ECSE420 Lab1 Report

Liang Zhao 260781081
Hongshuo Zhou 260792817

October 19 2020

Group 6

1 Introduction

Generally in this lab, we are required to implement a logic gate simulation cuda program. We need to write code that simulates logic gate sequentially and then parallelize our code using explicit memory allocation and unified memory allocation.

2 Design And Testing

2.1 Implementation

For this lab, we split the code into three individual block: *sequential.cu*, $Parallel1explicit$,$Parallel1unified$.

2.2 Sequential

Our sequential implementation is to directly loop through the input file, and generates the output from the evaluation function. For input A and B, the output is computed as follows:

AND	OR	NAND	XOR	NOR	XNOR
A.B	A+B	$\overline{A.B}$	A $\bigoplus B$	$\overline{A+B}$	$\overline{A \bigoplus B}$

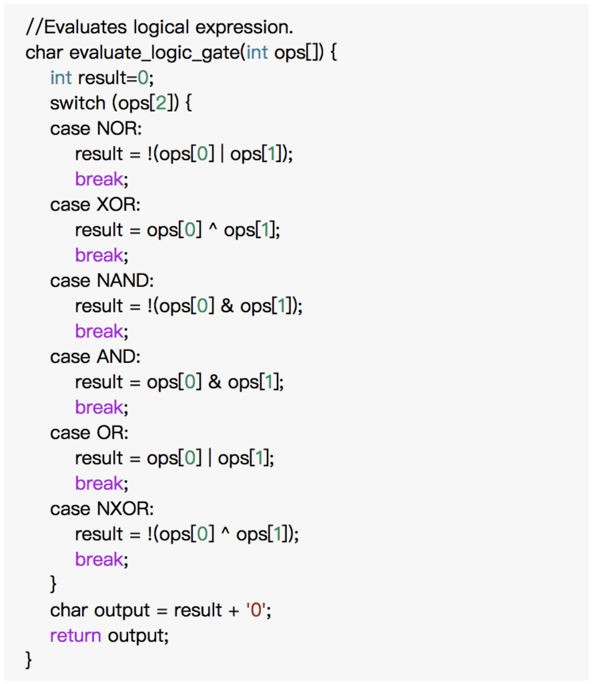
Table 1: Execution Time

Below is the execution time table, as required, we only report the execution time of the computation, and exclude reading and writing times. We can observe that the execution time increased as the input size increased. It is also worth mentioning that, the ratio between the execution time for different input file is close to the ratio of their file size. It is also the evidence that our code computes the result correctly, and the sequential code shall save no execution time.

Input Size	Execution time(ms)
10000	1.42
100000	13.22
1000000	137.56

Table 2: logic gate table

Here is our code snippet of logic gates computation implementation:
The function takes input boolean values and logic gate, then evaluate the



```
//Evaluates logical expression.
char evaluate_logic_gate(int ops[]) {
    int result=0;
    switch (ops[2]) {
    case NOR:
        result = !(ops[0] | ops[1]);
        break;
    case XOR:
        result = ops[0] ^ ops[1];
        break;
    case NAND:
        result = !(ops[0] & ops[1]);
        break;
    case AND:
        result = ops[0] & ops[1];
        break;
    case OR:
        result = ops[0] | ops[1];
        break;
    case NXOR:
        result = !(ops[0] ^ ops[1]);
        break;
    }
    char output = result + '0';
    return output;
}
```

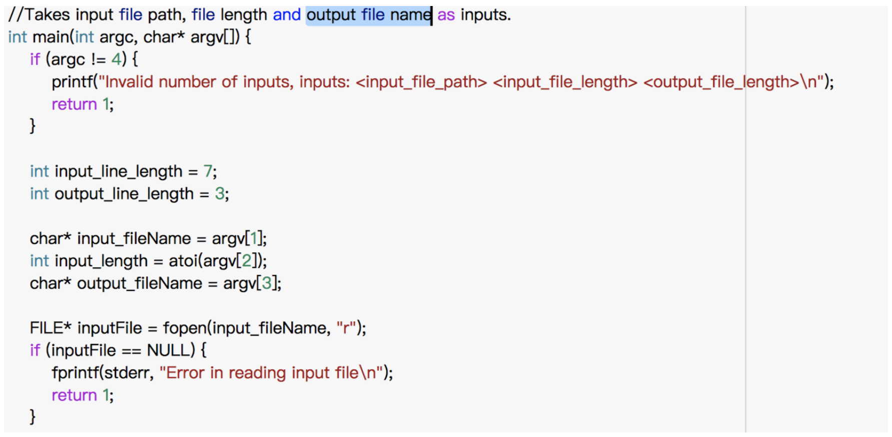
Figure 1: Logic Gate Evaluation

corresponding result by identifying logic gate.

