INSTITUTO SUPERIOR TÉCNICO

Parallel and Heterogeneous Computing Systems

Assignment 2

85229 – Rui Cavalheiro
93179 – Santiago Quintas
94030 – Luís Canas

Group 11
Shift LS03

October 21, 2022

1. CUDA SAXPY

The objective of this function was to execute the Ax + Y to given vectors using Cuda. We start by creating three pointers to float called device_x, device_y, device_result and allocating memory for N elements to each of them using cudaMalloc() function.

We then copy the X and Y arrays to the devices using cudaMemcpy(host to device)

We call the saxpy_kernel function indicating the number of threads per block and the number of blocks. this functions uses a thread to make each flop (aX+Y) and to store it in the device_result.

Lastly we copy the result from the device_result to the resultarray using once again cudaMemcpy() only now in device to host mode.

Before comparing this new implementation with the one from the previous lab class, we first remember that 20M value array ran in 20.331 ms. This number, when compared to our Kernel duration row in table 2, that shows the results for our new implementation when running for 20M value arrays also.

We notice that we got significant improvements for all tests and even achieved a speed up of more than 13 for the timing test 3.

	Speedup	Bandwidth (GB/s)	GFLOPS
Serial	-	14.441	1.938
ISPC only	1.11	16.072	2.157
ISPC + tasks	1	14.443	1.938

Tab. 1 – Results of saxpy program previous work.

	Timing test 1	Timing test 2	Timing test 3
Effective BW by CUDA saxpy	49.764 ms	49.264 ms	45.596 ms
Kernel duration	5.533 ms	5.496 ms	1.539 ms
Bandwidth	4.492 GB/s	4.537 GB/s	4.902 GB/s

Tab. 2 – Results of saxpy program.

2. CUDA Parallel Prefix-Sum

2.1. Exclusive-Scan

The objective of this function is to compute a set of operations between positions of a given array described in fig. 1, the problem is divided in 2 parts, the up sweep and down sweep phases, each phase has 2 for cycles, the outer loop goes through the rows represented in the figure below and for every iteration of the outer loop we have a inner loop going through the different positions of the array computing the wanted values.

