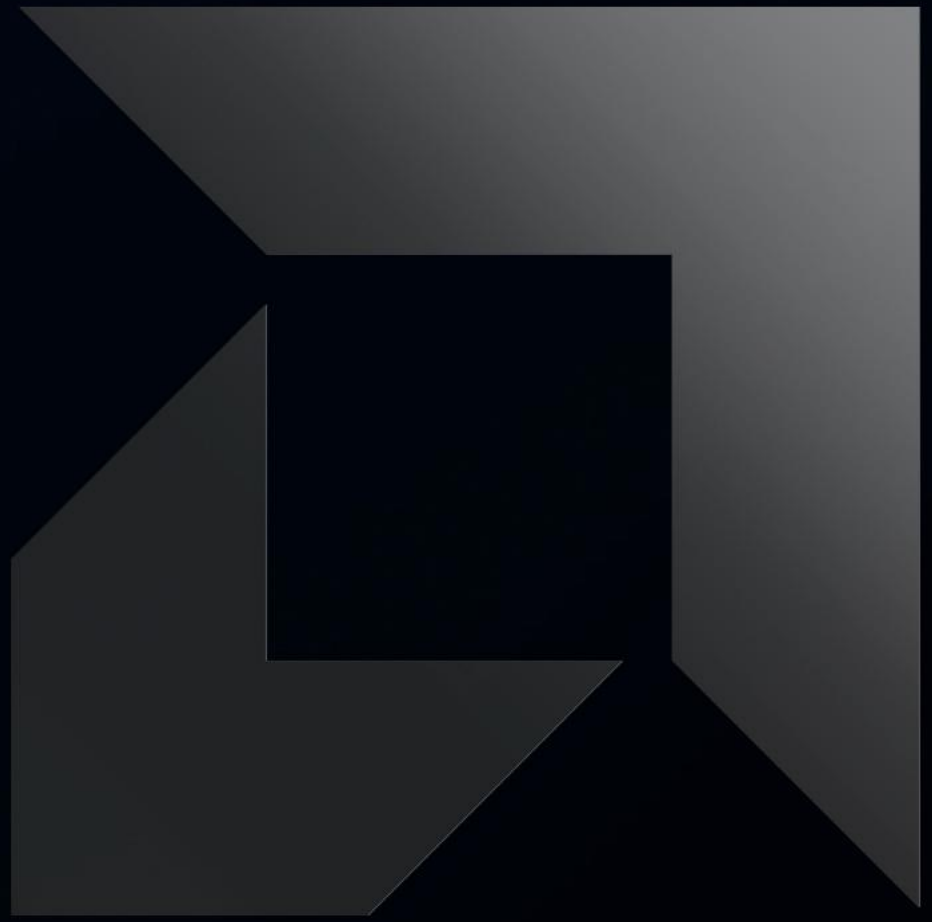


An Introduction to OpenMP® Programming on GPUs

UCL

DiRAC TRAINING ACADEMY

AMD together we advance_

What is covered in the Introduction to OpenMP® on GPUs presentation

- Learning the basis for the OpenMP language and standard with respect to GPUs
- Available compilers for OpenMP offload to GPUs
- Required compiler flags for OpenMP offload to GPUs
- How to incorporate OpenMP compiling and linking into Make and CMake build systems
- Conceptual understanding of Host and Device and offloading parallel work to Device for computation
- Basic understanding of what are the appropriate workloads for GPUs

OpenMP® : high level view

Fortran:

```
!$omp parallel do private(i) shared(NI,…)
do i=1,NI
  …
end do
!$omp end parallel
```

