Advanced OpenMP®

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

AMD
together we advance_

Advanced OpenMP®

1. Region Concept
2. Memory Management
   - Memory Management Capabilities
   - Optimizing Memory Movement Between Host and Device
3. Kernel Resources and Optimization

Introduction

With GPU programming we have two considerations that must be addressed

1. **Memory and Data Management**
   1. Between the host and the device
   2. From GPU main memory to the Compute Unit or Device
2. **Code Execution**
   - Managing compute resources
   - Which device to execute operation on
   - Expression of parallelism

- We'll tackle how to address each of these considerations in the following slides and exercises

OpenMP® heavily relies on region concept

- What are regions?
  - A part of the code where a pragma applies
  - Default is the normal “block” of code following the directive
  - Can be specified by { }s in C or an end directive in Fortran

- What kinds of regions are there?
  - Data regions – data is on the GPU in this code region
  - Target regions – code in region is executed on the GPU
  - Parallel regions – code in region is executed in parallel

- Original OpenMP specification only had structured data regions
  - How to handle Object-oriented code and other patterns?

➔ Later version of the standard added unstructured data region concept

Structured vs Unstructured Data regions

Structured data region

```
#pragma omp target data map(tofrom: x[0:n])
{
#pragma omp target teams distribute parallel for simd
    for (int i = 0; i < n; n++){
       x[i] = 0.0;
    }
}
```

Unstructured data region

```
class myclass (int n) {
    myclass(){
        x=new double[n];
        #pragma omp target enter data map(alloc: x[0:n])
    }

    ~myclass(){
       #pragma omp target exit data map(delete: x[0:n])
       delete [] x;
       }
```

While object exists

Memory Management Capabilities

Different Memory Management Capabilities

Explicit Memory Management

Requires explicit memory movement directives.

	Host	Device
x	0x000000000174b0e0	0x00007f617c434000
y	0x000000000175e970	0x00007f617c448000
z	0x0000000001772200	0x00007f617c420000

OpenMP® 5.0

Managed Memory

export HSA_XNACK=1

- The Operating System will move memory automatically between host and device.

	Host	Device
x	0x000000000174b0e0	0x00007f617c434000
y	0x000000000175e970	0x00007f617c448000
z	0x0000000001772200	0x00007f617c420000

OpenMP® 5.0

Single Memory address

#pragma omp requires unified_shared_memory

- a pointer will always refer to the same location in memory from all devices accessible through OpenMP

	Host/Device
x	0x000000000174b0e0
y	0x000000000175e970
z	0x0000000001772200

Optimizing Memory Movement Between Host and Device

Understanding the behavior of the memory movement pragmas

- The full set of examples is at https://github.com/AMD/HPCTrainingExamples in the HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/memory_pragmas directory

- We'll experiment with different combinations of clauses in the pragmas. By setting LIBOMPTARGET_INFO=-1, we can see what the OpenMP® runtime does behind the scenes.

Summary of OpenMP® memory pragmas and what they do

OpenMP clause	Allocates/deletes device memory	Modifies reference counter	Copies data
Map to/from[1]	Yes	Yes	Yes
Map always[2]	Yes	Yes	Yes
Map alloc/delete[3]	Yes[4]	Yes	No
Map release	If reference counter 0, delete	Decrements	No
Update to/from	No	No	Yes

Notes:

1. "Map to" checks if the memory is already allocated for the device.
   a. If not allocated, the device memory is allocated and the reference counter is set to one, and the data is copied to the device
   b. If allocated, the size is checked, and the reference counter is incremented

   Similar for "map from“
2. Same as to/from, but always copies the memory over even if it already exists
3. "Map alloc" checks if the memory is already allocated for the device.
   a. if not allocated, the device memory is allocated, and the reference counter is set to one
   b. if allocated, the size is checked, and the reference counter is incremented.

   "Map delete" will delete the memory and set the reference counter to zero
4. More generally, to cover single memory spaces, the data must be available in the memory space.

