Using shifter to run tensorflow 1.12 on Blue Waters

Compiling TensorFlow can be non-trivial in particular on a system not directly supported like Blue Waters. There are a number of challenges that one faces:

- TensorFlow requires a number of prerequisites to be installed, which themselves are not trivial to install
- TensorFlow requires a newer version of glibc than may be installed on Blue Waters
- TensorFlow's build system is somewhat unusual and not well supported on Blue Waters
- Binary packages cannot be used since they use CPU instructions not supported by Blue Waters' CPUs

There are however decent instructions on how to compile TensorFlow on Ubuntu 16.04 LTS which can be run on Blue Waters inside of Shifter. This document contains files and instructions to build TensorFlow using

- TensorFlow 1.12 which is the last version for which Google builds their default binary using CUDA 9.1
- cuDNN 7.5
- nccl 2.4
- mpich and the Cray MPI-ABI layer

Using Docker to build the image

Since Shifter uses Docker images you will need to install Docker on a machine where you have sufficient privileges to run the docker daemon (usually this means that you can become `root`).

Downloading binary codes

NVIDIA requires that one registers before downloading CUDA, cuDNN and nccl which makes it impractical to download them as part of an automated build, thus the first step is to download `cuda_9.1.85.2_linux`, `cuda_9.1.85.1_linux`, `cuda_9.1.85.3_linux`, `cuda_9.1.85_387.26_linux`, `cudnn-9.0-linux-x64-v7.5.0.56.tgz` and `nccl-repo-ubuntu1604-2.4.2-ga-cuda9.0_1-1_amd64.deb` from the NVIDIA servers.

Installing CUDA drivers and NVIDIA software

NIVIDIA's software is large enough that it is worthwhile to build a temporary image to install it which we then pilfer for the interesting files afterwards:

NVIDIA software

```
# Distributed under the MIT License.
# See LICENSE.txt for details.

 FROM ubuntu:16.04 as intermediate

 COPY cudnn-9.0-linux-x64-v7.5.0.56.tgz nccl-repo-ubuntu1604-2.4.2-ga-cuda9.0_1-1_amd64.deb cuda_9.1.85*_linux
/tmp/

 RUN apt-get update -y && \
   apt-get install -y curl build-essential && \
   apt-get clean

# NVidia's deb packages put files in places where tensorflow does not find them, fix this manually
RUN cd /tmp && \
   ( test -r cuda_9.1.85_387.26_linux || \
     curl -L -O https://developer.nvidia.com/compute/cuda/9.1/Prod/local_installers/cuda_9.1.85_387.26_linux )
&& \
   ( test -r cuda_9.1.85.1_linux || \
     curl -L -O https://developer.nvidia.com/compute/cuda/9.1/Prod/patches/1/cuda_9.1.85.1_linux ) && \
   ( test -r cuda_9.1.85.2_linux || \
     curl -L -O https://developer.nvidia.com/compute/cuda/9.1/Prod/patches/2/cuda_9.1.85.2_linux ) && \
   ( test -r cuda_9.1.85.3_linux || \
     curl -L -O https://developer.nvidia.com/compute/cuda/9.1/Prod/patches/3/cuda_9.1.85.3_linux ) && \
   bash ./cuda_9.1.85_387.26_linux --toolkit --override --silent && \
   bash ./cuda_9.1.85.1_linux --silent --accept-eula && \
   bash ./cuda_9.1.85.2_linux --silent --accept-eula && \
   bash ./cuda_9.1.85.3_linux --silent --accept-eula && \
   rm cuda_9.1.85_387.26_linux cuda_9.1.85.?_linux && \
   apt-get install -y ./nccl-repo-ubuntu1604-2.4.2-ga-cuda9.0_1-1_amd64.deb && \
   apt-get update -y && \
   apt-get install -y --allow-unauthenticated libnccl2=2.4.2-1+cuda9.0 libnccl-dev=2.4.2-1+cuda9.0 && \
   mv /usr/lib/x86_64-linux-gnu/libnccl.* /usr/local/cuda/lib64 && \
   mv /usr/include/nccl.h /usr/local/cuda/include && \
   apt-get remove -y nccl-repo-ubuntu1604-2.4.2-ga-cuda9.0 && \
   apt-get clean && \
   rm nccl-repo-ubuntu1604-2.4.2-ga-cuda9.0_1-1_amd64.deb && \
   tar xvf cudnn-9.0-linux-x64-v7.5.0.56.tgz && \
   cp -a cuda/include/cudnn.h /usr/local/cuda/include && \
   cp -a cuda/lib64/libcudnn* /usr/local/cuda/lib64/ && \
   chmod a+r /usr/local/cuda/include/cudnn.h && \
   rm -r cudnn-9.0-linux-x64-v7.5.0.56.tgz cuda
```

