AMD Node Memory Model

Presenter: Bob Robey
AMD @HLRS
Sept 25-28th, 2023

together we advance_

Authors and Contributors

Alessandro Fanfarillo

Carlo Bertolli

Michael Rowan

Nicholas Curtis

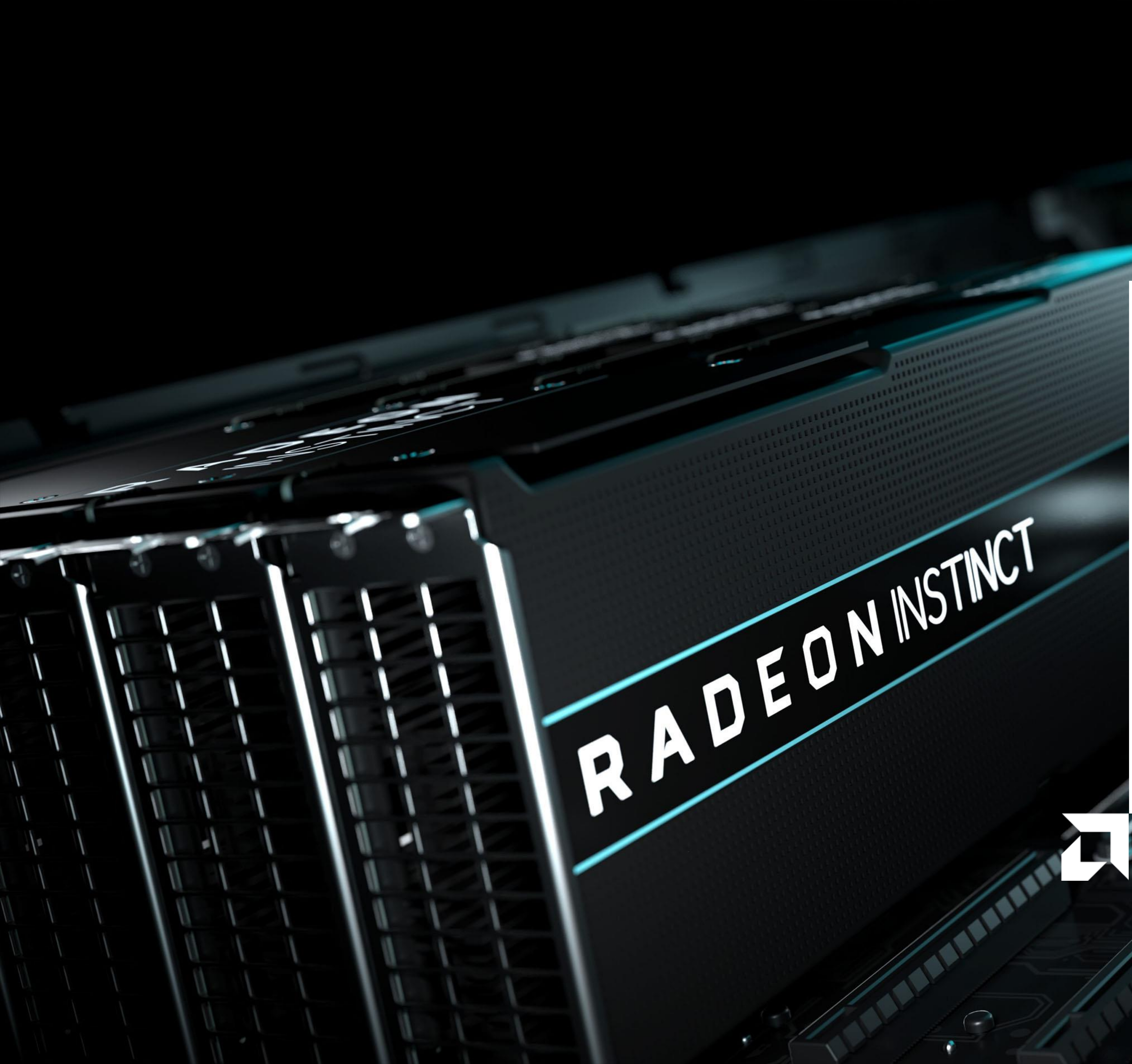
Thanks to all the AMD contributors for their work on creating these materials.

Agenda

- Coarse/Fine grain memory
- Floating-Point (FP) hardware atomics in HIP (safe vs. unsafe)
- Preliminary performance study of coarse vs. fine grain memory
- HMM, XNACK, page migration in HIP
- ROCm™ OpenMP® memory granularity and HW atomics
- Conclusions and future work

Memory Model

- **Memory Model** – a memory model defines the rules for the synchronization of memory modifications between threads, compute hardware and cache. A memory model is critical for parallel computing to help both system developers and application programmers avoid data hazard or race conditions where memory is modified by one entity, but another compute unit fails to get the updated value.



