Characterization Report: GPU Assisted Malware detection on Mobile Devices

Jared Carlson

Abstract

Produced as part of the DARPA CFT Proposal: *GPU Assisted Malware Detection*. This document is the deliverable for Milestone I.

Index Terms

Security, ARM, GPU, DARPA, CUDA, NVIDIA, PowerVR, Imagination Technologies, Memory, OpenCL, Architecture, Power Consumption

I. Introduction

THIS document characterizes the landscape for mobile GPU capabilities with an emphasis for general security capabilities, it begins to address the question, ”How might the GPU be used today as well as tomorrow for cyber security tasks”. In particular we’re concerned with how ARM SoC (System on a Chip) or PoP (Package on Package) chips are used today and will become more or less suitable for security tasks.

We note that one of the best reasons for cyber GPU investigation is that a shader can be compiled dynamically. This allows for an adaptability, using a unique set of instructions for cyber tasks on a separate chip. This one of the reasons why the GPU has been used to assist malware, but in this effort we look for defensive capabilities.

Another reason for GPU investigation is the increasingly generality seen in GPU utilization. While the mobile chips allow for programmable graphics pipelines and look to leverage the GPU’s vectorized capabilities, the desktop has an already developed set of capabilities. OpenCL, CUDA are two examples of resources that allow current programmers to utilize their resources for various investigations. For example, OpenCL has been used by mathematicians to investigate prime numbers and similar cryptographic explorations on their personal machines when supercomputing resources aren’t available. These resources and interfaces are being increasingly developed by the user community, for example we’ll see - via both sample code and a brief discussion - that the Python community has created a usable module to allow for quick OpenCL calculations.

A. Terminology

First we note a few terms that we’ll use during this report. We mean *mobile* devices in the sense of a device such as the iPhone, iPad, Android Nexus-S, Transformer Prime Tablet and the like. These devices are not plugged in while in use generally and use a touch screen interface; have a premium on multi-functionality (phone, browser, music player, etc) that can be easily carried by the user. We mean *desktop* in the sense of a device that is plugged into a power source while being used for a significant amount of time (including laptops then), it uses a keyboard or mouse as the tool for an interface and

J. Carlson is an independent consultant in the Boston, MA, area.
E-mail: jared.carlson23@gmail.com
Manuscript developed as part of DARPA CFT project

primarily is not in use while the user is going somewhere (for example a laptop is not usually used to play music while the owner is walking for coffee). We'll use this simplistic categorization throughout the report.

II. iOS

IOS and Apple's mobile devices account for a significant portion - the majority in fact - of Apple's success in recent years, and during this period Apple has been arguably most successful company in the world. Given the revenue Apple generates and its efforts into mobile development this is a highly valued research platform. It has been a consistent leader in both hardware and software development. What is unique about Apple as compared to the other popular vendors today is that Apple both designs hardware and software for its products whereas the major competitor today, namely Android devices, splits hardware and software makers, for example Motorola is a large hardware vendor of the phones while using Google's Android operating system. In this sense Apple is unique in the amount of control it has over its offering.

A. Parters

Apple's iOS operating system for its mobile devices, namely the iPhone, iPod and iPad, uses Imagination's PowerVR SGX5* series GPU. Apple has invested heavily into OpenCL to generalize GPU access and resources, this is installed by default in its XCode developer framework. Apple has used NVIDIA, PowerVR as well as Radeon and Intel chips for graphics capabilities in its devices, but exclusively uses Imagination Technologies PowerVR GPU's for its mobile offerings.

B. Apple's Role in OpenCL

Apple is an editor within the OpenCL working group, which is lead by the Khronos Group. Apple is also a significant shareholder of Imagination Technologies, 8.9%, which creates the GPU used in iOS devices. Apple has submitted various iOS devices to the Khronos Group for OpenCL suitability and testing for OpenCL 1.1 [3]. It appears that while OpenCL underpinnings such as LAPACK are available as part of the SDK, the majority of the OpenCL API is not public at the moment.

III. Android

ANDROID devices - being an open source operating system - has varied hardware and really depends on the manufacturer. Instead it's more useful here to see which graphics chip manufacturers are working with ARM in mind and have readily available hardware solutions. The major manufacturers for Android phones offer the following family of GPUs.

1) Mali GPU – ARM Holdings GPU offering, according to their literature, [4] ARM is in the unique position to provide an optimized compute platform that uses ARM Cortex processors, Mali GPU and ARM CoreLink CCI-400 technologies. This heterogeneous approach means that a range of applications can be processed more efficiently when shared between the CPU and the GPU.
2) NVIDIA Tegra 2/3 – NVIDIA has had mixed reviews with the Tegra 2 chip but the Tegra 3 has improved graphics capabilities and lower power consumption and has been received well by the gaming community to date.
3) Adreno 220/200 – The current offering by Qualcomm [5] which supports programmable graphics, DSP, MDP and streaming textures. The 220 chip is reportedly 5 times more powerful than the 200.

Earlier generations of these devices, such as the Adreno 120, aren't considered in this document as they did not offer a programmable graphics pipeline that could be used for more general tasks. In fact most of these chips will provide OpenGL ES 1.x functionality via a series of preset shaders shipped with the device.

Of these, NVIDIA is the notable player, as they're clearly moving to integrate its chipset into the multicore ARM environment. For example, CARMA, [6], is clearly an effort to allow GPGPU efforts, following their CUDA development environment, and to move this into ARM cores for hobbyists, entrepreneurs and other enthusiasts. This will be available during the second quarter of this year and allows for CUDA development on an ARM platform, running a linux distribution, tied to an NVIDIA chip (and GPU of course). ARM processors are being more and more frequently used for supercomputing, with CARMA's technology being integrated into the Barcelona Supercomputing Center's GPU Supercomputer. Las Alamos National Laboratory has also used ARM based chips to prototype large computing clusters.

IV. Platform Comparison

FOR this document we draw comparisons desktop GPU development and recent mobile advances.

A. Mobile Devices

As we're concerned with using the GPU and, perhaps in later work, other chips for cyber tasks we're primarily concerned with Apple's iOS-based devices and Android, allowing that GPU's for blackberry and the like will be comparable to Android devices - see Table **??** for a few comparisons and shows excellent performance from Apple's iOS products which are only matched in most regards by very recent Android devices. We would also argue that because Android is open that should allow for the most rapid System on a Chip (SoC) development for GPU/CPU system architectures outside of Apple. In fact Android is an ideal resource when once past algorithm development and when we consider how a vendor might want to integrate cyber-based GPU tasks into the operating system.

