AMD HPC Training Examples Document

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

AMD HPC Solutions & Performance Analysis Team

Last Review - September 9th 2024

Contents

Summary

Welcome to AMD's HPC Training Examples Document

Here you will find a variety of examples to showcase the capabilities of AMD's GPU software stack. Please be aware that this document is continuously updated to keep up with the most recent releases of the AMD software.

Examples Repository Structure

Please refer to this table of contents to locate the exercises you are interested in sorted by topic.

1. **HIP**
   1. ***Basic Examples***
      1. `Stream_Overlap` : this example shows how to share the workload of a GPU offload compation using several overlapping streams. The result is an additional gain in terms of time of execution due to the additional parallelism provided by the overlapping streams. `README` .
      2. `dgemm` : a (d)GEMM application created as an exercise to showcase simple matrix-matrix multiplications on AMD GPUs. `README` .
      3. `basic_examples` : a collection of introductory exercises such as device to host data transfer and basic GPU kernel implementation. `README` .
      4. `hip_stream` : modification of the STREAM benchmark for HIP. `README` .
      5. `jacobi` : distributed Jacobi solver, using GPUs to perform the computation and MPI for halo exchanges. `README` .
      6. `matrix_addition` : example of a HIP kernel performing a matrix addition.
      7. `saxpy` : example of a HIP kernel performing a saxpy operation. `README` .
      8. `stencil_examples` : examples stencils operation with a HIP kernel, including the use of timers and asyncronous copies.
      9. `vectorAdd` : example of a HIP kernel to perform a vector add. `README` .
      10. `vector_addition_examples` : another example of a HIP kernel to perform vector addition, including different versions such as one using shared memory, one with timers, and a CUDA one to try `HIPIFY` and `hipifly` tools on. The examples in this directory are not part of the HIP test suite.
   2. ***CUDA to HIP Porting***
      1. `HIPIFY` : example to show how to port CUDA code to HIP with HIPIFY tools. `README` .
      2. `hipifly` : example to show how to port CUDA code to HIP with hipifly tools. `README` .
   3. `HIP-Optimizations` : a daxpy HIP kernel is used to show how an initial version can be optimized to improve performance. `README` .
   4. `HIPFort` : a gemm example in Fortran using hipfort.
   5. `HIPStdPar` : several examples showing C++ Std Parallelism on AMD GPUs. `README` .
   6. `HIP-OpenMP` : example on HIP/OpenMP interoperability.
2. **MPI-examples**
   1. ***Benchmarks***: GPU aware benchmarks (`collective.cpp` and `pt2pt.cpp`) to assess the performance of the communication libraries. `README` . **NOTE**: for more detailed instructions on how to run GPU aware MPI examples, see GPU_aware_MPI.
   2. ***GhostExchange***: slimmed down example of an actual physics application where the solution is initialized on a square domain discretized with a Cartesian grid, and then advanced in parallel using MPI communications. **NOTE**: detailed `README` files are provided here for the different versions of the `GhostExchange_ArrayAssign` code, that showcase how to use `Omnitrace` to profile this application.
3. **ManagedMemory**: programming model exercises, topics covered are APU programming model, OpenMP, performance protability frameworks (Kokkos and RAJA) and discrete GPU programming

model. `README` .

4. **MLExamples**: a variation of PyTorch's MNIST example code and a smoke test for mpi4py using cupy. Instructions on how to run and test other ML frameworks are in the `README` .
5. **Occupancy**: example on modifying thread occupancy, using several variants of a matrix vector multiplication leveraging shared memory and launch bounds.
6. **OmniperfExamples**: several examples showing how to leverage Omniperf to perform kernel level optimization. **NOTE**: detailed READMEs are provided on each subdirectory. `README` . `Video of Presentation` .
7. **Omnitrace**
    1. ***Omnitrace on Jacobi***: Omnitrace used on the Jacobi solver example. `README` .
    2. ***Omnitrace by Example***: Omnitrace used on several versions of the Ghost Exchange example. `READMEs` available for each of the different versions of the example code. `Video of Presentation` .
8. **Pragma_Examples**: OpenMP (in Fortran, C, and C++) and OpenACC examples. `README` .
9. **Speedup_Examples**: examples to show the speedup obtained going from a CPU to a GPU implementation. `README` .
10. **atomics_openmp**: examples on atomic operations using OpenMP.
11. **Kokkos**: runs the Stream Triad example with a Kokkos implementation. `README` .
12. **Rocgdb**: debugs the `HPCTrainingExamples/HIP/saxpy` example with Rocgdb. `README` . `Video of Presentation` .
13. **Rocprof**: uses Rocprof to profile `HPCTrainingExamples/HIPIFY/mini-nbody/hip/` . `README` .
14. **GPU_aware_MPI**: OSU Mini Benchmarks with GPU aware MPI. `README` . `Video of Presentation` .
15. **rocm_blog_codes**: this directory contains accompany source code examples for select HPC ROCm blogs found at https://rocm.blogs.amd.com. `README` .
16. **login_info**
    1. ***AAC***: instructions on how to log in to the AMD Accelerator Cloud (AAC) resource. `README` .

Run the Tests

Most of the exercises in this repo can be run as a test suite by doing:

```
git clone https://github.com/amd/HPCTrainingExamples && \
cd HPCTrainingExamples && \
cd tests && \
./runTests.sh
```

You can also run a subset of the whole test suite by specifying the subset you are interested in as an input to the `runTests.sh` script. For instance: `./runTests.sh --pytorch` . To see a full list of the possible subsets that can be run: `./runTests.sh --help` .

NOTE: tests can also be run manually from their respective directories, provided the necessary modules have been loaded and they have been compiled appropriately.

Feedback

We welcome your feedback and contributions, feel free to use this repo to bring up any issues or submit pull requests. The software made available here is released under the MIT license, more details can be found in `LICENSE.md` .

Porting Applications to HIP

Hipify Examples

NOTE: these exercises have been tested on MI210 and MI300A accelerators using a container environment. To see details on the container environment (such as operating system and modules available) please see `README.md` on this repo.

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 -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/HIPIFY/
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 CUDAC 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 `#ifndef __CUDACC__` around a block of code there. We also need the #ifndef 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 [...]`

```
        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*)}

\begin{verbatim}
>>> 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
```

CMake option

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
```

HIPifly Example: Vector Addition

Original author was Trey White, at the time with HPE and now with ORNL.

The HIPifly method for converting CUDA code to HIP, is straight-forward and works with minimal modifications to the source code. This example applies the HIPifly method to a simple vector addition problem offloaded to the GPU using CUDA.

All CUDA functions are defined in the `src/gpu_functions.cu` file. By including the `cuda_to_hip.h` file when using HIP, all the CUDA functions will be automatically replaced with the analogous HIP function during compile time.

By default, the program is compiled for NVIDIA GPUs using `nvcc` . To compile for CUDA just run `make` .

To compile for AMD GPUs using `hipcc` run `make DFLAGS=-DENABLE_HIP` . Note that the Makefile applies different GPU compilation flags when compiling for CUDA or for HIP.

The paths to the CUDA or the ROCm software stack as `CUDA_PATH` or `ROCM_PATH` are needed to compile.

After compiling run the program: `./vector_add`

Contributions from Suyash Tandon, Noel Chalmers, Nick Curtis, Justin Chang, and Gina Sitaraman.

Description document for the GPU-based Jacobi solver

Contents:

Application overview

This is a distributed Jacobi solver, using GPUs to perform the computation and MPI for halo exchanges. It uses a 2D domain decomposition scheme to allow for a better computation-to-communication ratio than just 1D domain decomposition.

The flow of the application is as follows: * The MPI environment is initialized * The command-line arguments are parsed, and a MPI grid and mesh are created * Resources (including host and device memory blocks, streams etc.) are initialized * The Jacobi loop is executed; in every iteration, the local block is updated and then the halo values are exchanged; the algorithm converges when the global residue for an iteration falls below a threshold, but it is also limited by a maximum number of iterations (irrespective if convergence has been achieved or not) * Run measurements are displayed and resources are disposed

The application uses the following command-line arguments: * `-g x y` - mandatory argument for the process topology, `x` denotes the number of processes on the X direction (i.e. per row) and `y` denotes the number of processes on the Y direction (i.e. per column); the topology size must always match the number of available processes (i.e. the number of launched MPI processes must be equal to x * y) * `-m dx dy` - optional argument indicating the size of the local (per-process) domain size; if it is omitted, the size will default to `DEFAULT_DOMAIN_SIZE` as defined in `defines.h` * `-h | --help` - optional argument for printing help information; this overrides all other arguments

Prerequisites

To build and run the jacobi application on A+A hardware, the following dependencies must be installed first:

- an MPI implementation (openMPI, MPICH, etc.)
- ROCm 2.1 or later.

Build Instructions

A `Makefile` is included along with the source files that configures and builds multiple objects and then stitches them together to build the binary for the application `Jacobi_hip`. To build, simply run:

```
make
```

An alternative cmake build system is also include

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

Run instructions

To run use:

```
mpirun -np 2 ./Jacobi_hip -g 2 1
```

Major revisions by Suyash Tandon.

(d)GEMM Application

About

A simple (d)GEMM application created as an exercise to showcase simple matrix-matrix multiplications on AMD GPUs. The simpler interface makes it easier to be used in training modules as opposed to other optimized and complicated gemm libraries.

Requirements

- cmake > 2.8
- ROCm > 3.9

Build

Follow the instructions below to configure and build the `dgemm` binary using the commands:

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

To install at a specific location, provide `-DCMAKE_INSTALL_PREFIX=<path-to-install>` , for example to install in `$HOME` dir:

```
cmake -DCMAKE_INSTALL_PREFIX=$HOME ..
```

Usage

Sample usage is shown below:

```
dgemm \
   -m 8192 \
   -n 8192 \
   -k 8192 \
   -i 10 \
   -r 10 \
   -d DEVICE_LIST \
   -o $(hostname)_dgemm.json
```

where, - `m` is row count of matrix A - `n` is column count of matrix A - `k` is column count of matrix B - `i` is iteration count to perform - `r` is number of repetitions of dgemm to perform when evaluating flops - `d` is a comma separated list of devices to use, indexed at zero (e.g. 0,1,…) - `o` to give filename to write all data. If not `.csv` , will write in `.json` (optional)

Output

GEMM operations on each GPU are run asynchronously, and the data printed to `stdout` aims only to show the progress for each GPU:

```
./dgemm -m 8192 -n 8192 -k 8192 -i 3 -r 10 -d 0,1,2,3 -o dgemm.csv
2     1   27.56
0     1   27.63
3     1   27.56
1     1   27.63
2     2   27.56
0     2   27.63
3     2   27.56
1     2   27.63
2     3   27.56
0     3   27.63
3     3   27.56
1     3   27.63
```

Summary of the results are dumped to `stdout` at the end of the run:

```
  DEV |         MIN |         MAX |     AVERAGE |     STD Dev
-----------------------------------------------------------
    0 |     27.6259 |     27.6259 |     27.6259 |           0
    1 |     27.6259 |     27.6259 |     27.6259 |           0
    2 |     27.5567 |     27.5567 |     27.5567 |           0
    3 |     27.5567 |     27.5567 |     27.5567 |           0
```

If an output file is specified, the complete data for each iteration including timestamps are printed to file in either `json` or `csv` format, depending on the specific output file extension via `-o` .

json

Default output format is json unless `csv` extension is specified in output filename. The results per iteration are recorded along with the local timestamp when the flop-rate was estimated. Output format is

