Parallel Programming Lab Assignment: Diffusion Algorithm

Jan-Hendrik Niemann and Andreas Radke

October 27, 2017

1 Operations and Complexity

We chose the Diffusion Algorithm for our lab assignment. The main part of the algorithm is determined by the loops. We have a triple nested loop in *sum_values* (called twice) for summing all the values of the 3D-matrix, a triple nested loop in *init* for setting the initial values and the diffusion function with a four-fold nested loop which is the main part of the computation. Allocating the memory for the array with malloc is in regard to the other functions rather negligible.

Therefore we get for a 3D-matrix with the size of $N_x \times N_y \times N_z$ the number of operations:

$$2N_xN_yN_z + N_xN_yN_z + TN_xN_yN_z$$

where $T = \frac{N_xN_yN_z}{1000}$ (in the code it is named *count*). The first two summands are for bigger input sizes way smaller and hence can be neglected. This leads us to the complexity of $\mathcal{O}(N_x^2N_y^2N_z^2)$.

2 Base Version's Performance and Correctness

As we can see in Figure 1 the baseline version is very slow. For an input size of $N = 200$ the improved version is nearly 12.6 times faster. It can handle $569 \cdot 10^6$ operations per second. The baseline version scores only $45 \cdot 10^6$ operations per second.

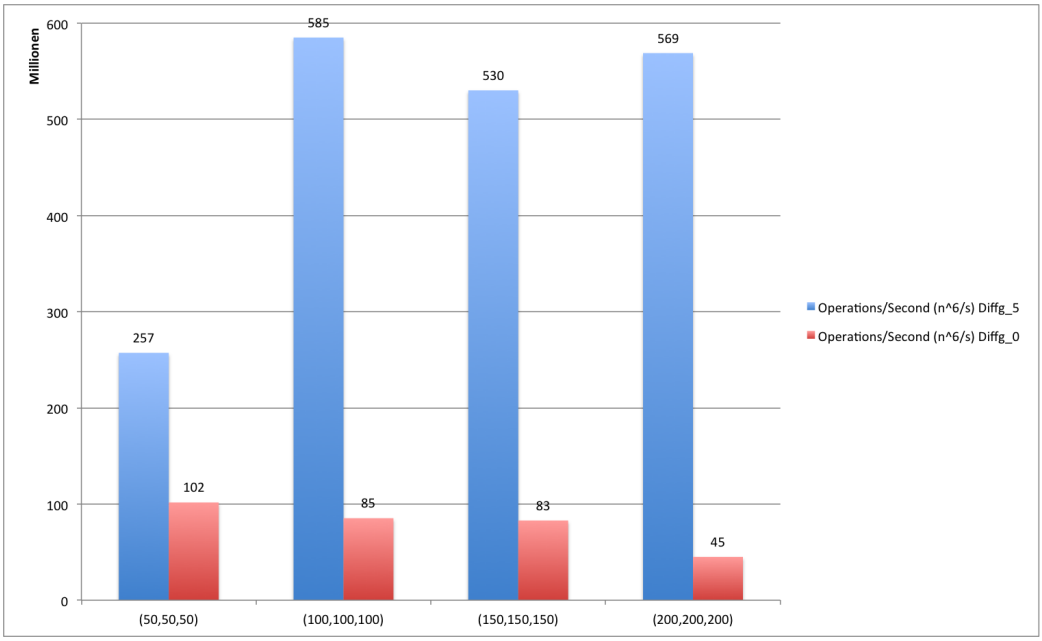

Figure 1: Million Operations per Second

3 Effect of Problem Size and Variable Type

As derived in Section 1 we can see in Figure 2 that the computation time indeed grows exponentially fast. Note that the y-axis has a logarithmic scale. We can see as well that the baseline code is slower than the improved one. It is independent of the input size.

Later we will optimize different stages (initialization, diffusion and error computation (*sum_value*) at which we will see that the initialization has no real impact on the performance (for slightly bigger input values).

Obviously the *float* version of our program fares better in terms of performance (compare Figure 3) than with *double*. Instructions per cycle and operations per second are around 10 to 20% better and for cycles, instructions, task-clock and execution time it takes only 80% of the *double* version. But the most glaring distinction is the decreased number of cache-misses (especially for higher input numbers) of the *float* program. With this one can clearly see where the smaller data types come in handy.

