POLITECNICO
MILANO 1863

# Benchmark porting on GPU using OpenACC

Alessandro Artoni

# The objective

The scope of the project is to take a program and to parallelize it using openACC.

Program: https://github.com/yuhc/gpu-rodinia/tree/master/openmp/streamcluster

Pc components:

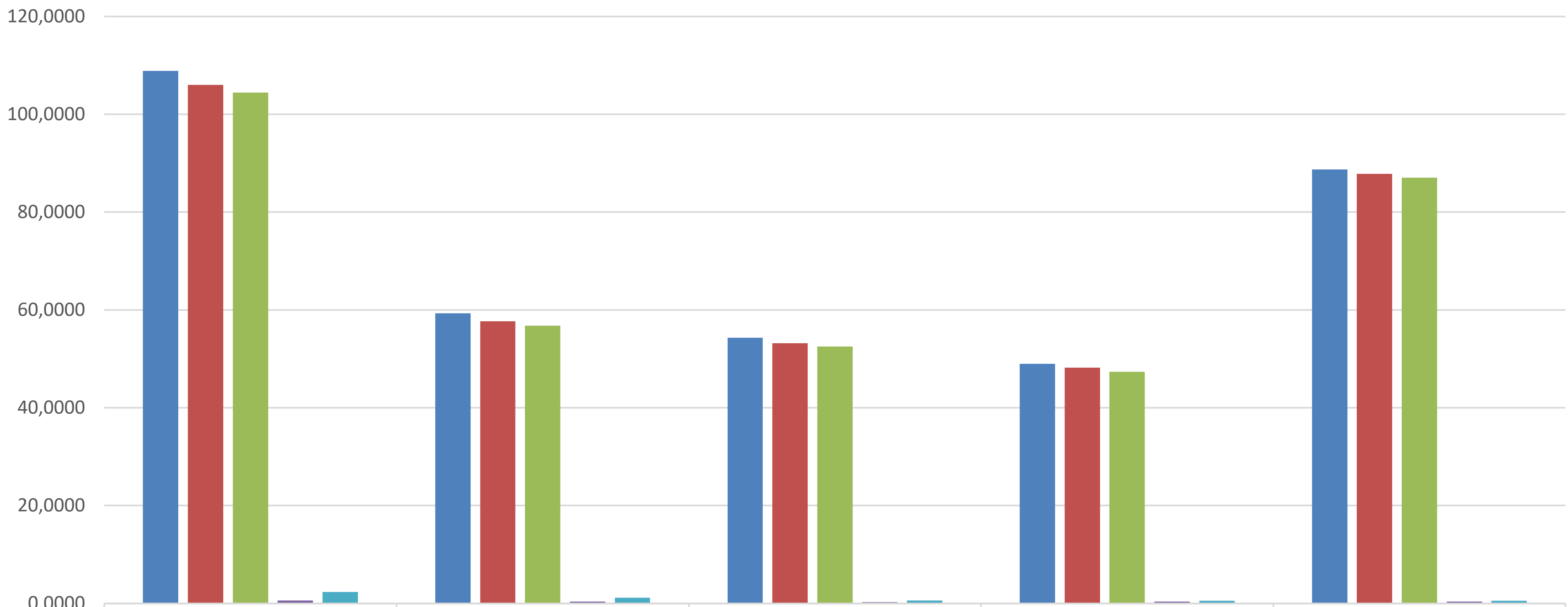
```
Detected 1 CUDA Capable device(s)

Device 0: "GeForce 920MX"
  CUDA Driver Version / Runtime Version          9.1 / 9.1
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 2048 MBytes (2147483648 bytes)
  ( 2) Multiprocessors, (128) CUDA Cores/MP:     256 CUDA Cores
  GPU Max Clock rate:                            993 MHz (0.99 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 1048576 bytes
```

| | |
|---|---|
| Produttore: | ASUSTek Computer Inc. |
| Modello: | X556UV |
| Processore: | Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz 2.59 GHz |
| Memoria installata (RAM): | 12,0 GB (11,9 GB utilizzabile) |

Note: This processor has only 2 physical cores!

