NNStreamer: Stream Processing Paradigm for Neural Networks, Toward Efficient Development and Execution of On-Device AI Applications

MyungJoo Ham[1] Ji Joong Moon[1] Geunsik Lim[1] Wook Song[1] Jaeyun Jung[1] Hyoungjoo Ahn[1]
Sangjung Woo[1] Youngchul Cho[1] Jinhyuck Park[2] Sewon Oh[1] Hong-Seok Kim[1,3]

[1]Samsung Research, Samsung Electronics
[2]Biotech Academy, Samsung BioLogics
[3]Left the affiliation
[1,2]{myungjoo.ham, jijoong.moon, geunsik.lim, wook16.song, jy1210.jung, hello.ahn, sangjung.woo, rams.cho, jinhyuck83.park, sewon.oh, hongse.kim}@samsung.com

Abstract

We propose *nnstreamer*, a software system that handles neural networks as filters of stream pipelines, applying the stream processing paradigm to neural network applications. A new trend with the wide-spread of deep neural network applications is on-device AI; i.e., processing neural networks directly on mobile devices or edge/IoT devices instead of cloud servers. Emerging privacy issues, data transmission costs, and operational costs signifies the need for on-device AI especially when a huge number of devices with real-time data processing are deployed. *Nnstreamer* efficiently handles neural networks with complex data stream pipelines on devices, improving the overall performance significantly with minimal efforts. Besides, *nnstreamer* simplifies the neural network pipeline implementations and allows reusing off-shelf multimedia stream filters directly; thus it reduces the developmental costs significantly. *Nnstreamer* is already being deployed with a product releasing soon and is open source software applicable to a wide range of hardware architectures and software platforms.

1 Introduction

We have witnessed the wide spread of deep neural networks and their applications in the last decade. With ever growing computing power of embedded or edge devices, neural networks are being adopted to such devices, further assisted by AI accelerators [2, 10, 24, 25, 27, 38, 44, 48]. In mobile phone industry, this trend has already become obvious enough to be adopted by major manufacturers including Samsung and Apple [2, 44]. Running intelligence mechanisms such as deep neural networks directly in edge devices including consumer electronics is often called as on-device AI [37].

On-device AI is becoming more attractive to manufacturers because of the following three advantages:

1. Avoid data privacy and protection issues by keeping the data in user devices without uploading to cloud servers.
2. Reduce data transmission latency of high-bandwidth real-time stream data including live video feeds from cameras.
3. Reduce operating cost by off-loading computation to devices from servers. This cost is often neglected; it is, however, significant if billions of devices are to be deployed.

We do not discuss the potential advantage, distributing workloads across edge devices, because it is not in the scope of this paper, but of future work.

On-device AI achieves such advantages by processing directly in the nodes where data exist so that data transmission is reduced and sensitive data are kept inside. However, on-device AI induces significant challenges. Normally, edge devices including mobile phones, TVs, home appliances, and IoT devices have limited computing power compared to workstations of conventional AI systems. In the meanwhile, on-device AI applications often require short response time or high processing rates; e.g., AR Emoji [43], Animoji [3], and offline speech recognition and translations. Besides, on-device AI applications often use sensors incurring data stream with high bandwidth: video cameras.

On-device AI applications are becoming more complex, incurring even more challenges. A neural network may process multiple input streams and multiple neural networks may process streams simultaneously. Besides, networks may share input data, and an input of a network may be outputs of other networks. For example, inputs of a facial recognition and an emotion recognition may share outputs of an object recognition, saving computing resources. Composing systems with multiple networks allows to train smaller networks (more efficient) and to save training costs. However, stream pipelines for such systems may become highly complicated with interconnections of networks and sensors along with fluctuating latency and synchronization issues. Another challenge is that the topology of pipelines may be dynamic.

We have observed another issue in practice; developers may use their own neural network frameworks such as TensorFlow and Caffe or their own tweaked version of such frameworks. With stable releases of applications, we may enforce developers to share a framework or to provide independent binaries without the need for any neural network frameworks. However, for researching and prototyping, developers want to ex-

periment with various frameworks while testers still need to execute the whole system integrated. Thus, we need to compose such complicated interactions between neural networks and sensors along with multiple neural network frameworks.

We apply the stream processing paradigm [46] to on-device AI systems. Our approach is to handle a neural network as a stream filter and then, to construct the system as a stream pipeline. We propose data standards for tensors and containers of tensors, which allow varying neural network frameworks in a pipeline along with conventional stream filters, conventional media types (video, audio, and text), and tensors. In order to construct complicated pipelines, we also propose various stream path manipulators with synchronization policies and tensor operators.

Handling stream pipelines efficiently along with various synchronization methods, filters, and data types has been a main topic of multimedia frameworks for decades. Among multimedia frameworks, we choose GStreamer [19] as the basis because GStreamer offers highly modular architecture, is fully open source software, and is compatible with various operating systems and hardware architectures. We can also reuse what GStreamer offers; hundreds of media processing plugins and developmental infrastructure. GStreamer has been widely deployed to desktop computing, video conferencing, broadcasting, and consumer electronics, even including televisions of Samsung and LG, where the multimedia performance is extremely critical.

An experimental product deployed to hospitals and households, which will be commercially released soon, is based on *nnstreamer* to process recognize events from multiple sensors. Product developers have significantly improved the performance and simplified the code, which have enabled to add more sensors and neural networks with inexpensive hardware and minimal developmental costs. *Nnstreamer* is open source software licensed with LGPL [16]. We plan to use *nnstreamer* for more products in the affiliation including robots and conventional consumer electronics.

2 Related Work

2.1 Multimedia Stream Processing in Practice

Multimedia stream frameworks have recently evolved to process high quality audio and video with mobile phones and TVs whose computing resources are limited.

GStreamer [19] is the standard multimedia pipeline framework for Tizen and many Linux distributions. GStreamer provides APIs and utilities to construct stream pipelines for multimedia applications of various platforms including Linux, Android, Windows, iOS, and macOS. GStreamer is highly modular; every filter and path control is implemented as a plugin attached in run-time. GStreamer has been applied to various systems where multimedia performance matters. BBC uses GStreamer for their broadcasting system [11]. Samsung and LG use GStreamer as the multimedia engine of televisions. Centricular [14, 15, 23, 49, 50] releases GStreamer for TVs, set-top boxes, medical devices, in-vehicle infotainment, video-on-demand, and on-demand streaming solutions.

Another popular framework for Linux, **FFmpeg** [5] is not modular, but everything is built-in and has limited cross-platform support. Note that GStreamer can embed FFmpeg as yet another plugin.

StageFright [17] is the multimedia stream framework of Android. StageFright depends on Android services and, unlike GStreamer, is not flexible enough for other generic Linux distributions. Besides, it does not allow to construct arbitrary stream pipelines.

AVFoundation [4] is the multimedia stream framework of iOS and macOS. Along with Core ML, a machine learning framework, we may have AVFoundation as an input to Core ML; however, it is not supposed to write arbitrary pipelines with Core ML entities as stream filters.

DirectShow [9] is the multimedia framework of Windows. We cannot alter and use DirectShow or AVFoundation for the given purpose because it is proprietary software and supports proprietary platforms only.

2.2 Stream Processing in General

StreamCloud [21] has addressed the bottleneck of a single input node in a stream pipeline. StreamCloud provides a scalable and elastic stream framework to process large data input and execute pipeline elements in parallel transparently by splitting queries to allocate multiple nodes.

There are studies to help manage large data streams efficiently; however, they do not provide a general stream processing framework for neural networks and complex topology. Kumar [32] has proposed a network data stream analysis tool with higher accuracy and lower overhead than prior work. In order to reduce power consumption of edge devices receiving multimedia data streams, a client-side statistical prediction scheme [51] estimates the data patterns of incoming data streams and tries to make processors sleep longer. A fragmental proxy-caching scheme [47] tries to improve the quality of multimedia streams by adjusting caching units with finer granularity. Apache Flink [7] proposes a platform with a universal dataflow engine designed to perform both stream and batch analytics for processing streaming and batch data.