Test	Apple iPad 2	ASUS Eee Pad Transformer	Apple iPhone 4	Motorola Droid 3
Branching Test: Balanced	20631 kShaders	6199 kShaders	3086 kShaders	6961 kShaders
Common Test: Balanced	5934 kShaders	6023 kShaders	1015 kShaders	5067 kShaders
Fill Test: Warm Up	1002350 Texels/sec	167488 Texels/sec	171959 Texels/sec	238643 Texels/sec

TABLE I: Sample GPU Performance

1) Chipsets: The primary competitors right now are the PowerVX, Adreno, Mali and NVIDIA Tegra graphics processing units. If we classify mobile manufacturers by our operating system, so simply Apple and everyone else (note that a few Android phones, for example Droid X [7], use the PowerVR gpu), we can see that Apple as a hardware and software designer prefers to use Imagination Technologies' PowerVR GPU for its low power consumption (discussed in subsequent section) and perhaps more importantly its ability to integrate the GPU into its tailor-made System on Chip architecture, allowing it to differentiate itself. This is only important for us as Apple is a mobile leader and hence decisions involving OpenCL, GPGPU and similar SDK's are likely going to prove highly influential.

2) OpenCL: Validated OpenCL support is not available in either of the platforms today but this is likely to change rapidly. The belief is that with the next PowerVX GPU, the SGX544, validated OpenCL support will be in place for the next Apple SoC chip. This is a guess based on released prototypes and existing OpenCL underpinnings already present but it does seem increasingly likely given Apple's role in OpenCL, Imagination Technologies, and its profits within the mobile device products offerings.

3) Power Considerations: Power considerations is a focus for mobile devices and here is a strong differentiator in the platforms. In fact, one could argue the point of processors on a mobile device is to deliver performance with a minimum of power. The graphics community, for example, has commented that the Tegra 2 (NVIDIA) has too high a power draw for a number of platforms. The Tegra 3 looks to solve some of these issues [2]. Outside of apple, chip and other developers work to optimize games and other applications for their capabilities to help demonstrate both what can be done and how best to do it.

B. Desktop

In general the Desktop GPGPU development environment is considered NVIDIA's domain. NVIDIA has a large and growing market share for desktop GPGPU development. CUDA is the most broadly used GPGPU SDK, installed for a variety of operating systems (Windows, Mac OS X, Linux all have SDKs) and has been demonstrated for use in graphics, computational, medical and a variety of other environments.

C. Chip Development

Next generation chips are being developed. As was mentioned, CARMA, NVIDIA's ARM chip will be available in Q2 of 2012.

Apple looks to continuously invest in high end SoC chips. In a Barron's report (April 1, 2011) it's noted that the Tegra 2 sells within the \$15-\$20 range while Apple is believed to spend \$25 to Samsung. In other words, any difference in performance in Apple's SoC architecture could be justified by Apple's ability to pay for a better chip. Again, we reiterate the need to look at Apple's investments and consider where Apple is going, and where mobile device hardware will grow.

The following quote, from analyst Didier Scemama, is particularly interesting:

"Such difference in die size is important for ARM and for IMG as die size dictates chip ASP and therefore royalties for both companies. Given the current gap in performance between the A5 from Apple and other competing chips, we believe it is possible higher performance (and therefore potentially larger die size) processors may be required by the OEMs. This may explain the current race among ARM partners to launch higher performance processors in terms of CPU but also, crucially, GPU. It will be interesting to measure the die size of future ARM-based processors against the A5, once they matched it in terms of performance."

V. Minimal Requirements

IN order to demonstrate a feasible choice we have to describe a baseline mobile device and how the device could use the GPU to detect malware, even if somewhat crudely.

First we can rule out devices without OpenGL ES 2.0; and the reason for this is that we need a programmable graphics pipeline, as seen in Figure 1. With a fixed graphics pipeline we unable to specify the various operations we need for vertex and fragment processing. We would also be unable to take advantage of the adaptive environment the shader compilation process provides.

A. Pseudocode

Algorithm 1 Minimalistic CPU Responsibilities

```
1: buffer $\leftarrow$ a region-of-interest
2: $texture[i]$ $\leftarrow$buffer
3: Create frag shader to operate on region of interest
4: Bind $texture[i]$
5: Call Draw Functions to send data to GPU
6: Copy Texels from Framebuffer to obtain processed results
7: Respond to results, e.g.
8: if result $>$ threshold then
9:     Alter shader param or Alter shader program
10: end if
```

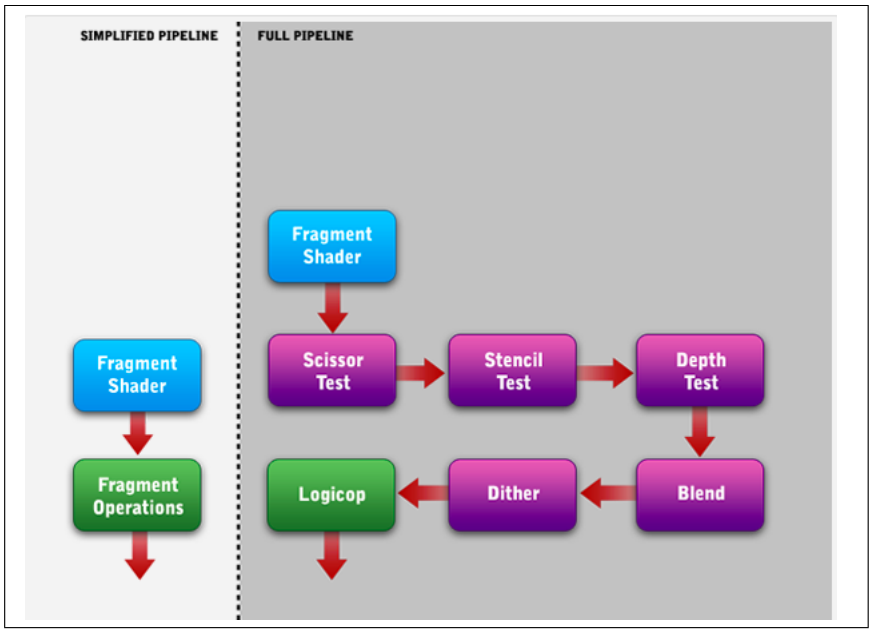


Fig. 1: Comparison of Graphics Pipeline

The shader itself is the focus of the research for Milestone II. Through the shader we can look for sequences (signatures), process the bytes or other relevant information via the same methods a texture could be processed. We remind the reader that this is only a minimalistic requirement as its clear that an increasing number of vectorized libraries will be available on the GPU in the very near timeframe.