# Analysis of the code




| | 1-thread | 2-thread | 4-thread | original | modificata |
|---|---|---|---|---|---|
| Total_time | 108,8460 | 59,3159 | 54,2959 | 48,9793 | 88,7404 |
| pgain | 106,0086 | 57,6799 | 53,1938 | 48,2014 | 87,8500 |
| pgain_dist | 104,4613 | 56,8079 | 52,5288 | 47,3426 | 87,052 |
| pgain_init | 0,6203 | 0,3622 | 0,2546 | 0,3776 | 0,3681 |
| pspeedy | 2,3482 | 1,1758 | 0,6249 | 0,5598 | 0,5608 |

Time is expressed in seconds

# Analysis of the code

Analysing the time the program took to execute, it has been noticed that the most expansive time part of the program was the pgain cycle. In particular this part of the loop:

```
for ( i = k1; i < k2; i++ ) {
  float x_cost = dist(points->p[i], points->p[x], points->dim)
    * points->p[i].weight;
  float current_cost = points->p[i].cost;

  if ( x_cost < current_cost ) {

    // point i would save cost just by switching to x
    // (note that i cannot be a median,
    // or else dist(p[i], p[x]) would be 0)

    switch_membership[i] = 1;
    cost_of_opening_x += x_cost - current_cost;

  } else {

    // cost of assigning i to x is at least current assignment cost of i

    // consider the savings that i's **current** median would realize
    // if we reassigned that median and all its members to x;
    // note we've already accounted for the fact that the median
    // would save z by closing; now we have to subtract from the savings
    // the extra cost of reassigning that median and its members
    int assign = points->p[i].assign;
    lower[center_table[assign]] += current_cost - x_cost;
  }
}
```

# Bottom-up optimization

At this point, it has been used a bottom up approach to let the GPU handle that cycle, starting from the pgain_dist loop using structured data region and kernels.

In order to manage better the memory from the host (cpu) to the device, some temporary arrays were introduced.

Because of this approach, the data were copyed in and out of the device each time they were used inside the parallel region.

This resulted in a lot of time spended in data trasfer from the CPU and the GPU.

Thus, it was decided to use unstructured data region, and «present_or_copy» clause. Since present table sometimes was giving problems, only «update» pragmas were used.

Then, data entering / exit were firstly made before pgain call (inside pFL), then inside localSearch.

Finally, compiling with –ta=tesla:managed gave reasonable results.

# Bottom-up optimization
# *Structured data region*

```
  memset(temp_diff, 0, k2*sizeof(double));
  #pragma acc data copy(temp_diff[k1:k2]) copyin(dim_points) copyin(points[0:1]) copyin(points->weight_p[0:k2])\
   copyin(points->assign_p[0:k2]) copyin(points->cost_p[0:k2]) copyin(points->coord_p[k1:k2][0:dim_points]) \
   copy(switch_membership[k1:k2]) copy(cost_of_opening_x) copy(result)//copyin(current_point_coords[0:dim_points]) copy(center_table[0:centerDim])
  {
    #pragma acc kernels// reduction(+: result)
    {
      for ( i = k1; i < k2; i++ ) {
        result=0.0;
          //#pragma acc loop reduction(+: result)
          //{
            for (int j=0;j<dim_points;j++)
            {
              result += (points->coord_p[i][j] - points->coord_p[x][j])*(points->coord_p[i][j] - points->coord_p[x][j]);
              /*#ifdef INSERT_WASTE
                double s = waste(result);
                result += s;
                result -= s;
              #endif*/
            }
        float x_cost = result* points->weight_p[i];
        float current_cost = points->cost_p[i];
        float temp = x_cost - current_cost;
        if ( temp < 0) {
          // point i would save cost just by switching to x(note that i cannot be a median,or else dist(p[i], p[x]) would be 0)
          switch_membership[i] = 1;
          cost_of_opening_x += temp;
        } else {
          temp_diff[i] = temp*(-1);
        }
      }
    }
```

# Bottom-up optimization
# *Unstructured data region*

```c
#pragma acc enter data copyin(temp_diff[k1:k2]) copyin(points[0:1]) \
 copyin(points->assign_p[0:k2]) copyin(points->cost_p[0:k2]) \
copyin(switch_membership[k1:k2]) \
pcopyin(points->weight_p[0:k2]) pcopyin(points->coord_p[k1:k2][0:dim_points]) \
copyin(result) copyin(dim_points) // copyin(cost_of_opening_x)

#pragma acc declare present(points[0:1], switch_membership[0:k2], temp_diff[k1:k2])

#pragma acc kernels// reduction(+: result)
  {
    for ( i = k1; i < k2; i++ ) {
      result=0.0;
        //#pragma acc loop reduction(+: result)
        //{
          for (int j=0;j<dim_points;j++)
          {
            result += (points->coord_p[i][j] - points->coord_p[x][j])*(points->coord_p[i][j] - points->coord_p[x][j]);
            /*#ifdef INSERT_WASTE
              double s = waste(result);
              result += s;
              result -= s;
            #endif*/
          }
      //  }
      //}
      float x_cost = result* points->weight_p[i];
      float current_cost = points->cost_p[i];
      float temp = x_cost - current_cost;
      if ( temp < 0) {
        // point i would save cost just by switching to x(note that i cannot be a median,or else dist(p[i], p[x]) would be 0)
        switch_membership[i] = 1;
        cost_of_opening_x += temp;
      } else {
        temp_diff[i] = temp*(-1);
      }
  }
}
#ifdef DEBUGGER
  fprintf(stdout, "PostPragma\n" );
#endif

#pragma acc exit data copyout(temp_diff[k1:k2]) copyout(switch_membership[0:k2]) \
copyout(result)  copyout(points->assign_p[0:k2]) //copyout(cost_of_opening_x)

#pragma acc exit data delete(temp_diff[k1:k2]) delete(dim_points) delete(points[0:1]) delete(points->weight_p[0:k2])\
 delete(switch_membership[k1:k2])  delete(result) \
```

# Bottom-up optimization
## *Going-up: data entering before pgain*

```
#pragma acc enter data copyin(points[0:1])
#pragma acc enter data copyin(points->weight_p[0:k2])
#pragma acc enter data copyin(points->coord_p[0:k2][0:dim_points])
#pragma acc enter data create(points->assign_p[0:k2], points->cost_p[0:k2], switch_membership[0:k2])
#ifdef DEBUGGER
fprintf(stdout, "Post acc enter data main ext loop\n");
#endif
for (i=0;i<iter;i++) {
  x = i%numfeasible;
  //printf("iteration %d started********\n", i);
  change += pgain(feasible[x], points, z, k, pid, barrier);
  c++;

  //printf("iteration %d/%d finished @@@@@@\n", i,iter);
}
//sembra non sia neanche necessario copiare indietro
#pragma acc exit data copyout(points->weight_p[0:k2])
#pragma acc exit data copyout(points->coord_p[0:k2][0:dim_points])
#pragma acc exit data copyout(points->assign_p[0:k2], points->cost_p[0:k2], switch_membership[0:k2], points[0:1])
#pragma acc exit data delete(points->weight_p[0:k2])
```

# Bottom-up optimization

## *Going-up: data entering before pFL*

```
//creating space on the device
#pragma acc enter data copyin(points[0:1],points->weight_p[0:points_num],points->coord_p[0:points_num][0:dim_points])
#pragma acc enter data copyin(temp_diff[0:points_num])
#pragma acc enter data create(points->assign_p[0:points_num], points->cost_p[0:points_num], switch_membership[0:points_num])

  while(1) {
    d++;
#ifdef PRINTINFO
    if( pid==0 )
      {
  printf("loz = %lf, hiz = %lf\n", loz, hiz);
  printf("Running Local Search...\n");
      }
#endif
    /* first get a rough estimate on the FL solution */
    //    pthread_barrier_wait(barrier);

    //lastcost = cost;

    cost = pFL(points, feasible, numfeasible,
         z, &k, cost, (long)(ITER*kmax*log((double)kmax)), 0.1, pid, barrier);

    /* if number of centers seems good, try a more accurate FL */
    if (((k <= (1.1)*kmax)&&(k >= (0.9)*kmin))||
  ((k <= kmax+2)&&(k >= kmin-2))) {

#ifdef PRINTINFO
      if( pid== 0)
  {
    printf("Trying a more accurate local search...\n");
  }
#endif
```

# Using GPU parallelism

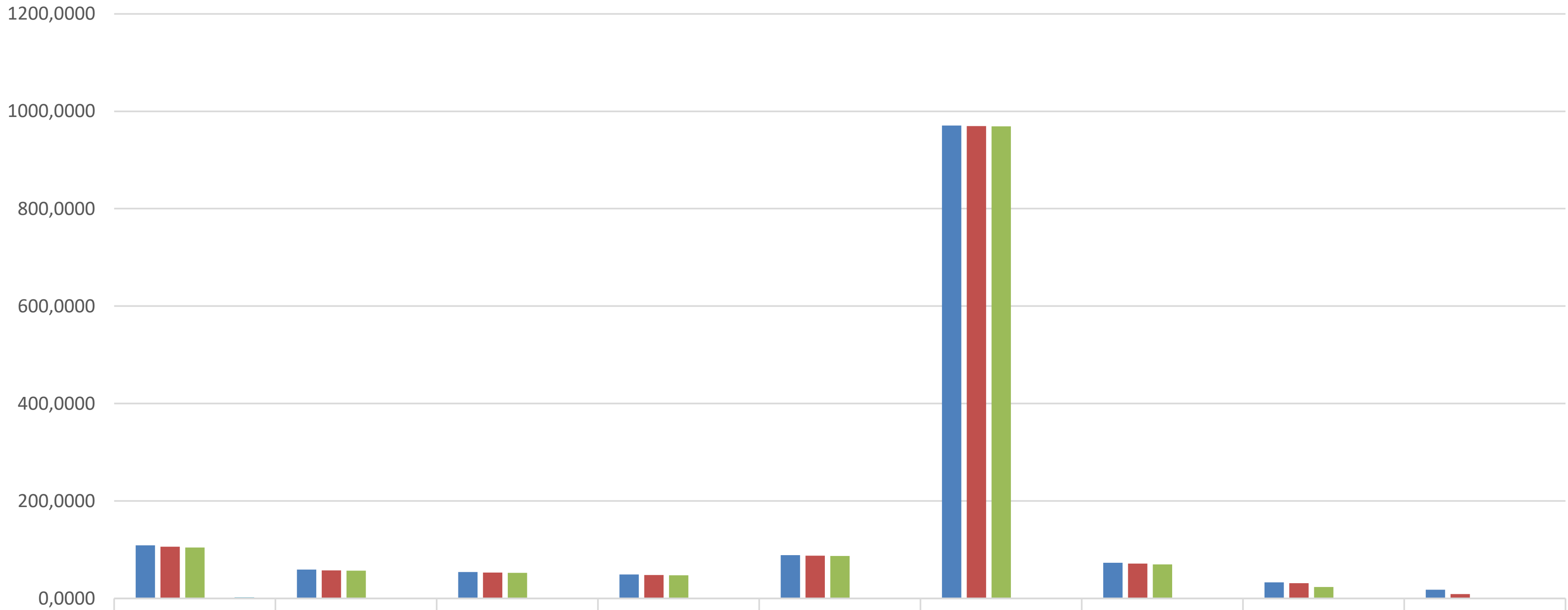
At this point, it was tried to exploit the GPU thread/warp/blocks.

The division in the bottom picture, resulted the best one after several tries.

```
#pragma acc declare present(points[0:1], switch_membership[0:k2], temp_diff[k1:k2]) \
present(points->weight_p[0:k2], points->coord_p[k1:k2][0:dim_points])

#pragma acc parallel loop reduction(+: result) device_type(nvidia) gang vector_length(64)//
  {
    for ( i = k1; i < k2; i++ ) {
      result=0.0;
        #pragma acc loop gang vector //vector_length(64)//gang(16) vector_length(32) // vector
          for (int j=0;j<dim_points;j++)
          {
            result += (points->coord_p[i][j] - points->coord_p[x][j])*(points->coord_p[i][j] - points->coord_p[x][j]);
```

