Verifying Code Correctness with PCAST

By: Daniel Howard dhoward@ucar.edu, Consulting Services Group, CISL & NCAR

Date: May 19th, 2022

In this notebook, we return to the MiniWeather application to learn how to use PCAST, a tool specific to the NVIDIA HPC SDK for verifying code correctness. We will cover:

- Benefits and Challenges of Validating Scientific Software
- Usage of Parallel Compiler Assisted Software Testing (PCAST)
    - Comparing CPU and GPU Code Execution
    - PCAST with a Golden File
    - PCAST with OpenACC and Autocompare

Head to the NCAR JupyterHub portal and **start a JupyterHub session on Casper login** (or batch nodes using 1 CPU, no GPUs) and open the notebook in `08_PCAST/08_PCAST.ipynb`. Be sure to clone (if needed) and update/pull the NCAR GPU_workshop directory.

```
# Use the JupyterHub GitHub GUI on the left panel or the below shell commands
git clone git@github.com:NCAR/GPU_workshop.git
git pull
```

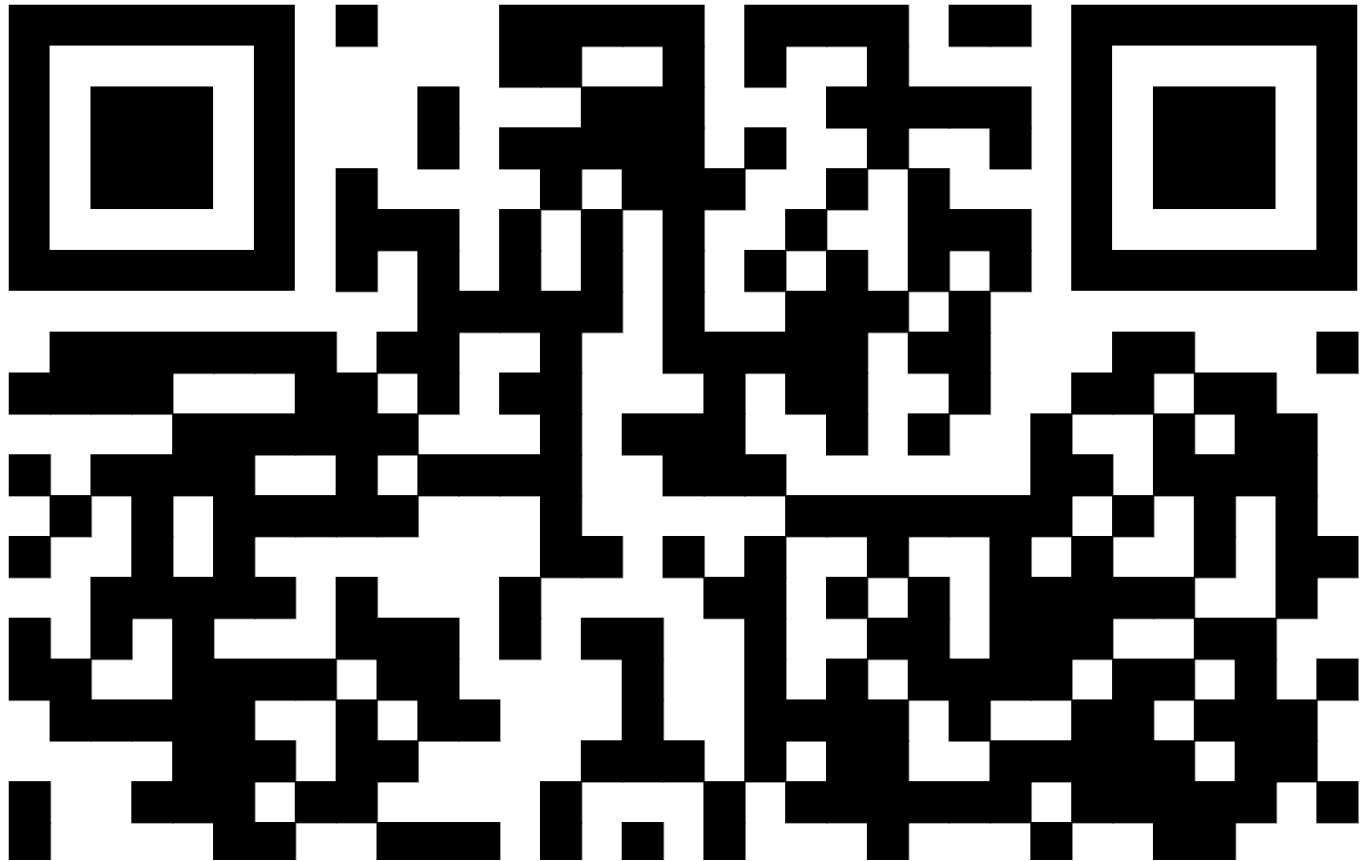
Workshop Etiquette

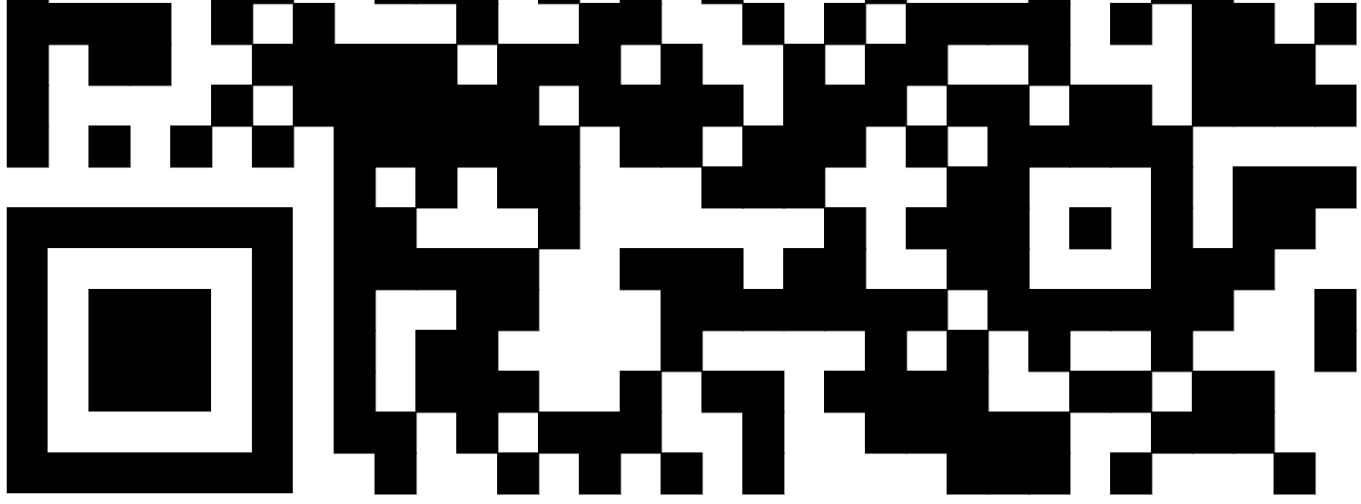
- Please mute yourself and turn off video during the session.
- Questions may be submitted in the chat and will be answered when appropriate. You may also raise your hand, unmute, and ask questions during Q&A at the end of the presentation.
- By participating, you are agreeing to UCAR's Code of Conduct
- Recordings & other material will be archived & shared publicly.
- Feel free to follow up with the GPU workshop team via Slack or submit support requests to support.ucar.edu
    - Office Hours: Asynchronous support via Slack or schedule a time with an organizer

Complete Mid-Workshop Series Survey

In order to get feedback on and improve future GPU workshop series sessions, please complete the below survey. We will spend 3-5 minutes at the start of today's session to collect your feedback.

