Get the stuff…

- Get the examples and slides in 3 different ways:
  - git clone https://github.nrel.gov/tkaiser2/h100buildrun.git h100
  - git clone $USER@kestrel.hpc.nrel.gov:/nopt/nrel/apps/examples/gpu/0824 h100
  - tar -xzf /nopt/nrel/apps/examples/gpu/h100.tgz
- Tarball with output from the runs
  - tar -xzf /nopt/nrel/apps/examples/gpu/ran.tgz
- Just the slides:
  - scp $USER@kestrel.hpc.nrel.gov:/nopt/nrel/apps/examples/gpu/0824/slides.pdf slides.pdf
- Reservation for today:
  - gpututorial

Kestrel GPUs
Building and Running

Timothy H. Kaiser, Ph.D.
tkaiser2@nrel.gov
August(ish) 2024

Kestrel Hardware Overview

Number of Nodes	Processors	Memory	Accelerators	Local Storage
2304	Dual socket Intel Xeon Sapphire Rapids 52-core processors (104 cores total)	256 GB DDR5	N/A	256 nodes with 1.92 TB NVMe M.2
132	Dual socket AMD Genoa 64-core processors (128 cores total)	384 GB	4 NVIDIA H100 SXM GPUs, 80 GB Memory	2 x 1.6 TB NVMe
10	Dual socket Intel Xeon Sapphire Rapids 52-core processors (104 cores total)	2 TB DDR5	N/A	8 x 1.6 TB NVMe
8	Dual socket Intel Xeon Sapphire Rapids 52-core processors (104 cores total)	256 GB DDR5	2 NVIDIA A40 GPUs	2 x 3.84 TB NVMe

https://www.nrel.gov/hpc/kestrel-system-configuration.html

Kestrel Hardware Overview

Number of Nodes	Processors	Memory	Accelerators	Local Storage
2304	Dual socket Intel Xeon Sapphire Rapids 52-core processors (104 cores total)	256 GB DDR5	N/A	256 nodes with 1.92 TB NVMe M.2
132	Dual socket AMD Genoa 64-core processors (128 cores total)	384 GB	4 NVIDIA H100 SXM GPUs, 80 GB Memory	2 x 1.6 TB NVMe
10	Dual socket Intel Xeon Sapphire Rapids 52-core processors (104 cores total)	2 TB DDR5	N/A	8 x 1.6 TB NVMe
8	Dual socket Intel Xeon Sapphire Rapids 52-core processors (104 cores total)	256 GB DDR5	2 NVIDIA A40 GPUs	2 x 3.84 TB NVMe

https://www.nrel.gov/hpc/kestrel-system-configuration.html

Kestrel GPU login nodes

- kestrel-gpu.hpc.nrel.gov
  - kl5.hpc.nrel.gov
  - kl6.hpc.nrel.gov
- Compiles for GPU nodes should be done on GPU login or compute nodes
- GPU jobs should be submitted from GPU login nodes
- Will GPU jobs run from or apps built on CPU login node work?
  - Maybe
  - Maybe not
  - For you own sanity, just don't

NEW: gpu debug partiton

- One node with 1 GPU visible

```
salloc --account=XXXX --nodes=1 --time=01:00:00 --partition=debug-gpu --gres=gpu:h100:1
```

- One node with 2 GPU visible

```
salloc --account=XXXX --nodes=1 --time=01:00:00 --partition=debug-gpu --gres=gpu:h100:2
```

- Two nodes with 1 GPU visible per node

```
salloc --account=XXXX --nodes=2 --time=01:00:00 --partition=debug-gpu -gres=gpu:h100:1
```

Quick Start with Prgenv-cray

```
#!/bin/bash
#SBATCH –time=0:10:00
#SBATCH --partition=gpu-h100
#SBATCH --nodes=1
#SBATCH --gres=gpu:h100:4
#SBATCH --exclusive
#SBATCH --output=quick.out
#SBATCH –error=quick.out
…
```

This is also run as part of the full test set.

```
cat > hellof.f90 << END
!*********************************************
!  This is a simple hello world program. Each processor
!  prints out its name, rank and  number of processors
!  in the current MPI run.
!*********************************************
      program hello
      use iso_fortran_env
      include "mpif.h"
      integer myid,numprocs,ierr,nlength
      character(len=MPI_MAX_LIBRARY_VERSION_STRING+1) :: version
      character (len=MPI_MAX_PROCESSOR_NAME+1):: myname
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      call MPI_Get_processor_name(myname,nlength,ierr)
      call MPI_Get_library_version(version, nlength, ierr)
      write (*,*) "Hello from ",trim(myname)," # ",myid," of ",numprocs
      if (myid .eq. 0)then
              write(*,*)trim(version)
              write(*,*)"compiler: ",compiler_version()
      endif
      call MPI_FINALIZE(ierr)
      stop
      end

END

cat > helloc.c << END
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

/*************************************************************
This is a simple hello world program. Each processor prints
name, rank, and total run size.
*************************************************************/
int main(int argc, char **argv)
{
    int myid,numprocs,resultlen;
    char version[MPI_MAX_LIBRARY_VERSION_STRING];
    char myname[MPI_MAX_PROCESSOR_NAME] ;
    int vlan;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(myname,&resultlen);
    printf("Hello from %s %d %d\n",myname,myid,numprocs);
    if (myid == 0 ) {
               MPI_Get_library_version(version, &vlan);
               printf("%s\n",version);
    }
    MPI_Finalize();
}
END
```

```
…
echo our default modules
ml

echo This should work but does not
echo Compiling C
cc helloc.c –o helloc
echo
echo Compiling Fortran
ftn hellof.f90 –o hellof

echo
echo unloading nvhpc
module unload nvhpc
echo
echo We need cuda to get this to work
ml cuda

echo Compiling and running C
cc helloc.c –o helloc
srun –n 2 ./helloc

echo Compiling and running fortran
ftn hellof.f90 –o hellof
srun –n 2 ./hellof
```

Quick start Output

```
[tkaiser2@kl6 h100b]$cat quick.out
our default modules

Currently Loaded Modules:
  1) craype-x86-genoa          7) cce/17.0.0
  2) libfabric/1.15.2.0        8) craype/2.7.30
  3) craype-network-ofi        9) cray-dsmml/0.2.2
  4) perftools-base/23.12.0  10) cray-mpich/8.1.28
  5) nvhpc/24.1               11) cray-libsci/23.12.5
  6) craype-accel-nvidia90    12) PrgEnv-cray/8.5.0


This should work but does not
Compiling C
Error invoking pkg-config!
Package nccl was not found in the pkg-config search path.
Perhaps you should add the directory containing `nccl.pc'
to the PKG_CONFIG_PATH environment variable
Package 'nccl', required by 'virtual:world', not found
Package 'ucx-xpmem', required by 'virtual:world', not found
Package 'hcoll', required by 'virtual:world', not found
Package 'ucx-fuse', required by 'virtual:world', not found
Package 'ucx-knem', required by 'virtual:world', not found
Package 'opal', required by 'virtual:world', not found
Package 'sharp', required by 'virtual:world', not found

Compiling Fortran
Error invoking pkg-config!
Package nccl was not found in the pkg-config search path.
Perhaps you should add the directory containing `nccl.pc'
to the PKG_CONFIG_PATH environment variable
Package 'nccl', required by 'virtual:world', not found
Package 'ucx-xpmem', required by 'virtual:world', not found
Package 'hcoll', required by 'virtual:world', not found
Package 'ucx-fuse', required by 'virtual:world', not found
Package 'ucx-knem', required by 'virtual:world', not found
Package 'opal', required by 'virtual:world', not found
Package 'sharp', required by 'virtual:world', not found

unloading nvhpc

We need cuda to get this to work
Compiling and running C
Hello from x3100c0s13b0n0 1 2
Hello from x3100c0s13b0n0 0 2
MPI VERSION    : CRAY MPICH version 8.1.28.15 (ANL base 3.4a2)
MPI BUILD INFO : Wed Nov 15 20:31 2023 (git hash 1cde46f)

Compiling and running fortran
 Hello from x3100c0s13b0n0 #  1  of  2
 STOP
 Hello from x3100c0s13b0n0 #  0  of  2
 MPI VERSION    : CRAY MPICH version 8.1.28.15 (ANL base 3.4a2)
MPI BUILD INFO : Wed Nov 15 20:31 2023 (git hash 1cde46f)

 compiler: Cray Fortran : Version 17.0.0
 STOP
[tkaiser2@kl6 h100b]$
```

PrgEnv-gnu and PrgEnv-intel

```
echo Similar for PrgEnv-gnu and PrgEnv-intel


module unload nvhpc
ml cuda
ml PrgEnv-gnu
cc -march=skylake helloc.c -o gcc.exe
srun -n 2 ./gcc.exe



module unload nvhpc
ml cuda
ml PrgEnv-intel
ml cray-libsci/22.12.1.1
ml binutils
cc  helloc.c -o intel.exe
srun -n 2 ./intel.exe
```

Covering Today:

- Build and run on Kestrel's GPU nodes using several programming paradigms.
- Multiple GPUs & multiple nodes.
  - Pure Cuda programs
  - Cuda aware MPI programs
  - MPI programs without Cuda,
  - MPI programs with Cuda
  - MPI programs with Openacc
  - Pure Openacc programs.
  - Library routines
- We'll build with
  - Cray's standard programming environment
  - NVIDIA's environment
  - Gcc
  - A few examples with Intel MPI
- A repository will be given with programs, along with build and run scripts and a driver script.

