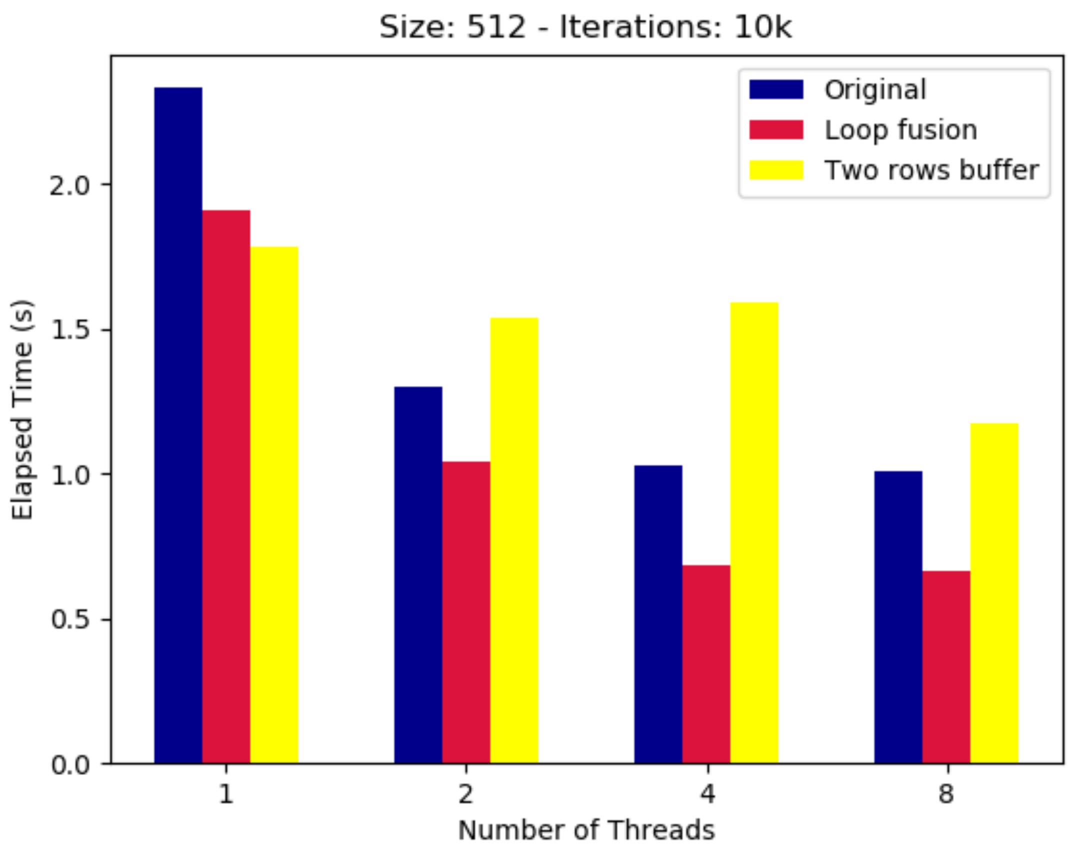
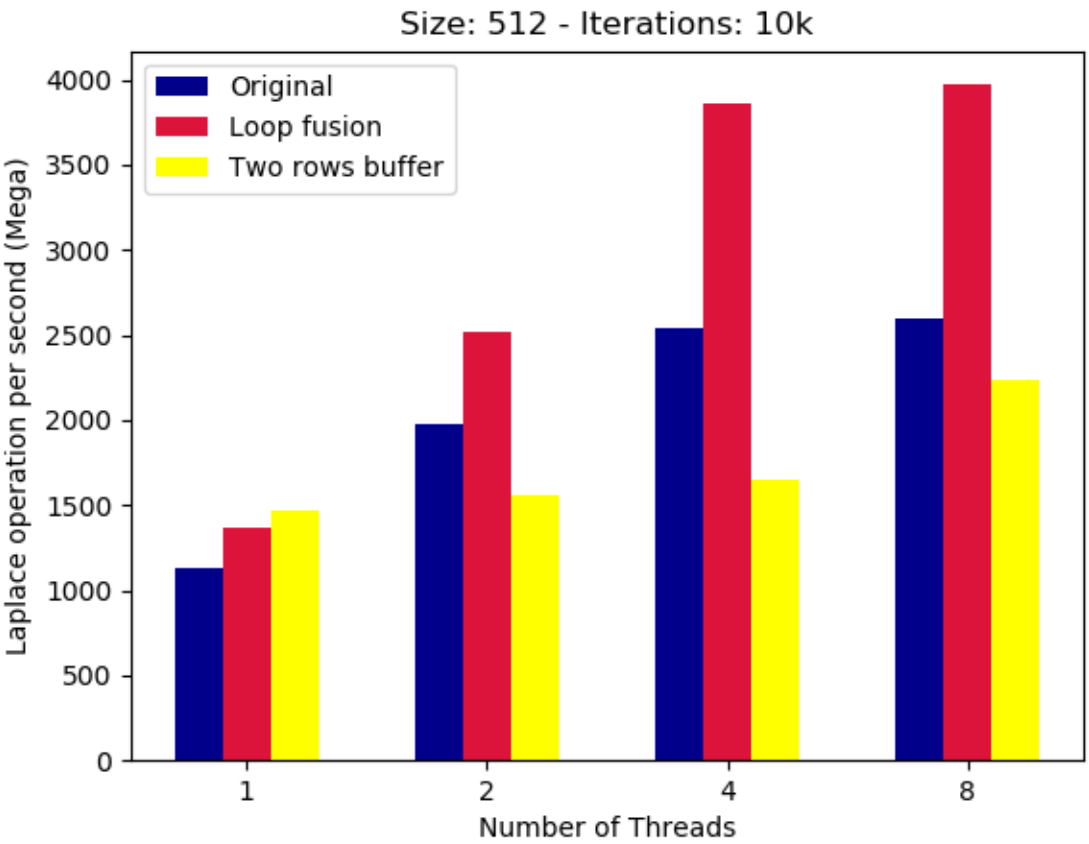
Performance improvement Laplace Equation Algorithm

