C Programming and Performance Engineering

Lab Work - C Programming for CPU and Performance Engineering

October 27, 2017

Spyridoula Chrysikopoulou-Soldatou and Jeremy Williams

Diffusion Algorithm (3D stencil)

The selected application for the assignment is the Diffusion Algorithm.

For the measurement of the performance, the following was used, 1) Performance (stencils/second), 2) Coding Efficiency (stencils/instructions) and 3) Processor Throughput (IPC).

The Algorithmic Complexity of the application is given by the function Θ(iter*nx*ny*nz), where nx, ny and nz are the dimensions of the 3D matrix.

All the executions were realized through the remote connection to the lab.

Furthermore, the gcc compiler version 6.1.0, was used with the -Ofast optimization.

Base version of the program(diff.c)

Firstly, the base version of the program (diff.c) was executed.

The results are presented below:

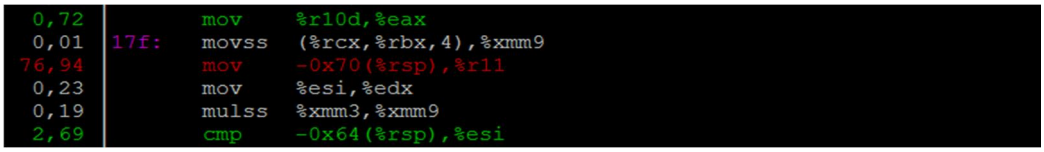
Cycles	612.132.116.750	**Performance**	0,02
Instructions	199.785.950.604	**IPC**	0,33
Cache-misses (M/sec)	57,165	**Coding efficiency**	22,01
Accuracy	-5,068750E+01		
Time (seconds)	**181,605**		

Afterwards, the <time.h> was included in the base code (diftime.c), in order to measure the time consumed by each of the three functions. The results after the execution are the following:

init() took 0.12 seconds to execute
sum_values() took 0.01 seconds to execute
diffusion() took 180.65 seconds to execute

Above shows that, the diffusion function consumes most of the time (via seconds to execute).

By using the perf command, we can also see that the diffusion function has a greater overhead percentage and specifically (after assembling the function) the mov instruction; which indicates that data are being copied from one register to another [1].



The results above shows part of the program (the diffusion function) that we will try to optimize.

Optimizations

Loop reordering (difreord.c)

The first optimization that was used to improve the performance of the program was the loop reordering. The best optimization was successful when all the three loops of the program were changed to the order x-y-z (instead of z-x-y that was in the baseline version).

The results are presented below:

Cycles	68.560.064.039	**Performance**	0,22
Instructions	168.673.077.452	**IPC**	2,46
Cache-misses (M/sec)	0,825	**Coding efficiency**	26,07
Accuracy	3,125000E-01		
Time (seconds)	**20,394**		

It is obvious that the elapsed time has been reduced significantly.

The loop reordering was successful, because the elements are now accessed in the order in which they are presented in the memory (first x, second y and third z) so, the memory access is better.

Loop unrolling

The second optimization is the loop unrolling [2,3]. Firstly it was done two times, and then four.

The results are presented below:

2 times (difunr2.c)

Cycles	64.889.698.852	**Performance**	0,23
Instructions	155.583.676.875	**IPC**	2,4
Cache-misses (M/sec)	0,825	**Coding efficiency**	28,27
Accuracy	2,812500E-01		
Time (seconds)	**19,321**		

4 times (difunr4.c)

Cycles	56.575.839.333	**Performance**	0,26
Instructions	143.514.085.119	**IPC**	2,54
Cache-misses (M/sec)	0,940	**Coding efficiency**	30,64
Accuracy	2,968750E-01		
Time (seconds)	**16,931**		

The loop unrolling was successful, because the number of iterations are reduced. More unrolling signifies less iterations.

Use of operators (difoper.c)

The next optimization that took place, was the use of operators in the diffusion function [3].

The results are presented below:

Cycles	56.297.069.408	**Performance**	0,26
Instructions	143.511.428.351	**IPC**	2,55
Cache-misses (M/sec)	0,933	**Coding efficiency**	30,64
Accuracy	2,968750E-01		
Time (seconds)	**16,769**		

The use of operators didn't change the results of the execution, because there is still a hypothesis that has to be checked.