Algorithm 2 Minimalistic GPU Responsibilities

```
1: if frag is byte sequence then
2:     $texel \leftarrow marked$
3: end if
```

Alternatively, a more holistic view can be seen in Figure 2, which labels the path along side comments and technologies.

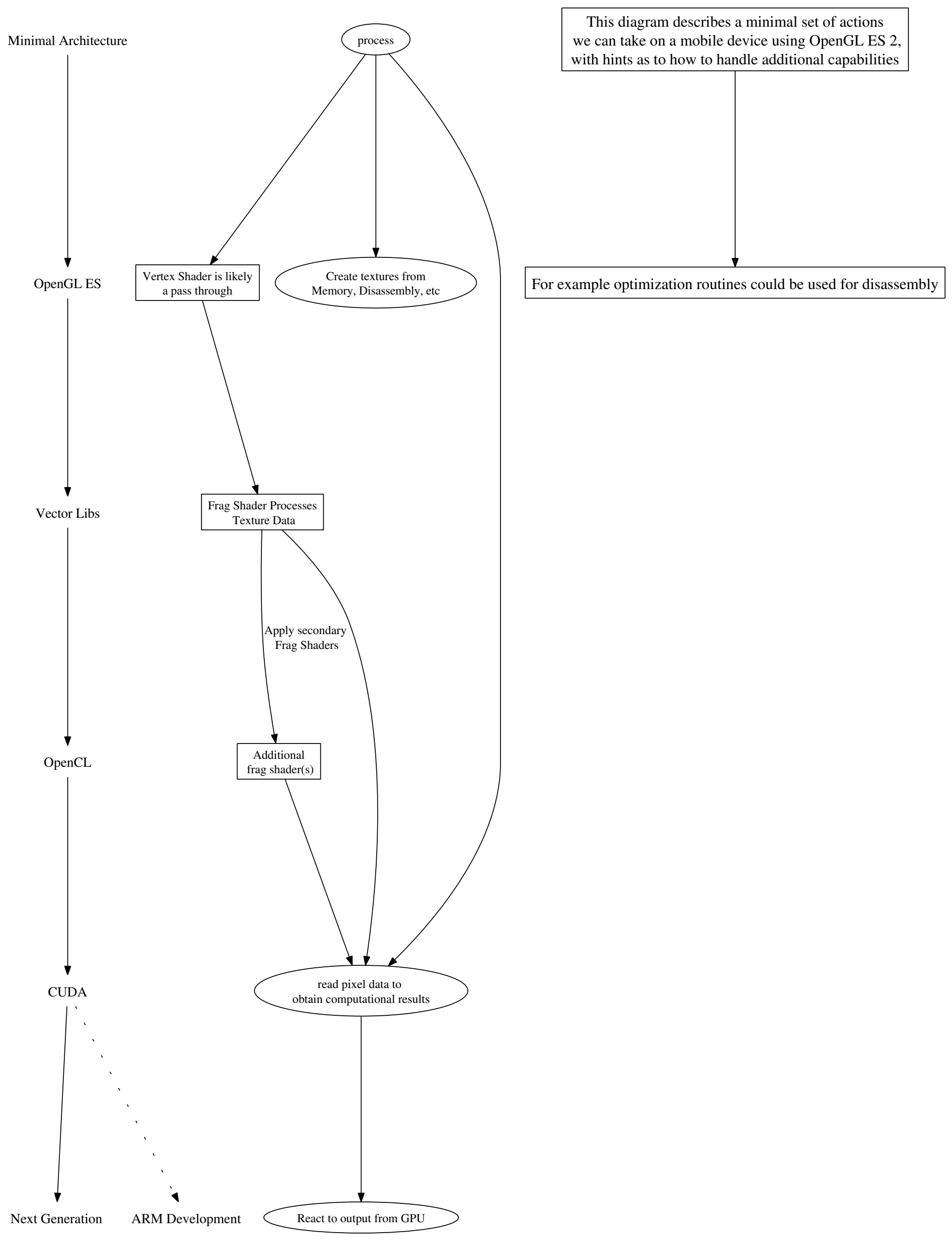

Minimal Architecture
OpenGL ES
Vector Libs
OpenCL
CUDA
Next Generation
ARM Development
process
Vertex Shader is likely a pass through
Create textures from Memory, Disassembly, etc
Frag Shader Processes Texture Data
Apply secondary Frag Shaders
Additional frag shader(s)
read pixel data to obtain computational results
React to output from GPU
This diagram describes a minimal set of actions we can take on a mobile device using OpenGL ES 2, with hints as to how to handle additional capabilities
For example optimization routines could be used for disassembly

Fig. 2: Wholisitic Representation

B. Vectorized Libraries

As discussed in the CFT (Cyber Fast Track) proposal, iOS's Accelerate Framework offers vectorized functionality that will run on the GPU if available. iOS also utilizes a NEON accelerator chip (SIMD) that allows for media and graphics acceleration. In recent Android devices, there is also a trend towards using an accelerator chip.

While controlling tasking for the various chips is not a "simple" task on mobile, we can control which devices we want using the desktop (for example, if we run the pyopencl sample code - Listing 6). This is unlikely to occur at the user level for mobile devices in the short term but it does expose key functionality that could benefit us later. It also means that for true effective cyber tasks at a production level, kernel integration will probably be necessary.

The Accelerate Framework contains the following libraries.

Listing 1: ”Contents of the Accelerate Framework”

```
$ otool -L /Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS5.0.sdk/System/↩
    Library/Frameworks/Accelerate.framework/Accelerate
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS5.0.sdk/System/Library/↩
    Frameworks/Accelerate.framework/Accelerate (architecture armv6):
  /System/Library/Frameworks/Accelerate.framework/Accelerate (compatibility version 1.0.0,↩
      current version 4.0.0)
  /System/Library/Frameworks/Accelerate.framework/Frameworks/vImage.framework/vImage (↩
      compatibility version 1.0.0, current version 185.0.0)
  /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/vecLib (↩
      compatibility version 1.0.0, current version 348.5.0)
  /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libvDSP.↩
      dylib (compatibility version 1.0.0, current version 348.5.0)
  /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 161.1.0)
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS5.0.sdk/System/Library/↩
    Frameworks/Accelerate.framework/Accelerate (architecture armv7):
  [snippet]
```

You can see that the library is basically arm v6 and v7 instructions for the libraries:

- vImage; vectorized Image processing libraries
- vecLib; vectorized numerical routines, such as LAPACK
- libvDSP; vectorized DSP routines

Of particular interest to the author is the *vecLib* library, as this opens up general computational capabilities that we can use. We should note that the author is comfortable with LAPACK and similar libraries from time at the MathWorks, and others will likely have other preferences for investigation.

