INSTITUTO SUPERIOR TÉCNICO

Parallel and Heterogeneous Computing Systems

Assignment 1

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

Group 11
Shift LS03

October 7, 2022

1. Parallel Fractal Generation Using Threads

1) The code developed in this question divided the work (two halves of the image) between 2 threads. The speedup obtained for view 1 is 2. 2) In order to extend our code to be able to use more threads we had to add code to the workerThreadStart() function. Our implementation consists of a conditional statement to make sure that when we are dividing the image in rows there are no lines left unassigned (last thread computes remaining lines in case of the division of total number of rows by number of threads is not an integer) and then a call to the function mandelbrotSerial() (like in the single thread runs) where we simply changed the parameters of the StartRow and totalRows.

We ran the code for two to eight threads for both views and noted the speedup results and the computed the following graph in the image 1:

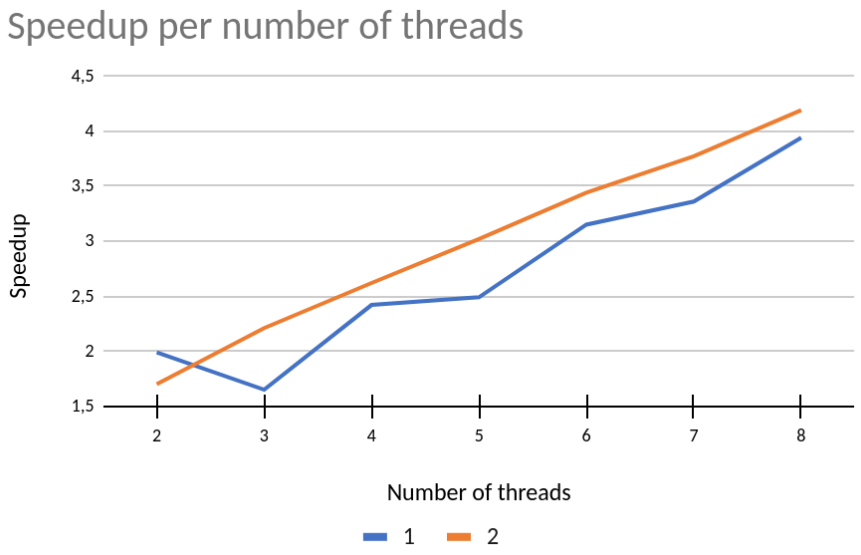

Fig. 1 – Speedup per number of threads

We noticed that the speedup in view one is very not linear, it is especially slower with an odd number of threads than what we'd at first expect. This is due to the data set for view 1 being substantially "heavier" in the middle resulting in one thread geting a block that required much more computations than what was required from the others.

3) To prove what was stated above (that one thread was taking a lot more time than the others due to the heavy workload) we ran the program measuring the time for the worst case observed which was view 1 with 3 threads and got the following results in the table 1:

Thread ID	Time (ms)
0	78.8
1	243.5
2	78.8

Tab. 1 – Time required by each thread.

4) In order to improve the speedup we had to make it so that the heavier computations are divided by all the threads so we opted to change the way we split the data-set to the threads. Instead of dividing in blocks of lines as we were doing previously we now opted to intercalate the lines that each thread does, so if there were 6 lines to compute and three threads the lines would be divided like this:

Thread 0
Thread 1
Thread 2
Thread 0
Thread 1
Thread 2

This makes it so that most difficult parts of the data-set are more evenly distributed by all the threads.

In doing so and using eight threads for both views we obtained the speedups shown in table 2.

	Speedup
View 1	7.38
View 2	7.21

Tab. 2 – Speedup with 8 threads for both views.

5) When running our newly improved code with 16 threads we noticed that there were no improvements in speedup, in fact we noticed a decrease. This is due to the maximum threads that the machine could handle at the same time is 8 (four cores with hyper-threading), and going past it only makes it ”waste” time managing additional threads that would only be waiting for the scheduler when the others are running.