Basic OpenMP® daxpy code (2 slides) – mem1.cc

```
33 int main(int argc, char* argv[])
34 {
35    int num_iteration=NTIMERS;
36    int n = 100000;
37    double main_timer = 0.0;
38    double main_start = omp_get_wtime();
39    if (argc > 1) {
40       n=atoi(argv[1]);
41    }
42    double a = 3.0;
43    double *x = new double[n];
44    double *y = new double[n];
45    double *z = new double[n];
46
47    for (int i = 0; i < n; i++) {
48         x[i] = 2.0;
49         y[i] = 1.0;
50    }
51
52    double * timers = (double *)calloc(num_iteration,sizeof(double));
53    for (int iter=0;iter<num_iteration; iter++)
54    {
55         double start = omp_get_wtime();
56
57         daxpy(n, a, x, y, z);
58
59         timers[iter] = omp_get_wtime()-start;
60    }
61
```

Adding alignment to the memory allocation is highly recommended. This will be discussed in the next section. The line of code with alignment specification is
double *x = new (std::align_val_t(128)) double[n];

Basic OpenMP® daxpy code (continued)

```
62    double sum_time =  0.0;
63    double max_time = -1.0e10;
64    double min_time =  1.0e10;
65    for (int iter=0; iter<num_iteration; iter++) {
66         sum_time += timers[iter];
67         max_time  = max(max_time,timers[iter]);
68         min_time  = min(min_time,timers[iter]);
69    }
70
71    double avg_time = sum_time / (double)num_iteration;
72
73    cout << "-Timing in Seconds: min=" << fixed << setprecision(6) << min_time << ", max=" <<max_time << ", avg=" << avg_time << endl;
74
75    main_timer = omp_get_wtime()-main_start;
76    cout << "-Overall time is " << main_timer << endl;
77
78    cout << "Last Value: z[" << n-1 << "]=" << z[n-1] << endl;
79
80    delete [] x;
81    delete [] y;
82    delete [] z;
83
84    return 0;
85 }
86
87 void daxpy(int n, double a, double *__restrict__ x, double *__restrict__ y, double *__restrict__ z)
88 {
89 #pragma omp target teams distribute parallel for simd map(to: x[0:n], y[0:n]) map(from: z[0:n])
90         for (int i = 0; i < n; i++)
91                 z[i] = a*x[i] + y[i];
92 }
```

Mem1.cc version

Map clause on pragma line just before computational loop

```
mem1.cc:89 #pragma omp target teams distribute parallel for simd map(to: x[0:n], y[0:n]) map(from: z[0:n])
```

Running this with LIBOMPTARGET_INFO=-1, we can see the memory operations. All occur from the pragma at line 89.

```
LIBOMPTARGET_INFO Report
Libomptarget info: Entering OpenMP kernel at mem1.cc:89:1 with 5 arguments:
Libomptarget info: firstprivate(n)[4] (implicit)   ← Note implicit firstprivate for scalar arguments
Libomptarget info: from(z[0:n])[80000]
Libomptarget info: firstprivate(a)[8] (implicit)
Libomptarget info: to(x[0:n])[80000]
Libomptarget info: to(y[0:n])[80000]
Libomptarget info: Creating new map entry with <...> TgtPtrBegin=0x00007f90b6a20000, Size=80000, DynRefCount=1, HoldRefCount=0, Name=z[0:n]
Libomptarget info: Creating new map entry with <...> TgtPtrBegin=0x00007f90b6a34000, Size=80000, DynRefCount=1, HoldRefCount=0, Name=x[0:n]
Libomptarget info: Copying data from host to device, HstPtr=0x0000000000c2f0e0, TgtPtr=0x00007f90b6a34000, Size=80000, Name=x[0:n]
Libomptarget info: Creating new map entry with <...> TgtPtrBegin=0x00007f90b6a48000, Size=80000, DynRefCount=1, HoldRefCount=0, Name=y[0:n]
Libomptarget info: Copying data from host to device, HstPtr=0x0000000000c42970, TgtPtr=0x00007f90b6a48000, Size=80000, Name=y[0:n]
Libomptarget info: Mapping exists with HstPtrBegin=0x0000000000c56200, TgtPtrBegin=0x00007f90b6a20000, Size=80000, DynRefCount=1 (update suppressed), HoldRefCount=0
Libomptarget info: Mapping exists with HstPtrBegin=0x0000000000c2f0e0, TgtPtrBegin=0x00007f90b6a34000, Size=80000, DynRefCount=1 (update suppressed), HoldRefCount=0
Libomptarget info: Mapping exists with HstPtrBegin=0x0000000000c42970, TgtPtrBegin=0x00007f90b6a48000, Size=80000, DynRefCount=1 (update suppressed), HoldRefCount=0
Libomptarget info: Mapping exists with HstPtrBegin=0x0000000000c42970, TgtPtrBegin=0x00007f90b6a48000, Size=80000, DynRefCount=0 (decremented, delayed deletion) <...>
Libomptarget info: Mapping exists with HstPtrBegin=0x0000000000c2f0e0, TgtPtrBegin=0x00007f90b6a34000, Size=80000, DynRefCount=0 (decremented, delayed deletion) <...>
Libomptarget info: Mapping exists with HstPtrBegin=0x0000000000c56200, TgtPtrBegin=0x00007f90b6a20000, Size=80000, DynRefCount=0 (decremented, delayed deletion) <...>
Libomptarget info: Copying data from device to host, TgtPtr=0x00007f90b6a20000, HstPtr=0x0000000000c56200, Size=80000, Name=z[0:n]
Libomptarget info: Removing map entry with HstPtrBegin=0x0000000000c42970, TgtPtrBegin=0x00007f90b6a48000, Size=80000, Name=y[0:n]
Libomptarget info: Removing map entry with HstPtrBegin=0x0000000000c2f0e0, TgtPtrBegin=0x00007f90b6a34000, Size=80000, Name=x[0:n]
Libomptarget info: Removing map entry with HstPtrBegin=0x0000000000c56200, TgtPtrBegin=0x00007f90b6a20000, Size=80000, Name=z[0:n]
```

Device memory allocated and Reference count incremented

Data copied

Device array deleted

Mem2.cc version -- Add enter/exit data alloc/delete when memory is created/freed

After new

```
mem2.cc:#pragma omp target enter data map(alloc: x[0:n], y[0:n], z[0:n])
```

Keep map on computational loop. The map to/from should check if the data exists. If not, it will allocate/delete it. Then it will do the copies to and from. This will increment the Reference Counter and decrement it at end of loop.

```
mem2.cc:#pragma omp target teams distribute parallel for simd map(to: x[0:n], y[0:n]) map(from: z[0:n])
```

Before delete

```
mem2.cc:#pragma omp target exit data map(delete: x[0:n], y[0:n], z[0:n])
```

LIBOMPTARGET_INFO Report

After new:

```
Libomptarget info: Creating new map entry with <…>TgtPtrBegin=0x00007ff58f020000, Size=80000, DynRefCount=1, HoldRefCount=0, Name=x[0:n]
```

Computational Loop:

```
   Libomptarget info: Mapping exists with HstPtrBegin=0x000000000161d200, TgtPtrBegin=0x00007ff58f048000, Size=80000, DynRefCount=2
(incremented), HoldRefCount=0, Name=z[0:n]
   Libomptarget info: Mapping exists with HstPtrBegin=0x000000000161d200, TgtPtrBegin=0x00007ff58f048000, Size=80000, DynRefCount=1
(decremented), HoldRefCount=0
```