Multi-threads versions

In this report we provide a multithreded version of the various versions of the algorithm furnishing comparisons and results. We will study two execution cases: problem size 512 with 10k iterations and problem size 4096 with 100 iterations, running them with 1, 2, 4 and 8 threads on the laboratory processor.

Size: 512 – Iterations: 10k





The two grapsh above illustrate the elapsed time and the work per second performed by the three versions of the algorithm: orignal code, loop fusion and double buffer optimization. We can see that using more thread leads generally to better performance if the work can be divided between the threads. In fact here we notice a reduction of elapsed time and an increse of the work per second that follow the increasing number of the threads, but not in a linear way.

Between the case of 4 and 8 threads there is not so much difference of performances.

This happens for two main reasons: because creating threads, especially using openMP, is an heavy operation which needs lot of overhead and also for a synchronization problem. In fact more threads we use more is the work that the processor have to perform for synchronize the operations executed. This synchronization overhead depends also on the data structure used and how much complex are the operations that have to be performed.

Loop fusion code

```
float laplace_step_error(float *in, float *out, int n)
{
  int i, j;
  float error=0.0f;
  #pragma omp parallel for reduction (max:error)
  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;
}
```

Loop fusion 4 threads

```
Samples: 11K of event 'cycles', Event count (approx.): 8294957092
Overhead  Command    Shared Object      Symbol
  92,91%  L2Tfusion  L2Tfusion          [.] laplace_step_error._omp_fn.0
   3,14%  L2Tfusion  libgomp.so.1.0.0   [.] 0x0000000000010559
   1,09%  L2Tfusion  libgomp.so.1.0.0   [.] 0x000000000001054b
Samples: 11K of event 'instructions', Event count (approx.): 11732259806
Overhead  Command    Shared Object      Symbol
  98,11%  L2Tfusion  L2Tfusion          [.] laplace_step_error._omp_fn.0
   1,02%  L2Tfusion  libgomp.so.1.0.0   [.] 0x0000000000010559
```

Loop fusion 8 threads

```
Samples: 22K of event 'cycles', Event count (approx.): 17675061647
Overhead  Command    Shared Object      Symbol
  89,45%  L2Tfusion  L2Tfusion          [.] laplace_step_error._omp_fn.0
   7,14%  L2Tfusion  libgomp.so.1.0.0   [.] gomp_team_barrier_wait_end
   2,28%  L2Tfusion  libgomp.so.1.0.0   [.] gomp_barrier_wait_end
Samples: 22K of event 'instructions', Event count (approx.): 12241752840
Overhead  Command    Shared Object      Symbol
  93,61%  L2Tfusion  L2Tfusion          [.] laplace_step_error._omp_fn.0
   4,57%  L2Tfusion  libgomp.so.1.0.0   [.] gomp_team_barrier_wait_end
   1,37%  L2Tfusion  libgomp.so.1.0.0   [.] gomp_barrier_wait_end
```

The images above show how the synchronization overhead affects the performances of the algorithm for the loop fusion optimization. Using 4 threads, only round the 4% of the cycles is employed in heavy thread setting/synchronization operations instead, using 8 threads, these operations take a percentage of the cycles around the 10%.

This analysis can explain both the improvement and its kind of growth.

Furthermore, we can notice a particular behavior of the double buffer optimization.

Double Buffer code

```
#pragma omp parallel for reduction(max:err) private(j) shared( A, n)
for ( j=2; j < n-1; j++ )
  { // Main Body

    err= step_error_line ( err, ROW(A,j-1), ROW(A,j), ROW(A,j+1), ROW(TMP,(j+1)%2), n );
    copy_line            ( ROW(TMP,j%2), ROW(A,j-1), n );

  }
 copy_line ( ROW(TMP, j%2), ROW(A,n-2), n );  // Epilogue
 iter++;

}
```

We have implemented it as showed in the picture above, parallelizing the for in the main body. This implementation doesn't lead to a correct result, because there are data dependencies that should have to be managed. So if a thread copy back in the "wrong moment" a tmp line in the A matrix, another thread which is computing the tmp matrix could compute result with the wrong data and this lead to a "chain reaction" of wrong computations. However, solve the problem requires fine-grain synchronizations of the threads which requires time to be implemented and are out of our scopes. In fact, the performance of this implementation should be similar, maybe a bit better because we avoid the synch-overhead which would make the program perform in the right way.

The single thread version of the double buffer performs correctly better than the other versions, while the multi-thread version have an increasing performance w.r.t. the single thread and its same versions execued with less threads, but still worse than the original code and the loop fusion case. The images that follow should explain why.

Double buffer 2 threads

```
Samples: 13K of event 'cycles', Event count (approx.): 10603900548
Overhead  Command    Shared Object      Symbol
  98,04%  L2Tbuffer  L2Tbuffer          [.] main._omp_fn.0
   0,51%  L2Tbuffer  libgomp.so.1.0.0   [.] gomp_team_barrier_wait_end
   0,42%  L2Tbuffer  libgomp.so.1.0.0   [.] gomp_barrier_wait_end
```

```
Samples: 13K of event 'instructions', Event count (approx.): 15811922399
Overhead  Command    Shared Object     Symbol
  99,32%  L2Tbuffer  L2Tbuffer         [.] main._omp_fn.0
   0,25%  L2Tbuffer  libgomp.so.1.0.0  [.] gomp_team_barrier_wait_end
   0,11%  L2Tbuffer  libgomp.so.1.0.0  [.] gomp_barrier_wait_end
```

Double buffer 4 threads

```
Samples: 83K of event 'cycles', Event count (approx.): 66168669812
Overhead  Command    Shared Object     Symbol
  36,36%  L2Tbuffer  L2Tbuffer         [.] main._omp_fn.0
  31,34%  L2Tbuffer  libgomp.so.1.0.0  [.] gomp_team_barrier_wait_end
  29,92%  L2Tbuffer  libgomp.so.1.0.0  [.] gomp_barrier_wait_end
Samples: 82K of event 'instructions', Event count (approx.): 33440087589
Overhead  Command    Shared Object     Symbol
  46,99%  L2Tbuffer  L2Tbuffer         [.] main._omp_fn.0
  26,82%  L2Tbuffer  libgomp.so.1.0.0  [.] gomp_team_barrier_wait_end
  25,73%  L2Tbuffer  libgomp.so.1.0.0  [.] gomp_barrier_wait_end
```

