Introduction to Portable GPU Programming

Dr. Joe Schoonover
Fluid Numerics

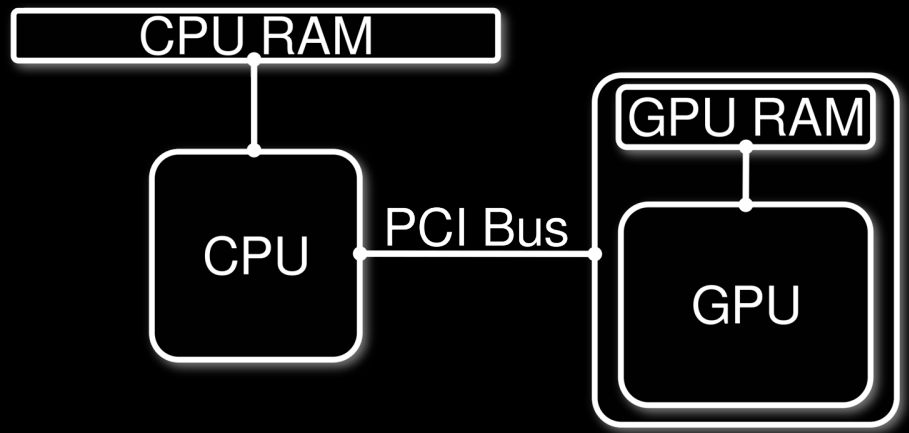
AMD
together we advance_

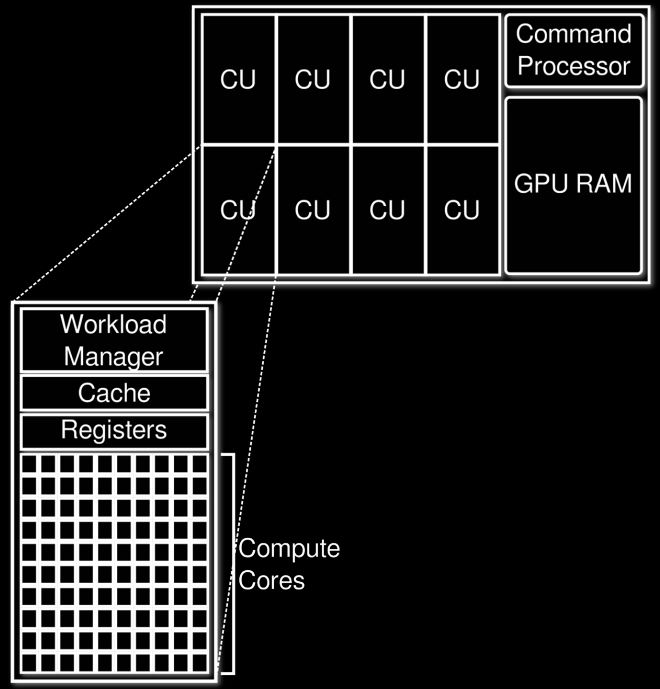
Topics

You will learn about …

- A model for thinking about GPU hardware and GPU accelerated platforms
- AMD GPU architecture
- The ROCm Software ecosystem
- Programming with HIP & HIPFort
- Programming with OpenMP
- Nvidia to AMD porting strategies

