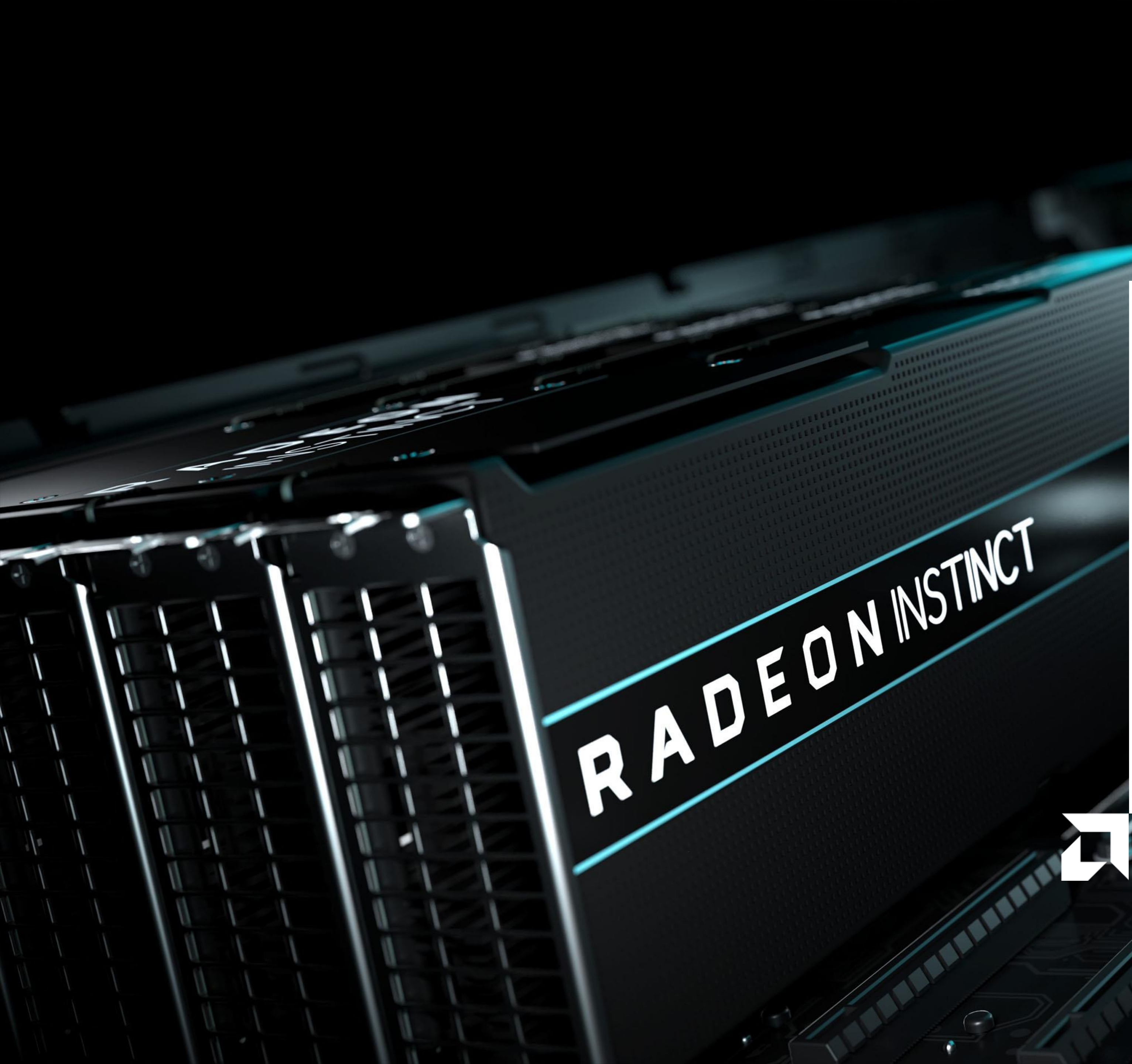
Getting Started with OpenMP® Offload Applications on AMD Accelerators

Jose Noudohouenou, Bob Robey
AMD @HLRS
Sept 25-28th, 2023

AMD
together we advance_

Agenda

1. AMD Open Source Software Stack: Recap
2. Building Simple OpenMP® Offload Applications
3. Hybrid MPI + OpenMP® Offload Applications Support
4. Runtime Report for Performance Investigation
5. Summary



AMD ROCm™ Software Stack: Recap

OPEN SOFTWARE PLATFORM FOR GPU COMPUTE

AMD ROCm

- Unlocked GPU Power To Accelerate Computational Tasks
- Optimized for HPC and Deep Learning Workloads at Scale
- Open Source Enabling Innovation, Differentiation, and Collaboration

Benchmarks & App Support	Optimized Training/Inference Models & Applications; MLPERF, HPL/HPCG, Life Science, Geo Science, Physics
Operating Systems Support	RHEL, CentOS, SLES, Ubuntu
Cluster Deployment	Singularity, Kubernetes®, Docker®, SLURM
Framework Support	Kokkos/RAJA, PyTorch, TensorFlow
Libraries	BLAS, RAND, FFT, MIGraphX, MIVisionX, PRIM, SOLVER, ALUTION, SPARSE, THRUST, MIOpen, RCCL
Programming Models	OpenMP® API, OpenCL™, HIP API
Development Toolchain	Compiler, Profiler, Tracer, Debugger, hipify, GPUFort
Drivers & Runtime	GPU Device Drivers and ROCm Run-Time
Deployment Tools	ROCm Validation Suite, ROCm Data Center Tool, ROCm SMI

Compilers(1)

- ROCmCC and AOMP
  - ROCmCC provides support for both HIP and OpenMP®
  - AOMP: the AMD OpenMP® research compiler for AMD Instinct™ accelerators. It is used to prototype new OpenMP® features for ROCmCC

- GNU Compilers (GNU Compiler Collection - https://gcc.gnu.org/wiki/Offloading)
  - Provide offloading support to AMD GPUs (OpenMP®, OpenACC)
    - GCC 11 supports additionally:
      - AMD GCN MI100 (gfx908) GPUs.
    - GCC 13 (under development) adds support for AMD's Instinct MI200 (gfx90a) GPU series.
    - The devel/omp/gcc-12 (OG12) branch augments the GCC 12 branch with OpenMP and offloading features.

Compilers(2)

- Siemens® Compilers (Sourcery CodeBench Lite – C/C++/Fortran)
  - Siemen's free GCC-based compilers
  - Supports all GCC 12 features, enriched by OpenMP® features from GCC's development branch and AMD GCN improvements such as support for offloading debugging.
  - Still under development.