Head to https://forms.gle/RRkfwnHnDsqqe1zE9 or scan the below QR code.





If you've finished the survey, feel free to ask questions about any past material or other GPU topics during this time.

Notebook Setup

Set the `PROJECT` code to a currently active project, ie `UCIS0004` for the GPU workshop, and `QUEUE` to the appropriate routing queue depending on if during a live workshop session (`gpuworkshop`), during weekday 8am to 5:30pm MT (`gpudev`), or all other times (`casper`). Due to limited shared GPU resources, please use `GPU_TYPE=gp100` during the workshop. Otherwise, set `GPU_TYPE=v100` (required for `gpudev`) for independent work. See Casper queue documentation for more info.

Notebook Setup

Set the `PROJECT` code to a currently active project, ie `UCIS0004` for the GPU workshop, and `QUEUE` to the appropriate routing queue depending on if during a live workshop session (`gpuworkshop`), during weekday 8am to 5:30pm MT (`gpudev`), or all other times (`casper`). Due to limited shared GPU resources, please use `GPU_TYPE=gp100` during the workshop. Otherwise, set `GPU_TYPE=v100` (required for `gpudev`) for independent work. See Casper queue documentation for more info.

In [ ]:

```
export PROJECT=UCIS0004
export QUEUE=casper
export GPU_TYPE=v100

module load nvhpc/22.2 &> /dev/null
export PNETCDF_INC=/glade/u/apps/dav/opt/pnetcdf/1.12.2/openmpi/4.1.1/n
export PNETCDF_LIB=/glade/u/apps/dav/opt/pnetcdf/1.12.2/openmpi/4.1.1/n
```

Benefits of Validating Scientific Software

Building **trust** is paramount for the effective sharing and receiving of computational software. From NASA's Open Source Science for Data Processing and Archives Workshop, **verification** and **validation** is a key component of building trust in scientific software. They contribute towards effective **transparency** and **reproducibility**.

