BedeDoc Documentation

Release

Mozhgan K. Chimeh

Mar 19, 2021

Contents

Bede is a supercomputer (otherwise known as an HPC system) run by the N8 group of research intensive universities in the north of England, on behalf of EPSRC, and hosted at Durham University.

The system has a specialist architecture, optimised for large memory problems on GPUs and for multi-node multi-gpu programs. This has a range of applications in Machine Learning and Imaging.

The site encourages user contributions via GitHub. Feel free to add items of concern to the Github issues section.

Please note that the system is still under active development, and so some functionality may temporarily break.

CHAPTER 1

Site Contents

1.1 Hardware

The system is based around the IBM POWER9 CPU and NVIDIA Tesla GPUs. Connectivity within a node is optimised by both the CPUs and GPUs being connected to an NVIDIA NVLink 2.0 bus, and outside of a node by a dual-rail Mellanox EDR InfiniBand interconnect allowing GPUDirect RDMA communications (direct memory transfers to/from GPU memory).

Together with IBM's software engineering, the POWER9 architecture is uniquely positioned for:

- Large memory GPU use, as the GPUs are able to access main system memory via POWER9's large model feature.
- Multi node GPU use, via IBM's Distributed Deep Learning (DDL) software.

There are:

- 2x "login" nodes, each containing:
  - 2x POWER9 CPUs @ 2.4GHz (40 cores total and 4 hardware threads per core), with NVLink 2.0
  - 512GB DDR4 RAM
  - 4x Tesla V100 32G NVLink 2.0
  - 1x Mellanox EDR (100Gbit/s) InfiniBand port
- 32x "gpu" nodes, each containing:
  - 2x POWER9 CPUs @ 2.7GHz (32 cores total and 4 hardware threads per core), with NVLink 2.0
  - 512GB DDR4 RAM
  - 4x Tesla V100 32G NVLink 2.0
  - 2x Mellanox EDR (100Gbit/s) InfiniBand ports
- 4x "infer" nodes, each containing:
  - 2x POWER9 CPUs @ 2.9GHz (40 cores total and 4 hardware threads per core)

  - 256GB DDR4 RAM
  - 4x Tesla T4 16G PCIe
  - 1x Mellanox EDR (100Gbit/s) InfiniBand port

The Mellanox EDR InfiniBand interconnect is organised in a 2:1 block fat tree topology. GPUDirect RDMA transfers are supported on the 32 "gpu" nodes only, as this requires an InfiniBand port per POWER9 CPU socket.

Storage is provided by a 2PB Lustre filesystem capable of reaching 10GB/s read or write performance, supplemented by an NFS service providing modest home and project directory needs.

1.2 Usage

Bede is running Red Hat Enterprise Linux 7 and access to its computational resources is mediated by the Slurm batch scheduler.

1.2.1 Registering

Access to the machine is based around projects:

- To register a new project:
  - Principle Investigators at an N8 institution should see the advice <here>
  - Principle Investigators at other institutions should see the advice <here>
- To create an account to use the system:
  - Identify an existing project, or register a new one.
  - Create an EPCC SAFE account and login to the SAFE system at: https://safe.epcc.ed.ac.uk/
  - Once there, select "Project->Request access" from the web interface and then register against your project

1.2.2 Login

Bede offers an SSH service running on host `bede.dur.ac.uk` (which fronts the two login nodes, `login1.bede.dur.ac.uk` and `login2.bede.dur.ac.uk`). SSH should be used for all interaction with the machine (including shell access and file transfer).

The login nodes are shared between all users of the service and therefore should only be used for light interactive work, for example: downloading and compiling software, editing files, preparing jobs and examining job output. Short test runs using their CPUs and GPUs are also acceptable.

Most of the computational power of the system is accessed through the batch scheduler, and so demanding applications should be submitted to it (see "Running Jobs").