2.3 Parallel

2.3.1 Explicit Memory Allocation

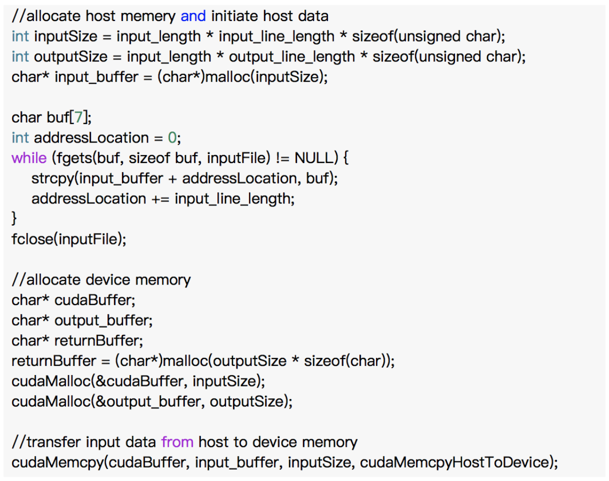
Here is our code snippet of file processing, the code would parse the input into the input file path, file length and output file name. As the format of both input and output file are set, the size for each line in the file is constant for memory allocation.



```
//Takes input file path, file length and output file name as inputs.
int main(int argc, char* argv[]) {
    if (argc != 4) {
        printf("Invalid number of inputs, inputs: <input_file_path> <input_file_length> <output_file_length>\n");
        return 1;
    }

    int input_line_length = 7;
    int output_line_length = 3;

    char* input_fileName = argv[1];
    int input_length = atoi(argv[2]);
    char* output_fileName = argv[3];

    FILE* inputFile = fopen(input_fileName, "r");
    if (inputFile == NULL) {
        fprintf(stderr, "Error in reading input file\n");
        return 1;
    }
```

Figure 2: File Processing

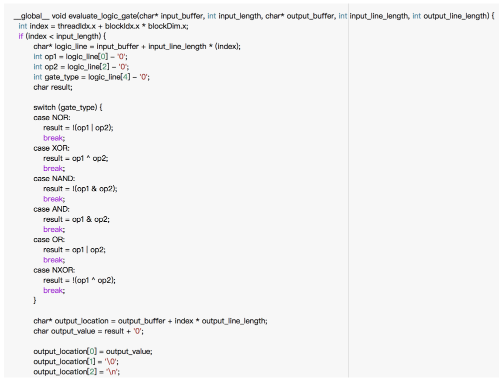
Our code then performs the explicit memory allocation, first allocate host memory, initiate the host data, and allocate device memory. After complete this process, the code would transfer input data from host to device memory.



```
//allocate host memery and initiate host data
int inputSize = input_length * input_line_length * sizeof(unsigned char);
int outputSize = input_length * output_line_length * sizeof(unsigned char);
char* input_buffer = (char*)malloc(inputSize);

char buf[7];
int addressLocation = 0;
while (fgets(buf, sizeof buf, inputFile) != NULL) {
    strcpy(input_buffer + addressLocation, buf);
    addressLocation += input_line_length;
}
fclose(inputFile);

//allocate device memory
char* cudaBuffer;
char* output_buffer;
char* returnBuffer;
returnBuffer = (char*)malloc(outputSize * sizeof(char));
cudaMalloc(&cudaBuffer, inputSize);
cudaMalloc(&output_buffer, outputSize);

//transfer input data from host to device memory
cudaMemcpy(cudaBuffer, input_buffer, inputSize, cudaMemcpyHostToDevice);
```

