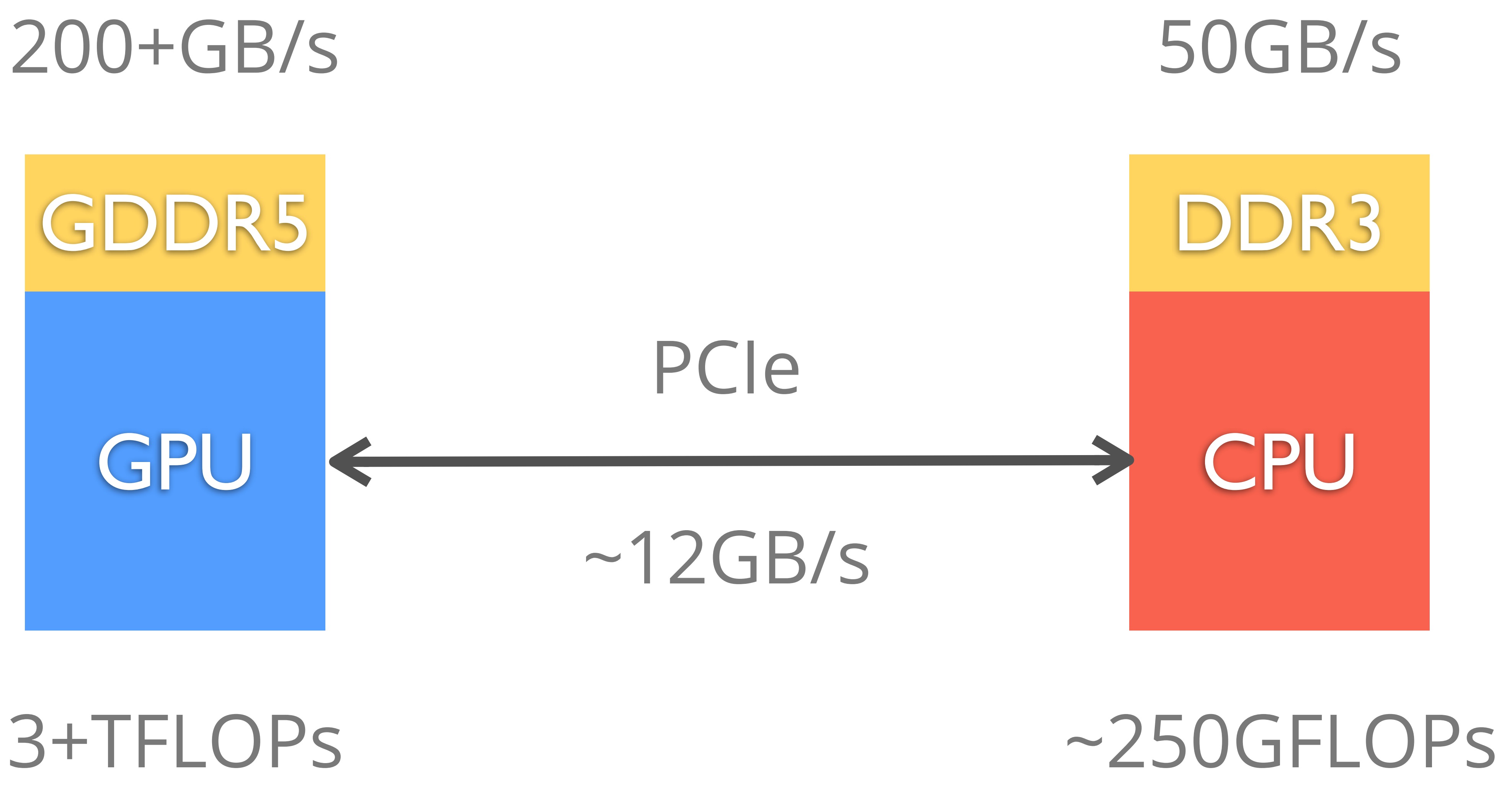
High Performance Apps With RenderScript

Tim Murray and Jason Sams
Google

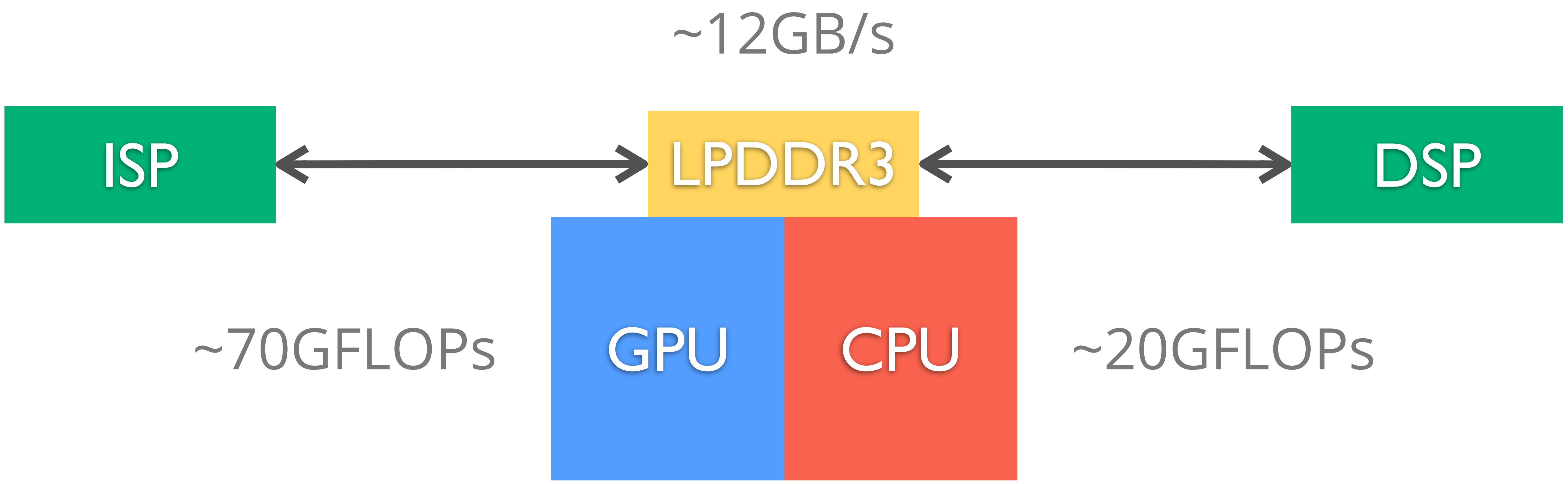
Some GPU Background

- GPUs have become useful for a lot more than just graphics
  - Lots of FLOPs versus CPUs
  - Great at many data parallel tasks
- Growth of GPU compute largely spurred by HPC adoption
- Programmable GPUs are arriving on tablets and phones

Desktop/Server System Architecture



Mobile System Architecture



Mobile vs Desktop: Very Different

- Mobile has a lot of architectural diversity
  - Lots of CPUs, even more GPUs
  - GPUs in particular are extremely different between vendors
