CHAPTER 10

# x86 NDK and C/C++ Optimizations

> *Technological progress has merely provided us with more efficient means for going backwards.*
>
> —Aldous Leonard Huxley

In the previous chapter we introduced the basic principles of performance optimization, optimization methodologies, and related tools for Android application development. Since Java is the primary application development language for Android developers, optimization tools presented in the previous chapter were mainly for Java. We know that the Java application is running in a virtual machine, and its speed is inherently slower than C/C++ applications, which are directly compiled and run on the hardware instruction. In addition, due to the underlying and fundamental nature of C/C++, developers for C/C++ applications have created more optimization tools.

## Vectorization

The Intel compiler supports advanced code generation, including *auto-vectorization*. For the Intel C/C++ compiler, vectorization is loop unrolling with the generation of single instruction, multiple data (SIMD) instructions operating on several elements at the same time. The developer can unroll loops manually and insert appropriate function calls corresponding to SIMD instructions. This approach is not forward-scalable and incurs high development costs. The work has to be redone when the new microprocessor with advanced instruction support is released. For example, early Intel Atom microprocessors did not benefit from vectorization of loops processing double-precision floating point while single-precision was processed by SIMD instruction effectively.

Auto-vectorization simplifies programming tasks because the programmer doesn't have to learn the instruction sets for each particular microprocessor. For example, the Intel compiler always supports the latest generations of Intel microprocessors.

The `-vec` options turn on vectorization at the default optimization level for microprocessors supporting IA32 architecture—both Intel and non-Intel. To improve the quality of vectorization, you need to specify the target microprocessor on which the

code will execute. For optimal performance on Android smartphones based on Intel architecture, it's best to use the `-xSSSE3_ATOM` option. Vectorization is enabled with the Intel C++ compiler at optimization levels of `-O2` and higher.

Many loops are vectorized automatically and most of the time the compiler generates optimal code on its own. However, sometimes it may require guidance from the programmer. The biggest problem with efficient vectorization is making the compiler estimate data dependencies as precisely as possible.

To take full advantage of Intel compiler vectorization, the following techniques are useful:

- Generate and understand a vectorization report
- Improve performance by pointer disambiguation
- Improve performance using inter-procedural optimization
- Use compiler pragmas

## Vectorization Report

This section starts with the implementation of memory copying. The loop takes the structure commonly used in Android source code:

***Listing 10-1.*** Memory Copying Implementation

```
// It is assumed that the memory pointed to by dst
// does not intersect with the memory pointed to by src

void copy_int(int* dst, int* src, int num)
{
int left = num;
if ( left <= 0 ) return;
do {
    left--;
    *dst++ = *src++;
} while ( left > 0 );
}
```

For experiments with vectorization, you'll reuse the `hello-jni` project. To do so, add the function to the new file called `jni/copy_cpp.cpp`. Add this file to the list of source files in `jni/Android.mk` as follows:

***Listing 10-2.*** Vectorization Failure

```
LOCAL_SRC_FILES := hello-jni.c copy_int.cpp
```

To enable a detailed vectorization report, add the `-vec-report3` option to the `APP_CFLAGS` variable in `jni/Application.mk`:

```
APP_CFLAGS := -O3 -xSSSE3_ATOM  -vec-report3
```

If you rebuild `libhello-jni.so`, you will notice that several remarks are generated:

```
jni/copy_int.cpp(6): (col. 5) remark: loop was not vectorized: existence of
vector dependence.
jni/copy_int.cpp(9): (col. 10) remark: vector dependence: assumed ANTI
dependence between src line 9 and dst line 9.
jni/copy_int.cpp(9): (col. 10) remark: vector dependence: assumed FLOW
dependence between dst line 9 and src line 9.
...
```

Unfortunately auto-vectorization failed, because too little information was available to the compiler. If the vectorization were successful, the assignment would be replaced as follows:

```
*dst++ = *src++;
//The previous statement would be replaced with
*dst = *src;
*(dst + 1) = *(src + 1);
*(dst + 2) = *(src + 2);
*(dst + 3) = *(src + 3);
dst += 4; src += 4;
```

The first four assignments would be performed in parallel by SIMD instructions. But parallel execution of assignments is invalid if the memory accessed on the left sides is also accessed on the right sides of assignment. Consider, for example, the case when `dst+1` is equal to `src+2`. In this case the final value at `dst+2` would be incorrect.

The remarks indicate which types of dependencies are conservatively assumed by the compiler preventing vectorization:

- *Flow* dependence is a dependence between the earlier store and the later load from the same memory location.
- *Anti* dependence is a dependence between an earlier load and a later store to the same memory location.
- *Output* dependence is between two stores to the same memory location.

From the code comment, it is safe to assume that the author required that the memory pointed to by `dst` and `src` not overlap. To communicate information to the compiler, it is sufficient to add restrict qualifiers to the `dst` and `src` arguments:

```
void copy_int(int * __restrict__ dst, int * __restrict__ src, int num)
```

The restrict qualifier was added to the C standard published in 1999. To enable support of C99, you need to add `-std=c99` to options. Alternatively, you may use the `-restrict` option to enable it for C++ and other C dialects. In the previous code, the `__restrict__` keyword has been inserted and is always recognized as a synonym for the `restrict` keyword.

If you rebuild the library again, you will notice that the loops are vectorized:

```
jni/copy_int.cpp(6): (col. 5) remark: LOOP WAS VECTORIZED.
```

In this example, vectorization failed due to compiler conservative analysis. There are other cases when the loop is not vectorized, including when:

- The instruction set does not allow for efficient vectorization. The following remarks indicate this type of issue:
    - “Non-unit stride used”
    - “Mixed data types”
    - “Operator unsuited for vectorization”
    - “Contains unvectorizable statement at line XX”
    - “Condition may protect exception”
- Compiler heuristics prevent vectorization. Vectorization is possible but may actually lead to a slow down. If this is the case, the diagnostics will contain:
    - "Vectorization possible but seems inefficient"
    - “Low trip count”
    - “Not inner loop”
- Vectorizer’s shortcomings:
    - “Condition too complex”
    - “Subscript too complex”
    - “Unsupported loop structure”

The amount of information produced by vectorizer is controlled by `-vec-reportN`. You may find additional details in the compiler documentation.