1.2.3 File Storage

Each project has access to the following shared storage:

- Project home directory (`/projects/<project>`)
  - Intended for project files to be backed up (note: backups not currently in place)
  - Modest performance

  - A default quota of 20GB
- Project Lustre directory (`/nobackup/projects/<project>`)
  - Intended for bulk project files not requiring backup
  - Fast performance
  - No quota limitations

By default, files created within a project area are readable and writable by all other members of that project.

In addition, each user has:

- Home directory (`/users/<user>`)
  - Intended for per-user configuration files.
  - Modest performance
  - A default quota of 20GB

Please note that, as access to Bede is driven by project use, no personal data should be stored on the system.

Current utilisation and limits of a user's home directory can be found by running the `quota` command. Similar information can be found for the project home directory using the `df -h /projects/<project>` command.

To examine how much space is occupied by a project's Lustre directory, a command of the form `du -csh /nobackup/projects/<project>` is required. As `du` will check each and every file under the specified directory, this may take a long time to complete. We plan to develop the service and provide this information in a more responsive format in the future.

1.2.4 Running Jobs

Access beyond the two login node systems should only be done through the Slurm batch scheduler, by packaging your work into units called jobs.

A job consists of a shell script, called a job submission script, containing the commands that the job will run in sequence. In addition, some specially formatted comment lines are added to the file, describing how much time and resources the job needs.

Resources are requested in terms of the type of node, the number of GPUs per node (for each GPU requested, the job receives 25% of the node's CPUs and RAM) and the number of nodes required.

There are a number of example job submission scripts below.

Requesting resources

Part of, or an entire node

Example job script for programs written to take advantage of a GPU or multiple GPUs on a single computer:

```
#!/bin/bash

# Generic options:

#SBATCH --account=<project>  # Run job under project <project>
#SBATCH --time=1:0:0         # Run for a max of 1 hour

# Node resources:
# (choose between 1-4 gpus per node)
```

```
#SBATCH --partition=gpu     # Choose either "gpu" or "infer" node type
#SBATCH --nodes=1           # Resources from a single node
#SBATCH --gres=gpu:1        # One GPU per node (plus 25% of node CPU and RAM per GPU)

# Run commands:

nvidia-smi  # Display available gpu resources

# Place other commands here

echo "end of job"
```

Multiple nodes (MPI)

Example job script for programs using MPI to take advantage of multiple CPUs/GPUs across one or more machines:

```
#!/bin/bash

# Generic options:

#SBATCH --account=<project>  # Run job under project <project>
#SBATCH --time=1:0:0         # Run for a max of 1 hour

# Node resources:

#SBATCH --partition=gpu     # Choose either "gpu" or "infer" node type
#SBATCH --nodes=2           # Resources from a two nodes
#SBATCH --gres=gpu:4        # Four GPUs per node (plus 100% of node CPU and RAM per␣
↪node)

# Run commands:

bede-mpirun --bede-par 1ppc <mpi_program>

echo "end of job"
```

The `bede-mpirun` command takes both ordinary `mpirun` arguments and the special `--bede-par <distrib>` option, allowing control over how MPI jobs launch, e.g. one MPI rank per CPU core or GPU.

The formal specification of the option is: `--bede-par <rank_distrib>[:<thread_distrib>]` and it defaults to `1ppc:1tpt`

Where `<rank_distrib>` can take `1ppn` (one process per node), `1ppg` (one process per GPU), `1ppc` (one process per CPU core) or `1ppt` (one process per CPU thread).