- System resources vary statically and dynamically
  - CPU/GPU clock speed? Battery? Processor load? Display resolution? Additional processors?

- Goal: high performance applications without sacrificing performance portability

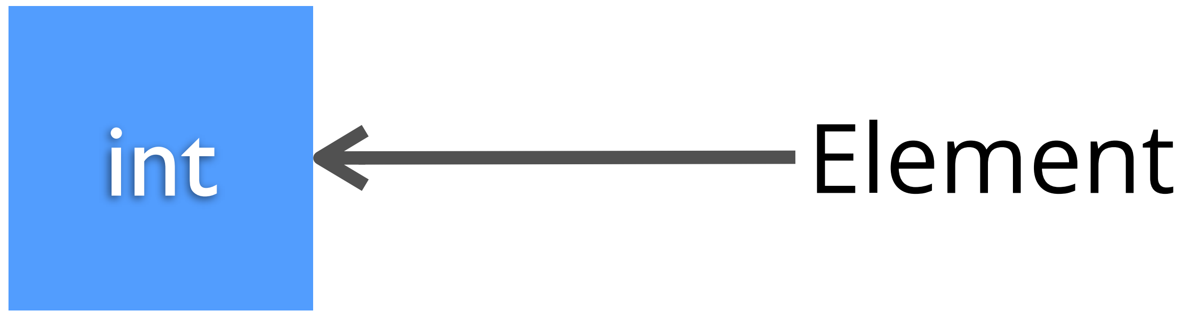
RenderScript

- Platform for high-performance computing across different hardware
  - API focused on entire system, not specific devices
  - Runtime handles processor selection dynamically
- Provides a consistent target that runs well across all SoCs
  - Performance is the objective, not running on a specific processor
- Influenced by other data-parallel runtimes

