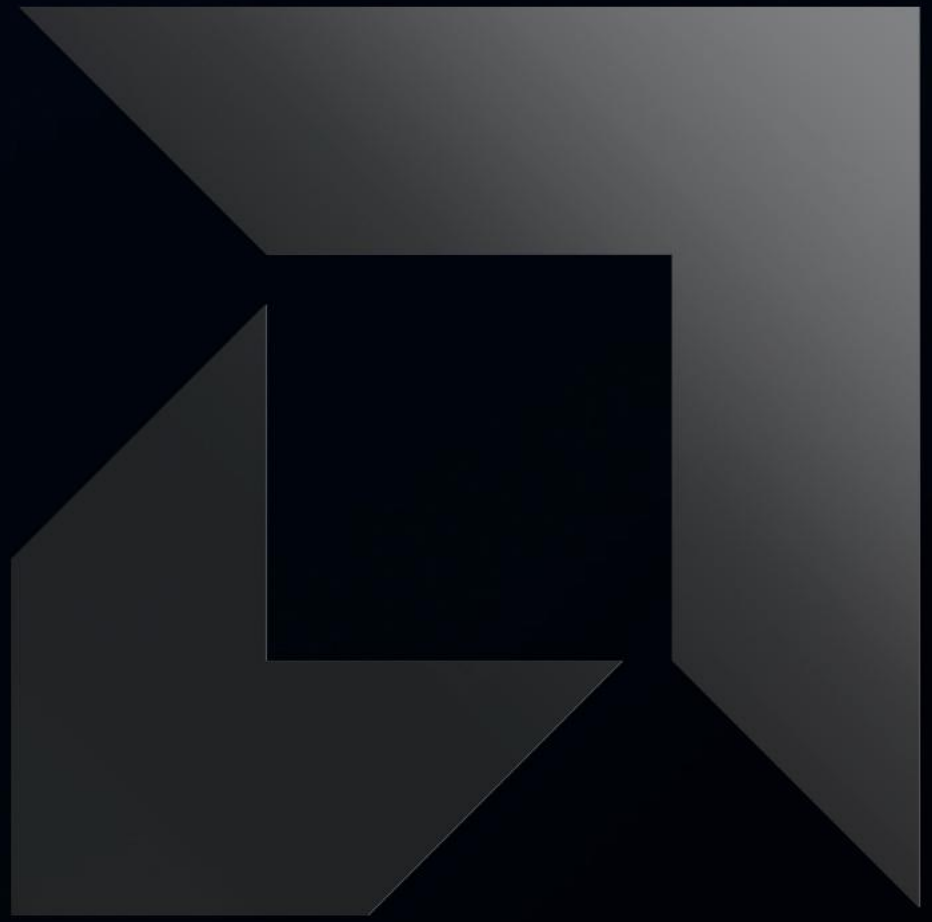


Real-World OpenMP® Language Constructs

Presenter:
AMD @ Tsukuba University
Oct 21-23, 2025

AMD
together we advance_

Complex compute directives

Breaking down the compute directive

Directive/Clause	Meaning
target	offloads the enclosed code to the device (GPU)
teams	creates a “league of teams” with the initial thread of each executing the code region with all of the data on each thread
distribute	loop iterations are distributed out across the teams and executed on the main thread
parallel	Create multiple threads (of a team)
for/do	spread out different portions of work over threads
simd	vectorization (use SIMD instructions), but not used by most compilers, including amdclang
loop	effectively replaces "distribute parallel for simd" In OpenMP 6.0, also "teams distribute parallel for simd"

