MEGH COMPUTING

Home Products White paper Blog About Contact

Implementing a real-time, deep learning pipeline with Spark Streaming

By Megh Computing, Inc.

January 27, 2019

2:22 pm

Deep Learning, Image Classification, Intel Analytics Zoo, Real-time Analytics, Spark Streaming

Subscribe

Third Wave Blog

Megh Computing News

Megh Computing

Enabling the 3rd Wave of Computing in the Data Center

Home

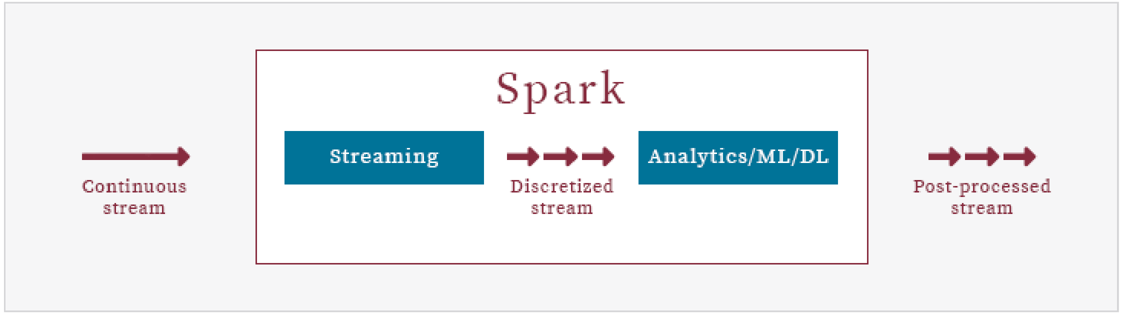
Products

```
def bytesToImageObjects(inputStream: InputStream): Iterator[Ima

  val imageNameLen = 28
  val dataInputStream = new DataInputStream(is)
  val sw = new StringWri

  def getNext(): Unit = {
    logger.info("Start get next")
```

About

With the current information age defining the third wave, we are facing an explosion of real-time data, which is in turn increasing demand for real-time analytics. A real-time analytics solution pipeline typically utilizes a streaming library and an analytics platform.

Apache Spark is an open-source, distributed computing platform designed to run analytics payloads on a cluster of computing nodes. Spark provides a unified framework to build an end-to-end pipeline of extract, transform, and load (ETL) tasks. The Spark Streaming library supports ingestion of live data streams from sources like TCP sockets, Kafka, Apache Storm, and Flink. The data streams are read into DStreams, discretized micro batches of resilient distributed datasets (RDDs). The RDDs are then fed to the Spark engine for further processing.



Spark DStream pipeline

Spark core includes libraries like SQL and ML-Lib that support data analytics and machine learning processing pipelines. Intel's BigDL-based Analytics Zoo library seamlessly integrates with Spark to support deep learning payloads. BigDL leverages Spark's resource and cluster management.

To illustrate enabling a real-time streaming, deep learning pipeline, let's experiment with the Image classification example provided with Analytics Zoo. The sample implementation deals with computation in batch mode. We modified it to handle a continuous stream of images to fit our target use cases. Our implementation involves streaming pipelines targeting both CPU-only and FPGA + CPU platforms.

We implemented the socketStream connector to receive image streams transmitted through the TCP socket for our CPU implementation.

```
val ssc = new StreamingContext(sc, new Duration(batchDur
val imageDStream = ssc.socketStream(host, port, bytesToI
```

We also created a custom converter function to transform the byte array to custom objects. The snippet below demonstrates implementation of the getNext method of the converter function, which extends the Iterator class.

