Poisson solver in CUDA

Louis Jaugey
louis.jaugey@epfl.ch

6 mai 2021

1 Introduction

In this work, we investigate the efficiency of parallelisation on GPU using CUDA. The code is a 2D finite differences Poisson equation solver. Two different ways of splitting the work among threads are tested. The "1D" versions use one dimensional thread blocks and each thread compute a whole dimension of the matrix. The "2D" versions use two dimensional thread blocks and each thread computes one entry of the matrix. The program performs a given number of iterations (1000 in this work) on a $N \times N$ matrix (here $N = 4096$). The code was executed on an NVIDIA V100 GPU (Compute Capability 7.0).

All values presented below are the average of 3 separate executions and are plotted on a log log graph. This means that polynomial expressions should be linear, with a slope equal to the exponent (since $\log(x^y) = y \cdot \log(x)$).

2 1D

The number of thread per block is varied between $2^0 = 1$ and $2^{10} = 1024$, which is the maximum allowed by this GPU. The number of blocks is given by :

$$n_{blocks} = \left\lfloor \frac{N-1}{n} \right\rfloor + 1 \tag{1}$$

Note that the global memory accesses are coalesced in the "One thread per column" scenario but non-coalesced using one thread per row. Indeed, when threads process columns, they access contiguous memory addresses.

Figures (1) shows the time per iteration obtained using both row-wise and column-wise processing.

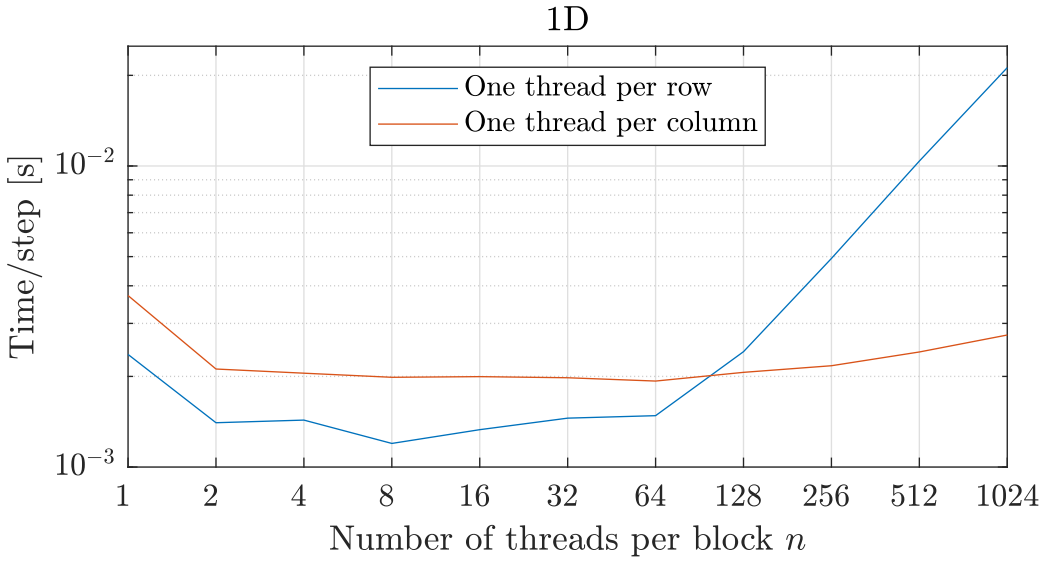

Figure 1 – Time per iteration for both 1D methods.

At first, one could expect that having one thread per column would be faster since accesses are coalesced. However, this is not what Fig.1 shows for less than 32 threads per block.

A potential explanation would would be that, although only one memory element is requested by a thread, the 128 bytes memory chunk is fetched anyways. Therefore, 32 (number of threads in a warp) accesses will be issued and executed sequentially but 32×128 bytes will be fetched. This is a slow process but once all these values are cached, each thread can process the next $128/4 = 32$ (since `sizeof(float) = 4`) without waiting for global memory.

Using one thread per column means that warps switch between computing and fetching from the global memory every 32 elements (against 32×32 for the other scenario). This could explain the observed difference.