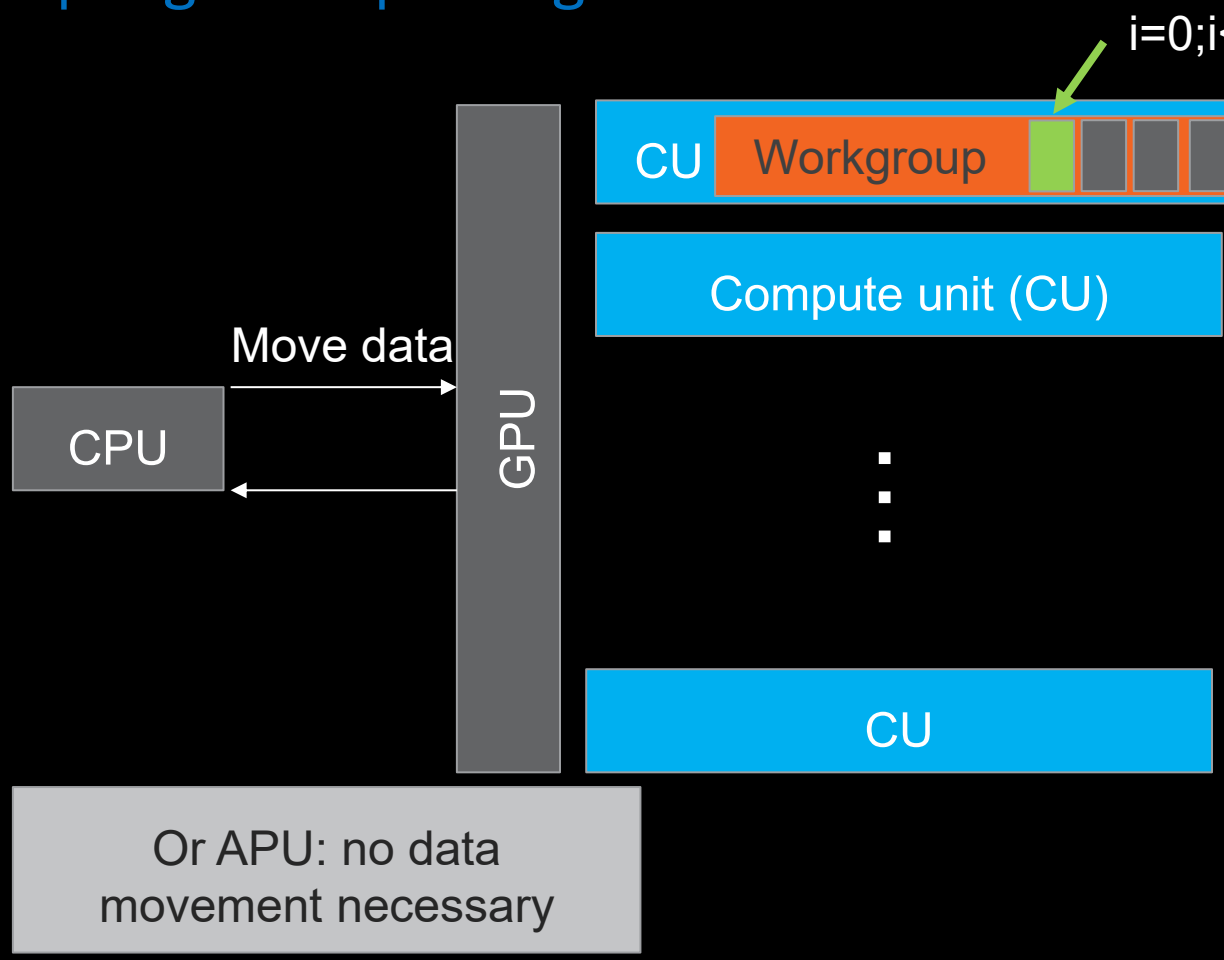
SIMD – Single Instruction Multiple Data is a term from Flynn’s Taxonomy that categorizes types of computer architectures. In this architecture, a single instruction is applied to multiple data. One of the examples of this type is a vector unit where one instruction is applied to multiple lanes. The GPU is very much like a wide vector unit and is also a SIMD architecture. It is often described as SIMT, Single Instruction Multiple Thread, a subcategory of SIMD with some important distinctions.

Breaking down the compute directive – C/C++ and Fortran

Directive/Clause	C/C++	Fortran
target	**#pragma omp target** {structured code block}	**!$omp target** <code block> **!$omp end target**
teams	**#pragma omp target teams** {structured code block}	**!$omp target teams** <code block> **!$omp end target teams**
distribute	**#pragma omp target teams distribute** {structured code block}	**!$omp target teams distribute** <code block> **!$omp end target teams distribute**
parallel for/do simd	**#pragma omp target teams distribute parallel for simd** {structured code block}	**!$omp target teams distribute parallel do simd** <code block> **!$omp end target teams distribute parallel do simd**
loop	**#pragma omp target teams loop** {structured code block}	**!$omp target teams loop** <code block> **!$omp end target teams loop**
loop (OpenMP 6.0 standard alternate)	**#pragma omp target loop*** {structured code block} * both forms valid	**!$omp target loop** <code block> **!$omp end target loop**