In Figure 1 we can see the operations per second in $[N^6/s]$. Note that the baseline code slows down for higher input size. In contrast the improved version "jumps" when doubling the input size from $N = 50$ to $N = 100$. There are almost no differences for higher input sizes. A small variation can be seen for $N = 150$.

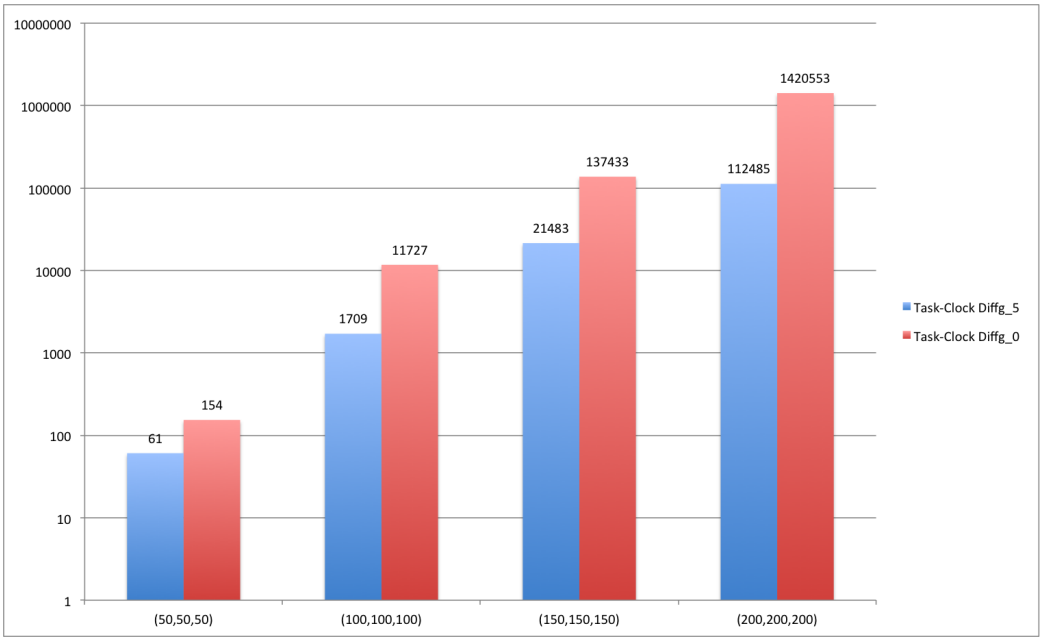

Figure 2: Task-Clock (msec)

4 Compilers

There is a slight difference between the GCC and the ICC compiler. However, in general there cannot be made a statement whether GCC or ICC is better. In Figure 4 we can see that ICC reaches a higher value for instructions per cycle than GCC. If we take a look at the improved version, we see that the GCC compiler reaches a higher value. The result looks similar if we compare values like instructions, cycles or time.

In Figure 5 and 6 one can see that the flag options bring a lot of performance improvements. The biggest jump is attained when activating *-O1* (*Diffg_0O1*). It can be further improved with *-O2* (*Diffg_0O2*) but between *-O2*, *-O3* and *-OFast* there is almost no further improvement.

5 Optimized Version without SIMD

The most important part for our non-SIMD optimization was to change the order of the loops in the different functions. We started with reordering the *init* method (*Diff_1*), followed by *sum_values* (*Diff_2*) and finally the *diffusion* part (*Diff_3*). Additionally restructured some *if*-conditions or variable assignments to outer loops because they had no dependence on the inner