Loop Vectorization (difvect.c)

In order to take advantage of the SIMD, the boundaries of the last loop in the diffusion function were moved outside of the loop. The vectorization was in that way successful.

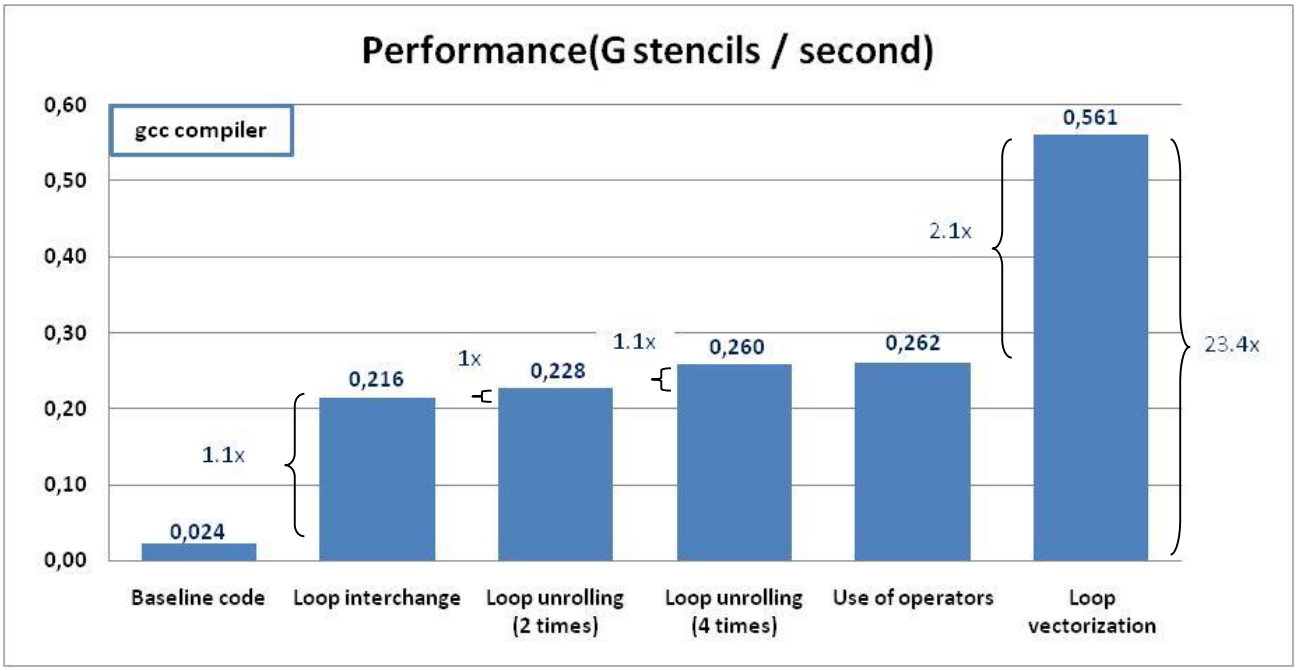
The results are presented below:

Cycles	25.939.590.871	**Performance**	0,56
Instructions	44.397.195.447	**IPC**	1,71
Cache-misses (M/sec)	3,777	**Coding efficiency**	99,05
Accuracy	2,849219E+02		
Time (seconds)	**7,846**		