## Pragmas

As you saw, you can use the `restrict` pointer qualifier to avoid conservative assumptions about data dependencies. But sometimes it’s tricky to insert `restrict` keywords. If many arrays are accessed in the loop, it might also be too laborious to annotate all pointers. To simplify vectorization in these cases, you can use the Intel-specific pragma `simd`. You can use it to vectorize inner loops, assuming there are no dependencies between iterations.

Pragma `simd` applies only to `for` loops operating on native integer and floating-point types:

- The `for` loop should be countable with the number of iterations known before the loop starts.
- The loop should be innermost.
- All memory references in the loop should not fault (it is important for masked indirect references).

To vectorize the loop with a pragma, you need to rewrite the code into a `for` loop, as shown in Listing 10-3.

***Listing 10-3.*** Memory Copying Implementation that Can Be Vectorized

```
void copy_int(int* dst, int* src, int num)
{
#pragma simd
for ( int i = 0; i < num; i++ ) {
*dst++ = *src++;
}
}
```

Rebuild the example and note that the loop is vectorized. Simple loop restructuring for `pragma simd` and insertions of `#pragma simd` in Android OS sources allow you to improve the performance of the Softweg benchmark by 1.4x without modifying the benchmark itself.

# Auto-Vectorization and Limits

The previous sections' examples were based on the assumption that you had a good understanding of the code before starting your optimization efforts. If you are not familiar with the code, you can help the compiler to analyze it by extending the scope of the analysis. In the example with copying, the compiler should make conservative assumptions because it knows nothing about the `copy_int` routine's parameters. If call sites are available for analysis, the compiler can try to prove that the parameters are safe for vectorization.

To extend the scope of the analysis, you need to enable *interprocedural optimizations*. A few of these optimizations are enabled by default during single file compilation. Interprocedural optimizations are described in a separate section.

Vectorization cannot be used to speed up the Linux kernel code, because SIMD instructions are disabled in kernel mode with the `-mno-sse` option. This was done intentionally by kernel developers.

## Interprocedural Optimizations

The compiler can perform additional optimizations if it can optimize across function boundaries. For example, if the compiler knows that some function call argument is constant, then it can create a special version of the function specifically tailored to that constant argument. This special version later can be optimized with knowledge of the parameter value.

To enable optimization within a single file, specify the `-ip` option. When this option is specified, the compiler generates a final object file that can be processed by the system linker. The disadvantage of generating an object file is almost complete information loss; the compiler does not even attempt to extract information from the object files.

Single file scope may be insufficient for the analysis due to the information loss. In this case, you need to add the `-ipo` option. When you use this option, the compiler compiles files into an intermediate representation that is later processed by special Intel tools: xiar and xild.

You use the xiar tool for creating static libraries instead of the GNU archiver ar, and you use xild instead of the GNU linker ld. It is only required when the linker and archiver are called directly. A better approach is to use the compiler drivers `icc` or `icpc` for final linking. The downside of the extended scope is that the advantage of separate compilation is lost—each modification of the source requires relinking and relinking causes complete recompilation.

There is an extensive list of advanced optimization techniques that benefit from global analysis. **Chapter 9: Performance Optimizations for Android Applications on x86** introduced some of these techniques, and others will be discussed later this chapter. Note that some optimizations are Intel-specific and are enabled with `-x*` options.

Unfortunately, things are slightly more complicated in Android with respect to shared libraries. By default, all global symbols are preemptable. Preemptability is easy to explain by example. Consider the instance where the following libraries are linked into the same executable:

***Listing 10-4.*** libone.so, a Linked Library Example

```
int id(void) {
    return 1;
}
```

Listing 10-4 is the first library linked, and the second is described in Listing 10-5.

***Listing 10-5.*** libtwo.so, a Second Linked Library Example

```
int id( void ) {
    return 2;
}
int foo( void ) {
    return id();
}
```

Assume that the libraries were created simply by executing `icc -fpic -shared -o <libname>.so <libname>.c`. Only the strictly required options, `-fpic` and `-shared`, are given.

If the system dynamic linker loads the library `libone.so` before the library `libtwo.so`, the call to the function `id()` from the function `foo()` is resolved in the `libone.so` library.

When the compiler optimizes the function `foo()`, it cannot use its knowledge about `id()` from the `libtwo.so` library. For example, it cannot inline the `id()` function. If the compiler inlined the `id()` function, it would break the scenario involving `libone.so` and `libtwo.so`.

As a consequence, when you write shared libraries you should carefully specify which functions can be preempted. By default all global functions and variables are visible outside a shared library and can be preempted. The default setup is not convenient when you implement few native methods. In this case, you need to export only symbols that are called directly by the Dalvik Java virtual machine.

A symbol's visibility attribute specifies whether a symbol is visible outside the module and whether it can be preempted:

- "Default" visibility makes a global symbol visible outside the shared library and able to be preempted.
- "Protected" visibility makes a symbol visible outside the shared library, but the symbol cannot be preempted.
- "Hidden" visibility makes a global symbol visible only within the shared library and forbids preemption.

Returning to the `hello-jni` application, it is necessary to specify that the default visibility is hidden and that the functions exported for JVM have protected visibility.

To set default visibility to hidden, add `-fvisibility=hidden` to the `APP_CFLAGS` variable in `jni/Application.mk`:

```
APP_CFLAGS := -O3 -xSSSE3_ATOM  -vec-report3 -fvisibility=hidden -ipo
```

To override the visibility of `Java_com_example_hellojni_HelloJni_stringFromJNI`, add the attribute to the function definition:

```
Jstring __attribute__((visibility("protected")))
  Java_com_example_hellojni_HelloJni_stringFromJNI(JNIEnv* env, jobject
thiz)
```

With this flag set, the default visibility is hidden. This is the extent of the interprocedural optimizations that exist for the Intel NDK applications.

# Optimization with Intel IPP

You know from Chapter 7 that Android applications can bypass NDK development tools and use existing `.so` shared libraries developed by third parties. We use the Intel IPP libraries as an example in this chapter. Typical applications that use this library include multimedia and streaming applications, but any application where any time performance is an issue would benefit from this tool.

