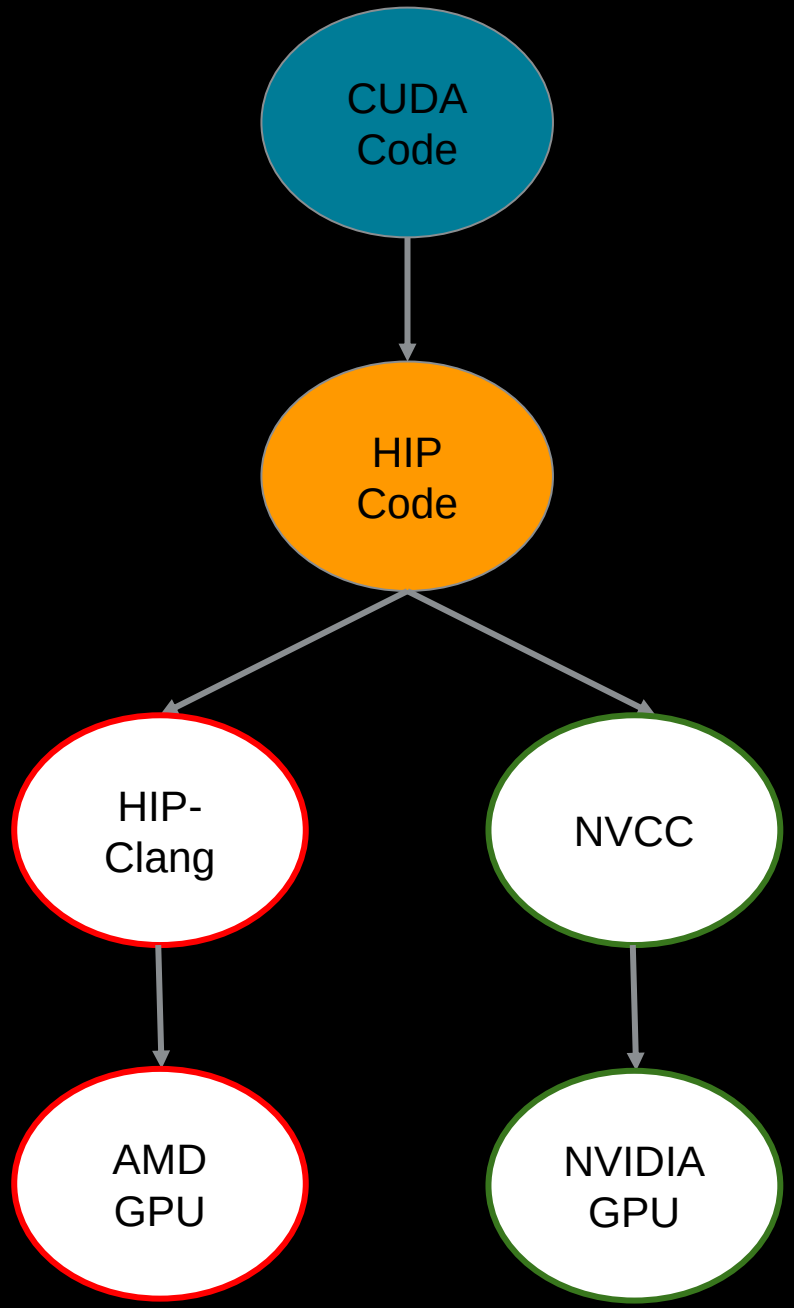
AMD
ROCm

HIP Porting

ROCm Tutorial – Part 4



Introduction

1. Code portability is important in today's world
2. Write code once and use on different platforms is a major convenience
3. HIP provides the answer to this portability
4. HIP code can also run on NVIDIA GPUs using nvcc as the internal compiler
5. HIP also enables easy porting of code from CUDA using a rich set of tools
6. This allows developers to run CUDA applications on ROCm with ease
7. In this module, we will be looking at this porting process in detail through examples

Prerequisites

1. Basic understanding of GPU programming is helpful to get started (covered in the third module)

2. Ensure that:
   - You have access to a ROCm enabled GPU
   - ROCm and HIP is correctly installed

Hipifying using HIP tools

1. HIP allows developers the flexbility to port their CUDA based application to HIP
2. Achieved by using a source to source translator which is covered in this tutorial
3. This means, a developer does not have to manually translate source code from CUDA to HIP
4. You can start running your favorite CUDA applications on ROCm based GPUs because of this support

