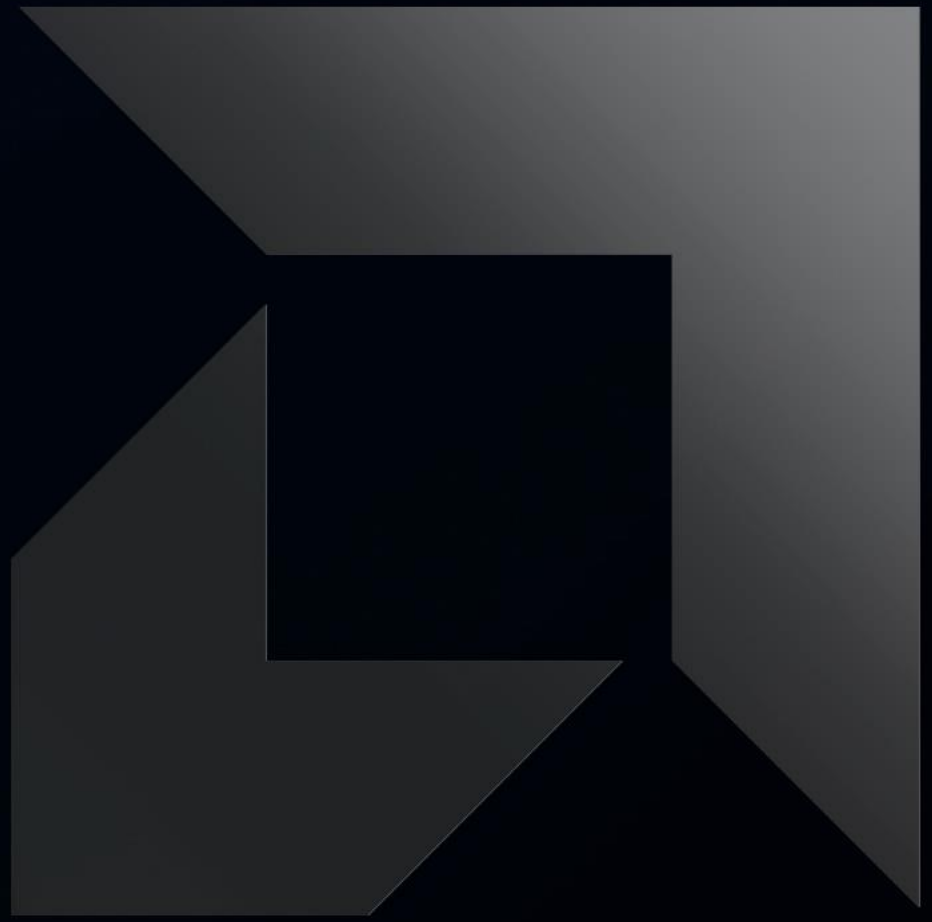


Introduction to Porting and Running Applications to AMD GPUs

Bob Robey, Giacomo Capodaglio
AMD GPU Training Team

June 25th, 2024

AMD
together we advance_

Training Workshop Guidelines – help us create a positive environment

Please help us create a positive collaborative environment for this event. Our top goal is to attract and build the community in high performance computing. Please consider not only the technical aspects of your interactions, but also how it affects other participants.

- Start your comments with a positive statement before your comment or question
- Give equal time to all participants
- Please, help others with technical issues and without negative comments
- And most importantly, respect all participants, regardless of background, language, culture

AMD @ EPCC Workshop Schedule – Tuesday, June 25th 2024

1. AMD Architecture and Memory Model - (**Bob Robey**)
2. The Software Porting Process - (**Bob Robey**)
3. HIP and ROCm - (**Giacomo Capodaglio**)
4. OpenMP on AMD GPUs - (**Bob Robey**)
5. Performance Portability Languages - (**Bob Robey**)
6. Overview of AMD Tools - (**Giacomo Capodaglio**)

Acknowledgements

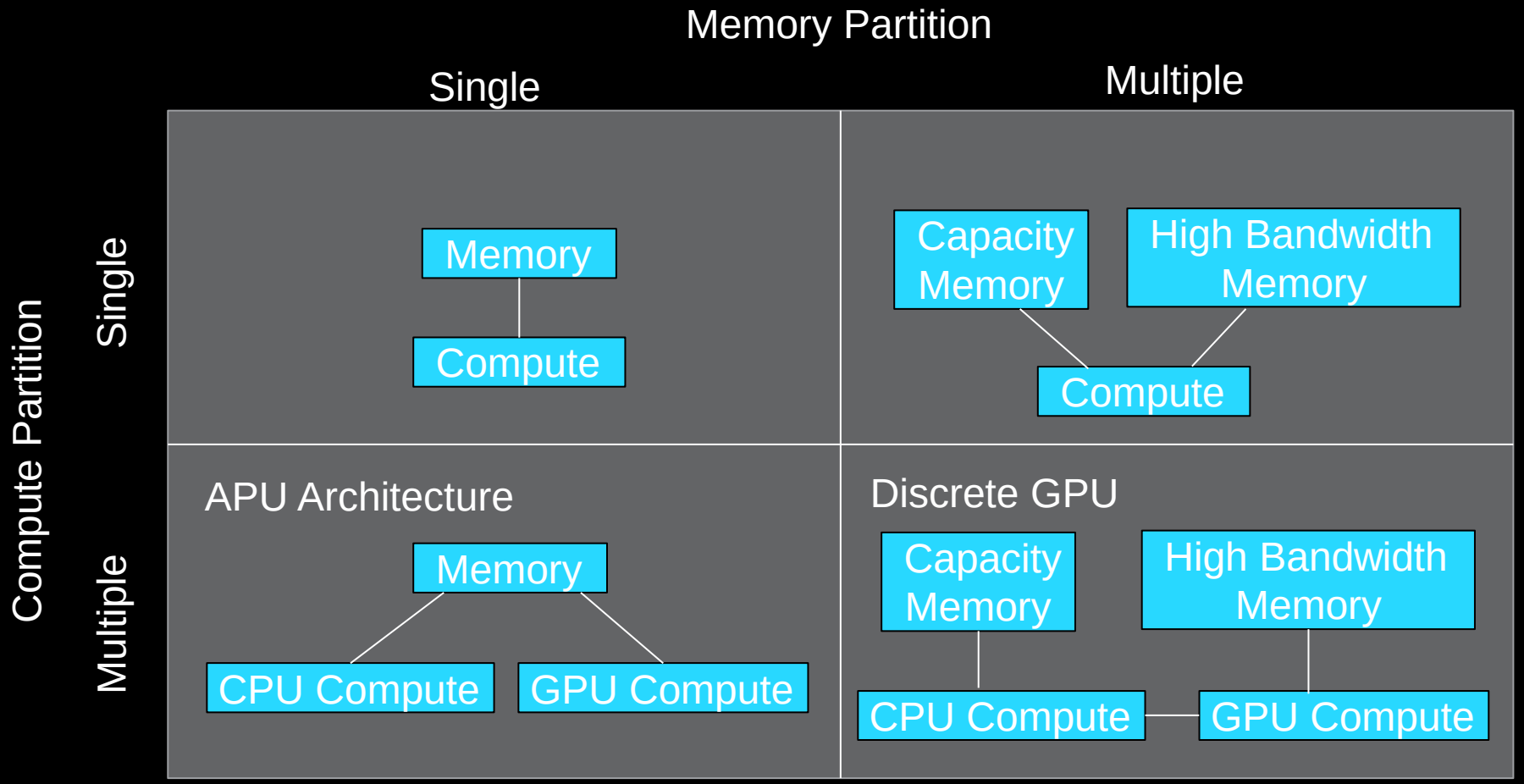
- Leopold Grinberg
- Nicholas Malaya
- Maria del Carmen Ruiz Varela
- Suyash Tandon
- Justin Chang
- Julio Maia
- Noel Chalmers
- Paul T. Bauman
- Nicholas Curtis
- Alessandro Fanfarillo
- Jose Noudohouenou
- Ian Bogle
- Paul Bauer
- Chip Freitag
- Damon McDougall
- Noah Wolfe
- Jakub Kurzak
- Samuel Antao
- George Markomanolis
- Gina Sitaraman
- Asitav Mishra
- Johanna Potyka
- Shelby Lockhart
- And many other colleagues

1. AMD Architecture and Memory Model

What AMD offers

- **Accelerated Processing Unit (APU) Architecture** from HPC center to Desktop
  - Only product that is a true APU
- Commitment to **APU Programming model** for GPU products
- AMD Values Customer friendly policies
  - Open-source
  - No vendor lock-in
  - Portability
- Industry leading CPU performance
- Commitment to HPC customers
  - Leading FP64 performance, based on true FP64 (IEEE-754) performance specs unless otherwise stated
  - Increasing every generation
  - No special tricks to get FP64 for general applications
- Commitment to ML/AI customers
  - Drop-in support for ML/AI applications

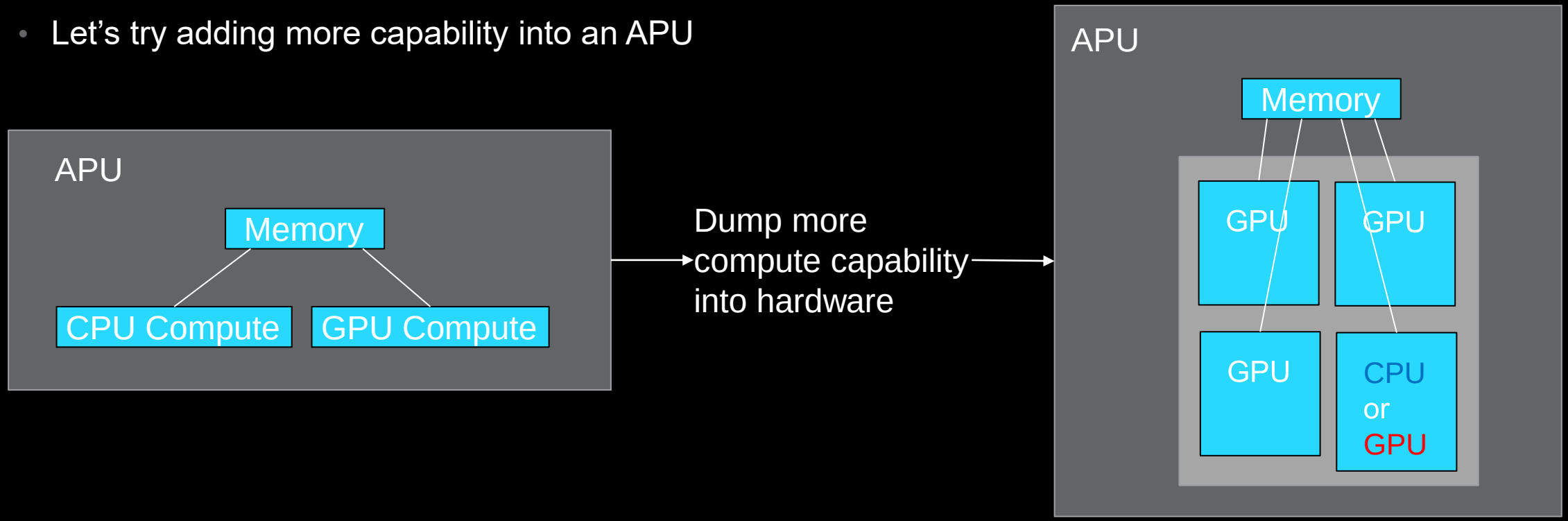
Taxonomy of Compute Architectures



- Taxonomy categorizes architecture by dominance of hardware components
  - Memory Dominant – architecture revolves around a single memory space
  - Compute Dominant – architecture centered around a single compute resource

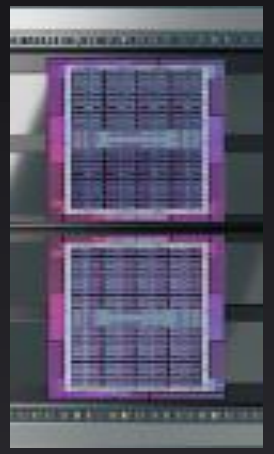
Breakthrough in compute capability of an APU

- Integrated GPUs have traditionally been limited by how much GPU compute capability can be included
  - Silicon Chip only has so much space
  - Chiplets allow us to expand that space
- Let's try adding more capability into an APU



Bringing it to AMD Instinct™ Accelerator Products

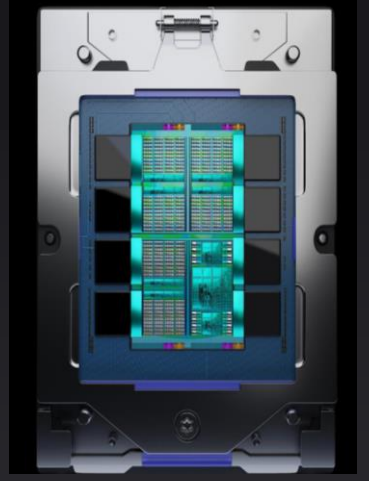
MI200 Series



Extreme Compute Architecture with leading memory capacity and bandwidth

- Technology in first Exascale systems
- High compute to power ratio
- Tight integration with memory
- Infinity Fabric™ for data transfers

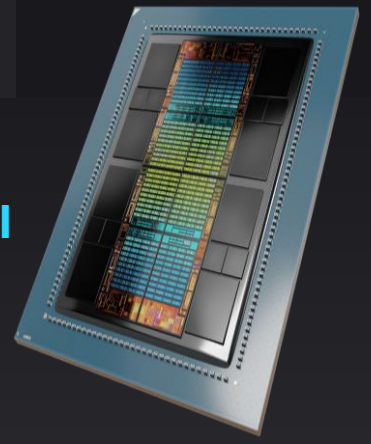
MI300A



First True APU Architecture for HPC and AI

- Memory Bandwidth Workloads
- Hybrid CPU + GPU Capability
- GPUs can drive full bandwidth

MI300X



Leadership generative AI accelerator

- Extreme Compute Workloads
- Suitable for typical AI work
- Other work entirely on GPU

Memory Model

Definition

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.

AMD MI200 GPUs and Memory Addressing

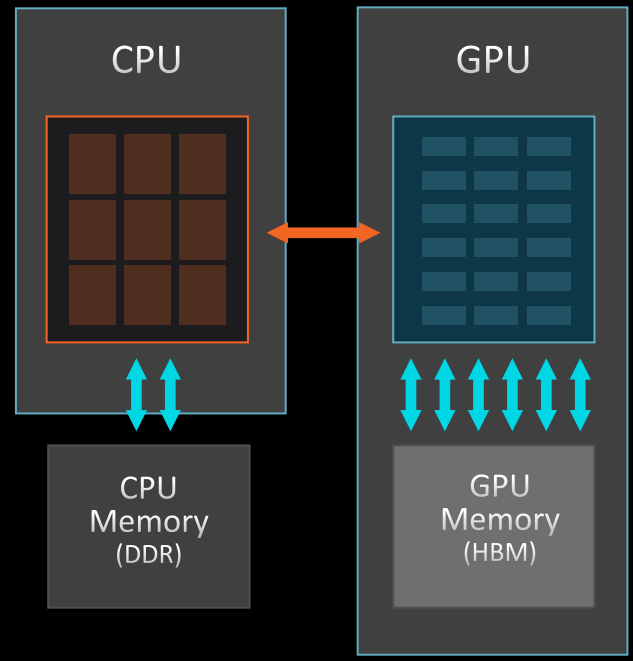
In discrete GPU systems, CPU and GPU memory spaces are **separate** and data needs to be **moved** between the two spaces. This data movement can be performed in two ways:

1. By the **programmer**, explicitly
2. By the **Operating System** (OS), who helps move pages on access and subsequent page fault
   - We call this managed memory – short for "memory is managed by the operating system"
   - If no corresponding address is found, the program will fail with a segmentation fault

AMD MI200 GPUs (MI210, MI250, MI250X) are **discrete** GPUs

- Implement **managed memory**
- To enable managed memory, `export HSA_XNACK=1`

AMD CDNA™ 2 Coherent Memory Architecture



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

Definition

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

xnack environment variable

On MI250X, it 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

Run `rocminfo | grep xnack` to check if xnack is enabled

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]
- `--offload-arch=gfx90a:xnack+` [ROCm™ >= 4.5]

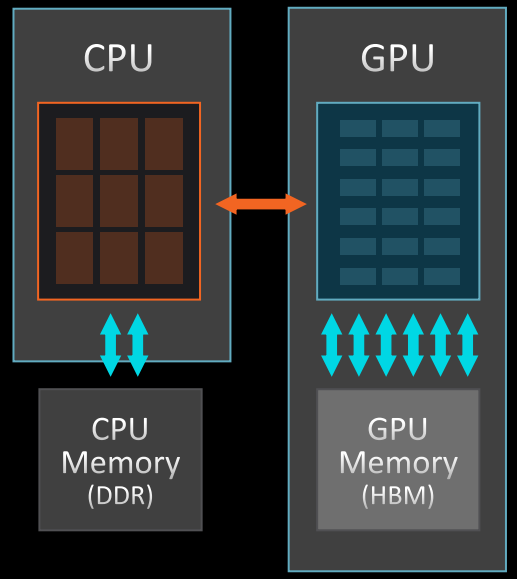
Supplying multiple xnack options will yield a "fat-binary" with both modes enabled.

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

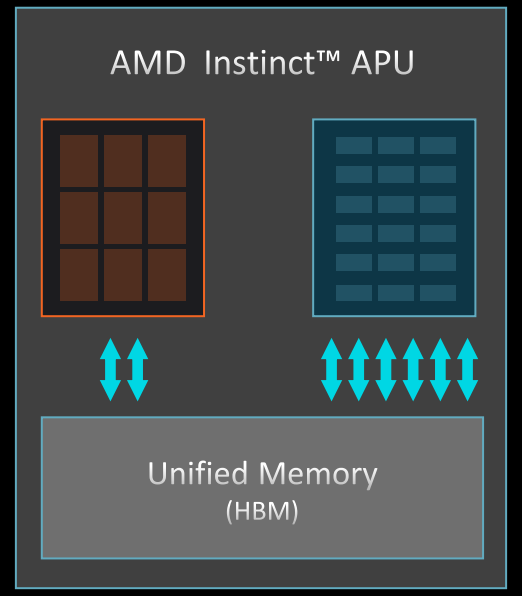
Code compiled with “xnack any” will run in any case.

MI300A UNIFIED MEMORY APU ARCHITECTURE BENEFITS

AMD CDNA™ 2 Coherent Memory Architecture → AMD CDNA™ 3 Unified Memory APU Architecture



- **Eliminate Redundant Memory Copies**
- **No programming distinction between host and device memory spaces**
- **High performance, fine-grained sharing between CPU and GPU processing elements**
- **Single process can address all memory, compute elements on a socket**



APU PROGRAMMING MODEL

CPU CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);


for (int i=0; i<M; i++) // initialize
   in_h[i] = …;

cpu_func(in_h, out_h, M);


for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```

GPU CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);
hipMalloc(&in_d, Msize);
hipMalloc(&out_d, Msize);

for (int i=0; i<M; i++) // initialize
   in_h[i] = …;
hipMemcpy(in_d,in_h,Msize);
gpu_func<< >>(in_d, out_d, M);
hipDeviceSynchronize();
hipMemcpy(out_h,out_d,Msize);

for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```

APU CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);

for (int i=0; i<M; i++) // initialize
   in_h[i] = …;

gpu_func<< >>(in_h, out_h, M);
hipDeviceSynchronize();

for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```

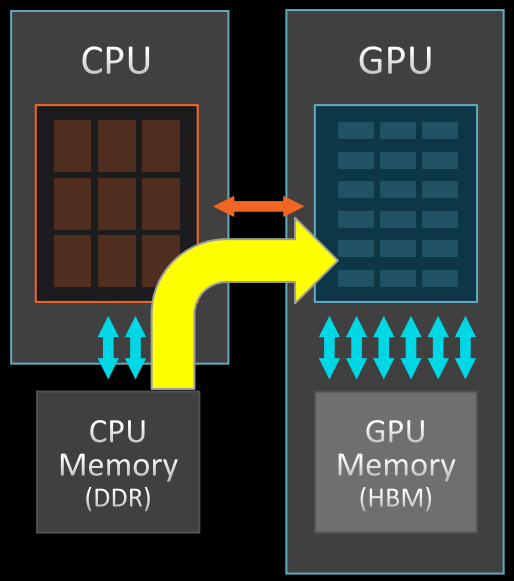
https://github.com/amd/HPCTrainingExamples/tree/main/ManagedMemory

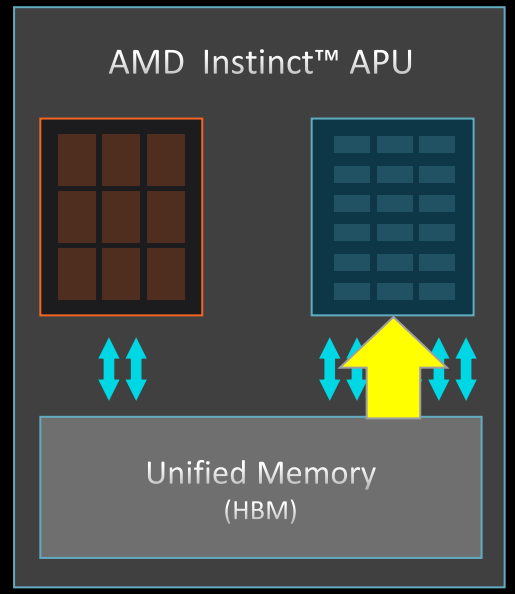
- GPU memory allocation on Device
- Explicit memory management between CPU & GPU
- Synchronization Barrier

APU PROGRAMMING: PERFORMANCE IMPLICATIONS

GPU CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);
hipMalloc(&in_d, Msize);
hipMalloc(&out_d, Msize);


for (int i=0; i<M; i++) //initialize
   in_h[i] = …;
hipMemcpy(in_d,in_h,Msize);
gpu_func<< >>(in_d, out_d, M);
hipDeviceSynchronize();
hipMemcpy(out_h,out_d,Msize);

for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```





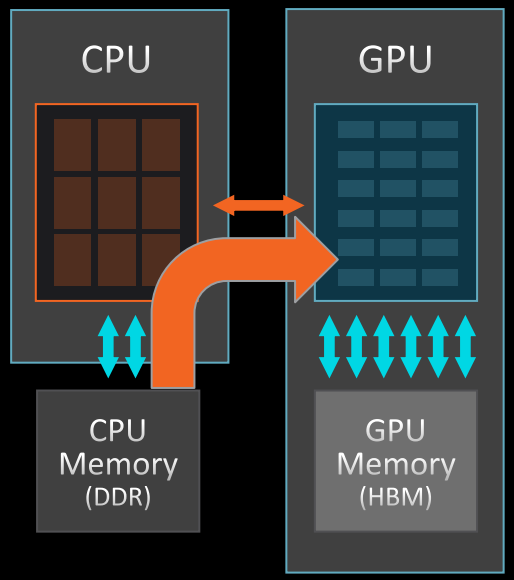
Operation	MI250X (MCM)	MI300A
H2D Copy	O(10) GB/s	O(TB/s)

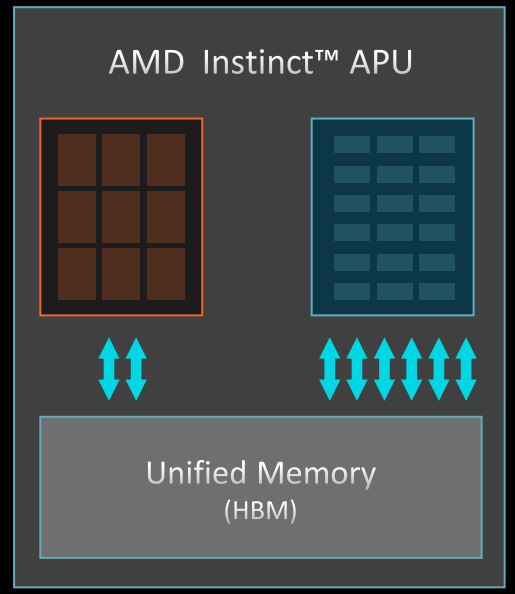
- GPU memory allocation on Device
- Explicit memory management between CPU & GPU
- Synchronization Barrier

APU PROGRAMMING: PERFORMANCE IMPLICATIONS

APU CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);




for (int i=0; i<M; i++) //initialize
   in_h[i] = …;

gpu_func<< >>(in_h, out_h, M);
hipDeviceSynchronize();


for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```





Operation	MI250X (MCM)	MI300A
Coherent Access	O(10) GB/s	N/A

- ~~GPU memory allocation on Device~~
- ~~Explicit memory management between CPU & GPU~~
- Synchronization Barrier

2. The Software Porting Process

Portability, Performance, and Productivity

Performance portability has been identified by the US Department of Energy (DOE) as a critical issue for Exascale computing*

Importance of Portability

- Too much work to port from CPU to GPU if a separate port must be made for each vendor
- Would be nice to have the same code for CPU and GPU

Importance of Performance

- Good performance is a necessity (for motivation and code capabilities)
- Straight-forward code should give good performance across a range of computing hardware

Importance of Productivity

- Cannot afford to increase the number of developers
- Cost of initial port and long-term application support

*Dubey, Anshu, et al. *"Performance portability in the exascale computing project: exploration through a panel series."* Computing in Science & Engineering 23.5 (2021): 46-54.

Multiple Language Paths for AMD GPUs

Native or Low-level languages — HIP, OpenCL

Pragma-based languages — OpenMP, OpenACC

Higher Level Performance Portability languages – Frameworks — Kokkos, RAJA

Native or Low-level Languages

Heterogeneous-compute Interface for Portability (HIP)

- A portable layer on top of ROCm and CUDA

AMD also supports OpenCL™, the original portable low-level language

Converting CUDA to HIP is straightforward

- Hipify scripts do the majority of the work
- Still requires some optimization effort to get good performance

Requires a different source on CPU and GPU

- Requires effort for porting, an overhead for large applications

Pragma-based Languages

OpenMP® – primary supported option for AMD GPUs

- Implemented through LLVM™

OpenACC – Supported through Cray, LLVM™ (CLACC) and GCC compilers

- Also available are source-to-source translation tools from OpenACC to OpenMP (Intel® and CLACC)
- Not as well supported as other options

Standard Language support – mostly a future option

- Long-time discussed “for_each”, “do_concurrent”, “for_all”,
- Parallel C++ through parallel “execution policies”

Higher Level Performance Portability Frameworks

Kokkos – Sandia National Lab (SNL) C++ performance portable programming model

- The Kokkos team has agressively developed support for AMD GPUs via a HIP backend
- Kokkos handles many of the unique attributes of the AMD GPUs for you
- Parts being integrated into the C++ standard

RAJA – Lawrence Livermore National Lab (LLNL) C++ performance portability layer

- Modular in structure with separation of compute and data management
- Supports AMD GPUs
- Key kernel patterns have been optimized by AMD

Advantages of Performance Portability Frameworks

- True single-source application code (at least if you restrict yourself to C++)
- Many of these framework support both CPUs and GPUs

Other Languages

Many other languages also work on AMD GPUs to some level and are continually improving

- SYCL
- Julia
- Python™

ML/AI -- Support for these languages is excellent and portable

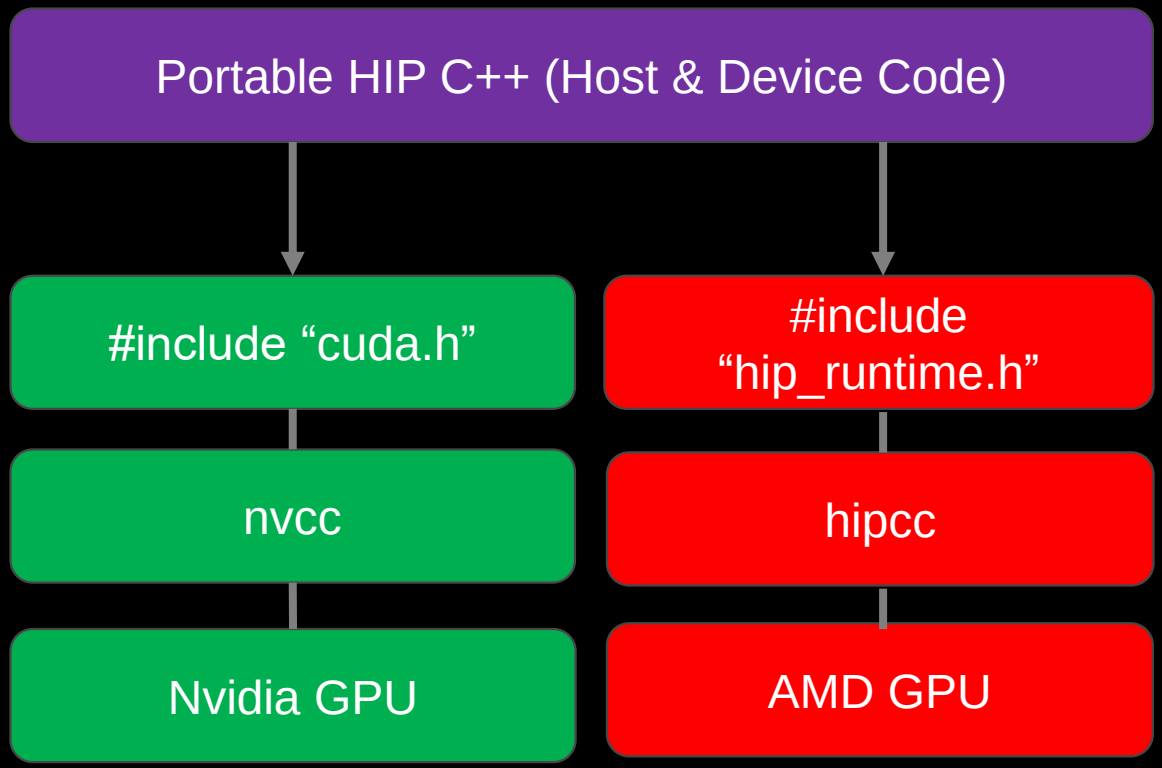
- PyTorch – AMD is a founding member of the PyTorch Foundation
- TensorFlow
- And many other ML/AI packages

3. HIP and ROCm

What is HIP?

AMD's **H**eterogeneous-compute **I**nterface for **P**ortability, or **HIP**, is a **C++** runtime API and kernel language that allows developers to create portable applications that can run on AMD's accelerators as well as CUDA devices

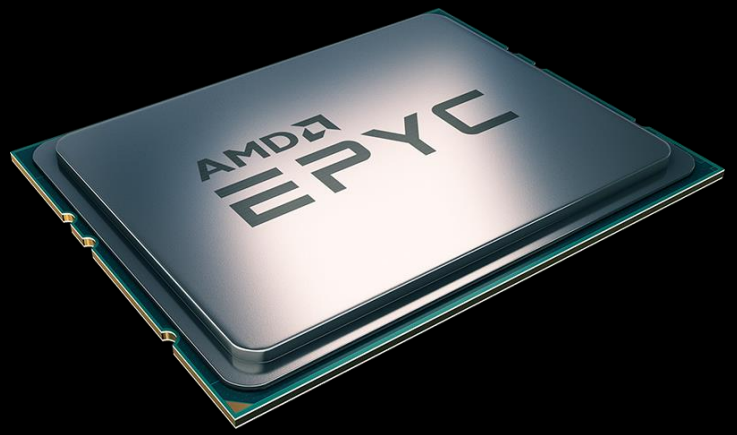
- **Open-source**
- Syntactically similar to CUDA. Most CUDA API calls can be converted in place: cuda -> hip
- Supports a strong subset of CUDA runtime functionality



A Tale of Host and Device

Source code in HIP has two flavors: Host code and Device code

- The Host is the CPU
- Host code runs here
- Usual C++ syntax and features
- Entry point is the ‘main’ function
- HIP API can be used to create device buffers, move between host and device, and launch device code.



- The Device is the GPU
- Device code runs here
- C-like syntax
- Device codes are launched via “kernels”
- Instructions from the Host are enqueued into “streams”

https://rocm.docs.amd.com/projects/HIP/en/latest/.doxygen/docBin/html/index.html

HIP API

Device Management:

- `hipSetDevice()`, `hipGetDevice()`, `hipGetDeviceProperties()`

Memory Management

- `hipMalloc()`, `hipMemcpy()`, `hipMemcpyAsync()`, `hipFree()`

Streams

- `hipStreamCreate()`, `hipDeviceSynchronize()`, `hipStreamSynchronize()`, `hipStreamDestroy()`

Events

- `hipEventCreate()`, `hipEventRecord()`, `hipStreamWaitEvent()`, `hipEventElapsedTime()`

Device Kernels

- `__global__`, `__device__`, `hipLaunchKernelGGL()`

Device code

- `threadIdx`, `blockIdx`, `blockDim`, `__shared__`, 200+ math functions covering entire CUDA math library.

Error handling

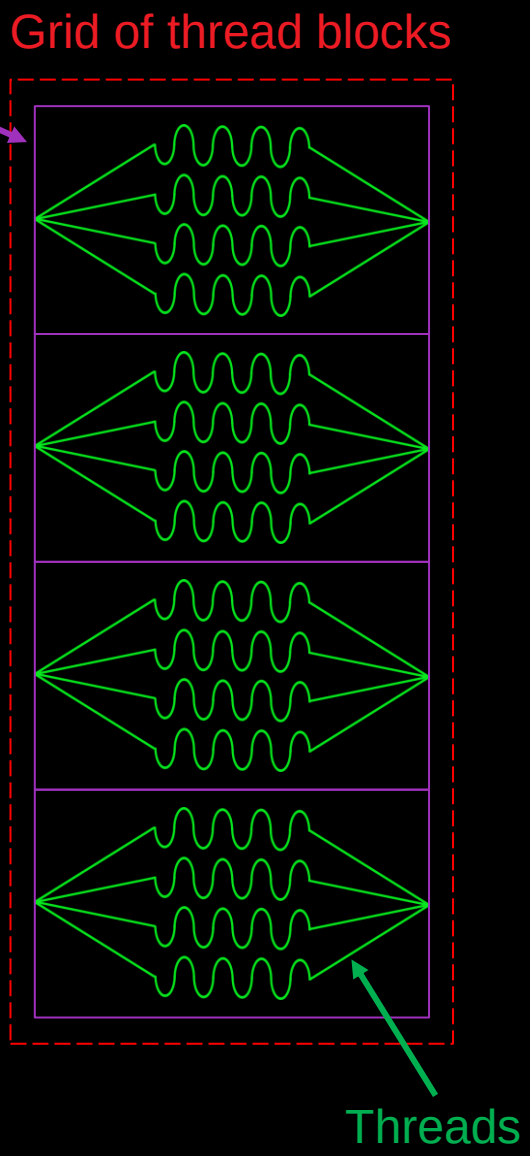
- `hipGetLastError()`, `hipGetErrorString()`

Device Kernels: Grid Hierarchy

- In HIP, kernels are executed on a **grid** of threads that run on a GPU
  - 1D, 2D, and 3D grids are supported, but most work maps well to 1D
  - The grid is what you map your problem to
- Each dimension of the grid is partitioned into equal sized **blocks** of threads
- Each block is made up of multiple **threads**
- The grid and its associated blocks are just organizational constructs, the threads are the things that do the work
- If you're familiar with CUDA already, the grid+block structure is very similar in HIP

TERMINOLOGY

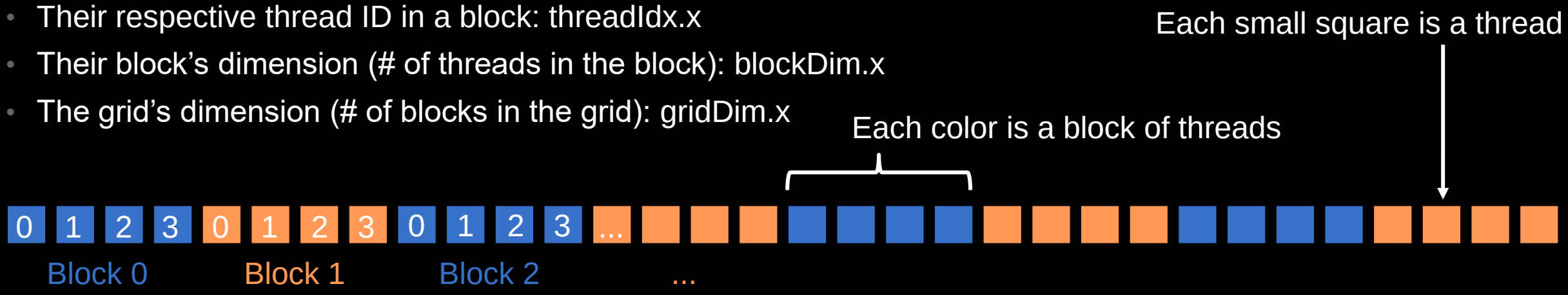
AMD	NVIDIA
Grid	Grid
Workgroup	Thread Block
Thread	Thread
Wavefront (64)	Warp (32)



The Grid: blocks of threads in 1D

Threads in grid have access to:

- Their respective block (workgroup): blockIdx.x
- Their respective thread ID in a block: threadIdx.x
- Their block's dimension (# of threads in the block): blockDim.x
- The grid's dimension (# of blocks in the grid): gridDim.x



Global thread ID

```
int id = blockDim.x * blockIdx.x + threadIdx.x;
```

For example, thread 3 of block 2 would have a global thread ID of 11

```
= 4          * 2          + 3
= 11
```

Example: Simple GPU Multiply

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"

__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}

int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }
```

```
  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```

Example: Simple GPU Multiply

Include header for HIP runtime

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"

__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}

int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }

  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```

Example: Simple GPU Multiply

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"
```

GPU kernel

```
__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}
```

```
int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }

  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```

Example: Simple GPU Multiply

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"

__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}
```

Allocate and initialize host memory buffer

```
int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }

  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```

Example: Simple GPU Multiply

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"

__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}

int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }
```

Allocate GPU buffer and copy values from CPU buffer to GPU buffer

```
  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```

Example: Simple GPU Multiply

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"

__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}

int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }
```

```
  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```

Launch GPU kernel

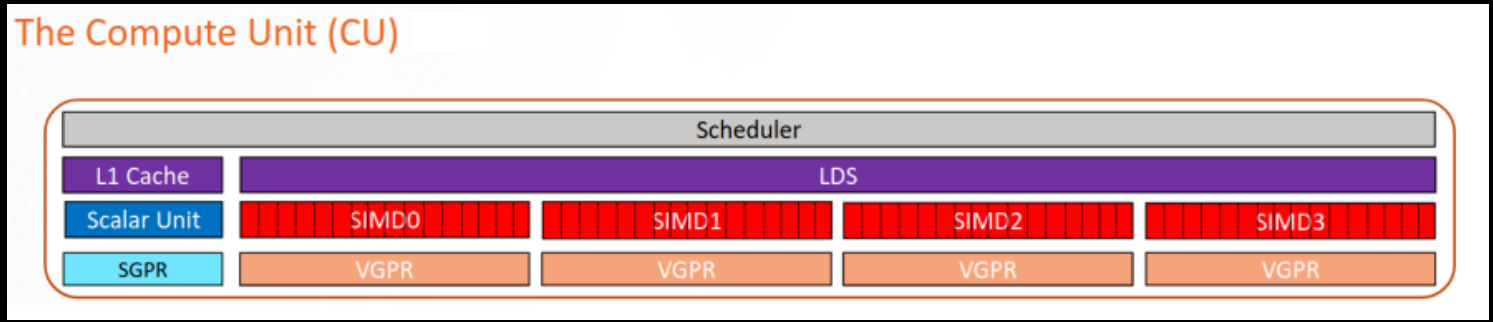
Example: Simple GPU Multiply

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"

__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}

int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }
```

```
  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```

Copy data from GPU buffer to CPU buffer and free memory

Software to hardware mapping



Blocks and threads allow a natural mapping of kernels to hardware:

- Upon kernel launch, a grid of thread blocks is launched to compute the kernel on the compute units (CUs)

Threads within a thread block (workgroup):

- **Execute on the same CU in chunks of 64 threads** called **wavefronts** (or waves).
- Share Local Data Share (LDS) memory and L1 cache
- Can synchronize

About wavefronts:

- Wavefronts execute on SIMD units (located inside the CU)
- If a wavefront stalls (e.g., data dependency) CUs can quickly **context switch** to another wavefront

A good practice is to make the **block size** a multiple of 64 and have several wavefronts (e.g., 256 threads)

Blocking vs Nonblocking API functions

- Launching a kernel is **non-blocking for the host**
  - After sending instructions/data, the host continues to do more work while the device executes the kernel

- However, `hipMemcpy` is **blocking for the host**
  - The data pointed to in the arguments can be safely accessed/modified after the function returns

- To make asynchronous copies, we need to allocate non-pageable (pinned) host memory using hipHostMalloc and copy using hipMemcpyAsync

```
hipHostMalloc(h_a, Nbytes, hipHostMallocDefault);
hipMemcpyAsync(d_a, h_a, Nbytes, hipMemcpyHostToDevice, stream);
```

- It is not safe to access/modify the arguments of `hipMemcpyAsync` without some sort of synchronization.

Side Note: H2D/D2H bandwidth **increases significantly when host memory is pinned**
- It is good practice to use pinned host memory where data is frequently transferred to/from the device

Streams

- A stream in HIP is a **queue of tasks** (e.g. kernels, memcpys, events).
  - Tasks enqueued in a stream **complete in order on that stream**.
  - Tasks being executed in different streams are allowed to overlap and share device resources.

- Streams are created via:

```
hipStream_t stream;
hipStreamCreate(&stream);
```

- And destroyed via:

```
hipStreamDestroy(stream);
```

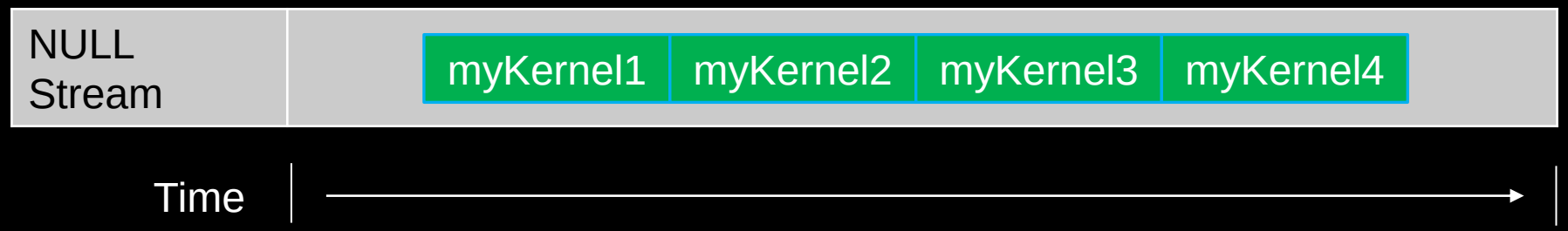
- Passing `0` or `NULL` as the `hipStream_t` argument to a function instructs the function to execute on a stream called the '**NULL Stream**':
  - No task on the NULL stream will begin until **all previously enqueued tasks in all other streams have completed**.
  - Blocking calls like `hipMemcpy` run on the NULL stream.

Streams

- Suppose we have 4 small kernels to execute:

```
myKernel1<<<dim3(1), dim3(256), 0, 0>>>(256, d_a1);
myKernel2<<<dim3(1), dim3(256), 0, 0>>>(256, d_a2);
myKernel3<<<dim3(1), dim3(256), 0, 0>>>(256, d_a3);
myKernel4<<<dim3(1), dim3(256), 0, 0>>>(256, d_a4);
```

- Even though these kernels use only one block each, they'll execute in serial on the NULL stream:



Streams

- With streams we can effectively share the GPU's compute resources:

```
myKernel1<<<dim3(1), dim3(256), 0, stream1>>>(256, d_a1);
myKernel2<<<dim3(1), dim3(256), 0, stream2>>>(256, d_a2);
myKernel3<<<dim3(1), dim3(256), 0, stream3>>>(256, d_a3);
myKernel4<<<dim3(1), dim3(256), 0, stream4>>>(256, d_a4);
```

NULL Stream			
Stream1		myKernel1	
Stream2		myKernel2	
Stream3		myKernel3	
Stream4		myKernel4	

Note 1: Kernels must modify different parts of memory to avoid data races.
Note 2: With large kernels, overlapping computations may not help performance.

Streams

- There is another use for streams besides concurrent kernels:
  - Overlapping kernels with data movement.

- AMD GPUs have separate engines for:
  - Host->Device memcpys
  - Device->Host memcpys
  - Compute kernels.

- These three different operations can overlap without dividing the GPU’s resources.
  - The overlapping operations should be in separate, non-NULL, streams.
  - The host memory should be **pinned.**

Streams

Suppose we have 3 kernels which require moving data to and from the device:

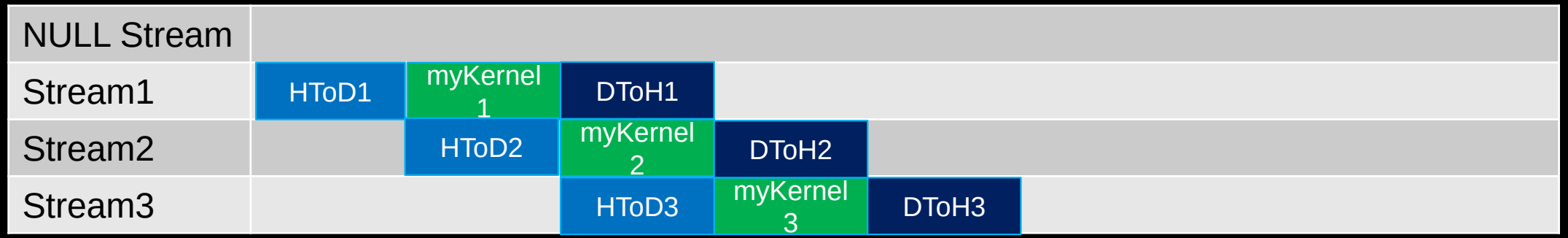
```
hipMemcpy(d_a1, h_a1, Nbytes, hipMemcpyHostToDevice));
hipMemcpy(d_a2, h_a2, Nbytes, hipMemcpyHostToDevice));
hipMemcpy(d_a3, h_a3, Nbytes, hipMemcpyHostToDevice));

myKernel1<<<blocks, threads, 0, 0>>>(N, d_a1);
myKernel2<<<blocks, threads, 0, 0>>>(N, d_a2);
myKernel3<<<blocks, threads, 0, 0>>>(N, d_a3);

hipMemcpy(h_a1, d_a1, Nbytes, hipMemcpyDeviceToHost);
hipMemcpy(h_a2, d_a2, Nbytes, hipMemcpyDeviceToHost);
hipMemcpy(h_a3, d_a3, Nbytes, hipMemcpyDeviceToHost);
```

NULL Stream	

Streams

Changing to asynchronous memcpys and using streams:

```
hipMemcpyAsync(d_a1, h_a1, Nbytes, hipMemcpyHostToDevice, stream1);
hipMemcpyAsync(d_a2, h_a2, Nbytes, hipMemcpyHostToDevice, stream2);
hipMemcpyAsync(d_a3, h_a3, Nbytes, hipMemcpyHostToDevice, stream3);

myKernel1<<<blocks, threads, 0, stream1>>>(N, d_a1);
myKernel2<<<blocks, threads, 0, stream2>>>(N, d_a2);
myKernel3<<<blocks, threads, 0, stream3>>>(N, d_a3);

hipMemcpyAsync(h_a1, d_a1, Nbytes, hipMemcpyDeviceToHost, stream1);
hipMemcpyAsync(h_a2, d_a2, Nbytes, hipMemcpyDeviceToHost, stream2);
hipMemcpyAsync(h_a3, d_a3, Nbytes, hipMemcpyDeviceToHost, stream3);
```



Synchronization

How do we coordinate execution on device streams with host execution? Need some synchronization points.

- `hipDeviceSynchronize();`
  - Heavy-duty sync point.
  - Blocks host until **all work** in **all device streams** has reported complete.

  - `hipStreamSynchronize(stream);`
  - Blocks host until **all work in stream** has reported complete.

Can a stream synchronize with another stream? For that we need ‘Events’:
https://rocm.docs.amd.com/projects/HIP/en/latest/.doxygen/docBin/html/group___event.html

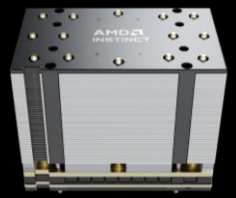
ROCm

ROCm is an open-source platform for GPU computing (including drivers, development tools, APIs, and libraries) on AMD GPUs.

- ROCm drivers allow the OS to communicate with the GPU hardware.
- ROCm libraries provide optimized routines for scientific computing and machine learning tasks, such as BLAS, FFT, etc.
- **ROCm is powered by AMD's HIP programming environment and runtime.**

ROCm is supported on AMD INSTINCT & certain RADEON GPUs.

For the full list, please visit https://rocm.docs.amd.com/en/latest/release/gpu_os_support.html#linux-supported-gpus



ROCm GPU Libraries

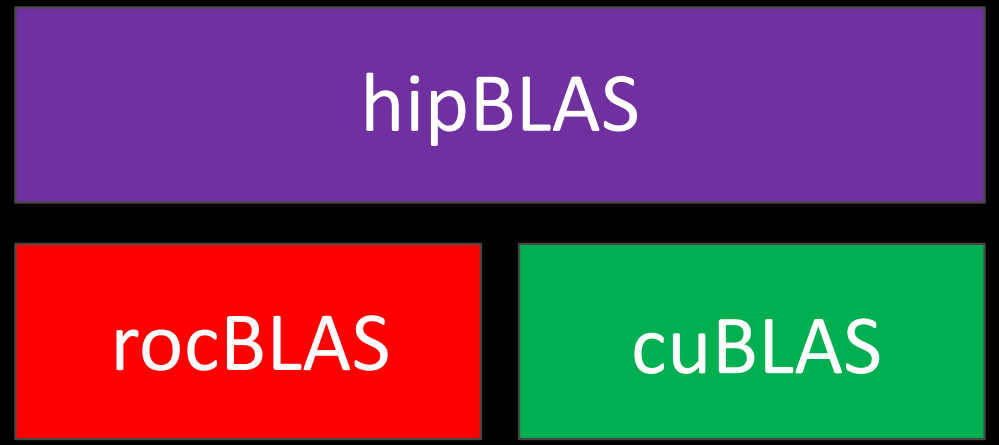
ROCm provides several GPU math libraries

- Typically, two versions:
  - roc* -> AMD GPU library, usually written in HIP
  - hip* -> Thin interface between roc* and Nvidia cu* library

When developing an application meant to target both CUDA and AMD devices, use the hip* libraries (portability)

When developing an application meant to target only AMD devices, may prefer the roc* library API (performance).

- Some roc* libraries perform **better** by using additional APIs not available in the cu* equivalents



AMD Math Library Equivalents: “Decoder Ring”

CUBLAS	**ROCBLAS**	Basic Linear Algebra Subroutines
CUFFT	**ROCFFT**	Fast Fourier Transforms
CURAND	**ROCRAND**	Random Number Generation
THRUST	**ROCTHRUST**	C++ Parallel Algorithms
CUB	**ROCPRIM**	Optimized Parallel Primitives

AMD Math Library Equivalents: “Decoder Ring”

CUSPARSE	ROCSPARSE	Sparse BLAS, SpMV, etc.
CUSOLVER	ROCSOLVER	Linear Solvers
AMGX	ROCALUTION	Solvers and preconditioners for sparse linear systems

GITHUB.COM/ROCM-DEVELOPER-TOOLS/HIP → HIP_PORTING_GUIDE.MD FOR A COMPLETE LIST

Querying the System

- rocminfo: Queries and displays information on the system's hardware
  - More info at: https://github.com/RadeonOpenCompute/rocminfo
- Querying ROCm version:
  - If you install ROCm in the standard location (/opt/rocm) version info is at: /opt/rocm/.info/version-dev
  - Can also run the command 'dkms status' and the ROCm version will be displayed
- rocm-smi: Queries and sets AMD GPU frequencies, power usage, and fan speeds
  - sudo privileges are needed to set frequencies and power limits
  - sudo privileges are not needed to query information
  - Get more info by running 'rocm-smi -h' or looking at: https://github.com/RadeonOpenCompute/ROC-smi

```
$ /opt/rocm/bin/rocm-smi
========================ROCm System Management Interface========================
================================================================================
GPU   Temp    AvgPwr   SCLK      MCLK     Fan    Perf     PwrCap   VRAM%   GPU%
1     38.0c   18.0W    1440Mhz   945Mhz   0.0%   manual   220.0W     0%    0%
================================================================================
==============================End of ROCm SMI Log ===============================
```

4. OpenMP ® on AMD GPUs

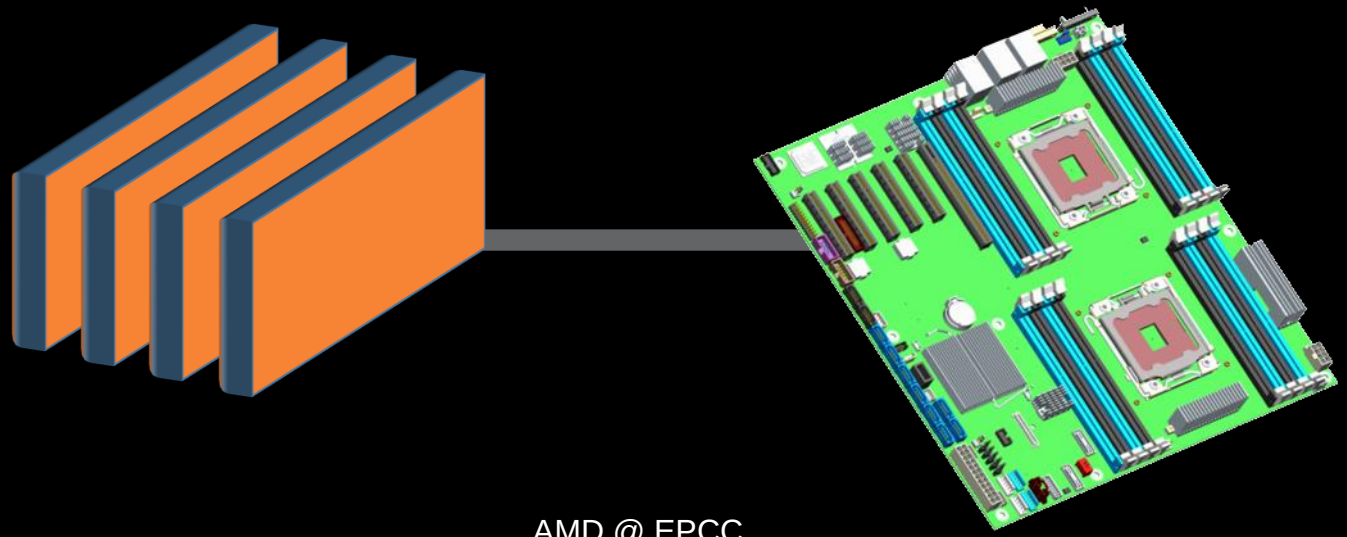
Enabling OpenMP® on AMD Hardware

		LLVM		GCC	
Compiler Module ➡		amdclang/aomp/clacc		gcc/og	
		Command	Flags	Command	Flags
Language	C	`amdclang clang`	`-fopenmp --offload-arch=<gfx###>`	`gcc`	`-fopenmp --foffload=-march=<gfx###>`
	C++	`amdclang++ clang++`	`-fopenmp --offload-arch=<gfx###>`	`g++`	`-fopenmp --foffload=-march=<gfx###>`
	Fortran	`amdflang flang`	`-fopenmp --offload-arch=<gfx###>`	`gfortran`	`-fopenmp --foffload=-march=<gfx###>`

Offloading Target (CPU/GPU/GCD)	Architecture <gfx###>
AMD MI300	`gfx942`
AMD MI200 Series	`gfx90a`
AMD MI100	`gfx908`
Native Host (CPU)	`-fopenmp-targets=amdgcn-amd-amdhsa`

OpenMP® Device Model

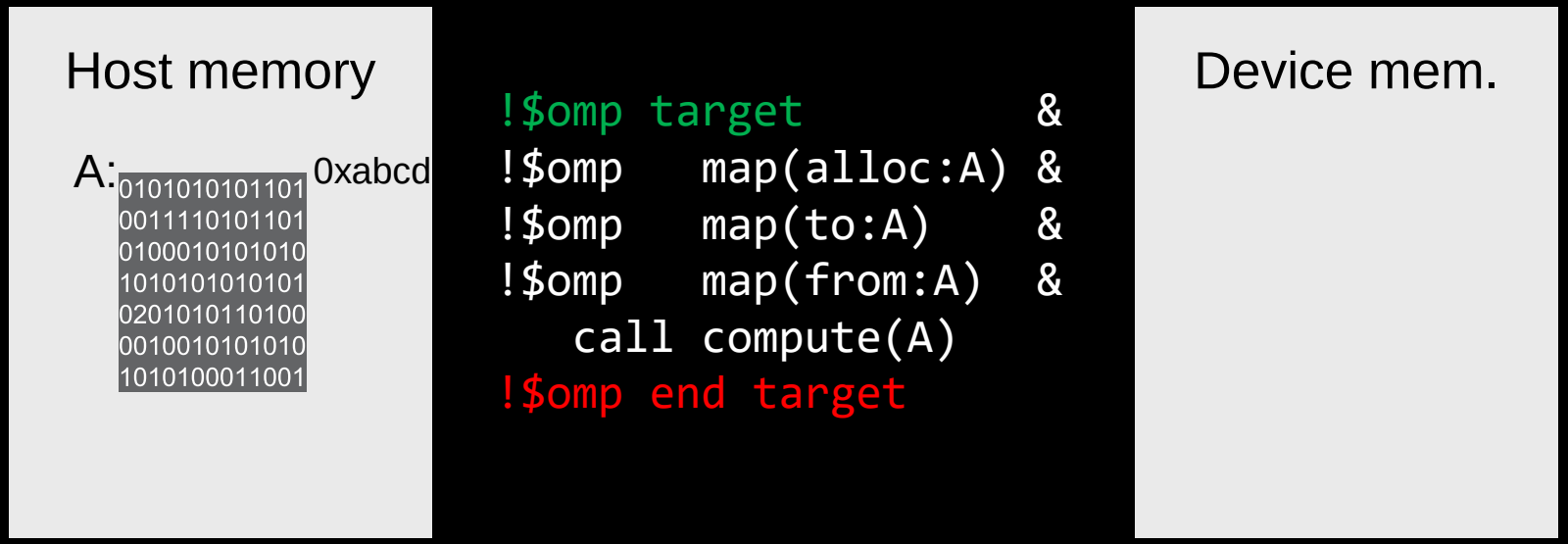
- As of version 4.0 the OpenMP API supports accelerators/coprocessors.
- Device model:
  - One host for “traditional” multi-threading
  - Multiple accelerators/coprocessors of the same kind for offloading
  - Devices are accessible through a device ID (from 0 to *n*-1 for *n* devices)
- OpenMP device model is *agnostic* of actual technology. In theory, devices only need to:
  - Send data to and receive data from the host
  - Perform computation upon request



OpenMP® Execution Model for Devices

Offload region and its data environment are bound to the lexical scope of the construct

- Data environment is created at the opening curly brace
- Data environment is automatically destroyed at the closing curly brace
- Data transfers (if needed) are done at the curly braces, too:
  - ❖ Upload data from the host to the target device at the opening curly brace.
  - ❖ Download data from the target device at the closing curly brace.



Running Example for this Presentation: saxpy

```
void saxpy() {
    float a, x[N], y[N];
    // left out initialization
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp parallel for firstprivate(a)
    for (int i = 0; i < N; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

This is the code we want to execute on a target device (i.e., GPU)

Don’t do this at home!
Use a BLAS library for this!

OpenMP® Device Constructs

- Transfer control *and data* from the host to the device
- Syntax (C/C++)

```
#pragma omp target [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran)

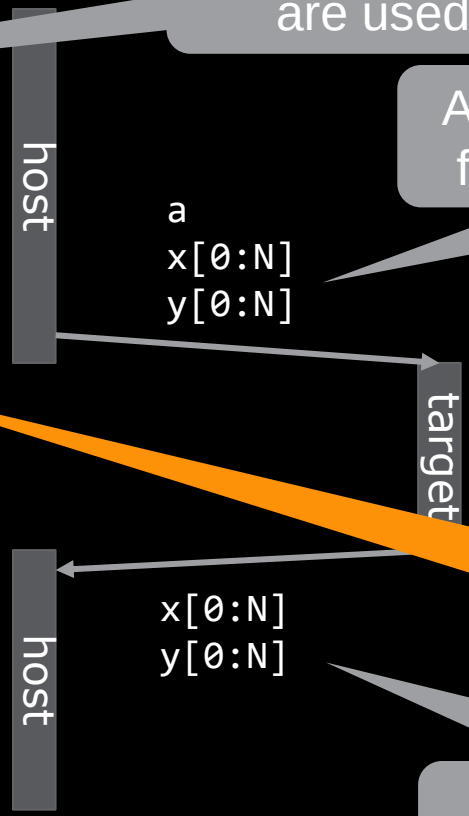
```
!$omp target [clause[[,] clause],…]
structured-block
!$omp end target
```

- Clauses

```
device(scalar-integer-expression)
map([{alloc | to | from | tofrom}:] list)
if(scalar-expr)
```

Example: saxpy

```
void saxpy() {
    float a, x[N], y[N];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp target “map(tofrom:y[0:N])”
    for (int i = 0; i < N; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```



The compiler identifies variables that are used in the `target` region.

All accessed arrays are copied from host to device and back.

Presence check: only transfer if not yet allocated on the device.

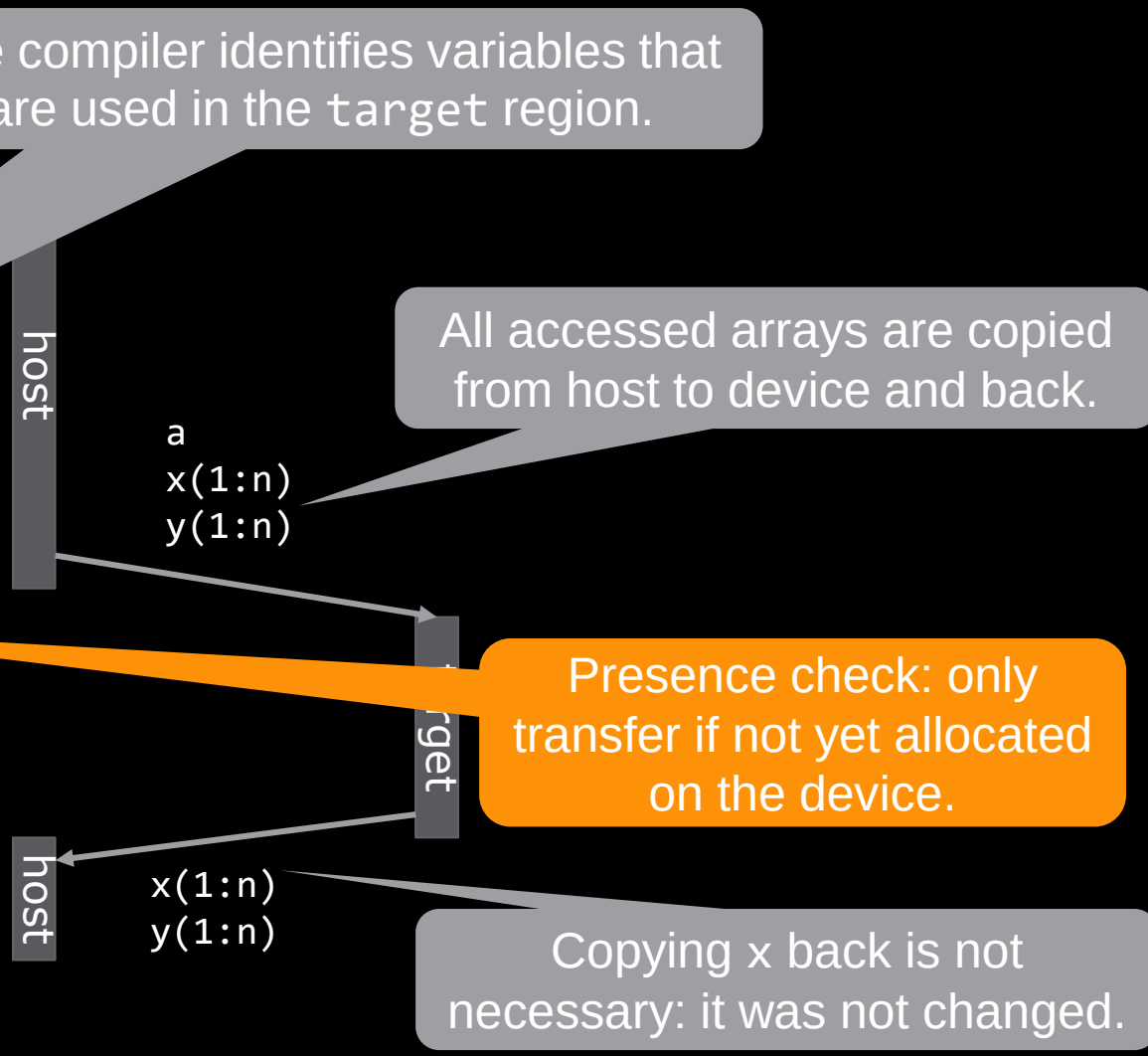
Copying x back is not necessary. It was not changed.

```
amdclang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

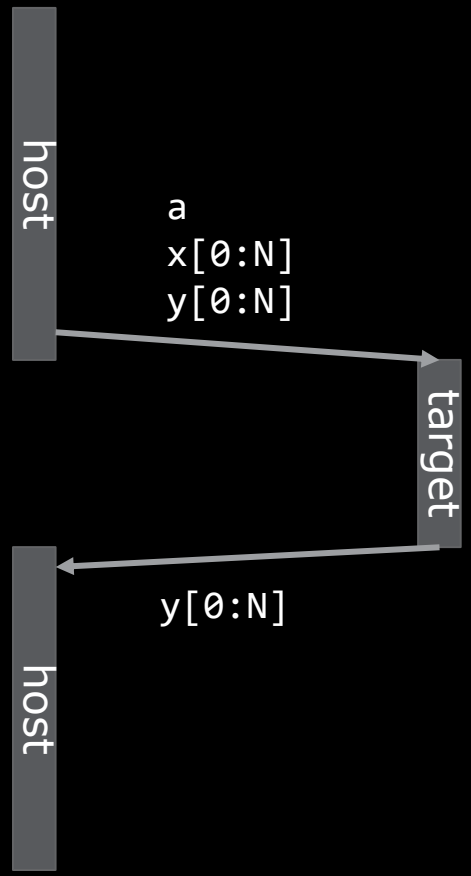
```
subroutine saxpy(a, x, y, n)
    use iso_fortran_env
    integer :: n, i
    real(kind=real32) :: a
    real(kind=real32), dimension(n) :: x
    real(kind=real32), dimension(n) :: y

    !$omp target “map(tofrom:y(1:n))”
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
    !$omp end target
end subroutine
```

```
amdflang -fopenmp --offload-arch=gfx90a ...
```



Example: saxpy

```
void saxpy() {
    double a, x[N], y[N];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp target map(to:x[0:N]) \
                       map(tofrom:y[0:N])
    for (int i = 0; i < N; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

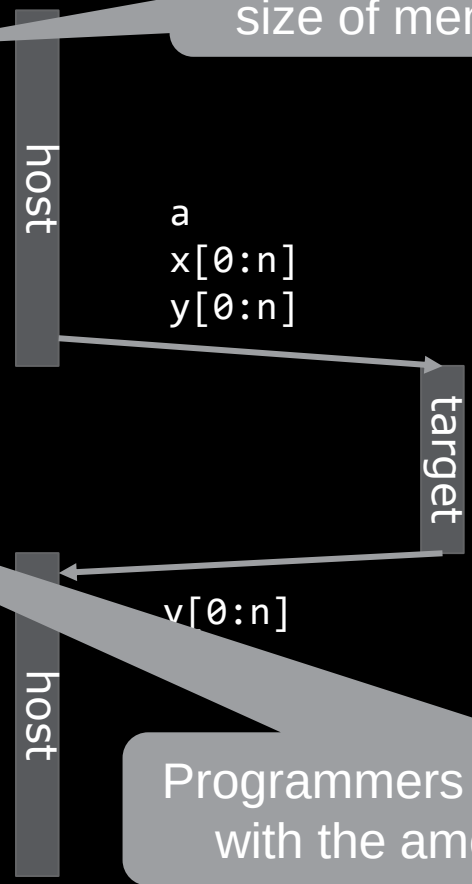



```
amdclang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int n) {
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp target map(to:x[0:n]) \
                       map(tofrom:y[0:n])
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

```
amdclang -fopenmp --offload-arch=gfx90a ...
```



The compiler cannot determine the size of memory behind the pointer.

Programmers have to help the compiler with the amount of data to transfer.

Creating Parallelism on the Target Device

- The `target` construct transfers the control flow to the target device
  - Transfer of control is sequential and synchronous.
  - This is intentional!

- OpenMP® separates offload and parallelism
  - Programmers need to explicitly create parallel regions on the target device.
  - In theory, this can be combined with any OpenMP construct.
  - In practice, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int n) {
#pragma omp target map(to:x[0:n]) \
                   map(tofrom(y[0:n])
#pragma omp parallel for simd
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i]   y[i];
    }
}
```

GPUs are multi-level devices: SIMD, threads, thread blocks

Create a team of threads to execute the loop in parallel using SIMD instructions.

```
clang -fopenmp --offload-arch=gfx90a
```

teams Construct

- Support multi-level parallel devices
- Syntax (C/C++):

```
#pragma omp teams [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran):

```
!$omp teams [clause[[,] clause],…]
structured-block
```

- Clauses

```
num_teams(integer-expression), thread_limit(integer-expression)
default(shared | firstprivate | private none)
private(list), firstprivate(list), shared(list), reduction(operator:list)
```

Optimizing Data Transfers is Key to Performance

- Connections between host and accelerator are typically lower-bandwidth, higher-latency interconnects
  - Bandwidth host memory: hundreds of GB/sec
  - Bandwidth accelerator memory: TB/sec
  - PCIe® Gen 4 bandwidth (16x): tens of GB/sec
- Unnecessary data transfers must be avoided, by
  - only transferring what is actually needed for the computation, and
  - making the lifetime of the data on the target device as long as possible.

Summary on OpenMP®

- AMD OpenMP® compilers can offload computation to AMD GPUs
  - ❖Good support for C and C++ languages
  - ❖Basic support for Fortran with active development underway
  - ❖Mature offload model w/ support for asynchronous offload/transfer
  - ❖Being used in production by many applications
- Backed by an Industry language standard
  - ❖Managed by the OpenMP® Architecture Review Board
- Portability across GPU platforms for core OpenMP® constructs
  - ❖Tightly integrates with OpenMP multi-threading on the host
- Composability across programming languages (C,C++,Fortran)
  - ❖Interoperable with some other GPU programming languages and libraries

AMD

5. Performance Portability Languages (ex. Kokkos)

Performance portable languages

Three P's critical for HPC application developer:

Performance **Portability** **Productivity**

Custom implementations for each new computer hardware vendor/type is not sustainable

For code maintenance , single-source application code is desirable

Department of Energy (DOE) has sponsored conferences and workshops on this topic

Two popular P3 languages have emerged in the DOE

- RAJA – LLNL C++ performance portability layer
  - Modular in structure with separation of compute and data management
  - Adaptable for how each application team implements in their code
- Kokkos – SNL C++ performance portable programming model
  - Comprehensive approach to performance portability
  - Parts being integrated into C++ standard

What is Kokkos, why Kokkos, and how does it work?

What

- Kokkos (κόκκος) is greek for “grains”, “seed” or “kernels”
- Developed at Sandia National Laboratory
  - Original purpose was to provide an abstraction layer for mathematical solvers
- Supports many backends including OpenMP® threading, CUDA, HIP, and others

Why

- Attractive P3 development language for C++ application
- provides a single source capability for C++ codes to run on a variety of parallel CPU and GPU architectures
- Kokkos is well-supported and relatively mature
- performs nearly as well or better than lower-level languages

How

- Library based on C++ templates
  - Libraries are quicker to implement and distribute
  - Eventually these techniques can migrate to compilers
- Easily integrated with CMake as external or in-line build

HIP backend support in Kokkos

Kokkos has long had a HIP backend for selected AMD Processors
- Both CPUs and GPUs

The Kokkos team has aggressively developed their implementation for new AMD systems coming online

In the Fall of 2022, the HIP backend was promoted to production status

Kokkos handles many of the unique attributes of the AMD GPUs for you

Kokkos abstractions for GPUs (and parallelism on CPUs)

- Two basic requirements for a GPU programming language
  - Actually, for any fine-grained parallel language that runs on either GPUs or CPUs

- **Execution capability** – this handles how to generate the execution code within a program to run on the target architecture. Generally, this is for loops, but may also include single lines of computation.
- **Memory handling** – the control of the allocation and movement of memory between the CPU and GPU or other memory locations.

- Kokkos, as a portability layer for various fine-grained programming languages, must have an abstract representation of these two requirements.

Execution and memory abstraction in Kokkos

Execution Spaces -- compute hardware where computations are done

- Execution Patterns
  - Simple loops -- parallel_for
  - Reductions -- parallel_reduce
  - Scans -- parallel_scan
- Execution Policies
  - Range policies -- basically index sets that need to be operated on
  - Team policies – grouping threads into teams as a subset of the execution space for hierarchical parallelism.

Memory Spaces – memory hardware where the data is stored

- Memory Layout
  - LayoutRight vs LayoutLeft or automatic conversion between the two for different execution spaces
- Memory Traits
  - atomic access, random access (shader memory), streaming stores

6. Overview of AMD Tools

AMD Tools Overview

1. **Hipify** tools: port your code from CUDA to HIP

2. **Debug** tools: make your code run on AMD hardware

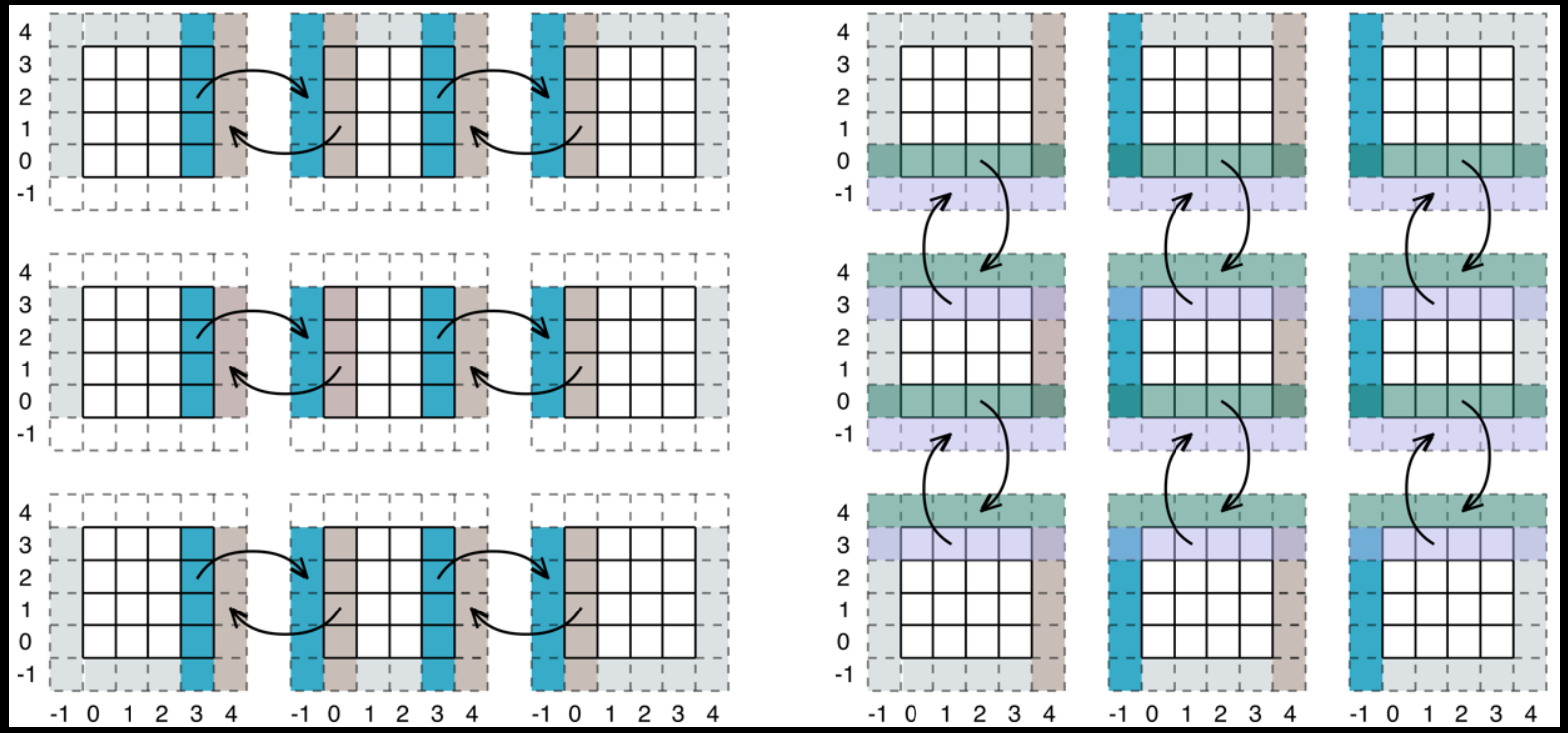
3. **Profile** tools: get initial idea on how to optimize your code

4. **Trace** tools: get deeper idea of what happens in your code

5. **Performance** tools: optimize specific kernels to leverage hardware capabilities

We will use the tools on: MPI Ghost Exchange Example

- A rectangular domain is partitioned into a 2D computational grid, distributed among MPI processes
- An initial solution in specified on a cell-wise basis, then advanced with a 5-point stencil averaging operator
- Halo cells are located along the boundary, and around MPI domains (*ghost cells*) when doing parallel runs
- Boundary conditions are of *outflow* type, enforced prior to ghost halo exchanges
- Example of 2-step halo exchange with 3x3 grid of processes, each owning a 4x4 subset of the mesh:



Available at:

https://github.com/amd/HPCTraining Examples.git

Robey, Robert, and Yuliana Zamora. Parallel and high-performance computing. Manning, 2021.

1. HIPIFY TOOLS

hipify-perl

- Easiest to use; point at a directory and it will hipify CUDA code
- It is located in `${HIP_PATH}/bin/`
- It replaces cuda with hip, `sed -e 's/cuda/hip/g'`, (e.g., cudaMemcpy becomes hipMemcpy)
- Recommended for quick scans of projects
- It will not translate if it does not recognize a CUDA call and it will report it

hipify-clang

- More robust translation of the code
- Generates warnings and assistance for additional analysis
- High quality translation, particularly for cases where the user is familiar with the make system
- Build from source (**needs clang compiler**)
- Hipification requires same headers that would be needed to compile it with clang

other tools

- Recursive directory tools (hipconvertinplace.sh, hipconvertinplace-perl.sh, hipexamine.sh, hipexamine-perl.sh)
- Header file interception layer (HIPIFLY)

Gotchas

- Hipify tools are not running your application, or checking correctness
- Code relying on specific NVIDIA hardware aspects (e.g., warp size = 32) may need attention after conversion (grep for "32" *just in case)*
- Certain functions may not have a correspondent HIP version
- Hipifying can’t handle inline PTX assembly or CUDA intrinsics
- Hipify-clang modifies the code **in place**, so better to make a copy of the code before running the tool
- None of the tools convert your build system script such as CMAKE or whatever else you use.
  The user is responsible to find the appropriate flags and paths to build the new converted HIP code

Notes

- Hipify-perl and hipify-clang **can** both convert library calls (i.e. cuBLAS becomes hipBLAS)
- CMake v3.21 onwards can be used to automatically set up basic compilation flags by using enable_language(HIP), supports CMAKE_HIP_ARCHITECTURES for setting devices to build for

Portability Blog Post: https://rocm.blogs.amd.com/software-tools-optimization/hipify/README.html

HIPIFLY: Intercept API method to choose GPU backend

- Enable running existing code on different backends with single header
- Can change between targeting CUDA and ROCm in one place
- Only works if no difference between API calls
- Existing code cannot use any CUDA specific hard coded values
- Performance needs to be evaluated on a case by case basis

```
#ifndef CUDA_TO_HIP_H
#define CUDA_TO_HIP_H

#include <hip/hip_runtime.h>

#define WARPSIZE 64
static constexpr int maxWarpsPerBlock = 1024/WARPSIZE;

#define CUFFT_D2Z HIPFFT_D2Z
#define CUFFT_FORWARD HIPFFT_FORWARD
#define CUFFT_INVERSE HIPFFT_BACKWARD
#define CUFFT_Z2D HIPFFT_Z2D
#define CUFFT_Z2Z HIPFFT_Z2Z

#define cudaDeviceSynchronize hipDeviceSynchronize
#define cudaError hipError_t
#define cudaError_t hipError_t
#define cudaErrorInsufficientDriver hipErrorInsufficientDriver
#define cudaErrorNoDevice hipErrorNoDevice
#define cudaEvent_t hipEvent_t
#define cudaEventCreate hipEventCreate
#define cudaEventElapsedTime hipEventElapsedTime
#define cudaEventRecord hipEventRecord
#define cudaEventSynchronize hipEventSynchronize
#define cudaFree hipFree
#define cudaFreeHost hipHostFree
#define cudaGetDevice hipGetDevice
#define cudaGetDeviceCount hipGetDeviceCount
#define cudaGetErrorString hipGetErrorString
#define cudaGetLastError hipGetLastError
#define cudaHostAlloc hipHostMalloc
#define cudaHostAllocDefault hipHostMallocDefault
#define cudaMalloc hipMalloc
#define cudaMemcpy hipMemcpy
#define cudaMemcpyAsync hipMemcpyAsync
#define cudaMemcpyDeviceToHost hipMemcpyDeviceToHost
#define cudaMemcpyDeviceToDevice hipMemcpyDeviceToDevice
#define cudaMemcpyHostToDevice hipMemcpyHostToDevice
#define cudaMemGetInfo hipMemGetInfo
#define cudaMemset hipMemset
#define cudaReadModeElementType hipReadModeElementType
```

Link to the header file:
https://github.com/amd/HPCTrainingExamples/blob/main/hipifly/vector_add/src/cuda_to_hip.h

Example: hipify using hipify-perl

1. Load ROCm module: `module load rocm`

2. Do:

```
hipify-perl GhostExchange.cu > GhostExchange.hip
```

the code is in:
HPCTrainingExamples/MPI-examples/GhostExchange/GhostExchange_ArrayAssign_HIP/Ver1Cuda

2. DEBUG TOOLS

Rocgdb

AMD ROCm ™ **source-level debugger for Linux®**

- based on the GNU Debugger (GDB)
  - tracks upstream GDB master
  - standard GDB commands for both CPU and GPU debugging

- considered a prototype
  - focus on source line debugging
  - no symbolic variable debugging yet

As GDB fork it can be used with other tools that use GDB as backend

Let’s use Rocgdb to debug our Ghost_Exchange hip code

The code in `HPCTrainingExamples/MPI-examples/GhostExchange/GhostExchange_ArrayAssign_HIP/Ver1WithBug` will produce an error: let’s go track it down!

- `module load rocm amdclang`
- `mkdir build && cd build`
- `cmake -DCMAKE_BUILD_TYPE=Debug ..`
- `make VERBOSE=1`
- `export HSA_XNACK=1`

Considering a small problem size for debugging

- `mpirun -np 4 ./GhostExchange -x 4 -y 1 -i 4 -j 4 -h 1 -t -c -I 30`

The above command will produce a 4x4 grid (16 cells) partitioned among 4 MPI ranks, the partitioning is done with vertical strips, rank 0 gets the left most strip, rank 1 the next and so on.

Let's look at some output

process 0 process 1 process 2 process 3

How it should be

Initial State

	-1	0	1	-1	0	1	-1	0	1	-1	0	1
4:	400.3	400.3	400.4	400.3	400.4	400.5	400.4	400.5	400.6	400.5	400.6	400.6
3:	400.3	400.3	400.4	400.3	400.4	400.5	400.4	400.5	400.6	400.5	400.6	400.6
2:	400.2	400.2	400.3	400.2	400.3	400.4	400.3	400.4	400.5	400.4	400.5	400.5
1:	400.1	400.1	400.2	400.1	400.2	400.3	400.2	400.3	400.4	400.3	400.4	400.4
0:	400.0	400.0	400.1	400.0	400.1	400.2	400.1	400.2	400.3	400.2	400.3	400.3
-1:	400.0	400.0	400.1	400.0	400.1	400.2	400.1	400.2	400.3	400.2	400.3	400.3

Values at cells owned by process 0

How it is instead

Initial State

	-1	0	1	-1	0	1	-1	0	1	-1	0	1
4:	5.0	5.0	6.0	5.0	6.0	7.0	6.0	7.0	8.0	7.0	8.0	8.0
3:	5.0	5.0	6.0	5.0	6.0	7.0	6.0	7.0	8.0	7.0	8.0	8.0
2:	5.0	5.0	6.0	5.0	6.0	7.0	6.0	7.0	8.0	7.0	8.0	8.0
1:	400.1	400.1	400.2	400.1	400.2	400.3	400.2	400.3	400.4	400.3	400.4	400.4
0:	400.0	400.0	400.1	400.0	400.1	400.2	400.1	400.2	400.3	400.2	400.3	400.3
-1:	400.0	400.0	400.1	400.0	400.1	400.2	400.1	400.2	400.3	400.2	400.3	400.3

Owned values are not initialized correctly

Let’s use Rocgdb to identify where the problem is

- `rocgdb -tui --args GhostExchange -x 1 -y 1 -i 4 -j 4 -h 1 -t -c -I 30`
  (it is not an MPI problem so run in serial for simplicity)

- `(gdb) b 202` places a break point here `if(rank==0) printf(“Initial State \n”);`

- place two more break points after the init kernels and the synchronization:
  `(gdb) b 188`
  `(gdb) b 197`
  NOTE: if compiling in optimized mode (-O3) the compiler may not allow you to place breakpoints on *empty* lines.
- to see the breakpoints, do: `(gdb) i b`, show the breakpoints locations and their `Num`

- you can delete a break point by doing: `(gdb) d Num` (Num for a breakpoint is shown by the above command)

- to run the debugger just do: `(gdb) r`
- to step to the next line: `(gdb) n`
- to continue after a break point: `(gdb) c`

Let's use Rocgdb to identify where the problem is

- you can print to terminal the entries of the vector with: `(gdb) p x[2][0]` (the first entry is the y in the x array)
  NOTE: this also works for array allocated on *device*.

- printing at breakpoint 198 we see that the value of `x[2][0]` has not been changed from a 5 to a 400.2
  therefore, the error is somewhere close to `init_core2`

- to further investigate, let's get info on the thread grid we are using to run `init_core2` with:
  1. `(gdb) b 22:` places a breakpoint inside `init_core2` at
  2. `(gdb) i dispatches` shows the thread grid information:

```
(gdb) i dispatches
  Id   Target Id                         Grid     Workgroup Fence   Kernel Function
* 1    AMDGPU Dispatch 1:4:1 (PKID 3) [64,2,1] [64,2,1]  B|Aa|Ra init_core2(double**, int, int, int, int, int, int, int, int, int)
```

The blocks in the y-direction in the grid, only have 2 threads, meaning that `tidy` only has 0 and 1 as values, when the number of cells in the y-direction is actually 4, so the values with index 2 and 3 are not initialize by `init_core2`

Error in the block size definition:
`dim3 block2(64, 2, 1);`

3. PROFILE TOOLS

Rocprof

ROC-profiler (rocprof) is the **command line front-end for AMD's GPU profiling libraries**

- provides tracing of **GPU kernels, HIP API, HSA API, and Copy activity**
- JSON traces can be viewed in Perfetto (type `https://ui.perfetto.dev/` on Chrome browser)
- can be used to collect hardware counters (with additional overhead)
- tool distributed with ROCm (like Rocgdb)
  - To get help: `${ROCM_PATH}/bin/rocprof -h`

Use Rocprof to get kernel information

rocprof can collect kernel(s) execution stats:

```
rocprof --stats --basenames on ./GhostExchange -x 1 -y 1 -i 20000 -j 20000 -h 2 -t -c -I 100
```

This will output two csv files:

- results.csv: information per each call of the kernel
- results.stats.csv: statistics grouped by each kernel

Content of results.stats.csv to see the list of GPU kernels with their durations (in nano seconds) and percentage of total GPU time:

```
"Name","Calls","TotalDurationNs","AverageNs","Percentage"
"blur",100,7419204646,74192046,93.55898042711705
"init_core",1,494436570,494436570,6.235032403375073
"enforce_bcs_left",101,11131533,110213,0.1403728469240027
"enforce_bcs_rght",101,2339840,23166,0.029506268556779958
"enforce_bcs_top",101,1482880,14681,0.01869967840428314
"enforce_bcs_bot",101,1357920,13444,0.01712388547876036
"init_core2",1,16320,16320,0.00020580138079810966
"__amd_rocclr_initHeap.kd",1,6240,6240,7.868876324633605e-05
```

In a spreadsheet viewer, it is easier to read:
from this information we see that the `blur` and `init_core` kernels are the ones that take up the most time, hence they are good candidates for optimization

Name	Calls	TotalDurationNs	AverageNs	Percentage
blur	100	7419204646	74192046	93.55898043
init_core	1	494436570	494436570	6.235032403
enforce_bcs_left	101	11131533	110213	0.140372847
enforce_bcs_rght	101	2339840	23166	0.029506269
enforce_bcs_top	101	1482880	14681	0.018699678
enforce_bcs_bot	101	1357920	13444	0.017123885
init_core2	1	16320	16320	0.000205801
__amd_rocclr_initHeap.kd	1	6240	6240	7.87E-05

Use Rocprof to get application traces

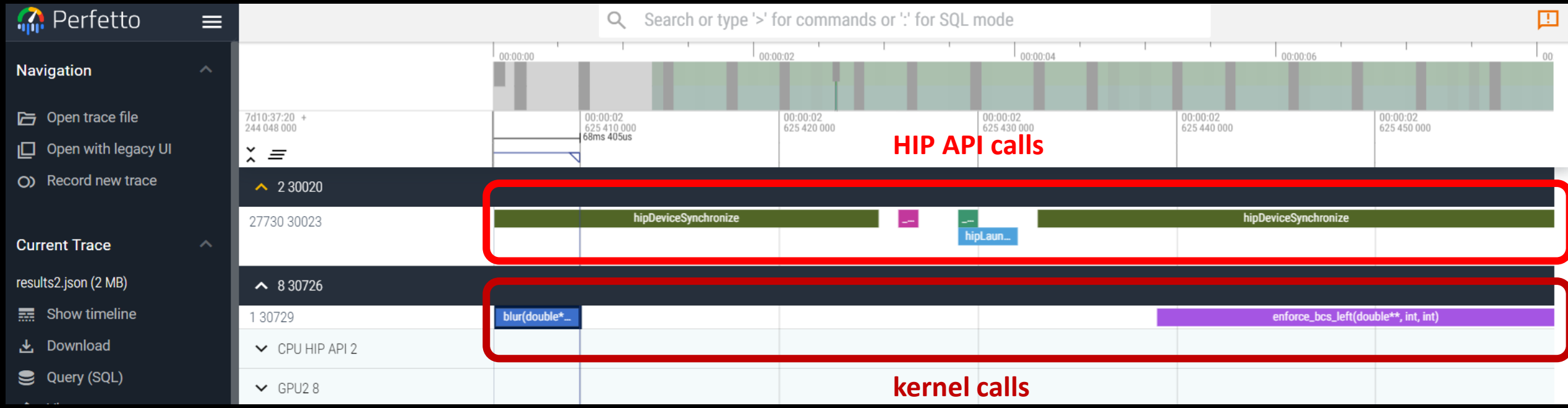
`rocprof` can collect a variety of trace event types, and generate timelines in JSON format for use with Perfetto:

Trace Event	rocprof Trace Mode
HIP API call	`--hip-trace`
GPU kernels	`--hip-trace`
host ←→ device memory copies	`--hip-trace`
CPU HSA Calls	`--hsa-trace`
user code markers	`--roctx-trace`
combine HIP/HSA APIs and GPU	`--sys-trace`

Modes can be combined, such as: `--hip-trace --hsa-trace`

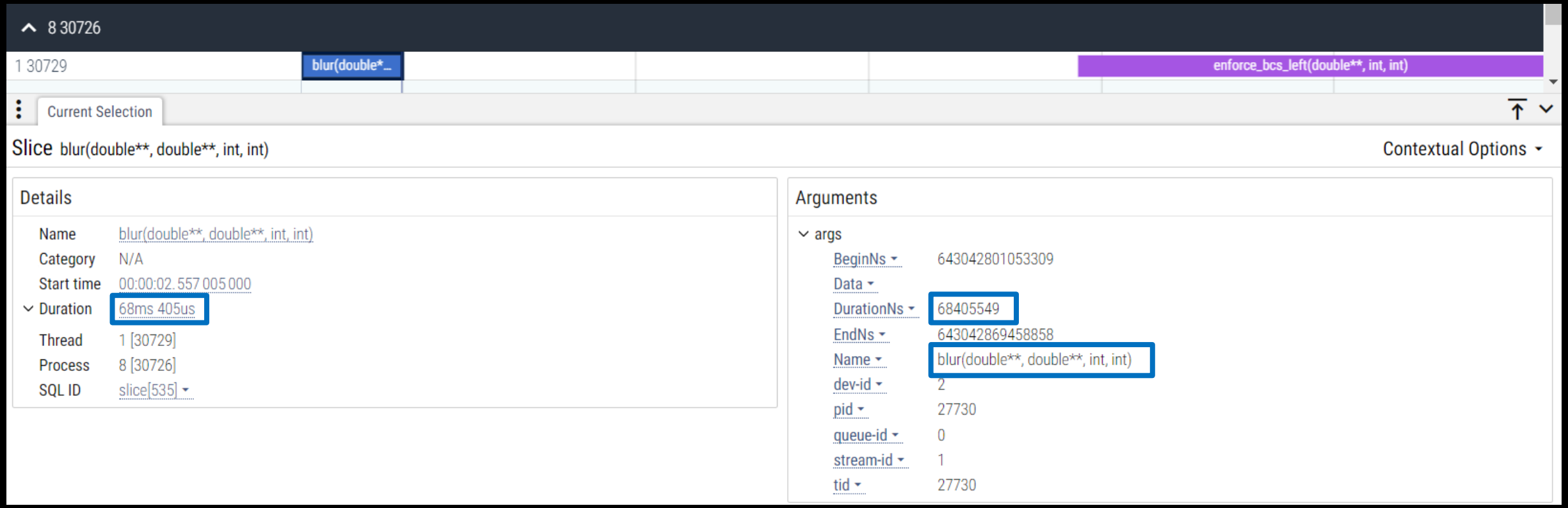
Use Rocprof to get application traces

```
rocprof --hip-trace  ./GhostExchange -x 1 -y 1 -i 20000 -j 20000 -h 2 -t -c -I 100
```



Use Rocprof to get application traces

Clicking on the kernel, you can get detailed information about it:



Other features provided by Rocprof

1. Collect user defined regions or markers using rocTX

Annotate code with roctx regions:

```
#include <roctx.h>
roctxRangePush(“Region");
// some function
roctxRangePop();
```

Annotate code with roctx markers:

```
roctxMark("start of some code");
// some_code
roctxMark("end of some code");
```

2. Collect hardware counters: you can inspect what the counters are by doing

```
rocprof --list-basic
rocprof --list-derived
```

Then, specify counters in a counter file. For example:

```
rocprof -i rocprof_counters.txt  ./GhostExchange -x 1 -y 1 -i 20000 -j 20000 -h 2 -t -c -I 100
```

4. TRACE TOOLS

Omnitrace

AMD Research holistic application analysis and tracing tool

- provides a holistic view of CPU, GPU, and system activity (**MPI**, PAPI, Kokkos, DynInst, Timemory, AMDuProf, etc.)
- can perform binary rewrite to be used by Omnitrace (does pre-instrumentation of the executable)
- proto files can be viewed in Perfetto (type `https://ui.perfetto.dev/` on Chrome browser)
- can be used to collect hardware counters (through Rocprof or PAPI)
- **NOT** part of the ROCm stack, so it needs to be installed separately

  https://github.com/amd/HPCTrainingDock

Use Omnitrace to visualize MPI calls

1. First, create the Omnitrace configuration file:

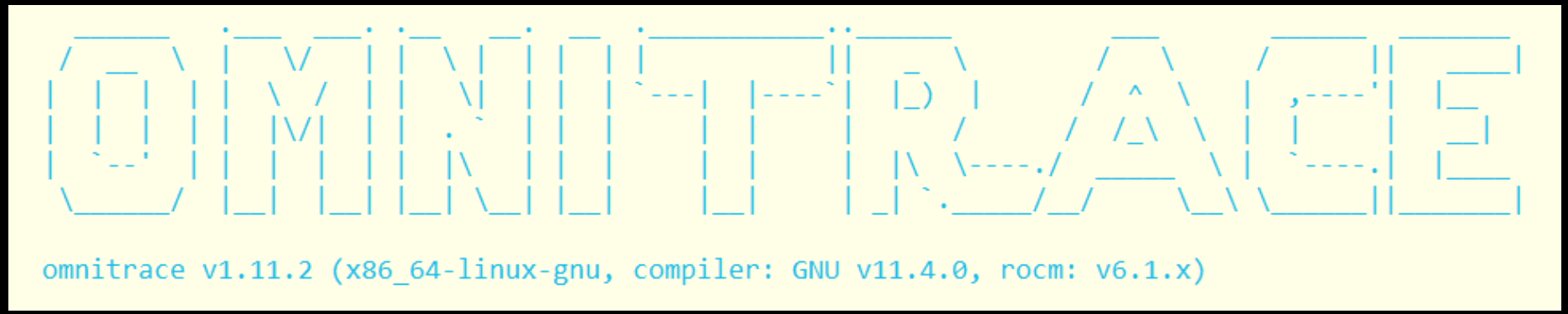
```
module load omnitrace
omnitrace-avail -G ~/.omnitrace.cfg
export OMNITRACE_CONFIG_FILE=~/.omnitrace.cfg
```

2. Then, create the instrumented binary:

```
omnitrace-instrument -o ./GhostExchange.inst -- ./GhostExchange
```

3. Run the instrumented binary to see MPI activity on Perfetto:

```
mpirun -np 4 omnitrace-run -- ./GhostExchange.inst  -x 4 -y 1 -i 20000 -j 20000 -h 2 -t -c -I 100
```



Use Omnitrace to visualize MPI activity

With Omnitrace, you can keep track of MPI overhead on Perfetto, look for the `perfetto-trace-<procID>.proto` files:

Useful Omnitrace config flag options

```
# auto-generated by omnitrace-avail (version 1.11.2) on 2024-05-22 @ 11:30

OMNITRACE_CONFIG_FILE                                =
OMNITRACE_TRACE                                      = true
OMNITRACE_ROCTRACER_HSA_ACTIVITY                     = true
OMNITRACE_ROCTRACER_HSA_API                          = true
OMNITRACE_PROFILE                                    = true
OMNITRACE_USE_SAMPLING                               = false
OMNITRACE_USE_ROCTRACER                              = true
OMNITRACE_USE_ROCM_SMI                               = true
OMNITRACE_ROCM_EVENTS                                = VALUUtilization,FetchSize
OMNITRACE_USE_ROCPROFILER                            = true
OMNITRACE_USE_ROCTX                                  = true
OMNITRACE_SAMPLING_CPUS                              = 0
OMNITRACE_SAMPLING_GPUS                              = 0
OMNITRACE_TIMEMORY_COMPONENTS                        = wall_clock
```

New presentation on Omnitrace done on May 29 2024 for the Oak Ridge Leadership Computing Facility (OLCF) (look for "Omnitrace By Example"): https://docs.olcf.ornl.gov/training/training_archive.html

5. PERFORMANCE TOOLS

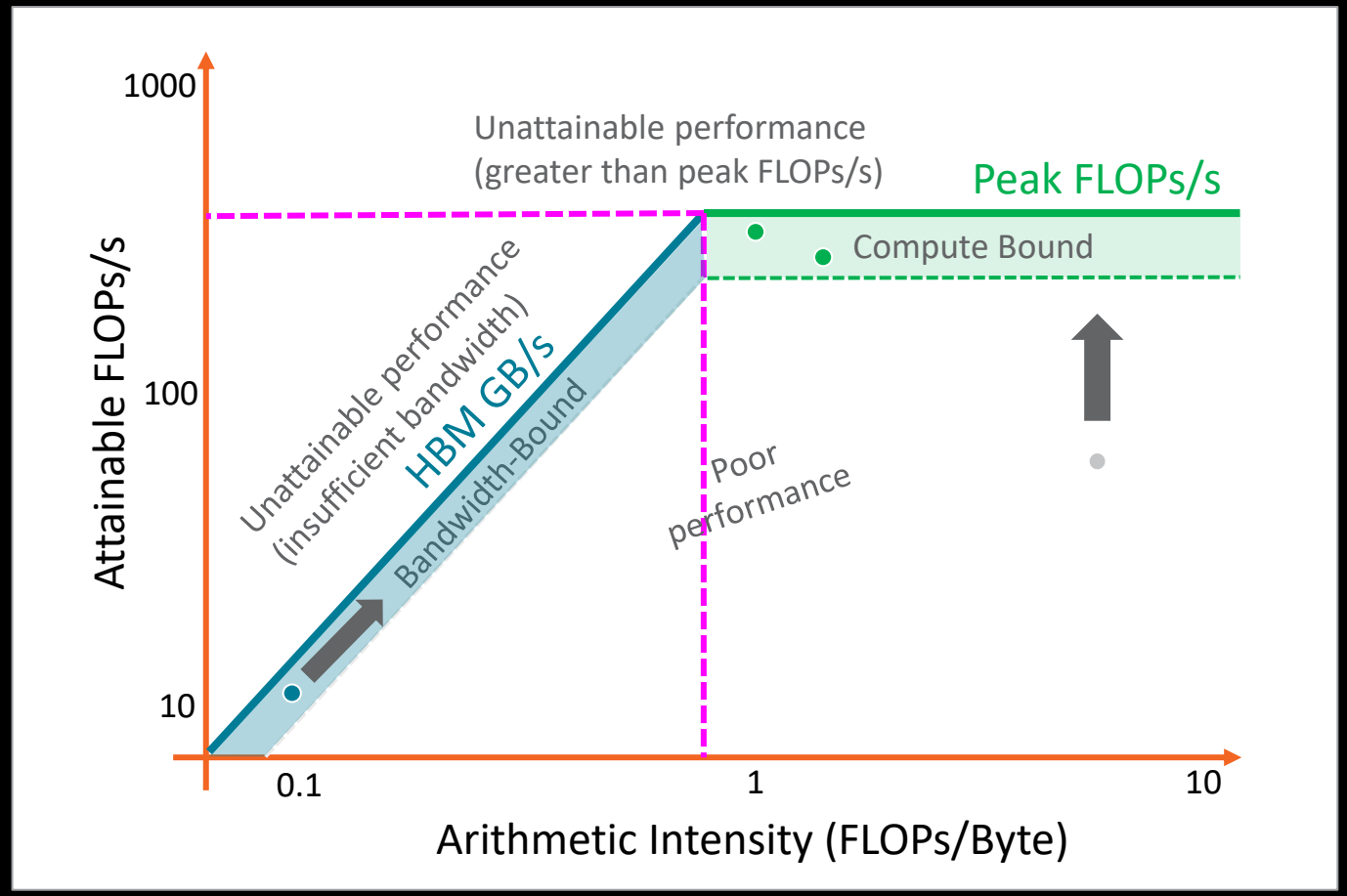
Omniperf

AMD performance analysis tool. Most notable features:

- roofline analysis to quantify performance of kernels based on hardware limits
- kernel comparison to quantify improvements an visualize their impact on hardware memory
- provides automated collection of hardware counters
- supports speed of light and memory chart
- **NOT** part of the ROCm stack, so it needs to be installed separately

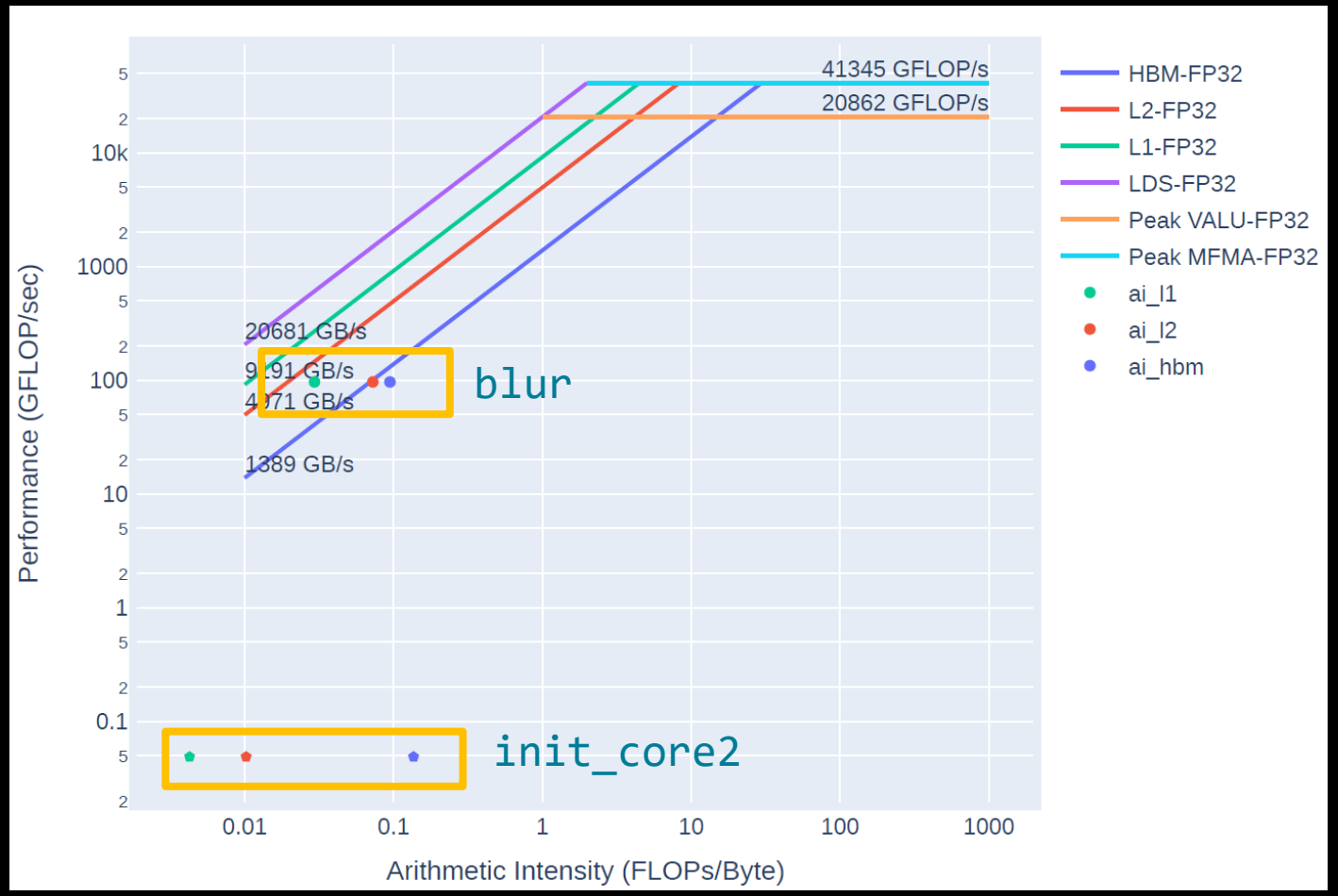
What is a roofline?

- Attainable FLOPs/s =
  - $min \begin{cases} Peak\ FLOPs/s \\ AI * Peak\ GB/s \end{cases}$
- Machine Balance:
  - Where $AI = \frac{Peak\ FLOPs/s}{Peak\ GB/s}$
- Five Performance Regions:
  - Unattainable Compute
  - Unattainable Bandwidth
  - Compute Bound
  - Bandwidth Bound
  - Poor Performance



Visualize rooflines with Omniperf

```
module load omniperf
unset ROOFLINE_BIN
omniperf profile -n rooflines_PDF --roof-only --kernel-names -- ./GhostExchange -x 1 -y 1 -i 20000 -j 20000 -h 2 -t -c -I 100
```



Compare performance of kernels with Omniperf

To generate profiling data do:

```
omniperf profile -n v1 --no-roof -- ./GhostExchange -x 1 -y 1 -i 20000 -j 20000 -h 2 -t -c -I 100
```

Modify the `init_core2` kernel grid:

```
dim3 block2(256, 4, 1);
```

Then compile a generate the profiling data for this new version:

```
omniperf profile -n v2 --no-roof -- ./GhostExchange -x 1 -y 1 -i 20000 -j 20000 -h 2 -t -c -I 100
```

You can compare the two versions by using `omniperf analyze`:

```
omniperf analyze -p workloads/v1/MI200 -p workloads/v2/MI200  --dispatch 1 --block 7.1.0 7.1.1 7.1.2
7.1.0: Grid size    7.1.1: Workgroup size   7.1.2: Total Wavefronts
```

New presentation on Omniperf done on April 25 2024 for HLRS in Germany:
fs.hlrs.de/projects/par/events/2024/GPU-AMD/day4/Introdution to omniperf.mp4

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