Exercise 3

Step 1:

For making use of global/device memory, additional functions were made. For the code, see *Globmem\_Av\_Product (), Globmem\_FindNormW (), Globmem\_NormalizeW (),* and *Globmem\_ComputeLamda ()*. The table below shows the time comparison (in seconds) when using global memory vs shared memory for the default matrix size of 5000x5000 and a fixed number of iterations (1000).

Observe that as expected, the runtime when using shared memory is greater than double that of when using global memory. This is because shared memory is located on-chip unlike the global device memory. This is analogous to the relation between cache and main memory in computers. However, like cache, the drawback of shared memory is that it is much smaller in size compared to the device memory.

|  |  |  |
| --- | --- | --- |
| memory type | total time (sec) | iterations |
| shared | 0,97 | 1000 |
| global | 2,24 | 1000 |

Step 2:

The table below shows how the runtime varies with the matrix size and the number of threads within each block while keeping the number of blocks fixed for 1000 iterations. The tests could not be done for 128 threads per block because the shared memory sizes were exceeding the allocation size limits.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1000 iterations | threads-per-block | |  |  |  |  |
| matrix size | 2 | 4 | 8 | 16 | 32 | 64 |
| 50 | 0,295 | 0,293 | 0,293 | 0,315 | 0,260 | 0,312 |
| 500 | 0,347 | 0,340 | 0,326 | 0,297 | 0,349 | 0,360 |
| 2000 | 0,702 | 0,556 | 0,548 | 0,522 | 0,565 | 0,547 |

The results show that the runtimes increase with increasing matrix dimensions, however this gap becomes clearer as the size increases. We also see that there is an initial decrease in the execution time as the number of threads per block increases. But after a point, the runtime again starts increasing. Since the number of blocks remains fixed, the initial decrease is expected because the total number of threads in the entire grid is increasing, hence more parallelism. After a point, increasing the number of threads will not improve the execution time because the total number of threads spawned is greater than the number of rows in the matrix. Hence there are numerous threads that are just idle. The increasing overhead of managing more threads, coupled with almost no increase in parallelism could be the reason for the time to increase again.

Step 3:

An additional timer was created to measure the time spent in CPU-GPU memory transfers. For code, see *mt\_start* and *mt\_end.* Two sizes of the matrix were tested (5000 and 500), to see the effect of the dimension on the ratio of memory access time to the total time. Shared memory configuration was used for 1000 iterations.

|  |  |  |
| --- | --- | --- |
| size | total | memory |
| 5000 | 1,01 | 0,29 |
| 500 | 0,35 | 0,27 |

For max efficiency, the compute-to-communication ratio should be as has high as possible. The table above shows that increasing the matrix size leads to a significant increase in the overall compute time, with only a small change in the CPU-GPU communication. Hence a GPU is a good choice when dealing with larger matrix sizes. For small matrix sizes, the additional cost of transferring the data to the GPU may not outweigh the runtime of a single CPU, so in that case running on a CPU would be the better option.