



Putting the code from the stackoverflow post into godbolt, looks like the loops are not getting vectorized: <https://godbolt.org/z/67TW9GeK4> even with aggressive flags. Another debug step v

**al...@novarumdx.com** <al...@novarumdx.com> [#7](#)

Thanks both!

This is just my hunch though - if `-mcpu=kryo` etc. doesn't bring performance on par with Renderscript, we need to investigate further.

I've added the device specific `-mcpu` to `cmake` for all phones i could get my hands on. Unfortunately it didn't make any change to performance times or at least not more than the stdev of the

it's probably worth checking the generated code for hints as to what build flags you might be using that are causing bad code... (i assume you've tried different optimization levels?)

yes i've tried different `-Ox` with no visible results. I didn't check the generated code but i'm happy to check it - is there a quick way to see this? (sorry never done it before)

What i can tell you that resulted in a tangible gain for one particular smartphone (Kryo - Pixel 2) is that using NDK 22 instead of 20 resulted in a 25% speed up. All other phones maintained th

Vulkan has on-device compilation as well and should perform similarly to Renderscript kernels. That we could speed up `invoke()` functions with RS (compared to the NDK) was merely a nic

I have also implemented a parallelizable edge detection in Vulkan following the (very helpful) renderscript migration sample you provide. The performance between the RS version and the VK

The problem is really when a piece of code is not parallelizable (like this one) and if relies on RS `invoke()` and not kernels with `forEach()`. I've also tried this non parallelizable algorithm in

Again, thank you so much for your help. I'm happy to help in anyway i can in order to get to the best performance possible replacing the deprecated RS.

**al...@novarumdx.com** <al...@novarumdx.com> [#8](#)

Putting the code from the stackoverflow post into godbolt, looks like the loops are not getting vectorized: <https://godbolt.org/z/67TW9GeK4> even with aggressive flags. Another debug step

Oh wow, didn't know about this godbolt tool. Very cool. Does it matter if the target is `armv7-linux-android` instead of `armv8`? Just wondering...

`mcpu=kryo` seems to not vectorize either.

CodeCacheDir doesn't seem to hold anything relevant. In one phone is actually empty, in another there's 4 files that doesn't seem to relate to code generated: `install_server-50cdcce7`, `...`

**al...@novarumdx.com** <al...@novarumdx.com> [#9](#)

Morning,

Is there anything i can do to follow up on this? I think i've run out of options to try to overcome the difference in performance.

I can provide the benchmark app if it helps. Is this the best place to follow up with issues or github android/ndk issues is more appropriate?

I guess, for now, my question is really simple. Is NDK supposed to match RS w/invoke or do i need to start living with the speed perf loss?

Regards

**da...@google.com** <da...@google.com> [#10](#)

Does `-mcpu=kryo` automatically enable neon (we are talking about 32-bit, it seems)? One thing you got from r21 was automatically enabling neon. The vector instructions are not available in

If this is a 32-bit thing, I'm curious how 64-bit performs by comparison, and if that's sufficient to make this a non-issue. Depending on where your users are, 32-bit performance could be eithe

**da...@google.com** <da...@google.com> [#11](#)

Is there anything i can do to follow up on this?

Pirama is the expert, but he's OOO this week and next.

I can provide the benchmark app if it helps.

I can guess that he'd appreciate that a lot though :)

**al...@novarumdx.com** <al...@novarumdx.com> [#12](#)

*Comment has been deleted.*

Message last modified on Jul 15, 2022 08:49PM

**al...@novarumdx.com** <al...@novarumdx.com> [#13](#)

Thanks for the update and your answer.

Does `-mcpu=kryo` automatically enable neon (we are talking about 32-bit, it seems)? One thing you got from r21 was automatically enabling neon. The vector instructions are not available i

If this is a 32-bit thing, I'm curious how 64-bit performs by comparison, and if that's sufficient to make this a non-issue. Depending on where your users are, 32-bit performance could be eitl

al...@novarumdx.com <al...@novarumdx.com> [#14](#)

*Comment has been deleted.*

 **deleted**  
0 B 

al...@novarumdx.com <al...@novarumdx.com> [#15](#)

Hi there!

I've attached an example of this issues. See above. Apologies for the delay - my attention is now back to this performance issue RS vs NDK.