Intel IPP (Integrated Performance Primitives) is one of the high-performance libraries that Intel provides. It is a powerful function library for Intel processors and chipsets, and it covers math, signal processing, multimedia, image and graphics processing, vector computing, and other areas. A prominent feature of Intel IPP is that its code has been extensively optimized based on the features of any Intel processor, using a variety of methods. It can be said that it is a highly optimized, high-performance service library associated with Intel processors. Intel IPP has cross-platform features; it provides a set of cross-platform and operating system general APIs, which can be used for Windows, Linux, and other operating systems, and supports embedded, desktop, server, and other processor-scale systems.

In fact, Intel IPP is a set of libraries, each with different function areas within the corresponding library, and the libraries within Intel IPP differ slightly by the number of functions supported in different processor architectures. For example, Intel IPP 5.X image processing functions can support 2,570 functions in the Intel Architecture, while it supports only 1,574 functions in the IXP processor architecture.

The services provided by a variety of high-performance libraries, including Intel IPP, are multifaceted and multilayered. Applications can use Intel IPP directly or indirectly. Intel IPP provides support for applications, as well as for other components and libraries.

Applications using Intel IPP can be at two levels—they use the Intel IPP function interface directly, or use sample code to indirectly use Intel IPP. In addition, using the OpenCV library (a cross-platform Open Source Computer Vision Library) is equivalent to indirectly using the Intel IPP library. Both the Intel IPP and Intel MKL libraries eventually run on high-performance Intel processors on various architectures.

Taking into account the power of Intel IPP, and in accordance with the characteristics of optimized features of the Intel processor, you could use the Intel IPP library to replace some key source code that runs often and consumes a lot of time. You can obtain much higher performance acceleration than with general code. This is simply a "standing on the shoulders of giants" practical optimization method. Users can achieve optimization without manually writing code in critical areas.

Intel recently released Intel Android development environment code named Beacon Mountain. It provides both Intel IPP and Intel Threaded Building Blocks (TBB) for Android application developers. The average user can easily use Intel IPP, Intel TBB, Intel GPA, and other tools for Android application development. Examples of the Intel IPP can be found at `http://software.intel.com/en-us/articles/intel-integrated-performance-primitives-intel-ipp-intel-ipp-sample-code`.

# NDK Integrated Optimization Examples

We have introduced optimization based on the knowledge and basic theory of NDK-based optimization. This section uses a case study to demonstrate comprehensive optimization techniques by integrating NDK with C/C++.

The case is divided into two steps. The first step demonstrates a technique used on a local function compiled from C/C++ code to accelerate the computing tasks in a traditional Java-based program. The second step demonstrates the use of NDK compiler optimizations to achieve the C/C++ optimization task itself. We introduce each step in the following two sections of the chapter, which are closely linked.

# C/C++: The Original Application Acceleration

In the previous chapter, we introduced the use of Java code examples (SerialPi) to calculate π. In this section, we will change the computing tasks from Java to C code, using NDK to turn it into a local library. We then compare it with the original Java code tasks and you'll get some first-hand experience using C/C++ native library functions to achieve the traditional Java-based tasks acceleration.

The application used for this case study is named NDKExp, and we are using the Lenovo K800 as the target cell phone, which runs the interface shown in Figure 10-1.

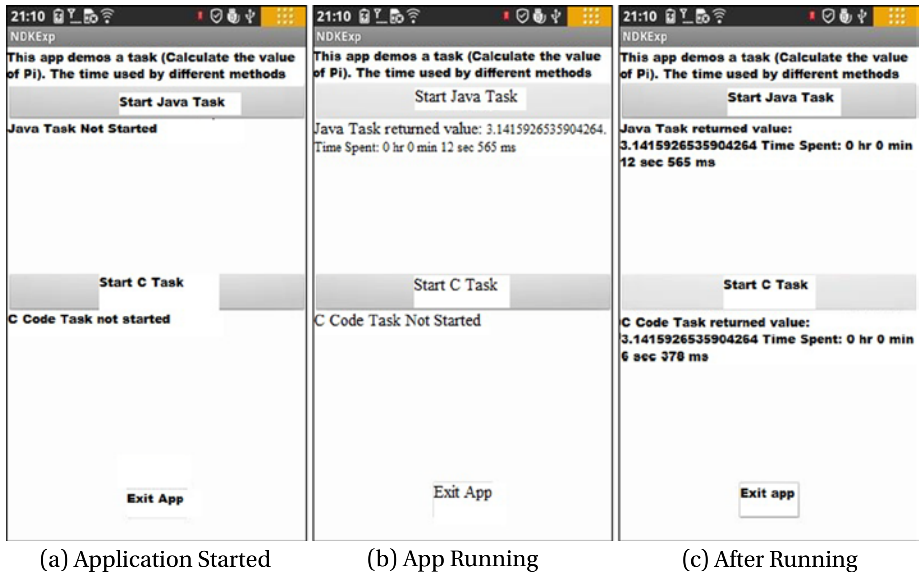



***Figure 10-1.*** *Original Version of NDKExp Running Interface*

Figure 10-1 (a) shows the application's main interface, including three buttons—Start Java Task, Start C Task, and Exit App. Clicking the Start Java Task button will start a traditional Java task (as shown in the source code of SerialPi written in Java, which calculates π). When the task is completed, the calculated results will be displayed below the button with the time spent, as shown in Figure 10-1 (b). Clicking the Start C Task button will start a computing task written in C using the same math formula to calculate π. When the task is completed, the calculated results will be displayed below the button with the time spent, as shown in Figure 10-1 (c).

As seen in Figure 10-1, for the same task, the application written in traditional Java takes 12.565 seconds to complete; the application written in C and compiled by the NDK development tool takes 6.378 seconds to complete. This example allows you to visually experience the power of using the NDK to achieve performance optimization.

This example is implemented as follows.

## Step 1: Create a New Android Application Project

1. Generate the project in Eclipse, name the project NDKExp, and choose the Build SDK option to support the x86 version of the API (in this case, the Android 4.0.3). Others are using the default values. After you have completed all these steps, generate the project.