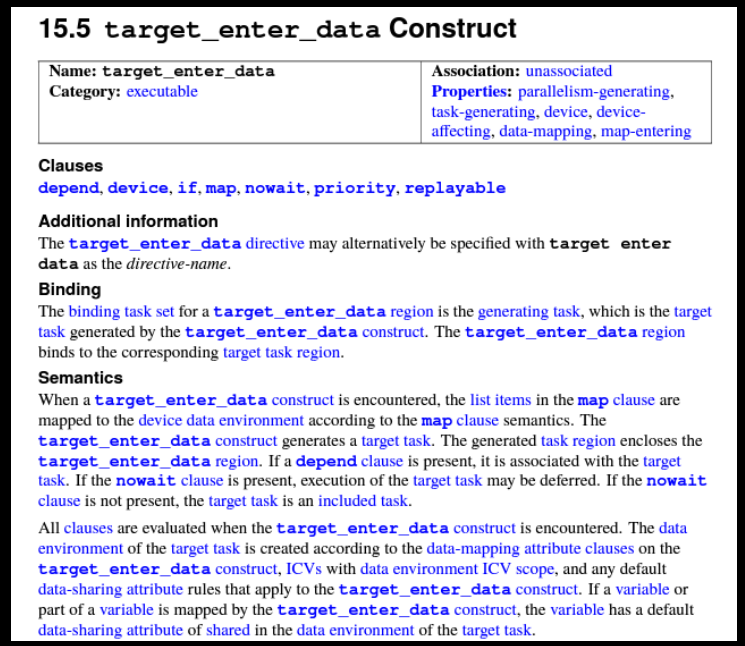
C/C++:

```
#pragma omp parallel for private(i) shared(NI,…)
for(int i=0, i<NI, i++){
…
}
```

- Why use OpenMP for porting to GPUs?
  - ✓ OpenMP is standardized
  - ✓ Portable code
  - ✓ Fortran is supported
- Many HPC codes are already OpenMP (+MPI) parallelized on CPUs
  - ✓ Porting requires only a few code changes
  - ✓ Easy to learn
- Interoperability with HIP and ROCm libraries
  - ✓ Flexibility
- GPU code can be compiled for the CPU too
  - ✓ Start porting before having access to GPUs
  - ✓ Majority of correctness checks possible without access to GPUs

OpenMP® : documented and standardized

- https://www.openmp.org/specifications/
  - OpenMP 6.0 standard – Nov 2024
  - Reference Guide ("Cheat sheet")
  - Examples



15.5 `target_enter_data` Construct

Name: `target_enter_data`	Association: unassociated
Category: executable	Properties: parallelism-generating, task-generating, device, device-affecting, data-mapping, map-entering

Clauses

`depend`, `device`, `if`, `map`, `nowait`, `priority`, `replayable`

Additional information

The `target_enter_data` directive may alternatively be specified with `target enter data` as the *directive-name*.

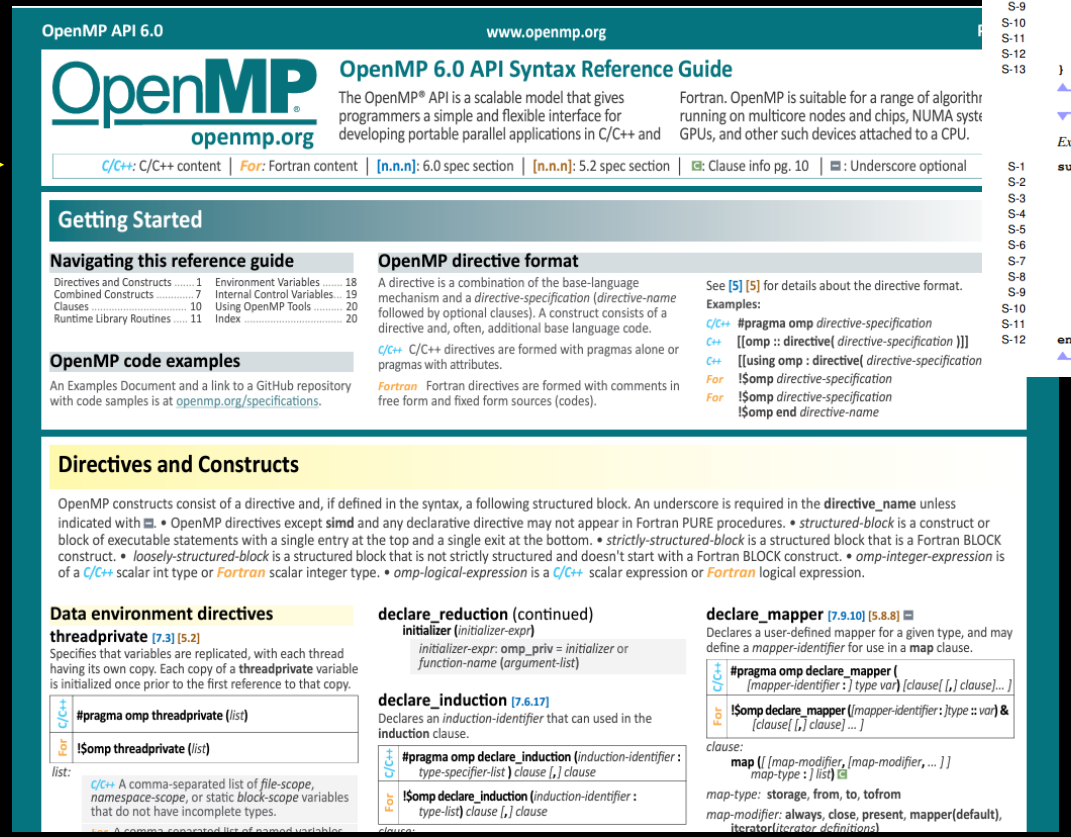
Binding

The binding task set for a `target_enter_data` region is the generating task, which is the target task generated by the `target_enter_data` construct. The `target_enter_data` region binds to the corresponding target task region.

Semantics

When a `target_enter_data` construct is encountered, the list items in the `map` clause are mapped to the device data environment according to the `map` clause semantics. The `target_enter_data` construct generates a target task. The generated task region encloses the `target_enter_data` region. If a `depend` clause is present, it is associated with the target task. If the `nowait` clause is present, execution of the target task may be deferred. If the `nowait` clause is not present, the target task is an included task.

All clauses are evaluated when the `target_enter_data` construct is encountered. The data environment of the target task is created according to the data-mapping attribute clauses on the `target_enter_data` construct, ICVs with data environment ICV scope, and any default data-sharing attribute rules that apply to the `target_enter_data` construct. If a variable or part of a variable is mapped by the `target_enter_data` construct, the variable has a default data-sharing attribute of shared in the data environment of the target task.



OpenMP API 6.0 www.openmp.org

OpenMP 6.0 API Syntax Reference Guide

The OpenMP® API is a scalable model that gives programmers a simple and flexible interface for developing portable parallel applications in C/C++ and Fortran. OpenMP is suitable for a range of algorithm running on multicore nodes and chips, NUMA syste GPUs, and other such devices attached to a CPU.

C/C++: C/C++ content | For: Fortran content | [n.n.n]: 6.0 spec section | [n.n.n]: 5.2 spec section | : Clause info pg. 10 | : Underscore optional

Getting Started

Navigating this reference guide

Directives and Constructs ..... 1 | Environment Variables 18
Combined Constructs ..... 7 | Internal Control Variables 19
Clauses ..... 10 | Using OpenMP Tools 20
Runtime Library Routines ..... 11 | Index 20

OpenMP code examples

An Examples Document and a link to a GitHub repository with code samples is at openmp.org/specifications.

OpenMP directive format

A directive is a combination of the base-language mechanism and a *directive-specification* (*directive-name* followed by optional clauses). A construct consists of a directive and, often, additional base language code.

C/C++ C/C++ directives are formed with pragmas alone or pragmas with attributes.

Fortran Fortran directives are formed with comments in free form and fixed form sources (codes).

See [5] [5] for details about the directive format.

Examples:

C/C++ `#pragma omp` *directive-specification*
C++ `[[omp :: directive(` *directive-specification* `)]]`
C++ `[[using omp : directive(` *directive-specification*
For `!$omp` *directive-specification*
For `!$omp` *directive-specification*
`!$omp end` *directive-name*

Directives and Constructs

OpenMP constructs consist of a directive and, if defined in the syntax, a following structured block. An underscore is required in the **directive_name** unless indicated with . • OpenMP directives except **simd** and any declarative directive may not appear in Fortran PURE procedures. • *structured-block* is a construct or block of executable statements with a single entry at the top and a single exit at the bottom. • *strictly-structured-block* is a structured block that is a Fortran BLOCK construct. • *loosely-structured-block* is a structured block that is not strictly structured and doesn't start with a Fortran BLOCK construct. • *omp-integer-expression* is of a C/C++ scalar int type or Fortran scalar integer type. • *omp-logical-expression* is a C/C++ scalar expression or Fortran logical expression.

