OpenAcc - Jacobi solver for linear equations
Parallel Programming

Arturo del Cerro (1593930)   Nil Bellmunt (1588664)

January 2021

1 Introduction

A code of the Jacobi solver for linear equations has been provided to us in order to fix it, optimize the accelerated code and explain the improvements we have done with performance data, time and kernel metrics. In this project all this work have been done even exploring the details and improvements for the accelerated code we generated.

2 Fixing the functionality

If we try to execute the `jsolve.cpp` program using OpenAcc we see that the result given is wrong: it converges after 1 iteration giving a solution error = `-nan` when the compiling for CPU single-thread and for CPU multi-thread gives a solution error = 0.000998928. So, as the assignment statement says, there exists a problem when executing this program using OpenAcc on `c++`.
Executing the `jsolve.cpp` program with `OpenAcc` gives the next complain error:

```
Could not find allocated-variable index for symbol A.
```

So one could think that maybe in the access of the A matrix by its index is where the problem lies. Taking a deep look at (1) and (2) we figured out that although for CPU single and multi thread executings it works well it doesn't work for OpenAcc since the access to the A matrix elements are to complex for the compiler. In concrete, we have a 2-dimensional `c++` array that is being accessed single dimensionally by

$$A[i \cdot \mathrm{nsize} + j]$$

but using a 2-dimensional expression (i, j). So, as `tesla` code warns us, when trying to GPU copying it causes problems besides the compiler does not know how many information is need to be copied from the A matrix.

3 Results

3.1 First improvement

In order to solve this problem above we need to copy from host to device the information of the A matrix. We use the `copyin` clause instead the `copy` one because we do not need to copy back from device to host. So when doing this `copyin` we specify to the compiler the size to be copied by `copyin(A[0:nsize*nsize])` in accordance with the issues about accessing the matrix we just explained. Furthermore, inside this parallelized region we add `pragma acc loop reduction(+:rsum)` for the inner `for` loop. Then we write the same `reduction` clause for `(+:residual)` for the outer and third `for` loop. On section 4 the code will be presented.
In table 1 we can see the differences of time between the different compiler options and programs within different matrix sizes and iterations for Jacobi solver for linear equations.

First improvement	nsize	iters	CPU time(s)	CPU Multicore time (s)	GPU time (s)
	500	2500	0.5	3.45	1.05
	1000	5000	3.64	14	6.83
	2000	10000	30	61	54
	4000	20000	254	302	Time computation exceeded

Table 1: Results compared for first improvement

As we can see from the results of table 1 both CPU Multicore and GPU programs are way worse than the CPU single thread in all the different sizes of the matrix and number of iterations. Besides, although the GPU program starts as more efficient in time than the CPU multicore, when increasing the metrics it get a very worse performance in comparison. But why is this happening? In theory, GPU should be more efficient in this type of problems than a single thread CPU. We analyse the profiler data in order to reach some significant conclusion.

```
Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   91.31%  341.89ms     10430  32.779us     800ns  187.46us  [CUDA memcpy HtoD]
                    5.65%  21.142ms      2086  10.135us  9.7300us  12.387us  main_109_gpu
                    0.92%  3.4505ms      2086  1.6540us  1.6320us  1.9200us  main_123_gpu
                    0.92%  3.4492ms      2086  1.6530us  1.5360us  2.0160us  main_123_gpu__red
                    0.89%  3.3418ms      4172     801ns     608ns  1.1850us  [CUDA memcpy DtoH]
                    0.31%  1.1580ms      2086     555ns     512ns     833ns  [CUDA memset]
```

So here the problem can be seen very easily. A 91.31% of execution time is being taken by copying memory from host to device, just the part of the code we changed to make it functional. Also, as we have taken 2500 iterations and the Jacobi solver converged after 2086 iterations, we can see in the profiling that this number agrees with the numbers of calls of each main kernel, but not for the number of calls to copy memory from host to device: it is 4 times larger. To understand this, we can see that every time an iteration of the while is made, the program has to allocate and copy data over the device, run the kernel and copy the results back to the device and deallocate the data. This every iteration and for each `for` loop. This is the reason why this memory copy performs so bad. Fortunately we can make changes in the code to fix this.
Now, taking a look at the grid configuration of the GPU kernels we can see that the `main_109_gpu` kernel, the first `for` loop, has a grid size and block size of `(500 1 1)` and `(128 1 1)` respectively, resulting in 64000 threads, which is a good number to work with. Besides, the `main_123_gpu`, the second outer `for` loop, has a grid configuration of `(4 1 1)` and `(128 1 1)` for grid and block size, resulting in 512 threads, way less than the previous one. It is good for our program to have this distribution of grid configuration for our kernels because as we have seen, the first `for` loop is more complex than the second one.

