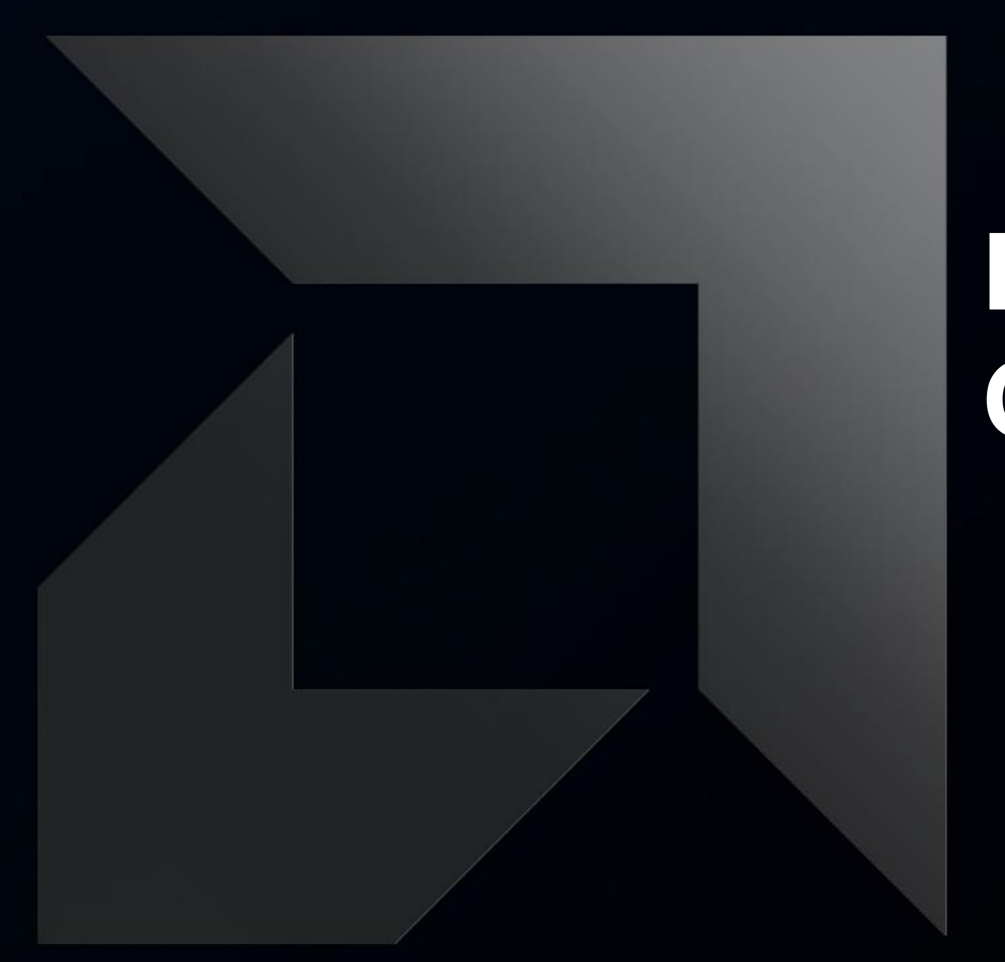
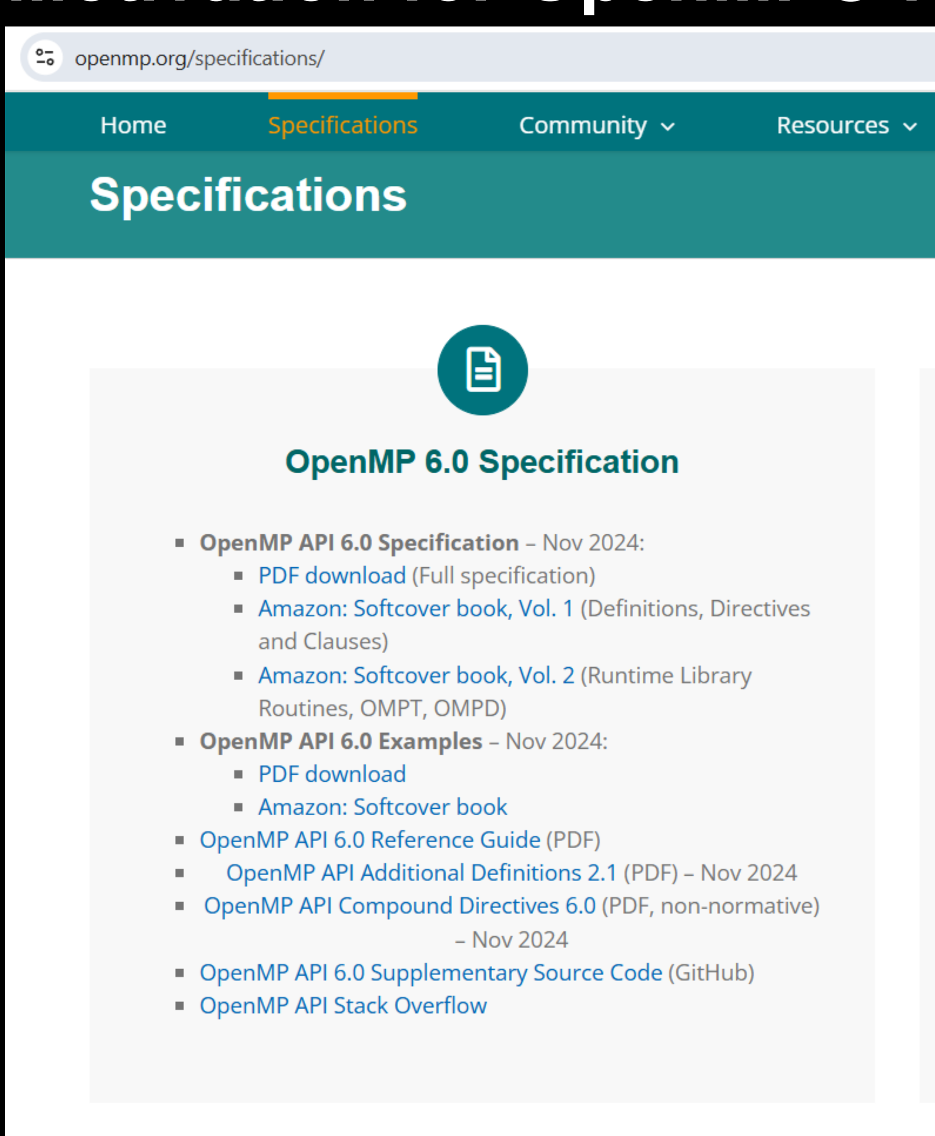


Introduction to OpenMP® Offload on AMD GPUs

Presenter: Johanna Potyka
AMD @ CASTIEL
Oct 28-30, 2025

Motivation for OpenMP® for GPUs



- Why use OpenMP for porting to GPUs?
  - ✓ OpenMP is standardized
  - ✓ Portable code
  - ✓ Fortran, C, and C++ supported
- Many HPC codes are already OpenMP (+MPI) parallelized on CPUs
  - ✓ Porting requires only 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