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

```
#pragma omp target
```

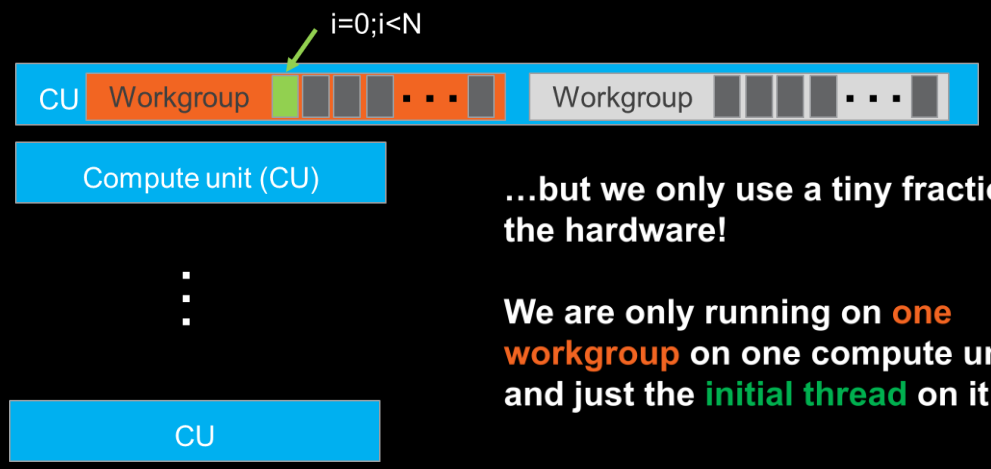


…but we only use a tiny fraction of the hardware!

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

Comparing subsets of GPU parallel compute directive

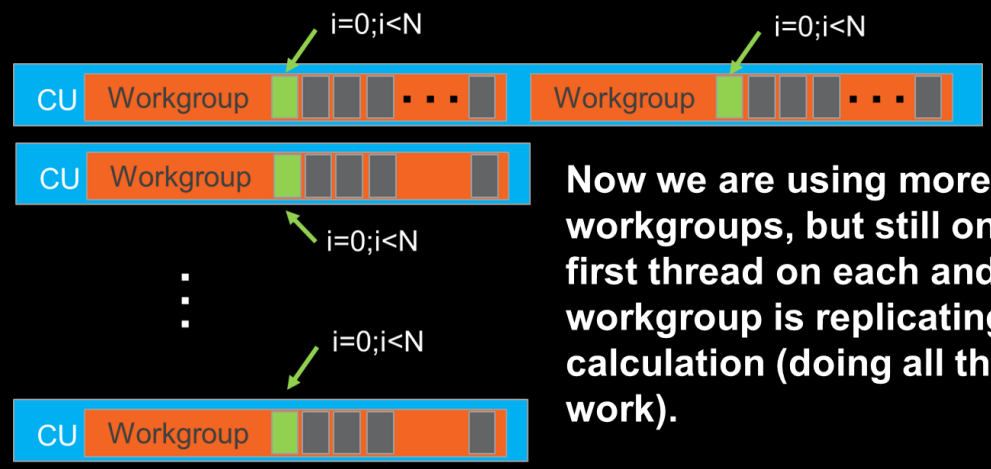
#pragma omp target



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

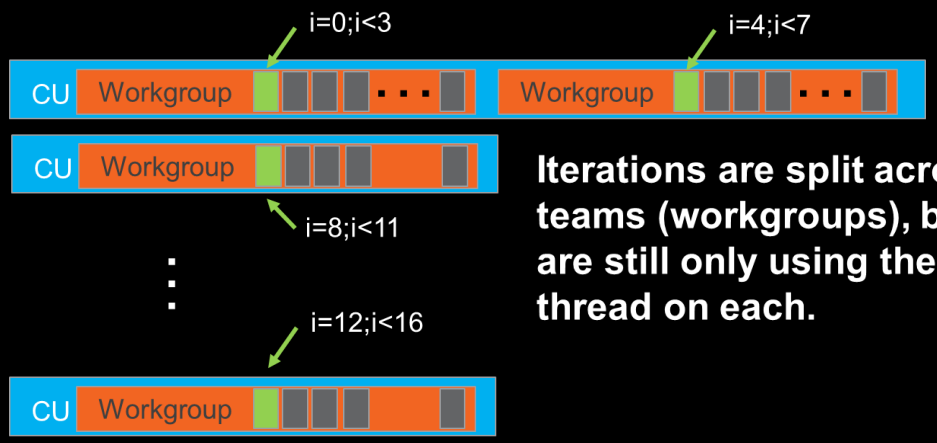
We are only running on one workgroup on one compute unit and just the initial thread on it.

