Optimizing HIP Applications

Presenter: Gina Sitaraman
AMD @ HLRS
May 6th, 2025

[Public]

What is covered in the HIP: Optimization Presentation

GPUs are high throughput devices; programmers must expose parallelism to properly utilize them.

- Memory bound kernels optimization
- Compute bound kernels optimization
- Latency bound kernels optimization

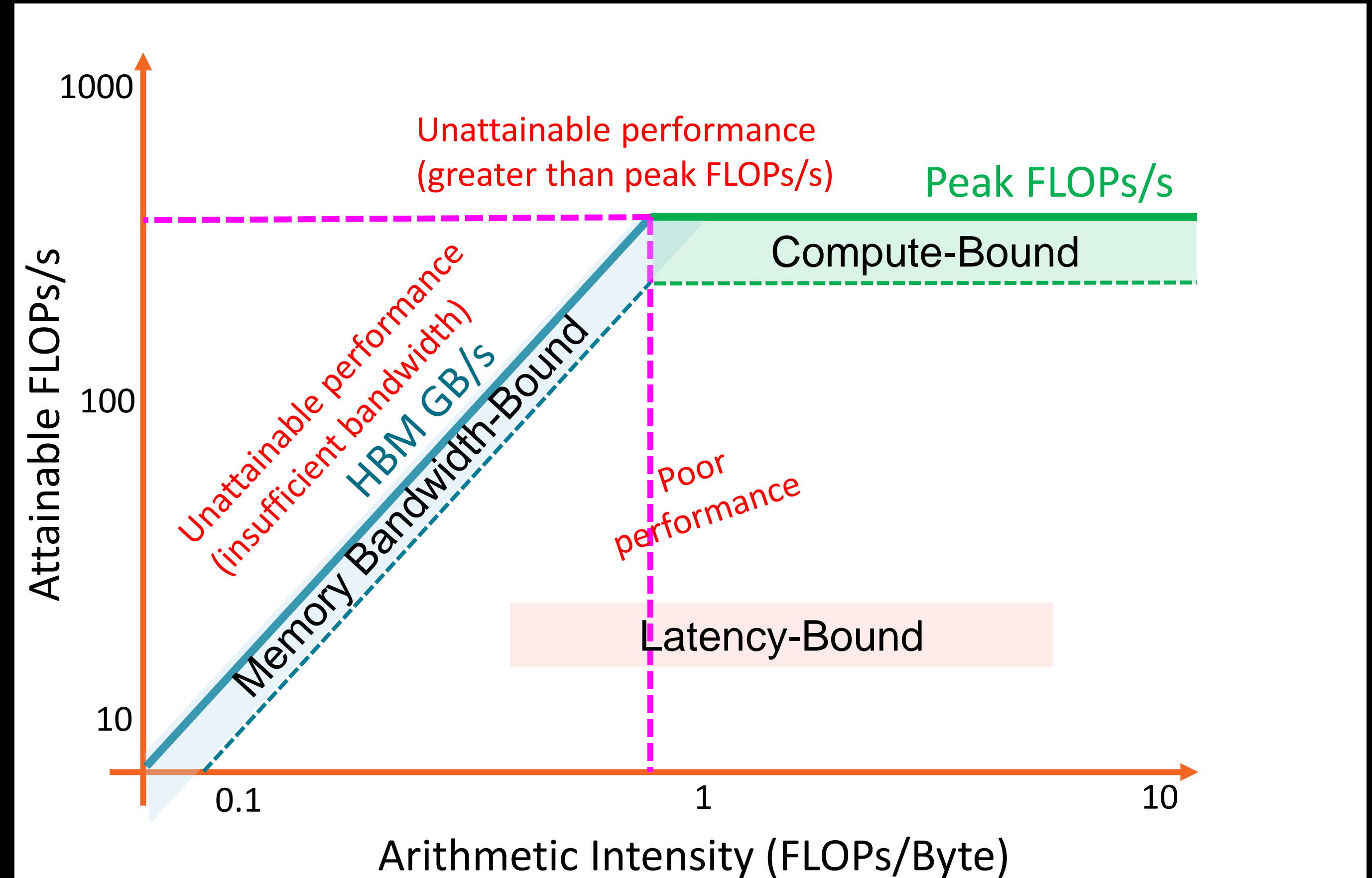
[Public]

Optimization strategy depends on performance limiters

Memory bound
- Low arithmetic intensity, memory units saturated

Compute bound
- High arithmetic intensity, compute units saturated

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



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

AMD
together we advance_

Memory Bound Kernel: Data Movement Considerations

- Memory bound kernels have low arithmetic intensity and perform few ops per byte moved
- 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

[Public]

Memory Bound Kernel: 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

- 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

- 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];
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  }
```

Each work item loads and stores two elements

```
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

[Public]

Compute Bound Kernel 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
  - Call optimized libraries to extract best performance (rocBLAS, rocSPARSE, etc.)
- 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
- 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
- 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 – conversions between data types

```
__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;
}
```

FP64 ops

Type Conversions

```
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_fmac_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, v5, v0
global_store_dword v4, v0, s[0:1]
s_endpgm
```

FP32 ops

```
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_fmac_f32_e32 v1, 2.0, v2
global_store_dword v0, v1, s[0:1]
s_endpgm
```

Fewer instructions!!

AMD together we advance_

[Public]

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

AMD
together we advance_

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

[Public]

Occupancy by Example (daxpy)

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

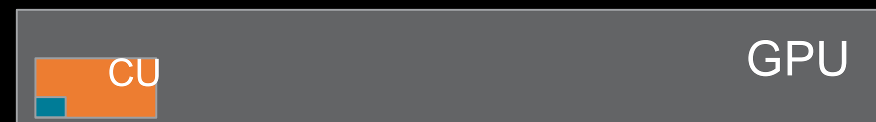
We know that

- 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:

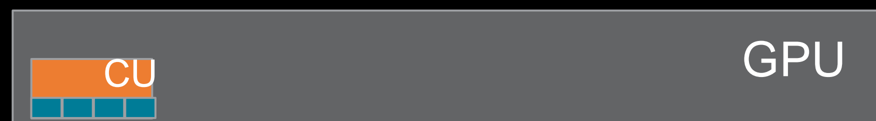
1 workgroup with 64 work-items

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

~1000 workgroups = ~16000 waves = good occupancy

But that’s not the whole picture..

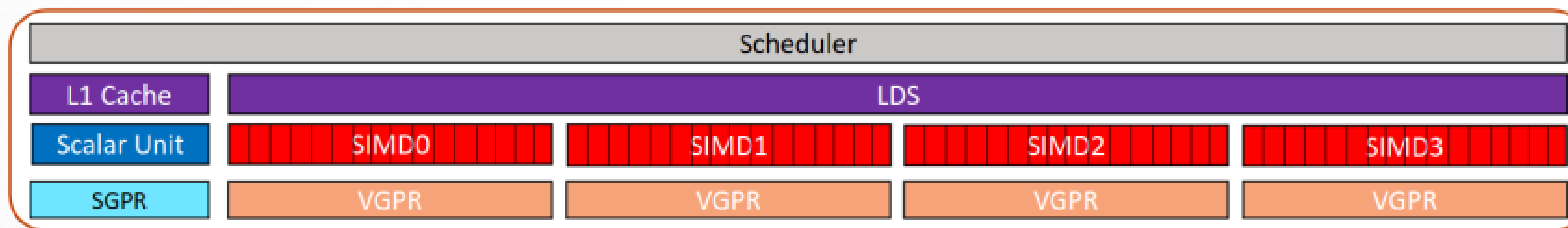
AMD
together we advance_

[Public]

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)

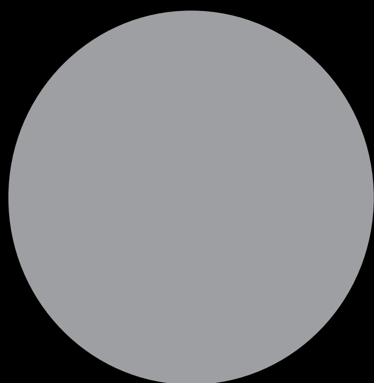


The GPU can only schedule more work if there are enough resources available

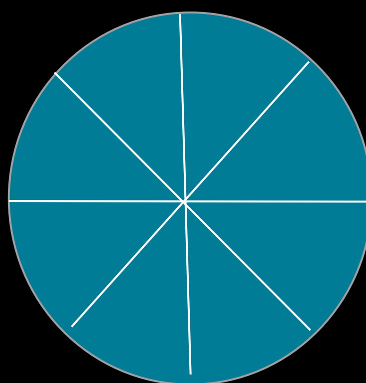
[Public]

How LDS affects Occupancy

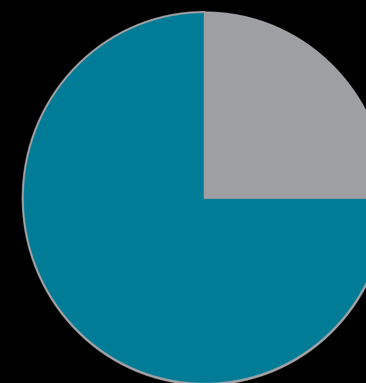
- Fast, on-CU, software managed memory to efficiently share data between work-items of a workgroup
- Each CU in a MI200 GPU has 64 KiB of LDS available
- Shared among workgroups on CU



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

[Public]

How Registers affect Occupancy

- In vector registers (VGPRs), each thread in the wavefront can save its own value
  - On an MI200 CU - max 512 VGPRs of size 4 bytes
- In scalar registers (SGPRs), one value is shared across all work-items of the wavefront
  - On an MI200 CU - max 102 SGPRs of size 4 bytes per wavefront

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.

AMD
together we advance_

[Public]

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
- Register Pressure blog: https://rocm.blogs.amd.com/software-tools-optimization/register-pressure/README.html

AMD
together we advance_

[Public]

Launching kernels has a cost

- “Cold” Launch Latency
  - If device is idle when kernel is launched, it takes a while for waves to be scheduled
  - Once waves start being scheduled, it can still take some time for the device to fill
- “Hot” Launch Latency
  - Launching kernel when device is busy can hide much of the startup cost
  - However, kernels in the same HIP stream are ordered. Therefore, all waves in a kernel in a HIP stream must complete before any wave from the next kernel in the stream can be scheduled.
    - Some cycles are spent at kernel boundaries for flushing writes from kernel
- Kernels that are too short (<< 1ms) suffer from kernel launch overhead

AMD together we advance_

[Public]

Fuse kernels to reduce launch latencies

Added benefit: reduced data movement

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 overuse 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

[Public]

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)

[Public]

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

Even threads write 1.0, and odd threads write 0.0

Having each thread produce two output values removes the divergent section in the kernel

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

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 - GPU page tables populated, registered on CPU only on first touch
  - malloc – CPU page tables populated; GPU only registers them on first touch
  - hipHostMalloc (or) malloc + hipHostRegister – page tables populated on both CPU and GPU
- 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

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, the AMD Arrow logo, ROCm 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.

© 2024 Advanced Micro Devices, Inc. All rights reserved.

AMD