Performance improvement Laplace Equation Algorithm

Abstract

In this report we show the improvements obtained on the Laplace Equation Algorithm code provided using some well-known techniques.
The original code provided, once executed, shows these parameters:

```
Jacobi relaxation Calculation: 2048 x 2048 mesh, maximum of 100 iterations
Total Iterations:   100, ERROR: 0.079694, A[16][16]= 0.006917

 Performance counter stats for './original 2048 100':

   102.419.540.320      cycles                    #    3,373 GHz
    10.901.284.820      instructions              #    0,11  insns per cycle
     2.586.596.672      cache-misses              #   85,172 M/sec
      30368,984182      task-clock (msec)         #    0,998 CPUs utilized

      30,425325821 seconds time elapsed
```

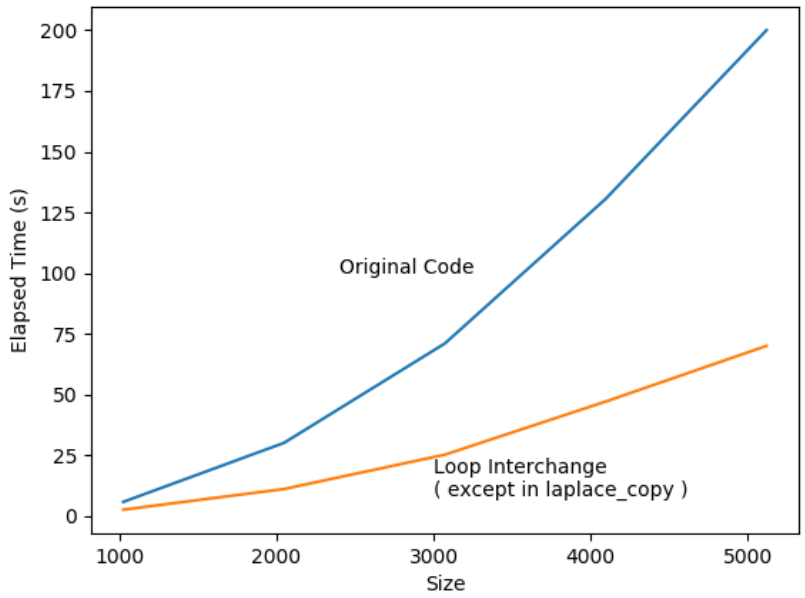
We can notice a very high elapsed time for a relative small size of the problem (dimension n of the matrix equal to 2048) and an IPC of 0,11, that indicates a poor exploitation of the processor's potential.
Furthermore, we have a big number of cache-misses that shows a not so clever access to the data and a bad exploitation of the *Data Locality* principle.
We now try to apply some techniques, as said before, and we will analyse our results.

Loop interchange (not in laplace_copy function)

```
float laplace_error (float *old, float *new, int n)
{
  int i, j;
  float error=0.0f;
  for ( j=1; j < n-1; j++ )
    for ( i=1; i < n-1; i++ )
      error = max_error( error, old[j*n+i], new[j*n+i] );
  return error;
}
void laplace_step(float *in, float *out, int n)
{
  int i, j;
  for ( j=1; j < n-1; j++ )
    for ( i=1; i < n-1; i++ )
      out[j*n+i]= stencil(in[j*n+i+1], in[j*n+i-1], in[(j-1)*n+i], in[(j+1)*n+i]);
}
```

```
Jacobi relaxation Calculation: 2048 x 2048 mesh, maximum of 100 iterations
Total Iterations:   100, ERROR: 0.079694, A[16][16]= 0.006917

 Performance counter stats for './noCopyInterchange 2048 100':

    36.588.036.834      cycles                    #    3,369 GHz
     5.742.883.813      instructions              #    0,16  insns per cycle
       849.066.770      cache-misses              #   78,173 M/sec
     10861,441926       task-clock (msec)         #    0,998 CPUs utilized

      10,883244960 seconds time elapsed
```

