Cabeus, named after a lunar crater in the south polar region of the Moon, is a cluster system that supports HPC and machine learning/artificial intelligence (ML/AI) applications that can benefit from the GPU technology.

GPU Hardware Summary

The node statistics are shown in two groups: nodes based on the x86_64 operating system (OS), and those based on the aarch64 OS.

Cabeus x86_64 GPU Nodes

- 12 racks
- 128 nodes
- 8,192 CPU cores + 1,769,472 double-precision GPU cores
- Total memory: 106 terabytes (65 TB from CPU host memory + 41 TB from GPU memory)
- Theoretical-double precision peak performance: 5.31 Petaflop/s (0.32 Pflop/s from CPUs + 4.99 Pflop/s from GPUs)

Cabeus aarch64 GPU Nodes

- 23 racks
- 538 nodes
- 38,736 CPU cores + 4,545,024 double-precision GPU cores
- Total memory: 258,240 GB (CPU) + 51,648 GB (GPU)
- Theoretical-double precision peak performance: 2,147 Teraflops (CPU) + 17,998 Tflops (GPU)

GPU Nodes by Model

Each GPU compute node includes one CPU socket as the host and one or four GPU cards as the coprocessors. These compute nodes are grouped by the CPU processor models as listed below:

- **model=mil_a100**

  128 nodes total. Each node contains one AMD Milan CPU socket and four NVIDIA Amper A100 GPU cards, where the CPU and GPUs are connected via PCI Express 4.0 bus.

  The Milan CPU socket used in the `mil_a100` nodes are AMD EPYC 7763.

- **model=gh200**

  538 nodes total. Each node contains one NVIDIA GH200 Grace Hopper Superchip, which consists of one Grace CPU (Neoverse V2 core with ARM v9.0-A architecture) and one H100 GPU, connected by a high-bandwidth, memory-coherent NVLink-C2C interconnect.

Cabeus Processor, Memory, and Network Subsystems Statistics

	Node Details	
	mil_a100	**gh200**
Architecture	Apollo 6500 Gen10 Plus	Supermicro SuperServer ARS-111GL-NHR
Total # of Nodes	128	538
Host name	cb[01-05,08-09]n[01-12], cb[06-07]n[01-10], cb[10-12]n[01-08]	cgh[1-3,6-15,17-19,21-23]n[01-26], cgh[4,20]n[01-12], cgh16n[01-20]
	CPU Host	
Processors Model	64-core EPYC 7763	72-core Grace (Neoverse V2)
# of Physical CPU Cores/Node	64	72
CPU-Clock	2.45 GHz	3.465 GHz
Maximum Double Precision Floating Point Operations per cycle per core	16	16
Total CPU Double Precision Flops/Node	2,508 GFlops	3.99 TFlops
Host Memory/Node	512 GB (8 x 64 GB DDR4)	480 GB (LPDDR5X)
	GPU	
GPU Architecture	Ampere	Hopper
Compute Capability**	cc80	cc90
Device Name	NVIDIA A100-SXM4-80GB	NVIDIA GH200 480GB

Clock Rate (Base/Boost)	1,275 MHz/1,410 MHz	(n/a)/1,980 MHz
# of coprocessors/Node	4	1
# of Streaming Multiprocessors (SMs)/coprocessor	108	132
# of Single Precision CUDA Cores/coprocessor	6,912	16,896
# of Double Precision CUDA Cores/coprocessor	3,456	8,448
# of Tensor Cores/coprocessor	432	528
Maximum double precision floating point operations per cycle per CUDA core	2	2
Total GPU Double Precision Flops/Coprocessor (Base/Boost) using Double Precision CUDA Cores	8.8 TFlops /9.7 TFlops	33.4 TFlops
Total GPU Double Precision Flops/Node (Base/Boost) using Double Precision CUDA Cores	35.2 TFlops/38.8 TFlops	33.4 TFlops
L2 Cache	40 MB	60 MB
Global Memory/coprocessor	80 GB (HBM2e)	96 GB (HBM3)
Memory Clock Rate	1,593 MHz	2,619 MHz
Memory Bus Width	5,120 bits	6,144 bits
Memory Bandwidth	2,039 GB/s (if using SXM form factor; 1,935 GB/s if using PCI form factor)	4,000 GB/s
GPU-to-GPU Communication	with NVLink 3.0: 600 GB/s	N/A
CPU-to-GPU Communication	PCIe 4.0 with 31.5 GB/s bandwidth for each direction	NVLink-C2C with 900 GB/s bi-directional
	Inter-node Network	
IB Device on Node	1 HDR 200 Gb 2-port card + 1 GbE 4-port card	NDR 200 Gb 1-port
	Local Disk	
SSD/node***	raw (2 x 1.92 TB SSD drives + 2 x 3.84 TB NVMe SSD) or (6 x 2.93 TB NVMe SSD)	960 GB Samsung NVMe M.2

** The compute capability identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU. Do not confuse the compute capability number with the CUDA version.

*** Not yet available for public usage, pending configuration decisions.

For more hardware details, access a GPU node through a PBS session and run one of the following commands:

For CPU info:

```
cat /proc/cpuinfo
```

For host memory info:

```
cat /proc/meminfo
```

Note: For the gh200 nodes with the unified memory feature,
the total memory shown in the output of /proc/meminfo is the sum
of the CPU memory and the GPU memory.

For GPU info:

```
/usr/bin/nvidia-smi -q
```

or load a nvhpc module and run the command pgaccelinfo or nvaccelinfo.

For example, for systems with x86_64 architecture:

```
module avail nvhpc
---------- /nasa/modulefiles/toss4 ----------

nvhpc-nompi/22.3 nvhpc-nompi/23.11 nvhpc/23.5 nvhpc/24.1
nvhpc-nompi/23.5 nvhpc-nompi/24.1 nvhpc/23.7 nvhpc/openmpi-3.1.5
nvhpc-nompi/23.7 nvhpc/22.3 nvhpc/23.11 nvhpc/openmpi-4.0.5

module load nvhpc/23.11
```

For example, for systems with aarch64 architecture:

module avail nvhpc
module load nvhpc/24.11

pgaccelinfo
or
nvaccelinfo

Cabeus includes 538 GPU nodes based on the NVIDIA GH200 Grace Hopper Superchip.

To help you prepare for running jobs on the Grace Hopper GPU nodes, this short user guide includes information on the general configuration of the nodes, compiling your code, and running PBS jobs.

Overview of Cabeus Nodes with NVIDIA Grace Hopper Superchip

The nodes are accessed via two front ends:

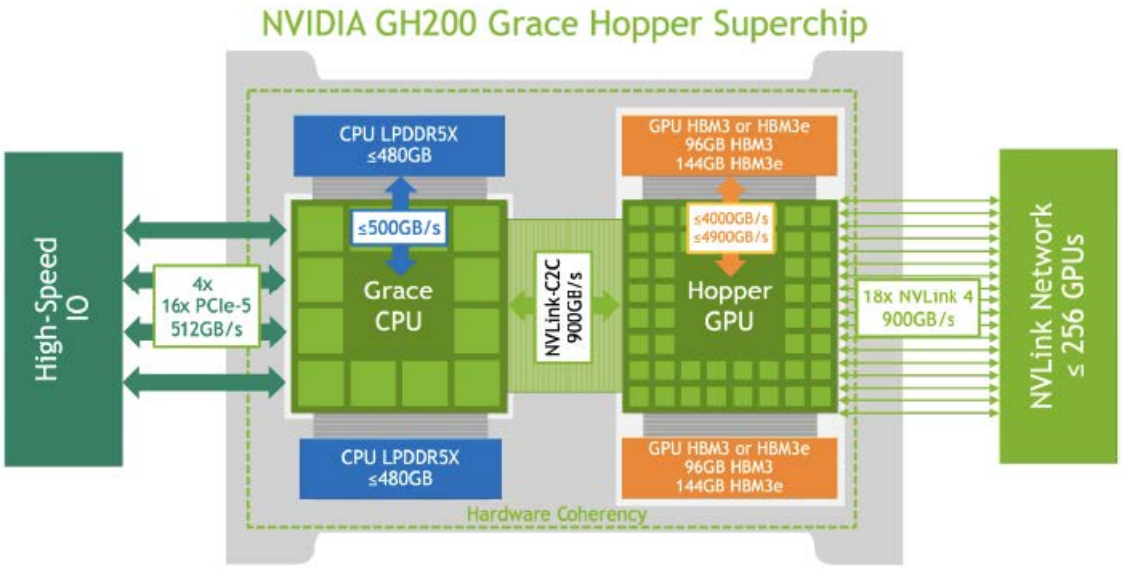
- cghfe01
- cghfe02

The hostnames for the GH200 compute nodes are as follows:

- cgh[1-3,6-15,17-19,21-23]n[01-26]
- cgh[04,20]n[01-12]
- cgh16n[01-20]

The PBS resource model name is `gh200`.

Each of the front ends and compute nodes contains a single GH200 superchip, which is shown in this diagram published by NVIDIA:



As shown in the diagram, the GH200 superchip includes:

- One Grace CPU containing 72 Arm Neoverse V2 physical cores (maximum 3.465 GHz on the front ends; slight variation on compute nodes but should be all above 3.4 GHz; use the `lscpu` command to check) and up to 480 GB Low-Power Double Data Rate 5X (LPDDR5X) host memory with a memory bandwidth of up to 500 GB/s.

  Note: The memory bandwidth varies with memory capacity: 384 GB/s with a capacity of 480 GB as in all NAS GH200 nodes, and 512 GB/s with a capacity of 120 GB or 240 GB.

  The Arm Neoverse V2 CPU uses the Armv9.0-A architecture which comes with four 128-bit Single Instruction Multiple Data (SIMD) units that support Scalable Vector Extension 2 (SVE2) instructions.
- One H100 (Hopper) GPU containing 132 Streaming Multiprocessors (SMs) with a clock rate of 1.980 GHz and 96 GB High Bandwidth Memory 3 (HBM3) device memory with a memory bandwidth of up to 4000 GB/s. Each SM contains 128 single precision CUDA cores (total is 16,896 per H100 GPU), 64 double precision CUDA cores (total is 8,448 per H100 GPU), and 4 Tensor cores (total is 528 per H100 GPU).
- A high-bandwidth, memory-coherent NVIDIA NVLink-C2C (chip-2-chip) interconnect that connects the CPU and GPU with a bidirectional bandwidth of 900 GB/s.

  NVLink-C2C enables applications to oversubscribe the Hopper GPU's memory and directly utilize the Grace CPU's memory at high bandwidth. The Grace CPU can also directly access the GPU memory. The memory coherency enables programming with a unified programming model.
- 4x (4 links) PCIe 5.0 x16 (16 lanes) with a total bi-directional bandwidth of 512 GB/s.

  The GH200 nodes at NAS use the NVIDIA MGX platform design where the nodes are interconnected with PCIe and host channel adapter (HCA) on each node and NVIDIA's internode InfiniBand networking solution. Specifically, InfiniBand Next-Generation Data Rate (NDR) HCAs, switches, and cables are utilized in a fat tree topology.

  Note: The 18x NVLink 4 network shown on the right side of the diagram above does not apply to the NVIDIA MGX platform. It is used in the NVIDIA DGX platform for connecting up to 256 GPUs together.

For more information about the Grace Hopper superchip, see NVIDIA's GH200 architecture whitepaper.

Requesting GPU Allocations

A Standard Billing Unit (SBU) allocation specifically for the GPUs is required to use all GPU resources, including the GH200 compute nodes. The SBU rate for the GH200 nodes is 18.34.

Accessing the Cabeus Grace Hopper Front Ends and Compute Nodes

The GH200 front end nodes (cghfe01 and cghfe02) are for building, testing your software dependencies, and running short tests of your executables.

Note: Termination of long-running CPU and GPU processes may be enforced in the future.

You can access the GH200 front ends from your local desktop using one of the following methods:

One-Step Connection With SSH passthrough

If you normally connect to NAS front-ends using SSH passthrough, you can modify your existing .ssh/config file on your local host to add cghfe01 and cghfe02. If you have been using a configuration file based on the NAS config template, you can simply replace this line:

```
Host *.nas.nasa.gov lou lou? ?fe ?fe? ?fe?? *-last pbspl?
```

with the following line:

```
Host *.nas.nasa.gov lou lou? *fe* *-last pbs*
```

where *fe* matches any NAS front-end system. Then, run:

```
your_localhost% ssh cghfe01 (or ssh cghfe02)
```

Two-Step Connection Without SSH Passthrough

To access the front ends without SSH passthrough, run:

```
your_localhost% ssh sfe6 (or ssh sfe7, ssh sfe8)
sfe% ssh cghfe01  (or ssh cghfe02)
Password: xxxxxxxxxxx
cghfe01%
```

The gh200 compute nodes are for running long testing or production jobs. They are accessible only through PBS jobs submitted from the cghfe01 or cghfe02 front ends. The PBS server for the GH200 nodes is pbs05a (users are not allowed to SSH into pbs05a).

You can access your filesystems from the GH200 front-ends and compute nodes, including: $HOME, Lustre /nobackup, /nobackupnfs1, /nasa, /nasa2, and /vast_swbuild.

Operating Systems

The operating system on all GH200-based front ends and compute nodes is TOSS 5 with Red Hat Enterprise Linux 9.8. The Linux kernel is Release 5.x for the Arm architecture 64-bit execution state (aarch64).

```
% cat /etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="9.8 (Plow)"

% uname -r
5.14.0-687.39.1.1toss.t5.aarch64
```

Software Modules

For AI/ML Applications

If you run AI/ML applications: the miniconda3/gh2 module in the /swbuild/analytix/tools directory is the only one that will work with the GH200 nodes:

```
cghfe01% module use -a /swbuild/analytix/tools/modulefiles
cghfe01% module load miniconda3/gh2
```

For more information, including the conda environments that can be used with the Grace Hopper nodes, see Using Conda Environments on NAS Systems.

The procedure to access JupyterLab on the GH200 system is the same as those outlined in Secure Setup for Using Jupyter Notebook on NAS Systems and Using Jupyter Notebook for Machine Learning Development on NAS Systems, except on the local system the user needs to proxyjump through a Grace Hopper front end. For example:

```
your_local_system% ssh -l username -o "StrictHostKeyChecking ask" \
-L 8080:/tmp/user.sock -o ProxyJump=sfe,cghfe01
```

For HPC Applications

You can find HPC software modules for the Arm-based GH200 nodes in the /nasa/arm directory. The current default $MODULEPATH includes /nasa/arm/modulefiles/toss5, where you can find many of the modules mentioned in this article.

```
cghfe01% echo $MODULEPATH
/etc/scl/modulefiles:/usr/share/Modules/modulefiles:/etc/modulefiles:/usr/share/ \
modulefiles:/nasa/arm/modulefiles/toss5
```

Note: The command line above is too long to be formatted as one line, so it is broken with a backslash (\).

Some of the available software packages are mentioned in the Programming and Compiling for Grace Hopper Nodes and Running a GPU Application sections. Be aware that many nvhpc or nvhpc-xxx modules (packaged and pre-built by NVIDIA) contain multiple components (such as CUDA, compiler, math, MPI, .., and so on.). If you want to use only one of the components, you can try other modules, for example, those with module names such as cuda, hpcx, openmpi, and so on.

WARNING: Software modules and executables built for the x86_64 platform (such as NAS PFEs, AFEs, and CFEs; bro_ele, sky_ele, cas_ait, rom_ait, mil_ait, and mil_a100 GPU nodes) cannot be used on the aarch64 platform (i.e., gh200 GPU nodes) because x86_64 and aarch64 architectures have different instruction sets. You must use modules under the /nasa/arm directory to recompile and run your applications for the Arm-based GH200 systems.

If your system startup script loads modules meant for use on x86_64 based systems (for example, the .bashrc, .profile, .cshrc, or .login scripts, which are accessible from all NAS x86_64 and Arm systems), you should unload those modules when you are on the GH200 front-ends and compute nodes.

Programming and Compiling for Grace Hopper Nodes

In addition to the information in this section, read Programming for NVIDIA GPUs and the references listed there for information on the recommended programming models for developing applications suitable for running on the NVIDIA GPUs. Some of the most commonly used software packages needed with these programming languages are CUDA toolkits, compilers, and math and MPI libraries.

CUDA Toolkit

You can find the CUDA version and its associated driver installed on the GH200 front ends and compute nodes by running the nvidia-smi command. Currently, it is CUDA version 13.3 and driver is 610.57.04.

