ECSE420 Parallel Computing
Lab 1 Logic Gates Simulation
Group 43

Hanwen Wang 260778557
Mai Zeng 260782174

1. Write code that simulates logic gate sequentially:

Code:

The code for sequential part is running on CPU only and because the program is only running on CPU then we just running the program on macOS system terminal, allocating the memory on RAM using malloc.

In the main, we get the input file pointer and output file pointer and then pass these parameters along with the length of input file length (the total number of lines in the input file) to the parseFile function. The reason we multiply the input file length by 6 is because there are 6 characters for each line in the input file. The further details we will talk later.

```
int main(int argc, char* argv[])
{

    if ( argc < 4)
    {
        printf("You must enter 3 input files!\n");
        exit(1);
    }

    // argv[1] : input_file_path
    // argv[2] : input_file_length
    // argv[3] : output_file_path

    char* fileName = argv[1];
    FILE* input = fopen(fileName, "r");
    if (input == NULL)
        exit(EXIT_FAILURE);

    char* outputFileName = argv[3];
    FILE* output = fopen(outputFileName, "w");
    if (output == NULL)
        exit(EXIT_FAILURE);

    parseFile(input, atoi(argv[2])*6, output);

    fclose(input);
    fclose(output);

    return 0;

}
```

Figure 1.1: CPU Code Snippet Main

The code snippet below is how we allocate the memory for writing the input file content to the array in the memory and how we allocate the memory for the output file array. The line 70 to 73 shows that. Note here is that the data and results variable containing the addresses of the first byte of the two arrays (one for input file content and the other is for the output file). These two pointers and the length of the whole input file array will be passing to the processData function and this function is for the simulation of the logical operation. Note here the length of the results array is $\frac{length}{3}$ and the length of the data array is $length$. The reason of that is for each line in the input file contains 6 characters while for each line in the output file contains 2 characters (one if the result of the logical operation and the other is the new line sign '\n'). The start and end variables are for recording the timing.

```c
void parseFile(FILE* fp, int length, FILE* output)
{

    char* data;
    char* results;
    data = (char*)malloc(length);
    results = (char*)malloc(length / 3);
    fread(data, 1, length, fp);
    clock_t start = clock();
    processData(data, length, results);
    clock_t end = clock();
    fputs(results, output);
    // printf("Clocks per second == %f ", CLOCKS_PER_SEC);
    printf("Time used: %f\n", (float)(end-start)/CLOCKS_PER_SEC);
    free(data);
    free(results);
}
```

Figure 1.2: CPU Code Snippet parseFile

The code snippet below shows the processData function. The for loop will iterate every element of the results array for the output file. For each line in the output file there are two characters one is the result of the logical operation and the other is the new line sign ‘\n’ so in each if block we process one line of the input file. We first need to get which logic gate for this line (AND, NAND, OR, NOR, XOR, or XNOR). Each of them is representing by a number and the mapping is shown below given in the lab manual. The fifth element of each line in the input file is the type of logic gate for this operation. We first compare the data[i*3+4] with the number representing each logic gate. The reason we multiply i by 3 is that the length of each input file line is 6 and the length of each output file length is 2 and we are iterate every element of the array for the output file in order to map the input file array’s element index with the output file array’s element index we need to multiply the input file array’s element index by 3.

```c
void processData(char* data, int length, char* results)
{
    for(int i=0; i<length/3; i++)
    {
        // a = data[i];
        // b = data[i + 2];
        // opCode = data[i + 4];
        // in ascii code 1 is 49 and 0 is 48
        if(data[i*3 + 4] == '0')
        {
            int result = (((data[i*3]) - '0') & ((data[i*3 + 2]) - '0'));
            results[i] = (result + '0');
            i++;
            results[i] = '\n';
        }
        else if(data[i*3 + 4] == '1')
        {
            int result = ((data[i*3]) - '0') | ((data[i*3 + 2]) - '0');
            results[i] = (result + '0');
            i++;
            results[i] = '\n';
        }
        else if(data[i*3 + 4] == '2')
        {
            int result = !(((data[i*3]) - '0') & ((data[i*3 + 2]) - '0'));
            results[i] = (result + '0');
            i++;
            results[i] = '\n';
        }
        else if(data[i*3 + 4] == '3')
        {
            int result = !(((data[i*3]) - '0') | ((data[i*3 + 2]) - '0'));
            results[i] = (result + '0');
            i++;
            results[i] = '\n';
        }
        else if(data[i*3 + 4] == '4')
        {
            int result = (((data[i*3]) - '0') ^ ((data[i*3 + 2]) - '0'));
            results[i] = (result + '0');
            i++;
            results[i] = '\n';
        }
        else if(data[i*3 + 4] == '5')
        {
            int result = !(((data[i*3]) - '0') ^ ((data[i*3 + 2]) - '0'));
            results[i] = (result + '0');
            i++;
            results[i] = '\n';
        }
    }
}
```

Figure 1.3 : CPU Code Snippet Simulation

```c
# define AND  0
# define OR   1
# define NAND 2
# define NOR  3
# define XOR  4
# define XNOR 5
```

Figure 1.4 : CPU Code Logical Operations Representation

In figure below we can see the arrangement of the input array and the line in the input file. The last character is for csv file to represent this line is at the end. As shown in the figure, data[i*3+0] and data[i*3+2] are the first and second operands, the data[i*3+1] and data[i*3+3] are the comma for the csv file. The data[i*3+5] is the new line sign.

data[i*3+0]	data[i*3+1]	data[i*3+2]	data[i*3+3]	data[i*3+4]	data[i*3+5]
0	,	1	,	4	'\n'

Figure 1.5: Array Element Mapping With Characters In Each Line Of Input File

The reason in each if block that we need a i++ is that we need to get the index for putting the new line sign in the output file array. Because C is using ascii code and the ‘0’ and ‘1’ are next to each other (‘0’ in the ascii code is 48 and ‘1’ is 49) so we simply need to use the value inside the input array to minus the ‘0’ character to cast the character to integer and then do the logical operation on those to make the simulation. From the official C document, we know that how to do these logical operations in C and it is shown in the table below.