Double buffer 8 threads

```
Samples: 320K of event 'cycles', Event count (approx.): 253847809139
Overhead  Command    Shared Object        Symbol
  42,64%  L2Tbuffer  libgomp.so.1.0.0     [.] gomp_barrier_wait_end
  42,42%  L2Tbuffer  libgomp.so.1.0.0     [.] gomp_team_barrier_wait_end
  13,49%  L2Tbuffer  L2Tbuffer            [.] main._omp_fn.0
Samples: 320K of event 'instructions', Event count (approx.): 105117819764
Overhead  Command    Shared Object     Symbol
  42,46%  L2Tbuffer  libgomp.so.1.0.0  [.] gomp_barrier_wait_end
  42,12%  L2Tbuffer  libgomp.so.1.0.0  [.] gomp_team_barrier_wait_end
  14,97%  L2Tbuffer  L2Tbuffer         [.] main._omp_fn.0
```

These images show a very big increase of the instructions and cycles devoted to the synchronization problems with the increase of the number of threads. The incresing number of threads still leads to improvements in the work per second performed, but it doesn’t allow the performance to be better than the previous implementations of the algorithm. Maybe with a correct and precise openMP usage this could be possible.

Size: 4096 – Iterations: 100


Size: 4096 - Iterations: 100
Original
Loop fusion
Two rows buffer
Laplace operation per second (Mega)
0
200
400
600
800
1000
1200
1
2
4
8
Number of Threads


Size: 4096 - Iterations: 100
Original
Loop fusion
Two rows buffer
IPC per thread
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1
2
4
8
Number of Threads

In the previous graph is showed the elapsed time, the work per second and the IPC per thread of the various algorithm's versions executed. We can notice a specular behaviour between the elapsed time and the work performed per second: to the increasing number of threads, firstly correspond a diminuition of elapsed time and an increase of the work per second, and after the opposite trend, that is a little increse of elapsed time and a little decrese of the work performed per second.

Before the performance bottleneck was the synchronization overhead, especially in the double buffer optimization. Is it the case also here?

Double buffer 8 threads

```
Samples: 43K of event 'cycles', Event count (approx.): 34271596526
Overhead  Command    Shared Object       Symbol
  96,97%  L2Tbuffer  L2Tbuffer           [.] main._omp_fn.0
   1,96%  L2Tbuffer  libgomp.so.1.0.0    [.] gomp_team_barrier_wait_end
```

```
Samples: 42K of event 'instructions', Event count (approx.): 9732772752
Overhead  Command    Shared Object     Symbol
  96,12%  L2Tbuffer  L2Tbuffer         [.] main._omp_fn.0
   3,03%  L2Tbuffer  libgomp.so.1.0.0  [.] gomp_team_barrier_wait_end
```

The two previous picture show the cycles and the instruction performed by the 8 threads version of the double buffer code, the version that should have the highest overhead for synchronization.

We can easily notice that the synchronization work is not that much, it take around the 2% of the cycles and the 3% of the instructions in our worse case, far from the 10% of the size 512.

So we have get that parallelisation comes with some overhead.

Only if we spawn enough parallel work, the gain of distributing work over parallel threads will outweigh the incurred overhead costs.

As the graph above show, the cache-misses rate increse with the increse of the number of threads. This is due to memory bounds: every core (so each thread) in our processor have the access to two level of cache and share with the other core the level 3. When the size of the problem is too big, the the data have to be retrieved from the L3 cache or from the DRAM and this leads, as seen also for the single thread version, to a delay. Using more threads this aspect is emphasized, because more threads use the L3 shared cache, the data change there more probably and increse the possibility of misses and so of delay.

The double buffer optimization has a bit different behaviour on the last graph both for the not precise implementation commented before and for the different use of the memory that is the core of the technique.

Conclusions

Using multiple threads can lead to better performances only if the code don't have complex data dependencies to be managed and if the size is neither too small or too big: in the first case, we have too much overhead for the synhronization tasks, in the second case the bottleneck is the memory bound.