After delete:

```
   Libomptarget device 0 info: Mapping exists with HstPtrBegin=0x000000000161d200, TgtPtrBegin=0x00007ff58f048000, Size=80000,
DynRefCount=0 (reset, delayed deletion), HoldRefCount=0
   Libomptarget device 0 info: Removing map entry with HstPtrBegin=0x000000000161d200, TgtPtrBegin=0x00007ff58f048000, Size=80000,
Name=z[0:n]
```

Mem3.cc version – replace map on computation loop with updates

LIBOMPTARGET_INFO Report

At update – check device array exists and copies data. Reference counter not incremented

```
   Libomptarget info: to(x[0:n])[80000]
   Libomptarget info: Mapping exists with HstPtrBegin=0x0000000000fe10e0, TgtPtrBegin=0x00007ff998a20000, Size=80000,
DynRefCount=1 (update suppressed), HoldRefCount=0
   Libomptarget info: Copying data from host to device, HstPtr=0x0000000000fe10e0, TgtPtr=0x00007ff998a20000, Size=80000,
Name=x[0:n]
```

At computational loop (no map directive) – note implicit checks, increments and decrements of reference counter

```
   Libomptarget device 0 info: use_address(x)[0] (implicit)
   Libomptarget device 0 info: Mapping exists (implicit) with HstPtrBegin=0x0000000000fe10e0, TgtPtrBegin=0x00007ff998a20000,
Size=0, DynRefCount=2 (incremented), HoldRefCount=0, Name=x
   Libomptarget device 0 info: Mapping exists with HstPtrBegin=0x0000000000fe10e0, TgtPtrBegin=0x00007ff998a20000, Size=0, Dy
   Libomptarget device 0 info: Mapping exists with HstPtrBegin=0x0000000000fe10e0, TgtPtrBegin=0x00007ff998a20000, Size=0,
DynRefCount=1 (decremented), HoldRefCount=0nRefCount=2 (update suppressed), HoldRefCount=0
```

Set LIBOMPTARGET_INFO flag at runtime

```
extern “C” void __tgt_set_info_flag(uint32_t);

<...>

__tgt_set_info_flag(-1);
#pragma omp target teams distribute parallel for simd map(to: x[0:n], y[0:n]) map(from: z[0:n])
for (int i = 0; i < n; i++)
   z[i] = a*x[i] + y[i];
__tgt_set_info_flag(0);
```

By setting this LIBOMPTARGET_INFO flag at runtime, you can see the detailed information at a particular point in your code

Optimizing Memory Bandwidth Utilization to GPU Main Memory

Memory layout and alignment

Memory allocations with longer memory alignment can get some improvement in performance. The suggested length is the cache line length (64 bytes on MI200 series and 128 bytes on MI300).

The default memory alignment obtained with system allocators such as "malloc" or “new” is 16 bytes.

How to change the memory alignment

- Compiler flag (`-faligned-allocation -fnew-alignment=64`)
- Specifying alignment property in source code.
- Using system allocators such as posix_memalign
- More effort needed to get vector and valarray classes to allocate desired memory alignment

The impact of alignment seems to be less with more recent GPU models and may vary with your application.

```
#pragma omp requires unified_shared_memory
int main(){

  double * X, * Y, *Z;
  size_t N = (size_t) 1024*1024*1024/sizeof(double);
  X = new double[N];   Y = new double[N];
  X =  new (std::align_val_t(128)) double[N];
  if (N < 10)   Y =  new (std::align_val_t(16)) double[N];
  else          Y =  new (std::align_val_t(128)) double[N];

  #pragma omp target teams distribute parallel for if(target:N>2000)
  for (size_t i = 0; i < N; ++i)
     X[i] = 0.000001*i;

  #pragma omp target teams distribute parallel for if(target:N>2000)
  for (size_t i = 0; i < N; ++i)
     Y[i] = X[i]

  delete[] X;  delete[] Y;
  return 0;
}
```

Measure alignment and workgroup size impact on bandwidth

-- Examine the source code

- Get test code from training examples repo

  ```
  git clone https://github.com/amd/HPCTrainingExamples
  ```