Revision: OpenMP® on CPUs

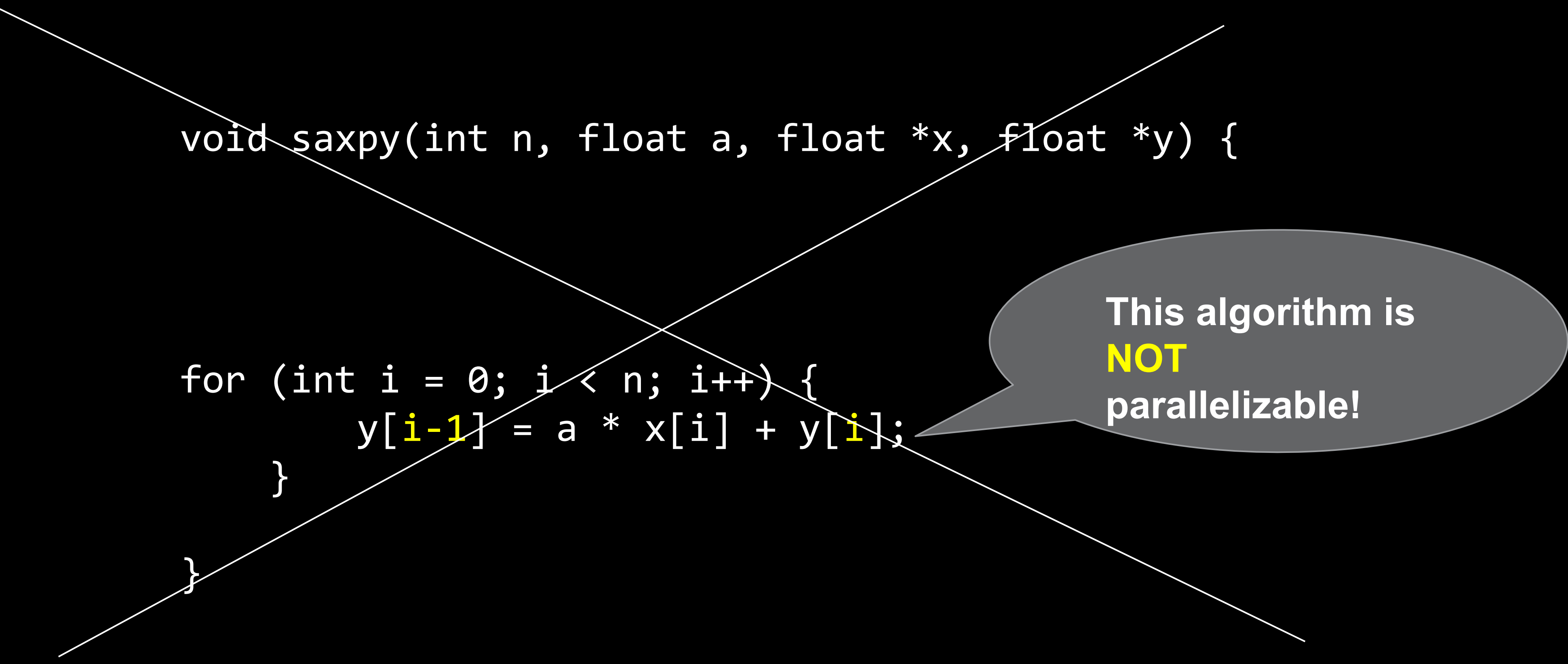
[Public]

OpenMP® on CPUs (C/C++)

```
void saxpy(int n, float a, float *x, float *y) {



    for (int i = 0; i < n; i++) {
            y[i] = a * x[i] + y[i];
        }

}
```

This algorithm is parallelizable!

No good candidate for OpenMP® on CPUs (C/C++)



Good candidate for OpenMP® on CPUs (C/C++)

```
void saxpy(int n, float a, float *x, float *y) {



for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }

}
```

This algorithm is parallelizable!

[Public]

OpenMP® on CPUs (C/C++)

```
void saxpy(int n, float a, float *x, float *y) {



#pragma omp parallel for
for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

OpenMP® on CPUs (C/C++)

Start a parallel region
“Use OMP_NUM_THREADS number of threads to execute the following code”

```
void saxpy(int n, float a, float      float *y) {

    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
            y[i] = a * x[i] + y[i];
        }
    }
```

“this line is OpenMP”

Distribute the for iterations among the threads
Thread 0 gets a chunk
Thread 1 gets a chunk
…
Distribution depends on schedule

[Public]

OpenMP® on CPUs (Fortran)

```
subroutine saxpy(a, x, y, n)
  ! Declarations omitted
  !$omp parallel do
  do i=1,n
        y(i) = a * x(i) + y(i)
  end do
  !$omp end parallel do
end subroutine
```

OpenMP® on CPUs (Fortran)

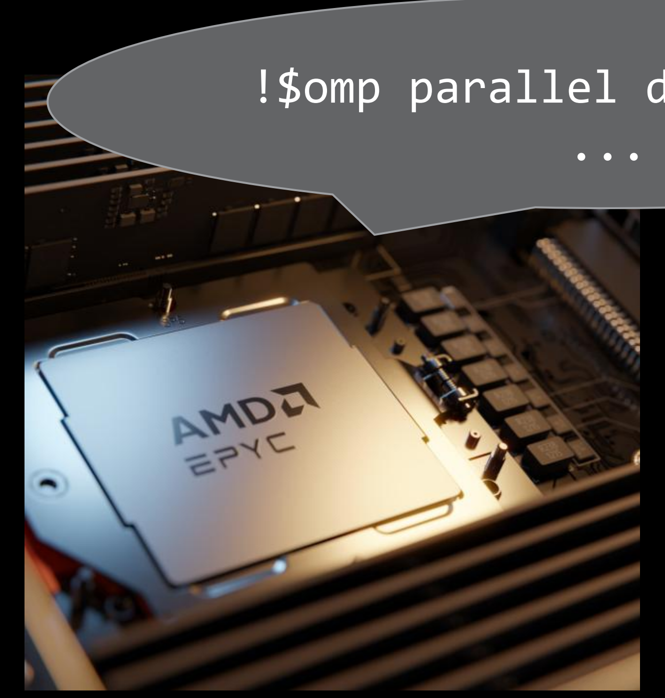
Start a parallel region
“Use OMP_NUM_THREADS number of threads to execute the following code”

“this line is OpenMP”

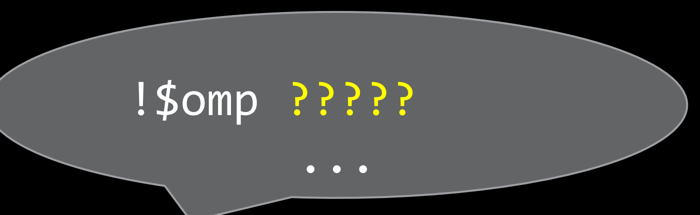
```
s   outine saxpy(a, x, y   )
  ! Declarations omi  ted
  !$omp parallel do
  do i=1,n
        y(i) = a * x(i)   y(i)
  end do
  !$omp end parallel do
end subroutine
```

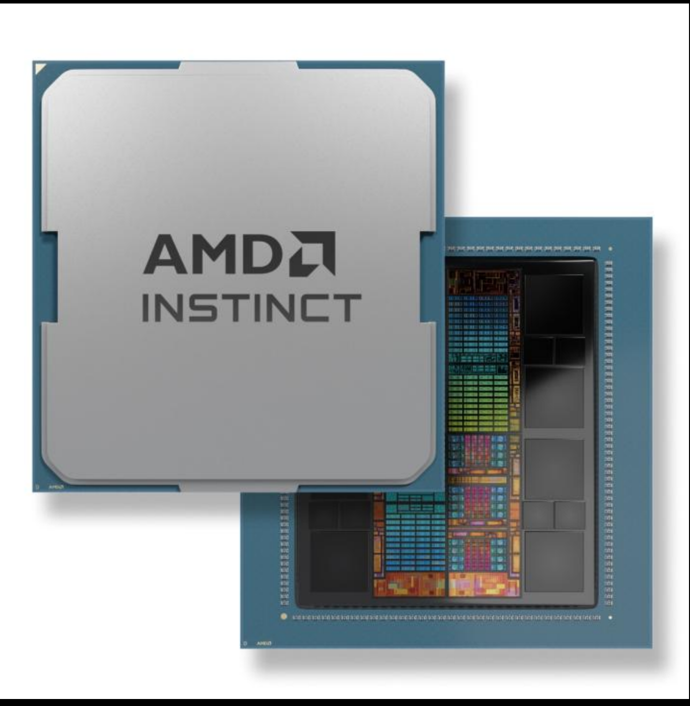
Distribute the do iterations among the threads
Thread 0 gets a chunk
Thread 1 gets a chunk
…
Distribution depends on schedule

How does this work on the GPU?



????





- ✓ OpenMP parallelization of loops for the CPU
  - Compile with -fopenmp
  - Before run e.g. export OMP_NUM_THREADS=24

- How does OpenMP work on the GPU?
  - Compilation slightly different
  - MUCH more parallelism
  - New OpenMP directives
  - CPU and GPU interaction

OpenMP® offload: Compilation for GPUs

[Public]

Compilers – two major strands

Primary Support

- LLVM™ based
  - ROCm compilers provide support for OpenMP®
    - amdclang -- /opt/rocm/llvm/bin
    - amdflang -- /opt/rocm/llvm/bin (starting from rocm 7.0)
  - AOMP: AMD OpenMP® research compiler for prototyping new features

➢ We will use compilers in ROCm in the exercises

Use LLVM based compilers for production (or Cray if available)

Functionality Only

- GCC based -- GNU Compilers
  - Provide offloading support to AMD GPUs
    - GCC 11 added offloading for the AMD MI100 GPUs
    - GCC 13 adds support for the AMD MI200 GPU series.
  - https://gcc.gnu.org/wiki/Offloading

amdflang beta pre-release of AMD Fortran compiler

Pre-releases are public on the AMD Infinity Hub: https://repo.radeon.com/rocm/misc/flang/

Installed on aac6 to use in this training:

module load amdflang-new/<version>

Module will set export FC=amdflang

AMD is working on a Fortran compiler and is ready to share a beta version. During the training, you will have hands-on access to the beta. Going forward, AMD will continue to provide early access via AFARs (Advanced Feature Access Releases) as improvements and features are added to the beta. **As with any beta, we are looking to gather feedback on functionality, usability and user experience.**

An earlier version of the AMD Next Generation Fortran Compiler is released in the ROCm 7.0 version!

Blog post about Next Generation Fortran compiler: https://rocm.blogs.amd.com/ecosystems-and-partners/fortran-journey/README.html

PRE-PRODUCTION SOFTWARE: The software accessible on this training may be a **pre-production version**, intended to provide advance access to features that may or may not eventually be included into production version of the software. Accordingly, pre-production software **may not be fully functional, may contain errors, and may have reduced or different security, privacy, accessibility, availability, and reliability standards** relative to production versions of the software. Use of pre-production software may result in unexpected results, loss of data, project delays or other unpredictable damage or loss. Pre-production software is not intended for use in production, and your use of pre-production software is at your own risk.

[Public]

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 Series	gfx942
AMD MI200 Series	gfx90a
AMD MI100	gfx908
Native Host (CPU)	-fopenmp-targets=amdgcn-amd-amdhsa

Sample OpenMP® Makefile

