15618 Project Report

Optimizing an Object Tracker on an Android smartphone.

-Ashutosh Tadkase(atadkase) -Shreedutt Hegde(shreeduh)

(Special Credit mention- Peter Abeles, original author of BoofCV software suite, who we worked with during the course of project, and provided extensive support and guidance)

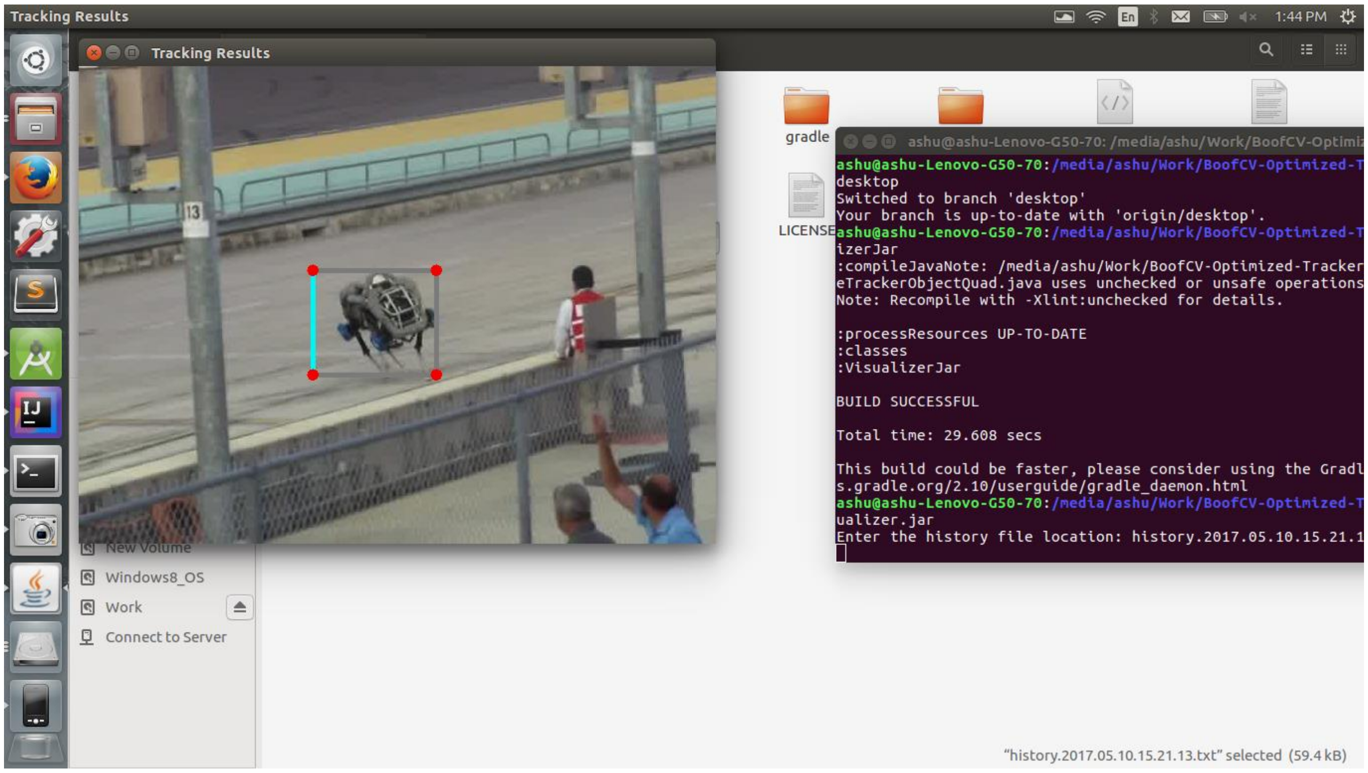
We chose to optimize the circulant object tracker component of the open source CV software suite, BoofCV, written purely in Java, as an optimized single threaded code, and integrated for Android as well. On our initial profiling, we found out the object tracker could process at ~30 fps performance on a typical Android smartphone. The inspiration for parallelism also came from the computationally heavy functions like FFT, Dense Gaussian Kernels, and offline learning performed repetitively over frames to achieve a robust tracking.

