ECE 449 Honors Option
Attempting to Speed Up Maltab Simulations Using Android Programming

Jacob Honer
College of Electrical and Computer Engineering
Michigan State University
East Lansing, United States of America
honerja1@msu.edu

***Abstract*—This paper will outline the process of converting Matlab files to java to be run on Android in an attempt to speed up simulations through using RenderScript on Android. RenderScript is a framework for running computationally intensive tasks on Android, that utilizes multiple CPU cores or the phones GPU, depending on what process the library deems to be the fastest. RenderScript, therefore, has different performances on different phones, but Google claims that it picks the fastest process possible for each phone. This paper will first explain how the Matlab code was converted to Android and ran using exclusively the Anroid CPU. Next, the computationally intensive parts of each simulation were converted to C and stored in a RenderScript script, where the Android CPU would transfer the required data and where the required computations would be performed. After computing, the Android CPU would then grab the updated data from the script and display the visualized information on the Android app if that process is desired. This paper will breakdown this process and share data from three different Matlab converted scripts to Android. This paper will over an explanation for results and future improvements that could perhaps be made.**

I. Introduction

Matlab code, being written in C and C++, is already quite are fast to begin with, as C is known by many to be one of the most efficient programming languages. Despite being generally very fast, Matlab code alone is limited to a computers CPU. This means that all computations in a regular Matlab script are done in series. Running complex computations in parallel many times drastically speeds up performance. Traditionally, as far as Maltab is concerned, people will use Cuda to speed up computations. Cuda is a parallel computing platform that utilizes a computers GPU to run computations. Traditionally, people will create Cuda script, pass data to the Cuda script from Matlab, perform the computations, and then collect the data from the Cuda script. Obviously transferring the data to Cuda takes time, but if the parallel computation time on the GPU is significantly faster the CPU, using Cuda can lead to some very impressive speed up times.

Here, we wanted to consider if it made sense to use external computations. Lets say someone had a computer that didn't have a GPU, but still wanted to speed up Matlab computations. Lets say that this person also purchased a high quality Android phone. We wanted to explore if it would make any sense for that person to try to use the computation ability of the Android phone for speeding up their Matlab code. This paper will not explore in particular how that interfacing works, but we will explore some basic comparisons of Matlab code speeds to the computations speeds achievable on Android. We decided to use RenderScript to try to speed up computations. RenderScript works extremely similarly to Cuda, but is an Android library that utilizes the multiple CPU cores or GPU of an Android device for faster computations.

In addition to simply speeding up computations on Android, we also wanted to explore the feasibility of running simulations on Android apps. This could be a good learning tool, as students could download an app that shows a bunch of different simulations to their Android phone, and using that app better understand a concept that is being explored in class. Rather than every student having to have a computer with Matlab, if it is feasible to run simulations on Android, students can simply use their phones. Furthermore, Android is free to use for all, so the hastle of making sure each students has a Matlab license could be avoided. With the new Windows update allowing for Android apps to be ran locally on Windows, the scope of Android apps and their use in education has perhaps turned a corner! It is quite possible that we will see Android be a bigger and bigger part of education moving forward, especially if the Android OS could handle the rigorous computations that many computers handle with ease. Through this project, we wanted to explore this idea.

II. Method

A. Converting code to java

The first thing to be performed was to convert the Matlab code to java so that it could be run on the Android CPU. Overall, this was a rather straight forward process. All static variables were simply copied over to the java code and updated to match the java initialization criteria. The initialization of larger arrays was a less straight forward process, as some of the short cuts used in Matlab did not exist in java, but with some basic coding background, it was easy to make sure that the initialization all matched. The Android debugger

```matlab
% second time step
it = 2;
for ix = 2:nx - 1
    for iy = 2:ny - 1
        p(ix, iy, it) = p(ix, iy, it - 1) ...
            + c^2 * dt^2 / 2 / dx^2 * (p(ix + 1, iy, it - 1) - 2 * p(ix, iy, it - 1) + p(ix - 1, iy, it - 1)) ...
            + c^2 * dt^2 / 2 / dy^2 * (p(ix, iy + 1, it - 1) - 2 * p(ix, iy, it - 1) + p(ix, iy - 1, it - 1));
    end
end
```

Fig. 1. Matlab for loop for second time step of computations in linesource.m file

```java
pow = Math.pow(c, 2) * Math.pow(dt, 2) / Math.pow(dx, 2);
delta = pow / 2;
for (int ix = 1; ix < nx - 2; ix++) {
    for (iy = 1; iy < ny - 2; iy++) {
        double temp1 = delta * (p_current[ix+1][iy] - 2* p_current[ix][iy] + p_current[ix-1][iy]);
        double temp2 = delta * (p_current[ix][iy+1] - 2* p_current[ix][iy] + p_current[ix][iy-1]);
        p[it - 1][ix][iy] = p_current[ix][iy] + temp1 + temp2;
    }
}
```

Fig. 2. Java for loop for second time step of computations in linesource.m file

was also very helpful in making sure that the type, size, and initialization information of each variable was correct before any computations were performed.