```
{
  "flop_rates": {
     "<device_id_1>": [<flop_rates>, ...],
     "<device_id_2>": [<flop_rates>, ...],
     ...
  },
  "times": {
     "<device_id_1>": ["<timestamp>", ...],
     "<device_id_2>": ["<timestamp>", ...],
     ...
  },
  "args": {
     <input args as key-value pairs>
  }
}
```

where `<timestamp>` format is `YYYY-mm-dd HH:MM:SS.ZZZ` , and `<device_id_N>` are specified input device ids (0, 1, 2, . . .).

csv

If the output file name has `csv` as extension, data is written in comma-saparated format. There is a single header line, followed by data, where the header for this case is as follows

```
t_<N>,flops_<N>,[...]
```

where `N` is an integer representing the device id. Device ids are not guaranteed to be ordered.

Example `csv` output:

```
$ cat dgemm.csv
t_3,flops_3,t_2,flops_2,t_0,flops_0,t_1,flops_1
"2022-12-08 15:17:15.232",27.556682,"2022-12-08 15:17:15.087",27.556682,"2022-12-08 15:17:15.228",27.625920,"2022-12-08 15:17:15.437",27.625920
"2022-12-08 15:17:15.631",27.556682,"2022-12-08 15:17:15.486",27.556682,"2022-12-08 15:17:15.627",27.625920,"2022-12-08 15:17:15.836",27.625920
"2022-12-08 15:17:16.030",27.556682,"2022-12-08 15:17:15.885",27.556682,"2022-12-08 15:17:16.025",27.625920,"2022-12-08 15:17:16.234",27.625920
```

Optimizing DAXPY HIP

In this exercise, we will progressively make changes to optimize the DAXPY kernel on GPU. Any AMD GPU can be used to test this.

DAXPY Problem:

```
Z = aX + Y
```

where `a` is a scalar, `X` , `Y` and `Z` are arrays of double precision values.

In DAXPY, we load 2 FP64 values (8 bytes each) and store 1 FP64 value (8 bytes). We can ignore the scalar load because it is constant. We have 1 multiplication and 1 addition operation for the 24 bytes moved per element of the array. This yields a low arithmetic intensity of 2/24. So, this kernel is not compute bound, so we will only measure the achieved memory bandwith instead of FLOPS.

Inputs

- `N` , the number of elements in `X` , `Y` and `Z` . `N` may be reset to suit some optimizations. Choose a sufficiently large array size to see some differences in performance.

Build Code

```
make
```

Run exercises

```
./daxpy_1 10000000
./daxpy_2 10000000
./daxpy_3 10000000
./daxpy_4 10000000
./daxpy_5 10000000
```

Things to ponder about

daxpy_1

This shows a naive implementation of the daxpy problem on the GPU where only 1 wavefront is launched and the 64 work-items in that wavefront loop over the entire array and process 64 elements at a time. We expect this kernel to perform very poorly because it simply utilizes a part of 1 CU, and leaves the rest of the GPU unutilized.

daxpy_2

This time, we are launching multiple wavefronts, each work-item now processing only 1 element of each array. This launches `N/64` wavefronts, enough to be scheduled on all CUs. We see a big improvement in performance here.

daxpy_3

In this experiment, we check to see if launching larger workgroups can help lower our kernel launch overhead because we launch fewer workgroups if each workgroup has 256 work-items. In this case too, an improvement in measured bandwidth achieved is seen.

daxpy_4

If we ensured that the array has a multiple of `BLOCK_SIZE` elements so that all work-items in each workgroup have an element to process, then we can avoid the conditional statement in the kernel. This could reduce some instructions in the kernel.. Do we see any improvement? In this trivial case, this does not matter. Nevertheless, it is something we could keep in mind.

Question: What happens if `BLOCK_SIZE` is `1024` ? Why?

daxpy_5

In this experiment, we will use double2 type in the kernel to see if the compiler can generate `global_load_dwordx4` instructions instead of `global_load_dwordx2` instructions. So, with same number of load and store instructions, we are able to read/write two elements from each array in each thread. This should help amortize on the cost of index calculations.

To show this difference, we need to generate the assembly for these two kernels. To generate the assembly code for these kernels, ensure that the `-g --save-temps` flags are passed to `hipcc` . Then you can find the assembly code in `daxpy_*-host-x86_64-unknown-linux-gnu.s` files. Examining `daxpy_3` and `daxpy_5` , we see the two cases (edited here for clarity):

`daxpy_3` :

```
    global_load_dwordx2 v[2:3], v[2:3], off
    v_mov_b32_e32 v6, s5
    global_load_dwordx2 v[4:5], v[4:5], off
    v_add_co_u32_e32 v0, vcc, s4, v0
    v_addc_co_u32_e32 v1, vcc, v6, v1, vcc
    s_waitcnt vmcnt(0)
    v_fmac_f64_e32 v[4:5], s[6:7], v[2:3]
    global_store_dwordx2 v[0:1], v[4:5], off
```

`daxpy_5` :

```
    global_load_dwordx4 v[0:3], v[0:1], off
    v_mov_b32_e32 v10, s5
    global_load_dwordx4 v[4:7], v[4:5], off
    s_waitcnt vmcnt(0)
    v_fmac_f64_e32 v[4:5], s[6:7], v[0:1]
    v_add_co_u32_e32 v0, vcc, s4, v8
    v_fmac_f64_e32 v[6:7], s[6:7], v[2:3]
    v_addc_co_u32_e32 v1, vcc, v10, v9, vcc
    global_store_dwordx4 v[0:1], v[4:7], off
```

We observe that, in the `daxpy_5` case, there are two `v_fmac_f64_e32` instructions as expected, one for each element being processed.

Notes

- Before timing kernels, it is best to launch the kernel at least once as warmup so that those initial GPU launch latencies do not affect your timing measurements.
- The timing loop is typically several hundred iterations.

ROCgdb

NOTE: these exercises have been tested on MI210 and MI300A accelerators using a container environment. To see details on the container environment (such as operating system and modules available) please see `README.md` on this repo.

We show a simple example on how to use the main features of the ROCm debugger `rocgdb` .

Saxpy Debugging

Let us consider the `saxpy` kernel in the HIP examples:

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

Get an allocation of a GPU and load software modules:

```
salloc -N 1 --gpus=1
module load rocm
```

You can see some information on the GPU you will be running on by doing:

```
rocm-smi
```

To introduce an error in your program, comment out the `hipMalloc` calls at line 71 and 72, then compile with:

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

Running the program, you will see the expected runtime error:

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

To run the code with the `rocgdb` debugger, do:

```
rocgdb saxpy
```

Note that there are also two options for graphical user interfaces that can be turned on by doing:

```
rocgdb -tui saxpy
cgdb -d rocgdb saxpy
```

For the latter command above, you need to have `cgdb` installed on your system.

In the debugger, type `run` (or just `r`) and you will get an error similar to this one:

```
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:57
57    y[i] += a*x[i];
```

Note that the cmake build type is set to `RelWithDebInfo` (see line 8 in CMakeLists.txt). With this build type, the debugger will be aware of the debug symbols. If that was not the case (for instance if compiling in `Release` mode), running the code with the debugger you would get an error message ***without*** line info, and also a warning like this one:

```
Reading symbols from saxpy...
(No debugging symbols found in saxpy)
```

The error report is at a thread on the GPU. We can display information on the threads by typing `info threads` (or `i th`). It is also possible to move to a specific thread with `thread <ID>` (or `t <ID>`) and see the location of this thread with `where` . For instance, if we are interested in the thread with ID 1:

```
i th
th 1
where
```

You can add breakpoints with `break` (or `b`) followed by the line number. For instance to put a breakpoint right after the `hipMalloc` lines do `b 72` .

When possible, it is also advised to compile without optimization flags (so using `-O0`) to avoid seeing breakpoints placed on lines different than those specified with the breakpoint command.

You can also add a breakpoint directly at the start of the GPU kernel with `b saxpy` . To run to the next breakpoint, type `continue` (or `c`).

To list all the breakpoints that have been inserted type `info break` (or `i b`):

```
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000020b334 in main() at /HPCTrainingExamples/HIP/saxpy/saxpy.hip:74
2       breakpoint     keep y   0x000000000020b350 in main() at /HPCTrainingExamples/HIP/saxpy/saxpy.hip:78
```

A breakpoint can be removed with `delete <Num>` (or `d <Num>`): note that `<Num>` is the breakpoint ID displayed above. For instance, to remove the breakpoint at line 74, you have to do `d 1` .

To proceed to the next line you can do `next` (or `n`). To step into a function, do `step` (or `s`) and to get out do `finish` . Note that if a breakpoint is at a kernel, doing `n` or `s` will switch between different threads. To avoid this behavior, it is necessary to disable the breakpoint at the kernel with `disable <Num>` .

It is possible to have information on the architecture (below shown on MI250):

```
(gdb) info agents
  Id State Target Id                  Architecture Device Name                            Cores Threads Location
* 1  A     AMDGPU Agent (GPUID 64146) gfx90a       Aldebaran/MI200 [Instinct MI250X/MI250] 416   3328    29:00.0
```

We can also get information on the thread grid:

```
(gdb) info dispatches
  Id   Target Id                      Grid      Workgroup Fence   Kernel Function
* 1    AMDGPU Dispatch 1:1:1 (PKID 0) [256,1,1] [128,1,1] B|Aa|Ra saxpy(int, float const*, int, float*, int)
```

For the rocgdb documentation, please see: `/opt/rocm-<version>/share/doc/rocgdb` .

OpenMP Intro Examples

NOTE: these exercises have been tested on MI210 and MI300A accelerators using a container environment. To see details on the container environment (such as operating system and modules available) please see `README.md` on this repo.

Checking out makefiles and compiler toolchain

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

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`

OpenMP Offload – The Basics

We start out with the OpenMP threaded code for the CPU. This code is in

`~/HPCTrainingExamples/Pragma_Examples/OpenMP/Intro`

in the saxpy_cpu.cpp file. This is the code on slide 16. We first load the amdclang module which will set the CXX environment variable. This variable will get picked up by the Makefile for the build.

```
module load amdclang
make saxpy_cpu
./saxpy_cpu
```

The next example, saxpy1, is from slide 18 where the first version of OpenMP offloading is tried. In this code, there is no map clause. The compiler can figure out the arrays that need to be copied over and their sizes.

```
make saxpy1
./saxpy1
```

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.

The Fortran version of the saxpy code is shown in saxpy1f.F90. It is very similar to the C and C++ OpenMP pragmas. In Fortran, the compiler hints are technically directives that are contained in specially formatted comments. One of the strengths of OpenMP is that the language can be used in C, C++, and Fortran code and they can even be mixed in an application. Here is how to run the Fortran example.

```
make saxpy1f
./saxpy1f
```

The compile line uses the specific GPU architecture type. It grabs it from the rocminfo command with a little bit of string manipulation.

Let's now add a map clause as shown in quotes on slide 18 – map(tofrom:y[0:N])

```
make saxpy2
./saxpy2
```

A lot of the initial optimization of an OpenMP offloading port is to minimize the data movement from host to device and back. What is the optimum mapping of data for this example? See saxpy3.cpp for the optimal map clauses.

```
make saxpy3
./saxpy3
```

In the example we have been working with so far, the compiler can determine the sizes and will move the data for you. Let's see what happens when we have a subroutine with pointers where the compiler does not know the sizes.