2. Modify the main layout file. Put three text views and three buttons in the layout, set their Text and ID attributes, and adjust their size and position, as shown in Figure 10-2.

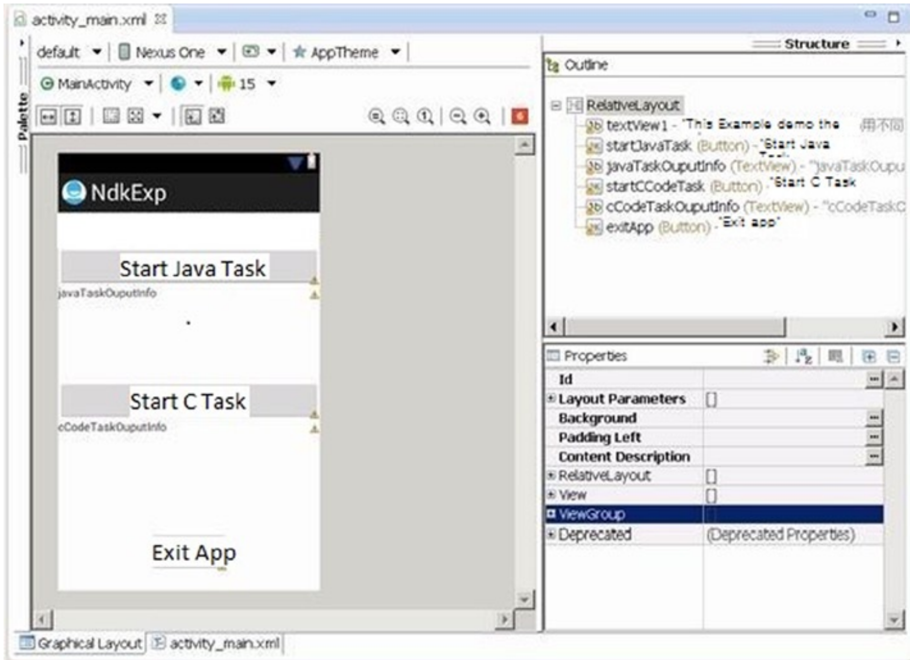



***Figure 10-2.*** *Layout of the Original Version NDKExp*

3. Modify the main layout of the class source code file MainActivity.java. It reads as follows:

```
1.  package com.example.ndkexp;
2.  import android.os.Bundle;
3.  import android.app.Activity;
4.  import android.view.Menu;
5.  import android.widget.Button;
6.  import android.view.View;
7.  import android.view.View.OnClickListener;
```

```
8.  import android.os.Process;
9.  import android.widget.TextView;
10. import android.os.Handler;
11. import android.os.Message;
12.
13. public class MainActivity extends Activity {
14.    private JavaTaskThread javaTaskThread = null;
15.    private CCodeTaskThread cCodeTaskThread = null;
16.      private TextView tv_JavaTaskOuputInfo;
17.      private TextView tv_CCodeTaskOuputInfo;
18.      private Handler mHandler;;
19.    private long end_time;
20.    private long time;
21.    private long start_time;
22.      @Override
23.      public void onCreate(Bundle savedInstanceState) {
24.          super.onCreate(savedInstanceState);
25.          setContentView(R.layout.activity_main);
26.          tv_JavaTaskOuputInfo = (TextView)findViewById
    (R.id.javaTaskOuputInfo);
27.        tv_JavaTaskOuputInfo.setText("Java the task is not
    started ");
28.          tv_CCodeTaskOuputInfo = (TextView)findViewById
    (R.id.cCodeTaskOuputInfo);
29.          tv_CCodeTaskOuputInfo.setText("C  code task is not
    start ");
30.          final Button btn_ExitApp = (Button) findViewById
    (R.id.exitApp);
31.          btn_ExitApp.setOnClickListener(new /*View.*/
    OnClickListener(){
32.              public void onClick(View v) {
33.                exitApp();
34.              }
35.          });
36.          final Button btn_StartJavaTask = (Button)
    findViewById(R.id.startJavaTask);
37.          final Button btn_StartCCodeTask = (Button)
    findViewById(R.id.startCCodeTask);
38.           btn_StartJavaTask.setOnClickListener(new /*View.*/
    OnClickListener(){
39.              public void onClick(View v) {
40.                btn_StartJavaTask.setEnabled(false);
41.                btn_StartCCodeTask.setEnabled(false);
42.                btn_ExitApp.setEnabled(false);
43.                startJavaTask();
44.                }
45.          });
```

```
46.          btn_StartCCodeTask.setOnClickListener(new /*View.*/
    OnClickListener(){
47.              public void onClick(View v) {
48.                btn_StartJavaTask.setEnabled(false);
49.                btn_StartCCodeTask.setEnabled(false);
50.                btn_ExitApp.setEnabled(false);
51.                startCCodeTask();
52.                }
53.          });
54.          mHandler = new Handler() {
55.              public void handleMessage(Message msg) {
56.              String s;
57.               switch (msg.what)
58.                {
59.                case JavaTaskThread.MSG_FINISHED:
60.                    end_time = System.currentTimeMillis();
61.                    time = end_time - start_time;
62.                    s = " The return value of the Java task "+
    (Double)(msg.obj) +"  Time consumed:"
63.               + JavaTaskThread.msTimeToDatetime(time);
64.                    tv_JavaTaskOuputInfo.setText(s);
65.                    btn_StartCCodeTask.setEnabled(true);
66.                  btn_ExitApp.setEnabled(true);
67.                break;
68.                case CCodeTaskThread.MSG_FINISHED:
69.                    end_time = System.currentTimeMillis();
70.                    time = end_time - start_time;
71.                    s = " The return value of the C code
    task"+ (Double)(msg.obj) +"  time consumed:"
72.               + JavaTaskThread.msTimeToDatetime(time);
73.                    tv_CCodeTaskOuputInfo.setText(s);
74.                     btn_StartJavaTask.setEnabled(true);
75.                  btn_ExitApp.setEnabled(true);
76.                break;
77.                default:
78.                  break;
79.                }
80.              }
81.          };
82.      }
83.
84.      @Override
85.      public boolean onCreateOptionsMenu(Menu menu) {
86.          getMenuInflater().inflate(R.menu.activity_main, menu);
87.          return true;
88.      }
89.
```

```
90.      private void startJavaTask() {
91.        if (javaTaskThread == null)
92.          javaTaskThread = new JavaTaskThread(mHandler);
93.        if (! javaTaskThread.isAlive())
94.        {
95.               start_time = System.currentTimeMillis();
96.               javaTaskThread.start();
97.               tv_JavaTaskOuputInfo.setText("The Java task is
    running...");
98.        }
99.      }
100.
101.     private void startCCodeTask() {
102.       if (cCodeTaskThread == null)
103.         cCodeTaskThread = new CCodeTaskThread(mHandler);
104.       if (! cCodeTaskThread.isAlive())
105.       {
106.              start_time = System.currentTimeMillis();
107.              cCodeTaskThread.start();
108.              tv_CCodeTaskOuputInfo.setText("C codes task is
    running...");
109.       }
110.     }
111.     private void exitApp() {
112.       try {
113.         if (javaTaskThread !=null)
114.         {
115.           javaTaskThread.join();
116.           javaTaskThread = null;
117.         }
118.       } catch (InterruptedException e) {
119.       }
120.       try {
121.         if (cCodeTaskThread  !=null)
122.         {
123.           cCodeTaskThread.join();
124.           cCodeTaskThread = null;
125.         }
126.       } catch (InterruptedException e) {
127.       }
128.     finish();
129.       Process.killProcess(Process.myPid());
130.     }
131.
132.   static {
133.     System.loadLibrary("ndkexp_extern_lib");
134.   }
135. }
```

