MediaPipe: A Framework for Perceiving and Processing Reality

Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Uboweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Guang Yong, Juhyun Lee, Wan-Teh Chang, Wei Hua, Manfred Georg and Matthias Grundmann
Google Research
mediapipe@google.com

Abstract

Building an application that processes perceptual inputs involves more than running an ML model. Developers have to harness the capabilities of a wide range of devices; balance resource usage and quality of results; run multiple operations in parallel and with pipelining; and ensure that time-series data is properly synchronized. The MediaPipe framework addresses these challenges. A developer can use MediaPipe to easily and rapidly combine existing and new perception components into prototypes and advance them to polished cross-platform applications. The developer can configure an application built with MediaPipe to manage resources efficiently (both CPU and GPU) for low latency performance, to handle synchronization of time-series data such as audio and video frames and to measure performance and resource consumption. We show that these features enable a developer to focus on the algorithm or model development, and use MediaPipe as an environment for iteratively improving their application, with results reproducible across different devices and platforms. MediaPipe will be open-sourced at https://github.com/google/mediapipe.

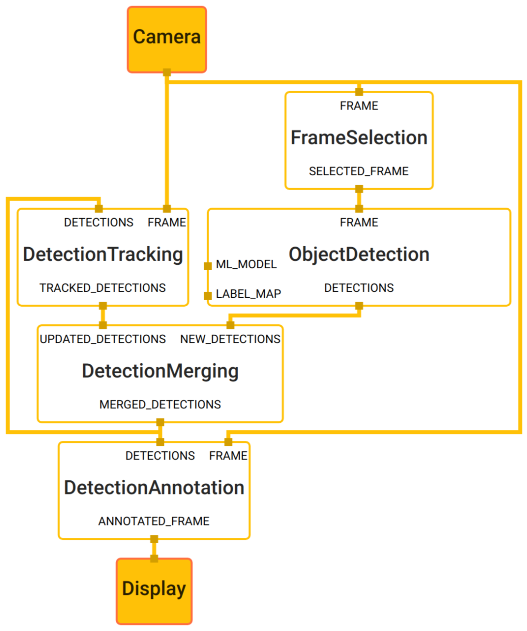

Figure 1: Object detection using MediaPipe.

1. Introduction

To enable augmented reality (AR), a typical application processes sensory data such as video and audio at high frame rates to enhance the user experience. Modifying such a perception application to incorporate additional processing steps or inference models can be difficult due to excessive coupling between steps. Further, developing the same application for different platforms is time consuming as it usually involves optimizing inference and processing steps to run correctly and efficiently on a target device.

MediaPipe addresses these challenges by abstracting and connecting individual perception models into maintainable pipelines. With MediaPipe, a perception pipeline can be built as a graph of modular components, including, for instance, inference models and media processing functions. Sensory data such as audio and video streams enter the graph, and perceived descriptions such as object detection results or face landmark annotations leave the graph. An example in shown in Figure 1.

Graphs of operations are used in projects such as TensorFlow [1], PyTorch [4], CNTK [5] or MXNet [2] to define a neural network model. MediaPipe takes a complementary role: our graphs do not define the internals of a neural network, but instead specify larger-scale pipelines in which one or model models are embedded.

OpenCV 4.0 introduced the Graph API (G-API) [3] which allows specifications of sequences of OpenCV image processing operations in the form of a graph. However, MediaPipe allows operations on arbitrary data types and provides native support for streaming time-series data.

2. Basic Concepts

MediaPipe consists of three main parts: (1) a framework for inference from sensory data, (2) a set of tools for performance evaluation, (3) a collection of reusable inference and processing components.

A pipeline is defined as a directed graph of components, where each component is a `Calculator`. Developers can define custom calculators. The graph description is specified via a `GraphConfig` protocol buffer and then run using a `Graph` object.

In a graph, data flows through each calculator via data `Streams`. The basic data unit in MediaPipe is a `Packet`. A stream carries a sequence of packets with monotonically increasing timestamps. Calculators and streams together define a data-flow graph. Each stream in a graph maintains its own queue to allow the receiving graph node to consume packets at its own pace.