Online Guides

1. All online guides for ROCm can be found here (https://rocmdocs.amd.com/en/latest/)

2. Following links are helpful for more in-depth information as well as information about latest updates:https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html#hip-documentation for HIP programming

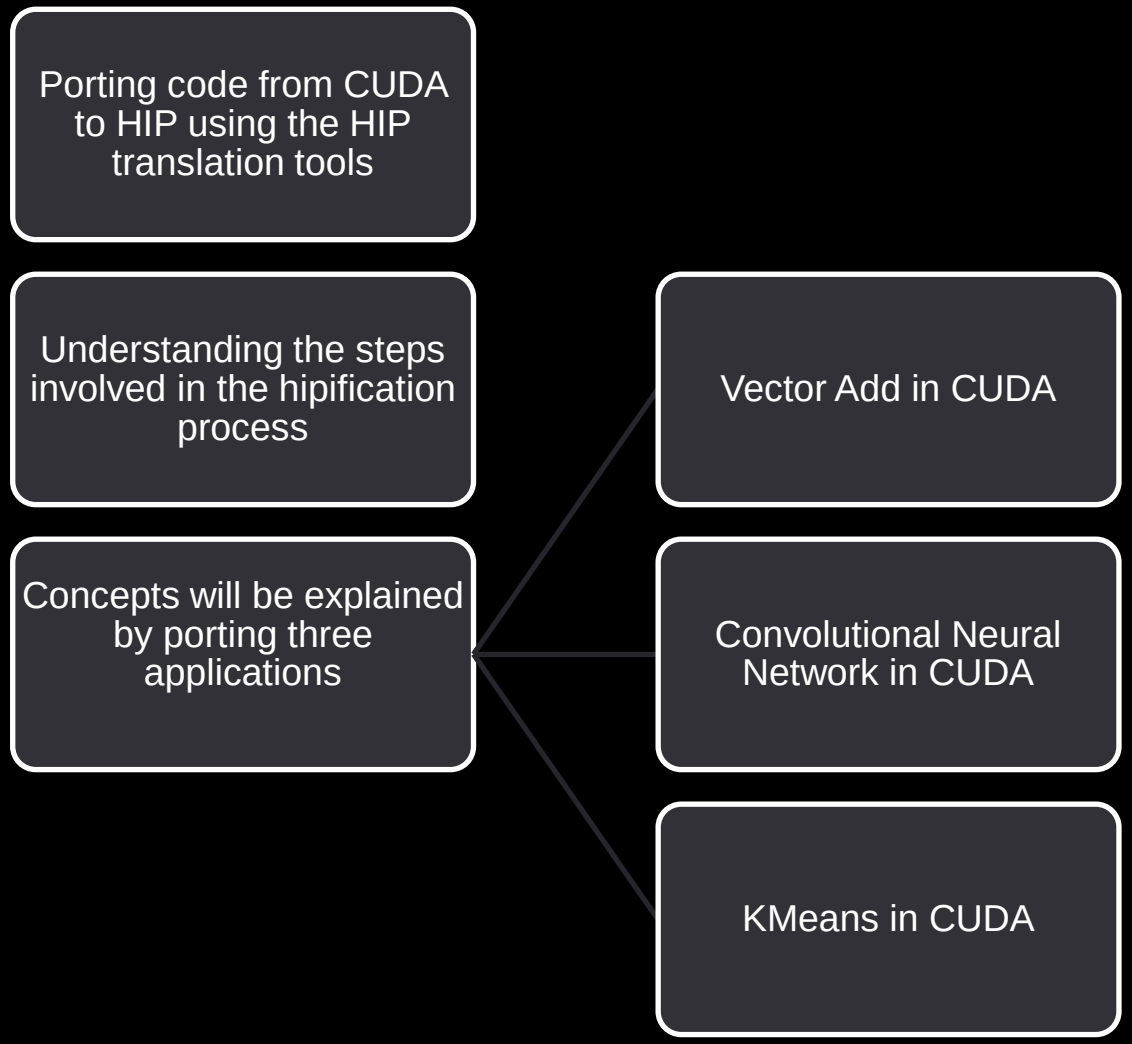
   - All supported runtime API calls and related syntax for HIP can be found “https://rocmdocs.amd.com/en/latest/ROCm_API_References/HIP-API.html#hip-api ”

