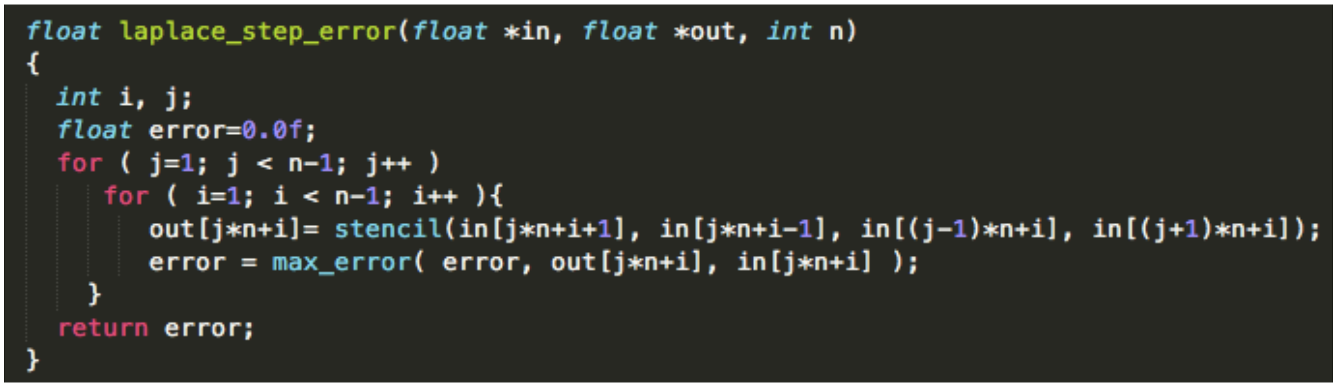
Performance improvement Laplace Equation Algorithm

In the next lines we show the performance of the algorithm in the current state (that we call ‘original code’.

```
Performance counter stats for './L2gcc 2048 1000':

28.153.993.006      cycles                #    3,375 GHz
34.889.078.920      instructions               #    1,24  insns per cycle
44.010.037          cache-misses               #    5,275 M/sec
8342,540570         task-clock (msec)     #    0,998 CPUs utilized
50.521.326          cache-references
8,360739493         seconds time elapsed
```

Loop Fusion – float



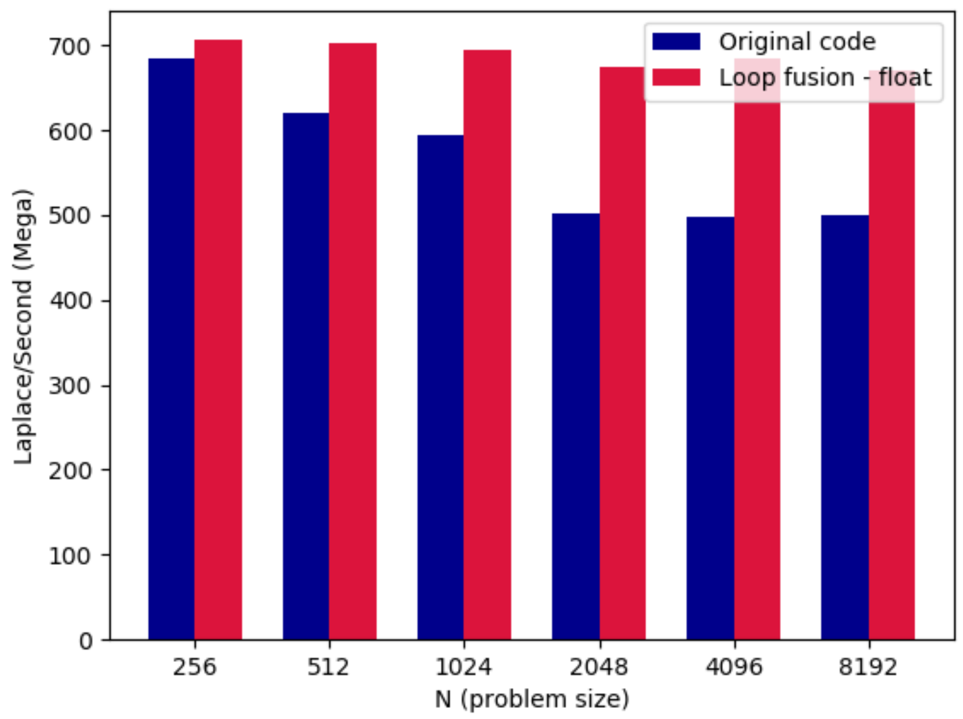
```
float laplace_step_error(float *in, float *out, int n)
{
  int i, j;
  float error=0.0f;
  for ( j=1; j < n-1; j++ )
    for ( i=1; i < n-1; i++ ){
      out[j*n+i]= stencil(in[j*n+i+1], in[j*n+i-1], in[(j-1)*n+i], in[(j+1)*n+i]);
      error = max_error( error, out[j*n+i], in[j*n+i] );
    }
  return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

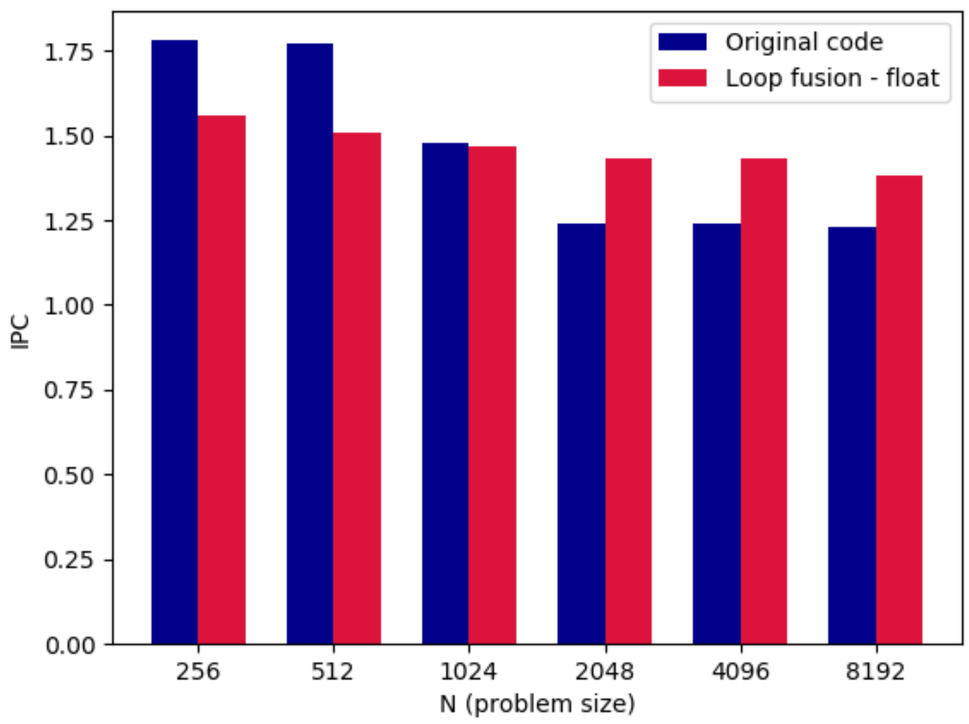
```
Performance counter stats for './L2opt 2048 1000':