The preceding code is basically the same as the example code for SerialPi. The code in lines 123 through 134 is the only new part. This code requires that the `libndkexp_extern_lib.so` shared library file be loaded before the application is running. The application needs to use local functions in this library.

4. The new threads task class `JavaTaskThread` in the project is used to calculate π. The code is similar to the `MyTaskThread` class code in the SerialPi example and is omitted here.

5. The thread task class `CCodeTaskThread` in the new project calls the local function to calculate π; its source code files `CCodeTaskThread.java` read as follows:

```
1.  package com.example.ndkexp;
2.  import android.os.Handler;
3.  import android.os.Message;

4.  public class CCodeTaskThread extends Thread {
5.    private Handler mainHandler;
6.    public static final int MSG_FINISHED = 2;
        // The message after the end of the task
7.    private native double cCodeTask();
       // Calling external C functions to accomplish computing
    tasks
8.    static String msTimeToDatetime(long msnum){
9.    long hh,mm,ss,ms, tt= msnum;
10.      ms = tt % 1000; tt = tt / 1000;
11.      ss = tt % 60; tt = tt / 60;
12.      mm = tt % 60; tt = tt / 60;
13.      hh = tt % 60;
14.      String s = "" + hh +" Hour "+mm+" Minute "+ss + " Second
    " + ms +" Millisecond ";
15.      return s;
16.      }
17.      @Override
18.      public void run()
19.      {
20.      double pi = cCodeTask();
        // Calling external C function to complete the calculation
21.      Message msg = new Message();
22.      msg.what = MSG_FINISHED;
23.      Double dPi = Double.valueOf(pi);
24.        msg.obj = dPi;
25.        mainHandler.sendMessage(msg);
26.      }
```

```
27.      public CCodeTaskThread(Handler mh)
28.      {
29.         super();
30.         mainHandler = mh;
31.      }
32.  }
```

The previous code is similar to the code framework of the `MyTaskThread` class of the SerialPi example. The main difference is at line 20. The original Java code for calculating π is replaced by calling a local function `cCodeTask`. To state that the `cCodeTask` function is a local function, you must add the local declaration of the function in line 7.

6. Build the project in Eclipse. Similarly here we have just a build, rather than run.

7. Create the `jni` subdirectory in the project's root directory.

## Step 2: Write the C Implementation Code of the cCodeTask Function

According to the method described in the **NDK Examples** section of **Chapter 7: Creating and Porting NDK-based Android Applications,** you need to compile the file into a `.so` library file. The main steps are as follows:

1. Create a C interface file. Since the case is a `CCodeTaskThread` class using a local function, you need to generate the class header file according to the class file of this class. At the command line, go to the project directory and then run the following command:

```
E:\temp\Android Dev\workspace\NdkExp> javah -classpath "D:\
Android\android-sdk\platforms\android-15\android.jar";bin/classes
com.example.ndkexp.CCodeTaskThread
```

This command will generate a file in the project directory named `com_example_ndkexp_CCodeTaskThread.h`. The main content of the document is as follows:

```
   ...
23.  JNIEXPORT jdouble JNICALL Java_com_example_ndkexp_
   CCodeTaskThread_cCodeTask
24.  (JNIEnv *, jobject);
   ...
```

In lines 23–24, the prototype of local function `cCodeTask` is defined.

2. Based on the previous header files, you create corresponding C code files in the `jni` directory of the project. In this case, we named it `mycomputetask.c`, which reads as follows:

```
1.  #include <jni.h>
2.  jdouble Java_com_example_ndkexp_CCodeTaskThread_cCodeTask
    (JNIEnv* env,  jobject thiz )
3.  {
4.    const long num_steps = 100000000;  // The total step length
5.    const double step = 1.0 / num_steps;
6.      double x, sum = 0.0;
7.      long i;
8.    double pi = 0;
9.
10.    for (i=0; i< num_steps; i++){
11.          x = (i+0.5)*step;
12.          sum = sum + 4.0/(1.0 + x*x);
13.      }
14.      pi = step * sum;
15.
16.    return (pi);
17.  }
```

Lines 4 through 16 are the body of the function—the code calculating π, which is the code that corresponds to the `MyTaskThread` class in the SerialPi example. Note that in line 4, the value of the variable `num_steps` (the total step length) must be the same as the value of the same step size that the `JavaTaskThread` class represents. Otherwise, there is no significance in comparing.

The first line of each `jni` file must contain the headers. Line 2 is the `cCodeTask` function prototype and is based on a slightly modified header files obtained in the previous step.

