Is it possible to use sun.misc.Unsafe to call C functions without JNI?

Asked 4 years, 5 months ago Active 1 year, 6 months ago Viewed 5k times

23

15

A piece of C/C++ code could provide a JNI method with an array of function pointers. But is there a way to call to the stack the functions that array's pointers are pointing to, directly from inside Java code (without using JNI or similar)? JNI somehow does something like that, so there must be a way. How does JNI do it? Is it via sun.misc.Unsafe? Even if it is not, could we use some Unsafe workaround to get our hands on the JVM code that does that?

I don't plan to use that commercially of course. I'm not even a professional, I just really enjoy coding and I've been studying CUDA lately so I thought maybe I could experiment with mixing everything together, but the overhead of JNI calls would defeat the purpose of having GPU accelerated code.

java c jvm java-native-interface native-code

edited Mar 30 '16 at 13:10

apangin
70.5k 9 148 181

asked Mar 30 '16 at 1:35

FinnTheHuman
1,005 10 27

JNI does it by being built into the JVM. – user253751 Mar 30 '16 at 1:42

2 Answers

Active | Oldest | Votes

68

Is JNI that slow?

JNI has already been optimized a lot, you should give it a try first. But it indeed has certain overhead, see details.

This overhead can be significant if a native function is simple and is called frequently. JDK has a private API called **Critical Natives** to reduce overhead of calling functions that do not require much of JNI functionality.

Critical Natives

A native method must satisfy the following conditions to become a critical native:

- must be **static** and **not synchronized**;
- argument types must be **primitive** or **primitive arrays**;
- implementation must not call JNI functions, i.e. it cannot allocate Java objects or throw exceptions;
- should not run for a long time, since it **will block GC** while running.

The declaration of a critical native looks like a regular JNI method, except that

- it starts with `JavaCritical_` instead of `Java_`;
- it does not have extra `JNIEnv*` and `jclass` arguments;
- Java arrays are passed in two arguments: the first is an array length, and the second is a pointer to raw array data. That is, no need to call `GetArrayElements` and friends, you can instantly use a direct array pointer.

```
JNIEXPORT jint JNICALL
Java_com_package_MyClass_nativeMethod(JNIEnv* env, jclass klass, jbyteArray array) {
    jboolean isCopy;
    jint length = (*env)->GetArrayLength(env, array);
    jbyte* buf = (*env)->GetByteArrayElements(env, array, &isCopy);
    jint result = process(buf, length);
    (*env)->ReleaseByteArrayElements(env, array, buf, JNI_ABORT);
    return result;
}
```

will turn to

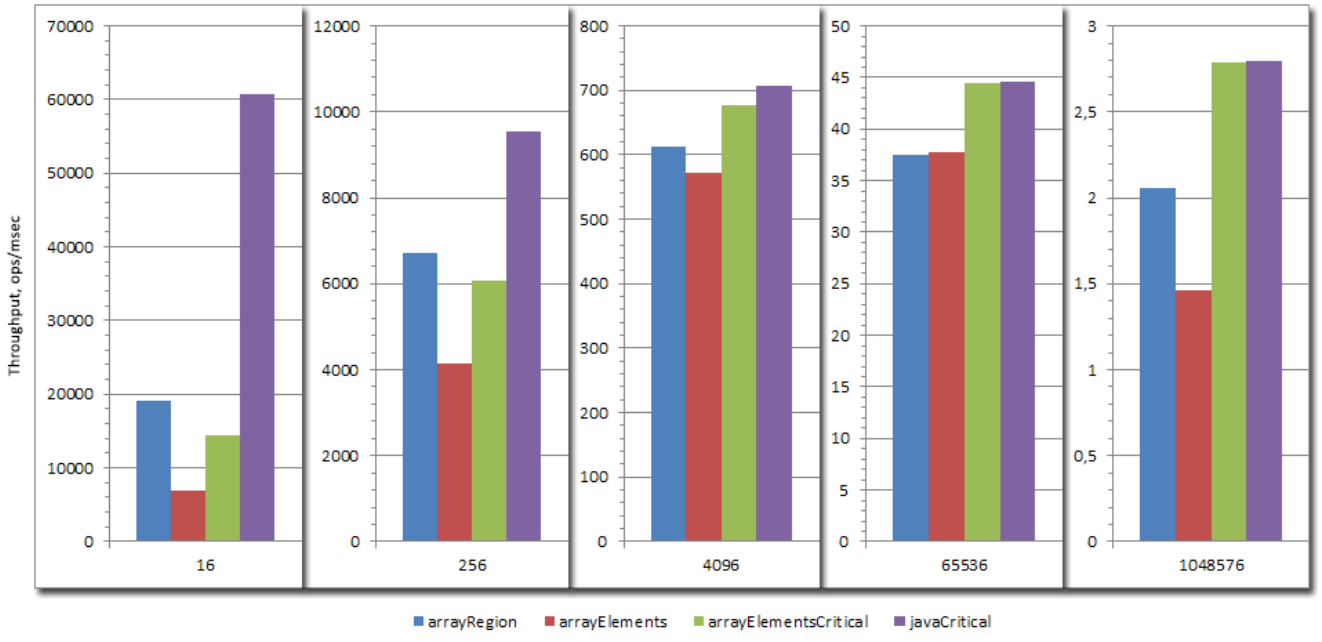
```
JNIEXPORT jint JNICALL
JavaCritical_com_package_MyClass_nativeMethod(jint length, jbyte* buf) {
    return process(buf, length);
}
```

