Last updated May 5, 2020

Supplementary Materials - *F5N* : Nanopore DNA Data Analysis Toolkit for Android Smartphones

List of Figures

Contents

I. Adjusting memory governing parameters of the tools

The following steps can reduce peak RAM usage, to run *F5N* on a broad spectrum of mobile devices. For more details visit *Minimap2* man page[1], *Samtools* man page[2] and *F5C* man page[3].

1) Increase the number of partitions the genome reference is split into, to reduce peak RAM usage in *Minimap2 alignment*
2) In *Minimap2 alignment* reduce the parameter value, *number of bases loaded into memory to process in a mini-batch [-K]*
3) In *Samtools sort* reduce the parameter value, *the maximum required memory per thread [-m]*
4) In *F5C call methylation* reduce the parameter value, *batch size (max number of reads loaded at once) [-K]*
5) In *F5C call methylation* reduce the parameter value, *max number of bases loaded at once [-B]*
6) In *F5C call methylation* skip ultra long reads by setting the option *[–skip-ultra FILE]*
7) In *F5C call methylation* reduce the threshold value to skip ultra long reads *[–ultra-thresh INT]*

II. Reconstructing sequence analysis tools for ARM/Android

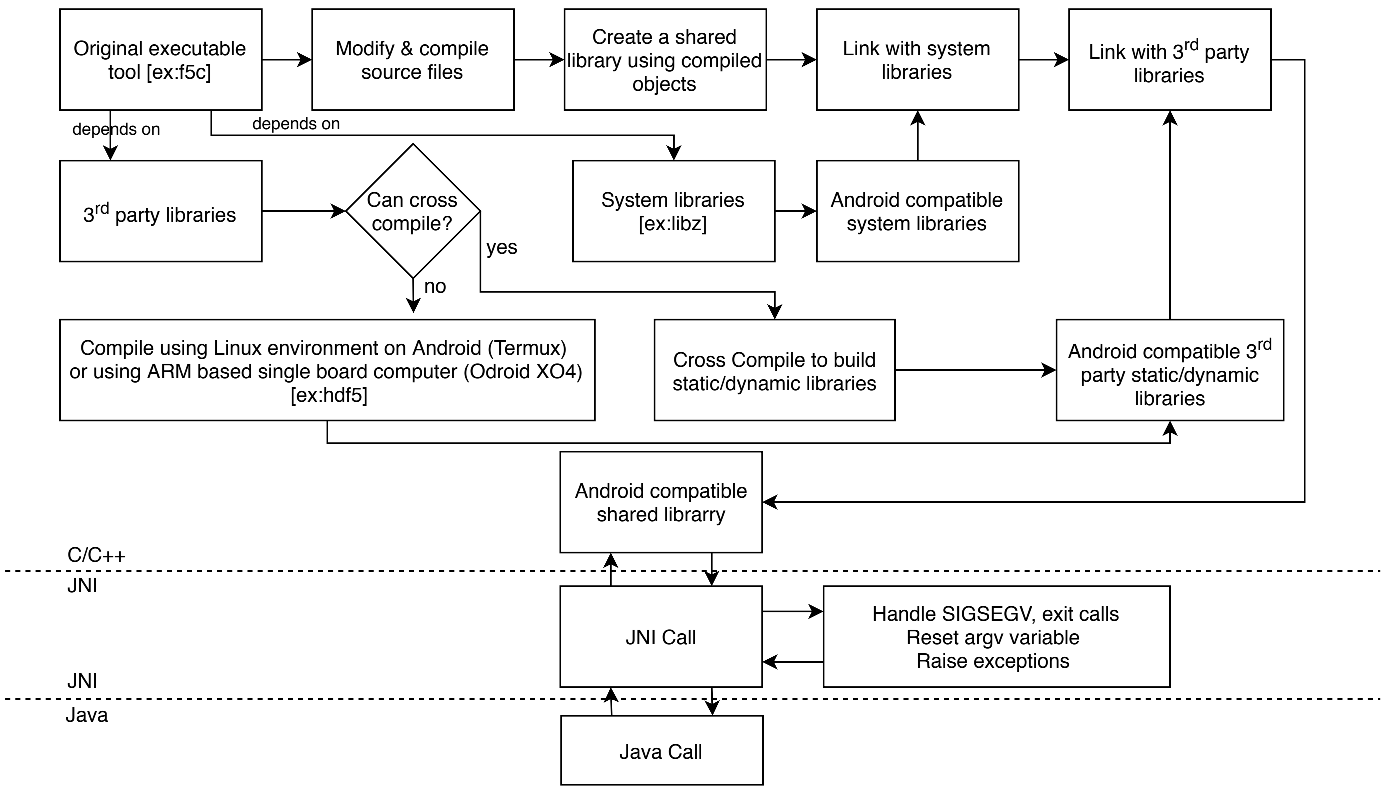
Original executable tool [ex:f5c]
Modify & compile source files
Create a shared library using compiled objects
Link with system libraries
Link with 3rd party libraries
depends on
depends on
3rd party libraries
Can cross compile?
yes
no
System libraries [ex:libz]
Android compatible system libraries
Compile using Linux environment on Android (Termux) or using ARM based single board computer (Odroid XO4) [ex:hdf5]
Cross Compile to build static/dynamic libraries
Android compatible 3rd party static/dynamic libraries
Android compatible shared librarry
C/C++
JNI
JNI Call
Handle SIGSEGV, exit calls
Reset argv variable
Raise exceptions
JNI
Java
Java Call

