Contents

AMD Accelerator Cloud (AAC) 2
Login Instructions 2
SSH-Key Generation 2
Login with SSH-Key 2
Login with password 2
Login Troubleshooting 3
Directories and Files 3
Container Environment 4
Explore Modules 4
Slurm Information 5
Exercise Examples 5
Training Examples Repo 5

Programming Model Exercises – Managed Memory and Single Address Space (APU) 6
CPU Code baseline 6
Standard GPU Code example 6
Managed Memory Code 7
APU Code – Single Address Space in HIP 8
OpenMP APU or single address space 8
RAJA Single Address Code 9
Kokkos Unified Address Code 10

OpenMP C Build systems: make and cmake 10
Make 11
CMake 11

OpenMP CXX Build systems: make and cmake 12
Make 12
CMake 13

OpenMP Fortran Build systems: make and cmake 13
Make 14
CMake 14

First OpenMP C offload: 15
Part 1: Unified shared memory 15
Part 2: Impact of USM 17
Part 3: Map clauses 17

First Fortran OpenMP offload: Porting saxpy step by step and explore the discrete GPU and APU programming models: 18
Part 1: Porting with unified shared memory enabled 19
Part 2: explore the impact of unified shared memory 20
Part 3: with map clauses 20

OpenMP Single Line Compute Constructs: 21
CPU version 21

OpenMP Single Line Compute Constructs: 23
CPU version 23

OpenMP complex compute constructs in C 25
Full combined compute directive 25
Target directive 26

Teams clause . . . 26
Distribute clause . . . 26
parallel for without the teams distribute clauses . . . 26
Split multi-level directive . . . 27

Reduction excercise: **27**

Porting exercise: reduction **28**
Part 2: Port with map clause . . . 28
2.1 Porting excercise . . . 28
Part 1: Fortran with interface blocks . . . 31
Part 2: Fortran with modules . . . 33
C++ member function . . . 34
C++ member function external . . . 34
C++ virtual methods . . . 35
The `usm` Sub-directory . . . 36
The `daxpy` Sub-directory . . . 36
The `operations` Sub-Directory . . . 36
The `explicit` Sub-directory . . . 36
The `daxpy` Sub-directory . . . 37

HIP/basic_examples Documentation **37**
Table of Contents . . . 37

Add the device-to-host data transfer **38**

Complete the square elements kernel **38**

Complete the matrix multiply kernel **38**

Complete the matrix multiply kernel **39**

Porting Applications to HIP **40**
Hipify Examples . . . 40
Exercise 1: Manual code conversion from CUDA to HIP (10 min) . . . 40
Exercise 2: Code conversion from CUDA to HIP using HIPify tools (10 min) . . . 40
HIPifly Example: Vector Addition . . . 41
Full OpenMP Application Code . . . 42
OpenMP Application Calling a HIP Kernel . . . 42
APU Programming Model Version . . . 42
HIP application calling an OpenMP Kernel . . . 42
OpenMP and HIP Kernels in the Same Source File . . . 42

Running a Fortran to HIP interop example **43**
Explicit Memory Management . . . 43
Unified Shared Memory . . . 44

Optimizing DAXPY HIP **44**
Inputs . . . 44
Build Code . . . 44
Run exercises . . . 44
Things to ponder about . . . 45

`daxpy_1` . . . 45
`daxpy_2` . . . 45
`daxpy_3` . . . 45
`daxpy_4` . . . 45
`daxpy_5` . . . 45
Notes . . . 46

Kokkos examples **46**
Stream Triad . . . 46
Step 1: Build a separate Kokkos package . . . 46
Step 2: Modify Build . . . 46
Step 3: Add Kokkos views for memory allocation of arrays . . . 47
Step 5: Add Kokkos timers . . . 48
6. Run and measure performance with OpenMP . . . 48
Portability Exercises . . . 48

GPU Aware MPI **48**
Point-to-point and collective . . . 48
OSU Benchmark . . . 49
Ghost Exchange example . . . 49
FP32 . . . 59

ROCgdb **59**
Saxpy Debugging . . . 59

ROCm™ Systems Profiler aka `rocprof-sys` **61**
Environment setup . . . 61
Build and run . . . 61
`rocprof-sys` config . . . 62
Instrument application binary . . . 62
Run instrumented binary . . . 63
Visualizing traces using `Perfetto` . . . 63
Additional features . . . 63
Flat profiles . . . 63
Hardware counters . . . 64
Sampling . . . 64
Profiling multiple MPI processes . . . 64
Next steps . . . 64

Stream Overlap Example **64**
Folder `0-Orig` . . . 65
Folder `1-split-copy-compute-hw-queues` . . . 65
Folder `2-pageable-mem` . . . 65
Self-guided tour of the Stream Overlap example . . . 65

OmniperfExamples **66**

Exercise 1: Launch Parameter Tuning **66**

Results on MI210 . . . 66
Initial Roofline Analysis: . . . 66
Exercise instructions: . . . 67
Omniperf Command Line Comparison Feature: . . . 69
More Kernel Filtering: . . . 70
Solution Roofline . . . 72
Roofline Comparison . . . 72
Summary and Take-aways . . . 72
Results on MI300A . . . 72
Roofline Analysis: . . . 72
Exercise Instructions: . . . 72
Omniperf Command Line Comparison Feature: . . . 74
More Kernel Filtering: . . . 75

Exercise 2: LDS Occupancy Limiter **77**
Results on MI210 . . . 77
Initial Roofline Analysis . . . 77
Exercise Instructions: . . . 77
Solution Roofline . . . 82
Roofline Comparison . . . 82
Summary and Take-aways . . . 82
Results on MI300A . . . 82
Roofline Analysis: . . . 82

Exercise 3: Register Occupancy Limiter **86**
Results on MI210 . . . 86
Initial Roofline Analysis . . . 86
Exercise Instructions: . . . 87
Solution Roofline . . . 91
Roofline Comparison . . . 91
Summary and Take-aways . . . 91
Results on MI300A . . . 91
Roofline Analysis: . . . 91

Exercise 4: Strided Data Access Patterns (and how to find them) **95**
Results on MI210 . . . 96
Initial Roofline Analysis . . . 96
Exercise Instructions: . . . 96
Solution Roofline Analysis . . . 99
Roofline Comparison . . . 99
Summary and Take-aways . . . 100
Results on MI300A . . . 100

Exercise 5: Algorithmic Optimizations **104**
Results on MI210: . . . 105
Initial Roofline Analysis . . . 105
Exercise Instructions: . . . 105
Solution Roofline Analysis . . . 108
Roofline Comparison . . . 108
Summary and Take-aways . . . 109
Results on MI300A . . . 109

AMD Accelerator Cloud (AAC)

File: login_info/AAC/README.md at https://github.com/amd/HPCTrainingExamples

To support trainings, we can upload training containers to the AMD Accelerator Cloud (AAC), and have attendees login using the instructions below. This set of instructions assumes that users have already received their `<username>` and `<port_number>` for the container, and that they have either provided an ssh key to the training team, or they have received a password from the training team.

Login Instructions

The instructions below rely on ssh to access the AAC. Remember that when a container is brought down, it will not be possible to access the user data on it, so make sure to backup your data frequently if you want to keep it.

SSH-Key Generation

Generate an ssh key on your local system, which will be stored in `.ssh` :

```
cd $HOME
ssh-keygen -t ed25519 -N ''
```

To examine the content of your public key do:

```
cat $HOME/.ssh/id_ed25519.pub
```

NOTE: at first login, you will be presented with the AAC user agreement form. This covers the terms of use of the compute hardware as well as how we will handle your data. Scroll down with the down arrow and type `yes` when prompted. Note that if you will scroll down too much, then `no` will be received as answer and you will be logged out.

Login with SSH-Key

IMPORTANT: if you are supposed to login with an ssh key and you are prompted a password, do not type any password! Instead, type `Ctrl+C` and contact us to let us know about the incident.

To login to an AAC MI300A system using the ssh key use the `<username>` and `<port_number>` that the training team has provided you, for instance:

```
ssh <username>@aac6.amd.com -i <path/to/ssh/key> -p <port_number> (1)
```

For an MI210 or MI250 system, use aac1.amd.com

```
ssh <username>@aac1.amd.com -i <path/to/ssh/key> -p <port_number> (1)
```

Login with password

For a password login, the command is the same as in `(1)` , except that it is not necessary to specify a path to the ssh key. Just type the password that has been given to you when prompted:

```
ssh <username>@aac6.amd.com -p <port_number>
```

IMPORTANT: It is fundamental to not type the wrong password more than two times otherwise your I.P. address will be blacklisted and you will not be allowed access to AAC until we modify our firewall to get you back in. This is especially important if you are at an event where all the attendees are connecting to the same wireless network.

If you are using a password login, you can upload an ssh key with the following command to avoid using a password

```
ssh-copy-id -i <path/to/ssh/key.pub> -p <port_number> -o UpdateHostKeys=yes <username>@aac6.amd.com
```

In the commands above, `-p` refers to the port number and `-i` points to the path of your ssh key. The `-i` option is not needed if your default key is used. Note that different port numbers will be associated with different containers on the AAC, and anytime a container is brought up, the port number will change in general.

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

```
# AMD AAC cluster
Host aac
   User <username>
   Hostname aac6.amd.com // this may be different depending on the container
   IdentityFile <path/to/ssh/key> // this points to the private key file
   Port <port_number>
   ServerAliveInterval 600
   ServerAliveCountMax 30
```

The `ServerAlive*` lines in the config file may be added to avoid timeouts when idle. You can then login using:

```
ssh aac -p <port_number>
```

It may also happen that a message like the following will show after logging into AAC:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
```

In such a case, remove in your local system the offending keys located in `.ssh/known_hosts`, as indicated by the warning message.

Login Troubleshooting

Here are some troubleshooting tips if you cannot login to AAC following the instructions above:

1. Check the spelling of the command ssh, in particular `<username>`, `<port_number>` and password.
2. Turn off VPN if on.
3. Try logging in from a different machine if available (and migrate the ssh key to the new machine or generate a new one and send it to us).
4. Try a ***jump host***: this is a local server that you ssh to and then do a second ssh command from there.

In case none of these options work, send us the output of the ssh command followed by `-vv` and also the output of `traceroute aac6.amd.com`. Additionally, let us know if the command `ping aac1.amd.com` works on your end.

Directories and Files

Persistent storage is at `/home/aac/shared/teams/hackathon-testing/<group>/<username>`. Your home directory will be set to this directory:

```
$HOME=/home/aac/shared/teams/hackathon-testing/<group>/<username>`
```

Files in the above directory will persist across container starts and stops and even be available from another container with the same `<username>` on systems at the same hosting location. Remember that it will not be possible to retrieve your data once the container has been brought down.

You can copy files in or out of AAC with the `scp` or the `rsync` command.

Copy into AAC from your local system, for instance:

```
scp -i <path/to/ssh/key> -P <port_number> <file> <username>@aac6.amd.com:~/<path/to/file>
```

Copy from AAC to your local system:

```
scp -i <path/to/ssh/key> -P <port_number> <username>@aac6.amd.com:~/<path/to/file> .
```

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

```
rsync -avz -e "ssh -i <path/to/ssh/key> -p <port_number>" <file> <username>@aac6.amd.com:~/<path/to/fil
```

Container Environment

Please consult the container's README to learn about the latest specs of the training container.

The container is based on the Ubuntu 22.04 Operating System with the latest version of the ROCm software stack. It contains multiple versions of AMD, GCC, and LLVM compilers, hip libraries, GPU-Aware MPI, AMD profiling tools and HPC community tools. The container also has modules set up with the lua modules package and a slurm package and configuration. It includes the following additional packages:

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

Explore Modules

To see what modules are available do:

```
module avail
```

The output list of `module avail` should show:

```
----------------------------------- /etc/lmod/modules/Linux ------------------------------------
  clang/base      gcc/base      miniconda3/24.9.2    miniforge3/24.9.0

------------------------------------ /etc/lmod/modules/ROCm ------------------------------------
  amdclang/18.0.0-6.4.0         opencl/6.4.0                      rocprofiler-systems/6.4.0 (D)
  amdflang-new/rocm-afar-6.0.0  rocm/6.4.0
  hipfort/6.4.0                 rocprofiler-compute/6.4.0 (D)
```

```
------------------------------- /etc/lmod/modules/ROCmPlus-MPI -------------------------------
  mpi4py/4.0.3    openmpi/5.0.7-ucc1.3.0-ucx1.18.0

------------------------ /etc/lmod/modules/ROCmPlus-AMDResearchTools -------------------------
  rocprofiler-compute/develop    rocprofiler-systems/amd-staging

------------------------- /etc/lmod/modules/ROCmPlus-LatestCompilers -------------------------
  hipfort_from_source/6.4.0

------------------------------- /etc/lmod/modules/ROCmPlus-AI --------------------------------
  cupy/14.0.0a1    jax/0.4.35    pytorch/2.7.0

----------------------------------- /etc/lmod/modules/misc -----------------------------------
  fftw/3.3.10  hpctoolkit/2024.11.27dev  netcdf-c/4.9.3-rc1        scorep/9.0-dev
  hdf5/1.14.5  hypre/2.33.0              netcdf-fortran/4.6.2-rc1  tau/dev
  hipifly/dev  kokkos/4.6.00             petsc/3.23.0

 Where:
  D:  Default Module
```

There are several modules associated with each ROCm version. One is the rocm module which is needed by many of the other modules. The second is the amdclang module when using the amdclang compiler that comes bundled with ROCm. The third is the hipfort module for the Fortran interfaces to HIP. Also, there is an OpenCL module and one for each of the AMD profilers.

Compiler modules set the C, CXX, FC flags. Only one compiler module can be loaded at a time. hipcc is in the path when the rocm module is loaded. Note that there are several modules that set the compiler flags and that they set the full path to the compilers to avoid path problems.

Slurm Information

The training container comes equipped with Slurm. Slurm configuration is for a single queue that is shared with the rest of the node. Run the following command to get info on Slurm:

```
sinfo
```

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

The Slurm `salloc` command may be used to acquire a long term session that exclusively grants access to one or more GPUs. Alternatively, the `srun` or `sbatch` commands may be used to acquire a session with one or more GPUs and only exclusively use the session for the life of the run of an application. `squeue` will show information on who is currently running jobs.

Exercise Examples

The exercise examples are preloaded into the `/Shared` directory. Copy the files into your home directory with:

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

Training Examples Repo

Alternatively, you can get the examples from our repo. This repo contains all the code that we normally use during our training events:

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

Programming Model Exercises – Managed Memory and Single Address Space (APU)

From `HPCTrainingExamples/ManagedMemory/README.md` in the training exercises repository

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.

The source code for these exercises is based on those in the presentation, but with details filled in so that there is a working code. You may want to examine the code in these exercises and compare it to the code in the presentation and to the code in the other exercises.

CPU Code baseline

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

First, run the standard CPU version. This is a working version of the original CPU code from the programming model presentation. The example will work with any C compiler and run on any CPU. To set up the environment, we need to set the CC environment variable to the C compiler executable. We do this by loading the amdclang module which sets `CC=/opt/rocm-<version>/llvm/bin/amdclang`. The makefile uses the CC environment which we have set. In our modules, we set the "family" to compiler so that only one compiler can be loaded at a time.

```
cd HPCTrainingExamples/ManagedMemory/CPU_Code
module load amdclang
make
```

will compile with `/opt/rocm-<version>/llvm/bin/amdclang -g -O3 cpu_code.c -o cpu_code` Then run code with

```
./cpu_code
```

Standard GPU Code example

This example shows the standard GPU explicit memory management. For this case, we must move the memory ourselves. This example will run on any AMD Instinct GPU (data center GPUs) and most workstation or desktop discrete GPUs and APUs. The AMD GPU driver and ROCm software needs to be installed.

For the environment setup, we need the ROCm bin directory added to the path. We do this by loading the ROCm module with `module load rocm`. This will set the path to the rocm bin directory. We could also do this with export `PATH=/opt/rocm-<version>/bin` or by supplying the full path `/opt/rocm-<version>/bin/hipcc` to the compile line. Note that even this may not be necessary as the ROCm install may have placed a link to `hipcc` in `/usr/bin/hipcc` during the ROCm install.

We also supply a `--offload-arch=${AMDGPU_GFXMODEL}` option to the compile line. While not necessarily required, it helps in cases where the architecture is not autodetected properly. We use the following line to query what the architecture string `AMDGPU_GFXMODEL` should be. We can also set our own `AMDGPU_GFXMODEL` variable in cases where we want to cross-compile or compile for more than one architecture.

```
AMDGPU_GFXMODEL ?= $(strip $(shell rocminfo |grep -m 1 -E gfx[^0]{1} | sed -e 's/ *Name: *//'))
```

The `AMDGPU_GFXMODEL` architecture string is `gfx90a` for MI200 series and `gfx942` for MI300A and MI300X. We can also compile for more than one architecture with `export AMDGPU_GFXMODEL="gfx90a,gfx942"` .

```
cd ../GPU_Code
make
```

This will compile with `hipcc -g -O3 --offload-arch=${AMDGPU_GFXMODEL} gpu_code.hip -o gpu_code` .

Then run the code with

```
./gpu_code
```

Managed Memory Code

In this example, we will set the `HSA_XNACK` environment variable to 1 and let the Operating System move the memory for us. This will run on AMD Instinct GPUs for the data center including MI300X, MI300A, and MI200 series. To set up the environment, `module load rocm` .

```
export HSA_XNACK=1
module load rocm
cd ../Managed_Memory_Code
make
./gpu_code
```

To understand the difference between the explicit memory management programming and the managed memory, let's compare the two codes.

```
diff gpu_code.hip ../GPU_Code/
```

You should see the following:

```
34a35,37
>    double *in_d, *out_d;
>    HIP_CHECK(hipMalloc((void **)&in_d, Msize));
>    HIP_CHECK(hipMalloc((void **)&out_d, Msize));
38a42,43
>    HIP_CHECK(hipMemcpy(in_d, in_h, Msize, hipMemcpyHostToDevice));
>
41c46
<    gpu_func<<<grid,block,0,0>>>(in_h, out_h, M);
---
>    gpu_func<<<grid,block,0,0>>>(in_d, out_d, M);
43a49
>    HIP_CHECK(hipMemcpy(out_h, out_d, Msize, hipMemcpyDeviceToHost));
```

It may be more instructive to look at the lines of hip code that are required compared to the explicit memory management GPU code.

```
grep hip ../GPU_Code/gpu_code.hip
```

which gets the following output

```
#include "hip/hip_runtime.h"
    hipError_t gpuErr = call;                           \
    if(hipSuccess != gpuErr){                           \
         __FILE__, __LINE__, hipGetErrorString(gpuErr)); \
   HIP_CHECK(hipMalloc((void **)&in_d, Msize));
   HIP_CHECK(hipMalloc((void **)&out_d, Msize));
   HIP_CHECK(hipMemcpy(in_d, in_h, Msize, hipMemcpyHostToDevice));
   HIP_CHECK(hipDeviceSynchronize());
   HIP_CHECK(hipMemcpy(out_h, out_d, Msize, hipMemcpyDeviceToHost));
```

```
grep hip gpu_code.hip
```

And for the managed memory program, we essentially get just the addition of the `hipDeviceSynchronize` call plus including the hip runtime header and the error checking macro.

```
#include "hip/hip_runtime.h"
    hipError_t gpuErr = call;                           \
    if(hipSuccess != gpuErr){                           \
         __FILE__, __LINE__, hipGetErrorString(gpuErr)); \
   HIP_CHECK(hipDeviceSynchronize());
```

APU Code – Single Address Space in HIP

We'll run the same code as we used in the managed memory example. Because the memory pointers are addressable on both the CPU and the GPU, no memory management is necessary. First, log onto an MI300A node. Then compile and run the code as follows.

```
export HSA_XNACK=1
module load rocm
cd ../APU_Code
make
./gpu_code
```

It may be confusing why we need `HSA_XNACK=1`. Even with the APU, we need to map the pointers into the GPU page map though the memory itself does not need to be copied.

OpenMP APU or single address space

For this example, we have a simple code with the loop offloading in the main code, `openmp_code`, and a second version, `openmp_code1`, with the offloaded loop in a subroutine where the compiler cannot tell the size of the array. Running this on the MI200 series, it passes, despite that it does not have a single address space. We add `export LIBOMPTARGET_INFO=-1` or for less output `export LIBOMPTARGET_INFO=$((0x1 | 0x10))` to verify that it is running on the GPU.

```
export HSA_XNACK=1
module load amdclang
cd ../OpenMP_Code
make
```

You should see some warnings that are basically telling you the AMD clang compiler is ignoring the `simd` clause is being ignored. You can remove the `simd` from the OpenMP pragmas, but at the expense of portability to some other OpenMP compilers. Now run the code.

```
./openmp_code
./openmp_code1
export LIBOMPTARGET_INFO=$((0x1 | 0x10)) # or export LIBOMPTARGET_INFO=-1
./openmp_code
./openmp_code1
```

If the executable is running on the GPU you will see some output as a result of the `LIBOMPTARGET_INFO` environment variable being set. If it is not running on the GPU, you will not see anything.

For more experimentation with this example, comment out the first line of the two source codes.

```
//#pragma omp requires unified_shared_memory
make
export LIBOMPTARGET_INFO=-1
./openmp_code
./openmp_code1
```

Now with the `LIBOMPTARGET_INFO` variable set, we get a report that memory is being copied to the device and back. The OpenMP compiler is helping out a lot more than might be expected even without an APU.

RAJA Single Address Code

First, set up the environment

```
module load amdclang
module load rocm
```

For the Raja example, we need to build the Raja code first

```
cd ~/HPCTrainingExamples/ManagedMemory/Raja_Code

PWDir=`pwd`

git clone --recursive https://github.com/LLNL/RAJA.git Raja_build
cd Raja_build

mkdir build_hip && cd build_hip

cmake -DCMAKE_INSTALL_PREFIX=${PWDir}/Raja_HIP \
      -DROCM_ROOT_DIR=/opt/rocm \
      -DHIP_ROOT_DIR=/opt/rocm \
      -DHIP_PATH=/opt/rocm/bin \
      -DENABLE_TESTS=Off \
      -DENABLE_EXAMPLES=Off \
      -DRAJA_ENABLE_EXERCISES=Off \
      -DENABLE_HIP=On \
      ..

make -j 8
make install

cd ../..

rm -rf Raja_build

export Raja_DIR=${PWDir}/Raja_HIP
```

Now we build the example. Note that we just allocated the arrays on the host with malloc. To run it on the MI200 series, we need to set the `HSA_XNACK` variable.

