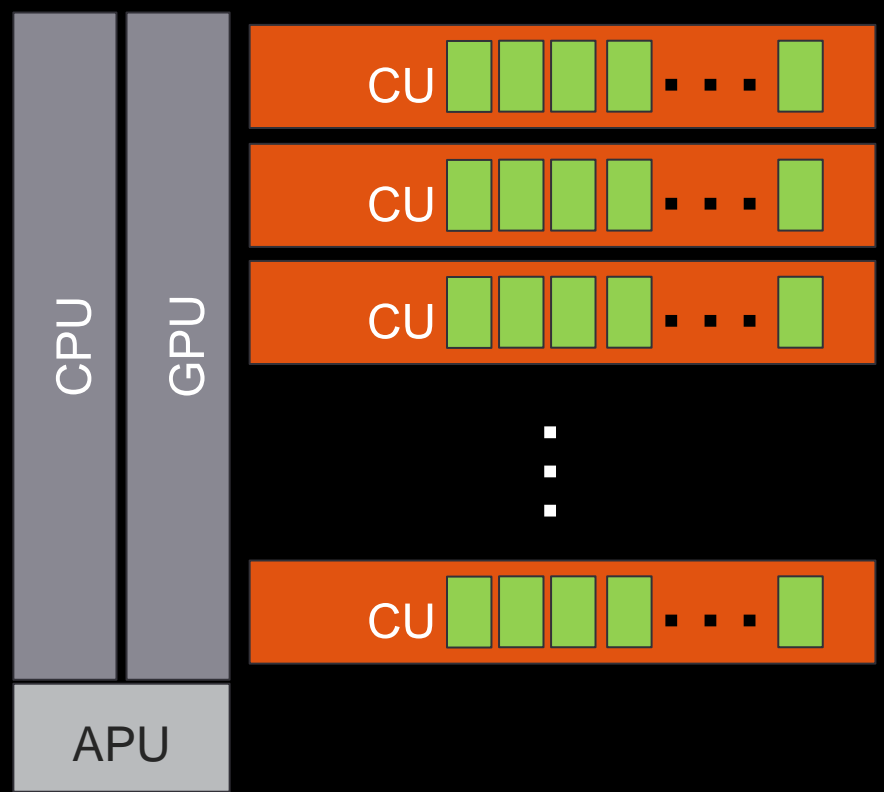




OpenMP® Offload Programming for APUs

Presenter: Johanna Potyka

Parallelism required for GPUs



	AMD Instinct™ MI300A
# Active CU / XCD	38
CDNA™ 3 Accelerated Compute Dies (XCD)	6
Stream Processors	$38 * 6 * 64 = 14,592$
L1 Cache / CU	32 KB
L2 Cache Shared Between CUs	4 MB
LLC Cache Shared Across XCDs	256 MB (also shared with CCDs on MI300A)

Long loops with independent data are good candidates for computation on the GPU!

Running Example for this Presentation: saxpy

Don't do this at home!
Use a BLAS library for this!

```
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 is the code we want to execute on
a target device (i.e., GPU)

```
SUBROUTINE saxpy(n, a, x, y)  
    IMPLICIT NONE  
    integer, intent(in) :: n  
    real(kind=real32), intent(in) :: a  
    real(kind=real32), allocatable, dimension(:), intent(in) :: x  
    real(kind=real32), allocatable, dimension(:), intent(inout) :: y  
    do i=1,n !not exactly the same range of iterations as in C example but typical in Fortran  
        y(i) = a * x(i) + y(i)  
    end do  
END SUBROUTINE saxpy
```

Recap for those who know OpenMP or crash course for those who do not!

Recap: OpenMP® loop parallelization for CPUs

Example: saxpy (serial version)

```
void saxpy(int n, float a, float *x, float *y) {  
  
    for (int i = 0; i < n; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
  
}
```

Assume: n is large such that computation in parallel is worth it!

Independent iterations!

Recap: OpenMP® on CPUs

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

“this line is OpenMP”

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

Recap: OpenMP® on CPUs

```
void saxpy(int n, double a, double *x, float *y) {  
    double sum = 0.0;  
    for (int i = 0; i < n; i++) {  
        sum += a * x[i] + y[i];  
    }  
}
```

Start a parallel region
“Use OMP_NUM_THREADS (env var)
number of OpenMP threads to
execute the following code in parallel”

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

Recap: OpenMP® on CPUs

```
void saxpy(int n, float a, float *x, float *y) {  
  
#pragma omp parallel for schedule(static) private(i) shared(x,y,a,n)  
for (int i = 0; i < n; i++) {  
    v[i] = a * x[i] + v[i];  
}
```

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

Recap: OpenMP® on CPUs

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

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

First thread gets the first $n/\text{OMP_NUM_THREADS}$ iterations, second thread the second etc.

There are also other schedules how to distribute the work.

Recap: OpenMP® on CPUs

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

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

Tell the compiler which variables are shared among OpenMP threads and which have private copies. There are defaults so this is optional if the defaults are good already.

Recap: OpenMP® on CPUs

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

Start a parallel region
"Use OMP_NUM_THREADS number of OpenMP threads to execute the following code"

"this line is OpenMP"

Tell the compiler which variables are shared among OpenMP threads and which have private copies. There are defaults so this is optional if the defaults are good already.

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

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

First thread gets the first n/OMP_NUM_THREADS iterations, second thread the second etc.

There are also other schedules how to distribute the work.



Recap: OpenMP® on CPUs (Fortran)

“do” in Fortran

“this line is OpenMP”

