Júlia López and Tània Pazos
u214964, u185115

Lab 3: GPU

AXPY

1. Explain how you have parallelized the code, detailing the directives used and their clauses.

First and foremost, we defined the function axpy_cpu so that it performs the vector operation defined in the statement.

```
void axpy_cpu(int n, double alpha, double* x, double* y)
{
   for(int i = 0; i < n; i++) {
       y[i] = alpha * x[i] + y[i];
   }
}
```

The function axpy_gpu does exactly the same but we use the #pragma acc parallel loop which parallelizes the execution of the loop across multiple threads on the GPU. We added the present(x[0:n], y[0:n]) to ensure that arrays x and y are present inside the GPU before the loop starts. In other words, the “present” directive informs the compiler that both arrays are already allocated inside the GPU and can be directly used in the loop.

```
void axpy_gpu(int n, double alpha, double* x, double* y)
{
   #pragma acc parallel loop present(x[0:n], y[0:n])
   for(int i = 0; i < n; i++) {
       y[i] = alpha * x[i] + y[i];
   }
}
```

Then, in the main loop we added #pragma acc enter data before calling the axpy_gpu function. We also added the copyin(x[0:vec_size], y_gpu[0:vec_size]). The enter data directive is used to allocate memory for the arrays x and y_gpu on the GPU and initialize them with the values from the same arrays present in the CPU. On the other hand, the copyin clause ensures that the arrays x and y_gpu are inside the GPU before the GPU computation starts by copying data from the CPU to the GPU.

Similarly, after calling the function axpy_gpu, we used the #pragma acc exit data copyout(x[0:vec_size], y_gpu[0:vec_size]). The directive exit data deallocated memory for the arrays x and y_gpu on the GPU. Then, the copyout clause moves the data from the GPU

back to the CPU so that one can retrieve the updated values of the arrays x and y_gpu from the host (CPU).

```
#pragma acc enter data copyin(x[0:vec_size], y_gpu[0:vec_size])
   // GPU computation
   time_start = omp_get_wtime();

   for (int i = 0; i < 100; i++)
       axpy_gpu(vec_size, alpha, x, y_gpu);

   time_end = omp_get_wtime();
   time_gpu = time_end - time_start;
   #pragma acc exit data copyout(x[0:vec_size], y_gpu[0:vec_size])
```

Finally, we implemented the definition of the speedup to be able to evaluate the acceleration provided by the GPU.

```
double speed_up = time_cpu / time_gpu;
```

After executing the code, we obtained the following output.

```
nvc -acc=gpu -Minfo=accel  -o axpy.x axpy.c
axpy comparison cpu vs gpu error: 0.000000e+00, size 1000000
CPU Time: 0.063905 - GPU Time: 0.007513 - speed-up = 8.505871
```

Figure 1: Results of axpy.c with n=1000000.

As can be seen in Figure 1, both the sequential version running in the CPU and the OpenACC version running in the GPU give the same results, as the difference between their results is 0. Furthermore,the GPU code runs significantly faster than the CPU code, resulting in a speedup of approximately 8.5. In other words, the GPU-accelerated version executed 8.5 times faster than the CPU version.

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?