2. Vectorizing Code using our “fake” SIMD Intrinsics

1) The most problematic part of the code to vectorize was the while cycle, to solve this we make the while cycle run until there are zero count positions with a number higher than zero, every loop we subtract 1 to the value of every count position and in every loop we multiply the numbers that still have a count higher than zero. The code can be seen below.

```
for (int i=0; i<N; i+=8) {
    __vfloat vX = _vload(&values[i]);
    __vint vY = _vload(&exponents[i]);
    __vfloat zeros = _vbcast(0.f);
    __vfloat ones = _vbcast(1.f);
    __vfloat nines = _bcast(9.999999f);

    __vbool mask = _veq(vY, zeros);
    __vfloat vResult = _vcopy(ones, mask);

    mask = _vnot(mask);

    vResult = _vcopy(vX, mask);
    __vint vCount = _vbcast(0.f);
    vCount = _vsub(vY, ones);

    mask = _vgt(vCount, zeros);
    int k = _vpopcnt(mask);
    while (k > 0) {
     vResult = _vmul(vResult, vX, mask);
     vCount = _vsub(vCount, ones);
     mask = _vgt(vCount, zeros);
     k = _vpopcnt(mask);
    }

    mask = _vgt(vResult, nines);
    vResult = _vcopy(nines, mask);

    _vstore(&output[i], vResult);
}
```

For a vector width of 8 we got a total of 81 vector instructions and a vector utilization of 80.71%.

2) If we change the width of the vector, the performance of the program also changes, in the Tab. 3

we see the values of execution time (cycles), total vector instructions used and vector utilization for vectors with dimension 4, 8 and 16.

Dimension	Execution time (cycles)	Total vector instructions	Vector utilization (%)
4	188	132	83.90
8	119	81	80.71
16	62	42	80.21

Tab. 3 – Performance of the vectorized code

As we increase the dimension the execution time decreases, this is expected because we are reducing the number of times the for loop has to run by using more memory per iteration. The total vector instructions also decrease for the same reason. We achieve a better speedup for higher dimensions since we are allowing the computation of more values at the same time.

The vector utilization tend to decrease as we increase the dimension, the explanation for this might be because with higher dimensions the vectorial groups are bigger which cause a longer waiting time for more positions when there is a number bigger than the others in the group. With smaller dimensions those big numbers have an effect on a smaller group of positions.

For example, let's imagine a vector y that looks something like the one below (1)

$$y = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 9] \tag{1}$$

Using vectors of dimension 16 the while cycle will perform 8 loops where the multiplication instruction will perform 8 times with only 1 of the positions being used, this corresponds to a lower vector utilization.

If instead we use vectors of size 2, there will be 7 while cycles with only 1 loop and full vector utilization in the multiplication instruction, only the final one will have 8 loops. This leads to a higher vector utilization.

3. Parallel Fractal Generation Using ISPC

3.1. ISPC Basics

Considering the ISCP compiler is configured to emit 8-wide AVX2 vector instructions, the maximum expected speedup is 8x. This could be achieved if the whole program had the same workload for the different data therefore dividing this data by 8 we would also divide the workload by and achieve the optimal speedup.

The experimental result obtained is 5.15x for the view 1. This difference between expected and experimental values is due to the fact that not the entire program benefits from the vectorization. Furthermore, the workload given to each ISPC instance is not balanced. Each instance is assigned blocks of rows of the image and parts of this image are harder to generate (white zones). Therefore, the the execution time of the ISPC program is slower due to the unbalance.

The speedup achieved for the view 1 is 5.15x and for the view 2 is 4.38x. These results confirm our hypothesis because to the fact that the view 1 has more intensive computation (more white zones). So the serial execution is slower, however the benefits of the ISPC implementation is bigger due to having a larger amount of workload with data which can be equally distributed during computation relative to view 2.