GPU version

We now run the algorithm on the lab GPU and we will compare the performances obtained running the same code with different problem size on the CPU.

```
float laplace_step(float *in, float *out, int n)
{
  int i, j;
  float error=0.0f;
  #pragma acc data copy ( in[0:n*n], out[0:n*n] )
  #pragma acc parallel loop reduction(max:error)
  for ( j=1; j < n-1; j++ )
    #pragma acc loop gang vector
    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;
}
```

In the first attempt to utilise GPU, we used the directive *#pragma acc data copy* in order to move data from the main memory to the GPU's memory in the *laplace_step()* function, so performing the data movement between the host and the device each iteration. This lead to execute the most heavy operation that a GPU can perform many useless times, pushing the elapsed time to a very high value, around 22,32 seconds. We can see the performance summed up in the next picture.

```
-bash-4.2$ pgprof ./1GPUmemcpyProblem 2048 1000
Jacobi relaxation Calculation: 2048 x 2048 mesh, maximum of 1000 iterations
==531== PGPROF is profiling process 531, command: ./1GPUmemcpyProblem 2048 1000
Total Iterations:  1000, ERROR: 0.018806, A[16][16]= 0.011906
==531== Profiling application: ./1GPUmemcpyProblem 2048 1000
==531== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   58.54%  7.71118s      3000  2.5704ms  1.1200us  5.7548ms  [CUDA memcpy HtoD]
                   40.35%  5.31586s      5000  1.0632ms  1.2160us  2.7542ms  [CUDA memcpy DtoH]
                    1.08%  142.13ms      1000  142.13us  121.64us  742.20us  laplace_step_23_gpu
                    0.03%  4.1576ms      1000  4.1570us  3.8720us  32.481us  laplace_step_23_gpu__red
      API calls:   60.68%  6.01542s      4000  1.5039ms     997ns  4.2575ms  cuStreamSynchronize
                   27.59%  2.73499s      9996  273.61us     836ns  3.0297ms  cuEventSynchronize
                    4.87%  482.81ms         1  482.81ms  482.81ms  482.81ms  cuDevicePrimaryCtxRetain
                    1.98%  196.49ms      5000  39.297us  5.5700us  790.81us  cuMemcpyDtoHAsync
                    1.56%  155.11ms         1  155.11ms  155.11ms  155.11ms  cuDevicePrimaryCtxRelease
                    1.39%  138.26ms      3000  46.086us  8.8680us  560.54us  cuMemcpyHtoDAsync
                    0.61%  60.719ms      2000  30.359us  9.0170us  625.67us  cuLaunchKernel
                    0.56%  55.332ms      9998  5.5340us  1.4210us  464.82us  cuEventRecord
                    0.41%  40.880ms         1  40.880ms  40.880ms  40.880ms  cuMemHostAlloc
                    0.17%  16.683ms      4000  4.1700us     656ns  47.151us  cuPointerGetAttributes
                    0.09%  8.6643ms         1  8.6643ms  8.6643ms  8.6643ms  cuMemFreeHost
                    0.05%  5.0735ms         5  1.0147ms  605.87us  2.3113ms  cuMemAlloc
                    0.02%  1.7209ms         1  1.7209ms  1.7209ms  1.7209ms  cuMemAllocHost
                    0.00%  430.07us         1  430.07us  430.07us  430.07us  cuModuleLoadData
                    0.00%  38.496us         1  38.496us  38.496us  38.496us  cuStreamCreate
                    0.00%  22.570us         4  5.6420us     509ns  14.752us  cuEventCreate
                    0.00%  18.348us         3  6.1160us     283ns  15.976us  cuCtxSetCurrent
                    0.00%  11.653us         3  3.8840us     257ns  9.4380us  cuDeviceGetCount
                    0.00%  5.2430us         4  1.3100us     228ns  2.3260us  cuDeviceGet
                    0.00%  3.6410us         2  1.8200us     769ns  2.8720us  cuModuleGetFunction
                    0.00%  3.4120us         6     568ns     326ns  1.2570us  cuDeviceGetAttribute
                    0.00%  2.1090us         1  2.1090us  2.1090us  2.1090us  cuMemFree
                    0.00%  1.2810us         2     640ns     239ns  1.0420us  cuDeviceComputeCapability
                    0.00%     932ns         1     932ns     932ns     932ns  cuCtxGetCurrent
```

In order to use the GPU in a proper way and have performance improvements, we applied some modifications to the code:

```
  #pragma acc data copy ( A[0:n*n], temp[0:n*n] )
  while ( error > tol*tol && iter < iter_max )
  {
    iter++;
    error= laplace_step (A, temp, n);
    float *swap= A; A=temp; temp= swap; // swap pointers A & temp
  }
  error = sqrtf( error );
  printf("Total Iterations: %5d, ERROR: %0.6f, ", iter, error);
  printf("A[%d][%d]= %0.6f\n", n/128, n/128, A[(n/128)*n+n/128]);

  free(A); free(temp);
}
```

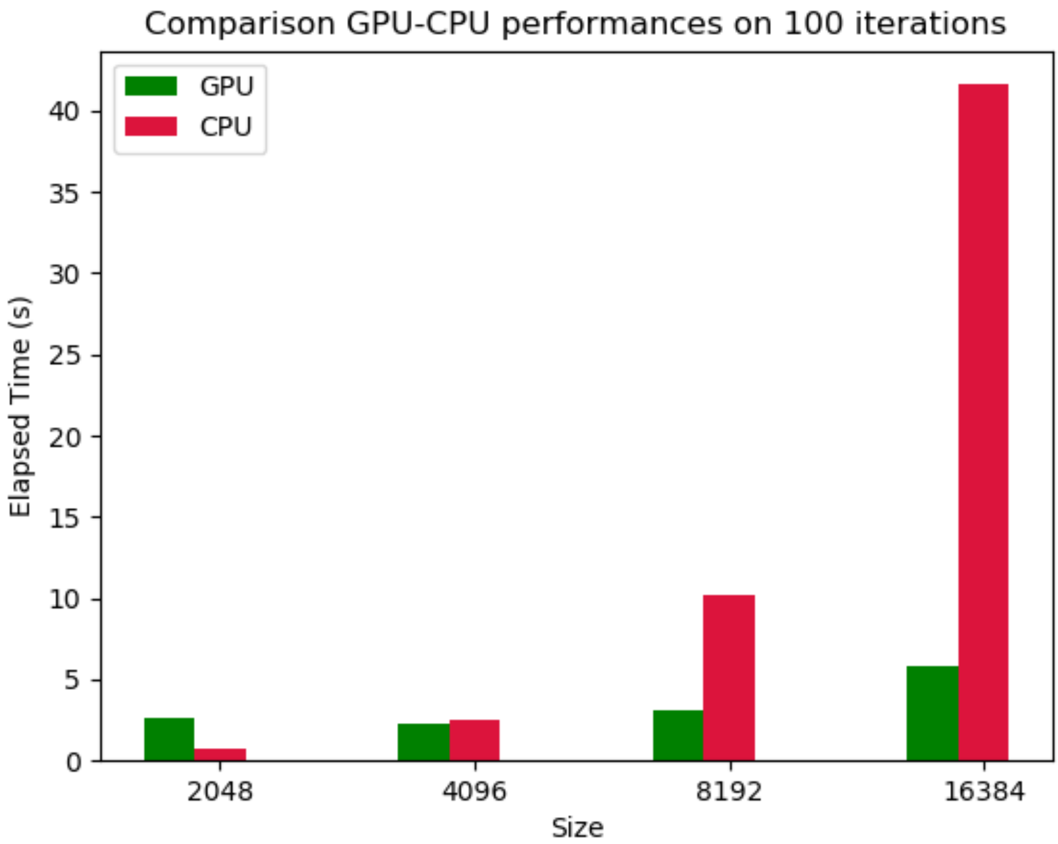
```
float laplace_step(float *in, float *out, int n)
{
  int i, j;
  float error=0.0f;
  #pragma acc data present ( in[0:n*n], out[0:n*n] )
  #pragma acc parallel loop reduction (max:error)
  for ( j=1; j < n-1; j++ )
  #pragma acc loop gang vector
    for ( i=1; i < n-1; i++ )
    {
      out[j*n+i]= 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;
}
```

We add *#pragma acc data copy* directive in the *main* function before the while clause, and the *#pragma acc data present* directive in the *laplace_step* function in order to tell to the compiler that the data which it needs for performing the function are already presenti nt the GPU memory. Using also the *#pragma acc parallel reduction* clause in the outermost loop we tell to the compiler where the parallel region starts and to share the work between the SMs of the GPU, and a *#pragma acc loop gang vector* in the innermost loop in order to share the work between the CTA and vectorize the operations: the compiler infer automatically on the dimension and use 128 element's vectors.

```
[-bash-4.2$ pgprof ./1GPU 2048 1000
Jacobi relaxation Calculation: 2048 x 2048 mesh, maximum of 1000 iterations
==1217== PGPROF is profiling process 1217, command: ./1GPU 2048 1000
Total Iterations:  1000, ERROR: 0.018806, A[16][16]= 0.011906
==1217== Profiling application: ./1GPU 2048 1000
==1217== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   94.41%  752.08ms      1000  752.08us  725.69us  970.73us  laplace_step_24_gpu
                    3.91%  31.119ms      1000  31.118us  30.241us  33.633us  laplace_step_24_gpu__red
                    0.86%  6.8496ms      1002  6.8350us  1.1840us  2.8698ms  [CUDA memcpy HtoD]
                    0.83%  6.6024ms      1004  6.5760us  1.3760us  2.5913ms  [CUDA memcpy DtoH]
      API calls:   57.77%  787.11ms      1004  783.98us  6.3960us  1.0098ms  cuMemcpyDtoHAsync
                   22.95%  312.73ms         1  312.73ms  312.73ms  312.73ms  cuDevicePrimaryCtxRetain
                   13.08%  178.18ms         1  178.18ms  178.18ms  178.18ms  cuDevicePrimaryCtxRelease
                    2.11%  28.694ms         1  28.694ms  28.694ms  28.694ms  cuMemHostAlloc
                    1.49%  20.325ms      2000  10.162us  7.1200us  168.22us  cuLaunchKernel
                    0.66%  8.9276ms         1  8.9276ms  8.9276ms  8.9276ms  cuMemFreeHost
                    0.62%  8.4076ms      2002  4.1990us     910ns  2.7412ms  cuStreamSynchronize
                    0.55%  7.4499ms      1002  7.4350us  5.7640us  46.374us  cuMemcpyHtoDAsync
                    0.32%  4.4003ms         5  880.06us  481.28us  2.1280ms  cuMemAlloc
                    0.19%  2.6086ms         6  434.77us     906ns  2.5843ms  cuEventSynchronize
                    0.17%  2.2557ms      4004     563ns     324ns  4.5630us  cuPointerGetAttributes
                    0.08%  1.1145ms         1  1.1145ms  1.1145ms  1.1145ms  cuMemAllocHost
                    0.02%  228.86us         1  228.86us  228.86us  228.86us  cuModuleLoadData
                    0.00%  37.676us         1  37.676us  37.676us  37.676us  cuStreamCreate
                    0.00%  28.501us         8  3.5620us  1.6180us  10.256us  cuEventRecord
                    0.00%  11.627us         4  2.9060us     734ns  5.6010us  cuEventCreate
                    0.00%  7.1510us         3  2.3830us     411ns  5.5040us  cuCtxSetCurrent
                    0.00%  4.2120us         3  1.4040us     361ns  3.3340us  cuDeviceGetCount
                    0.00%  2.5480us         2  1.2740us     539ns  2.0090us  cuModuleGetFunction
                    0.00%  2.1990us         6     366ns     253ns     783ns  cuDeviceGetAttribute
                    0.00%  2.1700us         4     542ns     200ns     994ns  cuDeviceGet
                    0.00%  1.9380us         1  1.9380us  1.9380us  1.9380us  cuMemFree
                    0.00%     596ns         2     298ns     192ns     404ns  cuDeviceComputeCapability
                    0.00%     286ns         1     286ns     286ns     286ns  cuCtxGetCurrent
```

As we can see from the image above, now a very little percentage of the GPU activity is devoted to move the data between the Host and the Device and almost the 100% of the work is used to perform the algorithm's core. Here the elapsed time is around 10 times better, around 2,46 seconds.

Finally we have compared the CPU and GPU performances with different problem sizes. The CPU results to perform better than the GPU for small problem sizes, instead the GPU perform much better than the CPU as bigger is the problem size. This happens because in order to give an improvement and fully exploit the GPU parallelization potential, the problem size have to be big, so each SM can have its slice of work incresing the throughput and hiding the GPU low latency. Furthemore, if the size is very small, the time spent for moving the data between host and device in the GPU will result a very big slice of the work, damaging the performances. Below is showed a graph that compare the GPU and CPU elapsed time on different size of the same problem.



Raffaele Bongo
Naci Kurdoglu