- *Will not get to it today but we include slides that:*
  - *Discuss building and running the multi-gpu python Tensorflow/Horovod module.*

Just Do It

- git clone https://github.nrel.gov/tkaiser2/h100buildrun.git h100
- git clone $USER@kestrel.hpc.nrel.gov:/nopt/nrel/apps/examples/gpu/0824 h100
- tar -xzf /nopt/nrel/apps/examples/gpu/h100.tgz
- cd h100
- sbatch --account=XXXX --reservation=YYYY script
- Builds and Runs all of the examples in about 22 minutes
- Have added some things since "friendly users"
  - Multinode/GPU stream benchmark
  - More MPI gpu aware examples (used for testing new versions of MPI)
  - Select base gcc, test subset, other minor enhancements
  - Script "onnodes" to show what's happening
  - Hack to reset modules to a know state

onnodes

```
#!/usr/bin/bash
# ./onnodes [JOB_ID ...]
# For each specified or running JOB
#  For each NODE of the JOB
#    Show what user has running on each core
#    If NODE has GPUs show what is running on the GPUs
if [[ $# -eq 0 ]] ; then
  jobs=`squeue -hu $USER -o %A%t | grep R | sed "s/R//"`
else
  jobs="${@:1}"
fi
for j in $jobs ; do
  echo JOB $j
  list=`squeue -u $USER -j $j -ho %N`
  long=`scontrol show hostnames $list | sort -u`
  echo NODES $long
  for l in $long ; do
    echo $l
    #ssh $l uptime
    echo "   PID    LWP PSR COMMAND         %CPU"
    ssh $l ps -U $USER -L -o pid,lwp,psr,comm,pcpu | grep -v COMMAND | sort -k3n
    ssh $l "if  command -v nvidia-smi  ; then nvidia-smi; fi"
  done
done
```

Compiling for GPUs

- Currently the GPU environment is only available on GPU nodes
- You must compile your apps
  - On GPU login nodes
    - ssh kl5.hpc.nrel.gov
    - ssh kl5.hpc.nrel.gov
  - Via an interactive session in the gpu-h100 partition
  - Within your "run" script as we do today
- Interactive session:

```
salloc --exclusive --mem=0 --nodes=1 -t 02:00:00
--partition=gpu-h100 --gres=gpu:h100:4
--reservation=YYY --account=XXXX
```

- Batch script should have:

```
#SBATCH --partition=gpu-h100
#SBATCH --gres=gpu:h100:4
```

About gcc

- Almost all compiling/running on a linux system will at some point reference or in some way use some portion of the GNU (gcc/ gfortran/linker) system
- Kestrel has many versions of gcc
- They fall in three categories
  - Native to the Operating system
  - Built by Cray
  - Built by NREL
  - "mixed" are just duplicates of others and should not be loaded

gcc on Kestrel - 3 different builders

```
ml gcc-native/12.1
which gcc
/opt/rh/gcc-toolset-12/root/usr/bin/gcc

ml gcc/12.2.0
which gcc
/opt/cray/pe/gcc/12.2.0/bin/gcc

ml gcc-standalone/13.1.0
which gcc
/nopt/nrel/apps/gpu_stack/compilers/03-24/.../gcc-13.1.0.../bin/gcc

ml gcc-standalone/12.3.0
which gcc
/nopt/nrel/apps/cpu_stack/compilers/06-24/.../gcc-12.3.0.../bin/gcc
```

These are the compilers we might use. However, none of these compilers will build all of our examples without errors and/or warnings

More module Info

- The command module avail will show all modules
- module spider gcc will show modules pertaining to gcc
- Any module with nrel in the path is locally written
- Any module that stars with PrgEnv- give Cray MPI with someones compiler backend
  - Not all PrgEnv- modules work
  - Kept around to avoid breaking other things.

We want to start with a known Environment

- All our examples start with:

  ~~module restore~~

  myrestore
- Again, this must be done on a GPU node
- Starts fresh
- Gives us access to the latest **C**ray **P**rograming **E**nvironment (23) modules => compilers, libs...
- In most case we will want to unload (However, I use these in some examples)

  #module unload PrgEnv-cray/8.5.0
  #module unload nvhpc/24.1

  As of the October system time these are not loaded by default so they do not need to be unloaded in most cases

module reset / myrestore

Module reset is broken

Some things don't load properly after reset

Unsetting some variables before calling module reset fixes some things

If your PATH contains nonstandard directories they get moved to the middle of PATH

We define a "hack" function to fix this


```
myrestore () {
    unset __LMOD_REF_COUNT_MANPATH
    unset __LMOD_REF_COUNT_MODULEPATH
    unset __LMOD_REF_COUNT_PATH
    module reset
    export PATH=`myfront`
}
```

myrestore and myfront are defined in whack.sh

whack.sh

```
#!/bin/bash
## https://codingfleet.com/code-converter/python/bash
## used to convert original python to bash

# Function to move specified paths to the front of a colon-separated list
upfront() {
  local p="$1"
  local move="$2"
  local myset=""
  local therest=""

  # Split the move string into individual paths
  IFS=, read -ra move <<< "$move"

  # Split the path string into individual paths
  IFS=: read -ra p <<< "$p"

  # Iterate through each move path
  for m in "${move[@]}"; do
    # Iterate through each path in the original path list
    for a in "${p[@]}"; do
      # Check if the move path is found in the current path
      if [[ "$a" =~ "$m" ]]; then
        # Append the path to the myset string
        myset="${myset}$a:"
      else
        # Append the path to the therest string
        therest="$therest$a:"
      fi
    done
  done

  # Return the combined string with moved paths at the front
  echo "${myset}${therest%:}"
}
```

```
# Function to modify the PATH environment variable
myfront() {
  local me="$USER"
  local p="$PATH"
  local np=$(upfront "$p" "$me")
  echo "$np"
}

# Execute the myfront function if the script is run directly
if [[ "$0" == "$BASH_SOURCE" ]]; then
  myfront
fi
```

```
myrestore () {
    unset __LMOD_REF_COUNT_MANPATH
    unset __LMOD_REF_COUNT_MODULEPATH
    unset __LMOD_REF_COUNT_PATH
    module reset
    export PATH=`myfront`
}
```

This slide has been superseded

module reset / myrestore

- The command module reset is broken
- Some things don’t load properly after reset
- The function myrestore sources the file /nopt/nrel/apps/env.sh which sets your module environment back to the login state
- myrestore is defined in whack.sh

As of October 23, 2024 myrestore is also defined in /nopt/nrel/apps/env.sh

- /nopt/nrel/apps/env.sh sets modules back to the original state
- Myrestore also modifies $PATH and $LD_LIBRARY_PATH putting paths with your home directory at the beginning.

After myrestore

```
[tkaiser2@x3102c0s13b0n0 nvidiaopenmpi]$myrestore
[tkaiser2@x3102c0s13b0n0 nvidiaopenmpi]$ml

Currently Loaded Modules:
  1) craype-x86-genoa     4) perftools-base/23.12.0   7) cce/17.0.0         10) cray-mpich/8.1.28
  2) libfabric/1.15.2.0   5) nvhpc/24.1               8) craype/2.7.30      11) cray-libsci/23.12.5
  3) craype-network-ofi   6) craype-accel-nvidia90    9) cray-dsmml/0.2.2   12) PrgEnv-cray/8.5.0
```

Our Examples

- Our driver script is just "script"
- Each example directory contains a file "doit"
- Our driver looks for each directory with an example; Goes there and sources doit
- We can select the default compiler to use by setting the environmental variable MYGCC; This can be done outside of the script before sbatch
- If we know that an example will not run with the chosen version of gcc "doit" will substitute on the fly
- You can run a subset of the tests by setting the variable doits

More notes

- While all of these examples work...
  - In most cases there maybe other combinations of modules that can be used
  - These might not be the best
  - The machine is evolving
    - Somethings might not work later
    - There are already know changes coming to Kestrel that will deprecate some of the module choices

Common issues

- Can’t find library at run time
  - Need to set LD_LIBRARY_PATH to point to directory containing the library. Try to load modules at run time.
- Module xxx is not compatible with your cray-libsci
  - Load an different version: cray-libsci/22.10.1.2 or cray-libsci/22.12.1.1 or cray-libsci/23.05.1.4
- Can’t find some function in the c++ library
  - Load a newer version of gcc
- At link time libgcc_s.so.1: file not recognized: File format not recognized
  - Linker is missing after some combinations of loads. module load binutils
- Examples shown here don’t work
  - Make sure you are running and or launching from a GPU node
- cc1: error: bad value ‘znver4’ for ‘-march=’ switch
  - -march=skylake
    - Or
  - module load craype-x86-milan
- Package 'nccl', required by 'virtual:world', not found…
  - module unload nvhpc

Our Examples

```
[tkaiser2@kl1 jun20]$find . -name doit | sort -t/ -k2,2
./cuda/cray/doit
./cuda/gccalso/doit
./cuda/nvidia/doit
./cudalib/factor/doit
./cudalib/fft/doit
./mpi/cudaaware/doit
./mpi/normal/cray/doit
./mpi/normal/intel+abi/doit
./mpi/normal/nvidia/nrelopenmpi/doit
./mpi/normal/nvidia/nvidiaopenmpi/doit
./mpi/openacc/cray/doit
./mpi/openacc/nvidia/nrelopenmpi/doit
./mpi/openacc/nvidia/nvidiaopenmpi/doit
./mpi/withcuda/cray/doit
./mpi/withcuda/nvidia/nrelopenmpi/doit
./mpi/withcuda/nvidia/nvidiaopenmpi/doit
./openacc/cray/doit
./openacc/nvidia/doit
[tkaiser2@kl1 jun20]$
```

Name of directory says what we are testing

Driver Script

```
#!/bin/bash
#SBATCH --time=0:30:00
#SBATCH --partition=gpu-h100
#SBATCH --nodes=2
#SBATCH --gres=gpu:h100:4
#SBATCH --exclusive
#SBATCH --output=output-%j.out
#SBATCH --error=infor-%j.out

if echo $SLURM_SUBMIT_HOST | egrep "kl5|kl6" >> /dev/null  ; then : ; else echo Run script from a GPU node;
exit ; fi

# a simple timer
dt ()
{
    now=`date +"%s.%N"`;
    if (( $# > 0 )); then
        rtn=$(printf "%0.3f" `echo $now - $1 | bc`);
    else
        rtn=$(printf "%0.3f" `echo $now`);
    fi;
    echo $rtn
}

printenv > env-$SLURM_JOB_ID.out
cat $0 > script-$SLURM_JOB_ID.out
```