3.2. ISPC Tasks

In this question we altered the piece of code that controls number of rows per task (by changing the denominator to task number) and the number of launches created. Worth noting that the number of rows (height of image is 800) divided by the number of tasks must be and integer. The experimental results for this question are presented in Table 4 and Figure 2:

Number of Tasks	View 1 - Speedup (with tasks)	View 2 - Speedup(with tasks)
2	10.29x	7.39 x
4	12.91x	11.41x
8	19.63x	18.13x
16	33.70x	21.85x
32	34.14x	25.88x
100	35.58x	29.04x
400	35.68x	30.15x
800	35.79x	30.49x

Tab. 4 – Speedup of each both views for different number of tasks.

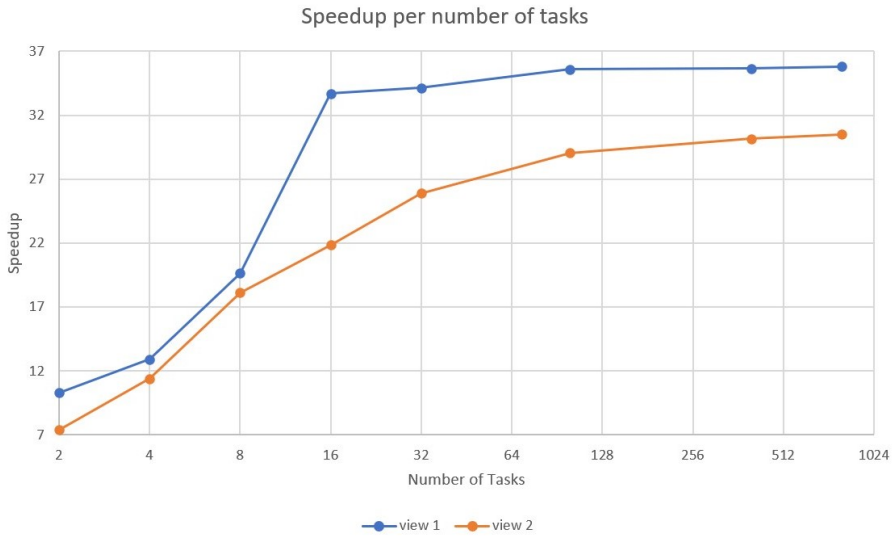

Fig. 2 – Speedup per number of threads

1) By analyzing the Table 4 we can observe that for 2 tasks the speedup of the view 1 using 2 tasks is 10.29x. Remembering the value of 5.15x for the program speedup without tasks we can infer that the speedup over the version of mandelbrot_ispc that does not partition that computation into tasks is 2.00x.

2) Using the view 1, to achieve a speedup of 30x we used 16 tasks (33.70x), if we continue increasing the number of tasks the speedup remains roughly the same with a small increase for this view. The architecture of the CUDA machine has 4 cores and each core 2 threads. Each thread will work on the tasks created by the compiler which has divided the workload evenly. So by continuing dividing the workload futher due to the increase of the number of threads the workload is better balanced achieving a higher speedup. If we observe Table 4 we can infer that with 800 tasks we divide the workload of the image optimally (maximum amount of data divided) and achieve a speedup over 30x on both views meaning this is the best number of tasks for this problem. This conclusion is supported by analyzing Figure 2 in which both views are converging to a speedup with the increase of the number of tasks. As we cannot go above 800 tasks, this value works best.

