Part 3: HIP Coding
Introduction

- The Heterogeneous Interface for Portability (HIP) is AMD’s dedicated GPU programming environment for designing high performance kernels on GPU hardware.

- HIP is a C++ runtime API and programming language that allows developers to create portable applications on AMD and NVIDIA platforms.

- This means developers can write their GPU applications and with very minimal changes be able to run their code in either environment.

- Syntax wise, HIP is similar to CUDA and has virtually no performance overhead on NVIDIA systems.

- HIP includes a rich set of libraries and tools support.
Prerequisites

- Basic understanding of GPU programming

- Ensure that:
  - You have access to a ROCm enabled GPU
  - ROCm and HIP is correctly installed based on the information found in the installation guides
Ensure that:
All online guides for ROCm can be found at: https://rocmdocs.amd.com/en/latest/

The following links are helpful for detailed information and latest updates:


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

- A comprehensive overview of porting CUDA code to HIP: https://github.com/ROCm-Developer-Tools/HIP/blob/master/docs/markdown/hip_porting_guide.md


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

- Basics of the GPU programming and execution model
- Developing GPU applications in HIP for ROCm using the HIP APIs
- Optimizing GPU applications using shared memory
- Profiling GPU applications
- Debugging Page Not Present Errors for GPU applications
GPU Programming Model
GPU Programming Model

Program for each thread

```c
// HIP kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

// Execute the kernel
hipLaunchKernelGGL(vecAdd, dim3(gridSize), dim3(blockSize), 0, 0, d_a, d_b, d_c, n);
hipDeviceSynchronize();
```
GPU Programming Model

```
// Execute the kernel
hipLaunchKernelGGL(vecAdd, dim3(gridSize), dim3(blockSize), 0, 0, d_a, d_b, d_c, n);
hipDeviceSynchronize();
```

- Launching a kernel forms a grid of threads
  - GridSize X BlockSize

```
ID = 0  ID = 1  ID = 2  ...  ID = N
```
## GPU Programming Model

- Grids can be 1D, 2D, or 3D
- Match with the underlying problem

<table>
<thead>
<tr>
<th>IDx</th>
<th>IDy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>Ny</td>
</tr>
<tr>
<td>1</td>
<td>Ny</td>
</tr>
<tr>
<td>2</td>
<td>Ny</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>Ny</td>
</tr>
</tbody>
</table>

- Match with the underlying problem
Inter-Thread Communication

- Shared Memory
  - In-core
  - Independently addressed
  - Shared by all the threads in a block
  - Helps to avoid going to the main memory for repeated memory access

```c
__global__ void shared_memory(int *d, int n) {
  // Allocate shared memory
  __shared__ int s[64];

  int t = threadIdx.x;
  int tr = n - t - 1;

  // Write into shared memory
  s[t] = d[t];

  __syncthreads();

  // Read from shared memory
  d[t] = s[t];
}
```
Synchronization with Barriers

- **Barriers**
  - Synchronizes all threads in a block

- **Kernel-level synchronization**
  - Need to stop the kernel
  - Launch a new kernel

---

```c
__global__ void shared_memory(int *d, int n) {
  // Allocate shared memory
  __shared__ int s[64];

  int t = threadIdx.x;
  int tr = n - t - 1;

  // Write into shared memory
  s[t] = d[t];

  __syncthreads();

  // Read from shared memory
  d[t] = s[t];
}
```
How a GPU Executes Kernels

Necessary knowledge for performance optimization
Blocks Dispatching

- GPU's Command Processor
  - Breaks down kernels to blocks
  - Dispatches blocks to Compute Unit

- Block executes on Compute Units
  - Threads from one block execute on the same Compute Unit
  - One Compute Unit can execute multiple Blocks
  - A kernel can have more blocks that the Compute Unit can fit
SIMD Instruction Execution

- 64 threads in a block are scheduled together
  - Warp / wavefront
  - Single instruction multiple data

Cycle 2

Cycle 1

SIMD Unit

...
SIMD Instruction Execution

- 64 threads in a block are scheduled together
  - Warp / wavefront
  - Single instruction multiple data
Memory Access Coalescing

- Combine memory access to the same cache line
- Increase effective memory throughput
What’s Next

We have learnt the basics of the GPU programming model.