```
# To run with managed memory
export HSA_XNACK=1

mkdir build && cd build
CXX=hipcc cmake ..
make
./raja_code

cd ..
rm -rf build

cd ${PWDir}
rm -rf Raja_HIP

cd ..
rm -rf ${PROB_NAME}
```

Kokkos Unified Address Code

First, set up the environment

```
module load amdclang
module load rocm
```

For the Kokkos example, we also need to build the Kokkos code first

```
cd ~/HPCTrainingExamples/ManagedMemory/Kokkos_Code

PWDir=`pwd`

git clone https://github.com/kokkos/kokkos Kokkos_build
cd Kokkos_build

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

make -j 8
make install

cd ../..

rm -rf Kokkos_build

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

Now we build the example. Note that we have not had to declare the arrays in Kokkos Views.

```
# To run with managed memory
export HSA_XNACK=1

mkdir build && cd build
CXX=hipcc cmake ..
make
./kokkos_code

cd ${PWDir}
rm -rf Kokkos_HIP

cd ..
rm -rf ${PROB_NAME}
```

With recent versions of Kokkos, there is support for a single memory copy for the MI300A GPU.

```
-Dkokkos_ENABLE_IMPL_HIP_UNIFIED_MEMORY=ON in Kokkos 4.4+
```

Makes it easy to switch between host/device duplicate arrays to single memory copy on the MI300A.

OpenMP C Build systems: make and cmake

README.md in `HPCTrainingExamples/Pragma_Examples/OpenMP/C/BuildExamples` of the Training Examples repository

Build systems for make and cmake are an important starting step to working with OpenMP. We'll start with samples for C builds. We'll test them with some of our sample code to make sure your system is setup properly.

Make

```
cd HPCTrainingExamples/Pragma_Examples/OpenMP/C/BuildExamples
```

First let's take a look at the makefile

```
cat Makefile
```

The output should be

```
all: openmp_code

ROCM_GPU ?= $(strip $(shell rocminfo |grep -m 1 -E gfx[^0]{1} | sed -e 's/ *Name: *//'))

CC1=$(notdir $(CC))

ifneq ($(findstring amdclang,$(CC1)),)
  OPENMP_FLAGS = -fopenmp --offload-arch=${ROCM_GPU}
else ifneq ($(findstring clang,$(CC1)),)
  OPENMP_FLAGS = -fopenmp --offload-arch=${ROCM_GPU}
else ifneq ($(findstring gcc,$(CC1)),)
  OPENMP_FLAGS = -fopenmp -foffload=-march=${ROCM_GPU}
else ifneq ($(findstring CC,$(CC1)),)
  OPENMP_FLAGS = -fopenmp
endif

CFLAGS = -g -O3 -fstrict-aliasing ${OPENMP_FLAGS}
LDFLAGS = ${OPENMP_FLAGS} -fno-lto -lm

openmp_code: openmp_code.o
    $(CC) $(LDFLAGS) $^ -o $@

# Cleanup
clean:
    rm -f *.o openmp_code
    rm -rf build
```

```
module load amdclang
make
```

Now run the executable

```
./openmp_code
```

CMake

Looking at the CMakeLists.txt

```
cat CMakeLists.txt
```

The output should be

```
cmake_minimum_required(VERSION 3.21 FATAL_ERROR)
project(Memory_pragmas LANGUAGES C)

if (NOT CMAKE_BUILD_TYPE)
   set(CMAKE_BUILD_TYPE RelWithDebInfo)
endif(NOT CMAKE_BUILD_TYPE)

execute_process(COMMAND rocminfo COMMAND grep -m 1 -E gfx[^0]{1} COMMAND sed -e "s/ *Name: *//" OUTPUT_STRIP_TRAILI

string(REPLACE -O2 -O3 CMAKE_C_FLAGS_RELWITHDEBINFO ${CMAKE_C_FLAGS_RELWITHDEBINFO})
set(CMAKE_C_FLAGS_DEBUG "-ggdb")
```

```
set(CMAKE_C_FLAGS "-fstrict-aliasing -faligned-allocation -fnew-alignment=256")
if ("${CMAKE_C_COMPILER_ID}" STREQUAL "Clang")
   set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fopenmp --offload-arch=${ROCM_GPU}")
elseif ("${CMAKE_C_COMPILER_ID}" STREQUAL "GNU")
   set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fopenmp -foffload=-march=${ROCM_GPU}")
elseif (CMAKE_C_COMPILER_ID MATCHES "Cray")
   set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fopenmp")
   #the cray compiler decides the offload-arch by loading appropriate modules
   #module load craype-accel-amd-gfx942 for example
endif()

add_executable(openmp_code openmp_code.c)
```

```
module load amdclang
mkdir build && cd build && cmake ..
make
```

Now run the executable

```
./openmp_code
```

OpenMP CXX Build systems: make and cmake

README.md in `HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/BuildExamples` of the Training Examples repository

Build systems for make and cmake are an important starting step to working with OpenMP. We'll show samples for CXX builds. We'll test them with some of our sample code to make sure your system is setup properly.

Make

```
cd ../../CXX/BuildExamples
```

First let's take a look at the makefile

```
cat Makefile
```

The output should be

```
all: openmp_code

ROCM_GPU ?= $(strip $(shell rocminfo |grep -m 1 -E gfx[^0]{1} | sed -e 's/ *Name: *//'))

CXX1=$(notdir $(CXX))

ifneq ($(findstring amdclang,$(CXX1)),)
  OPENMP_FLAGS = -fopenmp --offload-arch=${ROCM_GPU}
else ifneq ($(findstring clang,$(CXX1)),)
  OPENMP_FLAGS = -fopenmp --offload-arch=${ROCM_GPU}
else ifneq ($(findstring gcc,$(CXX1)),)
  OPENMP_FLAGS = -fopenmp -foffload=-march=${ROCM_GPU}
else ifneq ($(findstring CC,$(CXX1)),)
  OPENMP_FLAGS = -fopenmp
endif

CXXFLAGS = -g -O3 -fstrict-aliasing ${OPENMP_FLAGS}
LDFLAGS = ${OPENMP_FLAGS} -fno-lto -lm

openmp_code: openmp_code.o
```

```
    $(CXX) $(LDFLAGS) $^ -o $@

# Cleanup
clean:
    rm -f *.o openmp_code
    rm -rf build
```

```
module load amdclang
make
```

Now run the executable

```
./openmp_code
```

CMake

Looking at the CMakeLists.txt

```
cat CMakeLists.txt
```

The output should be

```
cmake_minimum_required(VERSION 3.21 FATAL_ERROR)
project(Memory_pragmas LANGUAGES CXX)

set (CMAKE_CXX_STANDARD 17)

if (NOT CMAKE_BUILD_TYPE)
   set(CMAKE_BUILD_TYPE RelWithDebInfo)
endif(NOT CMAKE_BUILD_TYPE)

execute_process(COMMAND rocminfo COMMAND grep -m 1 -E gfx[^0]{1} COMMAND sed -e "s/ *Name: *//" OUTPUT_STRIP_TRAILI

string(REPLACE -O2 -O3 CMAKE_CXX_FLAGS_RELWITHDEBINFO ${CMAKE_CXX_FLAGS_RELWITHDEBINFO})
set(CMAKE_CXX_FLAGS_DEBUG "-ggdb")
set(CMAKE_CXX_FLAGS "-fstrict-aliasing -faligned-allocation -fnew-alignment=256")
if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fopenmp --offload-arch=${ROCM_GPU}")
elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU")
   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fopenmp -foffload=-march=${ROCM_GPU}")
elseif (CMAKE_CXX_COMPILER_ID MATCHES "Cray")
   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fopenmp")
   #the cray compiler decides the offload-arch by loading appropriate modules
   #module load craype-accel-amd-gfx942 for example
endif()

add_executable(openmp_code openmp_code.cc)
```

```
module load amdclang
mkdir build && cd build && cmake ..
make
```

Now run the executable

```
./openmp_code
```

OpenMP Fortran Build systems: make and cmake

README.md in `HPCTrainingExamples/Pragma_Examples/OpenMP/Fortran/BuildExamples` of the Training Examples repository

Build systems for make and cmake are an important starting step to working with OpenMP. We'll show samples for Fortran builds. We'll test them with some of our sample code to make sure your system is setup properly.

Make

```
cd ../../Fortran/BuildExamples
```

First let's take a look at the makefile

```
cat Makefile
```

The output should be

```
all:openmp_code

ROCM_GPU ?= $(strip $(shell rocminfo |grep -m 1 -E gfx[^0]{1} | sed -e 's/ *Name: *//'))

FC1=$(notdir $(FC))

ifneq ($(findstring amdflang, $(FC1)),)
  OPENMP_FLAGS = -fopenmp --offload-arch=${ROCM_GPU}
  FREE_FORM_FLAG = -ffree-form
else ifneq ($(findstring flang, $(FC1)),)
  OPENMP_FLAGS = -fopenmp --offload-arch=${ROCM_GPU}
  FREE_FORM_FLAG = -Mfreeform
else ifneq ($(findstring gfortran,$(FC1)),)
  OPENMP_FLAGS = -fopenmp --offload=-march=$(ROCM_GPU)
  FREE_FORM_FLAG = -ffree-form
else ifneq ($(findstring ftn,$(FC1)),)
  OPENMP_FLAGS = -fopenmp
endif

FFLAGS = -g -O3 ${FREE_FORM_FLAG} ${OPENMP_FLAGS}
ifeq (${FC1},gfortran-13)
  LDFLAGS = ${OPENMP_FLAGS} -fno-lto
else
  LDFLAGS = ${OPENMP_FLAGS}
endif

openmp_code.o: openmp_code.F90
    $(FC) -c $(FFLAGS) $^

openmp_code: openmp_code.o
    $(FC) $(LDFLAGS) $^ -o $@

# Cleanup
clean:
    rm -f *.o openmp_code *.mod
    rm -rf build
```

```
module load amdflang-new
make
```

Now run the executable

```
./openmp_code
```

CMake

Looking at the CMakeLists.txt

```
cat CMakeLists.txt
```

The output should be

```
cmake_minimum_required(VERSION 3.21 FATAL_ERROR)
project(Memory_pragmas LANGUAGES Fortran)

if (NOT CMAKE_BUILD_TYPE)
   set(CMAKE_BUILD_TYPE RelWithDebInfo)
endif(NOT CMAKE_BUILD_TYPE)

execute_process(COMMAND rocminfo COMMAND grep -m 1 -E gfx[^0]{1} COMMAND sed -e "s/ *Name: *//" OUTPUT_STRIP_TRAILI

string(REPLACE -O2 -O3 CMAKE_Fortran_FLAGS_RELWITHDEBINFO ${CMAKE_Fortran_FLAGS_RELWITHDEBINFO})
set(CMAKE_Fortran_FLAGS_DEBUG "-ggdb")
message(${CMAKE_Fortran_COMPILER_ID})
if ("${CMAKE_Fortran_COMPILER_ID}" STREQUAL "Clang")
   set(CMAKE_Fortran_FLAGS "${CMAKE_Fortran_FLAGS} -fopenmp --offload-arch=${ROCM_GPU}")
elseif ("${CMAKE_Fortran_COMPILER_ID}" STREQUAL "GNU")
   set(CMAKE_Fortran_FLAGS "${CMAKE_Fortran_FLAGS} -fopenmp -foffload=-march=${ROCM_GPU}")
elseif (CMAKE_Fortran_COMPILER_ID MATCHES "Cray")
   set(CMAKE_Fortran_FLAGS "${CMAKE_Fortran_FLAGS} -fopenmp")
   #the cray compiler decides the offload-arch by loading appropriate modules
   #module load craype-accel-amd-gfx942 for example
endif()

add_executable(openmp_code openmp_code.F90)
```

```
module load amdflang-new
mkdir build && cd build && cmake ..
make
```

Now run the executable

```
./openmp_code
```

First OpenMP C offload:

This README.md is at HPCTrainingExamples/Pragma_Examples/OpenMP/C/1_saxpy/README.md

Porting of saxpy step by step and explore the discrete GPU and APU programming models:

- Part 1: Unified shared memory after Part 1 you may want to explore the excercises 2-5 first with usm before you come to explore the behavior without USM.
- Part 2: Explore differences of `HSA_XNACK=0` and `1`
- Part 3: Map clauses

This excercise will show in a step by step solution how to port a your first kernels.

Part 1: Unified shared memory

For now, set

```
export HSA_XNACK=1
```

to make use of the APU programming model (unified memory).

There are 6 different enumerated folders. (Reccomendation: `vimdiff saxpy.cpp ../<X_saxpy_version>/saxpy.cpp` may help you to see the differences):

0) the serial CPU code.

```
cd 0_saxpy_serial_portyourself
```

Try to port this example yourself. If you are stuck, use the step by step solution in folders 1-6 and read the instructions for those excersices below. Recommendation for your first port: use `#pragma omp requires unified_shared memory` and `export HSA_XNACK=1` (before running) that you do not have to worry about map clauses. Steps 1-3 of the solution assume unified shared memory. Map clauses and investigating the behaviour of `export HSA_XNACK=0` or `=1` is added in the later steps.

- Compile the serial version. Note that `-fopenmp` is required as omp_get_wtime is used to time the loop execution.

```
amdclang++ -fopenmp saxpy.cpp -o saxpy
```

or with the cray environment (aac7):

```
CC -fopenmp saxpy.cpp -o saxpy
```

- Run the serial version.

```
./saxpy
```

Note: you can also use the Makefile.

```
make
```

instead of compiling manually.

You can now try to port the serial CPU version to the GPU

```
vi saxpy.cpp
```

and don't forget to port the Makefile (Hint: What has to be added to compile for the GPU? Note: for cray compilers)

```
vi Makefile
```

or follow the step by step solution: #### 1) Move the computation to the device

```
cd ../1_saxpy_omptarget
```

```
vi saxpy.cpp
```

add `#pragma omp target` to move the loop in the saxpy subroutine to the device. - Compile this first GPU version. Make sure you add `--offload-arch=gfx942` (on MI300A, find out what your system's gfx… is with `rocminfo`)

```
amdclang++ -fopenmp --offload-arch=gfx942 saxpy.cpp -o saxpy
```

(or use the Makefile) - Run

```
./saxpy
```

The observed time is much larger than for the CPU version. More parallelism is required!

2) Add parallelism

```
cd ../2_saxpy_teamsdistribute
vi saxpy.cpp
```

add "teams distribute" - Compile again

```
amdclang++ -fopenmp --offload-arch=gfx942 saxpy.cpp -o saxpy
```

- run again

```
./saxpy
```

The observed time is a bit better than in case 1 but still not the full parallelism is used.

3) Add multi-level parallelism

```
cd ../3_saxpy_parallelforsimd
vi saxpy.cpp
```

add "parallel for" for more parellelism - Compile again

```
amdclang++ -fopenmp --offload-arch=gfx942 saxpy.cpp -o saxpy
```

- run again

```
./saxpy
```

The observed time is much better than all previous versions. Note that the initialization kernel is a warm-up kernel here. If we do not have a warm-up kernel, the observed performance would be significantly worse. Hence the benefit of the accelerator is usually seen only after the first kernel. You can try this by commenting the !$omp target… in the initialize subroutine, then the meassured kernel is the first which touches the arrays used in the kernel.

Reccomendation: After Part 1 you may want to explore the excercises 2-5 first with usm before you come to explore the behavior without USM.

Part 2: Impact of USM

4) Explore impact of unified memory:

```
cd ../4_saxpy_nousm
vi saxpy.cpp
```

The `#pragma omp requires...` line is removed. - Compile again

```
amdclang++ -fopenmp --offload-arch=gfx942 saxpy.cpp -o saxpy
```

- run again

```
./saxpy
```

so far we worked with unfied shared memory and the APU programming model. This allows good performance on MI300A, but not on discrete GPUs. In case you will work on discrete GPUs or want to write portable code for both discrete GPUs and APUs, you have to focus on data management, too.

```
export HSA_XNACK=0
```

to get similar behaviour like on discrete GPUs (with memory copies). Compiling and running this version without any map clauses will result in much worse performance than with unified shared memory and `HSA_XNACK=1` (no memory copies on MI300A).

Part 3: Map clauses

5) map clauses this version introduces map clauses for each kernel.

```
cd ../5_saxpy_map
vi saxpy.cpp
```

see where the map clasues where added. The x vector only has to be maped "to". - Compile again

```
amdclang++ -fopenmp --offload-arch=gfx942 saxpy.cpp -o saxpy
```

- run again

```
./saxpy
```

The performance is not much better than version 4.

6) unstructured data region with enter and exit data clauses the memory is only moved once at the beginning the time to solution should be roughly in the order of magnitude of the unified shared memory version, but still slightly slower as the memory is copied like on discrete GPUs. Test yourself:

```
cd ../6_saxpy_targetdata
```

```
vi saxpy.cpp
```

- Compile again

```
amdclang++ -fopenmp --offload-arch=gfx942 saxpy.cpp -o saxpy
```

- run again

```
./saxpy
```

Additional excercise: What happens to the result, if you comment the `omp target update` ?

```
vi saxpy.cpp
```

Don't forget to recompile after commenting it.

```
amdclang++ -fopenmp --offload-arch=gfx942 saxpy.cpp -o saxpy
```

The results will be wrong! This shows, that proper validation of results is crutial when porting! Before you port a large app, think about your validation strategy before you start. Incremental testing is essential to capture such errors like missing data movement.

Note that this version uses the `new` allocator with an alignment of 128 instead of malloc to control the memory alignment. This is beneficial for improved performance.

7) parameter tuning experiment with num_teams

```
cd ../7_saxpy_numteams
vi saxpy.cpp
```

specify num_teams(...) choose a number of teams you want to test - Compile again

```
amdclang++ -fopenmp --offload-arch=gfx942 saxpy.cpp -o saxpy
```

- run again

```
./saxpy
```

investigating different numbers of teams you will find that the compiler default (without setting this) was already leading to good performance. Tuning e.g. `num_teams` or `thread_limit` may be required for some kernels, but the defaults are chosen quite well for saxpy. saxpy is a very simple kernel, this finding may differ for very complex kernels.

First Fortran OpenMP offload: Porting saxpy step by step and explore the discrete GPU and APU programming models:

This is HPCTrainingExamples/Pragma_Examples/OpenMP/Fortran/1_saxpy/README.md in the training examples repository.

This excercise will show in a step by step solution how to port a your first kernels. This simple example will not use a Makefile to practice how to compile for the GPU or APU. All following excercises will use a Makefile.

There are 6 different enumerated folders. (Reccomendation: `vimdiff saxpy.f90 ../<X_saxpy_version>/saxpy.f90` may help you to see the differences):

First, prepare the environment (load modules, set environment variables), if you didn't do so before.

Part 1: Porting with unified shared memory enabled

For now, set

```
export HSA_XNACK=1
```

and load the amdflang-new compiler module

```
module load amdflang-new
```

to make use of the APU programming model (unified memory). 0) the serial CPU code.

```
cd 0_saxpy_serial_portyourself
```

Try to port this example yourself. If you are stuck, use the step by step solution in folders 1-6 and read the instructions for those excersices below. Recommendation for your first port: use `!$omp requires unified_shared memory` (in the code after `implicit none` in each module) and `export HSA_XNACK=1` (before running) that you do not have to worry about map clauses. Steps 1-3 of the solution assume unified shared memory. Map clauses and investigating the behaviour of `export HSA_XNACK=0` or `=1` is added in the later steps.

- Compile the serial version. Note that `-fopenmp` is required as omp_get_wtime is used to time the loop execution.

```
amdflang -fopenmp saxpy.F90 -o saxpy
```

- Run the serial version.

```
./saxpy
```

You can now try to port the serial CPU version to the GPU or follow the step by step solution: 1) Move the computation to the device

```
cd ../1_saxpy_omptarget
```

```
vi saxpy.f90
```

add `!$omp target` to move the loop in the saxpy subroutine to the device. - Compile this first GPU version. Make sure you add `--offload-arch=gfx942` (on MI300A, find out what your system's gfx… is with `rocminfo`) on aac6 or aac7 with amdflang-new:

```
amdflang -fopenmp --offload-arch=gfx942 saxpy.F90 -o saxpy
```

or on on aac7 only with ftn:

First, make sure you loaded the right module that offload is enabled before you compile with

```
ftn -fopenmp saxpy.F90 -o saxpy
```

- Run

```
./saxpy
```

The observed time is much larger than for the CPU version. More parallelism is required!

2) Add parallelism

```
cd ../2_saxpy_teamsdistribute
vi saxpy.f90
```

add "teams distribute" - Compile again - run again The observed time is a bit better than in case 1 but still not the full parallelism is used.

3) Add multi-level parallelism

```
cd ../3_saxpy_paralleldosimd
vi saxpy.f90
```

add “parallel do” for more parellelism - Compile again - run again The observed time is much better than all previous versions. Note that the initialization kernel is a warm-up kernel here. If we do not have a warm-up kernel, the observed performance would be significantly worse. Hence the benefit of the accelerator is usually seen only after the first kernel. You can try this by commenting the !$omp target… in the initialize subroutine, then the meassured kernel is the first which touches the arrays used in the kernel.

Part 2: explore the impact of unified shared memory

4) Explore impact of unified memory:

```
cd ../4_saxpy_nousm
vi saxpy.f90
```

The `!$omp requires...` line is removed. - Compile again - run again so far we worked with unfied shared memory and the APU programming model. This allows good performance on MI300A, but not on discrete GPUs. In case you will work on discrete GPUs or want to write portable code for both discrete GPUs and APUs, you have to focus on data management, too.

```
export HSA_XNACK=0
```

to get similar behaviour like on discrete GPUs (with memory copies). Compiling and running this version without any map clauses will result in much worse performance than with unified shared memory and `HSA_XNACK=1` (no memory copies on MI300A).

Part 3: with map clauses

Set

```
export HSA_XNACK=0
```

that the map clauses do have an effect on MI300A.

5) this version introduces map clauses for each kernel.

```
cd ../5_saxpy_map
vi saxpy.f90
```

see where the map clasues where added. The x vector only has to be maped “to”. - compile again - run again The performance is not much better than version 4.