4. Iterative sqrt

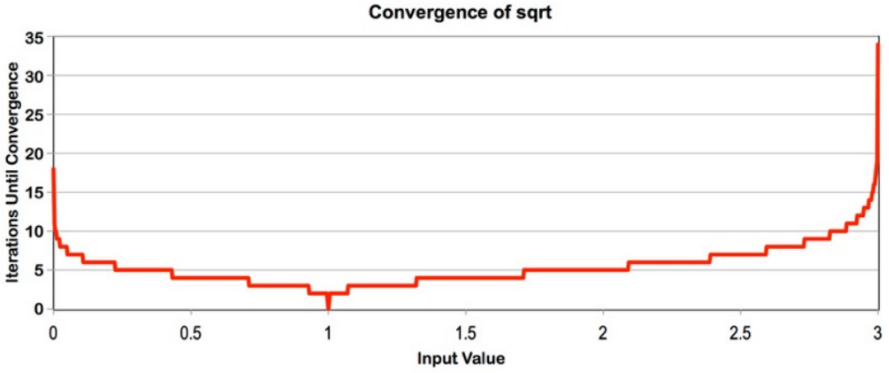

Fig. 3 – Convergence of the square roots

1) The obtained speedups for the base square root problem are logged in table 5

	Speedup
ISCP only	4.39
ISCP + tasks	31.74
Multi-core paralelization	7.23

Tab. 5 – Base speedups obtained for the square root problem.

The speedup obtained of the ISPC implementation is 4.39 when using a single CPU core and 31.74 when using all cores. Therefore, the speedup due to to SIMD implementation is 4.39 and the speedup due to multi-core parallelization is 7.23.

2) In order to improve the ISPC relative speedup and in accordance with figure 3, what we chose to do was make the values so that they are as close to three as possible (we accomplished this by setting the values of the array as 2.998 $\pm$0.001), and in doing so maximizing the number of iterations that each square root took to compute and thus obtaining the speedup values in table 6. By increasing the number of iterations we are also increasing the number of instructions which can benefit from the SIMD (we can parallelize the data in these instructions). So there will be an increase in the ISPC implementaion over the serial implementation.

	Speedup
ISPC only	7.56
ISPC + tasks	48.92
Multi-core paralelization	6.47

Tab. 6 – Best speedups obtained for the square root problem.

With this changes to the values we raised the SIMD speedup from 4.39 to 7.56 yet the multi-core speedup stays more or less the same. This is due to it already being close to the maximum benefit we can get from it which is around 8.

3) In order to decrease the ISPC relative speedup and in accordence with figure 3, what we chose to do was make the values so that they are as close to one as possible (we accomplished this by setting the values of the array as 1.000 $\pm$0.001), and in doing so minimizing the number of iterations that each square root took to compute and thus obtaining the speedup values in table 6. By minimizing the number of iterations we are also diminishing the number of instructions which can benefit from the SIMD (we can parallelize the data in these instructions). So there will be only a small increase in the ISPC implementation over the serial implementation.

	Speedup
ISPC only	2.40
ISPC + tasks	2.42
Multi-core paralelization	1.01

Tab. 7 – Worst speedups obtained for the square root problem.

With this changes to the values we decreased the SIMD speedup from 4.39 to 2.40 (from the initial version) and the multi-core speedup to around 1 so we basically got no advantage from using it whatsoever . This is due to each square root converging in a very short number of iterations and thus no having any advantage in distributing the computation cost to multiple tasks

5. BLAS saxpy

	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. 8 – Results of saxpy program.

The speedup achieved by running the program with tasks over without is 0.90x, there is no real speedup instead we actually notice a slight decrease in the performance of the program, the reason for this is the bottleneck we find in the bandwidth for fetching information from the memory, the program used in this problem is written below.

```
for (int i=0; i<N; i++) {
    result[i] = scale * X[i] + Y[i];
}
```

For every loop of the for cycle we do 3 memory operations, alot of information is being transmitted back and forth, even if we increase the capability of handling simultaneous operations there will be a limit to the capacity of information transmitted at the same time (bandwidth), the memory can't keep up with the amount of information being requested at the same time. Not only it does not help achieve a better performance, it can slightly decrease the performance of the program because the computer is doing unnecessary work, this is confirmed by the speedup described above.

To overcome this problem we would have to either reorganize the code in a way that makes memory fetching happen less often or request less data from memory, in our case we found it impossible to do these two so the only remaining solution to achieve a better speedup by running the program with tasks would be to increase the bandwidth via hardware upgrading.