AI APPLICATION COMPONENT DEVELOPMENT FOR THE STUDY OF TRADEOFF BETWEEN USER-PERCEIVED PERFORMANCE AND AI MODEL INFERENCE THROUGHPUT IN MOBILE AUGMENTED REALITY APPS

Research Report

Joseph Robert Bommarito

Advisors:
Dr. Marco Brocanelli
Nilofar Didar

Wayne State University
College of Engineering,
Detroit, Michigan
# Table of Contents

Abstract ...................................................................................................................................................... 4  
Introduction ............................................................................................................................................... 5  
Application outline ................................................................................................................................... 6  
  ImageClassifier ....................................................................................................................................... 6  
  Kotlin Coroutines ................................................................................................................................... 6  
  Flow ......................................................................................................................................................... 7  
  Producer - DynamicBitmapSource ......................................................................................................... 7  
  Intermediary - BitmapUpdaterApi ........................................................................................................ 7  
  Consumer - BitmapCollector ................................................................................................................ 8  
  AiRecyclerViewAdapter ...................................................................................................................... 9  
  Item View Model .................................................................................................................................... 10  
  MainActivity ......................................................................................................................................... 11  
Using The App .......................................................................................................................................... 13  
Experiments ............................................................................................................................................. 14  
  Experiment: Throughput vs Number of AI Task Inception V4 .............................................................. 15  
  Experiment: Throughput vs Number of AI Task MobileNet V2 .......................................................... 16  
  Experiment: Throughput vs Total Triangle Count ............................................................................ 17  
  Experiment Results .............................................................................................................................. 18  
  Limitations ........................................................................................................................................... 19  
Conclusion ............................................................................................................................................... 20  
References ............................................................................................................................................... 21  
Appendix .................................................................................................................................................. 22  

**Abstract**

The purpose of this research was to develop an application that is able to run multiple AI inferences on multiple devices (CPU, GPU, NPU) at the same time, while simultaneously performing AR tasks. The team had an application that could run one model on one device and infer on a static bitmap, it was also my goal to run the AI on the same camera frames that the AR operates on in real time. The added sections of the Android application were written in Kotlin and Java. Adding Kotlin to the project and taking advantage of its coroutines allowed for quick and reliable development to solve a large part of the problem at hand. The results of this effort produced an application that is able to do what we set out for it to do. There is still optimizations that could be added for easier experimenting and therefor faster and more robust data collection, but the app performs quite well for basic experiments to study the effect on resources and user perception while using AR and AI.
Introduction

As society progresses there is a major shift toward reliance on mobile devices reliant on batteries, such as smartphones. While battery technology is improving, the computations on mobile devices are increasing in both quantity and complexity. Augmented reality (AR) and artificial intelligence (AI) are technologies that are both finding homes in modern mobile device applications, these two technologies also often share processing time on the GPU. To save in computational complexity the quality of AR models can be reduced by decreasing the amount of faces that compose the meshes rendered to the screen, at some reduction of quality will have a subjective impact on the user-perceived performance. With the resources freed from the mesh decimation, there should be an impact on the throughput of AI inferences.

Along with executing AR tasks, the ideal application would also execute AI inferences on the same camera image being captured for the AR. In order to study the throughput of the models, the application would be required to execute AI inferences from multiple different models on various hardware selected by the user such as the central processing unit (CPU), graphics processing unit (GPU) and neural processing unit (NPU). While there are many solutions available to execute AI model inferences on mobile devices, there were no applications that allowed to process multiple AI tasks at the same time, let alone on different devices.

In order to perform the experiments required to study the trade-off between user-perceived performance and AI Model Inference throughput in Mobile AR applications, an application would need to be developed that could execute both AI and AR tasks synchronously. The following outlines the AI portion of this application and how the data flows to produce AI inferences in parallel.
Application outline

ImageClassifier

ImageClassifier(Activity activity) throws IOException {...}

An ImageClassifier abstract class contains the API to use the TensorFlow light models, this parent class was provided by tensor flow light in the Image Classification demo. In the new demo, there is an ImageClassifier extended class for each TensorFlow light AI model available. (see Appendix for instructions for how to add more image classifiers to the application). Some modifications were made such as adding variables to track which device the model was running on and how many threads the model is using for the inference. In the original demo, the classifyFrame function showed the top three classifications with accuracy percentages on the screen; this function was overloaded to no longer provide information on the classifications themselves since the output itself does not have significance on the experiment.