there is nothing very special in here, other than the script accepting most of the driver and CUDA files (which are freely downloadable but large) as either being already present on the host or downloadable. The Docker file moves everything to `/usr/local/cuda` including the nccl and cuDN files which makes it easier to pilfer them later. Note that the CUDA version matches exactly the one installed on Blue Waters. This is not required but helpful if one wants to compile in the container but then use Blue Waters' system CUDA.

Installing TensorFlow prerequisites

The second part of the Docker file downloads and installs everything else required to build TensorFlow. Note that we will not build TensorFlow on the host used to prepare the Docker image. Instead we will use the image to compile TensorFlow on Blue Waters taking advantage of gcc's `-march=native` flag to use optimized instructions.

TensorFlow build prerequisites

```
FROM ubuntu:16.04

# CUDA is not compatible with gcc 6 by default, BW gcc 6.3 has a fix but that
# does not help us, plus there's no gcc-6 for xenial
RUN apt-get update -y && \
    apt-get install -y --no-install-recommends \
                       gcc-4.9 g++-4.9 gfortran-4.9 libopenblas-base build-essential zsh \
                       mpich2? libmpich2?-dev libhugetlbfs-dev nvidia-modprobe && \
    apt-get clean && \
    mkdir -p /opt/cray /work

# /usr/bin/python is a symlink to python2 on xenial, but since I do not install
# python-minimal I am missing that link
RUN apt-get update -y && \
  apt-get install -y --no-install-recommends curl git unzip openjdk-8-jdk libcurl3-dev \
    python3-dev python3-numpy python3-pip python3-virtualenv python3-setuptools python3-six python3-mock swig
python3-wheel && \
  apt-get clean  && \
  ln -s python3 /usr/bin/python

# it should be possible to avoid installing any CUDA and just use blue waters
# as long as nothing depends on it, though cudnn may not be present on BW
# Tensorflow expects to find libmpi.so in MPI_HOME but mpich names this libmpich.so
COPY --from=intermediate /usr/local/ /usr/local/
RUN echo "/usr/local/cuda/lib64" > /etc/ld.so.conf.d/cuda.conf && \
  echo "/usr/local/cuda/lib64/stubs" >> /etc/ld.so.conf.d/cuda.conf && \
  ln -s libcuda.so /usr/local/cuda/lib64/stubs/libcuda.so.1 && \
  ln -s /usr/lib/libmpi.so /usr/lib/mpich/lib/ && \
  ldconfig

RUN pip3 install --no-cache-dir keras_applications==1.0.6 keras_preprocessing==1.0.5 --no-deps

# bazel 0.19.1 will need a bugfix https://github.com/tensorflow/tensorflow/pull/25114/files
# https://github.com/tensorflow/tensorflow/issues/23401
RUN cd /tmp && \
  curl -O -L https://github.com/bazelbuild/bazel/releases/download/0.18.1/bazel-0.18.1-installer-linux-x86_64.
sh && \
  bash bazel-0.18.1-installer-linux-x86_64.sh && \
  rm bazel-0.18.1-installer-linux-x86_64.sh

RUN pip3 install --no-cache-dir mpi4py==1.
3.1
```

Things to note:

- we pilfer `/usr/local` from the intermediary image so that we are not stuck with (any more) large Docker layers
- we create a mount points `/opt/cray` and `/work` which we will later use to inject Cray's custom libraries into the image and as a work area during build respectively
- we install gcc 4.9 since this compiler is supported on BW and by Ubuntu 16.04 LTS and does not have the bug that gcc 6.3 has when vectorizing code for `-march=bdver1`
- we install MPICH rather than `OpenMPI` since Cray's MPI stack is based on MPI and is binary API compatible with it
- we needs to fudge some files using symbolic links so that TensorFlow's configure script finds them, namely
    - python
    - libcuda.so
    - libmpi.so
- add CUDA library paths to `ld.so.conf`
- we install `mpi4py` 1.3.1 mostly because this is the same version as provided by Ubuntu 16.04

Build the image and push to Dockerhub

Since Shifter expects its images to be on Dockerhub, we now must build and push the image. The Blue Waters Shifter page contains instructions on how to use private images instead.

building the image

```
sudo docker build -t $USER/tensorflow:16.04 -f Dockerfile.tensorflow .
sudo docker push $USER/tensorflow:16.04
```

The full Dockerfile can be found here.

