Parallel Processing using Android

Prateek Swamy[1] Hemant Wade[2]
[1,2]Department of Computer Science & Engineering
[1,2]Nagpur University, India

Abstract— Although OpenMP isn't formally supported in android platform as of writing, it's attainable to induce C/C++ code using OpenMP for parallel optimizations.
Key words: OpenMP, Android, C, C++, Parallel

I. INTRODUCTION

Android could be a mobile package developed by Google, supported the UNIX system kernel and designed primarily for touchscreen mobile devices like smartphones and tablets. Android's computer program is especially supported direct manipulation, mistreatment bit gestures that loosely correspond to real-world actions, like swiping, sound and pinching, to govern on-screen objects, alongside a virtual keyboard for text input. additionally to touchscreen devices, Google has additional developed mechanical man TV for televisions, mechanical man automotive vehicle for cars, and mechanical man Wear for wrist joint watches, every with a specialised computer program. Variants of mechanical man are used on notebooks, game consoles, digital cameras, and different natural philosophy.

The OpenMP Application Programming Interface is one of the best emerging standards for parallel programming on shared-memory multiprocessors. It extends existing languages such as FORTRAN and C/C++ with a set of directives. To use parallelism with the code in OpenMP, the compiler directives are used. Using OpenMP, parallel processing is achievable in Android[1].

Parallel processing is a type of computation in which many calculations or the execution of processes are carried out simultaneously.[1] Large problems can often be divided into smaller ones, which can then be solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors.

II. USING OPENMP IN ANDROID

Although OpenMP isn't formally supported in android platform as of writing, it is possible to induce C/C++ code using OpenMP parallel optimizations to work in Android environment as follows:

A. Step 1: Use NDK to compile C/C++ for Android

Java is that the default programing language for android application development, and Java so works fine for writing general purpose application logic. However, it's doable to incorporate additionally C/C++ routines into android Java applications by assembling the C/C++ routines into a dynamic-link library exploitation that is Native Development Kit and calling these routines from android application using Java's JNI "Native" interface.

Using C/C++ code can be convenient in Android, because:

– Lot of useful pieces C/C++ software already exists and these can be desirable to reuse in Android/Java applications
– Execution speed for native C/C++ code is several times faster than Java code in Android, so algorithms involving intensive calculations can be desirable to write in C/C++ rather than Java.

B. Step 2: Setup OpenMP compiler flags

The Android NDK uses gcc cross-compiler for compiling the C/C++ source codes into Android executable binaries. To enable the OpenMP support in compilation, edit the Android NDK C/C++ project's Android.mk script file and add –fopenmp switch into compiler & linker settings:

– LOCAL_CFLAGS += -fopenmp
– LOCAL_LDFLAGS += -fopenmp

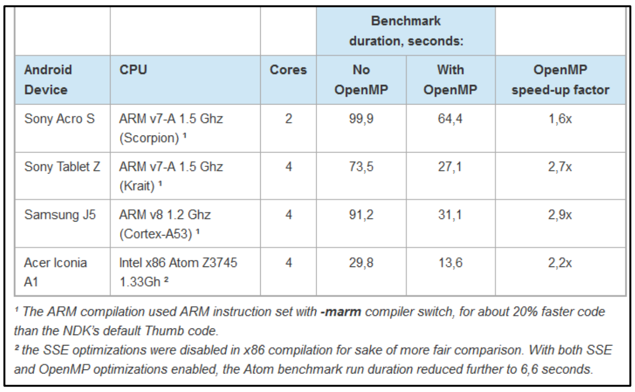
C. Step 3: Working with Android native threading bug

Android NDK v10 has an issue that OpenMP threading common storage settings do not get properly initialized for other threads than the Application's main thread.

This means that OpenMP-optimized functions can work ok if they're invoked from the robot Application main UI thread, however, invoking OpenMP calculations from the other thread than the most one can cause the full application to crash with terse fatal signal eleven because of access violation. This is undesirable behavior, specifically as intensive calculations don't seem to be to be run from main thread to avoid unresponsive UI.

A workaround for this thread storage issue is to invoke a fast JNI call at starting of the robot program from the most UI thread, to repeat a pointer to the most thread's OMP threading storage knowledge, and later utilize this keep pointer to initialize OpenMP support for alternative threads properly. With this arrangement OpenMP optimizations shall work conjointly once dead from robot application background threads

OpenMP benchmark results in Android is given below



			Benchmark duration, seconds:		
Android Device	**CPU**	**Cores**	**No OpenMP**	**With OpenMP**	**OpenMP speed-up factor**
Sony Acro S	ARM v7-A 1.5 Ghz (Scorpion) [1]	2	99,9	64,4	1,6x
Sony Tablet Z	ARM v7-A 1.5 Ghz (Krait) [1]	4	73,5	27,1	2,7x
Samsung J5	ARM v8 1.2 Ghz (Cortex-A53) [1]	4	91,2	31,1	2,9x
Acer Iconia A1	Intel x86 Atom Z3745 1.33Gh [2]	4	29,8	13,6	2,2x

[1] *The ARM compilation used ARM instruction set with* ***-marm*** *compiler switch, for about 20% faster code than the NDK's default Thumb code.*
[2] *the SSE optimizations were disabled in x86 compilation for sake of more fair comparison. With both SSE and OpenMP optimizations enabled, the Atom benchmark run duration reduced further to 6,6 seconds.*

Fig. 1: OpenMP Benchmark Results

III. MULTITHREADING IN ANDROID

To realize the maximum potential of the available processing power on multi-core devices, write your application with concurrency in mind. The application should be designed so that tasks which can be executed in parallel are set up to run on separate threads.

The priority that your app's threads receive depends partly on where the app is in the app lifecycle. As you create and manage threads in your application, it's important to set their priority so that the right threads get the right priorities at the right times. If set too high, your thread may interrupt the UI thread and Render Thread, causing your app to drop frames. If set too low, you can make your async tasks (such as image loading) slower than they need to be.

Every time you create a thread, you should call setThreadPriority(). The system's thread scheduler gives preference to threads with high priorities, balancing those priorities with the need to eventually get all the work done. Generally, threads in the foreground group get about 95% of the total execution time from the device, while the background group gets roughly 5%.

The system also assigns each thread its own priority value, using the Process class.

By default, the system sets a thread's priority to the same priority and group memberships as the spawning thread. However, your application can explicitly adjust thread priority by using setThreadPriority().

The Process class helps reduce complexity in assigning priority values by providing a set of constants that your app can use to set thread priorities. For example, THREAD_PRIORITY_DEFAULT represents the default value for a thread. Your app should set the thread's priority to THREAD_PRIORITY_BACKGROUND for threads that are executing less-urgent work.

A. The UI Thread

In Android, the main thread is the same as the UI thread. This thread is responsible for handling all the UI events.

When writing multi-threaded applications in Android, follow these things in mind about the UI thread (or main thread):

– Only the main thread should update the UI. All other threads in the application should return data back to the main thread to update the UI.
– There is no single point of entry in an Android application. An Android application can be entered from an Activity, Service or a Broadcast Receiver, all of which run on the UI thread.
– Very important – The UI thread should not perform tasks that take longer than a few seconds, or else you run the risk of a sluggish user experience in your app.

As a rule of thumb, whenever your application needs to perform a longer task(s) from the UI thread, then it should parallelize using one of the parallelization techniques provided by Android: Java threads, AsyncTask or IntentService.

IV. PARALLELIZATION TECHNIQUE: JAVA THREADS

The standard ways of creating threads in Java are also available in Android: extending the Thread class or implement the Runnable interface.

However, if you want to pass messages to and from a thread, you need to implement message queues using android.os.Message, android.os.Handler, android.os.Looper, etc.

Also, if your application involves creating multiple threads then you might have to take care of multi-threading concurrency issues like race conditions, deadlocks and starvation

V. PARALLELIZATION TECHNIQUE: ASYNCTASK

AsyncTask provides the functionality to run short tasks (a couple of seconds long) in the background from the UI thread using a method called doInBackground(). You do not need to implement any message passing to and from the UI thread.

An AsyncTask uses 3 types of data:

– Params - parameters passed to the background method as inputs.
– Progress - data passed to the UI thread for updating progress.
– Result - data returned from the background method upon completion.

You can implement an AsyncTask using the following steps, found in Android's AsyncTask documentation:

– Extend the AsyncTask class.
– Implement the following methods:
    1) onPreExecute() - performs setup like showing a progress dialog before executing the task. This method is invoked on the UI thread.
    2) doInBackground(Params...) - executes all of the code that you want to run in the background and sends updates to onProgressUpdate() and the result to onPostExecute(Result). It is invoked on a pool of background threads.
    3) onProgressUpdate() - invoked when publishProgress() is called from the doInBackground() method. It is invoked on the UI thread.
    4) onPostExecute() - receives the return value from doInBackground(). It is invoked on the UI thread.
    5) onCancelled() - invoked when cancel() is called. Invoked on the UI thread.
– Create an instance of your extended AsyncTask class on the UI thread.
– Call the execute() method.

Use an AsyncTask whenever you have a short background task which needs communication with the UI thread. AsyncTask is appropriate for short tasks only because it creates and manages threads for you, and you don't want to tie up resources with thread(s) that you did not create. Use Java threads and handlers from inside a service for longer-running tasks

VI. Conclusion

We can see that android devices takes advantage of OpenMP optimizations. the advance depends on the processor generation, and devices with a quad-core central processing unit naturally can gain a bigger profit than devices with a dual-core central processing unit.

References

[1] Mitsuhisa Sato, “OpenMP: Parallel programming API for shared memory multiprocessors and on-chip multiprocessors”, International Symposium on System Synthesis (ISSS) 2002, Kyoto, Japan, 2002**.**
[2] P. Swamy, M. M. Raghuwanshi and A. Gholghate, "An Improved Approach for k-Means Using Parallel Processing," 2015 International Conference on Computing Communication Control and Automation, Pune, 2015, pp. 358-361.

Website

[3] developer.qualcomm.com
[4] www.softwarecoven.com
[5] developer.android.com/studio