Kotlin Coroutines

Passing the camera frame to multiple image classifiers is a producer-consumer relationship. The app was able to skip considering the AR components as a consumer, this will be explained later. There are many, well documented, ways to handle the producer-consumer problem, the chosen solution was using cooperative routines (Coroutines) provided in the Kotlin language. Coroutines were chosen over “classical” multi-threading for quicker and more reliable development. Threading and coroutines differ in that threading uses preemptive or non-cooperative multitasking, and coroutines use cooperative multitasking. In non-cooperative multitasking, the operating system initiates context switches between processes, whereas in cooperative multitasking, the processes voluntarily yield to other processes and
share the same scheduling schema [1]. Coroutines execute asynchronously, when threading tasks can execute synchronously in parallel; coroutines begin execution concurrently but will process in parallel.

Flow

In Kotlin, the data type Flow is used to emit multiple values sequentially. Flows are built on top of coroutines and allow for consumers to collect the data asynchronously off of the main thread [2]. A flow can be modified by an intermediary before it gets to a consumer. The app has one producer, many consumers, and an intermediary that holds the latest camera frame that will be used for the producer to update the flow.

Producer - DynamicBitmapSource

class DynamicBitmapSource(private val bitmapUpdaterApi: BitmapUpdaterApi){...}

In the application the producer’s job is to pass a bitmap representation of the latest data captured by the camera to the consumer. This producer is defined in the DynamicBitmapSource class (a class BitmapSource is also included in the demo to pass a static bitmap for bench-marking and debugging purposes), it has two variables, a boolean run and bitmapStream which contains the Flow of Bitmap type. There are three functions to start the stream, pause the stream, and to run the stream if the variable run is true. The components of the flow are straight forward. The flow emits the value stored in the intermediary and runs on the default dispatcher thread pool, which gives access to the same amount of threads as there are cores.

Intermediary - BitmapUpdaterApi

class BitmapUpdaterApi {}

Between the producer and consumer is the intermediary, BitmapUpdaterApi is the class that handles the data between the point at which the flow starts and when the data is emitted. The
intermediary is very simple, it contains a variable that holds the latest bitmap. Thanks to the way Kotlin is structured, setters and getters are automatically generated in the java code it compiles to. This allows for quick development of simple and reliable code.

**Consumer - BitmapCollector**

class BitmapCollector(private val bitmapSource: DynamicBitmapSource?,

Finally, the consumers are instantiated with BitmapCollector objects. The consumer is doing the most work in the relationship both with handling the asynchronous tasks and producing data for experiments. The consumer collects the value emitted by the producer in a suspend function. The suspend keyword allows the various jobs in the coroutine to cooperatively multitask.

The class starts with a constructor that includes a bitmap source (producer), an image classifier (both nullable\(^1\)), the index at which the consumer is stored in an array in the main activity, and an activity (main) itself for reference to file storage. When running experiments, the image classifier has to be chosen in the constructor of BitmapCollector object, if the image classifier is changed, the BitmapCollector should be destroyed and a new one should be made.

A background job is created in the view model scope and launched in the dispatchers IO thread pool (The IO thread pool was chosen because it has access to 64 threads). As long as there is a value emitted by the producer, the job will collect it, if there is no value then it will collect when there is until the job is canceled. The job then scales the bitmap to the size required by the image classifier, calls the classifier to classify the frame, and produces output consisting of overhead (the time from the end of the previous classification to the start of the current), classification time (the time it takes the model to classify the frame), and other relevant data.

---

\(^1\) While these two variables are nullable, the coroutine and classification will fail if the variables are null. They are nullable to initialize a consumer before the experiment is ready to start.
classify the frame), and response time (the sum of the overhead and classification time). The amount of
times classified and the sum of all of the response times will be stored to later calculate the AI model’s
throughput.