Managed Memory and Page Migration

HMM and XNACK

HMM: Heterogenous Memory Management

- Feature of the Linux kernel
- It provides infrastructure and helpers to integrate non-conventional memory (GPU memory) into regular Linux® kernel
- Any valid pointer on the CPU is also a valid pointer for the GPU and vice versa
- Enables page migration between CPU and GPU
- HMM never frees CPU memory when migration happens
- In this case, the migratable CPU memory is still swappable

XNACK

XNACK

Refers to the AMD GPU's ability to retry memory accesses that fail due to a page fault.

On MI250X, this can be enabled on a per-process based using the environment variable HSA_XNACK=1 and disabled using HSA_XNACK=0. Default decided at boot time.

xnack compiler flag

Compilation mode that can assume three possible values: xnack+, xnack-, xnack any.

To change the xnack compilation mode of a program, xnack+ or xnack- may be appended to the architecture flags:

- `--amdgpu-target=gfx90a:xnack+` [ROCm™ < 4.5] or
- `--offload-arch=gfx90a:xnack+` [ROCm™ >= 4.5]

Supplying multiple xnack options will yield a "fat-binary" with both modes enabled.

When not specified, the default “xnack any” mode will be used.

Code compiled with “xnack any” will run in any case.

Page Migration

Two possible ways to have page migration:

1. Explicitly move pages from/to CPU/GPU via HIP API (e.g., hipMemPrefetchAsync())
2. Automatic page moves from CPU/GPU on a page fault

HMM is always needed for page migration

HMM, HSA_XNACK=1, xnack+(or any) needed for page migration on GPU page fault

Currently, there is no way to detect whether page migration occurred

Only two of the four allocators allow page migration:

- malloc(), system allocators
- hipMallocManaged()

Different allocators show different performance and behavior

Malloc() – system allocator

```
a = (float *) malloc (n*sizeof(float));
Init_on_cpu(a,b,c,n);
Vadd_kernel<<<blocks,threads>>> (array, n); // N=256M
First time: 0.39 seconds  - Following times: 0.005071 seconds

a = new (std::align_val_t(4096)) float[n];
Init_on_cpu(a,b,c,n);
vadd_kernel<<<blocks,threads>>> (array, n); // N=256M
First time: 1.824573 seconds - Following times: 0.002457 seconds
```

When HSA_XNACK=0, no migration will ever happen. It is regular host memory.

When HSA_XNACK=1, migration can happen, but performance may vary based on alignment requirements.

malloc() can place this memory anywhere on the host, the runtime has no control over alignment.

Not having the right alignment may prevent page migration (working on improving this).

hipMallocManaged()

hipMallocManaged() is the most reliable allocator to obtain page migration.

```
hipMallocManaged(&a,n*sizeof(float)); //same for b and c
Init_on_cpu(a,b,c,n);
Vadd_kernel<<<blocks,threads>>> (a,b,c,n); // N=256M
First time: 0.51 seconds  - Following times: 0.0023 seconds // same as hipMalloc memory

Host_vadd(a,b,c,n); // N=256M
First time: 0.94 seconds  - Following times: 0.12 seconds // same as regular CPU memory
```

hipMemPrefetchAsync and hipMemAdvise

hipMemPrefetchAsync() is currently not asynchronous, plans to make it asynchronous.

Works in both directions, CPU < --- > GPU.

HMM is needed to have hipMemPrefetchAsync() working correctly. No action when HMM not available.

hipMemAdvise() current status:

1) hipMemAdviseSetPreferredLocation/Mostly Read (have known limitations)
2) hipMemAdviseSetAccessedBy working

How to use hipMemAdvise():

```
hipDevice_t device = -1;
hipGetDevice(&device);
float *a = (float *) malloc(n * sizeof(float));
hipMemAdvise(a, n * sizeof(float), hipMemAdviseSetCoarseGrain, device);
```

Notes

- hipMemAdvise() currently operates at a page granularity.
- More details/docs on hipMemAdvise() are needed.

Table for HSA_XNACK=0

	malloc()	hipMallocManaged()	hipMalloc()	hipHostMalloc()
CPU Access	In place, local	In place, local *	In place, remote	In place, local
GPU Access	Seg fault	In place, remote *	In place, local	In place, remote
Automatic Migration To GPU	No	No	No	No
Support for hipMemPrefetchAsync	No	Yes	No	No
Support hipMemAdvise	Yes	Yes	No	No
Default granularity	N/A	Fine	Coarse	Fine

* Current behavior may be different from future behavior

Table for HSA_XNACK=1

	malloc()	hipMallocManaged()	hipMalloc()	hipHostMalloc()
CPU Access	May migrate	Migrates	In place, remote	In place, local
GPU Access	May migrate	Migrates	In place, local	In place, remote
Automatic Migration To GPU	Yes	Yes	No	No
Support for hipMemPrefetchAsync	Yes	Yes	No	No
Support hipMemAdvise	Yes	Yes	No	No
Default granularity	Fine	Fine	Coarse	Fine