VI. Minimal Types of Implementations

A. Considerations

While not an explicit goal of this document, implementation is important as we compare capabilities and weight these against our algorithm goals as well as actually implementing a cyber solution on the candidate device.

B. Fragment Shader

As described in Minimal Requirements, a starting point for an algorithm would use a fragment shader to operate on a texture that contains relevant data. The texture contains $n{\cdot}m{\cdot}4$ bytes that we can organize via the CPU, containing a *region-of-interest*. Within the shader a simplistic algorithm can match signatures by looking a for given byte sequence. Heuristic algorithms such as ROP detection will be initially investigated by looking for stack smashing or improper return addresses by capturing the stack states into texture data - a shadow stack [9] or by identifying ROP gadgets [10]. Another heuristic algorithm goal might produce pseudo-disassembly for analysis. This group of heuristic goals center around pre/post analysis - either by their effects or by trying to recognize gadgets. While we’re uncertain at this time, this seems like a noteworthy goal and so should be brought into our device and operating system consideration. These paths are depicted in Figure 3

If necessary and forced to operate within the constraints of the shader then floating point arithmetic to represent a byte stream may be necessary. This is not available on devices earlier than the iPhone 4S and iPad2 and majority of other mobile devices, excluding very recent Android devices. At the moment, and very likely for signature checking we will not need any floating point conversions as we can capture bytes and insert these directly into graphic data for shader calculations but this certainly weighs in favor of recent iOS and Android devices.

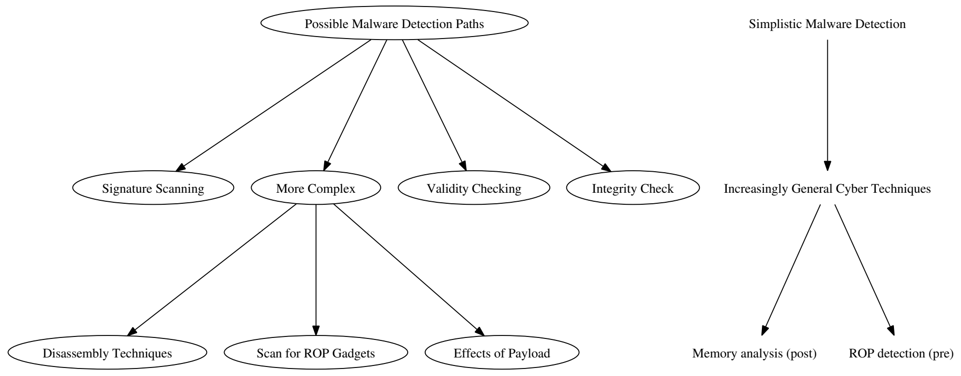


Fig. 3: Possible Malware Handing Opportunities

C. Other Considerations

One item of interest that we need to emphasize is that we do not need to consider the same kind of efficiencies as a typical OpenGL ES developer might. For instance, the normal paradigm is to achieve a constant frame-rate for the application assuming the consumer is trying to have a consistent interface. However, in our case, it's more likely to have a burst of activity to screen for high risk activity, such as during USB pairing or processing an email attachment. In our case we're less concerned with a consistent frame rate and rather how many bytes of data can be processed in a short time period.

Algorithm 3 Heuristic Algorithm Prototypes

```
1: buffer ← a region-of-interest
2: $texture[i]$ ←buffer
3: $texture[j]$ ←pattern-of-interest
4: frag shader looks at minimal surface
5: Call **Draw** functions to send data to GPU
6: Copy Texels from Framebuffer to obtain processed results
7: **if** resultant surface score > threshold **then**
8:     react to possible malware
9: **end if**
```

VII. Expansive Algorithms

EXPANDING on the minimal components we've discussed to this point can be extrapolated by looking at desktop SDK's such as CUDA and OpenCL. The same parallelization and general techniques are the core of these APIs but there are, of course, more access and control points in the desktop environment. However, one significant point to make is that in the development of heuristic algorithms, not only can we prototype using these more generalized APIs but we can also use them to train input or other peripheral sets that can be used by the GPU - for example using optimization algorithms for pseudo-disassembly, often uses probability tables which could be generated on any machine. By leveraging graphics kernels (operations) and data sets made for vectorized algorithms this offers a faster path to migrate general cyber algorithms to the GPU itself.

A. CUDA

CUDA is NVIDIA's offering to generalize GPU capabilities for tasks outside of graphics and rendering. A couple of key concepts that CUDA uses have become common place within the graphics community.

- kernel – An operation to be applied throughout the geometry, in other words to all elements.

- Device and Host Memory interaction. CUDA has various calls to transfer memory from the host (assumed to be a PC) and the GPU.
- Thread Batching – A batch of threads that operates with a limited amount of shared memory.

One of the noteworthy items of CUDA is the utilization of memory. Of particular interest is CUDA's Uniform Virtual Addressing (UVA) which uses the address as a way to detect which device houses a region of memory. This is designed to aid in scaling GPU tasks. CUDA hardware uses a set of SIMD multiprocessors with on-chip shared memory, the SIMD behavior can be grouped by threads into units called warps. Further details for hardware and software can be found at the following CUDA Overview [8].

A sample CUDA program can be seen in Listing 4 of Appendix B.

B. *RenderScript*

Android's migration to Honeycomb meant a leap forward in Renderscript, which now moves more towards a CUDA-like API - see http://android-developers.blogspot.com/2011/02/introducing-renderscript.html. This is another reason to pursue the current state of GPGPU research on the desktop. While a guess, it looks like Android will be an active research and development area for third party graphics and chip makers to investigate embedded and mobile development.

C. *OpenCL*

OpenCL is not solely dedicated to the GPU but more broadly about balancing the GPU and CPU to optimize performance. OpenCL takes advantage of vectorized libraries that will run efficiently on the GPU, a reason why many within the iOS community have concluded that OpenCL will come to iOS based on the libraries within the Accelerate Framework.

OpenCL also uses a terminology similar to CUDA, where a kernel is typically a name given to a parallelized operation. see Listings 5 and 6 within Appendix B for examples.

VIII. Platform Considerations in GPU Algorithm Choices

GPU performance can be measured in several ways. A common place to start is to measure various benchmark results in various GPU operations.

GLBenchmark 2.1 http://www.glbenchmark.com/compare.jsp

A. *Optimizing Shaders*

Shader optimization is typically handled through strict vectorization, enforcing vector operations as opposed to vector-scalar operations. This is important in terms of algorithm development, and throughput the development cycle. As discussed previously the GPU power consumption is non-negligible and so we should make a strong attempt to optimize GPU algorithms. Vector libraries and other tools should be used to prototype development efforts. An example of this (as described in Apple's guidelines, [1]) kind of optimization is seen in Listing 2 and 3.

Listing 2: "Poor use of vector operations"

```
highp float f0, f1;
highp vec4 v0, v1;
v0 = (v1 * f0) * f1;
```

This same calculation can be done more efficiently via

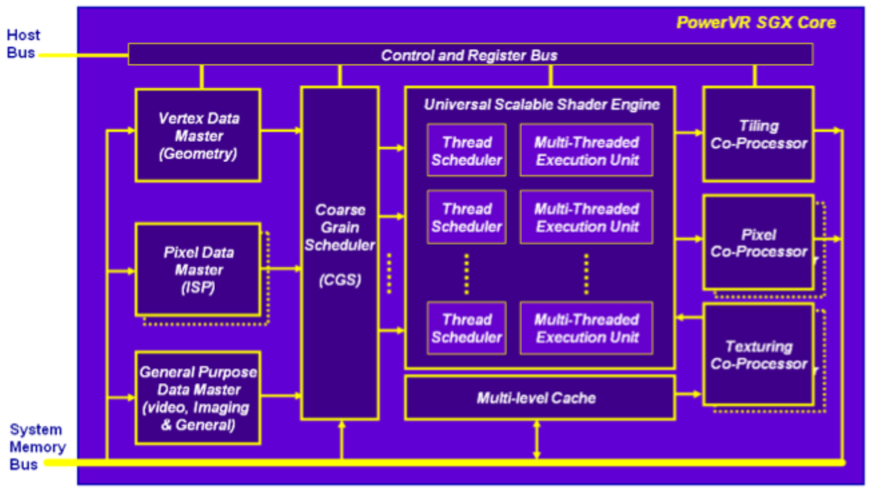


Fig. 4: PowerVR Architecture

Listing 3: ”Proper use of vector operations”

```
highp float f0, f1;
highp vec4 v0, v1;
v0 = v1 * (f0 * f1);
```

B. Handling Memory

Memory is handled differently by the different platforms. iOS has a shared main memory, and NVIDIA’s Unified memory model. For graphics applications, memory bandwidth is seen as a limiting factor. Apple solves this by using a shared memory model. NVIDIA is evolving to a similar structure, that borrows from their unified memory layout.

C. Integration

The iOS kernel allows for a single IOSurface, which is the graphics surface shared between all applications on the device. We note this because it is a way for the kernel to interact via the GPU to all the running applications on the device. However it’s unlikely to use this during our project, although if time permits it would be interesting to research this as a means of communication as well as differentiating gains and losses as compared to rendering to a texture and not presenting the framebuffer to the screen.

Depending on the manufacturer, Android can handle graphics integration slightly different. As an example, and to consider a device, we draw upon the following source, http://androidandme.com/2011/01/news/nvidia-details-the-future-of-high-end-graphics-for-mobile-devices/

The Tegra SoC chip is similar in design to NVIDIA’s desktop product. The reader will notice in Figure 6 that the unified memory model is in place for NVIDIA’s mobile offering as well.

Another point of interest is the ability to reverse engineer or disassemble shader code. CUDA ships with tools to produce ptx code that could be analyzed. This is noteworthy because it would give the CPU a strong tool to dissect the graphics code to be run. As malware tasks can be performed by the GPU this provides a path to an integrated SoC cyber effort.

Development of the Tegra 3 shows gains in memory size, and performance and upgrading from a dual core to a quad core are the high level advances, while video handling, encoding and decoding are also substantially advanced, see figure 7 for more details, along with an accompanying URL.