```
subroutine saxp(a, x, y, n)  
  !$omp parallel do schedule(static) private(i) shared(x,y,a,n)  
  do i=1,n  
    y(i) = a * x(i) + y(i)  
  end do  
  !$omp end parallel do  
end subroutine
```

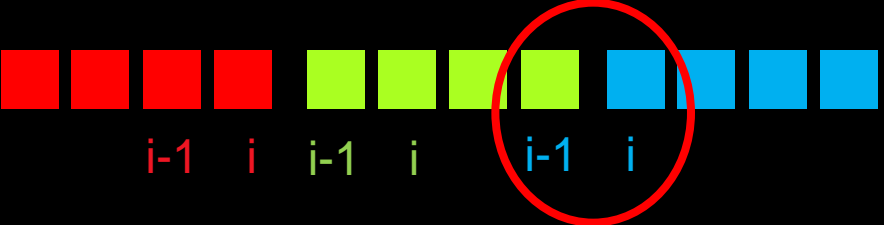
Mark the end in Fortran
(optional if clear where the end is)

Recap: OpenMP® on CPUs Shared memory & race conditions

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

Array is shared among threads!

➤ Race condition due to distribution of iterations among threads!



Array is shared among threads!

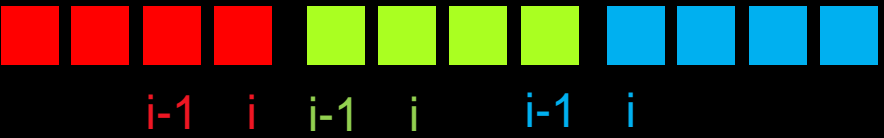
Recap: OpenMP® on CPUs Race Condition

```

subroutine saxpy(a, x, y, n)
  ! Declarations omitted
  !$omp parallel do schedule(static) private(i) shared(x,y,a,n)
  do i=1,n
    y(i) = a * x(i) + y(i-1)
  end do
  !$omp end parallel do
end subroutine

```

- Race condition due to distribution of iterations among threads!



Check if your algorithm is parallelizable!



May get even worse with a dynamic schedule

Recap: OpenMP® on CPUs: Reduction

Data sharing
attribute of
"asum"?

```
subroutine saxpy(a, x, y, n)
  ! Declarations omitted
  asum= 0.0_real64
  !$omp parallel do schedule(static) private(i) shared(a,n)
  do i=1,n
    asum = asum + a(i)
  end do
  !$omp end parallel do
end subroutine
```

Recap: OpenMP® on CPUs: Reduction

Needed: sum up per thread,
then sum up the values of all
threads

```
subroutine saxpy(a, x, y, n)
  ! Declarations omitted
  asum= 0.0_real64
  !$omp parallel do schedule(static) private(i) shared(a,n)
  do i=1,n
    asum = asum + a(i)
  end do
  !$omp end parallel do
end subroutine
```

Recap: OpenMP on CPUs: Reduction

reduction among all threads
+: add
min: compare minimum
max: compare maximum
or: compare or
and: compare and