6) with enter and exit data clauses the memory is only moved once at the beginning the time to solution should be roughly in the order of magnitude of the unified shared memory version, but still slightly slower as the memory is copied like on discrete GPUs. Test yourself:

```
cd ../6_saxpy_targetdata
```

```
vi saxpy.f90
```

- compile again
- run again Additional excercise: What happens to the result, if you comment the !$omp target update (in line 29)?

```
vi saxpy.f90
```

- Don’t forget to recompile after commenting it.

The results will be wrong! This shows, that proper validation of results is crutial when porting! Before you port a large app, think about your validation strategy before you start. Incremental testing is essential to capture such errors like missing data movement.

7) experiment with num_teams

```
cd ../7_saxpy_numteams
vi saxpy.f90
```

specify num_teams(...) choose a number of teams you want to test - compile again - run again investigating different numbers of teams you will find that the compiler default (without setting this) was already leading to good performance. saxpy is a very simple kernel, this finding may differ for very complex kernels.

After finishing this introductory excercise, go to the next excercise in the Fortran folder:

```
cd ../..
```

OpenMP Single Line Compute Constructs:

README.md in `HPCTrainingExamples/Pragma_Examples/OpenMP/C/SingleLineConstructs` of the Training Exercises repository.

We start with adding a single line directive to move the computation of a loop to the GPU. The exercises for this will utilize the saxpy example.

CPU version

This example uses OpenMP on the CPU with threading for parallelism. The pragma used is

```
#pragma omp parallel for
```

We go to the directory with the example and load the amdclang module. We can then build and run the code.

```
cd HPCTrainingExamples/Pragma_Examples/OpenMP/C/SingleLineConstructs
module load amdclang
make saxpy_cpu
./saxpy_cpu
```

You should get some output like:

```
Time of kernel: 0.188165
check output:
y[0] 4.000000
y[N-1] 4.000000
```

You can use the CPU example and port it to the GPU on your own to get more experience at a later point in time. We will step through the process in these exercises to show you how it is done.

First we will work with a very simple case. It has all the code in a single subroutine with arrays allocated on the stack. This permits the compiler to have as much information as possible. Note that we could also load the new amdflang beta which has a perfectly good amdclang compiler. Also, we have made the array size smaller so that it won't run out of stack space.

```
make saxpy_gpu_singleunit_autoalloc
./saxpy_gpu_singleunit_autoalloc
```

You will get a warning about vectorization that is telling you that you do not need the simd clause for the amdclang compiler. But it compiles fine and creates an executable. We run the executable.

```
./saxpy_gpu_singleunit_autoalloc
```

The output

```
Time of kernel: 0.016511
check output:
y[0] 4.000000
y[N-1] 4.000000
```

We note that we did not have to supply any explicit memory management such as a map clause. The compiler can detect the array sizes and that the arrays need to be moved.

Now let's move on to the next example where we dynamically allocate the arrays. We are still using a single subroutine as the previous example.

```
make saxpy_gpu_singleunit_dynamic
./saxpy_gpu_singleunit_dynamic
```

This time we get the following output on a MI200 series GPU.

```
Queue error - HSA_STATUS_ERROR_MEMORY_FAULT
Display only launched kernel:
Kernel 'omp target in main @ 19 (__omp_offloading_34_4474430_main_l19)'
OFFLOAD ERROR: Memory access fault by GPU 8 (agent 0x5ebda70) at virtual address 0x7f81e79dd000. Reasons: Unknown (
Use 'OFFLOAD_TRACK_ALLOCATION_TRACES=true' to track device allocations
Aborted (core dumped)
```

The error message makes it very clear that we are missing the data for the array. We could follow the advice to get a more detailed report if we do not know what array it is. But we'll take a simpler approach. We'll set the `HSA_XNACK` environment variable to tell the system to manage the memory for us. This will work on the data center AMD Instinct GPUs. For workstation GPUs, you may need to add an explicit map clause.

```
export HSA_XNACK=1
./saxpy_gpu_singleunit_dynamic
```

Now we get the expected output:

```
Time of kernel: 0.063025
check output:
y[0] 4.000000
y[N-1] 4.000000
```

So the compiler can sometimes help with moving the memory in very simple cases. But it doesn't take much complexity before it doesn't have enough information. We return to our original `saxpy_cpu.c` example and change the pragma to direct the compiler to offload the calculation to the GPU as already done in `saxpy_gpu_parallelfor.c`. We keep the `HSA_XNACK=1` setting from before.

```
#pragma omp target teams distribute parallel for simd
```

And building and running the example.

```
make saxpy_gpu_parallelfor
./saxpy_gpu_parallelfor
```

Output

```
Time of kernel: 0.061191
check output:
y[0] 4.000000
y[N-1] 4.000000
```

OpenMP has added a simpler loop directive that you can also use. The pragma line is pretty long for the original directive, so this should make it simpler to add to your program. The new pragma is

```
#pragma omp target teams loop
```

This form generally will produce the same results as the earlier directive. But, in principle, it may give the compiler more freedom how to generate the parallel GPU code.

```
make saxpy_gpu_loop
./saxpy_gpu_loop
```

Even the example is a bit easier to run with less typing.

The output

```
Time of kernel: 0.061429
check output:
```

```
y[0] 4.000000
y[N-1] 4.000000
```

So now we have demonstrated how easy it is to add a pragma to a loop to cause it to run on the GPU. And we have seen a little on how the managed memory capability makes the process a little easier. We can focus on parallelizing each loop rather than worrying about where our array data is located.

You can experiment with these examples on both a MI300A APU and a discrete GPU such as MI300X or MI200 series GPU. You should see a performance difference since the MI300A only has to map the pointer and not move the whole array.

OpenMP Single Line Compute Constructs:

README.md from `HPCTrainingExamples/Pragma_Examples/OpenMP/Fortran/SingleLineConstructs` in the Training Examples repository

We start with adding a single line directive to move the computation of a loop to the GPU. The exercises for this will utilize the saxpy example.

`NOTE` : the examples in Fortran also work without setting `HSA_XNACK=1` . The reason is that Fortran passes the array size information along with the array. So the compiler has more information to work with. In Fortran, the additional information is called the "dope" vector. It is last century slang for "give me the dope on it". We would say "beta" in today's slang.

CPU version

This example uses OpenMP on the CPU with threading for parallelism. The pragma used is

```
#pragma omp parallel for
```

We go to the directory with the example and load the amdclang module. We can then build and run the code.

```
cd HPCTrainingExamples/Pragma_Examples/OpenMP/Fortran/SingleLineConstructs
module load amdflang-new
make saxpy_cpu
./saxpy_cpu
```

You should get some output like:

```
Time of kernel: 0.151135
plausibility check:
y(1) 4.000000
y(n-1) 4.000000
```

You can use these CPU examples and port them to the GPU on your own to get more experience at a later point in time. We will step through the process in these exercises to show you how it is done.

First we will work with a very simple case. It has all the code in a single subroutine with statically allocated arrays on the stack. This permits the compiler to have as much information as possible. Note that we could also load the regular amdclang module instead of the new amdflang. Also, we have made the array size smaller so that it won't run out of stack space.

```
make saxpy_gpu_singleunit_autoalloc
./saxpy_gpu_singleunit_autoallloc
```

The output

```
Time of kernel: 0.022465
 plausibility check:
y(1) 4.000000
y(n) 4.000000
```

We note that we did not have to supply any explicit memory management such as a map clause. The compiler can detect the array sizes and that the arrays need to be moved.

Now let's move on to the next example where we dynamically allocate the arrays. We are still using a single subroutine as the previous example. Note that, unlike the C case, we are not setting `HSA_XNACK=1` to make the example run (see note at the beginning of this README):

```
make saxpy_gpu_singleunit_dynamic
./saxpy_gpu_singleunit_dynamic
```

This time we get the following output:

```
Time of kernel: 0.022440
 plausibility check:
y(1) 4.000000
y(n) 4.000000
```

We return to our original `saxpy_cpu.c` example and change the pragma to direct the compiler to offload the calculation to the GPU as already done in `saxpy_gpu_paralleldo.F90` . setting from before.

```
#pragma omp target teams distribute parallel for simd
```

And building and running the example.

```
make saxpy_gpu_paralleldo
./saxpy_gpu_paralleldo
```

Output

```
Time of kernel: 0.052156
 plausibility check:
y(1) 4.000000
y(n) 4.000000
```

OpenMP has added a simpler loop directive that you can also use. The pragma line is pretty long for the original directive, so this should make it simpler to add to your program. The new pragma is

```
#pragma omp target teams loop
```

This form generally will produce the same results as the earlier directive. But, in principle, it may give the compiler more freedom how to generate the parallel GPU code.

```
make saxpy_gpu_loop
./saxpy_gpu_loop
```

Even the example is a bit easier to run with less typing.

The output

```
Time of kernel: 0.052010
 plausibility check:
y(1) 4.000000
y(n) 4.000000
```

So now we have demonstrated how easy it is to add a pragma to a loop to cause it to run on the GPU. And we have seen a little on how the managed memory capability makes the process a little easier. We can focus on parallelizing each loop rather than worrying about where our array data is located.

You can experiment with these examples on both a MI300A APU and a discrete GPU such as MI300X or MI200 series GPU. You should see a performance difference since the MI300A only has to map the pointer and not move the whole array.

We have one less example to look at. Many scientific codes have multi-dimensional data that need to be operated on. We can use the collapse clause to spread out the work from both loops rather than just the

outer one. This can be helpful if the outer loop is small. But since we are always trying to generate more work and parallelism, it can also have some benefit for larger outer loops.

We'll consider the case of Fortran since 2-dimensional arrays are much easier to work with. The directive will now become

```
!$omp target teams distribute parallel do collapse(2)
```

Building and running the example

```
make saxpy_gpu_collapse
./saxpy_gpu_collapse
```

And the output

```
Time of kernel: 0.029263
 plausibility check:
y(1,1) 4.000000
y(m,n) 4.000000
```

OpenMP complex compute constructs in C

README.md from `HPCTrainingExamples/Pragma_Examples/OpenMP/C/ComplexComputeConstructs` in the Training Examples repository

These exercises explore more complex compute constructs. We begin with breaking apart the meanings of each of the clauses in the single combined compute directive.

First retrieve the examples for this part.

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

Full combined compute directive

We'll start with a baseline from the full combined compute directive

```
#pragma omp target teams distribute parallel for simd
```

Setting up the environment

```
module load amdclang
export HSA_XNACK=1
export LIBOMPTARGET_KERNEL_TRACE=1
```

This example is in the previous exercises on simple single line compute constructs

```
cd HPCTrainingExamples/Pragma_Examples/OpenMP/C/SingleLineConstructs
```

```
make saxpy_gpu_parallelfor
./saxpy_gpu_parallelfor
```

Check the output. It should be something like the following, but with some variation depending on the GPU model you are on.

```
DEVID:  0 SGN:5 ConstWGSize:256  args: 5 teamsXthrds:( 416X 256) reqd:(   0X   0) lds_usage:0B sgpr_count:24 vgpr_c
Time of kernel: 0.082906
```

There are 416 teams (workgroups) of size 256. There is a low vector register usage a 8. We'll also look at the run-time of 0.082906 for comparison.

Target directive

We'll start with what happens with just the target directive

cd HPCTrainingExamples/Pragma_Examples/OpenMP/C/ComplexComputeConstructs

Setting up the environment

```
module load amdclang
export HSA_XNACK=1
export LIBOMPTARGET_KERNEL_INFO=1
```

```
make saxpy_gpu_target
./saxpy_gpu_target
```

The output will be similar to the following:

```
DEVID:  0 SGN:3 ConstWGSize:257  args: 5 teamsXthrds:(   1X 256) reqd:(   0X   0) lds_usage:16B sgpr_count:16 vgpr_
Time of kernel: 5.407085
```

We only have one team of 256 workgroup size. Basically we are running serial – one thread on one team (workgroup). The runtime reflects that with 65 times longer than the combined directive.

Teams clause

The teams exercise will add the teams clause after the target directive.

```
make saxpy_gpu_target_teams
./saxpy_gpu_target_teams
```

The output

```
DEVID:  0 SGN:3 ConstWGSize:257  args: 5 teamsXthrds:( 624X 256) reqd:(   0X   0) lds_usage:16B sgpr_count:12 vgpr_
Time of kernel: 11.166301
```

There are 624 workgroups, but each one is doing all the work. This duplicates the effort and ends up taking twice the time as the target directive alone. Note that this is also creating a race condition when threads are trying to write to the same location, which produces an incorrect output that is also non deterministic. One could add `num_teams(1)` to the pragma directive to require the creation of a single team, in which case no race condition can occur.

Distribute clause

Adding the distribute clause starts to get some parallelism by partitioning the work across the workgroups. But still with only one thread per workgroup.

```
make saxpy_gpu_target_teams_distribute
./saxpy_gpu_target_teams_distribute
```

Output

```
DEVID:  0 SGN:3 ConstWGSize:257  args: 5 teamsXthrds:( 624X 256) reqd:(   0X   0) lds_usage:16B sgpr_count:24 vgpr_
Time of kernel: 0.149113
```

We have more workgroups at 624 than the baseline case, but we are not using all the threads. This is using more of the compute capacity at 624/416 times as many workgroups and associated compute units. The runtime is much closer to the baseline. As a further exploration, try changing the array size in the example or trying a different kernel with more work.

parallel for without the teams distribute clauses

As a further experiment, let's try just adding parallel for to engage all the threads on one workgroup. The directive is the following:

```
#pragma omp target parallel for
```

Building and running it

```
make saxpy_gpu_parallel_for
./saxpy_gpu_parallel_for
```

Output should be something like

```
DEVID:  0 SGN:2 ConstWGSize:256  args: 5 teamsXthrds:(   1X 256) reqd:(   0X   0) lds_usage:32B sgpr_count:25 vgpr_
Time of kernel: 0.126748
```

This gives a pretty good runtime while using fewer GPU compute units.

Split multi-level directive

Build both the collapse and split level C examples.

```
make saxpy_gpu_collapse
./saxpy_gpu_collapse
make saxpy_gpu_split_level
./saxpy_gpu_split_level
```

Compare the output from LIBOMPTARGET_KERNEL_TRACE=1.

```
DEVID:  0 SGN:5 ConstWGSize:256  args: 6 teamsXthrds:(3907X 256) reqd:(   0X   0) lds_usage:0B sgpr_count:29 vgpr_c
```

```
Time of kernel: 0.027777
```

```
DEVID:  0 SGN:3 ConstWGSize:257  args: 6 teamsXthrds:( 416X 256) reqd:(   0X   0) lds_usage:36B sgpr_count:27 vgpr_
```

```
Time of kernel: 0.027449
```

On your own: try different array sizes and ratios of iterations between the loop levels.

Reduction excercise:

README.md from `HPCTrainingExamples/Pragma_Examples/OpenMP/C/2_reduction` from the Training Examples repository.

This excercise will show how to port a reduction.

0) serial CPU version Version to port yourself. Don't forget to port the Makefile.

```
cd 0_reduction_portyourself
```

Build

```
make
```

run

```
./vecadd
```

Remember the output result for the serial version to validate the offload version. Adapt Makefile for offload. Port the example, build and run after every kernel you ported to ensure correctness.

1) solution with unified shared memory Set `export HSA_XNACK=1` to test this version.

```
cd 1_reduction_usm
```

Build

```
make
```

run

```
./reduction
```

Note: you may want to use `vimdiff <file1> <file2>` to compare your solution with this version.

2) solution with map clauses Set `export HSA_XNACK=0` to test this version.

```
cd 2_reduction_map
```

Build

```
make
```

run

```
./reduction
```

Note: you may want to use `vimdiff <file1> <file2>` to compare your solution with this version.

Porting exercise: reduction

README.md from `HPCTrainingExamples/Pragma_Examples/OpenMP/Fortran/2_reduction` from the Training Examples repository.

This excercise focusses on two things: - Part 1: how to port a reduction to the GPU - Part 2: importance of map clauses on discrete GPUs or HSA_XNACK=0 on MI300A

First, prepare the environment (loading modules, set environment variables), if you didn't do so before. ## For Part 1 and 2: serial CPU version to port 0) a version to port yourself.

```
cd 0_reduction_portyourself
vi freduce.F
```

- Only port the Makefile and the reduction itself. This excercise focusses on how to implement a reduction, not on porting the full example.

How to build all versions:

```
make
```

and run:

```
./freduce
```

The other folders 1 and 2 have different flavors of the solution: ## Part 1: Port with unified shared memory

```
cd 1_reduction_solution_usm
vi freduce.F
```

contains a sample solution for unified shared memory / APU programming model (correct output: each element 1010) run this with setting `export HSA_XNACK=1` in advance

Part 2: Port with map clause

2.1 Porting excercise

```
cd 2_reduction_solution
vi freduce.F
```

Contains a sample solution for discrete GPUs (correct output: each element 1010) run this with setting `export HSA_XNACK=0` in advance ### 2.2 Behaviour with and without USM The third folder contains an excercise to explore the behavior with and without USM:

```
cd 3_reduction_solution
vi freduce.F
```

This example intentionally does the mapping wrong (from instead of to). You can see how the result changes (output 1000 instead of 1010) when you use export `export XSA_XNACK=0` . No error is shown, but the result is wrong. Test the same wrong code with `export HSA_XNACK=1` , then the result is correct again as mapping clauses are ignored. Take home message: if you develop for both APUs and discrete GPUs on MI300A, check if the results are the same for `HSA_XNACK=0` and `=1` as map clauses will be ignored with `HSA_XNACK=1` ! Ignoring memory copies is great for code portability and performance without code changes, but be careful to include proper validation checks during development for both discrete GPUs and APUs. # Porting excercise reduction of multiple scalars in one kernel

README.md from `HPCTrainingExamples/Pragma_Examples/OpenMP/C/4_reduction_scalars` from the Training Examples repository.

This folder has two code versions:

0) a serial cpu version to port yourself. Hint: don't forget to port the Makefile.

Build:

```
make
```

Run:

```
./reduction_scalar
```

1) an openmp offload ported solution. The solution shows how you can do a reduction of multiple scalars in one kernel. Note that scalars do not need to be explicitly mapped. # Porting excercise reduction of multiple scalars in one kernel

README.md from `HPCTrainingExamples/Pragma_Examples/OpenMP/Fortran/4_reduction_scalars` from the Training Examples repository.

This folder has two code versions:

0) a serial cpu version to port yourself.

Hint: don't forget to port the Makefile.

Build:

```
make
```

Run:

```
./reduction_scalar
```

1) an openmp offload ported solution. It shows how you can do a reduction of multiple scalars in one kernel. Note that scalars do not need to be explicitly mapped. # Porting excercise reduction into an array

README.md from `HPCTrainingExamples/Pragma_Examples/OpenMP/C/6_reduction_array` from the Training Examples repository.

This folder has two code versions:

0) a serial cpu version to port yourself. Hint: don't forget to port the Makefile.

Build:

```
make
```

Run:

```
./reduction_array
```

1) an openmp offload ported solution. The solution shows how you can do a reduction of multiple values into an array in one kernel. # C Code – Porting device routine exercises

README.md from `HPCTrainingExamples/Pragma_Examples/OpenMP/C/5_device_routines` in Training Examples repository

This exercise will show how to port kernels which call a subroutine or function. Each version has a sub-folder with a - serial CPU version to port yourself and a - solution for unified memory and - a solution with map clauses.

Build and run analogous to the previous exercises.

There are three different versions:

```
cd 1_device_routine
```

Explore the serial CPU code first.

```
cd 0_device_routine_portyourself
module load amdclang
make
./device_routine
```

The rocm module will be loaded with the amdclang module. And the current rocm module will set `HSA_XNACK=1` . If there are no modules set up on your system, set the `CC` environment variable to the full path to the C compiler you want to use. Add the ROCm directory to the PATH and also the LLVM directory under ROCm. Also add the lib directory to the `LD_LIBRARY_PATH` . And finally set `HSA_XNACK` with `export HSA_XNACK=1` .

```
make
./device_routine
```

You should see the result:

```
Result: sum of x is 1000.000000
```

Now try and convert the example to run on the GPU. Start with adding `#pragma omp target teams distribute parallel` before the for loops in the main program in `device_routine.c` . Note that one of the loops also needs a `reduction(+:sum)` clause added to the target directive. How do you show the compiler to compile the function in the other file, compute.c, for the GPU? Try adding the `#pragma omp declare target` directive to the subroutine declaration in compute.c.

There are two solutions for this exercise. One with the APU programming model using unified shared memory. The other has explicit map clauses for when unified shared memory is not available or not being used. We'll look at the unified shared memory version first.

```
cd ../1_device_routine_usm
```

Look at the two C source files and compare to the originals in `0_device_routine_portyourself` . To build and run the example:

```
make
./device_routine
```

Similarly with the solution using map clauses:

```
cd ../2_device_routine_map
```

Look for the map clauses in the `device_routine.c` source file. In this case, The memory is only accessed on the GPU. So, we use map(alloc:x[0:N]) and map(release:x[0:N]) in the clauses. Build and run the examples.

```
make
./device_routine
```

```
cd 2_device_routine_wglobaldata
```

First look at the original code in `0_device_routine_wglobaldata_portyourself` .

```
cd 0_device_routine_wglobaldata_portyourself
```

Note the addition of the `global_data.c` file with the definition of the constants array. Build and run the example.

```
make
./device_routine
```

Now try modifying the example to run on the GPU. How do you use the global data from the `global_data.c` file in your device subroutine?

For the solution, lets look at the example in `1_device_routine_wglobaldata` .

```
cd 1_device_routine_wglobaldata
```

Look at the directive `#pragma omp declare target` in the `global_data.c` file. Is this necessary for your version of the compiler?

It is a bit more complicated if the data being used is dynamically allocated. We have to be sure and map it over to the GPU after the memory allocation. We can experiment with this case in the next example.

```
cd ../3_device_routine_wdynglobaldata
```

Again there is a version that you can try and port before looking at the solution.

```
cd 0_device_routine_wdynglobaldata_portyourself
```

Look at the `global_data.c` file and experiment with the right directive to move the data to the GPU.

The solution is also available.

```
cd 1_device_routine_wdynglobaldata
```

See the directives used to move the constants array to the GPU. Note that we also need to add declare target on the pointer to the array.

```
#pragma omp target enter data map(alloc:constants[0:isize])
```

In this example, we initialize the data on the GPU with:

```
#pragma omp target teams distribute parallel for
   for (int i = 0; i< isize; i++) {
      constants[i] = (double)i;
   }
```

How would this be different if we initialized the data on the CPU?