Building **trust in the scientific process** through
transparency, accessibility, inclusivity, and reproducibility

Open (**Transparent**) Science

Both the scientific process and results should be visible, accessible and understandable.

Open (**Accessible**) Science

Data, tools, software, documentation, publications should be accessible to all (FAIR).

Open (**Inclusive**) Science

The process and participants should welcome participation by and collaboration with diverse people and organizations.

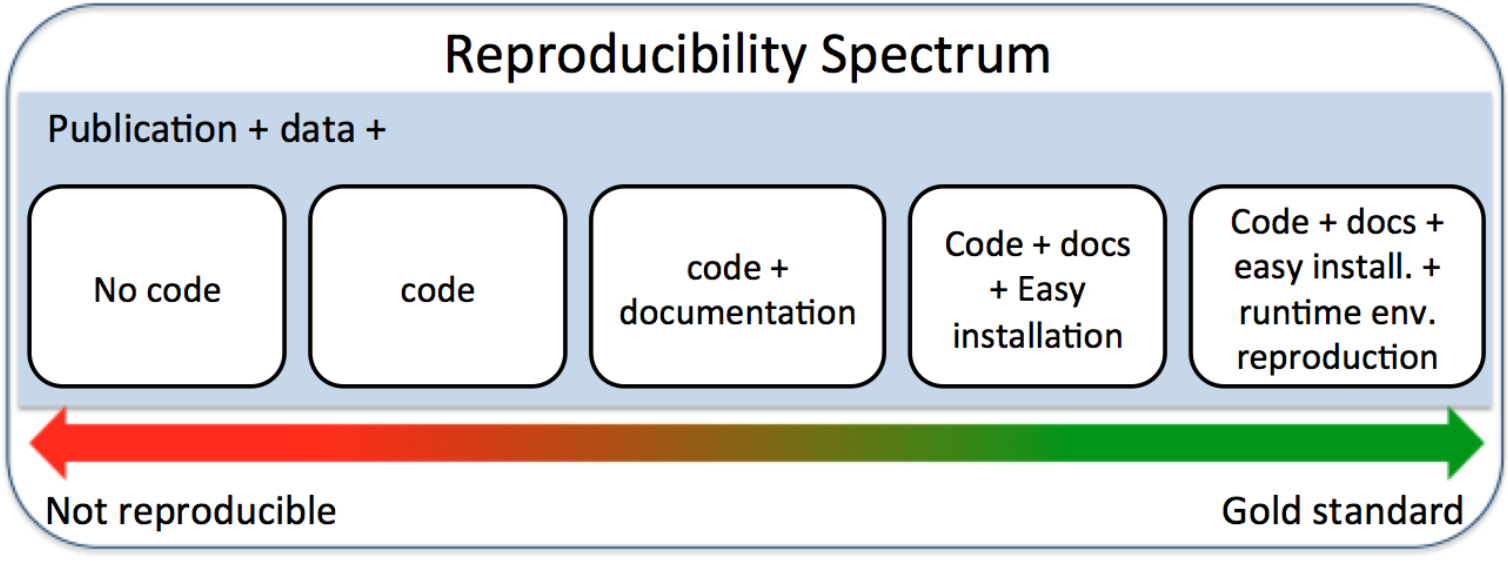
Created by Gregor Cresnar from Noun Project

Open (**Reproducible**) Science

The scientific process and results should be open such that they are reproducible by members of the community.

Open Science and Reproducibility

Including tools, methods, and documentation for validating software can meaningfully build greater **trust in the accuracy** of scientific codes and **enhance reproducibility**.



From Altuna Akalin, Scientific Data Analysis Pipelines and Reproducibility

Promoting and utilizing open science best practices, including making data, code, documentation, and associated tools like validation suites open source, tends to lead to increased recognition and citation rates. See How Open Science Helps Researchers Succeed (McKiernan, et. al., NIH eLife) and Reproducibility in Scientific Software (Heroux, PASC18).

Validation and the Perils of Software Bug Mismanagment

Invariably, **bugs WILL be added to your code**. On average in industry, this can range from 1-25 errors per 1,000 lines of code. Recounting from Testing of HPC Scientific Software (ISC 2018, Anshu Dubey), let's look at the case of modeling protein structures by Geoffrey Chang.

- New code inadvertently transposed columns of data for an electron-density map
- Model code then produced an incorrect protein structure
- Led to the retraction of 5 publications, one with 364 citations
- Other papers and grants that conflicted with this result were rejected
- Chang did find and report the error himself

Being able to catch errors in code through **unit testing** or other means is vitally important to avoid such issues.

Validation and the Perils of Software Bug Mismanagment

Invariably, **bugs WILL be added to your code**. On average in industry, this can range from 1-25 errors per 1,000 lines of code. Recounting from Testing of HPC Scientific Software (ISC 2018, Anshu Dubey), let's look at the case of modeling protein structures by Geoffrey Chang.

- New code inadvertently transposed columns of data for an electron-density map
- Model code then produced an incorrect protein structure
- Led to the retraction of 5 publications, one with 364 citations
- Other papers and grants that conflicted with this result were rejected
- Chang did find and report the error himself

Being able to catch errors in code through **unit testing** or other means is vitally important to avoid such issues.

PCAST does not specifically do "unit testing" but learn more about the broader landscape of software reliability under the scope of research software via the Better Scientific Software (BSSw) organization:

- BSSW's Better Reliability blog post category
- BSSW's Software Verification blog post

Challenges of Validating Scientific Software

Achieving appropriate validation workflows is often not easy, particularly in the case of constantly changing scientific software. The paper Challenges for Verifying and Validating Scientific Software in Computational Materials Science by Vogel, et. al. at Humboldt University and DLR highlights common issues while implementing testing frameworks.

1. Lack of Precise Oracles
    - Knowing the precise output of a science/engineering model is typically impossible a priori
    - IEEE floating point computations are inexact and change depending on operation ordering and type
    - See What every computer scientist should know about floating-point arithmetic by David Goldberg

1. Lack of Precise Oracles
    - Knowing the precise output of a science/engineering model is typically impossible a priori
    - IEEE floating point computations are inexact and change depending on operation ordering and type
    - See What every computer scientist should know about floating-point arithmetic by David Goldberg

1. Large Configuration Space
    - Experimental nature of scientific software promotes the selection of many different algorithms and approaches to problem solving

1. Lack of Precise Oracles
    - Knowing the precise output of a science/engineering model is typically impossible a priori
    - IEEE floating point computations are inexact and change depending on operation ordering and type
    - See What every computer scientist should know about floating-point arithmetic by David Goldberg

1. Large Configuration Space
    - Experimental nature of scientific software promotes the selection of many different algorithms and approaches to problem solving

1. Large-Scale, Heterogeneous Data
    - Selecting test and validation data at pre- and post-processing steps results in high data variablilty
    - Test data is then inherently cumbersome and expensive to manage

1. Lack of Precise Oracles
    - Knowing the precise output of a science/engineering model is typically impossible a priori
    - IEEE floating point computations are inexact and change depending on operation ordering and type
    - See What every computer scientist should know about floating-point arithmetic by David Goldberg

1. Large Configuration Space
    - Experimental nature of scientific software promotes the selection of many different algorithms and approaches to problem solving

1. Large-Scale, Heterogeneous Data
    - Selecting test and validation data at pre- and post-processing steps results in high data variablilty
    - Test data is then inherently cumbersome and expensive to manage

1. Global Software Development
    - Modern large scale software projects, like in climate science, is difficult to manage across global teams
    - Standardizing testing frameworks across disparate teams is a political process

PCAST: Parallel Compiler Assisted Software Testing

The PCAST tool serves as a **convenient compiler aided framework** for quickly building in some level of **software testing** into your development workflow, particularly to **compare CPU data to GPU data**.

However, PCAST won't cover all the benefits of software validation or resolve all the challenges previously discussed.

PCAST: Parallel Compiler Assisted Software Testing

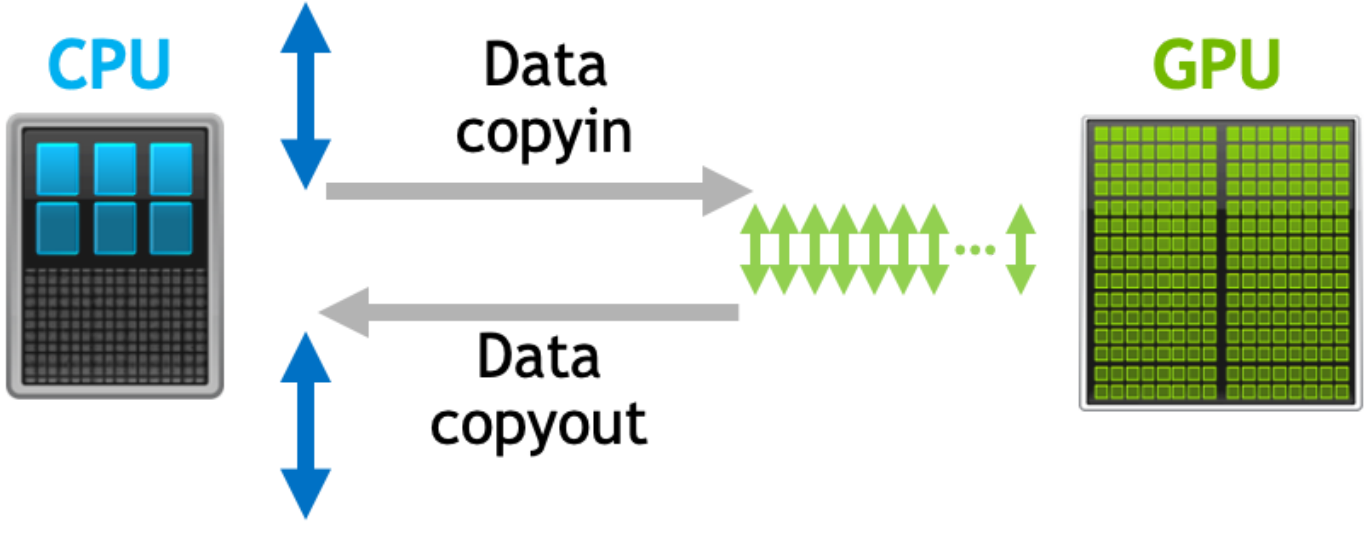
The PCAST tool serves as a **convenient compiler aided framework** for quickly building in some level of **software testing** into your development workflow, particularly to **compare CPU data to GPU data**.