- Go to test directory

  ```
  cd HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/optimization/alignment
  ```

- Examine one of the source files – align_two_kernels.cc

- X = new **(std::align_val_t(alignment_length))** double[N];
  - Alignment_length is set by an input argument and set to various sizes: 16 32 64 128 256
- #pragma omp target teams distribute parallel for **thread_limit(BLOCKSIZE)**
  - BLOCKSIZE is set to the following sizes 64 128 256 512 1024

Measure alignment and workgroup size impact on bandwidth

-- Run the example

- Now let's run the example with alignment and workgroup size settings to see the effect

  Set up environment

  ```
  module load amdclang
  export HSA_XNACK=1
  ```

- Build codes

  ```
  make
  ```

- Run codes

  ```
  ./run_align_two_kernels.sh
  ```

- There is another single copy kernel that can be run with `./run_align_simple_copy.sh` and to run both, `./run_align.sh`
- Try different array sizes and how the alignment impact varies

Alignment study results by AMD Instinct™ GPU

- MI100 with ROCm 5.x (Original Study) Theoretical Bandwidth 1.23 TB/s or 1230 GB/s[1] (about 1145 GiB/s)

Alignment →	16	32	64	128	256
OpenMP: thread_limit(128)	540 GB/s [2]	750 GB/s [2]	750 GB/s [2]	680 GB/s [2]	870 GB/s [2]
OpenMP: thread_limit(1024)	990 GB/s [2]	1000 GB/s [2]	1010 GB/s [2]	960 GB/s [2]	1040 GB/s [2]

- MI210 with ROCm 6.3.3 Theoretical Bandwidth 1.6 TB/s or 1600 GB/s[1] (about 1490 GiB/s)

Alignment →	16	32	64	128	256
OpenMP: thread_limit(128)	623 GiB/s	1216 GiB/s	1214 GiB/s	1158 GiB/s	1275 GiB/s
OpenMP: thread_limit(1024)	598 GiB/s	1258 GiB/s	1256 GiB/s	1240 GiB/s	1320 GiB/s

- MI300A with ROCm 6.3.3 Theoretical Bandwidth 5.3 TB/s or 5300 GB/s[1] (about 4936 GiB/s)

Alignment →	16	32	64	128	256
OpenMP: thread_limit(128)	3390 GiB/s	3650 GiB/s	3662 GiB/s	3748 GiB/s	3772 GiB/s
OpenMP: thread_limit(1024)	3650 GiB/s	3729 GiB/s	3726 GiB/s	3756 GiB/s	3778 GiB/s

Notes: MI210 and MI300A system had other users and jobs running. Try the exercise and see if you get similar results.

[1] These are decimal units (GB,TB, not GiB,TiB)

[2] Not clear whether these are GiB or GB

A note on bandwidth measurement

- The stream measure of bandwidth is "application bandwidth"
- Each read is counted as 1 load, and each write is counted as 1 store
- Typical hardware implements writes as a load of a cache line and then a store. In this case each write is a load and a store. For a copy, there is only a store for the write.
- To get hardware bandwidth, the numbers from the application bandwidth must be multiplied by a correction factor.
- We have shown the hardware bandwidth for the MI210 and the MI300A. We have also shown them in powers of two (binary) units – GiB and TiB.
- Original measurement units for the MI100 is not certain
- Theoretical bandwidths are given in GBs

- Bandwidth tests at rocm Docs – these are general bandwidth measurements and not stream bandwidth https://rocm.docs.amd.com/projects/rocm_bandwidth_test/en/latest/index.html#rocm-bandwidth-test-documentation

Background material: hardware bandwidth vs application bandwidth

- George Hager on stream bandwidth https://blogs.fau.de/hager/archives/8263
  - Write requires a write allocate (read and then write), nonstreaming stores
  - Correction factors – 1.33 for add/triad, 1.5 for scale, 1.5 for compiler implemented copy and 1.0 for memcpy
- McCalpin on counting bytes https://www.cs.virginia.edu/stream/ref.html#counting
- Stack Exchange -- https://superuser.com/questions/1815148/expected-results-of-a-stream-memory-bandwidth-benchmark