   - A comprehensive coverage of porting CUDA code to HIP can be found here “https://github.com/ROCm-Developer-Tools/HIP/blob/master/docs/markdown/hip_porting_guide.md ”

   - Information about ROCm libraries (https://rocmdocs.amd.com/en/latest/ROCm_Libraries/ROCm_Libraries.html)

   - System level debugging (https://rocmdocs.amd.com/en/latest/Other_Solutions/Other-Solutions.html)

Goals


Porting code from CUDA to HIP using the HIP translation tools
Understanding the steps involved in the hipification process
Concepts will be explained by porting three applications
Vector Add in CUDA
Convolutional Neural Network in CUDA
KMeans in CUDA

Hipifying Basics

HIP Source to Source Translator

1. HIP provides a source to source translator to convert CUDA code to HIP

2. This translator replaces CUDA calls with equivalent HIP calls
   - In layman terms, it does a find and replace

3. From the last module we know HIP API calls are very similar to CUDA

4. Not all CUDA APIs are supported at the moment
   - Refer to (https://github.com/ROCm-Developer-Tools/HIP/blob/master/docs/markdown/hip_porting_guide.md) for the latest information

Hipifying Process

1. The process of hipifying a CUDA source file/files to HIP involves three major steps:

    - **Scanning**: This step involves scanning the codebase to know and understand what can and cannot be converted to HIP

    - **Porting:** This step involves using the translator to convert the CUDA files to HIP

    - **Verifying:** This step involves compiling and running the application

Scanning Process

1. Script “hipexamine-perl.sh” is responsible for scanning code in a particular directory and subdirectories:

    - Goes through all source and header files
    - Its present in /opt/rocm/bin
    - Prints what the tool has detected as possible conversion candidates form CUDA to HIP

2. The script will show a output similar to what is shown in the next slide(image from conversion of a vector add in CUDA):

    - Basically examines what CUDA APIs can be converted and if there are any warnings

```
  info: converted 17 CUDA->HIP refs ( error:0 init:0 version:0 device:1 context:0 module:0 memory:9 virtual_memory:0 addressing
:0 stream:0 event:0 external_resource_interop:0 stream_memory:0 execution:0 graph:0 occupancy:0 texture:0 surface:0 peer:0 grap
hics:0 profiler:0 openGL:0 D3D9:0 D3D10:0 D3D11:0 VDPAU:0 EGL:0 thread:0 complex:0 library:0 device_library:0 device_function:3
 include:0 include_cuda_main_header:0 type:0 literal:0 numeric_literal:3 define:0 extern_shared:0 kernel_launch:1 )
  warn:0 LOC:107 in './vadd_cuda.cu'
  hipMalloc 3
  hipMemcpy 3
  hipFree 3
  hipMemcpyHostToDevice 2
  hipMemcpyDeviceToHost 1
  hipLaunchKernelGGL 1
  hipDeviceSynchronize 1
```

- Looking the output we can see some interesting things
- First, we see that 17 CUDA calls were converted to HIP
- A complete breakdown of what those 17 correspond to are then provided in the brackets
- A count of how many HIP API calls of each type were added is also shown
- The value of warn is 0 which means all API calls were converted successfully

Porting Process

1. Script “hipconvertinplace-perl.sh” is responsible for converting code in a particular directory and subdirectories

2. It will go through all source and header files and do the actual conversion

3. It will generate two files for each file where it converted something
   - fileName.prehip: Actual file before the hipifying process
   - fileName: Converted file

4. If you modify the original CUDA source file, you must delete the file with the extension “.prehip” so that HIP can generate the HIP source file correctly

Compiling Converted Code

- Compiling process remains the same
- No difference from compiling native HIP code
- “hipcc” is the compiler again
- Depending upon the application, correct flags must be passed

Learning with Example: Using Vector Add

1. We now know the steps involved in the hipification process
2. So let us apply it to a simple vector Add application written in CUDA
3. This is the same application we saw at the start of Module 3
4. Now the goal is to start with a CUDA version and run it on a ROCm GPU
5. Please follow Chapter 4.1 Porting CUDA Vector Add to HIP for the hands-on tutorial demonstrating the process of hipifying