20.716.017.698      cycles                #    3,346 GHz
29.539.932.855      instructions          #    1,43  insns per cycle
15.009.020          cache-misses          #    2,424 M/sec
6190,918837         task-clock (msec)     #    0,998 CPUs utilized
61.932.586          cache-references
6,204965157         seconds time elapsed
```

In the original code, we had 2 nested *for* loops: the first one for calculating the iterations and the second one for comparing the old and the new results in orfer to compute the error. Applying Loop Fusion optimization, and so calculating the error and the stencil iteration for a cell in the same time, we can eliminate the nested loops used for computing the error in the original code and this is explain the decrease in the number of cycles and instructions.

Memory required is the same since we still store 2 different matrixes but read operations decreased by half. The data are accessed in a more clever way using the same memory cells for computing the stencil and the error in the same cycle of the loop: this allow us to increse the cache-hits, because the compiler don’t need to demend extra data to higher level of cache or to the DRAM, that could be a very heavy task, but just reuse pre-demanded data.

We show down two bar-graphs that compare the performance in term of work-per-second and IPC w.r.t. various sizes. We can see that the difference between the work per second performed by the optimized code and the original code increase with the increment of the problem’size, that is what we expect with the reduction of instruction performed by the loop fusion, and also the IPC seems to follow the same trend in the size-space analyzed.

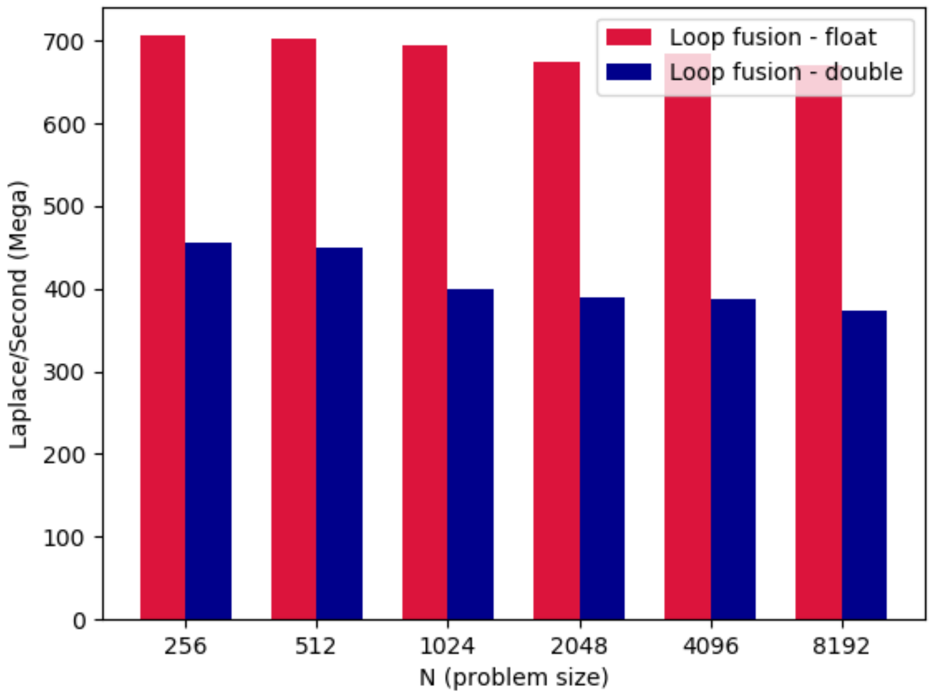

Original code
Loop fusion - float
Laplace/Second (Mega)
0
100
200
300
400
500
600
700
256
512
1024
2048
4096
8192
N (problem size)

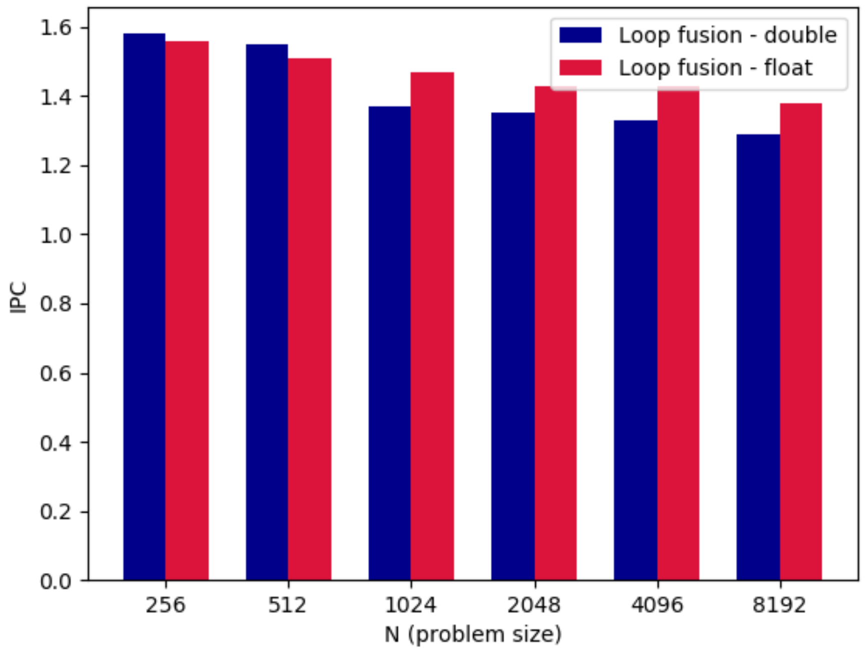

Original code
Loop fusion - float
IPC
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
256
512
1024
2048
4096
8192
N (problem size)

Loop Fusion : difference between float and double data Types

By changing float data type to double, we manage to see an increment in the number of cycles and required number of instructions. Double data type require more storage than floats and it consumes more time to load and store data. Also mathematical operations require more time and cycles to be completed.

In the graph showed after this text, we don't see a big difference concerning to the IPC between the two versions of the code, but instead we have big gap in the work-per-second: the float version seems to work two times more per second w.r.t. double version of the code. This is what we expect since the double numbers requires two times the memory space of floating numbers, so all the operations performed should require two time the work, and this is what actually happens.




Loop fusion - double
Loop fusion - float
IPC
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
256
512
1024
2048
4096
8192
N (problem size)

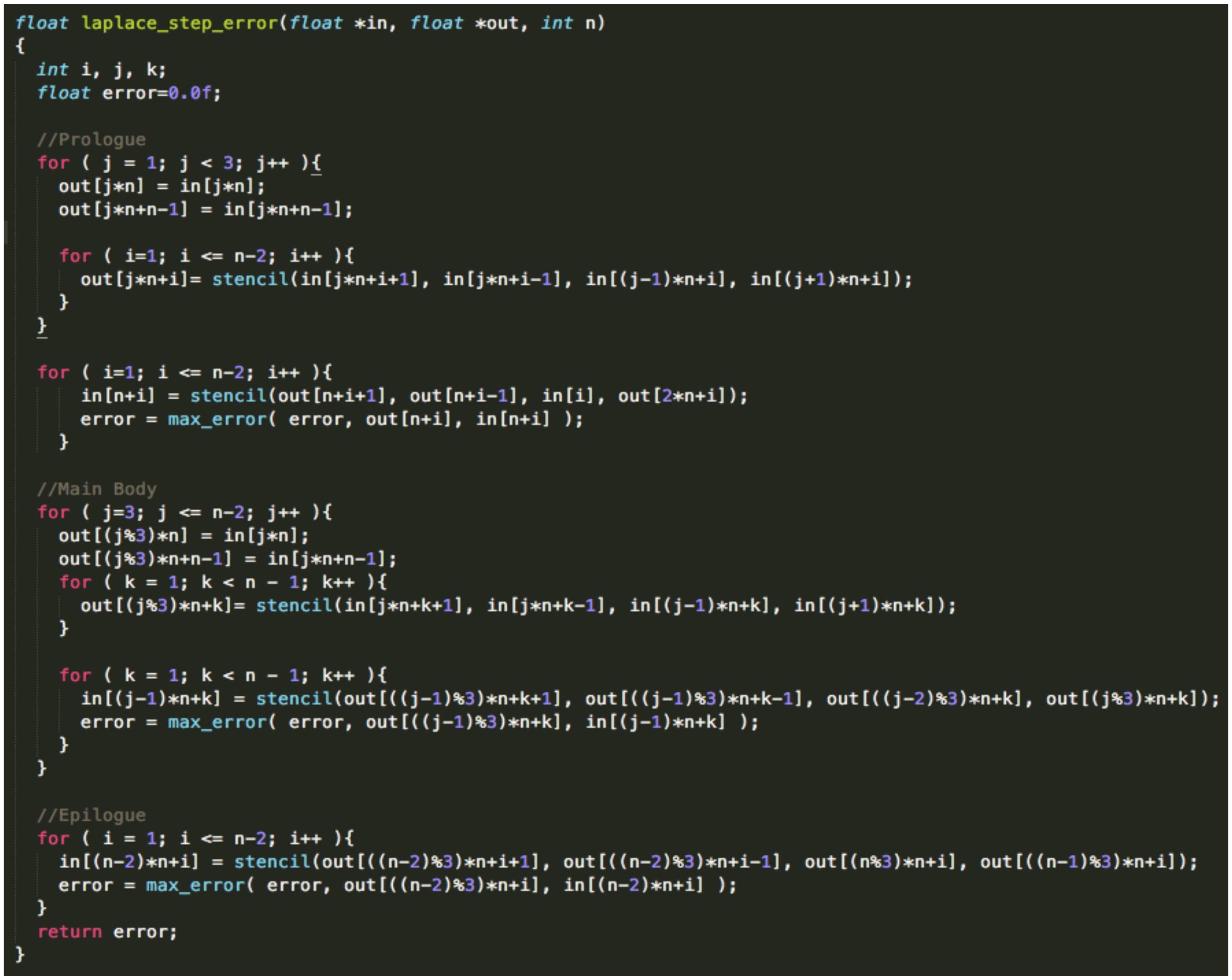
Temporal Blocking

LOOP FUSION: Performance counter stats for './L2opt 2048 1000':