#pragma omp target teams



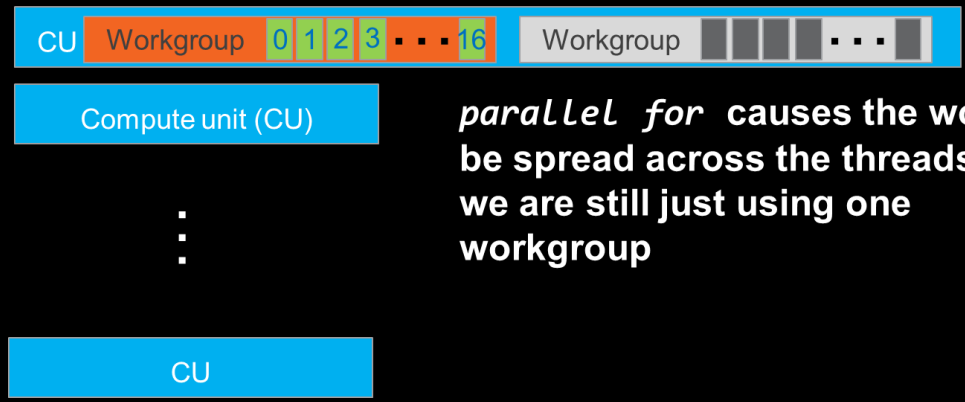
Now we are using more workgroups, but still only the first thread on each and each workgroup is replicating the calculation (doing all the work).

#pragma omp target teams distribute



Iterations are split across the teams (workgroups), but we are still only using the first thread on each.

#pragma omp target parallel for



parallel for causes the work to be spread across the threads but we are still just using one workgroup

The OpenMP® API is directed towards a generic device

- Interpretation varies a little on how it applies to a GPU
- Compiler implementations may vary in their implementation and support
- Some combinations might not make sense
  - Distribute without teams? – generates error with amdclang

Exploring subsets of the full compute pragma

Replacing the pragma with subsets of the full pragma
See each of the following slides

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

   printf("check output:\n");
   printf("y[0] %lf\n",y[0]);
   printf("y[N-1] %lf\n",y[N-1]);
}
```

Requires "export HSA_XNACK=1" or map clause

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

Querying what implementation does with each compute directive

- Add `export LIBOMPTARGET_KERNEL_TRACE=1`

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

```
DEVID:  0 SGN:5 ConstWGSize:256  args: 5 teamsXthrds:( 416X 256) reqd:(   0X   0)
lds_usage:0B sgpr_count:24 vgpr_count:8 sgpr_spill_count:0 vgpr_spill_count:0
tripcount:10000000 rpc:0 n:__omp_offloading_34_8975356_saxpy_l8
Time of kernel: 0.082906
```

- The next slide shows four different directives and the results in the same quad chart layout as the earlier slide
  - Note that the compiler is generating 624 teams or 6 workgroups x 104 compute units for the MI210. We can launch 32 waves per compute unit (CU) for this low register usage. At a workgroup size of 256, we are using 4 waves per workgroup. 4 waves (per workgroup) times 6 workgroups is 24 waves, a little under the 32 wave limit.
  - Later versions of the AMD compiler also report the occupancy.

Comparing subsets of GPU parallel compute directive

```
#pragma omp target
DEVID:  0 SGN:3 ConstWGSize:257  args: 5
teamsXthrds:(   1X 256) reqd:(   0X   0)
lds_usage:16B sgpr_count:16 vgpr_count:3
sgpr_spill_count:0 vgpr_spill_count:0
tripcount:0 rpc:0
n:__omp_offloading_34_5c4ed40a_saxpy_l8
Time of kernel: 5.407085
```

```
#pragma omp target teams
DEVID:  0 SGN:3 ConstWGSize:257  args: 5
teamsXthrds:( 624X 256) reqd:(   0X   0)
lds_usage:16B sgpr_count:12 vgpr_count:3
sgpr_spill_count:0 vgpr_spill_count:0
tripcount:0 rpc:0
n:__omp_offloading_34_5c4ed40b_saxpy_l8
Time of kernel: 11.166301
```