Demo: Vector Add Conversion from CUDA to HIP

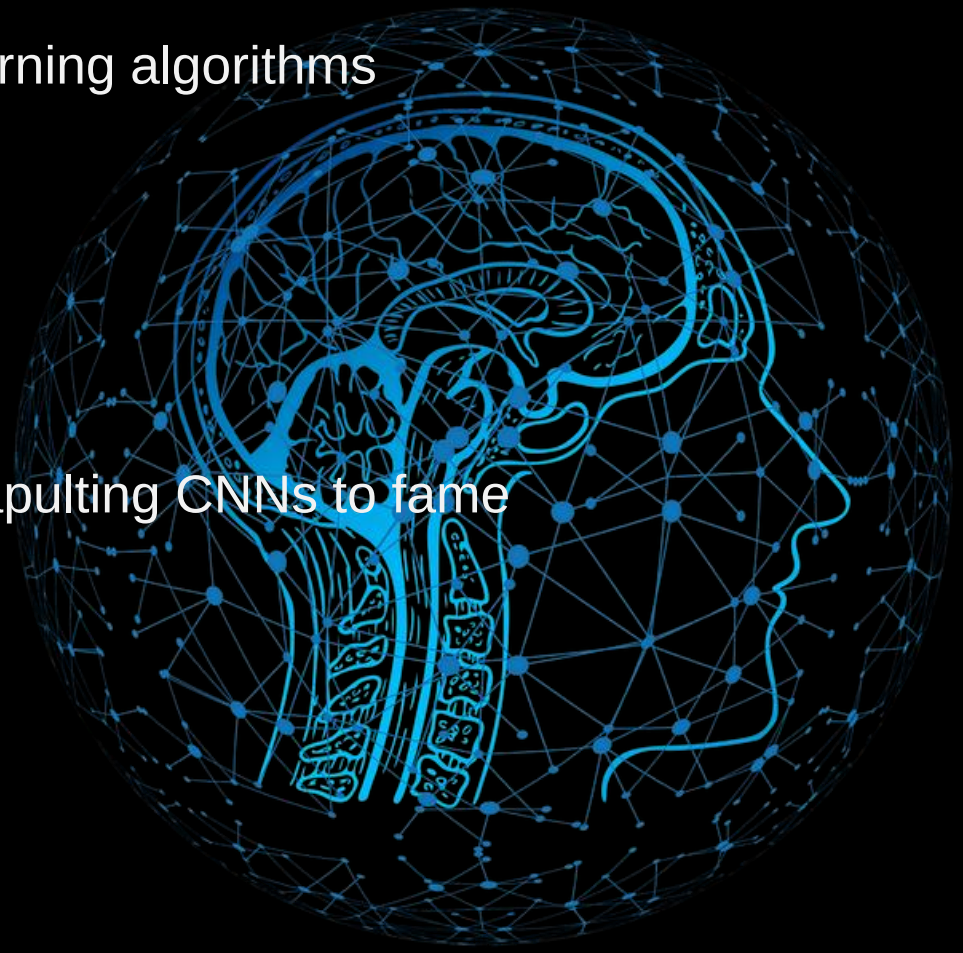
Converting two popular Machine Learning applications to HIP