Applying streaming framework for GPUs fits well with general architectures of GPUs. GStream [54] provides a general-purpose and scalable data streaming framework for GPUs. Nvidia Tesla architecture [36] provides a flexible, programmable graphics and high-performance computing along with the Compute Unified Device Architecture (CUDA) platform. Jiangiang Dong [13] demonstrates how modern GPUs and CUDA can improve the performance of BIRCH, one of well-known clustering techniques for streaming data.

2.3 Stream Processing for Neural Networks

Nvidia DeepStream [28, 40] provides neural network filters for GStreamer. DeepStream requires the input and output of a neural network to be conventional media stream. Other critical issues include: a) DeepStream is proprietary, b) it supports Nvidia hardware only, and c) general neural network frameworks cannot be applied directly.

In GStreamer Conference 2018, other approaches to apply neural networks to GStreamer are introduced as well. Intel [52] proposes an approach similar with DeepStream, sharing the same issues. Pexip [45] proposes to use OpenCV to apply neural networks as prior work [35]. This approach is the least invasive method; however, this cannot be applied to neural networks with arbitrary input and output streams or with general neural network frameworks. RidgeRun [26] has proposed an approach somewhat similar with *nnstreamer*. However, it does not support neural networks with arbitrary input and output streams or path controls and, according to the opened code [18], it is far from releases.

In contrast, we recognize "tensor" as a stream data type, not limiting streams to conventional media types (audio, video, and text), allowing outputs of neural networks to be inputs of other networks in general and stream path controls discussed in Chapter 4. We provide efficient and high-performing methods to process neural network pipelines in parallel with arbitrary frameworks, hardware, and platforms.

3 Approaches

We describe the approaches of *nnstreamer* with example pipelines showing the need for each component. Later in Chapter 4, we show how such components are implemented.

3.1 Basic Components

Neural network frameworks (NNFWs), such as Caffe and TensorFlow, allow developers to describe, train, and execute neural network models. NNFWs also provide execution environments along with hardware acceleration.

An **NNFW filter**, "Neural Network" in Figure 1, allows to attach a neural network model and its NNFW as a stream filter of a pipeline. Each NNFW filter may use its own NNFW. A **stream sink** transmits the results to external entities such as applications and visualization services.

We need a standard stream data format representing inputs and outputs of neural networks, called "tensor". With the standard, we may run different NNFWs in a single pipeline and implement common data and path manipulators, directing outputs of networks as inputs of others. Then, we need **Converters** converting media streams to "tensor" streams.

We need **preprocessors** for tensors and media streams. A neural network input may have strict requirements; e.g., AlexNet [31] requires a normalized tensor of 3 x 224 x 224. To meet such requirements, we need various preprocessors including conventional media filters (e.g., Video Decode & Scale in Figure 1) and **tensor operators** (e.g., Normalize & Transpose in Figure 1.)

3.2 Isodimensional Stream Path Control

In Figure 2, we show three dimension-preserving stream path controls. Rectangles with dotted lines represent sets of filters. Shaded rectangles show data formats. Thick arrows are streams where each frame has multiple instances of "tensor"s.

Tee allows multiple paths to share same data.

Mux multiplexes streams into a single stream having multiple instances of "tensor"s in each frame, referred as "tensor**s**". In the figure, properties of "tensors" are denoted with curly brackets. Because Mux has a single output stream, its input streams are synchronized by Mux, which provides multiple synchronization policies. For example, "slowest" synchronizes with the slowest input, "base" synchronizes with a specific input, and "fastest" generates an output whenever an input is available. With "base" or "fastest", Mux may need to reuse input frames from a slower stream. For example, in Figure 2, if the policy is "fastest", previous frames from Infra-Red may be reused to meet 60 Hz.

Mux needs to look at time-stamps of incoming stream, stamped by sensors. Mux needs to choose each frame from the input streams with the closest time-stamp. For example, if frames of time-stamps, $\{14, 30, 49\}$, are available from Infra-Red and $\{29\}$ from RGB Camera has just arrived, Mux chooses 30 unless a user specifies otherwise.

Demux demultiplexes an incoming "tensor**s**" stream into multiple "tensor" streams as shown in the figure. We do not need synchronization mechanisms for Demux.

Recurrent neural networks (RNN) are popular for handwriting recognition and speech recognition applications. If a recurrence occurs in a single element of a pipeline without affecting other elements, referred as an *internal recurrence*, it is not exposed to its stream pipeline. However, as shown in Figure 3, if the recurrence affects and is visible to the pipeline, referred as an *external recurrence*, we need **Recurrence Helper**. Recurrent Helper resolves the bootstrapping issue; the output of Model 2 in the figure is not available at the start, which, in turn, makes the input of Model 1 not available and blocks the the whole pipeline. Besides, if we have meta data streams flowing back and forth as in GStreamer pipelines for QoS, cyclic paths can be prohibited; Recurrence Helper should resolve such conflicts as well.

3.3 Non-isodimensional Stream Path Control

Non-isodimensional stream path controls may alter the dimensions of tensors. Figure 4 shows two of such controls, **Merge** and **Split**. As shown in the figure, the dimension being merged or splited needs to be specified. Merge needs

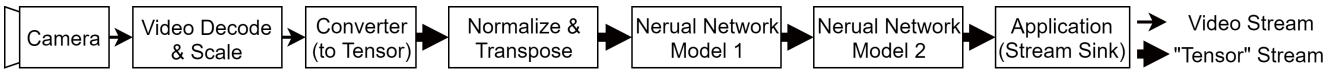

Figure 1: Basic linear pipeline with two neural networks and single input and output without path manipulations

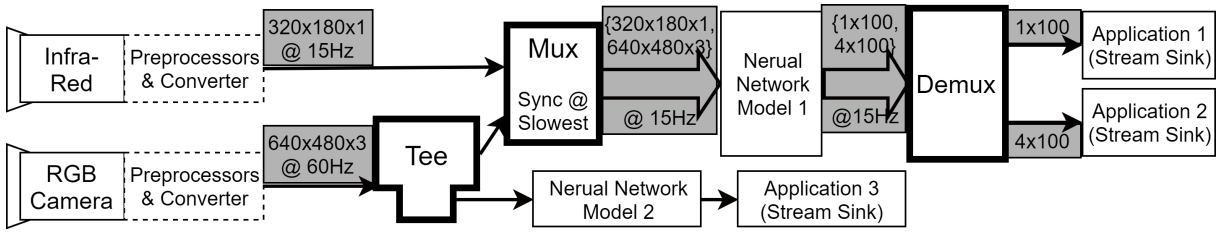

Figure 2: Pipeline with isodimensional stream path controls: Tee, Mux, and Demux

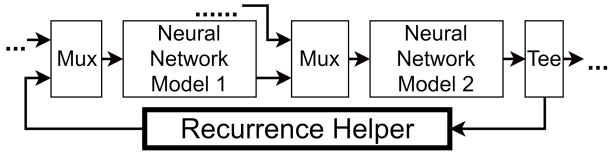

Figure 3: Pipeline with external recurrences

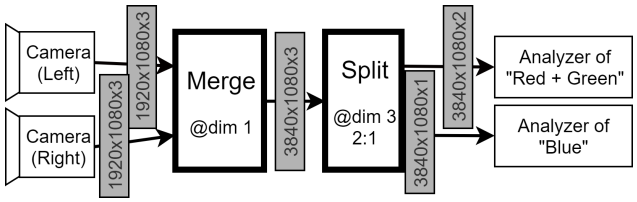

Figure 4: Pipeline with Merge and Split

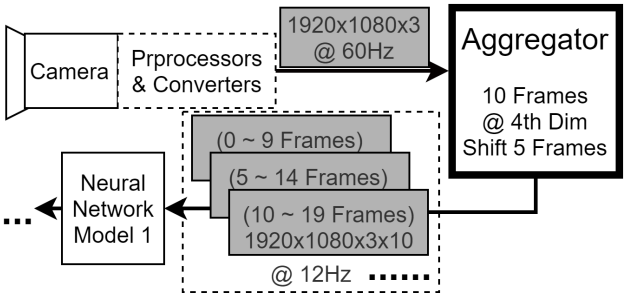

Figure 5: LSTM pipeline with Aggregator

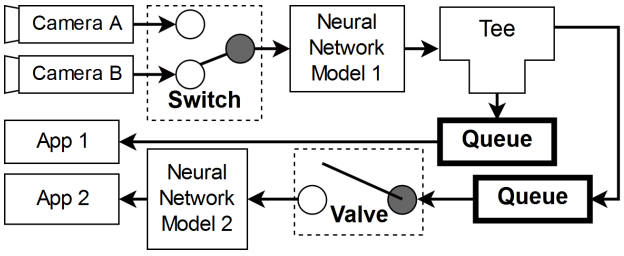

Figure 6: Pipeline with queues, a switch, and a valve

synchronization and time-stamp mechanisms like Mux.