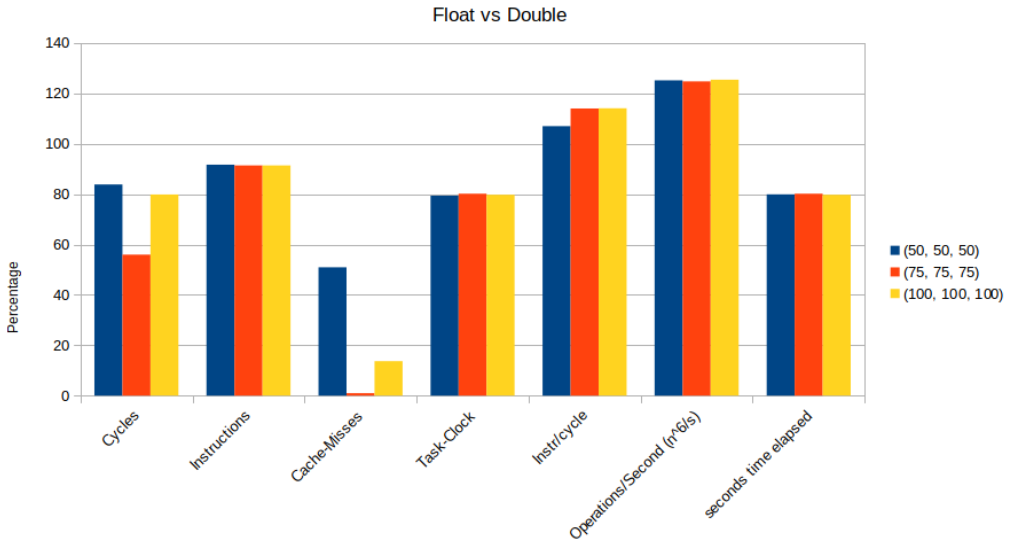

Figure 3: Comparison of float vs double compilation. 100% = double.

loop variables.

One can see in Figure 7 that the first two optimizations had barely an effect. But the change of the diffusion method with its four-fold loop leads to a execution time of roughly 10-12 % compared to the base program.

Although the number of instructions has not decreased vastly (around 83-85 %, compare Figure 8) in every other aspect *Diff_3* performs well better than *Diff_0* with only 10-11% cycles, task-clock and elapsed time and even less than 1% in cache-misses.

6 Optimized Version with SIMD

For the vectorization we had to adapt the *diffusion* method a bit more. Because of the specific boundary conditions the most inner loop could only go from $1 < z < N_z - 1$. Therefore computed the two boundary points separately before and after the loop. *Diff_4* contains the directive only for the inner *diffusion* loop whereas *diff_5* has additionally active SIMD directives for *init* and *sum_values*. Again one can see (compare Figure 8) that the latter two functions have an insignificant role. But overall the SIMD instruction is a big improvement again. Especially the relative part of instructions (compared to *Diff_0*) shrunk from 84-85% to 19-25%. Equally the stats in cycles, task-clock and elapsed time improved further.

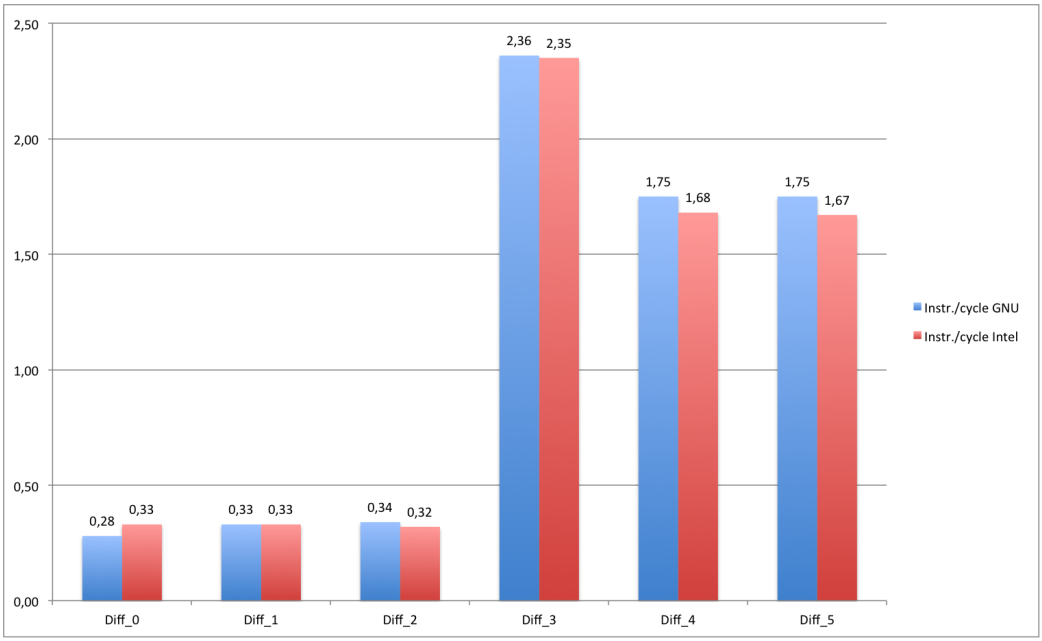

Figure 4: Instructions per cycle

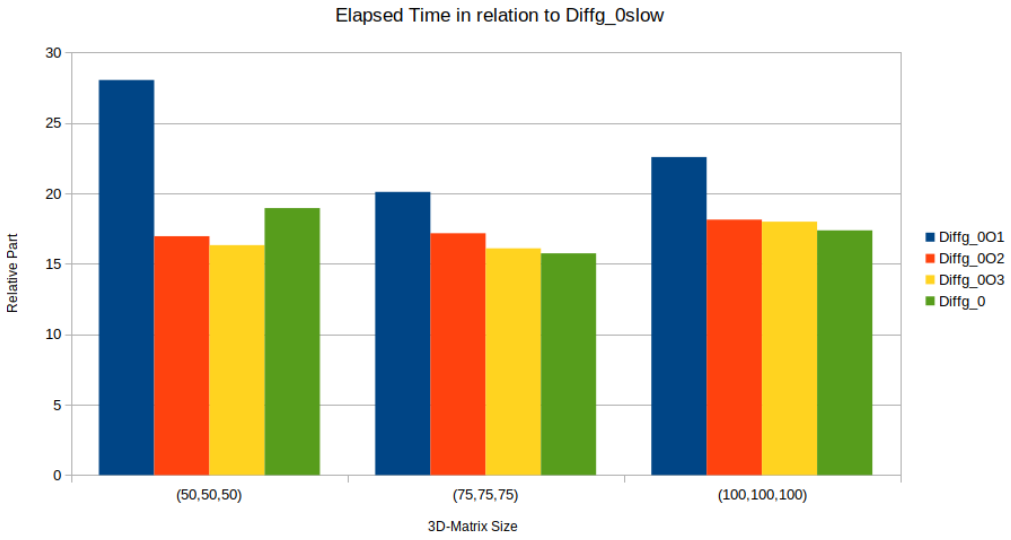

Figure 5: Relation of elapsed Time of different Optimization Flags compared to no Optimization at all.

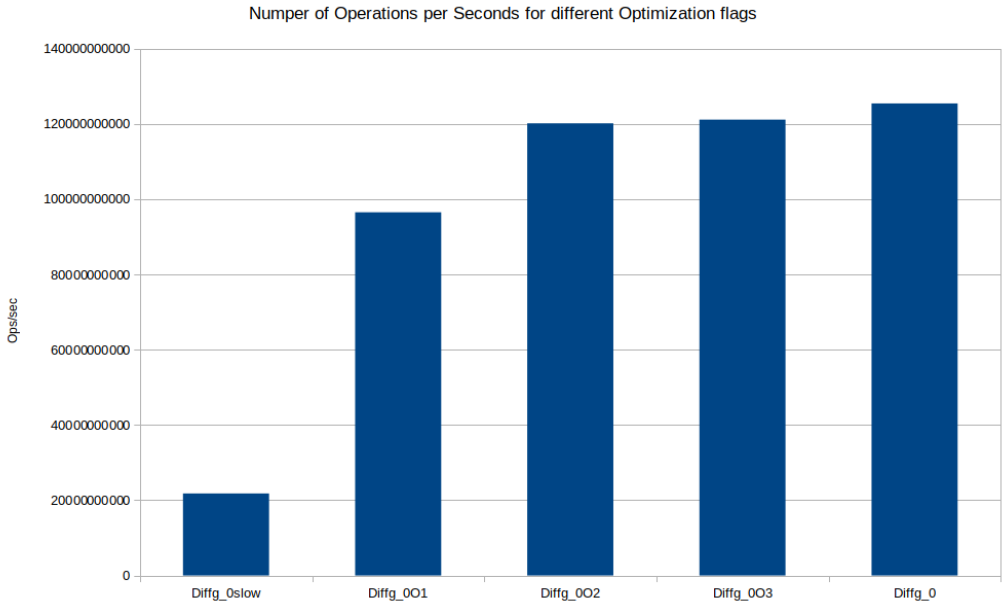

Figure 6: Development of number of Operations per second for several flag options.

