Universitat Pompeu Fabra *Barcelona*

Lab 3

Pablo Arias[*], Ricard Borrell Pol[†], Sergi Laut Turón[‡], Pavel Pratyush[§], Daniel Santos-Oliván[#]

Instructions

In this Lab session we will implement a small linear algebra library using GPU parallelization. This Lab session consists of 6 problems with several questions each. You must compile your answers to the exercises in a report, explaining how you solved the problem and answering the questions. This lab assignment is worth 20% of the total grade of the subject, and the deadline for it is June 7th at 23:59h.

Each group must submit a compressed file named `lab3_TxGyy.zip`, where `TxGyy` is your group identifier. A `.tar` or a `.tgz` file is also accepted (e.g., `lab3_T1G1.zip` or `lab3_T2G21.zip`). The compressed file **must** contains five folders (`1_axpy`, `2_dot`, `3_spmv`, `4_cg` and `5_spvm_cuda`) and **all** the requested files with the following structure. Additional files will not be considered and will not be penalized.

```
lab3_TxGyy.zip ─┬─ 1axpy ─── axpy.c
                ├─ 2dot ─── dot.c
                ├─ 3spmv ─── spmv.c
                ├─ 4cg ─┬─ cg_cpu.c
                │       └─ cg_gpu.c
                └─ 5spmv_cuda ─┬─ spmv.cu
                               └─ spmv_optim.cu
```

A sample file named `lab3_TxGyy.zip` containing the reference codes has been published in Aula Global. You need to create your job scripts to perform your tests according to what is asked. The Makefiles should not be modified unless the exercise says so. Focus on the code and the work requested for each exercise.

If you have any questions, please post them on the lab class forum in Aula Global. However, do not post your code in the Forum. If you have other questions regarding the assignment that you consider cannot be posted in the Forum (e.g., personal matters or code), please contact the lab responsible person.

Criteria

The codes will be tested and evaluated on the same cluster where you work. The maximum grade on each part will only be given to these exercises that solve in the most specific way and 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.
- A code giving wrong results.
- A code that does not adhere to all the input/output requests.
- `lab3_TxGyy.zip` delivered files not structured or named as described previously.

Exercises with penalty: A code with warnings in the compilation.

[*] `pablo.arias@upf.edu`; [†] `ricard.borrell@upf.edu`.
[‡] `sergi.laut@upf.edu`; [§] `pratyush@upf.edu`; [#] `daniel.santos@upf.edu`.

1. AXPY

The `axpy` routine is a very extended routine that performs the following vector-vector operation:

$$y_i = \alpha x_i + y_i, \qquad i = 1, ..., n, \tag{1}$$

where α is a scalar and $\mathbf{x} = (x_1, ..., x_n), \mathbf{y} = (y_1, ..., y_n)$ are vectors with n elements.

Write a sequencial C code and a parallel OpenACC version that implement the `axpy` operation, where both vectors are of double precision.

The declaration of the functions for the sequential CPU and OpenACC versions are the following:

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

where `n` is the vectors length.

`axpy_cpu` must perform the operation on sequentially on the CPU, while `axpy_gpu` must perform it in the GPU using OpenACC. Both versions should give the same results (differences smaller than 10^{-10}). The GPU code must run faster than the CPU code for large values of n. To get the maximum score, you should introduce all execution clauses that expose parallelism explicitly.

Report Questions 1 — axpy (20%)

1. Explain how you have parallelized the code, detailing the directives used and their clauses.
2. Plot the speed-up as a function of the vector length n, for $n = 10^k$, $k = 1, 2, ..., 7$. What do you observe from the speed-up plot?

2. DOT

Next we are going to implement a dot product between two vectors $\mathbf{x}, \mathbf{y}$ with n variables, given by the following expression:

$$s = \langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} x_i y_i, \tag{2}$$

Write a sequencial C code and a parallel OpenACC version that implement the `dot` operation, where both vectors are of double precision. The declaration of the functions for the sequential CPU and OpenACC versions are the following:

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

where `n` is length of the vectors.

`dot_product_cpu` must perform the operation on sequentially on the CPU, while `dot_product_gpu` must perform it in the GPU using OpenACC. Both versions should give the same results (differences smaller than 10^{-10}). The GPU code must run faster than the CPU code for large values of n. To get the maximum score, you should introduce all execution clauses that expose parallelism explicitly.

