# Introduction to CUDA and OpenCL

Łukasz Gut, Grzegorz Litarowicz

Lab 09, 07.05.2020

## Introduction

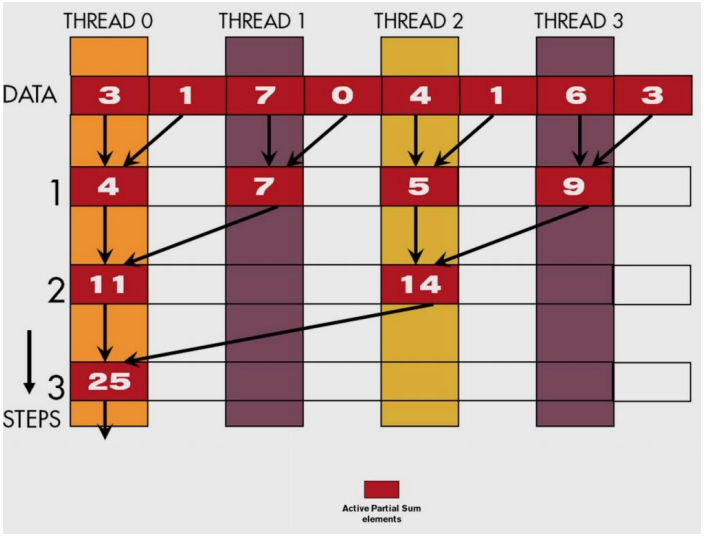
During ninth laboratories, we had to implement both naive and optimized parallel vector sum algorithm using a reduction technique.

## Reduction

The reduction is a parallel computation algorithm for vector summation which allows us to perform summation in $T(n) = log(n)$ time.

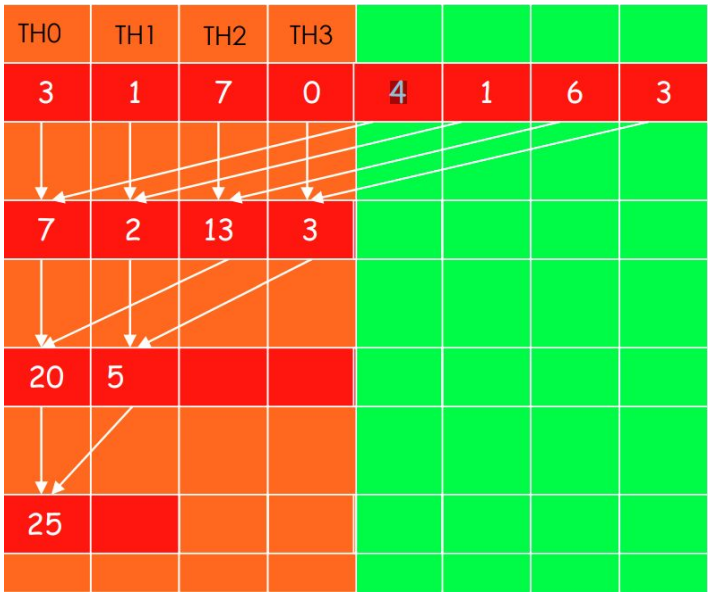
Each thread basically performs the addition on some chunk of data and saves it at thread id's index. In our implementation, the size of this chunk is 2. After each step, we halve the number of threads being used and then proceed.

The idea of this algorithm is presented in the image below.



So instead of iterating through the whole vector of numbers we pair them together and sum each pair separately. In our first approach, each thread is responsible for an even index of the vector. After each step of the algorithm half of the threads are no longer needed because they point to the number that is already taken to the final sum. The down sight of this approach is that working threads are far away from each other.

The idea of fixing this is to change indexing to put working threads close to each other. This approach is presented in the image below.




So we match all working threads with the first half of the table and sum each of them with one from the other half. In the next step, we treat the first half as the whole table and do exactly the same thing up to the point when there is only one element in the table and it contains our result.

## Implementation

Below we provide the implementation details regarding the naive and optimized versions of kernels.

**Naive implementation**

```
__global__ void reduce_naive(int* input, int size)
{
    __shared__ int partialSum[2 * THREADS_PER_BLOCK];

    partialSum[threadIdx.x] = 0;
    __syncthreads();

    std::size_t t = threadIdx.x;
    std::size_t start = 2 * blockIdx.x * blockDim.x;

    if (t < size && start < size) {
        partialSum[t] = input[start + t];
        partialSum[blockDim.x + t] = input[start + blockDim.x + t];

        for (std::size_t stride = 1; stride <= blockDim.x; stride *= 2) {
            __syncthreads();
            if (t % stride == 0) {
                partialSum[2 * t] += partialSum[2 * t + stride];
             }
        }
    }

    if (t == 0) {
        input[blockIdx.x] = partialSum[0];
    }
}
```

This implementation is fairly straight-forward. Its biggest flaw is the usage of the % (modulo) operator and the fact that threads diverge over time.

Thread divergence is kind of bad because of the “warp” mechanism. Warps are hardware’s smallest execution unit. In order to optimize kernel performance, we want to ensure that most of the threads that belong to warp are not idle. However, in the case of naive implementation, it is not a thing. When stride becomes larger very few threads will be active in a single block.

Therefore they will not belong to a single warp, which implies that we will need many warps for a tiny number of active threads.

**Optimized implementation**

```
__global__ void reduce_optimized(int* input, std::size_t size)
{
    __shared__ int partialSum[THREADS_PER_BLOCK];

    partialSum[threadIdx.x] = 0;
    __syncthreads();

    int t_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (t_id < size) {
        partialSum[threadIdx.x] = input[t_id];
        __syncthreads();

        for (int s = blockDim.x / 2; s > 0; s /= 2) {
            int index = threadIdx.x;
            if (index < s) {
                partialSum[index] += partialSum[index + s];
            }

            __syncthreads();
        }

        if (threadIdx.x == 0) {
            input[blockIdx.x] = partialSum[0];
        }
    }
}
```

The optimized solution removes both problems that were present in the previous implementation. We got rid of the % (modulo) operator as well as we will now use consecutive threads in order to perform calculations. What is more, this implementation also addresses the shared memory bank conflict.

## Benchmarks

Below we provide benchmarks of only kernel execution time and whole program execution time. By measuring the "whole program" execution time we mean timing of:

- Allocation
- Data copying to GPU
- Kernel launch
- Data copying back to CPU
- Fetching final result

Table 1. Only kernel execution time

| Naive [ms] | Optimized [ms] |
|---|---|
| 4.2383 | 4.0980m |

Table 2. Whole program execution time

| Naive [ms] | Optimized [ms] | CPU [ms] |
|---|---|---|
| 30.1 | 28.683 | 17.177 |

After many labs, this comes as no surprise - 1D problems are easily vectorizable on the CPU side. The real bottleneck of GPUs is data movement from host to device and vice versa and it cannot be overcome with such a simple problem like vector summation.

What is more, we only did single-threaded CPU implementation. This could be parallelized on the CPU to gain even better performance. Thread launches, even though they require context switch, are still much cheaper than copying the data to and from the device.

## Conclusion

As we see in the results, the naive kernel is about 4% slower than the optimized one which alone seems like a good result but when we take in to account whole time of execution with the copying of memory to the GPU, it is a minor improvement.

But as we see even an optimized version is much slower than a simple for loop on the CPU which is not the best way for big inputs and could be optimized by using a parallel version of this algorithm.