Report of Project 4: GPU Computing

Zhan Wang
Department of Data Science
Fudan University
Shanghai, China 200433
17307110421@fudan.edu.cn

Abstract

This project completes 2 seperate tasks about GPU coding and CUDA programming. For each task, we introduce the detailed solutions and implement the corresponding codes. The analysis results are presented by directly printing and ploting. The whole project is done on C and CUDA programmers.

1 Environment & Code

This project is done on Ubuntu 16.04, gcc 5.4.0, Cuda compiler driver 10.1, and python 3.6.2. The detailed codes can be found in the file folder `codes`.

2 Question 1: Common Errors

In this section, we correct several pieces of inaccurate code. The corrected codes are shown for each piece and executed in C enviroment. All codes in this part are stored in file `pointer.c` with its executable file `pointer`.

2.1 Pointer-1

This task requires us to create an integer pointer, set the value to which it points to 3, add 2 to this value, and print said value. The original code sets the variable `a` as a pointer but tries to assign it a integer value, which leads to the compiler errors. Since a pointer should be assigned the address of what we'd like it to point to, not the variable itself, we get the address of a integer variable by putting an ampersand before the variable's name and assign the pointer this address.

```
void test1()
{
    int *a;
    int b = 3;
    a = &b;
    *a = *a + 2;
    printf("%d\n", *a);
}
```

2.2 Pointer-2

This piece of code creates two integer pointers and sets the values to which they point to 2 and 3, respectively. However, the original code create a integer pointer and a integer variable which is assigned a pointer, which results to the warnings and errors. Hence, we just replace the integer variable `b` with a pointer variable `*b`.

```
void test2()
{
    int *a, *b;
    a = (int *) malloc(sizeof (int));
    b = (int *) malloc(sizeof (int));
    if (!(a && b))
    {
        printf("Out of memory\n");
        exit(-1);
    }
    *a = 2;
    *b = 3;
}
```

2.3 Pointer-3

This task wants to allocate an array of 1000 integers, and for $i = 0, ..., 999$, sets the i-th element to i. However, the original code only allocates dynamic memory for 1000 bytes which is not enough for 1000 integers, which. Therefore, I just expanded the allocated memory to accommodate 1000 integers (4000 bytes).

```
void test3()
{
    int i, *a = (int *) malloc(1000 * sizeof (int));
    if (!a)
    {
        printf("Out of memory\n");
        exit(-1);
    }
    for (i = 0; i < 1000; i++)
        *(i + a) = i;
}
```

2.4 Pointer-4

This task would like to create a two-dimensional array of size 3x100, and sets element (1,1) (counting from 0) to 5. The original only allocates memory for a array of 5 pointer variables but no memory for each pointer. Hence, I added some code to allocate memory of a array of 100 integers for each pointer.

```
void test4()
{
    int **a = (int **) malloc(3 * sizeof (int *));
    for (int i = 0; i < 3; i++)
        a[i] = (int *) malloc(100 * sizeof(int));
    a[1][1] = 5;
}
```

2.5 Pointer-5

This task sets the value pointed to by a to an input, checks if the value pointed to by a is 0, and prints a message if it is. This piece of code has no compiler error but can not print "Value is 0" when I input zero. We can find the original code detects the address of the input value not the value itself, so we change the judgment object from `a` to `*a` in the `IF` sentence.

```
void test5()
{
    int *a = (int *) malloc(sizeof (int));
    scanf("%d", a);
    if (!*a)
```

```
        printf("Value is 0\n");
}
```

3 Question 2: Vector addition

In this section, we implemented the vector addition program both on the CPU and GPU in the CUDA programmers. We visualize the execution time for the CPU and GPU runs for better comparison. The CPU and GPU implementation are stored in files `addition_cpu.cu` and `addition_gpu.cu` respectively.

3.1 CPU implementation

In this part, all execution is done on the CPU. The vector length N is a command line input argument (argv) when running the executable file `addition_cpu`. We generate random numbers to create the input vectors A and B. Define A, B, and $C = A + B$ as single precision (float) arrays. Finally, output the sum s of all elements in C, $s = \sum_{i=0}^{N-1} c_i$. The code is stored in `addition_cpu.cu` with its corresponding executable file `addition_cpu`.

```
//
// Addition function
//

void VecAdd(float* A, float* B, float* C, int N)
{
    for (int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}

//
// main code
//

int main(int argc, char **argv)
{
    cudaSetDevice(1);
    // Input the vector length
    int N = atoi(argv[1]);

    // Number of bytes to allocate for N float
    size_t bytes = N*sizeof(float);

    // Generate randomly vectors A and B
    float *A = (float *)malloc(bytes);
    float *B = (float *)malloc(bytes);
    float *C = (float *)malloc(bytes);

    for (int i = 0; i < N; i++)
    {
        A[i] = rand()%100;
        B[i] = rand()%100;
    }

    VecAdd(A, B, C, N);

    int s = 0;
    for (int j = 0; j < N; j++) s += C[j];

    printf("\nCPU Vector Length: %d Sum: %d\n", N, s);

    // Free CPU memory
    free(A);
    free(B);
```

```
    free(C);

    // CUDA exit -- needed to flush printf write buffer
    cudaDeviceReset();

    return 1;
}
```