Managed Memory Example – Original Code

- git clone https://github.com/ROCm-Developer-Tools/HIP-Examples.git
- `cd HIP-Examples/vectorAdd`
- Load ROCm™ – `module load rocm`
- Compile and run
- `make vectoradd_hip.exe`
- `./vectoradd_hip.exe`
- Should run and report PASSED!

```
float *hostA, *hostB, *hostC;
float *deviceA, *deviceB, *deviceC;

int i, errors;

hostA = (float*)malloc(NUM * sizeof(float));
hostB = (float*)malloc(NUM * sizeof(float));
hostC = (float*)malloc(NUM * sizeof(float));

// initialize the input data
for (i = 0; i < NUM; i++) {
  hostB[i] = (float)i;
  hostC[i] = (float)i*100.0f;
}

HIP_ASSERT(hipMalloc((void**)&deviceA, NUM * sizeof(float)));
HIP_ASSERT(hipMalloc((void**)&deviceB, NUM * sizeof(float)));
HIP_ASSERT(hipMalloc((void**)&deviceC, NUM * sizeof(float)));

HIP_ASSERT(hipMemcpy(deviceB, hostB, NUM*sizeof(float), hipMemcpyHostToDevice));
HIP_ASSERT(hipMemcpy(deviceC, hostC, NUM*sizeof(float), hipMemcpyHostToDevice));


hipLaunchKernelGGL(vectoradd_float,
                dim3(WIDTH/THREADS_PER_BLOCK_X, HEIGHT/THREADS_PER_BLOCK_Y),
                dim3(THREADS_PER_BLOCK_X, THREADS_PER_BLOCK_Y),
                0, 0,
                deviceA ,deviceB ,deviceC ,WIDTH ,HEIGHT);


HIP_ASSERT(hipMemcpy(hostA, deviceA, NUM*sizeof(float), hipMemcpyDeviceToHost));
```

Managed Memory Example

- Now let's modify the memory allocation for managed memory
- Globally change all “host” strings to “vector”
- Globally change all “device” strings to “vector”
- Remove duplicate float declarations
- Move both allocations above initialization loop
- Comment out all hip data copies from host to device and device to host
- Add hipDeviceSynchronize(); after the kernel launch
- First experiment: comment out the hipMalloc/hipFrees
  - Test should fail with an
  - Memory access fault
- Set `export HSA_XNACK=1`
  - Rerun and test should pass

```
vectorA = (float*)malloc(NUM * sizeof(float));
vectorB = (float*)malloc(NUM * sizeof(float));
vectorC = (float*)malloc(NUM * sizeof(float));

//HIP_ASSERT(hipMalloc((void**)&vectorA, NUM * sizeof(float)));
//HIP_ASSERT(hipMalloc((void**)&vectorB, NUM * sizeof(float)));
//HIP_ASSERT(hipMalloc((void**)&vectorC, NUM * sizeof(float)));

// initialize the input data
for (i = 0; i < NUM; i++) {
  vectorB[i] = (float)i;
  vectorC[i] = (float)i*100.0f;
}

//HIP_ASSERT(hipMemcpy(vectorB, vectorB, NUM*sizeof(float), hipMemcpyHostToDevice));
//HIP_ASSERT(hipMemcpy(vectorC, vectorC, NUM*sizeof(float), hipMemcpyHostToDevice));

hipLaunchKernelGGL(vectoradd_float,
                dim3(WIDTH/THREADS_PER_BLOCK_X, HEIGHT/THREADS_PER_BLOCK_Y),
                dim3(THREADS_PER_BLOCK_X, THREADS_PER_BLOCK_Y),
                0, 0,
                vectorA ,vectorB ,vectorC ,WIDTH ,HEIGHT);

hipDeviceSynchronize();

//HIP_ASSERT(hipMemcpy(vectorA, vectorA, NUM*sizeof(float), hipMemcpyDeviceToHost));

// verify the results
errors = 0;
for (i = 0; i < NUM; i++) {
  if (vectorA[i] != (vectorB[i] + vectorC[i])) {
    errors++;
  }
}
if (errors!=0) {
  printf("FAILED: %d errors\n",errors);
} else {
    printf ("PASSED!\n");
}

//HIP_ASSERT(hipFree(vectorA));
//HIP_ASSERT(hipFree(vectorB));
//HIP_ASSERT(hipFree(vectorC));

free(vectorA);
free(vectorB);
free(vectorC);
```

Managed Memory Example

- Second experiment: comment out the malloc/frees instead and unset the HSA_XNACK variable or set it to 0
  - Test should pass

```
//vectorA = (float*)malloc(NUM * sizeof(float));
//vectorB = (float*)malloc(NUM * sizeof(float));
//vectorC = (float*)malloc(NUM * sizeof(float));

HIP_ASSERT(hipMalloc((void**)&vectorA, NUM * sizeof(float)));
HIP_ASSERT(hipMalloc((void**)&vectorB, NUM * sizeof(float)));
HIP_ASSERT(hipMalloc((void**)&vectorC, NUM * sizeof(float)));

// initialize the input data
for (i = 0; i < NUM; i++) {
  vectorB[i] = (float)i;
  vectorC[i] = (float)i*100.0f;
}

//HIP_ASSERT(hipMemcpy(vectorB, vectorB, NUM*sizeof(float), hipMemcpyHostToDevice));
//HIP_ASSERT(hipMemcpy(vectorC, vectorC, NUM*sizeof(float), hipMemcpyHostToDevice));

hipLaunchKernelGGL(vectoradd_float,
                dim3(WIDTH/THREADS_PER_BLOCK_X, HEIGHT/THREADS_PER_BLOCK_Y),
                dim3(THREADS_PER_BLOCK_X, THREADS_PER_BLOCK_Y),
                0, 0,
                vectorA ,vectorB ,vectorC ,WIDTH ,HEIGHT);

hipDeviceSynchronize();

//HIP_ASSERT(hipMemcpy(vectorA, vectorA, NUM*sizeof(float), hipMemcpyDeviceToHost));

// verify the results
errors = 0;
for (i = 0; i < NUM; i++) {
  if (vectorA[i] != (vectorB[i] + vectorC[i])) {
    errors++;
  }
}
if (errors!=0) {
  printf("FAILED: %d errors\n",errors);
} else {
    printf ("PASSED!\n");
}

HIP_ASSERT(hipFree(vectorA));
HIP_ASSERT(hipFree(vectorB));
HIP_ASSERT(hipFree(vectorC));

//free(vectorA);
//free(vectorB);
//free(vectorC);
```

Managed Memory Example

- Third experiment: Change hipMalloc to hipMallocManaged
  - Test should pass

```
float *vectorA, *vectorB, *vectorC;

int i, errors;

HIP_ASSERT(hipMallocManaged((void**)&vectorA, NUM * sizeof(float)));
HIP_ASSERT(hipMallocManaged((void**)&vectorB, NUM * sizeof(float)));
HIP_ASSERT(hipMallocManaged((void**)&vectorC, NUM * sizeof(float)));

// initialize the input data
for (i = 0; i < NUM; i++) {
  vectorB[i] = (float)i;
  vectorC[i] = (float)i*100.0f;
}

hipLaunchKernelGGL(vectoradd_float,
                dim3(WIDTH/THREADS_PER_BLOCK_X, HEIGHT/THREADS_PER_BLOCK_Y),
                dim3(THREADS_PER_BLOCK_X, THREADS_PER_BLOCK_Y),
                0, 0,
                vectorA ,vectorB ,vectorC ,WIDTH ,HEIGHT);

hipDeviceSynchronize();

// verify the results
errors = 0;
for (i = 0; i < NUM; i++) {
  if (vectorA[i] != (vectorB[i] + vectorC[i])) {
    errors++;
  }
}
if (errors!=0) {
  printf("FAILED: %d errors\n",errors);
} else {
    printf ("PASSED!\n");
}

HIP_ASSERT(hipFree(vectorA));
HIP_ASSERT(hipFree(vectorB));
HIP_ASSERT(hipFree(vectorC));
```