Compiling TensorFlow on Blue Waters

Using the prepared image we can build TensorFlow on Blue Waters. Please refer to the Shifter documentation for generic information on how to use Shifter on Blue Waters. We use an interactive session on an XK node to compile and test TensorFlow:

interactive shifter session

```
qsub -I -l nodes=1:x:ppn=16 -l walltime=3:00:00 -l gres=shifter
module load shifter
shifterimg pull $USER/tensorflow:16.04
```

This will take a while to pull the image from DockerHub and convert it to a Shifter image.

Interactive logins to shifter containers

TensorFlow's configure script is designed for interactive use. To gain interactive access to the container one can have Shifter start an ssh daemon in it (this is documented in the Blue Waters Shifter documentation):

sshd

```
# note the lowercase -v after UDI to mount volumes
qsub -I -l nodes=1:x:ppn=16 -l walltime=3:00:00 -l gres=shifter -v UDI="$USER/tensorflow:16.04 -v /dev
/shm:/work"

export CRAY_ROOTFS=SHIFTER
aprun -b -n 1 -N 1 -d 16 -cc none /bin/bash -c 'Connect to: $(hostname) ; sleep 86400' &

ssh -F $HOME/.shifter/config nidXXXX

./configure
```

where `nidXXXXX` is the name of the compute node output by the aprun command (which sleeps so that the container stays around) and use this to interactively configure TensorFlow.

The alternative method used by me here is to do a scripted build by setting the environment variables that the configure script is looking for. Thus in this example we are using a script to drive the compilation since the interactive configure script cannot be easily used with shifter and aprun due to the lack of a full login capability. The script for the most part follow's Google's instructions closely.

Since building TensorFlow is quite IO intensive it is best to build it in a RAM disk in /dev/shm on the compute node. To this end we mount the `/dev/shm` shared memory file system onto the `/work` directory prepared before.

aprun

```
# note the uppercase -V option to mount volumes
aprun -b -n 1 -N 1 -d 16 -cc none -- shifter --image=$USER/tensorflow:16.04 -V /dev/shm:/work -- bash $PWD
/compile_tensorflow.sh
```

We first switch to the /work directory and check out the last release of TensorFlow still compatible with CUDA 9.1 (release r1.12):

checkout

```
cd /work
git clone https://github.com/tensorflow/tensorflow.git
cd tensorflow
git checkout r1.12
```

Next, the script has to configure TensorFlow. The settings shown below are based on a mixture of starting from someone else's work, running installation once in a Docker image on my laptop (where I can get a proper terminal more easily) and inspecting the configure.py script.

configure

```
# set up env variables so that configure does not actually ask any questions
# skeleton from https://gist.github.com/PatWie/0c915d5be59a518f934392219ca65c3d
# actual numbers from compiling locally to be able to respond to interactive
# prompt, then (mostly) from .tf_configure.bazelrc

export PYTHON_BIN_PATH=/usr/bin/python3
export PYTHON_LIB_PATH="$($PYTHON_BIN_PATH -c 'import site; print(site.getsitepackages()[0])')"
export CUDA_TOOLKIT_PATH=/usr/local/cuda
export CUDNN_INSTALL_PATH=/usr/local/cuda-9.1
export NCCL_INSTALL_PATH=/usr/local/cuda/lib64

export TF_NEED_GCP=0
export TF_NEED_CUDA=1
export TF_CUDA_VERSION=9.1
export TF_CUDA_COMPUTE_CAPABILITIES=3.5
export TF_NEED_IGNITE=0
export TF_NEED_ROCM=0
export TF_NEED_HDFS=0
export TF_NEED_OPENCL=0
export TF_NEED_JEMALLOC=1
export TF_ENABLE_XLA=0
export TF_NEED_VERBS=0
export TF_CUDA_CLANG=0
export TF_CUDNN_VERSION=7
export TF_NEED_MKL=0
export TF_DOWNLOAD_MKL=0
export TF_NEED_AWS=0
export TF_NEED_MPI=1
export MPI_HOME=/usr/lib/mpich
export TF_NEED_GDR=0
export TF_NEED_S3=0
export TF_NEED_OPENCL_SYCL=0
export TF_SET_ANDROID_WORKSPACE=0
export TF_NEED_COMPUTECPP=0
export GCC_HOST_COMPILER_PATH=/usr/bin/gcc
export CC_OPT_FLAGS="-march=native"
export TF_NEED_KAFKA=0
export TF_NEED_TENSORRT=0
export TF_NCCL_VERSION=2

export GCC_HOST_COMPILER_PATH=$(which gcc)
export CC_OPT_FLAGS="-march=native"

PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin

./configure
```

