Contents

AMD @HLRS Workshop

Basics

SSH-Key Generation

Generate SSH key as shown below

```
cd $HOME
ssh-keygen -t ed25519 -N ''
cat $HOME/.ssh/id_ed25519.pub
```

Logging in

```
ssh <username>@aac1.amd.com -i id_ed25519 -p ####
```

The port number will be established when the container environment starts up and will be given out at that time.

At first login, you will be presented with the AMD Accelerator Cloud use agreement form. It covers the terms of use of the compute hardware as well as how we will handle your data.

To simplify the login even further, you can add the following to your `.ssh/config` file.

The `ServerAlive*` lines in the config file may be added to avoid timeouts when idle.

```
# AMD AAC cluster
Host aac
   User <USERNAME>
   Hostname aac1.amd.com
   IdentityFile id_ed25519 -- can put full path as well -- $HOME/.ssh/id_ed25519
   Port #### -- but this might change, so add on the command line shown below
   ServerAliveInterval 600
   ServerAliveCountMax 30
```

and then login using

```
ssh aac -p ####
```

Directories and Files

Persistent storage is at `/datasets/teams/hackathon-testing/<userid>`. Your home directory will be set to this directory.

```
$HOME=/datasets/teams/hackathon-testing/<userid>
```

Files in that directory will persist across container starts and stops and even be available from another container with the same userid on systems at the same hosting location.

You should be able to copy files in or out with the `scp` command.

Copy into AAC from your local system

```
scp -i <path>/<keyfile> -P #### <file> USER@aac1.amd.com:~/<path>/<file>
```

Copy from AAC to your local system

```
scp -i <path>/<keyfile> -P #### USER@aac1.amd.com:~/path/to/your/file ./
```

To copy files in or out of the container, you can also use `rsync` as shown below:

```
rsync -avz -e "ssh -i <path>/<keyfile> -p ####" <file> <USER>@aac1.amd.com:~/path/to/your/files
```

Explore environment

This container is based on the Ubuntu 22.04 Operating System with the ROCm 5.6.0 software stack. It contains multiple versions of AMD, GCC, and LLVM compilers, hip libraries, GPU-Aware MPI (OpenMPI), and AMD Profiling tools with perfetto and graphana. The container also has modules set up with the lua modules package and a slurm package and configuration. It includes the following additional packages:

- emacs
- vim
- miniconda
- autotools
- cmake
- tmux
- boost
- eigen
- fftw
- gmp
- gsl
- hdf5-openmpi
- lapack
- magma
- matplotlib
- parmetis
- mpfr
- mpi4py
- openblas
- openssl
- swig
- numpy
- scipy
- h5sparse

Check modules available:

```
module avail
```

Output list of modules avail command

- amdclang/5.6.0 – AMD Clang compiler with OpenMP
- clang/14-15 – Clang/LLVM standard compiler installations
- gcc/11-12 – GCC standard compiler installations
- hipfort/0.4 – Fortran wrappers for hip calls
- openmpi/4.1.5 – GPU-aware MPI
- omniperf/1.0.8 – AMD performance analysis tool
- omnitrace/1.9.0 – AMD trace profiler
- rocm/5.6.0 – ROCm software stack including hip and hip libraries

Compiler modules set the C, CXX, FC flags. Only one compiler module can be loaded at a time. hipcc is in the path when the rocm module is loaded.

Slurm

The SLURM configuration is for a single queue that is shared with the rest of the node.

```
sinfo
```

PARTITION	AVAIL	TIMELIMIT	NODES	STATE	NODELIST
LocalQ	up	2:00:00	1	idle	localhost

The SLURM salloc command may be used to acquire a long term session that exclusively grants access to one or more GPUs. Alternatively, the srun or sbatch commands may be used to acquire a session with one or more GPUs and only exclusively use the session for the life of the run of an application. squeue and sinfo will show information about the current state of the SLURM system.

Exercise examples

The exercise examples are preloaded into the /Examples directory. Copy the files into your home directory with

```
mkdir -p $HOME/HPCTrainingExamples
scp -pr /Examples/HPCTrainingExamples/* $HOME/HPCTrainingExamples/
```

or

```
scp -pr /users/examples/HPCTrainingExamples/* $HOME/HPCTrainingExamples
```

If you need to refer to the examples separately, they are at the repository below.

Examples repo

Alternatively, you can get the examples from our repo. This repo contains all the code that we will use for the exercises that follow

```
cd $HOME
git clone https://github.com/amd/HPCTrainingExamples.git
```

Introduction to HIP Exercises

`git clone https://github.com/amd/HPCTrainingExamples.git`

For the first interactive example, get an slurm interactive session

‘salloc -N 1 -p LocalQ –gpus=1 -t 10:00

Basic examples

`cd HPCTrainingExamples/HIP/vectorAdd`

Examine files here - README, Makefile, CMakeLists.txt and vectoradd.hip. Notice that Makefile requires `ROCM_PATH` to be set. Check with module show rocm or echo `$ROCM_PATH` Also, the Makefile builds and runs the code. We'll do the steps separately. Check also the HIPFLAGS in the Makefile. There is also a CMakeLists.txt file to use for a cmake build.

For the portable Makefile system

```
module load rocm

make vectoradd
./vectoradd
```

Pro tip for Makefile builds. Run `make clean` before `make` to be sure nothing is left over from a previous build.

This example also runs with the cmake system

```
module load rocm

mkdir build && cd build
cmake ..
make
./vectoradd
```

Pro tip for cmake builds. To rebuild after changing CMake options or using a different compiler, either

- Remove the CMakeCache.txt, or
- clean out all files from the ./build directory

We can use a SLURM submission script, let's call it `hip_batch.sh`. There is a sample script for some systems in the example directory.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -p LocalQ
#SBATCH --gpus=1
#SBATCH -t 10:00

module load rocm
cd $HOME/HPCTrainingExamples/HIP/vectorAdd

make vectoradd
./vectoradd
```

Submit the script `sbatch hip_batch.sh`

Check for output in `slurm-<job-id>.out` or error in `slurm-<job-id>.err`

To use the cmake option in the batch file, change the build to

```
mkdir build && cd build
cmake ..
make
./vectoradd
```

Now let's try the hip-stream example. This example is from the original McCalpin code as ported to CUDA by Nvidia. This version has been ported to use HIP.

```
module load rocm
cd $HOME/HPCTrainingExamples/HIP/hip-stream
make
./stream
```

Note that it builds with the hipcc compiler. You should get a report of the Copy, Scale, Add, and Triad cases.

On your own:

1. Check out the saxpy example in `HPCTrainingExamples/HIP`
2. Write your own kernel and run it
3. Test the code on an Nvidia system – Add `HIPCC=nvcc` before the make command or `-DCMAKE_GPU_RUNTIME=CUDA` to the cmake command. (See README file)

More advanced HIP makefile

The jacobi example has a more complex build that incorporates MPI. The original Makefile has not been modified, but a CMakeLists.txt has been added to demonstrate a portable cmake build. From an interactive session, build the example.

```
cd $HOME/HPCTrainingExamples/HIP/jacobi

module load rocm
module load openmpi

mkdir build && cd build
cmake ..
make
```

Since we will be running on two MPI ranks, you will need to alloc 2 GPUs for a quick run. Exit your current allocation with `exit` and then get the two GPUs. Keep the requested time short to avoid tying up the GPUs so others can run the examples. The requested time shown is in the format hours:minutes:seconds so it is for one minute.

```
salloc -p LocalQ --gpus=2 -n 2 -t 00:01:00
module load rocm openmpi
mpirun -n 2 ./Jacobi_hip -g 2
```

Porting Applications to HIP

Hipify Examples

Exercise 1: Manual code conversion from CUDA to HIP (10 min)

Choose one or more of the CUDA samples in `HPCTrainingExamples/HIPIFY/mini-nbody/cuda` directory. Manually convert it to HIP. Tip: for example, the cudaMalloc will be called hipMalloc. You can choose from `nbody-block.cu, nbody-orig.cu, nbody-soa.cu`

You'll want to compile on the node you've been allocated so that hipcc will choose the correct GPU architecture.

Exercise 2: Code conversion from CUDA to HIP using HIPify tools (10 min)

Use the `hipify-perl` script to "hipify" the CUDA samples you used to manually convert to HIP in Exercise 1. hipify-perl is in `$ROCM_PATH/hip/bin` directory and should be in your path.

First test the conversion to see what will be converted

```
hipify-perl -examine nbody-orig.cu
```

You'll see the statistics of HIP APIs that will be generated. The output might be different depending on the ROCm version.

```
[HIPIFY] info: file 'nbody-orig.cu' statistics:
  CONVERTED refs count: 7
  TOTAL lines of code: 91
  WARNINGS: 0
[HIPIFY] info: CONVERTED refs by names:
  cudaFree => hipFree: 1
  cudaMalloc => hipMalloc: 1
  cudaMemcpyDeviceToHost => hipMemcpyDeviceToHost: 1
  cudaMemcpyHostToDevice => hipMemcpyHostToDevice: 1
```

`hipify-perl` is in `$ROCM_PATH/hip/bin` directory and should be in your path. In some versions of ROCm, the script is called `hipify-perl`.

Now let's actually do the conversion.

```
hipify-perl nbody-orig.cu > nbody-orig.cpp
```

Compile the HIP programs.

