

The Need for Speed

Avoiding pitfalls and taking the right turns to make your ARM[®] device fast

By Jonathan Morton, Senior Software Developer, Movial

Achieving your dream user interface and multimedia experience often proves overly difficult due to mismatches between the hardware's capabilities or design, the software toolkit, and the expectations of what can be achieved with them. This article examines some of these difficulties and gives a few tips on how you can avoid pitfalls and make your ARM[®] device as fast as it can be.

It's important to remember that embedded or mobile devices just don't have the same capabilities as PCs do. With their enormously powerful CPUs and graphics processors, linked by high-bandwidth buses, PCs can get away with a lot of brute-force-and-ignorance in the software department. This ability comes at the cost of extremely high-power consumption, which can easily reach into the hundreds of watts. But this is unacceptable in the mobile domain, where a pocket-sized battery is often required to last for days. Hardware manufacturers therefore have to work hard to produce highly efficient processors inside a one-watt power envelope. The ARM Cortex[™]-A9 chips (for example, OMAP[™] 4 and Tegra[®] 2) are about as fast as a good netbook. Older chips – which may be cheaper – have less performance. To ensure a good user experience, the chosen hardware must be used to maximum efficiency, and special features of the hardware must match up to what is required. When this is not the case, your device will be slow.

One obvious pitfall to avoid is the use of high level interpreted scripting languages, such as JavaScript[™], for core functionality, as these languages are incapable of using the CPU efficiently. Java[™] is usually recommendable, since many ARM CPUs have special support for Java, and modern JVMs are often well optimized for it. While often harder to use, a compiled language will in most cases be a lot faster than an interpreted one, all other things being equal.

Most software toolkits, such as Adobe Flash[®], Qt, GStreamer, and X11, offer a very rich array of capabilities to applications. But what they don't offer is any indication of whether your command will be done quickly or smoothly. Most toolkits don't provide any way for you to determine what can and can't be done efficiently, a feature called introspection. If the toolkit doesn't know how to make the



hardware do something efficiently, it's pretty much guaranteed it will do it in an inefficient way – and without telling the application that it's doing so. Usually this means retrieving all necessary image data from the GPU (an extremely slow operation in itself), doing the job using generic routines on the CPU, and then pushing the completed image back to the GPU. Sometimes the software fallback can be run directly on the graphics memory, but since this is an uncached area, this will still be much slower than expected – the CPU cannot use its extensive latency-hiding techniques to optimize loads from uncached memory.

This problem is mostly hidden on desktop hardware. Not only are the drivers for desktop GPUs very well-featured, but the connections between modern CPUs and GPUs are very fast, which allows software fallbacks to run relatively efficiently. These advantages are not available on typical ARM hardware on embedded or mobile devices.

By way of a concrete example, the reason why some simple blitting hardware does not accelerate alpha-blending properly even though it has been successfully used for fills and copies can be that the hardware can only accelerate non-premultiplied alpha-blending, whereas the graphics framework (e.g. XRender

or Qt) requires premultiplied alpha-blending. The workaround may vary from coaxing some more capable part of the hardware into life, to completely replacing the hardware platform with a more capable one.

Sometimes the problem can be that features mentioned in the hardware documentation simply don't work. Another typical problem area is integrating video decode acceleration into a rich graphics framework. Video is one of the most demanding tasks asked of a typical mobile device, with many videos now being at 720p or 1080p resolution (requiring 30-60 megapixels per second) and often requiring rescaling (without blockiness!) to fit the device's screen. In a one-watt power envelope, this level of capability requires dedicated hardware acceleration.

The video decoder's output buffers are usually provided in a variety of formats which are often not directly interpretable by the main graphics APIs, and the buffers (being uncached) cannot be efficiently read by the CPU. After a while, writing yet another convert-YUV-to-RGB routine to run on uncached source memory, and seeing it eat up nearly all of the CPU because of the memory-access inefficiency, can be frustrating. Even so, copying RGB data from those same uncached buffers would be even more taxing, because RGB data is larger than 4:2:0 YUV data.

Another potential pitfall is when the display hardware can read the video buffers, but only for display directly on the framebuffer or on some overlay structure. This is acceptable for many simple applications, but Adobe Flash requires that the video frames are sent through the whole graphics pipeline, so retrieving them from the framebuffer or an overlay is unacceptable. If your idea of "the full desktop/Web experience" includes Flash – or even just YouTube – you will need hardware designed to accommodate Flash, which requires extremely tight integration of the video and graphics accelerators.

The above paragraph certainly goes a long way towards explaining Steve Jobs' attitude towards Flash on the iPhone and iPad.

To set up your project for success, you can avoid the above mentioned pitfalls with the following tips:

Tip 1: Give your designers the real target hardware to test on.

If you want good results, provide your UI designers with the real target hardware to try out. Also representative hardware might do the trick, for example fit-PC Trim-Slice, based on Tegra 2. It won't take up much desk space, and can usually be multiplexed into the existing keyboard, mouse and monitor. It also has the advantage of being usable in a desktop-like way, so it can be used for debugging.

Tip 2: Understand the hardware before specifying what you need it to do.

Do you want simple 2D graphics? Video? Alpha blending (and is it premultiplied or not)? Alpha blending on top of video (which probably requires "textured video")? 3D? Two-way video? High definition? Dual displays? Touch gestures – with or without stylus? ClearType? More than one of these simultaneously? What about sound? Security? Updates? Recovery? Power consumption? Battery recharging? Ethernet? Wi-Fi? Bluetooth? USB (as host and/or slave)? GPS (does it require Assistance)? Mobile data?