With loop interchange in ***laplace_step*** and ***laplace_error*** we improve the elapsed time of 20 seconds. This is due to the different way in which the matrix's elements are accessed. In the previous version, the matrix was visited by column, and since the elements of a general matrix are stored in the memory as vectors, for every inner cycle a jump in the memory was needed to reach the pointed element.
Indeed, exchanging the indexes is possible to access to adjacent elements in the inner cycles and this lead to safe lot of computational time reducing the number of cycles needed and the operations performed.
We have similar IPC because both number of instructions and cycles are reduced, and almost 2 million of cache-misses less because we access to adjacent cells of memory and the compiler can generate optimized assembly code for loading in the cache the data to use in the subsequent operations exploiting the locality principle.
This help to safe computational time because the cache memory is smaller than the main memory, and so it is much faster.
Here we show in a graph the comparison between original and improved code w.r.t. elapsed time and the problem's size.

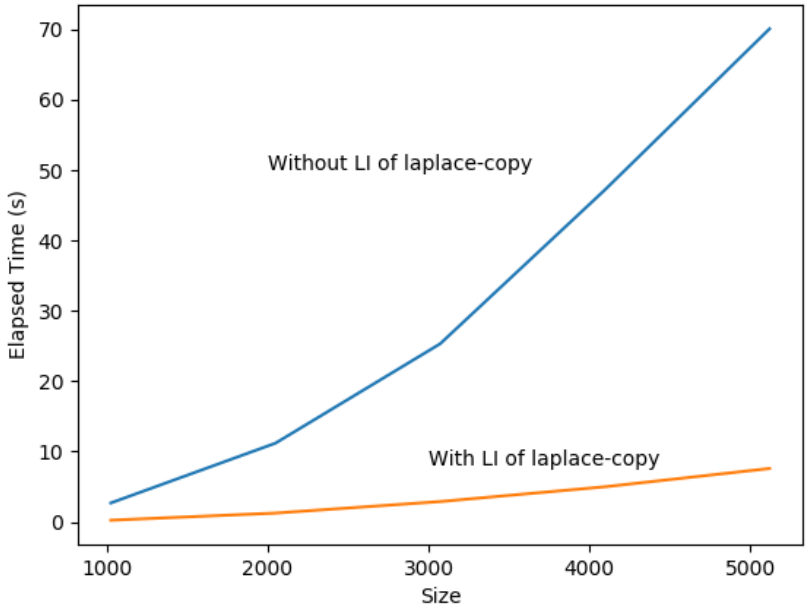


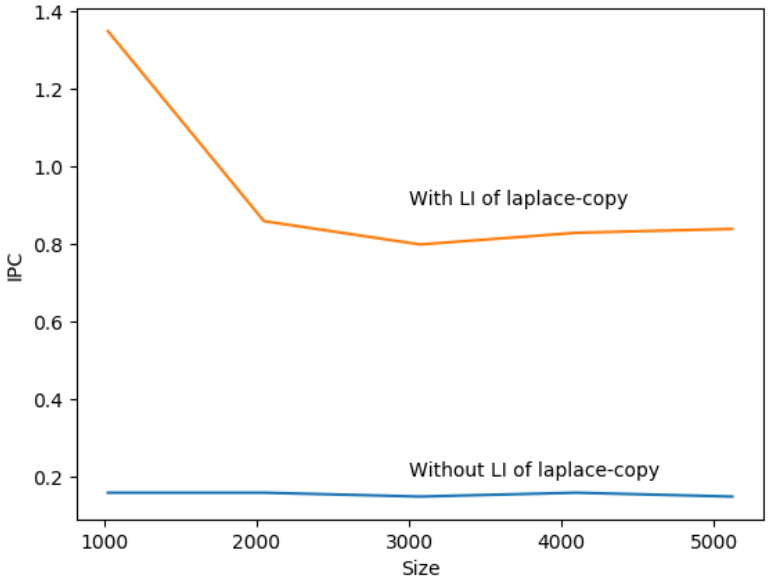
Loop interchange in laplace_copy function

```c
void laplace_copy(float *in, float *out, int n)
{
  int i, j;
  for ( j=1; j < n-1; j++ )
    for ( i=1; i < n-1; i++ )
      out[j*n+i]= in[j*n+i];
}
```

```
Jacobi relaxation Calculation: 2048 x 2048 mesh, maximum of 100 iterations
Total Iterations:   100, ERROR: 0.079694, A[16][16]= 0.006917

 Performance counter stats for './InterchangeOfAll 2048 100':

     4.049.412.506      cycles                    #    3,217 GHz
     3.468.557.889      instructions              #    0,86  insns per cycle
        36.462.147      cache-misses              #   28,965 M/sec
      1258,818020       task-clock (msec)         #    0,997 CPUs utilized

       1,262395696 seconds time elapsed
```

Exchanging also the index in the ***laplace_copy*** function we see a huge improvement of performance that carry our elapsed time from 10 seconds to approximately at 1,26 seconds. We can notice that the instructions are over than 2 billion less, due to the smarter data access, and the number of cycle fall dramatically from 36.6B to 4B. This lead to have a very good IPC of 0,86 and a very good percentage of the processor's potential. How we can expect, also the cache-misses are highly reduced from 849M to 36.5M exploiting properly the principle of locality loading cleverly adjacent elements in the various cache level.
Now we show the comparison between the previous code and the improved one w.r.t elapsed time and IPC using the subsequent graphs.





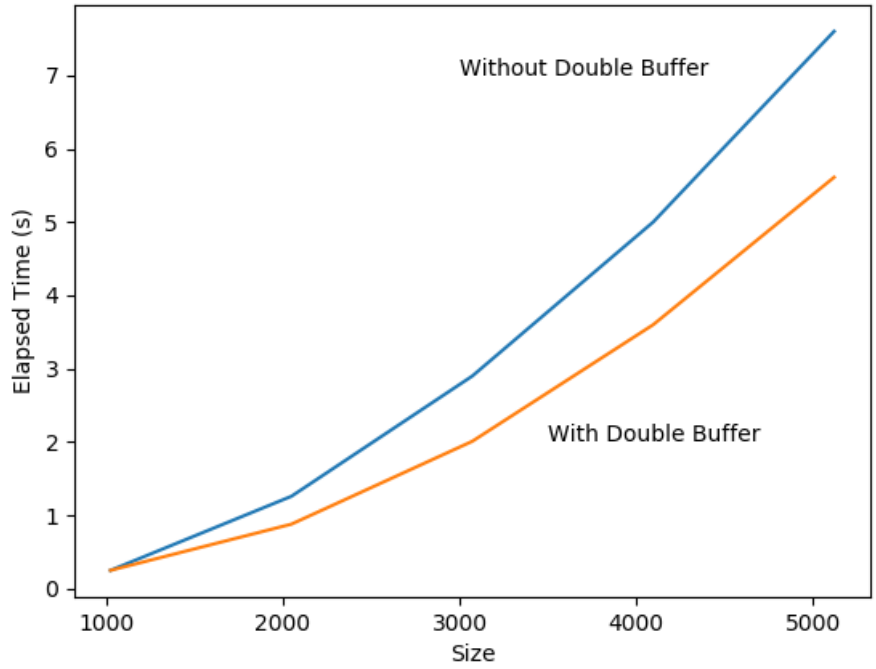
Double Buffer

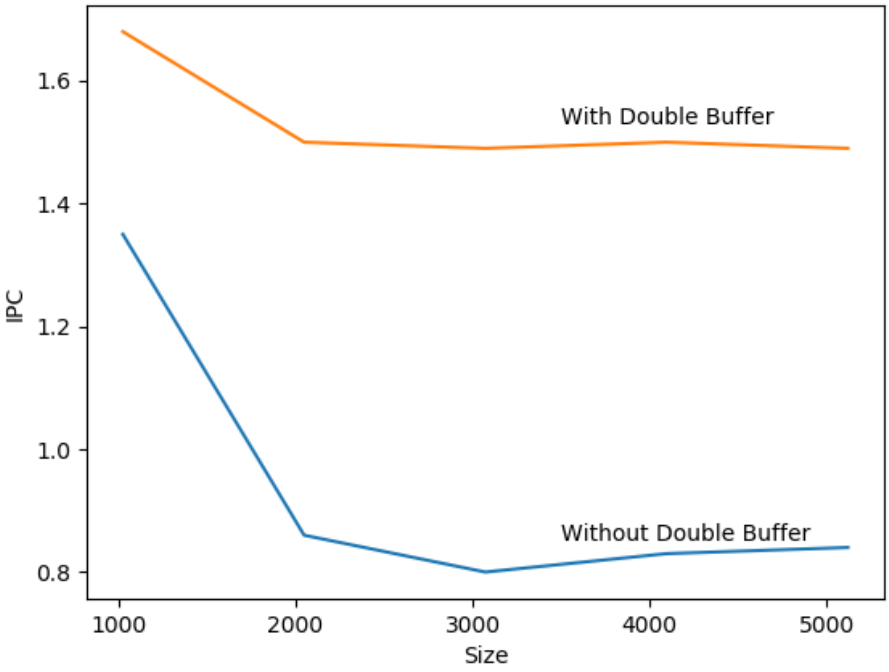
```
int iter = 0;
float* t;
while ( error > tol && iter < iter_max )
{
  iter++;
  laplace_step (A, temp, n);
  error= laplace_error (A, temp, n);
  t = out;
  out = in;
  in = t;
}
```

```
[-bash-4.2$ perf stat -e cycles,instructions,cache-misses,task-clock ./doubleBuffer 2048 100
 Jacobi relaxation Calculation: 2048 x 2048 mesh, maximum of 100 iterations
 Total Iterations:   100, ERROR: 0.079694, A[16][16]= 0.006917

  Performance counter stats for './doubleBuffer 2048 100':

      2.935.278.284      cycles                    #    3,174 GHz
      4.437.370.518      instructions              #    1,51  insns per cycle
          3.773.446      cache-misses              #    4,080 M/sec
        924,767449       task-clock (msec)         #    0,997 CPUs utilized

        0,927896078 seconds time elapsed
```

Instead of copying the whole input matrix in the output one, We can obtain better performance interchanging the matrix address using the current input matrix as the output matrix of the subsequent iteration and vice versa. We perform this playing a bit with the matrices' addresses, as showed in the previous picture. This modification is very performing! We notice from the compiler's results that the number of cycles is 1B less and the number of instructions is increased by the almost same amount. This growth in the number of

instructions performed and drop of the number of cycles leads to a higher IPC value, which pass from 0.86 to 1.51.
Also the cache-misses are highly reduced from 36.4M to 3.7M. This happens because we have deleted the nested loops operations used for copying the matrix, so all the memory accesses and the related cache-misses has been avoided. The elapsed time is consequent reduced too.
As usual we show the improvement obtained in terms of elapsed time and IPC using the subsequent two graphs located in the next page.



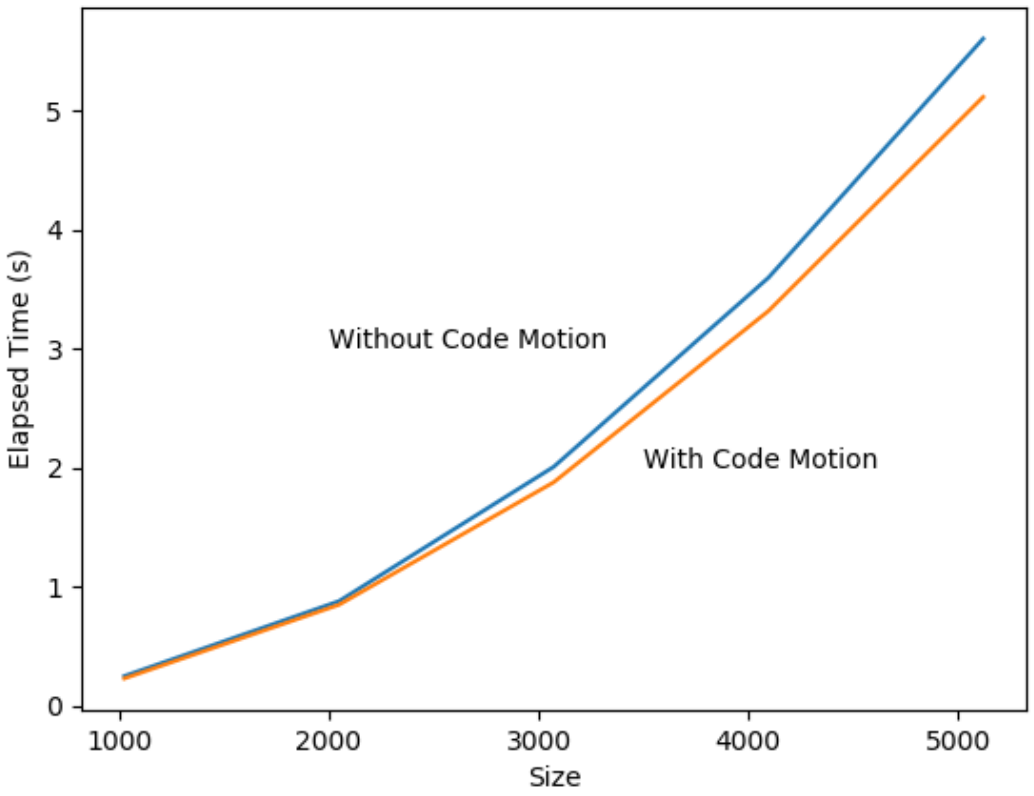


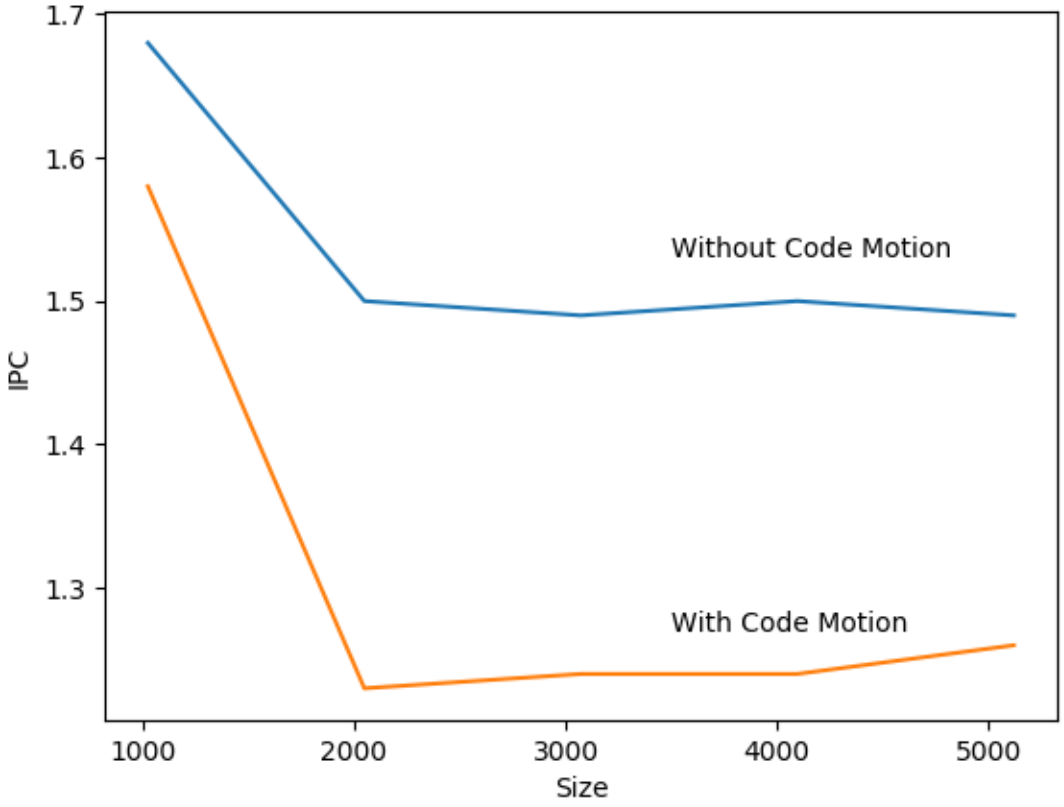
Code Motion

```
float laplace_error (float *old, float *new, int n)
{
  int i, j;
  float error=0.0f;
  for ( i=1; i < n-1; i++ )
    for ( j=1; j < n-1; j++ )
      error = max_error( error, old[j*n+i], new[j*n+i] );
  return sqrtf( error );
}

float max_error ( float prev_error, float old, float new )
{
  float t= fabsf( new - old );
  if (t> prev_error)
    return t;
  return prev_error;
}
```

```
Jacobi relaxation Calculation: 2048 x 2048 mesh, maximum of 100 iterations
Total Iterations:   100, ERROR: 0.079694, A[16][16]= 0.006917

 Performance counter stats for './codeMotion 2048 100':

     2.781.770.970      cycles                    #    3,165 GHz
     3.496.803.138      instructions              #    1,26  insns per cycle
         4.652.973      cache-misses              #    5,294 M/sec
       878,833725       task-clock (msec)         #    0,997 CPUs utilized

       0,881912120 seconds time elapsed
```

Moving the sqrtf function outside the ***max_error*** function that was called in a loop contained in the ***laplace_error*** function, we execute this operation only one time obtaining the same desired effect.
This obviously improves the program's performance, avoiding the execution of useless and heavy operations. In fact we have almost 1B of instruction less than the previous case and over than 1M of cache-misses less. However, the number of cycles remains more or less the same, so we obtain a bit worse IPC but a better elapsed time. The two graphs below shows the results achieved.


Elapsed Time (s)
0
1
2
3
4
5
1000
2000
3000
4000
5000
Size
Without Code Motion
With Code Motion

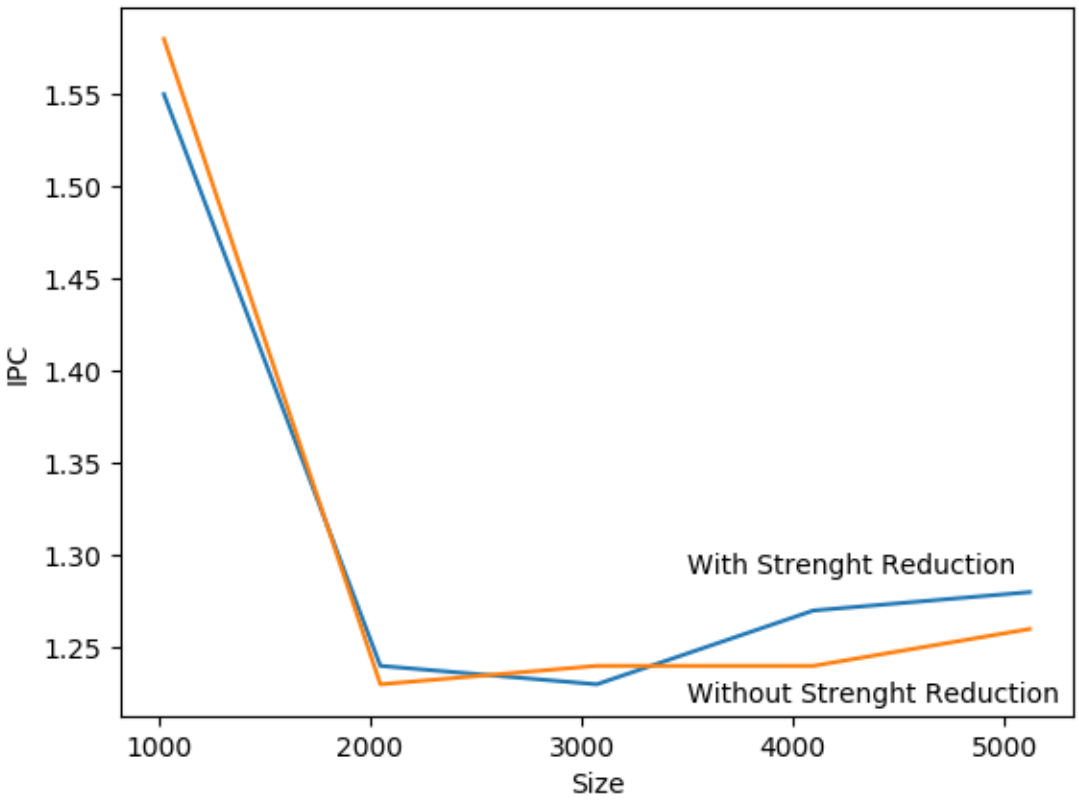

IPC
1.3
1.4
1.5
1.6
1.7
1000
2000
3000
4000
5000
Size
Without Code Motion
With Code Motion

