Implementations and optimizations:

We started by implementing a basic version of the function using the GPU. Measurements for this implementation can be found in graph (b). The goal was to divide the array in thread blocks, each thread computing a single cell. We initially used `cudaDeviceSynchronize` to synchronize between iterations, however we quickly realised that it was not necessary because the GPU waits until the last kernel terminates before starting a new one, so the explicit synchronization was pointless. Our first optimization was then to remove the `cudaDeviceSynchronize`, resulting in (a). We didn't use `cudaDeviceSynchronize` in any of our other versions. We obviously gained time by avoiding useless communications between CPU and GPU especially that it is executed `iterations` times.

The second observation we made was that we were rewriting the four central cells in global memory for every cell. When computing `output` we weren't checking if the indexes were in the middle, we rewrote it anyway and in order to keep correct results, we had to go over the center and rewrite them again to 1000 "by hand". The optimization in (c) takes care of that issue by checking if the current index is one of the four central cells, and if it is then we don't compute the mean value and we return immediately. That technique creates thread divergence since we add a condition on the indexes and thus a few threads behave differently from the rest. However the overhead created by the 9 global memory accesses in `input` and the 5 rewrites in `output` is bigger than the one induced by thread divergence, so we get a better time with that optimization, as we can see in (c). Turns out we could not get a better time than with that version, even when applying the same concept to the other algorithms, so we chose to keep the original version in the remaining optimizations in order to better analyse the results.

Next we tried to play with the thread blocks size and the number of thread blocks. So far we only used thread blocks of size 8x8. We tried to run our program with 16x16 and 32x32 thread blocks, keeping the same computation for the number of thread blocks. 32x32 was our maximum because past this we would get over 1024 threads per thread block, which is not supported by the SCITAS's GPU. Another reason why we needed smaller blocks is because when using smaller blocks, we have more of them (by the computation explained before) and then they are better distributed among the GPU's SM. Our goal was to avoid unused SMs. We also tried different block shapes. Instead of having squared blocks, we tried rectangular blocks of different sizes. To make sure that this rapport keeps a decent length, we decided to only keep the most "extreme" size, aka the row thread blocks. Basically we decided to have `length` blocks of size `length` x 1. The results can be seen in (d), and are not really interesting in the sense that they don't differ from (a) much. None of the size tries we made had an improvement, nor a deterioration. All results we got were pretty similar to (d) and (a). The explanation for the lack of changes in performances is that the amount of computation is almost identical for all sizes.

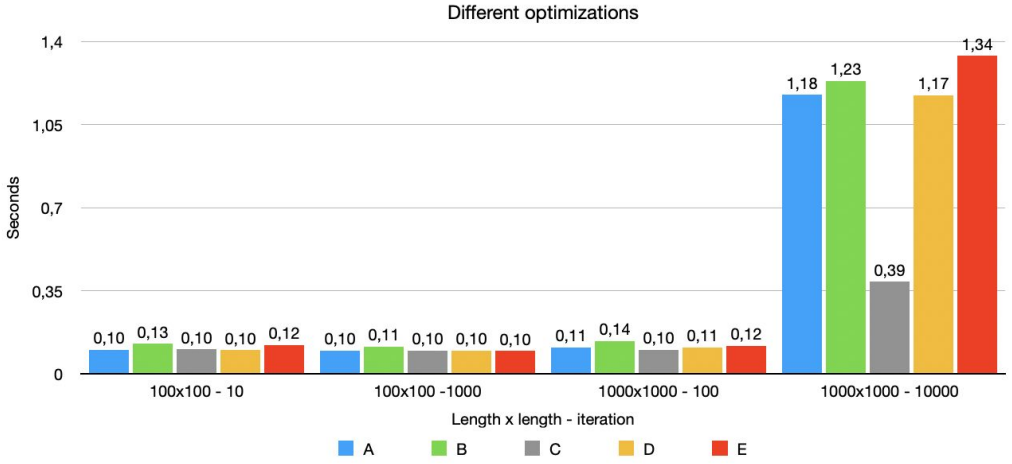
As seen in class, using shared memory could lead to better performances. Indeed, shared memory between threads in the same thread block is faster than global memory. The heat transfer algorithm could gain speed from the shared memory since every value in `input` is accessed 9 times for each iteration thus amortizing the overhead of copying the data from global to shared memory. The tricky part when implementing a version with shared memory is the border handling. After multiple attempts, we chose, for simplicity and to avoid too many divergent paths, to implement the kernel such that the only job of threads on the border of thread blocks is to load the value of their own cell into the shared memory. Thus thread blocks need to overlap in the array, giving version (e). As explained above in our baseline version we did not observe much performance variation by varying the block size since the amount of computation does not really change. This is no longer the case for the shared memory version. Indeed for each thread block, all border threads are spent to load data and are not used in the computation. Using larger blocks, we get a better computation / thread ratio. For example for a 8x8 thread block, 28 out of the 64 threads are not used for computation, which is almost half of the thread block, whereas for 32x32 block, the percentage of thread that is only used to load data drops to 12%. This is why we used a 32x32 thread block for that version. Again we could not use a bigger size because otherwise we would have more than 1024 threads per block. We chose to use square threads blocks because it is the shape that minimizes the borders length for a given area. As we can

see in the graph (e), the results we get for the shared memory version are not that promising, especially for the 4th combination. We did not get the speedup we thought we would get while writing that version. On second thoughts, we get these results because in the versions not using shared memory, we are using relatively small-sized thread blocks, which can benefit from the L1 caches that are shared among all threads within blocks. Since the blocks are small, the part we are working on can fit in the cache so there are not many cache misses and the time it takes to compute the output is not that big. We do not access 9 times the global memory, in reality we access it less often, so the cost of copying data to the shared memory is in reality not that worth it.