Plan was to use the hardware resources offered by modern smartphone SoCs, specifically by ARM-based Qualcomm SoCs, like DSP, GPU, Multiple CPU cores, each with NEON SIMD support, to achieve heterogenous parallelism.

- Planned to use native C/C++ parallel frameworks – OpenCL and Hexagon SDK, to offload computations to GPU and DSP.
- Use RenderScript and Java multithreaded interface for multi-core parallelism.

How we started?

- Extracted the object tracker application from the BoofCV suite, as a standalone desktop Java project.
- Ported this Java code to an Android app , which grabs frames from an image file, initializes tracker position around an object, and keeps tracking it.



```
ashu@ashu-Lenovo-G50-70: /media/ashu/Work/BoofCV-Optimiz
ashu@ashu-Lenovo-G50-70:/media/ashu/Work/BoofCV-Optimized-T
desktop
Switched to branch 'desktop'
Your branch is up-to-date with 'origin/desktop'.
ashu@ashu-Lenovo-G50-70:/media/ashu/Work/BoofCV-Optimized-T
izerJar
:compileJavaNote: /media/ashu/Work/BoofCV-Optimized-Tracker
eTrackerObjectQuad.java uses unchecked or unsafe operations
Note: Recompile with -Xlint:unchecked for details.

:processResources UP-TO-DATE
:classes
:VisualizerJar

BUILD SUCCESSFUL

Total time: 29.608 secs

This build could be faster, please consider using the Gradl
s.gradle.org/2.10/userguide/gradle_daemon.html
ashu@ashu-Lenovo-G50-70:/media/ashu/Work/BoofCV-Optimized-T
ualizer.jar
Enter the history file location: history.2017.05.10.15.21.1
```

The desktop visualizer application that plays the video file with tracker results of tracker application.

Axis of parallelism

- Due to offline learning, inter-frame pipelining or direct parallel processing of tracking pipeline across multiple frames was not possible.
- Even parallel pipelining of computations happening on a single frame was not possible, due to sequential nature of tracking algorithm.
- All heavy computations happen over a small 64x64 tracking window for each frame.
- So try and use the pixel-data parallelism over this small window.

Android platforms chosen

- **LG G5** quad-core CPU with Hexagon 600 DSP, and Adreno 530 GPU.
- **Moto G4** plus octa-core CPU with Hexagon 500 DSP, and Adreno 405 GPU.