Since the CUDA driver is backward compatible, an application compiled with an older CUDA Toolkit version can run with a newer driver release. For example, you can build your GPU application with CUDA Toolkit 12.6, and run it on a system with CUDA 13.3. See CUDA Toolkit and CUDA Driver for more information.

The following CUDA Toolkits are available for use on the GH200 nodes:

- **CUDA 12.6**

  ```
  module load nvhpc/25.1 (or module load nvhpc-nompi/25.1)
  nvcc --version
  nvcc: NVIDIA (R) Cuda compiler driver
  Copyright (c) 2005-2024 NVIDIA Corporation
  Built on Wed_Oct_30_00:08:18_PDT_2024
  Cuda compilation tools, release 12.6, V12.6.85
  Build cuda_12.6.r12.6/compiler.35059454_0

  OR

  module load nvhpc/24.11 (or module load nvhpc-nompi/24.11)
  nvcc --version
  nvcc: NVIDIA (R) Cuda compiler driver
  Copyright (c) 2005-2024 NVIDIA Corporation
  Built on Thu_Sep_12_02:27:38_PDT_2024
  Cuda compilation tools, release 12.6, V12.6.77
  Build cuda_12.6.r12.6/compiler.34841621_0
  ```

- **CUDA 12.3**

  ```
  module load nvhpc/24.3 (or nvhpc-nompi/24.3 or nvhpc-openmpi3/24.3)
  nvcc --version
  nvcc: NVIDIA (R) Cuda compiler driver
  Copyright (c) 2005-2023 NVIDIA Corporation
  Built on Fri_Nov__3_17:22:02_PDT_2023
  Cuda compilation tools, release 12.3, V12.3.103
  Build cuda_12.3.r12.3/compiler.33492891_0
  ```

Compilers

There are several compiler options for building your applications on the GH200 front ends. With any of these compilers, make sure your application builds, runs successfully, and validates correctly.

NVIDIA Compilers: nvc, nvc++, nvfortran, nvcc

To use an NVIDIA compiler, load one of the nvhpc modulefiles on either the cghfe01 or cghfe02 front end or on the GH200 compute nodes. Below are examples of a few possible choices:

- For the 25.1 version of the NVIDIA compilers, use:

  ```
  % module load nvhpc/25.1
  or
  % module load nvhpc-nompi/25.1
  ```

- For the 24.11 version of the NVIDIA compilers, use:

  ```
  % module load nvhpc/24.11
  or
  % module load nvhpc-nompi/24.11
  ```

- For the 24.3 version of the NVIDIA compilers, use:

  ```
  % module load nvhpc/24.3
  or
  % module load nvhpc-nompi/24.3
  ```

The NVIDIA compilers include the HPC compilers and the CUDA C/C++ compiler:

- NVIDIA HPC Compilers: nvc, nvc++, nvfortran

  These compilers can be used to compile pure CPU applications or GPU applications written in C/C++ and Fortran language standards (through the -stdpar[=gpu,multicore] flag); OpenACC (-acc[=gpu,multicore] flag); OpenMP (-mp[=gpu,multicore] flag), and CUDA Fortran. The gpu and multicore options let you choose whether to compile the application to run on the GPUs or on the multicore CPUs.

  Use the -tp=native flag or the -mcpu=native flag when compilation and execution occur on the same CPU processor (in this case, the Grace CPU) and use the -gpu=cc90 flag (i.e., compute capability 9.0) to target the Hopper GPU.

- NVIDIA CUDA C/C++ Compiler: nvcc

  The nvcc compiler is used to build CUDA C/C++ applications to run on the GPUs. This compiler splits the source code into *device code* and *host code*. The device code is compiled by nvcc. nvcc sends the host code to a C/C++ compiler on the host for compilation—for example, the gcc and g++ GNU compilers in the /usr/bin directory. You can choose a different host compiler—for example, a different version of GNU compiler—through the nvcc flag -ccbin=/*path_to_compiler* or --compiler-bindir=/*path_to_compiler*.

For more information, see Compiling GPU Applications for Different Generations of NVIDIA GPUs at NAS.

GNU Compiler Collection: gcc, g++, gfortran

- GNU compilers at the system location:

  ```
  cghfe01% which gcc
  /usr/bin/gcc
  cghfe01% gcc --version
  gcc (GCC) 11.5.0 20240719 (Red Hat 11.5.0-14)

  cghfe01% which gfortran
  /usr/bin/gfortran
  cghfe01% gfortran --version
  GNU Fortran (GCC) 11.5.0 20240719 (Red Hat 11.5.0-14)
  ```

  These compilers can only be used for building CPU-only applications, as they were not built with offloading support.

For all of the above compilers, NVIDIA recommends:

- Use the -mcpu=native flag instead of the -march=xxx or -mtune=xxx flags to optimize for the native CPU and use its features.
- Depending on the char sign type you assume for your application. you may need to use one of these C language flags: -fsigned-char, -funsigned-char, or -fnosigned-char.
  Note: For NVIDIA HPC compilers (nvc, nvc++, nvfortran), the default char sign type is -fsigned-char on x86_64 systems and -funsigned-char on aarch64 systems. For the Arm Compiler for Linux, the default is -fno-signed-char.

For the GNU compiler and/or Arm compiler:

- If using -Ofast results in unacceptable fast math optimizations, use -O3 -ffp-contract=fast. For even more accuracy, use -ffp-contract=off to disable floating point operation contraction used in fused multiply-add.
- Use -flto to enable link-time optimization, which can be significant.

For the GNU compiler, you can use -fsanitize=address -fsanitize=undefined to help debugging memory issues that might appear on

aarch64 systems (but are hidden on x86 systems due to differences on how compiler generate codes and layout data).

Math Libraries

There are two sets of math libraries. One targets the GPUs and the other targets the Grace CPUs.

- **CUDA libraries targeting the Hopper GPUs:**
  This set of libraries includes: cuBLAS, cuFFT, cuRAND, cuSOLVER, cuSPARSE, and cuTENSOR. They are located in the following directories:
  - If using module nvhpc/25.1:
    /nasa/arm/software/nvhpc/25.1/Linux_aarch64/25.1/math_libs/lib64
  - If using module nvhpc/24.11: /nasa/arm/software/nvhpc/24.11/Linux_aarch64/24.11/math_libs/lib64
  - If using module nvhpc/24.3: /nasa/arm/software/nvhpc/24.3/Linux_aarch64/24.3/math_libs/lib64

  When using the NVIDIA HPC compilers (nvc, nvc++, and nvfortran), these libraries can be linked with the flag -cudalib[=cublas|cufft|...].

- **NVIDIA Performance Libraries (NVPL) targeting the Grace CPUs:**
  NVPL is a set of optimized math libraries (BLAS, FFT, LAPACK, RAND, SCALAPACK, SPARSE, and TENSOR) that are used for the Grace CPUs. The NVPL set is available with NVIDIA HPC SDK version 24.3 (and later) in the /nasa/arm/.../math_libs/nvpl/libs directory.
  When using the NVIDIA HPC compilers (nvc, nvc++, and nvfortran), these libraries (with the exception of SCALAPCK) can be linked with the flag -Mnvpl[=blas,fft,...]. To link with NVPL SCALAPACK, use both -Mscalapack and -Mnvpl.
  If using the GNU compiler for linking with NVPL, use the -l, -L, and -lnvpl_.... options explicitly.

MPI Libraries

The GH200 nodes are not supported by HPE, and therefore HPE MPT library is not available on these nodes.

There are many modules containing OpenMPI. If you are not sure which one to use, start with one of these OpenMPI modules. Read Using OpenMPI on NAS Systems to learn how to use OpenMPI.

OpenMPI Version 4.1.7

- nvhpc/25.1: provided by NVIDIA, and built with gcc, g++, nvfortran, HPCX 2.21, and CUDA 12.6.
- nvhpc/24.11: provided by NVIDIA, and built with gcc, g++, nvfortran, HPCX 2.20, and CUDA 12.5.
- nvhpc/24.3: provided by NVIDIA, and built with gcc, g++, nvfortran, HPCX 2.17.1, and CUDA 12.2.
- hpcx/2.21: open source version of precompiled HPCX. The OpenMPI provided in this distribution was built with gcc, g++, gfortran, HPCX 2.21, and CUDA 12.6.
- openmpi/4.1.7-cuda12.6-gnu: built by NAS staff with GNU compiler, UCX 1.17, and CUDA 12.6.
  Loading openmpi/4.1.7-cuda12.6-gnu automatically loads nvhpc-nompi/24.11, which provides access to CUDA 12.6.
- openmpi/4.1.7-cuda12.6-nv: built by NAS staff with nvc, nvc++, nvfortran, UCX 1.17, and CUDA 12.6.
  Loading openmpi/4.1.7-cuda12.6-nv automatically loads nvhpc-nompi/24.11 which provides access to NVIDIA HPC compilers and CUDA 12.6.

OpenMPI Version 3.1.5

- nvhpc-openmpi3/24.3: provided by NVIDIA, and built with nvc, nvc++, nvfortran, CUDA 11.0, and without UCX.

TIP: When using newer versions of OpenMPI with UCX, for example, from nvhpc/24.11 or hpcx/2.21, setting the UCX_PROTO_ENABLE=n and UCX_GVA_ENABLE=n environment variables may help to reduce run-to-run variations or degraded performance in MPI communication. Set them as follows:

```
mpiexec -np .. -x UCX_PROTO_ENABLE=n -x UCX_GVA_ENABLE=n ...
```

For applications that perform UCX atomic operations on the Grace CPU, you may also need to add:

```
UCX_IB_MLX5_DEVX=no
```

See NVIDIA HPCX Known Issues (3606445, 3884209, 4026461 and 4050321) for more information.

Running a GPU Application

PBS jobs running on the GH200 nodes are served by a dedicated PBS server, pbs05a, running with PBS version 2024.1.4.

A GH200 compute node (72 CPU cores; 1 GPU; all host and device memory) cannot be divided or shared between PBS jobs or between different chunks of a PBS job (for example, -lselect=*N* means *N* chunks). The GH200 node model name is gh200. In your PBS resource request, you must use :model=gh200. You do not need to use -lplace=scatter:excl.

To run on the GH200 nodes, submit your PBS interactive or batch job from one of the two Grace Hopper front ends, cghfe01 or cghfe02. Use one of the following queues:

- gpu_normal: default priority (0), maximum walltime 24 hours.
- gpu_devel: priority 10, maximum walltime 2 hours, number of nodes 2.
- gpu_wide: priority 20, maximum walltime 12 hours (wide is considered 85+ nodes requested for the job).

- gpu_long: default priority (0); maximum walltime 120 hours; number of nodes 8; number of running jobs 2.

Note: The available queues will change from time to time. Use the command `qstat -Q` and `qstat -fQ` to find what queues have been created and enabled to be used by your group.

See the following sample PBS script.

Sample PBS Script for Running Jobs on Grace Hopper Nodes

```
#!/bin/bash
#PBS -lselect=2:ncpus=72:ngpus=1:model=gh200
#PBS -q name_of_queue

module load nvhpc/24.11

# for running 144 MPI ranks on the CPU cores across two nodes
# with 72 MPI ranks per node
# if necessary, also add -x LD_LIBRARY_PATH
mpiexec -np 144 -N 72 --machinefile $PBS_NODEFILE ./a_CPU_executable

# for running 2 MPI ranks on two GPUs across two nodes
# with 1 MPI rank per node
# if necessary, also add -x LD_LIBRARY_PATH
mpiexec -np 2 -N 1 --machinefile $PBS_NODEFILE ./a_GPU_executable
```

After job submission, you should see a JOBID similar to *xxx*.pbs0a5.gh.nas.nasa.gov.

Note: The versions of qstat on newer NAS systems (including Athena and Cabeus GH200 nodes) and older NAS systems (including Electra, Aitken, and Endeavour, and Cabeus V100 and A100 nodes) are different. To get the same look and feel of qstat on the newer systems as the older systems, it is recommended to use a NAS provided script called nas_qstat, available under /usr/local/bin on athfe[01-04] and cghfe[01-02]. In addition, the script showq under /u/hsp/bin can be used to visually see all jobs that are running or pending and their status, along with node availability at the bottom of the output.

References

- NVIDIA Grace Hopper Superchip Architecture White Paper
- NVIDIA Grace Performance Tuning Guide

The Cabeus cluster comprises multiple NVIDIA GPU nodes, including:

- 128 nodes with Milan CPU and NVIDIA A100 GPUs (model=mil_a100)
- 350 nodes with NVIDIA GH200 Superchip (model=gh200)

Among them, the gh200 nodes are served by their own front ends (cghfe01 and cghfe02) and PBS server (pbs05a), while the other GPU nodes are served by the front ends cfe01, cfe02 and PBS server pbspl4.

In this article, the term *Cabeus A100 GPU nodes* refers to the 128 nodes with Milan CPU and NVIDIA A100 GPUs.

To help you prepare for running jobs on the Cabeus A100 GPU nodes, this short user guide includes information on the general configuration of the nodes, compiling your code, and running PBS jobs.

Overview of Cabeus Nodes with AMD EPYC CPU and NVIDIA A100 GPUs

The Cabeus cluster includes two Cabeus front-end nodes (cfe01 and cfe02) for the GPU nodes with x86_64-based CPUs (mil_a100). Each of the CFEs includes one 16-core AMD EPYC 7313p processor and 256 GB memory.

Note: The Cabeus compute nodes are not accessible from the PFEs.

Each of the 128 A100 compute nodes, referred to as the mil_a100 nodes, contains:

- One 64-physical-core AMD EPYC 7763 Milan CPU socket (2.45 GHz) with 512 GB DDR4 host memory.
- Four NVIDIA A100 accelerator cards each with 3,456 double precision CUDA cores, 432 Tensor cores (boost clock 1410 MHz), and 80 GB HBM2e device memory
- Third-generation NVLink with 12 links yielding 600 GB/sec total bandwidth for GPU-to-GPU communication
- PCIe 4.0 bus connecting the CPU socket and the GPU accelerator cards with a bandwidth of 31.5 GB/s per direction

These 128 compute nodes are partitioned into twelve physical racks. They are connected in a fat-tree topology to the Cabeus InfiniBand network (ib0 and ib1) via four-lane High Data Rate (4X HDR) devices and 4X HDR switches. These HDR devices and switches provide a maximum bandwidth of 200 Gbits/s (i.e., 25 GB/s) for internode communication.

All existing HECC filesystems such as the home, VAST, and Lustre filesystems are mounted on both CFEs and on the Cabeus compute nodes.

PBS jobs for running on Cabeus A100 nodes are managed by the PBS server pbspl4.

Requesting GPU Allocations

An SBU allocation specifically for the GPUs is required to use the GPU resources. For more information about requesting GPU allocations, see this article.

The SBU rates for using one GPU node for one hour, for each type of node, are:

- mil_a100 with four A100s: 37.86
- gh200 with one GH200 Superchip: 18.34

Depending on how you request the GPU resources, your job may be charged the full rate or a fraction of the SBU rate. See the next section for some examples.

Accessing the CFEs and the mil_a100 nodes

You can access the Cabeus front ends (CFEs) either from your local desktop or from the PFEs. If you have set upSSH passthrough for the CFEs, then do:

```
your_localhost% ssh cfe01   (or ssh cfe02)
```

If you are connected to a PFE, then do:

```
pfe% ssh cfe01 (or ssh cfe02)
```

You can access the Cabeus Milan A100 compute nodes only through PBS jobs submitted from the CFEs or from pbspl4. To request Milan A100 (mil_a100) resources, specify one of the queue names shown below and the model type mil_a100.

The following PBS attributes are required when requesting the A100 resources:

- **model=mil_a100:** The A100 nodes with the AMD EPYC 7763 CPU are designated with a model name mil_a100.
- **-q**, followed by one of the queues shown below, to access the mil_a100 nodes.

The PBS queues for the Milan A100 nodes are as follows:

- **gpu_normal**
  - Maximum wall time: 24 hrs
  - Maximum node count: 32

- **gpu_devel**
  - Maximum wall time: 2 hrs
  - Maximum node count: 2
  - Maximum user jobs (queued or running): 1
  - Priority: 10

- **np_gpu_wide (non-primetime)**
  - Maximum wall time: 12 hrs
  - Maximum node count: 64
  - Minimum node count: 24
  - Maximum user jobs (queued or running): 2
  - Priority: 20

Important: The **np_gpu_wide** queue will run jobs between 7:00 a.m. Thursdays and 7:00 a.m. Saturdays (Pacific Time). You can submit jobs to the **np_gpu_wide** queue at any time during the week, but the jobs will run only during this period.