Performance of the optimizations:

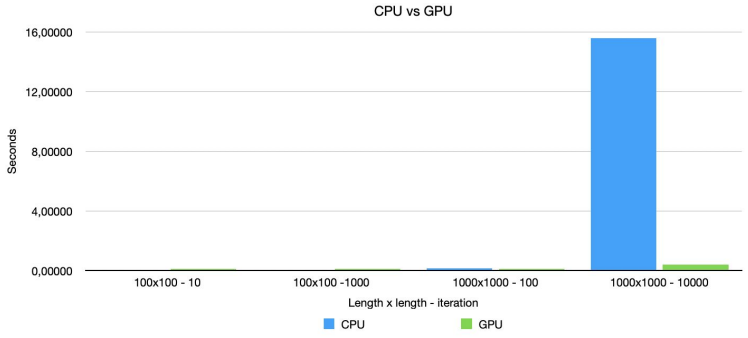
a) GPU Baseline
b) De-optimization: add `cudaDeviceSynchronize` to prove it is useless and it hurts performance
c) Optimization: Not rewriting central cells
d) Optimization: Changing the size of thread blocks into rows
e) Optimization: Using shared version shared memory

Below is a graph of the different optimizations we tried:

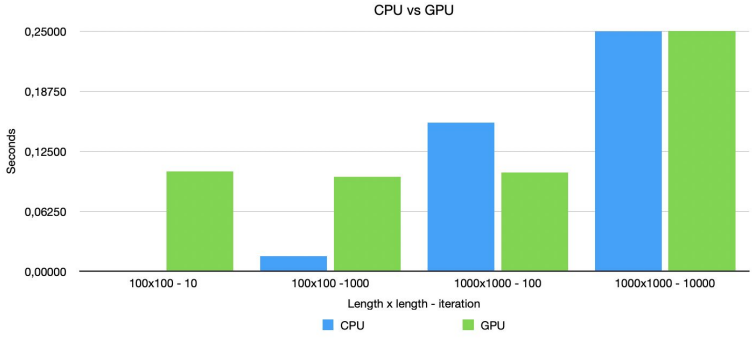


All the observations we can make on this graph are commented and explained in the first part of the report, above the graph.

CPU baseline vs GPU

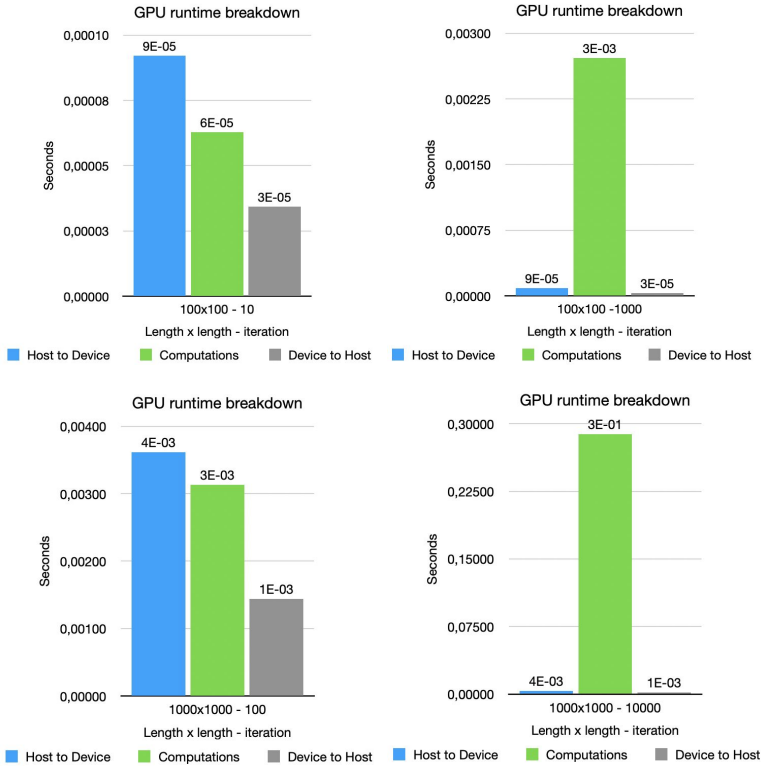


Above is the graph of the execution time measured for the CPU and our best GPU implementation, version (c). As we can see (or not see), that graph is not really speaking for the first three columns because of the huge difference in the order of magnitude. Here is the zoomed graph that has a better scale (1000x1000 - 10'000 columns are cut) :



GPU runtime breakdown

Here are the 4 graphs of the breakdown for each combination, using the version (c):



Results discussion

The CPU vs GPU graph could no better demonstrate the speed up of using GPU for large identical computations. Now if we take a look at the first three columns in the zoomed graph, the CPU runtime is increasing where GPU runtime stays constant. At first for a small array and a few iterations it is faster to directly compute the result on the CPU, indeed the overhead created by transferring data from CPU to GPU and back takes more time than the computation time. With the array size growing the computation part is so efficient on GPU that it is worth spending most of the runtime copying the data back and forth. As the number of iterations grows the power of the GPU speaks for itself. To be honest, we do not fully understand how total time can stay the same with the copy and computation section taking more and more time. We did the measurement multiple times and always obtained similar results. Our results are still good enough to demonstrate that for few computations there is an overhead of copying data when using GPU that could lead to longer runtime, but as soon as the amount of identical computation grows the speed up gained by using the GPU is huge (at most 40x speed up in our case).