```
subroutine saxpy(a, x, y, n)
  ! Declarations omitted
  sum= 0.0_real64
  !$omp parallel do schedule(static) private(i) shared(a,n) reduction(+:sum)
  do i=1,n
    sum = sum + a(i)
  end do
  !$omp end parallel do
end subroutine
```

- ✓ OpenMP parallelization of loops for the CPU
- How does this work on the GPU?
 - Compilation slightly different
 - New OpenMP directives

Compile OpenMP® code on GPUs

How to compile with different compilers for the GPU?

e.g. on Hunter

- HPE CCE

- Uses modules

module load PrgEnv-cray / PrgEnv-amd -> module for the cray environment (see HPE presentation for all options)

module load cce / amd -> enables of compilers through compiler wrappers CC and ftn

module load craype-accel-amd-gfx942 -> module to offload to gfx942 (use rocminfo to find gfx... architecture)

module load craype-x86-genoa -> module for CPU part of MI300A

module load rocm -> the compilers require rocm for offload

Compile with

CC -fopenmp -O2

ftn -fopenmp -O3

-> usual optimization flags -O2 or -O3, ...

- AMD Next Gen Fortran Compiler (under development)

- module load amdflang-new
 - amdflang **-fopenmp --offload-arch=gfx942**
 - Usual optimization options: -O2, or -O3,...

Note: Hunter currently does not yet have the AMD Next Generation Fortran compiler, try it on AAC6, it is also installed on other European HPC systems

Compile OpenMP GPU code for CPUs (functionality check)

- Remove the `--offload-arch=gfx942` and use the CPU architecture (`x86_64`) as offload architecture
 - flags are dependent on the compiler and environment you work with
 - `amdflang: --offload-arch=x86_64`
 - With the cray compilers: `module load craype-accel-host`

- No duplication of code required to run on the CPU

Greatly facilitates development, testing, portability and maintenance of codes!

- Compilers supporting OpenMP 4.0 or greater needed

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

- AMD Next Gen Fortran compiler is publicly available in a beta version which is frequently updated:

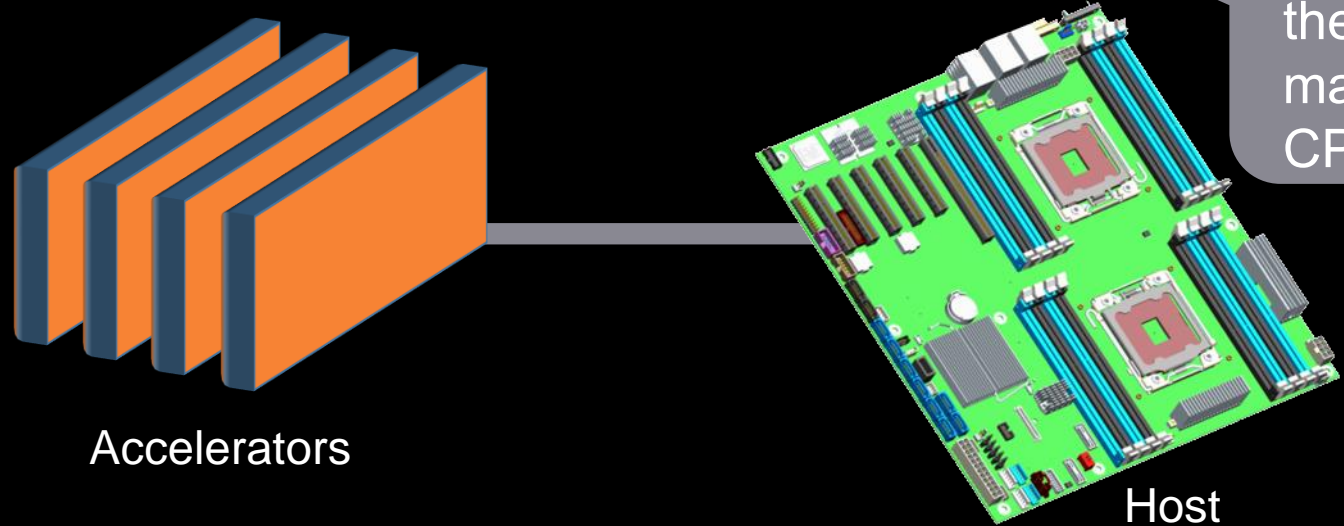
<https://rocm.blogs.amd.com/ecosystems-and-partners/fortran-journey/README.html>

<https://github.com/amd/InfinityHub-CI/blob/main/fortran/README.md>

OpenMP® Offload Basics

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
 - be able to receive data from the host and send data back and
 - perform computation upon request



A device can also be the CPU itself! This makes code portable CPU<-> GPU

Special directive for the APU: unified_shared_memory

- To use the APU programming model add

C/C++:

```
#pragma omp requires unified_shared_memory
```

Or set the compiler flag
-fopenmp-force-usm

Fortran:

```
!$omp requires unified_shared_memory
```

at the beginning of each file/module (in Fortran: after implicit none, C: after the includes).

To run code compiled with unified_shared_memory:

```
export HSA_XNACK=1
```

Otherwise you get and error!

- Should be default on Hunter
- NOT default on AAC6
- If you also run on other systems: check with rocminfo that gfx942:xnack+ is set!

Compute on the GPU

“Compute this on the GPU”

```
void saxpy(int n, float a, float *x, float *y) {  
  // [...some CPU code...]  
  
  #pragma omp target  
  for (int i = 0; i < n; i++) {  
    y[i] = a * x[i] + y[i];  
  }  
  
  // [...more CPU code...]  
}
```

host

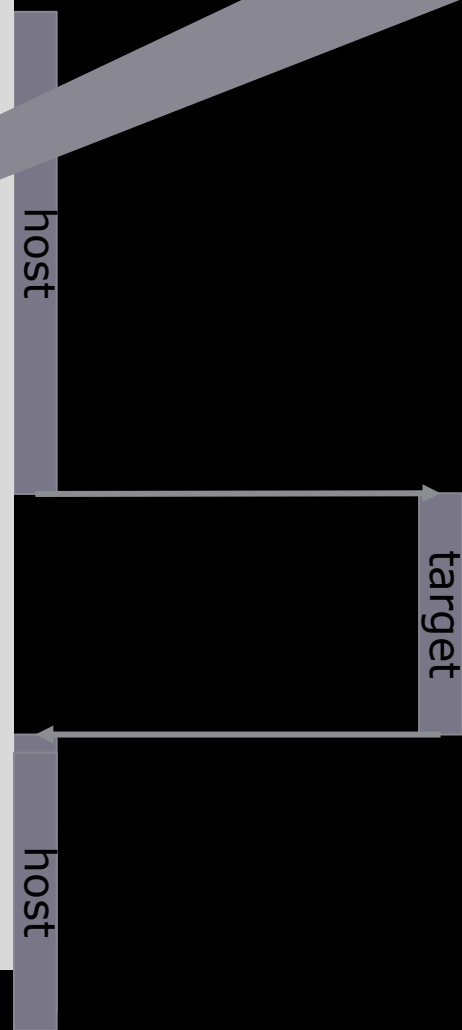
target

host

Compute on the GPU (Fortran)

“Compute this on the GPU”