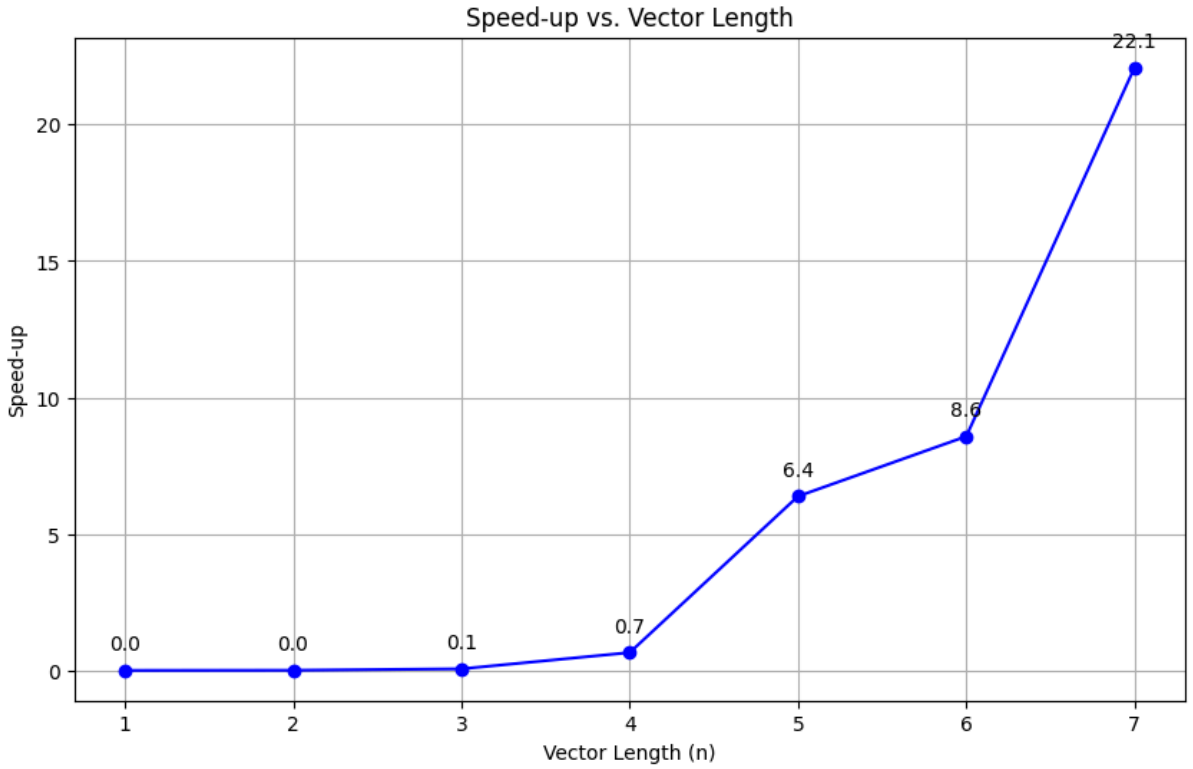

Figure 2: Speed-up plot for n= 10^1, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7.

This plot represents the performance advantage of the GPU as the CPU increases along with the vector length (n). In small values of n it can be seen that the speed-up is nearly 0 which indicates that the GPU does not provide computational benefits, in fact it most likely causes extra overhead. However, for larger vector lengths the speed-up gradually increases which proves that the acceleration that the GPU provides becomes highly beneficial, that is, that it outweighs the computational costs that come with its implementation.
To sum up what we can conclude from the plot is that the vector size is very important when it comes to determining whether when using the GPU we will obtain beneficial results or not.

DOT

1. Explain how you have parallelized the code, detailing the directives used and their clauses.

First of all, we defined the function dot_product_cpu so that it performs the vector product operation defined in the statement.

```
double dot_product_cpu(int n, double* x, double* y)
{
   double sum = 0.0;
   for (int i = 0; i < n; i++) {
       sum += x[i] * y[i];
   }
   return sum;
}
```

The function dot_product_gpu performs the same operation but we use #pragma acc parallel loop to parallelize the loop across all available GPU threads. Moreover, we add the reduction(+:sum) clause to indicate that each thread will have its own private copy of sum and at the end all the partial sums will be reduced into a single value. Also the present(x[0:n], y[0:n]) clause is needed to ensure that x and y variables are present and lastly the copy(sum) clause so that the sum variable is copied back to the CPU.

```
double dot_product_gpu(int n, double* x, double* y)
{
   double sum = 0.0;
   #pragma acc parallel loop reduction(+:sum) present(x[0:n], y[0:n]) copy(sum)
   for (int i = 0; i < n; i++) {
       sum += x[i] * y[i];
   }
   return sum;
}
```

As we did in the previous exercise we managed the data transfers in the main function before the dot_product_gpu function is executed. We specify by adding #pragma acc enter data copyin(x[0:vec_size], y[0:vec_size]) that x and y should be copied from the host (CPU) to the device (GPU) initialized with the same value, and after the block of code is completed, we added #pragma acc exit data delete(x[0:vec_size], y[0:vec_size]) to copy back variables x and y with their current value to the CPU.

```
#pragma acc enter data copyin(x[0:vec_size], y[0:vec_size])
   time_start = omp_get_wtime();
```

```
for(int i = 0; i < 100; i++)
    dot_gpu = dot_product_gpu(vec_size, x, y);

time_end = omp_get_wtime();
time_gpu = time_end - time_start;
#pragma acc exit data delete(x[0:vec_size], y[0:vec_size])
```

Again, we implemented the definition of the speedup to be able to evaluate the acceleration provided by the GPU.

```
double speed_up = time_cpu / time_gpu;
```

After executing the code this is the obtained results:

```
nvc -acc=gpu -Minfo=accel  -o dot.x dot.c
dot product comparison cpu vs gpu -2.521539e-12, size 1000000
CPU Time: 0.143534 - GPU Time: 0.008707 - Speed up: 16.484830
```

Figure 3: Results of dot.c with n=1000000.

