EDISON

Electromagnetic Design of
flexIble SensOrs

FNP Team-Tech

Acceleration of classical Gram-Schmidt algorithm with reorthogonalization

Adam Dziekoński
June 22, 2018

European Funds Smart Growth

FNP Foundation for Polish Science

European Union
European Regional Development Fund

The „EDISOn - Electromagnetic Design of flexIble SensOrs" project, agreement no TEAM TECH/2016-1/6, is carried out within the TEAM-TECH programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

Revision	Date	Author(s)	Description
1.0	27.04.2018	Adam Dziekoński	created
1.1	07.05.2018	Adam Dziekoński	updated
1.2	22.06.2018	Adam Dziekoński	updated
1.3	16.07.2018	Adam Dziekoński	updated

Contents

1 Introduction

In this report a classical Gram-Schmidt algorithm with reorthogonalization (CGS-RO) [1] is accelerated with OpenACC and OpenMP v4.5.

The C/C++ implementation of the algorithm is presented in Listing 1. One can distinguish two main loops:

1. outer (lines 4-27) over columns of the matrix **A**
2. inner (lines 8-22) in which reothogonalization is performed.

There are several loops that can be parallelized: reduction (lines 12-14, 19-20), copy (lines 4-5,6-7,9-10), axpy (lines 15-18) and scal (lines 24-25). Thus, OpenACC and OpenMP v4.5 were used to parallelize computations on a CPU and a GPU (Listing 2). However, in order to significantly save memory, which is crucial from the utilization of a GPU point of view, minimize the transfer between a CPU and a GPU and achieve better performance on a GPU three key modifications were applied:

- originally, the size of matrix **v** is m x n x steps, where m and are the number of rows and columns of matrix **A** and steps is the number of reorthogonalization steps. In the approach propose in this report the size of matrix **v** was significantly reduced since matrix **v** requires only m x n x 2 (doubles or complex doubles). In this case, the temporary results of **Q** are stored in **v** through the entire Gram-Schmidt orthogonalization process and final update of **Q** is done for the last column after reorthogonalization is finished (lines 83-94). To make it possible an additional table of relatively smaller size is needed (**tab_denominator**).
- additional pragmas were applied (lines 2, 3, 97) in order to reduce the transfers between a CPU and a GPU
- to achieve a significantly better performance on a GPU the original loop (lines 19-20, Listing 2) in which 'a new' **v** is calculated was divided into two stages (the first stage (Listing 4, lines 47-56) and the second stage (Listing 4, lines 57-68), respectively). To make it possible an additional table of relatively smaller size is needed (**tab_tmp1**). Moreover, in the second stage the order of performing computations was changed (outer loop over rows, inner over columns), which allowed achieving better results on a GPU.

What distinguish the proposed implementation in Listing 4 from the implementations from Listing 2 and implementation proposed in [2] is the higher parallelization of the computations. In fact, in the implementation from Listing 2 the outer loop (lines: 14 - 21) is not paralellized and two inner loops are parallelized. This may be beneficial in case the matrix **A** has significantly fewer columns than rows. In [2] only the outer loop (Listing 1, lines 11-17) is parallelized, while two inner loops are not parallelized. Thanks to the approach (rearrangement of computations and dividing one of the loops into two stages (loops)) proposed above (third optimization item) in this report both outer and inner loops are paralellized. In fact it occurred that the implementation from Listing 2 is more suitable for a CPU, while implementation from Listing 4 is dedicated for a GPU.

Algorithm 1 Implementation of the classical Gram-Schmidt algorithm with reorthogonalization (CGS-RO).

```
1. Inputs: A[m * n x 1],Q[m * n x 1]=[0], v [m * n x (steps+1)]=[0], aj [m x 1], //Setup: Q = [0], v = [0]
2. double sqrttmp = 0.0;
3. for ( int j = 0; j < n; j++){
4.    for (int row = 0; row < m; row++)
5.       aj[row] = A[row + j*m];
6.    for (int row = 0; row < m; row++)
7.       v[row + j*m ] = aj[row];
8.    for ( k = 0; k < steps; k++){ // re-orthogonalization loop
9.       for (int i = 0; i < m; i++)
10.         v[i + j*m + (k+1)*m*n] = v[i + j*m + k*m*n ] ;
11.      for ( i = 0; i <= j-1; i++){
12.         for ( row = 0; row < m; row++{
13.            tmp1 += Q[row + i*m] * v[row + j*m + k*m*n];
14.         }
15.         for ( row = 0; row < m; row++)
16.            v[row + j*m+ (k+1)*m*n] = v[row + j*m + (k+1)*m*n] - tmp1*Q[row + i*m];
17.      }
18.      double tmp = 0.0;
19.      for ( row = 0; row < m; row++)
20.         tmp += v[row+j*m+(k+1)*m*n] * v[row+j*m+(k+1)*m*n] ;
21.      sqrttmp = sqrt ( tmp );
22.   } // end of re-orthogonalization
23.   k–;
24.   for ( int row = 0; row < m; row++){
25.      Q[row + j*m] = v[row+j*m+(k+1)*m*n]/sqrttmp;
26.   }
27.} // end loop over columns
28. Output: Q
```

Algorithm 2 OpenACC based implementation of the classical Gram-Schmidt algorithm with re-orthogonalization (CGS-RO).

```
1. Inputs: A[m * n x 1],Q[m * n x 1]=[0], v [m * n x (steps+1)]=[0], aj [m x 1], //Setup: Q = [0], v = [0]
2.    for ( j = 0; j < n; j++){
3.    #pragma acc parallel loop
4.    for (int row = 0; row < m; row++)
5.       aj[row] = A[row + m*j];
6.    #pragma acc parallel loop
7.    for (int row = 0; row < m; row++)
8.       v[row + j*m ] = aj[row] ;
9.    for ( k = 0; k < steps; k++){ // re-orthogonalization loop
10.       double sqrttmp = 0.0;
11.       #pragma acc parallel loop
12.       for (int row = 0; row < m; row++)
13.          v[row + j*m + (k+1)*(m*n) ] = v[row + j*m + k * (m*n) ] ;
14.      for ( i = 0; i <= j-1; i++){
15.         #pragma acc parallel loop reduction(+:tmp1)
16.         for ( int row = 0; row < m; row++)
17.            tmp1 += Q[row+i*m] * v[row + j*m + k*m*n];
18.         #pragma acc parallel loop
19.         for ( int row = 0; row < m; row++)
20.            v[row + j*m + (k+1)*m*n ] = v[row + j*m + (k+1)*m*n ] - tmp1*Q[row + i*m];
21.    }
22.    #pragma acc parallel reduction(+:tmp)
23.    for ( row = 0; row < m; row++)
24.       tmp += v[row + j*m + (k+1)*m*n ] * v[row + j*m + (k+1)*m*n ] ;
25.    sqrttmp = sqrt(tmp);
26.   } // end of re-orthogonalization
27.   k–;
28.   #pragma acc parallel loop
29.   for ( row = 0; row < m; row++)
30.      Q[row+j*m] = v[row + j*m + (k+1)*m*n]/sqrttmp;
31.} // end loop over columns
32. Output: Q
```

Algorithm 3 OpenMP based implementation of the classical Gram-Schmidt algorithm with reorthogonalization (CGS-RO).

```
1. Inputs: A[m * n x 1],Q[m * n x 1]=[0], v [m * n x (steps+1)]=[0], aj [m x 1], //Setup: Q = [0], v = [0]
2.    for ( j = 0; j < n; j++){
3.    #pragma omp parallel for
4.    for (int row = 0; row < m; row++)
5.       aj[row] = A[row + m*j];
6.    #pragma omp parallel for
7.    for (int row = 0; row < m; row++)
8.       v[row + j*m ] = aj[row] ;
9.    for ( k = 0; k < steps; k++){ // re-orthogonalization loop
10.       double sqrttmp = 0.0;
11.       #pragma omp parallel for
12.       for (int row = 0; row < m; row++)
13.          v[row + j*m + (k+1)*(m*n) ] = v[row + j*m + k * (m*n) ] ;
14.      for ( i = 0; i <= j-1; i++){
15.         #pragma omp parallel for reduction(+:tmp1)
16.         for ( int row = 0; row < m; row++)
17.            tmp1 += Q[row+i*m] * v[row + j*m + k*m*n];
18.         #pragma omp parallel for
19.         for ( int row = 0; row < m; row++)
20.            v[row + j*m + (k+1)*m*n ] = v[row + j*m + (k+1)*m*n ] - tmp1*Q[row + i*m];
21.    }
22.    #pragma omp parallel for reduction(+:tmp)
23.    for ( row = 0; row < m; row++)
24.       tmp += v[row + j*m + (k+1)*m*n ] * v[row + j*m + (k+1)*m*n ] ;
25.    sqrttmp = sqrt(tmp);
26.   } // end of re-orthogonalization
27.   k–;
28.   #pragma omp parallel for reduction
29.   for ( row = 0; row < m; row++)
30.      Q[row+j*m] = v[row + j*m + (k+1)*m*n]/sqrttmp;
31.} // end loop over columns
32. Output: Q
```