Part 1: Fortran with interface blocks

README.md from `HPCTrainingExamples/Pragma_Examples/OpenMP/Fortran/5_device_routines` in Training Examples repository

Let's start with the device routine in a separate file with an interface.

```
cd device_routine_with_interface
```

there are six code versions in enumerated folders:

```
0_device_routine_portyourself
1_device_routine_wrong
2_device_routine_usm
3_device_routine_map
4_device_routine_device_type
5_device_routine_enter_data
```

Starting with the CPU version to try and porting yourself

```
cd 0_device_routine_portyourself
```

Build and run

```
make
./device_routine
```

The result should be

```
Result: sum of x is 1000.000000000000
```

Now add the directive to the three loops in `device_compute.f90`

```
!$omp target teams distribute parallel do
```

For the last loop, it is also necessary to add `reduction(+:sum)`

This has been done for you in the `1_device_routine_wrong` directory

```
cd ../1_device_routine_wrong
```

Build the code

```
make
```

You should see an error.

```
ld.lld: error: undefined symbol: compute_
```

The compute routine is created only for the host and not for the device. So we need to add the device target directive to the compute subroutine definition in `compute.f90`.

Moving to the next version at 2_device_routine_usm directory where the device target directive has been added.

```
cd ../2_device_routine_usm
```

Note the additions. In compute.f90:

```
      subroutine compute(x)
          implicit none
          !$omp requires unified_shared_memory
          !$omp declare target
```

and in device_compute.f90

```
 program device routine
...
         implicit none
         !$omp requires unified_shared_memory

...
```

Now build and run the example

```
make
./device_routine
```

For the case where we want to do explicit memory movement, we use maps as show in `03_device_routine_map`
.

```
cd ../03_device_routine_map
```

We take out the `!$omp requires unified_shared_memory` and add `map(tofrom:x)` and `map(to:x)` clauses. We can run this example as before:

```
make
./device_routine
```

Some of the other clauses that can be uses are the `device_type(nohost)` that only generates device code for the declare target clauses. Check out the example at

```
cd ../4_device_routine_device_type
make
./device_routine
```

The last example shows the use of the enter/exit data directives. This is an example of the use of unstructured data movement directives.

```
!$omp target enter data map(alloc:x(1:N))
!$omp target exit data map(delete:x)
```

These are added to the code in `5_device_routine_enter_data`

```
cd ../5_device_routine_enter_data
make
./device_routine
```

Part 2: Fortran with modules

There are three versions

```
0_device_routine_with_module_portyourself
1_device_routine_with_module
2_device_routine_with_module_usm
```

We first check out the original code in `0_device_routine_with_module_portyourself`

```
cd 0_device_routine_with_module_portyourself
```

Build and run

```
make
./device_routine
make clean
```

Now try and add the directives to port the example code to run on the device (GPU).

The solution for explicit data movement using unstructured memory directives is in `1_device_routine_with_module`

```
cd ../1_device_routine_with_module
make
./device_routine
```

Examining the two source files, we see that we first need to add the compute directives:

```
!$omp target teams distribute parallel do
!$omp target teams distribute parallel do reduction(+:sum)
```

In addition, we need the explicit memory movement directives

```
!$omp target enter data map(alloc:x(1:N))
!$omp target exit data map(delete:x)
```

But that is not all we need to do. We also need to add `!$omp declare target` in compute.f90 to tell the compiler to generate a device version of the compute subroutine.

The next example shows the unified shared memory version.

```
cd ../2_device_routine_with_module_usm
```

We need to add `!$omp requires unified_shared_memory` to both source code files since they both will have OpenMP target directives. Now we just need to add the compute directives as above and also add the `!$omp declare target` directive inside the subroutine definition in computemod.f90.

Now build and run

```
make
./device_routine
```

C++ member function

README.md from `HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/5_device_routines` in Training Examples repository

The first example is where there is a compute method in the Science class that is called from a parallel target region.

```
cd 1_member_function
```

The original code is shown in `0_member_function_portyourself`

```
cd 0_member_function_portyourself
```

Try adding the `#pragma omp target teams loop` directive to the loop in the `bigscience.cc` routine to port it to run on the device.

To see the solution to the porting, see the code in `1_member_function` directory.

Looking at the loop in `bigscience.cc` :

```
#pragma omp target teams loop
   for (int k = 0; k < N; k++){
      myscienceclass.compute(&x[k], N);
   }
```

To try out the code, compile it and run it.

```
make
./bigscience
```

Note that nothing needs to be done to the class in `Science.hh` . Why is this? Basically, the defined method function in `Science.hh` is in-lined into the `bigscience.cc` file. So it is handled by the directive added around the loop in `bigscience.cc` .

C++ member function external

So what happens when the compute method is defined in a different file? For this case, let's take a look at the next example in `2_method_function_external` .

```
cd ../2_method_function_external
```

Try porting the code in `0_member_function_external_portyourself` . Note that in this example, the compute member function is defined in `Science_member_functions.cc`

For the solution, go to the `1_member_function_external` directory

```
cd ../1_member_function_external
```

Note that now we have to add `#pragma omp declare target` around the compute method definition. We also need a `#pragma omp end declare target` directive to close out the declare target region.

Let's try compiling and running the example

```
make
./bigscience
```

The next example, `2_member_function_external_data` uses a data value `init_value` from the Science class. The thing to note is that we do not need to add a `#pragma omp declare target` around the declaration in the class.

Check that this runs fine with your compiler

```
cd ../2_member_function_external_data
make
./bigscience
```

C++ virtual methods

Additional complexity in C++ classes can cause difficulties with porting to GPUs. Fundamentally, the GPU language is C with only a little support for C++. So let's take a look at a simple virtual method where class inheritance is used.

```
cd ../../3_virtual_methods
```

The original CPU C++ code is given in `0_virtual_methods_portyourself`. We create a new HotScience class that is based on the Science class. The new class is defined in `HotScience.hh`. It overrides the compute method. The method definition for the new compute function is in `HotScience_member_functions.cc`.

First, let's verify that the original code works.

```
cd 0_virtual_methods_portyourself
make
./bigscience
```

Try porting this version and see what might be required.

The solution is given in `1_virtual_methods` directory.

```
cd ../1_virtual_methods
```

Examine the source code files to see what is needed. Note that now the `#pragma omp declare target` block is needed around the method definition in `HotScience_member_functions.cc`. Let's verify that this works with your current compiler.

```
make
./bigscience
```

A special note here for the current amdclang++ compiler. With the changes to the source code, the compiler issues a warning about maybe not being mapped correctly

```
warning: type 'HotScience' is not trivially copyable and not guaranteed to be mapped correctly
```

The code still compiles and runs properly. To suppress the warning, `-Wno-openmp-mapping` has been added to CXXFLAGS in the Makefile. # OpenMP Offloading for C++ Codes that use Classes

README.md in `HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/cpp_classes` from the Training Examples repository

These examples show how to use OpenMP for GPU offloading in the context of a C++ code that makes uses of classes, and a programming paradigm where the most relevant members of the class are private, with their associated values accessed and modified by appropriate `get` and `set` functions.

In the present directory, you will find two subdirectories, one called `usm` and one called `explicit`.

The `usm` Sub-directory

In this context, `usm` stands for unified shared memory, which is what we are requiring for the code samples in this directory. To compile the code in the `usm` directory, do:

```
module load rocm
module load amdclang
export HSA_XNACK=1
make
```

If the `amdclang` module is not available on your system, make sure to do:

```
export CXX=$ROCM_PATH/llvm/bin/amdclang++
```

before running the `make` command.

Note that if one was to not set `HSA_XNACK=1` the code would not compile, because we are requiring unified shared memory with the following pragma line in `main.cpp`:

```
#pragma omp requires unified_shared_memory
```

You may have noticed many compiler warnings such as this one:

```
main.cpp:22:20: warning: Type 'daxpy' is not trivially copyable and not guaranteed to be mapped correctly [-Wopenmp
   22 |       double val = data.getConst() * data.getX(i) + data.getY(i);
```

From the warning, you can already see what potential issues can arise in a C++ programming paradigm like the one we decided to set ourselves in. When possible, using unified shared memory can help get around those warnings.

In the `usm` directory, there are two subdirectories, `daxpy` and `operations`.

The `daxpy` Sub-directory

Here we are defining a class object to perform a daxpy operation. Notice that the daxpy operation is performed within the `main.cpp`. Moreover, we are using the `get` and `set` member functions of the daxpy class from within the target region without using any maps, thanks to the unified shared memory framework.

The `operations` Sub-Directory

The code in the `operations` directory adds one layer of complexity and performs a daxpy from the `main.cpp` file but using a class called `operations` that has two members of class type: one of type `daxpy`, already mentioned before, and one of type `norm`, which will compute a user-defined norm of an input vector, in this case the output of the daxpy operation. Note that everything works seamlessly even when calling member functions from the `ops` object: these member functions are wrappers to the member functions of the `daxpy` and `norm` class members.

The `explicit` Sub-directory

This sub-directory contains example code that is meant to work even without enabling unified shared memory, meaning that it will compile and run regardless of whether `HSA_XNACK=1`. This is achieved by creating an appropriate data environment with the use of maps, as it will explained next. To compile:

```
module load rocm
module load amdclang
make
```

Again, make sure that the CXX environment variable is set as below, before running the `make` command:

```
export CXX=$ROCM_PATH/llvm/bin/amdclang++
```

The directory is named `explicit` because we are explicitly taking care of all the data movement between host and device, helping the compiler with figuring out how to perform the offload to GPU. The only sub-directory here is `daxpy` .

The `daxpy` Sub-directory

The explicit memory movement scenario gets tricky really quickly, as you have seen with the numerous warning messages produced by the compiler when building the `usm` examples. Things get particularly complicated when using anything that is not just a pointer for our data members, such as for instance standard vectors, like we were doing in the `usm` directory. In the `daxpy.hpp` file where the `daxpy` class is declared, we have now included in the constructor the following pragma:

```
#pragma omp target enter data map(alloc: x_[0:N_],y_[0:N_]) map(to: a_)
```

The above pragma creates a data environment for an unstructured data region and maps x__, y__, N__ and a__ to the device. Note that we also had to explicitly map the scalars to make sure that they are available on the device when we call the `apply` function, which is defined in `daxpy.cpp` . The following pragma is included in the destructor for the class:

```
#pragma omp target exit data map(delete: x_[0:N_],y_[0:N_], a_)
```

HIP/basic_examples Documentation

Table of Contents

1. `01_error_check`
2. `02_add_d2h_data_transfer`
3. `03_complete_square_elements`
4. `04_complete_matrix_multiply`
5. `05_compare_with_library`
6. `06_hipify_pingpong`
7. `07_matrix_multiply_shared`

Please refer to the individual directories for documentation specific to each exercise. # Find the error

Compile and run the vector addition program and use the error from the error-checking macro to decide how to fix the problem.

To compile and run:

```
$ make

$ sbatch -A <account-name> submit.sh
```

where `account-name` is your account name for the system (may be required for certain systems). A job file titled `<name-of-exercise>-%J.out` will be produced, where `%J` is the job id number of your run. To check your program output, simply run:

```
cat <name-of-exercise>-%J.out
```

Add the device-to-host data transfer

This example simply initializes an array of integers to 0 on the host, sends the 0s from the host array to the device array, then adds 1 to each element in the kernel, then sends the 1s back to the host array.

However, the device-to-host data transfer call (`hipMemcpy`) is missing. Please add in the missing call and run the program. Look for the TODO.

This is the API call to use:

```
hipError_t hipMemcpy(void *dst, void *src, size_t size_in_bytes, hipMemcpyKind kind)
```

To compile and run:

```
$ make

$ sbatch -A <account-name> submit.sh
```

where `account-name` is your account name for the system (may be required for certain systems). A job file titled `<name-of-exercise>-%J.out` will be produced, where `%J` is the job id number of your run. To check your program output, simply run:

```
cat <name-of-exercise>-%J.out
```

Complete the square elements kernel

In this exercise, there is a host array and a device array. The host array is initialized in a loop so each element is given the value of the iteration from 0 to N-1. Then the host array is copied to the device array, and the GPU kernel simply squares each element of the array. Then the results are sent back from the device array to the host array.

However, the kernel is not complete. So you must complete the kernel by adding in the line where the value is squared, and make sure to guard for going out of the array bounds. Look for the TODO.

To compile and run:

```
$ make

$ sbatch -A <account-name> submit.sh
```

where `account-name` is your account name for the system (may be required for certain systems). A job file titled `<name-of-exercise>-%J.out` will be produced, where `%J` is the job id number of your run. To check your program output, simply run:

```
cat <name-of-exercise>-%J.out
```

Complete the matrix multiply kernel

In this exercise, a matrix multiply is performed on the GPU. In the code, the indices `row_index` and `col_index` iterate through the arrays in row-major (across the first row, then across the second row, etc.) and column-major (down the first column, then down the second column, etc.) order, respectively.

Look at the matrix multiply kernel and decide which of these two indices should define the elements of arrays A and B. Look for the TODO.

To compile and run:

```
$ make

$ sbatch -A <account-name> submit.sh
```

where `account-name` is your account name for the system (may be required for certain systems). A job file titled `<name-of-exercise>-%J.out` will be produced, where `%J` is the job id number of your run. To check your program output, simply run:

```
cat <name-of-exercise>-%J.out
```

Complete the matrix multiply kernel

In this exercise, we will use the `matrix_multiply` kernel we completed in `04_complete_the_kernel` and compare its performance against the hipBLAS version of DGEMM.

You will not need to make any code changes. Instead, you will simply compile the code and submit the job. This will run the code under the `rocprof` profiling tool and parse the results.

To compile and run:

```
$ make

$ sbatch -A <account-name> submit.sh
```

where `account-name` is your account name for the system (may be required for certain systems). A job file titled `<name-of-exercise>-%J.out` will be produced, where `%J` is the job id number of your run. To check your program output, simply run:

```
cat <name-of-exercise>-%J.out
```

To view the resulting profile, run the python script:

```
./parse_output.py
```

It should be clear from the performance difference that using existing libraries is typically the right choice instead of re-inventing the (slower) wheel. # hipify the CUDA pingpong code

This code sends data back and forth between the host and device 50 times and calculates the bandwidth.

Your job is to `hipify` the code, then compile and run it. For this exercise, it is recommend to use `hipify-perl` on the CUDA program and redirect the output to a new file titled `pingpong.cpp` . NOTE: The `#include "hip/hip_runtime.h" doesn't always get added when a code is` hipify‘-ed, so it might need to be added manually.

To compile and run:

```
$ make

$ sbatch -A <account-name> submit.sh
```

where `account-name` is your assigned Frontier username. A job file titled `<name-of-exercise>-%J.out` will be produced, where `%J` is the job id number of your run. To check your program output, simply run:

```
cat <name-of-exercise>-%J.out
```

or open the file directly using `vim` .

Recall that the CPU and GPU are connected with PCIe4 (x16), which has a peak bandwidth of 32 GB/s. What percentage of the peak performance do we achieve? # Complete the matrix multiply with shared memory

In this example, a matrix multiply is performed with shared memory, where each thread computes 1 element of the resultant matrix.

NOTE: The shared memory allocations are only of size `THREADS_PER_BLOCK` , which is smaller than the array size. So each thread must loop through its dot-product (since that's what each element of the resultant matrix is) in chunks until it completes the full dot product.

Your job in this exercise is to correctly copy the data from global memory into the shared memory arrays, then compile and run the program.

To compile and run:

```
$ make
```

```
$ sbatch -A <account-name> submit.sh
```

where `account-name` is your assigned Frontier username. A job file titled `<name-of-exercise>-%J.out` will be produced, where `%J` is the job id number of your run. To check your program output, simply run:

```
cat <name-of-exercise>-%J.out
```

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

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 `hipifly.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` # HIP and OpenMP Interoperability

README.md in `HPCTrainingExamples/HIP-OpenMP/CXX` from the Training Examples in repository. If the `amdclang` is not available in your system, make sure to do `export CXX=amdclang++` .

Full OpenMP Application Code

The first example is just a straightforward openmp offload version of saxpy. Any C++ compiler that supports OpenMP offload to hip should work.

```
cd HPCTrainingExamples/HIP-OpenMP/CXX/saxpy_openmp_offload
module load rocm
module load amdclang
make
```

OpenMP Application Calling a HIP Kernel

Now we move on to an OpenMP main calling a HIP version of the saxpy kernel. Note that we have to get the device version of the array pointers to pass into the HIP kernel, using `use_device_ptr(x,y)` .

```
cd HPCTrainingExamples/HIP-OpenMP/CXX/saxpy_openmp_hip
module load rocm
module load amdclang
unset HSA_XNACK
make
```

Try to create an equivalent version of the code in `saxpy_openmp.cc` that uses `omp target enter data` and `omp target exit data` instead of `omp target data` .

APU Programming Model Version

With the APU programming model the explicit memory management handled with OpenMP in the `saxpy_open_hip` directory can now be removed. The code has to be run after setting `HSA_XNACK=1` to enable unified shared memory:

```
cd HPCTrainingExamples/HIP-OpenMP/CXX/saxpy_APU
module load rocm
module load amdclang
export HSA_XNACK=1
make
```

HIP application calling an OpenMP Kernel

The next example does the converse of what we saw: it is a HIP application code calling an OpenMP kernel for saxpy executing on the GPU:

```
cd HPCTrainingExamples/HIP-OpenMP/CXX/saxpy_hip_openmp
module load rocm
module load amdclang
unset HSA_XNACK
make
```

Try to create a version that leverages the APU progamming model, in a similar way done for the OpenMP application calling a HIP kernel.

OpenMP and HIP Kernels in the Same Source File

You can put both OpenMP and HIP code in the same source file with some care. The next hands-on exercise shows how in 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
module load 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
```

Running a Fortran to HIP interop example

README.md `HPCTrainingExamples/Pragma_Examples/OpenMP/Fortran/8_interop` from the Training Examples repository

This is a simple example to demonstrate fortran to HIP interoperability

```
module load rocm
module load amdflang-new
make
```

run the code:

```
./interop
```

Code will run to completion if it passes verification ## Calling GEMM from Fortran

README.md in `HPCTrainingExamples/HIP-OpenMP/F/Calling_DGEMM` from the Training Examples repository

The files in this directory show how to call a rocblas dgemm function from an OpenMP application code written in Fortran. If the `amdclang` module is not available in your system, set `FC=amdflang` or to the next generation AMD Fortran compiler.

Explicit Memory Management

In this explicit memory management example, a target data region is created, from which a wrapper to the rocblas dgemm is called. Pay particular attention to the items passed to the wrapper call. Also notice the use `use_device_addr(A,B,C)` before the call to the wrapper.

What happens if you instead use `use_device_ptr(A,B,C)` ? Check the output by setting `OMPLIBTARGET_INFO=-1` . Remember that the behavior of OpenMP directives may be different across languages, such as Fortran and C++.

To compile and run:

```
module load rocm
module load amdclang
make
```

Unified Shared Memory

In the `usm` directory, we are showing how the code can be simplified rather dramatically by removing all the explicit data management due to the use of unified shared memory, setting HSA_XNACK=1. To compile and run:

```
module load rocm
module load amdclang
make
```

Try to use hipfort to avoid having to include the explicit rocm_interface that we are using in this example.

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 12 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

```
git clone https://github.com/amd/HPCTrainingExamples.git
cd HPCTrainingExamples/HIP-Optimizations/daxpy
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.
- You may find that the various optimizations work differently in MI210 vs MI300A devices, and this may be due to differences in hardware architecture.

Kokkos examples

Stream Triad

Step 1: Build a separate Kokkos package

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.

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

Build Kokkos with OpenMP backend

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

make -j 8
make install

cd ..
```

Build Kokkos with HIP backend

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

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

Set Kokkos_DIR to point to external Kokkos package to use

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

Step 2: Modify Build

Get example

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

Test serial version with

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

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

Add to CMakeLists.txt

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

Retest with

```
cmake ..; make
```

and run ./StreamTriad again

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

Step 3: Add Kokkos views for memory allocation of arrays

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

Add include file

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

Add initialize and finalize

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

} Kokkos::finalize();
```

Replace static array declarations with Kokkos views

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

Rebuild and run

```
CXX=hipcc cmake ..
make
./StreamTriad
```

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

At start of loop

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

At end of loop, replace closing brace with

```
});
```

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

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

How much speedup do you observe?