```
hipcc -DSHMOO -I ../ nbody-orig.cpp -o nbody-orig
```

The `#define SHMOO` fixes some timer printouts. Add `--offload-arch=<gpu_type>` to specify the GPU type and avoid the autodetection issues when running on a single GPU on a node.

- Fix any compiler issues, for example, if there was something that didn't hipify correctly.
- Be on the lookout for hard-coded Nvidia specific things like warp sizes and PTX.

Run the program

```
./nbody-orig
```

A batch version of Exercise 2 is:

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --ntasks=1
#SBATCH --gpus=1
#SBATCH -p LocalQ
#SBATCH -A <project id>
#SBATCH -t 00:10:00

pwd
module load rocm

cd HPCTrainingExamples/HIPIFY/mini-nbody/cuda
hipify-perl -print-stats nbody-orig.cu > nbody-orig.cpp
hipcc -DSHMOO -I ../ nbody-orig.cpp -o nbody-orig
./nbody-orig
```

Notes:

- Hipify tools do not check correctness
- `hipconvertinplace-perl` is a convenience script that does `hipify-perl -inplace -print-stats` command

Mini-App conversion example

Load the proper environment

```
cd $HOME/HPCTrainingExamples/HIPFY/
module load rocm
```

Get the CUDA version of the Pennant mini-app.

```
wget https://asc.llnl.gov/sites/asc/files/2020-09/pennant-singlenode-cude.tgz
tar -xzvf pennant-singlenode-cude.tgz

cd PENNANT

hipexamine-perl.sh
```

And review the output

Now do the actual conversion. We want to do the conversion for the whole directory tree, so we'll use hipconvertinplace-sh

```
hipconvertinplace-perl.sh
```

We want to use `.hip` extensions rather than `.cu`, so change all files with `.cu` to `.hip`

```
mv src/HydroGPU.cu src/HydroGPU.hip
```

Now we have two options to convert the build system to work with both ROCm and CUDA

Makefile option

First cut at converting the Makefile. Testing with `make` can help identify the next step.

- Change all occurances of CUDA to HIP (e.g. sed -i 's/cuda/hip/g' Makefile)
- Change the CXX variable to `clang++` located in `${ROCM_PATH}/llvm/bin/clang++`
- Change all the HIPC variables to HIPCC
- Change HIPCC to point to hipcc
- Change HIPCCFLAGS with CUDA options to HIPCCFLAGS_CUDA
- Remove `-fast` and `-fno-alias` from the CXXFLAGS_OPT
- Change all `.cu` to `.hip` in the Makefile

Now we are just getting compile errors from the source files. We will have to do fixes there. We'll tackle them one-by-one.

The first errors are related to the double2 type.

```
compiling src/HydroGPU.hip
(CPATH=;hipcc -O3 -I.  -c -o build/HydroGPU.o src/HydroGPU.hip)
In file included from src/HydroGPU.hip:14:
In file included from src/HydroGPU.hh:16:

src/Vec2.hh:35:8: error: definition of type 'double2' conflicts with type alias of the
same name

struct double2
       ^
```

```
/opt/rocm-5.6.0/include/hip/amd_detail/amd_hip_vector_types.h:1098:1: note: 'double2'
declared here

__MAKE_VECTOR_TYPE__(double, double);

^

/opt/rocm-5.6.0/include/hip/amd_detail/amd_hip_vector_types.h:1062:15: note: expanded
from macro '__MAKE_VECTOR_TYPE__'

        using CUDA_name##2 = HIP_vector_type<T, 2>;\

              ^

<scratch space>:316:1: note: expanded from here
double2
```

HIP defines double2. Let's look at Vec2.hh. At line 33 where the first error occurs. We see an `#ifdef __CUDACC__` around a block of code there. We also need the #ifdef to include HIP as well. Let's check the available compiler defines from the presentation to see what is available. It looks like we can use `__HIP_DEVICE_COMPILE__` or maybe `__HIPCC__`.

Change line 33 in Vec2.hh to #ifndef `__HIPCC__`

The next error is about function attributes that are incorrect for device code.

```
compiling src/HydroGPU.hip
(CPATH=;hipcc -O3 -I.  -c -o build/HydroGPU.o src/HydroGPU.hip
src/HydroGPU.hip:168:23: error: no matching function for call to 'cross
    double sa = 0.5 * cross(px[p2] - px[p1],  zx[z] - px[p1]);
                      ^~~~

src/Vec2.hh:206:15: note: candidate function not viable: call to __host__ function from
__device__ function
```

The FNQUALIFIER macro is what handles the attributes in the code. We find that defined at line 22 and again we see a `#ifdef __CUDACC__`. It is another `#ifdef __CUDACC__`. We can see that we need to pay attention to all the CUDA ifdef statements.

Change line 22 to `#ifdef __HIPCC__`

Finally we get an error about already defined operators on double2 types. These appear to be defined in HIP, but not in CUDA. So we change line 84

```
compiling src/HydroGPU.hip
(CPATH=;hipcc -O3 -I.  -c -o build/HydroGPU.o src/HydroGPU.hip)

src/HydroGPU.hip:149:15: error: use of overloaded operator '+=' is ambiguous (with operand
types 'double2' (aka 'HIP_vector_type<double, 2>') and 'double2')

        zxtot += ctemp2[sn];
        ~~~~~ ^  ~~~~~~~~~~
/opt/rocm-5.6.0/include/hip/amd_detail/amd_hip_vector_types.h:510:26: note: candidate function
        HIP_vector_type& operator+=(const HIP_vector_type& x) noexcept
                         ^

src/Vec2.hh:88:17: note: candidate function
inline double2& operator+=(double2& v, const double2& v2)
```

Change line 85 to `#elif defined(__CUDACC__)`

Now we start getting errors for HydroGPU.hip. The first is for the atomicMin function. It is already defined in HIP, so we need to add an ifdef for CUDA around the code.

```
compiling src/HydroGPU.hip
(CPATH=;hipcc -O3 -I.  -c -o build/HydroGPU.o src/HydroGPU.hip)
src/HydroGPU.hip:725:26: error: static declaration of 'atomicMin' follows non-static declaration
static __device__ double atomicMin(double* address, double val)
                         ^
/opt/rocm-5.6.0/include/hip/amd_detail/amd_hip_atomic.h:478:8: note: previous definition is here
double atomicMin(double* addr, double val) {
       ^
1 error generated when compiling for gfx90a.
```

Add `#ifdef __CUDACC__/endif` to the more block of code in `HydroGPU.hip` from line 725 to 737

We finally got through the compiler errors and move on to link errors

```
linking build/pennant

/opt/rocm-5.6.0//llvm/bin/clang++ -o build/pennant build/ExportGold.o build/ImportGMV.o
build/Parallel.o build/WriteXY.o build/HydroBC.o build/QCS.o build/TTS.o build/main.o
build/Mesh.o build/InputFile.o build/GenMesh.o build/Driver.o build/Hydro.o build/PolyGas.o
build/HydroGPU.o -L/lib64 -lcudart

ld.lld: error: unable to find library -lcudart
```

In the Makefile, change the LDFLAGS while keeping the old settings for when we set up the switch between GPU platforms.

```
LDFLAGS_CUDA := -L$(HIP_INSTALL_PATH)/lib64 -lcudart
LDFLAGS := -L${ROCM_PATH}/hip/lib -lamdhip64
```

We then get the link error

```
linking build/pennant

/opt/rocm-5.6.0//llvm/bin/clang++ -o build/pennant build/ExportGold.o build/ImportGMV.o
build/Parallel.o build/WriteXY.o build/HydroBC.o build/QCS.o build/TTS.o build/main.o
build/Mesh.o build/InputFile.o build/GenMesh.o build/Driver.o build/Hydro.o build/PolyGas.o
build/HydroGPU.o -L/opt/rocm-5.6.0//hip/lib -lamdhip64

ld.lld: error: undefined symbol: hydroInit(int, int, int, int, int, double, double,
double, double, double, double, double, double, double, int, double const*, int, double
const*, double2 const*, double2 const*, double const*, double const*, double const*,
double const*, double const*, double const*, double const*, int const*, int const*, int
const*, int const*, int const*, int const*)

>>> referenced by Hydro.cc
>>>               build/Hydro.o:(Hydro::Hydro(InputFile const*, Mesh*))


ld.lld: error: undefined symbol: hydroGetData(int, int, double2*, double*, double*, double*)
>>> referenced by Hydro.cc
>>>               build/Hydro.o:(Hydro::getData())
```

This one is a little harder. We can get more information by using `nm build/Hydro.o |grep hydroGetData` and `nm build/HydroGPU.o |grep hydroGetData`. We can see that the subroutine signatures are slightly different due to the double2 type on the host and GPU. You can also switch the compiler from clang++ to g++ to get a slightly more informative error. We are in a tough spot here because we need the hipmemcpy in the body of the subroutine, but the types for double2 are for the device instead of the host. One solution is to just compile and link everything with hipcc, but we really don't want to do that if only one routine needs to use the device compiler. So we cheat by declaring the prototype arguments as `void *` and casting the type in the call with `(void *)`. The types are really the same and it is just arguing with the compiler.

```
nm build/Hydro.o |grep hydroGetData
                 U _Z12hydroGetDataiiP7double2PdS1_S1_
nm build/HydroGPU.o |grep hydroGetData
0000000000003750 T _Z12hydroGetDataiiP15HIP_vector_typeIdLj2EEPdS2_S2_
```

In HydroGPU.hh

- Change line 38 and 39 to from `const double2*` to `const void*`
- Change line 62 from `double2*` to `void*`

In HydroGPU.hip

- Change line 1031 and 1032 to `const void*`
- Change line 1284 to `const void*`

In Hydro.cc

- Add `(void *)` before the arguments on lines 59, 60, and 145

Now it compiles and we can test the run with

```
build/pennant test/sedovbig/sedovbig.pnt
```

So we have the code converted to HIP and fixed the build system for it. But we haven't accomplished our original goal of running with both ROCm and CUDA.