At long last we are able to build TensorFlow. Before starting the build process it is advisable though to redirect bazel's cache from `$HOME/.cache` to our work directory to keep IO requests away from the (slower) Lustre file system and redirect them to `/work` (fast since in `/dev/shm`).

build

```
# clean up env and PATH to use only data in the container
unset JAVA_HOME SDK_HOME JDK_HOME JAVA_ROOT JAVA_BINDIR
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin

mkdir -p /work/bazel_cache $HOME/.cache
ln -sf /work/bazel_cache  $HOME/.cache/bazel

bazel build  --config=opt --config=cuda //tensorflow/tools/pip_package:build_pip_package
./bazel-bin/tensorflow/tools/pip_package/build_pip_package /work/
```

Finally we clean up and copy the produced tensorflow wheel to a safe location.

copy

```
rm $HOME/.cache/bazel

mkdir -p $PBS_O_WORKDIR/packages/
cp /work/tensorflow-1.12.1-cp35-cp35m-linux_x86_64.whl $PBS_O_WORKDIR/packages/
```

The full compile script as well as a pbs script to submit via qsub can be found here and here. The final pip wheel file is tensorflow-1.12.1-cp35-cp35m-linux_x86_64.whl.

Installing TensorFlow on Blue Waters

Having successfully built a TensorFlow wheel on Blue Waters it can be installed in a virtualenv spun off from the python3 installation in the container.

install tensorflow

```
mkdir tensorflow
cd tensorflow
/usr/bin/python3 -m virtualenv --system-site-packages --no-download -p /usr/bin/python3 $PWD
source bin/activate
pip3 install numpy==1.13.3 h5py==2.7.1 grpcio==1.8.6
pip3 install ../packages/tensorflow-1.12.1-cp35-cp35m-linux_x86_64.whl
```

These commands need to execute inside the container for example by putting them into a script file `install.sh` and using aprun:

aprun to install wheel

```
#!/bin/bash
#PBS -l nodes=1:xk:ppn=16
#PBS -l walltime=0:10:0
#PBS -l gres=shifter

cd $PBS_O_WORKDIR

module load shifter

aprun -b -n 1 -N 1 -d 16 -cc none -- shifter --image=rhaas/tensorflow:16.04s -V $(pwd -P):/work -V /dsl/opt
/cray:/opt/cray -- /bin/bash ./install.sh
```

Test

These tests showcase how to use the container and tensorflow. We will run them using a somewhat more complex invocation of shifter to link the Cray libraries to the container using the `/opt/cray` mount point. We can obtain a limited interactive shell inside of the container:

complex aprun

```
#!/bin/bash
#PBS -l nodes=1:xk:ppn=16
#PBS -l walltime=0:10:0
#PBS -l gres=shifter

cd $PBS_O_WORKDIR

module load cudatoolkit
module unload PrgEnv-cray
module load PrgEnv-gnu
module load cray-mpich-abi
module load shifter

export CUDA_VISIBLE_DEVICES=0

export TF_LD_LIBRARY_PATH="/work/tensorflow/lib:$(readlink -f /opt/cray/wlm_detect/default/lib64):$(readlink -f
/opt/cray/nvidia/default/lib64):/usr/local/cuda/lib64:$LD_LIBRARY_PATH:$CRAY_LD_LIBRARY_PATH"

aprun -b -n 1 -N 1 -d 16 -cc none -- shifter --image=rhaas/tensorflow:16.04s -V $(pwd -P):/work -V /dsl/opt
/cray:/opt/cray -- /bin/bash -i
```