3.2 GPU implementation

In this part, we did the same task - vector addition in the above subsection but modified the computation $C = A + B$ to the GPU platform. The code is stored in `addition_gpu.cu` with its corresponding executable file `addition_gpu`.

```
//
// kernel routine
//

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

//
// main code
//

int main(int argc, char **argv)
{
    cudaSetDevice(1);
    // Input the vector length
    int N = atoi(argv[1]);

    // Number of bytes to allocate for N float
    size_t bytes = N*sizeof(float);

    // Generate randomly vectors A and B
    float *A = (float *)malloc(bytes);
    float *B = (float *)malloc(bytes);
    float *C = (float *)malloc(bytes);

    // Allocate memory for arrays d_A, d_B, and d_C on device
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    for (int i = 0; i < N; i++)
    {
        A[i] = rand()%100;
        B[i] = rand()%100;
    }

    cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

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

    // Copy data from device array d_C to host array C
    cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```

```
51
52      int s = 0;
53      for (int j = 0; j < N; j++) s += C[j];
54
55      printf("\nGPU Vector Length: %d Sum: %d\n", N, s);
56
57      // Free CPU memory
58      free(A);
59      free(B);
60      free(C);
61
62      // Free GPU memory
63      cudaFree(d_A);
64      cudaFree(d_B);
65      cudaFree(d_C);
66
67      // CUDA exit -- needed to flush printf write buffer
68      cudaDeviceReset();
69
70      return 1;
71  }
```

3.3 CPU vs GPU comparison

For better comparison between CPU and GPU, we use `time` command to execute the CPU and GPU code for different vector lengths $N = 10^6, 5 \times 10^6, 10^7, 5 \times 10^7$. The results are shown in the below figure. We also fit execution time t as a linear function of the vector length N and make a comparison plot.

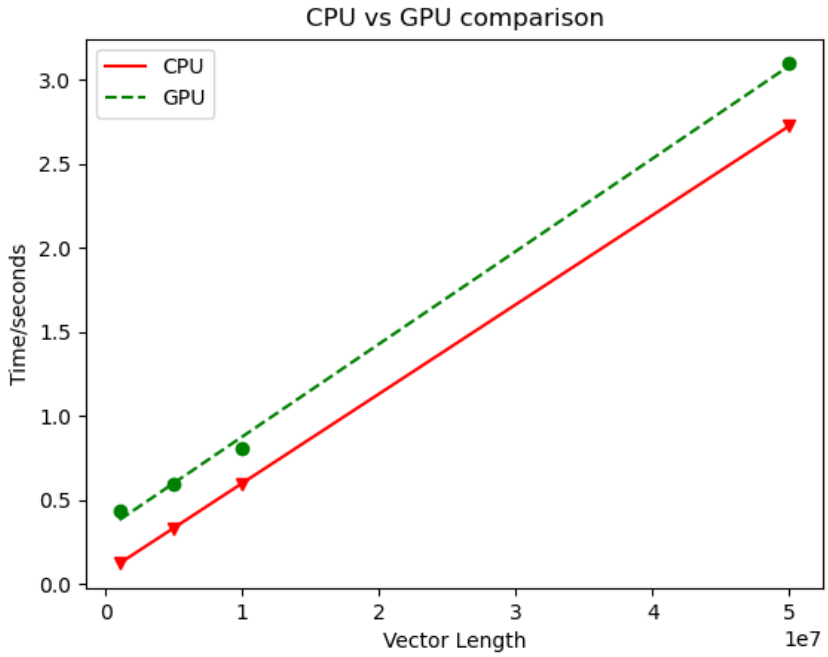

Figure 1: The fitted linear function of execution time: CPU vs GPU

Vector Length	CPU vs GPU	Sum	Time (Real)
n=10^6	CPU	98990536	0.127s
	GPU	98990536	0.441s
n=5×10^6	CPU	495059776	0.336s
	GPU	495059776	0.600s
n=10^7	CPU	991457984	0.601s
	GPU	991457984	0.813s
n=5×10^7	CPU	706973376	2.729s
	GPU	706973376	3.098s

Table 1: CPU and GPU implementations' sum result and execute time of different vector lengths.

3.4 Conclusions

From the above shown results, we can find the GPU code runs slower, which does not meet our expectations. A CPU is designed to handle a wide-range of tasks quickly (as measured by CPU clock speed), but are limited in the concurrency of tasks that can be running. A GPU is designed to quickly render high-resolution images and video concurrently. Considering that our code is simple, GPU implementation needs to spend time to transfer data between host and device, which may explain our unexpected results. Therefore, we suggest that CPU is much better when trying to solve most tasks and consider GPU when requiring the concurrency of tasks.

4 Makefile

I also implement a Makefile to execute all code files mentioned above.

```
INC := -I$(CUDA_HOME)/include -I.
LIB := -L$(CUDA_HOME)/lib64 -lcudart

NVCCFLAGS := -lineinfo -arch=sm_35 --ptxas-options=-v --use_fast_math
# NVCCFLAGS := -lineinfo -arch=sm_50 --ptxas-options=-v --use_fast_math

all:  pointer addition_cpu addition_gpu

pointer:  pointer.c Makefile
  gcc -Wall -std=c99 -o pointer pointer.c

addition_cpu: addition_cpu.cu Makefile
  nvcc addition_cpu.cu -o addition_cpu $(INC) $(NVCCFLAGS) $(LIB)

addition_gpu: addition_gpu.cu Makefile
  nvcc addition_gpu.cu -o addition_gpu $(INC) $(NVCCFLAGS) $(LIB)

clean:
  rm -f pointer addition_cpu addition_gpu
```