Algorithm 4 OpenACC based implementation of the classical Gram-Schmidt algorithm with re-orthogonalization (CGS-RO) dedicated for a GPU accelerator.

```
0: OpenACC based implementation of the classical Gram-Schmidt algorithm with reorthogonalization (CGS-RO) dedicated for a
   GPU accelerator.
   1. Inputs: Q[m * n]=[0], v [m * n]=A, aj [m x 1], tab_tmp1[n] =[0], tab_denominator[n]=[0]
   2. #pragma acc enter data copyin(v[0:m*n], aj[0:m], tab_tmp1[0:n],tab_denominator[0:n])
   3. #pragma acc data copy(Q[0:m*n])
   4. for ( j = 0; j < n; j++){
   5.    #pragma acc kernels
   6.{
   7.    #pragma acc loop independent
   8.    for (int row = 0; row < m; row++)
   9.       aj[row] = v[row + m*j];
   11.}
   12.    for ( k = 0; k < steps; k++){ // re-orthogonalization loop
   13.       double sqrttmp = 0.0;
   14.       if (k==0)
   15.          # pragma acc kernels
   16. {
   17.          # pragma acc loop independent
   18.          for (int row = 0; row < m; row++)
   19.             Q [row + j*m ] = v [row + j*m ] ;
   21. }
   22.       else
   23.          # pragma acc kernels
   24. {
   25.          # pragma acc loop independent
   26.          for (int row = 0; row < m; row++)
   27.             v [row + j*m ] = Q [row + j*m ] ;
   29. }
   30.       # pragma acc kernels
   31. {
   32.       # pragma acc loop independent
   33.       for(int tt = 0; tt < j; tt++)
   34.          tab_tmp1[tt] = 0.0;
   35. }
   36.       # pragma acc kernels
   37. {
   38.       # pragma acc loop independent
   39.       for ( int i = 0; i <= j-1; i++){
   40.          double tmp1 = 0.0;
   41.          # pragma acc loop reduction(+:tmp1)
   42.          for ( int rowi = 0; rowi < m; rowi++)
   43.             tmp1 += (Q[rowi+i*m]) * ( v[rowi + j*m ] ) ;
   44.          tab_tmp1[i] = tmp1*tab_denominator[i];
   45.       }
   46. }
   47.       // First stage:
   48.       #pragma acc kernels
   49. {
   50.       #pragma acc loop independent device_type(nvidia)
   51.       for ( int i = 0; i <= j-1; i++){
   52.          #pragma acc loop independent device_type(nvidia)
   53.          for ( int rowi = 0; rowi < m; rowi++)
   54.             v[rowi + i*m ] = tab_tmp1[i]* (Q[rowi + i*m] *tab_denominator[i]) ;
   55.       }
   56. }
   57.       // Second stage: (reduction)
   58.       #pragma acc kernels
   59. {
   60.       #pragma acc loop independent device_type(nvidia)
   61.       for ( int rowi = 0; rowi < m; rowi++){
   62.          double tmpx = 0.0;
   63.          #pragma acc loop reduction(+:tmpx)
   64.          for ( int i = 0; i <= j-1; i++)
   65.             tmpx += v[rowi + i*m ];
   66.          Q[rowi + j*m ] -= tmpx; //v[rowi + i*m + (k)*m*n ];
   67.       }
   68. }
   =0
```

```
0: Algorithm continued here...
   69.      double tmp = 0.0;
   70.      #pragma acc kernels
   71. {
   72.      # pragma acc loop reduction(+:tmp)
   73.      for ( int row = 0; row < m; row++)
   74.        tmp += v[row + j*m ] * v[row + j*m ] ;
   75. }
   76.      sqrttmp = sqrt(tmp);
   77.    } // end re-orthogonalization
   78.    k–;
   79.    # pragma acc kernels
   80. {
   81.    tab_denominator[j] = 1.0/sqrttmp;
   82. }
   83.    if ((j+1==n)){ // final update of Q
   84.      #pragma acc kernels 85. {
   86.      double sub_tab;
   87.      #pragma acc loop independent
   88.      for ( int jj = 0; jj < n; jj++){
   89.        sub_tab = tab_denominator[jj];
   90.        #pragma acc loop independent
   91.        for ( row = 0; row < m; row++)
   92.          Q[row+jj*m ] = Q[row+jj*m] * sub_tab;
   93.      }
   94. }
   95.    }
   96. } // end loop over columns
   97. #pragma acc exit data delete (tab_denominator, tab_tmp1, aj, v)
   98. Output: Q
   =0
```

2 OpenACC

In this section performance of the CGS-RO implementations (dedicated for real-valued and complex-valued matrices) based on the OpenACC parallelization is discussed.

1. Even if the proposed modifications allowed reducing memory requirements for real valued problems with m=6M and n = 150 (and larger) could not be analyzed with Tesla Pascal P100 due to the lack of memory. Also complex-valued problems {(m x n)} = {(3M x 150), (6M x 100), (6M x 150)} required more memory than it is available on a graphics accelerator equipped with 12 GB.

2. Figures 1 i 2 show the acceleration achieved with OpenACC over a reference sequential implementation for CGS algorithm with no and with one additional re-orthogonalizations:

   - a blue line is assigned for the acceleration obtained with a OpenACC directives executed on a CPU (12 cores, 12 threads) (Listing 2)
   - red and orange lines are assigned for the acceleration obtained with OpenACC directives executed on a Tesla K40 and Tesla P100, respectively (Listing 4). Due to the rearrangement of computations significant performance improvement is observed with the increase of the number of columns (n). For example: for a test problem (m=3M x n=1) the acceleration is only 1.5x over the sequential CPU code, while for a test problem (m=3M x n=150) the speedup by a factor of 14 was obtained.

3. For real-valued problems if the number of columns of matrix **A** is smaller than 30 then one may observe better performance on a CPU than on a GPU thanks to OpenACC directives. For larger number of columns a GPU-accelerated implementation achieves much better performance (Figs. 3 and 4)

4. All figures consider times of transferring data to a GPU and back to a CPU. Due to more computations performed in variant with reorthogonalization higher accelerations were obtained for a CGS-RO over a CGS since the overhead caused by data transfers less significant.

Algorithm 5 OpenMP v4.5 based implementation of the classical Gram-Schmidt algorithm with reorthogonalization (CGS-RO) dedicated for a GPU accelerator.

```
0: .
1. Inputs: Q[m * n]=[0], v [m * n]=A, aj [m x 1], tab_tmp1[n] =[0], tab_denominator[n]=[0]
2. #pragma omp target enter data map(to:v[0:m*n], to:aj[0:m], to:tab_tmp1[0:n],to:tab_denominator[0:n])
3. #pragma omp target data map(to:Q[0:m*n]) map(tofrom:tmp1)
4. for ( j = 0; j < n; j++){
5.    #pragma omp targets
6.{
7.    #pragma omp teams distribute parallel for simd
8.    for (int row = 0; row < m; row++)
9.      aj[row] = v[row + m*j];
11.}
12.    for ( k = 0; k < steps; k++){ // re-orthogonalization loop
13.      double sqrttmp = 0.0;
14.      if (k==0)
15.        # pragma omp targets
16. {
17.        #pragma omp teams distribute parallel for simd
18.        for (int row = 0; row < m; row++)
19.          Q [row + j*m ] = v [row + j*m ] ;
21. }
22.      else
23.        # pragma omp targets
24. {
25.        # pragma omp teams distribute parallel for simd
26.        for (int row = 0; row < m; row++)
27.          v [row + j*m ] = Q [row + j*m ] ;
29. }
30.      #pragma omp targets
31. {
32.      #pragma omp teams distribute parallel for
33.      for(int tt = 0; tt < j; tt++)
34.        tab_tmp1[tt] = 0.0;
35. }
36.
37. {
38.      #pragma omp parallel for reduction
39.      for ( int i = 0; i <= j-1; i++){
40.        double tmp1 = 0.0;
41.        #pragma omp parallel for reduction(+:tmp1)
42.        for ( int rowi = 0; rowi < m; rowi++)
43.          tmp1 += (Q[rowi+i*m]) * ( v[rowi + j*m ] ) ;
44.        tab_tmp1[i] = tmp1*tab_denominator[i];
45.      }
46. }
47.      // First stage:
48.
49. {
50.      #pragma omp target teams distribute parallel for collapse(2) schedule(static,1)
51.      for ( int i = 0; i <= j-1; i++){
52.
53.        for ( int rowi = 0; rowi < m; rowi++)
54.          v[rowi + i*m ] = tab_tmp1[i]* (Q[rowi + i*m] *tab_denominator[i]) ;
55.      }
56. }
57.      // Second stage: (reduction)
58.
59. {
60.      #pragma omp target teams distribute parallel for simd
61.      for ( int rowi = 0; rowi < m; rowi++){
62.        double tmpx = 0.0;
63.        #pragma omp reduction(+:tmpx)
64.        for ( int i = 0; i <= j-1; i++)
65.          tmpx += v[rowi + i*m ];
66.        Q[rowi + j*m ] -= tmpx; //v[rowi + i*m + (k)*m*n ];
67.      }
68. }
=0
```