Fig. S1. Cross-compiling work flow and JNI interface design involved in including a new sequence analyis tool to *F5N*.

A. Introduction and prerequisites

This section provides fundamental guidelines on how to compile an analysis tool written in C/C++ for Android. Except for a few essential amendments (discussed below), our approach does not alter the original C/C++ code, which permits straightforward integration of new C/C++ analysis tools into *F5N*.

Having some prior experience with Android SDK (Software Development Kit), Android Studio[4], NDK (Native Development Kit) [5], Java Native Interface (JNI)[6], ADB (Android Debug Bridge)[7], CMake build manager[8] and Ninja build system[9] certainly helps someone who is interested in rebuilding *F5N* or extending it with new tools.

B. Native compiling an analysis tool

In our case, the fact that all the necessary tools (*Minimap2, Samtools* and *F5C*) were written in C/C++ let us to follow the same approach to compile them for Android. The approach can be summarized as shown in Fig. S1.

Before starting the compilation we tested the tools using *Termux* (a terminal emulator with Linux environment for Android)[10]. In *Termux*, the tools' git repositories were cloned and built as instructed on the respective installation guides. This method can be known as *native compiling*. If a tool gets *natively compiled* in *Termux*, we can reasonably expect the tool to get cross-compiled for Android. There are two major *Instruction Set Architectures (ISA)* for *ARM* mobile devices called *armeabi-v7a and arm64-v8a*. *Termux* was installed on multiple devices with different architectures to make sure that the tools can be *natively compiled* in both the architectures. It is noteworthy that in *Termux*, the executable versions (*.exe* format) of the tools were built in contrast to their dynamic library versions (*.so* format). To build an Android application, the dynamic version of an original tool is required[11]. All the Android applications are a subset of Java programs and the gateway to the native (C/C++) code is obtained by loading the dynamic version of the native tool. To obtain a dynamic version of the tool (*.so* format), we can change the build configuration and natively compile in *Termux*. However, this method is not encouraged as it can impose device based restrictions on *.so* files which can result in intensive and costly debugging. The recommended method to obtain the dynamic version of a tool is to cross-compile using the Android Tool-chain[12]. This eliminates the burden associated with *native compiling* and simplifies the process of updating the tools to their latest versions. This in return automates continuous integration and delivery. In Suppliementary Sections II-E and III we provide further details on how to cross-compile.

