Report 7
Reduction algorithms.

Paweł Lipiór, Michał Żoczek

December 19, 2019

1 Code

The program focuses on testing reduction algorithms performance in comparison to naive approach. To achieve this, two kernel functions were used: first calculates sum of vector elements simply adding each value one by one, and the second one utilizes reduction algorithm.

1.1 Naive approach

```
__global__ void NaiveTotalKernel(VectorType *input, VectorType *output,
    int size)
{
       if(BLOCK_SIZE * blockIdx.x + threadIdx.x < size)
           atomicAdd(output, input[BLOCK_SIZE * blockIdx.x +
                threadIdx.x]);
}
```

This part is pretty straightforward - using `atomicAdd` all array elements all summed into `output`. Atomic functions guarantee that threads won't be accessing memory at the same time. The `VectorType` is defined type of array. In this case double-precision floating-point number was used as single-precision rounding errors were too big.

1.2 Using reduction algorithm

```
 __global__ void ReductionTotalKernel(VectorType *input, VectorType
     *output, int size)
{
    //Creating shared memory array
   __shared__ VectorType partialSum[2*BLOCK_SIZE];

   unsigned int thread{threadIdx.x};
   unsigned int start{2 * blockIdx.x * BLOCK_SIZE};

    //Copying data from input into shared memory
   if (start + thread < size)
      partialSum[thread] = input[start+thread];
   else
```

```
    partialSum[thread] = 0;

  if (start + BLOCK_SIZE + thread < size)
    partialSum[BLOCK_SIZE + thread] = input[start + BLOCK_SIZE +
        thread];
  else
    partialSum[BLOCK_SIZE + thread] = 0;

   //Calculating sum in each block
  for (int stride{BLOCK_SIZE}; stride >= 1; stride >>= 1)
  {
    __syncthreads();
    if (thread < stride)
      partialSum[thread] += partialSum[thread + stride];
  }

  //Saving sum of each block into output
  if (thread == 0)
  {
    atomicAdd(output, partialSum[0]);
  }
}
```

This approach is little more advanced. The use of reduction removes greatest bottleneck of previous take, which was the fact that writing to one place in memory must be done sequentially. Use of shared memory allows threads to access different fields in array in every iteration without visible loss of performance.

The code can be split into four parts:

1. Creating shared memory array:
   The array has size of two blocks as in first iteration each thread will be adding two elements.

2. Copying data from input into shared memory:
   As each thread will be accessing the same fields from array representing vector elements multiple times, it's more efficient to use shared memory.

3. Calculating sum of each block:
   Each block is summed using reduction algorithm: on every iteration each active thread adds two elements, reducing size of vector by two times.

4. Saving sum of each block into output:
   First thread of each block saves the result of summation from previous step into `output` using atomic function.

2 Performance tests

After code all necessary functionalities we were ready to test performance of these three implemented solutions. Using `nvprof` tool we were able to get time of execution for all kernels.To get time of execution for CPU single threaded function we have used `gettimeofday` function for C language. We decided to test the performance for a vector with length with successive powers of two. All collected data are shown in pictures 1, 2, 3:

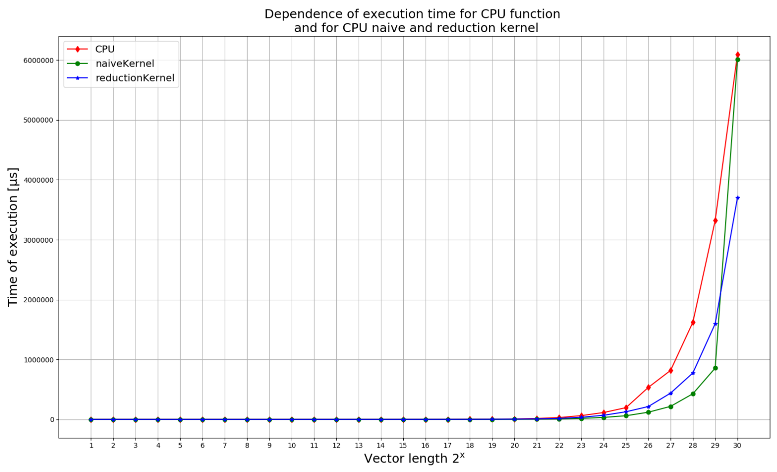

Figure 1: Dependence of execution time.