Here's the breakdown of the settings

- we load the modules to provide `cudatoolkit` and `cray-mpich-abi` which requires the use of the GNU compiler environment
- we make the GPU visible using `CUDA_VISIBLE_DEVICE=0`
- we need to capture and extend `LD_LIBRARY_PATH` to include
    - our private lib directory in the virtualenv (more on that later)
    - helper libraries for MPI (`wlm_detect`)
    - CUDA
    - the mpich-abi libraries (in `CRAY_LD_LIBRARY_PATH`)
- the aprun line binds `/dsl/opt/cray` to `/opt/cray` in the container using Cray's software directory
- `LD_LIBRARY_PATH` needs to be reset since shifter clears it (it is an setuid executable)
- start an interactive shell in the container (`bash -i`)

Provided tests

I provide a set of simple tests for the setup

- mpi.c which outputs the MPI library identification string
    - newer version of mp4py expose the same functionality in `mpi4py.MPI.Get_library_version()`
- cuda.c which output the CUDA runtime and driver version
- vecAdd.cu which perform a GPU assisted vector addition
- word2vec.py which is TensorFlow's word2vec example code ported to python3
- simpleMPI which showcases the use of MPI+CUDA on Blue Waters

Compiling the tests

Codes can be compiled in the container using regular configure/make options giving the libraries in the container as targets:

compile examples

```
cd /work/tests

# query MPI version
/usr/bin/gcc -L/usr/lib/mpich/lib -I/usr/lib/mpich/include mpi.c -o bin/mpi -lmpi

# query CUDA version
/usr/bin/gcc -L/usr/local/cuda/lib64 -L/usr/local/cuda/lib64/stubs -I/usr/local/cuda/include cuda.c -o bin/cuda
-lcuda -lcudart

# a simple CUDA code
/usr/local/cuda/bin/nvcc vecAdd.cu -o bin/vecAdd

# mixed CUDA+MPI
wget -N https://bluewaters.ncsa.illinois.edu/liferay-content/document-library/Documentation%20Documents
/simpleMPI.h
wget -N https://bluewaters.ncsa.illinois.edu/liferay-content/document-library/Documentation%20Documents
/simpleMPI.cpp
wget -N https://bluewaters.ncsa.illinois.edu/liferay-content/document-library/Documentation%20Documents
/simpleMPI.cu

/usr/bin/g++ -I/usr/lib/mpich/include -o simpleMPI.o -c simpleMPI.cpp
/usr/local/cuda/bin/nvcc -o simpleMPIcuda.o -c simpleMPI.cu
/usr/bin/g++ -L/usr/lib/mpich/lib -L/usr/local/cuda/lib64 -L/usr/local/cuda/lib64/stubs -o bin/simpleMPI
simpleMPI.o simpleMPIcuda.o -lmpi -lcuda -lcudart
```

which showcases how to use compilers inside of the container and how to pass the required libraries. Using `LD_LIBRARY_PATH` we will redirect the executables to use the "real" libraries at runtime.

MPI and CUDA

These tests show basic properties of MPI and CUDA to verify that indeed we are using Blue Water's MPI stack and CUDA drivers. When compiling MPI code in the container it will link against `/usr/lib/mpich/lib/libmpich.so.12` however Cray's compatibility libraries only provide `libmpi.so.12` so we fudge this by providing a symbolic link to the correct file using the correct name. Note that `$MPICH_DIR` is set by the `cray-mpich-abi` module, which needs to be loaded before `aprun` for this to work.

make MPI library visible under expected name

```
cd /work
ln -s $MPICH_DIR/lib/libmpi.so.12 tensorflow/lib/libmpich.so.12
```

Once done we can run the tests. Without `LD_LIBRARY_PATH` the mpi example will use the container's MPICH library and produce output like this:

mpi-MPICH

```
tests/bin/mpi

MPICH Version:  3.2
MPICH Release date:     Wed Nov 11 22:06:48 CST 2015
MPICH Device:   ch3:nemesis
MPICH configure:        --build=x86_64-linux-gnu --prefix=/usr --includedir=${prefix}/include --mandir=${prefix}
/share/man --infodir=${prefix}/share/info --sysconfdir=/etc --localstatedir=/var --disable-silent-rules --
libdir=${prefix}/lib/x86_64-linux-gnu --libexecdir=${prefix}/lib/x86_64-linux-gnu --disable-maintainer-mode --
disable-dependency-tracking --enable-shared --prefix=/usr --enable-fortran=all --disable-rpath --disable-
wrapper-rpath --sysconfdir=/etc/mpich --libdir=/usr/lib/x86_64-linux-gnu --includedir=/usr/include/mpich --
docdir=/usr/share/doc/mpich --with-hwloc-prefix=system --enable-checkpointing --with-hydra-ckpointlib=blcr
CPPFLAGS= CFLAGS= CXXFLAGS= FFLAGS= FCFLAGS=
MPICH CC:       gcc  -g -O2 -fstack-protector-strong -Wformat -Werror=format-security  -O2
MPICH CXX:      g++  -g -O2 -fstack-protector-strong -Wformat -Werror=format-security -O2
MPICH F77:      gfortran  -g -O2 -fstack-protector-strong -O2
MPICH FC:       gfortran  -g -O2 -fstack-protector-strong -O2
```