**AiRecyclerViewAdapter**

class AiRecyclerViewAdapter(var mList: MutableList<AiItemsViewModel>,
val streamSource: DynamicBitmapSource, val activity: Activity) :
RecyclerView.Adapter<AiRecyclerViewAdapter.ViewHolder>() {...}

The android RecyclerView is similar to the ListView, it allows the developer to create a
dynamic scroll-able user interface widget that is populated by a list of items. To populate the view with
data an adapter is required, “An Adapter object acts as a bridge between an AdapterView and the
underlying data for that view”[3]. The application uses a recycler view to both display info about the
active models\(^2\), and allow the user to change settings such as on which device the model runs (CPU,
GPU, NNAPI), the amount of threads used to perform an inference, and the actual model itself. The
way data is displayed to the user about the AI models and the functions to respond to user input are
defined in the AiRecyclerViewAdapter.

When the recycler view is created, the layout file ai_settings_card_view_design.xml is inflated
to the recycler view UI object. As each tile is loaded to be rendered (the view is bound) each object in
a list of AiItemsViewModel objects populates a card in its matching index and setting the default
selections for each of the objects. The card view elements also lists, they contain a thread quantity,
selected model, and selected device. The bind also sets the on change listener for each of these to
modify the bitmapCollector information passed to the recycler view from the Main Activity. The
change listeners call the updateActiveModel function, which performs the following:

\(^2\) Model and Image Classifier are often used synchronously because in this demo the only AI models implemented are
image classifiers. More types of models will allow for more experiments.
• Stop the producer from emitting the stream, stop the consumers from collecting, and switch the UI element that starts the AI tasks off.
• Gather the current card information, if the card information matches the user request, return.
• Set the info in the AiItemsViewModel to the user selected information, and close the classifier\(^3\).
• Get the string representations of the user selected settings
• Load the new classifier with the selected device and threads settings
• Display the string of the user selected settings
• Set the collector in the AiItemsViewModel to a new collector using the updated classifier.

**Item View Model**

class AiItemsViewModel {...}

While also a simple class, AiItemsViewModel is very important to the user interaction of the application and aiding in creating clearly defined objects to perform multi-threaded collections. The object contains a list of devices and models\(^4\), a BitmapCollector, an ImageClassifier\(^5\), the currently selected device, model, and amount of threads. This is the object that will be used in the main activity to store the information about the consumers.

---

\(^3\) This is separate from pausing the collector. The AI ImageClassifier object needs to be closed before setting a new classifier or changing its settings.

\(^4\) These are constants, they are the same for each card view, the display the text to the user, they show which devices and models the user can select to run AI tasks on

\(^5\) AiItemsViewModel contains a BitmapCollector and an ImageClassifier. BitmapCollector already has an ImageClassifier! The ImageClassifier in BitmapCollector is private and final, the collector gets destroyed and rebuilt when a new classifier is chosen. While this redundancy can likely be removed with some refactoring, it was useful for keeping the collector simple.
public class MainActivity extends AppCompatActivity implements AdapterView.OnItemSelectedListener {

    All of these components come together in the main activity. The BitmapUpdaterApi and ArrayList of AiItemsViewModels are given global scope. Over 90% code relating to AI in the main activity happen in the onCreate function, the flow of these components follows:

    • Initialize the DynamicBitmapSource
    • Add an AiViewItemsModel to the list global
    • Initialize the recycler view and the adapter
    • Set the recycler view’s adapter and layout manager
    • Initialize buttons and click listeners for pushing and popping AI items and text to display the quantity
      ◦ Push/Pop:
        ▪ Increment/Decrement the displayed number of models
        ▪ Stop the stream
        ▪ Add/remove an item to/from the end of the AI list
        ▪ update the adapter
        ▪ Reset the recycler view adapter
    • Initialize and set click listener for the switch to start and stop the stream
      ◦ Off → On
        ▪ If all classifiers are non-null, start the producer. Else, prompt user to set the classifiers
        ▪ Start the collectors
      ◦ On → Off
        ▪ Stop stream
        ▪ Stop collectors