Changes to the pipeline to add or remove components can be made by updating the `GraphConfig` file. A developer can also configure global graph-level settings to modify graph execution and resource consumption in this file. This is useful for tuning the performance of the graph on different platforms (*e.g*., on desktop vs. on mobile).

3. Usage Examples

Object Detection A common requirement for AR applications is real-time object detection from a live camera feed. Depending on the target device platform, running ML-based object detection at a full frame rate (*e.g*., 30 FPS) can require high resource consumption or be potentially infeasible due to long inference times. An alternative is to apply object detection to a temporally sub-sampled stream of frames and propagate the detection results, *i.e*., bounding boxes and the corresponding class labels, to all frames using a lightweight tracker. For optimal performance, tracking and detection should be run in parallel, so the tracker is not blocked by the detector and can process every frame. This perception pipeline can be easily implemented with MediaPipe, as presented in the example graph in Figure 1.

There are two branches in the beginning of the graph: a slow branch for detection and a fast branch for tracking. Calculators for these tasks can be configured to run on parallel threads with the specification of executors in the pipeline's graph configuration.

In the detection branch, a frame-selection node picks a subset of frames on which to run the detection model (based its decision on, e.g., a frequency limit, or scene-change analysis), dropping the rest. The object-detection node uses an ML model and the associated label map to perform inference on the incoming selected frames using an inference engine (*e.g*., TFLite [6]) and outputs detection results.

In parallel to the detection branch, the tracking branch updates earlier detections and advances their locations to the current camera frame.

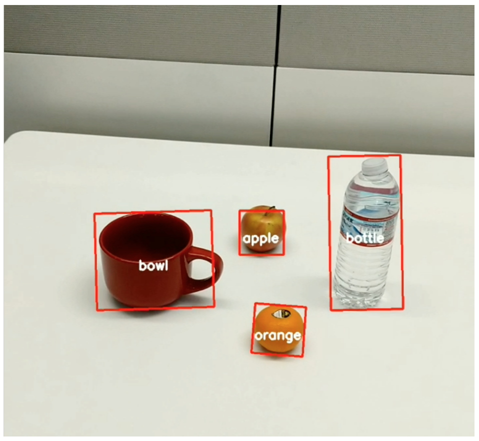

Figure 2: Object detection output.

After detection, the detection-merging node compares results and merges them with detections from earlier frames, removing duplicate results based on their location in the frame and/or class proximity.

The detection-merging node operates on the same frame that the new detections were derived from. This is automatically handled by the framework as it aligns the timestamps of the two sets of detection results before they are processed together. The node also sends merged detections back to the tracker to initialize new tracking targets if needed.

For visual display, the detection-annotation node adds overlays with the annotations representing the merged detections on top of the camera frames, and the synchronization between the annotations and camera frames is automatically handled by the framework before drawing takes place in this calculator. The result is an annotated viewfinder output, as seen in Figure 2. The output may be slightly delayed (*e.g*. by a couple of frames), but the frame rate remains smooth and all annotations are correctly synchronized.

Face Landmark Detection and Segmentation Face landmark estimation is another common perception application. Figure 3 shows a MediaPipe graph that performs face landmark detection along with portrait segmentation.

To reduce the computational load needed to run both tasks simultaneously, one strategy is to apply the tasks on two disjoint subsets of frames. This can be done easily in MediaPipe using a demultiplexing node that splits the packets in the input stream into interleaving subsets of packets, with each subset going into a separate output stream.

To derive the detected landmarks and segmentation masks on all frames, the landmarks and masks are temporally interpolated across frames. The target timestamps for interpolation are simply those of all incoming frames. Fi-