```
EXEC = reduction
default: ${EXEC}
all: ${EXEC}

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

CC1=$(notdir $(CC))

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

${EXEC}: ${EXEC}.o codelet.o
	$(CC) $(LDFLAGS) $^ -o $@

# Cleanup
clean:
	rm -f *.o ${EXEC}
```

[Public]

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)

string(REPLACE -O2 -O3 CMAKE_CXX_FLAGS_RELWITHDEBINFO ${CMAKE_CXX_FLAGS_RELWITHDEBINFO})
set(CMAKE_CXX_FLAGS_DEBUG "-ggdb")
if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
   # using Clang
   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fopenmp --offload-arch=gfx942 -fstrict-aliasing")
elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU")
   # using GCC
   set(CMAKE_CXX_FLAGS
       "${CMAKE_CXX_FLAGS} -fopenmp -foffload=-march=gfx942 -fstrict-aliasing -fopt-info-optimized-omp")
elseif (CMAKE_CXX_COMPILER_ID MATCHES "Cray")
endif()

add_executable(mem1 mem1.cc)
```

OpenMP® offload: Computation on GPUs

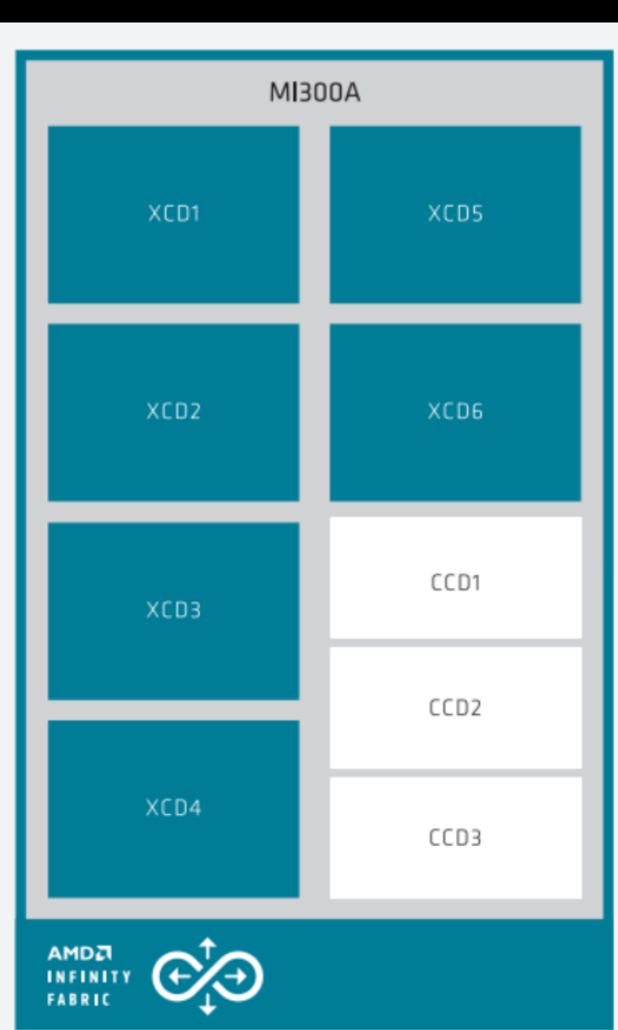
[Public]

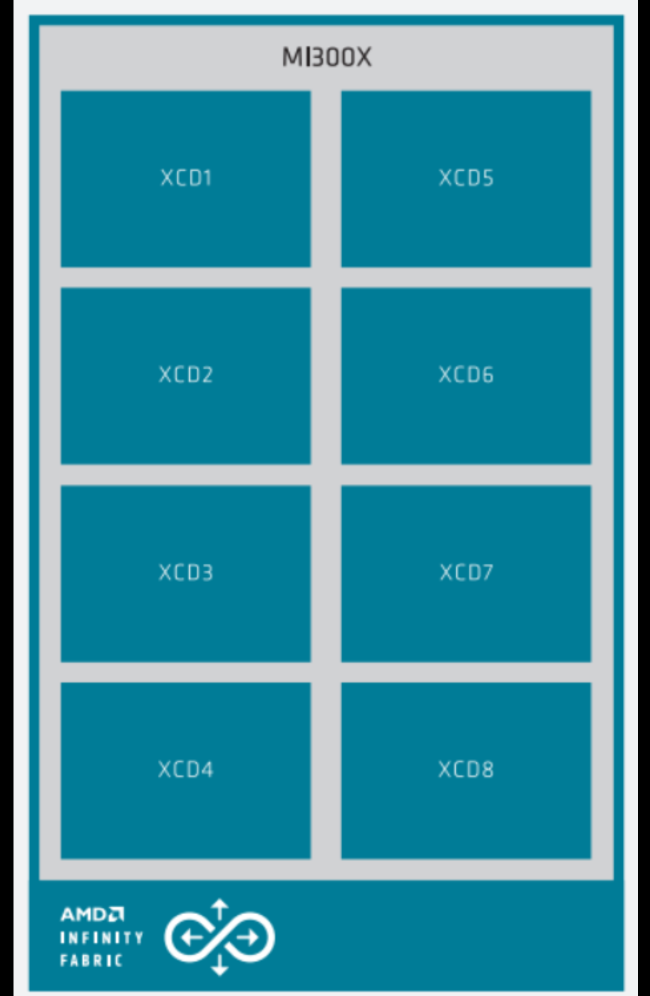
Parallelism in GPUs

6 x 38 = 228 CUs
228 x 64 = 14,592
Stream processors!

3 x 8 = 24 CPU cores

Shared Infinity cache
Shared HBM





Your **algorithm** needs to be at least that parallel (better more)!

Long loop counts needed!

8 x 38 = 304 CUs
304 x 64 = 19,456
Stream processors!

Discrete GPU

AMD
together we advance_

[Public]

OpenMP® on CPUs (Fortran)

```
subroutine saxpy(a, x, y, n)
  ! Declarations omitted
  !$omp parallel do
  do i=1,n
        y(i) = a * x(i) + y(i)
  end do
  !$omp end parallel do
end subroutine
```

[Public]

OpenMP® on GPUs (Fortran)

```
subroutine saxpy(a, x, y, n)
  ! Declarations omitted
  !$omp target teams distribute parallel do
  do i=1,n
        y(i) = a * x(i) + y(i)
  end do
  !$omp end parallel do
end subroutine
```


AMD
together we advance_

[Public]

OpenMP® on GPUs (Fortran)

If you forget any of those:
performance on GPU will be bad!

```
subroutine saxpy(a, x, y, n)
  ! Declarations omitted
  !$omp target teams distribute parallel do
  do i=1,n
        y(i) = a * x(i) + y(i)
  end do
  !$omp end parallel do
end subroutine
```

[Public]

“target” shifts the computation to the device, ...