C. Determining third party libraries used by an analysis tool

It is necessary to determine the third party libraries used by the analysis tools (*Minimap2, Samtools* and *F5C*). A tool to function correctly, associated third party libraries should also be linked statically or dynamically with the tool, i.e., third party libraries should also be cross-compiled. *Samtools* depends on *htslib* [13] library. F5C depends on both *htslib* and *HDF5* [14] libraries. Hdf5 library is used to handle fast5 files and there exists no straightforward method to cross compile hdf5. Hence we used native compiled instances of *HDF5* library (the instances that were built using Termux). We maintained two instances of *HDF5*, one for each Architecture (*armeabi-v7a* and *arm64-v8a*). These third party libraries were statically linked to respective shared libraries. If a third party library is being used by two dependent tools, make sure to link a dynamic version of the library. For an example in our case, *htslib* was used by *Samtools* and *F5C*. Trying to statically link *htslib* separately to each tool, caused the software to crash on some devices. This was resolved by linking *htslib* dynamically.

D. Designing JNI interface

JNI acts as the bridge between native (C/C++) methods and Java function calls. In JNI interface, each tool's *int main(int argc, char* argv[])* function was called. The function name, *int main(int argc, char* argv[])* was renamed as *int init_X(int argc, char* argv[])* where X was *Minimap2,Samtools* or *F5C*. This function renaming is necessary as the tools are not stand-alone executable applications but dynamic libraries. Moreover, a header file was introduced for each tool that contained the function signature *int init_X(int argc, char* argv[])*. JNI interface was extended to facilitate the following,

1) Handle *exit* signals returned by native code.
2) Handle *SIGSEGV* signals returned by native code.
3) Raise exceptions on behalf of the native functions.
4) Reset *argv* variable before calling another native function [*int init_X(int argc, char* argv[])*].
5) A tunnel between the native code and the Java program to communicate *standard error* messages.
6) If original code does not define an output file path argument, redirect *standard ouput* to a file.

It is important to handle different types of signals and errors thrown by the native code to prevent JVM from crashing. For example the original code most probably will exit with an error if something goes wrong. This kills the JVM if not handled. We want to keep the JVM running throughout a *F5N* session. In order to do that, the exit call should be caught and handled in JNI [15]. In a similar manner, *SIGSEGV* signals should be handled safely [16]. Once an exit call or *SIGSEGV* signal is handled this should be informed to Java program so that the user can investigate the problem. To do this, exceptions are thrown from JNI to Java [17].

The original C/C++ tools take input as command line arguments. When a pipeline with more than one step is executed, the native code attempts to read the same argument vector multiple times. If the argument vector is not reset after the completion of the first step, arguments do not get parsed in the second step as desired. This resetting part is not implemented in most of the original libraries. In JNI this should be implemented to avoid failures associated with arguments parsing [18].

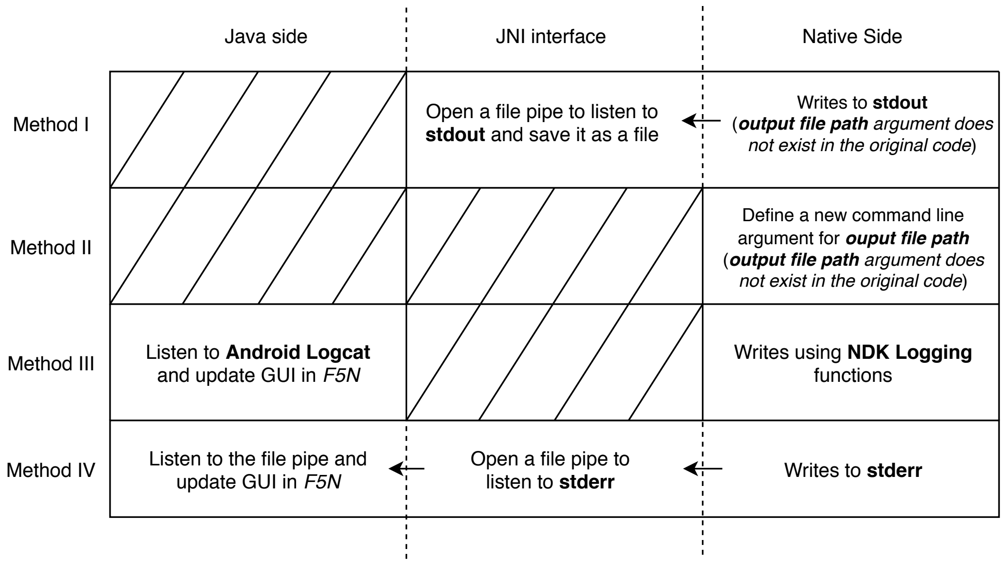
Java side
JNI interface
Native Side
Method I
Method II
Method III
Method IV
Open a file pipe to listen to **stdout** and save it as a file
Writes to **stdout** (***output file path** argument does not exist in the original code*)
Define a new command line argument for ***ouput file path***
(***output file path** argument does not exist in the original code*)
Listen to **Android Logcat** and update GUI in *F5N*
Writes using **NDK Logging** functions
Listen to the file pipe and update GUI in *F5N*
Open a file pipe to listen to **stderr**
Writes to **stderr**