3.2 Second improvement

We just have seen a very big problem that our OpenAcc program has that performs way worse than a single thread CPU execution. We have that a huge amount of memory is being copied every iteration two times (each for the two `for` loops). This could be fixed copying this data - that is neccesary to have copied for the well performance of the program as we saw firstly - just before the while loop. The code before the `while` loop is something like:

```
#pragma acc data copyin(A[0:nsize*nsize], b[0:nsize]), copy(x1[0:nsize],x2[0:nsize])
```

and now we explain why this specifications. We need to copy from host to device also the `b` vector and to copy the `x1` and `x2` vectors that compute the Jacobi solution. We copy this `x1` and `x2` instead of `xold` and `xnew` because OpenAcc uses the adress of the variables, not the names (2). The rest of the code remains as before and in table 2 we can see the results for executing and studying the improvent's metrics compared.

Second improvement	nsize	iters	CPU Multicore time(s)	Improvement	GPU time(s)	Improvement
	500	2500	0.18	x2.77	0.33	x1.51
	1000	5000	1.36	x2.67	0.54	x6.74
	2000	10000	15	x2	1.39	x21.58
	4000	20000	134	x1.89	7.8	x32.56
	8000	40000			58.5	

Table 2: Results compared for second improvement

Now improvements are being noticed. We can see that due to this copying data before the `while` the GPU program improves in an uprising and non-linear trend with respect to the CPU basic code when the size of the matrix is increased. This increasing improvement is due to the time that GPU expend in starting a parallelized region: with low size values for the matrix, as the computation is fast enough then this time devoted to create parallelized regions dominates the full time. However, when size value is large enough we see the effects of the improvement. To confirm the results we are talking about we take a look to the profiler.

```
Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   69.74%  27.181ms      2086  13.030us  12.575us  16.159us  main_109_gpu
                   11.37%  4.4333ms      2086  2.1250us  2.0150us  7.2640us  main_123_gpu
                   11.23%  4.3758ms      2086  2.0970us  1.9510us  2.4960us  main_123_gpu__red
                    4.03%  1.5689ms      2088     751ns     703ns  1.0880us  [CUDA memcpy DtoH]
                    3.21%  1.2529ms      2086     600ns     575ns     896ns  [CUDA memset]
                    0.42%  164.69us         4  41.173us     959ns  161.75us  [CUDA memcpy HtoD]
```

Now almost 70% of execution time is taken by the `for` loop that manages the computation of the solution vector. So this is a good new that tells that our code improvements have done enough effect as the % of time dedicated to copying memory from host to device has been improved from a 91% to a 0.42% (x2079 in time terms). Taking a look for the grid configuration of the kernels we can see that it mantained the grid and block size and number of threads, so in this terms we have not done any improvement.
Now we could think that this is enough improved and we reached some top improving limit. Let's discuss about it below.

3.3 Third improvement

Most of OpenAcc programs can be parallelized using asynchronised `for` loops (2). In order to do not waste time when doing the `for` loops while waiting to end the first iteration to start the second one, we can make use of the `async` and `wait` clauses that let us in some manner improve the performance of our program. Writting the `async` clause in both outer `for` loops and then writting the `wait` clause after, we find this results we show in table 3.

Third improvement	nsize	iters	CPU Multicore time(s)	Improvement	GPU time(s)	Improvement
	500	2500	0.2	x2.5	0.41	x1.21
	1000	5000	1.39	x2.61	0.45	x8.11
	2000	10000	15.3	x1.96	1.29	x23.25
	4000	20000	135.3	x1.87	7.55	x32.64
	8000	40000			57	

Table 3: Results compared for thrid improvement

Here we see a curious event that happens. On one hand, we have that the OpenAcc code has been slightly improved by this asynchronised performance. Time compared with the synchronised GPU code is nearly same but improved up between 1.2 and 1.02 times. On the other hand we have that this asynchronised performance does not improves the multicore compiling. It worsens the performance by a 0.9 to 0.99 factor. We wondered why this could happen and maybe on GPU while waiting the GPU is able to do some other work but the CPU multicore do not work at all until all the processes has arrived.
We are not going to show the profiling data because it is nearly exact as the second improvement's one. As we said, we reduced time between kernels, not improved the performance of the kernel. In section 4 we show the code with all the improvements made until here. As there have not been any structural changes between second and third improvement, the grid configuration have been remained as the same. Finally we are showing in figure 1 a graph with all this improvements compared. As we can see, it is in accordance with all we mentioned before.