Step 5: Add Kokkos timers

Add Kokkos calls

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

Remove

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

6. Run and measure performance with OpenMP

Find out how many virtual cores are on your CPU

```
lscpu
```

First run with a single processor:

Average runtime ________________

Then run the OpenMP version:

Average runtime ________________

Portability Exercises

1. Rebuild Stream Triad using Kokkos build with HIP

Set Kokkos_DIR to point to external Kokkos build with HIP

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

2. Run and measure performance with AMD Radeon GPUs

HIP build with ROCm

Ver4 - Average runtime is _________ msecs

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
# MPI Ghost Exchange Optimization Examples

## Changes Between Example Versions
This code contains several implementations of the same ghost exchange algorithm at varying stages
of optimization:
```

- **Orig**: Shows a CPU-only implementation that uses MPI, and serves as the starting point for further optimizatio
- **Ver1**: Shows an OpenMP target offload implementation that uses the Managed memory model to port the code to GP
- **Ver2**: Shows the usage and advantages of using `roctx` ranges to get more easily readable profiling output fro
- **Ver3**: Under Construction, not expected to work at the moment
- **Ver4**: Explores heap-allocating communication buffers once on host.
- **Ver5**: Explores unrolling a 2D array to a 1D array.
- **Ver6**: Explores using explicit memory management directives to specify when data movement should happen.
- **Ver7**: Under Construction, not expected to work at this time.

<details>
<summary><h3>Background Terminology: We're Exchanging <i>Ghosts?</i></h3></summary>
<h4>Problem Decomposition</h4>
In a context where the problem we're trying to solve is spread across many compute resources,
it is usually inefficient to store the entire data set on every compute node working to solve our problem.
Thus, we "chop up" the problem into small pieces we assign to each node working on our problem.
Typically, this is referred to as a problem decomposition.
<h4>Ghosts, and Their Halos</h4>
In problem decompositions, we may still need compute nodes to be aware of the work that other nodes
are currently doing, so we add an extra layer of data, referred to as a halo of ghosts.
This region of extra data can also be referred to as a domain boundary, as it is the boundary
of the compute node's owned domain of data.
We call it a halo because typically we need to know all the updates happening in the region surrounding a si
These values are called ghosts because they aren't really there: ghosts represent data another
compute node controls, and the ghost values are usually set unilaterally through communication
between compute nodes.
This ensures each compute node has up-to-date values from the node that owns the underlying data.
These updates can also be called ghost exchanges.
</details>

Overview of the Ghost Exchange Implementation
The implementations presented in these examples follow the same basic algorithm.
They each implement the same computation, and set up the same ghost exchange, we just change where computation happ

The code is controlled with the following arguments:
- `-i imax -j jmax`: set the total problem size to `imax*jmax` elements.
- `-x nprocx -y nprocy`: set the number of MPI ranks in the x and y direction, with `nprocx*nprocy` total processes
- `-h nhalo`: number of halo layers, typically assumed to be 1 for our diagrams.
- `-t (0|1)`: whether time synchronization should be performed.
- `-c (0|1)`: whether corners of the ghost halos should also be communicated.
- `-p (0|1)`: whether matrix, if small enough, should be printed. Used only for debugging.

The computation done on each data element after setup is a blur kernel, that modifies the value of a
given element by averaging the values at a 5-point stencil location centered at the given element:

`xnew[j][i] = (x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i])/5.0`

The communication pattern used is best shown in a diagram that appears in [Parallel and high performance computing,
<p>
![image](ghost_exchange2.png" \>
</p>
In this diagram, a ghost on a process is represented with a dashed outline, while owned data on a process is repres

C++ Standard Parallelism on AMD GPUs

Here are some instructions on how to compile and run some tests that exploit C++ standard parallelism, which is ava

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`

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

`## hipstdpar_saxpy_foreach example`

export HSA_XNACK=1 module load amdclang

cd ~/HPCTrainingExamples/HIPStdPar/CXX/saxpy_foreach

make export AMD_LOG_LEVEL=3 ./saxpy clean

`## hipstdpar_saxpy_transform example`

export HSA_XNACK=1 module load amdclang

cd ~/HPCTrainingExamples/HIPStdPar/CXX/saxpy_transform

make export AMD_LOG_LEVEL=3 ./saxpy clean

`## hipstdpar_saxpy_transform_reduce example`

export HSA_XNACK=1 module load amdclang

cd ~/HPCTrainingExamples/HIPStdPar/CXX/saxpy_transform_reduce

make export AMD_LOG_LEVEL=3 ./saxpy clean

`## Traveling Salesperson Problem`

#!/bin/bash

git clone https://github.com/pkestene/tsp cd tsp git checkout 51587 wget -q https://raw.githubusercontent.com/ROCm/roc-stdpar/main/data/patches/tsp/TSP.patch

patch -p1 < TSP.patch

cd stdpar

export HSA_XNACK=1 module load amdclang export STDPAR_CXX=$CXXexportROCM_GPU =$ ‘$rocminfo|grep - m1 - Egfx[^0]1|sed - e's/ * Name: * //'$‘$exportSTDPAR_TARGET =${ROCM_GPU}

export AMD_LOG_LEVEL=3 #optional

make tsp_clang_stdpar_gpu ./tsp_clang_stdpar_gpu 13 #or more...

make clean cd ../.. rm -rf tsp

`## hipstdpar_shallowwater_orig.sh`

cd ~/HPCTrainingExamples/HIPStdPar/CXX/ShallowWater_Orig

mkdir build && cd build cmake .. make ./ShallowWater

cd .. rm -rf build

`## hipstdpar_shallowwater_ver1.sh`

cd ~/HPCTrainingExamples/HIPStdPar/CXX/ShallowWater_Ver1

mkdir build && cd build cmake .. make ./ShallowWater

cd .. rm -rf build

`## hipstdpar_shallowwater_ver2.sh`

export HSA_XNACK=1 module load amdclang

cd ~/HPCTrainingExamples/HIPStdPar/CXX/ShallowWater_Ver2

make #export AMD_LOG_LEVEL=3 ./ShallowWater

make clean

hipstdpar_shallowwater_stdpar.sh

export HSA_XNACK=1 module load amdclang

cd ~/HPCTrainingExamples/HIPStdPar/CXX/ShallowWater_StdPar

make #export AMD_LOG_LEVEL=3 ./ShallowWater

make clean

Mix and Match

The examples contained in the MixandMatch directory demonstrate how to correctly combine
StdPar with other commonly used programming models, such as OpenMP and HIP.

All examples require the user to specify the path to the StdPar header in the Makefile:

module load rocm export STDPAR_PATH=${ROCM_PATH}/include/thrust/system/hip/hipstdpar export HSA_XNACK=1

Note HIPSTDPAR assumes the device is HMM enabled and setting `HSA_XNACK` to one is also required. In devices where

* omp_stdpar: demonstrates how to integrate StdPar and OpenMP within the same application.
It utilizes object-oriented programming techniques to implement the same interface in specialized ways.

* std_cpu_gpu: shows how to combine StdPar sections using `par` and `par_unseq`
to run on both the CPU and GPU within the same application.

* hip_stdpar: illustrates how to use HIP routines to allocate and transfer data to GPU buffers
for use in StdPar sections.

* atomic_stdpar_omp: explains how atomic operations can be safely performed within a StdPar
section using the `par_unseq` policy. The example also includes an equivalent OpenMP implementation.

AI and ML exercises

Last revision of this README: **April 14th 2025**.

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`

Throughout these exercises we'll be leveraging the existing ROCm installation. We can use the existing module to se

module purge module load rocm

Setting the virtual environments
These exercises include use cases for PyTorch and TensorFlow using Horovod. Let's prepare the environments to insta

We'll be leveraging the system Python installation, so we'll be creating virtual environments to add the Python pac

Let's create a virtual environment for PyTorch:

python3 -m venv –system-site-packages $HOME/venv-pt

And one for Tensorflow:

python3 -m venv –system-site-packages $HOME/venv-tf

Installing the frameworks
Let's install a PytTorch and Tensorflow suitable for the ROCm version we have available. To check the ROCm version

cat $ROCM_PATH/.info/version

Two minor versions before or after the current ROCm level should work.

Let's activate our environment for PyTorch.

source $HOME/venv-pt/bin/activate

and check the available versions:

pip install –index-url https://download.pytorch.org/whl/ torch== |& grep -o ‘[ˆ]*rocm[ˆ]*’ pip install –index-url https://download.pytorch.org/whl/ torchvision== |& grep -o ‘[ˆ]*rocm[ˆ]*’ pip install –index-url https://download.pytorch.org/whl/ torchaudio== |& grep -o ‘[ˆ]*rocm[ˆ]*’

It should yield something like:

// torch ... 2.6.0+rocm6.1, 2.6.0+rocm6.2.4, // torchvision ... 0.21.0+rocm6.1, 0.21.0+rocm6.2.4, // torchaudio ... 2.6.0+rocm6.1, 2.6.0+rocm6.2.4,

If you do not see the ROCm version you have in your system, you can find additional wheels [`here`](https://repo.ra

pip3 install torch==2.6.0 torchaudio==2.6.0 torchvision==0.21.0 -f https://repo.radeon.com/rocm/manylinux/rocm-rel-6.3.3/ –no-cache-dir

Let's do a quick smoke test to check that PyTorch can detect all GPUs:

python3 -c ‘import torch; print(“I have this many devices:”, torch.cuda.device_count())’

On an MI250, you should see `I have this many devices: 8`

Next, let's install TensorFlow in its respective environment. First, we deactivate the current environment that we

deactivate source $HOME/venv-tf/bin/activate

The latest wheel for TensorFlow with ROCm can be found [`here`](https://pypi.org/project/tensorflow-rocm/). As of A

pip install tensorflow-rocm==

Therefore, we install with:

pip3 install tensorflow-rocm==2.15.1 -f https://repo.radeon.com/rocm/manylinux/rocm-rel-6.3.3/ –no-cache-dir

Once again, let's do a quick smoke tests to see if TensorFlow detects the AMD GPUs:

python3 -c ‘from tensorflow.python.client import device_lib ; device_lib.list_local_devices()’

On MI250, it should show something like this:

2025-04-14 13:48:21.243911: I tensorflow/core/common_runtime/gpu/gpu_device.cc:2021] Created device /device:GPU:0 with 63828 MB memory: -> device: 0, name: AMD Instinct MI250X/MI250, pci bus id: 0000:29:00.0 2025-04-14 13:48:21.500504: I tensorflow/core/common_runtime/gpu/gpu_device.cc:2021] Created device /device:GPU:1 with 63828 MB memory: -> device: 1, name: AMD Instinct MI250X/MI250, pci bus id: 0000:2c:00.0 2025-04-14 13:48:21.748008: I tensorflow/core/common_runtime/gpu/gpu_device.cc:2021] Created device /device:GPU:2 with 63828 MB memory: -> device: 2, name: AMD Instinct MI250X/MI250, pci bus id: 0000:2f:00.0 2025-04-14 13:48:21.994639: I tensorflow/core/common_runtime/gpu/gpu_device.cc:2021] Created device /device:GPU:3 with 63828 MB memory: -> device: 3, name: AMD Instinct MI250X/MI250, pci bus id: 0000:32:00.0 2025-04-14 13:48:22.242978: I tensorflow/core/common_runtime/gpu/gpu_device.cc:2021] Created device /device:GPU:4 with 63828 MB memory: -> device: 4, name: AMD Instinct MI250X/MI250, pci bus id: 0000:ad:00.0 2025-04-14 13:48:22.489896: I tensorflow/core/common_runtime/gpu/gpu_device.cc:2021] Created device /device:GPU:5 with 63828 MB memory: -> device: 5, name: AMD Instinct MI250X/MI250, pci bus id: 0000:b0:00.0 2025-04-14 13:48:22.736439: I tensorflow/core/common_runtime/gpu/gpu_device.cc:2021] Created device /device:GPU:6 with 63828 MB memory: -> device: 6, name: AMD Instinct MI250X/MI250, pci bus id: 0000:b3:00.0 2025-04-14 13:48:22.982904: I tensorflow/core/common_runtime/gpu/gpu_device.cc:2021] Created device /device:GPU:7 with 63828 MB memory: -> device: 7, name: AMD Instinct MI250X/MI250, pci bus id: 0000:b6:00.0

We are interested in using Horovod with TensorFlow, so let's install it. Horovod build system was not made ready to

mkdir -p $HOME/cmake cat > $HOME/cmake/cmake « EOF #!/bin/bash -e

if [[“$@” == *“–build”*]] ; then $(which cmake) $@ else $(which cmake) - DCMAKE_{M}ODULE_{P}ATH$ =ROCM_PATH/lib/cma
$@ fi EOF chmod +x $HOME/cmake/cmake

We can now build using our tuned cmake script:

module load rocm module load openmpi

PATH=$HOME/cmake$:PATH
CPATH=$ROCM_{P}ATH/include/rccl\ HOROVOD_{W}ITHOUT_{M}XNET = 1\ HOROVOD_{W}ITHOUT_{G}LOO =$
$1\ HOROVOD_{G}PU = ROCM\ \ HOROVOD_{R}OCM_{H}OME$ =ROCM_PATH
HOROVOD_GPU_OPERATIONS=NCCL
HOROVOD_CPU_OPERATIONS=MPI
HOROVOD_WITH_MPI=1
HOROVOD_ROCM_PATH=$ROCM_{P}ATH\ \ HOROVOD_{R}CCL_{H}OME$ =ROCM_PATH/include/rccl
HOROVOD_RCCL_LIB=$ROCM_PATH/lib
HCC_AMDGPU_TARGET=gfx90a,gfx942
HOROVOD_WITH_TENSORFLOW=1
HOROVOD_WITHOUT_PYTORCH=1
pip install –no-cache-dir –force-reinstall –verbose horovod==0.28.1

Let's define a work directory for us to try some examples.

mkdir -p $HOME/ai-with-rocm

PyTorch MNIST example

MNIST is a quite popular data set for computer vision training. We are fortunate that are many examples on the inte

Let's take one of the PyTorch official examples for this - we are training just two epochs:

cd $HOME/ai-with-rocm

deactivate source $HOME/venv-pt/bin/activate

curl -LO https://raw.githubusercontent.com/pytorch/examples/main/mnist/main.py module load rocm
python -u main.py –epochs 2 –batch-size 256

You may get an MIOpen error saying:

MIOpen(HIP): Error [FlushUnsafe] File is unwritable: “/tmp/gfx90a68.HIP.3_3_0_d22d5a13f-dirty.ufdb.txt”

In that case, the following commands should fix it:

mkdir -p $HOME/tmpdirexportTMPDIR$ =HOME/tmpdir rm -rf $HOME/tmpdir/* echo “TMPDIR set
to” $TMPDIR

We can control which GPU to use with the environmental variable ROCR_VISIBLE_DEVICES and can use the `rocprofv3` to

ROCR_VISIBLE_DEVICES=2
rocprofv3 –stats –kernel-trace – python -u main.py –epochs 1 –batch-size 256

The resulting `*.csv` files show the different GPU kernels invoked for this application.

PyTorch MNIST example - distributed

We might now be interested in distributing our training across devices. A way to accomplish this is by taking a dis

There are also several examples on how to do this. We will use the following code as starting example:

cd $HOME/ai-with-rocm curl -LO https://raw.githubusercontent.com/kubeflow/examples/master/pytorch_mnist/training/ddp

```
Download a modified version of this script and compare the differences:
```

curl -LO https://raw.githubusercontent.com/amd/HPCTrainingExamples/main/MLExamples/mnist_DDP_modified.py
vimdiff mnist_DDP.py mnist_DDP_modified.py

```
There are a couple of differences: one controls the batch size another one controls how the distributed run is init
```

dist.init_process_group(backend='nccl', init_method='env://', world_size=int(os.environ['WORLD_SIZE']),
rank=int(os.environ['RANK']))

```
PyTorch provides an object to control the distributed run environment:
```

import torch.distributed as dist

```
Here we are instructing that we want to use RCCL (AMD implementation for NCCL) and also want the tool to leverage t

Other relevant bits to enable distributed run are in:
```

def run(modelpath, gpu): ... model = Net() ... model = torch.nn.parallel.DistributedDataParallel(model)
... optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5) ...

```
Here the model is wrapped into the `DistributedDataParallel` object to enable the distributed training.

Now to train the model on multiple ranks we will leverage MPI to start the processes and translate the MPI environm
```

cat > run-me.sh « EOF #!/bin/bash -e export MASTER_ADDR=localhost export MASTER_PORT=29500
export WORLD_SIZE=$OMPI_COMM_WORLD_SIZE export RANK=$OMPI_COMM_WORLD_RANK
export ROCR_VISIBLE_DEVICES=$OMPI_COMM_WORLD_LOCAL_RANK

python -u mnist_DDP_modified.py
–gpu –modelpath $HOME/ai-with-rocm/model

EOF chmod +x run-me.sh module load rocm module load openmpi mpirun -np 2 ./run-me.sh

```
Master address and port are defined for the different ranks to communicate between themselves. We then leverage the

Another popular way to spin a distributed run is to leverage the `torchrun` utility. However, this requires the app
```

torch.cuda.set_device(int(os.environ['RANK']))

```
This will make GPUs to be indexed by the rank.

We can then run our application with `torchrun` as:
```

ROCR_VISIBLE_DEVICES=0,1
torchrun –nnodes 1 –nproc_per_node 2
./mnist_DDP_modified.py –gpu –modelpath $HOME/ai-with-rocm/model

```
## TensorFlow with Horovod example

Similarly to PyTorch, TensorFlow examples should not need changes due to the GPU architecture.

Let's pick a TensorFlow example from the Horovod project - a synthectic training example already rigged to use Horo
```

deactivate source $HOME/venv-tf/bin/activate cd $HOME/ai-with-rocm

curl -LO https://raw.githubusercontent.com/horovod/horovod/master/examples/tensorflow2/tensorflow2_synthetic_benchmar

module load rocm module load openmpi mpirun -np 2
python -u tensorflow2_synthetic_benchmark.py –batch-size 256

```
You should see an output similar to this one:
```

Running benchmark... Iter #0: 559.4 img/sec per GPU Iter #1: 557.7 img/sec per GPU Iter #2: 559.2
img/sec per GPU Iter #3: 552.1 img/sec per GPU Iter #4: 553.6 img/sec per GPU Iter #5: 551.0 img/sec

per GPU Iter #6: 550.9 img/sec per GPU Iter #7: 553.8 img/sec per GPU Iter #8: 553.4 img/sec per GPU Iter #9: 546.8 img/sec per GPU Img/sec per GPU: 553.8 +-7.4 Total img/sec on 2 GPU(s): 1107.6 +-14.9

`If you encounter this runtime error:`

To enable the following instructions: SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags. [1744644100.553280] [024ac696037d:188662:0] ib_iface.c:1230 UCX ERROR mlx5_0: ibv_create_cq(cqe=4096) failed: Cannot allocate memory : Please set max locked memory (ulimit -l) to 'unlimited' (current: 64 kbytes)

`Then export the following variable to fix it:`

export UCX_TLS=self,shm

`Horovod makes it rather straightforward to start a distributed learning model as it leverages the already existing`

`We can inspect the test case and see the relevant bits where Horovod `hvd` is being leveraged, namely the selection`

vi tensorflow2_synthetic_benchmark.py

`And inspect the file, paying particular attention to the following lines of code:`

import tensorflow as tf import horovod.tensorflow as hvd ... tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU') ... with tf.GradientTape() as tape: ... tape = hvd.DistributedGradientTape(tape, compression=compression)

`We can further inspect the GPU activity by leveraging `rocprofv3` to obtain a trace for one of the ranks. We'll do`

mpirun -np 2
bash -c 'if [$OMPI_COMM_WORLD_RANK -eq 1] ; then
profiler="rocprofv3 –hip-trace –output-format pftrace –" ;
fi ;
$profiler python -u tensorflow2_synthetic_benchmark.py
–batch-size 256
–num-warmup-batches 2
–num-iters 2'

`A directory named with a hash will be created, inside of which the two trace files for each of the MPI processes wi`

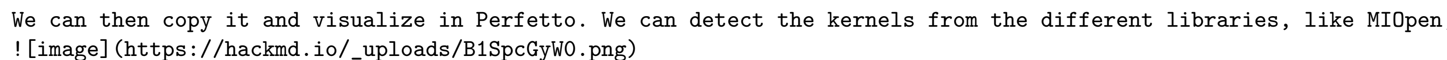
193546_results.pftrace 193556_results.pftrace

`We can also concatenate these trace files to be visualized in a single one:`

cat *.pftrace > perfetto_trace.pftrace

`Then, we can visualize the trace file using the [`Perfetto`](https://www.ui.perfetto.dev/) tool. It is often a good`

xz -T8 -9 perfetto_trace.pftrace



`We can then copy it and visualize in Perfetto. We can detect the kernels from the different libraries, like MIOpen,`
`![image](https://hackmd.io/_uploads/B1SpcGyW0.png)`

`## Examples with Huggingface transformers`

`There are several repositories of examples. A popular one is the Huggingface transformers. These examples should ju`

`We can install the transformer package from source as:`

deactivate source $HOME/venv-tf/bin/activate

git clone
https://github.com/huggingface/transformers.git
$HOME/ai-with-rocm/transformers cd $HOME/ai-with-rocm/transformers pip3 install -e .

`It is useful to point the implementation to a suitable place to store and cache information and datasets. That can`

export HF_HOME=$HOME/ai-with-rocm/hf-homeexportHUGGINGFACE_{H}UB_{C}ACHE$ =HOME/ai-with-rocm/hf-cache mkdir -p $HF_HOME $HUGGINGFACE_HUB_CACHE

```
We can now try the examples.

### Image classification
Let's look at the image classification one. We need first to install the example dependencies:
```

cd $HOME/ai-with-rocm/transformers/examples/pytorch/image-classification pip3 install -r requirements.txt
pip3 install scikit-learn pip3 install -U pillow

```
We are now ready to run the example. We'll be experimenting with using mixed precision (with BF16 datatypes) or not
```

for precision in ' ' '–bf16' ; do ROCR_VISIBLE_DEVICES=2
python3 run_image_classification.py
–dataset_name beans
–label_column_name labels
–output_dir $HOME/ai-with-rocm/hf-output
–overwrite_output_dir
–remove_unused_columns False
–do_train
–learning_rate 2e-5
–num_train_epochs 2
–per_device_train_batch_size 8
–torch_compile True
–seed 1337
$precision done

```
Depending on what is bounding the performance it could be good idea to use mixed precision or not - so this is a fa
```

... # No mixed precision ***** train metrics ***** epoch = 2.0 total_flos = 149248978GF train_loss = 0.4118 train_runtime = 0:01:17.52 train_samples_per_second = 26.675 train_steps_per_second = 3.354 ... # With mixed precision (BF16) ***** train metrics ***** epoch = 2.0 total_flos = 149248978GF train_loss = 0.4126 train_runtime = 0:01:23.73 train_samples_per_second = 24.698 train_steps_per_second = 3.105

```
### Language modeling
A growing set of applications for distributed learning is language modeling. A typical approach is to leverage an e
```

cd $HOME/ai-with-rocm/transformers/examples/pytorch/question-answering pip3 install -r requirements.txt

```
Then, we are ready to run the fine-tuning, e.g., on 2 GPUs for different training precisions:
```

for precision in ' ' '–bf16' '–fp16' ; do ROCR_VISIBLE_DEVICES=0,1
torchrun –nproc_per_node 2 run_qa.py
–model_name_or_path bert-base-uncased
–dataset_name squad
–do_train
–per_device_train_batch_size 12
–learning_rate 3e-5
–num_train_epochs 1
–max_seq_length 384
–doc_stride 128
–output_dir $HOME/ai-with-rocm/hf-output2
–torch_compile True
–overwrite_output_dir
$precision done

FP32

***** train metrics ***** epoch = 1.0 total_flos = 16159030GF train_loss = 1.3039 train_runtime = 0:10:09.70 train_samples = 88524 train_samples_per_second = 145.192 train_steps_per_second = 6.051

BF16 ***** train metrics ***** epoch = 1.0 total_flos = 16159030GF train_loss = 1.3013 train_runtime = 0:07:34.81 train_samples = 88524 train_samples_per_second = 194.636 train_steps_per_second = 8.111

FP16 ***** train metrics ***** epoch = 1.0 total_flos = 16159030GF train_loss = 1.3064 train_runtime = 0:06:35.75 train_samples = 88524 train_samples_per_second = 223.686 train_steps_per_second = 9.322 "'

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
2       breakpoint     keep y   0x000000000020b350 in main() at /HPCTrainingExamples/HIP/saxpy/saxpy.hip
```

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
* 1  A     AMDGPU Agent (GPUID 64146) gfx90a       Aldebaran/MI200 [Instinct MI250X/MI250] 416   3328
```

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*,
```

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

ROCm™ Systems Profiler aka `rocprof-sys`

NOTE: extensive documentation on how to use `rocprof-sys` for the GhostExchange examples is also available as `README.md` in this exercises repo. Here, we show how to use `rocprof-sys` tools considering the example in HPCTrainingExamples/HIP/jacobi.