The long short-term memory (LSTM) is popular for object tracking, machine translation, image captioning, and various applications. In order to generate inputs for LSTM, we need to aggregate multiple frames of a stream into a single frame: **Aggregator** in Figure 5. Aggregator merges frames temporally while Mux and Merge merges frames spatially. Figure 5 shows an LSTM model processing 10 recent frames simultaneously. Similar configuration can be seen in activity recognition sensors [33], which use *nnstreamer*, in Chapter 5.1.

3.4 Other Requirements

From the beginning, we have been cooperating with potential product developers, who have requested the followings:

- Dynamic flow control to enable and disable neural networks quickly: **Valve** in Figure 6.
- Change stream sources dynamically and easily in case of sensor faults or mode changes: **Switch** in Figure 6.
- Memcpy-less data transmission between stream filters.
- Transparent parallelism. In Figure 6, developers may want to execute `Model 1`, `Model 2`, and `Application 1` in parallel without explicitly implementing threads or synchronization methods.
- Dynamic stream pipeline topology. Add, replace, realign, or remove elements of a pipeline.
- Manage time-stamps from sensors. With Mux, Merge, and Aggregator, this is not trivial; which time-stamp is valid if we have multiple values for a frame?

4 Implementation

Nnstreamer is not an experimental prototype, but software with imminent schedule and pressure of commercialization; thus, it should be ready for tiring quality assurance processes.

Some of authors have been working for the autonomous vehicle project, where the need for applying stream processing paradigm has emerged. The vehicle has a lot of heterogeneous sensors and neural networks inter-connected with complex topology, transmitting large data streams, and processed simultaneously. The huge data transmission overhead between nodes had been the core cause requiring *nnstreamer*. Then, managing complex and ever-changing data paths, allowing different NNFWs, and parallelism have been incurring the need for *nnstreamer*.

Spinning off this project from the autonomous vehicle project, we have discovered that such needs are common: robots, manufacturing plants, and consumer electronics. As a result, *nnstreamer* has already had a lot of potential applications in the affiliation from the beginning, which enforces to release partial but practical solutions quickly.

> Don't reinvent the wheel, just realign it.
> – Anthony J. D'Angelo

If we implement yet another stream framework, it would take too much time and effort. Fortunately, a stable, widely-deployed, and open-source stream framework with rich features already exists: GStreamer. GStreamer is a multimedia stream framework battle-tested with a lot of services including BBC [11] and consumer electronics including televisions (both Samsung and LG), where multimedia performance is critical. Releasing large number of products, a lot of vendors have made sure GStreamer is reliable, high-performing, and rich-featured. Tizen, which runs a wide range of devices including most types of consumer electronics [22], also has GStreamer as its standard multimedia engine. As a result, most of requirements in Chapter 3.4 and the above are already satisfied by GStreamer.

We only need to "realign" GStreamer for neural networks and their applications. We implement what GStreamer is missing: the features described in Chapter 3 except for "Tee", "Valve", "Switch", "Queue", and conventional media filters.

4.1 Standard Stream Data Types

We have designed two new data types for GStreamer, "`other/tensor`" and "`other/tensor`**`s`**", representing a stream of a multidimensional array and a stream of a container of multiple instances of such arrays, respectively. The definitions of both stream data types in the following paragraphs show how each frame instance of a stream appears.

```
other/tensor
    framerate = (fraction) [0/1, 2147483647/1]
    dimension = Dim
    type = Type

other/tensors
    num_tensors = [1, 16]
    framerate = (fraction) [0/1, 2147483647/1]
    dimensions = Dims
    types = Types
# num_tensors == # types == # dimensions.

Types = Type | Type,Types
Type = { uint8, int8, uint16, int16, uint32,
       int32, uint64, int64, float32, float64}
Dims = Dim | Dim,Dims
Dim = [1,65535]:[1,65535]:[1,65535]:[1,65535]
# [min, max] denotes the allowed value ranges.
```

Unlike other approaches [26,28,45,52], neural networks in *nnstreamer* do not use conventional media types, but use the standard defined above because of the advantages mentioned in Chapter 3.1. For demonstrations, we may assume that the neural network output is a video stream; however, if an application uses the output, it usually wants the raw data, not the rasterized video frames. More critically, if there is another neural network using the output as its input, conventional media types do not fit often; e.g., an array of label probabilities is neither audio or video. Besides, a neural network may require inputs preprocessed in the form not compatible with conventional media types or the input itself might be not compatible with conventional media types. Therefore, standard tensor data stream types are critically required.

4.2 Stream Filters

Nnstreamer, a GStreamer plugin, v0.1.0-1rc1, has the following elements released as stream filters.

- **tensor_converter** converts audio, video, text, or arbitrary binary streams to `other/tensor` streams.
- **tensor_decoder** converts `other/tensor(s)` to video or text stream with assigned sub-plugins.
- **tensor_filter** invokes a neural network model with the given model path and NNFW name.
- **tensor_transform** applies various operators to tensors including typecast, add, mul, transpose, and normalize. For faster processing, it supports SIMD instructions and multiple operators in a single filter.
- **tensor_mux**, **tensor_demux**, **tensor_merge**, **tensor_split**, and **tensor_aggregator** support tensor stream path controls.
- **tensor_reposink** and **tensor_reposrc** implement Recurrent Helper of Figure 3. In a pipeline of the figure, tensor_reposink is attached to "Tee" and tensor_reposrc is attached to the second "Mux", cutting the cycle in the pipeline. There is a shared repository for the two elements to transmit tensors without GStreamaer stream paths.

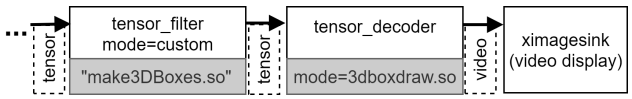

Figure 7: User sub-plugins (shaded) applied to elements

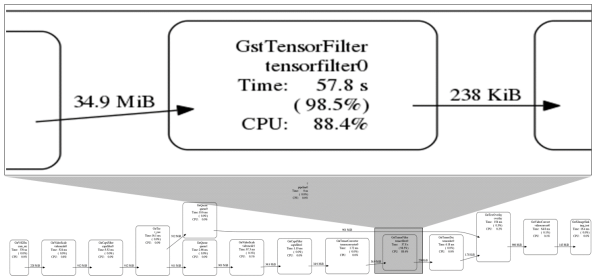

Figure 8: Screenshot of profiling an *nnstreamer* pipeline.