As shown in the examples below, adding @pbspl4 to the queue name is optional, whether you submit the job from a CFE or from pbspl4.

```
cfe% qsub -I -q gpu_devel@pbspl4
   -lselect=1:ncpus=10:ngpus=1:mem=xxGB:model=mil_a100
```

or

```
pbspl4% qsub -I -q gpu_devel
   -lselect=1:ncpus=10:ngpus=1:mem=xxGB:model=mil_a100
```

Note: The qsub commands are shown on two lines due to a KB formatting issue. They should be on one line.

When not specified, the default value of ncpus, ngpus, and host mem for all GPU node types (including mil_a100) is 1, 1, and 32 GB, respectively. The minimum value of ncpus, ngpus, and host mem is 1, 1, and 256 MB, respectively.

From the perspective of PBS, one physical node can be made up of multiple virtual nodes (*vnodes*). In the case of mil_a100, one physical node with 4 GPUs is partitioned as 4 vnodes, where each vnode includes 1 GPU, 16 physical cores, and 128 GB of memory.

PBS allocates resources in the vnodes differently depending on the actual selection specified in -lselect=... and the choice of the PBS -lplace statement, which includes an arrangement attribute (free, pack, scatter, vscatter) and a sharing attribute (shared, excl, and exclhost). Read **man qsub** for the definition of each value of the arrangement and the sharing attribute.

If you plan to use fewer than 4 GPUs per physical mil_a100 node, you can use the shared sharing attribute which allows other jobs to use the remaining GPUs in the node. It also allows your job to be charged for only the number of GPUs assigned to your job. For example, the following request will be charged for the use of only 1 GPU in a mil_a100 node:

```
-lselect=1:ncpus=16:ngpus=1:mem=60GB:model=mil_a100 -lplace=scatter:shared
```

If you do not want other jobs to share resources in the mil_a100 nodes assigned to your job, it is easier if you simply request the use of all 4 GPUs in each mil_a100 node. For example:

```
-lselect=1:ncpus=16:ngpus=4:mem=60GB:model=mil_a100 -lplace=scatter:excl
```

This request will result in being charged for all 4 GPUs in a mil_a100 node.

Operating System and Software Modules

The operating system on all A100 compute nodes is TOSS 4 (Red Hat Enterprise Linux-based Tri-Lab Operating System Stack Version 4). Batch jobs use :aoe=toss4 by default, so you do not need to add :aoe=toss4 in your PBS resource request. The default $MODULEPATH includes:

- /usr/share/Modules/modulefiles
- /nasa/modulefiles/toss4
- /nasa/modulefiles/spack
- /nasa/modulefiles/pkgsrc/toss4

Programming and Compiling For Cabeus A100 Nodes

Read Programming for NVIDIA GPUs and the references listed there for information on the recommended programming models for developing applications suitable for running on the NVIDIA GPUs.

Depending on the programming model and language chosen, use an appropriate compiler available in the NVIDIA HPC SDK to compile your application:

- NVIDIA HPC Compilers
  - nvfortran
  - nvc
  - nvc++
- NVIDIA CUDA C/C++ Compiler - nvcc

Run the command module avail nvhpc to find available versions of NVIDIA HPC SDK on the system. The examples shown in this article

use nvhpc/23.7. However, you can choose a different version.

```
% module load nvhpc/23.7
```

or

```
% module load nvhpc-nompi/23.7
```

For building NVIDIA GPU applications with MPI for inter-GPU or inter-node communication, you can use either OpenMPI or HPE MPT:

OpenMPI from nvhpc

Example 1:

```
module load nvhpc/23.7
mpif90 -o your_application.exe -acc=gpu -gpu=cc80
    -Minfo=accel your_application.f
```

Note: The mpif90 line is shown on two lines due to KB formatting issues. It should be a single line.

Example 2:

```
module load nvhpc/23.7
mpicc -o main.o -c main.c
nvcc  -o cuda_code.o -c cuda_code.cu
mpicc -o your_program.exe main.o cuda_code.o -lcudart
```

Note: The default OpenMPI version in nvhpc/23.7 is 3.1.5.

HPE MPT library

This is recommended for building/running applications when using NAS GPU nodes supported by HPE.

Example 1:

```
module load nvhpc-nompi/23.7
module load mpi-hpe/mpt
nvfortran -o your_application.exe \
-acc=gpu -gpu=cc80 -Minfo=accel your_application.f -lmpi
```

Example 2:

```
module load nvhpc-nompi/23.7
module load mpi-hpe/mpt
nvc  -o main.o -c main.c -lmpi
nvcc -o cuda_code.o -c cuda_code.cu
nvc  -o your_program.exe main.o cuda_code.o -lcudart -lmpi
```

Instead of explicitly using nvfortran/nvc/nvc++ ... -lmpi in the command line, you can use the mpif90, mpicc, and mpicxx wrappers from HPE MPT. However, it is better to set the following variables before invoking mpif90, mpicc, and mpicxx to ensure nvfortran/nvc/nvc++ are used instead of gfortran/gcc/g++:

```
setenv MPIF90_F90 nvfotran
mpif90 [options] program.f

setenv MPICC_CC   nvc
mpicc  [options] program.c

setenv MPICXX_CXX nvc++
mpicxx [options] program.cc
```

TIP for Advanced Developers: Instead of using MPI, you can develop applications using the NVIDIA Collective Communications Library (NCCL) for multi-GPU and multi-node communication. Read this NVIDIA page to learn more.

Running a CUDA Application

The following environment variables may be useful during runtime:

- MPI_USE_CUDA: If using HPE MPT library for running a CUDA application across multiple GPUs or across multiple nodes, setting this variable will enable CUDA support so that the buffers on the GPUs can be used with sends, receives, collectives, and MPI RMA calls.
- CUDA_VISIBLE_DEVICES: Use this variable to control which GPUs your application uses. This is useful when you want to target a specific GPU or a set of GPUs for debugging or testing.

Sample PBS Script

```
#PBS -l select=1:ncpus=16:ngpus=4:mem=240GB:model=mil_a100
#PBS -l walltime=8:00:00
## If you request more than 1 chunk (i.e., select=2 or more),
## with resources which are scattered (i.e., not using the same physical nodes)
## and you want exclusive access to the vnodes assigned,
## uncomment the following place statement
##PBS -l place=scatter:excl
#PBS -q gpu_devel

module load nvhpc-nompi/23.7
module load mpi-hpe/mpt

cd $PBS_O_WORKDIR

setenv MPI_USE_CUDA 1

## if you want to run on two specific GPUs,
## for example, GPU0 and GPU2

setenv CUDA_VISIBLE_DEVICES 0,2
mpiexec -np 2 ./your_application.exe

## if you want to run on all four GPUs in the node

setenv CUDA_VISIBLE_DEVICES 0,1,2,3
mpiexec -np 4 ./your_application.exe
```

References

- Cabeus Configuration Details
- PBS Job Sharing on GPU Nodes
- Requesting GPU Resources
- Programming for NVIDIA GPUs

UPDATE IN PROGRESS: On August 24, 2026, the Skylake/V100 and Cascade Lake/V100 nodes were decommissioned from the Cabeus supercomputer. This article is currently undergoing updates to reflect the changes in the NAS environment and may not accurately reflect the resources available to users.

To develop code for use on the NVIDIA GPU nodes, you can choose one of the three NVIDIA-recommended programming models or combine them to take advantage of the underlying NVIDIA acceleration libraries:

- Standard-language-based parallel programming model.
- Directive-based OpenACC model.
- Language-extension-based CUDA model.

Each model is briefly described in this article. Read the relevant references for each model listed at the end of this article to learn more information and limitations.

Note: NVIDIA provides limited support for using OpenMP on the GPUs. This model is not described in this article.

Compiler and CUDA Support

Compiler support for these models is provided through the NVIDIA HPC Software Development Kit (SDK), which includes these compilers:

- NVIDIA HPC Compilers for NVIDIA GPUs and Intel, AMD, OpenPower, and Arm CPUs:
  - NVFORTRAN: Fortran compiler with support for CUDA Fortran, originally developed by PGI
  - NVC: C11 compiler
  - NVC++: C++17 compiler
- NVIDIA CUDA C and C++ Compiler Driver for NVIDIA GPUs: NVCC

Note: The PGI compilers and tools were rebranded to become parts of the NVIDIA HPC SDK.

As these models are under active development or improvement, we recommend using the latest version of the NVIDIA HPC SDK, as it should have better support for new features when they are added to these programming models. Be aware that the available versions of NVIDIA HPC SDK at NAS vary between the GPU nodes with x86_64-based CPU (sky_gpu, cas_gpu, rom_gpu, mil_a100) and those with the aarch64-based CPU (gh200). Run the command `module avail nvhpc` to see what are available on each system before using the command `module load` to load one of them.

For more information on each compiler, read its man page or help page after loading a module. For example:

- man nvfortran
- nvfortran -h
- nvfortran -gpu -h

During runtime, your CUDA application requires one CUDA-capable GPU (for example, a V100, A100, or H100 GPU) and a CUDA driver that is compatible with the version of CUDA Toolkit used for building the application. Be aware that the CUDA driver is backward compatible. Therefore, an application compiled with an older CUDA Toolkit version can run with a newer CUDA driver. See CUDA Toolkit and CUDA Driver to learn more.

Note: The CUDA driver is installed by the system administrator on the GPU. Run either the `nvidia-smi` or the `nvaccelinfo` command to find the CUDA driver version installed on the system.

The Standard-Language-Based Parallel Programming Model

In this model, parallelism for running on the NVIDIA GPUs is achieved solely using features provided by the standard C++ and Fortran languages without the use of compiler directives such as OpenMP and OpenACC or language extensions such as CUDA. In addition, applications written with standard language parallelism can run on any platform—such as different multi-core CPUs or GPUs from different vendors—that supports these languages.

International Organization for Standardization (ISO) C++

The C++17 Standard introduced higher-level parallelism features that allow you to request a specific execution policy as the first parameter to a Standard Library algorithm. Most Standard C++ algorithms support the execution policies. The four available policies are:

- **std::execution::seq:** Sequential execution. No parallelism is allowed.
- **std::execution::unseq:** Vectorized execution on the calling thread (this execution policy was added in C++20).
- **std::execution::par:** Parallel execution on one or more threads.
- **std::execution::par_unseq:** Parallel execution on one or more threads, with each thread possibly vectorized.

Here is an example from NVIDIA documentation—a sort algorithm with the par execution policy:

```
std:sort(std::execution::par,
employees.begin(), employees.end(),
CompareByLastName());
```

NVC++ can compile Standard C++ algorithms with the `par` or the `par_unseq` execution policies through the use of the `-stdpar` compiler

flag and automatically offloads C++ parallel algorithms to NVIDIA Volta, Turing, Amphere, or Hopper GPUs.

ISO Fortran

ISO Fortran 2018 introduced the DO CONCURRENT construct to allow you to express loop-level parallelism:

```
DO CONCURRENT (i=1:n)
  loop body
END DO
```

When the NVFORTRAN -stdpar option is specified, the compiler parallelizes the DO CONCURRENT loops and offloads them to the GPU.

Example codes that use this model are available in the NVIDIA HPC SDK installed on NAS systems.

For GPU nodes with x86_64-based CPU:

```
/nasa/nvidia/hpc_sdk/..../examples/stdpar
```

For GPU nodes with aarch64-based CPU:

```
/nasa/arm/software/nvhpc/..../examples/stdpar
```

You can compile code either on a GPU node; on a Cabeus front end (CFE), to run on the GPUs with x86_64-based CPUs (sky_gpu, cas_gpu, rom_gpu, mil_a100); or on cghfe01 and cghfe02 to run on the GPUs with aarch64 based CPU (gh200):

```
nvc++ -stdpar[=gpu] [-gpu=ccXY] program.cpp
nvfortran -stdpar[=gpu] [-gpu=ccXY] program.f90
```

In the event you want to compile the same code for running on multicore CPUs, use:

```
nvc++ -stdpar=multicore program.cpp
nvfortran -stdpar=multicore program.f90
```

To facilitate such standard language parallelism, NVC++ and NVFORTRAN rely on CUDA Unified Memory for all data movement between CPU and GPU memory. Through support in both the CUDA device driver and the NVIDIA GPU hardware, the CUDA Unified Memory manager automatically moves some types of data based on usage.

NVC++ and NVFORTRAN auto-detect the host CPU and device GPU architectures of the system where you perform the compilation. An executable built on one system may not run on another system if the host CPU instructions and/or the GPU instructions in the executable are not compatible between the two systems.

To generate an executable for a specific CPU architecture, add the -tp option. For example, use -tp=skylake for a CPU host with Intel Skylake or Cascade Lake processors, -tp=zen2 for the AMD Rome processor, or -tp=zen3 for the AMD Milan processor. For the GH200 GPU nodes (gh200), NVIDIA recommends using the flag -tp=native or -mcpu=native.

To generate an executable for a specific GPU architecture, add the -gpu=*ccXY* option, where *ccXY* stands for the compute capability of an architecture. For example, use -gpu=cc70 for V100 nodes, -gpu=cc80 for A100 nodes, and -gpu=cc90 for the GH200 nodes. Using multiple -gpu=*ccXY* options in a single compilation will generate one version of code for each architecture, but the compilation time will be longer.

Executables built for the x86_64 platform (PFE, AFE, CFE, ivy, has, bro, sky_ele, cas_ait, rom_ait, mil_ait, sky_gpu, cas_gpu, rom_gpu, and mil_100 GPU nodes) cannot be used on the aarch64 platform (gh200 GPU nodes) because x86_64 and aarch64 architectures have different instruction sets.

Note: A executable built with just -gpu=cc70 can run on an architecture with compute capability cc70 (V100) or cc80 (A100). However, an executable built with just -gpu=cc80 can run on an architecture with compute capability cc80 (A100) but not cc70 (V100).

Note: -stdpar defaults to -stdpar=gpu. If you compile the application with -stdpar on a Cabeus front end (cfe01 and cfe02, for running on the GPU nodes with x86-based CPUs) or on cghfe01 or cghfe02 (for running on the GPU nodes with aarch64-based CPUs) but if you do not explicitly specify one or more -gpu=*ccXY*, it will automatically enable: -gpu=cc50 -gpu=cc60 -gpu=cc61 -gpu=cc70 -gpu=cc75 -gpu=cc80 -gpu=cc86 -gpu=cc90

The Directive-Based OpenACC Model

OpenACC is a programming standard developed by PGI, CRAY, CAPS, and NVIDIA for offloading programs written in C, C++, and Fortran from a host CPU to an attached accelerator device.

Similar to OpenMP, OpenACC requires you to annotate your source code with directives describing sections of your code and corresponding data that are to be executed on the GPU:

- For C and C++, the directive is #pragma acc ...
- For Fortran, the directive is !$acc ...

Example OpenACC codes from NVIDIA are available on the NAS systems under:

For GPU nodes with x86_64-based CPUs:

```
/nasa/nvidia/hpc_sdk/..../examples/OpenACC
```

For GPU nodes with aarch64-based CPUs:

/nasa/arm/software/nvhpc/..../examples/OpenACC

To compile your OpenACC code, use the NVIDIA HPC compilers (nvfortran, nvc, or nvc++) and options on a system with NVIDIA GPUs. For example:

```
nvfortran -acc=gpu -fast -Minfo=all your_program.f
nvc       -acc=gpu -fast -Minfo=all your_program.c
nvc++     -acc=gpu -fast -Minfo=all your_program.cc
```

where -acc=gpu will parallelize and offload OpenACC regions to the NVIDIA GPUs. The option -Minfo will generate informational messages about optimization. You can specify -acc=multicore to parallelize for a multicore CPU or -acc=host to generate an executable that will run serially on the host CPU.

As in the case of the standard-language-based model described above, you can also use the -tp option to specify a target CPU architecture and/or the -gpu=*ccXY* option to target GPUs with the specified compute capability.

NVIDIA OpenACC supports multiple environment variables to control behavior of accelerator-enabled programs at execution. See the Environment Variables section of the OpenACC Getting Started Guide listed in the References.