Line 16 shows the return results. With the `double` type of Java, which corresponds to the `jdouble` type of C, C can have a `pi` variable in type `double` returned directly to it. This is something we discussed in the introduction to this chapter.

3. In the project `jni` directory, you must create the `Android.mk` and `Application.mk` files. The content of `Android.mk` reads as follows:

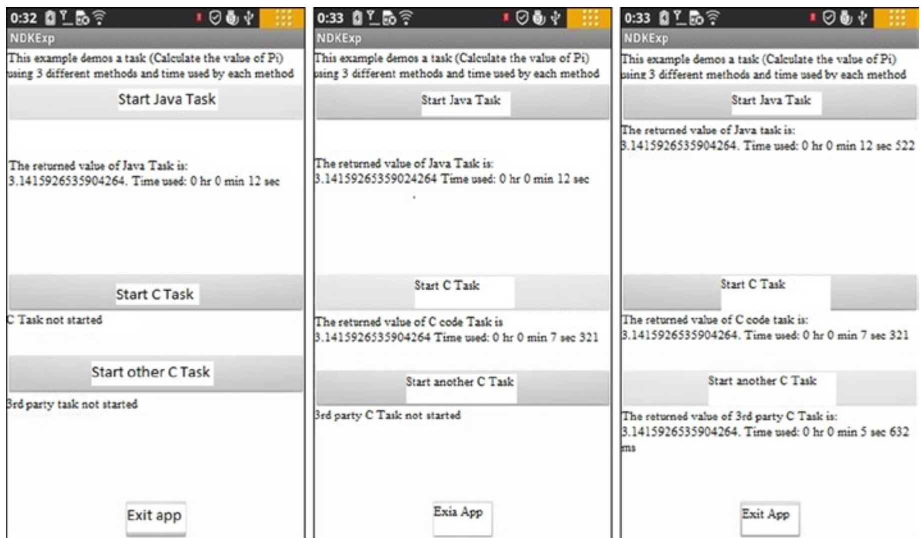
```
1.  LOCAL_PATH := $(call my-dir)
2.  include $(CLEAR_VARS)
3.  LOCAL_MODULE     := ndkexp_extern_lib
4.  LOCAL_SRC_FILES  := mycomputetask.c
5.  include $(BUILD_SHARED_LIBRARY)
```

Line 4 specifies the C code in the case file. Line 3 indicates the filename of the generated library, and its name must be consistent with the parameters of the `System.loadLibrary` function in line 133 of the project file `MainActivity.java`.

4. According to the method described in **Chapter 7**'s section on **NDK Examples**, you compile the C code into the `.so` library file under the `lib` directory of the project.

5. Deployment: run the project.

The application running interface is shown in the Figure 10-3.




(a) Screen After Task (b) App Unoptimized (c) App Optimized

***Figure 10-3.** Extended Version of the NDKExp Running Interface*

## Compiler Optimization Extension Application

In the previous example, you witnessed the capabilities of NDK for application acceleration. However, this application implemented only one local function and can't provide you with information to compare the effects of compiler optimizations. For this purpose, you need to rebuild the application and use it to experiment with the effects of compiler optimizations.

The application running the interface is shown in Figure 10-3.

The application has four buttons. When you click on the Start Java Task button, the response code does not change.

When you click the Start C Task or Start Other C Task button, the application will start a local function to run.

The code (the function body) of the two functions is the same. It calculates the values of π, but with a different name. The first one calls the `cCodeTask` function, while the second one calls the `anotherCCodeTask` function. These two functions are located in the `mycomputetask.c` and `anothertask.c` files and they correspond to the library files `libndkexp_extern_lib.so` and `libndkexp_another_lib.so`, respectively, after being compiled. In this case, you compile `libndkexp_extern_lib.so` using the `-O0` option and compile `libndkexp_another_lib.so` using the `-O3` option, so one is compiled non-optimized and the other is compiled optimized.

Therefore, clicking Start C Task will run the unoptimized version of the C function, as shown in Figure 10-3 (b), and clicking Start Other C Task will run the optimized version of the C functions, such as in Figure 10-3 (c). After task execution, the system displays the calculated results for the consumption of time.

As can be seen from the figure, whether or not the compiler optimizations are used, the running time of the local function is always shorter than the running time (12.522 seconds) of Java functions. Relatively speaking, the execution time (5.632 seconds) of the `-O3` optimization function is shorter than the execution time (7.321 seconds) of the unoptimized (`-O0` compiler option) function.

From this comparison, you can see that using compiler optimizations actually reduces application execution time. Not only that, it is even shorter than the original application running time (6.378 seconds). This is because the original application without compiler options defaults to the `-O1` level of optimization, whereas the `-O3` optimization level is even higher than the original application, so it's not surprising that it has the shortest running time.

The following application is a modified and extended version of the original application NDKExp. The steps are as follows.

## Step 1: Modify the Android Part of the Application

1. Modify the main layout file. Add a text view and a button in a layout. Set their Text and ID properties, and adjust their size and position, as shown in Figure 10-4.