Data environment directives

threadprivate [7.3] [5.2]
Specifies that variables are replicated, with each thread having its own copy. Each copy of a **threadprivate** variable is initialized once prior to the first reference to that copy.

C/C++ `#pragma omp threadprivate (`*list*`)`
For `!$omp threadprivate (`*list*`)`

list: C/C++ A comma-separated list of *file-scope*, *namespace-scope*, or static *block-scope* variables that do not have incomplete types.

declare_reduction (continued)
initializer (*initializer-expr*)
initializer-expr: **omp_priv** = *initializer* or *function-name* (*argument-list*)

declare_induction [7.6.17]
Declares an *induction-identifier* that can used in the **induction** clause.

C/C++ `#pragma omp declare_induction (`*induction-identifier* : *type-specifier-list* `)` *clause* `[,]` *clause*
For `!$omp declare_induction (`*induction-identifier* : *type-list*`)` *clause* `[,]` *clause*

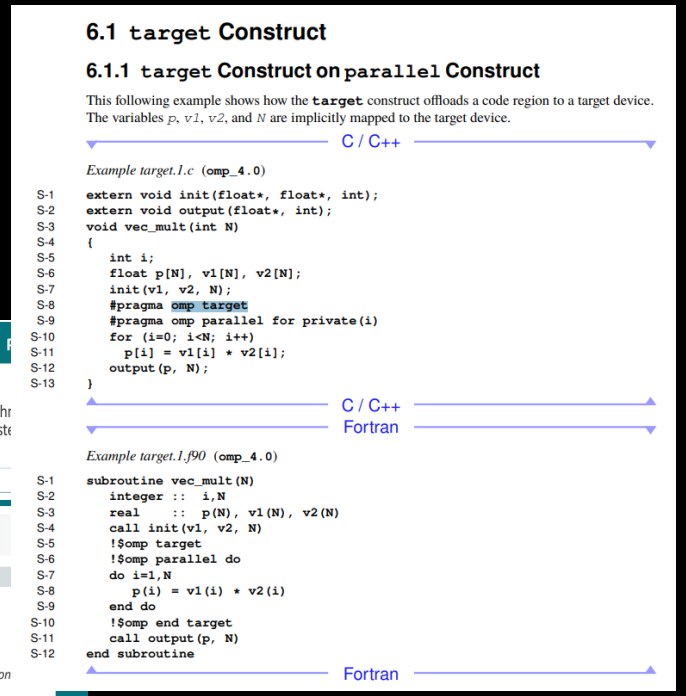
declare_mapper [7.9.10] [5.8.8]
Declares a user-defined mapper for a given type, and may define a *mapper-identifier* for use in a **map** clause.

C/C++ `#pragma omp declare_mapper (` *[mapper-identifier :] type var*`)` *[clause[[,] clause]...]*
For `!$omp declare_mapper (`*[mapper-identifier :]type :: var*`) &` *[clause[[,] clause] ...]*

clause:
map (*[[map-modifier,] [map-modifier,* ... *]] map-type :] list*)

map-type: **storage**, **from**, **to**, **tofrom**

map-modifier: **always**, **close**, **present**, **mapper(default)**, **iterator(***iterator-definitions***)**



6.1 `target` Construct

6.1.1 `target` Construct on `parallel` Construct

This following example shows how the `target` construct offloads a code region to a target device. The variables `p`, `v1`, `v2`, and `N` are implicitly mapped to the target device.

C / C++

Example target.1.c (`omp_4.0`)

```
S-1   extern void init(float*, float*, int);
S-2   extern void output(float*, int);
S-3   void vec_mult(int N)
S-4   {
S-5     int i;
S-6     float p[N], v1[N], v2[N];
S-7     init(v1, v2, N);
S-8     #pragma omp target
S-9     #pragma omp parallel for private(i)
S-10    for (i=0; i<N; i++)
S-11      p[i] = v1[i] * v2[i];
S-12    output(p, N);
S-13  }
```

C / C++
Fortran

Example target.1.f90 (`omp_4.0`)

```
S-1   subroutine vec_mult(N)
S-2     integer ::  i,N
S-3     real    ::  p(N), v1(N), v2(N)
S-4     call init(v1, v2, N)
S-5     !$omp target
S-6     !$omp parallel do
S-7     do i=1,N
S-8        p(i) = v1(i) * v2(i)
S-9     end do
S-10    !$omp end target
S-11    call output(p, N)
S-12  end subroutine
```

Fortran

AMD Compilers – two major strands

Primary Support

- LLVM™ based
  - ROCmCC provides support for both HIP and OpenMP®
    - hipcc -- /opt/rocm/bin
    - amdclang -- /opt/rocm/llvm/bin
  - AOMP: AMD OpenMP® research compiler for prototyping new features for ROCmCC

Functionality Only

- GCC based -- GNU Compilers
  - Provide offloading support to AMD GPUs (OpenMP®, OpenACC)
    - GCC 11 added offloading for the AMD MI100 GPUs
    - GCC 13 adds support for the AMD MI200 GPU series.
    - OG -- The devel/omp/gcc-13 (OG13) is the development
  - https://gcc.gnu.org/wiki/Offloading

Enabling OpenMP® on AMD Hardware

		LLVM		GCC	
Compiler Module ➡		amdclang/aomp/clacc		gcc/og	
		Command	Flags	Command	Flags
Language	C	amdclang clang	-fopenmp --offload-arch=<gfx###>	gcc	-fopenmp --foffload=-march=<gfx###>
	C++	amdclang++ clang++	-fopenmp --offload-arch=<gfx###>	g++	-fopenmp --foffload=-march=<gfx###>
	Fortran	amdflang(-new) flang(-new)	-fopenmp --offload-arch=<gfx###>	gfortran	-fopenmp --foffload=-march=<gfx###>

Offloading Target (CPU/GPU/GCD)	Architecture <gfx###>
AMD MI300	gfx942
AMD MI200 Series	gfx90a
AMD MI100	gfx908
Native Host (CPU)	-fopenmp-targets=amdgcn-amd-amdhsa

Sample OpenMP® Makefile

```
all: openmp_code

ROCM_GPU ?= $(strip $(shell rocminfo |grep -m 1 -E gfx[^0]{1} | sed -e 's/ *Name: *//'))

ifeq ($(findstring amdclang,$(CC1)), amdclang)
  OPENMP_FLAGS = -fopenmp --offload-arch=${ROCM_GPU}
else ifeq ($(findstring clang,$(CC1)), clang)
  OPENMP_FLAGS = -fopenmp --offload-arch=${ROCM_GPU}
else ifeq ($(findstring gcc,$(CC1)), gcc)
  OPENMP_FLAGS = -fopenmp -foffload=-march=${ROCM_GPU}
else ifeq ($(findstring CC,$(CC1)), CC)
  OPENMP_FLAGS = -fopenmp
  #the cray compiler decides the offload-arch by loading appropriate modules
  #module load craype-accel-amd-gfx942 for example
endif

CFLAGS = -g -O3 –fstrict-aliasing ${OPENMP_FLAGS}
LDFLAGS = ${OPENMP_FLAGS} -fno-lto -lm

openmp_code: openmp_code.o
        $(CC) $(LDFLAGS) $^ -o $@
```