As we can see in Figure 3, the comparison between the result obtained in the GPU computation and in the CPU computation is the same since we can consider the error $-2.521539 \times 10^{-12}$ negligible. It should be noted that the GPU version of the computation runs around 17 times faster than the CPU version.

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?

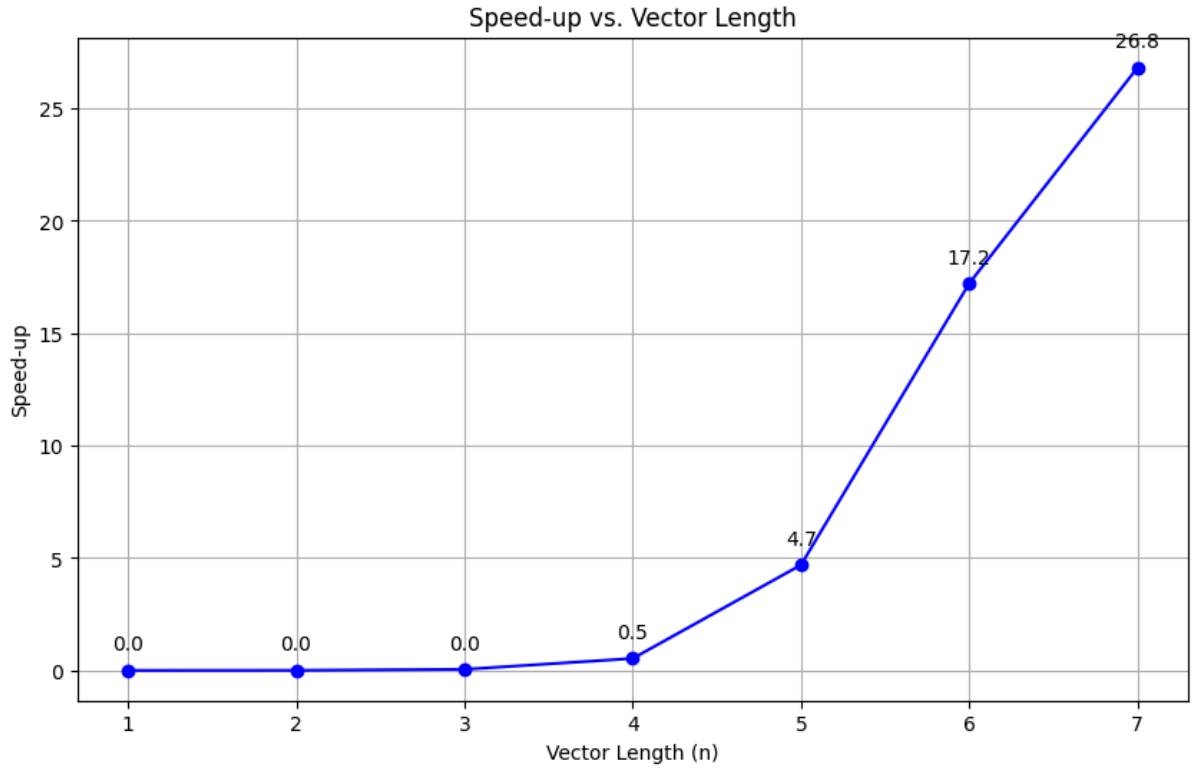

Figure 4: Speed-up plot for n= 10^1, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7.

As the plot in the previous exercise, it represents the performance advantage of the GPU as the CPU increases along with the vector length (n). Again, we can see that for small values of n the speed-up is nearly 0, that is that for 10^1, 10^2 and 10^3 the GPU does not affect positively in the computation time. Nevertheless for sizes of n from 10^4 to 10^7, the speed-up increases exponentially which is a sign that the GPU implication provides computational benefits time related.