The biggest thing to keep in mind when converting the code was the difference between indexing arrays in Matlab vs in java. In Matlab, array indexing starts with 1, where as it start in 0 for java. This became particularly important where for loops are concerned. Caution was taken to make sure that the java computations precisely matched the Matlab computations. An example of two equivalent codes, but one in Matlab and one in java can be seen in Fig. 1 and Fig. 2. Notice the subtle differences in syntax of the loop.

One of the problems that was quickly discovered was that the Android app didn't have enough allocated CPU storage to run the simulation. Each Android app can use a maximum of 500 MB, regardless of how much RAM a phone has. This means that all variables and processes must stay below this 500 MB limit or the app will crash. Interestingly, RenderScript is exempt from this requirement, but in our case, where we transfer all computations back to the CPU and the RenderScript script does not initialize new variables, this is simply something to know. If no visualization was required, this might have been helpful.

Upon inspection of the Matlab code, it was found that the Matlab code basically used a 3D array to record all computations that happen in the scripts lifetime. Each 2D grid was saved for each time-step. For example, in linesource.m, each grid was 401x401 doubles, and all 1001 timesteps were saved, so the total array in Matlab help 401x401x1001 doubles. This amounts to 1.29 GB size for this single array, which far exceeds the 500 MB max that Android offers.

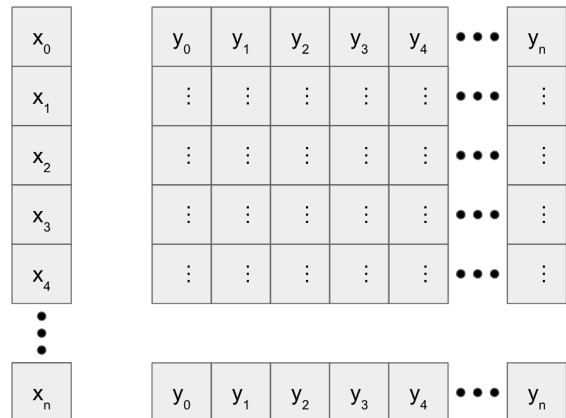

Fig. 3. Visualization of how RenderScript allocations were initialized

I quickly realized that it was unnecessary to save all this data, as the visualization took place in real-time and the computations only used data from the previous two time-steps. This meant that I was able to modify the code to only store three 2D arrays, meaning that array that was 1.29 GB in size was now only 3.68 MB, which was much more reasonable for an Android app. This change did not alter the computations at all. The only thing that it did mean was that the code had to basically shift the array after each update, so that a the array for two time-steps back was properly set, as well as the array for one time-step back, and then the array for the current time-step was reinitialized. Properly setting or initializing the

```java
void setupRS(){
    mRender = RenderScript.create(MainActivity.this);

    script = new ScriptC_compute(mRender);

    script.set_ny((int) ny);
    script.set_nx((int) nx);
    script.set_c(c);
    script.set_dx(dx);
    script.set_dy(dy);
    script.set_dt(dt);

    outputPrev = Allocation.createSized(mRender, Element.ALLOCATION(mRender), (int)nx);
    outputArrayPrev = new Allocation[(int)ny];
    for (int i = 0; i < nx; i++){
        outputArrayPrev[i] = Allocation.createSized(mRender, Element.F64(mRender), (int)ny);
        outputArrayPrev[i].copyFrom(p_current[i]);
    }
    outputPrev.copyFrom(outputArrayPrev);
    script.set_p_past(outputPrev);

    outputCurrent = Allocation.createSized(mRender, Element.ALLOCATION(mRender), (int)nx);
    outputArrayCurrent = new Allocation[(int)ny];
    for (int i = 0; i < nx; i++){
        outputArrayCurrent[i] = Allocation.createSized(mRender, Element.F64(mRender), (int)ny);
        outputArrayCurrent[i].copyFrom(outputCPU[i]);
    }
    outputCurrent.copyFrom(outputArrayCurrent);
    script.set_p_current(outputCurrent);
}
```

Fig. 4. RenderScript initialization in vibratingmembrane simulation

arrays might add minimally to the duration of each update, but this should be negligible and allowed for the code to run on Android. This method is also far less wasteful with respect to the devices memory. In addition, this would allow for the 2D arrays to be significantly larger, having a much better resolution for the simulations, on both Android and Matlab.

After this process, the code was able to run on the java CPU with no difficulties!

