Best Practices

Accelerating Spark ML Applications

Date published: 2020-01-16
Date modified:

https://docs.cloudera.com/

Legal Notice

Contents

Overview

Spark ML is one of the dominant frameworks for many major machine learning algorithms, such as the Alternating Least Squares (ALS) algorithm for recommendation systems, the Principal Component Analysis algorithm, and the Random Forest algorithm. However, frequent misconfiguration means the potential of Spark ML is seldom fully utilized. Using native math libraries for Spark ML is a method to achieve that potential.

This article discusses how to accelerate model training speed by using native libraries for Spark ML. In addition, it discusses why Spark ML benefits from native libraries, how to enable the native libraries with CDH Spark, and provides performance comparisons between Spark ML on different native libraries.

Native math libraries for Spark ML

Spark's MLlib uses the Breeze linear algebra package, which depends on netlib-java for optimized numerical processing. netlib-java is a wrapper for low-level BLAS, LAPACK, and ARPACK libraries.

By: Zuling Kang, Senior Solutions Architect at Cloudera, Inc.

Although Spark's MLlib can use these libraries, due to licensing issues with runtime proprietary binaries, neither the Cloudera distribution of Spark nor the community version of Apache Spark includes the netlib-java native proxies by default. So if you make no manual configuration, netlib-java only uses the F2J library, a Java-based math library that is translated from Fortran77 reference source code.

To check whether you are using native math libraries in Spark ML or the Java-based F2J, use the Spark shell to load and print the implementation library of netlib-java. For example, the following commands return information on the BLAS library and include that it is using F2J in the line, com.github.fommil.netlib.F2jBLAS, which is bolded below:

```
scala> import com.github.fommil.netlib.BLAS
import com.github.fommil.netlib.BLAS

scala> println(BLAS.getInstance().getClass().getName())
18/12/10 01:07:06 WARN netlib.BLAS: Failed to load implementation from: co
m.github.fommil.netlib.NativeSystemBLAS
18/12/10 01:07:06 WARN netlib.BLAS: Failed to load implementation from: com.
github.fommil.netlib.NativeRefBLAS
com.github.fommil.netlib.F2jBLAS
```

Adoption of Spark ML native math libraries

The range of acceleration provided by native libraries varies from model to model.

By: Zuling Kang, Senior Solutions Architect at Cloudera, Inc.

Anand Iyer and Vikram Saletore showed in their engineering blog post that native math libraries like OpenBLAS and Intel's Math Kernel Library (MKL) accelerate the training performance of Spark ML. As for the matrix factorization model used in recommendation systems (the Alternating Least Squares (ALS) algorithm), both OpenBLAS and Intel's MKL yield model training speeds that are 4.3 times faster than with the F2J implementation. Others, like the Latent Dirichlet Allocation (LDA), the Primary Component Analysis (PCA), and the Singular Value Decomposition (SVD) algorithms show 56% to 72% improvements for Intel MKL, and 10% to 50% improvements for OpenBLAS.

However, the blog post also demonstrates that there are some algorithms, like Random Forest and Gradient Boosted Tree, that receive almost no speed acceleration after enabling OpenBLAS or MKL. The reasons for this are largely that the training set of these tree-based algorithms are not vectors. This indicates that the native libraries, either OpenBLAS or MKL, adapt better to algorithms whose training sets can be operated on as vectors and are computed

as a whole. It is more effective to use math acceleration for algorithms that operate on training sets using matrix operations.

Enabling the libgfortran library

This procedure shows how to use Cloudera Manager to enable the libgfortran math library to accelerate Spark ML applications.

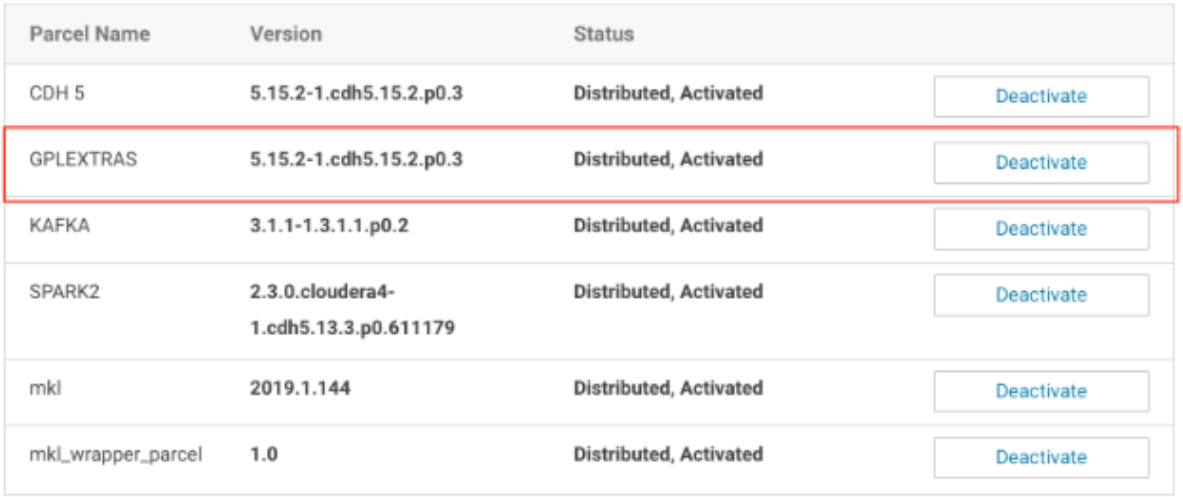
By: Zuling Kang, Senior Solutions Architect at Cloudera, Inc.