Sample OpenMP® CMakeLists

```
cmake_minimum_required(VERSION 3.21 FATAL_ERROR)
project(Memory_pragmas LANGUAGES CXX)

set (CMAKE_CXX_STANDARD 17)

if (NOT CMAKE_BUILD_TYPE)
   set(CMAKE_BUILD_TYPE RelWithDebInfo)
endif(NOT CMAKE_BUILD_TYPE)

execute_process(COMMAND rocminfo COMMAND grep -m 1 -E gfx[^0]{1} COMMAND sed -e "s/ *Name: *//“
   OUTPUT_STRIP_TRAILING_WHITESPACE OUTPUT_VARIABLE ROCM_GPU)

string(REPLACE -O2 -O3 CMAKE_CXX_FLAGS_RELWITHDEBINFO ${CMAKE_CXX_FLAGS_RELWITHDEBINFO})
set(CMAKE_CXX_FLAGS_DEBUG "-ggdb")
set(CMAKE_CXX_FLAGS "-fstrict-aliasing -faligned-allocation -fnew-alignment=256")
if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fopenmp --offload-arch=${ROCM_GPU} -fstrict-aliasing")
elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU")
   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fopenmp -foffload=-march=${ROCM_GPU} -fstrict-aliasing")
elseif (CMAKE_CXX_COMPILER_ID MATCHES "Cray")
endif()

add_executable(kernel1 kernel1.cc)
```

Compile OpenMP® GPU code on CPUs for early porting efforts

- OpenMP provides a path to first port and test “fine-grained” parallelism on CPUs
  - Fine-grained parallelism is the splitting of a loop across threads and operating on subsets on any order

Now to move the same OpenMP CPU code to the GPU →

- Remove the --offload-arch=…, `-fopenmp-targets=amdgcn-amd-amdhsa`
- Compilers supporting OpenMP 4.0 or greater needed

- Helpful to check functionality “at home” before moving to a GPU cluster

- No duplication of code required to run on the CPU
  - Precompiler instructions (#ifdef …) or other means to distinguish code required if architecture specific optimizations are used

Check, if kernel actually runs on the GPU

- Different methods, e.g.
  - Does rocm-smi show any activity on the GPU?
    - works with large/multiple kernels
  - export LIBOMPTARGET_INFO=-1 (compiler dependent)
    - Before running the executable, shows what is moved to the device (GPU) at runtime
  - In the code:

omp_is_initial_device [18.7.6] [3.7.6]

Returns *true* if the current task is executing on the host device; otherwise, it returns *false*.

C/C++	int omp_is_initial_device (void);
Fortran	logical function omp_is_initial_device ()

