Reduction

1. Use of reduction algorithms

Reduction algorithms have several uses, they are often used in data classification algorithms, (for example Nearest Neighbour Rule), dimensionality reduction (which can be used for data visualization, noise reduction, cluster analysis).

2. Workload of each thread

Each thread will at first have to write to shared memory (array used to calculate partial sum). Then in the loop, after each iteration, half of current threads will become idle (each thread writes to and reads from shared memory at least one time, one thread will perform log(n) iterations, where n is number of elements handled by current block). Then we use first thread to write partial sum to global memory (1 read and 1 write).

If by operation we mean writing to or reading from memory then:

Minimum number of operations: θ(1)

Maximum number of operations: θ(log(n))

Number of synchronizations: floor(log(blockDim.x)) + 1

3. Possible optimisations to the code

Original loop:

```
for (int stride=1; stride <= blockDim.x; stride *=2)
{
        __syncthreads();
        if (th % stride == 0)
                partSum[2*th] += partSum[2*th + stride];
}
```

Optimization by reversing the indexing, replacing very inefficient modulus operator and using each active thread to sum corresponding element from second, ‘inactive’, part of the array. Example of this optimisation:

```
for (int i=block_size; i > 0; i /= 2)
{
        __syncthreads();
        if (th < i)
                partSum[th] += partSum[th + i];
}
```

Optimization by reducing amount of shared memory used by half and skipping first loop iteration by adding elements when they are loaded from global memory.

```
if (th+start < data_size)
{
        if (th+start+block_size < data_size)
                partSum[th] = input[start+th] + input[start+th+block_size];
        else
                partSum[th] = input[start+th];
}
else
{
        partSum[th] = 0;
}
```

```
for (int i=block_size/2; i > 0; i /= 2)
{
        __syncthreads();
        if (th < i)
                partSum[th] += partSum[th + i];
}
```

4. There is no need to use atomic operations in this program, this algorithm is built in a way that prevents any thread from trying to access memory used by other threads.

Michał Orlewski