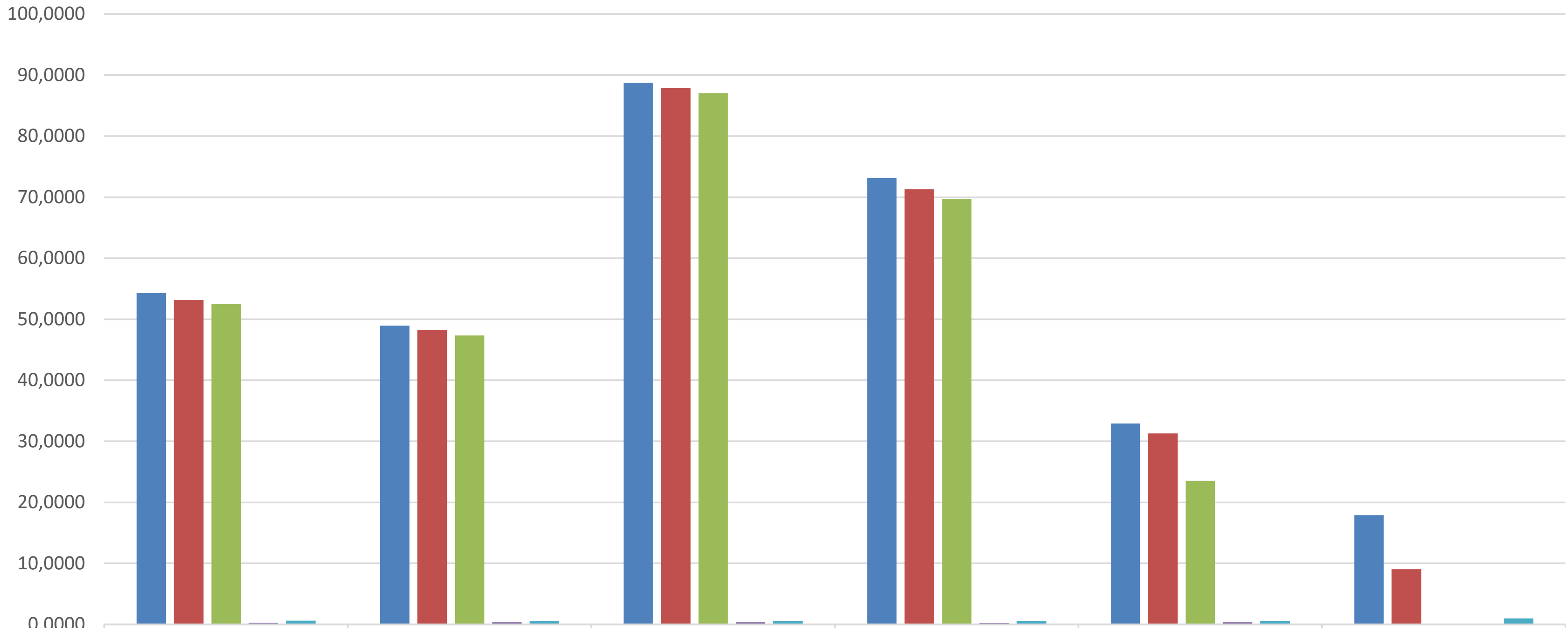
# Final results




| | 1-thread | 2-thread | 4-thread | original | modificata | open_acc_old | open_acc_no_mem | open_acc | cuda |
|---|---|---|---|---|---|---|---|---|---|
| Total_time | 108,8460 | 59,3159 | 54,2959 | 48,9793 | 88,7404 | 970,5862 | 73,1292 | 32,9145 | 17,8655 |
| pgain | 106,0086 | 57,6799 | 53,1938 | 48,2014 | 87,8500 | 969,3063 | 71,2902 | 31,2929 | 9,0095 |
| pgain_dist | 104,4613 | 56,8079 | 52,5288 | 47,3426 | 87,052 | 968,7582 | 69,7128 | 23,5421 | 0 |
| pgain_init | 0,6203 | 0,3622 | 0,2546 | 0,3776 | 0,3681 | 0,1938 | 0,2398 | 0,3433 | 0,0000 |
| pspeedy | 2,3482 | 1,1758 | 0,6249 | 0,5598 | 0,5608 | 0,5585 | 0,5621 | 0,5823 | 0,9832 |

Time is expressed in seconds.

# Final results - filtered




| | 4-thread | original | modificata | open_acc_no_mem | open_acc | cuda |
|---|---|---|---|---|---|---|
| Total_time | 54,2959 | 48,9793 | 88,7404 | 73,1292 | 32,9145 | 17,8655 |
| pgain | 53,1938 | 48,2014 | 87,8500 | 71,2902 | 31,2929 | 9,0095 |
| pgain_dist | 52,5288 | 47,3426 | 87,052 | 69,7128 | 23,5421 | 0 |
| pgain_init | 0,2546 | 0,3776 | 0,3681 | 0,2398 | 0,3433 | 0,0000 |
| pspeedy | 0,6249 | 0,5598 | 0,5608 | 0,5621 | 0,5823 | 0,9832 |

Time is expressed in seconds.

# Conclusions

Since the pc where original code using threads were tested didn't have more than two physical cores, the algorithm using thread didn't perform really well. Using other CPUs, performance significantly improves arriving to near 10 seconds – which is even better that CUDA!

The limits of OpenACC is the lack of possibility to allocate exactly the amount of block/threads needed without having to do a significant refactor of the code. This is also a reason why CUDA's code works better: the code was completely rebuilt around it.

Another limit of the solution proposed, is the limit of memory usable on the GPU.

However, OpenACC focus on portability, and the version should be able to perform well on diffrent devices.