    Returning to the earlier explanation of the producer-consumer problem in this application, it was stated that the AR activity is not a consumer, why? Since the AR portion heavily relies on the ARCore[4] library, the handling of the camera is well defined. It is not worth the risk of breaking the
highly complex AR components to multi-thread the camera. After digesting the flow of the AR components the object, Frame, was identified. Frame is provided by ARCore and has access to the data in the camera. Since AR has to constantly update the fragment using this camera data, it was clear that there was an entry point to access this frame. The function updateTracking was identified as a frequent (constantly repeated) caller of the Frame class. Using an open source library, yuv2buf[5], it was fairly low cost to get the reference to the Android Image from the frame, convert it to a bitmap with the library, and pass it to the BitmapUpdaterApi. With this in place all of the AI functionality was complete.
Using The App

1. Fragment window: This is where the camera image is displayed. The ATV in the image is not actually in my home. It is an AR Model!

2. Button to Toggle the AI Settings Recycler View [5]

3. Buttons to increment and decrement amount of AI models. The Number shows how many AI models are currently available

4. The switch to turn the Classification stream on and off

5. The AI settings recycler view. This is currently displaying one card. If the number in [3] is greater than 1, this can be swiped left and right to display the other models’ info.

6. Text info for the currently selected model

7. Number picker for the amount of threads

8. List to select the Model. The current model is MobileNet V2 1.0 Quantized 224

9. List to select the device to run the inference on. The current device is NNAPI (NPU)

Figure 1: Screenshot of demo running AR and AI
Experiments

The app was used to produce results for a few simple experiments. These experiments will help benchmark the performance of some AI models on various devices, and help understand the relationship between the variables tested. The majority of results are not surprising; this is a good thing. Since the app produces expected results, it means it is performing as expected and we can be confident in the results it produces when testing less than certain hypothesis.
Experiment: Throughput vs Number of AI Task Inception V4

Description:

This experiment tests the throughput when running the Inception V4 quantized and floating point models on various devices. The planned amount of models for each trial was 1-10, 15, and 20. The floating point version of this model is very heavy on the mobile device used, and throughput reduced very early into the experiment. The floating point trials (2 & 3) test less models per device.

Set up:

For each test, select the correct model, select the chosen device. Run the AI stream for more than 30 seconds, stop the stream, increment the model to the next amount required, repeat the process. The data writes to the app’s “data” directory. The CSVs contain the average throughput for each second the model executed. These averages are are then averaged to show the total average throughput for each instance.

Results:

Figure 2: Trial 1: Inception V4 Quantized NPU

Figure 3: Trial 2: Inception V4 Floating Point NPU

Figure 4: Trial 3: Inception V4 Floating Point GPU

There is an inverse relationship between throughput and the number of models. As more models are executed, the throughput rapidly declines. The throughput trends to 0.
**Experiment: Throughput vs Number of AI Task MobileNet V2**

This experiment is identical to the experiment 1, but instead the models tested are MobileNet V2 Quantized and Floating Point.

![Figure 5: Trial 1: MobileNet V2 Quantized NPU](image1)

![Figure 6: Trial 2: MobileNet Floating Point NPU](image2)

![Figure 7: MobileNet V2 Floating Point GPU](image3)

**Results**

The curves are not as smooth as the previous experiment, this is likely due to how much lighter the models are or the device getting too hot and affecting the performance. Like the previous, there is an inverse relation between the throughput and the number of models. The throughput trends to 0.
**Experiment: Throughput vs Total Triangle Count**

The goal of this research was to add a component to the application that allowed for more robust experiments to study the relationship between AR and AI and how executing both technologies has an effect on user perceived performance and AI throughput. This experiment is an example of a study between these two technologies. We ran 5 MobileNet floating point models on the GPU while periodically adding AR models to the screen to increase the triangle count. This experiment was performed over two trials, once to ~3,000,000 triangles, and again to ~500,000 triangles.

