ECSE 420 – Parallel Computing – Fall 2020

Lab 1 – Logic Gates Simulation

Aleksas Murauskas 260718389

Anne-Julie Côté 260803768

Group 16

Parts 1-4 describe the work we did to execute the different logic gate simulations. We then discussed the results (measured times) and compared the methods in part 5.

1. Sequential logic gate simulation

The following portion will describe the sequential portion of our code, held within the sequential.c file.

Within the sequential file, the processing of all the operation gates will be done with no parallel computing architecture. The input output file names, as well as the length of the input are taken in as arguments and stored in memory. Then the input file is opened with read only permissions, and the program terminates if the file does not exist under the given filename. Three arrays are formed: bool1 and bool2, which will hold the boolean operands of the gate, and op, which will hold the operator of the gate. The same amount of memory is allocated for the arrays, the size of an integer multiplied by the number of lines in the input denoted by the third argument passed. The transfer of the data to from the file to the three arrays is accomplished in a for loop. The loop iterates through the number of lines of input, and for each line, a char pointer line is updated with the current line of data from the file with the method getline(). The data held within the line pointer is broken up into operands 1, 2 and operator, and stored into their respective array, at that line number's index in the array. Then the program sequentially processes each member of the array. This is accomplished in a for loop that, again iterates from 0 to the number of lines given by the input. The result of the gate operation is found by using the gate() method, and then returned. The result is then copied in an output file with the name that was specified as an argument. Finally, the files are closed and the program terminates.

The gate() method is a series of if and else-if statements. It intakes two operand values and one operator value. The operand value is compared to a list of set constants, each representing an operator (AND==0,OR==1,etc.). If the comparison is true, that gate operates on the two operands and returns the resulting value. If the given operator does not represent any of the known gates, a negative value is returned to show the error.

We measured the time for the program to execute the logic gate operation sequentially. We will show, discuss and compare the results with the other methods used in part 5.

2. Logic gate simulation using explicit memory allocation in CUDA

The following portion will discuss our design of parallel logic gate simulation with use of explicit memory as it is coded in the file explicit.cu.

The process begins in main() with the same system of reading the given inputs as our sequential process. The first difference begins when we allocate memory for operands and operators. In this case we also allocate an array of input length to hold the results of our parallel gate computations. The operand and operator arrays are assigned with input values in a for loop identical to our sequential program. Using cudaMalloc(), a memory equivalent of our four arrays is allocated for cuda parallel computing. We copy the three input arrays to the cuda memory. Then the gate() method is called on a set number of threads, in our case 1024. This gate method has changed from the sequential in that it takes the cuda memory pointers of the operands, operator and result arrays as input, instead the simple values of operators and operand. After the cuda threads complete their gate calls, the dev_res pointer is copied from the device to the host and stored in the results array. After the results are copied, the memory for all the cuda arrays is freed. The results array is iterated through, printing each result to the output file. Finally the memory is freed and files are closed, and the program terminates.

The gate() method has changed so that it can be used in multiple threads. It is now marked as global so that the cuda threads may use it. Rather than taking value inputs, the gate() method takes in pointers to the array's. The index line of the array the individual thread should work on is found by the equation: blockId.x*blockDim.x+threadIdx.x. Using the found index, the operands and operator are found from their respective arrays. The gate calculation if statements function the same as in parallel, with one major difference. Rather than the method returning a value, it writes the found value into the allocated memory for results at the found index.

We measured the time for the program to migrate the data to the device (host to device) as well as the execution time for the kernel function. We will show, compare the results with the other methods used in part 5.

3. Logic gate simulation using unified memory allocation

The following portion will discuss our design of parallel logic gate simulation with use of explicit memory as it is coded in the file unified.cu.

The process begins in main and parses through the inputs in the same way as the explicit memory portion. When allocating unified memory, rather than allocating arrays the size of the input, simple pointers for the operands, operators, and results. Rather than use the method cudaMalloc() to create an explicit memory for the parallel computations, the method cudaMallocManaged() is used to give cuda access to the memory at the pointers, rather than allocating another location in memory and copying the values there. Next the input file is broken down into the array components as shown previously. Unlike our explicit memory program, this program allocates memory for cuda before the array values are assigned, since there is no need to copy these values for cuda to use. The parallel computing begins when the gate() method is called on our threads. The gate method is called with the unified memory locations for the operands, operator, and results. Due to the unified memory, there is no need to copy memory from the cuda to the original pointer array to retrieve the results. The results are written to the output file as shown in the explicit results section. Finally the memory is freed and files are closed, and the program terminates.

