6. Polaris

6.1 Getting Started on Polaris

6.1.1 Logging Into Polaris

To log into Polaris:

```
ssh <username>@polaris.alcf.anl.gov
```

Then, type in the password from your CRYPTOCard/MobilePASS+ token.

6.1.2 Hardware Overview

An overview of the Polaris system including details on the compute node architecture is available on the Machine Overview page.

6.1.3 Compiling Applications

Users are encouraged to read through the Compiling and Linking Overview page and corresponding pages depending on the target compiler and programming model.

6.1.4 Submitting and Running Jobs

Users are encouraged to read through the Running Jobs with PBS at the ALCF page for information on using the PBS scheduler and preparing job submission scripts. Some example job submission scripts are available on the Example Job Scripts page as well.

6.1.5 Lustre File Striping

In addition to the content above, here is a document on Lustre File Striping Basics.

- Lustre File Striping Basics

6.1.6 Proxy

If the node you are on doesn’t have outbound network connectivity, add the following to your ~/.bash_profile file to access the proxy host

```
# proxy settings
export HTTP_PROXY="http://proxy-01.pub.alcf.anl.gov:3128"
export HTTPS_PROXY="http://proxy-01.pub.alcf.anl.gov:3128"
export http_proxy="http://proxy-01.pub.alcf.anl.gov:3128"
export https_proxy="http://proxy-01.pub.alcf.anl.gov:3128"
export ftp_proxy="http://proxy-01.pub.alcf.anl.gov:3128"
export no_proxy="admin,polaris-adminvm-01,localhost,*.cm.polaris.alcf.anl.gov,polaris-*,*.polaris.alcf.anl.gov,*.alcf.anl.gov"
```

6.1.7 Getting Assistance

Please direct all questions, requests, and feedback to support@alcf.anl.gov.

6.2 Known Issues

This is a collection of known issues that have been encountered during Polaris's early user phase. Documentation will be updated as issues are resolved.

1. The `nsys` profiler packaged with `nvhpc/21.9` in some cases appears to be presenting broken timelines with start times not lined up. The issue does not appear to be present when `nsys` from `cudatoolkit-standalone/11.2.2` is used. We expect this to no longer be an issue once `nvhpc/22.5` is made available as the default version.
2. With `PrgEnv-nvhpc/8.3.3`, if you are using `nvcc` to indirectly invoke `nvc++` and compiling C++17 code (as, for example, in building Kokkos via `nvcc_wrapper`), you will get compilation errors with C++17 constructs. See our documentation on NVIDIA Compilers for a workaround.
3. `PrgEnv-nvhpc/8.3.3` currently loads the `nvhpc/21.9` module, which erroneously has the following lines:

   ```
   setenv("CC","/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/compilers/bin/nvc")
   setenv("CXX","/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/compilers/bin/nvc++")
   setenv("FC","/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/compilers/bin/nvfortran")
   setenv("F90","/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/compilers/bin/nvfortran")
   setenv("F77","/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/compilers/bin/nvfortran")
   setenv("CC","cpp")
   ```

   In particular, the final line can cause issues for C-based projects (e.g. CMake may complain because the `cpp` C preprocessor is not a compiler). We recommend running the following in such cases:

   ```
   unset CC
   unset F77
   unset CXX
   unset FC
   unset F90
   ```

4. Cray MPICH may exhibit issues when MPI ranks call `fork()` and are distributed across multiple nodes. The process may hang or throw a segmentation fault.

   In particular, this can manifest in hangs with PyTorch+Horovod with a `DataLoader` with multithreaded workers and distributed data parallel training on multiple nodes. We have built a module `conda/2022-09-08-hvd-nccl` which includes a Horovod built without support for MPI. It uses

NCCL for GPU-GPU communication and Gloo for coordination across nodes.

`export IBV_FORK_SAFE=1` may be a workaround for some manifestations of this bug; however it will incur memory registration overheads. It does not fix the hanging experienced with multithreaded dataloading in PyTorch+Horovod across multiple nodes with `conda/2022-09-08`, however (instead prompting a segfault).

This incompatibility also may affect Parsl; see details in the Special notes for Polaris section of the Parsl page.

5. For batch job submissions, if the parameters within your submission script do not meet the parameters of any of the execution queues (`small`, ..., `backfill-large`) you might not receive the "Job submission" error on the command line at all, and the job will never appear in history `qstat -xu <username>` (current bug in PBS). E.g. if a user submits a script to the `prod` routing queue requesting 10 nodes for 24 hours, exceeding "Time Max" of 6 hrs of the `small` execution queue (which handles jobs with 10-24 nodes), then it may behave as if the job was never submitted.
6. Job scripts are copied to temporary locations after `qsub` and any changes to the original script while the job is queued will not be reflected in the copied script. Furthermore, `qalter` requires `-A <allocation name>` when changing job properties. Currently, there is a request for a `qalter`-like command to trigger a re-copy of the original script to the temporary location.

6.3 Hardware Overview

6.3.1 Polaris

Polaris is a 560 node HPE Apollo 6500 Gen 10+ based system. Each node has a single 2.8 GHz AMD EPYC Milan 7543P 32 core CPU with 512 GB of DDR4 RAM and four NVIDIA A100 GPUs connected via NVLink, a pair of local 1.6TB of SSDs in RAID0 for the users use, and a pair of Slingshot network adapters. They are currently Slingshot 10, but are scheduled to be upgraded to Slingshot 11 in 2023. There are two nodes per chassis, seven chassis per rack, and 40 racks for a total of 560 nodes. More detailed specifications are as follows:

Polaris Compute Nodes

POLARIS COMPUTE	DESCRIPTION	PER NODE	AGGREGATE
Processor (Note 1)	2.8 GHz 7543P	1	560
Cores/Threads	AMD Zen 3 (Milan)	32/64	17,920/35,840
RAM (Note 2)	DDR4	512 GiB	280 TiB
GPUS	NVIDIA A100	4	2240
Local SSD	1.6 TB	2/3.2 TB	1120/1.8PB

Note 1: 256MB shared L3 cache, 512KB L2 cache per core, 32 KB L1 cache per core Note 2: 8 memory channels rated at 204.8 GiB/s

Polaris A100 GPU Information

DESCRIPTION	A100 PCIe	A100 HGX (Polaris)
GPU Memory	40 GiB HBM2	160 GiB HBM2
GPU Memory BW	1.6 TB/s	6.4 TB/s
Interconnect	PCIe Gen4 64 GB/s	NVLink 600 GB/s
FP 64	9.7 TF	38.8 TF
FP64 Tensor Core	19.5 TF	78 TF
FP 32	19.5 TF	78 TF
BF16 Tensor Core	312 TF	1.3 PF
FP16 Tensor Core	312 TF	1.3 PF
INT8 Tensor Core	624 TOPS	2496 TOPS
Max TDP Power	250 W	400 W

Polaris Device Affinity Information

CPU Affinity	NUMA Affinity		GPU0	GPU1	GPU2	GPU3	mlx5_0	mlx5_1
24-31,56-63	3	GPU0	X	NV4	NV4	NV4	SYS	SYS
16-23,48-55	2	GPU1	NV4	X	NV4	NV4	SYS	PHB
8-15,40-47	1	GPU2	NV4	NV4	X	NV4	SYS	SYS
0-7,32-39	0	GPU3	NV4	NV4	NV4	X	PHB	SYS
		mlx5_0	SYS	SYS	SYS	PHB	X	SYS
		mlx5_1	SYS	PHB	SYS	SYS	SYS	X

LEGEND:

X = Self **SYS** = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI) **NODE** = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node **PHB** = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU) **PXB** = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge) **PIX** = Connection traversing at most a single PCIe bridge **NV#** = Connection traversing a bonded set of # NVLinks

Links to detailed NVIDIA A100 documentation: - NVIDIA A100 Tensor Core GPU Architecture - NVIDIA Ampere Architecture In-Depth

Login nodes

There are four login nodes available to users for editing code, building code, submitting / monitoring jobs, checking usage (`sbank`), etc.. Their full hostnames are `polaris-login-N.hsn.cm.polaris.alcf.anl.gov` for `N` equal to `01` through `04`; there are an additional two login nodes that are not user-accessible which are used for running services such as JupyterHub. The various compilers and libraries are present on the logins, so most users should be able to build their code. However, if your build requires the physical presence of the GPU, you will need to build on a compute node.

All users share the same login nodes so please be courteous and respectful of your fellow users. For example, please do not run computationally or IO intensive pre- or post-processing on the logins and keep the parallelism of your builds to a reasonable level.

POLARIS LOGIN	DESCRIPTION	PER NODE	AGGREGATE
Processor (Note 1)	2.0 GHz 7713	2	12
Cores/Threads	AMD Zen 3 (Milan)	128/256	768/1536
RAM (Note 2)	DDR4	512 GiB	3 TiB
GPUs (Note 3)	No GPUs	0	0
Local SSD	None	0	0

Note 1: 256MB shared L3 cache, 512KB L2 cache per core, 32 KB L1 cache per core Note 2: 8 memory channels rated at 204.8 GiB/s per socket Note 3: If your build requires the physical presence of a GPU you will need to build on a compute node.

Gateway nodes

There are 50 gateway nodes. These nodes are not user accessible, but are used transparently for access to the storage systems. Each node has a single 200 Gbps HDR IB card for access to the storage area network. This gives a theoretical peak bandwidth of 1250 GB/s which

is approximately the aggregate bandwidth of the global file systems (1300 GB/s).

Storage

Polaris has access to the ALCF global file systems. Details on storage can be found here.

6.4 Compiling and Linking

6.4.1 Compiling and Linking Overview on Polaris

Compiling on Polaris Login and Compute Nodes

If your build system does not require GPUs for the build process, as is usually the case, compilation of GPU-accelerated codes is generally expected to work well on the Polaris login nodes. If your build system *does* require GPUs, you cannot yet compile on the Polaris login nodes, as they do not currently have GPUs installed. You may in this case compile your applications on the Polaris compute nodes. Do this by submitting an interactive single-node job, or running your build system in a batch job.

HOME FILE SYSTEM

Is it helpful to realize that there is a single `HOME` filesystem for users that can be accessed from the login and computes of each production resource at ALCF. Thus, users should be mindful of modifications to their environments (e.g. `.bashrc`) that may cause issues to arise due to differences between the systems.

An example is creating an alias for the `qstat` command to, for example, change the order of columns printed to screen. Users with such an alias that works well on Theta may run into issues using `qstat` on Polaris as the two system use different schedulers: Cobalt (Theta) and PBS (Polaris). Users with such modifications to their environments are encouraged to modify their scripts appropriately depending on `$hostname`.

Cray Programming Environment

The Cray Programming Environment (PE) uses three compiler wrappers for building software. These compiler wrappers should be used when building MPI-enabled applications.

- `cc` - C compiler
- `CC` - C++ compiler
- `ftn` - Fortran compiler

Each of these wrappers can select a specific vendor compiler based on the PrgEnv module loaded in the environment. The following are some helpful options to understand what the compiler wrapper is invoking.

- `--craype-verbose` : Print the command which is forwarded to the compiler invocation
- `--cray-print-opts=libs` : Print library information
- `--cray-print-opts=cflags` : Print include information

The output from these commands may be useful in build scripts where a compiler other than that invoked by a compiler wrapper is desired. Defining some variables as such may prove useful in those situations.

```
CRAY_CFLAGS=$(cc --cray-print-opts=cflags)
CRAY_LIB=$(cc --cray-print-opts=libs)
```

Further documentation and options are available via `man cc` and similar.

Compilers provided by Cray Programming Environments

The default programming environment on Polaris is currently `NVHPC`. The `GNU` compilers are available via another programming environment. The following sequence of `module` commands can be used to switch to the `GNU` programming environment (gcc, g++, gfortran) and also have `NVIDIA` compilers available in your path.

```
module swap PrgEnv-nvhpc PrgEnv-gnu
module load nvhpc-mixed
```

The compilers invoked by the Cray MPI wrappers are listed for each programming environment in the following table.

module	C	C++	Fortran
MPI Compiler Wrapper	cc	CC	ftn
PrgEnv-nvhpc	nvc	nvc++	nvfortran
PrgEnv-gnu	gcc	g++	gfortran

Note, while gcc and g++ may be available in the default environment, the `PrgEnv-gnu` module is needed to provide gfortran.

Additional Compilers Provided by ALCF

The ALCF additionally provides compilers to enable the OpenMP and SYCL programming models for GPUs via `LLVM` as documented here

Additional documentation for using compilers is available on the respective programming model pages: OpenMP and SYCL.

Linking

Dynamic linking of libraries is currently the default on Polaris. The Cray MPI wrappers will handle this automatically.

Notes on Default Modules

- `craype-x86-rome`: While the Polaris compute nodes currently have Milan CPUs, this module is loaded by default to avoid the `craype-x86-milan` module from adding a `zen3` target not supported in the default `nvhpc/21.9` compilers. The `craype-x86-milan` module is expected to be made default once a newer `nvhpc` version (e.g. 22.5) is made the default.
- `craype-accel-nvidia80`: This module adds compiler flags to enable GPU acceleration for `NVHPC` compilers along with gpu-enabled MPI libraries as it is assumed that the majority of applications to be compiled on Polaris will target the GPUs for acceleration. Users

building cpu-only applications may find it useful to unload this module to silence "gpu code generation" warnings.

Mixed C/C++ & Fortran Applications

For applications consisting of a mix of C/C++ and Fortran that also uses MPI, it is suggested that the programming environment chosen for Fortran be used to build the full application because of mpi.mod (and similar) incompatibilities.

Compiling for GPUs

It is assumed the majority of applications to be built on Polaris will make use of the GPUs. As such, the `craype-accel-nvidia80` module is in the default environment. This has the effect of the Cray compiler wrappers adding `-gpu` to the compiler invocation along with additional include paths and libraries. Additional compilers flags may be needed depending on the compiler and GPU programming model used (e.g. `-cuda`, `-acc`, or `-mp=gpu`).

This module also adds GPU Transport Layer (GTL) libraries to the link-line to support GPU-aware MPI applications.

Man Pages

For additional information on the Cray wrappers, please refer to the man pages.

```
man cc
man CC
man ftn
```

6.4.2 Programming Models on Polaris

The software environment on Polaris supports several parallel programming models targeting the CPUs and GPUs.

CPU Parallel Programming Models

The Cray compiler wrappers `cc`, `CC`, and `ftn` are recommended for MPI applications as they provide the needed include paths and libraries for each programming environment. A summary of available CPU parallel programming models and relevant compiler flags is shown below. Users are encouraged to review the corresponding man pages and documentation.

Programming Model	GNU	NVHPC	LLVM
OpenMP	-fopenmp	-mp	-fopenmp
OpenACC	--	-acc=multicore	--

Higher-level programming models such as Kokkos and Raja may also be used for CPU programming.

GPU Programming Models

A summary of available GPU programming models and relevant compiler flags is shown below for compilers that generate offloadable

code. Users are encouraged to review the corresponding man pages and documentation.

Programming Model	GNU	NVHPC	LLVM	ONEAPI
CUDA	--	-cuda [-gpu=cuda8.0,cc11.0]	--	--
HIP*	--	--	--	--
OpenACC	--	-acc	--	--
OpenCL*	--	--	--	--
OpenMP	--	-mp=gpu	-fopenmp-targets=nvptx64	--
SYCL	--	--	--	-fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend --cuda-gpu-arch=sm_80

Note, the `llvm` and `oneapi` modules are provided by ALCF to complement the compilers provided by the Cray PE on Polaris.

Higher-level programming models such as Kokkos and Raja may also be used for GPU programming.

OpenCL is supported, but does not require specific compiler flags per-se as the offloaded kernels are just-in-time compiled. Abstraction programming models, such as Kokkos, can be built on top of some of these programming models (see below).

A HIP compiler supporting the A100 GPUs is still to be installed on Polaris.

Mapping Programming Models to Polaris Modules

The table below offers some suggestions for how to get started setting up your environment on Polaris depending on the programming language and model. Note, mixed C/C++ and Fortran applications should choose the programming environment for the Fortran compiler because of mpi.mod and similar incompatibilities between Fortran-generated files from different compilers. Several simple examples for

testing the software environment on Polaris for different programming models are available in the ALCF GettingStart repo.

Note, users are encouraged to use `PrgEnv-nvhpc` instead of `PrgEnv-nvidia` as the latter will soon be deprecated in Cray's PE. They are otherwise identical pointing to compilers from the same NVIDIA SDK version.

Programming Language	**GPU Programming Model**	**Likely used Modules/ Compilers**	**Notes**
C/C++	CUDA	PrgEnv-nvhpc, PrgEnv-gnu, llvm	NVIDIA (nvcc, nvc, nvc++) and clang compilers do GPU code generation
C/C++	HIP	N/A	need to install with support for A100
C/C++	Kokkos	See CUDA	HIP, OpenMP, and SYCL/DPC++ also candidates
C/C++	OpenACC	PrgEnv-nvhpc	
C/C++	OpenCL	PrgEnv-nvhpc, PrgEnv-gnu, llvm	JIT GPU code generation
C/C++	OpenMP	PrgEnv-nvhpc, llvm	
C/C++	RAJA	See CUDA	HIP, OpenMP, and SYCL/DPC++ also candidates
C/C++	SYCL/DPC++	llvm-sycl	
Fortran	CUDA	PrgEnv-nvhpc	NVIDIA compiler (nvfortran) does GPU code generation; `gfortran` can be loaded via `gcc-mixed`
Fortran	HIP	N/A	need to install with support for A100
Fortran	OpenACC	PrgEnv-nvhpc	
Fortran	OpenCL	PrgEnv-nvhpc, PrgEnv-gnu	JIT GPU code generation
Fortran	OpenMP	PrgEnv-nvhpc	

6.4.3 Example Programs and Makefiles for Polaris

Several simple examples of building CPU and GPU-enabled codes on Polaris are available in the ALCF GettingStart repo for several programming models. If build your application is problematic for some reason (e.g. absence of a GPU), then users are encouraged to build and test applications directly on one of the Polaris compute nodes via an interactive job. The discussion below makes use of the `NVHPC` compilers in the default environment as illustrative examples. Similar examples for other compilers on Polaris are available in the ALCF GettingStarted repo.

CPU MPI+OpenMP Example

One of the first useful tasks with any new machine, scheduler, and job launcher is to ensure one is binding MPI ranks and OpenMP threads to the host cpu as intended. A simple HelloWorld MPI+OpenMP example is available here to get started with.

The application can be straightforwardly compiled using the Cray compiler wrappers.

```
CC -fopenmp main.cpp -o hello_affinity
```

The executable `hello_affinity` can then be launched in a job script (or directly in shell of interactive job) using `mpiexec` as discussed here.

```
#!/bin/sh
#PBS -l select=1:system=polaris
#PBS -l place=scatter
#PBS -l walltime=0:30:00
#PBS -l filesystems=home

# MPI example w/ 16 MPI ranks per node spread evenly across cores
NNODES=`wc -l < $PBS_NODEFILE`
NRANKS_PER_NODE=16
NDEPTH=4
NTHREADS=1

NTOTRANKS=$(( NNODES * NRANKS_PER_NODE ))
echo "NUM_OF_NODES= ${NNODES} TOTAL_NUM_RANKS= ${NTOTRANKS} RANKS_PER_NODE= ${NRANKS_PER_NODE} THREADS_PER_RANK= ${NTHREADS}"

mpiexec -n ${NTOTRANKS} --ppn ${NRANKS_PER_NODE} --depth=${NDEPTH} --cpu-bind depth ./hello_affinity
```

CUDA

Several variants of C/C++ and Fortran CUDA examples are available here that include MPI and multi-gpu examples.

One can use the Cray compiler wrappers to compile GPU-enabled applications as well. This example of simple vector addition uses the NVIDIA compilers.

```
CC -g -O3 -std=c++0x -cuda main.cpp -o vecadd
```

The `craype-accel-nvidia80` module in the default environment will add the `-gpu` compiler flag for `nvhpc` compilers along with appropriate include directories and libraries. It is left to the user to provide an additional flag to the `nvhpc` compilers to select the target GPU programming model. In this case, `-cuda` is used to indicate compilation of CUDA code. The application can then be launched within a batch job submission script or as follows on one of the compute nodes.

```
$ ./vecadd
# of devices= 4
  [0] Platform[ Nvidia ] Type[ GPU ] Device[ NVIDIA A100-SXM4-40GB ]
  [1] Platform[ Nvidia ] Type[ GPU ] Device[ NVIDIA A100-SXM4-40GB ]
  [2] Platform[ Nvidia ] Type[ GPU ] Device[ NVIDIA A100-SXM4-40GB ]
  [3] Platform[ Nvidia ] Type[ GPU ] Device[ NVIDIA A100-SXM4-40GB ]
Running on GPU 0!
Using single-precision

  Name= NVIDIA A100-SXM4-40GB
  Locally unique identifier=
  Clock Frequency(KHz)= 1410000
  Compute Mode= 0
  Major compute capability= 8
  Minor compute capability= 0
  Number of multiprocessors on device= 108
  Warp size in threads= 32
  Single precision performance ratio= 2

Result is CORRECT!! :)
```

GPU OpenACC

A simple MPI-parallel OpenACC example is available here. Compilation proceeds similar to the above CUDA example except for the use of the `-acc=gpu` compiler flag to indicate compilation of OpenACC code for GPUs.

```
CC -g -O3 -std=c++0x -acc=gpu -gpu=cc80,cuda11.0 main.cpp -o vecadd
```

In this example, each MPI rank sees all four GPUs on a Polaris node and GPUs are bound to MPI ranks round-robin within the application.

```
$ mpiexec -n 4 ./vecadd
# of devices= 4
Using single-precision

Rank 0 running on GPU 0!
Rank 1 running on GPU 1!
Rank 2 running on GPU 2!
Rank 3 running on GPU 3!

Result is CORRECT!! :)
```

If the application instead relies on the job launcher to bind MPI ranks to available GPUs, then a small helper script can be used to explicitly set `CUDA_VISIBLE_DEVICES` appropriately for each MPI rank. One example is available here where each MPI rank is similarly bound to a single GPU with round-robin assignment. The binding of MPI ranks to GPUs is discussed in more detail here.

