Remark: the results in this paper are obtained running remotely on the **aomaster** node of the lab. In all the improvements of the code that we made the final value of checkSum is always -1820001304576.0000000.

1 Baseline

These are the execution results of the `loops2.c`, already with the restrict options and the data copy avoided. We see a total execution time of around 400 milliseconds, and that the most of the time is spent in the serial execution of loops 4 and 5.

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	51.49%	201.10ms	1	201.10ms	201.10ms	201.10ms	loop_4_41_gpu
	47.89%	187.04ms	1	187.04ms	187.04ms	187.04ms	loop_5_48_gpu
	0.33%	1.2972ms	4	324.29us	1.4400us	647.06us	[CUDA memcpy HtoD]
	0.07%	289.58us	2	144.79us	144.23us	145.35us	loop_7_64_gpu
	0.06%	219.46us	1	219.46us	219.46us	219.46us	loop_0_6_gpu
	0.04%	157.25us	1	157.25us	157.25us	157.25us	loop_2_27_gpu
	0.03%	114.66us	1	114.66us	114.66us	114.66us	loop_6_55_gpu
	0.03%	113.54us	1	113.54us	113.54us	113.54us	loop_3_34_gpu
	0.03%	113.44us	1	113.44us	113.44us	113.44us	loop_1_20_gpu
	0.03%	113.35us	1	113.35us	113.35us	113.35us	loop_1_16_gpu
	0.01%	19.969us	2	9.9840us	9.6330us	10.336us	loop_7_65_gpu__red
	0.00%	5.6960us	2	2.8480us	2.8160us	2.8800us	[CUDA memcpy DtoH]

2 Parallelization of loop 4

To obtain the parallelization of loop 4 we created an auxiliary vector to store the initial values, so that the GPU threads can access it separately and in parallel.
We see that the total execution time is around 200 milliseconds, the half as before, and that the performance bottleneck of the program remains now loop 5. Indeed the execution of loop 4 takes now 0,115 milliseconds, 1700 times less than before.

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	98.10%	201.08ms	1	201.08ms	201.08ms	201.08ms	loop_5_51_gpu
	1.26%	2.5845ms	5	516.90us	1.7920us	860.55us	[CUDA memcpy HtoD]
	0.16%	325.68us	2	162.84us	160.58us	165.09us	loop_7_67_gpu
	0.11%	218.25us	1	218.25us	218.25us	218.25us	loop_0_6_gpu
	0.08%	157.99us	1	157.99us	157.99us	157.99us	loop_2_27_gpu
	0.06%	122.34us	1	122.34us	122.34us	122.34us	loop_6_58_gpu
	0.06%	115.30us	1	115.30us	115.30us	115.30us	loop_4_42_gpu
	0.06%	113.83us	1	113.83us	113.83us	113.83us	loop_1_20_gpu
	0.06%	113.70us	1	113.70us	113.70us	113.70us	loop_3_34_gpu
	0.06%	113.70us	1	113.70us	113.70us	113.70us	loop_1_16_gpu
	0.01%	21.185us	2	10.592us	10.144us	11.041us	loop_7_68_gpu__red
	0.00%	6.3680us	2	3.1840us	3.1680us	3.2000us	[CUDA memcpy DtoH]

3 Parallelization of loop 5

To parallelize the execution of loop 5 we can observe that, given the input vector $[x_0, x_1, \dots, x_n]$, the result is the output vector $[x_0, x_0 \cdot a, x_0 \cdot a^2, \dots, x_0 \cdot a^n] = x_0 \cdot [a^0, a^1, a^2, \dots, a^n]$. In particular we need a way to calculate the powers of a in a parallel way.
To do this we first taught to divide the vector in 3 parts of equal lenght (the last one can differ from the previouses in length, depending on the size of the vector) and to calculate the powers of a of the first part of the vector assigning to each thread in the GPU a position in the vector. After this we can assume that we have

calculated the values a, a^2, a^3, ... a^{n_1}.
Then we can store these values and use them to calculate the second part of the vector in a parallelized way: indeed the following third of the vector can be taught as $a^{n_1} \cdot [a, a^2, \dots, a^{n_1}]$, and this can be easily parallelized in the GPU after we have calculated and store the previous part. The result of the calculation is then stored again, and after that we can think we have calculated the values a, a^2, a^3, ... a^{n_2}.
Finally, to calculate the last part of the vector, we can think it as $a^{n_2} \cdot [a, a^2, \dots, a^{n_1}]$ (for simplicity of the explanation we assume that N is multiple of 3), and this is easy to parallelize in the GPU.
In this idea the calculations of the second and third parts of the vector should be very fast, since each thread in the GPU must execute only one operation, and needs only two data values in input. But to calculate the first part this is not a good strategy, since the threads which calculate the high powers of a need a lot more time than the threads that calculate the small powers of a.
To improve this idea we taught that we could make each thread in the GPU to execute only one operation each time is used in the following way: first we calculate and store $[a, a^2]$. Then we calculate $[a^3, a^4]$ as $a^2 \cdot [a, a^2]$ and store it. Now in the memory we have $[a, a^2, a^3, a^4]$, and hence we can calculate $[a^5, a^6, a^7, a^8]$ as $a^4 \cdot [a, a^2, a^3, a^4]$ and store it. And we can proceed in this way until all the powers of a we need are calculated. In this way the GPU is called k times, where $2^k \leq n < 2^{k+1}$, and each time it is called the number of used threads increases by a factor 2, and each thread must execute only one operation and receive two data values as an input.
Using this strategy the calculations should be very fast when the number of used threads in the GPU is big, and slow when this number is small: but since the number of threads increase exponentially the disadvantage of the first calculations should be reassorbed in the last ones.