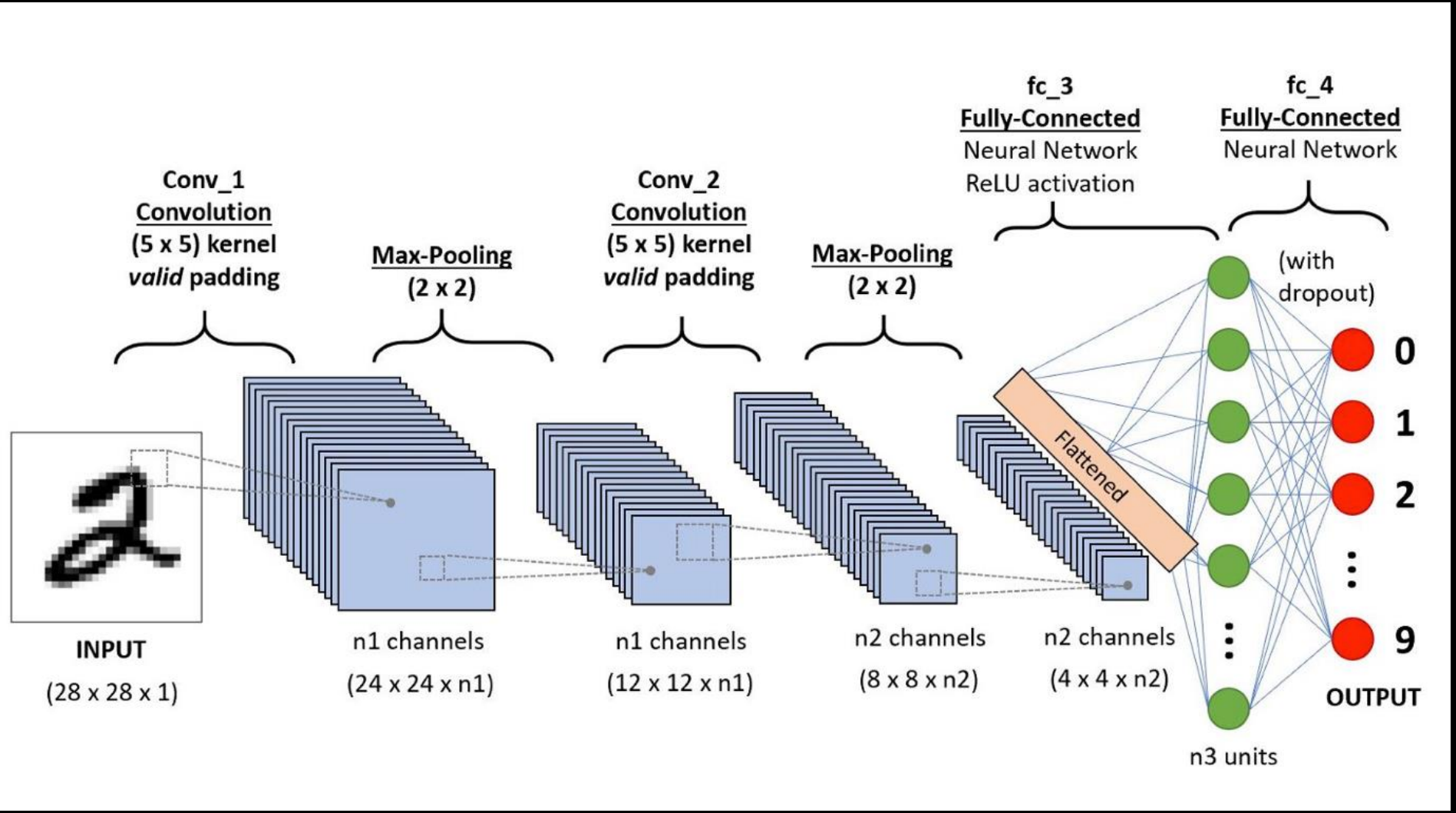
Goals

- In this tutorial we will be using HIP to convert machine learning applications
- We will be taking two applications from public GitHub repositories written in CUDA
  - A Convolutional Neural Network
  - KMeans Clustering Algorithm
- These examples will enable you to get a more hands on experience with the hipifying process
- You will also learn to install HIP libraries
- Converting CUDA based Makefiles to HIP will also be covered

Convolutional Neural Network (CNN)

- CNNs are one of the most popular class of machine learning algorithms
- They are widely used for:
  - Image and Video Recognition
  - Action Recognition
  - Document Classification
- GPU acceleration in the 2000s was responsible for catapulting CNNs to fame
- Some popular CNN architectures are:
  - Alexnet
  - VGG 16 and 19
  - Resnet
  - Inception





A Convolutional Neural Network to identify numbers

Porting CUDA CNN to HIP

- For this tutorial, we will take an open source implementation of a CNN available from github:
  - https://github.com/catchchaos/CUDA-CNN
- This CNN is designed to classify numeric digits by training on the popular MNIST dataset
- We will reuse the same tools for conversion i.e. the “hipconvertinplace-perl.sh” script
- This program also uses the BLAS libraries(hipblas)
  - BLAS libraries contains basic linear algebra subroutines
  - Installation instructions can be found in Chapter 4.2 of the hands-on manual