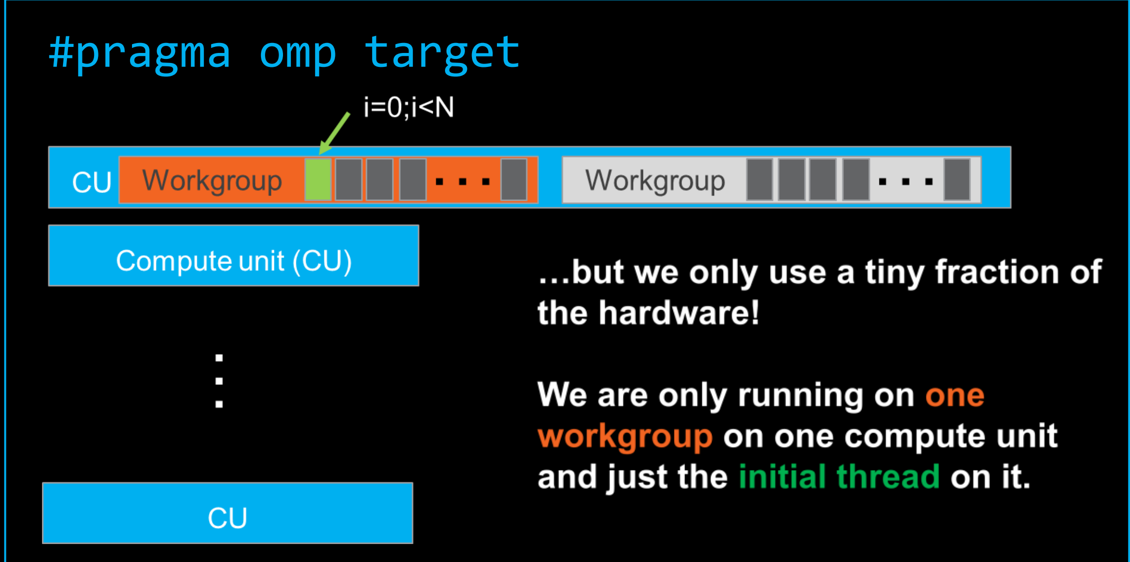
```
#pragma omp target
```


…but we only use a tiny fraction of the hardware!

We are only running on one workgroup on one compute unit and just the initial thread on it.

[Public]

Comparing subsets of GPU parallel compute directive



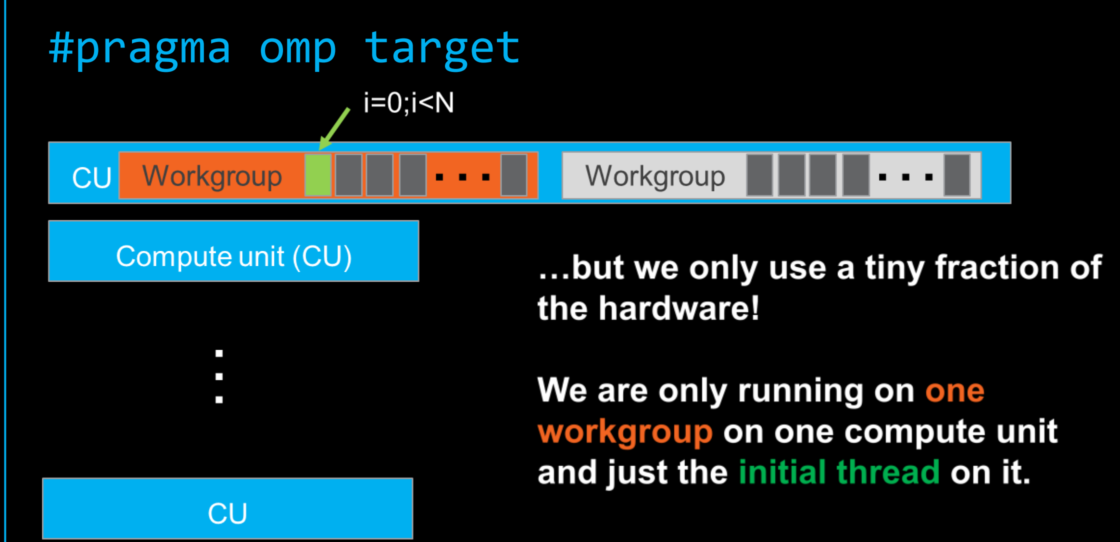
…but we only use a tiny fraction of the hardware!

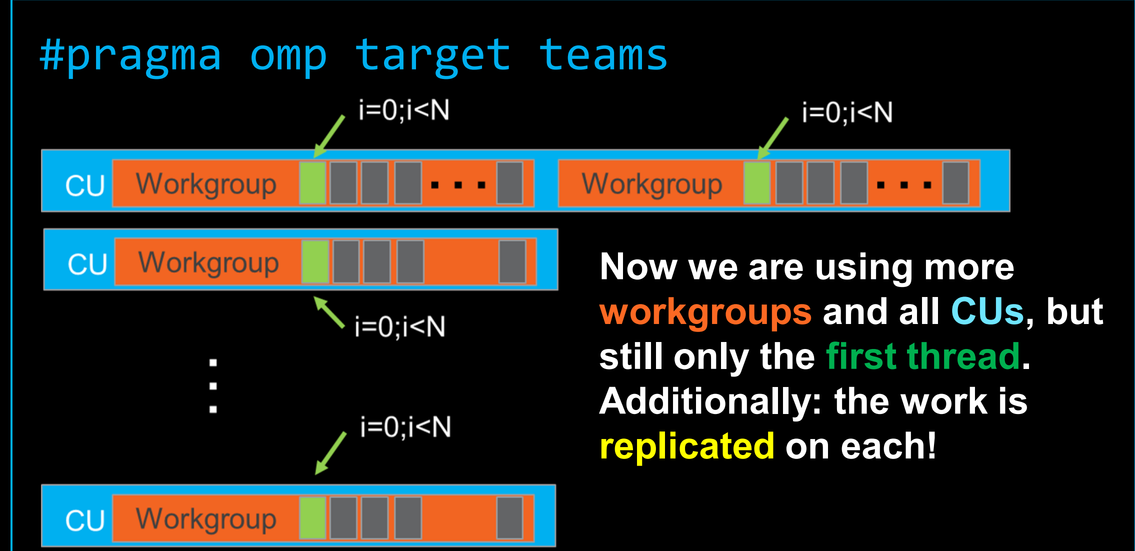
We are only running on one workgroup on one compute unit and just the initial thread on it.

AMD together we advance_

[Public]

Comparing subsets of GPU parallel compute directive

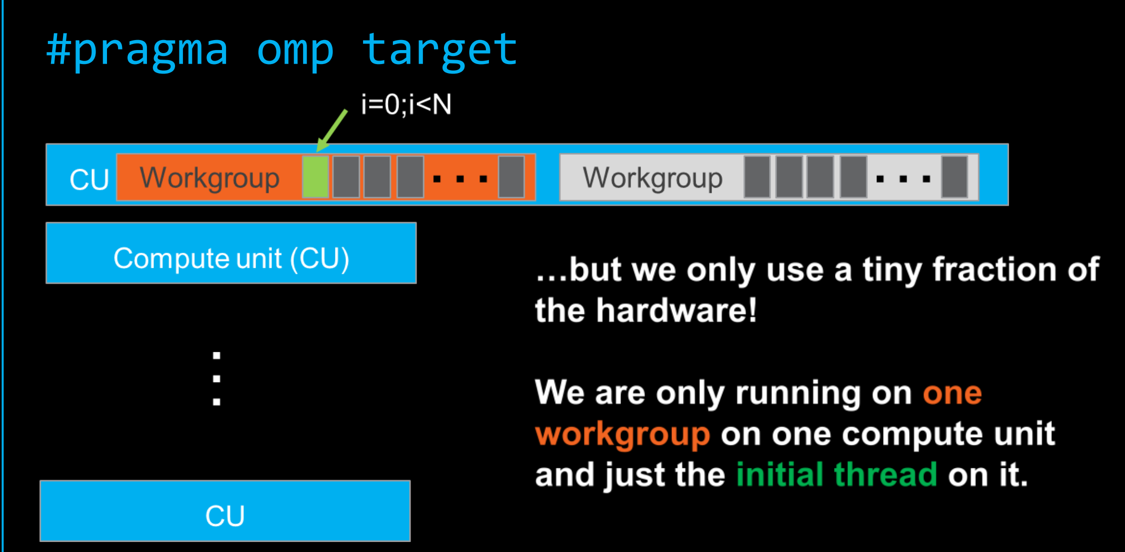




AMD
together we advance_

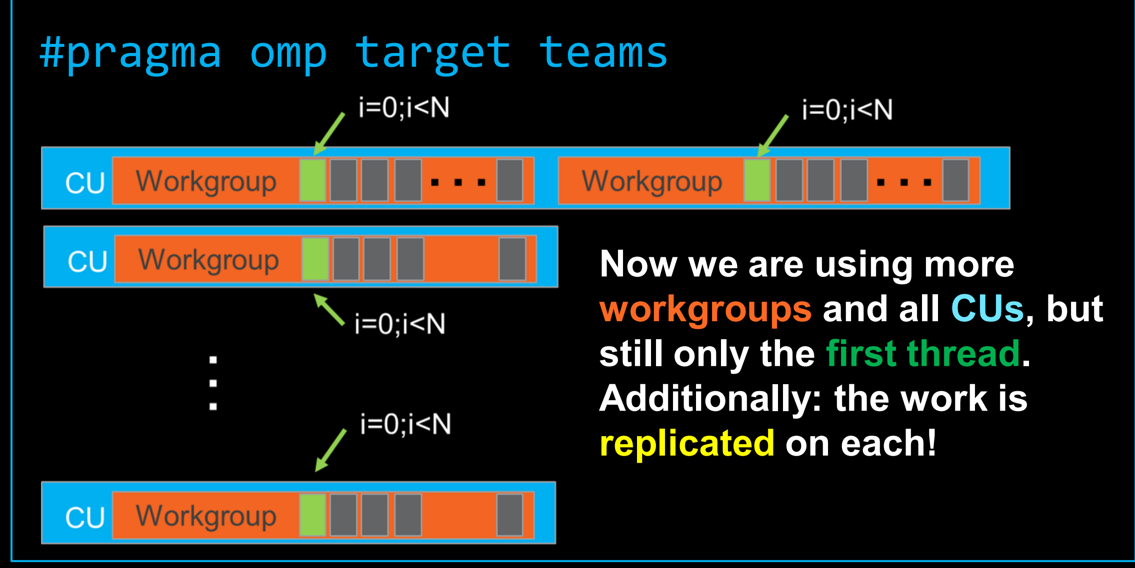
[Public]