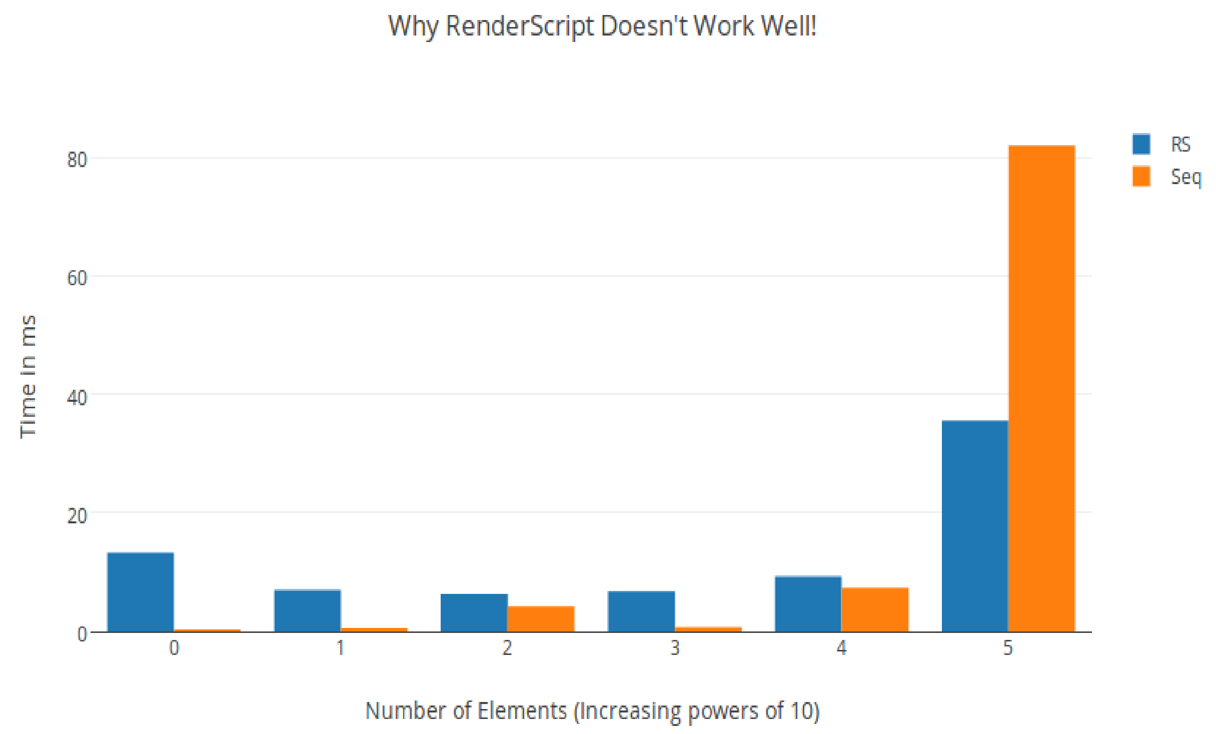
Native frameworks integration with Java-The hardest part.

The major motivation of Android is it's app portability on large variety of hardware devices, abstracting hardware implementations through Java platform.

The hardware specific optimizations require Java integration of native frameworks in C/C++, namely OpenCL for GPU, Hexagon SDK for DSP, and OpenMP/RenderScript for parallelism.

All such Java-to-native code migration, requires the use of JNI interface, which we quantified to conclude as having a very high overhead. The transform of objects from Java to C/C++ code space is particularly expensive and this communication to utilize heterogenous resources, has significantly impacted our design considerations.