Important: The following instructions work for Spark 1.6x in CDH 5.x and for Spark 2.x in CDH 6.x. In those versions, GPLEXTRAS automatically adds the classpath of JAR files needed to use the native libraries to /etc/spark/conf/classpath.txt. Then the spark-env.sh loads the extra Java libraries during bootstrap. However, GPLEXTRAS cannot do this for Spark 2.x in CDH 5.x. If you want to use Spark 2.x, you must upgrade to CDH 6.x, which automatically performs this configuration for Spark 2.x. CDH 5.15 on RHEL 7.4 was used in the following example.

1. Enable the libgfortran 4.8 library on every CDH node. For example, in RHEL, run the following command on each node:

```
yum -y install libgfortran
```

2. Install the GPLEXTRAS parcel in Cloudera Manager, and activate it:
   a. To install the GPLEXTRAS parcel, see Installing the GPL Extras Parcel in the Cloudera Manager documentation.
   b. To activate the package, see Activating a Parcel.
   c. After activating the GPLEXTRAS parcel, in Cloudera Manager, navigate to HostsParcels to confirm that the GPLEXTRAS parcel is activated:



   The GPLEXTRAS parcel acts as the wrapper for libgfortran.
3. Restart the appropriate CDH services as guided by Cloudera Manager.
4. As soon as the restart is complete, use the Spark shell to verify that the native library is being loaded by using the following commands:

```
scala> import com.github.fommil.netlib.BLAS
import com.github.fommil.netlib.BLAS

scala> println(BLAS.getInstance().getClass().getName())
18/12/23 06:29:45 WARN netlib.BLAS: Failed to load implementation from: co
m.github.fommil.netlib.NativeSystemBLAS
```

```
18/12/23 06:29:45 INFO jni.JniLoader: successfully loaded /tmp/jniloader
6112322712373818029netlib-native_ref-linux-x86_64.so
com.github.fommil.netlib.NativeRefBLAS
```

You might be wondering why there is still a warning message about NativeSystemBLAS failing to load. Don't worry about this. It is there because we are only setting the native library that Spark uses, not for system-wide use. You can safely ignore this warning.

Enabling the Intel MKL library

This procedure shows how to use Cloudera Manager to enable the Intel MKL math library to accelerate Spark ML applications.

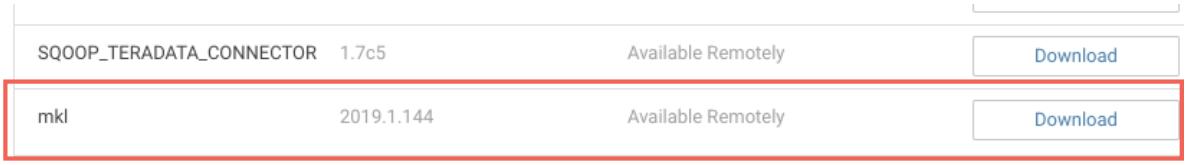
By: Zuling Kang, Senior Solutions Architect at Cloudera, Inc.

Important: The following instructions work for Spark 1.6x in CDH 5.x and for Spark 2.x in CDH 6.x. In those versions, GPLEXTRAS automatically adds the classpath of JAR files needed to use the native libraries to /etc/spark/conf/classpath.txt. Then the spark-env.sh loads the extra Java libraries during bootstrap. However, GPLEXTRAS cannot do this for Spark 2.x in CDH 5.x. If you want to use Spark 2.x, you must upgrade to CDH 6.x, which automatically performs this configuration for Spark 2.x. CDH 5.15 on RHEL 7.4 was used in the following example.