In this series of examples, we will demonstrate profiling with `rocprof-sys` on a platform using an AMD Instinct™ MI250X GPU. ROCm 6.3.2 release includes the `rocprofiler-systems` packge that you can install.

Note that the focus of this exercise is on `rocprof-sys` profiler, not on how to achieve optimal performance on MI250X.

First, start by cloning HPCTrainingExamples repository and loading ROCm:

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

Environment setup

For this training, one requires recent ROCm (>=6.3) which contains `rocprof-sys`, as well as an MPI installation.

```
module load rocm/6.3.2
module load openmpi
```

Build and run

No profiling yet, just check that the code compiles and runs correctly.

```
cd HPCTrainingExamples/HIP/jacobi
make
mpirun -np 1 ./Jacobi_hip -g 1 1
```

The above run should show output that looks like this:

```
Topology size: 1 x 1
Local domain size (current node): 4096 x 4096
Global domain size (all nodes): 4096 x 4096
Rank 0 selecting device 0 on host TheraC63
Starting Jacobi run.
Iteration:   0 - Residual: 0.022108
Iteration: 100 - Residual: 0.000625
Iteration: 200 - Residual: 0.000371
Iteration: 300 - Residual: 0.000274
Iteration: 400 - Residual: 0.000221
Iteration: 500 - Residual: 0.000187
Iteration: 600 - Residual: 0.000163
Iteration: 700 - Residual: 0.000145
Iteration: 800 - Residual: 0.000131
Iteration: 900 - Residual: 0.000120
Iteration: 1000 - Residual: 0.000111
Stopped after 1000 iterations with residue 0.000111
Total Jacobi run time: 1.2876 sec.
Measured lattice updates: 13.03 GLU/s (total), 13.03 GLU/s (per process)
Measured FLOPS: 221.51 GFLOPS (total), 221.51 GFLOPS (per process)
Measured device bandwidth: 1.25 TB/s (total), 1.25 TB/s (per process)
```

`rocprof-sys` config

First, generate the `rocprof-sys` configuration file, and ensure that this file is known to `rocprof-sys`.

```
rocprof-sys-avail -G ~/.rocprofsys.cfg
export ROCPROFSYS_CONFIG_FILE=~/.rocprofsys.cfg
```

Second, inspect configuration file, possibly changing some variables. For example, one can modify the following lines:

```
ROCPROFSYS_PROFILE                                  = true
ROCPROFSYS_USE_ROCTX                                = true
ROCPROFSYS_SAMPLING_CPUS                            = 0
```

You can see what flags can be included in the config file by doing:

```
rocprof-sys-avail --categories rocprofsys
```

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

```
rocprof-sys-avail -bd --categories rocprofsys
```

Note that the list of flags displayed by the commands above may not include all actual flags that can be set in the config. For a full list of options, please read the rocprof-sys documentation.

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

```
rocprof-sys-avail -G ~/rocprofsys_all.cfg --all
```

Instrument application binary

You can instrument the binary, and inspect which functions were instrumented (note that you need to change `<TIMESTAMP>` according to your generated folder path).

```
rocprof-sys-instrument -o ./Jacobi_hip.inst -- ./Jacobi_hip
for f in $(ls rocprofsys-Jacobi_hip.inst-output/<TIMESTAMP>/instrumentation/*.txt); do echo $f; cat $f; echo "#####
```

Currently `rocprof-sys` will instrument by default only the functions with >1024 instructions, so you may need to change it by using `-i #inst` or by adding `--function-include function_name` to select the functions you are interested in. Check more options using `rocprof-sys-instrument --help` or by reading the rocprof-sys documentation.

Let's instrument the most important Jacobi kernels.

```
rocprof-sys-instrument --function-include 'Jacobi_t::Run' 'JacobiIteration' -o ./Jacobi_hip.inst -- ./Jacobi_hip
```

The output should show that only these functions have been instrumented:

```
...
[rocprof-sys][exe] Finding instrumentation functions...
[rocprof-sys][exe]    1 instrumented funcs in JacobiIteration.hip
[rocprof-sys][exe]    1 instrumented funcs in JacobiRun.hip
[rocprof-sys][exe]    1 instrumented funcs in Jacobi_hip
...
```

This can also be verified with:

```
$ cat rocprofsys-Jacobi_hip.inst-output/<TIMESTAMP>/instrumentation/instrumented.txt

  StartAddress   AddressRange  #Instructions  Ratio Linkage Visibility  Module                     Function
      0x226440            332             71   4.68  unknown    unknown JacobiIteration.hip        JacobiIteration
      0x224ad0            677            146   4.64  unknown    unknown JacobiRun.hip              Jacobi_t::Run
      0x226370            205             38   5.39  unknown    unknown Jacobi_hip                 __device_stub__
```

Run instrumented binary

Now that we have a new application binary where the most important functions are instrumented, we can profile it using `rocprof-sys-run` under the `mpirun` environment.

```
mpirun -np 1 rocprof-sys-run -- ./Jacobi_hip.inst -g 1 1
```

Check the command line output generated by `rocprof-sys-run` , it contains some useful overviews and **paths to generated files**. Observe that the overhead to the application runtime is small. If you had previously set `ROCPROFSYS_PROFILE=true` , inspect `wall_clock-0.txt` 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`).

In many cases, simply checking the wall_clock files might be sufficient for your profiling!

If it is not, continue by visualizing the trace.

Visualizing traces using `Perfetto`

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

Click `Open trace file` and select the `perfetto-trace-0.proto` file. Below, you can see an example of how the trace file would be visualized on `Perfetto` :

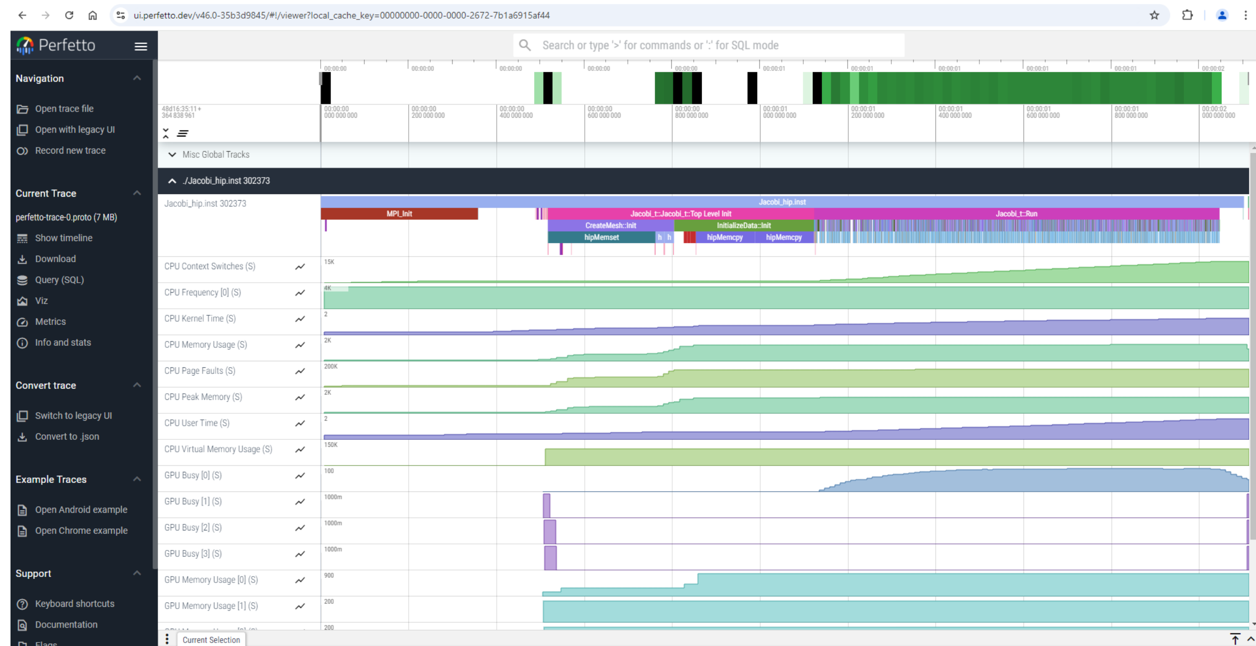


Figure 1: jacobi_hip-perfetto_screenshot

If there is an error opening trace file, try using an older `Perfetto` version, e.g., by opening the web page https://ui.perfetto.dev/v46.0-35b3d9845/#!/.

Additional features

Flat profiles

Append advanced option `ROCPROFSYS_FLAT_PROFILE=true` to `~/.rocprofsys.cfg` or prepend it to the `mpirun` command:

```
ROCPROFSYS_FLAT_PROFILE=true mpirun -np 1 rocprof-sys-run -- ./Jacobi_hip.inst -g 1 1
```

`wall_clock-0.txt` file now shows overall time in seconds for each function.

Note the significant total execution time for `hipMemcpy` and `Jacobi_t::Run` calls.

Hardware counters

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

```
rocprof-sys-avail --all
```

Select the counter you are interested in, and then declare them in your configuration file (or prepend to your `mpirun` command):

```
ROCPROFSYS_ROCM_EVENTS = VALUUtilization,FetchSize
```

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 `rocprof-sys`.

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.

```
ROCPROFSYS_ROCM_EVENTS=VALUUtilization,FetchSize mpirun -np 1 rocprof-sys-run -- ./Jacobi_hip.inst -g 1 1
cat rocprof-sys-Jacobi_hip.inst-output/<TIMESTAMP>/rocprof-device-0-VALUUtilization-0.txt
```

Sampling

To reduce the overhead of profiling, one can use call stack sampling. Set the following in your configuration file (or prepend to your `mpirun` command):

```
ROCPROFSYS_USE_SAMPLING = true
ROCPROFSYS_SAMPLING_FREQ = 100
```

Execute the instrumented binary, inspect `sampling*` files and visualize the `Perfetto` trace:

```
mpirun -np 1 rocprof-sys-run -- ./Jacobi_hip.inst -g 1 1
ls rocprofsys-Jacobi_hip.inst-output/<TIMESTAMP>/* | grep sampling
```

Profiling multiple MPI processes

Run the instrumented binary with multiple MPI ranks. Note separate output files for each rank, including `perfetto-trace-*.proto` and `wall_clock-*.txt` files.

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

Inspect output text files. Then visualize `perfetto-trace-*.proto` files in `Perfetto`. Note that one can merge multiple trace files into a single one using simple concatenation:

```
cat perfetto-trace-*.proto > merged.proto
```

Next steps

Try to use `rocprof-sys` to profile GhostExchange examples.

Finally, try to profile your own application!

Stream Overlap Example

This example is based on example 2 from Chapter 6 of the HIP Book: "Accelerated Computing with HIP", by Yifan Sun, Trinayan Baruah, and David R. Kaeli. The example demonstrates how to overlap data transfer and computation using HIP streams. The included directories step through different versions of the example.

Each directory contains a `README.md` file that includes a description of the version and instructions for building and running the example.

This multi-streamed example is traced with ROCm Systems Profiler, formerly known as Omnitrace. ROCm Systems Profiler is now available in ROCm 6.2.0+ version package directly and does not need to be installed separately anymore. The figures included in the `figs` directory, however, are generated using `Omnitrace v.1.11.3`. The command line trace instructions are included in the `README.md` file in each directory.

Folder `0-Orig`

This is the original version of the example. It demonstrates the basic structure of the example and provides a starting point for the other versions. The memory copies and kernel execution are done together sequentially in each of the multiple streams.

Folder `1-split-copy-compute-hw-queues`

This version of the example splits the host to device (and vice versa) memory copies and the kernel execution into separate loops over multiple streams. This allows for overlap of memory copies across across multiple streams in addition to overlap of kernel computations over multiple streams. This also enables overlap of data copies and kernel computations.

This example also exploits the environment variable controlling the GPU maximum hardware queues (`GPU_MAX_HW_QUEUES`) to achieve better performance for a multi-streamed application.

Folder `2-pageable-mem`

This version of the example uses *pageable* memory for data transfers instead of pinned memory. This example is to demonstrate how pageable memory degrades performance of a multi-streamed application. Ideally, pinned memory should be used for data transfers in a multi-streamed application wherever possible (depending on available memory resources).

Self-guided tour of the Stream Overlap example

The interested reader can follow these steps sequentially to understand the performance implications of use of multiple streams to overlap data transfers and kernel computations. The results shared in folder `figs` are obtained from running the example on an AMD Instinct MI250 single GCD.

1. Build the baseline example in `0-Orig` directory. The build and run instructions can be found in the `README.md` file in the directory.
2. Then run the example using multiple streams. Specifically choose 1, 2, and 4 streams and observe the performance improvements. Specifically note if the reduction in runtime scales linearly with the number of streams. See the figures in `figs/streams[1,2,4]_noQ_seq_copy.png` for reference.
3. Increase the number of streams to 8 and observe the performance degradation. This is because the GPU has a limited hardware resources and increasing the number of streams beyond the GPU's capability will degrade performance. See the figure in `figs/streams8_noQ_seq_copy.png` for reference.
4. Switch to `1-split-copy-compute-hw-queues` directory and build and run the example. Observe the performance improvements if any. Ideally, the performance improvement is only marginal. See the figure in `figs/streams8_noQ_split_copy.png` for reference.
5. Set the environment variable `GPU_MAX_HW_QUEUES` to 8 and observe the performance improvements. This is because the default number of hardware queues is 4. Increasing the number of hardware queues will improve the performance of a multi-streamed application, especially when the number of streams is more than the default number of hardware queues. Note that, the performance improvement is possible if the GPU resource is not yet fully saturated, for example, with limited register

pressure, or limited shared memory usage. The performance improvement is clearly visible in the figure `figs/streams8_Q_split_copy.png` .

6. [Optional] Switch to `2-pageable-mem` directory and build and run the example. Observe the performance degradation due to use of pageable memory for data transfers. Ideally, pinned memory should be used for data transfers in a multi-streamed application
7. Repeat the above steps for a different GPU and observe the performance implications.

OmniperfExamples

In this directory, users can find a variety of examples aimed at showcasing some of the most important features of Omniperf. For each example, an initial implementation labeled `problem` will be modified in order to show an improvement in performance using the Omniperf tools. The improved implementation will be reffered to as `solution` . Please refer to the single sub-directories `README.md` files for details.

Exercise 1: Launch Parameter Tuning

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

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 the `archive` 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

Results on MI210

In this section, we show results obtained running this exercise 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` .**

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	image	image
FP16/INT8	image	image

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 ms
```

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 |
-------------------------------------------------------------------------------------
```

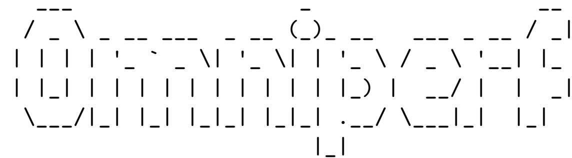
Looking through this data we see: - Workgroup Size (`7.1.1`) corresponds to the first argument of the block launch parameter - Total Wavefronts (`7.1.2`) corresponds to the first index of the grid launch parameter - Grid size (`7.1.0`) is Workgroup Size (`7.1.1`) times Total Wavefronts (`7.1.2`)

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 7.1.0 7.1.1 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 (-9
|    |  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                   |    Ma
--------------|-------------------|--------|----------------------|------------|--------|----------------------|-------
| 7.1.0        | Grid Size         | 256.00 | 131072.0 (51100.0%)  |  130816.00 | 256.00 | 131072.0 (51100.0%)  | 256.0
--------------|-------------------|--------|----------------------|------------|--------|----------------------|-------
| 7.1.1        | Workgroup Size    |  64.00 | 64.0 (0.0%)          |       0.00 |  64.00 | 64.0 (0.0%)          |  64.0
--------------|-------------------|--------|----------------------|------------|--------|----------------------|-------
| 7.1.2        | Total Wavefronts  |   4.00 | 2048.0 (51100.0%)    |    2044.00 |   4.00 | 2048.0 (51100.0%)    |   4.0
-------------------------------------------------------------------------------------------------------------------------------------
```

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 |
-----|------------------------------------------|---------|---------------|---------------|---------------|--------|-
|  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	image	image
FP16/INT8	image	image

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	image	image
FP16/INT8	image	image

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.

Results on MI300A

In this section, we show results obtained running this exercise on a system with MI300A, using ROCm `6.2.1` and the associated Omniperf, version `6.2.1` .

Roofline Analysis:

At present (September 28th 2024), rooflines are disabled on MI300A.

Exercise Instructions:

As for the MI210 case, build and run the problem code:

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

(*simulated output*)

```
yAx time: 540 ms
```

Once again, we launch the following command:

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

Followed by:

```
omniperf analyze -p workloads/problem/MI300A_A1 --dispatch 1 --block 7.1.0 7.1.1 7.1.2
```

Then inspect the output:

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

   INFO Analysis mode = cli
   INFO [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 | 541264224.00 | 541264224.00 | 541264224.00 | 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] |         4 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
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 |
--------------------------------------------------------------------------
```

As for the MI210 case, the workgroup size is 64 and the number of Wavefronts launched is 4.

To see improved performance, we turn to the code in the `solution` directory:

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

(*simulated output*)

```
yAx time: 9.7 ms
```

For the MI210 case, the compute time was about 42 times smaller when going from `problem` to `solution` . For the MI300A case, we see it is about 70 times smaller.

To visually confirm the updated launch parameters in the `solution` code, run:

```
omniperf profile -n solution --no-roof -- ./solution.exe
omniperf analyze -p workloads/solution/MI300A_A1 --dispatch 1 --block 7.1.0 7.1.1 7.1.2
```

Then see the number of Wavefronts now being 2048:

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

   INFO Analysis mode = cli
   INFO [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 | 9482864.00 | 9482864.00 |   9482864.00 | 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] |        4 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
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 |
-----------------------------------------------------------------------------------
```

Omniperf Command Line Comparison Feature:

We can compare the performance of `problem` and `solution` using `omniperf analyze` :

```
omniperf analyze -p workloads/problem/MI300A_A1/ -p solution/workloads/solution/MI300A_A1/ --dispatch 1 --block 7.1
```

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

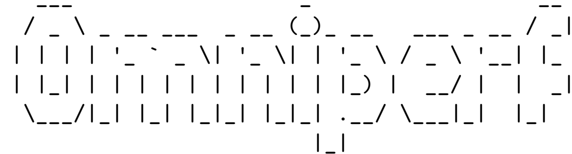
```
                       |_|

   INFO Analysis mode = cli
   INFO [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 | 541264224.00 | 9482864.0 (-98
|    |  double*) [clone .kd]                      |         |            |            |              |
--------------------------------------------------------------------------------------------------------------------------------
0.2 Dispatch List
-------------------------------------------------------------------------------------------------
|    |   Dispatch_ID | Kernel_Name                                                    |   GPU_ID |
-----|---------------|----------------------------------------------------------------|----------
|  0 |             1 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        4 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
7. Wavefront
7.1 Wavefront Launch Stats
--------------------------------------------------------------------------------------------------------------------------------
| Metric_ID     | Metric            |    Avg | Avg                   |   Abs Diff |    Min | Min                   |    Ma
--------------|-------------------|--------|-----------------------|------------|--------|-----------------------|------
| 7.1.0         | Grid Size         | 256.00 | 131072.0 (51100.0%)  |  130816.00 | 256.00 | 131072.0 (51100.0%)  | 256.0
--------------|-------------------|--------|-----------------------|------------|--------|-----------------------|------
| 7.1.1         | Workgroup Size    |  64.00 | 64.0 (0.0%)           |       0.00 |  64.00 | 64.0 (0.0%)           |  64.0
--------------|-------------------|--------|-----------------------|------------|--------|-----------------------|------
| 7.1.2         | Total Wavefronts |   4.00 | 2048.0 (51100.0%)     |    2044.00 |   4.00 | 2048.0 (51100.0%)     |   4.0
--------------------------------------------------------------------------------------------------------------------------------
```

Note that the new execution time for `solution` is about 1.75% of the original execution time for `problem`
.

More Kernel Filtering:

Run the following command to once again see a ranking of the top kernels that take up most of the kernel runtime:

```
cd ..
omniperf analyze -p workloads/problem/MI300A_A1/ --list-stats
```



```
   INFO Analysis mode = cli
   INFO [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] |        4 |
-----|---------------|---------------------------------------------------------------|-----------
|  1 |             1 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        4 |
------------------------------------------------------------------------------------------------
```

To see aggregated stats for the `yax` kernel, run

```
omniperf analyze -p workloads/problem/MI300A_A1/ -k 0 --block 7.1.0 7.1.1 7.1.2
```

Which will show an output similar to this one:

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

   INFO Analysis mode = cli
   INFO [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, |    2.00 | 1083496775.00 | 541748387.50 | 541748387.50 | 100.00 |
|    |  double*) [clone .kd]                    |         |               |              |              |        |
------------------------------------------------------------------------------------------------------------------------------
0.2 Dispatch List
------------------------------------------------------------------------------------------------
|    |   Dispatch_ID | Kernel_Name                                                   |   GPU_ID |
-----|---------------|---------------------------------------------------------------|-----------
|  0 |             0 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        4 |
-----|---------------|---------------------------------------------------------------|-----------
|  1 |             1 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        4 |
------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
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 |
```

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.

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

```
      <li><strong>FP(32/16):</strong> 32- or 16-bit Floating Point numeric types</li>
      <li><strong>FLOPs:</strong> Floating Point Operations Per second</li>
      <li><strong>HBM:</strong> High Bandwidth Memory is globally accessible from the GPU, and is a level of memory
 </ul>
```

Results on MI210

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`**

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 |image|image | |FP16/INT8 |image|image |

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: | Roofline Type | Roofline Legend | Roofline Plot | |——————|——————————————————————————|——————————————————————————————————| |FP32/FP64 |image|image | |FP16/INT8 |image|image |

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

Roofline Comparison

Roofline Type	Problem Roofline	Solution Roofline
FP32/FP64	image	image
FP16/INT8	image	image

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.

