



# Advanced HIP

**Presenter: Bob Robey**  
**AMD @ CASTIEL HPC**  
**Oct 28-30, 2025**

**AMD**   
together we advance\_

---

# Agenda

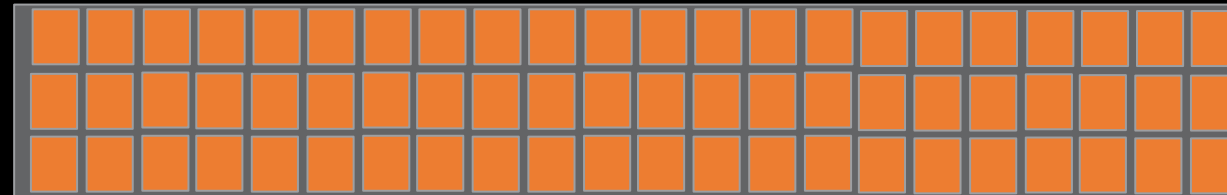
- 
1. Overview of Kernel Performance Limiters
  2. How to optimize memory bound kernels
  3. How to optimize compute bound kernels
  4. How to optimize latency bound kernels
  5. Considerations for MI300A APU architecture

# GPUs are high throughput devices

- Must expose parallelism to properly utilize them



GPU starvation – under-utilization of resources



Full utilization of resources

# Optimization strategy depends on performance limiters

## Memory bound

- Low arithmetic intensity, memory units saturated

## Compute bound

- High arithmetic intensity, compute units saturated
  - Typical machine balance: 5-10 FLOPs/B
    - 40-80 FLOPs per double to exploit compute capability
  - MI250x machine balance: ~16 FLOPs/B
    - 128 FLOPs per double to exploit compute capability
  - Difficult to get compute bound on current architectures – concept of flops are for free

$$\text{Arithmetic Intensity} = \frac{\text{Arithmetic Operations}}{\text{Bytes moved}}$$

## Latency bound

- Memory units not saturated and/or compute units not saturated

Focus of this presentation – what to do for these different types of kernels?

# Background – What is Roofline

• Attainable FLOPs/s =

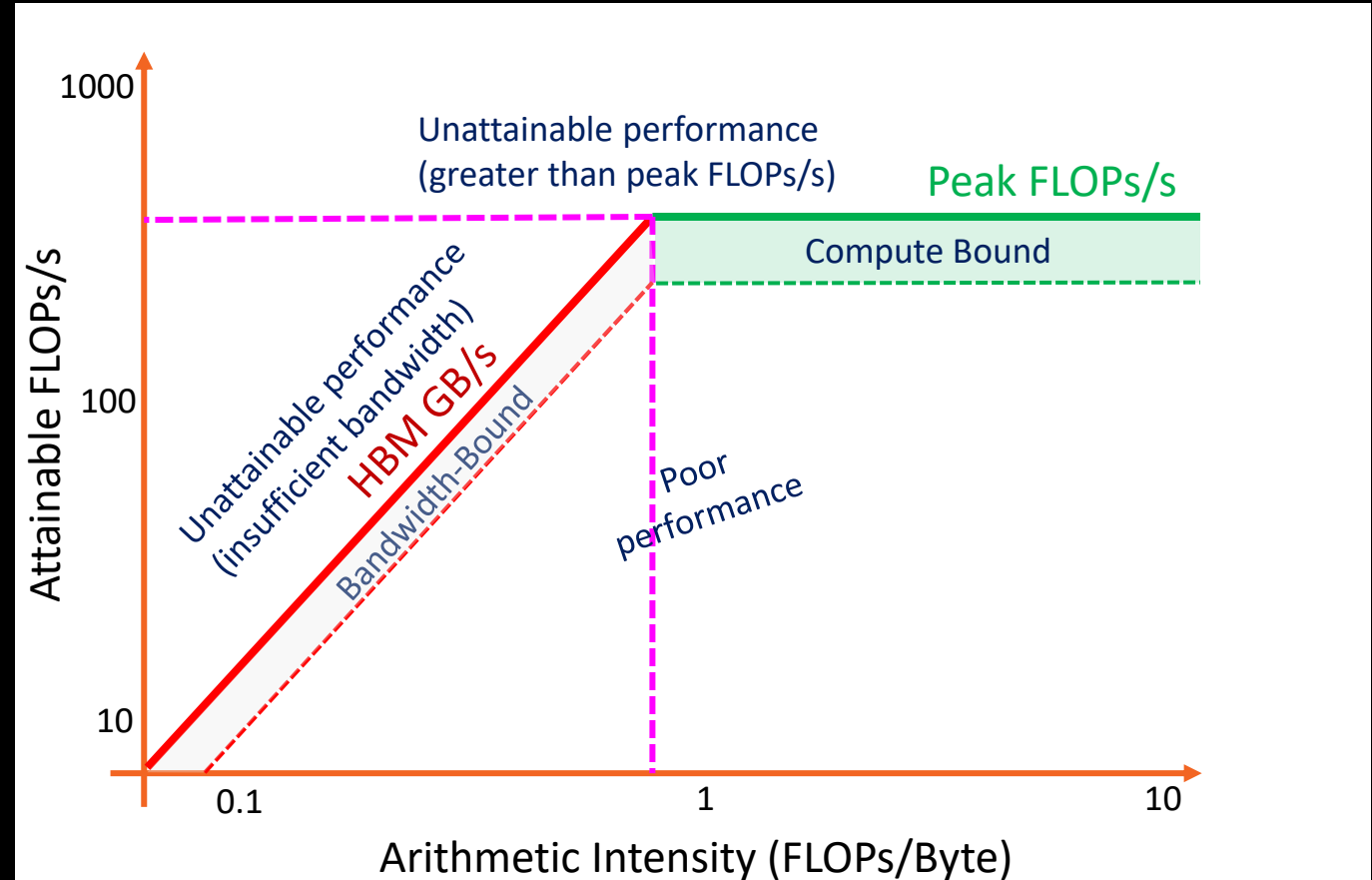
$$\min \left\{ \begin{array}{l} \text{Peak FLOPs/s} \\ AI * \text{Peak GB/s} \end{array} \right.$$