```
subroutine saxpy(n, a, x, y) {  
  integer :: n  
  real(kind=real32) :: a  
  real(kind=real32), dimension(N) :: x, y  
  real(kind=real64) :: t, tb, te  
  integer :: I  
  
  ![...some CPU code...]  
  !$omp target  
  do i = 1, n  
    y(i) = a * x(i) + y(i)  
  end do  
  !$omp end target  
  ![...some CPU code...]  
  
end subroutine saxpy
```



OpenMP® Device Constructs

- Transfer control (and data) from the host to the device

- Syntax (C/C++)

```
#pragma omp target [clause[[, clause],...]  
structured-block
```

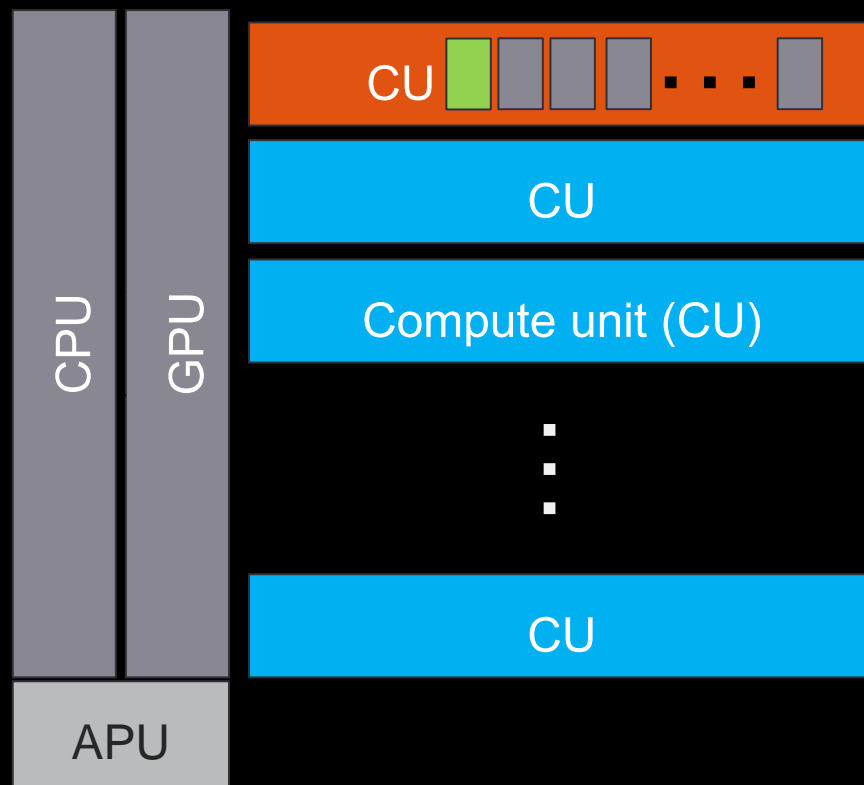
- Syntax (Fortran)

```
!$omp target [clause[[, clause],...]  
structured-block  
!$omp end target
```

- Clauses

```
device(scalar-integer-expression)  
if(scalar-expr)
```

We know how to shift the computation to the device,...



...but we only use a tiny fraction of the hardware: only a single thread runs at a time!

More parallelism on the GPU

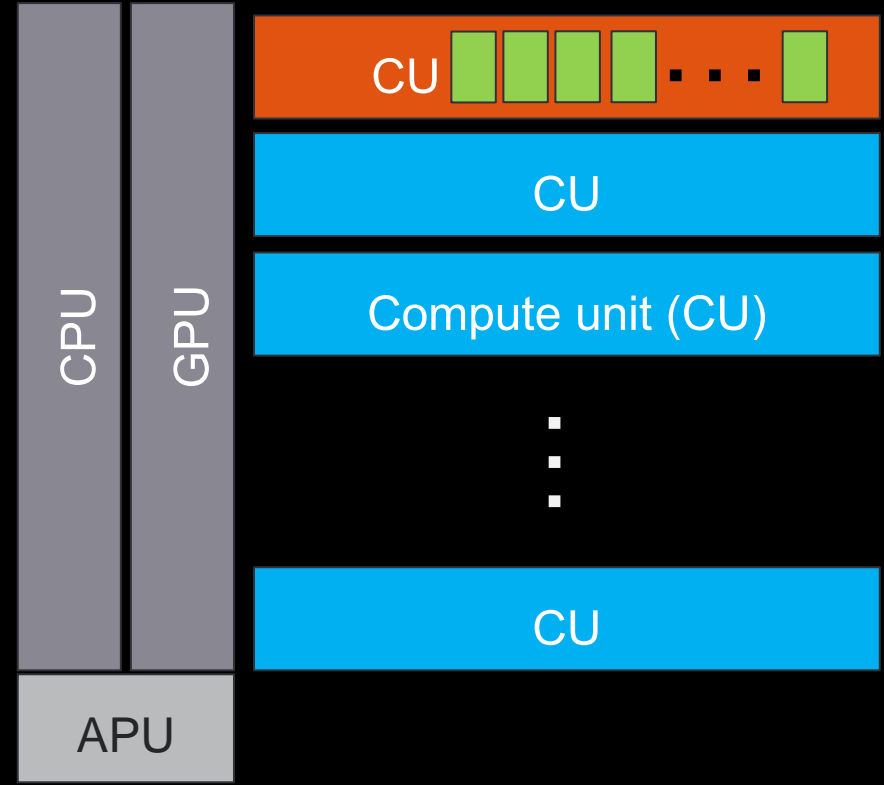
```
void saxpy(float a, float* x, float* y,
           int n) {
    #pragma omp target
    #pragma omp parallel for simd
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```