SpMV

1. How many floating point operations are needed for a **dense** matrix-vector product with an m x 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.

On the one hand, for the dense matrix case, that is, that there are no zero values (r = n), each element of the resulting vector y will need n multiplications (one for each element of a row of the matrix) and then n-1 sums to sum up all the products. Thus, the total number of floating point operations per element of y is n + (n - 1). Since y will have m components, the resulting number of flops will be m x (n + n - 1) = 2nm - m.

On the other hand, for a sparse matrix where most elements are zero we will have the same case as for the dense one only that since we will have r non zero elements, we will need to perform only r multiplications and r-1 sums. Hence, the resulting number of flops will be m x (r + r - 1) = 2nr - n.

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.

Firstly, the sequential version implemented in the CPU performs a simple sparse matrix vector multiplication. The first loop iterates over each row of the sparse matrix, whereas the second loop iterates over the -up to r- non-zero elements of each row. For a non-zero element, its column index and its value is retrieved from the cols and vals arrays. To compute the index of the non-zero element inside the cols array we use the expression i * r + j. This is the same expression that we will use to compute the index inside the vals array.

```
void spmv_cpu(int m, int r, double* vals, int* cols, double* x, double* y)
{
   for(int i = 0; i < m; i++) {
       double sum = 0.0;
       for(int j = 0; j < r; j++) {
          int col = cols[i * r + j];
          double val = vals[i * r + j];

          sum += val * x[col];
       }
       y[i] = sum;
```

```
    }
}
```

In order to parallelize the loop using OpenACC inside the GPU, we first add the directive #pragma acc parallel loop to distribute iterations among multiple threads. We add the clause present(vals[0:m*r], cols [0:m*r], x[0:m], y[0:m]) to indicate that the arrays vals, cols, x and y are already available in the GPU. We then initialize each row to 0 and add the directive #pragma acc loop seq to make sure that the inner loop is executed sequentially. This allows each thread to avoid race conditions in the update of the sum variable. We add the clause reduction (+:sum) to indicate that each thread will compute a private copy of sum and then add the values of the private copies at the end of the loop.

```
void spmv_gpu(int m, int r, double* vals, int* cols, double* x, double* y)
{
    #pragma acc parallel loop present(vals[0:m*r], cols[0:m*r], x[0:m], y[0:m])
    for(int i = 0; i < m; i++) {
        double sum = 0.0;  // Ensure each thread starts with y[i] = 0
        #pragma acc loop seq reduction (+:sum)
        for(int j = 0; j < r; j++) {
            int col = cols[i * r + j];
            double val = vals[i * r + j];

            sum += val * x[col];
        }
        y[i] = sum;
    }
}
```

Then, in the main function we have added #pragma acc data copyin(Avals[0:vec_size*ROWSIZE], Acols[0:vec_size*ROWSIZE], x[0:vec_size], y_gpu[0:vec_size]) copyout(y_gpu[0:vec_size]) directive before the computation of spmv_gpu function. This ensures that Avals, Acols, x and y arrays are inside the GPU by copying the data from the CPU. On the other hand the copyout(y_gpu[0:vec_size]) moves the data from the GPU back to the CPU after the computations.

```
 #pragma acc data copyin(Avals[0:vec_size*ROWSIZE], Acols[0:vec_size*ROWSIZE],
x[0:vec_size], y_gpu[0:vec_size]) copyout(y_gpu[0:vec_size])
    {
    time_start = omp_get_wtime();
    for(int i = 0; i < 100; i++)
        spmv_gpu(vec_size, ROWSIZE, Avals, Acols, x, y_gpu);
```

```
  time_end = omp_get_wtime();
  time_gpu = time_end - time_start;
  }
```

After executing the code, we obtained the following output.

```
nvc -acc=gpu -Minfo=accel  -o spmv.x spmv.c
spmv comparison cpu vs gpu error: 0.000000e+00, size 1048576
CPU Time: 1.045500
GPU Time: 0.049982
```

Figure 3: Results of spmv.c.

As Figure 3 shows, the difference in the output vector y between the CPU version and the GPU version is null. Furthermore, the GPU time is significantly smaller than the CPU time, indicating that the parallelization has succeeded.

CG

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.