And `<thread_distrib>` can take `1tpc` (set `OMP_NUM_THREADS` to the number of cores available to each process), `1tpt` (set `OMP_NUM_THREADS` to the number of hardware threads available to each process) or `none` (set `OMP_NUM_THREADS=1`)

Examples:

```
# - One MPI rank per node:
bede-mpirun --bede-par 1ppn <mpirun_options> <program>

# - One MPI rank per gpu:
bede-mpirun --bede-par 1ppg <mpirun_options> <program>
```

```
# - One MPI rank per core:
bede-mpirun --bede-par 1ppc <mpirun_options> <program>

# - One MPI rank per hwthread:
bede-mpirun --bede-par 1ppt <mpirun_options> <program>
```

Multiple nodes (IBM PowerAI DDL)

IBM PowerAI DDL (Distributed Deep Learning) is a method of using the GPUs in more than one node to perform calculations. Example job script:

```
#!/bin/bash

# Generic options:

#SBATCH --account=<project>  # Run job under project <project>
#SBATCH --time=1:0:0         # Run for a max of 1 hour

# Node resources:

#SBATCH --partition=gpu    # Choose either "gpu" or "infer" node type
#SBATCH --nodes=2          # Resources from a two nodes
#SBATCH --gres=gpu:4       # Four GPUs per node (plus 100% of node CPU and RAM per␣
↪node)

# Run commands:

# (assume IBM Watson Machine Learning Community Edition is installed
# in conda environment "wmlce")

conda activate wmlce

bede-ddlrun python $CONDA_PREFIX/ddl-tensorflow/examples/keras/mnist-tf-keras-adv.py

echo "end of job"
```

1.3 Software

1.3.1 Environments

The default software environment on Bede is called “builder”. This uses the modules system normally used on HPC systems, but provides a system of intelligent modules. To see a list of what is available, executing the command `module avail`.

In this scheme, modules providing access to compilers and libraries examine other modules that are also loaded and make the most appropriate copy (or “flavour”) of the software available. This minimises the problem of knowing what modules to choose whilst providing access to all the combinations of how a library can be built.

For example, the following command gives you access to a copy of FFTW 3.3.8 that has been built against GCC 8.4.0:

```
$ module load gcc/8.4.0 fftw/3.3.8
$ which fftw-wisdom
/opt/software/builder/developers/libraries/fftw/3.3.8/1/gcc-8.4.0/bin/fftw-wisdom
```

If you then load an MPI library, your environment will be automatically updated to point at a copy of FFTW 3.3.8 that has been built against GCC 8.4.0 and OpenMPI 4.0.5:

```
$ module load openmpi/4.0.5
$ which fftw-wisdom
/opt/software/builder/developers/libraries/fftw/3.3.8/1/gcc-8.4.0-openmpi-4.0.5/bin/
↪fftw-wisdom
```

Similarly, if you then load CUDA, the MPI library will be replaced by one built against it:

```
$ which mpirun
/opt/software/builder/developers/libraries/openmpi/4.0.5/1/gcc-8.4.0/bin/mpirun
$ module load cuda/10.2.89
$ which mpirun
/opt/software/builder/developers/libraries/openmpi/4.0.5/1/gcc-8.4.0-cuda-10.2.89/bin/
↪mpirun
```

Modules follow certain conventions:

- Logs of software builds can be found under `/opt/software/builder/logs/`.
- Installation recipes for modules can be found under directory `/home/builder/builder/`.
- Although modules do their best to configure your environment so that you can use the software, it is sometimes useful to know where the software is installed on disk. This is provided by the `<NAME>_HOME` environment variable, e.g. if the `gcc/8.4.0` module is loaded, environment variable `GCC_HOME` points to the directory containing its files.
- Software provided by modules sometimes use other modules for their functionality. It is not normally required to explicitly load these prerequisites but it can be useful, for example to mirror R's buld environment when installing an R library. Where this occurs, a list of modules is provided by the `<NAME>_BUILD_MODULES` environment variable, e.g. the `r` module sets environment variable `R_BUILD_MODULES`.

Software can be built on top of these modules in the following ways:

- Traditional - loading appropriate modules, manually unpacking, configuring, building and installing the new software (e.g. `./configure; make; make install`)
- Spack - automated method of installing software. Spack will automatically find the multiple flavours (or variants, in spack-speak) of libraries provided by builder, minimising the number of packages needing to be built.