Fig. S2. Method I - Using a file pipe to listen and save *standard output*. Method II - Define a new command line argument for *output file path*. Method III - Using NDK Logging functions to print *standard error* to Logcat. Method IV - Using a file pipe to listen *standard error*.

Typically a tool writes the results to the *standard ouput* and meta-information to the *standard error*. On Android, we want the results to be written to a file and meta-information to be displayed on a GUI in real-time. To write results to a file, most of the tools provide a command line argument called *file output path*. Otherwise, in terminal environments like in Linux, we can redirect the *standard output* to a file (using the output redirection operator '>'). On Android, this is not possible. To overcome this issue two methods can be adopted. The first is to open up a file pipe in JNI to listen to the *standard ouput*, where the results will get written to this file (Fig. S2 Method I). The second is to define the output file path as an argument in the original code (Fig. S2 Method II). However, all the libraries in *F5N* had output file path as an argument. Now we present two methods to catch the *standard error*, which should be displayed to the user in real-time. The first method is to replace all the *fprintf(stderr,...)* functions with functions defined in NDK logging [19]. Then the *standard error* will get written to Android Logcat [20]. From Android Logcat, it is again piped to be displayed (Fig. S2 Method III). In practice, this method does not guarantee to display the complete set of messages written to the *standard error* during an execution. The more robust method is to open a file descriptor to listen to the *standard error*. Then this file should be read in Java side and the GUI should be updated (Fig. S2 Method IV). This involves no amendments in the original libraries but a declaration of file descriptors in *JNI* code.

E. Cross-compiling an analysis tool

To facilitate the modifications discussed above, the original repositories of the tools were forked and changed. Different tools use different build configurations. The build scripts for each tool were re-written using CMake. Suppose an original tool is built with *GNU Make* using a *Makefile*. In that case, one has to extract the *source files, header files, compiler flags, linked libraries etc* to create a *CMakeLists.txt* file. Refer how a *CMakeLists.txt* was written for *htslib* [21]. CMake along with Ninja is the recommended native build setup for Android. Compiling with CMake allows ADB to go deep into the native code when debugging. This was really helpful to figure out the static/dynamic version issue related to *htslib* library. It was straightforward to link the necessary third party libraries(*HDF5* and *htslib*) and system libraries (*libz, liblog, libm* etc) with original libraries using CMake. To follow the full set of modifications please refer *Minimap2* [22] *Samtools* [23] and *F5C* [24]. One can build libraries without using CMake but by using already available *Standalone Toolchains*[25]. However, this eliminates the possibility to debug using ADB.

F. Dynamic object construction in Java side

The part of *F5N* written in Java is dynamic and adaptive. For example the arguments set for each tool is stored in JSON format and objects are created by converting JSON objects to Java objects. In this way, *F5N* and arguments are decoupled making it easy to alter the format of the arguments if needed. The widgets linked with arguments are drawn programmatically instead of manually drawing them on the layout. This makes it easy to extend *F5N* with new analysis tools (refer Supplementary Section III).

III. Integrating a new analysis tool to F5N

The following steps summarize the work flow to add a new DNA analysis tool written in C/C++ to *F5N*. Please refer Fig. S3 for *F5N* directory structure. *F5N* repository is available at https://github.com/SanojPunchihewa/f5n

```
f5n
└─app/src/main
    ├─cpp
    ├─java/com/mobilegenomics/f5n
    │   ├─activity
    │   ├─core
    │   ├─dto
    │   ├─fragments
    │   └─support
    ├─jniLibs
    │   ├─arm64-v8a
    │   └─armeabi-v7a
    └─res
```

Fig. S3. *F5N* project directory structure, only the important folders are shown. The directory *cpp* contains *CMakeLists.txt* and *interface_X.h* header files. The *.so* files are stored according their *ISA* inside *jniLibs* directory. The Java Class *PipelineStep* is inside directory *core*.

1) Identify third party libraries that the new tool depends on.
   e.g. *Samtools* depends on *htslib*.
2) Create dynamic versions of third party libraries if they are used by other analysis tools in *F5N*.
3) Create static versions of third party libraries if they are not used by other analysis tools in *F5N*.
4) Cross compile the new tool and link it with static third party libraries to create a dynamic library.
5) Place all the dynamic libraries in *jniLibs/[ANDROID_ABI]* directory.
6) Repeat the above steps for different *ANDROID_ABIs*. (*armeabi-v7a and arm64-v8a*)
7) Change *CMakeLists.txt* file in *cpp* directory to include dynamic third party dependencies.
   - `add_library(libnewdependency SHARED IMPORTED)`
   - `set_target_properties(libnewdependency PROPERTIES IMPORTED_LOCATION CMAKE_SOURCE_DIR/../jniLibs/$ANDROID_ABI/libnewdependency.so)`
8) Change *CMakeLists.txt* to include the new analysis library.
   - `add_library(libnew SHARED IMPORTED)`
   - `set_target_properties(libnew PROPERTIES IMPORTED_LOCATION CMAKE_SOURCE_DIR/../jniLibs/$ANDROID_ABI/libnew.so)`
9) Change *CMakeLists.txt* file to link the new analysis library. Refer Supplemenatary Section IV-E for more details.
   - `target_link_libraries(native-lib libminimap libsamtool libf5c libhts libnew libnewdependency1 libnewdependency2 ... $log-lib)`
10) Copy *interface_X.h* file with the function signature *init_X(int argc, char* argv[])* to *cpp* directory (X is the name of the new library).
11) Create *PipelineStep* objects in *PipelineStep* Class located in *core* directory for the sub-tools (commands) in the new analysis tool.
   - `new PipelineStep(COMMAND_ID, NEW_TOOL_NAME, "tool_name sub-tool_name(command)");`

   `ex: private static final PipelineStep f5cIndex = new PipelineStep(3, "F5C_INDEX", "f5c index");`

12) For each sub-tool (command) create a *JSON* file containing the list of arguments and their default values[26].
13) Link JSON files to the *configureArguments* method in *GUIConfiguration* Class located inside *java/com/mobilegenomics/f5n* directory, i.e. add new switch cases to match *NEW_SUB_TOOLs* and assign *JSON* files appropriately.
14) Import *interface_X.h* header file to *native-lib.cpp* source file located inside *cpp* directory.
    - `include "#interface_X.h"`
15) Follow a similar approach to other tools to call *init_X(int argc, char* argv[])* function in *native-lib.cpp* source file.

Moreover, the following list comprises of common mistakes that could happen when extending or rebuilding *F5N*. Please refer Supplementary Section II for details.

1) Overlooking compiler flags when creating a CMake build configuration.
   e.g. not including the compiler flag *-D_FILE_OFFSET_BITS=64* in *Minimap2* CMake build will result in file read failures.
2) Statically linking third party libraries that are used by more than one tool.
   e.g. Linking the static version of *htslib* library caused *Samtools* and *F5C* to fail.
3) Not Handling native exceptions in JNI interface
4) Not Handling native SIGSEGV signals and exit signals in JNI interface
5) Not Resetting command line argument vector before calling a new native function

IV. Advanced details

A. Battery power consumption

F5N is not built as an Android background process. That is because when processing a dataset, *F5N* may consume memory more than the recommended amount for a background process. Android kills such over memory consuming background processes. Hence, *F5N* is built as a regular Android application. However, this introduced a caveat. That is when running a pipeline, the device display should be kept on. That is because Android interrupts running applications once the display goes off. Hence, *F5N* has to keep the display on and this increases power consumption. Our workaround is to reduce the display brightness to its minimal value, once a pipeline execution starts. With this method, device B's battery (3060 mAh) drains approximately by 214.2 mAh for a complete methylation calling pipeline on a batch data-set ($\sim$34.49 Mbases on average).