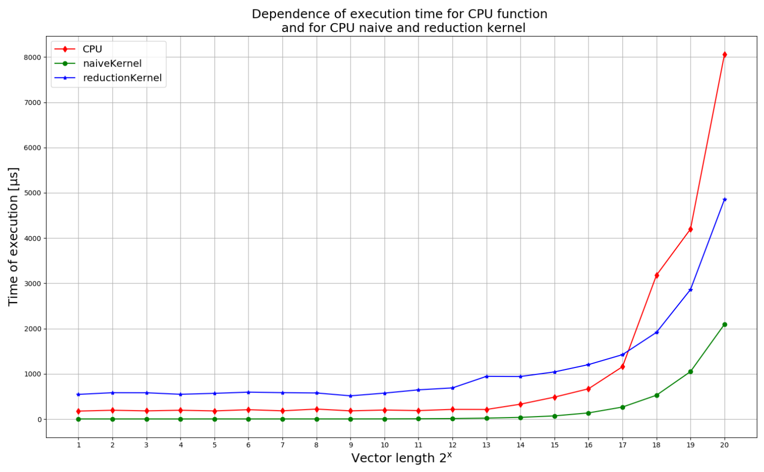

Figure 2: Dependence of execution time.

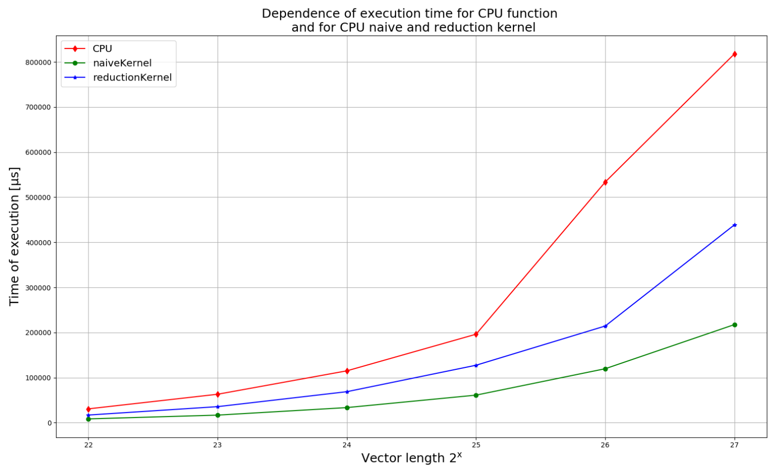

Figure 3: Dependence of execution time.

Based on data presented on pictures 1, 2 and 3 we can conclude that 'Naive' kernel implementation is the best solution for vectors length below 2^{29} elements. Only after exceeding that limit using 'Reduction' kernel implementation makes sense and that solution reaches better results. From the picture 2 it follows that CPU function is the second best solution but only below 2^{17} elements of vector. After that limit CPU reaches the worst results from all three tested solutions. Summarizing solution based on CUDA environment are very efficient in that kind of summation task. However using 'Reduction' kernel implementation with 'SharedMemory' are useful only for very large data structure. In other cases 'Naive' implementation works better. Moreover for not so big data structure CPU and GPU performance in summarizing elements is quite simmilar.

Table 1: Table of all collected data

x: 2^x	CPU μs	Navie μs	Reduction μs
1	177.0	3.808	546.5
2	196.0	3.808	583.7
3	182.0	3.744	582.2
4	196.0	3.456	548.4
5	181.0	4.352	569.5
6	208.0	3.776	595.8
7	183.0	4.096	584.7
8	222.0	4.288	577.9
9	182.0	4.16	514.3
10	201.0	5.344	574.7
11	188.0	7.616	646.7
12	215.0	12.32	689.7
13	213.0	21.82	945.1
14	330.0	38.17	941.8
15	486.0	71.04	1043.0
16	670.0	136.8	1204.0
17	1158.0	266.6	1424.0
18	3180.0	528.2	1917.0
19	4199.0	1050.0	2863.0
20	8061.0	2096.0	4855.0
21	15767.0	4186.0	8771.0
22	30637.0	8370.0	16900.0
23	63059.0	16730.0	35530.0
24	114992.0	33440.0	68730.0
25	196198.0	61100.0	127400.0
26	533889.0	119700.0	214400.0
27	818099.0	217600.0	438900.0
28	1622500.0	428300.0	775600.0
29	3320700.0	856700.0	1597300.0
30	6096300.0	6011000.0	3703900.0

All collected data are also available to get from table 1: