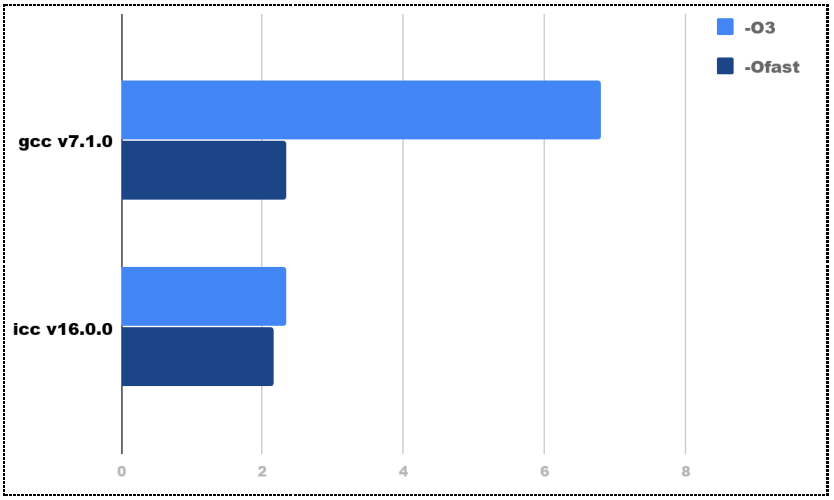
SERIAL OPTIMISATIONS **Name**: Lin Feng **ID**: rg18215

- **Compiler choice and flags**
  Generally speaking, newer versions of a compiler should perform better, and so shows the result.

Version	Runtime (1024*1024 per call of stencil)
gcc v4.4.7 (default)	41.03ms
gcc v7.1.0 (newest in available modules)	29.42ms

- **Compiler choice and flags**
  Some compilers could be more optimized than others for a specific architecture. I compared icc and gcc compilers of the available newest version (icc version 16.0.0, gcc version 7.1.0).
  Also, different combinations of optimization flags for the same compiler can greatly affect execution time. -O3 turns on all optimizations specified by -O2, also includes autovectorization. -Ofast with -ffast-math optimizes even more. Data shows that icc (version 16.0.0) with flag -Ofast compiler performs the best.



- **Data type**
  Accessing time of both double and float from cache could be very similar depending on the architecture. However, since more floats can be loaded by the CPU (4 floats or 2 doubles in SSE), algorithms that apply vectorization can have an extra boost in performance by switching to float type.

Data type	Runtime (total time of 1024*1024 100 iterations)
double	2.351021 s
float	3.828774 s

** Before vectorization*

Data type	Runtime (total time of 1024*1024 100 iterations)
double	0.236304 s
float	0.142978 s

** After vectorization*

- **Contiguous Access pattern**
  Accessing contiguous memory addresses is much more performant than non-contiguous access due to cache missing. In order to have contiguous memory access, the 2D matrix needs to be stored linearly in a 1D array row by row instead of column by column. Then, we can change the algorithm to increase the X direction iterator until the end before increasing the Y direction iterator (`i+j*nx` instead of `j+i*ny`).

```
Int nx=4, ny=4
for ( int j=0, j<ny, ++j) {
        for (int i=0, i<nx, ++i){
                printf( j+i*ny );
        }
}
Output : 0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15
```

* *Non-contiguous Access pattern.*

```
Int nx=4, ny=4
for ( int j=0, j<ny, ++j) {
        for (int i=0, i<nx, ++i){
                printf( i+j*nx );
        }
}
Output : 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
```

* *Contiguous Access pattern.*

Code	Runtime (total time of 4096*4096 100 iterations)
Non-contiguous Access pattern	93.766493 s
Contiguous Access pattern	9.386636 s

- **Vectorisation**

  Using ‘pragma ivdep’ and the flags -qopt-report=2 -qopt-report-phase=vec, we know that code between line 55~59 needs to be removed in order to support vectorization (img 1). Switch statements like if/else needs to be removed from the main loop (img 2). Therefore, we split the main loop into 3 different sections: Vertices, edges, and the inner image body where most of the optimizations will be applied.

```
remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below. Use level 5 report for details
remark #15346: vector dependence: assumed OUTPUT dependence between  line 55 and  line 59
```

* *image 1*

```
55          tmp_image[i+j*ny] = image[i+j*ny] * 3.0/5.0;
56          if (j > 0)    tmp_image[i+j*ny] += image[i  +(j-1)*ny] * 0.5/5.0;
57          if (j < nx-1) tmp_image[i+j*ny] += image[i  +(j+1)*ny] * 0.5/5.0;
58          if (i > 0)    tmp_image[i+j*ny] += image[i-1+j*ny] * 0.5/5.0;
59          if (i < ny-1) tmp_image[i+j*ny] += image[i+1+j*ny] * 0.5/5.0;
```

* *image 2*

Condition	Runtime (total time of 4096*4096 100 iterations)
Before optimizing the loop	16.910463 s
After optimizing the loop	2.414085 s

- **Blocked loop**

  Since we have contiguous access pattern but our algorithm still needs some non-contiguous access, blocked loop could enhance performance by partitioning the large image into smaller chunks that can stay in cache during many operations until it is not needed anymore. But it is hard to decide the size of the chunks since it depends on an accurate estimation of accessed array regions and the cache size of the machine [1]. Getting the size of the chunks wrong will slow down the progress.

- **Conclusion**

  With all data provided above, after optimizing loops, using icc v16.0.0 with flag -Ofast performs the best.

Parameter	Runtime
1024*1024 (100 iterations)	0.100827 s
4098*4098 (100 iterations)	2.415410 s
8000*8000 (100 iterations)	8.651464 s