With Builder and Spack, the opportunity arises for a project to inherit and supplement software, and for users to then inherit and supplement that in turn. In this way, the centre can concentrate on providing core software of general use and allow projects and users to concentrate on specialist software elements that support their work.

In addition, there are two other types of software environment on Bede, which are not currently recommended:

- The vendor-supplied set of modules that originally came with the machine. To use these, execute: `echo ocf > ~/.application_environment` and then login again.
- Easybuild - an automated method of installing software, rival to Spack. To use this, execute: `echo builder > ~/.application_environment` and then login again.

In both cases, executing `rm ~/.application_environment` and login again will return you to the default software environment.

Spack

Spack can be used to extend the installed software on the system, without requiring specialist knowledge on how to build particular pieces of software. Documentation for the project is here: https://spack.readthedocs.io/

To install spack, execute the following and then login again:

```
$ git clone https://github.com/spack/spack.git $HOME/spack

$ echo 'export SPACK_ROOT=$HOME/spack' >> ~/.bash_profile
$ echo 'source $SPACK_ROOT/share/spack/setup-env.sh' >> ~/.bash_profile
```

Example usage, installing an MPI aware, GPU version of gromacs and than loading it into your environment to use (once built, execute `spack load gromacs` before using):

```
$ spack install gromacs +mpi +cuda
```

Other useful spack commands: * `spack find` - show what packages have been installed * `spack list` - show what packages spack knows how to build * `spack compilers` - show what compilers spack can use * `spack info <package>` - details about a package, and the different ways it can be built * `spack spec <package>` - what pieces of software a package depends on

If a project wishes to create a spack installation, for example under `/projects/<project>/spack` and you would like an easy way for your users to add it to their environment, please contact us and we can make a module.

If you are a user who wishes to supplement your project's spack installation, follow the installation instructions above and then tell it where your project's copy of spack is:

```
cat > $SPACK_ROOT/etc/spack/upstreams.yaml <<EOF
upstreams:
  spack-central:
    install_tree: /projects/<project>/spack
    modules:
      tcl: /projects/<project>/spack/share/spack/modules
EOF
```

Easybuild

Not currently recommended.

The central Easybuild modules are available when a user executes the following command and then logs in again:

```
echo easybuild > ~/.application_environment
```

A user can create their own Easybuild installation to supplement (or override) the packages provided by the central install by:

```
echo 'export EASYBUILD_INSTALLPATH=$HOME/eb' >> ~/.bash_profile
echo 'export EASYBUILD_BUILDPATH=/tmp' >> ~/.bash_profile
echo 'export EASYBUILD_MODULES_TOOL=Lmod' >> ~/.bash_profile
echo 'export EASYBUILD_PARALLEL=8' >> ~/.bash_profile
echo 'export MODULEPATH=$HOME/eb/modules/all:$MODULEPATH' >> ~/.bash_profile
```

Login again, and then:

```
wget https://raw.githubusercontent.com/easybuilders/easybuild-framework/develop/
↪easybuild/scripts/bootstrap_eb.py
python bootstrap_eb.py $EASYBUILD_INSTALLPATH
```

Verify install by checking sensible output from:

```
module avail   # should show an EasyBuild module under user's home directory
module load EasyBuild
which eb       # should show a path under the user's home directory
```

Software can now be installed into the new Easybuild area using `eb <package>`

Project Easybuild installations can be created using a similar method. In this case, a central module to add the project's modules to a user's environment is helpful, and can be done on request.

1.3.2 Compilers

All compiler modules set the `CC`, `CXX`, `FC`, `F90` environment variables to appropriate values. These are commonly used by tools such as cmake and autoconf, so that by loading a compiler module its compilers are used by default.