Project should be plug and play, please let me know if not. There is some performance metrics in the end of the run when you launch the app (n=100 runs on image VGA resolution).

Is there something standing out on this example that can lead to have similar performance between both frameworks? Looking forward to hear from you.

al...@novarumdx.com <al...@novarumdx.com> [#16](#)

Morning,

Is there any updates on this?

da...@google.com <da...@google.com> [#17](#)

*Reassigned to pi...@google.com.*

(forgot to assign to pirama after asking him the question)

pi...@google.com <pi...@google.com> [#18](#)

FWIW, the attachment in comment#14 is marked "Restricted+" and is inaccessible to any of us on the team. We'll find the right folks to get that access.


al...@novarumdx.com <al...@novarumdx.com> [#19](#)

My apologies, i thought restricted+ was the way to attach without public access whilst still giving access to team. Uploading again.

Just retested to make sure it was plug and play. You should get something like this (depending on the device) in logcat towards the end.

```
Average times (nTimes = 100):  
RS: 9.833 +/- 1.572 ms  
NDK: 18.020 +/- 0.697 ms
```

Please do let me know if you have any questions. Regards

 **dt.zip**  
1.9 MB [Download](#)

al...@novarumdx.com <al...@novarumdx.com> [#20](#)

Morning @all,

Is there any insight on this or still pending to be looked at?

Thank you!

al...@novarumdx.com <al...@novarumdx.com> [#21](#)

Afternoon,

Just wondering if there is any updates on this. It's been a while we just want some clarification if this is intended or if there is something funny going on with those particular NDK versions.

Regards

sr...@google.com <sr...@google.com> [#22](#)


One suggestion from looking at the C++ code is that you might want to locally cache some non-modifiable values for `w`, `h`, `size`, and `max_val` in DTNDK.cpp, since you're doing global reads values once at the start of the JNI function. You could also change your core JNI function to accept these values as inputs instead of calling an `init` function that sets 4 global variables first.

al...@novarumdx.com <al...@novarumdx.com> [#23](#)

Thanks for your recommendation. I wasn't aware locals are raster (generally) than globals. Unfortunately that does not change anything in terms of performance it seems. I've attached a new version with your proposed changes and the results are "identical":

RS: 8.516 +/- 0.219 ms  
NDK: 17.463 +/- 0.268 ms

It would be really nice to get to the bottom of this because porting (serial) code from RS to NDK is something required now that RenderScript is deprecated. Even though RS true purpose is to Reading around, it seems like it happened in the past that changing NDK versions by itself caused decrease in performance - I lost the links but I'm pretty sure you are aware of that. All we really want at the moment is to know if this is "what it is" or if we can hope for similar performance either by proj setup/config/code or a future NDK version. I think it's only fair to ask

 **dt.zip**  
2.2 MB [Download](#)

**en...@google.com** <en...@google.com> [#24](#)

you might find simpleperf (or Studio's gui cpu profiler) helpful for seeing where the time goes --- <https://developer.android.com/ndk/guides/simpleperf>.

**al...@novarumdx.com** <al...@novarumdx.com> [#25](#)

Thanks.

I already chrono(C++) & profile(java) the code so I know exactly the method that is taking time. I'm not sure what I am going to gain in terms of granularity with those profilers... seems a bit li

**en...@google.com** <en...@google.com> [#26](#)

simpleperf will show you the \*instructions\*...

On Fri, Jan 13, 2023, 09:13 <[buganizer-system@google.com](mailto:buganizer-system@google.com)> wrote:

- [Show quoted text](#) -

**ja...@google.com** <ja...@google.com> [#27](#)

The following compiles on arm64 (required for vminvq\_s32). Not sure if it's any faster through.

```
for (i = 1; i < h-1; i++) {
    for (j = 1; j < w-1; j++) {
        int32_t foo[] = {dt[(i-1)*w+(j-1)]+4, dt[(i-1)*w+j]+3, dt[(i-1)*w+(j+1)]+4, dt[i*w+(j-1)]+3};
        int32x4_t bar = vld1q_s32(foo);

        int32_t smallest = vminvq_s32(bar);
        if (smallest < dt[i*w+j]) dt[i*w+j] = smallest;
    }
}

for (i = h-2; i > 0; i--) {
    for (j = w-2; j > 0; j--) {
        int32_t foo[] = {dt[i*w+(j+1)]+3, dt[(i+1)*w+(j-1)]+4, dt[(i+1)*w+j]+3, dt[(i+1)*w+(j+1)]+4};
        int32x4_t bar = vld1q_s32(foo);

        int32_t smallest = vminvq_s32(bar);
        if (smallest < dt[i*w+j]) dt[i*w+j] = smallest;
    }
}
```

**ja...@google.com** <ja...@google.com> [#28](#)

Using arm\_neon.h does not seem to help performance.

**ja...@google.com** <ja...@google.com> [#29](#)

I've been thinking about this a bit more. My understanding of the algorithm is, just as alopes says, it is not parallelizable. Each pass through the data, I think, depends on the calculations that

**ja...@google.com** <ja...@google.com> [#30](#)

The problem is that the "debug" build variant in Android Studio is compiled with -O0. If you optimize more aggressively, NDK is faster.

It turns out to be a bit tricky to change this. If you do `set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3")`, it gets inserted BEFORE -O0, and so has no effect. Instead, per <https://stackoverflow.com/over/10000000> overrides it.

You can see what flags are being passed by looking at `app/.cxx/cmake/debug/arm64-v8a/compile_commands.json`

Here are the results I got from dt.zip (Comment #23 above) on a Pixel 6 pro, making sure that the phone was awake when running the benchmark so everything ran on a performance core.

With -O0:

- Average RS: 7.85 +/- 2.402 ms
- Average NDK: 10.20 +/- 1.476 ms

With -Os:

- Average RS: 8.06 +/- 2.339 ms
- Average NDK: 3.74 +/- 1.399 ms

With -O2:

- Average RS: 8.49 +/- 4.359 ms
- Average NDK: 3.53 +/- 0.508 ms

With -O2 and the phone asleep, I got:

- Average RS: 26.81 +/- 13.839 ms
- Average NDK: 9.09 +/- 3.646 ms

---

**ja...@google.com** <ja...@google.com> [#31](#)

*Marked as fixed, reassigned to ja...@google.com.*

Out of curiosity, I tried `#include <arm_neon.h>` and my code from Comment #27. It's slower:

- Average RS: 8.52 +/- 3.096 ms
- Average NDK: 5.15 +/- 1.569 ms

I guess compilers are pretty smart. :)

---

**da...@google.com** <da...@google.com> [#32](#)

You don't need to do the thing mentioned in the stackoverflow post, afaict, you just have to build the release variant (Build -> Select build variant, switch to "release" in the window that opens)

---

**ja...@google.com** <ja...@google.com> [#33](#)

When I tried building the release variant, it wanted a signing key that I didn't have.

---

**da...@google.com** <da...@google.com> [#34](#)

Shouldn't be a problem for the app owner, I think. I've seen that before too but I don't really know how or why.

---

**ja...@google.com** <ja...@google.com> [#35](#)

BTW, big thanks to Antonio for providing a good repro case.

---

**al...@novarumdx.com** <al...@novarumdx.com> [#36](#)

Hi everyone.

Thank you so much for having a look at this. I didn't notice the email updates on the issue coming through but i now caught up with them.

I can confirm that forcing the `-O2` with `target_compile_options` works in debug build. Building in release also does this.

This is a significant development for us which makes our Vulkan/NDK code go faster so i thank you all for checking this. It is somehow "hidden away" that your cmake lists options are always

Unfortunately we build our native libs in debug meaning that we will have to change to release builds/with the forced `-O2 target_compile_options`. But that is minor considering the mass

All the best

---

**ja...@google.com** <ja...@google.com> [#37](#)

I agree that it should be easier to debug native builds. It would be nice if the compile commands that CMake uses are more discoverable.

---

**da...@google.com** <da...@google.com> [#38](#)

It is somehow "hidden away" that your cmake lists options are always overridden by debug builds and that might cause confusion to others too. Maybe there should be some kind of warning

I'm not sure what that warning would be? This is how CMake works. Looking at #30 again, I don't think that's what you want either, you need to append to `CMAKE_CXX_FLAGS_DEBUG` (and any

---

**al...@novarumdx.com** <al...@novarumdx.com> [#39](#)

I understand. Fair point.

I guess what i meant is from an Android Studio point of view... to see the flags being used and cmake out in a build window is useful. If that was the case probably it would have been seen earlier

Regards