AMD compilers for GPU ignore simd directives.

Create a team of threads to execute the loop in parallel using SIMD instructions.

Better use of the hardware with “parallel for (simd)”,...



**...but still not much better:
Only use one CU instead of
the whole GPU!**

Multi-level Parallel saxpy

- Manual code transformation
 - Tile the loop into an outer loop and an inner loop.
 - Assign the outer loop to “teams” (OpenCL™: work groups; HIP: blocks).
 - Assign the inner loop to the “threads” (OpenCL™: work items; HIP: threads).

```
void saxpy(float a, float* x, float* y, int n) {
    #pragma omp target teams
    {
        int bs = n / omp_get_num_teams(); // could also use nCUs
        #pragma omp distribute
        for (int i = 0; i < n; i += bs) {
            #pragma omp parallel for
            for (int ii = i; ii < i + bs; ii++) {
                y[ii] = a * x[ii] + y[ii];
            }
        }
    }
}
```

teams Construct

- Support multi-level parallel devices
- Syntax (C/C++):

```
#pragma omp teams [clause[[,] clause],...]  
structured-block
```
- Syntax (Fortran):

```
!$omp teams [clause[[,] clause],...]  
structured-block  
!$omp end teams
```
- Clauses

```
num_teams(integer-expression), thread_limit(integer-expression)  
default(shared | firstprivate | private | none)  
private(list), firstprivate(list), shared(list), reduction(operator:list)
```

Distribute Construct

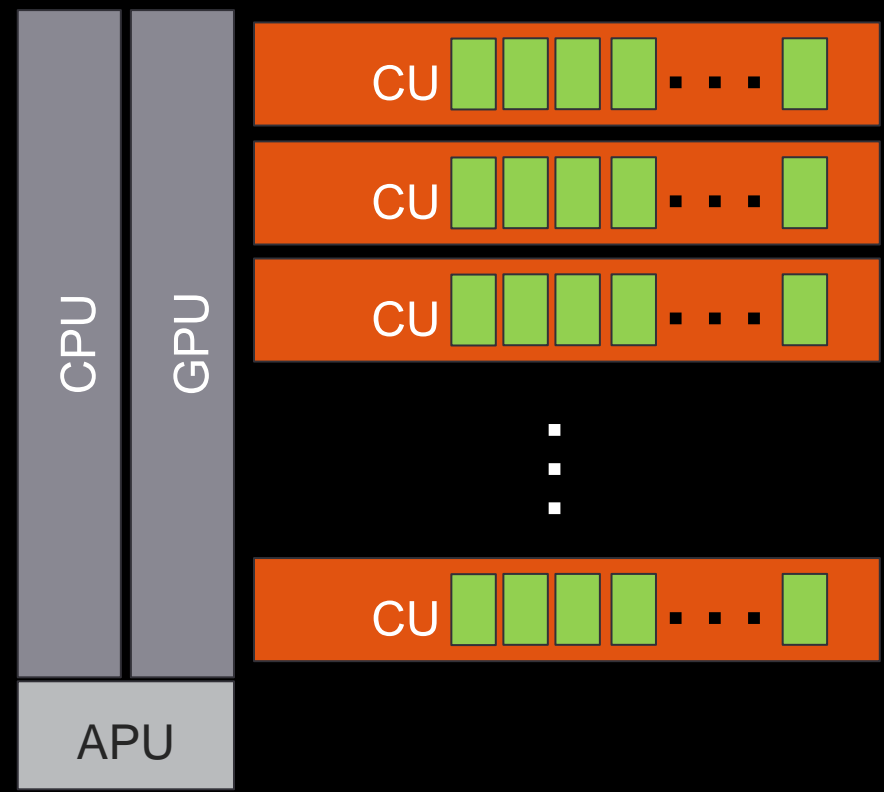
- Worksharing construct w/o implicit (and explicit) barrier
- Syntax (C/C++):

```
#pragma omp distribute [clause[[,] clause],...]  
Loop-nest
```
- Syntax (Fortran):

```
!$omp distribute [clause[[,] clause],...]  
Loop-nest  
[!$omp end distribute]
```
- Clauses

```
collapse(integer-constant)  
private(list), firstprivate(list), shared(list), last_private(list)
```

Better use of the hardware with “target” + “teams distribute” + “parallel for/do (simd)”



...full device used!

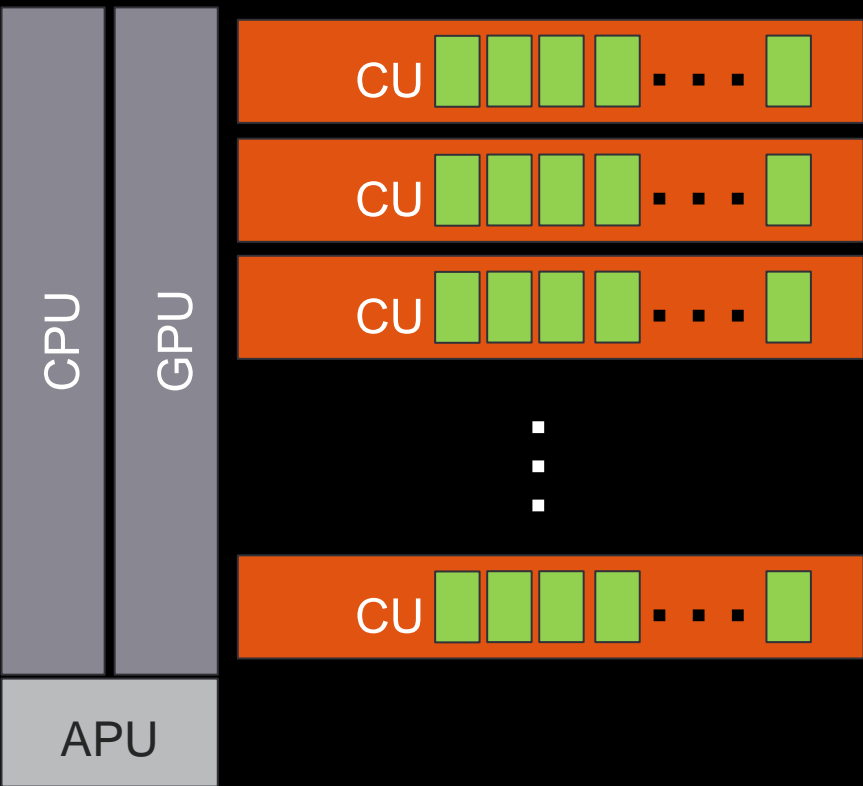
Multi-level Parallelism: Simplification

- For convenience, the OpenMP® languages defines composite constructs to implement the required code transformations.

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

```
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 target teams distribute parallel do  
end subroutine
```

Why does it not work to simply use “parallel” for the whole device?



- Some features allowed with “parallel” require synchronization among threads according to the OpenMP® standard
 - This is possible inside a CU
 - This is not (easily) possible among CUs!
- “teams distribute” is suitable to parallelize across multiple CUs!

OpenMP® Offload Porting code from CPU to GPU

Recap: Original CPU code

```
void something(...) {  
    #pragma omp parallel for  
    for (int i = 0; i < n; i++) {  
        ...  
    }  
}
```

```
SUBROUTINE something(...)  
    IMPLICIT NONE  
    ...  
    !$omp parallel do  
    do i=1,n  
        ...  
    end do  
END SUBROUTINE something
```

Minimal changes required to run on the APU

```
void something(...) {  
    #pragma omp target teams distribute parallel for  
    for (int i = 0; i < n; i++) {  
        ...  
    }  
}
```

Offloading is a “find and replace” job for large portions of CPU OpenMP code!

```
SUBROUTINE something(...)  
    IMPLICIT NONE  
    ...  
    !$omp target teams distribute parallel do  
    do i=1,n  
        ...  
    end do  
END SUBROUTINE something
```

Loop directive

Less work when you don't have an existing OpenMP implementation

- For convenience, the OpenMP® languages defines composite constructs to implement the required code transformations.

```
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] + y[i];  
}  
}
```

Works with the cray compiler, amdclang compiler, and the AMD Next Generation Fortran compiler

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

OpenMP® Porting: some hints

Data-sharing Defaults

Loop indices: private
Local scalars: firstprivate
Global / module wide variables: shared

Careful:

“module wide” includes SAVE variables in Fortran

e.g. `real(kind=rt) :: tmp = 0.0_rt` !this is a SAVE variable!

Arrays: shared

firstprivate: each thread gets a private copy of the variable and the value before entering OpenMP region is assigned to each
private: each thread has a private version of the variable, no copy of the value
shared: every thread can access the variable...

```
!$omp target teams distribute parallel do default(none) shared(...) private(...)
do i=...
...
end do
!$omp end do
```

May keep you from unintentional errors

```
#pragma omp target teams distribute parallel for default(none) shared(...) private(...)
for (int i = 0; i < n; i++) {
    ...
}
```

Nested loops: Not enough parallelism?

```
subroutine testroutine(a, x, y, n)
  ! Declarations omitted
  !$omp target teams distribute parallel do
  do k=1,nz
    do j=1,ny
      do i=1,nx
        y(i,j,k) = a * x(i,j,k) + y(i,j,k)
      end do
    end do
  end do
  !$omp end target teams distribute parallel do
end subroutine
```

Only the k loop is parallelized!
Often $nz \ll nCUs * 64 * \text{"a few"}$: very bad use of the GPU

Nested loops: Collapse clause

“collapse(3)” tells the compiler to parallelize all three nested loops:

$nz \cdot ny \cdot nx$ iterations to parallelize: often sufficient parallelism for the GPU

```
subroutine testroutine(a, x, y, n)
  ! Declarations omitted
  !$omp target teams distribute parallel do collapse(3)
  do k=1,nz
    do j=1,ny
      do i=1,nx
        y(i,j,k) = a * x(i,j,k) + y(i,j,k)
      end do
    end do
  end do
  !$omp end target teams distribute parallel do
end subroutine
```

reduction on GPU

Reduction possible in a GPU kernel

```
subroutine saxpy(a, x, y, n)
  ! Declarations omitted
  sum= 0.0_real64
  !$omp target teams distribute parallel do private(i) shared(a,n) reduction(+:sum)
  do i=1,n
    sum = sum + a(i)
  end do
  !$omp end target teams distribute parallel do
end subroutine
```

- There is also an OMP ATOMIC clause, but if it is not necessary, prefer reduction (if possible)
 - reduction allows the compiler to optimize better -> better performance

Calling functions on the Target Device (C/C++)

```
#include <stdlib.h>
#include <stdio.h>

#pragma omp requires unified_shared_memory

#pragma omp declare target
void compute(double *x){
    *x = 1.0;
}
#pragma omp end declare target

int main(){
    int N=1000;
    double *x = (double *)malloc(N*sizeof(double));

    #pragma omp target teams distribute parallel for simd
    for (int k = 0; k < N; k++){
        compute(&x[k]);
    }
    double sum = 0.0;
    #pragma omp target teams distribute parallel for simd reduction(+:sum)
    for (int k = 0; k < N; k++){
        sum += x[k];
    }
}
```

- Mark all functions that are called **inside a GPU kernel** with the pragma **declare target**.
- The OpenMP® compiler identifies the function and operator as needed to correctly generate code for the GPU
- This support **is not available** when the implementation of a function, called from within a target region, is not visible when compiling the file that contains the target region.
 - Use #include ... if the function is in another file

Calling functions on the Target Device (C/C++)

```
#include <stdlib.h>
#include <stdio.h>

#pragma omp requires unified_shared_memory

#pragma omp declare target device_type(nohost) link(compute)
void compute(double *x){
    *x = 1.0;
}

#pragma omp end declare target

int main(){
    int N=1000;
    double *x = (double *)malloc(N*sizeof(double));

    #pragma omp target teams distribute parallel for simd
    for (int k = 0; k < N; k++){
        compute(&x[k]);
    }
    double sum = 0.0;
    #pragma omp target teams distribute parallel for simd reduction(+:sum)
    for (int k = 0; k < N; k++){
        sum += x[k];
    }
}
```

Optional, if routine
not needed for CPU!

- Mark all functions that are called **inside a GPU kernel** with the pragma **declare target**.
- The OpenMP® compiler identifies the function and operator as needed to correctly generate code for the GPU
- This support is **not available** when the implementation of a function, called from within a target region, is not visible when compiling the file that contains the target region.
 - Use #include ... if the function is in another file

Calling Subroutines and functions on the Target Device (Fortran)

```

module computemod
public :: compute

contains
subroutine compute(x)
  !$omp declare target
  !example routine called from kernel
  integer,parameter :: rk=8
  real(kind=rk),intent(inout) :: x
  x = 1.0_rk
end subroutine compute
end module
program device_routine
  !if in other file
  use computemod, only: compute
  implicit none

  !declaration and initialisation omitted

  [...]
  !--- call a device subroutine in kernel
  !$omp target teams distribute parallel do simd
  do k=1,N
    call compute(x(k))
  end do
  !$omp end target teams distribute parallel do simd

  [...]
end program device_routine

```

Ends “automatically” at subroutine end, there is no end declare target in Fortran

The OpenMP® compiler identifies the function and operator as needed to correctly generate code for the GPU if it is in the same compilation unit

If in another compilation unit

- use <modulename>, only: <subroutine, function>
- Mark all functions that might be called **inside a GPU kernel** with the pragma **declare target**.

device_type(nohost) link(compute) also exists in Fortran to limit compilation of that routine to GPU only

Debug flags: Information about what runs on the GPU

HPE Cray compilers:

```
export CRAY_ACC_DEBUG=1,2,3
```

AMD compilers:

```
export LIBOMPTARGET_KERNEL_TRACE=1,...
```

```
export LIBOMPTARGET_INFO = -1 (all flags), 1, 2,... (increasing amount of debug information)
```

Example:

```
$ HSA_XNACK=1 LIBOMPTARGET_KERNEL_TRACE=1 ./ftn_axpy  
DEVID: 0 SGN:3 ConstWGSize:256 args: 6 teamsXthrds:( 1X 256) reqd:( 0X 0) lds_usage:0B  
sgpr_count:18 vgpr_count:3 sgpr_spill_count:0 vgpr_spill_count:0 tripcount:0 rpc:0 md:0 md_LB:-1  
md_UB:-1 Max Occupancy: 10 Achieved Occupancy: 2%  
n:__omp_offloading_2d_21dab__QMftn_axpy_implPaxpy_omp_gpu_172  
checksum 5.00 expected 5.00
```