• Machine Balance:

• Where  $AI = \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}}$

• Five Performance Regions:

- Unattainable Compute
- Unattainable Bandwidth
- Compute Bound
- Bandwidth Bound
- Poor Performance



# Memory Bandwidth Bound

# Data Movement

- Reducing data movement is still very important for GPU performance
  - Move data, compute as much as possible with that data
- Reuse data when possible – temporal reuse and spatial reuse
- Stage data in shared memory (LDS) or registers for faster access
- Lower precision data types move fewer bytes, evaluate their use for your algorithm
- Move more data per work-item to improve streaming efficiency

# Data Access Considerations

- Coalesced loads/stores improve achieved bandwidth of transfers
  - L1 cacheline size is 64 bytes in MI200 GPUs, and 128 bytes in MI300 GPUs
  - Use as much as possible of each cacheline read
  - Strided accesses may load more data than needed (more details in rocprof-compute by Examples talk)
- Use vector data types such as float4, float2
  - Compiler generates fewer, wider load instructions
  - Amortize on cost of address/index calculations
  - Improve data streaming efficiency
- Use non-temporal loads for data that will not be reused
- Aligned memory accesses avoid excess data from being fetched

# Sometimes compiler generates wider loads/stores for free

```
5  __global__ void add2(const int N,  
6      float *__restrict__ x,  
7      float *__restrict__ y) {  
8  
9      int n = threadIdx.x + blockDim.x * blockIdx.x;  
10     y[2*n+0] = x[2*n+0];  
11     y[2*n+1] = x[2*n+1]; Each work item loads and stores two elements  
12 }  
13  
14 __global__ void add1(const int N,  
15     float *__restrict__ x,  
16     float *__restrict__ y) {  
17  
18     int n = threadIdx.x + blockDim.x * blockIdx.x;  
19     y[n] = x[n];  
20 }
```

```
1  add2(int, float*, float*):                               ; @add2(int, float*, float*)  
2      s_load_dword s3, s[0:1], 0x24  
3      s_load_dwordx4 s[4:7], s[0:1], 0x8  
4      s_waitcnt lgkmcnt(0)  
5      s_and_b32 s0, s3, 0xffff  
6      s_mul_i32 s2, s2, s0  
7      v_add_lshl_u32 v0, s2, v0, 1  
8      v_ashrrev_i32_e32 v1, 31, v0  
9      v_lshlrev_b64 v[0:1], 2, v[0:1]  
10     v_lshl_add_u64 v[2:3], s[4:5], 0, v[0:1]  
11     global_load_dwordx2 v[2:3], v[2:3], off  
12     v_lshl_add_u64 v[0:1], s[6:7], 0, v[0:1]  
13     s_waitcnt vmcnt(0)  
14     global_store_dwordx2 v[0:1], v[2:3], off  
15     s_endpgm  
16 add1(int, float*, float*):                               ; @add1(int, float*, float*)  
17     s_load_dword s3, s[0:1], 0x24  
18     s_load_dwordx4 s[4:7], s[0:1], 0x8  
19     s_waitcnt lgkmcnt(0)  
20     s_and_b32 s0, s3, 0xffff  
21     s_mul_i32 s2, s2, s0  
22     v_add_u32_e32 v0, s2, v0  
23     v_ashrrev_i32_e32 v1, 31, v0  
24     v_lshlrev_b64 v[0:1], 2, v[0:1]  
25     v_lshl_add_u64 v[2:3], s[4:5], 0, v[0:1]  
26     global_load_dword v2, v[2:3], off  
27     v_lshl_add_u64 v[0:1], s[6:7], 0, v[0:1]  
28     s_waitcnt vmcnt(0)  
29     global_store_dword v[0:1], v2, off  
30     s_endpgm
```

Wider load/store instruction

<https://godbolt.org/z/WYzMjxKzr>

# Compute Bound

# Compute Optimizations

- Compute bound kernels perform  $O(100)$  operations per byte loaded
  - Large GEMMs are an example of compute bound kernels, but HPC workloads are typically memory bound
- Pre-compute values to look up in kernel
- Use faster math intrinsic functions, e.g., `__cosf(x)` instead of `cosf(x)`
  - More details: [https://rocm.docs.amd.com/projects/HIP/en/latest/reference/math\\_api.html](https://rocm.docs.amd.com/projects/HIP/en/latest/reference/math_api.html)
- Avoid general math functions where possible
  - `a * a * a` uses two instructions whereas `pow(a, 3.0f)` uses many
  - Godbolt link: <https://godbolt.org/z/8hz8P4oc9>
- Explore use of packed FP32 operations that process two FP32 values in one instruction
  - For example, using `float2` instead of `float` can result in the use of packed instructions

# Compute Optimizations (contd.)

- Where you stage data for your compute matters
  - To make your kernel truly compute-bound, read from registers
  - Moving data from shared memory and/or cache takes  $O(10)$  cycles
- For specific matrix multiplication like calculations, special hardware units exist (rocWMMA)
  - AMD Matrix Cores ROCm Blog: <https://rocm.blogs.amd.com/software-tools-optimization/matrix-cores/README.html>