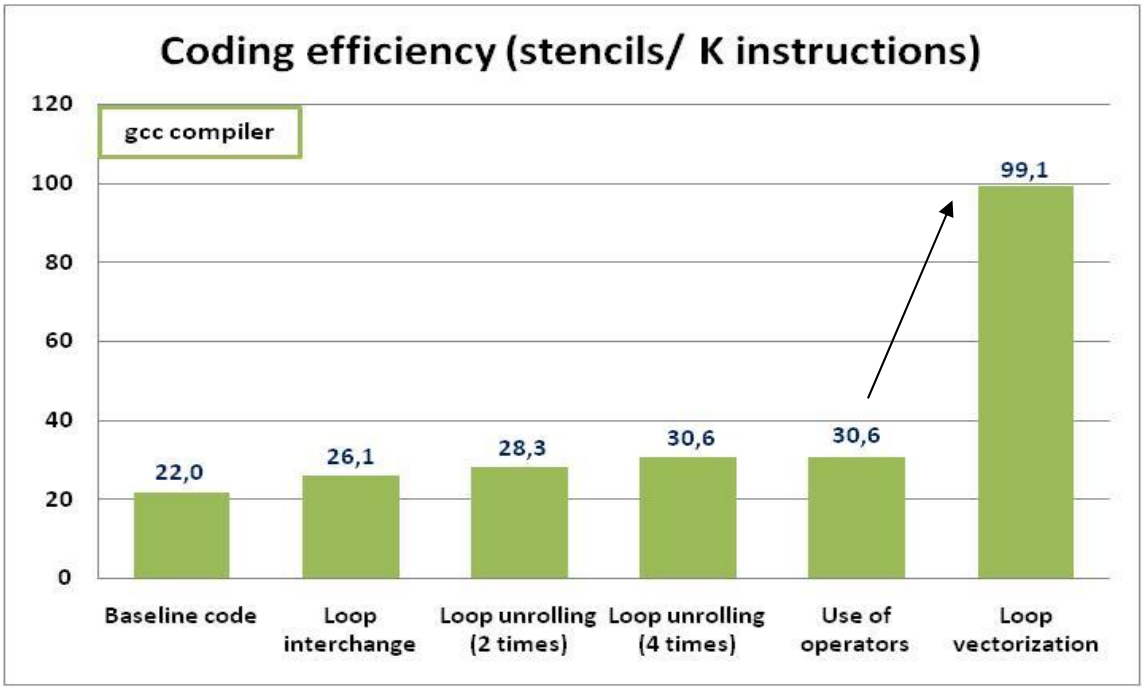
The appearance of the mulps instruction indicates the realization of the vectorization.

```
diffusion  /home/master/ppM/ppM-1-1/Escritorio/exercise1/dif13
  0,33          movups (%r10,%rax,1),%xmm8
  5,72          mulps  %xmm13,%xmm7
  0,18          mulps  %xmm12,%xmm8
  0,75          addps  %xmm8,%xmm7
  0,61          movups (%rbx,%rax,1),%xmm8
 10,02          mulps  %xmm11,%xmm8
  0,58          addps  %xmm8,%xmm7
  1,20          movups (%r12,%rax,1),%xmm8
```

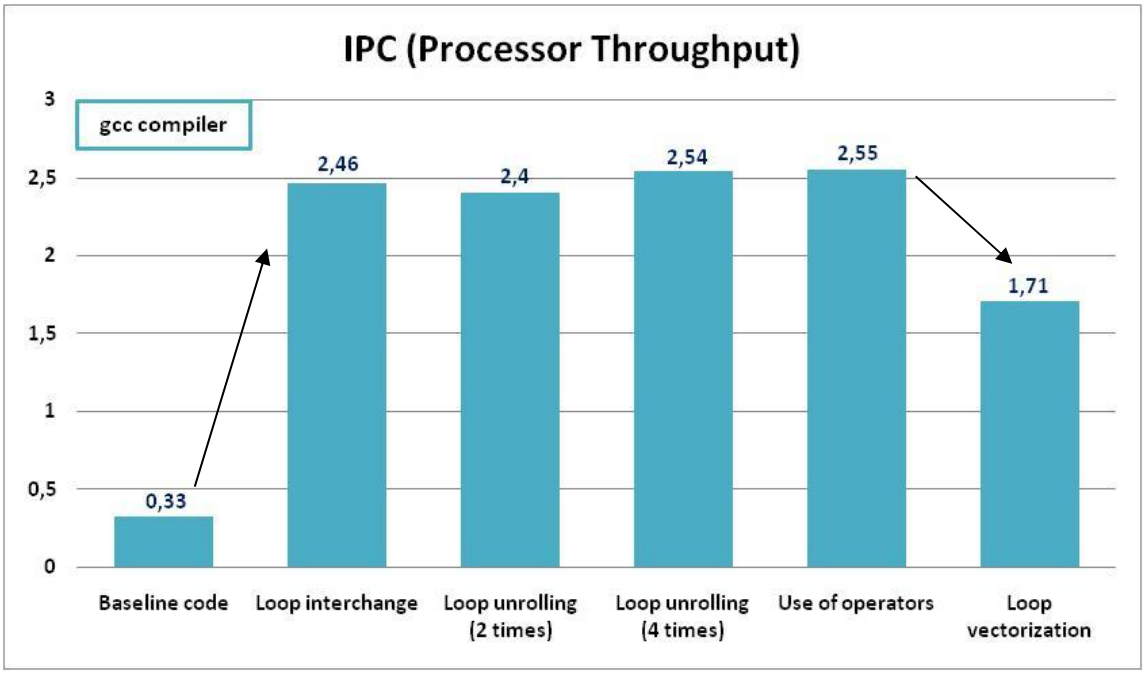
Below shows the graphs of Performance, IPC and Coding efficiency for all the optimizations.