However, PCAST won't cover all the benefits of software validation or resolve all the challenges previously discussed.

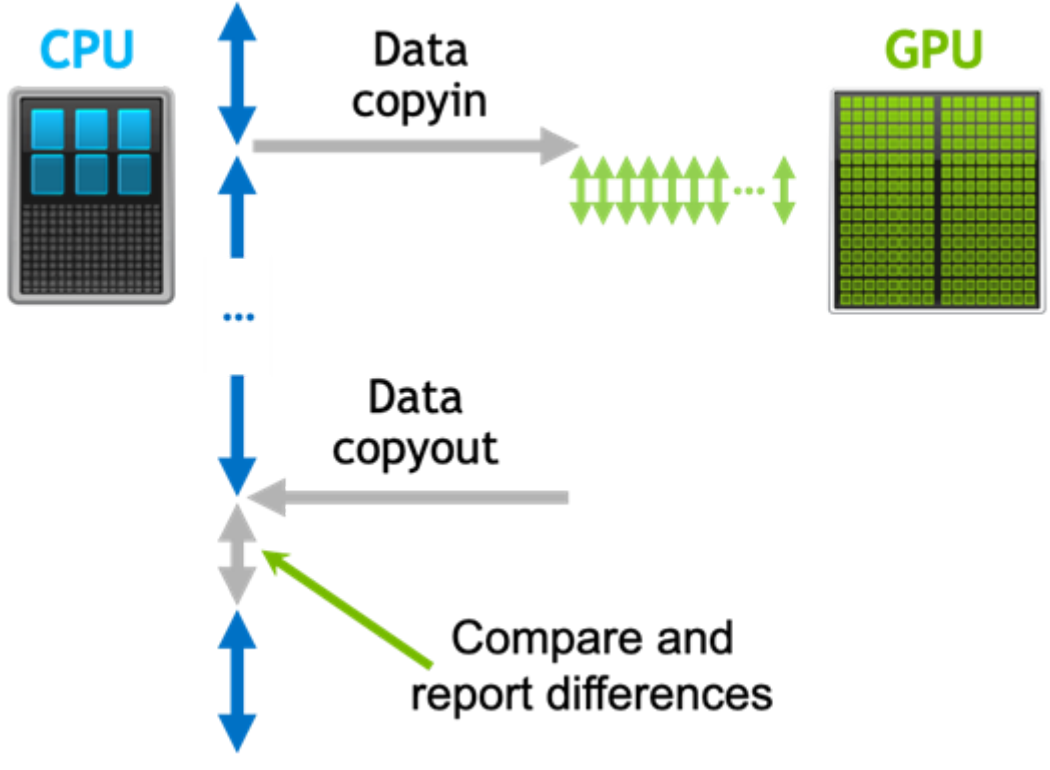
Primarily, PCAST allows for verifying that either

1. Output from previously run code is similar or equal to the output of modified code
2. Calculations performed on the CPU is similar or equal to calulations performed on the GPU

GPU Program Execution - Normal



GPU Program Execution - PCAST



Usage of PCAST

The main documentation for using PCAST can be found within NVIDIA's HPC SDK Documentation, HPC Compiler's User Guide. Alternatively, the NVIDIA blog post Detecting Divergence Using PCAST to Compare CPU to GPU Results may be referenced as a more casual read.

Usage with a Golden File: Run PCAST using calls to `pcast_compare` or `!$nvf compare()` directives (latter requires compiler flag `-Mpcast`) for CPU resident comparisons to a **golden file**.

Usage with OpenACC: Run PCAST using calls to `acc_compare` or `!$acc compare()` directives alongside usage of compiler flag `-gpu=redundant` or `-gpu=autocompare`.

Additional options can be set using the `PCAST_COMPARE={option-list}` environment variable.

Usage of PCAST - Golden File

In this first approach, PCAST can be run to compare successive program runs against a ground truth golden file. Essentially, CPU results will be compared to CPU results, with the assumption that the results in the golden file are correct.

NOTE: It's up to the programmer to determine if the results are in fact correct according to the model.

With calls to `pcast_compare(...)` or `!$nvf compare(var-list)` directives, a data file named `pcast_compare.dat` by default will either be created if it does not exist or read to compare computed data with saved data from the data file. If using directives, they will only be enabled if the program is compiled with the `-Mpcast` flag.

Notably, the directive is much more portable and significantly easier to use, ie `!$nvf compare(a(1:N))`, only requiring the input to be a `var-list` with an inferred full size of the array or specified sub-slices.

`pcast_compare()` Prototype

Example arguments for the `pcast_compare(...)` function is given below, where the last 4 arguments are flexible to the descriptiveness required:

```
pcast_compare(state,"real(8)",(2*hs+nx)*
(2*hs+nz)*NUM_VARS,"state","miniweather_orig.F90","main",155)
```

1. The address of the data to be saved or compared.
2. A string containing the data type, ie `real(2,4,8)` `integer(2,4,8)` `complex(4,8)`
3. The number of elements to compare.
4. A string treated as the variable name.
5. A string treated as the source file name.
6. A string treated as the function name.
7. An integer treated as a line number.

Additional PCAST Options

Additonal options can be supplied to the PCAST runtime by modifying the `PCAST_COMPARE={option-list}` environment variable.

- `datafile="name.dat"` : Change the name of the golden file
- `create` : Explicitly force the creation of the golden file
- `compare` : Explicitly force the comparison of the golden file
- `disable` : Disable any PCAST actions from taking place and force PCAST functions to immediately return with no effect. This does not disable `-gpu=redundant` execution if enabled for OpenACC codes

All `PCAST_COMPARE` environmental variable options can be found in the NVIDIA Documentation.

EXERCISE: Usage of PCAST - Create a Golden File

We will first use the original OpenACC version of MiniWeather in order to create a **golden file** from the `openacc_orig` executable. To do this, edit the file miniWeather_mpi_openacc_orig.F90 and add either `call pcast_compare()` or directives `!$nvf compare()` where desired. If you use the function call, make sure to add `use openacc` to head of program (already done) and follow the function prototype given previously.

A simple recommendation is to save results of the `state` variable into the golden file near the end of program execution. Of course, additional variables could be saved but keep in mind, if `pcast_compare()` or its associated directive is called too many times across program execution, the golden file can grow to very large sizes.

Then, run the next two cells to compile and run the code on the GPU. For future exercises, be sure to keep the configuration of the model consistent.

EXERCISE: Usage of PCAST - Create a Golden File

We will first use the original OpenACC version of MiniWeather in order to create a **golden file** from the `openacc_orig` executable. To do this, edit the file miniWeather_mpi_openacc_orig.F90 and add either `call pcast_compare()` or directives `!$nvf compare()` where desired. If you use the function call, make sure to add `use openacc` to head of program (already done) and follow the function prototype given previously.

A simple recommendation is to save results of the `state` variable into the golden file near the end of program execution. Of course, additional variables could be saved but keep in mind, if `pcast_compare()` or its associated directive is called too many times across program execution, the golden file can grow to very large sizes.

Then, run the next two cells to compile and run the code on the GPU. For future exercises, be sure to keep the configuration of the model consistent.

In [ ]:
```
export OPENACC_FLAGS="-acc -gpu=cc60,cc70 -Mpcast"

mpif90 -I${PNETCDF_INC} -Mextend -O0 -DNO_INFORM -c miniWeather_mpi_ope
-D_NX=200 -D_NZ=100 -D_SIM_TIME=2.01 -D_OUT_FREQ=1 -D_DATA_SPEC=DATA_SP

mpif90 -Mextend -O3 -DNO_INFORM miniWeather_mpi_openacc_orig.F90.o -o c
rm -f miniWeather_mpi_openacc_orig.F90.o
```

EXERCISE: Usage of PCAST - Create a Golden File

We will first use the original OpenACC version of MiniWeather in order to create a **golden file** from the `openacc_orig` executable. To do this, edit the file miniWeather_mpi_openacc_orig.F90 and add either `call pcast_compare()` or directives `!$nvf compare()` where desired. If you use the function call, make sure to add `use openacc` to head of program (already done) and follow the function prototype given previously.

A simple recommendation is to save results of the `state` variable into the golden file near the end of program execution. Of course, additional variables could be saved but keep in mind, if `pcast_compare()` or its associated directive is called too many times across program execution, the golden file can grow to very large sizes.

Then, run the next two cells to compile and run the code on the GPU. For future exercises, be sure to keep the configuration of the model consistent.