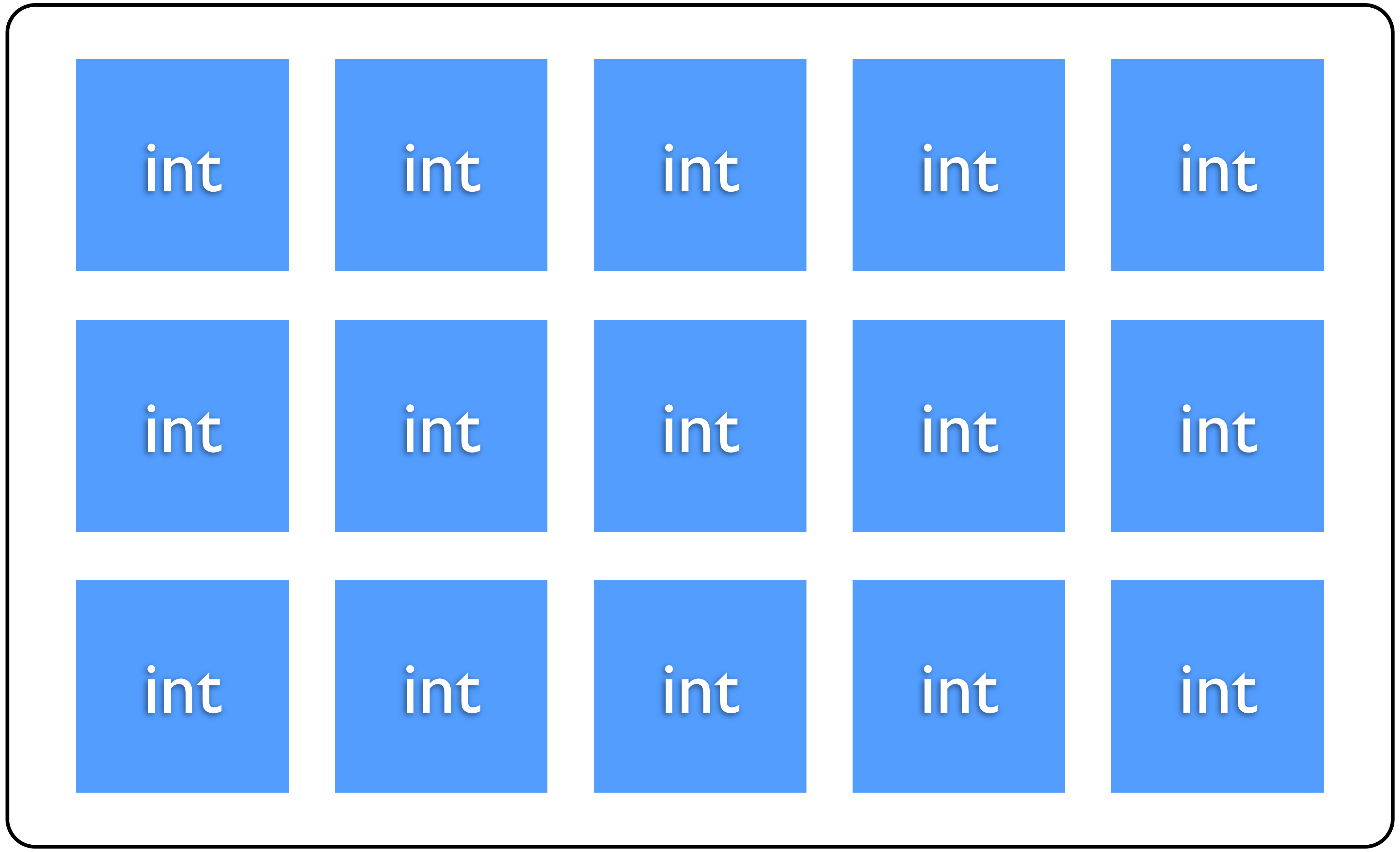
RS at 10,000 Feet

- Performance-critical kernels are written in a C99-based language
  - Kernels distributed with APK as architecture-independent bitcode
  - Compiled from bitcode to one or more processor targets at runtime
- Java classes automatically generated for easy integration with existing applications
- Resource management and execution handled by Java APIs