The loop interchange resulted in a large improvement (speedup is 1.1x). The loop unrolling (2 times) has speedup the program by 1x, and the loop unrolling (4 times) by 1.1x. Also, the use of operators doesn't have an important effect on the performance, while the loop vectorization has speedup the program by 2.1x. The overall program improvement is 23.4x from the baseline code.



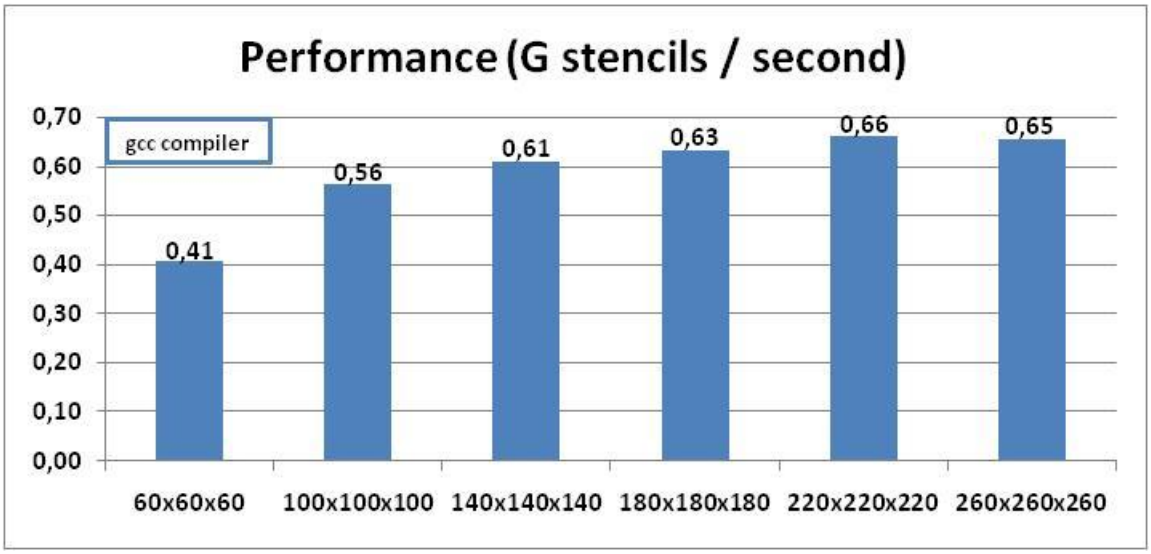
With the optimizations, the coding efficiency is increasing, because the same work is being done with less instructions. With the first three optimizations the instructions are decreasing with a similar manner (around 12M), while with the loop vectorization, the instructions have been decreased dramatically.