GPU OpenCL

A simple OpenCL example is available here. The OpenCL headers and library are available in the NVHPC SDK and cuda toolkits. The environment variable `NVIDIA_PATH` is defined for the `PrgEnv-nvhpc` programming environment.

```
CC -o vecadd -g -O3 -std=c++0x  -I${NVIDIA_PATH}/cuda/include main.o -L${NVIDIA_PATH}/cuda/lib64 -lOpenCL
```

This simple example can be run on a Polaris compute node as follows.

```
$ ./vecadd
Running on GPU!
Using single-precision

    CL_DEVICE_NAME: NVIDIA A100-SXM4-40GB
    CL_DEVICE_VERSION: OpenCL 3.0 CUDA
    CL_DEVICE_OPENCL_C_VERSION: OpenCL C 1.2
    CL_DEVICE_MAX_COMPUTE_UNITS: 108
    CL_DEVICE_MAX_CLOCK_FREQUENCY: 1410
    CL_DEVICE_MAX_WORK_GROUP_SIZE: 1024

Result is CORRECT!! :)
```

GPU OpenMP

A simple MPI-parallel OpenMP example is available here. Compilation proceeds similar to the above examples except for use of the `-mp=gpu` compiler flag to indicated compilation of OpenMP code for GPUs.

```
CC -g -O3 -std=c++0x -mp=gpu -gpu=cc80,cuda11.0 -c main.cpp -o vecadd
```

Similar to the OpenACC example above, this code binds MPI ranks to GPUs in a round-robin fashion.

```
$ mpiexec -n 4 ./vecadd
# of devices= 4
Rank 0 running on GPU 0!
Rank 1 running on GPU 1!
Rank 2 running on GPU 2!
Rank 3 running on GPU 3!
```

```
Result is CORRECT!! :)
```

6.4.4 CCE Compilers on Polaris

The Cray Compiling Environment (CCE) compilers are available on Polaris via the `PrgEnv-cray` module.

The CCE compilers currently on Polaris only support AMD GPU targets for HIP and are thus not usable with the A100 GPUs.

The `nvhpc` and `llvm` compilers can be used for compiling GPU-enabled applications.

6.4.5 GNU Compilers on Polaris

The GNU compilers are available on Polaris via the `PrgEnv-gnu` and `gcc-mixed` modules. The `gcc-mixed` module can be useful when, for example, the `PrgEnv-nvhpc` compilers are used to compile C/C++ MPI-enabled code and gfortran is needed.

The GNU compilers currently on Polaris do not support GPU code generation and thus can only be used for compiling CPU codes.

The `nvhpc` and `llvm` compilers can be used for compiling GPU-enabled applications.

6.4.6 NVIDIA Compilers on Polaris

The NVIDIA compilers (`nvc` , `nvc++` , `nvcc` , and `nvfortran`) are available on Polaris via the `PrgEnv-nvhpc` and `nvhpc` modules. There is currently a `PrgEnv-nvidia` module available, but that will soon be deprecated in Cray's PE, thus it is not recommend for use.

The Cray compiler wrappers map to NVIDIA compilers as follows.

```
cc -> nvc
CC -> nvc++
ftn -> nvfortran
```

Users are encouraged to look through NVIDIA's documentation for the NVHPC SDK and specific information on the compilers, tools, and libraries.

Notes on NVIDIA Compilers

PGI COMPILERS

The NVIDIA programming environments makes available compilers from the NVIDIA HPC SDK. While the PGI compilers are available in this programming environment, it should be noted they are actually symlinks to the corresponding `NVIDIA` compilers.

```
pgcc -> nvc
pgc++ -> nvc++
pgf90 -> nvfortran
pgfortran -> nvfortran
```

While `nvcc` is the traditional CUDA C and CUDA C++ compiler for NVIDIA GPUs, the `nvc` , `nvc++` , and `nvfortran` compilers additionally target CPUs.

NVHPC SDK DIRECTORY STRUCTURE

Users migrating from CUDA toolkits to the NVHPC SDK may find it beneficial to review the directory structure of the `hpc-sdk` directory to find the location of commonly used libraries (including math libraries for the CPU). With the `PrgEnv-nvhpc` module loaded, the `NVIDIA_PATH`

environment variable can be used to locate the path to various NVIDIA tools, libraries, and examples.

- `compiler/bin` - cuda-gdb, ncu, nsys, ...
- `examples` - CUDA-Fortran, OpenMP, ...
- `comm_libs` - nccl, nvshmem, ...
- `compiler/libs` - blas, lapack, ...
- `cuda/lib64` - cudart, OpenCL, ...
- `math_libs/lib64` - cublas, cufft, ...

DIFFERENCES BETWEEN NVCC AND NVC/NVC++

For users that want to continue using `nvcc` it is important to be mindful of differences with the newer `nvc` and `nvc++` compilers. For example, the `-cuda` flag instructs `nvcc` to compile `.cu` input files to `.cu.cpp.ii` output files which are to be separately compiled, whereas the same `-cuda` flag instructs `nvc`, `nvc++`, and `nvfortran` to enable CUDA C/C++ or CUDA Fortran code generation. The resulting output file in each case is different (text vs. object) and one may see `unrecognized format error` when `-cuda` is incorrectly passed to `nvcc`.

KNOWN ISSUES AND WORKAROUNDS

If you are using `nvcc` to invoke `nvc++` and compiling C++17 code, and are seeing the following warning and unable to compile C++17 constructs:

```
polaris-login-01(~)> nvcc --std=c++17 -ccbin nvc++ ~/smalltests/bool_constant.cpp
nvcc warning : The -std=c++17 flag is not supported with the configured host compiler. Flag will be ignored.
"/home/zippy/smalltests/bool_constant.cpp", line 10: error: namespace "std" has no member class "bool_constant"
      : std::bool_constant<(UnaryPred<Ts>::value || ...)> {};
              ^

"/home/zippy/smalltests/bool_constant.cpp", line 10: error: class or struct definition is missing
      : std::bool_constant<(UnaryPred<Ts>::value || ...)> {};
                           ^

2 errors detected in the compilation of "/home/zippy/smalltests/bool_constant.cpp".
polaris-login-01(~)>
```

you will need to work around it by loading the latest cudatoolkit module atop PrgEnv-nvhpc:

```
module load cudatoolkit-standalone/11.6.2
```

6.4.7 LLVM Compilers on Polaris

This page is not about LLVM-based Cray Compiling Environment (CCE) compilers from `PrgEnv-cray` but about open source LLVM compilers. If LLVM compilers are needed without MPI support, simply load the `llvm` module.

Cray Programming Environment does not offer LLVM compiler support. Thus cc/CC/ftn compiler wrappers using LLVM compilers currently are not available. To use Clang with MPI, one can load the `mpiwrappers/cray-mpich-llvm` module which loads the following modules.

- `llvm`, upstream llvm compilers
- `cray-mpich`, MPI compiler wrappers mpicc/mpicxx/mpif90. mpif90 uses gfortran because flang is not ready for production use.
- `cray-pals`, MPI launchers mpiexec/aprun/mpirun

Limitation There is no GPU-aware MPI library linking support by default. If needed, users should manually add the GTL (GPU Transport Layer) library to the application link line.

OpenMP offload

When targeting the OpenMP or CUDA programming models for GPUs, the `cudatoolkit-standalone` module should also be loaded.

6.4.8 OneAPI Compilers and Support

The Intel OneAPI compiler and Codeplay plugins for Nvidia GPUs are available on Polaris. The oneAPI compilers are not enabled under the Cray Programming Environment system but can be used separately. Two oneAPI variants are provided, the first being a "release" version based on Intel's officially released oneAPI toolkit. Intel Release Notes

> **Note**
>
> The 2023.1 release of oneAPI Toolkit does not support oneMKL or oneDPL on Nvidia.

The other variant being a build of main from the open source. This variant will be more up-to-date at the risk of bugs and breakages based on code that has not undergone a full release cycle. The documentation is located on the SYCL page.

Compile and Link

OneAPI uses the clang (or icx/icpx wrapper) for compiling and linking for the Nvidia A100 SM80 architecture.

```
module load PrgEnv-nvhpc
module use /soft/compilers/oneapi/release/modulefiles
module load compiler
icpx -std=c++17 -fsycl -fsycl-targets=nvptx64-nvidia-cuda  -Xsycl-target-backend --cuda-gpu-arch=sm_80 test.cpp
```

```
harms@polaris-login-04:~/working/polaris/oneapi> icpx --version
Intel(R) oneAPI DPC++/C++ Compiler 2023.1.0 (2023.1.0.20230320)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /soft/compilers/oneapi/release/2023.1/compiler/2023.1.0/linux/bin-llvm
Configuration file: /soft/compilers/oneapi/release/2023.1/compiler/2023.1.0/linux/bin-llvm/../bin/icpx.cfg
```

Running

The library should select the GPU by default, but selection of the GPUs can be forced via the ONEAPI_DEVICE_SELECTOR

```
$ ONEAPI_DEVICE_SELECTOR=ext_oneapi_cuda:gpu ./a.out
```

or a specific GPU.

```
$ ONEAPI_DEVICE_SELECTOR=ext_oneapi_cuda:gpu:3 ./a.out
```

sycl-ls

Expected output of sycl-ls and which platforms are available.

```
harms@x3004c0s7b0n0:~> which sycl-ls
/soft/compilers/oneapi/release/2023.1/compiler/2023.1.0/linux/bin/sycl-ls

harms@x3004c0s7b0n0:~> sycl-ls
[opencl:acc:0] Intel(R) FPGA Emulation Platform for OpenCL(TM), Intel(R) FPGA Emulation Device 1.2 [2023.15.3.0.20_160000]
[opencl:cpu:1] Intel(R) OpenCL, AMD EPYC 7543P 32-Core Processor                3.0 [2023.15.3.0.20_160000]
[ext_oneapi_cuda:gpu:0] NVIDIA CUDA BACKEND, NVIDIA A100-SXM4-40GB 0.0 [CUDA 11.4]
[ext_oneapi_cuda:gpu:1] NVIDIA CUDA BACKEND, NVIDIA A100-SXM4-40GB 0.0 [CUDA 11.4]
[ext_oneapi_cuda:gpu:2] NVIDIA CUDA BACKEND, NVIDIA A100-SXM4-40GB 0.0 [CUDA 11.4]
[ext_oneapi_cuda:gpu:3] NVIDIA CUDA BACKEND, NVIDIA A100-SXM4-40GB 0.0 [CUDA 11.4]
```

6.5 Build Tools

6.5.1 CMake

CMake

CMake is a build configuration system that uses higher-level description files to automatically generate Makefiles.

CMAKE DOCUMENTATION

– CMake website

CMAKE ON POLARIS

To use CMake on Polaris, run

```
module use /soft/modulefiles
module load cmake
```

6.6 Running Jobs on Polaris

6.6.1 Queues

There are five production queues you can target in your qsub (`-q <queue name>`):

Queue Name	Node Min	Node Max	Time Min	Time Max	Notes
debug	1	2	5 min	1 hr	max 8 nodes in use by this queue ay any given time
debug-scaling	1	10	5 min	1 hr	max 1 job running/accruing/queued **per-user**
prod	10	496	5 min	24 hrs	Routing queue; See below
preemptable	1	10	5 min	72 hrs	max 20 jobs running/accruing/queued **per-project**; see note below
demand	1	56	5 min	1 hr	***By request only***; max 100 jobs running/accruing/queued **per-project**

Note: Jobs in the demand queue take priority over jobs in the preemptable queue. This means jobs in the preemptable queue may be preempted (killed without any warning) if there are jobs in the demand queue. Please use the following command to view details of a queue: `qstat -Qf <queuename>`

`prod` is routing queue and routes your job to one of the following six execution queues:

Queue Name	Node Min	Node Max	Time Min	Time Max	Notes
small	10	24	5 min	3 hrs	
medium	25	99	5 min	6 hrs	
large	100	496	5 min	24 hrs	
backfill-small	10	24	5 min	3 hrs	low priority, negative project balance
backfill-medium	25	99	5 min	6 hrs	low priority, negative project balance
backfill-large	100	496	5 min	24 hrs	low priority, negative project balance

- **Note 1:** You cannot submit to these queues directly, you can only submit to the routing queue "prod".
- **Note 2:** All of these queues have a limit of ten (10) jobs running/accruing **per-project**
- **Note 3:** All of these queues have a limit of one hundred (100) jobs queued (not accruing score) **per-project**
- **Note 4:** As of January 2023, it is recommended to submit jobs with a maximum node count of 476-486 nodes given current rates of downed nodes (larger jobs may sit in the queue indefinitely).

6.6.2 Running MPI+OpenMP Applications

Once a submitted job is running calculations can be launched on the compute nodes using `mpiexec` to start an MPI application.

Documentation is accessible via `man mpiexec` and some helpful options follow.

- `-n` total number of MPI ranks
- `-ppn` number of MPI ranks per node
- `--cpu-bind` CPU binding for application
- `--depth` number of cpus per rank (useful with `--cpu-bind`)
- `--env` set environment variables (`--env OMP_NUM_THREADS=2`)
- `--hostfile` indicate file with hostnames (the default is `--hostfile $PBS_NODEFILE`)

A sample submission script with directives is below for a 4-node job with 32 MPI ranks on each node and 8 OpenMP threads per rank (1 per CPU).

```
#!/bin/bash -l
#PBS -N AFFINITY
#PBS -l select=4:ncpus=256
#PBS -l walltime=0:10:00
#PBS -q debug-scaling
#PBS -A Catalyst

NNODES=`wc -l < $PBS_NODEFILE`
NRANKS=32 # Number of MPI ranks to spawn per node
NDEPTH=8 # Number of hardware threads per rank (i.e. spacing between MPI ranks)
NTHREADS=8 # Number of software threads per rank to launch (i.e. OMP_NUM_THREADS)

NTOTRANKS=$(( NNODES * NRANKS ))

echo "NUM_OF_NODES= ${NNODES} TOTAL_NUM_RANKS= ${NTOTRANKS} RANKS_PER_NODE= ${NRANKS} THREADS_PER_RANK= ${NTHREADS}"

cd /home/knight/affinity
mpiexec --np ${NTOTRANKS} -ppn ${NRANKS} -d ${NDEPTH} --cpu-bind depth -env OMP_NUM_THREADS=${NTHREADS} ./hello_affinity
```

6.6.3 Running GPU-enabled Applications

GPU-enabled applications will similarly run on the compute nodes using the above example script. - The environment variable `MPICH_GPU_SUPPORT_ENABLED=1` needs to be set if your application requires MPI-GPU support whereby the MPI library sends and receives data directly from GPU buffers. In this case, it will be important to have the `craype-accel-nvidia80` module loaded both when compiling your application and during runtime to correctly link against a GPU Transport Layer (GTL) MPI library. Otherwise, you'll likely see `GPU_SUPPORT_ENABLED is requested, but GTL library is not linked` errors during

runtime. - If running on a specific GPU or subset of GPUs is desired, then the `CUDA_VISIBLE_DEVICES` environment variable can be used. For example, if one only wanted an application to access the first two GPUs on a node, then setting `CUDA_VISIBLE_DEVICES=0,1` could be used.

Binding MPI ranks to GPUs

The Cray MPI on Polaris does not currently support binding MPI ranks to GPUs. For applications that need this support, this instead can be handled by use of a small helper script that will appropriately set `CUDA_VISIBLE_DEVICES` for each MPI rank. One example is available here where each MPI rank is similarly bound to a single GPU with round-robin assignment.

A example `set_affinity_gpu_polaris.sh` script follows where GPUs are assigned round-robin to MPI ranks.

```
#!/bin/bash -l
num_gpus=4
# need to assign GPUs in reverse order due to topology
# See Polaris Device Affinity Information:
# https://www.alcf.anl.gov/support/user-guides/polaris/hardware-overview/machine-overview/index.html
gpu=$((${num_gpus} - 1 - ${PMI_LOCAL_RANK} % ${num_gpus}))
export CUDA_VISIBLE_DEVICES=$gpu
echo “RANK= ${PMI_RANK} LOCAL_RANK= ${PMI_LOCAL_RANK} gpu= ${gpu}”
exec "$@"
```

This script can be placed just before the executable in the `mpiexec` command like so.

```
mpiexec -n ${NTOTRANKS} --ppn ${NRANKS_PER_NODE} --depth=${NDEPTH} --cpu-bind depth ./set_affinity_gpu_polaris.sh ./hello_affinity
```

Users with different needs, such as assigning multiple GPUs per MPI rank, can modify the above script to suit their needs.

6.6.4 Interactive Jobs on Compute Nodes

Here is how to submit an interactive job to, for example, edit/build/test an application Polaris compute nodes:

```
qsub -I -l select=1 -l filesystems=home:eagle -l walltime=1:00:00 -q debug
```

This command requests 1 node for a period of 1 hour in the debug queue, requiring access to the /home and eagle filesystems. After waiting in the queue for a node to become available, a shell prompt on a compute node will appear. You may then start building applications and testing gpu affinity scripts on the compute node.

NOTE: If you want to `ssh` or `scp` to one of your assigned compute nodes you will need to make sure your `$HOME` directory and your `$HOME/.ssh` directory permissions are both set to `700`.

6.6.5 Running Multiple MPI Applications on a node

Multiple applications can be run simultaneously on a node by launching several `mpiexec` commands and backgrounding them. For performance, it will likely be necessary to ensure that each application runs on a distinct set of CPU resources and/or targets specific GPUs. One can provide a list of CPUs using the `--cpu-bind` option, which when combined with `CUDA_VISIBLE_DEVICES` provides a user with specifying exactly which CPU and GPU resources to run each application on. In the example below, four instances of the application are simultaneously running on a single node. In the first instance, the application is spawning MPI ranks 0-7 on CPUs 24-31 and using GPU 0. This mapping is based on output from the `nvidia-smi topo -m` command and pairs CPUs with the closest GPU.

```
export CUDA_VISIBLE_DEVICES=0
mpiexec -n 8 --ppn 8 --cpu-bind list:24:25:26:27:28:29:30:31 ./hello_affinity &

export CUDA_VISIBLE_DEVICES=1
mpiexec -n 8 --ppn 8 --cpu-bind list:16:17:18:19:20:21:22:23 ./hello_affinity &

export CUDA_VISIBLE_DEVICES=2
mpiexec -n 8 --ppn 8 --cpu-bind list:8:9:10:11:12:13:14:15 ./hello_affinity &

export CUDA_VISIBLE_DEVICES=3
mpiexec -n 8 --ppn 8 --cpu-bind list:0:1:2:3:4:5:6:7 ./hello_affinity &

wait
```

6.6.6 Compute Node Access to the Internet

Currently, the only access the internet is via a proxy. Here are the proxy environment variables for Polaris:

```
export http_proxy="http://proxy-01.pub.alcf.anl.gov:3128"
export https_proxy="http://proxy-01.pub.alcf.anl.gov:3128"
export ftp_proxy="http://proxy-01.pub.alcf.anl.gov:3128"
```

In the future, though we don't have a timeline on this because it depends on future features in slingshot and internal software development, we intend to have public IP addresses be a schedulable resource. For instance, if only your head node needed public access

your select statement might looks something like: `-l select=1:pubnet=True+63`.

6.6.7 Controlling Where Your Job Runs

If you wish to have your job run on specific nodes form your select like this: `-l select=1:vnode=<node name1>+1:vnode=<node name2>...` . Obviously, that gets tedious for large jobs.

If you want to control the location of a few nodes, for example 2 out of 64, but the rest don't matter, you can do something like this: `-l select=1:vnode=<node name1>+1:vnode=<node name2>+62:system=foo`

Every node has a PBS resource called `tier0` with a rack identifier and `tier1` with a dragonfly group identifieer. If you want all your nodes grouped in a rack, you can add the group specifier `-l select=8:system=foo,place=scatter:group=tier0`. If you wanted everything in the same dragonfly group, replace `tier0` with `tier1`. Note that you have to also explicitly specify the place when you use group. If you wanted a specific rack or dragonfly group instead of any of them, you are back to the select: `-l select 10:tier0=x3001-g0`.

Network: Rack and Dragonfly Group Mappings

- Racks contain (7) 6U chassis; each chassis has 2 nodes for 14 nodes per rack
- The hostnames are of the form xRRPPc0sUUb[0|1]n0 where:
- RR is the row {30, 31, 32}
- PP is the position in the row {30 goes 1-16, 31 and 32 go 1-12}
- c is chassis and is always 0
- s stands for slot, but in this case is the RU in the rack and values are {1,7,13,19,25,31,37}
- b is BMC controller and is 0 or 1 (each node has its own BMC)
- n is node, but is always 0 since there is only one node per BMC

- So, 16+12+12 = 40 racks * 14 nodes per rack = 560 nodes.
- Note that in production group 9 (the last 4 racks) will be the designated on-demand racks
- The management racks are x3000 and X3100 and are dragonfly group 10
- The TDS rack is x3200 and is dragonfly group 11
- Each compute node will have a PBS resource named `tier0` which will be equal to the values in the table below. This allows you to group your jobs within a rack if you wish. There is also a resource called `tier1` which will be equal to the column headings. This allows you to group your jobs within a dragonfly group if you wish.

g0	**g1**	**g2**	**g3**	**g4**	**g5**	**g6**	**g7**	**g8**	**g9**
x3001-g0	x3005-g1	x3009-g2	x3013-g3	x3101-g4	x3105-g5	x3109-g6	x3201-g7	x3205-g8	x3209-g9
x3002-g0	x3006-g1	x3010-g2	x3014-g3	x3102-g4	x3106-g5	x3110-g6	x3202-g7	x3206-g8	x3210-g9
x3003-g0	x3007-g1	x3011-g2	x3015-g3	x3103-g4	x3107-g5	x3111-g6	x3203-g7	x3207-g8	x3211-g9
x3004-g0	x3008-g1	x3012-g2	x3016-g3	x3104-g4	x3108-g5	x3112-g6	x3204-g7	x3208-g8	x3212-g9

6.7 Applications and Libraries

6.7.1 Applications

Gromacs on Polaris

WHAT IS GROMACS?

GROMACS is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles. It is primarily designed for biochemical molecules like proteins, lipids, and nucleic acids that have a lot of complicated bonded interactions, but since GROMACS is extremely fast at calculating the nonbonded interactions (that usually dominate simulations) many groups are also using it for research on non-biological systems, e.g. polymers.

USING GROMACS AT ALCF

ALCF offers assistance with building binaries and compiling instructions for GROMACS. For questions, contact us at support@alcf.anl.gov.

BUILDING GROMACS

1. Download latest source code: http://manual.gromacs.org/documentation/2022.1/download.html
2. tar -xzf gromacs-2022.1.tar.gz
3. module swap PrgEnv-nvhpc PrgEnv-gnu
4. module load cudatoolkit-standalone/11.2.2
5. module load gcc/10.3.0
6. module load cmake
7. cd gromacs-2022.1
8. mkdir build
9.
```
cmake -DCMAKE_C_COMPILER=cc -DCMAKE_CXX_COMPILER=CC \
      -DBUILD_SHARED_LIBS=OFF -DGMX_BUILD_OWN_FFTW=ON \
```

```
        -DCMAKE_INSTALL_PREFIX=/path-to/gromacs-2022.1/build \
        -DGMX_MPI=ON -DGMX_OPENMP=ON -DGMX_GPU=CUDA \
        -DCUDA_TOOLKIT_ROOT_DIR=/soft/compilers/cudatoolkit/cuda-11.2.2
```

10. make –j 8
11. make install
12. The installed binary is `build/bin/gmx_mpi`.

RUNNING GROMACS ON POLARIS

Prebuilt Gromacs binaries can be found in the directory `/soft/applications/Gromacs/gromacs-2022.1`.

A sample pbs script follows that will run GROMACS on two nodes, using 4 MPI ranks per node, and each rank with four OpenMP threads. The PME kernel owns one MPI rank and one GPU per node, while the nonbonded kernel uses 3 MPI ranks and 3 GPUs per node.

```
#!/bin/sh
#PBS -l select=2:system=polaris
#PBS -l place=scatter
#PBS -l walltime=0:30:00
#PBS -q debug
#PBS -A PROJECT
#PBS -l filesystems=home:grand:eagle

cd ${PBS_O_WORKDIR}

module swap PrgEnv-nvhpc PrgEnv-gnu
module load cudatoolkit-standalone/11.2.2

export OMP_NUM_THREADS=4

mpirun --np 8 /soft/applications/Gromacs/gromacs-2022.1/gmx_mpi \
      mdrun -gputasks 0123 -nb gpu -pme gpu -npme 1 -ntomp 4 \
      -dlb yes -resethway -pin on -v deffnm step5_1 -g test.log
```

We strongly suggest that users try combinations of different numbers of nodes, MPI ranks per node, number of GPU tasks/devices, GPU task decomposition between nonbonded and PME kernels, and OMP threads per rank to find the optimal throughput for their particular workload.

LAMMPS

OVERVIEW

LAMMPS is a general-purpose molecular dynamics software package for massively parallel computers. It is written in an exceptionally clean style that makes it one of the more popular codes for users to extend and it currently has dozens of user-developed extensions.

For details bout the code and its usage, see the LAMMPS home page. This page provides information specific to running on Polaris at the ALCF.

USING LAMMPS AT ALCF

ALCF provides assistanc with build instructions, compiling executables, submitting jobs, and providing prebuilt binaries (upon request). A collection of Makefiles and submission scripts are available in the ALCF GettingStarted repo here. For questions, contact us at support@alcf.anl.gov.

HOW TO OBTAIN THE CODE

LAMMPS is an open-source code, which can be downloaded from the LAMMPS website.

BUILDING ON POLARIS USING KOKKOS PACKAGE

After LAMMPS has been downloaded and unpacked an ALCF filesystem, users should see a directory whose name is of the form `lammps-<version>`. One should then see the Makefile `lammps-<version>/src/MAKE/MACHINES/Makefile.polaris` in recent versions that can be used for compilation on Polaris. A copy of the Makefile is also available in the ALCF GettingStarted repo here. For older versions of LAMMPS, you may need to take an existing Makefile (e.g. Makefile.mpi) for your specific version of LAMMPS used and edit the top portion appropratiately to create a new Makefile.polaris files.

The top portion of `Makefile.polaris_kokkos_nvidia` used to build LAMMPS with the KOKKOS package using the NVIDIA compilers is shown as an example.

```
# polaris_nvidia = Flags for NVIDIA A100, NVIDIA Compiler, Cray MPICH, CUDA
# module load craype-accel-nvidia80
# make polaris_kokkos_nvidia -j 16

SHELL = /bin/sh

# ---------------------------------------------------------------------
# compiler/linker settings
# specify flags and libraries needed for your compiler

KOKKOS_DEVICES = Cuda,OpenMP
KOKKOS_ARCH = Ampere80
KOKKOS_ABSOLUTE_PATH = $(shell cd $(KOKKOS_PATH); pwd)
export NVCC_WRAPPER_DEFAULT_COMPILER = nvc++

CRAY_INC = $(shell CC --cray-print-opts=cflags)
CRAY_LIB = $(shell CC --cray-print-opts=libs)

CC =        $(KOKKOS_ABSOLUTE_PATH)/bin/nvcc_wrapper
CCFLAGS =  -g -O3 -mp -DLAMMPS_MEMALIGN=64 -DLAMMPS_BIGBIG
CCFLAGS += $(CRAY_INC)
SHFLAGS =   -fPIC
DEPFLAGS =  -M

LINK =      $(CC)
LINKFLAGS = $(CCFLAGS)
LIB = $(CRAY_LIB)
SIZE =      size
```

With the appropriate LAMMPS Makefile in place an executable can be compiled as in the following example, which uses the NVIDIA compilers.

```
module load craype-accel-nvidia80
cd lammps-<version>/src
make yes-KOKKOS
make polaris_kokkos_nvidia -j 16
```

RUNNING JOBS ON POLARIS

An example submission script for running a KOKKOS-enabled LAMMPS executable is below as an example. Additional information on LAMMPS application flags and options is described on the LAMMPS website.

```
#!/bin/sh
#PBS -l select=64:system=polaris
#PBS -l place=scatter
#PBS -l walltime=0:15:00
#PBS -l filesystems=home:grand:eagle
#PBS -q prod
#PBS -A Catalyst

export MPICH_GPU_SUPPORT_ENABLED=1

NNODES=`wc -l < $PBS_NODEFILE`

# per-node settings
NRANKS=4
NRANKSSOCKET=2
NDEPTH=8
NTHREADS=1
NGPUS=4

NTOTRANKS=$(( NNODES * NRANKS ))
```

```
EXE=/home/knight/bin/lammps_polaris_kokkos_nvidia
EXE_ARG="-in in.reaxc.hns -k on g ${NGPUS} -sf kk -pk kokkos neigh half neigh/qeq full newton on "

# OMP settings mostly to quiet Kokkos messages

MPI_ARG="-n ${NTOTRANKS} --ppn ${NRANKS} --depth=${NDEPTH} --cpu-bind depth --env OMP_NUM_THREADS=${NTHREADS} --env OMP_PROC_BIND=spread --env OMP_PLACES=cores"

COMMAND="mpiexec ${MPI_ARG} ${EXE} ${EXE_ARG}"
echo "COMMAND= ${COMMAND}"
${COMMAND}
```

PERFORMANCE NOTES

Some useful information on accelerator packages and expectations can be found on the LAMMPS website here.

OpenMM on Polaris

WHAT IS OPENMM?

OpenMM is a high-performance toolkit for molecular simulations that can be used as a stand-alone application or as a library. It provides a combination of flexibility (through custom forces and integrators), openness, and high-performance (especially on recent GPUs).

USING OPENMM AT ALCF

ALCF offers assistance with building binaries and compiling instructions for OpenMM. For questions, contact us at support@alcf.anl.gov.

BUILDING OPENMM USING CONDA MODULE

1. Update environment

```
$ module load conda/2022-07-19
```

2. Install OpenMM

```
$ mkdir conda
$ conda create --prefix /path-to/conda/openmm_env
$ conda activate /path-to/conda/openmm_env
$ conda install -c conda-forge openmm cudatoolkit=11.4
$ conda deactivate /path-to/conda/openmm_env
```

3. Validate installation: if successful, then info on code version, platform types, CUDA initialization, and force error tolerance will be shown.

```
$ cd /path-to/conda/openmm_env/share/openmm/examples
$ python -m openmm.testInstallation
```

4. Benchmark testing using PBS job script below.

```
$ cd /path-to/conda/openmm_env/share/openmm/examples
$ qsub ./submit.sh
```

RUNNING OPENMM BENCHMARK ON POLARIS

A sample pbs script follows that will run OpenMM benchmark on one node.

```
#!/bin/sh
#PBS -l select=1:system=polaris
#PBS -l place=scatter
#PBS -l walltime=0:30:00
#PBS -q debug
#PBS -A PROJECT
#PBS -l filesystems=home:grand:eagle

cd ${PBS_O_WORKDIR}

module load cudatoolkit-standalone/11.4.4

python benchmark.py --platform=CUDA --test=pme --precision=mixed --seconds=30 --heavy-hydrogens > test.output
```

BUILDING OPENMM FROM SOURCE

1. Update environment

```
$ module load cudatoolkit-standalone/11.4.4
$ module load cray-python/3.9.12.1
```

2. Download OpenMM

```
$ git checkout https://github.com/openmm/openmm.git
$ cd openmm ; mkdir build
```

3. Download and build doxygen

```
$ git clone https://github.com/doxygen/doxygen.git
$ cd doxygen ; cmake ; make ; make install ; cd ../
```

4. Download and install swig in OpenMM directory.

```
$ tar xzf swig-4.0.2.tar.gz
$ cd swig-4.0.2
$ ./configure --prefix=/path-to/openmm/swig-4.0.2 ; make -j 8 ; make install
```

5. Build OpenMM

```
$ cmake -DDOXYGEN_EXECUTABLE=/path-to/openmm/doxygen/bin/doxygen \
        -DSWIG_EXECUTABLE=/path-to/openmm/swig-4.0.2/bin/swig \
        -DCMAKE_INSTALL_PREFIX=/path-to/openmm/build \
         -DCUDA_HOME=/soft/compilers/cudatoolkit/cuda-11.4.4 \
         -DCUDA_INCLUDE_DIR=/soft/compilers/cudatoolkit/cuda-11.4.4/include \
         -DCUDA_LIB_DIR=/soft/compilers/cudatoolkit/cuda-11.4.4/lib64
```

```
$ make -j 8
$ make install
```

6. Validate installation: if successful, then info on code version, platform types, CUDA initialization, and force error tolerance will be shown.

```
$ cd /path-to/openmm/examples
$ python -m openmm.testInstallation
```

7. Benchmark testing using the PBS job script above.

```
$ cd /path-to/openmm/examples
$ qsub ./submit.sh
```

VASP

VASP 6.X.X IN POLARIS (NVHPC+OPENACC+OPENMP+CUDA MATH+CRAYMPI)

The Vienna Ab initio Simulation Package (VASP) is a software package for performing electronic structure calculations with periodic boundary conditions. It is most commonly used that to perform density functional theory (DFT) calculations in a planewave basis using the projector augemented wave (PAW) method. A more complete description of VASP can be found here.

Users must have a license to use this code on ALCF systems. More information on how to get access to VASP binaries can be found here.

General compiling/installing instructions provided by VASP support

Instructions and samples of `makefile.include` could be found in the `vasp.at` wiki page.

The follow `makefile.include` was tailored for Polaris, originally taken from here.

```
# Precompiler options
CPP_OPTIONS = -DHOST=\"LinuxNV\" \
              -DMPI -DMPI_BLOCK=8000 -Duse_collective \
              -DscaLAPACK \
              -DCACHE_SIZE=4000 \
              -Davoidalloc \
              -Dvasp6 \
              -Duse_bse_te \
              -Dtbdyn \
              -Dqd_emulate \
              -Dfock_dblbuf \
              -D_OPENMP \
              -D_OPENACC \
              -DUSENCCL -DUSENCCLP2P\

CPP        = nvfortran -Mpreprocess -Mfree -Mextend -E $(CPP_OPTIONS) $*$(FUFFIX)  > $*$(SUFFIX)

FC         = ftn -acc -gpu=cc80 -mp -target-accel=nvidia80
FCL        = ftn -acc -gpu=cc80 -c++libs -target-accel=nvidia80

FREE       = -Mfree

FFLAGS     = -Mbackslash -Mlarge_arrays

OFLAG      = -fast

DEBUG      = -Mfree -O0 -traceback

# Specify your NV HPC-SDK installation, try to set NVROOT automatically
NVROOT     =$(shell which nvfortran | awk -F /compilers/bin/nvfortran '{ print $$1 }')
# ...or set NVROOT manually
NVHPC      ?= /opt/nvidia/hpc_sdk
NVVERSION  = 20.9
#NVROOT     = $(NVHPC)/Linux_x86_64/$(NVVERSION)

# Use NV HPC-SDK provided BLAS and LAPACK libraries
LIBAOCL=/soft/libraries/aocl/3.2.0
BLAS       = ${LIBAOCL}/lib/libblis-mt.a
LAPACK     = ${LIBAOCL}/lib/libflame.a

BLACS      =
SCALAPACK  =
#SCALAPACK  = -Mscalapack
```

```
#SCALAPACK  = ${LIBAOCL}/lib/libscalapack.a

CUDA       = -cudalib=cublas,cusolver,cufft,nccl -cuda

LLIBS      = $(SCALAPACK) $(LAPACK) $(BLAS) $(CUDA)

# Software emulation of quadruple precsion
QD         ?= $(NVROOT)/compilers/extras/qd
LLIBS      += -L$(QD)/lib -lqdmod -lqd
INCS       += -I$(QD)/include/qd

#INCS       += -I/usr/include/linux
#INCS       += -I/usr/include/c++/7/tr1
#INCS       += -I/usr/include/c++/7
#INCS       += -I/usr/include/x86_64-linux-gnu/c++/7
#INCS       += -I/lus/theta-fs0/software/spack/spack-dev/opt/spack/linux-sles15-x86_64/gcc-9.3.0/gcc-10.2.0-r7v3naxd5xgzzaqxoe73jj2ytwuddamr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/include/

# Use the FFTs from fftw
FFTW       ?= ${LIBAOCL}
LLIBS      += -L$(FFTW)/lib -lfftw3 -lfftw3_omp -lomp
#INCS       += -I/soft/libraries/aocl/3.2.0/include_LP64/
INCS       += -I$(FFTW)/include

OBJECTS    = fftmpiw.o fftmpi_map.o fftw3d.o fft3dlib.o

# Redefine the standard list of O1 and O2 objects
SOURCE_O1  := pade_fit.o
SOURCE_O2  := pead.o

# For what used to be vasp.5.lib
CPP_LIB    = $(CPP)
FC_LIB     = nvfortran
CC_LIB     = cc
CFLAGS_LIB = -O $(INCS) -c++libs -cuda
FFLAGS_LIB = -O1 -Mfixed
FREE_LIB   = $(FREE)

OBJECTS_LIB= linpack_double.o getshmem.o

# For the parser library
#CXX_PARS   = nvc++ --no_warnings -I/lus/theta-fs0/software/spack/spack-dev/opt/spack/linux-sles15-x86_64/gcc-9.3.0/gcc-10.2.0-r7v3naxd5xgzzaqxoe73jj2ytwuddamr/include/c++/10.2.0/ -I/lus/
theta-fs0/software/spack/spack-dev/opt/spack/linux-s
les15-x86_64/gcc-9.3.0/gcc-10.2.0-r7v3naxd5xgzzaqxoe73jj2ytwuddamr/include/c++/10.2.0/x86_64-pc-linux-gnu -I/lus/theta-fs0/software/spack/spack-dev/opt/spack/linux-sles15-x86_64/gcc-9.3.0/
gcc-10.2.0-r7v3naxd5xgzzaqxoe73jj2ytwuddamr/lib/gcc
/x86_64-pc-linux-gnu/10.2.0/include -I/lus/theta-fs0/software/spack/spack-dev/opt/spack/linux-sles15-x86_64/gcc-9.3.0/gcc-10.2.0-r7v3naxd5xgzzaqxoe73jj2ytwuddamr/lib/gcc/x86_64-pc-linux-
gnu/10.2.0/include-fixed/
CXX_PARS   = nvc++ --no_warnings

# Normally no need to change this
SRCDIR     = ../../src
BINDIR     = ../../bin
```

Setting up compiler and libraries with `module`

The follow modules will update the include and libraries paths used by the Cray compiler wrapper `ftn` to load additional math libraries for the CPU.

```
module purge
module load PrgEnv-nvhpc
module load cray-libsci
module load craype-accel-nvidia8
```

Compiling VASP

Once the `modules` are loaded and a `makefile.include` is in the `vasp` folder, compiling all the object files and binaries is done with:

```
make -j1
```

Running VASP in Polaris

An example of a submission script could be found here `/soft/applications/vasp/submit-polaris2023-2.sh` , which would looks something similar to:

```
#!/bin/sh
#PBS -l select=1:system=polaris
#PBS -l place=scatter
#PBS -l walltime=0:30:00
#PBS -l filesystems=home:grand:eagle
#PBS -q debug
#PBS -A Catalyst

module load PrgEnv-nvhpc
module load cray-libsci

export MPICH_GPU_SUPPORT_ENABLED=1
NNODES=`wc -l < $PBS_NODEFILE`
NRANKS=2
NDEPTH=4
NTHREADS=4
NGPUS=2
NTOTRANKS=$(( NNODES * NRANKS ))

mpiexec -n ${NTOTRANKS} --ppn ${NRANKS} --depth ${NDEPTH} --cpu-bind depth --env OMP_NUM_THREADS=${NTHREADS} /path_to_vasp/bin/vasp_std
```

Submission scripts should have executable attibutes to be used with `qsub` script mode.

```
chmod +x example-script.sh
qsub  example-script.sh
```

Known issues versions: >= 6.4.0 in Polaris

- Undefined `MPIX_Query_cuda_support` function at linking binary: This function is called in `src/openacc.F` . The `MPIX_Query_cuda_support` is not included in `cray-mpich` . One workaround to this issue is to comment this function call. See the follow suggested changes marked by `!!!!! CHANGE HERE` in the `file:src/openacc.F`

```
+!!!!!CHANGE HERE
-      INTERFACE
-        INTEGER(c_int) FUNCTION MPIX_Query_cuda_support() BIND(C, name="MPIX_Query_cuda_support")
-        END FUNCTION
-      END INTERFACE

       CHARACTER(LEN=1) :: ENVVAR_VALUE
       INTEGER :: ENVVAR_STAT

       ! This should tell us if MPI is CUDA-aware
+!!!!!CHANGE HERE
-       CUDA_AWARE_SUPPORT = MPIX_Query_cuda_support() == 1
+       CUDA_AWARE_SUPPORT = .TRUE.
       ! However, for OpenMPI some env variables can still deactivate it even though the previous
       ! check was positive
       CALL GET_ENVIRONMENT_VARIABLE("OMPI_MCA_mpi_cuda_support", ENVVAR_VALUE, STATUS=ENVVAR_STAT)
       IF (ENVVAR_STAT==0 .AND. ENVVAR_VALUE=='0') CUDA_AWARE_SUPPORT = .FALSE.
       CALL GET_ENVIRONMENT_VARIABLE("OMPI_MCA_opal_cuda_support", ENVVAR_VALUE, STATUS=ENVVAR_STAT)
       IF (ENVVAR_STAT==0 .AND. ENVVAR_VALUE=='0') CUDA_AWARE_SUPPORT = .FALSE.
       ! Just in case we might be non-OpenMPI, and their MPIX_Query_cuda_support behaves similarly
       CALL GET_ENVIRONMENT_VARIABLE("MV2_USE_CUDA", ENVVAR_VALUE, STATUS=ENVVAR_STAT)
       IF (ENVVAR_STAT==0 .AND. ENVVAR_VALUE=='0') CUDA_AWARE_SUPPORT = .FALSE.
       CALL GET_ENVIRONMENT_VARIABLE("MPICH_RDMA_ENABLED_CUDA", ENVVAR_VALUE, STATUS=ENVVAR_STAT)
       IF (ENVVAR_STAT==0 .AND. ENVVAR_VALUE=='0') CUDA_AWARE_SUPPORT = .FALSE.
       CALL GET_ENVIRONMENT_VARIABLE("PMPI_GPU_AWARE", ENVVAR_VALUE, STATUS=ENVVAR_STAT)
       IF (ENVVAR_STAT==0) CUDA_AWARE_SUPPORT =(ENVVAR_VALUE == '1')
```

```
+!!!!!CHANGE HERE
+       CALL GET_ENVIRONMENT_VARIABLE("MPICH_GPU_SUPPORT_ENABLED", ENVVAR_VALUE, STATUS=ENVVAR_STAT)
+       IF (ENVVAR_STAT==0) CUDA_AWARE_SUPPORT =(ENVVAR_VALUE == '1')
```

6.7.2 Libraries

Math Libraries

BLAS, LAPACK, AND SCALAPACK FOR CPUS

Some math libraries targeting CPUs are made available as part of the `nvhpc` modules and are based on the OpenBLAS project. Additional documentation is available from NVIDIA.

- BLAS & LAPACK can be found in the `$NVIDIA_PATH/compilers/lib` directory.
- ScaLAPACK can be found in the `$NVIDIA_PATH/comm_libs` directory.

NVIDIA MATH LIBRARIES FOR GPUS

Math libraries from NVIDIA are made available via the `nvhpc` modules. Many of the libraries users typically use can be found in the `$NVIDIA_PATH/math_libs` directory. Some examples follow and additional documentation is available from NVIDIA.

- libcublas
- libcufft
- libcurand
- libcusolver
- libcusparse

Cabana

CABANA

Cabana is built atop Kokkos. It provides class templates useful for implementing particle codes

Cabana Documentation

– Cabana Wiki

– Cabana github

Cabana on Polaris

Built against the prebuilt Kokkos on polaris, the prebuilt Cabana includes 3 backends: Serial and OpenMP for CPU execution and CUDA for GPU execution. To use it, run

```
module use /soft/modulefiles
module load cabana
```

Currently, Cabana is a headers-only installation; there are no libraries per se.

6.8 Data Science

6.8.1 Julia

Julia is a high-level, high-performance dynamic programming language for technical computing. It has a syntax familiar to users of many other technical computing environments. Designed at MIT to tackle large-scale partial-differential equation simulation and distributed linear algebra, Julia features a robust ecosystem of tools for optimization, statistics, parallel programming, and data visualization. Julia is actively developed by the Julia Labs team at MIT and in industry, along with hundreds of domain-expert scientists and programmers worldwide.

Contributing

This guide is a first draft of the Julia documentation for Polaris. If you have any suggestions or contributions, please open a pull request or contact us by opening a ticket at the ALCF Helpdesk.

