HCL’s capability in Android Performance Optimization

Table of Contents

Objective

Though Android has matured over period of time but still systems going into Production with Android has performance considerations to meet certain minimum criteria especially if iOS is the competitor. These optimization vary from system to system and are dependent of number of factors like OS version, HW components, CPU, GPU, Memory etc. Apart from these limiting factors, there are some provisions available in the system which lets you improve Graphics, I/O and Multimedia performance of production ready android systems. Though most of the performance improvements suggested in this paper are tested while running Android in virtualized environment with focus on future products but they should be applicable on systems running Android natively.

Android Overview

Android is an OS based on linux designed primarily for mobile devices but marking its presence into various consumer products. Android was unveiled in 2007 along with the founding of the OHA with HTC Dream as first publicly available smartphone on Oct 2008.

The UI of Android is based on direct manipulation, using touch inputs that loosely correspond to real-world actions, like swiping, tapping, pinching and reverse pinching to manipulate on-screen objects. Internal hardware such as accelerometers, gyroscopes and proximity sensors are used by some applications to respond to additional user actions. Android allows users to customize their home screens with shortcuts to applications and widgets.

Android's source code is released by Google under Apache License which allows the software to be freely modified and distributed. Most Android devices ship with a combination of open source and proprietary software. Android is popular with technology companies who require a ready-made, low-cost and customizable operating system for high tech devices.Despite being primarily designed for phones and tablets, it also has been used in TVs, Game consoles, Digicam & other CE products. It's open nature has encouraged use of source code as a foundation for community-driven projects, which adds new features for advanced users or bring Android to devices which were officially released running other operating systems.

Android Emulator or Android in virtual machine

KVM, the Linux Kernel-based Virtual Machine, is a virtualization solution for Linux designed to be tightly integrated with upstream kernel development.

qemu-kvm provides the basic user interface for launching and controlling VMs, and provides the main framework for running VMs. qemu-kvm handles essentially all hardware emulation { when the emulated machine performs an IO instruction (such as an operation on x86 IO ports), the kernel module returns to userspace, where qemu-kvm emulates the operation and then makes a ioctl to resume execution.

With the help of qemu-kvm Android could be run in virtual machine and that's what we have done.

The Android SDK includes a virtual mobile device emulator. The emulator lets you prototype, develop and test Android applications without using a physical device.

The Android emulator mimics all of the hardware and software features of a typical mobile device. It provides a variety of navigation and control keys, which you can "press" using your mouse or keyboard to generate events for your application. It also provides a screen in which your application is displayed, together with any other active Android applications.

Performance Bottlenecks

Problem Area I : Graphics

The major area in graphics where performance is bottleneck are:

Draw Calls & Indexed Draw Calls:

Draw calls tend to be an expensive operation in the world of 3D graphics,it might be possible that you're making too many individual calls per frame and hence results in slow rendering.

Vertex Count & Indexed Vertex Count:

It might be possible that number of verticies you're pumping through is very large and hence results in reduced performance of 2D.

Buffer Creations:

Depending on your application, buffer creations should occur in the setup phase, as they tend to be a slow operation, if you find that you're creating buffers at run-time and hence results in reduced performance in 3D.

Error Gets:

It might be possible that glGetError gets called every time we draw. Hence it decreases performance drastically.

TA (Tile Accelerator) Load and USSE (Universal Scalable Shader Engine) Vertex Load:

It might be possible that if graphics scene is too complex so the time that TA is being used and the percentage of time the shader engine is processing vertex instructions becomes critical. Ideally these loads should be balanced for best performance.

TSP Load, Texture Unit Load and USSE Pixel Load:

It might be possible that if texture types are not optimized and design of fragment shaders is too much complex, then the percentage time that the Texture Shading processor and texture units are busy and the time the shader units are processing pixel instructions becomes critical.

Problem Area II : Multimedia

When running android virtually, multimedia performance might proved to be a major bottleneck.

Problem Area III : I/O

It might be possible that your app's performance is being affected by IO operations. Hence in that case it means no longer butter smooth animations effect is there.

Problem Area IV : Network

It might be possible that your app's performance is being affected by Network operations due to which App take long time to respond and user experience will not be good.

Problem Area V : Memory

This is one of the major bottleneck for the system. It might be possible that your app's memory usage may exceed the available system memory and hence results in app leaking memory, so in that case performance issues are related to lack of available system memory.

Graphics Performance Optimizations

Graphics stack is a complex software/Hardware structure that is only as strong as its weakest link. Performance bottlenecks can arise from any inconsistencies between components in the stack, and it is often very difficult to identify and correct these problems without domain-specific expertise in graphics performance optimization.

We have made changes in software algorithm as suggested below in article to optimize the graphics performance.