In the CPU version we simply copied the other CPU implementations of the previous exercises so no changes in the code were made.
When putting to test the code executing the job_cpu.sh bash file we obtain the desired error within the specified iterations.

```
Iteration 480, residual 5.377847e-14
cg error in cpu solution: 4.171721e-04, size 1048576
Time CPU: 9.703786
```

Figure 4: Results of cg_cpu.c executed by job_cpu.sh

The time as indicated in the output is around 10 seconds which is coherent given that the matrix is very large and computing we are running 500 iterations that involve entire matrix computations which are very costly.

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.

The profiler output given by the execution of the job_profile_cpu.sh to profile the CPU code is the following:

```
======== CPU profiling result (flat):
Time(%)      Time  Name
 65.62%  18.1366s  spmv_cpu
 18.53%  5.12185s  dot_product_cpu
  9.99%    2.761s  axpy_cpu
  5.32%  1.47053s  cg_cpu
  0.36%  100.04ms  fill_matrix
  0.14%  40.014ms  __fd_cos_1_avx2
  0.04%  10.004ms  main

======== Data collected at 100Hz frequency
```

Figure 5: Profile CPU code

This output indicates the percentage of time spent in different parts of the program. We can see that the sum of percentage of the total time of the first three main operations corresponds to 93.96%. It is worth mentioning that due to the profiling tools used in this

execution, the total running time goes up to around 27 seconds (as seen in the next figure) since these tools cause significant overhead.

```
Iteration 480, residual 5.377847e-14
cg error in cpu solution: 4.171721e-04, size 1048576
Time CPU: 27.448094
```

Figure 6: Results of cg_cpu.c executed by job_profile_cpu.sh

Analyzing the profiling result, we see that in the first place, being the most time-consuming operation, is the smpv_cpu function taking up 18.1366 seconds which represents 65.62% of the total computation time. This was expected since the sparse matrix-vector multiplication involves nested loops of size N where we perform dot products with the corresponding vector elements. Since we are working with a relatively large N (1024), it results in a matrix with over a million elements. Handling this size of data is very demanding on memory bandwidth and processing power. The memory access of a sparse matrix is also very time consuming since it involves indirect memory accesses.

In second place taking up 18.53% of the total computation time is the dot_product_cpu function which takes 5.12185 seconds. Although it is not as costly as the sparse matrix-vector multiplication, this function involves N^2 multiplications and N^2-1 additions. Even though we are dealing with a lot of operations, the fact that we need to access and process contiguous memory spaces implies that it is less time consuming than the previous function.

Lastly, the computation of the axpy_cpu function corresponds to 9.99% of the total time, that is, 2.761 seconds. The computations of this function consist of N^2 multiplications and N^2 additions. Similarly to the dot_product_cpu function, the function accesses contiguous memory addresses which do not take up as much time as indirect memory accesses. However, it is still a significant part of the computation due to the large number of operations performed.

3. For the GPU version of the code provide an optimal OpenACC implementation. Use the GPU kernels developed in the previous 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 transfer. Use the script job_gpu.sh to compile and test the code.

First of all, this code reuses spmv_cpu, spmv_gpu, axpy_gpu and dot_product_gpu which, those with GPU use we have already explained their parallelization and data transfers management previously.

Moving on to the new part of code, we firstly encounter the create_solution_and_rhs function. This function, that we also had in the CPU version, initializes the solution vector xsol and computes the right-hand side vector rhs. To do so we use the spmv_cpu function. To correctly manage data transfers we have added #pragma acc data copyin(xsol[0:vecsize], rhs[0:vecsize]) . The copyin clause indicates that we copy the xsol and rhs from the host to the device before computations since they have already been initialized in the CPU (main function).

```
void create_solution_and_rhs(int vecsize, double* Avals, int* Acols, double* xsol,
double* rhs)
{
   #pragma acc data copyin(xsol[0:vecsize], rhs[0:vecsize])
   {
       for(int i = 0; i < vecsize; i++)
           xsol[i] = sin(i*0.1) + cos(i*0.01);

       spmv_cpu(vecsize, ROWSIZE, Avals, Acols, xsol, rhs);

   }
}
```

Moving on to the cg_cpu function. This function, as expected, performs the Conjugate Gradient method on the GPU. First of all we add the clause #pragma acc data copyin(Avals[0:vecsize * ROWSIZE], Acols[0:vecsize * ROWSIZE], rhs[0:vecsize], x[0:vecsize]) which indicates the allocation of memory on the GPU for these variables as well as the copy of their initialization values from the CPU. On the other hand, we also add the create(Ax[0:vecsize], r0[0:vecsize], p0[0:vecsize]) clause that specifies the allocation of memory in the GPU for this variables which have not been initialized in the CPU.