There are two elements allowing adding user created features in run-time: tensor_filter and tensor_decoder. Figure 7 shows how user created neural networks and decoders can be attached to tensor_filter and tensor_decoder in a pipeline. In the figure, "`make3DBoxes.so`" is a neural network without NNFWs and "`3dboxdraw.so`" is a user decoder sub-plugin, decoding results of "`make3DBoxes.so`" to video streams. If the tensor_filter element has "`mode=tensorflow-lite`" and "`model=x.tflite`", TensorFlow-lite processes "`x.tflite`". *Nnstreamer* provides APIs to implement sub-plugins. And sub-plugins can be implemented and attached in run-time except for adding a new NNFW to tensor_filter, which is to be supported with later releases.

4.3 Developmental Environments

A CI system [34] builds and tests every pull-request in GitHub for Tizen/{x64,arm64,arm32,x86} and Ubuntu/x64. We check *nnstreamer* in Ubuntu/{arm32,arm64} daily. We occasionally check Yocto and Android releases as well.

GStreamer supports various programming languages, hardware architectures, and operating systems. As a plugin of GStreamer, we ensure the compatibility by writing C89 code and using only the libraries GStreamer uses except for detachable sub-plugins and test cases. APIs are available for C, C++, .NET, Java, Python, Rust, Perl, Qt, Haskell, D, Guile, Ruby, and Vala. It supports Linux, Android, Windows, Mac OS X, iOS, BSD, Unix, Solaris, and Symbian along with x86, ARM, MIPS, SPARC and PowerPC.

We can reuse a wide range of stream analysis tools including profilers, tracers, test suites, and debuggers from GStreamer. Figure 8 is a screenshot of a tool analyzing a *nnstreamer* pipeline. We heavily depend on such tools to optimize or to fix bugs of commercialization projects.

5 Evaluations

We experiment with *nnstreamer* 0.1.0-1rc1 in devices described in Table 1 for two applications, an activity recognition sensor (ARS) based on [33] and Multi-Task Cascaded Convolutional Networks (MTCNN) [12,53]. Device `A`, Artik 530s, is the computing engine of ARS consisting of a dynamic vision sensor (DVS) [6] and an ultra-wideband (UWB) [8] sensor. Device `B`, Odroid-XU4, is an inexpensive development board with the SoC of Samsung Galaxy S5. Device `A` and `B` represent mid to low-end embedded devices. Device `C` is a development board with Exynos 8890 used by Samsung Galaxy S7 and automotive industry [42]. This represents high-end embedded devices. Device `D` is a desktop PC. We do not use GPUs in the experiments.

For ARS, we use Device `A` and compare an *nnstreamer* pipeline against a conventional implementation as the control. For MTCNN, we use Device `B`, `C`, and `D` and compare an *nnstreamer* pipeline against ROS [41], popular middleware for robotics projects, implementation as the control. The control represents the product development before *nnstreamer*.

For both applications, *nnstreamer* improves the performance significantly. The implementation is significantly simplified along with the improved code quality as well.

5.1 Experiments with ARS

ARS is a product experimentally deployed to households and public facilities and will be commercially available soon. We have applied *nnstreamer* to ARS to simplify the implementation, to improve the performance, and to add more sensors and features. Because the hardware, `A` in Table 1, has limited resources, processing multiple neural networks and streams is challenging. To evaluate the performance with the exactly same data, we use preprocessed input data stored as a file.

Figure 9 describes algorithms for ARS and Figure 10 shows the *nnstreamer* pipeline for ARS. `A` in Figure 9 is the initial CNN-based DVS processing algorithm [33], where each instance of CNN accepts 8 consecutive images with offsets of 4 frames. Then, it detects activity events by applying the argument max operator to 6 latest CNN results. `B` improves `A` by adding another neural network, LSTM, instead of the simple argument max. Both `A` and `B` use NEON SIMD to accelerate for both *nnstreamer* and conventional implementations. They are attached as custom sub-plugins of tensor_filter. `C` is a CNN-based neural network implemented with TensorFlow-lite processing 75 consecutive frames from two standardized streams from UWB, which outputs two streams.

Table 2 compares the *nnstreamer* implementation and Control, which implements ARS with Python and NumPy computation library. The last column shows the performance improvement achieved by *nnstreamer* with geometric means; i.e., how much CPU time or memory is saved by *nnstreamer* or how much faster is *nnstreamer* than Control.

	A / Artik 530s	B / Odroid-XU4	C / 8890	D / PC
SoC	Nexell S5P4418	Samsung Exynos 5422	Samsung Exynos 8890	Intel i7-7700
ISA	armv7 (32 bits)	armv7 (32 bits)	armv8 (64 bits)	x64 (64 bits)
CPU Microarchitecture	4 CA9	4 CA15 + 4 CA7	4 Mongoose + 4 CA53	4 Kaby-Lake
CPU Clock Speed	1.2 GHz	2 GHz / 1.5GHz	2.3 GHz / 1.6 GHz	3.6 GHz
DRAM Capacity	1 GiB DDR3	2 GiB LPDDR3	4 GiB LPDDR4	16 GiB DDR4
DRAM Throughput[1]	0.51 / 0.75	2.62 / 4.08	5.86 / 6.40	18.54 / 13.78
OS & Linux Kernel	Ubuntu 16.04 / 4.4	Tizen 5.0 / 4.1	Tizen 5.0 / 3.18	Ubuntu 16.04 / 4.15

Table 1: Specification of experimented devices. CA denotes Cortex-A. Linux kernels are supplied by the vendors.

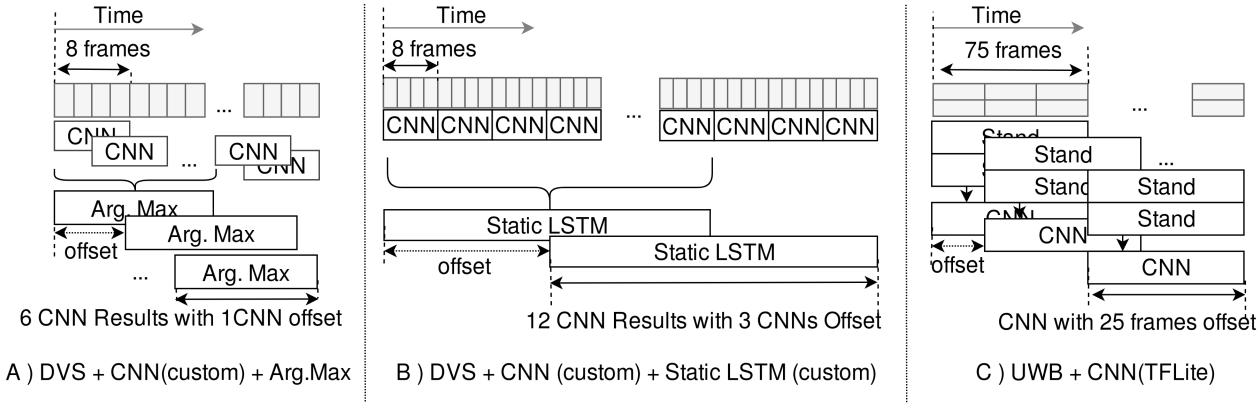

Figure 9: Neural network algorithms of activity recognition sensors (ARS)

Row 1 shows the total lines of codes except for neural network models, which indirectly suggests that *nnstreamer* may lower the developmental costs. Both Control and *nnstreamer* share C binaries of neural network models, `A` and `B`. *Nnstreamer* C codes, `A` (169) and `B` (186), are trivial wrappers for the models based on the *nnstreamer* template. Control Python codes, `A` (364) and `B` (359), handle pipelines and threads, which are replaced by a single-line shell script with *nnstreamer* described in the last paragraph of this chapter. Surprisingly, *nnstreamer* has taken few days of a single developer, with no known bugs, to implement what Control has taken a few weeks of developers along with known bugs.

Other rows in Table 2 evaluate the performance. Row 2 suggests that *nnstreamer* incurs less memory copies, which is critical for embedded devices with low memory throughput.