```
#pragma omp target teams distribute
DEVID:  0 SGN:3 ConstWGSize:257  args: 5
teamsXthrds:( 624X 256) reqd:(   0X   0)
lds_usage:16B sgpr_count:24 vgpr_count:3
sgpr_spill_count:0 vgpr_spill_count:0
tripcount:10000000 rpc:0
n:__omp_offloading_34_5c4ed40c_saxpy_l8
Time of kernel: 0.149113
```

```
#pragma omp target parallel for
DEVID:  0 SGN:2 ConstWGSize:256  args: 5
teamsXthrds:(   1X 256) reqd:(   0X   0)
lds_usage:32B sgpr_count:25
vgpr_count:17 sgpr_spill_count:0
vgpr_spill_count:0 tripcount:0 rpc:0
n:__omp_offloading_34_5c4ed416_saxpy_l8
Time of kernel: 0.126748
```

For reference; `#pragma omp target teams distribute parallel for` from previous slide `Time of kernel: 0.082906`

Breaking up the OpenMP® Compute Constructs

- The single-line directives can be split apart into separate directives. We've been using the single line compute construct something like the following
  - `#pragma omp target teams distribute parallel for simd`
- But we are not limited to just a single line. We can break up the compute directive into multiple lines. The simplest multi-line directives are equivalent to the single line form.

- Compute on the GPU with all teams (workgroups) and data partitioned, but only the main thread
  - `#pragma omp target teams distribute`
- Parallelize the following for loop (use all the threads in a workgroup)
  - `#pragma omp parallel for simd`

Proper nomenclature is that alone or first on a line, it is a directive. When it follows a directive, it is a modifier or a clause.

Multi-level Parallel saxpy

- We can split the directives across an outer loop and an inner loop to have more control. Usually, it is best to let the compiler do this as it generally does a better job. But there are special cases where the application developer may have information about something like typical sizes of loops. Note that this is somewhat different than the previous saxpy example in that it is built around a 2D data structure.

```
void saxpy(float a, float **x, float **y, int M, int N) {
   double tb, te;

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

   printf("Time of kernel: %lf\n", te - tb);

   printf("check output:\n");
   printf("y[0][0] %lf\n",y[0][0]);
   printf("y[N-1][M-1] %lf\n",y[N-1][M-1]);
}
```

Notes

- While this example shows split level pragmas that might be useful in special cases:
  - We do not recommend doing this in simple cases – let the compiler decide how to do the parallelism
  - Add a collapse clause instead – it increases the parallel work
  - Generally, it is not a good idea to use split level to force performance optimization, but only to address special cases

- Special cases
  - Small sizes of the outer or inner loop (maybe even unroll a loop?)
  - Something special being done on the inner loop where the synchronization benefits the required work

Exploring the split level directive

- Again, we'll use the export `LIBOMPTARGET_KERNEL_TRACE=1` setting to explore what the compiler does with each case

```
cd HPCTrainingExamples/Pragma_Examples/OpenMP/C/ComplexComputeConstructs
module load amdclang
make saxpy_gpu_collapse
./saxpy_gpu_collapse
```

```
DEVID:  0 SGN:5 ConstWGSize:256  args: 6 teamsXthrds:(3907X 256) reqd:(   0X   0) lds_usage:0B
sgpr_count:29 vgpr_count:17 sgpr_spill_count:0 vgpr_spill_count:0 tripcount:1000000 rpc:0 md:0
md_LB:-1 md_UB:-1 Max Occupancy: 8 Achieved Occupancy: 100%
n:__omp_offloading_34_5c4ed40e_saxpy_l9
Time of kernel: 0.027777
```

Exploring the split level directive

- Now running the split level directive