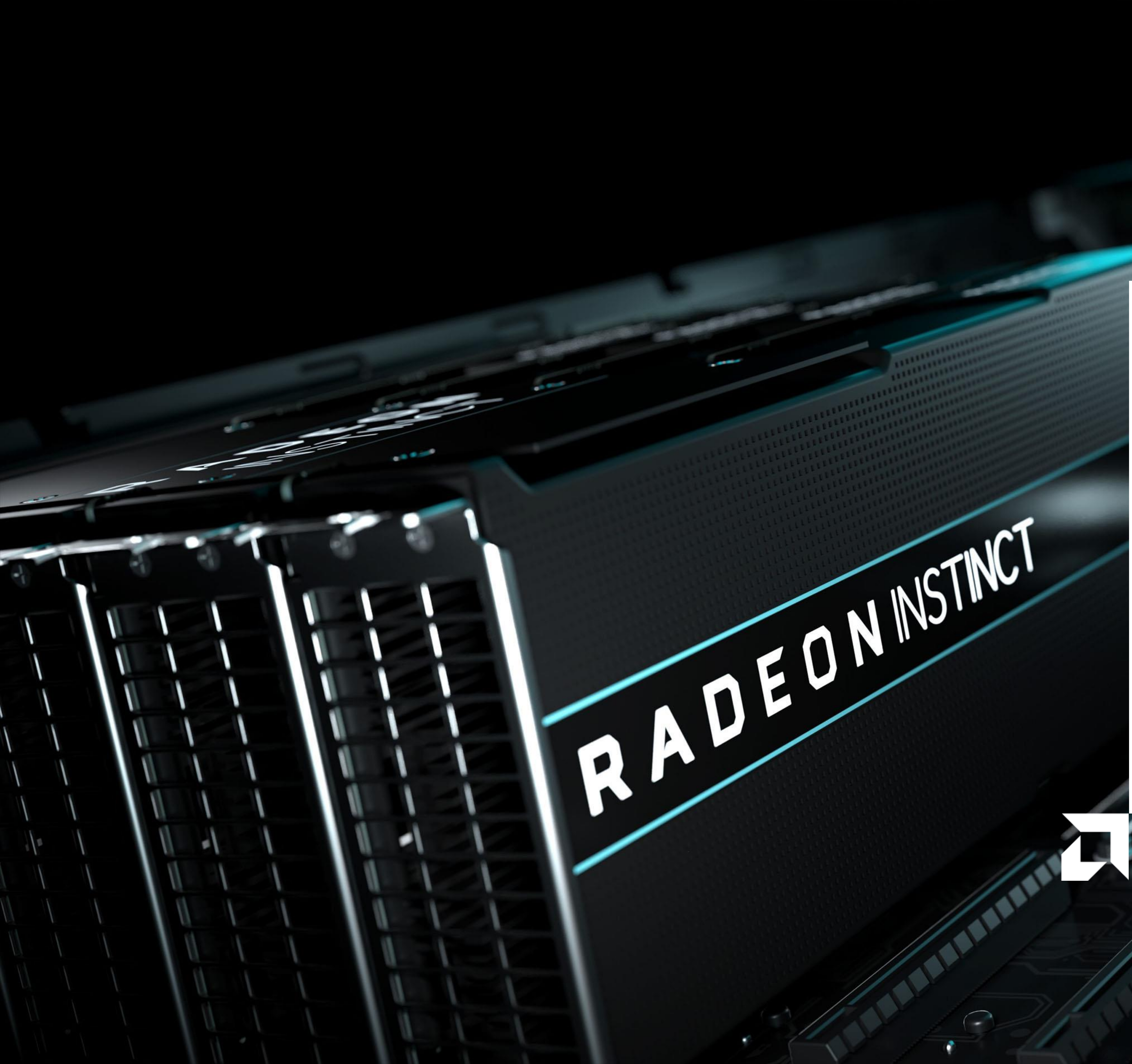

AMD together we advance_

Recommendations

- Unified or Managed Memory can be very helpful in the initial porting of an application
- Explicit memory management may be preferable for:
  - Portability to systems without unified memory support
  - Performance might be slightly better
- For page migration, hipMallocManaged() provides the best performance.
- malloc() provides support for page migration, but alignment may impact the performance and ability to migrate pages.
- hipHostMalloc() and hipMalloc() memory can be accessed by CPU and GPU, respectively, but pages will not migrate.

- More data is needed on performance implications on xnack +/- in real applications

Coarse/Fine grain memory



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.

In HIP there are currently four main allocators:

- hipHostMalloc() returns fine grain memory by default
- hipMalloc() **always** returns coarse grain memory
- hipMallocManaged() returns fine grain memory by default
- malloc(), “new” returns fine grain memory by default

hipMallocManaged() and malloc()/new will be discussed later in more details

Coarse/Fine Grain Memory Allocations

- **HipMemAdvise also has options for coarse/fine grain**
- hipMemAdvise() current status:
    1) coarse/fine grain setting is working

hipMallocManaged() and malloc()/new will be discussed later in more details

Coarse/Fine Grain for hipHostMalloc

hipHostMalloc flags	Granularity	Meaning
hipHostMallocDefault	Fine grain	Memory is mapped and portable
hipHostMallocPortable	Fine grain	Registered by all contexts
hipHostMallocMapped	Fine grain	Map allocation into device
hipHostMallocWriteCombined	Fine grain	More efficient writes?
hipHostMallocNumaUser	Fine grain	Follow NUMA policy set by user
hipHostMallocCoherent	Fine grain	Set memory to be coherent
hipHostMallocNonCoherent	Coarse grain	Set memory to be non coherent

Example:

```
hipHostMalloc((void**)&ptr, (size_t)bytes, hipHostMallocDefault);
```

Coarse/Fine Grain for hipMallocManaged() and malloc()/new

API	Flag/memAdvise	Result
hipMallocManaged()	Default	Fine grain
hipMallocManaged()	Default + hipMemAdvise - hipMemAdviseSetCoarseGrain	Coarse grain
Malloc()/new		Fine grain
Malloc()/new	hipMemAdvise- hipMemAdviseSetCoarseGrain	Coarse grain