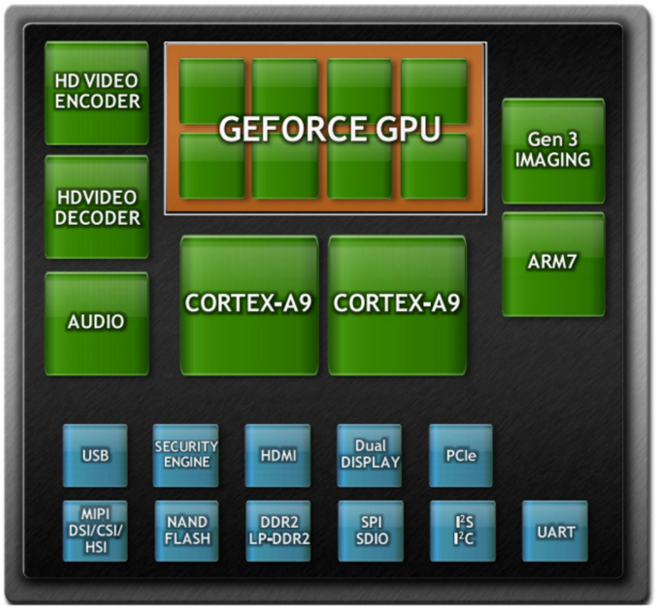


Fig. 5: NVIDIA Tegra SoC Architecture

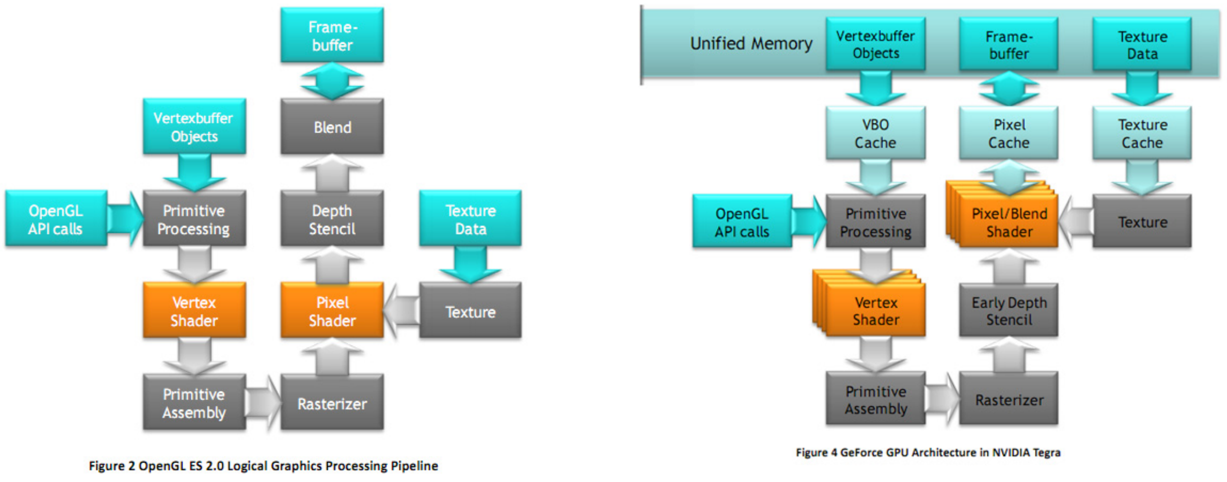


Fig. 6: NVIDIA technical Illustration of ULP GeForce GPU

D. Hardware Considerations

One of the issues developers will look at is the implementation of the graphics driver. For example S3TC (...) is generally considered weaker than PVRTC. A sample justification is PVRTC's lower RMSE, higher signal to noise ratio, than S3TC. Also, S3TC looks to have an upper limit of compression factors around 8, whereas PVRTC looks to have a ceiling around 16, including a an alpha, or compression of 8 at worst. S3TC dates back to DirectX 6-7 and hasn't altered greatly since, where as PVRTC is more broadly supported within the OpenGL community.

IX. Desktop Counterparts

A. When is this appropriate

Mobile development is likely to follow desktop counterparts because of its familiarity. Moreover because of their computational capabilities GPU capabilities are being increasingly looked at for candidates to use vectorized libraries, and an increasing number of existing libraries are being ported to use the GPU.

B. When isn't this appropriate

As we've summarized to this point, desktop solutions do not require the same kind of power considerations that a mobile device does. The mobile sector is rapidly moving to expand on familiar desktop environments, however there are noted differences.

Power considerations look like a hinderance to the NVIDIA offerings until recently, although this is likely to change as substantial efforts were made for the Tegra 3.

X. Performance Effect

A. Performance Control

Optimizations and other shader profiling tools can be done to look at algorithm efficiency. However, device testing is considered mandatory to truly estimate performance. More importantly for our research is the power consumption, which we can estimate based on long-standing tests, i.e., running the chosen shader a given number of times and querying batter state.

B. Additional Devices

There are a number of device options in the market, particularly within the Android device family. The best options for our work seem to clearly be with the recent devices, especially those driven for gaming considerations. While other options such as RIM's Blackberry, or Windows Mobile devices are available these devices either seem to use graphics processors similar to Android, or seem to have a smaller or shrinking marketshare.

XI. Devices Choices

FOR the project, the mobile devices we're going to use are the iPhone 4S (4 to be investigated as well as there exists a number of good jailbreaking tools) and the ASUS Transformer Prime Tablet. This gives us an opportunity to investigate the two most prominent mobile platforms, a phone form factor as well as a tablet. The graphics card differ, as noted earlier and prototyping can be done on a Mac-Mini for iOS, or similar machines using OpenCL/CUDA and similar vectorized approaches.

A. iOS

iOS 5 is the target operating system, although the graphics differences between 4 and 5 are not all that substantial yet. As we've explained we do expect to see upcoming versions of iOS to support OpenCL. Our target device for iOS is the iPhone 4S as we currently have one in our possession as well as its graphics capabilities (performance) and handling of texture data via the PowerVR. It's also likely that validated OpenCL support will shortly be part of iOS.

iPhone 4S, uses an A5 SoC chip with a PowerVR GPU. It represents a powerful hardware set and, with recent additions to the SDK in iOS 5, a strong development position for GPU research.

B. Android

Android is the most popular mobile operating system. The open source nature allows us to rapidly investigate the software integration of graphics capabilities.

The tablet allows us to evaluate a strong offering from NVIDIA, using the Tegra 3 graphics card. The Tegra 3 is better suited to mobile solutions with better power consumption and still has very capable graphics handling. It's upgradable to Ice Cream Sandwich (Android 4) as well as giving us a good but different graphics model for mobile devices (using NVIDIA's chipset). Moreover, for this kind of work, NVIDIA is likely a good research investment given their success in the graphics community in general and especially within programmable graphics engines, with efforts such as CUDA which is mentioned as a model for programmable graphics functionality in Android.

XII. Future Considerations

BASED on the existing work to leverage GPUs for broader capabilities we can conclude that a number of these technologies, such as OpenCL, will be available in the mobile environment shortly.

XIII. Conclusion

THIS document presents a landscape of GPU capabilities, leveraging broader ideas to present supporting materials for this project mobile implementation as well as future considerations.

A. Next Phase

The project will now move into an implementation phase, developing the ideas suggested in the minimal algorithm discussion and using OpenCL on the desktop as a method to prototype vectorized kernels and both integrate with and generate proper data sets.

Acknowledgment

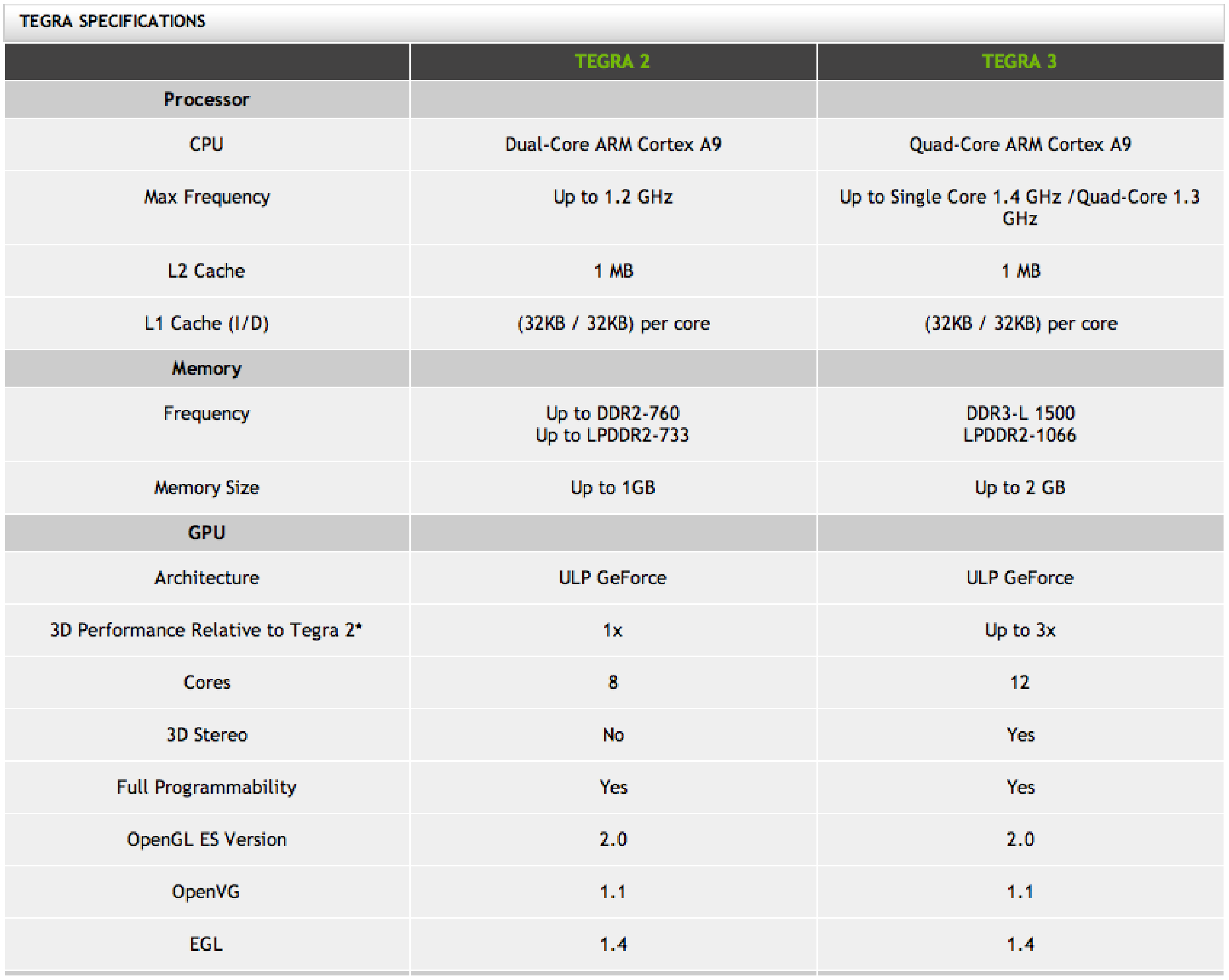
The author is thankful to the following contributors: Alan Stone, Ayal Spitz, Rob Dingwell and Seth Landsman of the *MITRE Corporation*. In addition the author wishes to thank Zachary Carlson of *Lycos*, and George Gal and Dan Rosenberg of *Virtual Security Research* for comments and thoughts regarding this project.

References

[1] Apple, Apple Guidelines for OpenGL ES https://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/BestPracticesforShaders/BestPracticesforShaders.html#//apple_ref/doc/uid/TP40008793-CH7-SW3
[2] Ubergizmo, *NVIDIA Tegra 3* http://www.ubergizmo.com/2011/11/nvidia-tegra-3/
[3] Streamcomputing.eu, *Is OpenCL coming to Apple iOS?* http://www.streamcomputing.eu/wp-content/uploads/kalins-pdf/singles/is-opencl-coming-to-apple-ios.pdf
[4] ARM Holdings, *Mali Graphics Hardware* http://www.arm.com/products/multimedia/mali-graphics-hardware/index.php
[5] Qualcomm Developer Network, *Adreno Graphics Processing Units* https://developer.qualcomm.com/discover/chipsets-and-modems/adreno
[6] NVIDIA, *Meet 'CARMA' - The CUDA on ARM Development Kit* http://blogs.nvidia.com/2011/12/meet-carma-the-cuda-on-arm-development-kit/
[7] anandtech.com *Motorola Droid X: Thoroughly Reviewed* http://www.anandtech.com/show/3826/motorola-droid-x-thoroughly-reviewed/5
[8] Bastien Chopard, CUDA Overview http://cui.unige.ch/~chopard/GPGPU/2-cuda-overview.pdf
[9] Davi, L., Sadeghi, A.R., Winandy, M. Sytem Security Lab - Ruhr University Bochum, Germany *ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks*
[10] Polychronakis, M., Keromytis, A.D., Columbia University *ROP Payload Detection Using Speculative Code Execution*

APPENDIX A
MOBILE RESOURCES AND EXAMPLE CODE

http://www.nvidia.com/object/tegra-superchip.html



TEGRA SPECIFICATIONS

	TEGRA 2	TEGRA 3
Processor		
CPU	Dual-Core ARM Cortex A9	Quad-Core ARM Cortex A9
Max Frequency	Up to 1.2 GHz	Up to Single Core 1.4 GHz /Quad-Core 1.3 GHz
L2 Cache	1 MB	1 MB
L1 Cache (I/D)	(32KB / 32KB) per core	(32KB / 32KB) per core
Memory		
Frequency	Up to DDR2-760 Up to LPDDR2-733	DDR3-L 1500 LPDDR2-1066
Memory Size	Up to 1GB	Up to 2 GB
GPU		
Architecture	ULP GeForce	ULP GeForce
3D Performance Relative to Tegra 2*	1x	Up to 3x
Cores	8	12
3D Stereo	No	Yes
Full Programmability	Yes	Yes
OpenGL ES Version	2.0	2.0
OpenVG	1.1	1.1
EGL	1.4	1.4

Fig. 7: Summary of improvements Tegra 2 to Tegra 3

APPENDIX B
OTHER RESOURCES AND SAMPLE CODE

NVIDIA needs the CUDA Toolkit, GPU SDK, and Driver (3 packages) to be installed. These can be installed from the following site.

Cuda Development Toolkit http://developer.nvidia.com/cuda-toolkit-41

Listing 4: "Cuda Example Program"

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <driver_types.h>
```

```
#include <stdio.h>
/*
  nvcc square.cu -o square

results in:

0 0.000000
1 1.000000
2 4.000000
3 9.000000
4 16.000000
5 25.000000
6 36.000000
7 49.000000
8 64.000000
9 81.000000

*/

__global__ void
square_array(float *a, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if ( idx < N ) a[idx] = a[idx] * a[idx];
}

int main( void ) {
  float *a_h, *a_d;
  int i;
  const int N = 10;
  size_t size = N * sizeof(float);
  a_h = (float*) malloc( size );
  cudaMalloc( (void**) &a_d, size );

  // initialize and send to device
  for (i=0; i<N; i++)
    a_h[i] = (float) i;
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);

  // do the device calculation
  int block_size = 4;
  int n_blocks = N / block_size + ( N % block_size == 0 ? 0 : 1 );
  square_array<<<n_blocks, block_size>>>(a_d,N);

  // get result
  cudaMemcpy(a_h,a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