Recap: Porting from CPU with unified_shared_memory

- Port long loops to GPU kernels by adding directives
- CPU and GPU access the same memory on MI300A with `omp requires unified_shared_memory`
This directive should be used for the APU or the **compiler flag -fopenmp-force-usm**
 - Run with: `export HSA_XNACK=1` (may or may not be default ...check with rocminfo for xnack+!)
- Control over hardware elements and computation parallelism
 - Select to compute on GPU – **target**
 - Parallelize computation – **teams distribute**
 - Parallelize computation – **parallel for/do**
 - A lot of parallelism required to use the whole device efficiently!
 - Nested loops: **collapse** clause may help!
- **reduction** usable on GPU
- Device routines: mark them with **declare target** if in other compilation unit
- Beware of defaults for data sharing in OpenMP®, consider **default(none)**

Summary

- AMD and Cray Fortran OpenMP[®] compilers can offload computation to AMD GPUs
 - Good support for C and C++ languages
 - Being used in production by many applications
 - Basic support for Fortran in rocm (Recommendation: right now use the AMD Next Gen Fortran compiler, it is the future of rocm)
 - AMD Next Gen Fortran compiler beta released (not yet on Hunter, try on AAC6; best option today for Hunter cce 18.0.1 cray ftn)
- Backed by an industry language standard
 - Managed by the OpenMP[®] Architecture Review Board
 - Portable across different vendors as well as CPU and GPU!
- Portability across GPU platforms for core OpenMP[®] constructs
 - Tightly integrates with OpenMP multi-threading on the host

How to ensure portability with performance for discrete GPUs: See next presentation!

Acknowledgements

- Michael Klemm
- Giacomo Rossi
- Bob Robey
- Giacomo Capodaglio

- More colleagues from HPC Solutions and Performance Analysis group in AMD

- Participants and Mentors of the past HLRS Trainings, Hunter preparation Hackathons and other Hackathons 😊

OpenMP® exercises:

git clone <https://github.com/amd/HPCTrainingExamples.git>

cd HPCTrainingExamples/tree/main/Pragma_Examples/OpenMP

C or Fortran versions available!

Read about the general setup:

There are general instructions in the C/Fortran top level READMEs for the Cray compiler and the Next Generation Fortran Compiler and different systems (aac6).

Recommendation: Read the README in the Browser online

Exercises for this presentation:

1_saxpy README Part 1 with USM

2_reduction README Part 1 with USM

5_device_routines

Start with the “portyourself” versions in each folder, look at the solutions only if you are stuck!

Note: We will do the target data and map exercises later.

They are not relevant for MI300A with xnack+, only for portability to discrete GPUs!

HPCTrainingExamples / Pragma_Examples / OpenMP / Fortran / 1_saxpy / ↑ Top

6_saxpy_targetdata	added guided Fortran excercises for porting on MI300A and exploring X...	4 months ago
7_saxpy_numteams	adjust num_teams	4 months ago
README.md	Update README.md	3 months ago

README.md ✎ ☰

First OpenMP offload: Porting saxpy step by step and explore the discrete GPU and APU programming models:

This excercise will show in a step by step solution how to port a your first kernels. This simple example will not use a Makefile to practice how to compile for the GPU or APU. All following excercises will use a Makefile.

There are 6 different enumerated folders. (Reccomendation: `vimdiff saxpy.f90 ../<X_saxpy_version>/saxpy.f90` may help you to see the differences):

First, prepare the environment (load modules, set environment variables), if you didn't do so before.

Part 1: Porting with unified shared memory enabled

For now, set

```
export HSA_XNACK=1
```

to make use of the APU programming model (unified memory). 0) the serial CPU code.

Exercises: clarification of usual confusions

- Choose Fortran or C:
 - https://github.com/amd/HPCTrainingExamples/tree/main/Pragma_Examples/OpenMP/Fortran
 - https://github.com/amd/HPCTrainingExamples/tree/main/Pragma_Examples/OpenMP/C
- Do each of the exercises with unified shared memory
 - There is always a folder with the CPU code to port yourself
 - A Readme which explains what to do (recommendation: open it in your favorite browser that it is displayed nicely)
 - Folders with solution steps
 - For USM (fitting this presentation)
 - With map clauses (next presentation, do those steps later)
 - The numbering order of the folders makes sense if you want to start from easy to more complex

Now:
Exercises with USM on the APU

Exercise solutions for portability to discrete GPUs explained in next presentation

Read Part 1 for USM

First Fortran OpenMP offload to the discrete GPU and APU

- If you completed saxpy Part 1 USM, feel free to continue with other exercises in the C / Fortran main folder

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.

OpenCL™ is a trademark of Apple Inc. used by permission by Khronos Group, Inc.

HPE is a registered trademark of Hewlett Packard Enterprise Company and/or its affiliates.

LLVM™ is a trademark of LLVM Foundation

AMD 