Report for Lab 6

Task 1 – Basic Implementation

- Text 1...
- Text 2...
- Text 3...
- Text 4...
- Text 5...

Task 2 – Tiling Implementation using Shared Memory

- The used machine has 49152 bytes shared memory per thread block.
- Since we are using `floats` as a data type, each thread block can store 12288 `floats` in its shared memory. Therefore, each of the shared-memory matrices (`a`, `b` and `c`) can contain up to 4096 entries.
- The following snippet shows the implementation of the new tiling matrix multiplication kernel.

```
__global__ static void matMultCUDA(const float *a, const float *b, float *c, int n)
{
    __shared__ float shared_a_tile[BLOCK_SIZE * BLOCK_SIZE];
    __shared__ float shared_b_tile[BLOCK_SIZE * BLOCK_SIZE];
    __shared__ float shared_c_tile[BLOCK_SIZE * BLOCK_SIZE];

    int j = threadIdx.x + blockIdx.x * BLOCK_SIZE;
    int i = threadIdx.y + blockIdx.y * BLOCK_SIZE;

    if (0 <= i && i < n && 0 <= j && j < n)
    {
    // Init shared output tile
    shared_c_tile[threadIdx.y * BLOCK_SIZE + threadIdx.x] = 0;

    // Iterate over all tiles
    for (int k_tile = 0; k_tile < n; k_tile += BLOCK_SIZE)
    {
        // Copy current tiles
        shared_a_tile[threadIdx.y * BLOCK_SIZE + threadIdx.x] =
            a[i * n + (threadIdx.x + k_tile)];
        shared_b_tile[threadIdx.y * BLOCK_SIZE + threadIdx.x] =
            b[(threadIdx.y + k_tile) * n + j];
```

```
            __syncthreads();

            // Compute tile of elements
            for (int k = k_tile; k < k_tile + BLOCK_SIZE && k < n; ++k)
            {
                shared_c_tile[threadIdx.y * BLOCK_SIZE + threadIdx.x] +=
                    shared_a_tile[threadIdx.y * BLOCK_SIZE + (k % BLOCK_SIZE)]
                    * shared_b_tile[(k % BLOCK_SIZE) * BLOCK_SIZE + threadIdx.x];
            }

            __syncthreads();
        }

        c[i * n + j] = shared_c_tile[threadIdx.y * BLOCK_SIZE + threadIdx.x];
        }
    }
```

- The `BLOCK_SIZE` variable in the following snippet is instantiated with either 4, 8, 16 or 32. It is limited to 32 since $32 \cdot 32 = 1024$ is the maximum of threads per thread block. In the case of a block size of 4×4, for example, there are `ceil(n / BLOCK_SIZE) = 250` blocks per dimension (250×250 blocks in total).

  ```
  // 2D config with given block size
  dim3 dimGrid(ceil(n / BLOCK_SIZE), ceil(n / BLOCK_SIZE), 1);
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE, 1);
  ```

- Table 1 shows the execution times from 5 runs for all different configurations. Since the differences are not significant, there is no configuration that is better suited than the others. However, we expected that blocks of size 32×32 are best suited for the tiling matrix kernel since the larger the block size is, the less `__syncthreads()` calls there are, which means less overhead.

	4	8	16	32
Run 1	3.252 s	3.189 s	3.323 s	3.085 s
Run 2	3.138 s	3.183 s	3.285 s	3.355 s
Run 3	3.127 s	3.115 s	3.111 s	3.317 s
Run 4	3.123 s	3.288 s	3.097 s	3.100 s
Run 5	3.140 s	3.139 s	3.094 s	3.104 s
Average	3.156 s	3.183 s	3.182 s	3.192 s

Table 1: Execution times for different configurations.