Timers in CUDA framework

1. Analysis of the timedReduction kernel

The kernel is used to find a minimum of an array and calculate how much time it took for every block. In order to parallelize the solution to the task, this kernel performs a reduction on that array. We launch the kernel with NUM_BLOCKS blocks and NUM_THREADS threads. The input array has 2*NUM_THREADS elements. The kernel starts by declaring array in shared memory (let's call it shared), large enough to store input array. We use array called timer to store the start and end time of our kernel executing for each processing block (to calculate the time we use clock() function). Then each thread is responsible to fill 2 elements of that array (shared[threadIdx.x] and shared[threadIdx.x+blockDim.x]) with appropriate values from input array. Next, in a loop each thread will compare elements shared[threadIdx.x] and shared[threadIdx.x+blockDim.x]). If the element with larger index is lower we assign it to shared[threadIdx.x]. In the next loop we use only half of the threads we used before (because the second half of shared consists of elements higher than those in the first half). We repeat the proccess until we have the minimum element in shared[0]. Because all threads work on the same shared memory, we need to synchronize them at the start of every loop. After the loop we use thread 0 to call clock() function and store it in timer array.

2. Different data sizes and processing grid configurations

Number of elements in the input array is directly connected to number of threads per block(number of elements = number of threads per block * 2), if we increase number of threads we also increase size of input array which means that each block will take longer to execute. For example configuration with 64 blocks and 64 threads per block results in each block taking 0.001828s. to execute (on average). Increasing number of threads to 1024 results in time: 0.003246s.

If we set number of processing blocks in a grid to be lower than or equal to the number of SMs on our device (30 in this case) the results are very similar(~0.2350s. per block). With more processing blocks than the number of SMs there is higher latency in memory which results in longer execution of each block (~0.002768s. per block).

3. About clock() function

Function clock() returns processor time consumed by the program up to the point when the function is used. The return value is type clock_t and is expressed by clock ticks (which are system-dependent units of time). This function can be used to calculate how long a specific block of code in our program run. We need to call clock() function twice (at the start of the block of code and at the end). By substracting the start time from end time we will receive number of clock ticks. If we divide that number by a constant CLOCKS_PER_SEC we can calculate how many seconds it took to execute.

4. Analysis of matrix multiplication kernel

This kernel will use tiling approach to calculate matrix C which is the result of multiplying matrices A and B. This is a template kernel and we use BLOCK_SIZE variable as a dimension of a single tile (each tile will be of size BLOCK_SIZE x BLOCK_SIZE). We create a fe variables that will be used in the kernel: aBegin is index of first element in first tile in matrix A that will be calculated by current block, bBegin is the same but for matrix B, aEnd is the index of the last tile of matrix A calculated by current block, aStep and bStep are used to get from current tile to the next one (aStep = BLOCK_SIZE, bStep = BLOCK_SIZE * width of B). Then we create a variable to store the element of the block which is

computed by current thread. In a loop we create arrays As and Bs of size BLOCK_SIZE x BLOCK_SIZE (these arrays use shared memory and will represent pairs of tiles(sub-matrices) that will be multiplied). Then each thread assignes corresponding values from matrices A and B to sub-matrices As and Bs and we synchronize threads. Then each thread computes one element of sub-matrix. We repeat this process for each pair of tiles, and add the results to a variable that is local for every thread. After synchronizing threads we can assign dot products calculated by each thread to corresponding elements in matrix C which when the kernel finishes execution is the result of multiplying matrices A and B.

5. Timing using CUDA events

CUDA event API is an alternative to CPU timers that allows to create events and compute elapsed time between those events. Cuda events are of type cudaEvent_t and can be created with cudaEventCreate() adn destroyed with cudaEventDestroy(). Using cudaEventRecord() places two events(start and stop) in default stream and the device will record time for the event when it reaches when it reaches that event in a stream. Then using cudaEventElapsedTime() returns time (in miliseconds) elapsed between two results passed as arguments. Major differences between using CUDA events and CPU timers: CUDA Events use streams, recording time happends on the device which makes it more precise, CUDA events have lower latency.

6. Testing out with different data sizes

Now we are going to check time performance of our programs using different data sizes:

Matrix A	Matrix B	Matrix C (number of elements)	Block size(tiles dimension)	Time elapsed (ms)
320 x 320	320 x 640	204800	32	0.192
160 x 160	160 x 320	51200	16	0.030
80 x 80	80 x 160	12800	8	0.008
40 x 40	40 x 80	3200	4	0.005