What will happen if you need to use a lot of textures, a lot of pixmaps, and very very large pixmap/texture? A very prominent and popular website uses several extremely tall images, over 20,000 pixels on one side to store its customizable skins, so this is a relevant question even if you "only" want a Web browser.

A lot of hardware fails to effectively support at least one of the above, but will seem attractive because it's inexpensive. But trying to add even one of these features after committing to the hardware spec will be really expensive. Beware of false economies and make sure you have the best possible understanding of what is needed in advance of specifying the hardware.

Tip 3: Pick hardware that works right out of the box and can reliably support everything you will ask of it.

Get assurances from the vendor, in writing and tied into penalties for non-compliance, that the features you require will actually work at the performance you need. It doesn't matter if the vendor's own tech demos show something working if the drivers are so unreliable or non-standards-compliant that you can't integrate them into your product.

Get the vendor to set the hardware up with your favorite operating system, with your engineers present (and your subcontractor, if they're doing the work) so that they can later replicate it easily. At this stage, the features on your checklist should all be working (individually is okay). If more than one hardware vendor is involved, get them all in the same room for this purpose.

Do this *before* starting billable engineering effort on software integration. Meanwhile, get your software working on those netbooks.

It shouldn't take two weeks to figure out how to flash an OS image in and boot the device, followed by six man-months to make the 3D engine work. That's what you're paying the hardware vendor for.

Tip 4: Pick middleware carefully.

Most software toolkits, such as Adobe Flash, Qt, GStreamer, and X11, offer a very rich array of capabilities to applications.

But as we discussed earlier most toolkits don't enable *introspection*. They don't even always match up with the hardware's capabilities. There is one prominent graphics API which does not share this problem: OpenGL® ES. The base API is designed specifically around the capabilities of common GPUs, and new GPUs are expected to accelerate these features as a minimum. Extra capabilities are explicitly advertised at runtime via queryable constants and extensions – you can write simple test programs to see them yourself.

GL ES hardware vendors generally don't advertise features which they haven't managed to get running acceptably fast, for one simple reason: it risks games running slowly on their hardware. There is no such built-in restraint for most other APIs.

You can still make GL ES run slowly by simply giving it too much to do, or by using a feature which is not expected to be fast (like reading back the contents of a texture or the framebuffer). But the hardware-centric design does make it far less likely that you will be surprised by it. At the very least, if you use GL ES directly, you get to choose whether you need to read back the framebuffer.

GL ES can be used for 2D UIs as well. The iPhone uses it to provide its famously slick UI. If you choose another API because it supports more features, ask yourself exactly how it implements them, and whether you'll get the performance you require.

Some APIs are designed to run well on top of GL ES, explicitly using its strengths and avoiding its weaknesses. Others run into trouble when they stumble across something that they promise to do but the hardware can't accelerate, and there is no forethought to avoid a major penalty. A select few APIs are actually performance-tested regularly, using real application traces – Cairo is among these.

Tip 5: Insist on usable video acceleration support if you need video.

Many vendors provide some kind of video decode accelerator, which can cope with typical H.264 video at 720p30, and some are now appearing with claims of 1080p30 support. However, ARM CPUs, even the latest multicore NEON-enabled versions, should not be expected to decode high-definition video unassisted. You will need one of the following features to use your decoder:

1) Video-to-GL ES-texture support. This is usually done via OpenMAX® and various EGL extensions, and is essentially required for accelerated Adobe Flash support. Often called “textured video.”

2) Direct scaled output from the video decoder to the framebuffer or a hardware overlay. This is not sufficient for Adobe Flash support, but it's useful for many relatively simple applications, including two-way video calls. Note that you may need to scale small videos up and large videos down, so check that both work properly and look good.

3) Cached (or otherwise fast) CPU access to the video decoder's output buffers. This is the only truly acceptable alternative to explicit video-texture support, as you can copy (or convert) the data into any point in the graphics pipeline. But implementations in this category are rare – the video decoder (along with the rest of the GPU) always seem to hang directly off the main bus rather than the CPU cache and flushing the cache is not made fast enough to make that a viable method of maintaining coherency. Note that standard uncached access is too slow to be useful.

At least include a fast address-range cache flush and expose it via a kernel API or an unprivileged cp15 instruction. The problem to avoid is a full column-address (or even row-address) latency on the memory bus for every single load instruction in a performance-critical graphics routine.

Tip 6: Resist the temptation to add features once the project is underway.

This has been the most basic tenet of Project Management since *The Mythical Man Month* was published decades ago. Yet it still happens today, and adding features always ends up adding months to the project schedule.

Once you've specified your platform for a specific job, expanding that job increases the risk that the platform won't live up to it. You can't “just bolt on” a video player or a Flash plugin. You might not even be able to run the video decoder and the 3D engine at the same time. You might run into VRAM limitations if you add something as “simple” as permitting custom theming of the UI, or a marginally acceptable fillrate might be destroyed if you add a tiny translucent corner or shadow to a window. So think very carefully when considering any change to the spec.

Tip 7: Consult an expert with an excellent track record.

As you probably have noticed, achieving your dream user interface and multimedia experience really isn't a walk in the park. You can seriously shorten time-to-market, reduce the number of defects and achieve a better user experience by consulting an expert company with an excellent track record. For example, if you have any doubt as to whether a particular toolkit or API uses the hardware or the underlying APIs efficiently and effectively (which is often not clear from the marketing claims or the desktop performance), consider partnering with an expert company who can investigate it for you – saving time, effort and money.

By avoiding these pitfalls and taking these tips into consideration even before you start your project your ARM device will be faster and the development process should be a smoother ride.

END