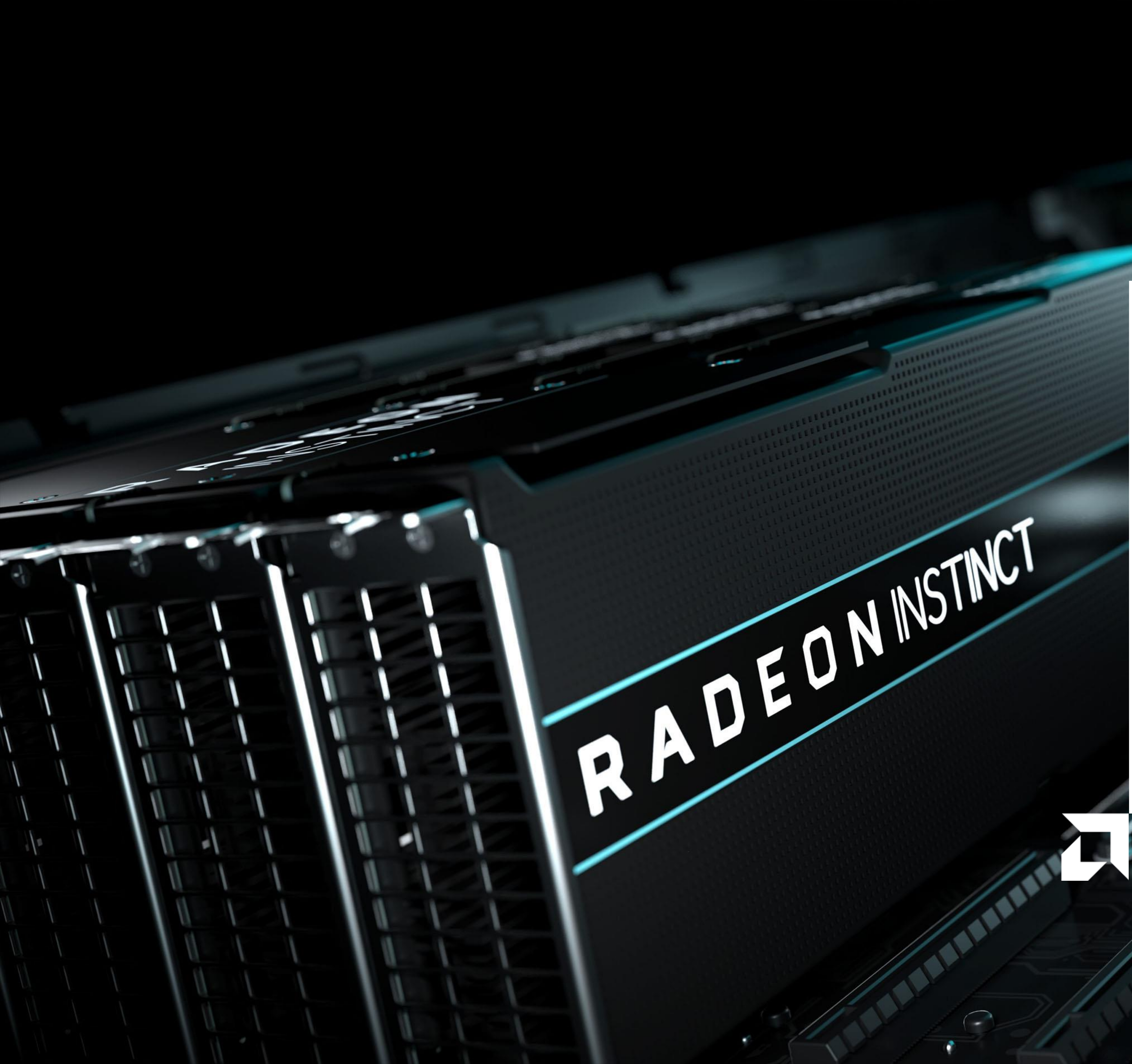
Example:

```
float *a;
hipMallocManaged(&a, n * sizeof(float));
hipMemAdvise(a, n * sizeof(float), hipMemAdviseSetCoarseGrain, device);
```

Conclusions

- More data is needed on performance implications of fine vs. coarse grain memory in real applications
- Future work will show more examples of advanced synchronization patterns with fine grain memory

Floating Point Atomics on fine/coarse grained memory



Atomics on Coarse/Fine Grain Regions

MI250X provides a set of HW atomics for INT and FP (e.g. atomicAdd()).

HW-accelerated FP atomics applied to fine grain memory regions will silently fail.

By default, the compiler replaces all language level FP atomics with CAS loops regardless of the granularity of the memory region.