In [ ]:
```
export OPENACC_FLAGS="-acc -gpu=cc60,cc70 -Mpcast"

mpif90 -I${PNETCDF_INC} -Mextend -O0 -DNO_INFORM -c miniWeather_mpi_ope
-D_NX=200 -D_NZ=100 -D_SIM_TIME=2.01 -D_OUT_FREQ=1 -D_DATA_SPEC=DATA_SP

mpif90 -Mextend -O3 -DNO_INFORM miniWeather_mpi_openacc_orig.F90.o -o c
rm -f miniWeather_mpi_openacc_orig.F90.o
```

In [ ]:

```
# export PCAST_COMPARE="create"
qcmd -A $PROJECT -q $QUEUE -l select=1:ncpus=1:ngpus=1 -l gpu_type=$GPU
cd $PWD && ./openacc_orig
```

EXERCISE: Usage of PCAST - Test against a Golden File

Add the same `pcast_comapre()` calls or directives to the non-original file miniWeather_mpi_openacc.F90. A bug has been introduced somewhere in the code.

Make and run the non-original file and **use the PCAST report to try to determine where the bug is**.

1. **Would testing additional variables across the MiniWeather code fascilitate this search better?**
2. **What would be an easier way to find the source of the bug?**
3. **How would you incorporate PCAST testing in a development workflow to minimize creation of bugs as changes are made to source files?**

EXERCISE: Usage of PCAST - Test against a Golden File

Add the same `pcast_comapre()` calls or directives to the non-original file miniWeather_mpi_openacc.F90. A bug has been introduced somewhere in the code.

Make and run the non-original file and **use the PCAST report to try to determine where the bug is**.

1. **Would testing additional variables across the MiniWeather code fascilitate this search better?**
2. **What would be an easier way to find the source of the bug?**
3. **How would you incorporate PCAST testing in a development workflow to minimize creation of bugs as changes are made to source files?**

In [ ]:
```
export OPENACC_FLAGS="-acc -gpu=cc60,cc70 -Mpcast"

mpif90 -I${PNETCDF_INC} -Mextend -O0 -DNO_INFORM -c miniWeather_mpi_ope
-D_NX=200 -D_NZ=100 -D_SIM_TIME=2.01 -D_OUT_FREQ=1 -D_DATA_SPEC=DATA_SP

mpif90 -Mextend -O3 -DNO_INFORM miniWeather_mpi_openacc.F90.o -o openac
rm -f miniWeather_mpi_openacc.F90.o
```

EXERCISE: Usage of PCAST - Test against a Golden File

Add the same `pcast_comapre()` calls or directives to the non-original file miniWeather_mpi_openacc.F90. A bug has been introduced somewhere in the code.

Make and run the non-original file and **use the PCAST report to try to determine where the bug is**.

1. **Would testing additional variables across the MiniWeather code fascilitate this search better?**
2. **What would be an easier way to find the source of the bug?**
3. **How would you incorporate PCAST testing in a development workflow to minimize creation of bugs as changes are made to source files?**

In [ ]:
```
export OPENACC_FLAGS="-acc -gpu=cc60,cc70 -Mpcast"

mpif90 -I${PNETCDF_INC} -Mextend -O0 -DNO_INFORM -c miniWeather_mpi_ope
-D_NX=200 -D_NZ=100 -D_SIM_TIME=2.01 -D_OUT_FREQ=1 -D_DATA_SPEC=DATA_SP

mpif90 -Mextend -O3 -DNO_INFORM miniWeather_mpi_openacc.F90.o -o openac
rm -f miniWeather_mpi_openacc.F90.o
```

In [ ]:
```
export PCAST_COMPARE="compare,summary"
```

```
qcmd -A $PROJECT -q $QUEUE -l select=1:ncpus=1:ngpus=1 -l gpu_type=$GPU
cd $PWD && ./openacc
```

Usage of PCAST - OpenACC and `-gpu=redundant` / `-gpu=autocompare`

In this second aproach, PCAST can be run to directly compare the calculations between the CPU and GPU.

In `redundant` mode, CPU code is generated alongside GPU code and ran redundantly. Then, every time a `acc_compare` call or `!$acc compare()` directive is encountered, the compiler will verify the values between any specified variables.

In `autocompare` mode, CPU code is again generated alongside GPU code and ran reduntantly. Then, every time data is moved between CPU and GPU, such as through an `!$acc update host()` directive or at the edges of data regions, the compiler will verify the values between the data that was to be moved.

`autocompare` is the easiest method to quickly test correctness between CPU and GPU code particularly when using OpenACC and automaticaly implies `redundant`. Since the source file stays the same, ie directive comments are ignored when generating CPU code, this can highlight if the OpenACC runtime is introducing any divergence in the GPU target code either due to inappropriate loop directives, ie missing `private()` or `reduction()`, or bad managagement of data movement.

Additional PCAST Options

Additonal options can be supplied to the PCAST runtime by modifying the `PCAST_COMPARE={option-list}` environment variable. These options are also relevant for golden file mode.