Driver Script

```
#runs script to put our restore function in our environment
. whack.sh
myrestore

#some possible values for gcc module
#export MYGCC=gcc-native/12.1
#export MYGCC=gcc-stdalone/10.1.0
#export MYGCC=gcc-stdalone/12.3.0
#export MYGCC=gcc-stdalone/13.1.0


if [ -z ${MYGCC+x} ]; then export MYGCC=gcc-native/12.1 ; else echo MYGCC already set ; fi
echo MYGCC=$MYGCC

if [ -z ${doits+x} ]; then
        doits=`find . -name doit | sort -t/ -k2,2`
else
        echo doits already set
fi

for x in $doits ; do
        echo running example in `dirname $x`
done
```

Driver Script

```
startdir=`pwd`
t1=`dt`
for x in $doits ; do
 dir=`dirname $x`
 echo ++++++++ $dir >&2
 echo ++++++++
 echo $dir
 cd $dir
 tbegin=`dt`
 . doit | tee  $SLURM_JOB_ID
 echo Runtime `dt $tbegin` $dir `dt $t1` total
 cd $startdir
done
echo FINISHED `dt $t1`

# post  (this is optional)
mkdir -p /scratch/$USER/gputest/$SLURM_JOB_ID
cp *out  /scratch/$USER/gputest/$SLURM_JOB_ID
#. cleanup
```

A "simple" cuda code with Cray Environment

```
[tkaiser2@kl1 jun20]$cd ./cuda/cray
[tkaiser2@kl1 cray]$ls
doit  stream.cu  stream.sm_90
[tkaiser2@kl1 cray]$cat doit
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
#module unload nvhpc/24.1

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
ml PrgEnv-nvhpc/8.4.0
ml cray-libsci/23.05.1.4
: << ++++
 Compile our program
 CC as well as cc, and ftn are wrapper compilers. Because
 we have PrgEnv-nvidia loaded they map to Nvidia's compilers
 but use would use Cray MPI if this was an MPI program.
 Note we can also use nvcc since this is not an MPI program.
++++

rm -rf ./stream.sm_90
CC -gpu=cc90  -cuda -target-accel=nvidia90  stream.cu  -o stream.sm_90
# nvcc -std=c++11 -ccbin=g++ stream.cu -arch=sm_90 -o stream.sm_90

: Run on all of our nodes
nlist=`scontrol show hostnames | sort -u`
for l in $nlist ; do
  echo $l
  for GPU in 0 1 2 3 ; do
: stream.cu will read the GPU on which to run from the command line
      srun -n 1 --nodes=1 -w $l ./stream.sm_90 -g $GPU
  done
  echo
done
```

Steam.cu runs a standard benchmark showing the computational speed of the gpu for simple math operations.

We use PrgEnv-nvhpc which is a combination or Cray (mostly MPI) and NVIDIA's back end compilers. CC is a wrapper, that will build both serial and MPI codes along with Cuda if it is enabled.

Various compilers need specific versions of cray-libsci.

We run on each of the GPUs one at a time.

A "simple" cuda code with NVIDIA's Environment

```
[tkaiser2@kl1 cuda]$cd ./cuda/gccalso/
[tkaiser2@kl1 gccalso]$ls
cuda.cu  doit  doswift  extras.h  normal.c
[tkaiser2@kl1 gccalso]$cat doit
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
#module unload nvhpc/24.1

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
ml nvhpc-nompi/24.1

ml 2>&1 | grep gcc-stdalone/13.1.0 ; if [ $? -eq 0 ]  ; then echo REPLACING gcc-stdalone/13.1.0 ; ml gcc-stdalone/12.3.0  ; fi

: << ++++
 Compile our program
 The module nvhpc-nompi gives us access to Nvidia's compilers
 nvc, nvc++, nvcc, nvfortran as well as the Portland Group
 compilers which are actually links to these.  We do not
 have direct access to MPI with this set of modules loaded.
 Here we compile routines that do not containe cuda with g++.
++++


g++ -c normal.c
nvcc -std=c++11 -arch=sm_90 cuda.cu normal.o -o stream.sm_90

: Run on all of our nodes
nlist=`scontrol show hostnames | sort -u`
for l in $nlist ; do
  echo $l
  for GPU in 0 1 2 3 ; do
: stream.cu will read the GPU on which to run from the command line
      srun -n 1 --nodes=1 -w $l ./stream.sm_90 -g $GPU
  done
  echo
done
[tkaiser2@kl1 gccalso]$
```

Steam.cu runs a standard benchmark showing the computational speed of the gpu for simple math operations. We have broken steam into cuda and c files.

We use nvhpc-nompi which is a NREL written environment that builds cuda programs without MPI. Here we build the non cuda portion of the program with gcc. This is optional and not required.

We "REPLACE" gcc/13.1.0 because it is too new to work with this version of NVIDIA's compiler. This issue should go away in the near future.

We run on each of the GPUs one at a time.

A "simple" cuda code with NVIDIA's Environment

```
[tkaiser2@kl1 jun20]$cd ./cuda/nvidia
[tkaiser2@kl1 nvidia]$ls
doit  doswift  stream.cu
[tkaiser2@kl1 nvidia]$cat doit
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
#module unload nvhpc/24.1

ml nvhpc-nompi/24.1
: << ++++
 Compile our program
 The module nvhpc-nompi gives us access to Nvidia's compilers
 nvc, nvc++, nvcc, nvfortran as well as the Portland Group
 compilers which are actually links to these.  We do not
 have direct access to MPI with this set of modules loaded.
++++


nvcc -std=c++11 -arch=sm_90 stream.cu -o stream.sm_90

: Run on all of our nodes
nlist=`scontrol show hostnames | sort -u`
for l in $nlist ; do
  echo $l
  for GPU in 0 1 2 3 ; do
: stream.cu will read the GPU on which to run from the command line
      srun -n 1 --nodes=1 -w $l ./stream.sm_90 -g $GPU
  done
  echo
done
[tkaiser2@kl1 nvidia]$
```

Steam.cu runs a standard benchmark showing the computational speed of the gpu for simple math operations.

We use nvhpc-nompi which is a NREL written environment that builds cuda programs without MPI.

We run on each of the GPUs one at a time.

Simple MPI, no cuda with 3 versions of Cray-PE

```
[tkaiser2@kl1 jun20]$cd ./mpi/normal/cray
[tkaiser2@kl1 cray]$ls
doit  helloc.c  hellof.f90
[tkaiser2@kl1 cray]$cat doit
: Start from a known module state, the default
module purge
myrestore


: Load modules
#module unload nvhpc/24.1
ml PrgEnv-cray/8.4.0

ml cuda
: << ++++
 Compile our program.

 Here we use cc and ftn.  These are wrappers
 that point to Cray C (clang) Cray Fortran
 and Cray MPI. cc and ftn are part of PrgEnv-cray
 with is part of the default setup.
++++

cc helloc.c -o helloc
ftn hellof.f90 -o hellof

: We run with two tasks per nodes an two tasks on one node.
for arg in "--tasks-per-node=2" "-n 2 --nodes=1" ; do
   echo runnning Fortran version
   srun $arg hellof
   echo
   echo runnning C version
   srun $arg helloc
   echo
done
...
...
```

VERSION 1

Hello world in Fortran and C.

The default programming environment is PrgEnv-cray. cc and ftn will compile serial and MPI programs

Note we “ml cuda” here. This make almost no sense!

Simple MPI, no cuda with 3 versions of Cray-PE

```
...
...

: With PrgEnv-intel we get the Intel backend compilers
ml PrgEnv-intel
ml cray-libsci/23.05.1.4
#ml gcc-stdalone/13.1.0
ml binutils

cc helloc.c -o helloc.i
ftn hellof.f90 -o hellof.i

: We run with two tasks per nodes an two tasks on one node.
for arg in "--tasks-per-node=2" "-n 2 --nodes=1" ; do
   echo runnning Fortran version with Intel backend
   srun $arg hellof.i
   echo
   echo runnning C version with Intel backend
   srun $arg helloc.i
   echo
done


...
...
```

VERSION 2

Hello world in Fortran and C.

With PrgEnv-intel we get Intel backend compilers. cc and ftn will compile serial and MPI programs

Simple MPI, no cuda with 3 versions of Cray-PE

```
...
...

: With PrgEnv-gnu we get the gnu backend compilers
: As of 04/04/24 the -march=znver3 flag is required
: because the default version of gcc does not support the
: current CPU on the GPU nodes.  Or you could
: ml craype-x86-milan
ml PrgEnv-gnu
ml cray-libsci/23.05.1.4
cc  -march=znver3 helloc.c -o helloc.g
ftn -march=znver3 hellof.f90 -o hellof.g

: We run with two tasks per nodes an two tasks on one node.
for arg in "--tasks-per-node=2" "-n 2 --nodes=1" ; do
   echo runnning Fortran version with gnu backend
   srun $arg hellof.g
   echo
   echo runnning C version with gnu backend
   srun $arg helloc.g
   echo
done
```

VERSION 3

Hello world in Fortran and C.