Comparing subsets of GPU parallel compute directive

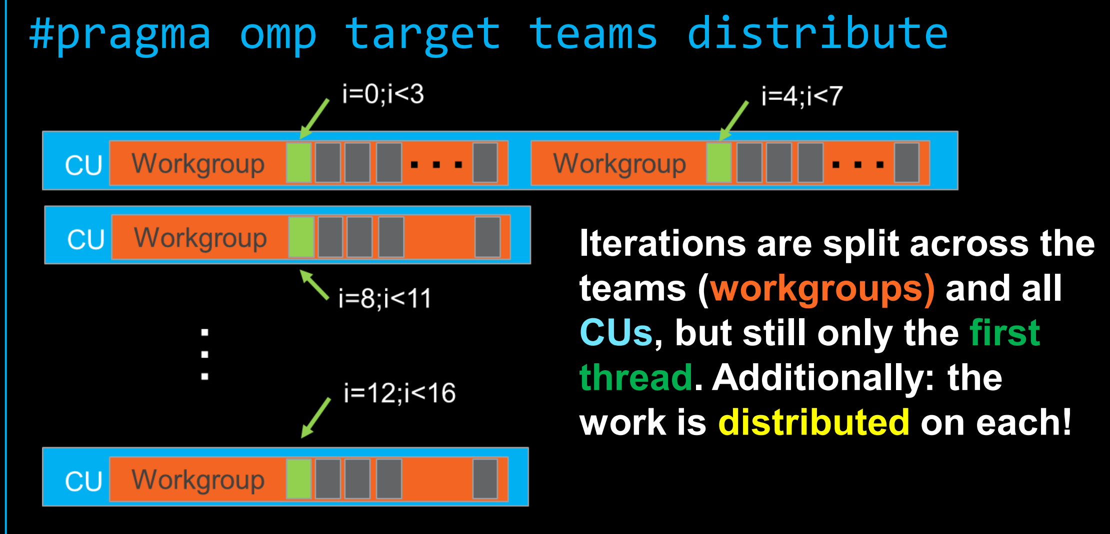


…but we only use a tiny fraction of the hardware!

We are only running on one workgroup on one compute unit and just the initial thread on it.

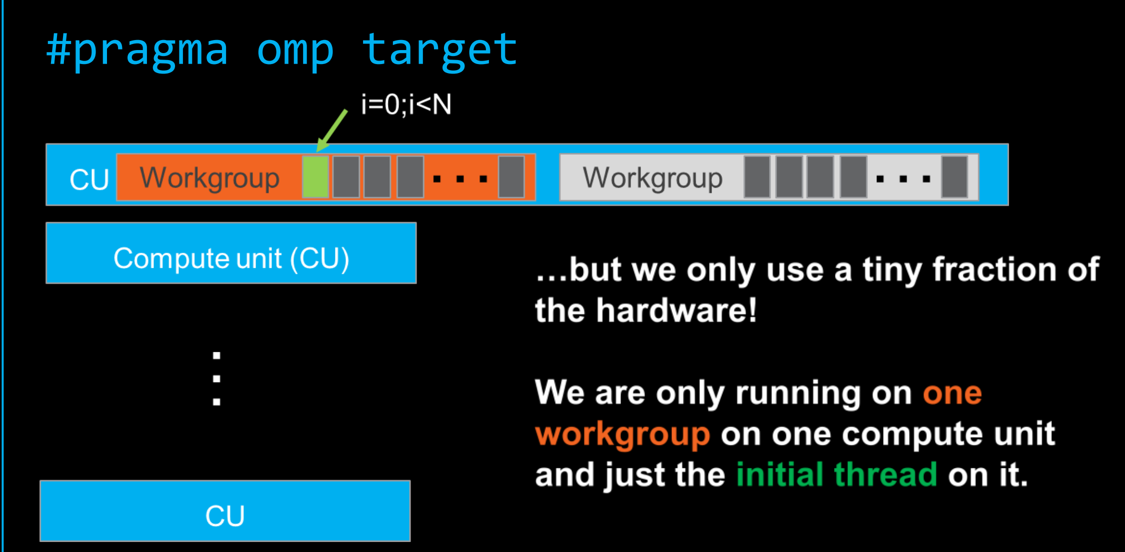


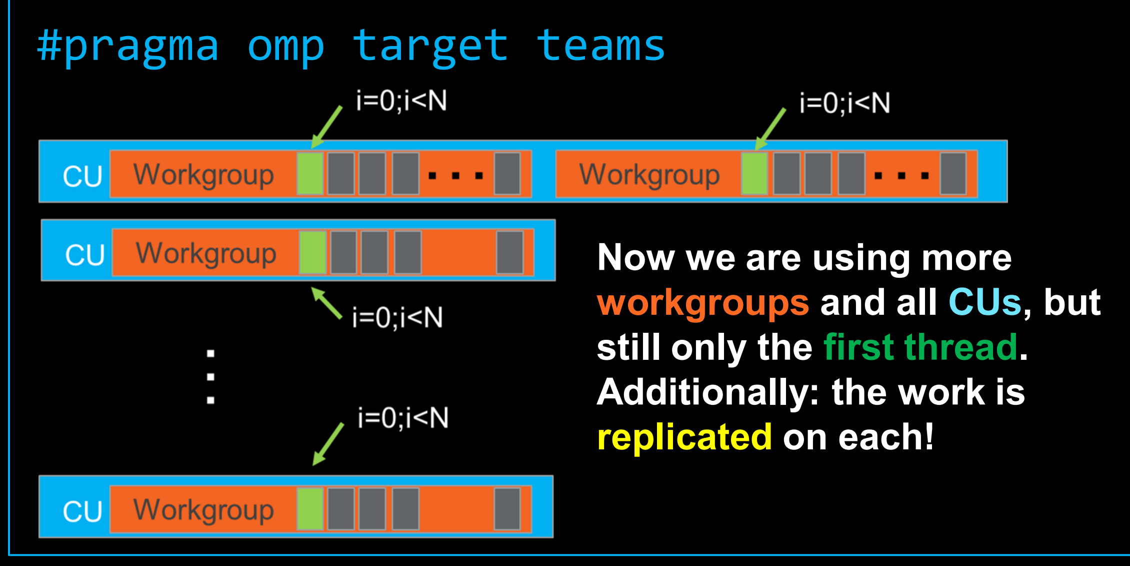
Now we are using more workgroups and all CUs, but still only the first thread. Additionally: the work is replicated on each!