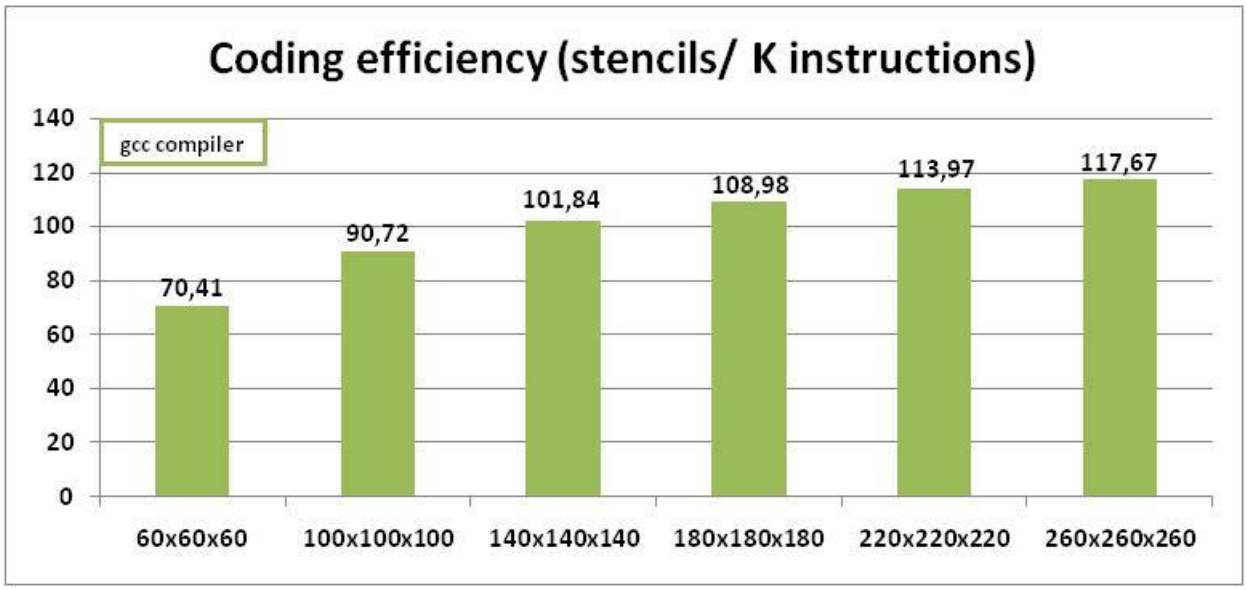
With the loop interchange, the loop unrolling and the use of operators, the processor efficiency has improved approximately in the same level. With reference to the loop vectorization, the IPC is lower, because the number of instructions have been decreased by three times.

Effect of the problem size to performance

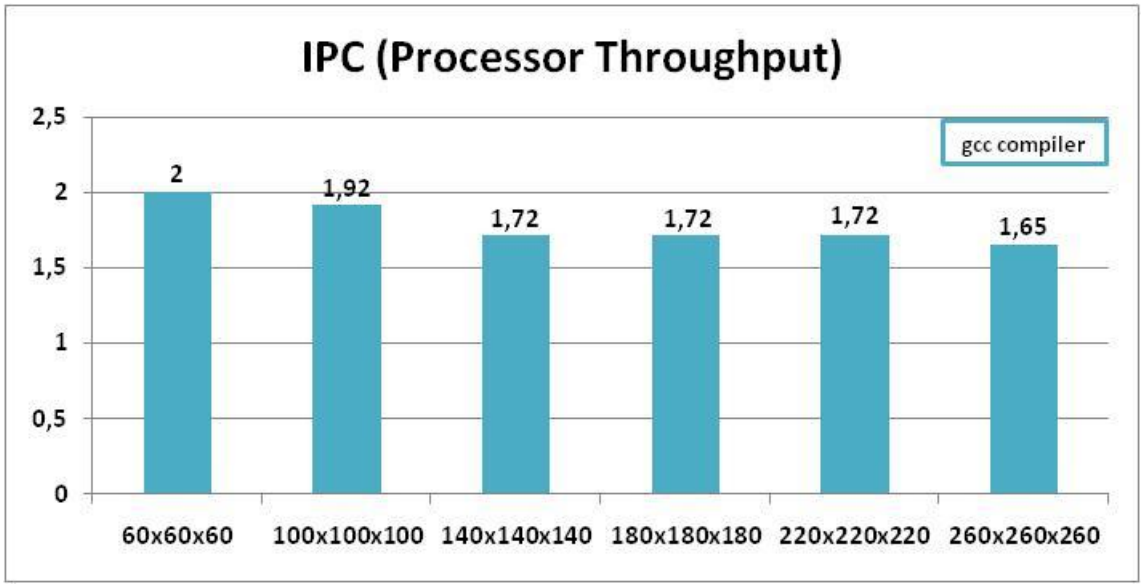
In order to check the effect of the problem size to the performance, was measured the performance for different dimension sizes. For this task was used the difvect.c code. The results are presented in the following graphs:



As it can be seen in the graph above, the problem size grows with the performance increasing.