Figure 3: Memory allocation

After that, the code would call the kernel function to evaluate the logic gates. We define the number of threads to use equal to the number of gates, so each thread take charge of one specific logic gate and the execution of the corresponding input line .The code compute the result boolean value similar to sequential part. We start the clock function just before the execution of kernel function, and end it when kernel function stop.



```
__global__ void evaluate_logic_gate(char* input_buffer, int input_length, char* output_buffer, int input_line_length, int output_line_length) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  if (index < input_length) {
      char* logic_line = input_buffer + input_line_length * (index);
      int op1 = logic_line[0] – '0';
      int op2 = logic_line[2] – '0';
      int gate_type = logic_line[4] – '0';
      char result;

      switch (gate_type) {
      case NOR:
        result = !(op1 | op2);
        break;
      case XOR:
        result = op1 ^ op2;
        break;
      case NAND:
        result = !(op1 & op2);
        break;
      case AND:
        result = op1 & op2;
        break;
      case OR:
        result = op1 | op2;
        break;
      case NXOR:
        result = !(op1 ^ op2);
        break;
      }

      char* output_location = output_buffer + index * output_line_length;
      char output_value = result + '0';

      output_location[0] = output_value;
      output_location[1] = '\0';
      output_location[2] = '\n';
```

Figure 4: Kernel function

Below is the execution time table, as required, we only report the execution time of the computation, and exclude reading and writing times. We can observe that the execution time of explicit memory allocation is far better than using sequential simulation.

Input Size	Execution time(ms)
10000	0.015
100000	0.018
1000000	0.02

Table 3: Execution Time

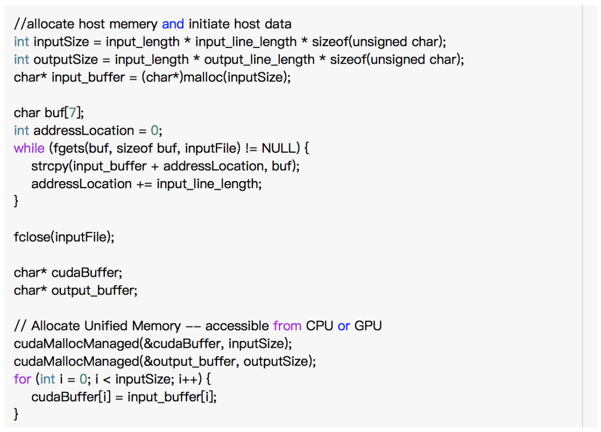
Below is the data migration time table, this is the time consumed to copy data from host to device. We can observe that the data migration time significantly increased as the size of file increased.

Input Size	Data Migration Time(ms)
10000	0.033
100000	0.209
1000000	1.547

Table 4: Execution Time

2.3.2 Unified Memory Allocation

For unified memory allocation, the file processing and logic gates computation function are same to our code for explicit memory allocation. The difference part is the memory allocation.
Here is our code snippet of unified memory allocation. Unified memory is a single memory address space that is accessible for any system processor, it allows applications to allocate data that can be read or written from code running on either CPUs or GPUs. So we do not need the host and device allocation that we implemented in explicit memory allocation. As our code shown, *cudaMallocManaged()* would allocate memory to automatically managed by the Unified Memory system. Unlike the *cudaMalloc()* that used in explicit memory allocation for transportation between host and device.



```
//allocate host memery and initiate host data
int inputSize = input_length * input_line_length * sizeof(unsigned char);
int outputSize = input_length * output_line_length * sizeof(unsigned char);
char* input_buffer = (char*)malloc(inputSize);

char buf[7];
int addressLocation = 0;
while (fgets(buf, sizeof buf, inputFile) != NULL) {
    strcpy(input_buffer + addressLocation, buf);
    addressLocation += input_line_length;
}

fclose(inputFile);

char* cudaBuffer;
char* output_buffer;

// Allocate Unified Memory -- accessible from CPU or GPU
cudaMallocManaged(&cudaBuffer, inputSize);
cudaMallocManaged(&output_buffer, outputSize);
for (int i = 0; i < inputSize; i++) {
    cudaBuffer[i] = input_buffer[i];
}
```

Figure 5: Unified Memory Allocation

Below is the execution time table, as required, we only report the execution time of the computation, and exclude reading and writing times. We can observe that the execution time of explicit memory allocation is better than using sequential simulation, but slower than explicit memory allocation.

Input Size	Execution time(ms)
10000	0.279
100000	0.687
1000000	2.303

Table 5: Execution Time

3 Conclusion

By summarizing all of the data above, we can conclude that, parallelism certainly improve the execution speed compared to sequential execution. For the two parallel solutions, explicit memory allocation and unified memory allocation, explicit parallelized code performed much better. From tutorial we knew that unified memory allocation is more convenient to implement, but less flexible than explicit and have slower execution speed. Our lab result strongly support this statement.

Input Size	Sequential time(ms)	Explicit time(ms)	Unified time(ms)
10000	1.42	0.015	0.279
100000	13.22	0.018	0.687
1000000	137.56	0.02	2.303

Table 6: Execution Time Comparison