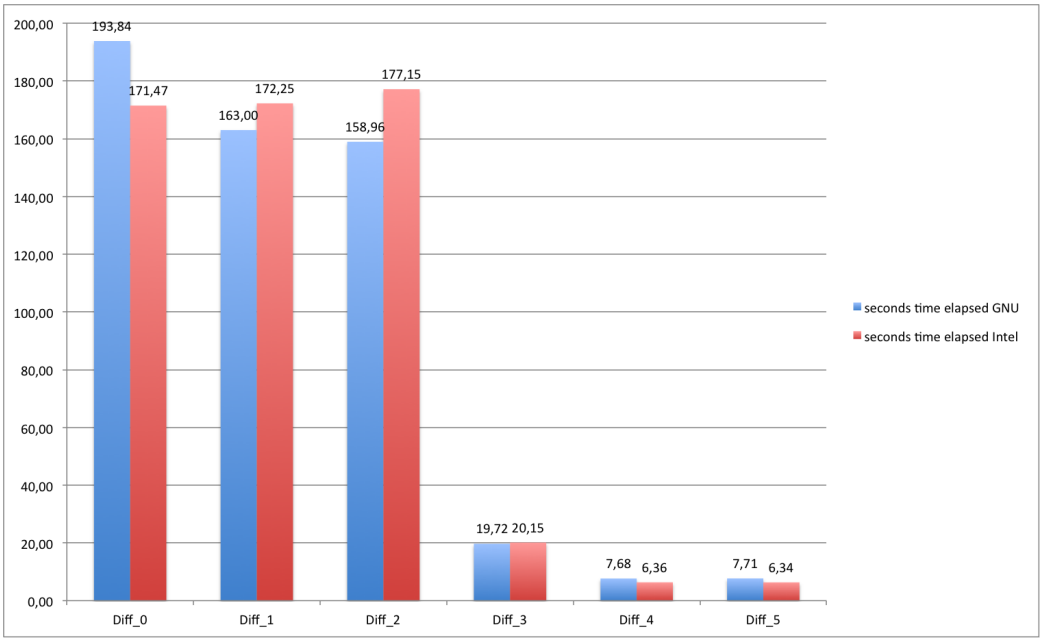

Figure 7: Elapsed Time in Seconds for the different Code Versions

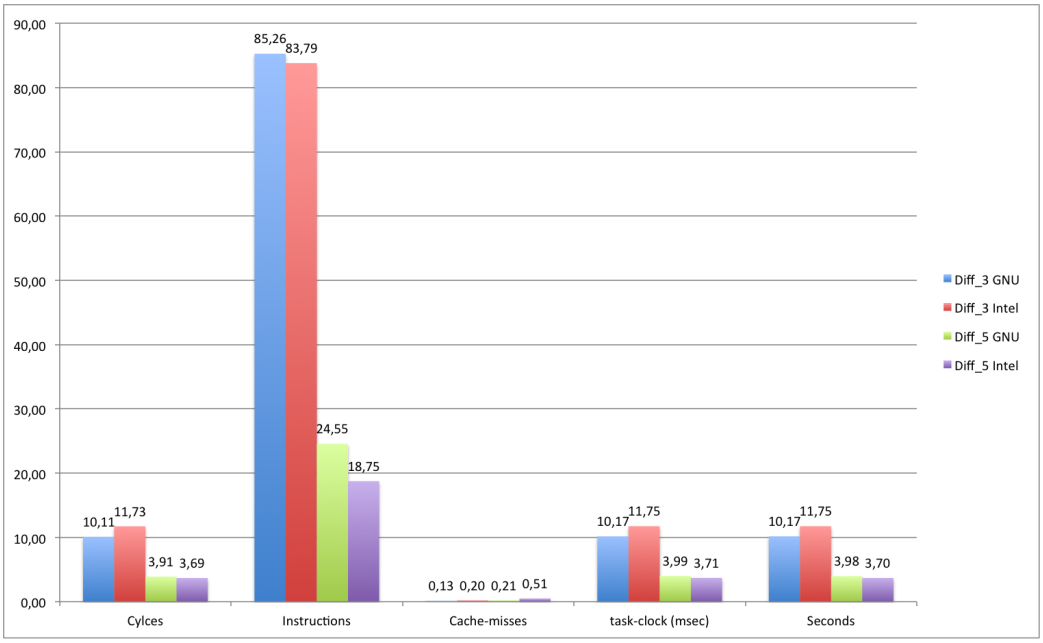

Figure 8: Relative part of Cycles, Instructions, Cache-misses, task-clock and Seconds for the non-SIMD-optimized *Diff_3* and the SIMD-*Diff_5*. The percentages are in relation to the base program.

7 Performance Bottlenecks