Partitioning and NUMA regions

- Compute partitions and memory partitions are possible with MI300A
  - ROCm Blogs post: Deep dive into the MI300 compute and memory partition modes — ROCm Blogs
- Default compute partition is the Single Partition X-celerator (SPX) partitioning mode
  - All XCDs on the device are seen as a single logical compute element
  - Workgroups launched are round-robined across the GPU devices. Memory accesses from another GPU memory has an overhead of about 15%
- In a recent ROCm release, information on the compute partition Core Partitioned X-celerator (CPX) mode and different NUMA regions (memory partition) was announced
  - With CPX, each GPU is seen as a single device
- Changing the partition is done with the `amd-smi` command
- On the MI300X, the stream benchmark improved by 5%
- Other experiences have shown bandwidth improvements in real applications
- As a new configuration, there have also been some bug reports
- Even the HPL benchmark is limited by bandwidth. Also GCDs may be able to run at higher frequencies.

Kernel traces and optimizations

Kernel Optimizations

- These examples are at https://github.com/AMD/HPCTrainingExamples in the HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/kernel_pragmas directory

- We'll experiment with different optimizations with pragmas. By setting `LIBOMPTARGET_KERNEL_TRACE=1` or 2, we can see what the OpenMP® runtime does behind the scenes.
  - Setting to 1 shows the name of every kernel, number of teams, threads, and register usage.
  - Setting to 2 prints timing and data transfer information

- LIBOMPTARGET_DEBUG=1 will show more information about data transfer operations and kernel launch
- HPE/Cray
  - `CRAY_ACC_DEBUG=[1,2,3]`
  - `-hlist=aimd` at compile time

Kernel1.cc

- `export HSA_XNACK=1`
- `export LIBOMPTARGET_KERNEL_TRACE=1`
- `mkdir build && cd build`
- `CXX=amdclang++ cmake ..`
- `make`
- `./kernel1`

```
LIBOMPTARGET_KERNEL_TRACE Report
DEVID: 0 SGN:2 ConstWGSize:256  args: 3 teamsXthrds:( 391X 256) reqd:(   0X   0) lds_usage:9784B sgpr_count:106 vgpr_count:58 sgpr_spill_count:39 vgpr_spill_count:0
tripcount:100000 rpc:1 n:__omp_offloading_3d_1a2def2_main_l52
DEVID: 0 SGN:2 ConstWGSize:256  args: 5 teamsXthrds:( 391X 256) reqd:(   0X   0) lds_usage:9784B sgpr_count:106 vgpr_count:56 sgpr_spill_count:47 vgpr_spill_count:0
tripcount:100000 rpc:1 n:__omp_offloading_3d_1a2def2__Z5daxpyidPdS_S__l97
```

kernel2.cc

- Change number of threads – add num_threads(64)

- New report

```
LIBOMPTARGET_KERNEL_TRACE Report
DEVID: 0 SGN:2 ConstWGSize:64   args: 3 teamsXthrds:( 416X  64) reqd:(   0X  64) lds_usage:9784B sgpr_count:106 vgpr_count:59 sgpr_spill_count:42 vgpr_spill_count:0
tripcount:100000 rpc:1 n:__omp_offloading_3d_1a2def6_main_l52
DEVID: 0 SGN:2 ConstWGSize:64   args: 5 teamsXthrds:( 416X  64) reqd:(   0X  64) lds_usage:9784B sgpr_count:106 vgpr_count:57 sgpr_spill_count:48 vgpr_spill_count:0
tripcount:100000 rpc:1 n:__omp_offloading_3d_1a2def6__Z5daxpyidPdS_S__l97
```

kernel3.cc

- Add thread limit for kernel – num_threads(64) thread_limit(64)