- `outputfile="name.dat"` : Specify the file to write comparison output. Default is `stderr`
- `summary` : Print summary of comparisons at the end of execution
- `abs=n` , `rel=n` , `ulp=n` , or `ieee` : Specify the types and tolerance of comparisons performed, where `n` is the magnitude of relative 10^n, absolute 10^n, or number of units precision difference tolerated respectively. Add `ieee` to enable NaN checks
- `report=n` : Modify the default number (50) differences reported at each comparison where `n` is the number of differences to report
- `stop` : Stop at the first difference outside of tolerance

All `PCAST_COMPARE` environmental variable options can be found in the NVIDIA Documentation.

EXERCISE: Usage of PCAST - OpenACC and `-gpu=redundant` / `-gpu=autocompare`

Try out PCAST with `autocompare`. Initially, you will notice that many "errors" are reported. However, all of them are within machine precision error. Add the `PCAST_COMPARE` flag `abs=12` in order to tolerate errors within a reasonable bounds. If you like, add additional `!$acc update host()` directives in different parts of MiniWeather to automatically compare in additional regions.

1. **How does this impact your confidence in the correctness of the GPU code?**
2. **Try adding the `tile(32,32,NUM_VARS)` clause like in a previous OpenACC exercise that produced incorrect results. Can you verify the incorrect results more clearly with PCAST?**

EXERCISE: Usage of PCAST - OpenACC and `-gpu=redundant` / `-gpu=autocompare`

Try out PCAST with `autocompare`. Initially, you will notice that many "errors" are reported. However, all of them are within machine precision error. Add the `PCAST_COMPARE` flag `abs=12` in order to tolerate errors within a reasonable bounds. If you like, add additional `!$acc update host()` directives in different parts of MiniWeather to automatically compare in additional regions.

1. **How does this impact your confidence in the correctness of the GPU code?**
2. **Try adding the `tile(32,32,NUM_VARS)` clause like in a previous OpenACC exercise that produced incorrect results. Can you verify the incorrect results more clearly with PCAST?**

In [ ]:
```
export OPENACC_FLAGS="-acc -gpu=autocompare,cc60,cc70"

mpif90 -I${PNETCDF_INC} -Mextend -O0 -DNO_INFORM -c miniWeather_mpi_ope
-D_NX=200 -D_NZ=100 -D_SIM_TIME=2.01 -D_OUT_FREQ=1 -D_DATA_SPEC=DATA_SP

mpif90 -Mextend -O3 -DNO_INFORM miniWeather_mpi_openacc.F90.o -o openac
rm -f miniWeather_mpi_openacc.F90.o
```

EXERCISE: Usage of PCAST - OpenACC and `-gpu=redundant` / `-gpu=autocompare`

Try out PCAST with `autocompare`. Initially, you will notice that many "errors" are reported. However, all of them are within machine precision error. Add the `PCAST_COMPARE` flag `abs=12` in order to tolerate errors within a reasonable bounds. If you like, add additional `!$acc update host()` directives in different parts of MiniWeather to automatically compare in additional regions.

1. **How does this impact your confidence in the correctness of the GPU code?**
2. **Try adding the `tile(32,32,NUM_VARS)` clause like in a previous OpenACC exercise that produced incorrect results. Can you verify the incorrect results more clearly with PCAST?**

In [ ]:
```
export OPENACC_FLAGS="-acc -gpu=autocompare,cc60,cc70"

mpif90 -I${PNETCDF_INC} -Mextend -O0 -DNO_INFORM -c miniWeather_mpi_ope
-D_NX=200 -D_NZ=100 -D_SIM_TIME=2.01 -D_OUT_FREQ=1 -D_DATA_SPEC=DATA_SP

mpif90 -Mextend -O3 -DNO_INFORM miniWeather_mpi_openacc.F90.o -o openac
rm -f miniWeather_mpi_openacc.F90.o
```

In [ ]:
```
export PCAST_COMPARE="summary,report=2"
```

```
qcmd -A $PROJECT -q $QUEUE -l select=1:ncpus=1:ngpus=1 -l gpu_type=$GPU
$PWD/openacc
```

Final Points

1. **PCAST is not able to verify that the original CPU code is correct** accoring to the science or model specification.
2. However, you can **use PCAST to verify that CPU+GPU results stay consistent** across minor refactoring edits using a golden file.
3. **PCAST can verify that the results computed between CPU and GPU are in agreement and correct**, up to machine precision error.
    - Differences between how GPU and CPU code is compiled can still introduce machine precision error.
4. Implementing some form of **validation and verification in a development workflow establishes greater trust in the software and minimizes bugs**.

Suggested Resources

- HPC Compiler's User Guide
- Detecting Divergence Using PCAST to Compare CPU to GPU Results

In [ ]: