Directive Based Programming with OpenACC

By: Daniel Howard, Consulting Services Group, CISL & NCAR, March 31st 2022

In this notebook we present techniques and code examples for using OpenACC to develop for GPUs. We will cover:

1. Comparison of descriptive & perscriptive programming and their portability
    - OpenACC
    - OpenMP
    - ISO Standard Language Parallelism
2. Fork-Join Execution Model for Attached GPU Accelerators
3. OpenACC API Directives with Unified Memory
    - Compute construct directives
        - `!$acc kernels ...`
        - `!$acc parallel ...`
        - `!$acc serial ...`
    - `!$acc loop` and other specification clauses

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

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=gpudev
export GPU_TYPE=v100
```

Useful Definitions

- **Device**: Accelerator on which execution can be offloaded (ex : GPU).
- **Host**: Machine (ie CPU) hosting 1 or more accelerators and in charge of execution control.
- **Kernel**: Computational runtime derived from a section of parallelized code that is scheduled to run on an accelerator.
- **Execution thread**: Sequence of kernels to be executed on an accelerator.
- **Thread**: A single Processing Element (PE) or execution unit. On a NVIDIA GPU, run on a single CUDA core.

- **Streaming Multiprocessor (SM)**: Highest level processing unit in NVIDIA GPU that processes blocks/gangs of threads. Each SM provides a shared memory cache, similar to L1, accessible by the block/gang of threads running on that SM (V100 - 96kB, A100 - 160kB).
- **Grid**: *Collection of blocks/gangs* of threads that are distributed for execution across SMs. Can be organized in Euclidean dimensions.
- **Gang (OpenACC) / Teams (OpenMP)**: Coarse-grain parallelism structure, assigned to a SM. Contains a *block* of threads at size num_workers times vector_length and has a shared memory/L1 cache.
- **Worker (OpenACC)**: Fine-grain parallelism that executes vectors of threads. Equivalent to a *warp*.
- **Vector**: Group of threads executing the same *SIMT* instruction and executed by a worker. Note: Different vendors (ie NVIDIA vs AMD) use different terms that mean equivalent concepts

Portability and Comparing OpenACC, OpenMP, and ISO Standard Language Parallelism

Recall from earlier sessions the difference between **prescriptive** and **descriptive** programming. In general, **descriptive** paradigms, given the flexibility afforded to compilers, are able to achieve greater portability across different hardware types.

- OpenMP has longer history from 1997 and predominantly is **prescriptive** while OpenACC is more **descriptive**, beginning in 2011
- ISO Standard Language Parallelism (stdPar) tends **descriptive**, but still early stage implementation across compilers
- More compilers support OpenMP but fewer support OpenACC (links list current compiler support)
- OpenACC is more mature for GPUs (esp. NVIDIA) while OpenMP only recently has been expanding GPU offload support
    - See Oak Ridge National Lab's "Introduction to OpenMP GPU Offload" from Dec 2021 if interested.
    - Note: **Legacy OpenMP will NOT run well off the shelf on GPUs**
- stdPar is in language standard and aims to replace need for directives, see Burying the OpenACC vs OpenMP Hatchet by Michael Wolfe

Nonetheless, directive based & stdPar landscape constantly changes and different developers have their own opinion which is best. When deciding yourself, **most important is to consider any long term portability needs of your code**.

Each cycle, a HPC system's hardware typically is set on order 3-5 years while a software project can more easily extend to 10+ years if designed with longevity in mind. See Better Scientific Software supported by the Department of Energy and National Labs alongside the Exascale Computing Project.

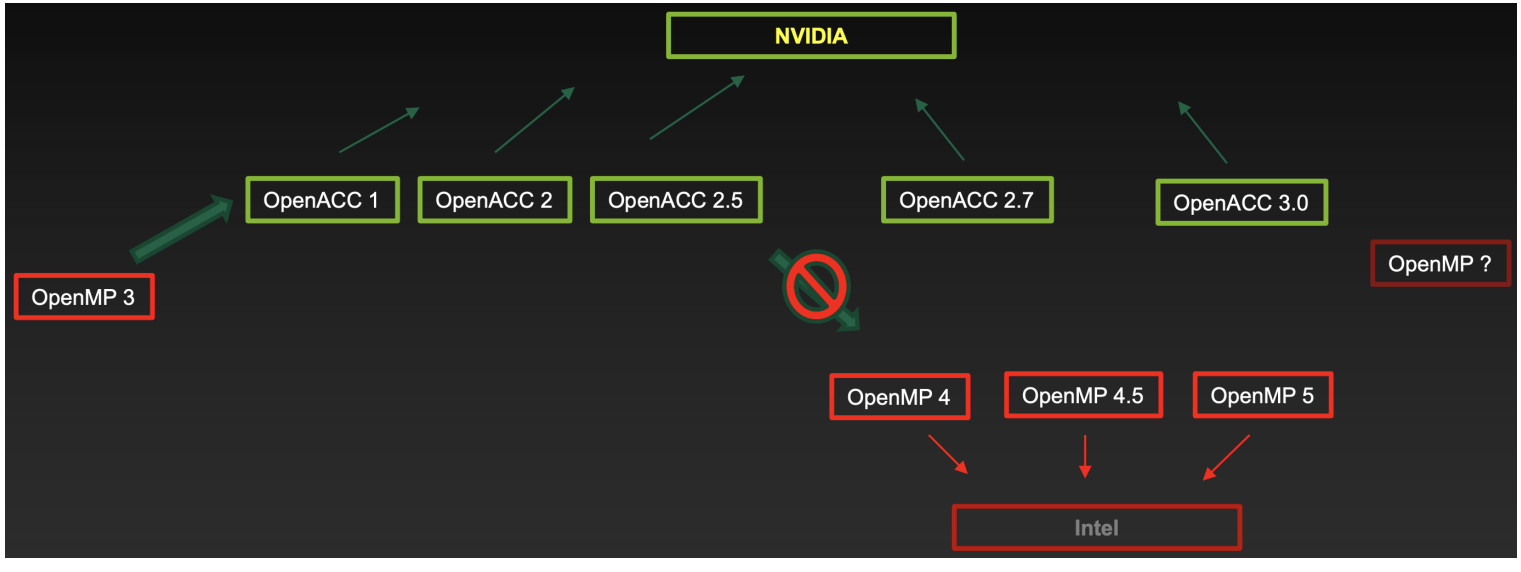
Philosophical Differences between OpenACC and OpenMP Programming Models

- **OpenACC**
    - **Compilers are allowed flexibility** in how to parallelize
    - **Programmer augments information available to compiler** and can optionally provide suggestions on how to map threads on accelerator
    - **More portable across target devices** since compiler expects freedom in how to parallelize for each target type
    - Non-parallel code must be made parallel. **Programmer can safely suggest parallel regions** since compiler checks if loops are actually parallel (unless `independent` clause is used)

- **OpenMP**
    - **Compilers must follow user-directed parallelization** and the programmer must explicitly specify how the parallelism is achieved
        - Only recently allowed for compiler-generated automatic parallelization using `loop` clause
    - **Less portable**, different target devices (ie GPUs vs CPUs) require different directives
    - Non-parallel code can be optionally restructured. **Responsibility of programmer to ensure correct implementation** of parallel regions

- **ISO Standard Language Parallelism**
    - Still **allows flexibility to the compiler** depending on available target device
    - **Removes the need for directives** or other extra instructions to compilers
    - **Standard within language** and not in a separate organization
    - Not yet as robust as OpenACC/OpenMP, ie missing support for reductions. Over time, should gain breadth of scope of directives like OpenACC/OpenMP

OpenACC and OpenMP Are Relatives



From "OpenMP and GPUs" Urbanic, 2021

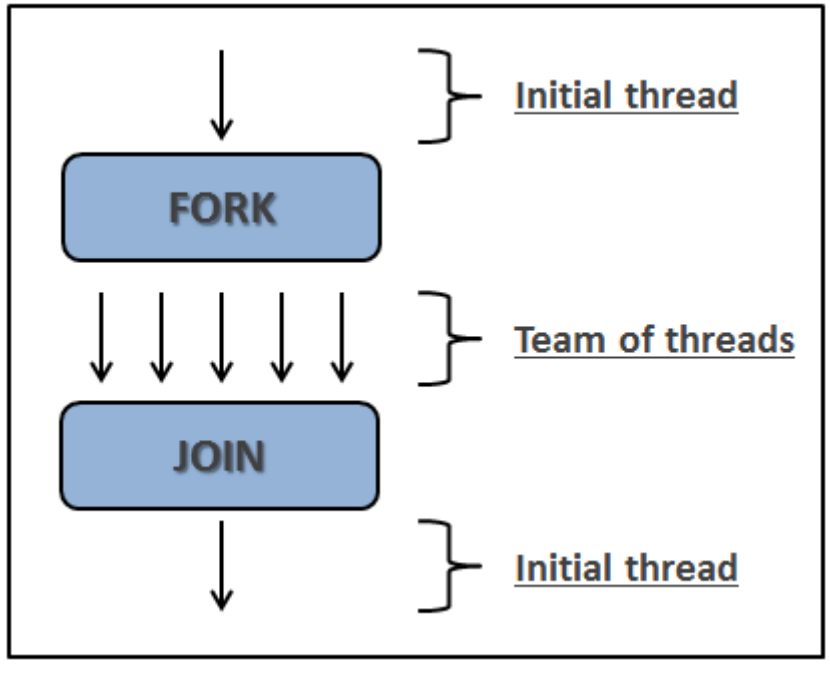
- **OpenMP** - Began in 1997 (GPU offload in aproximately 2013-2015)
    - Latest 5.2 standard Nov 2021, broad community adoption, Intel strongly influences development
- **OpenACC** - Began 2010
    - Latest 3.1 standard Nov 2020, GPU-only community adoption, NVIDIA strongly influences development
- Compiler support for GPU target variable across vendors
- Uncertain if standards will merge and/or be replaced by ISO Language Standards
- However, source translation between all these approaches is relatively straightforward

Translating between OpenACC and OpenMP

OpenACC	OpenMP	Description
Regional Directives		Initializes parallel runtime regions
`!$acc parallel ...`	`!$omp target teams ...`	Establishes a parallel runtime region/compute kernel
`!$acc kernels ...`	`!$omp target teams loop`	Similar but gives optimization flexibility to compiler
`!$acc loop ...`	`!omp ...`	Defines a parallel loop within a compute kernel
Parallelization Clauses		Specifies types of parallelization in a region
`gang`	`distribute` or `distribute parallel for`	Specifies a gang work unit
`worker`	`parallel for`	Specifies a worker work unit within a gang
`vector`	`simd` or `parallel for num_threads(1) simd`	Specifies a SIMD work unit, best with coalesced memory
`num_gangs()`	`num_teams()`	Specifies number of gangs/teams
`num_workers()`	`num_threads()`	Specifies number of workers (threads in CPU context)
`vector_length()`	`simdlen()`	Specifies size of SIMD type operation
Data Clauses		Specifies data movement between CPU & GPU in parallel/data regions
`create()`	`alloc()`	Allocates memory on target device for data object
`copy()`	`map(tofrom:)`	Allocates memory if needed, copies data at region entry/exit
`copyin()`	`map(to:)`	Allocates memory if needed, copies data at region entry
`copyout()`	`map(from:)`	Allocates memory if needed and copies data object at region exit
`present()`	`assert(omp_target_is_present())`	Asserts that a data object is already present on GPU

Details for all of these directives and control statements for directive based computing will be covered later. Point is that **most statements have clear translations** and it wouldn't be a significant loss of effort to choose one programming paradigm but then later refactor to another. See CCAMP paper (Lambert, et al) for language translation details.

Fork-Join Execution Model



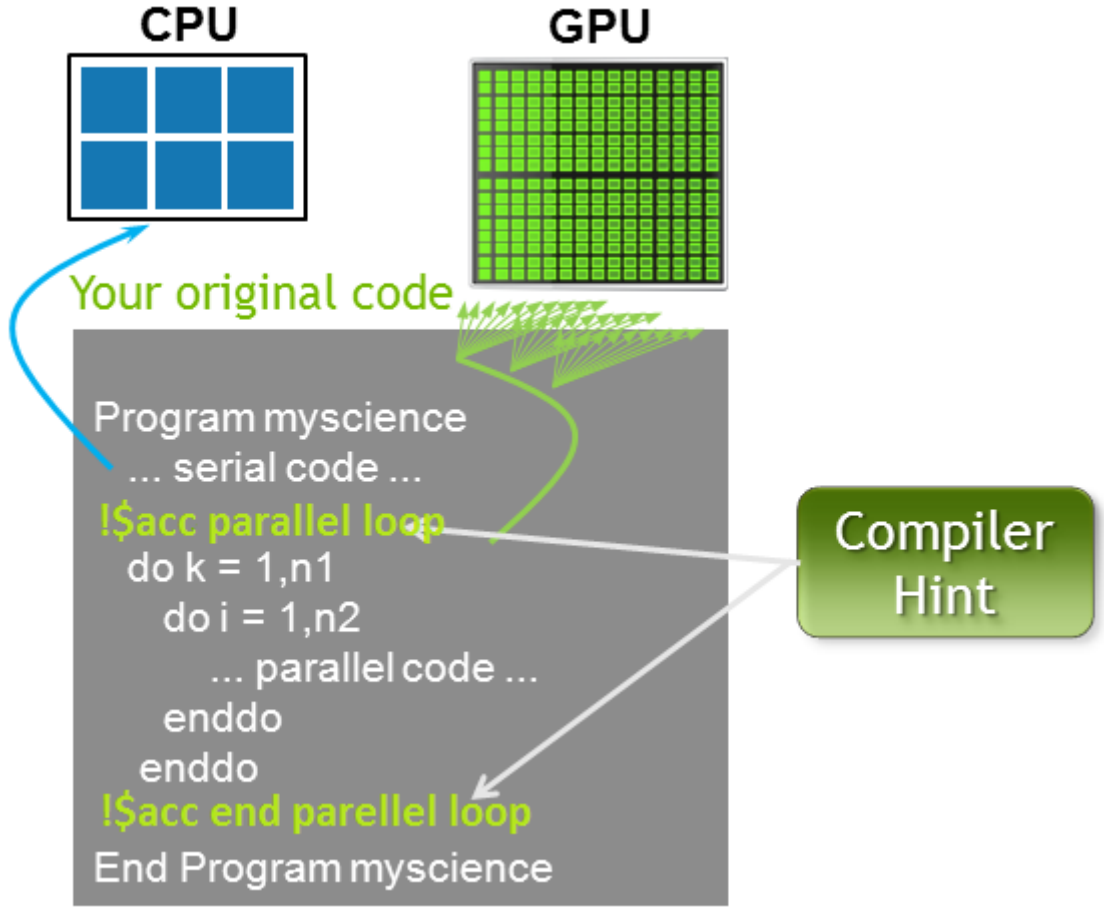
Rita, et al, 2018

Both OpenMP and OpenACC employ similar execution models

1. First, the host CPU runs serial code until it encounters a parallel region
2. The host process then **forks off many threads** to process the parallel code
    - OpenMP allows some degree of thread divergence and nested parallelism while OpenACC (and GPU programming) is less flexible and expects all threads to perform the same tasks in a kernel
3. Once all threads complete their work, the threads **join back together** and continue

The difference for GPU offload is there can be additional steps at both the fork and join to transfer data from the host to the device or vice versa. OpenMP often refers to a "master" thread that leads execution on a host device but the concept of a "master" GPU thread is not practical.

Directive Based Programming Example

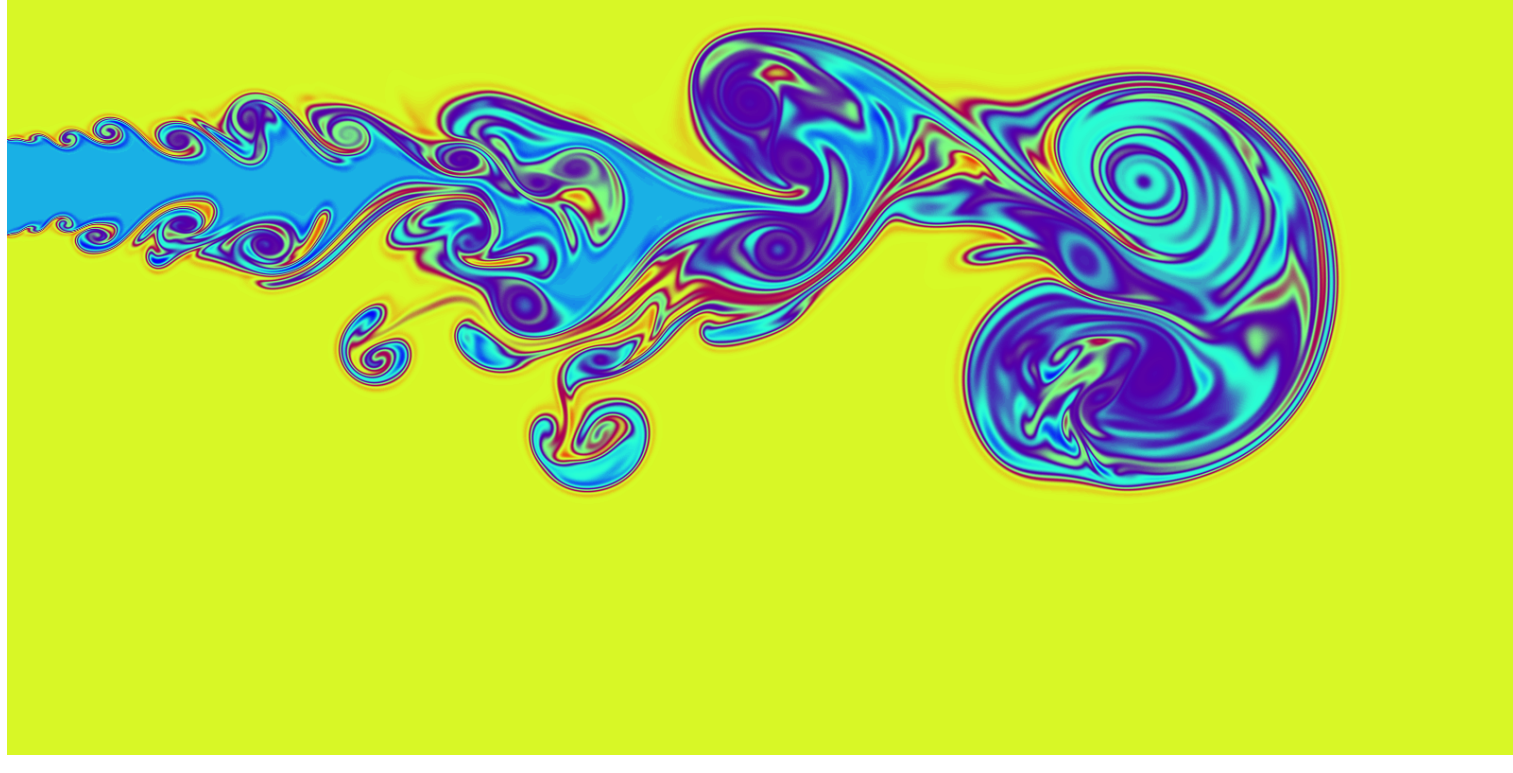


Essentially, both **OpenACC and OpenMP take existing codes and decorate them with directives that define parallel regions** alongside other details given by the programmer in associated clauses. Compilers can then choose to honor these directives and build an executable that forks and joins parallel threads across the program's execution.

In this session, we will focus on OpenACC given it's maturity and limited time to only focus on one programming model in a single session.

MiniWeather for Simulating Weather-like Flows

For this session, we will use the MiniWeather mini-app to explore how you can implement OpenACC. This mini-app simulates weather-like flows, specifically to fascilitate training in parallelizing accelerated HPC architectures and has been developed by Matt Norman (ORNL), Jeff Larkin (Nvidia), and Isaac Lyngaas (ORNL). For example, MiniWeather can model injection jet streams into a stable atmosphere, like below.



Test that MiniWeather Builds Correctly

First, let's build MiniWeather using cmake_casper_nvhpc.sh and run some tests to make sure the model builds correctly. To note, we will focus on the `FORTRAN` and `OpenACC` versions of the model but MiniWeather is also a great tool for exploring other programming models like `C` and `C++`, each utilizing `MPI`, `OpenMP`, or `do concurrent` / `std::par`. See other language folders and associated programming paradigm source files for examples. A script file for Casper has been adapted for the other build folders to fascilitate this exploration if you would like to do this on your own time. See MiniWeather's README "Compiling and Running the Code" section for more info about this.

Test that MiniWeather Builds Correctly

First, let's build MiniWeather using cmake_casper_nvhpc.sh and run some tests to make sure the model builds correctly. To note, we will focus on the `FORTRAN` and `OpenACC` versions of the model but MiniWeather is also a great tool for exploring other programming models like `C` and `C++`, each utilizing `MPI`, `OpenMP`, or `do concurrent` / `std::par`. See other language folders and associated programming paradigm source files for examples. A script file for Casper has been adapted for the other build folders to fascilitate this exploration if you would like to do this on your own time. See MiniWeather's README "Compiling and Running the Code" section for more info about this.

Initially, we will run this test with the base `mpi` model using miniWeather_mpi.F90 and the already implemented `openacc` model using miniWeather_mpi_openacc.F90. The file in fortran/build/cmake_casper_nvhpc.sh has its final make line modified to only build these two implementations but you can simply change the final line back to the singular `make` command without targets to build all of miniWeather's executables or specify different targets as desired.

Test that MiniWeather Builds Correctly

First, let's build MiniWeather using cmake_casper_nvhpc.sh and run some tests to make sure the model builds correctly. To note, we will focus on the `FORTRAN` and `OpenACC` versions of the model but MiniWeather is also a great tool for exploring other programming models like `C` and `C++`, each utilizing `MPI`, `OpenMP`, or `do concurrent` / `std::par`. See other language folders and associated programming paradigm source files for examples. A script file for Casper has been adapted for the other build folders to fascilitate this exploration if you would like to do this on your own time. See MiniWeather's README "Compiling and Running the Code" section for more info about this.

Initially, we will run this test with the base `mpi` model using miniWeather_mpi.F90 and the already implemented `openacc` model using miniWeather_mpi_openacc.F90. The file in fortran/build/cmake_casper_nvhpc.sh has its final make line modified to only build these two implementations but you can simply change the final line back to the singular `make` command without targets to build all of miniWeather's executables or specify different targets as desired.

In [ ]:

```
cd fortran/build
source cmake_casper_nvhpc.sh
cd ../..
# After running this, there will be the executables `mpi` and `openacc` in "fortran/build"
```

Validate the Executable

Now we can run validation tests on the compiled programs. Below, we use the check_output.sh script to do this. For miniWeather, you could also run `make` then `make test` to validate all the different executable types for the model. To note, we can set the environment variable `NVCOMPILER_ACC_TIME=1` in order to provide some contextual performance runtime information for comparison later.

First, run the serial `mpi_test`.

First, run the serial `mpi_test`.

In [ ]:

```
cd fortran/build
qcmd -A $PROJECT -q $QUEUE -l select=1:ncpus=1:ngpus=1 -l gpu_type=$GPU_TYPE -l walltime=60 -v NVCOMPILER_ACC_TIME=0 -- \
$PWD/check_output.sh $PWD/mpi_test 1e-13 4.5e-5
cd ../..
```

Now run the parallel `openacc_test` offloaded on a GPU. **Pay attention to how much faster this one runs**. Feel free to set `NVCOMPILER_ACC_TIME=1` to see more info about GPU compute kernel performance.

Now run the parallel `openacc_test` offloaded on a GPU. **Pay attention to how much faster this one runs**. Feel free to set `NVCOMPILER_ACC_TIME=1` to see more info about GPU compute kernel performance.

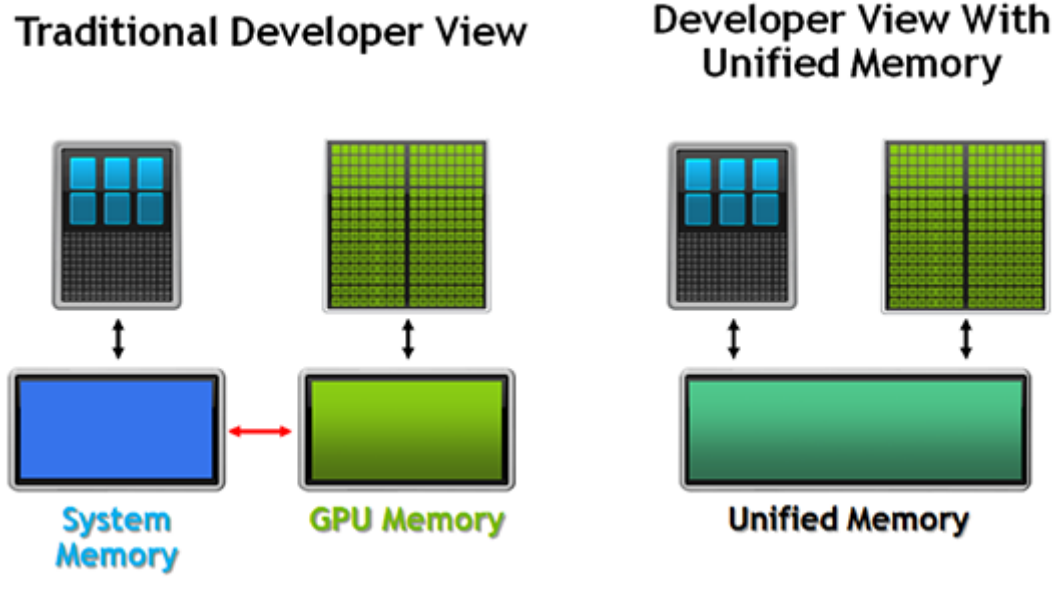
In [ ]:

```
cd fortran/build
qcmd -A $PROJECT -q $QUEUE -l select=1:ncpus=1:ngpus=1 -l gpu_type=$GPU_TYPE -l walltime=60 -v NVCOMPILER_ACC_TIME=0 -- \
$PWD/check_output.sh $PWD/openacc_test 1e-13 4.5e-5
cd ../..
```

OpenACC Directives - `kernels` and `parallel` with Unified Managed Memory

For reference, here is the OpenACC 2.7 Quick Reference Guide. You can also read through the official OpenACC 3.1 Full Standard Specification or OpenACC Programming and Best Practices Guide when your time allows.

We first introduce the `kernels` and `parallel` directives in OpenACC. For an in depth comparison, read the blog post OpenACC Kernels and Parallel Constructs by Michael Wolfe. In order to simplify initial work, we will use **managed memory**.



With managed memory, the address space between the CPU and the GPU is abstracted to one unified construct. This approach is optional for `OpenACC` & `OpenMP` using the flag `-gpu=managed` but is enabled by default for `stdPar`. To note, you may use data directives from `OpenACC` / `OpenMP` alongside `stdPar` with some compilers. With unified memory, the programmer does not need to worry about data movement explicitly. Instead, the runtime will automatically move data as needed between the CPU and GPU whenever a page fault occurs, ie GPU tries to access memory that is not available/updated in its physical high bandwidth memory space.

In later sessions, we will spend some time with `!$acc data` regions since using managed memory can lead to sub-optimal performance. To note, due to non-optimized data movement patterns, some kernels have already been specified with OpenACC directives in the MPI section of `subroutine set_halo_values_x(state)`.

Using the Descriptive `!$acc kernels` Compute Construct Directive

When you have a parallelizable loop or tightly nested loops, you can very easily **suggest** to the compiler to run it on the GPU using the **descriptive `!$acc kernels` directive**. When targetting NVIDIA GPUs, the compiler then essentially builds a CUDA kernel for you and assigns parallel execution in a close to optimal arrangement across gangs, workers, and vectors. This is a **descriptive** approach and the easiest way to port an application to a GPU using OpenACC.

With **descriptive** programming, **the compiler does all the heavy lifting** but regardless, only takes the directives as advice. If the compiler determines it can't parallelize your code, ie sees a potential data race, serial GPU code will be compiled instead. Thus, optimal performance is often difficult to achieve with the `!$acc kernels` directive alone as the compiler often won't parallelize code and instead would benefit from additional information provided about each compute region. Here are some important points:

- Use `!$acc kernels` and `!$acc end kernels` to encapsulate loop(s) or multiple sets of loop(s) that run in sequence
    - Each loop or nested loop set must not have any data dependencies between loop iterations that, for example, would cause a data race condition.
- By default, there is an **implicit barrier** at the end of a parallel execution region. Host thread execution will pause until the kernel completes
    - Use the `async()` and `wait()` clauses to permit asynchronous execution, discussed at future session
- You may specify `num_gangs()`, `num_workers()`, and `vector_length()` but only the compiler gets to decide which execution type is appied to each loop level(s) within an `!$acc kernels` region.
- You may not nest compute construct regions. Only one `kernels`, `parallel`, or `serial` context may be in scope at a time

An example of using the `!$acc kernels` in FORTRAN is below:

```
!$acc kernels [optional clauses]
    do i = 1, n
        do j = 1, m
            ...
        enddo
    enddo
!$acc end kernels
```

EXERCISE: Autoparallelization Using `!$acc kernels` Descriptive Directive

Add `!$acc kernels` regions the TODO sections of miniWeather_mpi_exercise.F90 (use `CTRL+F` or `CMD+F` TODO). You may enter a `[x]` in the raw text of this cell to track when you're finished with each section. Some have already been completed for you.

- [ ] Line 225
- [ ] Line 274
- [ ] Line 306
- [ ] Line 334
- [x] Line 371
- [x] Line 454
- [x] Line 871

Once this is done, run the below commands to make a new executable from the exercise source file. To note, within the fortran/CMakeLists.txt at Line 147 we added:

- The `-gpu=managed` flag so that you do not have to worry about data movement yet during this exercise
- The `-Minfo=accel` flag so information about how the compiler is targetting the GPU is also printed during the make/compilation process.

Investigate the output from the `-Minfo=accel` compiler flag. **What types of parallelizations did the compiler find and perform? Which loops were not able to be parallelized by the compiler?**

Once this is done, run the below commands to make a new executable from the exercise source file. To note, within the fortran/CMakeLists.txt at Line 147 we added:

- The `-gpu=managed` flag so that you do not have to worry about data movement yet during this exercise
- The `-Minfo=accel` flag so information about how the compiler is targetting the GPU is also printed during the make/compilation process.

Investigate the output from the `-Minfo=accel` compiler flag. **What types of parallelizations did the compiler find and perform? Which loops were not able to be parallelized by the compiler?**

In [ ]:

```
make -C fortran/build openacc_ex openacc_test_ex
```

EXERCISE: Run the Autoparalellized OpenACC MiniWeather Program

Once you are satisfied with your changes and compiled the new executable from previous exercise, run the below cell to test to make sure you have not introduced any bugs. If you get stuck, check miniWeather_mpi_openacc.F90.

To note, we add here the environment variable `NVCOMPILER_ACC_TIME=1`. Use this to compare to previous timing results of the already implemented openacc program for each kernel. **Do you notice any timing differences?**

EXERCISE: Run the Autoparalellized OpenACC MiniWeather Program

Once you are satisfied with your changes and compiled the new executable from previous exercise, run the below cell to test to make sure you have not introduced any bugs. If you get stuck, check miniWeather_mpi_openacc.F90.

To note, we add here the environment variable `NVCOMPILER_ACC_TIME=1`. Use this to compare to previous timing results of the already implemented openacc program for each kernel. **Do you notice any timing differences?**

In [ ]:

```
cd fortran/build
qcmd -A $PROJECT -q $QUEUE -l select=1:ncpus=1:ngpus=1 -l gpu_type=$GPU_TYPE -l walltime=60 -v NVCOMPILER_ACC_TIME=1 -- \
$PWD/check_output.sh $PWD/openacc_test_ex 1e-13 4.5e-5
cd ../..
```

Once you are confident there are no bugs, you can run the next `qcmd` cell to check performance of the non-test program, `openacc_ex`. If you want to modify the resolution, simulation time, or problem type (see MiniWeather's Altering the Code's Configuration), you may edit the parameters at line 53 of fortran/miniWeather_mpi_exercise.F90. You will have to rebuild the executable using the below `make` command.

Once you are confident there are no bugs, you can run the next `qcmd` cell to check performance of the non-test program, `openacc_ex`. If you want to modify the resolution, simulation time, or problem type (see MiniWeather's Altering the Code's Configuration), you may edit the parameters at line 53 of fortran/miniWeather_mpi_exercise.F90. You will have to rebuild the executable using the below `make` command.

In [ ]:

```
make -C fortran/build openacc_ex
```

Once you are confident there are no bugs, you can run the next `qcmd` cell to check performance of the non-test program, `openacc_ex`. If you want to modify the resolution, simulation time, or problem type (see MiniWeather's Altering the Code's Configuration), you may edit the parameters at line 53 of fortran/miniWeather_mpi_exercise.F90. You will have to rebuild the executable using the below `make` command.

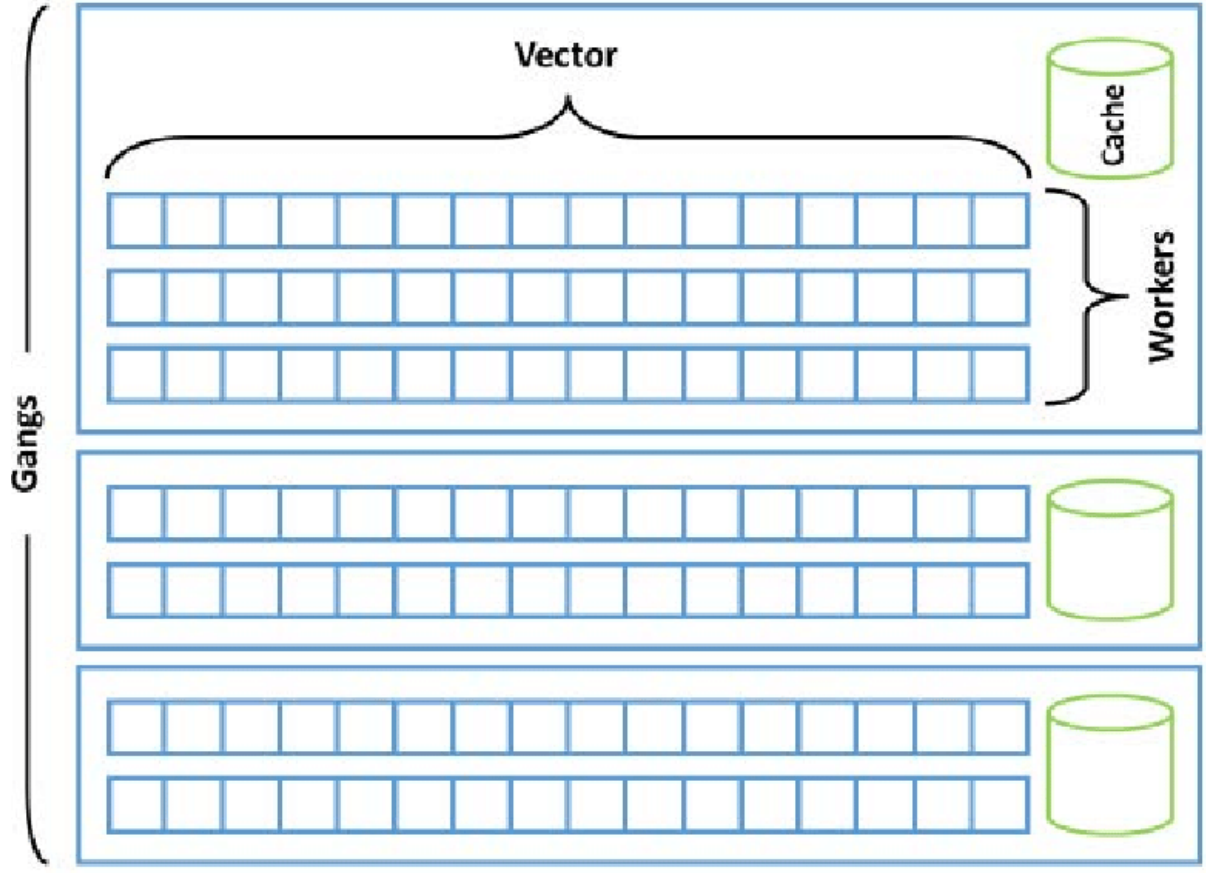
In [ ]:

```
make -C fortran/build openacc_ex
```

In [ ]:

```
cd fortran/build
qcmd -A $PROJECT -q $QUEUE -l select=1:ncpus=1:ngpus=1 -l gpu_type=$GPU_TYPE -l walltime=60 -- \
mpiexec -n 1 $PWD/openacc_ex
cd ../..
```

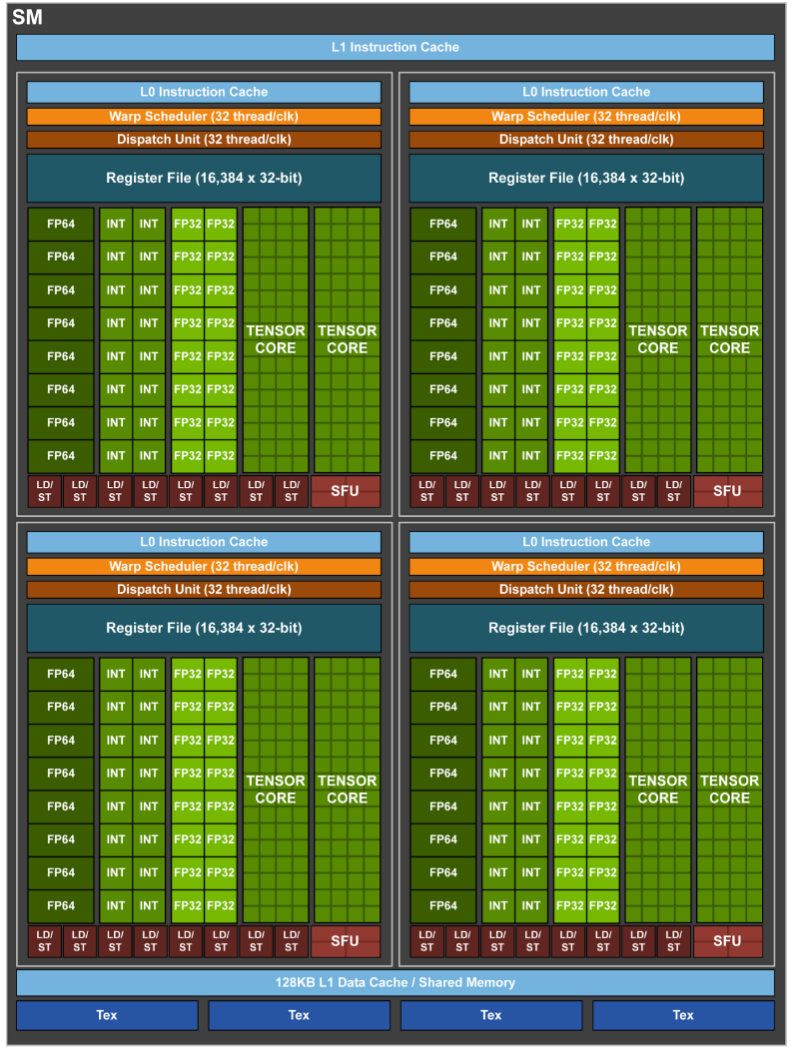
GPU Execution Task Granularity



From "Porting LASG/ IAP Climate System Ocean Model to GPUs Using OpenACC" Jiang, et al 2019

OpenACC, regardless if you specify it manually or not, schedules execution threads organized in gangs, workers, and vectors structures. Each generation of hardware has different hardware configurations that alter the limitations, arrangement, and number allowed of each of these execution structures.

One important detail to remember, especially if you're interested in optimization techniques, is that **each gang is assigned it's own shared memory/L1 cache in the SM**. Each gang, or thread block in CUDA terms, does not migrate between SMs. An architecture diagram for an NVIDIA V100 SM follows:


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Using the `!$acc parallel` and `!$acc loop` Compute Construct Directives

You can do a little extra work and instead use the more **prescriptive** `!$acc parallel` **directive** and **clasues** like `loop`. This directive expects more direction from the programmer to specify how the parallel work takes place and tends to perform better than `!$acc kernels`, however automatic parallelization analysis from the compiler still takes place for each. One key difference however is that a `!$acc kernels` construct can encapsulate multiple distinct parallel regions, ie compute kernels, and then create either a fused single kernel or multiple kernels depending on compiler analysis while a `!$acc parallel` region can only specify one singular CUDA kernel compute region at a time, typically at the default `gang` level without any additional clauses.

An example of using the `!$acc parallel` in FORTRAN is below:

```
!$acc parallel loop collapse(2) reduction(+:sum)
    do i = 1, n
        do j = 1, m
            ...
            sum = sum + tke
        enddo
    enddo
!$acc end parallel
```

More important however are these added **clauses** you should specify alongside the compute regions. Here are some clause examples to consider:

- You may use `!$acc loop ...` within a `!$acc kernels/parallel ...` region on specific loop(s) or alongside a compute construct directive such as `!$acc parallel loop ...`.
- Use `!$acc parallel loop [gang/worker/vector]` to distribute work in...
    - **Gangs**: across GPU's SMs. Each gang has a shared memory cache. When not collapsing loops, is usualy recommended for the outermost loop
    - **Workers**: within GPU's SMs. Each worker executes a vector and is one or multiple warps. This clause is often not used with only two levels of parallelism/for loops
    - **Vectors**: within GPU's SMs. Should be of the order of SIMT or CPU like SIMD operations, ie length 128 or in multiples of warp size 32

- Use `!$acc parallel loop collapse(N)` to unroll tightly N nested loops into one large loop to equally distribute work across GPU
    - Best utilized when your innermost loop is not of optimal size for vector work
    - Often useful when there are more than three levels of parallelism that are beyond a GPU's typical three levels of parallelism
    - Provides more flexibility towards dynamic loop dimension sizes

- Use `!$acc parallel loop reduction(op:var)` to indicate that a reduction should be performed on a variable to avoid a race condition
    - op -> `+` `*` `max` `min` `iand` `ior` `ieor` `.and.` `.or.` `.eqv.` `.neqv.`
    - var -> A scalar varaiable
    - May also use `!$acc atomic update` construct, paricular to wrap around an array type object that must avoid updating the same memory locations.

- Use `!$acc parallel private(var1,var2,...)` to specify that each variable listed should be private to whichever execution level scheduled.
    - If you use `!$acc parallel loop gang private(var1,var2,...)`, each variable will be private to each gang.
    - If you use `!$acc parallel loop vector private(var1,var2,...)`, each variable will be private to each thread associated with each vector lane.

Using the `!$acc serial` Compute Construct Directive

Sometimes its useful to specify a serial compute region that will run on the GPU. This is most useful in cases where

- The size of the loops is not ideal for the high level of parallelism achieveable by a GPU, ie O(10-100) loop size, but the cost to move data between GPU and host outweighs the performance gain to let the serial code optimized CPU host do the work.
- There is non-loop serial code that, though would run redundantly on GPU, would be better kept on GPU to avoid data transfers.

Essentially, `!$acc serial` always executes with a single gang of a single worker with a vector length of one. MiniWeather does not have any good examples of this use case, but you may try to play around with this concept, particularly on the serial code sections between for loops, to investigate how it changes performance.

EXERCISE: Parallelization Using the `!$acc parallel` Prescriptive Directive

Modify the previously added `!$acc kernels` regions to provide more information to the compiler using the `!$acc parallel` directive. Again, look for the TODO sections of fortran/miniWeather_mpi_exercise.F90 (use `CTRL+F` or `CMD+F` TODO). You may enter a `[x]` in the raw text of this cell to track when you're finished with each section.

- [ ] Line 225
- [ ] Line 274
- [ ] Line 306
- [ ] Line 334
- [ ] Line 371
- [ ] Line 454
- [ ] Line 871

Investigate the output from the `-Minfo=accel` compiler flag. **What types of parallelizations did the compiler find and perform? Any improvements from last time?**

Investigate the output from the `-Minfo=accel` compiler flag. **What types of parallelizations did the compiler find and perform? Any improvements from last time?**

In [ ]:

```
make -C fortran/build openacc_ex openacc_test_ex
```

EXERCISE: Run Your Improved OpenACC Program

Once you are satisfied with your changes and compiled the new executable from previous exercise, run the below cell to test to make sure you have not introduced any bugs.

To note, we enable here the environment variable `NVCOMPILER_ACC_TIME=1`. Use this to compare to previous timing results of each kernel. **Do you notice any timing differences?**

EXERCISE: Run Your Improved OpenACC Program

Once you are satisfied with your changes and compiled the new executable from previous exercise, run the below cell to test to make sure you have not introduced any bugs.

To note, we enable here the environment variable `NVCOMPILER_ACC_TIME=1`. Use this to compare to previous timing results of each kernel. **Do you notice any timing differences?**

In [ ]:

```
cd fortran/build
qcmd -A $PROJECT -q $QUEUE -l select=1:ncpus=1:ngpus=1 -l gpu_type=$GPU_TYPE -l walltime=60 -v NVCOMPILER_ACC_TIME=1 -- \
$PWD/check_output.sh $PWD/openacc_test_ex 1e-13 4.5e-5
cd ../..
```

Once you are confident there are no bugs, you can run the next `qcmd` cell to check performance of the non-test program, `openacc_ex` (or simply use the `openacc` program for an already fully optimized version). If you want to modify the resolution, simulation time, or problem type (see Altering the Code's Configuration), you may edit the parameters at line 53 of fortran/miniWeather_mpi_exercise.F90. You will have to rebuild the executable using the below `make` command.

Results of the program may be viewed by initiating a ssh X session with Casper on a terminal `ssh -Y [username]@casper.ucar.edu` then running `module load ncview` and `ncview output.nc` on the output file that should now be in your `$HOME` directory or the folder where you ran the MiniWeather executable from.

Once you are confident there are no bugs, you can run the next `qcmd` cell to check performance of the non-test program, `openacc_ex` (or simply use the `openacc` program for an already fully optimized version). If you want to modify the resolution, simulation time, or problem type (see Altering the Code's Configuration), you may edit the parameters at line 53 of fortran/miniWeather_mpi_exercise.F90. You will have to rebuild the executable using the below `make` command.

Results of the program may be viewed by initiating a ssh X session with Casper on a terminal `ssh -Y [username]@casper.ucar.edu` then running `module load ncview` and `ncview output.nc` on the output file that should now be in your `$HOME` directory or the folder where you ran the MiniWeather executable from.

In [ ]:

```
make -C fortran/build openacc_ex
```

Once you are confident there are no bugs, you can run the next `qcmd` cell to check performance of the non-test program, `openacc_ex` (or simply use the `openacc` program for an already fully optimized version). If you want to modify the resolution, simulation time, or problem type (see Altering the Code's Configuration), you may edit the parameters at line 53 of fortran/miniWeather_mpi_exercise.F90. You will have to rebuild the executable using the below `make` command.

Results of the program may be viewed by initiating a ssh X session with Casper on a terminal `ssh -Y [username]@casper.ucar.edu` then running `module load ncview` and `ncview output.nc` on the output file that should now be in your `$HOME` directory or the folder where you ran the MiniWeather executable from.

In [ ]:

```
make -C fortran/build openacc_ex
```

In [ ]:

```
cd fortran/build
qcmd -A $PROJECT -q $QUEUE -l select=1:ncpus=1:ngpus=1 -l gpu_type=$GPU_TYPE -l walltime=60 -- \
mpiexec -n 1 $PWD/openacc_ex
cd ../..
```

Suggested Resources

- Matt Norman's A Practical Introduction to GPU Refactoring in FORTRAN with Directives for Climate
- May 2021, OpenACC Programming and Best Practices Guide and Github
- OpenACC 2.7 Quick Reference Guide
- Official OpenACC 3.1 Full Standard Specification - Not all updated features are implemented yet by compatible compilers
- If you want to dive deep into lower level control and optimization of GPU performance, check out Oak Ridge National Lab's CUDA Training Series.