## Instructions for submitting

Each group has to submit a compressed file named **lab3_TxGyy.zip**, where *TxGyy* is your group identifier. A .tar or a .tgz files are also accepted (e.g. lab3_T1G1.zip or lab3_T2G21.zip). The compressed file has to contain three 5 files:

```
lab3_TxGyy.zip ─┬─ axpy.c
                ├─ dot.c
                ├─ spmv.c
                ├─ cg.c
                ├─ spmv.cu
                └─ bonus.cu
```

The template file named *lab3_TxGyy.zip* containing the reference codes has been published in the Aula Global.

This lab assignment has a weight of **20%** of the total grade of the subject and the deadline for this assignment is **June 18th at 11:59pm**.

For any arising question please, post it in the *Lab class forum* in the Aula Global, however, **do not post your code in the Forum**. For other questions regarding the assignment that you consider cannot be posted in the Forum (e.g. personal matters or code), please contact the responsible of the lab **guillermoandres.oyarzun@upf.edu.**

# Criteria

The codes will be tested and evaluated on the same cluster where you will be working. The maximum grade on each part will only be given to the exercises that solve in the most specific way and that tackle all the functionalities and work requested. All the following criteria will be applied while reviewing your labs in the cluster.

Exercises that will not be evaluated:

- A code that does not compile.
- Code giving wrong results.

Exercises with penalty:

- A code with warnings in the compilation.
- *lab3_TxGyy.zip* delivered files not structured as described previously.

## Previous considerations

We will be using the new cluster with GPUs, therefore the IP has changed to **10.49.0.82** and your password is listed on the passwords_gpu file on the Aula Virtual.

Remember that the compilation cannot be done in the login node and that it has to be submitted to the compute nodes. To ease the process, all the job files and Makefiles are in the file lab3_TxGyy.zip prepared to compile the code and run it, and you do not have to modify them. Just focus on the code and the work requested for each exercise. In the case the compilation fails, the job will finish and it will not try to run the binary.

## Notice

- Remember that it is totally forbidden to copy in these exercises. It is understood that there may be communication between the students during the realization of the activity, but the delivery of these exercises has to be differentiated from the rest.

- Therefore, deliveries that contain an identical part regarding deliveries of other groups will be considered copies and all the people involved (without the link between them being relevant) will suspend the activity delivered.

# Practice 3

In this practice, we will implement a small linear algebra library to solve linear problems in computational fluid dynamics simulations.

## 1. AXPY (15%)

The *axpy* routine is a very extended routine that performs a vector-vector operation as described in the following formula.

$$y_i = \alpha x_i + y_i$$

Where:

- α is a scalar
- *x* and *y* are vectors each with *n* elements.

Write a C and OpenACC that Implements the *axpy* operation where both vectors are double precision. The declaration of the functions for the sequential CPU and OpenACC versions are the following:

```
void axpy_cpu(int offset, int n,  double alpha double* x, double* y);
void axpy_gpu(int offset, int n,  double alpha double* x, double* y);
```

**Offset** is an integer number that permits to start the operation at any component of the array, and **n** is the number of components that will be considered. So, if offset =0 and n=vector_size, the operation is applied to the full vector.

Axpy_cpu: must perform the operation only on the CPU

Axpy_gpu: must perform the operation on the GPU using OpenACC

Note that the GPU version should run faster than the CPU one. It is requested to introduce all the execution clauses that expose parallelism explicitly. The compiler is smart and sometimes

includes them automatically. The code may run well, but **to get the maximum score you need to add them.**

Some minor differences (less than $10^{-14}$) between CPU and GPU results are acceptable due to truncations errors.

## 2. DOT (15%)

**Dot product (or scalar product)** As we already know from the subject Linear Algebra, the dot product is an algebraic operation that calculates a scalar from two vectors. Its usage is very extended and common in numerical methods. The formula is described as follows.

$$dot = \sum_{i=0}^{n-1} x_i y_i$$

Where:

- *dot* is a scalar.
- x and *y* are vectors each with n elements.

Write a C code that Implements the scalar product of *x* and *y* where both vectors components are double precision. The definition of these function for the sequential and OpenMP versions are the following:

```
double dot_product_cpu (int offset, int n, double *x, double *y);
double dot_product_gpu (int offset, int n, double *x, double *y);
```

**Offset** is an integer number that permits to start the operation at any component of the array, and **n** is the number of components in which it will be applied. So, if offset =0 and n=vector_size, the operation is applied to the full vector.

dot_product_cpu: must perform the operation only on the CPU

dot_product_gpu: must perform the operation on the GPU using OpenACC

Note that the GPU version should run faster than the CPU one. It is requested to introduce all the execution clauses that expose parallelism explicitly. The compiler is smart and sometimes includes them automatically. The code may run well, but **to get the maximum score you need to add them.**

Some minor differences (less than $10^{-14}$) between CPU and GPU results are acceptable due to truncations errors.

