1 Heat transfer on GPU

1.1 Introduction and suppositions

First of all, let's take a moment to appreciate the irony[1] of the situation, simulating heat transfer on a device that is known for often overheating.

Second of all, let's talk about the suppositions ; most of them are announced in the assignment :

- The table is a square,
- the length of a side is even and smaller or equal to $2^{10} + 2$,
- the 4 centre cases are maintained to exactly 1000 at all time,
- all other cases are initiated at 0[2],
- the first and last row and column (the border of the table) must be 0 at all time.

1.2 Optimizations explanation

For this assignment, I used the GPU capacity to schedule new TBs (and threads per TB) when too many are requested. Indeed, the GPU we use can run 60 TB at a time, and 512 threads per TB. But if we request more, the GPU will launch as many as possible and schedule TB to be run when a TB is completed. Similarly, if we ask for more than 512 threads in a TB, they will be run as soon as previous threads are completed. However, a GPU can only schedule a thread twice, meaning we can't launch a kernel with more that $512 \cdot 2 = 1024$ threads per TB.

An other important optimization is the use of kernels. Indeed, in our case, we need to completely finish an iteration before launching the next one. If we did all in the GPU, we would need to synchronize the threads inside the TBs, but also synchronize the TB between them. This is clearly not doable. To address this matter, we launch a kernel for every iteration. As a kernel can't start before the previous is finished, this implicitly forces every TB to synchronize in-between iterations. Once this is done, we can simple switch the reading-writer order (switching input-output pointers) and start again, with the certitude that all the previous computations are over.

Finally, as we don't ever need to write anything in the border cells, we can allocate fewer threads and TBs (2 less TBs and 2 less threads per TB) to save computation time.

1.3 Implementation explanation

As explained above, as long as we don't need more that 1024 threads per TB, we can have the following implementation : we allocate one TB per row, and in each TB we allocate one thread per case. Thus, using the optimizations mentioned before, for a $1'000x1'000$

[1] A visual representation

[2] Actually, the presented method would still work if not.

we will allocate 998 TBs, and in each of them 998 threads. The TBs and threads are allocated as a row (we don't pass a `dim3` argument) ; our view of the system lets us easily find the equivalent of row and column : the column is the `blockIdx.x` and the column is the `threadIdx.x`[3].

The rest is trivial : we allocate our arrays in the GPU, copy the input (not the output, as it's only 0's at this point) to the GPU, do our computation with pointer switching between each iteration, copy the result back to `output` and finally free all that needs to be.

1.4 Results comparison

{side, iterations}		{100, 10}	{100,1k}	{1k, 100}	{1k,10k}
CPU baseline		0.00015600	0.022096	0.149040	21.550000
CPU 16-threaded		0.00116125	0.277660	0.043718	5.381800
GPU	Total run	0.17076667	0.178400	0.200020	2.633400
	HostToDevice	0.00005936	0.000060	0.002480	0.002475
	Computation	0.00009900	0.007844	0.024710	2.455400
	DeviceToHost	0.00004941	0.000051	0.002443	0.002606

Table 1: Different methods with different size/iterations combinations

There are a lot of things to notice in the above table :

- For small inputs, the baseline is clearly much better than the two other methods. This is because the multithreading and the GPU calculation induce a overhead that is superior to the gain. Remember kids, Amdahl's law is Truth !
- In the GPU breakdown, the copies (H2D or D2H) are proportional to the side length (which is logical), but still represent nothing compared to the computation part.
- The computation part of the GPU is completely proportional to the iterations ; apparently, the overhead caused by the kernel start is negligible.

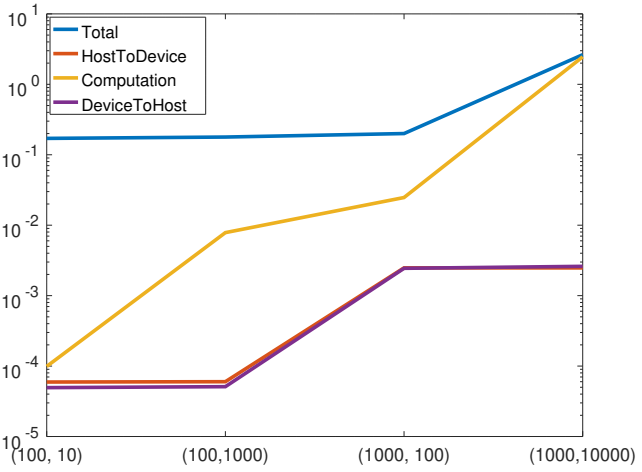

Figure 1: Visual representation of the GPU compute time breakdown. Y-scale in seconds

[3] Note that setting a TB per column and a thread per row would also work, as the table is a perfect square.

- The GPU version is better that the multithreaded at {100, 1k} and {1k, 10k}. We can conclude that launching multiple kernels one after the other induces less overhead than launching 16 threads multiple times.

- But, in contrast, fewer iterations with a big table (case {1k, 100}) is favoured by the 16-threaded, and thus in a purely computational point of view, the CPU is faster. This is probably due to the rescheduling of the TBs and the threads ; indeed, each time a block is over it needs to 'go back' to the CPU to ask for a new ID.

As we can see in Figure 1, the copy times (HostToDevice and DeviceToHost) not only take approximately the same time (logical, the amount of data to copy is the same both ways), but it is also linked to the size of the matrix (still very logical).

Furthermore, the computation is by far the largest part, proportional to the number of cases (size of the matrix multiplied by the number of iterations).