Basic RS

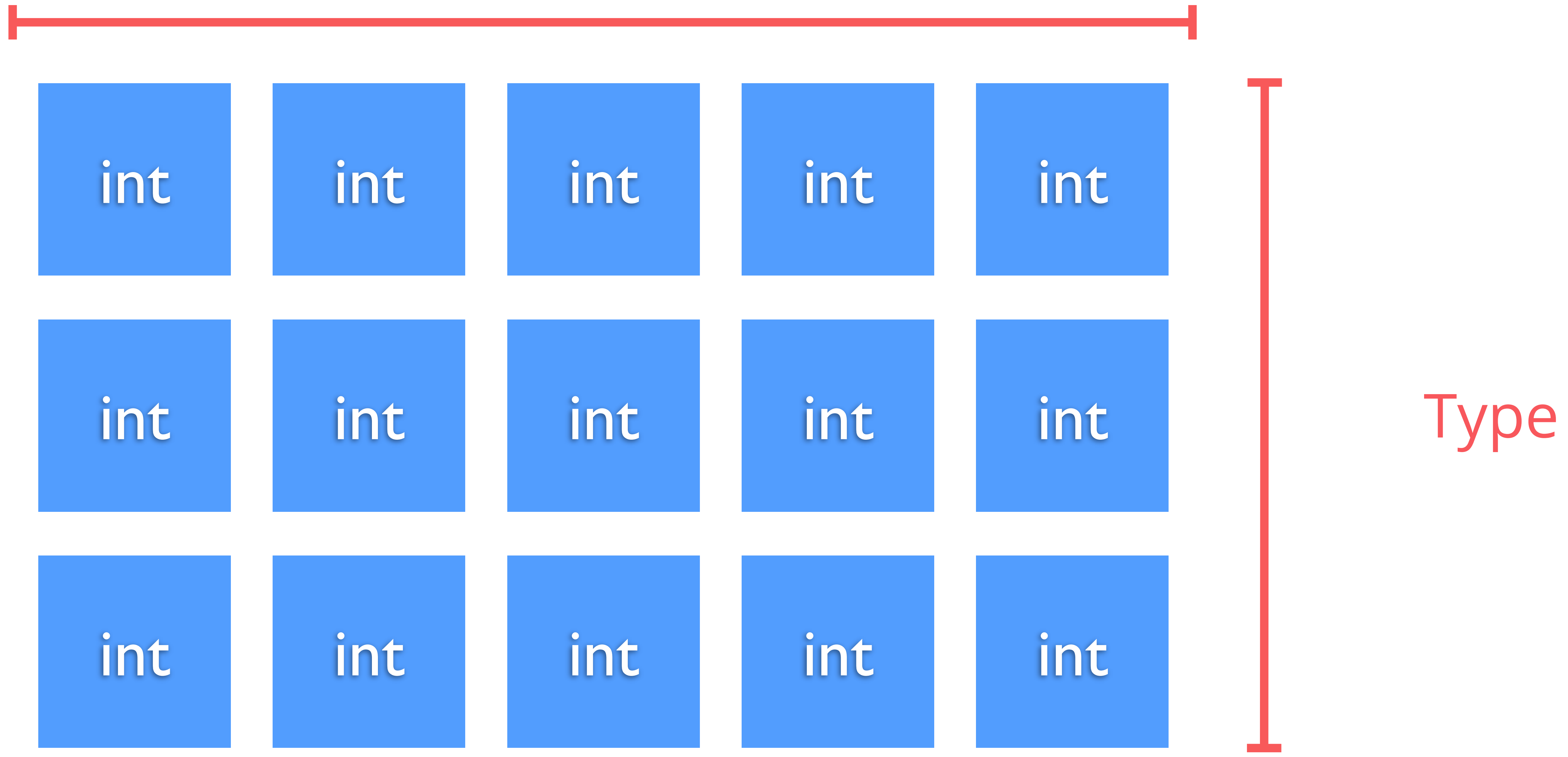


Basic RS



Allocation

Basic RS



Basic RS

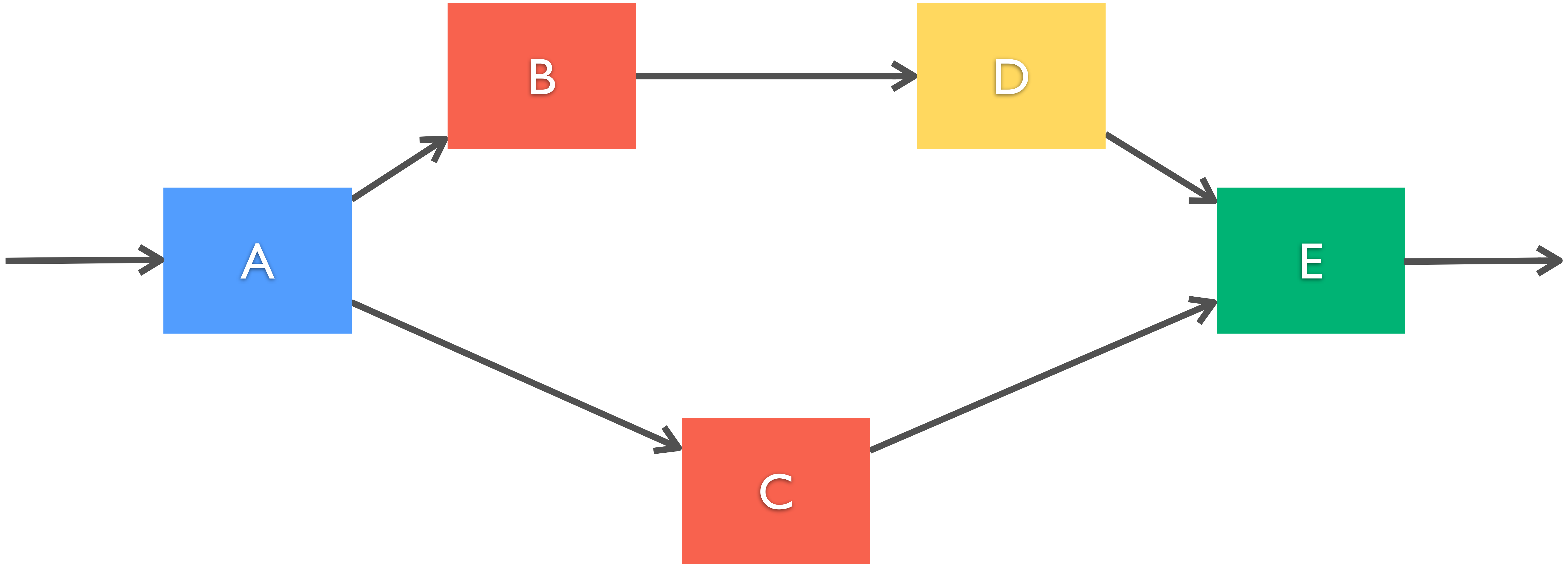
RS

```
#pragma version(1) // RS language revision, 1 for now
#pragma rs java_package_name(com.rs.io2013) // Java package used for reflected classes

int addVal = 3; // script global, local to script, reflects set_addVal(int) to Java

// a kernel; reflects forEach_kernel(Allocation, Allocation)
int __attribute__((kernel)) kernel(int in, uint32_t x, uint32_t y) {
    return in + addVal;
}
```