Different tools used for analysis

- ✓ GLTrace
- ✓ OProfile
- ✓ DDMS and adb logcat

Suggestion I : Hardware Composition

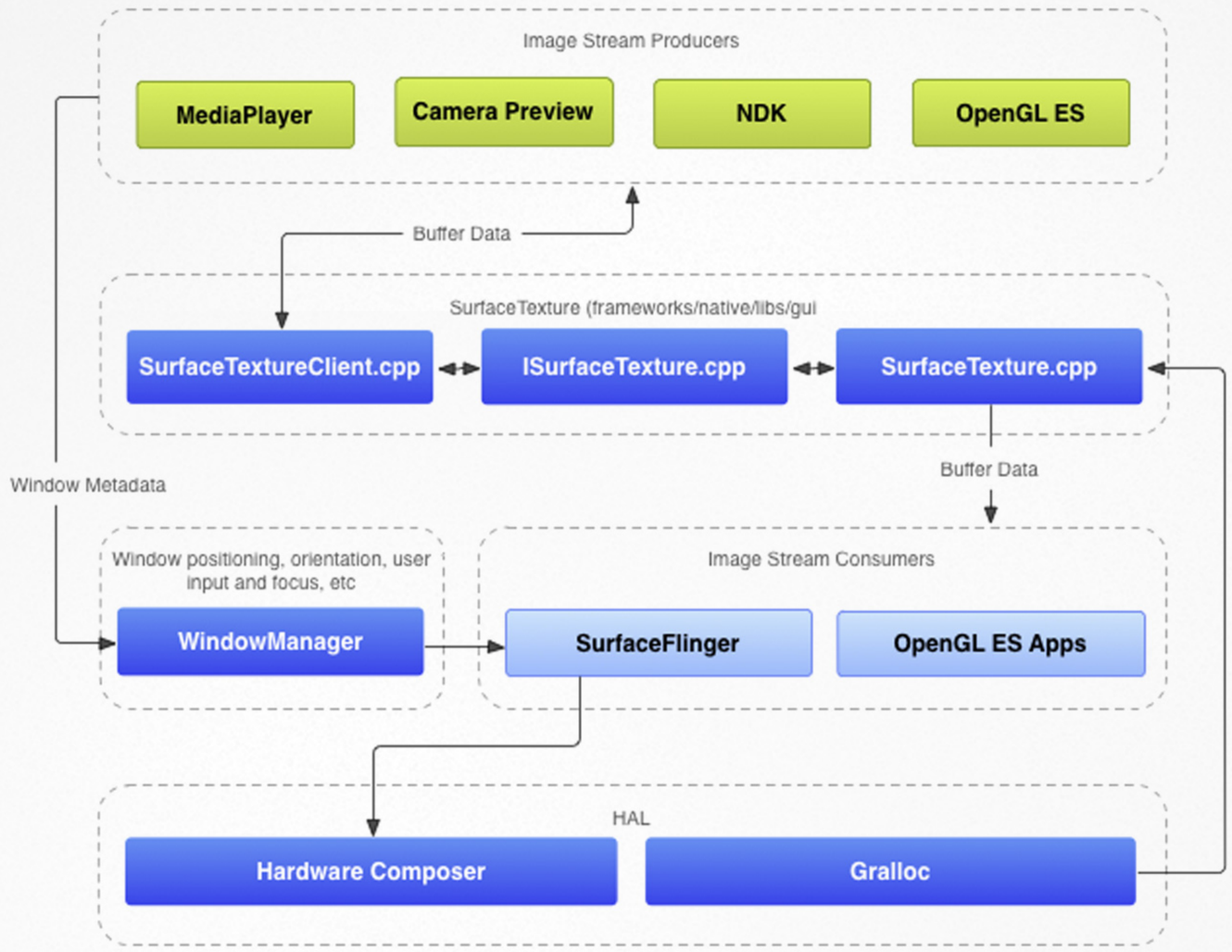
Hardware Composer defines an abstraction layer between the Android compositor Surface Flinger and SOC display subsystems. Surface Flinger can delegate certain composition work to the hardware composer to offload work from the OpenGL and the GPU.
It was originally introduced to provide SOC vendors with the ability to take advantage of:

- ✓ Additional overlay planes
- ✓ 2D Blitters

It abstracts things like overlays and 2D blitters and helps offload some things that would normally be done with OpenGL. This makes compositing faster than having SurfaceFlinger do all the work.

Major Improvements:

- ✓ Multi Display Support
- ✓ Clone/Extended Mode Support



The GFX throughput of 2.2 GB/s safely accommodates user experience:

Layer	w	h	d	op	size x 1	size x 24	size x 30	size x 60
Wallpaper	1280	672	4	2	6881280	165150720	206438400	412876800
Widgets	1280	672	4	3	10321920	247726080	309657600	619315200
Status bar	1280	48	2	2	245760	5898240	7372800	14745600
Cursor	22	28	4	3	7392	177408	221760	443520
					17448960	418775040	523468800	1046937600
GB/s					0.017449	0.41877504	0.5234688	1.0469376

layer	w	h	depth	op	size x 1	size x 24	size x 30	size x 60
wallpaper	1920	1000	4	2	15360000	368640000	460800000	921600000
widgets	1920	1000	4	3	23040000	552960000	691200000	1382400000
status bar	1920	80	2	2	614400	14745600	18432000	36864000
cursor	64	64	4	3	49152	1179648	1474560	2949120
					39014400	936345600	1170432000	2340864000
GB/s					0.039014	0.9363456	1.170432	2.340864