For larger blocks, the GPU becomes heavily under-utilized. Indeed, using $N = 4096$ and $n = 1024$ in eq.1 gives a total of only 4 blocks. V100 GPU's have 80 streaming multiprocessor (SM) and can therefore run many threads blocks simultaneously. Running only 4 at a time is a waste of resources. On the other extreme, having too few threads (<32) per block gives rise to a "warp under-utilization" where SM's run only a fraction of the maximal number of threads per cycle. The minimal execution time is somewhere in the middle of those two extremes, leading to a "U" shape, as shown on Fig.1.

3 2D

In this section, square 2D thread blocks are used. The number of blocks in each dimension is given by (1).

Two implementations are tested. The first is the standard one and the second uses shared memory. Instead of using values directly fetched from memory (as with the standard implementation), these values are stored in shared memory. The rest of the kernel is the same but uses the array stored in shared memory instead.

Note that blocks have a size of $n \times n$, hence only perfect square number of threads per block are measured. Global memory accesses can be coalesced using row-wise warps.

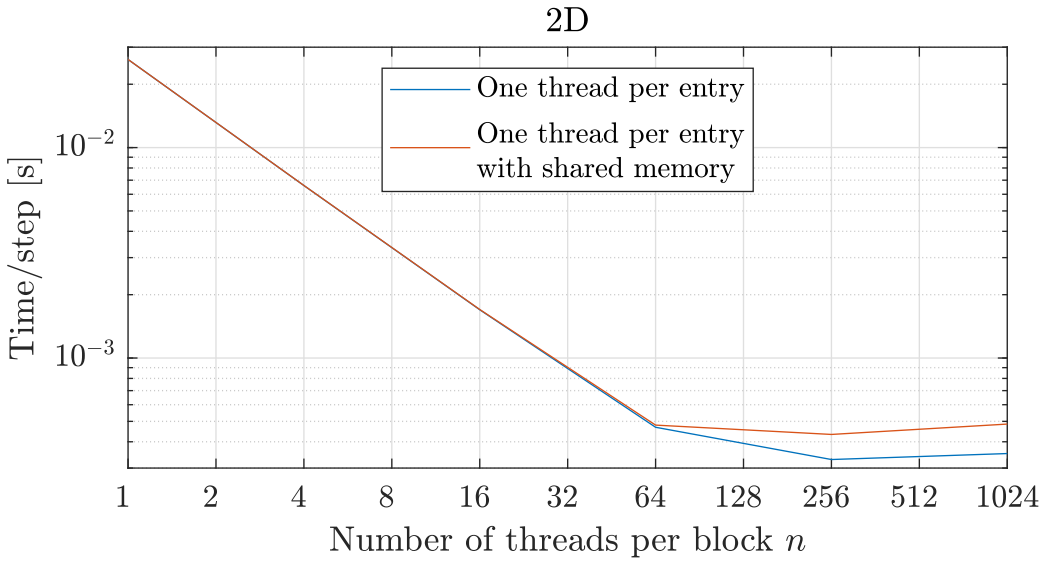

Figure 2 – Time per iteration for both 2d methods

This graph shows that execution time decreases linearly with the number of threads per block, until 64. Since up to 32 threads per block can run at the same time, this speed-up is expected for value below or equal to 32. At 64 threads per block, it seems like the GPU is able to overlap loading time by switching between warps, which further decrease time (linearly). However, with more than 64 threads per block, this gain mostly fades out and the execution time becomes almost constant.

Again, one could expect better results with shared memory but this is not what Fig.2 shows. Indeed, shared memory seems not to change execution time with less than 32 or 64 threads per block. However, the overhead caused by filling shared memory seems to negatively impact performance above 64 threads per block.

The absence of gain using shared memory may come from the nature of the problem itself. Indeed, accesses performed by this kernel are very standard (no huge array or column-wise accesses are required) and simple caching may be sufficient.

Since there is no need to manually fill the cache, the GPU does it in a more efficient way, leading to slightly better performances.

A Appendix

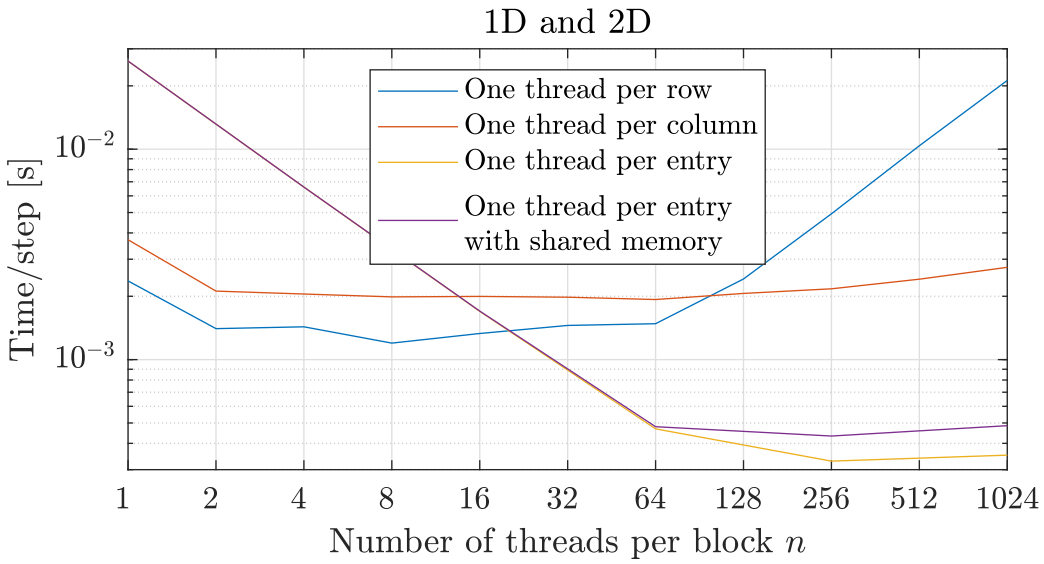

Figure A.1 – Summary of all execution types

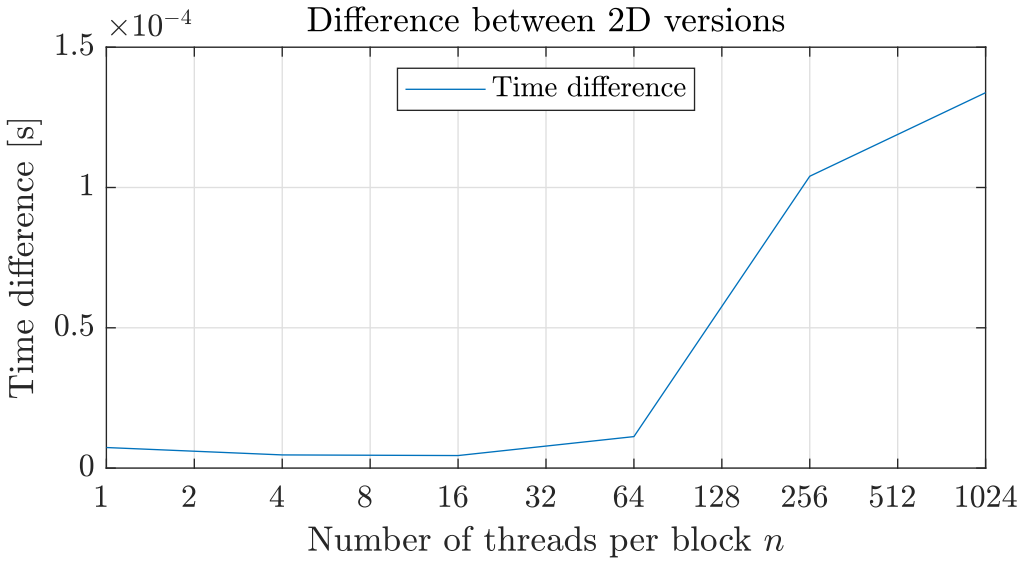

Figure A.2 – Time difference between using and not using shared memory