The gate() method operates identically from the explicit memory portion.

We measured the time for the program to execute the kernel function. We will show, discuss and compare the results with the other methods used in part 5.

4. Parallelization with data prefetching

We chose not to complete this optional part.

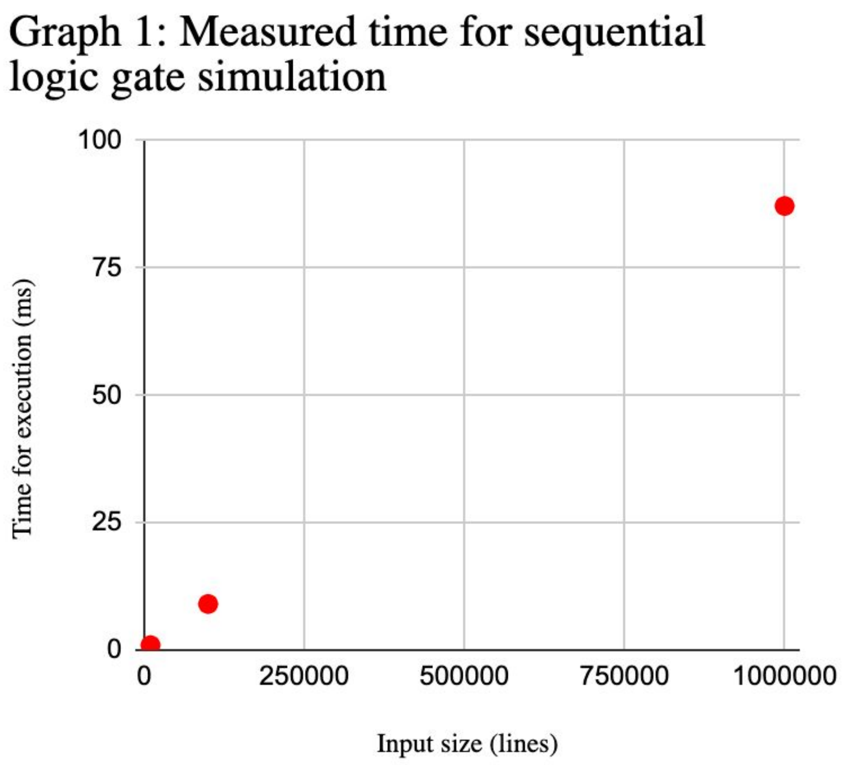
5. Discussion and comparison of results

Sequential

We measured the execution time for the sequential operations. Below is a table and a graph that show the results for each input size.

Table 1 : Measured time for sequential logic gate simulation

Input size (lines)	Time for execution (ms)
10,000	0.947000
100,000	9.039000
1,000,000	87.176000



What we can see from looking at the collected data is that the time increases as input size increases. The results suggest a linear correlation (in both the input size and time increase by a factor of ~10). For a small input size, the execution time is small,which is desirable, but it quickly becomes a lot larger as the input size increases.

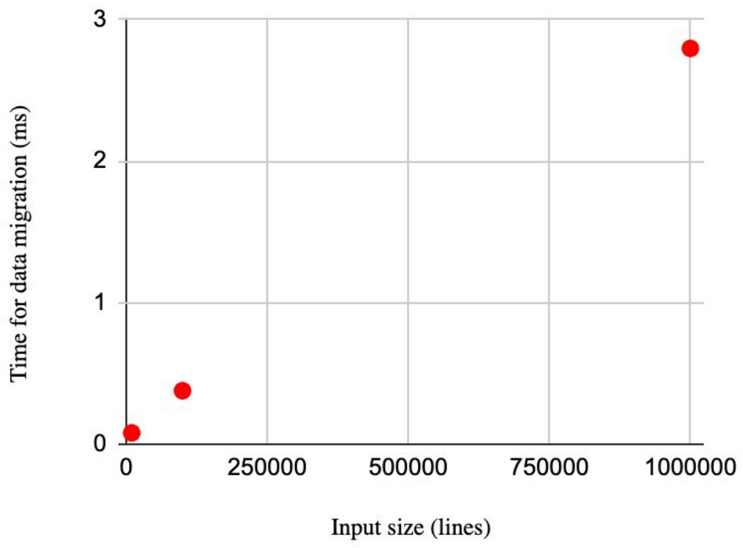
Explicit memory allocation

We measured the time for the program to migrate the data to the device (host to device) as well as the execution time for the kernel function. Below is a table and graphs that show the results for each input size.

Table 2: Measured time for explicit memory allocation logic gate simulation

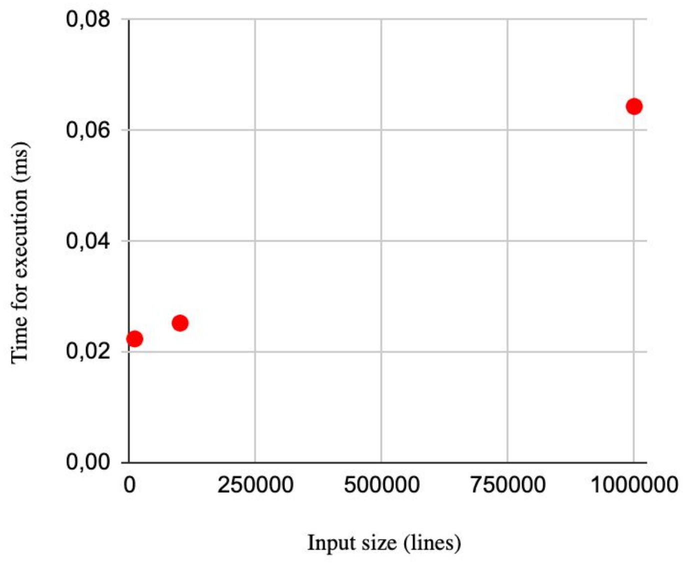
Input size (lines)	Time for data migration (ms)	Time for execution (ms)
10,000	0.079808	0.022368
100,000	0.377696	0.025216
1,000,000	2.796512	0.064352

Graph 2: Measured data migration time for explicit memory parallelization logic gate simulation



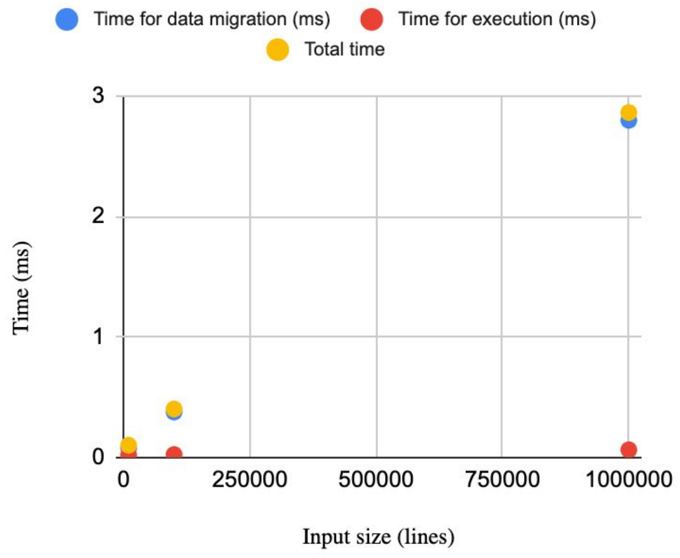
Graph 2 shows the time for data migration from the host to the device before the kernel function is executed. We can see that the time for data migration increases with input size, but on a different scale than with the sequential method (the increase is slower).

Graph 3: Measured execution time for explicit memory parallelization logic gate simulation



Graph 3 shows the time for the execution of the kernel function (execution of the logic gate simulation). There is a slight increase in time for the input size, but the execution time still seems very short for large input sizes (less than 1 ms).

Graph 4: Total measured time for explicit memory parallelization logic gate simulation



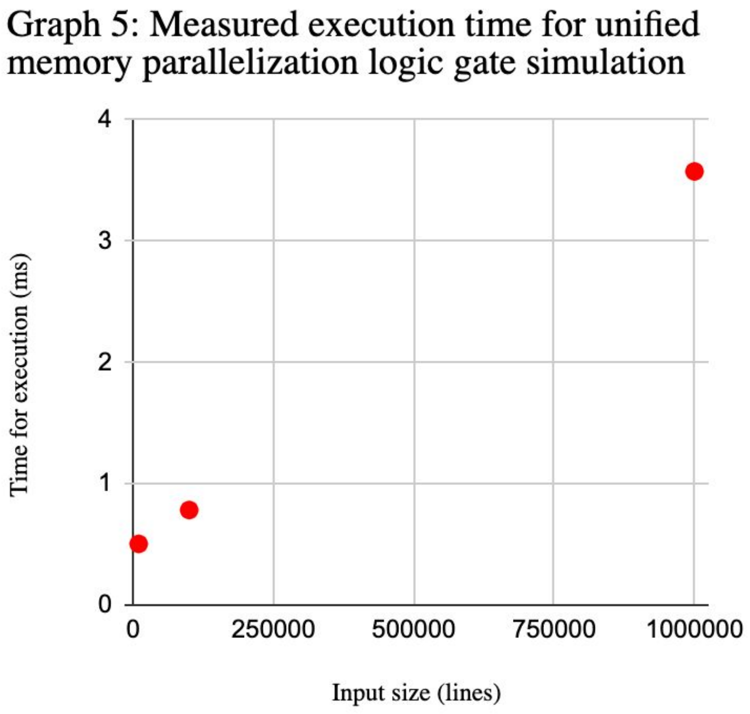
Graph 4 aims to show the breakdown of data migration and execution time, by showing the total time on top of the dots. We can see that the majority of the time spent in the explicit memory allocation method is migrating data, and the actual execution time is very short even for large output sizes.

Unified memory allocation

Below is a table and a graph that show the results for each input size.

Table 3: Measured time for unified memory allocation logic gate simulation

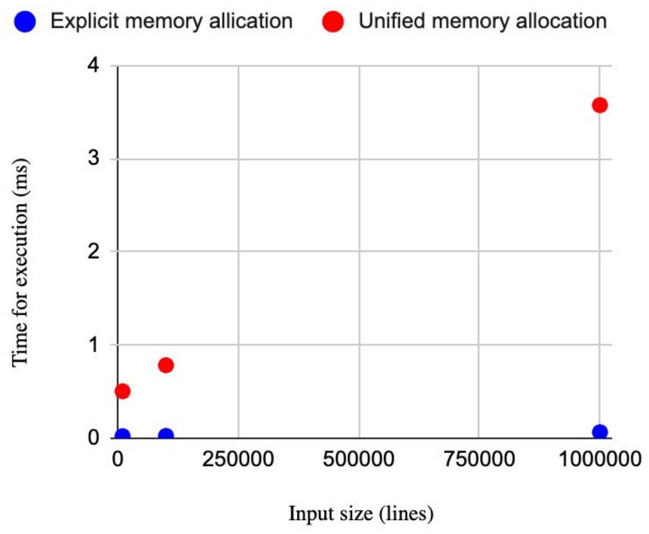
Input size (lines)	Time for execution (ms)
10,000	0.504096
100,000	0.783200
1,000,000	3.574272



Graph 5 shows the measured execution time for the unified memory allocation method. We can see that the execution time when using unified memory increases with input size, but again not as much as with the sequential method. This is likely due to having to access the different values in memory for each operation, but we will see below that it is still shorter than having to copy the data to the device entirely. The execution time is still relatively small for large input sizes.

Comparison of methods

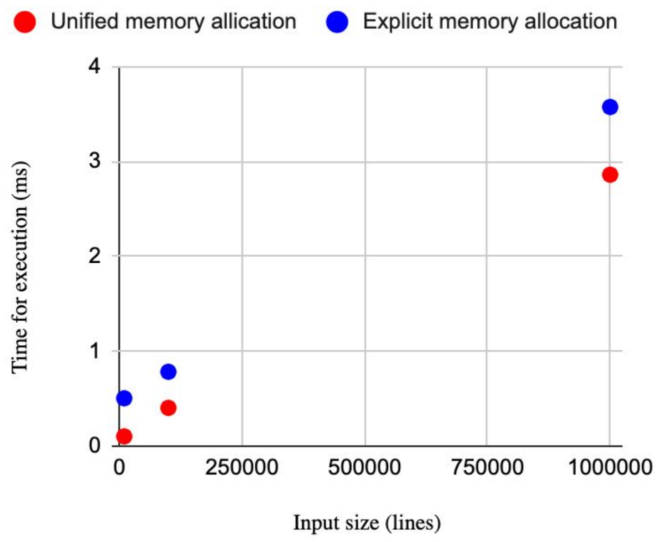
Graph 6: Comparing execution time for explicit and unified memory allocation



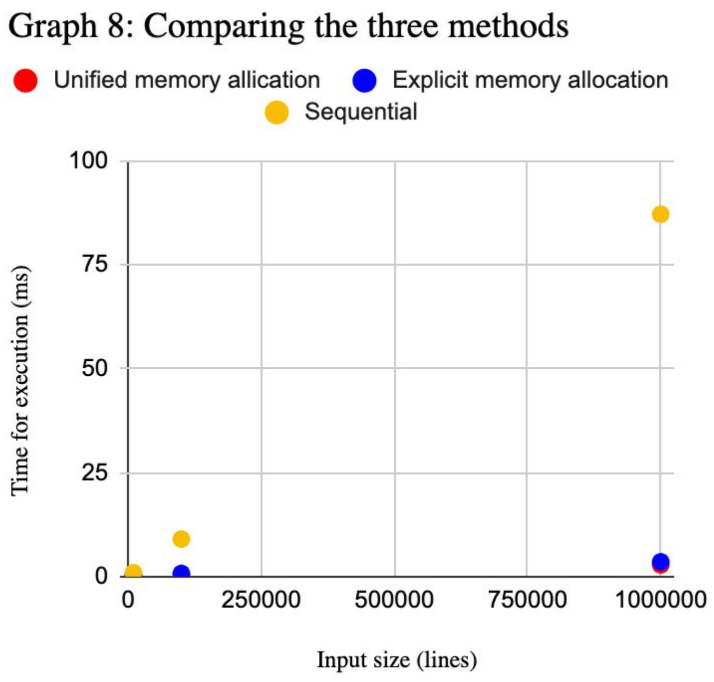
In Graph 6, we show strictly the kernel function execution time for the parallelization methods: explicit memory allocation and unified memory allocation. We see that the explicit memory is much faster no matter the input size. This is due to the unified memory method having to access the values in memory to complete the operations, whereas the explicit memory allocation has a copy of the data onto the device.

A more accurate comparison will be made below, taking into account the data migration time as well.

Graph 7: Comparing total time for explicit and unified memory allocation



In Graph 7, we compare the total time for both parallelization methods, which is more relevant than the previous comparison. Here, we see that the unified memory allocation is actually faster than the explicit memory method. This is attributed to the time it takes to copy the data from the host to the device, which takes more time than accessing unified memory.



Graph 8 shows the three methods, to illustrate that the sequential method can be ruled out from the list, it is clearly not the most efficient of the three methods, even with relatively small input sizes.

Conclusion

We conclude that the unified memory allocation is preferable, as the total execution time is faster at all input lengths, as shown in graphs 7 and 8.

We believe this is due to unified memory not needing to allocate and copy memory for the GPU's use. In our implementation of explicit memory, two times the memory is required. The input length of memory is copied to and from the explicit memory a total of four times (operator and operands to the CUDA, and once back to Host with the results array). Unified memory avoids these, and therefore is more memory and time efficient. Since these copies and allocations happen during sequential portions of the program, their time execution is not affected by parallelism.

Memory Efficiency: M_Exp(n) > M_Seq(n) = M_Uni(n)

Time Efficiency: T_Seq(n) >T_Exp(n) > T_Uni(n)