For those familiar with CUDA, the programming model is similar.

Now we will be using the knowledge we learnt for developing real applications.
Developing GPU applications in HIP: Vector Add Example
Goals

- We will be looking at how to write a GPU application using the HIP APIs
- A simple vector add application will be used to understand the process
- We will also be looking at how to compile HIP applications
Vector Add

- We will write a very simple Vector Add GPU application in HIP
- The code simply computes the value of $a[i] + b[i]$ over a range of different values and stores it in $c[i]$
- The code is in “Chapter3/01_HIP_Vector_Add/Vector_Add_GPU”
Each thread in the kernel will take one element of \(a\) and one element of \(b\) and add them to produce one element of \(c\).
Vector Add: Header and Macros

- First thing to observe in the GPU version is the inclusion of a new header “#include `<hip/hip_runtime.h>`” in the HIP version.

- This header file is necessary to use the HIP runtime calls

- We will also add a macro “HIP_ASSERT” to check if any of the runtime API calls fail
  This is helpful to catch errors during development

```c
#include "hip/hip_runtime.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define HIP_ASSERT(x) (assert((x)==hipSuccess))
```
Vector Add: Memory Allocation

- Next step is to allocate memory on the GPU

- This is achieved by using the runtime call “hipMalloc”

- For example: using the API call `hipMalloc(&d_a, bytes)` will allocate the variable `d_a` and reserve a total of “bytes” storage for it

```c
// Allocate memory for each vector on GPU
HIP_ASSERT(hipMalloc(&d_a, bytes));
HIP_ASSERT(hipMalloc(&d_b, bytes));
HIP_ASSERT(hipMalloc(&d_c, bytes));
```
Next, we want to transfer the initialized data to the GPU

- Done using the API call `hipMemcpy`

- For example: doing `hipMemcpy(d_a, h_a, arraySize, hipMemCpyHostToDevice)` will copy a total of "bytes" bytes from the host array "h_a" to the device array "d_a"

```c
// Copy host vectors to device
HIP_ASSERT(hipMemcpy( d_a, h_a, bytes, hipMemcpyHostToDevice));
HIP_ASSERT(hipMemcpy(d_b, h_b, bytes, hipMemcpyHostToDevice));
```
Vector Add: Kernel Code

- Now we will add the kernel that is responsible for performing computations on the device.
- Each thread will be responsible for doing one addition between \( d_a[i] + d_b[i] \) and the corresponding output will be stored in \( d_c[i] \).
- The qualifier `__global__` means this function is for the GPU.
- The if condition inside this function ensures we are not accessing any out of bounds element which can trigger a segmentation fault.

```c
// HIP kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
    c[id] = a[id] + b[id];
}
```
Vector Add: Kernel Launch

- First, we need to define the grid size and block size for the kernel.

- The “vecAdd” kernel is then launched using the macro `hipLaunchKernelGGL`.
  

- `hipDeviceSynchronize` ensures that the computation on the GPU is complete.

```c
// Execute the kernel
hipLaunchKernelGGL(vecaAdd, dim3(gridSize), dim3(blockSize), 0, 0, d_a, d_b, d_c, n);
hipDeviceSynchronize();
```
Vector Add: Memory Copy from the GPU

- Now we are ready to copy the data back from the GPU to the CPU and store it in h_c.
- We will use `hipMemcpy` again to achieve this.

```c
// Copy array back to host
HIP_ASSERT(hipMemcpy( h_c, d_c, bytes, hipMemcpyDeviceToHost));
```
Vector Add: Result Verification

- It is recommended to verify the result of a kernel with its corresponding CPU counterpart.
- When using float variables, there is a chance of mismatches due to variances in precision.
- Thus, the comparison should be done within an error tolerance.

```c
//Compute for CPU
for(i=0; i <n; i++)
{
    h_verify_c[i] = h_a[i] + h_b[i];
}

//Verify results
for(i=0; i <n; i++)
{
    if (abs(h_verify_c[i] - h_c[i]) > 1e-5)
    {
        printf("Error at position i %d, Expected: %f, Found: %f \n", i, h_c[i], d_c[i]);
    }
}
```
Vector Add: Freeing Memory