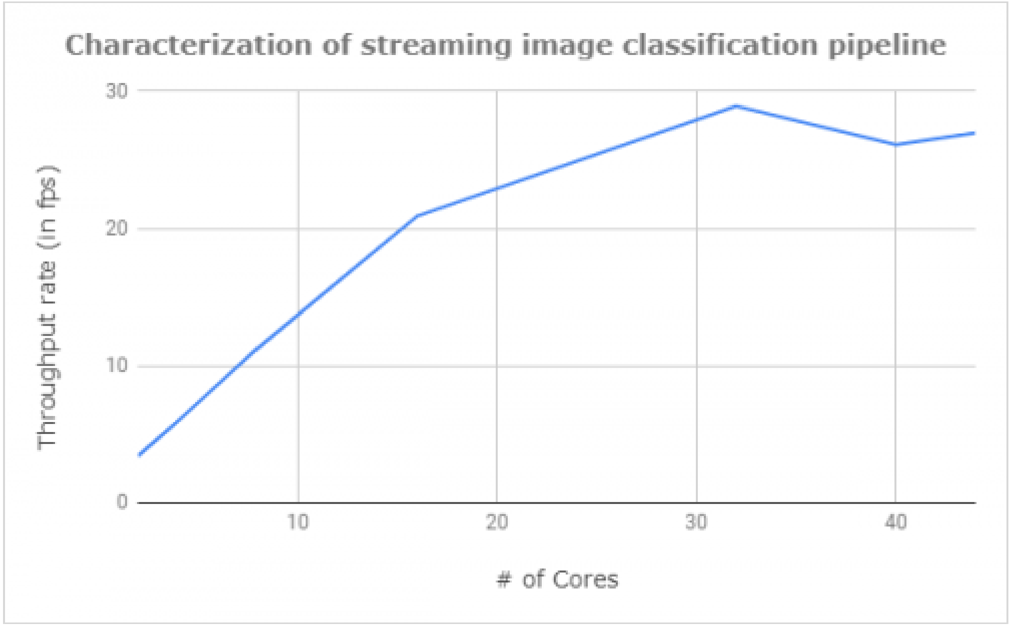
```
def bytesToImageObjects(inputStream: InputStream): Itera

  val imageNameLen = 28
  val dataInputStream = new DataInputStream(is)
  val sw = new StringWriter

  def getNext(): Unit = {
    logger.info("Start get next")
    try {
      logger.info("Start reading record")
      val name = new Array[Byte](imageNameLen)
      val imageName = new String(name)
      val imgLen = new Array[Byte](4)
      dis.readFully(name, 0, imageNameLen)
      dis.readFully(imgLen, 0, 4)
```

```
        val len = ByteBuffer.wrap(imgLen).getInt()
        val data = new Array[Byte](len)
        dis.readFully(data, 0, len)

        try {
          nextValue = ImageFeature(data, uri = imageName)
          if (nextValue.bytes() == null) {
            logger.info("Next value empty!")
            finished = true
            dis.close()
            return
          }
        }
        catch {
          case e: Exception => e.printStackTrace(new Print
          finished = true
          dis.close()
          logger.error(sw.toString())
        }
      }
      catch {
        case e: Exception => e.printStackTrace(new PrintWr
        finished = true;
      }
      gotNext = true
    }
  }
```

Application stability and pipeline performance varied considerably depending on:

- Batch duration
- Number of cores
- Number of executors
- RDD partitions

Early numbers we observed on characterizing the image inference pipeline demonstrate scaling in performance with increase in number of cores per executor.



For the FPGA + CPU platform we implemented the receiverStream connector to receive the image stream from the FPGA through a custom receiver.

```
val ssc = new StreamingContext(sc, new Duration(batchDur
val imgDStream = ssc.receiverStream(new meghImageReceive
```

The custom receiver interfaces with Megh Computing's proprietary runtime libraries to receive the byte stream from the FPGA NIC. For this implementation, the Spark application layer APIs themselves need zero code change.

We will share details on our custom CPU + FPGA pipeline implementation and our performance comparisons between CPU-only and CPU + FPGA platforms in a future post. We'll leave you with a few good references for the custom converter function and receiver implementations:

- https://spark.apache.org/docs/2.3.1/streaming-custom-receivers.html

- https://github.com/apache/spark/blob/master/streaming/src/main/scala/org/apache/spark/streaming/dstream/SocketInputDStream.scala
- https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/util/NextIterator.scala

Stay tuned for part II.

Deep Learning, Image Classification, Intel Analytics Zoo, Real-time Analytics, Spark Streaming

Megh Computing, Inc.

All posts by this author

Previous
Megh Computing secures $1.75 mill…

Next
AI technology press affirming Megh'…

Share this page

Facebook Google+ Twitter LinkedIn

Megh Computing
Enabling the 3rd Wave of Computing in the Data Center

1600 NE Compton Drive
Suite 202
Hillsboro, Oregon, 97006
USA

Contact us
Find us on LinkedIn
Find us on Twitter

Privacy Policy