B. Recover F5N after crashing

Android tends to kill a process or destroy an activity if something goes wrong. In *F5N* usually it is the case when the native code tries to over consume memory. Since this kill signal comes from the Android Kernel, it cannot be handled. The simple solution is to be aware of the device memory and set memory governing parameters accordingly (see Supplementary Section I). Saving the state of the application, i.e, saving previously executed command and loading it later, saves time time taken to configure the pipeline again. In the next attempt, the user is advised to tweak memory parameters as it is the solution most of the time.

C. Interrupting an executing pipeline

Already executing native code on Android cannot be stopped arbitrarily. The ramification of this is that the user cannot suspend or stop an executing pipeline. On the other hand, JVM being a multi-threaded process and JNI calls not being *POSIX async-signal-safe*, it is not possible to run the native code in a cloned process. As a consequence peak RAM usage will record the highest RAM usage for the whole application session rather than for the latest pipeline execution. To circumvent both problems the "safest" solution is to restart *F5N*. An advanced method to suspend or stop an executing command is by adding interrupt listeners to the original code. That is while the original code is being executed, it periodically checks its environment for an interrupt signal, e.g, state of a flag value in a file. The flag value can be changed on-the-fly to stop or suspend the execution. Please note that in *F5N*, when a pipeline is running the user can still go to the previous activity. In such attempts, a warning message is displayed saying that the pipeline will stop. This does not guarantee the complete termination of the native process but the termination of the running Java thread. Hence, the user is advised to restart the application to safely terminate a pipeline.

D. F5N storage and compatible file systems

A mobile phone has two storage types - internal storage and external storage (SD card storage). *FAT32* is a popular file system format used in SD cards. However, *FAT32* does not support files with more than the size of ∼4GB. Usually, the partitioned genome reference file exceeds the size 4GB. Therefore, the user is advised to use file system formats like *ext3, ext4, exFAT32* etc as the SD card file system format. Out of these formats, *exFAT32* is recommended. It is noteworthy that Google has introduced (from Android 8.0) a virtual file system wrapper called *SDCardFS* to regulate SD card access by Android applications. However, still, the SD card should have one of the compatible file system formats to work with larger files.

Downloading and extracting a dataset to the SD card can be done after setting SD card permission. On Android, writing to the SD card storage via native code (C/C++) can only be done using Storage Access Framework (SAF)[27]. To implement SAF, most of the original code has to be changed. One workaround is to use the *Method I* as illustrated in Fig. S2. In JNI interface, we can use SAF to write the *standard output* to the SD card. However, there are sub-tools that directly write to files, e.g., *F5C index* writes files automatically to the dataset directory. In such scenarios, the original tool should be reconfigured to facilitate SAF. The current version of *F5N* does not support this feature. Therefore the user cannot perform writes to the SD card but can read from it.

E. Including a new analysis tool

The Supplementary Section III explains the procedure to include a new analysis tool in *F5N*. However if the new tool creates a comparatively large .so file, instead of directly linking it with the existing .so files (step 14), it can be kept as a separate .so file which can be loaded to JVM as required. *Nanopolish's* .so file is ∼50 MB while the sum of the other .so files is ∼17 MB. Hence, we separated out *Nanopolish's* .so file and it is loaded to *F5N* only if the user set the pipeline type to *"Mode 2"* inside *F5N* settings (add screenshot). Every time the user switches between pipeline types - *"Mode 1"* and *"Mode 2"*, *F5N* needs to be restarted for the changes to take effect. This is because the JVM has to unload the existing library and load a new one. Please note that, in both pipeline types *Minimap2* and *Samtools* commands can be executed but *F5C* is only supported in *"Mode 1* and *Nanopolish* is only supported in *"Mode 2"*.

F. Creating a custom pipeline

F5N source code can be modified to have user defined custom pipelines as necessary. *F5N* already consists of three such pipelines (methylation calling pipelline, event alignment pipeline and Artic pipeline). A developer can easily choose to create a desired pipeline from the available analysis tools or after adding necessary tools as explained in Sections III and IV-E. Please refer [28] for an example custom pipeline source code.