B. *Converting computations to RenderScript*

Converting the computations to RenderScript was definitely the most difficult part of this project. Like many other parallel computing libraries or methods, RenderScript does not generate very specific errors, so debugging can be particularly difficult.

We first had to handle the initialization process of RenderScript. We first create a RenderScript object in java. We next created the script. We set all of the variable to the script, basically passing all constant values in the java code used in computations to be constants in the script object. Lastly, we had to create all of our RenderScript allocations for our arrays. What we did was basically create an allocation of allocations. All y-direction indices will be passed to an allocation that acts essentially as a 1D array, and then each of these 1D array allocations will be passed to their corresponding index of another 1D array of allocations, serving as x-direction indices. A graphic demonstrating how this works out visually can be seen in Fig. 3. After these initialization, the java variables were passed, which has to happen after each update as well, then these allocations were passed to the script. How these allocations were initialized gives some insight into how Renderscipt attempts to parallelize the computations. RenderScript only works with 1D arrays. In this configuration, each y-direction array runs its computations in parallel. So, for linesource.m, where our grid is 401x401 in size, Renderscipt computes 401 different scripts, where each scripts does 401 computations in parallel. Perhaps converting the 401x401 array to a 1x(401*401) array would be quicker, but this would

require a more complex indexing process in the computations, would require converting the array back and forth, which takes time, and it is unlikely that it would speed up computations any further with all things considered.

The initialization process was organized in a single method, and each Matlab converted script follows a similar process. This process is shown in Fig. 4. This figure in particular may be a good reference if this work is to be replicated.

```
output = getReinitializedOutput((int)nx, (int)ny);

//compute script (parallel process)
script.forEach_compute(output);
mRender.finish();

outputPrev.destroy();

for (Allocation allocation : outputArrayPrev) {
    allocation.destroy();
}

outputArrayPrev = outputArrayCurrent;

outputPrev = outputCurrent;
outputCurrent = output;

outputArrayCurrent = outputArrayMem;

script.set_p_past(outputPrev);
script.set_p_current(outputCurrent);

it++;
```

Fig. 5. Visualization of how RenderScript allocations were initialized

The RenderScript updating process will now be outlined. We first initialize a new 2D grid in the form of allocations of allocations, as outlined above, where the returned response from RenderScript will be stored. Next, we perform our computation and save its responses to these initialized allocations of allocations. We destroy the variable that held the data from two time-steps back, essentially freeing that memory, and then we shift our arrays so that what was the array from one time-step back becomes the array from two time-steps back, the array that was the current data becomes the array for one time-step back, and then the computed responses from the script become the current data. Lastly, we re-pass the array from one time-step back and two time-steps back to the RenderScript object, and iterate our time variable. Although it is not fully necessary to include the code in this report, it has been included in Fig. 5, again, for reference if this work is ever to be replicated.

Next, we will discuss how the RenderScript script performs computations. All in all, this is probably the most straight forward process of the RenderScript implementation. All of the values that are needed are already passed to the script. We simply iterate through the arrays and perform the computations, then save them to our output array. The one thing to note is that you can't simply access values within arrays as one would normally. You have to use rsGetElementAt() to get the desired index of an array, and it is a good practice to specify the variable type that you are calling (i.e. rsGetElementAt_double()).

Fig. 6. Visualization of simulations on Android (Left to right: linesource, vibratingmembrane, and centered2Dgaussian

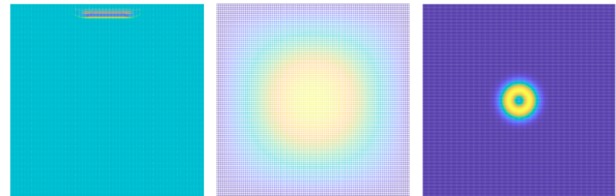

Fig. 7. Visualization of simulations on Matlab for comparison (Left to right: linesource, vibratingmembrane, and centered2Dgaussian

C. *Visualizing on Android*

The next challenge that was faced in converting the Matlab code was how to handle visualization. Matlab has many convenient graphing tools, and is arguably the best software to exist for 3D graphing. Android, unfortunately, has no equivalent libraries or way to visualize 3D graphs, hence, I had to get creative on how I was to visualize these simulations. I settled on a rather straight forward visualization. What I did was convert the x and y grid to a bitmap of equal dimension. For the z component, I took the value in the array corresponding to each x and y index, which corresponded with the z value of the graph, and I scaled that value by 255. If that value was positive, I plotted it on the bitmap as red, with the intensity corresponding with the scaled value. Likewise, if the value was negative, I plotted it on the bitmap as blue, with again, the intensity corresponding with the scaled value. This meant that all positive z values appeared as red and negative as blue on a bitmap that took on a birds-eye view of the 3D graph. This provided a clear visualization of what was going on in the simulation through a simple, computationally minimal, process. This visualization can be seen in Fig. 6. For comparisons sake, the birds eye views of the same simulations ran in Matlab can be seen in Fig. 7. As we can see, the two are very comparable! With a little bit more work, the Android simulation can definitely approach the clarity of the Matlab simulations, but due to time constraints and the fact that this project didn't focus on visualization, the visualization was deemed good enough.