Julia Installation

Using the official Julia 1.9 binaries from the Julia webpage is recommended. Juliaup provides a convenient way to install Julia and manage the various Julia versions.

```
curl -fsSL https://install.julialang.org | sh
```

You may then list the available Julia versions with `juliaup list` and install a specific version with `juliaup install <version>`. You can then activate a specific version with `juliaup use <version>` and set the default version with `juliaup default <version>`. `juliaup update` will update the installed Julia versions. In general, the latest stable release of Julia should be used.

```
juliaup add release
```

JULIA PROJECT ENVIRONMENT

The Julia built-in package manager allows you to create a project and enable project-specific dependencies. Julia manages packages in the Julia depot located by default in `~/.julia`. However, that NFS filesystem is not meant for high-speed access. Therefore, this Julia depot folder should be located on a fast filesystem of your choice (grand, eagle). The Julia depot directory is set via the environment variable `JULIA_DEPOT_PATH`. For example, you can set the Julia depot to a directory on Polaris grand filesystem by adding the following line to your `~/.bashrc` file:

```
export /lus/grand/projects/$PROJECT/$USER/julia_depot
```

Programming Julia on Polaris

There are three key components to using Julia for large-scale computations:

1. MPI support through MPI.jl
2. GPU support through CUDA.jl
3. HDF5 support through HDF5.jl

In addition, we recommend VSCode with the Julia extension for a modern IDE experience, together with the ssh-remote extension for remote interactive development.

MPI SUPPORT

MPI support is provided through the MPI.jl.

```
julia> ] add MPI
```

This will install the MPI.jl package and default MPI prebuilt binaries provided by an artifact. For on-node debugging purposes the default artifact is sufficient. However, for large-scale computations, it is to use the system MPI library that is loaded via `module`. As of MPI.jl v0.20 this is handled through MPIPrefences.jl.

```
julia --project -e 'using MPIPreferences; MPIPreferences.use_system_binary()'
```

Check that the correct MPI library is targeted with Julia.

```
julia --project -e 'using MPI; MPI.versioninfo()'
MPIPreferences:
  binary:  system
  abi:     MPICH
  libmpi:  libmpi_cray
  mpiexec: mpiexec

Package versions
  MPI.jl:             0.20.11
  MPIPreferences.jl:  0.1.8

Library information:
  libmpi:  libmpi_cray
  MPI version:  3.1.0
  Library version:
    MPI VERSION    : CRAY MPICH version 8.1.16.5 (ANL base 3.4a2)
    MPI BUILD INFO : Mon Apr 18 12:05 2022 (git hash 4f56723)
```

When running on the login node, switch back to the default provided MPI binaries in `MPI_jll.jl` by removing the `LocalPreferences.toml` file.

GPU SUPPORT

NVIDIA GPU support is provided through the CUDA.jl package.

```
julia> ] add CUDA
```

In case you want write portable GPU kernels we highly recommend the KernelAbstractions.jl package. It provides a high-level abstraction for writing GPU kernels that can be compiled for different GPU backends.

```
julia> ] add KernelAbstractions
```

By loading either oneAPI.jl, AMDGPU.jl, or CUDA.jl (see quickstart guide below).

HDF5 SUPPORT

Parallel HDF5 support is provided by

```
module load cray-hdf5-parallel
```

After setting `export JULIA_HDF5_PATH=$HDF5_DIR` we can install the HDF5.jl package.

```
julia> ] add HDF5
```

Quickstart Guide

The following example shows how to use MPI.jl, CUDA.jl, and HDF5.jl to write a parallel program that computes the sum of two vectors on the GPU and writes the result to an HDF5 file. A repository with an example code computing an approximation of pi can be found at Polaris.jl. In this repository, you will also find a `setup_polaris.sh` script

that will build the HDF5.jl and MPI.jl package for the system libraries. The dependencies are installed with the following commands:

```
julia --project
```

```
julia> ] up
```

```
using CUDA
using HDF5
using MPI
using Printf
using Random

function pi_kernel(x, y, d, n)
    idx = (blockIdx().x-1) * blockDim().x + threadIdx().x
    if idx <= n
        d[idx] = (x[idx] - 0.5)^2 + (y[idx] - 0.5)^2 <= 0.25 ? 1 : 0
    end
    return nothing
end

function approximate_pi_gpu(n::Integer)
    x = CUDA.rand(Float64, n)
    y = CUDA.rand(Float64, n)
    d = CUDA.zeros(Float64, n)

    nblocks = ceil(Int64, n/32)

    @cuda threads=32 blocks=nblocks pi_kernel(x,y,d,n)

    return sum(d)
end

function main()
    n = 100000  # Number of points to generate per rank
    Random.seed!(1234)  # Set a fixed random seed for reproducibility

    dsum = MPI.Allreduce(approximate_pi_gpu(n), MPI.SUM, MPI.COMM_WORLD)

    pi_approx = (4 * dsum) / (n * MPI.Comm_size(MPI.COMM_WORLD))

    if MPI.Comm_rank(MPI.COMM_WORLD) == 0
        @printf "Approximation of π using Monte Carlo method: %.10f\n" pi_approx
        @printf "Error: %.10f\n" abs(pi_approx - π)
    end
    return pi_approx
end

MPI.Init()
if !isinteractive()
    pi_approx = main()
    h5open("pi.h5", "w") do file
        write(file, "pi", pi_approx)
    end
end
```

JOB SUBMISSION SCRIPT

This example can be run on Polaris with the following job submission script:

```
#!/bin/bash -l
#PBS -l select=1:system=polaris
#PBS -l place=scatter
#PBS -l walltime=0:30:00
#PBS -l filesystems=home:grand
#PBS -q debug
#PBS -A PROJECT

cd ${PBS_O_WORKDIR}

# MPI example w/ 4 MPI ranks per node spread evenly across cores
NNODES=`wc -l < $PBS_NODEFILE`
NRANKS_PER_NODE=4
NDEPTH=8
NTHREADS=1
module load cray-hdf5-parallel
# Put in your Julia depot path
export JULIA_DEPOT_PATH=MY_JULIA_DEPOT_PATH
# Path to Julia executable. When using juliaup, it's in your julia_depot folder
```

```
JULIA_PATH=$JULIA_DEPOT_PATH/juliaup/julia-1.9.1+0.x64.linux.gnu/bin/julia
NTOTRANKS=$(( NNODES * NRANKS_PER_NODE ))
echo "NUM_OF_NODES= ${NNODES} TOTAL_NUM_RANKS= ${NTOTRANKS} RANKS_PER_NODE=${NRANKS_PER_NODE} THREADS_PER_RANK= ${NTHREADS}"

mpiexec -n ${NTOTRANKS} --ppn ${NRANKS_PER_NODE} --depth=${NDEPTH} --cpu-bind depth julia --check-bounds=no --project pi.jl
```

Verify that `JULIA_DEPOT_PATH` is set to the correct path and `JULIA_PATH` points to the Julia executable. When using `juliaup`, the Julia executable is located in the `juliaup` folder of your `JULIA_DEPOT_PATH`.

Advanced features

CUDA-AWARE MPI

MPI.jl supports CUDA-aware MPI. This is enabled by setting the following environment variables

```
export JULIA_CUDA_MEMORY_POOL=none
export MPICH_GPU_SUPPORT_ENABLED=1
export JULIA_MPI_PATH=$PATH_TO_CUDA_MPI # /opt/cray/pe/mpich/8.1.16/ofi/nvidia/20.7
export JULIA_MPI_HAS_CUDA=1
```

Note that `MPI.jl` needs to be rebuilt for the changes to take effect.

```
julia --project -e 'using Pkg; Pkg.build("MPI"; verbose=true)'
```

LARGE-SCALE PARALLELISM

`CUDA.jl` uses the `nvcc` compiler to compile GPU kernels. This will create object files in the `TEMP` filesystem. Per default, the `tempdir` is a global directory that can lead to name clashes of the compiled kernel object files. To avoid this, we recommend setting the `tempdir` to a local directory on the compute node.

```
export TMPDIR=/local/scratch
```

6.8.2 Python

Conda

We provide prebuilt `conda` environments containing GPU-supported builds of `torch`, `tensorflow` (both with `horovod` support for multi-node calculations), `jax`, and many other commonly-used Python modules.

Users can activate this environment by first loading the `conda` module, and then activating the base environment.

Explicitly (either from an interactive job, or inside a job script):

```
$ module load conda
$ conda activate base
(base) $ which python3
/soft/datascience/conda/2022-09-08/mconda3/bin/python3
```

In one line, `module load conda; conda activate`. This can be performed on a compute node, as well as a login node.

As of writing, the latest `conda` module on Polaris is built on Miniconda3 version 4.14.0 and contains Python 3.8.13. Future modules may contain entirely different major versions of Python, PyTorch, TensorFlow, etc.; however, the existing modules will be maintained as-is as long as feasible.

While the shared Anaconda environment encapsulated in the module contains many of the most commonly used Python libraries for our users, you may still encounter a scenario in which you need to extend the functionality of the environment (i.e. install additional packages)

There are two different approaches that are currently recommended.

VIRTUAL ENVIRONMENTS VIA `VENV`

Creating your own (empty) virtual Python environment in a directory that is writable to you is simple:

```
python3 -m venv /path/to/new/virtual/environment
```

This creates a new folder that is fairly lightweight folder (<20 MB) with its own Python interpreter where you can install whatever packages you'd like. First, you must activate the virtual environment to make this Python interpreter the default interpreter in your shell session.

You activate the new environment whenever you want to start using it via running the activate script in that folder:

```
/path/to/new/virtual/environment/bin/activate
```

In many cases, you do not want an empty virtual environment, but instead want to start from the `conda` base environment's installed packages, only adding and/or changing a few modules.

To extend the base Anaconda environment with `venv` (e.g. `my_env` in the current directory) and inherit the base enviroment packages, one can use the `--system-site-packages` flag:

```
module load conda; conda activate
python -m venv --system-site-packages my_env
source my_env/bin/activate
# Install additional packages here...
```

You can always retroactively change the `--system-site-packages` flag state for this virtual environment by editing `my_env/pyvenv.cfg` and changing the value of the line `include-system-site-packages = false`.

To install a different version of a package that is already installed in the base environment, you can use:

```
pip install --ignore-installed  ... # or -I
```

The shared base environment is not writable, so it is impossible to remove or uninstall packages from it. The packages installed with the above `pip` command should shadow those installed in the base environment.

CLONING THE BASE ANACONDA ENVIRONMENT

If you need more flexibility, you can clone the conda environment into a custom path, which would then allow for root-like installations via `conda install <module>` or `pip install <module>`. Unlike the `venv` approach, using a cloned Anaconda environment requires you to copy the entirety of the base environment, which can use significant storage space.

This can be performed by:

```
$ module load conda
$ conda activate base
(base) $ conda create --clone base --prefix /path/to/envs/base-clone
(base) $ conda activate /path/to/envs/base-clone
(base-clone) $ which python3
/path/to/base-clone/bin/python3
```

The cloning process can be quite slow.

> **Warning**
>
> In the above commands, `path/to/envs/base-clone` should be replaced by a suitably chosen path.

USING `PIP INSTALL --USER` (NOT RECOMMENDED)

With the conda environment setup, one can install common Python modules using `pip install --users <module-name>` which will install packages in `$PYTHONUSERBASE/lib/pythonX.Y/site-packages`. The `$PYTHONUSERBASE` environment variable is automatically set when you load the base conda module, and is equal to `/home/$USER/.local/polaris/conda/YYYY-MM-DD`.

Note, Python modules installed this way that contain command line binaries will not have those binaries automatically added to the shell's `$PATH`. To manually add the path:

```
export PATH=$PYTHONUSERBASE/bin:$PATH
```

Be sure to remove this location from `$PATH` if you deactivate the base Anaconda environment or unload the module.

Cloning the Anaconda environment, or using `venv` are both more flexible and transparent when compared to `--user` installs.

6.8.3 Containers on Polaris

Since Polaris is using NVIDIA A100 GPUs, there can be portability advantages with other NVIDIA-based systems if your workloads use containers. In this document, we'll outline some information about containers on Polaris including how to build custom containers, how to run containers at scale, and common gotchas.

Container creation can be achieved one of two ways either by using Docker on your local machine as mentioned in Docker section of Theta(KNL) and publishing it to DockerHub, or by using a Singularity recipe file and building on a Polaris worker node. If you are not interested in building a container and only want to use the available containers, you can read the section on available containers.

Singularity

The container system on Polaris is `singularity`. You can set up singularity with a module (this is different than, for example, ThetaGPU!):

```
# To see what versions of singularity are available:
module avail singularity

# To load the Default version:
module load singularity

# To load a specific version:
module load singularity/3.8.7 # the default at the time of writing these docs.
```

WHICH SINGULARITY?

There used to be a single `singularity` tool, which in 2021 split after some turmoil. There are now two `singularity`s: one developed by Sylabs, and the other as part of the Linux Foundation. Both are open source, and the split happened around version 3.10. The version on Polaris is from Sylabs but for completeness, here is the Linux Foundation's version. Note that the Linux Foundation version is renamed to `apptainer` - different name, roughly the same thing though divergence may happen after 2021's split.

Build from Docker Images or Argonne Github container registry

Docker containers require root privileges, which users do not have on Polaris. That doesn't mean all your docker containers aren't useful, though. If you have an existing docker container, you can convert it to singularity pretty easily on the login node. To build the latest NVIDIA container for PyTorch you can run the following:

```
module load singularity
singularity build pytorch:22.06-py3.sing docker://nvcr.io/nvidia/pytorch:22.06-py3
```

Note that `latest` here mean when these docs were written, summer 2022. It may be useful to get a newer container if you need the latest features. You can find the PyTorch container site here. The tensorflow containers are here (though note that LCF doesn't prebuild the TF-1 containers typically). You can search the full container registry here.

You can also use our custom built containers using Github OCI container registry. Here's a list of containers distributed by ALCF staff tailored for Polaris.

```
module load singularity
singularity pull IMAGE_NAME oras://ghcr.io/argonne-lcf/IMAGE_NAME:latest
```

Build with a Recipe

You can also build a singularity container using a recipe file. Detailed instructions for recipe construction are available on the Singularity Recipe Page. You can also check our singularity recipe example for building a mpich version 4 container on Polaris.

Once you have a recipe file, you can build it on Polaris, but only on compute nodes. You can launch an interactive job using the attribute `singularity_fakeroot=true` to build on a compute node.

```
qsub -I -A <project_name> -q <queue> -l select=1 -l walltime=60:00 -l singularity_fakeroot=true -l filesystems=home:eagle:grand
```

You need to replace the `<project_name>` with the appropriate project to charge and `<queue>` with `debug`, or `preemptable` queues since we only request a single node.

After your interactive job has started, you need to load the `singularity` module on the compute node and export the proxy variables for internet access. Then you can build the container as shown below.

```
module load singularity
export HTTP_PROXY=http://proxy.alcf.anl.gov:3128
export HTTPS_PROXY=http://proxy.alcf.anl.gov:3128
export http_proxy=http://proxy.alcf.anl.gov:3128
export https_proxy=http://proxy.alcf.anl.gov:3128
singularity build --fakeroot <image_name>.sif <def_filename>.def
```

Alternatively, you can just pull the mpich 4 image distributed by us and build on top of it

```
singularity pull oras://ghcr.io/argonne-lcf/mpich-4:latest
```

Running Singularity container on Polaris

EXAMPLE SUBMISSION SCRIPT ON POLARIS

To run a container on Polaris you can use the submission script described here. Below we have described the submission script for your understanding.

First we define our job and our script takes the container name as an input parameter.

```
#!/bin/sh
#PBS -l select=2:system=polaris
#PBS -q debug
#PBS -l place=scatter
#PBS -l walltime=0:30:00
#PBS -l filesystems=home:grand
#PBS -A <project_name>
cd ${PBS_O_WORKDIR}
echo $CONTAINER
```

We move to current working directory and enable network access at run time by setting the proxy. We also load singularity.

```
# SET proxy for internet access
module load singularity
export HTTP_PROXY=http://proxy.alcf.anl.gov:3128
export HTTPS_PROXY=http://proxy.alcf.anl.gov:3128
export http_proxy=http://proxy.alcf.anl.gov:3128
export https_proxy=http://proxy.alcf.anl.gov:3128
```

This is important for system (Polaris - Cray) mpich to bind to containers mpich. Set the following environment variables

```
ADDITIONAL_PATH=/opt/cray/pe/pals/1.1.7/lib/
module load cray-mpich-abi
export SINGULARITYENV_LD_LIBRARY_PATH="$CRAY_LD_LIBRARY_PATH:$LD_LIBRARY_PATH:$ADDITIONAL_PATH"
```

Set the number of ranks per node spread as per your scaling requirements

```
# MPI example w/ 16 MPI ranks per node spread evenly across cores
NODES=`wc -l < $PBS_NODEFILE`
PPN=16
PROCS=$((NODES * PPN))
echo "NUM_OF_NODES= ${NODES} TOTAL_NUM_RANKS= ${PROCS} RANKS_PER_NODE= ${PPN}"
```

Finally launch your script

```
echo C++ MPI
mpiexec -hostfile $PBS_NODEFILE -n $PROCS -ppn $PPN singularity exec -B /opt -B /var/run/palsd/ $CONTAINER /usr/source/mpi_hello_world

echo Python MPI
mpiexec -hostfile $PBS_NODEFILE -n $PROCS -ppn $PPN singularity exec -B /opt -B /var/run/palsd/ $CONTAINER python3 /usr/source/mpi_hello_world.py
```

The job can be submitted using:

```
qsub -v CONTAINER=mpich-4_latest.sif job_submission.sh
```

Available containers

If you just want to know what containers are available, here you go.

- For running mpich/MPI containers on Polaris, it can be found here
- For running databases on Polaris. It can be found here
- For using shpc - that allows for running containers as modules. It can be found here
- Some containers are found in /soft/containers

The latest containers are updated periodically. If you have trouble using containers, or request a newer or a different container please contact ALCF support at `support@alcf.anl.gov`.

Troubleshooting

1. **Permission Denied Error**: One may get a `permission denied` error during the build process, due to a nasty permission setting, quota

limitations, or simply due to an unresolved symbolic link. You can try one of the solutions below:

- Check your quota and delete any unnecessary files.
- Clean-up singularity cache, `~/.singularity/cache`, and set the singularity tmp and cache directories as below:

```
export SINGULARITY_TMPDIR=/tmp/singularity-tmpdir
  mkdir $SINGULARITY_TMPDIR
  export SINGULARITY_CACHEDIR=/tmp/singularity-cachedir/
  mkdir $SINGULARITY_CACHEDIR
```

- Make sure you are not on a directory accessed with a symlink, i.e. check if `pwd` and `pwd -P` returns the same path.
- If any of the above doesn't work, try running the build in your home directory.

2. **Mapping to rank 0 on all nodes**: This is mainly due to container mpich not binding to system mpich. It is imperative for the container to have mpich which can bind dynamically to system mpich at runtime. Ensure your submission script has the following variables and modules loaded (see below). If this does not resolve, ensure the containers mpich is built with the '--disable-wrapper-rpath' flag. Please refer to this link to find examples of building a mpich based container from scratch and running on Polaris.

```
ADDITIONAL_PATH=/opt/cray/pe/pals/1.1.7/lib/
module load cray-mpich-abi
export SINGULARITYENV_LD_LIBRARY_PATH="$CRAY_LD_LIBRARY_PATH:$LD_LIBRARY_PATH:$ADDITIONAL_PATH"
singularity exec -B /opt -B /var/run/palsd/
```

1. **libmpi.so.40 not found**: This may be due to mpich binding to the wrong system mpich. Try removing .conda & .cache & .local folders from your home directory. Also rebuild your container and try again.
2. Containers built with openmpi may not work correctly. Please ensure your container is built with mpich and the base image is of Debian architecture (For e.g. Ubuntu) image.

6.8.4 Frameworks

TensorFlow on Polaris

TensorFlow is a popular, open-source deep learning framework developed and released by Google. The TensorFlow home page has more information about TensorFlow, which you can refer to. For trouble shooting on Polaris, please contact support@alcf.anl.gov.

INSTALLATION ON POLARIS

TensorFlow is already pre-installed on Polaris, available in the `conda` module. To use it from a compute node, please do:

```
module load conda
conda activate
```

Then, you can load TensorFlow in `python` as usual (below showing results from the `conda/2022-07-19` module):

```
>>> import tensorflow as tf
>>> tf.__version__
'2.9.1'
>>>
```

This installation of TensorFlow was built from source and the CUDA libraries it uses are found via the `CUDA_HOME` environment variable (below showing results from the `conda/2022-07-19` module):

```
$ echo $CUDA_HOME
/soft/datascience/cuda/cuda_11.5.2_495.29.05_linux
```

If you need to build applications that use this version of TensorFlow and CUDA, we recommend using these cuda libraries to ensure compatibility. We periodically update the TensorFlow release, though updates will come in the form of new versions of the `conda` module.

TensorFlow is also available through NVIDIA containers that have been translated to Singularity containers. For more information about containers, please see the Containers documentation page.

TENSORFLOW BEST PRACTICES ON POLARIS

Single Node Performance

When running TensorFlow applications, we have found the following practices to be generally, if not universally, useful and encourage you to try some of these techniques to boost performance of your own applications.

1. Use Reduced Precision. Reduced Precision is available on A100 via tensorcores and is supported with TensorFlow operations. In general, the way to do this is via the `tf.keras.mixed_precision` Policy, as descibed in the mixed precision documentation. If you use a custom training loop (and not `keras.Model.fit`), you will also need to apply loss scaling.
2. Use TensorFlow's graph API to improve efficiency of operations. TensorFlow is, in general, an imperative language but with function decorators like `@tf.function` you can trace functions in your code. Tracing replaces your python function with a lower-level, semi-compiled TensorFlow Graph. More information about the `tf.function` interface is available here. When possible, use jit_compile, but be aware of sharp bits when using `tf.function`: python expressions that aren't tensors are often replaced as constants in the graph, which may or may not be your intention.
3. Use XLA compilation on your code. XLA is the Accelerated Linear Algebra library that is available in tensorFlow and critical in software like JAX. XLA will compile a `tf.Graph` object, generated with `tf.function` or similar, and perform optimizations like operation-fusion. XLA can give impressive performance boosts with almost no user changes except to set an environment variable `TF_XLA_FLAGS=--tf_xla_auto_jit=2`. If your code is complex, or has dynamically sized tensors (tensors where the shape changes every iteration), XLA can be detrimental: the overhead for compiling functions can be large enough to mitigate performance improvements. XLA is particularly powerful when combined with reduced precision, yielding speedups > 100% in some models.