# Unexpected Instructions

```
__global__ void conversions (float *a) {
    float f1 = a[threadIdx.x] * 0.3;
    float f2 = 2.0 * (f1 * 3.0);
    a[threadIdx.x] = f1 + f2;
}
```

```
__global__ void no_conversions (float *a) {
    float f1 = a[threadIdx.x] * 0.3f;
    float f2 = 2.0f * (f1 * 3.0f);
    a[threadIdx.x] = f1 + f2;
}
```

```
s_load_dwordx2 s[0:1], s[4:5], 0x0
v_lshlrev_b32_e32 v4, 2, v0
s_mov_b32 s2, 0x33333333
s_mov_b32 s3, 0x3fd33333
s_waitcnt lgkmcnt(0)
global_load_dword v0, v4, s[0:1]
s_waitcnt vmcnt(0)
v_cvt_f64_f32_e32 v[0:1], v0
v_mul_f64 v[0:1], v[0:1], s[2:3]
v_cvt_f32_f64_e32 v5, v[0:1]
s_mov_b32 s2, 0
v_cvt_f64_f32_e32 v[0:1], v5
s_mov_b32 s3, 0x40080000
v_mul_f64 v[2:3], v[0:1], s[2:3]
v_fmacc_f64_e32 v[2:3], s[2:3], v[0:1]
v_cvt_f32_f64_e32 v0, v[2:3]
v_add_f32_e32 v0, v0, v0
global_store_dword v4, v0, s[0:1]
s_endpgm
```

```
s_load_dwordx2 s[0:1], s[4:5], 0x0
v_lshlrev_b32_e32 v0, 2, v0
s_waitcnt lgkmcnt(0)
global_load_dword v1, v0, s[0:1]
s_waitcnt vmcnt(0)
v_mul_f32_e32 v1, 0x3e99999a, v1
v_mul_f32_e32 v2, 0x40400000, v1
v_fmacc_f32_e32 v1, 2.0, v2
global_store_dword v0, v1, s[0:1]
s_endpgm
```

Fewer instructions, FP32 ops

Wait, what?!

# Latency bound

# Main Ideas for Optimizing Latency Bound Kernels

- Increase parallelism to utilize all GPU resources
- Reduce number of synchronization barriers
- Reduce thread divergence
- Avoid register spilling to slower "scratch" memory

# Motivation for Launching Many Wavefronts

- The GPU has a lot of resources
  - Wavefronts can stall for various reasons:
    - Waiting for data to load
    - Waiting at a synchronization barrier
  - GPU is good at switching to wavefronts with instructions ready to be executed
- Good to launch a lot of wavefronts and hide latencies of stalls

# What is Occupancy?

- # Resident wavefronts / Maximum #wavefronts the GPU can have in-flight
- Hardware Perspective (let's consider a MI210 GPU):
  - There are 104 Compute Units (CU)
  - Up to 32 wavefronts can be scheduled to each CU = max 3328 wavefronts
- Developers' Perspective:
  - Am I launching enough units of work to use all CUs?
  - Am I launching more wavefronts than the number of CUs to hide latencies?
- Higher occupancy can help improve performance, but not always

# Occupancy by Example (daxpy)

$Z = aX + Y$  where  $Z$ ,  $X$  and  $Y$  are 1D arrays of length  $N = 1,000,000$  elements

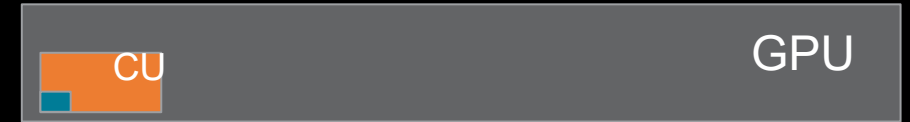
We know that

- a workgroup can have 64 to 1024 work-items = 1 to 16 wavefronts
- all wavefronts of a workgroup will be scheduled to the same CU

We can launch the daxpy kernel in many ways:

1 workgroup with 64 work-items

Only 1 wave on 1 CU = No latency hiding



1 workgroup with 256 work-items

Only 4 waves on 1 CU = All other CUs idle



$N/1024$  workgroups, each workgroup has 16 waves

$\sim 1000$  workgroups =  $\sim 16000$  waves = good occupancy

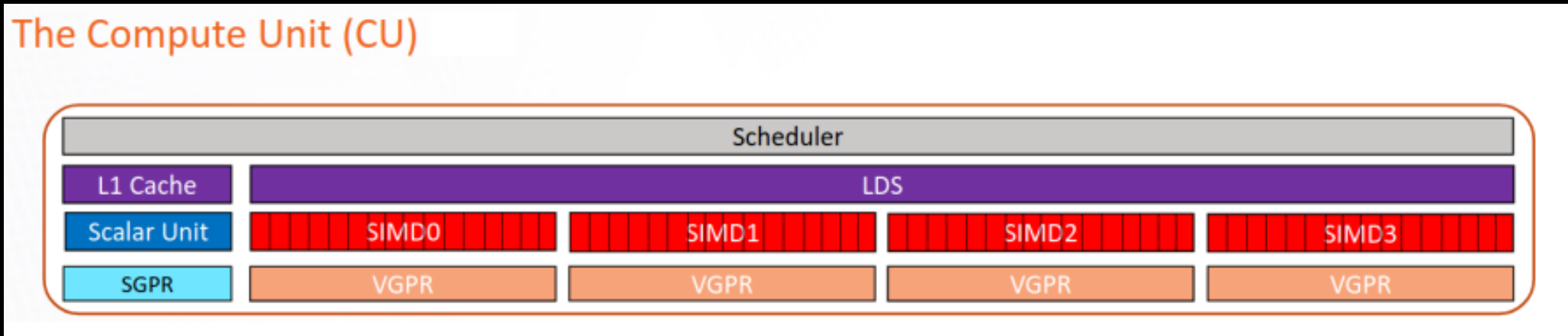


But that's not the whole picture..

