Charles Dufour
257587
charles.dufour@epfl.ch

CUDA 2D Poisson solver

P&HPC, MATH-454

2022-11-29

Simulation framework We implement the Poisson solver with CUDA kernels following two parallelization strategies. We fix $N = 8192$ and the number of iterations to 1000 for all the experiments. We use a Tesla V100-PCIE-32GB with compute capability 7.0 for the experiment (for more details see vola architecture white paper). The maximum number of blocks per streaming multiprocessor (SM) in our GPU is 32 (up to 1024 threads/block) and we have 80 SM available (up to 2048 threads/SM).

Per row strategy uses one thread per row of the grid. The number of blocks will then be the number of rows (N) divided by the number of threads per block. We let the number of thread per block vary between 2 and 1024 and repeat the experiment 10 times.

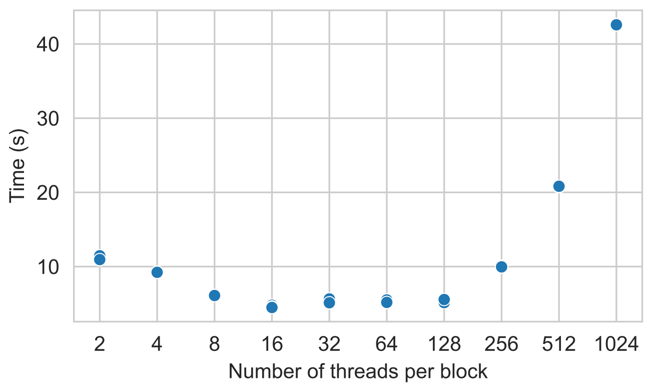

Figure 1: Time for 1000 iterations for different block sizes, per row strategy.

- Between 2 and 32 threads per block, we are not maximizing the occupancy. Each block will run its threads using warps of 32 threads, regardless of the number of active threads. If we add threads per block (while staying in this range), each block will take the same time to run and require the same resource thread-wise. Adding threads per block keep the time for one block to be computed, but reduce the number of blocks to compute. It is unclear why 16 threads per block performs as well as/better than 32.
- Between 32 and 128, we are using block sizes that are multiple of 32, so occupancy is maximized. At the same time, we have between 256 and 64 blocks, meaning we have no problem running all of them at the same time (≤ 2560) and warp and SM occupancy are maximized.

  Supposing the GPU tries to maximize the number of clock cycle per wave (probably not true, but will give an idea), we look at the number of clock cycles needed to do one operation on all the blocks. One SM has 64 FP32 cores, meaning it needs one clock cycle to issue a single instruction to 2 warps.

  - For 64 threads per block, we have 128 blocks that each need to run 2 warps. One SM can finish one block per clock cycle, but we need each SM to run 2 blocks, so at least 2 clock cycles.
  - For 128 we have 64 blocks, meaning that one SM needs 2 clock cycles to finish one block, but we can run all of them in parallel since we have less than 80 blocks.

  Hence 64 and 128 threads per block will take approximately the same time, as we can see in Figure 1.
- After 128 threads per block, one SM needs at least 4 clock cycles to issue one instruction for the block, meaning that 256 will double the amount of time than 128, and so forth, which is what we observe in the empirical measurements.

Per entry strategy uses one thread per grid entry. The number of blocks will then be the number of entries (N^2) divided by the number of threads per block. We let the dimensions of the block vary between 2 and 512 for the x and y axis, and only keep the feasible configuration (`dim.x` $\times$ `dim.y` ≤ 1024). We repeat the experiment 10 times.

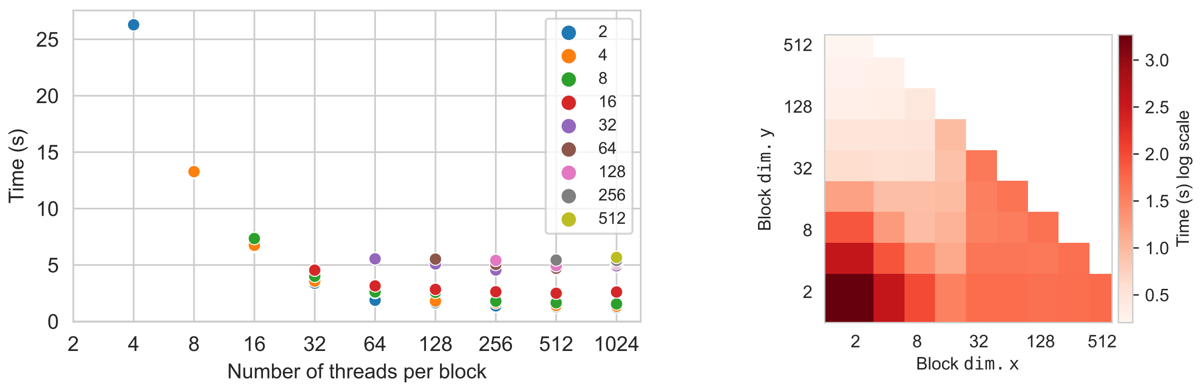

Figure 2: Time for 1000 iterations for different block sizes, per entry strategy. The color on the left hand size represent `blockDim.x`, the number of rows of the problem considered by one block.

- The same behaviour for the occupancy explains the gain of performance between 2 and 32 threads per block.[1] At 64 threads per block, each SM issues can issue one instruction for one block in one clock cycle, which is the optimal speed.
- We do not have an issue of having too few blocks: even with 1024 threads per block, we have $8192^2/1024 = 65536$ blocks. Meaning we cannot under utilize the SMs available. The huge number of blocks allows this strategy to not suffer from the bottleneck of the per row strategy: no SM core will be idle while others are doing a lot of work. The workload is more balanced by the scheduler.
- We can see the influence of the number of cache misses in performance. For a fixed number of threads (`dim.x` $\times$ `dim.y`), if a block has a high `dim.x`, it will deal with more rows than columns, leading to more cache misses. On the other hand if `dim.y` is high, it will deal with a lot of columns for a comparatively small number of rows. This behaviour is more clearly seen by looking at the right hand side of Figure 2, and noticing the difference between the upper left corner and the bottom right.

Creating threads and blocks is not expensive for a GPU, its capability to handle lots of threads at the same time means that it is better to have one thread per entry, rather than having less threads with more work each, meaning that a block won't wait on a few threads to finish while the rest are done. Having enough blocks with balanced workload also allows more flexibility for the GPU to schedule the blocks.

[1]In our experiment we tried values that were not multiple of 32 but higher than 32 (i.e. $32 \times k + 1$, $k > 1$), and saw a drop of performance because of the warps not being fully occupied.