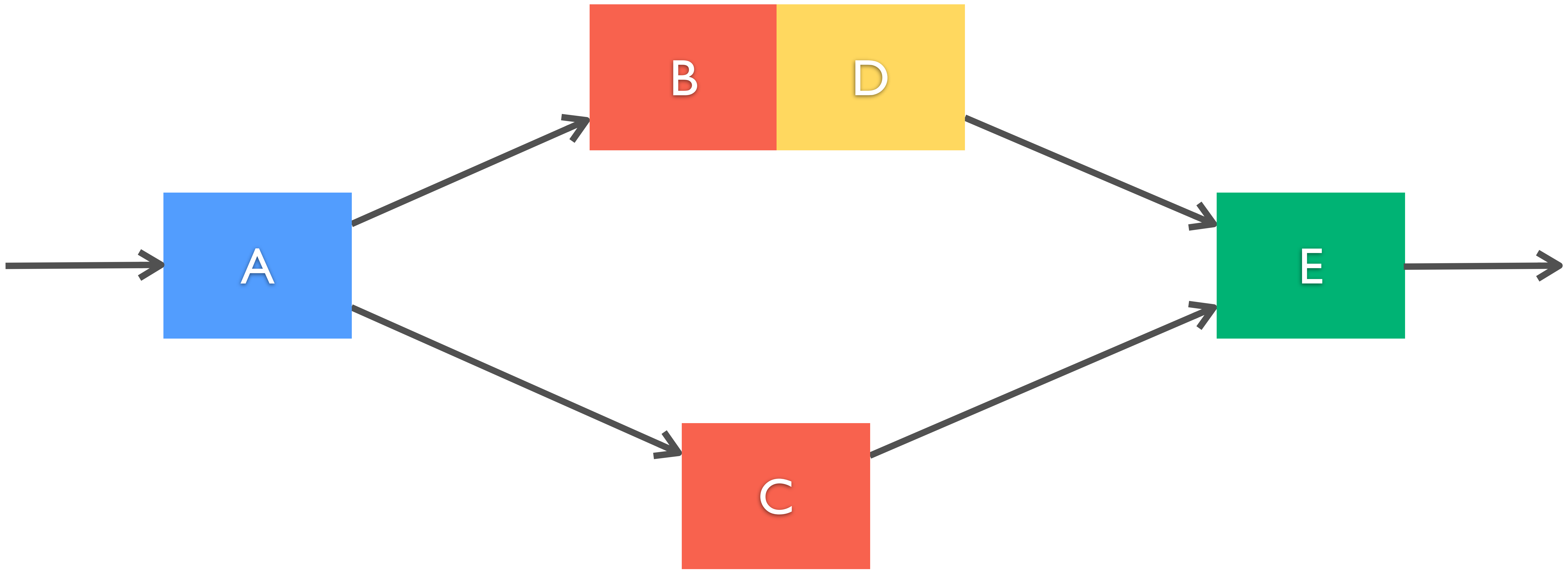
Script Groups

- Allows a group of kernels to be executed as a single call
- Enables the driver to perform various optimizations by knowing the entire workload
    - Scheduling across devices, tiling, kernel fusion…
- Significantly faster in some cases today than individual scripts

Script Groups: An Example



Script Groups: An Example



Features Coming Soon

- Compatibility library for Gingerbread
- rsSetElementAt_<type>
- Debug runtime
- Additional script intrinsics
- YUV Allocations
- Improved launch latency

RS Compatibility Library

- Enables modern RS on devices running Gingerbread or higher
- CPU only, compiles kernels offline for a specific CPU architecture
- Devices running modern Android will use the native RS library
    - With GPU acceleration on appropriate devices

App Walkthroughs: Gaussian Blur, Histogram

- RS is very good at common image processing tasks
- Gaussian blur is a common task where RS has built-in support
- Histogram demonstrates some more advanced techniques
    - Clipped kernel launches
    - rsSetElementAt
    - Multi-pass processing

General Image Processing in RS

Step 0: Setup

Java

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

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

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

Simple Blur in RS

Java

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

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

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

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

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

Histogram

- The algorithm presented does a two step reduction
  - The first pass uses a large group of workers
  - Each worker has its own independent sum
- A second pass sums the sums from first pass
- Will ignore the rendering pass to save time

Histogram

Script Globals

RS

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

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

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

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

Histogram

Script Globals Continued

RS

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

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

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

Histogram

Kernel for First Pass

RS

```
// The start of the first kernel
void __attribute__((kernel)) pass1(int in, uint x, uint y) {
    // Note that x and y will indicate the coordinates of the pixel being processed

    // This kernel will be run on a range of x = [0] and y = [0 to steps]

    // Clear our output buffer for this thread.
    for (int i=0; i < 256; i++) {

        // Set the value at i,y to 0
        rsSetElementAt_int(gSums, 0, i, y);
    }
```