# Memory Resources that affect Occupancy

Compute Units have finite resources that are shared between work items

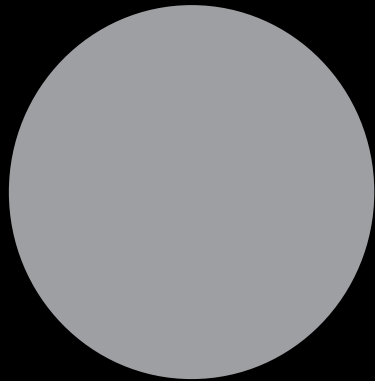
- Local Data Share (LDS)
- Vector General Purpose Registers (VGPRs)
- Scalar General Purpose Registers (SGPRs)



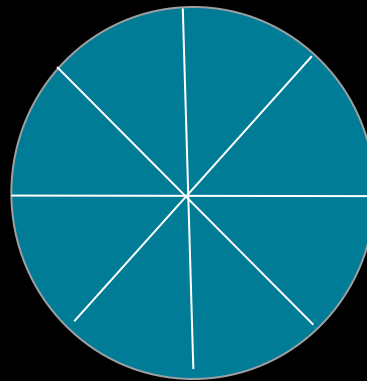
The GPU can only schedule more work if there are enough resources available

# How LDS affects Occupancy

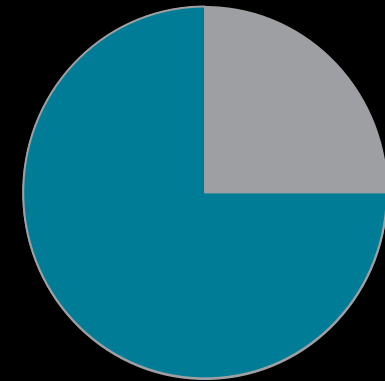
- Fast, on-CU, software managed memory to efficiently share data between work-items of a workgroup
- Each CU in a MI200 GPU has 64 KiB of LDS available
- Shared among workgroups on CU



No LDS used, LDS does not limit occupancy



8 KiB of LDS per WG, 8 WGs can fit in CU



48 KiB of LDS per WG, only 1 WG can fit in CU

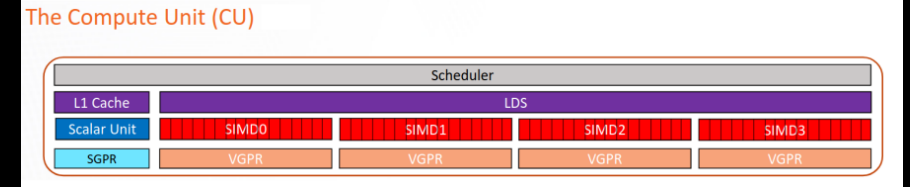
# What is Register Pressure?

- Register pressure is a commonly used slang term for the demand in GPU kernel code for both vector and scalar registers
  - Vector registers are often referred to as VGPRs, which stands for Vector General Purpose Registers
    - These registers have an array of values with one value for each thread in a wavefront
  - Scalar registers are often referred to as SGPRs, which stands for Sector General Purpose Registers
    - These registers have one value for all the threads in a wavefront
- When there is high usage of vector registers, it can have effects on program performance.
  - Reduces occupancy by limiting the number of waves that can be schedule on a GPU compute unit.
  - For very high usage, can result in “spilling” of registers to global memory and the slower access times for variables
- More generally, memory pressure and resource pressure are similar terms used to refer to the demand for other types of resources for GPU kernels.
- Register pressure also occurs on CPUs. In both CPU and GPU architectures, registers are a scarce resource that compilers use to locate variables close to the processor.

For a quick review of the GPU resources → next slides

# How Vector Registers affect Occupancy

- In VGPRs, each thread in the wavefront can save its own value
- Each MI200 CU has a limited size vector register file (max 512 VGPRs of size 4 bytes)



Num VGPRs	Occupancy per SIMD	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	1 waves	4 waves

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

# How Scalar Registers (SGPRs) affect Occupancy

- In SGPRs, one value is shared across all work-items of the wavefront
- Each MI200 CU has a limited size scalar register file (max 102 SGPRs of size 4 bytes per wavefront)

# A Note about Register Spilling

- Register allocation is done by the compiler at compilation time
- When the required number of VGPRs is too much (i.e.,  $> 256$ ), the compiler may “spill” registers to slower “scratch” memory
  - Better to avoid spilling in most cases
- By default, the compiler assumes workgroups are going to have 1024 work-items
  - Use `__launch_bounds__` on smaller workgroups to allow the compiler to use more registers
- The compiler may spill SGPRs to VGPRs, this seldom limits scheduling
  - Don't take this as a challenge

ROCm blog about Register Pressure:

<https://rocm.blogs.amd.com/software-tools-optimization/register-pressure/README.html>

# Register Pressure Hints - See [Register pressure in AMD CDNA™ 2 GPUs](#)

- Compiler for GPU kernels will generally keep data in registers
- Registers/variables in functions will persist (functions are currently in-lined)
- Vector registers greater than 256 will spill to main memory
- Reducing number of vector registers can help occupancy
- Methods to try and reduce vector registers
  - Reduce the scope where variable is live
  - Limit register usage by reducing workgroup size (`__launch_bounds__`)
  - Avoid asserts
  - Avoid mathematical intrinsics
  - Use LDS to store some variables
  - Manually reuse variables
  - Use restrict keyword
  - Avoid stack arrays and keep them as small as possible
  - Use `constexpr` when variables are constants ( or `#define` or equivalent compile time constant)
- Scalar registers – spill to vector registers
  - Generally not a performance issue
  - Avoid passing in large structs of scalar variables with only a few actually used

# Fuse kernels to reduce launch latencies

- Also reduce data movement as shown here:

One read of "a" and "b" and one write of "c"

```
__global__ void kernel1 (float *a, float *b, float *c) {  
    int32_t tid = blockIdx.x * blockDim.x + threadIdx.x;  
    c[tid] = a[tid] + 2 * b[tid];  
}  
  
__global__ void kernel2 (float *a, float *b, float *c) {  
    int32_t tid = blockIdx.x * blockDim.x + threadIdx.x;  
    c[tid] = c[tid] - a[tid] * b[tid];  
}
```

```
__global__ void kernel_fused (float *a, float *b, float *c) {  
    int32_t tid = blockIdx.x * blockDim.x + threadIdx.x;  
    float a = a[tid];  
    float b = b[tid];  
    c[tid] = a + 2 * b - a * b;  
}
```

2 reads of "a" and "b", "c" written out and read back before being written out again!