Note:
initial device = CPU

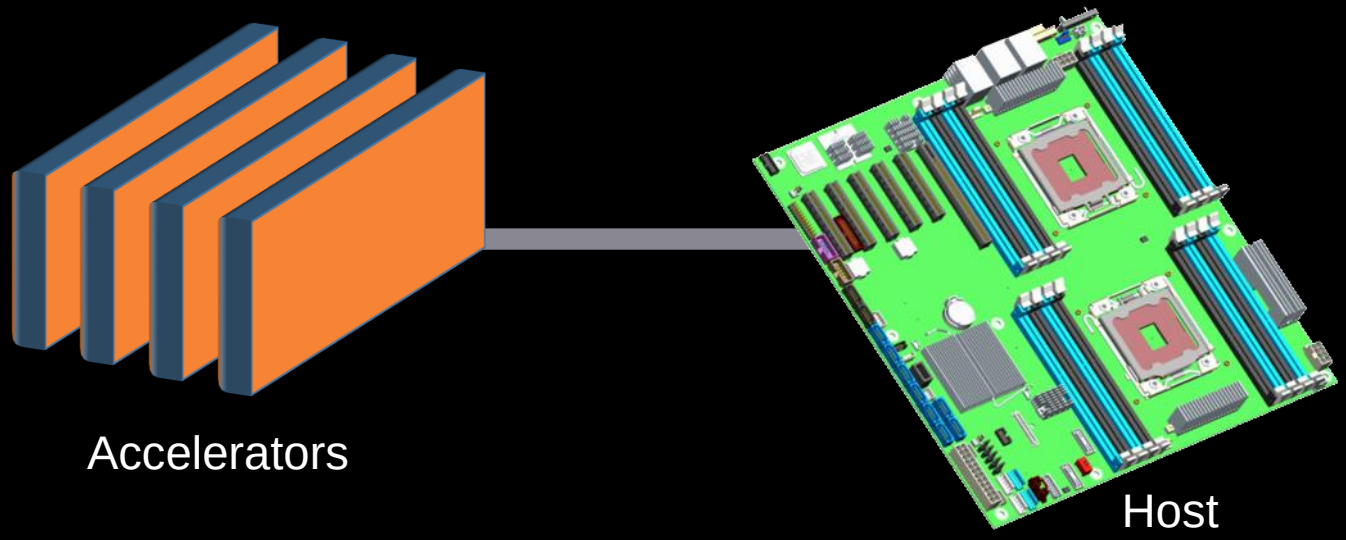
OpenMP® Device Model

- As of version 4.0 the OpenMP API supports accelerators/coprocessors.

- Device model:
  - One host for “traditional” multi-threading
  - Multiple accelerators/coprocessors of the same kind for offloading
  - Devices are accessible through a device ID (from 0 to *n*-1 for *n* devices)

- OpenMP device model is *agnostic* of actual technology. In theory, devices only need to:
  - Send data to and receive data from the host
  - Perform computation upon request

Optimizing Data Transfers is Key to Performance on discrete GPUs



- Connections between host and accelerator are typically lower-bandwidth, higher-latency interconnects
  - Bandwidth host memory: hundreds of GB/sec
  - Bandwidth accelerator memory: TB/sec
  - PCIe® Gen 4 bandwidth (16x): tens of GB/sec

- **Unnecessary data transfers must be avoided**, by
  - only transferring what is actually needed for the computation, and
  - making the lifetime of the data on the target device as long as possible.

Creating Parallelism on the Target Device

- OpenMP® **separates** **offload** to the GPU and **parallel** execution on the GPU
  - Programmers need to explicitly create parallel regions on the target device.
  - In theory, this can be combined with any OpenMP® construct.
  - In practice, there is only a useful subset of OpenMP® features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

When GPUs are the Device

- Not all applications are appropriate to run on GPUs – need to have a lot of parallel work
- Current high-end GPUs are capable of operating on upwards of 1000 separate blocks of computation
- Massively parallel work should be first targeted to offload to GPUs
- Serial work is usually slower on GPUs
- Often best to do serial or low-parallel work on GPU to avoid moving data back and forth from GPU to CPU
- Exceptions:
  - It may be faster to do light work on CPU when using True APUs where data does not have to move – MI300A fits this category
- Memory allocations are generally done on the CPU and can be significant bottlenecks

Summary for Introduction to OpenMP®

- AMD OpenMP compilers can offload computation to AMD GPUs
  - Good support for C and C++ languages
  - Being used in production by many applications
  - Newly added support for Fortran with AMD Next Gen Fortran compiler: https://rocm.blogs.amd.com/ecosystems-and-partners/fortran-journey/README.html
- Backed by an Industry language standard
  - Managed by the OpenMP Architecture Review Board
- Portability across GPU platforms for core OpenMP constructs
  - Tightly integrates with OpenMP multi-threading on the host
- Composability across programming languages (C,C++,Fortran)
  - Interoperable with some other GPU programming languages and libraries
- Offload to the GPU and parallel execution on the GPU are separate for OpenMP

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Third-party content is licensed to you directly by the third party that owns the content and is not licensed to you by AMD. ALL LINKED THIRD-PARTY CONTENT IS PROVIDED “AS IS” WITHOUT A WARRANTY OF ANY KIND. USE OF SUCH THIRD-PARTY CONTENT IS DONE AT YOUR SOLE DISCRETION AND UNDER NO CIRCUMSTANCES WILL AMD BE LIABLE TO YOU FOR ANY THIRD-PARTY CONTENT. YOU ASSUME ALL RISK AND ARE SOLELY RESPONSIBLE FOR ANY DAMAGES THAT MAY ARISE FROM YOUR USE OF THIRD-PARTY CONTENT.

LLVM™ is a trademark of LLVM Foundation
The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board
PCIe® is a registered trademark of PCI-SIG Corporation.