```
make saxpy4
./saxpy4
```

Try removing the map clause – the program will now fail.

Multilevel Parallelism

We have been running on the GPU, but with only one thread in serial. Let's start adding parallelism. The first thing we can do is add `#pragma omp parallel for simd` before the loop to tell it to run in parallel.

```
make saxpy5
./saxpy5
```

We have told it to run the loop in parallel, but we haven't given it any hardware resources. To add more compute units, we need to add the teams clause. Then to spread the work across the threads, we need the distribute clause. (This code is currently not working . . .)

```
make saxpy6
./saxpy6
```

More commonly, we add the triplet of `target teams distribute` to the pragma to enable all hardware elements to the computation.

```
make saxpy7
./saxpy7
```

And in Fortran.

```
make saxpy2f
./saxpy2f
```

Structured and Unstructured Target Data Regions

This example from slide 29 shows the use of a structured block region that encompasses several compute loops. The data region persists across all of them, eliminating the need for map clauses and data transfers.

```
make target_data_structured
./target_data_structured
```

This example shows the use of the target data to map the data to the device and then updating it with the target update in the middle of the target data block.

```
make target_data_unstructured
./target_data_unstructured
```

When using larger data regions, it can be necessary to move data in the middle of the region to support MPI communication or I/O. This example shows the use of the update clause to copy new input from the host to the device.

```
make target_data_update
./target_data_update
```

Advanced OpenMP Presentation

Here, we will discuss some examples that show more advanced OpenMP features.

Memory Pragmas

First, we will consider the examples in the `CXX/memory_pragmas` directory:

```
cd ~/HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/memory_pragmas
```

Exercises Setup

Setup your environment:

```
export LIBOMPTARGET_INFO=-1
export OMP_TARGET_OFFLOAD=MANDATORY
```

The first flag above will allow you to see OpenMP activity, while the second terminates the program if code fails to be executed on device (as opposed to falling back on the host). 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))
```

Create a build directory and compile using `cmake` : this will place all executables in the `build` directory:

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

Mem1 (Initial Version)

There are 12 versions of an initial example code called `mem1.cc`, which is an implementation of a `daxpy` kernel with a single pragma with a map clause at the computational loop:

```
void daxpy(int n, double a, double *__restrict__ x, double *__restrict__ y, double *__restrict__ z)
{
#pragma omp target teams distribute parallel for simd map(to: x[0:n], y[0:n]) map(from: z[0:n])
        for (int i = 0; i < n; i++)
                z[i] = a*x[i] + y[i];
}
```

Run `mem1` to have an idea of what output is produced by the `LIBOMPTARGET_INFO=-1` flag, which should include OpenMP calls 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
```

Not all versions are discussed in this document. Using `vimdiff` to compare versions is useful to explore the differences, e.g.:

```
vimdiff mem1.cc mem2.cc
```

Mem2 (Add enter/exit data alloc/delete when memory is created/freed)

The initial code in `mem1.cc` is modified to obtain `mem2.cc` with the following additions:

```
#pragma omp target enter data map(alloc: x[0:n], y[0:n], z[0:n]) // line 52
```

```
#pragma omp target exit data map(delete: x[0:n], y[0:n], z[0:n]) // line 82
```

Mem3 (Replace map to/from with updates to bypass unneeded device memory check)

In `mem3.cc` , in addition to the changes in `mem2.cc` , the `daxpy` kernel is modified as follows:

```
void daxpy(int n, double a, double *__restrict__ x, double *__restrict__ y, double *__restrict__ z)
{
#pragma omp target update to (x[0:n], y[0:n])
#pragma omp target teams distribute parallel for simd
        for (int i = 0; i < n; i++)
                z[i] = a*x[i] + y[i];
#pragma omp target update from (z[0:n])
}
```

Mem4 (Replace delete with release to use reference counting)

Compared to `mem2.cc` , `mem4.cc` differs only at line 82, where a delete is replaced with a release:

```
#pragma omp target exit data map(release: x[0:n], y[0:n], z[0:n]) // line 82
```

Mem5 (Use enter data map to/from alloc/delete to reduce memory copies)

Similar to `mem2.cc` . this version differs from the original only at lines 52 and 82:

```
#pragma omp target enter data map(to: x[0:n], y[0:n]) map(alloc: z[0:n]) // line 52
```

```
#pragma omp target exit data map(from: z[0:n]) map(delete: x[0:n], y[0:n]) // line 82
```

Mem7 (Use managed memory to automatically move data)

In this example, we epxloit automatic memory management by the operating system. To enable it, export:

```
export HSA_XNACK=1
```

We also need to include the following pragma:

```
#pragma omp requires unified_shared_memory // line 22
```

Mem8 (Use unified shared memory with maps for backward compatibility)

Compared to `mem7.cc` , `mem8.cc` supports backward compatibility using maps and also:

```
#ifndef NO_UNIFIED_SHARED_MEMORY
#pragma omp requires unified_shared_memory
#endif
```

Mem12 (Only runs on MI300A)

This example uses the APU programming model of MI300A and unified addresses in OpenMP.

Kernel Pragmas

This set of exercises is in: `HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/kernel_pragmas` .

Exercises Setup

You should unset the `LIBOMPTARGET_INFO` environment flag if previously set.

```
unset LIBOMPTARGET_INFO
```

Then, set these environment variable

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

Brief Exercises Description

The example `kernel1.cc` is the same as `memory_pragmas/mem11.cc` except for the pragma line below (from `kernel1.cc`):

```
cout << "-Overall time is " << main_timer << endl;
#pragma omp target update from(z[0])
```

The example `kernel2.cc` differs from `kernel1.cc` as it adds `num_threads(64)` to the pragma line in the `daxpy` kernel:

```
void daxpy(int n, double a, double *__restrict__ x, double *__restrict__ y, double *__restrict__ z)
{
#pragma omp target teams distribute parallel for simd num_threads(64)
        for (int i = 0; i < n; i++)
                z[i] = a*x[i] + y[i];
}
```

Similarly, example `kernel3.cc` differs from `kernel1.cc` as it adds `num_threads(64) thread_limit(64)` to the pragma line in the `daxpy` kernel:

```
void daxpy(int n, double a, double *__restrict__ x, double *__restrict__ y, double *__restrict__ z)
{
#pragma omp target teams distribute parallel for simd num_threads(64) thread_limit(64)
        for (int i = 0; i < n; i++)
                z[i] = a*x[i] + y[i];
}
```

Something to test On your own: uncomment line 15 in CMakeLists.txt (the one with -faligned-allocation -fnew-alignment=256).

Another option to explore is adding the attribute (std::align_val_t(128)) to each new line, for example:

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

Real-World OpenMP Language Constructs

For all excercises in this section:

```
module load amdclang
git clone https://github.com/AMD/HPCTrainingExamples
```

either choose

```
cd ~/HPCTrainingExamples/Pragma_Examples/OpenMP/Fortran
```

or

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

Note: make sure the compilers are set to your preference. This can be obtained by exporting the `FC` and `CC` environment variables:

```
export FC=<my favorite Fortran compiler>
export CC=<my favorite C compiler>
```

It is suggested for those that want to truly experience the effort, that you take all the pragma statements out of these examples and do the port yourself.

Simple Reduction

The first example is a simple reduction:

```
cd reduction_scalar
make
./reduction_scalar
```

Now try the array form

```
cd ../reduction_array
make
./reduction_array
```

If your compiler passes, it supports at least simple array reduction clauses

Device Routine

Subroutines called from within a target region also cause some difficulties. We must tell the compiler that we want these compiled for the GPU. Note that device routines are not (yet) supported by all compilers!

For this example

```
cd ../device_routine
```

there are multiple versions to choose from in Fortran, either with an interface and an external routine or using a module. Hence one first needs to enter the selected subfolder, and then:

```
make
./device_routine
```

Device Routine with Global Data

Including the use of data from global scope in device routines also causes difficulties. We have examples for both statically sized arrays and dynamically allocated global data. Note that device routines are not (yet) supported by all compilers! Also, this excercise only exists in the C version at the moment.

```
cd ../device_routine_wglobaldata
make
./device_routine
```

```
cd ../device_routine_wdynglobaldata
make
./device_routine
```

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
```

GPU Aware MPI

Point-to-point and collective

NOTE: these exercises have been tested on MI210 and MI300A accelerators using a container environment. To see details on the container environment (such as operating system and modules available) please see `README.md` on this repo.

Allocate at least two GPUs and set up your environment

```
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 -mca pml ucx ./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-7.3.tar.gz
tar -xvf osu-micro-benchmarks-7.3.tar.gz
```

Create a build directory and cd to osu-micro-benchmarks-7.3

```
mkdir build
cd osu-micro-benchmarks-7.3
module load rocm openmpi
```

Build and install OSU micro-benchmarks

```
./configure --prefix=`pwd`/../build/ \
                CC=`which mpicc` \
                CXX=`which mpicxx` \
                CPPFLAGS=-D__HIP_PLATFORM_AMD__=1 \
                --enable-rocm \
                --with-rocm=${ROCM_PATH}
make -j12
make install
```

If you get the error "cannot include hip/hip__runtime__api.h", grep for **HIP__PLATFORM__HCC** and replace it with **HIP__PLATFORM__AMD** in configure.ac and configure files.

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

```
mpirun -n 2 -mca pml ucx ../build/libexec/osu-micro-benchmarks/mpi/pt2pt/osu_bw \
                -m $((16*1024*1024)) D D
```

Notes: - Try different pairs of GPUs. - Run the command "rocm-smi –showtopo" to see the link type between the pairs of GPUs. - How does the bandwidth vary for xGMI connected GPUs vs PCIE connected GPUs?

Ghost Exchange example

This example takes an MPI Ghost Exchange code that runs on the CPU and ports it to the GPU and GPU-aware MPI.

```
module load amdclang openmpi
git clone https://github.com/amd/HPCTrainingExamples.git
cd HPCTrainingExamples/MPI-examples/GhostExchange/GhostExchange_ArrayAssign/Orig
mkdir build && cd build
cmake ..
make
mpirun -n 8 --mca pml ucx ./GhostExchange \
-x 4  -y 2  -i 20000 -j 20000 -h 2 -t -c -I 1000
```

We can improve this performance by using process placement so that we are using all the memory channels.