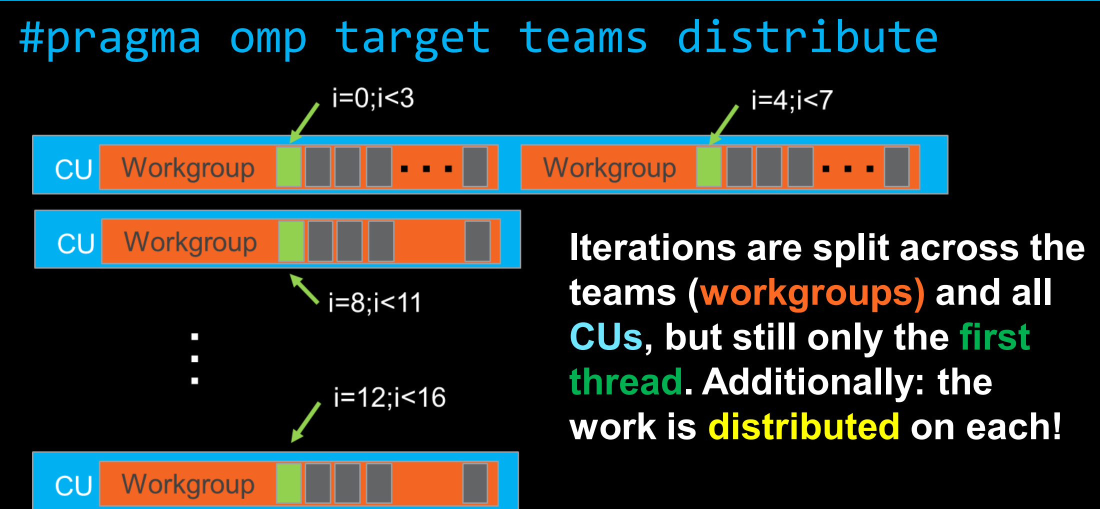


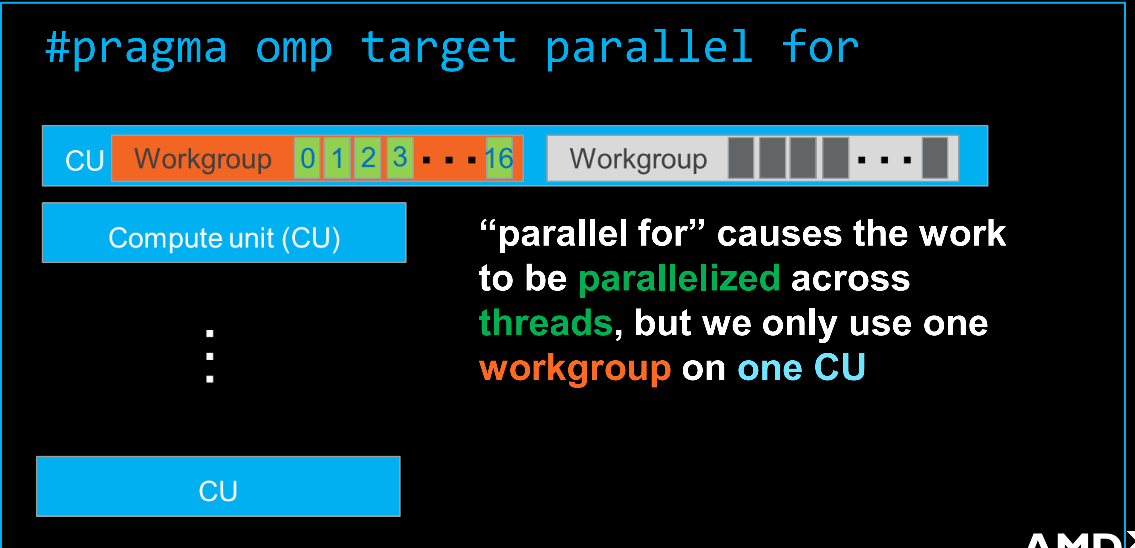
Iterations are split across the teams (workgroups) and all CUs, but still only the first thread. Additionally: the work is distributed on each!

Comparing subsets of GPU parallel compute directive







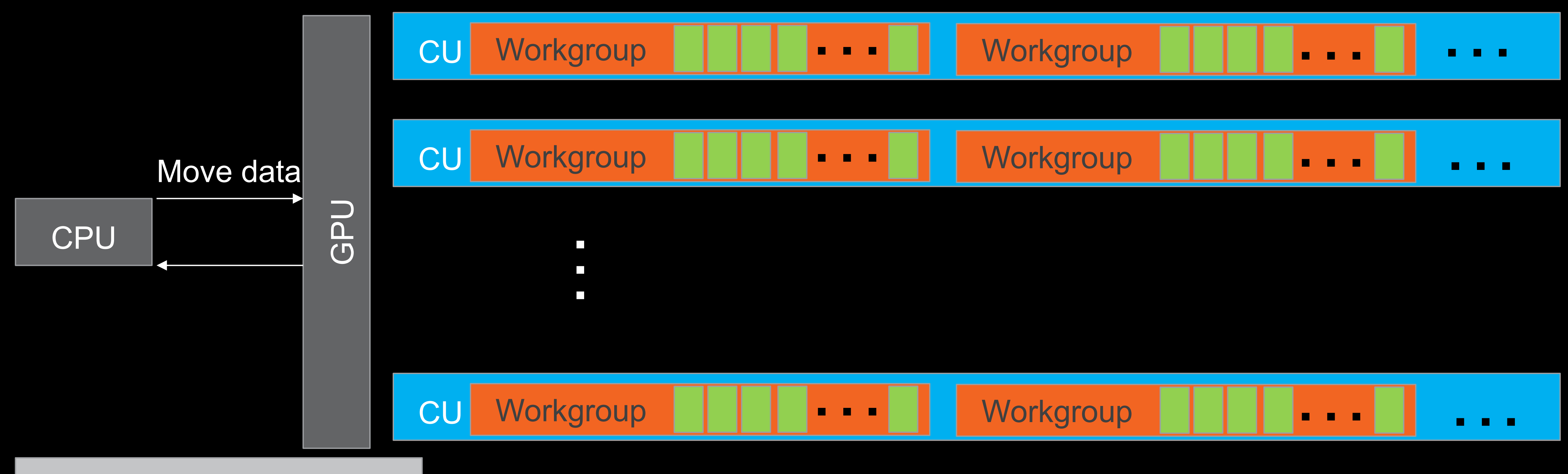


AMD
together we advance_

[Public]

Computation on the whole device

```
#pragma omp target teams distribute parallel do
```



Fully active device, all threads compute a different chunk of the loop!

Other compute clauses -- loop

- The loop clause was added as a simpler option that gives the compiler more freedom in implementing the parallelism for a for loop

```
#pragma omp target teams loop
```

  - Reduces the pragma complexity
  - Compilers may not optimize (or implement) this case as well because it is new.

- ROCm will generate an optimized target region implementation for AMD GPUs to use

```
#pragma omp target teams loop
```

Instead of

```
#pragma omp target teams distribute parallel for
```

OpenMP 6.0 is the future in most compilers, not yet implemented!

In OpenMP 6.0, replaces `teams distribute parallel for`

```
#pragma omp target loop
```


AMD together we advance_

[Public]

Example: saxpy

```
void saxpy(float a, float *x, float *y, int N) {
#pragma omp target teams loop
   for (int i = 0; i < N; i++) {
      y[i] += a * x[i];
   }

}
```

```
amdclang -fopenmp --offload-arch=$GPU_ARCH ...
```

[Public]

A note on the simd clause

- C/C++
  - `#pragma omp target teams distribute parallel for simd`
- Fortran
  - `!$omp target teams distribute parallel do simd`

- Note on the compute construct:
  - The `simd` clause doesn't do anything with the AMD OpenMP compiler and can be dropped.
  - Nvidia also doesn't use it.
  - Cray used to be the most common compiler where it was used, but they are moving to drop it as well.
  - But…there are still OpenMP compilers such as the Intel® compiler that recognize it and use it.
  - So…for maximum portability, we include it. But you may want to drop it depending on the systems you plan to run on.
  - If you use the `simd` clause, you will get warnings about not being able to vectorize during compilation. These are harmless and can be ignored.

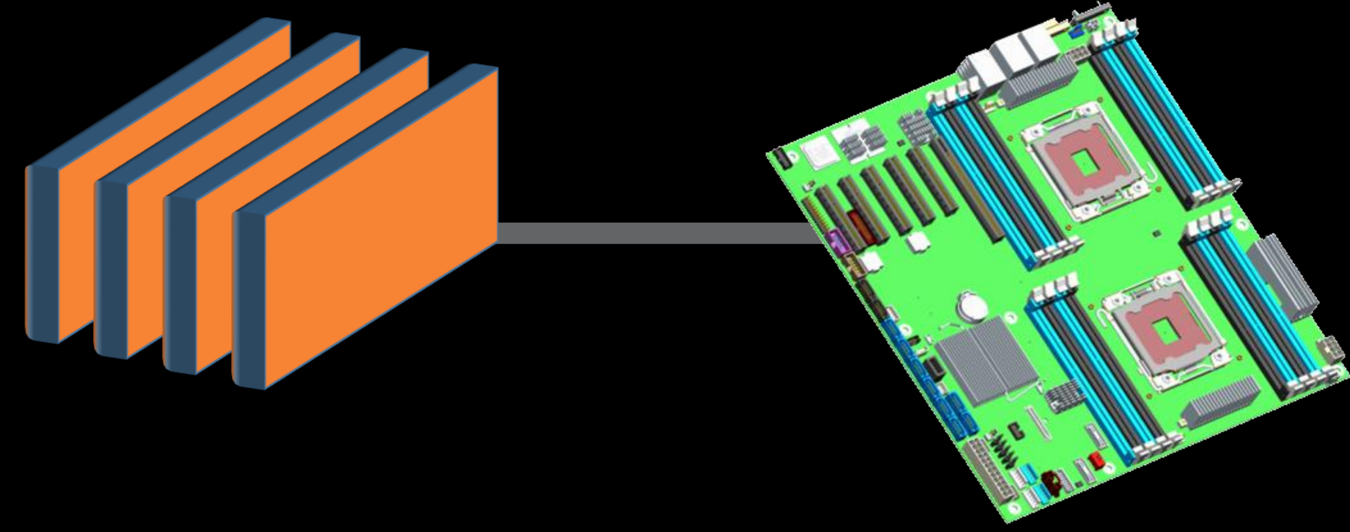
May still want to add for portability to discrete GPUs!