# Reduce or Avoid Synchronization

- Thread block synchronization
  - Synchronizes wavefronts in a thread block
  - Expensive in large work groups, don't over use it
- Host-side synchronization
  - Memory operations (hipMalloc, hipFree, etc.) implicitly synchronize activity on the device => unexpected low perf
  - Move memory allocations out of inner loops. This may cause a rethinking of the current algorithm
- Use asynchronous memory copies (H<->D) with pinned host buffers
  - avoid host-side synchronization
  - overlap copies with compute

# A Note about Atomics

- If using atomic operations on MI200, compile with `-munsafe-fp-atomics` to use hardware atomics on FP data in GPU memory
  - Not needed on MI300
- Reducing contention in atomic operations can improve performance
- On MI300 GPUs, atomics are performed in the AMD Infinity Cache™ instead of the L2 cache
  - Infinity Cache is a Memory Adjacent Last Level (MALL) cache
  - L2 is distributed and local to Accelerator Compute Dies (XCDs)

# Minimize Thread Divergence

- Instructions in divergent paths are executed multiple times, some threads masked off each time
- Try minimizing divergent sections even if it means values computed by some threads will be discarded eventually

```
size_t idx = threadIdx.x + blockDim.x * blockIdx.x;
if (threadIdx.x % 2 == 0) {
    out[2 * idx] = 1.0;
} else {
    out[2 * idx + 1] = 0.0;
}
```

```
size_t idx = threadIdx.x + blockDim.x * blockIdx.x;
double2 *ptr = (double2 *) (out + idx);
ptr[0] = {1.0, 0.0};
```

To compare assembly for both cases: <https://godbolt.org/z/4fEqvE8zP>

# Warp shuffle/cross-lane functions

- Exchange data in registers between threads in wavefront
- Uses the same hardware fabric as LDS, but no storage in LDS
- Works on a common “width” where every thread is using the same width up to the wavefront size of 64

# Considerations for the MI300A APU architecture

- Single allocation, zero copy
  - No page migrations, CPU and GPU share same physical memory
- Choice of allocator can affect latency of first touch
  - hipMalloc - device page tables populated, registered on CPU only on first touch
  - hipHostMalloc (or) malloc + hipHostRegister – page tables populated on both CPU and device
  - malloc – CPU page tables populated, GPU only registers them on first touch
- Page size matters
  - System allocators defaults to 4KB pages, GPU prefers 2MB pages
  - hipMalloc everything to guarantee 2MB pages
- What resources on the device do you want to use for copies?
  - SDMA engines or kernels
  - Single or multi-threaded on CPU

# Summary

- Kernel performance may be limited by
  - memory bandwidth
  - lack of compute resources
  - latencies
- Performance optimization involves balancing many constraints
  - Reduce data movement and access data in a coalesced manner
  - Avoid unnecessary compute and excessive synchronization
  - Adjust occupancy while considering resource requirements

# Hands-on exercises

Located in our HPC Training Examples repo:

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

A table of contents for the READMEs if available at the top-level README in the repo

Relevant exercises for this presentation located in:

- [HIP-Optimizations](#) directory

Link to instructions on how to run the HIPIFY tests: [HIP-Optimizations/daxpy/README.md](#)

Log into the AAC node and clone the repo:

```
ssh <username>@aac6.amd.com -p 7000 -i <path_to_ssh_key>  
git clone https://github.com/amd/HPCTrainingExamples.git
```

# Hands-on Exercises and Examples

- Try the following suggestions on the example code for blog post
- First retrieve the examples and find the directory where they are located

```
git clone https://github.com/amd/HPCTrainingExamples
cd HPCTrainingExamples/rocm-blogs-codes/registerpressure
```

- Get the compiler resource report for the kernel

```
hipcc --offload-arch=<gfxcode> lbm.cpp -Rpass-analysis=kernel-resource-usage -c
```

```
lbm.cpp:16:1: remark:      SGPRs: 100 [-Rpass-analysis=kernel-resource-usage]
lbm.cpp:16:1: remark:      VGPRs: 104 [-Rpass-analysis=kernel-resource-usage]
lbm.cpp:16:1: remark:      AGPRs: 0 [-Rpass-analysis=kernel-resource-usage]
lbm.cpp:16:1: remark:      ScratchSize [bytes/lane]: 0 [-Rpass-analysis=kernel-resource-usage]
lbm.cpp:16:1: remark:      Dynamic Stack: False [-Rpass-analysis=kernel-resource-usage]
lbm.cpp:16:1: remark:      Occupancy [waves/SIMD]: 4 [-Rpass-analysis=kernel-resource-usage]
lbm.cpp:16:1: remark:      SGPRs Spill: 0 [-Rpass-analysis=kernel-resource-usage]
lbm.cpp:16:1: remark:      VGPRs Spill: 0 [-Rpass-analysis=kernel-resource-usage]
lbm.cpp:16:1: remark:      LDS Size [bytes/block]: 0 [-Rpass-analysis=kernel-resource-usage]
```

- Note that the number of registers (SGPRs and VGPRs) are slightly different than in the blog. They will vary slightly for different compiler versions. Also note that the Occupancy is 4 waves/SIMD. We want to improve that. To accomplish that, we need to get the VGPRs down to 96 as seen from the earlier table.

# Things to try from the blog post

1. Remove unnecessary math functions
  - `pow(current_phi, 2.0)` on line 37 can be changed to `current_phi * current_phi`
  - This C function raises the argument to a floating point power in software. It is not a very efficient way to do the operation and also consumes a lot of registers.
2. Rearrange code so variables are declared close to use
3. Add restrict attribute to function arguments

## Result of optimizations

	SGPRs	VGPRs	Occupancy
lbm.cpp	100	104	4
lbm_1_nopow.cpp	88	104	4
lbm_2_rearrange.cpp	100	104	4
lbm_3_restrict.cpp	84	100	4

In this case, we have not been able to get the VGPRs to 96 or below. Try adding restrict to more arguments and see if you can.

