FFT benchmark on Android devices: Java versus JNI

Antonio D. de Carvalho Jr[1] **, Max Rosan**[1] **, André Bianchi**[1] **, Marcelo Queiroz**[1]

[1]Computer Science Department – University of São Paulo

{dj,maxrosan,ajb,mqz}@ime.usp.br

Abstract. *This work presents a comparison of running times for Java and C/C++ implementations of the FFT algorithm on Android devices. We compare a pure Java implementation with the widely used FFTW library in C/C++, considering also the possibility of multi-threading. 35 different devices were benchmarked, and results on specific combinations of device model and operating system version are presented and discussed. We also discuss similarities between single- and multi-thread versions of FFTW on multicore devices and consider when developers can take advantage of each approach.*

1. Introduction

The Fast Fourier Transform (FFT) is an important algorithm for signal processing applications that can be used in many scenarios, for example, for creating tactile feedback by analyzing audio data (Lim et al., 2013), reducing noise on mobile voice communication (Jonathan and Leahy, 2010), performing face recognition (Cheng and Wang, 2011; Wang et al., 2010), and also for image processing on medical applications (Jonathan and Leahy, 2010). Since the FFT running time is $O(n \log n)$ (where n is the FFT block length in samples) (Cooley and Tukey, 1965), application development on devices with low performance may require fine-tuning some of the FFT internal details in order to ensure realtime operation.

FFTW (http://www.fftw.org/) is one of the fastest and most used FFT libraries (Lin et al., 2011), and its use on Android devices would appear to be a step forward for realtime signal processing compared to pure Java implementations on the Application level. Nowadays, multicore devices are becoming cheaper, thus turning multi-thread into a good approach both for developers and for users. On the other hand, since saving battery in mobile devices is also a primordial concern, using more than one processor just for the FFT algorithm is not an easy task. Another important consideration concerns devices' specific scheduling policies that might decide when to split processing into two or more cores or not. Multi-thread methods can give strange results depending on each device model internal peculiarities.

2. Methodology

This work presents the results of a benchmark of the realtime FFT on blocks of varying sizes using a Java implementation and the FFTW library, which is written in C/C++ and is called through the Java Native Interface (JNI). We have set up an environment to run arbitrary DSP algorithms over an audio stream segmented into blocks of N samples, allowing for the variation of algorithm parameters during execution. The software used is the Android DSP benchmarking application (Bianchi and Queiroz, 2012), an open source project available at https://github.com/andrejb/DspBenchmarking/, with some modifications to include the FFTW via JNI. To compare the performance of different implementations of the FFT algorithm, we considered a pure Java implementation, the single-thread

FFTW and the multi-thread FFTW. As the FFTW implementation is written in C, JNI was used to include the code into the benchmark. All performance measurements are made by the application started by the user. User interactive assistance is kept at a bare minimum, by starting the experiment and pressing a button to e-mail the results to the authors. To obtain as many results as possible, we launched an open call for participation through e-mail, and got responses comprising 35 different devices. Instructions were sent to stop all applications and turn off communication (tests could only be started after the user enabled flight mode) to impose an ”idle” scenario on every device. The result of imposing these constraints is an overall experiment that automatically cycles through all benchmarking algorithms, and then sends an e-mail report with results back to the authors.