```
make saxpy_gpu_split_level
./saxpy_gpu_split_level
```

```
DEVID:  0 SGN:3 ConstWGSize:257  args: 6 teamsXthrds:( 416X 256) reqd:(   0X   0) lds_usage:36B
sgpr_count:27 vgpr_count:24 sgpr_spill_count:0 vgpr_spill_count:0 tripcount:1000 rpc:0 md:0
md_LB:-1 md_UB:-1 Max Occupancy: 8 Achieved Occupancy: 50%
n:__omp_offloading_34_5c4ed411_saxpy_l9
Time of kernel: 0.027449
```

- We only get a report for the outer loop. Run time is slightly faster. Vector register count has gone up and occupancy is much lower.

Other compute clauses – `tile` and more

- `tile` – block the loops into small tiles rather than a standard loop traversal of all x and then y.
- `num_teams(x)` – launch the kernel with the specified number of thread blocks
- `num_threads(x)` – launch the kernel with the specified number of threads
- `thread_limit(x)` – cause the compiler to generate code with a maximum number of threads, reducing register pressure (some compilers are still adding this optimization)
- nowait – do not wait at end of compute kernel. Default is to wait. This is one of the optimization options, but it can lead to race conditions and incorrect results
- reduction(op: x) – special case where multiple iterations write to common location(2). This might be a sum, min, max or similar type of operation

The `thread_limit` clause: for better compiler optimization

- The most commonly used of these compute clauses is thread_limit
- The `thread_limit` clause specifies the maximum workgroup size for the compiler generated GPU code
- It frees up some additional memory resources in the kernel code such as registers that can make the code more efficient

- num_threads specifies the threads for the code generated for this specific case (but not all situations)

Understanding hardware options

- `rocminfo`
  - 110 CUs
  - Wavefront of size 64
  - 4 SIMDs per CU

```
Node:                    11
Device Type:             GPU
Cache Info:
  L1:                      16(0x10) KB
  L2:                      8192(0x2000) KB
Chip ID:                 29704(0x7408)
Cacheline Size:          64(0x40)
Max Clock Freq. (MHz):   1700
BDFID:                   56832
Internal Node ID:        11
Compute Unit:            110
SIMDs per CU:            4
Shader Engines:          8
Shader Arrs. per Eng.:   1
WatchPts on Addr. Ranges:4
Features:                KERNEL_DISPATCH
Fast F16 Operation:      TRUE
Wavefront Size:          64(0x40)
Workgroup Max Size:      1024(0x400)
Workgroup Max Size per Dimension:
  x                        1024(0x400)
  y                        1024(0x400)
  z                        1024(0x400)
Max Waves Per CU:        32(0x20)
Max Work-item Per CU:    2048(0x800)
```

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

Options for `#pragma omp target teams`:

- `num_teams(220)`: it is good practice to set the number of workgroups as a multiple of the CUs (which is 110 in this case)
- `thread_limit(256)`: the number of threads per workgroup should be a multiple of 64

The total number of threads is:
`num_teams*thread_limit` which should evenly divide the trip count of a loop

Reductions to scalar or arrays

OpenMP® Offloading Example: Reduction

```
#include <stdio.h>
#include <stdlib.h>
#define N 5000000
int main(){
 double *a,  *b;
  a = (double*)malloc(sizeof(double) * N);
  b = (double*)malloc(sizeof(double) * N);
  for(int i = 0; i < N; i++){
    a[i] = 1.0;
    b[i] = 1.0;
  }

  double sum = 0.0;
  #pragma omp target data map(to:a[0:N], b[0:N])
  ///#pragma omp target teams distribute parallel for private(i) map(tofrom:sum) reduction(+:sum)
  #pragma omp target teams distribute parallel for reduction(+:sum)
  for(int i = 0; i < N; i++)
    sum += a[i] * b[i];

  printf("SUM = %f\n", sum);
  free(a);
  free(b);
  return 0;
}
```

Sum is automatically set to zero for sum reductions. It does not need to be set for OpenMP, but it is needed for serial code compiled without OpenMP.

Note: Scalars are implicitly firstprivate in target constructs (as of OpenMP 4.5)

Data directive to move data to device (GPU)

Compute loop on GPU, copy sum to and from GPU and do a sum reduction on sum variable

`map` clause should not be included if it is a reduction variable. Each reduction variable is initialized based on the type of reduction and then the host version of the reduction variable is updated with the final result. `private(i)` is also not needed – index variables are automatically private.

Real-world cases

It is pretty common in physics applications that there is a long computational loop and at the end there is a reduction into an array variable

* Weather/Climate codes where energy contributions are summed into ocean or atmospheric levels
* Reaction energies are summed into a particle array

OpenMP® Offloading Example: Reduction to Array