# Extra credit

- Add `__launch_bounds__` (256) to function attributes
  - Drops SGPRs to 84 and VGPRs to 96. Occupancy jumps to 5 waves per SIMD

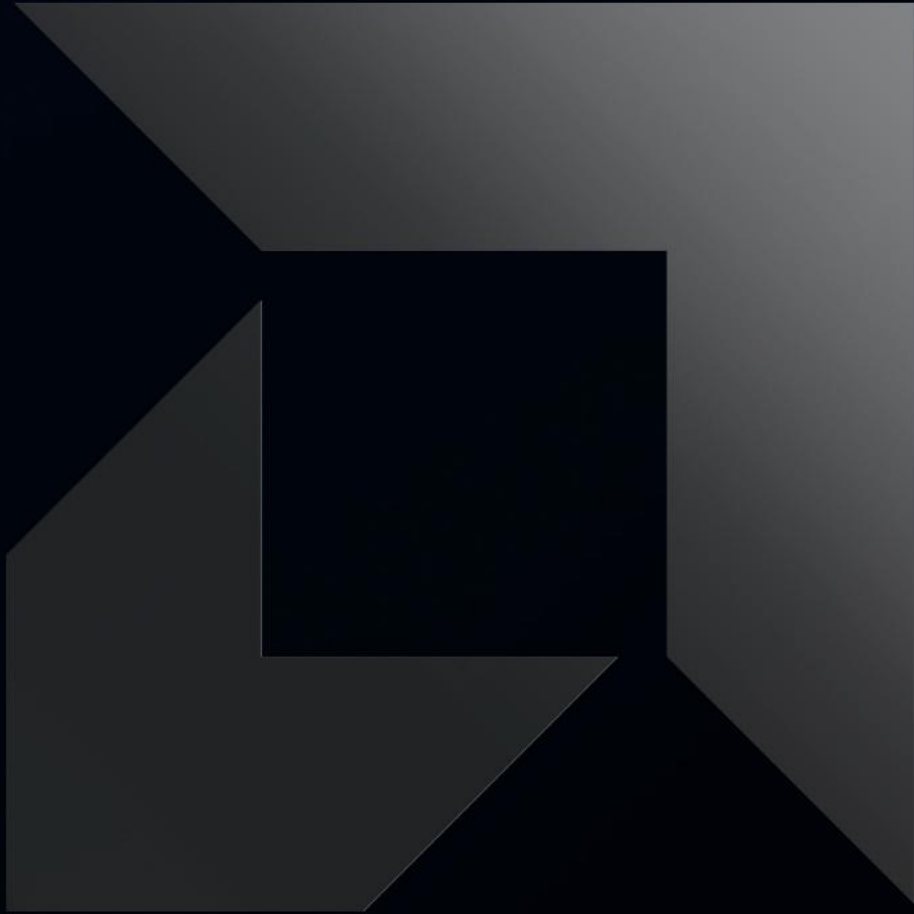
# Register usage is reported by most tools

- For OpenMP<sup>®</sup> offload, use export LIBOMPTARGET\_KERNEL\_TRACE=1, compile and run, more recent compiler versions will also report occupancy

## 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_I52
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_I97
```

- Rocprofv3 reports registers in some of the .csv files and in traces
- Rocprof-sys shows registers in the kernel details popup
- Rocprof-compute shows registers in the counter output tables
- Rocgdb will show registers and their data when stopped in a GPU routine – use ‘info registers’



# Transpose Examples

## Data Ordering and Coalescing

### Memory Loads

# Goals

- Examine data ordering and its effects on performance
- Understand what contiguous means in the context of GPUs
- Understand what coalesced memory load means
- Know how to use local data share (LDS) to mitigate non-contiguous memory access
- Know how to avoid bank conflicts

# Data ordering

- The order that data is processed has a large effect on performance
  - Contiguous data will allow the GPU to coalesce memory reads and/or writes
  - Result is that fewer cache lines need to be fetched
  - Also, fewer page loads will be needed

# What are “coalesced” memory loads

- If the threads in a wavefront access contiguous data, it will be loaded in a minimal number cache line loads
  - If the data is aligned, and if the data the wavefront touches fits wholly inside one 64-128 byte cache, it can be done in one cache line load (64-128 bytes on Instinct GPUs)
- If the data is scattered, multiple cache line loads are necessary
- Once the data is in the GPU, the GPU is very efficient at using the data
  - Similar for vector operations on the CPU
- But with memory bandwidth limited kernels, these multiple cache line loads limit performance
  - When we say an application is memory bandwidth limited, we are saying that it is **cache line load limited** or page table limited. We cannot load less than a cache line or a partial page. This results in wasted bandwidth.

# How to write kernels for “coalesced” memory loads 1/2

- Use contiguous, unit-stride access per thread:
  - Every 64-byte cache line brought into the L1/L2 cache is fully consumed before the next line is needed
  - Do it by having threads in a wavefront access sequential indices
  - If looping, map by strides to keep waves in lockstep over contiguous ranges.
- Prefer *Structure of Arrays* (SoA) over *Array of Structures* (AoS) for vectorized field-wise access:
  - If each lane needs field x from many records, SoA keeps the x values contiguous, enabling full coalescing.

## SoA

```
struct Particles {
    float *x, *y, *z;
    float *vx, *vy, *vz;
    int *id;
};

__global__ void update_pos(Particles p, float dt)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    p.x[i] += dt * p.vx[i];
    p.y[i] += dt * p.vy[i];
    p.z[i] += dt * p.vz[i];
}
```

## AoS

```
struct Particle { float x, y, z; float id; };

__global__ void update_pos(Particle *p, float dt)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    p[i].x += dt * p[i].vx;
    p[i].y += dt * p[i].vy;
    p[i].z += dt * p[i].vz;
}
```

Code generated with GPT-OSS:20b

# How to write kernels for “coalesced” memory loads 2/2

- Align data:
  - Ensure arrays of 4/8/16-byte elements are aligned to 64 B or 128 B if possible.
  - Use `hipMalloc` – it generally does the correct alignment for the GPU
  - Pad/promise alignment on structs and arrays (e.g., `alignas(16/32/64)`) so that common vector types (`float2/float4`) are naturally aligned.