- For best practices it is recommended to free the device memory once the work has been done.
- This is done by using the API call “hipFree”.

```c
// Release device memory
HIP_ASSERT(hipFree(d_a));
HIP_ASSERT(hipFree(d_b));
HIP_ASSERT(hipFree(d_c));
```
Vector Add: Compiling and Running

- Now that the application is ready, we will compile it using hipcc which is the compiler for HIP
  - Run “hipcc vadd_hip.cpp -o vadd_hip”
  - This will produce the binary “vadd_hip” which can be executed by “./vadd_hip”
  - The hands-on tutorial for this example is provided in “Chapter 3.1: Vector Add in HIP”
Printing output from the GPU side

- There are times where you will want to inspect the values of some buffer

- For example:
  - Debugging for incorrect result
  - Viewing intermediate results in a long kernel

- A simple example on how to achieve this functionality for the vector add application is provided in the tutorial repo
Additional Tips on Device Side Printing

- Device side printing can be helpful for:
  - To check the contents of arrays if you have incorrect output
  - To debug out of bound accesses

- However, if problem sizes are large and many threads are launched; programs can slow down significantly as all threads are waiting to write their output into a buffer.
Demo: Demonstration of Vector Add and Kernel printing in HIP
Optimizing a matrix transpose application using guided-profiling in HIP
Goals

- Determining the bottlenecks of an application can be tricky
- A profiler can help in understanding the application characteristics
- We will be looking at how to do such profile guided optimizations
Matrix Transpose

1. In this section, we will be developing a matrix transpose example using HIP
   • It is a common routine in many scientific application

2. Transpose of a matrix flips the matrix over its diagonal

3. Can benefit significantly from GPU acceleration

4. Naive implementation is not high performant

5. We will be using shared memory in HIP to accelerate Matrix Transpose

6. We will be learning how we can leverage the insights provided by ROCm profiler to optimize our application step by step
Demonstration of Matrix Transpose of a 2 x 2 matrix. As seen, the transpose of a matrix flips the matrix over its diagonal.
ROCM Profiler

1. ROCM profiler( https://github.com/ROCm-Developer-Tools/rocprofiler ) enables profiling support on AMD GPU

2. Insights captured from profiling can help to:
   - Understand application behavior
   - Understand resource usage
   - Understand bottlenecks

3. ROCM profiler provides such insights into an application

4. Two common modes:
   - Performance Measurement Mode to measure execution time
   - Performance Counter Mode to measure hardware performance counters.
ROCm Profiler: Performance Execution Mode

1. In this mode, kernel performance can be collected
2. Useful to see what are the dominant kernels in an application
3. Provides per kernel execution time stats
ROCm Profiler: Performance Counter Mode

1. In this mode, the profiler uses hardware performance counters to collect metrics

2. Example of such counters:
   - GPU Compute Busy Time
   - Memory Transactions
   - Cache Statistics
   - Register Usage

3. Can be used to study the application bottlenecks in detail by understanding resource usage
Application of Interest

1. Three different application kernels are going to be studied in the upcoming slides
   • All are related to the matrix-transpose application

2. Copy Kernel
   • Copies the elements from one GPU array to another
   • We would like to achieve similar performance in our Matrix Transpose

3. Matrix Transpose Naive:
   • Naive version that uses no optimization and has poor performance

4. Matrix Transpose Shared Memory:
   • Optimized version that uses shared memory/LDS on the GPU
Host Side Code

```c
hipMalloc((void **)&d_in, width * height * sizeof(float));
hipMalloc((void **)&d_out, width * height * sizeof(float));

hipMemcpy(d_in, matrix_in.data(), width * height * sizeof(float),
    hipMemcpyHostToDevice);

int block_x = width / tile_dim;
int block_y = height / tile_dim;

//GPU Kernel Launch

hipMemcpy(matrix_out.data(), d_out, width * height * sizeof(float),
    hipMemcpyDeviceToHost);
hipDeviceSynchronize();
```
Host Side Code

1. CPU/Host side code is similar to the previous example
2. We allocate both the input and output for the GPU
3. Then we copy the input buffer to GPU.
4. As observed, we are allocating a flattened array on the GPU
5. So we need to ensure our kernel code uses the correct index