3. Results and discussion

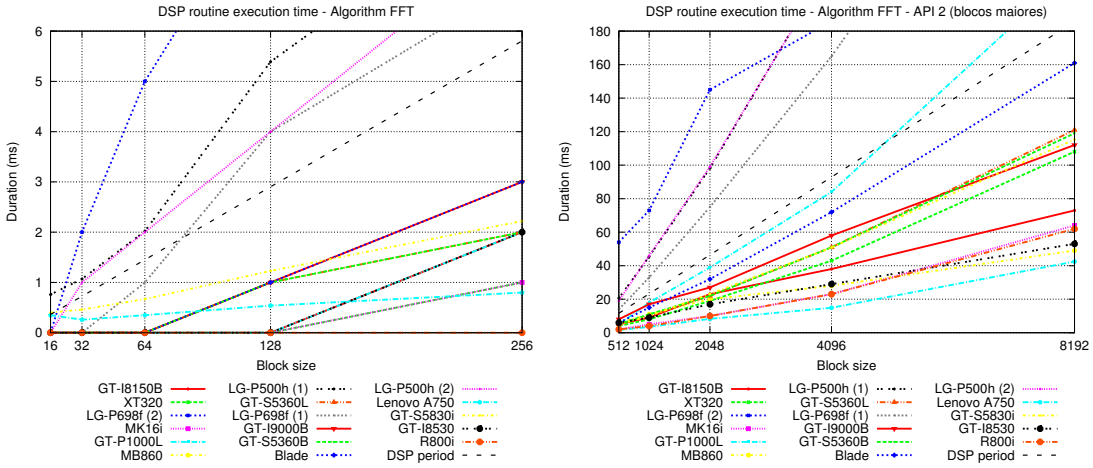

Figure 1: Benchmark of FFT implemented in pure JAVA in devices with API version 2.X for smaller (above) and larger (below) block sizes.

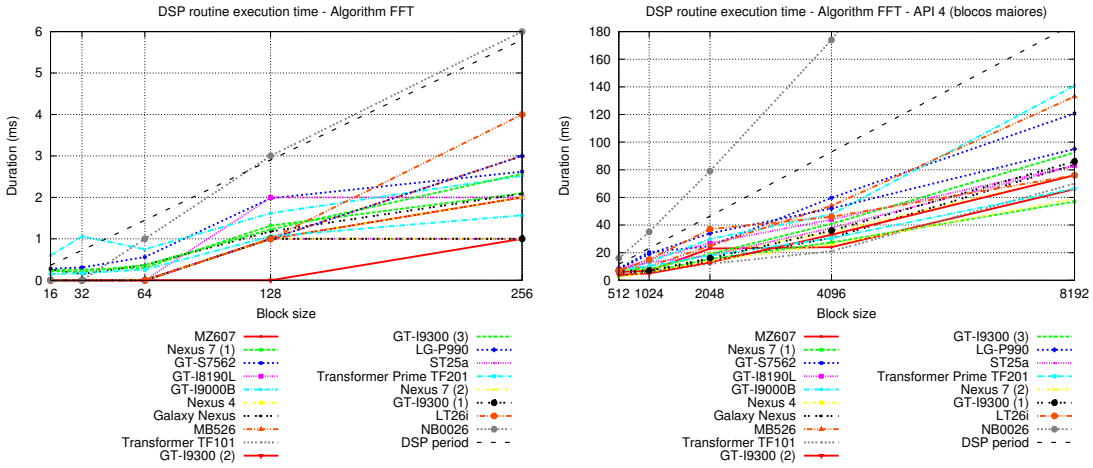

Figure 2: Benchmark of FFT implemented in pure JAVA in devices with API version 4.X for smaller (above) and larger (below) block sizes.

Figures 1 through 6 show the time taken by each of the FFT algorithms to be executed on each device as a function of block size. By comparing the Java FFT and single-thread FFTW results, on Figures 1, 2, 3, and 4 respectively, it is possible to notice that single-thread FFTW is faster than Java FFT on blocks larger than 16 samples. For N=16 the overhead of loading a dynamic library is not compensated by the efficiency of

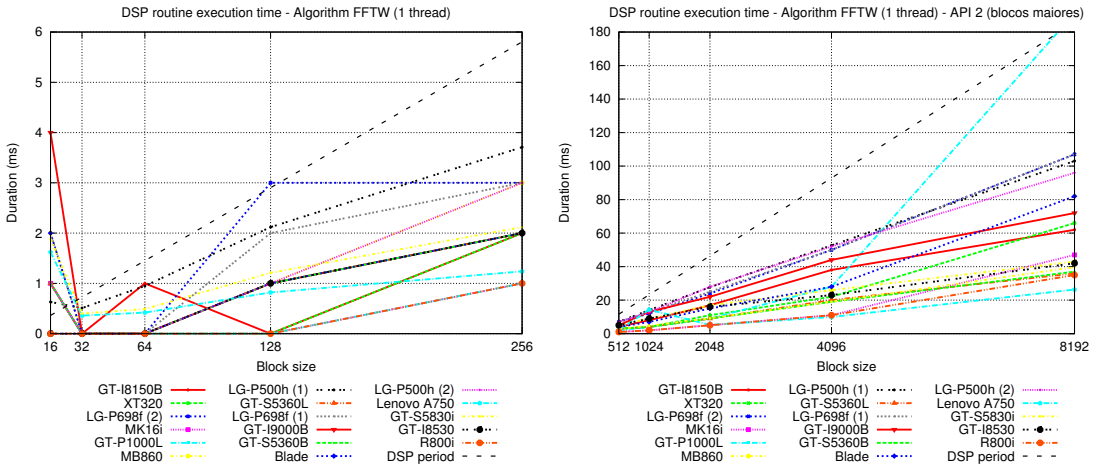

Figure 3: Benchmark of FFT using single-thread FFTW in native code in devices with API version 2.X smaller (above) and larger (below) block sizes.

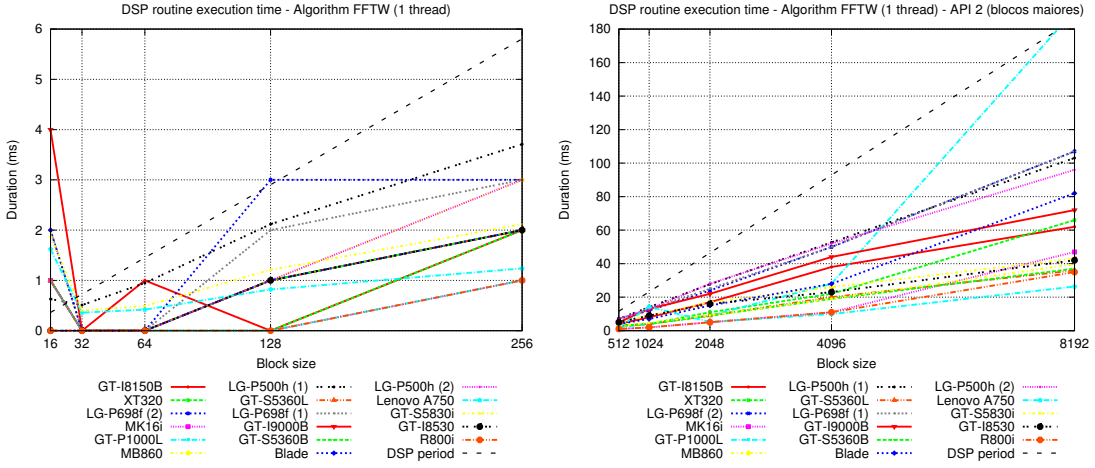

Figure 4: Benchmark of FFT using single-thread FFTW in native code in devices with API version 4.X smaller (above) and larger (below) block sizes.

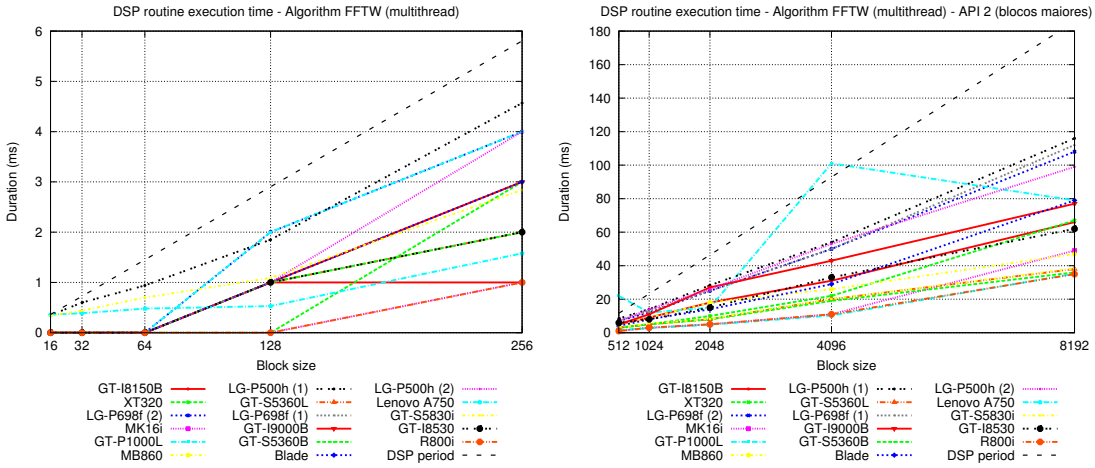

Figure 5: Benchmark of FFT using multi-thread FFTW in native code in devices with API version 2.X for smaller (above) larger (below) block sizes.