Row 3 suggests that both implementations try to exploit parallelism with multi-threading. Row 4, 5, and 6 show the CPU usage (resource consumption), output frames-per-second (throughput), and the efficiency (throughput / resource consumption) respectively in case we have unlimited input rate; i.e., push as many input frames as stream filters process. They show that *nnstreamer* implementation is far more efficient (75.0 %) and has higher throughput (65.5 %) although *nnstreamer* has incurred significantly less time and effort of developers. Note that the output rate may be slower than the input rate because Aggregator aggregates multiple frames.

The efficiency of real product is shown with Row 7; *nnstreamer* saves 43.5 % of CPU workloads in average. Note that *nnstreamer* takes only a quarter of CPU time compared to Control to process UWB. With higher efficiency, *nnstreamer* enables inexpensive devices to process more sensors and neural networks with reduced developmental costs.

Nnstreamer does not incur memory-copy for inter-filter data transmissions. Even with Tee, there is no memory-copy between a source and sinks of the Tee unless there is an in-place operation–writing the output directly to the input buffer–in a sink. Therefore, as long as there are no rogue *nnstreamer* custom filters, memory-copy operations are expected to be minimized while developing such optimized but complex pipelines with the conventional method is not so trivial.

Nnstreamer promotes parallelism with minimal developmental efforts. The stream pipeline paradigm allows to distribute workloads to multiple elements even if there is a single stream pipeline. Functional parallelism is promoted by processing elements or sub-pipelines in different processors with less efforts; the short code in the last paragraph of this chapter is sufficient. Besides, efficiency is further promoted by regulating processing rates; e.g., a producer will not process faster

[1] Read / write throughput in GiB/s. Benchmarked with "pmbw", https://github.com/bingmann/pmbw. Big cores are used for `B` and `C`.

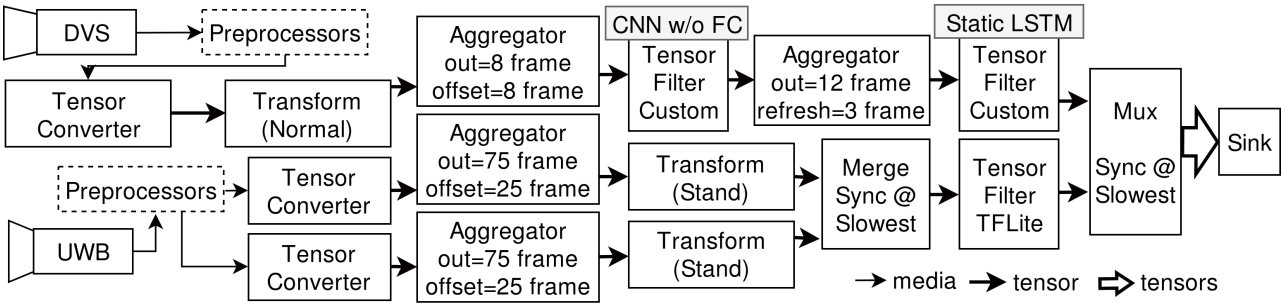

Figure 10: ARS Pipeline. The top half, B of Figure 9, processes DVS. The bottom half, C of FIgure 9, processes UWB.

Row	Metric	A) DVS Arg Max Control	A) DVS Arg Max *Nns*	B) DVS LSTM Control	B) DVS LSTM *Nns*	C) UWB Control	C) UWB *Nns*	Improvement by *Nns* (%)
1	LOC	364	169	359	186	383	1	-
2	Total mmap amount (MiB)	145	80	161	96	142	58	47.5
3	Max # threads	12	9	17	9	16	8	-
4	CPU (%), ∞ input rate	31.18	28.50	56.52	49.50	44.00	46.50	5.4
5	Output FPS, ∞ input rate	46.0	59.4	2.5	3.2	9.3	25.5	65.5
6	Output FPS / CPU (%), ∞ input rate	1.48	2.08	0.044	0.065	0.211	0.548	75.0
7	CPU (%), 30 FPS input rate	29.38	17.35	38.95	28.50	22.10	5.50	43.5

Table 2: Experimental results of ARS with 1100 input frames. FPS denotes frames per second. Nns denotes *nnstreamer*.

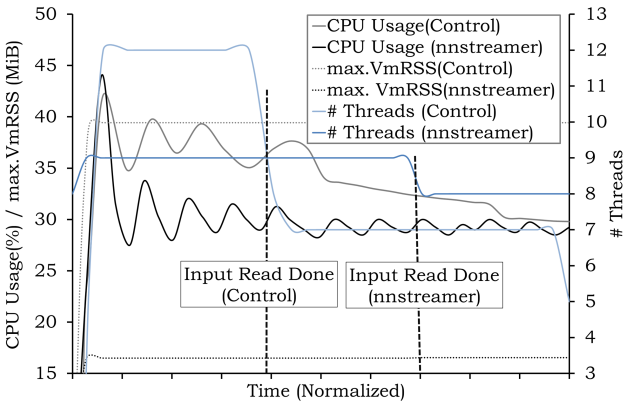

Figure 11: Resource utilization over time

than if its only consumer.

Figure 11 shows resource utilization over time with unlimited input rates. The time on x-axis is normalized so that both cases are completed at the end of x-axis. Both CPU Usage and # Threads suggest that *nnstreamer* not only incurs less workloads, but also distributes workloads more evenly over time. Memory, an expensive resource for embedded devices, is significantly saved by *nnstreamer* as well: 39 vs 17 MiB.

The major barrier to using complex pipelines with multiple sensors and neural networks has been the difficulties of implementing the pipeline topology with data protections, processing rate regulations, and efficient data transfers. After the demonstration of the *nnstreamer* pipeline (Figure 10), ARS developers have abandoned their implementations and started using *nnstreamer* for its simplicity and efficiency. The following code is a shell script implementing the whole pipeline, which enables to test and modify it quickly along with off-the-shelf GStreamer debugging and profiling tools.

```
$ gst-launch-1.0 tensor_mux name=mux ! fakesink
! tensor_converter ! tensor_trans mode=arith
! tensor_aggregator in=1 out=8 flush=8
! tensor_filter frame=custom m=./cnn.so
! tensor_aggregator in=1 out=12 flush=3
! tensor_filter frame=custom m=./lstm.so
! mux.sink_0
tensor_merge name=merge sync-mode=slowest
! tensor_filter frame=tflite m=./uwb.tflite
! mux.sink_1
multifilesrc location="./input_uwb0_%04d.data"
! tensor_converter dim=1:1:32:1 type=float32
! tensor_aggregator in=1 out=75 flush=25
! tensor_trans mode=stand ! merge.sink_0
multifilesrc location="./uwb1_%04d.data"
! tensor_converter dim=1:1:32:1 type=float32
! tensor_aggregator in=1 out=75 flush=25
! tensor_transform mode=stand ! merge.sink_1
```

5.2 Experiments with MTCNN

We evaluate *nnstreamer* with MTCNN [12, 53] because MTCNN has a complex pipeline topology and real life use cases for many applications; it detects faces and their alignments in a video frame. We compare the *nnstreamer* and ROS [41] implementations with Full-HD videos in various devices, `B` (low/mid-end embedded), `C` (high-end embedded), and `D` (PC) described in Table 1.

Figure 12 shows the *nnstreamer* pipeline topology of MTCNN, which has two large sub-pipelines. One, the top dotted rectangle in Figure 12, composites (GStreamer/cairooverlay) and displays (GStreamer/ximagesink) the live camera feed and the result of the other sub-pipeline. The other sub-pipeline, representing the MTCNN algorithm, is divided into three stages: Proposal Network (P-Net), Refine Network (R-Net), and Output Network (O-Net). MTCNN uses a cascade of convolutional networks in an image pyramid. In other words, each input frame is scaled down to various sizes comprising an image pyramid in P-Net, where each size is represented by a layer. Therefore, P-Net Stage has multiple streams (one for each layer) and an instance of P-Net including pre- and post-processing is processed in each stream. Then, in the order of P-Net, R-Net, and O-Net, each stage is linked and processed, which is post-processed by Non-Maximum Suppression (NMS) and Bounding-Box Regression (BBR).

