Report - Assignment 2

Romain Jufer

October 27, 2017

1 Introduction

We compare the results of two different kinds of implementation and optimisation for the algorithm of heat transfer. The first uses tiling while the second uses loop-fusion. We run the algorithms on the SCITAS cluster in order to compare the performance of each optimisation.

2 Code

2.1 Tiling

To implement tiling, we have defined a variable *tile_size* whose value is the ratio between the length and the number of threads. Initially, the value was set so that we could reduce the number of cache misses while computing a tile. The idea was that if after computing a whole row we could reuse data to compute the next, then it will reduce the number of misses. In order to do this, the side of a tile must be set such that a whole row can be kept in the cache. However, it is clear that this value depends upon the cache size and the architecture, hence it is not possible to set it dynamically. Furthermore, due to $prefetching$, the runtime is generally better using the ratio mentionned previously (in average 1.1 times better).

The code consists of four nested loops. The two outermost loops iterate over the tiles while the two innermost iterate over a specific tile. We parallelize the two outermost in order to distribute tiles across threads. Furthermore, we use the directive *collapse*[1] to partition the iterations of both loops.

Inside the loop, we compute the value of a cell as specified by the handout. However, since the number of iterations is small, instead of using a loop to iterate over the neighbors, we have unrolled it to avoid the cost of branching instruction[2].

2.2 Loop-fusion

The four nested loops of the tiling version are replaced by only one loop which iterates over $length * length$. We then use the spatial locality of consecutive cells to distribute the work among threads. It is worth mentionning however, that in theory, in large arrays, this will induce a lot of cache misses since we use cells that are not consecutive in memory to compute the average. The tiling version must therefore have better execution time.

For the computation of the average, we also use the loop-unrolling as described above.

[1]https://software.intel.com/en-us/articles/openmp-loop-collapse-directive

[2]We have explicitly done the unrolling but the compiler might have done it automatically during the optimization phase.

3 Performance

The following table shows the runtime on SCITAS with the parameters : iterations = 500, length = 10000.

	Threads				
	1	2	4	8	16
Tiling	101.9s	48.53s	27.76s	18.41s	15.97s
Loop-Fusion	188.3s	105.4s	62.24s	40.76s	31.96s

We see that the tiling version runs faster than the loop-fusion, as expected. This might be explained by the cache misses caused by neighbouring cells which are not spatially close as mentionned previously. However, the parallelization in both cases yields a large speedup, i.e 6.38 times faster between 1 and 16 threads in the tiling case, and 5.89 times in the loop-fusion case.

We also tried to run a version of the algorithm without the optimisation, which means no loop-unrolling and two nested loops that iterate over the sides of the array on SCITAS. In every run, the job stops because the execution time is too long and this solution is therefore not usable.

We now consider the loop unrolling optimisation and compare the tiling version with and without the optimisation. Both programs are generated wihtout the flags for optimisation to be sure that the compiler cannot unroll the loop itself. We get the following results on SCITAS using the parameters : iterations = 500, length = 2000.

	Threads				
	1	2	4	8	16
Tiling with unrolling	42.97 s	21.93s	10.94s	6.03s	3.169s
Tiling without unrolling	57.39s	28.18s	14.68s	7.923s	4.117s

We see that the version with the unrolling runs approximatively 1.3 times faster than the one without, independantly of the number of threads. Hence the optimization is considerable and worth implementing.