On MI2100 nodes, we have 2 NUMA per node. So we can assign 4 ranks per NUMA when running with 8 ranks:

```
mpirun -n 8 --mca pml ucx --bind-to core --map-by ppr:4:numa --report-bindings \

./GhostExchange  -x 4  -y 2  -i 20000 -j 20000 -h 2 -t -c -I 1000
```

On MI300A node, we have 4 NUMA per node. So we can assign 2 ranks per NUMA when running with 8 ranks:

```
mpirun -n 8 --mca pml ucx --bind-to core --map-by ppr:2:numa --report-bindings \
./GhostExchange -x 4  -y 2  -i 20000 -j 20000 -h 2 -t -c -I 1000
```

For the port to the GPU, we are going to take advantage of Managed Memory (or single memory space on MI300A)

```
export HSA_XNACK=1
cd ../Ver1
mkdir build && cd build
cmake ..
make
mpirun -n 8 --mca pml ucx --bind-to core --map-by ppr:4:numa \
-x HIP_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 \
                    ./GhostExchange -x 4  -y 2  -i 20000 -j 20000 -h 2 -t -c -I 1000
```

Alternatively, on MI300A, we can run with:

```
mpirun -n 8 --mca pml ucx --bind-to core --map-by ppr:2:numa \
-x HIP_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 \
                    ./GhostExchange -x 4  -y 2  -i 20000 -j 20000 -h 2 -t -c -I 1000
```

The MPI buffers are only used on the GPU, so we can just allocate them there and save memory on the CPU.

```
export HSA_XNACK=1
cd ../Ver3
mkdir build && cd build
cmake ..
make
mpirun -n 8 --mca pml ucx --bind-to core --map-by ppr:4:numa \
-x HIP_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 \

                    ./GhostExchange -x 4  -y 2  -i 20000 -j 20000 -h 2 -t -c -I 1000
```

Alternatively, on MI300A, we can run with:

```
mpirun -n 8 --mca pml ucx --bind-to core --map-by ppr:2:numa \
-x HIP_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 \
                    ./GhostExchange -x 4  -y 2  -i 20000 -j 20000 -h 2 -t -c -I 1000
```

Memory allocations can be expensive for the GPU. This next version just allocates the MPI buffers once in the main routine.

```
export HSA_XNACK=1
cd ../Ver3
mkdir build && cd build
cmake ..
make
mpirun -n 8 --mca pml ucx --bind-to core --map-by ppr:4:numa \
-x HIP_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 \
                    ./GhostExchange -x 4  -y 2  -i 20000 -j 20000 -h 2 -t -c -I 1000cd
```

Alternatively, on MI300A, we can run with:

```
mpirun -n 8 --mca pml ucx --bind-to core --map-by ppr:2:numa \
-x HIP_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 \
                   ./GhostExchange -x 4  -y 2  -i 20000 -j 20000 -h 2 -t -c -I 1000
```

Rocprof

NOTE: these exercises have been tested on MI210 and MI300A accelerators using a container environment. To see details on the container environment (such as operating system and modules available) please see `README.md` on this repo.

We discuss an example on how to use the tools from `rocprof` .

Initial Setup

First, setup the environment:

```
salloc --cpus-per-task=8 --mem=0 --ntasks-per-node=4 --gpus=1
module load rocm
```

Download the 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 and run the `nbody-orig.hip` program (the script below will do both, for several values of `nBodies`):

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

To compile explicitly without `make` you can do (considering for example `nbody-orig`):

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

And then run with:

```
./nbody-orig <nBodies>
```

The procedure for compiling and running a single example applies to the other programs in the directory. The default value for `nBodies` is 30000 for all the examples.

Run ROCprof and Inspect the Output

Run `rocprof` to obtain the hotspots list (considering for example `nbody-orig`):

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

In the above command, the `--basenames on` flag removes the kernel arguments from the output, for ease of reading. Throughout this example, we will always use 65536 as a value for `nBodies` , since `nBodies` is used to define the number of work groups in the thread grid:

```
nBlocks = (nBodies + BLOCK_SIZE - 1) / BLOCK_SIZE
```

Check `results.csv` to find, for each invocation of each kernel, details such as grid size (`grd`), workgroup size (`wgr`), LDS used (`lds`), scratch used if register spilling happened (`scr`), number of SGPRs and VGPRs used, etc. Note that grid size is equal to the total number of ***work-items (threads)***, not the number of work groups. This is the output that is useful if you allocate shared memory dynamically, for instance.

Additionally, you can check the statistics result file called `results.stats.csv` , displayed one line per kernel, sorted in descending order of durations.

You can trace HIP, GPU and Copy activity with `--hip-trace` :

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

The output is the file `results.hip_stats.csv` , which lists the HIP API calls and their durations, sorted in descending order. This can be useful to find HIP API calls that may be bottlenecks.

You can also profile the HSA API by adding the `--hsa-trace` option. This is useful if you are profiling OpenMP target offload code, for instance, as the compiler implements all GPU offloading via the HSA layer:

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

In addition to `results.hip_stats.csv` , the command above will create the file `results.hsa_stats.csv` which contains the statistics information for HSA calls.

Visualization with Perfetto

The `results.json` JSON file produced by `rocprof` can be downloaded to your local machine and viewed in Perfetto UI. This file contains the timeline trace for this application, but shows only GPU, Copy and HIP API activity.

Once you have downloaded the file, 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

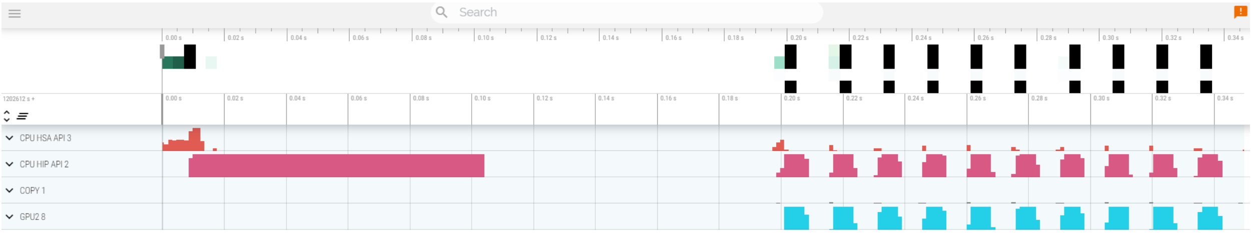


Figure 1: image

To read about the GPU hardware counters available, inspect the output of the following command:

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

In the output displayed, look for the section associated with the hardware on which you are running (for instance gfx90a).

Create a `rocprof_counters.txt` file with the counters you would like to collect, for instance:

```
touch rocprof_counters.txt
```

and write this in `rocprof_counters.txt` as an example:

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

Execute with the counters we just added, including the `timestamp on` option which turns on GPU kernel timestamps:

```
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.

View the contents of `rocprof_counters.csv` for the collected counter values for each invocation of each kernel:

```
cat rocprof\_counters.csv
```

Omniperf Examples

Exercise 1: Launch Parameter Tuning

Simple kernel implementing a version of yAx, to demonstrate effects of Launch Parameters on kernel execution time.

Note: This exercise was tested on a system with MI210s, on a recent commit of Omniperf version `2.0.0` and ROCm `6.0.0` . **Any Omniperf version `2.0.0` or greater is incompatible with versions of ROCm less than `6.0.0` .**

Client-side installation instructions are available in the official omniperf documentation, and provide all functionality demonstrated here.

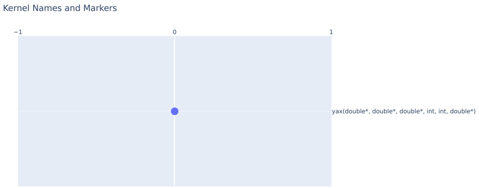
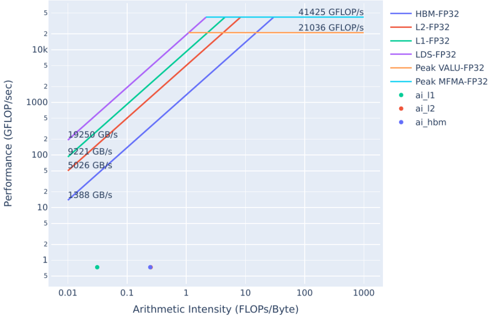
If your system has an older version of Omniperf, please refer to the archived READMEs in this directory and use a ROCm version lesser than `6.0.0` .

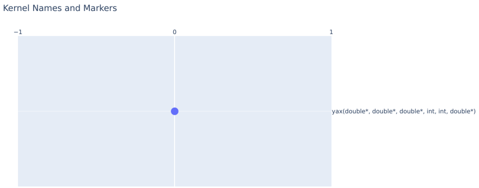
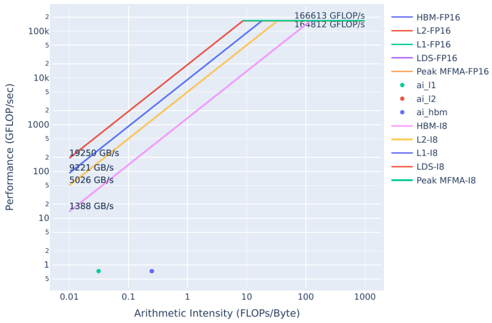
Background: Acronyms and terms used in this exercise

- yAx: a vector-matrix-vector product, yAx, where y and x are vectors, and A is a matrix
- FP(32/16): 32- or 16-bit Floating Point numeric types
- FLOPs: Floating Point Operations Per second
- HBM: High Bandwidth Memory is globally accessible from the GPU, and is a level of memory above the L2 cache

Initial Roofline Analysis:

The roofline model is a way to gauge kernel performance in terms of maximum achievable bandwidth and floating-point operations. It can be used to determine how efficiently a kernel makes use of the available hardware. It is a key tool in initially determining which kernels are performing well, and which kernels should be able to perform better. Below are roofline plots for the yAx kernel in problem.cpp:

Roofline Type	Roofline Legend	Roofline Plot
FP32/FP64		

Roofline Type	Roofline Legend	Roofline Plot
FP16/INT8		

These plots were generated by running:

```
omniperf profile -n problem_roof_only --roof-only --kernel-names -- ./problem.exe
```

The plots will appear as PDF files in the `./workloads/problem_roof_only/MI200` directory, if generated on MI200 hardware.

We see that the kernel's performance is not near the achievable bandwidth possible on the hardware, which makes it a good candidate to consider optimizing.

Exercise instructions:

From the roofline we were able to see that there is room for improvement in this kernel. One of the first things to check is whether or not we have reasonable launch parameters for this kernel.

To get started, build and run the problem code:

```
make
./problem.exe
```

(*simulated output*)

```
yAx time: 2911 milliseconds
```

The runtime of the problem should be very slow, due to sub-optimal launch parameters. Let's confirm this hypothesis by looking at the omniperf profile. Start by running:

```
omniperf profile -n problem --no-roof -- ./problem.exe
```

This command requires omniperf to run your code a few times to collect all the necessary hardware counters. - `-n problem` names the workload, meaning that the profile will appear in the `./workloads/problem/MI200/` directory, if you are profiling on an MI200 device. - `--no-roof` turns off the roofline, which will save some profiling time by avoiding the collection of achievable bandwidths and FLOPs on the device. - Everything after the `--` is the command that will be profiled.

After the profiling data is collected, we can view the profile by using this command:

```
omniperf analyze -p workloads/problem/MI200 --dispatch 1 --block 7.1.0 7.1.1 7.1.2
```

This allows us to view nicely formatted profiling data directly in the command line. The command given here has a few arguments that are noteworthy: - `-p workloads/problem/MI200` must point to the output directory of your profile run. For the above `omniperf profile` command, this will be `workloads/problem/MI200` . - `--dispatch 1` filters kernel statistics by dispatch ID. In this case kernel 0 was a "warm-up" kernel, and kernel 1 is what the code reports timings for. - `--block` displays only the requested metrics, in this case we want metrics specific to Launch Parameters: - `7.1.0` is the Grid Size - `7.1.1` is the Workgroup Size - `7.1.2` is the Total Wavefronts Launched

The output of the `omniperf analyze` command should look something like this:

```
  ___                  _                  __
 / _ \ _ __ ___  _ __ (_)_ __   ___ _ __ / _|
| | | | '_ ` _ \| '_ \| | '_ \ / _ \ '__| |_
| |_| | | | | | | | | | | |_) |  __/ |  |  _|
 \___/|_| |_| |_|_| |_|_| .__/ \___|_|  |_|
                        |_|

Analysis mode = cli
[analysis] deriving Omniperf metrics...