while with `LD_LIBRARY_PATH` set it uses Cray's MPI library:

mpi-CRAYMPI

```
export LD_LIBRARY_PATH=$TF_LD_LIBRARY_PATH
tests/bin/mpi

MPI VERSION    : CRAY MPICH version 7.5.0 (ANL base 3.2rc1)
MPI BUILD INFO : Built Fri Oct 21 13:57:53 2016 (git hash 1cb66d6) MT-G
```

Note that in neither case did we use `mpirun` (or `aprun`) inside of the container since we already used `aprun` to start the container.

Similar results can be obtained for CUDA:

CUDA

```
tests/bin/cuda

runtime: 9010 driver: 9010

tests/bin/vecAdd

final result: 1.000000
```

simpleMPI

The simpleMPI tests shows how one can combine MPI and CUDA on Blue Waters. The original example was for "bare metal" Blue Waters but works from inside of containers just as well:

run simpleMPI

```
#!/bin/bash
#PBS -l nodes=2:xk:ppn=16
#PBS -l walltime=0:30:0
#PBS -l gres=shifter

cd $PBS_O_WORKDIR

module load cudatoolkit
module unload PrgEnv-cray
module load PrgEnv-gnu
module load cray-mpich-abi
module load shifter

export CUDA_VISIBLE_DEVICES=0

TF_LD_LIBRARY_PATH="/work/tensorflow/lib:$(readlink -f /opt/cray/wlm_detect/default/lib64):$(readlink -f /opt
/cray/nvidia/default/lib64):/usr/local/cuda/lib64:$LD_LIBRARY_PATH:$CRAY_LD_LIBRARY_PATH"

NODES=$(sort -u $PBS_NODEFILE | wc -l)

aprun -b -n $NODES -N 1 -d 16 -cc none -- shifter --image=rhaas/tensorflow:16.04s -V $(pwd -P):/work -V /dsl/opt
/cray:/opt/cray -- bash -c "LD_LIBRARY_PATH=$TF_LD_LIBRARY_PATH tests/bin/simpleMPI"
```

which also showcases how to use more than 1 MPI rank. Note that this use of shifter is limited to 1 MPI rank per node which is usually what you want to do for GPU accelerated code. Please see the Shifter documentation for how to start multiple MPI ranks per node. Output of the simpleMPI test is

simpleMPI output

```
Running on 2 nodes
Average of square roots is: 0.667279
PASSED
```

Scripts to compile and run the tests

- compile_tests.sh compiles all tests from within a job compile_tests.pbs
- run_tests.pbs and run_tests.sh run all basic tests
- simpleMPI.pbs runs the simpleMPI CUDA test on 2 nodes

TensorFlow example

As a final example we will run the word2vec.py TensorFlow example using Shifter on a single BW GPU node. Extensions to using multiple nodes using e. g. the Cray Machine Learning plugin or Horovod are left as exercises to the reader. The qsub script is almost the same as for the basic tests with the only change being that `LD_LIBRARY_PATH` is immediately set up

TensorFlow submitscipt

```
#!/bin/bash
#PBS -l nodes=1:xk:ppn=16
#PBS -l walltime=0:30:0
#PBS -l gres=shifter

cd $PBS_O_WORKDIR

module load cudatoolkit
module unload PrgEnv-cray
module load PrgEnv-gnu
module load cray-mpich-abi
module load shifter

export CUDA_VISIBLE_DEVICES=0

TF_LD_LIBRARY_PATH="/work/tensorflow/lib:$(readlink -f /opt/cray/wlm_detect/default/lib64):$(readlink -f /opt
/cray/nvidia/default/lib64):/usr/local/cuda/lib64:$LD_LIBRARY_PATH:$CRAY_LD_LIBRARY_PATH"

aprun -b -n 1 -N 1 -d 16 -cc none -- shifter --image=rhaas/tensorflow:16.04s -V $(pwd -P):/work -V /dsl/opt
/cray:/opt/cray -- bash -c "LD_LIBRARY_PATH=$TF_LD_LIBRARY_PATH tests/tensorflow.sh"
```