This can also be done in your own build scripts and make files. e.g.

```
module load gcc
$CC -o myprog myprog.c
```

GCC

Note that the default GCC provided by Red Hat Enterprise Linux 7 (4.8.5) is quite old, will not optimise for the POWER9 processor (either use POWER8 tuning options or use a later compiler), and does not have CUDA/GPU offload support compiled in. The module `gcc/native` has been provided to point to this copy of GCC.

The copies of GCC available as modules have been compiled with CUDA offload support:

```
module load gcc/10.2.0
```

LLVM

LLVM has been provided for use on the system by the `llvm` module. It has been built with CUDA GPU offloading support, allowing OpenMP regions to run on a GPU using the `target` directive.

Note that, as from LLVM 11.0.0, it provides a Fortran compiler called `flang`. Although this has been compiled and can be used for experimentation, it is still immature and ultimately relies on `gfortran` for its code generation. The `lvm/11.0.0` module therefore defaults to using the operating system provided `gfortran`, instead.

1.3.3 BLAS/LAPACK

The following numerical libraries provide optimised CPU implementations of BLAS and LAPACK on the system:

- ESSL (IBM Engineering and Scientific Subroutine Library)
- OpenBLAS

The modules for each of these libraries provide some convenience environment variables: `N8CIR_LINALG_CFLAGS` contains the compiler arguments to link BLAS and LAPACK to C code; `N8CIR_LINALG_FFLAGS` contains the same to link to Fortran. When used with variables such as `CC`, commands to build software can become entirely independent of what compilers and numerical libraries you have loaded, e.g.

```
module load gcc essl/6.2
$CC -o myprog myprog.c $N8CIR_LINALG_CFLAGS
```

1.3.4 MPI

The main supported MPI on the system is OpenMPI.

For access to a cuda-enabled MPI: `module load gcc cuda openmpi`

We commit to the following convention for all MPIs we provide as modules:

- The wrapper to compile C programs is called `mpicc`
- The wrapper to compile C++ programs is called `mpicxx`
- The wrapper to compile Fortran programs is called `mpif90`

1.3.5 HDF5

When loaded in conjunction with an MPI module such as `openmpi`, the `hdf5` module provides both the serial and parallel versions of the library. The parallel functionality relies on a technology called MPI-IO, which is currently subject to the following known issue on Bede:

- HDF5 does not pass all of its parallel tests with OpenMPI 4.x. If you are using this MPI and your application continues to run but does not return from a call to the HDF5 library, you may have hit a similar issue. The current workaround is to instruct OpenMPI to use an alternative MPI-IO implementation with the command: `export OMPI_MCA_io=ompio` The trade off is that, in some areas, this alternative is extremely slow and so should be used with caution.

1.3.6 NetCDF

The `netcdf` module provides the C, C++ and Fortran bindings for this file format library. When an MPI module is loaded, parallel support is enabled through the PnetCDF and HDF5 libraries.

Use of NetCDF's parallel functionality can use HDF5, and so is subject to its known issues on Bede (see above).

1.3.7 PyTorch Quickstart

The following should get you set up with a working conda environment (replacing <project> with your project code):

```
rm -rf ~/.conda .condarc
export DIR=/nobackup/projects/<project>/$USER
rm -rf $DIR
mkdir $DIR
pushd $DIR

wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-ppc64le.sh

sh Miniconda3-latest-Linux-ppc64le.sh -b -p $DIR/miniconda
source miniconda/bin/activate
conda update conda -y
conda config --set channel_priority strict

conda config --prepend channels \
```

```
        https://public.dhe.ibm.com/ibmdl/export/pub/software/server/ibm-ai/conda/

conda config --prepend channels \
        https://opence.mit.edu

conda create --name opence pytorch=1.7.1 -y
conda activate opence
```

This has some limitations such as not supporting large model support. If you require this you can try the instructions below, these provide an older version of PyTorch however.

1.3.8 PyTorch and TensorFlow: IBM PowerAI and Watson Machine Learning Community Edition (wmlce) [Possibly Out of Date]

IBM have done a lot of work to port common Machine Learning tools to the POWER9 system, and to take advantage of the GPUs abililty to directly access main system memory on the POWER9 architecture using its "Large Model Support".

This has been packaged up into what is variously known as IBM Watson Machine Learning Community Edition (wmlce) or the catchier name PowerAI.

Documentation on wmlce can be found here: https://www.ibm.com/support/pages/get-started-ibm-wml-ce

Installation is via the IBM channel of the anaconda package management tool. **Note: if you do not use this channel you will not find all of the available packages.** First install anaconda (can be quite large - so using the /nobackup area):

```
cd /nobackup/projects/<project>

wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-ppc64le.sh
sh Miniconda3-latest-Linux-ppc64le.sh
conda update conda
conda config --set channel_priority strict
conda config --prepend channels https://public.dhe.ibm.com/ibmdl/export/pub/software/
↪server/ibm-ai/conda/
conda create --name wmlce
```

Then login again and install wmlce (GPU version by default - substitute `powerai-cpu` for `powerai` for the CPU version):

```
conda activate wmlce
conda install powerai ipython
```

Running `ipython` on the login node will then allow you to experiment with this feature using an interactive copy of Python and the GPUs on the login node. Demanding work should be packaged into a job and launched with the `python` command.

If a single node with 4 GPUs and 512GB RAM isn't enough, the Distributed Deep Learning feature of PowerAI should allow you to write code that can take advantage of multiple nodes.

1.4 Profiling

1.4.1 NVIDIA Profiling Tools

HPC systems typically favour batch jobs rather than interactive jobs for improved utilsation of resources. The Nvidia profiling tools can all be used to capture all required via the command line, which can then be interrogated using the GUI tools locally.

Nsight Systems and Nsight Compute are the modern Nvidia profiling tools, introduced with CUDA 10.0 supporting Pascal+ and Volta+ respectivley.

The NVIDIA Visual Profiler is the legacy profiling tool, with full support for GPUs up to pascal (SM < 75), partial support for Turing (SM 75 and no support for Ampere (SM80).

Compiler settings for profiling

Applications compiled with `nvcc` should pass `-lineinfo` (or `--generate-line-info`) to include source-level profile information.

Additionally, NVIDIA Tools Extension SDK can be used to enhance these profiling tools.

Nsight Systems and Nsight Compute

Note:

- Nsight Systems supports Pascal and above (SM 60+)
- Nsight Compute supports Volta and aboce (SM 70+)

Generate an application timeline with Nsight Systems CLI (`nsys`):

```
nsys profile -o timeline ./myapplication
```

Use the `--trace` argument to specify which APIs should be traced. See the nsys profiling command switch options for further information.

```
nsys profile -o timeline --trace cuda,nvtx,osrt,openacc ./myapplication <arguments>
```

Note: On Bede (Power9) the `--trace` option `osrt` can lead to `SIGILL` errors. As this is a default, consider passing `--trace cuda,nvtx` as an alternative minimum.

Once this file has been downloaded to your local machine, it can be opened in `nsys-ui`/`nsight-sys` via `File > Open > timeline.qdrep`:

Fine-grained kernel profile information can be captured using remote Nsight Compute CLI (`ncu`/`nv-nsight-cu-cli`):

```
ncu -o profile --set full ./myapplication <arguments>
```

Note: `ncu` is available since CUDA v11.0.194, and Nsight Compute v2020.1.1. For older versions of CUDA use `nv-nsight-cu-cli` (if Nsight Compute is installed).

This will capture the full set of available metrics, to populate all sections of the Nsight Compute GUI, however this can lead to very long run times to capture all the information.

For long running applications, it may be favourable to capture a smaller set of metrics using the `--set`, `--section` and `--metrics` flags as described in the Nsight Comptue Profile Command Line Options table.