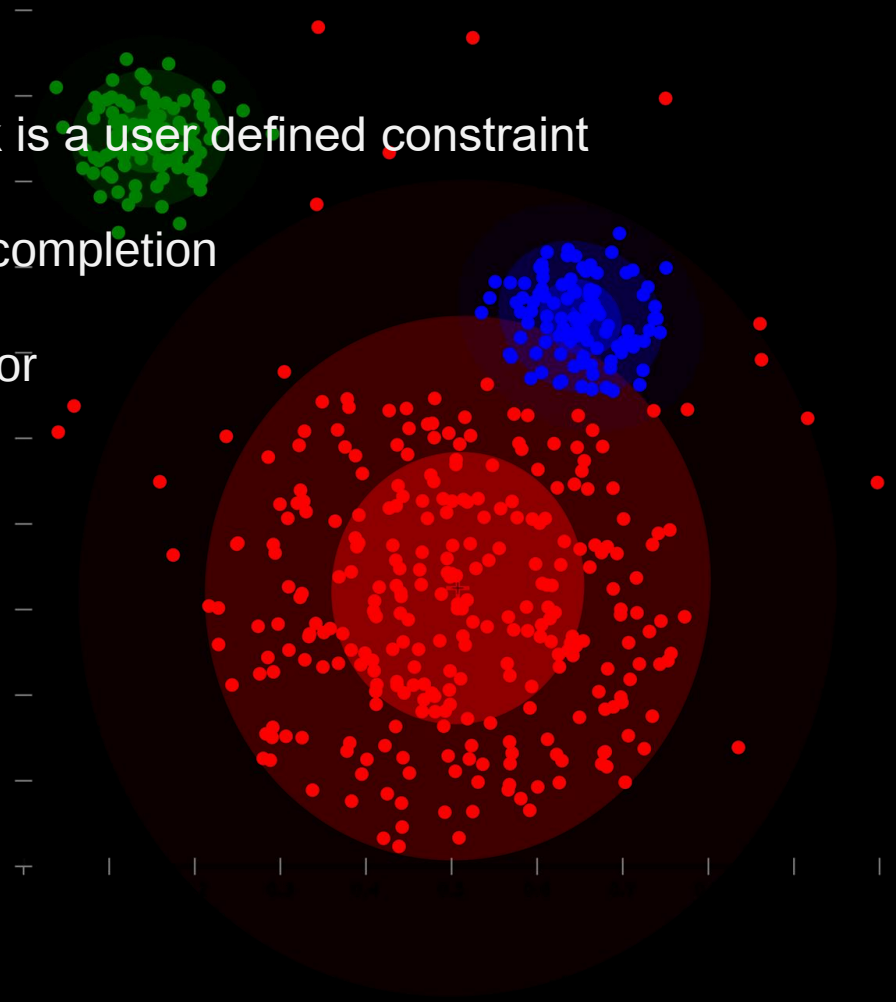
Modifying Makefiles for Compilation

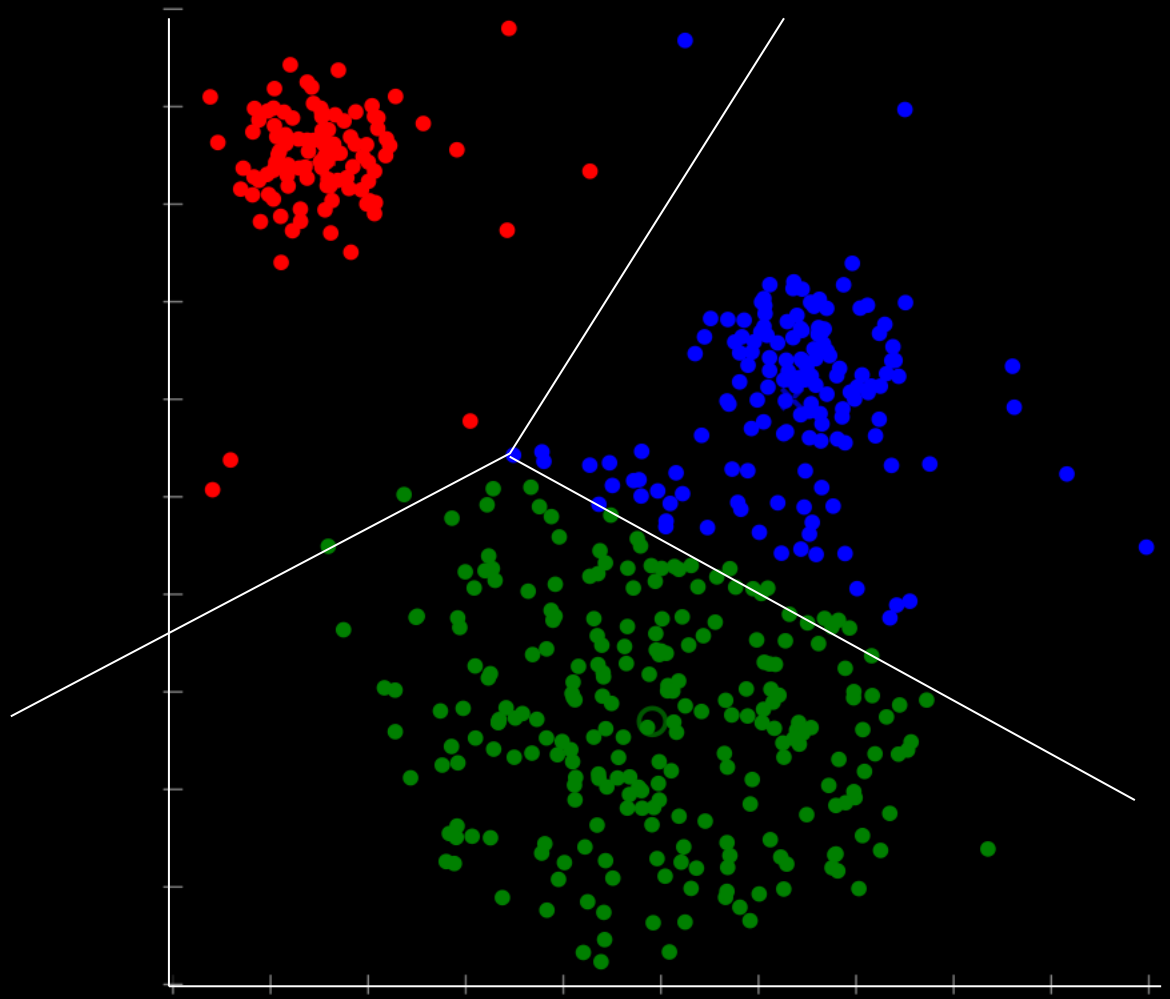
- Open source programs written in CUDA will typically have a Makefile for compilation
- This Makefile is not automatically converted by HIP
- Required modifications:
  - Compiler : nvcc -> hipcc
  - Any NVIDIA library to HIP library. For example cublas -> hipblas
  - Any CUDA specific library paths to /opt/rocm/lib
  - Replace any arch=compute_xx or CUDA specific compiler flags with equivalent HIP flags
  - Ensure ROCM paths are included in the makefile: -I/opt/rocm/include
  - More porting for specific CUDA options are available here https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-porting-guide.html
- Please follow the PDF “Chapter 4.2: Porting CUDA CNN to HIP” for the hands-on tutorial

Demo: Convolutional Neural Network Code Conversion from CUDA to HIP

KMeans

- KMeans is one of the most popular unsupervised Machine Learning Algorithms based on the idea of clustering
- Algorithm clusters “n” observations into “k” clusters where k is a user defined constraint
  - Similar elements are clustered closer to each other at completion
- Being highly parallel in nature, KMeans is great candidate for GPU acceleration
- Some common applications for KMeans are:
  - Document classification
  - Insurance fraud detection
  - Identifying crime localities
  - Rideshare data analysis





KMeans Algorithm classifying data into three different clusters (red, blue and green)