MULTI-GPU / MULTI-NODE SCALE UP

TensorFlow is compatible with scaling up to multiple GPUs per node, and across multiple nodes. Good scaling performance has been seen up to the entire Polaris system, > 2048 GPUs. Good performance with tensorFlow has been seen with horovod in particular. For details, please see the Horovod documentation. Some polaris specific details that may be helpful to you:

1. CPU affinity and NCCL settings can improve scaling performance, particularly at the largest scales. In particular, we encourage users to try their scaling measurements with the following settings:
2. Set the environment variable `NCCL_COLLNET_ENABLE=1`
3. Set the environment varialbe `NCCL_NET_GDR_LEVEL=PHB`
4. Manually set the CPU affinity via mpiexec, such as with `--cpu-bind verbose,list:0,8,16,24`
5. Horovod works best when you limit the visible devices to only one GPU. Note that if you import `mpi4py` or `horovod`, and then do something like `os.environ["CUDA_VISIBLE_DEVICES"] = hvd.local_rank()`, it may not actually work! You must set the `CUDA_VISIBLE_DEVICES` environment variable prior to doing `MPI.COMM_WORLD.init()`, which is done in `horovod.init()` as well as implicitly in `from mpi4py import MPI`. On Polaris specifically, you can use the environment variable `PMI_LOCAL_RANK` (as well as `PMI_LOCAL_SIZE`) to learn information about the node-local MPI ranks.

TensorFlow Dataloaders

Additional information to be provided.

PyTorch on Polaris

PyTorch is a popular, open source deep learning framework developed and released by Facebook. The PyTorch home page has more information about PyTorch, which you can refer to. For trouble shooting on Polaris, please contact support@alcf.anl.gov.

INSTALLATION ON POLARIS

PyTorch is installed on Polaris already, available in the `conda` module. To use it from a compute node, please do:

```
module load conda
conda activate
```

Then, you can load PyTorch in `python` as usual (below showing results from the `conda/2022-07-19` module):

```
>>> import torch
>>> torch.__version__
'1.12.0a0+git67ece03'
>>>
```

This installation of PyTorch was built from source and the cuda libraries it uses are found via the `CUDA_HOME` environment variable (below showing results from the `conda/2022-07-19` module):

```
$ echo $CUDA_HOME
/soft/datascience/cuda/cuda_11.5.2_495.29.05_linux
```

If you need to build applications that use this version of PyTorch and CUDA, we recommend using these cuda libraries to ensure compatibility. We periodically update the PyTorch release, though updates will come in the form of new versions of the `conda` module.

PyTorch is also available through nvidia containers that have been translated to Singularity containers. For more information about containers, please see the containers documentation page.

PYTORCH BEST PRACTICES ON POLARIS

Single Node Performance

When running PyTorch applications, we have found the following practices to be generally, if not universally, useful and encourage you to try some of these techniques to boost performance of your own applications.

1. Use Reduced Precision. Reduced Precision is available on A100 via tensorcores and is supported with PyTorch operations. In general, the way to do this is via the PyTorch Automatic Mixed Precision package (AMP), as descibed in the mixed precision documentation. In PyTorch, users generally need to manage casting and loss scaling manually, though context managers and function decorators can provide easy tools to do this.
2. PyTorch has a `JIT` module as well as backends to support op fusion, similar to TensorFlow's `tf.function` tools. However, PyTorch JIT capabilities are newer and may not yield performance improvements. Please see TorchScript for more information.

MULTI-GPU / MULTI-NODE SCALE UP

PyTorch is compatible with scaling up to multiple GPUs per node, and across multiple nodes. Good scaling performance has been seen up to the entire Polaris system, > 2048 GPUs. Good performance with PyTorch has been seen with both DDP and Horovod. For details, please see the Horovod documentation or the Distributed Data

Parallel documentation. Some Polaris-specific details that may be helpful to you:

1. CPU affinity and NCCL settings can improve scaling performance, particularly at the largest scales. In particular, we encourage users to try their scaling measurements with the following settings:
2. Set the environment variable `NCCL_COLLNET_ENABLE=1`
3. Set the environment varialbe `NCCL_NET_GDR_LEVEL=PHB`
4. Manually set the CPU affinity via mpiexec, such as with `--cpu-bind verbose,list:0,8,16,24`
5. Horovod and DDP work best when you limit the visible devices to only one GPU. Note that if you import `mpi4py` or `horovod`, and then do something like `os.environ["CUDA_VISIBLE_DEVICES"] = hvd.local_rank()`, it may not actually work! You must set the `CUDA_VISIBLE_DEVICES` environment variable prior to doing `MPI.COMM_WORLD.init()`, which is done in `horovod.init()` as well as implicitly in `from mpi4py import MPI`. On Polaris specifically, you can use the environment variable `PMI_LOCAL_RANK` (as well as `PMI_LOCAL_SIZE`) to learn information about the node-local MPI ranks.

DeepSpeed

DeepSpeed is also available and usable on Polaris. For more information, please see the DeepSpeed documentation directly.

PYTORCH DATALOADER AND MULTI-NODE HOROVOD

Please note there is a bug that causes a hang when using PyTorch's multithreaded data loaders with distributed training across multiple nodes. To workaround this, NVIDIA recommends setting `num_workers=0` in the dataloader configuration, which serializes data loading.

For more details, see Polaris Known Issues.

JAX

JAX is another popular python package for accelerated computing. JAX is built on XLA (the same XLA TensorFlow uses) as well as AutoGrad, and additionally has acceleration tools that operate on functions such as `vmap`, `jit`, etc. JAX is not as widespread in machine learning as TensorFlow and PyTorch for traditional models (Computer Vision, Language Models) though it is quickly gaining promienence. JAX is very powerful when a program needs non-traditional autodifferentiation or vectorizatoin, such as: forward-mode AD, higher order derivatives, Jacobians, Hessians, or any combination of the above. Users of JAX on Polaris are encouraged to read the user documentation in detail, particularly the details about pure-functional programming, no in-place operations, and the common mistakes in writing functions for the `@jit` decorator.

JAX ON POLARIS

JAX is installed on Polaris via the `conda` module, available with:

```
module load conda; conda activate
```

Then, you can load JAX in `python` as usual (below showing results from the `conda/2022-07-19` module):

```
>>> import jax
>>> jax.__version__
'0.3.15'
>>>
```

NOTES ON JAX 0.3.15

On Polaris, due to a bug, an environment variable must be set to use JAX on GPUs. The following code will crash:

```
import jax.numpy as numpy
a = numpy.zeros(1000)
```

outputting an error that looks like:

```
jaxlib.xla_extension.XlaRuntimeError: UNKNOWN: no kernel image is available for execution on the device
```

You can fix this by setting an environment variable:

```
export XLA_FLAGS="--xla_gpu_force_compilation_parallelism=1"
```

SCALING JAX TO MULTIPLE GPUS AND MULTIPLE NODES

Jax has intrinsic scaling tools to use multiple GPUs on a single node, via the `pmap` function. If this is sufficient for your needs, excellent. If not, another alternative is to use the newer package mpi4jax.

mpi4Jax is a relatively new project and requires setting some environment variables for good performance and usability: - Set `MPI4JAX_USE_CUDA_MPI=1` to use CUDA-Aware MPI, supported in the `conda` module, to do operations directly from the GPU. - Set `MPICH_GPU_SUPPORT_ENABLED=1` to use CUDA-Aware MPI.

The following code, based off of a test script from the mpi4jax repository, can help you verify you are using mpi4jax properly:

```
import os
from mpi4py import MPI
import jax
import jax.numpy as jnp
import mpi4jax

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
local_rank = int(os.environ["PMI_LOCAL_RANK"])

available_devices = jax.devices("gpu")
if len(available_devices) <= local_rank:
    raise Exception("Could not find enough GPUs")

target_device = available_devices[local_rank]


@jax.jit
def foo(arr):
   arr = arr + rank
   arr_sum, _ = mpi4jax.allreduce(arr, op=MPI.SUM, comm=comm)
   return arr_sum

with jax.default_device(target_device):
    a = jnp.zeros((3, 3))
    print(f"Rank {rank}, local rank {local_rank}, a.device is {a.device()}")
    result = foo(a)
    print(f"Rank {rank}, local rank {local_rank}, result.device is {result.device()}")

    import time
    print("Sleeping for 5 seconds if you want to look at nvidia-smi ... ")
    import time
    time.sleep(5)
    print("Done sleeping")

if rank == 0:
   print(result)
```

JAX and mpi4jax are both still somewhat early in their software lifecycles. Updates are frequent, and if you require assistance please contact support@alcf.anl.gov.

DeepSpeed

The base `conda` environment on Polaris comes with Microsoft's DeepSpeed pre-installed. Instructions for using / cloning the base environment can be found here.

A batch submission script for the following example is available here.

We describe below the steps needed to get started with DeepSpeed on Polaris.

We focus on the `cifar` example provided in the DeepSpeedExamples repository, though this approach should be generally applicable for running any model with DeepSpeed support.

RUNNING DEEPSPEED ON POLARIS

> **Note**
>
> The instructions below should be **ran directly from a compute node**.
>
> Explicitly, to request an interactive job (from `polaris-login`):
>
> ```
> qsub -A <project> -q debug-scaling -l select=2 -l walltime=01:00:00 -I
> ```
>
> Refer to job scheduling and execution for additional information.

1. Load `conda` module and activate base environment:

   ```
   module load conda ; conda activate base
   ```

2. Clone microsoft/DeepSpeedExamples and navigate into the directory:

   ```
   git clone https://github.com/microsoft/DeepSpeedExamples.git
   cd DeepSpeedExamples/cifar
   ```

Launching DeepSpeed

Launching with DeepSpeed Launching with MPICH

1. Create a DeepSpeed compliant `hostfile`, specifying the `hostname` and number of GPUs (`slots`) for each of our available workers:

```
cat $PBS_NODEFILE > hostfile
sed -e 's/$/ slots=4/' -i hostfile
```

2. Create a `.deepspeed_env` containing the environment variables our workers will need access to:

```
echo "PATH=${PATH}" >> .deepspeed_env
echo "LD_LIBRARY_PATH=${LD_LIBRARY_PATH}" >> .deepspeed_env
echo "http_proxy=${http_proxy}" >> .deepspeed_env
echo "https_proxy=${https_proxy}" >> .deepspeed_env
```

> **Warning**
>
> The `.deepspeed_env` file expects each line to be of the form `KEY=VALUE`. Each of these will then be set as environment variables on each available worker specified in our `hostfile`.

We can then run the `cifar10_deepspeed.py` module using DeepSpeed:

```
deepspeed --hostfile=hostfile cifar10_deepspeed.py \
    --deepspeed \
    --deepspeed_config ds_config.json
```

1. Get total number of available GPUs:

a. Count number of lines in `$PBS_NODEFILE` (1 host per line)

b. Count number of GPUs available on current host

c. `NGPUS="$((${NHOSTS}*${NGPU_PER_HOST}))"`

```
NHOSTS=$(wc -l < "${PBS_NODEFILE}")
NGPU_PER_HOST=$(nvidia-smi -L | wc -l)
NGPUS="$((${NHOSTS}*${NGPU_PER_HOST}))"
```

2. Launch with `mpiexec`:

```
mpiexec \
  --verbose \
  --envall \
  -n "${NGPUS}" \
```

AssertionError: Micro batch sizer per gpu: 0 has to be greater than 0

Depending on the details of your specific job, it may be necessary to modify the provided `ds_config.json`.

If you encounter an error:

```
x3202c0s31b0n0: AssertionError: Micro batch size per gpu: 0 has to be greater than 0
```

you can modify the `"train_batch_size": 16` variable in the provided `ds_config.json` to the (total) number of available GPUs, and explicitly set `"gradient_accumulation_steps": 1`, as shown below.

```
$ export NHOSTS=$(wc -l < "${PBS_NODEFILE}")
$ export NGPU_PER_HOST=$(nvidia-smi -L | wc -l)
$ export NGPUS="$((${NHOSTS}*${NGPU_PER_HOST}))"
$ echo $NHOSTS $NGPU_PER_HOST $NGPUS
24 4 96
$ # replace "train_batch_size" with $NGPUS in ds_config.json
$ # and write to `ds_config-polaris.json`
$ sed \
    "s/$(cat ds_config.json| grep batch | cut -d ':' -f 2)/ ${NGPUS},/" \
    ds_config.json \
    > ds_config-polaris.json
$ cat ds_config-polaris.json
{
    "train_batch_size": 96,
    "gradient_accumulation_steps": 1,
    ...
}
```

6.8.5 Applications

Instructions for `gpt-neox` :

We include below a set of instructions to get `EleutherAI/gpt-neox` running on Polaris.

A batch submission script for the following example is available here.

Warning

The instructions below should be **ran directly from a compute node**.

Explicitly, to request an interactive job (from `polaris-login`):

```
$ qsub -A <project> -q debug-scaling -l select=2 -l walltime=01:00:00
```

Refer to job scheduling and execution for additional information.

1. Load and activate the base `conda` environment:

```
module load conda
conda activate base
```

2. We've installed the requirements for running `gpt-neox` into a virtual environment. To activate this environment,

```
source /soft/datascience/venvs/polaris/2022-09-08/bin/activate
```

3. Clone the `EleutherAI/gpt-neox` repository if it doesn't already exist:

```
git clone https://github.com/EleutherAI/gpt-neox
```

4. Navigate into the `gpt-neox` directory:

```
cd gpt-neox
```

> **Note**
>
> The remaining instructions assume you're inside the `gpt-neox` directory

5. Create a DeepSpeed compliant `hostfile` (each line is formatted as `hostname, slots=N`):

```
cat $PBS_NODEFILE > hostfile
sed -e 's/$/ slots=4/' -i hostfile
export DLTS_HOSTFILE=hostfile
```

6. Create a `.deepspeed_env` file to ensure a consistent environment across all workers

```
echo "PATH=${PATH} > .deepspeed_env"
echo "LD_LIBRARY_PATH=${LD_LIBRARY_PATH} >> .deepspeed_env"
echo "http_proxy=${http_proxy} >> .deepspeed_env"
echo "https_proxy=${https_proxy} >> .deepspeed_env"
```

7. Prepare data:

```
python3 prepare_data.py -d ./data
```

8. Train:

```
python3 ./deepy.py train.py -d configs small.yml local_setup.yml
```

Danger

If your training seems to be getting stuck at

```
Using /home/user/.cache/torch_extensions as PyTorch extensions root...
```

there may be a leftover `.lock` file from an aborted build. Cleaning either the whole `.cache` or the extensions' sub-directory should force a clean build on the next attempt.

6.9 Programming Models

6.9.1 OpenMP

Overview

The OpenMP API is an open standard for parallel programming. The specification document can be found here: https://www.openmp.org. The specification describes directives, runtime routines, and environment variables that allow an application developer to express parallelism (e.g. shared memory multiprocessing and device offloading). Many compiler vendors provide implementations of the OpenMP specification (https://www.openmp.org/specifications).

Setting the environment to use OpenMP on Polaris

Many of the programming environments available on Polaris have OpenMP support.

module	OpenMP CPU support?	OpenMP GPU support?
PrgEnv-nvhpc	yes	yes
llvm	yes	yes
PrgEnv-gnu	yes	no
PrgEnv-cray	yes	yes*

*Currently PrgEnv-cray is not recommended for OpenMP offload.

By default, the PrgEnv-nvhpc module is loaded. To switch to other modules, you can use `module switch`.

USING PRGENV-NVHPC

This is loaded by default, so there's no need to load additional modules. You can confirm that it is loaded by running `module list` to check that PrgEnv-nvhpc is in the list.

USING LLVM

To use the LLVM module, load the following.

```
module load mpiwrappers/cray-mpich-llvm
module load cudatoolkit-standalone
```

See the the LLVM compiling page here for more information.

USING PRGENV-GNU

To switch from PrgEnv-nvhpc to PrgEnv-gnu you can run:

```
module switch PrgEnv-nvhpc PrgEnv-gnu
```

The gcc/gfortran on Polaris was not built with GPU support. To use OpenMP on the CPU, you need to unload craype-accel-nvidia80:

```
module unload craype-accel-nvidia80
```

USING PRGENV-CRAY

To switch from PrgEnv-nvhpc to PrgEnv-cray you can run:

```
module switch PrgEnv-nvhpc PrgEnv-cray
```

To use OpenMP on the CPU only, also unload craype-accel-nvidia80:

```
module unload craype-accel-nvidia80
```

To use OpenMP on the GPU, load cudatoolkit-standalone, although this is not recommended at the moment.

```
module load cudatoolkit-standalone
```

Building on Polaris

The following table shows what compiler and flags to use with which PrgEnv:

module	compiler	flags
PrgEnv-nvhpc	cc/CC/ftn (nvc/nvc++/nvfortran)	-mp=gpu -gpu=cc80
llvm	mpicc/mpicxx (clang/clang++)	-fopenmp -fopenmp-targets=nvptx64-nvidia-cuda
PrgEnv-gnu	cc/CC/ftn (gcc/g++/gfortran)	-fopenmp
PrgEnv-cray	cc/CC/ftn	-fopenmp

For example to compile a simple code hello.cpp:

FOR PRGENV-NVHPC, AFTER LOADING THE MODULES AS DISCUSSED ABOVE WE WOULD USE:

```
CC -mp=gpu -gpu=cc80 hello.cpp
ftn -mp=gpu -gpu=cc80 hello.F90
```

FOR LLVM, AFTER LOADING THE MODULES AS DISCUSSED ABOVE:

```
mpicxx -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda hello.cpp
```

FOR PRGENV-GNU, AFTER LOADING THE MODULES AS DISCUSSED ABOVE WE WOULD USE:

```
CC -fopenmp hello.cpp
ftn -fopenmp hello.F90
```

FOR PRGENV-CRAY, AFTER LOADING THE MODULES AS DISCUSSED ABOVE WE WOULD USE:

```
CC -fopenmp hello.cpp
ftn -fopenmp hello.F90
```

Running on Polaris

To run, you can run the produced executable or with mpiexec in a job script, and then submit the script to the Polaris queue, like:

```
$ cat submit.sh
#!/bin/sh
#PBS -l select=1:system=polaris
#PBS -l walltime=0:30:00
#PBS -q debug
#PBS -A Catalyst
#PBS -l filesystems=home:eagle

cd ${PBS_O_WORKDIR}
 mpiexec -n 1 ./executable
$ # submit to the queue:
$ qsub -l select=1:system=polaris -l walltime=0:30:00 -l filesystems=home:eagle -q debug -A Catalyst ./submit.sh
```

In the above, having the PBS options in the script and on the command line is redundant, but we put it there to show both ways of launching. This submits the script to one node in the debug queue on

Polaris, requesting 30 min and the eagle and home filesystems. It will charge project Catalyst for the time.

More details for setting up the job script are in Job Scheduling and Execution section.

Example

```
$ cat hello.cpp
#include <stdio.h>
#include <omp.h>

int main( int argv, char** argc ) {

  printf( "Number of devices: %d\n", omp_get_num_devices() );

  #pragma omp target
  {
    if( !omp_is_initial_device() )
      printf( "Hello world from accelerator.\n" );
    else
      printf( "Hello world from host.\n" );
  }
  return 0;
}

$ cat hello.F90
program  main
  use omp_lib
  implicit none
  integer flag

  write(*,*) "Number of devices:", omp_get_num_devices()

  !$omp target map(from:flag)
    if( .not. omp_is_initial_device() ) then
      flag = 1
    else
      flag = 0
   endif
  !$omp end target

   if( flag == 1 ) then
      print *, "Hello world from accelerator"
   else
      print *, "Hello world from host"
   endif

 end program main

$ # To compile
$ CC -mp=gpu -gpu=cc80 hello.cpp -o c_test
$ ftn -mp=gpu -gpu=cc80 hello.F90 -o f_test

$ # To run
$ mpiexec -n 1 ./c_test
Number of devices: 4
Hello world from accelerator.
$ mpiexec -n 1 ./f_test
 Number of devices:           4
 Hello world from accelerator
```

6.9.2 SYCL

SYCL (pronounced ‘sickle’) is a royalty-free, cross-platform abstraction layer that enables code for heterogeneous processors to be written using standard ISO C++ with the host and kernel code for an application contained in the same source file.

- Specification: https://www.khronos.org/sycl/
- Source code of the compiler: https://github.com/intel/llvm
- ALCF Tutorial: https://github.com/argonne-lcf/sycltrain

```
module load oneapi
```

:warning: This module (compilers, libraries) gets built periodically from the latest open-source rather than releases. As such, these compilers will get new features and updates quickly that may break on occasion.