Note on OpenMP Support: Although NVIDIA does not emphasize the use of OpenMP, NVIDIA HPC compilers do support a subset of the OpenMP API with a focus on OpenMP 5.0 features for running on CPUs and GPUs. Similar to compiling OpenACC code with -acc=gpu -gpu=cc80, use the compiler flags -mp=gpu -gpu=cc80 to compile OpenMP code for offloading to run on the A100 GPUs. For codes that are well-structured for the GPUs, these should compile and execute with performance equal to or close to equivalent OpenACC. Codes that are not well-structured for GPUs may perform poorly but should execute correctly. Read the OpenMP chapter of the NVIDIA HPC Compilers User's Guide for more information.

The Language-Based CUDA Model

The Compute Unified Device Architecture (CUDA) was created by NVIDIA and implemented on the NVIDIA GPUs. With CUDA, code developers write programs in C, C++, and Fortran, and incorporate CUDA extensions and optimized libraries. This is the approach to take if you want to obtain the best possible performance on the NVIDIA platform.

A CUDA code usually includes the following sequence of operations:

1. Declare and allocate the host memory (i.e., the CPU) and device (i.e. the GPU) memory.
2. Initialize host data.
3. Copy data from host to device.
4. Run the kernels on the GPUs.
5. Copy the results from device to host.
6. Free the allocated host and device memory.

You can use either of the following programming environments:

CUDA C/C++

CUDA C/C++ is the original CUDA programming environment developed by NVIDIA as extensions to the C/C++ languages for GPUs. It allows direct programming for the GPU from a high-level language. This NVIDIA technical blog provides an easy introduction to CUDA C and C++.

If you choose to write or rewrite portions of your code in CUDA C/C++, you will need to use the NVIDIA CUDA C/C++ compiler (nvcc) to build the executable. For example:

nvcc *your_program*.cu

The suffix .cu indicates it is a CUDA C/C++ file.

A CUDA application binary (with one or more GPU kernels) can contain compiled GPU code in two forms: (1) binary cubin objects and (2) forward-compatible PTX assembly for each kernel. During application load time, the PTX form is compiled just-in-time (JIT) to a cubin for execution.

To build an application that can run on either V100 or A100 nodes, or on a future architecture, use the -gencode option as follows:

```
nvcc -gencode=arch=compute_70,code=sm_70
    -gencode=arch=compute_80,code=sm_80
    -gencode=arch=compute_80,code=compute_80
    your_program.cu
```

where the first and second -gencode options ending with code=sm_70 and sm_80 cause the generation of a cubin targeting V100 or A100, respectively, and the last -gencode option ending with code=compute_80 allows for the generation of PTX assembly to provide compatibility of future architectures through JIT compiling.

To build an application that can run on the GH200 nodes, use:

```
nvcc -gencode=arch=compute_90,code=sm_90
    -gencode=arch=compute_90,code=compute_90
    your_program.cu
```

CUDA Fortran

CUDA Fortran is a product of a collaboration between PGI and NVIDIA. It is based on the CUDA C/C++ runtime API, and includes a Fortran 2003 compiler and tool chain for programming NVIDIA GPUs using Fortran. Read this NVIDIA Developer Blog to get a quick introduction of CUDA Fortran.

Example CUDA Fortran codes are available in the NVIDIA HPC SDK installed on NAS systems:

For GPU nodes with x86_64-based CPUs:

/nasa/nvidia/hpc_sdk/..../examples/CUDA-Fortran

For GPU nodes with aarch64-based CPUs:

/nasa/arm/software/nvhpc/..../examples/CUDA-Fortran

To use CUDA Fortran, use the compiler nvfortran to build the executable. For example:

nvfortran *your_program*.cuf

Files with .cuf or .CUF extensions automatically enable CUDA Fortran compilation. If the filename of your program does not end with the .cuf or .CUF suffix, use the -cuda option. For example:

nvfortran -cuda *your_program.xx*

As in the case of the previous two programming models described above, you can also use the -tp option to specify a target CPU architecture and/or the -gpu=*ccXY* option to target GPUs with the specified compute capability.

If you have any questions or need assistance, contact the NAS Control Room at (800) 331-8737, (650) 604-4444, or support@nas.nasa.gov.

References for NVIDIA HPC Compilers

- NVIDIA HPC Compilers User's Guide

References for the Standard-Language-Based Parallel Programming Model

- Video - No More Porting: Coding for GPUs with Standard C++, Fortran and Python
- Developing Accelerated Code with Standard Language Parallelism
- C++ Parallel Algorithms
- Accelerating Standard C++ with GPUs Using stdpar
- Multi-GPU programming with standard Parallel C++, Part 1
- Multi-GPU programming with standard Parallel C++, Part 2
- Accelerating Fortran DO CONCURRENT with GPUs and the NVIDIA HPC SDK
- Using Fortran Standard Parallel Programming for GPU Acceleration
- Bringing Tensor Cores to Standard Fortran

Reference for the OpenACC Programming Model

- OpenACC Getting Started Guide

References for the the Language-Extension-Based CUDA Model

- An Even Easier Introduction to CUDA
- CUDA C++ Programming Guide
- NVIDIA Amphere GPU Architecture Compatability Guide for CUDA Applications
- An Easy Introduction to CUDA Fortran
- NVIDIA CUDA Fortran Programming Guide
- Using Tensor Cores in CUDA Fortran

The Cabeus cluster comprises two types of NVIDIA GPU nodes, including nodes with x86_64-based CPUs and nodes with aarch64-based CPUs.

Note: You must have a Standard Billing Unit (SBU) allocation specifically for GPUs in order to use the GPU nodes.

Nodes with x86_64-Based CPUs

- 128 nodes each with one Milan CPU and four NVIDIA A100 GPUs (model=mil_a100)

These GPU nodes are served by PBS server pbspl4 and are accessible from the Cabeus front ends, cfe01 and cfe02.

The nodes are sharable by different jobs. Instead of each node being treated as one unit for exclusive access by a single job, each node is logically split into four *vnodes*.

If you want exclusive access to a node, request the use of all the resources in a node and specify the placement -lplace=scatter:excl. for example:

```
#PBS -lselect=1:ncpus=128:ngpus=4:mem=480GB:model=mil_a100
#PBS -lplace=scatter:excl
#PBS -q queue_name@pbspl4
```

where the use of @pbspl4 in the line #PBS -q is optional.

Note: There are 64 physical cores per mil_a100 node. However, PBS sees each mil_a100 node as having 128 logical cores due to hyperthreading enabled. If you specify :ncpus=64, PBS will assign 64 logical cores all from the first 32 physical cores. To get all 64 physical cores assigned to your job, use :ncpus=128.

Nodes with aarch64-based CPU

Cabeus contains 538 nodes with NVIDIA GH200 Superchips (model=gh200).

These nodes are accessible from the Cabeus Grace Hopper front ends, cghfe01 and cghfe02. PBS jobs are served by pbs05a, where user login is not allowed.

The gh200 nodes have only one GPU per node. A node is configured via PBS for exclusive use by a single job, and it is not necessary to explicitly request the amount of resources. For example, the following two PBS resource requests provide you with the same exclusive access to all resources of a whole node:

```
#PBS -lselect=1:model=gh200
or
#PBS -lselect=1:ncpus=72:ngpus=1:mem=460GB:model=gh200
#PBS -q queue_name@pbs05a
```

where the use of @pbs05a in the line #PBS -q is optional.

Requesting AMD Milan A100 GPU Nodes (mil_a100)

Each mil_a100 node contains only one socket; each socket is further split into four vnodes where eachmil_a100 vnode contains 1 GPU, 16 CPU physical cores, and 128 GB of memory.

The new queue structure utilizes the PBS primetime feature. A "nonprime-time" queue, np_gpu_wide, will run jobs only from 7:00 a.m. Thursday to 7:00 a.m. Saturday (Pacific Time). The gpu_normal and gpu_devel queues will run during both primetime and non-primetime (i.e., any time). You can submit jobs to any of the queues at any time during the week, but the non-prime time jobs will run only during their designated period.

The following PBS queues are enabled for use:

- **gpu_normal**
  - Maximum wall time: 24 hrs
  - Maximum node count: 32

- **gpu_devel**
  - Maximum wall time: 2 hrs
  - Maximum node count: 2
  - Maximum user jobs (queued or running): 1
  - Priority: 10

- **np_gpu_wide (non-primetime)**
  - Maximum wall time: 12 hrs
  - Maximum node count: 64
  - Minimum node count: 24
  - Maximum user jobs (queued or running): 2
  - Priority: 20

Notes:

- Long running jobs need to implement checkpoint/restart if they cannot fit into the wall-time limit of these queues.
- The maximum amount of resources or certain settings for these queues may change from time to time. To get the current setting, run the command `qstat -fQ queue_name`. Note that the `resources_max.nodect` value refers to the number of physical nodes allowed per job.

In your job request, specify the number of CPU cores, the number of GPUs, and the amount of host memory you want for your job. The following example shows how to request exclusive usage of a whole node:

```
#PBS -lselect=1:ncpus=128:ngpus=4:mem=480GB:model=mil_a100
#PBS -lplace=scatter:excl
```

If `ncpus`, `ngpus`, or `mem` is not specified, the default values (`ncpus=1`, `ngpus=1`, `mem=32GB`) will be used.

Requesting NVIDIA GH200 GPU Nodes (gh200)

To run on the GH200 nodes, submit your PBS interactive or batch job from one of the two Grace Hopper front ends, cghfe01 or cghfe02. Use one of the following queues:

- **gpu_normal**
  - Maximum wall time: 24 hrs
  - Priority: Default (0)

- **gpu_devel**
  - Maximum wall time: 2 hrs
  - Maximum node count: 2
  - Maximum user jobs (queued or running): 1
  - Priority: 10

- **gpu_wide**
  - Maximum wall time: 12 hrs
  - Priority: 20
  - Wide is considered 85 or more nodes requested for a job.

- **gpu_long**
  - Maximum wall time: 120 hrs
  - Maximum node count: 8
  - Maximum user jobs (queued or running): 2
  - Priority: Default (0)

Note: Since each `gh200` node is set up to not allow job sharing, the use of PBS placement such as `-lplace=scatter:excl` is not required.

TIP: The available queues will change from time to time. Use the command `qstat -Q` and `qstat -fQ` to find what queues have been created and enabled to be used by your group.

Charging for Use of the GPU Nodes

A separate SBU allocation exclusively for the GPU nodes is required. For information on requesting SBU allocation for the GPU nodes, see this article.

The SBU rates for using 1 GPU node for 1 hour are:

- `mil_a100` with four A100s: 37.86
- `gh200` with one H100: 18.34

Note: For the GPU nodes with aarch64-based CPU (`model=gh200`), your job will be charged for the SBU cost of a 1 whole node regardless of how the resources are requested, since these nodes are not set up for job sharing (each job is allocated with all of the resources in a whole node).

For the GPU nodes with x86_64 based CPUs, since a node is sharable by multiple jobs, your job may be charged for the SBU cost of 1 whole node or a fraction of the node depending on the resource request and the place statement in your PBS job. In addition, charging is based on the number of GPUs allocated to or excluded from use by other jobs, with no consideration of CPUs and memory. Below are three examples.

Example 1

This resource request will be charged for one-quarter of the `mil_a100` SBU rate of 37.86, because only 1 GPU will be allocated to the job. In addition, it allows the unused CPU, GPU, and memory in the node to be used by other jobs:

```
#PBS -lselect=1:ncpus=1:ngpus=1:mem=60GB:model=mil_a100
#PBS -lplace=scatter:shared
```

Example 2

This job will be charged for 2 times the SBU rate of 37.86, because all four GPUs in each of the two`mil_a100` nodes will be allocated

and no other jobs will be able to share the resources:

```
#PBS -lselect=2:ncpus=1:ngpus=4:mem=60GB:model=mil_a100
#PBS -lplace=scatter:excl
```

Example 3

This job will be charged for one half (2 x 0.25) the SBU rate of 37.86, because only 1 GPU each from 2 different nodes will be allocated and the remaining 3 GPUs in each node can be used by other jobs:

```
#PBS -lselect=2:ncpus=1:ngpus=1:mem=60GB:model=mil_a100
#PBS -lplace=scatter:excl
```

Checking Job Status

To check the status for all of your jobs submitted to a particular queue, run the qstat command. The following examples use the queue gpu_normal served by the pbspl4 PBS server. For jobs submitted to queues served by the pbs05a PBS server, change the front end from cfe to cghfe01 or cghfe02, the PBS server from pbspl4 to pbs05a, and use an appropriate queue name.

```
cfe% qstat gpu_normal@pbspl4 -u your_username
```

You can also check the status of a single job. For example, to check the status of job 140101 on pbspbl4:

```
cfe% qstat 140101.pbspl4.nas.nasa.gov
```

For single-node I/O that uses memory-based temporary filesystem, we recommend using a job-based temporary directory, $TMPDIR, instead of the node-wide /tmp directory. This recommendation is particularly important for jobs running on the NAS GPU nodes with x86_64-based CPUs (mil_a100) where node sharing is enabled.

Background

All of the NAS compute nodes—including the CPU nodes in Electra, Aitken, and the GPU nodes in Cabeus—are configured to allow up to 50% of the node's main memory for use as a temporary filesystem (/tmp) to achieve faster single-node I/O. However, using the /tmp directory is not feasible for multi-node I/O, as /tmp is a filesystem local to a single node.

Many users use the /tmp directory on the CPU nodes to speed up their jobs' I/O. When these jobs end, data under/tmp are not always removed by the application or user. In order to provide a clean /tmp directory for a new job that uses the same node, the PBS prologue will first unmount /tmp to clean up leftover data from the previous job, then will remount it for the new job.

On the GPU nodes where node sharing is enabled, the resources on each node (including/tmp) can be shared by multiple jobs. Therefore, the process of cleaning up /tmp depends on whether a new job is the only job on a node or whether there are other jobs running on the node:

- If a new job starting is the only job on a node, the PBS prologue will use the same process as on CPU nodes by unmounting and remounting /tmp.
- If there are other jobs currently running on the same node, the prologue will not unmount /tmp, because doing so would remove the data needed by the other running jobs. Instead, the prologue will look for /tmp/pbs.*jobID* directories that do *not* have a running job, and delete those directories if they exist. The $TMPDIR path is /var/tmp/pbs.*jobid*.

Therefore, to ease the removal of user data from old jobs, you should use the $TMPDIR directory instead of writing directly to /tmp.

Using the $TMPDIR Directory

Note: For some NAS systems, /var/tmp is a symlink of /tmp. For the Milan A100 (mil_a100) nodes, /var/tmp and /tmp are two different directories, but they both use the same 50% main memory pool of the nodes.

How much $TMPDIR space can a job use?

On the GPU nodes, $TMPDIR cannot use more than the amount of CPU host memory requested by a job. For example, in the following PBS job request, the 16 GB of CPU host memory requested by the job will be shared by the memory usage of the application; buffer cache usage of the application (if buffered I/O is enabled); and the usage for storing data under /tmp or $TMPDIR:

```
#PBS -lselect=1:ncpus=10:ngpus=1:mem=16GB:model=mil_a100
```

Therefore, the maximum space available for $TMPDIR is 16 GB in this example.

In addition, the combined $TMPDIR usage by all jobs running on the same node cannot exceed 50% of the node's main memory. If another job running on the same node as your job already has data stored in the job's $TMPDIR with an amount equal to the size of 50% of the node's main memory, you will encounter a "No space left on device" error when you attempt to write data into your job's $TMPDIR.

Sample Script for Using $TMPDIR

A job's resources (CPU, GPU, and memory) are accessible only during the lifetime of the job. To use $TMPDIR for I/O, you must copy data into $TMPDIR before the data are needed by your application. Any generated data needed for post-processing or restarting the next job must be copied to the $HOME or Lustre /nobackup directory before the current job is terminated. You can accomplish this by using the following sample PBS script:

```
#PBS -lselect=1:ncpus=16:ngpus=1:mem=16GB:model=mil_a100
#PBS -q gpu_devel

module load ....

cd $TMPDIR

cp /nobackup/username/project/input .

mpiexec -np ....

cp output /nobackup/username/project
#End of script
```

The NAS Cabeus cluster currently includes NVIDIA GPUs with the Ampere (mil_a100) and Hopper (gh200) microarchitectures. This article provides some information on the compute capability of the various microarchitectures and how to compile applications written in C/C++ with CUDA extensions for a target microarchitecture.