0. Top Stats
0.1 Top Kernels
-----------------------------------------------------------------------------------------------------------------
|    | Kernel_Name                              |   Count |      Sum(ns) |     Mean(ns) |   Median(ns) |    Pct |
-----|------------------------------------------|---------|--------------|--------------|--------------|---------
|  0 | yax(double*, double*, double*, int, int, |    1.00 | 751342314.00 | 751342314.00 | 751342314.00 | 100.00 |
|    |  double*) [clone .kd]                    |         |              |              |              |        |
-----------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
|    |   Dispatch_ID | Kernel_Name                                                   |   GPU_ID |
-----|---------------|---------------------------------------------------------------|-----------
|  0 |             1 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        2 |
-------------------------------------------------------------------------------------------------


7. Wavefront
7.1 Wavefront Launch Stats
--------------------------------------------------------------------------
| Metric_ID    | Metric           |    Avg |    Min |    Max | Unit       |
--------------|------------------|--------|--------|--------|-------------
| 7.1.0        | Grid Size        | 256.00 | 256.00 | 256.00 | Work items |
--------------|------------------|--------|--------|--------|-------------
| 7.1.1        | Workgroup Size   |  64.00 |  64.00 |  64.00 | Work items |
--------------|------------------|--------|--------|--------|-------------
| 7.1.2        | Total Wavefronts |   4.00 |   4.00 |   4.00 | Wavefronts |
--------------------------------------------------------------------------
```

Looking through this data we see: - Workgroup Size (`7.1.1`) is 64 threads, which corresponds with the size of a wavefront. - Total Wavefronts (`7.1.2`) shows that we are launching only 4 Wavefronts.

We can definitely get better performance by adjusting the launch parameters of our kernel. Either try out some new values for the launch bounds, or run the provided solution to see its performance:

```
cd solution
make
./solution.exe
```

(*simulated output*)

```
yAx time: 70 ms
```

We get much better performance with the new launch parameters. Note that in general it can be difficult to find the most optimal launch parameters for a given kernel due to the many factors that impact performance, so determining launch parameters experimentally is usually necessary.

We should also confirm that our updated launch parameters are reported by omniperf, we need to run:

```
omniperf profile -n solution --no-roof -- ./solution.exe
```

This command is the same as before, except the workload name has changed to `solution` . Once the `profile` command has completed, run:

```
omniperf analyze -p workloads/solution/MI200 --dispatch 1 --block 7.1.0 7.1.1 7.1.2
```

Again, this command largely uses the same arguments as before, except for the workload name. The output should look something like this:

```
  ___                  _                  __ 
 / _ \ _ __ ___  _ __ (_)_ __   ___ _ __ / _|
| | | | '_ ` _ \| '_ \| | '_ \ / _ \ '__| |_ 
| |_| | | | | | | | | | | |_) |  __/ |  |  _|
 \___/|_| |_| |_|_| |_|_| .__/ \___|_|  |_|  
                        |_|                  

Analysis mode = cli
[analysis] deriving Omniperf metrics...

0. Top Stats
0.1 Top Kernels
---------------------------------------------------------------------------------------------------------------
|    | Kernel_Name                              |   Count |     Sum(ns) |    Mean(ns) |   Median(ns) |    Pct |
-----|------------------------------------------|---------|-------------|-------------|--------------|---------
|  0 | yax(double*, double*, double*, int, int, |    1.00 | 69512860.00 | 69512860.00 |  69512860.00 | 100.00 |
|    |  double*) [clone .kd]                    |         |             |             |              |        |
---------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
|    |   Dispatch_ID | Kernel_Name                                                   |   GPU_ID |
-----|---------------|---------------------------------------------------------------|-----------
|  0 |             1 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        2 |
-------------------------------------------------------------------------------------------------


7. Wavefront
7.1 Wavefront Launch Stats
-----------------------------------------------------------------------------------
| Metric_ID    | Metric           |       Avg |       Min |       Max | Unit       |
--------------|------------------|-----------|-----------|-----------|-------------
| 7.1.0        | Grid Size        | 131072.00 | 131072.00 | 131072.00 | Work items |
--------------|------------------|-----------|-----------|-----------|-------------
| 7.1.1        | Workgroup Size   |     64.00 |     64.00 |     64.00 | Work items |
--------------|------------------|-----------|-----------|-----------|-------------
| 7.1.2        | Total Wavefronts |   2048.00 |   2048.00 |   2048.00 | Wavefronts |
-----------------------------------------------------------------------------------
```

Looking through this data we see: - Workgroup Size (`7.1.1`) corresponds to the first argument of the block launch parameter - Total Wavefronts (`7.1.2`) corresponds to the first index of the grid launch parameter - Grid size (`7.1.0`) is Workgroup Size (`7.1.1`) times Total Wavefronts (`7.1.2`)

Omniperf Command Line Comparison Feature

On releases newer than Omniperf 1.0.10, the comparison feature of omniperf can be used to quickly compare two profiles. To use this feature, use the command:

```
omniperf analyze -p workloads/problem/MI200 -p solution/workloads/solution/MI200 --dispatch 1 --block 7
```

This feature sets the first `-p` argument as the baseline, and the second as the comparison workload. In this case, problem is set as the baseline and is compared to solution. The output should look like:

```
  ___                  _                  __ 
 / _ \ _ __ ___  _ __ (_)_ __   ___ _ __ / _|
| | | | '_ ` _ \| '_ \| | '_ \ / _ \ '__| |_ 
| |_| | | | | | | | | | | |_) |  __/ |  |  _|
 \___/|_| |_| |_|_| |_|_| .__/ \___|_|  |_|  
                        |_|                  

Analysis mode = cli
[analysis] deriving Omniperf metrics...

--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
---------------------------------------------------------------------------------------------------------------------------
|    | Kernel_Name                              |   Count | Count      |   Abs Diff |      Sum(ns) | Sum(ns)              |
-----|------------------------------------------|---------|------------|------------|--------------|----------------------|
|  0 | yax(double*, double*, double*, int, int, |    1.00 | 1.0 (0.0%) |       0.00 | 751342314.00 | 69512860.0 (-90.75%) |
|    |  double*) [clone .kd]                    |         |            |            |              |                      |
-----------------------------------------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------------------------------------------------
|    | Kernel_Name                              | Mean(ns)             |   Median(ns) | Median(ns)           |    Pct | Pct          |
-----|------------------------------------------|----------------------|--------------|----------------------|--------|--------------
|  0 | yax(double*, double*, double*, int, int, | 69512860.0 (-90.75%) | 751342314.00 | 69512860.0 (-90.75%) | 100.00 | 100.0 (0.0%) |
|    |  double*) [clone .kd]                    |                      |              |                      |        |              |
--------------------------------------------------------------------------------------------------------------------------------------

0.2 Dispatch List
-------------------------------------------------------------------------------------------------
|    |   Dispatch_ID | Kernel_Name                                                   |   GPU_ID |
-----|---------------|---------------------------------------------------------------|----------
|  0 |             1 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        2 |
-------------------------------------------------------------------------------------------------


---------------------------------------------------------------------------------
7. Wavefront
7.1 Wavefront Launch Stats
-----------------------------------------------------------------------------------------------------------------------------------------------------------
| Metric_ID    | Metric           |    Avg | Avg                 |   Abs Diff |    Min | Min                 |    Max | Max                 | Unit       |
---------------|------------------|--------|---------------------|------------|--------|---------------------|--------|---------------------|------------
| 7.1.0        | Grid Size        | 256.00 | 131072.0 (51100.0%) |  130816.00 | 256.00 | 131072.0 (51100.0%) | 256.00 | 131072.0 (51100.0%) | Work items |
---------------|------------------|--------|---------------------|------------|--------|---------------------|--------|---------------------|------------
| 7.1.1        | Workgroup Size   |  64.00 | 64.0 (0.0%)         |       0.00 |  64.00 | 64.0 (0.0%)         |  64.00 | 64.0 (0.0%)         | Work items |
---------------|------------------|--------|---------------------|------------|--------|---------------------|--------|---------------------|------------
| 7.1.2        | Total Wavefronts |   4.00 | 2048.0 (51100.0%)   |    2044.00 |   4.00 | 2048.0 (51100.0%)   |   4.00 | 2048.0 (51100.0%)   | Wavefronts |
-----------------------------------------------------------------------------------------------------------------------------------------------------------
```

Note that the comparison workload shows the percentage difference from the baseline. This feature can be used to quickly compare filtered stats to make sure code changes fix known issues.

More Kernel Filtering

For this exercise, it is appropriate to filter the `omniperf analyze` command with the `--dispatch 1` argument. This `--dispatch 1` argument filters the data shown to only include the kernel invocation with dispatch ID 1, or the second kernel run during profiling.

However, there is another way to filter kernels that may be more applicable in real use-cases. Typically real codes launch many kernels, and only a few of them take most of the overall kernel runtime. To see a ranking of the top kernels that take up most of the kernel runtime in your code, you can run:

```
omniperf analyze -p workloads/problem/MI200 --list-stats
```

This command will output something like:

```
 / _ \ _ __ ___  _ __ (_)_ __   ___ _ __ / _|
| | | | '_ ` _ \| '_ \| | '_ \ / _ \ '__| |_
| |_| | | | | | | | | | | |_) |  __/ |  |  _|
 \___/|_| |_| |_|_| |_|_| .__/ \___|_|  |_|
                        |_|

Analysis mode = cli
[analysis] deriving Omniperf metrics...

Detected Kernels (sorted descending by duration)
----------------------------------------------------------------------
|    | Kernel_Name                                                   |
-----|----------------------------------------------------------------
|  0 | yax(double*, double*, double*, int, int, double*) [clone .kd] |
----------------------------------------------------------------------

Dispatch list
-------------------------------------------------------------------------------------------------
|    |   Dispatch_ID | Kernel_Name                                                   |   GPU_ID |
-----|---------------|---------------------------------------------------------------|----------
|  0 |             0 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        2 |
-----|---------------|---------------------------------------------------------------|----------
|  1 |             1 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        2 |
-------------------------------------------------------------------------------------------------
```

Using Omniperf versions greater than `2.0.0` , `--list-stats` will list all kernels launched by your code, in order of runtime (largest runtime first). The number displayed beside the kernel in the output can be used to filter `omniperf analyze` commands. **Note that this will display aggregated stats for kernels of**

the same name, meaning that the invocations could differ in terms of launch parameters, and vary widely in terms of work completed. This filtering is accomplished with the `-k` argument:

```
omniperf analyze -p workloads/problem/MI200 -k 0 --block 7.1.0 7.1.1 7.1.2
```

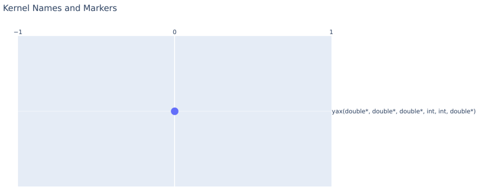
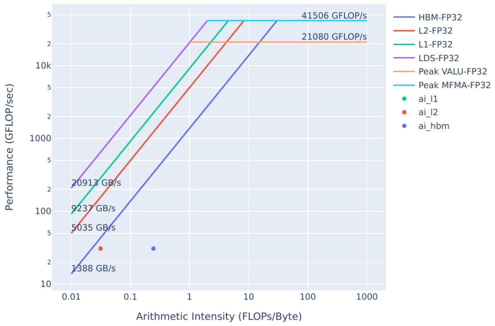
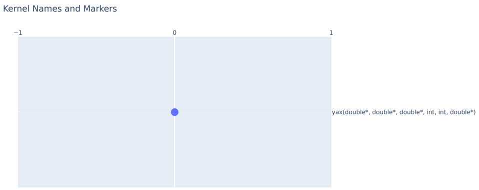
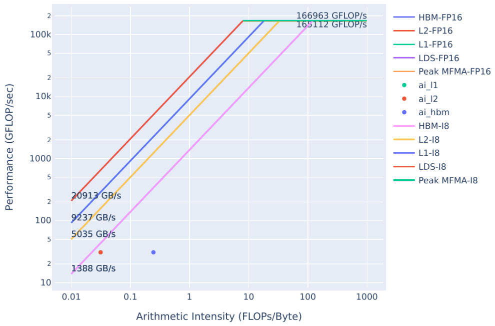
Which should show something like:

```
 / _ \ _ __ ___  _ __ (_)_ __   ___ _ __ / _|
| | | | '_ ` _ \| '_ \| | '_ \ / _ \ '__| |_
| |_| | | | | | | | | | | |_) |  __/ |  |  _|
 \___/|_| |_| |_|_| |_|_| .__/ \___|_|  |_|
                        |_|

Analysis mode = cli
[analysis] deriving Omniperf metrics...