  // print the results
  for (int i=0; i<N; i++)
    printf("%d %f\n",i,a_h[i]);

  free(a_h);
  cudaFree(a_d);

  return 0;
}
```

OpenCL is part of the developer's environment for Mac OS X; and is part of the XCode download.

Listing 5: "OpenCL Example"

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
```

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <OpenCL/opencl.h>

/*
 * minimal OpenCL program... to build (on Mac OS X (lion)
 * g++ example.cpp -o example -framework OpenCL
 *
 * Execution results in squaring of the input array [0,1,2,3...]
 * Becomes [0,1,4,9,...]
 */

// simple example, just use static data size
#define DATA_SIZE 1024

// simple kernel that computes the square of an input array
const char *KernelSource = "\n" \
  "__kernel void square(                                       \n" \
  "   __global float* input,                                   \n" \
  "   __global float* output,                                  \n" \
  "   const unsigned int count)                                \n" \
  "{                                                           \n" \
  "   int i = get_global_id(0);                                \n" \
  "   if ( i < count )                                         \n" \
  "       output[i] = input[i]*input[i];                       \n" \
  "}                                                           \n" \
  "\n";

// main

int main(int argc, char *argv[] )
{
  int err; // error code

  float data[DATA_SIZE];
  float results[DATA_SIZE];
  unsigned int correct;

  size_t global;
  size_t local;

  cl_device_id device_id;
  cl_context context;
  cl_command_queue queue;
  cl_program program;
  cl_kernel kernel;

  cl_mem input;
  cl_mem output;

  // get data to operate on...
  int i=0;
  unsigned int count = DATA_SIZE;
  for (i=0; i<count; i++)
    data[i] = i;

  // get an id for the device
  int gpu = 1;
  err = clGetDeviceIDs(NULL, gpu ? CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU,
         1, &device_id, NULL );

  if ( err != CL_SUCCESS )
    {
      printf("Unable to obtain device id\n");
      return -1;
    }

  // create device context
  context = clCreateContext(0,1,&device_id,NULL,NULL,&err);
  if ( !context )
    {
      printf("Unable to create device context\n");
      return -2;
    }

  // create queueing context
  queue = clCreateCommandQueue(context, device_id, 0, &err );
  if ( !queue )
    {
      printf("Unable to create CL queue\n");
      return -3;
    }

  // create program
  program = clCreateProgramWithSource(context, 1,
             (const char**)&KernelSource,
             NULL,&err);

  // build the program executable
  err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL );
  if ( err != CL_SUCCESS )
    {
      size_t len;
      char buffer[1024];

      printf("Unable to build CL program\n");

      clGetProgramBuildInfo(program,device_id,CL_PROGRAM_BUILD_LOG,
          sizeof(buffer),buffer,&len);
      printf("%s\n",buffer);

      return -4;
    }


  // create the kernel
  kernel = clCreateKernel( program, "square",&err );
  if (!kernel || err != CL_SUCCESS )
    {
      printf("Unable to create kernel\n");
      return -5;
    }

  // create input and output arrays in device memory..
  input = clCreateBuffer(context, CL_MEM_READ_ONLY,
       sizeof(float) * count,
       NULL, NULL );
  output = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
        sizeof(float) * count,
        NULL, NULL );

  if ( !input || !output )
    {
      printf("Failed to create input or output streams\n");
      return -6;
    }

  // write our inputs into device memory...
  err = clEnqueueWriteBuffer( queue, input, CL_TRUE, 0,
              sizeof(float) * count,
              data, 0, NULL, NULL );
  if ( err != CL_SUCCESS )
    {
      printf("Unable to write inputs to device\n");
      return -7;
    }

  // set arguments
  err = 0;
  err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input );
  err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output );
  err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);
  if ( err != CL_SUCCESS )
    {
      printf("Unable to set kernel arguments\n");
      return -8;
    }

  // get maximum work-group size...
  err = clGetKernelWorkGroupInfo( kernel, device_id,
            CL_KERNEL_WORK_GROUP_SIZE,
            sizeof(size_t), &local, NULL);
  if ( err != CL_SUCCESS )
    {
      printf("Unable to obtain maximum work group size\n");
      return -9;
    }

  // execute kernel over range of data set!
  global = count;
  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
          &global, &local, 0, NULL, NULL);


  // wait fo rthe command queue to get servied before reading
  // back results
  clFinish(queue);


  // read results from the device...
  err = clEnqueueReadBuffer(queue, output, CL_TRUE, 0,
            sizeof(float) *count, results,
            0, NULL, NULL );
  if ( err != CL_SUCCESS )
    {
      printf("Unable to read results back from device\n");
      return -10;
    }

  printf("results[1]       = %f\n",results[1]);
  printf("results[10]      = %f\n",results[10]);;
  printf("results[11]      = %f\n",results[11]);

  // shut down and clean up...
  clReleaseMemObject(input);
  clReleaseMemObject(output);
  clReleaseProgram(program);
  clReleaseKernel(kernel);
  clReleaseCommandQueue(queue);
  clReleaseContext(context);

  return 0;

}
```

Listing 6: "OpenCL via Python"

```python
#!/bin/env python
import pyopencl as cl
import numpy
import numpy.linalg as la

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY,b.nbytes)

prg = cl.Program(ctx, """
    __kernel void sum(__global const float * a,
    __global const float *b, __global float *c)
    {
       int gid = get_global_id(0);
       c[gid] = a[gid] + b[gid];
    }
    """).build()

prg.sum( queue, a.shape, None, a_buf, b_buf, dest_buf )

a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()
print la.norm(a_plus_b - (a+b))
```

Listing 7: "Sample Usage of Accelerate Framework"

```c
float *example( void )
{
    __CLPK_integer n = 3;
    __CLPK_integer info;

    float A[9] = { 2.0f, -3.0f, -2.0f, // first column
                   1.0f, -1.0f, 1.0f,  // second column
                    -1.0f, 2.0f, 2.0f };

    __CLPK_integer ipiv[3];

    sgetrf_(&n, &n, A, &n, ipiv, &info);

    if ( info != 0 ) {
        printf("sgetrf failed with error code %d\n",(int)info);
        return 0;
    }

    float b[3] = {8.0f, -11.0f, -3.0f };

    char transpose = 'N';
    __CLPK_integer nrhs = 1;

    sgetrs_(&transpose, &n, &nrhs, A, &n, ipiv, b, &n, &info);

    if ( info != 0 ) {
        printf("sgetrs failed with error code %d\n",(int)info);
        return 0;
    }

    printf("x = [ %f %f %f ]\n",b[0],b[1],b[2]);
    return b;
}
```