- If you are on an AMD/HPE HPC system, there are additional options

  Cray Compilers (HPE compilers)
  - Provide offloading support to AMD GPUs (OpenMP®, HIP, OpenACC)

- List of OpenMP Compilers & Tools :
  - https://github.com/ROCm-Developer-Tools/aomp
  - https://www.openmp.org/resources/openmp-compilers-tools

Compilers(3)

	Cray			AMD			
Module	**cce**			**ROCm™**			
Language	**C**	**C++**	**Fortran**	**C**	**C++**	**Fortran**	**HIP**
Compiler Command Line	craycc	crayCC	crayftn	amdclang clang clang-cl	amdclang++ clang++	amdflang flang	hipcc

hipcc: wrapper for amdclang/amdclang++. It also makes sure to call either amdclang or nvcc based on the platform. Useful portability.

In case of porting CUDA codes to HIP, ROCm™ provides 'HIPification' tools to do the heavy-lifting

Please consider using HIPify tools like Hipify-perl or Hipify-clang

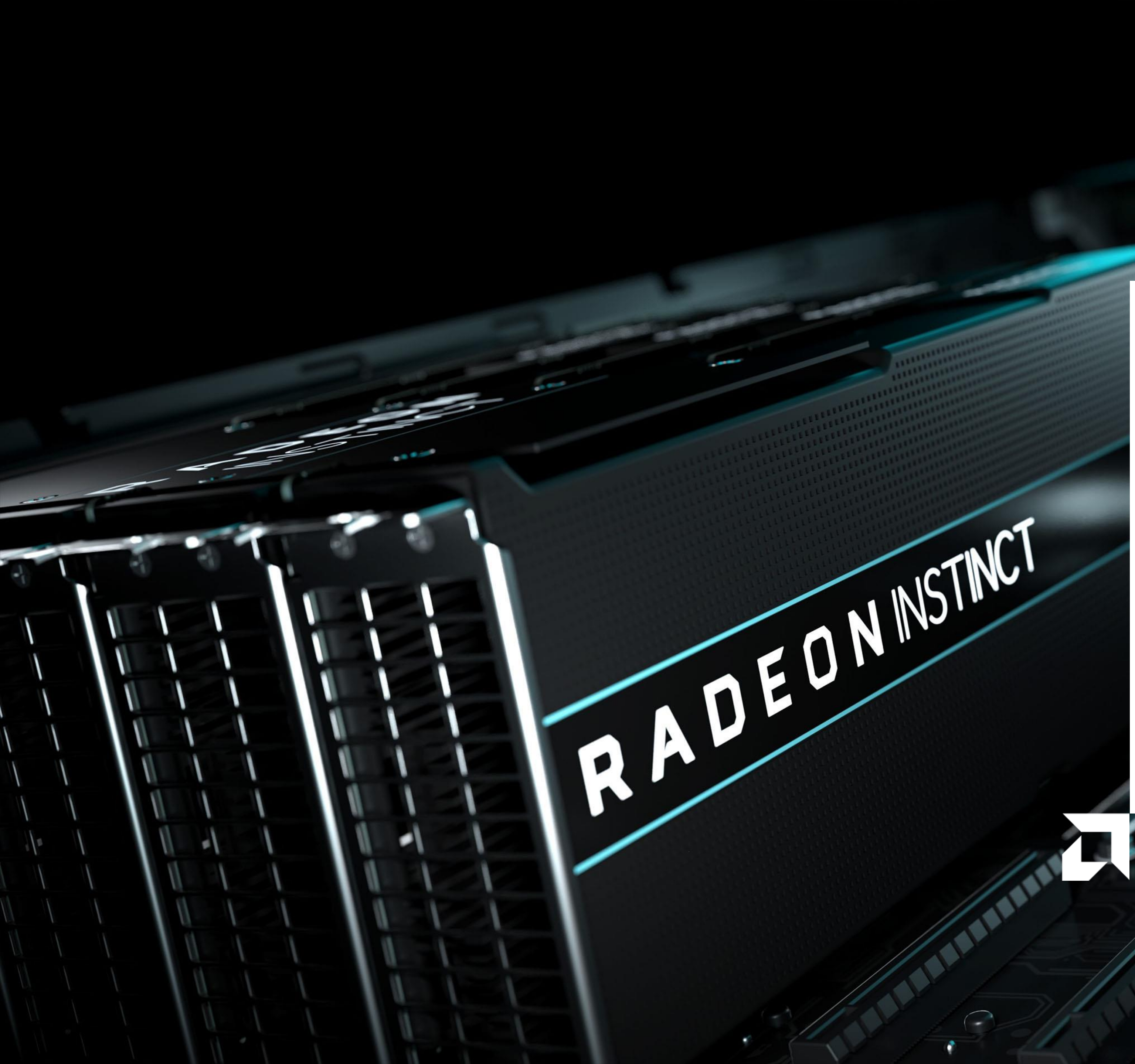
ROCm Libraries

ROCm Library	Note/comment
hipBLAS/rocBLAS	Basic Linear Algebra Subroutines
hipFFT/rocFFT	Fast Fourier Transfer Library
hipSPARSE/rocSPARSE	Sparse BLAS + SPMV
hipSolver/rocSolver	Lapack Library
rocALUTION	Sparse iterative solvers & preconditioners with Geometric & Algebraic MultiGrid
hipThrust/rocThrust	C++ parallel algorithms library
rocPRIM	Low Level Optimized Parallel Primitives
MIOpen	Deep learning Solver Library
hipRAND/rocRAND	Random Number Generator Library
EIGEN – HIP port	C++ template library for linear algebra: matrices, vectors, numerical solvers
RCCL	Communications Primitives Library based on the MPI equivalents

Latest status available at: https://github.com/ROCm-Developer-Tools/HIP

This talk focuses on simple and hybrid (MPI +) OpenMP Offload Application building and running on AMD GPUs

Building Simple OpenMP® Offload Applications



Enabling OpenMP® on AMD Hardware

	AMD			GCC		
Module	**ROCm™**			**gcc**		
Language	**C**	**C++**	**Fortran**	**C**	**C++**	**Fortran**
Compiler Command Line	amdclang	amdclang++	amdflang	$GCC_PATH/bin/gcc	$GCC_PATH/bin/g++	$GCC_PATH/bin/gfortran
Compiler Flags (CPU)	-fopenmp			-fopenmp		

Compiling OpenMP® Offload Codes using ROCm™

Module: rocm

Offloading Target (CPU/GPU/GCD)	Required Flags	New ROCm Options (can be used in lieu of Required Flags)	
AMD MI200[1]	-fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx90a	ROCm>=4.5	--offload-arch=gfx90a
AMD MI100[1]	-fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908	ROCm>=5.0	--offload-arch=gfx908
Native Host (CPU)	-fopenmp-targets=amdgcn-amd-amdhsa		

Furthermore, using **amdclang, amdclang++, amdflang** requires the following flags: **-fopenmp -target x86_64-pc-linux-gnu**

AMD's commercially available Radeon Instinct™ GPU code names "MI100" and "MI200"

Compiling OpenMP® Offload Codes: GCC Compilers

Syntax for all GCC versions:
-foffload=disable //to generate code for all supported offload targets
-foffload=default //to generate code only for the host fallback
-foffload=target-list //to generate code only for the specified comma-separated list of offload targets

In GCC12:
-foffload-options=options // GCC passes the specified options to the compilers for all enabled offloading targets.
-foffload-options=target-triplet-list=options

Examples:
-foffload=amdgcn-amdhsa=-march=gfx908
-foffload-options=-lgfortran -foffload-options=-lm
-foffload-options=amdgcn-amdhsa=-march=gfx906 -foffload-options=-lm
https://gcc.gnu.org/wiki/Offloading

Offload targets are specified in GCC's internal target-triplet format. You can run the compiler with **gcc -v** to show the list of configured offload targets under **OFFLOAD_TARGET_NAMES**.

Additional Compilers

Enabling OpenMP® on AMD Hardware

	Cray		
Module	**cce**		
Language	**C**	**C++**	**Fortran**
Compiler Command Line	craycc	crayCC	crayftn
Compiler Flags (CPU)	-fopenmp		-homp -fopenmp

Compiling OpenMP® Offload Codes using Cray Compilers

Offloading Target (CPU/GPU/GCD)	Cray Accelerator Modules
AMD MI200[1]	craype-accel-amd-gfx90a
AMD MI100[1]	craype-accel-amd-gfx908
Native Host (CPU)	craype-accel-host

Cray Compilers also have wrappers:
use ftn, cc, and CC to compile Fortran, C and C++ codes, respectively, instead of invoking the native compilers

OpenMP® Offloading Example: Reduction(1)

```
#include <stdio.h>
#include <stdlib.h>
#define N 5000000
int main(){
 double *a,  *b;
  a = (double*)malloc(sizeof(double) * N);
  b = (double*)malloc(sizeof(double) * N);
  for(int i = 0; i < N; i++){
    a[i] = 1.0;
    b[i] = 1.0;
  }

  double sum = 0;
  #pragma omp target data map(to:a[0:N], b[0:N])
  #pragma omp target teams distribute parallel for map(tofrom:sum) 
reduction(+:sum)
  for(int i = 0; i < N; i++)
    sum += a[i] * b[i];
```

Data directive to move data to device (GPU)

Compute loop on GPU, copy sum to and from GPU and do a sum reduction on sum variable

OpenMP® Offloading Example: Reduction(2)

```
  printf("SUM = %f\n", sum);
  free(a);
  free(b);
  return 0;
}
```

module purge
module load rocm/5.4.3
export PATH=$ROCM_PATH/llvm/bin/:$PATH

[jnoudoho@TheraC60 projects]$ amdclang -O3 -std=c99 -g -fopenmp --offload-arch=gfx90a reduction_test.c -o reduction_test.exe

[jnoudoho@TheraC60 projects]$ ./reduction_test.exe
SUM = 5000000.000000

Alternative: *amdclang -O3 -std=c99 -g -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx90a reduction_test.c -o reduction_test.exe*

AMD together we advance_

Fortran and OpenMP® offloading

- AOMP compiler (LLVM™) with Flang
- GCC compiler with gfortran

- Many features are still being added to Fortran compilers
- Use the latest compiler version
- Expect features to be added with every release

A Simple Fortran OpenMP® Kernel Offloading (1)

```
program my_fib
   integer ::i,j
   real ::sum, sum2
   real,pointer ::array(:),buffer(:)
   allocate(array(10))
   allocate(buffer(10))
   do j=1, 10
      array(j)=1.0
   end do

   !$OMP TARGET TEAMS DISTRIBUTE MAP(TO:array(1:10)) MAP(TOFROM:buffer(1:10)) PRIVATE(sum,sum2)
   do i=1, 10
      sum2=0.0
      sum=1000.0
      !$OMP PARALLEL DO REDUCTION(+:sum2)
      do j=1, 10
         sum2=sum2+array(j)
      end do
      !$OMP END PARALLEL DO
      buffer(i)=sum+sum2
   end do
   !$OMP END TARGET TEAMS DISTRIBUTE
   do i=1, 10
      write(*, *) "sum=", buffer(i)
   end do
end program
```

Create parallel code region and copy data to and from GPU. Create a private variable sum for each compute thread.

Compute loop in parallel on GPU with a sum reduction

End parallel compute on GPU

End parallel code region. Data marked from will be copied back to CPU.

A Simple Fortran OpenMP® Kernel Offloading (2)

```
module load rocm/5.4.3
```

[jnoudoho@TheraC60 fortran-test]$ `amdflang -O2 -fopenmp --offload-arch=gfx90a myfib-test.f90 -o myfib-test`

[jnoudoho@TheraC60 fortran-test]$ ./myfib-test

```
 sum= 1010.
 sum= 1010.
 sum= 1010.
 sum= 1010.
 sum= 1010.
 sum= 1010.
 sum= 1010.
 sum= 1010.
 sum= 1010.
 sum= 1010.
```

***Alternative**:* `amdflang -O2 -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx90a myfib-test.f90 -o myfib-test`

OpenMP® Offloading Example: Unified Shared Memory Support (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define SIZE 1024
#pragma omp requires unified_shared_memory
int main() {
      int deviceID = (omp_get_num_devices() > 0) ? omp_get_default_device()
                   : omp_get_initial_device();
      printf("ID=%d\n",deviceID);

      int *x = (int *)omp_target_alloc(SIZE, deviceID);
      // (int *)malloc(sizeof(int)*SIZE);
      //posix_memalign((void**) &x, 2*1024*1024, sizeof(int)*SIZE);
      int *y = (int *)omp_target_alloc(SIZE, deviceID);
      // (int *)malloc(sizeof(int)*SIZE);
      //posix_memalign((void**) &y, 2*1024*1024, sizeof(int)*SIZE);
```

OpenMP® Offloading Example: Unified Shared Memory Support (2)

```
        for (int i = 0; i < SIZE; i++) {
                x[i] = i;
                y[i] = SIZE - i;
        }

        #pragma omp target teams distribute parallel for
        for (int i = 0; i < SIZE; i++) {
                x[i] += y[i];
        }

        omp_target_free(x, deviceID); // free(x);
        omp_target_free(y, deviceID); // free(y);
        printf("%s passed\n", __func__);
        return EXIT_SUCCESS;
}
```

OpenMP® Offloading Example: Unified Shared Memory Support (3)