We can copy a sample portable Makefile from `HPCTrainingExamples/HIP/saxpy/Makefile` and modify it for this application.

```
EXECUTABLE = pennant
BUILDDIR := build
SRCDIR = src
all: $(BUILDDIR)/$(EXECUTABLE) test

.PHONY: test

OBJECTS =  $(BUILDDIR)/Driver.o $(BUILDDIR)/GenMesh.o $(BUILDDIR)/HydroBC.o
OBJECTS += $(BUILDDIR)/ImportGMV.o $(BUILDDIR)/Mesh.o $(BUILDDIR)/PolyGas.o
OBJECTS += $(BUILDDIR)/TTS.o $(BUILDDIR)/main.o $(BUILDDIR)/ExportGold.o
OBJECTS += $(BUILDDIR)/Hydro.o $(BUILDDIR)/HydroGPU.o $(BUILDDIR)/InputFile.o
OBJECTS += $(BUILDDIR)/Parallel.o $(BUILDDIR)/QCS.o $(BUILDDIR)/WriteXY.o

CXXFLAGS = -g -O3
HIPCC_FLAGS = -O3 -g -DNDEBUG

HIPCC ?= hipcc

ifeq ($(HIPCC), nvcc)
   HIPCC_FLAGS += -x cu
   LDFLAGS = -lcudadevrt -lcudart_static -lrt -lpthread -ldl
endif
ifeq ($(HIPCC), hipcc)
   HIPCC_FLAGS += -munsafe-fp-atomics
   LDFLAGS = -L${ROCM_PATH}/hip/lib -lamdhip64
endif

$(BUILDDIR)/%.d : $(SRCDIR)/%.cc
    @echo making depends for $<
    $(maketargetdir)
```

```
    @$(CXX) $(CXXFLAGS) $(CXXINCLUDES) -M $< | sed "1s![^ \t]\+\.o!$(@:.d=.o) $@!" >$@

$(BUILDDIR)/%.d : $(SRCDIR)/%.hip
    @echo making depends for $<
    $(maketargetdir)
    @$(HIPCC) $(HIPCCFLAGS) $(HIPCCINCLUDES) -M $< | sed "1s![^ \t]\+\.o!$(@:.d=.o) $@!" >$@

$(BUILDDIR)/%.o : $(SRCDIR)/%.cc
    @echo compiling $<
    $(maketargetdir)
    $(CXX) $(CXXFLAGS) $(CXXINCLUDES) -c -o $@ $<

$(BUILDDIR)/%.o : $(SRCDIR)/%.hip
    @echo compiling $<
    $(maketargetdir)
    $(HIPCC) $(HIPCC_FLAGS) -c $^ -o $@

$(BUILDDIR)/$(EXECUTABLE) : $(OBJECTS)
    @echo linking $@
    $(maketargetdir)
    $(CXX) $(OBJECTS) $(LDFLAGS) -o $@

test : $(BUILDDIR)/$(EXECUTABLE)
    $(BUILDDIR)/$(EXECUTABLE) test/sedovbig/sedovbig.pnt

define maketargetdir
    -@mkdir -p $(dir $@) > /dev/null 2>&1
endef

clean :
    rm -rf $(BUILDDIR)
```

To test the makefile,

```
make build/pennant
make test
```

or just `make` to both build and run the test

To test the makefile build system with CUDA (note that the system used for this training does not have CUDA installed so this exercise is left to the student)

```
module load cuda
HIPCC=nvcc CXX=g++ make
```

To create a cmake build system, we can copy a sample portable CMakeLists.txt and modify it for this applicaton.

HPCTrainingExamples/HIP/saxpy/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.21 FATAL_ERROR)
project(Pennant LANGUAGES CXX)
include(CTest)

set (CMAKE_CXX_STANDARD 14)

if (NOT CMAKE_BUILD_TYPE)
   set(CMAKE_BUILD_TYPE RelWithDebInfo)
```

```
endif(NOT CMAKE_BUILD_TYPE)

string(REPLACE -O2 -O3 CMAKE_CXX_FLAGS_RELWITHDEBINFO ${CMAKE_CXX_FLAGS_RELWITHDEBINFO})

if (NOT CMAKE_GPU_RUNTIME)
   set(GPU_RUNTIME "ROCM" CACHE STRING "Switches between ROCM and CUDA")
else (NOT CMAKE_GPU_RUNTIME)
   set(GPU_RUNTIME "${CMAKE_GPU_RUNTIME}" CACHE STRING "Switches between ROCM and CUDA")
endif (NOT CMAKE_GPU_RUNTIME)
# Really should only be ROCM or CUDA, but allowing HIP because it is the currently built-in option
set(GPU_RUNTIMES "ROCM" "CUDA" "HIP")
if(NOT "${GPU_RUNTIME}" IN_LIST GPU_RUNTIMES)
    set(ERROR_MESSAGE "GPU_RUNTIME is set to \"${GPU_RUNTIME}\".\nGPU_RUNTIME must be either HIP,
        ROCM, or CUDA.")
    message(FATAL_ERROR ${ERROR_MESSAGE})
endif()
# GPU_RUNTIME for AMD GPUs should really be ROCM, if selecting AMD GPUs
# so manually resetting to HIP if ROCM is selected
if (${GPU_RUNTIME} MATCHES "ROCM")
   set(GPU_RUNTIME "HIP")
endif (${GPU_RUNTIME} MATCHES "ROCM")
set_property(CACHE GPU_RUNTIME PROPERTY STRINGS ${GPU_RUNTIMES})

enable_language(${GPU_RUNTIME})
set(CMAKE_${GPU_RUNTIME}_EXTENSIONS OFF)
set(CMAKE_${GPU_RUNTIME}_STANDARD_REQUIRED ON)

set(PENNANT_CXX_SRCS src/Driver.cc src/ExportGold.cc src/GenMesh.cc src/Hydro.cc src/HydroBC.cc
                     src/ImportGMV.cc src/InputFile.cc src/Mesh.cc src/Parallel.cc src/PolyGas.cc
                     src/QCS.cc src/TTS.cc src/WriteXY.cc src/main.cc)

set(PENNANT_HIP_SRCS src/HydroGPU.hip)

add_executable(pennant ${PENNANT_CXX_SRCS} ${PENNANT_HIP_SRCS} )

# Make example runnable using ctest
add_test(NAME Pennant COMMAND pennant ../test/sedovbig/sedovbig.pnt )
set_property(TEST Pennant
             PROPERTY PASS_REGULAR_EXPRESSION "End cycle   3800, time = 9.64621e-01")

set(ROCMCC_FLAGS "${ROCMCC_FLAGS} -munsafe-fp-atomics")
set(CUDACC_FLAGS "${CUDACC_FLAGS} ")

if (${GPU_RUNTIME} MATCHES "HIP")
   set(HIPCC_FLAGS "${ROCMCC_FLAGS}")
else (${GPU_RUNTIME} MATCHES "HIP")
   set(HIPCC_FLAGS "${CUDACC_FLAGS}")
endif (${GPU_RUNTIME} MATCHES "HIP")

set_source_files_properties(${PENNANT_HIP_SRCS} PROPERTIES LANGUAGE ${GPU_RUNTIME})
set_source_files_properties(HydroGPU.hip PROPERTIES COMPILE_FLAGS ${HIPCC_FLAGS})

install(TARGETS pennant)
```

To test the cmake build system, do the following

```
mkdir build && cd build
cmake ..
make VERBOSE=1
ctest
```

Now testing for CUDA

```
module load cuda

mkdir build && cd build
cmake -DCMAKE_GPU_RUNTIME=CUDA ..
make VERBOSE=1
ctest
```

OpenMP Examples

The goal of this exercise is to offload simple OpenMP codelets onto AMD GPUs.

By default, GNU compilers are used to build these mini-apps that can then be executed on host (CPU).

There are two examples we'll work with, one in C and the other in Fortran.

```
~/HPCTrainingExamples/Pragma_Examples/OpenMP_CPU/C/Make/saxpy
~/HPCTrainingExamples/Pragma_Examples/OpenMP_CPU/Fortran/Make/freduce
```

In this next section, we will walk through how to convert the CPU OpenMP code to run on the GPU. You can also skip ahead past this section to just run the resulting GPU OpenMP code that is created.

Modifying CPU OpenMP code to run on the GPU

To make the changes, you will instruct the compiler to offload certain code sections (loops) within these mini-apps.

Find the C/C++ codelet (saxpy) example, `codelet.c`

```
cd ~/HPCTrainingExamples/Pragma_Examples/OpenMP_CPU/C/Make/saxpy
vim codelet.c
```

In `codelet.c`, replace

```
#pragma omp parallel for simd
```

with

```
#pragma omp target teams distribute parallel for simd map(to: x[0:n],y[0:n]) map(from: z[0:n])
```

In `Makefile`, - replace `gcc` by `amdclang` in `CC` - add `--offload-arch=gfx90a` to compiler flags in `CFLAGS` and `LDFLAGS` to enable offloading to the AMD MI200 series GPU
- Re-compile

Note that you may see warnings about loops not being vectorized in llvm version 16. A fix appears to have been made to llvm to suppress the message. You can safely ignore it in this example.

```
module load rocm
make
```

Run this codelet on an AAC node using an input size of your choice like 123456789.

```
./saxpy 231124421
```

If you have multiple GPUs, you can also specify the GPU to use by setting `ROCR_VISIBLE_DEVICES`

```
ROCR_VISIBLE_DEVICES=0 ./saxpy 231124421
```

Find the Fortran codelet

```
cd ~/HPCTrainingExamples/Pragma_Examples/OpenMP_CPU/Fortran/Make/freduce
vim freduce.F90
```

Add the following instruction just before the beginning of the innermost loop

```
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO SIMD REDUCTION(+:sum2) &
!$OMP MAP(TO:array(1:10))
```

Add the following instruction right after the end of the innermost loop code section

```
!$OMP END TARGET TEAMS DISTRIBUTE PARALLEL DO SIMD
```

Build with the AMD Fortran compiler: - replace the gfortran compiler with amdflang: export FC=amdflang - edit the Makefile and add `--offload-arch=gfx90a` to `LDFLAGS and FFLAGS` - compile again

```
make
```

If the compiler complains about the –offload-arch parameter, see the notes below.

Run this codelet on an AAC node.

```
./freduce
```

While running one of these codelets, it may be useful to watch the GPU usage. Here are two approaches.

- open another terminal and `ssh` to the AAC node you are working on, or
- use the tmux command
- run `watch -n 0.5 rocm-smi` command line from that terminal to visualize GPU activities.

Note that the basic tmux survival commands are: cntl+b " - splits the screen
cntl+b (up arrow) - move to the upper session cntl+b (down arrow) - move to lower session exit - end tmux session

Next, run the codelet on your preferred GPU device if you have allocated more than 1 GPU. For example, to execute on GPU ID #2, set the following environment variable: `export ROCR_VISIBLE_DEVICES=2` then run the code.

Profile the codelet and then compare output by setting

```
export LIBOMPTARGET_KERNEL_TRACE=1
export LIBOMPTARGET_KERNEL_TRACE=2
```

Note:

rocminfo can be used to get target architecture information.

If for any reason `--offload-arch=gfx90a` is not working as expected, try using alternative flags to enable offloading on AMD MI200 GPUs.

```
-fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx90a -fopenmp=libomp
```

Skipping the coding and just running the resulting GPU OpenMP code

Running the first OpenMP example: `Pragma_Examples/OpenMP/C/Make/saxpy`

```
module load gcc/11
salloc -N 1 --gpus=1
module list
```

Output

```
No modules loaded
```

Verify that modules get forwarded with a slurm allocation. If not, reissue the module load command when you are inside the salloc'd session.

```
cd ~/HPCTrainingExamples/Pragma_Examples/OpenMP/C/Make/saxpy
```

Building with system compiler

```
make
```

Output is

```
cc -g -O3 -fstrict-aliasing -fopenmp -foffload=-march=gfx90a -fopt-info-optimized-omp
      -c -o saxpy.o saxpy.c
cc -g -O3 -fstrict-aliasing -fopenmp -foffload=-march=gfx90a -fopt-info-optimized-omp
      -c -o codelet.o codelet.c
```

```
cc -fopenmp -foffload=-march=gfx90a -fopt-info-optimized-omp -lm saxpy.o codelet.o -o saxpy
x86_64-linux-gnu-accel-amdgcn-amdhsa-gcc-11: error: unrecognized argument in option '-march=gfx90a'
x86_64-linux-gnu-accel-amdgcn-amdhsa-gcc-11: note: valid arguments to '-march=' are:
      fiji gfx900 gfx906 gfx908; did you mean 'gfx900'?
x86_64-linux-gnu-accel-amdgcn-amdhsa-gcc-11: error: unrecognized argument in option '-march=gfx90a'
x86_64-linux-gnu-accel-amdgcn-amdhsa-gcc-11: note: valid arguments to '-march=' are:
      fiji gfx900 gfx906 gfx908; did you mean 'gfx900'?
mkoffload: fatal error: x86_64-linux-gnu-accel-amdgcn-amdhsa-gcc-11 returned 1 exit status
compilation terminated.
lto-wrapper: fatal error: /usr/lib/gcc/x86_64-linux-gnu/11//accel/amdgcn-amdhsa/mkoffload returned 1 ex

compilation terminated.
/usr/bin/ld: error: lto-wrapper failed

collect2: error: ld returned 1 exit status

make: *** [Makefile:27: saxpy] Error 1
```

This build used the system compiler /usr/bin/cc (gcc 11). It failed but note that it is generating AMD GPU code. It would have worked if we were on an MI100 GPU. The offload support is in gcc-[11|12]-offload-amdgcn. GCC documentation says that MI200 (gfx90a) will not be supported until the GCC 13 release.

Build with AMDClang compiler

```
module load amdclang
make clean
make
./saxpy
```

Confirm running on GPU with

```
export LIBOMPTARGET_KERNEL_TRACE=1
./saxpy
```

- confirms that we are running on the GPU and also gives us the register usage
- Also could use `AMD_LOG_LEVEL=[0|1|2|3|4]` or `LIBOMPTARGET_KERNEL_TRACE=2`

Advanced OpenMP

Memory Pragmas

Download the exercises and go to the directory with the memory pragma examples

```
git clone https://github.com/amd/HPCTrainingExamples.git
cd HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/memory_pragmas
```

Setup your environment

```
export CXX=amdclang++
export LIBOMPTARGET_INFO=-1
export OMP_TARGET_OFFLOAD=MANDATORY
```

You can also be more selective in the output generated by using the individual bit masks

```
export LIBOMPTARGET_INFO=$((0x01 | 0x02 | 0x04 | 0x08 | 0x10 | 0x20))
```

The first example code uses just a single pragma with a map clause at the computational loop. Examine this code and then compile and run.

```
mkdir build && cd build
cmake ..
make
./mem1
```

You should get some output like the following

```
Libomptarget device 0 info: Entering OpenMP kernel at mem1.cc:89:1 with 5 arguments:
Libomptarget device 0 info: firstprivate(n)[4] (implicit)
Libomptarget device 0 info: from(z[0:n])[80000]
Libomptarget device 0 info: firstprivate(a)[8] (implicit)
Libomptarget device 0 info: to(x[0:n])[80000]
Libomptarget device 0 info: to(y[0:n])[80000]
Libomptarget device 0 info: Creating new map entry with HstPtrBase=0x0000000001772200, ...
Libomptarget device 0 info: Creating new map entry with HstPtrBase=0x000000000174b0e0, ...
Libomptarget device 0 info: Copying data from host to device, HstPtr=0x000000000174b0e0, ...
Libomptarget device 0 info: Creating new map entry with HstPtrBase=0x000000000175e970, ...
Libomptarget device 0 info: Copying data from host to device, HstPtr=0x000000000175e970, ...
Libomptarget device 0 info: Mapping exists with HstPtrBegin=0x0000000001772200, ...
Libomptarget device 0 info: Mapping exists with HstPtrBegin=0x000000000174b0e0, ...
Libomptarget device 0 info: Mapping exists with HstPtrBegin=0x000000000175e970, ...
Libomptarget device 0 info: Mapping exists with HstPtrBegin=0x000000000175e970, ...
Libomptarget device 0 info: Mapping exists with HstPtrBegin=0x000000000174b0e0, ...
Libomptarget device 0 info: Mapping exists with HstPtrBegin=0x0000000001772200, ...
Libomptarget device 0 info: Copying data from device to host, TgtPtr=0x00007f617c420000, ...
Libomptarget device 0 info: Removing map entry with HstPtrBegin=0x000000000175e970, ...
Libomptarget device 0 info: Removing map entry with HstPtrBegin=0x000000000174b0e0, ...
Libomptarget device 0 info: Removing map entry with HstPtrBegin=0x0000000001772200, ...
-Timing in Seconds: min=0.010115, max=0.010115, avg=0.010115
-Overall time is 0.010505
Last Value: z[9999]=7.000000
```

Explore examples 2 through 5 and observe the output produced when the `LIBOMPTARGET_INFO` environment variable is set.

Mem2 pattern : Add enter/exit data alloc/delete when memory is created/freed

After new

```
mem2.cc:#pragma omp target enter data map(alloc: x[0:n], y[0:n], z[0:n])
```

Loop around computational loop and keep map on computational loop. The map to/from should check if the data exists. If not, it will allocate/delete it. Then it will do the copies to and from. This will increment the Reference Counter and decrement it at end of loop.

```
mem2.cc:#pragma omp target teams distribute parallel for simd map(to: x[0:n], y[0:n])
map(from: z[0:n])
```

Before delete

```
mem2.cc:#pragma omp target exit data map(delete: x[0:n], y[0:n], z[0:n])
```

Mem3 pattern: Replacing map to/from with updates to bypass unneeded device memory check

After new

```
mem3.cc:#pragma omp target enter data map(alloc: x[0:n], y[0:n], z[0:n])
```

Before computational loop. Data should be copied. Reference counter should not change.

```
mem3.cc:#pragma omp target update to (x[0:n], y[0:n])
mem3.cc:#pragma omp target teams distribute parallel for simd
```

After computational loop

```
mem3.cc:#pragma omp target update from (z[0:n])
```

Before delete

```
mem3.cc:#pragma omp target exit data map(delete: x[0:n], y[0:n], z[0:n])
```

Mem4 pattern: Replacing delete with release to use Reference Counting

```
mem4.cc:#pragma omp target enter data map(alloc: x[0:n], y[0:n], z[0:n])
mem4.cc:#pragma omp target exit data map(release: x[0:n], y[0:n], z[0:n])
mem4.cc:#pragma omp target teams distribute parallel for simd map(to: x[0:n], y[0:n]) map(from: z[0:
```

Mem5 pattern: Using enter data map to/from alloc/delete to reduce memory copies

```
mem5.cc:#pragma omp target enter data map(to: x[0:n], y[0:n]) map(alloc: z[0:n])
mem5.cc:#pragma omp target exit data map(from: z[0:n]) map(delete: x[0:n], y[0:n])
mem5.cc:#pragma omp target teams distribute parallel for simd map(to:x[0:n], y[0:n]) map(from: z[0:n
```

One solution that miminizes data transfer

Mem6 pattern: Using enter data alloc/delete with update clause at end

```
mem6.cc:#pragma omp target enter data map(alloc: x[0:n], y[0:n], z[0:n])
mem6.cc:#pragma omp target teams distribute parallel for simd
mem6.cc:#pragma omp target update from(z[0])
mem6.cc:#pragma omp target exit data map(delete: x[0:n], y[0:n], z[0:n])
mem6.cc:#pragma omp target teams distribute parallel for simd
```

Unified Shared Memory

Mem7 pattern: Using Unified Shared Memory to automatically move data

```
mem7.cc:#pragma omp requires unified_shared_memory
mem7.cc:#pragma omp target teams distribute parallel for simd
mem7.cc:#pragma omp target teams distribute parallel for simd
```

For this example, HSA_XNACK=1 needs to be set

```
export HSA_XNACK=1
make mem7
./mem7
```

Unified Shared Memory with backwards compatibility

Mem8 pattern: Demonstrating Unified Shared Memory with maps for backward compatibility

```
set HSA_XNACK=1 at runtime

mem8.cc:#pragma omp requires unified_shared_memory
mem8.cc:#pragma omp target enter data map(alloc: x[0:n], y[0:n], z[0:n])
mem8.cc:#pragma omp target teams distribute parallel for simd
mem8.cc:#pragma omp target update from(z[0])
mem8.cc:#pragma omp target exit data map(delete: x[0:n], y[0:n], z[0:n])
mem8.cc:#pragma omp target teams distribute parallel for simd
```

Kernel Pragmas

Download the exercises and go to the directory with the kernel pragma examples

```
git clone https://github.com/amd/HPCTrainingExamples.git
cd HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/kernel_pragmas
```

Setup your environment. You should unset the `LIBOMPTARGET_INFO` environment from previous exercise.

```
unset LIBOMPTARGET_INFO
```

```
export CXX=amdclang++
export LIBOMPTARGET_KERNEL_TRACE=1
export OMP_TARGET_OFFLOAD=MANDATORY
export HSA_XNACK=1
```

The base version 1 code is the Unified Shared memory example from the previous exercises

```
mkdir build && cd build
cmake ..
make kernel1
./kernel1
```

```
Kernel2 : add num_threads(64)
```

```
Kernel3 : add num_threads(64) thread_limit(64)
```

On your own: Uncomment line in CMakeLists.txt with -faligned-allocation -fnew-alignment=256

Another option is to add the attribute (std::align_val_t(128)) to each new line. For example:

```
double *x = new (std::align_val_t(128) ) double[n];
```

HIP and OpenMP Interoperability

This hands-on exercise uses the code in HPCTrainingExamples/HIP-OpenMP/daxpy. We have code that uses both OpenMP and HIP. These require two separate passes with compilers: one with amdclang++ and the other with hipcc. Go to the directory containing the example and set up the environment:

```
cd HPCTrainingExamples/HIP-OpenMP/CXX/daxpy
module load rocm
export CXX=amdclang++
```

View the source code file daxpy.cc and note the two #ifdef blocks.

The first one is **DEVICE_CODE** that we want to compile with hipcc.

The second is **HOST_CODE** that we will use the C++ compiler to compile.

All of the HIP calls and variables are in the first block. The second block contains the OpenMP pragmas.

While we can use hipcc to compile standard C++ code, it will not work on code with OpenMP pragmas. The call to the HIP daxpy kernel occurs near the end of the host code block. We could split out these two code blocks into separate files, but this may be more intrusive with a code design.

Now we can take a look at the Makefile we use to compile the code in the single file. In the file, we create two object files for the executable to be dependent on.

We then compile one with the CXX compiler with `-D__HOST_CODE__` defined.

The second object file is compiled using hipcc and with `-D__DEVICE_CODE__` defined.

This doesn't completely solve all the issues with separate translation units, but it does help workaround some code organization constraints.

Now on to building and running the example.

```
make
./daxpy
```

Kokkos examples

Stream Triad

Step 1: Build a separate Kokkos package

```
cd $HOME/HPCTraining/Examples
git clone https://github.com/kokkos/kokkos Kokkos_build
cd Kokkos_build
```

Build Kokkos with OpenMP backend

```
mkdir build_openmp && cd build_openmp
cmake -DCMAKE_INSTALL_PREFIX=${HOME}/Kokkos_OpenMP -DKokkos_ENABLE_SERIAL=On \
      -DKokkos_ENABLE_OPENMP=On ..

make -j 8
make install

cd ..
```

Build Kokkos with HIP backend

```
mkdir build_hip && cd build_hip
cmake -DCMAKE_INSTALL_PREFIX=${HOME}/Kokkos_HIP -DKokkos_ENABLE_SERIAL=ON \
      -DKokkos_ENABLE_HIP=ON -DKokkos_ARCH_ZEN=ON -DKokkos_ARCH_VEGA90A=ON \
      -DCMAKE_CXX_COMPILER=hipcc ..

make -j 8; make install
cd ..
```

Set Kokkos_DIR to point to external Kokkos package to use

```
export Kokkos_DIR=${HOME}/Kokkos_HIP
```

Step 2: Modify Build

Get example

```
git clone --recursive https://github.com/EssentialsOfParallelComputing/Chapter13 Chapter13
cd Chapter13/Kokkos/StreamTriad
cd Orig
```

Test serial version with

```
mkdir build && cd build; cmake ..; make; ./StreamTriad
```

If the run fails (SEGV), try reducing the size of the arrays, by reducing the value of the nsize variable in StreamTriad.cc.

Add to CMakeLists.txt

```
(add) find_package(Kokkos REQUIRED)
add_executables(StreamTriad ....)
(add) target_link_libraries(StreamTriad Kokkos::kokkos)
```

Retest with

```
cmake ..; make
```

and run ./StreamTriad again

Check Ver1 for solution. These modifications have already been made in Ver1 version.

Step 3: Add Kokkos views for memory allocation of arrays

(peek at ver4/StreamTriad.cc to see the end result)

Add include file

```
#include <Kokkos_Core.hpp>
```

Add initialize and finalize

```
Kokkos::initialize(argc, argv);  {

} Kokkos::finalize();
```

Replace static array declarations with Kokkos views

```
int nsize=80000000;
Kokkos::View<double *> a( "a", nsize);
Kokkos::View<double *> b( "b", nsize);
Kokkos::View<double *> c( "c", nsize);
```

Rebuild and run

Step 4: Add Kokkos execution pattern - parallel_for Change for loops to Kokkos parallel fors.

At start of loop

```
Kokkos::parallel_for(nsize, KOKKOS_LAMBDA (int i) {
```

At end of loop, replace closing brace with

```
});
```

Rebuild and run. Add environment variables as Kokkos message suggests:

```
 export OMP_PROC_BIND=spread
 export OMP_PLACES=threads
 export OMP_PROC_BIND=true
```

How much speedup do you observe?

Step 5: Add Kokkos timers

Add Kokkos calls

```
Kokkos::Timer timer;
timer.reset(); // for timer start
time_sum += timer.seconds();
```

Remove

```
#include <timer.h>
struct timespec tstart;
cpu_timer_start(&tstart);
time_sum += cpu_timer_stop(tstart);
```

6. Run and measure performance with OpenMP

Find out how many virtual cores are on your CPU

```
lscpu
```

First run with a single processor:

Average runtime _____

Then run the OpenMP version:

Average runtime _____

Portability Exercises

1. Rebuild Stream Triad using Kokkos build with HIP

Set Kokkos_DIR to point to external Kokkos build with HIP

```
export Kokkos_DIR=${HOME}/Kokkos_HIP/lib/cmake/Kokkos_HIP
cmake ..
make
```

2. Run and measure performance with AMD Radeon GPUs

HIP build with ROCm

Ver4 - Average runtime is _____ msecs

AMD Node Memory Model

Setup environment

```
module load amdclang rocm (or aomp)
```

Go to `ManagedMemory` Directory

```
cd ~/HPCTrainingExamples/ManagedMemory/vectorAdd
```

Replace `/opt/rocm` in Makefile (only if needed)

```
sed -i 's/\/opt\/rocm/${ROCM_PATH}/g' Makefile
```

Compile

```
make vectoradd_hip1.exe
```

Run

```
./vectoradd_hip1.exe
```

Change to a managed memory version - Replace all instances of the word `host` to `vector` - Replace all instances of the word `device` to `vector` - Delete Duplicate `vector` Declarations - Move both allocations (malloc and hipMalloc) above for loop initialization - Comment out all lines containing hipMemcpy - Add `hipDeviceSynchronize();` after the kernel launch

First Experiment: Comment out `hipMalloc` and `hipFree` lines and recompile

```
make vectoradd_hip1.exe
```

Run (Test should fail with a Memory Access Fault)

```
./vectoradd_hip1.exe
```

Re-run with `HSA_XNACK=1` (Test should pass)

```
HSA_XNACK=1 ./vectoradd_hip1.exe
```

Second Experiment: Comment out `malloc` and `free` lines instead of `hipMalloc` and `hipFree` lines and recompile

```
make vectoradd_hip1.exe
```

Run with `HSA_XNACK=0` or unset with `unset HSA_XNACK` (Test should pass)

```
HSA_XNACK=0 ./vectoradd_hip1.exe
```

OpenMP Atomics

The examples for this exercise are in ~/HPCTrainingExamples/atomics_openmp.

Set up your environment

```
module load amdclang rocm (or aomp)
```

Find the `arraysum1.c` file

```
cd ~/HPCTrainingExamples/atomics_openmp
vim arraysum1.c
```

Key lines

```
#pragma omp target teams distribute parallel for map(tofrom: a[:n]) map(to: b[:n])
for(int i = 0; i < n; i++) {
    a[i] += b[i];
}
```

Compile `arraysum1`

```
make arraysum1
```

Run

```
./arraysum1
```

Remove map clause from the pragma in `arraysum1.c`. Add the following before `main` function

```
#pragma omp requires unified_shared_memory
```

The code must now look like arraysum2.c. Compile `arraysum2`

```
make arraysum2
```

Run `arraysum2` it should fail with a Memory Access Fault

```
./arraysum2
```

Running `arraysum2` with `HSA_XNACK=1` (Test should pass)

```
export HSA_XNACK=1
./arraysum2
```

The memory in the main loop remains coarse-grained even though a map clause is not used. Once the memory is allocated, it stays that type. Note that the initialization loop is not run in parallel on the GPU. How would you fix that?

Follow the slides for arraysum3 and arraysum4.

Now on to `arraysum5.c`

Key loop shown below

```
#pragma omp target teams distribute parallel for
  for(int i = 0; i < n; i++) {
    #pragma omp atomic hint(AMD_fast_fp_atomics)
    a[i] += b[i];
  }
```

Compile `arraysum5`

```
make arraysum5
```

Run `arraysum5`

```
./arraysum5
```

This test should fail because the memory is fine-grained. Add the map clause to the pragma as implemented in arraysum5.c. Note that the ret variable also needs to be mapped.

Map clause in `arraysum6.c`

```
map(to: b[:n]) map(tofrom: ret)
```

Compile `arraysum6`

```
make arraysum6
```

Run `arraysum6` (It should pass now)

```
./arraysum6
```

Another solution to fix the problem is to change the atomic pragma to a safe version. This is shown in `arraysum10.c`

```
#pragma omp atomic hint(AMD_safe_fp_atomics)
```

The safe atomic will use a compare and swap (CAS) loop instead of an atomic add. This will work, but it will likely be slower.

The arraysum8.c through arraysum10.c repeat the fast atomics but with a scalar variable. With the ROCm 5.6.0 compiler, arraysum8 generates a LLVM compiler error: Cannot select intrinsic %llvm.amdgcn.flat.atomic.fadd. The arraysum9 fails as well.

GPU Aware MPI

Point-to-point and collective

Allocate at least two GPUs and set up your environment

```
salloc --account=project_XXX -p LocalQ -N 1 --gres=gpu:8 -ntasks=8 -time=00:15:00

module load openmpi rocm
export OMPI_CXX=hipcc
```

Find the code and compile

```
cd HPCTrainingExamples/MPI-examples
mpicxx -o ./pt2pt ./pt2pt.cpp
```

Set the environment variable and run the code

```
mpirun -n 2 ./pt2pt
```

OSU Benchmark

Get the OSU micro-benchmark tarball and extract it

```
mkdir OMB
cd OMB
wget https://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-benchmarks-7.1-1.tar.gz
tar -xvf osu-micro-benchmarks-7.1-1.tar.gz
```

Create a build directory and cd to osu-micro-benchmarks-7.1-1

```
module load gcc/12 rocm openmpi
mkdir build
cd osu-micro-benchmarks-7.1-1
```

Build and install OSU micro-benchmarks

```
./configure --prefix=`pwd`/../build/ \
                CC=`which mpicc` \
                CXX=`which mpicxx` \
                --enable-rocm \
                --with-rocm=${ROCM_PATH}
make -j12
make install
```

After make -j12 and make install, you might get some errors related to upc_memcpy. You can ignore these errors.

Check if osu microbenchmark is actually built

```
ls -l ../build/libexec/osu-micro-benchmarks/mpi/
```

if you see files collective, one-sided, pt2pt, and startup, your build is successful.

Allocate 2 GPUs, and make those visible

```
export HIP_VISIBLE_DEVICES=0,1
```

Make sure GPU-Aware communication is enabled and run the benchmark

```
salloc -p LocalQ --gpus=2 -t 00:02:00 --exclusive
mpirun -N 2 -n 2 ../build/libexec/osu-micro-benchmarks/mpi/pt2pt/osu_bw -m 10240000
```

Notes: - Try different pairs of GPUs. How does the bandwidth vary? - Try different communication options (blit kernel and SDMA) using the env variable HSA_ENABLE_SDMA. How does the bandwidth vary?

Affinity

Understanding your system

Setup environment

```
salloc -p LocalQ --cpus-per-task=16 --mem=0 --ntasks-per-node=8 --gres=gpu:8 -t 05:00
module load gcc/11 rocm
```

View System Configuration using lstopo from the hwloc package

```
lstopo-no-graphics ~/out.svg
```

Copy the out.svg file to your workstation to view

```
rsync -avz <USERNAME>@aac:~/out.svg .
```

Check CPU configuration. Note number of sockets, cores per socket, threads per core, NUMA domains, etc.

```
lscpu
```

Check NUMA configuration

```
numactl -H
```

Check how the GPU affinities are set up

```
rocm-smi --showtoponuma
```

Verifying Process and Thread Binding

Case 1. MPI + OpenMP

Here, we will experiment with a serial program (only one process) that has OpenMP support and try to pin different hardware threads to each process' OpenMP threads (`OMP_PLACES`, `OMP_NUM_THREADS`). In addition, we will ensure that all threads of a given process run on cores that are close to the primary thread's core (`OMP_PROC_BIND`).

Allocate some cores (optional)

```
salloc -p LocalQ --cpus-per-task=16 --mem=0 --ntasks-per-node=8 --gres=gpu:8 -t 05:00
```

Set up your environment

```
module load gcc/11 rocm openmpi
```

Download hello_mpi_omp.c and Makefile from

```
git clone https://code.ornl.gov/olcf/hello_mpi_omp.git
cd hello_mpi_omp/
```

Modify Makefile to change the compiler `COMP`

```
COMP  = mpicc
```

Compile

```
make
```

Run the commands:

Run 4 ranks, 2 OpenMP threads per rank

```
OMP_NUM_THREADS=2 OMP_PROC_BIND=close mpirun -np 4 -mca btl ^openib --map-by L3cache ./hello_mpi_omp
```

You should see the two threads of each rank pinned to different hardware threads (HWT), placed close and sharing L3 cache and each rank getting 2 cores from a set of 8 cores:

```
MPI 001 - OMP 000 - HWT 008 - Node de32fd6d9b3a
MPI 002 - OMP 000 - HWT 016 - Node de32fd6d9b3a
MPI 002 - OMP 001 - HWT 017 - Node de32fd6d9b3a
MPI 003 - OMP 000 - HWT 024 - Node de32fd6d9b3a
MPI 003 - OMP 001 - HWT 025 - Node de32fd6d9b3a
MPI 001 - OMP 001 - HWT 009 - Node de32fd6d9b3a
MPI 000 - OMP 000 - HWT 000 - Node de32fd6d9b3a
MPI 000 - OMP 001 - HWT 001 - Node de32fd6d9b3a
```

Run 2 ranks, 2 OpenMP threads per rank

```
OMP_NUM_THREADS=2 OMP_PROC_BIND=close mpirun -np 2 -mca btl ^openib --map-by L3cache ./hello_mpi_omp
```

You should see the following output, each thread of a process pinned to a different physical core (HWT), packed closely and sharing L3 cache:

```
MPI 000 - OMP 000 - HWT 000 - Node de32fd6d9b3a
MPI 000 - OMP 001 - HWT 001 - Node de32fd6d9b3a
MPI 001 - OMP 000 - HWT 008 - Node de32fd6d9b3a
MPI 001 - OMP 001 - HWT 009 - Node de32fd6d9b3a
```

Generate Binding Report Look for the binding report printed by mpirun using the –report-bindings option:

```
OMP_NUM_THREADS=2 OMP_PROC_BIND=close mpirun -np 4 -mca btl ^openib --map-by L3cache --report-bindings
```

This should print sets of cores for each rank where the rank and its threads may be scheduled to run.

```
[de32fd6d9b3a:33326] MCW rank 3 bound to socket 0[core 24[hwt 0]], socket 0[core 25[hwt 0]], socket 0[c
[de32fd6d9b3a:33326] MCW rank 0 bound to socket 0[core 0[hwt 0]], socket 0[core 1[hwt 0]], socket 0[cor
[de32fd6d9b3a:33326] MCW rank 1 bound to socket 0[core 8[hwt 0]], socket 0[core 9[hwt 0]], socket 0[cor
[de32fd6d9b3a:33326] MCW rank 2 bound to socket 0[core 16[hwt 0]], socket 0[core 17[hwt 0]], socket 0[c
```

Case 2. MPI + OpenMP + HIP

Here, we will explore a standalone that has support for MPI, OpenMP and HIP. Our goal is to set up affinity for CPU cores as well as GPU devices such that a given process is assigned cores and GPU devices belonging to the same NUMA domain.

Allocate 8 GPUs, 8 cores per GPU, all memory on node:

```
salloc -N 1 --cpus-per-task=16 --mem=0 --gres=gpu:8 -p LocalQ --ntasks-per-node=8
```

Download hello_jobstep.cpp:

```
git clone https://code.ornl.gov/olcf/hello_jobstep.git
cd hello_jobstep/
```

- Modify Makefile with the contents below, remember to replace spaces in the beginning of rules with tabs.

```
SOURCES = hello_jobstep.cpp
OBJECTS = $(SOURCES:.cpp=.o)
EXECUTABLE = hello_jobstep

CXX=mpic++
CXXFLAGS = -fopenmp -I${ROCM_PATH}/include -D__HIP_PLATFORM_AMD__
LDFLAGS = -L${ROCM_PATH}/lib -lhsa-runtime64 -lamdhip64

all: ${EXECUTABLE}
```

```
%.o: %.cpp
    $(CXX) $(CXXFLAGS) -o $@ -c $<

$(EXECUTABLE): $(OBJECTS)
    $(CXX) $(CXXFLAGS) $(OBJECTS) -o $@ $(LDFLAGS)

clean:
    rm -f $(EXECUTABLE)
    rm -f $(OBJECTS)
```

Compile

```
make
```

Set up helper script for setting up `ROCR_VISIBLE_DEVICES` and `GOMP_CPU_AFFINITY` for indicating GPU and CPU core affinity respectively. The script below is set up for a node with 128 physical cores (2 CPUs with 64 cores each) and 8 GPU devices in the node.

```
vim helper.sh
```

Contents for helper script, `helper.sh`:

```
#!/bin/bash

export global_rank=${OMPI_COMM_WORLD_RANK}
export local_rank=${OMPI_COMM_WORLD_LOCAL_RANK}
export ranks_per_node=${OMPI_COMM_WORLD_LOCAL_SIZE}

if [ -z "${NUM_CPUS}" ]; then
    let NUM_CPUS=128
fi

if [ -z "${RANK_STRIDE}" ]; then
    let RANK_STRIDE=$(( ${NUM_CPUS}/${ranks_per_node} ))
fi

if [ -z "${OMP_STRIDE}" ]; then
    let OMP_STRIDE=1
fi

if [ -z "${NUM_GPUS}" ]; then
    let NUM_GPUS=8
fi

if [ -z "${GPU_START}" ]; then
    let GPU_START=0
fi

if [ -z "${GPU_STRIDE}" ]; then
    let GPU_STRIDE=1
fi

cpu_list=($(seq 0 127))
let cpus_per_gpu=${NUM_CPUS}/${NUM_GPUS}
let cpu_start_index=$(( ($RANK_STRIDE*${local_rank})+${GPU_START}*$cpus_per_gpu ))
let cpu_start=${cpu_list[$cpu_start_index]}
```

```
let cpu_stop=$(($cpu_start+$OMP_NUM_THREADS*$OMP_STRIDE-1))

gpu_list=(0 1 2 3 4 5 6 7)
let ranks_per_gpu=$((${ranks_per_node}+${NUM_GPUS}-1)/${NUM_GPUS}))
let my_gpu_index=$(($local_rank*$GPU_STRIDE/$ranks_per_gpu))+${GPU_START}
let my_gpu=${gpu_list[${my_gpu_index}]}

export GOMP_CPU_AFFINITY=$cpu_start-$cpu_stop:$OMP_STRIDE
export ROCR_VISIBLE_DEVICES=$my_gpu

"$@"
```

Make `helper.sh` an executable

```
chmod +x helper.sh
```

Run Commands

Run 8 ranks, 2 threads per rank, 1 GPU per rank, 2 cores from each set of 16 cores for each rank The `mpirun` option `--bind-to none` is required for the affinities we set using the helper script to take effect.

```
OMP_NUM_THREADS=2 mpirun -np 8 -mca btl ^openib --bind-to none ./helper.sh ./hello_jobstep
```

Output will be similar to this:

```
MPI 000 - OMP 000 - HWT 000 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 000 - OMP 001 - HWT 001 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 001 - OMP 000 - HWT 016 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 001 - OMP 001 - HWT 017 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 002 - OMP 000 - HWT 032 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID 03
MPI 002 - OMP 001 - HWT 033 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID 03
MPI 003 - OMP 000 - HWT 048 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID 26
MPI 003 - OMP 001 - HWT 049 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID 26
MPI 004 - OMP 000 - HWT 064 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID e3
MPI 004 - OMP 001 - HWT 065 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID e3
MPI 005 - OMP 000 - HWT 080 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID c3
MPI 005 - OMP 001 - HWT 081 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID c3
MPI 006 - OMP 000 - HWT 096 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID 83
MPI 006 - OMP 001 - HWT 097 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID 83
MPI 007 - OMP 000 - HWT 112 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID a3
MPI 007 - OMP 001 - HWT 113 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID a3
```

Run 2 ranks per GPU packed closely (ranks 0 and 1 run on GPU 0) and bind 2 cores from each set of 8 cores for each rank: Re-allocate a node for 16 task, 8 CPUs per task and 8 GPUs:

```
salloc -p LocalQ --cpus-per-task=8 --mem=0 --ntasks-per-node=16 --gres=gpu:8
```

Setup environment

```
module load gcc/11 rocm
```

Run the command

```
cd hello_jobstep/
OMP_NUM_THREADS=2 mpirun -np 16 -mca btl ^openib --bind-to none ./helper.sh ./hello_jobstep
```

The output will look similar to this where we see that ranks 0 and 1 got GPU ID 0 and cores 0-1 and 8-9 respectively:

```
MPI 009 - OMP 000 - HWT 072 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID e3
MPI 009 - OMP 001 - HWT 073 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID e3
MPI 013 - OMP 000 - HWT 104 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID 83
MPI 013 - OMP 001 - HWT 105 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID 83
MPI 005 - OMP 000 - HWT 040 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID 03
MPI 005 - OMP 001 - HWT 041 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID 03
MPI 008 - OMP 000 - HWT 064 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID e3
MPI 008 - OMP 001 - HWT 065 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID e3
MPI 000 - OMP 000 - HWT 000 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 000 - OMP 001 - HWT 001 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 001 - OMP 000 - HWT 008 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 001 - OMP 001 - HWT 009 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 002 - OMP 000 - HWT 016 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 002 - OMP 001 - HWT 017 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 003 - OMP 000 - HWT 024 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 003 - OMP 001 - HWT 025 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 004 - OMP 000 - HWT 032 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID 03
MPI 004 - OMP 001 - HWT 033 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID 03
MPI 006 - OMP 000 - HWT 048 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID 26
MPI 006 - OMP 001 - HWT 049 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID 26
MPI 007 - OMP 000 - HWT 056 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID 26
MPI 007 - OMP 001 - HWT 057 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID 26
MPI 014 - OMP 000 - HWT 112 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID a3
MPI 014 - OMP 001 - HWT 113 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID a3
MPI 011 - OMP 000 - HWT 088 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID c3
MPI 011 - OMP 001 - HWT 089 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID c3
MPI 012 - OMP 000 - HWT 096 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID 83
MPI 012 - OMP 001 - HWT 097 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID 83
MPI 015 - OMP 000 - HWT 120 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID a3
MPI 015 - OMP 001 - HWT 121 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID a3
MPI 010 - OMP 000 - HWT 080 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID c3
MPI 010 - OMP 001 - HWT 081 - Node de32fd6d9b3a - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID c3
```

ROCgdb

Saxpy debugging

Go to the saxpy kernel

```
cd HPCTrainingExamples/HIP/saxpy
```

Get an allocation of a GPU and load software modules

```
salloc
module load rocm
```

Comment out the `hipmalloc` lines in main()

Compile with

```
mkdir build && cd build
cmake ..
make VERBOSE=1
```

Run the code

```
./saxpy
```

You should get an error output such as

```
Memory access fault by GPU node-2 (Agent handle: 0x2284d90) on address (nil). Reason: Unknown.
Aborted (core dumped)
```

Now run the code with the rocgdb debugger

```
rocgdb -tui saxpy
```

In the debugger, type `run`

```
Thread 3 "saxpy" received signal SIGSEGV, Segmentation fault.
0x00007ffff7ec1094 in saxpy(int, float const*, int, float*, int)
```

Compile with `-ggdb -O0` added to the compile flags

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
make VERBOSE=1
```

Rerun the code with the debugger. You should now get the line number where the error occurs.

```
Thread 3 "saxpy" received signal SIGSEGV, Segmentation fault.
[Switching to thread 3, lane 0 (AMDGPU Lane 1:2:1:1/0 (0,0,0)[0,0,0])]
0x00007ffff7ec1094 in saxpy() at saxpy.cpp:10
10    y[i] += a*x[i];
```

The error report is at a thread on the GPU

To get the CPU stack trace

```
info threads
thread 1
where
```

Put a breakpoint right after the `hipmalloc` lines.

```
b 43
```

run to breakpoint with `run` insert breakpoint at start of GPU kernel with `b saxpy` continue with `continue`

Try another example in HIP examples

Rocprof

Setup environment

```
salloc --cpus-per-task=8 --mem=0 --ntasks-per-node=4
module load rocm
```

Download examples repo and navigate to the `HIPIFY` exercises

```
cd ~/HPCTrainingExamples/HIPIFY/mini-nbody/hip/
```

Update the bash scripts with `$ROCM_PATH`

```
sed -i 's/\/opt\/rocm/${ROCM_PATH}/g' *.sh
```

Compile all

```
./HIP-nbody-orig.sh
```

or compile and run one case

```
hipcc -I../ -DSHMOO nbody-orig.cpp -o nbody-orig
```

Run `rocprof` on nbody-orig to obtain hotspots list

```
rocprof --stats nbody-orig 65536
```

Check Results

```
cat results.csv
```

Check the statistics result file, one line per kernel, sorted in descending order of durations

```
cat results.stats.csv
```

Using `--basenames on` will show only kernel names without their parameters.

```
rocprof --stats --basenames on nbody-orig 65536
```

Check the statistics result file, one line per kernel, sorted in descending order of durations

```
cat results.stats.csv
```

Trace HIP calls with `--hip-trace`

```
rocprof --stats --hip-trace nbody-orig 65536
```

Check the new file `results.hip_stats.csv`

```
cat results.hip_stats.csv
```

Profile also the HSA API with the `--hsa-trace`

```
rocprof --stats --hip-trace --hsa-trace nbody-orig 65536
```

Check the new file `results.hsa_stats.csv`

```
cat results.hsa_stats.csv
```

On your laptop, download `results.json`

```
scp scp://USER@aac1.amd.com:<PORT>/~/HPCTrainingExamples/HIPIFY/mini-nbody/hip/results.json ./
```

Open a browser and go to https://ui.perfetto.dev/. Click on `Open trace file` in the top left corner. Navigate to the `results.json` you just downloaded. Use WASD to navigate the GUI

Read about hardware counters available for the GPU on this system (look for gfx90a section)

```
less $ROCM_PATH/lib/rocprofiler/gfx_metrics.xml
```

Create a `rocprof_counters.txt` file with the counters you would like to collect

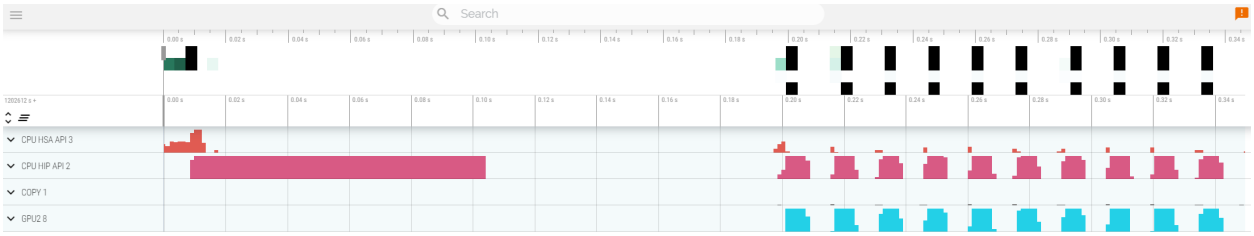

Figure 1: image

`vi rocprof_counters.txt`

Content for `rocprof_counters.txt`:

```
pmc : Wavefronts VALUInsts
pmc : SALUInsts SFetchInsts GDSInsts
pmc : MemUnitBusy ALUStalledByLDS
```

Execute with the counters we just added:

```
rocprof --timestamp on -i rocprof_counters.txt  nbody-orig 65536
```

You'll notice that `rocprof` runs 3 passes, one for each set of counters we have in that file.

Contents of `rocprof_counters.csv`

```
cat rocprof_counters.csv
```

```
exit
```

Omnitrace

Setup environment

```
module load rocm openmpi
```

Basic Omnitrace setup

List the various options and environment settings available for the `omnitrace` category:

```
omnitrace-avail --categories omnitrace
```

To add brief descriptions, use `-bd` option

```
omnitrace-avail -bd --categories omnitrace
```

Create an Omnitrace configuration file with description per option.

```
omnitrace-avail -G ~/omnitrace_all.cfg --all
```

To create a configuration file without descriptions, drop the `--all` option:

```
omnitrace-avail -G ~/omnitrace.cfg
```

Declare to use this configuration file:

```
export OMNITRACE_CONFIG_FILE=~/omnitrace.cfg
```

Setup Jacobi Example

Go to the jacobi code in the examples repo:

```
cd ~/HPCTrainingExamples/HIP/jacobi
```

Compile

```
make
```

Execute the binary to make sure it runs successfully: - Note: To get rid of `Read -1, expected 4136, errno = 1` add `--mca pml ucx --mca pml_ucx_tls ib,sm,tcp,self,cuda,rocm` to the `mpirun` command line

```
mpirun -np 1 ./Jacobi_hip -g 1 1
```

Dynamic Instrumentation

(WARNING) - this may in the current container Run the code with omnitrace to get runtime instrumentation. Time it to see overhead of dyninst loading all libraries in the beginning.

```
mpirun -np 1 omnitrace-instrument -- ./Jacobi_hip -g 1 1
```

Check available functions to instrument using the `--print-available functions` option. Note, the `--simulate` option will not execute the binary.

```
mpirun -np 1 omnitrace-instrument -v 1 --simulate --print-available functions -- ./Jacobi_hip -g 1 1
```

Binary Rewrite

Create instrumented binary

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

Executing the new instrumented binary, time it to see lower overhead:

```
mpirun -np 1 omnitrace-run -- ./Jacobi_hip.inst -g 1 1
```

See the list of the instrumented GPU calls:

```
cat omnitrace-Jacobi_hip.inst-output/<TIMESTAMP>/roctracer-0.txt
```

Visualization

Copy the `perfetto-trace-0.proto` to your laptop, open the web page https://ui.perfetto.dev/

```
scp scp://USER@aac1.amd.com:<PORT>/~/HPCTrainingExamples/HIP/jacobi/omnitrace-Jacobi_hip.inst-output/TI
```

Click `Open trace file` and select the `.proto` file

Hardware Counters

See a list of all the counters

```
omnitrace-avail --all
```

Declare in your configuration file:

```
OMNITRACE_ROCM_EVENTS = GPUBusy,Wavefronts,MemUnitBusy
```

Check again:

```
grep OMNITRACE_ROCM_EVENTS $OMNITRACE_CONFIG_FILE
```

Run the instrumented binary, and visualize the Perfetto trace produced to see the hardware counters:

```
mpirun -np 1 omnitrace-run -- ./Jacobi_hip.inst -g 1 1
```

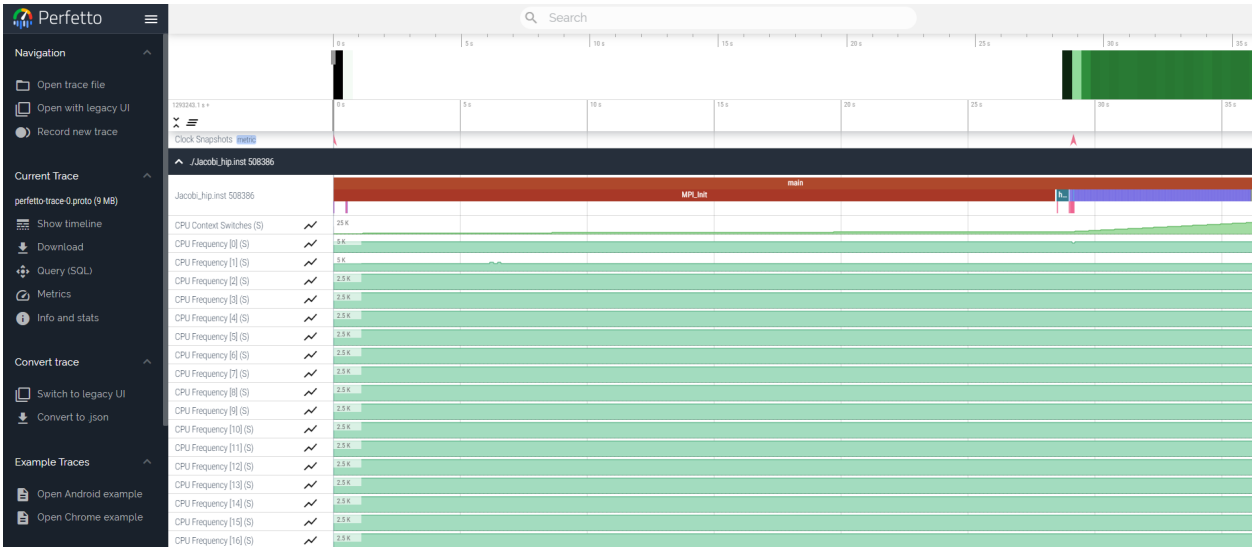

Figure 2: image

Profiling Multiple Ranks

Run the instrumented binary with multiple ranks. You'll find multiple `perfetto-trace-*.proto` files, one for each rank.

```
mpirun -np 2 omnitrace-run -- ./Jacobi_hip.inst -g 2 1
```

You can visualize them separately in Perfetto, or combine them using cat and visualize them in the same Perfetto window.

```
cat perfetto-trace-0.proto perfetto-trace-1.proto > allprocesses.proto
```

Sampling

Set the following in your configuration file:

```
OMNITRACE_USE_SAMPLING = true
OMNITRACE_SAMPLING_FREQ = 100
```

Execute the instrumented binary and visualize the perfetto trace.

```
mpirun -np 1 omnitrace-run -- ./Jacobi_hip.inst -g 1 1
```

Scroll down to the very bottom to see the sampling output. Those traces will be annotated with a (S) as well.

Kernel Timings

Open the `wall_clock-0.txt` file:

```
cat omnitrace-Jacobi_hip.inst-output/<TIMESTAMP>/wall_clock-0.txt
```

In order to see the kernel durations aggregated in your configuration file, make sure to set in your config file or in the environment:

```
OMNITRACE_USE_TIMEMORY = true
OMNITRACE_FLAT_PROFILE = true
```

Execute the code and check the `wall_clock-0.txt` file again.

```
OMNITRACE_USE_TIMEMORY=true OMNITRACE_FLAT_PROFILE=true mpirun -np 1 omnitrace-run -- ./Jacobi_hip.inst
```

Omniperf

vcopy

Setup environment

```
module load rocm openmpi
```

Get sample code

```
wget https://github.com/AMDResearch/omniperf/raw/main/sample/vcopy.cpp
```

Compile

```
hipcc -o vcopy vcopy.cpp
```

NOTE: THIS FAILS BECAUSE ROOFLINE CODE IS MISSING

Profile with omniperf

```
omniperf profile -n vcopy_all -- ./vcopy 1048576 256
```

A new directory will be created named `workloads/vcopy_all`.

Analyze the collected profile using the built-in CLI

```
omniperf analyze -p workloads/vcopy_all/mi200/ &> vcopy_analyze.txt
```

View `vcopy_analyze.txt`

```
less vcopy_analyze.txt
```

We can select specific IP Blocks to analyze

```
omniperf analyze -p workloads/vcopy_all/mi200/ -b 7.1.2
```

If you've installed Omniperf on your laptop (No ROCm Required), you can download `workloads/vcopy_all/mi200/` to your laptop and run omniperf with `--gui` option

```
omniperf analyze -p workloads/vcopy_all/mi200/ --gui
```

Open the following in your browser

```
http://172.21.7.117:8050/
```

dgemm

```
cd HPCTrainingExamples/dgemm

mkdir build && cd build
cmake ..
make
```

Profile and analyze code

```
omniperf profile -n dgemm -- ./dgemm
omniperf analyze -p workloads/dgemm/mi200 >& omniperf.out
less omniperf.out
```

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,

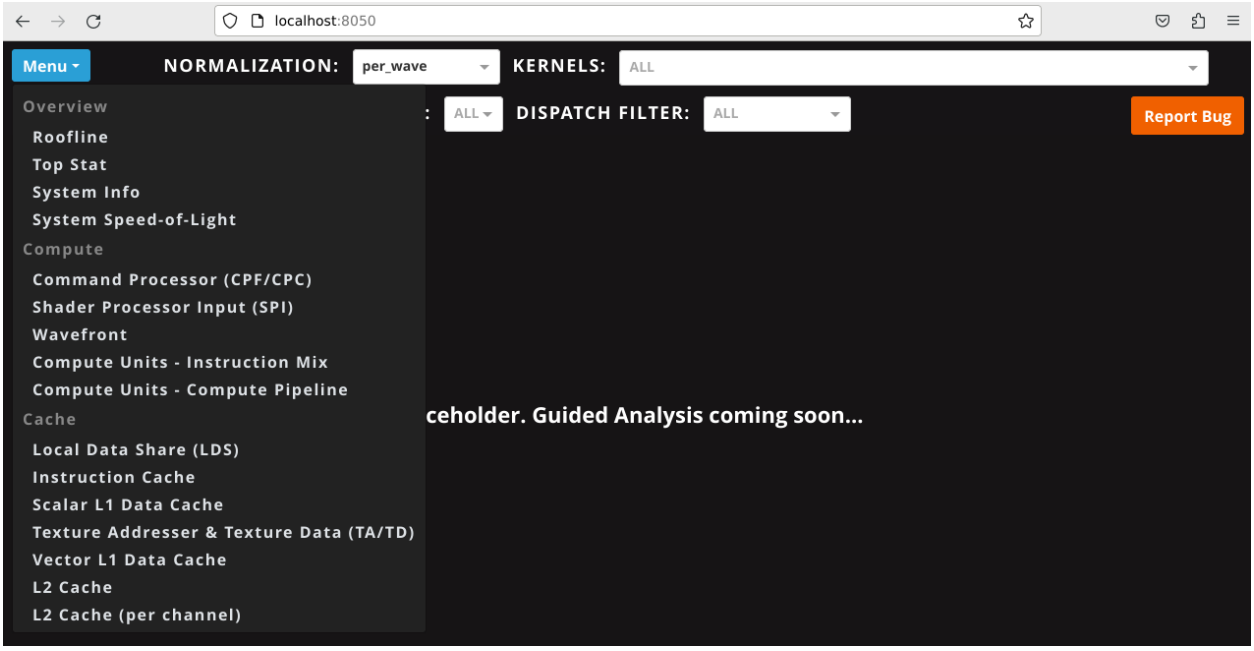

Figure 3: image

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.