FP Atomics based on a CAS loop are slower than HW FP Atomics.

The flag *-munsafe-fp-atomics* **currently** suggests to the compiler to generate HW FP atomic instructions.

HIP exposes the *unsafeAtomicAdd()* function to **always** emit HW FP Atomics.

From ROCm-5.2.0 the use of the *-munsafe-fp-atomics* flag will **enforce** HW FP atomics.

The word “unsafe” refers to the fact that HW FP atomics performed on fine grain memory will fail.

Using “unsafe” atomics is perfectly safe when used on coarse grain memory.

Integer atomics will always be based on HW Atomics.

OpenMP® equivalent covered later

Summary Atomics on Coarse/Fine Grain Regions

Compiler flag	atomicAdd	unsafeAtomicAdd
-mno-unsafe-fp-atomics / default	CAS loop	HW FP Atomics
-munsafe-fp-atomics	CAS loop / HW FP Atomics	HW FP Atomics
-munsafe-fp-atomics (ROCm >= 5.2)	HW FP Atomics	HW FP Atomics

Compiler flag	Fine grained	Coarse grained
-munsafe-fp-atomics	Incorrect results	Correct results, fast
-mno-unsafe-fp-atomics / default	Correct results, slow	Correct results, slow

Performance Impact in an Application

- Extreme Test case for a single kernel with multiple atomicadds
- CAS is a compare and swap operation with multiple lines of code
- Performance for each is shown relative to the coarse-grain memory with hardware atomics
- Fine-grain memory is faster, but it gives an incorrect answer (expected)

WRONG Answer
(expected for fine-grain allocation and hardware atomic)

	Coarse-grain	Fine-grain
Hardware atomics	1	.93
CAS loop	39	1678

Notes:
- MI250x
- Managed memory turned off
- Single precision

Conclusions

- **From ROCm-5.2.0 onwards** the use of the *-munsafe-fp-atomics* flag will **enforce** HW FP atomics.
- *-munsafe-fp-atomics* is safe on memory allocated using hipMalloc(). Check various tables for all other cases.
- HIP exposes the *unsafeAtomicAdd()* function to **always** emit HW FP Atomics.
- Fast atomic operations for datatypes other than FP do not require the *-munsafe-fp-atomics* flag.

ROCm™ OpenMP®

Shared Memory and Floating
Point Atomics

For this exploration of OpenMP® memory behavior

For this example, we use the AMD Clang OpenMP compiler

- `Module load aomp rocm (or amdclang)`
- `--offload-arch=$(ROCM_GPU)` or `--offload-arch=gfx90a`

  The xnack setting can also be set in your environment

- `export HSA_XNACK=1`

- === To check what is happening under the covers

- `export LIBOMPTARGET_KERNEL_TRACE=1`
- `export LIBOMPTARGET_INFO=$((0x20 | 0x02 | 0x01 | 0x10))`

Default and Unified Shared Memory Modes – arraysum1

Default Mode

```
int main(int argc, char *argv[]) {
   int errors=0, n=10000;
   double *a = (double*)malloc(n*sizeof(double));
   double *b = (double*)malloc(n*sizeof(double));
   for (int i = 0; i < n; i++){
     a[I] = 0.0;
     b[I] = 1.0;
   }
   #pragma omp target teams distribute parallel for map(tofrom: a[:n]) map(to: b[:n])
   for(int i = 0; i < n; i++){
     a[i] += b[i];
   }
}
```

- Device (global) memory allocation
- Host-to-Device and Device-to-Host memory copy
- Default for memory allocated by "map" is **coarse grain**

OpenMP

Default and Unified Shared Memory Modes – arraysum2

Unified Shared Memory Mode

```
#pragma omp requires unified_shared_memory
int main(int argc, char *argv[]) {
   int errors=0, n=10000;
   double *a = (double*)malloc(n*sizeof(double));
   double *b = (double*)malloc(n*sizeof(double));
   for (int i = 0; i < n; i++){
     a[I] = 0.0;
     b[I] = 1.0;
   }
   #pragma omp target teams distribute parallel for map(tofrom: a[:n]) map(to: b[:n])
   for(int i = 0; i < n; i++){
     a[i] += b[i];
   }
}
```

- Maps are not required
- OpenMP® runtime does not issue memory allocation and memory copy requests
- OS allocator returns fine grain memory pointer

Default and Unified Shared Memory Modes – arraysum3

Unified Shared Memory Mode

```
#pragma omp requires unified_shared_memory
int main(int argc, char *argv[]) {
   int errors=0, n=10000;
   double *a = (double*)malloc(n*sizeof(double));
     double *b = (double*)malloc(n*sizeof(double));
     for (int i = 0; i < n; i++){
       a[I] = 0.0;
       b[I] = 1.0;
     }
   #pragma omp target teams distribute parallel for map(tofrom: a[:n]) map(to: b[:n])
   for(int i = 0; i < n; i++){
     a[i] += b[i];
   }
}
```