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	98.51%	200.36ms	1	200.36ms	200.36ms	200.36ms	loop_4_41_gpu
	0.85%	1.7232ms	4	430.81us	1.7920us	859.81us	[CUDA memcpy HtoD]
	0.16%	320.88us	2	160.44us	159.69us	161.19us	loop_7_76_gpu
	0.11%	218.57us	1	218.57us	218.57us	218.57us	loop_0_6_gpu
	0.08%	158.41us	1	158.41us	158.41us	158.41us	loop_2_27_gpu
	0.06%	119.33us	1	119.33us	119.33us	119.33us	loop_6_67_gpu
	0.06%	117.96us	19	6.2080us	1.0240us	50.658us	loop_5_57_gpu
	0.06%	113.96us	1	113.96us	113.96us	113.96us	loop_1_20_gpu
	0.06%	113.57us	1	113.57us	113.57us	113.57us	loop_1_16_gpu
	0.06%	113.41us	1	113.41us	113.41us	113.41us	loop_3_34_gpu
	0.01%	21.025us	2	10.512us	10.496us	10.529us	loop_7_77_gpu__red
	0.00%	6.3360us	2	3.1680us	3.1680us	3.1680us	[CUDA memcpy DtoH]

We can see that the total execution time for the loop 5 is now 0.117 milliseconds, instead of the 187 milliseconds of the baseline version, 1600 times better.
It is interesting to see that the kernel `loop_5_57_ gpu` is called 19 times, with a maximum time of 50 microseconds and a minimum time of 1 microsecond. Accordingly to our hyphotesis the slow calls should correspond to the ones needed to calculate the small powers of a while the fast ones should calculate the high powers. The mean value of 6.2 microseconds confirm the reassorbement hyphotesis.

4 Loop fusion

We fuse together some loops, the ones from 0 to 3 and then the sixth and the seventh. The execution results are the following:

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	51.28%	201.08ms	1	201.08ms	201.08ms	201.08ms	loop_4_16_gpu
	48.17%	188.86ms	1	188.86ms	188.86ms	188.86ms	loop_5_23_gpu
	0.44%	1.7221ms	3	574.03us	2.0160us	860.58us	[CUDA memcpy HtoD]
	0.06%	218.79us	1	218.79us	218.79us	218.79us	loop_fusion_6_gpu
	0.05%	208.52us	1	208.52us	208.52us	208.52us	fuse_loop_67_31_gpu
	0.00%	11.072us	1	11.072us	11.072us	11.072us	fuse_loop_67_35_gpu__red
	0.00%	3.1360us	1	3.1360us	3.1360us	3.1360us	[CUDA memcpy DtoH]

The total execution time does not change significantly, since the loops we fuse together occupied less than 0.5% of the total execution time in the baseline version.
But, if we sum the time spent in computing these loops in the baseline version we obtain around 1.2

milliseconds, while the time used to compute these cycles after the fusion is around 0.44 milliseconds, which is more than two times faster.

5 Loop fusion and parallelization of loops 4 and 5

Here we make the loop fusion and the parallelization of loops 4 and 5 to run together.

```
           Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   78.85%  2.5818ms         4  645.45us  2.0800us  860.64us  [CUDA memcpy HtoD]
                    6.82%  223.27us         1  223.27us  223.27us  223.27us  fuse_loop_67_46_gpu
                    6.72%  220.01us         1  220.01us  220.01us  220.01us  loop_fusion_6_gpu
                    3.63%  118.85us        19  6.2550us  1.0240us  51.426us  loop_5_35_gpu
                    3.51%  115.05us         1  115.05us  115.05us  115.05us  loop_4_17_gpu
                    0.37%  12.001us         1  12.001us  12.001us  12.001us  fuse_loop_67_50_gpu__red
                    0.10%  3.2000us         1  3.2000us  3.2000us  3.2000us  [CUDA memcpy DtoH]
```

Their improvements sum up, as we can see by the performance results, so we can conclude that they act independently. Now the bottleneck becomes the copy of the data from the Host to the Device, which occupies now the 79% of the execution time.
The total execution time is now around 3 milliseconds, and is reduced of 133 times from the baseline version.