With PrgEnv-gnu we get gcc and fortran as backend compilers. cc and ftn will compile serial and MPI programs

Simple MPI, Compile with Intel MPI run with Cray MPI

```
[tkaiser2@kl1 jun20]$cd ./mpi/normal/intel+abi
[tkaiser2@kl1 intel+abi]$ls
docpu  doit  doswift  helloc.c  hellof.f90  oncpu
[tkaiser2@kl1 intel+abi]$cat doit
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
#module unload nvhpc/24.1

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
ml intel-oneapi-mpi
ml intel-oneapi-compilers

: << ++++
 Compile our program.

 There are many ways to compile using Intel MPI.
 Here we use the "Intel Suggested" way using mpiicx
 and mpifc.  This gives us new Intel backend compilers
 with Intel MPI. mpif90 and mpicc would give us gcc
 and gfortan instead
++++

mpiicx helloc.c -o helloc
mpifc hellof.f90 -o hellof

: We run with two tasks per nodes an two tasks on one node.
for arg in "--tasks-per-node=2" "-n 2 --nodes=1" ; do
   echo runnning Fortran version
   srun $arg hellof
   echo
   echo runnning C version
   srun $arg helloc
   echo
done
```

VERSION 3

Load Intel compilers, build and run. The output from hello world says we are running using Intel mpi

```
runnning Fortran version
Running:hellof
 Intel(R) MPI Library 2021.11 for Linux* OS
 Hello from x3102c0s25b0n0 #        0  of        4
 Hello from x3102c0s25b0n0 #        1  of        4
 Hello from x3112c0s17b0n0 #        2  of        4
 Hello from x3112c0s17b0n0 #        3  of        4

runnning C version
Running: helloc
Intel(R) MPI Library 2021.11 for Linux* OS
Hello from x3102c0s25b0n0 0 4
Hello from x3102c0s25b0n0 1 4
Hello from x3112c0s17b0n0 2 4
Hello from x3112c0s17b0n0 3 4
```

Simple MPI, Compile with Intel MPI run with Cray MPI

```
: Finally we module load cray-mpich-abi.  With this module
: loaded Intel MPI is replaced with Cray MPI without needing
: to recompile. After the load we rerun and see Cray MPI
: in the output

ml craype
ml cray-mpich-abi

for arg in "--tasks-per-node=2" "-n 2 --nodes=1" ; do
   echo runnning Fortran version
   srun $arg hellof
   echo
   echo runnning C version
   srun $arg helloc
   echo
done
```

With load cray-mpich-abi and rerun without rebuilding and it shows we are not running with Cray MPI

```
Running:hellof
 MPI VERSION    : CRAY MPICH version 8.1.28.15 (ANL base 3.4a2)
MPI BUILD INFO : Wed Nov 15 21:00 2023 (git hash 1cde46f)

 Hello from x3102c0s25b0n0 #        0  of        4
 Hello from x3102c0s25b0n0 #        1  of        4
 Hello from x3112c0s17b0n0 #        2  of        4
 Hello from x3112c0s17b0n0 #        3  of        4

Running: helloc
MPI VERSION    : CRAY MPICH version 8.1.28.15 (ANL base 3.4a2)
MPI BUILD INFO : Wed Nov 15 21:00 2023 (git hash 1cde46f)

Hello from x3102c0s25b0n0 0 4
Hello from x3102c0s25b0n0 1 4
Hello from x3112c0s17b0n0 2 4
Hello from x3112c0s17b0n0 3 4
```

Simple MPI, OpenMPI and NVIDIA's backend compilers

```
[tkaiser2@kl1 jun20]$cd mpi/normal/nvidia/nrelopenmpi
[tkaiser2@kl1 nrelopenmpi]$ls
doit  doswift  helloc.c  hellof.f90
[tkaiser2@kl1 nrelopenmpi]$cat doit
: Start from a known module state, the default
module purge
myrestore

:  Enable a newer environment
: Load modules
#module unload PrgEnv-cray/8.5.0
#module unload nvhpc/24.1

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
ml openmpi/4.1.6-nvhpc
ml nvhpc-nompi/24.1

: << ++++
 Compile our program
 Here we use mpicc and mpif90.  There is support for Cuda
 but we are not using it in this case.
++++

mpicc helloc.c -o helloc
mpif90 hellof.f90 -o hellof

: We run with two tasks per nodes an two tasks on one node.
for arg in "--tasks-per-node=2" "-n 2 --nodes=1" ; do
   echo runnning Fortran version
   srun $arg hellof
   echo
   echo runnning C version
   srun $arg helloc
   echo
done
```

This is a NREL build version of OpenMPI that uses NVIDIA's backend compilers. Not that useful here but will be in other context. We don't have cuda but if we did we could mix it in with ease.

Simple MPI, NVIDIA's MPI compilers

```
[tkaiser2@kl1 jun20]$cd ./mpi/normal/nvidia/nvidiaopenmpi
[tkaiser2@kl1 nvidiaopenmpi]$ls
doit  doswift  helloc.c  hellof.f90
[tkaiser2@kl1 nvidiaopenmpi]$cat doit
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
##module unload nvhpc/24.1

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
#ml nvhpc-hpcx-cuda12/24.1

: << ++++
 Compile our program
 Here we use mpicc and mpif90.  There is support for Cuda
 but we are not using it in this case.
++++

mpicc helloc.c -o helloc
mpif90 hellof.f90 -o hellof

: This version of MPI does not support srun so we use mpirun
: We run with two tasks per nodes an two tasks on one node.
for arg in "-N 2" "-n 2" ; do
   echo runnning Fortran version
   mpirun $arg hellof
   echo
   echo runnning C version
   mpirun $arg helloc
   echo
done
```

This is NVIDIA's compilers. Not that useful here but will be in other context. We don't have cuda but if we did we could mix it in with ease.

MPI codes compiled with NVIDIA's MPI need to be launched with mpirun.

Might actually want to use nvhpc/23.5 instead

```
Currently Loaded Modules:
  1) craype-x86-genoa          2) libfabric/1.15.2.0   3) craype-network-ofi
  4) perftools-base/23.12.0    5) nvhpc/24.1           6) craype-accel-nvidia90
  7) gcc/12.1.0
```

Back to cuda

- OpenACC
- MPI with OpenACC
- MPI with cuda
  - Send data to/from GPU via "normal" push/pull and MPI calls
  - MPI coda that calls a cuda kernel
- Cuda aware MPI - MPI calls send/recv data directly to/from GPUs
- Finish up with some libraries.

Openacc using PrgEnv-nvhpc

```
[tkaiser2@kl1 jun20]$cd ./openacc/cray
[tkaiser2@kl1 cray]$cat doit
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
#module unload nvhpc/24.1

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
ml PrgEnv-nvhpc/8.5.0


: << ++++
 Compile our program
 The module PrgEnv-nvhpc/8.5.0 gives us access to Nvidia's
 compilers nvc, nvc++, nvcc, nvfortran as well as the Portland
 Group compilers which are actually links to these.  Since we
 are not using MPI we could have also used nvhpc-nompi/24.1 or
 even nvhpc-native/24.1.
++++

nvc -fast -Minline -Minfo -acc -DFP64 nbodyacc2.c -o nbody


: Run on all of our nodes
nlist=`scontrol show hostnames | sort -u`
for l in $nlist ; do
  echo $l
  for GPU in 0 1 2 3 ; do
: This is one way to set the GPU on which a openacc program runs.
      export CUDA_VISIBLE_DEVICES=$GPU
      echo running on gpu $CUDA_VISIBLE_DEVICES
: Since we are not running MPI we actaully do not need srun here.
      srun -n 1 --nodes=1 -w $l ./nbody
  done
  echo
done

unset CUDA_VISIBLE_DEVICES
```

Our example from NVIDIA. We build with Prg_Env-nvhpc with combines Cray MPI with NVIDIA's compilers. However, this example does not use MPI.

We run on each of the GPUs one at a time by setting CUDA_VISABLE_DEVICES.

We launch with srun although it is not needed in this case.

Might want to use PrgEnv-nvhpc/8.4.0 instead

Openacc using nvhpc-native

```
[tkaiser2@kl1 jun20]$cd ./openacc/nvidia
[tkaiser2@x3100c0s21b0n0 nvidia]$cat doit
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
#module unload nvhpc/24.1

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
ml nvhpc/24.1


: << ++++
 Compile our program
 The module nvhpc-native gives us access to Nvidia's compilers
 nvc, nvc++, nvcc, nvfortran as well as the Portland Group
 compilers which are actually links to these.  Since we are
 not using MPI we could have also used nvhpc-nompi/24.1 or
 even PrgEnv-nvhpc/8.5.0.
++++


nvc -fast -Minline -Minfo -acc -DFP64 nbodyacc2.c -o nbody


: Run on all of our nodes
nlist=`scontrol show hostnames | sort -u`
for l in $nlist ; do
  echo $l
  for GPU in 0 1 2 3 ; do
: This is one way to set the GPU on which a openacc program runs.
      export CUDA_VISIBLE_DEVICES=$GPU
      echo running on gpu $CUDA_VISIBLE_DEVICES
: Since we are not running MPI we actaully do not need srun here.
      srun -n 1 --nodes=1 -w $l ./nbody
  done
  echo
done

unset CUDA_VISIBLE_DEVICES
```

Our example from NVIDIA. We build with NVIDIA's compilers. However, this example does not use MPI.

We run on each of the GPUs one at a time by setting CUDA_VISABLE_DEVICES.

We launch with srun although it is not needed in this case.

MPI and Openacc

MPI and Openacc using PrgEnv-nvhpc

```
[tkaiser2@x3100c0s13b0n0 jun20]$cd mpi/openacc/cray
[tkaiser2@x3100c0s13b0n0 cray]$ls
acc_c3.c  doit
[tkaiser2@x3100c0s13b0n0 cray]$cat doit
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
#module unload nvhpc/24.1

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
ml PrgEnv-nvhpc
ml cray-libsci/23.05.1.4

: << ++++
 Compile our program.

 Here we use cc and ftn.  These are wrappers
 that point to Cray C (clang) Cray Fortran
 and Cray MPI. cc and ftn are part of PrgEnv-cray
 which is part of the default setup.
++++

cc -acc -Minfo=accel -fast acc_c3.c  -o jacobi

: We run with 4 tasks per nodes.
srun --tasks-per-node=4 ./jacobi 46000 46000 5 nvidia
```

This example just combines some Openacc Code with MPI calls.

We are using PrgEnv-nvhpc which combines Cray MPI with NVIDIA's compilers.

The example just uses MPI to ask each node to do the same calculation with Openacc and then uses MPI to gather and report stats.

MPI and Openacc using PrgEnv-nvhpc

```
[tkaiser2@x3100c0s13b0n0 jun20]$cd mpi/openacc/nvidia/nrelopenmpi
[tkaiser2@x3100c0s13b0n0 nrelopenmpi]$ls
acc_c3.c  doit  doswift
[tkaiser2@x3100c0s13b0n0 nrelopenmpi]$cat doit
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
#module unload nvhpc/24.1

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
ml openmpi/4.1.6-nvhpc
ml nvhpc-nompi/24.1

: << ++++
 Compile our program
 Here we use mpicc and mpif90.  There is support for Cuda
 but we are not directly using it in this case, just openacc.
++++

mpicc -acc -Minfo=accel -fast acc_c3.c -o jacobi

: We run with 4 tasks per nodes.
srun --tasks-per-node=4 ./jacobi 46000 46000 5 nvidia
```

This example just combines some Openacc Code with MPI calls.

We are using openmpi which is build with NVIDIA's compilers and also loading nvhpc-nompi. This "nompi" module is loaded because then nvhpc (NVIDIA) toolkit normally has mpi also. (See the next slide.)

The example just uses MPI to ask each node to do the same calculation with Openacc and then uses MPI to gather and report stats.

MPI and Openacc using PrgEnv-nvhpc

```
[tkaiser2@x3100c0s13b0n0 jun20]$cd mpi/openacc/nvidia/nvidiaopenmpi
[tkaiser2@x3100c0s13b0n0 nvidiaopenmpi]$ls
acc_c3.c  doit  doswift
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
##module unload nvhpc/24.1

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi

: << ++++
 Compile our program
 Here we use mpicc and mpif90.  There is support for Cuda
 but we are not using it in this case but we are using
 openacc.
++++

mpicc -acc -Minfo=accel -fast acc_c3.c -o jacobi

: We run with 4 tasks per nodes.
: This version of MPI does not support srun so we use mpirun
mpirun -N 4 ./jacobi 46000 46000 5 nvidia
```

This example just combines some Openacc Code with MPI calls.

The module nvhpc/24.1 has NVIDIA's complete toolkit. Here we use their pcx version of MPI which might give a little better performance than OpenMPI. (Stay tuned for Updates about OpenMPI.)

The example just uses MPI to ask each node to do the same calculation with Openacc and then uses MPI to gather and report stats.

```
Currently Loaded Modules:
  1) craype-x86-genoa          2) libfabric/1.15.2.0   3) craype-network-ofi
  4) perftools-base/23.12.0    5) nvhpc/24.1           6) craype-accel-nvidia90   7) gcc/12.1.0
```

MPI and Cuda

MPI and Cuda using PrgEnv-nvhpc

```
[tkaiser2@kl1 jun20]$cd mpi/withcuda/cray
[tkaiser2@x3100c0s13b0n0 cray]$cat doit
: Start from a known module state, the default
module restore

:  Enable a newer environment
source /nopt/nrel/apps/gpu_stack/env_cpe23.sh
: Load modules
ml craype-x86-genoa
ml >&2
if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
#######
ml 2>&1 | grep gcc-native/12.1 ; if [ $? -eq 0 ]  ; then echo REPLACING gcc-native/12.1 ; ml gcc/13.1.0 ; fi
#######
ml >&2

ml PrgEnv-nvhpc
ml cray-libsci/23.05.1.4

: << ++++
 Compile our program.
 Here we use CC. If we were compiling Fortran
 then ftn instead of CC.  These are wrappers
 that point to Cray MPI and with PrgEnv-nvhpc
 we get Nvidia's back end compilers.
++++

CC -gpu=cc90   ping_pong_cuda_staged.cu -o staged

: We run with 2 tasks total. One 1 and two nodes
echo running staged on node
srun  --nodes=1 --tasks-per-node=2 ./staged

echo running staged off node
srun  --nodes=2 --tasks-per-node=1 ./staged

echo running multi-gpu stream
CC -gpu=cc90  -DNTIMES=1000  mstream.cu -o mstream
export VSIZE=3300000000
export VSIZE=330000000
srun --tasks-per-node=4  ./mstream -n $VSIZE
```

This is a MPI benchmark. We have data on a GPU. We copy it to the CPU; MPI it to another task; Move it to the GPU

Again we are using PrgEnv-nvhpc which combines Cray MPI and NVIDIA's compilers.

Can we MPI directly from on GPU to another. Yep! Will see that in a bit and compare results.

The second program runs the Stream benchmark using MPI to run it on each GPU in parallel.

MPI and Cuda using NVIDIA's Environment

```
[tkaiser2@kl1 jun20]$cd mpi/withcuda/nvidia/nvidiaopenmpi
[tkaiser2@kl1 nvidiaopenmpi]$ls
doit  doswift  mstream.cu  ping_pong_cuda_staged.cu
[tkaiser2@kl1 nvidiaopenmpi]$cat doit
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
#module unload nvhpc/24.1

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
ml 2>&1 | grep gcc-stdalone/13.1.0 ; if [ $? -eq 0 ]  ; then echo REPLACING gcc-stdalone/13.1.0 ; ml gcc-stdalone/12.3.0 ; fi

#ml nvhpc-hpcx-cuda12/24.1
ml nvhpc/24.1

: << ++++
 Compile our program
 Here we use mpiCC which uses Nvidia's version of MPI and
 their backend compiler. The "hpcx" has a few more optimizations.
++++

mpiCC ping_pong_cuda_staged.cu -o staged

: We run with 2 tasks total.
: This version of MPI does not support srun so we use mpirun

echo Run on a single node
mpirun -n 2 -N 2 ./staged

echo Run on two nodes
mpirun -n 2 -N 1 ./staged


echo running multi-gpu stream
mpiCC -gpu=cc80  -DNTIMES=1000  mstream.cu -o mstream
export VSIZE=3300000000
export VSIZE=330000000
mpirun -n 8 -N 4  ./mstream -n $VSIZE
```

This is a MPI benchmark. We have data on a GPU. We copy it to the CPU; MPI it to another task; Move it to the GPU

We build with a 'Pure' NVIDIA toolset

The second program runs the Stream benchmark using MPI to run it on each GPU in parallel.

MPI and Cuda using OpenMPI and NVIDIA

```
[tkaiser2@x3100c0s13b0n0 jun20]$cd mpi/withcuda/nvidia/nrelopenmpi
[tkaiser2@x3100c0s13b0n0 nrelopenmpi]$ls
doit  doswift  mstream.cu  ping_pong_cuda_staged.cu
[tkaiser2@x3100c0s13b0n0 nrelopenmpi]$cat doit
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
#module unload nvhpc/24.1

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
ml openmpi/4.1.6-nvhpc
ml nvhpc-nompi/24.1

: << ++++
 Compile our program
 Here we use mpiCC which uses, in this case a NREL built  version
 of MPI and Nvidia's backend compiler.
++++

mpiCC ping_pong_cuda_staged.cu -o staged

: We run with 2 tasks total.
: This version of MPI does not support srun so we use mpirun

echo Run on a single node
srun --tasks-per-node=2 --nodes=1 ./staged

echo Run on two nodes
srun --tasks-per-node=1 --nodes=2 ./staged

echo running multi-gpu stream
mpiCC -gpu=cc90  -DNTIMES=1000  mstream.cu -o mstream
export VSIZE=3300000000
export VSIZE=330000000
srun --tasks-per-node=4  ./mstream -n $VSIZE
```

This is a MPI benchmark. We have data on a GPU. We copy it to the CPU; MPI it to another task; Move it to the GPU

Here we use OpenMPI built with NVIDIA's compilers.

The second program runs the Stream benchmark using MPI to run it on each GPU in parallel.

Cuda Aware MPI (Direct GPU/GPU MPI)

```
[tkaiser2@kl1 jun20]$cd mpi/cudaaware/
[tkaiser2@kl1 cudaaware]$ls
check.cu  doit  doswift  ping_pong_cuda_aware.cu  src
[tkaiser2@kl1 cudaaware]$cat doit
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload nvhpc/24.1
#module unload PrgEnv-cray/8.5.0

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
ml PrgEnv-nvhpc
ml cray-libsci/23.05.1.4
ml

: << ++++
 Compile our program.

 Here we use cc and CC.  These are wrappers
 that point to Cray MPI but use Nvidia backend
 comilers.
++++

CC -gpu=cc90  -cuda -target-accel=nvidia90  -c ping_pong_cuda_aware.cu
cc -gpu=cc90  -cuda -target-accel=nvidia90 -lcudart -lcuda ping_pong_cuda_aware.o -o pp_cuda_aware

export MPICH_GPU_SUPPORT_ENABLED=1
export MPICH_OFI_NIC_POLICY=GPU
srun -n 2 --nodes=1 ./pp_cuda_aware
srun --tasks-per-node=1 --nodes=2 ./pp_cuda_aware
unset MPICH_GPU_SUPPORT_ENABLED
unset MPICH_OFI_NIC_POLICY
```

This is a MPI benchmark. We have data on a GPU. We copy it to the CPU; MPI it to another task; Move it to the GPU

Here we use PrgENV-nvhpc (OpenMPI - work in progress)

Note: we need to set MPICH_GPU_SUPPORT_ENABLED

Check.cu is basically the same program with some extra checks and options.