Histogram

Kernel for First Pass Continued

RS

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

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

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

Histogram

Kernel for Second Pass

RS

```
// This kernel is run on the Sum allocation, so its called once
// for each of the 256 levels

// Note, this is a 1D kernel
int __attribute__((kernel)) pass2(uint x) {
    int sum = 0;

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

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

Histogram
Rescale Function

RS

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

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

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

Histogram

Java Setup

Java

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

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

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

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

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

Histogram

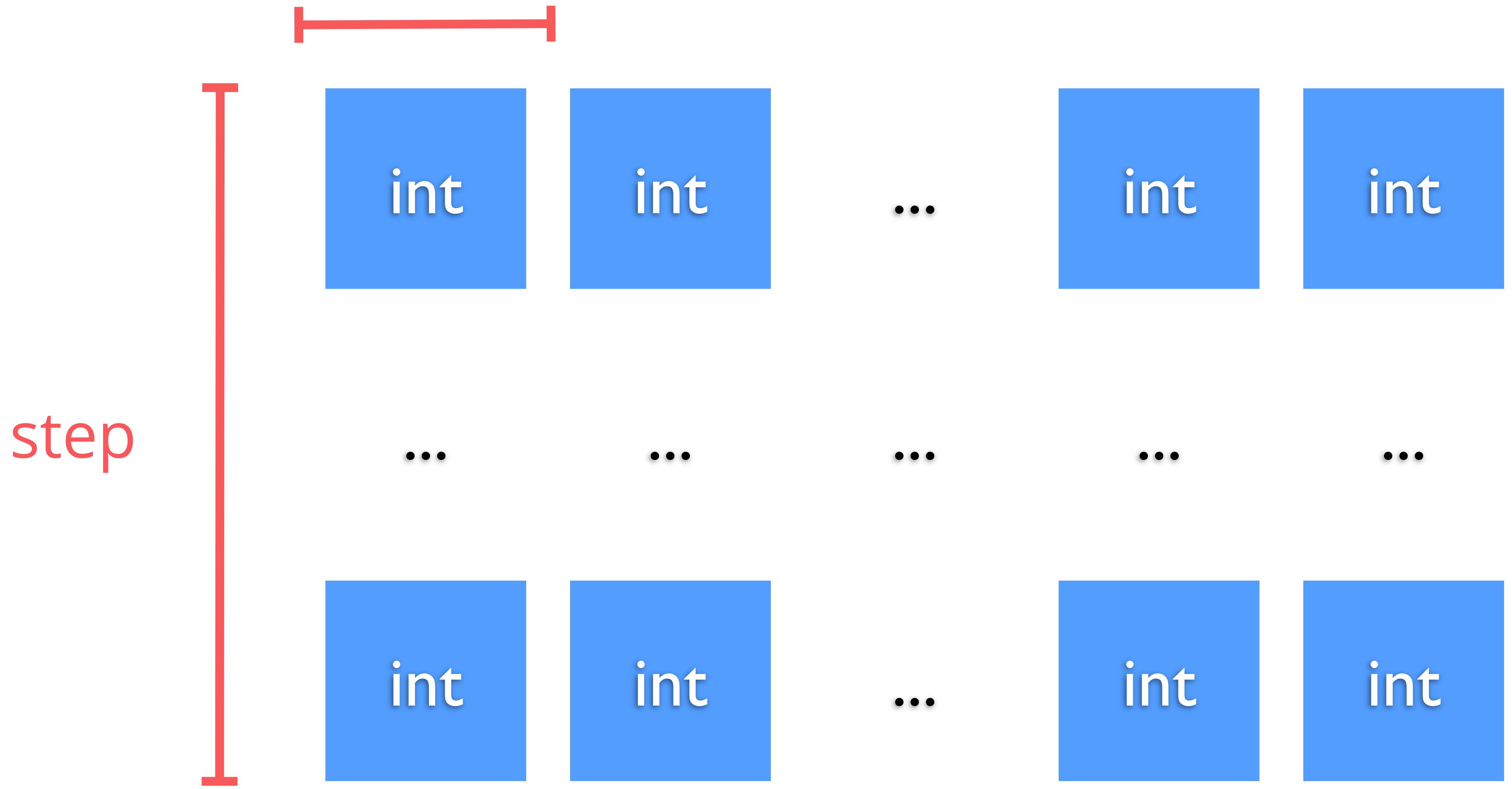
Java: Running the First Pass

Java

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

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

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

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

Histogram

Java: Running the First Pass

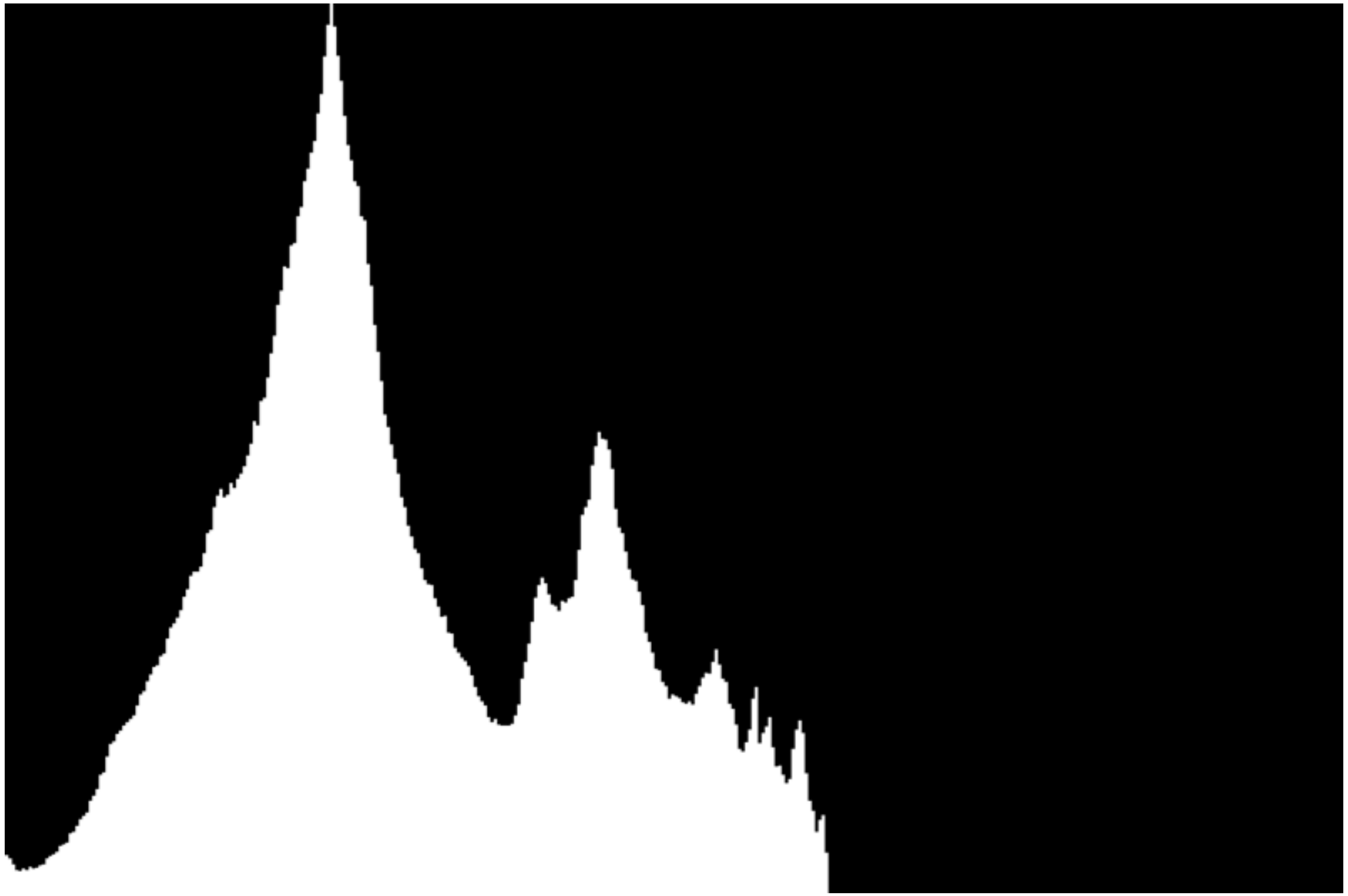


Histogram

Java: Running the Second Pass

Java

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

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

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



<Thank You!>