1. Intel provides the MKL native library as a Cloudera Manager parcel on its website. You can add it as a remote parcel repository in Cloudera Manager. Then you can download the library and activate it:
   a. In Cloudera Manager, navigate to HostsParcels .
   b. Select Configuration.
   c. In the section, Remote Parcel Repository URLs, click the plus sign and add the following URL:

      ```
      http://parcels.repos.intel.com/mkl/latest
      ```

   d. Click Save Changes, and then you are returned to the page that lists available parcels.
   e. Click Download for the mkl parcel:



   f. Click Distribute, and when it finishes distributing to the hosts on your cluster, click Activate.
2. The MKL parcel is only composed of Linux shared library files (.so files), so to make it accessible to the JVM, a JNI wrapper has to be made. To make the wrapper, use the following MKL wrapper parcel. Use the same procedure described in Step 1 to add the following link to the Cloudera Manager parcel configuration page, download the parcel, distribute it among the hosts and then activate it:

   ```
   https://raw.githubusercontent.com/Intel-bigdata/mkl-wrappers-parcel-repo/
   master/
   ```

3. Restart the corresponding CDH services as guided by Cloudera Manager, and redeploy the client configuration if needed.
4. In Cloudera Manager, add the following configuration information into the Spark Client Advanced Configuration Snippet (Safety Valve) for spark-conf/spark-defaults.conf:

   ```
   spark.driver.extraJavaOptions=-Dcom.github.fommil.netlib.BLAS=com.intel.
   mkl.MKLBLAS -Dcom.github.fommil.netlib.LAPACK=com.intel.mkl.MKLLAPACK
   spark.driver.extraClassPath=/opt/cloudera/parcels/mkl_wrapper_parcel/lib/j
   ava/mkl_wrapper.jar
   ```

```
spark.driverEnv.MKL_VERBOSE=1
spark.executor.extraJavaOptions=-Dcom.github.fommil.netlib.BLAS=com.intel.
mkl.MKLBLAS -Dcom.github.fommil.netlib.LAPACK=com.intel.mkl.MKLLAPACK
spark.executor.extraClassPath=/opt/cloudera/parcels/mkl_wrapper_parcel/lib
/java/mkl_wrapper.jar
spark.executorEnv.MKL_VERBOSE=1
```

This configuration information instructs the Spark application to load the MKL wrapper and use MKL as the default native library for Spark ML.

Important:

- By setting MKL_VERBOSE=1, MKL logs what computational functions are called, what parameters are passed to them, and how much time is spent to execute the functions. This information can be useful for implementation, but it consumes large amounts of space on HDFS in your cluster. In my experimental cases that are discussed in the following section, the logs for each job could consume hundreds of GBs of space.
- If the UnsatisfiedLinkError message is returned when verifying the native library being used as shown below, add the /opt/cloudera/parcels/mkl/linux/mkl/lib/intel64 directory to the LD_LIBRARY_PATH environment variable for each cluster node.

```
Native code library failed to load.
java.lang.UnsatisfiedLinkError: /opt/cloudera/parcels/mkl_wrappe
r_parcel-1.0/lib/native/mkl_wrapper.so: libmkl_rt.so: cannot open
 shared object file: No such file or directory
```

5. Open the Spark shell again to verify the native library, and you should see the following output:

```
scala> import com.github.fommil.netlib.BLAS
import com.github.fommil.netlib.BLAS

scala> println(BLAS.getInstance().getClass().getName())
com.intel.mkl.MKLBLAS
```

Performance comparisons

In this topic, we use the ALS algorithm to compare the training speed with different underlying math libraries, including F2J, libgfortran, and Intel's MKL.

By: Zuling Kang, Senior Solutions Architect at Cloudera, Inc.

The hardware we are using are the r4.large VM instances from Amazon EC2, with 2 CPU cores and 15.25 GB of memory for each instance. In addition, we are using CentOS 7.5 and CDH 5.15.2 with the Cloudera Distribution of Spark 2.3 Release 4. The training code is taken from the core part of the ALS chapter of Advanced Analytics with Spark (2nd Edition) by Sandy Ryza, et al, O'Reilly (2017). The training data set is the one published by Audioscrobbler, which can be downloaded at:

```
https://storage.googleapis.com/aas-data-sets/profiledata_06-May-2005.tar.gz
```

Usually the rank of the ALS model is set to a much larger value than the default of 10, so we use the value of 200 here to make sure that the result is closer to real world examples. Below is the code used to set the parameter values for our ALS model:

```
val model = new ALS().
     setSeed(Random.nextLong()).
     setImplicitPrefs(true).
     setRank(200).
     setRegParam(0.01).
     setAlpha(1.0).
```

```
      setMaxIter(20).
      setUserCol("user").
      setItemCol("artist").
      setRatingCol("count").
      setPredictionCol("prediction")
```

The following table and figure shows the training time when using different native libraries of Spark ML. The values are shown in minutes. We can see that both libgfortran and Intel's MKL do improve the performance of training speed, and MKL seems to outperform even more. From these experimental results, libgfortran improves by 18% to 68%, while MKL improves by 92% to 213%.

# of Workers/Executors	F2J	libgfortran	Intel MKL
3 workers (9 executors)	426	360	222
9 workers (26 executors)	282	168	90