For the parallelization, we have parallelized three loops: the initialization of r0 (the residual vector r0 = b - Ax) and p0 (search direction vector) variables and the final value for the latter. We have added #pragma acc parallel loop in before the three for loops to ensure parallel computation.

```
#pragma acc data copyin(Avals[0:vec_size * ROWSIZE], Acols[0:vec_size * ROWSIZE],
```

```
rhs[0:vec_size], x[0:vec_size]) create(Ax[0:vec_size], r0[0:vec_size],
p0[0:vec_size])
   {
       #pragma acc parallel loop
       for(int i = 0; i < vec_size; i++)
       {
           r0[i] = rhs[i];
       }
```

```
       #pragma acc parallel loop
       for(int i = 0; i < vec_size; i++)
       {
           p0[i] = r0[i];
       }
```

```
           #pragma acc parallel loop
           for(int i = 0; i < vec_size; i++)
               p0[i] = r0[i] + beta*p0[i];
```

Finally for the main function before the cg_gpu call we manage the data transfers by adding #pragma acc data copyin(Avals[0:ROWSIZE * vec_size], Acols[0:ROWSIZE * vec_size], rhs[0:vec_size], x_gpu[0:vec_size], x_sol[0:vec_size]) which again copies all the necessary and already initialized variables from the host to the device, and we add copyout(x_gpu[0:vec_size]) so that this array is copied back to the host after execution.

```
#pragma acc data copyin(Avals[0:ROWSIZE * vec_size], Acols[0:ROWSIZE * vec_size],
rhs[0:vec_size], x_gpu[0:vec_size], x_sol[0:vec_size]) copyout(x_gpu[0:vec_size])
   {
       time_start = omp_get_wtime();

       cg_gpu(vec_size, Avals, Acols, rhs, x_gpu);

       time_gpu = omp_get_wtime() - time_start;
```

Lastly we parallelized the error calculation in the main function adding #pragma acc parallel loop reduction(+:norm2) which indicates that each thread will have its own private copy of norm2 and at the end all the partial norm2 variables will be reduced into a single value. We have to ensure that this final norm2 value is copied back to the CPU hence we put copy(norm2) to ensure that.

```
double norm2 = 0.0;
       #pragma acc parallel loop reduction(+:norm2) copy(norm2)
```

```
    for(int i = 0; i < vec_size; i++)
        norm2 += (x_gpu[i] - x_sol[i])*(x_gpu[i] - x_sol[i]);

    printf("cg error in gpu solution: %e, size %d\n", sqrt(norm2), vec_size);
```

After implementing this data transfer handling and parallel processing, we execute the job_gpu.sh bash file and obtain the following result:

```
Iteration 480, residual 5.377847e-14
cg error in gpu solution: 4.171721e-04, size 1048576
Time GPU: 0.473107
```

Figure 7: Results of cg_gpu.c executed by job_gpu.sh

We obtain the desired result with a significant time decrease with respect to the CPU execution. We can observe that this version is not more accurate, however it is much faster in execution. This GPU-accelerated code is around 20.6 times faster than the CPU 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.

The profiler output given by the execution of the job_profile_gpu.sh to profile the GPU code is the following:

```
==834108== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   60.61%  2.56591s       501  5.1216ms  5.0401ms  6.2908ms  spmv_gpu_26_gpu
                   17.86%  756.17ms      1500  504.11us  494.39us  621.34us  dot_product_gpu_52_gpu
                   13.29%  562.76ms      1001  562.20us  552.60us  691.61us  axpy_gpu_43_gpu
```

Figure 8: Profile of GPU

We can see that the results are fairly similar to the ones of the CPU. The order of weights of the main three operations is the same and the time percentage is fairly similar. Due to the GPU-acceleration, the times are very different with respect to the CPU. The spmv_gpu function takes 60.61% of the total time that corresponds to 2.56 seconds, the dot_product_gpu takes 17.86% of the time, 756.17 milliseconds and lastly axpy_gpu takes 13.29% of the time that are 562.76 milliseconds. This sums up to 91.76% of the total execution time.