Logical Operation	AND	OR	NAND	NOR	XOR	XNOR
Bitwise Operation In C	&	|	!(op1&op2)	!(op1|op2)	^	!(op1^op2)

Result:

The figure below shows the comparation result between our output files and the solutions given on mycourses. There are no errors which means that our method is right.

```
PS C:\Users\a8785\source\repos\ECSE420_Lab1\Compare> ./compare sol_10000.txt sequential_output_10000.txt
Total Errors : 0
PS C:\Users\a8785\source\repos\ECSE420_Lab1\Compare> ./compare sol_100000.txt sequential_output_100000.txt
Total Errors : 0
PS C:\Users\a8785\source\repos\ECSE420_Lab1\Compare> ./compare sol_1000000.txt sequential_output_1000000.txt
Total Errors : 0
```

Figure 1.6: Results of comparison between explicit outputs and solutions

The figure below shows the time record for the sequential part. Note here we used the built in C library time.h and use the function clock() inside the library. The clock() will give us how many cycles that the CPU clock has run right now so we need to use the number of clock cycles that the CPU has been used after the processData function substract the number of clock cycles that the CPU has been used right before the processData. Here we have a convention issue. Since what we got is the number of clock cycle we need to divide it by the constant CLOCKS_PER_SEC but this constant is different between macOS (Unix), Linux and Windows. For the figure below it is running on macOS and it is representing microsecond.

```
(base) Riverflixs-Mac:Lab1 zengmai-river$ ./sequential Resources/input_10000.csv 10000 output10000.txt
Time used: 0.169000 ms
(base) Riverflixs-Mac:Lab1 zengmai-river$ ./sequential Resources/input_100000.csv 100000 output100000.txt
Time used: 2.381000 ms
(base) Riverflixs-Mac:Lab1 zengmai-river$ ./sequential Resources/input_1000000.csv 1000000 output1000000.txt
Time used: 22.067999 ms
```

Figure 1.7: Host function times on macOS

2. Parallelize your code using explicit memory allocation in CUDA:

Code:

In this question, we are asked to parallelize our code using explicit memory allocation. As a result, we need to do memory allocation in both CPU and GPU. We first use malloc to allocate memory in CPU and cudaMalloc to do that in GPU. Then we use cudaMemcpy to copy memory from CPU to GPU (cudaMemcpyHostToDevice) in order to ensure that their memories are equal. A timer is started before this cudaMemcpy function and stops immediately after memory copy is done. This time period is the explicit data migration time. This part of code is shown below.

```
void parallel_explicit(FILE* fp_in, int length, FILE* fp_out) {
    // input has length 'length' and output has length 'length/3'
    // output file has only 2 elements in one line (a number and a '\n') while input file has 6 (
    char* data, * d_data, * results, * d_results;
    // timer_kernel records time for kernel function
    // timer_migration records explicit data migration time (copy data from host to device)
    GpuTimer timer_kernel, timer_migration;
    data = (char*)malloc(length);
    results = (char*)malloc(length/3);
    cudaMalloc(&d_data, length);
    cudaMalloc(&d_results, length/3);
    fread(data, 1, length, fp_in);
    timer_migration.Start();
    cudaMemcpy(d_data, data, length, cudaMemcpyHostToDevice);
    cudaMemcpy(d_results, results, length/3, cudaMemcpyHostToDevice);
    timer_migration.Stop();
```

Figure 2.1: Explicit Code Snippet

Since the input file has a great number of lines, which is also number of logic gates, and the maximum number of threads per block is 1024, we use the method “parallel blocks and parallel threads” in the kernel function. This is implemented as int i = threadIdx.x + blockIdx.x * blockDim.x. Since our program only needs to launch one thread per logic gate, which is also one thread per line of the input file, this i should be less than the total number of lines in the file to ensure the prerequisite. One line of an input file has 6 characters, the first and third numbers are inputs, the fifth number represents its corresponding gate, the second and fourth numbers are commas while the sixth character is a newline character. Consequently, we first check the fifth number in each line, which should be [i * 6 + 4], after that we check the inputs and conclude a result according to the truth table, finally we store it into the output at position [2 * i] with a newline character at [2 * i + 1]. This part of code is shown below.

```
__global__ void classify(char* d_data, int SIZE, char* d_results) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    // since only needs to launch one thread per logic gate, SIZE should equal to the number of rows in the file
    if (i < SIZE) {
        if (d_data[i * 6 + 4] == '0')
        {
            int result = (((d_data[i * 6]) - '0') & ((d_data[i * 6 + 2]) - '0'));
            d_results[2 * i] = (result + '0');
            d_results[2 * i + 1] = '\n';
        }
        else if (d_data[i * 6 + 4] == '1')
        {
            int result = ((d_data[i * 6]) - '0') | ((d_data[i * 6 + 2]) - '0');
            d_results[2 * i] = (result + '0');
            d_results[2 * i + 1] = '\n';
        }
        else if (d_data[i * 6 + 4] == '2')
        {
            int result = !(((d_data[i * 6]) - '0') & ((d_data[i * 6 + 2]) - '0'));
            d_results[2 * i] = (result + '0');
            d_results[2 * i + 1] = '\n';
        }
        else if (d_data[i * 6 + 4] == '3')
        {
            int result = !(((d_data[i * 6]) - '0') | ((d_data[i * 6 + 2]) - '0'));
            d_results[2 * i] = (result + '0');
            d_results[2 * i + 1] = '\n';
        }
        else if (d_data[i * 6 + 4] == '4')
        {
            int result = (((d_data[i * 6]) - '0') ^ ((d_data[i * 6 + 2]) - '0'));
            d_results[2 * i] = (result + '0');
            d_results[2 * i + 1] = '\n';
        }
        else if (d_data[i * 6 + 4] == '5')
        {
            int result = !(((d_data[i * 6]) - '0') ^ ((d_data[i * 6 + 2]) - '0'));
            d_results[2 * i] = (result + '0');
            d_results[2 * i + 1] = '\n';
        }
    }
}
```

Figure 2.2: Explicit Code Snippet Simulation

Before we call this kernel function on GPU, we need to pass in two parameters, totalBlocks and maxThreadNum. Our idea is that we use the number of lines to divide the maxThreadNum, whose default value is set to 1024, we continue decrease this value by one until this division returns a remainder 0. In this situation, the total number of lines is equally distributed to all the blocks we used. A timer is also set before and after we call the kernel function to record its execution time. After we finish all the execution on GPU, cudaMemcpy is used again but this time it is from GPU to CPU (cudaMemcpyDeviceToHost), its aim is still to ensure the memories on GPU and CPU are equal. At last, we use cudaFree() to free memory on GPU and free() to do that on CPU. This part of code is shown below, it is in the same class and right after the code shown first in this question.

```
    int maxThreadNum = 1024;
    // distribute the total threads equally in blocks
    while(1)
    {
        if(length/6 % maxThreadNum != 0)
        {
            maxThreadNum--;
        }
        else
        {
            break;
        }
    }
    int totalBlocks = length / 6 / maxThreadNum;
    timer_kernel.Start();
    classify <<<totalBlocks, maxThreadNum>>> (d_data, length/6, d_results);
    timer_kernel.Stop();
    cudaMemcpy(data, d_data, length, cudaMemcpyDeviceToHost);
    cudaMemcpy(results, d_results, length/3, cudaMemcpyDeviceToHost);
    printf("Time for kernel functions: %f ms\n", timer_kernel.Elapsed());
    printf("Time for explicit data migration: %f ms\n", timer_migration.Elapsed());

    fputs(results, fp_out);

    cudaFree(d_data);
    cudaFree(d_results);
    free(data);
    free(results);
}
```

Figure 2.3: Explicit Code Snippet Main

Result:

The times for three input files are listed below:

```
For input file ./input_10000.csv
Time for kernel functions: 0.016384 ms
Time for explicit data migration: 0.193216 ms
```

Figure 2.4: Necessary times for input file input_10000.csv

```
For input file ./input_100000.csv
Time for kernel functions: 0.057856 ms
Time for explicit data migration: 0.255104 ms
```

Figure 2.5: Necessary times for input file input_100000.csv

```
For input file ./input_1000000.csv
Time for kernel functions: 0.448512 ms
Time for explicit data migration: 1.378400 ms
```

Figure 2.6: Necessary times for input file input_1000000.csv

The compare results between three output files and solutions are listed below:

```
PS C:\Users\a8785\source\repos\ECSE420_Lab1\Compare> ./compare sol_10000.txt explicit_output_10000.txt
Total Errors : 0
PS C:\Users\a8785\source\repos\ECSE420_Lab1\Compare> ./compare sol_100000.txt explicit_output_100000.txt
Total Errors : 0
PS C:\Users\a8785\source\repos\ECSE420_Lab1\Compare> ./compare sol_1000000.txt explicit_output_1000000.txt
Total Errors : 0
```

Figure 2.7: Results of comparison between explicit outputs and solutions

This means that our function is correct.

3. Parallelize your code using unified memory allocation in CUDA:

Code:

In this question, we are asked to parallelize the code using unified memory allocation. In a unified memory, if we understand the concept in a simple way, that is both CPU and GPU share the same memory space. As a result, we do not need to use two different methods to malloc space, keep updating them by copying memory and use two different ways to free them at last, instead we use the method cudaMallocManaged to malloc the space once and cudaDeviceSynchronize() to force the program to ensure the streams' kernels are complete before continuing, we only use cudaFree() once at last. Apart from this change, the remaining logics are the same as question 2. There is no data migration time because this is not needed in unified memory allocation. The part of code that is changed is shown below.

```
void parallel_unified(FILE* fp_in, int length, FILE* fp_out) {
    // input has length 'length' and output has length 'length/3'
    // output file has only 2 elements in one line (a number and a '\n') while input file has 6 (listed in main)
    char* data, * results;
    // timer_kernel records time for kernel function
    GpuTimer timer_kernel;
    // Unified memory allocation methods
    cudaMallocManaged(&data, length);
    cudaMallocManaged(&results, length / 3);
    fread(data, 1, length, fp_in);
    int maxThreadNum = 1024;
    // distribute the total threads equally in blocks
    while (1)
    {
        if (length / 6 % maxThreadNum != 0)
        {
            maxThreadNum--;
        }
        else
        {
            break;
        }
    }
    int totalBlocks = length / 6 / maxThreadNum;
    timer_kernel.Start();
    classify <<<totalBlocks, maxThreadNum >>> (data, length / 6, results);
    timer_kernel.Stop();
    cudaDeviceSynchronize();
    printf("Time for kernel functions: %f ms\n", timer_kernel.Elapsed()); // Convert unit from ms to s

    fputs(results, fp_out);

    cudaFree(data);
    cudaFree(results);
}
```

Figure 3.1: Unified Code Snippet

Result:

The times for three input files are listed below:

```
For input file ./input_10000.csv
Time for kernel functions: 0.017184 ms
```

Figure 3.2: Kernel function time for input file input_10000.csv

```
For input file ./input_100000.csv
Time for kernel functions: 0.059744 ms
```

Figure 3.3: Kernel function time for input file input_100000.csv

```
For input file ./input_1000000.csv
Time for kernel functions: 0.449312 ms
```

Figure 3.4: Kernel function time for input file input_1000000.csv

The compare results between three output files and solutions are listed below:

```
PS C:\Users\a8785\source\repos\ECSE420_Lab1\Compare> ./compare sol_10000.txt unified_output_10000.txt
Total Errors : 0
PS C:\Users\a8785\source\repos\ECSE420_Lab1\Compare> ./compare sol_100000.txt unified_output_100000.txt
Total Errors : 0
PS C:\Users\a8785\source\repos\ECSE420_Lab1\Compare> ./compare sol_1000000.txt unified_output_1000000.txt
Total Errors : 0
```

Figure 3.5: Results of comparison between unified outputs and solutions

This means that our function is correct. For all the kernel functions' execution times, they are a little bit larger (around 0.1 ms to 0.2 ms) in unified memory compared to those in explicit memory.

As we can see in the timing records, the unified memory is taking slightly longer than explicit parallel method running the kernel function. More details will be talked in the fifth section.