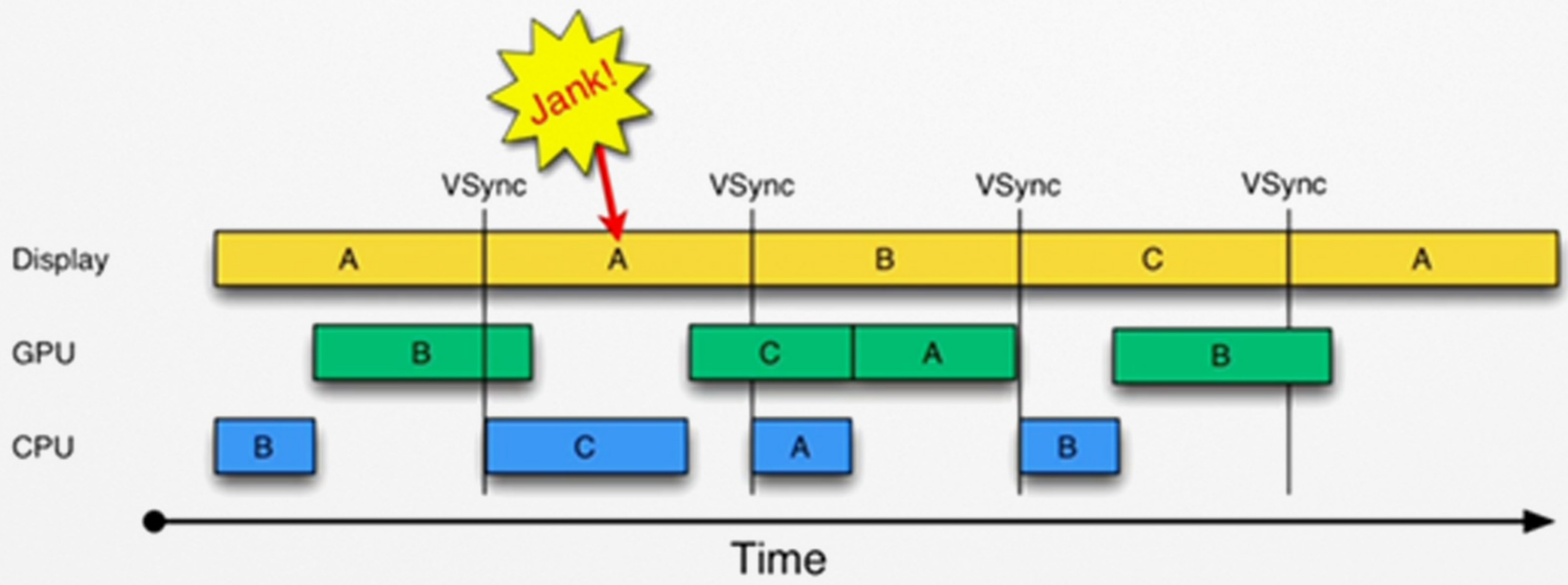
Suggestion II : Write Combining for Graphics/Display

The support for Write combining enabled Display memory in Platform display driver increases the graphics performance.

The WC memory improves the processor write performance by combining the individual writes that a processor may make to a particular memory region (like a video controllers frame buffer) into a burst write containing many aggregated individual writes. It is easy to see the performance advantages of WC when for example we change a processor that writes 32 pixels to a frame buffer via 32 individual bus transactions into a processor which writes all 32 pixels at the same time with little to no incremental transaction overhead.

Suggestion III : Enable triple buffering

Triple buffering provides a speed improvement. In double buffering the program must wait until the finished drawing is copied or swapped before starting the next drawing. This waiting period could be some milliseconds during which neither buffer can be touched. In triple buffering the program has two back buffers and can immediately start drawing in the one that is not involved in such copying. The third buffer, the front buffer, is read by the graphics hardware to display the image on the display. Once the display has been drawn, the front buffer is flipped with (or copied from) the back buffer holding the last complete screen. Since one of the back buffers is always complete, the graphics hardware never has to wait for the software to complete. Consequently, the software and the graphics hardware are completely independent, and can run at their own pace. Finally, the displayed image was started without waiting for synchronization and thus with minimum lag.



How triple buffering works:

If B takes too long, and A is in use displaying the current frame. This time though, instead of wasting processing time during the repeated buffer, the systems create a C buffer, and get to work on the next frame. Triple buffering stops the stutter fest, and after the initial skip, the user sees a smooth animation. It's all about presenting a stiff upper lip to the user even though things aren't going so smoothly under the hood.