```
#include <stdio.h>
#include <stdlib.h>
#define N 5000000
int main(){
 double **a,  **b;
 // allocate 2D arrays a and b

  double sum[n];
  #pragma omp target data map(to:a[0:M][0:N], b[0:M][0:N])
  #pragma omp target teams distribute parallel for reduction(+:sum[0:n])
 for(int j = 0; j < N; j++){
    for(int i = 0; i < N; i++){
       sum[j] += a[j][i] * b[j][i];
    }
  }

 for(int j = 0; j < N; j++){
    printf("SUM = %f\n", sum);
 }

  // free 2D arrays

  return 0;
}
```

Data directive to move data to device (GPU)

Compute loop on GPU, copy sum to and from GPU and do a sum reduction on sum variable

Calling a subroutine from a target region

Originally, pragma based languages required the in-lining of subroutines in target regions

To call a subroutine from a target region, it must have the pragma declare target added to the specification block and if not visible, to the prototype

```
#pragma omp declare target
void *compute(){
   ….
};
#pragma omp end declare target
```

Data can also be declared to be on the target

```
#pragma omp declare target
double constants[10] = { ….}
#pragma omp end declare target
```

Not all compilers recognize device routines or can have difficulties with global variables that are used in device routines.

Global data used in “compute” subroutine

- Could be global data in another file (extern), object or common block

```
#pragma omp declare target
double constants[10];
#pragma omp end declare target
```

- // another file or lots of code

- Compute is called from a target region

```
#pragma omp declare target
Extern double constants[10];
#pragma omp end declare target

#pragma omp declare target
void *compute (int cindex, double *x){
  *x = 1.0 + constants[cindex];
}
#pragma omp end declare target
```

-

ROCm Note:

- ROCm and LLVM™ trunk are able to build device routines in declare target blocks to be used from within target regions.
- There is also the feature of "implicit declare target" where the following code will produce the desired effect:

```
int foo(int x) {
  return x+1;
}

int main() {
  int *a = new int[N];

  #pragma omp target teams loop
  for(size_t i = 0; i < N; i++)
    a[i] = foo(i);
}
```

- foo will be implicitly made "declare target" by the compiler because when building the call site in the target region, the *definition* of foo is available.
- Conversely, if the definition of foo is in a separate file, then implicit declare target will not be able to build it for the device, and unless the programmer adds a "declare target" around it, it will result in a linker error.

- The second point is that global variables are accessible from within declare target functions when using ROCm.

Complex cases

- Usually, the difficulty in porting a code to OpenMP® is due to complex combinations of code
- Watch out for structs and classes or any data type that doesn’t map to simple arrays
- Allocatables, pointers or implicit allocation/reallocation are tricky to get right, both for the programmer and the compiler developer
- The `declare` pragma can be tricky for both subroutines and data. Some variation in implementations also exist across compilers
- Deep copies (structs or classes that contain pointers and data that won’t be copied over to the device)
- Anything that has been added more recently to the OpenMP standard may have portability issues

Dynamic arrays dependent on program input

- Original code has a statically sized array.

```
#pragma omp target data
double atmos_temp[nsize];
#pragma omp end target data
```

- We want to change it to a dynamic size based on input

```
#pragma omp target data
double *atmos_temp;
#pragma omp end target data

atmos_temp = (double *) malloc(nsize_input * sizeof(double));
// use in device computation loop
```

- The malloc is going to assign a new pointer location to atmos_temp. Instead use `enter data` after allocation and preferably set values in a device loop.

```
#pragma omp target enter data (alloc:atmos_temp[0:nsize_input])
```

Harder to catch in Fortran and C++

Dynamic array as part of a struct

```
#pragma omp target data
struct {
   int n;
   double *x;
}
#pragma omp end target data

x = (double *)malloc(nsize*sizeof(double));
#pragma omp target enter data  (alloc:x[0:nsize]))
/// lots of code
#pragma omp target exit data (release:x)
```

– don’t use delete? (open question)

- There is some variation in current compilers on how they handle release vs delete
- OpenMP release has same behavior as OpenACC delete

Summary

- AMD OpenMP® compilers can offload computation to AMD GPUs
  - Good support for C, C++, and Fortran languages
  - Mature offload model w/ support for asynchronous offload/transfer
- Backed by an Industry language standard
- Composability across programming languages (C,C++,Fortran)

- Portability across GPU platforms for core OpenMP® constructs
  - Tightly integrates with OpenMP multi-threading on the host

- Exercise Instructions:
  - HPCTrainingExamples/Pragma_Examples/README.md

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.


AMD