Not needed for APU, i.e. MI300A !

OpenMP® offload: Memory management for GPUs

[Public]

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

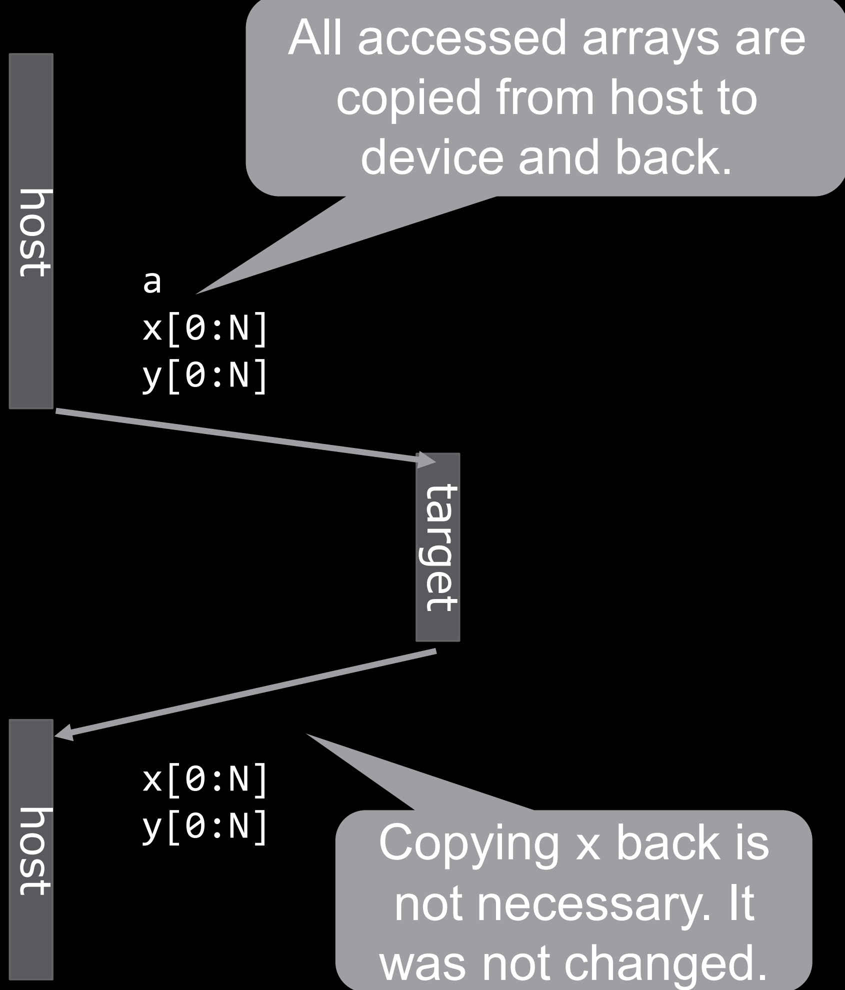
- Unnecessary data transfers must be avoided, by
  - only transferring what is actually needed for the computation
  - making the lifetime of the data on the target device as long as possible.

Example: saxpy_gpu_singleunit_static

Compiler does the work

```
int main(int argc, char *argv[]){
   int N=100000;
   float a=2.0f;
   float x[N], y[N];

   for (int i = 0; i < N; i++) {
      x[i] = 1.0f; y[i] = 2.0f;
   }

#pragma omp target teams distribute parallel for simd
   for (int i = 0; i < N; i++) {
      y[i] += a * x[i];
   }

  }
```

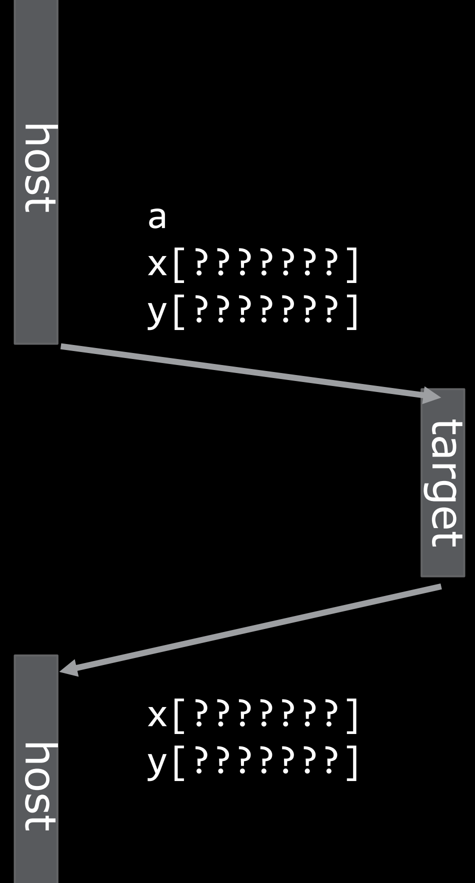
The compiler identifies variables that are used in the target region



Example: saxpy_gpu_singleunit_dynamic (C/C++)

```
int main(int argc, char *argv[]){
   int N=10000000;
   float a=2.0f;

   float *x = (float *)malloc(N*sizeof(float));
   float *y = (float *)malloc(N*sizeof(float));

   for (int i = 0; i < N; i++) {
      x[i] = 1.0f; y[i] = 2.0f;
   }

#pragma omp target teams distribute parallel for simd
   for (int i = 0; i < N; i++) {
      y[i] += a * x[i];
   }

   free(x); free(y);
}
```

The compiler identifies variables that are used in the target region but cannot get the sizes



In Fortran:
allocatable arrays are mapped by default
assumed size arrays are NOT mapped by default

AMD
together we advance_

Example: saxpy_gpu_singleunit_dynamic (C/C++)

```
int main(int argc, char *argv[]){
   int N=10000000;
   float a=2.0f;

   float *x = (float *)malloc(N*sizeof(float));
   float *y = (float *)malloc(N*sizeof(float));

   for (int i = 0; i < N; i++) {
      x[i] = 1.0f; y[i] = 2.0f;
   }

#pragma omp target teams distribute parallel for simd
   for (int i = 0; i < N; i++) {
      y[i] += a * x[i];
   }

   free(x); free(y);
}
```

The compiler identifies variables that are used in the target region but cannot get the sizes

Requires "export HSA_XNACK=1" or map clause

host

a
x[???????]
y[???????]

target

host

x[???????]
y[???????]

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int n) {

#pragma omp target teams \
 distribute parallel for map(to:x[0:n]) \
                         map(tofrom:y[0:n])
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

The compiler cannot determine the size of memory behind the pointer.

host

a
x[0:n]
y[0:n]

target

y[0:n]

host

Programmers have to help the compiler with the amount of data to transfer (C/C++, assumed size arrays Fortran).

Summary: OpenMP® on discrete GPU and APU

[Public]

GPU and APU Programming model with OpenMP

CPU CODE

```
!allocation on host
ALLOCATE(var(1:N))

!compute on host
!$omp parallel do &
!$omp  private(i), shared(var)
DO i=1,N
   var(i) = …
END DO
!$omp end parallel do
!sync barrier at omp end …
…
!deallocation
DEALLOCATE(var)
```

DISCRETE GPU CODE

```
!allocation on host
ALLOCATE(var(1:N))

!compute on device, expl. mem movement!
!$omp target teams distribute parallel do &
!$omp  map(tofrom:var) private(i),shared(var)
 DO i=1,N
   var(i) = …
 END DO
!$omp end target teams distribute parallel do
!host-device sync barrier at omp end …
…
!deallocation
DEALLOCATE(var)
```

Costly to switch CPU -> GPU!