Suggestion IV : Modifications in OpenGL calls

OpenGL never signals errors but simply records them; it's must to determine whether an error occurred. During the debugging phase, the program should call glGetError() to look for errors frequently (for example, once per redraw) until glGetError() returns GL_NO_ERROR. We found that glGetError gets called every time we draw. It is recommended calling glGetError after every gl call, but with a macro that is only defined when debugging.

http://www.mesa3d.org/brianp/sig97/perfopt.htm

http://www-f9.ijs.si/~matevz/docs/007-2392-003/sgi_html/ch15.html

Any form of glGet or glIs. Getting state values slows the application. Unless the application is a "middle ware" application, it is not required to retrieve state values. During development, however, it's quite common to call glGetError. When the application is ready to go into production, make sure to remove glGetError calls and any other state getting and checking functions. As an alternative during development, you can look for errors by setting OpenGL Profiler to break on errors.

https://developer.apple.com/library/mac/documentation/graphicsimaging/conceptual/OpenGLProfiler UserGuide/Strategies/Strategies.html

For better usage we have taken the glGetError with OpenGL traces. If it's required to see the error logs then anybody can enable the logs from user settings by going:

Settings -> {} Developer Options -> Enable OpenGL Traces

Suggestion V : Change in Build properties

- ✓ Force GPU usage for all drawing, composition & rendering Graphics are handled in one of two ways, "software", which means the primary CPU does the heavy lifting, and "hardware" which means the GPU does the lifting. Hardware rendering is better because it frees up CPU clock cycles for other stuff, so the phone/Tablet moves faster/smoother (plus GPUs are designed to excel at the types of calculations graphic intensive applications do). This feature supposedly forces programs to use the GPU to paint 2D objects on the screen (2D was previously unsupported, so most apps and the base UI were software accelerated, 3D should be hardware by default).
- ✓ Purge unused assets to free memory
- ✓ Improve dalvik VM execution speed by forcing use of JIT compilation Dalvik operates under a JIT (Just In Time) compilation method which means that when developers make their apps, they partially compile their code into bytecode which is interpreted by the java virtual machine. Dalvik converts bytecode to machinecode as the app runs to increase performance.
- ✓ Improve dalvik VM execution speed by disabling bytecode verification

  When Java source code is compiled, it is converted into bytecode, saved in one or more class files, and executed by the JVM. Java class files may be compiled on one machine and executed on another machine. A properly generated class file is said to be conforming. When the JVM loads a class file, it

has no way of knowing whether the class file is conforming. The class file could have been created by some other process, or an attacker may have tampered with a conforming class file.

The Java bytecode verifier is an internal component of the JVM that is responsible for detecting nonconforming Java bytecode. If bycode verification is skipped, the speed of dalvik VM increases.

- ✓ Disable native code error checking the uncessory error checking in native code is removed for performance improvement.
- ✓ Disable error profiler

Suggestion VI : Added support for pixel format ABGR

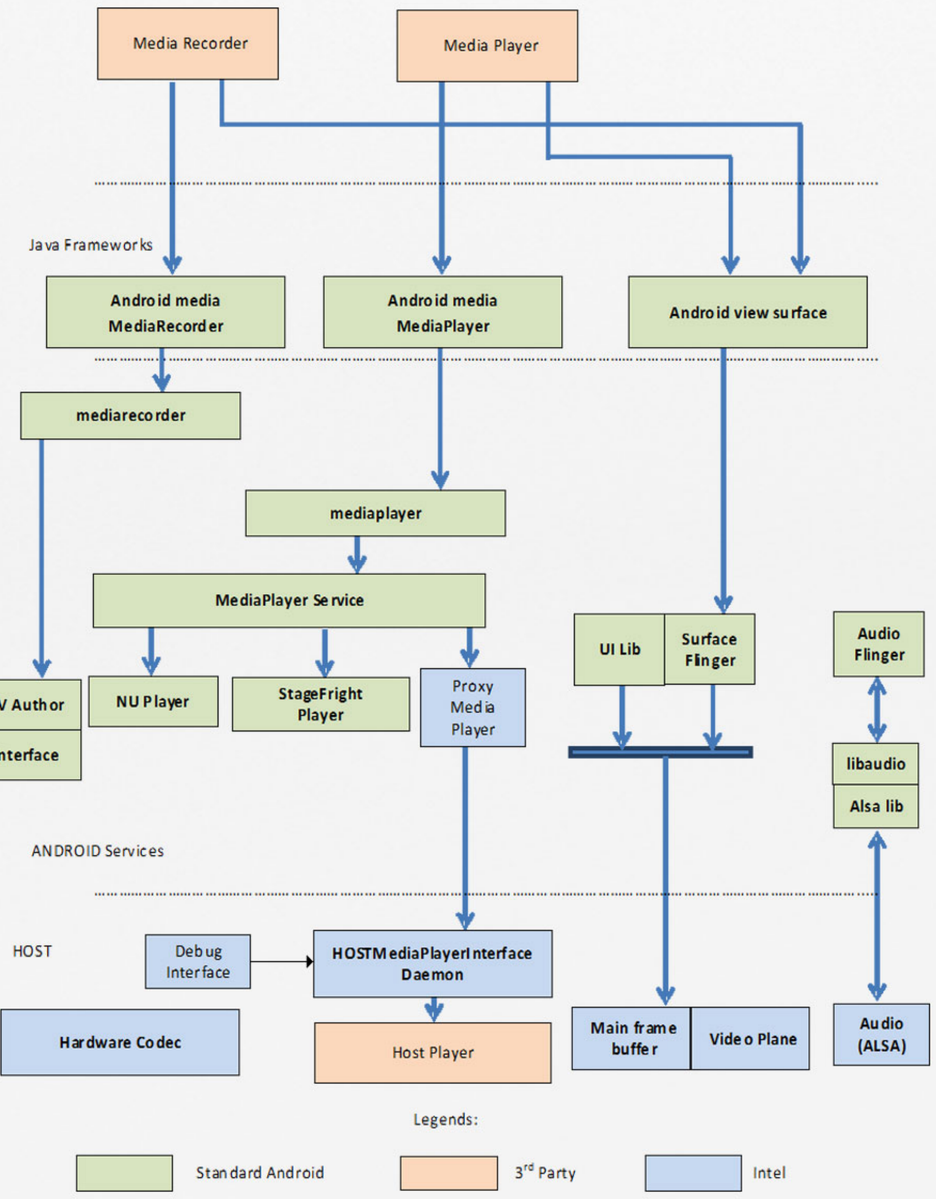
By using pixel format ABGR instead of ARGB, make things easier for graphics media accelerator and it would also allow us to avoid expensive format conversions in some architectures. The result of 2D and 3D graphics is expected to rise by 10 %.

Multimedia Performance Optimizations

Suggestion I : Hardware Composition

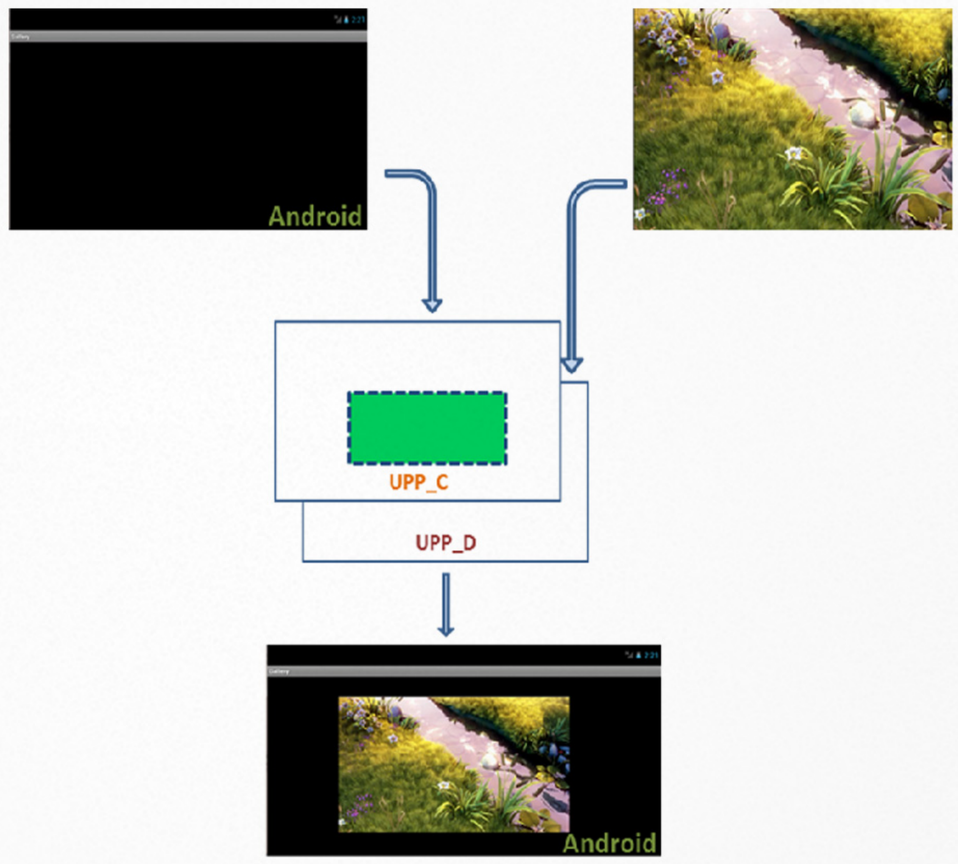
In the Android* media playback architecture, there is a single place through which all requests to play media travel, whether the request leverages the NU PLAYER (for HTTP Live Streaming), or the Android* Stagefright (and its Awesome Player) media player service. The Media Player Service library is responsible for coordinating all media playback requests from all the various sources to the various players that may be registered on the platform.

For all formats such as MP4, 3GP, M4V, MPEG-2 and H.264 in well-known container formats that the Platform® proxy media player supports, the Host Player (or any other third party media player) is called instead of the regular players installed in Android*, from the Media Player Services. In this way all videos are routed to the Proxy Media Player; thereafter, the request is routed from Android* to the host OS using the socket communication



Graphics Overlay Using Surface Flinger in Android

When an Android application requests a video to be played, it is doing so at a particular location on screen, and may even have its own graphics being overlaid on top of the video. We need to ensure that the video is properly positioned and in the correct z-order with the Android graphics even though graphics and video will be on different planes. Figure illustrates our approach:



If Media player will render to a dedicated UPP while the final composited Android UI will use another UPP. To compose the two UPPs and create the final scene, we have two approaches.

- ✓ To leverage the Chroma keying capability by setting a specific color, for example green, to the Surface Flinger allocated memory that maps to the player in Android.
- ✓ The Android* video surface can be made opaque using surface flinger in Android*. This makes the video running in Host visible in Android*.

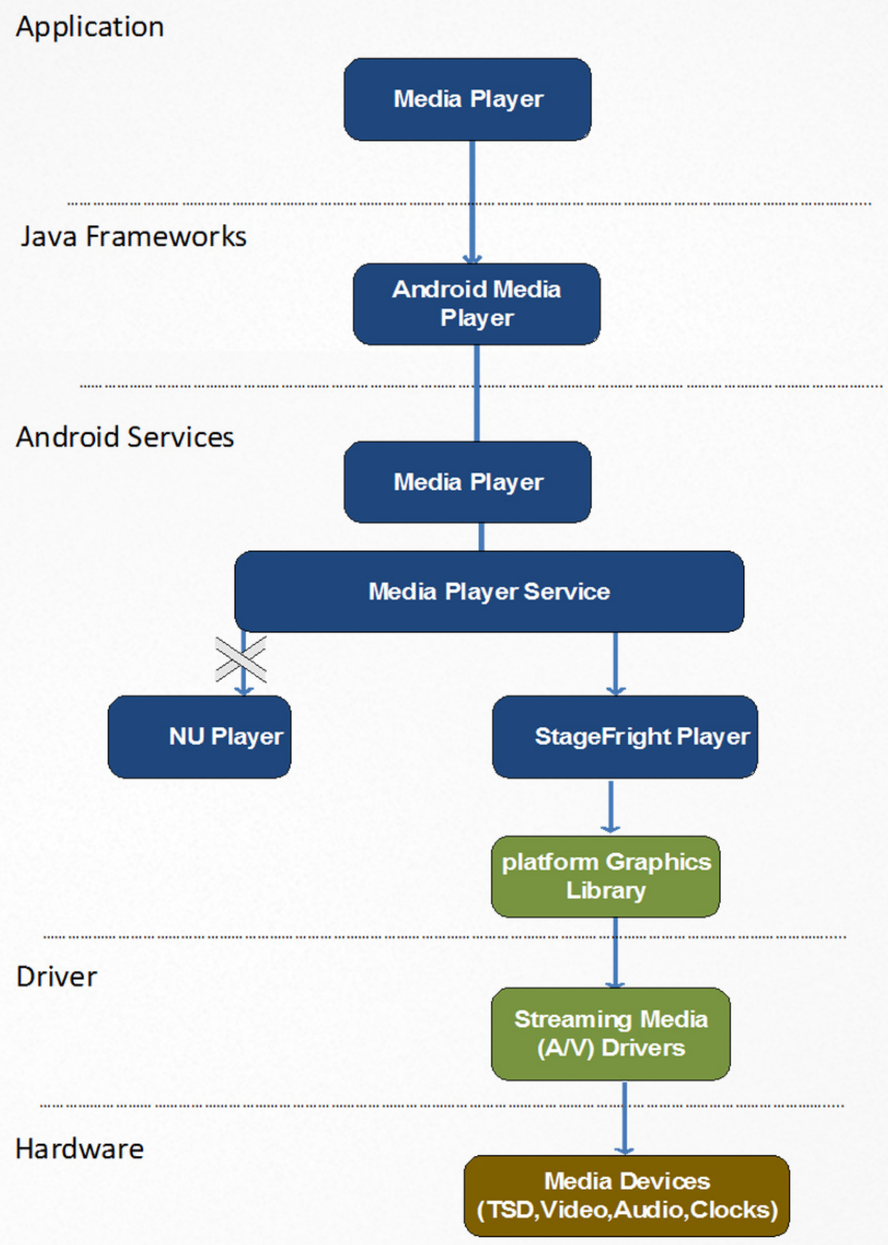
We have implemented second approach. The changes required for the implementation are as follows:

/android-jellybean/frameworks/native/services/surfaceflinger/Layer.cpp

```
// if not everything below us is covered, we plug the holes!
Region holes(clip.subtract(under));
if (!holes.isEmpty()) {
  clearWithOpenGL(holes, 0, 0, 0, 0);      //clearWithOpenGL(holes, 0, 0, 0, 1);
}
return;
```

Suggestion II : Leverage Hardware codecs

In Android StageFrightPlayer use software codec to play video file . but to support hardware codec we can integrate platform specific graphics library with Stage Fright player to improve media playback performance.