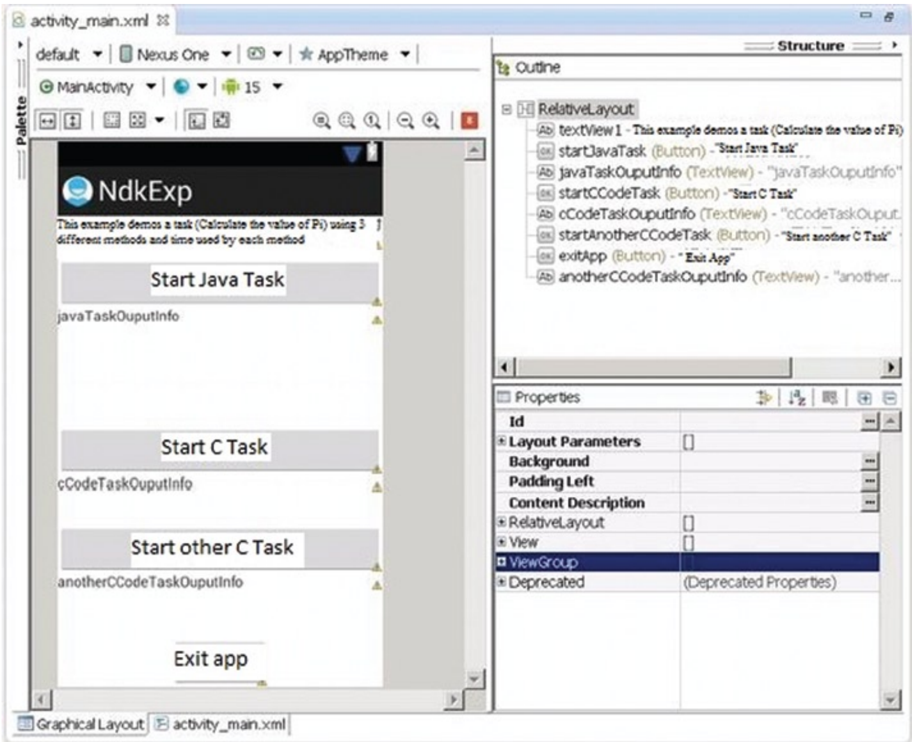



*Figure 10-4.* *Extended Version NDKExp Layout*

2. Modify the class source code file `MainActivity.java` of the main layout. The main changes are as follows:

```
    ...
13.  public class MainActivity extends Activity {
14.     private JavaTaskThread javaTaskThread = null;
15.     private CCodeTaskThread cCodeTaskThread = null;
16.     private AnotherCCodeTaskThread anotherCCodeTaskThread = null;
17.     private TextView tv_JavaTaskOuputInfo;
18.     private TextView tv_CCodeTaskOuputInfo;
19.     private TextView tv_AnotherCCodeTaskOuputInfo;
    ...
182.    static {
183.        System.loadLibrary("ndkexp_extern_lib");
184.        System.loadLibrary("ndkexp_another_lib");
185.    }
186. }
```

In lines 16 and 19, add the required variables for the new Start Other C Task button.

The key change is in the line 184. In addition to loading the original shared library files, those files are also added into another library file.

3. In the project, add a thread task class with the name `AnotherCCodeTaskThread` that calls a local function to calculate π. Its source code file `AnotherCCodeTaskThread.java` reads as follows:

```
1.  package com.example.ndkexp;
2.  import android.os.Handler;
3.  import android.os.Message;

4.  public class AnotherCCodeTaskThread extends Thread {
5.    private Handler mainHandler;
6.    public static final int MSG_FINISHED = 3;
      // The message after the end of the task
7.    private native double anotherCCodeTask();
     // Calling external C functions to complete computing tasks

8.    static String msTimeToDatetime(long msnum){
9.      long hh,mm,ss,ms, tt= msnum;
10.     ms = tt % 1000; tt = tt / 1000;
11.     ss = tt % 60; tt = tt / 60;
12.     mm = tt % 60; tt = tt / 60;
13.     hh = tt % 60;
14.     String s = "" + hh +"Hour "+mm+"Minute "+ss + "Second " +
    ms +"Millisecond";
15.     return s;
16.     }

17.    @Override
18.    public void run()
19.    {
20.    double pi = anotherCCodeTask();
       // Calling external C function to complete the calculation
21.        Message msg = new Message();
22.        msg.what = MSG_FINISHED;
23.        Double dPi = Double.valueOf(pi);
24.        msg.obj = dPi;
25.        mainHandler.sendMessage(msg);
26.    }

27.    public CCodeTaskThread(Handler mh)
28.    {
29.    super();
30.    mainHandler = mh;
31.    }
32.  }
```

The previous code is almost transcribing the code of the CCodeTaskThread class. It only does a little processing by calling another external C function called anotherCCodeTask to complete computing tasks in line 20. In line 7 it provides appropriate instructions for local functions and changes the value of the message type in line 6. In this way it distinguishes itself with a completed message by the previous C function. Line 4 shows that the task class is inherited from the thread class.

4. Build the project in Eclipse. Similarly here, you have just a build, rather than a run.

## Step 2: Modify the Makefile File of mycomputetask.c and Rebuild the Library Files

1. Modify the Android.mk file under the jni directory of the project, which reads as follows:

```
1.  LOCAL_PATH := $(call my-dir)
2.  include $(CLEAR_VARS)
3.  LOCAL_MODULE    := ndkexp_extern_lib
4.  LOCAL_SRC_FILES := mycomputetask.c
5.  LOCAL_CFLAGS    := -O0
6.  include $(BUILD_SHARED_LIBRARY)
```

Unlike the original application, in line 5 you add the parameters of the command LOCAL_CFLAGS passed to gcc. The value -O0 means no optimization.

2. Compile the C code file into the .so library file in the lib directory of the project.

3. Save the .so library files in the lib directory of the project (in this example, the file is libndkexp_extern_lib.so) to some other directory in the disk somewhere. The following operations will delete this .so library file.

## Step 2: Write the C Implementation Code for the anotherCCodeTask Function

Copy the processing steps for the cCodeTask function from the previous section. Then compile the file into the .so library file. The main steps are as follows:

1. Create a C interface file. At the command line, go to the project directory, and then run the following command:

```
E:\temp\Android Dev\workspace\NdkExp> javah -classpath
"D:\Android\android-sdk\platforms\android-15\android.jar";bin/
classes  com.example.ndkexp.AnotherCCodeTaskThread
```

This command will generate a directory named `project com_example_ndkexp_AnotherCCodeTaskThread.h` file. The main contents of the file are:

```
    ...
23.  JNIEXPORT jdouble JNICALL Java_com_example_ndkexp_
AnotherCCodeTaskThread_anotherCCodeTask
24.  (JNIEnv *, jobject);
    ...
```

Lines 23–24 define the local function `anotherCCodeTask` prototype.

2. According to the previously mentioned header files in the project `jni` directory, establish corresponding C code files, in this case named `anothertask.c`, the content of which is based on the `mycomputetask.c` modification. The modification as follows:

```
1.  #include <jni.h>
2.  jdouble Java_com_example_ndkexp_AnotherCCodeTaskThread_
anotherCCodeTask (JNIEnv* env,  jobject thiz )
3.  {
    ...
17. }
```

The second line of `mycomputetask.c` is replaced by the prototype of the `anotherCCodeTask` function. This is the same function prototype copied from the description about the function prototype of the `.h` file, which was created in the previous step with minor revisions. The final form can be seen in code line 2.