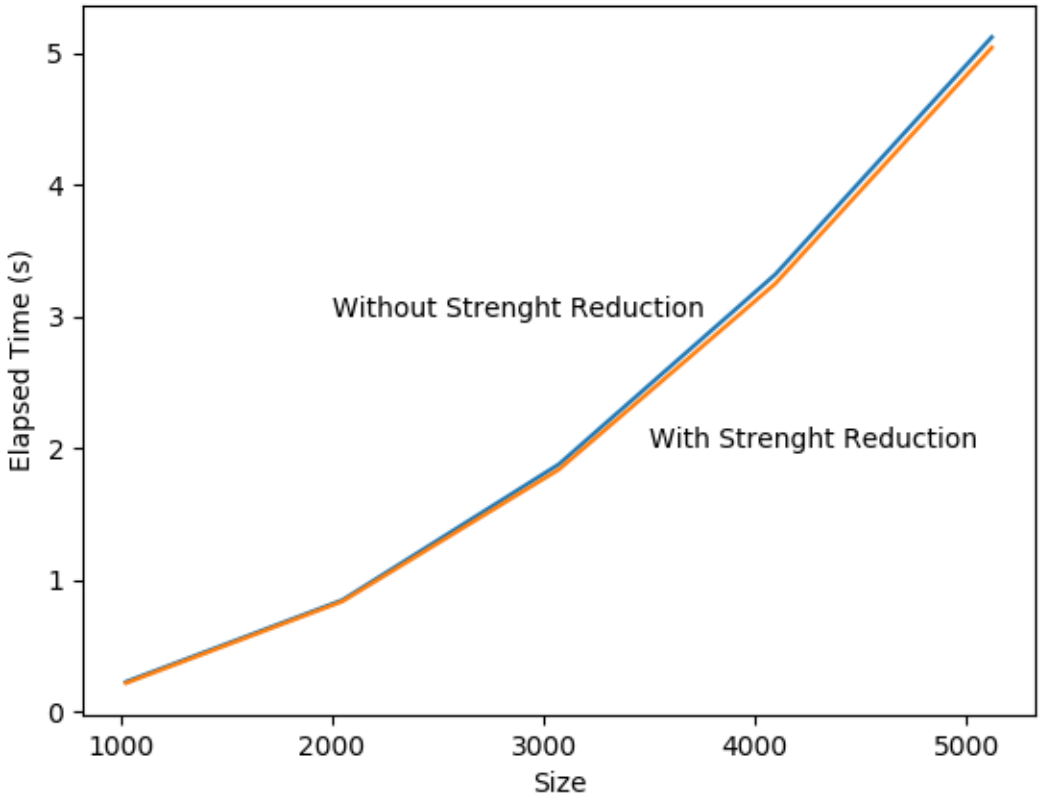
Strength Reduction

```
float stencil ( float v1, float v2, float v3, float v4)
{
  return (v1 + v2 + v3 + v4) * 0.25f;
}
```

```
Jacobi relaxation Calculation: 2048 x 2048 mesh, maximum of 100 iterations
[Total Iterations:   100, ERROR: 0.079694, A[16][16]= 0.006917

 Performance counter stats for './strenghtReduction 2048 100':

     2.824.207.748      cycles                    #    3,361 GHz
     3.496.636.693      instructions              #    1,24  insns per cycle
         4.761.102      cache-misses              #    5,667 M/sec
       840,209061       task-clock (msec)         #    0,997 CPUs utilized

       0,842807086 seconds time elapsed
```

We can use the principle of strength reduction in order to improve the performance of our algorithm. In the stencil function we can transform the division in a multiplication, because multiply operations are faster than the division ones for the different needs of transistors and computational step. However, this time we don't really notice a big difference of performance: we have a bit smaller elapsed time and a very similar IPC number.





Conclusions

Most of the computational time in the original code has been spent for the matrices memory accesses. Changing the access strategy in a clever way, that is accessing adjacent elements using the Loop Interchange technique, we have had a **speedoff** of almost **30x**.
The other tecniques applied has had less impact on the overall performaces, but not for this trascurable.
The memory access in a very important issue and a developer, expecially if he has to handle big quantity of data, must take into account.

Raffaele Bongo
Naci kurdoglu