# Feedback for assignment (report based) for Group 51

In Assignment 4 we first create a quadtree structure and then use it to compute forces.
Possible tree construction:
- tree leafs contain information  about 0 or 1 star. For each four sibling nodes, the parent node represents a "big" star combining masses of all it children stars storing the center of mass coordinates and the computed mass of the "big" star (sum of masses of children nodes).
One should avoid allocation of extra memory except the memory for the tree node. For example, tree nodes should not allocate memory for storing arrays of stars a given quadrant.

Here is a list of optimizations and programs discussed in the Lectures 4-8 and Labs 1-4. This list does not mean you should try all these optimizations/programs, t goal is to give an overview of the optimizations/programs discussed in the course so far and which you can try to implement/use.

Optimizations discussed in your report (even ones which did not improve the preformance) are marked in the list as ☑.
NOTE:  optimization is reported here only if it was clear from the report that you used it (even if it did not improve the performance).

☐ use symmetry due to the Newton's third law (harder to implement in the assignment 4)

List of optimizations:
- ☐ cache usage optimization and improving data locality (nearby memory accesses in time should be to nearby locations in memory, lecture 5)
- ☑ skip brightness (read it to separate array since it does not participate in the calculations)

- ☐ do as little as possible inside loops (lecture 4)
- ☐ faster boolean evaluations (lab 2)
- ☐ inlining (lecture 4)
- ☑ strength reduction (use cheaper operations, lecture 4)
- ☐ modified bounds checking (two comparisons can be replaced by a single comparison, lecture 4)

- ☑ reduced number of functions calls

compiler optimizations flags:
- ☑ -O1, -O2, -O3
- ☐ -funroll-loops
- ☑ -march=native or -mtune
- ☑ -Ofast or  -ffast-math
- ☐ other
- ☐ usage of pure functions (in gcc __attribute__ ((pure)), lecture 5)
- ☐ const keyword (lecture 5)
- ☐ restrict keyword (lecture 5)
- ☑ improve branch prediction (remove if statements inside loops, simpler loop control condition, lecture 6)
- ☑ utilize ILP, get more independent instructions (manual loop unrolling, loop fusion, lecture 6)
- ☐ packed attribute in gcc for structures (lecture 7)
- ☐ using SIMD instructions explicitly in the code (SSE, AVX..., lecture 8)
- ☑ using auto-vectorization (lecture 8)

List of suggested programs:
debuggers:
- ☐ gdb debugger (breafly described how it was used)  (labs 1 and 2)
- ☐ valgrind memcheck (memory debugger)  (lab 3)
- ☐ other

profilers:
- ☑ gprof (lab 4)
- ☑ valgrind cachegrind (lecture 7)
- ☐ valgrind callgrind (mentioned in lecture 7)
- ☑ valgrind massif (lecture 7)
- ☐ other

Other:
- ☐ discussion of the assembly code (eg. explaining why some optimization works)

REPORT WRITING:
- ☑ reproducibility of results (included all needed information)
- ☑ the problem section (introduction)
- ☑ the solution section (algorithm, data structures, implementation details, etc)
- ☑ the performance and discussion section
- ☐ timing plots (execution time for various problem sizes) for O(N^2) algorithm
- ☑ timing plots (execution time for various problem sizes) for O(NlogN) algorithm
- ☑ obtained max_theta is reported
  - ☑ accuracy  table or plot for different theta values

Additional comments:

It is one of the best reports I read so far! Good work!

You reported that you did not see any timing difference by using various data structures. I guess in this assignment it does not play a very important role how do you store your stars in the array. We are mostly working the the tree structure, thus it is more important to optimize the tree construction and force calculation using this tree.

When you use valgrind cachegrind for checking cache misses, you can use cg_annotate (as you did for branch prediction part). Then you will see cache usage for each function and the output it is more readable. (You can also try cg_annotate --auto=yes.)

Describe which tests case you used for profiling, how many timesteps. Which compiler flags did you use?

I see that you discussed auto-vectorization. But this sentence "Then, since we did all of our tests on @fredholm.it.uu.se , we did not have the newest flags regardi SSE and AVX." is unclear. You do have SSE instruction set on fredholm.
If you want to try avx, you can use vitsippa.it.uu.se.
You can check using -fopt-info-vec (see Lab 4) if your code is vectorized.