4. Parallel code using unified memory allocation with data prefetching in CUDA:

Since the CUDA windows version is not supporting the prefetching function we are going to use colab for this part.
The code implementation is simple. We just need to add the code snippet right before we call the kernel function like the Figure 4.1 below.

```
// Prefetch the data to the GPU
int device = -1;
cudaGetDevice(&device);
cudaMemPrefetchAsync(data, length, device, NULL);
cudaMemPrefetchAsync(results, length/3, device, NULL);

timer_kernel.Start();
classify <<<totalBlocks, maxThreadNum >>> (data, length / 6, results);
timer_kernel.Stop();
cudaDeviceSynchronize();
printf("Time for kernel functions: %f ms\n", timer_kernel.Elapsed());
```

Figure 4.1: Code Snippet For Unified Memory With Prefetching

In this way we can first prefetching the memory in Unified memory for GPU to avoid the page fault for GPU memory which will be talked in the latter section.

The result for prefetching is shown below. Note here that since it is running on Google Colab so the time may not be comparable with the previous part. More details will be talked in the latter section.

```
[24] !nvcc parallel_prefetch.cu -o parallel_prefetch
     !./parallel_prefetch input_10000.csv 10000 output_parallel_prefetch_10000.txt
     !./parallel_prefetch input_100000.csv 100000 output_parallel_prefetch_100000.txt
     !./parallel_prefetch input_1000000.csv 1000000 output_parallel_prefetch_1000000.txt

     For input file input_10000.csv
     Time for kernel functions: 0.011680 ms
     For input file input_100000.csv
     Time for kernel functions: 0.022592 ms
     For input file input_1000000.csv
     Time for kernel functions: 0.124448 ms
```

Figure 4.2: Time Recording For Unified Memory With Prefetching

Comparation For Unified Memory With Prefetching

```
!gcc compareResults.c -o compareResults
!./compareResults output_parallel_prefetch_10000.txt sol_10000.txt
!./compareResults output_parallel_prefetch_100000.txt sol_100000.txt
!./compareResults output_parallel_prefetch_1000000.txt sol_1000000.txt

Total Errors : 0        Total Errors : 0        Total Errors : 0
```

Figure 4.3: Comparation Between Ouput Files and Solutions

5. Discuss the results for each architecture:

We did the experiments using the CPU code with sequential method, CUDA code with explicit memory allocation method (Ordinary CUDA code), and CUDA code with unified memory.
The results are shown below:

Foe explicit parallel:

```
For input file ./input_10000.csv
Time for kernel functions: 0.016384 ms
Time for explicit data migration: 0.193216 ms
For input file ./input_100000.csv
Time for kernel functions: 0.057856 ms
Time for explicit data migration: 0.255104 ms
For input file ./input_1000000.csv
Time for kernel functions: 0.448512 ms
Time for explicit data migration: 1.378400 ms
```

Figure 5.1: Time Recording For Executing csv Files For Explicit Parallel

For unified parallel:

```
For input file ./input_10000.csv
Time for kernel functions: 0.017184 ms
For input file ./input_100000.csv
Time for kernel functions: 0.059744 ms
For input file ./input_1000000.csv
Time for kernel functions: 0.449312 ms
```

Figure 5.2: Time Recording For Executing csv Files For Unified Parallel

For sequential part:

```
(base) Riverflixs-Mac:Lab1 zengmai-river$ ./sequential Resources/input_10000.csv 10000 output10000.txt
Time used: 0.169000 ms
(base) Riverflixs-Mac:Lab1 zengmai-river$ ./sequential Resources/input_100000.csv 100000 output100000.txt
Time used: 2.381000 ms
(base) Riverflixs-Mac:Lab1 zengmai-river$ ./sequential Resources/input_1000000.csv 1000000 output1000000.txt
Time used: 22.067999 ms
```

Figure 5.3: Time Recording For Executing csv Files For CPU Code

As we can see from the timing records. The code running only on CPU is much slower than the CUDA code with either the methods. Comparing the two CUDA codes, with the unified parallel method the time for running the kernel functions is a little bit longer but almost the same as the time running the kernel functions with explicit parallel method for both 3 files but note here the parallel explicit method need the data migration time.

The explicit memory architecture can be represented like the figure below:

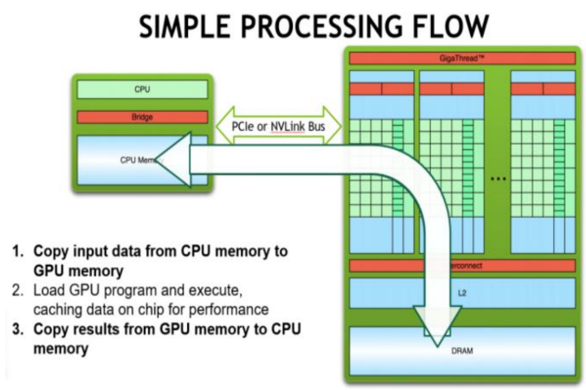

Figure 5.4: Explicit memory architecture

The data first need to copy from CPU memory to GPU memory and that is why data migration taking a significant amount of time. After the data migration, it will load the GPU program and then execute.
The unified memory architecture can be represented like the figure below:

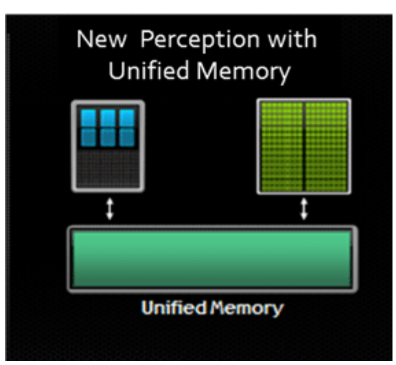

Figure 5.5: Unified memory architecture

As shown in the Figure 5.5, unified Memory is a single memory address space accessible from any processor in a system. This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs. When code running on a CPU or GPU accesses data allocated this way (often called CUDA managed data), the CUDA system software and/or the hardware takes care of migrating memory pages to the memory of the accessing processor. [1]

CUDA 6 adds one extra layer of convenience to the CPU/GPU memory management task with the introduction of unified or managed memory. Data is now stored and migrated in a user-transparent fashion that enables, under circumstances spelled out shortly, data access/transfer at latencies and bandwidths of the host and of the device, for host-side and device-side memory operations, respectively. Moreover, the use of the cudaHostAlloc and cudaMemcpy combination is no longer a requirement, which allows for a cleaner and more natural programming style.

This means that after CUDA6 the developer does not need to concern the cudaMemcpy and cudaMalloc instead they can just use cudaMallocManaged function. This eliminate the need for explicit copy and it still allows explicit hand tuning.

The reason that the unified memory is a little bit slower than the explicit is that the memory is unified but it still need to communicate between the GPU memory with the unified memory and that will take some time but with the unified memory, it does not need the data migration time which is actually a significant amount of time.

For better understand the difference between explicit parallel, unified parallel and prefetch with unified memory. We tested our code on the Google Colab and run the command ‘nvprof’ to see the whole running profile.

For explicit parallel without using unified memory:

```
!nvprof ./parallel_explicit input_10000.csv 10000 output_parallel_explicit_10000.txt
```

```
For input file input_10000.csv
==509== NVPROF is profiling process 509, command: ./parallel_explicit input_10000.csv 10000 output_parallel_explicit_10000.txt
Time for kernel functions: 0.027392 ms
Time for explicit data migration: 0.112224 ms
==509== Profiling application: ./parallel_explicit input_10000.csv 10000 output_parallel_explicit_10000.txt
==509== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   47.38%  11.840us         2  5.9200us  3.5840us  8.2560us  [CUDA memcpy HtoD]
                   31.63%  7.9040us         2  3.9520us  2.4000us  5.5040us  [CUDA memcpy DtoH]
                   21.00%  5.2480us         1  5.2480us  5.2480us  5.2480us  classify(char*, int, char*)
      API calls:   99.42%  167.03ms         6  27.838ms     565ns  167.02ms  cudaEventCreate
                    0.22%  368.95us         1  368.95us  368.95us  368.95us  cuDeviceTotalMem
                    0.10%  170.71us         2  85.353us  6.7840us  163.92us  cudaMalloc
                    0.08%  137.51us        97  1.4170us     134ns  52.172us  cuDeviceGetAttribute
                    0.06%  106.27us         4  26.568us  18.283us  32.950us  cudaMemcpy
                    0.06%  102.40us         2  51.198us  15.017us  87.379us  cudaFree
                    0.02%  25.548us         1  25.548us  25.548us  25.548us  cudaLaunchKernel
                    0.01%  22.767us         1  22.767us  22.767us  22.767us  cuDeviceGetName
                    0.01%  14.466us         6  2.4110us  1.2370us  6.1700us  cudaEventRecord
                    0.00%  8.2990us         3  2.7660us  1.7550us  3.3620us  cudaEventElapsedTime
                    0.00%  7.3050us         3  2.4350us  2.1230us  2.6880us  cudaEventSynchronize
                    0.00%  4.7900us         6     798ns     427ns  1.8140us  cudaEventDestroy
                    0.00%  3.8250us         1  3.8250us  3.8250us  3.8250us  cuDeviceGetPCIBusId
                    0.00%  2.1390us         3     713ns     140ns  1.2460us  cuDeviceGetCount
                    0.00%  1.5590us         2     779ns     393ns  1.1660us  cuDeviceGet
                    0.00%     250ns         1     250ns     250ns     250ns  cuDeviceGetUuid
```

Figure 5.6: Explicit 10000

```
!nvprof ./parallel_explicit input_100000.csv 100000 output_parallel_explicit_100000.txt

For input file input_100000.csv
==524== NVPROF is profiling process 524, command: ./parallel_explicit input_100000.csv 100000 output_parallel_explicit_100000.txt
Time for kernel functions: 0.021568 ms
Time for explicit data migration: 0.559360 ms
==524== Profiling application: ./parallel_explicit input_100000.csv 100000 output_parallel_explicit_100000.txt
==524== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   48.13%  70.655us         2  35.327us  18.688us  51.967us  [CUDA memcpy HtoD]
                   45.79%  67.231us         2  33.615us  16.384us  50.847us  [CUDA memcpy DtoH]
                    6.08%  8.9280us         1  8.9280us  8.9280us  8.9280us  classify(char*, int, char*)
      API calls:   99.12%  156.56ms         6  26.094ms     952ns  156.56ms  cudaEventCreate
                    0.34%  544.91us         4  136.23us  104.78us  184.16us  cudaMemcpy
                    0.25%  392.76us         1  392.76us  392.76us  392.76us  cuDeviceTotalMem
                    0.08%  130.25us         2  65.122us  7.1480us  123.10us  cudaMalloc
                    0.08%  130.00us        97  1.3400us     136ns  48.297us  cuDeviceGetAttribute
                    0.07%  105.88us         2  52.942us  16.223us  89.661us  cudaFree
                    0.02%  27.656us         1  27.656us  27.656us  27.656us  cudaLaunchKernel
                    0.01%  16.805us         6  2.8000us  1.2240us  8.0460us  cudaEventRecord
                    0.01%  14.874us         1  14.874us  14.874us  14.874us  cuDeviceGetName
                    0.01%  8.8720us         3  2.9570us  1.8220us  3.6410us  cudaEventElapsedTime
                    0.00%  7.7440us         3  2.5810us  2.1890us  2.8550us  cudaEventSynchronize
                    0.00%  6.1340us         6  1.0220us     453ns  2.9440us  cudaEventDestroy
                    0.00%  3.2270us         1  3.2270us  3.2270us  3.2270us  cuDeviceGetPCIBusId
                    0.00%  2.3370us         3     779ns     167ns  1.6030us  cuDeviceGetCount
                    0.00%  1.1160us         2     558ns     265ns     851ns  cuDeviceGet
                    0.00%     266ns         1     266ns     266ns     266ns  cuDeviceGetUuid
```

Figure 5.7: Explicit 100000

```
!nvprof ./parallel_explicit input_1000000.csv 1000000 output_parallel_explicit_1000000.txt

For input file input_1000000.csv
==537== NVPROF is profiling process 537, command: ./parallel_explicit input_1000000.csv 1000000 output_parallel_explicit_1000000.txt
Time for kernel functions: 0.073632 ms
Time for explicit data migration: 4.961696 ms
==537== Profiling application: ./parallel_explicit input_1000000.csv 1000000 output_parallel_explicit_1000000.txt
==537== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   61.31%  1.5204ms         2  760.20us  398.27us  1.1221ms  [CUDA memcpy HtoD]
                   36.06%  894.13us         2  447.06us  155.17us  738.96us  [CUDA memcpy DtoH]
                    2.63%  65.215us         1  65.215us  65.215us  65.215us  classify(char*, int, char*)
      API calls:   96.38%  166.19ms         6  27.698ms     542ns  166.18ms  cudaEventCreate
                    2.89%  4.9862ms         4  1.2465ms  755.75us  1.8855ms  cudaMemcpy
                    0.24%  407.62us         1  407.62us  407.62us  407.62us  cuDeviceTotalMem
                    0.22%  378.29us         2  189.15us  115.91us  262.39us  cudaFree
                    0.14%  233.65us         2  116.82us  108.19us  125.46us  cudaMalloc
                    0.08%  133.21us        97  1.3730us     142ns  49.563us  cuDeviceGetAttribute
                    0.02%  30.196us         1  30.196us  30.196us  30.196us  cudaLaunchKernel
                    0.02%  27.501us         6  4.5830us  1.2220us  12.873us  cudaEventRecord
                    0.01%  17.041us         1  17.041us  17.041us  17.041us  cuDeviceGetName
                    0.00%  8.2150us         3  2.7380us  1.6510us  3.3190us  cudaEventElapsedTime
                    0.00%  8.1830us         3  2.7270us  2.2850us  3.0850us  cudaEventSynchronize
                    0.00%  6.5630us         6  1.0930us     464ns  3.3840us  cudaEventDestroy
                    0.00%  2.9120us         1  2.9120us  2.9120us  2.9120us  cuDeviceGetPCIBusId
                    0.00%  1.9810us         3     660ns     183ns  1.1260us  cuDeviceGetCount
                    0.00%  1.2200us         2     610ns     360ns     860ns  cuDeviceGet
                    0.00%     303ns         1     303ns     303ns     303ns  cuDeviceGetUuid
```

Figure 5.8: Explicit 1000000

For Unified memory without prefetching:

```
!nvprof ./parallel_unified input_10000.csv 10000 output_parallel_unified_10000.txt

For input file input_10000.csv
==790== NVPROF is profiling process 790, command: ./parallel_unified input_10000.csv 10000 output_parallel_unified_10000.txt
Time for kernel functions: 0.605248 ms
==790== Profiling application: ./parallel_unified input_10000.csv 10000 output_parallel_unified_10000.txt
==790== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  556.76us         1  556.76us  556.76us  556.76us  classify(char*, int, char*)
      API calls:   88.08%  160.93ms         2  80.466ms     948ns  160.93ms  cudaEventCreate
                   11.17%  20.407ms         2  10.204ms  34.904us  20.373ms  cudaMallocManaged
                    0.31%  565.49us         1  565.49us  565.49us  565.49us  cudaDeviceSynchronize
                    0.24%  430.73us         1  430.73us  430.73us  430.73us  cuDeviceTotalMem
                    0.08%  146.94us        97  1.5140us     135ns  47.650us  cuDeviceGetAttribute
                    0.07%  134.20us         2  67.098us  28.889us  105.31us  cudaFree
                    0.03%  45.986us         1  45.986us  45.986us  45.986us  cudaLaunchKernel
                    0.01%  19.309us         2  9.6540us  3.0040us  16.305us  cudaEventRecord
                    0.01%  13.381us         1  13.381us  13.381us  13.381us  cuDeviceGetName
                    0.00%  3.5650us         1  3.5650us  3.5650us  3.5650us  cudaEventSynchronize
                    0.00%  3.3330us         2  1.6660us     610ns  2.7230us  cudaEventDestroy
                    0.00%  3.1900us         1  3.1900us  3.1900us  3.1900us  cuDeviceGetPCIBusId
                    0.00%  3.0890us         1  3.0890us  3.0890us  3.0890us  cudaEventElapsedTime
                    0.00%  1.7780us         3     592ns     163ns  1.1050us  cuDeviceGetCount
                    0.00%  1.1700us         2     585ns     240ns     930ns  cuDeviceGet
                    0.00%     274ns         1     274ns     274ns     274ns  cuDeviceGetUuid

==790== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       2  32.000KB  4.0000KB  60.000KB  64.00000KB  11.16800us  Host To Device
       5  25.600KB  4.0000KB  60.000KB  128.0000KB  22.78400us  Device To Host
       2         -         -         -           -  549.0880us  Gpu page fault groups
Total CPU Page faults: 3
```

Figure 5.9: Unified without prefetching 10000

```
!nvprof ./parallel_unified input_100000.csv 100000 output_parallel_unified_100000.txt
```

```
For input file input_100000.csv
==777== NVPROF is profiling process 777, command: ./parallel_unified input_100000.csv 100000 output_parallel_unified_100000.txt
Time for kernel functions: 0.676928 ms
==777== Profiling application: ./parallel_unified input_100000.csv 100000 output_parallel_unified_100000.txt
==777== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  637.56us         1  637.56us  637.56us  637.56us  classify(char*, int, char*)
      API calls:   88.38%  165.26ms         2  82.630ms  1.0250us  165.26ms  cudaEventCreate
                   10.85%  20.297ms         2  10.148ms  25.071us  20.272ms  cudaMallocManaged
                    0.30%  568.94us         1  568.94us  568.94us  568.94us  cudaDeviceSynchronize
                    0.24%  445.93us         1  445.93us  445.93us  445.93us  cuDeviceTotalMem
                    0.11%  200.83us         2  100.42us  69.844us  130.99us  cudaFree
                    0.07%  132.86us        97  1.3690us     148ns  48.198us  cuDeviceGetAttribute
                    0.02%  33.550us         1  33.550us  33.550us  33.550us  cudaLaunchKernel
                    0.01%  16.952us         2  8.4760us  4.6210us  12.331us  cudaEventRecord
                    0.01%  14.174us         1  14.174us  14.174us  14.174us  cuDeviceGetName
                    0.00%  3.9760us         1  3.9760us  3.9760us  3.9760us  cudaEventSynchronize
                    0.00%  3.6880us         2  1.8440us     651ns  3.0370us  cudaEventDestroy
                    0.00%  3.5430us         1  3.5430us  3.5430us  3.5430us  cudaEventElapsedTime
                    0.00%  2.6510us         1  2.6510us  2.6510us  2.6510us  cuDeviceGetPCIBusId
                    0.00%  1.8170us         3     605ns     197ns  1.2240us  cuDeviceGetCount
                    0.00%  1.0930us         2     546ns     244ns     849ns  cuDeviceGet
                    0.00%     280ns         1     280ns     280ns     280ns  cuDeviceGetUuid

==777== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      38  26.947KB  4.0000KB  244.00KB  1.000000MB  144.0000us  Host To Device
      10  51.199KB  4.0000KB  252.00KB  512.0000KB  49.63200us  Device To Host
       2         -         -         -           -  620.3840us  Gpu page fault groups
Total CPU Page faults: 9
```

Figure 5.10: Unified without prefetching 100000

```
!nvprof ./parallel_unified input_1000000.csv 1000000 output_parallel_unified_1000000.txt
```

```
For input file input_1000000.csv
==801== NVPROF is profiling process 801, command: ./parallel_unified input_1000000.csv 1000000 output_parallel_unified_1000000.txt
Time for kernel functions: 2.741632 ms
==801== Profiling application: ./parallel_unified input_1000000.csv 1000000 output_parallel_unified_1000000.txt
==801== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  2.6897ms         1  2.6897ms  2.6897ms  2.6897ms  classify(char*, int, char*)
      API calls:   87.52%  170.18ms         2  85.089ms     977ns  170.18ms  cudaEventCreate
                   10.52%  20.459ms         2  10.230ms  43.812us  20.416ms  cudaMallocManaged
                    1.39%  2.7029ms         1  2.7029ms  2.7029ms  2.7029ms  cudaDeviceSynchronize
                    0.22%  434.79us         2  217.39us  121.24us  313.55us  cudaFree
                    0.21%  415.24us         1  415.24us  415.24us  415.24us  cuDeviceTotalMem
                    0.07%  142.17us        97  1.4650us     158ns  54.434us  cuDeviceGetAttribute
                    0.03%  50.315us         1  50.315us  50.315us  50.315us  cudaLaunchKernel
                    0.01%  17.454us         1  17.454us  17.454us  17.454us  cuDeviceGetName
                    0.01%  15.674us         2  7.8370us  3.1340us  12.540us  cudaEventRecord
                    0.00%  3.9970us         1  3.9970us  3.9970us  3.9970us  cudaEventSynchronize
                    0.00%  3.6670us         2  1.8330us     642ns  3.0250us  cudaEventDestroy
                    0.00%  3.2710us         1  3.2710us  3.2710us  3.2710us  cuDeviceGetPCIBusId
                    0.00%  3.1350us         1  3.1350us  3.1350us  3.1350us  cudaEventElapsedTime
                    0.00%  2.2370us         3     745ns     200ns  1.4080us  cuDeviceGetCount
                    0.00%  1.2720us         2     636ns     274ns     998ns  cuDeviceGet
                    0.00%     317ns         1     317ns     317ns     317ns  cuDeviceGetUuid

==801== Unified Memory profiling result:
Device "Tesla P100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
     239  24.519KB  4.0000KB  724.00KB  5.722656MB  813.1840us  Host To Device
      12  170.67KB  4.0000KB  0.9961MB  2.000000MB  173.6640us  Device To Host
      16         -         -         -           -  2.901920ms  Gpu page fault groups
Total CPU Page faults: 24
```

Figure 5.11: Unified without prefetching 1000000

For Unified Memory with Prefetching:

```
!nvprof ./parallel_prefetch input_10000.csv 10000 output_parallel_prefetch_10000.txt
```

```
For input file input_10000.csv
==361== NVPROF is profiling process 361, command: ./parallel_prefetch input_10000.csv 10000 output_parallel_prefetch_10000.txt
Time for kernel functions: 0.042752 ms
==361== Profiling application: ./parallel_prefetch input_10000.csv 10000 output_parallel_prefetch_10000.txt
==361== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  6.6870us         1  6.6870us  6.6870us  6.6870us  classify(char*, int, char*)
      API calls:   88.58%  167.88ms         2  83.938ms     960ns  167.88ms  cudaEventCreate
                   10.91%  20.669ms         2  10.335ms  34.143us  20.635ms  cudaMallocManaged
                    0.19%  362.81us         1  362.81us  362.81us  362.81us  cuDeviceTotalMem
                    0.13%  252.41us         2  126.20us  21.344us  231.06us  cudaMemPrefetchAsync
                    0.07%  138.20us        97  1.4240us     133ns  58.893us  cuDeviceGetAttribute
                    0.06%  109.14us         2  54.572us  29.121us  80.023us  cudaFree
                    0.02%  35.367us         1  35.367us  35.367us  35.367us  cudaLaunchKernel
                    0.01%  27.919us         1  27.919us  27.919us  27.919us  cuDeviceGetName
                    0.01%  14.578us         2  7.2890us  2.8190us  11.759us  cudaEventRecord
                    0.01%  10.446us         1  10.446us  10.446us  10.446us  cudaDeviceSynchronize
                    0.00%  3.2220us         1  3.2220us  3.2220us  3.2220us  cuDeviceGetPCIBusId
                    0.00%  3.1670us         2  1.5830us     740ns  2.4270us  cudaEventDestroy
                    0.00%  2.5730us         1  2.5730us  2.5730us  2.5730us  cudaGetDevice
                    0.00%  2.0400us         1  2.0400us  2.0400us  2.0400us  cudaEventSynchronize
                    0.00%  1.7500us         3     583ns     139ns  1.1810us  cuDeviceGetCount
                    0.00%  1.7030us         1  1.7030us  1.7030us  1.7030us  cudaEventElapsedTime
                    0.00%  1.0620us         2     531ns     305ns     757ns  cuDeviceGet
                    0.00%     272ns         1     272ns     272ns     272ns  cuDeviceGetUuid

==361== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       2  32.000KB  4.0000KB  60.000KB  64.00000KB  12.76800us  Host To Device
       5  16.000KB  4.0000KB  56.000KB  80.00000KB  13.66400us  Device To Host
Total CPU Page faults: 3
```