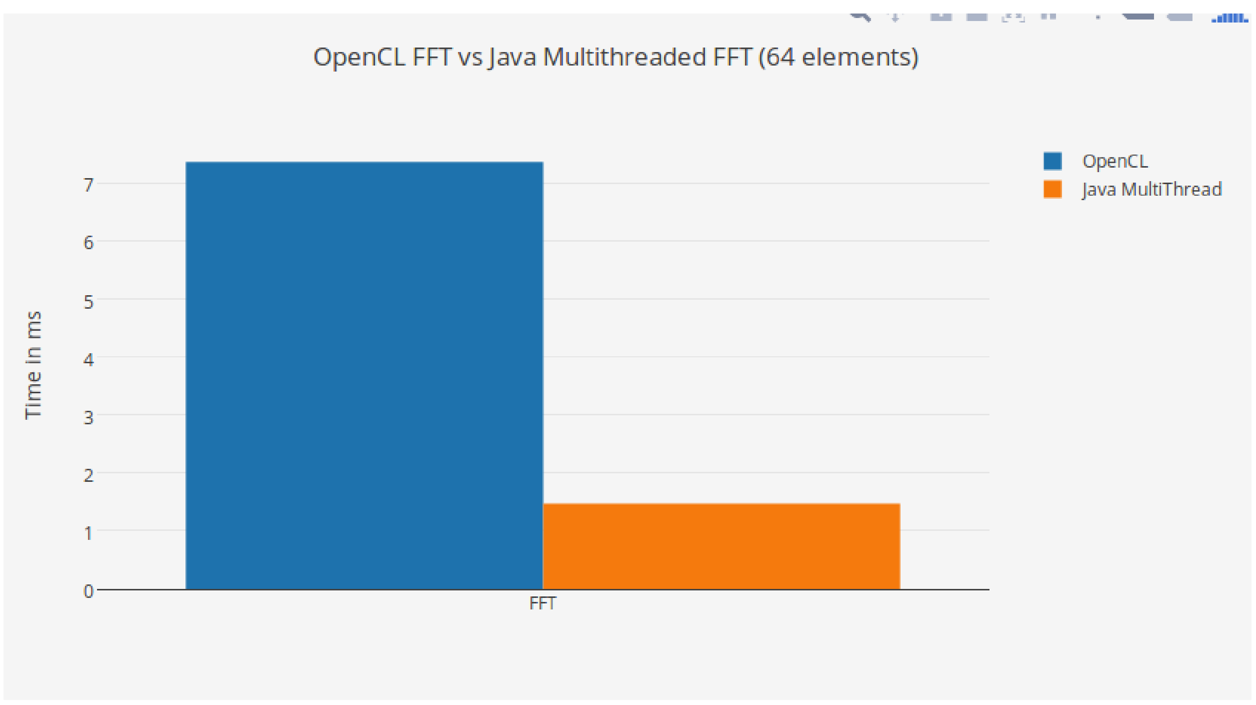
RenderScript



Renderscript is an Android framework, that offloads computations to GPU and CPU under the hood, by using JNI interface.

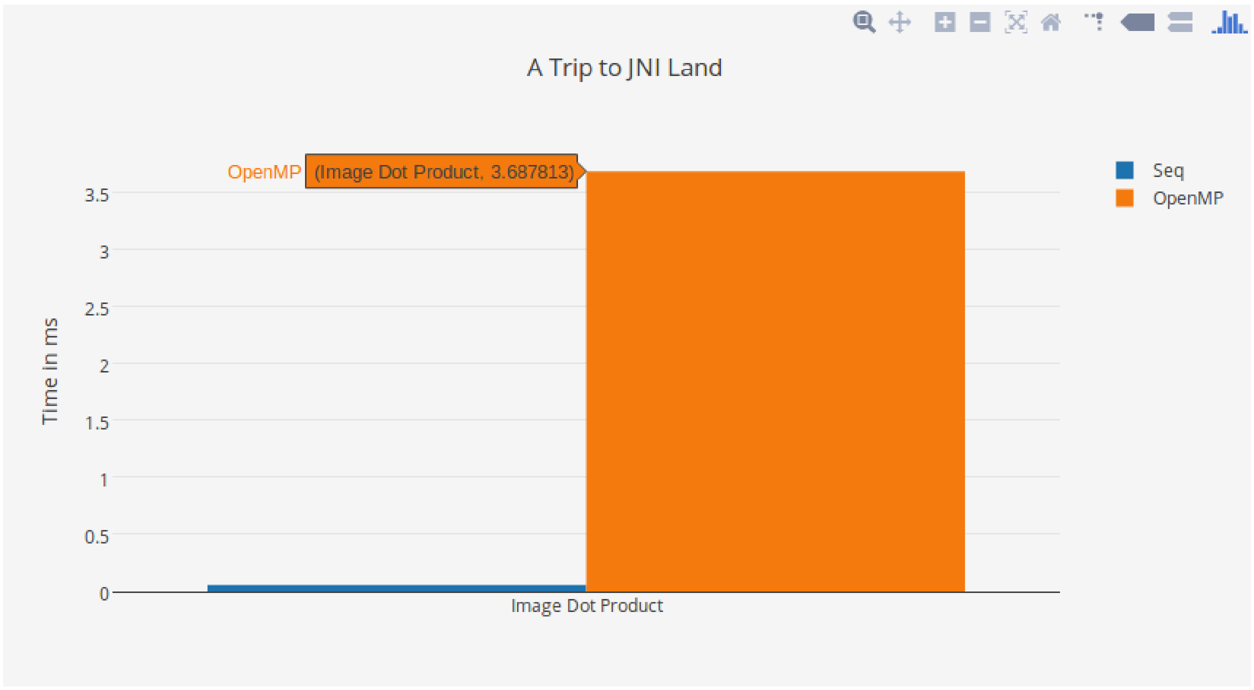
The graph above shows the overhead of a simple array reduction kernel being parallelized using RenderScript, against a sequential version, and plotted against the array size. The use of render script kicks in apparently for sizes beyond 5th power of 10.

OpenCL



The graph above shows the OpenCL implementation of FFT(GPU accelerated) compared to the same FFT algorithm being parallelized using Java multithread library. The FFT shown above is done for 64 elements, being used in tracking window of size 64x64. As clearly evident, OpenCL, being a native framework, that uses JNI for implementation, carries a significant overhead baggage, compared to simple multi-core CPU thread parallelism(8 threads). Since the FFT was being done several times in the object tracker application, we obtained a **speedup of tracker from 30fps to 40 fps**, using multithreaded Java library(Note, this is a purely Java library without any JNI overhead).

So how expensive is this JNI?



The above graph shows the JNI call overhead for an image dot product computation, using the OpenMP framework, as compared to a single threaded Java implementation. The overhead was too much for us to use OpenMP efficiently for accelerating the object tracker computations.

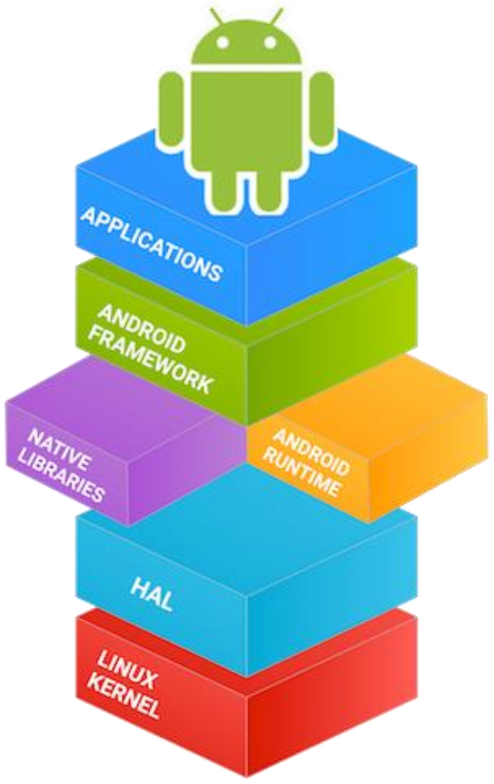
But isn't speedup the real deal?

Though we couldn't achieve a significant speedup of the object tracker(30 fps to 40 fps) due to limitations as discussed before, we have successfully integrated following external frameworks: OpenCL, OpenMP, RenderScript(GPU+CPU) and Neon library(for ARM-SIMD) for Android through JNI interfaces. OpenCV and FFMPEG for frame grabbing and several other CV functions have also been integrated.

Our application, along with the desktop visualizer tool, serves as a benchmarking tool, for any kind of heterogenous workload evaluation for Android devices. A developer using our codebase to start off need not go through the painful integration of these native frameworks with Android, and then start actual parallelization. Native library support has been added for x86 and ARM based Android devices.

Conclusion

The android platform is great because of its portability offered by it’s software stack above the kernel, which abstracts much of device details. However to use device specific acceleration, like GPU or DSP, we need to go native C/C++ from Java, and that’s where the android app would lose it’s portability, but more importantly such offloads would need to be really computationally intensive (may be not the small tracking window of object tracker), to amortize the JNI overheads quantified in this project. The JNI overhead limited us to use pure-Java multithreaded library to extract data parallelism in the object tracker.


APPLICATIONS
ANDROID FRAMEWORK
NATIVE LIBRARIES
ANDROID RUNTIME
HAL
LINUX KERNEL