I/O Performance Optimizations

- I/O performance optimization with noauto_da_alloc option (Auto-fsync behavior):

  Some applications do not always properly fsync() after renaming an existing file, or truncating and rewriting, ext4 defaults to automatic syncing of files after replace-via-rename and replace-via-truncate operations. This behavior is largely consistent with older ext3 filesystem behavior. However, fsync() operations can be time consuming, so if this automatic behavior can be disabled by using the noauto_da_alloc option with the mount command.
  The change is recommended for I/O performance optimization.

Network Performance Optimizations

By enable Network connection as type Virtio from Host when running Android in virtual environment produces significant improvement in network performance. For this Android kernel configuration is modified as follow:

@@ @@ CONFIG_MII=y
# CONFIG_NETCONSOLE is not set
# CONFIG_NETPOLL is not set
# CONFIG_NET_POLL_CONTROLLER is not set
# CONFIG_TUN is not set
CONFIG_TUN=y
# CONFIG_VETH is not set
# CONFIG_VIRTIO_NET is not set
CONFIG_VIRTIO_NET=y
# CONFIG_ARCNET is not set

Suggestions and Recommendations

Have a workshop to discuss the whitepaper and explore the way forward.

Appendix

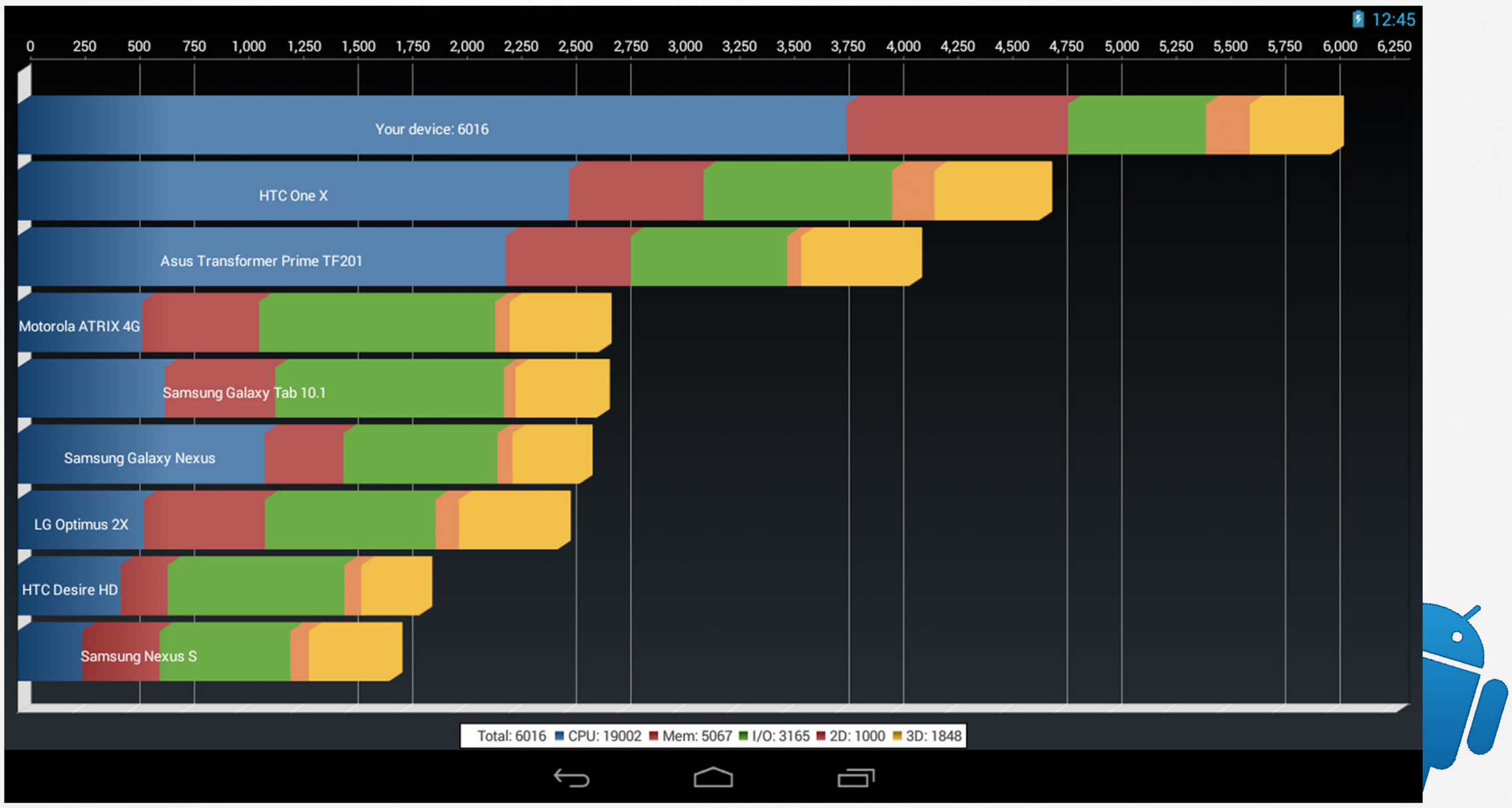
Benchmarking Tools:

Quadrant

Quadrant is a benchmark for mobile devices, capable of measuring CPU, memory, I/O and 3D graphics performance.The initial release for Android smartphones is now available on Android Market. This release targets Android SDK 1.5 and OpenGL ES 1.1.