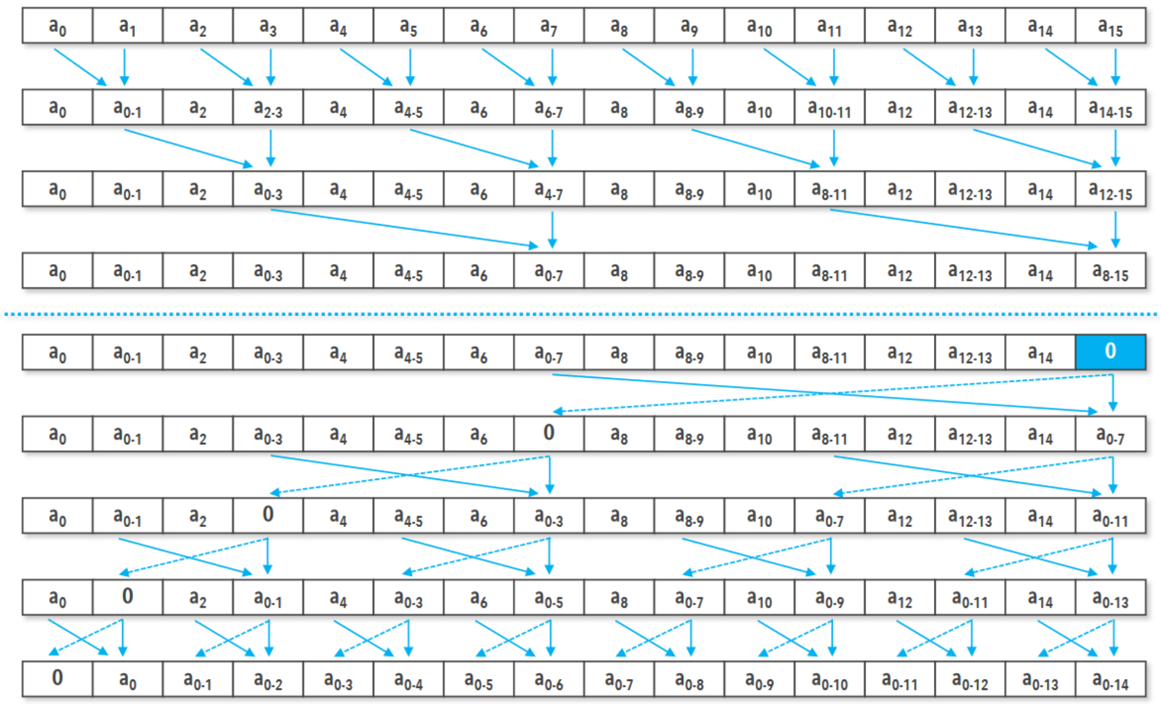

Fig. 1 – Computation of the array

Our first approach to increase the performance of our program was to add parallelism to the inner loop of both phases (up sweep and down sweep), this way we can do the operations in every iteration of the outer loop at the same time, there are no data conflicts because each operation is independent from the result of the other ones.

For the up sweep phase our kernel does the operation, it is launched across multiple threads but only a certain amount are used, the rest skip the operation and immediately shut down.

In the down sweep phase something similar happens, the only difference is the operation in itself.

The results obtained for the run time can be seen in the Tab.3

N	Ref Timings	Exp Timings
1000000	2.957 ms	2.760 ms
10000000	13.755 ms	12.798 ms
20000000	27.106 ms	25.515 ms
40000000	53.822 ms	49.574 ms

Tab. 3 – Results of exclusive_scan program.

Our first implementation launched the same amount of threads for every outer loop iteration, this is not very optimal because the number of working threads decreases every iteration of the outer loop (or increase if we are in the down sweep phase), our first step in optimizing the program was to control the number of threads launched in every iteration, this number is correlated to the number of operations made in that loop. The Tab.3 represents the values with this optimization, the times we achieved were slightly better than the reference timings even though there is great room for improvement.

2.2. Find-Repeats

This function has the purpose of finding the repeated values in a given array, more specifically, the repeated values in a consecutive way.

First we initialize a mask vector to 0, we do this in a parallel way by launching multiple threads using the kernel 'init()'. Secondly we go through the given array of numbers using the function 'find_pairs()' and check which numbers have the same value as the one next to them, for the positions where this is true we place the value 1 in the mask vector in that same position. After this we run the function 'exclusive_scan()' (explained in 2.1.) with the mask vector as the input array, by doing this we are able to find the number of 1's in the mask. The last position of the new array created by 'exclusive_scan()' tells us the final sum and we save it to a variable returned by the function. Finally the 'device_output' is then prepared by executing 'index_pairs()', this function places the index of each pair of the mask in a new vector.

N	Ref Timings	Exp Timings
1000000	3.715 ms	185.260 ms
10000000	21.905 ms	1739.624 ms
20000000	32.239 ms	3504.216 ms
40000000	63.599 ms	6973.697 ms

Tab. 4 – Results of find_repeats program.

We got significantly worse results then what we expected, they are two orders of magnitude worse than the Reference timings.

This is due to a bad implementation of a function where instead of using all the computational power available to us, we just use one thread to run over an entire array of millions of elements.

We tested the timing for the rest of the code and it showed an improvement in comparison to the ref timings, leading us to conclude that if we managed to optimize this last function of the code we would produce results that were positive.

Bellow is an example of an optimized alternative that we developed that produced good timing lower than the ref timings.

```
global void index_pair(int A,intB,int N, int count)
{
    int i = threadIdx.x + blockDim.x blockIdx.x;
    if (i<N-1)
        if (A[i]==1){
            B[atomicAdd(count,1)]=i;
        }
}
```

However the problem with this code is that the B[N] array we create ends up being out of order because the threads don't place the indexes in a specific order, they do it in a ”first come, first served” fashion.

The solution to this problem would be to sort the output array after the function runs (we just need to order it from lowest to highest value) using quicksort. Due to the Cuda machines being down, we weren't able to implement this solution in time.