5. For complex-valued problems (Figs 5-8) very similar conclusions can be drawn, however, one may observe that GPU-accelerated implementations achieve better performance than CPU (OpenACC)-accelerated starting from problems with 10 and more columns.

```
0: Algorithm continued here...
69.      pragma omp target teams distribute map(tofrom:tmp)
70.      for(int q = 0; q < 1; q++)
71.           tmp = 0.0;
72.      # pragma omp target teams distribute parallel for reduction(+:tmp) map(tofrom:tmp)
73.      for ( int row = 0; row < m; row++){
74.         tmp += v[row + j*m ] * v[row + j*m ] ;
75.      }
76.      sqrttmp = sqrt(tmp);
77.    } // end re-orthogonalization
78.    k–;
79.    # pragma omp target teams distribute map(tofrom:sqrttmp)
80.      for(int q = 0; q < 1; q++) {
81.        tab_denominator[j] = 1.0/sqrttmp;
82.      }
83.    if ((j+1==n)){ // final update of Q
84.
85. {
86.
87.       #pragma omp target teams distribute
88.       for ( int jj = 0; jj < n; jj++){
89.          sub_tab = tab_denominator[jj];
90.          #pragma omp parallel for
91.          for ( row = 0; row < m; row++)
92.             Q[row+jj*m ] = Q[row+jj*m] * sub_tab;
93.       }
94. }
95.    }
96. } // end loop over columns
97. Output: Q
=0
```

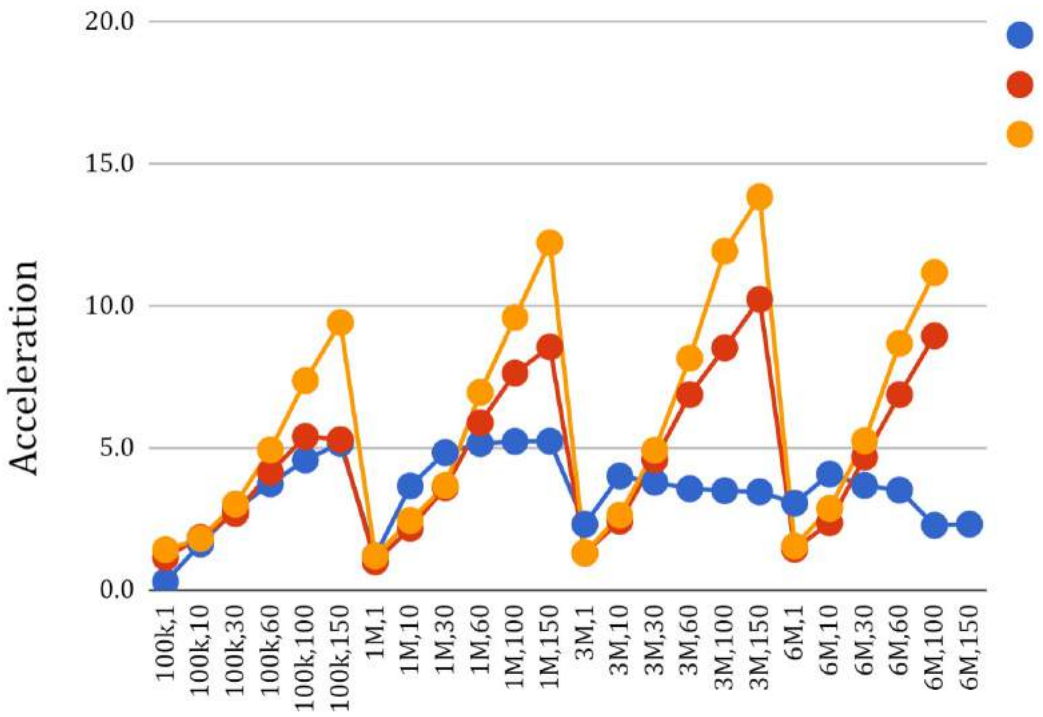

Figure 1: Acceleration achieved thanks to the OpenACC on Xeon E5-2680 v3 (12 cores), GPU Tesla K40 and Tesla Pascal P100 over a sequential code. Algorithm: CGS. Matrix type: real-valued. Colors: blue - CPU parallel, red - K40, orange - P100.

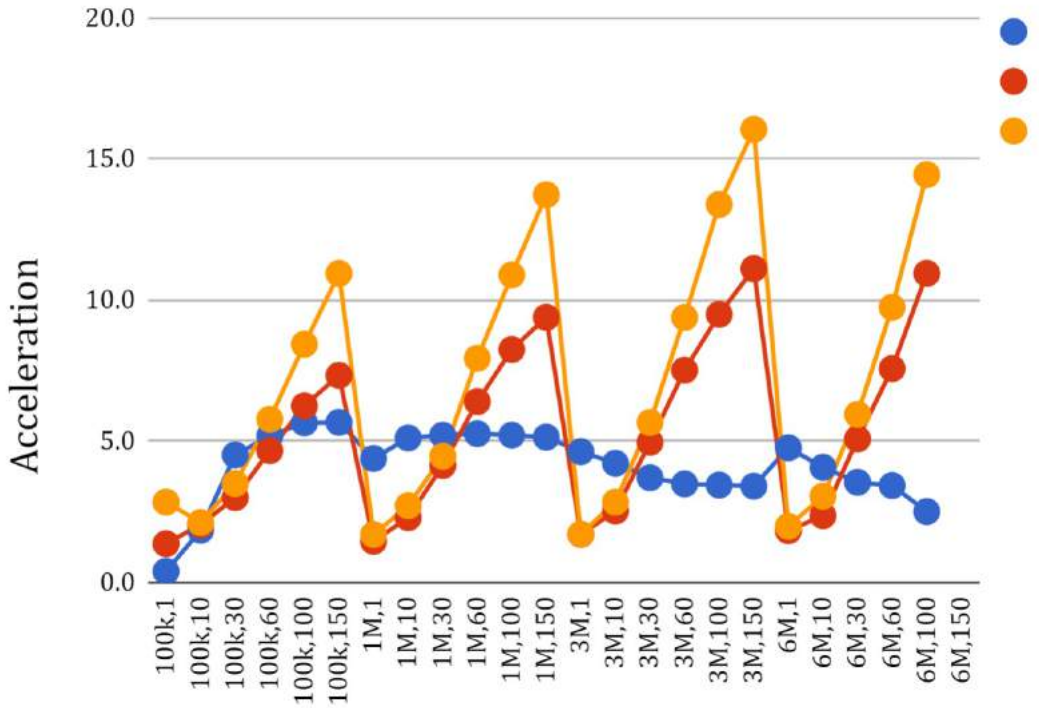

Figure 2: Acceleration achieved thanks to the OpenACC on Xeon E5-2680 v3 (12 cores), GPU Tesla K40 and Tesla Pascal P100 over a sequential code. Algorithm: CGS with one reorthogonalization. Matrix type: real-valued. Colors: blue - CPU parallel, red - K40, orange - P100.

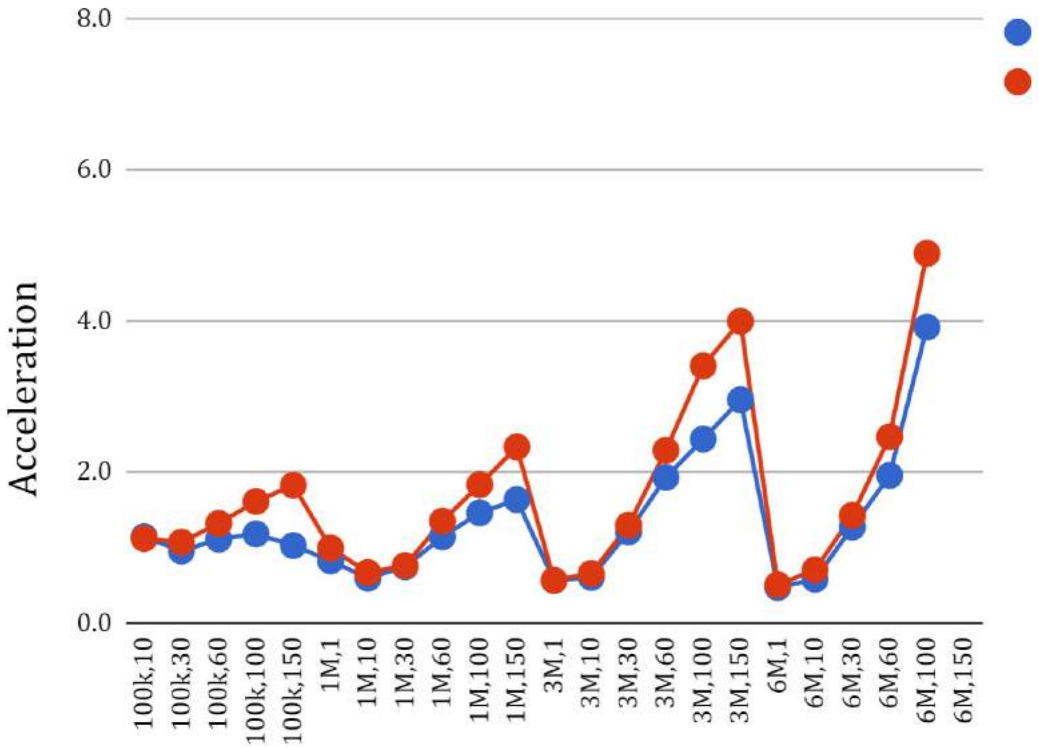

Figure 3: Acceleration of a GPU over a CPU implementations which both take advantage of a OpenACC. CPU: Xeon E5-2680 v3 (12 cores), GPU: Tesla K40, Tesla Pascal P100. Algorithm: CGS. Matrix type: real-valued. Colors: blue - K40 over CPU parallel, red - P100 over CPU parallel

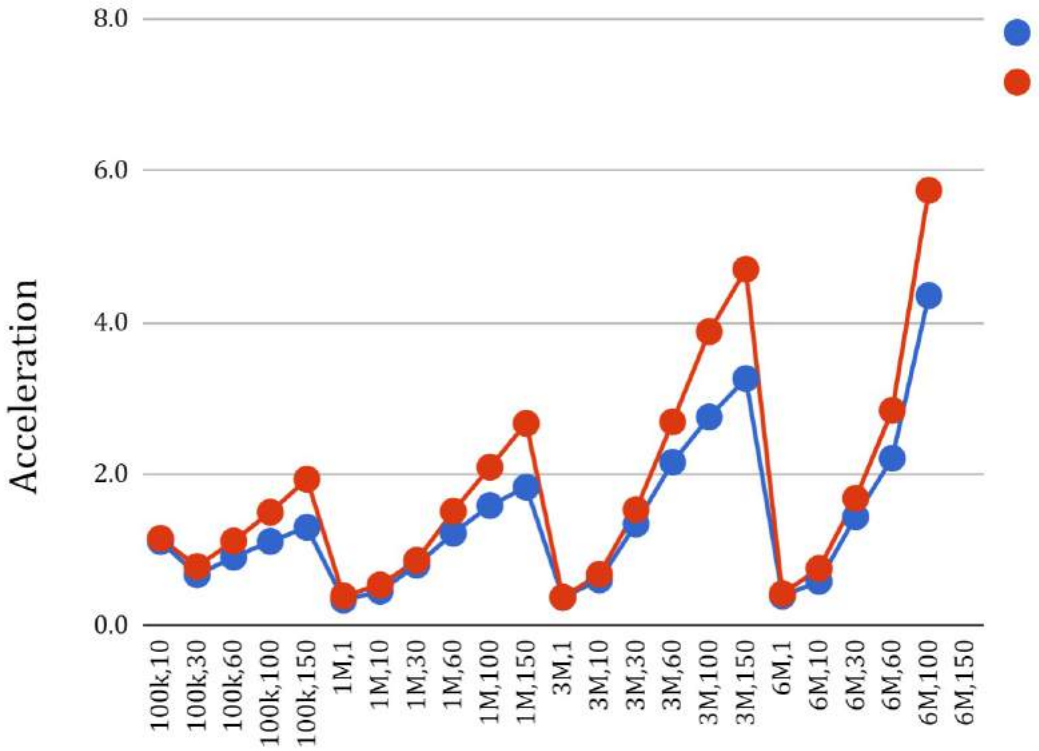

Figure 4: Acceleration of a GPU over a CPU implementations which both take advantage of a OpenACC. CPU: Xeon E5-2680 v3 (12 cores), GPU: Tesla K40, Tesla Pascal P100. Algorithm: CGS with one reorthogonalization. Matrix type: real-valued. Colors: blue - K40 over CPU parallel, red - P100 over CPU parallel.

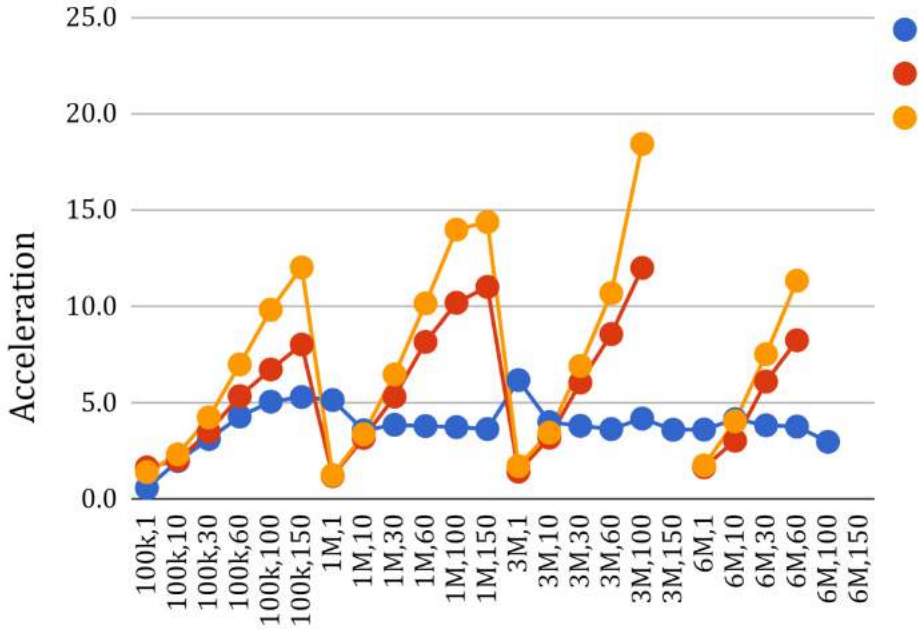

Figure 5: Acceleration achieved thanks to the OpenACC on Xeon E5-2680 v3 (12 cores), GPU Tesla K40 and Tesla Pascal P100 over a sequential code. Algorithm: CGS. Matrix type: complex-valued. Colors: blue - CPU parallel, red - K40, orange - P100.

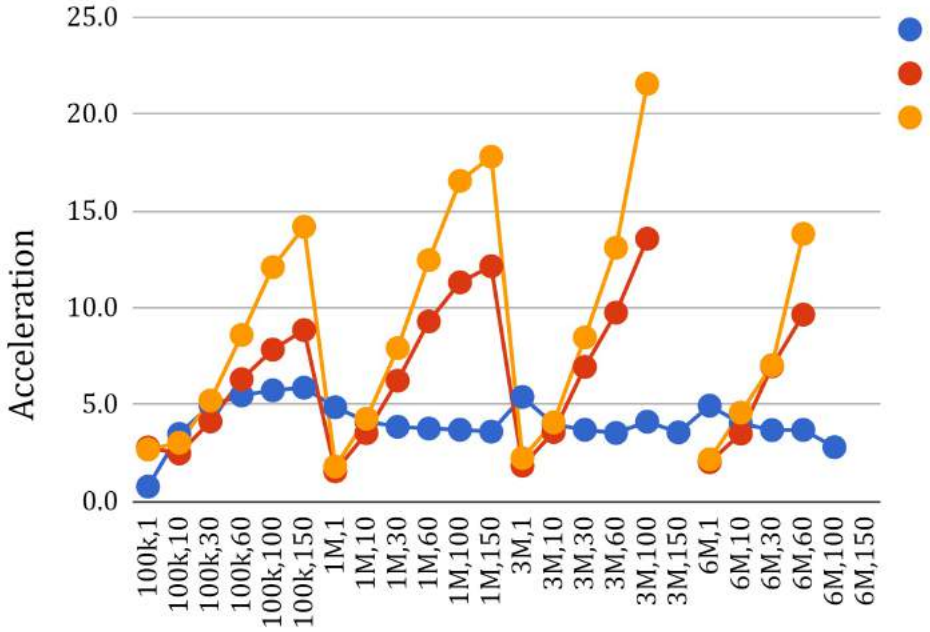

Figure 6: Acceleration achieved thanks to the OpenACC on Xeon E5-2680 v3 (12 cores), GPU Tesla K40 and Tesla Pascal P100 over a sequential code. Algorithm: CGS with one reorthogonalization. Matrix type: complex-valued. Colors: blue - CPU parallel, red - K40, orange - P100.

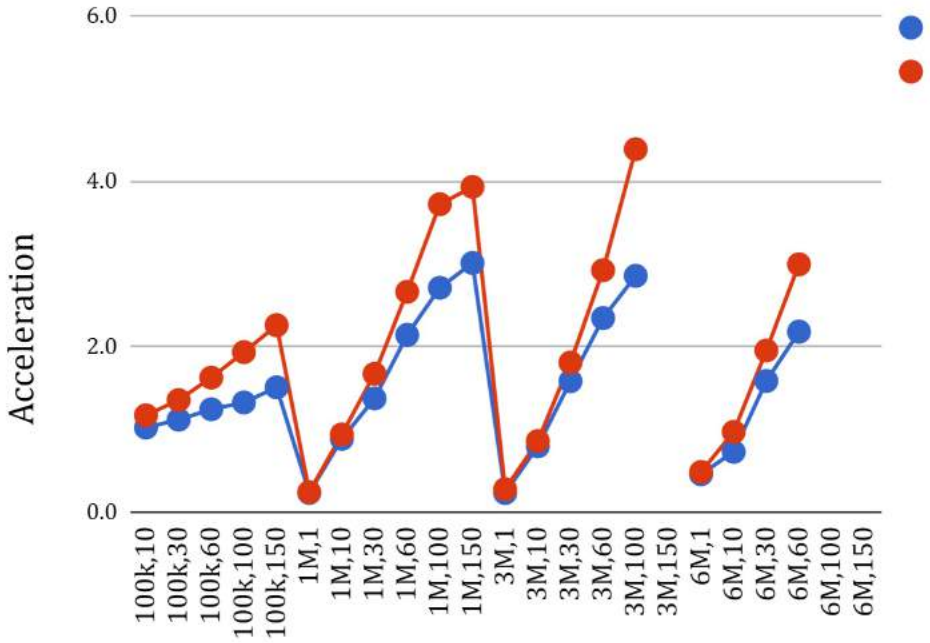

Figure 7: Acceleration of a GPU over a CPU implementations which both take advantage of a OpenACC. CPU: Xeon E5-2680 v3 (12 cores), GPU: Tesla K40, Tesla Pascal P100. Algorithm: CGS. Matrix type: complex-valued. Colors: blue - K40 over CPU parallel, red - P100 over CPU parallel.

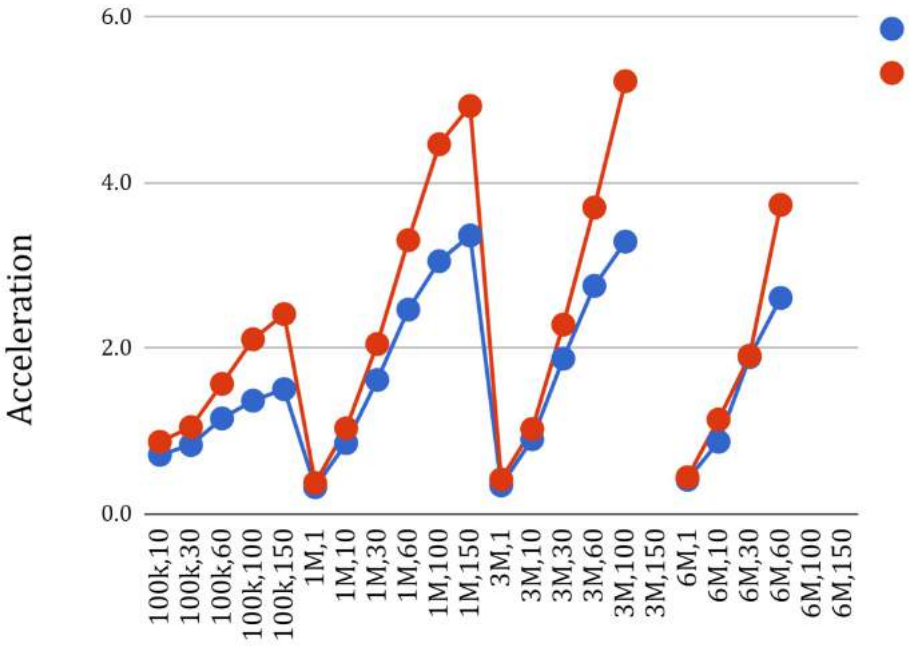

Figure 8: Acceleration of a GPU over a CPU implementations which both take advantage of a OpenACC. CPU: Xeon E5-2680 v3 (12 cores), GPU: Tesla K40, Tesla Pascal P100. Algorithm: CGS with one reorthogonalization. Matrix type: complex-valued. Colors: blue - K40 over CPU parallel, red - P100 over CPU parallel.

3 OpenMP v4.5

In this section results obtained for codes written with OpenMP directives are presented. One may observe that GPU offloading of computations allowed achieving acceleration over a sequential reference not higher than 2 (Figs. 9-10 - real valued, Figs. 13-14 - complex valued). Moreover, Figures 11-12 (real-valued) and Figures 15-16 (complex-valued) show that CPU-based mulithreaded CGS and CGS-RO implementations obtain better performance than offloaded GPU-based implementations. The reasons why the performance is so poor is presented in Section 4 where one can find that the capabilities of a graphics accelerator are not sufficiently used.

Listing 5 shows the implementation of the CGS-RO with OpenMP v4.5 dedicated for offloading. The implementation is based on implementation from Listing 4, however, OpenACC directives were replaced with OpenMP v4.5 directives according to [4, 5].

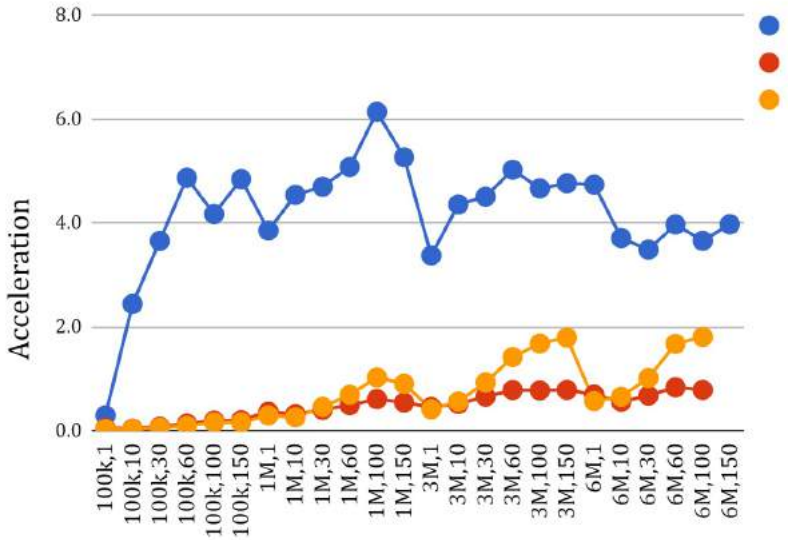

Figure 9: Acceleration achieved thanks to the OpenMPv4.5 on Xeon E5-2680 v3 (12 cores), GPU Tesla K40 and Tesla Pascal P100 over a sequential code. Algorithm: CGS. Matrix type: real-valued. Colors: blue - CPU parallel, red - K40, orange - P100.