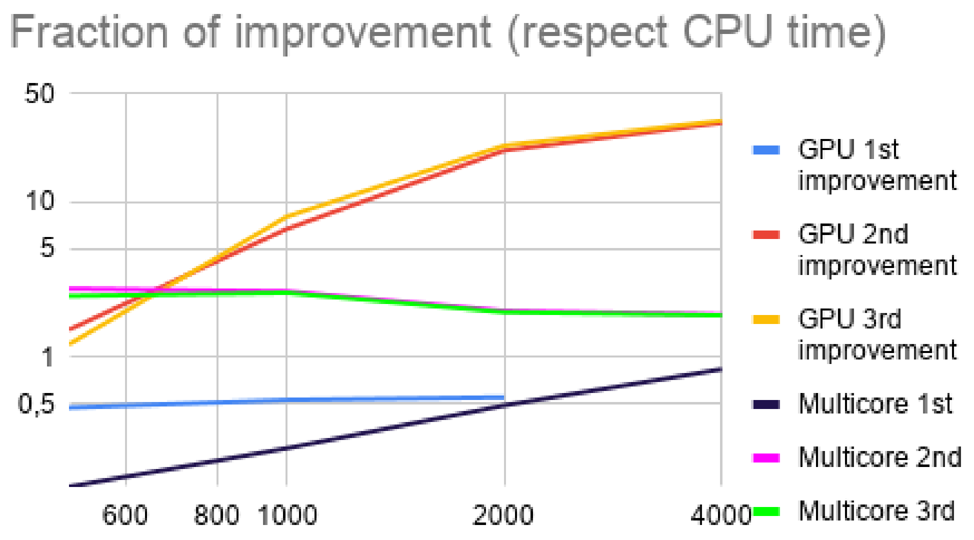

Figure 1: Improvement respect the CPU time for the 3 improvements done in GPU and CPU multicore. Both x and y axis are in logarithmic scale

4 Code

Here we present the final improved and accelerated code for third improvement.

```
#pragma acc data copyin(A[0:nsize*nsize], b[0:nsize]), copy(x1[0:nsize],x2[0:nsize]) {
while ((residual > TOLERANCE) && (iters < max_iters)) {
  ++iters;
  xtmp = xnew; xnew = xold;  xold = xtmp;
  #pragma acc parallel loop copyin(A[0:nsize*nsize]) async
  for (i = 0; i < nsize; ++i) {
    TYPE rsum = (TYPE)0;
    #pragma acc loop reduction(+:rsum)
    for (j = 0; j < nsize; ++j) {
      if (i != j)
        rsum += A[i*nsize + j] * xold[j];
    }
    xnew[i] = (b[i] - rsum) / A[i*nsize + i];
  }
  // test convergence, sqrt(sum((xnew-xold)**2))
  residual = 0.0;
  #pragma acc parallel loop reduction(+:residual) async
  for (i = 0; i < nsize; i++) {
    TYPE dif = xnew[i] - xold[i];
    residual += dif * dif;
  }
  #pragma acc wait
  residual = sqrt((double)residual);
  if (iters % riter == 0 ) cout << "Iteration " << iters << ", residual is " << residual << "\n"
  ;
}}
```

5 Conclusions

In this project we have studied the different metrics and times GPU provide us through all improvements of the Jacobi solver for linear equations code. In a first moment, we saw the non-functionality of the OpenAcc code due to `c++` and `OpenAcc` issues, then fixed it and saw the worst performance compared with a single thread CPU. As this was not the expected result by the second and third improvement we saw how our code perform in a highly efficient way to solve what we need to. Through all this report this different performances have been explained using the profiler in order to understand what is happening.

References

[1] C++ https://www.cplusplus.com/
[2] OpenAcc https://www.openacc.org/