**Set up:**

Add 5 Mobilenet floating point models on the GPU, start the AI stream. Using the GPU_usage.csv file, to monitor the amount of triangles, slowly add groups of models (depending on the size) to the screen. Pause (do nothing) for a period of time while the application reports the average throughput to the data. Add more models and repeat. The goal should be to add 10,000-80,000 triangles for each group.

To post process this data, compare the amount of triangles during the pauses with the average of all the throughputs reported during those pauses.
Results

The first trial showed a large concentration and a similar slope as the previous experiments from 0-500,000 models. From 500,000 – 3,000,000, the slope becomes more linear downward trending toward zero. The GPU utilization was already at 99% before the 500,000 mark, so it is interesting to see that the throughput still degrades slowly with more and more triangles added.

The second trial shows holds a similar slope as the first 2 experiments, there is an inverse relationship between the amount of triangles and the effect on AI throughput.

Experiment Results

These experiments show results that were expected for the relationships between throughput vs number of models and throughput vs number of triangles. The application can now be used to test experiments that may have less certain hypothesis.


**Limitations**

The device that these experiments were performed on is a Samsung Galaxy S10, this device performed well under the conditions. Due to the compact nature of mobile devices, overheating poses an issue. After running many experiments it was clear that when the experiment was approaching the end, the heat generated within the device was a cause for less consistent data. Given the mobile nature of the problem, studying the effect on heat and it’s relationship with throughput might be useful, but the larger research problem applies to devices beyond smartphones, such as cars, raspberrypis, laptops, and much more that have a much greater ability to manage temperature.
**Conclusion**

To aid in the study of trade-off between user-perceived performance and AI model inference throughput in mobile AR applications, there needed to be a way to execute multiple AI model inferences on various devices while at the same time performing AR tasks. As a baseline, the app needed to be able to show the inverse relationship between the amount of AI models and the average throughput, and the inverse relationship between AR triangle count and AI model average throughput. Using the demo the team was already working on, the modules were created to perform these tasks. Using Kotlin’s coroutines and carefully selecting where to access the camera frame, the app successfully ran multiple AR models on whichever devices the user selected. The results of development provided an application that could run simple experiments, while providing data that met expectations. There is much more experimenting and testing to be done using AI on multiple devices and various AR tasks, and with this contribution the research is one step closer to conclusion.
References


Appendix

Adding Image Classifiers to the app

Download the zipped file from the tflite&pb link for any of the models to add.

Extract the zip

Move the .tflite model to the app’s asset directory. If the zip included a label file move this as well and take note of the name.

Copy and paste an existing ImageClassifier extended class, for new floating point models its easiest to copy a class built for an existing floating point model, and for quantized its the same.

Change the model name and path in the functions getModelName and getModelPath to match the model. The path should be the the path from the asset directory. If the pasted file in the directory is “name.tflite”, the model name is “name” and the path is “name.tflite”.

```java
@Override
protected String getModelName() {
    // you can download this file from
    // see build.gradle for where to obtain this file. It should be auto
    // downloaded into assets.
    return "mobilenet_v1_1.0_224_quant";
}

@Override
protected String getModelPath() {
    // you can download this file from
    // see build.gradle for where to obtain this file. It should be auto
    // downloaded into assets.
    return "mobilenet_v1_1.0_224_quant.tflite";
}
```

Change the label path in the getLabelPath function if one was included with the model. Not all zips include labels, if the downloaded model was a mobilenet, use the same label that the other mobilenet models use.
Change the image size by updating the `getImageSizeX` and `getImageSizeY` functions. The image size can normally be found in the model name ("name_v1_299.tflite" means the image is 299 x 299 pixels) or in the info included in the zip.

Finally, update the return value for `getNumBytesPerChannel`. Return 4 for floating point models and return 1 for quantized models.
Figure 10: Experiment 1 Trial 1. Inception Quant NPU

Figure 11: Experiment 1 Trial 2 Inception FP NPU
Figure 12: Experiment 1 Trial 3 Inception FP GPU

Figure 13: Experiment 2: Trial 1: MobileNet Quant NPU
Figure 14: Experiment 2: Trial 2: MobileNet FP NPU

Figure 15: Experiment 2: Trial 3: MobileNet FP GPU