- If maps are used, pages used by a and b **switch to coarse grain**
- OpenMP® runtime still does not issue device memory allocation, nor memory copies

OpenMP

Features and Limitations – no test example

Coarse grain scope

```
#pragma omp requires unified_shared_memory
int main(int argc, char *argv[]) {
   int errors=0, n=10000;
   double *a = (double*)malloc(n*sizeof(double));
   double *b = (double*)malloc(n*sizeof(double));
   for (int i = 0; i < n; i++){
     a[I] = 0.0;
     b[I] = 1.0;
   }
   #pragma omp target teams distribute parallel for map(tofrom: a[:n]) map(to: b[:n])
   for(int i = 0; i < n; i++) a[i] += b[i];

   #pragma omp target teams distribute parallel for
   for(int i = 0; i < n; i++)
      a[i] += 1.0;
}
```

- a and b **remain coarse grain for the remainder of their life**
- No need to map them again to make them coarse grain

Features and Limitations – arraysum4

Coarse Grain is Sticky

```
#pragma omp requires unified_shared_memory
int main(int argc, char *argv[]) {
   int errors=0, n=10000;
   double *a = (double*)malloc(n*sizeof(double));
   double *b = (double*)malloc(n*sizeof(double));
   init(a,b);

   #pragma omp target teams distribute parallel for
   for(int i = 0; i < n; i++)
     a[i] += b[i];
}
```

```
void init(double *a, double *b) {
  #pragma omp target map(from:a[:n],b[:n])
  for(int i = 0; i < n; i++){
    a[i] = b[i] = 1.0;
  }
}
```

a and b **change to coarse grain**

- a and b **are not mapped here**
- They are coarse grain due to previous map

Details

OpenMP® Allocation Mechanism	Behavior
“map” clause default mode	**device allocation + host/device transfer**
“map” clause in unified_shared_memory mode	memory pages switch to coarse grain permanently upon map
omp_target_alloc (both modes)	device allocation (coarse grain)
Allocator with pinned trait set	memory lock

Fast Floating Point Atomics – arraysum8

Test case will fail because *a* and *ret* are fine grain memory

```
#include <omp.h>

#pragma omp target teams distribute parallel for reduction(+:ret)
for(int i = 0; i < n; i++) {
  #pragma omp atomic hint(AMD_fast_fp_atomics)
  ret += b[i];
}
```

Fails with ROCm 5.6 with compiler error

Force compiler to use fast FP atomics

There are two ways this can be fixed. They are shown on the following two slides.

Fast Floating Point Atomics – arraysum9

```
#include <omp.h>

#pragma omp target teams distribute parallel for map(to: b[:n]) map(tofrom: ret)
for(int i = 0; i < n; i++) {
  #pragma omp atomic hint(AMD_fast_fp_atomics)
  ret += b[i];
}
```

a and `ret` switched to coarse grain

fails with reduction clause (5.4.3)
fails with 5.6.0

Force compiler to use fast FP atomics

Fast Floating Point Atomics – arraysum10

```
#include <omp.h>

#pragma omp target teams distribute parallel for reduction(+:ret)
for(int i = 0; i < n; i++) {
  #pragma omp atomic hint(AMD_safe_fp_atomics)
  ret += a[i];
}
```

fails with ROCm 5.6.0

Force compiler to use safe FP atomics (Compare-And-Swap)

Fast Floating Point Atomic with arrays – arraysum5-7

- The test cases for arraysum5.c to array7.c are similar to the previous example, but with arrays.
- These cases are working and demonstrate the same methods for fast floating point atomics.
- arraysum5 will fail, arraysum6 and arraysum7 will pass with the fixes for fast floating point atomics

Fast Floating Point Atomics

Hint Clause Value

Compiler Option

	none	AMD_fast_fp_atomics	AMD_safe_fp_atomic s
none	CAS-loop	Fast FP Atomics	CAS-loop
-munsafe-fp-atomics	Fast FP atomics	Fast FP Atomics	CAS-loop
-mno-unsafe-fp-atomics	CAS-loop	Fast FP Atomics	CAS-loop

Conclusions and Cautionary Statements

OpenMP®

- Some of the OpenMP pragmas and behaviors are specific to AMD. The extensions to OpenMP may change as more experience is gained with the more advanced hardware in AMD GPUs.
- Portability of OpenMP pragmas extensions are not guaranteed or even likely even among OpenMP compilers for AMD GPUs.

Memory Model in general

- Some of the behavior of managed memory, coarse/fine grain memory, and atomics are still under investigation for best implementation in the ROCm™ software and compilers. Compiler flags, environment variables, and pragmas might change in future releases.

Disclaimer

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

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

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