```
20.716.017.698     cycles                  #    3,346 GHz
29.539.932.855     instructions              #    1,43  insns per cycle
15.009.020         cache-misses              #    2,424 M/sec
6190,918837        task-clock (msec)         #    0,998 CPUs utilized
61.932.586         cache-references

6,204965157        seconds time elapsed
```

TEMPORAL BLOCKING: Performance counter stats for './L2opt7 2048 1000':

```
20.396.904.828      cycles                  #    3,342 GHz
26.353.421.304      instructions              #    1,29  insns per cycle
13.530.550          cache-misses              #   28,989 % of all cache refs
6102,726333         task-clock (msec)         #    0,998 CPUs utilized
46.675.419          cache-references

6,117027448         seconds time elapsed
```



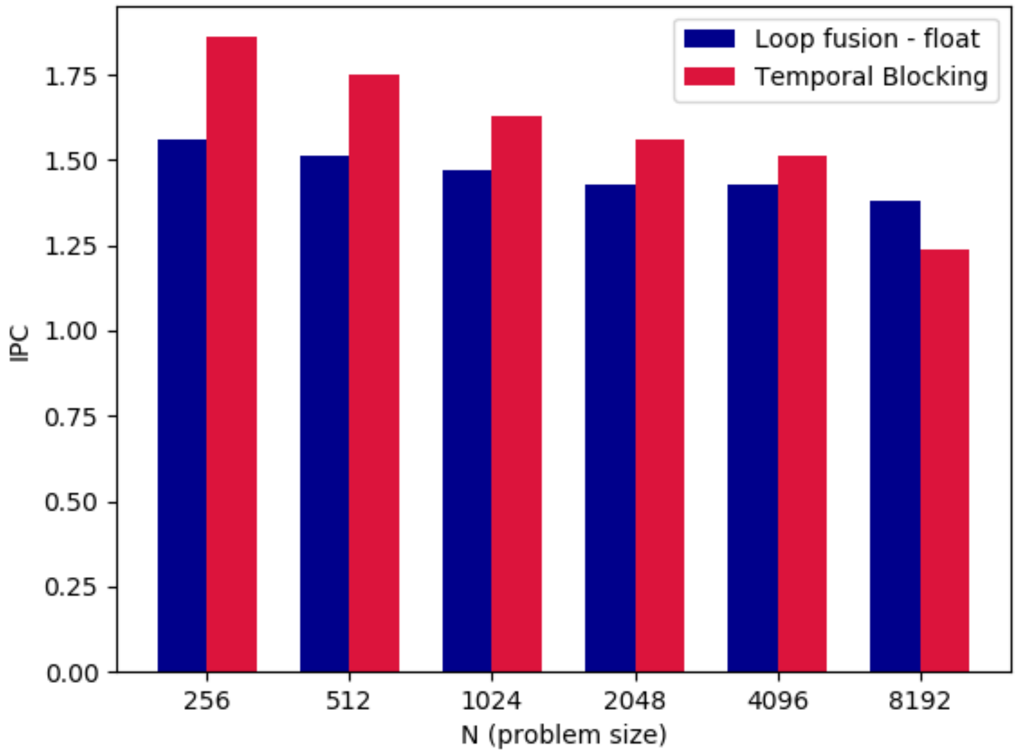
```
float laplace_step_error(float *in, float *out, int n)
{
  int i, j, k;
  float error=0.0f;

  //Prologue
  for ( j = 1; j < 3; j++ ){
    out[j*n] = in[j*n];
    out[j*n+n-1] = in[j*n+n-1];

    for ( i=1; i <= n-2; 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]);
    }
  }

  for ( i=1; i <= n-2; i++ ){
      in[n+i] = stencil(out[n+i+1], out[n+i-1], in[i], out[2*n+i]);
      error = max_error( error, out[n+i], in[n+i] );
    }

  //Main Body
  for ( j=3; j <= n-2; j++ ){
    out[(j%3)*n] = in[j*n];
    out[(j%3)*n+n-1] = in[j*n+n-1];
    for ( k = 1; k < n - 1; k++ ){
      out[(j%3)*n+k]= stencil(in[j*n+k+1], in[j*n+k-1], in[(j-1)*n+k], in[(j+1)*n+k]);
    }

    for ( k = 1; k < n - 1; k++ ){
      in[(j-1)*n+k] = stencil(out[((j-1)%3)*n+k+1], out[((j-1)%3)*n+k-1], out[((j-2)%3)*n+k], out[(j%3)*n+k]);
      error = max_error( error, out[((j-1)%3)*n+k], in[(j-1)*n+k] );
    }
  }

  //Epilogue
  for ( i = 1; i <= n-2; i++ ){
    in[(n-2)*n+i] = stencil(out[((n-2)%3)*n+i+1], out[((n-2)%3)*n+i-1], out[(n%3)*n+i], out[((n-1)%3)*n+i]);
    error = max_error( error, out[((n-2)%3)*n+i], in[(n-2)*n+i] );
  }
  return error;
}
```

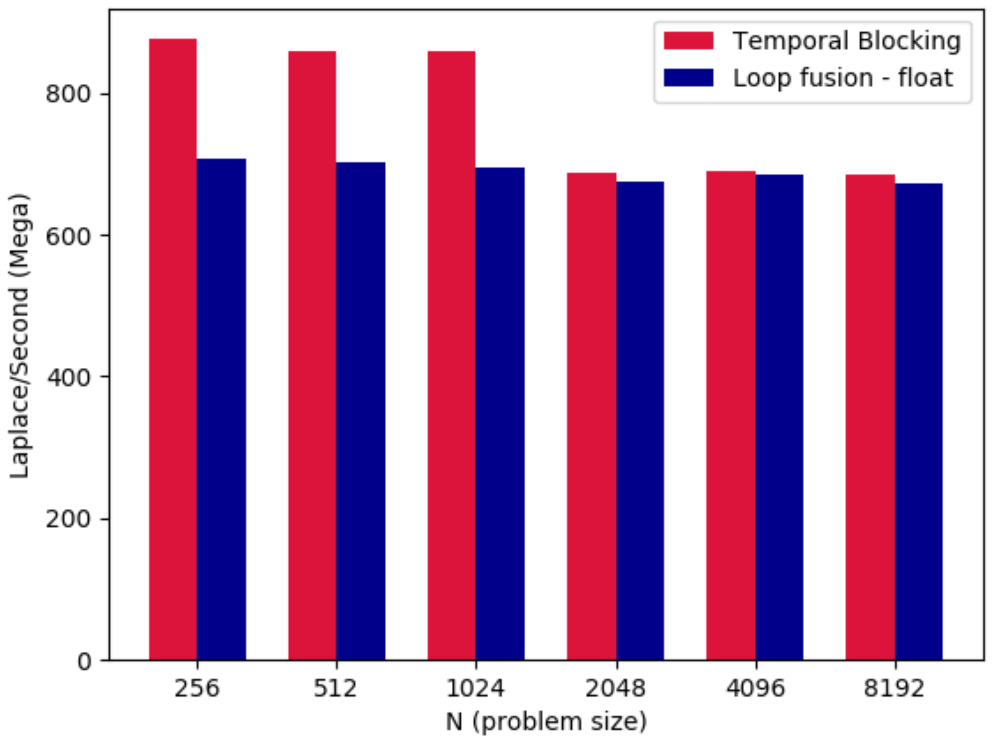
By changing the size of 'temp' array from n*n to 3*n, we manage to decrease the number of load-store operations, number of instructions used, cache misses and cache-references. We are calculating 2 laplace step iterations in each cycle of the *for* loop. The number of load and store operations is decreased significantly since we are not constantly loading temp matrix over and over again: this lead to decrease of the number of instructions used. Since less instructions are needed, less clock-cycles are performed for each iteration.

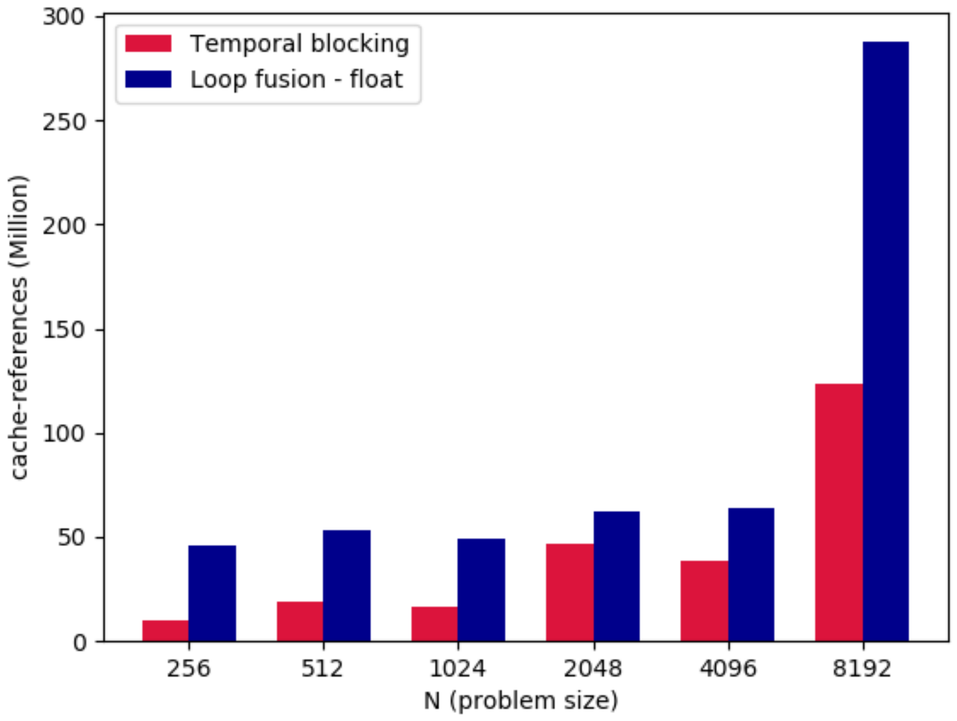
Using only three rows instead of the entire matrix, lead to exploit the better the principle of locality for the memory and so to reuse data already loaded in the nearest possible level of cache and to make all the computations faster due to the less time needed for retriving the data. This lead to a reduction in cache-references, so the hit in the last level of cache, and this is more accentuated bigger is the size of the problem (this behaviour is clearly showed in the graph below). decreased cache reference results in faster computation and operations since we are using a lower level.

The bottleneck that we have overcome with this technique concern the memory and we overcome it using the memory in a more clever way.

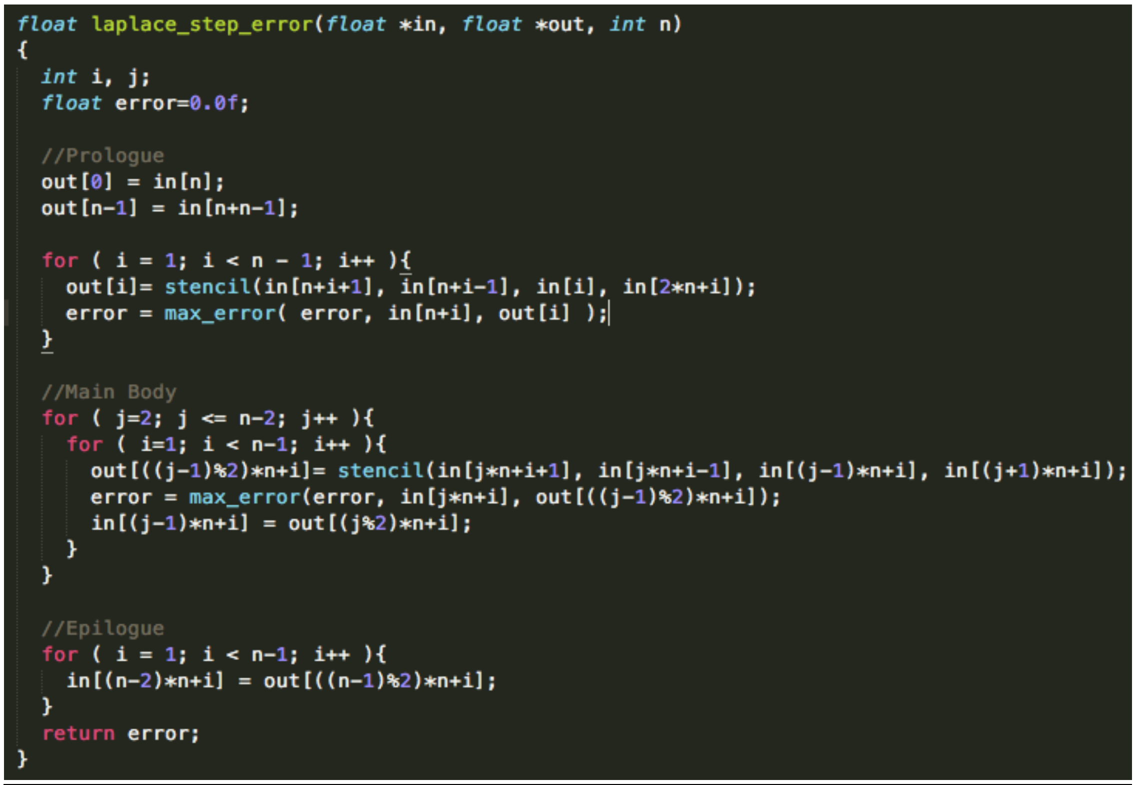
The graphs of the IPC and the work-per-second show an increase of performance for the temporal blocking version, but the biggest contribute seems to be given by the bigly reduced access to the last level cache.




Temporal Blocking
Loop fusion - float
Laplace/Second (Mega)
0
200
400
600
800
256
512
1024
2048
4096
8192
N (problem size)


Temporal blocking
Loop fusion - float
cache-references (Million)
0
50
100
150
200
250
300
256
512
1024
2048
4096
8192
N (problem size)

Two Rows Buffer



```c
float laplace_step_error(float *in, float *out, int n)
{
  int i, j;
  float error=0.0f;

  //Prologue
  out[0] = in[n];
  out[n-1] = in[n+n-1];

  for ( i = 1; i < n - 1; i++ ){
    out[i]= stencil(in[n+i+1], in[n+i-1], in[i], in[2*n+i]);
    error = max_error( error, in[n+i], out[i] );
  }

  //Main Body
  for ( j=2; j <= n-2; j++ ){
    for ( i=1; i < n-1; i++ ){
      out[((j-1)%2)*n+i]= stencil(in[j*n+i+1], in[j*n+i-1], in[(j-1)*n+i], in[(j+1)*n+i]);
      error = max_error(error, in[j*n+i], out[((j-1)%2)*n+i]);
      in[(j-1)*n+i] = out[(j%2)*n+i];
    }
  }

  //Epilogue
  for ( i = 1; i < n-1; i++ ){
    in[(n-2)*n+i] = out[((n-1)%2)*n+i];
  }
  return error;
}
```

TEMPORAL BLOCKING: Performance counter stats for './L2opt7 2048 1000':

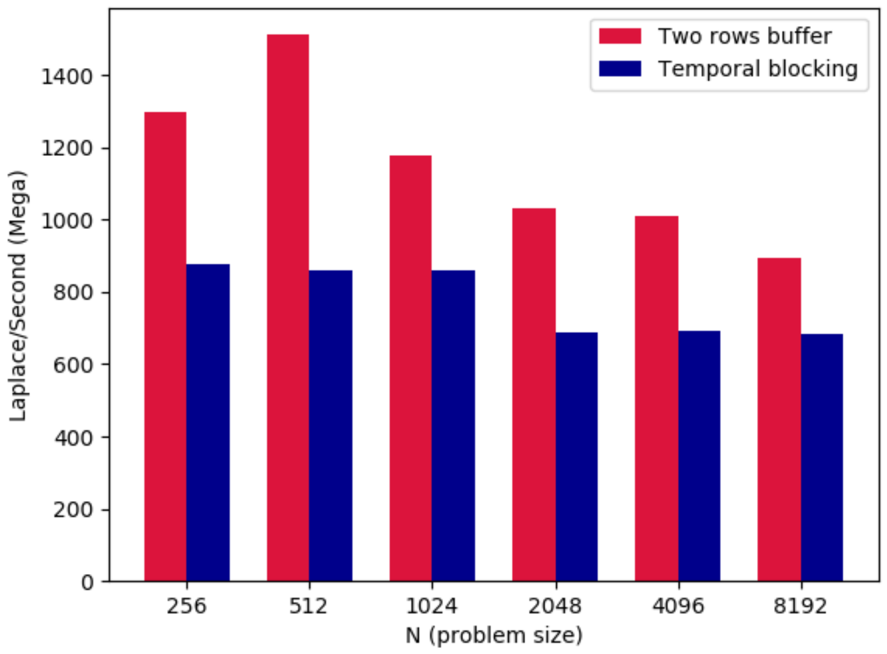
```
20.396.904.828      cycles              #    3,342 GHz
26.353.421.304      instructions          #    1,29  insns per cycle
13.530.550          cache-misses           #   28,989 % of all cache refs
6102,726333         task-clock (msec)       #    0,998 CPUs utilized
46.675.419          cache-references

6,117027448         seconds time elapsed
```

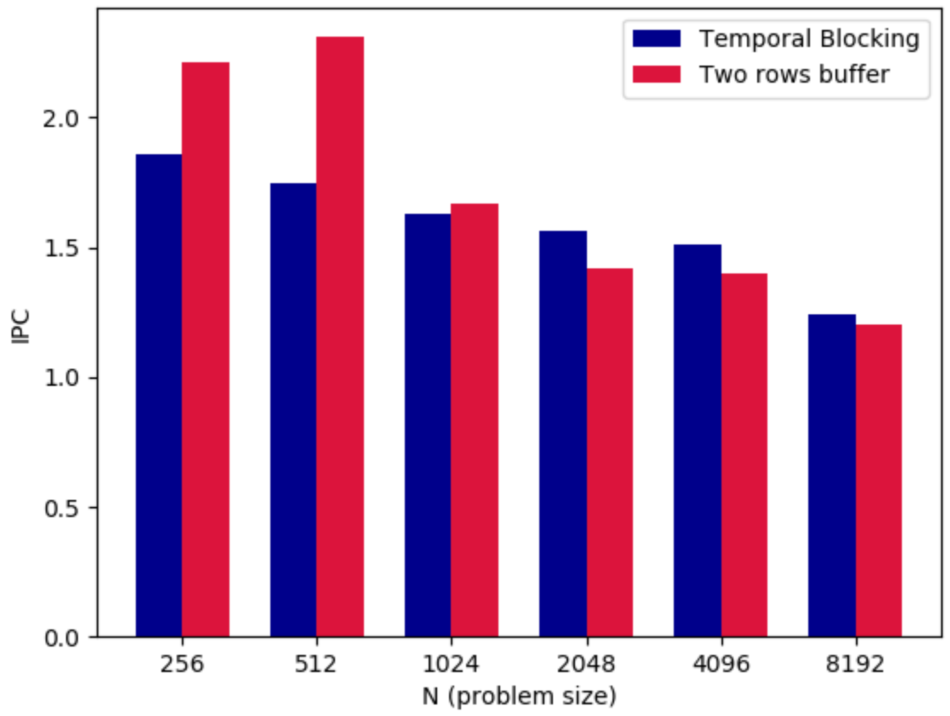
TWO ROWS BUFFER: Performance counter stats for './L2opt6 2048 1000':

```
13.676.369.929      cycles              #    3,367 GHz
19.487.873.131      instructions          #    1,42  insns per cycle
10.625.521          cache-misses           #   50,580 % of all cache refs
4062,099537         task-clock (msec)       #    0,998 CPUs utilized
21.007.156          cache-references        #    5,172 M/sec

4,071942124         seconds time elapsed
```

Using only two rows prevents us from doing two iterations in a single loop but instead of doing that, we can directly store the result of a stencil iteration in the original matrix. This decreases the load-store operations since data is already in the cache. As a result each cycle requires less instructions therefore overall efficiency is better, both in terms of memory and speed.

As we can see in the graphs below, the big difference is not given by the IPC, that is better but of a little amount. What really changes and improves the performance is the work performed per second that, due to the big gap of instructions between the two versions, gives much better results for the two rows version.





Conclusions

With these optimizations we overcome both problems of memory and computation compound and we get that working cleverly, which means retrieve adjacent cells of memory avoiding big jumps and not reserving extra space in the memory that we could leave free just thinking about how to reuse data and memory position which we have already reserved, can bring our code to an much higher level of efficiecy.

Studying the case of the loop fusion using float and double data type, and analysing the work-per-second graph, we got the importance of not under-estimating the weight of a right data type use, because the program is deeply affected by that.

Raffaele Bongo
Naci kurdoglu