3. Modify the `Android.mk` file under the `jni` directory in the project, as follows:

```
1.  LOCAL_PATH := $(call my-dir)
2.  include $(CLEAR_VARS)
3.  LOCAL_MODULE       := ndkexp_another_lib
4.  LOCAL_SRC_FILES    := anothertask.c
5.  LOCAL_CFLAGS       := -O3
6.  include $(BUILD_SHARED_LIBRARY)
```

In line 4, the value is replaced with the new C code file `anothertask.c`. In line 3, the value is replaced with new library filename, which is consistent with the parameters of the `System.loadLibrary` function (which is in line 184 of the `MainActivity.java` file in the project). In line 5, the value of the `LOCAL_CFLAGS` parameter for the passed gcc command is replaced with `-O3`, which represents the highest level of optimization.

4. Compile the C code file into the .so library file under the lib directory of the project. Then you can see that the libndkexp_extern_lib.so documents under the lib directory in the project disappeared and were replaced by a newly generated libndkexp_another_lib.so file. It is very important to save library files.

5. Put the previously saved libndkexp_extern_lib.so library file back into the libs directory in the project.

There are now two files in the directory. You can use the dir command to verify:

```
E:\temp\Android Dev\workspace\NdkExp>dir libs\x86
2013-02-28  00:31            5,208 libndkexp_another_lib.so
2013-02-28  00:23            5,208 libndkexp_extern_lib.so
```

6. You redeploy, and run the project.

The application running the interface is shown in Figure 10-3, earlier in this chapter.

# Multiple Situations Comparison of Compiler Optimization Extensions

Through the case studies in this chapter, you have first-hand experience about the effects of compiler optimization. Task execution time was shortened from 7.321 seconds before optimization to 5.632 seconds after optimization. We only compared the difference of the gcc -O3 and -O0 command. You can extend this configuration by modifying the Android.mk file when compiling the two files—mycomputetask.c and anothertask.c—and then continue to compare the differences in the optimizing effects when using different compiler command options. To modify the Android.mk file, you only need to modify the value of the LOCAL_CFLAGS item. You can select many options of the gcc command to compare. Here are a few examples to illustrate this process.

## Example: Compare the Optimization Results by Using SSE Instructions

You can have the Start C Task button correspond to the Android.mk file of mycomputetask.c compile:

```
LOCAL_CFLAGS      := -mno-sse
```

and have the Start Other C Task button correspond to the Android.mk file of the anothertask.c compile:

```
LOCAL_CFLAGS      := -msse3
```

The former tells the compiler not to compile SSE instructions; the latter allows the compiler to program into SSE3 instructions. The reason to choose SSE3 instructions is that SSE3 is the highest level of instructions that the Intel Atom processor supports.

The results of running the application are shown in Figure 10-5.

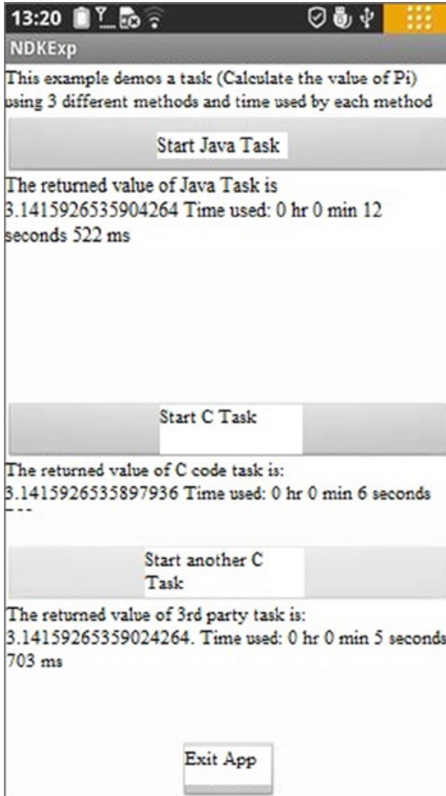



***Figure 10-5.*** *Optimization Comparison of Compiler SSE Instructions for Application NDKExp*

Seen from Figure 10-5, the same task using an SSE instruction execution time is shorter than not using an SSE instruction. The execution time is shortened from the original 6.759 seconds to 5.703 seconds.

It needs to be noted that, in this example, we finished modifying `Android.mk` files and reran `ndk-build` to generate the `.so` library file. We immediately deployed and ran the NDKExp project, but found out that we could not reach the desired effect. The reason is because only the `.so` library files are updated. The project manager of Eclipse does not

detect that that the project needs to rebuild. As a result, the apk did not get updates, and NDKExp application on the target machine code would not update the original code. Considering this situation, you can use the following methods to avoid this problem:

1. Uninstall the application from the phone.
2. Delete the classes.dex, jarlist.cache, and NdkExp.apk documents from the bin subdirectory of host project directory.
3. Delete the project in Eclipse.
4. In Eclipse, re-import the project.
5. Finally, redeploy and run projects, so you can have the desired effect.

This example only compares the effect of the SSE instructions. Interested readers can try other gcc compiler options and compare their operating results.

In addition, in the previous examples, what we are concerned with is the NDK effect only, so the C functions still use single-threaded code. Interested readers could combine the NDK optimization knowledge they learned from this chapter with the multithreading optimization in the previous chapter and change the C function to multithreading to implement along with the compiler optimization. Such a written set of optimization techniques in a variety of applications will certainly allow the application to run faster.

# Overview

Similar to Chapter 9, this chapter focused heavily on the code and technical aspects of the Android NDK on Intel's x86 architecture. We walked through creating a simple Android NDK application to show off how all of these pieces connect and ran it on an x86 emulator. The chapter also provided a high-level look at the optimizations that the Android NDK compiler can provide its developers. We then looked at Intel's Integrated Performance Primitives library (IPP), a high-performance library provided to x86 developers. Finally, we wrapped the chapter up with some examples of how to use all of the tools and tricks discussed.