As shown in Figure 12, MTCNN requires image processors: scale, transpose, color space conversion, and data type conversion. Before applying *nnstreamer*, ROS developers have relied on OpenCV for image processing, requiring additional time and effort. Besides, the performance enhancement with parallelism has not been successful with the ROS implementation. To promote parallelism, first, they have tried to adopt the pipe and filter style [39] (implementing the stream pipeline paradigm) by dividing the stages as ROS nodes. However, adding more ROS nodes have increased the message overheads (we use Full-HD videos!) and deteriorated the throughput significantly. As a result, they have abandoned the pipe and filter style despite of the observation that we can significantly accelerate P-Net by exploiting parallelism. Applying parallelization libraries such as POSIX Threads to such complex architecture is not a trivial task for many AI project developers, either.

As in ARS, exploiting parallelism has been highly simplified and trivial with *nnstreamer*. The following code implements a single layer of P-Net[2], where `gst_parse_launch` [20] constructs a pipeline, which can be a part of a larger pipeline.

```
gchar *desc1 = g_strdup_printf (
  "queue ! videoscale ! "
  "video/x-raw,width=%d,height=%d ! "
  "tensor_converter ! "
  "tensor_transform mode=arithmetic "
    "option=typecast:float32,add:-127.5,"
    "mul:0078125 ! "
  "tensor_transform mode=transpose"
    "option=0:2:1:3 ! "
  "tensor_filter framework=tflite "
    "model=./pnet.tflite ! "
  "tensor_filter framework=custom "
    "model=./pnet_pb.so",
  width1, height1);

GstElement *layer1 = gst_parse_launch (
  desc1, NULL);
```

The above code of a single P-Net layer processes a given size in the image pyramid (`width1` × `height1`), and is linked to Tee and Mux as shown in Figure 12. Then, each layer is executed as a thread, enabling functional parallelism for P-Net Stage, synchronized by the Mux between P-Net Stage and R-Net Stage. The Queue between the two Tees before P-Net Stage allows to drop frames for MTCNN if MTCNN cannot follow the frame rates of incoming video while not affecting the live feed displayed to Video Sink. For proper and optimized behaviors, we also need to configure queueing policies appropriately; i.e., how buffers are leaked and how many buffers may wait in a queue.

Table 3 shows the total lines of codes and the number of effective threads and their decompositions. Control C++ codes has three ROS nodes: Input, MTCNN (70 LOC), and Display (1574 LOC). The ROS-MTCNN node is a single threaded implementation that does not require any inter-node data transmissions. We do not count LOC of Input because we reuse open source code for Input. We do not count threads of ROS core and middleware as they are not effective for parallel executions of MTCNN.

Nnstreamer C/C++ codes have a pipeline (Display (51 LOC) and Pipeline (904 LOC) sub-pipelines) and tensor_filter custom sub-plugins (1004 LOC). Displays of both implementation have the same functionality and MTCNN of ROS is equivalent to Pipeline and Custom Filters of *nnstreamer*. Pipeline LOC counts the stream pipeline description of the MTCNN sub-pipeline. As with ARS, the two custom sub-plugins are the trivial wrappers based on the *nnstreamer* template. Compared to Control (1644), *nnstreamer* (1959) has more LOC. However, considering that only a few more trivial codes are required (333 LOC) to achieve parallelism in such a complex pipeline, the result suggests that *nnstreamer* allows to make development cost lower with higher efficiency and better quality than the conventional method. Besides, compared to Control, *nnstreamer* developers have added a lot more features including extensive error handling, appropriate frame dropping, dynamic number of layers, and dynamic in-

[2]Unlike ARS, the MTCNN implementation uses C APIs instead of shell scripts in order to construct a pipeline dynamically.

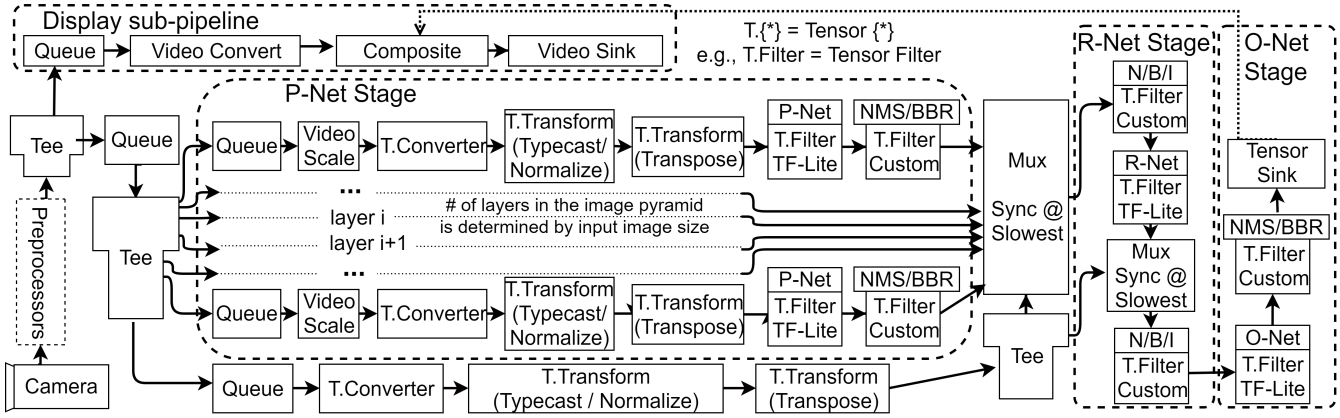

Figure 12: Pipeline Topology of *nnstreamer* MTCNN. N/B/I denotes NMS/BBR/Image Patch Generation

Row	Metric	Control	*nnstreamer*
1	LOC	1644	1959
2	Decomposed LOC	Display: 70, MTCNN: 1574	Display: 51, Pipeline: 904, Custom Filters: 1004
3	# effective threads	3	19
4	Decomposed # eff. threads	Input: 1, Display: 1, MTCNN: 1	P-Net: 14, Others: 5

Table 3: Lines of codes and the number of effective threads except ROS core threads with Full-HD video inputs.

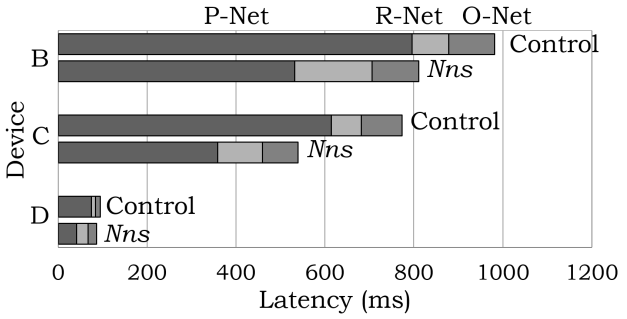

Figure 13: Breakdown of the face detection latency.

put video sizes and rates along with the reliability to handle camera inputs continuously.

Control consists of 3 single-threaded ROS nodes, incurring 3 effective threads. Besides, most computation happens with MTCNN node, which implies that there is little effective parallelism with Control except for OpenCV functions. *Nnstreamer* has much higher number of effective threads (19) although we do not write any multi-threaded codes explicitly. Particularly, P-Net Stage has 14 threads automatically and dynamically (according to the size of input images) created by the framework, which implies that *nnstreamer* highly promotes parallelism easily and efficiently.

Table 4 compares the performance of the two implementations in various devices. Row 1 shows the end-to-end latency between the sensor input to the display output for a single image frame with the input rate (1 FPS) slow enough to have a single frame in a pipeline. Moreover, Figure 13 shows the decompositions of latency, which suggests that most of the latency reduction comes from P-Net. More specifically, the performance, defined as $1/\text{latency}$, of P-Net Stage is improved by 49.73%, 71.53%, and 80.83% while the performance of other stages is degraded by 33.44%, 12.54% and 55.49% for Device `B`, `C`, and `D`, respectively. Even in such a case, *nnstreamer* improves the performance by 20.15 % (up to 30.55 % for Device `B`) because P-Net Stage is the most critical part. Note that P-Net Stage of ROS-MTCNN contributes 82.29% to the total latency. In other words, Row 1 shows the performance comparison without the impact of pipelined data-parallelism [29], but with the impact functional parallelism of P-Net despite of less efficient R-Net and O-Net.