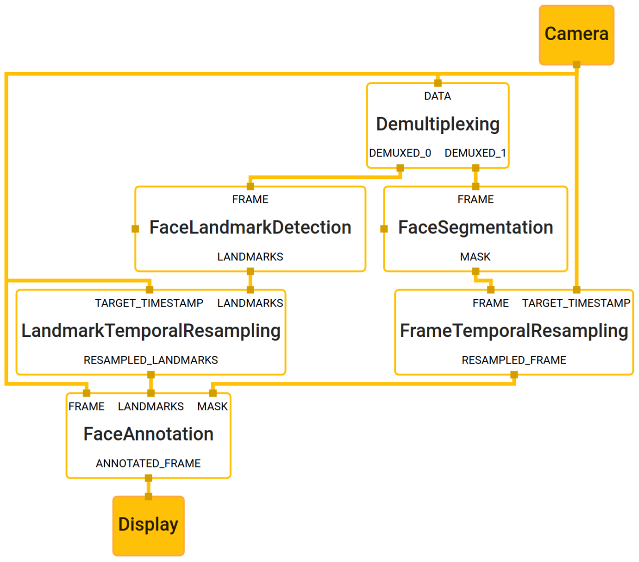

Figure 3: Face landmark detection and segmentation using MediaPipe.

nally, for visualization the annotations from the two tasks are overlaid onto the camera frames, with the three streams automatically synchronized when entering the annotation node. A snapshot of the visual annotation is shown in Figure 4.

The pipeline can be further accelerated with GPU compute while reusing most of the pipeline configuration. For example, the face-landmark-detection node can switch to a GPU-based implementation, using a GPU inference engine (*e.g.*, [7]). Additionally, temporal resampling and annotation can also have a GPU-based implementation. Together with the GPU support embedded in the framework, the entire data flow and compute can stay on the GPU end-to-end, avoiding bottlenecks commonly observed from GPU-to-CPU data transfer. Furthermore, it is also straightforward to leverage heterogeneous computing with the detection branch being computed on GPU while in parallel the segmentation branch is running on CPU.

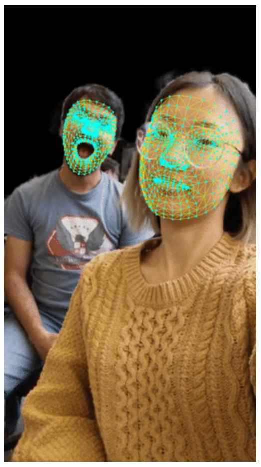

Figure 4: Landmark detection and segmentation output.

4. Tooling and Platforms

MediaPipe offers developer tools to inspect the run-time behavior of graphs and analyze their performance.

The MediaPipe tracer module follows individual data packets across a graph and records timing events along the way. The MediaPipe visualizer tool can help developers understand the overall behavior of their pipelines by visualizing the recorded timing events. The visualizer also offers a detailed view of the topology of the graph and lets a user observe the full state of the graph, its calculators and packets being processed, at any point in time.

MediaPipe facilitates the deployment of perception technology into applications on a wide variety of hardware platforms. MediaPipe supports GPU compute and rendering nodes, and allows combining multiple GPU nodes, as well as mixing them with CPU based nodes. GPU nodes can be built using various APIs (*e.g.*, OpenGL ES or Metal); CPU nodes can use popular image processing nodes such as OpenCV.

5. Conclusion

As newer devices with sophisticated CPUs and GPUs surface in the mobile market, it has become challenging for developers to quickly build and experiment with AR applications that utilize the hardware efficiently. In this paper, we introduced MediaPipe, a framework for building a perception pipeline as a graph of reusable components called calculators.

MediaPipe makes it easy to build a perception pipeline, optimize and improve it using its rich configuration language and performance evaluation tools. As shown in the above use cases, a developer can conveniently define custom calculators, configure their graph to use resources efficiently, and process media streams in parallel and at different rates to perform a complicated task in real-time. It is easy to reuse the calculators in different pipelines across successive applications as they share a common interface oriented around time-series data. The pipelines can run on a variety of platforms, enabling the developer to build the application on workstations and then deploy it on mobile.

A key element of MediaPipe's success is the ecosystem of reusable calculators and graphs. We have built and shipped many successful AR applications across millions of users using this framework, such as planar object tracking, augmented face effect filters, and real-time object augmentation.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. 1

[2] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015. 1

[3] Dmitry Matveev. OpenCV Graph API. Intel Corporation, 2018. 1

[4] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. 1

[5] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135, 2016. 1

[6] TensorFlow. TensorFlow Lite, 2017. https://www.tensorflow.org/lite, Last accessed on 2019-04-11. 2

[7] TensorFlow. TensorFlow Lite on GPU, 2019. https://www.tensorflow.org/lite/performance/gpu_advanced, Last accessed on 2019-04-11. 3