Dependencies

- SYCL programming model is supported through `oneapi` compilers that were built from source-code
- Loading this module switches the default programming environment to GNU and with the following dependencies
- PrgEnv-gnu
- cudatoolkit-standalone
- Environment Variable set: `SYCL_DEVICE_SELECTOR=ext_oneapi_cuda:gpu`

Example (memory intilization)

```
#include <sycl/sycl.hpp>

int main(){
    const int N= 100;
    sycl::queue Q;
    float *A = sycl::malloc_shared<float>(N, Q);

    std::cout << "Running on "
              << Q.get_device().get_info<sycl::info::device::name>()
              << "\n";

    // Create a command_group to issue command to the group
    Q.parallel_for(N, [=](sycl::item<1> id) { A[id] = 0.1 * id; }).wait();

    for (size_t i = 0; i < N; i++)
```

```
        std::cout << "A[ " << i << " ] = " << A[i] << std::endl;
    return 0;
}
```

Compile and Run

```
$ clang++ -std=c++17 -sycl-std=2020 -fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend --cuda-gpu-arch=sm_80 main.cpp
$ ./a.out
```

Example (using GPU-aware MPI)

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

#include <sycl/sycl.hpp>

// Modified from NERSC website:
// https://docs.nersc.gov/development/programming-models/mpi
int main(int argc, char *argv[]) {

    int myrank, num_ranks;
    double *val_device;
    double *val_host;
    char machine_name[MPI_MAX_PROCESSOR_NAME];
    int name_len=0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);
    MPI_Get_processor_name(machine_name, &name_len);

    sycl::queue q{sycl::gpu_selector_v};

    std::cout << "Rank #" << myrank << " runs on: " << machine_name
              << ", uses device: "
              << q.get_device().get_info<sycl::info::device::name>() << "\n";

    MPI_Barrier(MPI_COMM_WORLD);
    int one=1;
    val_host = (double *)malloc(one*sizeof(double));
    val_device = sycl::malloc_device<double>(one,q);

    const size_t size_of_double = sizeof(double);
    *val_host = -1.0;
    if (myrank != 0) {
        std::cout << "I am rank " << myrank
                  << " and my initial value is: " << *val_host << "\n";
    }

    if (myrank == 0) {
        *val_host = 42.0;
        q.memcpy(val_device,val_host,size_of_double).wait();
        std::cout << "I am rank " << myrank
                  << " and will broadcast value: " << *val_host << "\n";
    }

    MPI_Bcast(val_device, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    double check = 42.0;
    if (myrank != 0) {
        //Device to Host
        q.memcpy(val_host,val_device,size_of_double).wait();
        assert(*val_host == check);
        std::cout << "I am rank " << myrank
                  << " and received broadcast value: " << *val_host << "\n";
    }

    sycl::free(val_device,q);
    free(val_host);

    MPI_Finalize();

    return 0;
}
```

Load Modules

```
module load oneapi
module load mpiwrappers/cray-mpich-oneapi
export MPICH_GPU_SUPPORT_ENABLED=1
```

Compile and Run

```
$ mpicxx -L/opt/cray/pe/mpich/8.1.16/gtl/lib -lmpi_gtl_cuda -std=c++17 -fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend --cuda-gpu-arch=sm_80 main.cpp
$ mpiexec -n 2 --ppn 2 --depth=1 --cpu-bind depth ./set_affinity_gpu_polaris.sh ./a.out
```

For further details regarding the arguments passed to `mpiexec` command shown above, please visit the Job Scheduling and Execution section. A simple example describing the details and execution of the `set_affinity_gpu_polaris.sh` file can be found here.

Note: By default, there is no GPU-aware MPI library linking support. The example above shows how the user can enable the linking by specifying the path to the GTL (GPU Transport Layer) library (`libmpi_gtl_cuda`) to the link line.

6.9.3 oneAPI Math Kernel Library (oneMKL) Interfaces

oneMKL Interfaces is an open-source implementation of the oneMKL Data Parallel C++ (DPC++) interface according to the oneMKL specification. It works with multiple devices (backends) using device-specific libraries underneath.

oneMKL is part of oneAPI. Various backend supported are shown below. More Information here. | User Application | Third-Party Library | |-----------------|--| | | cuBLAS | | oneMKL interface | cuSOLVER | | | cuRAND |

Example (using onemkl::gemm)

The following snippet shows how to compile and run a SYCL code with oneMKL library. For instance, a GPU-based GEMM is performed using `mkl::gemm` API and the results are compared to a CPU-based GEMM performed using the traditional blas (e.g., AOCL-BLIS) library.

```
#include <limits>
#include <random>

#include <sycl/sycl.hpp>

#include <oneapi/mkl.hpp>  // ONEMKL GPU header
#include <cblas.h>         // BLIS   CPU header

// Matrix size constants
#define SIZE 4800 // Must be a multiple of 8.
#define M SIZE / 8
#define N SIZE / 4
#define P SIZE / 2

//////////////////////////////////////////////////////////////////////////////////////////

bool ValueSame(double a, double b) { return std::fabs(a - b) < 1.0e-08; }
int VerifyResult(double *c_A, double *c_B) {
  bool MismatchFound = false;
```

```
  for (size_t i = 0; i < M; i++) {
    for (size_t j = 0; j < P; j++) {
      if (!ValueSame(c_A[i * P + j], c_B[i * P + j])) {
        std::cout << "fail - The result is incorrect for element: [" << i << ", " << j
                  << "], expected: " << c_A[i * P + j] << " , but got: " << c_B[i * P + j]
                  << std::endl;
        MismatchFound = true;
      }
    }
  }

  if (!MismatchFound) {
    std::cout << "SUCCESS - The results are correct!" << std::endl;
    return 0;
  } else {
    std::cout << "FAIL - The results mis-match!" << std::endl;
    return -1;
  }
}

////////////////////////////////////////////////////////////////////////////////

int main() {
  std::random_device rd;  // Will be used to obtain a seed for the random number engine
  std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
  std::uniform_real_distribution<> dis(1.0, 2.0);

  // C = alpha * op(A) * op(B)  + beta * C
  oneapi::mkl::transpose transA = oneapi::mkl::transpose::nontrans;
  oneapi::mkl::transpose transB = oneapi::mkl::transpose::nontrans;

  // matrix data sizes
  int m = M;
  int n = P;
  int k = N;

  // leading dimensions of data
  int ldA = k;
  int ldB = n;
  int ldC = n;

  // set scalar fp values
  double alpha = 1.0;
  double beta = 0.0;

  // 1D arrays on host side
  double *A;
  double *B;
  double *C_host_onemkl, *C_cblas;

  A = new double[M * N]{};
  B = new double[N * P]{};
  C_cblas = new double[M * P]{};
  C_host_onemkl = new double[M * P]{};

  // prepare matrix data with ROW-major style
  // A(M, N)
  for (size_t i = 0; i < M; i++)
    for (size_t j = 0; j < N; j++)
      A[i * N + j] = dis(gen);
  // B(N, P)
  for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < P; j++)
      B[i * P + j] = dis(gen);

  std::cout << "Problem size: c(" << M << "," << P << ") = a(" << M << "," << N << ") * b(" << N
            << "," << P << ")" << std::endl;

  // Resultant matrix: C_cblas
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k, alpha, A, ldA, B, ldB, beta,
              C_cblas, ldC);

  // Resultant matrix: C_onemkl
  sycl::queue q(sycl::property_list{sycl::property::queue::in_order{}});
  std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl << std::endl;

  double* A_dev        = sycl::malloc_device<double>(M*N, q);
  double* B_dev        = sycl::malloc_device<double>(N*P, q);
  double* C_dev_onemkl = sycl::malloc_device<double>(M*P, q);

  q.memcpy(A_dev, A, (M*N) * sizeof(double));
  q.memcpy(B_dev, B, (N*P) * sizeof(double));

  auto gemm_event = oneapi::mkl::blas::column_major::gemm(q, transB, transA, n, m, k, alpha, B_dev, ldB, A_dev, ldA, beta, C_dev_onemkl, ldC);

  q.memcpy(C_host_onemkl, C_dev_onemkl, (M*P) * sizeof(double));

  q.wait();
  std::cout << "Verify results between OneMKL & CBLAS: ";
  int result_cblas = VerifyResult(C_cblas, C_host_onemkl);

  delete[] A;
  delete[] B;
  delete[] C_cblas;
```

```
  delete[] C_host_onemkl;
  sycl::free(A_dev, q);
  sycl::free(B_dev, q);
  sycl::free(C_dev_onemkl, q);
  return result_cblas;
}
```

Compile and Run The user would need to provide paths the math-libraris as shown below. Also please provide AOCL library for CPU GEMM by `module load aocl`. Environment variables `MKLROOT` is defined with `oneapi` module & `AOCL_ROOT` is defined with `aocl` module. Note: Please pay attention to the linker options for AOCL & oneMKL libraries.

```
$ clang++ -std=c++17 -sycl-std=2020 -O3 -fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend --cuda-gpu-arch=sm_80 -L$AOCL_ROOT/lib -lblis -L$MKLROOT/lib -lonemkl
sycl_onemkl_gemm.cpp -o sycl_onemkl_gemm.out
```

6.9.4 Kokkos

Kokkos

Kokkos Core implements a programming model in C++ for writing performance portable applications targeting all major HPC platforms. For that purpose it provides abstractions for both parallel execution of code and data management. Kokkos is designed to target complex node architectures with N-level memory hierarchies and multiple types of execution resources. It currently can use Serial and OpenMP (threads) for CPU execution spaces ("backends") and CUDA, HIP, SYCL, and OpenMPTarget for GPU execution spaces. By convention, Kokkos only allows one GPU backend at a time.

KOKKOS DOCUMENTATION

- Kokkos-core Wiki
- Kokkos github

KOKKOS ON POLARIS

The prebuilt Kokkos on polaris includes 3 backends: Serial and OpenMP for CPU execution and CUDA for GPU execution. To use it, run

```
module use /soft/modulefiles
module load kokkos
```

This sets the following environment variables, some of which are used by `cmake`:

- `KOKKOS_HOME` - path to the `lib64/`, `include/` files installed
- `LIBRARY_PATH` - prepends `$KOKKOS_HOME/lib64` to this variable used by `cmake`
- `CPATH` - prepends `$KOKKOS_HOME/include` to this variable used by `cmake`
- `LD_LIBRARY_PATH` - prepends `$KOKKOS_HOME/lib64` to this variable

BUILDING A KOKKOS APPLICATION USING `CMAKE`

Add these lines to `CMakeLists.txt`:

```
find_package(Kokkos REQUIRED)
target_link_libraries(myTarget Kokkos::kokkoscore)
```

Here is a simple example `CMakeLists.txt` to compile an example program:

```
cmake_minimum_required(VERSION 3.22)
project(buildExample)
find_package(Kokkos REQUIRED)

set(buildExample_SOURCE_DIR ".")

set(top_SRCS
  ${buildExample_SOURCE_DIR}/example1.cpp)

set(SOURCE_FILES ${top_SRCS})

add_executable(example1_sycl_aot ${SOURCE_FILES})
target_link_libraries(example1_sycl_aot Kokkos::kokkoscore)
target_include_directories(example1_sycl_aot PUBLIC ${buildExample_SOURCE_DIR})
```

Configure and build it like this:

```
mkdir build
cd build
cmake -DCMAKE_CXX_COMPILER=CC -DCMAKE_C_COMPILER=cc ..
make
```

BUILDING A KOKKOS APPLICATION USING `MAKE`

Here's an example `Makefile`:

```
# KOKKOS_HOME set via:
#   module load kokkos

# You can look at the first lines of $KOKKOS_HOME/KokkosConfigCommon.cmake to
# see the flags used in cmake configuration of the kokkos library build. The
# default Kokkos module on Polaris was built with PrgEnv-nvhpc and includes
# Serial, OpenMP (threads) and CUDA backends. So you should have that
# environment module loaded and include compiler flags for cuda and openmp:

# Cray MPI wrapper for C++ and C compilers:
CXX=CC
CC=cc

CPPFLAGS=-cuda -fopenmp
LDFLAGS=

LDFLAGS=$(CPPFLAGS) $(LDFLAGS)
LDLIBS=-L$(KOKKOS_HOME)/lib64 -lkokkoscore -lkokkossimd -lpthread

SRCS=example1.cpp
OBJS=$(subst .cpp,.o,$(SRCS))

all: example1_polaris

example1_polaris: $(OBJS)
	$(CXX) $(LDFLAGS) -o example1_polaris $(OBJS) $(LDLIBS)

example1.o: example1.cpp

clean:
	rm -f $(OBJS)

distclean: clean
	rm -f example1_polaris
```

CONFIGURING YOUR OWN KOKKOS BUILD ON POLARIS

Here are recommended environment settings and configuration to build your own kokkos libraries on Polaris:

Environment

To match what was done in the centrally-built kokkos associated with the modules discussed above, use the default programming environment `PrgEnv-nvhpc`, and use the Cray wrapper `CC` as the C++ compiler. To build Kokkos, you'll need cmake. You may also use `PrgEnv-gnu` to build kokkos (also using the Cray wrapper `CC` as the C++ compiler).

To use C++17, you'll need to work around a bug with the current `PrgEnv-nvhpc/8.3.3` environment by loading a cudatoolkit-standalone module:

```
module load cmake cudatoolkit-standalone/11.6.2
```

CMake Configuration

This example builds three backends: OpenMP, Serial, and Cuda.

```
git clone git@github.com:kokkos/kokkos.git
cd kokkos
mkdir build
cd build

cmake\
 -DCMAKE_BUILD_TYPE=RelWithDebInfo\
 -DCMAKE_INSTALL_PREFIX="./install"\
 -DCMAKE_CXX_COMPILER=CC\
 -DKokkos_ENABLE_OPENMP=ON\
 -DKokkos_ENABLE_SERIAL=ON\
 -DKokkos_ARCH_ZEN3=ON\
 -DKokkos_ARCH_AMPERE80=ON\
 -DKokkos_ENABLE_CUDA=ON\
 -DKokkos_ENABLE_AGGRESSIVE_VECTORIZATION=ON\
 -DKokkos_ENABLE_TESTS=OFF\
 -DBUILD_TESTING=OFF\
 -DKokkos_ENABLE_CUDA_LAMBDA=ON\
 -DKokkos_ENABLE_IMPL_DESUL_ATOMICS=OFF\
 -DCMAKE_CXX_STANDARD=17\
 ..

make -j16 -l16 install
```

6.10 Debugging Tools

6.10.1 CUDA-GDB

References

NVIDIA CUDA-GDB Documentation

Introduction

CUDA-GDB is the NVIDIA tool for debugging CUDA applications running on Polaris. CUDA-GDB is an extension to GDB, the GNU Project debugger. The tool provides developers with a mechanism for debugging CUDA applications running on actual hardware. This enables developers to debug applications without the potential variations introduced by simulation and emulation environments.

Step-by-step guide

DEBUG COMPILATION

NVCC, the NVIDIA CUDA compiler driver, provides a mechanism for generating the debugging information necessary for CUDA-GDB to work properly. The -g -G option pair must be passed to NVCC when an application is compiled for ease of debugging with CUDA-GDB; for example,

```
nvcc -g -G foo.cu -o foo
```

Using this line to compile the CUDA application `foo.cu` * forces `-O0` compilation, with the exception of very limited dead-code eliminations and register-spilling optimizations. * makes the compiler include debug information in the executable

RUNNING CUDA-GDB ON POLARIS COMPUTE NODES

Start an interactive job mode on Polaris as follows:

```
$ qsub -I -l select=1 -l walltime=1:00:00

$ cuda-gdb --version
NVIDIA (R) CUDA Debugger
11.4 release
Portions Copyright (C) 2007-2021 NVIDIA Corporation
GNU gdb (GDB) 10.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

$ cuda-gdb foo
```

A quick example with a stream benchmark on a Polaris compute node

```
jkwack@polaris-login-02:~> qsub -I -l select=1 -l walltime=1:00:00
qsub: waiting for job 308834.polaris-pbs-01.hsn.cm.polaris.alcf.anl.gov to start
qsub: job 308834.polaris-pbs-01.hsn.cm.polaris.alcf.anl.gov ready


Currently Loaded Modules:
  1) craype-x86-rome          4) perftools-base/22.05.0   7) cray-dsmml/0.2.2   10) cray-pmi-lib/6.0.17  13) PrgEnv-nvhpc/8.3.3
  2) libfabric/1.11.0.4.125   5) nvhpc/21.9               8) cray-mpich/8.1.16  11) cray-pals/1.1.7      14) craype-accel-nvidia80
  3) craype-network-ofi       6) craype/2.7.15            9) cray-pmi/6.1.2     12) cray-libpals/1.1.7

jkwack@x3008c0s13b1n0:~/BabelStream/build_polaris_debug> nvcc -g -G -c ../src/cuda/CUDAStream.cu  -I ../src/

jkwack@x3008c0s13b1n0:~/BabelStream/build_polaris_debug> nvcc -g -G -c ../src/main.cpp -DCUDA -I ../src/cuda/ -I ../src/

jkwack@x3008c0s13b1n0:~/BabelStream/build_polaris_debug> nvcc -g -G main.o CUDAStream.o -o cuda-stream-debug

jkwack@x3008c0s13b1n0:~/BabelStream/build_polaris_debug> ./cuda-stream-debug
BabelStream
Version: 4.0
Implementation: CUDA
Running kernels 100 times
Precision: double
Array size: 268.4 MB (=0.3 GB)
Total size: 805.3 MB (=0.8 GB)
Using CUDA device NVIDIA A100-SXM4-40GB
Driver: 11040
Function    MBytes/sec  Min (sec)   Max         Average
Copy        1313940.694 0.00041     0.00047     0.00047
Mul         1302000.791 0.00041     0.00048     0.00047
Add         1296217.720 0.00062     0.00070     0.00069
Triad       1296027.887 0.00062     0.00070     0.00069
Dot         823405.227  0.00065     0.00076     0.00075

jkwack@x3008c0s13b1n0:~/BabelStream/build_polaris_debug> cuda-gdb ./cuda-stream-debug
NVIDIA (R) CUDA Debugger
11.4 release
Portions Copyright (C) 2007-2021 NVIDIA Corporation
GNU gdb (GDB) 10.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./cuda-stream-debug...
(cuda-gdb) b CUDAStream.cu:203
Breakpoint 1 at 0x412598: CUDAStream.cu:203. (2 locations)
(cuda-gdb) r
Starting program: /home/jkwack/BabelStream/build_polaris_debug/cuda-stream-debug
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
BabelStream
Version: 4.0
Implementation: CUDA
Running kernels 100 times
Precision: double
Array size: 268.4 MB (=0.3 GB)
Total size: 805.3 MB (=0.8 GB)
[Detaching after fork from child process 58459]
[New Thread 0x15554c6bb000 (LWP 58475)]
Using CUDA device NVIDIA A100-SXM4-40GB
Driver: 11040
[New Thread 0x15554c4ba000 (LWP 58476)]
[Switching focus to CUDA kernel 0, grid 5, block (0,0,0), thread (0,0,0), device 0, sm 0, warp 3, lane 0]

Thread 1 "cuda-stream-deb" hit Breakpoint 1, triad_kernel<double><<<(32768,1,1),(1024,1,1)>>> (a=0x155506000000, b=0x1554f6000000, c=0x1554e6000000)
    at ../src/cuda/CUDAStream.cu:203
203   a[i] = b[i] + scalar * c[i];
(cuda-gdb) c
Continuing.
[Switching focus to CUDA kernel 0, grid 5, block (1,0,0), thread (0,0,0), device 0, sm 0, warp 32, lane 0]

Thread 1 "cuda-stream-deb" hit Breakpoint 1, triad_kernel<double><<<(32768,1,1),(1024,1,1)>>> (a=0x155506000000, b=0x1554f6000000, c=0x1554e6000000)
    at ../src/cuda/CUDAStream.cu:203
203   a[i] = b[i] + scalar * c[i];
(cuda-gdb) info locals
i = 1024
(cuda-gdb) p b[i]
$1 = 0.040000000000000008
```

```
(cuda-gdb) p scalar
$2 = 0.40000000000000002
(cuda-gdb) p c[i]
$3 = 0.14000000000000001
(cuda-gdb) d 1
(cuda-gdb) c
Continuing.
Function    MBytes/sec  Min (sec)   Max         Average
Copy        1314941.553 0.00041     0.00041     0.00041
Mul         1301022.680 0.00041     0.00042     0.00041
Add         1293858.147 0.00062     0.00063     0.00063
Triad       1297681.929 0.00062     0.00063     0.00062
Dot         828446.963  0.00065     0.00066     0.00065
[Thread 0x15554c4ba000 (LWP 58476) exited]
[Thread 0x15554c6bb000 (LWP 58475) exited]
[Inferior 1 (process 58454) exited normally]
(cuda-gdb) q

jkwack@x3008c0s13b1n0:~/BabelStream/build_polaris_debug>
```

6.11 Performance Tools

6.11.1 NVIDIA Nsight tools

References

NVIDIA Nsight Systems Documentation
NVIDIA Nsight Compute Documentation

Introduction

NVIDIA® Nsight™ Systems provides developers a system-wide visualization of an applications performance. Developers can optimize bottlenecks to scale efficiently across any number or size of CPUs and GPUs on Polaris. For further optimizations to compute kernels developers should use Nsight Compute.

The NVIDIA Nsight Compute is an interactive kernel profiler for CUDA applications. It provides detailed performance metrics and API debugging via a user interface and command line tool.

In addition, the baseline feature of this tool allows users to compare results within the tool. NVIDIA Nsight Compute provides a customizable and data-driven user interface, metric collection, and can be extended with analysis scripts for post-processing results.

Step-by-step guide

COMMON PART ON POLARIS

Build your application for Polaris, and then submit your job script to Polaris or start an interactive job mode on Polaris as follows:

```
$ qsub -I -l select=1 -l walltime=1:00:00 -l filesystems=home:grand -q debug -A <project-name>

$ module load cudatoolkit-standalone/11.8.0
$ module li

Currently Loaded Modules:
  1) craype-x86-rome          6) craype/2.7.15        11) cray-pals/1.1.7
  2) libfabric/1.11.0.4.125   7) cray-dsmml/0.2.2     12) cray-libpals/1.1.7
  3) craype-network-ofi       8) cray-mpich/8.1.16    13) PrgEnv-nvhpc/8.3.3
  4) perftools-base/22.05.0   9) cray-pmi/6.1.2       14) craype-accel-nvidia80
  5) nvhpc/21.9              10) cray-pmi-lib/6.0.17  15) cudatoolkit-standalone/11.8.0

$ nsys --version
NVIDIA Nsight Systems version 2022.4.2.1-df9881f

$ ncu --version
```

```
NVIDIA (R) Nsight Compute Command Line Profiler
Copyright (c) 2018-2022 NVIDIA Corporation
Version 2022.3.0.0 (build 31729285) (public-release)
```

NSIGHT SYSTEMS

Run your application with Nsight Systems as follows:

```
$ nsys profile -o {output_filename} --stats=true ./{your_application}
```

NSIGHT COMPUTE

Run your application with Nsight Compute.

```
$ ncu --set detailed -k {kernel_name} -o {output_filename} ./{your_application}
```

Remark: Without -o option, Nsight Compute provides performance data as a standard output

POST-PROCESSING THE PROFILED DATA

Post-processing via CLI

```
$ nsys stats {output_filename}.qdrep
$ ncu -i {output_filename}.ncu-rep
```

Post-processing on your local system via GUI

- Install NVIDIA Nsight Systems and NVIDIA Nsight Compute after downloading both of them from the NVIDIA Developer Zone.
  Remark: Local client version should be the same as or newer than NVIDIA Nsight tools on Polaris.
- Download nsys output files (i.e., ending with .qdrep and . sqlite) to your local system, and then open them with NVIDIA Nsight Systems on your local system.
- Download ncu output files (i.e., ending with .ncu-rep) to your local system, and then open them with NVIDIA Nsight Compute on your local system.

MORE OPTIONS FOR PERFORMANCE ANALYSIS WITH NSIGHT SYSTEMS AND NSIGHT COMPUTE

```
$ nsys --help
$ ncu --help
```

A quick example

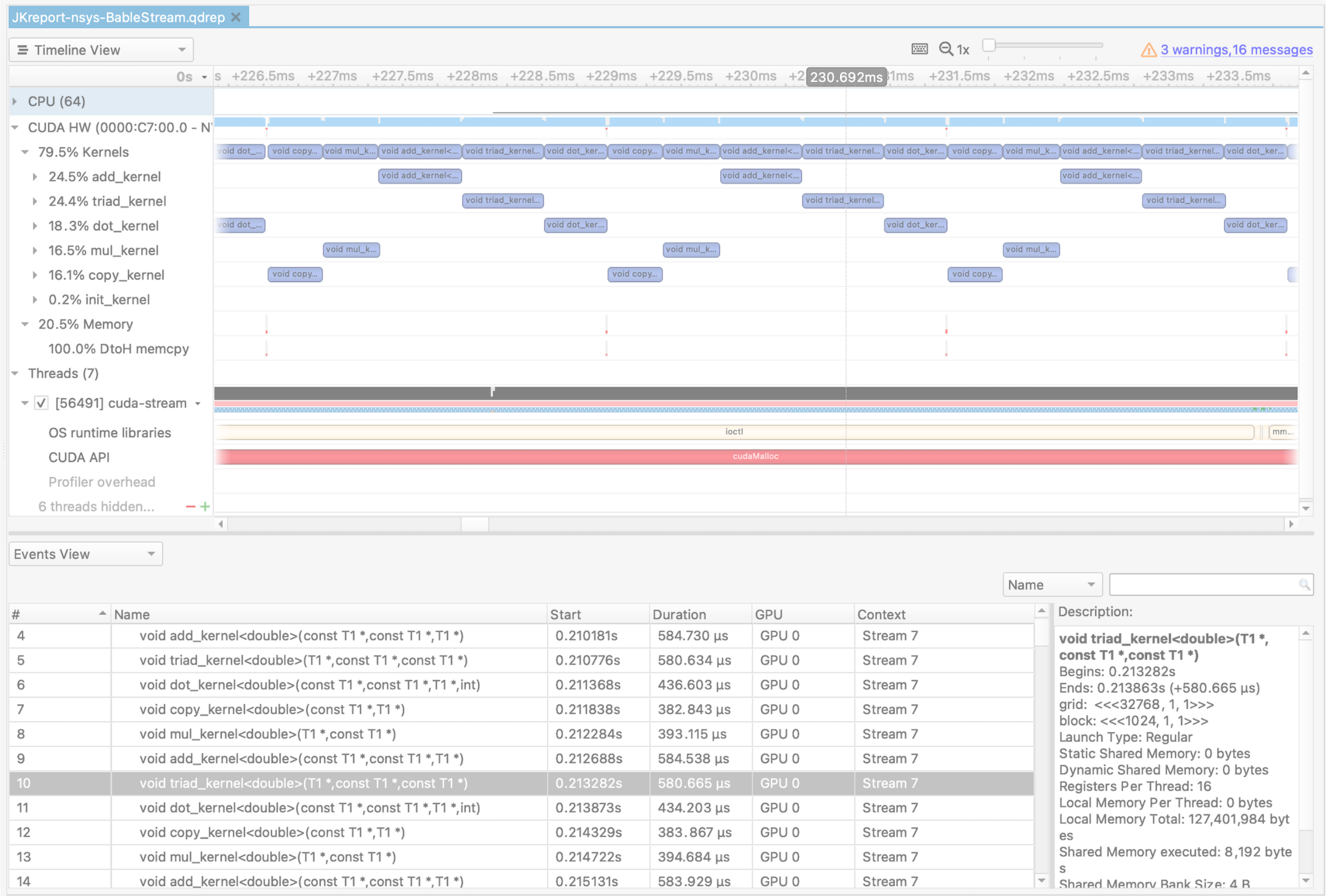
NSIGHT SYSTEMS

Running a stream benchmark with Nsight Systems

```
jkwack@x3008c0s13b1n0:~/BabelStream/build_polaris> nsys profile -o JKreport-nsys-BableStream --stats=true ./cuda-stream
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
Collecting data...
BabelStream
Version: 4.0
Implementation: CUDA
Running kernels 100 times
Precision: double
Array size: 268.4 MB (=0.3 GB)
Total size: 805.3 MB (=0.8 GB)
Using CUDA device NVIDIA A100-SXM4-40GB
Driver: 11040
Function    MBytes/sec  Min (sec)   Max         Average
Copy        1368294.603 0.00039     0.00044     0.00039
Mul         1334324.779 0.00040     0.00051     0.00041
Add         1358476.737 0.00059     0.00060     0.00059
Triad       1366095.332 0.00059     0.00059     0.00059
Dot         1190200.569 0.00045     0.00047     0.00046
Processing events...
Saving temporary "/var/tmp/pbs.308834.polaris-pbs-01.hsn.cm.polaris.alcf.anl.gov/nsys-report-f594-c524-6b4c-300a.qdstrm" file to disk...

Creating final output files...
Processing [==============================================================100%]
Saved report file to "/var/tmp/pbs.308834.polaris-pbs-01.hsn.cm.polaris.alcf.anl.gov/nsys-report-f594-c524-6b4c-300a.qdrep"
Exporting 7675 events: [=================================================100%]

Exported successfully to
/var/tmp/pbs.308834.polaris-pbs-01.hsn.cm.polaris.alcf.anl.gov/nsys-report-f594-c524-6b4c-300a.sqlite


CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls  Average (ns)  Minimum (ns)  Maximum (ns)  StdDev (ns)           Name
 -------  ---------------  ---------  ------------  ------------  ------------  ------------  ----------------------
    41.5      197,225,738        401     491,834.8       386,695       592,751      96,647.5  cudaDeviceSynchronize
    35.4      168,294,004          4  42,073,501.0       144,211   167,547,885  83,649,622.0  cudaMalloc
    22.5      106,822,589        103   1,037,112.5       446,617    20,588,840   3,380,727.4  cudaMemcpy
     0.4        1,823,597        501       3,639.9         3,166        24,125       1,228.9  cudaLaunchKernel
     0.2        1,166,186          4     291,546.5       130,595       431,599     123,479.8  cudaFree



CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average (ns)  Minimum (ns)  Maximum (ns)  StdDev (ns)                                                  Name
 -------  ---------------  ---------  ------------  ------------  ------------  -----------  ----------------------------------------------------------
    24.5       58,415,138        100     584,151.4       582,522       585,817        543.0  void add_kernel<double>(const T1 *, const T1 *, T1 *)
    24.4       58,080,329        100     580,803.3       579,802       582,586        520.5  void triad_kernel<double>(T1 *, const T1 *, const T1 *)
    18.3       43,602,345        100     436,023.5       430,555       445,979      2,619.5  void dot_kernel<double>(const T1 *, const T1 *, T1 *, int)
    16.5       39,402,677        100     394,026.8       392,444       395,708        611.5  void mul_kernel<double>(T1 *, const T1 *)
    16.1       38,393,119        100     383,931.2       382,556       396,892      1,434.1  void copy_kernel<double>(const T1 *, T1 *)
     0.2          523,355          1     523,355.0       523,355       523,355          0.0  void init_kernel<double>(T1 *, T1 *, T1 *, T1, T1, T1)



CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Count  Average (ns)  Minimum (ns)  Maximum (ns)  StdDev (ns)      Operation
 -------  ---------------  -----  ------------  ------------  ------------  -----------  ------------------
   100.0       61,323,171    103     595,370.6         2,399    20,470,146  3,439,982.0  [CUDA memcpy DtoH]



CUDA Memory Operation Statistics (by size):

 Total (MB)  Count  Average (MB)  Minimum (MB)  Maximum (MB)  StdDev (MB)      Operation
 ----------  -----  ------------  ------------  ------------  -----------  ------------------
    805.511    103         7.820         0.002       268.435       45.361  [CUDA memcpy DtoH]



Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls  Average (ns)  Minimum (ns)  Maximum (ns)  StdDev (ns)        Name
 -------  ---------------  ---------  ------------  ------------  ------------  ------------  --------------
    85.9      600,896,697         20  30,044,834.9         3,477   100,141,768  42,475,064.1  poll
    13.5       94,610,402      1,201      78,776.4         1,002    11,348,375     402,562.6  ioctl
     0.2        1,374,312         79      17,396.4         3,486       434,715      48,015.2  mmap64
     0.1          877,705         51      17,209.9         1,031       748,723     104,491.6  fopen
     0.1          741,969         12      61,830.8        17,272       256,852      64,706.5  sem_timedwait
     0.1          529,563        120       4,413.0         1,292        20,579       2,134.3  open64
     0.0          251,602          4      62,900.5        57,337        72,126       6,412.6  pthread_create
     0.0           93,461         18       5,192.3         1,011        19,386       4,401.0  mmap
     0.0           37,621         11       3,420.1         1,302        11,672       2,867.6  munmap
     0.0           35,735          9       3,970.6         1,723         6,251       1,477.2  fgetc
     0.0           33,533          1      33,533.0        33,533        33,533           0.0  fgets
```

```
     0.0         26,832         13      2,064.0       1,452       3,366       542.6  write
     0.0         21,341          5      4,268.2       1,213       9,738     3,378.3  putc
     0.0         20,838          6      3,473.0       1,763       6,853     1,801.1  open
     0.0         17,016         10      1,701.6       1,523       1,834        96.9  read
     0.0         11,430          8      1,428.8       1,082       1,583       151.9  fclose
     0.0          6,202          1      6,202.0       6,202       6,202         0.0  pipe2
     0.0          5,961          2      2,980.5       2,254       3,707     1,027.4  socket
     0.0          5,670          2      2,835.0       2,795       2,875        56.6  fwrite
     0.0          5,481          1      5,481.0       5,481       5,481         0.0  connect
     0.0          5,279          2      2,639.5       1,743       3,536     1,267.8  fread
     0.0          1,082          1      1,082.0       1,082       1,082         0.0  bind

Report file moved to "/home/jkwack/BabelStream/build_polaris/JKreport-nsys-BableStream.qdrep"
Report file moved to "/home/jkwack/BabelStream/build_polaris/JKreport-nsys-BableStream.sqlite"
```

Reviewing the Nsight Systems data via GUI



NSIGHT COMPUTE

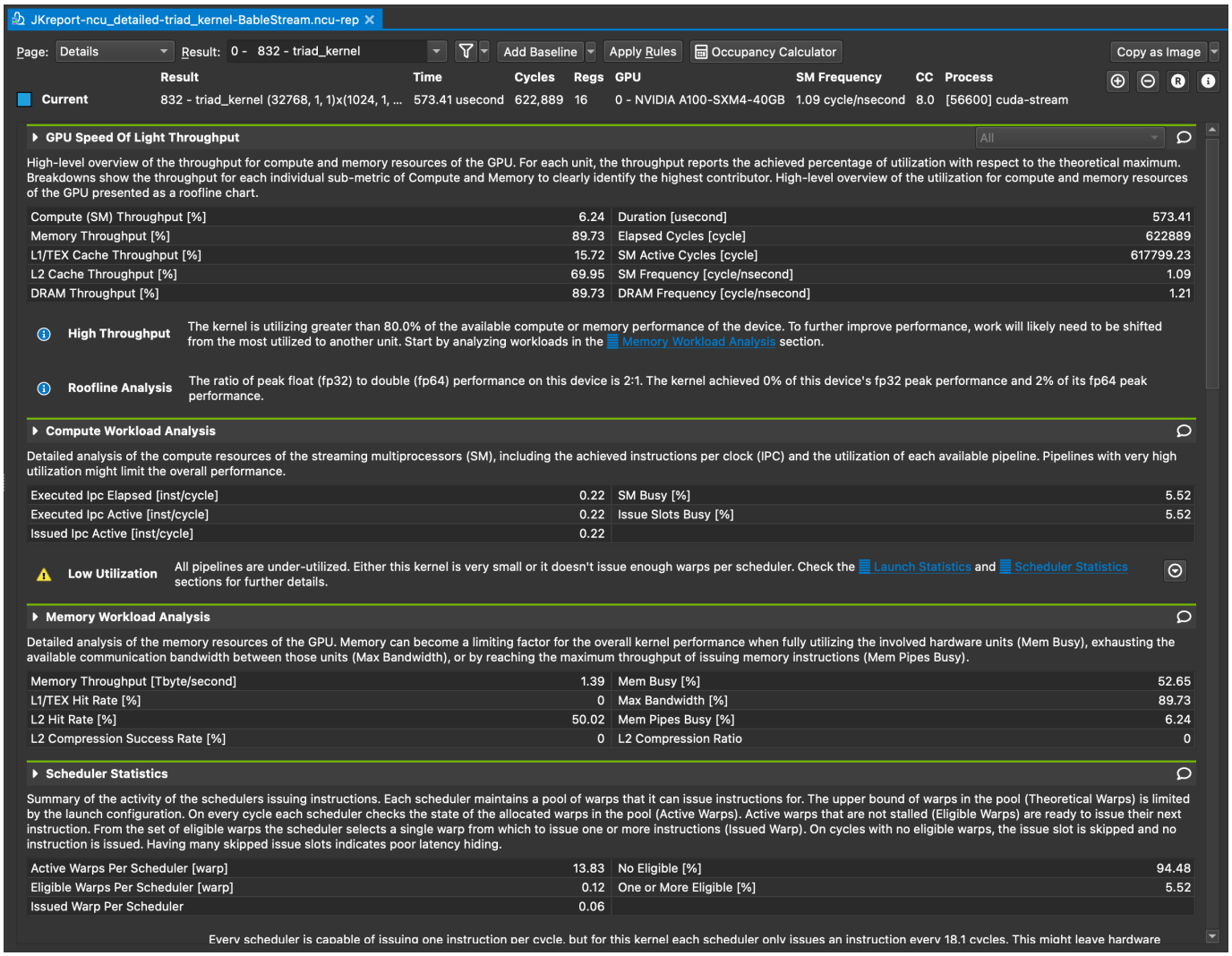
Running a stream benchmark with Nsight Compute for triad_kernel

```
jkwack@x3008c0s13b1n0:~/BabelStream/build_polaris> ncu --set detailed -k triad_kernel -o JKreport-ncu_detailed-triad_kernel-BableStream ./cuda-stream
BabelStream
Version: 4.0
Implementation: CUDA
Running kernels 100 times
Precision: double
Array size: 268.4 MB (=0.3 GB)
Total size: 805.3 MB (=0.8 GB)
==PROF== Connected to process 56600 (/home/jkwack/BabelStream/build_polaris/cuda-stream)
Using CUDA device NVIDIA A100-SXM4-40GB
Driver: 11040
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
```

```
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
==PROF== Profiling "triad_kernel": 0%....50%....100% - 18 passes
Function    MBytes/sec  Min (sec)   Max         Average
Copy        1331076.105 0.00040     0.00042     0.00041
Mul         1304696.608 0.00041     0.00043     0.00042
Add         1322600.587 0.00061     0.00062     0.00061
```

