Vincent Rinaldi *239759 IC*
Introduction to Multiprocessor Architecture *Project*

Report

Assignment 4

1- Explain your « GPU_array_process » implementation and its optimizations.

A. : There are 5 parts in our « GPU_array_process » function : preprocessing, copying input and output arrays from host to device, computation of the final output array, copying output array from device to host, and postprocessing.

Preprocessing :

After creating an available device with « cudaSetDevice » function, we do two major things. We first allocate memory for two arrays on the device side which will receive the content of the initialized input and output arrays from the host side, to perform the computation on the device side. To do this, we first declare our two pointers of « double », then we use the function « cudaMalloc » on both of them.

The second thing, which will participate a lot in the optimization of the function, is that we will set the number of blocks used and the number of threads that will work into each block, to perform the computation on the device side. In general we want to size our blocks/grid to match the data and simultaneously maximize occupancy, that is, how many threads are active at one time. Usually, a thread block size (the number of threads running in a block) should always be a multiple of 32, because kernels issue instructions in warps (which are groups of 32 threads) and putting a value not equal to a multple of 32 would waste a part of created threads.

Then we should try to size our blocks based on the maximum number of threads and blocks that each corresponds to the compute capability of each Streaming Multiprocessors. A common objective when coding a GPU function, is to try to make use of the maximum number of threads that can run at the same time in each SM to reach maximum occupancy. But because shared memory and register usage may also be limiting our occupancy, the number of blocks per grid and threads per block is then better determined by the problem size, such as the matrix dimensions in the case of matrix computation (which is our case). So we decided to set the number of active threads (number of threads in a block * number of blocks in the grid) to be equal (or a little bit greater if the number of cells in the matrix is not a multiple of the chosen number of threads in a block) to the number of cells in the matrix.

To set this total number of active threads, we first have to choose a fixed number of threads per block, and then calculate the minimum number of blocks needed to have the number of cells, in a single row of the matrix, being less or equal than the total number of threads. Finally, we square these numbers of active threads and blocks (because we always have a squared matrix) with the « dim3 » vector type, which will give threads and blocks two dimensional indexes.

Even if it's common to recommend running more threads per thread block to hide latency, it seems that for some common computations (like matrix multiplications), their executions are faster when they run at lower occupancy, and so for a lower number of threads per block. This means that, when maximizing occupancy, we may lose performance, and it's known that faster codes run at lower occupancy (because fewer threads means more registers per thread, and only registers are fast enough to get the peak bandwidth).

Then after testing different lower fixed values of threads per block, such that we couldn't set more than 32 threads per one dimension block (certainly because when squaring it, we go over a maximum number of active threads limit, giving the reason why no computation happens), and keeping the same formula to calculate the minimum number of required blocks on one row of the matrix (which depends of the number of threads in a single block, and the length of the array, which is equivalent to the number of columns or rows), by testing different power of 2 values (which must be strictly greater than 4 so that, when squaring them, we are sure that they can be divisible by 32 to have a whole number of warps and not wasting any created threads), the best computation time we found was for 8 threads per one dimension block (which then goes to 8 * 8 = 64 threads per two dimensions block). Having 64 total active threads per block is great because then, the kernel can issue instructions, among each block, in warps without wasting a part of the created threads.

Copying input and output arrays from host to device :

We simply use « cudaMemcpy » function on both input and output arrays on the host side to copy their content in the existing ones on the device side. This transfer of data from the host side to the device side is the first one between the two sides in our code.

Computation of the final output array :

To compute the final output array, we will execute a « for » loop « iterations » times (where « iterations » is the given argument), and at the end of each iteration, we will swap the input and output array to make the computation faster, instead of copying at each time the content of the computed output array into the input one (we obviously don't swap at the last iteration because we have computed the final output array at this moment).

At each iteration, the computation is made through a __global__ auxiliary function, a kernel, to which, in addition of the input and output arrays as arguments, we specify, as parameters, the number of blocks in the grid, and the number of threads per block, that will be used by the invoked kernel to perform the computation on the device side. In this kernel function, we simply calculate the coordinates i and j of the element of the matrix to compute thanks to the « threadIdx », the « blockIdx » and the « blockDim » values, associated to the concerning thread currently executing the body of the function. Note that the « blockDim » value is always the same for all threads, since it corresponds to the value of the number of existing threads in a block (which we chose and gave to the kernel function at its call). Then we do the usual computation of the new value of the element (i,j) of the matrix (the mean of its value and the value of each of its neighbors) if its coordinates are inside the matrix and not on the borders (where values stay at 0) or in the middle square (where values stay at 1000).

At the end, when the « for » loop finishes, we call « cudaThreadSynchronize » function which blocks until the device has completed all preceding requested tasks, such that no other operations are performed during the computation of the output array.

Copying output array from device to host :

This is the same procedure as the second step when we copied the content of the input and output arrays from the host side to the device side. The only difference here is that we will only copy the content of the output array from the device side to the host side, since that we are only interested in the final result. This is the last transfer of data between the two sides in our code.

Postprocessing :

We simply free the previously allocated memory for the input and output arrays on the device side with the function « cudaFree » to avoid memory leaks.

2- For each of the <length, iterations> combinations {<100,10>, <100,1000>, <1000,100>, <1000,10000>}:

a. Report the results for running the baseline (reference CPU function), your optimized GPU version, and your optimized CPU version (OpenMP multithreaded version from assignment 2, running for 16 threads).

A : Here are the results in seconds of the total execution time of each three functions (Reference CPU function, Optimized GPU function and Optimized CPU function from assignment 2 running for 16 threads) for each required combination <length, iterations> (note that each resulting output matrix has been verified and are the same for each function) :

Results in seconds for all three functions with combination <100,10> :

Reference CPU function	*Optimized GPU function*	*Optimized CPU function (16 threads)*
0.000266	0.1711	0.000522

Results in seconds for all three functions with combination <100,1000> :

Reference CPU function	*Optimized GPU function*	*Optimized CPU function (16 threads)*
0.02111	0.1811	0.01856

Results in seconds for all three functions with combination <1000,100> :

Reference CPU function	*Optimized GPU function*	*Optimized CPU function (16 threads)*
0.1489	0.232	0.03195

Results in seconds for all three functions with combination <1000,10000> :

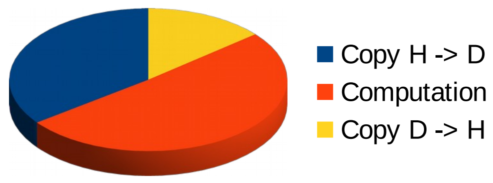
Reference CPU function	*Optimized GPU function*	*Optimized CPU function (16 threads)*
21.68	5.59	4.091

b. Present graphically the breakdown of the GPU runtime for copying data from host to device, computation, and copying data from device to host.

A : We first graphically represented the breakdown of each part of the GPU execution time (copying data from host to device, array computation, and copying data from device to host) without counting the CPU computation time to have a better representation of the proportion of time each operation takes compared to the other two, and then we included the CPU execution part to have a global view of the fraction of the total execution time each concerned operation represents. Each combination are represented as <length,iterations>.

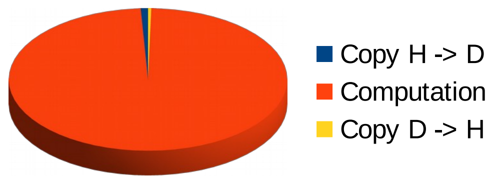
Breakdown of the GPU runtime excluding CPU runtime for the combination <100,10> :

Copy H → D takes 0.0001002 seconds
Array computation takes 0.0001444 seconds
Copy D → H takes 0.00003984 seconds



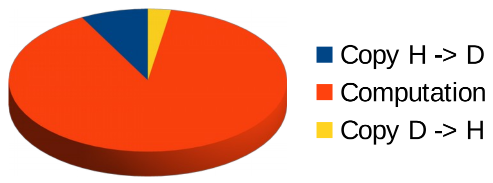
Breakdown of the GPU runtime excluding CPU runtime for the combination <100,1000> :

Copy H → D takes 0.00009267 seconds
Array computation takes 0.01086 seconds
Copy D → H takes 0.00004125 seconds



