Efficient Parallel Computation on Android
Mobile Landscape Today

- Mobile applications continue to grow more resource-intensive
- Mobile architectures are including more types of processors
- GPUs and DSPs are already interesting targets for mobile developers
An Intro To RenderScript

- Android’s platform for parallel computation
- Different approach to parallel computation than existing APIs
- Mobile has unique constraints that required something new
Constraint #1: Device Compatibility

- ~1B Android devices currently in users’ hands
- App developers want to target as much of this as possible
- Huge variety of devices and capabilities
  - ARM is dominant CPU ISA
  - No dominant GPU vendor, definitely no dominant GPU architecture
  - Dominant OS version is usually ~2 versions behind latest
  - Most devices still sold without compute-capable GPUs
Constraint #2: Different Developer Audience

• Average HPC developer: familiar with FORTRAN, MPI, OpenMP, maybe others
  - Cares primarily about specific platforms
  - Very familiar with performance tuning of large codebases

• Average Android developer: familiar with Java, maybe some C/C++
  - Cares primarily about broad adoption, not specific devices
  - Probably not familiar with much low-level performance tuning
Constraint #3: Architectural Diversity

• More vendors in mobile than in desktop/HPC
  - Desktop/HPC: AMD, Intel, NVIDIA
  - Mobile: ARM, Imagination, Intel, Qualcomm, NVIDIA, ...

• Different vendors’ architectures are substantially more different in mobile
  - Available floating point precision? Double precision support?
  - Scalar vs vector, VLIW vs superscalar vs in-order, ...
  - Resources: # of independent processors, warp/wavefront width, shared memory amount?
  - Constraints: maximum block size?

• Apps need to run everywhere
GPU Occupancy

- Occupancy: how many blocks can fit on a single processor at a time
- One metric of how much work the GPU is doing at a time
  - Not the only metric, not even the best metric
  - see everything written by Vasily Volkov
- At least one block **must** fit in order to run
- Really hard to guarantee minimum occupancy across mobile GPUs
  - Can't use more than minimum max amount of shared memory per block on any arch
  - Can't use more than minimum max block size on any arch
GPU Functional Differences

- Some devices support double precision, many don’t
- Precision varies a lot between devices
  - Some GPUs are full IEEE 754, others are not
  - ARM NEON actually isn’t full IEEE 754 pre-AArch64
- Other discrepancies between architectures
  - Work creation? printf support?
  - Warp-synchronous programming is totally out
Architectural Consistencies?

- Unified memory, no copies required
  - No more PCIe!
  - Usually same bandwidth between CPU and GPU
- Flush caches to move data between devices
  - alternately, use cache coherence
- More processors than just CPU and GPU
  - ISP on camera, maybe multiple DSPs
  - Interesting tradeoffs for perf/W versus raw perf
GPU or CPU?

- HPC: write both, benchmark, pick which one is faster
- Mobile: that approach doesn’t really work, even for a single SoC

- Reminder: mobile GPUs don’t have nearly the advantage over CPUs like in desktop
- Mobile GPUs are almost always busy rendering at least the display
- Mobile displays have a lot of pixels, especially when you consider GPU FLOPs
Mobile Performance vs Display Resolution

• Steam Hardware Survey, February 2014:
  - Most common display resolution: 1920x1080
  - Most common GPU: Intel HD 4000 (~300 GFLOPS)
  - Most common discrete GPU: GeForce GTX 660 (~1900 GFLOPS)
• Nexus 5, released November 2013:
  - Display: 1920x1080
  - GPU: Qualcomm Adreno 330 (~130 GFLOPS)
• Moto G, released December 2013:
  - Display: 720p
  - GPU: Qualcomm Adreno 305 (~21 GFLOPS)
Constraint #4: Power and Thermal Limits

• Average HPC node: dedicated cooling system, guaranteed amount of power
  - Clocks are (mostly) fixed
  - Initial performance == sustained performance
• Mobile: forget everything you know about performance from HPC
Mobile Performance Realities

- Mobile SoCs are optimized for bursty workloads because that’s what users see
  - “Hurry up and sleep”
  - Will hit thermal throttling within a few seconds
  - Thermal throttling == lower clocks on CPU, GPU, and memory
  - No fixed amount or ratio for how much clocks will decrease

- Different SoCs hit thermal throttling at different times with different workloads

- Even if you don’t hit thermal throttling, using more power will kill battery life
  - Nobody likes lowered battery life caused by an app
Six seconds
Device Compatibility?

• Everything has to be able to run on every device, including those with old OS versions
• Can’t assume anything about GPU compute support, so CPU performance is important
• Has to be something that can be packaged entirely with an application, so can’t be too big
  - if we’re successful, many copies on a low-end phone
  - probably don’t want to put all of Clang in every app
Different Audience?

- Java API is most important, C API is a close second
- Ease of use and ability to integrate with existing apps is paramount
- Make common cases very fast and very easy
  - Major opportunity to increase adoption of GPU compute
- Make harder cases possible and run as well as possible
Architectural Diversity?

- Don’t expose language or runtime features that break compatibility
- No blocks, no grids, no shared memory, definitely no warp-synchronous programming
  - Introduce higher-level primitives necessary to use those sorts of HW features
- Memory must be known to our runtime in order to be accessible

- Focus on making sure everything works initially rather than peak performance
Power and Thermal Limits?

• Assume migration between processors is cheap thanks to unified memory
• Let the runtime schedule work on specific processors rather than developers
  - Migrate to a slower but more energy efficient processor if the system is thermally constrained
  - Don’t put more work on a processor that’s already overloaded
• Let developers inform runtime about how to optimize
  - “Does my app care about performance, power consumption, latency, something else?”
RenderScript

- Built around these requirements
- Designed to let compute workloads run portably with good performance on any SoC
- Letting people tune for absolute peak performance is not our #1 goal
RenderScript API

- Both Java and C++ APIs available to all developers
- Will look familiar to anyone that’s used CUDA or OpenCL
- Allocate memory, copy buffers, launch work, synchronize
- All work within a context is sequential
  - No streams, no asynchronous command queues
Intrinsics

- Built-in high-level library functions for simple image processing workloads
  - Android apps do a lot of image processing
- Driver ships heavily optimized versions of these
  - Multicore NEON or SSE, GPU assembly, DSP assembly, ...
- Really easy to use
- Much, much faster than Java or unoptimized C
RenderScript Kernel Language

- It’s C99 with extensions
- No blocks, no grids, no __shared__, no warps, no barriers
- Both sequential and parallel kernels
- Compiled into LLVM bitcode and JIT’d to appropriate arch on device
- Reflected Java/C++ classes for API integration
What About Devices?

• Your work will run on some device, but you don’t know which
  - No device properties, no device selection
  - Left up to vendor drivers on a given SoC
• In practice, which device runs a kernel can and will change, even on a given SoC
  - Especially true as amount of devices grows with dark silicon
Script Groups

• Higher-level primitive to get better performance
• DAG of kernels passed to the runtime as one atomic unit
• Any state not specified as an output is not visible to developer
• Enables lots of optimizations for vendor drivers
  - Kernel fusion, shared memory, generation of warp-synchronous code, concurrent processor use, better power management, ...
Support Library

• Run modern RS back to Android 2.2 (Froyo)
• Bundles both precompiled CPU binaries with LLVM bitcode
  - One package can run with CPU support path on 2.2 and GPU JIT on 4.4
• Same API exposed to developers
• Really successful in past six months
Performance
Desktop Performance Today

• Desktop performance today
• Roughly ~75 to ~300 watts
• GPUs have a clear advantage
Desktop & Mobile Performance Today

- Mobile still well behind desktop performance
- Power limited
- Can only do so much with ~5 watts
Mobile Performance Today

- Mobile SoCs tend to be “CPU heavy”
- Resources are shared
  - Neither has a bandwidth advantage
  - Neither has a thermal budget advantage
- FLOPS are closer than desktop
Mobile Performance Today (shipping)

- Mobile SoCs tend to be “CPU heavy”
- Resources are shared
  - Neither has a bandwidth advantage
  - Neither has a thermal budget advantage
- FLOPS are closer than desktop
Usable Mobile Performance Today

- Drawing the screen also takes FLOPS
  - ~208 per pixel for a simple application
- Only the leftovers are available for compute
- Graphically rich applications will use more
Performance Trends

• Screen resolution is still growing
• Power is not
• Performance improvements are coming from perf/W improvements
  - Gains here are typically slow
  - More specialized processors are likely
Tegra K1?

• NVIDIA allowed us to run some of our RenderScript tests on a pre-production K1 device
  - Including pre-production RenderScript GPU driver
• Ran our internal benchmark called “IP-18”
  - Image processing workload with some synthetic tests
  - Some tests derived from actual shipping workloads such as Gallery
  - Tegra 3 used as baseline
• Test source is available in AOSP
  - platform/frameworks/rs/java/tests/ImageProcessing jb
<table>
<thead>
<tr>
<th>Feature</th>
<th>Tegra K1</th>
<th>Tegra 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Levels (Vec4 Relaxed)</td>
<td>10</td>
<td>3.5</td>
</tr>
<tr>
<td>Blur radius (25)</td>
<td>5</td>
<td>1.6</td>
</tr>
<tr>
<td>Greyscale</td>
<td>10</td>
<td>3.6</td>
</tr>
<tr>
<td>Grain</td>
<td>9</td>
<td>3.0</td>
</tr>
<tr>
<td>Fisheye Approximate (Relaxed)</td>
<td>9</td>
<td>3.0</td>
</tr>
<tr>
<td>Vignette Approximate (Relaxed)</td>
<td>9</td>
<td>3.0</td>
</tr>
<tr>
<td>Convolve (3x3)</td>
<td>10</td>
<td>3.3</td>
</tr>
<tr>
<td>ColorMatrix</td>
<td>5</td>
<td>1.8</td>
</tr>
<tr>
<td>Copy</td>
<td>10</td>
<td>3.3</td>
</tr>
<tr>
<td>Convolve (5x5)</td>
<td>12</td>
<td>4.0</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>12</td>
<td>4.0</td>
</tr>
<tr>
<td>Vibrance</td>
<td>10</td>
<td>3.3</td>
</tr>
<tr>
<td>BW Filter</td>
<td>13</td>
<td>4.5</td>
</tr>
<tr>
<td>Shadows</td>
<td>10</td>
<td>3.3</td>
</tr>
<tr>
<td>Contrast</td>
<td>13</td>
<td>4.5</td>
</tr>
<tr>
<td>Exposure</td>
<td>8</td>
<td>2.6</td>
</tr>
<tr>
<td>Color Cube</td>
<td>6</td>
<td>1.9</td>
</tr>
</tbody>
</table>

Normalized Performance vs Tegra 3
Performance Lessons

• Operations that were already bandwidth limited don’t get a lot faster
  - Color Matrix
  - Just not enough math per load
• Operations that use expensive math ops show big increases
  - Vignette
  - CPUs not ideal for pow() or transcendental functions
• Texture ops still not tuned in the driver
  - Fisheye
  - Expect large improvements as the driver improves use of texture sampler
Questions?