Results on MI300A

In this section, we show results obtained running this exercise on a system with MI300A, using ROCm `6.2.1` and the associated Omniperf, version `6.2.1` .

Roofline Analysis:

At present (September 28th 2024), rooflines are disabled on MI300A.

As for the MI210 case, build and run the problem code:

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

(*simulated output*)

```
yAx time: 7.27 ms
```

Unlike the MI210 case, the runtime of `problem` is already smaller than it was for the previous `solution` on example `1-LaunchParameters` .

Once again, we launch the following command to collect complete profiling data for analysis:

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

Followed by:

```
omniperf analyze -p workloads/problem/MI300A_A1 --dispatch 1 --block 2.1.15 6.2.7
```

Then inspect the output:

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

   INFO Analysis mode = cli
   INFO [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 | 7241298.00 | 7241298.00 |   7241298.00 | 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] |        4 |
-----------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
-------------------------------------------------------------------------------------
| Metric_ID   | Metric               |    Avg | Unit       |    Peak |   Pct of Peak |
--------------|----------------------|--------|------------|---------|----------------
| 2.1.15      | Wavefront Occupancy  | 177.86 | Wavefronts | 7296.00 |          2.44 |
-------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
----------------------------------------------------------------------
| Metric_ID   | Metric               |   Avg |   Min |   Max | Unit   |
--------------|----------------------|-------|-------|-------|---------
| 6.2.7       | Insufficient CU LDS  | 58.11 | 58.11 | 58.11 | Pct    |
----------------------------------------------------------------------
```

The results are similar to the MI210 case, in terms of Wavefront Occupancy (`2.44%` , for MI210 it was `3.10%`) and Insufficient CU LDS (around `58%` , for MI210 it was `79%`). Let us look first at the solution that completely eliminates LDS usage:

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

(*simulated output*)

```
yAx time: 9.79 ms
```

As in the MI210 case, completely eliminating LDS usage makes the runtime worse.

Let's run the following commands and inspect the output:

```
omniperf profile -n solution --no-roof -- ./solution.exe
omniperf analyze -p workloads/solution/MI300A_A1/ --dispatch 1 --block 2.1.15 6.2.7
```

Output:

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

   INFO Analysis mode = cli
   INFO [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 │ 9484503.00 │ 9484503.00 │   9484503.00 │ 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] │        4 │
╘════╧═══════════════╧═══════════════════════════════════════════════════════════════╧══════════╛


--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
╒═════════════╤═════════════════════╤════════╤════════════╤═════════╤═══════════════╕
│ Metric_ID   │ Metric              │    Avg │ Unit       │    Peak │   Pct of Peak │
╞═════════════╪═════════════════════╪════════╪════════════╪═════════╪═══════════════╡
│ 2.1.15      │ Wavefront Occupancy │ 437.16 │ Wavefronts │ 7296.00 │          5.99 │
╘═════════════╧═════════════════════╧════════╧════════════╧═════════╧═══════════════╛


--------------------------------------------------------------------------------
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    │
╘═════════════╧═════════════════════╧═══════╧═══════╧═══════╧════════╛
```

From the ouput above, we see that Insufficient CU LDS is now zero as expected, and that Wavefront Occupancy has gone up to around `6%` from `2.44%` that it was before for MI210. Next, let's compare

these results with the code in the `solution` directory: this implementation reduces the amount of LDS requested to address the occupancy limit, but still uses some LDS to speed up memory accesses. First, run:

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

(*simulated output*)

```
yAx time: 5.80 ms
```

This shows that an appropriate reduction of LDS usage did improve the performance of the example. To see the specific values of the metrics of interest, we run:

```
omniperf profile -n solution --no-roof -- ./solution.exe
omniperf analyze -p workloads/solution/MI300A_A1 --dispatch 1 --block 2.1.15 6.2.7
```

With output:

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

   INFO Analysis mode = cli
   INFO [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 | 5766574.00 | 5766574.00 |   5766574.00 | 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] |        4 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
-------------------------------------------------------------------------------------
| Metric_ID   | Metric               |    Avg | Unit       |    Peak |   Pct of Peak |
--------------|----------------------|--------|------------|---------|----------------
| 2.1.15      | Wavefront Occupancy | 421.57 | Wavefronts | 7296.00 |          5.78 |
-------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
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    |
---------------------------------------------------------------------
```

We see that the example is still not occupancy limited by LDS allocations (Insufficient CU LDS is zero). The Wavefront Occupancy has remained approximately the same. As seen above, the runtime has improved by approximately `20%` (going from `7.27` ms of `problem.exe`, to the current time of `5.8` ms).

Exercise 3: Register Occupancy Limiter

More complex yAx implementation to demonstrate a register limited kernel using an innocuous looking function call. The register limit no longer shows up for recent versions of ROCm on certain accelerators. Nevertheless, this exercise is useful for learning how to find register limited kernels using Omniperf and asks you to imagine the limiter exists for the sake of the exercise. This is an example of how many things influence performance bugs: they exist on hardware, with a software stack, at a certain time. They may never exist outside that context.

Background: Acronyms and terms used in this exercise

VGPR: Vector General Purpose Register, holds distinct values for each thread in a wavefront

SGPR: Scalar General Purpose Register, holds a single value for all threads in a workgroup

AGPR: Accumulation vector General Purpose Register, used for Matrix Fused Multiply-Add (MFMA) instructions, or low-cost register spills

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

```
    <li><strong>FP(32/16):</strong> 32- or 16-bit Floating Point numeric types</li>
    <li><strong>FLOPs:</strong> Floating Point Operations Per second</li>
    <li><strong>HBM:</strong> High Bandwidth Memory is globally accessible from the GPU, and is a level of memory
</ul>
```

Results on MI210

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`**

Initial Roofline Analysis

This kernel is slightly different from the one we used in previous exercises. Let's see how well it measures up in the roofline:

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

You can generate these plots 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 is still a considerable amount below the maximum achievable bandwidth, so there should still be room for improvement.

Exercise Instructions:

Let's get an idea of the runtime of this code:

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

(*simulated output*)

```
yAx time 71 ms
```

We see that this kernel seems to be on par with some of our other exercises, but let's see what omniperf shows us:

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

(*lots of output from this command*)

```
omniperf analyze -p workloads/problem/MI200 --dispatch 1 --block 2.1.15 6.2.5 7.1.5 7.1.6 7.1.7
```

- `2.1.15` Shows Wavefront occupancy
- `6.2.5` Shows Insufficient SIMD VGPRs – indicating if this kernel is occupancy limited by VGPR usage
- `7.1.5-7` Shows the register usage: VGPRs, SGPRs, and AGPRs

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

   INFO Analysis mode = cli
   INFO [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 │ 77266823.00 │ 77266823.00 │  77266823.00 │ 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 | 433.52 | Wavefronts | 3328.00 |         13.03 |
-------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
--------------------------------------------------------------------------
| Metric_ID   | Metric                  |   Avg |   Min |   Max | Unit   |
--------------|-------------------------|-------|-------|-------|---------
| 6.2.5       | Insufficient SIMD VGPRs |  0.10 |  0.10 |  0.10 | Pct    |
--------------------------------------------------------------------------


--------------------------------------------------------------------------------
7. Wavefront
7.1 Wavefront Launch Stats
-----------------------------------------------------------------
| Metric_ID   | Metric   |    Avg |    Min |    Max | Unit      |
--------------|----------|--------|--------|--------|------------
| 7.1.5       | VGPRs    |  92.00 |  92.00 |  92.00 | Registers |
--------------|----------|--------|--------|--------|------------
| 7.1.6       | AGPRs    | 132.00 | 132.00 | 132.00 | Registers |
--------------|----------|--------|--------|--------|------------
| 7.1.7       | SGPRs    |  48.00 |  48.00 |  48.00 | Registers |
-----------------------------------------------------------------
```

Looking at this data, we see: - Insufficient SIMD VGPRs (`6.2.5`) shows that we are slightly occupancy limited by VGPRs - VGPRs (`7.1.5`) shows we are using a moderate amount of VGPRs and we are using 132 AGPRs (`7.1.6`), which can indicate low-cost register spills in the absence of MFMA instructions.

In problem.cpp, the limiter is due to a call to `assert` that checks if our result is zeroed out on device. To make sure the problem is gone in solution.cpp, let's look at the solution code:

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

(*simulated output*)

```
yAx time: 70 ms
```

Our runtime seems fairly similar with or without the `assert` , but we should also check that omniperf reports that our limiters are gone:

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

(*omitted output*)

```
omniperf analyze -p workloads/solution/MI200 --dispatch 1 --block 2.1.15 6.2.5 7.1.5 7.1.6 7.1.7
```

The output of this command should look something like:

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

   INFO Analysis mode = cli
   INFO [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 | 71714804.00 | 71714804.00 |  71714804.00 | 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 | 439.96 | Wavefronts | 3328.00 |         13.22 |
-------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
---------------------------------------------------------------------------
| Metric_ID   | Metric                  |   Avg |   Min |   Max | Unit    |
--------------|-------------------------|-------|-------|-------|----------
| 6.2.5       | Insufficient SIMD VGPRs |  0.00 |  0.00 |  0.00 | Pct     |
---------------------------------------------------------------------------


--------------------------------------------------------------------------------
7. Wavefront
7.1 Wavefront Launch Stats
--------------------------------------------------------------
| Metric_ID   | Metric   |   Avg |   Min |   Max | Unit      |
--------------|----------|-------|-------|-------|------------
| 7.1.5       | VGPRs    | 32.00 | 32.00 | 32.00 | Registers |
--------------|----------|-------|-------|-------|------------
| 7.1.6       | AGPRs    |  0.00 |  0.00 |  0.00 | Registers |
--------------|----------|-------|-------|-------|------------
```

```
| 7.1.7       | SGPRs    | 96.00 | 96.00 | 96.00 | Registers |
--------------------------------------------------------------
```

Looking at this data, we see: - Insufficient SIMD VGPRs (`6.2.5`) shows we are now not occupancy limited by VGPR usage. - VGPRs (`7.1.5`) are down by 60, AGPRs (`7.1.6`) are down by 132, and SGPRs (`7.1.7`) are up, showing more efficient register usage. - Wave Occupancy (`2.1.26`) shows our occupancy is slightly increased.

More generally, you can use this command to look at all SPI "insufficient resource" stats in the same screen, to determine if any resource is currently limiting occupancy. In fact, we can use this to ensure our problem implementation no longer has any SPI-related occupancy limiters with the newer version of ROCm:

```
omniperf analyze -p workloads/problem/MI200 --dispatch 1 --block 6.2
```

Which will show output similar to this (note, fields `6.2.4` to `6.2.8` show resources which currently limit occupancy):

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

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 | 69960451.00 | 69960451.00 |  69960451.00 | 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 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
-----------------------------------------------------------------------------------------
| Metric_ID   | Metric                                  |   Avg |   Min |   Max | Unit   |
--------------|-----------------------------------------|-------|-------|-------|---------
| 6.2.0       | Not-scheduled Rate (Workgroup Manager)  |  0.00 |  0.00 |  0.00 | Pct    |
--------------|-----------------------------------------|-------|-------|-------|---------
| 6.2.1       | Not-scheduled Rate (Scheduler-Pipe)     |  0.00 |  0.00 |  0.00 | Pct    |
--------------|-----------------------------------------|-------|-------|-------|---------
| 6.2.2       | Scheduler-Pipe Stall Rate               |  0.00 |  0.00 |  0.00 | Pct    |
--------------|-----------------------------------------|-------|-------|-------|---------
| 6.2.3       | Scratch Stall Rate                      |  0.00 |  0.00 |  0.00 | Pct    |
--------------|-----------------------------------------|-------|-------|-------|---------
| 6.2.4       | Insufficient SIMD Waveslots             |  0.00 |  0.00 |  0.00 | Pct    |
--------------|-----------------------------------------|-------|-------|-------|---------
```

```
| 6.2.5        | Insufficient SIMD VGPRs                 |  0.00 |  0.00 |  0.00 | Pct     |
--------------|----------------------------------------|-------|-------|-------|---------
| 6.2.6        | Insufficient SIMD SGPRs                 |  0.00 |  0.00 |  0.00 | Pct     |
--------------|----------------------------------------|-------|-------|-------|---------
| 6.2.7        | Insufficient CU LDS                     |  0.00 |  0.00 |  0.00 | Pct     |
--------------|----------------------------------------|-------|-------|-------|---------
| 6.2.8        | Insufficient CU Barriers                |  0.00 |  0.00 |  0.00 | Pct     |
--------------|----------------------------------------|-------|-------|-------|---------
| 6.2.9        | Reached CU Workgroup Limit              |  0.00 |  0.00 |  0.00 | Pct     |
--------------|----------------------------------------|-------|-------|-------|---------
| 6.2.10       | Reached CU Wavefront Limit              |  0.00 |  0.00 |  0.00 | Pct     |
-----------------------------------------------------------------------------------------
```

Solution Roofline

With similar performance, we expect to see similar plots in the roofline for problem and solution:

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

You can generate these plots with:

```
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.

The plots are indistinguishable, which is further confirmation performance is now unchanged between problem and solution. However, we see there is still room for improvement as this kernel is not getting the maximum achievable bandwidth.

Roofline Comparison

Roofline Type	Problem Roofline	Solution Roofline
FP32/FP64	image	image
FP16/INT8	image	image

Summary and Take-aways

Function calls inside kernels can have surprisingly adverse performance side-effects. However, performance issues in general may be subject to compiler versions or other environment details. Calling assert, printf and even excessive use of math functions (e.g. pow, sin, cos) can limit performance in difficult-to-predict ways. If you see unexpected resource usage, try eliminating or reducing the use of these sorts of function calls.

Results on MI300A

In this section, we show results obtained running this exercise on a system with MI300A, using ROCm `6.2.1` and the associated Omniperf, version `6.2.1`.

Roofline Analysis:

At present (September 28th 2024), rooflines are disabled on MI300A.

As for the MI210 case, build and run the problem code:

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

(*simulated output*)

```
yAx time: 10 ms
```

Let's run the following commands to explore some metrics:

```
omniperf profile -n problem --no-roof -- ./problem.exe
omniperf analyze -p workloads/problem/MI300A_A1 --dispatch 1 --block 2.1.15 6.2.5 7.1.5 7.1.6 7.1.7
```

Then explore the output:

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

   INFO Analysis mode = cli
   INFO [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 | 10064928.00 | 10064928.00 |  10064928.00 | 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] |        4 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
-------------------------------------------------------------------------------------
| Metric_ID    | Metric               |    Avg | Unit       |    Peak |   Pct of Peak |
--------------|----------------------|--------|------------|---------|----------------
| 2.1.15       | Wavefront Occupancy | 432.15 | Wavefronts | 7296.00 |          5.92 |
-------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
--------------------------------------------------------------------------
| Metric_ID    | Metric                   |   Avg |   Min |   Max | Unit   |
--------------|--------------------------|-------|-------|-------|---------
| 6.2.5        | Insufficient SIMD VGPRs |  0.06 |  0.06 |  0.06 | Pct    |
--------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
7. Wavefront
7.1 Wavefront Launch Stats
-----------------------------------------------------------------
| Metric_ID   | Metric   |    Avg |    Min |    Max | Unit      |
--------------|----------|--------|--------|--------|------------
| 7.1.5       | VGPRs    |  92.00 |  92.00 |  92.00 | Registers |
--------------|----------|--------|--------|--------|------------
| 7.1.6       | AGPRs    | 132.00 | 132.00 | 132.00 | Registers |
--------------|----------|--------|--------|--------|------------
| 7.1.7       | SGPRs    |  48.00 |  48.00 |  48.00 | Registers |
-----------------------------------------------------------------
```

As expected, there is minor limiting due to Insufficient SIMD VGPRs, which is similar to the MI210 case. A similar scenario is seen when running solution:

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

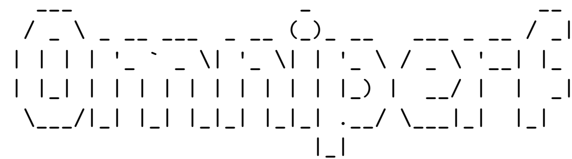
(*simulated output*)

```
yAx time: 9.82 ms
```

The runtime is practically the same as the `problem` implementation. For performance metrics, let's run:

```
omniperf profile -n solution --no-roof -- ./solution.exe
omniperf analyze -p workloads/solution/MI300A_A1 --dispatch 1 --block 2.1.15 6.2.5 7.1.5 7.1.6 7.1.7
```

With output:



```
   INFO Analysis mode = cli
   INFO [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 | 9794300.00 | 9794300.00 |   9794300.00 | 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] |        4 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
-------------------------------------------------------------------------------------
```

```
| Metric_ID   | Metric              |    Avg | Unit       |    Peak |   Pct of Peak |
--------------|---------------------|--------|------------|---------|----------------
| 2.1.15      | Wavefront Occupancy | 430.69 | Wavefronts | 7296.00 |          5.90 |
----------------------------------------------------------------------------------


--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
--------------------------------------------------------------------------
| Metric_ID   | Metric                  |   Avg |   Min |   Max | Unit    |
--------------|-------------------------|-------|-------|-------|---------
| 6.2.5       | Insufficient SIMD VGPRs |  0.00 |  0.00 |  0.00 | Pct     |
--------------------------------------------------------------------------


--------------------------------------------------------------------------------
7. Wavefront
7.1 Wavefront Launch Stats
-----------------------------------------------------------------
| Metric_ID   | Metric   |    Avg |    Min |    Max | Unit      |
--------------|----------|--------|--------|--------|------------
| 7.1.5       | VGPRs    |  32.00 |  32.00 |  32.00 | Registers |
--------------|----------|--------|--------|--------|------------
| 7.1.6       | AGPRs    |   0.00 |   0.00 |   0.00 | Registers |
--------------|----------|--------|--------|--------|------------
| 7.1.7       | SGPRs    | 112.00 | 112.00 | 112.00 | Registers |
-----------------------------------------------------------------
```

Just like the case of MI210, the Wavefront Launch Stats differ between `problem` and `solution`. As we did for MI210, let's run:

```
cd ..
omniperf analyze -p workloads/problem/MI300A_A1 --dispatch 1 --block 6.2
```

With output:

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

   INFO Analysis mode = cli
   INFO [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 | 10226783.00 | 10226783.00 |  10226783.00 | 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] |        4 |
--------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
-----------------------------------------------------------------------------------------
| Metric_ID    | Metric                                   |   Avg |   Min |   Max | Unit   |
--------------|------------------------------------------|-------|-------|-------|---------
| 6.2.0       | Not-scheduled Rate (Workgroup Manager)  |  0.01 |  0.01 |  0.01 | Pct    |
--------------|------------------------------------------|-------|-------|-------|---------
| 6.2.1       | Not-scheduled Rate (Scheduler-Pipe)     |  0.03 |  0.03 |  0.03 | Pct    |
--------------|------------------------------------------|-------|-------|-------|---------
| 6.2.2       | Scheduler-Pipe Stall Rate               |  0.02 |  0.02 |  0.02 | Pct    |
--------------|------------------------------------------|-------|-------|-------|---------
| 6.2.3       | Scratch Stall Rate                      |  0.00 |  0.00 |  0.00 | Pct    |
--------------|------------------------------------------|-------|-------|-------|---------
| 6.2.4       | Insufficient SIMD Waveslots             |  0.00 |  0.00 |  0.00 | Pct    |
--------------|------------------------------------------|-------|-------|-------|---------
| 6.2.5       | Insufficient SIMD VGPRs                 |  0.06 |  0.06 |  0.06 | Pct    |
--------------|------------------------------------------|-------|-------|-------|---------
| 6.2.6       | Insufficient SIMD SGPRs                 |  0.00 |  0.00 |  0.00 | Pct    |
--------------|------------------------------------------|-------|-------|-------|---------
| 6.2.7       | Insufficient CU LDS                     |  0.00 |  0.00 |  0.00 | Pct    |
--------------|------------------------------------------|-------|-------|-------|---------
| 6.2.8       | Insufficient CU Barriers                |  0.00 |  0.00 |  0.00 | Pct    |
--------------|------------------------------------------|-------|-------|-------|---------
| 6.2.9       | Reached CU Workgroup Limit              |  0.00 |  0.00 |  0.00 | Pct    |
--------------|------------------------------------------|-------|-------|-------|---------
| 6.2.10      | Reached CU Wavefront Limit              |  0.00 |  0.00 |  0.00 | Pct    |
-----------------------------------------------------------------------------------------
```

Exercise 4: Strided Data Access Patterns (and how to find them)

This exercise uses a simple implementation of a yAx kernel to show how difficult strided data access patterns can be to spot in code, and demonstrates how to use omniperf to begin to diagnose them.

Background: Acronyms and terms used in this exercise

L1: Level 1 Cache, the first level cache local to the Compute Unit (CU). If requested data is not found in the L1, the request goes to the L2

L2: Level 2 Cache, the second level cache, which is shared by all Compute Units (CUs) on a GPU. If requested data is not found in the L2, the request goes to HBM

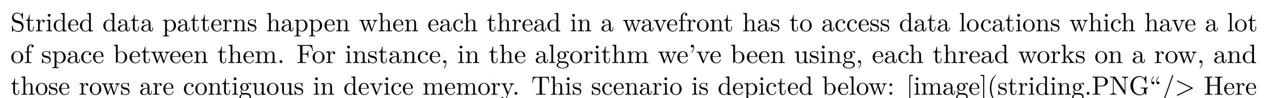
HBM: High Bandwidth Memory is globally accessible from the GPU, and is a level of memory above the L2 cache

CU: The Compute Unit is responsible for executing the User's kernels

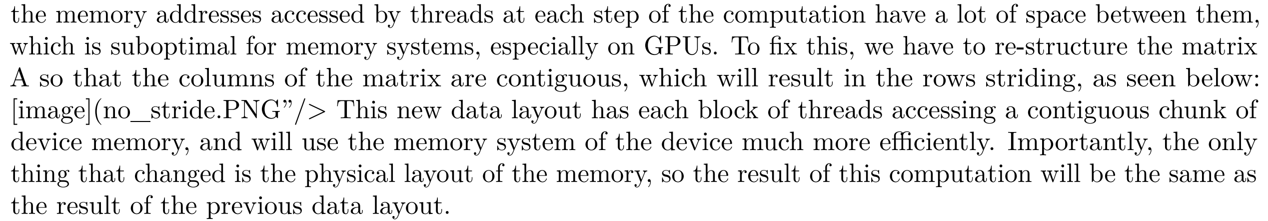
yAx: a vector-matrix-vector product, yAx, where y and x are vectors, and A is a matrix

FP(32/16): 32- or 16-bit Floating Point numeric types

Background: What is a "Strided Data Access Pattern"?



Strided data patterns happen when each thread in a wavefront has to access data locations which have a lot of space between them. For instance, in the algorithm we've been using, each thread works on a row, and those rows are contiguous in device memory. This scenario is depicted below: [image](striding.PNG"/> Here



the memory addresses accessed by threads at each step of the computation have a lot of space between them, which is suboptimal for memory systems, especially on GPUs. To fix this, we have to re-structure the matrix A so that the columns of the matrix are contiguous, which will result in the rows striding, as seen below: [image](no_stride.PNG"/> This new data layout has each block of threads accessing a contiguous chunk of device memory, and will use the memory system of the device much more efficiently. Importantly, the only thing that changed is the physical layout of the memory, so the result of this computation will be the same as the result of the previous data layout.

Results on MI210

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`**

Initial Roofline Analysis

To start, we want to check the roofline of `problem.exe`, to make sure we are able to improve it. These plots can be generated with:

```
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.