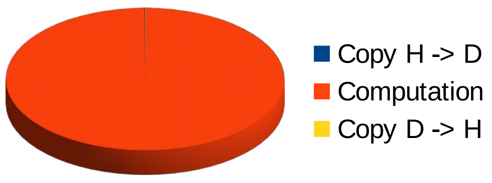
Breakdown of the GPU runtime excluding CPU runtime for the combination <1000,100> :

Copy H → D takes 0.004856 seconds
Array computation takes 0.05521 seconds
Copy D → H takes 0.001651 seconds

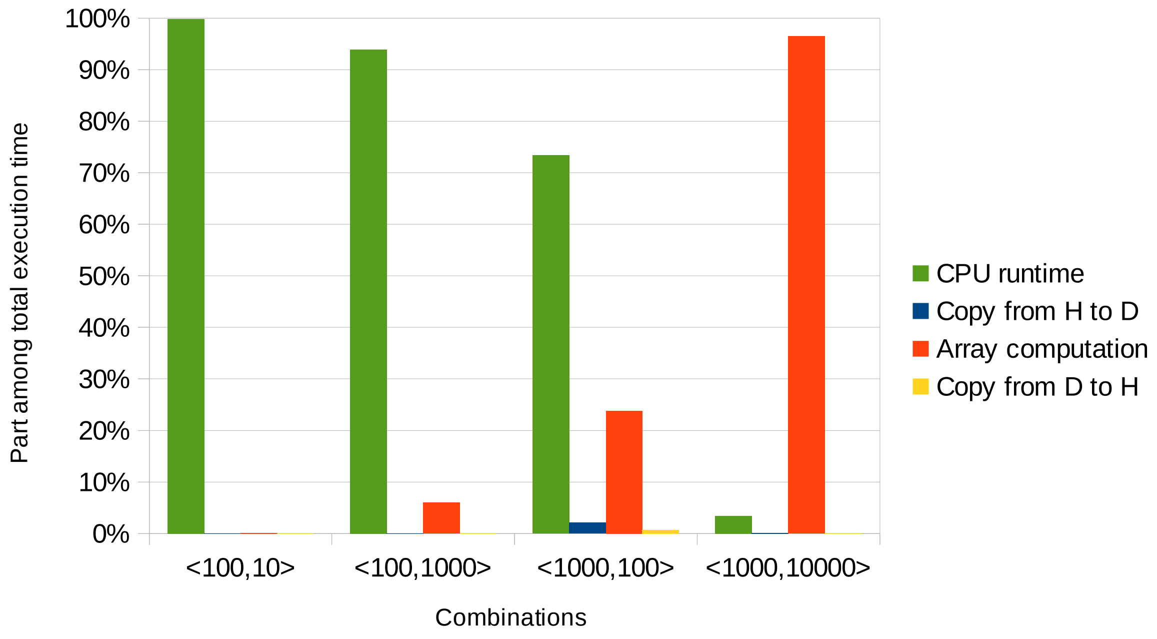


Breakdown of the GPU runtime excluding CPU runtime for the combination <1000,10000> :

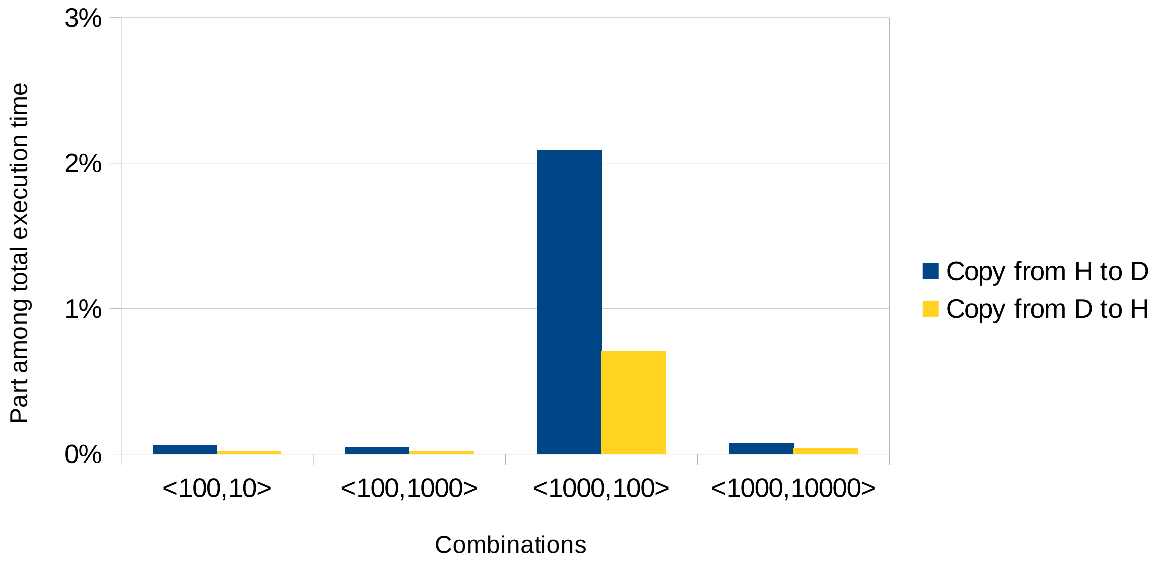
Copy H → D takes 0.004346 seconds
Array computation takes 5.394 seconds
Copy D → H takes 0.002411 seconds



Breakdown of the GPU runtime including CPU runtime for all combinations :



More precise graphic representation of the breakdown of « copying data from host to device » and « copying data from device to host » runtimes only, among total execution time, for every combinations :



c. Analyze and explain the results of parts a and b.

A : We see in part *a* that the execution time of the GPU optimized function is surprisingly not better than the CPU optimized function and the reference CPU function for the first three combinations. But since the execution time for these combinations is relatively very fast and doesn't require lots of time (less than 0.5 second), we will more focus on the fact that the execution time of the GPU function curiously doesn't vary a lot between these three combinations (we have an interval of 0,0621 seconds for the GPU function, and 0.1486 seconds for the referenced CPU one). This shows that the GPU function executes in a different way. The optimizations of the GPU function are highlighted for the last and heaviest combination in term of computations due to its huge number of iterations (10000). The GPU optimized function and the CPU optimized one have a way better execution time than the reference CPU function. These results are due to the fact that the reference CPU function is basic and absolutely not optimized by openMP directives or any optimizing programming stuff. Then these results against the reference CPU function were pretty predictable. About the two optimized versions (GPU and CPU), their execution times are almost equivalent. Even if the GPU one is a little bit higher, these results still show that clearly significant optimizations have been made to these functions. We could expect an even better execution time for the GPU implementation than the optimized CPU one, which lets us imagine that some more advanced GPU optimizations could have been added to the GPU part of the code, like in the kernel, such as how to better schedule the amount of work among threads and choose how many of them to create depending on the size of the arrays, to have better results.

In part *b*, concerning the circular graphs (or « cakes ») we first see that, the more iterations there are to do, the more the computation of the output array takes time. The copy from the host side to the device side takes more time than the copy in the opposite way, because we actually copy two arrays (the input and the output one) to the device side, but we then copy back only the computed output array to the host side. If we only increase the number of iterations, then only the computation time of the array will increase. In the case where only the length of the array increases, then the three steps of the GPU function (copy from host to device, computation, and copy from device to host) will all increase and still keep a significant fraction of the total time resulting only from the addition of the execution times of these three operations.

Concerning the bar graphs, we wanted to represent the proportion of time the three GPU operations take among the whole execution of the program (with the CPU execution part). We first see that the two copy steps of the array(s) from host to device, and from device to host, almost represent nothing among the total execution time (less than 0,5 % for three out of the four combinations, and a little bit less than 3 % for the fourth one which is <1000,100>). This result is not really surprising since these operations are really fast and simple. The most important detail is the fraction of the total execution time the GPU array computation takes depending of the combination. When the length of the array increases, the GPU array computation obviously takes more time to execute, and then represents a bigger fraction of the whole program execution, reducing the part taken by the CPU operations, and increasing the total execution time. But the major aspect, which then highly participates in the increase of the whole program total execution time, is when we increase the number of iterations, depending of the length of the array. Indeed, giving more iterations to a larger array will increase significantly the execution time of the GPU array computation, and that's why we clearly see on the graph, for <1000,10000>, the total domination of the concerned GPU operation over the CPU execution part (the GPU array computation represents here more than 95 % of the total execution time).