APU CODE

```
!$omp requires unified_shared_memory
!allocation of unified m…
ALLOCATE(var(1:N))

!compute on device, no expl. mem movement!
!$omp target teams distribute parallel do &
!$omp                private(i),shared(var)
 DO i=1,N
   var(i) = …
 END DO
!$omp end target teams distribute parallel do
!host-device sync barrier at omp end …

!deallocation of unified memory
DEALLOCATE(var)
```

Cheap CPU -> GPU copy with APU

- Compute kernel
- Special directive to enable unified memory -> APU programming model only!
  Enforce with flag –fopenmp-force-usm
- Explicit memory management between CPU & GPU -> not needed for APU!
- Synchronization Barrier

[Public]

Shift the data and computation to the device

```
#pragma omp target teams distribute parallel do
```

Move data

CPU

GPU

map(to:…)
map(from:…)

CU Workgroup . . . Workgroup . . . . . .

CU Workgroup . . . Workgroup . . . . . .

CU Workgroup . . . Workgroup . . . . . .

Or APU: no data movement necessary

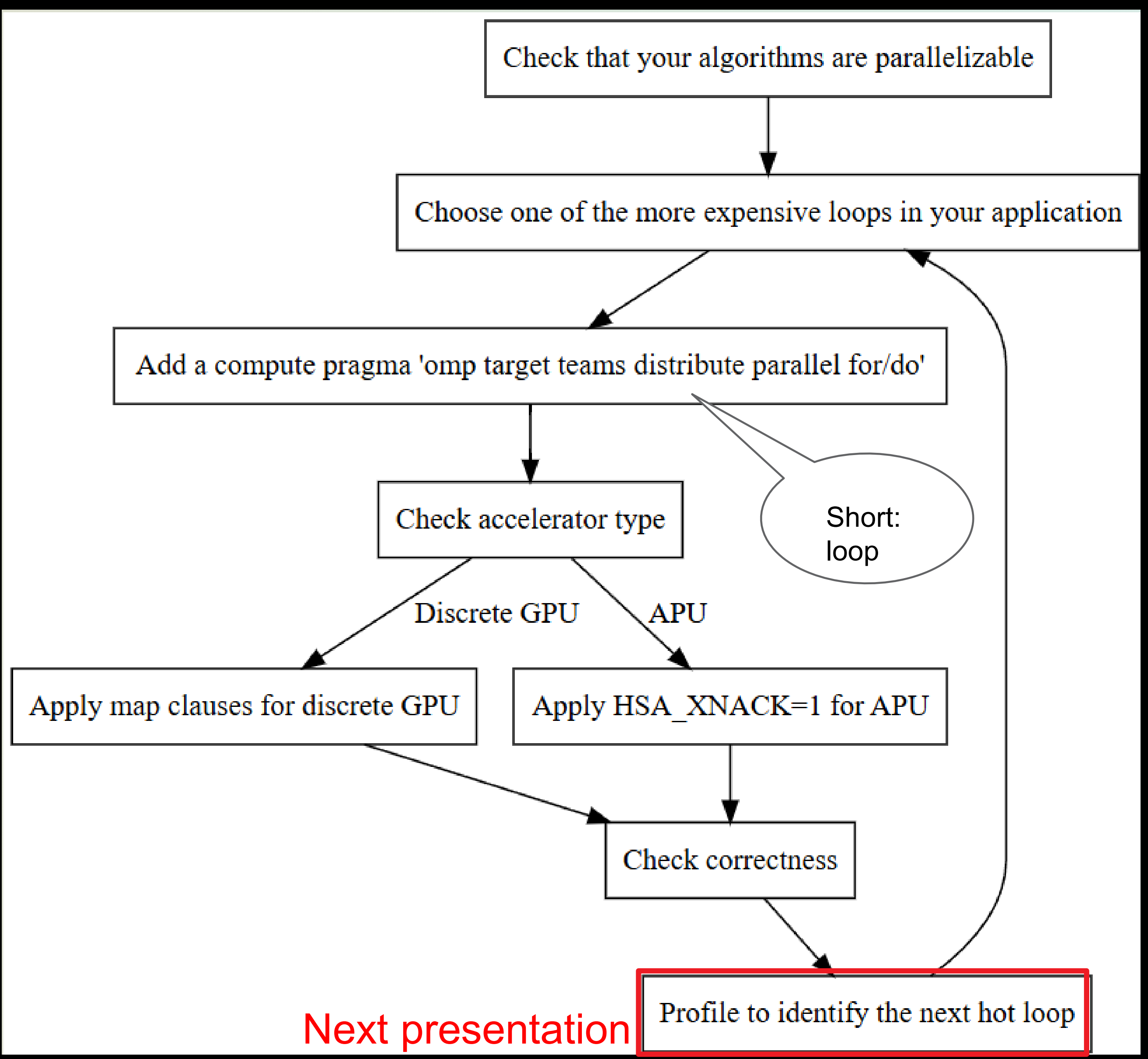
```
!$omp requires unified_shared_memory or compiler flag -fopenmp-force-usm
Run with: export HSA_XNACK=1
```

[Public]

How we begin porting

General recommendations:

- Select a representative test case
  - Identify “hot loops” relevant for your use case
- Start porting with simple loops to gain experience
- Prepare validation before you port
  - Set up a suite of validation tests
  - Unit tests and tests of the full application
  - What are expected what unexpected differences?
- More porting often more speedup than intricate optimizations (esp. with a flat profile)



Exercises Introduction to OpenMP Offload

https://github.com/amd/HPCTrainingExamples/tree/main/Pragma_Examples

C/C++ exercises (start with the C exercises, they are the simple ones, C++ more advanced):

Pragma_Examples/OpenMP/C/

Pragma_Examples/OpenMP/CXX/

Top-level README to set up environment

Further READMEs in sub-directories with guided exercises with and without HSA_XNACK=0 or 1 on MI300A

Fortran exercises:

Pragma_Examples/OpenMP/Fortran/

Top-level README to set up environment

Further READMEs in sub-directories with guided exercises with and without HSA_XNACK=0 or 1 on MI300A

Recommended exercises

Do the exercises in sub-directories:

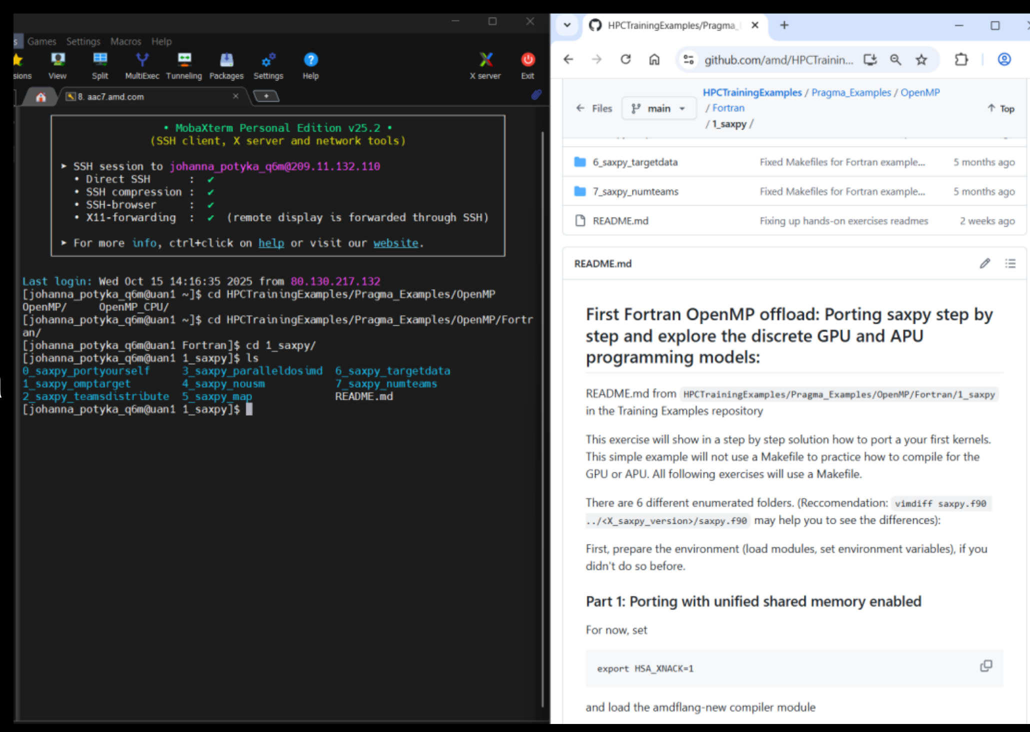
1_saxpy

2_vecadd

3_reduction

…

Each subdirectory contains a README to follow, best view it in a browser and put it side by side with the terminal.



Subdirectory
cd 1_saxpy/0_saxpy_portyourself

Contains a CPU version to start. If you are stuck there are sample solutions for different steps in the other subdirectories and the README to follow.

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.

© 2025 Advanced Micro Devices, Inc and OpenMP® Architecture Review Board. All rights reserved.

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Intel is a trademark of Intel Corporation or its subsidiaries
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.

AMD