Row 2 shows the output rates from 30 frames per second input of a Full-HD camera, processing Full-HD images with MTCNN. This allows pipelined data-parallelism, allowing processing different frames in filters. As a result, the performance improvement by *nnstreamer* is more significant: 83.21% (up to 171.62% for Device `B`.)

Note that the output frame rate is slower than the input rate; thus, input frames are often dropped in front of P-Net with the *nnstreamer* implementation. However, the frames for display compositing are not dropped and the final display output keeps showing 30 FPS video. On the other hand, the

Row	Metric	Input Rate	B / Odroid-XU4 Control	B / Odroid-XU4 *Nns*	C / 8890 Control	C / 8890 *Nns*	D / PC Control	D / PC *Nns*	Improvement by *Nns* (%)
1	End-to-end latency (ms)	1 FPS	981.78	811.00	704.49	539.64	94.28	85.91	15.35
2	Output rate (FPS)	30 FPS	1.01	1.73	1.48	4.02	10.41	13.76	83.21
3	Avg. CPU usage (%)	30 FPS	31.85	84.60	31.15	87.10	12.90	48.74	-
4	Memory usage (MiB)	30 FPS	136	307	129	474	248	440	-

Table 4: MTCNN performance with 1000 input frames. FPS denotes frames per second. Nns denotes *nnstreamer*.

ROS implementation has no such luxury and it cannot create 30 FPS display by dropping frames appropriately.

Analysing the effect of functional parallelism, we have mentioned that the performance of R-Net and O-Net have not been improved by the *nnstreamer* implementation; it is rather deteriorated as shown in Figure 13. This is mainly because the two custom tensor_filter sub-plugins (N/B/I) in R-Net Stage incur memory copies of Full-HD frames in order to generate small image patches for R-Net and O-Net. On the other hand, Control uses a global variable to avoid memory copies for the image patch generation. As a result, with devices of higher memory bandwidth, the performance deterioration in R-Net is reduced; 16.21 ms for Device `D`, 33.66 ms for Device `C`, and 91.98 ms for Device `B`.

The *nnstreamer* implementation uses more CPU time to process faster. Because, output rates of both implementations have not reached the input rates, it would be better to consume more CPU time to process more frames per second. The memory consumption is higher with the *nnstreamer* implementation although it is within the acceptable limit. However, MTCNN-*nnstreamer* consumes more than 10 times of the memory consumed by ARS-*nnstreamer*, which implies that the pipeline implementation is wasting the memory, probably by queues: we have attached too many queues that may store Full-HD frames. For the image pyramid of MTCNN, it would be significantly efficient (for both CPU and memory) if we write a custom tensor_filter sub-plugin that generates multiple layers of images directly from an input stream.

6 Future Work

The following filters and features are in our short-term plan.

- **tensor_ros_sink/src** allow to exchange tensor streams with Robot OS (ROS) [41] nodes for ROS1 and ROS2.
- **tensor_save/load** store tensor streams as files and load the files as tensor streams.
- **tensor_protobuf_sink/src** allow to use Google's protobuf as a transport layer.
- **tensor_source** accepts sensor data not compatible with GStreamer including LIDARs, RADARs, and others.
- More NNFWs for **tensor_filter**: Caffe/Caffe2 and Keras.
- Make `other/tensor(s)` a GStreamer standard type.

There are long-term plans as well. First, we will apply the actor model [1] to make pipelines fully-distributed. Robotics projects often incur multiple computers with different roles, which requires filters of a pipeline distributed across multiple computers.

Another long-term plan is to allow retraining a pipeline directly solely on device. Federated Learning [30] allows to update models by retraining models in cloud based on the data gathered from edges; however, it is difficult to apply Federated Learning if protected privacy data is required for training and the data cannot be transmitted. Therefore, we are sometimes required to retrain neural networks directly in edge devices.

7 Conclusion

From the experiences of on-device AI applications, our conjecture is that applying stream processing paradigm can greatly improve the performance, and the implementation productivity. By implementing and deploying *nnstreamer*, we show that such improvements can be achieved successfully for on-device AI applications: higher throughput and easier implementation with more features and higher code quality. As a side effect, traditional multimedia developers can now employ arbitrary neural network models in their own multimedia pipelines with *nnstreamer*.

This work is being deployed for commercial products by the affiliation of the authors. We plan to adopt *nnstreamer* to a wide range of devices by making it the standard intelligence framework of Tizen OS. This work is compatible and tested with many software platforms including Tizen, Android, and Ubuntu and is open source software licensed in LGPL.

Availability

Nnstreamer is open source software developed in a public GitHub repository. Ubuntu 16.04 and 18.04 users may use PPA to install *nnstreamer* and keep it updated:

```
$ sudo add-apt-repository ppa:nnstreamer/ppa
$ sudo apt-get update
$ sudo apt-get install nnstreamer
```

References

[1] AGHA, G. A. Actors: A model of concurrent computation in distributed systems. Tech. rep., MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[2] APPLE. iphone xs. https://www.apple.com/iphone-xs/specs/, 2018 (accessed 3 Jan 2019).

[3] APPLE. Use animoji on your iphone x and ipad pro. https://support.apple.com/en-us/HT208190, 2018 (accessed 3 Jan 2019).

[4] APPLE. Avfoundation. https://developer.apple.com/av-foundation/, (accessed 4 Jan 2019).

[5] BELLARD, F. Ffmpeg multimedia system. https://www.ffmpeg.org/, 2005 (accessed 4 Jan 2019).

[6] BERNER, R., LICHTSTEINER, P., DELBRÜCK, T., KIM, J. S., LEE, K., LEE, J., PARK, K., KIM, T., AND RYU, H. Dynamic vision sensor for low power applications. In *Proceedings of the 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)* (2014), IEEE, p. 1569961333.

[7] CARBONE, P., KATSIFODIMOS, A., EWEN, S., MARKL, V., HARIDI, S., AND TZOUMAS, K. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36*, 4 (2015).

[8] CHARLES FOWLER, JOHN ENTZMINGER, J. C. Assessment of ultra-wideband(uwb) technology. Tech. rep., OSD/DARPA, 1990.

[9] CHATTERJEE, A., AND MALTZ, A. Microsoft directshow: A new media architecture. *SMPTE Journal 106*, 12 (Dec 1997), 865–871.

[10] CHEN, T., MOREAU, T., JIANG, Z., ZHENG, L., YAN, E., SHEN, H., COWAN, M., WANG, L., HU, Y., CEZE, L., ET AL. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 578–594.

[11] CLARK, M., AND MOSCROP, L. Gstreamer for cloud-based live video handling. GStreamer Conference, 2018.

[12] DAI, J., HE, K., AND SUN, J. Instance-aware semantic segmentation via multi-task network cascades. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 3150–3158.

[13] DONG, J., WANG, F., AND YUAN, B. Accelerating birch for clustering large scale streaming data using cuda dynamic parallelism. In *Proceedings of the 14th International Conference on Intelligent Data Engineering and Automated Learning — IDEAL 2013 - Volume 8206* (New York, NY, USA, 2013), IDEAL 2013, Springer-Verlag New York, Inc., pp. 409–416.

[14] DRÖGE, S. Gstplayer - a simple cross-platform api for all your media playback needs. In *GStreamer Conference* (2015).

[15] DRÖGE, S. Synchronised multi-room media playback and distributed live media processing and mixing with gstreamer. In *GStreamer Conference* (2015).

[16] FREE SOFTWARE FOUNDATION. Gnu lesser general public license, version 2.1. https://www.gnu.org/licenses/lgpl-2.1.en.html, 1999 (accessed 3 Jan 2019).

[17] GOOGLE. Media (android developer manual). https://source.android.com/devices/media, (accessed 4 Jan 2019).

[18] GRUNER, M. A gstreamer deep learning inference framework. https://github.com/RidgeRun/gst-inference, 2018 (accessed 6 Jan 2019).

[19] GSTREAMER. gstreamer open source mutlimedia framework. https://gstreamer.freedesktop.org/, 1999 (accessed 3 Jan 2019).

[20] GSTREAMER. Gstparse — get a pipeline from a text pipeline description. https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer/html/gstreamer-GstParse.html#gst-parse-launch, 1999 (accessed 4 Jan 2019).

[21] GULISANO, V., JIMÉNEZ-PERIS, R., PATIÑO-MARTÍNEZ, M., SORIENTE, C., AND VALDURIEZ, P. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems 23*, 12 (Dec 2012), 2351–2365.

[22] HAM, M., AND LIM, G. Making configurable and unified platform, ready for broader future devices. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice Track* (2019).

[23] HERVEY, E. The new gststream api: Design and usage. In *GStreamer Conference* (2016).

[24] IGNATOV, A., TIMOFTE, R., SZCZEPANIAK, P., CHOU, W., WANG, K., WU, M., HARTLEY, T., AND VAN GOOL, L. Ai benchmark: Running deep neural networks on android smartphones. *arXiv preprint arXiv:1810.01109* (2018).

[25] IONICA, M. H., AND GREGG, D. The movidius myriad architecture's potential for scientific computing. *IEEE Micro 35*, 1 (2015), 6–14.

[26] JIMENEZ, J. M. Gstinference: A gstreamer deep learning framework. RidgeRun, GStreamer Conference, 2018.

[27] JOUPPI, N. P., YOUNG, C., PATIL, N., PATTERSON, D., AGRAWAL, G., BAJWA, R., BATES, S., BHATIA, S., BODEN, N., BORCHERS, A., BOYLE, R., CANTIN, P.-L., CHAO, C., CLARK, C., CORIELL, J., DALEY, M., DAU, M., DEAN, J., GELB, B., GHAEMMAGHAMI, T. V., GOTTIPATI, R., GULLAND, W., HAGMANN, R., HO, C. R., HOGBERG, D., HU, J., HUNDT, R., HURT, D., IBARZ, J., JAFFEY, A., JAWORSKI, A., KAPLAN, A., KHAITAN, H., KILLEBREW, D., KOCH, A., KUMAR, N., LACY, S., LAUDON, J., LAW, J., LE, D., LEARY, C., LIU, Z., LUCKE, K., LUNDIN, A., MACKEAN, G., MAGGIORE, A., MAHONY, M., MILLER, K., NAGARAJAN, R., NARAYANASWAMI, R., NI, R., NIX, K., NORRIE, T., OMERNICK, M., PENUKONDA, N., PHELPS, A., ROSS, J., ROSS, M., SALEK, A., SAMADIANI, E., SEVERN, C., SIZIKOV, G., SNELHAM, M., SOUTER, J., STEINBERG, D., SWING, A., TAN, M., THORSON, G., TIAN, B., TOMA, H., TUTTLE, E., VASUDEVAN, V., WALTER, R., WANG, W., WILCOX, E., AND YOON, D. H. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA '17, ACM, pp. 1–12.

[28] KHINVASARA, T. Introduction to deepstreamsdk. Nidia, GStreamer Conference, 2018.

[29] KING, C. T., CHOU, W. H., AND NI, L. M. Pipelined data parallel algorithms-i: Concept and modeling. *IEEE Trans. Parallel Distrib. Syst. 1*, 4 (Oct. 1990), 470–485.

[30] KONEČNÝ, J., MCMAHAN, H. B., YU, F. X., RICHTARIK, P., SURESH, A. T., AND BACON, D. Federated learning: Strategies for improving communication efficiency. In *NIPS Workshop on Private Multi-Party Machine Learning* (2016).

[31] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[32] KUMAR, A., SUNG, M., XU, J. J., AND WANG, J. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *SIGMETRICS Perform. Eval. Rev. 32*, 1 (June 2004), 177–188.

[33] LEE, H., KIM, J., YANG, D., AND KIM, J.-H. Embedded real-time fall detection using deep learning for elderly care. *arXiv preprint arXiv:1711.11200* (2017).

[34] LIM, G., HAM, M., MOON, J., SONG, W., WOO, S., AND OH, S. TAOS-CI: lightweight & modular continuous integration system for edge computing. In *2019 IEEE International Conference on Consumer Electronics (ICCE)* (Jan 2019, Accepted).

[35] LIN, C., AND TANG, Y. Research and design of the intelligent surveillance system based on directshow and opencv. In *2011 International Conference on Consumer Electronics, Communications and Networks (CECNet)* (April 2011), pp. 4307–4310.

[36] LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2 (March 2008), 39–55.

[37] MIT TECHNICAL REVIEW. On-device ai. https://www.technologyreview.com/hub/ubiquitous-on-device-ai/. (accessed 16 Dec 2018).

[38] MOREAU, T., CHEN, T., AND CEZE, L. Leveraging the vta-tvm hardware-software stack for fpga acceleration of 8-bit resnet-18 inference. In *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning* (New York, NY, USA, 2018), ReQuEST '18, ACM.

[39] NARUMOTO, M., BUCK, A., WASSON, M., AND BENNAGE, C. Pipes and filters pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters, 2017 (accessed 9 Jan 2019).

[40] NVIDIA. Deepstream sdk. *https://developer.nvidia.com/deepstream-sdk* (2018).

[41] QUIGLEY, M., CONLEY, K., GERKEY, B., FAUST, J., FOOTE, T., LEIBS, J., WHEELER, R., AND NG, A. Y. ROS: an open-source robot operating system. In *ICRA workshop on open source software* (2009), vol. 3, p. 5.

[42] ROCCO, N. L. Exynos 8890: Samsung liefert socs an audi für infotainmentsysteme. *Computer Base* (2017).

[43] SAMSUNG. Galaxy s9 | s9+, augmented reality. https://www.samsung.com/global/galaxy/galaxy-s9/augmented-reality/, 2018 (accessed 3 Jan 2019).

[44] SAMSUNG. Samsung exynos 9820. https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-9-series-9820/, 2018 (accessed 3 Jan 2019).

[45] SELNES, S. Bringing deep neural networks to gstreamer. Pexip, GStreamer Conference, 2018.

[46] STEPHENS, R. A survey of stream processing. *Acta Informatica 34*, 7 (1997), 491–541.

[47] WANG, J. Z., AND YU, P. S. Fragmental proxy caching for streaming multimedia objects. *IEEE Transactions on Multimedia 9*, 1 (Jan 2007), 147–156.

[48] WANG, S., PRAKASH, A., AND MITRA, T. Software support for heterogeneous computing. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (July 2018), pp. 756–762.

[49] WATERS, M. Opengl desktop/es for the gstreamer pipeline. In *GStreamer Conference* (2015).

[50] WATERS, M. Vulkan, opengl and/or zerocopy. In *GStreamer Conference* (2016).

[51] WEI, Y., BHANDARKAR, S. M., AND CHANDRA, S. A client-side statistical prediction scheme for energy aware multimedia data streaming. *IEEE Transactions on Multimedia 8*, 4 (Aug 2006), 866–874.

[52] XIANG, H. gst-mfx, gst-msdk and the intel media sdk. Intel, GSTreamer Conference, 2018.

[53] ZHANG, K., ZHANG, Z., LI, Z., AND QIAO, Y. Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE Signal Processing Letters 23*, 10 (2016), 1499–1503.

[54] ZHANG, Y., AND MUELLER, F. Gstream: A general-purpose data streaming framework on gpu clusters. In *2011 International Conference on Parallel Processing* (Sept 2011), pp. 245–254.