0. Top Stats
0.1 Top Kernels
------------------------------------------------------------------------------------------------------------------------
|    | Kernel_Name                              |   Count |       Sum(ns) |     Mean(ns) |   Median(ns) |    Pct | S   |
-----|------------------------------------------|---------|---------------|--------------|--------------|--------|------
|  0 | yax(double*, double*, double*, int, int, |    2.00 | 1501207023.00 | 750603511.50 | 750603511.50 | 100.00 | *   |
|    |  double*) [clone .kd]                    |         |               |              |              |        |     |
------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
|    |   Dispatch_ID | Kernel_Name                                                   |   GPU_ID |
-----|---------------|---------------------------------------------------------------|-----------
|  0 |             0 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        2 |
-----|---------------|---------------------------------------------------------------|-----------
|  1 |             1 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        2 |
-------------------------------------------------------------------------------------------------


7. Wavefront
7.1 Wavefront Launch Stats
----------------------------------------------------------------------------
| Metric_ID    | Metric            |    Avg |    Min |    Max | Unit       |
---------------|------------------|--------|--------|--------|-------------
| 7.1.0        | Grid Size         | 256.00 | 256.00 | 256.00 | Work items |
---------------|------------------|--------|--------|--------|-------------
| 7.1.1        | Workgroup Size    |  64.00 |  64.00 |  64.00 | Work items |
---------------|------------------|--------|--------|--------|-------------
| 7.1.2        | Total Wavefronts |   4.00 |   4.00 |   4.00 | Wavefronts |
----------------------------------------------------------------------------
```

Note that the ‘count’ field in Top Stat is 2 here, where filtering by dispatch ID displays a count of 1, indicating that filtering with `-k` returns aggregated stats for two kernel invocations in this case. Also note that the “Top Stats” table will still show all the top kernels but the rightmost column titled “S” (think “Selected”) will have an asterisk beside the kernel for which data is being displayed. Also note that the dispatch list displays two entries rather than the one we see when we filter by `--dispatch 1` .

Solution Roofline

We’ve demonstrated better performance than problem.cpp in solution.cpp, but could we potentially do better? To answer that we again turn to the roofline model:

Roofline Type	Roofline Legend	Roofline Plot
FP32/FP64		
FP16/INT8		

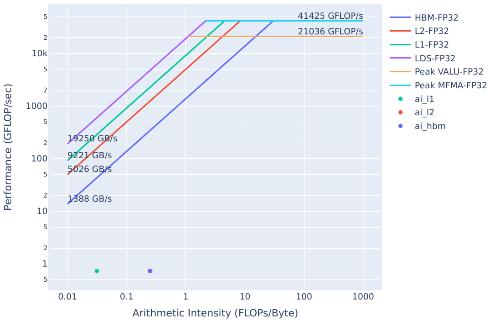
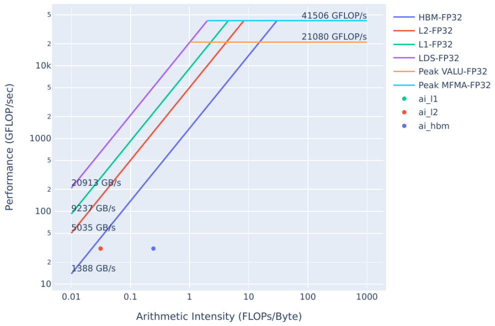
These plots were generated with:

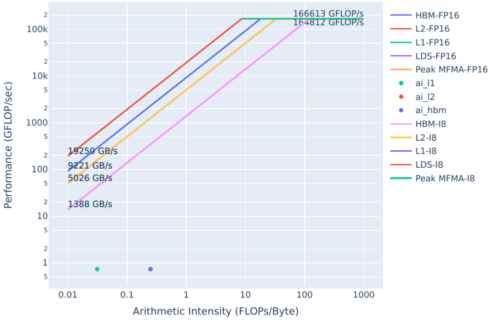
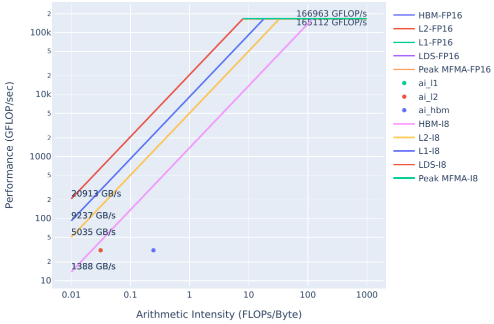
```
omniperf profile -n solution_roof_only --roof-only --kernel-names -- ./solution.exe
```

The plots will appear as PDF files in the `./workloads/solution_roof_only/MI200` directory, if generated on MI200 hardware.

We see that the solution is solidly in the bandwidth-bound regime, but even still there seems to be room for improvement. Further performance improvements will be a topic for later exercises.

Roofline Comparison

Roofline Type	Problem Roofline	Solution Roofline
FP32/FP64		

Roofline Type	Problem Roofline	Solution Roofline
FP16/INT8		

We see that the solution has drastically increased performance over the problem code, as shown by the solution points moving up closer to the line plotted by the bandwidth limit.

Note: on statically generated roofline images, it is possible for the L1, L2, or HBM points to overlap and hide one another.

Summary and Take-aways

Launch parameters should be the first check in optimizing performance, due to the fact that they are usually easy to change, but can have a large performance impact if they aren't tuned to your workload. It is difficult to predict the optimal launch parameters for any given kernel, so some experimentation may be required to achieve the best performance.

Exercise 2: LDS Occupancy Limiter

Simple kernel implementing a version of yAx, to demonstrate the downside of allocating a large amount of LDS, and the benefit of using a smaller amount of LDS due to occupancy limits.

Note: This exercise was tested on a system with MI210s, on omniperf version `2.0.0` and ROCm `6.0.2` **Omniperf `2.0.0` is incompatible with ROCm versions lesser than `6.0.0`**

Background: Acronyms and terms used in this exercise

- Wavefront: A collection of threads, usually 64.
- Workgroup: A collection of Wavefronts (at least 1), which can be scheduled on a Compute Unit (CU)
- LDS: Local Data Store is Shared Memory that is accessible to the entire workgroup on a Compute Unit (CU)
- CU: The Compute Unit is responsible for executing the User's kernels
- SPI: Shader Processor Input, also referred to as the Workgroup Manager, is responsible for scheduling workgroups on Compute Units
- Occupancy: A measure of how many wavefronts are executing on the GPU on average through the duration of the kernel
- PoP: Percent of Peak refers to the ratio of an achieved value and a theoretical or actual maximum. In terms of occupancy, it is how many wavefronts on average were on the device divided by how many can fit on the device.
- yAx: a vector-matrix-vector product, yAx, where y and x are vectors, and A is a matrix
- FP(32/16): 32- or 16-bit Floating Point numeric

- FLOPs: Floating Point Operations Per second
- HBM: High Bandwidth Memory is globally accessible from the GPU, and is a level of memory above the L2 cache

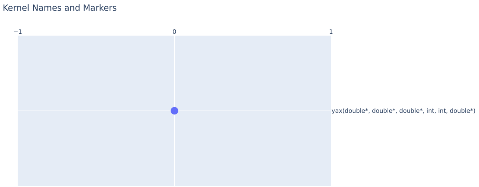
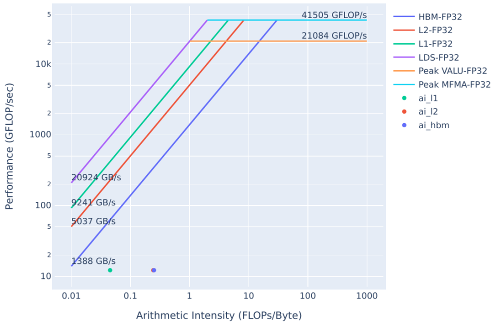
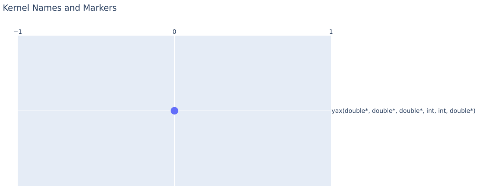
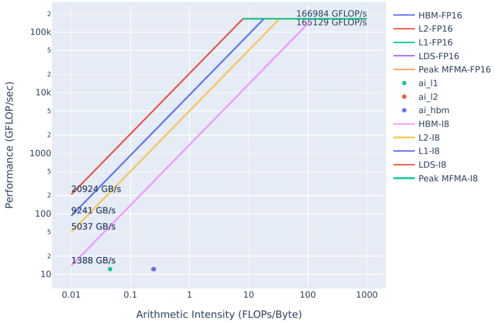
Initial Roofline Analysis

In this exercise we're using a problem code that is slightly different than where we left off in Exercise 1. Regardless, to get started we need to get a roofline by running:

```
omniperf profile -n problem_roof_only --roof-only --kernel-names -- ./problem.exe
```

The plots will appear as PDF files in the `./workloads/problem_roof_only/MI200` directory, if generated on MI200 hardware.

For convenience, the resulting plots on a representative system are below:

Roofline Type	Roofline Legend	Roofline Plot
FP32/FP64		
FP16/INT8		

We see that there looks to be room for improvement here. We'll use omniperf to see what the current limiters are.

Exercise Instructions:

First, we should get an idea of the code's runtime:

```
make
./problem.exe
```

(*simulated output*)

```
yAx time: 140 ms
```

This problem.cpp uses LDS allocations to move the x vector closer to the compute resources, a common optimization. However, we see that it ends up slower than the previous solution that didn't use LDS at all. In kernels that request a lot of LDS, it is common to see that the LDS usage limits the occupancy of the kernel. That is, more wavefronts cannot be resident on the device, because all of them need more LDS than is available. We need to confirm this hypothesis, let's start by running:

```
omniperf profile -n problem --no-roof -- ./problem.exe
```

The usage of `omniperf profile` arguments can be found here, or by running `omniperf profile --help` .

This `omniperf profile` command will take a minute or two to run, as omniperf must run your code a few times to collect all the hardware counters.

> **Note:** For large scientific codes, it can be useful to profile a small representative workload if possible, as profiling a full run may take prohibitively long.

Once the profiling run completes, let's take a look at the occupancy stats related to LDS allocations:

```
omniperf analyze -p workloads/problem/MI200 --dispatch 1 --block 2.1.15 6.2.7
```

The metrics we're looking at are: - `2.1.15` Wavefront occupancy – a measure of how many wavefronts, on average, are active on the device - `6.2.7` SPI: Insufficient CU LDS – indicates whether wavefronts are not able to be scheduled due to insufficient LDS

The SPI section (`6.2`) generally shows what resources limit occupancy, while Wavefront occupancy (`2.1.15`) shows how severely occupancy is limited in general. As of Omniperf version `2.0.0` , the SPI 'insufficient' fields are a percentage showing how frequently a given resource prevented the SPI from scheduling a wavefront. If more than one field is nonzero, the relative magnitude of the nonzero fields correspond to the relative severity of the corresponding occupancy limitation (a larger percentage means a resource limits occupancy more than another resource with a smaller percentage), but it is usually impossible to closely correlate the SPI 'insufficient' percentage with the overall occupancy limit. This could mean you reduce a large percentage in an 'insufficient' resource field to zero, and see overall occupancy only increase by a comparatively small amount.

Background: A note on occupancy's relation to performance

Occupancy has a fairly complex relation to achieved performance. In cases where the device is not saturated (where resources are available, but are unused) there is usually performance that can be gained by increasing occupancy, but not always. For instance, adversarial data access patterns (see exercise 4-StridedAccess) can cause occupancy increases to result in degraded performance, due to overall poorer cache utilization. Typically adding to occupancy gains performance up to a point beyond which performance degrades, and this point may have already been reached by an application before optimizing.

The output of the `omniperf analyze` command should look similar to this:

```
  ___                  _                  __
 / _ \ _ __ ___  _ __ (_)_ __   ___ _ __ / _|
| | | | '_ ` _ \| '_ \| | '_ \ / _ \ '__| |_
| |_| | | | | | | | | | | |_) |  __/ |  |  _|
 \___/|_| |_| |_|_| |_|_| .__/ \___|_|  |_|
                        |_|

Analysis mode = cli
[analysis] deriving Omniperf metrics...

-----------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
--------------------------------------------------------------------------------------------
|    | Kernel_Name                         |   Count |      Sum(ns) |     Mean(ns) |   Median(ns) |    Pct |
-----|-------------------------------------|---------|--------------|--------------|--------------|---------
```

```
|  0 | yax(double*, double*, double*, int, int, |    1.00 | 176224652.00 | 176224652.00 | 176224652.00 | 100.00 |
|    |  double*) [clone .kd]                    |         |              |              |              |        |
-----------------------------------------------------------------------------------------------------------------
0.2 Dispatch List
-------------------------------------------------------------------------------------------------
|    |   Dispatch_ID | Kernel_Name                                                   |   GPU_ID |
-----|----------------|---------------------------------------------------------------|-----------
|  0 |             1 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        8 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
-------------------------------------------------------------------------------------
| Metric_ID   | Metric              |    Avg | Unit       |    Peak |   Pct of Peak |
--------------|---------------------|--------|------------|---------|-----------------
| 2.1.15      | Wavefront Occupancy | 103.00 | Wavefronts | 3328.00 |          3.10 |
-------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
----------------------------------------------------------------------
| Metric_ID   | Metric              |   Avg |   Min |   Max | Unit   |
--------------|---------------------|-------|-------|-------|---------
| 6.2.7       | Insufficient CU LDS | 79.01 | 79.01 | 79.01 | Pct    |
----------------------------------------------------------------------
```

Looking through this data we see: - Wavefront occupancy (`2.1.15`) is 3%, which is very low - Insufficient CU LDS (`6.2.7`) contains a fairly large percentage, which indicates our occupancy is currently limited by LDS allocations.

There are two solution directories, which correspond to two ways that this occupancy limit can be addressed. First, we have `solution-no-lds` , which completely removes the LDS usage. Let's build and run this solution:

```
cd solution-no-lds
make
./solution.exe
```

(*simulated output*)

```
yAx time: 70 ms
```

We see that the runtime is much better for this solution than the problem, let's see if removing LDS did indeed increase occupancy:

```
omniperf profile -n solution --no-roof -- ./solution.exe
```

(*output omitted*)

Once the profile command completes, run:

```
omniperf analyze -p workloads/solution/MI200 --dispatch 1 --block 2.1.15 6.2.7
```

The output should look something like:

```
  ___                  _                  __
 / _ \ _ __ ___  _ __ (_)_ __   ___ _ __ / _|
| | | | '_ ` _ \| '_ \| | '_ \ / _ \ '__| |_
| |_| | | | | | | | | | | |_) |  __/ |  |  _|
 \___/|_| |_| |_|_| |_|_| .__/ \___|_|  |_|
                        |_|
Analysis mode = cli
[analysis] deriving Omniperf metrics...

--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
----------------------------------------------------------------------------------------------------------------
|    | Kernel_Name                              |   Count |      Sum(ns) |     Mean(ns) |   Median(ns) |     Pct |
-----|------------------------------------------|---------|-------------|-------------|--------------|---------
|  0 | yax(double*, double*, double*, int, int, |    1.00 | 69513618.00 | 69513618.00 |  69513618.00 | 100.00 |
|    |  double*) [clone .kd]                    |         |             |             |              |        |
----------------------------------------------------------------------------------------------------------------
0.2 Dispatch List
-------------------------------------------------------------------------------------------------
|    |   Dispatch_ID | Kernel_Name                                                   |   GPU_ID |
-----|---------------|---------------------------------------------------------------|-----------
|  0 |             1 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        8 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
-----------------------------------------------------------------------------------------
| Metric_ID    | Metric              |    Avg | Unit       |    Peak |   Pct of Peak |
--------------|---------------------|--------|------------|---------|----------------
| 2.1.15       | Wavefront Occupancy | 451.15 | Wavefronts | 3328.00 |         13.56 |
-----------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
----------------------------------------------------------------------
| Metric_ID    | Metric              |   Avg |   Min |   Max | Unit    |
--------------|---------------------|-------|-------|-------|---------
| 6.2.7        | Insufficient CU LDS |  0.00 |  0.00 |  0.00 | Pct     |
----------------------------------------------------------------------
```

Looking through this data we see: - Wave occupancy (`2.1.15`) is 10% higher than in problem.cpp - Insufficient CU LDS (`6.2.7`) is now zero, indicating solution-no-lds is not occupancy limited by LDS allocations.

Can we get some runtime advantage from using smaller LDS allocations?

This is the solution implemented in the `solution` directory:

```
cd ../solution
make
./solution.exe
```

(*simulated output*)

```
yAx time: 50 ms
```

This solution, rather than removing the LDS allocation, simply reduces the amount of LDS requested to address the occupancy limit. This gives us the benefit of having some data pulled closer than it was in `solution-no-lds` which is validated through the speedup we see. But is this solution still occupancy limited by LDS?

```
omniperf profile -n solution --no-roof -- ./solution.exe
```

(*output omitted*)

Once the profile command completes, run:

```
omniperf analyze -p workloads/solution/MI200 --dispatch 1 --block 2.1.15 6.2.7
```

The output should look something like:

```
  ___                  _                  __
 / _ \ _ __ ___  _ __ (_)_ __   ___ _ __ / _|
| | | | '_ ` _ \| '_ \| | '_ \ / _ \ '__| |_
| |_| | | | | | | | | | | |_) |  __/ |  |  _|
 \___/|_| |_| |_|_| |_|_| .__/ \___|_|  |_|
                        |_|

Analysis mode = cli
[analysis] deriving Omniperf metrics...

--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
---------------------------------------------------------------------------------------------------------------
|    | Kernel_Name                              |   Count |     Sum(ns) |    Mean(ns) |   Median(ns) |    Pct |
-----|------------------------------------------|---------|-------------|-------------|--------------|--------
|  0 | yax(double*, double*, double*, int, int, |    1.00 | 51238856.00 | 51238856.00 |  51238856.00 | 100.00 |
|    |  double*) [clone .kd]                    |         |             |             |              |        |
---------------------------------------------------------------------------------------------------------------
0.2 Dispatch List
-------------------------------------------------------------------------------------------------
|    |   Dispatch_ID | Kernel_Name                                                   |   GPU_ID |
-----|---------------|---------------------------------------------------------------|-----------
|  0 |             1 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        8 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
-------------------------------------------------------------------------------------
| Metric_ID    | Metric              |    Avg | Unit       |    Peak |   Pct of Peak |
--------------|---------------------|--------|------------|---------|----------------
| 2.1.15       | Wavefront Occupancy | 494.05 | Wavefronts | 3328.00 |         14.85 |
-------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
----------------------------------------------------------------------
| Metric_ID    | Metric              |   Avg |   Min |   Max | Unit   |
--------------|---------------------|-------|-------|-------|---------
| 6.2.7        | Insufficient CU LDS |  0.00 |  0.00 |  0.00 | Pct    |
----------------------------------------------------------------------
```

Looking at this data we see: - Wave Occupancy (`2.1.15`) is even higher than before - Insufficient CU LDS (`6.2.7`) shows we are not occupancy limited by LDS allocations.

Pulling some data from global device memory to LDS can be an effective optimization strategy, if occupancy limits are carefully avoided.

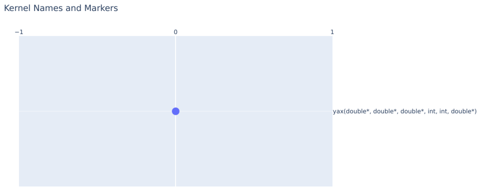
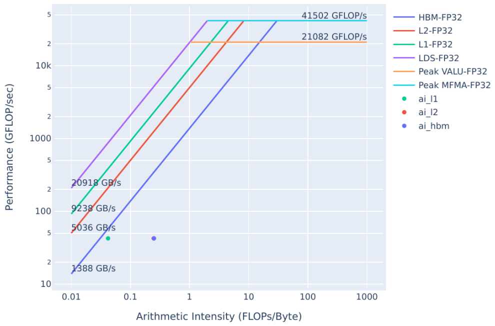
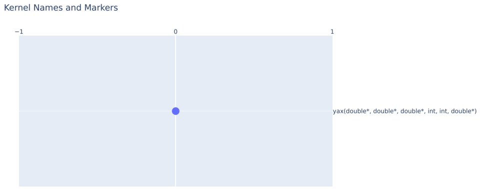
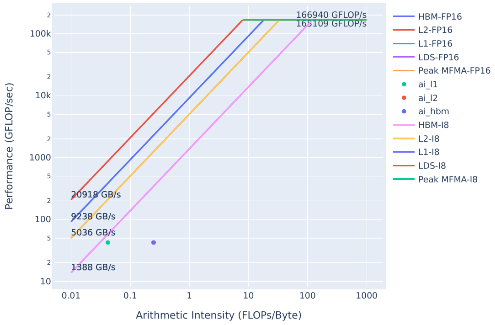
Solution Roofline

Let's take a look at the roofline for `solution` , which can be generated with:

```
omniperf profile -n solution_roof_only --roof-only -- ./solution.exe
```

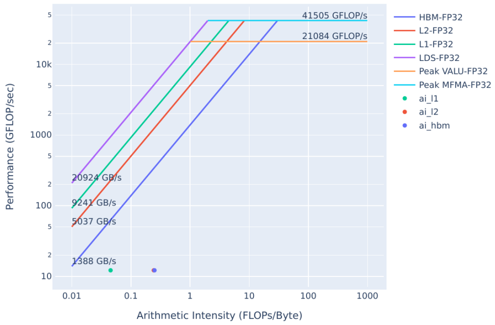
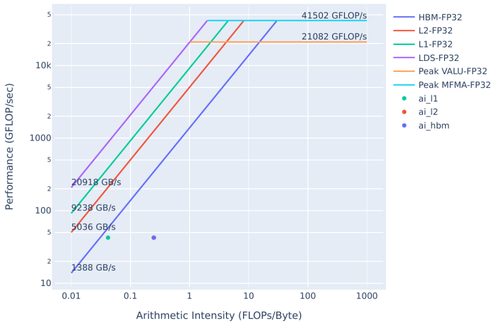
The plots will appear as PDF files in the `./workloads/problem_roof_only/MI200` directory, if generated on MI200 hardware.

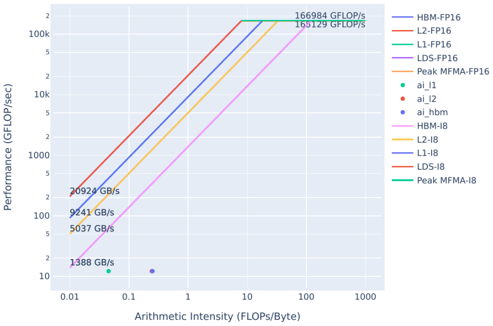
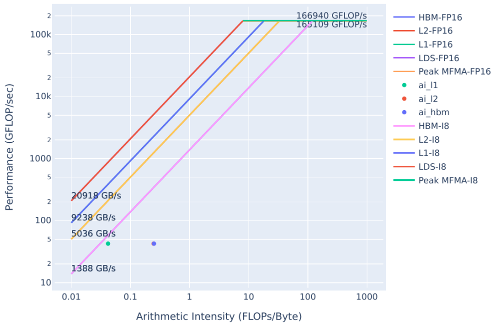
The plots are shown here:

<table>
<tr><th>Roofline Type</th><th>Roofline Legend</th><th>Roofline Plot</th></tr>
<tr><td>FP32/FP64</td><td>
</td><td>
</td></tr>
<tr><td>FP16/INT8</td><td>
</td><td>
</td></tr>
</table>

We see that there is still room to move the solution roofline up towards the bandwidth limit.

Roofline Comparison

<table>
<tr><th>Roofline Type</th><th>Roofline Legend</th><th>Roofline Plot</th></tr>
<tr><td>FP32/FP64</td><td>
</td><td>
</td></tr>
</table>

Roofline Type	Roofline Legend	Roofline Plot
FP16/INT8		

Again, we see that the solution's optimizations have resulted in the kernel moving up in the roofline, meaning the solution executes more efficiently than the problem.

Summary and Take-aways

Using LDS can be very helpful in reducing global memory reads where you have repeated use of the same data. However, large LDS allocations can also negatively impact performance by limiting the amount of wavefronts that can be resident in the device at any given time. Be wary of LDS usage, and check the SPI stats to ensure your LDS usage is not negatively impacting occupancy.

Omnitrace

NOTE: extensive documentation on how to use `omnitrace` for the `GhostExchange_Array` example is now available as `README.md` files in this exercises repo. While the testing has been done on Frontier in that documentation, most of the `omnitrace` tools apply in the same way, hence it could provide additional training matieral.

Here, we show how to use `omnitrace` tools considering the example in `HPCTrainingExamples/HIP/jacobi` .

Initial Setup

Setup environment:

```
module purge
module load omnitrace gcc/13
```

Next, create a configuration file for `omnitrace` :

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

If you do not provide a path to the config file, it will generate one in the current directory: `./omnitrace-config.cfg` . This config file contains several flags that can be modified to turn on or off several options that impact the visualization of the traces in `Perfetto` . You can see what flags can be included in the config file by doing:

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

To add brief descriptions, use the `-bd` option:

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

Note that the list of flags displayed by the commands above may not include all actual flags that can be set in the config.

You can also create a configuration file with description per option. Beware, this is quite verbose:

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

Next you have to declare that you want to use this configuration file. Note, this is only necessary if you had provided a custom path and/or filename for the config file when you created it.

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

Setup Jacobi Example

Go to the Jacobi code in the examples repo:

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

Compile the code:

```
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
```

Runtime Instrumentation

Run the code with `omnitrace-instrument` to perform runtime instrumentation: this will produce a series of directories whose name is define by the time they were crated. In one of these directories, you can find the `wall_clock-<proc_ID>.txt` file, which includes information on the function calls made in the code, such as how many times these calls have been called (`COUNT`) and the time in seconds they took in total (`SUM`):

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

The above command produces a folder called `instrumentation` that contains the `available.txt` file, which shows all the functions that can be instrumented. To instrument a specific function, include the `--function-include <fnc>` option in the `omnitrace-instrument` command, for example:

```
mpirun -np 1 omnitrace-instrument -v 1 -I 'Jacobi_t::Run' 'JacobiIteration' -- ./Jacobi_hip -g 1 1
```

The output provided by the above command will show that only those functions have been instrumented:

```
[...]
[omnitrace][exe]    1 instrumented funcs in JacobiIteration.hip
[omnitrace][exe]    1 instrumented funcs in JacobiRun.hip
[omnitrace][exe]    1 instrumented funcs in Jacobi_hip
[omnitrace][exe]    2 instrumented funcs in librocprofiler-register.so.0.3.0
[...]
```

Alternatively, you can use the `--print-available functions` option as shown below. The `--simulate` option will exit after outputting the diagnostics, the `- v` option is for verbose output:

(NOTE: the output of the next command may be lengthy, you may want to pipe it to a file using » out.txt at the end of the line to make searching it easier afterwards.)

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

Binary Rewrite

You can create an instrumented binary using `omnitrace-instrument` (notice that this doesn't take very long to run):

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

Execute the new instrumented binary using the `omnitrace-run` command inside `mpirun` . This is the recommended way to profile MPI applications as `omnitrace` will **separate the output files for each rank**:

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

To see the list of the instrumented GPU calls, make sure to turn on the `OMNITRACE_PROFILE` flag in your config file:

```
OMNITRACE_PROFILE                                  = true
```

Running the instrumented binary again, you can see that it generated a few extra files. One of those has a list of instrumented GPU calls and durations of those calls:

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

Debugging omnitrace-run

If you get errors when you run an instrumented binary or when you run with runtime instrumentation, add the following options `--monochrome -v 2 --debug` and try the following command. This would give you additional debug information to assist you in figuring out where the problem may lie:

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

Visualization

Copy the `perfetto-trace-0.proto` to your local machine, and using the Chrome browser open the web page https://ui.perfetto.dev/:

```
scp -i <path/to/ssh/key> -P <port_number> <username>@aac1.amd.com:~/<path/to/proto/file> .
```

Click `Open trace file` and select the `.proto` file. Below, you can see an example of how a `.proto` file would be visualized on `Perfetto` :

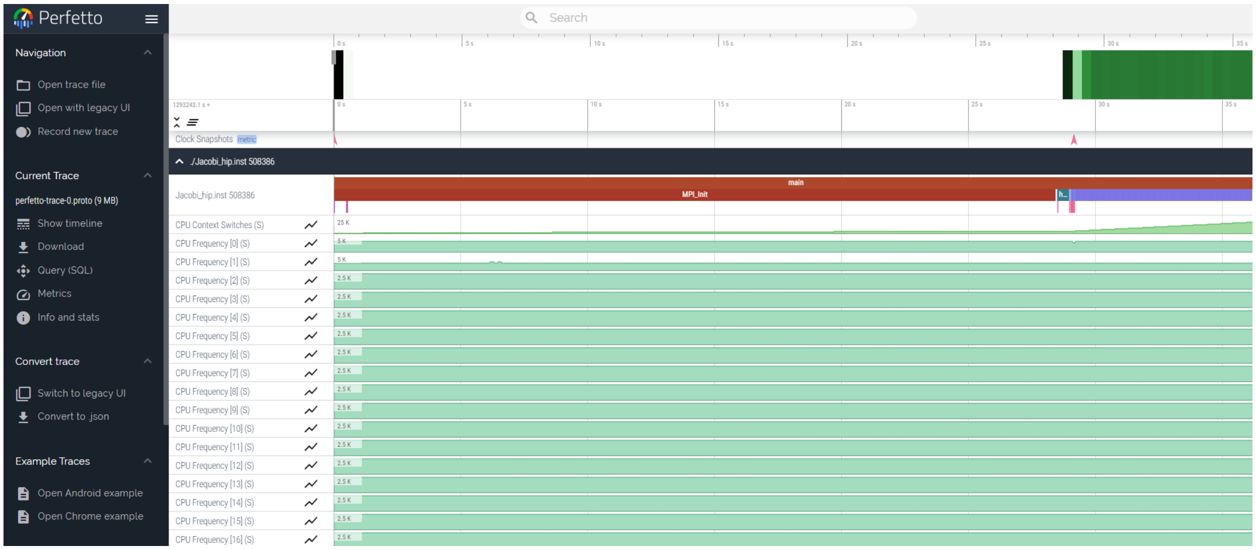


Figure 2: image

Hardware Counters

To see a list of all the counters for all the devices on the node, do:

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

Declare in your configuration file:

```
OMNITRACE_ROCM_EVENTS = VALUUtilization,FetchSize
```

Check again:

```
grep OMNITRACE_ROCM_EVENTS $OMNITRACE_CONFIG_FILE
```

Run the instrumented binary, and you will observe an output file for each hardware counter specified. You should also see a row for each hardware counter in the `Perfetto` trace generated by `omnitrace` .

Note that you do not have to instrument again after making changes to the config file. Just running the instrumented binary picks up the changes you make in the config file. Ensure that the `OMNITRACE_CONFIG_FILE` environment variable is pointing to your config file.

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

The output should show something like this:

```
...]> Outputting 'omnitrace-Jacobi_hip.inst-output/<TIMESTAMP>/rocprof-device-0-VALUUtilization-0.json'
...]> Outputting 'omnitrace-Jacobi_hip.inst-output/<TIMESTAMP>/rocprof-device-0-VALUUtilization-0.txt'
```

```
...]> Outputting 'omnitrace-Jacobi_hip.inst-output/<TIMESTAMP>/rocprof-device-0-FetchSize-0.json'
...]> Outputting 'omnitrace-Jacobi_hip.inst-output/<TIMESTAMP>/rocprof-device-0-FetchSize-0.txt'
```

If you do not want to see the details for every CPU core, modify the config file to select only what you want to see, say CPU cores 0-2 only:

```
OMNITRACE_SAMPLING_CPUS                              = 0-2
```

Now running the instrumented binary again will show significantly fewer CPU lines in the profile:

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

Profiling Multiple Ranks

Run the instrumented binary with multiple ranks. You'll find multiple `perfetto-trace-*.proto` files, one for each rank (note that depending on your system it may be necessary to do a `salloc` prior to the command below to ensure enough resources ara available):

```
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 (trace concatenation is not available in all `omnitrace` versions):

```
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_PROFILE = true
OMNITRACE_FLAT_PROFILE = true
```

Execute the code and check the `wall_clock-0.txt` file again. Instead of updating the config file, you can also set the environment variables to achieve the same effect.

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