G. Running any sub-tool available in an analysis tool

F5N has only a limited number of sub-tools with a GUI. However, the user can use *Terminal Activity* page, to write a command, which uses any sub-tool of the selected analysis tool. For an example, in *Samtools*, though GUIs are provided only for *Samtools sort* and *Samtools index*, *Samtools depth* commmands can be configured in *Terminal Activity* page. If a developer likes to build a GUI for a sub-tool, he only has to create a JSON file as explained in Section III (steps 12 and 13).

Supplementary References

[1] Heng Li. *Manual Reference Pages - Minimap2 (1)*. https://lh3.github.io/minimap2/minimap2.html.
[2] Samtools. *Manual page from samtools-1.10*. http://www.htslib.org/doc/samtools.html.
[3] Hasindu Gamaarachchi. *Manual Reference Pages - F5C*. https://hasindu2008.github.io/f5c/docs/overview.
[4] Android. *Android Studio and SDK*. https://developer.android.com/studio.
[5] Android. *Android NDK*. https://developer.android.com/ndk.
[6] Android. *JNI*. https://developer.android.com/training/articles/perf-jni.
[7] Android. *Android Debug Bridge*. https://developer.android.com/studio/command-line/adb.
[8] Kitware. *CMake build manager*. https://cmake.org/.
[9] Ninja. *Ninja build system*. https://ninja-build.org/.
[10] Fredrik Fornwall. *Termux Linux environment emulator*. https://play.google.com/store/apps/details?id=com.termux&hl=en.

[11] Ian F Darwin. *Android Cookbook: Problems and Solutions for Android Developers*. " O'Reilly Media, Inc.", 2017, "661–666".

[12] Android. *Android Cmake cross-compilation*. https://developer.android.com/ndk/guides/cmake.

[13] samtools. *htslib*. https://github.com/samtools/htslib.

[14] Mike Folk, Albert Cheng, and Kim Yates. "HDF5: A file format and I/O library for high performance computing applications". In: *Proceedings of supercomputing*. Vol. 99. 1999, pp. 5–33.

[15] *JNI handle exit calls*. http://jnicookbook.owsiak.org/recipe-no-016/.

[16] *JNI handle SIGSEGV calls*. http://jnicookbook.owsiak.org/recipe-no-015/.

[17] *JNI throw exceptions*. http://jnicookbook.owsiak.org/recipe-no-019/.

[18] Sanoj Punchihewa. *F5N JNI interface*. https://github.com/SanojPunchihewa/f5n/blob/master/app/src/main/cpp/native-lib.cpp.

[19] Android. *Logging Android NDK*. https://developer.android.com/ndk/reference/group/logging.

[20] Android. *Logcat command-line tool*. https://developer.android.com/studio/command-line/logcat.

[21] HTSLIB. *CMake build for HTSLIB*. https://github.com/hiruna72/htslib/blob/76f9eaa29a23573a70e37ca6ed842719e03cde55/INSTALL#L101.

[22] Minimap2. *CMake build for Minimap2*. https://github.com/SanojPunchihewa/minimap2-arm/tree/build-cmake#cmake-build.

[23] Samtools. *CMake build for Samtools*. https://github.com/hiruna72/samtools/blob/947c5b66cf91abc9b3e58b61642994dd8f4ae7e4/INSTALL#L103.

[24] F5C. *CMake build for F5C*. https://github.com/hiruna72/f5c/tree/cmake_build#building.

[25] Android. *Standalone Toolchains (Obsolete)*. https://developer.android.com/ndk/guides/standalone_toolchain.

[26] F5N. *F5N arguments JSON format*. https://github.com/SanojPunchihewa/f5n/blob/master/app/src/main/res/raw/minimap2.json.

[27] Android. *Open files using storage access framework*. https://developer.android.com/guide/topics/providers/document-provider.html.

[28] F5N. *Custom pipeline source code*. https://github.com/SanojPunchihewa/f5n/blob/master/app/src/main/java/com/mobilegenomics/f5n/core/PipelineStep.java.