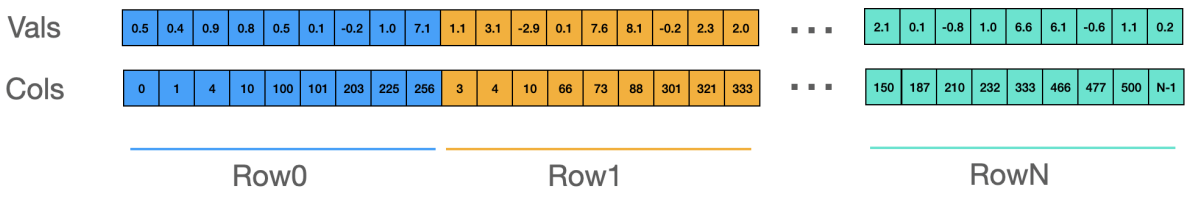
## 3. SpMV (15%)

**The Sparse Matrix vector multiplication (SpMV)** is a common kernel found in scientific codes. It consists of the multiplication of a sparse matrix with a vector. A sparse matrix is a matrix in which most of its components are zeros. So, instead of storing all the coefficients of the matrix, we can just save the non-zero elements. In this practice, we will be dealing with matrices of more than 16 million rows but with only 9 non-zero coefficients per row.

In this case, we are using a compressed storage based on two unidimensional arrays, also called ELLPACK format.

**Vals:** array of doubles that contains the non-zero coefficients of the matrix.

**Cols:** array of integers that contains the column index of the non-zero coefficient.




Note that by using the compressed format we are saving memory. If we had stored the full matrix A of NxN, we would require $N^2$ double elements, where N is the number of rows of the matrix. Moreover,

most of those entries had been zeros. Using the compressed format we only need 9N double elements + 9N integer elements.

The component i of the resulting vector (y) of the multiplication of the sparse matrix A by the vector x is:

$$y[i] = \sum_{j=0}^{8} A[i][j] * x[j]$$

where $0 \leq i < N$ and $0 \leq j < 9$. Note that in our case, the A[i][j] is read from the Vals array, and the j index is obtained using the cols array.

You will need to create a cpu and an openacc implementation of the sparse matrix vector multiplication

Write a C code that Implements the scalar product of *x* and *y* where both vectors have double components. The definition of these function for the sequential and OpenMP versions are the following:

```
double spmv_cpu (int offset, int n,double* vals, int* cols, double *x, double *y);
double spmv_gpu (int offset, int n,double* vals, int* cols, double *x, double *y);
```

**Offset** is an integer number that permits to start the operation at any row of the matrix, and **n** is the number of rows that are operated from that point. So, if offset =0 and n=vector_size, the operation is applied to the full matrix.

spmv_cpu: must perform the operation only on the CPU

spmv_gpu: must perform the operation on the GPU using OpenACC

Note that the GPU version should run faster than the CPU one. It is requested to introduce all the execution clauses that expose parallelism explicitly. The compiler is smart and sometimes includes them automatically. The code may run well, but **to get the maximum score you need to add them.**

Some minor differences (less than $10^{-14}$) between CPU and GPU results are acceptable due to truncations errors.

## 4. CG (25%)

The Conjugate gradient is an iterative method for solving linear systems of equations. It is widely used in scientific computing and its main algorithm is composed of the three kernels previously implemented.

The algorithm is described below:

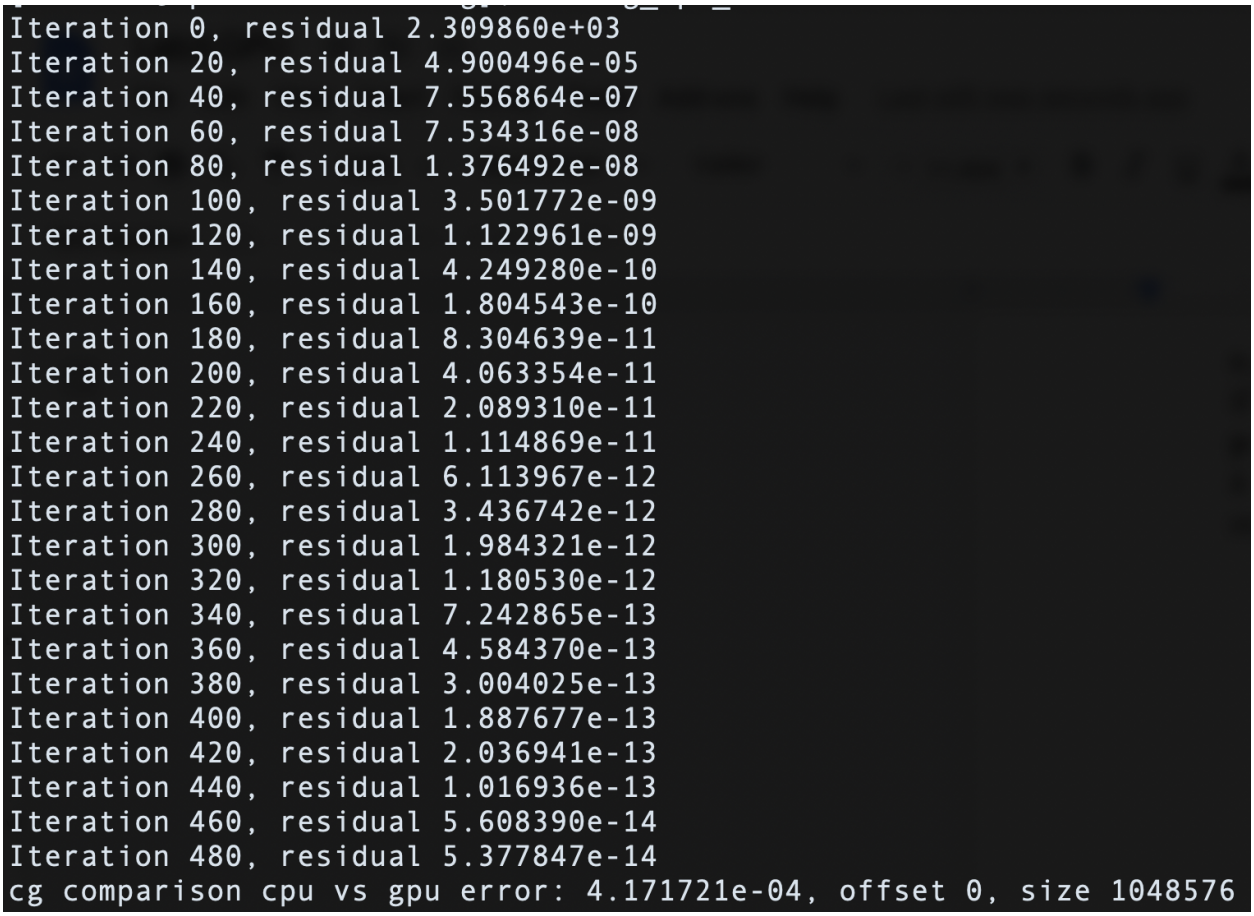
$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
if $\mathbf{r}_0$ is sufficiently small, then return $\mathbf{x}_0$ as the result
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
repeat
  $\alpha_k := \dfrac{\mathbf{r}_k^\mathsf{T} \mathbf{r}_k}{\mathbf{p}_k^\mathsf{T} \mathbf{A} \mathbf{p}_k}$
  $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$
  $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{p}_k$
  if $\mathbf{r}_{k+1}$ is sufficiently small, then exit loop
  $\beta_k := \dfrac{\mathbf{r}_{k+1}^\mathsf{T} \mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T} \mathbf{r}_k}$
  $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
  $k := k + 1$
end repeat
return $\mathbf{x}_{k+1}$ as the result

The code for the algorithm is given, you will need to copy your implementation of the 3 previous kernels. The remaining kernels are given too, however, they are not ported to GPU. Without introducing any changes, use the script job_profile_cpu.sh to measure the relative weight of the three operations. Use the script job.sh to measure the time of a pure CPU version of the code.

Check out the code cg.c and provide an optimal OpenACC implementation of the code. Make sure that you minimize the data transfers, and parallelize the remaining kernels using OpenACC. Run the

job_profile_gpu.sh to compare with the previous results. Compare CPU and GPU times.

If the CG runs correctly you should see an output like this:




The output shows how the linear solver is converging to the final solution. Note that for sake of time, we have run only 500 iterations of the solver, that's why the solver is not fully converged. If your implementation is correct, you should obtain similar results. (not necessary exactly the same due to the truncation errors).

## 5. SpMV CUDA (30%)

Copy your cpu implementation of the spmv in exercise 3, and develop a cuda version.

In this exercise you have to implement:

- a CUDA implementation of the same operation of exercise 3.
- Allocates the data in the GPU
- Transfer data to the GPU
- Copy the result back to the CPU.
- Try to obtain the same performance or better than OpenACC.

Considerations:
Do not assume that the number of rows is divisible by the number of threads. Create a generic way of distributing the rows over the threads and blocks.

## 6. Bonus track (25%) (optional)

If you want to get an extra 25% in the practice, transform the data structure in which the matrix is stored to provide more optimal memory reads. Implement the code using CUDA. Deliver an extra file named bonus.cu, the code should compile and run faster than the previous version of spmv in CUDA.

**About the grading:** note that the final grade of the practice will be at maximum 10.

TIP: Use the number of threads to reorganize the data. The data must be ordered to ensure coalesced memory accesses.