```
// Every Particle object begins at a 64-byte boundary.
// pos and vel (float4) are therefore 16-byte aligned automatically.
struct alignas(64) Particle
{
    float4 pos;    // 16-byte SIMD vector
    float4 vel;    // 16-byte SIMD vector
    float  mass;   // 4 bytes
    float  pad;    // pad so the next Particle still starts on 64 B
};
```

The pad float is not necessary with `alignas` but it would be useful without

- Use vector types to encourage wide loads:
  - `float2/float4`, `int2/int4` map to `global_load_dwordx2/x4` in ISA and can reduce instruction count and improve alignment handling, provided the pointer is correctly aligned.

Code generated with OpenAI o3

# Effects of misalignment and non-contiguous memory access

- Misalignment:
  - If the first lane's address is not at a "good" boundary, the wave's combined footprint can span an extra cache-line.
    - May require an additional cache line load, i.e. 9 cache line loads instead of 8 for a 12.5% penalty
- Strides greater than 1 or random memory access:
  - Loops in kernels where threads access  $i$  in first iteration and  $i+1$  in second so that wavefront is accessing every other value.
  - column-wise access to row-major matrices,
  - AoS when each lane (thread) needs the same field but fields are interleaved in memory.
- Remedies:
  - transpose,
  - use SoA, or
  - stage through local data share (LDS).
  - If must use AoS, might help to pad/align and use vector loads.

# Using LDS to mitigate non-sequential memory access

- Background on LDS banks
  - To deliver high bandwidth at modest area cost, LDS is physically split into 32 independent banks, each 4 bytes wide per port.
- **NOTE:** memory loads to LDS and from LDS to vector unit use Data Share instructions and do not need coalescing
- Rather, LDS memory performance is subject to bank conflicts
  - A bank conflict happens when two or more wavefront lanes attempt to access the same LDS bank in the same clock cycle, and the ports on that bank cannot serve all those requests concurrently
  - Bank conflicts most commonly arise when the memory access pattern uses a stride that's a multiple of the number of banks (32)
- Typical pattern for performance
  - Load a tile from global memory with coalesced vector accesses.
  - Store to LDS in a pattern convenient for the compute (e.g., transposed).
  - Compute using LDS-resident data with bank-conflict-aware indexing.
  - Write results back to global in a coalesced way.
- Tips for using LDS
  - Add padding to avoid bank conflicts when accessing columns (e.g.,  $K+1$  stride in shared tiles).
  - Keep LDS usage per workgroup within the hardware limit to maintain occupancy.

# Vectorized loads and when to use them

- Vectorized loads include
  - double2/double3/double4,
  - float2/float3/float4,
  - int2/int3/int4,
  - uint2/uint3/uint4
- Pros:
  - fewer instructions
  - better alignment guarantees
  - often improves bandwidth utilization when data is aligned and grouped
  - naturally leads to loop unrolling
- Cons:
  - Requires alignment
  - Must handle tails if length not divisible by the vector width
  - Vectors of 3 may not be as performant as vectors that are powers of 2 due to alignment issues
- Compiler hints:
  - Use `restrict` to help alias analysis.
  - Use `alignas` or `attribute((aligned(N)))` on allocations/structs.
  - In HIP, `__builtin_assume_aligned(ptr, N)` can help the compiler pick wide ops if you guarantee alignment.

# Transpose example

- We examine the programming advice in the context of a transpose example
- The examples are in the [HPCTrainingExamples](https://github.com/AMD/HPCTrainingExamples) repository
- `git clone https://github.com/AMD/HPCTrainingExamples`
- `cd HPCTrainingExamples/HIP/transpose`
- We'll look at the examples in the following sequence
  1. `Transpose_read_contiguous`
  2. `Transpose_write_contiguous`
  3. `Transpose_tiled`

Keep in mind, the global index of a thread, with the x-dimension varying faster, followed by the y and then the z

```
tid_linear = threadIdx.x  
            + threadIdx.y * blockDim.x  
            + threadIdx.z * blockDim.x * blockDim.y;
```

# Transpose\_read\_contiguous example

- It is natural to read the data in the input data array pattern
- The input data in a C/C++ allocated array is generally row-major.
- Host allocation is done as a 1D array and then the indexing is done manually
- We do the 2D indexing in the kernel with a define statement to make it easier to understand

```
#define GIDX(y, x, sizex) y * sizex + x
```

- The transpose operation then looks like the following with the **read contiguous** and the **write striding** through the data

```
// Transpose: output[x][y] = input[y][x]
```

```
/* Basic version with read contiguous memory
Assume a 4 x 3 matrix (height = 4, width = 3) stored row-major:
After transposition we want a 3 x 4 matrix, also stored row-major:
height = 4, width = 3, height = 3 width = 4
input (row-major)  output(row_major)
| 0  1  2 |      | 0  3  6  9 |
| 3  4  5 |      | 1  4  7 10 |
| 6  7  8 |      | 2  5  8 11 |
| 9 10 11 |
reading -- 0 1 2 3 4 5 6 7 8 9 10 11
writing  -- 0 3 6 9 1 4 7 10 2 5 8 11
*/
```

# Read contiguous kernel code

- See the kernel code in `transpose_kernel_read_contiguous.cpp`

```
17 #define GIDX(y, x, sizex) y * sizex + x
18
19 __global__ void transpose_kernel_read_contiguous(
20     const double* __restrict__ input, double* __restrict__ output,
21     int srcHeight, int srcWidth) {
22     // Calculate source global thread indices
23     const int srcX = blockIdx.x * blockDim.x + threadIdx.x;
24     const int srcY = blockIdx.y * blockDim.y + threadIdx.y;
25
26     // Boundary check
27     if (srcY < srcHeight && srcX < srcWidth) {
28         // Transpose: output[x][y] = input[y][x]
29         const int input_gid = GIDX(srcY,srcX,srcWidth);
30         const int output_gid = GIDX(srcX,srcY,srcHeight); // flipped axis
31         output[output_gid] = input[input_gid];
32     }
33 }
```

- Build and run the code

```
make transpose_read_contiguous
./transpose_read_contiguous
```

- Output – selected