Flattened array:

```
1 2

3 4
```
HIP Copy Kernel

1. The copy kernel just performs a matrix copy by copying the value from “in” to “out”

2. We will be using this kernel for reference to compare performance

3. Ideally, matrix transpose and matrix copy should read and write the same amount of data

4. However, as we will observe a naive transpose implementation does not achieve this
A HIP Copy Kernel

```c
__global__ void copy_kernel(float *in, float *out, int width, int height) {
    int x_index = blockIdx.x * tile_dim + threadIdx.x;
    int y_index = blockIdx.y * tile_dim + threadIdx.y;

    int index = y_index * width + x_index;

    out[index] = in[index];
}
```
HIP Copy Kernel

1. Looking at the “copy kernel” we can observe a few things

2. We are obtaining the current x index and y index of the thread

3. We are then calculating the flattened index into the array

4. Finally, we are assigning a value of the output array from the input array

5. Let us now compile, run and profile this application

6. Please follow **Chapter 3.2_Copy_KERNEL** in your hands-on manual for the experiment
Demo: Understanding the performance of the Copy Kernel
Copy Kernel

1. We have just seen how we can collect some performance metrics. These are tabulated below

<table>
<thead>
<tr>
<th>App Name</th>
<th>Time(ns)</th>
<th>Read Transactions</th>
<th>Write Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy Kernel</td>
<td>280798</td>
<td>524312</td>
<td>524288</td>
</tr>
</tbody>
</table>

2. There are ~0.5 million read and write memory transactions

3. Ideally, we want our matrix-transpose to have similar amount of memory transactions and performance

4. We will now look at our naive Matrix-Transpose
Matrix Transpose

1. Let us convert our copy kernel to do an actual transpose

2. We simply calculate and use different input and output indexes as shown below

```c
__global__ void transpose_kernel(float *in, float *out, int width, int height) {
    int x_index = blockIdx.x * tile_dim + threadIdx.x;
    int y_index = blockIdx.y * tile_dim + threadIdx.y;

    int in_index = y_index * width + x_index;
    int out_index = x_index * height + y_index;

    out[out_index] = in[in_index];
}
```
Matrix Transpose

1. Let us analyze the last statement:
   • \text{out[out\_index]} = \text{in[in\_index]};

2. The compiler can convert this line of code into 3 instructions.
   • A load instruction that reads the element from the input matrix, an assignment instruction
   • A store instruction that writes the element to the write matrix

3. Load instruction:
   • Adjacent threads will read from adjacent memory
   • High coalescing

4. Store instruction:
   • Adjacent threads write to elements not adjacent in memory
   • No coalescing
Matrix Transpose

1. Now that the kernel is understood, let us run, compile and profile the application

2. We will be using rocProf to profile the application characteristics again

3. Please follow Chapter 3.2_Matrix_Transpose_Naive_Kernel in your hands-on manual for the experiment
Demo: Understanding the performance of the Naive Matrix Transpose Kernel
Matrix Transpose

1. Let us now compare the results so far

<table>
<thead>
<tr>
<th>App Name</th>
<th>Time(ns)</th>
<th>Read Transactions</th>
<th>Write Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy Kernel</td>
<td>279519</td>
<td>529227</td>
<td>524288</td>
</tr>
<tr>
<td>Matrix Transpose</td>
<td>15135463</td>
<td>524289</td>
<td>1686190</td>
</tr>
</tbody>
</table>

2. Naive matrix-transpose performs worse than the copy kernel

3. 54x slowdown compared to copy kernel

4. 3x more writes to memory by the naive version
   • Due to the loss of coalescing ability.

