**GPU Programming** – Lab 3 12.11.21 – 2021/22

**SHARED MEMORY**

**Goals:**

- To acquire confidence in the development and analysis of shared memory on GPU programs

**Tutorial: Shared Memory**

Please declare shared memory in CUDA C/C++ device code using the *__shared__* variable declaration specifier. There are multiple ways to declare shared memory inside a kernel, depending on whether the amount of memory is known at compile time or at run time. The following code illustrates various methods of using shared memory. Try to execute it on the jetson nano kit.

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}

__global__ void dynamicReverse(int *d, int n)
{
  extern __shared__ int s[];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}

int main(void)
{
  const int n = 64;
  int a[n], r[n], d[n];

  for (int i = 0; i < n; i++) {
    a[i] = i;
    r[i] = n-i-1;
    d[i] = 0;
  }
  int *d_d;
  cudaMalloc(&d_d, n * sizeof(int));
  // run version with static shared memory
  cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
  staticReverse<<<1,n>>>(d_d, n);
  cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
  // run dynamic shared memory version
  cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
  dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);
  cudaMemcpy(d, d_d, n * sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
}
```

**Exercise 1:**
Write a simple CUDA program to create an array of 1048576 (which is 2^20) double values in the GPU's global memory using *cudaMalloc* and using the shared memory. Initialize the array considering that index zero holds the value zero, index one holds the value one, index two holds the value two, and so on. Evaluate the different performances.

**Exercise 2:**

Please consider a cuda code segment with a block shape with the following size (32, 32, 1). Let data be a (float *) pointing to global memory and let data be 128 byte aligned (so data % 128 == 0).

Consider each of the following access patterns.

(a) data[threadIdx.x + blockSize.x * threadIdx.y] = 1.0;

Is this write coalesced? How many 128 byte cache lines does this write to?

(b) data[threadIdx.y + blockSize.y * threadIdx.x] = 1.0;

Is this write coalesced? How many 128 byte cache lines does this write to?

(c) data[1 + threadIdx.x + blockSize.x * threadIdx.y] = 1.0;

Is this write coalesced? How many 128 byte cache lines does this write to?

Please write a code to demonstrate the answer to the questions a, b and c.