They are also provided below for easy reference:

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

We have plenty of space to improve this kernel, the next step is profiling.

Exercise Instructions:

To start, let's build and run the problem executable:

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

(*simulated output*)

```
yAx time: 70 ms
```

From our other experiments, this time seems reasonable. Let's look closer at the memory system usage with omniperf:

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

(*omitted output*)

```
omniperf analyze -p workloads/problem/MI200 --dispatch 1 --block 16.1 17.1
```

> Previous examples have used specific fields inside metrics, but we can also request a group of metrics with just two numbers (i.e. 16.1 vs. 16.1.1)

These requested metrics are: - `16.1` L1 memory speed-of-light stats - `17.1` L2 memory speed-of-light stats

The speed-of-light stats are a more broad overview of how the memory systems are used throughout execution of your kernel. As such, they're great statistics for seeing if the memory system is generally being used efficiently or not. Output from the `analyze` command should look 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 | 70270856.00 | 70270856.00 |  70270856.00 | 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 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
16. Vector L1 Data Cache
16.1 Speed-of-Light
---------------------------------------------------
| Metric_ID   | Metric      |   Avg | Unit        |
--------------|-------------|-------|--------------
| 16.1.0      | Hit rate    |  0.00 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.1      | Bandwidth   |  8.64 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.2      | Utilization | 87.71 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.3      | Coalescing  | 25.00 | Pct of peak |
---------------------------------------------------


--------------------------------------------------------------------------------
17. L2 Cache
17.1 Speed-of-Light
-----------------------------------------------------------------
| Metric_ID   | Metric                         |    Avg | Unit   |
--------------|--------------------------------|--------|---------
| 17.1.0      | Utilization                    |  98.66 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.1      | Bandwidth                      |  28.10 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.2      | Hit Rate                       |  93.45 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.3      | L2-Fabric Read BW              | 125.05 | Gb/s   |
--------------|--------------------------------|--------|---------
| 17.1.4      | L2-Fabric Write and Atomic BW  |   0.00 | Gb/s   |
-----------------------------------------------------------------
```

Looking at this data, we see: - L1 Cache Hit (`16.1.0`) is 0%, so the kernel's memory requests are never found in the L1. - L2 Cache Hit (`17.1.2`) is 93.46%, so most requests are found in the L2, with about 7% needing to go out to HBM. - We are never finding data in the L1 and generating a lot of requests to the L2, so restructuring our data accesses should provide better performance

Since our implementation of yAx simply uses 1 for all values in y, A, and x, we do not have to change how we populate our data. Since A is implemented as a flat array, we don't need to change our allocation either. >In real-world use-cases, these considerations add non-trivial development overhead, so data access patterns may be non-trivial to change.

To observe the performance effects of a different data access pattern, we simply need to change our indexing scheme. Let's see how this performs by running `solution` :

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

(*simulated output*)

```
yAx time: 12 ms
```

We see the runtime here is significantly better than our previous kernel, but we need to check how the caches behave now:

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

(*output omitted*)

```
omniperf analyze -p workloads/solution/MI200 --dispatch 1 --block 16.1 17.1
```

The output from this analyze command should look 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 | 12364156.00 | 12364156.00 |   12364156.00 | 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 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
16. Vector L1 Data Cache
16.1 Speed-of-Light
```

```
--------------------------------------------------
| Metric_ID    | Metric      |   Avg | Unit        |
--------------|-------------|-------|--------------
| 16.1.0       | Hit rate    | 49.98 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.1       | Bandwidth   | 12.29 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.2       | Utilization | 98.12 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.3       | Coalescing  | 25.00 | Pct of peak |
--------------------------------------------------


--------------------------------------------------------------------------------
17. L2 Cache
17.1 Speed-of-Light
-----------------------------------------------------------------
| Metric_ID    | Metric                         |    Avg | Unit   |
--------------|--------------------------------|--------|---------
| 17.1.0       | Utilization                    |  98.56 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.1       | Bandwidth                      |  10.03 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.2       | Hit Rate                       |   0.52 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.3       | L2-Fabric Read BW              | 694.86 | Gb/s   |
--------------|--------------------------------|--------|---------
| 17.1.4       | L2-Fabric Write and Atomic BW  |   0.00 | Gb/s   |
-----------------------------------------------------------------
```

Looking at this data, we see: - L1 Cache Hit (`16.1.0`) is around 50%, so half the requests to the L1 need to go to the L2. - L2 Cache Hit (`17.1.2`) is 0.52%, so almost all the requests to the L2 have to go out to HBM. - L2-Fabric Read BW (`17.1.3`) has increased significantly, due to the increase in L2 cache misses requiring HBM reads.

Solution Roofline Analysis

We should check where our new kernel stands on the roofline. These plots can be generated with:

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

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

They are also provided below for easy reference:

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

We appear to be very close to being bound by the HBM bandwidth from the fp32 roofline. To get more performance we need to look closer at our algorithm.

Roofline Comparison

Roofline Type	Problem Roofline	Solution Roofline
FP32/FP6	image	image
FP16/INT8	image	image

We see that the HBM roofline point moves up, while the L1/L2 points move up and to the right from problem to solution. This means that our arithmetic intensity is increasing for the caches, so we are moving less data through the caches to do the same computation.

Summary and Take-aways

This exercise illustrates the at times insidious nature of strided data access patterns. They can be difficult to spot in code, but profiling more readily shows when adversarial access patterns occur, by showing poor cache hit rates, low cache bandwidth, and potentially low utilization. Data access patterns can be non-trivial to change, so these sorts of optimizations can involve significant development and validation overhead.

Results on MI300A

Note: Roofline is not available on MI300A at the time of this writing

For MI300A, if we run the problem.exe and solution.exe, we see different performance.

Running the problem:

```
./problem.exe
```

Shows a runtime like this:

```
9.64 milliseconds
```

While running solution:

```
./solution.exe
```

Shows a runtime like this:

```
12.17 milliseconds
```

Now, if we use omniperf to profile these executables we see much the same stats for MI300A as MI200:

```
omniperf analyze -p workloads/problem/MI300A_A1 --dispatch 1 --block 16.1 17.1
```

Shows

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

   INFO Analysis mode = cli
   INFO [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 | 9599042.00 | 9599042.00 |   9599042.00 | 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] |        4 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
16. Vector L1 Data Cache
16.1 Speed-of-Light
---------------------------------------------------
| Metric_ID   | Metric      |   Avg | Unit        |
--------------|-------------|-------|--------------
| 16.1.0      | Hit rate    |  0.00 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.1      | Bandwidth   | 23.36 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.2      | Utilization | 85.90 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.3      | Coalescing  | 25.00 | Pct of peak |
---------------------------------------------------


--------------------------------------------------------------------------------
17. L2 Cache
17.1 Speed-of-Light
-----------------------------------------------------------------
| Metric_ID   | Metric                         |    Avg | Unit   |
--------------|--------------------------------|--------|---------
| 17.1.0      | Utilization                    |  96.87 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.1      | Bandwidth                      |  55.52 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.2      | Hit Rate                       |  93.67 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.3      | L2-Fabric Read BW              | 908.10 | Gb/s   |
--------------|--------------------------------|--------|---------
| 17.1.4      | L2-Fabric Write and Atomic BW  |   0.01 | Gb/s   |
-----------------------------------------------------------------
```

While analyzing the solution with:

```
omniperf analyze -p workloads/solution/MI300A_A1 --dispatch 1 --block 16.1 17.1
```

Shows:

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

   INFO Analysis mode = cli
   INFO [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 | 12104495.00 | 12104495.00 |   12104495.00 | 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] |        4 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
16. Vector L1 Data Cache
16.1 Speed-of-Light
----------------------------------------------------
| Metric_ID   | Metric      |    Avg | Unit        |
--------------|-------------|--------|--------------
| 16.1.0      | Hit rate    |  75.00 | Pct of peak |
--------------|-------------|--------|--------------
| 16.1.1      | Bandwidth   |   4.63 | Pct of peak |
--------------|-------------|--------|--------------
| 16.1.2      | Utilization | 100.00 | Pct of peak |
--------------|-------------|--------|--------------
| 16.1.3      | Coalescing  |  25.00 | Pct of peak |
----------------------------------------------------


--------------------------------------------------------------------------------
17. L2 Cache
17.1 Speed-of-Light
-----------------------------------------------------------------
| Metric_ID   | Metric                         |    Avg | Unit   |
--------------|--------------------------------|--------|---------
| 17.1.0      | Utilization                    |  98.72 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.1      | Bandwidth                      |   2.77 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.2      | Hit Rate                       |   0.68 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.3      | L2-Fabric Read BW              | 710.06 | Gb/s   |
--------------|--------------------------------|--------|---------
| 17.1.4      | L2-Fabric Write and Atomic BW  |   0.01 | Gb/s   |
-----------------------------------------------------------------
```

So we see a slowdown despite increasing our L1 hit rate (`16.1.0`) by a large amount. Let's see how the runtime compares to the number of cycles required for problem and solution, as well as atomic latencies per channel for both approaches:

```
omniperf analyze -p workloads/problem/MI300A_A1 -p workloads/solution/MI300A_A1 --dispatch 1 --block 7.2.0 7.2.1 17
```

Which shows:

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

```
 \___/|_| |_| |_|_| |_|_| .__/ \___|_|  |_|
                         |_|

   INFO Analysis mode = cli
   INFO [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 | 9599042.00 | 12104495.0 (26.1
|    |  double*) [clone .kd]                    |         |            |            |            |
-----------------------------------------------------------------------------------------------------------------------
0.2 Dispatch List
-------------------------------------------------------------------------------------------------
|    |   Dispatch_ID | Kernel_Name                                                    |   GPU_ID |
-----|---------------|----------------------------------------------------------------|-----------
|  0 |             1 | yax(double*, double*, double*, int, int, double*) [clone .kd] |        4 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
7. Wavefront
7.2 Wavefront Runtime Stats
-----------------------------------------------------------------------------------------------------------------------
| Metric_ID   | Metric                 |         Avg | Avg                   |   Abs Diff |         Min | Min
--------------|------------------------|-------------|-----------------------|------------|-------------|-------------
| 7.2.0       | Kernel Time (Nanosec) |  9599042.00 | 12104495.0 (26.1%)    | 2505453.00 |  9599042.00 | 12104495.0 (
--------------|------------------------|-------------|-----------------------|------------|-------------|-------------
| 7.2.1       | Kernel Time (Cycles)  | 19350602.00 | 25141619.0 (29.93%)   | 5791017.00 | 19350602.00 | 25141619.0 (
-----------------------------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
17. L2 Cache
17.2 L2 - Fabric Transactions
-----------------------------------------------------------------------------------------------------------------------
| Metric_ID   | Metric         |     Avg | Avg               |   Abs Diff |     Min | Min
|     Max | Max              | Unit   |
--------------|----------------|---------|-------------------|------------|---------|------------------|---------|--
| 17.2.11     | Atomic Latency | 6406.73 | 9547.67 (49.03%) |    3140.94 | 6406.73 | 9547.67 (49.03%) | 6406.73 | 9
-----------------------------------------------------------------------------------------------------------------------
```

Atomic Latency `17.2.11` shows that our solution is more stressful on atomics, likely due to our more highly optimized cache access. Fixing striding ironically caused our threads to issue atomics more quickly, degrading our performance!

To fix this, we can attempt to mitigate contention by doing what is called a “shuffle reduction” on each wavefront. This utilizes a HIP intrinsic called `__shfl_down` to reduce numbers across threads in a wavefront without requiring atomics. These implementations can be found in `mi300a_problem` and mi300a_solution‘. The code contained in those subdirectories simply implements this shuffle reduction, which allows both problem and solution to only have the first thread of each wavefront issue the atomic add, rather than all threads.

Let’s run mi300a_problem.exe and mi300a_solution.exe to see if this addresses our problem:

```
./mi300a_problem.exe
```

shows:

```
yAx time: 9.577708 milliseconds
```

While

```
./mi300a_solution.exe
```

Shows:

```
yAx time: 12.381036 milliseconds
```

Strangely, this seems to have little effect on our runtimes. The reader may notice that these problems are running very quickly, and the reader would be right. The mi300a_problem.exe and mi300a_solution.exe both provide an argument to test different problem sizes. Since we assume our matrix in question is square (for the time being this is an arbitrary assumption – the kernel is capable of handling rectangular matrices as well), increasing the argument by one increases the problem size nonlinearly. Let's try running at problem size 15:

```
./mi300a_problem.exe 15
```

Shows

```
yAx time: 312.857488 milliseconds
```

And

```
./mi300a_solution.exe 15
```

Shows

```
yAx time: 25.878859 milliseconds
```

It appears that at a smaller problem size, this kernel is more bounded by atomic contention than efficient cache memory usage. It is important to test different problem sizes to ensure that a run of a code being profiled is representative, otherwise the limiters shown in profiling may point optimizations in the wrong direction for a full scale run. As proof of this, you can try manually setting the problem.cpp and solution.cpp problem size to 15, and see that they run in a similar amount of time to mi300a_problem and mi300a_solution. At scale, the memory bandwidth dominates this specific kernel.

Exercise 5: Algorithmic Optimizations

A simple yAx kernel, and more efficient, but more complex yAx kernel to demonstrate algorithmic improvements.

Background: Acronyms and terms used in this exercise

L1: Level 1 Cache, the first level cache local to the Compute Unit (CU). If requested data is not found in the L1, the request goes to the L2

L2: Level 2 Cache, the second level cache, which is shared by all Compute Units (CUs) on a GPU. If requested data is not found in the L2, the request goes to HBM

HBM: High Bandwidth Memory is globally accessible from the GPU, and is a level of memory above the L2 cache

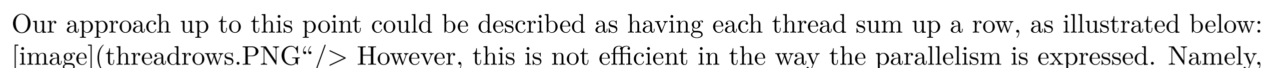
CU: The Compute Unit is responsible for executing the User's kernels

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

Background: yAx Algorithmic Improvement Explanation

Our approach up to this point could be described as having each thread sum up a row, as illustrated below:



[image](threadrows.PNG“/> However, this is not efficient in the way the parallelism is expressed. Namely,

we could add up all the partial sums for each row in parallel. This would make our approach to be: give a rows to wavefronts, and have the threads inside each wavefront sum up partial sums in parallel. Then, we reduce the partial sums atomically with shared memory, before completing the computation and reducing the final answer using global atomics. This approach expresses more of the parallelism that is available, and would look something like the figure below: [image](wavefrontrow.PNG"/> The expressed parallelism in each approach roughly corresponds to the number of red arrows in each figure.

Results on MI210:

Note: This exercise was tested on a system with MI210s, on omniperf version `2.0.0` and ROCm `6.1.2` **Omniperf `2.0.0` is incompatible with ROCm versions lesser than `6.0.0`**

Initial Roofline Analysis

We should start by doing a roofline to see where the problem executable stands. These plots can be generated with:

```
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.

They are also provided below for easy reference:

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

The performance of this kernel looks pretty close to being HBM bandwidth bound. In the case of algorithmic optimizations, there may not be obvious evidence other than a suspicion that poor usage of hardware resources may be improved by changing the overall approach. In this case, we should be able to make better usage of both L1 and L2 resources by using wavefronts more efficiently to better parallelize our computation.

Exercise Instructions:

To start, let's profile `problem.exe` :

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

(*simulated output*)

```
yAx time 12 ms
```

This should be in line with our last solution. From the last exercise, we saw this output from `omniperf analyze` for this kernel:

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

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 | 12364156.00 | 12364156.00 |  12364156.00 | 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 |
─────────────────────────────────────────────────────────────────────────────────────────────────


────────────────────────────────────────────────────────────────────────────────
16. Vector L1 Data Cache
16.1 Speed-of-Light
───────────────────────────────────────────────────
| Metric_ID   | Metric      |   Avg | Unit        |
--------------|-------------|-------|--------------
| 16.1.0      | Hit rate    | 49.98 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.1      | Bandwidth   | 12.29 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.2      | Utilization | 98.12 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.3      | Coalescing  | 25.00 | Pct of peak |
───────────────────────────────────────────────────


────────────────────────────────────────────────────────────────────────────────
17. L2 Cache
17.1 Speed-of-Light
─────────────────────────────────────────────────────────────────
| Metric_ID   | Metric                         |    Avg | Unit   |
--------------|--------------------------------|--------|---------
| 17.1.0      | Utilization                    |  98.56 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.1      | Bandwidth                      |  10.03 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.2      | Hit Rate                       |   0.52 | Pct    |
--------------|--------------------------------|--------|---------
| 17.1.3      | L2-Fabric Read BW              | 694.86 | Gb/s   |
--------------|--------------------------------|--------|---------
| 17.1.4      | L2-Fabric Write and Atomic BW  |   0.00 | Gb/s   |
─────────────────────────────────────────────────────────────────
```

Looking at this data again, we see: - L1 Cache Hit (`16.1.0`) is about 50%, which is fairly low for a “well performing” kernel. - L2 Cache Hit (`17.1.2`) is about 0%, which is very low to consider this kernel “well performing”.

This data indicates that we should be able to make better usage of our memory system, so let’s apply the algorithmic optimization present in `solution.cpp` :

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

(*simulated output*)

```
yAx time: 7.7 ms
```

It should be noted again that algorithmic optimizations are usually the most expensive optimizations to implement, as they usually entail re-conceptualizing the problem in a way that allows for a more efficient solution. However, as we see here, algorithmic optimization *can* result in impressive speedups. A better runtime is not proof that we are using our caches more efficiently, we have to profile the solution:

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

(*output omitted*)

```
omniperf analyze -p workloads/solution/MI200 --dispatch 1 --block 16.1 17.1
```

The output for the solution should look something like:

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

   INFO Analysis mode = cli
   INFO [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 | 7774568.00 | 7774568.00 |   7774568.00 | 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] |        2 |
-------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------
16. Vector L1 Data Cache
16.1 Speed-of-Light
---------------------------------------------------
| Metric_ID    | Metric      |   Avg | Unit        |
--------------|-------------|-------|--------------
| 16.1.0       | Hit rate    | 71.52 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.1       | Bandwidth   | 39.06 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.2       | Utilization | 97.85 | Pct of peak |
--------------|-------------|-------|--------------
| 16.1.3       | Coalescing  | 25.00 | Pct of peak |
---------------------------------------------------


--------------------------------------------------------------------------------
```

```
17. L2 Cache
17.1 Speed-of-Light
-------------------------------------------------------------------
| Metric_ID   | Metric                          |     Avg | Unit   |
--------------|---------------------------------|---------|---------
| 17.1.0      | Utilization                     |   91.55 | Pct    |
--------------|---------------------------------|---------|---------
| 17.1.1      | Bandwidth                       |   20.44 | Pct    |
--------------|---------------------------------|---------|---------
| 17.1.2      | Hit Rate                        |   21.23 | Pct    |
--------------|---------------------------------|---------|---------
| 17.1.3      | L2-Fabric Read BW               | 1110.67 | Gb/s   |
--------------|---------------------------------|---------|---------
| 17.1.4      | L2-Fabric Write and Atomic BW   |    0.00 | Gb/s   |
-------------------------------------------------------------------
```

Looking at this data, we see: - L1 Cache Hit (`16.1.0`) shows 71.52%, which is an increase of 1.43x over 49.98% for problem. - L2 Cache Hit (`17.1.2`) shows 21.23%, which is an increase of 40x over 0.52% for problem. - L2-Fabric Read BW (`17.1.3`) shows 1110.67 Gb/s, an increase of 1.6x over 694.86 Gb/s for problem.

Notice that the ratio between the runtimes in this case: 12/7.7 = 1.56x, which aligns closely with the L2-Fabric Read BW increases, suggesting this kernel is bounded primarily by memory bandwidth.

Solution Roofline Analysis

As a final step, we should check how this new implementation stacks up with the roofline. These plots can be 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.

They are also provided below for easy reference:

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

As the Omniperf stats indicate, we are more efficiently using the L1 cache, which shows in the roofline as a decrease in Arithmetic Intensity for that cache layer. We have a high hit rate in L1, with a comparatively lower hit rate in L2, and we were able to increase our L2-Fabric bandwidth for the same problem size, more efficiently requesting data from HBM.

Roofline Comparison

The comparison of these two rooflines is fairly straightforward.

Roofline Type	Problem Roofline	Solution Roofline
FP32/FP64	image	image
FP16/INT8	image	image

We see now that the optimization we apply in this example makes the kernel get very close to the HBM bandwidth-bound line. The fact that our kernel falls under the bandwidth line also confirms our suspicion that this kernel is, in fact, in the bandwidth bound regime.

Summary and Take-aways

This algorithmic optimization is able to work more efficiently out of the L1 and L2. Algorithmic optimizations are all but guaranteed to have significant development overhead, but finding a more efficient algorithm can have large impacts to performance. If profiling reveals inefficient use of the memory hardware, it could be worth thinking about alternative algorithms.

Results on MI300A

Under construction…