The scope of the section being profiled can also be reduced using NVTX Filtering; or by targetting specific kernels using `--kernel-id`, `--kernel-regex` and/or `--launch-skip` see the CLI docs for more information).

Once the `.ncu-rep` file has been downloaded locally, it can be imported into local Nsight CUDA GUI `ncu-ui`/`nv-nsight-cu` via:

```
ncu-ui profile.ncu-rep
```

Or `File > Open > profile.ncu-rep`, **or** Drag `profile.ncu-rep` into the `nv-nsight-cu` window.

Note: Older versions of Nsight Compute (CUDA < v11.0.194) used `nv-nsight-cu` rather than `ncu-ui`.

Note: Older versions of Nsight Compute generated `.nsight-cuprof-report` files, instead of `.ncu-rep` files.

More info

- Nsight Systems
- Nsight Compute
- OLCF: Nsight Systems Tutorial
- OLCF: Nsight Compute Tutorial

Use the following Nsight report files to follow the tutorial.

Cluster Modules

- `module load nvidia/20.5`

Visual Profiler (legacy)

Note:

- Nvprof does not support CUDA kernel profiling for Turing GPUs (SM75)
- Nvprof does not support Ampere GPUs (SM80+)

Application timelines can be generated using `nvprof`:

```
nvprof -o timeline.nvprof ./myapplication
```

Fine-grained kernel profile information can be genereted remotely using `nvprof`:

```
nvprof --analysis-metrics -o analysis.nvprof ./myapplication
```

This captuires the full set of metrics required to complete the guided analysis, and may take a (very long) while. Large applications request fewer metrics (via `--metrics`), fewer events (via `--events`) or target specific kernels (via `--kernels`). See the nvprof command line options for further information.

Once these files are downloaded to your local machine, Import them into the Visual Profiler GUI (`nvvp`)

- `File > Import`
- Select `Nvprof`
- Select `Single process`
- Select `timeline.nvvp` for `Timeline data file`
- Add `analysis.nvprof` to `Event/Metric data files`

Documentation

- Nvprof Documentation

Cluster Modules

- `module load cuda/10.1`
- `module load cuda/10.2`
- `module load nvidia/20.5`

1.4.2 NVIDIA Tools Extension

NVIDIA Tools Extension (NVTX) is a C-based API for annotating events and ranges in applications. These markers and ranges can be used to increase the usability of the NVIDIA profiling tools.

- For CUDA `>= 10.0`, NVTX version 3 is distributed as a header only library.
- For CUDA `< 10.0`, NVTX is distributed as a shared library.

The location of the headers and shared libraries may vary between Operating Systems, and CUDA installation (i.e. CUDA toolkit, PGI compilers or HPC SDK).

The NVIDIA Developer blog contains several posts on using NVTX:

- Generate Custom Application Profile Timelines with NVTX (Jiri Kraus)
- Track MPI Calls In The NVIDIA Visual Profiler (Jeff Larkin)
- Customize CUDA Fortran Profiling with NVTX (Massimiliano Fatica)

Custom CMake `find_package` modules can be written to enable use within Cmake e.g. ptheywood/cuda-cmake-NVTX on GitHub

Documentation

- NVTX Documentation
- NVTX 3 on GitHub

1.5 Useful Training Material

The following list is a useful catalogue of training material.

A list of useful training material is also available on the gpuhackathons site:

- GPUHackathon Resources webapge

1.5.1 Profiling Material

- OLCF: Nsight Systems Tutorial
- OLCF: Nsight Compute Tutorial

Use the following Nsight report files to follow the tutorial.

1.5.2 General Training Material

- OpenACC Online Course (NVIDIA)
- NVIDIA OpenACC Advanced Training Material Container
- CUDA training at Oakridge (slides and lecture recording)
- Sheffield One or Two Day Introduction to CUDA Course (slides and labs)
- Sheffield Parallel Computing with GPUs taught module (slides, labs, lecture recordings)