```
Testing Matrix dimensions: 8192 x 8192
Basic Transpose, Read Contiguous - Average Time: 4450.20 µs
```

# Transpose\_write\_contiguous example

- How about if we make the data write contiguous?
- The transpose operation then looks like the following with the write contiguous and the read striding through the data

```
// Transpose: output[y][x] = input[x][y]
```

```
/* Basic version with write contiguous memory
   Assume a 3 x 4 matrix (height = 3, width = 4) stored row-major:
   After transposition we want a 4 x 3 matrix, also stored row-major:
   height = 3, width = 4   height = 4, width = 3
   output (row-major)   input(row_major)
   | 0  1  2  3 |       | 0  4  8 |
   | 4  5  6  7 |       | 1  5  9 |
   | 8  9 10 11 |       | 2  6 10 |
                       | 3  7 11 |
   reading  -- 0 4 8 1 5 9 2 6 10 3 7 11
   writing   -- 0 1 2 3 4 5 6 7 8 9 10 11
   */
```

# Write contiguous kernel code

- See the kernel code in `transpose_kernel_write_contiguous.cpp`

```

16 #define GIDX(y, x, sizex) y * sizex + x
17
18 __global__ void transpose_kernel_write_contiguous(
19     const double* __restrict__ input, double* __restrict__ output,
20     int srcHeight, int srcWidth) {
21     // Calculate destination global thread indices
22     const int dstX = blockIdx.x * blockDim.x + threadIdx.x;
23     const int dstY = blockIdx.y * blockDim.y + threadIdx.y;
24     const int dstWidth = srcHeight;
25     const int dstHeight = srcWidth;
26
27     // Boundary check
28     if (dstY < dstHeight && dstX < dstWidth) {
29         // Transpose: output[y][x] = input[x][y]
30         const int input_gid = GIDX(dstX,dstY,srcWidth); // flipped axis
31         const int output_gid = GIDX(dstY,dstX,dstWidth);
32
33         output[output_gid] = input[input_gid];
34     }
35 }

```

- Build and run the code

```

make transpose_write_contiguous
./transpose_write_contiguous

```

- Output – selected

```

Testing Matrix dimensions: 8192 x 8192
Basic Transpose, Write Contiguous - Average Time: 2901.80 µs

```

This is substantially faster than the read contiguous version! Write has to do a load/store which is more work. If there were a lot of arrays being read, read contiguous might come out ahead.

# Applying some of the ideas above

- We'll use the shared memory (LDS) on the Compute Unit to create a small memory tile
- This allows us to read contiguous **and** write contiguous data from/to arrays
- We will pad the LDS tile to avoid bank conflicts
  - shared-memory tile → `__shared__ double tile[TILE_SIZE][TILE_SIZE+PAD]`
  - Note that the pad should be added to the second dimension, because that dimension is what impacts the striding pattern that is involved in the selection of an LDS bank
- We need to add a synchronization after loading the tile
  - `__syncthreads();`
- We use restrict on the function arguments
- We also use const for variables
- We use the same integer type for variables in the if tests to avoid having the compiler add instructions
- The `TILE_SIZE` variable is limited to the workgroup size of 1024. This means 32x32 is the largest tile that can be used.

# Tiled transpose kernel

```

14 __global__ void transpose_kernel_tiled(
15 double* __restrict input, double* __restrict output,
16 const int srcHeight, const int srcWidth)
17 {
18     // thread coordinates in the source matrix
19     const int tx = threadIdx.x;
20     const int ty = threadIdx.y;
21
22     // source global coordinates this thread will read
23     const int srcX = blockIdx.x * TILE_SIZE + tx;
24     const int srcY = blockIdx.y * TILE_SIZE + ty;
25
26     // allocate a shared (LDS) memory tile with padding to avoid bank conflicts
27     __shared__ double tile[TILE_SIZE][TILE_SIZE + PAD];
28
29     // Read from global memory into tile with coalesced reads
30     if (srcY < srcHeight && srcX < srcWidth) {
31         tile[ty][tx] = input[GIDX(srcY, srcX, srcWidth)];
32     } else {
33         tile[ty][tx] = 0.0;           // guard value – never used for writes
34     }
35
36     // Synchronize to make sure all of the tile is updated before using it
37     __syncthreads();
38
39     // destination global coordinates this thread will write
40     const int dstY = blockIdx.x * TILE_SIZE + ty; // swapped axes
41     const int dstX = blockIdx.y * TILE_SIZE + tx;
42
43     // Write back to global memory with coalesced writes
44     if (dstY < srcWidth && dstX < srcHeight) {
45         output[GIDX(dstY, dstX, srcWidth)] = tile[tx][ty];
46     }
47 }

```

swapping the order for the tile does not create memory access issues  
because memory loads to LDS and from LDS do not need coalescing

# Build tiled version and run

Build and run the tiled transpose

```
make transpose_tiled  
./transpose_tiled
```

- Output

```
Tiled Transpose, Read and Write Contiguous - Average Time: 2686.40 μs
```

# Build timed version that compares all versions

- We can run a combined version that does a comparison of all of the versions

```
make transpose_timed
./transpose_timed
```

- Output

```
Performance Summary:
```

```
Basic read contiguous 4439.60 μs
```

```
Basic write contiguous 2899.80 μs
```

```
Tiled - both contiguous 2686.80 μs
```

```
Speedup (Write Contiguous): 1.53x
```

```
Speedup (Tiled - Both Contiguous): 1.65x
```

```
Speedup (ROCblas): 1.22x
```

```
Verification: PASSED
```

There is a 1.53x speedup by using the contiguous write and 1.65x for this 8192x8192 matrix size on an MI210 GPU running ROCm 6.4.1.

# 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. All rights reserved. AMD, the AMD Arrow logo, ROCm, Radeon, CDNA, Instinct, Infinity Cache 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.

Git and the Git logo are either registered trademarks or trademarks of Software Freedom Conservancy, Inc., corporate home of the Git Project, in the United States and/or other countries

The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board

**AMD** 