```
Triad       1327.700    0.60654     0.62352     0.61106
Dot         850376.762  0.00063     0.00070     0.00065
==PROF== Disconnected from process 56600
==PROF== Report: /home/jkwack/BabelStream/build_polaris/JKreport-ncu_detailed-triad_kernel-BableStream.ncu-rep
```

Reviewing the Nsight Compute data via GUI


JKreport-ncu_detailed-triad_kernel-BableStream.ncu-rep
Page: Details
Result: 0 - 832 - triad_kernel
Add Baseline
Apply Rules
Occupancy Calculator
Copy as Image
Result Time Cycles Regs GPU SM Frequency CC Process
Current 832 - triad_kernel (32768, 1, 1)x(1024, 1, ... 573.41 usecond 622,889 16 0 - NVIDIA A100-SXM4-40GB 1.09 cycle/nsecond 8.0 [56600] cuda-stream
GPU Speed Of Light Throughput
All
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
Compute (SM) Throughput [%] 6.24
Memory Throughput [%] 89.73
L1/TEX Cache Throughput [%] 15.72
L2 Cache Throughput [%] 69.95
DRAM Throughput [%] 89.73
Duration [usecond] 573.41
Elapsed Cycles [cycle] 622889
SM Active Cycles [cycle] 617799.23
SM Frequency [cycle/nsecond] 1.09
DRAM Frequency [cycle/nsecond] 1.21
High Throughput
The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.
Roofline Analysis
The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 0% of this device's fp32 peak performance and 2% of its fp64 peak performance.
Compute Workload Analysis
Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.
Executed Ipc Elapsed [inst/cycle] 0.22
Executed Ipc Active [inst/cycle] 0.22
Issued Ipc Active [inst/cycle] 0.22
SM Busy [%] 5.52
Issue Slots Busy [%] 5.52
Low Utilization
All pipelines are under-utilized. Either this kernel is very small or it doesn't issue enough warps per scheduler. Check the Launch Statistics and Scheduler Statistics sections for further details.
Memory Workload Analysis
Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy).
Memory Throughput [Tbyte/second] 1.39
L1/TEX Hit Rate [%] 0
L2 Hit Rate [%] 50.02
L2 Compression Success Rate [%] 0
Mem Busy [%] 52.65
Max Bandwidth [%] 89.73
Mem Pipes Busy [%] 6.24
L2 Compression Ratio 0
Scheduler Statistics
Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.
Active Warps Per Scheduler [warp] 13.83
Eligible Warps Per Scheduler [warp] 0.12
Issued Warp Per Scheduler 0.06
No Eligible [%] 94.48
One or More Eligible [%] 5.52
Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 18.1 cycles. This might leave hardware

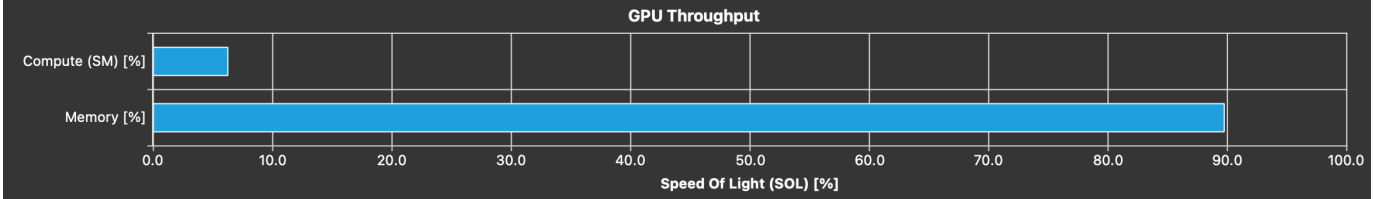
GPU Speed Of Light Throughput — All

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	6.24	Duration [usecond]	573.41
Memory Throughput [%]	89.73	Elapsed Cycles [cycle]	622889
L1/TEX Cache Throughput [%]	15.72	SM Active Cycles [cycle]	617799.23
L2 Cache Throughput [%]	69.95	SM Frequency [cycle/nsecond]	1.09
DRAM Throughput [%]	89.73	DRAM Frequency [cycle/nsecond]	1.21

High Throughput The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 0% of this device's fp32 peak performance and 2% of its fp64 peak performance.

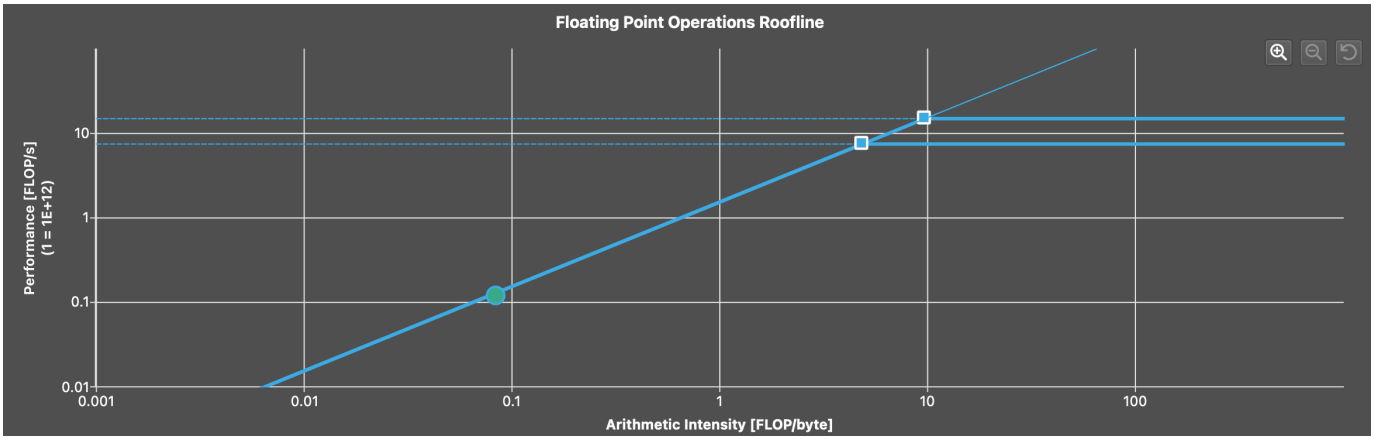


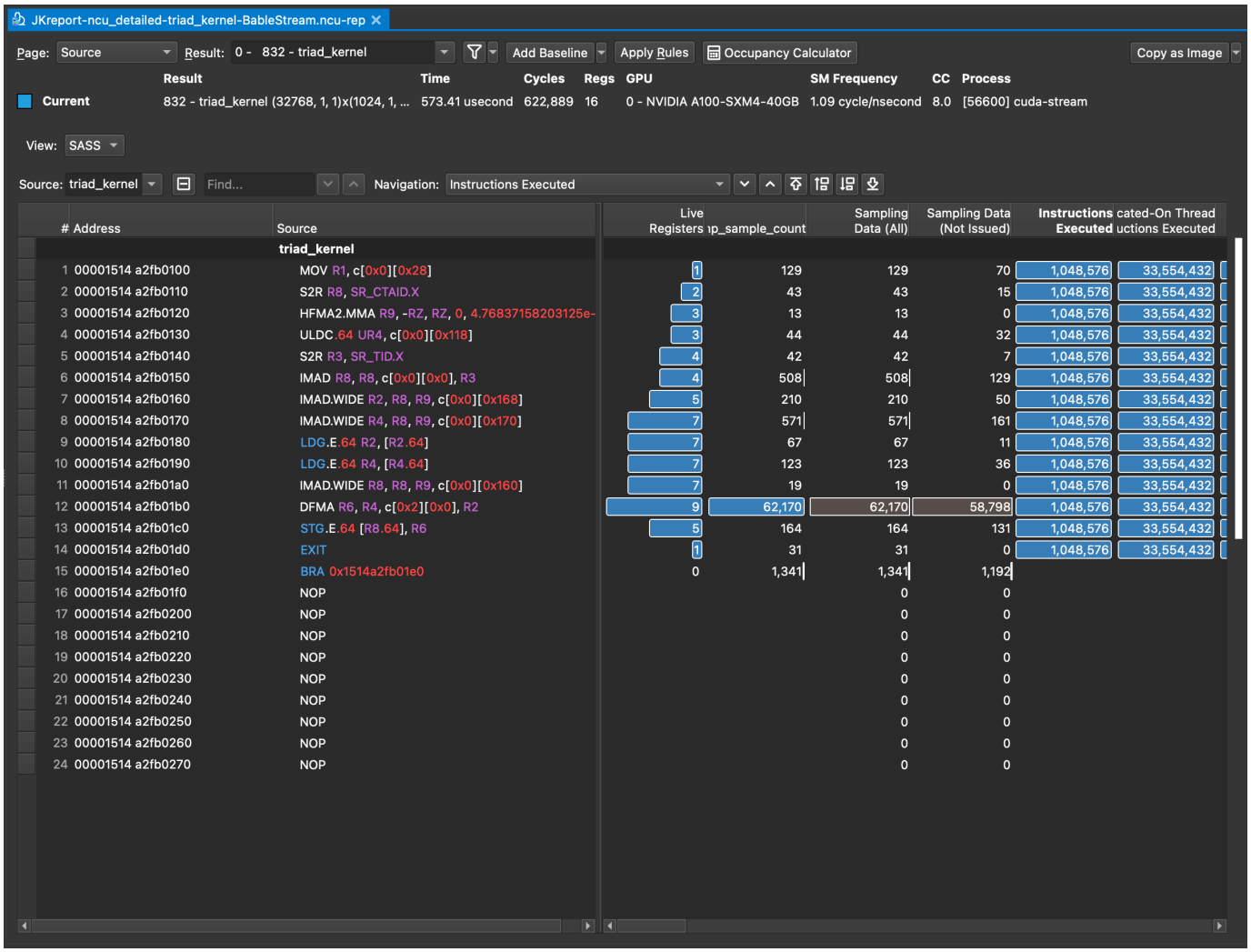
Compute Throughput Breakdown

SM: Inst Executed Pipe Lsu [%]	6.24
SM: Issue Active [%]	5.48
SM: Inst Executed [%]	5.46
SM: Mio2rf Writeback Active [%]	4.68
SM: Mio Inst Issued [%]	3.90
SM: Inst Executed Pipe Adu [%]	3.12
SM: Mio Pq Write Cycles Active [%]	3.12
SM: Mio Pq Read Cycles Active [%]	3.12
SM: Pipe Fma Cycles Active [%]	3.12
SM: Pipe Shared Cycles Active [%]	2.35
SM: Pipe Fp64 Cycles Active [%]	1.56
SM: Inst Executed Pipe Cbu Pred On Any [%]	1.56
SM: Pipe Tensor Cycles Active [%]	0.79
SM: Pipe Alu Cycles Active [%]	0.78
SM: Inst Executed Pipe Uniform [%]	0.78
SM: Inst Executed Pipe Xu [%]	0
SM: Inst Executed Pipe Tex [%]	0

Memory Throughput Breakdown

DRAM: Cycles Active [%]	89.73
L2: T Sectors [%]	52.65
DRAM: Dram Sectors [%]	51.93
L2: D Sectors [%]	37.09
L2: D Sectors Fill Device [%]	35.11
L2: T Tag Requests [%]	26.35
L2: Xbar2lts Cycles Active [%]	21.96
L2: Lts2xbar Cycles Active [%]	17.56
L1: M L1tex2xbar Req Cycles Active [%]	15.60
L1: Data Pipe Lsu Wavefronts [%]	12.48
L1: M Xbar2l1tex Read Sectors [%]	12.48
L1: Lsu Writeback Active [%]	7.80
L1: Lsuin Requests [%]	6.24
L1: Data Bank Reads [%]	4.68
L1: Data Bank Writes [%]	4.68
L1: Texin Sm2tex Req Cycles Active [%]	0.00
L1: F Wavefronts [%]	0.00




JKreport-ncu_detailed-triad_kernel-BableStream.ncu-rep
Page: Source
Result: 0 - 832 - triad_kernel
Add Baseline
Apply Rules
Occupancy Calculator
Copy as Image
Result Time Cycles Regs GPU SM Frequency CC Process
Current 832 - triad_kernel (32768, 1, 1)x(1024, 1, ... 573.41 usecond 622,889 16 0 - NVIDIA A100-SXM4-40GB 1.09 cycle/nsecond 8.0 [56600] cuda-stream
View: SASS
Source: triad_kernel
Find...
Navigation: Instructions Executed
Address Source Live Registers ıp_sample_count Sampling Data (All) Sampling Data (Not Issued) Instructions Executed cated-On Thread uctions Executed
triad_kernel
1 00001514 a2fb0100 MOV R1, c[0x0][0x28] 1 129 129 70 1,048,576 33,554,432
2 00001514 a2fb0110 S2R R8, SR_CTAID.X 2 43 43 15 1,048,576 33,554,432
3 00001514 a2fb0120 HFMA2.MMA R9, -RZ, RZ, 0, 4.76837158203125e- 3 13 13 0 1,048,576 33,554,432
4 00001514 a2fb0130 ULDC.64 UR4, c[0x0][0x118] 3 44 44 32 1,048,576 33,554,432
5 00001514 a2fb0140 S2R R3, SR_TID.X 4 42 42 7 1,048,576 33,554,432
6 00001514 a2fb0150 IMAD R8, R8, c[0x0][0x0], R3 4 508 508 129 1,048,576 33,554,432
7 00001514 a2fb0160 IMAD.WIDE R2, R8, R9, c[0x0][0x168] 5 210 210 50 1,048,576 33,554,432
8 00001514 a2fb0170 IMAD.WIDE R4, R8, R9, c[0x0][0x170] 7 571 571 161 1,048,576 33,554,432
9 00001514 a2fb0180 LDG.E.64 R2, [R2.64] 7 67 67 11 1,048,576 33,554,432
10 00001514 a2fb0190 LDG.E.64 R4, [R4.64] 7 123 123 36 1,048,576 33,554,432
11 00001514 a2fb01a0 IMAD.WIDE R8, R8, R9, c[0x0][0x160] 7 19 19 0 1,048,576 33,554,432
12 00001514 a2fb01b0 DFMA R6, R4, c[0x2][0x0], R2 9 62,170 62,170 58,798 1,048,576 33,554,432
13 00001514 a2fb01c0 STG.E.64 [R8.64], R6 5 164 164 131 1,048,576 33,554,432
14 00001514 a2fb01d0 EXIT 1 31 31 0 1,048,576 33,554,432
15 00001514 a2fb01e0 BRA 0x1514a2fb01e0 0 1,341 1,341 1,192
16 00001514 a2fb01f0 NOP 0 0
17 00001514 a2fb0200 NOP 0 0
18 00001514 a2fb0210 NOP 0 0
19 00001514 a2fb0220 NOP 0 0
20 00001514 a2fb0230 NOP 0 0
21 00001514 a2fb0240 NOP 0 0
22 00001514 a2fb0250 NOP 0 0
23 00001514 a2fb0260 NOP 0 0
24 00001514 a2fb0270 NOP 0 0

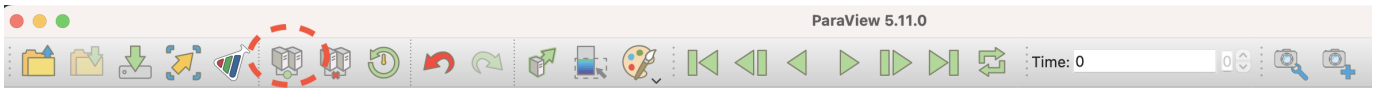
6.12 Visualization

6.12.1 Paraview on Polaris

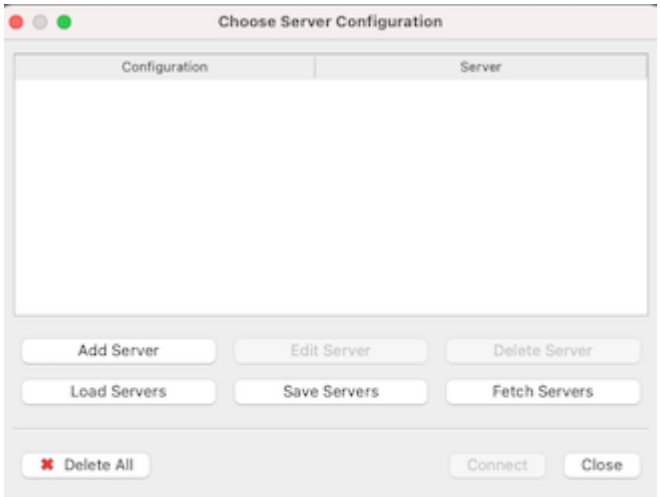
> **Note**
>
> At this time, we only support client/server mode where the user must manually launch the server on Polaris.

Setting up Paraview

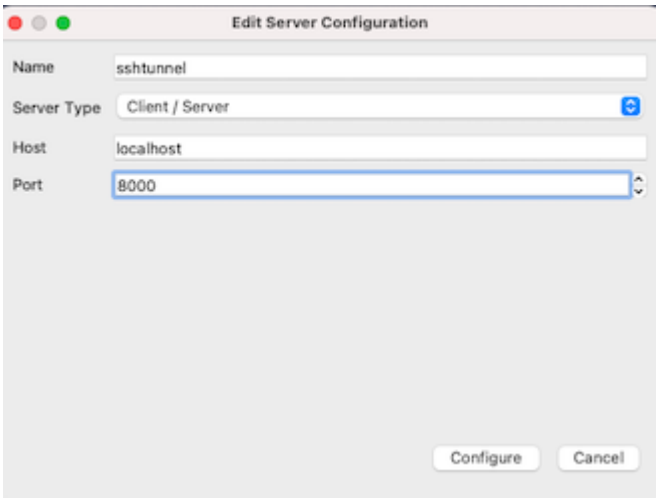
From your local client select Connect, either from the File menu, or by clicking on the icon circled below:



A new window will open where you can configure a server. Click on Add Server:



Give your server a name, select Client/Server, localhost, and a TCP port (8000 in this example)



Click "Configure". In the next window there is an option to set up how Paraview server will be launched, and the default is "Manual". Leave it on "Manual" and click "Save".

You will use these settings when establishing the connection.

Launching the Paraview server on Polaris

You can launch an interactive session on Polaris compute nodes with the following command (adjust parameters as needed to match your allocation, desired number of nodes, queue, walltime, and filesystems):

```
qsub -l walltime=01:00:00 -l select=2 -A yourallocation -q debug -I -l filesystems=home:grand
```

When the job starts you will receive a prompt on your head node like this:

```
username@x3005c0s7b0n0:~>
```

Make a note of the node hostname (`x3005c0s7b0n0` in the example above). You can also get this information from `qstat -fx jobID`

Now load the Paraview module

```
username@x3005c0s7b0n0:~> module load paraview

Lmod is automatically replacing "nvhpc/21.9" with "gcc/11.2.0".

-------------------------------------------------------------------------------
     Paraview v5.11.0 successfully loaded
-------------------------------------------------------------------------------
```

```
Due to MODULEPATH changes, the following have been reloaded:
  1) cray-mpich/8.1.16
```

and launch the Paraview server with

```
srizzi@x3005c0s7b0n0:~> mpirun -n 8 pvserver --server-port=8000
Waiting for client...
Connection URL: cs://x3005c0s7b0n0:8000
Accepting connection(s): x3005c0s7b0n0:8000
```

In this case `pvserver` will be listening on TCP port 8000 of your head node. You can change this port if you want.

Creating a tunnel over ssh

We need to establish an ssh tunnel to connect client to server. On your local machine open a new terminal and type:

```
ssh -v -N -L 8000:x3005c0s7b0n0:8000 polaris.alcf.anl.gov
```

where 8000 is a TCP port and `x3005c0s7b0n0` the name of your head node. Adjust these values accordingly.

Among multiple lines with debug information, you should see something like:

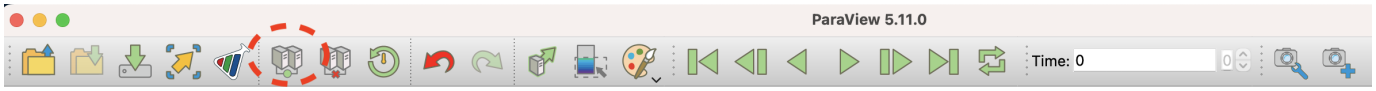
```
debug1: Local connections to LOCALHOST:8000 forwarded to remote address x3005c0s7b0n0:8000
```

Keep this terminal open for the duration of your session to keep the ssh tunnel active.

Now you are ready to launch your Paraview client locally. Keep in mind that client and servers versions must match. The Paraview version currently deployed on Polaris is 5.11.0

Connecting to Paraview server

Connect your Paraview client to the server configuration you created above. You can select Connect, either from the File menu, or the icon circled in the figure:



and selecting the configuration you created in a previous step.

The connection should point to:

```
localhost:8000
```

In the terminal where you launched the server you will see now that the connection is established. Note that Paraview may take a few seconds to connect. This is normal behavior.

```
username@x3005c0s7b0n0:~> mpirun -n 8 pvserver --server-port=8000
Waiting for client...
Connection URL: cs://x3005c0s7b0n0:8000
Accepting connection(s): x3005c0s7b0n0:8000
Client connected.
```

At this point you can use Paraview normally.

6.13 Workflows

6.13.1 Balsam

Balsam is a Python-based workflow manager that helps users execute large numbers of jobs, potentially with interjob dependencies, track job outcomes, and manage postprocessing analysis. A Balsam Site runs on a node with access to the job scheduler, where it can submit and monitor jobs. Overall job state is aggregated on the Balsam Server, making job data from all Sites accessible from any individual site (or the user's laptop), via the command-line interface or the Python API. To get information on how to use the command line tool, you can type `balsam --help` in your shell.

Full documentation for Balsam is available online.

Balsam requires Python 3.7+. To install Balsam on Polaris, first set up a virtual Python environment:

```
module load conda
conda activate base
python -m venv env
source env/bin/activate
pip install --upgrade pip
pip install --pre balsam
```

To use Balsam, users need an account on the Balsam server. Users can get an account by contacting the ALCF Help Desk. Once a user has an account, they can login and make a new site. A Balsam site is a project space for your workflow. You will be prompted to select what machine (Polaris) you are working on when creating a new site:

```
balsam login
balsam site init -n new-site new-site
cd new-site
balsam site start
```

See the Balsam documentation for full details.

6.13.2 Parsl on Polaris

> Parsl is a flexible and scalable parallel programming library for Python.
>
> -- *Parsl Documentation*

For many applications, managing an ensemble of jobs into a workflow is a critical step that can easily become a performance bottleneck. Many tools exist to address this, of which `parsl` is just one. On this page, we'll highlight some of the key pieces of information about `parsl` that are relevant to Polaris. `Parsl` is also extensively documented, has a dedicated Slack Channel, and a large community of users and developers beyond ALCF. We encourage you to engage with the `parsl` community for support with `parsl` specific questions, and for Polaris-specific questions or problems, please contact support@alcf.anl.gov.

Getting Parsl on Polaris

You can install parsl building off of the `conda` modules. You have some flexibility in how you want to extend the `conda` module to include parsl, but here is an example way to do it:

```
# Load the Conda Module (needed everytime you use parsl)
module load conda
conda activate

# Create a virtual env that uses the conda env as the system packages.
# Only do the next line on initial set up:
python -m venv --system-site-packages /path/to/your/virtualenv

# Load the virtual env (every time):
source /path/to/your/virtualenv/bin/activate

# Install parsl (only once)
pip install parsl
```

Using Parsl on Polaris

Parsl has a variety of possible configuration settings. As an example, we provide the configuration below that will run one task per GPU:

```
from parsl.config import Config

# PBSPro is the right provider for Polaris:
from parsl.providers import PBSProProvider
# The high throughput executor is for scaling to HPC systems:
from parsl.executors import HighThroughputExecutor
# You can use the MPI launcher, but may want the Gnu Parallel launcher, see below
```

```
from parsl.launchers import MpiExecLauncher, GnuParallelLauncher
# address_by_interface is needed for the HighThroughputExecutor:
from parsl.addresses import address_by_interface
# For checkpointing:
from parsl.utils import get_all_checkpoints

# Adjust your user-specific options here:
run_dir="/lus/grand/projects/yourproject/yourrundir/"

user_opts = {
    "worker_init":      f"source /path/to/your/virtualenv/bin/activate; cd {run_dir}", # load the environment where parsl is installed
    "scheduler_options":"#PBS -l filesystems=home:eagle:grand" , # specify any PBS options here, like filesystems
    "account":          "YOURPROJECT",
    "queue":            "debug-scaling",
    "walltime":         "1:00:00",
    "nodes_per_block":  3, # think of a block as one job on polaris, so to run on the main queues, set this >= 10
    "cpus_per_node":    32, # Up to 64 with multithreading
    "available_accelerators": 4, # Each Polaris node has 4 GPUs, setting this ensures one worker per GPU
    "cores_per_worker": 8, # this will set the number of cpu hardware threads per worker.
}

checkpoints = get_all_checkpoints(run_dir)
print("Found the following checkpoints: ", checkpoints)

config = Config(
        executors=[
            HighThroughputExecutor(
                label="htex",
                heartbeat_period=15,
                heartbeat_threshold=120,
                worker_debug=True,
                available_accelerators=user_opts["available_accelerators"], # if this is set, it will override other settings for max_workers if set
                cores_per_worker=user_opts["cores_per_worker"],
                address=address_by_interface("bond0"),
                cpu_affinity="block-reverse",
                prefetch_capacity=0,
                start_method="spawn",  # Needed to avoid interactions between MPI and os.fork
                provider=PBSProProvider(
                    launcher=MpiExecLauncher(bind_cmd="--cpu-bind", overrides="--depth=64 --ppn 1"),
                    # Which launcher to use?  Check out the note below for some details.  Try MPI first!
                    # launcher=GnuParallelLauncher(),
                    account=user_opts["account"],
                    queue=user_opts["queue"],
                    select_options="ngpus=4",
                    # PBS directives (header lines): for array jobs pass '-J' option
                    scheduler_options=user_opts["scheduler_options"],
                    # Command to be run before starting a worker, such as:
                    worker_init=user_opts["worker_init"],
                    # number of compute nodes allocated for each block
                    nodes_per_block=user_opts["nodes_per_block"],
                    init_blocks=1,
                    min_blocks=0,
                    max_blocks=1, # Can increase more to have more parallel jobs
                    cpus_per_node=user_opts["cpus_per_node"],
                    walltime=user_opts["walltime"]
                ),
            ),
        ],
        checkpoint_files = checkpoints,
        run_dir=run_dir,
        checkpoint_mode = 'task_exit',
        retries=2,
        app_cache=True,
)
```

Special notes for Polaris

On Polaris, there is a known bug where python applications launched with `mpi` and that use `fork` to spawn processes can sometimes have unexplaned hangs. For this reason, it is recommended to use `start_method="spawn"` on Polaris when using the `MpiExecLauncher` as is shown in the example config above. Alternatively, another solution is to use the `GNUParallelLauncher` which uses `GNU Parallel` to spawn processes. `GNU Parallel` can be loaded in your environment with the command `module`

`load gnu-parallel`. Both of these approaches will circumvent the hang issue from using `fork`.

Updates

For `parsl` versions after July 2023, the `address` passed in the `HighThroughputExecutor` needs to be set to `address = address_by_interface("bond0")`. With `parsl` versions prior to July 2023, it was recommended to use `address = address_by_hostname()` on Polaris, but with later versions this will not work on Polaris (or any other machine).

6.13.3 libEnsemble

libEnsemble is a Python toolkit for running dynamic ensembles of calculations. Users provide generator and simulator functions to express their ensembles, where the generator can steer the ensemble based on previous results. A library of example functions is available which can be modified as needed. These functions can submit external executables at any scale and in a portable way. System details are detected, and dynamic resource management is provided. libEnsemble can be used in a consistent manner on laptops, clusters, and supercomputers with minimal required dependencies.

Getting libEnsemble on Polaris

libEnsemble is provided on Polaris in the **conda** module:

```
module load conda
conda activate base
```

See the docs for more details on using python on Polaris.

ample: creating virtual environment and updating libEnsemble

E.g., to create a virtual environment that allows installation of further packages with pip:

```
python -m venv /path/to-venv --system-site-packages
. /path/to-venv/bin/activate
```

Where /path/to-venv can be anywhere you have write access. For future uses just load the conda module and run the activate line. You can also ensure you are using the latest version of libEnsemble:

```
pip install libensemble
```

libEnsemble examples

For a very simple example of using libEnsemble see the Simple Sine tutorial

For an example that runs a small ensemble using a C application (offloading work to the GPU), see the GPU app tutorial. The required files for the this tutorial can be found in this directory. Also, see the video demo.

Note that when initializing the MPIExecutor on Polaris (**run_libe_forces.py** in the example), you currently need to use the following options to pick up the correct MPI runner:

```
exctr = MPIExecutor(custom_info={'mpi_runner':'mpich', 'runner_name':'mpiexec'})
```

Job Submission

libEnsemble runs on the compute nodes on Polaris using either `multi-processing` or `mpi4py`. The user can set the number of workers for maximum concurrency. libEnsemble will detect the nodes available from the PBS environment and use these for running simulations. Polaris supports running multiple concurrent simulations on each node if desired,

A simple example batch script for a libEnsemble use case that runs four workers on one node:

```
#!/bin/bash -l
#PBS -l select=1:system=polaris
#PBS -l walltime=00:15:00
#PBS -l filesystems=home:grand
#PBS -q debug
#PBS -A <myproject>

export MPICH_GPU_SUPPORT_ENABLED=1
cd $PBS_O_WORKDIR
python run_libe_forces.py --comms local --nworkers 4
```

The script can be run with:

```
qsub submit_libe.sh
```

Or you can run an interactive session with:

```
qsub -A <myproject> -l select=1 -l walltime=15:00 -lfilesystems=home:grand -qdebug -I
```

Further links

Docs: https://libensemble.readthedocs.io
GitHub: https://github.com/Libensemble/libensemble