Cuda MPI Staged vs Cuda Aware (Direct)

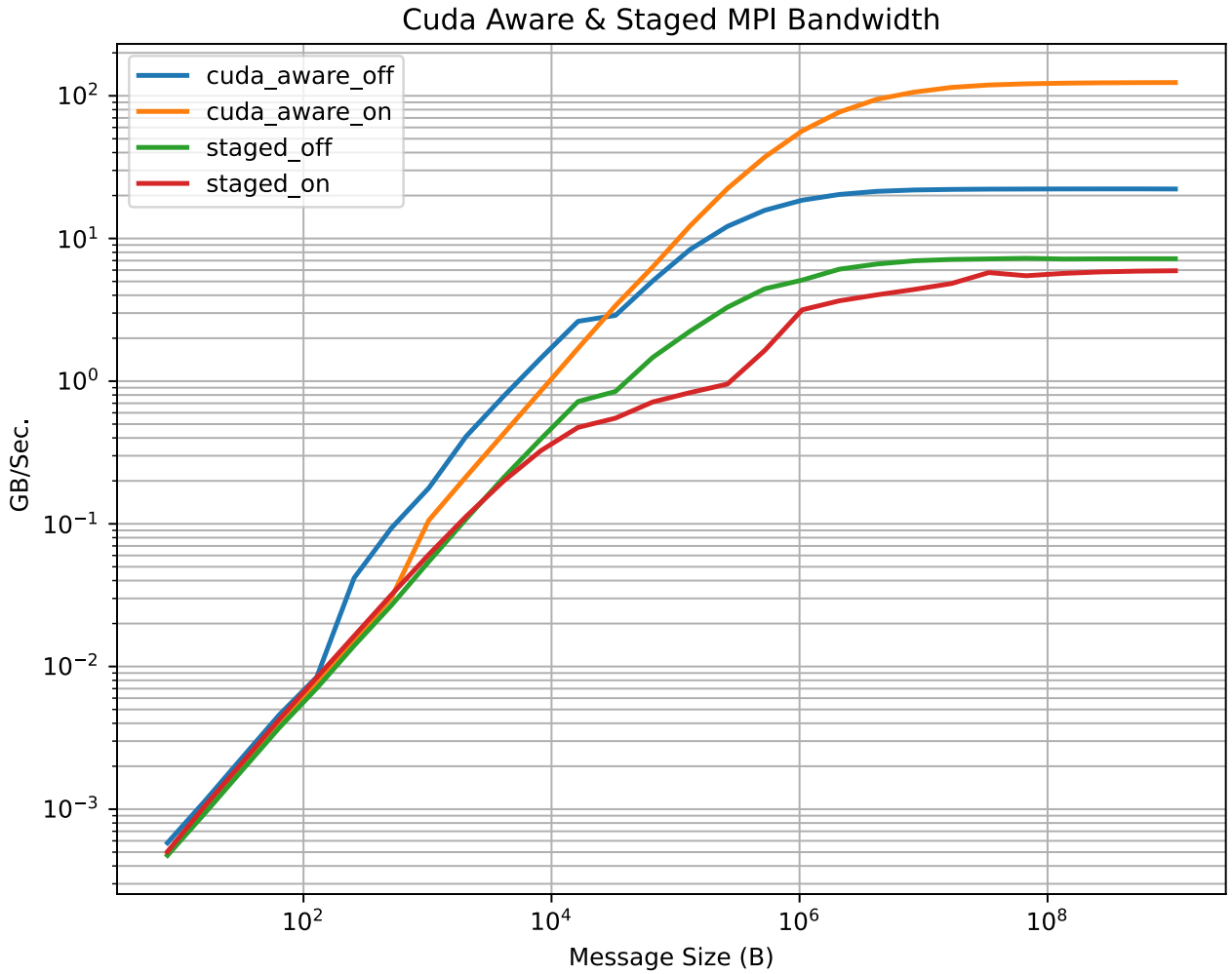
Staged

```
for(int i=1; i<=loop_count; i++){
	if(rank == 0){
		cudaMemcpy(A, d_A, N*sizeof(double), cudaMemcpyDeviceToHost) ;
		MPI_Send(A, N, MPI_DOUBLE, 1, tag1, MPI_COMM_WORLD);
		MPI_Recv(A, N, MPI_DOUBLE, 1, tag2, MPI_COMM_WORLD, &stat);
		cudaMemcpy(d_A, A, N*sizeof(double), cudaMemcpyHostToDevice) ;
	}
	else if(rank == 1){
		MPI_Recv(A, N, MPI_DOUBLE, 0, tag1, MPI_COMM_WORLD, &stat);
		cudaMemcpy(d_A, A, N*sizeof(double), cudaMemcpyHostToDevice) ;
		cudaMemcpy(A, d_A, N*sizeof(double), cudaMemcpyDeviceToHost) ;
		MPI_Send(A, N, MPI_DOUBLE, 0, tag2, MPI_COMM_WORLD);
	}
}
```

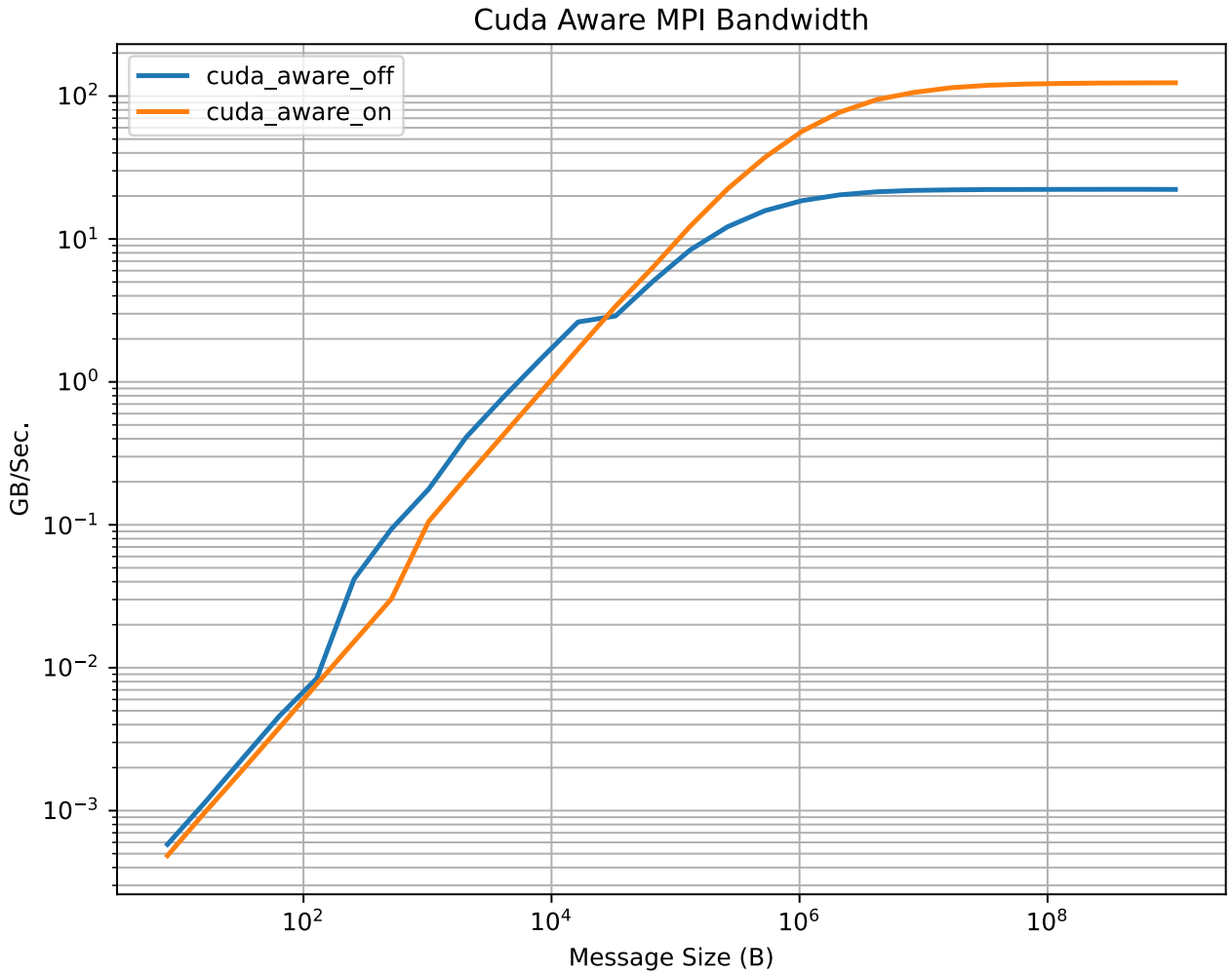
Cuda Aware

```
for(int i=1; i<=loop_count; i++){
	if(rank == 0){
		MPI_Send(d_A, N, MPI_DOUBLE, 1, tag1, MPI_COMM_WORLD);
		MPI_Recv(d_A, N, MPI_DOUBLE, 1, tag2, MPI_COMM_WORLD, &stat);
	}
	else if(rank == 1){
		MPI_Recv(d_A, N, MPI_DOUBLE, 0, tag1, MPI_COMM_WORLD, &stat);
		MPI_Send(d_A, N, MPI_DOUBLE, 0, tag2, MPI_COMM_WORLD);
	}
}
```

Cuda MPI Staged vs Cuda Aware (Direct)



Cuda MPI Staged vs Cuda Aware (Direct)



Libraries

- Writing cuda functions is relatively easy
- Getting cuda functions to work well is very difficult
- Use libraries when you can
  - NVIDIA has several
  - Many third party libs
  - Here we will look at NVIDIA libs and compare to Cray and Intel CPU libs

Linear Solve

```
[tkaiser2@x3104c0s41b0n0 jun20]$cd cudalib/factor/
[tkaiser2@x3104c0s41b0n0 factor]$ls
cpu.C  cusolver_getrf_example.cu  cusolver_utils.h  doit  doswift
[tkaiser2@x3104c0s41b0n0 factor]$cat doit
: Size of our matrix to solve
export MSIZE=4500

: Start from a known module state, the default
: We are going to Cray libsci version with the GPU
: environment even though it does not use GPUs
: Start from a known module state, the default
module purge
myrestore

: Load modules
#module unload PrgEnv-cray/8.5.0
#module unload nvhpc/24.1

ml PrgEnv-gnu/8.4.0
ml cuda

# Here we build the CPU version with libsci We don't actaully use Cuda but the compiler wants it
CC  -DMINE=$MSIZE  -fopenmp -march=native cpu.C -o invert.libsci

: << ++++
 Compile our GPU programs.
 The module nvhpc-native gives us access to Nvidia's compilers
 nvc, nvc++, nvcc, nvfortran as well as the Portland Group
 compilers which are actually links to these.
++++
#ml nvhpc-native
ml nvhpc-stdalone
: GPU version with libcusolver
export L1=$NVHPC_ROOT/math_libs/lib64
export L3=$NVHPC_ROOT/REDIST/cuda/12.3/targets/x86_64-linux/lib
nvcc  -DMINE=$MSIZE -L$L1 -lcusolver -L$L3 -lnvJitLink cusolver_getrf_example.cu -o invert.gpu
```

We are going to do some linear solves. cusolver_getrf_example.cu is a slight modification of one of NVIDIA's examples. cpu.C uses lapack routines.