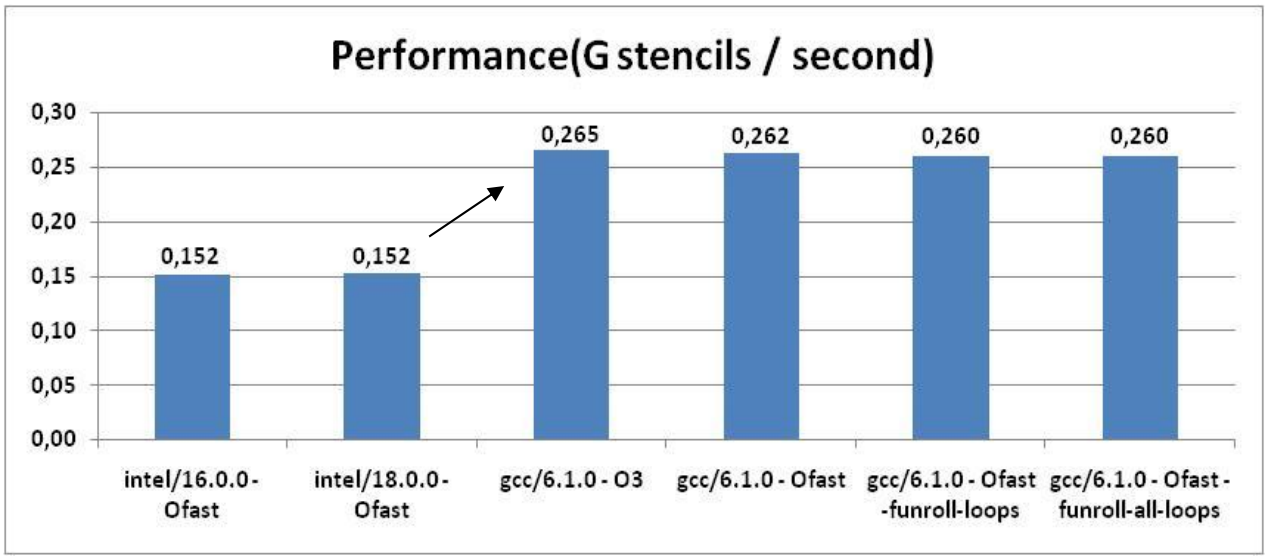
According to the graph, the coding efficiency is better for larger 3D meshes.



The Processor Throughput is being decreased as the size grows, because more accesses to DRAM, lowers the average IPC, because instructions have to wait longer.

Performance differences by different compilers

In the difoper.c code, various compiler flags and options have been used, and their performances are being presented below:



The two versions of the intel compiler have the lowest performance. The options for the gcc compiler -O3, -Ofast, and the flags -funroll-loops and funroll-all-loops don' t have a distinguished difference on the performance. However, the gcc complier, they are faster than the intel complier.

Time taken by the initialization stages and the main part of the algorithm

For this task the perf record and perf report commands have been used in difoper.c for different problem sizes. The most important overhead percentages were recorded; which are presented below:

Problem size: 20x20x20

```
Samples: 10  of event 'cycles', Event count (approx.): 3951124
Overhead  Command  Shared Object      Symbol
  31,59%  dif11    ld-2.17.so         [.] dl_main
  27,11%  dif11    libm-2.17.so       [.] __cos_sse2
  22,92%  dif11    [kernel.vmlinux]   [k] enqueue_task
  17,06%  dif11    [kernel.vmlinux]   [k] __kmalloc
   1,25%  dif11    [kernel.vmlinux]   [k] perf_event_aux
   0,07%  dif11    [kernel.vmlinux]   [k] native_write_msr_safe
```