For applications written in ISO C++, ISO Fortran, OpenACC or CUDA Fortran, read the various sections in Programming for NVIDIA GPUs to learn how to compile for a target microarchitecture.

Compute Capabilities

The high-level features of each architecture can be found using the command nvaccelinfo included in a NVIDIA HPC Software Development Kit (SDK) installed as a module such as nvhpc/23.11. For example, to find information about the A100 GPUs in a NAS mil_a100 node, perform the following:

```
Cabeus_Front_End% qsub -I -q gpu_devel -lselect=1:ngpus=1:model=mil_a100
PBS cb12n05> module use -a /nasa/modulefiles/testing
PBS cb12n05> module load nvhpc/23.11
PBS cb12n05> nvaccelinfo
CUDA Driver Version:           12030
Device Number:                 0
Device Name:                   NVIDIA A100-SXM4-80GB
Device Revision Number:        8.0
Global Memory Size:            84986691584
Number of Multiprocessors:     108
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
....
Max Threads Per SMP:           2048
Async Engines:                 5
Unified Addressing:            Yes
Managed Memory:                Yes
Concurrent Managed Memory:     Yes
Preemption Supported:          Yes
Cooperative Launch:            Yes
Default Target:                cc80
```

Lower level details of each microarchitecture are encapsulated in its compute capability. The compute capability of a GPU is represented as X.Y, where X is a generation number and Y is a version in that generation. For example, the compute capability of a A100 and H100 is 7.0, 8.0, and 9.0, respectively. This is reflected in the last line of the nvaccelinfo output, as shown in the sample output for an A100 GPU above, cc80.

Details of each compute capability are shown in two tables: (1) Feature Support per Compute Capability and (2) the Technical Specifications per Compute Capability.

A compute feature is introduced with a compute architecture with the intention that the feature will be available on all subsequent architectures. For example, the features "Tensor Cores" and "Mixed Precision Warp-Matrix Functions" were introduced starting with compute capability 7.x. The feature "Hardware-accelerated memcpy_async" was introduced in 8.x , and "Distributed Shared Memory" in 9.0.

An application must be compiled with the proper compute capability setting(s) in order to run on a target GPU architecture.

Compilation with NVCC

NVIDIA CUDA Compiler Driver (NVCC) is used for compiling CUDA C/C++ applications which usually use.cu suffix in their filenames. NVCC is available after loading an nvhpc modulefile. For example,

```
PBS cb12n05> module load nvhpc/23.11
PBS cb12n05> nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Fri_Sep__8_19:17:24_PDT_2023
Cuda compilation tools, release 12.3, V12.3.52
Build cuda_12.3.r12.3/compiler.33281558_0
```

NVCC Compilation Model

NVCC follows a split compilation model to handle the host code and the device code separately. An approximate flow is as follows:

- Preprocesses and separates the device code from the host code

- Parses the device code and the host code separately
- Compiles the device code using proprietary NVIDIA compilers and assembler
- Compiles the host code using a C/C++ host compiler (for example, gcc, g++ under /usr/bin)
- Embeds the compiled device code as fatbinary images in the host object file
- Links specific CUDA runtime libraries

The actual compilation trajectory is quite elaborate. Refer to The CUDA Compilation Trajectory for more information. You can also get some ideas by adding the nvcc flags --keep or -keep to keep all intermediate files that are generated during internal compilation steps and/or --verbose or -v to list the compilation commands generated by nvcc.

Note: NVCC provides the flexibility of choosing a different host compiler through the flag --compiler-bindir or --ccbin followed by the path and name of the compiler, for example, --ccbin /nasa/intel/..../bin/intel64/icc. If it produces an error stating the compiler version is not supported (such as gcc > version 12) and you want to proceed anyway, you can use the flag --allow-unsupported-compiler.

compute_XY, sm_XY and Portability Across Different Compute Capabilities

A complexity in compiling the device code is described in GPU Compilation. In short, NVCC is based on the LLVM open source compiler infrastructure and it compiles the device code in two stages:

1. Converts the device code (.cu) to intermediate PTX instructions (with suffix .ptx).

   Note: PTX stands for Parallel Thread Execution. A .ptx code, in ASCII format, is similar to an assembly code.

   This stage is controlled by the nvcc compiler flag --gpu-architecture <arch> (with a short-hand notation of -arch) where <arch> is a **virtual GPU architecture** which is specified with features supported by a GPU (as seen in the Feature Support per Compute Capability Table) but not its target-specific instruction sets.

   Some of the virtual GPU architectures are: compute_50, compute_60, compute_70, compute_80, compute_90. Run the command nvcc --list-gpu-arch to see a complete list supported by the nvcc compiler.

2. Optimizes and translates the .ptx file to generate a CUDA binary (.cubin) containing target-specific instructions for a real GPU architecture.

   This stage is dictated by the nvcc compiler flag --gpu-code <code> (with a short-hand notation of -code) where <code> specifies the name of the **real GPU architecture** of a NVIDIA GPU to assemble and optimize the PTX for.

   The real GPU architectures are represented as sm_XY such as sm_50, .., sm_90, in CUDA. Run the command nvcc --list-gpu-code to see a complete list supported by the nvcc compiler.

   This second stage can be performed during compile time or Just-in-Time (JIT) during run time.

   - **Compile Time**

     To generate a target-specific (for example, a GPU with compute capability 8.0) .cubin during compile time, use these flags:

     ```
     nvcc x.cu --gpu-architecture=compute_80 --gpu-code=sm_80
     ```

     or

     ```
     nvcc x.cu -arch=compute_80 -code=sm_80
     ```

     or

     ```
     nvcc x.cu --gencode arch=compute_80,code=sm_80
     ```

     Note: In this example, the intermediate .ptx will be discarded and the resulting executable containing a .cubin image only targets a GPU with compute capability 8.0. It will not run on GPUs with compute capability 7.0 or 9.0.

   - **JIT Compilation During Run Time**

     To delay the generation of a target-specific .cubin, the intermediate .ptx file will be included in the executable. During runtime, the current architecture is detected and the compiler builds the appropriate .cubin JIT (thus causing a start-up delay) for execution. This is allowed as long as the detected architecture has a compute capability identical to or newer than the specified virtual GPU architecture. Thus, this is a mechanism to **maintain the portability of an executable for newer generations of GPUs**. Using a virtual GPU architecture with a compute capability of 8.0 as an example, this is done using the command:

     ```
     nvcc x.cu --gpu-architecture=compute_80 --gpu-code=compute_80
     ```

     or

     ```
     nvcc x.cu -arch=compute_80 -code=compute_80
     ```

     or

     ```
     nvcc x.cu --gencode arch=compute_80,code=compute_80
     ```

     Note: In this example, the resulting executable will compile JIT and run successfully for GPUs with compute capability 8.0 and 9.0, but not 7.0.

TIP: To alleviate the start-up penalty of running on a GPU with the same compute capability as the specified virtual architecture while maintaining the portability to a newer GPU compute capability, one can include both sm_xy and compute_xy for the --gpu-code flag. For example:

```
nvcc x.cu --gpu-architecture=compute_80 --gpu-code=sm_80,compute_80
```

Note: No space should be used between sm_80 and ,compute_80.

Caveat: An executable built under x86_64 is not compatible with the aarch64 (LLVM-based ARM) operating system. Therefore, you can not JIT build/run an executable on an aarch64 system using an existing executable from x86_64.

Default

When the flags --gpu-architecture and --gpu-code are not specified as in:

```
nvcc x.cu
```

it defaults to

```
nvcc x.cu --gpu-architecture=compute_52 --gpu-code=sm_52,compute_52
```

where the compute capability 5.2 is for the NVIDIA Maxwell GPUs.

Exception for --gpu-architecture Flag

Normally, the option to this flag must be a virtual GPU architecture compute_XY. In the event that no --gpu-code is used, then one can provide a single real GPU architecture sm_XY to the --gpu-architecture flag. For example,

```
nvcc x.cu --gpu-architecture=sm_80
```

is equivalent to

```
nvcc x.cu --gpu-architecture=compute_80 --gpu_code=sm_80,compute_80
```

Fatbinaries

Another way to increase portability across GPUs with different compute capabilities is to create a fatbinary which includes multiple .cubin images and a .ptx image. For example,

```
nvcc x.cu --gpu-architecture=compute_70 --gpu-code=sm_70 \
        --gpu-architecture=compute_80 --gpu-code=sm_80 \
        --gpu-architecture=compute_80 --gpu-code=compute_80
```

will allow the CUDA driver to select the most appropriate .cubin according to the current native architecture (a V100 or a A100). If no appropriate .cubin is present for the current architecture (e.g., a H100 with compute capability 9.0), the .ptx will be loaded and be compiled JIT (for running on the H100).

WARNING: Fatbinaries built on x86_64 systems can not be ported to H100 used in GH200 Grace Hopper Superchip where the Grace CPU uses Arm architecture (aarch64).
TIP: You can use the cuobjdump utility (after loading a nvhpc module) to detect the available .cubin or .ptx in an executable. For example:

```
% module load nvhpc/23.11
% cuobjdump your_executable
```

References

- NVIDIA CUDA Compiler Driver NVCC
- CUDA C++ Programming Guide
- CUDA C++ Best Practice Guide

NVIDIA Nsight Systems is a low-overhead statistical sampling profiler with tracing features to check overall performance of your GPU applications. The Feature Table in Get Started with Nsight Systems provides a glimpse of its capabilities, including but not limited to: profiling both the CPUs and the GPUs; investigating CPU-GPU interactions; tracing CUDA Application Programming Interface (API) and CUDA libraries; correlating API and GPU workload; and identifying GPU idle and sparse usage.

Nsight Systems provides both a command line interface (CLI), called nsys, and a graphical user interface (GUI), called nsys-ui. Using the CLI facilitates a quick way to collect performance data on a target (the CPU, the GPU, or both). The collected data can then be copied to any system where Nsight Systems is installed for analysis through the Nsight Systems CLI and/or GUI.

Note: Starting with Nsight Systems version 2024.4, no source code line number is available in the profiling data.

This article provides basic information primarily to help you use the Nsight Systems CLI on NAS systems. The examples use the Cabeus front ends (CFEs) for GPUs with x86_64-based CPUs (mil_a100).

gh200 nodes: If you are using GPUs with aarch64-based CPUs (gh200), please be aware of the following differences from use of the GPUs with x86_64-based CPUs:

- Access to these nodes is via the Cabeus Grace Hopper front ends, cghfe01 or cghfe02.
- The available nvhpc modulefiles may be different.
- The location of the Nsight Systems Python scripts is different.
- HPE MPT is unavailable.

Loading a nvhpc Modulefile

Nsight Systems is included in the NVIDIA HPC Software Development Kit (SDK). To access the kit, load one of the HPC SDK modulefiles on a NAS GPU compute node either through an interactive PBS job or a batch job script. The example below starts an interactive PBS session on a mil_a100 node through a qsub command from a CFE (cfe01 or cfe02), loads an nvhpc module, checks the version of nsys, prints its help page, and returns info about the system's profiling environment:

```
cfe% qsub -I -q gpu_devel -lselect=1:ngpus=1:model=mil_a100
a_GPU_node> source /usr/local/lib/global.profile  # if using bash
a_GPU_node> module use -a /nasa/modulefiles/testing
a_GPU_node> module load nvhpc/24.3
a_GPU_node> nsys --version   # or nsys -v
a_GPU_node> nsys --help      # or nsys -h
a_GPU_node> nsys status -e
```

Notes:

- All of the current NAS NVIDIA GPU nodes use a CUDA driver version =/> 535.104.12, which uses driver 535 or newer to improve GPU profiling stability.
- Use the latest available nsys version, which includes more features, by loading the latest available nvhpc modulefile.

Profiling an Application

- For Single or Multi-Threaded non-MPI Applications:

  nsys profile [profile switches] application [application options]

- For Single-Node MPI Applications:

  nsys profile [profile switches] mpirun [mpirun options] application [application options]

- For Multi-Node MPI Applications:

  mpirun [mpirun options] nsys profile [profile switches] application [application options]

All available profile switches (including their description, possible parameters and default setting) are listed in theNsight Systems User Guide or through the nsys -h profile or nsys profile -h help page.

Profiling a Non-MPI Application

Note: If you want to profile an MPI application, see the Profiling and Analyzing MPI Applications section.

The simplest way to profile an application is by running:

GPU_node> nsys profile *your_application*

This will launch *your_application*, start collection immediately, and end collection when*your_application* stops. It will also trace CUDA, OpenGL, NVTX, and OS runtime libraries APIs, and collect CPU sampling information and thread scheduling information. At the end, a profiling output file with the suffix .nsys-rep is generated.

Note: The NVIDIA Tools Extension (NVTX) library is a set of functions that a developer can use to annotate the source code for profiling time line, customizing their appearance and assigning names to resources.

By default, the first profile output filename in a directory is report1.nsys-rep. A new output generated in the same directory will be named report2.nsys-rep, followed by report3.nsys-rep, and so on. You can add the option --output=filename (long notation, use of "=" is optional) or -o filename (short notation) to name the output file filename.nsys-rep. You can also add the option --force-overwrite=true or -f true to force overwriting existing *xxx*.nsys-rep with same output filename.

Instead of tracing the default APIs (CUDA, OPENGL, NVTX, OS Runtime), using -t or --trace switch allows you to choose what API(s) to trace. To keep the overhead small, try to narrow down to the set of features you specifically want to trace and only trace a smaller number of iterations. See the nsys profile -h help page for possible -t API options.

For example:

GPU_node> nsys profile --trace=cublas,openacc *your_application*

See the Example Single Command Lines section of the Nsight Systems User Guide for many more nsys profile examples.

Generating Profiling Statistics Reports

Nsight Systems includes many python scripts that read data from a SQLite database to generate various reports. You can find these scripts in (for example) the /nasa/nvidia/hpc_sdk/toss4/Linux_x86_64/24.3/profilers/Nsight_Systems/host-linux-64/reports directory. Therefore, the *xxx*.nsys-rep file needs to be exported to a *xxx*.sqlite file first before reports can be generated.

There are a few choices:

- **Profiling and generating statistics summary in one command**

  GPU_node> nsys profile --stats=true *your_application*

  This will profile your_application to create an .nsys-rep file and also generate an SQLite database (.sqlite). Furthermore, statistics summary reports in text format will be shown to the terminal (when using interactive PBS session) or the PBS standard output file (when using batch job).

- **Generating the SQLite database and the statistics summary in text format from an existing .nsys-rep file**

  nsys stats *xxx*.nsys-rep

  In this case, nsys will look for an accompanying SQLite file and use it. If no SQLite file exists, one will be exported and created.

- **Generating the statistics summary in text format from an existing .sqlite file**

  nsys stats *xxx*.sqlite

Notes:

- Post-processing an existing *xxx*.nsys-rep or *xxx*.sqlite file can be done on a GPU node or a NAS Cabeus front end (cfe01 or cfe02).
- By default, the statistics summary reports (such as nvtx_sum, osrt_sum, cuda_api_sum, etc.) generated are a subset of all available summary reports. See the --report section of the nsys stats -h printout for more information. In addition to summary reports, there are a few tracing reports (such as cuda_api_trace, cuda_gpu_trace, cuda_kern_exec_race, mpi_event_trace, etc.) one can generate. For a complete list, see nsys stats --help-report printout. To generate a tracing report for, say, cuda_api_trace, perform the following:

  nsys stats --report=cuda_api_trace *xxx*.nsys-rep
  or
  nsys stats -r cuda_api_trace *xxx*.sqlite

Identifying Optimization Opportunities

The nsys command option analyze can be used to get a high-level diagnosis of possible issues and suggestions for performance improvement in the areas of synchronization, data movement, overlap and parallelization.

nsys analyze *xxx*.nsys-rep
or
nsys analyze *xxx*.sqlite

Using the Nsight Systems GUI

For more detailed analysis of your application (such as looking into the timeline and correlation of events happening on different CPU cores, GPU devices, or threads), use the Nsight Systems GUI nsys-ui. For example,

nsys-ui *xxx*.nsys-rep

To learn and gain experience with nsys-ui, watch the videos listed in the reference section below.

Notes:

- Using a Cabeus front end (cfe01 or cfe02) through aVNC session is recommended for launching the Nsight Systems GUI.
- Beyond the use of Nsight Systems for application-level overall performance analysis, use Nsight Compute for detailed CUDA kernel performance analysis and seek optimization opportunities in GPU utilization, kernel implementation, and memory access.

Profiling and Analyzing MPI Applications

Nsight Systems supports the use of OpenMPI. The description here should work for OpenMPI.

- Single-node MPI applications

  nsys profile [profile switches] mpirun [mpirun options] application [application options]

  will generate a single xxx.nsys-rep file, which contains profiling data from all MPI ranks.

- Multi-node MPI applications

  mpirun [mpirun options] nsys profile [profile switches] application [application options]

  will generate one *xxx*.nsys-rep for each MPI rank. For example, if the directory already has a report1.nsys-rep file, then by default profiling a 4-rank MPI run will generate report2.nsys-rep, report3.nsys-rep, report4.nsys-rep, and report5.nsys-rep.

  In order to better correlate each .nsys-rep file with the corresponding MPI rank, it is recommended to add the OpenMPI rank identifier OMPI_COMM_WORLD_RANK into the .nsys-rep files. For example,

  For bash:

  mpirun [mpirun options] nsys profile -o application_%q{OMPI_COMM_WORLD_RANK} application [application_options]

  For csh:

  mpirun [mpirun options] nsys profile -o application_%q\{OMPI_COMM_WORLD_RANK\} application [application options]

  TIP: Not all OpenMPI modulefiles at NAS are PBS-aware. When running across nodes, you may need to use:

  mpirun -np *X* -npernode *Y* --hostfile $PBS_NODEFILE -x PATH -x LD_LIBRARY_PATH nsys profile .... or mpirun -np *X* --map-by ppr:Y:node --hostfile $PBS_NODEFILE -x PATH -x LD_LIBRARY_PATH nsys profile ....

  where *X* is the total number of MPI ranks and *Y* is the number of MPI ranks on each node.

  This will generate application.0.nsys-rep, application.1.nsys-rep, application.2.nsys-rep, and application.3.nsys-rep for a 4-rank MPI run.

  The -o option of nsys profile also allows the use of %h and %p to include the hostname and process id into the .nsys-rep output filename.

  To analyze multiple .nsys-rep files from multiple MPI ranks using Nsight Systems GUI nsys-ui in a single 'Timeline View', perform the following:

  - Start nsys-ui
  - From the File pulldown, select New multi-report view. You should then see:

  - Click the icon indicated by the red arrow to select and add multiple *xxx*.nsys-rep files one by one. After the first file is added, the **Apply** button indicated by the blue arrow will be enabled.
  - After all files you want to analyze together have been added, click **Apply**.

  WARNING: There may be too much data collected from all of the MPI ranks. Consider profile only a single or a few MPI processes to reduce the amount of data to be analyzed. See Profile a Single Process or a Subset of Processes for more information.

Use of Nsight Systems with HPE MPT to profile GPU applications may work but is not guaranteed. To proceed, load anvhpc-nompi modulefile followed by mpi-hpe/mpt. For example:

```
source /usr/local/lib/global.profile    # if using bash
module use -a /nasa/modulefiles/testing
module load nvhpc-nompi/24.3
module load mpi-hpe/mpt
```

Besides setting the CUDA_VISIBLE_DEVICES variable properly, you will also need to set two environment variables before launching HPE MPT mpiexec.

For bash:
```
export MPI_SHEPHERD=1
export MPI_USE_CUDA=1
```

mpiexec -perhost *Y* -np *X* nsys profile -o application_%q{MPT_MPI_RANK} application ..

For csh:
```
setenv MPI_SHEPHERD 1
setenv MPI_USE_CUDA 1
```

mpiexec -perhost *Y* -np *X* nsys profile -o application_%q\{MPT_MPI_RANK\} application ..

Note: The option -perhost can be replaced with -ppn. Both -perhost and -ppn must come before the -np option.

References

- Get Started With Nsight Systems
- Nsight Systems User Guide - Profiling from the CLI
- CUDA Pro Tip: Generate Custom Application Profile Timelines with NVTX
- Video recording: Performance Analysis with the Nsight Systems Timeline
- Video recording: What the Profiling is Telling You: How to Get the Most Performance out of Your Hardware

This article introduces the unified memory feature available in all of the NAS GPU nodes and describes how to take advantage of this feature to simplify GPU programming.

Introduction

In a heterogeneous compute system containing a CPU host and one or more GPU devices—connected either by PCI-Express bus (as in the NAS mil_a100 GPU nodes) or by NUMALink (as in the gh200 nodes with a Grace Hopper Superchip)—the system host memory and the GPU device memory are physically distinct and separated, as shown in the left graph of the image below. Typically, the device memory is smaller but faster than the host memory.

Without the CUDA Unified Memory feature, programming for running on these heterogeneous nodes requires that any data shared between the CPU and the GPU(s) are allocated on **both** the host memory and the device memory, and explicitly copied between them within the application. This makes programming for such heterogeneous systems error-prone and complicated.

Note: This is demonstrated in this NVIDIA Technical Blog post showing a sample CUDA C program with explicit data movement.

Fortunately, the Unified Memory feature was introduced with CUDA 6 and improved in later CUDA versions. Unified Memory has two key features:

- It creates a pool of virtual memory (shown in the right graph below) that is a single memory address space accessible through a single pointer from either the CPU or the GPU(s). Mapping of the virtual address to the physical address—invisible to the programmer—is tracked through page table(s).

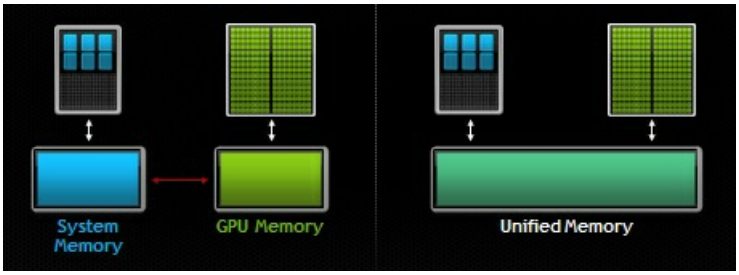


Image credit: NVIDIA

- It automatically takes care of migrating memory pages to the memory of the accessing processor to enable local access through support from software (CUDA driver, CUDA runtime, and operating system) and/or hardware (such as the page migration engine, hardware access counters, and hardware-accelerated memory coherency).

The combination of these two key features greatly simplifies memory management for heterogeneous systems, makes the source code cleaner and more concise, and reduces programming hurdles for many users. Specifically, when Unified Memory is used in the application, there is no need to allocate data twice (once on host memory and once on device memory), and explicit data movement is eliminated, as demonstrated in this NVIDIA Technical Blog post showing a simplified CUDA C program with Unified Memory.

For heterogeneous systems prior to the Grace Hopper Superchip, Unified Memory is also referred to as Managed Memory, and is only applicable to data that is dynamically allocated on the heap. For systems with the Grace Hopper Superchip, Unified Memory is sometimes referred to as Full Unified Memory. Variables in the heap (i.e., dynamically allocated), stack (i.e., local), and data (i.e., global) regions of an application's memory layout can all be placed in Full Unified Memory.

Read this section to learn more on some history and development of Unified Memory.

Using Unified Memory

The main benefit of Unified Memory is simplicity and ease of use. However, it does not necessarily provide the best performance. Depending on the data access pattern, explicit bulk data transfer without Unified Memory or using CUDA data prefetching API or memory usage hints APIs on top of Unified Memory may outperform the default Unified Memory mode (i.e., with just the implicit on-demand data movement).

When writing/porting an application for/to the GPUs, take advantage of Unified Memory to help you focus on developing algorithms and parallelizing your application first. You can work on data movement for performance tuning later in your development.

Use of Unified Memory with a few GPU programming models is briefly described here:

CUDA C/C++ applications

- Using CUDA Managed Memory

  This approach can be used on all NAS heterogeneous systems, including those with the Grace Hopper Superchip.

  In the application, use the CUDA memory management API `cudaMallocManaged()` to allocate data to be shared by the CPU and GPU(s) **once**; after the CUDA kernel is launched, use the CUDA device management API `cudaDeviceSynchronize()` to trigger the CUDA runtime to make data available to the CPU code again; at the end, use `cudaFree()` to deallocate. For example:

```
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
...
add<<<1, 1>>>(N, x, y);
cudaDeviceSynchronize();
...
cudaFree(x);
cudaFree(y);
```

Code segment taken from NVIDIA publications.

Note: The physical memory (host memory or device memory) that cudaMallocManaged() allocates data to is based on first touch, starting with Pascal GPU [compute capability 6.x].

To learn more, compare sample codes provided in this sample CUDA C program with explicit data movement and this simplified CUDA C program with Unified Memory.

Note: You can use CUDA __managed__ variables as well as the cudaMallocManaged() API. For example,

```
__managed__ int value;
```

- Using Full CUDA Unified Memory

  This approach can only be applied to systems containing the Grace Hopper Superchip.

  For applications written in C, simply use malloc() and free() to allocate and deallocate on the Unified Memory. For C++, use the new and delete operators. cudaDeviceSynchronize() should still be used after launching a CUDA kernel.

  By default, first touch policy is used when allocating data.

Use of CUDA data prefetching or memory usage hints APIs on top of Unified Memory may help to improve performance.

Compile the application with the NVIDIA nvcc compiler, which is described in Compiling CUDA Applications for Different Generations of NVIDIA GPUs at NAS. Add the compiler flags --gpu-architecture and --gpu-code to target certain GPU architecture identified with its compute capability. For example:

- For A100: --gpu-architecture=compute_80 --gpu-code=sm_80
- For H100 in the Grace Hopper Superchip: --gpu-architecture=compute_90 --gpu-code=sm_90

Read the **Unified Memory Programming** chapter in NVIDIA's CUDA C++ Programming Guide to learn more.

CUDA Fortran Applications

CUDA Fortran is an analog of CUDA C/C++.

Variables in CUDA Fortran use one of the following attributes to declare in which memory the data is allocated: device, managed, unified, pinned, constant, shared, or texture . Among them, the managed attribute refers to Managed Memory for dynamically allocated variables in heap. For example, A and B below are to be allocated in Managed Memory:

```
real, managed, allocatable, dimension(:,:) :: A, B
```

The unified attribute refers to Full Unified Memory for all variables (heap, stack, and global). This attribute is available starting with the NVIDIA HPC compiler 24.3 release and it should only be used on systems that support it—for example, the Grace Hopper Superchip.

CUDA Fortran, like CUDA C/C++, supports the use of the data prefetching and memory usage hints APIs for performance tuning.

Compile the application with NVIDIA HPC compiler, nvfortran. Add the flag -Minfo to check what the compiler does for your application. To target a specific GPU architecture, add the flag:

- For A100: -gpu=cc80
- For H100 in Grace Hopper Superchip: -gpu=cc90

OpenACC applications

OpenACC applications use the data directive and its clauses (create, present, copy, copyin, and/or copyout) to reserve, check presence of memory on the device, and define explicit data movement, as shown in this sample OpenACC Jacobi C code with explicit data movement.

The data region directive and clauses are not needed if the application intends to use Unified Memory, as shown in thissimplified OpenACC Jacobi C code with compiler-assisted use of Unified Memory.

If Unified Memory is to be used, explicit data region directive and clauses in the application will be treated as optimization hints to indicate preferable data placement.

Unlike CUDA C/C++, there are currently no OpenACC directives/clauses for data prefetching. However, it is possible to use CUDA prefetching functions in OpenACC applications.

To use Managed Memory or Full Unified Memory, OpenACC applications use the NVIDIA HPC nvc, nvc++, nvfortran compiler/link-time flags -gpu=managed or -gpu=unified:

- -gpu=managed

  From the man page: *Allocate any **dynamically allocated data** in CUDA Managed memory. This option must appear in both the compile and link lines. This option may not be used with -gpu=pinned.*

  The compiler replaces calls to malloc/free in C, new/delete in C++, and allocate/deallocate in Fortran with the CUDA driver APIs cuMemAllocManaged() and cuMemFree().

  Note: Directives are still required for moving global or stack data.

- -gpu=unified

  From the man page: *Compile for CUDA Unified memory capability, where system memory is accessible from the GPU.**All variables** in the CPU code are accessible from the GPU without explicit directives or attributes. This mode utilizes system and Managed Memory for dynamically allocated data unless explicit behavior is set through -gpu=[no]managed. This option must appear in both the compile and link lines. This option may not be used with -gpu=pinned.*

  Note: This flag can be used for the Grace Hopper Superchip, but not the A100 GPUs. In addition, it is only available in newer versions of the NVIDIA HPC compiler starting with nbhpc/23.11.

Note: Use the -acc[=gpu] flag to compile OpenACC directives for parallel execution on the GPU; use -Minfo=all to obtain information on what the compiler decides to do or not do for your application; and use -gpu=cc*XY* where *XY*=70, 80, and 90 to target a certain GPU architecture.

Standard Language Based Applications

ISO C++ standard algorithms or the ISO Fortran Do Concurrent construct are offloaded to the GPUs for parallel execution when compiled with the NVIDIA HPC nvc++, nvfortran compiler flag -stdpar[=gpu]. Data movement between host memory and device memory is done implicitly and automatically under the control of CUDA Unified Memory.

- On hetereogeneous systems prior to Grace Hopper Superchip, -stdpar=gpu implies -gpu=managed.
- On Grace Hopper Superchip systems, -stdpar=gpu implies -gpu=unified.

Adding the -Minfo compiler flag is recommended to check what the compiler does for your application. You can also add-gpu=cc*XY* where *XY*=70, 80 or 90, to target a specific GPU architecture.

Additional Background Information About Unified Memory

Unified Memory was developed on top of an earlier CUDA feature called Unified Virtual Addressing (UVA, introduced in CUDA 4) which provides a single virtual memory address space for all memory in the system. UVA allows:

- GPU accessing zero-copy memory (i.e., pinned, locked, not-allowed-to-be-paged-out host memory allocated through the CUDA API cudaMallocHost or cudaHostAlloc) directly without needing to call memcpy.

  Note: The speed of zero-copy access is limited by the interconnect between the CPU and the GPU. In addition, since zero-copy memory is located on the host, it's a non-local access by the GPU.

- cudaMemcpy API to be simplified where the exact location of the input and output parameters does not have to be specified. That is, you can use cudaMemcpyDefault instead of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, and cudaMemcpyDeviceToDevice as the transfer kind (i.e., fourth parameter of cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)).

In 2013, with CUDA 6, NVIDIA introduced software support for a limited form of Unified Memory, applicable to the Kepler GPUs [compute capability 3.x] and Maxwell GPUs [compute capability 5.x], which—in addition to having the UVA feature—is able to allocate data up to the size of the GPU memory. Using the CUDA API cudaMallocManaged and upon kernel launch, managed data is moved en-masse—even for data that the kernel may not appear to explicitly touch—to device memory. Any CPU access of the data would trigger a data transfer from device memory to CPU memory.

In later versions of CUDA and newer generations of NVIDIA GPUs—Pascal [compute capability 6.x] and later, unless specified—more and more software and/or hardware features were added for performance tuning or even easier programming. Some of these features are:

- GPU hardware Page Migration engine.

  This enables GPU page fault management and on-demand (page-by-page) migration.

- Concurrent access to the Managed Memory by both CPU and GPU.

  Note: On Kepler GPUs, accessing data in Managed Memory by the CPU while it is in use by a GPU kernel results in the application aborting with segmentation fault. On Pascal and later GPUs, page fault is issued to move data accordingly instead of segmentation fault.

- The cudaMemPrefetchAsync() data prefetching API.

  This API is used to prefetch data ahead of its use by the GPU and helps to eliminate the overhead of servicing page fault page-by-page and updating page table if there are many page faults.

- Explicit memory usage pattern hints API cudaMemAdvise(...,hints,..) that specifies placement and access policies on regions of managed data

Hints may be cudaMemAdviseSetReadMostly, cudaMemAdviseSetPreferredLocation, cudaMemAdviseSetAccessedBy, etc.

This helps to reduce data movement.

- Larger (x86_64: 47 bits, arm64: 48 bits, power: 49 bits) virtual memory address space and memory over-subscription.

  This allows using more memory than the size of GPU memory for scaling to larger problem sizes.

  When GPU memory is exhausted and page migration from the CPU to GPU is needed, some pages in the GPU memory are evicted in order to accommodate other pages to be migrated into the GPU.

- Hardware access counters to track remote access to pages.

  Starting with the Volta GPUs, hardware access counters are included to allow more accurate migration of memory pages to the processor that accesses them more frequently.

- High-bandwidth, low-latency, memory-coherent NVLink Chip-2-Chip (C2C) interconnect in the Grace Hopper Superchip.

  This enables applications to oversubscribe the Hopper GPU memory (96 GB of HBM3) and directly utilize the Grace CPU memory (480 GB of LPDDR5X) with low latency and high bandwidth (upto 900 GB/s).

  The hardware accelerated memory coherency in the Grace Hopper Superchip uses an Address Translation Service (ATS), which enables the CPU and GPU to share a single per-process page table instead of two separate page tables (one for the host and the other for the device). Translation between the virtual address and the physical address is accelerated by hardware. This capability avoids page faulting and minimizes page migration to the requesting processor, and allows the CPU and GPU threads to concurrently and transparently access both the CPU and GPU memory. In addition, the NVLink-C2C hardware coherency enables the Grace CPU to cache GPU memory at cache-line granularity. When contention occurs while different processor access the same cache line, only that cache line needs to be exchanged instead of a whole page. If different processors access different cache lines in the same page, there is no contention.

- Software-based Heterogeneous Memory Management (HMM) capability.

  This was introduced in 2023 to further simplify GPU programming for systems with PCI-E interconnect between CPU and GPU. HMM mimics the Grace Hopper Superchip's hardware accelerated memory coherency through software, and provides the same level of programming simplicity.

  HMM requires (1) CUDA 12.2; (2) CUDA driver 535 or newer; (3) Linux kernel 6.1.24+ or newer; (4) Turing, Ampere, Ada Lovelace, Hopper or newer GPUs; and (5) a 64-bit x86 CPU. Currently, HMM is not available on NAS GPU nodes with x86-based CPU, because the Linux kernel used is 4.18.

References

- Unified Memory in CUDA 6
- Unified Memory for CUDA Beginners
- Maximizing Unified Memory Performance in CUDA
- Improving GPU Memory Oversubscription Performance
- Simplifying GPU Programming for HPC with NVIDIA Grace Hopper Superchip
- Simplifying GPU Application Development with Heterogeneous Memory Management

This article provides basic hardware architecture information to help you get familiar with the NVIDIA A100 and H100 graphics proessing units (GPUs) used in the NAS GPU nodes (mil_a100 and gh200). Refer to the official guides from NVIDIA listed in the Reference section for detailed information.

Note: The H100 described in this article is integrated in a Grace Hopper Superchip (GH200) which includes an ARM-based Grace CPU and a H100 GPU.

Be aware that NVIDIA adds new features with each new generation of GPU and some of the descriptions here may not apply to GPUs newer than the H100.

Compute Hierarchy

The following describes a GPU, followed by a Streaming Multiprocessor (SM) inside a GPU, and lastly, the CUDA cores and Tensor cores inside an SM. In addition, we introduce the concepts of threads and warps needed for GPU programming.

GPUs

A GPU is also referred to as a coprocessor or a device. A GPU node (for example, the NAS mil_a100, gh200) may contain only 1 or multiple GPUs. A GPU:

- Connects to a CPU (also referred to as a host) via:
  - A100: PCIe 4.0 x16 with 64 GB/s bi-directional bandwidth.
  - H100 in GH200: NVLink 4.0 used as NVLink Chip-2-Chip (C2C) with 900 GB/s bi-directional bandwidth (7X that of PCIe 5.0 x16 with 128 GB/s).

- Has multiple NVLinks for connection with other GPUs or other hardware (NVLink is a high-bandwidth, low-latency shared memory interconnect):
  - A100: 12 NVLink 3.0 links, 600 GB/s bi-directional bandwidth
  - H100: 18 NVLink 4.0 links, 900 GB/s bi-directional bandwidth

- Includes many Streaming Multiprocessors (SMs):
  - A100: 108 SMs
  - H100: 132 SMs

- Includes L2 cache shared among the SMs:
  - A100: 40 MB (~379 KB/SM)
  - H100: 60 MB (~465 KB/SM)

- Includes device memory shared among the SMs:
  - A100: HBM2e 80 GB (~758 MB/SM)
  - H100: HBM3 96 GB (~745 MB/SM)

Streaming Multiprocessors

A Streaming Multiprocessor (SM) is an individual processing unit in a GPU. The SMs are responsible for executing threads in parallel. Each SM includes:

- Many single precision (FP32) CUDA cores, double precision (FP64) CUDA cores, and Tensor cores capable of mixed precision operations.
  - A100: 64 (single), 32 (double), 4 (Tensor)
  - H100: 128 (single), 64 (double), 4 (Tensor)
- Many 32-bit (4 bytes) registers:
  - A100: 65536, total size 256 KB
  - H100: 65536, total size 256 KB

  The large number of registers enable a GPU to run a very large number of threads.

- A combined L1 data cache/Shared Memory shared among threads running on the cores:
  - A100: 192 KB
  - H100: 256 KB

As an example, this diagram from a NVIDIA white paper shows the configuration of an SM.



The function and capability of an SM is more in line with a single core in a multi-core CPU.

- A CPU core containing vector registers (such as the xmm, ymm and zmm registers of x86 microprocessors) executes single instruction multiple data (SIMD) instructions on multiple data in vectors simultaneously. For example, an xmm, ymm, or zmm register can hold four, eight or sixteen single-precision floating-point numbers, respectively, for use by floating point functional units.
- An SM, with many scalar cores and registers, executes single instruction multiple threads (SIMT) instructions in a thread block in parallel. Multiple SMs may work together on executing many thread blocks.

With CUDA, launching a kernel (i.e., a function to be executed in parallel on a GPU) with

```
kernel_name<<<number_of_blocks, number_of_threads_per_block>>>(...);
```

results in executing the kernel once for every thread in every thread block.

With many SMs per GPU, and many cores per SM, one can run thousands of threads on thousands of cores simultaneously to hide memory latency.

Threads in CUDA

A thread is a stream of instructions and data assigned to one single CUDA core. It is extremely lightweight with very little overhead.

A thread block, also known as a cooperative thread array, is a group of threads that can cooperate with one another by synchronizing their execution and efficiently sharing data through a low latency shared memory.

In CUDA, threads are organized into thread blocks. Many thread blocks are organized into a grid. Each thread block is executed by one SM and it cannot be migrated to other SMs. Prior to the introduction of thread block clusters in the H100, thread blocks were required to execute independently, and thus can execute in any order.

The maximum number of threads per thread block is 1,024, as seen in output after running the `nvaccelinfo` command (available after loading an nvhpc module) on a GPU. This limit is imposed by the resources available in an SM to be shared by all threads in a thread block. The number of thread blocks in a grid is dictated by the size of the data to be processed.

Within a CUDA kernel, a thread is identified through a combination of:

blockIDx.x, blockIDx.y, blockIDx.z : the indices of a block in a 3D grid
blockDim.x * blockDim.y * blockDim.z : the number of threads in a 3D block
threadIdx.x, threadIDx.y, threadIDx.z : the indices of a thread in a 3D block

For example:

```
i = blockIDx.x * blockDim.x + threadIdx.x ;
j = blockIDx.y * blockDim.y + threadIdx.y ;
k = blockIDx.z * blockDim.z + threadIdx.z ;
```

Note: The H100 introduces a new feature called thread block clusters, which extends the CUDA programming hierarchy to include four levels: threads, thread blocks, (optional) thread block clusters, and grids. The thread block clusters enable multiple thread blocks running concurrently across multiple SMs to synchronize and collaboratively fetch and exchange data by accessing other SM's shared memory. This new feature is known as 'distributed shared memory (DSMEM)' because the shared memory's virtual address space is logically distributed across all the blocks in the cluster.

Warps in CUDA

A warp is a group of 32 threads within a thread block that executes the same instruction simultaneously. These threads are scheduled in batches by a warp scheduler to execute on an SM.

The size of a thread block is set by an application. However, it should be a multiple of 32 due to the warp scheduling. A common choice of a thread block size is 256.

Note: In concept, executing a warp (32 threads) on multiple cores of an SM is similar to executing vector instructions (using xmm/ymm/zmm registers on 4/8/16 data elements in a vector) on a CPU core.

CUDA Cores

CUDA cores perform general-purpose computing like the CPU cores. Some of the main differences between CPU cores and CUDA cores are as follows:

- A CPU core:
  - Is a complex and powerful core.
  - Runs at a higher clock speed,; for example, 2.45 GHz for AMD EPYC 7763, ~3.4 GHz for Arm Neoverse-V2 chip used in NVIDIA Grace CPU.
  - Can switch between multiple tasks such as out-of-order or speculative operations.
  - May have multiple vector registers and is capable of performing vector operations.
  - Has its own L1 cache and maybe also L2 cache.

  Limited parallelism is achieved from a combination of (1) vector operations for each core, and (2) multiple cores per CPU.

- A CUDA core:
  - Is a simple and less powerful core.
  - Runs at a lower clock speed (1410/1980 MHz for A100/H100, respectively).
  - Is designed to do simple and repetitive operations focusing on number crunching, much less on out-of-order or speculative operations.
  - By itself, can only do a scalar operation through the use of scalar registers and integer or floating point functional units.
  - Does not have its own cache (the L1 cache is shared among all cores in an SM).

  Very high parallelism is achieved through thousands of CUDA cores in a GPU working concurrently.

  Note: GPU devices of compute capabilitiy 2.0 and later are capable of single and double precision arithmetic following the IEEE 754 standard and have hardware units for performing fused multiply-add in both single and double precision.

  The compute capability of A100 and H100 is 8.0 and 9.0, respectively.

Tensor Cores

Tensor cores are specialized GPU cores designed to accelerate matrix-multiply-and-add (MMA) operations (D = A x B + C), which are at the core of deep learning/AI computations for AI model training and inference and many HPC applications. The first generation of Tensor cores was introduced in the NVIDIA Volta architecture, followed by the second generation in Turing, third generation in Ampere, and fourth generation in Hopper.

Note: Ampere and Hopper resources—but not Turing—are available at NAS.

In the A100, each 3rd-generation Tensor core performs 8 x 4 x 8 (that is, 256) FP16/FP32 mixed precision FMA operations per clock. A single A100 SM, with 4 Tensor cores, can deliver 4 x 256 = 1024 FP16 FMA operations or 2048 FP16 floating point operations per clock. Besides FP16 precision input, A100 Tensor cores also add support for, among others, FP64, new BrainFloat-16 (BF16), and new TensorFloat-32 (TF32) precision input.

In the H100 4th-generation Tensor cores, FP8 precision input is supported, which provides 2x the throughput of FP16.

Memory Hierarchy

Some noticeable differences between CPU memory hierarchy and GPU memory hierarchy are:

1. The CPU memory hierarchy looks like a pyramid where the faster forms of memory with smaller sizes sit on top and slower ones with larger sizes are below.

   The GPU memory hierarchy does not look like a pyramid. Specifically, as shown below, the L1 cache size can be smaller than the register size.

2. There tend to be more layers of caches in CPUs than GPUs. For example, the AMD Milan and NVIDIA Grace CPU processors all have three levels of caches: L1, L2, and L3.

   On the other hand, the NVIDIA A100 and H100 only have L1 and L2 caches.

These differences speak to the fact that GPUs are designed to maximize throughput instead of minimizing latency. The high throughput is realized via a large number of registers and the use of high bandwidth memory. Instead of minimizing latency, GPUs rely on hiding latency through asychronous SIMT operations.

This section describes different levels of memory in GPUs starting from the fastest to the slowest.

Note: The latencies listed below for different levels of GPU memory are all approximations. They are meant to provide a rough comparison of latencies in GPUs and CPUs.

Registers

A register is a small, on-chip high-speed memory location used for storing data to be immediately used by the processing functional units. A register in an SM is private to a thread. Registers read/write from/to either L1 cache or the shared memory of an SM.

Each register holds a single 4-byte data. With 65,536 registers per SM, the total register size is 256 KB per SM. This size is larger than the L1/shared memory in the SM.

The large number of registers, together with the large number of cores, allow many threads to run in parallel. When some threads are waiting for data from L1/L2/global memory, other threads can be executed. This effectively hides the latency.

L1/Shared Memory

The combined L1 data cache and shared memory is on-chip and is private to each SM.

The size of this combined L1/shared memory varies for different GPUs. For A100 (192 KB), this size is smaller than the size of the registers. For the H100, the size is 256 KB, same as the size of the registers.

The partition between L1 data cache and shared memory is set by an application. However, there is a limit on the configurable shared memory size per SM: 164 KB for the A100 and 228 KB for the H100.

The latency of L1/shared memory is approximately 20 - 30 cycles as reported by independent benchmarking works listed in the Reference section. In comparison, L1 latency in CPUs is roughly five cycles.

Shared Memory

Memory access latency is lower with shared memory (located inside the processor chip) than global memory (located off the processor chip). This makes shared memory ideal for storing frequently accessed data to significantly speed up data retrieval and computation.

In a CUDA application, use __shared__ to declare a variable or array in shared memory for use by a thread block.

All of the data in a shared memory region of a thread block can be accessed by any thread in a thread block. This allows threads in a thread block to communicate and share data with one another. On the other hand, data of a thread block in the shared memory is not visible to threads in other thread blocks.

According to the Tuning Guides of various GPU architectures:

- The A100 supports shared memory capacities of 0, 8, 16, 32, 64, 100, 132, or 164 KB per SM. A single thread block can address up to 163 KB of shared memory.
- The H100 supports shared memory capacities of 0, 8, 16, 32, 64, 100, 132, 164, 196, and 228 KB per SM. A single thread block can address up to 227 KB of shared memory.

To maintain architectural compatability, static shared memory allocation is limited to 48 KB. Usage above 48 KB needs to be dynamic shared memory.

All blocks on an SM can share the physical memory resources provided by the SM.

L1 Data Cache

L1 cache is a buffering space when reading/writing data from/to L2 cache.

In an SM, the space unused as shared memory is left as L1 data cache. The size of an L1 cache line is 128 bytes, which is the size of data for a warp of 32 threads.

L2 Cache

L2 cache is another on-chip cache shared across all SMs. All threads in a block can access L2 cache.

L2 cache keeps data read/written from/to device memory to service requests from SMs in the GPU.

The latency of L2 cache in GPU (shared by SMs) is approximately 200 cycles as reported by independent benchmarking works for A100, listed in the Reference section.

In comparison, latency of L3 cache in CPUs (shared by CPU cores) is approximately 50 - 70 cycles.

Note: Many multi-core CPU architectures include private L1 and L2 cache for each core and shared L3 cache among the cores.

Device Memory

The device memory is off-chip and is made of different generations of High Bandwidth Memory (HBM) for different GPUs. As listed below, HBM provides much higher memory bandwidth compared to Double Data Rate 4th generation (DDR4, ~200 GB/s from 1 socket) or Low-Power DDR5X (LPDDR5X, ~500 GB/s from 1 socket) memory used in CPUs.

- HBM2e for NAS mil_a00: 2039 GB/s
- HBM3 for NAS gh200 nodes: up to 4000 GB/s

Latency of device memory is between about 200 to 1000 cycles, as reported by independent benchmarking work for the A100, listed in the Reference section. Assuming a CUDA core runs at 1.5 GHz, the 200 to 1000 cycles would translate to 133 to 666 nanoseconds (ns).

In comparison, memory latency in CPUs varies between roughly 50 and 200 ns.

The device memory is used as different forms of memory:

- Global memory

  This is the primary storage for data that is shared across multiple blocks of threads.

  Most of the space in the device memory is used as global memory.

- Local memory

  This is for storing data private to each thread that cannot fit into registers.

  The maximum amount of local memory per thread is 512 KB as seen in this NVIDIA article.

- Constant memory

  This is a read-only memory space used for storing global data that do not change while a kernel is running. Its contents can be broadcast to multiple threads in a block. It is shared by all threads and has the lifetime of the application.

  The constant memory is very small in size - 64 KB as seen here.

- Texture memory

  This is for storing texture data used in video rendering.

  The size used as texture memory per SM varies - 28 to 192 KB for A100 and 28 to 256 KB for H100, as seen here.

References

- NVIDIA A100 Tensor Core GPU Architecture (white paper)
- NVIDIA H100 Tensor Core GPU Architecture (white paper)
- Tuning CUDA Applications for Volta
- Tuning CUDA Applications for NVIDIA Ampere GPU Architecture
- Tuning CUDA Applications for Hopper GPU Architecture
- NVIDIA Grace Performance Tuning Guide
- Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking
- Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis
- Floating Point and IEEE 754

CUDA stands for Compute Unified Device Architecture. It is a parallel computing platform and programming model created by NVIDIA in 2006 for accelerating computing on General-Purpose Graphics Processing Units (GPGPUs).

This article provides a high-level view of two of the components in CUDA—the CUDA Toolkit and the CUDA driver—and the compatability between them.

CUDA Toolkit

Everything needed to develop, optimize and deploy applications written in CUDA is included in the CUDA Toolkit. It is released multiple times per year by NVIDIA to provide new features, performance improvement, and critical bug fixes. The Toolkit includes:

A C/C++ Compiler

The NVIDIA CUDA compiler (nvcc) is for compiling applications written in standard C/C++ with CUDA extensions. The CUDA applications, which typically use the suffix .cu, are turned into machine codes by the compiler for running on the GPUs.

Notes:

- You can find many CUDA samples in github.
- The NVIDIA HPC compilers, nvc, nvc++ and nvfortran (rebranded from the PGI compilers) are not part of the CUDA Toolkit. However, both the HPC compilers and nvcc are included in the NVIDIA HPC Software Development Kit (SDK).

GPU-accelerated CUDA Libraries

Besides explicitly incorporating CUDA extensions to the standard C/C++ languages, an application may also utilize some of these pre-built GPU-accelerated Math Libraries included in the CUDA Toolkit:

- cuBLAS: basic linear algebra
- cuFFT: fast fourier transform
- cuRAND: random number generation
- cuSOLVER: dense and sparse direct solver
- cuSPARSE: BLAS for sparse matrix
- cuTENSOR: tensor linear algebra
- CUDA math library: standard mathematical function library

These libraries provide the highest level of abstraction in GPU programming.

The NVIDIA HPC compilers (nvc, nvc++, and nvfortran) provide a flag -cudalib[=cublas|cufft|..] for linking to these CUDA libraries. Check out examples available under the 4_CUDA_Libraries section of the CUDA samples or the directory **/nasa/arm/software/nvhpc/24.11/Linux_aarch64/24.11/examples/CUDA-Libraries** on a NAS GH200 front end system chghfe[01,02].

Besides the Math Libraries, the CUDA Toolkit also includes:

- Thrust, the C++ Parallel Algorithms Library: Allows C++ developers to accelerate operations such as sort, scan, transform, and reduction with just a few lines of code. Thrust, a header-only library, is available under, for example, /nasa/arm/software/nvhpc/24.11/Linux_aarch64/24.11/cuda/12.6/targets/sbsa-linux/include/thrust.
- Communication Library: Consists of MPI (OpenMPI and HPCX-enhanced OpenMPI), NVIDIA Collective Communication Library (NCCL), and NVSHMEM. They are available under, for example, /nasa/arm/software/nvhpc/24.11/Linux_aarch64/24.11/comm_libs.

NVIDIA also provides GPU-accelerated Data Processing Libraries, Deep Learning Core Libraries, and more. However, they are not included in the CUDA Toolkit. Read NVIDIA CUDA-X Libraries for more information.

A CUDA Runtime Library

The CUDA runtime (libcudart.so, cuda_runtime.h, cuda_runtime_api.h) is one of the software layers that can be implicitly or explicitly used in GPU programming. It sits below the user application and GPU-accelerated libraries and above the CUDA driver.

The CUDA runtime API provides developers with a high-level C++ interface for simplified device management, error handling, memory management, stream management, and more. For example, the functions cudaMalloc, cudaMemset, cudaMemcpy, cudaFree are included in the CUDA runtime memory management.

Debugging and Optimization Tools

- CUDA-GDB: A command-line tool for unified CPU and CUDA debugging.
- NVIDIA Nsight tools (A100 and newer GPUs)
  - Nsight Systems (nsys, nsys-ui): a high-level application-wide profiler for GPU and CPU sampling and tracing
  - Nsight Compute (ncu, ncu-ui): an interactive low-level kernel profiler for digging deeper into the CUDA kernels

CUDA Toolkit on NAS Systems

For systems running with the Linux operating system, the CUDA Toolkit is available for either the x86_64 architecture (for example, the NAS mil_a100 nodes) or the ARM architecture (Apple-developed arm64 and LLVM-developed aarch64, such as the NASgh200).

The CUDA Toolkit is included in the NVIDIA HPC SDK. At NAS, multiple versions of the HPC SDK are available as modules (for example, nvhpc/24.11). In an HPC SDK release, there may be one or more CUDA Toolkit versions included. For example, on the GH200 front-end or compute nodes, the available CUDA Toolkit versions and default are:

HPC SDK Module	Available CUDA Toolkit Versions	Default Version
nvhpc/24.3	11.8, 12.3	12.3
nvhpc/24.11	11.8, 12.6	12.6
nvhpc/25.1	11.8, 12.6	12.6

Notes:

- The CUDA Toolkit follows an .X.Y.Z version scheme where .X stands for major version, .Y stands for minor version, and .Z stands for release/patch version. Different major versions are not binary compatible. Same major version but different minor versions are binary compatible but may not be source compatible (i.e., you may have to modify your source code to comply with changes in the APIs introduced in a different monor version).
- .Z is not shown in the CUDA Toolkit versions in the above table.

TIPS:

- After loading a nvhpc modulefile, run the command nvcc --version to see which CUDA version is used.
- Add the -dryrun flag of the nvcc, nvc, nvc++, nvfortran compilers to check which CUDA Toolkit is being used.
- If using the NVIDIA HPC compilers (nvc, nvc++, nvfortran, but not nvcc), you can also explicitly choose a CUDA Toolkit version on any node using the flag -gpu=cudaX.Y (e.g., 12.3) if version X.Y is available in the CUDA Toolkit.
- Build and run your GPU application with the same nvhpc modulefile on the same NAS GPU node type to avoid a potential mismatch between compile time and runtime.

CUDA Driver

Complementary to the CUDA Toolkit whose components are mainly used for building applications, the CUDA driver components are mainly used for running applications.

In order to run a CUDA application, the system should have a CUDA enabled GPU and a NVIDIA driver that is compatible with the CUDA Toolkit that was used to build the application itself.

There are two components in the CUDA driver:

- User-mode driver (libcuda.so, cuda.h): The CUDA driver API provides a low-level interface for applications to target NVIDIA hardware.
- Kernel-mode driver (nvidia.ko): It provides low-level access to the NVIDIA GPU hardware.

CUDA Driver on NAS Systems

Unlike the CUDA Toolkit, which can have multiple versions installed, only one version of the CUDA driver can be installed on a GPU node by a system administrator. You can find the kernel module version installed using either the nvidia-smi or the modinfo command. For example:

```
a_GPU_node% nvidia-smi | grep -i version
| NVIDIA-SMI 610.57.04        KMD Version: 610.57.04     CUDA UMD Version: 13.3

a_GPU_node% modinfo /usr/lib/modules/`uname -r`/extra/nvidia/nvidia.ko | grep ^version
version:        610.57.04
```

Note: No CUDA driver is installed on the CPU-only nodes such as CFE and pbspl4.

WARNING: The CUDA driver version installed on GPU systems is changed from time to time. The driver version may be different on different NAS GPU nodes (mil_a100 and gh200).

CUDA Compatibility

Running a CUDA application requires the system with at least one CUDA-capable GPU and a driver that is compatible with the CUDA Toolkit.

- Each CUDA release has a Toolkit version and a corresponding driver version. For example, Toolkit 12.3 update 1 corresponds to driver 545.23.08, while Toolkit 13.3 update 1 corresponds to driver 610.43.02. See Table 3 in CUDA Toolkit and Corresponding Driver Versions in NVIDIA CUDA Toolkit Release Notes.
- The CUDA driver is backward compatible: an application compiled with an older CUDA Toolkit version can run with a newer driver release.
- Applications built with a new CUDA Toolkit require a minimum older CUDA driver release.
  - Application built with CUDA Toolkit 11.x require CUDA driver >= 450.80.2
  - Application built with CUDA Toolkit 12.x require CUDA driver >= 525.60.13
  - Application built with CUDA Toolkit 13.x require CUDA driver >= 580.65.06

TIP: If you bring an application built on a non-NAS system, check which version of CUDA Toolkit was used to build it and whether the CUDA driver on the NAS GPU nodes you intend to use is compatible with it.

References

- What is CUDA?
- CUDA Toolkit Release Notes
- CUDA Runtime API
- CUDA Driver API
- CUDA Compatibility

Nsight Compute is a tool that collects metrics via hardware counters and software instrumentations for deep-dive profiling and guided performance analysis of the CUDA kernels. GPU architectures supported by Nsight Compute started with the NVIDIA Volta processors.

To understand the profile results from Nsight Compute, you need some knowledge of the GPU hardware.

TIP: Before using Nsight Compute, consider using Nsight Systems to analyze the overall performance on both the CPU and GPU of your application.

Nsight Compute has a command-line interface (CLI), called ncu, for profiling in non-interactive mode; and a graphical user interface (GUI), called ncu-ui, for profiling in interactive mode. This article only provides basic information on using ncu. For detailed information about ncu and ncu-ui, refer to NVIDIA documentation listed in the References section.

Depending on the nature of the application, some CUDA kernels may be launched multiple times during a run (for example, a kernel called within a loop). For each kernel launch, the name of the kernel function (e.g., vectorAdd_A in the example below) and the progress of data collection is shown in the standard output (STDOUT). To collect all requested profile information for the many metrics that are included for profiling, you might need to replay the kernels multiple times. When the collection is completed, it will also show the number of replay passes of that kernel. For example:

==PROF== Profiling "vectorAdd_A" - 0: 0%....50%....100% - 46 passes

WARNING: Collecting performance data using ncu can incur significant runtime overhead on the application, as discussed here.

Nsight Compute is included in the CUDA Toolkit, which in turn is included in the NVIDIA HPC software development kit (SDK) distributions (for example, NVHPC). The latest available version of ncu is recommended, as it likely provides more features for profiling.

Command-Line Options

Some of the ncu options to consider when profiling are listed below. For all options and more details, read output from the ncu -h command after loading a nvhpc module.

- -o [--export] *report_name*

  Saves profile results in a binary report file called *report_name*.ncu-rep. For example, if you use -o my_ncu_output, the profile results are saved in the my_ncu_output.ncu-rep file. After profiling, the binary report file can be imported for analysis via ncu or ncu-ui. Without this option, a profile result file is created temporarily with some results written in text format to STDOUT. The profile result file is then removed.

- -k [--kernel_name] *kernel_name*

  Chooses a specific kernel, with exact match to kernel_name, to profile. For example, -k vectorAdd_A, which is an exact match to the kernel listed after ==PROF== shown above.

- -c [--launch-count] *a_number*

  Limits the number of collected profile results to *a_number*. For example, -c 2.

- -s [--launch-skip] *a_number*

  Skips a given number of kernel launches before data collection starts. The default is 0.

- --set *set_identifier*

  Chooses among the pre-defined profiling sets: basic, detailed, full, nvlink, pmsampling, roofline . The default is basic, which is less comprehensive and incurs less profiling overhead in runtime and memory usage.

  The basic set includes four sections: LaunchStats, Occupancy, SpeedOfLight, WorkloadDistribution . The detailed set and the full set include the same four sections as in the basic set, plus additional sections. Run the ncu --list-sets command to see what sections are included in each pre-defined set.

  Note: The WorkloadDistribution section is not available in nvhpc/24.3 and earlier.

- --section *section_identifier*

  Chooses a specific section to profile. For example, --section ComputeWorkloadAnalysis.

  Run the ncu --list-sections command to see the list of available sections. Each section includes a group of metrics to be profiled, as pre-defined in Nsight_Compute_installation_location/sections/section-identifier.section.

- --metrics *arg*

  Specify all metrics to be profiled, separated by comma. For example, --metrics sm__cycles_active.avg,sm__cycles_elapsed.sum.

  Run the ncu --list-metrics and ncu --query-metrics commands on a GPU node where you intend to run your application to see the available metrics.

Profile Your Application

The examples shown in this article use the Cabeus front ends (CFEs) for the GPUs with x86_64 based CPUs (mil_a100).

For GPUs with aarch64-based CPUs, such as gh200, please be aware of the following:

- You should use one of the following front ends: cghfe01 or cghfe02.
- The module path and available nvhpc modulefiles may be different.

In order to avoid version incompatibility, before profiling, it is recommended to build your application using the compilers, math, and CUDA libraries included in the same nvhpc version you plan to use ncu for profiling.

Within a PBS job, run the module use and module avail commands as shown below to find all available nvhpc versions, then load the latest version. The following example assumes the latest version is 24.7.

```
CFE% qsub -I -q gpu_devel -lselect=1:ngpus=4:model=mil_a100
a_GPU_node> source /usr/local/lib/global.profile   #if using bash
a_GPU_node> module avail nvhpc
a_GPU_node> module load nvhpc/24.7    # with OpenMPI version 4
or
a_GPU_node> module load nvhpc/24.7_openmpi3  # with OpenMPI version 3
```

For Single-Process Applications

Without any additional options, ncu launches and instruments your application, for example, a.out, and collects the basic set of profile results of all CUDA kernels in the application launched during execution.

ncu -o *report_name* [other ncu options] a.out [a.out options]

An output file *report_name*.ncu-rep containing the profile results is saved.

For Single-Node Multi-Process Applications

The following example will save profile results from all ranks into a single report file, *report_name*.ncu_rep.

```
ncu --target-processes all -o report_name [other ncu options] \
mpirun [mpirun options] a.out [a.out options]
```

Note: The mpirun command line above is too long to be formatted as one line, so it is broken with a backslash (\).

For Multi-Node Multi-Process Applications

The following will save profile results with one file per rank, e.g., *report_name*_0.ncu_rep, *report_name*_1.ncu_rep, and so on.

```
For bash,
mpirun [mpirun options] ncu -o report_name_%q{OMPI_COMM_WORLD_RANK} \
[other ncu options] a.out [a.out options]

For csh,
mpirun [mpirun options] ncu -o report_name_%q\{OMPI_COMM_WORLD_RANK\} \
[other ncu options] a.out [a.out options]
```

Note: The mpirun command lines above are too long to be formatted as one line, so they are broken with a backslash (\).

Profiling Just One Rank

NVIDIA suggests using the following script, wrap.sh, for profiling just one rank (e.g., rank 0):

(For bash):

```
#!/bin/bash
if [[ $OMPI_COMM_WORLD_RANK == 0 ]]; then
  ncu -o report-name_${OMPI_COMM_WORLD_RANK}  --target-processes all "$@"
else
  "$@"
fi
```

(For csh):

```
#!/bin/csh
if ( $OMPI_COMM_WORLD_RANK == 0 ) then
ncu -o report-name_${OMPI_COMM_WORLD_RANK} --target-processes all $*
else
$*
endif
```

To profile and then run the script, do:

```
mpirun [mpi options] ./wrap.sh a.out [a.out options]
```

View Profile Report

This step can be performed on either a GPU compute node or a front-end node where Nsight Compute is available.

To get a quick text summary from the binary report, do:

```
ncu --import report_name.ncu-rep
or
ncu -i report_name.ncu-rep
```

If the profiling was done with the `basic` set, this will generate four sections, including GPU Speed Of Light Throughput, Launch Statistics, Occupancy, and GPU and Memory Workload Distribution, for each profiled kernel. The output may also provide additional explanation on the profile results and optimization advice based on rules created by NVIDIA engineers.

You may also add the `--print-details all` option to show more details.

```
ncu --print-details all --import report-name.ncu-rep
```

Use of Nsight Compute `ncu-ui` may provide more comprehensive analysis, including the source line information if the application was compiled with `-gpu=lineinfo` for nvfortran, nvc, and nvc++ and `--generate-line-info` or `-lineinfo` for nvcc. However, the front-ends cfe[01,02] and cghfe[01,02] do not yet have the required Qt software installed to enable the use of `ncu-ui`. If you have Nsight Compute and Qt installed on your local workstation, you may copy your `report-name.ncu-rep` there for analysis.

References

- NVIDIA Nsight Compute
- Nsight Compute Documentation (latest); includes chapters on Kernel Profiling Guide, Nsight Compute, Nsight Compute CLI, and more
- Nsight Compute CLI User Manual (PDF, version 2023.3.1)
- Intro to NVIDIA Nsight Compute (video)
- Speed of Light Analysis with Nsight Compute (video)
- Introduction to Kernel Performance Analysis with NVIDIA Nsight Compute (video)
- Memory Analysis with NVIDIA Nsight Compute (video)