Porting CUDA KMeans to HIP

- For this tutorial, we will take an open source implementation of a KMeans available from GitHub:
  - https://github.com/serban/kmeans
- This KMeans implementation is working on clustering image data
- Its based on the parallel KMeans implementation:
  - http://users.eecs.northwestern.edu/~wkliao/Kmeans/index.html
- We will still reuse the same tools for conversion i.e. the “hipconvertinplace-perl.sh” script
- We will also need to modify the Makefile like how we did in the last example
- Please follow the pdf “Chapter 4.3: Porting CUDA KMeans to HIP” for the hands-on tutorial

Demo: KMeans conversion from CUDA to HIP

Does HIP port internal CUDA libraries?

1. Many GPU kernels from CUDA libraries are propietary to NVIDIA:
   - CUDNN, CuBlas, CuSparse etc

2. HIP does not port the same internal kernels
   - Not possible to port without access to source code
   - But it maintains the same interface/API for the library calls

3. Internally ROCms libraries MIOpen, hipBlas, hipSparse etc provide kernels similar to CUDA at similar performance level
   - Libraries developed and maintained by AMDs engineers

Conclusion

1. In this tutorial module we have learnt:
   - Basics of porting CUDA code to HIP
   - Understanding the tools involved in the hipifying process
   - Examples of porting CUDA code with varying complexity

2. In the next module, we are going to take a look at how to use Deep Learning frameworks on ROCm GPUs