Report Questions 2 — dot (20%)

1. Explain how you have parallelized the code, detailing the directives used and their clauses.
2. Plot the speed-up as a function of the vector length n, for $n = 10^k$, $k = 1, 2, ..., 7$. What do you observe from the speed-up plot?

3. SpMV

The **sparse matrix vector** multiplication (SpMV) is a common kernel found in scientific codes. A sparse matrix is a matrix in which most of the components are zeros. For this type of matrices, instead of storing all the coefficients in memory, we can just store the non-zero elements. In this exercise we will assume that the each row has at most r non-zero elements.

There are different ways to efficiently store sparse matrices in memory. We are going to use a compressed format that we will call the IPPD2024 format[1], as it is useful for cases in which the maximum number of non-zero elements per-row is constant (as it is our case).

This format stores the matrix as two unidimensional arrays, `cols`, `vals`, of length mr, where m is the number of rows, and r is the number of non-zero coefficients per-row. For each row of the matrix `vals` stores the r non-zero values and `cols` stores their column indices.

Thus, for the 7×9 sparse matrix $\mathbf{A}$ with at most $r = 3$ non-zero elements per row:

$$\mathbf{A} = \begin{pmatrix} 1.4 & 0 & 0 & 0 & 2.2 & 7.2 & 0 & 0 & 0 \\ 0 & 0 & 3.2 & 5.4 & 0 & 0 & 1.0 & 0 & 0 \\ 0 & 7.8 & 0 & 0 & 4.4 & 0 & 0 & 0 & 0 \\ 9.1 & 0 & 0 & 0 & 0 & 0 & 8.3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6.3 \\ 0 & 0 & 9.2 & 0 & 5.5 & 0 & 0 & 0 & 0.2 \\ 0 & 1.3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

[1]Which is a modification of the ELLPACK format.

we have:

$$\begin{array}{rl|l|l|l|l|l|l}
 & \text{row 0} & \text{row 1} & \text{row 2} & \text{row 3} & \text{row 4} & \text{row 5} & \text{row 6} \\
\texttt{vals} = [& 1.4\ \ 2.2\ \ 7.2 & 3.2\ \ 5.4\ \ 1.0 & 7.8\ \ 4.4\ \ 0.0 & 9.1\ \ 8.3\ \ 0.0 & 6.3\ \ 0.0\ \ 0.0 & 9.2\ \ 5.5\ \ 0.2 & 1.3\ \ 0.0\ \ 0.0\ ] \\
\texttt{cols} = [& 0\ \ \ 4\ \ \ 5 & 2\ \ \ 3\ \ \ 6 & 1\ \ \ 4\ \ \ 4 & 0\ \ \ 6\ \ \ 6 & 8\ \ \ 8\ \ \ 8 & 2\ \ \ 4\ \ \ 8 & 1\ \ \ 1\ \ \ 1\ ]
\end{array}$$

Note that the number of non-zero elements can be smaller than r. The compressed IPPD2024 format takes mr doubles and mr integers, whereas storing the entire matrix takes mn doubles. Our compressed format also allows to reduce the number of operations in a matrix-vector product.

Write a sequencial C code and a parallel OpenACC version for the sparse matrix-vector product using the IPPD2024 format. The matrix coefficients and the vectors are stored in double precision.

The declaration of the functions for the sequential CPU and OpenACC versions are the following:

```c
void spmv_cpu(int m, int r, double *vals, int *cols, double *x, double *y);
void spmv_gpu(int m, int r, double *vals, int *cols, double *x, double *y);
```

where `m` is the number of rows of $\mathbf{A}$, `n` the number of columns and r is the number of non-zero elements per-row. `x` is the input array with `n` components and `y` is the output array with `m` components, and should contain the product $\mathbf{y} = \mathbf{A}\mathbf{x}$.

`spmv_cpu` must perform the operation on sequentially on the CPU, while `spmv_gpu` must perform it in the GPU using OpenACC. Both versions should give the same results (differences smaller than 10^{-10}). The GPU version should run faster than the CPU one. To get the maximum score, you should introduce all execution clauses that expose parallelism explicitly.

Report Questions 3 — spmv (20%)

1. How many floating point operations are needed for a **dense** matrix-vector product with an $m \times n$ matrix? How many floating point operations are needed if the matrix is sparse with r non-zero elements per-row and stored in the IPPD2024 format.

2. Explain how you parallelized the code, detailing the execution clauses used. Detail how each loop was parallelized by OpenACC based on the information provided by the compiler.

4. CG

The **conjugate gradient** is an iterative method for solving a linear system of equations. It is widely used in scientific computing and its main algorithm is composed of the three kernels implemented previuosly. The algorithm is described below:

Algorithm 1: Conjugate gradient linear solver

```
input  : $m \times n$ matrix $\mathbf{A}$, right hand term $\mathbf{b} \in \mathbb{R}^m$, initial value $\mathbf{x}$,
         tolerance $\tau$, maximum number of iterations $k_{\max}$
output: estimated solution $\mathbf{x}$
$\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$
$\mathbf{p} = \mathbf{r}$
$k = 0$
for $\langle \mathbf{r}, \mathbf{r} \rangle > \tau$ and $k < k_{\max}$ do
    $\alpha = \frac{\langle \mathbf{r}, \mathbf{r} \rangle}{\langle \mathbf{p}, \mathbf{A}\mathbf{r} \rangle}$
    $\mathbf{x} = \mathbf{x} + \alpha \mathbf{p}$
    $\mathbf{r}_0 = \mathbf{r}$
    $\mathbf{r} = \mathbf{r} - \alpha \mathbf{A}\mathbf{p}$
    $\beta = \frac{\langle \mathbf{r}, \mathbf{r} \rangle}{\langle \mathbf{r}_0, \mathbf{r}_0 \rangle}$
    $\mathbf{p} = \mathbf{r} + \beta \mathbf{p}$
$\hat{\mathbf{x}} = \mathbf{x}$
```

You will find two incomplete files `cg_cpu.c` and `cg_gpu.c`, both containing a CPU version of the CG algorithm. The goal is to complete the code in these files to obtain CPU and GPU versions of the algorithm.

If the code runs correctly you should see an output similar to this one:

```
teration 0, residual 2.309860e+03
Iteration 20, residual 4.900496e-05
Iteration 40, residual 7.556864e-07
Iteration 60, residual 7.534316e-08
[...]
Iteration 420, residual 2.036941e-13
Iteration 440, residual 1.016936e-13
Iteration 460, residual 5.608390e-14
Iteration 480, residual 5.377847e-14
cg comparison x_sol vs x_gpu error: 4.171721e-04, offset 0, size 1048576
```

The output shows how the linear solver is converging to the final solution, and the final error with respect to the real solution.

Report Questions 4 — cg (20%)

1. For the CPU version, copy the your CPU implementations of `axpy`, `dot`, and `smvp`. Without introducing any other changes, use the script `job_cpu.sh` to measure the time of the CPU version of the code.

2. Use the script `job_profile_cpu.sh` to profile the CPU code. Based on the profiler output, indicate the relative weights of the main three operations in the code.

3. For the GPU version of the code provide an optimal OpenACC implementation. Use the GPU kernels developed in the previuos parts, and use OpenACC to parallelize any remaining kernel in the CG algorithm. Make sure that you minimize data transfers between CPU and GPU. Explain how you parallelized the code, detailing the handling of the data tranfers. Use the script `job_gpu.sh` to compile and test the code.

4. Use the script `job_profile_gpu.sh` to profile the GPU code. Based on the profiler output, indicate the relative weights of the main three operations in the code, and of the memory transfers between CPU and GPU.

5. SpMV CUDA

In this exercise we are going to develop a CUDA version of the sparse matrix vector product of Exercise 3, starting from the CPU implementation.

The CUDA version will require:

- a CUDA kernel `spmv` implementing the `spmv` operation
- memory allocation in the GPU
- data transfer to and from the GPU
- definition of the number of thread blocks and the number of threads per block

Report Questions 5 — spmv cuda (20%)

1. Implement the CUDA version of `spmv` described above, associating threads to rows. **Do not assume that the number of rows is a multiple of the number of threads. The way of distributing rows over threads and blocks should work for a generic number of rows.** Explain your inplementation in the report.

2. Do threads use a coalesced access pattern to load the entries of **A** from global memory? Jutify your answer describing how threads access global memory.

3. **[Bonus for a 25% extra points.]** Write a CUDA kernel with an improved memory access pattern using shared memory. Threads in the block first collaborate to load necessary data into shared memory. Deliver an extra file `spmv_optim.cu`. The code should compile, and run faster than the previous CUDA version. Use the `job_optim.sh` to compile and launch the optimized code.