```
module purge
module load rocm/5.3.0
export PATH=$ROCM_PATH/llvm/bin/:$PATH
```

[jnoudoho@TheraC60 usm]$ `amdclang -O2 -std=c99 -g -fopenmp --offload-arch=gfx90a test_usm.c -o test_usm`

***Alternative**:* *`amdclang –O2 -std=c99 -g -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx90a test_usm.c -o test_usm.exe`*

Building Applications With Libraries

Libraries are located in: **${ROCM_PATH}/lib**

#hipfft;hiprand; hipsolver; hipsparse; rccl; rocblas; rocalution_hip; rocsolver; etc ….

Depending on the libraries needed, add to your compiler options:

```
LDOPTS =-L${ROCM_PATH}/lib -lhipblas -lamdhip64
LDOPTS = -L${ROCM_PATH}/lib -lrocfft -lamdhip64
…
```

etc

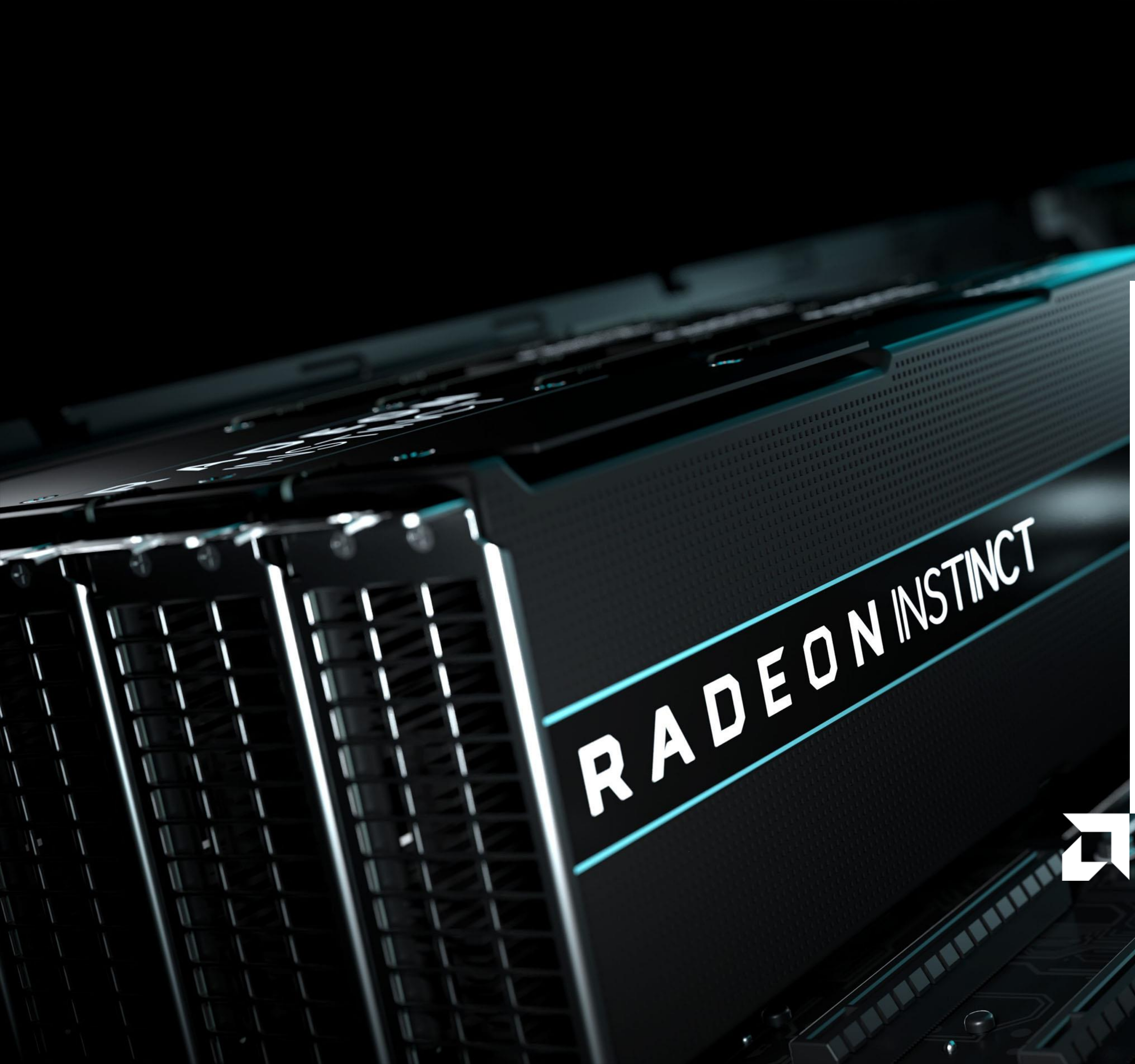
Examples:

```
module purge
module load cce/13.0.2 craype-accel-amd-gfx908 rocm/5.4.3

echo $ROCM_PATH
/opt/rocm-5.4.3

$ gfortran -I../modules rocfft.f03 ../modules/common.o -o rocfft -L/opt/rocm-5.4.3/lib -lrocfft -lamdhip64
$ gfortran -I../modules ../modules/common.o dgemm.f03 -o dgemm -L/opt/rocm-5.4.3/lib -lhipblas -lamdhip64
```

Hybrid MPI + OpenMP Offload Applications Support



Multi-GPU MPI Communications with ROCm™

- Most popular HPC applications rely on multi-GPU MPI programming models to scale their workloads.
- MPI is widely used to scale to multiple nodes in HPC applications.
- ROCm enables various technologies to facilitate the porting of applications to clusters with GPUs:
  - Allowing direct use of GPU pointers in MPI calls.
  - Enabling ROCm-aware MPI libraries to deliver optimal performance for both intra-node and inter-node GPU-to-GPU communication.
- Depending on the application, MPI binding might be necessary to get great performance

Multi-GPU Support: Getting Target Machine GPU IDs

```
[jnoudoho@TheraC60 ~]$ module load rocm/5.4.3
[jnoudoho@TheraC60 ~]$ rocm-smi -i
======================== ROCm™ System Management Interface ========================
========================================= ID =========================================
GPU[0]          : GPU ID: 0x740f
GPU[1]          : GPU ID: 0x740f
GPU[2]          : GPU ID: 0x740f
GPU[3]          : GPU ID: 0x740f
GPU[4]          : GPU ID: 0x740f
GPU[5]          : GPU ID: 0x740f
GPU[6]          : GPU ID: 0x740f
GPU[7]          : GPU ID: 0x740f
=======================================================================================
============================== End of ROCm™ SMI Log ==================================
```

Other command lines:
`rocm-smi --showhw`
`rocm-smi`

CPU/GPU NUMA Topologies
`rocm-smi --showtoponuma`

CPU Architecture Info
`lscpu`

AMD together we advance_

Hybrid MPI/OpenMP Offload Code: Compiling

Requirements: Compiler + Accelerator module + mpi modules (+ libs if necessary)

	Cray			AMD/GCC		
Modules	**cce, cray-mpich**			**ROCm™ , openmpi**		
Language	**C**	**C++**	**Fortran**	**C**	**C++**	**Fortran**
Compiler Command Line	craycc	crayCC	crayftn	mpicc	mpiCC mpic++ mpicxx	mpifort mpif77 mpif90
Compiler Flags	-fopenmp		-homp -fopenmp	-fopenmp		

Libraries/Include files:

- MPI header files are automatically linked to your program when using Cray compilers wrappers (ftn, cc, CC)
- When building MPI codes directly via clang/flang compilers, the developer might need to specify:

```
-L${MPICH_DIR}/lib -lmpi
-I${MPICH_DIR}/include
```

Hybrid MPI/OpenMP® Offload to AMD GPU(1)

```
int main(int argc, char* argv[])
{
   int rank;
   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

//    int n=atoi(argv[1]);
   int n=10240000;
   int num_iteration=100;
   double scalar=2.0;

   double *x = (double*)malloc(n*sizeof(double));
   double *y = (double*)malloc(n*sizeof(double));
   double *z = (double*)malloc(n*sizeof(double));
```

Hybrid MPI/OpenMP® Offload to AMD GPU(2)

```
for (int i = 0; i < n; i++) {
        x[i] = 0.0;
        y[i] = 2.0f;
        z[i] = (double)i+1;
   }
   printf("Rank=%d\n",rank);

//#pragma omp target enter data map(alloc: x[0:n]) map(to: y[0:n], z[0:n])
#pragma omp target enter data  map(to: x[0:n], y[0:n], z[0:n])

   double * timers = (double *)calloc(num_iteration,sizeof(double));
```

Hybrid MPI/OpenMP® Offload to AMD GPU(3)

```
for (int iter=0;iter<num_iteration; iter++)
  {
       double start = omp_get_wtime();
       #pragma omp target teams distribute parallel for
       for (int i=0; i<n; i++)
               x[i] = y[i]+scalar*z[i];

       timers[iter] = omp_get_wtime()-start;
  }
#pragma omp target exit data map(from: x[0:n])
  double sum_time =  0.0;
  double max_time = -1.0e10;
  double min_time =  1.0e10;
```

Hybrid MPI/OpenMP® Offload to AMD GPU(4)

```
for (int iter=0; iter<num_iteration; iter++) {
        sum_time += timers[iter];
        max_time  = MAX(max_time,timers[iter]);
        min_time  = MIN(min_time,timers[iter]);
   }

   double avg_time = sum_time / num_iteration;

   printf("-Timing in Seconds: min=%f, max=%f, avg=%f\n", min_time,
max_time, avg_time);

   double local_bw = (3*sizeof(double)*n*1E-9)/avg_time;
   double bw = 0;
```

Hybrid MPI/OpenMP® Offload to AMD GPU(5)

```
// Average BW achieved by the considered system
   MPI_Reduce(&local_bw, &bw, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
   if (rank == 0)
           printf("GB/s BW reached: %lf \n\n", bw);

   free(x);
   free(y);
   free(z);

MPI_Finalize();
   return 0;
}
```

Hybrid MPI/OpenMP® Offload to AMD GPU(6)

Example: Hybrid MPI + OpenMP offload of Triad codelet

Building the code:

```
clang -O3 -std=c99 -g -lmpi -fopenmp --offload-arch=gfx90a -o triad_mpi_omp triad_mpi_omp.c
```

Commands to run this codelet (implicit scaling):

Number of ranks: 16

Run on the FULL system: `mpirun -np 16 ./ triad_mpi_omp`

Single-GPU run (gpu # 2):

```
mpirun -x ROCR_VISIBLE_DEVICES=2 -np 16 ./triad_mpi_omp        #with openmpi
mpiexec -env ROCR_VISIBLE_DEVICES 2 -np 16 ./triad_mpi_omp     #with mpich
```

Four GPUs runs (on GPUs 2, 4, 5, 7):

```
mpirun -x ROCR_VISIBLE_DEVICES=2,4,5,7 -np 16 ./triad_mpi_omp      #with openmpi
mpiexec -env ROCR_VISIBLE_DEVICES 2,4,5,7 -np 16 ./triad_mpi_omp   #with mpich
```

Here, visible devices are visible to ALL MPI ranks. Is it what you want?

Example of Running Hybrid MPI/OpenMP® Applications

- Let's consider a 2-step run:
  - run.sh with sbatch command line to launch the app
  - sbatch.slurm This file contains sbatch parameters and the call to srun command line

- $cat run.sh

  load necessary modules

  export necessary environment variables

```
make clean all #To build the code
sbatch -p <partition> -w <node> sbatch.slurm
```

- $cat sbatch.slurm

```
#!/bin/bash
#SBATCH --job-name=triad_mpi
#SBATCH --ntasks=2
#SBATCH --ntasks-per-node=2
#SBATCH --gres=gpu:2
#SBATCH --cpus-per-task=2
#SBATCH --nodes=1
#SBATCH --distribution=block:block
#SBATCH --time=00:20:00
#SBATCH --output=triad.out
#SBATCH --error=triad.err

cd ${SLURM_SUBMIT_DIR}
```

- **load necessary modules**
- **export necessary environment variables**

```
srun ./triad_mpi
#mpirun ./<app> <args>
```

ROCm-aware MPI

- MPI Support for ROCm™
  - Open MPI (https://github.com/openucx/ucx/wiki/Build-and-run-ROCM-UCX-OpenMPI#Build)
  - MPICH (https://rocmdocs.amd.com/en/latest/Remote_Device_Programming/Remote-Device-Programming.html#mpich)
  - MVAPICH2-GDR (http://mvapich.cse.ohio-state.edu/features/#mv2gdr)

- MPI implementations can be made ROCm-aware by compiling them with UCX support (Unified Communication - X Framework (UCX)). One notable exception is MVAPICH2: it directly supports AMD GPUs without using UCX.

- UCX (https://rocmdocs.amd.com/en/latest/Remote_Device_Programming/Remote-Device-Programming.html#ucx)
  - Unified Communication X (UCX) is a communication library for building Message Passing (MPI), PGAS/OpenSHMEM libraries and RPC/data-centric applications
  - ROCm support for UCX is available
  - ROCm UCX backends for Open MPI (link), and MPICH
  - UCX is ROCm-aware and ROCm technologies are used directly to implement various network operation primitives.

In addition to ROCm-aware MPI and depending on the application, MPI binding might be necessary to get great performance (when scaling)

MPI Application Binding(1)

CPU-GPU binding might be necessary when scaling.
An example of binding script is available in ROCm™

Script Location: $ROCM_PATH/llvm/bin/gpurun
Example: /opt/rocm-5.4.3/llvm/bin/gpurun

Distributing GPUs and their CUs across multiple ranks of an MPI job

Wrapper script to execute a GPU application including OpenMPI GPU applications

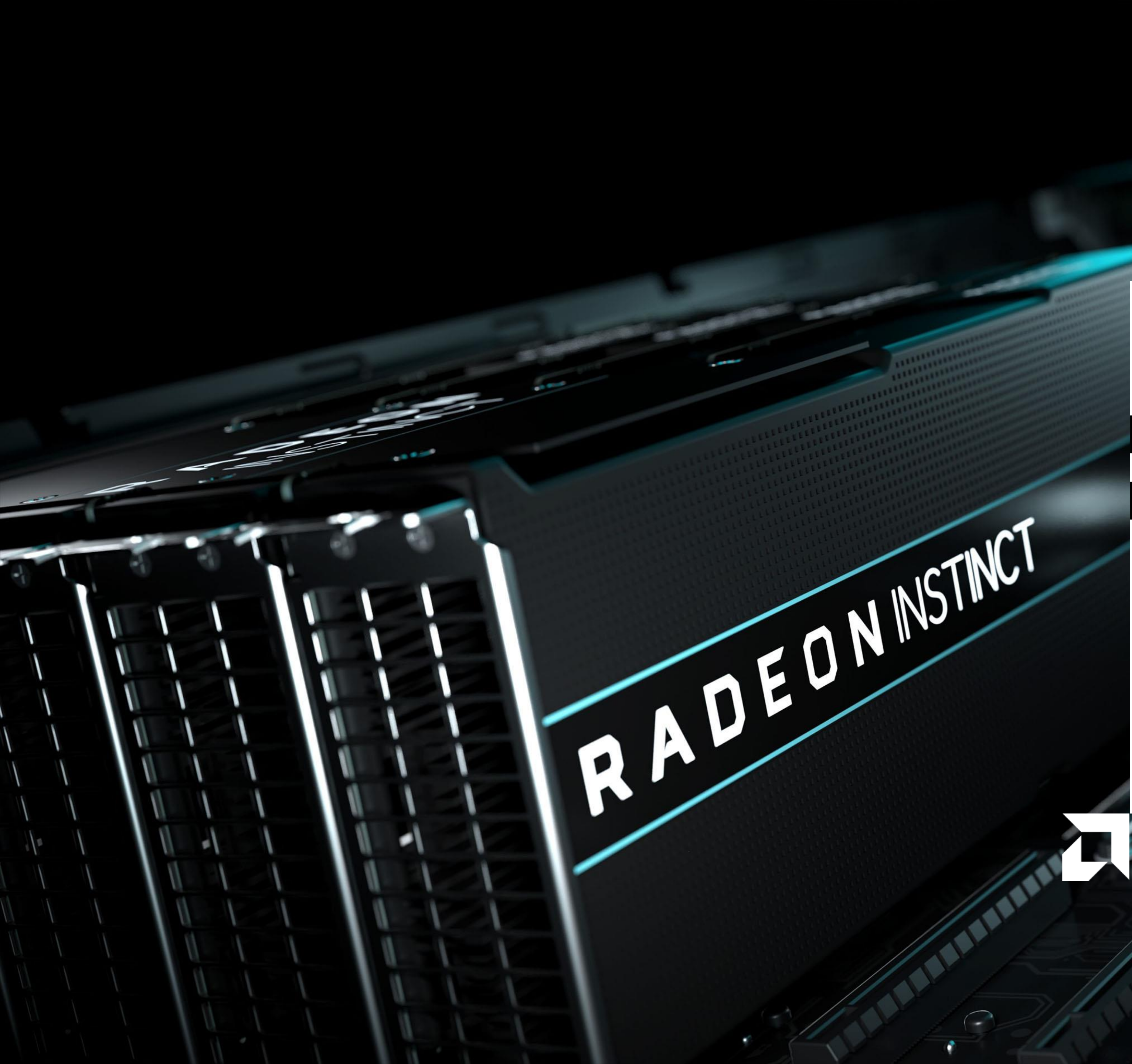
- This script launches the application with the Linux® 'taskset' utility to limit the application process to only CPUs in the same NUMA domain as the specified GPU.
- Sets environment variable ROCM_VISIBLE_DEVICES to specify the selected GPU.
- Sets OMPX_TARGET_TEAM_SLOTS to the number of CUs available to the process.
- If necessary, it sets HSA_CU_MASK to the subset of CUs for the specified OpenMPI rank when more than one OpenMPI rank will utilize the same GPU.

MPI Application Binding(2)

```
#  Example Setup:
#    Use a dummy application with no args
#      _appbin=true
#      _appargs=""
#    To get stats from rank 0 set GPURUN_VERBOSE to 1
#      export GPURUN_VERBOSE=1
#    For large numbers of ranks, increase slots with a hosfile.
#      _host_file="/tmp/host_file$$"
#      echo "`hostname` slots=64" >$_host_file

#  Usage Examples:
$     gpurun $_appbin $_appargs
#     mpirun -np  4 gpurun $_appbin $_appargs
#     mpirun -np  8 gpurun $_appbin $_appargs
#     mpirun -np 60 -hostfile $_host_file gpurun $_appbin $_appargs
```

Runtime Report for Performance Investigation



Generating Debug info (or Optimization Report)

- AMD AOMP/clang/LLVM compilers
  - To emit debugging symbols : -g
  - LIBOMPTARGET_KERNEL_TRACE= 1 | 2
    - Enable tracing of offload kernels on the GPU, shows kernel invocations, including:
      - n=1: kernel name at assembly level and number of teams, thread limits, register usage.
      - n=2: Same as n=1 data plus data transfers and mapped pointers, including per-kernel timing information.
- AMD HIP log info
  - AMD_LOG_LEVEL=0|1|2|3|4 (NONE, ERROR, WARNING, INFO, DEBUG) on AMD hardware to disable or enable different HIP logging
- Cray Compilers
  - CRAY_ACC_DEBUG=1 | 2 | 3

Outputs are different. This talk focuses on LIBOMPTARGET_KERNEL_TRACE

LIBOMPTARGET_KERNEL_TRACE is especially good for both compiler regression tracking and quick performance gap analysis

Generating Debug info (or Optimization Report)

```
void saxpy(int n, float a, float *restrict x, float *restrict y)
{
#pragma omp target teams distribute parallel for map(to: x[0:n]) map(tofrom: y[0:n]) num_teams(208)
thread_limit(1024)
	for (int i = 0; i < n; i++)
		y[i] = a*x[i] + y[i];
}
```

Computed avg time

```
double start = omp_get_wtime();

	saxpy(n, a, x, y);

double stop= omp_get_wtime();
```

```
[jnoudoho@TheraC60 saxpy]$ ./codelet_ofteam 212992000
-Execution Time in Seconds: avg=0.271770
```

The reported execution time comprises multiple elements

Generating Debug info: AMD clang/LLVM Example

[jnoudoho@TheraC60 saxpy]$ export LIBOMPTARGET_KERNEL_TRACE=1

[jnoudoho@TheraC60 saxpy]$ ./codelet_ofteam 212992000
DEVID: 0 SGN:2 ConstWGSize:1024 args: 4 teamsXthrds:(208X1024) reqd:(208X1024) lds_usage:68B sgpr_count:23 vgpr_count:15 sgpr_spill_count:0 vgpr_spill_count:0 tripcount:212992000 rpc:0 n:__omp_offloading_36_41e575c1_saxpy_l15

---**DESCRIPTION**-------------------------------------

DEVID: gpu# (or gpuid)
WGSize (Workgroup Size)
teamsXthrds (num_teams and thread_limit values)
lds_usage (Local Data Shared used)
Assembly code metrics:

- sgpr_count
- vgpr_count
- sgpr_spill_count
- vgpr_spill_count
- Tripcount (Loop tripcount)
- Function name (__omp_offloading_33_df175f_saxpy_l15)

Generating Debug info: AMD clang/LLVM Example

```
[jnoudoho@TheraC60 saxpy]$ export LIBOMPTARGET_KERNEL_TRACE=2

[jnoudoho@TheraC60 saxpy]$ ./codelet_ofteam 212992000
Call         __tgt_rtl_number_of_devices:        0us              8 )
Call           __tgt_rtl_is_valid_binary:       12us              1 (0x000000204cb0)
Call             __tgt_rtl_init_requires:        0us              1 (               1)
Call               __tgt_rtl_init_device:        6us              0 (               0)
Call               __tgt_rtl_load_binary:     2093us 0x00000208afd0 (               0, 0x000000204cb0)
Call                __tgt_rtl_data_alloc:      108us 0x7f5c54800000 (               0,      851968000, 0x7f5c876fe010)
Call         __tgt_rtl_data_submit_async:    15730us              0 (               0, 0x7f5c54800000, 0x7f5c876fe010,      851968000, 0x
7ffd244cc8a0)
Call                __tgt_rtl_data_alloc:      159us 0x7f5beea00000 (               0,      851968000, 0x7f5cba37f010)
Call         __tgt_rtl_data_submit_async:    16257us              0 (               0, 0x7f5beea00000, 0x7f5cba37f010,      851968000, 0x
7ffd244cc8a0)
Call __tgt_rtl_run_target_team_region_async:      245us              0 (               0, 0x00000208fbe0, 0x0000020a1600, 0x00000208fed0,
              4,             208,            1024,      212992000, 0x7ffd244cc8a0)
Call       __tgt_rtl_data_retrieve_async:   111413us              0 (               0, 0x7f5c876fe010, 0x7f5c54800000,      851968000, 0x
7ffd244cc8a0)
Call               __tgt_rtl_synchronize:   123621us              0 (               0, 0x7ffd244cc8a0)
Call               __tgt_rtl_data_delete:     1835us              0 (               0, 0x7f5beea00000)
Call               __tgt_rtl_data_delete:       70us              0 (               0, 0x7f5c54800000)
Call           __tgt_rtl_is_valid_binary:        6us              1 (0x000000204cb0)
DEVID: 0 SGN:2 ConstWGSize:1024 args: 4 teamsXthrds:( 208X1024) reqd:( 208X1024) lds_usage:68B sgpr_count:23 vgpr_count:15 sgpr_spill_c
ount:0 vgpr_spill_count:0 tripcount:212992000 rpc:0 n:__omp_offloading_36_41e575c1_saxpy_l15
```

Communication between host and device(target)

Generating Debug info: AMD clang/LLVM Example

```
[jnoudoho@TheraC60 saxpy]$ export LIBOMPTARGET_KERNEL_TRACE=2

[jnoudoho@TheraC60 saxpy]$ ./codelet_ofteam 212992000
Call         __tgt_rtl_number_of_devices:       0us             8 )
Call           __tgt_rtl_is_valid_binary:      12us             1 (0x000000204cb0)
Call             __tgt_rtl_init_requires:       0us             1 (               1)
Call               __tgt_rtl_init_device:       6us             0 (               0)
Call               __tgt_rtl_load_binary:    2093us 0x00000208afd0 (               0, 0x000000204cb0)
Call                __tgt_rtl_data_alloc:     108us 0x7f5c54800000 (               0,      851968000, 0x7f5c876fe010)
Call         __tgt_rtl_data_submit_async:   15730us             0 (               0, 0x7f5c54800000, 0x7f5c876fe010,      851968000, 0x
7ffd244cc8a0)
Call                __tgt_rtl_data_alloc:     159us 0x7f5beea00000 (               0,      851968000, 0x7f5cba37f010)
Call         __tgt_rtl_data_submit_async:   16257us             0 (               0, 0x7f5beea00000, 0x7f5cba37f010,      851968000, 0x
7ffd244cc8a0)
Call __tgt_rtl_run_target_team_region_async:      245us             0 (               0, 0x00000208fbe0, 0x0000020a1600, 0x00000208fed0,
               4,            208,           1024,      212992000, 0x7ffd244cc8a0)
Call       __tgt_rtl_data_retrieve_async:  111413us             0 (               0, 0x7f5c876fe010, 0x7f5c54800000,      851968000, 0x
7ffd244cc8a0)
Call               __tgt_rtl_synchronize:  123621us             0 (               0, 0x7ffd244cc8a0)
Call               __tgt_rtl_data_delete:    1835us             0 (               0, 0x7f5beea00000)
Call               __tgt_rtl_data_delete:      70us             0 (               0, 0x7f5c54800000)
Call           __tgt_rtl_is_valid_binary:       6us             1 (0x000000204cb0)
DEVID: 0 SGN:2 ConstWGSize:1024 args: 4 teamsXthrds:( 208X1024) reqd:( 208X1024) lds_usage:68B sgpr_count:23 vgpr_count:15 sgpr_spill_c
ount:0 vgpr_spill_count:0 tripcount:212992000 rpc:0 n:__omp_offloading_36_41e575c1_saxpy_l15
```

Collecting target offload runtime data (data transfer times, invocation times, per-kernel timing information)

Generating Debug info: AMD clang/LLVM Example

Pure Codelet Execution Time =__tgt_rtl_run_target_team_region_async: 245us

Offload Tax = Sum of [

__tgt_rtl_data_alloc:	108us
__tgt_rtl_data_submit_async:	15730us
__tgt_rtl_data_alloc:	159us
__tgt_rtl_data_submit_async:	16257us
__tgt_rtl_data_retrieve_async:	111413us
__tgt_rtl_synchronize:	123621us
__tgt_rtl_data_delete:	1835us
__tgt_rtl_data_delete:	70us

]

Useful metrics for performance modeling, poor performance investigation or performance gap analysis (including compiler regression tracking)

Example of analysis in next slides

Target dependent OpenMP® Plugin API Interface

Few functions defined for communicating between target independent OpenMP offload runtime library (libomptarget i.e. host) and target dependent plugin (target devices).

int32_t __tgt_rtl_device_type() - return an integer that identifies the device type
int32_t __tgt_rtl_number_of_devices() - Return the number of available devices
int32_t __tgt_init_device() - initialization of the specified device
__tgt_target_table* __tgt_rtl_load_binary() - load an executable section to device
void* __tgt_rtl_data_alloc() - memory allocation on target device
int32_t __tgt_rtl_data_delete() - de-allocate (or delete) memory on device
int32_t __tgt_rtl_data_submit_async() - asynchronously pass the data content to the specified device
int32_t __tgt_rtl_data_retrieve_async() - retrieve (or get) the data content from device asynchronously
int32_t __tgt_rtl_run_target_region() - Transfer control to the offloaded code entry then run this code on device
int32_t __tgt_rtl_run_target_team_region_async() - run offloaded code on device asynchronously

Reducing Data Transfer between Host and Device

Some technics used to minimize data transfers includes:

- Using “target enter data” and “target exit data” directives when variables are used by multiple target constructs

- Choosing the right map-type for a mapped variable (read-only, write, alloc)

- It is recommended to NOT map read-only scalar variables (to avoid unnecessary memory allocation on the device and copying data from host to device):
  - Consider instead listing scalar variables in a “firstprivate” clause on the target construct or not list in any clause at all

    Example: Loop bounds(lower bound, upper bound, or step)

How do you check the impact of these changes on your kernel performance?

1- export LIBOMPTARGET_KERNEL_TRACE=2
2- Run the kernel before and after these changes, then compare generated profiles

Reducing Memory Allocation on the Target GPU

map(to:) clause may not be the most efficient way to allocate memory for a variable (especially for a temporary variable) on the device:

- use the map(alloc:) clause instead
- place the declarations of the arrays between “declare target” and “end declare target” directives.

Example: Let's consider temporary work arrays A[SIZE], B[SIZE] that are moved to GPU using map(to:)

REPLACE map(to: A[0:SIZE], B[0,SIZE])

BY map(alloc: A[0:SIZE], B[0,SIZE])

OR BY

```
#pragma omp declare target
double A[SIZE], B[SIZE];
#pragma omp end declare target
```

How do you check the impact of these changes on your kernel performance?

1- export LIBOMPTARGET_KERNEL_TRACE=2

2- Run the kernel before and after these changes, then compare generated profiles

Summary & Conclusions

- Four compilers exist to effectively offload a kernel or an application onto AMD GPUs: AMD ROCmCC, Cray, GNU, Siemens

- Many libraries are available. Use them where possible
  - Use “module avail” command to check available libraries
  - Use “module show <module name>” to see the installation paths if needed

- Some modules may not interact well with compilers
  - Users need to provide both headers (include) and library (lib) paths, and certain libraries manually

- Learn from the compiler help

- Learn from the compiler verbose output (-v)

- Learn from debug information

- Learn from sbatch/srun options

Disclaimer

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

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

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

Questions?

AMD

Backup

- **Assembly File Generation**

  - `clang -g --save-temps -O2 -fopenmp --offload-arch=<gfx90a/gfx908>  helloworld.c`

  - `hipcc -g --save-temps -c helloworld.cpp`

  - From the generated assembly file, on can see register pressure, occupancy, etc..

- Alternative to get resource usage is to use the -Rpass-analysis=kernel-resource-usage compiler flag

Some srun Options

Options	Description (man srun)
-N, --nodes=<minnodes[-maxnodes]>	Request that a minimum of minnodes nodes be allocated to this job. A maximum node count may also be specified with maxnodes. If only one number is specified, this is used as both the minimum and maximum node count.
-n, --ntasks=<number>	Specify the number of tasks to run. Request that srun allocate resources for ntasks tasks. The default is 1 task per node, but note that the --cpus-per-task option will change this default. This option applies to job and step allocations.
-c, --cpus-per-task=<ncpus>	Request that ncpus be allocated per process (default is 1).
--gpus-per-task	Specify the number of GPUs required for the job on each task to be spawned in the job's resource allocation.
--gpu-bind=closest	Bind each task to the GPU(s) which are closest. In NUMA environment, each task may be bound to more than one GPU (i.e all GPUs in that NUMA environment)

Some srun Options

Options	Description (man srun)
--gpu-bind=map_gpu:<list>	Bind tasks to specific GPUs by setting GPU masks on tasks (or ranks) as specified where <list> is <gpu_id_for_task_0>,<gpu_id_for_task_1>,.... If the number of tasks (or ranks) exceeds the number of elements in this list, elements in the list will be reused as needed starting from the beginning of the list. To simplify support for large task counts, the lists may follow a map with an asterisk and repetition count. (For example map_gpu:0*4,1*4)
--ntasks-per-gpu=<ntasks>	Request that there are ntasks tasks invoked for every GPU.
--distribution=<value>[:<value>][:<value>]	Specify the distribution of MPI ranks across compute nodes, sockets, and cores, respectively. The default value for each distribution is specified by *

AMD together we advance_

Example of Binding(1)

- case "${OMPI_COMM_WORLD_LOCAL_RANK}" in
- 0)
- exec numactl --physcpubind=0-15,128-143 --membind=0 "${@}"
- ;;
- 1)
- exec numactl --physcpubind=16-31,144-159 --membind=1 "${@}"
- ;;
- 2)
- exec numactl --physcpubind=32-47,160-175 --membind=2 "${@}"
- ;;
- *)
- echo ==
- echo "ERROR: Unknown local rank $OMPI_COMM_WORLD_LOCAL_RANK"
- echo ==
- exit 1
- ;;
- esac

Example of Binding(2)

- `srun -n 4 -N 1 --cpu-bind=verbose,cores -l -c 16 --mpi=cray_shasta --gpu-bind=verbose,closest --gpus-per-task=1 --exclusive  <binaire>`

- `srun -n 16 -N 1 --cpu-bind=verbose,cores -l -c 8 --mpi=cray_shasta --gpu-bind=verbose,closest --gres=gpu:8 --exclusive <binaire>`

- `srun -n 4 -N 2 --ntasks-per-node=2 --cpu-bind=verbose,cores -l -c 16 --mpi=cray_shasta --gpu-bind=verbose,closest --gpus-per-task=1 -m block:block:block --exclusive <binaire>`