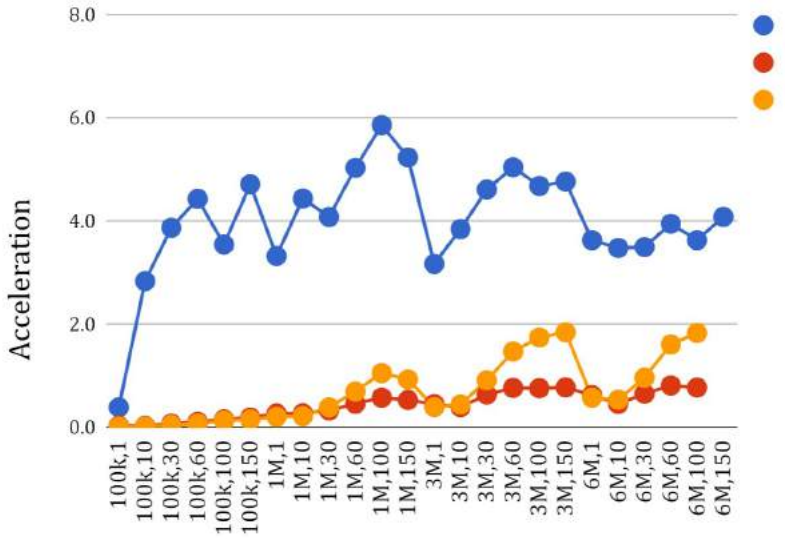

Figure 10: Acceleration achieved thanks to the OpenMPv4.5 on Xeon E5-2680 v3 (12 cores), GPU Tesla K40 and Tesla Pascal P100 over a sequential code. Algorithm: CGS with one reorthogonalization. Matrix type: real-valued. Colors: blue - CPU parallel, red - K40, orange - P100.

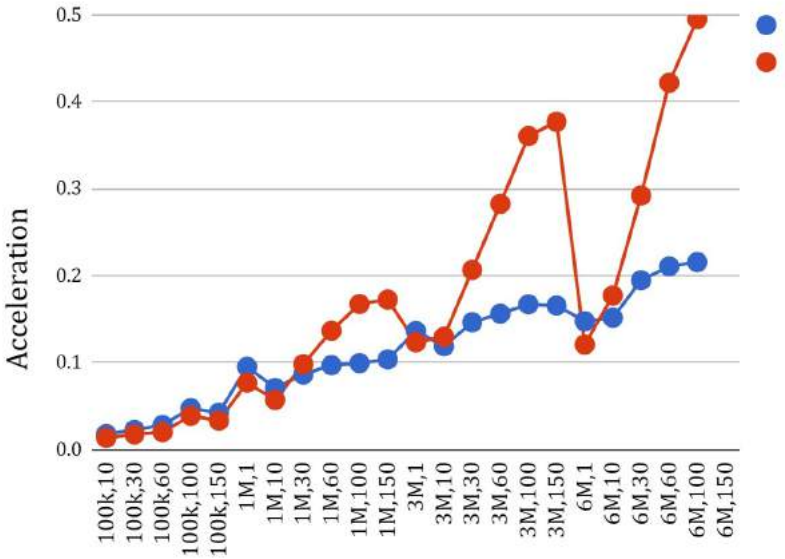

Figure 11: Acceleration of a GPU over a CPU implementations which both take advantage of a OpenMPv4.5. CPU: Xeon E5-2680 v3 (12 cores), GPU: Tesla K40, Tesla Pascal P100. Algorithm: CGS. Matrix type: real-valued. Colors: blue - K40 over CPU parallel, red - P100 over CPU parallel

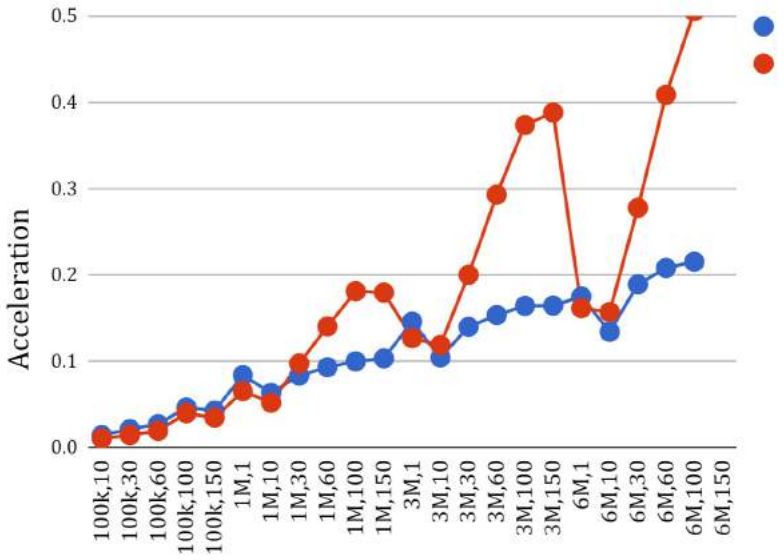

Figure 12: Acceleration of a GPU over a CPU implementations which both take advantage of a OpenMPv4.5. CPU: Xeon E5-2680 v3 (12 cores), GPU: Tesla K40, Tesla Pascal P100. Algorithm: CGS with one reorthogonalization. Matrix type: real-valued. Colors: blue - K40 over CPU parallel, red - P100 over CPU parallel.

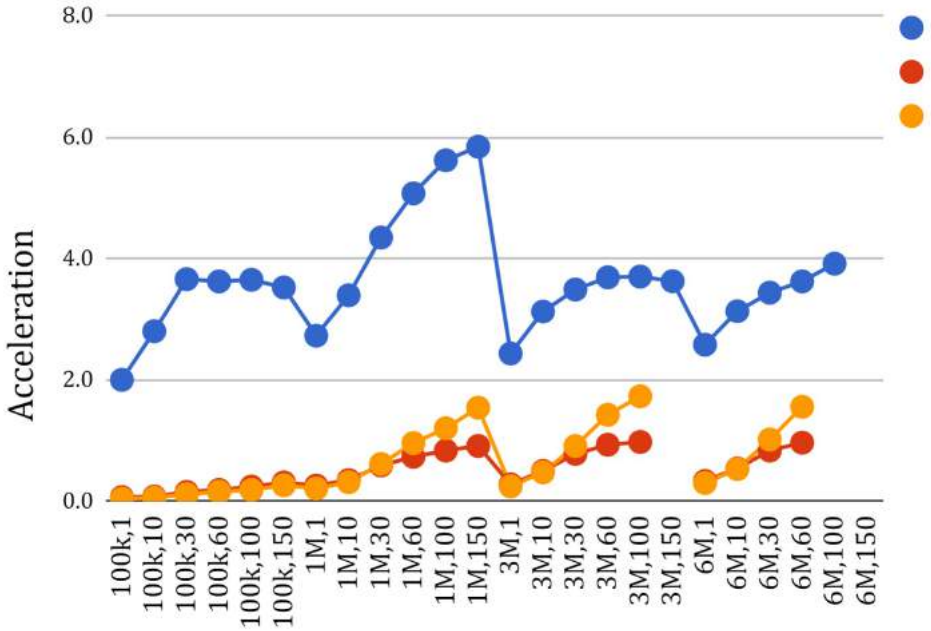

Figure 13: Acceleration achieved thanks to the OpenMPv4.5 on Xeon E5-2680 v3 (12 cores), GPU Tesla K40 and Tesla Pascal P100 over a sequential code. Algorithm: CGS. Matrix type: complex-valued. Colors: blue - CPU parallel, red - K40, orange - P100.

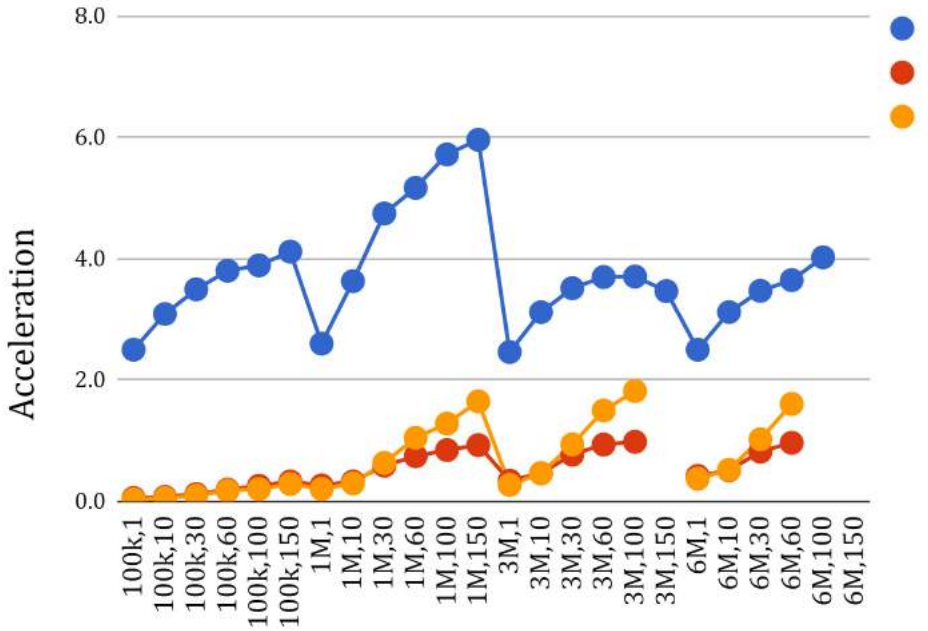

Figure 14: Acceleration achieved thanks to the OpenMPv4.5 on Xeon E5-2680 v3 (12 cores), GPU Tesla K40 and Tesla Pascal P100 over a sequential code. Algorithm: CGS with one reorthogonalization. Matrix type: complex-valued. Colors: blue - CPU parallel, red - K40, orange - P100.

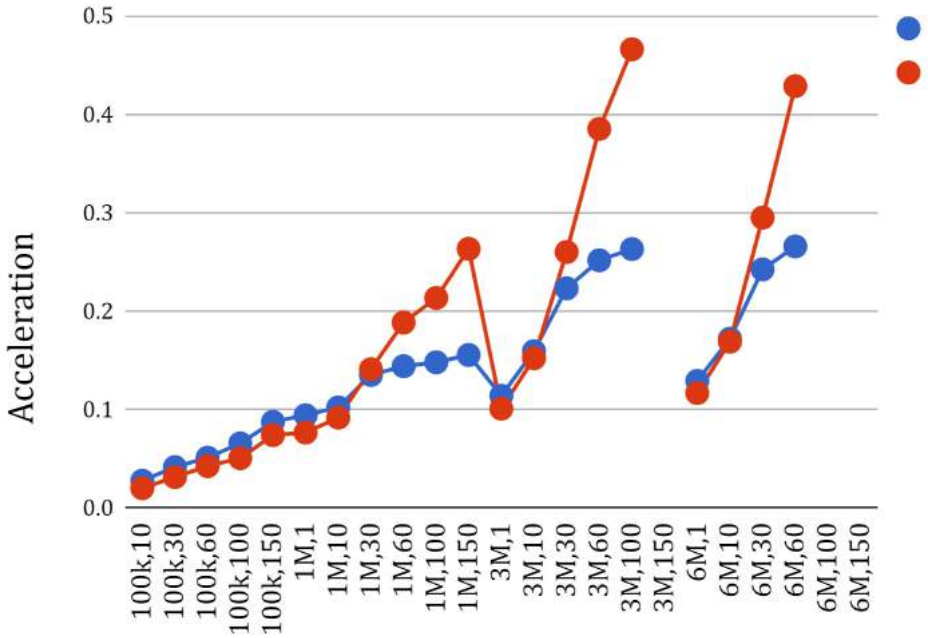

Figure 15: Acceleration of a GPU over a CPU implementations which both take advantage of a OpenMP v4.5. CPU: Xeon E5-2680 v3 (12 cores), GPU: Tesla K40, Tesla Pascal P100. Algorithm: CGS. Matrix type: complex-valued. Colors: blue - K40 over CPU parallel, red - P100 over CPU parallel.

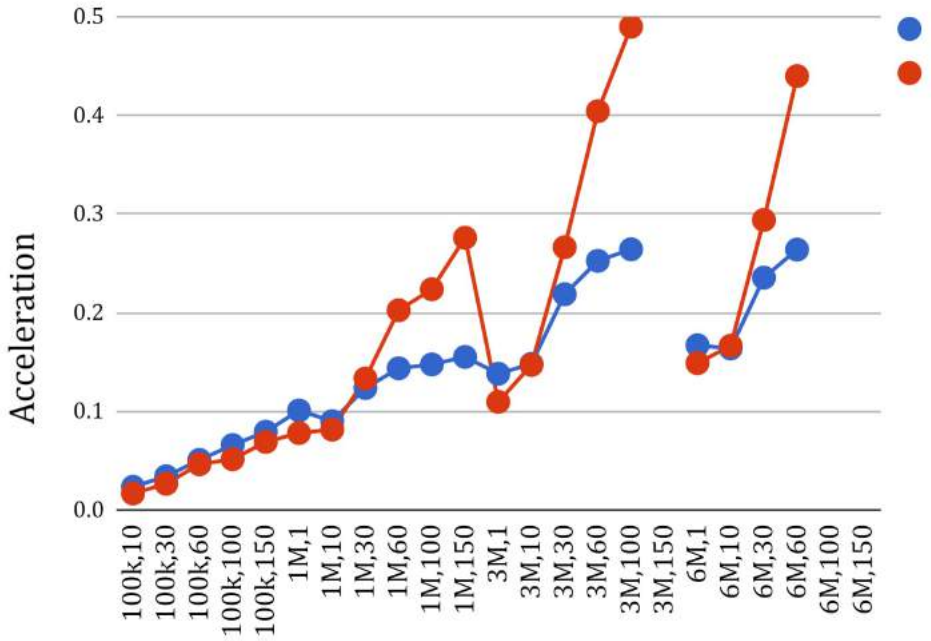

Figure 16: Acceleration of a GPU over a CPU implementations which both take advantage of a OpenMPv4.5. CPU: Xeon E5-2680 v3 (12 cores), GPU: Tesla K40, Tesla Pascal P100. Algorithm: CGS with one reorthogonalization. Matrix type: complex-valued. Colors: blue - K40 over CPU parallel, red - P100 over CPU parallel.

4 Performance comparison - OpenACC vs. OpenMP v4.5

Table 2 shows times taken by phases of GS for OpenACC and OpenMP v4.5 based implementations from Listing 4 and Listing 5, respectively. One may observe that OpenACC-based implementation outperforms OpenMP v4.5 (a speedup by a factor of 22.8). Each step of GS achieves better performance in OpenACC acceleration. The reasons why GS with OpenACC performs so good and the GS with OpenMP v4.5 is so poor are presented in Tab. 3 and 4, respectively. One may notice that each step (kernel) of GS is nicely parallelized thanks to OpenACC, since a very good execution setup is proposed. Step (kernel) 4 is the evident example: OpenACC adjusted to the changeable number of columns (noCols). On the other hand, OpenMP proposed one execution setup for all steps (kernels).

Tables 5-8 show performance measured in GFLOPs achieved by OpenACC and OpenMP v4.5 based mulithreaded (CPU) and GPU-accelerated implementations of GS and GS-RO algorithms, respectively. One may notice that CPU-based multithreaded OpenMP implementations achieve better performance over OpenACC. Multithreaded OpenMP performance is also better than GPU-based offloaded implementation (!). GPU-based OpenACC GS impelemnations outperform offloading proposed by OpenMP v4.5. Moreover, with one additional reorthogonalization (GS-RO) the performance is even better since computations exert greater influence than data transfer.

Table 2: Times taken by phases of GPU-accelerated GS algorithm for A with 1M rows and 30 columns (real-valued problem). Implementations from Listing 4 and Listing 5.

Step	Lines	OpenACC	OpenMP v4.5	Speedup
0	2-3	0.353	0.555	1.6
1	5—11	0.002	0.015	6.8
2	14—29	0.003	0.015	5.4
3	30—35	0.000	0.000	0.0
4	36—46	0.149	10.910	73.0
5	48—56	0.033	1.361	40.7
6	58—68	0.018	0.040	2.2
7	69—76	0.002	0.023	10.5
8	79 – 82	0.000	0.001	9.6
9	83—95	0.005	0.019	3.5
0-9	2–95	0.566	12.940	22.8

Table 3: Setup execution for OpenACC of the implementations from Listing 4. Matrix A with 1M rows and 30 columns (real-valued problem).

Step	Lines	Grid	Block	Registers/Thread
0	2-3	-	-	-
1	5—11	[7813,1,1]	[128,1,1]	16
2	14—29	[7813,1,1]	[128,1,1]	14
3	30—35	[1,1,1]	[128,1,1]	10
4	36—46	[*noCols*,1,1]	[128,1,1]	25
5	48—56	[31250,2,1]	[32,4,1]	28
6	58—68	[7813,1,1]	[128,1,1]	29
7	69—76	[7813,1,1]	[128,1,1]	16
8	79 – 82	[1,1,1]	[1,1,1]	12
9	83—95	[30,1,1]	[128,1,1]	16

Table 4: Setup execution for OpenMP v4.5 of the implementations from Listing 5. Matrix A with 1M rows and 30 columns (real-valued problem).

Step	Lines	Grid	Block	Registers/Thread
0	2-3	-	-	-
1	5—11	[45,1,1]	[32,8,1]	65
2	14—29	[45,1,1]	[32,8,1]	66
3	30—35	[45,1,1]	[32,8,1]	67
4	36—46	[45,1,1]	[32,8,1]	68
5	48—56	[45,1,1]	[32,8,1]	69
6	58—68	[45,1,1]	[32,8,1]	70
7	69—76	[45,1,1]	[32,8,1]	71
8	79 – 82	[45,1,1]	[32,8,1]	72
9	83—95	[45,1,1]	[32,8,1]	73

Table 5: GFLOPs achieved for GS implementations (multithreaded and GPU-based). Fourth row contains acceleration achieved by OpenACC over OpenMP v4.5. Real-valued problem.

GFLOPs	CPU-Multithreaded	GPU-K40	GPU-P100
OpenACC	5.4	13.1	18.4
OpenMP 4.5	7.2	1.2	2.6
Acceleration	0.8	11.0	7.1

Table 6: GFLOPs achieved for GS-RO (one additional reorhtogonalization) implementations (multithreaded and GPU-based). Fourth row contains acceleration achieved by OpenACC over OpenMP v4.5. Real-valued problem.

GFLOPs	CPU-Multithreaded	GPU-K40	GPU-P100
OpenACC	5.5	15.1	21.3
OpenMP 4.5	7.4	1.2	2.8
Acceleration	0.7	12.4	7.7

Table 7: GFLOPs achieved for GS implementations (multithreaded and GPU-based). Fourth row contains acceleration achieved by OpenACC over OpenMP v4.5. Complex-valued problem.

GFLOPs	CPU-Multithreaded	GPU-K40	GPU-P100
OpenACC	10.5	29.9	46.0
OpenMP 4.5	11.5	3.0	5.4
Acceleration	0.9	9.9	8.6

Table 8: GFLOPs achieved for GS-RO (one additional reorhtogonalization) implementations (multithreaded and GPU-based). Fourth row contains acceleration achieved by OpenACC over OpenMP v4.5. Complex-valued problem.

GFLOPs	CPU-Multithreaded	GPU-K40	GPU-P100
OpenACC	10.6	34.8	55.3
OpenMP 4.5	11.7	3.1	5.7
Acceleration	0.9	11.2	9.6

5 Compilations

5.1 OpenACC

All test were performed on Linux. OpenACC required installation of compiler PGICE (v.17.10). To compile code with PGICE compiler the following commands were used for a host (sequential, multithreaded) and a device (Tesla K40c, Tesla P100):

1. Host, sequential: pgc++ -o gs_host -fast -acc -ta=host gsmain.cpp
2. Host, multithreaded: pgc++ -o gs_multicore -fast -acc -ta=multicore gsmain.cpp
3. GPU, K40c: pgc++ -o gs_tesla -fast -acc -ta=tesla:cc35,lineinfo -Mlarge_arrays gsmain.cpp
4. GPU, P100: pgc++ -o gs_tesla_p100 -fast -acc -ta=tesla:cc60,lineinfo -Mlarge_arrays gsmain.cpp

One of the most important flags in compilation is a selection of a target. To perform computations on a CPU sequentially or parallel one has to use -ta=host or -ta=multicore, respectively. To execute code on a GPU a flag -ta=tesla:... with a proper parameter to select a graphics accelerator is needed. In order to allocate large arrays in global memory on a GPU one has to use a flag -Mlarge_arrays. In addition one may use -Minfo=accel flag to show information how OpenACC manages to select execution setup and as a result to accelerate computations.

5.2 OpenMP v4.5

In order to use OpenMP v4.5 one has to build GCC with support for offloading to NVIDIA GPU. Thus, procedures reported in [3] (section: *1.1 INSTALL OPENMP FOR GPU'S*) were used. After rebuild of GCC the following procedures were used to compile a code:

1. Host: /gcc/offload/wrk/install/bin/g++ -std=c++11 -O3 -fopenmp -DOPENMP -w gsmain.cpp -use_fast_math ;
2. GPU (offloading): gcc/offload/wrk/install/bin/g++ -std=c++11 -O3 -fopenmp -DOPENMP -foffload=nvptx-none -w gsmain.cpp -use_fast_math ;

To run a code on a CPU in parallel mode with n threads one has to use omp_set_num_threads(n). To perform computations on a GPU a flag -foffload=nvptx-none dedicated for offloading was used. If there is more than one accelerator one has to know an *id* of graphics accelerator and use omp_set_default_device(id).

6 Conclusions

This report is concluded with data presented in Figs. 17-18 which show the accelerations achieved by the OpenACC and OpenMP v4.5 over reference single-threaded implementation. It can be seen that:

- all GPU-based implementations achieves better performance if OpenACC directives are PGIE compiler are used.
- as long as performance on a CPU is considered, slightly better performance is achieved if OpenMP is used.

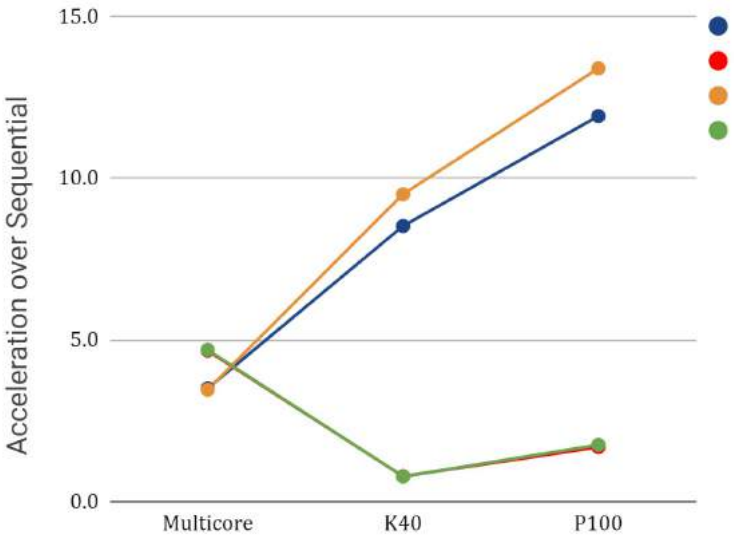

Figure 17: Acceleration of CGS and CGS-RO algorithms implemented with OpenACC and OpenMP v4.5 directives. Real-valued problem with 3M rows and 100 columns. Blue: CGS + OpenACC, red: CGS + OpenMP v4.5, orange: CGS-RO + OpenACC, green: CGS-RO + OpenMP v4.5.

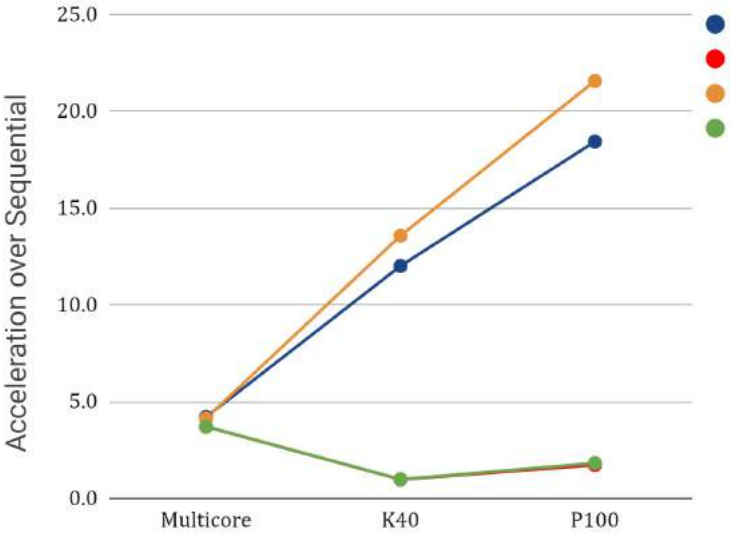

Figure 18: Acceleration of CGS and CGS-RO algorithms implemented with OpenACC and OpenMP v4.5 directives. Complex-valued problem with 3M rows and 100 columns. Blue: CGS + OpenACC, red: CGS + OpenMP v4.5, orange: CGS-RO + OpenACC, green: CGS-RO + OpenMP v4.5.

References

[1] Giraud, Luc, Julien Langou, and Miroslav Rozloznik. "The loss of orthogonality in the Gram-Schmidt orthogonalization process." Computers Mathematics with Applications 50.7 (2005): 1069-1075.

[2] The OpenACC Execution Model, R. Farber, 27 August 2012http://www.drdobbs.com/go-parallel/article/print?articleId=240006334&siteSectionName=

[3] Stefan Rosenberger, Note 17-11 First Steps with OpenMP 4.5 on Ubuntu and Nvidia GPUs (May 16, 2018) https://imsc.uni-graz.at/rosenberger/Arbeiten/Note17-11.pdf

[4] Sultana, Nawrin and Calvert, Alexander and Overbey, Jeffrey L. and Arnold, Galen, From OpenACC to OpenMP 4: Toward Automatic Translation, Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale, (2016).

[5] GTC16 - S6510 - Targeting GPUs with OpenMP 4.5 Device Directives