As mentioned before, due to the overhead of the profiling tools the sum of times add up to a different final value:

```
Iteration 480, residual 5.377847e-14
cg error in gpu solution: 4.171721e-04, size 1048576
Time GPU: 4.319333
```

Figure 9: Results of cg_gpu.c executed by job_profile_gpu.sh

For the memory transfers, we can see the following results:

```
0.30%  12.558ms        12  1.0465ms  362.99us  1.4361ms  [CUDA memcpy HtoD]
0.05%  2.1718ms      1502  1.4450us     960ns  639.13us  [CUDA memcpy DtoH]
```

Figure 10: Weights of memory transfers of GPU

First of all, it should be noted that memory transfers contribute relatively little to the overall GPU time.
The first statistic, CUDA memcpy HtoD, indicates the data transfers from host to device, that is, from CPU to GPU. It takes 0.30% of the total time which corresponds to 12.558 milliseconds. The second statistic corresponds to the data transfers from the GPU to the CPU which take 0.05% of the total time with 2.1718 milliseconds.

It is worth mentioning the difference between the weight of the data transfers from host to device and those from the device to the host. The design of our program ensures that the majority of the data once copied from the CPU to the GPU stays on the GPU during the computationally intensive parts, so that redundant data transfers to the CPU are avoided. This results in a higher impact of the host to device data transfers than device to host which we perform when strictly necessary.

SpMV CUDA

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 implementation in the report.

Firstly, the CUDA kernel function cuspmv performs sparse matrix-vector multiplication on the GPU. To compute the global row index for each thread, we use the formula blockIdx.x * blockDim.x + threadIdx.x. What is more, an if condition ensures that the computed row index does not exceed the number of rows.

```
	int row = blockIdx.x * blockDim.x + threadIdx.x;


	if (row < m) {
		double sum = 0.0;
	for (int j=0; j<r; j++) {
	    int col = dcols[row * r + j];
	    double val = dvals[row * r + j];
	    sum += val * dx[col];
	}
	dy[row] = sum;
```

Similarly, the function spmv_cpu operates exactly as the CUDA kernel but it runs on the CPU. Of course, there is no need to compute the global row index as rows are processed sequentially by a single thread.

```
for (int i=0; i<m; i++) {
	double sum = 0.0;
	for (int j=0; j<r; j++) {
	    int col = cols[i * r + j];
	    double val = vals[i * r + j];
	    sum += val * x[col];
	}
	y[i] = sum;
   }
```

Finally, the main function allocates memory for the input vector x, the output vector y_cpu and y_gpu -that store the result of the CPU computation and the GPU computation respectively-, the matrix Avals -that holds the non-zero values of the sparse matrix- and the matrix Acols -holding the column indices of the non-zero values of the matrix. Then, both the input vector x and the sparse matrix are filled.
Memory is then allocated on the GPU to be able to perform the matrix-vector multiplication.

We declare four pointers dx, dy_gpu, dAvals and dAcols, that will point to memory locations on the GPU. Then, we call cudaMalloc to allocate the specified amount of memory on the GPU and update each pointer to point to the corresponding allocated memory. We should bear in mind that both dx and dy_gpu are vectors of vec_size doubles, whereas dAvals and dAcols are matrices of ROWSIZE*vec_size (maximum number of non-zero elements*number of rows) doubles and integers, respectively.

```
   double* dx;
   double* dy_gpu;
   double* dAvals;
   int*    dAcols;

   cudaMalloc(&dx, vec_size * sizeof(double));
   cudaMalloc(&dy_gpu, vec_size * sizeof(double));
   cudaMalloc(&dAvals, ROWSIZE * vec_size * sizeof(double));
   cudaMalloc(&dAcols, ROWSIZE * vec_size * sizeof(int));
```

Before proceeding with the computation on the GPU, the necessary data must be copied from the CPU to the GPU using cudaMemcpy. Note that at this moment there is no need to copy the vector dy_gpu, since it will hold the results of the computation performed by the GPU kernel. The last parameter of the cudaMemcpy indicates that the direction of the data transfer occurs from host (CPU) to device (GPU).

```
   cudaMemcpy(dx, x, vec_size * sizeof(double), cudaMemcpyHostToDevice);
   cudaMemcpy(dAvals, Avals, ROWSIZE * vec_size * sizeof(double),
cudaMemcpyHostToDevice);
   cudaMemcpy(dAcols, Acols, ROWSIZE * vec_size * sizeof(int),
cudaMemcpyHostToDevice);
```

To ensure that all rows are managed by GPU threads, we compute the number of blocks using the following expression.

```
   int blocks = (vec_size + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
```

Instead of dividing directly the number of rows (vec_size) over the number of threads per block (THREADS_PER_BLOCK), we add THREADS_PER_BLOCK - 1 before division to handle the case where the number of rows is not a multiple of the number of threads. Indeed, adding THREADS_PER_BLOCK - 1 to vec_size rounds up the number of blocks needed and ensures that leftover rows that cannot create an entire block are processed.

We then record the start event for timing, launch the CUDA kernel 100 times inside a loop and record the top event.

```
   cudaEventRecord(start);
```

```
    for (int i=0; i<100; i++){
        cuspmv<<<blocks, THREADS_PER_BLOCK>>>(vec_size, ROWSIZE, dAvals, dAcols, dx,
dy_gpu);
    }
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time_gpu, start, stop);
```

Recall the first parameter between the <<<...>>> brackets specifies the number of blocks and the second parameter, the number of threads per block for the kernel launch. The parameters passed to the kernel function cuspmv are specified between parentheses: vec_size, ROWSIZE, dAvals, dAcols, dx and dy_gpu.

Once the GPU computation has finished, we must copy the result from the GPU back to the host using cudaMemcpy but, this time, setting the transfer direction to cudaMemcpyDeviceToHost.

```
    cudaMemcpy(y_gpu, dy_gpu, vec_size * sizeof(double), cudaMemcpyDeviceToHost);
```

Finally, we free the allocated memory on the GPU using cudaFree, compute the Euclidean norm of the difference between the results obtained from the CPU and GPU versions of the sparse matrix-vector multiplication, and deallocate the memory on the CPU with free.

```
    cudaFree(dx);
    cudaFree(dy_gpu);
    cudaFree(dAvals);
    cudaFree(dAcols);

    double norm2 = 0.0;
    for(int i = 0; i < vec_size; i++)
        norm2 += (y_cpu[i] - y_gpu[i])*(y_cpu[i] - y_gpu[i]);

    norm2 = sqrt(norm2);

    free(x);
    free(y_cpu);
    free(y_gpu);
    free(Acols);
    free(Avals);
```

Figure 11 compares the CPU and GPU implementations of the sparse matrix-vector multiplication. The difference between the CPU and GPU results is very small, indicating that both versions produce very similar results. Furthermore, the GPU implementation is much

faster than the CPU implementation, which demonstrates the benefits of parallelization using the GPU.

```
nvcc -O3  -o spmv.x spmv.cu
spmv comparison cpu vs gpu error: 6.352582e-14, size 1048576
CPU Time: 1.056544
GPU Time: 0.089937
```

Figure 11: CPU and GPU comparison regarding results and time to perform sparse matrix-vector multiplication.

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

In CUDA, threads within the same warp access consecutive memory locations in global memory transactions. When threads use a coalesced pattern to access memory, the memory controller is able to fetch contiguous memory addresses in a single transaction. As a result, the number of memory accesses is reduced, which maximizes memory bandwidth utilization.

In the code, we can see that the access pattern to dx[col] is not coalesced. Indeed, the value of col is dictated by dcols[row*r +j] which can result in non-contiguous indices. Consequently, the access pattern to dx[col] is irregular and not coalesced.

```
if (row < m) {
        double sum = 0.0;
        for (int j=0; j<r; j++) {
            int col = dcols[row * r + j];
            double val = dvals[row * r + j];
            sum += val * dx[col];
        }
        dy[row] = sum;
    }
```

3. 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.

```
nvcc -O3  -o spmv_optim.x spmv_optim.cu
spmv comparison cpu vs gpu error: 4.728198e+02, size 1048576
CPU Time: 1.061661
GPU Time: 0.007348
```

Figure 12: CPU and GPU comparison regarding results and time to perform optimized sparse matrix-vector multiplication.

We have tried the bonus implementation of this exercise. On the one hand, we have been able to reduce the GPU time significantly as shown in Figure 12. However, we could not get similar solutions for the CPU and the GPU hence the big comparison value. Nevertheless, we include the draft of the code in our submission.