with the call to TensorFlow wrapped into a shell script

TensorFlow invocation script

```
#!/bin/bash

source /work/tensorflow/bin/activate

export PYTHONIOENCODING=utf8
python3 -u tests/word2vec.py | tee word2vec.log
```

to activate the python virtualenv and set up IO encoding for python3. This is actually fairly slow even when using a GPU, and also does not make good use of the GPU (most likely b/c I did nothing to optimize the example).

word2vec output

```
Words count: 17005207
Unique words: 253854
Vocabulary size: 47135
Most common words: [('UNK', 444176), ('the', 1061396), ('of', 593677), ('and', 416629), ('one', 411764), ('in',
372201), ('a', 325873), ('to', 316376), ('zero', 264975), ('nine', 250430)]
2019-04-09 18:22:45.306922: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found device 0 with
properties:
name: Tesla K20X major: 3 minor: 5 memoryClockRate(GHz): 0.732
pciBusID: 0000:02:00.0
totalMemory: 5.57GiB freeMemory: 5.49GiB
2019-04-09 18:22:45.323894: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1511] Adding visible gpu
devices: 0
2019-04-09 18:22:49.634866: I tensorflow/core/common_runtime/gpu/gpu_device.cc:982] Device interconnect
StreamExecutor with strength 1 edge matrix:
2019-04-09 18:22:49.637807: I tensorflow/core/common_runtime/gpu/gpu_device.cc:988]      0
2019-04-09 18:22:49.637834: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 0:   N
2019-04-09 18:22:49.643831: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created TensorFlow device (
/job:localhost/replica:0/task:0/device:GPU:0 with 5278 MB memory) -> physical GPU (device: 0, name: Tesla K20X,
pci bus id: 0000:02:00.0, compute capability: 3.5)
Step 1, Average Loss= 540.6105
Evaluation...
"five" nearest neighbors: evening, routine, kinds, toole, deliberate, cahn, headline, hummer,
"of" nearest neighbors: fraught, unsolvable, pinpoint, ceases, locales, gladiator, taff, ie,
"going" nearest neighbors: slur, synchronised, narrowly, abkhaz, yana, wheatstone, oktoberfest, alces,
"hardware" nearest neighbors: atp, burned, foul, diodes, buffering, bhaskara, signifier, spinach,
"american" nearest neighbors: lambeth, minimising, technically, adornment, literals, infibulation, legionary,
micrometer,
"britain" nearest neighbors: infants, priori, lula, shapeshifting, cesare, dedications, consecrated, matisse,
Step 10000, Average Loss= 198.3677
Step 20000, Average Loss= 93.5416
Step 30000, Average Loss= 64.8307
Step 40000, Average Loss= 50.8407
Step 50000, Average Loss= 41.2500
Step 60000, Average Loss= 36.2364
Step 70000, Average Loss= 31.8867
Step 80000, Average Loss= 29.4877
Step 90000, Average Loss= 27.3099
Step 100000, Average Loss= 24.7121
Step 110000, Average Loss= 23.3436
Step 120000, Average Loss= 21.1763
Step 130000, Average Loss= 20.4042
Step 140000, Average Loss= 19.2406
Step 150000, Average Loss= 18.7244
Step 160000, Average Loss= 17.6856
Step 170000, Average Loss= 16.8901
Step 180000, Average Loss= 16.3079
Step 190000, Average Loss= 15.4208
Step 200000, Average Loss= 14.5649
Evaluation...
"five" nearest neighbors: four, three, seven, eight, six, two, nine, zero,
"of" nearest neighbors: the, first, were, a, as, was, with, from,
"going" nearest neighbors: it, was, used, be, been, is, known, system,
"hardware" nearest neighbors: sets, location, april, pre, three, five, arabic, four,
"american" nearest neighbors: UNK, and, about, s, in, when, first, from,
"britain" nearest neighbors: called, been, such, are, other, a, however, usually,
[...]
```

with the full output available here.

Combined archive of all scripts

All scripts and code fragments shown can be downloaded here and the pip wheel file from tensorflow-1.12.1-cp35-cp35m-linux_x86_64.whl.