	Average Time of Execution (seconds)		
Script	Matlab	Android CPU	RenderScript
linesource	3.90	2.54	60.06
vibratingmemebrane	0.48	2.16	14.74
centered2Dgaussian	1.95	3.95	24.42

Fig. 8. Total execution time for Matlab, Android CPU, and RenderScript

D. Results

This section will outline the results and takeaways from this project. The time of execution for the Matlab script, Android CPU, and Android RenderScript can be found in Fig. 8. All collected data omitted visualization time, as the Matlab visualization took longer than the Android visualization, but it was also much more comprehensive; the two aren't very comparable, so we will not compare them in this paper. The Matlab scripts were ran on a 2015 MacBook Pro with a 2.7 GHz Dual-Core Intel Core i5 Processor and 8 GB 1867 MHz DDR3 Memory. The Android phone used was the Sony Experia XZ Premium, model G8142, equipped with 8 CPU cores and 3.728 GB available RAM.

As we can see, the Android CPU has a generally comparable performance to the Matlab scripts. This makes sense, as the Matlab CPU frequency is 2.7 GHz and the Sony phones CPU frequency is 300 MHz - 2.458 GHz. In general, the Matlab performed quicker than the Android CPU, and this can probably be explained by the quicker CPU and more RAM. The discrepancies between the MacBook Pro's OS and Android's OS are also likely to blame to some extent. One case where the Android out performed the Matlab was for the linesource simulation. This is perhaps because the Matlab code was using most of the CPU, as this was the highest resolution simulation ran. This perhaps slowed down the script. The Android code used the more efficient array logic, however, outlined in Section A, keeping the RAM relatively available. Given my experience, this is often something that slows down computations, as it takes time to free previously used by no longer relevant memory.

Unfortunately, the RenderScript did not improve performance at all. The real reason for this is that it simply took too much time to pass the data to the RenderScript script and then collect that data after the computation was performed. This is many times the gridlock of speeding up a program using parallel programming, and I believe this to be the case here.

E. Discussion

RenderScripts failure to speed up computations was disappointing, but not entirely unexpected. Computers tend to out perform mobile phones in all areas of computation and speed, simply because the two devices have different motivations for their existence. Not many people need their phones to perform massive computations, so phones usually don't prioritize speed and many don't have a GPU. Furthermore, with phones always being connected to the internet, many find it more logical to simply use cloud computing or other methods to perform complex computations and then share that data with phones; not many people wish to perform these computations locally.

The biggest pitfall here appears to be that passing variables to RenderScript is extremely slow, especially compared to the simplicity of the computations used in this paper. It would be ideal to keep all variables in RenderScript and not have to pass them back and forth. In this case, passing variables back and forth was required in order to visualize what was going on, but future work might explore how well RenderScript performs if variables stay local to the script with no visualization. Future work can also perhaps sample every 10 or so frames, so that data transfer back and forth is not every update, but every few updates. This would hinder the visualization smoothness, but the user would still be able to get a general idea of what is going on in the simulation and it should improve RenderScript's computation time. Future work should also explore having the entire script work with exclusively 1D arrays, as this would allow Renderscipt to be entirely free in its parallelization and not constrained by the users 2D bounds. This would add to the complexity of the script and to the complexity of the code in general, but might make it faster.

Other work can look into how to speed up Matlab scripts with Android perhaps enhancing computation speed. It is possible to pass data between and Android phone and a computer in Matlab using Bluetooth. Is there a way to utilize this communication to improve the script's speed? This is a primitive idea that can perhaps be explored in the future.

All things considered, this paper did not explicitly show that RenderScript speeds up computation, but there are many sources out there that indicate that it can. Given my experience, using RenderScript in machine-learning applications sped up computations by over 8 times. This was not demonstrated here, most likely due to the limitations outlined above. For context, our script performed around twelve arithmetic processes; the machine-learning applications that I am familiar with that use RenderScript performed well into the thousands of arithmetic processes per script.

This paper does show that Android could be a powerful tool in education! The simulations were comparably fast, and on Android, they would be much more portable and accessible to students around the world. Although it takes a little bit more work to generate an Android app, educators should maybe consider incorporating Android into their demos and curriculum, as it is proven to be fast, portable, accessible, easy to understand and use, and can incorporate interactive activities that take advantage of touch and sound easier than a computer.