5. How can we solve this issue?
Local Data Share (LDS)

1. A user-managed cache is available on AMD GPUs
2. Enables data-sharing within threads that belong to the same thread-block
3. Similar to shared memory on NVIDIA systems
4. Reads and writes are 100x faster when compared to global memory
5. Let us optimize our naive transpose with LDS
Matrix Transpose using LDS

```c
__global__ void transpose_lds_kernel(float *in, float *out, int width, int height) {
    __shared__ float tile[tile_dim][tile_dim];

    int x_tile_index = blockIdx.x * tile_dim;
    int y_tile_index = blockIdx.y * tile_dim;

    int in_index =
        (y_tile_index + threadIdx.y) * width + (x_tile_index + threadIdx.x);
    int out_index =
        (x_tile_index + threadIdx.y) * height + (y_tile_index + threadIdx.x);

    tile[threadIdx.y][threadIdx.x] = in[in_index];
    __syncthreads();
    out[out_index] = tile[threadIdx.x][threadIdx.y];
}
```
Matrix Transpose using LDS

1. LDS allocation
   - With the “__shared__ modifier”
   - With respect to a thread block
   - Threads that are part of the thread block can use this data

2. Here, size of shared data is set equal to block size

3. Kernel works as follows:
   - Each thread loads one data element and stores the data in the shared memory
   - Then, each thread writes one element to the output matrix.
   - Both reads and writes are coalescable

4. syncthreads
   - Ensure that threads in the same block have completed their writes before the output array is populated
Matrix Transpose using LDS

1. Now that the kernel is understood, let us run, compile and profile the application

2. Again we will be using rocProf for the experiments

3. Please follow Chapter 3.2_Matrix_Transpose_LDS_Kernel in your hands-on manual for the experiment
Demo: Understanding the performance of the Matrix Transpose Kernel that uses LDS
Matrix Transpose LDS

1. Let us now compare all the results

<table>
<thead>
<tr>
<th>App Name</th>
<th>Time(ns)</th>
<th>Read Transactions</th>
<th>Write Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy Kernel</td>
<td>279519</td>
<td>529227</td>
<td>524288</td>
</tr>
<tr>
<td>Matrix Transpose</td>
<td>15198186</td>
<td>528347</td>
<td>1551894</td>
</tr>
<tr>
<td>Matrix Transpose LDS</td>
<td>3908787</td>
<td>524313</td>
<td>524288</td>
</tr>
</tbody>
</table>

2. LDS version has a speedup of ~4x times over the naive version

3. Slowdown reduced to 14x from copy kernel (naive version is 52x slower)

4. The number of write transactions similar to copy kernel

5. Why it is still slower than Copy Kernel?
   - LDS bank conflict is the primary reason
   - More optimizations required
Takeaways

1. Profile guided application optimization is very useful for development

2. Goal is to start with a simple version
   • Profile and understand the bottlenecks
   • Use features such as shared memory to improve performance
   • Analyze optimized versions under profiler
   • Rinse and repeat the process until you have the desired performance

3. Writing high-performant GPU kernels is a complex challenge

4. But using a profiler:
   • Prevents second guessing
   • Enables focused development
Debugging Page Not Present Errors
Goals

- Programming mistakes while writing GPU applications are common
- This can result in two common issues:
  - Program Crashes (Segmentation Faults)
  - Incorrect results
- We will be dealing with debugging program crashes
- ROCm provides functionality to debug these kernels by figuring out the faulty kernel
Debugging Basics

- We will be using the modified version of add4 application from HIP-Examples https://github.com/ROCm-Developer-Tools/HIP-Examples

- This application is intended to evaluate the memory transfer rates to and from the GPU global memory
  - It is part of the GPU-Stream benchmark suite https://github.com/pfultz2/GPU-STREAM

- The original version has been modified to run into issues to demonstrate the debugging method

- Please follow the PDF “Chapter 3.3:Debugging_Page_Not_Present_Errors” for the upcoming hands on tutorial
Demo: Demonstration of Debugging Page Not Present Errors
Debugging: Tips

- Always double check your memory allocation commands
- You can also use device side `printfs` to debug complex kernels
  - Useful if you are receiving incorrect results
  - Using prints with many threads can result in a kernel slowdown
    - If you decide to use `printf`, reduce the data size processed by the kernel and number of threads first
    - Then, debug your algorithm using kernel prints to fix the functionality
Conclusion

- In this module we have looked at the basics of the GPU programming model
- We looked at how to write GPU applications for ROCm using HIP
- We looked at the process of doing profile guided optimization for developing efficient GPU code
- We looked at how to debug errors on ROCm
- In the next module, we will be looking at how to port code from CUDA to HIP