GPU Accelerated Platforms

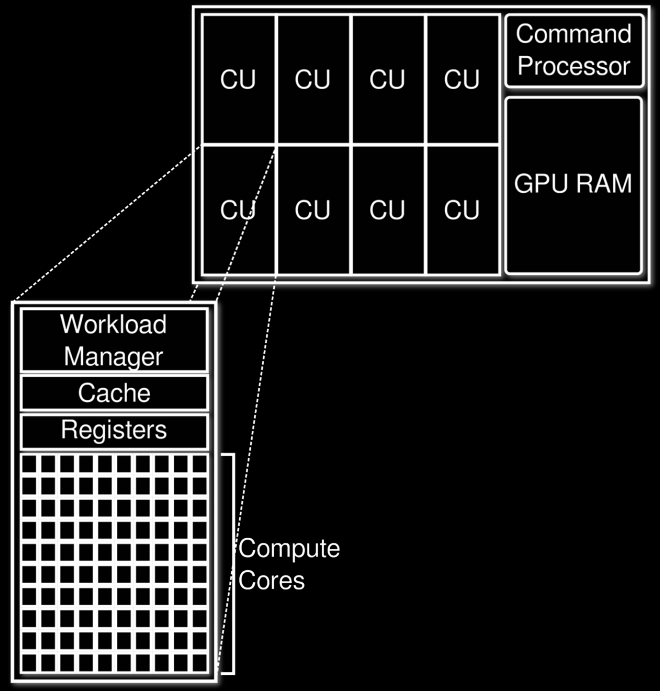




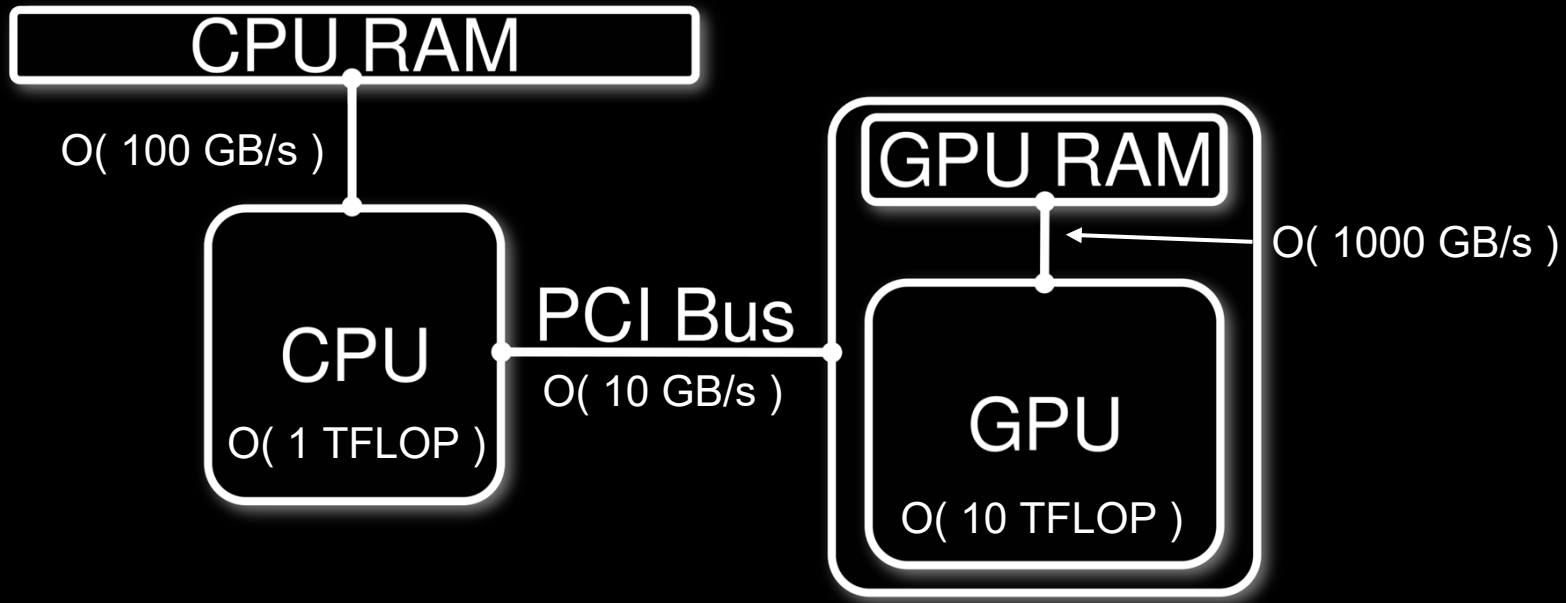
GPU Accelerated Platforms

Terminology

NVIDIA	AMD
Workload Distributor	Command Processor
Streaming Multiprocessor	Compute Unit
Warp	Wavefront

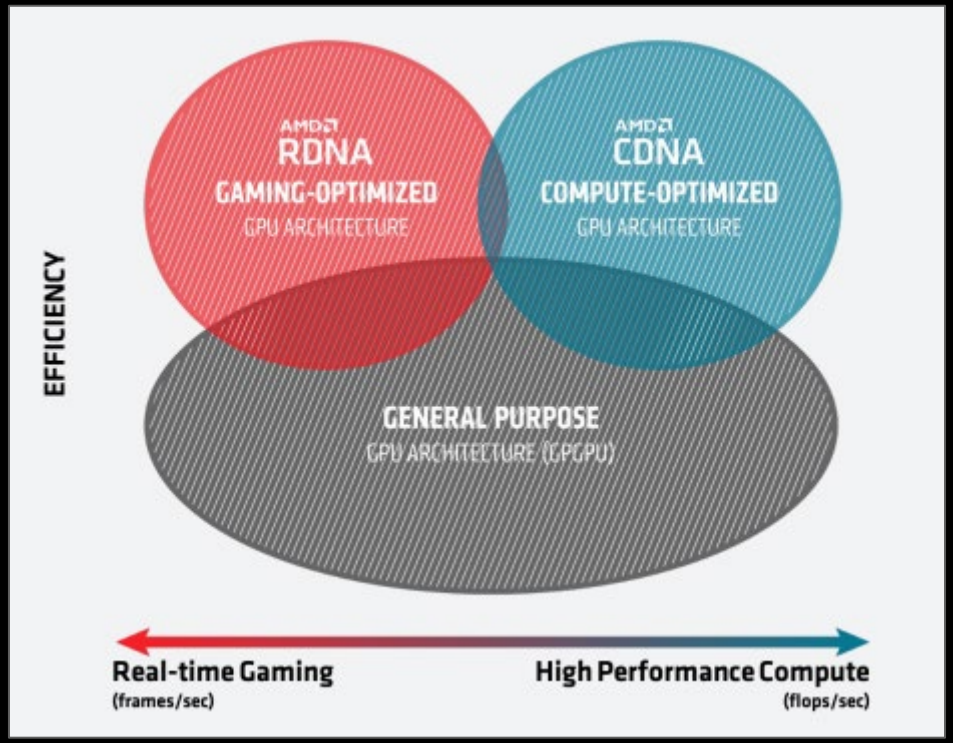


GPU Accelerated Platforms

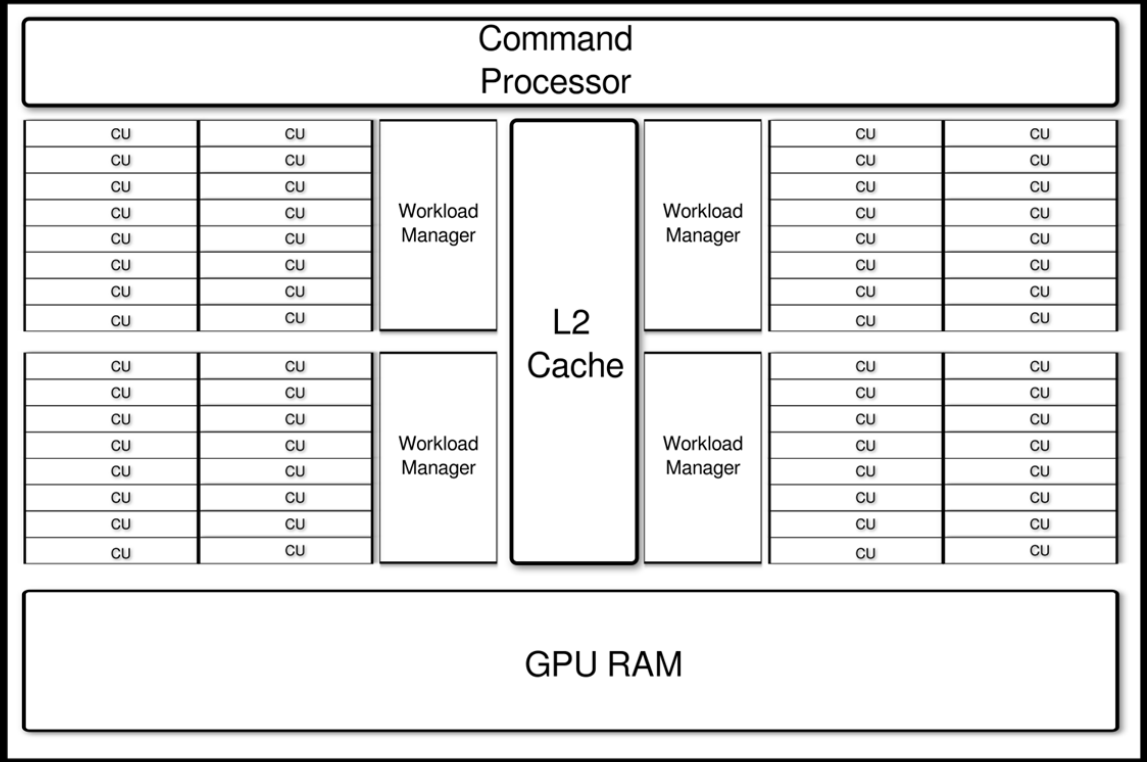


AMD GPU Hardware

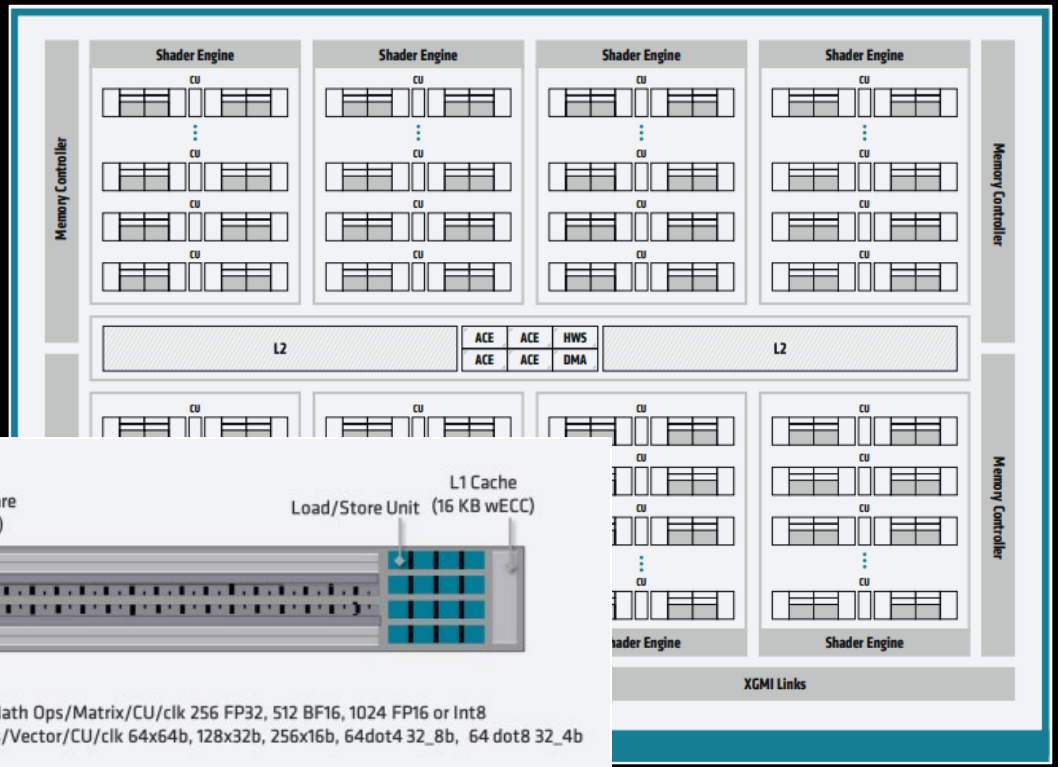
AMD Graphics Core Next (GCN) GPUs

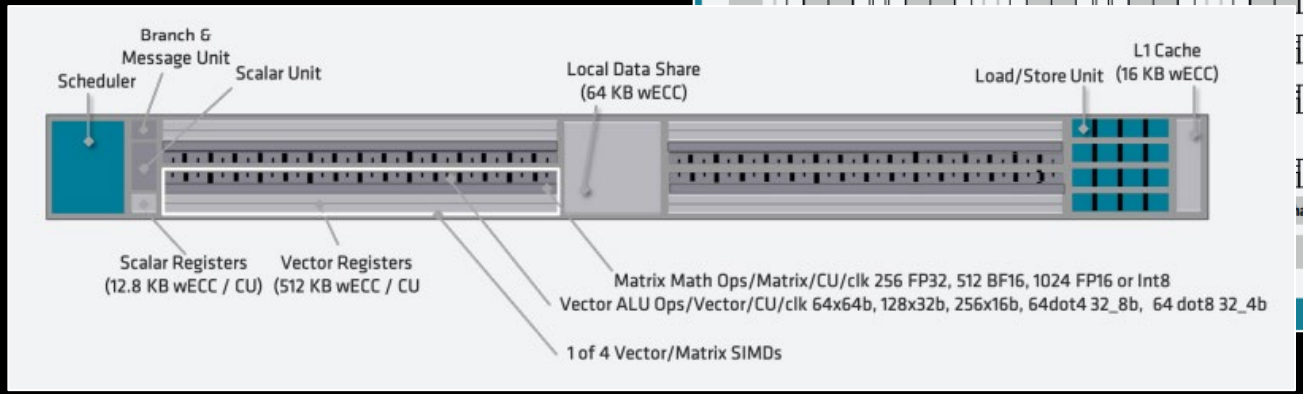


AMD Graphics Core Next (GCN) GPUs

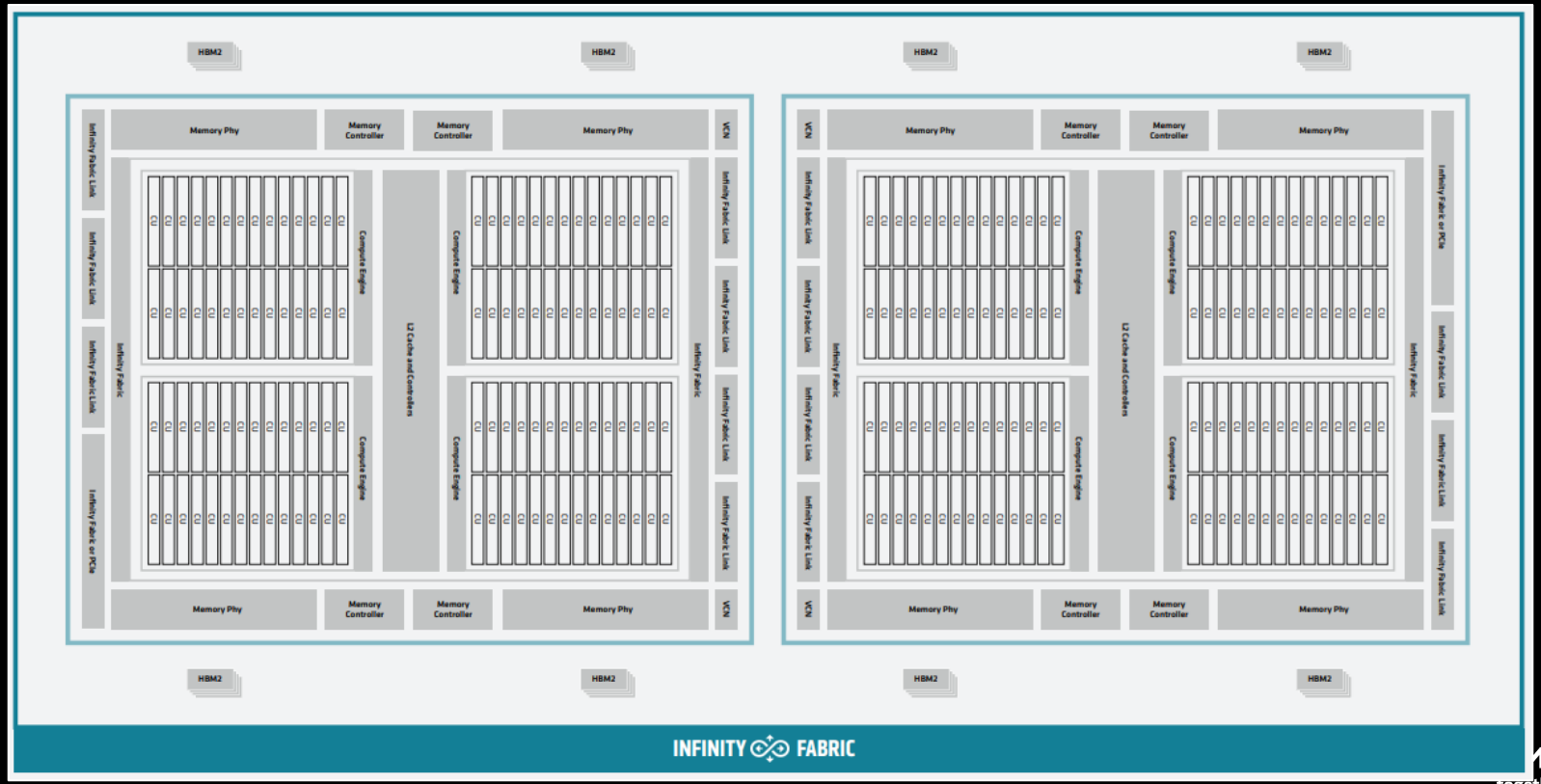


AMD MI100 Series (CDNA1)



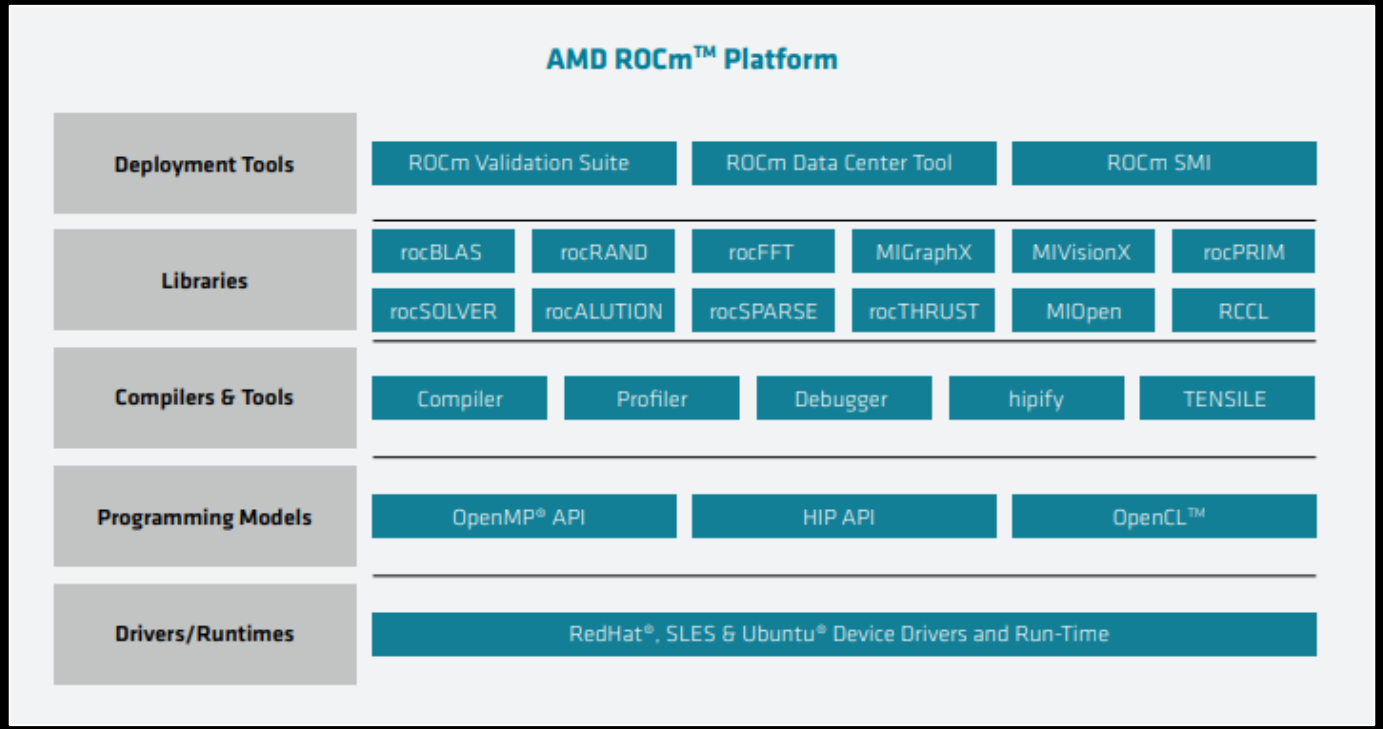


AMD MI200 Series (CDNA2)



ROCm Software Ecosystem

ROCm Overview



ROCm Source Code

ROCm Core Technology & Documentation
https://github.com/RadeonOpenCompute

ROCm Developer Tools and Programming Languages
https://github.com/ROCm-Developer-Tools

ROCm Software Platform Repository
https://github.com/ROCmSoftwarePlatform

ROCm Documentation

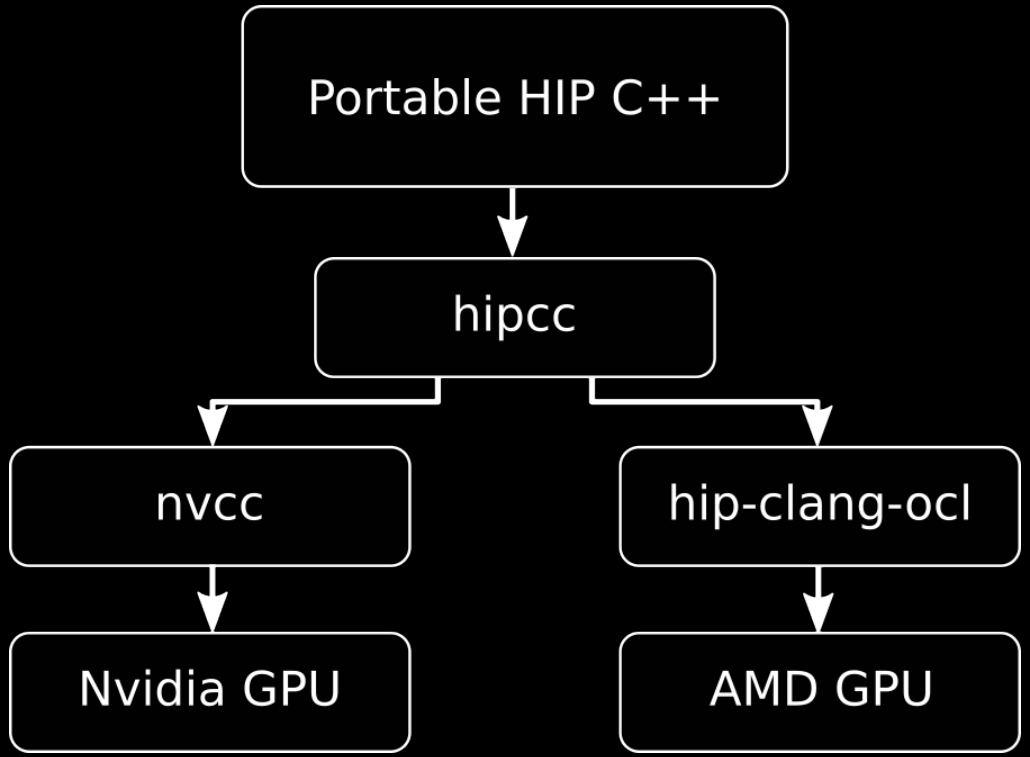
ROCm
https://docs.amd.com

AMD Developer Zone
https://developer.amd.com/

AMD Lab Notes
https://gpuopen.com/learn/amd-lab-notes

GPU Acceleration with HIP

Heterogeneous compute Interface for Portability (HIP)



Managing Memory

Allocate Memory

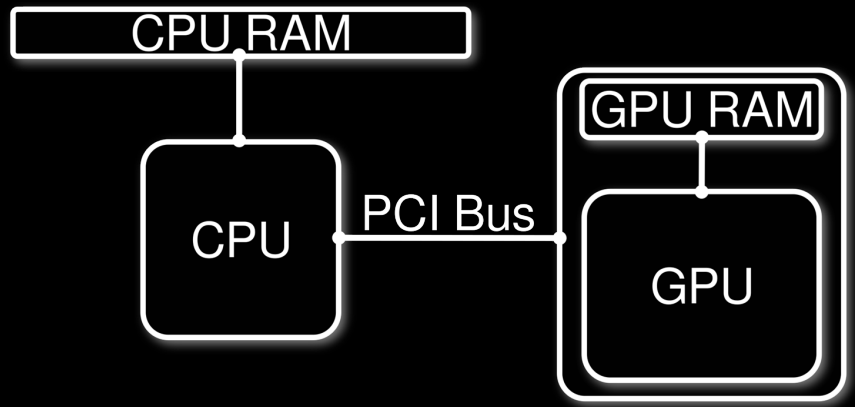
```
hipMalloc( ptr, size );
```

Copy Memory

```
hipMemcpy( dest, source, size, enum );
```

Deallocate Memory

```
hipFree( ptr );
```



Managing Memory (Example)

```
#include <hip/hip_runtime.h>
float *f_dev;
int N=100;

float *f = malloc(N*sizeof(float));
hipMalloc(f_dev,N*sizeof(float));
…
hipMemcpy(f_dev,f,N*sizeof(float),hipMemcpyHostToDevice);
…
hipMemcpy(f,f_dev,N*sizeof(float),hipMemcpyDeviceToHost);
…
free(f);
hipFree(f_dev);
```

Writing HIP Kernels

Thread

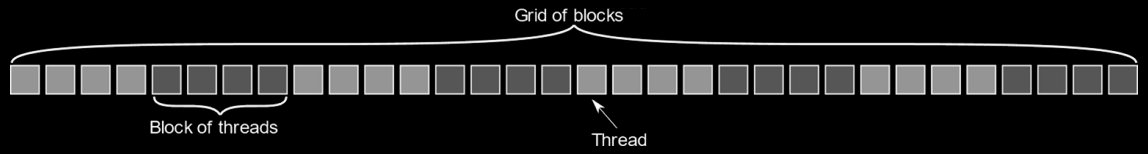
```
int tid = threadIdx.[x|y|z]
```

Block

```
int bid = blockIdx.[x|y|z]
int bdim = blockDim.[x|y|z]
```

Grid

```
int gdim = gridDim.[x|y|z]
```



Launching HIP Kernels

Chevron Syntax

```
kernel_name<<<nBlocks, nThreadsPerBlock>>>( args );
```

HIP API

```
hipLaunchKernelGGL((kernel_name),
                   nBlocks, nThreadsPerBlock,
                   args );
```

Writing & Launching HIP Kernels (Example)

CPU

```
float *a, *b, *c;
int N=1000;
a = (float*)malloc(N*sizeof(float));
b = (float*)malloc(N*sizeof(float));
c = (float*)malloc(N*sizeof(float));

…

for(int i=0; i<N; i++){
  c[i] = a[i] + b[i];
}
```

GPU

```
__global__ void vecadd(float *a, float *b, float *c,
int N){

size_t tid = threadIdx.x + blockIdx.x*blockDim.x;
size_t stride = gridDim.x;

for(int i = tid; i < N; i += stride){
  c[i] = a[i] + b[i];
}
```

```
#include <hip/hip_runtime.h>
…
float *a_dev, *b_dev, *c_dev;
hipMalloc(a_dev, N*sizeof(float));
…

dim3 nthread=(256,1,1);
dim3 nblock=(N/256+1,1,1);
vecadd<<<nblock,nthread>>>(a_dev, b_dev, c_dev, N);
```

Some Rules

The number of threads per block is limited, typically to 1024

Each block is executed independently

Order of execution is not guaranteed

Building code with hipcc

Platform (amd or nvidia)

```
export HIP_PLATFORM=amd
```

GPU Target

```
export GPU_TARGET=gfx908
```

```
hipcc /path/to/code.cpp
```

NVIDIA	AMD
P100 : sm_62	MI25 : gfx900
V100 : sm_72	MI50 : gfx906
A100 : sm_86	MI100 : gfx908
H100 : sm_90	MI200 : gfx90a

Building code with hipcc & CMake

```
# Get ROCm CMake Helpers onto your CMake Module Path

 if (NOT DEFINED ROCM_PATH )
   if (NOT DEFINED ENV{ROCM_PATH} )
     set(ROCM_PATH "/opt/rocm" CACHE PATH "ROCm path")
  else()
     set(ROCM_PATH $ENV{ROCM_PATH} CACHE PATH "ROCm path")
  endif()
endif()

set(CMAKE_MODULE_PATH "${ROCM_PATH}/lib/cmake" ${CMAKE_MODULE_PATH})
```

GPU Acceleration with HIPFort

HIPFort

https://github.com/ROCmSoftwarePlatform/hipfort

A Fortran interface for HIP and ROCm Accelerated Libraries

Managing Memory

Allocate Memory

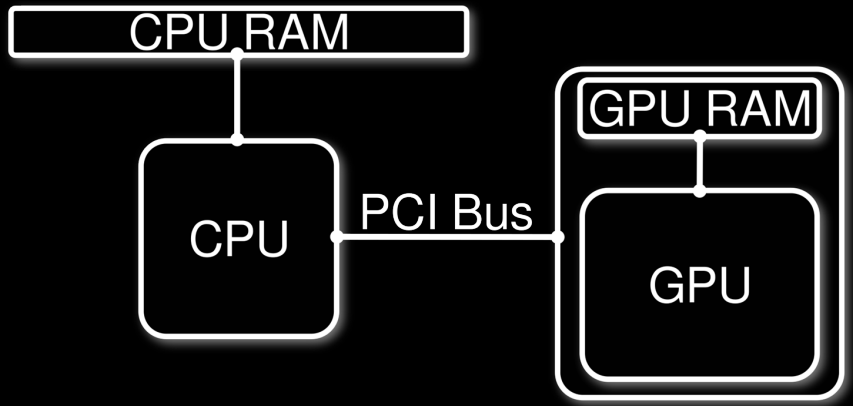
```
ierr = hipMalloc( ptr, size )
```

Copy Memory

```
ierr = hipMemcpy( dest, source, size, enum )
```

Deallocate Memory

```
ierr = hipFree( ptr )
```



Launching Kernels from Fortran

3 Ingredients

1. HIP Kernel (C)
2. Kernel Launch “wrapper” (C)
3. Subroutine Interface (Fortran)

Writing & Launching HIP Kernels (Fortran Example)

CPU

```
REAL, ALLOCATABLE :: a(:), b(:), c(:)
INTEGER, PARAMETER :: N=1000

ALLOCATE(a(1:N),b(1:N),c(1:N))

…

DO i=1,N
  c(i) = a(i) + b(i)
ENDDO
```

GPU

```
#include <hip/hip_runtime.h>

__global__ void vecadd(float *a, float *b, float *c,
int n){
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
  size_t stride = blockDim.x * gridDim.x;

  for (size_t i = tid; i < n; i += stride)
    c[i] = a[i] + b[i];
  }
}

extern "C"{
  void vecadd_wrapper(dim3* grid, dim3* block, float
*a, float *b, float *c, int N){

    vecadd<<<*grid, *block>>>(a,b,c,N);
  }
}
```

Writing & Launching HIP Kernels (Fortran Example)

CPU

```
REAL, ALLOCATABLE :: a(:), b(:), c(:)
INTEGER, PARAMETER :: N=1000

ALLOCATE(a(1:N),b(1:N),c(1:N))

…

DO i=1,N
  c(i) = a(i) + b(i)
ENDDO
```

GPU

```
program fortran_hip
  use hipfort
  use hipfort_check
  use iso_c_binding

  implicit none

  interface
     subroutine vecadd_wrapper(grid,block,a,b,c,N)
bind(c,name=”vecadd_wrapper”)
       use iso_c_binding
       use hipfort_types
       implicit none
       type(c_ptr) :: a, b, c
       integer(c_int), value :: N
       type(dim3) :: grid, block
     end subroutine
  end interface
```

Writing & Launching HIP Kernels (Fortran Example)

CPU

```
REAL, ALLOCATABLE :: a(:), b(:), c(:)
INTEGER, PARAMETER :: N=1000

ALLOCATE(a(1:N),b(1:N),c(1:N))

…

DO i=1,N
  c(i) = a(i) + b(i)
ENDDO
```

GPU

```
  REAL,POINTER :: a(:), b(:), c(:)
  REAL,POINTER :: da(:) => null(), &
                  db(:) => null(), &
                  dc(:) => null() ! Device Arrays

  ! Allocate host memory
  allocate(a(N),b(N),out(N))
  …
  ! Allocate array space on the device
  call hipCheck(hipMalloc(da,mold=a))
  call hipCheck(hipMalloc(db,mold=b))
  call hipCheck(hipMalloc(dc,mold=c))

  ! Transfer data from host to device memory
  call hipCheck(hipMemcpy(da, a, N,
hipMemcpyHostToDevice))
  call hipCheck(hipMemcpy(db, b, N,
hipMemcpyHostToDevice))

  ! launch kernel
  call vecadd_wrapper(grid,block,&
              c_loc(da),c_loc(db),c_loc(dc),N)
```

Building code with hipfc

Platform (amd or nvidia)

```
export HIP_PLATFORM=amd
```

GPU Target

```
export GPU_TARGET=gfx908
```

Fortran Compiler

```
export HIPFORT_COMPILER=gfortran
```

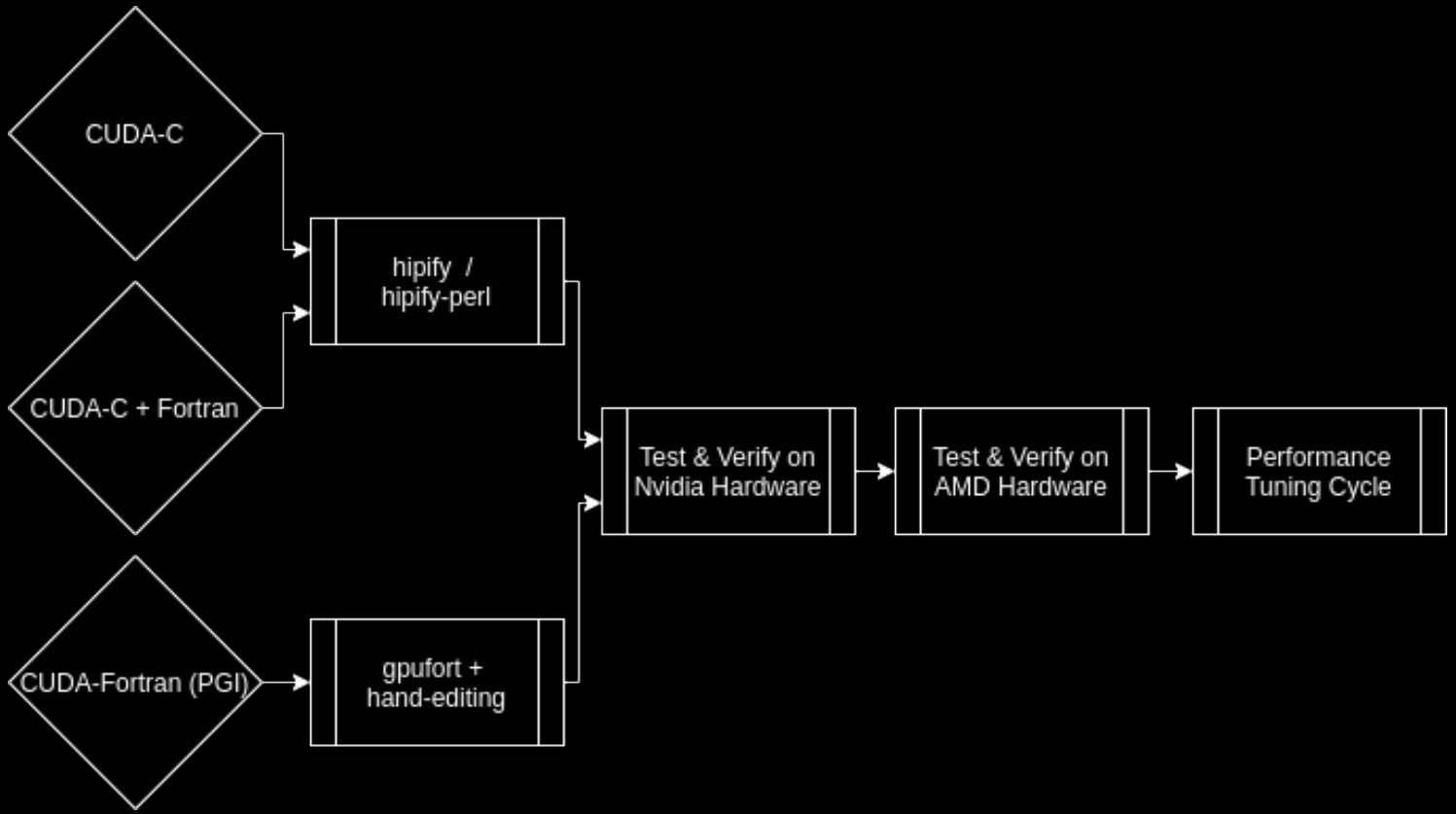
NVIDIA	AMD
P100 : sm_62	MI25 : gfx900
V100 : sm_72	MI50 : gfx906
A100 : sm_86	MI100 : gfx908
H100 : sm_90	MI200 : gfx90a

```
hipfc /path/to/code.f90

hipfc /path/to/code.cpp
```

Porting from Nvidia to AMD

HIPification Strategy



HIPification Strategy Examples : Darknet

https://github.com/AlexeyAB/darknet

Purpose : Object Detection and Classification

About : Open-source community code with 15.8K stars, 16.7K forks, ~50 contributors

Language : C/C++, CUDA

Build Options : Serial x86, CUDA, or cuDNN

https://www.youtube.com/watch?v=IdumjmPsTck

HIPification Strategy Examples : Darknet

HIPify (Alternative Approach)

- Add consolidated header “gpu.hpp”.
- Migrate all #include <cuda*> to gpu.hpp

```
#ifdef __HIP_ROCclr__

#include <hip/hip_runtime.h>
#include <rocrand.h>
#include <hiprand.h>
#include <hipblas.h>

#define cudaDeviceSynchronize hipDeviceSynchronize
#define cudaError hipError_t
#define cudaError_t hipError_t


#else

#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>
#include <curand.h>
#include <cublas_v2.h>
```

GPU Acceleration with OpenMP

GPU Acceleration with OpenMP

You will learn about …

- OpenMP 4.5+ standard and compiler support
- OpenMP target regions (C/C++ & Fortran)
- Accelerating do loops
- Building OpenMP accelerated applications

GPU Acceleration with OpenMP

- ≥4.5 supports `target` regions (GPU offloading)
- Compiler directives to “hint” at acceleration strategies
- Similar to OpenACC
- Supports Fortran and C/C++
- Supports multi-core CPU and GPU platforms (NVIDIA & AMD)
- OpenMP 5.0 provides support for
  - “Deep copies” of derived types in Fortran
  - Memory management for interoperability

OpenMP Programming Model

`target` and `map` directives

```
!$omp target map(to:A,x) map(Ax)
DO j = 1, N
  DO i = 1, N
    DO r = 1, N
      Ax(i,j) = Ax(i,j)+A(r,i)*x(r)
    ENDDO
  ENDDO
ENDDO
!$omp end target
```

```
#pragma omp target map(to:A,x) map(Ax)
{
  for( int j = 0; j < N; j++ ){
    for( int i = 0; i < N; i++ ){
      for( int r = 0; r < N; r++) {
        Ax[i+N*j] = Ax[i+N*j]+A[r+N*i]*x[r]
      }
    }
  }
}
```

Workflow

- Target directive creates a **target task** to be executed on the GPU
- Map clause allocates space on the GPU (if needed) and copies data between CPU and GPU

OpenMP Programming Model

`teams` & `parallel` directives

```
!$omp target map(to:A,x) map(Ax)
!$omp teams distribute parallel for
DO j = 1, N
  DO i = 1, N
    DO r = 1, N
      Ax(i,j) = Ax(i,j)+A(r,i)*x(r)
    ENDDO
  ENDDO
ENDDO
!$omp end target
```

```
#pragma omp target map(to:A,x) map(Ax)
{
  #pragma teams distribute parallel for
  for( int j = 0; j < N; j++ ){
    for( int i = 0; i < N; i++ ){
      for( int r = 0; r < N; r++) {
        Ax[i+N*j] = Ax[i+N*j]+A[r+N*i]*x[r]
      }
    }
  }
}
```

Workflow

- The teams directives creates a **league** of teams with a single **thread** per team.
- The distribute construct specifies the do/for loop iterations will be distributed amongst the teams
- The parallel construct populates the teams with multiple threads

OpenMP Programming Model

Loop Collapsing tightly nested loops

```
!$omp target map(to:A,x) map(Ax)
!$omp teams distribute parallel for collapse(2)
DO j = 1, N
  DO i = 1, N
    DO r = 1, N
      Ax(i,j) = Ax(i,j)+A(r,i)*x(r)
    ENDDO
  ENDDO
ENDDO
!$omp end target
```

```
#pragma omp target map(to:A,x) map(Ax)
{
  #pragma teams distribute parallel for collapse(2)
  for( int j = 0; j < N; j++ ){
    for( int i = 0; i < N; i++ ){
      for( int r = 0; r < N; r++) {
        Ax[i+N*j] = Ax[i+N*j]+A[r+N*i]*x[r]
      }
    }
  }
}
```

Workflow

- The collapse(2) clause “collapses” the outer two loops for distribution amongst the teams.

OpenMP Programming Model

Memory Management

```
!$omp target enter data map(alloc:A,x,Ax)
...
!$omp target update to(A,x)

!$omp target map(to:A,x) map(Ax)
!$omp teams distribute parallel for collapse(2)
DO j = 1, N
  DO i = 1, N
    DO r = 1, N
      Ax(i,j) = Ax(i,j)+A(r,i)*x(r)
    ENDDO
  ENDDO
ENDDO
!$omp end target

!$omp target update from(Ax)
...

!$omp target exit data map(delete:A,x,Ax)
```

Workflow

- `target enter data` directive maps variables to a **device data environment**.
  - The `map` clause is used to allocate and deallocate data on the device.
- The `target update` directive is used to move data between host and device.
  - `to` copies “to” the GPU
  - `from` copies “from” the GPU

Building code with OpenMP 4.5

OpenMP is a standard implemented to varying degrees in different compilers

We recommend using using `amdclang` and `amdflang` or Cray compilers (when available)

GNU compilers require building from source

Building code with amdclang/flang

Enable OpenMP parsing

-fopenmp

GPU Target

-fopenmp-targets=<target>
-Xopenmp-target=<target>

GPU Architecture

-march=<gpu-arch>

nvptx64-nvidia-cuda	amdgcn-amd-amdhsa
P100 : sm_62	MI25 : gfx900
V100 : sm_72	MI50 : gfx906
A100 : sm_86	MI100 : gfx908
H100 : sm_90	MI200 : gfx90a

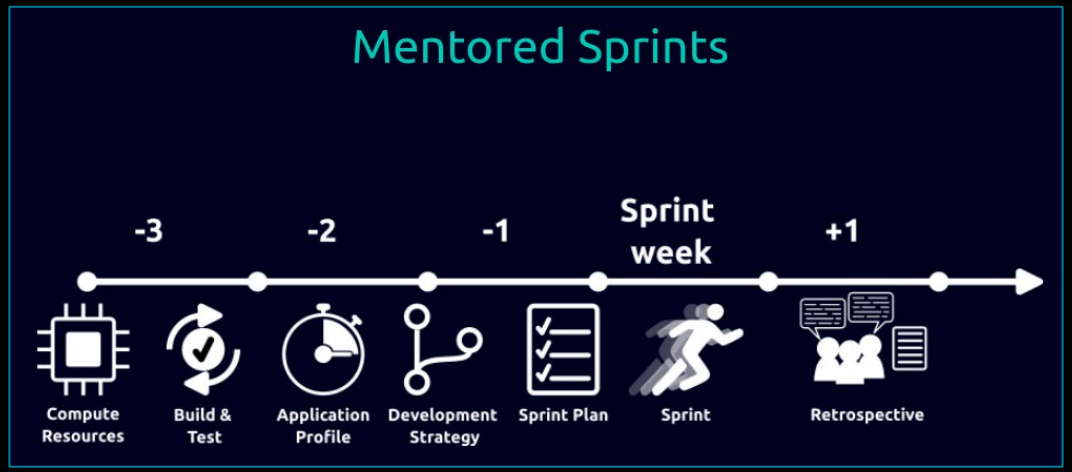
```
amdclang -fopenmp \
         -fopenmp-targets=<target> \
         -Xopenmp-target=<target> \
         -march=<gpu-arch>
         [other options]
         <source-code>
```

Codelabs

https://fluidnumerics.github.io/scientific-computing-edu/

- Accelerate Fortran application with OpenMP 5.0
- Accelerate Fortran application with hipfort
- Accelerate C/C++ application with OpenMP 5.0
- Accelerate C/C++ application with HIP

Work with us



www.fluidnumerics.com

Reach out to support@fluidnumerics.com