Lab 4: Introduction to CUDA Programming

ECE 455: GPU Algorithm and System Design

Due: Submit completed PDF to Canvas by 23:25 PM 10/10

Overview

This lab introduces CUDA, NVIDIA's parallel computing platform and API for programming GPUs. You will learn basic CUDA kernel programming, kernel launch configuration, and asynchronous execution with streams.

Learning Objectives

In this lab, students will:

- Understand the basic structure of a CUDA program, including host code and device kernels.
- Learn how to configure kernel execution using grid and block dimensions.
- Gain experience in writing and launching simple CUDA kernels.
- Explore the use of CUDA streams for overlapping computation and data transfer.
- Implement and test basic parallel operations such as vector addition and SAXPY.
- Practice compiling and running CUDA programs using `nvcc` on the SLURM cluster.
- Develop skills to debug and verify correctness of GPU-based computations.

Euler Instruction

```
~$ ssh your_CAE_account@euler.engr.wisc.edu
~$ sbatch your_slurm_scrip.slurm
```

You should NEVER run your program on the log-in node with the interactive mode. Doing so will risk your account being blocked by the IT. Instead, you should work on your local machine and set up a GitHub repo to transfer code from your local machine to your Euler node, and then compile and run it using a proper `sbatch` script.

Submission Instruction

Specify your GitHub link here: https://github.com/boomchrisvanden/ECE455/HW04

Note that your link should be of this format: `https://github.com/YourGitHubName/ECE455/HW04`

Problem 1: Hello World from GPU

Task: Write a CUDA kernel that prints "Hello from thread X" for each GPU thread.

Solution

hello.cu

```
#include <stdio.h>

__global__ void hello_kernel() {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    printf("Hello from thread %d\n", tid);
}

int main() {
    hello_kernel<<<2, 4>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

p1.slurm

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:01:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --time=00:01:00
#SBATCH --output=hello.output

module load nvidia/cuda
nvcc hello.cu -o hello
./hello
```

Problem 2: Understanding Thread Indexing

Task: Launch a kernel with a 2D grid and block. Each thread should print its $(blockIdx.x, blockIdx.y)$ and $(threadIdx.x, threadIdx.y)$.

Solution

thread_indexing.cu

```
#include <stdio.h>

__global__ void print_indices() {
    printf("Block(%d,%d) Thread(%d,%d)\n",
           blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y);
}
```

```
int main() {
    dim3 blocks(2, 2);
    dim3 threads(2, 3);
    print_indices<<<blocks, threads>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

p2.slurm

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:01:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --time=00:01:00
#SBATCH --output=thread_indexing.output

module load nvidia/cuda
nvcc thread_indexing.cu -o thread_indexing
./thread_indexing
```

Problem 3: Vector Addition

Task: Implement a CUDA kernel for `C[i] = A[i] + B[i]` for $N = 1,000,000$.

Solution:

vector_add.cu

```
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void vector_add(const float *A, const float *B, float *C, int N
   ) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    int N = 1000000;
    size_t size = N * sizeof(float);

    float *h_A = (float*)malloc(size);
    float *h_B = (float*)malloc(size);
    float *h_C = (float*)malloc(size);

    for (int i = 0; i < N; i++) { h_A[i] = 1.0f; h_B[i] = 2.0f; }
```

```
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    vector_add<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    printf("C[0] = %f\n", h_C[0]);

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    free(h_A); free(h_B); free(h_C);
    return 0;
}
```

p3.slurm

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:01:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --time=00:01:00
#SBATCH --output=vector_add.output

module load nvidia/cuda
nvcc vector_add.cu -o vector_add
./vector_add
```

Problem 4: SAXPY Kernel

Task: Implement `y[i] = a * x[i] + y[i]` for $N = 1,000,000$ with $a = 2.0$.

Solution

saxpy.cu

```
#include <stdio.h>

__global__ void saxpy(int n, float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) y[i] = a * x[i] + y[i];
}

int main() {
```

```
    int N = 1000000;
    size_t size = N * sizeof(float);
    float *x, *y, *d_x, *d_y;

    x = (float*)malloc(size);
    y = (float*)malloc(size);
    for (int i = 0; i < N; i++) { x[i] = 1.0f; y[i] = 2.0f; }

    cudaMalloc(&d_x, size);
    cudaMalloc(&d_y, size);
    cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    saxpy<<<blocksPerGrid, threadsPerBlock>>>(N, 2.0f, d_x, d_y);

    cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
    printf("y[0] = %f\n", y[0]);

    cudaFree(d_x); cudaFree(d_y);
    free(x); free(y);
    return 0;
}
```

p4.slurm

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:01:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --time=00:01:00
#SBATCH --output=saxpy.output

module load nvidia/cuda
nvcc saxpy.cu -o saxpy
./saxpy
```

Problem 5: Using CUDA Streams

Task: Split a vector addition into two halves and execute each in its own CUDA stream.

Solution

vector_add_streams.cu

```
#include <stdio.h>

__global__ void vector_add(const float *A, const float *B, float *C, int N
    ) {
```

```
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}

int main() {
    int N = 1000000;
    size_t size = N * sizeof(float);
    float *A, *B, *C;
    float *d_A, *d_B, *d_C;

    A = (float*)malloc(size);
    B = (float*)malloc(size);
    C = (float*)malloc(size);
    for (int i = 0; i < N; i++) { A[i] = 1.0f; B[i] = 2.0f; }

    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1);
    cudaStreamCreate(&stream2);

    int half = N / 2;
    size_t half_size = size / 2;

    cudaMemcpyAsync(d_A, A, half_size, cudaMemcpyHostToDevice, stream1);
    cudaMemcpyAsync(d_B, B, half_size, cudaMemcpyHostToDevice, stream1);
    cudaMemcpyAsync(d_A + half, A + half, half_size,
       cudaMemcpyHostToDevice, stream2);
    cudaMemcpyAsync(d_B + half, B + half, half_size,
       cudaMemcpyHostToDevice, stream2);

    int threads = 256;
    int blocks_half = (half + threads - 1) / threads;
    vector_add<<<blocks_half, threads, 0, stream1>>>(d_A, d_B, d_C, half);
    vector_add<<<blocks_half, threads, 0, stream2>>>(d_A + half, d_B +
       half, d_C + half, half);

    cudaMemcpyAsync(C, d_C, half_size, cudaMemcpyDeviceToHost, stream1);
    cudaMemcpyAsync(C + half, d_C + half, half_size,
       cudaMemcpyDeviceToHost, stream2);

    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2);

    printf("C[0] = %f, C[N-1] = %f\n", C[0], C[N-1]);

    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
```

```
    free(A); free(B); free(C);
    return 0;
}
```

p5.slurm

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:01:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --time=00:01:00
#SBATCH --output=vector_add_streams.output

module load nvidia/cuda
nvcc vector_add_streams.cu -o vector_add_streams
./vector_add_streams
```

Problem 6

Describe the challenges you encounter when completing this lab assignment and how you overcome these challenges.

I was confused on how the CUDA memory handling methods work, so I had to consult doccumentation