- New report

```
LIBOMPTARGET_KERNEL_TRACE Report
DEVID: 0 SGN:2 ConstWGSize:64   args: 3 teamsXthrds:( 416X  64) reqd:(   0X  64) lds_usage:9784B sgpr_count:106 vgpr_count:55 sgpr_spill_count:37 vgpr_spill_count:0
tripcount:100000 rpc:1 n:__omp_offloading_3d_1a2def8_main_l52
DEVID: 0 SGN:2 ConstWGSize:64   args: 5 teamsXthrds:( 416X  64) reqd:(   0X  64) lds_usage:9784B sgpr_count:106 vgpr_count:53 sgpr_spill_count:45 vgpr_spill_count:0
tripcount:100000 rpc:1 n:__omp_offloading_3d_1a2def8__Z5daxpyidPdS_S__l97
```

Summary

OpenMP compute loop clauses	Workgroup size	LDS Usage	SGPR	VGPR	SGPR Spill	VGPR Spill
Simple parallel loop	256 256	9784 B 9784 B	106 106	58 56	39 47	0 0
num_threads(64)	64 64	9784 B 9784 B	106 106	59 57	42 48	0 0
num_threads(64) thread_limit(64)	64 64	9784 B 9784 B	106 106	55 53	37 45	0 0

- Note that reducing the threads does not reduce the VGPRs
- Adding the thread_limit clause does reduce the VGPRs

This is a very simple kernel. We are below the VGPR limit for occupancy restrictions. So the impact in this case is small. Try these changes on your larger kernels and see what it does there.

Register pressure and occupancy for MI250X/MI300

Note: When greater than 256, additional vector registers are stored in scratch, a slower memory. In most cases this should be avoided. We are below that number, but we are still concerned with the limit on the number of waves that can be scheduled.

This is the column that corresponds to the compiler and profiler report.

Num VGPRs	Occupancy per EU	Occupancy per CU
<= 64	8 waves	32 waves
<= 72	7 waves	28 waves
<= 80	6 waves	24 waves
<= 96	5 waves	20 waves
<= 128	4 waves	16 waves
<= 168	3 waves	12 waves
<= 256	2 waves	8 waves
> 256 (+ spilling to scratch)	1 waves	4 waves

There are 4 arithmetic Execution Units per Compute Unit. The compiler generates the VGPRs for each wavefront to be run on each Execution Unit. The scheduler can place 4 of the same wavefronts to execute on the CU or wavefronts from other tasks.

Atomics

- Atomics are generally faster and safer than older approaches such as mutexes. The capability to perform the operation in one instruction avoids the possibility of interruption as well as being faster. Writing correct code with software locks without any possibility of a race condition is difficult.
- Note that some atomic implementations may be done with a software implementation and may not be strictly done with a single instruction

- atomic built-in functions
- omp atomic

- Heavy use of atomics can cause performance issues. Consider rewriting code with a lot of atomics to use a more efficient approach.
- Atomics have to consider memory coherency. This often means that they must done at the last level cache or take other approaches to guarantee correctness. Caches may have to be invalidated and refreshed and can cause cache update storms.

GPU model differences for Atomics

- MI300A generally does the atomic operation in the last level cache to ensure that the memory is coherent between the GPUs and CPUs.
- MI250 has the capability to do the atomics at different cache levels. This leads to the concept of coarse and fine memory allocations. While more complicated, it does make it possible to have faster atomics if you take some care in the coding.

Coarse/Fine Grain Memory Allocations

- **Coarse grain**: coherence and memory ordering with the whole system are legal at synchronization points (e.g. kernel boundaries). For optimization purposes it avoids coherence until needed.

- **Fine grain**: coherence and memory ordering with the whole system possible within GPU kernels. Allows CPU and GPU (and multiple GPUs) to synchronize while the GPU kernel is running. Reduced cacheability.

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 vlnerabilities 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.


AMD