Figure 5.12: Unified without prefetching 10000

```
!nvprof ./parallel_prefetch input_100000.csv 100000 output_parallel_prefetch_100000.txt
```

```
For input file input_100000.csv
==372== NVPROF is profiling process 372, command: ./parallel_prefetch input_100000.csv 100000 output_parallel_prefetch_100000.txt
Time for kernel functions: 0.022816 ms
==372== Profiling application: ./parallel_prefetch input_100000.csv 100000 output_parallel_prefetch_100000.txt
==372== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  17.151us         1  17.151us  17.151us  17.151us  classify(char*, int, char*)
      API calls:   88.83%  172.28ms         2  86.140ms  1.0980us  172.28ms  cudaEventCreate
                   10.54%  20.448ms         2  10.224ms  24.223us  20.424ms  cudaMallocManaged
                    0.21%  401.05us         2  200.52us  151.72us  249.33us  cudaMemPrefetchAsync
                    0.18%  339.46us         1  339.46us  339.46us  339.46us  cuDeviceTotalMem
                    0.08%  146.59us         2  73.296us  33.429us  113.16us  cudaFree
                    0.08%  146.48us        97  1.5100us     151ns  62.122us  cuDeviceGetAttribute
                    0.04%  71.964us         1  71.964us  71.964us  71.964us  cudaDeviceSynchronize
                    0.02%  44.569us         1  44.569us  44.569us  44.569us  cudaLaunchKernel
                    0.01%  27.757us         1  27.757us  27.757us  27.757us  cuDeviceGetName
                    0.01%  18.041us         2  9.0200us  6.1710us  11.870us  cudaEventRecord
                    0.00%  4.3550us         1  4.3550us  4.3550us  4.3550us  cuDeviceGetPCIBusId
                    0.00%  3.9640us         2  1.9820us     905ns  3.0590us  cudaEventDestroy
                    0.00%  3.8310us         1  3.8310us  3.8310us  3.8310us  cudaGetDevice
                    0.00%  2.7900us         1  2.7900us  2.7900us  2.7900us  cudaEventSynchronize
                    0.00%  2.1410us         1  2.1410us  2.1410us  2.1410us  cudaEventElapsedTime
                    0.00%  2.0810us         3     693ns     175ns  1.3140us  cuDeviceGetCount
                    0.00%  1.3430us         2     671ns     268ns  1.0750us  cuDeviceGet
                    0.00%     266ns         1     266ns     266ns     266ns  cuDeviceGetUuid

==372== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       2  392.00KB  196.00KB  588.00KB  784.0000KB  73.95200us  Host To Device
       3  90.666KB  4.0000KB  196.00KB  272.0000KB  25.63200us  Device To Host
Total CPU Page faults: 6
```

Figure 5.13: Unified without prefetching 100000

```
!nvprof ./parallel_prefetch input_1000000.csv 1000000 output_parallel_prefetch_1000000.txt
```

```
For input file input_1000000.csv
==384== NVPROF is profiling process 384, command: ./parallel_prefetch input_1000000.csv 1000000 output_parallel_prefetch_1000000.txt
Time for kernel functions: 0.170720 ms
==384== Profiling application: ./parallel_prefetch input_1000000.csv 1000000 output_parallel_prefetch_1000000.txt
==384== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  129.43us         1  129.43us  129.43us  129.43us  classify(char*, int, char*)
      API calls:   88.01%  166.18ms         2  83.092ms     962ns  166.18ms  cudaEventCreate
                   10.79%  20.369ms         2  10.184ms  52.491us  20.316ms  cudaMallocManaged
                    0.45%  857.49us         2  428.74us  25.128us  832.36us  cudaMemPrefetchAsync
                    0.33%  621.66us         2  310.83us  166.56us  455.11us  cudaFree
                    0.19%  361.84us         1  361.84us  361.84us  361.84us  cuDeviceTotalMem
                    0.09%  171.02us        97  1.7630us     141ns  84.938us  cuDeviceGetAttribute
                    0.07%  131.46us         1  131.46us  131.46us  131.46us  cudaDeviceSynchronize
                    0.02%  46.970us         1  46.970us  46.970us  46.970us  cuDeviceGetName
                    0.02%  36.938us         1  36.938us  36.938us  36.938us  cudaLaunchKernel
                    0.01%  13.962us         2  6.9810us  4.1210us  9.8410us  cudaEventRecord
                    0.00%  4.5980us         1  4.5980us  4.5980us  4.5980us  cudaGetDevice
                    0.00%  3.9000us         2  1.9500us     619ns  3.2810us  cudaEventDestroy
                    0.00%  3.6390us         1  3.6390us  3.6390us  3.6390us  cuDeviceGetPCIBusId
                    0.00%  2.1940us         3     731ns     167ns  1.5810us  cuDeviceGetCount
                    0.00%  2.1750us         1  2.1750us  2.1750us  2.1750us  cudaEventSynchronize
                    0.00%  1.7600us         1  1.7600us  1.7600us  1.7600us  cudaEventElapsedTime
                    0.00%  1.0430us         2     521ns     290ns     753ns  cuDeviceGet
                    0.00%     287ns         1     287ns     287ns     287ns  cuDeviceGetUuid

==384== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       3  1.9076MB  1.7227MB  2.0000MB  5.722656MB  501.9200us  Host To Device
      12  163.00KB  4.0000KB  928.00KB  1.910156MB  173.5040us  Device To Host
Total CPU Page faults: 24
```

Figure 5.14: Unified without prefetching 1000000

From Figure 5.9 to 5.11 and Figure 5.12 to 5.14, we can see that for the unified memory without the prefetching there will be host-to-device page fault generated for GPU but with the unified memory with prefetching there won't be any page fault and they both just call kernel once unlike the explicit CUDA code which will call kernel function several times. This actually makes senses since if we just use unified memory, managed memory may not be physically allocated when cudaMallocManaged() returns; it may only be populated on access (or prefetching) the memory entries may not be created until they are accessed by GPU or CPU and this may create page fault. After prefetching we can move the data to the GPU after initializing it and this will prevent the page fault.

Reference:

[1] M. Harris. "Unified Memory for CUDA Beginners". NVDIA Developer Blog. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/ (accessed Oct. 18, 2020).
[2] D. Negrut, R. Serban, A. Li, A. Seidl. "Unified Memory in CUDA 6: A Brief Overview". Dr.Dobb's. https://www.drdobbs.com/architecture-and-design/unified-memory-in-cuda-6-a-brief-overvie/240169095?pgno=2 (accessed Oct. 18, 2020)