Problem size: 60x60x60

```
Samples: 761  of event 'cycles', Event count (approx.): 623831615
Overhead  Command  Shared Object      Symbol
  94,99%  dif11    dif11              [.] diffusion
   2,93%  dif11    libm-2.17.so       [.] __cos_sse2
   0,64%  dif11    dif11              [.] init
   0,57%  dif11    [kernel.vmlinux]   [k] put_filp
   0,29%  dif11    dif11              [.] main
   0,13%  dif11    [kernel.vmlinux]   [k] account_entity_enqueue
```

Problem size: 100x100x100

```
Samples: 15K of event 'cycles', Event count (approx.): 12823572320
Overhead  Command  Shared Object      Symbol
  98,64%  dif11    dif11              [.] diffusion
   0,78%  dif11    libm-2.17.so       [.] __cos_sse2
   0,11%  dif11    dif11              [.] init
```

Problem size: 140x140x140

```
Samples: 115K of event 'cycles', Event count (approx.): 96650037677
Overhead  Command  Shared Object      Symbol
  99,28%  dif11    dif11              [.] diffusion
   0,28%  dif11    libm-2.17.so       [.] __cos_sse2
   0,04%  dif11    dif11              [.] init
```

Problem size: 200x200x200

```
Samples: 557K of event 'cycles', Event count (approx.): 447412323626
Overhead  Command  Shared Object      Symbol
  99,13%  dif11    dif11              [.] diffusion
   0,12%  dif11    libm-2.17.so       [.] __cos_sse2
   0,05%  dif11    [kernel.vmlinux]   [k] task_tick_fair
```

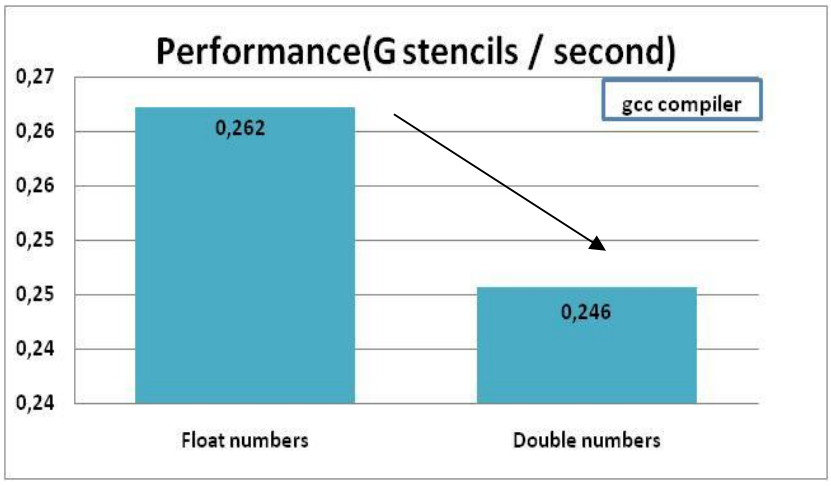
We can conclude that as the problem size grows, the initialization part is relatively less and less important than the main part of the program (the diffusion function).

Effect of the float and double numbers

In the difoper.c code, the float values are replace with double values (difdouble.c).

The results are presented below:

Cycles	59.866.332.914	**Performance**	0,25
Instructions	143.511.755.364	**IPC**	2,4
Cache-misses (M/sec)	1,729	**Coding efficiency**	30,64
Accuracy	-2.706656E-09		
Time (seconds)	**17,89681**		



While the Coding efficiency for the both is the same, the performance is decreasing when the values are double. Furthermore, the cache misses from 0,933 M/sec are being increased to 1,729 M/sec. The single precision values (float) have an accuracy of 6 decimal places, while the double precision values have an accuracy of 15 decimal places. And as a result, they are using more memory.

For this specific program, the values don't need to have an accuracy more than 6 decimal places. Therefore, for this reason, the single precision values are preferable.

References

1. https://www.recurse.com/blog/7-understanding-c-by-learning-assembly
2. http://icps.u-strasbg.fr/~bastoul/local_copies/lee.html
3. https://www.slideshare.net/EmertxeSlides/embedded-c-optimization-techniques