Performance Improvement Data Quadrant

Quadrant : 6016 (CPU,Memory, I/O , 2D, 3D)



Vellamo

Vellamo is designed to be an accurate, easy to use suite of system-level benchmarks for devices based on Android 2.3 forward. Vellamo began as a mobile web benchmarking tool that today has expanded to include two primary chapters. The HTML5 Chapter evaluates mobile web browser performance and the Metal Chapter measures the CPU subsystem performance of mobile processors.

- See more at: http://www.quicinc.com/vellamo/#sthash.wGNSLhRz.dpuf

Performance Improvement Data Vellamo:

Vellamo HTML5 (Total) : 1232

Tests	Present	Original	Percentage Increase
Sun Canvas	42	28	50
Pixel Blender	50	14	257
Canvas Crossfade	53	53	0
Aquarium Canvas	94	1	9300
Sun Spider	93	74	26
V8 Benchmark	90	78	15
Surf Wax Binder	81	75	8
DOM Node Surfer	71	76	-6
Reflo	73	54	35
Image Scroller	69	54	28
Ocean Scroller	71	1	7000
Ocean Zoomer	79	1	7800
WebGL Jellyfish	81	1	8000
Inline Video	73	1	7200
Load & Reload	212	1	21100

Performance Improvement Data Vellamo:

Vellamo Metal (Total) : 391

Tests	Present	Original	Percentage Increase
Dhrystone2.1	52	41	26.82
Linpack	51	47	8.51
Branch-K	62	60	3.33
Stream 5.9	85	74	14.86
RamJam	89	73	21.91
Storage	52	93	-44

Netperf

Netperf is a benchmark that can be used to measure various aspects of networking performance. Its primary focus is on bulk data transfer and request/response performance using either TCP or UDP and the Berkeley Sockets interface.This tool is maintained and informally supported by the IND Networking Performance Team. It is NOT supported via any of the normal Hewlett-Packard support channels.

Performance Improvement Data

Remote Server Tests	Units	Netperf Remote Server			Netperf Local Host		
		Present	Original	% Increase	Present	Original	% Increase
TCP STREAM Test	Throughput 10^6 bits/sec	275.63	75.74	263.9	709.55	568.95	24.7
TCP REQUEST/RESPONSE Test (512 byte requests,64 byte responses)	transaction rate per second	1263.55	548.54	130.3	2380.53	1778.37	33.9
TCP REQUEST/RESPONSE Test (1 byte requests, 1 byte responses)	transaction rate per second	1304.71	591.16	120.7	2340.63	1818.41	28.7
TCP CONNECT REQUEST/RESPONSE Test	transaction rate per second	534.49	213.49	150.4	1530.71	1112.63	37.6
UDP STREAM Test	Throughput 10^6 bits/sec	908.69	57.73	1474.0	2107.32	2156.13	-2.3
UDP REQUEST/RESPONSE Test (512 byte requests, 64 byte responses)	transaction rate per second	1442.99	578.17	149.6	2635.29	2102.06	25.4
UDP REQUEST/RESPONSE Test (1 byte requests, 1 byte responses)	transaction rate per second	1430.53	617.49	131.7	2699.64	2170.81	24.4

Author Info

Vikas Gupta
Software Architect/
Consumer Electronics Division

Nitin Garg
Technical Specialist/
Consumer Electronics Division

Priyank Dwivedi
Technical Leader /
Consumer Electronics Divison

This whitepaper is published by HCL Engineering and R&D Services.

For more information about HCL Engineering and R&D Services,
Please visit http://www.hcltech.com/engineering-rd-services

Hello, I'm from HCL's Engineering and R&D Services. We enable technology led organizations to go to market with innovative products and solutions. We partner with our customers in building world class products and creating associated solution delivery ecosystems to help bring market leadership. We develop engineering products, solutions and platforms across Aerospace and Defense, Automotive, Consumer Electronics, Software, Online, Industrial Manufacturing, Medical Devices, Networking and Telecom, Office Automation, Semiconductor and Servers & Storage for our customers.

For more details contact: ers.info@hcl.com
Follow us on twitter: http://twitter.com/hclers and
our blog http://www.hcltech.com/blogs/engineering-and-rd-services
Visit our website: http://www.hcltech.com/engineering-services/