Critical natives are supported only in HotSpot JVM starting from JDK 7. Moreover, "critical" version is called only from compiled code. Therefore you need both critical and standard implementation to make this work correctly.

This feature was designed for internal use in JDK. There is no public specification or something. Probably the only documentation you may find is in the comments to JDK-7013347.

Benchmark

This benchmark shows critical natives can be several times faster than regular JNI methods when the native workload is very small. The longer is method, the smaller is relative overhead.



P.S. There is an ongoing work in JDK to implement Native MethodHandles that will serve as a faster alternative to JNI. However it is unlikely to appear prior to JDK 10.

1. http://cr.openjdk.java.net/~jrose/panama/native-call-primitive.html

edited May 23 '17 at 12:34
Community ♦
1 1

answered Mar 30 '16 at 13:00
apangin
70.5k 9 148 181

How can you know whether your program will use the critical implementation or the standard one? I just tried similar example, where I compiled my java file with `javac` and ran it and it used the standard non-critical version. – Ivaylo Toskov May 5 '17 at 12:26

1 @IvayloToskov Only JIT compiled methods call Critical Natives. While a method is interpreted, it will call standard implementation. – apangin May 5 '17 at 17:09

Should the method declaration be present in both the .c and .h files? – Jaspreet Dec 27 '18 at 7:14

1 @Jaspreet It's not necessary to have it in .h – apangin Dec 27 '18 at 7:54

Will a jbyteArray always be exploded into a pointer and a length value or will it remain an jobject if desired? I prefer to take a jobject then reference the pointers to determine if it is an array or an long value. Is there any source in the JVM you can point do to explain this behavior better? – Johnny V Apr 23 '19 at 14:55

3

It's worth to mention here that another popular opensource JVM has a similar, documented, but not popularized way to speed JNI calls for some native methods.

> Faster native calls to the Java Native Interface (JNI) are available using the **@FastNative** and **@CriticalNative** annotations. These built-in ART runtime optimizations speed up JNI transitions and replace the now deprecated !bang JNI notation. The annotations have no effect on non-native methods and are only available to platform Java Language code on the bootclasspath (no Play Store updates).
>
> The **@FastNative** annotation supports non-static methods. Use this if a method accesses a jobject as a parameter or return value.
>
> The **@CriticalNative** annotation provides an even faster way to run native methods, with the following restrictions:
>
> - Methods must be static—no objects for parameters, return values, or an implicit this.
> - Only primitive types are passed to the native method.
> - The native method does not use the JNIEnv and jclass parameters in its function definition.
> - The method must be registered with RegisterNatives instead of relying on dynamic JNI linking.
>
> The **@FastNative** and **@CriticalNative** annotations disable garbage collection while executing a native method. Do not use with long-running methods, including usually-fast, but generally unbounded, methods.
>
> Pauses to the garbage collection may cause deadlock. Do not acquire locks during a fast native call if the locks haven't been released locally (i.e. before returning to managed code). This does not apply to regular JNI calls since ART considers the executing native code as suspended.
>
> **@FastNative** can improve native method performance up to 3x, and **@CriticalNative** up to 5x.

This doc refers to the now deprecated **!bang** notation that was used to speed up some native calls on Dalvik JVM.

answered Feb 23 '19 at 9:36
Alex Cohn