the C code, so that the results for the single-thread FFTW are actually worse than for the Java FFT, but for larger block sizes the library is already loaded. The same is observed with the multi-thread FFTW: it is executed after the single-thread FFTW and therefore

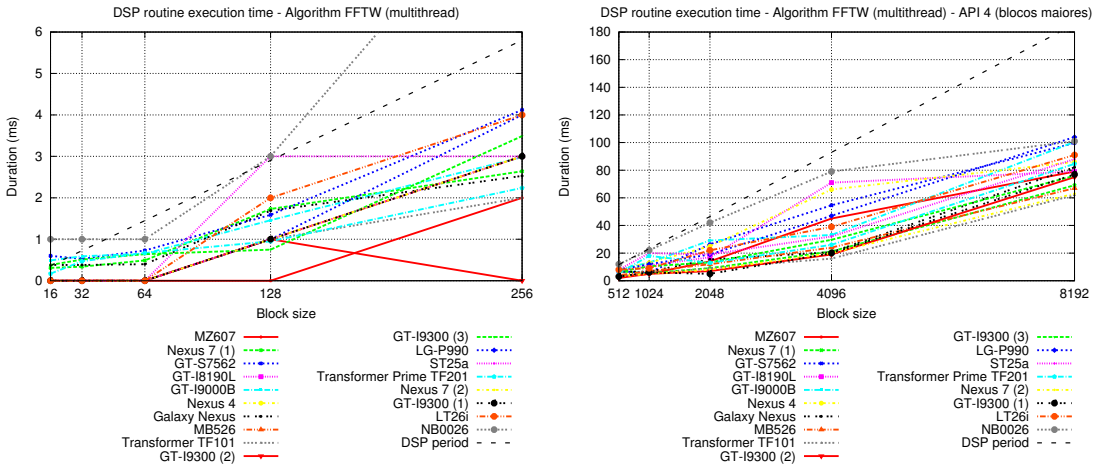

Figure 6: Benchmark of FFT using multi-thread FFTW in native code in devices with API version 4.X for smaller (above) larger (below) block sizes.

there is no overhead for the initialization of the library.

It is also possible to notice on Figures 5 and 6 that FFTW has worse performance when it is used with multiple threads. As it turned out, the Android kernel has different policies in different devices, and some devices move threads to different cores only when they are CPU-intensive and have been running for a sustained period. The use of native code does not automatically imply better performance on every situation. It had been noticed that using JNI increases application complexity and also has a cost associated with calls to non-Java code, which makes it indeed worse for some applications (Lin et al., 2011). Nevertheless, the implementation and comparison with native code for realtime signal processing evaluated in our work show that there are more subtleties that should be kept in mind, like the amount of data processed and the device specification about starting cores and moving threads around.

References

Bianchi, A. J. and Queiroz, M. (2012). On the performance of real-time dsp on android devices. *Proceedings of the 9th Sound and Music Computing Conference*, pages 113–120.

Cheng, K.-T. and Wang, Y.-C. (2011). Using mobile gpu for general-purpose computing: a case study of face recognition on smartphones. In *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pages 1–4.

Cooley, J. and Tukey, J. (1965). An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301.

Jonathan, E. and Leahy, M. (2010). Investigating a smartphone imaging unit for photoplethysmography. *Physiological Measurement*, 31(11):N79.

Lim, J.-M., Lee, J.-U., Kyung, K.-U., and Ryou, J.-C. (2013). An audio-haptic feedbacks for enhancing user experience in mobile devices. In *Consumer Electronics (ICCE), 2013 IEEE International Conference on*, pages 49–50.

Lin, C.-M., Lin, J.-H., Dow, C.-R., and Wen, C.-M. (2011). Benchmark dalvik and native code for android system. In *Innovations in Bio-inspired Computing and Applications (IBICA), 2011 Second International Conference on*, pages 320 –323.

Wang, Y.-C., Pang, S., and Cheng, K.-T. (2010). A gpu-accelerated face annotation system for smartphones. In *Proceedings of the international conference on Multimedia*, MM '10, pages 1667–1668, New York, NY, USA. ACM.