We'll build cpu.C against Cray's (libsci) and Intel's (MKL) libraries. The NVIDIA example uses NVIDIA's tools.

Here we just just build the the libsci version using one of the PrgEnv compilers

We build the GPU version with nvhpc.

Linear Solve

Running the libsci and gpu version

```
export OMP_NUM_THREADS=32
echo
echo
echo ++++++++++++++++++++++
echo running libsci version
echo ++++++++++++++++++++++
./invert.libsci

for GPU in 0 1 2 3 ; do
echo
echo
echo ++++++++++++++++++++++
echo running gpu version on GPU $GPU
echo ++++++++++++++++++++++
: invert.gpu will read the GPU on which to run from the command line
./invert.gpu $GPU
done
```

Linear Solve

For the Intel (MKL) version we load the CPU environment.

```
: We are going to compile the Intel version using
: the CPU environment
myrestore
source /nopt/nrel/apps/cpu_stack/env_cpe23.sh
ml intel-oneapi-mkl
ml intel-oneapi-compilers
icpx  -DMINE=$MSIZE -qopenmp -D__INTEL__ -march=native cpu.C -mkl -lmkl_rt -o invert.mkl

echo
echo
echo ++++++++++++++++++++++
echo running MKL version
echo ++++++++++++++++++++++

./invert.mkl

module unload  intel-oneapi-compilers
module unload intel-oneapi-mkl

unset L1
unset L3
unset OMP_NUM_THREADS
unset MSIZE
```

FFT

```
[tkaiser2@x3104c0s41b0n0 jun20]$cd cudalib/fft
[tkaiser2@x3104c0s41b0n0 fft]$ls
3d_mgpu_c2c_example.cpp  cufft_utils.h  doit  doswift  fftw3d.c
[tkaiser2@x3104c0s41b0n0 fft]$cat doit
: Start from a known module state, the default
module purge
myrestore


: Load modules
#module unload nvhpc/24.1
#module unload PrgEnv-cray/8.5.0

if [ -z ${MYGCC+x} ]; then module load gcc ; else module load $MYGCC ; fi
ml nvhpc-stdalone

ml 2>&1 | grep gcc-stdalone/13.1.0 ; if [ $? -eq 0 ]  ; then echo REPLACING gcc-stdalone/13.1.0 ; ml gcc-stdalone/12.3.0 ; fi

: << ++++
 Compile our GPU programs.
 The module nvhpc-native gives us access to Nvidia's compilers
 nvc, nvc++, nvcc, nvfortran as well as the Portland Group
 compilers which are actually links to these.
++++

nvcc -O3 -forward-unknown-to-host-compiler  --generate-code=arch=compute_90,code=[compute_90,sm_90] -std=c++11 -x cu 3d_mgpu_c2c_example.cpp -c
export L1=$NVHPC_ROOT/REDIST/math_libs/12.3/targets/x86_64-linux/lib
nvcc  -o 3dfft 3d_mgpu_c2c_example.o -L$L1 -lcufft
```

Here we are doing a fft on a cube. The GPU version will work on multiple GPUs. For the CPU version we call Cray's libsci version of fftw. We're using on of the NVIDIA's toolsets for the GPU version

FFT

```
: Run our program on a cube. The first parameter gives our cube size.
: 2048 should work on the H100s.
: Second parameter determines which algorithm runs first 1 GPU version or 4 GPU version
echo
echo
for DOIT in `seq 1 4` ; do
  echo set $DOIT
  echo ++++++++++++++
  echo RUN SINGLE GPU VERSION FIRST
  ./3dfft 512 1
  echo
  echo
  echo ++++++++++++++
  echo RUN FOUR GPU VERSION FIRST
  ./3dfft 512 2
  echo
  echo
done
```

Running our GPU version. Run our GPU version on a single and 4 GPUs. We do this twice in opposite orders

FFT

```
: Build and run a fftw version
module purge
myrestore
#module unload nvhpc/24.1
#module unload PrgEnv-cray/8.5.0
ml  PrgEnv-cray/8.4.0

ml cray-fftw
ml cuda
cc -O3 fftw3d.c -o fftw3.exe

echo
echo
echo +++++++++++++++
echo run fftw libsci version
./fftw3.exe 512
```

FFTW from Craylibsci

FFT

```
: Build and run a fftw version
module restore
source /nopt/nrel/apps/gpu_stack/env_cpe23.sh
ml cray-fftw
cc -O3 fftw3d.c -o fftw3.exe


echo
echo
echo ++++++++++++++
echo run fftw libsic version
./fftw3.exe 512
```

For the CPU version we call Cray's libsci version of fftw.

Summary

- Showed how to build and run many types of GPU enabled applications on Kestrel
- Provided a repo and tar ball for the examples
  - Xxxxxx
  - Xxxxxx
- Things will change. These scripts might not work in after updates
- Future work
  - Keep the scripts current
  - Multi node multi GPU cuda library examples

Running and Building Horovod on Kestrel's H100 GPU nodes

Timothy H. Kaiser, Ph.D.
tkasier2@nrel.gov

Horovod

Horovod is a distributed deep learning training framework for **TensorFlow**, Keras, PyTorch, and Apache MXNet. The goal of Horovod is to make distributed deep learning fast and easy to use.

See: https://horovod.readthedocs.io/en/stable/summary_include.html

Our scripts and documentation are at:

```
kestrel : /nopt/nrel/apps/horovod
```

Quick Start

- The first part of this document shows how to run Horovod with TensorFlow. As of this writing getting horovod to work with PyTorch on Kestrel is an ongoing effort.

- If you just want to try it you can:

```
cp /nopt/nrel/apps/horovod/testrun
sbatch testrun
```

- You may need to add your account and a reservation to the sbatch line.

Notes about what we are going to see

- Horovod and tensorflow are python packages. The version of python referenced in the script also contains: mpi4py, scipy, numpy, matplotlib. mpi4py is built against CrayMPI. jupyter-lab is also available.

- The python executable was created based on another preinstalled version of python using the venv https://docs.python.org/3/library/venv.html module. That is, it is a virtual environment.

- There is more about this in the Building section below. Virtual environments are enabled by sourcing the activate command in the python's bin directory.

Run script

```
#!/bin/bash

#SBATCH --job-name="runhorovod"
#SBATCH --nodes=2
#SBATCH --exclusive
#SBATCH --mem=0
#SBATCH --time=00:15:00
#SBATCH --partition=gpu-h100
##SBATCH --reservation=h100-testing
#SBATCH --gres=gpu:h100:4

# Make a directory for our run and go there
mkdir $SLURM_JOBID
cd $SLURM_JOBID

# Save out environment
printenv > env

# Save our script
cat $0 > script

# A useful function
addlib ()
{
    export
LD_LIBRARY_PATH=$1:$LD_LIBRARY_PATH

}
```

The first section just contains the slurm header.

We create a new directory for the run and save our runtime environment and run script.

Finally we define a shortcut function for adding library paths.

#1

Run script

```
# Set up a newer environment
module restore
source /nopt/nrel/apps/gpu_stack/env_cpe23.sh

# Activate our python
# This version o python contains many useful
# packages in addition to horovod.  The mpi4py
# version is built against CrayMPI.
if [ -z "$NEWPY" ]  ; then
#NEWPY=/nopt/nrel/apps/horovod/052124/3.11c
NEWPY=/nopt/nrel/apps/horovod/052124/3.11.7c
fi
. $NEWPY/bin/activate
echo running `which python`


# Need these modules for cuda stuff
ml PrgEnv-nvhpc
ml cudnn

# Other libs and settings required
addlib /nopt/nrel/apps/horovod/TensorRT/TensorRT-10.0.0.6/lib
addlib /nopt/nrel/apps/gpu_stack/compilers/03-24/.../12.2/nccl/lib
export XLA_FLAGS=--xla_gpu_cuda_data_dir=/nopt/cuda/12.3
```

Tensorflow and horovod both need several libraries and environmental variable settings to work properly.

Here we load the Cray/Nvidia programming environment and the extra cuda library cudnn.

Then we add two additional library paths and set one environmental variable. (We have truncated the path here so it fits on the slide.)

#2

Run script

Finally, we can run our examples. The first example is a glorified "hello world" program in mpi4py. With 128 tasks-per-node it takes a long time to launch. The TPN could be reduced.

The second example is pure tensorflow. It is not even a AI/ML code. It just is a benchmark for vector operations.

We then print out the horovod build information. It may say that it is build with PyTorch support but this is not currently working properly on Kestrel.

```
# Run some tests
export EXAMPLES=/nopt/nrel/apps/horovod/052124/examples

# mpi4py test
srun --tasks-per-node=128 $EXAMPLES/report.py |  sort -nk4,4  >
mpi4py.out


# Simple tf
python  $EXAMPLES/tfvector.py > vectors 2> vectors.info

# Horovod build info
horovodrun --check-build > horovod.info
```

#3

Run script

Horovod was built with support for doing communications with MPI.

On Kestrel this allows us to launch using srun.

In the first case we launch on a single node using all 4 GPUs. Then we run on two nodes if they are available.

```
# Run horovod with srun
srun -N 1 -n 4 python $EXAMPLES/new.py > four 2>four.info

# If there are at least 2 nodes run on 2
if [[ $SLURM_NNODES -gt 1 ]] ; then
  rm -rf checkpoint*
  srun -N 2 -n 8 python $EXAMPLES/new.py > eight 2>eight.info
fi
```

#4

Run script

Horovod was built with support for doing communications with gloo. The launch line is a bit more complicated.

NT is the total number of tasks we will run, 4 per node.

We need to create the nodes variable which contains a list of nodes and GPUs on which we want to run.

```
# We can also run with gloo.
# We need to generate the command line with node and gpu list.
export GPN=4
export NT=`expr $GPN '*' $SLURM_NNODES`

nodes=""
for x in `scontrol show hostname` ; do    nodes=$nodes,$x:$GPN; done
nodes=`echo $nodes | sed s/,//`
echo $nodes

rm -rf checkpoint*
horovodrun --gloo   -np $NT  -H $nodes  python $EXAMPLES/new.py > gloo 2>gloo.info
```

#5

Building Horovod

- The script /nopt/nrel/apps/horovod/052124/buildhv was used to create an instance of python with tensorflow and horovod installed.
- The script does five things.
  1. It creates a virtual python environment.
  2. It then installs a number of "standard" modules in the new python.
  3. It sets up the bash environment to be able to install tensorflow and horovod.
  4. Installs tensorflow and horovod with pip.
  5. Runs the examples discussed above.

Script Header

```
#!/bin/bash

#SBATCH --job-name="buildhorovod"
#SBATCH --nodes=2
#SBATCH --exclusive
#SBATCH --mem=0
#SBATCH --time=04:00:00
#SBATCH --partition=gpu-h100
#SBATCH --reservation=h100-testing
#SBATCH --gres=gpu:h100:4
#SBATCH --account=hpcapps

# save our script
cat $0 > script.$SLURM_JOBID

#export MYPYTHON=/nopt/nrel/apps/horovod/052124/pythons/052124_a/opt/linux-rhel8-zen3/gcc-12.2.1/python-3.11.7-5hnt7vwspc6ji7maaj7ll34dun35hcom/bin
export NEWPY=/nopt/nrel/apps/horovod/052124/3.11c
```

A standard slurm header.

We are saving a copy of our script.

More on MYPYTHON and NEWPY in two slides.

Some Functions

```
# some useful functions
addlib ()
{
    export LD_LIBRARY_PATH=$1:$LD_LIBRARY_PATH
}

addpath ()
{
    export PATH=$1:$PATH
}

rmpath ()
{
    lpaths=`echo $PATH | tr ":" " "`;
    export OPATH=$LD_LIBRARY_PATH;
    npath="";
    for l in $lpaths;
    do
        echo $l | grep --color=auto -v $1 > /dev/null;
        if [ $? -eq 0 ]; then
            npath=`echo $l | grep -v $1`:$npath;
        fi;
    done;
    export PATH=$npath
}
```

Some useful functions for modifying paths.

pipit updates pip and installs a bunch of useful things.

```
pipit () {
        rm -rf get-pip.py
        wget  https://bootstrap.pypa.io/get-pip.py
        which python
        python get-pip.py
        pip install --upgrade  --no-cache-dir pip
        pip3 install  --no-cache-dir matplotlib
        pip3 install  --no-cache-dir pandas
        pip3 install  --no-cache-dir scipy
        pip3 install  --no-cache-dir jupyterlab
        pip3 install  --no-cache-dir reframe-hpc
}
```

Set up a new python

```
#set up a newer environment
source /nopt/nrel/apps/gpu_stack/env_cpe23.sh
module list

# create a new python environment based on an existing one
if [ -z "$NEWPY" ]  ; then
NEWPY=/nopt/nrel/apps/horovod/052124/3.11a
fi
rm -rf $NEWPY
if [ -z "$MYPYTHON" ]  ; then
        ml cray-python
else
        addpath $MYPYTHON
fi
echo "old python " `which python`

python -m venv $NEWPY

# unload old python and activate our new one
module unload cray-python || rmpath $MYPYTHON
. $NEWPY/bin/activate
echo "new python " `which python`

# add important stuff to our python
pipit
```

This block of the script is not as complicated as it appears. It checks if you have defined two environmental variables and if not sets them to defaults.

We create a virtual python environment based on an existing python. $NEWPY points to the python we are creating and $MYPYTHON points to an existing python, Cray python by default

The *python -m venv $NEWPY* command actually creates the new instance. The module unload line removes our old python from out path. Then the new python is activated.

Finally we pip install useful things using our function we defined above.

Set up a new python

```
# some of tim's utilities
wget https://raw.githubusercontent.com/timkphd/examples/master/tims_tools/tymer -O tymer.py
wget https://raw.githubusercontent.com/timkphd/examples/master/tims_tools/setup.py
python setup.py install
mv tymer.py  $(dirname `which python`)/tymer
chmod 755  $(dirname `which python`)/tymer
rm -rf tymer* setup.py
```

```
wget https://raw.githubusercontent.com/timkphd/examples/master/mpi/mpi4py/spam.c
wget https://raw.githubusercontent.com/timkphd/examples/master/mpi/mpi4py/setup.py
python setup.py install
rm -rf spam.c setup.py
rm -rf build dist PackageName.egg-info
```

These are not critical to horovod. The "tymer" package has a number of simple utilities. The one used here is a nice wall clock timer.

spam.c has a routine to find the core on which a task is running

mpi4py

```
ml PrgEnv-gnu
pip  --no-cache-dir install mpi4py
```

Build mpi4py using Cray's MPI

mpi4py

```
ml PrgEnv-gnu
pip  --no-cache-dir install mpi4py
```

Build mpi4py using Cray's MPI

Get ready for Tensorflow and horovod

```
# need these modules for cuda stuff
ml PrgEnv-nvhpc
ml cudnn
module list

addlib /nopt/nrel/apps/horovod/TensorRT/TensorRT-10.0.0.6/lib
addlib /nopt/nrel/apps/gpu_stack/compilers/03-24/spack/opt/spack/linux-rhel8-zen3/gcc-12.3.0/
nvidia/hpc_sdk/Linux_x86_64/23.9/REDIST/cuda/12.2/targets/x86_64-linux/lib
export MPIROOT=/opt/cray/pe/mpich/8.1.28/ofi/nvidia/23.3
addpath $MPIROOT/bin
addlib $MPIROOT/lib
addlib /nopt/nrel/apps/gpu_stack/compilers/03-24/spack/opt/spack/linux-rhel8-zen3/gcc-12.3.0/
nvidia/hpc_sdk/Linux_x86_64/23.9/REDIST/comm_libs/12.2/nccl/lib
export CUDA_DIR=/nopt/nrel/apps/gpu_stack/compilers/03-24/spack/opt/spack/linux-rhel8-zen3/
gcc-12.3.0/nvidia/hpc_sdk/Linux_x86_64/23.9/compilers
```

Finish setting up our environment for the tensorflow and horovod packages.

We are using PrgEnv-nvhpc for Cray's MPI and NVIDIA's cuda.

The extra "adds" are for paths that don't normally get set by module loads.

Tensorflow

```
# finally install tf and horovod
# as of 05/20/24 the latest version of tf does not work with the
# latest version of horovod so we do 2.15.0
pip --no-cache-dir install tensorflow==2.15.0
pip --no-cache-dir install torch
pip --no-cache-dir install mxnet
```

Install tensorflow.

We also install torch/mxnet but I have not been able to get it to work with horovod.

Horovod

```
export CUDA_ARCHITECTURES=90
export CMAKE_CUDA_ARCHITECTURES=90
export HOROVOD_WITH_GLOO=1
export HOROVOD_WITH_MPI=1
export HOROVOD_WITH_TENSORFLOW=1
export HOROVOD_WITH_PYTORCH=1
export HOROVOD_GPU_OPERATIONS=NCCL
export HOROVOD_BUILD_CUDA_CC_LIST=90
export HOROVOD_WITH_PYTORCH=1
export HOROVOD_WITH_PYTORCH=0
export HOROVOD_WITH_MXNET=0
export HOROVOD_WITHOUT_MXNET=1
export HOROVOD_WITHOUT_PYTORCH=1
export HOROVOD_CUDA_HOME=/nopt/cuda/12.3
export CUDAToolkit_ROOT=/nopt/nrel/apps/gpu_stack/compilers/03-24/spack/opt/spack/linux-rhel8-
zen3/gcc-12.3.0/nvidia/hpc_sdk/Linux_x86_64/23.9/cuda/12.2/targets/x86_64-linux
export HOROVOD_NCCL_HOME=/nopt/nrel/apps/gpu_stack/compilers/03-24/spack/opt/spack/linux-rhel8-
zen3/gcc-12.3.0/nvidia/hpc_sdk/Linux_x86_64/23.9/comm_libs/12.2/nccl
#pip --no-cache-dir install horovod[tensorflow]
pip --no-cache-dir install horovod[tensorflow,torch]
```

Final install of horovod.

After this, the rest of the script just runs the examples we looked at before.

Summary

- Showed how to run tensorflow and horovod with tensorflow on Kestrel
  - /nopt/nrel/apps/horovod/testrun
- Showed how to create a virtual environment in python and pip install a number of important packages.
- Showed how to install tensorflow and horovod
  - /nopt/nrel/apps/horovod/052124/buildhv
- Future work
  - Get it working with torch